ELTON YOSHIO OKAWA
HÉLIO JIN WU KIM

Aplicação de *Machine Learning* em jogos do gênero AutoChess

São Paulo
2020

ELTON YOSHIO OKAWA
HÉLIO JIN WU KIM

Aplicação de *Machine Learning* em jogos do gênero AutoChess

Trabalho apresentado à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro de Computação.

São Paulo
2020

Prof. Dr. Ricardo Nakamura

ELTON YOSHIO OKAWA
HÉLIO JIN WU KIM

Aplicação de *Machine Learning* em jogos do gênero AutoChess

Trabalho apresentado à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro de Computação.

Áreas de Concentração:
Jogos
Aprendizado por reforço
Aprendizado por reforço inverso

Orientador:
Prof. Dr. Ricardo Nakamura

São Paulo
2020

Catalogação-na-publicação

Okawa, Elton
Aplicação de Machine Learning em jogos do gênero AutoChess / E. Okawa, H. Kim -- São Paulo, 2020.
102 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Jogos 2.Aprendizado por reforço 3.Aprendizado por reforço inverso I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t. III.Kim, Hélio

AGRADECIMENTOS

À empresa *VRMonkey*^{TM} e a todos nela que colaboraram direta ou indiretamente na construção deste trabalho, por ter auxiliado o grupo tanto com aspectos técnicos quanto com a disponibilização de recursos necessários para seu desenvolvimento.

RESUMO

Em jogos digitais de forma geral, é comum encontrar personagens que são controlados por um algoritmo programado pelos desenvolvedores que apresenta um conjunto de ações padrão e bem definido. Entretanto, quando o jogador depende deste comportamento pré-definido para alcançar o objetivo dentro do jogo, sem poder fazer alterações, isto pode ser um problema pois não permite adequá-lo ao seu estilo de jogo. Isso é bem evidente em jogos do gênero AutoChess, em que após posicionar as peças no tabuleiro, elas seguem um comportamento padrão e previsível.

Desta forma, o grupo se propôs a implementar uma versão simplificada do jogo de gênero AutoChess com um sistema de treinamento das peças que permite que o usuário apresente o comportamento desejado para cada uma delas de forma fácil que não necessite de experiência prévia de uso do sistema. A interface de treinamento foi desenvolvida com a engine de jogos *UnrealEngine4*^{TM} e o treinamento em si utiliza algoritmos de Aprendizado por Reforço Inverso (IRL) juntamente com Aprendizado por Reforço (RL) chamados de, respectivamente, *Apprenticeship Learning* e *Sarsa Lambda*, para generalizar o conjunto de ações demonstradas pelo jogador.

Assim, com o sistema e a modelagem apresentada neste trabalho, foi possível notar que o personagem é capaz de generalizar comportamentos de posicionamento e ações simples, apesar de eventualmente executar ações inesperadas. Com isso, apresenta-se sugestões de ajustes e melhorias que podem ser feitas para melhorar estes resultados.

Palavras-chave: Jogos, AutoChess, Aprendizado por reforço, Aprendizado por Reforço Inverso.

ABSTRACT

In digital games in general, it's common to find characters controlled by a developer programmed algorithm which presents a well-defined standard set of actions. However, when players depend on this pre-defined behavior to reach the game's objective, not having any ways to modify it, it may be a problem because it doesn't allow it to fit their game style or strategy. It becomes very evident in games of the AutoChess genre, in which after positioning pieces on the board, they follow a defined and previsible behavior.

Thus, the group wants to implement a simplified version of a game of the AutoChess genre with a training system which allows users to show a desired behavior for each piece in an easy way that doesn't require previous experience in using the system. The training interface was developed with the game engine *UnrealEngine4*^{TM} and the training uses algorithms of Inverse Reinforcement Learning (IRL) along with Reinforcement Learning (RL) called, respectively, Apprenticeship Learning and Sarsa Lambda, to generalize the set of actions showed by the player.

Finally, with the current system and model presented in this work, it has been possible to notice that the agent is capable of generalizing behaviors with simple positioning and actions, although it sometimes does some unexpected actions. At the end, suggestions of fixes and improvements that could be done to improve these results are presented.

Keywords: Games, AutoChess, Reinforcement Learning, Inverse Reinforcement Learning

LISTA DE ABREVIATURAS E SIGLAS

MDP	Markov Decision Process
ML	Aprendizado de Máquina **(**Machine Learning)
MOBA	Multiplayer Online Battle Arena
RL	Aprendizado por Reforço (Reinforcement Learning)
IRL	Aprendizado por Reforço Inverso (Inverse Reinforcement Learning)
TDL	Temporal Difference Learning
HP	Health Points
DMG	Damage
MS	Movement Speed
UE4	Unreal Engine 4
AI	Inteligência Artificial (Artificial Intelligence)

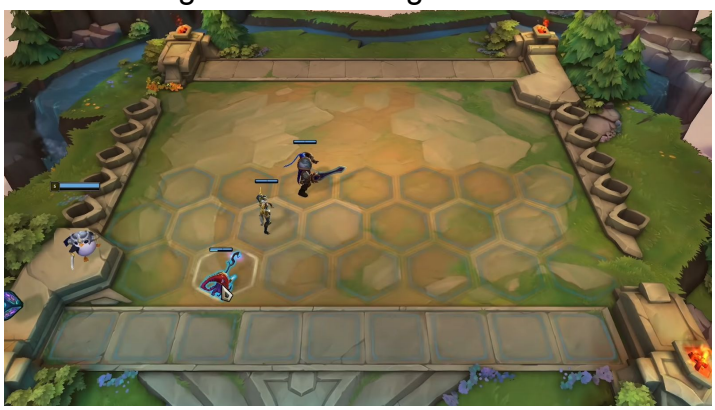
SUMÁRIO

1 INTRODUÇÃO

O gênero AutoChess surgiu no início de 2019 como uma extensão do jogo *Dota 2*™, um *Multiplayer Online Battle Arena* (MOBA) publicado pela *Valve™*. Dado o sucesso deste novo modo de jogo foram lançados uma variedade de títulos como *AutoChess*[1] *™* da *DragonNestGame*™, *Dota Underlords*[2] ™ da *Valve™* e *TeamFight Tactics*[3] *™* da *RiotGames™.*

Jogos desse gênero são baseados em rodadas de modo que os jogadores iniciam com 100 pontos de vida e a cada rodada têm à disposição um conjunto de peças com características próprias que podem ser compradas e posicionadas no seu próprio tabuleiro. Dessa forma, cada jogador vai construindo incrementalmente seu próprio time. Feito os preparos, as peças batalham automaticamente através de um comportamento pré-definido pela produtora do jogo, sem que o usuário possa ter influência decisiva sobre as ações que cada uma de suas peças terá. No fim da rodada, o jogador que perder primeiro todas as suas peças posicionadas recebe uma penalidade nos pontos de vida e o tabuleiro de todos os jogadores são resetados para o estado inicial antes da batalha. Assim, no decorrer do jogo, quem tiver zero ou menos pontos de vida é eliminado e vence o jogador que restar.

Figura 1 - TeamFight Tactics



Fonte: RiotGames

[1] DRAGONNEST™. AutoChess™. 2020. Disponível em: <https://ac.dragonest.com/en>
[2] VALVE™. Dota Underlords™. 2020. Disponível em: <https://www.underlords.com/>
[3] RIOT GAMES™. TeamFight Tactics™. 2020. Disponível em: <https://br.leagueoflegends.com/pt/featured/events/teamfight-tactics>

1.1 MOTIVAÇÃO

Nos jogos do gênero AutoChess existentes, os comportamentos das peças são fixos; dessa forma o desafio colocado para os jogadores está em selecionar as peças e posições mais adequadas de acordo com sua estratégia. Entretanto, os comportamentos muitas vezes não combinam com a estratégia do jogador ou simplesmente as peças não têm um comportamento ótimo que deveriam ter a cada turno. Assim, identifica-se o potencial de se adotar técnicas de inteligência artificial para tornar os comportamentos das peças mais flexíveis de acordo com as preferências de cada jogador. Dessa forma, um novo elemento de estratégia é adicionado ao jogo, permitindo a cada jogador o treinamento e personalização de suas peças.

1.2 OBJETIVO

O trabalho tem como objetivo aplicar técnicas de *Machine Learning (ML)* para que cada jogador seja capaz de possuir um conjunto de peças com comportamento exclusivo determinado através de uma interface intuitiva em que ele possa demonstrar o comportamento desejado.

Além disso, o jogo desenvolvido neste projeto tem potencial educacional por permitir um contato lúdico do usuário com ML através da interface de treinamento.

2 REVISÃO DA LITERATURA

Pode-se citar dois trabalhos [1] e [2] na área de jogos *Multiplayer Online Battle Arena* (MOBA) que buscam treinar inteligências artificiais capazes de derrotar jogadores profissionais e, para isso, propõe diferentes arquiteturas de algoritmo. Ambos enfrentaram desafios semelhantes a este trabalho:

1. Variedade de ações a serem realizadas ao longo da partida;
2. O resultado final da partida não depende exclusivamente da última ação selecionada, mas do conjunto de ações escolhidas durante a partida.

Semelhantemente, existe uma dissertação de mestrado [3] na qual foi desenvolvido em conjunto com algoritmos de ML, um jogo de corrida no qual o piloto deveria conseguir pilotar e andar na pista autonomamente visando conseguir percorrer o trajeto no menor tempo possível, uma vez que competiria contra outros pilotos. Esse jogo pode de certa forma ter comparações interessantes com o jogo sendo proposto para esse Trabalho de Conclusão de Curso, uma vez que também foi escolhido o algoritmo ML de aprendizado por reforço e pelo fato de que o piloto deveria constantemente estar tentando novas abordagens para pilotar na pista, ou seja, deve testar novos comportamentos e ver qual é o que irá maximizar sua recompensa. Esse comportamento do piloto de estar sempre tentando soluções novas para tentar percorrer a pista no menor tempo possível pode ser traduzido como os comportamentos que as peças de jogos do gênero AutoChess assumiriam durante os seus treinamentos e que vão gerar diferentes resultados; os que tiverem maior recompensa seriam os comportamentos que se quer assimilar às peças.

Além dos trabalhos citados anteriormente, no ano de 2019 houve um trabalho de conclusão de curso [4] que consistia em aplicar ML para poder tornar uma rede auto escalável, reconfigurando sua estrutura, adicionando ou removendo recursos baseado em fatores que podem ser variáveis como ociosidade de componentes, tráfego nela etc. Esse trabalho também faz uso de algoritmo de aprendizado por reforço para poder adaptar a rede baseado em diferentes situações. Isso se assemelha ao nosso projeto

proposto pois a cada estado e configuração variável em que o tabuleiro se encontra, as peças devem decidir quais as melhores decisões a serem tomadas de tal forma a maximizar sua recompensa no algoritmo.

Todos os trabalhos citados possuem um cenário em que o agente tem diversas ações a serem tomadas que são decididas com base nas informações sobre o ambiente em que estão inseridos. Com base nisso, os trabalhos implementaram o *Reinforcement Learning* (RL) com variações do algoritmo de aprendizado para solucionar o problema.

Abbeel e Ng [6] afirmam que uma das tarefas mais difíceis do RL é definir qual vai ser a função recompensa que vai fazer o agente convergir em direção ao comportamento ótimo. Tendo isso em mente, Ng e Russel [5] propuseram algoritmos de *Inverse Reinforcement Learning* (IRL) e mais tarde Abbeel e Ng [6] apresentaram uma forma de aprendizado em que um especialista do assunto demonstra para o agente como a tarefa deve ser executada, a partir disso, é aproximada uma função recompensa que tenta avaliar as informações do ambiente da mesma forma que o especialista para tomada de decisões.

Messer [7] implementou essa forma de aprendizado proposta por Abbeel e Ng [6] no ramo da música em que foram apresentadas sequências de melodias para o algoritmo e, a partir disso, ele foi capaz de criar outras melodias semelhantes às apresentadas.

3 ASPECTOS CONCEITUAIS

3.1 REINFORCEMENT LEARNING

Os sites [8] e [9] descrevem o RL como um tipo de técnica de ML que permite ao agente definir uma forma de agir e/ou interagir com o ambiente em que se encontra através de aprendizado por tentativa e erro que trará recompensas ou penalidades. O objetivo do algoritmo de RL é fazer com que o agente aprenda a realizar ações no ambiente que resultem na recompensa máxima.

Ainda no site [8], vê-se que falando em tipos e técnicas de ML, diferentemente do aprendizado supervisionado, no qual o feedback dado ao agente é um conjunto de ações corretas a serem tomadas, em RL o método utilizado é o de recompensar ou punir o agente baseado nas suas ações tomadas.

Vê-se que o RL tem objetivos diferentes das técnicas de aprendizado não-supervisionado. Enquanto que em técnicas não-supervisionadas o objetivo é o de se encontrar padrões de similaridades e diferenças, o RL visa encontrar um modelo e um método para que sua recompensa em um determinado ambiente seja maximizada.

Para a compreensão de como essa técnica funciona, alguns conceitos devem ser estabelecidos e entendidos. Eles são listados no site [9]:

- Agente: A entidade que está em um ambiente e realiza ações para ganhar recompensas
- Ação: Um movimento que o agente pode realizar
- Ambiente: Cenário em que o agente se encontra
- Estado: Situação atual em que o agente se encontra dado o ambiente
- Recompensa: Valor que representa o ganho imediato de uma ação realizada pelo agente no ambiente dado o estado
- Política: Estratégia utilizada pelo agente para determinar qual ação tomar baseada no estado atual

- Valor (value): Esperança a longo-prazo com fator de desconto do retorno ou recompensa do agente em um determinado estado seguindo uma determinada política.
- Q-value: Similar ao Value, mas leva em consideração a ação tomada. É o retorno a longo-prazo de se tomar uma determinada ação em um determinado estado seguindo uma determinada política

Ainda no site [9], há 3 formas muito utilizadas para se implementar um algoritmo de RL:

- Baseado em Valor: Algoritmo que tenta maximizar uma função valor V(s), para assim poder maximizar a recompensa a longo prazo obtida pelo agente.
- Baseado em Política: Neste tipo de algoritmo, precisa-se encontrar uma política tal que a ação que o agente irá tomar a cada estado seja ótima para se maximizar a recompensa.
- Baseado em Modelo: Cria-se um modelo virtual para cada ambiente e o agente aprende a atuar naquele determinado ambiente.

Diversos algoritmos e abordagens para se obter uma forma de ML através de aprendizado por reforço existem. A seguir serão abordados e apresentados os conceitos de *Markov Decision Process* (MDP) e Política, que são fundamentais para se poder modelar e entender diversos dos diferentes algoritmos utilizados.

3.1.1 MARKOV DECISION PROCESS

Em [10], pode-se ter uma boa explicação para se entender sobre o MDP, como resumido e discorrido a seguir:

O MDP é o processo utilizado em diversos algoritmos de RL.

Para se poder entender como chegar no MDP, alguns conceitos são apresentados que auxiliarão a entender algumas de suas propriedades:

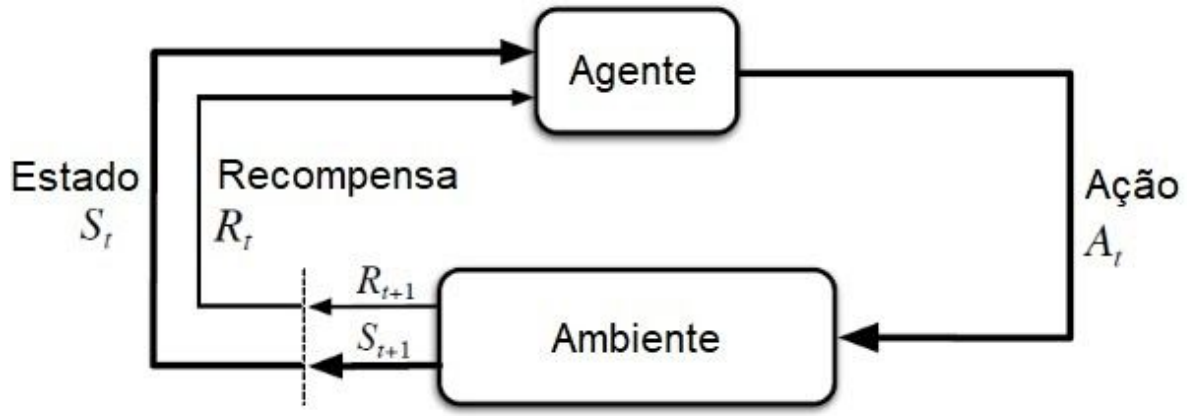
- **Markov Property**: segundo essa propriedade, dado um estado atual, não é necessário um histórico ou registro de informações de momentos anteriores para dizer e caracterizar o que acontecerá no futuro
- **Markov Process**: é um processo sem memória randômico (por exemplo uma sequência aleatória de estados S1, S2, ...) que tem a propriedade Markov. Faz uso de uma tupla (S, P), na qual S é o espaço de estados e P representa a probabilidade de se transicionar de um estado s para um s', ou seja, função de transição.
- **Markov Reward Process**: é um Markov Process que apresenta também o total de recompensas acumuladas ao longo da sequência de estados tomada. É a tupla (S, P, R, γ), na qual S é o espaço finito de estados, P é a função que apresenta a probabilidade de transição de cada estado, R é a função recompensa, que mostra qual a recompensa imediata espera-se conseguir no estado S no dado momento e γ (γ ∈ [0, 1]) é o fator de desconto, que informa ao agente quanto ele deve se importar com uma recompensa agora em detrimento de recompensas no futuro. Se γ é 0, então o agente só se importa com recompensas imediatas, se é 1, o agente tem visão a longo prazo e se importa com recompensas futuras. Esse fator de desconto de recompensas é necessário por exemplo para se garantir que as recompensas tenham seu valor reduzido, garantindo que o algoritmo irá convergir e evitará loops infinitos em certos processos Markov. Outro motivo para se incluir esse fator de desconto é o de que no caso em que o agente não sabe exatamente como o futuro será, talvez a melhor decisão seja a de se optar por pegar as melhores recompensas agora do que apostar que poderá pegar uma maior em um momento no futuro. Dessa forma, γ iria estabelecer a preferência do agente em focar em coletar recompensas de curto ou longo prazo.
- **Equação de Bellman**: Como já dito, a função valor irá fornecer a esperança das recompensas que serão obtidas pelo agente ao se realizar uma ação em determinado estado. O objetivo final do agente é o de maximizar a recompensa

final obtida. Para tanto, deve-se tentar obter a função valor de maior valor através do uso da Equação de Bellman, que pode ser entendida com mais detalhes em [11]. Ela irá prover uma função para o valor da esperança da recompensa em cada estado. Essa esperança leva em conta em suas variáveis a esperança da recompensa do próximo estado que o agente estará ao tomar dada ação a partir do estado atual além da recompensa recebida pela sua ação, possibilitando que se possa estimar qual a melhor maneira do agente tentar maximizar suas recompensas.

Finalmente, tendo em vista os conceitos apresentados, pode-se construir um MDP, que é um Markov Reward Process com decisões em um ambiente em que todos os estados são Markov. É o processo base utilizado por métodos e algoritmos de RL para que o agente possa definir e realizar a ação a ser tomada em cada estado de forma a maximizar a recompensa. Ele irá modelar a escolha e mecanismo de cada decisão tomada pelo agente ao permitir representar o ambiente e as possíveis variáveis a serem levadas em consideração pelo algoritmo de RL. Para isso, faz uso de um conjunto que consiste no conjunto finito S de estados que existem no dado ambiente, o conjunto A de todas as ações que podem ser tomadas em cada estado, uma função recompensa R, o modelo de transição P(s', s | a) que indica qual é a probabilidade de se realizar a ação a que irá realizar uma transição do estado s para o s' e γ, que é o fator de desconto da recompensa.

Ou seja, o MDP faz uso da tupla M = (S, A, P, R, γ) como mostra a Figura 2.

Figura 2 - Processo de Decisão de Markov



Fonte: Sutton; Barto (2018), traduzido [12].

Todavia, vale notar que a cada estado possível o agente precisa decidir qual ação ele irá tomar a seguir, que irá dar continuidade no algoritmo e ambiente. Para tomar essa decisão, faz uso da sua política.

3.1.2. POLÍTICA

Em [13] é possível se ter uma boa introdução para começar a se entender a ideia de política, que pode ser entendida mais a fundo e melhor juntamente com [14], como resumido e discorrido a seguir:

Política π ou policy é a estratégia que será utilizada pelo agente para determinar sua próxima ação baseado no estado atual. Ela irá tentar definir as ações que devem ser tomadas pelo agente de forma a maximizar suas recompensas obtidas, como já mencionado. Vale notar que a política depende apenas do estado atual, não do histórico de ações tomadas pelo agente até determinado momento.

Todavia, em um ambiente nem sempre é fácil de se determinar qual vai ser a ação a cada estado que irá maximizar a esperança de recompensas, uma vez que o agente pode estar inserido em um ambiente que possui muitas variáveis que irão afetar esse cálculo. Por isso, a política utilizada por um algoritmo deve ser atualizada e refinada até que se atinja uma política ótima estável que não precisa ou é pouco modificada nos

processos de refinamento. Com a política ótima o agente vai poder saber quais ações tomar que trarão a ele os melhores resultados.

Mas existe um possível problema no conceito de se utilizar políticas. Seu conceito diz que a política irá ditar qual deve ser a ação tomada por um agente em determinado estado de tal forma a se maximizar a recompensa esperada. Mas sabe-se também que o agente está inserido em um ambiente cheio de incertezas, de tal forma que é difícil saber a priori o resultado de cada ação tomada em cada estado. Assim, como é possível saber que a ação ditada pela política é de fato a melhor e que não há nenhuma outra ação que dará uma recompensa maior? Seguindo-se sempre as mesmas ações para cada estado, como o agente irá descobrir novos estados? Se o agente sempre seguir as ações já descobertas pela sua política, nunca irá conseguir explorar novas possibilidades, descobrir novos estados e recompensas que podem levar a resultados melhores que os já registrados. Por isso, deve-se executar cada iteração de cada episódio (no contexto de um jogo, um episódio pode ser por exemplo o início de uma partida até o seu final) de maneira significativa para se poder extrair a política ótima (uma política que já atingiu um nível de refinamento que não pode mais ser melhorado substancialmente, o que permite que o agente obtenha as melhores recompensas possíveis), de tal forma que o agente irá tomar as ações já conhecidas como melhores de tal forma a maximizar as recompensas obtidas (processo conhecido como exploitation) mas também irá tentar explorar o ambiente através de ações randômicas (processo conhecido como exploration). Para isso, adota-se o que é chamado de política ε-greedy. [9] discorre sobre os conceitos de exploitation e exploration, bem como sobre essa política.

Numa política ε-greedy o agente irá seguir a política greedy 1 - ε parte das vezes, seguindo a melhor ação conhecida até o momento, e tomar ações randômicas ε parte das vezes. Esse tipo de mecanismo permite ao agente descobrir novos estados e possíveis ações parte do tempo enquanto se aproveita das melhores ações já registradas na outra parte do tempo.

Uma vez que esses dois conceitos fundamentais foram apresentados, podem ser apresentados alguns métodos que são utilizados para implementar as ideias de ML através de RL

3.1.3. MONTE CARLO

Em [15] e [16] é possível se obter um bom conhecimento e entendimento sobre Monte Carlo, como resumido e discorrido a seguir:

Monte Carlo é um método utilizado em que o agente irá aprender os estados e recompensas ao interagir com o ambiente. O agente estará sujeito a vários cenários de iterações. Cada iteração será referida como episódio. No contexto do nosso projeto, um episódio seria um dado momento inicial até o final do round, por exemplo. Esse momento inicial não necessariamente é o início do round, ele pode ser qualquer momento desde o início até antes do final. A cada episódio o algoritmo irá calcular valores de esperanças das recompensas para cada estado ou estado-ação, que servirão para atualizar e refinar o comportamento do agente sob determinada política ao modificar as esperanças da recompensa de cada estado ou estado-ação.

Uma das vantagens desse método é que ele é simples de se entender e utilizar, além de ter boas propriedades de convergência.

Todavia, seus problemas incluem o fato de que o método e algoritmos não garantem que todos os estados serão visitados, exige que o ambiente possa funcionar e ser categorizado em episódios, o agente apenas irá aprender após a finalização dos episódios (o que seria um problema em jogos que não acabam) e o episódio precisa ser concluído para que o agente tenha algum progresso no refinamento de seus parâmetros.

3.1.4. TEMPORAL DIFFERENCE LEARNING

Fica claro observando-se o método Monte Carlo que um dos problemas dele é que o agente começa a refinar e aprimorar seu comportamento/ações apenas no final de cada episódio. Às vezes pode ser proveitoso que o esse refinamento já seja aprimorado a cada instante dele ao invés de só no final, obtendo-se assim maior rendimento e taxas maiores de aprimoramento por iteração, uma vez que o agente não teria seu comportamento ajustado a cada episódio que deveria ser seguido de um outro para gerar mais aprimoramentos, mas constantemente, o que poderia fazer que seu comportamento ficasse mais próximo do ótimo rapidamente.
Utilizando essa ideia existe o método *Temporal Difference Learning* (TDL), que realiza as atualizações nas esperanças a cada instante/ação do agente em um episódio [16].

3.1.5. Q-LEARNING

Em [17] e [18] é possível se obter um bom conhecimento e entendimento sobre Q-Learning, que é uma das implementações utilizando o método TDL [19], como resumido e discorrido a seguir:
Q-Learning é um algoritmo de ML por RL que visa encontrar a melhor ação a se tomar em determinado momento ao fazer uso de políticas ε-greedy e utilizar as ideias do método de TDL ao atualizar suas tabelas internas com valores de esperanças de recompensas a cada ação.
O “Q” em “Q Learning” vem de Quality, que nesse caso representa quão útil uma ação é para se conseguir uma recompensa no futuro.
O algoritmo faz uso de uma tabela chamada Q-Table que possui em suas linhas cada possível estado em suas colunas as possíveis ações a serem tomadas (ou vice-versa). Cada combinação de linha e coluna irá corresponder à um valor que será atualizado a cada ação do agente. Esse valor é justamente a esperança da recompensa que

determinada ação em determinado estado tem. Essa tabela é a que será utilizada pelo agente para que possa selecionar a melhor ação que irá maximizar a recompensa.

Inicialmente o agente não aprenderá muito com apenas um episódio, mas a cada iteração o algoritmo e valores da Q-Table irão cada vez mais convergir e ele irá atingir uma política ótima ou perto da ótima.

Com os conceitos apresentados, pode-se partir para uma implementação um pouco mais concreta do algoritmo:

Primeiramente, inicializa-se a Q-Table com os valores em 0. Ela será uma matriz cujas linhas representam cada estado possível e cujas colunas representam as possíveis ações em cada estado. Os valores que estão armazenados em cada elemento da matriz são a esperança da recompensa que, como já dito, serão atualizados a cada ação do agente.

A cada ação, o agente tem duas opções para interagir com o ambiente, uma vez que está sendo usada uma política ε-greedy: pode escolher utilizar a tabela e selecionar a ação que segundo ela irá trazer maior benefício (exploiting) ou pode agir de forma mais aleatória (exploring).

Vale notar que como a Q-Table começa inicialmente com os seus valores zerados, no começo do algoritmo, o agente precisa realizar ações de exploring, por isso o valor de ε deve ser ajustado para que isso ocorra. Conforme a matriz vai ficando cada vez mais com os valores estabilizados, ε deve ir mudando para balancear o exploiting e o exploring tanto quanto se queira.

Independente do que o agente fizer, após a ação ser finalizada, o agente irá receber a recompensa de sua ação e a matriz será atualizada com um novo valor para a ação tomada no estado em que estava. Esse valor será calculado de acordo com a Q-Function, que faz uso da Equação de Bellman juntamente com dois inputs (estado e ação). Uma das formas de implementar a função que irá calcular o novo valor de Q para atualizar a tabela pode ser encontrado em [18] e é:

$$Novo\ Q(s,a) = Q(s,a) + \alpha * [R(s,a) + \gamma * maxQ'(s',a') - Q(s,a)]$$

Na qual:

- $Novo\ Q(s,a)$ é o novo valor de Q calculado para dado estado e ação
- $Q(s,a)$ o valor atual de Q para dado estado e ação
- α é a taxa de aprendizado
- $R(s,a)$ é a recompensa recebida da ação para aquele estado
- γ é o fator de desconto
- $maxQ'(s',a')$ é a maior esperança de recompensa no futuro dado o novo estado s' e todas as possíveis ações desse novo estado

A taxa de aprendizado α é um fator que irá determinar o quanto esse novo valor será aceito em detrimento do antigo [17]. Dependendo do estado da Q-Table, deve-se dar mais ou menos peso à novos valores calculados: Um agente que acabou de começar a popular a matriz com valores calculados deve aceitar amplamente valores novos, já que possui apenas valores default, enquanto um agente que já está com valores estabilizados e possui uma política ótima ou perto da ótima provavelmente não deve ter seus valores atuais (que são ótimos) muito afetados por novos que podem possivelmente afetar negativamente o progresso do algoritmo ao modificá-los de forma errônea.

Nota-se que para que o cálculo seja realizado e para que o algoritmo possa ser usado, os valores das recompensas para cada ação em cada estado devem ser conhecidos e pré-estabelecidos.

Com isso, é possível atualizar os valores da Q-Table durante diversos episódios até que seus valores comecem a estabilizar e o algoritmo convirja.

3.1.6 SARSA

Em [19] e [20] é possível se obter um bom conhecimento e entendimento sobre o SARSA, como resumido e discorrido a seguir:

O SARSA (State Action Reward State Action) é um algoritmo muito semelhante ao Q-Leaning com uma pequena variação.

A política dos algoritmos de ML via RL podem ser classificados da seguinte maneira:

On Policy: O agente aprende a função valor de acordo com a ação que veio da política sendo empregada

Off Policy: O agente aprende a função valor de acordo com a ação que veio de uma outra política

Viu-se que o Q-Learning utiliza a política greedy para calcular os Q-Values. Por isso é classificado como Off Policy. O SARSA é um algoritmo On Policy e utiliza a ação que veio da política sendo utilizada para aprender o Q-Value.

A diferença é, portanto, a seguinte [20]:

1. Q-Learning:

$$Novo\ Q(s,a) = Q(s,a) + \alpha * [R(s,a) + \gamma * maxQ'(s',a') - Q(s,a)]$$

2. SARSA:

$$Novo\ Q(s,a) = Q(s,a) + \alpha * [R(s,a) + \gamma * Q'(s',a') - Q(s,a)]$$

Ao contrário do Q-Learning tradicional, que não apresenta restrições para a próxima ação, enquanto ela maximiza o Q-Value para o próximo estado, o SARSA sempre segue a política.

Além disso, o valor de Q no SARSA depende do estado atual, da ação atual, da recompensa, do próximo estado e da próxima ação. Por isso recebe o nome de SARSA, que simboliza a tupla (s,a,r,s',a')

A utilização do SARSA é conveniente pois é com o SARSA que podemos aplicar um método matemático que auxilia na performance de métodos Temporal Difference: o Eligibility Traces.

3.1.7 SARSA LAMBDA

Em [21] é possível se obter uma boa explicação para se entender sobre o conceito de Eligibility Traces, bem como também o Sarsa Lambda, um algoritmo que o utiliza, como resumido e discorrido a seguir:

A ideia de refinar mais ainda a performance em métodos Temporal Difference através de Eligibility Traces pode ser entendida através do seguinte raciocínio:

- Um agente está andando de forma randômica por um ambiente quando depois de n passos encontra um grande tesouro que computa no seu algoritmo uma enorme recompensa.

Seria interessante que fossem estudados quais os passos e o processo que o levou a obter tais resultados. Faz sentido atribuir maior importância aos passos mais recentes e que podem ter sido decisivos na descoberta do que os que foram tomados há muitas iterações passadas, que talvez nem tivessem tido tanto impacto na descoberta do tesouro.

De forma análoga, temos outra situação, como a levantada em [3]:

- Um piloto autônomo deve aprender a percorrer uma pista. Ele vem acelerando por um trecho da pista e se depara com uma curva. Nela, o agente (após realizar consultas em suas tabelas internas) descobre que precisa diminuir sua velocidade para realizá-la e começa então a operação de frenagem. Todavia, sua velocidade estava tão alta que a frenagem não foi o suficiente para que a curva pudesse ser feita e ele acaba saindo da pista, computando em seu algoritmo uma recompensa muito negativa.

Nesse cenário, seria interessante penalizar não a frenagem apenas, que na verdade foi uma ação certa do agente, mas sim a aceleração excessiva que levou o agente a estar com uma velocidade muito alta, impossibilitando que a operação de frenagem fosse suficiente para que a curva fosse feita com sucesso.

Essa é uma das vantagens de se utilizar essa técnica: a de atribuir pesos que serão utilizados para atribuir uma recompensa a ações passadas do agente. Esses pesos são maiores para as ações mais recentes e vão decrescendo de tal forma que as ações mais antigas têm peso muito menor.

Para essa ideia se materializar, é utilizado um vetor E chamado Eligibility Traces. Esse vetor terá a função de guardar o valor do peso que será utilizado para determinar a recompensa de cada estado de forma que ele irá diminuir de forma exponencial a cada

iteração, refletindo o fato de que ações mais distantes da atual têm relevância menor na recompensa da ação atual do agente.

Para explicar de forma mais clara como o Eligibility Traces é utilizado em um exemplo muito básico: (o grupo apresenta o seguinte exemplo):

- Suponha que um agente se move em um tabuleiro T 3x3. Ele pode a cada ação ir para qualquer casa da matriz e receberá uma recompensa arbitrária em cada posição.
- Cada posição do tabuleiro em que o agente escolher irá representar um possível estado do agente no ambiente em que se encontra.
- Inicialmente o vetor E Eligibility Traces, que irá mapear os 9 possíveis estados (que na prática serão cada casa do tabuleiro), tem todos os valores zerados.
- O vetor E terá seus pesos multiplicados por um fator de 0.9 a cada iteração, assim, E'[i,j] = E[i,j] * 0.9
- Para simplicidade, o cálculo do valor de Q'[i,j] após a ação para aquele estado será dado por Q'[i,j] = Q[i,j] + R * E[i,j]
- Inicialmente todos os valores de Q são 0

Seguindo as iterações do agente sobre o ambiente:

- Agente se move para T[2,2] e a recompensa daquela casa/estado é R = 10
    - Para a casa T[2,2]:
        - E[2,2] = E[2,2] + 1 = 0 + 1 = 1
        - Q'[2,2] = Q[2,2] + R * E[2,2] = 0 + 10 * 1 = 10
    - No final da iteração:
        - E[2,2] = 1
        - Q[2,2] = 10
- Agente se move para T[0,0] e a recompensa daquela casa/estado é R = 15
    - Para a casa T[0,0]:
        - E[0,0] = E[0,0] + 1 = 0 + 1 = 1
        - Q'[0,0] = Q[0,0] + R * E[0,0] = 0 + 15 * 1 = 15
    - Para a casa T[2,2]:

        - E’[2,2] = E[2,2] * 0.9 = 1 * 0.9 = 0.9
        - Q’[2,2] = Q[2,2] + R * E[2,2] = 10 + 15 * 0.9 = 23.5
    - No final da iteração:
        - E[0,0] = 1, E[2,2] = 0.9
        - Q[0,0] = 15, Q[2,2] = 23.5
- Agente se move para T[1,1] e a recompensa daquela casa/estado é R = -5
    - Para a casa T[1,1]:
        - E[1,1] = E[1,1] + 1 = 0 + 1 = 1
        - Q’[1,1] = Q[1,1] + R * E[1,1] = 0 + (-5) * 1 = -5
    - Para a casa T[0,0] :
        - E’[0,0] = E[0,0] * 0.9 = 1 * 0.9 = 0.9
        - Q’[0,0] = Q[0,0] + R * E[0,0] = 15 + (-5) * 0.9 = 10.5
    - Para a casa T[2,2]:
        - E’[2,2] = E[2,2] * 0.9 = 0.9 * 0.9 = 0.81
        - Q’[2,2] = Q[2,2] + R * E[2,2] = 23.5 + (-5) * 0.81 = 19.45
    - No final da iteração:
        - E[0,0] = 0.9, E[1,1] = 1, E[2,2] = 0.81
        - Q[0,0] = 10.5, Q[1,1] = -5, Q[2,2] = 19.45

Com esse exemplo básico, observa-se como o vetor E cumpre a ideia de amplificar a recompensa de ações mais recentes com um peso cada vez maior para ações mais recentes, ao passo que ações mais antigas irão ter seu peso tão baixo que essas amplificações podem chegar a ser até desprezíveis dependendo de quão no passado elas estão.

Vale notar que o estado atual sempre tem seu peso no vetor E atualizado com o valor atual do peso somado a 1. Assim, a recompensa relativa àquele estado e ação irá ser computada no mínimo integralmente no algoritmo.

O peso dos valores de E caem exponencialmente a uma determinada taxa. Essa taxa é o λ ($\lambda \in [0, 1]$).

3.1.7.1 PSEUDO-CÓDIGO

A seguir está o pseudo-código para a implementação do Algoritmo Sarsa(λ):

1. Inicializar Q(s,a) e E(s,a) = 0 para todos s,a
2. Repetir para cada passo do episódio:
3. Realizar ação a, observar r, s'
4. Escolher a' dado s' utilizando política adotada
5. δ = r + γQ(s',a') - Q(s,a) #Necessário para atualizar valores de Q(s,a)
6. E(s,a) = E(s,a) + 1
7. Para todos s,a:
8. Q(s,a) = Q(s,a) + αδE(s,a) #Atualizar valores de Q(s,a) utilizando eligibility traces
9. E(s,a) = γλE(s,a) #Decaimento dos pesos em E
10. s = s'; a = a'
11. Até s ser estado final

No qual:

- s representa o estado atual
- a representa a ação tomada
- r representa a recompensa obtida com a ação a no estado s
- Q(s,a) é o valor atual de Q para dado estado e ação
- E(s,a) é o valor do peso no vetor Eligibility Traces E para dado estado e ação
- α é a taxa de aprendizado
- γ é o fator de desconto
- λ é uma das taxas que serão utilizadas para o decaimento dos valores de E. Notar que nesse algoritmo o decaimento dos valores não depende exclusivamente de λ.

Para se entender o δ, é retomada a equação de atualização dos valores de Q(s,a) do Sarsa:

$$Novo\ Q(s,a) = Q(s,a) + \alpha * [R(s,a) + \gamma * Q'(s',a') - Q(s,a)]$$

No pseudo-código proposto, temos também as seguintes definições nas linhas 5 e 8:

$$\delta = r + \gamma Q(s',a') - Q(s,a)$$

e

$$Q(s,a) = Q(s,a) + \alpha\delta E(s,a)$$

Pode-se juntar as duas definições em:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))E(s,a)$$

Comparando a equação obtida com a equação de atualização dos valores de Q(s,a) do Sarsa, observa-se que ela pode ser entendida como uma forma análoga da atualização dos valores de Q(s,a) do Sarsa com a utilização dos pesos presentes no vetor Eligibility Traces.

3.2 INVERSE REINFORCEMENT LEARNING

O problema do IRL pode ser caracterizado como, segundo [22]:

Dado:

- Medidas do comportamento do agente ao longo do tempo em uma variedade de circunstâncias;
- Se necessário, medidas das entradas sensoriais do agente;
- Se disponível, o modelo do ambiente.

Determine a função recompensa sendo otimizada.

Existem duas motivações identificadas por [5] para este tipo de problema. A primeira vem do potencial uso de RL para modelar o aprendizado humano e animal em que as literaturas citadas pelo trabalho assumem que a função recompensa é fixa e conhecida. Entretanto, a priori, devemos considerar que a função recompensa que leva o comportamento humano e animal é desconhecida e deve ser confirmada através de investigação empírica, principalmente quando existem diversos fatores que afetam a recompensa obtida.

A segunda motivação surge da tarefa de construir um agente inteligente que se comporte de maneira ótima em determinado domínio. O designer pode ter apenas uma vaga ideia da função recompensa que vai gerar o comportamento esperado, pense na tarefa de “dirigir bem”.

3.2.1 APPRENTICESHIP LEARNING

O trabalho [6] afirma que frequentemente é difícil especificar manualmente uma função recompensa capaz de apresentar o comportamento desejado ao apresentar o exemplo de dirigir. O motorista leva em consideração diferentes fatores externos enquanto dirige, como distância do veículo da frente, velocidade atual, posição dos outros veículos, sinalização e, muitas vezes, executa um trade-off para alcançar um objetivo como frear para permitir o pedestre atravessar a rua sem ter obrigações legais de tomar esta atitude (ex. ruas sem semáforo). Com isso em mente, os autores de [6] afirmam que, apesar de serem bons motoristas, eles não são capazes de definir com confiança uma função recompensa que resulte na tarefa de “dirigir bem”. Isso significa que na prática, a função recompensa é manualmente refinada até alcançar o comportamento desejado, assim os autores de [6] acreditam que esta dificuldade de especificar a função recompensa manualmente é o que impede a aplicabilidade extensa do RL.

Ainda no exemplo de dirigir, é muito mais fácil ensinar um novo motorista a dirigir mostrando como deve ser feito ao invés de dizer qual a função de recompensa tomada. Essa tarefa de aprender com um especialista é chamada de *Apprenticeship Learning* que busca um comportamento tão bom quanto o demonstrado ao tentar aproximar uma função recompensa que o especialista intrinsecamente levou em conta.

3.2.1.1 DEFINIÇÕES

Abbeel e Ng [6] adotam as seguintes notações:

- $MDP \setminus R$, MDP sem a função recompensa, a tupla (S, A, P, γ, D);
- ϕ, vetor de atributos de estado. $S \rightarrow [0,1]^k$;
- w^*, vetor de pesos "verdadeiro", onde $w^* \in \mathbb{R}^k$;
- R^*, função recompensa "verdadeira", onde $R^*(s) = w^* \times \phi(s)$;
- μ, atributos esperados, nome sucinto para o vetor de atributos acumulados, descontados e esperados, onde $\mu(\pi) = E[\sum_{t=0}^{\infty} \gamma^t \times \phi(s) \mid \pi] \in \mathbb{R}^k$;
- π_E, demonstrações do especialista;
- μ_E, atributos esperados do especialista, onde $\mu_E = \mu(\pi_E)$;
- $\hat{\mu_E}$, estimativa empírica de μ_E dado um conjunto de m trajetórias $\{s_o^{(i)}, s_1^{(i)}, \ldots\}^m{}_{i=1}$, onde $\hat{\mu_E} = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{\infty} \gamma^t \times \phi(s_t^{(i)})$;
- ε, valor mínimo do módulo da diferença entre μ_E e μ para considerar que o algoritmo convergiu e parar a execução.

Além disso, outras considerações são tomadas:

- $\|w^*\|_1 \leq 1$, para garantir que as recompensas sejam limitadas em 1.

3.2.1.2 ALGORITMO

Messer [7] apresenta um pseudocódigo mais próximo da implementação do algoritmo de Abbeel e Ng [6], como segue:

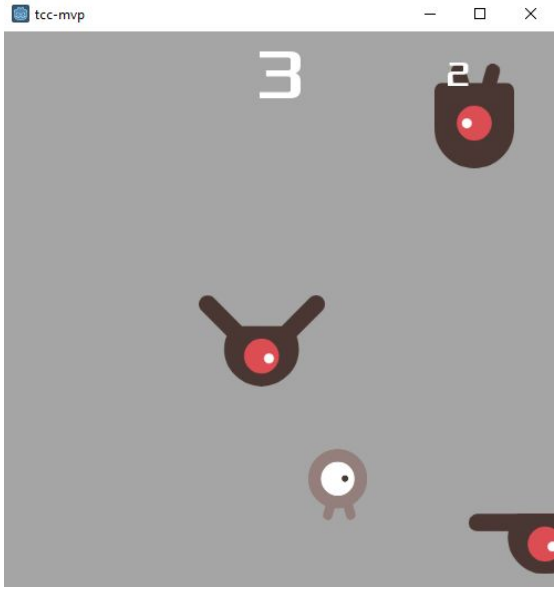
1. Escolha aleatoriamente uma política $\pi^{(0)}$, calcule ou aproxime $\mu^{(0)} = \mu(\pi^{(0)})$;
2. Faça $i = 1$;
3. Faça $w^{(1)} = \mu_E - \mu^{(0)}$;
4. Faça $\overline{\mu}^{(0)} = \mu^{(0)}$;
5. Faça $t^{(1)} = \|w^{(1)}\|_2$;
6. **if** $t^{(1)} \leq \varepsilon$ **then**:
7. Termine;
8. **else**

9. **loop**(enquanto $t^{(i)} > \varepsilon$);
10. Usando algoritmos RL, calcule a política ótima $\pi^{(i)}$ usando $R = (w^{(i)})^T \times \phi$
11. Calcule $\mu^{(i)} = \mu(\pi^{(i)})$;
12. Faça $i = i + 1$;
13. Faça $a = \mu^{(i-1)} - \overline{\mu}^{(i-2)}$;
14. Faça $b = \mu_E - \overline{\mu}^{(i-2)}$;
15. Faça $\overline{\mu}^{(i-1)} = \overline{\mu}^{(i-2)} + \frac{a^T b}{a^T a} a$;
16. Faça $w^{(i)} = \mu_E - \overline{\mu}^{(i-1)}$;
17. Faça $t^{(i)} = ||w^{(i)}||_2$
18. **end loop**
19. **end if**

4 PROTÓTIPOS INICIAIS

Com o objetivo de validar a ideia do projeto e encontrar pontos de atenção na modelagem do problema proposto, foi criado um jogo onde o agente pode se movimentar livremente pelo espaço disponível com outros agentes (inimigos) também se movimentando neste espaço, entretanto estes surgem apenas da parte superior do mapa e percorrem em linha reta até a parte inferior, com limite de 3 inimigos visíveis. Para vencer a partida, o agente principal deve desviar dos outros por 30 segundos, caso ocorra uma colisão neste intervalo de tempo ele perde. Por fim, o agente principal tem apenas 5 ações disponíveis, sendo estas a movimentação em x (esquerda, direita), y (cima, baixo) e permanecer parado. A Figura 3 apresenta o jogo.

Figura 3 - Jogo utilizado para testar os algoritmos de RL.



Fonte: “Your first game” Godot[4], modificado.

A escolha deste jogo foi baseada na simplicidade e apresentação de características semelhantes ao que vai ser abordado neste trabalho, como a movimentação livre e o cuidado com outros agentes no mapa.

[4] GODOT™. Godot Game Engine. 2020. Disponível em: <https://godotengine.org/>

4.1 MODELAGEM INICIAL

A primeira versão do jogo tinha como objetivo verificar se o algoritmo de RL era adequado para as características citadas anteriormente. No caso foi utilizado o Q-Learning.

A principal preocupação da modelagem era o agente principal ser capaz de detectar todos os inimigos ao redor com uma precisão razoável, para isso o mapa foi dividido em 256 regiões iguais (16x16 blocos). Assim, cada agente tem sua posição (x,y) e o conjunto de posições deles representam o estado do jogo.

Desta modelagem surge o primeiro problema, no total existem 4 agentes (o principal e o limite de 3 inimigos) que resulta em $(16x16)^4 = 4,294,967,296$ possibilidades de estado e o agente principal é capaz de tomar 5 ações diferentes. Como a Q-Table armazena o valor de cada ação em determinado estado, esta tabela necessita de $n_{ações} \times m_{estados}$ entradas que neste caso é de aproximadamente 20 bilhões. Considerando que cada entrada seja do tipo float 32 bits, a memória necessária para armazenar esta tabela seria de aproximadamente 80 GB.

Esta quantidade de memória necessária é inviável no contexto do projeto, assim pode-se levar como aprendizado o compromisso entre a precisão com que o agente observa o ambiente e a quantidade de estados resultante.

4.2 REDUÇÃO DO NÚMERO DE ESTADOS

Levando-se em consideração a lição aprendida do primeiro modelo, reduziu-se a divisão do mapa para 100 regiões iguais (10x10 blocos), resultando em $(10x10)^4 = 100,000,000$ possibilidade de estados que significa aproximadamente 2 GB de memória para o armazenamento do Q-Table seguindo o mesmo raciocínio anterior. Em termos de memória é um número aceitável ao levar em conta que os computadores atuais podem ter de 4 GB a 32 GB de memória RAM.

Vale lembrar que quanto maior a quantidade de entradas na Q-Table, maior o tempo de convergência para a política ótima, pois, idealmente é interessante que sejam visitadas todas as ações em cada estado para existirem valores a serem comparados, o exploration citado na parte teórica. Além disso, o algoritmo de IRL executa diversas iterações do RL para convergir ao comportamento do expert, o que multiplica o tempo de convergência do algoritmo final.

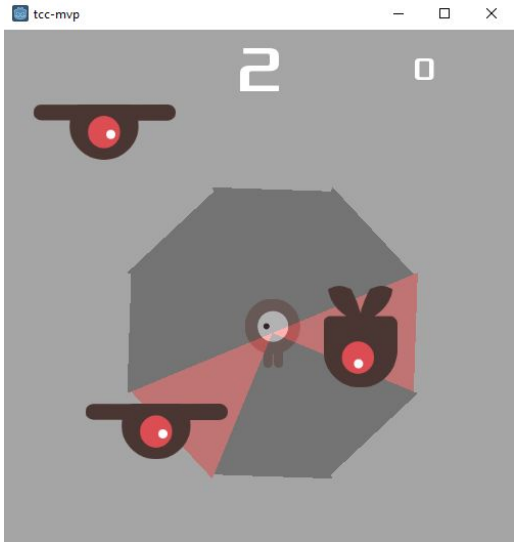
Na aplicação que este trabalho propõe, o tempo de convergência do algoritmo impacta na usabilidade do jogo no momento em que o jogador define o comportamento esperado e queira utilizá-lo em sua próxima partida.

Assim, no trade-off citado na modelagem inicial, também deve ser considerado o tempo de convergência resultante do número de estados e ações possíveis.

4.3 SENSOR DE PROXIMIDADE

Para reduzir o número de estados, adotou-se uma modelagem baseada em sensores de proximidade em que, ao invés do agente saber exatamente a posição dos inimigos, ele sabe sua posição atual e se existe algum inimigo perto através dos 8 sensores que abrangem 45º graus cada ao redor do agente principal. A Figura 4 apresenta a disposição dos sensores, caso exista um inimigo próximo o sensor corresponde muda para a cor vermelha.

Figura 4 - Disposição dos sensores



Fonte: “Your first game” Godot, modificado.

Nesta modelagem, reduziu-se a divisão do mapa para 9 regiões (3x3 blocos), cada sensor tem 2 valores possíveis e o estado do jogo visível para o algoritmo só possui a posição do agente principal, visto que a posição dos inimigos vai ser contemplada pelos sensores de proximidade. Essa mudança tem como resultado $9 \times 2^8 = 2,304$ possibilidades de estado.

4.4 MUDANÇA DE ALGORITMO

Com a grande redução do número de estados, esperava-se que a convergência do algoritmo ocorresse mais rapidamente para um comportamento ótimo, entretanto notava-se que as partidas em que o agente alcançasse a vitória eram raras e aparentava ser por sorte.

A equipe entendeu isso como uma falha na escolha de algoritmo para o problema a ser resolvido, pois a recompensa é dada para a última ação tomada, assim caso o agente colida com o inimigo e perca o jogo, apenas a última ação vai ser penalizada, entretanto pode ser que esta ação tenha sido na tentativa de sair da posição desfavorável para desviar do inimigo e não houve tempo hábil para isto ocorrer,

penalizando assim a ação que de fato teria sido a ideal ao invés de penalizar as anteriores que a levaram para este tipo de situação.

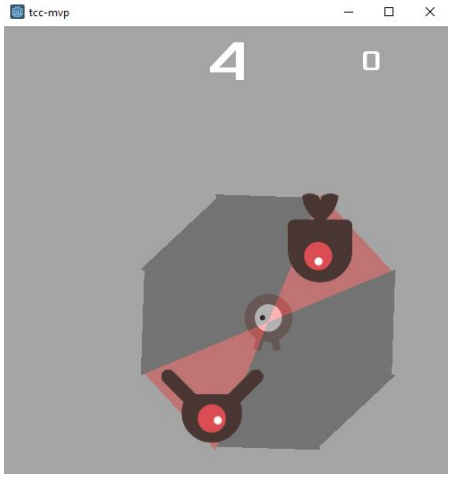
Um problema semelhante a este pode ser encontrado em [3] no simulador de corrida TORCS quando o carro entra em uma curva e acaba saindo da pista por não conseguir realizá-la corretamente. Pode ser que a última ação tomada tenha sido frear, mas ainda assim a velocidade do carro estava muito alta para realizar a curva, então o autor utiliza o conceito de traços de elegibilidade para penalizar não somente a última ação, mas sim todas as ações que levaram àquela situação.

Com isso em mente, trocou-se o algoritmo de RL para o SARSA Lambda que aplica os conceitos de traço de elegibilidade.

4.5 INFORMAÇÃO DOS SENSORES

A mudança de algoritmo trouxe resultados promissores em que o agente estava sendo capaz de sobreviver pelo tempo máximo com maior frequência. Uma situação específica em que o agente acabava não sabendo o que fazer é ilustrada pela Figura 5.

Figura 5. Situação confusa para o agente

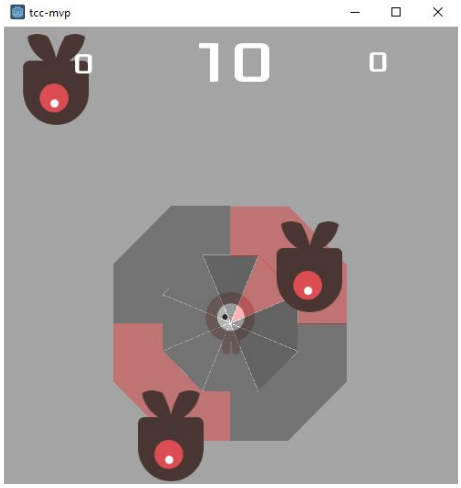


Fonte: “Your first game” Godot, modificado.

Em situações similares à esta, em que o agente “vê” os inimigos em lados opostos (ex. cima e baixo, superior-direita e inferior-esquerda) ele não tem informações suficientes sobre o quão perto eles estão para saber se é possível ir para um dos lados sem que ocorra a colisão. Isso acontece porque os sensores trazem apenas a ideia de estar perto ou não.

Para minimizar isso, foram incluídos uma nova volta de sensores para trazer a ideia de “perto” e “muito perto” para o agente, como mostra a Figura 6.

Figura 6. Novos sensores de proximidade



Fonte: “Your first game” Godot, modificado.

Agora, em situações como essa, o agente tem informações suficientes para decidir qual ação tomar.

5 ESPECIFICAÇÕES DO PROJETO

Após terem sido explicados e expostos os conceitos teóricos que serão empregados, serão discorridos aqui o game design, requisitos e detalhes do projeto e do que se espera dele.

5.1 GAME DESIGN

Como já dito, o projeto visa aprimorar o comportamento das peças de jogos do gênero AutoChess ao permitir que o próprio jogador defina seus comportamentos. Isso será realizado através da utilização de algoritmos de ML. Uma vez que o jogador o fizer, poderá utilizar as peças com os comportamentos customizados em partidas reais.
Assim, existem dois modos básicos de operação que o jogador pode estar:

- Modo Jogo: Modo em que o jogador está em uma partida real.
- Modo Treinamento: Modo em que o jogador está treinando suas peças e definindo seu comportamento.

Um outro conceito importante que deve ser explicado e que é comum para ambos os modos: o de tabuleiro. Ambos os modos apresentam um tabuleiro que pode ser dividido em 3 áreas:

- Banco aliado: área em que as peças do time aliado ficam e que não serão usadas no round seguinte.
- Banco inimigo: área em que as peças do time inimigo ficam e que não serão usadas no round seguinte.
- Campo: área em que todas as peças (aliadas e inimigas) são posicionadas e que irão participar do round a seguir.

Quando se refere ao tabuleiro de um jogador, está se referindo tanto ao seu próprio banco quanto ao campo do tabuleiro.

5.1.1 MODO JOGO

Neste modo o jogador poderá participar de uma partida.

Cada partida deverá implementar algumas regras básicas para o funcionamento de uma partida de jogos do gênero AutoChess. Todavia, devido ao escopo limitado do projeto, apenas algumas regras e mecânicas essenciais foram escolhidas para serem implementadas.

Assim, o jogo seguirá as seguintes regras básicas:

- O jogador terá no início do jogo uma quantidade de pontos de vida. Uma vez que seus pontos chegarem a 0, ele perde
- O jogador receberá no início do jogo uma peça aleatória e uma determinada quantidade de dinheiro.
- A cada rodada o jogador irá ganhar uma determinada quantidade de dinheiro, que é utilizado para comprar novas peças
- A cada rodada o jogador irá ganhar uma determinada quantidade de pontos de experiência, que deve ser acumulada para o jogador subir de nível
- Cada jogador terá um tabuleiro no qual poderá distribuir as peças por ele adquiridas
- A cada rodada o jogador terá alguns segundos para realizar compras de peças e modificações no seu tabuleiro. Esse período será referido como **Tempo de preparo**
- Durante o **Tempo de preparo** o jogador poderá comprar peças e posicioná-las em seu tabuleiro
- As peças que estarão disponíveis para o jogador comprar serão geradas de forma aleatória. A cada round serão geradas sempre 5 novas ofertas de peças
- Cada jogador terá um banco com número máximo de peças fixo, no qual poderá armazenar peças por ele adquiridas que não estão atualmente no seu campo
- O número de peças que um jogador pode ter no campo é menor ou igual ao seu nível

- A cada rodada, uma vez que o **Tempo de preparo** acabar, as peças do campo de um jogador irão lutar com outro time, sem que o jogador possa interferir de modo algum uma vez que a batalha iniciar. Todavia ele ainda poderá reposicionar as peças dentro do seu banco
- No final de cada luta a cada round, o jogador que tiver suas peças derrotadas irá perder pontos de vida

As peças do jogo podem ser de 4 tipos diferentes, chamadas classes, cada uma com um conjunto de características que as definem.

Primeiramente, as seguintes características são levadas em conta:

- HP(Health Points): Pontos de vida de uma peça. Uma vez que chega a 0, ela morre
- DMG(Damage): Pontos de dano de uma peça. Quanto maior o Dmg de uma peça, maior o dano causado por ela a cada ataque
- Range: O alcance do ataque da peça. Quanto maior, maior o alcance e a peça tem a possibilidade de atacar de mais longe.
- Target: Número de alvos que consegue atacar ao mesmo tempo. Pode ser dividido em duas categorias:
    - Single target: Ataca apenas um inimigo de cada vez
    - Multiple target: Ataca mais de um inimigo de cada vez
- MS (Movement Speed): A velocidade da peça. Quanto maior, mais rápido ela se movimenta pelo campo do tabuleiro.

Apresentadas as características que irão definir cada peça, são apresentadas as diferentes classes de peças e o quanto cada uma tem dessas características, além de uma descrição breve do seu comportamento típico:

- **Carry**: peça responsável por causar dano nas peças do time inimigo. São frágeis e devem ser protegidas.
    - HP: Baixo
    - Dmg: Alto
    - Range: Alto

    - Target: Single target
    - MS: Médio
- **Tank**: peça responsável por receber dano das peças inimigas, a fim de proteger as aliadas. São robustas e devem proteger as peças mais frágeis.
    - HP: Alto
    - Dmg: Baixo
    - Range: Baixo
    - Target: Single target
    - MS: Baixo
- **Assassino**: peça responsável por eliminar peças frágeis. São frágeis mas correm riscos para podem causar dano às peças vulneráveis inimigas.
    - HP: Baixo
    - Dmg: Alto
    - Range: Baixo
    - Target: Single target
    - MS: Alto
- **Mago**: peça responsável por causar dano em várias peças do time inimigo ao mesmo tempo. São frágeis e devem ser protegidas.
    - HP: Baixo
    - Dmg: Médio
    - Range: Médio
    - Target: Multiple target
    - MS: Médio

Vale notar que para esse modo, antes de iniciar uma partida, o jogador poderá definir qual é o comportamento de cada classe de peça que será utilizado pelo jogo:

- Default: Comportamento pré-definido pelo próprio jogo para cada peça. Seria o comportamento definido pela produtora do jogo.

- Personalizado: Comportamento que o jogador vai definir para cada tipo de peça através do uso de ML.

As partidas nesse modo podem ser de diferentes tipos:

- Contra outros jogadores (online): os jogadores serão capazes de confrontar-se uns aos outros em uma partida que envolve estratégia de jogo e o treinamento de suas próprias peças.
- Contra computador: terá diferentes níveis de dificuldade, que regulam quanto dinheiro o computador ganha a cada round, e no início da partida o jogador pode escolher se as peças do computador utilizarão o comportamento padrão das peças ou um comportamento treinado pelo próprio jogador ou por outros.

5.1.2 MODO TREINAMENTO

Neste modo o jogador poderá realizar episódios e/ou iterações de treinamentos que serão utilizadas para definir os comportamentos personalizados de cada classe de peça no modo jogo e também simular um round de jogo podendo utilizar os comportamentos customizados.

5.1.2.1 TREINAMENTO DE PEÇAS

É nesse modo que os algoritmos de ML serão utilizados no projeto. Ele foi dividido em três sub-modos diferentes que deveriam ser usados em sequência para no processo de ML:

- Criar Trajetórias: Modo em que o jogador assume papel de expert e mostra como gostaria que o novo comportamento fosse através de rounds. Cada round será uma nova trajetória.
- Selecionar Trajetórias: Modo em que o jogador seleciona dentre as trajetórias criadas quais gostaria que fossem utilizadas para definir o comportamento novo sendo criado.

- MLDev: Modo em que o código de ML poderá rodar baseado nas trajetórias selecionadas para convergir em um comportamento final.

Figura 7 - Menu para os modos de jogo referentes ao treinamento das peças.

Fonte: Autores.

5.1.2.1.1 CRIAR TRAJETÓRIAS

Nesse modo o jogador já começa a utilizar o algoritmo de IRL ao assumir o papel de expert e ao realizar rounds nos quais ele mesmo define no round o comportamento da peça cuja classe se quer criar uma trajetória.

Inicialmente o jogador deve montar o tabuleiro com peças aliadas e inimigas como ele quiser, bem como qual comportamento cada classe irá utilizar, se o padrão ou customizado já existente e também o HP atual de cada peça, com a finalidade de poder simular qualquer tipo de situação que possa vir a encontrar no meio de um jogo real. É indispensável que na configuração do tabuleiro, o time aliado apresente em campo uma peça especial com o comportamento “Treinamento”. Ela é única por time e é ela que será controlada pelo jogador uma vez que o round iniciar.

Uma vez que o round foi iniciado, o jogador poderá controlar a peça especial, podendo escolher para onde ela deve se posicionar no campo e/ou quais peças deve atacar.
Uma vez que o round se encerrar, o jogador deve nomear a trajetória da peça especial que acabou de mostrar para então poder salvá-la ou pode simplesmente descartá-la, caso não esteja satisfeito com como conduziu a peça no round.
Essa etapa de criar trajetórias é a etapa inicial do algoritmo de IRL na qual o expert irá demonstrar como deve ser o comportamento por ele desejado para aquela classe quando se deparar naquele cenário por ele montado.
Tendo criado trajetórias suficientes para mostrar ao algoritmo como deseja que a peça se comporte diante dos diferentes cenários, o jogador pode ir para o próximo modo.

Figura 8 - Modo de jogo Criar Trajetórias

Fonte: Autores

5.1.2.1.2 GERENCIAR COMPORTAMENTOS

Esse modo de jogo é apenas um fluxo de telas nas quais o jogador pode realizar algumas operações relacionadas aos comportamentos dos personagens.

Nele, o jogador terá a opção de listar os comportamentos já existentes ou de criar um novo comportamento.

Quando opta por listar os comportamentos já existentes, será levado a uma tela que lista todos os comportamentos criados e terá a opção de apagar o comportamento selecionado ou de continuar o treinamento desse comportamento ao rodar mais iterações no modo MLDev. Nesse último caso, ele será encaminhado ao modo MLDev.

Quando opta por criar um novo comportamento, será levado a uma tela em que o jogador irá selecionar, dentre todas as trajetórias criadas para uma classe, quais gostaria que fossem utilizadas pelo algoritmo de ML para gerar um novo comportamento. As trajetórias por ele selecionadas são as que de fato o novo comportamento tentará “assimilar” para imitar o comportamento demonstrado pelo expert.

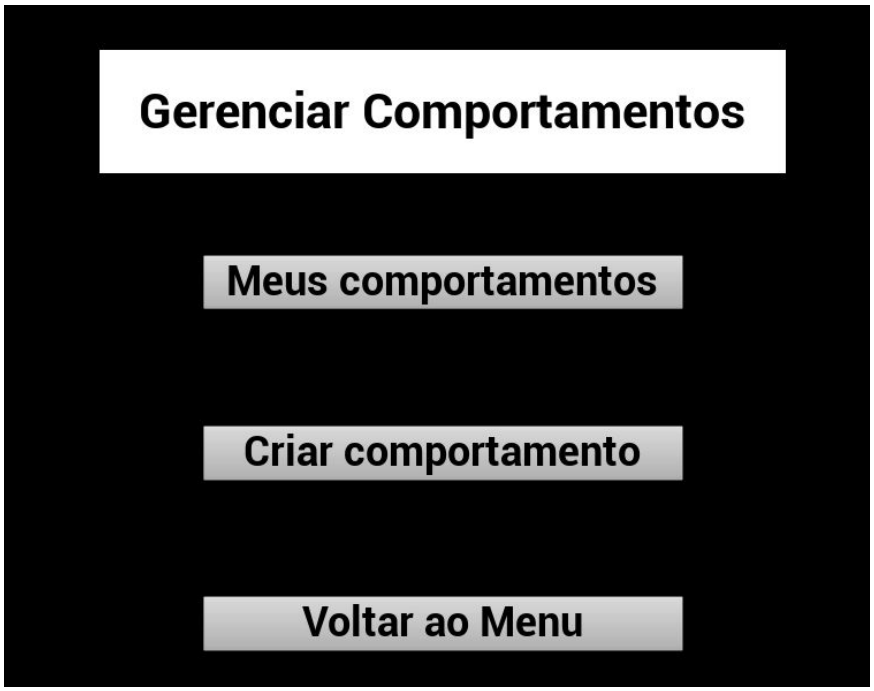
Nela, inicialmente o jogador seleciona para qual classe gostaria de criar um novo comportamento.

Feito isso, serão disponibilizadas todas as trajetórias demonstradas para aquela classe, das quais o jogador pode selecionar as que deseja para servir de modelo para o novo comportamento.

Finalmente, o jogador escolhe um nome para o novo comportamento a ser criado e confirma sua escolha. Uma vez que confirmar, caso o nome escolhido não esteja sendo utilizado por nenhum outro comportamento, será encaminhado diretamente para o modo MLDev. Caso o nome seja conflitante com algum outro comportamento, o jogador será levado a uma tela na qual terá a opção de retornar para escolher outro nome ou de continuar com o nome conflitante e sobrescrever o comportamento que já existia e começar um novo do zero, sendo então encaminhado para o modo MLDev.

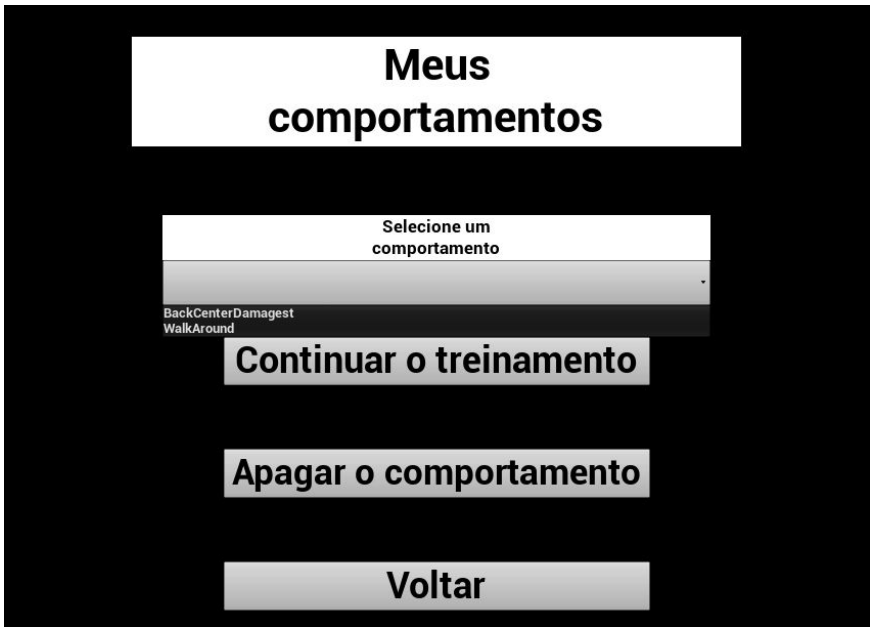
Esse modo de selecionar trajetórias ainda faz parte do algoritmo de IRL, no qual, tomando-se os exemplos dados pelo expert, o algoritmo irá gerar pesos de features para servir de base para o algoritmo de RL.

Figura 9 - Menu do modo de jogo Gerenciar Comportamentos



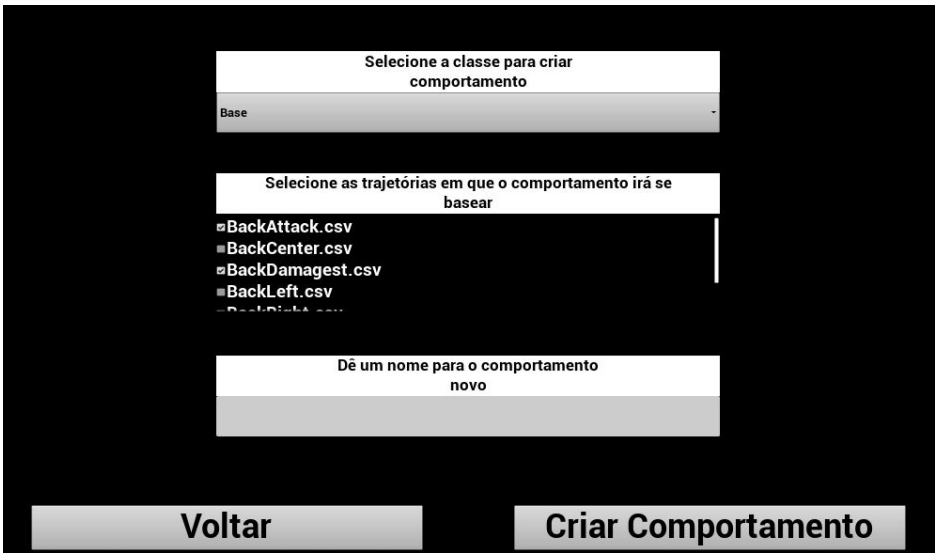
Fonte: Autores

Figura 10 - Menu de escolha entre continuar ou apagar um comportamento já existente.



Fonte: Autores

Figura 11 - Menu em que se pode criar um novo comportamento.



Fonte: Autores

5.1.2.1.3 MLDEV

Nesse modo o jogador poderá iniciar as iterações com o algoritmo de RL baseado nas trajetórias selecionadas anteriormente para que ele possa convergir para algum comportamento desejado.

Inicialmente o jogador deve montar o tabuleiro com peças aliadas e inimigas como ele quiser, bem como qual comportamento cada classe irá utilizar, se o padrão ou customizado já existente e também o HP atual de cada peça, com a finalidade de poder simular qualquer tipo de situação que possa vir a encontrar no meio de um jogo real. Para essa etapa, ao rodar as diferentes iterações, é interessante que as configurações e cenários de peças no campo de cada tabuleiro sejam diferentes a cada iteração, para poder treinar o comportamento também em diversos cenários. Por isso, nesse modo é possível se montar diversas configurações de tabuleiro antes de iniciar o treinamento. Cada configuração apresenta obrigatoriamente uma peça com o comportamento novo sendo treinado.

Uma vez que o treinamento foi iniciado, o algoritmo de RL será utilizado em cada round para gerar as tabelas recompensas a cada ação para cada estado, visando

aproximar-se das trajetórias escolhidas no modo anterior. Ao final de cada round, o comportamento da peça no round será informado ao algoritmo de IRL para que possa definir se foi perto o suficiente do esperado ou não e baseado nisso realizar ajustes nos pesos das features para a próxima iteração. Além disso, no final de um certo número de iterações, dados que são usados pelos algoritmos de ML serão armazenados para preservar o resultado do treinamento até então. Feito isso, a próxima iteração irá carregar um novo cenário criado pelo jogador e o processo se repetirá até que o jogador fique satisfeito com o comportamento atual da peça em treinamento ou até o algoritmo convergir.

Finalmente, após todo esse processo, o jogador poderá usar o novo comportamento no modo jogo ou para treinar outras peças.

Vale notar que para que os algoritmos de ML possam definir os estados e suas variáveis que as definem, terão acesso à algumas características de cada classe de cada peça, além de informações da peça relativa ao round atual (tanto no modo Criar Trajetórias quanto no modo MLDev), como dano total causado, HP atual, sua distância em relação à peça em treinamento etc.

Dessa forma, espera-se poder caracterizar de forma suficientemente fiel qual é o cenário planejado pelo jogador e qual o comportamento por ele esperado.

Esse modo pode ser acessado diretamente através do menu principal na implementação do grupo, todavia essa possibilidade existe apenas para fins de desenvolvimento do projeto. Em um produto que fosse uma versão final para usuários, para acessá-lo, seria necessário passar pelo processo de criar um novo comportamento (ou sobrescrever um já existente) após selecionar em quais trajetórias que o novo comportamento vai se basear ou de continuar o treinamento de um comportamento já existente.

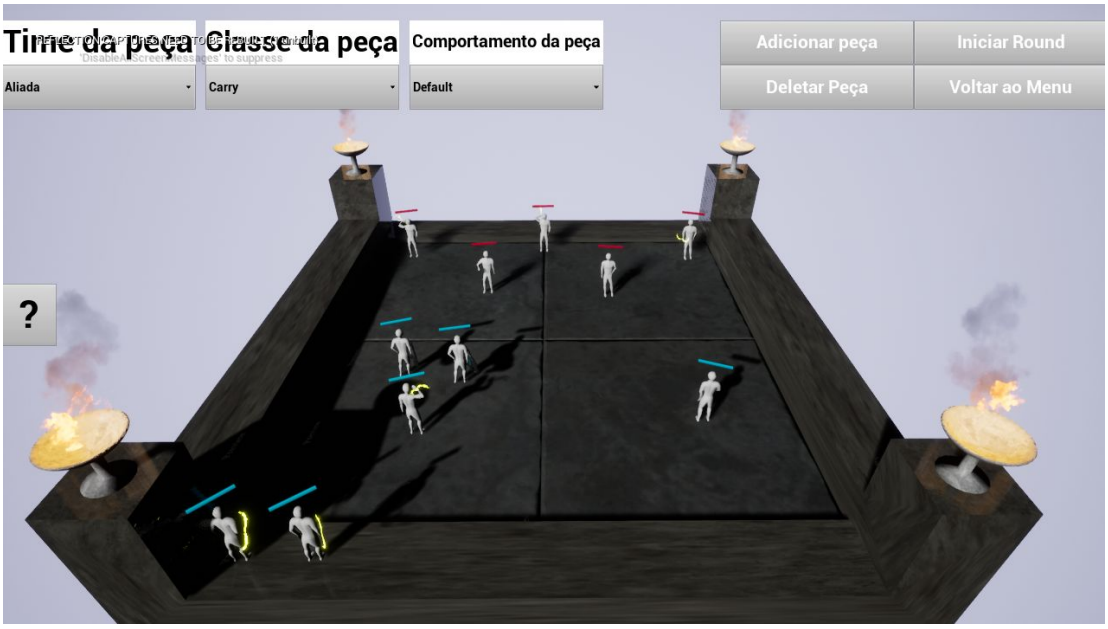
Figura 12 - Modo de jogo MLDev.

Fonte: Autores

5.1.2.2 SIMULAÇÃO DE ROUNDS

Nesse modo o jogador vai poder montar o tabuleiro com peças aliadas e inimigas como ele quiser, bem como qual comportamento cada classe irá utilizar, se o padrão ou customizado já existente e também o HP atual de cada peça, com a finalidade de poder simular qualquer tipo de situação que possa vir a encontrar no meio de um jogo real.
Após isso, ele pode iniciar o round para ver como as peças irão interagir e se comportar. Com isso o jogador poderá ter um retorno de como o treinamento da peça está até o momento, podendo decidir se deve aprimorá-lo mais ainda ou se está satisfeito.

Figura 13 - Modo de jogo Simular Rounds.



Fonte: Autores

5.2 FEATURES

Primeiramente, para que se possa expor as features e classificá-las, é necessário identificar e listar os requisitos funcionais e não-funcionais do projeto.

5.2.1 REQUISITOS FUNCIONAIS

Os requisitos funcionais são:

- **O jogador deve ser capaz de treinar as peças**: as peças devem apresentar comportamento semelhante ao demonstrado.
- **O jogador deve ser capaz de armazenar comportamentos**: o jogador deve ser capaz de armazenar o comportamento novo criado para ser utilizado mesmo que seja fechado e aberto novamente em algum outro momento.

- **Cada classe de peça deve possuir o próprio comportamento**: cada classe pode ser treinada pelo jogador resultando em comportamentos diferentes para cada um.
- **O jogador deve poder simular o round de uma partida**: para verificar o comportamento das peças, o jogador deve ser capaz de configurar o tabuleiro e executar o jogo, para assim, analisar os resultados. (Vide Seção 2.3.1 Game Design para mais detalhes sobre o round).
- **O jogador deve poder conseguir realizar uma partida de autochess**: uma partida consiste do início do jogo até um dos jogadores perder todos os pontos de vida. (Vide Seção 2.3.1 Game Design para mais detalhes sobre uma partida).
- **O jogador deve conseguir jogar contra 3 níveis diferentes de dificuldade**: quando o jogador estiver enfrentando um computador na partida, ele terá os níveis fácil, médio e difícil de dificuldade.
- **O jogador deve ser capaz de confrontar peças de outros jogadores**: o jogador consegue utilizar seus comportamentos personalizados contra outros comportamentos personalizados salvos de outros jogadores.
- **O jogador deve ser capaz de jogar online**: dois jogadores confrontam um ao outro ao mesmo tempo.

5.2.2 REQUISITOS NÃO-FUNCIONAIS

Os requisitos não-funcionais são:

- **O aprendizado do comportamento não deve impactar na experiência do jogador**: após treinar a peça, o jogador deve conseguir utilizá-lo com pouco tempo de espera, ou seja, o algoritmo de ML deve convergir em um tempo aceitável ao jogador.
- **O jogador deve ser capaz de armazenar vários comportamentos**: o jogo deve conseguir gravar mais de um comportamento customizado do jogador.

- **O jogo deve rodar em máquinas comuns do perfil médio dos jogadores**: não é necessário que o computador do usuário tenha hardware de última geração, ou seja, a aplicação deve conseguir ser executada em computadores utilizados para a realização de tarefas do dia a dia, que seria o perfil médio esperado.
- **A interface de treinamento deve ser de fácil utilização**: o modo treinamento deve ser claro o suficiente para um jogador novo sem experiência conseguir utilizá-lo.

5.2.3 LISTA E CLASSIFICAÇÃO DOS REQUISITOS

Com todos os requisitos especificados, pode-se organizá-los de acordo com seu tipo e prioridade para o projeto. Optou-se pelo esquema MoSCoW (do inglês “Must have, Should have, Could have, Won’t have” ou em português “Deve ter, Deveria ter, Poderia ter, Não terá”), com a classificação de cada uma das features caindo em uma categoria. Obteve-se a seguinte tabela:

Tabela 1 - Tabela de requisitos do projeto

Requisito	Tipo	Prioridade
O jogador deve ser capaz de treinar as peças	Funcional	Deve ter
O jogador deve ser capaz de armazenar comportamentos	Funcional	Deve ter
Cada classe de peça deve possuir o próprio comportamento	Funcional	Deveria ter
O jogador deve poder simular o round de uma partida	Funcional	Deve ter
O jogador deve poder conseguir realizar uma partida de autochess	Funcional	Deveria ter

(Continua na próxima página)

(Conclusão da tabela 1)

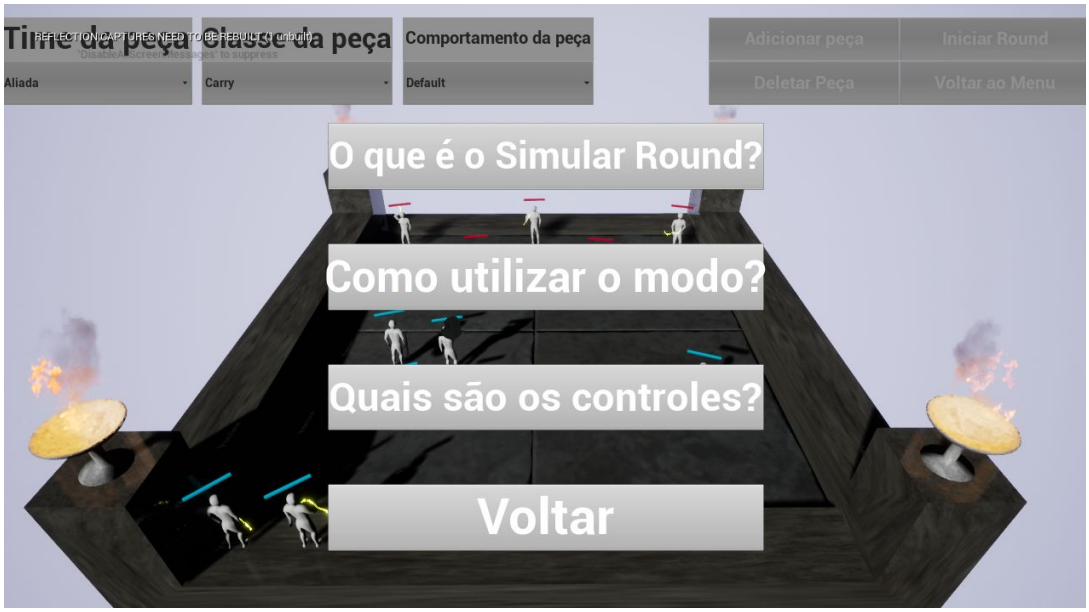
Requisito	Tipo	Prioridade
O jogador deve conseguir jogar contra 3 níveis diferentes de dificuldade	Funcional	Poderia ter
O jogador deve ser capaz de confrontar peças de outros jogadores	Funcional	Deveria ter
O jogador deve ser capaz de jogar online	Funcional	Poderia ter
O aprendizado do comportamento não deve impactar na experiência do jogador	Não funcional	Deveria ter
O jogador deve ser capaz de armazenar vários comportamentos	Não funcional	Deveria ter
O jogo deve rodar em máquinas comuns do perfil médio dos jogadores	Não funcional	Deveria ter
A interface de treinamento deve ser de fácil utilização	Não funcional	Deve ter

5.2.4 FACILIDADE DE USO DO SISTEMA

Um dos requisitos levantados para o projeto é o de que o jogador deve conseguir utilizar o sistema desenvolvido de forma fácil, sem precisar ter experiência prévia para tal, como uma forma de conseguir atingir um público maior, sem que ele tenha a necessidade de ter conhecimentos sobre programação ou algoritmos de ML e ainda assim conseguir utilizar a feature de conseguir treinar peças.

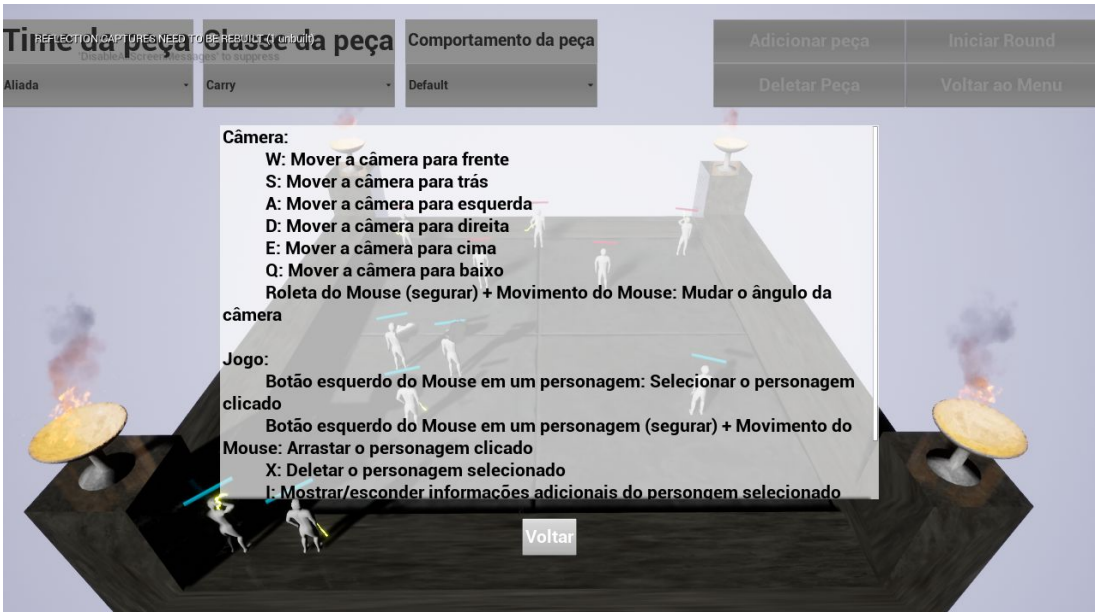
Tendo isso em mente, foram implementados alguns elementos visuais que auxiliam o usuário com algumas possíveis dúvidas e erros que podem aparecer conforme utiliza o sistema. Alguns exemplos são mostrados e explicados a seguir:

Figura 13 - Menu de ajuda do modo de jogo Simular Rounds.



Fonte: Autores

Figura 14 - Menu de ajuda sobre os controles do modo de jogo Simular Rounds.



Fonte: Autores

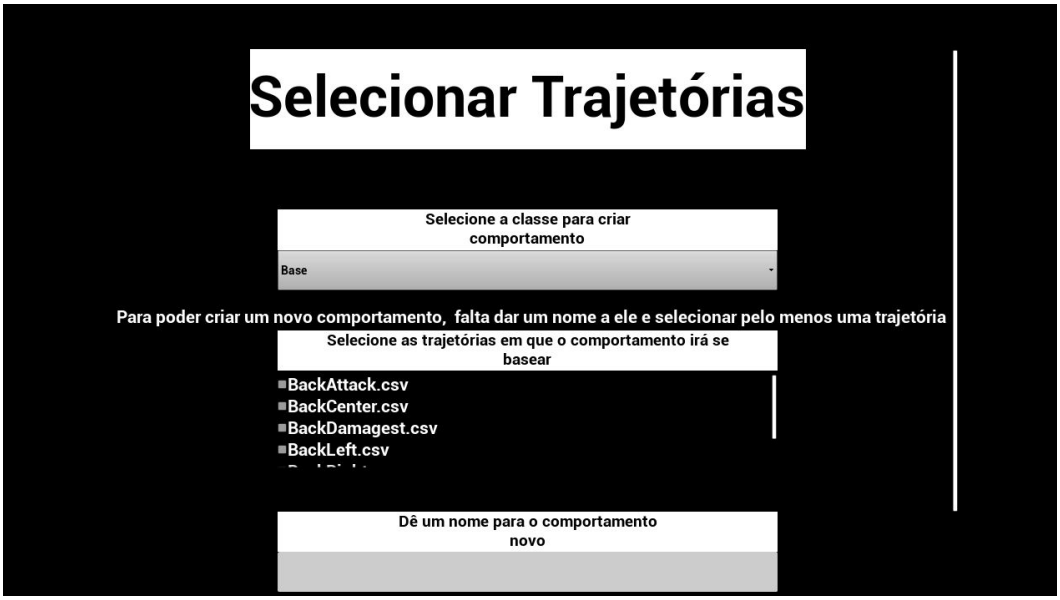
Na Figura 13, é visto o modo de jogo Simular Rounds. É possível ver que à esquerda existe um botão com o símbolo “?”. Ele é um botão que existe também nos modos de

jogo Criar Trajetórias e MLDev (ao entrar em qualquer um desses modos, é mostrada uma mensagem de ajuda que sugere ao jogador que clique neste botão para obter ajuda) e ao se clicar nele, é aberto um menu de ajuda, como pode ser visto na Figura 14. Nele, a pessoa pode obter algumas informações sobre o modo de jogo, como utilizá-lo e quais os controles. Um exemplo de explicação dos controles desse menu pode ser visto na Figura 14.

Figura 15 - Mensagem de erro ao tentar iniciar um Round.

Fonte: Autores

Figura 16 - Mensagem de erro ao tentar criar um novo comportamento no modo de jogo Gerenciar Comportamentos.



Fonte: Autores

Outra forma adotada e utilizada para auxiliar o usuário é mostrar mensagens na tela caso esteja tentando realizar de forma errada alguma ação. Exemplos de mensagens são na Figura 15, em que se está tentando iniciar um Round sem que o time inimigo tenha peças no campo, e na Figura 16, em que se está tentando criar um novo comportamento sem escolher um nome para ele nem selecionar quais trajetórias serão utilizadas para os algoritmos de ML.

6 IMPLEMENTAÇÃO

Para a implementação final do projeto foi utilizada a *UnrealEngine4*[5] ™, ferramenta motora (engine) de criação de jogos e aplicações da *EpicGames™.* Os arquivos utilizados no projeto vieram de diversas fontes:

Primeiramente, os autores acabaram por realizar algumas modelagens de alguns objetos, utilizando o software de criação *Blender*[6] ™, bem como a maior parte do código do projeto.

Há código que foi obtido da *VRMonkey*[7] ™: o código do shader que foi utilizado para fazer brilhar os diferentes locais fixo da arena, além do código C++ que foi utilizado para ser GameMode do MainMenu e Gerenciar Comportamentos, com a possibilidade de definir via nó de Blueprint (Blueprints Visual Scripting é um sistema de scripting na UE4 [23]) qual o GameMode a ser utilizado no próximo Level a ser carregado, que pode ser encontrado no diretório Source do projeto. Além disso, código de um projeto pertencente a ela foi utilizado como referência para o projeto, permitindo estudar como implementar na UE4 uma forma de buscar textos pré-definidos pelos desenvolvedores para serem utilizados no jogo.

Alguns arquivos relacionados à arte do projeto foram obtidos do Starter Content da UE4, que é um conjunto de determinados materiais, partículas e static meshes que pode ser incluído na etapa inicial da criação de um projeto, como pode ser visto em [24]. Os arquivos do Starter Content desse projeto podem ser encontrados quando se utiliza o Starter Content da UE4 na versão 4.24.3.

Finalmente, variadas partes do projeto puderam ser programadas/implementadas/modeladas com ajuda e tendo como base diversos tutoriais e documentações. Algumas referências muito utilizadas foram:

- Documentação da UE4 [25]
- AnswerHub da UE4 [26]

[5] EPIC GAMES™. Unreal Engine 4™. 2020. Disponível em: <https://www.unrealengine.com/en-US/>
[6] BLENDER™. Blender creation suite. 2020. Disponível em: <https://www.blender.org/>
[7] VRMONKEY™. 2020. Disponível em: <http://www.vrmonkey.com.br/>

- Canal do *YouTube*™ Mathew Wadstein [27]

Para a programação do jogo foi utilizada tanto programação através de Blueprints quanto código direto em C++, além da programação de shaders através da interface de edição de materiais do editor.

6.1 ESTRUTURA DE ARQUIVOS

A seguir é apresentada a estrutura geral dos arquivos e diretórios do projeto na Unreal Engine 4, com breves comentários e descrições a respeito deles.

Tabela 2 - Arquivos e diretórios em /Content

/Content	
Diretório/Arquivo	Descrição
./Art	Diretório contendo arquivos, assets e código do projeto relacionados à arte do projeto, como animações, meshes (skeletal e static), materiais, texturas e particle systems
./Dev	Diretório contendo arquivos da parte de programação do jogo com exceção de código da parte de arte do projeto
./Maps	Diretório contendo os levels utilizados no projeto

Tabela 3 - Arquivos e diretórios em /Content/Art

/Content/Art	
Diretório/Arquivo	Descrição
./Diverse	Diretório contendo alguns materiais, particle systems e texturas que podem ser encontrados no Starter Content da Unreal Engine. O particle system P_Fire veio do Starter Content também mas foi modificada para atender às necessidades do projeto.
./Gameplay	Diretório contendo arquivos da parte de arte utilizados pelos personagens, arena e cenário do projeto

Tabela 4 - Arquivos e diretórios em /Content/Dev

/Content/Dev	
Diretório/Arquivo	Descrição
./Data	Diretório contendo arquivos que são utilizados para o armazenamento e gerenciamento dos dados utilizados no código do projeto como Enums, DataTable e Structs
./Engine	Diretório contendo arquivos relacionados à aspectos de funcionamento da engine em si, como Controllers, GameInstance, GameModes, GameStates, Interface e SaveGame
./Gameplay	Diretório contendo arquivos e blueprints utilizados e relacionados à elementos mais específicos do jogo: as blueprints dos diferentes personagens, dos diferentes equipamentos que cada classe utiliza, do peão utilizado pelo jogador e da arena, bem como de uma posição fixa da arena
./Widgets	Diretório contendo os diferentes Widget Blueprints utilizados no código do projeto

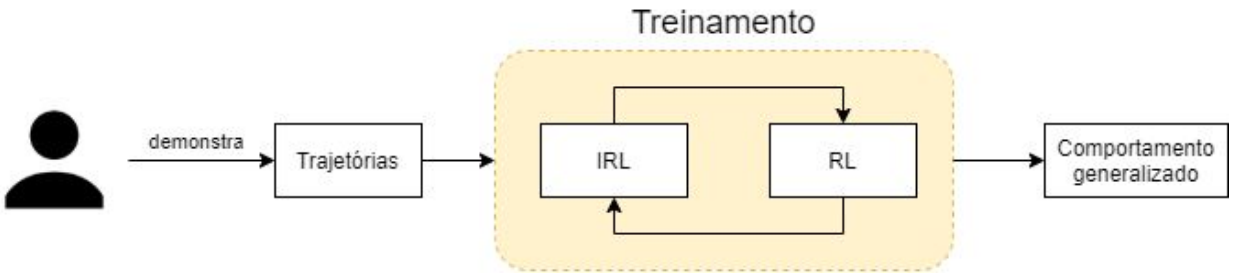
Tabela 5 - Arquivos e diretórios em /Content/Maps

/Content/Maps	
Diretório/Arquivo	Descrição
./Board	Level contendo o tabuleiro, utilizado para modos de jogo no qual seu uso é necessário, como o modo Jogo, Gravar Trajetórias, MLDev e Simular Round.
./MainMenu	Basicamente um level sem nada relevante nele em si. Utilizado para modos de jogo em que apenas se interage com menus, como o MainMenu e o Gerenciar Comportamentos

6.2 TREINAMENTO

A Figura 17 apresenta uma visão de alto nível do fluxo de treinamento, o jogador apresenta trajetórias do comportamento desejado para o algoritmo de treino que consiste na iteração de IRL e RL que então converge para um comportamento generalizado.

Figura 17 - Fluxo de alto nível do treinamento.



Fonte: Autores.

Para o treinamento é necessário definir cenários em que o agente vai tomar ações até o fim do round, a Figura 18 apresenta o interface para realizar isso.

Figura 18 - Interface de configuração de cenários e visualização do treinamento.

Fonte: Autores

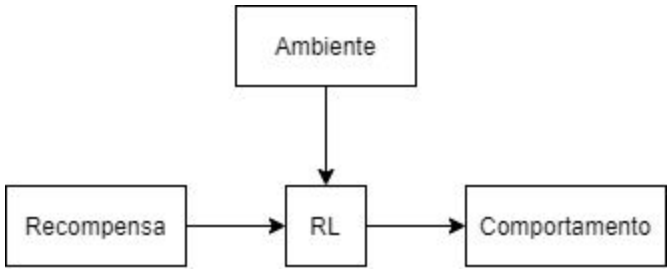
6.3 TREINAMENTO REINFORCEMENT LEARNING

A implementação do algoritmo SarsaLambda apresenta os seguintes valores:

- ε (Epsilon) = 0.2
- α (Taxa de aprendizado) = 0.1
- γ (Fator de desconto) = 0.95
- λ (Lambda) = 0.9

O algoritmo de RL necessita de uma função de recompensa e informação sobre o estado atual do agente para conseguir convergir em um comportamento, como mostra a Figura 19.

Figura 19. Entradas e saídas do RL.



Fonte: traduzido [28]

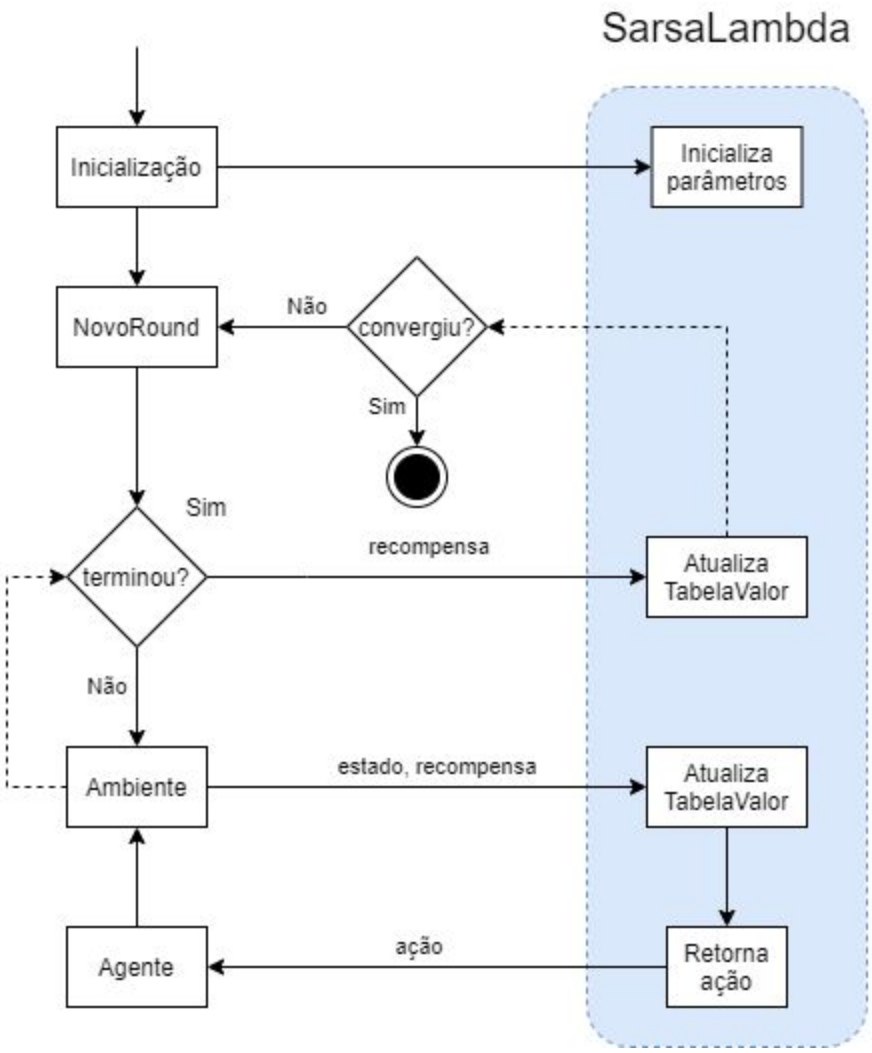
Para o teste e confirmação da convergência do algoritmo de RL foi adotada a seguinte recompensa:

- +1 a cada passo em que o agente está vivo;
- +10 em caso de vitória;
- +10 em caso de derrota;
- +x dependendo da quantidade de dano e vida no final do round.

A Figura 20 mostra o fluxo de treinamento. Primeiro ocorre a inicialização dos parâmetros e um round é iniciado. A cada passo do round, o SarsaLambda recebe o estado do ambiente e a recompensa da ação anterior que o agente tomou, com isso, o

algoritmo atualiza a Tabela Valor e retorna a próxima ação do agente podendo ser a melhor (greedy) ou aleatória para explorar. O agente toma esta ação que modifica o ambiente e o fluxo recomeça. Isso acontece até que o round acabar, o SarsaLambda recebe a recompensa do round para atualizar a Tabela Valor e então é iniciado um novo round até um critério de parada definido, como o número de rounds (episódios) realizados.

Figura 20 - Fluxo de treinamento RL.



Fonte: Autores

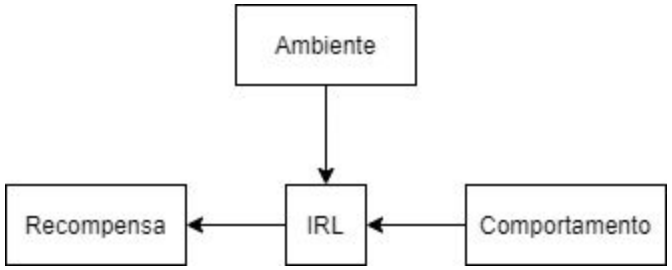
6.4 TREINAMENTO INVERSE REINFORCEMENT LEARNING

A implementação do Apprenticeship Learning apresenta os seguintes valores:

- ε (Epsilon - condição de parada) = 0.23
- γ (Taxa de desconto) = 0.99

O algoritmo de IRL funciona de maneira “contrária” a do RL, a partir do ambiente e do comportamento, busca-se encontrar uma recompensa que os represente, como mostra a Figura 21.

Figura 21 - Entradas e saídas do IRL.



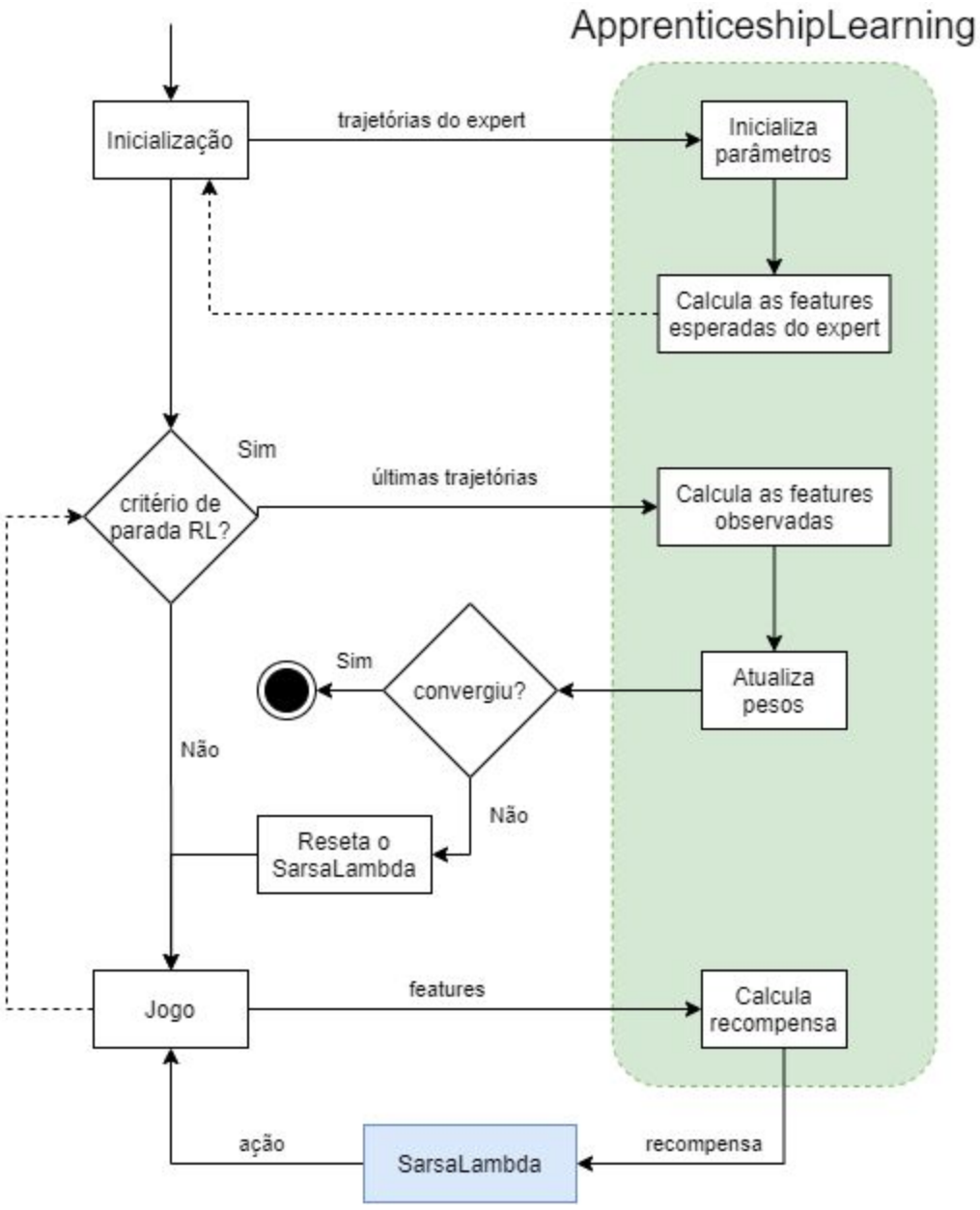
Fonte: traduzido [28]

A Figura 22 apresenta o fluxo de treinamento do ApprenticeshipLearning. Antes do início é necessário que haja trajetórias do expert gravadas como explicado anteriormente, a partir delas, é feita a inicialização dos parâmetros e calculada as features esperadas pelo expert para então iniciar as iterações do treinamento.

Cada iteração consiste em atualizar os pesos de modo que o comportamento apresente features observadas semelhantes ao que o expert demonstrou, estes pesos são utilizados para calcular a recompensa de cada ação do agente. Assim cada iteração do treinamento consiste em: dado um vetor de pesos, encontre o comportamento resultante através do algoritmo de RL, calcule as features observadas e atualize os pesos baseado na diferença entre o que o expert demonstrou e o

comportamento encontrado. Repita estas iterações até que o critério de convergência seja alcançado.

Figura 22 - Treinamento IRL.



Fonte: Autores

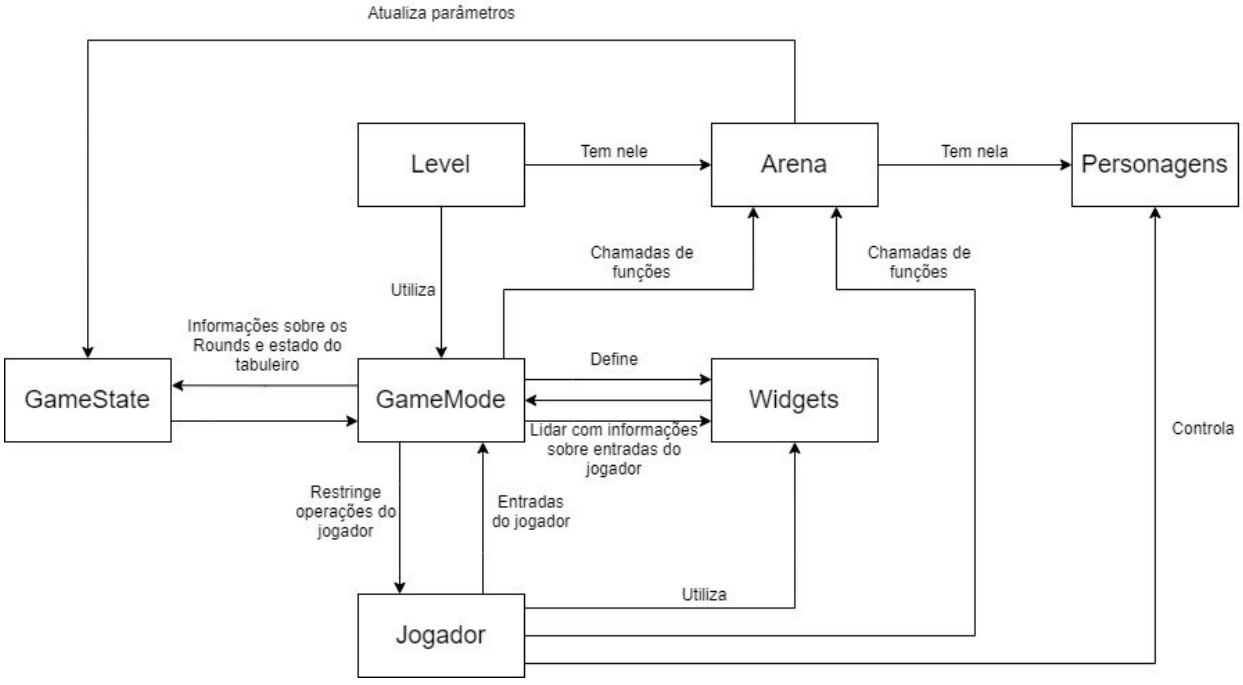
7 ARQUITETURA

7.1 RELACIONAMENTO ENTRE CLASSES E ASSETS NO PROJETO

Como já dito, um dos aspectos principais do projeto é o de criar uma versão simplificada de um jogo do estilo AutoChess, com peças utilizando comportamentos default ou personalizados, além de servir também como uma plataforma para aplicar os algoritmos de ML expostos.

Dessa forma, será apresentado de forma geral como alguns assets e classes utilizados no projeto se relacionam entre si tendo em mente os aspectos citados.

Figura 24 - Visão geral do funcionamento dos modos de jogo.



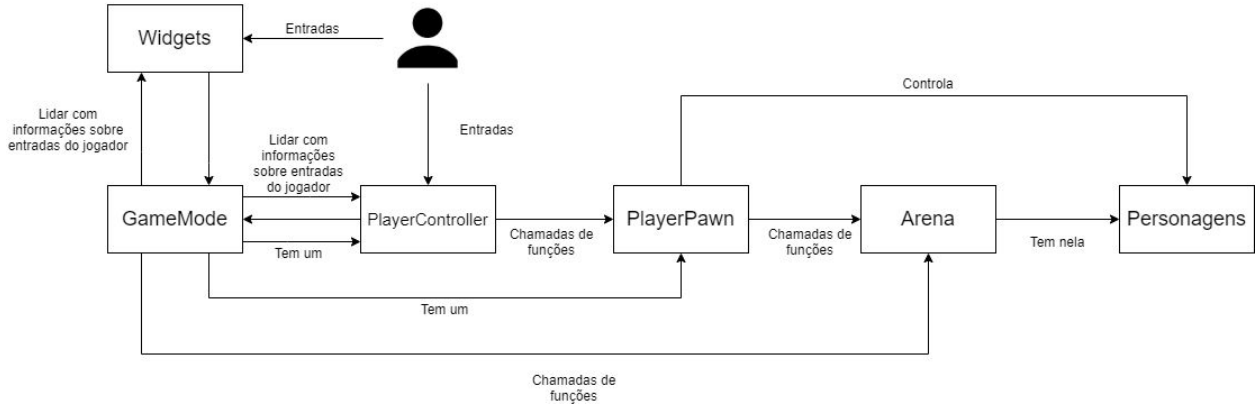
Fonte: Autores.

Primeiramente, a Figura 24 fornece uma visão geral do funcionamento dos modos de jogo do projeto (com exceção do Menu principal e do Gerenciar Trajetórias):

- O Level possui nele uma Arena e utiliza um GameMode baseado no modo de jogo (por exemplo, o modo Gravar Trajetórias tem um GameMode específico para ele, o modo SimularRound tem outro específico para ele etc)
- O GameMode, de forma geral, é o asset responsável por ditar as regras do jogo atual. Assim, algumas características como por exemplo o número de personagens que um jogador pode ter no seu banco, quantos personagens cada jogador pode ter no campo, se o round pode começar etc são definidos nele. Ele irá definir o Widget que será utilizado como interface para o jogador baseado no modo de jogo atual (mais uma vez, o modo Gravar Trajetórias possuirá um widget diferente do widget para o modo Simular Round). Além disso, ele irá também lidar com informações acerca das entradas do jogador (como por exemplo, quando o jogador inicia o round, tenta criar um novo personagem etc) com esse widget. Nele estão presentes também chamadas de funções relativas aos algoritmos de ML, como por exemplo começar a gravar as trajetórias, ajustar valores de recompensas etc.
- O GameState, de forma geral, é o asset que está gerenciando e armazenando algumas variáveis do estado atual de cada round, como número de peças vivas de cada time, quantas peças cada time tem no campo e no banco, funções que auxiliam no trabalho de voltar o campo para a configuração inicial antes do Round ser iniciado etc. Assim, ele irá interagir com o GameMode para poderem gerenciar o andamento do jogo e com a própria Arena por exemplo quando precisa atualizar algum valor do número de peças de um determinado time.
- A Arena representa a arena do jogo e é nela em que os personagens estão. De forma geral, ela recebe algumas chamadas de funções do GameMode quando precisa por exemplo criar um personagem novo ou do Jogador quando está arrastando um personagem pela arena.
- O Jogador, de forma geral, irá gerenciar os personagens (através de chamadas de funções com a Arena, através de entradas que serão lidadas diretamente pelo GameMode ou até mesmo com chamadas de funções do Controller

responsável pelo personagem) ou interagir diretamente com o Widget do modo de jogo para utilizar a interface.

Figura 25 - Visão geral do funcionamento do Jogador.



Fonte: Autores.

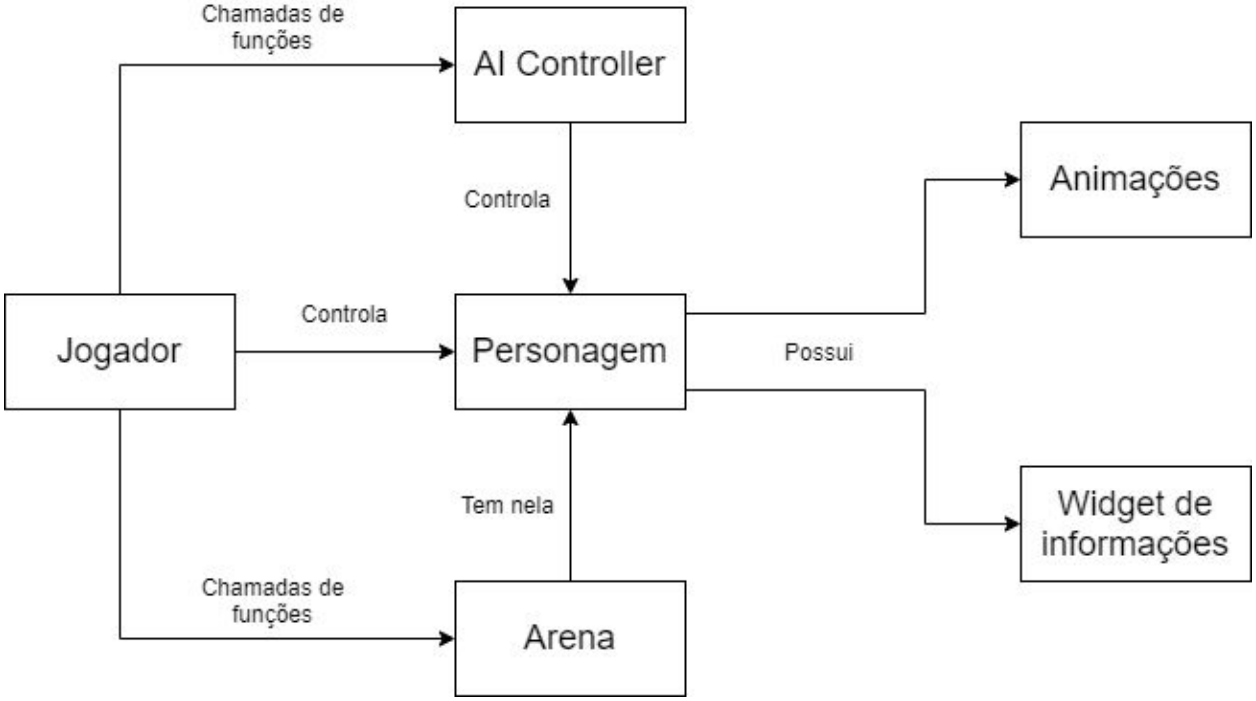
A Figura 25 fornece uma visão geral do funcionamento de como a pessoa consegue interagir com o jogo e como suas entradas são interpretadas no projeto. Ele pode ser entendido como uma versão mais detalhada e complementar do bloco “Jogador” das Figuras 24 e 26:

- As entradas da pessoa poderão ser diretamente no Widget (como já explicado) ou passar pelo PlayerController.
- O GameMode, como já explicado, lida em conjunto com esse Widget com as informações das entradas do jogador feitas no Widget e também chama funções da Arena. Além disso, ele também irá definir qual o PlayerController e o PlayerPawn a serem utilizados no modo de jogo.
- O PlayerController, de forma geral, é o asset que será responsável por mapear as entradas da pessoa em funções e eventos dentro do código do jogo. Assim, irá lidar juntamente com o GameMode com algumas dessas entradas e irá também chamar funções do PlayerPawn, que seriam as que deveriam acontecer

quando a pessoa pressiona determinada tecla ou realiza determinada ação dentro do jogo.

- O PlayerPawn, de forma geral, vai ser o asset para representar o jogador dentro do jogo, possuindo por exemplo o componente que serve para determinar o que o jogador enxerga do jogo (fora os elementos de interface), funções para mover esse componente, arrastar o personagem, controlar um personagem etc. Para algumas dessas funções, chama funções da Arena.

Figura 26 - Visão geral do funcionamento do Personagem.



Fonte: Autores.

Finalmente, a Figura 26 fornece uma visão geral do funcionamento de como um personagem é implementado dentro do projeto. Ele pode ser entendido como uma versão mais detalhada e complementar do bloco “Personagem” das Figuras 24 e 25:

- O Personagem, de forma geral, representa um personagem na Arena. Ele possui alguns atributos do personagem como o alcance do seu ataque, seu

estado (se está atacando, morto etc), sua classe etc. Ele pode ser controlado pelo Jogador (como já explicado, em conjunto algumas vezes de funções da Arena, por exemplo). O Jogador pode também atuar em conjunto com o AI Controller, ao chamar funções desse Controller, para mover o personagem no modo Criar Trajetórias. Não só isso, ele faz uso de Animações dentro do jogo, e possui um Widget para mostrar algumas de suas informações.

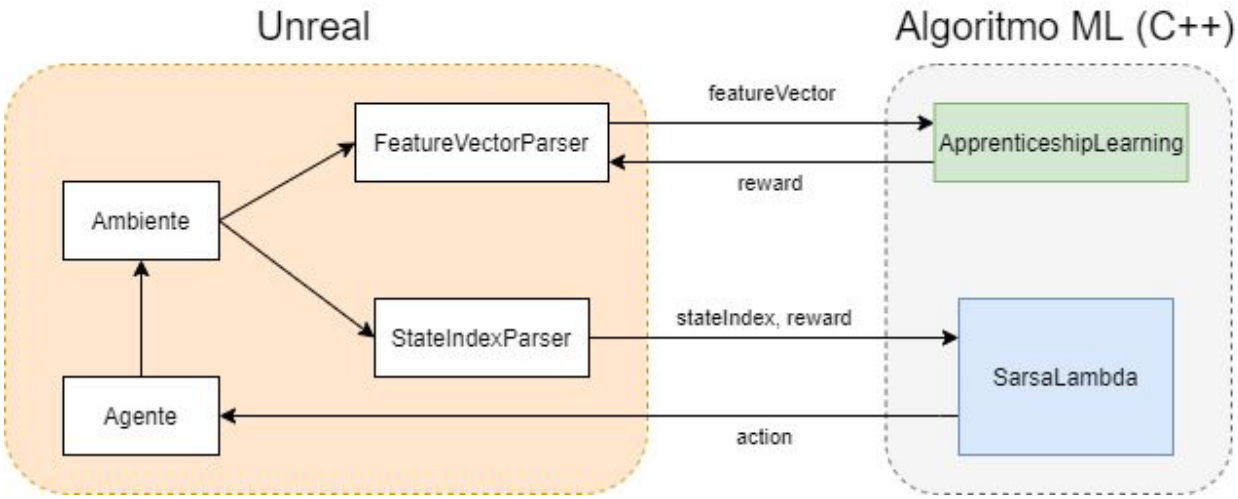
- O AI Controller, de forma geral, é o Controller que irá ditar o comportamento do Personagem no jogo. Ele possui o código que é utilizado para decidir sua próxima ação utilizando comportamentos personalizados obtidos através do treinamento com algoritmos de ML ou um comportamento default pré-definido.

7.2 INTERAÇÃO AMBIENTE E ML

A Figura 27 a seguir apresenta a interação do ambiente (jogo) e o algoritmo de ML, basicamente o algoritmo de ML não conhece nada do domínio do jogo, apenas recebe primitivas escalares (índice do estado) ou vetor de primitivas (vetor de features) e retorna primitivas (ação), sendo o ambiente responsável por realizar o mapeamento destes valores.

Este mapeamento é realizado por duas classes principais, o FeatureVectorParser recebe o estado do ambiente e mapeia em um vetor de features que é utilizado pelo ApprenticeshipLearning e o StateIndexParser recebe o estado do ambiente e mapeia para um inteiro que o representa.

Figura 27 - Interação ambiente e algoritmo de ML.



Fonte: Autores.

8 MODELAGEM RL

O agente deve ter informações suficientes sobre como percebe o ambiente para ser capaz de tomar “boas” decisões, ou seja, decisões que no final resultem na maior recompensa possível. Entretanto, quanto maior o número de detalhes, maior será o número de estados possíveis a serem visitados e avaliados.

Outro ponto importante a ser levado em consideração é a variação da quantidade de inimigos e aliados no campo sem a limitação no número de personagens de cada classe. Por exemplo, pode haver 3 personagens da classe “Tank” e nenhum de outras classes.

Dito isso, existe o desafio de modelar essa percepção de uma maneira que seja capaz de representar bem o ambiente com o menor número de estados possível. Assim, adotou-se a abordagem de trabalhar com intervalos que resultam em um valor para a característica em questão, por exemplo, o mapa é dividido em 9 regiões e a posição do agente é representada por uma coordenada (x,y) da região em que ele está, ou seja, x, y $\subset$ {0, 1, 2}. A Figura 28 apresenta um exemplo em que o agente azul está quase saindo da posição (1,0).

Figura 28. Exemplo de divisão do mapa em regiões.

Fonte: autores.

A seguir, apresenta-se a evolução da modelagem, partindo de uma simples até a modelagem final mais complexa. A notação adotada para a composição de estados é <descrição> (<número de valores possíveis>).

8.1 MODELAGEM 1 - INICIAL

Essa modelagem é composta por:

Ações possíveis:

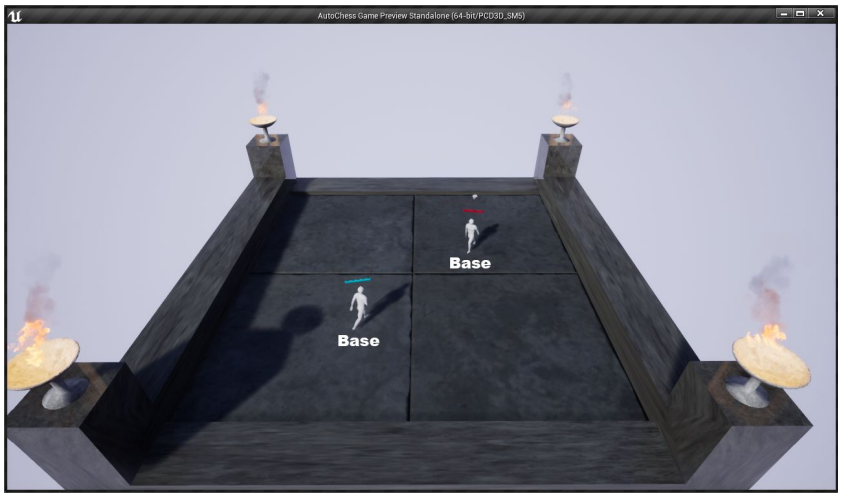
- Atacar o inimigo mais próximo;
- Mover em direção ao inimigo mais próximo;
- Ficar parado.

Composição do estado (Total 162 estados possíveis):

- Posição X do agente (3);
- Posição Y do agente (3);
- Vida do agente (3);
- Dano do agente (3);
- Inimigo mais próximo no alcance de ataque (2).

O teste foi realizado com 1v1 entre dois agentes de classe “Base”, como mostra a Figura 29 a seguir:

Figura 29 - Configuração para o teste da modelagem 1.



Fonte: Autores.

Analisando o vídeo do teste [29], foi possível concluir que o conjunto de ações e a configuração de treino estavam muito simples, pois o que estava definindo o vencedor era quem ataca primeiro.

8.2 MODELAGEM 2 - INFORMAÇÃO DE OPONENTES CHAVES E MOVIMENTAÇÃO DIRECIONAL

Dados os problemas apresentados anteriormente, a modelagem apresenta novas ações possíveis a serem tomadas, como a movimentação direcional, e mais informações para ser acrescentada à essas decisões, como dados sobre inimigos chaves, sendo assim a modelagem é composta por:

Ações possíveis:

- Atacar o inimigo mais próximo;
- Mover em direção ao inimigo mais próximo;
- Ficar parado;
- Mover para o Norte (Absoluto)
- Mover para o Sul (Absoluto)
- Mover para o Oeste (Absoluto)

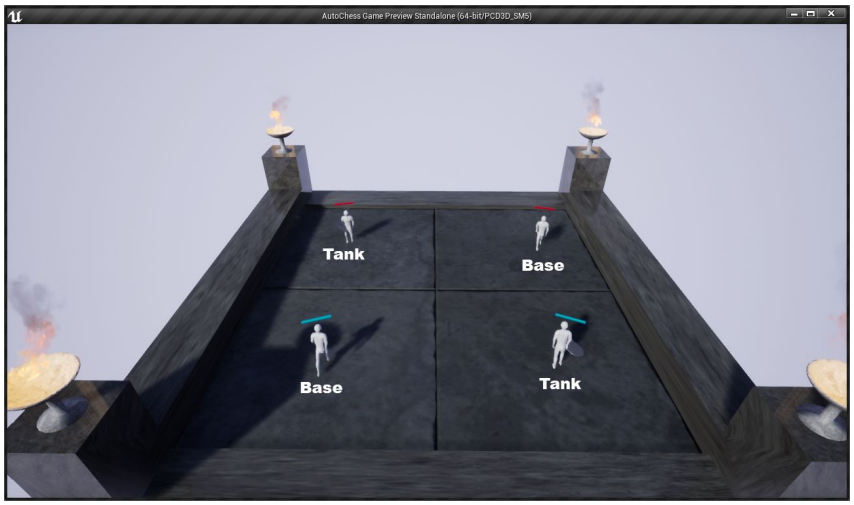
- Mover para o Leste (Absoluto)

Composição do estado (Total 26.244 estados possíveis)

- Posição X do agente (3);
- Posição Y do agente (3);
- Vida do agente (3);
- Dano do agente (3);
- Inimigo mais próximo no range de ataque (2);
- Vida do inimigo mais próximo (3);
- Dano do inimigo mais próximo (3);
- Inimigo que causou mais dano no range de ataque (2);
- Vida do inimigo que mais causou dano (3);
- Dano do inimigo que mais causou dano (3);

O teste realizado foi 2v2, com cada time uma classe “Base” e o outro a classe “Tank” em que um “Base” está mais perto do “Tank” do oponente e vice-versa, como mostra a Figura 30 a seguir:

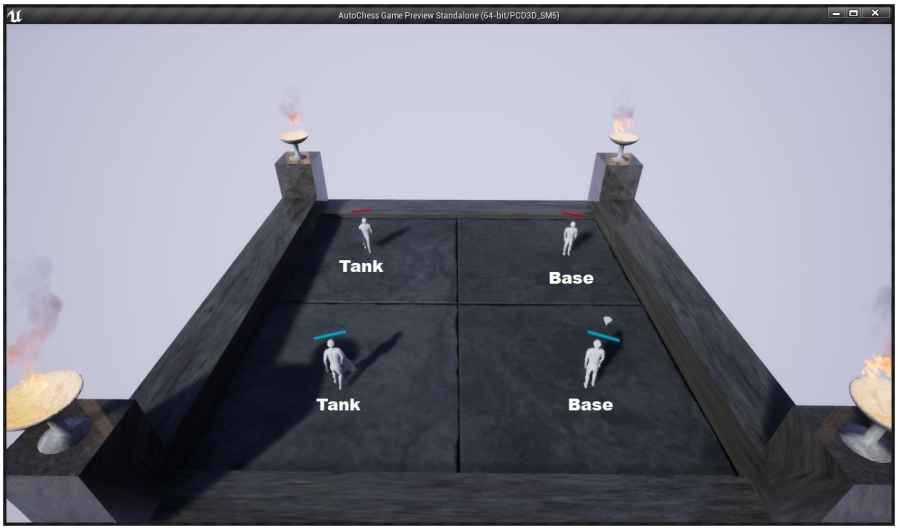
Figura 30. Configuração para o teste da modelagem 2.



Fonte: Autores.

Analisando o vídeo do teste [30], foi possível perceber que o agente foi capaz de aprender que a condição de vitória era movimentar-se e atacar não o inimigo mais perto, mas sim o inimigo que estava causando mais dano à direita. Entretanto, ao inverter a posição dos personagens de um jogador resultando em personagens da mesma classe próximos um do outro como mostra a Figura 30, obtêm-se o resultado do vídeo [31] que mostra que o agente ficou enviesado em questão de configuração do campo do tabuleiro pois ele continua a se movimentar para a direita procurando o personagem que causa mais dano.

Figura 31. Configuração invertida da modelagem 2.



Fonte: Autores.

Isso acontece porque as informações do estado não permitem saber onde o inimigo chave está e as ações de ataque ou movimento não o direciona para este inimigo.

8.3 MODELAGEM 3 - ATAQUE DE OPONENTES CHAVES

Para resolver o problema da falta de informação e ações mais específicas que resultava no enviesamento do agente, foram adicionadas duas novas ações de ataque

à personagens chaves e removidas algumas informações de estados que aparentemente não eram relevantes para tomada de decisão. Resultando em:
Ações possíveis:

- Ficar parado;
- Mover para o Norte (Absoluto);
- Mover para o Sul (Absoluto);
- Mover para o Oeste (Absoluto);
- Mover para o Leste (Absoluto);
- Atacar o inimigo mais próximo;
- Atacar o inimigo mais fraco;
- Atacar o inimigo mais danoso;

Composição do estado (Total 5832 estados possíveis)

- Posição X do agente (3);
- Posição Y do agente (3);
- Vida do agente (3);
- Dano do agente (3);
- Inimigo mais próximo no range de ataque (2);
- Inimigo mais danoso no range de ataque (2);
- Vida do inimigo mais danoso (3);
- Inimigo com menos vida no range de ataque (2);
- Vida do inimigo com menos vida (3).

O teste foi realizado 2v2 com a mesma configuração da Modelagem 2, dois personagens para cada lado, um “Base” e outro “Tank” em que o personagem do oponente mais próximo é de classe diferente.
Analisando o vídeo [32] foi possível verificar que o agente ainda foi capaz de encontrar a condição de vitória de atacar o inimigo que causou mais dano ao invés do inimigo mais próximo. Além disso, diferentemente da modelagem 2, o vídeo [33] apresenta que o agente não ficou enviesado em relação à movimentação para a direita, mesmo

invertendo a posição dos personagens de um lado, o agente movimentou-se para a esquerda para atacar o alvo certo.

8.4 MODELAGEM 4 - DISTÂNCIA DOS OPONENTES CHAVES

Foi adicionado mais informação em relação à distância em que os oponentes chaves se encontram com o intuito de melhorar a tomada de decisão sobre quem atacar e a distância em que se encontra o personagem. Isso resultou na seguinte modelagem:
Ações possíveis:

- Ficar parado;
- Mover para o Norte (Absoluto);
- Mover para o Sul (Absoluto);
- Mover para o Oeste (Absoluto);
- Mover para o Leste (Absoluto);
- Atacar o inimigo mais próximo;
- Atacar o inimigo mais fraco;
- Atacar o inimigo mais danoso;

Composição do estado (Total 19683 estados possíveis)

- Posição X do agente (3);
- Posição Y do agente (3);
- Vida do agente (3);
- Dano do agente (3);
- Distância do Inimigo mais próximo (3);
- Distância do Inimigo mais danoso (3);
- Vida do inimigo mais danoso (3);
- Distância do Inimigo com menos vida (3);
- Vida do inimigo com menos vida (3).

9 MODELAGEM IRL

A modelagem do IRL deve ser capaz de representar, por meio de features, o que o expert pode estar considerando ao tomar decisões durante a demonstração do comportamento esperado. Além disso, deve estar relacionada com a modelagem do RL para ser capaz de recompensar ou punir corretamente o conjunto estado-ação tomado. Com isso em mente, o vetor de features consiste em observar a posição do agente, o tipo de ação (parado, atacando, movimentando) e o tipo de alvo (mais próximo, mais fraco ou maior causador de dano):

- Posição (8)
- Ação (3):
    - Atacando;
    - Movendo;
    - Parado.
- Tipo de alvo (3)
    - Mais fraco;
    - Mais perto;
    - Maior causador de dano.

10 RESULTADOS

10.1 TESTES PROPOSTOS

Tendo o nosso projeto sido (pelo menos dos pontos definidos como "Deve ter" em 5.2.3) implementado e funcionando, são propostos aqui testes para validar se a pessoa consegue de fato fazer e com quanta fidelidade em relação ao esperado, de acordo com a modelagem proposta para os algoritmos de ML.

Tabela 6 - Testes para validar o funcionamento da modelagem do ML proposta

Número	Teste	Resultados esperados	Resultados obtidos
1	Selecionar trajetórias nas quais o jogador ficou sempre parado em uma posição específica	O comportamento faz que o agente não se mova no campo do tabuleiro.	O agente quase não se move e aparenta tentar ir para a posição específica.
2	Selecionar trajetórias nas quais o jogador ficou sempre andando	O comportamento faz que o agente apenas se mova no campo do tabuleiro	O agente se move bastante, mas não visitou todas as regiões demonstradas.
3	Selecionar trajetórias nas quais o jogador sempre atacou o personagem que tinha menos vida no campo do tabuleiro	O comportamento faz que o agente busque atacar as peças inimigas que estão com menos vida	O agente concentra os ataques no personagem com menos vida, mas em certos casos acaba favorecendo a movimentação.
4	Selecionar trajetórias nas quais o jogador sempre atacou o personagem que causou mais dano no campo do tabuleiro	O comportamento faz que o agente busque atacar as peças inimigas que estão causando mais dano no campo do tabuleiro	O agente concentra o ataque no personagem que causou mais dano, mas em certos casos ele favorece se manter na posição demonstrada na trajetória ao invés de atacar.

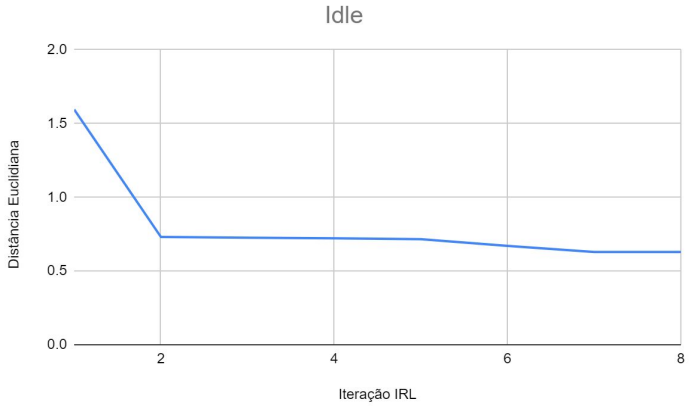
10.2 RESULTADO TESTE 1 - COMPORTAMENTO “Idle”

O vídeo [34] apresenta uma das trajetórias do expert, no geral, elas levam o agente para o canto inferior esquerdo da arena (Tile_0) e o mantém parado até o final do round.

O vídeo [35] apresenta o resultado do treinamento, nele pode-se observar que o agente com comportamento “Idle” apresentado no início do vídeo se mantém parado a maior parte do tempo próximo do canto inferior esquerdo.

A Figura 32 apresenta a distância euclidiana da projeção ortogonal das features do expert sobre o vetor formado entre as últimas features demonstradas pelo agente:

Figura 32 - Distância euclidiana do comportamento “WalkAround”.



Fonte: Autores.

A Tabela 7 apresenta o resultado do treinamento (Note que apenas as features mais interessantes estão sendo apresentadas):

Tabela 7. Resultados do treinamento do comportamento “Idle”

	Tile_0	Tile_1	Tile_2	Tile_3	Idle	Moving	Attacking
μ_E	0.864642	0.064986	0.070371	0.000000	0.866899	0.133101	0.000000
μ	0.431924	0.539640	0.018223	0.010214	0.617833	0.311189	0.070977
w	0.062166	0.065791	-0.099610	-0.029400	0.165129	-0.079870	-0.085250

A partir da Tabela 7, pode-se observar que os pesos favorecem o agente a se manter parado em comparação com se movimentando ou atacando e acaba induzindo ele a ficar entre as posições “Tile_0” e “Tile_1” apesar das features do expert em relação a posição ser majoritariamente “Tile_0”. Isso pode ser explicado pela forma que o algoritmo converge e que a distância euclidiana ainda está relativamente longe do ideal.

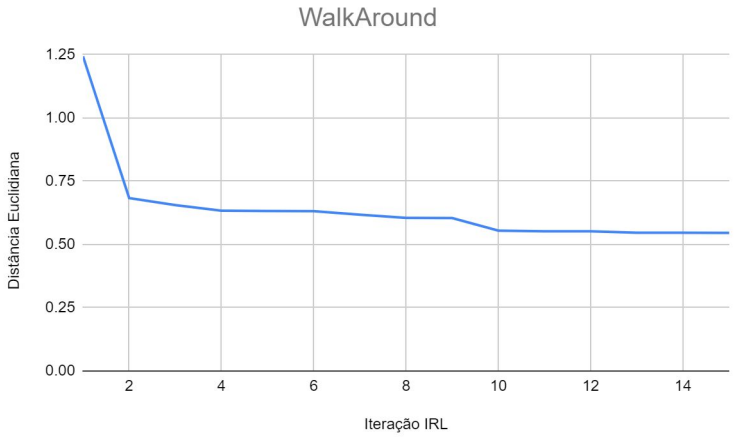
10.3 RESULTADO TESTE 2 - COMPORTAMENTO “WalkAround”

O vídeo [36] apresenta a trajetória que o expert demonstrou para servir como base do treinamento, nela o agente apenas se movimenta percorrendo a parte externa do campo do tabuleiro, evitando o centro.

O vídeo [37] apresenta o resultado do treinamento, note o agente com o comportamento “WalkAround” demonstrado no início, nele é possível perceber que o agente mantém majoritariamente ações de movimento pelo mapa entretanto não chega a percorrer toda a parte externa do campo do tabuleiro.

A Figura 33 apresenta a distância euclidiana da projeção ortogonal das features do expert sobre o vetor formado entre as últimas features demonstradas pelo agente:

Figura 33 - Distância euclidiana do comportamento “WalkAround”.



Fonte: Autores.

A Tabela 8 apresenta o resultado do treinamento (Note que apenas as features mais interessantes estão sendo apresentadas):

Tabela 8. Resultados do treinamento do comportamento “WalkAround”

	Tile_0	Tile_4	Tile_7	Idle	Moving	Attacking
μ_E	0.175936	0.000000	0.130417	0.043330	0.956670	0.000000
μ	0.013268	0.111803	0.481032	0.042041	0.894427	0.063532
w	-0.038340	-0.028360	0.055994	-0.016510	0.092428	-0.075920

Pode-se observar que as features relacionadas à ação que o agente está tomando se aproximam muito da demonstrada pelo expert, entretanto o mesmo não acontece para as features que representam a posição do agente.

Note que os pesos favorecem a ação de movimentação e desfavorecem as outras duas, mas o mesmo não acontece da mesma forma com as features de posição, tomando como exemplo a posição “Tile_0”, nota-se que o valor observado é menor que o valor esperado pois o peso desfavorece essa posição. Isso pode ser explicado pela Figura 33 que mostra que o algoritmo não chegou no valor ideal, então ainda existem ajustes nos pesos para chegar no comportamento demonstrado. Assim, ainda sobre o “Tile_0”, pode-se afirmar que na iteração anterior, o agente se manteve muito mais que o esperado nesta posição e fez o algoritmo penalizar esse comportamento na próxima iteração.

10.4 RESULTADO TESTE 3 - COMPORTAMENTO “FocusWeakest”

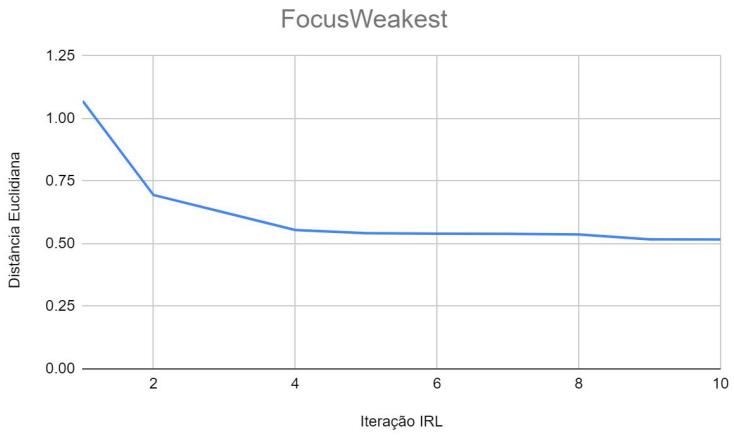
O vídeo [38] apresenta a trajetória demonstrada pelo expert, nele o busca-se atacar o personagem com menos vida, como consequência o agente acaba se posicionando no centro da arena na região em que geralmente acontecem os conflitos.

O vídeo [39] apresenta o resultado do treinamento, nele é possível observar que o agente com o comportamento “FocusWeakest” apresentado no início do vídeo aprendeu a concentrar os ataques no personagem com menos vida ao desviar de outro

personagem para alcançá-lo, entretanto é possível notar ações intercaladas de movimentação e ataque mesmo estando ao lado do personagem alvo.

A Figura 34 apresenta a distância euclidiana da projeção ortogonal das features do expert sobre o vetor formado entre as últimas features demonstradas pelo agente:

Figura 34 - Distância euclidiana do comportamento “FocusWeakest”.



Fonte: Autores.

A Tabela 9 apresenta o resultado do treinamento (Note que apenas as features mais interessantes estão sendo apresentadas):

Tabela 9. Resultados do treinamento do comportamento “FocusWeakest”

	Tile_4	Idle	Moving	Attacking	Damagest	Weakest	Nearest
μ_E	0.413286	0.123053	0.237288	0.639659	0.017840	0.914443	0.067717
μ	0.838522	0.049506	0.164770	0.785724	0.151841	0.484957	0.363202
w	-0.030810	0.006984	-0.097150	0.090169	-0.146500	0.400514	-0.254010

Nota-se que os pesos favorecem mais a feature que representa o ataque em comparação ao agente estar se movimentando ou parado e, dentro do ataque, observa-se que atacar o mais fraco traz uma recompensa muito maior. O peso negativo sobre o “Tile_4” (Posição central do mapa), é reflexo do agente apresentar trajetórias

em que fica muito mais tempo que o esperado, entretanto isso acontece porque os conflitos ocorrem no centro do mapa e o personagem mais fraco vai estar presente nesta localização, assim apesar da recompensa negativa na posição, ela é compensada por atacar o personagem mais fraco.

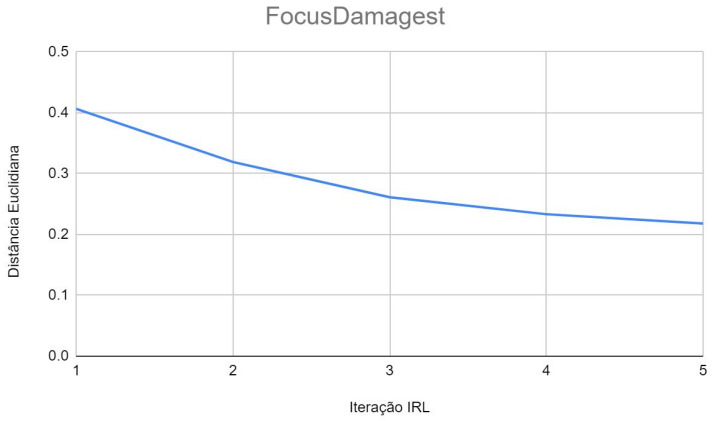
10.5 RESULTADO TESTE 4 - COMPORTAMENTO “FocusDamagest”

O vídeo [40] apresenta a trajetória do expert, nele o expert teve como objetivo priorizar o ataque à personagens inimigos que estivessem causando mais dano, no caso, focando as classes “Carry” > “Base” > “Tank”, como consequência o agente foi mantido no centro da arena onde geralmente ocorrem os conflitos.

O vídeo [41] apresenta o resultado do treinamento, nele é possível observar que o agente com comportamento “FocusDamagest” apresentado no início do vídeo ataca primeiro o personagem da classe “Base” localizado no centro da arena, realiza movimentos de ida e volta entre permanecer no centro da arena ou sair e atacar o “Carry”.

A Figura 35 apresenta a distância euclidiana da projeção ortogonal das features do expert sobre o vetor formado entre as últimas features demonstradas pelo agente:

Figura 35 - Distância euclidiana do comportamento “FocusDamagest”.



Fonte: Autores.

A Tabela 10 apresenta o resultado do treinamento (Note que apenas as features mais interessantes estão sendo apresentadas):

Tabela 10. Resultados do treinamento do comportamento “FocusDamagest”

	Tile_4	Tile_7	Tile_8	Idle	Moving	Attacking	Damagest	Weakest	Nearest
μ_E	0.436898	0.131984	0.156969	0.093232	0.204116	0.702652	0.537065	0.330881	0.132054
μ	0.229266	0.135994	0.189879	0.037026	0.177416	0.785558	0.437855	0.329515	0.232630
w	-0.006060	0.016703	-0.009640	-0.035060	-0.030700	0.065759	0.150684	0.004324	-0.155010

A partir da Tabela 10, pode-se notar que a feature de ataque é priorizada sobre a de movimentação ou ficar parado e, dentre os ataques, atacar o personagem que está causando mais dano possui um peso maior. Em relação ao posicionamento, foi observado um comportamento semelhante ao demonstrado

Assim, o agente de certa forma foi capaz de reproduzir o comportamento do expert, apresentando a menor distância euclidiana dentre os testes.

10.6 DISCUSSÃO E POSSÍVEIS MELHORIAS

De forma geral, com a modelagem de RL e IRL atual, percebe-se que o agente é capaz de replicar comportamentos e posicionamentos mais simples demonstrados pelo expert, como pôde ser visto nos comportamentos “FocusWeakest” e o “Idle” em que, respectivamente, ele concentrou os ataques no personagem mais fraco e durante a maior parte do round se manteve parado no canto inferior esquerdo. Entretanto, em algumas situações é possível observar ações inesperadas tomadas pelo agente, como deixar de atacar o personagem mais fraco para se distanciar dele.

Em relação à comportamentos mais complexos, o agente não foi capaz de replicar posicionamentos variados, como demonstrado na trajetória do “WalkAround” em que o expert visitou a área externa do campo do tabuleiro, mas o comportamento resultante se mantinha próximo da posição inicial.

Durante os testes pôde-se perceber a influência da configuração do campo do tabuleiro que era utilizada para realizar o treinamento, durante a convergência do algoritmo de RL, idealmente, o agente deve visitar todas as possibilidades de estados e ações mais de uma vez para conseguir julgar qual é a melhor ação a ser tomada dado o estado atual. Assim se, por exemplo, o treinamento ocorrer com apenas uma configuração específica, o agente vai visitar um conjunto restrito de situações e isso tem duas implicações importantes, a primeira que pode dificultar a convergência para a trajetória demonstrada pelo expert já que acaba restringindo as situações e ações que o agente pode encontrar e tomar, a segunda é que o comportamento resultante não vai "conhecer" outros cenários e, consequentemente, não vai conseguir tomar a melhor ação na maioria dos estados.

Além disso, nota-se a influência das trajetórias sobre o comportamento resultante, tomando como exemplo o comportamento "FocusDamagest" onde na trajetória apresentada o expert procurava atacar os personagens mais causadores de dano, como os confrontos acontecem majoritariamente na parte central da arena, o agente entendeu que estar no centro da arena tem o seu valor, assim foi possível observar ele na "dúvida" entre permanecer no centro ou sair e atacar o "Carry". Então entende-se que o cenário em que a trajetória é demonstrada influencia no resultado final, sendo importante apresentar mais de uma em diferentes cenários para evitar o problema mencionado, entretanto deve-se tomar cuidado para não apresentar trajetórias contraditórias que podem resultar em um comportamento que tenta fazer tudo e, no final, não está fazendo nada.

Entende-se que para obter melhores resultados seria necessário mais ciclos iterativos de ajustes e testes sobre a modelagem do RL e do IRL. Dentre os ajustes relacionados ao RL que podem melhorar o desempenho do comportamento resultante seria aumentar a granularidade dos estados atuais trazendo um maior número de intervalos de, por exemplo, distância e dano causado, assim a situação atual do agente vai ser menos generalizada e permitirá tomar ações mais específicas para cada uma delas. Outro ponto seria adicionar a informação sobre onde se encontra cada inimigo chave

por meio da posição atual na arena ou a direção relativa do agente, isso pode trazer a correlação entre a direção do movimento e o personagem inimigo. Entretanto as mudanças citadas aumentam o número total de estados possíveis e consequentemente o tempo de treinamento necessário, vale lembrar que o algoritmo de IRL usado necessita de várias iterações convergindo o algoritmo de RL. Pensando em um jogo comercial em que o jogador pode demonstrar o comportamento e depois usá-lo, o tempo de treinamento de um dia seria o máximo aceitável, mas no estado atual do sistema utilizado, o tempo de treinamento já está próximo deste limite. Sobre as melhorias no IRL seria interessante aumentar o número de episódios que rodam para convergir o algoritmo de RL (atualmente usa-se 800 episódios), acrescentar features que representam a quantidade de dano causado e sofrido e por fim, procurar apresentar mais de uma trajetória demonstrando o comportamento em diferentes situações do campo do tabuleiro e também adicionar estas diferentes situações nos cenários de treinamento do RL.

11 CONCLUSÃO

Tendo em vista o estado atual do projeto e sua implementação, bem como dos resultados obtidos e de sua análise, as seguintes conclusões podem ser tiradas.
Primeiramente, é apresentada a Tabela 11, indicando a completude dos requisitos que haviam sido levantados para o projeto:

Tabela 11 - Tabela de completude dos requisitos do projeto

Requisito	Tipo	Prioridade	Estado atual
O jogador deve ser capaz de treinar as peças	Funcional	Deve ter	Feito
O jogador deve ser capaz de armazenar comportamentos	Funcional	Deve ter	Feito
Cada classe de peça deve possuir o próprio comportamento	Funcional	Deveria ter	Não feito
O jogador deve poder simular o round de uma partida	Funcional	Deve ter	Feito
O jogador deve poder conseguir realizar uma partida de autochess	Funcional	Deveria ter	Não feito
O jogador deve conseguir jogar contra 3 níveis diferentes de dificuldade	Funcional	Poderia ter	Não feito
O jogador deve ser capaz de confrontar peças de outros jogadores	Funcional	Deveria ter	Não feito
O jogador deve ser capaz de jogar online	Funcional	Poderia ter	Não feito
O aprendizado do comportamento não deve impactar na experiência do jogador	Não funcional	Deveria ter	Não feito

(Continua na próxima página)

(Conclusão da tabela 11)

Requisito	Tipo	Prioridade	Estado atual
O jogador deve ser capaz de armazenar vários comportamentos	Não funcional	Deveria ter	Feito
O jogo deve rodar em máquinas comuns do perfil médio dos jogadores	Não funcional	Deveria ter	Não feito
A interface de treinamento deve ser de fácil utilização	Não funcional	Deve ter	Feito

Observando-se a Tabela 11, percebe-se que foi possível implementar todos os requisitos que haviam sido priorizados como “Deve ter”. Assim, o grupo conseguiu cobrir todos os pontos que foram considerados como sendo os mais essenciais do projeto, permitindo que os seus objetivos principais pudessem ser alcançados de forma satisfatória.

Observando-se os resultados obtidos com os testes realizados, percebe-se que está sendo de fato possível que o jogador realize, até certo nível, o treinamento de novos comportamentos através dos algoritmos e modelagem atuais. Assim, verifica-se que os algoritmos de ML escolhidos e implementados foram feitos de forma adequada. Todavia, como também já mostrado, para comportamentos mais sofisticados e elaborados, o sistema não foi capaz de reproduzir de forma tão satisfatória o que foi demonstrado pelo jogador. Assim, seria necessário que fossem realizadas outras abordagens ou pequenas mudanças na modelagem atual e a utilização de otimizações ao algoritmo atual, visando tentar melhorar esse quadro.

Considerando-se o exposto, o grupo considera que os resultados obtidos com o projeto são promissores: um sistema para poder treinar peças pôde ser desenvolvido, com uma interface que ajuda o usuário a utilizar corretamente o sistema e que permite, até certo nível, que peças possam ser treinadas. Caso melhorias nos algoritmos utilizados sejam feitas a fim de otimizar o treinamento, para que ele convirja em um

comportamento cada vez mais fiel ao demonstrado, o projeto poderia ter diversas aplicações na área de jogos, por exemplo:

- Possível extensão dos jogos atuais do gênero AutoChess, ao permitir com a criação dessa plataforma que os próprios usuários criem comportamentos para serem utilizados dentro do jogo, aumentando assim o número de possibilidades de estratégias presentes no jogo.
- Nova forma de programar e desenvolver o comportamento não só de personagens de jogos do gênero AutoChess, mas de qualquer jogo que exija que seus personagens tenham algum comportamento controlado por AI, ao permitir definir e desenvolver o comportamento utilizando os algoritmos de RL e IRL apresentados.
- Permitir que um jogador possa simular com maior fidelidade uma partida com alguma pessoa específica: ao permitir que pessoas treinem o comportamento do personagem controlado por alguma forma de AI, sem exigir que ela tenha conhecimento algum sobre computação, permite que o comportamento de personagens dentro de um jogo se assemelhe ao de um jogador profissional específico que realize o treinamento do comportamento, por exemplo. Assim, qualquer pessoa jogando o jogo pode acabar tendo a experiência de como seria jogar com ou contra esse jogador. Não só isso, poderia servir como uma ótima ferramenta inovadora de treinamento para jogadores competitivos, ao permitir que treinem jogar contra algumas outras pessoas específicas e estudar suas ações e estratégias, por exemplo.

Além disso, fora da área de jogos, o projeto também apresenta o ponto de que pode servir como primeiro contato ao público sobre o funcionamento de algoritmos de ML e AI, uma vez que mostra o seu funcionamento, explica brevemente o processo e permite que seja utilizado sem exigir nenhuma experiência prévia da pessoa. Assim, seria ótimo para pessoas que estão querendo aprender e ter um primeiro contato com RL e IRL ao ver na prática como funcionam ou para expor ao público em geral uma possível aplicação do ramo de AI em produtos da atualidade, mostrar como tais conceitos

podem estar mais próximos e presentes no seu cotidiano do que se imagina e como, uma vez que o sistema permite isso (como no caso do projeto, em que não é exigida nenhuma experiência prévia da pessoa com ele), podem ser utilizados amplamente por qualquer pessoa (no caso do projeto, qualquer pessoa poderia realizar o treinamento de peças utilizando esses algoritmos), por exemplo.

Assim, conclui-se como o projeto, caso desenvolvido futuramente, pode vir a ser importante e valoroso, tendo diversas aplicações na atualidade.

12 PRÓXIMOS PASSOS

Finalmente, são apresentados a seguir pontos e tópicos para possíveis extensões e continuações do projeto:

- Realizar as melhorias de algoritmo e modelagem sugeridas na seção de resultados.
- Finalização da implementação dos requisitos levantados que não se conseguiu implementar no projeto.
- Refatoração e melhorias no código atual. No código do projeto existem diversos comentários que começam com “Note:”, alguns deles contendo possíveis trechos de códigos candidatos a serem modificados a fim de torná-los mais otimizados e elegantes. Vale notar que esses não são os únicos trechos de código que podem ser melhorados. Qualquer trecho e parte do projeto que não estão ótimos poderiam ser candidatos a serem otimizados.
- Melhorar a estética do projeto: sabe-se, por exemplo, que podem ser encontrados diversos assets para serem utilizados em projetos (dado os devidos cuidados com licenças) na própria loja da EpicGames™ ou até mesmo na internet. Eles poderiam ser incluídos para substituir por exemplo os modelos atuais dos Personagens, acrescentar um cenário no level do jogo e do menu principal, etc. Com isso, o projeto ganharia uma aparência melhor e teria um acabamento melhor nesse aspecto, agregando mais valor.
- Melhorar a experiência do usuário com a interface atual: um dos requisitos levantados e importantes do projeto é que o sistema seja de fácil uso para as pessoas, não exigindo que tenha nenhuma experiência anterior para utilizá-lo. Assim, testes com usuários podem ser feitos para coletar feedback sobre como é sua experiência com o sistema da forma que está hoje, possíveis pontos de melhoria, quais foram suas dores ao utilizá-lo, bem como pontos positivos etc. Com esses resultados e os tomando como base, mudanças podem ser feitas a fim de tornar a interface mais adequada ao usuário final.

- Melhora nos algoritmos de ML empregados no projeto: uma das possibilidades seria a utilização de Redes Neurais Artificiais para realizar os treinamentos dos novos comportamentos. Caso se consiga chegar em resultados em que os comportamentos possam ser treinados para agirem de forma bem próxima do demonstrado, em tempos não muito longos que acabem impactando na experiência da pessoa e em máquinas comuns do perfil médio dos jogadores, as aplicações citadas anteriormente na área de jogos poderiam ser estudadas para serem adotadas na indústria de jogos, em produtos comerciais.
- Adoção do projeto como primeiro contato com AI, com fins pedagógicos: como descrito anteriormente, o projeto também pode servir como bom primeiro contato para pessoas aprendendo sobre os algoritmos de ML nele utilizados ou para mostrar às pessoas um exemplo de aplicação utilizando AI, como esse campo pode estar mais presente no seu cotidiano e ao seu alcance para ser utilizado do que parece.

REFERÊNCIAS

[1] ZHANG, Z. et. al. **Hierachical Reinfocement Learning for Multi-agent MOBA Game.** 6st version. [S. I.: s.n.], 2019.

[2] YE, D. et. al. **Mastering Complex Control in MOBA Games with Deep Reinforcement Learning.** 2nd version. China: [s.n.], 2020.

[3] DAROS, V. K. **Piloto Baseado em Aprendizagem por Reforço para o Simulador de Corridas TORCS.** Versão 1. São Paulo: [s.n.], 2015.

[4] COSTA, L. E. R.; GONÇALVES, D. H. **Alocação Dinâmica de Recursos em Rede Definida por Software utilizando Aprendizado de Máquina.** Versão 1. São Paulo: [s. n.], 2019.

[5] NG, A. Y.; RUSSEL S. **Algorithms for Inverse Reinforcement Learning**. 1st version. Berkeley: [s.n.], 2002.

[6] ABBEEL, P.; NG, A. Y. **Apprenticeship Learning via Inverse Reinforcement Learning**. 1st version. Stanford: [s.n.], 2004.

[7] MESSER, O. **The Use of Apprenticeship Learning Via Inverse Reinforcement Learning for Generating Melodies.** 1st version. Atenas: [s.n.], 2014.

[8] BHATT, Shweta. Reinforcement Learning 101. **Towards Data Science**, 2018. Disponível em: <https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292>. Acesso em: 03 de set. de 2020.

[9] SINGH, Anubhav. Introduction to Reinforcement Learning. **Datacamp**, 2018. Disponível em: <https://www.datacamp.com/community/tutorials/introduction-reinforcement-learning>. Acesso em: 10 de set. de 2020.

[10] ASHRAF, Mohammad. Reinforcement Learning Demystified: Markov Decision Processes (Part 1). **Towards Data Science**, 2018. Disponível em: <https://towardsdatascience.com/reinforcement-learning-demystified-markov-decision-processes-part-1-bf00dda41690>. Acesso em: 03 de set. de 2020.

[11] GREAVES, Josh. Understanding RL: The Bellman Equations. **Josh Greaves**, 2017. Disponível em: <https://joshgreaves.com/reinforcement-learning/understanding-rl-the-bellman-equations/>. Acesso em: 03 de set. de 2020.

[12] SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning. An Introduction.** 2nd Edition. Cambrigde: The MIT Press, 2018.

[13] SALLOUM, Ziad. Policy Based Reinforcement Learning, the Easy Way. **Towards Data Science**, 2019. Disponível em: <https://towardsdatascience.com/policy-based-reinforcement-learning-the-easy-way-8de9a3356083>. Acesso em: 10 de set. de 2020.

[14] SALLOUM, Ziad. Reinforcement Learning Policy for Developers. **Towards Data Science**, 2020. Disponível em: <https://towardsdatascience.com/revisiting-policy-in-reinforcement-learning-for-developers-43cd2b713182>. Acesso em: 10 de set. de 2020.

[15] SALLOUM, Ziad. Monte Carlo in Reinforcement Learning, the Easy Way. **Medium**, 2018. Disponível em: <https://medium.com/@zsalloum/monte-carlo-in-reinforcement-learning-the-easy-way-564c53010511>. Acesso em: 10 de set. de 2020.

[16] ROY, Baijayanta. Monte Carlo Learning. **Towards Data Science**, 2019. Disponível em: <https://towardsdatascience.com/monte-carlo-learning-b83f75233f92>. Acesso em: 10 de set. de 2020.

[17] VIOLANTE, Andre. Simple Reinforcement Learning: Q-learning. **Towards Data Science**, 2019. Disponível em: <https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56>. Acesso em: 10 de set. de 2020.

[18] ADL. An introduction to Q-Learning: reinforcement learning. **freeCodeCamp**, 2018. Disponível em: <https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/>. Acesso em: 10 de set. de 2020.

[19] HUANG, Kung-Hsiang. Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG). **Towards Data Science**, 2018. Disponível em: <https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287>. Acesso em: 10 de set. de 2020.

[20] GUPTA, Alind. SARSA Reinforcement Learning. **GeeksforGeeks**, 2019. Disponível em: <https://www.geeksforgeeks.org/sarsa-reinforcement-learning/>. Acesso em: 10 de set. de 2020.

[21] SALLOUM, Ziad. Eligibility Traces in Reinforcement Learning. **Towards Data Science**, 2019. Disponível em:

<https://towardsdatascience.com/eligibility-traces-in-reinforcement-learning-a6b458c019d6>. Acesso em: 10 de set. de 2020.

[22] RUSSEL, S.; **Learning agents for uncertain environments (extended abstract).** 1st version. Berkeley: [s.n.], 1998.

[23] Introduction to Blueprints. **Unreal Engine**, c[entre 2004 e 2020]. Disponível em: <https://docs.unrealengine.com/en-US/Engine/Blueprints/GettingStarted/index.html>. Acesso em: 19 de nov. de 2020.

[24] Starter Content. **Unreal Engine**, c[entre 2004 e 2020]. Disponível em: <https://docs.unrealengine.com/en-US/Engine/Content/Packs/index.html>. Acesso em: 19 de nov. de 2020.

[25] Epic Games. **Unreal Engine**, [entre 2004 e 2020]. Unreal Engine 4 Documentation. Disponível em: <https://docs.unrealengine.com/en-US/index.html>. Acesso em: 19 de nov. de 2020.

[26] Epic Games. **Unreal Engine**, [entre 2009 e 2019]. UE4 AnswerHub. Disponível em: <https://answers.unrealengine.com/index.html>. Acesso em: 19 de nov. de 2020.

[27] Mathew Wadstein. **YouTube**, [entre 2011 e 2020]. Mathew Wadstein - YouTube. Disponível em: <https://www.youtube.com/channel/UCOVfF7PfLbRdVEm0hONTrNQ>. Acesso em: 19 de nov. de 2020.

[28] GONFALONIERI, A. Inverse Reinforcement Learning. **Towards DataScience**. 2018. Disponível em: <https://towardsdatascience.com/inverse-reinforcement-learning-6453b7cdc90d>. Acesso em: 22 de nov. de 2020.

[29] Autores. **Modelagem 1**, 2020. Disponível em: <https://drive.google.com/file/d/1wD-roqgP_1waXckBIhSrgwjOM6rZtaj7/view?usp=sharing>. Acesso em: 20 de out. de 2020.

[30] Autores. **Modelagem 2.** 2020. Disponível em: <https://drive.google.com/file/d/1BoyjafWWjFpxkuj2Dhu7TIFN0wD099Bm/view?usp=sharing>. Acesso em: 20 de out. de 2020.

[31] Autores. **Modelagem 2 - problema.** 2020. Disponível em: <https://drive.google.com/file/d/1Pjt1rVISF26smIxRPh0mvKfQjDkhbAHZ/view?usp=sharing>. Acesso em: 20 de out. de 2020.

[32] Autores. **Modelagem 3**. 2020. Disponível em: <https://drive.google.com/file/d/1sZmkL3R58W0CODjc8tKQFTRHXh2IYArP/view?usp=sharing>. Acesso em: 20 de out. de 2020.

[33] Autores. **Modelagem 3 - resolução.** 2020. Disponível em: <https://drive.google.com/file/d/1zH4FDPoLBDIwmq_wMgCt_wFNo-xBoxVz/view?usp=sharing>. Acesso em: 20 de out. de 2020.

[34] Autores. **Idle - TrajectoryCenter.** 2020. Disponível em: <https://drive.google.com/file/d/1A1K9_6STwI5IuJ6VMxp_Rly7D83zEygp/view?usp=sharing>. Acesso em: 28 de nov. de 2020.

[35] Autores. **Idle - Center.** 2020. Disponível em: <https://drive.google.com/file/d/1czsBWi6y7GnbN37MgWOHNl91h3j6N2i4/view?usp=sharing>. Acesso em: 28 de nov. de 2020.

[36] Autores. **WalkAround - Trajectory.** 2020. Disponível em: <https://drive.google.com/file/d/1VAzTsiEcPH8NPly5TFi_o_Xa7xrTgLrV/view?usp=sharing>. Acesso em: 28 de nov. de 2020.

[37] Autores. **WalkAround - Center.** 2020. Disponível em: <https://drive.google.com/file/d/1pxYuhaNApRle4DKuuBsEhZJbZzOFescb/view?usp=sharing>. Acesso em: 28 de nov. de 2020.

[38] Autores. **FocusWeakest - Trajectory.** 2020. Disponível em: <https://drive.google.com/file/d/1JGd-M8FPILrIksNbHP_quWUsmTL0zrlt/view?usp=sharing>. Acesso em: 29 de nov. de 2020.

[39] Autores. **FocusWeakest - Right.** 2020. Disponível em: <https://drive.google.com/file/d/10GHXG0hCjzG90cxxCTFb-RiPpa-Qj-Hx/view?usp=sharing>. Acesso em: 29 de nov. de 2020.

[40] Autores. **FocusDamagest - Trajectory.** 2020. Disponível em: <https://drive.google.com/file/d/1H5zrWrMszto8faXdMI9IQJvYeGLWZe83/view?usp=sharing>. Acesso em: 29 de nov. de 2020.

[41] Autores. **FocusDamagest - Left.** 2020. Disponível em: <https://drive.google.com/file/d/1Or7eM27EM1HpQaxPJwJav5iWlK7bvYah/view?usp=sharing>. Acesso em: 29 de nov. de 2020.

GLOSSÁRIO

As informações a seguir foram retiradas da documentação oficial da UE4, do canal da Unreal Engine no YouTube, do site Couch Learn e do livro The Art of Game Design: A Book of Lenses, por Jesse Schell.

Actor - Na UE4, qualquer objeto que pode ser posicionado em um level.

Asset - Na UE4, um pedaço, uma porção de conteúdo de um projeto na Unreal Engine.

Controller - Na UE4, elemento que pode ser usado para definir o comportamento de um objeto da classe Pawn (ou classes derivadas), servindo para controlá-lo.

Data Table - Na UE4, uma tabela contendo informações diversas mas relacionadas e agrupadas de uma forma significativa e útil.

Enum (Enumeration) - Na UE4, um tipo de variável que permite acesso à listas customizadas de configurações ou estados.

Game Design - Ato de decidir como um jogo deve ser. Nesse processo são definidos aspectos como, dentre outros exemplos, a história do jogo, suas regras, como ele é visualmente, o ritmo do jogo, recompensas e tudo mais relacionado à experiência do jogador ao jogar o jogo.

Game Instance - Na UE4, uma classe de gerenciamento que não é destruída quando o jogo muda de level, permitindo que informações possam ser armazenadas e passadas entre levels.

Game Mode - Na UE4, uma classe que auxilia no gerenciamento de informações sobre o jogo sendo jogado, tendo informações sobre as regras do jogo.

Game State - Na UE4, uma classe que auxilia no gerenciamento de informações sobre o jogo sendo jogado, ao permitir armazenar informações atuais do jogo, permitindo que sejam compartilhadas e atualizadas para todos os jogadores.

Interface - Na UE4, classes úteis para garantir que um conjunto de classes que potencialmente não têm relação entre si implementem um conjunto de funcionalidades em comum.

Level - No contexto de vídeo games, todo objeto que se vê ou que se interage está em um elemento que é conhecido como Level. Na UE4, um Level é composto de uma

coleção de elementos como Static Meshes, Luzes e mais trabalhando em conjunto para trazer a experiência desejada ao jogador.

Skeletal Mesh - Na UE4, objetos compostos de duas partes: polígonos que compõem a superfície desse objeto e uma hierarquia de ossos (bones) interconectados que podem ser usados para animar os vértices dos polígonos.

Static Mesh - Na UE4, geometria estática composta por um conjunto de polígonos que pode ser armazenada e renderizada.

Material - Na UE4, um asset que pode ser aplicado à uma mesh para controlar o aspecto visual de uma cena, ao permitir definir o tipo de superfície do objeto, seu brilho, cor, opacidade, como a luz interage com a superfície etc.

Particle System - Na UE4, asset para definir algum efeito com partículas.

Partícula - Na UE4, um ponto no espaço que será instanciado com comportamentos específicos.

Pawn - Na UE4, classe base de todos os Actors que podem ser controlados por um jogador ou inteligência artificial.

Save Game - Na UE4, uma classe que permite salvar e carregar informações salvas do jogo.

Struct - Na UE4, uma coleção de diferentes tipos de dados que são relacionados e ficam em um único local para fácil acesso.

Textura - Na UE4, Imagens que são utilizadas nos Materials.

Widget Blueprint - Na UE4, asset que pode ser usado para ficar encarregado de mostrar elementos de UI, podendo ser usado para se editar o layout deles, bem como para realizar o script de suas funcionalidades.