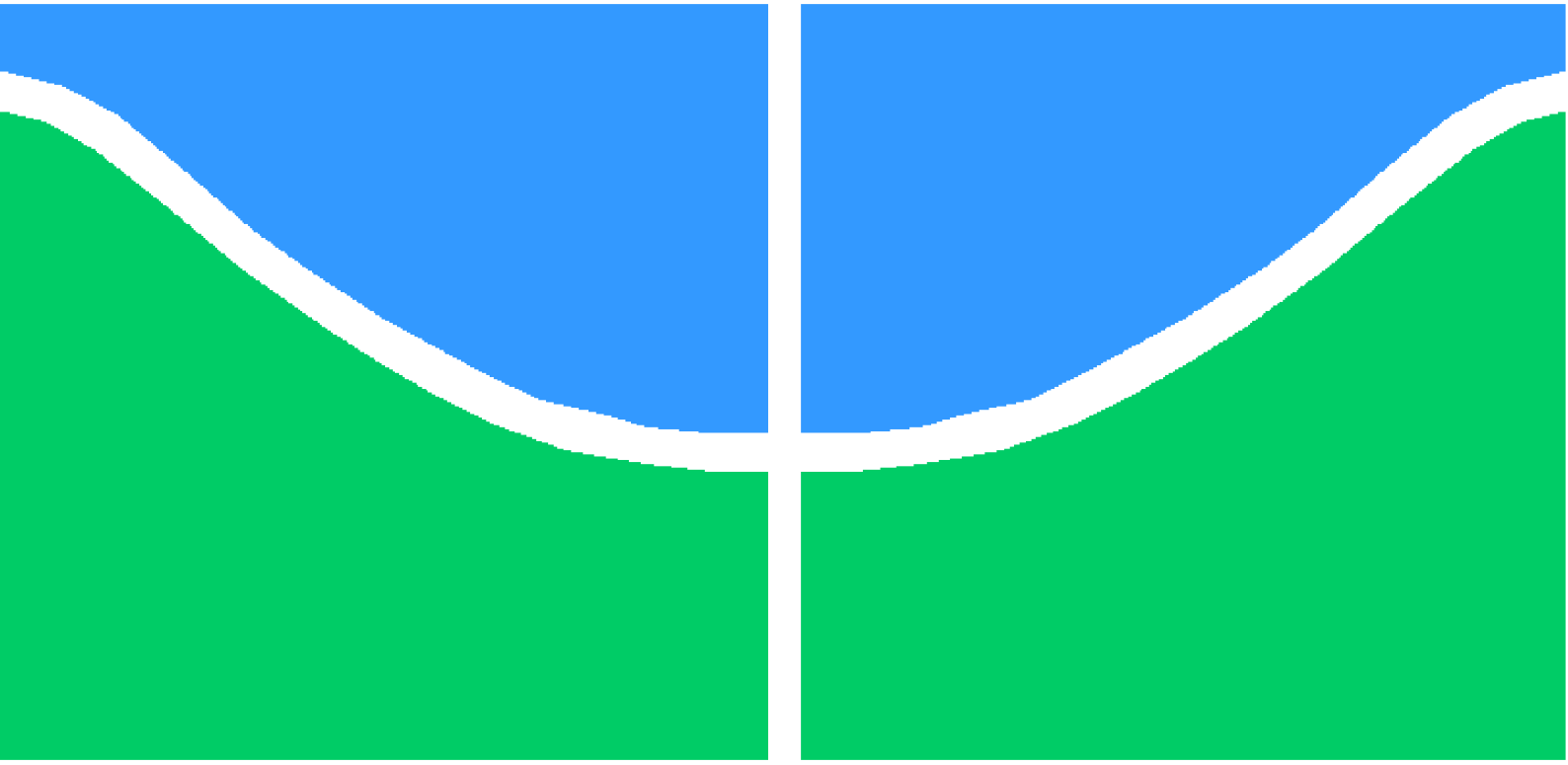

Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Abordagens de agentes autônomos para jogar o jogo *Go Dash*

Autor: Marlon Mendes Ciriático Guimarães
Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF
2019

Marlon Mendes Ciriático Guimarães

Abordagens de agentes autônomos para jogar o jogo *Go Dash*

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Supervisor: Prof. Dr. Edson Alves da Costa Júnior

Co-supervisor:

Brasília, DF

2019

Marlon Mendes Ciriático Guimarães
Abordagens de agentes autônomos para jogar o jogo *Go Dash*/ Marlon Mendes Ciriático Guimarães. – Brasília, DF, 2019-
62 p. : il. (algumas color.) ; 30 cm.

Supervisor: Prof. Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2019.

1. Desenvolvimento de jogos. 2. Agentes autônamos. I. Prof. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Abordagens de agentes autônomos para jogar o jogo *Go Dash*

CDU

Marlon Mendes Ciriático Guimarães

Abordagens de agentes autônomos para jogar o jogo *Go Dash*

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 01 de junho de 2019 – Data da aprovação do trabalho:

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. Dr. Fábio Macedo Mendes
Convidado 1

Prof. Dr. Fernando William Cruz
Convidado 2

Brasília, DF
2019

Resumo

Jogadores artificiais (*bots*) constituem uma peça fundamental de muitos jogos. *Bots* que exibem comportamento similar a de jogadores humanos proporcionam uma experiencia mais satisfatória para jogadores humanos. Criar agentes autônomos que se comportam de forma indistinguível de humanos é um problema aberto na computação e está ligado com o Teste de Turing (1950). Neste trabalho é discutido experimentos conduzidos para avaliar a capacidade de agentes de comportarem como jogadores humanos em jogos do gênero *2D Shooter Multiplayer*. Foi implementado o modelo de Máquina de Estados Finita para os agentes testados e é proposto outras duas abordagens para construção de *bots*: *Behavior Tree* e *Mirroring*, que não foram implementadas mas seus modelos são descritos. O jogo *Go Dash* foi desenvolvido neste trabalho para se testar estas abordagens de agentes autônomos.

Palavras-chave: game. artificial intelligence . 2d . shooter . multiplayer. mirroring . behavior trees.

Abstract

Artificial players (bots) are a key piece of many games. Bots that behave in similar way to that of human players provide a more satisfying experience for human players. Creating autonomous agents that behave indistinguishably from humans is an open problem in computing and is linked with the Turing Test (1950). This paper discusses experiments conducted to evaluate the ability of agents to behave as human players in games of the genre *2D Shooter Multiplayer*. The Finite State Machine model was used for the tested agents and two other approaches for building bots are proposed: Behavior Tree and Mirroring. The game *Go Dash* was developed in this work to test these approaches.

Keywords: game. artificial intelligence . 2d . platform . mirroring . behavior trees.

Lista de ilustrações

List of abbreviations and acronyms

Bot	Abreviatura de *robot*, um sistema autônamo (do inglês, *robot*)
NPC	Personagem Não Jogador (do inglês, *Non-player character*)
IA	Inteligência Artificial
FPS	Jogo de tiro em primeira pessoa, do inglês *First-person shooter*
API	Interface de programação de aplicações, do inglês *Application programming interface*
IDE	Ambiente de desenvolvimento integrado, do inglês *Integrated development environment*
RPC	Chamada Remota de Procedimento, do inglês *Remote Procedure Call*

Sumário

Introdução

Os Personagens Não Jogadores (NPCs) são uma parte fundamental de muitos jogos. No jogo *Pac-Man*, por exemplo, lançado em 1980, o jogador percorria um labirinto a fim de comer uma série de pontos espalhados pelo mapa, enquanto fugia de quatro fantasmas (que são NPCs). Em partidas online de *Counter-Strike: Global Offensive* (2012), jogo do gênero *Shooter*, os *bots* tem a função de substituir de jogadores que desconectaram da partida por algum motivo, assumindo uma posição de trabalho em equipe com os outros jogadores do seu time. Então nos jogos os NPCs podem atuar em diversas funções, como inimigos diretos, colegas de equipe, animais ou monstros espalhados pelo mapa, por exemplo.

Construir jogadores artificiais para enfrentar jogadores humanos é uma área em si só. Em 1997, o computador *Deep Blue* derrotou o então campião mundial, Gary Kasparov, em uma partida de xadrez. Kasparov é considerado um dos melhores jogadores da história. Em feito parecido, o software *AlphaGo*, produzido pela Google DeepMind, derrotou o 18 vezes campeão mundial, Lee Sedol em partidas do jogo Go, em 2016. Em jogos modernos, como *Dota 2*, lançado em 2013, esta história também se repetiu. Os melhores jogadores profissionais de *Dota 2* foram selecionados para enfrentar *bots* produzidos pela organização *OpenAI*. Em 2017, o software da organização conseguiu derrotar jogadores profissionais no modo 1 contra 1 (1v1). E pela primeira vez na história dos esportes eletrônicos, um time de *bots* derrotou uma organização profissional, em abril de 2019, ganhando partidas contra o melhor time do ano no jogo *Dota 2* (OpenAI, 2019).

Apesar de agentes autônomos terem tidos êxito em derrotar os melhores jogadores humanos em determinados jogos, o objetivo final de um *bot* nem sempre é ser o mais forte possível. Para (RABIN, 2013), o objetivo final dos *bots* em jogos é entreter o jogador. Neste sentido, o desafio não é um como construir um agente hábil e sim como construir um jogador artificial que proporciona entretenimento para o jogador humano. Uma das características que impacta na satisfação que um *bot* proporciona, é o quanto ele parece ser humano. Isto é, um agente passar a impressão de que é controlado por um humano.

Quando estes agentes autônomos são previsíveis e repetitivos, jogadores humanos aprendem rapidamente estes padrões e os utilizam ao seu favor. Por outro lado, NPCs extremamente habilidosos também não são divertidos para os jogadores humanos, pois não um há desafio equilibrado. Além de NPCs previsíveis ou extremamente habilidosos não gerarem um sentimento de satisfação nos jogadores, eles constituem, um fator de frustração para os jogadores.

Quanto mais um NPC parece ser controlado por um humano, mais satisfeito fica

o jogador (ORTEGA N. SHAKER, 2013). Portanto, o desafio é projetar e implementar um agente autônomo que se comporte como um jogador humano. Turing (1950) propôs o problema geral de construir uma máquina que possuísse uma inteligência genérica e que fosse indistinguível de um ser humano quando interagida por meio de conversas textuais, um problema ainda em aberto na computação. A motivação deste trabalho é construção de um agente autônomo que dê a impressão de ser controlado por um humano, ou seja, construir um NPC que seja o indistinguível, o quanto for possível, de um jogador humano. Turing levantou o problema geral, de construir uma máquina que seja indistinguível de um humano em uma conversa sobre qualquer assunto. Aqui estamos interessados em um problema mais específico, de construir um robô que seja indistinguível de um humano dentro de um jogo.

Não encontramos referências sobre modelos e implementação de agentes autônomos para jogos do gênero *2D Shooter Multiplayer*. Entretanto há referências de modelos para outros jogos como *Super Mario Bros* (AL., 2014) e *Unreal Tournament 2004* (POLCEANU, 2013). Portanto decidimos testar modelos clássicos (como Máquina de Estados Finita) no jogo *Go Dash*, do gênero *2D Shooter Multiplayer* e que foi desenvolvido neste trabalho com o propósito de ser um ambiente de experimentos dos modelos.

Uma versão adaptada do Teste de Turing (TURING, 1950), será utilizada para mensurar o quanto os agentes se comportam (jogam) como jogadores humanos. Uma analise qualitativa também será feita sobre os resultados dos experimentos.

Objetivos

O objetivo é construir um jogador artificial para o jogo *Go Dash* que se assemelha a um jogador humano.

Os objetivos específicos são:

- desenvolver o jogo *Go Dash*;
- projetar os módulos de Visão, Tiro e *Pathfinding* do agente;
- implementar os modelos de agentes autônomos: Máquina de Estados Finita, *Behavior trees* e *Mirroring*;
- comparar a humanidade de cada agente desenvolvido com base numa adaptação do Teste de Turing;
- realizar análise qualitativa sobre os resultados dos experimentos.

Estrutura do trabalho

Este trabalho está organizado em quatro capítulos Fundamentação Teórica, onde trabalhos correlatos são analisados e conceitos da área são definidos; Metodologia, onde é apresentado os projetos e implementações do jogo e dos sistemas inteligentes;Resultados e Discussões onde são discutidos resultados obtidos e Considerações Finais, onde são discutidos limitações conhecidas e trabalhos futuros.

1 Fundamentação Teórica

1.1 Jogos

A indústria de jogos é a mais lucrativa do setor de entretenimento, com receita de 116 bilhões de dólares em 2018, maior que o setor de televisão (U$ 105B), e duas vezes maior que a indústria de filmes (U$ 41B) e músicas somadas (Gamecrate, 2018).

1.2 Inteligência artificial em jogos

O objetivo principal de Inteligência Artificial (IA) em jogos é auxiliar o jogo em proporcionar uma experiência fascinante para o jogador (RABIN, 2013). Dentro do jogo, a IA pode se instanciar em Personagens Não Jogáveis (NPCs), agentes autônamos do jogo que agem de forma independente do jogador humano (CONNOLLY T. HAINEY, 2013). A credibidilidade do NPC em se comportar de formar similar à um humano é de grande interesse dos projetistas de jogos (PETRI, 2007), afetando diretamente a experiencia do jogador (TENCé C. BUCHE, 2010). Quanto mais o NPC se comporta como um humano, mais satisfeito fica o jogador (ORTEGA N. SHAKER, 2013). A credibilidade do personagem (do inglês, *player believability*) é a crença de que o mesmo é controlado por um ser humano e é um fator independente da aparência física do NPC (LIVINGSTONE, 2006).

1.3 Comportamento humano e o teste de Turing

Definir o que é comportamento de humanos e o que é comportamento de robôs não é trivial, e com este problema em aberto Turing (1950) propôs uma pergunta, em vez de uma definição formal: “Podem as máquinas pensar?”. Para responder a pergunta, Turing propôs um teste que ficou conhecido como o “Teste de Turing” e funciona da seguinte maneira: três jogadores participam de um jogo chamado O Jogo da Imitação (*The imitation game*): um robô que está sendo testado, um ser humano, e um segundo humano que é o juíz do teste. Inicialmente o juíz não sabe qual dos dois jogadores é humano, e o juiz ganha o jogo se descobrir quem o é. O juíz pode fazer perguntas através de textos (uma espécie de *chat*) para qualquer um dos dois jogadores, até ter seu veredito final. Caso o juíz não consiga distinguir de forma consistente quem é o humano, ele perde o jogo e o jogador robô é vitorioso. Caso o robô seja sucedido em enganar o juíz é dito que o robô passou no teste, concluindo que as máquinas podem, de fato, pensar.

Uma variante do teste de Turing foi utilizado na competição 2K BotPrize (HINGSTON, 2009) para classificar quais jogadores eram controlados por humanos e quais joga-

dores eram controlados por um programa (*bots*). O teste foi feito numa versão adaptada do jogo Unreal Tournament 2004, um jogo de tiro em primeira pessoa (FPS). Cada sessão do teste foi constituida por três jogadores: um *bot*, um humano e o juíz, os três jogadores interagem entre si e o juíz formulou o seu veredito e classificou cada um dos outros dois jogadores como *bot* ou como humano. Na competição de 2012, por exemplo, além dos juízes participaram 4 jogadores humanos e 6 *bots*, e cada um dos jogadores foram julgados aproximadamente 25 vezes. Uma vez terminado o teste, a "humanidade" dos jogadores (humanos e *bots*), foi calculada em porcentagem dividindo o número de vezes que o jogador foi julgado como humano pelo número total de votos do jogador (POLCEANU, 2013).

Ainda que não se tenha uma definição formal completa sobre o que é comportamento humano, adaptações do teste de Turing são utilizadas para mensurar a humanidade de agentes autônamos e e determinar se a implementação de um agente em específico pode de fato se passar por um humano em um jogo, assim como utilizado por Polceanu (2013) e Schrum I. Karpov (2011). É importante observar que o teste proposto por Turing (1950) investigava uma inteligência genérica, e não uma habilidade específica (como jogar um jogo) de escopo limitado. Ou seja, caso um robô seja bem sucedido em um teste de Turing adaptado não implica que ele consiga "pensar".

1.4 Algoritmos de seleção de comportamento

Antes de um NPC executar uma ação, como cumprimentar um jogador, por exemplo, ele primeiro deve decidir que irá executar esta ação. Algoritmos de seleção de comportamento representam o processo de tomada de decisão de um NPC e não necessariamente sobre como uma tarefa ou decisão deve ser concretizado. Um algoritmo que produz resultados satisfatórios em um jogo não necessariamente irá apresentar resultados similares em outro jogo (RABIN, 2013). Esta seção apresenta conceitos de abordagens que serão utilizadas no desenvolvimento do trabalho.

1.4.1 Máquina de Estados Finita

Máquina de Estados Finita é um dos modelos mais populares utilizado em IA para jogos na indústria (RABIN, 2013) para seleção de comportamentos. Uma Máquina de Estados Finita é um modelo que possui um número finito de *estados*, onde símbolos de entrada faz com que a máquina mude para um outro estado, através de uma *transição*. Em qualquer momento, a Máquina está em exatamente um estado (BUCKLAND, 2005).

1.4.2 *Behavior trees*

Cada nó em uma árvore *behavior tree* representa um comportamento. Um comportamento de um nó é executado se e somente se sua precondição determinada é verdadeira. Por exemplo, pode-se ter um nó em que o comportamento é "Recuperar pontos de vida" com a precondição de "Pontos de vida menor que 40%". Portanto, se uma precondição de um nó é avaliada em falso, nenhum dos comportamentos nesta subárvore serão executados. Caso a precondição seja avaliada em verdadeiro, esta avaliação booleana segue recursivamente na subárvore do nó avaliado, da esquerda para a direita.

A *behavior tree* permite o uso de diferentes tipos de nó. Por exemplo, um nó do tipo *sequência* especifica um comportamento em que todos os filhos deste nó serão executados em ordem sequencial, da esquerda para a direita. Um nó do tipo "Repita N vezes" determina um comportamento que será repetido por N vezes, ou até que uma condição de erro (uma precondição é avaliada em falso em algum dos filhos) aconteça.

Existe exatamente um nó raiz, que é a raiz da árvore implícita. Este nó é o ponto de partida para a execução do algoritmo. O objetivo do Nó seletor é escolher exatamente uma opção a partir de um conjunto de opções. Cada filho de um nó seletor representa uma opção. O seletor irá escolher o primeiro nó (da esquerda para a direita) que a precondição seja avaliada verdadeira. Caso nenhum filho apresente uma precondição verdadeira, o nó seletor retorna falso. Os nós do tipo sequência executam o comportamento de todos os seus filhos. Cada filho deve ter sua precondição avaliada em verdadeira para que o nó pai seja avaliado verdadeiro. O nó do tipo *repita enquanto* estabelece uma estrutura de repetição para sua subárvore. Ou seja, o comportamento da sua subárvore será executado enquanto uma condição determinada seja avaliada em verdadeira.

As folhas da árvore implícita representam ações concretas no ambiente do jogo, comportamentos que mudam o estado do jogador ou do ambiente jogável. Por exemplo, um nó folha pode especificar um comportamento *Caminhe até o ponto* x e modifica a posição do jogador.

1.5 Fontes de busca

Os artigos citados foram buscados no Portal de Periódicos Capes[1] e na base de dados bibliográfica Scopus [2]. Os livros citados, (RABIN, 2013) e (BUCKLAND, 2005) foram acessados por meio de compra.

Na base Periódicos Capes, utilizando o termpo de pesquisa *game ai* foram encontrados 2.755.360 resultados. Esta pequisa resultou em um elevado número de resultados, portanto tentou-se novamente uma pesquisa com a string de busca *game ai 2d platform*,

[1] http://www.periodicos.capes.gov.br/
[2] https://www.scopus.com/

que resultou em 1770 ítens encontrados, onde 790 destes eram periódicos revisados por pares.

Já na base *Scopus* foi utilizado a busca avançada com a string *((game AND ai) OR (game AND artificial intelligence)) AND 2d platform))* com 66 resultados.

1.6 Revisão da Literatura

(PROMSUTIPONG, 2017) criou jogadores com IA utilizando Máquina de Estados Finita. Estes jogadores obtiveram performance similar a performance de jogadores humanos. A performance era medida em termos de taxa de acerto dos ataques e porcentagem de vida que a AI conseguiu subtrair dos inimigos.

(ASENSIO J. PERALTA, 2014) desenvolveu três controles para jogar o jogo *Unreal Tournament 2004*, onde os controles guiam os comandos que o jogador artificial utiliza para jogar. Para comparar a performance destas três abordagen, utilizou uma versão modificada do Teste de Turing, baseado no teste utilizado na competição internacional BotPrize, um teste em que jogadores humanos avaliam outros jogadores em termos de quais jogadores são bots e quais não são. A técnica de ADANN que fez uso de (Automatic Design of Artificial Neural Networks) foi a que obteve o melhor resultado.

(WANG, 2018) comparou a habilidade de jogadores artificiais em dois jogos clássicos, utilizando três técnicas, e as avaliou em termos de Taxa de Humanidade (em %, quanto maior, melhor) alcançada pelo agente. A técnica de melhor performance utilizou Redes Neurais (60% de performance), seguida de Máquina de Estados Finita (50%) e a técnica de pior performance (30%) consistia de apenas movimentos randômicos. Wang concluiu que no jogo 2D de atirar a performance foi superior ao jogo de corrida pois no primeiro jogo os movimentos envolvidos são em formas de reta em comparação com os movimentos de curvas (matematicamente mais complexos de serem modelados) no jogo de corrida.

Características como hostilidade, distância dos inimigos e tempo de reação refletem em quão humano um agente aparenta ser . Apesar de traços específicos (como agressividade) estarem relacionados com comportamentos de jogadores humanos, (HALL L. A. CENYDD, 2016) mostrou que nenhuma característica isolada é suficiente para justificar a humanidade de um NPC, isto é, a combinação de mais uma característica que resulta em um comportamento similar ao de um jogador humano. Apesar de não terem encontrado um conjunto de características que passasse no Teste de Turing, foram encontradas características que podem ser estudadas em outros tipos de jogos para proporcionar humanidade nos NPC.

Em 2012 Polceanu (2013) ganhou a competição *BotPrize*, com o *bot MirrorBot* e

foi a primeira vez que um agente passou neste teste de Turing adaptado da competição. A taxa de humanidade deste *bot* foi de 52%, valor maior ou igual a três (3) dos quatro (4) jogadores humanos. A implementação deste agente consistia de dois modos de jogo: *Default behavior* e *Mirroring behavior*. O modo *default* consistia de estados como Mirar, Atirar e Navegar, cada um implementado em um módulo independente. O segundo modo, *mirroring*, selecionava um determinado jogador e registrava todos os seus movimentos, e logo em seguida, repetia os mesmos movimentos simetricamente.

Durante a revisão da literatura não foram encontrados trabalhos construíram agentes autônomos que demonstrem comportamento humano em jogos de mesmas categorias que o jogo *Go Dash*, isto é, jogos *2D* de tiro, plataforma e *online multiplayer*. Ou seja, os trabalhos correlatos analisados compartilham com este aqui desenvolvido o objetivo de implementar jogadores artificiais que se pareçam com jogadores humanos e se diferem nos estilos de jogos em que as abordagens de agentes autônomos foram estudadas.

2 Metodologia

2.1 Ferramentas de desenvolvimento e ambiente

O jogo foi desenvolvido utilizando a *engine* Unity[1], na versão 2018.4.2. Versões executáveis do jogo foram geradas para os sistemas operacionais *Windows* e *Linux*, utilizando o suporte nativo da própria Unity.

Para a implementação do jogo, a linguagem de programação C# foi utilizada, na versão 3.0[2], que a é a linguagem suportada e recomendada para fazer uso da *engine* escolhida (Unity). A IDE *Rider*[3] foi utilizada para todo o desenvolvimento do jogo. Esta possui integração direta com a *Unity* oferecendo ferramentas como *debugger* e gerenciamento de arquivos desta *engine*. A ferramenta *git*[4] foi utilizada para o controle de versões de todos os arquivos envolvidos no jogo. O repositório do projeto foi mantido no gerenciador de repositórios *GitLab*[5].

O jogo utiliza *assets* do pacote *Sunny Land*[6] disponbilizado gratuitamente na Unity Store, e constitui a principal origem da arte do jogo, incluindo animações e *sprites*. A ferramenta GIMP[7] foi utilizada para fazer edições pontuais de imagens, como mudanças de cores e resolução de imagens. Além disso, figuras geométricas como ponto e retas foram produzidas em tempo de execução utilizando a própria *engine*.

2.2 Ambiente de execução do jogo

O editor da Unity fornece um ambiente de execução próprio do jogo, sem a necessidade de exportar o jogo para uma plataforma específica. Além disso, foram exportados binários do jogo para as plataformas Windows (x86-64) e Linux (x86-64), utilizando o mesmo editor.

2.3 Hardware

Plataformas de computadores pessoais, como *notebooks* e *desktops* foram os alvos da distribuição do jogo. Portanto, dois computadores foram utilizados para testar e exe-

[1] https://unity.com/
[2] https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history#c-version-30
[3] https://www.jetbrains.com/rider/
[4] https://git-scm.com/
[5] https://about.gitlab.com/
[6] https://assetstore.unity.com/packages/2d/characters/sunny-land-103349
[7] https://www.gimp.org/

cutar o jogo em todas as etapas do desenvolvimento do mesmo. O primeiro computador foi um *desktop* Windows 10 com processador $Intel\ Pentium\ G4560\ 3.5\ GHz$, 8 GB de memória RAM e placa de vídeo $NVIDIA\ GeForce\ GTX\ 1050\ Ti$. O segundo computador foi um *noteobok* $Latitude-3540$, utilizando o sistema operacional $Ubuntu\ 16.04\ LTS$, 8 GB de memória RAM, processador $Intel\ Pentium\ i7-4510U\ 2\ GHz$.

2.4 O jogo *Go Dash*

O jogo *Go Dash* é um jogo de tiro, 2D, em plataformas e *online*. No modo *Deathmatch* o objetivo de cada jogador é maximizar o número de eliminações de jogadores inimigos em um intervalo de tempo pré-definido. No modo *Capture the Flag* dois times competem entre si no objetivo de levar a bandeira localizada na base do time inimigo para a sua própria base, em uma melhor de cinco (MD5).

2.4.1 Desenvolvimento do jogo

Os módulos do jogo desenvolvido foram implementados utilizando a API da Unity, e podem ser interpretados como abstrações construídas utilizando a interface da *engine*. Por exemplo, foi desenvolvido o módulo de entrada para o jogo, com base no próprio sistema de entrada (*unity*) com base no próprio sistema de entrada disponibilizado pela Unity. Entretanto, a Unity é uma *engine* de propósito geral e possui sua arquitetura própria e de código propietário (código fechado) Unity 3D (2019a). Será discutido nas seções posteriores a arquitetura os componentes do jogo desenvolvido, mas não será discutido a arquitetura da *engine* em si.

A arquitetura do jogo é baseado na arquitetura de componentes. Os módulos principais são os de física, entrada, arte, negócio e os modelos de *bot*.

2.4.2 Módulo de entrada

O módulo de entrada processa os comandos que um jogador executa. Por exemplo, quando o jogador aperta no teclado de seu computador a tecla "Seta para cima", que faz com o que o jogador pule no jogo, o módulo de entrada registra que esta tecla foi apertada, e sabe distinguir entre as diferentes teclas do teclado. Para cada tecla válida (seta para cima, lados e baixo, por exemplo) o módulo de entrada delega para o módulo responsável por executar de fato a ação correspondente.

Para este jogo o módulo de entrada é de interesse particular pois o jogo foi desenvolvido com o objetivo de suportar a implementação de diferentes tipos de agentes autônomos para jogar o jogo. Sendo assim, o módulo de entrada deve ser responsável por abstrair diferentes tipos de comandos de entrada, por exemplo, uma tecla física no

computador é uma forma de entrada, mas a entrada poderia ser uma mensagem de outro processo, ou mensagem poderia vir através de uma chamada de função do mesmo programa, por exemplo.

O Código fonte 2.1 é a interface pública do módulo de entrada. Classes especializadas em cada tipo de entrada implementarão as funções desta interface. No caso dos jogadores humanos, que interagem com o jogo através do mouse e de um teclado físico, a classe responsável irá registrar os comandos pressionados pelos jogadores em seus dispositivos de entrada. Já no caso dos agentes autônomos os comandos de entrada serão chamadas de funções dos modelos que implementam o sistema do agente.

Código 2.1 – Código em C# da interface do módulo de entrada

```
1  public interface Player
2  {
3      Vector3 MouseDirection(Vector3 relative);
4      bool Fire { get; }
5      bool HookDown { get; }
6      bool HookUp { get; }
7      bool Downwards { get;  }
8      bool Upwards { get; }
9
10     bool Left { get; }
11     bool Right { get; }
12     float Horizontal { get; }
13 }
```

2.4.3 Módulo de física

O módulo de física é responsável por lidar com os deslocamentos de posição dos jogadores e as colisões envolvidas neste processo. Exitem duas fontes primárias de força no jogo, a gravidade e os comandos de entrada (como "andar pra direita") de cada jogador. Estas movimentações podem resultar em colisões, como um jogador caindo sobre uma plataforma, ou um projétil atingindo um jogador inimigo. As principais responsabilidades deste módulo são: processar comandos de entrada dos jogadores, mover objetos físicos considerando colisões, simulação de gravidade e implementar a mecânica de gancho dos jogadores.

Apesar da *engine* utilizada possuir um sistema de física 2D, esta é uma física realística que não atende os objetivos do jogo e este módulo se fez necessário.

2.4.4 Módulo de arte

O objetivo deste módulo é implementar a arte do jogo, através de *sprites* (como as plataformas, os projétils) e animações dos jogadores (como correr, pular e repousar). As plataformas do mapa do foram implementadas utilizando o sistema de *Tilemap* da Unity,

onde o espaço do jogo é particionado em um grid, e cada posição do grid é ocupada por um sprite Unity 3D (2019b). As animações devem ser executadas em resposta aos eventos do jogo, e este módulo é responsável por esta parte também. Por exemplo, quando o jogador pula existe a animação da parte da subida do pulo, e em seguida, da descida do pulo.

2.4.5 Módulo de negócio

O jogo possui suas regras de negócio, por exemplo, quando um jogador é atingido por um projétil inimigo, sua vida deve ser reduzida em 10% do valor inicial de vida. O módulo de negócio utiliza interfaces dos outros módulos para implementar as regras de negócio do jogo, e suas responsabilidades fundamentais são: instanciar e posicionar jogadores, tratar e reagir as colisões entre jogadores e projéteis e realizar a manutenção do placar do jogo.

2.4.6 Módulo de modelos de *bots*

Diferentes abordagens de sistemas autônomos serão implementadas e serão concretizada em *bots* que se interagem com outros jogadores e o ambiente jogável. Este módulo abstrai uma interface pública de um modelo genérico, com o objetivo de permitir reuso de código entre implementações destes modelos, e facilitar a alteração, adição ou remoção de abordagens para *bots*. Suas responsabilidades são: fornecer implementações da interface *PlayerInput* mostrada no Código 2.1, implementar sistemas movimentação, disparos e visão dos agentes, possivelmente reutilizando estes subsistemas. A implementação de cada modelo será discutida em específico na seção 2.4.7.

2.4.7 Modelos de agentes autônomos

2.4.7.1 Máquina de Estados Finita

Uma Máquina de Estados Finita será utilizada para modelar o comportamento do *bot*. Cada estado especifica um conjunto de ações que o agente está executando em um dado momento no jogo. As transiçõem representam as reações do *bot* aos acontecimentos dentro do jogo, como o ambiente, o próprio jogador e os jogadores inimigos. Por definição, em qualquer momento do tempo o agente se encontra em exatamente um (1) estado, ou seja, nunca em mais de um ou nenhum estado.

2.4.7.1.1 Estados

Investigar

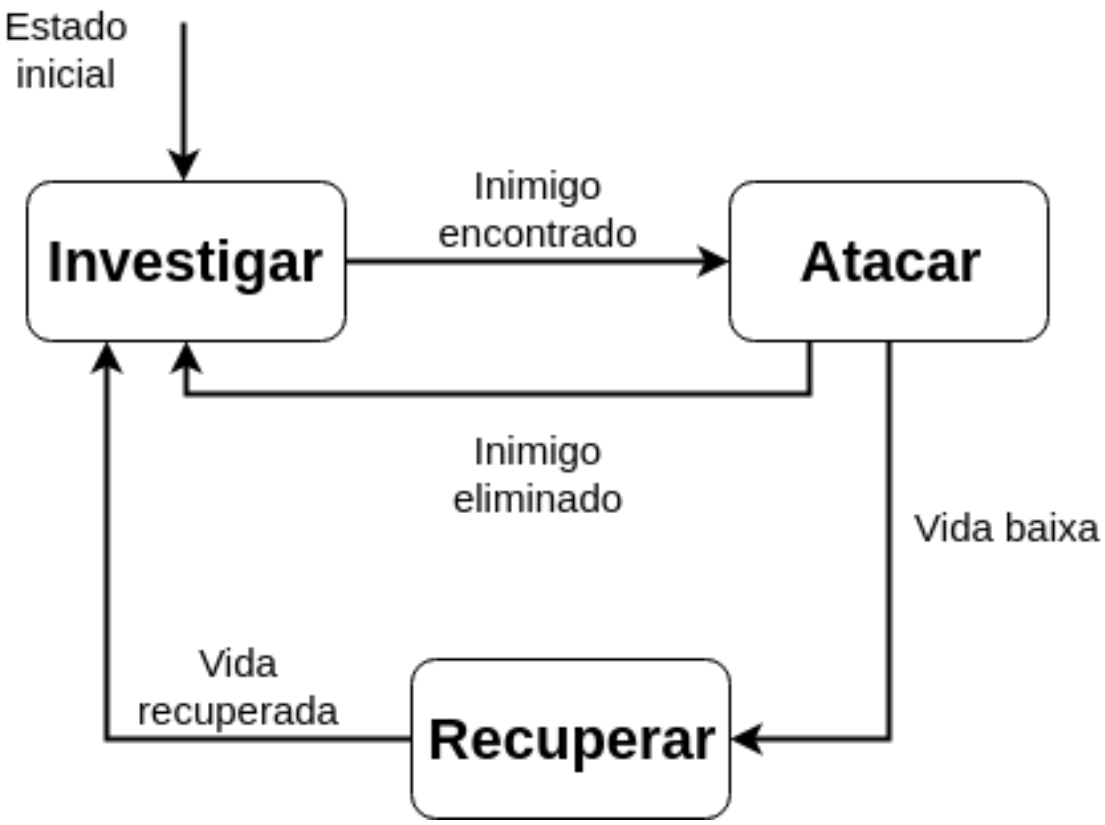

Figura 1 – Diagrama da máquina de estados para o *bot* jogar no modo *Deathmatch*

Este é o estado inicial do agente quando o jogo começa e o objetivo é localizar jogadores inimigos. O agente permanece neste estado indefinidamente até que pelo menos um jogador inimigo entre no seu raio de visão, a movimentação é feita na seguinte ordem:

1. É selecionado um ponto no espaço dentro do seu campo de visão
2. Inicia-se o processo de locomoção até o ponto selecionado no passo anterior
3. Ao chegar no ponto selecionado, este processo é reiniciado a partir do passo 1

Em paralelo a execução deste processo o campo de visão do agente é atualizado e é verificado a aparição de inimigos. Em caso afirmativo, a investigação é interrompida e é feita a transição para o estado de Atacar.

Atacar

O objetivo deste estado é maximizar o número de eliminações de jogadores inimigos. O agente realiza, em ordem:

1. É selecionado o inimigo com a maior probabilidade de ser eliminado
2. É feito o Ajuste de Mira

3. Efetua-se o disparo
4. Atualiza-se a posição do bot à fim de perseguir o inimigo
5. Volta-se ao passo 1

Diferentes abordagens nos subprocessos deste estados são possíveis. Por exemplo, o processo de selecionar o inimigo com maior probabilidade de ser morto pode ser feito escolhendo o inimigo mais próximo (mais fácil de acertar), ou o inimigo com a vida mais baixa, por exemplo. Existem duas vantagens em deixar os subprocessos do estado serem abstratos: a primeira é que é separado o modelo de sua implementação, isto permite modificar partes do modelo sem ser necessário alterar códigos que o implementa. A segunda vantagem é que códigos de subprocessos do modelo pode ser utilizado em outros modelos.

Duas transições são possíveis deste estado, e podem ocorrer em paralelo, à qualquer momento: todos os inimigos são eliminados, ocorrendo a transição *Inimigo eliminado*. O agente jogador fica com vida baixa e busca se proteger e se recuperar, transição *Vida baixa*

Recuperar

O objetivo deste estado é o agente não ser eliminado e recuperar sua vida. O *bot* então passa a procurar proteção dos ataques inimigos e buscando pelo mapa itens que recuperem sua vida. Para se recuperar, é realizado os seguintes passos, em ordem:

1. Procura-se no campo de visão itens ou utensílios que recuperam a vida do jogador. Caso haja pelo menos um, o mais próximo é selecionado como destino.
2. Caso não haja nenhum item de recuperação de vida no campo de visão, o ponto geográfico mais longe do do inimigo mais próximo é selecionado como destino.
3. Inicia-se o caminho até o ponto geográfico de destino selecionado em algum dos passos anteriores.
4. Caso a vida alcance 60%, este estado é encerrado. Caso contrafio, volta-se ao passo 1.

2.4.7.2 Transições

Inimigo encontrado

Esta transição ocorre quando o agente se encontra no estado *Investigar* e possui pelo menos um inimigo no seu campo de visão. A transição é feita para o estado *Atacar*.

Inimigo eliminado

Esta transição ocorre quando o agente se encontra no estado *Atacar* e o seu alvo atual é eliminado por algum jogador (independente se foi o próprio agente ou não). A transição é feita para o estado *Investigar*.

Vida baixa

Durante um combate (estado *Atacar*, é esperado que o *bot* perca pontos de vida. Ao abaixar de 30% do total de vida, é feita a transição para o estado *Recuperar*, o agente então corre risco de ser eliminado e passa a procurar por utensílios pelo mapa para que possa recuperar sua vida.

Vida recuperada

Quando o agente recupera sua vida para 60% ou mais do total é feita a transição para o estado *Investigar*.

2.4.8 *Behavior trees*

Esta seção apresenta um modelo de *Behavior tree* para modelar a tomada de decisão de um agente autônomo para jogar o jogo *Go Dash*.

Os objetivos alto nível são os mesmos da máquina de estados na seção 2.4.7.1: encontrar inimigos no mapa, eliminar inimigos, Recuperar vida e evitar sofrer danos.Apesar dos objetivos maiores serem idênticos, o processo de tomada de decisão para concretizá-los é diferente. Neste modelo é possível priorizar uma decisão, enquanto no modelo da máquina de estados, para uma dada entrada e um estado, a transição é sempre a mesma. Através dos nós do tipo *«Seletor»*, um conjunto de decisões são priorizadas.

A Figura 2 apresenta os nós e as arestas da árvore *behavior tree*. O tipo de cada nó, por exemplo *«Sequência* é especificado para cada unidade.

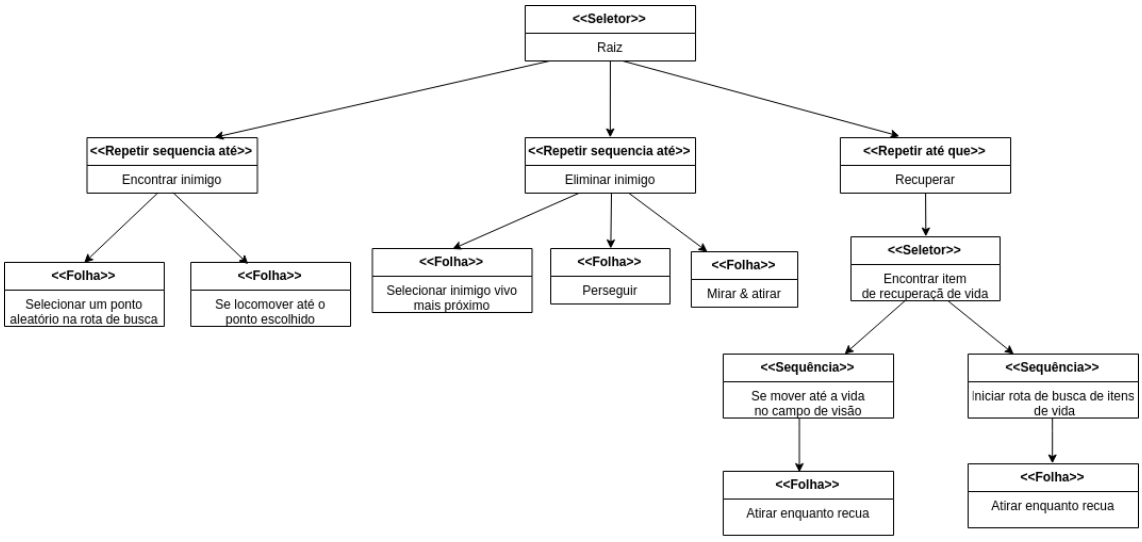

Figura 2 – Árvore do modelo *behavior tree* para o *bot* jogar no modo *Deathmatch*

2.4.9 *Mirroring*

Esta técnica consiste em copiar simetricamente os movimentos de um outro jogador, preferencialmente humano, por um determinado período de tempo. A hipótese desta técnica é que, ao copiar simetricamente (assim como num espelho) os movimentos de um jogador humano, o *bot* dará a impressão que está sendo controlado por um humano. Necessariamente o modelo de *mirroring* deve estar acompanhado por um ou mais modelos, pois se o *bot* copiar os movimentos de outro jogador durante todo o tempo, o agente acabará fazendo movimentos não humanos, como tentar atravessar uma parede. Um segundo problema é que, dado tempo suficiente, os jogadores humanos perceberão que estão lidando com um tipo de clone, e isso será percebido como um comportamento artificial. Portanto a abordagem escolhida foi combinar dois modelos, o de *mirroring* e o modelo de *behavior trees*, apresentado na seção 2.4.8. Uma máquina de estados controla qual dos dois estados está ativo em um dado momento. Por padrão, o *bot* irá estar no modo *default*, controlado pelo modelo da seção 2.4.8. Quando o agente enfrentar um candidato apropriado, ele irá fazer a transição para o estado de *mirror*, e passará a copiar simetricamente os movimentos do candidato apropriado.

2.4.9.1 Controlando os estados

A figura 3 mostra o estado e as transições da máquina de estados que controla o modelo de *mirroring*. O estado inicial é o *default*, e o agente permanece neste estado até que pelo menos um jogador inimigo entre no seu raio de visão. Quando é feita a transição para o estado de *mirror*, o agente pode retornar para o estado *default* pelo menos uma das seguintes condições são verdadeiras: o alvo foi eliminado, alvo saiu do campo de visão ou o tempo limite no estado de *mirror* de 10 segundos foi atingido. O valor de 10 segundos foi escolhido empiricamente e deve ser ajustado conforme o modo de jogo, número de jogadores no mapa e as características do mapa.

2.4.10 Campo de visão

Assim como jogadores humanos, o agente possui um campo de visão. Isto é, ele só possui informação do ambiente e dos jogadores do jogo que estão dentro do seu campo de visão. Jogadores humanos possuem um campo de visão retangular dentro do jogo, pois este é o formato mais comum dos monitores de computadores pessoais.

A visão do bot é um retângulo de dimensões de 1024×768 pixels. A câmera principal do jogo é fixa no personagem, e se move juntamente com o jogador. O jogador é o centro deste retângulo, e quanto se movimenta ele passa a enxergar novos pontos do espaço, e deixa de enxergar outros, assim como a visão dos jogadores humanos. Este valor foi escolhido pois é o mesmo valor da resolução alvo para o jogo ser executado. Isto significa que qualquer obstáculo, item ou outros jogadores que estão dentro ou nas bordas

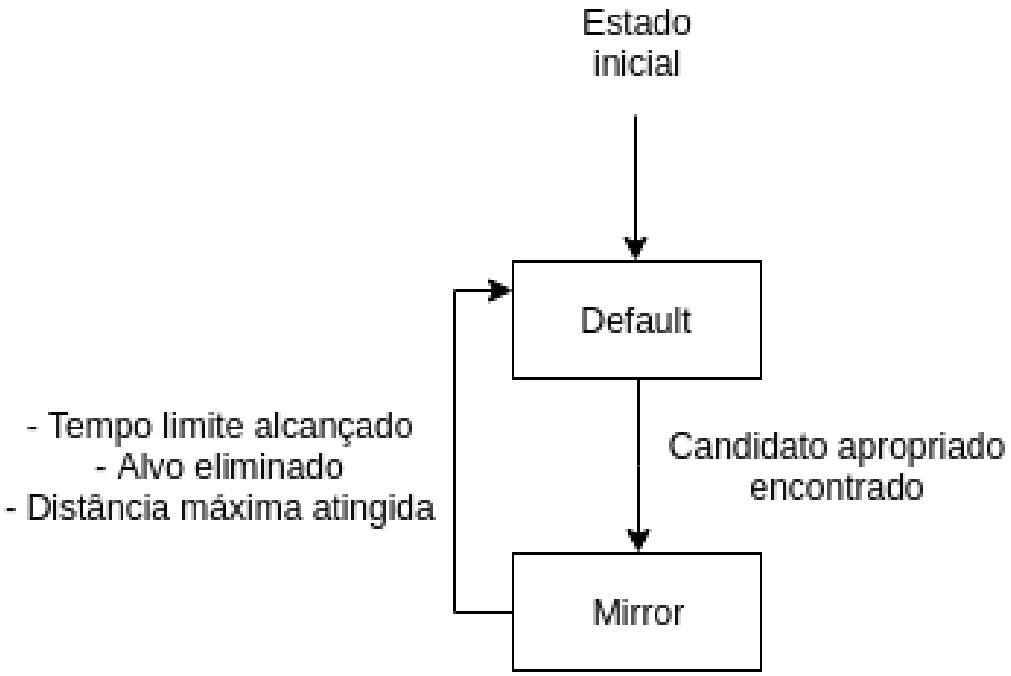

Figura 3 – Diagrama da máquina de estados para controlar o modelo de *mirroring*

deste retângulo são objetos visíveis para o agente e o mesmo pode reagir a estes objetos de acordo com o seu modelo.

2.4.11 Tempo de reação

Jogadores humanos não reagem instantaneamente aos eventos do jogo, portanto um agente autônomo que objetiva ser similar à jogadores humanos também deve exibir um tempo de reação humano. B. Richardson, D. Ellis, A. C. R. Greenwald, J. Cherry, C. Meador (2018) mostraram que o tempo de reação de jogares pode variar de acordo com o gênero e a experiência do jogador com o jogo, mas que na média, jogadores humanos não experientes demonstraram um tempo de reação de aproximadamente 346 milissegundos, e este será o tempo de reação utilizado nos modelos comparados. Isto significa que os agentes irão demonstrar reação exclusivamente para objetos que estão dentro do seu campo de visão e apenas após 350 milissegundos após o acontecimento do evento.

2.5 Testando a humanidade de um agente autônomo

Para avaliar a humanidade de cada modelo de *bot* no jogo *Go Dash*, é feito um teste onde quatro (4) jogadores se enfrentam no modo *deathmatch* (cada um por si) por dois (2) minutos. Ao fim deste tempo, cada jogador humano avalia os jogadores que enfrentou, categorizando cada jogador enfrentado como *bot* ou como humano. Ao final de

cada partida calcula-se a humanidade de cada jogador com um valor H em porcentagem:

$$H = \frac{h}{v} \tag{2.1}$$

onde $h, v \in \mathbf{N}$, h é o número de avaliações como humano que o jogador recebeu na partida e v é o total de avaliações que o jogador recebeu na partida.

Cada experimento é uma partida jogada onde o número de jogadores humanos varia de um (1) a quatro (4) e o número de *bots* varia de zero (0) a três (3). Por exemplo, um cenário de experimento é quatro (4) jogadores e nenhum *bot*, outro cenário seria dois (2) jogadores humanos e dois (2) jogadores artificiais. A Tabela 1 resume o tipo de jogador por experimento.

Tabela 1 – Número de jogadores humanos e artificiais por cenário de experimento

Cenário	**Número de humanos** $[1, 4]$	**Número de bots** $[0, 3]$
1	4	0
2	3	1
3	2	2
4	1	3

Para testar um determinado modelo de agente autônomo, por exemplo, o modelo de Máquina de Estados Finita, será feito três (3) experimentos para cada cenário da Tabela 1, totalizando doze (12) experimentos. Para cada experimento é conhecido a priori quais são os jogadores humanos e quais são os artificiais (caso haja).

3 Resultados e Discussões

Este capitulo apresenta os resultados obtidos. É apresentado o jogo e suas principais mecânicas e em seguida os modelos de agentes autônomos que implementam uma abordagem para construção de *bots* para jogar o jogo *Go Dash*.

3.1 O Jogo *Go Dash*

Para comparar diferentes abordagens de agentes autônomos, o jogo *Go Dash* foi desenvolvido com o objetivo de permitir com facilidade a adição e alteração de modelos de *bots*. Os jogadores podem se mover através de pulo único e duplo, *dash* verticais e horizontais e utilizando um gancho. Também podem efetuar disparos de longo alcance e todo o jogo pode ser jogado simultaneamente com outros jogadores, *online*. As Figuras 4 e 5 apresentam as principais localizações do jogo e um momento de combate entre jogadores.

Figura 4 – Combate entre jogadores, jogador de nome “Player3” foi eliminado. Tempo restante do modo de jogo ao centro e acima (0 minutos e 41 segundos) e o placar de pontuação

Figura 5 – Jogador utiliza um gancho para se movimentar ao longo das plataformas

3.2 Desenvolvimento do jogo

3.2.1 Módulo de entrada

A interface mostrada no Código 2.1 foi apresentada na seção . Foram desenvolvidas duas implementações desta interface, uma para jogadores humanos que utilizam teclado e mouse para jogar, e uma implementação para simular comandos de entrada de um *bot*. Esta segunda implementação é discutida na Seção 3.5.2.

3.2.1.1 Entrada para um jogador humano

A classe $PCPlayerInput$ captura os comandos de entrada de um jogador humano, que utiliza um teclado e um mouse para jogar. Esta classe é responsável, por exemplo, por responder requisições que perguntam qual a direção do mouse do jogador, se ele apertou o botão para atirar ou não, se o botão para acionar o gancho está pressionado para baixo. Sua implementação utiliza a própria API da Unity.

3.2.2 Módulo de física

A principal classe deste módulo é $PhysicsObject$, que representa um corpo genérico que está sujeito a física 2D do jogo. Objetos físicos devem herdar desta classe,

como é o caso da classe *PlayerControlller* que interage diretamente com diversos módulos, principalmente com o de entrada e o de física. Este módulo é responsável por receber requisições para mover um objeto numa direção bidimensional dada, aplicar a força da gravidade e tratar todas as colisões nestes movimentos.

3.2.3 Módulo de arte

O módulo de arte se deu, principalmente, na uso dos *sprites* e das animações dos movimentos do jogador.

3.2.3.1 Uso de *sprites*

O jogo utilizou o pacote de assets *Sunny Land*[1]. A manipulação destes *assets* foi utilizada na construção de todas as plataformas do mapa, utilizando o sistema de Unity 3D (2019b) da Unity, conforme exemplificado na Figura 6.

3.2.3.2 Animações

A animação do jogador foi feita utilizando o *Animator*, através do editor da Unity. A animação possui os estados de: pular para cima (*Player_jump_up*), pular para baixo (*Player_jump_down*), repouso (*Player_idle*) e correr (*Player_run*). A máquina de estados desta animação é apresentada na Figura 7.

3.2.4 Módulo de controle jogador

O objetivo deste módulo é implementar as movimentações e atributos dos jogadores humanos. Este módulo interage diretamente com outros módulos, como o de Física, Arte e *multiplayer*. No módulo de Controle de Jogador é feito o controle de vida do jogador, que se reage ao evento de ser atingindo por um projétil, onde o gacho é implementado e os comandos de entrada são processados. Aqui também se encontra parte do módulo de arte, pois é necessário informar o sistema de animações da Unity, o *Animator* [2] sobre os estados que o personagem se encontra, por exemplo, pulando ou não chão, em direção à esquerda ou a direita.

3.2.5 Módulo de modelos de *bots*

Este módulo implementa os modelos de agentes autônomos e operações que podem ser reaproveitas entre modelos diferentes e são independentes destes, como funções de atirar, enxergar e se mover.

Visão e tempo de reação

[1] https://assetstore.unity.com/packages/2d/characters/sunny-land-103349
[2] https://docs.unity3d.com/ScriptReference/Animator.html

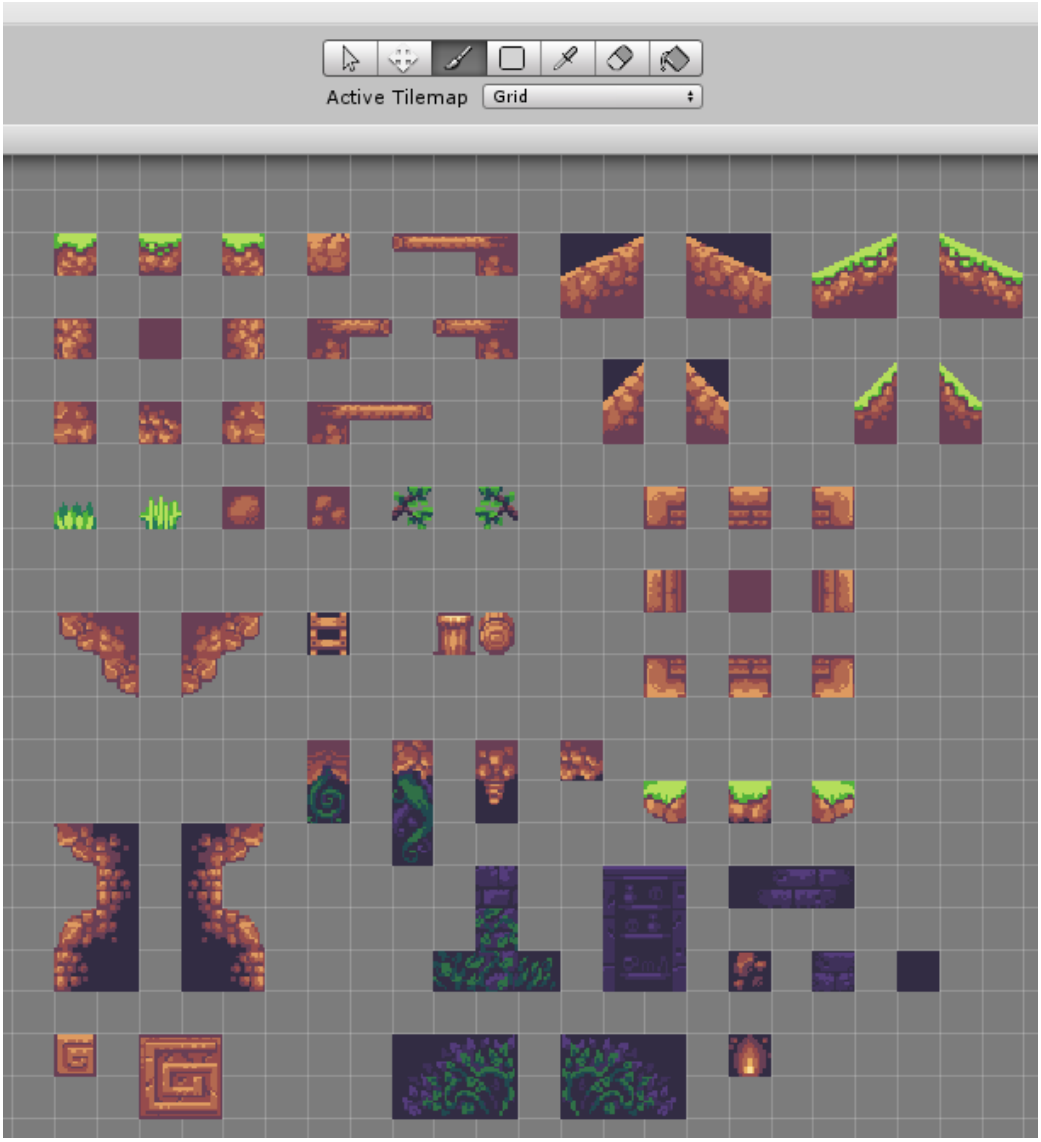

Figura 6 – Janela de *Tile Palette* no editor da Unity, utilizado para manipulação dos sprites do jogo.

Nas Seções 2.4.10 e 2.4.11 descrevemos que os agentes devem possuir um campo de visão limitada e um tempo de reação humano. Em implementação, conseguimos obter um resultado satisfatório neste sentido utilizando uma visão em formato circular de tamanho menor do que seria a visão de um jogador humano. Com este circulo, a visão do agente autônomo é limitada e ele não possui informação de fora do seu raio. Já o raio menor do que a de um jogador humano faz com que o agente reaja mais tarde, compensando desta forma o seu tempo de reação. O valor do raio em *pixels* foi obtido experimentalmente.

O Código 3.1 exemplifica o funcionamento da visão dos agentes: a função *ClosestPlayerInVision* retorna o jogador inimigo (incluindo *bots* e humanos) mais próximo e que esteja dentro do raio de visão. A implementação localiza o jogador inimigo mais próximo no mapa (fora ou dentro de visão) e verifica se a distância do agente até o jogador encontrado é menor ou igual ao raio de visão.

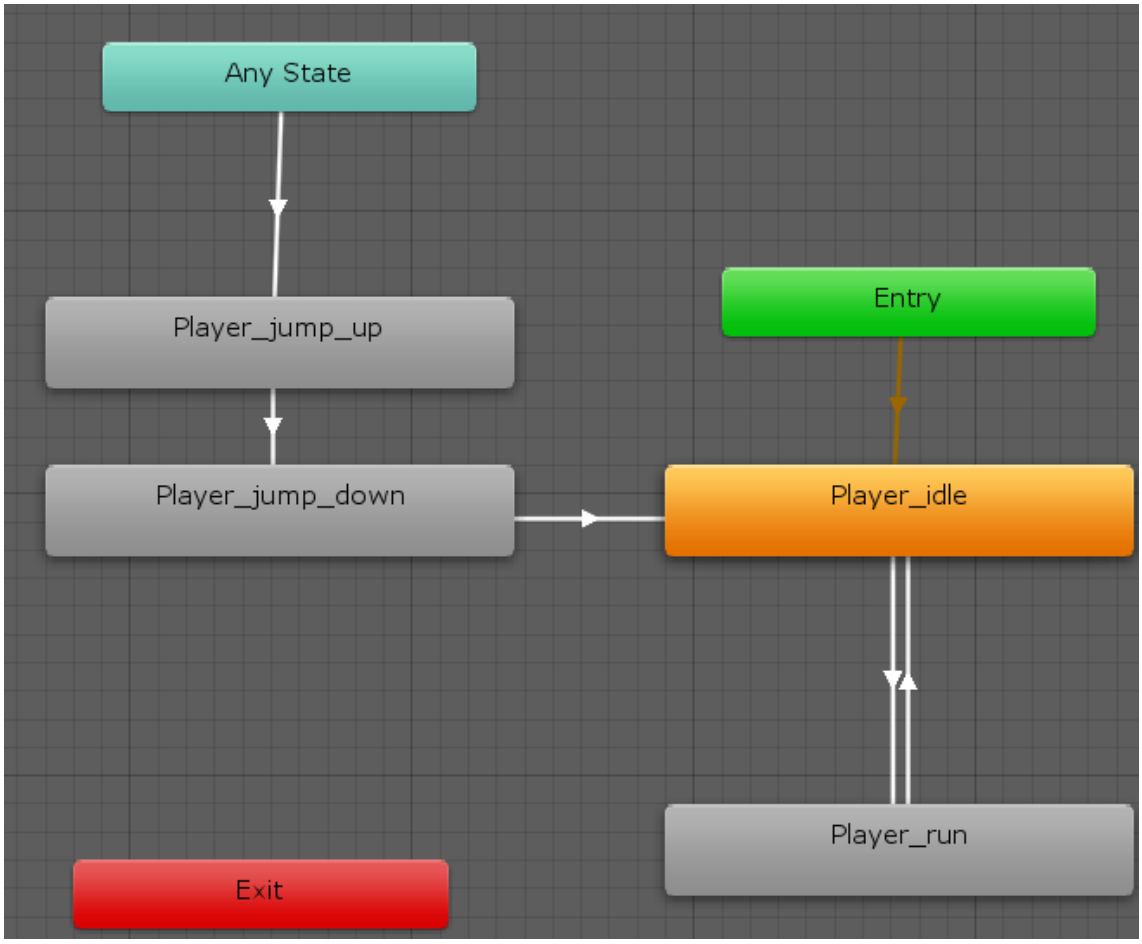

Figura 7 – Janela do *Animator* no editor da Unity, utilizado para representar a máquina de estados da animação do jogador.

Código 3.1 – Função de encontrar jogador inimigo dentro do raio de visão. Trecho de código do arquivo *AIVision.cs*

```
1  public bool ClosestPlayerInVision(GameObject obj, out Vector2 closest)
2  {
3      var closestFromPlayer = ClosestPlayer(obj);
4      closest = Vector2.positiveInfinity;
5      if (Vector2.Distance(obj.transform.position,
           closestFromPlayer.transform.position) <= VisionRadius)
6      {
7          closest = closestFromPlayer.transform.position;
8          return true;
9      }
10     return false;
11 }
```

Pathfinding (encontrando caminhos)

O objetivo deste componente é encontrar um caminho da posição atual do agente autônomo até o destino decidido pelo modelo que controla as decisões do *bot*, como uma Máquina de Estados Finita, por exemplo. Tentamos diferentes abordagens para imple-

mentar uma solução satisfatória de *pathfinding* ("encontrando caminhos" em tradução livre), estas tentativas e os detalhes da solução são discutido com profundidade na Seção 3.5. A principal função deste componente é encontrar um caminho até um destino e é implementada pela função *AStar*, implementação de do algoritmo A* sob um grafo implícito do mapa do jogo. A interface deste método, conforme o Código A.1, conforme apresentado na Seção A do Primeiro Apêndice, é receber a posição de origem $\overrightarrow{\boldsymbol{source}} \in R^2$, uma posição de destino $\overrightarrow{\boldsymbol{goal}} \in R^2$ e retornar um caminho até este, na forma de uma lista de arestas.

Tiros

Quando o *bot* decide atirar, ele seleciona o inimigo mais próximo dele, realiza um *raycast* com distância igual ao raio de visão, e verifica se o raio consegue atingir o inimigo. Se o raio consegue atingir o inimigo, isso quer dizer que não há obstáculos (como plataformas) entre o agente e o jogador inimigo, portanto ele realiza o disparo. Essa verificação é necessária pois jogadores humanos não vão atirar com frequência quando há um obstáculo óbvio entre seu jogador e o alvo. Tanto o raio do *raycast* e o disparo são feitos na direção do vetor diferença entre as posições do jogador inimigo e o agente. Este disparo ignora o movimento atual do alvo, não prevendo sua posição futura antes de atirar. Ainda sim o agente demonstrou boa precisão nos tiros.

3.3 Implementação do modelo de Máquina de Estados Finita

Foi proposto na Seção 2.4.6 três modelos para a construção de *bots*, e um teste na seção , entretanto este teste foi aplicado apenas no modelo utilizando Máquina de Estados Finita, conforme discutido na Seção 3.5.3.

A implementação do modelo de Máquina de Estados Finita utilizou dois estados em vez de três conforme originalmente planejado na Seção 2.4.7.1. O estado de *Recuperar* foi removido pois não foram implementados utensílios dentro do jogo que recuperam a vida dos jogadores. Discutiremos nas sessões a seguir o objetivo e a implementação dos estados e transições do modelo que estão resumidos na Figura 8.

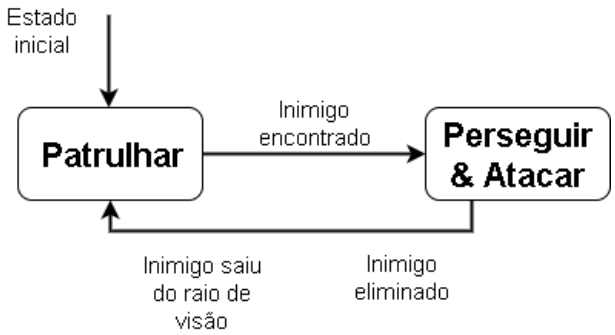

Figura 8 – Diagrama do modelo implementado utilizando Máquina de Estados Finita

3.3.1 Estado Patrulhar

O objetivo deste estado é fazer com que o agente se mova pelo mapa à fim de encontrar inimigos. A patrulha é realizada através da seleção aleatória de um ponto de interesse no mapa e movendo o agente até o ponto selecionado. Se durante o caminho um jogador inimigo for encontrado, é feito a transição para o estado de Perseguir e Atacar. Caso nenhum inimigo seja encontrado durante o caminho, um novo ponto de interesse é selecionado aleatoriamente e o agente continua no estado de Patrulhar por tempo indeterminado ou até que encontra um inimigo. Os Pontos de Interesse são um conjunto de pontos onde existe pelo menos um ponto por plataforma e são dispostos conforme a Figura 9.

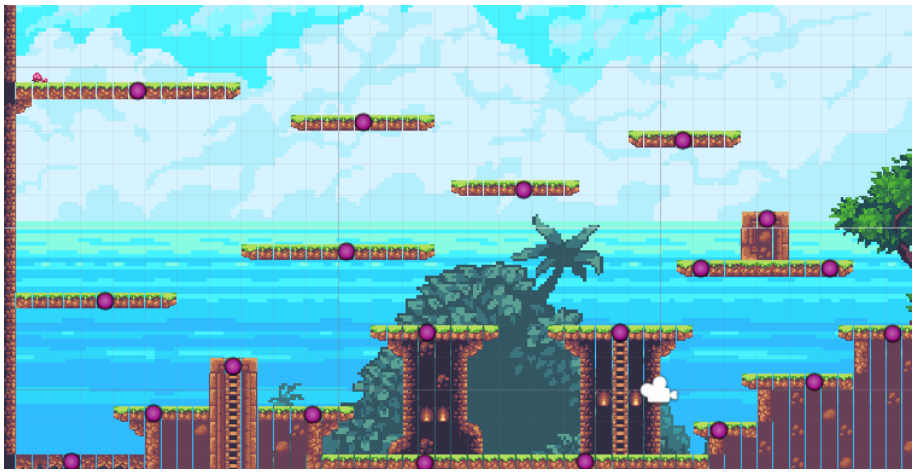

Figura 9 – Conjunto de pontos (destacados nos círculos de cor rosa) de interesse do mapa para a realização da patrulha

3.3.2 Estado Perseguir e Atacar

Assim quando pelo menos um jogador inimigo entra no raio de visão do agente, é feito uma transição para este estado. O inimigo mais próximo do *bot* dentro do raio de visão é escolhido como alvo. O agente então realiza duas ações em paralelo: atira em direção ao alvo (quando possível, conforme descrito na 3.2.5) e o persegue. A Figura 10 ilustra o momento em que um agente persegue e atira em direção a um alvo. A perseguição é feita selecionando um ponto próximo ao jogador alvo e iniciando um caminho até este ponto, utilização o componente de *pathfinding* para encontrar e seguir o caminho desejado. O agente pode trocar o seu alvo para um novo jogador inimigo caso um segundo inimigo fique mais próximo do agente do que o alvo atual. Quando não há nenhum jogador vivo dentro do raio de visão, é feita a transição para o estado de Patrulhar.

Figura 10 – *Bot* (Player2) no estado de Perseguir e Atacar atirando no jogador alvo (Player4)

3.4 Implementação do *multiplayer*

Toda a implementação do *multiplayer* foi utilizando a biblioteca UNet, que faz parte da própria Unity. Esta solução utiliza o paradigma cliente e servidor, mas permite que uma *build* consiga assumir o papel de cliente ou servidor, de acordo a selação feita no ício do jogo, conforme mostra a Figura 11. Ou seja, o servidor e os clientes são instâncias distintas de um mesmo programa.

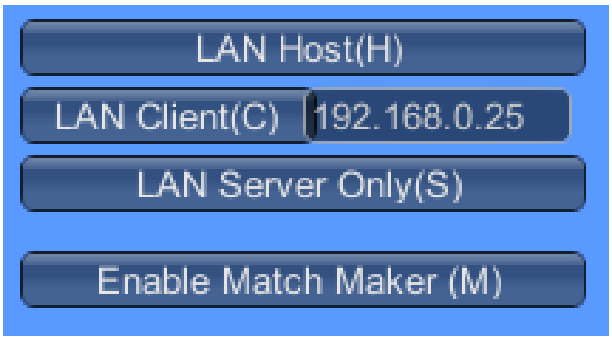

Figura 11 – Instância do jogo, jogador possui as opções de iniciar um servidor ou cliente

Como se trata do mesmo software tanto para clientes quanto para para servidores, é necessário separar em nível de código o que é executado em instâncias clientes e o que deve ser executado apenas no servidor. Clientes não podem se comunicar diretamente com outros clientes, portanto toda mensagem deve ser direcionada para o servidor, que por sua vez repassa a mensagem para todos os outros clientes conectados, sendo papel do

servidor sincronizar o estado de cada jogador.

Em código, funções que os clientes desejam que sejam executadas no servidor são chamadas *Commands*, e funções que o servidor deseja que todos os clientes executam são chamadas *ClientRpcCalls*, oriundo da sigla "*Remote Procedure Calls*" (RPC). O Código 3.2 exemplifica um cenário comum, onde uma instância de cliente deseja realizar um tiro e informa ao servidor a origem do tiro e o vetor de direção, para que o servidor possa criar o projétil em todos os clientes. Note que as funções que o cliente deseja executar no servidor são precedidas pela linha "[*Command*]".

Código 3.2 – Trecho de código responsável por iniciar um comando de tiro no cliente e repassar a mensagem para o servidor

```
1  [Command]
2  void CmdFire(Vector3 bulletPosition, Vector3 dir, NetworkInstanceId ownerID)
3  {
4          ServerFire(bulletPosition, dir, ownerID);
5  }
6
7  [Server]
8  void ServerFire(Vector3 bulletPosition, Vector3 dir, NetworkInstanceId
       ownerID)
9  {
10         var bulletObj = Instantiate(bulletPrefab, bulletPosition,
               Quaternion.identity);
11         var bullet = bulletObj.GetComponent<Bullet>();
12         bullet.Direction = dir;
13         bullet.ownerNetID = ownerID;
14         var bulletCollider = bulletObj.GetComponent<CircleCollider2D>();
15         Physics2D.IgnoreCollision(bulletCollider, playerCollider, true);
16         NetworkServer.Spawn(bulletObj);
17 }
```

Variáveis como a vida atual de cada jogador necessitam ter seus valores sincronizados entre o servidor e todos os clientes conectados. Para tanto, a UNet fornece um atributo decorador "[*SyncVar*]", que são variáveis compartilhadas entre todas as instâncias. Apenas o servidor consegue escrever nestas variáveis, e os clientes podem apenas ler. Por exemplo, no jogo a vida dos jogadores é sincronizada pela variável *hp*, conforme mostra o Código 3.3. Este código também contém um exemplo de servidor comunicando a todos os clientes que a vida de um determinado jogador mudou, através da RPC `RpcOnHealthChanged`.

Código 3.3 – Exemplo de variável sincronizada (*hp*) e *RpcClientCall*

```
1  [SyncVar] private int hp = 100;
2
3  [ClientRpc]
4  private void RpcOnHealthChanged(int newHP)
5  {
6          onHealthChanged(newHP);
```

```
7 }
```

Estes exemplos resumem as técnicas e tecnologias utilizadas para implementar toda a parte de redes do jogo.

3.5 Pathfinding (encontrando caminhos)

Os modelos de agentes autônomos estão ligados principalmente ao processo de tomada de decisão (atacar, fugir, procurar inimigos) dos jogadores artificiais. Por exemplo, o modelo pode concluir que o *bot* deve se mover da sua posição atual A até um ponto definido B. O caminho que o agente irá realizar para ir do ponto A ao B pode ser entendido como um problema à parte do modelo, ou seja, os modelos não necessariamente fornecem uma solução para este problema. Esta seção descreve as tentativas de implementação do módulo de *pathfinding* e detalha a implementação que obteve os melhores resultados.

A solução clássica para o problema de encontrar um caminho do ponto A ao B é representar o mapa do jogo como um grafo direcionado, onde os vértices representam posições físicas (x, y) dentro do jogo e as arestas indicam movimentos válidos que o jogador pode realizar. Os pesos das arestas em geral representam tempo ou distância para realizar tal movimento. Dado este grafo, utiliza-se algum algoritmo de encontrar caminhos de menor custo, como o Algoritmo de Dijkstra (CORMEN et al., 2009) ou o Algoritmo A* (NORVIG; RUSSELL, 2009). Entretanto, criar um grafo representativo do mapa do jogo é um problema específico, pois cada jogo possui as suas regras de movimentos válidos, física e mecânicas de movimentação (dois pulos, *dash*, gancho por exemplo).

3.5.1 Registro de movimentos de jogadores humanos

Uma abordagem para gerar o grafo representativo do mapa do jogo é registrar todos os movimentos feito por jogadores durante uma partida. Registra-se todas as mudanças de posições, juntamente com os comandos de entrada que resultaram nesta mudança de posição, dando origem a um grafo onde os vértices são pontos $(x, y) : x, y \in \mathbb{R}$ dentro do jogo e a aresta guarda informações sobre a distância destes e pontos e o comando de entrada. Esta abordagem demonstrou produzir rotas iguais ou similares a de rotas realizadas por jogadores humanos. Entretanto esta abordagem requer uma grande quantidade de registros de movimentações dentro do jogo para que todos os pontos dentro do mapa sejam cobertos por pelo menos uma rota e para que também haja variação de rotas entre dois pontos quaisquer. Outra desvantagem desta abordagem é que qualquer mudança nas plataformas do jogo foi necessário ter novos registros de movimentações. A implementação dos registros foi feita a cada quadro de física (calculado a cada 0.02 segundos). Registrou-se a posição do jogador em cada quadro, e quadros consecutivos

nos registros representavam uma aresta. Para encontrar uma rota de menor caminho, o algoritmo de A* foi utilizado, onde foi considerado que o peso das arestas era a distância euclidiana entre os dois vértices. Como não havia uma grande quantidade de dados sobre movimentações de jogador no jogo, esta abordagem foi descartada.

3.5.2 Simulação de comandos de entrada

Uma abordagem para gerar o grafo do mapa é simular os comandos de entrada de um jogador. Por exemplo, quando o jogador pressiona a tecla "D" ou Seta Para Direita, o jogo tenta mover o jogador para a direita, ao longo de vários quadros. A cada quadro, o módulo de física do jogo simula o movimento que o jogador está tentando fazer para tratar as possíveis colisões entre o jogador e as plataformas, como o chão e a parede. Este processo pode ser simulado, e os resultados intermediários podem ser registrados e representam um grafo do mapa. Neste grafo, os vértices são pontos (x, y) no mundo do jogo e as arestas representam qual tecla ou comando o jogador precisa pressionar para realizar o movimento. Por exemplo, a aresta (P, Q, i) representa que é possível o jogador ir do ponto P ao Q pressionando a tecla "i", onde i pode ser as teclas válidas do jogo como "A", "D", "Espaço" e "Mouse1".

A vantagem desta abordagem é que a sua implementação reutiliza código do próprio jogo, já que o processo de simular um comando de cada aresta é o mesmo que o jogo faria numa partida real. A desvantagem é o alto consumo de memória e tempo de execução, pois cada aresta representa um movimento do jogo de um ponto à outro durante um só quadro de física. Na Unity, cada quadro de física é calculado a cada 0.02 segundos, ou seja, para calcular um movimento de duração de 1 segundo, usaria-se pelo menos $\frac{1}{0.02} = 50$ arestas, correspondente ao número de quadros de física calculados em um (1) segundo.

Como este grafo possui um alto número de arestas e vértices, verificou-se que é inviável calcular rotas de menor caminho em tempo de execução, pois algumas rotas demoraram mais de 10 segundos para serem calculadas. Este tempo é inaceitável de se esperar, pois o *bot* ficaria imóvel durante todo cálculo da menor rota. Para evitar o cálculo de rotas durante tempo de execução do jogo, rotas específicas foram pré calculadas e registrada em um arquivo, que pode ser armazenado em disco e reutilizado durante as execuções. Esta abordagem resolve o problema de encontrar rotas de menor caminho em tempo de execução, mas o alto tempo de pré-processamento (cerca de 15 minutos) torna a solução pouco flexível, pois qualquer mudança no mapa ou na física do jogo demanda a computação de todas as rotas novamente, e por este motivo, optou-se por experimentar outras soluções.

3.5.3 Simulação de movimentos compostos

Uma limitação da abordagem de simular comandos de entrada, discutida na Seção 3.5.2, é que cada aresta representa um só movimento, feito durante um quadro de física, o que leva a um alto consumo de tempo e memória. Experimentou-se então utilizar arestas representando uma sequência (possivelmente longa) de movimentos em vez de um só movimento. Por exemplo, o movimento do jogador pular diretamente de uma plataforma à outra pode ser representado como uma só aresta nesta solução. Ainda que continue necessário simular a física de todo este movimento, o seu resultado final é registrado em uma única aresta, que pode representar dezenas de quadros de física, reduzindo significativamente o número de vértices e arestas do grafo. A Tabela 2 qual sequência de movimentos cada tipo de aresta representa.

Decidimos também remover as movimentações do jogo que envolvem *dash* (horizontais e verticais) e ganchos para reduzir o número de arestas no grafo, com o objetivo de reduzir o tempo de execução de algoritmos que buscam rotas de menor caminho. As movimentações de *dash* não adicionam complexidade significativa de implementação, já a movimentação utilizando gancho demonstrou ser significativamente mais complexa de se implementar, mas assumimos que um grafo que comporta movimentos utilizando gancho demandaria uma quantidade significativamente maior de recursos computacionais pois além de um novo tipo de aresta, tem-se novos vértices, ou seja, novos pontos no mundo do jogo que são alcançados apenas com movimentos de ganchxo.

Nesta abordagem, os vértices continuam representando pontos (x, y) no mundo do jogo, mas agora as arestas representam uma sequência de movimentos. Já os pesos das arestas, foram testadas duas formas: a primeira forma o peso era um número inteiro $w \in \mathbb{Z}$ que representava o número de quadros de física para realizar o movimento. Esta quantia é uma unidade de tempo pois cada quadro de física é calculado a cada 0.02 segundos, então rotas de menor caminho neste grafo resultavam em rotas que alcançavam o destino em menor tempo possível.

Um resultado peculiar de assinalar os pesos das arestas como o número de quadros entre os vértices é que os *bots* sempre andavam pulando para se mover horizontalmente ainda que o pulo em si não era necessário para se deslocar horizontalmente. Este resultado aconteceu pois a velocidade horizontal (a velocidade do personagem no eixo x) é a mesma quando se anda para a esquerda ou quando se pula para a esquerda, assim como para a direita. Havendo empate e sem nenhum outro critério de ordenação, as rotas com pulo tiveram prioridade na ordenação. Nunca andar sem pular mostrou ser uma forma artificial de se movimentar pelo mapa, pouco similar a forma que jogadores humanos se movimentam. Para evitar que os agentes sempre prefiram andar pulando, foi acrescentado uma constante $W = 10$ no peso das arestas envolvendo pulos. Desta forma, arestas de movimentos horizontais do vértice u ao vértice v tinham peso $w(u, v) = f$ onde f representa

o número inteiro de quadros de física entre os vértices e w é uma função que determina o peso entre estes nós. Já para arestas envolvendo envolvendo um pulo, o peso era de $w(u, v) = f + W$ e para pulos duplos o peso foi de $w(u, v) = f + 2W$. Com a adição desta constante, numa situação em que há diferentes rotas de menor caminho até o destino, as opções sem pulo iriam ter prioridade sobre aquelas que envolvem um pulo, que por sua vez teriam prioridade sobre aquelas que envolvem dois pulos.

Tabela 2 – Tipos de aresta por movimento e descrição de cada movimento

Tipo de aresta	**Sequência de movimentos**
left	Andar para a esquerda por 7 quadros ou enquanto não estiver tocando num chão
right	Andar para a direita por 7 quadros ou enquanto não estiver tocando num chão
jumpLeft	Pulo completo para a esquerda
jumpRight	Pulo completo para a direita
doubleJumpLeft	Pulo duplo completo para a esquerda
doubleJumpRight	Pulo duplo completo para a direita

O valor de sete (7) quadros foi obtido experimento diferentes valores. Valores próximos de um seria equivalente a simular movimentos horizontais a cada quadro, o que demanda um alto custo computacional. Por outro lado, valores muito altos pode resultar em movimentos horizontais muito longos, que não cobrem toda a superfície de todas as plataformas, pois cada movimento deve ser completo. Esta solução demonstrou ser mais rápida em tempo de execução, na ordem de dezenas, em relação a abordagem de simulação de comandos, entretanto não era rápida o suficiente para calcular rotas de menor caminho em tempo de execução. O algoritmo A* foi utilizado para encontrar rotas de menor custo, assim como na Seção 3.5.2.

A Figura 12 mostra uma visualização do grafo do jogo, onde os vetores desenhados na cor rosa representam movimentos envolvendo pulos, e vetores em cor amarelo são movimentos horizontais. A orientação dos vetores representam a ordem dos vértices nas arestas direcionadas, ou seja, cada vetor (de cor rosa ou amarela) presente na Figura 12 representa uma aresta no grafo do jogo. A visualização do grafo mostrou que existe uma grande redundância de arestas e vértices, ou seja, vértices muito próximos um dos outros que não possuem grande impacto no momento do jogo, por exemplo, se o *bot* deseja ir até o ponto $(0, 0.01)$, uma rota até o ponto $(0, 0.02)$ exibe um resultado visual e prático equivalente. As figuras 13 e 14 mostram os vetores do grafo com maior proximidade, onde pode-se observar a grande quantidade de vértices resultantes dos movimentos. Posições no mapa são representadas por números de ponto flutuante são necessárias pois as posições são resultados da simulação da física do jogo, que por sua vez realiza cálculos com números não inteiros. É necessário simular a física do movimento dos agentes para garantir que os jogadores artificiais se movem da mesma forma que jogadores humanos. Entretanto,

decidimos utilizar apenas duas (2) casa decimais para representar posições do mapa, ou seja, os pontos $(0, 3.009)$ e $(0, 3.011)$ são arredondados para o mesmo ponto, $(0, 3.01)$. Duas casa decimais foram suficientes pois mudanças de posições numa distância menor que 0.01 são praticamente imperceptíveis.

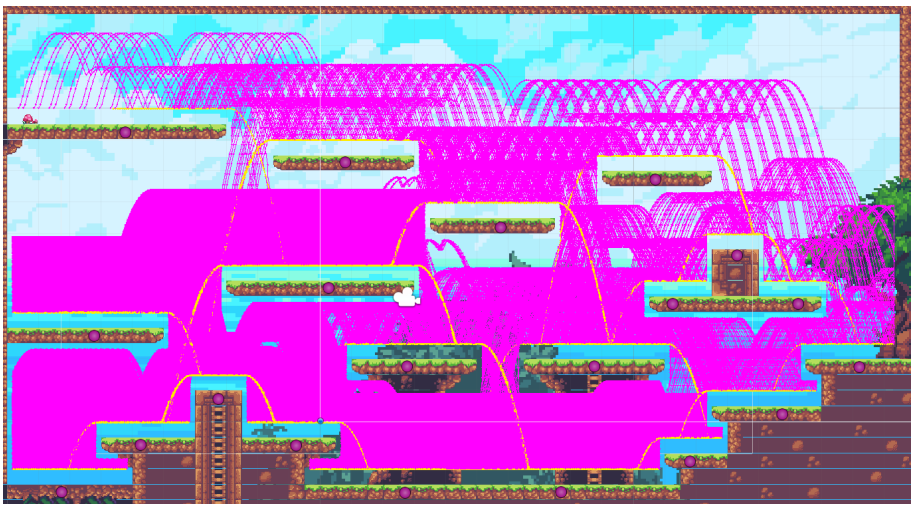

Figura 12 – Visualização do grafo do jogo resultante da execução do algoritmo A* para encontrar uma rota de menor custo

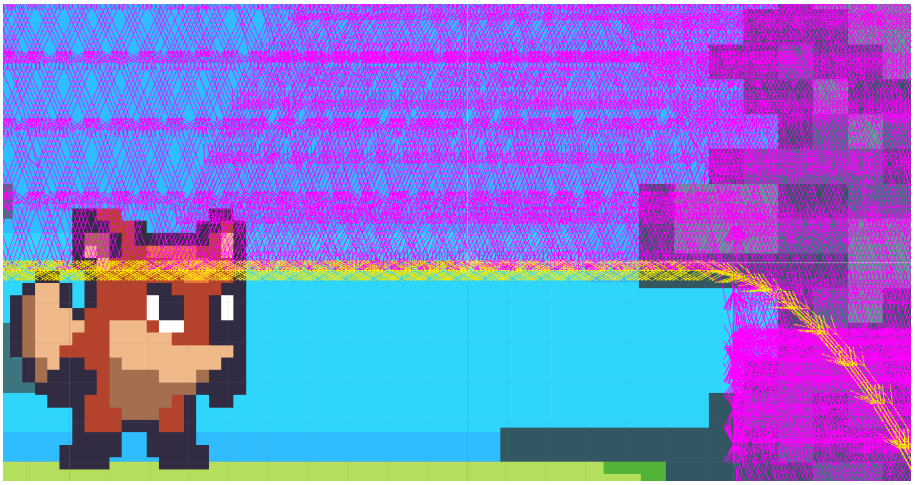

Figura 13 – Arestas do grafo em maior proximidade

3.6 Experimentos e Testes de Humanidade

A Seção 2.5 propôs um conjunto de experimentos para avaliar a humanidade de um determinado modelo de agente autônomo. Esta seção descreve os resultados dos experimentos para avaliar a humanidade de *bots* implementados utilizando o modelo de Máquina de Estados Finita descrito na Seção 2.4.7.1.

Ambiente físico e computadores

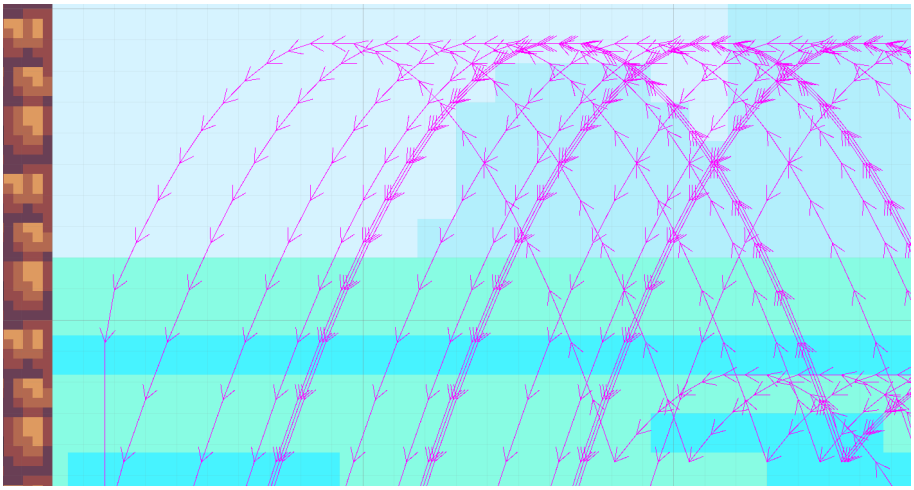

Figura 14 – Arestas do grafo em maior proximidade representando movimentos com pulos

Quatro (4) pessoas participaram dos experimentos. Cada participante usou um computador pessoal para participar do jogo. Os participantes estavam localizados em ambientes físicos separados entre si para evitar que se soubesse de antemão quantos jogadores humanos participariam da partida ou que recebessem algum tipo de informação que não fosse obtida dentro do próprio jogo. Cada participante obteve o software cliente do jogo através de *download* no link disponível em . O servidor do jogo e as instâncias cliente do jogo se conectaram na mesma rede de internet para que todos os clientes conseguissem se conectar diretamente ao servidor.

Início de partida

O operador do experimento iniciava o servidor do jogo, e para cada partida o servidor criava um número pré estabelecido de *bots*, de acordo com o planejamento do experimento. Em seguida foi solicitado aos participantes que se conectassem ao servidor através do jogo. Ao se conectarem, os participantes apertavam o botão de *Ready*, conforme a Figura 15, quando estivessem prontos e a partida iniciava quando todos os participantes apertassem o referido botão.

Identificação de cada jogador e placar

Os jogadores eram unicamente identificáveis, pois cada avaliador deveria categorizar cada jogador inimigo como humano ou *bot* ao final de cada partida. Para evitar que um participante fosse reconhecido pelo seu nome dentro do jogo, cada jogador (incluindo os jogadores artificiais) recebeu um nome aleatório ao se conectar na partida. Cada jogador recebia aleatoriamente um nome do conjunto $\{Player1, Player2, Player3, Player4\}$. O Placar do jogo marcava a quantidade de inimigos eliminados e a quantidade de mortes por jogador, e era atualizado dinamicamente durante a partida, conforme mostra a Figura 16.

Fim de partida

Figura 15 – Jogador *Player3* e o botão *Ready* para início de partida

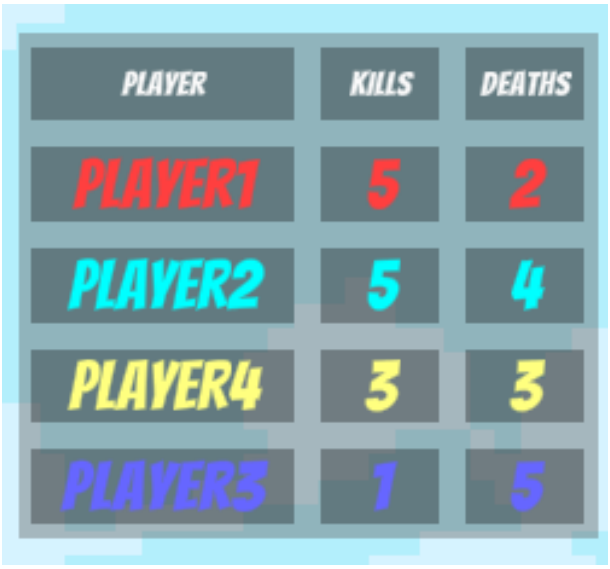

Figura 16 – Placar de pontuação de cada jogador durante a partida (Nome, número de eliminações e número de mortes)

Ao final de dois (2) minutos a partida se encerrava simultaneamente para todos os jogadores. Em seguida exibia-se no centro da tela os quatro (4) jogadores que participaram daquela partida, acompanhado dos seus nomes, conforme mostra a Figura 17, para que cada avaliador pudesse categorizar cada um dos jogadores inimigos como *bot* ou humano.

Folha de avaliação

Cada participante recebeu uma folha com uma tabela impressa, onde ao final de cada partida o jogador preenchia o nome recebido dentro do jogo e quais jogadores inimigos eram *bots* e quais não eram. A Tabela 3 é uma representação da folha que os participantes preenchiam. A primeira coluna identifica o número da partida e as demais colunas representavam o nome de cada jogador dentro do jogo. Cada participante do

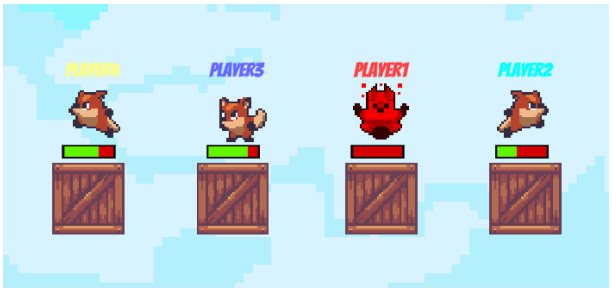

Figura 17 – Exibição dos quatro (4) jogadores presentes na partida ao final do jogo

experimento foi orientado que ao final de cada partida escrevesse nas células respectivas quem era ele próprio no jogo e quem ele acreditava que se tratava *bot*. As células em branco (não preenchidas) representavam os jogadores humanos na visão do avaliador. No exemplo da Tabela 3, o avaliador recebeu o nome "Player2" na segunda partida, avaliou o "Player4" como um jogador artificial e os demais jogadores como humanos.

Tabela 3 – Exemplo da folha de avaliação preenchida por participantes dos experimentos

Partida / Nome do jogador	**Player1**	**Player2**	**Player3**	**Player4**
1				Eu
2		Eu		Bot
...	...	...	...	...
12	Bot	Bot	Eu	Bot

Formato de cada experimento

Foram feitos 12 (doze) experimentos, ou seja, 12 partidas jogadas entre jogadores humanos e *bots*. Para simplificar as citações para cada tipo de cenário da Tabela 1, vamos nomear os cenários no formato $(x - y)$ onde x é o número de jogadores humanos e y é o número de jogadores artificiais em uma determinada partida. Por exemplo, o cenário (1-3) representa uma partida em que um (1) humano e três (3) *bots* jogaram uma partida. O planejamento era que cada cenário (4-0, 3-1, 2-2, 1-3) fosse repetido três (3) vezes cada, numa ordem previamente definida, e aleatória para evitar algum padrão que pudesse ser percebido pelos participantes. Entretanto, após uma partida no formato (4-0) um jogador abandonou o experimento por motivos pessoais. Sendo assim, optamos por continuar o experimento com apenas três jogadores humanos e não foram realizados novos experimentos no formato (4-0). A Tabela 4 identifica quais eram os jogadores artificiais em cada um dos experimentos. Por exemplo, no Experimento 1 não haviam *bots* enquanto no Experimento 7 os jogadores "Player2" e "Player3" eram artificais e os jogadores "Player1" e "Player4" eram humanos. Combinando as Tabelas 3 e 4 é possível extrair estatísticas sobre as avaliações feitas pelos participantes.

Calculando a humanidade dos jogadores por experimento

A Equação 2.1 mede a humanidade de um jogador, e também pode ser usada

para jogadores humanos. É interessante também mensurar a humanidade dos jogadores de maneira geral em um determinado experimento independentemente da humanidade calculada de um jogador em específico. Uma forma de fazer isso é considerar h como a soma de votos dados como "humano" que os *bots* receberam juntos e fazer o mesmo com os jogadores humanos. Assim, calcula-se quão humano (na visão dos avaliadores) os jogadores humanos foram em um determinado experimento, e quão humanos foram os jogadores artificiais. Para tanto, a Equação 3.1 calcula a humanidade dos jogadores humanos em um determinado experimento e a Equação 3.2 faz o mesmo para os jogadores artificiais:

$$H_{humans} = \frac{\sum_{i=1}^{4} h_i}{n(n-1)} \tag{3.1}$$

onde h_i é o número de avaliações como humano o jogador de nome $Playeri$ recebeu, e n é o número total de jogadores humanos presentes no experimento. Já para calcular a humanidade dos *bots*, utilizou-se:

$$H_{bots} = \frac{\sum_{i=1}^{4} h_i}{n(4-n)} \tag{3.2}$$

onde h_i é o número de avaliações como humano o *bot* de nome $Playeri$ recebeu, e n é o número total de jogadores humanos presentes no experimento. As Equações 3.1 e 3.2 não são aplicáveis quando $n \leq 1$, pois no caso de só um jogador humano presente, ele não recebeu nenhum voto, de modo que é descartado o caso em que não há pelo menos um humano jogando. A partir dos resultados experimentais e aplicando estas equações, tem-se os valores resumidos nas últimas colunas da Tabela 4.

Tabela 4 – Identificação dos jogadores artificiais por partida. Valor de humanidade por tipo de jogador por experimento.

Partida	h_1	h_2	h_3	h_4	**Bots**	H_{humans} (%)	H_{bots} (%)
1	2	1	3	1		58.3	-
2	2	1	2	1	4	83.3	33.3
3	1	0	1	1	1, 4	50	50
4	0	1	1	2	3	50	33.3
5	0	0	0	0	2, 3, 4	-	0
6	1	1	0	1	1, 3	100	25
7	0	2	1	1	2, 3	50	75.00%
8	2	0	1	0	2	50	0
9	1	2	2	2	1	100	33.3
10	0	0	0	1	2, 3, 4	-	33.3
11	1	2	0	2	3	83.3	0
12	1	1	1	2	3, 4	100	75

É importante observar que o número de votos que um jogador artificial recebe é diferente do número de votos recebidos por jogadores humanos, pois os participantes não votam em si mesmos, e os *bots* não avaliam ninguém.

Acerto médio por participante

Uma observação é que a taxa de acerto (julgar corretamente um jogador inimigo como humano ou *bot*) variou de acordo com o participante.

Tabela 5 – Acerto médio e desvio padrão por participante do ensaio

Participante / Estatística	**Acerto médio μ (%)**	**Desvio padrão σ**
Participante 1	55.5	35.1
Participante 2	88.9	31.4
Participante 3	75.0	22.1

O Participante 4 não foi incluso nesta Tabela pois não particiou durante todo o ensaio. Uma observação curiosa é que o Participante 2 anotou erroneamente quem era ele próprio em um experimento e obteve uma taxa de acerto de 0% naquele experimento. Este erro poderia ter sido evitado se o processo de avaliar os jogadores inimigos fosse feito dentro do jogo, *online*, ao final da partida. Ainda sim, se desconsiderarmos os votos deste participante neste experimento em específico, ele teria uma taxa de acerto de 100%, com desvio padrão $\sigma = 0$, ou seja, este avaliador consegui identificar corretamente todos os *bots* desde a primeira partida.

Análise qualitativa

Os resultados da Tabela 5 mostraram que a taxa de acerto (julgamentos corretos) variou entre os participantes do ensaio. Existe uma correlação entre a experiência prévia do jogador em jogos parecidos com sua taxa de acerto: quanto mais experiente em jogos do mesmo gênero que *Go Dash*, maior foi a taxa de acerto dos participantes. O Participante 1, por exemplo, relatou dificuldades de se movimentar e atirar simultaneamente durante o experimento, e se esforçou em tentar aprender e executar os comandos do jogo durante as primeiras partidas. Já os participantes 2 e 3 souberam rapidamente executar os comandos de movimentação dentro do jogo pois tinham experiência com jogos parecidos, e puderam concentrar seus esforços em identificar os jogadores artificiais. O Participante 1 comentou que observou alguns padrões nos jogadores artificiais: eles atiravam muito rápido (pouco intervalo de tempo entre os tiros), e apresentaram uma movimentação precisa, pulando entre as plataformas com precisão. O Participante 2 relatou que os tiros dos jogadores julgados como *bots* pareciam ser mais precisos mas não comentou sobre a movimentação em si.

3.7 Discussões sobre os experimentos

A primeira observação é que houve pouca variabilidade de jogadores: apenas três (3) jogadores humanos na maior parte dos experimentos. Uma quantidade maior de jogadores humanos poderia tornar a análise quantitativa mais confiável. Apesar da pequena quantidade de jogadores humanos, os participantes revelaram pontos que tornaram os jogadores artificiais menos humanos. A primeira característica não humana relatada foi que os tiros dos *bots* eram mais precisos e rápidos. Outra característica artificial dos agentes era que eles se movimentavam com maestria, executando pulos em sequência, que não pareciam naturais. Por exemplo foi comum pulos que imediatamente ao tocaram o chão da plataforma, outro pulo era executado em seguida, enquanto aparentemente, um jogador humano levaria mais tempo para executar o segundo pulo após tocar o chão.

O Participante 2 observou que os agentes artificiais nunca ficavam imóveis durante a partida, enquanto jogadores humanos ficavam imóveis ocasionalmente, por curto período de tempo, o que levava o Participante 2 imediatamente reconhecer estes jogadores como humanos.

A folha de avaliação fornecida aos participantes para que pudessem categorizar os inimigos durante a avaliação poderia ser substituída por um próprio sistema dentro do jogo. Este sistema evitaria confusões na hora de preencher a avaliação e permitiria estatísticas automatizadas sobre os resultados.

4 Considerações Finais

O trabalho mostrou o desenvolvimento do jogo *Go Dash*, em particular, o módulo de entrada demonstrou suportar diferentes abordagens de agentes autônomos, o que foi útil para a simulação de comandos de entrada no modelo de Máquina Estados Finita, conforme mostrou a Seção 3.2.1. Apesar de ser apresentado três modelos na Seção 2.4.7, apenas o modelo utilizando Máquina de Estados Finita foi implementado totalmente e submetido aos experimentos.

Segundo os experimentos, o jogador inexperiente conseguiu identificar corretamente o tipo dos jogadores inimigos (*bot* ou humano) cerca de 55% das vezes enquanto o jogador experiente obteve acerto médio de 88%. Apesar da pouca variabilidade de jogadores humanos nos experimentos, é possível concluir que os agentes não se comportaram de forma indistinguível dos humanos, já que os participantes experientes conseguiram identificar de forma consistente os jogadores artificiais. A análise qualitativa mostrou que a forma com que os *bots* se movimentam e atiram pareceram artificiais para os jogadores experientes.

Não é possível afirmar que a abordagem de Máquina de Estados Finita não pode produzir agentes com performance satisfatória para jogos do gênero *2D Shooter Multiplayer*, já que os pontos a melhorar identificados são relacionados com a implementação de subsistemas específicos, como movimentação e tiros. Entretanto este modelo demonstrou ser flexível para modificações em estados ou subsistemas.

4.1 Trabalhos futuros

Realizar os mesmos experimentos feitos com Máquina de Estados Finita nas abordagens de *Behavior tree* e *Mirroring* para comparar seus resultados entre si é a principal oportunidade de trabalho futuro. Além disso, a análise qualitativa dos experimentos demonstrou pontos a melhorar nos subsistemas de movimentação e tiros, o que seria viável já que o modelo experimentado, Máquina de Estados Finita, é flexível para adaptações em subsistemas específicos. Nos experimentos, as avaliações dos participantes foram feitas em folhas impressas, porém uma solução interna ao próprio jogo poderia evitar erros manuais dos participantes e também facilitar a coleta de dados dos experimentos.

Referências

AL., A. M. M. et. Creating autonomous agents for playing super mario bros game by means of evolutionary finite state machines. In: *Evolutionary Intelligence*. [S.l.: s.n.], 2014. Citado na página 16.

ASENSIO J. PERALTA, R. A. M. G. B.-P. C. A. L. P. J. M. L. Artificial intelligence approaches for the generation and assessment of believable human-like behaviour in virtual characters. In: *Expert Systems with Applications*. Barcelona, Espanha: [s.n.], 2014. v. 41, n. 16, p. 7281–7290. ISSN 0957-4174. Citado na página 22.

B. Richardson, D. Ellis, A. C. R. Greenwald, J. Cherry, C. Meador. *Reaction Times Differences in Video Game and Non Video Game Players*. 2018. [Online; acessado 16-junho-2019]. Disponível em: <https://digitalcommons.cwu.edu/cgi/viewcontent.cgi?article=1689&context=source>. Citado na página 33.

BUCKLAND, M. *Programming Game AI by Example*. [S.l.]: Wordware Publishing, Inc., 2005. 521 p. ISBN 1-55622-078-2. Citado 2 vezes nas páginas 20 e 21.

CONNOLLY T. HAINEY, E. B. G. B. P. M.-G. T. M. *Psychology, Pedagogy, and Assessment in Serious Games*. [S.l.]: IGI Global, 2013. 522 p. ISBN 14-6664773-6. Citado na página 19.

CORMEN, T. et al. *Introduction to Algorithms*. [S.l.]: The MIT Press; 3rd edition (July 31, 2009), 2009. 1292 p. ISBN 97-8026203384-8. Citado na página 44.

Gamecrate. *Statistically video game are now most popular and profitable form entertainment*. 2018. [Online; acessado 14-junho-2019]. Disponível em: <https://www.gamecrate.com/statistically-video-games-are-now-most-popular-and-profitable-form-entertainment/20087>. Citado na página 19.

HALL L. A. CENYDD, C. J. H. J. A. Identifying human-like qualities for non-player characters. In: COMMUNICATIONS IN COMPUTER AND INFORMATION SCIENCE. *Artificial Life and Intelligent Agents Symposium*. Birmingham, Reino Unido, 2016. v. 732. ISBN 978-3-319-90418-4. Citado na página 22.

HINGSTON, P. A turing test for computer game bots. 2009. ISSN 1943-0698. Citado na página 19.

LIVINGSTONE, D. Turing's test and believable ai in games. In: *Computers in Entertainment (CIE) - Theoretical and Practical Computer Applications in Entertainment*. [S.l.: s.n.], 2006. v. 4, n. 1, p. 93–104. ISSN 1544-3574. Citado na página 19.

NORVIG, P.; RUSSELL, S. *Artificial Intelligence: A Modern Approach*. [S.l.]: Pearson; 3 edition (December 11, 2009), 2009. 1152 p. ISBN 01-3604259-7. Citado na página 44.

OpenAI. *OpenAI Five is the first AI to beat the world champions in an esports game after defeating the reigning Dota 2 world champions, OG, at the OpenAI Five*

Finals on April 13, 2019.t. 2019. [Online; acessado 29-agosto-2019]. Disponível em: <https://https://openai.com/five/>. Citado na página 15.

ORTEGA N. SHAKER, J. T. G. N. Y. J. Imitating human playing styles in super mario bros. In: *Entertainment Computing*. [S.l.: s.n.], 2013. v. 4, n. 2, p. 93–104. ISSN 1875-9521. Citado 2 vezes nas páginas 16 e 19.

PETRI, B. S. L. Gameplay design patterns for believable non-player characters. In: *DiGRA '07 - Proceedings of the 2007 DiGRA International Conference: Situated Play*. Tóquio, Japão: [s.n.], 2007. v. 4, p. 416–423. ISSN 2342-9666. Citado na página 19.

POLCEANU, M. Mirrorbot: Using human-inspired mirroring behavior to pass a turing test. Nakhon Si Thammarat, Tailândia, 2013. ISSN 2325-4289. Citado 3 vezes nas páginas 16, 20 e 22.

PROMSUTIPONG, V. K. P. Enemy evaluation ai for 2d action-platform game. Nakhon Si Thammarat, Tailândia, 2017. Citado na página 22.

RABIN, S. *Collected Wisdom of Game AI Professionals*. [S.l.]: A K Peters/CRC Press, 2013. 626 p. ISBN 14-6656596-9. Citado 4 vezes nas páginas 15, 19, 20 e 21.

SCHRUM I. KARPOV, R. M. J. Ut^2: Human-like behavior via neuroevolution of combat behavior and replay of human traces. *Conference on Computational Intelligence and Games*, p. 169–186, 2011. ISSN 2325-4289. Citado na página 20.

TENCé C. BUCHE, P. L. O. M. F. The challenge of believability in video games: Definitions, agents models and imitation learning. 2010. Disponível em: <https://hal.archives-ouvertes.fr/hal-00514524>. Citado na página 19.

TURING, A. M. Computing machinery and intelligence. *Mind*, LIX, n. 236, p. 433–460, 1950. Citado 3 vezes nas páginas 16, 19 e 20.

Unity 3D. 2019. [Online; acessado 19-junho-2019]. Disponível em: <https://unity3d.com/unity>. Citado na página 26.

Unity 3D. 2019. [Online; acessado 19-junho-2019]. Disponível em: <https://docs.unity3d.com/Manual/class-Tilemap.html>. Citado 2 vezes nas páginas 28 e 37.

WANG, J. Classification of humans and bots in two typical two-player computer games. In: INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. *2018 3rd International Conference on Computer and Communication Systems (ICCCS)*. [S.l.], 2018. ISBN 978-1-5386-6350-9. Citado na página 22.

Appendix

APPENDIX A – Primeiro Apêndice

O Código A.1 recebe uma posição de origem, *source* e uma posição de destino, *goal*, que são utilizadas como vértices iniciais e finais para o algoritmo A*.

Código A.1 – Implementação do algoritmo A* para encontrar um caminho de menor custo de uma origem até um destino

```
1  public List<Edge> AStar(Vector2 source, Vector2 goal)
2  {
3      rb2d.position = source;
4      if (ReachedGoal(source, goal))
5      {
6          return new List<Edge>();
7      }
8      var sourceNode = new NodeStar(source, 0);
9      var gScore = new Dictionary<string, int> {[sourceNode.Name] = 0};
10     var fScore = new Dictionary<string, int> {[sourceNode.Name] = h(source,
           goal)};
11     var parent = new Dictionary<string, Edge>();
12     var activeVertices = new SortedSet<NodeStar> {new NodeStar(source,
           fScore[sourceNode.Name])};
13
14     while (activeVertices.Count > 0)
15     {
16         var u = activeVertices.First();
17         if (ReachedGoal(u.position, goal))
18         {
19             return BuildPath(parent, source, u.Name);
20         }
21
22         var removed = activeVertices.Remove(u);
23         var uGScore = gScore[u.Name];
24         var neighbors = Neighbors(u.position);
25         foreach (var v in neighbors)
26         {
27             if (v == null)
28             {
29                 continue;
30             }
31
32             var nameTo = NodeStar.NameStr(v.to);
33             if (!gScore.ContainsKey(NodeStar.NameStr(v.to)))
34             {
35                 gScore[nameTo] = int.MaxValue;
36                 fScore[nameTo] = int.MaxValue;
37             }
38             var newGScore = uGScore + v.weight;
39             if (newGScore >= gScore[nameTo])
```

```
40              {
41                  continue;
42              }
43              parent[nameTo] = v;
44              var vNode = new NodeStar(v.to, fScore[nameTo]);
45              removed = activeVertices.Remove(vNode);
46              var newFScore = newGScore + h(v.to, goal);
47              gScore[nameTo] = newGScore;
48              fScore[nameTo] = newFScore;
49              vNode.fScore = newFScore;
50
51              var added = activeVertices.Add(vNode);
52          }
53      }
54      return null;
55  }
```