UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

PAULO ARTUR MORGANTI JUNIOR

FERRAMENTA ANTITRAPAÇA PARA JOGOS DIGITAIS MULTIJOGADOR DESENVOLVIDOS NA UNITY COM INTEGRAÇÃO DO ML-AGENT

BENTO GONÇALVES

2023

PAULO ARTUR MORGANTI JUNIOR

FERRAMENTA ANTITRAPAÇA PARA JOGOS DIGITAIS MULTIJOGADOR DESENVOLVIDOS NA UNITY COM INTEGRAÇÃO DO ML-AGENT

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de graduação de Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Profa. Dra. Iraci Cristina da Silveira De Carli

BENTO GONÇALVES

2023

PAULO ARTUR MORGANTI JUNIOR

FERRAMENTA ANTITRAPAÇA PARA JOGOS DIGITAIS MULTIJOGADOR DESENVOLVIDOS NA UNITY COM INTEGRAÇÃO DO ML-AGENT

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de graduação de Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Aprovado em 06/12/2023

BANCA EXAMINADORA

Profa. Dra. Iraci Cristina da Silveira De Carli
Universidade de Caxias do Sul - UCS

Profa. Dra. Elisa Boff
Universidade de Caxias do Sul - UCS

Prof. Dr. Marcelo Luís Fardo
Universidade de Caxias do Sul - UCS

AGRADECIMENTOS

Agradeço a todos que me ajudaram nessa caminhada em especial minha família que nunca deixou de acreditar no meu potencial. Minha mãe que está sempre do meu lado e meu pai que está orgulhoso lá do céu. Aos meus amigos que sempre me apoiaram e aos professores que me ajudaram nessa longa jornada. Um agradecimento especial à professora Iraci que aceitou a minha ideia. Só tenho um frase: meu muito obrigado!

“Recupera-se o perdido,
Rompe-se a dura prisão
E no auge do furacão
Cede o mar embravecido.”
Oração de Santo Antônio

RESUMO

A trapaça em jogos digitais online *multiplayer* é um dos grandes problemas dessa indústria e proporciona uma experiência desagradável para jogadores honestos. Nesse trabalho realizou-se um estudo bibliométrico na base de dados *Web of Science* para determinar como o meio acadêmico busca a solução para problemas envolvendo trapaças. Buscou-se os trabalhos em dois momentos. No primeiro, utilizou-se os termos-chave *game* and (cheat* or security)* com período de cinco de anos e em segundo momento com termos de busca *online game* and cheat**. Também avaliou-se trabalhos que utilizam a ferramenta Unity no desenvolvimento de jogos. Com essa análise, observou-se a utilização de aprendizado por reforço para identificação de possíveis jogadores mal intencionados. É proposto um protótipo utilizando a *Engine Unity* para criar um ambiente em que se possa utilizar a biblioteca *ML-Agents* e validar através de inteligência artificial se um jogador é humano ou um *bot*. A implementação do protótipo foi desenvolvida na *Unity* na linguagem C#. Descreve-se o algoritmo implementado com a maioria das classes criadas. Após, detalha-se as etapas de integração da biblioteca *ML-Agents* com *Engine Unity* e é realizada a análise dos resultados obtidos.

Palavras-chave: Unity. Jogos digitais online. Taxonomia das trapaças. ML-Agents

ABSTRACT

Cheating in online multiplayer digital games is one of the biggest problems and provides an unpleasant experience for honest players. This paper presents the categorization of these cheats according to the taxonomy of Yan and Rendell (YAN; RANDELL, 2005). In addition, a bibliometric study was carried out in the database *Web of Science* to determine how academia solves these problems. The works were sought in two moments. In the first, the key terms were used *game* and (cheat* or security)* with a period of five years and in the second moment with search terms *online game* and cheat**. The paper also evaluated studies that used the Unity tool in the game development. With this analysis, it was observed the use of reinforced learning to identify possible malicious players. It is proposed a prototype using the Unity Engine is proposed to create an environment in which one can use the ML-Agents library and validate through artificial intelligence whether a player is human or a bot. The prototype implementation was developed in Unity with the C# language. The algorithm created with most of the created classes. Afterwards, the integration steps of the ML-Agents library with Engine Unity are detailed and it is executed the analysis about the results.

Keywords: Unity. Online digital games. Taxonomy of cheating. ML-Agents

LISTA DE FIGURAS

Figura 1 – Modelo da conexão LAN 20
Figura 2 – Classificação dos jogos online 26
Figura 3 – Produção científica anual da primeira busca 34
Figura 4 – Produção científica anual na busca específica para jogos online 34
Figura 5 – Autores com maior número de produções 35
Figura 6 – Mapa de calor dos autores filtrados pelo método Proknow-C 35
Figura 7 – Visão de coautoria 36
Figura 8 – Mapa do jogo Caçador de Recompensa 43
Figura 9 – Conceito inicial do jogo sem materias e texturas 43
Figura 10 – Diagrama de casos de uso 45
Figura 11 – Diagrama de componentes 46
Figura 12 – Diagrama de classes para projeto 46
Figura 13 – Diagrama de Estados 48
Figura 14 – Diagrama de Sequência para o caso de uso Atacar 49
Figura 15 – Diagrama de Sequencia para o caso de uso Marcar Ponto 50
Figura 16 – Diagrama de Sequencia para o caso de Avaliar e Recompensa 50
Figura 17 – Exemplo de arquivo YAML 51
Figura 18 – Tela Inicial da Engine da Unity 53
Figura 19 – Lista de Prefabs utilizadas na implementação 54
Figura 20 – *Prefab* do jogador 55
Figura 21 – Tela de Reinicio do jogo 55
Figura 22 – Tela inicial do Jogo com opção de *Host* e *Client* 56
Figura 23 – Netcode no gerenciador de pacotes da *Unity* 57
Figura 24 – Adição do componente *NetworkManager* no objeto 57
Figura 25 – Configuração do Player Prefab no componente *NetworkManager* 58
Figura 26 – Lista *Network Prfabs List* 59
Figura 27 – Lista de Scripts gerados para o protótipo 59
Figura 28 – Prompt de comando do Windows com comandos para criar ambiente virtual Python 72
Figura 29 – Comando de ajuda de MLAgent 72
Figura 30 – Prompt de comando no treinamento 77
Figura 31 – Gráfico da recompensa adquirida pelos agents 78
Figura 32 – Componente com o cérebro treinado 80

LISTA DE TABELAS

Tabela 1 – Resultado do Teste 1, um jogador humano versus um jogador bot 80
Tabela 2 – Resultado obtido no artigo 80
Tabela 3 – Resultado do Teste 2, dois jogadores humanos 81

LISTA DE ALGORITMOS

Algoritmo 1 Criação das variáveis da classe Jogador 60
Algoritmo 2 Métodos de inicialização da classe Jogador 60
Algoritmo 3 Métodos do comportamento do jogador 62
Algoritmo 4 Métodos de atualização do servidor no multiplayer 63
Algoritmo 5 Métodos de colisão da classe Jogador 64
Algoritmo 6 Variáveis inicializadas da classe Bot 65
Algoritmo 7 Métodos de inicialização da classe Bot 65
Algoritmo 8 Métodos para adicionar inimigos na lista 66
Algoritmo 9 Métodos para atualizar posição do bot 66
Algoritmo 10 Métodos da classe do inimigo 67
Algoritmo 11 Métodos da classe auxiliar FollowTransform 68
Algoritmo 12 Variáveis da classe GameController 68
Algoritmo 13 Método principal e de criação do jogador e bots na classe GameController 69
Algoritmo 14 Métodos para reiniciar cena na classe LevelController 69
Algoritmo 15 Métodos para inicializar componentes de rede 70
Algoritmo 16 Classe responsável pelo agente de aprendizado por reforço 73
Algoritmo 17 Parâmetros de treinamento no formato YAML 77

LISTA DE ABREVIATURAS E SIGLAS

AR *Augmented Reality*

AHCI *Arts and Humanities Citation Index*

API *Application Programming Interface*

CNN Convolutional neural network

CPU *Central Processing Unit*

CS-GO *Counter Strike - Global Offensive*

ESL *Electronic Sports League*

FPS *First Person Shooter*

GDD *Game Design Document*

GPU *Graphics Processing Unit*

IA Inteligência Artificial

IDE *integrated development environment*

IEEE *Institute of Electrical and Electronics Engineers*

ISI *Institute for Scientific Information*

LAN *Local Area Network*

ML-Agents *Machine Learning Agents Toolkit*

MMORPG *Massively Multiplayer Online Role-Playing Game*

NPC *Non-Player Character*

POCA *Parallel Online Continuous Arcing*

PPO *Proximal Policy Optimization*

PROKNOW-C *Knowlegde Development Constructivist*

RPC *Remote Procedure Call*

SAC *Soft Actor Critic*

SCI-EXPANDED *Science Citation Index Expanded*

SDK *Software Development Kit*

SSCI *Social Sciences Citation Index*

UML *Unified Modeling Language*

VR *Virtual Reality*

YAML *Ain't Markup Language*

SUMÁRIO

1 INTRODUÇÃO . . . 14
1.1 OBJETIVOS . . . 15
1.2 METODOLOGIA . . . 16
1.3 ESTRUTURA DO TRABALHO . . . 17

2 JOGOS DIGITAIS E TRAPAÇAS . . . 19
2.1 JOGOS DIGITAIS . . . 19
2.1.1 Jogos Digitais *Multiplayer* . . . 20
2.1.1.1 *Multiplayer* Local . . . 20
2.1.1.2 *Multiplayer Online* . . . 20
2.2 Trapaças em jogos Digitais . . . 21
2.2.1 Formas comuns de trapaça . . . 21
2.2.1.1 Trapaça de exploração da confiança no lado errado . . . 22
2.2.1.2 Trapaça por conluio . . . 22
2.2.1.3 Trapaça de abuso dos procedimentos do jogo . . . 22
2.2.1.4 Trapaças Relacionadas a Ativos Virtuais . . . 23
2.2.1.5 Trapaça na exploração de inteligência de máquina . . . 23
2.2.1.6 Trapaça de modificação da infraestrutura do cliente . . . 23
2.2.1.7 Trapaça de negação de serviço a jogadores adversários . . . 23
2.2.1.8 Trapaça de timing . . . 23
2.2.1.9 Trapaça de comprometimento de senhas . . . 24
2.2.1.10 Trapaça de exploração na falta de sigilo . . . 24
2.2.1.11 Trapaça de exploração da falta de autenticação . . . 24
2.2.1.12 Trapaça de exploração de bug ou brecha . . . 24
2.2.1.13 Trapaça de comprometimento dos servidores . . . 24
2.2.1.14 Trapaça relacionada ao uso interno indevido . . . 25
2.2.1.15 Trapaça por Engenharia Social . . . 25
2.2.2 A taxonomia . . . 25
2.2.2.1 Natureza das trapaças . . . 25
2.2.2.2 Por vulnerabilidade . . . 26
2.2.2.3 Consequência das trapaças . . . 27
2.2.2.4 Por ator da falha . . . 28
2.3 Considerações finais sobre a Taxonomia . . . 28

3 BIBLIOMETRIA SOBRE TRAPAÇAS EM JOGOS DIGITAIS . . . 30
3.1 Definições do estudo bibliométrico . . . 30

3.1.1 **Método Proknow-C** . . . **30**
3.1.1.1 Pesquisando em Bancos . . . 30
3.1.1.2 Selecionando Referencial Teórico . . . 31
3.2 Bibliometria aplicada sobre trapaças em jogos digitais . . . 31
3.2.1 **Escolha da base de dados** . . . **31**
3.3 Resultado da Análise Bibliométrica . . . 33
3.4 Considerações sobre a análise bibliométrica . . . 36

4 **TÉCNICAS DE IMPLEMENTAÇÃO ANTITRAPAÇA** . . . **37**
4.1 Implementações antitrapaça na bibliografia . . . 37
4.1.1 **Manipulação da infraestrutura do cliente** . . . **37**
4.1.2 **Utilizando Inteligência de máquina** . . . **38**
4.1.3 **Técnicas para trapaças genéricas** . . . **38**
4.2 Ferramenta antitrapaça na *Unity* . . . 38
4.2.1 ***Unity com ML-Agents*** . . . **39**
4.3 Considerações das técnicas antitrapaça . . . 40

5 **CAÇADOR DE RECOMPENSA - PROTÓTIPO DE JOGO *MULTI-PLAYER* 3D PARA DETECÇÃO DE TRAPAÇA** . . . **41**
5.1 Design do jogo . . . 41
5.1.1 **Jogabilidade** . . . **42**
5.1.2 **Mecânicas** . . . **42**
5.1.3 **Regras do Jogo** . . . **44**
5.2 Modelagem do jogo e processo de desenvolvimento do software . . . 44
5.2.1 **Diagrama de Casos de Uso** . . . **44**
5.2.2 **Diagrama de Componente** . . . **45**
5.2.3 **Diagrama de Classes** . . . **47**
5.2.4 **Diagrama de Estados** . . . **47**
5.2.5 **Diagrama de Sequência** . . . **48**
5.2.5.1 Diagrama de Sequência para Atacar . . . 48
5.2.5.2 Diagrama de Sequência para Marcar Ponto . . . 48
5.2.5.3 Diagrama de Sequência para Avaliar e Recompensa . . . 49
5.3 Implementação do Agente e integração com o jogo . . . 49
5.3.1 **Configuração de treinamento** . . . **51**
5.3.2 **Decisões do Agente** . . . **51**
5.4 Conclusão sobre modelagem do protótipo . . . 52

6 **IMPLEMENTAÇÃO DO PROTÓTIPO** . . . **53**
6.1 Praparando o ambiente . . . 53
6.1.1 **Elementos do jogo** . . . **54**

6.1.2 ***Multiplayer*** **56**
6.2 Codificação das Classes e Scripts 58
6.2.1 **Jogador** **58**
6.2.2 **Bot Anti-trapaça** **65**
6.2.3 **Enemy** **67**
6.2.4 **FollowTransform** **68**
6.2.5 **GameController** **68**
6.2.6 **LevelController** **69**
6.2.7 **NetworkController** **70**
6.3 Considerações da implementação 70

7 **ML-AGENT NO JOGO** **71**
7.1 Integrando o Agente 71
7.1.1 **Instalação de pacotes necessários** **71**
7.1.2 **Treinamento do Agente** **74**
7.1.3 **Etapa de testes** **79**
7.1.4 **Análise dos Resultados** **79**
7.1.5 **Discussão** **81**
7.1.6 **Conclusão sobre ML Agente no protótipo** **82**

8 **CONCLUSÃO** **83**
8.1 Sugestões de trabalhos Futuros 84

REFERÊNCIAS **85**

1 INTRODUÇÃO

O conceito de jogos ultrapassa séculos e identifica a vida comportamental de uma época. Culturas antigas tinham atividades relacionadas a jogos em seu cotidiano, dentro desse contexto, pode-se citar o xadrez, gamão e Pattoli, jogo do antigo império Azteca (ARIK; GEZER; TAYALI, 2022). Após John von Neumann definir a base da computação que se conhece hoje, surgiu uma nova frente de entretenimento: jogos digitais. Eles se diferenciam de muitas maneiras em relação a jogos considerados não digitais. A característica mais marcante é de que, o jogo digital pode suportar e calcular regras relacionadas à conduta do jogador, adicionando complexidade e automatização nesse processo, enquanto recriam um cenário artificial que produz uma pontuação mensurável(ARIK; GEZER; TAYALI, 2022).

Quando a Atari [1] em 1972 lançou o Pong, uns dos primeiros jogos comerciais de sucesso, não antevia certos aspectos do desenvolvimento que hoje são bastante importantes. Com o passar do tempo, jogos digitais se tornaram cada vez mais complexos até chegar atualmente em ambientes competitivos profissionais. A trapaça nesses jogos é um problema que vem crescendo rapidamente no cenário de esporte eletrônico (*e-Sports*). Aplicações (*Softwares*) de trapaças são facilmente adquiridos por jogadores casuais em site que oferecem esses programas por preços baixos ou gratuitamente (JONNALAGADDA *et al.*, 2021).

De acordo com as plataformas especializadas, o setor de games movimentou, em 2022, aproximadamente US $196,8 bilhões de dólares (ROCHA, 2022). O surgimento do cenário competitivo contribuiu para o aumento do mercado. O início dos anos 2000 trouxe novidades para os jogos digitais multijogador (*multiplayer*) online com jogos competitivos como *StarCraft*. Esse evento marcou o que atualmente é conhecido como *e-Sport* com campeonatos e transmissões para o público (JIN, 2020). Embora os organizadores dos campeonatos tenham a responsabilidade de zelar pela integridade de suas competições, em 2018 ocorreu um caso em evento organizado pela *Electronic Sports League* (ESL). Um jogador profissional da equipe Optic Gaming de *Counter Strike - Global Offensive* (CS-GO) levantou suspeitas dos organizadores e foi pego trapaceando em uma competição chinesa (DAUNTON, 2023). Nesse contexto é importante definir o que vem a ser trapaça. No livro *Security Issues in Online Games*, J. Yan e H.J. Choi definem trapaça como qualquer comportamento que um jogador usa para obter vantagem sobre outros jogadores em que as regras do jogo não permitem tal vantagem (YAN; CHOI, 2002).

A trapaça em jogos digitais não se trata apenas de vantagens adquiridas em jogos casuais mas também de prejudicar a indústria como um todo. A maior parte do faturamento gerado por esses jogos é através de patrocínios com cerca de 59,12%, direitos de uso em 17,76% e 11% da taxa dos publicadores do jogo. Companhias como Adidas, BMW e Mastercard podem desistir de apoiar o setor, e com isso retirar o investimento às empresas a qualquer indício de escândalo

[1] https://atari.com/

relacionado as trapaças, significando uma perda estimada em US$ 650 milhões de dólares para o segmento. (DIAZ, 2023)

Atualmente o desenvolvimento de jogos digitais é raramente realizado a partir do zero, mas implementados por meio de ferramentas conhecidas como motor (*Engine*) de jogos. Esses programas fornecem conjuntos de funcionalidades para equipe de desenvolvimento, seja programadores, artistas ou designers (DILLON, 2015). Um exemplo dessas ferramentas é o software *Unity Engine* [2], motor gráfico bastante difundido com jogos conhecidos pela comunidade de jogos digitais, como *HearthStone* desenvolvido e publicado pela empresa *Blizzard* [3] e *Pillars of Eternity* desenvolvido pela empresa *Obsidian Entertainment* [4] e publicado pela *Paradox Interactive* [5]. A *Unity* conta com completa documentação com exemplos para toda sua *integrated development environment* (IDE)(Ambiente de Desenvolvimento Integrado). Além disso há uma ativa comunidade online de desenvolvedores que fornecem auxílio para novos usuários (CRAIGHEAD; BURKE; MURPHY, 2007).

Apesar da popularidade da *Unity*, o software não possui componentes internos para detecção ou prevenção de trapaças. Devido a esse fato, a comunidade tem contribuído para fornecer soluções para esse problema. Um desses componentes é o *Anti-Cheat ToolKit* [6], o qual promete funcionalidades de proteção em jogos digitais implementados na *Unity*. Outro componente desenvolvido pela comunidade da *Unity* é a biblioteca *Machine Learning Agents Toolkit* (ML-Agents)(Ferramenta de Agentes para Aprendizado de Máquina) [7]. Essa biblioteca faz uso de Inteligência Artificial (IA) para implementar agentes com aprendizado por reforço (*reinforcement learning*) inseridos em um ambiente. Com essa ferramenta é possível utilizar códigos (scripts) de aprendizado por reforço em objetos dentro do jogo e avaliar comportamentos anômalos às regras do jogo (JULIANI *et al.*, 2020).

1.1 OBJETIVOS

O trabalho visa, como objetivo geral, realizar um estudo a respeito de trapaças em jogos digitais *multiplayer* online, analisando trabalhos acadêmicos acerca dessa temática, bem como a implementação de componentes de *software* que podem ser agregados a *Unity* para proteção em jogos digitais *multiplayer* contra trapaças.

O estudo busca elencar as diversas formas de trapaça existentes atualmente e apresentar possibilidades de soluções para prevenir esse problema. Para demonstrar uma possível alternativa revisada pela pesquisa, é proposta a implementação de um protótipo de jogo *multiplayer* utilizando o motor gráfico *Unity*, bem como componentes analisados através do estudo. Após

[2] https://unity.com/pt
[3] https://www.blizzard.com/pt-br/
[4] https://www.obsidian.net/
[5] https://www.paradoxinteractive.com/
[6] https://www.codestage.net/uas/actk/
[7] https://github.com/Unity-Technologies/ml-agents

o desenvolvimento, segue fase de teste da solução e posterior análise da viabilidade da implementação para jogos digitais construídos na *Unity*. Como objetivos específicos, propõe-se:

1. Apresentar e classificar as trapaças dentro de uma taxonomia.
2. Identificar o estado da arte em pesquisas sobre o objeto trapaças em jogos digitais online e sistemas antitrapaças.
3. Através da pesquisa bibliométrica, verificar técnicas e/ou componentes a fim de evitar trapaças em jogos digitais desenvolvidos na *Unity Engine*.
4. Analisar o componente *ML-Agent* através da implementação de protótipo de jogo *multiplayer* na *Unity*.

1.2 METODOLOGIA

O trabalho tem como finalidade uma busca exploratória a respeito do problema, bem como uma pesquisa qualitativa acerca de trapaças em jogos digitais online. No decorrer da elaboração do trabalho, a primeira etapa é a definição de jogos digitais e suas características. Também nessa etapa, são apresentadas as principais formas de trapaça segundo Yan e Choi (YAN; RANDELL, 2005) e sua taxonomia em relação às vulnerabilidades, às consequências e ao ator que está trapaceando.

Na próxima etapa, houve a definição da terminologia/vocabulário utilizado nas pesquisas que envolvem trapaças em jogos digitais. Nessa etapa, para adquirir maior número de produções, optou-se por ter dois conjuntos de termos para ampliar a base de dados. A análise de trabalhos acadêmicos e periódicos tornaram-se os principais procedimentos para coleta dos dados e posterior depuração desses. Métodos bibliométricos permitem aos pesquisadores agregar pesquisas bibliográficas produzidas por outros cientistas. O uso da análise bibliométrica suplementa a avaliação subjetiva das revisões literárias. Estabelecer indicadores confiáveis e de qualidade são pressupostos importantes para análises bibliométricas que influenciam nos processos de recuperação e tratamento de dados e informações (ARIK; GEZER; TAYALI, 2022).

Na terceira etapa do trabalho, com conhecimento prévio, é realizada uma análise bibliométrica para compreender o estado da arte da pesquisa nesta área e identificar técnicas e ferramentas utilizadas para prevenção ou detecção de trapaças. Essa etapa está subdividida em fases. A primeira fase envolve análise bibliométrica na identificação do estado da arte, autores, citações entre outras características. Já na segunda fase dessa etapa, aprofundou-se a pesquisa tendo a *Engine Unity* como termo principal, assim como seus componentes e técnicas de implementação em jogos digitais.

A quarta etapa segue com análises de ferramentas que fornecem técnicas para evitar trapaças em jogos digitais online vistas na pesquisa bibliométrica.

Na quinta etapa do trabalho, tem-se a aplicação da técnica vista na bibliometria com a implementação de um jogo virtual *multiplayer* utilizando motor gráfico. O protótipo do jogo será desenvolvido na *Engine* alvo dessa pesquisa: *Unity*. A modelagem seguirá padrões estabelecidos pela *Unified Modeling Language* (UML) auxiliando na descrição de softwares, principalmente daqueles construídos no estilo de orientação a objetos (FOWLER, 2005).

Na última etapa do projeto, destaca-se a integração do *ML-Agent* no protótipo proposto. Nessa fase, há toda a explicação de configuração, bem como os parâmetros utilizados para realizar o treinamento do agente com finalidade de reconhecer um jogador trapaceiro.

1.3 ESTRUTURA DO TRABALHO

O trabalho está divido da seguinte maneira:

- O Capítulo 2 apresenta definições de jogos digitais bem como as maneiras mais comuns de interação entre jogadores. Também explica conceitos de trapaça e como essas são exploradas nos jogos. Dentro dessa temática, é realizada a categorização dos diferentes tipos de trapaça.

- No Capítulo 3, tem-se a pesquisa bibliométrica. Nessa seção, discute-se como é realizada uma pesquisa bibliométrica, e é apresentado o método *Knowlegde Development Constructivist* (PROKNOW-C) (Desenvolvimento de Conhecimento Construtivista). Ainda no capítulo, é demonstrado a bibliometria a respeito do tema geral, destacando a base de dados, fatores de pesquisa e os resultados desses estudos.

- O Capítulo 4 aborda as técnicas coletadas a partir da pesquisa bibliométrica. Nele é identificado trabalhos que tratam de soluções antitrapaças e a introdução da ferramenta *ML-Agent*, componente para *Unity* que utiliza IA.

- O Capítulo 5 destaca a prototipação do jogo implementado da *Unity Engine*. Nesse capítulo é discutido o design e também as partes principais de um jogo: como é jogado (*gameplay*), as mecânicas envolvidas e as regras que jogo terá. Além disso, o capítulo segue na apresentação dos diagramas seguindo a UML(Linguagem de Modelagem Unificada) e o comportamento da biblioteca *ML-Agent*.

- O Capítulo 6 demostra a implementação do protótipo. Nele, tem-se a criação do cenário de jogo e todos os elementos para as interações entre jogadores. Também é descrito os algoritmos das classes implementadas com as devidas explicações da lógica utilizada.

- No Capítulo 7 é destacada a integração do *ML-Agent* no protótipo. Para utilização da biblioteca é necessário alguns passos para configuração do ambiente. Também é comentado alguns percalços ocorridos durante a utilização desse componente e os métodos para

contorná-los. Ainda nesse capítulo, tem-se o treinamento do agente, os testes do protótipo e a análise dos resultados para validação da solução.

- O Capítulo 8 relata a conclusão sobre o trabalho com a solução para detecção de trapaças em jogos digitais online. Também aponta trabalhos futuros sobre o tema.

2 JOGOS DIGITAIS E TRAPAÇAS

O estudo sobre jogos digitais é relativamente novo. O ano de 2001 é considerado o "ano um"dos estudos acerca de jogos digitais, com o lançamento do primeiro jornal revisado por pares dedicados a jogos digitais (AARSETH, 2001). De forma geral, é consenso nos estudos que jogos digitais são artefatos baseados em regras no qual, um ou mais jogadores se empenham para atingir um objetivo (DORMANS, 2012). Compreendendo jogos digitais, sendo de console ou computadores, a possibilidade de jogar de maneira *online* rompeu uma barreira física entre os jogadores. Antes desse período, a experiência desses jogos se dava através de jogador único (*single-player*) contra o computador ou *multiplayer* local. Por outro lado, jogos digitais *online* permitiram jogadores de várias partes do planeta competirem ou cooperarem entre si (CHEN *et al.*, 2020). Nesse contexto, alguns jogadores mal intencionados ganham vantagens indevidas sobre outros jogadores utilizando *plugins* ou outras formas ilegais cuja atitude traz grande prejuízo tanto para a saúde do jogo quanto para a diversão dos participantes (JIANRONG *et al.*, 2004).

2.1 JOGOS DIGITAIS

As definições sobre o conceito de jogos digitais difere dependendo do observador, entretanto a partir de oito visões distintas, Katie Salen e Eric Zimmerman conseguiram definir jogo como:

> Um jogo é um sistema em que os jogadores se envolvem em conflitos artificiais, definido por regras, que resulta em um resultado quantificável. (SALEN; ZIMMERMAN, 2004)

Essas noções apontadas pela definição fazem parte central do jogo digital. Todos os jogos são *sistemas* de muitas partes e que formam um todo complexo (SALEN; ZIMMERMAN, 2004). As *regras* determinam o quê o jogador pode ou não fazer dentro do sistema e seguindo essas regras, os *jogadores* se envolvem em *conflitos* entre si ou contra o sistema do jogo. A *artificialidade* da definição diz respeito ao distanciamento da vida real. E finalmente, o conceito de *resultado quantificável* é medido através da performance do jogador por algum tipo de placar, em que os jogadores podem ganhar ou perder (DORMANS, 2012).

As formas de interação com os jogos digitais se diversificaram com o passar dos anos, desde jogos de fliperama até jogos utilizando *Virtual Reality* (VR)(Realidade Virtual). A maneira tradicional desses jogos é aquela em que, um jogador tenta vencer o adversário controlado pelo sistema com o objetivo de superar a pontuação de outro jogador ou a própria pontuação da máquina. Essa forma de entretenimento é comumente chamados de *single player*, onde o jogador não interage diretamente com outro oponente humano mas apenas com as rotinas que o computador determina (SKAAR, 2010).

2.1.1 Jogos Digitais *Multiplayer*

Um jogo digital classificado como jogo de jogador único tem a principal característica de ser jogado apenas por um jogador através de toda a sua sessão (OOSTERHU; FEIREISS, 2006). Entretanto, os jogos mais populares atualmente são do tipo *(*multiplayer). Conceitualmente, esse tipo de jogo referencia todos aqueles que proporcionam que mais de uma pessoa possa jogar no mesmo ambiente e ao mesmo tempo sendo compartilhado o sistema ou rede localmente ou em diferentes sistemas através da rede de internet (SKAAR, 2010). Assim, no mercado atualmente, existem duas formas mais conhecidas de jogos *(*multiplayer) : local e *online*.

2.1.1.1 *Multiplayer* Local

Jogos *multiplayer* local implica que os jogadores estão jogando no mesmo sistema ou rede. Esse tipo se encontra nas mais diversas formas, como fliperama, console ou computador. Nos fliperamas, tem-se exemplos de famosos jogos de luta, como *Street Figher*, onde dois (ou mais) jogadores usam a mesma tela e têm a mesma visão dos acontecimentos. Já nos consoles, além dessa descrita nos fliperamas, há tipos de jogos *multiplayer* com a tela dividida, em que cada jogador tem sua área de ação apenas naquela visão. Em computadores, a forma bastante utilizada para jogos *multiplayer* é em redes *Local Area Network* (LAN) (Área Local de Rede). Obviamente nesse caso, os dispositivos dos jogadores devem estar conectados localmente na rede a fim de conseguirem jogar juntos como mostrado na Figura 1. Jogos em rede LAN retiram alguns problemas encontrados em jogos *online* como *lag* (termo usada para determinar atraso de uma ação de determinado jogador e a reação do jogo) e o anonimato (ROUSE, 2011).

Figura 1 – Modelo da conexão LAN

Fonte: O Autor (2023).

2.1.1.2 *Multiplayer Online*

Pode-se admitir vários tipos de jogos *multiplayer online*. Entretanto, os mais comuns são do gênero *First Person Shooter* (FPS), que podem possuir cerca de aproximadamente 64

(sessenta e quatro) jogadores no mundo do jogo simultaneamente e os *Massively Multiplayer Online Role-Playing Game* (MMORPG) (RPG Massivo Multijogador Online).

A maneira mais comum da conexão entre os jogadores dessa modalidade *multiplayer* é a internet. Os benefícios concretos do jogo *online* é a facilidade de interação entre os jogadores quando está envolvido uma grande distância entre eles. Entretanto, há alguns desafios a serem superados quando se está jogando dessa forma, sendo um deles o atraso (*lag*) que dependendo da conexão do jogador pode acarretar uma experiência bastante frustrante para os participantes do jogo. Como o referido *lag*, pode haver, também, problemas de perda de pacotes onde em jogos do gênero FPS, por exemplo, o servidor do jogo não registra o dano que o jogador causa no adversário mesmo que para aquele, os disparos devessem ferir seu oponente (LEE; CHANG, 2015).

2.2 TRAPAÇAS EM JOGOS DIGITAIS

Como já mencionado, jogos *multiplayer online* carregam milhares de jogadores em suas salas de entretenimento diariamente. Levando esse cenário em conta, faz-se necessário a discussão sobre os problemas de segurança que esses jogos possam possuir. A seção explora a literatura atual acerca desses comportamentos e apresenta uma possível classificação de tipos de trapaças segundo algumas categorias. Segundo Brian Rendell e Jeff Yan, no decorrer do contínuo estudo sobre trapaças, foram identificadas e classificadas as trapaças em 15 categorias (YAN; RANDELL, 2005).

2.2.1 Formas comuns de trapaça

Para entender as trapaças e as contra-medidas que podem ser implementadas para mitigar o efeito danoso delas, é importante uma forma de classificação do objeto de estudo. Em 2001, Pritchard introduziu em seu trabalho seis categorias diferentes (PRITCHARD, 2000).

a) Aumento de reflexo: utilização de programas de computador para alterar a reação humana para alcançar melhores resultados
b) Clientes autorizados: exploração de clientes comprometidos para enviar comandos modificados para os outros clientes que os aceitarão
c) Exposição de informação: prejudicar o acesso ou visibilidade para esconder informação, comprometendo o software do cliente
d) Servidores comprometidos: modificar configurações do servidor para ganhar vantagens
e) Falhas e brechas de design: aproveitar de falhas dentro do software
f) Fraqueza do ambiente: exploração de hardware particular ou condições operacionais

Entretanto, segundo Yan e Choi, (YAN; CHOI, 2002), essas definições foram criadas *ad hoc* e não proporcionam um olhar criterioso porque muitas trapaças não se encaixam nessas categorias. Baseado nisso, Rendell e Yan realizaram um estudo de categorização aprofundado e chegaram em um total de 15 categorias diferentes (YAN; RANDELL, 2005).

2.2.1.1 Trapaça de exploração da confiança no lado errado

Um jogador mal intencionado pode modificar o programa do cliente, dados ou ambos e substituir as cópias antigas por arquivos alterados para uso futuro. Esse comportamento é definido como trapaça de exploração da confiança no lado errado (*Exploiting misplaced trust*) e envolvem adulteração dos códigos do jogo e, em muitos casos, é realizado no momento da execução, acessando estados sensíveis que de outra maneira estariam indisponíveis para os jogadores. Típicos exemplos dessa trapaça incluem programas *maphack* que mostram o mapa em jogos de estratégia em tempo real como *Age of Empires*, onde o trapaceador consegue enxergar o mapa inexplorado. Essa forma de trapaça é devido aos desenvolvedores tenderem a confiar exageradamente no lado cliente do processo, onde o controle é totalmente realizado pelo cliente.

2.2.1.2 Trapaça por conluio

A trapaça por conluio (*Cheating by Collusion*) destaca-se em jogos *online* em que é possível que pessoas façam conluio com outros jogadores para ganhar vantagens em detrimento de jogadores honestos. Nesse contexto, um bom exemplo são os conluios de negociação de vitória (*win-trading*) possíveis de serem feitos no jogo *Starcraft*, em que dois jogadores operam alternadamente atividades na classificação competitiva. Nessa trapaça, cada jogador perde propositalmente sua partida para que o outro ganhe os pontos pela vitória, com isso, subindo seu ranking injustamente na classificação.

2.2.1.3 Trapaça de abuso dos procedimentos do jogo

Essa forma de trapaça se caracteriza por ser bastante simplória. Um caso comum da trapaça de abuso dos procedimentos do jogo (*Cheating by Abusing the Game Procedure*) é o escapando (*escaping*). O jogador desconecta sua sessão do jogo quando há uma eminente probabilidade de derrota da partida. Outro bom exemplo é no jogo GO, em que ao final da partida pedras "mortas"devem ser identificadas e removidas manualmente antes do sistema determinar quem é o vencedor. O trapaceador pode secretamente remover pedras "vivas"do oponente e assim alterar o resultado do jogo. Essa trapaça é chamada pela comunidade de GO de trapaça de resultado (*scoring cheating*).

2.2.1.4 Trapaças Relacionadas a Ativos Virtuais

A utilização de dinheiro real em aquisição de itens cosméticos ou visual dos personagens (*skins*) está cada vez mais comum em jogos atualmente. Trapaças relacionadas à ativos virtuais (*Cheating Related to Virtual Assets*), um jogador pode colocar um item para venda, receber o dinheiro e nunca entregar o item como combinado. Esse tipo de trapaça foi reportado na Coreia (PARK, 2001).

2.2.1.5 Trapaça na exploração de inteligência de máquina

Em alguns jogos *online*, jogadores podem explorar IA para trapacear em suas partidas. A utilização de IA nesses casos define a trapaça (*Cheating by Exploiting Machine Intelligence*). Um exemplo é o jogo de xadrez *online*, quando o jogador trapaceiro tem a oportunidade de visualizar a melhor jogada para dada situação apenas rodando um poderoso programa de IA. A situação que ocorre essa trapaça obedece algumas características específicas. O jogo precisa ser modelado de maneira que há um problema computacional e ainda deve haver maturidade suficiente da IA no cenário desse jogo.

2.2.1.6 Trapaça de modificação da infraestrutura do cliente

Na trapaça de modificação da infraestrutura do cliente (*Cheating by Modifying Client Infrastructure*) o jogador pode trapacear modificando a infra-estrutura, tais como dispositivos de *drivers* no seu próprio sistema operacional. Por exemplo, o *driver* gráfico pode ser alterado para que as paredes se tornem invisíveis e assim seja possível enxergar através delas. Essa trapaça é comumente chamada de *wallhack*.

2.2.1.7 Trapaça de negação de serviço a jogadores adversários

É possível um jogador tomar vantagens negando alguns serviços para os adversários. Esse comportamento é visto na trapaça de negação de serviço a jogadores adversários (*Cheating by Denying Service to Peer Players*) Há casos onde o trapaceador atrasa as respostas de seu adversário, inundando (*flooding*) sua conexão, levando os colegas do jogador trapaceado a removê-lo do jogo por julgarem estar com problemas de rede.

2.2.1.8 Trapaça de timing

Um jogador malicioso pode, em alguns jogos *online* em tempo real, atrasar sua própria jogada até saber dos movimentos de seu adversário, assim ganhando vantagem na partida. Isso é chamado de trapaça de timing (*Timing Cheating*).

2.2.1.9 Trapaça de comprometimento de senhas

A trapaça de comprometimento de senhas (*Cheating by Compromising Passwords*) pode levar o trapaceiro a acessar dados e autorizações que a vítima possui no sistema do jogo. Alguns operadores têm proposto guias na seleção de senhas e proteção de seus usuários.

2.2.1.10 Trapaça de exploração na falta de sigilo

Essa trapaça de exploração na falta de sigilo (*Cheating by Exploiting Lack of Secrecy*) se caracteriza por escutar pacotes de comunicação que são transferidos em texto comum. Após acessar os pacotes, é possível inserir, excluir ou modificar eventos ou comandos transmitidos na rede. Essa forma de trapaça pode causar comprometimento de senhas.

2.2.1.11 Trapaça de exploração da falta de autenticação

Em um sistema no qual ocorre falha no mecanismo de autenticação, um trapaceiro pode coletar diversos identificadores (IDs) e senhas de jogadores legítimos configurando um servidor do jogo falso. A trapaça correspondente a esse comportamento é a trapaça de exploração da falta de autenticação (*Cheating by Exploiting Lack of Authentication*). É crítico re-autenticar um jogador antes de qualquer mudança de senhas executados pelo próprio jogador.

2.2.1.12 Trapaça de exploração de bug ou brecha

Um caso conhecido da trapaça (*Cheating by Exploiting a Bug or Loophole*) aconteceu no jogo Habitat desenvolvido pela Lucasfilm's Games, no qual havia um erro na precificação dos itens do jogo em uma casa de penhora. O jogador poderia comprar os itens nas máquinas de venda e revender esses itens na penhora. Muitos jogadores ficaram milionários dentro do jogo por essa brecha que havia no sistema de preço.

2.2.1.13 Trapaça de comprometimento dos servidores

Após obter acesso ao sistema que hospeda o jogo, o trapaceiro pode adulterar os programas dos servidores ou modificar as configurações dos serviços. Essa trapaça é conhecida como *(Cheating by Compromising Game Servers)*. Em jogos digitais no formato de servidores-cliente, a qualidade do jogo estará atrelada a confiança no servidor onde o jogo será hospedado e também de quem executa o jogo. O ponto fundamental dessa trapaça é que, servidores executados por usuários, há grande possibilidade de ocorrer esses comportamentos (PRITCHARD, 2000).

2.2.1.14 Trapaça relacionada ao uso interno indevido

Normalmente o operador detém privilégios como administrador. É relativamente fácil um operador (um empregado) abusar de seu privilégio, esse comportamento é definido como *Cheating Related to Internal Misuse*. Por exemplo, ele pode gerar um super personagem modificando a base de dados do jogo no lado servidor.

2.2.1.15 Trapaça por Engenharia Social

Trapaceiros frequentemente iludem outros jogadores na crença que seja necessário a divulgação de suas credencias. Esse tipo de trapaça (*Cheating by Social Engineering*) tem sido um problema real para empresas de jogos digitais. Grandes empresas do setor, como a *Blizzard*, têm trabalhado no sentido de divulgação de guias para evitar esses esquemas em seus sites e plataformas.

2.2.2 A taxonomia

Na área das ciências biológicas, a taxonomia se refere a um campo da ciência voltada a classificação biológica dos seres. O objetivo da classificação é agrupar os elementos de acordo com características facilmente observadas e descritas permitindo fácil identificação desses elementos (CAIN, 2023). A taxonomia levantada por Rendell e Yan segue o mesmo conceito, cujo objetivo é estabelecer correlação entre as trapaças e características observáveis que elas possuem.

2.2.2.1 Natureza das trapaças

A categorização apresentada não deve ser vista de forma isolada, e muitas trapaças podem ser classificadas em mais de uma categoria. O Quadro 1 apresenta a divisão proposta por Rendell e Yan que classificaram as 15 formas de trapaça em duas divisões: tipo genérico e tipo de especial relevância em jogos *online*. As trapaças tipificadas como *genéricas* incluem oito formas comuns de trapaças tanto em jogos digitais *online*, quanto em aplicações de rede, em que nesses casos, podem ser tratados por outros termos como "ataques"ou "intrusões". Já a divisão de especial relevância, também contém trapaças específicas para jogos *online* e trapaças que, em outros contextos, podem conter nomes diversos em suas aplicações mas é de grande relevância para o contexto de jogos digitais *online* (YAN; RANDELL, 2005).

A taxonomia de Rendell e Yan apresenta um esquema de três dimensões e as as categorias de trapaças são classificadas por suas vulnerabilidades (o que é explorado), consequências (que tipo de falha causa) e ator da trapaça (quem está trapaceando). A Figura 2 mostra detalhes da taxonomia por vulnerabilidade, possibilidade de falhas e quem explora a falha. Através da figura é possível verificar que a mesma trapaça aparece pelo menos uma vez em cada uma das categorias.

Quadro 1 – Formas comuns de trapaça em jogos online

Tipo	Formas de Trapaça
Especial Relevância	Trapaça explorando a adulteração da confiança Trapaça por conluio Trapaça abusando os procedimentos do jogo Trapaças Relacionadas a Ativos Virtuais Trapaça explorando inteligência de máquina Trapaça modificando infraestrutura do cliente Trapaça de timing
Genérico	Trapacear negando serviço a jogadores adversários Trapaça ao comprometer senhas Trapaça explorando a falta de sigilo Trapaça explorando a falta de autenticação Trapacear explorando um bug ou brecha Trapaça comprometendo servidores Trapaça relacionada ao uso indevido interno Trapaça por Engenharia Social

Figura 2 – Classificação dos jogos online

Classificação de vários tipos de trapaça	Vulnerabilidades				Possíveis Falhas					Exploradores			
	Inedequação de Design do Sistema		Pessoas							Independente		Cooperativo	
	inadequações em Sistema Subjacente	inadequações no sistema do jogo	Jogador	Operadores do Jogo	Violação de honestidade	Mascarada	Violação de integridade	Negação de Serviço	Roubo de informação ou posse	Jogador Único	Operador do Jogo	Multiplos Jogadores	Operador e jogadores
Trapaça Explorando a Adulteração da Confiança		●					●		●	●			
Trapaça por Conluio		●			●				●			●	
Trapaça Abusando os Procedimentos do Jogo		●			●					●			
Trapaças Relacionadas a Ativos Virtuais			●		●					●			
Trapaça Explorando Inteligência de Máquina		●			●					●			
Trapaça Modificando Infraestrutura do Cliente	●						●			●			
Trapacear Negando Serviço a Jogadores Pares	●	●						●		●			
Trapaça de Timing		●			●		●			●			
Trapaça ao Comprometer Senhas			●						●	●			
Trapaça Explorando a Falta de Sigilo		●					●		●	●			
Trapaça Explorando a Falta de Autenticação		●				●				●			
Trapaça Explorando Um Bug ou Brecha		●			●					●			
Trapaça Comprometendo Servidores	●						●			●			
Trapaça Relacionada ao Uso Interno Indevido				●			●				●		●
Engenharia Social			●						●	●			

Fonte: Rendell e Yan, A systematic classification of cheating in online games (2005).

2.2.2.2 Por vulnerabilidade

Jogos *online* podem ser explorados através de suas inadequações. A trapaça *exploração de um bug ou brecha*, por exemplo, se aproveita de inadequações do design do jogo, da implementação em si ou ambos. Por outro lado, a *engenharia social* não explora defeitos técnicos, mas a interação interpessoal entre os jogadores ou profissionais envolvidos com o jogo. Através dessa distinção, a taxonomia de Rendell e Yan classifica as vulnerabilidade em duas divisões: *inadequações de design do sistema* que se refere a falhas de design técnico na fase de desen-

volvimento e a vulnerabilidade de várias *pessoas* envolvidas na operação ou jogando os jogos *online*. A partir dessas duas definições são criadas algumas subdivisões. A divisão *inadequações de design do sistema* se subdivide em *inadequações no sistema do jogo* e *inadequações em sistema subjacentes*. Sistemas subjacentes podem ser definidos como a infraestrutura que um jogo *online* precisa para ser executado. Um jogador malicioso pode explorar uma falha do jogo, da infraestrutura ou ambos.

Trapaças que fazem parte da divisão de *inadequações no sistema do jogo* são: *trapaça explorando a adulteração da confiança*, *Trapaça explorando a falta de autenticação*, *trapaça de timing* e *exploração de um bug ou brecha*. Todas essas trapaças se aproveitam de uma forma ou outra de inadequações técnicas e por isso são colocados nessa categoria. *Trapaça por conluio*, *trapaça de abuso dos procedimentos do jogo*, e *trapaça de exploração de inteligência de máquina* podem também serem caracterizadas como falha no design técnico do jogo pela "incapacidade de prever todas as situações que o sistema enfrentará durante sua vida operacional, ou a recusa em considerar alguns deles"(LAPRIE, 1992). *Inadequações em sistema subjacentes* são trapaças que exploram a infraestrutura do sistema. Entre elas estão *trapaça de modificação da infraestrutura do cliente* e *trapaça de comprometimento de servidores*.

Pode haver ainda trapaças que envolvem as duas divisões. A trapaça *negação de serviço a jogadores* geralmente é explorada pela fraqueza da camada de rede. Entretanto, há casos que a trapaça é explorada pela inadequação do design do jogo. Um exemplo dessa trapaça ocorre em jogos FPS(Tiro em Primeira Pessoa), em que o jogador trapaceiro espera, em um lugar seguro seu adversário nascer no cenário a fim de matá-lo. O jogador que sofre essa trapaça não tem tempo de reação. Um desenvolvedor poderia, nesse caso, garantir uma imunidade temporária aos jogadores que ressurgissem no cenário de jogo para que não tivessem prejuízo no momento de recomeçar a partida.

Outras formas, como *engenharia social*, *trapaça de comprometimento de senhas*, *trapaças relacionadas a ativos virtuais* e *trapaça relacionada ao uso interno indevido* são lateralmente relacionadas em alguma falha no design, entretanto elas são mais atreladas a trapaças de convivência entre jogadores ou operadores dentro do jogo (YAN; RANDELL, 2005).

2.2.2.3 Consequência das trapaças

O trabalho de Rendell e Yan classifica as consequências em quatro aspectos tradicionais de segurança computacional: confidencialidade (prevenção da divulgação não autorizada das informações), integridade (prevenção da modificação não autorizada das informações), disponibilidade (prevenção de retenção não autorizada de informações) e autenticidade (habilidade de assegurar a identidade do usuário). A violação na confidencialidade gera *furto de informação ou posse*, uma brecha no quesito de integridade resulta em modificações impróprias no jogo por consequência em *violação de integridade*, problemas na disponibilidade gera *negação de serviço* e a quebra na autenticidade resulta em um tipo de trapaça conhecido como *masca-*

rade(mascarado).

Trapaça por conluio, *trapaça ao comprometer senhas*, ou *engenharia social* podem resultar em furto de informação ou posse no jogo. A exploração da *trapaça de falta de autenticação* resulta no tipo mascarado. A negação de serviço é causada através da *trapaça de negação de serviço a jogadores pares* que consiste em negação seletiva de serviços. Já a *Trapaça de comprometimento de servidores*, *trapaça de modificação da infraestrutura do cliente* ou *trapaça relacionada ao uso interno indevido* geralmente envolvem modificações das características do jogo causando falha da integridade.

Embora a maioria das trapaças se encaixam nessas definições, há alguns comportamentos que não violam a confidencialidade, integridade, disponibilidade e autenticidade. Para esses casos é introduzida uma nova definição nessa taxonomia: *lealdade* entre jogadores. Podem ser classificados nessa categoria, as trapaças *abuso de procedimentos do jogo*, *exploração de um bug ou brecha*, *exploração de inteligência de máquina*, *relacionadas a Ativos Virtuais* e *conluio*. Quando há a ocorrência dessas trapaças, pode-se determinar que há violação de lealdade (YAN; RANDELL, 2005).

2.2.2.4 Por ator da falha

É possível que um jogador trapaceie individualmente tanto em jogos do tipo jogador único quanto *multiplayer*, enquanto que em jogos *multiplayer* dois ou mais participantes podem colaborar para realizar a trapaça. A identidade do ator da trapaça é utilizada para distinguir trapaças *cooperativas* das *independentes*.

A divisão das *cooperativas* se subdivide em *múltiplos jogadores* em que dois ou mais jogadores devem estar comprometidos entre eles para realizar a trapaça. Um exemplo dessa categoria é a *trapaça por conluio*. Ainda nessa categoria, tem-se a a subdivisão *operador e jogador* e tipicamente envolve o conluio entre pelo menos um jogador e uma pessoa interna do sistema.

Do mesmo modo que a divisão *cooperativa*, a divisão *independente* também se divide em duas subdivisões. A categoria *operador do jogo* acomoda trapaças relacionadas ao uso interno indevido mas não há conluio entre um jogador externo. A categoria de *jogador único* engloba todas as trapaças que o jogador pode realizar sozinho.

2.3 CONSIDERAÇÕES FINAIS SOBRE A TAXONOMIA

Apesar da indústria contemplar jogos do tipo jogador único, é bastante relevante o crescimento do setor devido aos jogos do tipo *multiplayer online*, como pode ser confirmado em campeonatos de *e-Sport*. Nesse contexto de grande responsabilidade, a integridade dos jogos digitais é vital. Com a análise da taxonomia é possível verificar que as vulnerabilidades são exploradas tanto nos sistemas dos jogos digitais quanto com pessoas envolvidas com o jogo

em questão (funcionários). Também verifica-se que um jogador pode cometer a maior parte das trapaças de maneira independente. Entretanto, há casos de conluio com outros jogadores ou pessoas internas do jogo. Falhas na segurança da aplicação promovem diversos tipos de trapaças que exploram defeitos no sistema do jogo. Como pode-se observar, as trapaças ocorrem por diversas formas, embora na maioria dos casos é devido a falhas de segurança.

3 BIBLIOMETRIA SOBRE TRAPAÇAS EM JOGOS DIGITAIS

A definição de diretrizes sólidas para desenvolvedores de jogos digitais necessita de um estudo profundo sobre os riscos da trapaça para o ecossistema desse entretenimento. Faz-se, então, o uso de métodos bibliométricos para extrair uma síntese coesa e relevante para auxiliar os desenvolvedores na escolha da melhor técnica antitrapaça. O objetivo dessa análise é identificar o estado da arte sobre o tema de trapaças em jogos digitais *online* a avaliar a base teórica acerca do assunto.

3.1 DEFINIÇÕES DO ESTUDO BIBLIOMÉTRICO

A partir de um conjunto definido de artigos, o portfólio bibliométrico, é possível evidenciar quantitativamente os parâmetros desse portfólio e assim gerenciar as informações e conhecimento relativo a tal assunto. O conhecimento construído é representado pela seleção do portfólio, bem como a posterior análise bibliométrica desses artigos (LACERDA; ENSSLIN; ENSSLIN, 2012). Para tanto, é importante apresentar conceitos e as leis que regem esse setor do conhecimento.

3.1.1 Método Proknow-C

O método PROKNOW-C (L *et al.*, 2010) inicia pelo interesse do pesquisador sobre um tema, e também suas delimitações e restrições intrínsecas ao contexto acadêmico, em busca da construção do conhecimento no pesquisador, a fim de que ele possa iniciar uma pesquisa científica com fundamentação (LACERDA; ENSSLIN; ENSSLIN, 2012). O conceito de análise bibliométrica se baseia na evidenciação quantitativa dos parâmetros do portfólio bibliográfico. Os parâmetros observáveis são: artigos selecionados, suas referências, autores, número de citações e periódicos mais relevantes (L *et al.*, 2010).

3.1.1.1 Pesquisando em Bancos

O processo de pesquisa científica se inicia com um problema, pergunta ou dúvida, que motiva os pesquisadores a procurarem informações sobre um dado tema em bibliotecas e bases bibliográficas digitais (TASCA *et al.*, 2010). O acesso mais facilitado às bases de dados, que em última análise, são sistemas de indexação de periódicos, livros, teses, relatórios anais de eventos dentre outros, proporcionou uma plataforma de utilização desses dados para pesquisas futuras (LACERDA; ENSSLIN; ENSSLIN, 2012).

Além de ser um instrumento que facilita a recuperação e utilização do conhecimento científico em pesquisas, as bases de dados também contribuem com o estabelecimento de in-

dicadores para visualizar potencial de impacto de um determinado periódico em uma área de conhecimento (PODSAKOFF *et al.*, 2005). Para mensurar, interpretar e avaliar os resultados obtidos das buscas, pesquisadores recorrem a técnicas bibliométricas, que são análises quantitativas com fins a mensurar a produção e disseminação científica (ARAúJO, 2006).

3.1.1.2 Selecionando Referencial Teórico

Um aspecto crítico para que o pesquisador consiga posicionar seu objetivo de pesquisa em um campo abrangente e disperso é a identificação do estágio atual do conhecimento sobre uma determinada área científica, seja ele teórico ou empírico (TRANFIELD; DENYER DAVID SMART, 2003). A revisão acadêmica sobre a área de interesse auxilia a atender a demanda. Para Karlsson (KARLSSON, 2008) essas atividades de análise da literatura auxiliam o pesquisador a:

a) Obter o respaldo científico sobre seu trabalho, ao se basear no que tem sido publicado no assunto de interesse
b) Justificar a escolha do tema e a contribuição da sua proposta de pesquisa
c) Gerar uma justificativa sobre o seu enquadramento metodológico
d) Restringir o escopo da pesquisa, tornando-a um projeto factível
e) Desenvolver no pesquisador habilidades em análise crítica da literatura e no tratamento de informações abrangentes e dispersas

3.2 BIBLIOMETRIA APLICADA SOBRE TRAPAÇAS EM JOGOS DIGITAIS

Para alcançar os objetivos propostos para análise bibliométrica, a metodologia seguiu quatro passos sequenciais.

a) Escolher a base de dados relevante para o tema
b) Definir os parâmetros da pesquisa
c) Filtrar dos resultados para análise
d) Agrupar os documentos obtidos na filtragem
e) Analisar os resultados

3.2.1 Escolha da base de dados

Na primeira etapa, selecionou-se a base de dados em que seria realizada a coleta das publicações. Optou-se pela base *Web of Science* para as consultas dos artigos e periódicos científicos. A escolha por essa base deve-se ao fato de que possui uma cobertura global e completa

dos assuntos propostos e é reconhecidamente estruturada para análise de informações para a produção de indicadores, sem necessidade de manipulações prévias dos dados (SANTOS, 2003). A base de dados da *Web of Science* faz parte do *Institute for Scientific Information* (ISI) (Instituto para Informação Científica) que indexa desde a década de 40, a *Science Citation Index Expanded* (SCI-EXPANDED) (Índice Expandido de Citações Científicas), a *Social Sciences Citation Index* (SSCI) (Índice de Citações em Ciências Sociais), e a *Arts and Humanities Citation Index* (AHCI) (Índice de Citações em Artes e Humanas) (PINTO; GONZALES-AGUILAR, 2014). Além disso, essa base contém um dos principais repositórios de produção científica do mundo (LóPEZ-BELMONTE *et al.*, 1920). Isso proporciona um alcance satisfatório para o tema pesquisado.

O segundo passo da metodologia constitui na configuração dos parâmetros de busca e período da cobertura. Foram realizadas duas pesquisas para abranger maior número de trabalhos. A primeira pesquisa utilizou os descritores *game* and (cheat* or security)* com alcance temporal de até cinco anos e produções em qualquer idioma, sendo trabalhos em inglês os mais numerosos.

Para o campo de pesquisa, utilizou-se *Tópico*, que engloba a consulta por título, resumo e as palavras-chave. Dessa busca, resultaram 4166 documentos não se limitando a artigos mas qualquer trabalho que envolva o tema. A segunda pesquisa utilizou os descritores *online game* and cheat** sem restrição temporal, ou seja, buscou-se na base desde 1945 até o momento. A escolha de não restringir o tempo de cobertura deve-se ao fato de que nessa pesquisa, os descritores já estavam limitando bastante os trabalhos buscados na base. Dessa forma, para aumentar o número de publicações, optou-se por essa abordagem. Nessa segunda busca, obteve-se 190 trabalhos.

Na terceira etapa depurou-se os resultados. Como a base de dados contém mais de um repositório, há a possibilidade de que a pesquisa resulte em trabalhos duplicados. Ainda, como os termos de pesquisa são genéricos, o resultado obtido possui vários trabalhos que não pertencem ao escopo do estudo. Para corrigir esse desvio, fez-se uso de planilhas a fim de retirar possíveis documentos duplicados, sem aderência à pesquisa ou publicações fora de espoco da temática.

Após a primeira filtragem, o total de resultados passou por novo processo de classificação. Tendo selecionado os trabalhos com título ou resumo alinhado ao tema, iniciou-se o processo de classificação desses artigos por número de citações. Nessa ordenação calculou-se a porcentagem das citações baseada no total de citações do portfólio buscado, ou seja, pegou-se o número de citações do artigo e dividiu-se pelo total de citações de todos os artigos. Após isso, obteve-se a porcentagem acumulada do artigo, calculando a porcentagem do artigo em questão mais o seu antecessor na listagem. Para selecionar os artigos mais relevantes, utilizou-se a regra de Pareto, na qual diz que uma pequena minoria da população representa a maior parte do efeito (LACERDA; ENSSLIN; ENSSLIN, 2012). Nesse contexto, 80% da citações acumuladas são causa-

das por 20% dos artigos, assim sendo, a linha de corte para seleção dos artigos, são aqueles que chegam até 80% na porcentagem acumulada.

Na quarta etapa, a partir do resultado anterior, foi possível agrupar todos documentos da pesquisa em uma única consulta na base da *Web of Science*. Com isso, exportou-se o resultado da busca com todos os dados em formato *.txt* .Com auxílio do programa Vos viewer [1], pode-se visualizar as redes dos dados criados.

3.3 RESULTADO DA ANÁLISE BIBLIOMÉTRICA

O objetivo dessa análise é avaliar a produção científica da temática *Trapaça em jogos Online*. Nesse cenário, é importante realizar uma linha histórica da produção em cada ano.

Nota-se na Figura 3 a visão para a busca acerca do tema com corte de cinco anos. Observa-se que na primeira busca, entre os anos de 2019 e 2022 obteve-se uma base de produção entre os 1000 trabalhos por ano, mais precisamente 982 em 2019, 971 em 2020, 1021 em 2021, 1027 em 2022 e 249 (até a produção dessa análise) em 2023. O primeiro trabalho indexado na *Web of Science* com termos abordados na pesquisa é datado de 1982. Em mais de 40 anos de trabalhos indexados na base é possível verificar que os últimos anos são os mais produtivos a respeito do tema, e que, gradualmente, todos os anos tem contabilizado mais trabalhos que o ano anterior. O ano com maior produtividade é de 2022 com 1027 trabalhos produzidos. Já para a segunda busca, em que se estreitou a pesquisa para considerar jogos *online*, obteve-se uma variação dos últimos 5 anos como é visto na Figura 4. O ano de 2020 contabilizou com o ano com maior número de trabalhos tendo 17 trabalhos registrados.

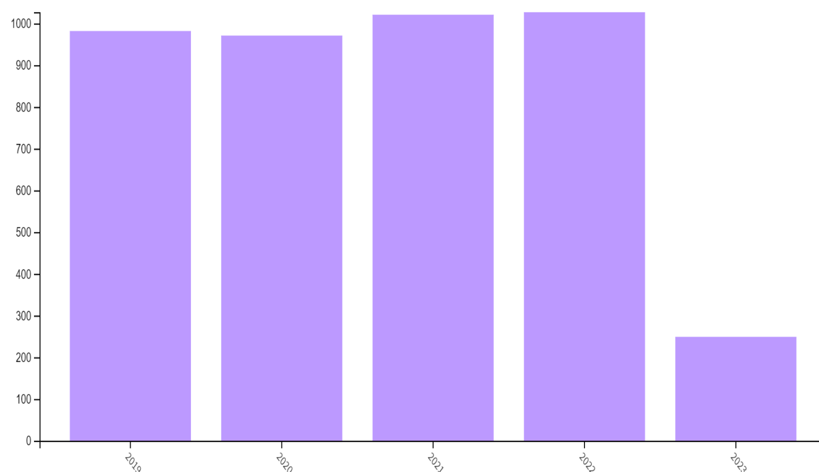
No quesito autores mais produtivos, pode-se apontar Zhu QY com 27 trabalhos publicados, seguido de Wang Y com 26 e Niyato D com 24 produções indexadas na base. Na Figura 5 é mostrado os autores com mais produções listadas na base *Web of Science*.

A partir desses resultados, utilizou-se o método PROKNOW-C para delimitar um conjunto com maior importância para a pesquisa. Para tanto, observa-se os autores que possuem maior relevância nesse contexto na Figura 6. No mapa de calor é demostrado que os autores Jianrong Tao e Changjie Fan têm grande importância para a pesquisa com mais número de trabalhos publicados. Ainda nessa análise, é bastante nítido que a maioria dos autores não possuem muitas produções, indicando pouco substrato teórico nessa temática visto que fora realizado o processo de filtragem através do método escolhido.

A análise de coautoria visa examinar as redes criadas pelos cientistas por colaboração e artigos científicos (ACEDO *et al.*, 2006). Por causa dos dados bibliográficos conterem informações sobre as filiações institucionais e localizações geográficas dos autores, a análise de coautoria pode examinar os problemas da colaboração em nível institucional e de país (ZUPIC; ČATER, 2015). Pode-se observar na Figura 7 que há poucos autores que colaboram entre si nos

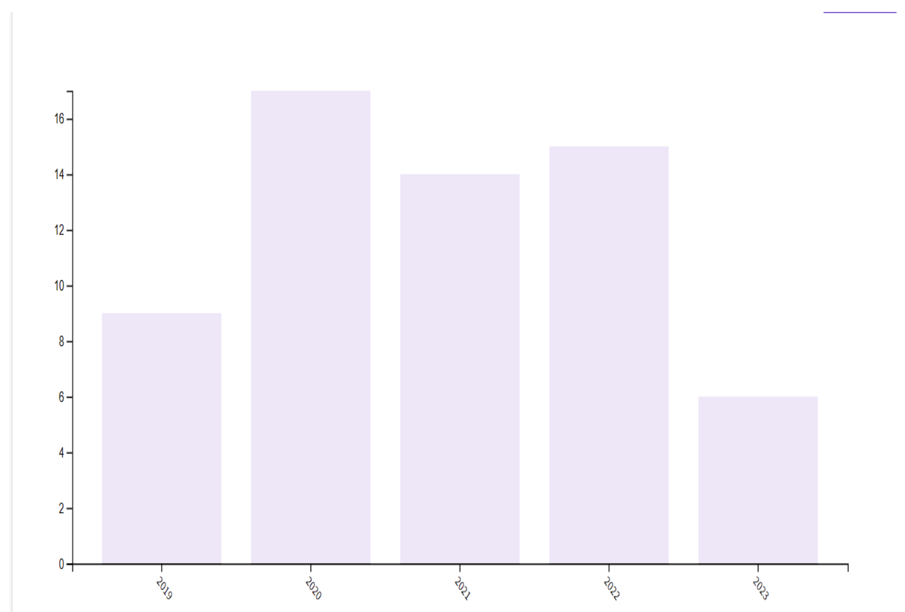
[1] https://www.vosviewer.com/

Figura 3 – Produção científica anual da primeira busca



Fonte: O Autor (2023).

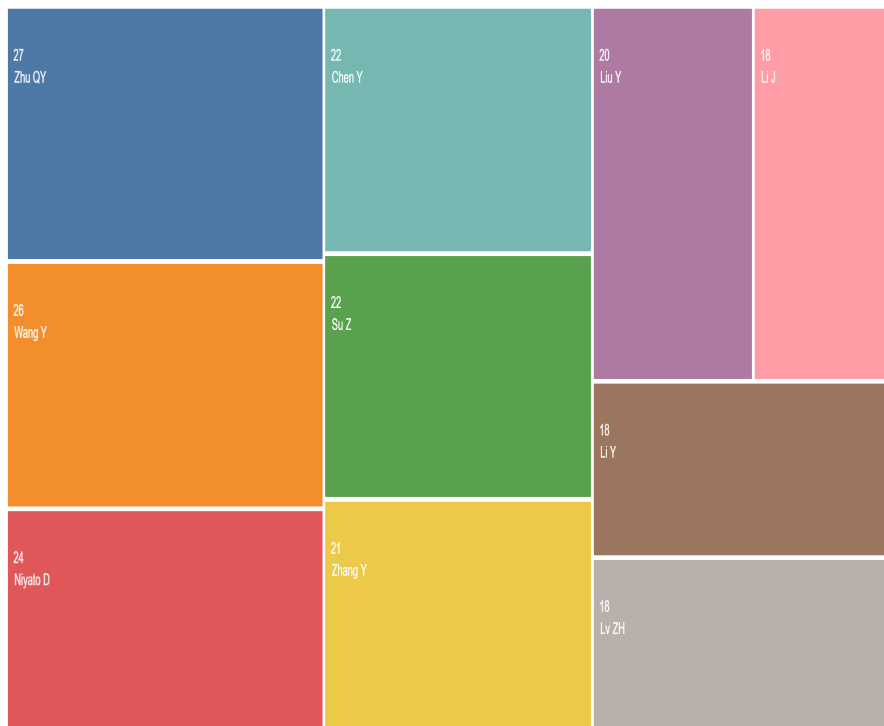
Figura 4 – Produção científica anual na busca específica para jogos online



Fonte: O Autor (2023).

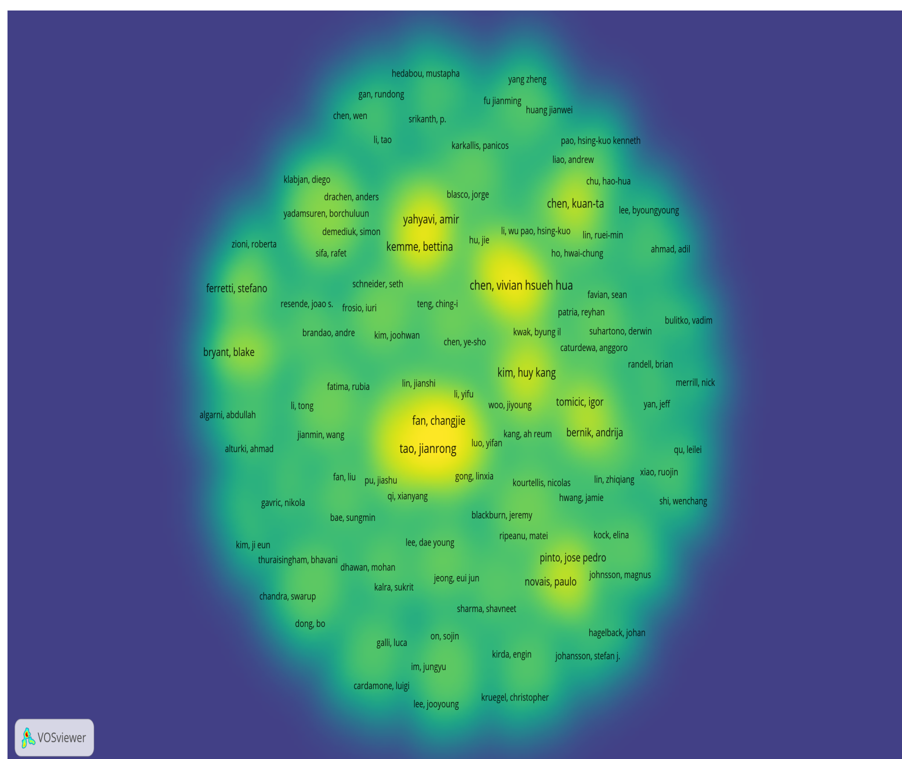
trabalhos avaliados. Essa característica corrobora a ideia que não há, de fato, uma grande quantidade de produções que visam a temática de trapaças em jogos *online*. É importante lembrar que essas avaliações são baseadas na filtragem do método e apenas os artigos mais relevantes são revisitados para observação.

Figura 5 – Autores com maior número de produções



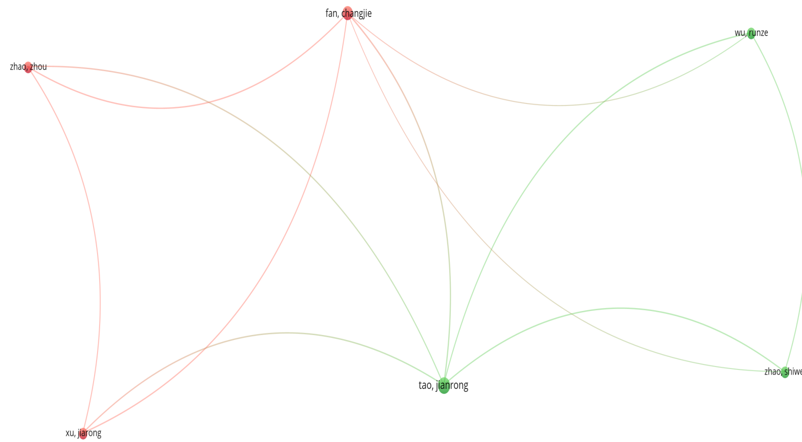
Fonte: O Autor (2023).

Figura 6 – Mapa de calor dos autores filtrados pelo método Proknow-C



Fonte: O Autor (2023).

Figura 7 – Visão de coautoria



Fonte: O Autor (2023).

3.4 CONSIDERAÇÕES SOBRE A ANÁLISE BIBLIOMÉTRICA

Observa-se, portanto, a importância do estudo do estado da arte referente ao tópico analisado pois atesta com evidências poucas publicações acerca do tema de trapaças em jogos digitais *online*. Apesar da primeira busca na base de dados coletar grande número de trabalhos, muitos desses tratavam de assuntos que não tinham relação ao escopo da pesquisa. Por outro lado, restringindo a consulta com termo mais específico, observou-se que, aparentemente essa área de conhecimento não é prioritária o suficiente para novos trabalhos. Espera-se que com novas tecnologias surgindo nos últimos anos na interação com os jogos digitais *online*, como VR e *Augmented Reality* (AR) (Realidade Aumentada), tenha também um aumento no interesse de novas pesquisas e técnicas para combater esse problema das trapaças.

4 TÉCNICAS DE IMPLEMENTAÇÃO ANTITRAPAÇA

Diante do cenário visto no capítulo 3, no qual é apresentada a pesquisa bibliométrica e seus resultados para o tema proposto, possibilitou-se a verificação de trabalhos que fazem uso de algum tipo de técnica ou sugestão antitrapaça. Dentre essas técnicas, analisou-se a utilização do componente *ML-Agent* na *Engine Unity*, biblioteca centrada na IA.

4.1 IMPLEMENTAÇÕES ANTITRAPAÇA NA BIBLIOGRAFIA

Em um cenário idealizado, o objetivo dos trabalhos revisados é propor abordagens para erradicar as trapaças dos jogos *online* de modo a não haver mais vantagens indevidas para alguns jogadores. No decorrer da análise bibliométrica, observou-se produções que trabalham o tema de trapaças em jogos digitais a nível operacional, com implementações visando se não a remoção por completo do problema mas, pelo menos, mitigando os malefícios dessas atitudes. É importante determinar as técnicas propostas inserindo essas abordagens nas categorias estudadas pela taxonomia no capítulo 2. Encontrou-se na bibliografia algumas das categorias de trapaças estudadas na taxonomia.

4.1.1 Manipulação da infraestrutura do cliente

Em muitos casos, trapaças em jogos digitais necessitam de ferramentas externas para acessar os processos da vítima i.e., um injetor. Os pesquisadores Panicos Karkallis, Jorge Blasco, Guillermo Suarez-Tangil e Sergio Pastrana abordam esse tema no trabalho *Detecting Video-Game Injectors Exchanged in Game Cheating Communities* (KARKALLIS *et al.*, 2021). Nele, os autores criaram um classificador de aprendizado de máquina que identifica injetores baseados em suas características comportamentais.

Outro trabalho que aborda problemas na infraestrutura do cliente é o *Robust Vision-Based Cheat Detection in Competitive Gaming* de autoria de Aditya Jonnalagadda, Iuri Frosio, Seth Schneider, Morgan McGuire e Joohwan Kim. Nesse artigo os autores focam o estudo na detecção de trapaças *wallhack* utilizando rede neural profunda (JONNALAGADDA *et al.*, 2021).

Um trabalho com relevância significativa é o trabalho *Behavioral-Based Cheating Detection in Online First Person Shooters Using MachineLearning Techniques* dos autores Hashem Alayed, Fotos Frangoudes e Clifford Neuman. Tal artigo, detém 1431 visualizações na base *Institute of Electrical and Electronics Engineers* (IEEE)(Instituto de Engenheiros Eletríco e Eletrônico) e 16 citações. Os pesquisadores, nesse artigo, exploram a detecção de jogos utilizando apenas logs nos comportamentos dos jogadores. Depois da coleta dos registro de logs, são aplicadas técnicas de aprendizado de máquina supervisionado para criar modelos de detecção

(ALAYED; FRANGOUDES; NEUMAN, 2013). O objetivo com esse processo é detectar jogadores que utilizam trapaça na ajuda de mira em jogos do tipo FPS.

O próximo trabalho, utiliza aprendizagem de máquina no jogo Unreal Tournament III. Os autores, Luca Galli, Daniele Loiacono, Luigi Cardamone, Pier Luca Lanzi, criaram um *framework* que consiste em três componentes: servidor que coleta informações, um *backend* de pré-processamento e um *frontend* que analisa os dados. Após o pré-processamento dos dados, o *backend* explora modelos de decisões para detectar comportamentos de trapaça (GALLI; LOIACONO; CARDAMONE LUIGI; LANZI, 2011).

No trabalho *Machine Learning and Anti-Cheating in FPS Games* Alkhalifa analisa trapaças no gênero de FPS. O jogo escolhido foi o CS-GO, cujo objetivo principal é verificar a utilização de *aimbots*. Os dados de cada jogador são coletados e analisados, gerando saídas e através da utilização do modelo *Markov* é determinado se o jogador está trapaceando. Esse estudo se baseia primordialmente na direção de visão do jogador (ALKHALIFA, 2016). O trabalho demonstra o design e implementação de sistemas antitrapaça para um jogo do gênero FPS.

4.1.2 Utilizando Inteligência de máquina

O objetivo dos autores Hsing-Kuo Pao, Kuan-Ta Chen e Hong-Chung Chang no trabalho *Game Bot Detection via Avatar Trajectory Analysis* é detectar o uso de *bots* em jogos *online* baseado na trajetória dos jogadores no jogo Quake II. O estudo é baseado no fato do jogador quase nunca utilizar o caminho similar ou idêntico, enquanto *bots*, por utilizarem algoritmos para otimizar tempo e espaço, utilizam esses caminhos. As medições são combinadas com múltiplas técnicas de aprendizado e classificação para detecção e a abordagem é generalizável para qualquer jogo em que os movimentos dos personagens são controlados diretamente pelos jogadores (PAO; CHEN; CHANG, 2010).

4.1.3 Técnicas para trapaças genéricas

No artigo *Probabilistic Approaches to Cheating Detection in Online Games*, os autores, Laetitia Chapel, Dmitri Botvich e David Malone, analisam o nível (*rank*) do jogador através de um *framework*. Segundo os autores, o *framework* possibilita a detecção de trapaças em qualquer gênero de jogo, desde MMORPG até FPS. Para realizar a detecção, o único atributo que é necessário reconhecer no jogo é o nível atual do jogador. Supõe-se que o jogador trapaceiro tenha um posto artificialmente aumentado, e com tal suposição, os autores tentam detectar o trapaceiro calculando modelos probabilísticos (CHAPEL; BOTVICH; MALONE, 2010).

4.2 FERRAMENTA ANTITRAPAÇA NA *UNITY*

A pesquisa bibliométrica foi focada para implementações de jogos na *Engine Unity*, com isso, verificou-se a utilização da biblioteca ML-Agent para esse propósito. Como visto nos

trabalhos revisados, a IA é uma área de conhecimento da Ciência da Computação com modelos que auxiliam desenvolvedores na implementação de soluções antitrapaça. Sendo assim, a IA pode ser utilizada como um valioso instrumento para determinadas soluções na área da computação, mais precisamente, na análise ou implementação de modelos antitrapaça. Para esse trabalho, o foco é apresentar a biblioteca *ML-Agent*, componente para *Unity Engine* que utiliza aprendizado por reforço.

4.2.1 *Unity com ML-Agents*

Como já discutido, a IA tem fundamental relevância nas implementações para detecção de trapaça nos jogos digitais *online*. Entretanto, poucas soluções são utilizadas sobre a *Unity*. Portanto, para que seja desenvolvida solução nessa plataforma com auxílio de IA, escolheu-se o opção de utilização do componente *ML-Agents*, pois ela é relativamente simples de configurar e não há custo na utilização.

Sobre o kit de ferramentas *ML-Agents*, o artigo *Unity: A General Platform for Intelligen tAgents* descreve brevemente as principais características do componente. O *ML-Agents* é um projeto de código aberto que permite jogos e simulações a servirem como ambientes para treinamento de agentes inteligentes usando editor da *Unity* e interagindo com eles via *Application Programming Interface* (API)(Interface de Programação de Aplicações) *Python*. O *Software Development Kit* (SDK)(Conjunto de Desenvolvimento de Software) da ferramenta providencia funcionalidades e implementações baseadas em *PyTorch* (biblioteca para aprendizado de máquina) de algoritmos de estado da arte para permitir desenvolvedores a facilmente treinar agentes inteligentes para jogos 2D, 3D, VR/ AR em scripts C#. Pesquisadores podem utilizar a API em *Python* disponibilizada para treinar agentes usando aprendizado por reforço, aprendizado de imitação, neuroevolução, ou qualquer outro método. Esses agentes treinados podem ser usados para múltiplos propósitos, incluindo controle de comportamento de *Non-Player Character* (NPC)(Personagem não Jogável), testes automatizados de construção de jogos e avaliação de diferentes designs de jogos de pré-lançamento.

As três entidades centrais do SDK do *ML-Agents* são sensores, agentes e a academia. O componente Agente se refere ao *GameObject* dentro da cena na *Unity* que podem coletar observações, tomar ações ou receber recompensas. O agente pode coletar observações usando uma variedade de possíveis sensores correspondendo a várias formas de informações como renderização de imagens, resultado *ray-cast* ou arbitrar tamanhos de matrizes. Cada componente Agente contém uma política rotulada com *nome do comportamento*.

Qualquer número de agentes pode possuir a política com o mesmo nome de comportamento. Essa política será executada por esses agentes e compartilharão os dados durante o treinamento. Também, há possibilidade de coexistirem, dentro da cena, qualquer número de nomes de comportamento para políticas permitindo cenários de construção de multi-agentes com agentes em grupos ou individuais executando muitas atividades diferentes. Uma política pode

referenciar várias tomadas de decisão incluindo comandos do jogador, scripts diretos no código, modelos de redes neurais internas, ou através da API *Python*. A função de recompensa é usada para providenciar um sinal para o agente. Ela pode ser definida ou modificada a qualquer momento durante a simulação usando o sistema de scripts da *Unity*.

A Academia é um padrão *singleton*, na qual garante que uma classe tenha somente uma instância dentro do escopo e fornece apenas um ponto de acesso para classe (GAMMA *et al.*, 2000). A academia é usada para rastrear os passos da simulação e gerenciar os agentes. A academia também contém a habilidade de definir parâmetros de ambiente, que podem ser usados para mudar as configurações no ambiente, isso no momento da execução da aplicação. Aspectos da física e texturas, tamanhos e existência de *GameObjects* são controladas através de parâmetros expostos que podem ser reorganizados e alterados durante todo o treinamento. Isso permite avaliação de agentes no treinamento a separar as variações do ambiente e facilita a criação de cenários de aprendizagem (JULIANI *et al.*, 2020).

4.3 CONSIDERAÇÕES DAS TÉCNICAS ANTITRAPAÇA

Nota-se, pelos trabalhos analisados que a IA é peça fundamental de soluções antitrapaça atualmente, visto que, mesmo estratégias que utilizam apenas logs de registros, realizam algum tipo de processamento com IA. Também, é possível perceber que a maioria das soluções para detecção de trapaça se encaixam numa determinada categoria da taxonomia apresentada nesse trabalho. Nesse contexto, para implementação de uma solução na *Unity*, escolheu-se trabalhar com a biblioteca *ML-Agents*, o nível de complexidade não é elevado e é um componente gratuito.

5 CAÇADOR DE RECOMPENSA - PROTÓTIPO DE JOGO *MULTIPLAYER* 3D PARA DETECÇÃO DE TRAPAÇA

Será desenvolvido um protótipo de jogo a fim de implementar uma técnica antitrapaça através do *ML-Agent* no motor gráfico *Unity*. Esse protótipo será modelado através do *Game Design Document* (GDD)(Documento de Design de Jogos), onde constará informações pertinentes ao desenvolvimento e implementação da solução. Também serão empregados os diagramas da UML , determinando as características estruturais e comportamentais do projeto tais como: diagrama de casos de uso, diagrama de componentes, diagrama de classes, diagrama estados e diagrama de sequência. A implementação seguirá com a criação de um jogo *multiplayer* na *Engine Unity* com o nome de *Caçador de Recompensa*. No desenvolvimento, será integrado o componente externo ao sistema da *Unity*, o *ML-Agents*, o qual terá a responsabilidade de observar o ambiente de jogo. Para o escopo desse trabalho, optou-se pela implementação do protótipo para computador e tendo como foco o gênero *Arcade*. Seguindo a classificação de trapaças vista no capítulo 2, a trapaça que a implementação visa detectar é *Trapaça de exploração de inteligência de máquina*. No segundo momento do trabalho, após a implementação do protótipo, será realizada a fase de testes, em que se avaliará a viabilidade da solução antitrapaça.

5.1 DESIGN DO JOGO

É importante antes da implementação do protótipo, definir as principais características do jogo desenvolvido. Nesse contexto, o processo de desenvolvimento de jogos normalmente é guiado através de um documento específico que descreve partes importantes do sistema. Esse documento é conhecido como GDD, o qual pode ser criado de diversas formas. Entretanto, é necessário, para que não se tenha retrabalho no desenvolvimento, que o documento estabeleça diretrizes alinhadas com todo o time envolvido (SALAZAR *et al.*, 2012). Detalhes do jogo, história, mecânicas, arte, sons e música, recompensas e regras são alguns exemplos de descrições que um GDD fornece, embora não seja consenso de que modo esse artefato é produzido (DEWINTER; MOELLER, 2016). Segundo Joris Dormans, em sua dissertação *Engineering Emergence: Applied Theory for Game Design*, "[GDDs] tipicamente contêm descrições de mecânicas centrais do jogo, design dos níveis, notas para direção de arte, personagens e suas histórias, etc. Alguns advogam em descrições detalhadas cobrindo qualquer detalhe do jogo, enquanto outros são a favor de documentos breves que capturam os objetivos de design e suas filosofias"(DORMANS, 2012). Para o GDD do proposto protótipo, elencou-se as principais características do jogo como *gameplay* (como é jogado), mecânicas dos personagens e as regras do jogo.

5.1.1 Jogabilidade

O jogo será desenvolvido em um ambiente 3D, com dois times adversários. O objetivo do jogo é coletar todos os objetos (caixas) do mapa e conduzi-los para pontos específicos demarcados como alvo. Entretanto, essas caixas são protegidas por inimigos, assim, para que o jogador possa coletá-las é necessário a eliminação desses inimigos. Os inimigos surgem no cenário de jogo em posições aleatórias na parte central do mapa. O *layout* do mapa será como visto na Figura 8. Para o protótipo, escolheu-se determinar cores para diferenciar os times, sendo o time azul e o time vermelho. Cada time tem seu respectivo local de surgimento e objetivo. O local de surgimento do time azul é determinado como Time Azul e o objetivo como OBJETIVO AZUL no mapa. De maneira semelhante, o local de surgimento do time vermelho é determinado como Time Vermelho e o objetivo como OBJETIVO VERMELHO. O jogo terá um tempo estipulado de 10 (dez) minutos e se o tempo se esgotar, o time com maior pontuação vence o jogo. Por se tratar de um protótipo, serão utilizadas apenas formas geométricas, sendo elemento do tipo cápsula para os jogadores e inimigos, e formas quadradas para os elementos restantes como é mostrado na Figura 9.

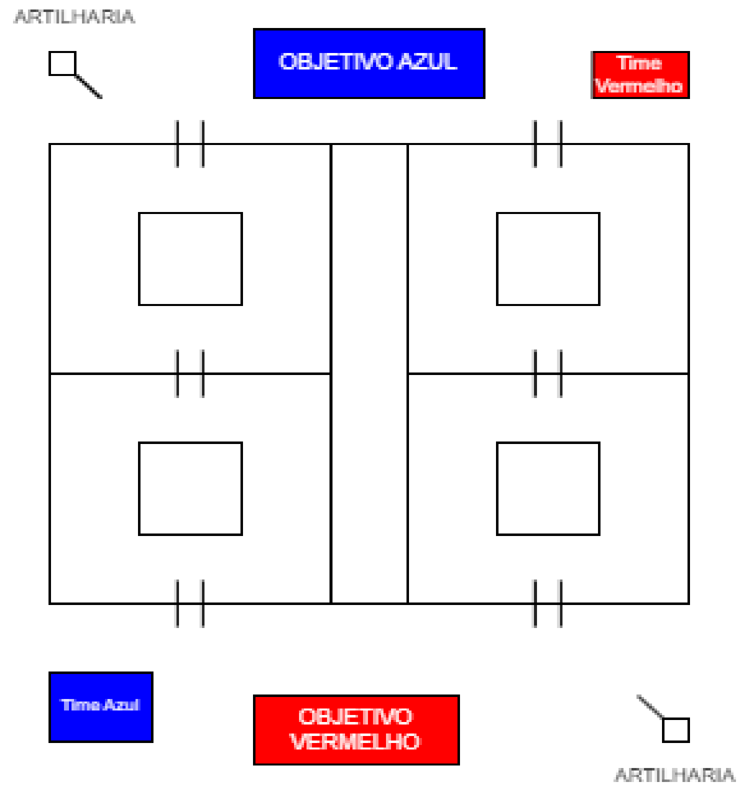
O protótipo terá três atores principais: o jogador, o bot-antitrapaça e o agente. O jogador é o personagem controlado por uma pessoa, sendo ele de qualquer um dos times. Enquanto o bot-antitrapaça é um personagem do jogo, porém controlado pela IA. O bot-antitrapaça simulará um jogador trapaceiro e tem todas as valências e objetivos que o jogador possui. E por final, o agente é o componente que analisa os comportamentos de ambos atores (jogador e bot-antitrapaça).

5.1.2 Mecânicas

Os comandos dos jogadores seguirão o modelo padrão para esse gênero de jogo, utilizando as teclas 'W', 'S', 'A', 'D' para movimentação, botão esquerdo do mouse para atacar o inimigo e a tecla 'E' para interagir com as caixas. Para o personagem controlado pela IA, será utilizado o componente *NavMeshAgent* da *Unity*. Esse componente próprio da *Engine* é atrelado a um personagem móvel e permite executar consultas espaciais, como encontrar caminho (*pathfinfing*) e testes de confiabilidade, ou seja, a classe disponibiliza propriedades para movimentação para o computador (FILHO, 2019).

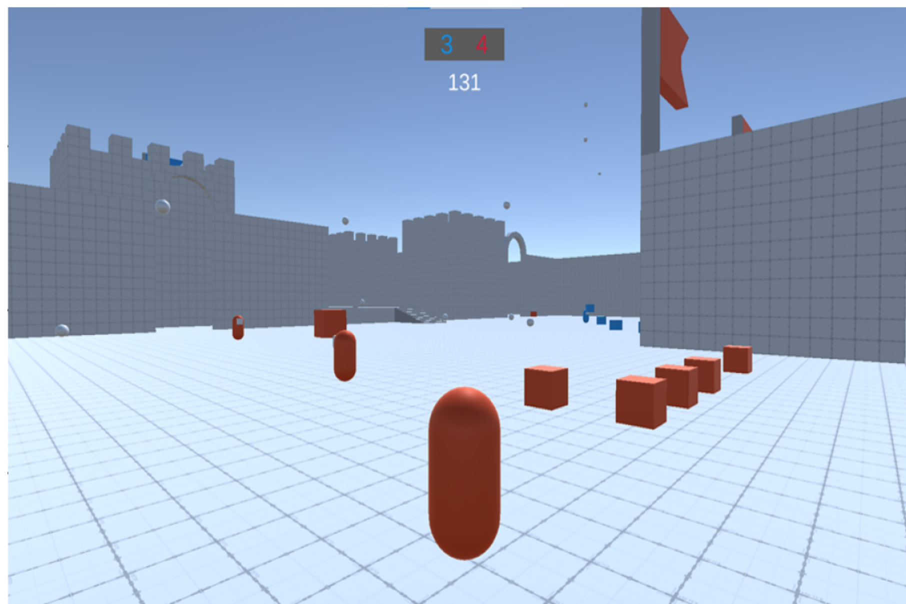
As interações dos bots-antitrapaça seguirão comportamentos definidos por uma série de verificações. No momento do surgimento dentro do jogo, ele fará a verificação de objetos mais próximos a ele, tanto um objeto do tipo inimigo ou do tipo caixa. Se encontrar um desses objetos, ele, então, se moverá até esse objeto e realizará uma ação de acordo com esse objeto. Se for do tipo inimigo, o bot-antitrapaça, atacará o inimigo até esse largar a caixa. Dessa forma, será novamente avaliado o objeto mais próximo e retornará o objeto do tipo caixa. Nessa condição o bot-antitrapaça irá até a caixa e fará a coleta dela. De posse da caixa, ele irá para o ponto de destino e largará a caixa para marcar a pontuação de sua equipe.

Figura 8 – Mapa do jogo Caçador de Recompensa



Fonte: Anticheat System Based on Reinforcement Learning Agents in Unity; Mihael, Lukas; 2022

Figura 9 – Conceito inicial do jogo sem materias e texturas



Fonte: Anticheat System Based on Reinforcement Learning Agents in Unity; Mihael, Lukas; 2022

5.1.3 Regras do Jogo

As principais regras são por conta de colisão entre jogador e caixa, e jogador com o inimigo. O jogador somente poderá atacar o inimigo se estiver tocando nele e o inimigo não causará dano ao jogador. No momento da eliminação do inimigo, a caixa é imediatamente largada no chão. Poderá ter mais de um jogador atacando o mesmo inimigo, mas somente um jogador poderá carregar a caixa. Na ação do jogador com a caixa, ele somente poderá coletar a caixa se estiver colidindo com ela e não pode possuir outra caixa, ou seja, só é possível carregar uma caixa de cada vez. De maneira análoga, o jogador somente poderá largar a caixa se estiver no local de objetivo de seu time.

5.2 MODELAGEM DO JOGO E PROCESSO DE DESENVOLVIMENTO DO SOFTWARE

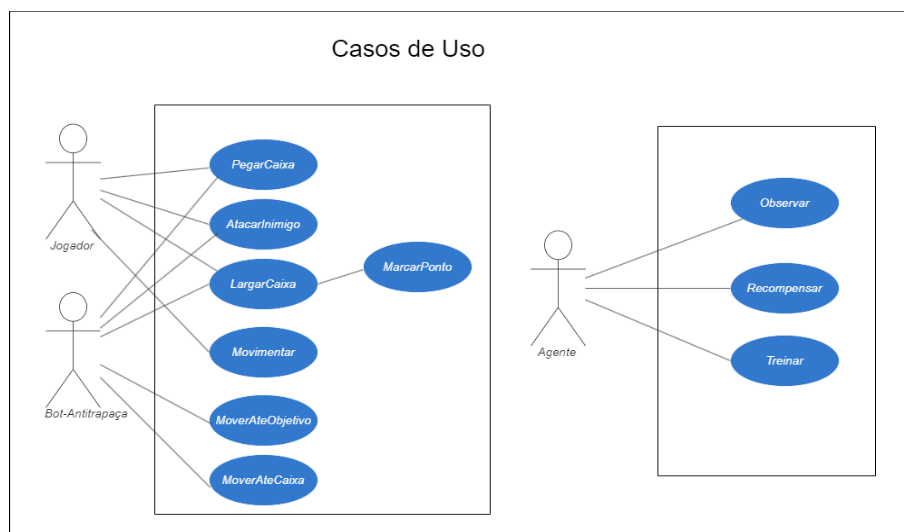
Para realizar a modelagem comportamental, utilizou-se diagramas UML de casos de uso para determinar as maneiras que o jogador interage com o jogo. Como o protótipo dispõe de bibliotecas externas para alcançar o objetivo da avaliação de detecção da trapaça, criou-se o diagrama de componentes para demonstrar como o sistema do jogo na *Unity* se comunica com a biblioteca *ML-Agents*. Para melhor visualizar o comportamento do sistema, fez-se uso do diagrama de estados. Por fim, para cada caso de uso elaborou-se diagramas de sequências. Na parte estrutural, criou-se o diagrama de classes para representar os aspectos estáticos.

5.2.1 Diagrama de Casos de Uso

O diagrama de casos de uso é uma técnica para captar os requisitos funcionais de um sistema. Eles servem para descrever as interações típicas entre os usuários de um sistema e o próprio sistema, fornecendo uma narrativa sobre como o sistema é utilizado (FOWLER, 2005).

Alguns casos de uso são compartilhados em ambos jogador e bot-antitrapaça. Essas ações serão detalhadas no diagrama de sequência. A única diferença no caso de uso do jogador (humano) refere-se as ações de relacionada a movimentação, já que a movimentação do bot-antitrapaça é realizada através do componente *NavMesh* em que verifica qual estado o bot-antitrapaça está para realizar a ação de movimentar até a caixa ou inimigo. Os comportamentos compartilhadas são de pegar e largar a caixa, e atacar o inimigo. Dessas ações que são realizadas pelo ator (jogador ou bot-antitrapaça), o componente *ML-Agents* observa os eventos para determinar se aquele é humano ou não. Como é utilizado um agente para realizar a avaliação da trapaça, o agente deve previamente ser treinado. A avaliação do jogador realizada pelo agente resulta em recompensas para o agente, seja positiva, negativa ou neutra. Essa interação pode ser visualizada na Figura 10.

Figura 10 – Diagrama de casos de uso



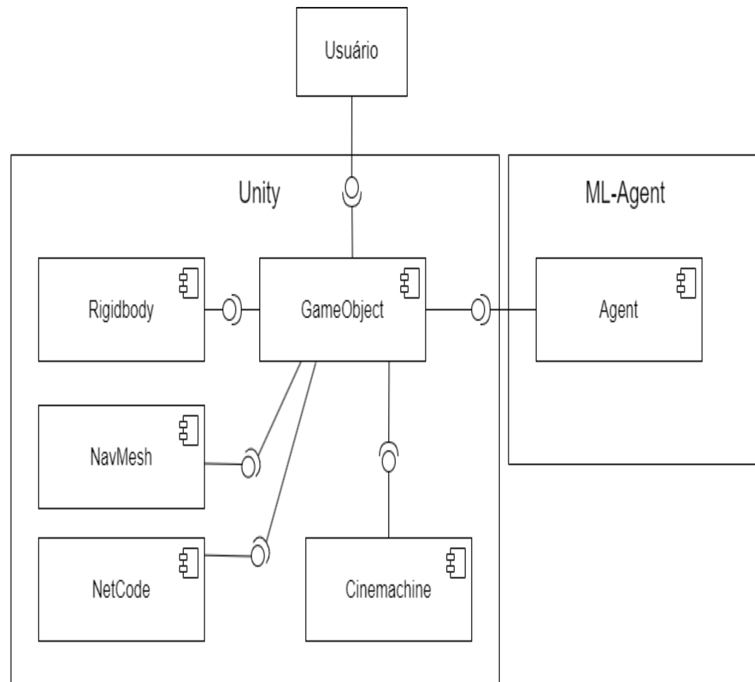
Fonte: Próprio autor

5.2.2 Diagrama de Componente

O diagrama de componente mostrado na Figura 11 apresenta a interação dos principais componentes utilizados na *Unity* como *GameObject*, o componente de física(*Rigidbody*) e o *NavMeshAgent* bem como a integração com componente ML-Agents. De modo geral, o usuário controlará um *GameObject*, termo utilizado para determinar qualquer objeto na cena de jogo, sendo jogador humano ou um bot-antitrapaça. Se o jogador for humano ele fará suas ações de movimentação através do componente de física do próprio software da *Unity*. Em contra partida, o bot-antitrapaça utilizará sistema de *NavMeshAgent*, componente próprio da *Engine* para movimentação automatizada. Esse componente é atribuído a qualquer *Gameobjects* móvel e permite que a movimentação seja controlada por IA. A implementação do *multiplayer* ocorreu através da utilização de um componente não nativo da *Unity* chamado *NetCode*. Nele, é importado todas funcionalidades para manipular dados de rede. O pacote *Cinemachine* permite a utilização de métodos mais sofisticados para gerenciamento da câmera de jogo.

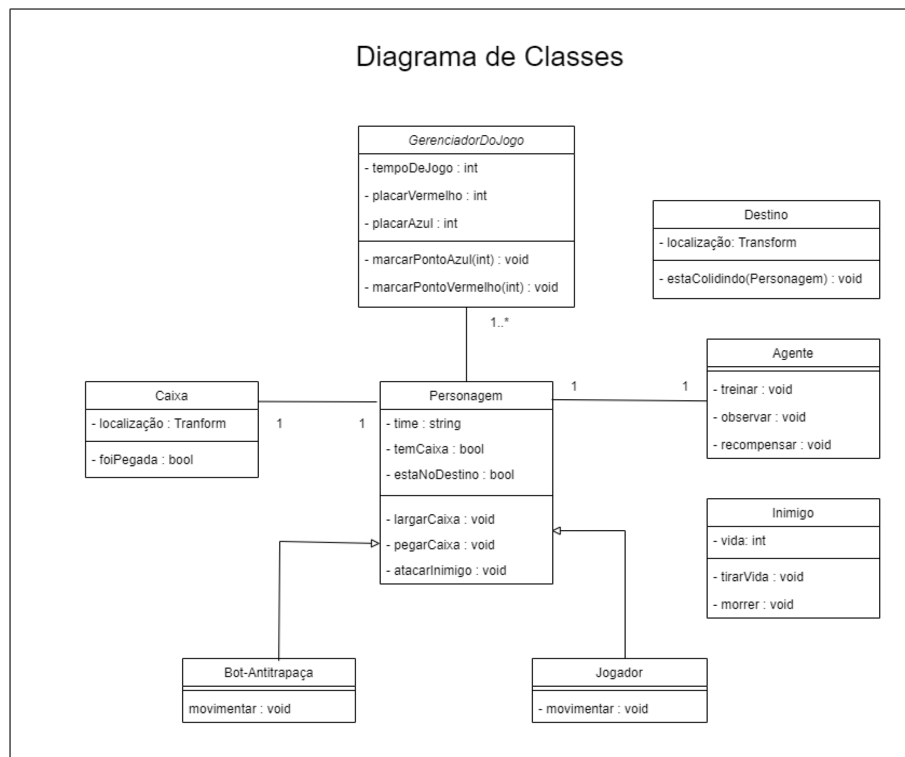
O *Gameobject* também se conecta com o *ML-Agent* através da classe *Agent* dessa biblioteca. Isso é realizado por meio da inserção de scripts no *Gameobject* em questão. Por exemplo, se for inserido um script referente ao componente *ML-Agent* (classe herdando a classe Agent) a algum *Gameobject*, esse objeto automaticamente terá todas funcionalidades herdadas da biblioteca como, treinamento, aprendizado, o sistema de recompensa entre outras funções. Sendo assim, componente Agente se comunicará diretamente com o *Gameobject* (jogador ou bot-antitrapaça) para realizar todas as ações necessárias no protótipo.

Figura 11 – Diagrama de componentes



Fonte: Próprio autor

Figura 12 – Diagrama de classes para projeto



Fonte: Próprio autor

5.2.3 Diagrama de Classes

O diagrama de classes, Figura 12, destaca as classes usadas para a modelagem do projeto. Obviamente, muitas ações que ocorrem dentro do sistema são gerenciados pela própria *Unity* e não estão dispostos nesse diagrama, como o comportamento do componente *NavMeshAgent* ou o componente de Física.

A classe "GerenciadorDoJogo"é responsável pela manutenção do jogo de forma geral. Nela é implementado o tempo de jogo e marcação de pontos para ambos os times. Apenas terá dois métodos: "marcarPontoAzul"e "marcarPontoVermelho"que não retornarão valores.

A classe "Personagem"é uma classe genérica para implementação das classes "Jogador"e "BotAntitrapaca". O "Personagem"apresenta os atributos do time e uma referência para classe "Caixa". Além disso, a classe possui atributos booleanos para verificar se a instância da classe está em posse de uma caixa e também se está no destino. Ainda nessa classe, há método responsável para a ação de ataque e as ações de pegar e largar a caixa. Na subclasse "Jogador"são implementados os métodos responsáveis pela movimentação. A subclasse "BotAntitrapaca"apenas implementa a movimentação controlada pelo componente *NavMesh*.

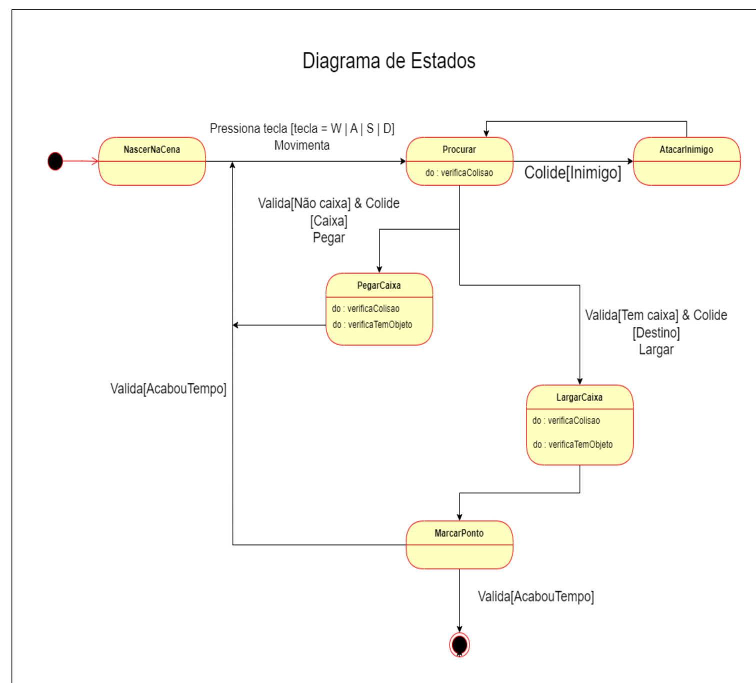
O "Destino"apenas tem a referência da localização e um método para validar a colisão com os jogadores. A classe "Caixa"também possui somente a localização e um método para disparar quando é coletada por um jogador. A classe "Inimigo"apenas tem um atributo referente a vida e dois métodos. O método "tirarVida"para remover um ponto de vida quando atacado e o método "morrer"quando sua vida chegar a zero. A classe "Agente"tem as funcionalidades herdadas da biblioteca *ML-Agent* que compreendem a ação de treinar IA, bem como, observar os comportamentos dos jogadores e recompensar o próprio agente quando esse consegue diferenciar um humano de um bot-antitrapaça.

5.2.4 Diagrama de Estados

O sistema terá alguns estados específicos e pode ser verificada na Figura 13. Inicialmente, o jogo começa com o surgimento dos personagens na cena no estado "Nascer na Cena"e a partir disso, com a ação de pressão da tecla de movimentação (W, S, A, D), o jogador se movimentará pelo cenário 3D. Nesse estado, ele entrará no estado de "Procurar", seja uma caixa ou inimigo. Nesse estado é validado um evento de colisão com o inimigo ou com a caixa. Se o personagem estiver próximo o bastante do inimigo é transicionado para o estado de atacar o inimigo. Esse estado fica em constante laço até que o inimigo seja derrotado. Quando o inimigo é eliminado, ele larga a caixa no chão como recompensa. Nesse momento, é avaliada a colisão do jogador com a caixa. Se o jogador não possuir outra caixa, ele pode coletá-la. Tendo a posse da caixa, é alterado o estado novamente para "Procurar"em que o personagem se movimentará até colidir com o respectivo local específico para depositar a caixa. Nesse estado também é validado se o personagem possui uma caixa e se está no local correto. Se essa condição for

atendida, o jogador largará a caixa e marcará o ponto para seu time.

Figura 13 – Diagrama de Estados



Fonte: Próprio autor

5.2.5 Diagrama de Sequência

Os diagramas de sequência auxiliam na compreensão dos comportamentos dos elementos determinados nos casos de uso. Nele é demostrado como as diferentes partes do sistema interagem para realizar um determinado processo. Nesse protótipo, têm-se os casos de uso para o personagem do jogo: atacar e marcar ponto. O agente possui apenas o caso de uso para avaliar e de recompensa.

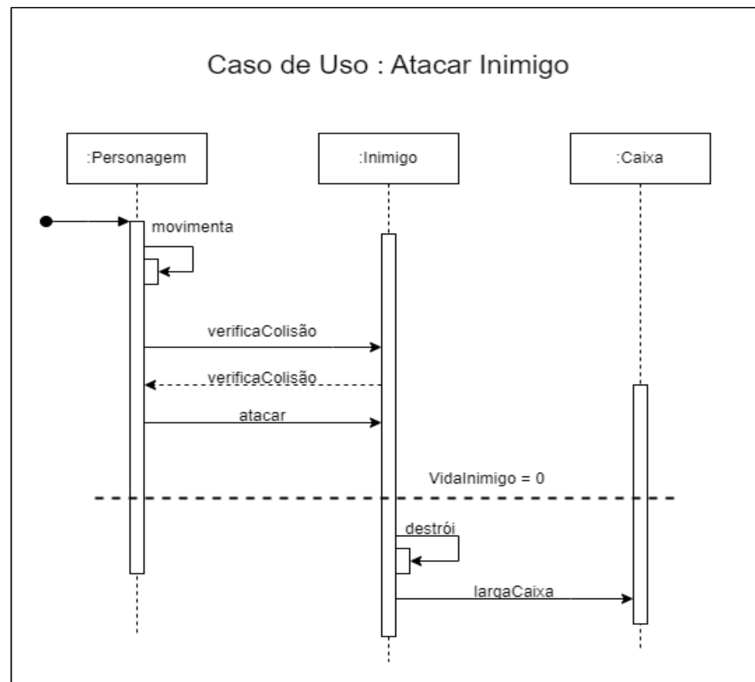
5.2.5.1 Diagrama de Sequência para Atacar

A sequência para caso de uso Atacar Inimigo pode ser vista na Figura 14 e é de fundamental importância para o ciclo do jogo. Para que o objetivo seja alcançado é obrigatório que o jogador derrote o inimigo. Nesse caso de uso, o personagem se movimenta pelo cenário até encontrar um inimigo. A instância do personagem verifica a colisão com o inimigo e, se essa for confirmada, o jogador pode atacá-lo. Após derrotado, o inimigo larga a caixa.

5.2.5.2 Diagrama de Sequência para Marcar Ponto

No diagrama de sequência para o caso de uso Marcar Ponto, tem-se o fluxo principal do personagem dentro do jogo como é apresentada na Figura 15. Essa sequência compreende

Figura 14 – Diagrama de Sequência para o caso de uso Atacar



Fonte: Próprio autor

desde a coleta da caixa até o ato de depositar a caixa no destino. No início do ciclo, o personagem desempenhará a ação de movimentação representado pela auto-referência do "movimenta". Após, a instância do personagem realiza validação com o objeto caixa e se, confirmada a colisão, ele pode coletar o objeto, seguindo as regras discutidas no diagrama de estado. A próxima sequência do personagem volta a ser a movimentação pelo cenário e após a validação do local de objetivo. Dentro da área de depósito da caixa, ele pode inserir o objeto no local e o evento de marcar ponto é disparado. As caixas que forem depositadas se tornam indisponíveis para nova coleta.

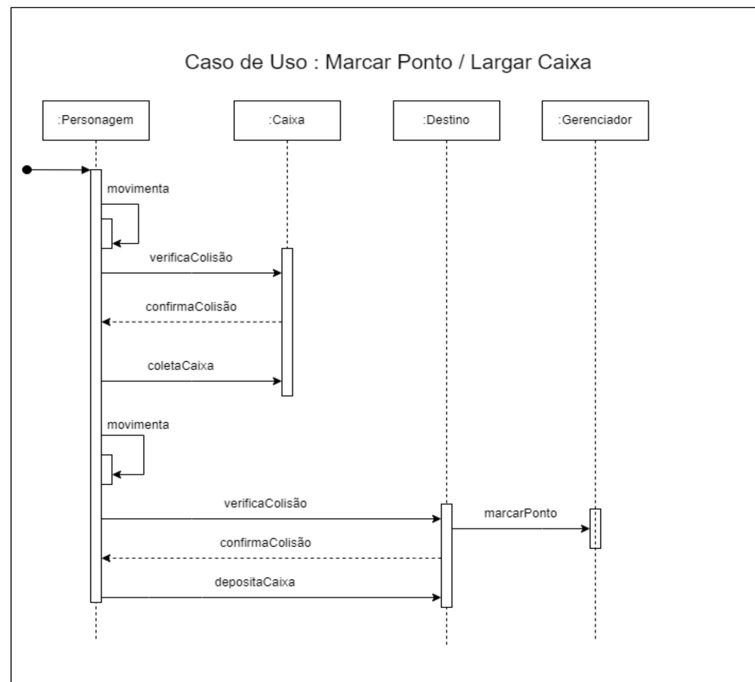
5.2.5.3 Diagrama de Sequência para Avaliar e Recompensa

Nos casos que são implementados para o Agente, tem-se apenas a avaliação que é realiza pelo *ML-Agent* sobre os personagens do jogo e o subsequente evento de recompensa que é disparado quando o agente toma qualquer decisão como pode ser visto na Figura 16. Na sessão de implementação do Agente é discutido de que forma é realizado o evento de recompensa.

5.3 IMPLEMENTAÇÃO DO AGENTE E INTEGRAÇÃO COM O JOGO

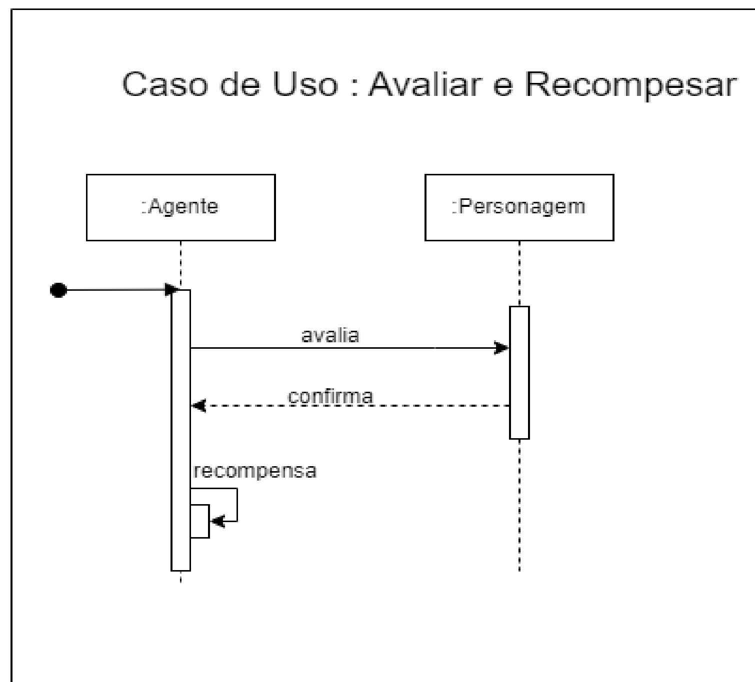
A implementação dos agentes é realizada através da biblioteca de código aberto *ML-Agents* baseada em *Pytorch*. Bots antitrapaça e jogadores reais são implementados de maneira que seja possível o reconhecimento dentro do servidor, entretanto, o agente somente recebe informações dos dados de movimentação dos personagens não sabendo se é um jogador ou é

Figura 15 – Diagrama de Sequencia para o caso de uso Marcar Ponto



Fonte: Próprio autor

Figura 16 – Diagrama de Sequencia para o caso de Avaliar e Recompensa



Fonte: Próprio autor

um bot-antitrapaça. De acordo com essas informações deve aprender a diferenciar por conta própria.

Os agentes utilizam ações com numeração discreta, sendo 0 (verdadeiro), 1 (neutro) e 2 (falso). Se o agente marcar um personagem como 0, então conclui-se que esse personagem é

um bot-antitrapaça. Se a decisão for 1, o personagem não é bot-antitrapaça ou humano e se a decisão for 2, o personagem é humano. Para que o agente consiga aprender a diferenciação é necessário sistema de recompensa. O objetivo principal do agente é maximizar a recompensa. Se o agente reconhece um personagem humano como humano, ele recebe uma recompensa positiva. De forma contrária, se identificar um humano como bot-antitrapaça, ele recebe um recompensa negativa e vice-versa. Se o agente decide permanecer neutro, não é computado recompensa.

5.3.1 Configuração de treinamento

A configuração é necessária para começar o procedimento de testes. Para otimizar o aprendizado do agente, é utilizado um arquivo de configuração *Ain't Markup Language* (YAML)(Não é uma Linguagem de Marcação) contendo informações e parâmetros para o aprendizado por reforço. É de suma importância que as informações estejam corretas para que não haja inconsistências no objetivo do projeto. O arquivo YAML utilizado no protótipo pode ser visto na Figura 17. Os parâmetros utilizados para o projeto serão discutidos na seção 7.1.2, em que é descrito as características de cada parâmetro.

Figura 17 – Exemplo de arquivo YAML

```
E: > TCC > config > ! agent.yaml > {} behaviors
1   behaviors:
2     Agent:
3       trainer_type: ppo
4       hyperparameters:
5         batch_size: 32
6         buffer_size: 10240
7         learning_rate: 0.0003
8         beta: 0.005
9         epsilon: 0.2
10        lambd: 0.95
11        num_epoch: 3
12        learning_rate_schedule: linear
13      network_settings:
14        normalize: false
15        hidden_units: 128
16        num_layers: 2
17        vis_encode_type: simple
18      reward_signals:
19        extrinsic:
20          gamma: 0.99
21          strength: 1.0
22      max_steps: 500000
23      time_horizon: 64
24      summary_freq: 10000
```

Fonte: Próprio autor

5.3.2 Decisões do Agente

Para cada personagem(jogador ou bot-antitrapaça) dentro do cenário de jogo, há uma descrição da decisão do agente. É realizado um cálculo entre as decisões positivas do agente

divido pelas as últimas 3000 decisões. Desse total é validado se o personagem em questão é um bot-antitrapaca ou humano. A porcentagem desse cálculo é tomado a cada segundo e o valor médio é calculado no final. A descrição somente é visível no lado do servidor por isso não é mostrado aos jogadores (LUKAS; TOMICIC; BERNIK, 2022). Por exemplo, se nas últimas 3000 decisões, o agente marcou 2700 decisões positivas para o personagem do tipo bot-antitrapaça, então há 90% de chance desse jogador estar trapaceando.

5.4 CONCLUSÃO SOBRE MODELAGEM DO PROTÓTIPO

Para uma implementação bem-sucedida é vital o processo de modelagem do projeto. Observou-se o detalhamento do jogo junto às suas mecânicas e regras. Explicou-se como será a jogabilidade do protótipo bem como os elementos determinantes para alcançar o objetivo do protótipo. A criação dos diagramas definem a visão geral como será realizado a codificação do jogo, as funcionalidades de cada elemento do jogo e a integração da biblioteca *ML-Agent* com o protótipo.

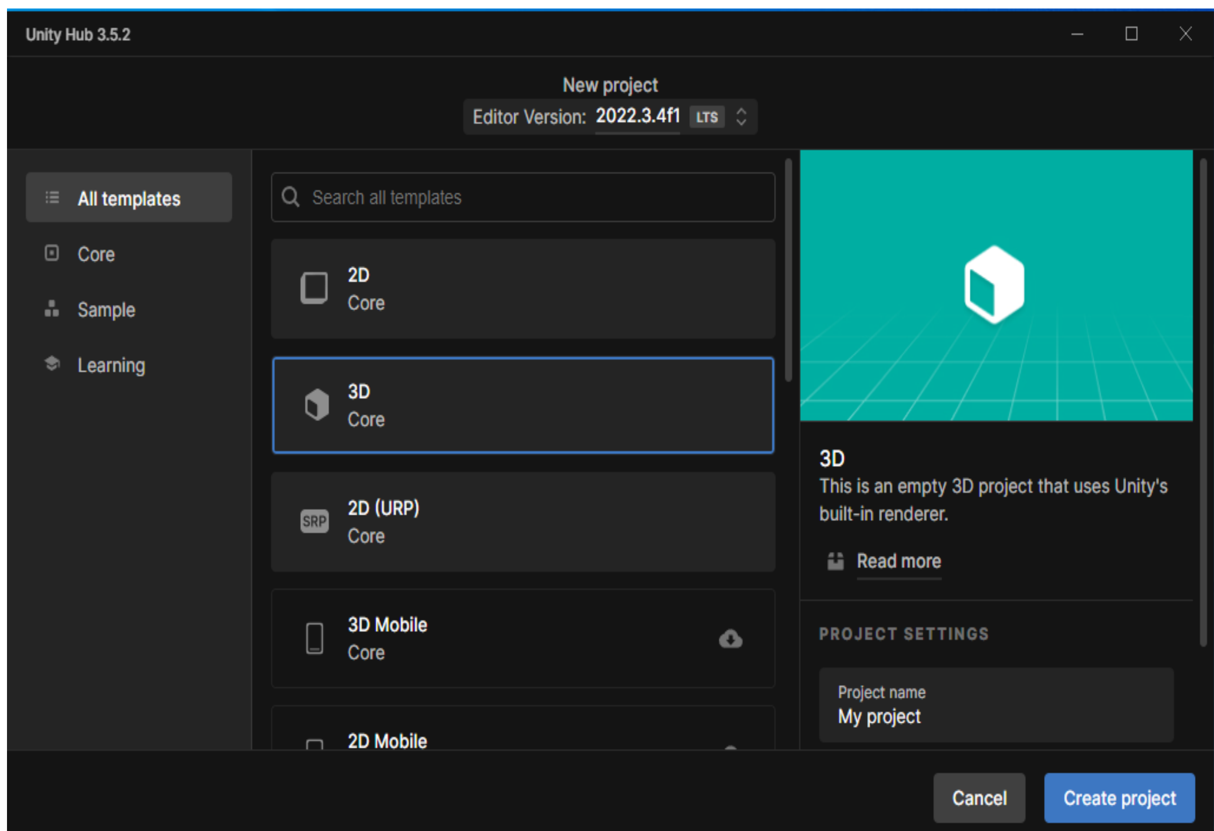
6 IMPLEMENTAÇÃO DO PROTÓTIPO

Nesse capítulo é demonstrada a implementação do protótipo do jogo, desde a configuração do ambiente até a elaboração dos algoritmos das classes criadas. É identificado elementos importantes dentro da *Unity*, como *Prefabs* e Scripts. Uma parte importante da implementação, deu-se na adição do *multiplayer* através do componente *Netcode*.

6.1 PRAPARANDO O AMBIENTE

Desenvolveu-se o jogo no modo 3D da *Unity*. Para isso, na inicialização do aplicativo, é selecionado o modo 3D para que o editor da *Unity* seja inicializado com as ferramentas necessárias para o desenvolvimento 3D (Figura 18). Uma vez dentro do aplicativo, pode-se incluir bibliotecas externas que farão parte do produto final de jogo. A *Engine* provê uma extensa lista de componentes que podem ser inseridos diretamente pelo gerenciador de pacotes (*Package Manager*) do aplicativo. Para demostração do protótipo, tomou-se alguns pacotes que não fazem parte da versão padrão do software, entre eles estão o *ML-Agents*, AI *Navigation*, *Cinemachine*, *Netcode for GameObjects* e *TextMeshPro*.

Figura 18 – Tela Inicial da Engine da Unity



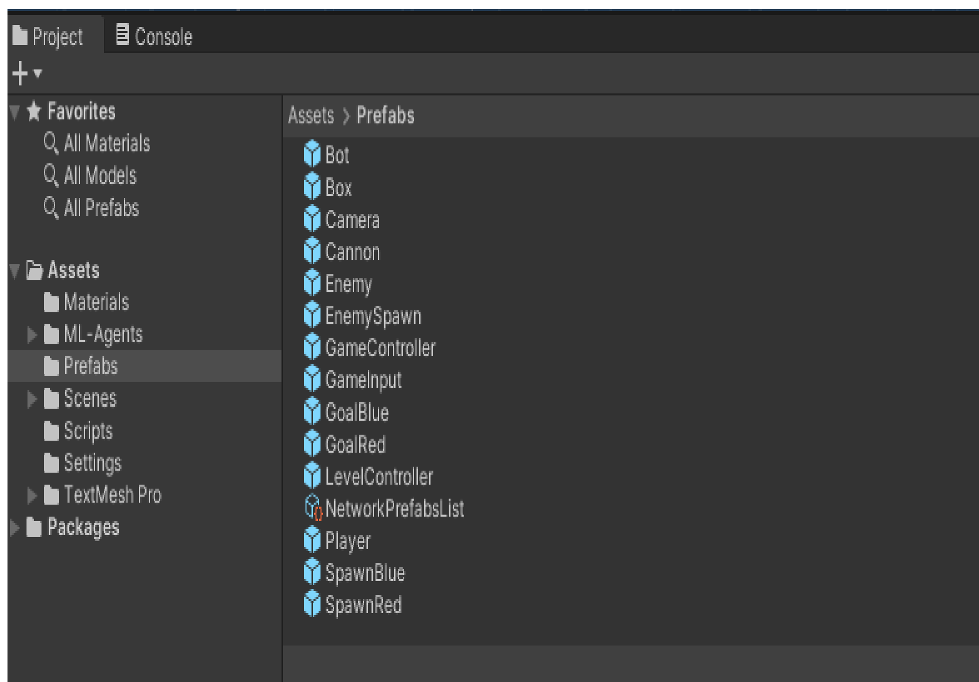
Fonte: Próprio autor

O próximo passo da implementação foi a criação do mapa (cenário) de jogo. Utilizou-se objetos de forma retangular para construir as paredes e o chão em que os jogadores farão suas ações. Nessa fase, também criou-se os pontos de nascimento dos jogadores e inimigos, bem como o local de objetivo, lugar esse, em que os jogadores após coletar a caixa, depositarão ela para receber pontuação. Tanto os locais de nascimento quanto os de objetivo são adicionados com um material de coloração para identificar a qual time pertence. Para o ponto de nascimento dos inimigos, criou-se um *GameObject* (objeto de jogo genérico da *Unity*) invisível apenas para marcar a posição onde deverá surgir o inimigo.

6.1.1 Elementos do jogo

Para criação dos elementos centrais do jogo, utilizou-se o sistema de pré-fabricamento da *Unity* (*Prefabs*). Esse sistema permite a criação, configuração do *GameObject* e armazena todos os componentes que fazem parte desse objeto. Na Figura 19 é mostrado a lista de *Prefabs* que foram utilizados na implementação. Alguns deles, como *GameInput* e *LevelController* não possuem correspondente visual na cena de jogo mas são responsáveis por gerenciar outros comportamentos do jogo.

Figura 19 – Lista de Prefabs utilizadas na implementação

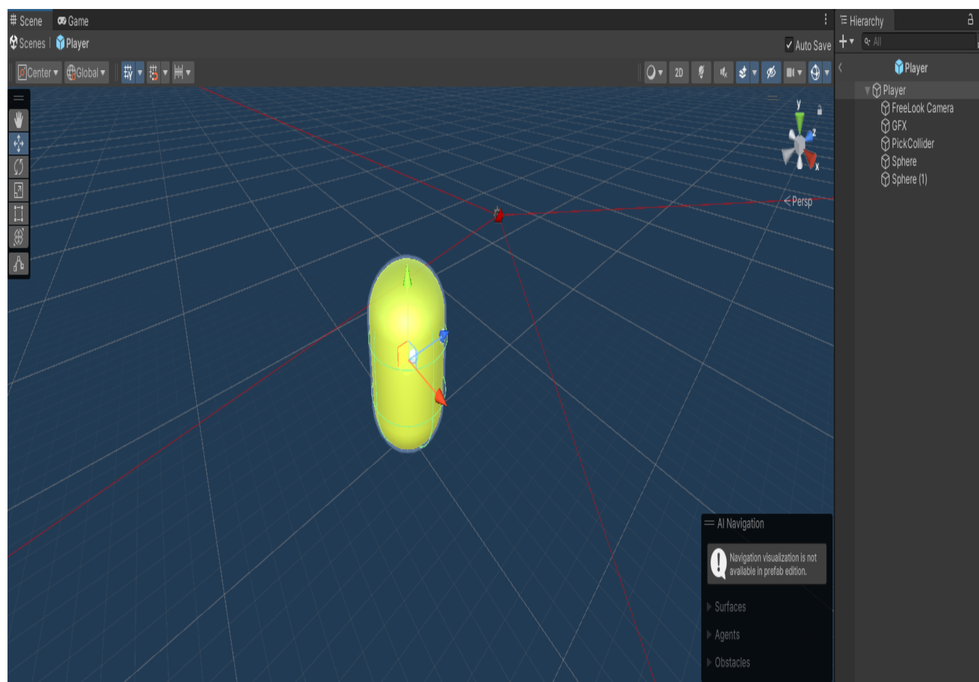


Fonte: Próprio autor

O *Prefab* do jogador pode ser visualizado na Figura 20 e nele observa-se que é adicionado um material genérico na cor amarela e somente é atualizado esse componente (*MeshRenderer*) em tempo de execução para a cor do time correspondente.

O protótipo apresenta somente duas cenas. A tela de jogo em que serão executadas

Figura 20 – *Prefab* do jogador



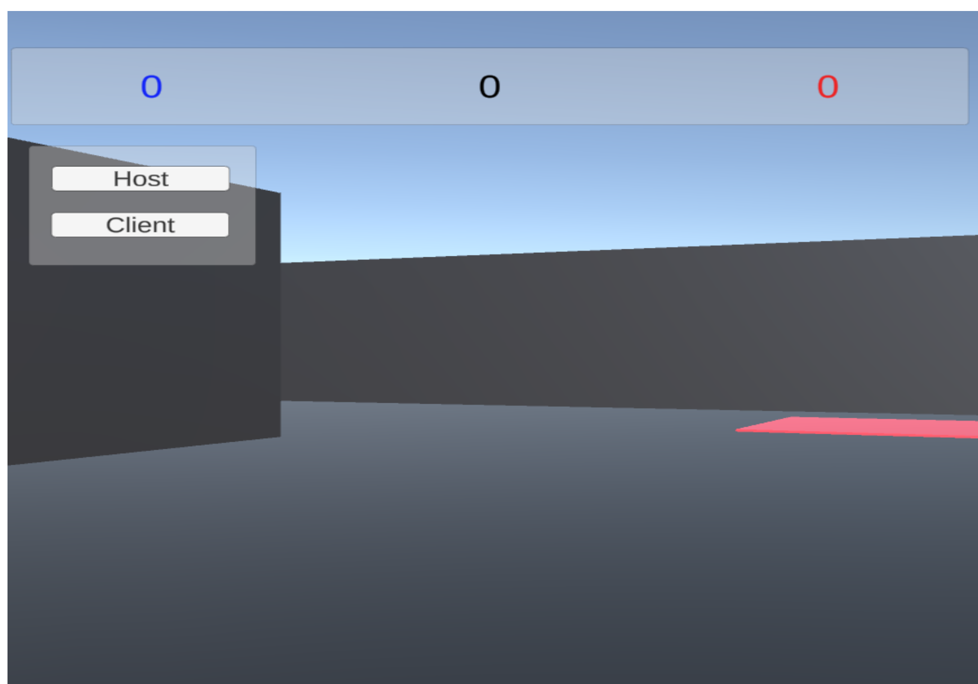
Fonte: Próprio autor

as ações dos jogadores e a tela de reinício do jogo como pode ser visto na Figura 21. Antes de começar a execução do jogo é visualizada uma tela para seleção do jogo no modo *Host* ou *Client* demonstrada na Figura 22. Essas duas opções determinam como o jogador daquela sessão se comportará.

Figura 21 – Tela de Reinicio do jogo

Fonte: Próprio autor

Figura 22 – Tela inicial do Jogo com opção de *Host* e *Client*



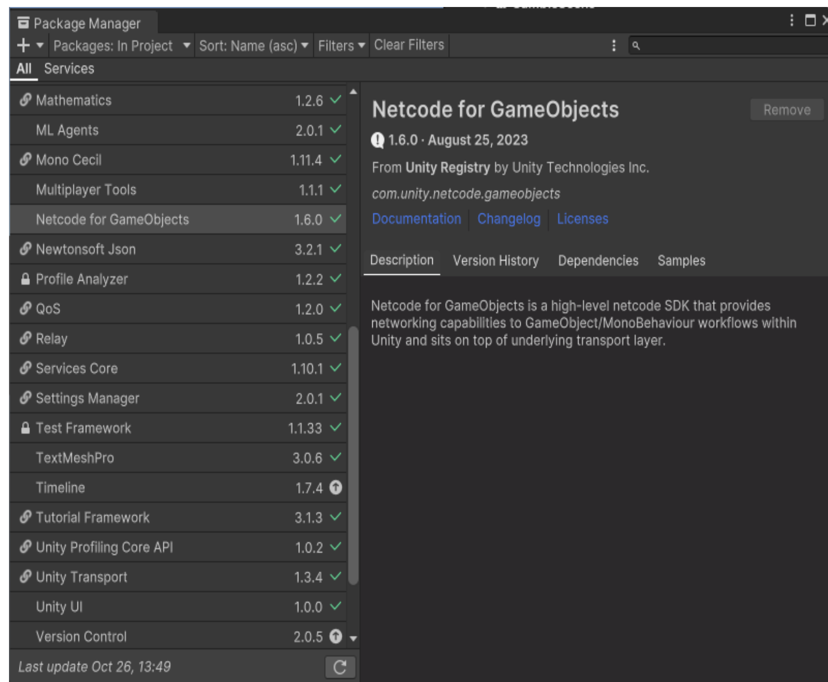
Fonte: Próprio autor

6.1.2 *Multiplayer*

Realizou-se a implementação da funcionalidade de *multiplayer* através da biblioteca *Netcode*. Tal biblioteca abstrai a lógica de rede e permite transmitir dados pelas sessões da rede para todos os jogadores conectados. As bibliotecas externas à *Unity* podem ser incluídas através do gerenciador de pacotes da *Unity* como mostra a Figura 23.

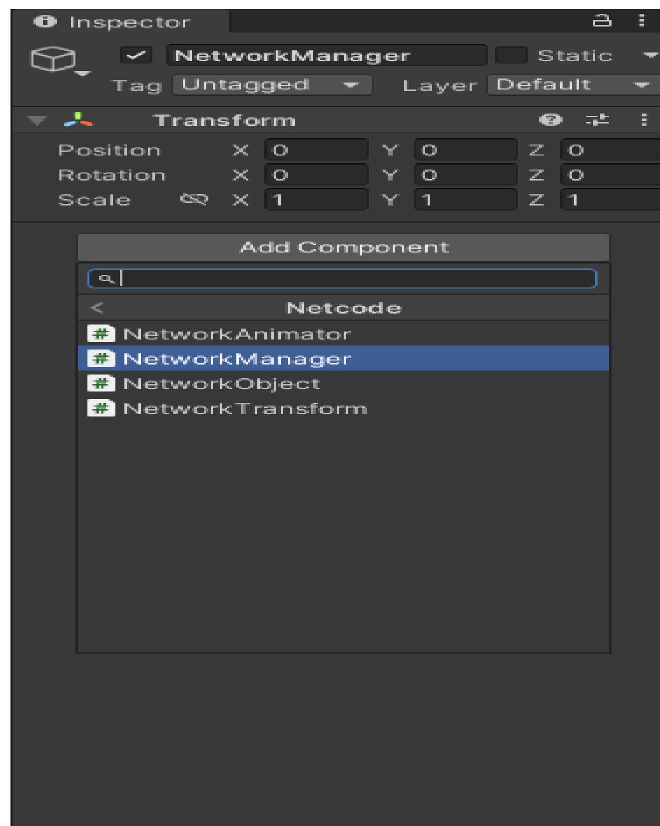
Inicialmente, deve-se criar um elemento do tipo *NetworkManager* que ficará responsável por gerenciar os processos de rede do jogo. Para isso, cria-se um objeto vazio na cena de jogo e é adicionado a esse objeto um componente *NetworkManager* observado na Figura 24. A partir disso, tem-se a possibilidade de configurar um *Prefab* genérico (campo *Player Prefab*) que nascerá cada vez que um cliente se conectar ao servidor mostrado na Figura 25. No contexto do protótipo, selecionou-se o *Prefab* do jogador (*Player*) para ser instanciado no momento da conexão do cliente. Os objetos que precisam ser sincronizados com todos os clientes devem ser incluídos na lista *Network Prefabs List* do componente *NetworkManager*. Essa lista é criada através da criação do elemento *ScriptObject* de mesmo nome e nela são adicionados os objetos que serão sincronizados na rede. A Figura 26 apresenta a lista utilizada na implementação do *multiplayer*. É importante ressaltar que todos os objetos nessa lista precisam obrigatoriamente possuir um componente de *NetworkObject*. Tal componente indica que esse objeto se refere a um objeto de rede e tem a capacidade de chamar todos os métodos de rede. Para sincronização da posição do objeto, é necessário o componente *ClientNetworkTransform*. Ambos componentes são incluídos através do editor da *Unity*.

Figura 23 – Netcode no gerenciador de pacotes da *Unity*



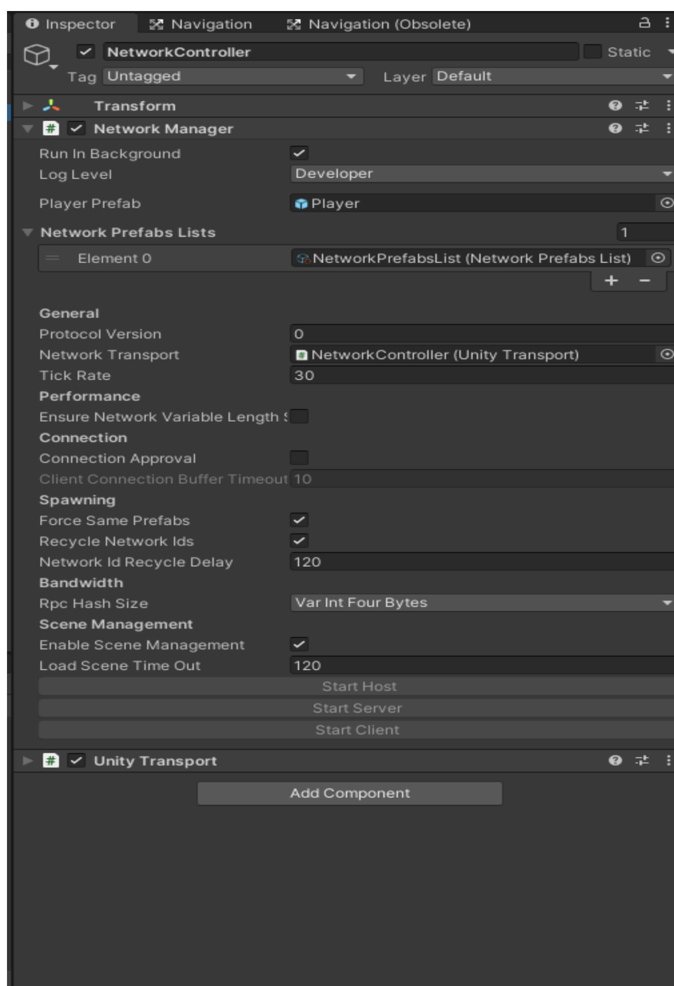
Fonte: Próprio autor

Figura 24 – Adição do componente *NetworkManager* no objeto



Fonte: Próprio autor

Figura 25 – Configuração do Player Prefab no componente *NetworkManager*



Fonte: Próprio autor

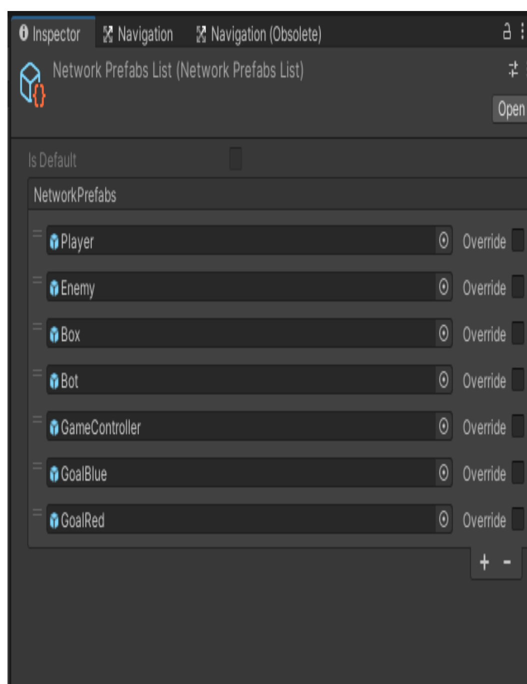
6.2 CODIFICAÇÃO DAS CLASSES E SCRIPTS

A maior parte da lógica programática dentro da *Unity* é coordenada através de Scripts. Esses podem ser responsáveis por controlar elementos, disparar eventos, criar efeitos gráficos entre outros comportamentos. Serão demonstrados isoladamente os Scripts centrais criados para o protótipo. A lista de Scripts criados para o protótipo pode ser visualizada na Figura 27.

6.2.1 Jogador

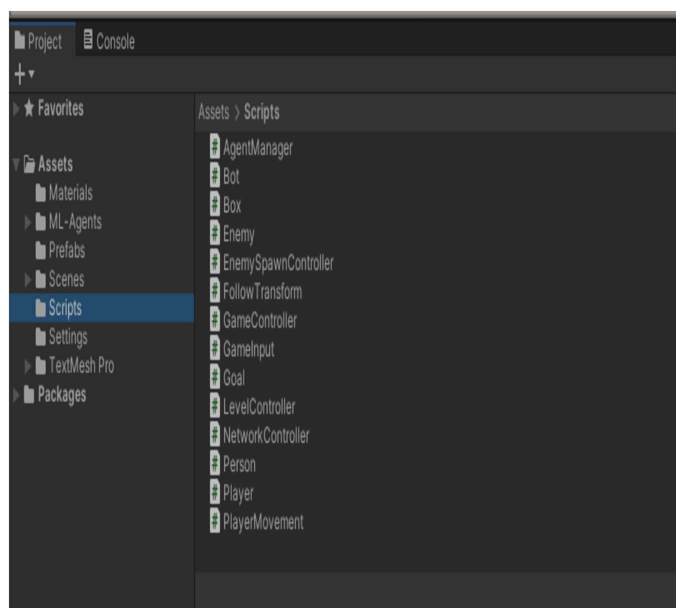
A classe jogador herda os atributos da classe *Personagem* em que é atribuído o time do jogador e os métodos de interações tanto com a caixa quanto com o inimigo. No trecho de código Algoritmo 1 apresenta as variáveis do Jogador. Para atributos que necessitam que sejam referenciados através do editor, a *Unity* fornece uma estrutura para serializar o atributo (*SerializeField*). Com essa funcionalidade, a classe não precisa instanciar o objeto em questão mas apenas utilizar a referencia nos devidos lugares. A linha 1 referencia a classe de entrada (*GameInput*) de movimentação do jogador. A seguir tem-se o objeto *pickupObject* que é respon-

Figura 26 – Lista *Network Prfabs List*



Fonte: Próprio autor

Figura 27 – Lista de Scripts gerados para o protótipo



Fonte: Próprio autor

sável de ter a posição onde a caixa ficará após coletada pelo jogador. O *cinemacheFreeLook* na linha 3 referencia a câmera que ficará atrás do jogador. A variável *velocity* guarda a velocidade em que o jogador se movimentará. Na linha 5, tem a referência do Material que será atualizada dinamicamente no jogador. Esses materiais são as cores azul e vermelho para representar cada time. Após, observa-se a referência do componente de *MeshRenderer* para alterar os materiais em tempo de execução. Esse componente é próprio de todos os *GameObjects* dentro do jogo

e com a manipulação dele é possível realizar alterações da parte gráfica do objeto. Na linha 9, tem-se a referência ao *GameController* em que é gerenciado o comportamento do jogo em âmbito geral. Na linha 10 e 12, guardam componentes próprios da *Unity*. O componente *Rigidbody* é responsável pela física dos objetos e a classe *Vector3* é utilizada para representar uma estrutura de posições e direções em um ambiente 3D.

Algoritmo 1 – Criação das variáveis da classe Jogador

```
1  [SerializeField] private GameInput gameInput;
2  [SerializeField] GameObject pickupObject;
3  [SerializeField] CinemachineFreeLook cineMachine;
4  [SerializeField] int velocity;
5  [SerializeField] Material mBlue, mRed;
6  [SerializeField] private MeshRenderer _meshRenderer;
7  GameObject goal;
8  Box localBox;
9  GameController gameController;
10 Rigidbody myRigidbody;
11 bool isAttacking = false, hasBox = false;
12 Vector3 lastInteractionDir;
```

Fonte: O Autor (2023)

Observa-se no código do Algoritmo 2 os métodos de construção da classe do Jogador. O método *Awake()* representa o primeiro método invocado quando o objeto é instanciado dentro do jogo. Nele é inserido o componente do *MeshRenderer* do próprio elemento pré-fabricado a variável da classe. É necessário essa atribuição para que o objeto instanciado tenha autorização para modificar elementos dentro desse componente como por exemplo, a alteração do tipo Material. O método *OnNetworkSpawn()* também é um método chamado na criação do objeto, entretanto apenas em objetos que fazem parte da rede da biblioteca *Netcode*. Para questões hierárquicas, o método *Awake()* é chamado antes do que o *OnNetworkSpawn()*. Nesse método, como o jogador é um objeto de rede e todos os objetos de rede da mesma classe compartilham o mesmo código, é realizado a validação se esse script que está sendo executado é de fato propriedade do objeto criado. Se essa condição for verdadeira, então são atribuídas as variáveis *gameConbtroller*, *cineMachine* e *rigidbody* as correspondentes classes. Na linha 7, é chamado o método para criar o jogador na cena com seus materiais correspondentes e posição de nascimento relativa ao cenário. O segundo argumento dessa chamada é referente ao identificador único de cada elemento da rede. Com esse atributo é validado se o jogador deve ser do time azul ou vermelho. E a última atribuição é para definir que a câmera deve ser apenas da instância criada.

Algoritmo 2 – Métodos de inicialização da classe Jogador

```
1  private void Awake() {
2      _meshRenderer = GetComponentInChildren<MeshRenderer>();
3  }
```

```
4  public override void OnNetworkSpawn() {
5      if (IsOwner) {
6          gameController = FindObjectOfType(typeof(GameController));
7          gameController.CreatePlayer(this, OwnerClientId);
8          myRigidbody = GetComponent<Rigidbody>();
9          cineMachine.Priority = 1;
10     } else {
11         cineMachine.Priority = 0;
12     }
13 }
```

Fonte: O Autor (2023)

O comportamento do jogador é gerenciado através dos métodos *HandleMovement()* e *HandleInteractions()* os quais podem ser observados no Algoritmo 3. Esses métodos são chamados nas funções reservadas da *Unity*, *Update()* e *FixedUpdate()* onde ocorrem os laços principais do jogo. Eles se diferem na ocorrência do laço, uma vez que o *FixedUpdate* atualiza o jogo em intervalo constante e o *Update* atualiza em todo o quadro. Essa diferenciação é importante porque a movimentação do jogador se atualizada no *Update* ocorre problemas nas colisões e na física do jogo. Por causa disso, escolheu-se realizar a atualização da movimentação no *FixedUpdate*. De maneira similar a criação da classe, esses comportamentos devem ser executados apenas pelo proprietário da classe e por isso é inserido a validação na linha 2 e 9. Enquanto no *FixedUpdate* verifica a movimentação, o método *Update* valida o material dos jogadores e as interações do jogadores com outros elementos do jogo. A linha 4, *SetMaterialClientRpc()* tem o objetivo de comunicar a todos os clientes do jogo o material que o *host* possui e assim os clientes podem atualizar na sua sessão o componente *MeshRenderer* do *host*.

O método de movimentação apenas verifica se há alguma entrada do jogador chamando a instância do *gameInput* e com isso adiciona uma força no componente de física *Rigidbody* que é guardada através da variável *rigidBody*. Já a implementação das interações na linha 18 verifica as ações que o jogador pode tomar. Na linha 25, é realizado a validação de qual objeto o jogador está olhando e com isso valida se é um inimigo ou uma caixa, linhas 27 e 38 respectivamente. Se o jogador estiver de frente ao inimigo, ele pode atacá-lo com entrada do mouse da esquerda. A linha 32 atualiza a lista de inimigos na classe *Bot*. Se o jogador estiver de frente a uma caixa, e interagir com ela com a entrada 'E' do teclado e não estiver com uma caixa já coletada, ele obterá essa caixa e colocará na posição do *pickupObject*. Como essa ação deve ser atualizada para os jogadores, é necessário que seja realizada através de procedimentos remotos *Remote Procedure Call* (RPC). Essas chamadas RPC (Chamadas de Procedimentos Remotos) são enviadas para o servidor e esse, se encarregará de atualizar em todas as instâncias essa ação. Por fim, se a variável *goal* estiver populada, é devido ao jogador se encontrar no ponto de objetivo. Dessa forma, se possuir uma caixa, ele pode colocá-la nesse local com entrada 'E' do teclado. Quando colocar a caixa no objetivo, é chamado a classe de *gameController* e adicio-

nado pontuação ao time. Feito esse fluxo, é limpado todas as referencias do *goal*, *hasBox* e o *localBox* para que o laço continue de maneira correta.

Algoritmo 3 – Métodos do comportamento do jogador

```
1  private void Update() {
2      if (!IsOwner) return;
3      if (IsHost) {
4          SetMaterialClientRpc(team);
5      }
6      HandleInteractions();
7  }
8  void FixedUpdate() {
9      if (!IsOwner) return;
10     HandleMovement();
11 }
12 void HandleMovement() {
13     if (gameInput.GetMovementVectorNormalized().magnitude > 0.1f) {
14         Vector3 relPos = transform.position - vc.transform.position;
15         myRigidbody.AddForce(relPos * velocity);
16     }
17 }
18 public void HandleInteractions() {
19     float interactDistance = 3f;
20     Vector3 moveDir = (transform.position - vc.transform.position)
21     * velocity;
22     if (moveDir != Vector3.zero) {
23         lastInteractionDir = moveDir;
24     }
25     if (Physics.Raycast(transform.position, lastInteractionDir,
26     out RaycastHit raycast, interactDistance)) {
27         if (raycast.transform.TryGetComponent(out Enemy enemy)) {
28             if (Input.GetMouseButtonDown(0)) {
29                 isAttacking = true;
30                 enemy.TakeDamage();
31                 //atualiza lista de inimigos no bot
32                 Bot bc = FindAnyObjectByType(typeof(Bot)) as Bot;
33                 if (bc != null) {
34                     bc.AppendEnemiesList();
35                 }
36             }
37         }
38         if (raycast.transform.TryGetComponent(out Box box)) {
39             if (Input.GetKeyDown(KeyCode.E) && !hasBox) {
40                 localBox = box;
41                 goal = null;
42                 PickupBoxServerRpc(localBox.getBoxNetworkObject());
43                 hasBox = true;
```

```
44              }
45          }
46      }
47      if (goal != null) {
48          if (Input.GetKeyDown(KeyCode.E) && hasBox) {
49              goal.TryGetComponent(out Goal goalObject);
50              DropBoxServerRpc(localBox.getBoxNetworkObject(),
51              goalObject.getGoalNetworkObject());
52              gameController.AddScore(this.team);
53              goal = null;
54              hasBox = false;
55              localBox = null;
56          }
57      }
58  }
```

Fonte: O Autor (2023)

Quando o jogo é *multiplayer* é necessário realizar atualizações tanto do lado cliente quando no servidor. Para isso, utilizando a biblioteca *Netcode*, é possível executar esses processos utilizando modelo de chamadas remotas RPC. O código Algoritmo 4 apresenta os métodos utilizados nessa comunicação. Do lado do cliente para atualizar os dados para o servidor é utilizado a diretiva *ServerRpc* acima do método que implementará a funcionalidade e esse método obrigatoriamente deve conter o sufixo *ServerRpc*. Na diretiva é possível passar um parâmetro de propriedade *(RequireOwnership = false)* no qual libera para todos os clientes chamarem esse método. Nesse caso, o método *SetMaterialServerRpc* é responsável por atualizar o material do cliente no servidor. Da mesma forma os métodos *PickupBoxServerRpc* e *DropBoxServerRpc* atualizam o processo do jogador pegar e largar a caixa. Esse últimos métodos recebem como argumento um tipo especial de objeto *NetworkObjectReference* que detêm os comportamentos de rede do objeto. Com isso é possível manipular os dados do objeto passado no argumento, como por exemplo, sua posição. De maneira similar, o servidor pode enviar os dados para todos os clientes utilizando a diretiva *ClientRpc* e implementando métodos com sufixo *ClientRpc* como na linha 12 *SetMaterialClientRpc* em que é implementada a funcionalidade de atualização do material do jogador que está servindo de *host* no jogo.

Algoritmo 4 – Métodos de atualização do servidor no multiplayer

```
1
2  [ServerRpc(RequireOwnership = false)]
3  private void SetMaterialServerRpc(string team) {
4      if (team == "red") {
5          _meshRenderer.material = mRed;
6      } else {
7          _meshRenderer.material = mBlue;
8      }
9      SetMaterialClientRpc(team);
```

```
10  }
11  [ClientRpc]
12  private void SetMaterialClientRpc(string team) {
13      if (team == "red") {
14          _meshRenderer.material = mRed;
15      } else {
16          _meshRenderer.material = mBlue;
17      }
18  }
19  [ServerRpc(RequireOwnership = false)]
20  private void PickupBoxServerRpc(NetworkObjectReference nor) {
21      nor.TryGet(out NetworkObject boxNetworkObject);
22      Box box =  boxNetworkObject.GetComponent<Box>();
23      box.setBoxParent(pickupObject);
24  }
25  [ServerRpc(RequireOwnership = false)]
26  private void DropBoxServerRpc(NetworkObjectReference nor,
27      NetworkObjectReference norg) {
28      nor.TryGet(out NetworkObject boxNetworkObject);
29      Box box = boxNetworkObject.GetComponent<Box>();
30      norg.TryGet(out NetworkObject goalNetworkObject);
31      GameObject goal = goalNetworkObject.GetComponent<Goal>().gameObject;
32      box.setBoxParent(goal);
33      box.tag = "Untagged";
34      box.DestroyBox();
35  }
```

Fonte: O Autor (2023)

A *Unity* disponibiliza métodos reservados para colisão entre elementos como mostra o código Algoritmo 5. O código método *OnCollisionEnter* é disparado todas as vezes que o objeto colide com outro que possui um colisor. A implementação desse método apenas verifica o time da instância do jogador e também a *tag* do objetivo colidido. Se ambas as validações forem verdadeiras, a variável *goal* é atribuída com o valor do objeto colidido. A *tag* do objeto é inserida no elemento pré-fabricado no editor da *Unity*.

Algoritmo 5 – Métodos de colisão da classe Jogador

```
1
2  private void OnCollisionEnter(Collision _collision) {
3      if (this.team == "red" && _collision.gameObject.tag == "redGoal") {
4          goal = _collision.gameObject;
5      }
6      if (this.team == "blue" && _collision.gameObject.tag == "blueGoal") {
7          goal = _collision.gameObject;
8      }
9  }
```

Fonte: O Autor (2023)

6.2.2 Bot Anti-trapaça

De maneira semelhante ao visto na implementação do jogador, a codificação da classe *Bot* tem uma variável serializada e outras que referem-se a outras classes como visto no Algoritmo 6. Na linha 1, o *pickupObject* guarda um *GameObject* invisível para coleta da caixa. O componente *NavMeshAgent* na linha 4 tem o objetivo da utilização do elemento de navegação realizada pela IA dentro do jogo. Para que seja implementada essa funcionalidade é necessário importar a biblioteca *AI Navigation*. Após instalada essa biblioteca, é possível adicionar o componente *NavMesh Surface* que determinará a área que a IA percorrerá. Um ponto importante para destacar é que os objetos que implementam esse componente de navegação devem obrigatoriamente colidir com o terreno do *NavMesh* para que funcione de maneira correta. A linha 8 apenas guarda uma lista de inimigos que estão na cena de jogo. Como os inimigos mortos renascem após um determinado tempo, é necessário atualizar a referência dos inimigos que os *bots* precisam se dirigir.

Algoritmo 6 – Variáveis inicializadas da classe Bot

```
1 [SerializeField] GameObject pickupObject = null;
2 GameObject enemyCollider = null, boxCollider, goalCollider, goal = null;
3 Enemy closerEnemy = null;
4 private NavMeshAgent agent;
5 private Box boxController;
6 private bool hasBox = false, goingToBox = false, isAttacking = false;
7 private GameController gc;
8 List<Enemy> enemiesList = new List<Enemy>();
```

Fonte: O Autor (2023)

Na *Unity*, como visto nos métodos *Awake* e *OnNetworkSpawn* na classe do jogador, o método *Start* é chamada apenas na criação da classe. Nesse contexto da classe *Bot*, é codificado para que a instância busque a referência do *GameController*, guarde o componente *NavMeshAgent* para poder usá-lo na atualização da posição do *bot* e também adiciona os inimigos na lista como visto na linha 4 do Algoritmo 7.

Algoritmo 7 – Métodos de inicialização da classe Bot

```
1 void Start() {
2     gc = FindObjectOfType(typeof(GameController)) as GameController;
3     agent = GetComponent<NavMeshAgent>();
4     AppendEnemiesList();
5 }
```

No Algoritmo 8 pode-se observar a lógica utilizada para atualizar a lista de inimigos na cena de jogo. Na linha 2 é buscado dentro do jogo todas as referências dos inimigos que estão

ativos. Após essa busca, é realizada a verificação para identificar se a instância do *bot* possui algum inimigo. Se a lista ainda possuir inimigos, então ela é limpada e são adicionados novos inimigos na lista como mostra a linha 7. Utilizou-se essa estratégia porque se não houvesse a limpeza da lista, duplicaria as instâncias dos inimigos e com isso causaria um comportamento indesejado.

Algoritmo 8 – Métodos para adicionar inimigos na lista

```
1   public void AppendEnemiesList() {
2       Enemy[] enemies = FindObjectsOfType(typeof(Enemy)) as Enemy[];
3       if (enemiesList.Count != 0) {
4           enemiesList.Clear();
5       }
6       foreach (var item in enemies) {
7           enemiesList.Add(item);
8       }
9   }
```

O laço principal da classe é realizada no *FixedUpadte* (Algoritmo 9) em que é realizada a validação para adicionar novos inimigos na lista (linha 2) e toda a lógica de movimentação e ações do *bot*. Na linha 5 é realizada a verificação se o *bot* possui a caixa, e se retornar falsa então essa validação, o *bot* se movimentará até o próximo inimigo da lista na linha 12. Chegando próximo do inimigo, o *bot* terá a ação de atacá-lo e assim se movimentará até a caixa (linha 15) poderá coleta-lá na linha 22. Disposto com a caixa, é atualizada a direção do *bot* fazendo com que o objetivo desse seja o ponto de depósito da caixa para cada time. Uma vez depositado a caixa, é chamado o gerenciador do jogo da classe *GameController* e adicionado um ponto para o time correspondente.

Algoritmo 9 – Métodos para atualizar posição do bot

```
1   void FixedUpdate() {
2       if (enemiesList.Count == 2) {
3           AppendEnemiesList();
4       }
5       if (!hasBox) {
6           goalCollider = null;
7           if (enemyCollider == null && !goingToBox) {
8               if (closerEnemy == null) {
9                   GetCloserEnemy();
10                  enemiesList.Remove(closerEnemy);
11              }
12              agent.destination = closerEnemy.transform.position;
13          }
14          if (boxCollider == null && boxController != null) {
15              agent.destination = boxController.transform.position;
16          }
17          if (boxCollider != null) {
```

```
18                var pickup = new GenericCollider() {
19                    Position = pickCollider.transform.position
20                };
21                hasBox = true;
22                PickupBoxServerRpc(pickup);
23            }
24        }
25    }
```

6.2.3 Enemy

A classe do inimigo tem a responsabilidade de contabilizar o dano sofrido, destruir sua própria instância quando a vida chegar a zero e depois disso, criar a caixa. Como mostrado na linha 4 do Algoritmo 10, o método *TakeDamage* tem o objetivo de realizar a lógica de tomar dano. Esse método é chamado pela classe Personagem, a qual, as classe *bot* e Jogador herdam. Se a variável chegar a zero então é chamado outra função utilizando o procedimento remoto na linha 8. Essa implementação é devido ao fato de que todos na cena de jogo precisam atualizar esses dados do inimigo. Para objetos de rede, há um método nativo da biblioteca *Netcode* que elimina da cena e do servidor o objeto como no exemplo da linha 16. Após a eliminação do inimigo é chamado novo procedimento remoto para criar a caixa.

Algoritmo 10 – Métodos da classe do inimigo

```
1  [SerializeField] public float life;
2  [SerializeField] GameObject box;
3  private bool isDead = false;
4  public void TakeDamage() {
5      life--;
6      if (life <= 0) {
7          isDead = true;
8          DestroyEnemyServerRpc();
9      }
10 }
11 public bool isEnemyDead() {
12     return isDead;
13 }
14 [ServerRpc(RequireOwnership = false)]
15 private void DestroyEnemyServerRpc() {
16     NetworkObject.Despawn();
17     InstanciateBoxServerRpc();
18 }
19 [ServerRpc(RequireOwnership = false)]
20 private void InstanciateBoxServerRpc() {
21     GameObject instance = Instantiate(box, transform.position,
22     transform.rotation);
23     instance.GetComponent<NetworkObject>().Spawn(true);
```

```
24 }
```

6.2.4 FollowTransform

No Algoritmo 11 é demostrada a implementação da classe auxiliar *FollowTransform* e proporciona uma forma de manipular a posição da caixa. Um modo bastante comum de realizar esse comportamento é alterando o parentesco do objeto, colocando ele como filho de outro objeto na cena. Entretanto, essa forma de alteração de parentesco de objetos de rede não é trivial de ser implementada e dessa forma, escolheu-se o processo de utilizar uma classe auxiliar para modificar a posição da caixa quando coletada.

Algoritmo 11 – Métodos da classe auxiliar FollowTransform

```
1  public void setTargetTransform(Transform targetTransform) {
2      this.targetTransform = targetTransform;
3  }
4  private void LateUpdate() {
5      if (this.targetTransform == null) {
6          return;
7      }
8      transform.position = this.targetTransform.position;
9      transform.rotation = this.targetTransform.rotation;
10 }
```

6.2.5 GameController

O *GameController* tem a função de gerenciar o jogo, criando jogadores e *bots*, guardando o tempo decorrido e também a pontuação dos times. As variáveis utilizadas são mostradas no Algoritmo 12. Nessa parte, é criada as variáveis de rede (linhas 1, 4 e 8). Essas variáveis são compartilhadas com todos os clientes do servidor e por isso torna-se uma alternativa para guardar os dados de pontuação e de tempo de jogo. Essas variáveis só podem ser alteradas pelo servidor mas é possível que todos dentro da rede possam ler as informações dessas variáveis, para isso, na inicialização da variável é passado o primeiro argumento de valor inicial e o segundo argumento de permissão de leitura.

Algoritmo 12 – Variáveis da classe GameController

```
1  private NetworkVariable<float> serverTimer =
2      new NetworkVariable<float>(0,
3      NetworkVariableReadPermission.Everyone);
4  private NetworkVariable<int> scoreRedNum =
5      new NetworkVariable<int>(0,
6      NetworkVariableReadPermission.Everyone,
7      NetworkVariableWritePermission.Owner);
8  private NetworkVariable<int> scoreBlueNum =
```

```
9       new NetworkVariable<int>(0,
10      NetworkVariableReadPermission.Everyone,
11      NetworkVariableWritePermission.Owner);
```

O laço de atualização da classe verifica constantemente a pontuação dos times e também o tempo de jogo. Esse comportamento é demostrado no Algoritmo 13. Na primeira iteração do laço são criados os *bots* chamando o método *CreateBots*. Ainda no laço principal, há a validação do tempo decorrido do jogo e se o valor chegar a zero, é chamado o *LevelController* para alterar a cena de jogo. Para criação dos *bots*, é validado o número de jogadores humanos configurados no inicio da execução do jogo e subtraído pelo número total de jogadores. Se no início da execução é configurado com um total de quatro jogadores e três humanos, esse método criará apenas um *bot* na cena de jogo. Os *bots* precisam ser instanciados a partir de procedimentos remotos para que o servidor e todos os clientes recebam os dados do *bot*.

Algoritmo 13 – Método principal e de criação do jogador e bots na classe GameController

```
1
2  public void FixedUpdate() {
3      scoreRed.text = scoreRedNum.Value.ToString();
4      scoreBlue.text = scoreBlueNum.Value.ToString();
5      if (IsServer) {
6          if (NetworkManager.ConnectedClients.Count == humanPlayers) {
7              if (!hasCreatedBots) {
8                  CreateBots();
9              }
10             serverTimer.Value += Time.deltaTime;
11         }
12         if (serverTimer.Value >= 10000) {
13             levelController.ExitGame();
14         }
15     }
16     timerTxt.text = serverTimer.Value.ToString("0");
17 }
```

6.2.6 LevelController

No Algoritmo 14 é implementado a lógica de manipulação das cenas do jogo. Quando é chamado o método *ExitGame*, é interrompido o servidor e carregado a nova cena de reinício. Já nessa cena, se o jogador clicar no botão de reinício, o jogo volta para a tela inicial.

Algoritmo 14 – Métodos para reiniciar cena na classe LevelController

```
1  public void ExitGame() {
2      Cursor.lockState = CursorLockMode.Confined;
3      NetworkController n = new NetworkController();
4      n.stopHost();
```

```
5        SceneManager.LoadScene("GameoverScene");
6    }
7    public void EnterGame() {
8        SceneManager.LoadScene("SampleScene");
9    }
```

6.2.7 NetworkController

O objetivo da Classe *NetworkController* (Algorimo 15) é inicializar o serviço de rede no jogo. Tanto o método *startHost* e *startClient* estão vinculados aos botões vistos na primeira tela do jogo na Figura 22.

Algoritmo 15 – Métodos para inicializar componentes de rede

```
1    public void startHost() {
2        NetworkManager.Singleton.StartHost();
3    }
4    public void startClient() {
5        NetworkManager.Singleton.StartClient();
6    }
7    public void stopHost() {
8        NetworkManager.Singleton.Shutdown();
9    }
```

6.3 CONSIDERAÇÕES DA IMPLEMENTAÇÃO

A codificação é parte principal da implementação de qualquer projeto na computação. Na tentativa de deixar uma base para trabalhos futuros, tentou-se descrever todo o processo de configuração bem como a explicação da maior parte da lógica do projeto. Para a implementação do protótipo houve grande pesquisa sobre componentes tanto nativos quanto não nativos da *Unity*. Um bom exemplo dessa pesquisa foi a implementação do modelo multijogador o qual se tornou uma tarefa não trivial, levando mais tempo que o planejado para implementá-lo. Embora tenha havido bloqueios durante a codificação dessa funcionalidade, o saldo foi bastante positivo visando conhecimento adquirido.

7 ML-AGENT NO JOGO

A viabilidade do protótipo é relacionada à integração da biblioteca *ML-Agent* com o jogo desenvolvido. Nesse contexto, é demonstrado nesse capítulo as etapas necessárias para instalação dos componentes e os comandos executados para utilização da biblioteca. Tem-se a criação e ativação de um ambiente virtual *Python* para execução dos comandos de treinamento e para visualização de gráficos. Ainda nesse capítulo é realizado a fase de testes, bem como a análise dos resultados.

7.1 INTEGRANDO O AGENTE

A seção a seguir apresentará a integração do agente da biblioteca *ML-Agents* ao protótipo implementado. Além da inclusão da biblioteca do *ML-Agents* através do gerenciador de pacotes, são necessários alguns passos anteriores de instalação de bibliotecas externas para que o *ML-Agents* se comporte de maneira correta. O monitoramento dessa fase será realizada através do software TensorBoard[1], no qual é possível observar o nível de recebimento das recompensas.

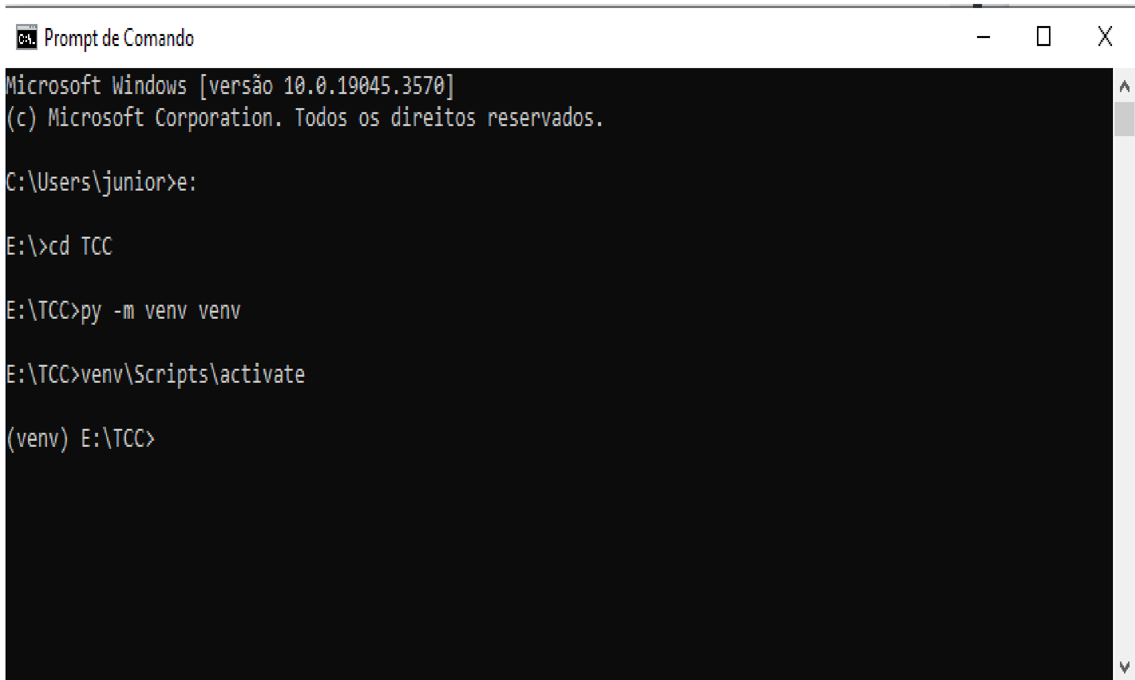
7.1.1 Instalação de pacotes necessários

A ferramenta *ML-Agents* contém diversos recursos que compreendem a utilização da IA. Os processos de IA são realizados por meio de componentes executados em *Python* e por isso, é necessário a instalação na máquina que implementará o treinamento em ambiente *Python*. No momento da implementação do protótipo, a versão recomendada para instalação é *Python* 3.10. No objetivo de manter a implementação clara, criou-se na pasta do projeto um ambiente virtual *Python* como mostrado na Figura 28. No comando *py -m venv venv*, tem-se a execução do *python* (py) e é passado o argumento -m para criação do módulo e o tipo do módulo (venv) de ambiente virtual com o nome do ambiente que nesse caso, escolheu-se também venv. Após isso, pode ser executado o ambiente virtual acessando o arquivo *activate*. Pode-se observar que a última linha do *prompt* de comando está entre parênteses o nome do ambiente virtual que está sendo executado.

Após o processo da criação do ambiente, deve-se instalar e atualizar o pacote *pip* do *Python*. O pacote *pip* é um gerenciador de pacotes do *Python* que auxilia na instalação e atualização de novos pacotes. O comando para realizar a atualização do *pip* é *py -m pip install –upgrade pip*. Com o pacote devidamente instalado, é possível instalar o pacote *mlagents* com o comando *pip install mlagents*. Em alguns casos pode ocorrer problemas de versão do *mlagents*. Depois da instalação concluída, pode-se verificar se o pacote está funcional executando o comando *mlagents-learn –help* em que é descrito a documentação de ajuda na utilização da

[1] https://www.tensorflow.org/tensorboard?hl=pt-br

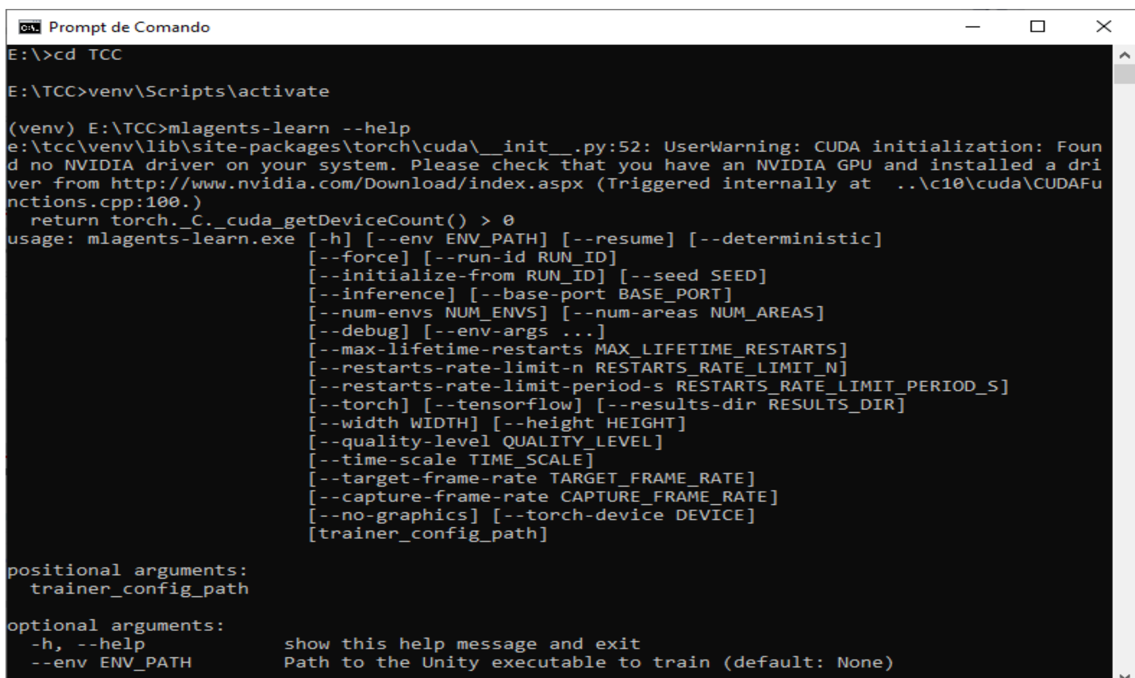
Figura 28 – Prompt de comando do Windows com comandos para criar ambiente virtual Python



Fonte: Próprio autor

biblioteca como visto na Figura 29. Nesse momento o pacote *mlagents* está pronto para realizar os processos de IA. A etapa da integração no editor da *Unity* passa necessariamente na importação da biblioteca a partir do gerenciador de pacotes. Depois disso, é possível criar classes dessa biblioteca para serem utilizadas no treinamento do agente.

Figura 29 – Comando de ajuda de MLAgent



Fonte: Próprio autor

A implementação de um agente é bastante similar ao da criação de uma classe na *Unity*. A classe que implementará as observações e decisões do agente precisa herdar da classe *Agent*. A partir dessa herança, a classe poderá realizar os processos de aprendizado por reforço como mostrado no Algoritmo 16. A primeira parte do código define variáveis para controle do cálculo percentual do jogador ser ou não humano. A linha 12 com o método *OnEpisodeBegin*, apenas atribui os objetos *Player* e *Bot* para suas respectivas variáveis de classe. Esse método é chamado todas as vezes que inicializa um episódio de treinamento. Depois de um agente receber a recompensa é finalizado o episódio para que o ciclo se reinicie. O método a seguir (linha 20) é responsável por coletar informações que serão passadas para o agente e esse, pode tomar a melhor decisão. Nesse contexto, é passado para o agente a posição do objeto observado e se ele está atacando o inimigo. A última função da classe *AgentManager* realiza o processo de avaliar as decisões do agente. É configurado que o agente pode tomar três decisões: 0, 1 ou 2. Como definido na prototipação, no momento em que o agente escolher a opção 0, ele está avaliando um jogador como *bot*; se ele escolher a opção 2, avalia como um humano e se escolher 1 não seleciona nem humano nem *bot*. A linha 33 faz o cálculo da porcentagem de acerto do agente e depois tem-se a verificação se as escolhas do agente foram acertadas e consequentemente recebe as recompensas correspondentes. A variável "nm"corresponde ao contador de métrica que está sendo iterado.

Algoritmo 16 – Classe responsável pelo agente de aprendizado por reforço

```
1  public class AgentManager : Agent {
2
3      private float actionsCountMax = 3000f;
4      private float actionsCount = 0f;
5      private float actionsAgentCount = 0f;
6      private float nm = 1f;
7      private float f = 0f;
8      private string player;
9      private Player playerObject;
10     private Bot botObject;
11
12     public override void OnEpisodeBegin() {
13         base.OnEpisodeBegin();
14         if (this.tag == "Player") {
15             playerObject = GetComponent<Player>();
16         } else {
17             botObject = GetComponent<Bot>();
18         }
19     }
20     public override void CollectObservations(VectorSensor sensor) {
21         sensor.AddObservation(transform.position);
22         sensor.AddObservation(transform.rotation);
23         if (this.tag == "Player") {
24             sensor.AddObservation(playerObject.IsAttacking());
```

```
25            } else {
26                sensor.AddObservation(botObject.IsAttacking());
27            }
28        }
29        public override void OnActionReceived(ActionBuffers actions) {
30            int choice = actions.DiscreteActions.Array[0];
31            actionsCount++;
32            if (actionsCount == actionsCountMax) {
33                float p = actionsAgentCount / actionsCountMax * 100f;
34                f += p;
35                Debug.Log("Metrica"+nm+"-Percentual do"+this.tag+"-"+p+"%");
36                Debug.Log("Porcentagem acumulada"+this.tag+"-"+(f / nm)+"%");
37                nm++;
38                actionsAgentCount = 0;
39                actionsCount = 0;
40            }
41            if (choice == 1) {
42                SetReward(0f);
43                EndEpisode();
44                return;
45            }
46            if (this.tag == "Player") { //humano
47                if (choice == 2) {
48                    actionsAgentCount++;
49                    SetReward(1f);
50                } else {
51                    SetReward(-1f);
52                }
53            } else {                        //bot
54                if (choice == 2) {
55                    SetReward(-1f);
56                } else {
57                    actionsAgentCount++;
58                    SetReward(1f);
59                }
60            }
61            EndEpisode();
62        }
63    }
```

7.1.2 Treinamento do Agente

O processo treinamento é inicializado através do *prompt* de comando. A maneira básica de execução é utilizar o comando *mlagents-learn* e após isso o treinamento começará pressionando o botão de *Play* no editor da *Unity*. Entretanto, essa forma de inicialização ocorre apenas em máquinas que tenham *Graphics Processing Unit* (GPU) da NVidia. Todos os processos deste

trabalho foram executados em uma máquina com GPU (Unidade de Processamento Gráfico) da AMD e por isso o comando para inicializar o treinamento deve ser diferenciado. Nesse caso, o processo de treinamento pôde ser realizado através do comando *mlagents-learn –torch-device cpu* que altera a execução para *Central Processing Unit* (CPU) - Unidade Central de Processamento - ao invés da GPU. A biblioteca do *ML-Agent* possibilita a adição de parâmetros de treinamento no formato YAML. O protótipo implementado utilizou o código YAML mostrado no Algoritmo 17. A maioria dos parâmetros utilizados no protótipo seguem ao proposto no trabalho (LUKAS; TOMICIC; BERNIK, 2022) e, segundo o autor, é a melhor configuração para o aprendizado do agente.

Inicialmente tem-se o chave *behaviors* que indica o comportamento do agente. Após, o nome *Agent* é o nome do arquivo que será feito referência dentro do editor da *Unity*. Nessa estrutura é adicionado os parâmetros de treinamento do agente. Na linha 3, *trainer_type* indica o tipo de algoritmo utilizado, nesse caso, é utilizado *Proximal Policy Optimization* (PPO)(Otimização de política proximal). Outras possibilidades a esse algoritmo são *Parallel Online Continuous Arcing* (POCA)(Arco online contínuo paralelo) e o *Soft Actor Critic* (SAC) (Ator crítico suave). Essa definições podem ser consultadas no repositório da biblioteca (UNITY TECHNOLOGIES, 2019).

O segundo parâmetro define um conjunto de parâmetros com a chave de *hyperparameters*. Nesse conjunto, tem-se o campo *batch_size* que descreve o número de experiências em cada iteração de descida do gradiente. É recomendado que para ações do tipo contínuo, o qual é utilizado no protótipo, esse valor seja baixo. Já o *buffer_size* corresponde a quantas experiências devem ser coletadas antes de realizar qualquer aprendizado ou atualização do modelo. O atributo *learning_rate* representa a taxa de aprendizagem inicial para descida do gradiente e corresponde à força de cada etapa de atualização do gradiente descendente. Normalmente, isso deve ser diminuído se o treinamento for instável e a recompensa não aumentar consistentemente. O *beta* é a força da regularização da entropia que torna a política “mais aleatória”. Isso garante que os agentes explorem adequadamente o espaço de ação durante o treinamento. Aumentar isso garantirá que mais ações aleatórias sejam tomadas. Isso deve ser ajustado de forma que a entropia (mensurável no TensorBoard) diminua lentamente junto com o aumento da recompensa. O campo *epsilon* influencia a rapidez com que a política pode evoluir durante a formação. Corresponde ao limite aceitável de divergência entre as políticas antigas e novas durante a atualização do gradiente descendente. Definir esse valor pequeno resultará em atualizações mais estáveis, mas também retardará o processo de treinamento. O *lambd* pode ser entendido como o quanto o agente depende da sua estimativa de valor atual ao calcular uma estimativa de valor atualizada. Valores baixos correspondem a confiar mais na estimativa do valor atual (que pode ser um viés alto), e valores altos correspondem a confiar mais nas recompensas reais recebidas no ambiente (que podem ser de alta variância). O parâmetro fornece uma compensação entre os dois, e o valor certo pode levar a um processo de treinamento mais estável. O campo *num_epoch* representa o número de passagens a serem feitas no *buffer* de experiência ao

executar a otimização de descida de gradiente. Diminuir esse campo garantirá atualizações mais estáveis, ao custo de um aprendizado mais lento. O atributo *learning _rate_schedule* determina como a taxa de aprendizagem muda ao longo do tempo. Para PPO, recomenda-se diminuir a taxa de aprendizagem até *max_steps* então a aprendizagem converge de forma mais estável.

O próximo parâmetro define as configurações da rede neural (*network _settings*). O campo *normalize* verifica se a normalização é aplicada às entradas de observação vetorial. Essa normalização é baseada na média e na variância da observação do vetor. A normalização pode ser útil em casos com problemas complexos de controle contínuo, mas pode ser prejudicial em problemas de controle discreto mais simples. O campo *hidden _units* é o número de unidades nas camadas ocultas da rede neural. Corresponde a quantas unidades existem em cada camada totalmente conectada da rede neural. Para problemas simples onde a ação correta é uma combinação direta das entradas de observação, este valor deve ser pequeno. Para problemas onde a ação é uma interação muito complexa entre as variáveis de observação, este valor deverá ser maior. O campo *num _layers* corresponde a quantas camadas ocultas estão presentes após a entrada da observação ou após a codificação de Convolutional neural network (CNN) a qual determina a Rede neural convolucional da observação visual. Para problemas simples, é provável que menos camadas treinem de forma mais rápida e eficiente. Mais camadas podem ser necessárias para problemas de controle mais complexos.

Na sessão *reward_signals* há a especificação da configuração intrínseca e extrínseca. No caso do protótipo tem-se apenas extrínseca, e essa requer a definição de dois parâmetros *gamma* e *strength*. O *gamma* é fator de desconto para recompensas futuras provenientes do meio ambiente. Isto pode ser pensado como o quão longe no futuro o agente deve se preocupar com possíveis recompensas. Em situações em que o agente deveria estar agindo no presente para se preparar para recompensas em um futuro distante, esse valor deve ser grande. Nos casos em que as recompensas são mais imediatas, podem ser menores. Deve ser estritamente menor que 1. Já o *strength* é o fator pelo qual multiplicar a recompensa dada pelo meio ambiente. Os intervalos típicos variam dependendo do sinal de recompensa.

O *max _steps* determina o número total de etapas (ou seja, observações coletadas e ações tomadas) que devem ser executadas no ambiente (ou em todos os ambientes, se vários ambientes forem usados em paralelo) antes de encerrar o processo de treinamento. Se o ambiente tiver vários agentes com o mesmo nome de comportamento em seu ambiente, todas as etapas executadas por esses agentes contribuirão para o mesmo contador de *max_steps*.

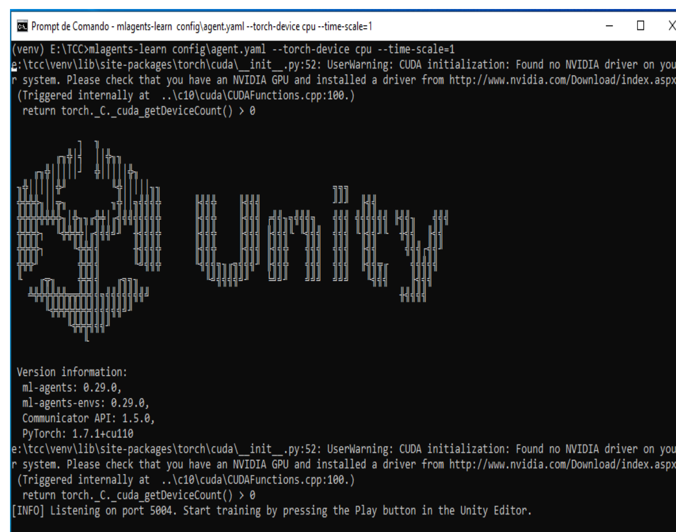
O campo *time _horizon* representa quantas etapas de experiência coletar por agente antes de adicioná-lo ao *buffer* de experiência. Quando este limite é atingido antes do final de um episódio, uma estimativa de valor é usada para prever a recompensa global esperada a partir do estado atual do agente. Como tal, este parâmetro oscila entre uma estimativa menos tendenciosa, mas com maior variância (horizonte temporal longo) e uma estimativa mais tendenciosa, mas menos variada (horizonte temporal curto). Nos casos em que há recompensas frequentes em

um episódio ou em que os episódios são proibitivamente grandes, um número menor pode ser o ideal. Este número deve ser grande o suficiente para capturar todos os comportamentos importantes dentro de uma sequência de ações de um agente.

O atributo *summary _freq* é o número de experiências que precisam ser coletadas antes de gerar e exibir estatísticas de treinamento. Isso determina a granularidade dos gráficos no Tensorboard.

Para inicializar o treino com os parâmetros é necessário passar o caminho do arquivo YAML como argumento no comando de execução do treinamento, então, o comando final para executar o treino é *mlagents-learn config\agent.yaml –torch-device cpu –time-scale=1* como mostrado na Figura 30. O último argumento do comando serve para deixar a execução do treinamento mais lenta e possibilitar uma jogabilidade normal para um jogador humano participar do treinamento.

Figura 30 – Prompt de comando no treinamento



Fonte: Próprio autor

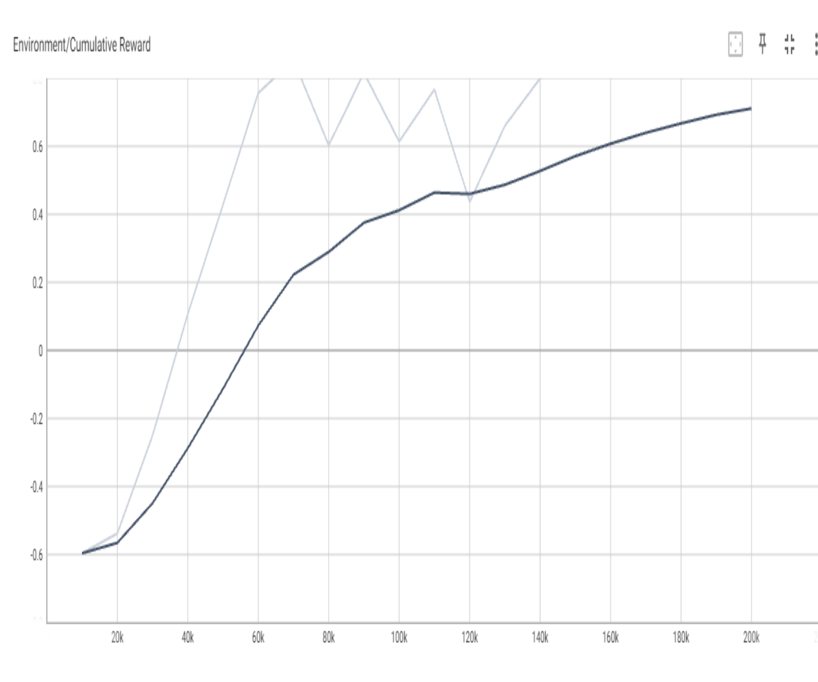
Algoritmo 17 – Parâmetros de treinamento no formato YAML

```
1  behaviors:
2    Agent:
3      trainer_type: ppo
4      hyperparameters:
5          batch_size: 32
6          buffer_size: 10240
7          learning_rate: 0.0003
8          beta: 0.005
9          epsilon: 0.2
10         lambd: 0.95
11         num_epoch: 3
12         learning_rate_schedule: linear
13     network_settings:
```

```
14          normalize: false
15          hidden_units: 128
16          num_layers: 2
17          vis_encode_type: simple
18      reward_signals:
19          extrinsic:
20              gamma: 0.99
21              strength: 1.0
22      max_steps: 500000
23      time_horizon: 64
24      summary_freq: 10000
```

A execução do treino no ambiente da *Unity* inicializa quando o botão *Play* é pressionado, nesse momento é apresentada a primeira tela do jogo e depois da seleção de *Host* ou *Client* o treinamento é começado. O treinamento foi executado no modelo 1 X 1, sendo um jogador humano e um jogador *bot* (controlado pelo computador). O tempo total de treinamento foi de 1 hora e 13 minutos compreendendo 200000 *steps* (passos). Para visualização do gráfico é necessário executar o comando *tensorboard –logdir results* e acessar o caminho do *localhost* que é indicado. Na Figura 31 é possível identificar na linha mais escura do gráfico o valor da recompensa adquirida pelos agentes durante o treinamento. Tal linha é suavizada, já que a recompensa dos agentes por vezes é negativa e por isso é necessário suavizar para um melhor entendimento do recebimento da recompensa.

Figura 31 – Gráfico da recompensa adquirida pelos agents



Fonte: Próprio autor

7.1.3 Etapa de testes

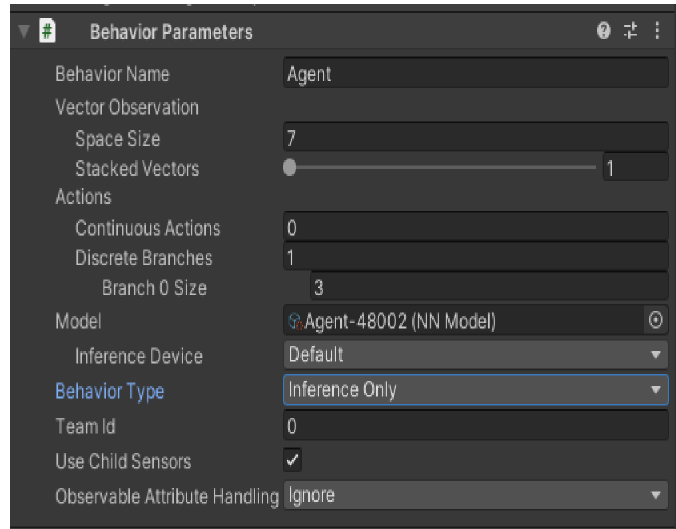
Nessa etapa do trabalho, buscou-se avaliar duas situações para viabilidade do agente na identificação de trapaça. Como discutido na fase de prototipação do projeto, a categoria de trapaça que o protótipo visa avaliar é a *Trapaça de exploração de inteligência de máquina*. Após o treinamento, é gerado um arquivo no formato .onnx que determina o cérebro treinado. Com esse arquivo é possível inseri-lo a partir do editor da *Unity* nos agentes observados e validar o resultado do treinamento como pode ser visto na Figura 32, em que é atribuído o cérebro ao campo *Model* e selecionado o tipo de comportamento (*Behavior Type*) como *Inference Only* que indica que não haverá treinamento mas a utilização do modelo no jogo. No trabalho *Anticheat System Based on Reinforcement Learning Agents in Unity* (LUKAS; TOMICIC; BERNIK, 2022), os autores obtêm nos testes de 1v1 um resultado de 84,15 % de acerto do agente na identificação de um jogador trapaceando. Uma meta ideal para o contexto deste protótipo é ter um resultado aproximado ao obtido no artigo. Para não ocorrer falsos positivos na análise dos resultados, realizou-se dois testes. O primeiro (teste 1) ocorreu em cenário em que teve um jogador humano contra um jogador *bot*. No segundo (teste 2), colocou-se um jogador humano contra outro jogador humano. Como a complexidade de realizar a distribuição do *multiplayer* acarretaria mais tempo de implementação, optou-se por utilizar o ambiente local para servir o *multiplayer*. Desse modo, pode-se realizar a *build* (construção do executável) para que um jogador jogue através do executável e outro por meio do editor da *Unity*.

Tanto o teste 1 quanto o teste 2, foram jogados dez jogos cada. Entretanto, como a distribuição do jogo *multiplayer* se tornou dificultoso, houve um impedimento para realização desse teste com dois jogadores humanos. Dessa forma optou-se realizar o teste da seguinte maneira. Nos primeiros cinco jogos o time azul teve o jogador humano controlando o personagem enquanto o time vermelho ficava sem movimentação, e por consequência, mantinha-se estático no local de nascimento de seu time. Na outra bateria de cinco jogos, inverteu-se os times, colocando o jogador humano no controle do time vermelho enquanto o time azul ficava sem movimentação. Adotou-se essa estratégia para que fosse possível a observação da jogabilidade entre dois humanos.

7.1.4 Análise dos Resultados

Os resultados do teste 1 pode ser visto na Tabela 1. Pode-se observar que o agente decidiu positivamente na identificação de jogador do tipo *bot* em torno de 80 % em todos os jogos computados alcançando uma média de 83.27 % de chance daquele jogador analisado ser um *bot*. Entretanto, as ações do agente na identificação do jogador humano é de apenas 17.94 %. Isso mostra que o agente seguindo os parâmetros de treinamento e observação não conseguiu identificar um humano. Comparativamente, o resultado obtido para ambos personagens do jogo (humano e bot-antitrapaça) foram bastante similares aos que foram analisados pelos autores do artigo *Anticheat System Based on Reinforcement Learning Agents in Unity* (LUKAS; TOMICIC;

Figura 32 – Componente com o cérebro treinado



Fonte: Próprio autor

BERNIK, 2022). Entretanto, os autores julgam o cálculo obtido como percentual de trapaça e avaliam que no Teste 1, o jogador Bot teve um percentual de 84.15% de chance de estar trapaceando enquanto o jogador humano obteve um taxa de 12.18%. Esses dados podem ser vistos na Tabela 2.

Tabela 1 – Resultado do Teste 1, um jogador humano versus um jogador bot

	Humano	**Bot**
Jogo 1	17.56 %	82.7 %
Jogo 2	17.86 %	82.45 %
Jogo 3	18.01 %	83.78 %
Jogo 4	16.89 %	82.74 %
Jogo 5	17.9 %	82.74 %
Jogo 6	18.01 %	83.94%
Jogo 7	18.24 %	83.07%
Jogo 8	17.8 %	83.7 %
Jogo 9	18.02 %	83.68 %
Jogo 10	19.08 %	83.89 %
Média	17.94 %	83.27 %

Fonte: Próprio Autor

Tabela 2 – Resultado obtido no artigo

	Humano	**Bot**
Teste 1	12.18 %	84.15 %

Fonte: Anticheat System Based on Reinforcement Learning Agents in Unity; Mihael, Lukas; 2022

Já no teste 2 com o modelo de realizar cinco jogos com cada jogador sendo devidamente controlado por humano, pode-se observar na Tabela 3 que ambos os modelos ficam bastante parecidos. Nos primeiros cinco jogos, o agente teve um acerto entre 18.53 % e 19.9 % para o time azul (sendo controlado) enquanto que para o time vermelho (não controlado), o agente obteve uma taxa de acerto na faixa de 17.38% a 18.01%. Esse intervalo de porcentagem pode-se ser observada nos últimos cinco jogos mas com os times invertidos. Desse modo, o agente teve uma taxa de acerto bastante similar nas duas situações, sendo que o jogador não controlado tem uma taxa levemente menor. O jogador controlado obteve uma porcentagem parecida com o teste 1.

Tabela 3 – Resultado do Teste 2, dois jogadores humanos

	Azul	**Vermelho**
Jogo 1	19.9 %	17.38 %
Jogo 2	18.9 %	17.6 %
Jogo 3	19.21 %	17.4 %
Jogo 4	19.17 %	17.8 %
Jogo 5	18.53 %	18.01 %
Jogo 6	17.83 %	19.07%
Jogo 7	17.35 %	19.81%
Jogo 8	17.8%	19.07 %
Jogo 9	16.7 %	19.9 %
Jogo 10	17.05 %	18.53 %

Fonte: Próprio Autor

Nas duas situações vistas, pode-se perceber que o jogador humano não é tão identificado quanto o jogador do tipo *bot*. Uma possível explicação para esse comportamento pode ser por causa dos poucos atributos observados pelo agente para tomada de decisão, uma vez que somente são observados a posição do jogador no jogo e se ele está atacando o inimigo. Apesar do resultado relativamente baixo na identificação do jogador humano, tem-se um resultado promissor na identificação de jogadores utilizando trapaça através do agente. O protótipo construído tentou identificar apenas um cenário de trapaça, entretanto há potencial de ampliar e validar outros caminhos para atingir esse objetivo através da manipulação das informações de entrada no aprendizado ou na alteração dos parâmetros de configuração YAML de treinamento.

7.1.5 Discussão

Embora os resultados da identificação de jogadores humanos terem sido abaixo do esperado, o agente conseguiu identificar os jogadores do tipo *bot* nos testes realizados. Essa tendência na identificação de jogadores humanos pode ser sido provocada pela falta de outros parâmetros observáveis no treinamento do agente. Assim, com estudos mais detalhados e com ênfase maior na parte da aprendizagem de máquina, poderá ser observado resultados mais significativos. Além dessas novas observações, o período de treinamento poderia ser estendido,

dessa forma, levando o agente a um nível mais aperfeiçoado na tomada de decisões. Alguns parâmetros que poderiam fazer parte da abordagem de validação seriam:

a) jogador está com a caixa;
b) jogador está no local de destino;
c) jogador está olhando para qual lado;
d) aumentar a vida do inimigo e validar tempo de ataque do jogador;
e) jogador colidiu com a parede;
f) tempo de resposta na movimentação no início da cena.

Trabalhos futuros poderão ampliar e expandir diferentes visões sobre a implementação de um agente capaz de identificar possíveis trapaceiros nos mais variados estilos de jogos. Nesses casos, o agente pode ter mais funcionalidades, como validar um jogador utilizando *bot* e também outro tipo de auxílio dentro do jogo.

7.1.6 Conclusão sobre ML Agente no protótipo

A integração do agente ao protótipo tem papel fundamental para atingir o objetivo do trabalho. Observou-se o processo de instalação das dependências da biblioteca bem como a instalação da biblioteca através do gerenciador de pacotes da *Unity*. Criou-se as condições de recompensas do agente a partir da classe *AgentManager* e com isso iniciou-se a fase de treinamento. Após essa etapa, sucedeu-se a etapa de teste em que colheu-se os resultados dos testes. Observa-se que solução atingiu em partes os objetivos propostos já que por um lado conseguiu identificar positivamente os jogadores trapaceiros, entretanto por outro lado, teve-se um resultado abaixo do esperado na identificação de jogadores humanos apesar desses resultados serem similares aos registrados no artigo *Anticheat System Based on Reinforcement Learning Agents in Unity* (LUKAS; TOMICIC; BERNIK, 2022).

8 CONCLUSÃO

Em todos os campos da área da computação, a segurança é um dos fatores que mais demandam atenção e atualizações. Jogos digitais multijogador online também possuem nível crítico nas questões que envolvem a integridade dos jogos. A taxonomia apresentada no trabalho detalha as categorias em que as principais trapaças podem ser classificadas.

A bibliometria permitiu verificar trabalhos publicados no meio científico. Conclui-se através de métodos bibliométricos que não há quantidade significativa de trabalhos que utilizam métodos antitrapaça no *Unity Engine*. Um dos trabalhos revisados trouxe a viabilidade de implementação pela *Unity* utilizando aprendizado por reforço. Dessa forma, tornou-se importante a análise e validação extraídos desse trabalho. Para tal, escolheu-se o caminho da implementação de um protótipo de jogo digital multijogador em que, através do *ML-Agent* fosse possível a identificação um jogador trapaceiro.

A implementação do protótipo junto da integração da biblioteca *ML-Agent* ao jogos são partes fundamentais para atingir o objetivo na criação de método antitrapaça em jogos desenvolvidos na *Unity*. No segmento de codificação do protótipo, observou-se componentes externos da *Unity*, como *Netcode* e *NavMesh*, integrados ao projeto, o que tornou o aprendizado mais rico e um tanto desafiador. A maior barreira vista nesse quesito, deu-se na implementação do *multiplayer* em que paradigmas mais sofisticados devem ser seguidos para que o código funcione corretamente.

Após a seção de código, integrou-se a biblioteca *ML-Agent* ao projeto. Nessa parte houve a explicação de todos os passos para a execução correta dessa biblioteca dentro do jogo desenvolvido na *Unity*. Também aqui, apresenta-se as etapas de treinamento do agente e de testes no jogo. Feito os testes, há a análise dos resultados, parte esta, que observou-se que o agente teve uma taxa de acerto aceitável para identificação do *bot*, entretanto não houve uma porcentagem muito alta na identificação de um humano. A discussão realizada levanta pontos que, um dos motivos para que haja esse comportamento é a implementação do protótipo com poucos atributos observáveis no momento do treinamento do agente. Essa pensamento leva a crer que novos treinamentos com outros atributos possam ter resultados mais satisfatórios.

Para trabalhos futuros, como já discutido, observa-se falta de bibliografia para o tópico apresentado. Portanto, seria importante a conscientização não só do meio acadêmico mas todos os envolvidos na indústria através de palestras, simpósios, conferências, uma vez que, essa indústria é uma das mais importantes no quesito entretenimento e sua integridade e confiabilidade são de suma importância. Além disso, pode-se destacar a evolução da IA e as possíveis soluções que os modelos dessa área podem auxiliar na detecção das falhas que um jogo digital, seja online ou não, possa apresentar. Já na parte de implementação, acredita-se que trabalhos que

dediquem maior atenção ao treinamento do agente possam conseguir levantar mais hipóteses acerca da viabilidade da solução proposta no trabalho.

8.1 SUGESTÕES DE TRABALHOS FUTUROS

Como trabalhos futuros sugere-se:

a) fomentar pesquisa sobre o tema de trapaças em jogos digitais;
b) realizar processo de treinamento com novos parâmetros;
c) implementar protótipo para outros gêneros de jogos;

REFERÊNCIAS

AARSETH, E. Computer game studies, year one. the international journal of computer game research, v. 1, 2001.

ACEDO, F. J. *et al.* Co-authorship in management and organizational studies: An empirical and network analysis. Journal of Management Studies, v. 43, p. 957–983, 2006.

ALAYED, H.; FRANGOUDES, F.; NEUMAN, C. Behavioral-based cheating detection in online first person shooters using machine learning techniques. IEEE, p. 1–8, 2013. Disponível em: <https://ieeexplore.ieee.org/abstract/document/6633617/keywords#keywords>. Acesso em: 22 mai 2023.

ALKHALIFA, S. Machine learning and anti-cheating in fps games. 2016. Disponível em: <https://www.researchgate.net/publication/308785899_Machine_Learning_and_Anti-Cheating_in_FPS_Games>. Acesso em: 18 mai 2023.

ARAúJO, C. A. A. Bibliometria: evolução histórica e questões atuais. Em Questão, 2006. Disponível em: <https://seer.ufrgs.br/index.php/EmQuestao/article/view/16>. Acesso em: 25 mai 2023.

ARIK, K.; GEZER, M.; TAYALI, S. T. Bibliometric analysis of scientific studies published on game customer churn analysis between 2008 and 2022. v. 5, p. 55–75, 06 2022.

CAIN, A. **taxonomy**. Encyclopedia Britannica, 2023. Disponível em: <https://www.britannica.com/science/taxonomy>. Acesso em: 22 June 2023.

CHAPEL, L.; BOTVICH, D.; MALONE, D. Probabilistic approaches to cheating detection in online games. IEEE, p. 195–201, 2010. Disponível em: <https://www.researchgate.net/publication/221157498_Probabilistic_Approaches_to_Cheating_Detection_in_Online_Games>. Acesso em: 17 mai 2023.

CHEN, Y.-C. *et al.* Online gaming crime and security issue - cases and countermeasures from taiwan. In: . [S.l.: s.n.], 2020. p. 131–136.

CRAIGHEAD, J.; BURKE, J.; MURPHY, R. Using the unity game engine to develop sarge: A case study. **Computer**, v. 4552, 01 2007.

DAUNTON, N. **One map away from winning: "How one player rocked Indian esports**. 2023. Disponível em: <https://www.euronews.com/next/2022/07/09/one-map-away-from-winning-how-one-player-rocked-indian-esports>. Acesso em: 19 jun 2023.

DEWINTER, J.; MOELLER, R. M. **Computer Games and Technical Communication : Critical Methods and Applications at the Intersection**. [S.l.]: Routledge, 2016. 109 -110 p.

DIAZ, E. **Cheat gamers, a threat to the video game industry**. 2023. Disponível em: <https://impactotic.co/en/cheating-gamers-a-threat-to-the-video-game-industry/>. Acesso em: 19 jun 2023.

DILLON, R. Rise of the game engines. In: ____. [S.l.: s.n.], 2015. p. 87–104.

DORMANS, J. **Engineering Emergence Applied Theory for Game Design**. Routledge, 2012. 44 p. Disponível em: <https://eprints.illc.uva.nl/id/eprint/2118/1/DS-2012-12.text.pdf>. Acesso em: 14 jun 2023.

FILHO, M. **Unity — NavMesh Tutorial**. 2019. Disponível em: <https://medium.com/@moesio.f/https-medium-com-moesio-f-unity-navmesh-tutorial-390a7b794817>. Acesso em: 17 ago 2023.

FOWLER, M. **UML essencial: um breve guia para a linguagem-padrão de modelagem de objetos**. Porto Alegre: Bookman, 2005.

GALLI, L.; LOIACONO, D.; CARDAMONE LUIGI; LANZI, P. L. A cheating detection framework for unreal tournament iii: A machine learning approach. IEEE, p. 266–272, 2011. Disponível em: <https://ieeexplore.ieee.org/abstract/document/6032016>. Acesso em: 18 mai 2023.

GAMMA, E. *et al.* **Design patterns: Elements of Reusable Object-Oriented Softwares**. 2000. Disponível em: <https://integrada.minhabiblioteca.com.br/#/books/9788577800469/>. Acesso em: 17 ago 2023.

JIANRONG, T. *et al.* Xai-driven explainable multi-view game cheating detection. p. 144–151, 2004.

JIN, D. Y. Historiography of korean esports: Perspectives on spectatorship. International Journal of Communication, v. 14, p. 3727–3745, 2020. Disponível em: <https://ijoc.org/index.php/ijoc/article/view/13795/3149>. Acesso em: 19 jun 2023.

JONNALAGADDA, A. *et al.* **Robust Vision-Based Cheat Detection in Competitive Gaming**. New York, NY, USA: Association for Computing Machinery, 2021. 1-18 p.

JULIANI, A. *et al.* Unity: A general platform for intelligent agents. **arXiv preprint arXiv:1809.02627**, 2020.

KARKALLIS, P. *et al.* Detecting video-game injectors exchanged in game cheating communities. European Symposium on Research in Computer Security, v. 12972, p. 305–324, mar 2021.

KARLSSON, C. Researching operations management. London: Routledge, 2008.

L, E. *et al.* Proknow-c, knowledge development process – constructivist. processo técnico com patente de registro pendente junto ao inpi rio de janeiro. 2010.

LACERDA, R. T. d. O.; ENSSLIN, L.; ENSSLIN, S. R. Uma análise bibliométrica da literatura sobre estratégia e avaliação de desempenho. Gestão E Produção, 2012. Disponível em: <http://dx.doi.org/10.1108/03090591011070761>. Acesso em: 25 mai 2023.

LAPRIE, J. **Dependability: Basic Concepts and Terminology**. [S.l.]: Springer, Viena, 1992. v. 5.

LEE, W.-K.; CHANG, R. K. C. Evaluation of lag-related configurations in first-person shooter games. In: **2015 International Workshop on Network and Systems Support for Games (NetGames)**. [S.l.: s.n.], 2015. p. 1–3.

LUKAS, M.; TOMICIC, I.; BERNIK, A. Anticheat system based on reinforcement learning agents in unity. Information, v. 13, p. 173, mar 2022. Disponível em: <https://doi.org/10.3390/info13040173>. Acesso em: 15 mai 2023.

LóPEZ-BELMONTE, J. *et al.* Scientific mapping of gamification in web of science. European Journal of Investigation in Health, Psychology and Education, v. 10, p. 832–847, 1920.

OOSTERHU, K.; FEIREISS, L. **The Architecture Co-laboratory: Game Set and Match II : on Computer Games, Advanced Geometries, and Digital Technologies**. [S.l.: s.n.], 2006. ISBN 9789059730366.

PAO, H.-K.; CHEN, K.-T.; CHANG, H.-C. Game bot detection via avatar trajectory analysis. IEEE, v. 2, n. 3, p. 162–175, 2010. Disponível em: <https://ieeexplore.ieee.org/document/5560779/keywords#keywords>. Acesso em: 18 mai 2023.

PARK, C. **Online cheating is ubiquitous**. 2001. Disponível em: <http://legacy.www.hani.co.kr/section-005100025/2001/05/005100025200105091907004.html>. Acesso em: 07 mai 2023.

PINTO, A.; GONZALES-AGUILAR, L. A. **Visibilidad de los estudios en análisis de redes sociales en América del Sur: su evolución y métricas de 1990-2013**. Campinas: TransInformação, 2014. 253-267 p. Disponível em: <http://www.scielo.br/pdf/tinf/v26n3/0103-3786-tinf-26-03-00253.pdf>. Acesso em: 25 mai 2023.

PODSAKOFF, P. M. *et al.* The influence of management journals in the 1980s and 1990s. Strategic Management Journal, 2005. Disponível em: <http://dx.doi.org/10.1002/smj.454>. Acesso em: 25 mai 2023.

PRITCHARD, M. **How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It**. 2000. Disponível em: <https://www.gamedeveloper.com/design/how-to-hurt-the-hackers-the-scoop-on-internet-cheating-and-how-you-can-combat-it>. Acesso em: 10 mai 2023.

ROCHA, M. **Mercado de games cresce no país e atrai cada vez mais empreendedores**. 2022. Disponível em: <https://www1.folha.uol.com.br/mpme/2022/08/mercado-de-games-cresce-no-pais-e-atrai-cada-vez-mais-empreendedores.shtml#:~:text=\%24196\%2C8\%20bi\%20\%C3\%A9\%20a,movimentar\%C3\%A1\%20no\%20mundo\%20em\%202022>. Acesso em: 23 abr 2023.

ROUSE, M. **What Does Lag Mean?** 2011. Disponível em: <https://www.techopedia.com/definition/17182/lag-gaming>. Acesso em: 05 mai 2023.

SALAZAR, M. G. *et al.* Proposal of game design document from software engineering requirements perspective. **International Conference on Computer Games**, 2012.

SALEN, K.; ZIMMERMAN, E. **Rules of Play: Game Design Fundamentals**. [S.l.: s.n.], 2004.

SANTOS, R. N. M. **Indicadores estratégicos em ciência e tecnologia: refletindo a sua prática como dispositivo de inclusão/exclusão**. Campinas: TransInformação, 2003. 129-140 p.

SKAAR, O. The potential of trusted computing for strengthening security in massively multiplayer online games. 2010.

TASCA, J. E. *et al.* An approach for selecting a theoretical framework for the evaluation of training programs. Journal of European Industrial Training, 2010. Disponível em: <http://dx.doi.org/10.1108/03090591011070761>. Acesso em: 25 mai 2023.

TRANFIELD, D.; DENYER DAVID SMART, P. Towards a methodology for developing evidence-informed management knowledge by means of systematic review. British Journal of Management,, 2003. Disponível em: <http://dx.doi.org/10.1111/1467-8551.00375>. Acesso em: 25 mai 2023.

UNITY TECHNOLOGIES. **Training-ML-Agent.md**. [s.n.], 2019. Disponível em: <https://github.com/yosider/ml-agents-1/blob/master/docs/Training-ML-Agents.md>. Acesso em: 21 nov 2023.

YAN, J. J.; CHOI, H. Security issues in online games. The Electronic Library, v. 20, p. 125–133, 2002.

YAN, J. J.; RANDELL, B. A systematic classification of cheating in online games. p. 1–9, 10 2005.

ZUPIC, I.; ČATER, T. Bibliometric methods in management and organization. Organization Research Methods, v. 18, p. 429–472, 07 2015.