UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

KEVIN MITCHELL SPILLER

ANÁLISE DE QUALIDADE DE STEERING BEHAVIORS E DESEMPENHO DA UNITY 5

CHAPECÓ
2018

KEVIN MITCHELL SPILLER

ANÁLISE DE QUALIDADE DE STEERING BEHAVIORS E DESEMPENHO DA UNITY 5

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Prof. Dr. Emilio Wuerges

CHAPECÓ
2018

PROGRAD/DBIB - Divisão de Bibliotecas

Spiller, Kevin Mitchell
Análise de Qualidade de Steering Behaviors e Desempenho da Unity 5/ Kevin Mitchell Spiller. -- 2018.
82 f.

Orientador: Emilio Wuerges.
Trabalho de conclusão de curso (graduação) - Universidade Federal da Fronteira Sul, Curso de Ciência da Computação , Chapecó, SC, 2018.

1. Steering Behaviors. 2. Técnicas de inteligência artificial no contexto de jogos. 3. Unity. 4. Personagens Autônomos. I. Wuerges, Emilio, orient. II. Universidade Federal da Fronteira Sul. III. Título.

Elaborada pelo sistema de Geração Automática de Ficha de Identificação da Obra pela UFFS com os dados fornecidos pelo(a) autor(a).

KEVIN MITCHELL SPILLER

ANÁLISE DE QUALIDADE DE STEERING BEHAVIORS E DESEMPENHO DA UNITY 5

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Prof. Dr. Emilio Wuerges

Este trabalho de conclusão de curso foi defendido e aprovado pela banca em: 05/07/18

BANCA EXAMINADORA:

Dr. Emilio Wuerges - UFFS

Dr. José Carlos Bins Filho - UFFS

Me. Adriano Sanick Padilha - UFFS

AGRADECIMENTOS

Agradeço primeiramente a minha família que sempre acreditou em mim e não mediu esforços para que eu chegasse onde estou.

Agradeço ao meu orientador que sempre dentro do possível me ajudou e guiou durante a realização deste trabalho.

Por fim agradeço também aos meus colegas e amigos que de alguma forma confiaram em mim.

RESUMO

A inteligência artificial é um campo fascinante e repleto de áreas praticáveis, uma dessas áreas é a área de jogos digitais. Nesta área é possível encontrar diversas técnicas de inteligência artificial, sendo uma delas os *steering behaviors*, comportamentos de direção que tem como objetivo ajudar personagens autônomos a mover-se em uma maneira mais realista, usando simples técnicas envolvendo vetores. Neste trabalho é abordado três diferentes técnicas de *steering behaviors* que foram implementados e modificados no motor de jogo *Unity* 5, sendo eles *Arrival, Collision Avoidance* e *Wall Avoidance*. Este trabalho tem como objetivo analisar a qualidade destes *steering behaviors* como também analisar o desempenho deste motor de jogo. Para alcançar estes objetivos um número de métricas de qualidade e desempenho foram coletadas e analisadas após a realização de simulações em nove cenários adaptados de um benchmark para a *Unity* 5 que podem representar cenários do mundo real. Ao final do trabalho foi possível comprovar diferenças notáveis nas métricas de qualidade devido aos diferentes cenários, número de agentes e os comportamentos como também comprovar que a taxa de quadros por segundo é mais sensível ao número de agentes e seus comportamentos enquanto foi somente possível comprovar parcialmente que o uso total de memória do sistema é sensível referente ao número de objetos de jogo(agentes, obstáculos, etc), número de agentes e seus comportamentos.

Palavras-chave: Steering behaviors. Técnicas de inteligência artificial no contexto de jogos. Unity. Personagens Autônomos.

ABSTRACT

Artificial intelligence is a fascinating field and filled with practicable areas, one of these areas is the area of digital games. In this area it is possible to find several techniques of artificial intelligence, being one of them the steering behaviors, directional behaviors that aims to help autonomous characters to move in a more realistic way, using simple techniques involving vectors. In this paper we discuss three different steering behavior techniques that have been implemented and modified in the Unity 5 game engine, such as Arrival, Collision Avoidance and Wall Avoidance. This work aims to analyze the quality of these steering behaviors as well as analyze the performance of this game engine. To achieve these objectives a number of quality and performance metrics were collected and analyzed after simulations were performed in nine scenarios adapted from a benchmark for Unity 5 that can represent real world scenarios. At the end of the work it was possible to verify notable differences in quality metrics due to the different scenarios, number of agents and behaviors, as well as to verify that the frame rate per second is more sensitive to the number of agents and their behaviors while it was only partially possible to prove that the total memory usage of the system is sensitive regarding the number of game objects (agents, obstacles, etc.), number of agents and their behaviors.

Keywords: Steering behaviors. Artificial intelligence techniques in the context of games. Unity 5. Autonomous Characters.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE APÊNDICES

LISTA DE ABREVIATURAS E SIGLAS

NPC	*Non-playable character*
FPS	*Frames Per Second*
CPU	*Central Process Unit*
GPU	*Graphics Processing Unit*
TSMU	*Total System Memory Usage*

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização do Problema

A indústria de vídeo games cresce mais a cada ano[13] e já superou a indústria musical e cinematográfica. Vídeo games, hoje vistos como uma forma de entretenimento, trabalho e competição, estão cada vez mais deixando de serem vistos como algo para apenas passar o tempo. Grandes empresas de jogos, tais como Rockstar Games, Bethesda Software e Valve, entre outras presam por um sistema de IA robusto em seus jogos. Porém não somente empresas grandes como também empresas pequenas, chamadas de *Indie*, presam por isso.

Além disto com a disponibilidade de *game engines* gratuitas como a Unity 5 e outras, é possível que pessoas sem vínculo algum a uma empresa, também criem jogos e os disponibilizem gratuitamente ou não. Sem dúvida é preciso saber programação, arte visual, *level design*, áudio *design*, inteligência artificial, animação, criação de personagens, entre outras áreas que englobam um video game. Por isso, quanto mais pessoas você tiver em sua equipe de desenvolvimento melhor[8].

A qualidade de IA é uma característica de alto nível para os fãs de jogos no ato de fazerem suas decisões de compra e é uma área com potencial incrível de aumentar a imersão e diversão dos jogadores[14]. Desenvolvidos pelo cientista da computação Craig Reynolds no final de 1980, *steering behaviors* permitiriam elementos individuais a navegar seus ambientes digitais em uma maneira realista com estratégias para fugir, vagar, chegar, perseguir, evitar, etc[18].

Os *Steering behaviors* são técnicas de comportamentos de direção de inteligência artificial no contexto de jogos. *Steering behaviors* aplicados para personagens autônomos, visam a habilidade de navegar ao redor de seu mundo em uma maneira natural e improvisada[17].

Uma necessidade fundamental de quase todos agentes autônomos em mundos virtuais é a condução: a habilidade de um agente navegar para um destino/objetivo, através de um ambiente que inclua objetos estáticos, como prédios, e objetos dinâmicos, como outros personagens virtuais.

Steering é um problema desafiador para agentes autônomos. Na realidade, condução é um resultado de um processo complexo: Um agente faz decisões de condução baseados em informação sensorial, predições de movimentos de objetos dinâmicos, etiqueta social, experiência

pessoal, parâmetros específicos de situações, objetivos cognitivos e desejos[9].

Mesmo quando um agente cognitivamente decidiu seu destino, o problema de manobrar ao redor de vários obstáculos e outros agentes em um ambiente é extremamente complexo, deixando o agente com um grande número arbitrário de possíveis escolhas de condução[20].

Este conjunto rico de escolhas de *steering* reflete em um grande número de técnicas que podem ser implementadas e usadas em mundos virtuais. Algoritmos atuais são normalmente focados em um subconjunto de desafios de um problema, e frequentemente artigos tem somente espaço suficiente para mostrar suas novas características[20].

Portanto *steering behaviors* proporcionam um efeito mais natural ao personagem autônomo, fazendo com que seus movimentos sejam mais suaves ao invés de bruscos. São usadas forças simples, tais como vetores e não estratégias complexas envolvendo planejamento de caminho ou cálculos globais.

Apesar de serem simples de serem implementadas, essas técnicas são capazes de produzir padrões complexos de movimentação, sendo importante fazer a simulação deles na *Unity 5* e coletar seus dados para termos uma análise sobre o desempenho dos mesmos na *Unity 5* e descobrir se existe vantagem ou não usar em usá-la para *steering behaviors*.

1.2 Objetivos

1.2.1 Objetivo geral

O objetivo geral é avaliar a qualidade dos *steering behaviors* implementados no motor de jogo Unity 5 e avaliar o desempenho deste motor de jogo durante as simulações.

1.2.2 Objetivos específicos

- Abordar os diferentes tipos de *steering behaviors* que foram implementados por Anton Pantev[15] na *Unity 5*. Sendo eles *Arrival*, *Collision Avoidance*, *Wall Avoidance*.
- Modificar estes comportamentos de direção no motor de jogo *Unity* 5 em C# para rodar em simulações os cenários do benchmark *SteerBench* adaptados para a *Unity* 5.
- Aplicar métricas e casos de teste para avaliação de *steering behaviors* na *Unity* 5 e da performance da mesma, fazendo uma análise dos resultados, para que então possa ser levantado informações importantes sobre qualidades dos *steering behaviors* na *Unity* 5 e

a performance da *Unity* 5.

1.3 Justificativa

A importância deste trabalho justifica-se principalmente pelo crescimento dos jogos digitais no mercado atual, alavancando milhares de empregos e ajudando para a economia de todos envolvidos.

É importante também para desenvolvedores individuais que sonham um dia em trabalhar no mercado em grandes empresas, visto que muitas empresas de jogos requerem um currículo tendo feito/participado na criação de pelo menos um jogo e com *steering behaviors* é possível criar jogos com comportamentos muito próximos do mundo real.

Mesmo que este trabalho seja focado no contexto de inteligência artificial para jogos, grande parte dele pode ser usado como referência ou inspiração para outras simulações do mundo real que possam ser representados no mundo virtual. *Steering behaviors* podem ser usados para robótica, como por exemplo em[1] onde um robô tem a capacidade de andar por um campo, pegar uma bola e levar até o gol. Também podem ser usados para simulações de evacuações de locais em casos de emergência[25] e outros campos.

1.4 Estrutura do trabalho

Este trabalho está estruturado em 8 capítulos. No capítulo 1 foi apresentada a introdução, a contextualização do problema, os objetivos e a justificativa. No capítulo 2 foi apresentado o referencial teórico. No capítulo 3 foi descrito quais foram os trabalhos relacionados mais importantes e debatido a importância de cada. No capítulo 4 é apresentada a metodologia do trabalho. No capítulo 5 foi apresentado a implementação e explicação dos comportamentos de Anton Pantev juntamente com as modificações necessárias para rodar nos cenários do *SteerBench*. No capítulo 6 é descrito como foram feitas as simulações dos cenários de casos de teste. No capítulo 7 é apresentado e explicados os resultados obtidos. No capítulo 8 são realizadas as considerações finais. Após isto são apresentadas as referências utilizadas neste trabalho.

2 REFERENCIAL TEÓRICO

2.1 Personagens Autônomos

Um mundo virtual sem personagens autônomos é um ambiente vazio e "morto". Mesmo se houver interação com o ambiente, a sensação de ambientar um mundo sem vida ainda pode ser sentida. Personagens autônomos são personagens que possuem certo comportamento com o intuito de parecer o mais real e improvisado possível. Esta característica pode ser usada para descrever personagens tais como, pessoas, NPCs, animais, veículos e até mesmo ambientes. Seja para movimentos simples como: pular, andar, correr, atacar, etc ou mais complexos.

Personagens autônomos representam um personagem em uma história ou jogo e possuem alguma habilidade de improvisar suas ações. Personagens autônomos são um tipo de agente autônomo destinado a ser usado em animação por computador e mídia interativa, como jogos e realidade virtual[17]. Podem ser aplicados para jogos, animações para televisão, filmes e outras formas de interação.

Algumas habilidades adequadas para estes personagens (muitas vezes simulando seres humanos) incluem: uma aparência realista, a habilidade de produzir movimentos naturais e uma aptidão para raciocinar e agir de forma imprevisível[19]. É preciso considerar três componentes chaves sobre agentes autônomos[18]:

- Um agente autônomo tem uma habilidade limitada para perceber o ambiente: o poder de armazenar referências de outros objetos e portanto "perceber"seu ambiente;
- Um agente autônomo processa a informação de seu ambiente e calcula uma ação: a partir da informação processada do seu ambiente, o agente autônomo calcula uma ação, ou seja, uma força;
- Um agente autônomo não tem um líder: esta regra vale para ambos *steering behaviors* individuais ou pares e grupos ou combinações de *steering behaviors*. Agentes autônomos que exibem as propriedades de sistemas complexos - dinâmicas de grupo inteligente e estruturada não emergem a partir de um líder, mas de interações locais dos próprios elementos.

"Um agente autônomo é um sistema situado dentro de uma parte de um ambiente que sente esse ambiente e age sobre ele, ao longo do tempo, em busca de sua própria agenda e também para afetar o que sente no futuro[6]".

O movimento de um agente autônomo pode ser quebrado em três camadas:

- **Action Selection**: Essa é a parte do comportamento do agente responsável por escolher seus objetivos e decidir qual plano seguir. Podemos dizer que é a parte que diz "vá aqui"e "faça A, B, e depois C".

- **Steering**: Camada responsável por calcular a trajetória desejada necessária para satisfazer os objetivos e planos setados pela camada *action selection*. *Steering behaviors* são a implementação dessa camada.

- **Locomotion**: A camada inferior, locomoção, representa os aspectos mais mecânicos do movimento de um agente. É o *como* de viajar de A até B. Como exemplo, "Se você tivesse implementado as mecânicas de um camelo, um tanque, e um peixe-dourado e então dado um comando para eles viajarem ao norte, eles iriam usar processos mecânicos diferentes para criar movimento mesmo apesar de suas intenções(mover-se ao norte) serem idênticas[6]. Separando esta camada da camada *steering*, é possível utilizar com pouca modificação, os mesmos *steering behaviors* para tipos completamente diferentes de locomoção.

Reynolds faz uso de uma excelente analogia para descrever cada uma das três camadas em seu artigo *"Steering Behaviors for Autonomous Characters"*. Uma ótima exemplificação gráfica de sua analogia pode ser vista na figura 2.1 e presenciada em tempo real em determinadas ações em uma missão do aclamado jogo de sucesso da Rockstar Games: *Red Dead Redemption*. O objetivo da missão é levar o rebanho de um local ao outro.

Nesta missão o jogador controla o protagonista John enquanto ajuda sua amiga chefe da trilha Bonnie a levar o rebanho de gado a um certo local. Durante muitas vezes, algumas vacas fogem do rebanho, nestes momentos Bonnie pede para o jogador ir atrás delas e trazer-las de volta ao rebanho, comandando seu cavalo e possivelmente desviando de obstáculos durante o caminho. Neste momento, a chefe de trilha representa *action selection*: notando que o estado do mundo mudou(uma vaca fugiu do rebanho) e estabelecendo um objetivo(recuperar a vaca)[17].

O *steering level* é representado pelo jogador, o qual decompõe o objetivo em uma série de sub-objetivos simples(aproximar-se da vaca, evitar obstáculos, recuperar a vaca)[17]. Um

Figura 2.1: Movimento de um agente autônomo

sub-objetivo corresponde a um *steering behavior* para a equipe jogador-e-cavalo. Ao comandar o cavalo, o jogador usa vários comandos de controle, como comandos vocais, esporas e rédeas, dirigindo seu cavalo em direção a vaca. Esses comandos servem para dizer ao cavalo ir mais rápido, mais devagar, virar à esquerda, à direita e assim por diante.

A implementação do cavalo representa a camada *locomotion*. Recebendo os comandos de controle do jogador como entrada, o cavalo move-se na direção indicada. Este movimento é o resultado de uma interação complexa da percepção visual do cavalo, seu senso de equilíbrio, e seus músculos aplicando esforços para as juntas de seu esqueleto[17].

2.2 Steering Behaviors

Muitos tipos de comportamentos podem ser implementados a personagens autônomos, sendo um desses os *steering behaviors*. *Steering behaviors* foram desenvolvidos por Craig W. Reynolds e tem como objetivo simular comportamentos de direção visualmente agradáveis e naturais sem cálculos complexos em personagens autônomos.

Steering behaviors podem ser usados para jogos, robóticas, inteligência artificial, vida artificial e outros campos. Para jogos *Steering behaviors* podem ser usados para:

- Técnicas de IA de um jogo de corrida: veículos automatizados seguindo a rua ou navios que seguem um líder;
- Qualquer jogo que usa técnicas de IA de navegação: NPCs, personagens de jogos de

ação;

- Um jogo com necessidades de técnicas de comportamentos de IA: *seek*, *arrival*, *collision avoidance*, *wall avoidance*, etc.

Existem técnicas usadas para indivíduos e pares como também técnicas para comportamentos combinados e em grupo. Reynolds lista 12 comportamentos como *steering behaviors* para indivíduos e pares: Procurar, Fugir, Perseguir, Evitar, Vagar, Evitação de colisão, Evitação de paredes, Contenção, Seguimento de parede, Seguimento de caminho e Campo de fluxo seguinte.

Para comportamentos combinados e em grupo existem cinco: Seguir o caminho de multidão, Seguimento de líder, Evitação de colisão não alinhada, Filas e Andar em bando.

Steering behaviors são procedimentos reativos que tomam informações locais sobre o ambiente como entrada e produzem um vetor de direção desejado como saída. Eles realizam uma sub-tarefa específica, como seguir um corredor, evitar obstáculos ou interceptar um objeto em movimento, e podem ser combinados para formar um comportamento de nível mais alto[1].

Neste trabalho foi abordado os *steering behaviors* individuais e pares *Seek*, *Arrival*, *Collision Avoidance* e *Wall Avoidance*.

"A implementação de todas as forças envolvidas nos comportamentos de direção pode ser alcançada usando vetores matemáticos. Uma vez que essas forças irão influenciar a velocidade do personagem e posição, é uma boa abordagem usar vetores para representá-los também[2]". Neste trabalho os personagens autônomos foram representados como veículos pois representa melhor um simples modelo de locomoção, "...podendo abranger uma vasta gama de meios de transportes, desde dispositivos de rodas até cavalos, de aeronaves a submarinos para incluir a locomoção pelas próprias pernas de um personagem[17]."

2.2.1 *Seek*

Apesar deste comportamento ter sido implementado para ajudar em outros comportamentos, ele não foi testado separadamente nos casos de teste e métricas, mas é importante entendê-lo para a comportamento *arrival*. O objetivo deste *steering behavior* é agir para movimentar o personagem em direção a uma posição específica em um espaço global. Pode ser visto também como uma perseguição a um alvo estático. Este comportamento ajusta o personagem para que sua velocidade seja radicalmente alinhada em direção ao alvo[17].

A primeira coisa a ser considerada neste comportamento é calcular a *desired velocity*. Ela será a velocidade que o personagem precisa para alcançar a posição do alvo em um mundo ideal. Representando o vetor a partir do personagem até o alvo, escalado para ser o comprimento da máxima velocidade possível do personagem. O cálculo para a posição do alvo e do personagem é resultante da integração de Euler, resultando no cálculo em que posição = posição + velocidade.

A velocidade desejada é um vetor do personagem até o alvo, sendo o menor caminho entre o personagem e o alvo. Seu cálculo é resultante da subtração da posição do alvo menos a posição do personagem. O vetor de direção é a diferença entre esta "velocidade desejada"e a velocidade atual do personagem, como mostra na figura 2.2.

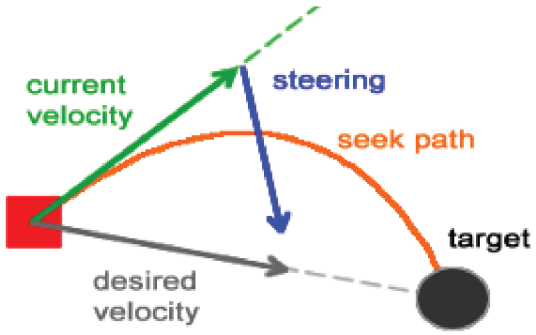

Figura 2.2: Seek
Imagem original retirada de : Understanding Steering Behaviors: Seek [5]

A *steering force* irá empurrar o personagem em direção ao alvo, assim criando o *seek path* conforme visto na figura 2.2. Caso o personagem continue a procurar, ele eventualmente atravessará o alvo e então virar-se e atravessá-lo novamente. Isso produz um movimento semelhante ao de uma mariposa zumbindo em torno de uma lâmpada[17]. Ao vermos o *steering behavior* arrival, esse exemplo ficará mais claro. *Seek* é muito útil para vários tipos de ações, como veremos nos próximos *steering behaviors* que fazem uso dele.

2.2.2 *Arrival*

O comportamento *Seek* faz com que o personagem mova-se em direção ao alvo porém ao chegar nele, ele passa pelo mesmo e volta a procurá-lo, fazendo com que ele fique indo e voltando ao alvo. O comportamento Arrival previne que o personagem atravesse o alvo, fazendo com que o personagem diminua a velocidade à medida que se aproxima do destino, eventualmente parando no alvo[2], como visto na Figura 2.3.

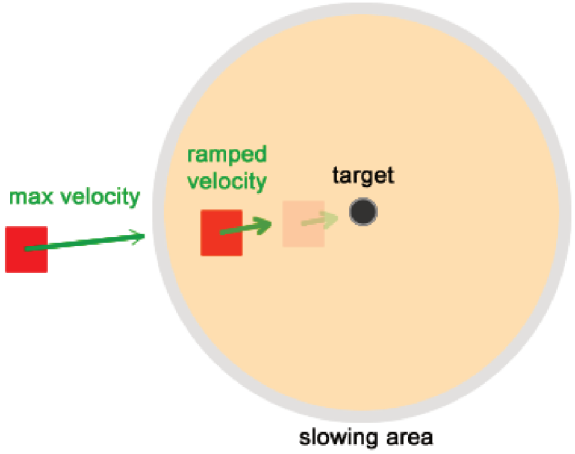

Figura 2.3: Arrival - Área de desaceleração
Imagem original retirada de : Understanding Steering Behaviors: Flee and Arrival [4]

Seu comportamento é composto por duas etapas:

- **Longe do alvo**: Quando o personagem está longe do alvo e atua da mesma maneira que o *Seek*, ou seja, buscando o alvo estático;
- **Perto do alvo**: Esta etapa ocorre quando o personagem está perto do alvo, dentro da área de "desaceleração". Ao entrar nessa área a sua velocidade é reduzida linearmente para zero. Para que isso seja possível, é necessário adicionar uma nova *steering force*, chamada de *Arrival force* ao vetor de velocidade do personagem.

Para garantir que o personagem irá gradualmente desacelerar antes de parar, sua velocidade não deve tornar-se zero imediatamente. O cálculo do processo gradual de desaceleração é baseado no raio da área de desaceleração e na distância entre o personagem e o alvo. Portanto, se a distância é maior que o raio de desaceleração, isto significa que o personagem está longe do alvo e sua velocidade deve continuar a mesma, caso contrário sua velocidade deve ser reduzida[2]. Na figura 2.4 podemos ver claramente o processo gradual de desaceleração, onde dv significa a velocidade desejada e sua multiplicação a cada etapa de aproximação do personagem. Essa variação linear fará com que a velocidade seja suavemente reduzida.

Exemplos do mundo real desse comportamento incluem um jogador de *baseball* correndo, e depois parando na base; ou um automóvel dirigindo em direção a um cruzamento e vindo a parar em um semáforo[17].

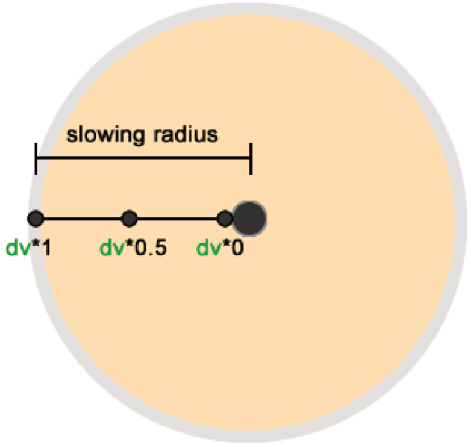

Figura 2.4: Arrival - Distância do raio
Imagem original retirada de : Understanding Steering Behaviors: Flee and Arrival [4]

2.2.3 *Collision Avoidance*

Jogadores não-jogáveis decentes geralmente necessitam da habilidade de desviar de obstáculos ou agentes, para um entendimento mais claro, foi escolhido chamar um agente também de obstáculo, este sendo o objetivo deste comportamento. A ideia básica por trás dele é gerar uma *steering force* para desviar de obstáculos toda vez que um esteja próximo o suficiente para bloquear a passagem. Mesmo que o ambiente tenha vários obstáculos, este comportamento irá usar um de cada vez para calcular a força de evasão.

Para que o personagem desvie somente do obstáculo mais próximo dele, somente obstáculos a sua frente são analisados. O mais próximo é dito como o mais ameaçador, assim sendo o primeiro a ser selecionado para avaliação. Como resultado o personagem é capaz de desviar todos obstáculos na área, passando de um para outro graciosamente e perfeitamente[3].

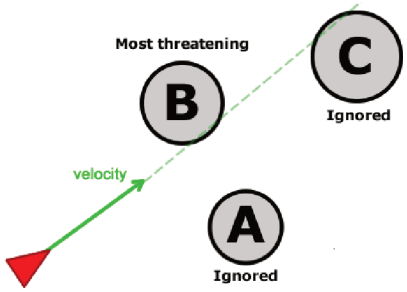

Figura 2.5: Collision Avoidance
Imagem original retirada de : Understanding Steering Behaviors: Collision Avoidance [3]

Algo importante a ser notado é que este comportamento não é um algoritmo de busca de caminho, ele faz com que personagens movam-se pelo ambiente, evitando obstáculos, even-

tualmente encontrando uma rota para passar pelos obstáculos[3].

O primeiro passo para evitar obstáculos no ambiente é perceber eles. Porém não é preciso preocupar-se diretamente com todos os obstáculos que podem ou não apresentarem uma ameaça ao personagem, os únicos obstáculos que o personagem precisa preocupar-se são aqueles que estão em sua frente e estão diretamente bloqueando sua rota atual.

Para isso um novo vetor chamado de *ahead* ou à frente, será usado para "perceber"obstáculos a sua frente, sendo uma cópia do vetor velocidade porém com um tamanho diferente conforme a figura 2.6.

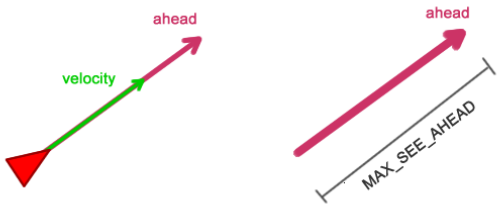

Figura 2.6: Collision Avoidance - Vetor *ahead*
O vetor *ahead* é a linha de visão do personagem. Imagem retirada de : Understanding Steering Behaviors: Collision Avoidance [3]

Vendo a sua frente. O vetor *ahead* ao ser ajustado somando a posição mais velocidade e multiplicado por um valor que chamaremos de MAX_SEE_AHEAD, que representa o máximo que pode ser visto a sua frente, irá definir o quão longe o personagem irá ver. Quanto maior for esse MAX_SEE_AHEAD, mais cedo o personagem irá começar a agir para desviar de um obstáculo, pois irá percebê-lo como uma ameaça mesmo que este esteja longe.

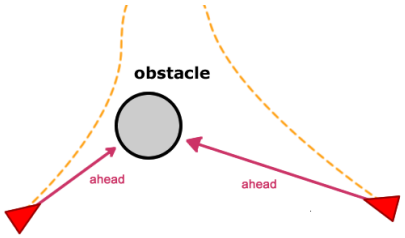

Figura 2.7: Collision Avoidance - Comprimento do vetor *ahead*
Imagem retirada de : Understanding Steering Behaviors: Collision Avoidance [3]

Verificando por colisão. Para que a verificação por colisão seja feita, todo obstáculo(ou sua caixa delimitante) deve ser descrita como uma forma geométrica. Usando-se de uma esfera, obtém-se um resultado melhor, então todo obstáculo no ambiente será descrito como tal[3]. O vetor *ahead* é usado para produzir outro vetor com metade de seu comprimento.

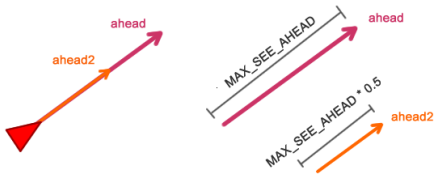

Figura 2.8: Collision Avoidance
Mesma direção, metade do comprimento. Imagem retirada de : Understanding Steering Behaviors: Collision Avoidance [3]

A verificação de colisão realizada testa se um dos dois vetores estão dentro da esfera obstáculo. Para isso, a distância entre o final da esfera e o centro dela é comparada. Se a distância é menor ou igual ao raio da esfera, então o vetor está dentro da esfera e uma colisão foi encontrada[3], conforme a figura 2.9.

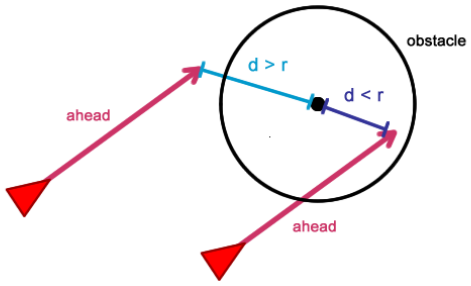

Figura 2.9: Collision Avoidance - O vetor *ahead* está interceptando o obstáculo se d < r.
Imagem retirada de : Understanding Steering Behaviors: Collision Avoidance [3]

Caso qualquer um dos dois vetores *ahead* estejam dentro da esfera do obstáculo, então este obstáculo está bloqueando o caminho. A distância *Euclidiana* entre dois pontos pode ser usada. Se mais de um obstáculo está bloqueando o caminho, então o mais próximo será selecionado para o cálculo.

Calculando a força de evasão. A força de evasão deve ser adicionada ao vetor de velocidade do personagem. Todas as *steering forces* podem ser combinadas de forma a tornarem-se apenas uma, produzindo uma força que representa todos comportamentos ativos em um personagem[3].

Dependendo do ângulo da força de evasão e direção, ela não interromperá outras *steering forces*, como *seek* e outras. Como todos *steering behaviors* são pré-calculados a cada *frame* do jogo, a força de evasão será mantida ativa enquanto o obstáculo estiver bloqueando o caminho[3].

Assim que o obstáculo não estiver interceptando a linha do vetor *ahead*, a força de

evasão irá tornar-se nula ou será recalculada para evitar um novo obstáculo ameaçador.

Melhorando a precisão. Existem dois problemas que precisam ser tratados ainda para que o comportamento fique o mais realista possível. O primeiro é quando os vetores *ahead* estão fora da esfera do obstáculo, mas o personagem está muito perto ou dentro do obstáculo.

Quando isso ocorre, o personagem irá tocar ou "entrar"no obstáculo, escapando do processo de evasão pois nenhuma colisão foi detectada, conforme a figura 2.10.

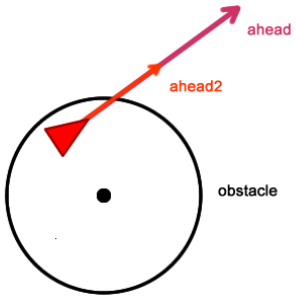

Figura 2.10: Collision Avoidance
Primeiro problema. Imagem retirada de : Understanding Steering Behaviors: Collision Avoidance [3]

Para resolver este problema, basta adicionar um terceiro vetor para a verificação da colisão: o vetor de posição do personagem. O uso de três vetores melhora muito a detecção de colisão[3].

O segundo problema acontece quando o personagem está perto do obstáculo, direcionando-se longe dele. Algumas vezes a manobra irá causar uma colisão, mesmo que o personagem esteja apenas rotacionando para encarar outra direção.

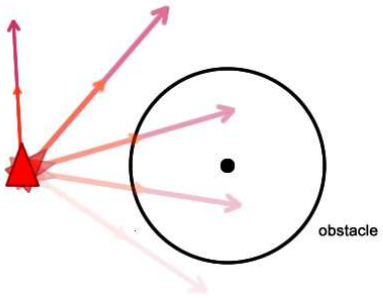

Figura 2.11: Collision Avoidance
Segundo problema. Imagem retirada de : Understanding Steering Behaviors: Collision Avoidance [3]

Este problema pode ser corrigido dimensionando os vetores anteriores de acordo com a velocidade atual do personagem.

2.2.4 *Wall Avoidance*

Este comportamento é parecido com *Collision Avoidance*, porém o personagem irá desviar-se de uma possível parede que encontra-se em seu caminho. Uma parede é um segmento de linha (em 3D, um polígono) com uma direção normal na direção em que está voltado para o *Wall Avoidance* direcionar para evitar possíveis colisões com paredes. Uma das formas de projetar este comportamento é através de três "antenas"na frente do personagem e testando para ver se qualquer uma delas intersecta com qualquer parede no mundo do jogo. "Isto é similar a como gatos e roedores usam seus bigodes para navegar em um ambiente no escuro[6]."

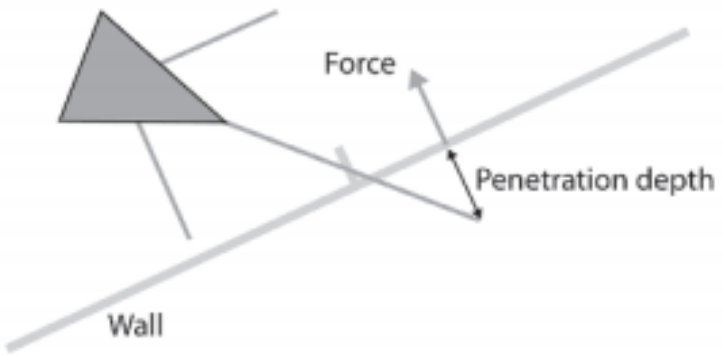

Figura 2.12: Wall Avoidance
Imagem retirada de : Programming Game AI by Example [6]

Quando a parede com a intersecção mais próxima é encontrada, uma *steering force* é calculada. Isto é deduzido calculando o quão longe a ponta da antena penetrou a parede e então criando uma força daquela magnitude na direção da normal da parede[6].

2.3 Game Engines

Atualmente muitas empresas implementam suas próprias *game engines* de alta qualidade ou utilizam alguma gratuita como a *Unity* 5. Mesmo esta sendo gratuita, caso você decidir publicar seu jogo, você é obrigado a pagar uma quantia de royalty dependendo do seu lucro.

Portanto antes de entrarmos em detalhe sobre a Unity 5 é preciso entender o que são *game engines* e como funcionam.

Game engine é uma plataforma de fazer tarefas comuns relacionadas a jogos como renderização, computação e física relacionada à entrada, para que os desenvolvedores(artistas, *designers*, *scripters* e outros programadores) possam focar nos detalhes que fazem seu jogo

único[16].

Visto que *game engines* são *frameworks* de desenvolvimento, criar a sua própria *game engine* permite que você optimize especificamente o seu jogo mas comprar uma ou usar uma gratuita irá lhe salvar possivelmente milhares de horas em tempo de codificação.

Game engines fornecem aos desenvolvedores uma série de componentes e auxílios que podem ser usados para construir seus jogos mais rápidos e com menos incômodo, mas o mecanismo de fator mais importante que fornecem são a interoperabilidade entre os vários sistemas de jogos possíveis[10].

Diferente de outros *frameworks* de desenvolvimento específicos para um sistema operacional ou somente uma plataforma, muitas *game engines* atuais possuem a possibilidade de desenvolver para multiplataformas. *Game engines* são feitas especificamente para criar jogos e todos seus componentes são organizados para fazer apenas isso, em detrimento de outras formas de aplicações[10].

2.3.1 *Unity 5*

A *Unity* é a *game engine* mais usada no mercado atual, atingindo uma porcentagem de 45% do mercado, conforme a figura 2.13.

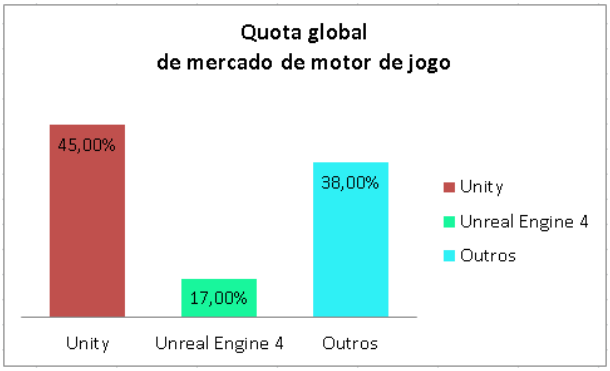

Figura 2.13: Unity e outras game engines
Imagem original retirada de : This engine is dominating the gaming industry right now [7]

Para distinguir melhor os motivos por trás de ter sido usada a *Unity*, eles foram divididos em tópicos.

- **Curva de aprendizagem**: A *Unity* possui uma abundância de tutoriais de suporte online. É muito possível que desenvolvedores profissionais independentes(*indie*) de aplicativos *mobile* peguem o "jeito"da *Unity* em uma semana ou mais e comecem a fazer jogos pequenos[23].

- **O fator do preço**: A *Unity* possui uma versão gratuita, porém carece de muitas de suas melhores características de seu motor e para uso profissional é necessário escolher a versão *Unity* Pro, a qual sai por U$ 125.00 por mês[21].

- **Linguagens de codificação subjacentes**: A *Unity* trabalha com Javascript e C#, portanto, requer que usuários possuam uma boa base em alguma delas ou ambas.

- **Qualidade dos gráficos**: A *Unity* possui ótimos gráficos mas para obter-se a qualidade máxima de suas ferramentas é preciso comprar a versão paga pois a versão gratuita produz somente visuais medianos.

- **Utilidade para criar jogos mobile**: A *Unity* é a *game engine* mais recomendada para jogos *mobile*. A razão para isto é simples, mesmo que a versão gratuita da *Unity* ofereça suporte completo para todas as principais plataformas *mobile*, ela é geralmente vista como mais conveniente para projetos relativamente pequenos.

- **Recursos da loja**: A loja de recursos da *Unity* possui muitas ferramentas a mais que a maioria das outras *game engines* disponíveis no mercado, isto inclui acessórios gerais de jogos e personagens.

2.4 Uma breve introdução a desempenho em jogos

Antes de entrarmos no capítulo Metodologia onde foi discutido sobre a coleta de dados é importante entender como funciona o desempenho em jogos. Taxa de quadros é uma medida padrão de desempenho em jogos. Em jogos, um quadro é como um quadro em uma animação, é uma imagem parada de nosso jogo que é desenhada na tela. Desenhando um quadro na tela é conhecido como renderização de quadro. Taxa de quadros, ou quão rápido os quadros estão sendo renderizados, é uma medida em quadros por segundos(FPS).

A maioria dos jogos modernos tem como objetivo alcançar uma taxa de quadros de 60 FPS. Geralmente uma taxa de quadros acima de 30 FPS é consideravelmente aceitável, especialmente para jogos que não requerem rápidas reações como jogos de quebra-cabeça ou jogos de aventura. Em taxas de quadros abaixo de 30 FPS, jogadores geralmente acham a experiência não agradável, gráficos podem parecer irregulares e controles podem parecer não responsivos[22].

Para cada quadro que é renderizado, a *Unity* deve realizar várias tarefas diferentes. Em

termos simples, a *Unity* deve atualizar o estado do jogo, tirar uma foto instantânea do jogo e então desenhar esta foto na tela[22]. Tarefas que devem ocorrer durante cada quadro incluem coisas como ler entradas do usuário, executar *scripts* e desempenhar cálculos de iluminação. Além disto, existem operações que podem acontecer muitas vezes durante um único quadro, como cálculos de física.

Quando todas estas tarefas estão sendo executadas rápido o suficiente, nosso jogo terá uma taxa de quadros consistente e aceitável. Caso contrário, quando todas estas tarefas não podem ser executadas rápido o suficiente, quadros irão levar muito tempo para renderizar e a taxa de quados irá cair.

3 TRABALHOS RELACIONADOS

Ano	Título do trabalho	Autor(es)	Característica principal
1999	Steering Behaviors For Autonomous Characters	Craig W. Reynolds	Introdução a personagens autonomos e steering behaviors
2003	Fast,Neat, and Under-Control: Arbitrating Between Steering Behaviors	Heni Ben Amor, Jan Murray e Oliver Obst	Navegação usando Inverse Steering Behaviors
2005	Programming Game AI by Example	Mat Buckland	Sólida fundação prática para IA de jogos
2005	Autonomous behaviors for interactive vehicle animations	Jared Go, Thuc D. Vu, James J. Kuffner	Framework de planejamento de caminhos
2010	Natural steering behaviors for virtual pedestrians	Renato Silveira, Fabio Dapper, Edson Prestes e Luciana Nedel	Uma solução formalmente completa e de baixo custo para controlar steering behaviors de personagens em ambientes dinâmicos
2012	The nature of code	Daniel Shiffman	Algoritmos e suas técnicas de programação afiliadas
2014	Understanding Steering Behaviors	Fernando Bevilacqua	Tutorial Steeering Behaviors

Tabela 3.1: Trabalhos relacionados

Dentre os trabalhos relacionados, alguns foram mais importantes e tiveram maior influência ao decorrer do trabalho. Entre eles estão *Steering Behaviors For Autonomous Characters, Motion in Games, SteerBench: A Benchmark Suite for Evaluating Steering Behaviors e The nature of code* e *Understanding Steering Behaviors*.

Steering Behaviors For Autonomous Characters é o artigo apresentado por Craig Reynolds na *Game Developers Conference* em 1999 descrevendo detalhadamente sua criação. Foi de suma importância sua leitura e compreensão, não somente para o referencial teórico como também para a definição do tema.

Motion in Games ajudou na parte de encontrar possíveis casos de testes e métricas para validar diversos tipos de movimentos em jogos e também o livro em que foi encontrado o *benchmark* para validar os *steering behaviors*.

SteerBench: A Benchmark Suite for Evaluating Steering Behaviors é o artigo original em que foi encontrado o resumo no livro *Motion in Games*, tendo informações mais detalha-

das sobre os casos de teste, métricas, avaliações das métricas e como validar devidamente um algoritmo ou duas abordagens diferentes de algoritmos de *steering behaviors*.

The nature of code é um livro online porém pode ser comprado também impresso publicado por Daniel Shiffman em que auxiliou não somente na descrição de *steering behaviors* mas principalmente sobre personagens autônomos.

Understanding Steering Behaviors permitiu principalmente entender a teoria e dar uma ideia de como foi a implementação. Possibilitou claro entendimento sobre *steering behaviors*, com exemplos com figuras e pseudo-códigos que foram úteis para entendimento no momento da implementação.

4 METODOLOGIA

4.1 Abordagem

O método de abordagem escolhido para este trabalho é o quantitativo, onde as variáveis observadas são poucas, objetivas e medidas em escalas numéricas. Segundo[24] filosoficamente, a pesquisa quantitativa baseia-se numa visão dita positivista onde:

- As variáveis a serem observadas são consideradas objetivas, isto é, diferentes observadores obterão os mesmos resultados em observações distintas.
- Não há desacordo do que é melhor e o que é pior para os valores dessas variáveis objetivas.
- Medições numéricas são consideradas mais ricas que descrições verbais, pois elas adéquam-se à manipulação estatística.

A essência da pesquisa quantitativa em ciência da computação é verificar o quão "melhor" é usar um programa/sistema novo frente à(s) alternativa(s)[24].

Os métodos tomados neste trabalho possuem teor indutivo como base para a metodologia, ou seja, não sabemos a verdade sobre a performance da Unity 5 para *steering behaviors* e estamos à procura de descobrir, podendo haver diferenças significativas ou não, levando à afirmação ou negação sobre o resultado da sua performance.

Indução é um processo mental por intermédio do qual, partindo de dados particulares, suficientemente constatados, infere-se uma verdade geral ou universal, não contida nas partes examinadas[12].

4.2 Ferramentas

Nesta seção são apresentadas as ferramentas utilizadas e o ambiente de desenvolvimento, desde as configurações físicas do computador ao software utilizado. A tabela 4.1 informa as especificações do ambiente de desenvolvimento. A tabela 4.2 cita as informações sobre as ferramentas que foram utilizadas.

Tabela 4.1: Configuração de hardware do ambiente de desenvolvimento

Item	Descrição
Processador	Intel Core i5-5200 2.20Ghz
Placa Mãe	ASUSTeK COMPUTER INC. X555LF
Memória	DDR3L 1600 MHz SDRAM 8GB
Armazenamento	1TB 5400R SATA
Placa de Vídeo	NVIDIA GeForce 930M com 2GB DDR3 VRAM

Tabela 4.2: Ferramentas escolhidas para desenvolvimento

Ferramenta	Linguagem	Versão Ferramenta
Unity	C#	5.6.2f1

4.3 Casos de teste

Os casos de teste usados para avaliar cada abordagem de cada comportamento foram do *Benchmark: SteerBench: A Benchmark Suite for Evaluatin Steering Behaviors*. A partir de um conjunto de cenários é possível capturar uma ampla gama de situações em que um algoritmo de *steering* pode encontrar em aplicações práticas[20].

Antes de entrarmos em detalhes, é importante ressaltar que os casos de teste do *SteerBench* servem para descobrir o desempenho do algoritmo e/ou diferentes abordagens, assim valendo-se apenas do desempenho do algoritmo durante os testes. Com isso em mente, foi coletado dados importantes sobre métricas de qualidade que servem para validar o comportamento dos algoritmos e também para métricas de desempenho que servem para validar a qualidade de desempenho da *Unity* 5.

O *benchmark* em si não foi usado pois os casos de teste foram simulados na própria *Unity* 5 em que é possível criar os cenários e aproveitar isto para descobrir seu desempenho não somente para os *steering behaviors* em si mas também com eles em ação com outros agentes/obstáculos. Sendo assim as informações coletadas referentes as métricas foram feitas a partir da *Unity* 5 e não a partir do *benchmark*.

4.3.1 Descrição dos cenários

Os casos de teste escolhidos para validação são classificados nas seguintes categorias conceituais:

- Cenários simples de validação;
- Interações básicas um-a-um;
- Interações agente-a-agente incluindo obstáculos.

Cenários simples de validação: Estes cenários são projetados para testar habilidades básicas e fundamentais de um agente de direção. Enquanto tais comportamentos são triviais, ainda é importante para um algoritmo lidar com sucesso estes casos:

- *Cenário 1 - Simples:* um agente direcionando-se até um alvo localizado na esquerda, direita ou atrás;
- *Cenário 2 - Parede-simples:* dois agentes direcionando-se ao redor de uma parede para chegar ao alvo;
- *Cenário 3 - Curvas:* um agente direcionando-se através de uma curva-S para chegar ao alvo.

Interações básicas um-a-um: Estes cenários testam a habilidade dos agentes de direcionarem-se ao redor de si. Nesta categoria, a ênfase está nas interações naturais sem outras ameaças para distrair os agentes:

- *Cenário 4 - Sentido contrário:* dois agentes viajando em direções opostas em direção a uma colisão frontal;
- *Cenário 5 - Atravessando:* dois agentes que atravessam caminhos em vários ângulos;
- *Cenário 6 - Direção-similar:* dois agentes com objetivos pouco diferentes viajando em uma direção similar.

Interações agente-a-agente incluindo obstáculos: Estes cenários testam a habilidade de um agente para navegar ao redor de objetos estáticos enquanto interagindo com outros agentes:

- *Cenário 7 - Sentido contrário-obstáculo:* dois agentes de direções contrárias, com um obstáculo adicional no caminho;
- *Cenário 8 - Atravessando-obstáculo:* dois agentes atravessando, com um obstáculo adicional no caminho;

- *Cenário 9 - Aperto:* dois agentes de sentidos contrários andando através de um corredor estreito.

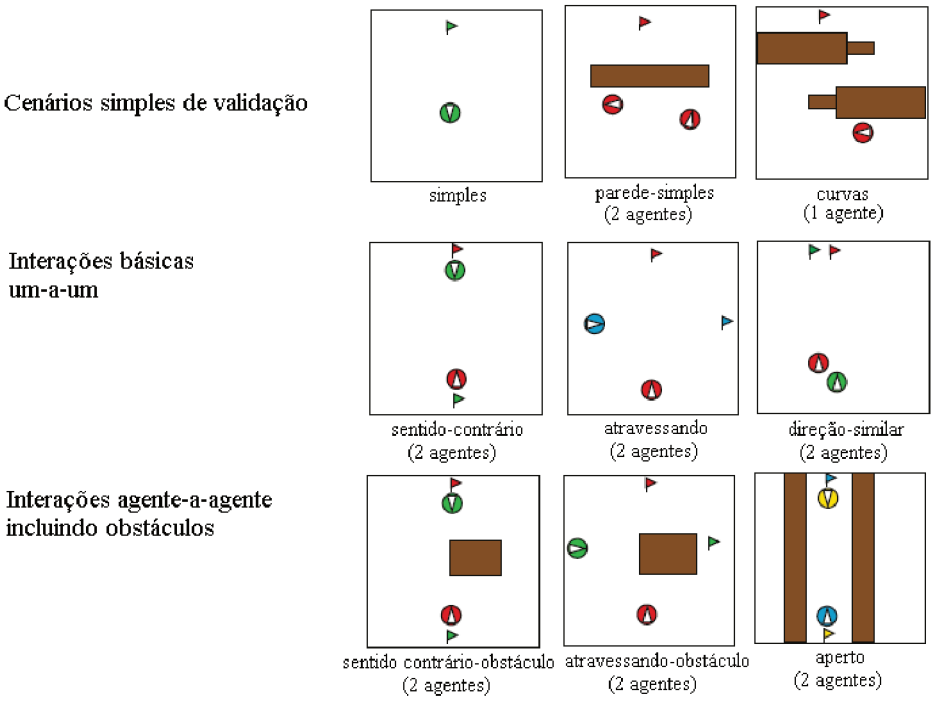

Figura 4.1: Representação aproximada dos cenários do *Benchmark SteerBench*. Os números em parênteses representam quantos agentes são especificados para cada cenário . Imagem original retirada de: Motion in Games[9]

4.4 Métricas de Avaliação

As métricas de avaliação foram dividas em duas categorias, qualidade e desempenho. A primeira categoria visa descobrir a qualidade do comportamento do agente e a segunda tem como objetivo detectar o desempenho da *Unity* 5 durante as simulações. Na tabela 4.3 podemos ver quais foram as métricas e a qual categoria corresponderam.

4.5 Importância das métricas escolhidas

Nesta seção foi descrito a importância de coletar dados sobre cada métrica. Primeiramente vamos falar sobre as métricas de qualidade:

- Velocidade média: é uma grandeza que identifica o deslocamento de um corpo num determinado tempo. Logo a velocidade média mede num intervalo de tempo médio a rapidez, da deslocação de um corpo;

Métrica	Categoria
Velocidade Média	Qualidade
Aceleração Média	Qualidade
Deslocamento	Qualidade
Distância Percorrida	Qualidade
Tempo Percorrido	Qualidade
FPS Baixo	Desempenho
FPS Médio	Desempenho
FPS Alto	Desempenho
TSMU	Desempenho

Tabela 4.3: Métricas de qualidade

- Aceleração média: é a grandeza física que representa a variação da velocidade por unidade de tempo e por isto é importante para descobrirmos qual foi a variação da velocidade de cada agente;

- Deslocamento: é a medida da linha reta que une a posição inicial e a posição final, o seu valor só depende destas posições, não depende da trajetória. Ambos distância percorrida como deslocamento serão importantes ao serem comparados para descobrir se a distância percorrida ultrapassou muito o deslocamento;

- Distância percorrida: a distância percorrida (ou espaço percorrido) é a medida sobre a trajetória descrita no movimento, o seu valor depende da trajetória;

- Tempo percorrido: sendo uma grandeza escalar, é a medida que mede quanto tempo um agente demorou para chegar ao alvo;

As métricas de desempenho são importantes para relatar o desempenho da *game engine* no uso de *steering behaviors*:

- FPS baixo: FPS representa a taxa de quadros em um jogo, o tempo que leva para renderizar uma imagem na tela, um FPS abaixo de 30 é considerado baixo e prejudica a experiência do jogador pois haverá atraso na renderização de imagens, controles podem pareces não responsivos e entre outros;

- FPS médio: Geralmente uma taxa de quadros acima de 30 FPS é considerada aceitável, especialmente para jogos que não requerem rápidas reações como jogos de quebra cabeça ou de aventura;

- FPS alto: Jogos modernos tem como objetivo uma taxa de quadros de 60 FPS, acima disso ela pode ser considerada alta, porém alguns projetos tem requisitos especiais, realidade virtual, por exemplo, 90 FPS é considerado crítico[22];

- TSMU(Memória Total de Uso do Sistema): é o tamanho total usado da memória pelo processo da *Unity*. Se TSMU > 1GB o sistema pode paginar alguma memória para o disco, mas isto não é garantido e pode ocasionar péssimo desempenho. Portanto é sempre mais seguro ter TSMU < 1GB[26]. Este dado é coletado a partir da própria ferramenta da *Unity*, o *memory profiler*.

Embora a *Unity 5* seja capaz de proporcionar gráficos incríveis, assim sendo provavelmente importante coletar dados de desempenho da GPU e não somente CPU como foi o caso deste trabalho, este trabalho preocupou-se apenas em coletar dados de CPU pois os cenários e objetos de jogo são em 2D e não possuem gráficos relevantes a ponto de ser necessário coletar dados de GPU.

4.6 Coleta de Dados

Para uma compreensão melhor sobre as métricas é importante destacar como foram coletados os dados e também quais são suas unidades de medida:

- Velocidade média: o quociente entre o deslocamento e o correspondente intervalo de tempo;

- Aceleração média: o quociente da variação de uma velocidade sofrida em um dado tempo;

- Deslocamento: a variação na posição de um corpo independente da trajetória;

- Distância percorrida: o comprimento total do caminho percorrido entre duas posições, onde depende da trajetória;

- Tempo percorrido: o tempo em segundos que o agente levou para chegar ao alvo;

As métricas de qualidade exceto deslocamento são coletadas a cada quadro do jogo e somente retornadas ao final da simulação ao *console*. O deslocamento não é preciso coletar a cada quadro das simulações pois ele é diretamente a subtração da posição do alvo com a posição inicial do agente. Por outro lado para calcular a distância percorrida, a cada quadro do

jogo é coletada a última posição do agente, ao final da simulação, esta posição é subtraída da posição inicial dele. Ao multiplicar cada quadro do jogo da variável que recebe o tempo com *Time.deltaTime* é garantido que a unidade de tempo seja em segundos. Por padrão a unidade de medida de comprimento da *Unity* 5 é metros, portanto as unidades de medida são em m/s.

- FPS baixo: coletado quando a taxa de quadros foi < 30 FPS;
- FPS médio: coletado no intervalo de 30 <= FPS <= 60;
- FPS alto: coletado quando a taxa de quadros foi > 60 FPS;
- TSMU: coletada no fim da simulação.

Diferente do TSMU, onde é coletado a partir da ferramenta *Memory profiler* presente na *Unity* 5, o FPS foi coletado a partir do seguinte cálculo a cada chamada do método *Update*: **FPS = (int)(1f / Time.deltaTime)**. Onde *Time.deltaTime* retorna o tempo em segundos que levou para completar o último quadro.

Ao contrário de quase todas métricas de qualidade, que retornam seus dados somente assim que cada agente atingiu seu objetivo, estas métricas retornam os dados a cada quadro de jogo, indiferente de n agentes terem chegado no objetivo ou não. Quando todos agentes chegam em seus objetivos a coleta destas métricas como a simulação é encerrada.

Ao final da coleta de dados são calculados média aritmética simples, desvio padrão populacional e coeficiente de variação.

A média aritmética representa o "centro da gravidade"da distribuição, isto é, o ponto de qualquer distribuição em torno do qual se equilibram as discrepâncias positivas e negativas. Situa-se entre o valor máximo e mínimo da distribuição. É importante pois é um valor que pretende ser o resumo de todos os valores da distribuição, dessa forma pode vir a ser uma valor não presente na distribuição[11].

O desvio padrão é uma medida que indica a dispersão dos dados dentro de uma amostra com relação à média. Assim, quando se calcula o desvio padrão juntamente com a média de diferentes grupos, obtém-se mais informações para avaliar e diferenciar seus comportamentos. Assim, quando o desvio padrão da distribuição é pequeno a amostra é homogênea, quando o valor é alto a amostra é heterogênea.

O coeficiente de variação (CV) é uma medida de dispersão relativa, pois expressa a relação percentual do desvio padrão em relação a média. Usado para definir se os desvios

padrões das médias foram baixos, médios ou altos. Como o CV analisa a dispersão em termos relativos, ele é dado em porcentagem. Quanto menor for o valor do coeficiente de variação, mais homogêneos serão os dados, ou seja, menor será a dispersão em torno da média. A análise do CV é dado da seguinte maneira:

- Menor ou igual a 15%: baixa dispersão, logo, os dados são homogêneos;
- Entre 15% e 30%: média dispersão;
- Maior que 30%: alta dispersão, logo, os dados são heterogêneos.

5 IMPLEMENTAÇÃO

5.1 Trabalhando com a *Unity*

Antes de explicarmos os *scripts* é importante vermos brevemente como tudo isto funciona na interface da *Unity*. A figura 5.1 mostra como é o processo de funcionamento da Unity:

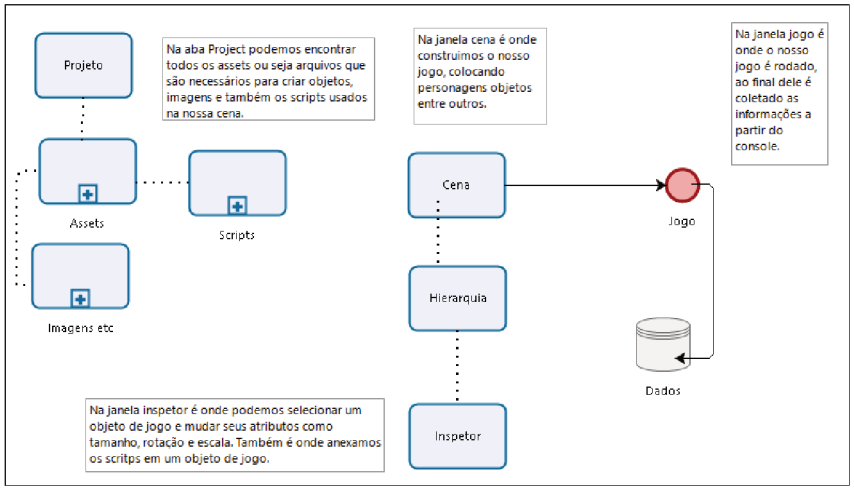

Figura 5.1: Fluxograma funcionamento Unity 5

Logo ao abrirmos a *Unity* nos deparamos com a tela inicial que possui as seguintes janelas:

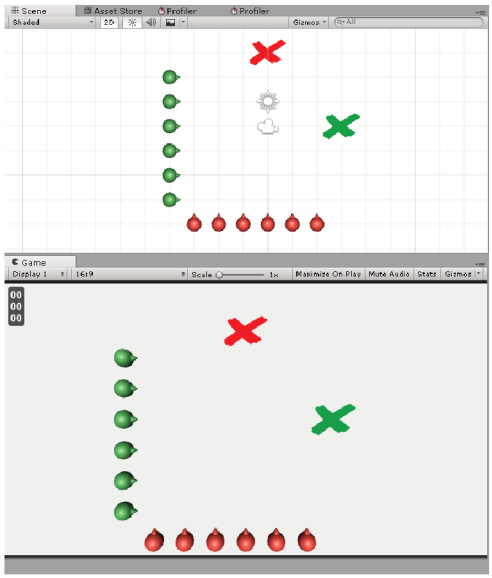

Figura 5.2: Janelas Tela e Jogo

A visão da cena contêm os ambientes e menus do seu jogo. Pense em cada arquivo de cena exclusivo como um nível único. Em cada cena, você coloca seus ambientes, obstáculos e

decorações, essencialmente projetando e construindo seu jogo em pedaços. Já a visão do jogo é processada a partir da(s) câmera(s) do seu jogo. Ela representa o seu jogo final.

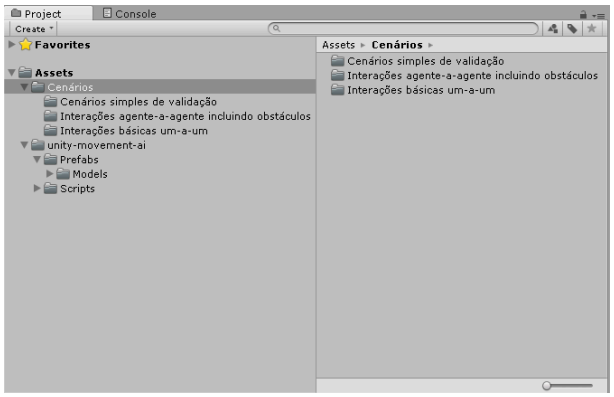

Figura 5.3: Janelas Projeto e Console

Na janela do projeto, o painel da esquerda do navegador mostra a estrutura do projeto como uma lista hierárquica. Quando uma pasta é selecionada ao ser clicada, seus conteúdos são mostrados no painel da direita. Como podemos ver na figura 5.3 temos os cenários, *prefabs*, que são objetos personalizados que fazem parte da maior parte de seu jogo, modelos e os *scripts*. E na segunda aba temos a janela *console* que mostra erros, advertências e outras mensagens geradas pela *Unity*.

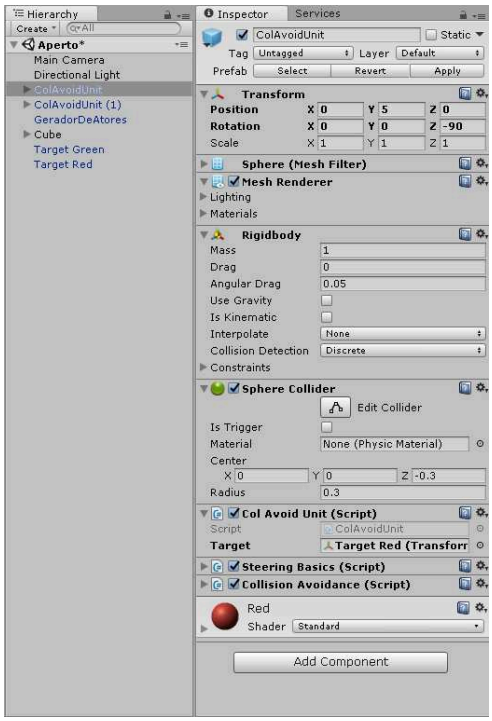

Figura 5.4: Janelas Hierarquia e Inspetor

A janela da hierarquia contém uma lista de cada objeto de jogo(*GameObject*) na presente

cena. Alguns destes são instâncias diretas de um conjunto de arquivos(como modelos 3D), e outros instâncias de *prefabs*. Conforme objetos são adicionados e removidos da cena, eles irão aparecer ou desaparecer respectivamente da hierarquia.

Projetos no editor da *Unity* são feitos com múltiplos objetos de jogo que contém *scripts*, sons, malhas, e outros elementos gráficos como luz. A janela Inspetor mostra infomações detalhadas sobre um objeto de jogo selecionado, incluindo todos componentes anexados e suas propriedades, e permite que você modifique a funcionalidade de um objeto de jogo na sua cena.

5.2 Implementação dos comportamentos

Nesta seção foram tratados como foram implementados os comportamentos *Arrival, Collision Avoidance e Wall Avoidance* e também quais foram as suas modificações. Eles foram implementados e modificados com base nos *scripts* de Anton Pantev: *Unity Movement AI*[15]. As classes *Steeringbasics, Arrive Unit, Collision Avoidance Unit e Wall Avoidance Unit* foram modificadas um pouco para atender os cenários das simulações. A classe *TargetSphere* foi criada também para atender aos cenários das simulações e foi explicada.

As implementações estão separadas entre dois tipos de classes: *MonoBehaviour* e *UnityScript*. O primeiro tipo de classe é a classe base de qual todo *Unity script* deriva. Ao iniciarmos um novo *script* na Unity sempre teremos dois métodos: *Start* e *Update*. O método *Start* é chamado somente no inicio do jogo, ou seja, no primeiro quadro a ser rodado. Por outro lado o método *Update* é chamado a cada quadro do jogo.

Por fim a classe do tipo *UnityScript* não precisa explicitamente derivar de *MonoBehaviour*. As classes *Steeringbasics,Collision Avoidance, Wall Avoidance,NearSenro* e *TargetSphere* são do tipo *UnityScript* e as demais do tipo *MonoBehaviour*. Cada personagem tem anexado a ele mesmo as classes *Steeringbasics e TargetSphere* e as classes dos tipos *MonoBehaviour* e *UnityScript* de acordo com o comportamento requerido.

Possivelmente por limitação da *Unity*, não foi possível usar como condição de se o a posição atual do agente é menor que a posição atual do alvo, pois ao usar esta abordagem, o agente procurava sempre o centro do raio do alvo, assim quando haviam muitos agentes procurando o mesmo alvo e estavam próximos, eles ficavam empurrando uns aos outros e dificilmente chegando exatamente no centro do raio do alvo. Tendo isto em mente, foi usada a abordagem de usar uma esfera de colisão no alvo para detectar a colisão com a posição atual do alvo ou com sua esfera de colisão quando necessária, assim podendo haver vários alvos chegando ao

mesmo tempo de diferentes posições.

5.3 *Steering basics*

Primeiramente precisamos entender a classe *Steeringbasics*, uma classe que auxilia o objeto a movimentar-se em 2D. As seguintes variáveis globais foram inicializadas com valores padrão:

- `maxVelocity`: a velocidade máxima do personagem;
- `maxAcceleretation`: a aceleração máxima do personagem;
- `slowRadius`: o raio do alvo para que comecemos a desacelerar;
- `timeToTarget`: o tempo que queremos para alcançar a velocidade alvo;
- `turnSpeed`: a velocidade de virar do personagem;
- `rb`: variável do tipo *rigidbody*, ou seja, corpo rígido que possui propriedades como velocidade, massa, entre outros;
- `targetSphere`: variável que não estava no código original, do tipo *TargetSphere* usada para acessar seus métodos e conseguir detectar a colisão com a esfera do alvo;
- `nearSensor`: variável que não estava no código original, do tipo *NearSensor* usada para acessar os seus métodos;
- `col`: variável que não estava no código original, do tipo bool usada para selecionar se o comportamento que o agente possui colisão ou não.

O primeiro método da classe é o *Start* e nele temos:

Algorithm 1 Método *Start*

```
1: rb = GetComponent<Rigidbody> ();
2: targetSphere = GetComponent<TargetSphere>();
3: if col = true then
4:     nearSensor = transform.Find("ColAvoidSensor").GetComponent<NearSensor>();
5: end if
```

A primeira linha é apenas uma atribuição da variável `rb` para receber um componente do tipo *Rigidbody*. As próximas linhas foram criadas para atender as necessidades dos códigos

que usam esta classe, na segunda linha a variável `targetSphere` recebe um componente do tipo *TargetSphere* que é nossa classe que será explicada mais a frente. A condição do *if* seguinte é para verificarmos se o nosso comportamento usará também um outro raio de esfera que serve para evitar colisões com outros agentes, caso seja verdadeiro a variável `nearSensor` recebe o componente *ColAvoidSensor*, isto ficará mais claro quando a classe *NearSensor* for explicada.

Os métodos mais importantes são *steer*, *seek*, *arrive* e *getBoundingRadius*. Na seção de apêndices é possível encontrar esta classe completa. O método *steer* recebe como parâmetro um vector3D *linearAcceleration* e atualiza a velocidade atual do objeto do jogo, no caso o personagem dado a aceleração linear:

Algorithm 2 Método *Steer*

```
1: rb.velocity =+ linearAcceleration * Time.deltaTime
2: if rb.velocity.magnitude > maxVelocity then
3:     rb.velocity = rb.velocity.normalized * maxVelocity
4: end if
```

Note que na primeira linha o campo `velocity` da variável rb está somando ele mesmo com a aceleração linear e multiplicando por `Time.deltaTime`, este sendo um método da *Unity* para tornar a taxa de quadros do jogo independente, ou seja, ao multiplicar por ela, estou expressamente dizendo quero mover este objeto a 10m/s e não a 10 metros por quadros.

O método *seek* retorna a direção para o personagem buscar dada posição. Sendo um método do tipo Vector3, ela recebe como parâmetro um vector3D da posição do alvo e um *float* da máxima aceleração do *seek*:

Algorithm 3 Método *seek*

```
1: Vector3 acceleration = targetPosition - transform.position;
2: acceleration.z = 0;
3: acceleration.Normalize ();
4: acceleration *= maxSeekAccel;
5: return acceleration;
```

Podemos notar que na primeira linha o vetor `acceleration` está recebendo a subtração da posição do alvo e da posição atual do personagem. Na linha seguinte é removido a coordenada z pois estamos trabalhando em 2D e em seguida é normalizado este vetor. Por fim multiplicamos este vetor `acceleration` com `maxSeekAccel` e retornamos a aceleração final.

O método *arrive* recebe como parâmetro um vector3D da posição do alvo e retorna uma direção para o personagem para que então ele chegue ao alvo, o método *OnTriggerEnterTarget*

foi adicionado ao código original para funcionar nos cenários das simulações.

Algorithm 4 Método *Arrive*

```
1:  Vector3 targetVelocity = targetPosition - transform.position;
2:  targetVelocity.z = 0;
3:  float dist = targetVelocity.magnitude;
4:  if !col then
5:     if dist <= targetSphere.sphereRadius then
6:        OnTriggerEnterTarget (targetSphere.sphereTarget);
7:        return Vector3.zero;
8:     end if
9:  else if col then
10:    if dist <= targetSphere.sphereRadius + nearSensor.actorSphere.radius) then
11:       OnTriggerEnterTarget (targetSphere.sphereTarget);
12:       return Vector3.zero;
13:    end if
14: end if
15: float targetSpeed;
16: if dist > slowRadius then
17:    targetSpeed = maxVelocity;
18: else if targetSpeed = maxVelocity * (dist / slowRadius) then
19: end if
20: targetVelocity.Normalize();
21: targetVelocity *= targetSpeed;
22: Vector3 acceleration = targetVelocity - new Vector3(rb.velocity.x, rb.velocity.y, 0);
23: acceleration *= 1/timeToTarget;
24: if acceleration.magnitude > maxAcceleration then
25:    acceleration.Normalize();
26:    acceleration *= maxAcceleration;
27: end if
```

Na linha 2 a coordenada do vetor `targetVelocity` é setada para zero, ou seja, removida pois a implementação foi feita em 2D. As próximas linhas são linhas não presentes no código original e foram criadas, para verificar se a variável `bol` é verdadeira ou não. Esta verificação é usada para definir se o comportamento que estamos usando possui ou não uma esfera de detecção de colisão(isto será explicado melhor na classe *NearSensor*). Caso seja falsa, a distância do personagem é verificada se menor ou igual ao raio de uma esfera pertencente ao alvo. A linha *OnTriggerEnterTarget* significa que ao colidirmos com a esfera do alvo recebemos uma mensagem confirmando esta colisão. Logo se esta distância for menor então chegamos no alvo e paramos.

Caso a variável `bol` seja verdadeira, isto significa que o comportamento que estamos lidando com, possui sua própria esfera de detecção de colisão. Portanto a distância deve ser comparada se ela é menor que a soma do raio da esfera do alvo e o raio da esfera do agente. Se

for menor então chegamos no alvo e paramos. Da linha 15 até a linha 18 é calculada a velocidade alvo, sendo velocidade máxima na distância do raio de desaceleração e 0 na distância 0.

Conforme a linha 21 é preciso dar a `targetVelocity` a velocidade correta, mas antes normalizar o vetor. A linha 23 é calculada a aceleração linear que desejamos, ao invés de acelerar o personagem para a velocidade correta em um segundo, aceleramos para que chegue à velocidade desejada em `timeToTarget` segundos. Por fim certificamos que a aceleração esteja na aceleração máxima conforme as linhas 24 a 26, assim ao final retornando a aceleração.

O método criado *OnTriggerEnterTarget* funciona da seguinte maneira:

Algorithm 5 Método *getBoundingRadius*

```
1: if GetComponent<Collider> ().GetType () == typeof(SphereCollider) then
2:     rb.velocity = Vector3.zero;
3:     this.transform.localScale = new Vector3 (0, 0, 0);
4: end if;
```

A única condição verifica se o componente do tipo colisor encontrado é do tipo colisor de esfera, caso seja, a velocidade do personagem é setada para 0 e sua posição setada para (0,0,0) para fazer com que ele desapareça da cena do jogo.

Outro método importante usada para o comportamento *Collision Avoidance* é a *getBoundingRadius*. Ela recebe como parâmetro uma variável t do tipo *Transform*:

Algorithm 6 Método *getBoundingRadius*

```
1: SphereCollider col = t.GetComponent<SphereCollider>();
2: return Mathf.Max(t.localScale.x, t.localScale.y, t.localScale.z) * col.radius;
```

Na primeira linha é criado uma variável do tipo *SphereCollider*, conforme o nome ela é um colisor de esfera usada para detectar colisões entre os personagens. Ela recebe uma variável que acessa o método *GetComponent* que recupera um componente, neste caso um *SphereCollider*. A última linha retorna o maior valor entre dois ou mais valores, sendo estes valores a multiplicação da escala das três coordenadas x,y,z com o raio do colisor de esferas. O comando *localScale* retorna a escala da transformação em relação ao pai.

5.4 *Arrival Unit*

Esta é a classe que chama os métodos do comportamento *arrival* e recebe o objeto que é o alvo. Nesta classe temos somente duas variáveis globais sendo que a variável *target* é a variável nova criada:

- `steeringBasics`: variável do tipo *Steering Basics* usada para acessar seus métodos;
- `target`: variável do tipo *Transform* que possibilita que seja escolhido um objeto de jogo como alvo e é possível acessar a posição, rotação e tamanho do objeto;

Dentro do método *Update* desta classe temos:

Algorithm 7 Método *Update*

```
1: Vector3 accel = steeringBasics.arrive(target.position);
2: steeringBasics.steer(accel).
```

A primeira linha possui um vector3D que passa a posição do alvo para o método *arrive* da classe *steeringBasics* e após isso o método *steer* recebe como parâmetro o mesmo vetor para direcioná-lo em direção do alvo. Vale lembrar que o método *Update* é um método chamado a cada frame do jogo.

5.5 *Collision Avoidance*

Nesta classe implementada para o comportamento *Collision Avoidance* temos três variáveis globais:

- `maxAcceleration`: variável do tipo float representando a máxima aceleração do personagem;
- `characterRadius`: raio do personagem do tipo float;
- `rb`: variável do tipo *rigidbody*, ou seja, corpo rígido que possui propriedades como velocidade, massa, entre outros.

Nesta classe existem somente dois métodos, um delas sendo a *Start*, do tipo *void* sendo chamado somente ao iniciar o jogo, para as variáveis `characterRadius` e `rb` receberem suas devidas atribuições. O objetivo do método *getSteering* é desviar dos objetos a sua frente, ele recebe como parâmetro uma coleção de variáveis do tipo *Rigidbody*:

Algorithm 8 Método *Start*

```
1: characterRadius = SteeringBasics.getBoundingRadius(transform);
2: rb = GetComponent<Rigidbody>();
```

Algorithm 9 Método *getSteering*

```
1: Vector3 acceleration = Vector3.zero;
2: float shortestTime = float.PositiveInfinity;
3: Rigidbody firstTarget = null;
4: float firstMinSeparation = 0, firstDistance = 0, firstRadius = 0;
5: Vector3 firstRelativePos = Vector3.zero, firstRelativeVel = Vector3.zero;
6: for each Rigidbody r in targets do
7:     Vector3 relativePos = transform.position - r.position;
8:     Vector3 relativeVel = rb.velocity - r.velocity;
9:     float distance = relativePos.magnitude;
10:    float relativeSpeed = relativeVel.magnitude;
11:    if relativeSpeed == 0 then
12:        continue;
13:    end if
14:    float timeToCollision = -1 * Vector3.Dot(relativePos, relativeVel) / (relativeSpeed * re-
   lativeSpeed);
15:    Vector3 separation = relativePos + relativeVel * timeToCollision;
16:    float minSeparation = separation.magnitude;
17:    float targetRadius = SteeringBasics.getBoundingRadius(r.transform);
18:    if minSeparation > characterRadius + targetRadius then
19:        continue;
20:    end if
21:    if timeToCollision > 0 && timeToCollision < shortestTime then
22:        shortestTime = timeToCollision;
23:        firstTarget = r;
24:        firstMinSeparation = minSeparation;
25:        firstDistance = distance;
26:        firstRelativePos = relativePos;
27:        firstRelativeVel = relativeVel;
28:        firstRadius = targetRadius;
29:    end if
30: end foreach
31: if firstTarget == null then
32:    return acceleration;
33: end if
34: if firstMinSeparation <= 0 || firstDistance < characterRadius + firstRadius then
35:    acceleration = transform.position - firstTarget.position;
36: else if
37:     thenacceleration = firstRelativePos + firstRelativeVel * shortestTime;
38: end if
39: acceleration.Normalize();
40: acceleration *= maxAcceleration;
41: return acceleration;
```

Primeiramente da linha 2 até 30 procuramos o primeiro alvo que se irá se colidir com o personagem. Na linha 2 temos o tempo da primeira colisão. Entre as linhas 3 e 5 temos o primeiro alvo que irá colidir e outros dados que precisamos e podem evitar recálculo. A seguir temos um `for each` para cada corpo rígido de cada alvo da nossa coleção que é passada no parâmetro no início de nosso método *getSteering*. Dentro deste laço, entre as linhas 7 e 15 é calculado o tempo para colisão e entre as linhas 15 e 20 é verificado se irão colidir ou não.

Agora é preciso verificar qual foi a colisão de menor tempo e isto é feito dentro do `if` da linha 21 a 29. Na segunda parte do algoritmo é calculado a direção do personagem. Da linha 31 a 33 é feita uma comparação que caso seja verdadeira significa que não encontramos alvo então saímos e retornamos a aceleração.

Na próxima condição verificamos se haverá colisão sem separação ou se já estamos colidindo, caso estejamos a direção é baseada na posição atual. No caso de entrar no `else` calculamos a futura posição relativa. Nas duas linhas finais antes de retornarmos a aceleração, é feita a normalização do vetor e a multiplicação dele mesmo com a aceleração máxima, para evitar o alvo.

5.5.1 *Collision Avoidance Unit*

Esta classe tem os mesmos objetivos da classe *Arrival Unit* e *target* é a variável nova criada. Chamar os métodos e receber o objeto que será o alvo. Neste caso chamará os métodos do comportamento *Collision Avoidance* e *Arrival Unit* pois todos casos de teste envolvem chegar a dado objetivo e parar nele.

Temos 4 variáveis globais nesta classe, sendo elas:

- `steeringBasics`: variável do tipo *Steering Basics* usada para acessar seus métodos;
- `colAvoid`: variável do tipo *Collision Avoidance* usada para acessar seus métodos;
- `colAvoidSensor`: variável do tipo *NearSensor*, uma classe que possui dois métodos para este sensor, estes métodos serão explicados mais a frente;
- `target`: variável do tipo *Transform* que possibilita que seja escolhido um objeto de jogo como alvo e é possível acessar a posição, rotação e tamanho do objeto.

O método *Start* é um pouco similar ao mesmo método da classe *Arrival Unit* e por isso não será explicada aqui. O método void *Update* é mais importante pois como explicado

anteriormente é chamado a cada quadro do jogo:

Algorithm 10 Método*Update*

```
1: Vector3 accel = colAvoid.getSteering(colAvoidSensor.targets);
2: if accel.magnitude < 0.005f then
3:     accel = steeringBasics.arrive (target.position);
4: end if
5: steeringBasics.steer(accel);
```

A comparação do `if` está verificando se caso a magnitude do vetor foi menor que 0.05f então chamamos o método *arrive* que irá fazer com que o personagem comece a parar e pare na posição do alvo. E a última linha mantém o personagem em direção do nosso alvo a cada quadro do jogo.

5.5.2 *Near Sensor*

Usada como sensor para a detecção de obstáculos, esta classe possui somente uma variável: `targets`. Sendo do tipo *HashSet<Rigidbody>*, ou seja, praticamente um dicionário mas sem o valor para cada chave, é apenas uma coleção de chaves que podemos testar rapidamente se o valor é parte de um conjunto ou não.

Algorithm 11 Método *OnTriggerEnter*

```
1: targets.Add (other.GetComponent<Rigidbody>());
```

Algorithm 12 Método *OnTriggerExit*

```
1: targets.Remove (other.GetComponent<Rigidbody>());
```

Ambos métodos do tipo *void* recebem como parâmetro uma variável do tipo *collider* chamada *other* que possui métodos usados para colisão. O primeiro método é chamado quando o colisor *other* entra na *trigger*. Esta mensagem é mandada à *trigger* do colisor e do *Rigidbody*(se houver) ao qual o *trigger* pertencente do colisor e para o *Rigidbody* ao encostar no *trigger*. Caso ela seja chamada, o corpo rígido deste alvo será adicionado a variável `targets` do tipo *HashSet*.

O segundo método é o oposto do primeiro, ele é chamado quando o colisor *other* parou de encostar no *trigger*. Caso isso aconteça a variável `targets` do tipo *HashSet* remove este corpo rígido que parou de encostar na *trigger*.

5.6 *Wall Avoidance*

Esta é a classe do comportamento *Wall Avoidance* no qual seu objetivo é desviar de paredes. As seguintes variáveis globais são usadas:

- `mainWhiskerLen`: o comprimento do "bigode"principal, ou seja, o quão longe o raio deve estender;
- `wallAvoidDistance`: a distância da colisão que desejamos ir;
- `sideWhiskerLen`: o comprimento do "bigode"do lado;
- `sideWhiskerAngle`: o ângulo do "bigode"do lado;
- `maxAcceleration`: a aceleração máxima do personagem
- `rb`: variável do tipo *rigidbody*, ou seja, corpo rígido que possui propriedades como velocidade, massa, entre outros.
- `steerbingBasics`: variável do tipo *Steering Basics* usada para acessar seus métodos.

Algorithm 13 Método *getSteering*

```
1: return getSteering(rb.velocity);
```

Algorithm 14 Método *getSteering*

```
1: Vector3 acceleration = Vector3.zero;
2: Vector3[] rayDirs = new Vector3[3];
3: rayDirs[0] = facingDir.normalized;
4: float orientation = Mathf.Atan2(rb.velocity.y, rb.velocity.x);
5: rayDirs[1] = orientationToVector(orientation + sideWhiskerAngle * Mathf.Deg2Rad);
6: rayDirs[2] = orientationToVector(orientation - sideWhiskerAngle * Mathf.Deg2Rad);
7: RaycastHit hit;
8: if !findObstacle(rayDirs, out hit) then
9:     return acceleration;
10: end if
11: Vector3 targetPostition = hit.point + hit.normal * wallAvoidDistance;
12: Vector3 cross = Vector3.Cross(rb.velocity, hit.normal);
13: if cross.magnitude < 0.005f then
14:     targetPostition = targetPostition + new Vector3(-hit.normal.y, hit.normal.x, hit.normal.z);
15: end if
16: return steeringBasics.seek(targetPostition, maxAcceleration);
```

Como pode ser visto na página anterior temos um *overloading* do método *getSteering*.O primeiro método não recebe um parâmetro e retorna o vetor `rb.velocity` que será usado na classe *Wall Avoidance Unit* para chamar o segundo método enviando este vetor `rb.velocity`. Já o segundo método tem como parâmetro um vector3D chamado `facingDir` e retorna a posição do alvo e aceleração máxima do personagem através da chamada do método *seek*.

Na primeira linha do segundo método temos a criação do vector3D aceleração com a atribuição de zero em suas três coordenadas. Na linha seguinte criamos o vetor de direção do raio, alocamos espaço para três elementos. O primeiro elemento do vetor de direção do raio recebe o vector3D `facingDir` normalizado.

Na linha 4 tem-se a criação e atribuição da orientação, o método *Math.Atan* retorna o ângulo em radianos cuja a tangente é y/x, nesse caso a coordenada y e x do vetor velocidade. Para entendermos as próximas duas linhas precisamos entender o método *orientationToVector* que recebe a variável `orientation` e retorna a orientação como um vetor unitário:

Algorithm 15 Método *orientationToVector*

```
1: return new Vector3(Mathf.Cos(orientation), Mathf.Sin(orientation), 0);
```

Os métodos *Math.Cos* e *Math.Sin* recebem como parâmetro um ângulo em radianos e retornam respectivamente um valor entre -1 e +1.

Portanto temos na linha 5 o segundo elemento do vetor de direção do raio recebendo o resultado do método discutida acima, no qual é passado como parâmetro o resultado da soma da orientação mais o ângulo do "bigode"do lado(que representa 45 graus) vezes *Mathf.Deg2Rad*. *Mathf.Deg2Rad* sendo a conversão de graus para radianos. Obviamente é preciso calcular também o elemento 3 do vetor de direção do raio, a sua única diferença é que passamos agora a orientação menos o o ângulo do "bigode"do lado(-45) vezes *Mathf.Deg2Rad*. Assim temos os nossos três vetores de direção do raio, principal, 45 graus e -45 graus.

A linha 7 tem-se a criação de uma variável do tipo *RaycastHit*. Esta variável lança um raio, de origem pontual, na direção de comprimento `maxDistance`, contra todos os colisores na cena do jogo. Seu retorno é do tipo *bool*, retornando verdadeiro se o raio intersectar com um colisor, caso contrário falso. A primeira condição deste método testa caso não haja colisão, caso não houver retorna apenas a aceleração e mais nada.

Conforme a linha 11, cria-se um alvo longe da parede para procurarmos. A linha seguinte temos o produto cruzado de dois vetores resultando em um terceiro vetor que é perpen-

dicular aos dois vetores de entrada sendo atribuído a variável `cross`. Entre as linhas 13 a 15 temos uma condição usada para se a velocidade e a colisão normal forem paralelas então movemos o alvo um pouco para a esquerda ou direita da normal.

O último método da classe *Wall Avoidance* é a *findObstacle* que recebe como parâmetro o vector3D dinâmico `rayDirs` e por referência `firstHit` do tipo *RaycastHit* retornando verdadeiro ou falso caso através da variável `foundObs`:

Algorithm 16 Método *findObstacle*

```
 1: firstHit = new RaycastHit();
 2: bool foundObs = false;
 3: for int i = 0; i < rayDirs.Length; i++ do
 4:     float rayDist = (i == 0) ? mainWhiskerLen : sideWhiskerLen;
 5:     RaycastHit hit;
 6:     if Physics.Raycast(transform.position, rayDirs[i], out hit, rayDist) then
 7:         foundObs = true;
 8:         firstHit = hit;
 9:         break;
10:     end if
11: end for
12: return foundObs;
```

Na segunda linha setamos a variável `foudObs` que significa que encontramos um obstáculo para falso. Na linha seguinte temos um laço que percorre todos os elementos do vetor de direção do raio. A variável `rayDist` irá receber a distância de cada bigode conforme a condição ternária dependendo de qual elemento do vetor estiver sendo verificado no laço. Dentro da condição da linha 6 está sendo verificado um método chamado *Physics.Raycast* que retorna verdadeiro se o raio intersecta em algum colisor e falso caso contrário.

Este método tem como parâmetro o ponto inicial do raio em coordenadas mundiais, a direção do raio, uma variável do tipo *hit* que ao ser verdadeira retorna mais informações sobre onde o colisor foi acertado e por último a distância máxima que o raio deve checar por colisões.

Caso a condição seja válida então encontramos um obstáculo, ou seja, `foundObs = true`, `firstHit` recebe `hit` que contém informações importantes sobre onde o colisor foi acertado e saímos da condição. Por último retornamos *foundObs* como verdadeiro.

5.6.1 *Wall Avoidance Unit*

A última classe é usada para chamar os métodos da classe *Wall Avoidance*. Possuindo três variáveis globais sendo que *target* é a variável nova criada:

- `steeringBasics`: variável do tipo *Steering Basics* usada para acessar seus métodos;
- `wallAvoidance`: variável do tipo *Wall Avoidance* usada para acessar seus métidos;
- `target`: variável do tipo *Transform* que possibilita que seja escolhido um objeto de jogo como alvo e é possível acessar a posição, rotação e tamanho do objeto.

Similarmente a classe *Collision Avoidance Unit* o método *Update*:

Algorithm 17 Método *Update*

```
1: Vector3 accel = wallAvoidance.getSteering();
2: if accel.magnitude < 0.005f then
3:     accel = steeringBasics.arrive (target.position);
4: end if
5: steeringBasics.steer(accel);
```

Como podemos notar o objetivo é bastante parecido com o da classe *Collision Avoidance Unit*: chegar ao alvo, porém neste caso estamos evitando colisão com paredes e não obstáculos em movimento. A única diferença está na linha 1, onde estamos chamando o método *getSteering* da classe *wallAvoidance*, porém sem mandar um parâmetro para ela, pois chamamos o método *getSteering* com a assinatura *getSteering()* para enviarmos o vetor velocidade do nosso personagem para o mesmo método porém com a assinatura *getSteering(Vector3 facingDir)*.

5.6.2 TargetSphere

Esta classe é simplesmente uma classe usada para pegar o raio da esfera do alvo. Foi criada para não haver repetição de código nas classes dos comportamentos que a usam. As seguintes variáveis foram usadas:

- `arriveUnit`: variável do tipo *Arrival Unit* usada para acessar seus métodos;
- `wallavoidanceUnit`: variável do tipo *Wall Avoidance Unit* usada para acessar seus métodos;
- `colavoidUnit`:variável do tipo *Collision Avoidance Unit* usada para acessar seus métodos;
- `behavior`: *string* para selecionar qual comportamento está sendo usado;
- `sphereTarget`:variável do tipo *SphereCollider* usada para acessar seus métodos;

- `sphereRadius`: variável do tipo que representa o raio da esfera.

Dentro do método *Start* temos:

Algorithm 18 Método *Start*

```
1:  switch(behavior):
2:  case "ArriveUnit":
3:    arriveUnit = GetComponent<ArriveUnit> ();
4:    sphereTarget = arriveUnit.target.transform.GetComponent<SphereCollider> ();
5:  break;
6:  case "WallAvoidanceUnit":
7:    wallavoidanceUnit = GetComponent<WallAvoidanceUnit> ();
8:    sphereTarget = wallavoidanceUnit.target.transform.GetComponent<SphereCollider> ();
9:  break;
10: case "ColAvoidUnit"
11:   colavoidUnit = GetComponent<ColAvoidUnit> ();
12:   sphereTarget = colavoidUnit.target.transform.GetComponent<SphereCollider> ();
13: break;
14: sphereRadius = sphereTarget.radius;
```

Como podemos ver o *switch* é apenas usado para retornar as variáveis com as classes necessárias de cada comportamento e a esfera do raio do alvo. r

6 SIMULAÇÕES DOS CASOS DE TESTE

As simulações foram dividas em dois tipos. O primeiro tipo foi conforme os dados da tabela 6.1. O outro tipo de simulação foi sobre o cenário 2(Tabela 6.1) com o número de agentes crescente de 5 até 45. Este cenário foi escolhido pois igualmente ao cenário 7 desvia de paredes, agentes e chega no alvo mas também possui mais agentes que o cenário 7. Na seção de apêndices é possível ver como ficaram os cenários para as simulações na *Unity* 5, estes com mais agentes do que os cenários do *steerbench* pois quisemos estressar a simulação, podendo haver diferenças notáveis ou não nas métricas de qualidade e desempenho.

Para as simulações foram estabelecidos as seguintes características para cada cenário:

Cenário	**Número de agentes**	**Comportamento**	**Raio do agente**
1	500	Arrival	0
2	45	Arrival, Wall Avoidance	1
3	20	Arrival, Wall Avoidance	0
4	18	Arrival, Collision Avoidance	1.5
5	12	Arrival, Collision Avoidance	1.5
6	24	Arrival, Collision Avoidance	1.5
7	12	Arrival, Wall Avoidance	1
8	12	Arrival, Wall Avoidance	0
9	16	Arrival, Collision Avoidance	1.5

Tabela 6.1: Número de agentes e comportamentos usados

O número de agentes e quais comportamentos foram escolhidos com base em cada cenário e suas limitações, sejam elas tamanho, obstáculos, etc. O cenário 1 por exemplo tem 500 agentes pois neste cenário o comportamento é muito simples e não existem obstáculos, ou seja, chegar ao alvo. E nos demais cenários existem estas limitações como citadas no ínicio deste parágrafo.

Poucos cenários tiveram os mesmos arranjos dos agentes,portanto, é importante destacar como foram posicionados e a diferença de posição de cada um:

- 1: uma matriz 25x20 - 1m de distância;
- 2: uma matriz 9x5 - 1m de distância;
- 3: uma fila - 1m de distância;
- 4: duas matrizes 3x3 - 1m de distância;
- 5: uma coluna de 6 e uma linha 6 - 1m de distância;
- 6: duas filas de 12 - 1.5m de distância entre as colunas e 1m de distância entre as linhas;
- 7: duas matrizes de 3x2 - 2m de distância entre as colunas e 3m de distância entre as linhas;
- 8: uma coluna de 6 e uma linha 6 - 1m de distância;
- 9: duas matrizes de 2x4 - 2m de distância entre as colunas e 1m de distância entre as linhas;

Nos cenários em que o comportamento do agente tem com objetivo somente chegar no alvo e desviar de paredes, o raio de cada agente é 0 pois não faz uso da classe *NearSensor*, nos demais cenários eles possuem raio 1 ou 1,5 e usam esta classe. Essa diferença de raio esteve particularmente vinculada ao cenário, ou seja, ao tamanho de seus obstáculos ou quantidade de agentes. Os agentes dos cenários 2 e 7 além de desviarem de paredes e chegarem ao seu alvo, também desviam de outros agentes pois possuem anexado a eles a classe *NearSensor*.

O cenário 7 é similar ao cenário 4 porém tem um obstáculo no caminho, o problema deste cenário é que ao colocar um número de agentes igual ao 4 os agentes tiveram um desempenho imprevisível o que ocasionou resultados fora do padrão. Para solucionar isto foi diminuído o número de agentes e aumentado a distância entre os agentes.

Os seguintes dados foram coletados de cada agente e da *Unity* 5 ao final de cada simulação:

- Velocidade média;
- Aceleração média;
- Deslocamento
- Distância percorrida;

- Tempo percorrido;
- FPS Baixo;
- FPS Médio;
- FPS Alto;
- Uso total de memória do sistema(TSMU);

Vale ressaltar que os dados FPS baixo, médio, alto e TSMU foram coletados desde o início da simulação até o final, pois são independentes, ou seja, não é preciso esperar que cada agente chegue no alvo para coletá-los, os mesmos foram coletados a cada quadro da simulação.

7 RESULTADOS OBTIDOS

Nesta seção foi apresentado os resultados obtidos sobre todas as métricas para a primeira simulação comparando com outros cenários e também os resultados obtidos durante a segunda simulação comparando com o mesmo cenário(2) porém com mais agentes. Foi feito uma interpretação baseada nos maiores coeficientes de variação entre os cenários, pois este, é uma medida para definir se um desvio padrão é baixo, médio ou alto.

Cenário	**Velocidade média**	**Aceleração média**	**Deslocamento**	**Distância percorrida**	**Tempo percorrido**
1	2,96	0,67	18,11	18,83	6,95
2	3,7	0,41	9,37	41,51	11,7
3	4,22	0,29	19,54	53,83	13,15
4	4,77	0,81	16,05	20,96	4,42
5	4,87	1,86	7,22	9,26	2,05
6	4,83	1,14	14,57	18,82	4,87
7	4,06	0,34	19,64	47,78	12,16
8	4,67	1,24	8,05	13,82	3,02
9	4,82	0,58	21,29	29,11	6,15

Tabela 7.1: Média de todas métricas de qualidade por cenário

Cenário	**Velocidade média**	**Aceleração média**	**Deslocamento**	**Distância percorrida**	**Tempo percorrido**
1	0,62	0,44	2,1	5,72	3,15
2	0,47	0,24	1,4	20,82	6,52
3	0,65	0,09	5,44	14,42	4,32
4	0,45	0,13	1,32	2,74	0,71
5	1,5	0,59	0,21	1,29	0,73
6	2,22	0,84	5,16	8,1	3,87
7	0,48	0,13	1,49	16,45	5,04
8	0,57	0,31	0,85	2,87	0,81
9	0,56	0,09	1,12	2,97	1,24

Tabela 7.2: Desvio padrão de todas métricas de qualidade por cenário

Segundo a tabela 7.3 os coeficientes de variação das métricas de qualidade por cenário foram:

- Velocidade média: os cenários 1 e 3(Figura B.1) tiveram dispersão média, os cenários 5 e 6(Figura B.2) tiveram dispersão alta, logo seus dados são heterogêneos e os cenários restantes apresentaram dispersão baixa, portanto, seus dados são homogêneos. O motivo dos cenários 5 e 6 terem um desvio padrão alto, é pelo falo de que em ambos cenários os agentes precisam desviar de outros agentes em um ambiente pequeno;

- Aceleração média: somente os cenários 4, 8 e 9(Figuras B.2 e B.3) tiveram dispersão média, os demais cenários apresentaram um desvio padrão alto, logo dispersão alta. Levando em conta o coeficiente de variação do tempo percorrido, podemos notar que os cenário 4, 8 e 9 também apresentaram dispersão média e os demais tiveram dispersão alta, ou seja, o tempo percorrido esteve diretamente associado com a aceleração média;

- Deslocamento: o cenários 3 teve dispersão média, por outro lado o cenário 6 possui uma dispersão alta. A razão desta dispersão alta no cenário 6 é pois os agentes estão enfileirados,logo, a posição inicial de cada agente é diferente. Os cenários restantes tiveram uma dispersão baixa;

- Distância percorrida: os cenário 3 e 8(Figuras B.1 e B.3) foram os únicos cenários com dispersão média, os cenários 4, 5 e 9(Figuras B.2 e B.3) tiveram dispersão baixa enquanto os cenários 1, 2, 6, e 7(Figuras B.1,B.2 e B.3) tiveram dispersão alta. A razão destes últimos cenários tiverem um desvio padrão alto é pelo fato de haver um grande número de agentes e desvio de entre agentes e/ou obstáculos. Os agentes do cenário 1 não possuem a habilidade de desviar de outros agentes e obstáculos, porém como existe muitos agentes nele, resultou em uma pequena fila no momento de chegar no alvo, assim aumentando a distância de vários agentes;

- Tempo percorrido: como dito no item aceleração média, os cenário 4,8 e 9 possuem dispersão média e os demais cenários possuem dispersão alta. Conforme a tabela 7.1 dentre estes demais cenários, os cenários 2, 3 e 7(Figuras B.1 e B.3) obtiveram uma grande diferença entre deslocamento e distância percorrida, assim consequentemente tiveram o maior tempo percorrido para chegarem em seus alvos. Isto se deve ao fato de os agentes demorarem para desviarem dos outros agentes e/ou obstáculos. O cenário 1 apesar de ter

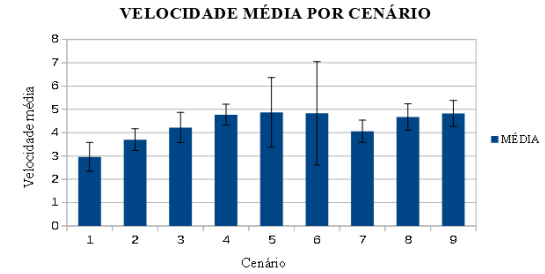

Figura 7.1: Velocidade média

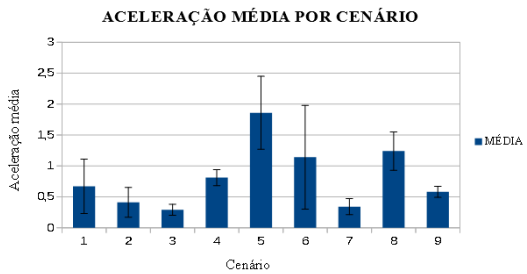

Figura 7.2: Aceleração média

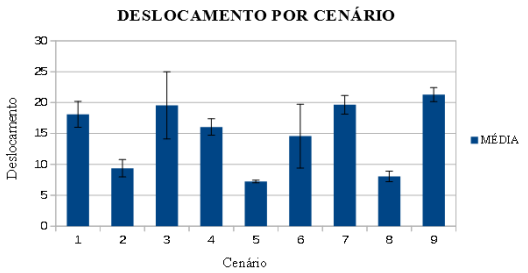

Figura 7.3: Deslocamento

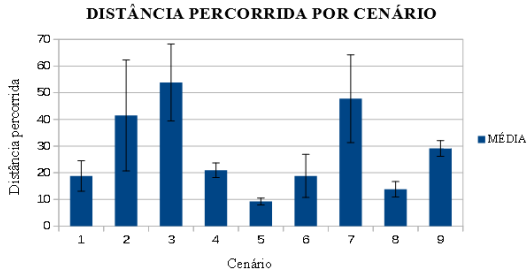

Figura 7.4: Distância percorrida

uma pequena diferença entre deslocamento e distância percorrida, teve um desvio padrão alto no tempo percorrido, a razão disto é que muitos agentes tentaram chegar junto no alvo e então tiveram de esperar que os agentes a sua frente chegassem antes de poder chegar, consequentemente, aumentando o seu tempo de chegada.

Para uma visão mais clara sobre o resultado de cada métrica de qualidade por cenário é possível ver nas figuras: 7.1 até 7.5.

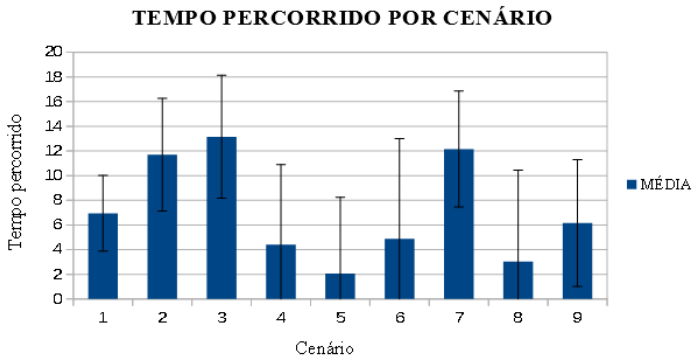

Figura 7.5: Tempo percorrido

Portanto os cenários que obtiveram as maiores distâncias percorridas tiveram os maiores tempos percorridos, visto que a distância percorrida influencia diretamente no tempo percorrido.

Cenário	Velocidade média	Aceleração média	Deslocamento	Distância percorrida	Tempo percorrido
1	20,95%	65,67%	11,6%	30,38%	45,32%
2	12,7%	58,54%	14,94%	50,16%	55,73%
3	15,4%	31,03%	27,84%	26,79%	32,85%
4	9,43%	16,05%	8,22%	13,07%	16,06%
5	30,8%	31,72%	2,91%	13,93%	35,61%
6	45,96%	73,68%	35,42%	43,04%	79,47%
7	11,82%	38,24%	7,59%	34,43%	41,45%
8	12,21%	25%	10,56%	20,77%	26,82%
9	11,62%	15,52%	5,26%	10,2%	20,16%

Tabela 7.3: Coeficiente de variação de todas métricas de qualidade por cenário

Cenário	FPS Baixo	FPS Médio	FPS Alto	TSMU
1	23,47	38,03	73,82	0,9
2	23,08	45,17	101,1	0,79
3	21,14	52,8	117,27	0,78
4	20,74	44,95	130,54	0,74
5	20,32	44,98	134,75	0,81
6	17,72	45,11	116,1	0,73
7	23,2	53,84	138,56	0,76
8	18,75	44,96	150,47	0,8
9	21,66	45,13	127,42	0,79

Tabela 7.4: Média de todas métricas de desempenho por cenário

Cenário	FPS Baixo	FPS Médio	FPS Alto	TSMU
1	3,06	6,33	9,34	0,01
2	4,56	8,89	4,74	0,01
3	4,99	7,44	20,79	0
4	6,48	8,85	22,19	0
5	6,2	9,13	27,17	0,01
6	8,12	9,05	13,14	0,01
7	4,71	5,38	30,95	0,04
8	7,43	8,92	25,06	0
9	5,14	8,93	24,78	0

Tabela 7.5: Desvio padrão de todas métricas de desempenho por cenário

Conforme a tabela 7.6 de coeficiente de variação de todas métricas de desempenho por cenário:

- FPS baixo: somente o cenário 1(Figura B.1) teve dispersão baixa, os cenários 2, 3, 7 e 9(Figuras B.1,B.2 e B.3) tiveram dispersão média e os cenários 4, 5, 6, e 8(Figuras B.2 e B.3) tiveram dispersão alta;

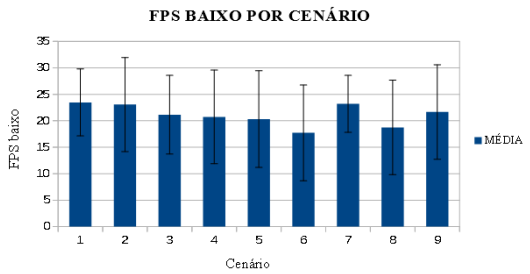

Figura 7.6: FPS baixo

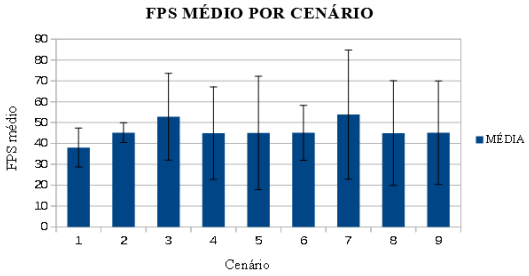

Figura 7.7: FPS médio

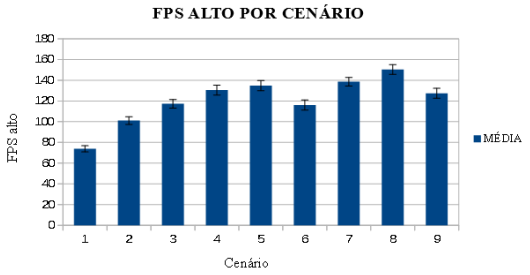

Figura 7.8: FPS alto

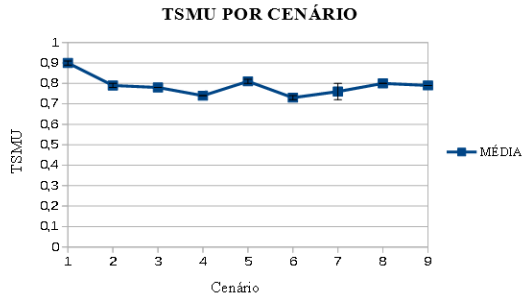

Figura 7.9: TSMU

- FPS médio: nenhum cenário obteve dispersão alta, os cenários 3 e 7 tiveram dispersão baixa e os demais cenários tiveram dispersão média;
- FPS alto: nenhum cenário obteve dispersão alta, os cenários 1, 2 e 6 tiveram dispersão baixa enquanto os demais cenários tiveram dispersão média;
- TSMU: apesar de todos os cenários terem tido dispersão baixa, somente os cenários 3,4,8 e 9 tiveram dispersão nula , ou seja, 0%.

As figuras 7.6 a 7.9 demonstram as variações das métricas de desempenho entre todos os cenários. Assim é possível concluir que o FPS alto foi o que teve a menor discrepância de valores entre os tipos de FPS e que o TSMU teve um desvio padrão baixíssimo em todos cenários.

Na figura 7.10 podemos ver as oscilações entre as métricas de qualidade aceleração média, deslocamento, distância percorrida e tempo percorrido. Como também podemos ver como em alguns cenários algumas métricas tiveram resultados próximos, como no cenário 2 onde o deslocamento e tempo percorrido. No cenário 5 a aceleração média e o tempo percorrido estiveram próximos e o deslocamento e distância percorrida também estiveram próximos. Essa última observação pode ser usada também para o cenário 8.

Cenário	FPS Baixo	FPS Médio	FPS Alto	TSMU
1	13,04%	16,64%	12,65%	1,11%
2	19,76%	19,68%	4,69%	1,27%
3	23,49%	14,09%	17,74%	0%
4	31,24%	19,69%	17%	0%
5	30,51%	20,3%	20,16%	1,23%
6	45,82%	20,06%	11,32%	1,37%
7	20,3%	9,99%	22,34%	5,26%
8	39,63%	20,02%	16,65%	0%
9	23,73%	19,79%	19,45%	0%

Tabela 7.6: Coeficiente de variação de todas métricas de desempenho por cenário

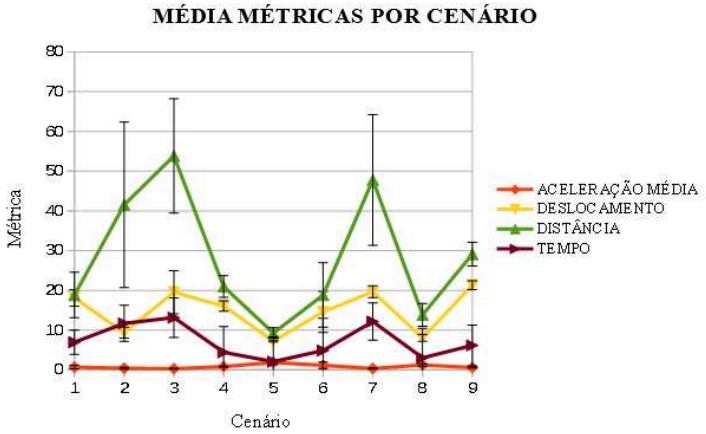

Figura 7.10: Média para todos cenários

Número de agentes	Velocidade média	Aceleração média	Deslocamento	Distância percorrida	Tempo percorrido
5	4,13	0,49	9,86	32,9	8,08
10	3,95	0,41	9,68	38,7	10,01
15	3,84	0,4	9,53	38,61	10,25
20	3,72	0,37	9,41	44,29	12,23
25	3,77	0,44	9,33	37,32	10,26
30	3,73	0,43	9,28	38,7	10,78
35	3,68	0,41	9,28	40,23	11,32
40	3,7	0,42	9,3	38	10,62
45	3,7	0,41	9,37	41,51	11,7

Tabela 7.7: Média de todas métricas de qualidade do cenário 2

Segundo a tabela 7.11 os coeficientes de variação das métricas de qualidade por número de agentes foram do cenário 2:

- Velocidade média: indiferente do número de agentes do cenário 2, a dispersão foi baixa, logo, os dados foram homogêneos;

Número de agentes	Velocidade média	Aceleração média	Deslocamento	Distância percorrida	Tempo percorrido
5	0,28	0,16	1,29	10,48	2,86
10	0,27	0,16	1,33	14,53	4,25
15	0,27	0,14	1,36	13,94	4,17
20	0,27	0,19	1,39	20,78	6,34
25	0,44	0,23	1,4	15,85	4,9
30	0,45	0,23	1,41	19,33	6,01
35	0,43	0,23	1,4	19,99	6,21
40	0,43	0,22	1,4	16,40	5,09
45	0,47	0,24	1,49	20,82	6,52

Tabela 7.8: Desvio padrão de todas métricas de qualidade do cenário 2

- Aceleração média: indiferente do número de agentes do cenário 2, a dispersão foi alta, logo, os dados foram heterogêneos. Como relatado anteriormente o tempo percorrido está diretamente influenciando a aceleração média, é possível ver logo mais pelo resultado do coeficiente de variação da métrica tempo percorrido.

- Deslocamento: as simulações com 5,10,15 e 20 agentes tiveram dispersões baixas, ou seja, os dados foram homogêneos. Os demais cenários tiveram dispersão média;

- Distância percorrida: indiferente do número de agentes do cenário 2, a dispersão foi alta, portanto, os dados são heterogêneos. Este resultado é obra dos agentes terem que desviar de outros agentes e de um obstáculo, assim, levando em conta suas trajetórias para chegarem no alvo;

- Tempo percorrido: indiferente do número de agentes do cenário 2, a dispersão foi alta. A razão de o desvio padrão desta métrica ser alto é pois em todas simulações o desvio padrão da distância percorrida foi alto e consequentemente o tempo que levou para chegar no alvo também foi alto, caso a diferença entre deslocamento e distância percorrida tivesse sido baixa, o desvio padrão desta métrica seria baixo.

Conforme a tabela 7.12 os coeficientes de variação das métricas de desempenho por número de agentes do cenário 2 foram:

- FPS baixo: indiferente do número de agentes no cenário, a dispersão foi média;

- FPS médio: as simulações de número de agentes 15, 20, 25, 30 e 35 tiveram dispersão baixa, portanto, estes dados são homogêneos. O restante das simulações tiveram dispersão média;

Número de agentes	FPS Baixo	FPS Médio	FPS Alto	TSMU
5	22,73	45,02	149,27	0,78
10	18,89	48,13	95,37	0,78
15	20,82	54,06	83,69	0,78
20	23,73	54,16	81,44	0,78
25	19,85	53,32	79,24	0,78
30	20,72	55,72	78,82	0,78
35	22,65	57,87	74,27	0,8
40	21,08	51,2	89,57	0,79
45	23,08	45,17	101,1	0,79

Tabela 7.9: Média de todas métricas de desempenho do cenário 2

Número de agentes	FPS Baixo	FPS Médio	FPS Alto	TSMU
5	5,8	8,92	19,78	0
10	5,95	10,29	27,71	0
15	5,68	6,6	4,47	0
20	4,76	5,51	4,52	0
25	5,04	7,69	3,52	0
30	5,26	7,58	3,3	0
35	5,21	3,65	4,67	0
40	5,14	9,42	3,73	0,01
45	4,56	8,89	4,74	0,01

Tabela 7.10: Desvio padrão de todas métricas de desempenho do cenário 2

- FPS alto: somente a simulação com número de agentes igual a 10 teve dispersão média, o restante teve dispersão baixa, logo, seus dados foram homogêneos;
- TSMU: todas simulações tiveram dispersão baixa, somente os cenários com 40 e 45 agentes não tiveram dispersão nula(0%).

Nas figuras 7.11 e 7.12 podemos ver como ficou todas as métricas e desvios padrões para o cenário 2 como também a média e desvio padrão do TSMU.

Sendo assim somente uma simulação de FPS alto retornou uma dispersão que não fosse baixa, concluindo assim nesta métrica que a maior parte das simulações teve dados homogêneos, ou seja, a menor discrepância entre os tipos de FPS. Por fim a dispersão do TSMU foi novamente baixa, comprovando que os dados desta métrica pouco mudou entre uma simulação e outra.

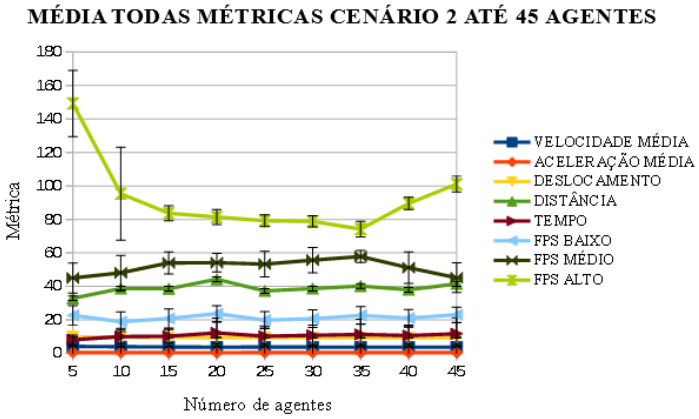

Figura 7.11: Média todas métricas cenário 2 até 45 agentes

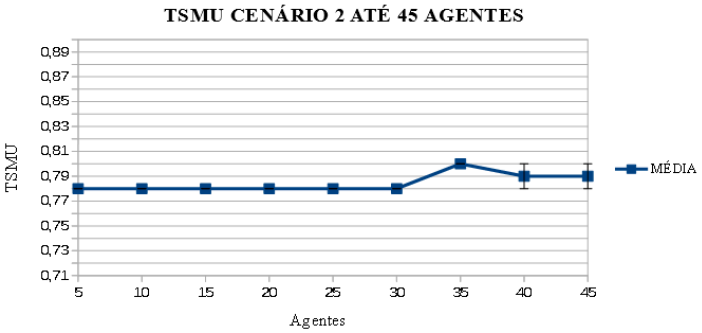

Figura 7.12: TSMU cenário 2 até 45 agentes

Número de agentes	Velocidade média	Aceleração média	Deslocamento	Distância percorrida	Tempo percorrido
5	6,78%	32,65%	13,08%	31,85%	35,4%
10	6,84%	39,02%	13,74%	37,55%	42,46%
15	7,03%	35%	14,27%	36,1%	40,68%
20	7,26%	51,35%	14,77%	46,92%	51,84%
25	11,67%	52,27%	15,01%	42,47%	47,76%
30	12,06%	53,49%	15,19%	49,95%	55,75%
35	11,68%	56,1%	15,09%	49,69%	54,86%
40	11,62%	52,38%	15,05%	43,16%	47,93%
45	12,7%	58,54%	15,9%	50,16%	55,73%

Tabela 7.11: Coeficiente de variação de todas métricas de qualidade do cenário 2

Número de agentes	FPS Baixo	FPS Médio	FPS Alto	TSMU
5	25,52%	19,81%	13,25%	0%
10	31,5%	21,38%	29,06%	0%
15	27,28%	12,21%	5,34%	0%
20	20,06%	10,17%	5,55%	0%
25	25,39%	14,42%	4,44%	0%
30	25,39%	13,6%	4,19%	0%
35	23%	6,31%	6,29%	0%
40	24,38%	18,4%	4,16%	1,27%
45	19,76%	19,68%	4,69%	1,27%

Tabela 7.12: Coeficiente de variação de todas métricas de desempenho do cenário 2

8 CONSIDERAÇÕES FINAIS

Neste trabalho foi apresentado uma abordagem que teve como objetivo evidenciar informações importantes sobre a qualidade de *steering behaviours* no motor de jogo *Unity* 5 e o desempenho deste motor de jogo. Para que os objetivos deste trabalho fossem alcançados, a metodologia de pesquisa estruturada foi:

Etapa 1: Definição dos objetivos específicos;

Etapa 2: Abordagem da metodologia;

Etapa 3: Implementação;

Etapa 4: Definição dos cenários das simulações;

Etapa 5: Análise dos resultados obtidos;

Na etapa 1 foram definidos os seguintes objetivos específicos: (a) "abordar os diferentes tipos de *steering behaviors* que foram implementados na *Unity 5*: sendo eles *Arrival*, *Collision Avoidance*, *Wall Avoidance*", (b) "modificar estes comportamentos de direção no motor de jogo *Unity* 5 em C# para rodar em simulações dos cenários do benchmark *SteerBench* adaptados para a *Unity* 5", (c) "aplicar métricas e casos de teste para avaliação de cada *steering behavior* na Unity 5 e da performance da mesma, fazendo uma análise dos resultados, para que então possa ser levantado informações importantes sobre a qualidade dos *steering behaviors* na Unity 5 e a performance da Unity 5."

Para entender e atingir objetivo (a) foi realizado a leitura de trabalhos relacionados a personagens autônomos e *steering behaviors*. Para atingir o objetivo (b) foi necessário conhecer o motor de jogo usado, conhecer mais sobre a linguagem C# e definir as simulações. O objetivo (c) foi necessário para definir métricas e métodos de coletar e analisar os dados obtidos. O objetivo (a) colaborou na construção do referencial teórico deste trabalho. O objetivo (b) colaborou na etapa de implementação, modificação dos comportamentos e adaptação dos cenários para as simulações. O objetivo (c) colaborou com as etapas de metodologia, coleta e análise dos resultados obtidos.

Na etapa 2 foram definidos a abordagem de metodologia, as ferramentas usadas, as métricas de qualidade e desempenho e por fim os métodos estatísticos de avaliação dos dados coletados. As seguintes métricas de qualidade serviram para avaliar a qualidade dos *steering behaviors*: velocidade média, aceleração média, deslocamento, distância percorrida e tempo percorrido. Por fim foi definido as métricas de desempenho com o objetivo de avaliar o desem-

penho do motor do jogo: FPS baixo, FPS médio, FPS alto e TSMU.

Na etapa 3 foi dado uma breve introdução sobre a interface da *Unity* e como é o seu funcionamento. Após isto foi explicado os diferentes algoritmos implementados e modificados e seus métodos mais importantes.

Na etapa 4 foi definido os diferentes cenários, quantidade de agentes por cenário, distribução dos agentes e quais comportamentos seriam usados. Após isso foi realizado as simulações e coletado os dados para a próxima etapa: análise dos resultados obtidos.

A etapa final foi o momento de analisar os resultados obtidos para cada cenário e cada métrica. A partir de métodos estatísticos foi possível avaliar e extrair informações importantes sobre cada comportamento e sobre o desempenho do motor de jogo escolhido. Conforme explicado na seção Resultados Obtidos, a partir de cenários com obstáculos e/ou um número considerável de agentes foi possível distinguir diferenças notáveis nas métricas de qualidade resultantes de comportamentos diferentes, número de agentes e diferentes cenários.

Para as métricas de desempenho foi evidente que o coeficiente de variação do TSMU(Tabelas 7.6 e 7.12) foi baixo, portanto, a diferença entre as médias para os dois tipos de simulações(Tabelas 7.4 e 7.9) também pouco variaram, independente do número de agentes,comportamento deles e objetos de jogo. Porém a análise dos resultados não foi conclusiva a ponto de termos uma resposta definitiva sobre quais fatores são mais considerados para o TSMU. Esta análise levou a acreditar parcialmente que TSMU é sensível para número de objetos de jogo(agentes, obstáculos, etc), número de agentes e seus comportamentos.

Por outro lado, ao levarmos em conta discrepância dos FPS altos(Tabela 7.6 e 7.12), pois foram o que tiveram os desvios padrões mais baixos em todos os simulações, é possível notar que o FPS é mais sensível ao número de agentes e qual comportamento estes agentes possuem, se for um número baixo de agentes e um comportamento simples o FPS será provavelmente maior, caso contrário, menor.

Por fim, com este trabalho foi possível visualizar e debater sobre os diferentes resultados entre as métricas de qualidade devido aos seus comportamentos, cenários e número de agentes e afirmar que apesar das variações entre os tipos de FPS e TSMU, a *Unity* 5 é uma ótima opção de motor de jogo para usarmos *steering behaviors* em nossos jogos ou outros tipos de aplicações que são possíveis com eles.

8.1 Trabalhos futuros

Neste trabalho foi apresentado uma abordagem de como foi implementado e modificado *steering behaviors* para rodar em cenários de um benchmark na *Unity* 5, uma análise destes comportamentos e também do desempenho deste motor de jogo. Portanto é importante que este trabalho sirva de inspiração para outros trabalhos relacionados sobre este tema, sendo assim recomenda-se para trabalhos futuros:

- Abordagem de outros comportamentos mais complexos, como: *Flocking, Path Following, Leader Following, Queue*;

- Aplicar métricas de qualidade específicas para estes comportamentos;

- Usar cenários mais interativos e complexos, até mesmo com artefatos 2D relevantes que possam ser considerados para métricas de desempenho gráfico(GPU).

REFERÊNCIAS

[1] H. B. Amor, J. Murray, O. Obst, et al. Fast, neat, and under control: Arbitrating between steering behaviors. *AI Game Programming Wisdom*, 3:221–232, 2006.

[2] F. Bevilacqua. Understanding steering behaviors. https://goo.gl/4kUuRq, 2012.

[3] F. Bevilacqua. Understanding steering behaviors: Collision avoidance. https://goo.gl/ePGHkg, 2012.

[4] F. Bevilacqua. Understanding steering behaviors: Flee and arrival. https://goo.gl/N1d4by, 2012.

[5] F. Bevilacqua. Understanding steering behaviors: Seek. https://goo.gl/5y1rSM, 2012.

[6] M. Buckland. *Programming game AI by example*. Jones & Bartlett Learning, 2005.

[7] T. DEALS. This engine is dominating the gaming industry right now. https://tnw.to/e4mGG, 2016.

[8] S. Egenfeldt-Nielsen, J. H. Smith, and S. P. Tosca. *Understanding video games: The essential introduction*. Routledge, 2015.

[9] A. Egges, A. Kamphuis, and M. Overmars. *Motion in Games: First International Workshop, MIG 2008, Utrecht, The Netherlands, June 14-17, 2008, Revised Papers*, volume 5277. Springer, 2008.

[10] M. Enger. Game engines: How do they work? https://goo.gl/ZosPRq, 2013.

[11] A. M. L. C. d. Feijoo. A pesquisa e a estatística na psicologia e na educação. 2010.

[12] M. d. A. Marconi and E. M. Lakatos. *Fundamentos de metodologia científica*. 5. ed.-São Paulo: Atlas, 2003.

[13] E. McDonald. The global games market will reach $108.9 billion in 2017 with mobile taking 42%. https://goo.gl/f4TSMl, 2017.

[14] A. Nareyek. Ai in computer games. *Queue*, 1(10):58, 2004.

[15] A. Pantev. Unity movement ai. http://antonpantev.com/, 2015.

[16] P. S. Paul, S. Goon, and A. Bhattacharya. History and comparative study of modern game engines. *International Journal of Advanced Computed and Mathematical Sciences*, 3(2):245–249, 2012.

[17] C. W. Reynolds. Steering behaviors for autonomous characters. In *Game developers conference*, volume 1999, pages 763–782, 1999.

[18] D. Shiffman, S. Fry, and Z. Marsh. *The nature of code*. D. Shiffman, 2012.

[19] R. Silveira, F. Dapper, E. Prestes, and L. Nedel. Natural steering behaviors for virtual pedestrians. *The Visual Computer*, 26(9):1183–1199, 2010.

[20] S. Singh, M. Kapadia, P. Faloutsos, and G. Reinman. Steerbench: a benchmark suite for evaluating steering behaviors. *Computer Animation and Virtual Worlds*, 20(5-6):533–548, 2009.

[21] U. Technologies. Unity store. https://goo.gl/v6Fi3R, 2017.

[22] Unity. Diagnosing performance problems using the profiler window. goo.gl/LNFZAv, 2018.

[23] Unity. Unity para jogos móveis. https://unity3d.com/pt/mobile/solution-guide, 2018.

[24] J. Wainer et al. Métodos de pesquisa quantitativa e qualitativa para a ciência da computação. *Atualização em informática*, 1:221–262, 2007.

[25] M. H. Zaharia, F. Leon, C. Pal, G. Pagu, N. Baykara, and N. Mastorakis. Agent-based simulation of crowd evacuation behavior. In *WSEAS International Conference. Proceedings. Mathematics and Computers in Science and Engineering*, number 11. World Scientific and Engineering Academy and Society, 2009.

[26] A. Zakharov. Used total vs total system memory usage. https://goo.gl/EEyvVJ, 2013.

APÊNDICES

APÊNDICE A – Classe SteeringBasics

```
using UnityEngine;
using System.Collections;

/* Uma classe auxiliar para direcionar um objeto de jogo em 2D*/
using System.Collections.Generic;


[RequireComponent (typeof (Rigidbody))]
public class SteeringBasics : MonoBehaviour {

        public float maxVelocity = 3.5f;

        /* A aceleracao maxima */
        public float maxAcceleration = 10f;

        /* O raio do alvo que significa que estamos proximos o
           suficiente e chegamos*/

        /* O raio do alvo onde comecamos a desacelerar */
        public float slowRadius = 1f;

        /* O tempo em que queremos alcancar a velocidade alvo */
        public float timeToTarget = 0.1f;

        public float turnSpeed = 20f;

        private Rigidbody rb;

        private TargetSphere targetSphere;

        private NearSensor nearSensor;

        public bool col;

        public bool smoothing = true;
        public int numSamplesForSmoothing = 5;
        private Queue<Vector2> velocitySamples = new
           Queue<Vector2>();

        // Usado para inicialiazacao
        void Start () {
                rb = GetComponent<Rigidbody> ();
                targetSphere = GetComponent<TargetSphere>();
                if(col == true)

                nearSensor =
                   transform.Find("ColAvoidSensor").GetComponent<NearSensor>()
```

```
}

/* Atualiza a velocidade atual do objeto de jogo ao dar a
   aceleracao linear */
public void steer(Vector3 linearAcceleration) {
        rb.velocity += linearAcceleration * Time.deltaTime;

        if (rb.velocity.magnitude > maxVelocity) {
                rb.velocity = rb.velocity.normalized *
                   maxVelocity;
        }
}

public void steer(Vector2 linearAcceleration) {
        this.steer (new Vector3 (linearAcceleration.x,
           linearAcceleration.y, 0));
}

/* O comportamento de direcao seek. Ira retornar a direcao
   para o objeto de jogo atual para procurar uma direcao
   dada */
public Vector3 seek(Vector3 targetPosition, float
   maxSeekAccel) {
        //Pega a direcao
        Vector3 acceleration = targetPosition -
           transform.position;

        //Removemos a coordenada z
        acceleration.z = 0;

        acceleration.Normalize ();

        //Aceleramos ate o alvo
        acceleration *= maxSeekAccel;

        return acceleration;
}

public Vector3 seek(Vector3 targetPosition)
{
        return seek(targetPosition, maxAcceleration);
}

/* Faz com que objeto de jogo atual olhe para onde eles esta
   indo */
public void lookWhereYoureGoing() {
        Vector2 direction = rb.velocity;

        if (smoothing) {
                if (velocitySamples.Count ==
                   numSamplesForSmoothing) {
```

```
                velocitySamples.Dequeue ();
            }

            velocitySamples.Enqueue (rb.velocity);

            direction = Vector2.zero;

            foreach (Vector2 v in velocitySamples) {
                direction += v;
            }

            direction /= velocitySamples.Count;
        }

        lookAtDirection (direction);
    }

    public void lookAtDirection(Vector2 direction) {
        direction.Normalize();

        // Se temos uma direcao diferente de 0 entao olhe
           para essa direcao, caso contrario, nao faca nada
        if (direction.sqrMagnitude > 0.001f) {
            float toRotation = (Mathf.Atan2
                (direction.y, direction.x) *
                Mathf.Rad2Deg);
            float rotation =
                Mathf.LerpAngle(transform.rotation.eulerAngles.z,
                toRotation, Time.deltaTime*turnSpeed);

            transform.rotation = Quaternion.Euler(0, 0,
                rotation);
        }
    }

    public void lookAtDirection(Quaternion toRotation)
    {
        lookAtDirection(toRotation.eulerAngles.z);
    }

    public void lookAtDirection(float toRotation)
    {
        float rotation =
            Mathf.LerpAngle(transform.rotation.eulerAngles.z,
            toRotation, Time.deltaTime * turnSpeed);

        transform.rotation = Quaternion.Euler(0, 0,
            rotation);
    }
```

```
/* Retornamos a direcao para um personagem para que entao
   chegue no alvo*/
public Vector3 arrive(Vector3 targetPosition) {
        /* Pegamos a direcao correta para a aceleracao
           linear */
        Vector3 targetVelocity = targetPosition -
           transform.position;
        // Removemos a coordenada z
        targetVelocity.z = 0;

        /* Pega a distancia ate o alvo */
        float dist = targetVelocity.magnitude;
        /* Se estamos dentro da area de raio entao paramos */

        if (!col) {
                if (dist <= targetSphere.sphereRadius) {
                        OnTriggerEnterTarget
                           (targetSphere.sphereTarget);
                        return Vector3.zero;
                }
        } else if(col) {
                if (dist <= targetSphere.sphereRadius +
                   nearSensor.actorSphere.radius) {
                        OnTriggerEnterTarget
                           (targetSphere.sphereTarget);
                        return Vector3.zero;
                }
        }

        /* Calculamos a velocidade alvo, velocidade maxima
           na distancia ate slowRadius e 0 de velocidade na
           distancia igual a 0*/
        float targetSpeed;
        if(dist > slowRadius) {
                targetSpeed = maxVelocity;
        } else {
                targetSpeed = maxVelocity * (dist /
                   slowRadius);
        }

        /* Damos a targetVelocity a velocidade correta */
        targetVelocity.Normalize();
        targetVelocity *= targetSpeed;

        /* Calculamos a aceleracao linear que queremos */
        Vector3 acceleration = targetVelocity - new
           Vector3(rb.velocity.x, rb.velocity.y, 0);
        /*
         Ao inves de acelerarmos o personagem ate a
            velocidade correta em 1 segundo,
```

```
                aceleramos para que entao alcancamos a velocidade
                    desejada em timeToTarget segundos
                */
                acceleration *= 1/timeToTarget;

                /* Garantindo que estamos acelerando a aceleracao
                   maxima */
                if(acceleration.magnitude > maxAcceleration) {
                        acceleration.Normalize();
                        acceleration *= maxAcceleration;
                }

                return acceleration;
        }

        void OnTriggerEnterTarget(Collider other){
                if (GetComponent<Collider> ().GetType () ==
                   typeof(SphereCollider)){
                        rb.velocity = Vector3.zero;
                        this.transform.localScale = new Vector3 (0,
                           0, 0);
                }
        }

        /* Verificamos para ver se o alvo esta na frente do
           personagem */
        public bool isInFront(Vector3 target)
        {
                return isFacing(target, 0);
        }

        public bool isFacing(Vector3 target, float cosineValue) {
                Vector2 facing = transform.right.normalized;

                Vector2 directionToTarget = (target -
                   transform.position);
                directionToTarget.Normalize();

                return Vector2.Dot(facing, directionToTarget) >=
                   cosineValue;
        }

        public static float getBoundingRadius(Transform t)
        {
                SphereCollider col =
                   t.GetComponent<SphereCollider>();
                return Mathf.Max(t.localScale.x, t.localScale.y,
                   t.localScale.z) * col.radius;
        }
}
```

Listing A.1: C# SteeringBasics

APÊNDICE B – Cenários das simulações

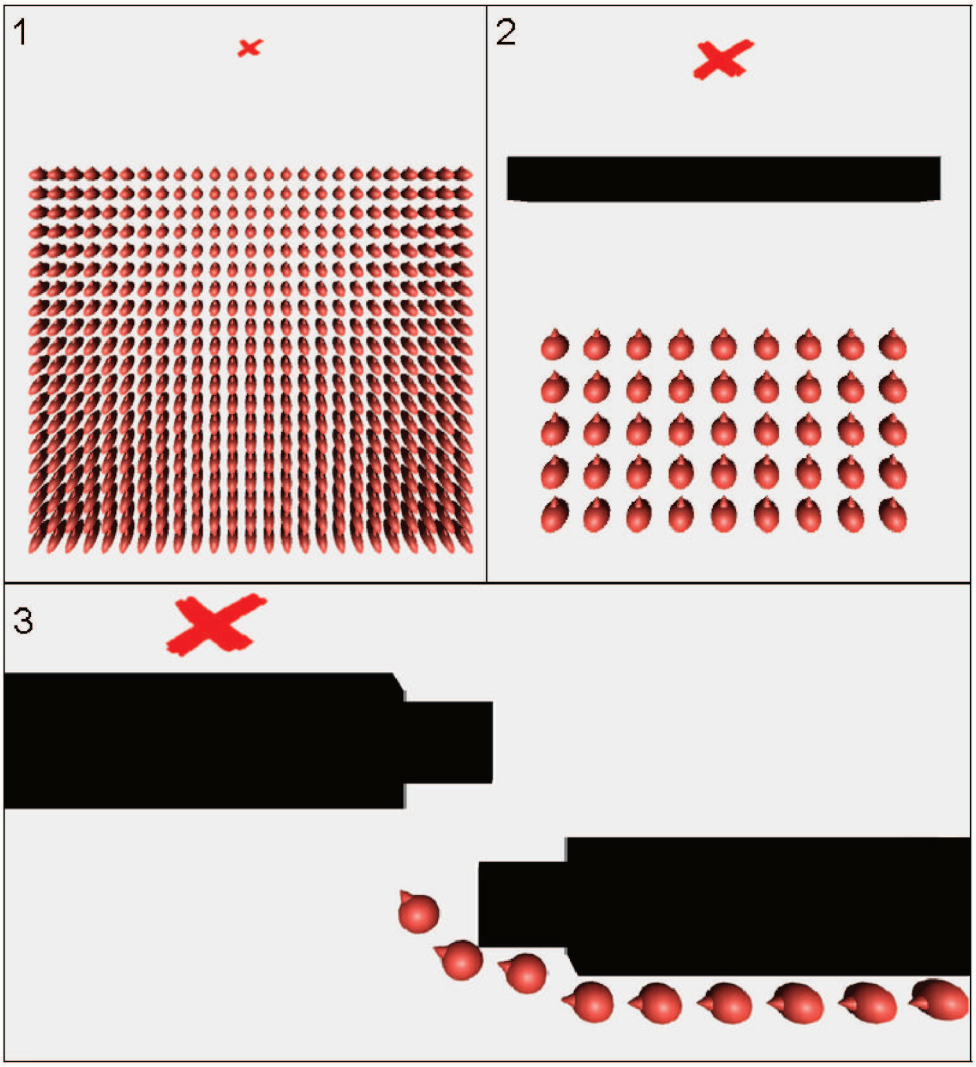

Figura B.1: Cenários 1 a 3

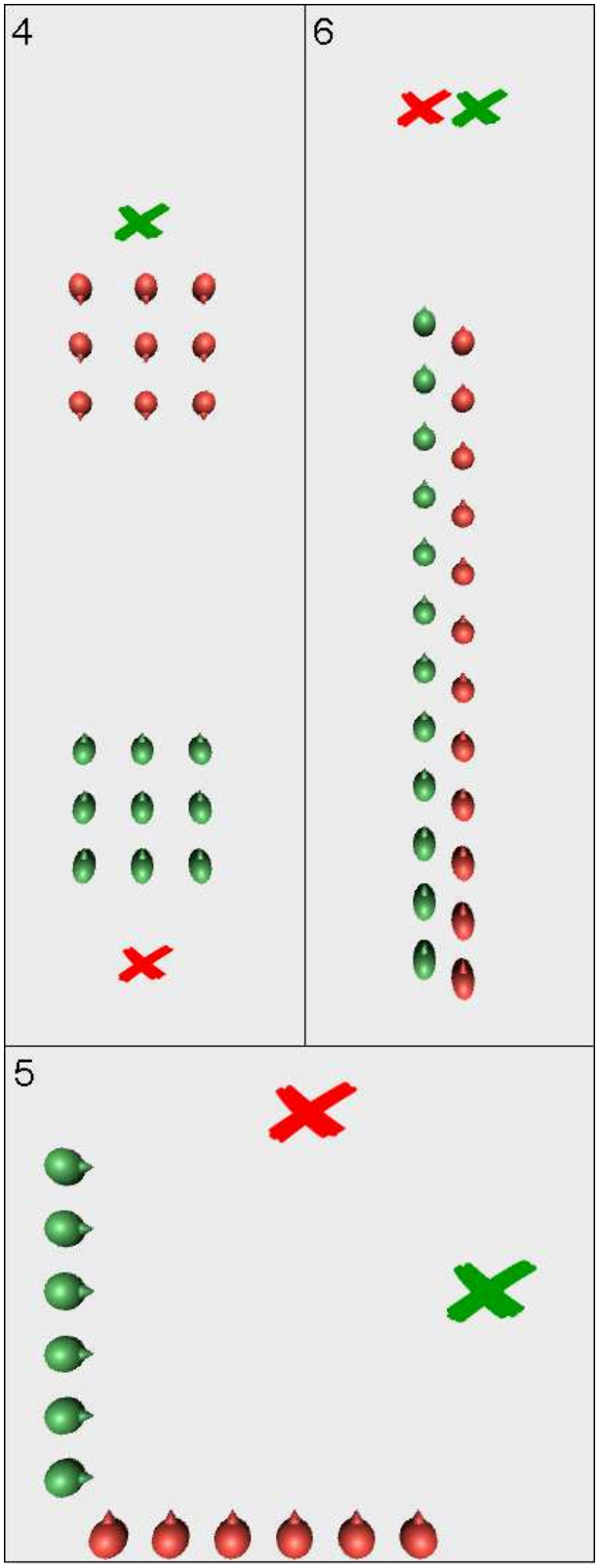

Figura B.2: Cenários 4 a 6

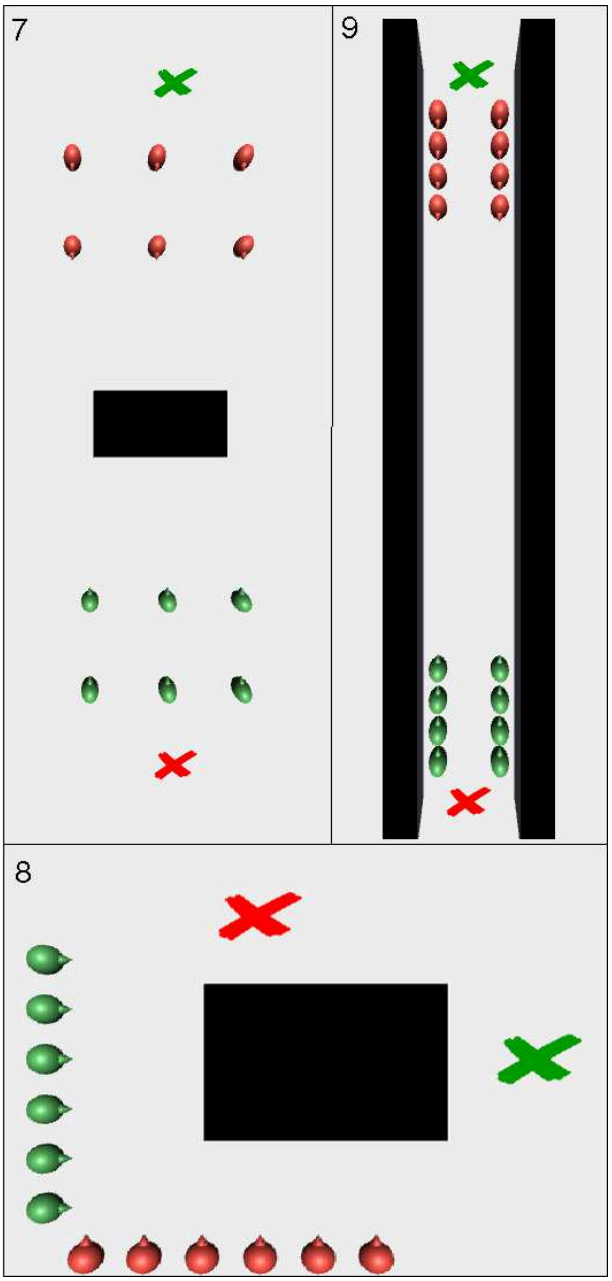

Figura B.3: Cenários 7 a 9