Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Criação de um jogo com dificuldade dinâmica e aprendizagem por reforço

João Gabriel Basi

Monografia Final

mac 0499 — Trabalho de Formatura Supervisionado

Supervisor: Dr. Roberto Hirata Jr.

São Paulo
3 de dezembro de 2019

Criação de um jogo com dificuldade dinâmica e aprendizagem por reforço

João Gabriel Basi

Esta é a versão original da monografia
elaborada pelo candidato João Gabriel Basi,
tal como submetida à Comissão Julgadora.

Resumo

João Gabriel Basi. **Criação de um jogo com dificuldade dinâmica e aprendizagem por reforço**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2019.

Jogos do tipo "Player versus Environment"(PvE) são definidos por terem agentes inimigos autônomos que batalham contra o jogador, porém existem dois desafios que surgem na fase de desenvolvimento desses jogos: balancear o jogo de forma que a curva de aprendizado do jogador seja suave e criar os agentes de forma que eles pareçam inteligentes. Este trabalho pretende mostrar técnicas para resolver esses desafios através da criação de um protótipo de jogo PvE, desenvolvido usando o motor de jogos Godot, integrado com um interpretador Python para executar algoritmos de aprendizagem. Para resolver o primeiro desafio foi utilizado o algoritmo CAP, que quantifica a habilidade do jogador e a dificuldade do jogo, podendo assim equiparar a dificuldade do jogo à habilidade do jogador, garantido assim uma curva de aprendizado suave. Já para o segundo desafio foi utilizado o algoritmo de aprendizado por reforço *Q-Learning* em conjunto com redes neurais, possibilitando que os agentes aprendam como devem se portar por meio de iterações na fase de desenvolvimento.

Palavras-chave: Jogos eletrônicos. Adaptação de dificuldade. Aprendizagem de máquina. Aprendizagem por reforço. Redes neurais.

Abstract

João Gabriel Basi. **Creation of a game with dynamic difficulty and reinforcement learning**. Undergraduate Thesis (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2019.

Player versos Environment (PvE) games are defined by their battles between autonomous enemies agents and the player, however their development faces two main challenges: balance the game in a way that it provides a smooth learning curve for the player and create the agents in a way that they seem intelligent. This work's purpose is to show techniques to solve these challenges through the creation of a PvE game prototype, developed using Godot game engine, integrated with a Python interpreter for executing learning algorithms. CAP algorithm was used to solve the first challenge. It quantifies the player's skill and the game's difficulty, so it can match the game's difficulty with the player's skill, ensuring a smooth learning curve. Reinforcement learning algorithm Q-Learning along with neural networks were used to solve the second challenge, allowing the agents to learn how they should behave through iterations at development stage.

Keywords: Serious games. Adaptative difficulty. Machine learning. Reinforcement learning. Neural networks.

Sumário

Capítulo 1

Introdução

Jogos no estilo *Player versus Environment* (PvE), em contraste com jogos no estilo *Player versus Player* (PvP), são jogos onde o jogador combate vários inimigos controlados por inteligências artificiais para cumprir os objetivos do jogo. Nesse estilo, é comum que os desenvolvedores modelem os inimigos como sendo agentes autônomos, ou agentes inteligentes, como definido por Russell *et al.*, 2010, pelo fato deles tomarem decisões sem a intervenção humana.

A criação e o balanceamento dos agentes ocupa boa parte do desenvolvimento de jogos PvE, já que boa parte da experiência de jogo é proporcionada por eles. Para que o jogo se torne interessante, é desejável que os agentes ofereçam desafios fáceis o suficiente para que o jogador não fique frustrado, e difíceis o suficiente para que ele não fique entediado (Koster, 2013), e também que haja desafios novos a cada fase (Vorderer *et al.*, 2003).

1.1 Motivação

O desenvolvimento desses agentes pode ser uma tarefa difícil e trabalhosa se métodos tradicionais forem utilizados, como aponta Rasmussen, 2016. É necessário considerar todas as situações de jogo possíveis e sintetizá-las em um algoritmo complexo. Como consequência muitas vezes, ou a dificuldade de jogo fica acima ou abaixo do esperado, ou o algoritmo acaba tomando decisões que não foram previstas na fase de desenvolvimento, fazendo com que o jogador perca a imersão no mundo do jogo.

Alguns autores, como Andrade *et al.*, 2006 e Demasi e Cruz, 2003, mostram através de experimentos com usuários que jogos com dificuldade dinâmica são agradáveis ao jogador, porém é muito difícil encontrar jogos no mercado que de fato utilizam tais algoritmos de

adaptação.

1.2 Objetivos

Este trabalho tem como objetivo a criação de um protótipo de jogo PvE para computador que saiba adaptar sua dificuldade de acordo com a progressão e maestria do jogador, além de um sistema que facilite a criação de IAs para o jogo. Para isso o jogo contará com um sistema que identifica a proficiência do jogador e com agentes adversários que consigam aprender estratégias através de aprendizado de máquina.

Como algoritmos de aprendizado em geral requerem muito processamento, é necessário também adicionar a restrição que o processamento do jogo deve ocorrer rápido o suficiente para que a redução no número de quadros por segundo não seja perceptível, impactando minimamente a experiência do jogador.

Além disso, para que o jogo seja o mais simples e portátil possível ele não deve utilizar informações externas ao programa (como por exemplo os quadros da janela do jogo), dados de sensores físicos acoplados ao jogador ou ainda informações armazenadas na nuvem.

1.3 Descrição dos capítulos

O capítulo 2 introduz alguns conceitos de desenvolvimento de jogos que serão usados durante o trabalho e introduz também as tecnologias utilizadas na confecção do jogo, como o motor de jogos Godot e a biblioteca PyTorch.

Já o capítulo 3 introduz os conceitos matemáticos utilizados, como o algoritmo *Q-Learning*, para aprendizado por reforço, redes neurais artificiais, para aproximação de função, e o algoritmo CAP, para adaptação de dificuldade.

A história e a jogabilidade do jogo são descritas no capítulo 4, enquanto o capítulo 5 utiliza os conceitos dos capítulos anteriores para descrever a implementação do jogo, desde a persistência de dados, passando pelos algoritmos e organizações internas, finalizando com as interações visuais com o usuário.

Por fim o capítulo 6 conclui o trabalho e menciona como ele pode ser estendido no futuro.

Capítulo 2

Fundamentos Tecnológicos

Este capítulo introduz conceitos básicos de desenvolvimento de jogos, como armazenamento de imagens, loop do jogo, motor de jogos e nomenclaturas, introduz o funcionamento e a organização dos arquivos no motor de jogos Godot e mostra como foi feita a integração entre o motor e o interpretador Python, para que fosse viável executar algoritmos de aprendizado de máquina.

2.1 Terminologia básica de desenvolvimento de jogos

Nesta seção relembramos alguns conceitos e elementos básicos de um jogo.

2.1.1 *Sprite* e *spritesheet*

Qualquer imagem utilizada no jogo pode ser considerada uma *sprite*. No caso de animações, cada quadro da animação é considerado uma *sprite* diferente.

Um arquivo de imagem que contém várias *sprites* é chamado de *spritesheet*. Esses arquivos são feitos com o intuito de otimizar a leitura do disco, pois ao lê-lo o jogo já terá tudo que precisa na memória para reproduzir uma animação inteira por exemplo.

Nesse projeto os *spritesheets*, ou contém todas as animações de um determinado personagem, ou contém um *ninepatch* ou um *tileset*, porém em outros projetos é possível que os desenvolvedores queiram otimizar ainda mais a performance e escolhem por colocar em um mesmo *spritesheet* todas as *sprites* utilizadas em uma determinada tela do jogo. No

entanto isso requer um mapeamento das posições e tamanho de cada *sprite*, já que muito provavelmente elas terão tamanhos diferentes.

Figura 2.1: Spritesheet *com a animação de ataque do goblin.*

2.1.2 *Tile* e *tileset*

Uma *tile* é uma *sprite* que representa parte de um cenário, já um *tileset* é um *spritesheet* que contém *tiles*.

Tilesets podem ser criados seguindo o padrão 2x2 ou o padrão 3x3. No padrão 2x2 há quatro *tiles* para cantos externos, quatro para cantos internos, quatro para bordas retas, duas para diagonais e um para preenchimento interno, já no padrão 3x3 há uma *tile* para cada combinação de vizinhança 8 possível, com a restrição que uma *tile* em um canto implica que as duas *tiles* vizinhas dela e da *tile* de referência também estão ocupadas, totalizando 47 tipos de *tiles* diferentes.

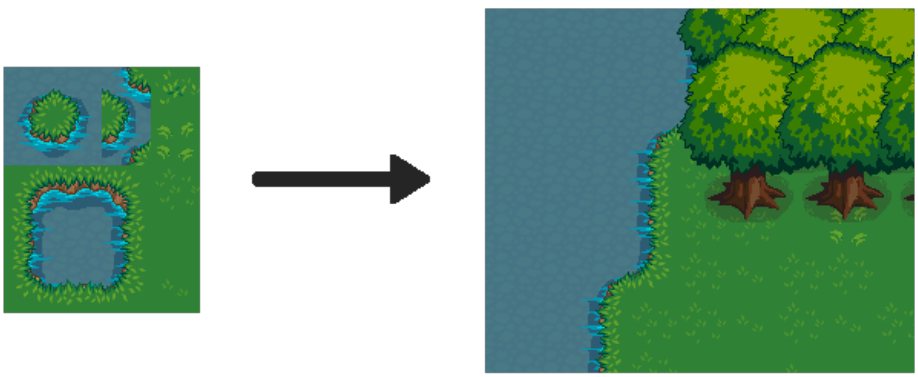

Figura 2.2: Tileset *2x2 (à esquerda) e exemplo de uso dentro do jogo (à direita).*

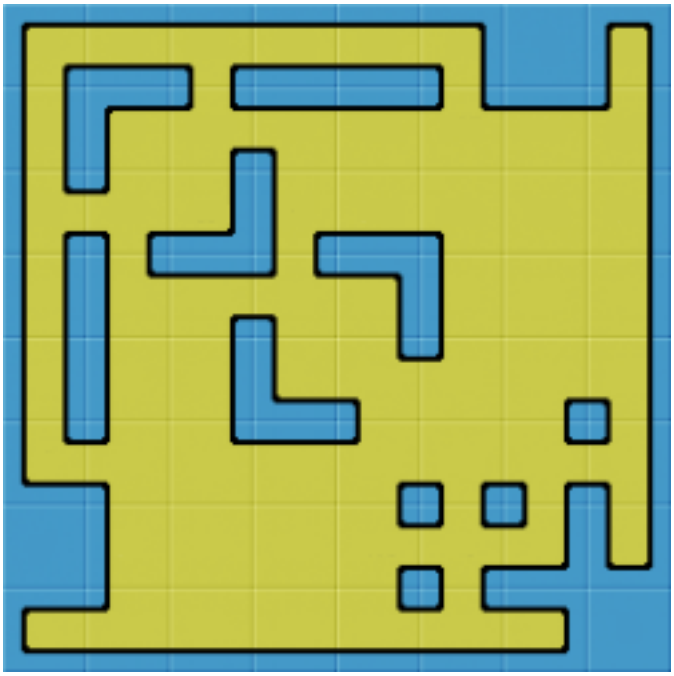

Figura 2.3: *Molde para criar* tilesets *3x3.*

2.1.3 *Ninepatch*

O *ninepatch* é uma forma de compactar os planos de fundo do jogo. Um *ninepatch* é constituído de nove *sprites*, quatro que representam os cantos da textura, quatro que representam as bordas e uma que representa o preenchimento interno.

Ao utilizar um *ninepatch* o jogo mantém as *sprites* dos cantos, replica as *sprites* das bordas e expande o preenchimento interno, criando assim um plano de fundo com o tamanho desejado.

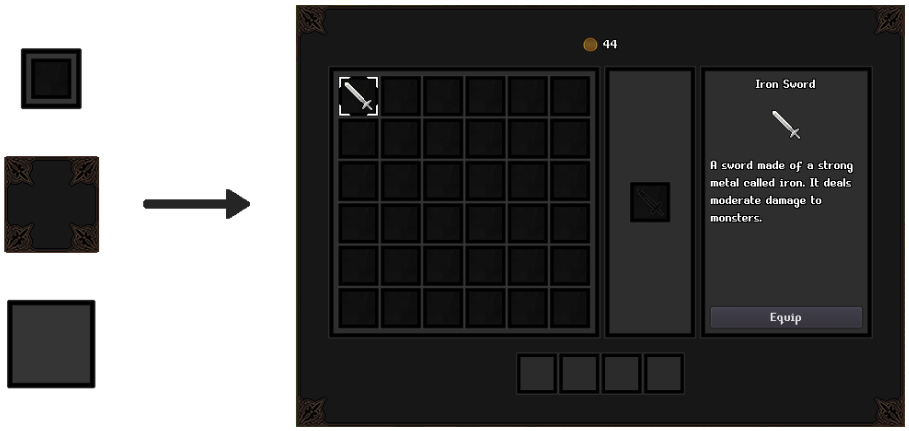

Figura 2.4: Popup *do jogo (à direita) criado usando* ninepatches *(à esquerda).*

2.1.4 Loop do jogo

O loop do jogo é um loop infinito que é o coração de todo jogo. Um loop de jogo simples executa os seguintes passos: Processar entrada, atualizar o estado do jogo e renderizar a tela, como mostrado na Figura 2.5.

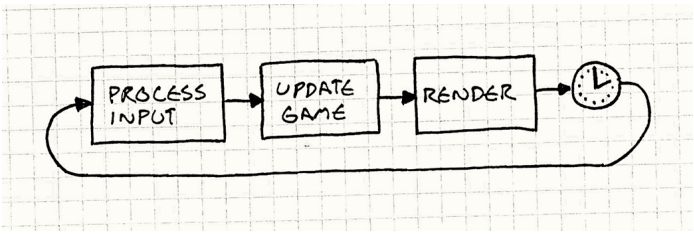

Figura 2.5: *Esquematização do loop do jogo (retirado de Nystrom, 2014)*

À medida que os jogos e a tecnologia foram se desenvolvendo, o game loop começou a tomar outras formas.

A função de renderização por exemplo pode ser executada em paralelo às funções de processar a entrada e atualizar o estado do jogo, já que a renderização precisa somente do estado atual do jogo. Em alguns casos as funções que atualizam a física do jogo são chamadas separadamente das funções que atualizam o resto do estado do jogo, fazendo com que esses processos possam ser executados em paralelo também.

Em máquinas diferentes com *hardwares* diferentes o loop pode rodar mais rápido ou mais devagar, causando diferenças nas velocidades de atualização do estado do jogo, por isso também é importante controlar o tempo de cada iteração dele. Na maioria dos casos isso é feito chamando uma função que faz o processador esperar um tempo até que o programa continue.

É comum nos motores de jogos que todo objeto do jogo tenha um método que é chamado quando o objeto é adicionado ao jogo, um método que é chamado na atualização do jogo e um método que é chamado na renderização do jogo. A ordem que esses métodos são chamados dentre todos os objetos do jogo depende da implementação do motor. Isso torna mais fácil controlar o comportamento de cada objeto durante cada passo do loop do jogo.

2.1.5 FPS

FPS é a sigla em inglês para “quadros por segundo”. É de conhecimento geral que as funções executadas pelo jogo têm que ser otimizadas, caso contrário haverão quedas no

FPS durante sua execução, o que incomoda o jogador.

Sempre que não é possível contornar o problema de performance é necessário sinalizar para o jogador que o jogo está processando conteúdo utilizando uma tela de carregamento.

2.1.6 Tela

Tela é o termo genérico que se refere a conjuntos de objetos do jogo que ocupam a tela inteira, como por exemplo menus, *popups* ou mapas do jogo.

2.1.7 Motor de jogos

Um motor de jogos, também chamado de *game engine* em inglês, é todo programa produzido com a intenção de facilitar a criação de jogos do começo ao fim. Motores de jogos mais populares, como Unity3D[1], Unreal[2], GameMaker[3] e Godot[4] têm interfaces gráficas e ferramentas para auxiliar ao máximo não-programadores na criação, enquanto outros como Phaser[5], Pygame[6] e LÖVE[7] se parecem mais com arcabouços, focados somente nos programadores.

Algumas funções essenciais em motores de jogos são a capacidade de abrir uma janela e renderizar *sprites* nela, a capacidade de reproduzir vários arquivos de som ao mesmo tempo, a capacidade de simular interações físicas entre formas geométricas, a capacidade de tratar a entrada do usuário, seja ela pelo teclado, mouse ou qualquer outro dispositivo de entrada, e a capacidade de criar um executável contendo todo o conteúdo necessário para a execução do jogo.

2.2 Godot

A principal ferramenta utilizada no projeto foi o motor de jogos Godot. O Godot, escrito em C++, foi criado em 2007 pelos argentinos Juan Linietsky e Ariel Manzur e tornou-se

[1]Site principal do Unity3D: https://unity.com/solutions/game/ (Último acesso em 16/11/2019)
[2]Site principal do Unreal: https://www.unrealengine.com/en-US/ (Último acesso em 16/11/2019)
[3]Site principal do Unreal: https://www.yoyogames.com/gamemaker (Último acesso em 16/11/2019)
[4]Site principal do Godot: https://godotengine.org/ (último acesso em 10/10/2019)
[5]Site principal do Phaser: https://phaser.io/ (último acesso em 10/10/2019)
[6]Site principal do Pygame: https://www.pygame.org/ (Último acesso em 19/11/2019)
[7]Site principal do LÖVE: https://love2d.org/ (último acesso em 10/10/2019)

código aberto em 2014. O motor pode ser usado para criar tanto jogos 2D quanto 3D e seus jogos podem ser exportados para várias plataformas, incluindo OSX, Linux, Windows, Android, iOS e Web. Neste projeto foi utilizada a versão 3.0 do Godot por dependência de outras bibliotecas.

O Godot possui dois tipos de arquivos principais: as cenas, que que contém uma hierarquia de objetos gráficos e físicos, e os *scripts*, que contém a parte lógica do jogo implementada pelo desenvolvedor. Para editar as cenas o programa oferece uma interface gráfica com visualização em tempo real dos objetos da árvore, já para a parte lógica ela oferece um editor de texto e uma linguagem de programação própria, chamada GDScript[8].

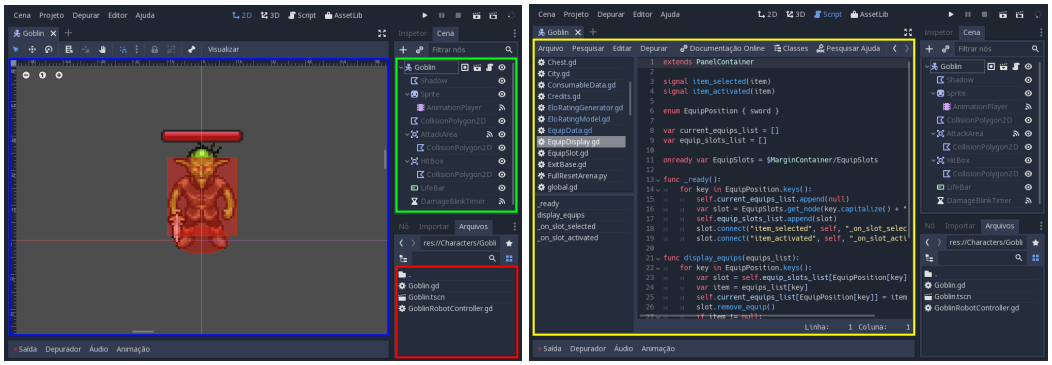

Figura 2.6: *Fotos do editor do Godot. A área em azul se refere à área de pré-visualização da cena, a em verde à árvore de nós da cena, a em vermelho ao gerenciador de arquivos e a em amarelo ao editor de texto.*

2.2.1 Nós

O nó é a unidade básica para a construção de um objeto no jogo. Cada nó é constituído de um tipo, um *script*, uma lista de grupos, uma lista de sinais e uma lista de filhos.

O Godot disponibiliza vários tipos de nós, cada um com um conjunto de atributos e propriedades, que podem incluir interações com outros nós. Alguns exemplos são o nó `Sprite`, que pode ser associado à uma *sprite* de uma *spritesheet*, mostrando-a na tela quando o nó é instanciado, e o nó `KinematicBody2D`, que controla a física de um corpo móvel e necessita de ao menos um nó filho do tipo `CollisionPolygon2D` ou `CollisionShape2D` para definir sua forma de colisão. Tipos personalizados podem ser criados usando GDScript ou C++ e adicionados ao motor por meio de extensões.

Os grupos de um nó são equivalentes à rótulos, e são utilizados para agrupá-los. É possível testar se um nó pertence à um grupo e recuperar todos os nós pertencentes à um

[8]Documentação do GDScript: http://docs.godotengine.org/en/3.0/index.html (último acesso em 10/10/2019)

grupo via *script*.

Os sinais são eventos associados aos nós. Um sinal pode ser associado à funções de outros nós, assim, quando o sinal é emitido, todas as funções associadas à ele são executadas. Um exemplo de sinal é o sinal `pressed` do nó `Button`, que é emitido sempre que o botão é pressionado com o mouse ou com a tecla enter.

Nós podem ter um conjunto de nós filhos, formando assim uma estrutura de árvore, explicada melhor na seção 2.2.2.

Nós podem ter também um *script* associado, que é explicado melhor na seção 2.2.3.

2.2.2 Cenas

Uma cena do Godot é uma árvore de nós. Cenas podem ser salvas em arquivos e podem ser instanciadas como filhas de um nó pertencente à outra cena. Essa organização de nós faz com que o jogo na visão do Godot nada mais seja que uma árvore de nós, na qual cada cena instanciada é uma sub-árvore.

2.2.3 Script

O *script* é um arquivo contendo código GDScript que é utilizado para criar parte lógica de um nó. Ele se comporta como uma classe que pode ser instanciada por um nó ou instanciada ou herdada por outro *script*. Todo *script* deve herdar de um tipo de nó do Godot ou outro *script* (herança múltipla não é permitida).

Todo *script* têm alguns métodos abstratos que podem ser implementados, como é o caso do método `_ready`, que é executado assim que o nó é colocado na árvore do jogo, ou os métodos `_process` e `_physics_process`, que são executados a cada mudança de estado do jogo.

2.2.4 Variáveis exportadas

Os *scripts* podem conter variáveis exportadas. Tais variáveis aparecem na interface gráfica do Godot e podem ser modificadas a cada instância do nó que contém aquele *script*. Isso facilita a modificação dos atributos dos nós por não-programadores, já que não é preciso modificar o *script* para isso.

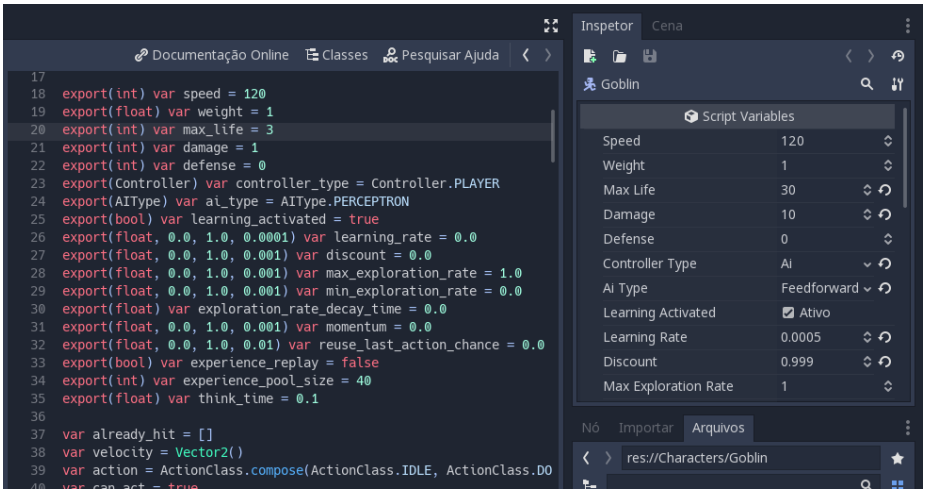

Figura 2.7: *Variáveis exportadas definidas no* script *(à esquerda) e valores definido no nó raiz da cena* `Goblin` *(à direita). Note que é possível definir tipo, intervalo de valores e valor padrão para as variáveis através do* script.

2.2.5 GDScript

A linguagem GDScript é uma linguagem de alto nível, interpretada e dinamicamente tipada criada especificamente para ser utilizada no Godot. Segundo a documentação do motor, seus criadores escolheram criar uma linguagem nova pois tiveram dificuldades em integrar outras linguagens como Lua, Python e Squirel no motor, além de melhorar a performance de execução do jogo, já que é possível otimizá-la especificamente para a arquitetura do Godot. Para que a linguagem fosse fácil de ler e aprender eles a criaram com a sintaxe muito parecida com a de Python.

2.3 Python

Nesta seção explicamos como a linguagem Python foi usada na implementação do jogo.

2.3.1 Pytorch

Nos scripts Python foi utilizado a versão 1.1.0 da biblioteca de aprendizado de máquina PyTorch[9], para facilitar o desenvolvimento de IAs baseadas em aprendizado de máquina e viabilizar seu uso, já que o GDScript não possui bibliotecas com essa finalidade.

[9]Site principal do PyTorch: https://pytorch.org/ (último acesso em 13/10/2019)

2.4 Extensão Pythonscript para Godot

Junto ao motor foi utilizado a versão 0.11.3 da extensão Pythonscript[10], disponível na *Asset Library* do Godot. A extensão criada por Emmanuel Leblond oferece uma interface entre o Godot e o interpretador Python, fazendo com que os *scripts* possam ser escritos em Python e bibliotecas Python possam ser utilizadas. A extensão suporta várias versões de Python, mas nesse projeto foi utilizada a versão 3.6.

A integração entre as duas linguagens provida pela extensão não é muito performática quando se trata de chamadas de função. Para ilustrar essa deficiência foi feito um teste em um notebook comum com processador Intel i7 no qual a função identidade é chamada em loop um milhão de vezes. A tabela 2.1 mostra o tempo de execução desse programa quando a função identidade é implementada na linguagem da primeira coluna e é chamada através linguagem da segunda coluna.

Chamada da função	Implementação da função	Tempo de execução (s)
GDScript	GDScript	0.2
GDScript	Python	5.6
Python	GDScript	342.0
Python	Python	374.0

Tabela 2.1: *Tempo total de um milhão de chamadas da função identidade.*

Explorando um pouco mais, foi possível constatar que a chamada de uma função implementada em GDScript por um *script* Python é muito ineficiente, e a ineficiência na quarta linha se dá pelo fato de que o intermédio entre um *script* Python e um *script* GDScript é feito por meio de funções do Godot.

Para reduzir as consequências desse problema, o número de *scripts* Python e a comunicação entre Python e GDScript foram reduzidos ao mínimo possível para que não haja quedas bruscas de FPS durante o jogo.

[10]Repositório do Pythonscript: https://github.com/touilleMan/godot-python (último acesso em 10/10/2019)

Capítulo 3

Fundamentos Matemáticos

Este capítulo mostra como funciona o algoritmo *Q-Learning* e como é possível utilizar heurísticas para melhorar seus resultados. Mostra também como as redes neurais artificiais foram desenvolvidas, como elas aprendem e como elas foram usadas para melhorar os resultados do algoritmo *Q-Learning*. Finalmente, mostra o algoritmo CAP, utilizado pelo jogo para se adaptar às habilidades do jogador.

3.1 Aprendizagem por reforço

Nesta seção apresentamos um algoritmo de aprendizado por reforço, o algoritmo *Q-Learning* e alguns dos parâmetros importantes que influenciam no seu funcionamento.

3.1.1 *Q*-Learning

Para que os agentes autônomos desse projeto conseguissem aprender uma política quase ótima foi utilizado o algoritmo *Q-Learning*. Esse algoritmo foi desenvolvido por Watkins, 1989, baseado na teoria de programação dinâmica de Bellman, 1952, para achar a solução ótima do processo de decisão de Markov que emerge ao analisarmos um agente autônomo.

Suponha um agente autônomo que vive em um mundo de espaço discreto e finito e de tempo discreto. Em um dado instante de tempo $t \in \mathbb{N}$ o mundo e o agente podem ser representados por um estado s_t do conjunto de todos os estados possíveis S. O agente possui um conjunto $\mathcal{A}$ de ações disponíveis e uma função $A : S \longrightarrow \mathcal{P}(\mathcal{A})$ que associa

cada estado do mundo à um conjunto de ações legais nele, onde $\mathcal{P}(\mathcal{A})$ é o conjunto das partes de $\mathcal{A}$.

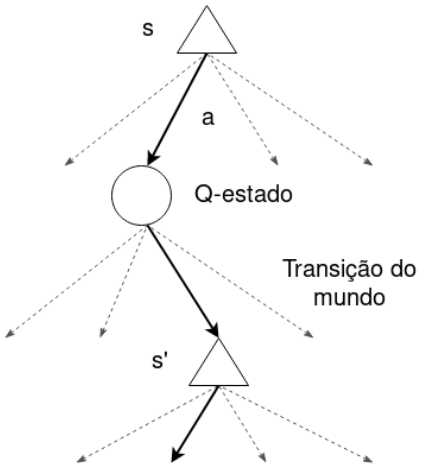

Figura 3.1: *Árvore de busca do agente. Começando em um estado s o agente executa uma de suas ações disponíveis a, criando um estado intermediário virtual chamado de Q-estado. A partir do Q-estado escolhido o mundo transiciona para um estado s'.*

Em cada instante de tempo o agente executa uma ação $a_t \in A(s_t)$, após executar tal ação o estado do mundo é transicionado para um estado s_{t+1}, segundo uma distribuição de probabilidade desconhecida que representa fatores externos que alteram o mundo, e o agente recebe uma recompensa $R(s_t, a_t, s_{t+1})$ de acordo com uma função de recompensa $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \longrightarrow \mathbb{R}$. Tal lógica pode ser representada pela árvore de busca ilustrada na Figura 3.1.

Como o agente é autônomo, ele depende de uma política $\pi : \mathcal{S} \longrightarrow \mathcal{A}$ para decidir qual ação tomar a cada instante de tempo. O objetivo do agente é aprender uma política ótima π^* que maximiza a recompensa futura esperada, para isso Watkins, 1989 introduz o algoritmo *Q-learning*.

O algoritmo se baseia na criação de uma função recursiva $Q : \mathcal{S} \times \mathcal{A} \longrightarrow \mathbb{R}$ de forma que $Q^\pi(s, a)$ seja a recompensa descontada esperada se o agente executar a ação a no estado s e seguir a política π a partir do próximo estado. O valor $Q^\pi(s, a)$ é chamado de Q-valor da ação a no estado s seguindo a política π. É possível então maximizar a recompensa recebida em cada transição para criar uma função ótima Q^* (Equação 3.1) e utilizar tal função para criar uma política ótima π^* que executa as ações com maior valor de Q^* (Equação 3.2).

$$Q^*(s, a) = E[R(s, a, S') + \gamma \max_{a' \in A(S')} Q^*(S', a')|(s, a)] \tag{3.1}$$

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}}\, \mathcal{Q}^*(s, a) \tag{3.2}$$

O fator de desconto $\gamma \in (0, 1)$ reflete a importância das recompensas futuras para o estado atual do agente, e geralmente é próximo de 1.

Se desenvolvermos a recorrência, é possível notar que as recompensas recebidas nos j-ésimo e k-ésimo passos, $j < k$, são multiplicadas por γ^{j-1} e γ^{k-1} respectivamente, ou seja a recompensa recebida no passo k tem um peso menor no cálculo da função $\mathcal{Q}$ que a recebida no passo j.

Como o número de estados futuros é indeterminado, sem o fator de desconto a função $\mathcal{Q}$ pode ser resumida como um somatório infinito das recompensas futuras, que podem ser maiores que zero, fazendo com que a função divirja. Ao introduzir o fator de desconto o somatório sempre será menor que $\frac{R_{max}}{1-\gamma}$, onde R_{max} é a maior recompensa possível.

Como o agente não sabe quais são as transições do mundo e quais são as suas probabilidades não é possível calcular a esperança condicional analiticamente, portanto a função tem que ser aproximada através de experimentações.

Convém agora introduzir o conceito de *episódio*. Um episódio $e = (s, a, s', r)$ é uma tupla de valores observados pelo agente em determinada transição. Um episódio contém um estado inicial s, uma ação a executada pelo agente, um estado s', observado depois do agente executar a ação a no estado s, e a recompensa recebida r. A *experiência* E do agente é um conjunto de episódios observados por ele.

Suponha que a função $\mathcal{Q}$ é representada por uma matriz na qual as linhas são estados, as colunas são ações e as entradas são os valores da função. O agente é inicializado com uma função $\mathcal{Q}_0$ com valores aleatórios, e a cada instante de tempo ele observa um episódio. A cada novo episódio $e_t = (s_t, a_t, s'_t, r_t)$, observado no instante t, a função do instante de tempo anterior $\mathcal{Q}_{t-1}$ pode ser atualizada conforme a seguinte equação

$$\mathcal{Q}_t(s, a) = \begin{cases} (1 - \alpha)\mathcal{Q}_{t-1}(s, a) + \alpha(r_t + \gamma \max_{a' \in A(s_t)} \mathcal{Q}_{t-1}(s'_t, a')) & \text{se } s = s_t \text{ e } a = a_t \\ \mathcal{Q}_{t-1}(s, a) & \text{c.c.} \end{cases} \tag{3.3}$$

na qual $\alpha \in [0, 1)$ é a taxa de aprendizado, que geralmente assume um valor próximo de zero.

O algoritmo consiste em iterar sobre o mundo várias vezes e atualizar a função $\mathcal{Q}$ a

cada novo episódio até que ela convirja. Não necessariamente a função resultante será ótima, porém há heurísticas, como a introdução de uma taxa de exploração (Subseção 3.1.2) e a técnica de *replay* de episódios (Subseção 3.1.3), que fazem com que a função resultante se aproxime mais do ótimo.

Apesar do mundo conter um número finito de estados, no caso do jogo deste projeto o número de estados é grande o suficiente para que a criação de uma matriz para representar a função Q se torne inviável. A Subseção 3.2.4 descreve como é possível utilizar uma rede neural para aproximar a função, economizando muito espaço a custo de aproximações menos precisas.

Watkins e Dayan, 1992 fornecem uma introdução muito mais detalhada no assunto, além de provar a convergência do método.

3.1.2 Taxa de exploração

Como a atualização da função Q depende das recompensas recebidas pelo agente, as recompensas recebidas dependem das ações que ele executa e as ações dependem da função Q, percebe-se que a atualização da função Q é enviesado pelo estado anterior dela, ou seja se a função inicial Q_0 não escolher ações de forma que o agente receba recompensas positivas, ela dificilmente será ótima.

Uma heurística para fazer com que o agente explore mais estados é a introdução de uma taxa de exploração $\epsilon \in (0, 1)$. A cada tomada de decisão, o agente escolhe uma ação válida aleatória com chance ϵ ou escolhe a ação com base nos Q-valores com chance $1 - \epsilon$, assim o algoritmo introduz episódios que não dependem da função Q, facilitando assim a criação de uma função que receba recompensas positivas.

Nesse projeto a taxa de exploração decai constantemente ao longo do tempo, começando em um valor máximo ϵ_M até um valor mínimo ϵ_m, desse modo o agente explora mais quando sua experiência é pequena e utiliza mais sua experiência quando ela é grande.

3.1.3 *Replay* de episódios

É possível que o agente receba recompensas positivas escassas e, se essas recompensas não conseguirem compensar as recompensas negativas o algoritmo acaba não conseguindo aprender uma função Q útil.

Para que o agente consiga revisitar episódios antigos e aprender mais com recompensas antigas este trabalho utiliza a técnica chamada de *replay* de episódios (Mnih *et al.*, 2015).

Nesse método os episódios são armazenados e, a cada episódio novo e_n, a função Q_{n-1} é atualizada usando um episódio passado e_i, $0 \leq i \leq n$, escolhido seguindo uma distribuição de probabilidade X_n. Nesse projeto a distribuição X_n é a distribuição uniforme discreta $U(0, n)$.

Outra propriedade dessa heurística é que ela permite que as recompensas sejam assíncronas, já que a atualização de Q_{n-1} não depende diretamente do episódio e_n.

3.2 Redes neurais artificiais

Nesta seção revisitamos conceitos básicos de redes neurais para o seu uso neste trabalho.

3.2.1 Introdução

Com a criação dos computadores no século XX várias pessoas começaram a idealizar usos para essa máquina revolucionária. Em específico McCulloch e Pitts, 1943, criaram um sistema baseado no cérebro humano para classificar dados. A unidade lógica básica desse sistema foi chamada de neurônio, que é composta por um conjunto de entradas e um conjunto de saídas, além de ser possível criar redes com essas unidades utilizando saídas de umas como entradas de outras. Anos depois Rosenblatt, 1961, descreve um modelo matemático para tais neurônios chamado perceptron, que consegue aprender a processar e classificar entradas sensoriais, que mais tarde é refinado por Minsky e Papert, 1969.

Sendo $\mathcal{S}$ o conjunto de todas as entradas sensoriais possíveis, Minsky e Papert descrevem um perceptron $P : \mathcal{S} \longrightarrow \mathbb{R}$ como sendo uma função que pode ser escrita usando uma função de processamento da entrada $\Phi : \mathcal{S} \longrightarrow \mathbb{R}^n$, um vetor de pesos $w \in \mathbb{R}^n$, uma constante $b \in \mathbb{R}$ e uma função de ativação $\theta : \mathbb{R} \longrightarrow \mathbb{R}$. É importante notar que a função Φ serve especialmente para processar os resultados dos sensores e extrair informações relevantes.

$$P(s) = \theta(w \cdot \Phi(s) + b) \tag{3.4}$$

O propósito do perceptron é dividir o conjunto $\mathcal{S}$ em dois subconjuntos disjuntos $F^+ = \{s \in \mathcal{S} : P(s) > 0\}$ e $F^- = \{s \in \mathcal{S} : P(s) < 0\}$, esse processo é chamado de classificação.

Uma propriedade importante do perceptron é que ele consegue modificar seus pesos para adaptar-se à dados pré-classificados, processo chamado de aprendizado. Contudo perceptrons clássicos só conseguem aprender a classificar dados linearmente separáveis, o que é uma desvantagem muito grande já que é impossível aprender a função Booleana XOR por exemplo. Uma das alternativas seria criar redes de perceptrons, assim como sugerido por Minsky e Papert, porém eles não conseguiram criar algoritmos que possibilitassem o aprendizado de tais redes. Outra alternativa é usar um algoritmo que admite erros, como o algoritmo *Pocket Perceptron* (Abu-Mostafa *et al.*, 2012).

Pouco mais de uma década depois, Rumelhart *et al.*, 1986, utilizaram o método do gradiente descendente para desenvolver o algoritmo de retropropagação de erro que possibilita o aprendizado de redes de perceptrons, também chamadas de redes neurais artificiais.

Uma rede neural é composta de uma camada de entrada, m camadas internas e uma camada de saída. A camada de entrada é composta da função de processamento de entrada Φ e as camadas internas e a de saída são compostas de conjuntos de perceptrons, sendo que a entrada de cada perceptron é o vetor de saída da camada anterior.

Podemos classificar as redes neurais em dois grupos: redes *feedforward* e redes recorrentes. Em redes *feedforward* a saída de uma camada não pode ser utilizada como entrada de uma camada anterior, enquanto em redes recorrentes isso é possível. Esta subseção trata somente de redes do tipo *feedforward*.

Para facilitar a representação faremos as seguintes simplificações:

- A camada de entrada será considerada como camada zero e a de saída como camada $m + 1$;
- Um conjunto de perceptrons será representado por uma matriz, onde as linhas são os pesos de cada um, e por uma única função de ativação comum;
- A aplicação de uma função qualquer $f : X \longrightarrow X$, definida em um conjunto qualquer X, em um vetor $x = (x_1, x_2, ..., x_n) \in X^n$ será definida por $f(x) = (f(x_1), f(x_2), ..., f(x_n))$;
- Vetores em $\mathbb{R}^n$ serão equivalentes à matrizes em $\mathcal{M}_{n\times 1}$, tornando possível a utilização de operações matriciais em vetores e operações de vetores em matrizes.

Sejam então $s \in \mathcal{S}$ a entrada sensorial, p_i o número de perceptrons na i-ésima camada, $W_i \in \mathcal{M}_{p_i \times p_{i-1}}$ a matriz de pesos da i-ésima camada, $b_i \in \mathbb{R}^{p_i}$ o vetor de constantes da i-ésima camada e $\theta_i : \mathbb{R} \longrightarrow \mathbb{R}$ a função de ativação da i-ésima camada, a camada i pode ser expressa pelo valor a_i abaixo

$$a_0 = \Phi(s) \tag{3.5}$$

$$a_i = \theta_i(W_i a_{i-1} + b_i),\ i = 1, ..., m+1 \tag{3.6}$$

Nas camadas internas é comum utilizar funções de ativação contínuas, de derivadas contínuas e de imagem compacta. A continuidade da função e da derivada garantem que o algoritmo do gradiente descendente tenha bons resultados e a imagem compacta ajuda a combater possíveis *overflows* nos valores de saída das camadas internas. As funções mais populares em redes desse tipo são a função sigmóide $\sigma : \mathbb{R} \longrightarrow [0, 1]$ e a função tangente hiperbólica $\tanh : \mathbb{R} \longrightarrow [-1, 1]$.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.7}$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.8}$$

Redes neurais foram inicialmente utilizadas para aproximar funções classificadoras, mas a partir da década de 90, com a publicação de artigos como o de Funahashi, 1989, que provaram alguns teoremas importantes sobre quais funções podem ser representadas por redes neurais artificiais, esse método começou a ser utilizado também para aproximar funções genéricas. Em especial foi provado por Funahashi que redes com uma única camada interna finita podem aproximar qualquer função contínua de domínio compacto, dado que a camada interna tenha o número suficiente de perceptrons.

3.2.2 Aprendizado

Dado um conjunto de dados pré-anotados $X \subset S \times \mathbb{R}^N$ e uma rede com saída $p_{m+1} = N$, é possível utilizar o método do gradiente descendente para fazer com que a rede aprenda a anotar dados seguindo o modelo desse conjunto. Para isso escolhe-se um elemento aleatório $(x, y) \in X$, calcula-se a saída da rede a_{m+1} a partir de x e calcula-se a distância E entre o resultado da rede e o resultado esperado y, também conhecida como erro, utilizando a seguinte função

$$E = \frac{1}{2}||a_{m+1} - y||^2 \tag{3.9}$$

Como queremos modificar os pesos de forma a minimizar o erro em x, calcula-se as derivadas parciais de cada camada em relação à eles.

$$\delta_i = \left(\frac{\partial E}{\partial a_i}\right)^T \odot \theta_i'(W_i a_{i-1} + b_i) \tag{3.10}$$

$$\frac{\partial E}{\partial a_i} = \begin{cases} (a_{m+1} - y)^T & \text{se } i = m+1 \\ (\delta_{i+1})^T W_{i+1} & \text{c.c.} \end{cases}$$

$$\frac{\partial E}{\partial W_i} = a_{i-1}(\delta_i)^T \tag{3.11}$$

$$\frac{\partial E}{\partial b_i} = (\delta_i)^T \tag{3.12}$$

nas quais θ_i' é a derivada da função de ativação da i-ésima camada e $\odot$ é o produto entrada a entrada da matriz, também chamado de produto de Hadamard. A partir das derivadas, atualiza-se os pesos seguindo as fórmulas abaixo

$$\tilde{W}_i = W_i - \alpha \left(\frac{\partial E}{\partial W_i}\right)^T \quad , i = 1, ..., m+1 \tag{3.13}$$

$$\tilde{b}_i = b_i - \alpha \left(\frac{\partial E}{\partial b_i}\right)^T \quad , i = 1, ..., m+1 \tag{3.14}$$

nas quais $\alpha \in [0, 1]$ é a taxa de aprendizado e $\tilde{W}_i$ são os novos pesos e $\tilde{b}_i$ as novas constantes depois de uma iteração do algoritmo. Depois de atualizá-los, o processo é repetido até a soma dos erros de todos os elementos do conjunto X estabilizar.

Neste projeto foi utilizada a heurística de estimação de momento adaptativa, também chamada de Adam, proposta por Kingma e Ba, 2017. Tal heurística faz o uso dos momentos estimados do gradiente para escolher uma taxa de aprendizado diferente para cada iteração, de forma a melhorar consideravelmente a performance do algoritmo ao estimar funções ruidosas.

3.2.3 Redes recorrentes e memória

Como introduzido anteriormente, redes recorrentes são redes neurais nas quais perceptrons de uma camada podem se conectar com perceptrons de camadas anteriores, formando um loop. Redes desse tipo mantém um estado interno h, recebem uma sequência de entradas $(s_t)_{t=1}^{T}$ e produzem uma sequência de saídas $(o_t)_{t=1}^{T}$, nas quais $s_t \in S$ e $o_t \in \mathbb{R}^n$ para todo t, $n \in \mathbb{N}$ é o tamanho do vetor de saída e $T \in \mathbb{N}$ é o número de recorrências. Ao processar a entrada s_1 a rede produz um estado interno h_1 e uma saída o_1, ao processar a entrada s_2 a rede usa também o estado interno h_1 para produzir a saída o_2 e um segundo estado interno h_2, e assim por diante até a última camada da recorrência.

O estado interno torna possível utilizar informação do processamento de entradas passadas nas entradas futuras, fazendo com que a rede tenha um senso de "progressão temporal", útil no processamento de texto ou identificação de padrões em séries temporais por exemplo.

É comum que a rede tenha não seja composta somente das camadas recorrentes, mas também de algumas camadas *feedforward* que processam as entradas que as camadas recorrentes irão receber e algumas que processam as saídas das camadas recorrentes. Para que elas não sejam confundidas com camadas *feedforward* de agora em diante as camadas recorrentes que compartilham o mesmo estado interno serão referidas como pertencentes à um mesmo módulo recorrente.

Há várias arquiteturas de módulos recorrentes, em especial nesse trabalho foi utilizado o chamado LSTM[1], introduzido por Hochreiter e Schmidhuber, 1997 e modificado por Gers *et al.*, 2000. O módulo é constituído de quatro portas, uma célula de memória e um estado interno, sendo que cada porta é representada por duas matrizes de peso, duas constantes e uma função de ativação, a célula aplica operações nas saídas das portas e o estado interno é representado por um vetor.

Nesse projeto foi utilizada a biblioteca de aprendizado de máquina PyTorch, que implementa o LSTM da seguinte forma

[1] Sigla em inglês para *Long short-term memory*.

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \tag{3.15}$$

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \tag{3.16}$$

$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \tag{3.17}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{3.18}$$

$$h_t = o_t \odot \tanh(c_t) \tag{3.19}$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \tag{3.20}$$

nas quais h_t é a matriz de memória, h_0 é a matriz zero, x_t é a t-ésima entrada do módulo, i_t é a t-ésima saída da porta de entrada, f_t é a t-ésima saída da porta de esquecimento, g_t é a t-ésima saída da porta da célula, c_t é a t-ésima saída da célula de memória, o_t é a t-ésima saída do módulo, para $0 < t \leq T$, W_{xy} são os pesos das camadas e b_{xy} são as constantes das camadas.

O aprendizado do módulo pode ser feito usando o método do gradiente descendente, porém como o gradiente depende do gradiente da memória, para utilizar o gradiente descendente é necessário "desenrolar" a rede copiando os módulos recorrentes T vezes, calcular o gradiente de cada uma das cópias e atualizar os pesos do módulo utilizando a soma dos gradientes das cópias no módulo (Hochreiter e Schmidhuber, 1997).

3.2.4 Relação entre redes neurais artificiais e *Q-learning*

Dado um estado $s \in \mathcal{S}$ do mundo, redes neurais podem ser usadas de três jeitos diferentes para obter a melhor ação a ser executada no algoritmo de *Q-Learning*.

No primeiro a rede é utilizada para aproximar uma função $f : \mathcal{S} \longrightarrow \mathbb{R}$ que atribui um valor para um estado. Utilizando a função f em conjunto à uma função $g : \mathcal{S} \times \mathcal{A} \longrightarrow \mathcal{S}$, criada pelo desenvolvedor, que devolve um aproximação $\hat{s'} \in \mathcal{S}$ do próximo estado a partir de um estado inicial s e uma ação $a \in \mathcal{A}$, é possível criar uma função $Q(s, a) = f(g(s, a))$, então é possível calcular um erro seguindo a Equação 3.21 e otimizá-la utilizando o método Adam.

$$E = \frac{1}{2}\|R(s, a, s') + \gamma \max_{a' \in A(s')} Q(s', a') - Q(s, a)\|^2 \tag{3.21}$$

Essa abordagem é vantajosa pois permite que estados futuros sejam aproximados pela função g, fazendo com que eles possam ser usados a qualquer momento para o aprendizado,

porém a desvantagem é que a função g pode não ser uma aproximação muito boa para eventos probabilísticos, fazendo com que o estado aproximado $\hat{s'}$ não seja muito próximo da realidade e a qualidade do resultado final do aprendizado caia. Um exemplo de utilização dessa abordagem são os exercícios de aprendizagem por reforço do curso CS188x da UC Berkeley[2], nos quais a técnica é utilizada na criação de um agente que joga o jogo *Pacman*.

O segundo método não necessita de uma função intermediária para estimar o próximo estado. Nesse caso a rede é uma função $f : \mathcal{S} \longrightarrow \mathbb{R}^{|\mathcal{A}|}$ que dado um estado s devolve um vetor de Q-valores $q = (Q(s, a_1), Q(s, a_2), ..., Q(s, a_{|\mathcal{A}|}))$, $a_i \in \mathcal{A}$. Essa rede também pode ser otimizada utilizando o método Adam, com a mesma função de erro da anterior.

Uma das desvantagens desse método é a falta de informação sobre as consequências das ações; por exemplo, se uma das ações for muito complexa, a rede pode demorar muito para aprender quais estados devem ser rotulados com essa ação, consequentemente ela não é muito útil para aplicações em tempo real como esse projeto, porém a qualidade do aprendizado é superior ao método anterior, já que a rede aprende quais são as consequências de suas ações, essas consequências não são fixadas pelo desenvolvedor. Essa abordagem é utilizada por Mnih *et al.*, 2015 para criar um agente que joga jogos de Atari.

No terceiro método é utilizado uma rede neural recorrente ao invés de uma rede *feedforward*. Nesse método a melhor ação é aproximada a partir dos estados anteriores, fazendo com que o agente tenha uma noção temporal de suas ações e consiga desenvolver estratégias mais complexas. Como o método não limita o que a rede pode aproximar, essa estratégia pode ser usada para complementar tanto o primeiro quanto o segundo método citados anteriormente. Ela é otimizada do mesmo modo que as anteriores, porém como ela possui um módulo recorrente é necessário retropropagar o erro por todas as camadas de recorrência e atualizá-la com a média dos erros, como descrito na Subseção 3.2.3.

A desvantagem desse método é que ele aumenta o tempo de aprendizado proporcionalmente à profundidade da recorrência, ou seja quanto mais estados anteriores a rede considerar, mais o aprendizado demorará. Esse fato em conjunto com o segundo método faz com que o método se torne mais inviável ainda para ser utilizado em um jogo, porém em conjunto com o primeiro método é possível que isso não seja um empecilho. Essa abordagem foi utilizada por Lample e Chaplot, 2017 para criar um agente que joga o jogo Doom.

[2]Material disponível em http://ai.berkeley.edu/project_overview.html (Último acesso em 9/11/2019)

3.3 Sistema de adaptação de dificuldade

3.3.1 Por que adaptar a dificuldade?

Primeiro é necessário entender que existem vários tipos de jogadores, como aponta o famoso artigo de Bartle, 1996. A existência de diferentes tipos de jogadores implica que nem todos eles progridem do mesmo modo no jogo e que cada um deles se sente feliz com uma certa progressão, logo um método para adaptar a dificuldade às habilidades dos jogadores é essencial para que o jogo consiga agradar o maior número de pessoas.

Atualmente a maioria dos jogos ou fornece níveis de dificuldade pré-programados, assim o jogador consegue comunicar ao jogo qual a sua proficiência, ou não possui sistema de adaptação, o que o torna difícil para os jogadores que não estão acostumados com o estilo do jogo.

Csikszentmihalyi, 1990, faz uma análise qualitativa de como a dificuldade de uma tarefa pode impactar no estado mental da pessoa que a está executando. Segundo ele o usuário pode se sentir entediado se a tarefa for fácil demais ou frustrado se for difícil demais, porém ao equilibrar a dificuldade com a habilidade do usuário ele tem a possibilidade de entrar em um estado mental chamado *flow*.

No *flow* a pessoa tem um senso grande de controle sobre todas as suas ações, tem o senso de percepção do ambiente elevado e sente que o tempo passa mais rápido enquanto executa a tarefa, fazendo com que ela se sinta relaxada e satisfeita ao mesmo tempo após executá-la. Conclui-se então que ao balancear a dificuldade do jogo a chance do jogador entrar nesse estado cresce, aumentando assim sua satisfação.

3.3.2 Cálculo da dificuldade

Nesse projeto os inimigos autônomos que o jogador enfrenta são escolhidos segundo um sistema de ordenação parcial. Esse sistema é baseado no algoritmo CAP[3], introduzido no artigo de Nyamsuren *et al.*, 2018, que foi criado seguindo as equações de Maris e Van der Maas, 2012.

O CAP foi baseado no sistema de Elo do xadrez (Elo, 1978), mas diferente do sistema de Elo, que busca comparar a habilidade de dois jogadores em um jogo "jogador contra jogador", o CAP busca comparar a habilidade do jogador com a dificuldade de uma tarefa em um jogo "jogador contra jogo".

[3] sigla em inglês para *Computerized Adaptative Practice*

Nesse sistema a habilidade do m-ésimo jogador é representada por um número $\theta_m \in \mathbb{R}$ e a dificuldade do i-ésimo problema é representada por um número $\beta_i \in \mathbb{R}$, de grandezas iguais, ou seja se $\theta_m = \beta_i$ o jogador m tem 50% de chance de solucionar o problema i. A partir desses valores é possível ordenar as dificuldades dos problemas, criando assim uma ordenação parcial deles.

Ao tentar resolver o problema i são registrados dois valores sobre a performance do jogador m: $x_{im} \in \{0, 1\}$, que é 1 se o jogador solucionou o problema ou 0 caso contrário, e $t_{im} \in \mathbb{R}_+$, que é o tempo que o jogador demorou para resolver o problema. A partir desses valores uma pontuação S_{im} é calculada seguindo a fórmula

$$S_{im} = \begin{cases} (2x_{im} - 1)\left(1 - \dfrac{t_{im}}{d_i}\right) & \text{, se } t_{im} \leq d_i \\ 0 & \text{, c.c.} \end{cases} \tag{3.22}$$

onde $d_i \in \mathbb{R}_+$ é o tempo máximo que o jogador tem para resolver o problema i.

Essa fórmula bonifica o jogador que sabe solucionar o problema e demora pouco para resolver e o que não sabe solucionar e demora muito para resolver, além de penalizar os que sabem solucionar e demoram para resolver e os jogadores que não sabem solucionar e resolvem rápido.

Maris e Van der Maas, 2012 tratam as variáveis como variáveis aleatórias e concluem que a pontuação esperada para um jogador de habilidade θ_m em um problema β_i é dada pela seguinte fórmula

$$E[S_{im}|\theta_m - \beta_i] = \frac{e^{2(\theta_m - \beta_i)} + 1}{e^{2(\theta_m - \beta_i)} - 1} - \frac{1}{\theta_m - \beta_i} \tag{3.23}$$

Dado que sabemos a pontuação esperada do par jogador-problema podemos atualizar a habilidade do jogador e a dificuldade do problema para que a pontuação esperada seja mais próxima da pontuação observada. Fazemos isso seguindo as fórmulas

$$\tilde{\theta}_m = \theta_m + K_m(S_{im} - E[S_{im}|\theta_m - \beta_i]) \tag{3.24}$$

$$\tilde{\beta}_i = \beta_i + H_i(E[S_{im}|\theta_m - \beta_i] - S_{im}) \tag{3.25}$$

Onde $\tilde{\theta}_m$ é a nova habilidade do jogador m, $\tilde{\beta}_i$ é a nova dificuldade do problema i e K_m e H_i são constantes.

A Subseção 5.6 fala sobre como o sistema é utilizado no jogo.

3.3.3 Inicializando o sistema

Um dos problemas do CAP é que inicialmente não sabemos as dificuldades dos problemas e as habilidades dos jogadores, fazendo com que no começo do jogo a chance de apresentarmos um problema de dificuldade elevada para um jogador de habilidade pequena seja grande.

Por isso o ideal é tentar criar uma função de pontuação inicial que explora as dificuldades intrínsecas dos problemas, para assim criar uma ordenação parcial inicial dos problemas. Ela não precisa ser perfeita (caso contrário não precisaríamos do CAP) ela só precisa fazer com que problemas conhecidamente difíceis não sejam apresentados para jogadores iniciantes.

Por exemplo Van der Maas e Nyamsuren, 2017 apresentam um jogo para ensinar matemática à crianças, no qual os problemas podem incluir operadores de soma, subtração, multiplicação e divisão, e podem ter um ou dois operadores. Nesse caso os problemas de soma e subtração deveriam ter dificuldades iniciais menores que os problemas de multiplicação e divisão, assim como os problemas com um operador deveriam ter dificuldades iniciais menores que os problemas com dois operadores.

3.3.4 Escolhendo o problema ideal

Segundo Maris e Van der Maas, 2012, a probabilidade de um jogador com habilidade θ_m resolver um problema de dificuldade β_i é dada pela fórmula

$$P(X = 1|\theta_m - \beta_i) = \frac{e^{\theta_m - \beta_i}}{1 + e^{\theta_m - \beta_i}} \tag{3.26}$$

Onde X é a variável aleatória que representa a corretude da resolução do jogador ao problema.

Dado uma probabilidade p e a habilidade do jogador podemos calcular a dificuldade ideal δ para que o jogador consiga resolver o problema com probabilidade p

$$\delta = \theta_m + \ln\left(\frac{1-p}{p}\right) \tag{3.27}$$

A partir da dificuldade ideal δ podemos escolher um problema j tal que $|\delta - \beta_j|$ seja o menor possível dentre todos os problemas, desse modo a dificuldade do problema fica o mais perto possível da idealizada.

Segundo Eggen e Verschoor, 2006, o ideal para que o jogador se sinta motivado é que a probabilidade de resolução dos problemas seja em média 75%, mas não maior que isso. No artigo de Nyamsuren *et al.*, 2018 o valor de p é escolhido segundo uma distribuição normal truncada $\mathcal{N}_{-}^{+}(0.75, 0.5, 1.0, 0.1)$, que possui média 0.75, porém isso faz com que o jogador possa receber problemas infinitamente mais fáceis que sua habilidade (se $p \approx 1$), o que não é uma característica desejada nesse projeto, portanto essa distribuição foi modificada para $\mathcal{N}_{-}^{+}(0.725, 0.5, 0.95, 0.07)$.

Capítulo 4

O Jogo

Este capítulo contém a história de fundo do jogo e introduz os conceitos, as mecânicas e a jogabilidade do jogo final.

4.1 História

Em um mundo de fantasia existem pessoas que arriscam suas vidas explorando lugares desconhecidos, como florestas, cavernas e ruínas, tais pessoas são chamadas de aventureiros. Durante suas aventuras eles encontram os mais temíveis monstros e as mais ardilosas armadilhas, e também encontram os mais valiosos tesouros, esquecidos no tempo por civilizações antigas. Alguns aventureiros são famosos por toda a nação por serem corajosos e destemidos. O protagonista é um jovem que sonha em se tornar aventureiro, porém não tem nem habilidade nem dinheiro suficiente para isso.

4.2 Jogabilidade

4.2.1 Objetivo

O objetivo do jogador é ajudar o protagonista a realizar seu sonho; para isso ele irá comandá-lo nessa dura jornada. Tudo começa na cidade natal do protagonista, onde moram um ferreiro e uma alquimista muito habilidosos e um comerciante famoso que irão apoiá-lo em suas aventuras vendendo armas e suprimentos e comprando itens valiosos. O jogador pode sair da cidade em uma aventura a qualquer momento, retornando à ela ao final da aventura.

O jogador utiliza as teclas direcionais do teclado para controlar o personagem, a barra de espaço para atacar e a tecla E para interagir com pessoas e objetos.

Figura 4.1: *Mapa da cidade. É possível ver a alquimista Merlara em frente à casa do canto esquerdo de cima, o ferreiro Baldric em frente à casa do canto esquerdo de baixo, o comerciante Satoshi no canto direito de cima e o protagonista no centro. É possível ver também uma saída ao leste da cidade, pela qual o jogador pode entrar em uma aventura.*

4.2.2 Aventuras

As aventuras são constituídas de uma rede de salas gerada aleatoriamente e o conteúdo delas é gerado proceduralmente de acordo com a maestria do jogador e do personagem.

Uma sala é composta de um espaço, geralmente do tamanho da janela do jogo, que contém um número pré-determinado de tesouros e monstros. O jogador pode se mover de uma sala para outra por meio das saídas da sala.

4.2.3 Combate

O personagem principal e seus inimigos têm pontos de vida, de ataque e de defesa. Quando os pontos de vida chegam a zero o personagem morre.

Cada personagem tem a sua maneira de atacar, e se seu ataque atingir um oponente retira uma quantidade de pontos de vida dele de acordo com os pontos de ataque do atacante e os pontos de defesa do oponente. Quanto mais pontos de ataque o atacante tem, mais pontos de vida ele retira do oponente e quanto mais pontos de defesa o oponente tem, menos vida o atacante retira dele, seguindo a fórmula

Figura 4.2: *Jogador lutando contra um goblin em uma sala da aventura. Note que a sala tem saídas para cima, para a esquerda e para baixo, que levam para outras salas ou para a cidade.*

$$\tilde{v}_d = \begin{cases} 0 & \text{, se } v_d + d_d \leq a_a \\ v_d & \text{, se } d_d \geq a_a \\ v_d + d_d - a_a & \text{, c.c.} \end{cases} \tag{4.1}$$

onde v_d é a vida do defensor, $\tilde{v}_d$ é a vida do defensor após o ataque, a_a é o ataque do atacante e d_d é a defesa do defensor.

4.2.4 Lojas e itens

O jogador possui uma mochila na qual é possível armazenar itens; ela pode ser acessada apertando a tecla I. Atualmente existem dois tipos de itens: armas, que podem ser equipadas no jogador para aumentar sua pontuação de ataque, e poções, que podem recuperar pontos de vida, porém as possibilidades de itens diferentes são infinitas, como armaduras, comida, chaves, pedras encantadas, mapas, ferramentas, etc.

O jogador consegue também adicionar itens na chamada “barra de uso rápido”, assim é possível usar itens sem tem que acessar a interface da mochila utilizando as teclas de 1 à 4.

Através dos comerciantes da cidade é possível comprar espadas e poções de vida e também é possível vender os itens adquiridos ao longo das aventuras.

Figura 4.3: *Barra de uso rápido destacada em amarelo abaixo da imagem e botão da mochila destacado em azul no canto direto de baixo.*

Figura 4.4: *Loja de poções no jogo.*

Capítulo 5

Implementação

Este capítulo mostra como foram implementados os sistemas do jogo através do motor Godot. Ao longo dele é possível identificar que os sistemas são divididos em quatro grupos de utilidade: armazenamento de dados, personagens, mundo e interface do usuário.

5.1 Persistência de dados

Jogos normalmente são feitos para entreter o jogador por vários dias, sendo assim é necessário que ele consiga persistir a progressão do usuário na memória do dispositivo, possibilitando que o jogador possa desligá-lo e voltar a jogar outra hora.

O Godot fornece funções para ler e escrever arquivos em um diretório específico reservado para a persistência do jogo, porém é o desenvolvedor quem deve especificar quais dados serão persistidos e como serão persistidos. Nesse projeto foram criados dois *scripts* para auxiliar e centralizar a escrita em disco.

O primeiro é o `SaveManager`, que é um *singleton* (Gamma *et al.*, 1994) global que centraliza toda a escrita em disco do jogo, gerenciando o arquivo JSON que contém o estado do jogo e fornecendo métodos para a escrita e leitura desse arquivo.

O segundo é o `ModelBase` (Figura 5.1) que é instanciado em todo objeto que precisa persistir alguma informação. Os *scripts* derivados dele são responsáveis por definir o modelo de dados específico de outros objetos, através da função estática `_get_model`, e por fornecer funções específicas de persistência. As funções desse *script* utilizam as do `SaveManager` para persistir os dados no disco. O modelo definido nesse arquivo segue o

formato JSON Schema[1].

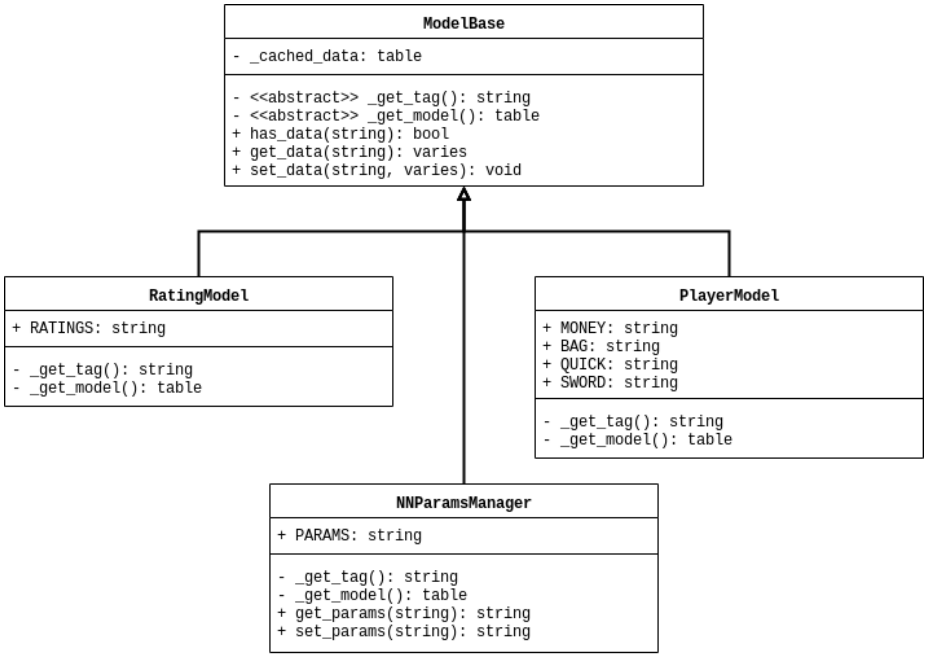

Figura 5.1: *Diagrama UML referente à classe* `ModelBase`. *A classe* `RatingModel` *é utilizada para armazenar as dificuldades dos encontros, a* `NNParamsMAnager` *para armazenar os pesos das IAs dos inimigos e a* `PlayerModel` *para armazenar as informações do jogador.*

5.2 Bancos de dados estáticos

O jogo pode conter inúmeros itens, cada um com suas utilidades e atributos, portanto é importante que o jogo possa adquirir de algum modo uma lista de itens com características parecidas, que ele possa consultar quais são as propriedades de um item e que seja fácil para os desenvolvedores criarem itens novos.

Com isso em mente foi criado uma cena chamada `ItemDatabase` que faz o papel de um banco de dados de itens. A cena contém um nó filho para cada item do jogo, cada um deles contendo um *script* com as atributos do item. Os valores associados a cada atributo são atribuídos através da própria interface gráfica do Godot por meio de variáveis exportadas.

Todo item tem quatro atributos básicos: nome, *sprite*, descrição e preço. O nome serve como identificador único do item no banco e como identificador textual para o jogador, a *sprite* é utilizada para representar o item visualmente para o jogador, a descrição contém uma frase que descreve brevemente a utilidade do item e o preço é o preço de compra do

[1]Site principal do JSON Schema: https://json-schema.org/ (Último acesso em 03/11/2019)

item dentro do jogo. O preço de venda do item é 70% do seu preço de compra, porcentagem definida arbitrariamente.

É possível agrupar os nós de itens com propriedades parecidas através dos grupos dos nós, sendo possível assim recuperar todos os itens que pertencem à um mesmo grupo através da função `get_nodes_in_group` do Godot. É possível também recuperar um item pelo nome através da função `get_node` do Godot.

Quando itens possuírem atributos e propósito em comum eles compartilham o mesmo *script*. Exemplos disso são as poções, que são utilizadas pelo jogador para recarregar vida, logo todas precisam de um valor que indica essa quantidade, as armas, que são utilizadas para combater inimigos, logo todas precisam de um valor que indica a quantidade extra de ataque que o jogador receberá ao utilizá-la.

Além do banco de dados de itens foi criado também um para inimigos chamado `EnemyDatabase`, facilitando assim a criação inimigos com atributos diferentes a partir da mesma cena de personagem e evitando a duplicação de arquivos. A lista dos atributos dos personagens pode ser encontrada na Seção 5.3.

5.3 Personagens

Os personagens são derivados de uma cena base `CharacterBase` (Figura 5.3). O *script* dessa cena implementa os métodos necessários para o combate e para a visualização do personagem. Todo *script* derivado dessa base deve implementar métodos para controlar as animações do personagem dada uma ação. Os personagens herdam do *script* base um atributo de velocidade `speed`, um de peso `weight`, um de vida máxima `max_life`, um de pontuação de ataque `damage` e um de pontuação de defesa `defense`.

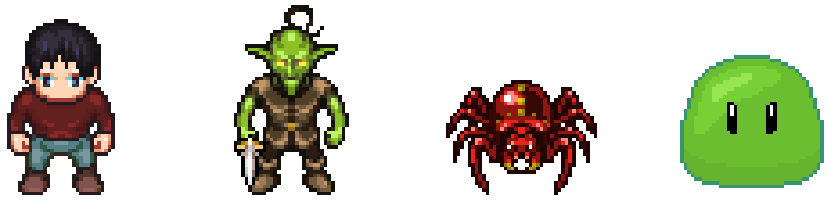

Figura 5.2: *Personagens implementados no jogo: Humano, Goblin, Aranha e Slime, da esquerda para a direita.*

Os personagens podem ser controlados pelo jogador ou por uma IA. Os que são controlados pelo jogador devem implementar *scripts* que mapeiam a entrada do jogador

Figura 5.3: *Diagrama UML referente à classe* `CharacterBase`. *A classe pai tem uma classe derivada para cada inimigo do jogo, cada uma implementa os métodos de acordo com as animações disponíveis para aquele inimigo. Note que a classe* `Player` *contém métodos extras para administrar os recursos do jogador.*

para ações do personagem. Já os que são controlados por uma IA devem ter um *script* derivado da classe `BaseRobotController` (Figura 5.4) que implementa uma interface básica com um nó de IA. Essa interface contém funções para indicar qual o estado atual do jogo percebido pelo personagem, quais as ações disponíveis para o personagem dado um estado e qual foi a recompensa adquirida após uma troca de estado.

Os nós de IA são constituídos de um *script* que implementa métodos para consultar qual ação o personagem deve tomar dado o estado atual e para atualizar o estado interno dada uma tupla de episódio. A interface foi criada com o principal objetivo de se utilizar aprendizagem por reforço, porém é possível utilizá-la para criar IAs pré-programadas também (utilizando árvores de comportamento por exemplo).

Todas as IAs disponíveis no momento utilizam *scripts* Python para que seja possível

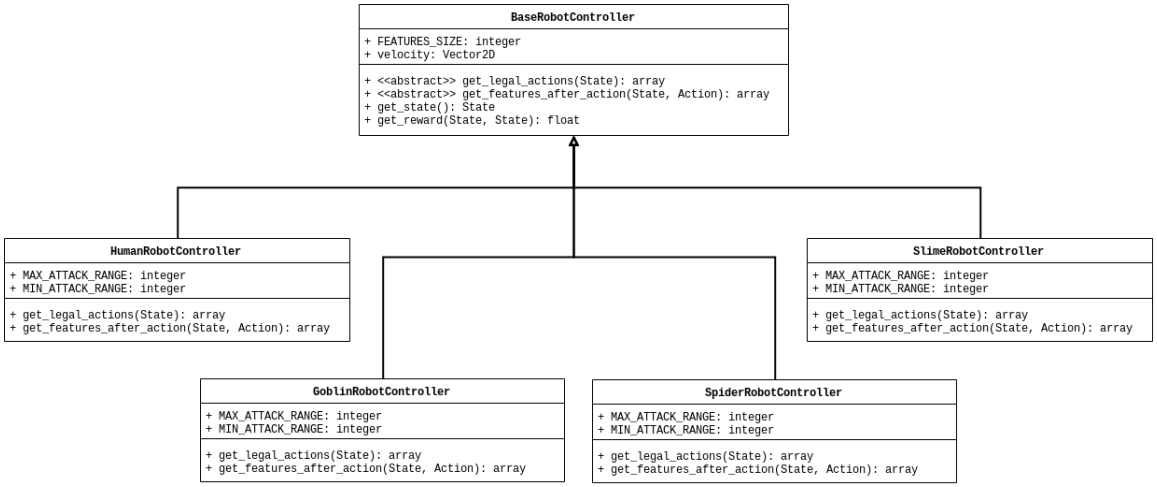

Figura 5.4: *Diagrama UML referente à classe* `BaseRobotController`. *A classe pai tem uma classe derivada para cada inimigo do jogo, cada uma implementa os métodos de acordo com as ações disponíveis para aquele inimigo.*

utilizar bibliotecas externas, mas é possível criar IAs escritas em GDScript sem que o código do controlador seja alterado.

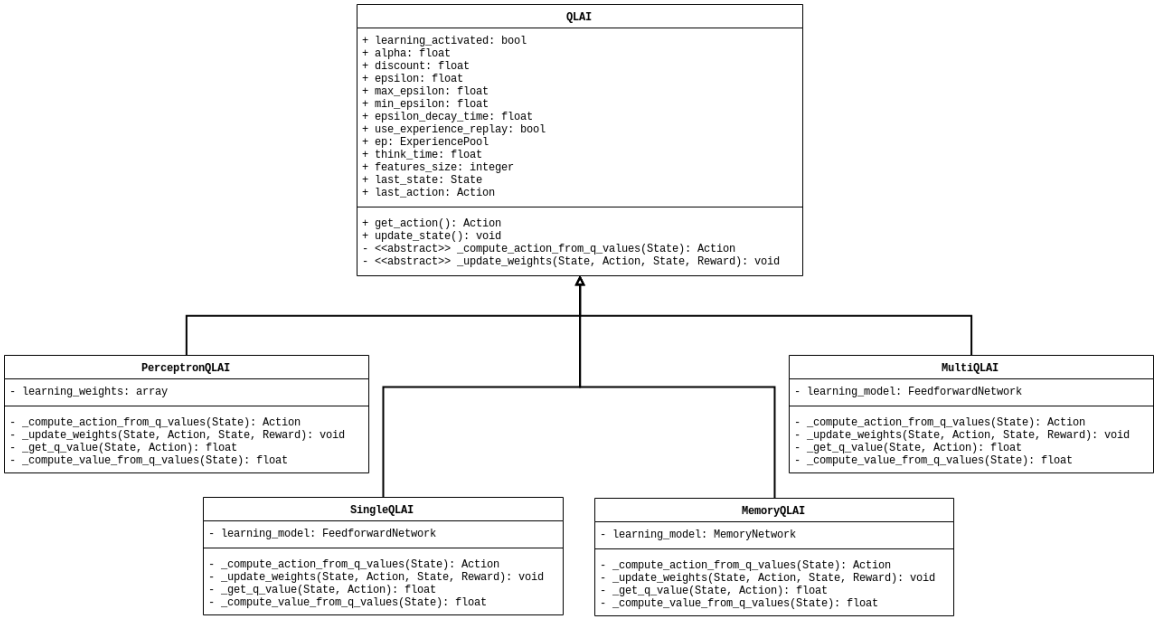

Figura 5.5: *Diagrama UML referente à classe* `QLAI`.

A classe base `QLAI` e suas derivadas recebem em sua inicialização um conjunto de valores para controlar o aprendizado. São eles:

- `learning_activated`: Valor Booleano que indica se o aprendizado está ativado ou não;
- `alpha`: A taxa de aprendizado;

- `discount`: A taxa de desconto de recompensa do $\mathcal{Q}$-*learning*;
- `max_exploration_rate`: A taxa de exploração máxima;
- `min_exploration_rate`: A taxa de exploração mínima;
- `exploration_rate_decay_time`: O tempo de decaimento da taxa de exploração;
- `experience_replay`: Valor Booleano indica se o *replay* de episódios está ativo ou não;
- `experience_pool_size`: O tamanho mínimo da memória de episódios antes de ser possível utilizar o *replay* de episódios;
- `think_time`: O tempo entre uma ação e outra.

Foram criados quatro tipos de IAs derivadas da classe `QLAI` (Figura 5.5) (as notações utilizadas são as mesmas da Subseção 3.2.4):

- `PerceptronQLAI`: Utiliza um perceptron, que segue a Equação 3.4, para aproximar os $\mathcal{Q}$-valores. O perceptron tem função de ativação $\theta(x) = x$. Aprende por meio do método do gradiente descendente.
- `SingleQLAI`: Utiliza uma rede *feedforward* para aproximar os $\mathcal{Q}$-valores. A rede possui uma camada interna, composta de 16 perceptrons e funções de ativação tanh, e uma camada de saída, composta de um perceptron e função de ativação $\theta(x) = x$. Aprende por meio do método Adam.

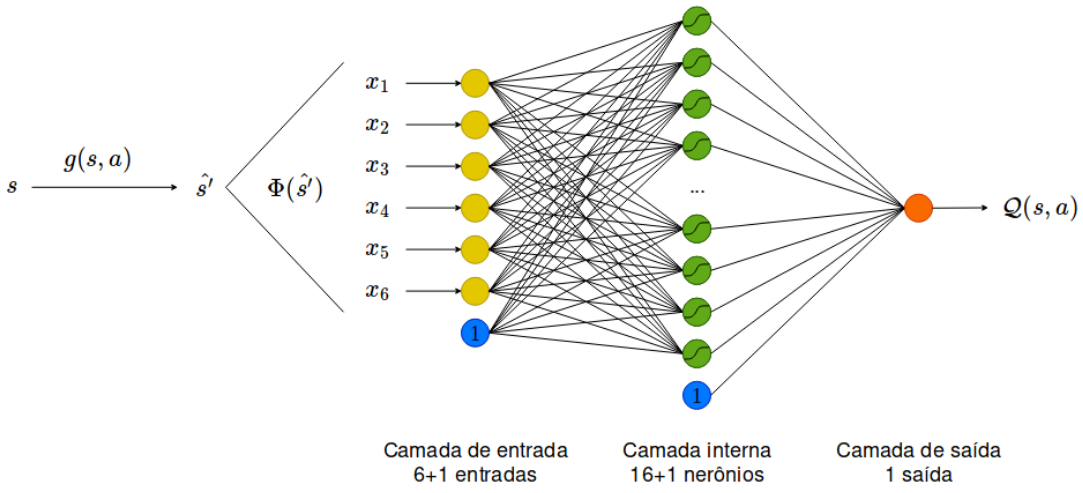

Figura 5.6: *Diagrama da rede utilizada na função $\mathcal{Q}$ da classe* `SingleQLAI`.

- `MemoryQLAI`: Utiliza uma rede com módulo de memória para aproximar os $\mathcal{Q}$-valores de estados consecutivos. A rede possui uma camada, composta por 9 perceptrons e função de ativação tanh, seguida de dois módulos recorrentes LSTM sequenciais

com estado interno de tamanho 16, seguidos pela camada de saída, composta de um perceptron e função de ativação $\theta(x) = x$. Aprende por meio do método Adam e tem profundidade de recorrência $T = 4$.

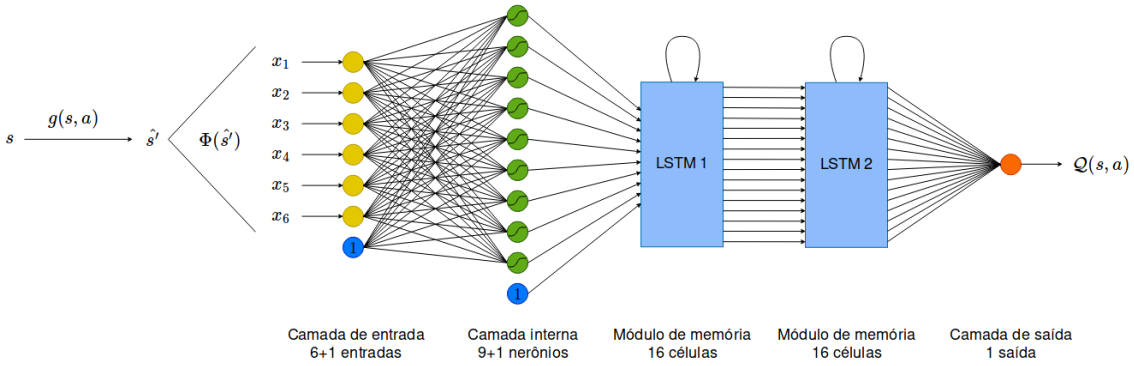

Figura 5.7: *Diagrama da rede utilizada na função Q da classe* `MemoQLAI`.

- `MultiQLAI`: Utiliza uma rede *feedforward* para aproximar ao mesmo tempo todos os Q-valores das ações disponíveis a partir de um estado. A rede possui duas camadas internas, compostas por 12 perceptrons e função de ativação tanh, e uma camada de saída, composta por tantos perceptrons quanto ações disponíveis e função de ativação $\theta(x) = x$. Aprende por meio do método Adam.

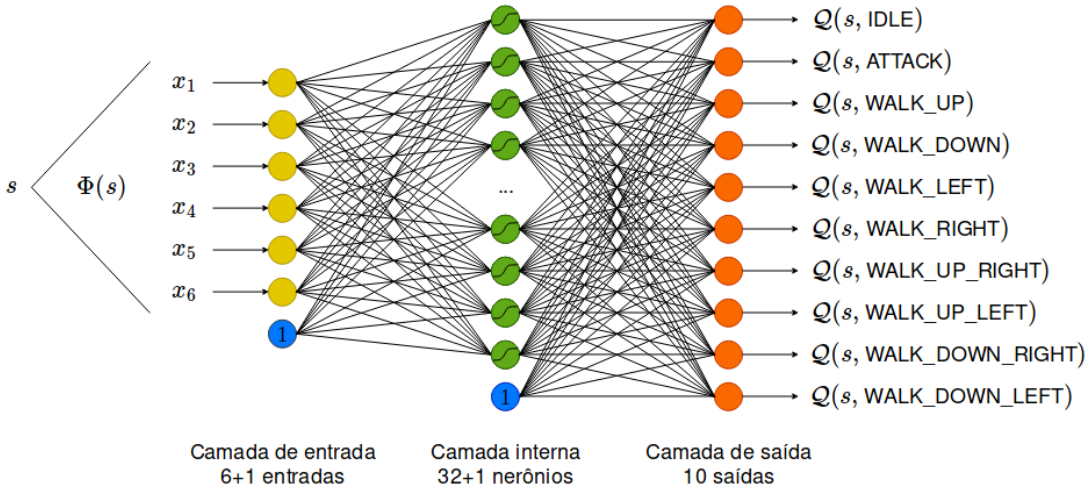

Figura 5.8: *Diagrama da rede utilizada na função Q da classe* `MultiQLAI`.

Em todas foi implementado a taxa de exploração com decaimento (Subseção 3.1.2) e o *replay* de episódios (Subseção 3.1.3).

Um estado do jogo na visão de uma IA foi definido como sendo composto pelos seguintes valores:

- `self_pos`: Posição do agente;
- `self_life`: Pontos de vida do agente;
- `self_maxlife`: Pontuação máxima de vida do agente;
- `self_damage`: Pontuação de ataque do agente;
- `self_defense`: Pontuação de defesa do agente;
- `self_act`: Última ação executada pelo agente;
- `enemy_pos`: Posição do inimigo;
- `enemy_life`: Pontos de vida do inimigo;
- `enemy_maxlife`: Pontuação máxima de vida do inimigo;
- `enemy_damage`: Pontuação de ataque do inimigo;
- `enemy_defense`: Pontuação de defesa do inimigo;
- `enemy_act`: Última ação executada pelo inimigo.

Os métodos `get_features` e `get_features_after_action` fazem o papel da função Φ que processa o estado e devolve um vetor de entrada para a rede neural. Tal vetor é composto dos seguintes valores:

- `enemy_dist`: Distância do agente até o inimigo normalizada pelo tamanho da diagonal da janela do jogo;
- `self_life`: Pontos de vida do agente normalizados por sua pontuação máxima de vida;
- `enemy_life`: Pontos de vida do inimigo normalizados por sua pontuação máxima de vida;
- `enemy_attacking`: 1 se o inimigo está atacando ou -1 caso contrário;
- `enemy_dir_x`: Componente x do vetor de direção do inimigo;
- `enemy_dir_y`: Componente y do vetor de direção do inimigo.

Todos os personagens possuem um conjunto de ações disponíveis a cada instante do jogo. Atualmente o conjunto de ações de todos os personagens são iguais, mas é possível criar ações adicionais. As ações disponíveis são: ficar parado, atacar e andar para oito direções diferentes (cima, baixo, direita, esquerda e diagonais).

Os quatro tipos de IAs foram minimamente testadas pelo desenvolvedor em um ambiente controlado dentro o jogo chamado “sala de testes” (Figura 5.9), onde o jogador pode jogar contra o mesmo inimigo várias vezes, testando os limites do sistema, ou duas IAs podem batalhar uma contra a outra várias vezes, testando as diferenças entre uma e outra.

Quando um personagem derrota o outro nessa sala ou quando se passaram 20 segundos sem que isso aconteça a sala é reiniciada e os personagens são colocados em posições aleatórias com a vida recuperada e com as mesmas IAs da rodada anterior. Nesse ambiente, as IAs têm um minuto para aprender alguma política útil que maximize a função recompensa abaixo

$$R(s, a, s') = \frac{v_s^{(E)} - v_{s'}^{(E)}}{v_s^{(E)}} - \frac{v_s^{(A)} - v_{s'}^{(A)}}{v_s^{(A)}} \tag{5.1}$$

na qual $v_s^{(E)} \in [0, 1]$ e $v_s^{(A)} \in [0, 1]$ são as vidas normalizadas do inimigo e do agente, respectivamente, no estado s e $v_{s'}^{(E)} \in [0, 1]$ e $v_{s'}^{(A)} \in [0, 1]$ são as vidas normalizadas do inimigo e do agente, respectivamente, no estado s'.

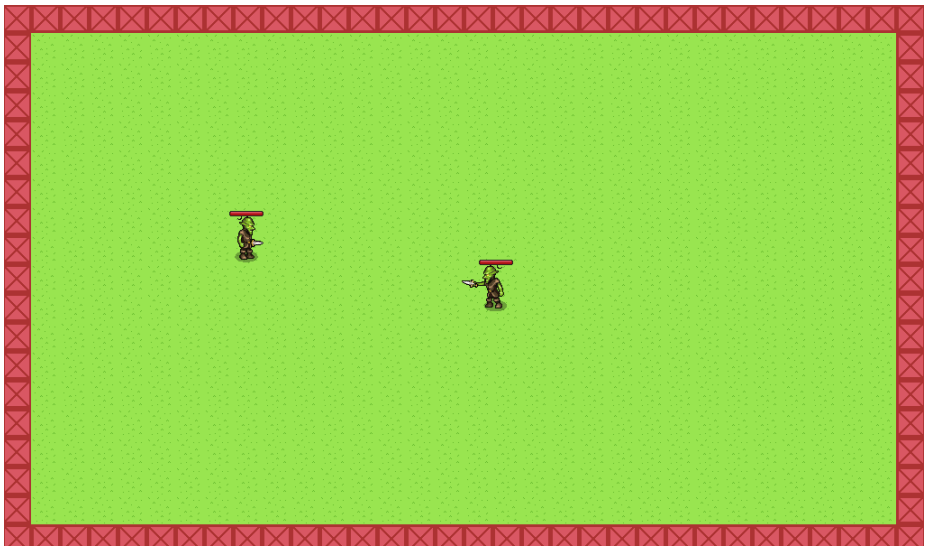

Figura 5.9: *Dois goblins batalhando na sala de testes.*

Neste contexto, política útil se refere à uma política estável na qual é possível visualizar claramente o objetivo da IA, ou seja, uma política que se distingue o máximo do aleatório possível. Duas políticas úteis emergiram nos testes: uma na qual o inimigo foge do jogador, tentando assim minimizar o segundo termo da função de recompensa, e outra na qual o inimigo vai na direção do jogador e o ataca, tentando assim maximizar o primeiro termo da função de recompensa. Desses dois casos, a política desejada é a segunda, portanto ela será chamada de política ótima de agora em diante.

A partir desses experimentos constatou-se que a `PerceptronQLAI` é a que demora menos para atualizar os pesos, porém quase nunca aprende a política ótima; a `SingleQLAI` demora um pouco mais para atualizar os pesos que a `PerceptronQLAI` porém aprende a política ótima três quartos das vezes; a `MemoryQLAI` demora duas vezes mais que a `SingleQLAI` para atualizar os pesos, mas tem a mesma performance na batalha que ela; e a `MultiQLAI` demora o mesmo que a `SingleQLAI` para atualizar os pesos porém não consegue aproximar nenhuma política útil em alguns minutos de treino.

Conclui-se a partir dos experimentos que a `SingleQLAI` é a AI que tem a melhor performance, já que ela minimiza o tempo de aprendizado e aprende a política ótima na maioria das vezes, por esse motivo ela foi utilizada em todos os agentes do jogo.

Apesar desses resultados são necessários mais experimentos para analisar estatisticamente como cada atributo de aprendizado influencia nos resultados da IA, para estudar como tornar possível a integração da `MultiQLAI` no jogo e para otimizar o código para diminuir o tempo de atualização dos pesos.

5.4 Aventuras

Aventuras são constituídas de um conjunto de salas. As salas têm formato retangular de mesmo tamanho e uma saída para cada lado, isso faz com que o mapa de uma aventura possa ser representado por uma grade de salas.

No código uma aventura é representada por um arquivo de configuração que contém os tipos de salas que podem aparecer na aventura, os tipos de inimigos que podem aparecer e os tipos de recursos que podem aparecer. Como não é possível obter dados sobre a árvore das cenas antes de instanciá-las, é necessário especificar também os metadados das salas (como a lista de saídas que ela possui, o número máximo de inimigos e o número máximo de recursos) e dos recursos (como o valor máximo e mínimo dos atributos numéricos) nesse arquivo.

Foram implementados dois algoritmos para a criação das aventuras: o aleatório e o baseado no CAP, ambos criam um grafo em grade de vizinhança 4 aleatório para representar o mapa, onde os nós são salas e as arestas indicam se há ou não conexão entre salas vizinhas. Para cada um dos algoritmos foi criada uma classe que contém o método de geração de aventura, tais classes são derivadas de uma classe base chamada `GeneratorBase` (Figura 5.10).

O algoritmo aleatório escolhe os inimigos e os recursos das salas seguindo distribuições

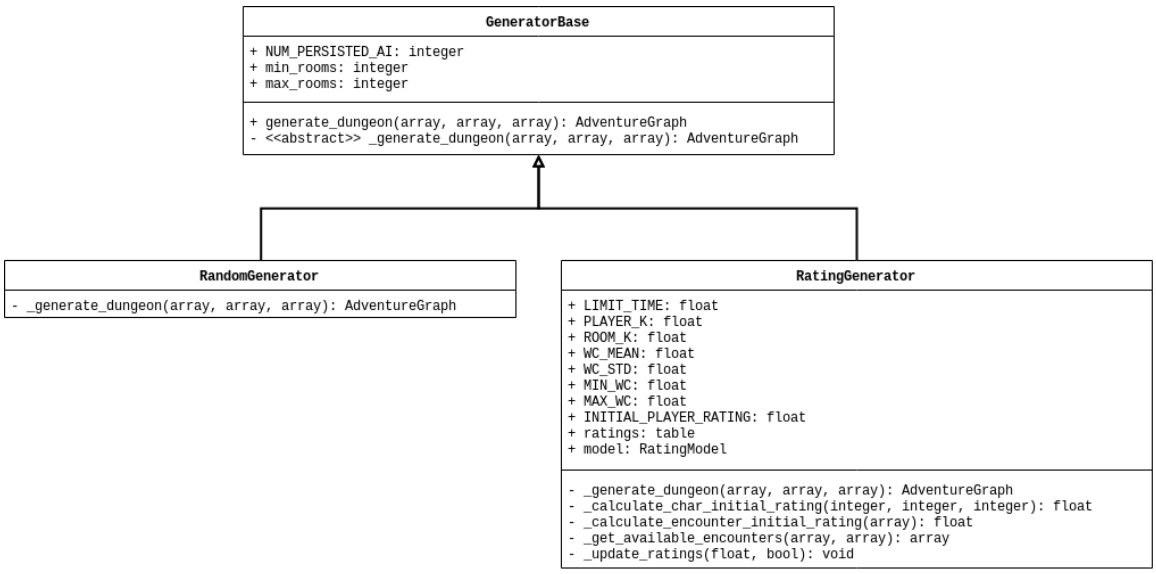

Figura 5.10: *Diagrama UML referente à classe* `GeneratorBase`*. O diagrama mostra também as classes derivadas* `RandomGenerator` *e* `RatingGenerator`*.*

uniformes. Os inimigos de cada sala são escolhidos a partir da lista de inimigos disponíveis na aventura, onde todos os inimigos têm a mesma chance de serem escolhidos. Os recursos e os valores de seus atributos também são escolhidos a partir da lista de recursos disponíveis na aventura segundo distribuição uniforme.

O algoritmo baseado no CAP sorteia uma probabilidade de vitória do jogador $p \sim \mathcal{N}^{+}_{-}(0.725, 0.5, 0.95, 0.07)$ para cada sala, calcula uma dificuldade ideal seguindo a Equação 3.27 e coloca na sala o conjunto de inimigos com dificuldade mais parecida com o valor obtido, assim como é descrito no algoritmo da Subseção 3.3.4. Os recursos da sala são escolhidos aleatoriamente, assim como no algoritmo aleatório, porém os valores dos atributos são definidos de acordo com o p escolhido: quanto maior o valor de p, menores serão as recompensas, e vice-versa, fazendo com que o jogador receba recompensas maiores em salas mais difíceis e recompensas menores em salas mais fáceis.

5.5 Salas

Esse trabalho considera como sala toda cena que contém uma instância do jogador como um de seus descendentes. Essa definição engloba por exemplo a cena que define a cidade principal e as cenas que definem as salas da aventura.

Cada sala é constituída de três nós do tipo `TileMap`, dois nós do tipo `Position2D` que definem os limites para a câmera e um conjunto de nós que definem as saídas da sala.

Um nó do tipo `TileMap` é literalmente um mapa de ladrilhos. O nó contém uma grade de *tiles*, na qual cada posição pode receber uma *tile* diferente, tornando possível assim decorar uma sala usando um *tileset*. Os três nós do tipo `TileMap` são responsáveis por conter o chão, as paredes e o teto da sala respectivamente, e são desenhados pelo Godot nessa ordem.

Como os personagens têm que ser desenhados acima do chão, abaixo do teto e junto com as paredes, eles são instanciados como filhos do `TileMap` que representa as paredes, porém, para que eles possam ser desenhados na ordem certa é necessário ativar o atributo `y_sort` do `TileMap`, assim ele desenhará os filhos pela posição no eixo y, colocando objetos com menor valor de y atrás de objetos com y maior (incluindo as paredes).

Cada *tile* pode conter uma caixa de colisão personalizada, assim é possível criar paredes que bloqueiam os personagens.

Os *tilesets* podem ser configurados para usar o algoritmo *autotile* do Godot. Esse algoritmo seleciona automaticamente o *tile* que será desenhado de acordo com os espaços vizinhos, fazendo com que o desenvolvedor não precise se preocupar em trocar os *tiles* das bordas do mapa por *tiles* de borda.

A câmera do jogo segue o jogador por onde ele passa, porém é necessário definir limites para ela caso o desenvolvedor não queira que o jogador visualize partes da sala que não foram feitas para visualização. A partir dos nós `Position2D` é possível definir visualmente através do editor quais os limites da câmera em cada sala, funcionalidade que não está implementada no Godot.

Qualquer objeto que defina uma posição pela qual o jogador possa ser colocado, emita um sinal `interacted` quando o jogador passa por ele e implementa um método `remove` pode ser adicionado no conjunto de saídas de uma sala. O método `remove` deve ser implementado de forma a remover visualmente a saída caso ela não seja necessária para a sala e a posição do jogador deve ser interpretada como a posição em que o jogador irá aparecer caso ele volte para a sala pela saída.

As salas das aventuras também têm um conjunto de nós `Position2D` que indicam as posições que os inimigos podem aparecer e um outro conjunto de nós que indicam as posições que os recursos podem aparecer, seguindo os algoritmos descritos na subseção 5.4.

5.6 Cálculo de dificuldade dos encontros

Esse projeto utiliza uma versão mais específica do algoritmo CAP na qual só há um jogador, portanto iremos denotar sua habilidade somente por θ. Vamos então definir como são os problemas enfrentados pelo jogador dentro do jogo.

Seja $\mathcal{I} = \{I_1, I_2, ..., I_k\}$ o conjunto de tipos de inimigos definidos no arquivo de configuração de uma aventura, no qual I_i, $1 \leq i \leq k$, são tipos diferentes entre si definidos no banco de dados de inimigos (ver subseção 5.2), e seja $\mathcal{E} \subset \mathbb{N}^k$ o conjunto de encontros possíveis na aventura, onde os k elementos de cada encontro $E \in \mathcal{E}$ representam as multiplicidades dos respectivos k tipos de inimigos de $\mathcal{I}$. Definimos então o número β_E como sendo a dificuldade do encontro E.

Ao criar uma aventura, para cada sala j da aventura, com capacidade máxima de inimigos M_j, o jogo associa um encontro E_j pertencente à um conjunto $\mathcal{E}_j = \{(e_1, e_2, ..., e_k) \in \mathcal{E} : 0 < \sum_{n=1}^{k} e_n \leq M_j\}$ de acordo com o algoritmo de geração de aventura que está sendo usado. O algoritmo baseado no CAP em específico associa uma probabilidade de vitória p_j para cada sala, obtém uma dificuldade ideal δ_j seguindo a fórmula 3.27 e escolhe E_j seguindo a fórmula

$$E_j = \underset{E \in \mathcal{E}_j}{\operatorname{argmin}} |\delta_j - \beta_E| \tag{5.2}$$

Segundo Van der Maas e Nyamsuren, 2017 as habilidades dos jogadores e as dificuldades dos problemas do algoritmo CAP são inicializadas com zero e é necessário que uma certa quantidade de pessoas resolva os problemas disponíveis para que os valores das dificuldades convirjam. Em um jogo de um jogador isso não é possível, já que ele só jogará o jogo inteiro uma vez. Uma alternativa seria inicializar os valores de acordo com o resultado de um conjunto controlado de jogadores, porém isso assume que os desenvolvedores tenham tempo para acompanhá-los e dinheiro para contratá-los.

A solução utilizada nesse projeto foi baseada na 4ª edição das regras do jogo *Dungeons & Dragons* (Wyatt, 2008). *D&D*, como é comumente chamado, é um jogo multijogador do estilo RPG[2] de mesa com temática de fantasia medieval. Nele os jogadores criam personagens fictícios seguindo o manual de instruções, e um dos jogadores, chamado de mestre, é responsável por pensar em um mundo onde todos os personagens se encontram e se aventuram por ele. Cada jogador interpreta e controla seu personagem durante o jogo. Nas aventuras é possível que os personagens encontrem inimigos, e o sistema de combate

[2] do inglês *Role Playing Game*, ou jogo de interpretação de papéis

é complexo o suficiente para que eles consigam realizar uma variedade ilimitada de ações para combatê-los.

O manual de instruções conta com uma descrição detalhada de mais de 400 monstros diferentes que podem ser usados como inimigos durante o jogo, cada um conta com um conjunto de atributos de combate. Como os atributos são vários e têm vários propósitos, vamos resumi-los em: pontos de ataque, pontos de defesa, pontos de vida e nível. Segundo o manual, se um inimigo tem nível X então quatro jogadores de nível X conseguem enfrentá-lo sem dificuldades.

Apesar do nível ser calculado segundo os outros atributos, no manual esse cálculo não é explicitado, porém é falado sobre um método para aumentar ou diminuir o nível de um monstro do manual. Segundo esse método, para cada nível aumentado é necessário aumentar em um a pontuação de defesa e em média oito pontos de vida, e a cada dois níveis é necessário aumentar a pontuação de ataque em um também. Uma forma mais genérica de expressar essas mudanças é usando o conjunto de equações abaixo

$$\begin{cases} \dfrac{\partial l}{\partial a} = r_a \\ \dfrac{\partial l}{\partial d} = r_d \\ \dfrac{\partial l}{\partial v} = r_v \end{cases} \tag{5.3}$$

onde $l \in \mathbb{R}_+$ representa o nível do inimigo, $a \in \mathbb{R}_+$ os pontos de ataque, $d \in \mathbb{R}_+$ os pontos de defesa, $v \in \mathbb{R}_+$ os pontos de vida e $r_a, r_d, r_v \in \mathbb{R}_+$ são constantes.

Podemos então integrá-las em suas respectivas variáveis e obter um segundo conjunto de equações

$$\begin{cases} l = r_a(a - c_a) \\ l = r_d(d - c_d) \\ l = r_v(v - c_v) \end{cases} \tag{5.4}$$

nas quais as constantes $c_a, c_d, c_v \in \mathbb{R}_+$ representam as pontuações de ataque, defesa e vida iniciais (no nível 0) respectivamente.

Esse conjunto de equações não é muito flexível, já que dado as constantes e um dos atributos é possível inferir os outros três. Isso limita a criatividade do desenvolvedor, já que os atributos possíveis ficam limitados, e aumenta a previsibilidade do jogo, tornando-o

menos divertido para o jogador, por isso a fórmula abaixo foi criada a partir dele para contornar esses problemas

$$l(a, d, v) = \frac{r_a(a - c_a) + r_d(d - c_d) + r_v(v - c_v)}{3} \tag{5.5}$$

Todas as soluções do conjunto 5.4 são também soluções da fórmula anterior, porém ela permite obter o nível a partir dos outros três atributos, e não o oposto, tornando-a muito mais flexível.

Como é mais difícil enfrentar dois inimigos de nível X do que um inimigos de nível $2X$, já que há menos espaço para fugir e a variedade de habilidades é maior, o manual do *D&D* também especifica que para calcular o nível de um grupo de monstros maior que o número de jogadores é necessário multiplicar a soma dos níveis de cada monstro do grupo por $\frac{1}{2}\left(1 + \left\lfloor \frac{M}{P} \right\rfloor\right)$, onde M é o número de monstros do grupo e P é o número de jogadores.

Nesse projeto os atributos de combate dos personagens são exatamente os mesmos usados nas fórmulas anteriores, portanto é possível utilizá-las para calcular o nível dos encontros do jogo e utilizar esse nível como base para inicializar a dificuldade do encontro no algoritmo CAP. Dado um encontro $(e_1, e_2, ..., e_k) \in \mathcal{E}$ referente à um conjunto de tipos de inimigos $\mathcal{I} = \{I_1, I_2, ..., I_k\}$ e sejam a_I, d_I e v_I os atributos de um inimigo $I \in \mathcal{I}$, este projeto utiliza a função $L : \mathcal{I} \longrightarrow \mathbb{R}$ para calcular a dificuldade de inimigos individualmente

$$L(I) = \frac{9a_I + 9d_I + v_I - 10}{90} \tag{5.6}$$

Que pode ser obtida a partir da função 5.5 fixando $r_a = \frac{3}{10}$, $r_d = \frac{3}{10}$, $r_v = \frac{1}{30}$, $c_a = 0$, $c_d = 0$ e $c_v = 10$. Utilizando essa função em conjunto com o multiplicador para grupos de inimigos o jogo calcula a dificuldade inicial β_E de um encontro E seguindo a fórmula

$$\beta_E = \frac{1}{2}\left(1 + \sum_{i=1}^{k} e_i\right) \sum_{i=1}^{k} e_i L(I_i) \tag{5.7}$$

Dado que o jogador também é definido por seus pontos de ataque, defesa e vida, a função L também é utilizada para calcular a habilidade inicial θ dele.

Ao longo das aventuras o jogador enfrenta encontros de inimigos, então o jogo atualiza a habilidade do jogador θ e a dificuldade do encontro de acordo com as fórmulas abaixo, que foram adaptadas das fórmulas 3.22, 3.23, 3.24 e 3.25 do CAP.

$$S_E = \begin{cases} (2x_E - 1)\left(1 - \dfrac{t_E}{60}\right) & \text{, se } t_E < 60 \\ 0 & \text{, c.c.} \end{cases} \tag{5.8}$$

$$E[S_E|\theta - \beta_E] = \frac{e^{2(\theta-\beta_E)} + 1}{e^{2(\theta-\beta_E)} - 1} - \frac{1}{\theta - \beta_E} \tag{5.9}$$

$$\tilde{\theta} = \theta + \frac{4}{10}(S_E - E[S_E|\theta - \beta_E]) \tag{5.10}$$

$$\tilde{\beta}_E = \beta_E + \frac{4}{10}(E[S_E|\theta - \beta_E] - S_E) \tag{5.11}$$

onde $x_E \in \{0, 1\}$ é 1 se o jogador venceu o encontro E ou 0 caso contrário, $t_E \in \mathbb{R}_+$ é o tempo em segundos que o jogador demorou para vencer ou perder do encontro E, $\tilde{\theta} \in \mathbb{R}$ é a nova habilidade do jogador e $\tilde{\beta}_E \in \mathbb{R}$ é a nova dificuldade do encontro E.

Foi estabelecido que o tempo máximo para que o jogador vença um encontro é de um minuto, pois é tempo suficiente para que a maioria dos jogadores iniciantes derrotem um grupo de inimigos básicos do jogo, e foi estabelecido também que os coeficientes K e H das equações 5.10 e 5.11 sejam constantes e iguais à $\frac{4}{10}$, pois esse valor conseguiu balancear bem a progressão do jogo nos testes realizados.

5.7 Interface do usuário

Em um jogo chamamos de interface do usuário todo elemento visual que não pertence ao mundo do jogo que foi feito para facilitar processos dentro e fora do jogo para o jogador, como menus, *popups* e *displays*.

O Godot disponibiliza vários tipos de nós que facilitam a criação de interfaces, todos eles possuem atributos que indicam tamanho e posição do objeto relativo ao nó pai e aos nós irmãos. Os nós utilizados na criação das interfaces desse jogo foram:

- `CenterContainer`: Centraliza os nós filhos de acordo o tamanho e a posição desse nó.
- `MarginContainer`: Limita o conteúdo dos nós filhos para que eles fiquem inteiramente dentro da margem definida nesse nó.
- `PanelContainer`: Cria um painel no estilo *ninepatch* de tamanho variado e limita o conteúdo dos nós filhos para que eles fiquem dentro do painel, assim como o

`MarginContainer` faz.

- `VBoxContainer`: Organiza os filhos seguindo um eixo vertical, de forma que eles fiquem o mais perto possível uns dos outros e de forma que o conteúdo deles não se sobreponha.
- `HBoxContainer`: Mesmo comportamento do `VBoxContainer`, porém utiliza um eixo horizontal.
- `ItemList`: Contém uma lista de itens internos que são constituídos de uma *sprite* e/ou um texto, e mostra esses itens em formato de lista. Cada item pode ser selecionado, com um clique, ou ativado, com dois cliques, fazendo com que um sinal com informações do item clicado ou ativado seja emitido. É possível mudar o estilo do fundo da lista e o estilo do item selecionado usando *Ninepatch*.
- `Button`: Um botão que pode conter um texto e/ou uma imagem. Quando o ele é pressionado um sinal é emitido. É possível mudar o estilo da imagem de fundo de cada um de seus estados (normal, pressionado, focado, desabilitado e *hover*) usando *Ninepatch*.
- `Label`: Contém um texto que pode ser formatado.
- `TextureRect`: Contém uma *sprite* que pode ser expandida de vários jeitos diferentes.

A partir desses nós foram criados alguns objetos reutilizáveis com a temática do jogo, para que assim seja fácil modificar o estilo de toda a interface ao mesmo tempo

- `SmallLabel`, `MediumLabel` e `BigLabel`: São `Labels` que utilizam a fonte do jogo e já têm os tamanhos das fontes pré-configurados. Foram criadas para que seja fácil modificar o estilo de todos os textos do jogo ao mesmo tempo.
- `SmallButton`: Botão que usa o mesmo estilo de texto do `SmallLabel`.
- `BigBorderlessButton`: Botão sem imagem de fundo e com o mesmo tamanho de texto e fonte que o `BigLabel`. Como ele não possui imagem de fundo, o estado do botão é indicado pela cor do texto.
- `BronzeBorderPannel`: `PanelContainer` que utiliza uma *sprite* de fundo na cor marrom e bordas decoradas de cor bronze como *ninepatch*.
- `MattePanel`: `PanelContainer` que utiliza uma *sprite* de fundo fosco e bordas pretas como *ninepatch*.

A partir desses objetos foram criadas interfaces reutilizáveis mais complexas, com lógica e interface de código próprias

- `MoneyDisplay`: Contém um símbolo de moeda e um texto pequeno para mostrar uma quantidade de dinheiro. Sua interface de código contém somente uma função para modificar a quantidade mostrada.

- `ItemDisplay`: Contém um `ItemList` configurado para mostrar uma lista de itens do jogo em forma de grade. Um item do jogo é representado visualmente por uma *sprite*, definida no banco de dados de itens. A imagem de fundo desse elemento é a mesma do `MattePanel`, porém ela contém vários ladrilhos para acomodar cada item. Sua interface de código é constituída de uma função para mostrar uma lista de itens e uma função que diz se a lista está cheia. Quando o jogador seleciona um item com o botão esquerdo, um sinal é emitido com informações do item selecionado, o mesmo acontece ao selecionar com o botão direito ou ativar os itens.

- `ItemInfo`: Componente criado para mostrar informações de um item. Ele posui um `VBoxContainer` com nós filhos que mostram o nome, *sprite*, descrição e preço do item, além de conter também um botão de ação. Sua interface de código é constituída de uma função para mostrar um item e uma ação e uma função para remover o item que está sendo mostrado. Ao pressionar o botão de ação, o componente emite um sinal de acordo com a ação que foi recebida junto do item, passando junto informações sobre o item que está sendo mostrado. As ações disponíveis são: não fazer nada (nesse caso o botão não aparece), comprar, vender, usar, equipar e desequipar.

- `EquipDisplay`: Componente que mostra uma lista de equipamentos, possuindo uma entrada para cada tipo de equipamento. Atualmente o personagem só pode ser equipado com uma espada, porém é possível adicionar tipos de equipamentos novos, como chapéu/capacete, cota, perneiras, botas, luvas, escudo, etc. Sua interface de código possui somente uma função para mostrar os equipamentos de acordo com uma tabela. Ao selecionar um item equipado, o componente emite um sinal com as informações do item.

A partir desses componentes lógicos foram criadas as seguintes telas do jogo:

- Mochila do personagem: Um *popup* cujo o fundo é um `BronzeBorderPannel`, tem um `MoneyDisplay` no topo para mostrar quanto dinheiro o jogador possui, um `ItemDisplay` para mostrar os itens que o jogador tem na mochila, um `EquipDisplay` para mostrar os equipamentos que estão equipados no personagem e um `ItemInfo` para mostrar as informações dos itens e para que o jogador possa usá-los, equipá-los ou desequipá-los.

Figura 5.11: *Mochila do personagem no jogo.*

- Loja: Também é um popup com fundo `BronzeBorderPanel` e possui um `MoneyDisplay` no topo para mostrar quanto dinheiro o jogador possui, um `ItemDisplay` para mostrar os itens disponíveis na loja e um `ItemInfo` para mostrar as informações dos itens e para que o jogador possa comprá-los.

Figura 5.12: *Loja de poções no jogo.*

- Menu principal: A imagem de fundo é um `TextureRect` e a tela possui uma caixa com fundo `MattePannel` onde estão botões do tipo `BigBorderlessButton`. A partir do menu é possível criar um jogo novo (nesse caso o jogo salvo será deletado), carregar um jogo salvo, ir para a tela de créditos ou sair do jogo.

Figura 5.13: *Menu principal do jogo.*

- Tela de créditos: Utiliza a mesma imagem de fundo do menu, porém deslocada, e possui um título feito com `BigLabel`, subtítulos para cada área de trabalho (programadores e artistas), que são `MediumLabel`, nomes dos contribuidores, que são `SmallLabel`, e um `BigBorderlessButton` para voltar para o menu principal.

Figura 5.14: *Menu de créditos do jogo.*

5.8 Arte

A grande maioria das *sprites* utilizadas no jogo foram feitas por outras pessoas sob licença Creative Commons Attribution-ShareAlike 3.0[3] (CC BY-SA 3.0) ou GNU General Public License 3.0[4] (GNU-GPL 3.0), e compartilhadas através site Open Game Art[5]. As *sprites* feitas por mim estão sob licença CC BY-SA 3.0, assim como as do site.

Na tela de créditos do jogo (Figura 5.14) é possível visualizar a lista dos autores das *sprites* utilizadas.

[3]Mais informações sobre CC BY-SA 3.0 no site https://creativecommons.org/licenses/by-sa/3.0/ (Último acesso em 10/11/2019)

[4]Mais informações sobre GNU-GPL 3.0 no site https://www.gnu.org/licenses/gpl-3.0.html (Último acesso em 10/11/2019)

[5]Site principal do Open Game Art: https://opengameart.org/ (Último acesso em 10/11/2019)

Capítulo 6

Conclusões

Neste trabalho foi implementado um jogo PvE que adapta sua dificuldade à habilidade do jogador e que utiliza aprendizagem por reforço na criação dos agentes adversários. Na criação do jogo foi utilizado o motor de jogos Godot e a biblioteca de aprendizado de máquina PyTorch.

Dentre as técnicas de adaptação de dificuldade pesquisadas, foi escolhido o algoritmo CAP para essa função. Junto com uma ordenação preliminar dos encontros de inimigos baseada no jogo D&D, o algoritmo ordena os encontros de acordo com a performance do jogador, garantindo que o ele tenha uma curva de aprendizado suave e aproveite o jogo, independente de seu estilo de jogo.

Para a criação dos agentes foi usado o algoritmo *Q-Learning* em conjunto com redes neurais, escolhido dentre outros algoritmos de aprendizagem por reforço. Tal algoritmo permite que o desenvolvedor crie agentes com as mais diversas habilidades facilmente, pois o algoritmo se encarrega de integrá-las de forma que o agente pareça inteligente ao jogador.

A biblioteca PyTorch, otimizada para aprendizado, se encarrega de garantir que o aprendizado e a execução dos agentes não consumam muito processamento, garantindo assim que quedas no FPS não incomodem o jogador.

De acordo com os poucos testes informais feitos, o projeto cumpriu o objetivo de ser um jogo com dificuldade dinâmica e utilizar aprendizagem por reforço nos agentes adversários diminuindo o mínimo possível o FPS do jogo.

6.1 Futuro do projeto

Durante esse ano de desenvolvimento houveram várias ideias que tiveram que ser despriorizadas por conta do pouco tempo de projeto. Essa seção compila algumas das ideias para estender o trabalho e para resolver alguns problemas no projeto final.

Se o número de agentes ativos ao mesmo tempo for superior à três, a experiência do usuário é prejudicada consideravelmente pela queda no FPS. Esse problema pode ser resolvido de forma rápida se, de algum modo, for possível executar instruções em paralelo ou assincronamente entre o Godot e o interpretador Python, o que atualmente não parece ser possível. Outras soluções envolvem implementar uma biblioteca nativa de aprendizado de máquina no Godot, ou ainda reescrever o jogo em Python utilizando o motor PyGame, evitando assim as dificuldades da interface entre o Godot e o Python.

Seria interessante também testar outros algoritmos de aprendizado, como árvores de decisão ou algoritmos evolutivos, otimizar os já existentes e também compará-los em várias tarefas com métodos tradicionais, como máquinas de estados, para visualizar suas vantagens e desvantagens.

Como dito no primeiro capítulo, o jogo por enquanto é só uma prova de conceito, porém para que o sistema possa ser testado com mais precisão é necessário adicionar mais conteúdo no jogo. Com isso o jogo se torna mais complexo, as batalhas se tornam mais complexas e difíceis e o sistema pode ser visto em ação em casos extremos. Conteúdos novos incluem adicionar novas cidades, novas aventuras e novos inimigos com habilidades únicas por exemplo.

Além disso seria ideal melhorar a experiência do usuário no geral, adicionando efeitos visuais e sonoros, tutorial e interação com personagens e com a história do jogo.

Foram feitos poucos testes informais com usuários. O ideal seria que fossem feitos testes formais, principalmente depois de adicionar mais conteúdo no jogo, para assim comprovar de fato a eficácia do método de adaptação de dificuldade. Esses testes podem ser feitos remotamente com ajuda de um arquivo de *log*, que indicaria como a ordenação do algoritmo CAP se desenvolveu ao longo do tempo, além de ajudar no processamento desses dados.

Referências

[Abu-Mostafa *et al.* 2012] Yaser S Abu-Mostafa, Malik Magdon-Ismail e Hsuan-Tien Lin. *Learning from data*. Vol. 4. AMLBook New York, NY, USA: 2012 (citado na pg. 18).

[Andrade *et al.* 2006] Gustavo Andrade, Geber Ramalho, Alex Sandro Gomes e Vincent Corruble. "Dynamic game balancing: an evaluation of user satisfaction." Em: *AIIDE* 6 (2006), pgs. 3–8 (citado na pg. 1).

[Bartle 1996] Richard Bartle. "Hearts, clubs, diamonds, spades: Players who suit MUDs". Em: *Journal of MUD research* 1.1 (1996), pg. 26 (citado na pg. 24).

[Bellman 1952] Richard Bellman. "On the Theory of Dynamic Programming". Em: *Proceedings of the National Academy of Sciences* 38.8 (1 de ago. de 1952), pgs. 716–719. issn: 0027-8424, 1091-6490. doi: 10.1073/pnas.38.8.716 (citado na pg. 13).

[Csikszentmihalyi 1990] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper Collins, 1990. 439 pgs. isbn: 978-0-06-187672-1 (citado na pg. 24).

[Demasi e Cruz 2003] Pedro Demasi e Adriano Cruz. "On-line coevolution for action games". Em: *International Journal of Intelligent Games & Simulation* 2.2 (2003) (citado na pg. 1).

[Elo 1978] Arpad E. Elo. *The rating of chessplayers, past and present*. Arco Pub., 1978. 216 pgs. isbn: 978-0-668-04721-0 (citado na pg. 24).

[Eggen e Verschoor 2006] Theo J. H. M. Eggen e Angela J. Verschoor. "Optimal Testing With Easy or Difficult Items in Computerized Adaptive Testing". Em: *Applied Psychological Measurement* 30.5 (1 de set. de 2006), pgs. 379–393. issn: 0146-6216. doi: 10.1177/0146621606288890 (citado na pg. 27).

[Funahashi 1989] Ken-Ichi Funahashi. "On the approximate realization of continuous mappings by neural networks". Em: *Neural Networks* 2.3 (1 de jan. de 1989), pgs. 183–192. ISSN: 0893-6080. DOI: 10.1016/0893-6080(89)90003-8 (citado na pg. 19).

[Gamma *et al.* 1994] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 31 de out. de 1994. 457 pgs. ISBN: 978-0-321-70069-8 (citado na pg. 33).

[Gers *et al.* 2000] Felix A. Gers, Jürgen Schmidhuber e Fred Cummins. "Learning to Forget: Continual Prediction with LSTM". Em: *Neural Computation* 12.10 (out. de 2000), pgs. 2451–2471. ISSN: 0899-7667, 1530-888X. DOI: 10.1162/089976600300015015 (citado na pg. 21).

[Hochreiter e Schmidhuber 1997] Sepp Hochreiter e Jürgen Schmidhuber. "Long short-term memory". Em: *Neural Computation* 9.8 (1 de nov. de 1997), pgs. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735 (citado nas pgs. 21, 22).

[Kingma e Ba 2017] Diederik P. Kingma e Jimmy Ba. "Adam: a method for stochastic optimization". Em: (29 de jan. de 2017). arXiv: 1412.6980 (citado na pg. 20).

[Koster 2013] Raph Koster. *Theory of Fun for Game Design*. O'Reilly Media, Inc., 8 de nov. de 2013. 367 pgs. ISBN: 978-1-4493-6317-8 (citado na pg. 1).

[Lample e Chaplot 2017] Guillaume Lample e Devendra Singh Chaplot. "Playing FPS Games with Deep Reinforcement Learning". Em: (13 de fev. de 2017), pg. 7 (citado na pg. 23).

[Mnih *et al.* 2015] Volodymyr Mnih *et al.* "Human-level control through deep reinforcement learning". Em: *Nature* 518.7540 (fev. de 2015), pgs. 529–533. ISSN: 1476-4687. DOI: 10.1038/nature14236 (citado nas pgs. 16, 23).

[McCulloch e Pitts 1943] Warren S. McCulloch e Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". Em: *The bulletin of mathematical biophysics* 5.4 (1 de dez. de 1943), pgs. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259 (citado na pg. 17).

[Minsky e Papert 1969] Marvin Minsky e Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969. 317 pgs. ISBN: 978-0-262-53477-2 (citado nas pgs. 17, 18).

[Maris e Van der Maas 2012] Gunter Maris e Han Van der Maas. "Speed-Accuracy Response Models: Scoring Rules based on Response Time and Accuracy". Em: *Psychometrika* 77.4 (out. de 2012), pgs. 615–633. ISSN: 0033-3123, 1860-0980. DOI: 10.1007/s11336-012-9288-y (citado nas pgs. 24–26).

[Nyamsuren *et al.* 2018] Enkhbold Nyamsuren, Han Van der Maas e Matthias Maurer. "Set-theoretical and Combinatorial Instruments for Problem Space Analysis in Adaptive Serious Games". Em: *International Journal of Serious Games* 5.1 (26 de mar. de 2018). ISSN: 2384-8766. DOI: 10.17083/ijsg.v5i1.219 (citado nas pgs. 24, 27).

[Nystrom 2014] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 3 de nov. de 2014. 353 pgs. ISBN: 978-0-9905829-1-5 (citado na pg. 6).

[Rasmussen 2016] Jakob Rasmussen. *Are Behavior Trees a Thing of the Past?* Gamasutra. 27 de abr. de 2016. URL: https://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php (acesso em 20/11/2019) (citado na pg. 1).

[Rumelhart *et al.* 1986] David E. Rumelhart, Geoffrey E. Hinton e Ronald J. Williams. "Learning representations by back-propagating errors". Em: *Nature* 323.6088 (out. de 1986), pgs. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0 (citado na pg. 18).

[Rosenblatt 1961] Frank Rosenblatt. *Principles of Neurodynamics. Perceptrons and the Theory of Brain Mechanisms*. Cornell Aeronautical Lab Inc Buffalo NY, 1961 (citado na pg. 17).

[Russell *et al.* 2010] Stuart J. Russell, Stuart Jonathan Russell, Peter Norvig e Ernest Davis. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010. 1153 pgs. ISBN: 978-0-13-604259-4 (citado na pg. 1).

[Vorderer *et al.* 2003] Peter Vorderer, Tilo Hartmann e Christoph Klimmt. "Explaining the Enjoyment of Playing Video Games: The Role of Competition". Em: *Proceedings of the second international conference on Entertainment computing* (jan. de 2003), pgs. 1–9. DOI: 10.1145/958720.958735 (citado na pg. 1).

[Van der Maas e Nyamsuren 2017] Han Van der Maas e Enkhbold Nyamsuren. "Cognitive Analysis of Educational Games: The Number Game". Em: *Topics in Cognitive Science* 9.2 (2017), pgs. 395–412. ISSN: 1756-8765. DOI: 10.1111/tops.12231 (citado nas pgs. 26, 45).

[Watkins 1989] Christopher Watkins. *Learning from Delayed Rewards*. Cambridge University, 1989. 468 pgs. (citado nas pgs. 13, 14).

[Watkins e Dayan 1992] Christopher Watkins e Peter Dayan. "Q-learning". Em: *Machine Learning* 8.3 (maio de 1992), pgs. 279–292. issn: 1573-0565. doi: 10.1007/BF00992698 (citado na pg. 16).

[Wyatt 2008] James Wyatt. *Dungeon & Dragons: Dungeon Master's Guide*. 4ª ed. Wizards of the Coast, 2008. 221 pgs. (citado na pg. 45).