UNIVERSIDADE ESTADUAL DO SUDOESTE DA BAHIA (UESB)
DEPARTAMENTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS (DCET)
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

IGOR VIEIRA DE SOUZA

AMBIENTE DE SIMULAÇÃO BASEADO NO JOGO SUPER BOMBERMAN PARA ENSINO E PESQUISA EM INTELIGÊNCIA ARTIFICIAL

VITÓRIA DA CONQUISTA- BA
2016

IGOR VIEIRA DE SOUZA

AMBIENTE DE SIMULAÇÃO BASEADO NO JOGO SUPER BOMBERMAN PARA ENSINO E PESQUISA EM INTELIGÊNCIA ARTIFICIAL

Monografia apresentada ao Colegiado do Curso de Ciência da Computação (CCCC) como pré-requisito para obtenção do Grau de Bacharel em Ciência da Computação pela Universidade Estadual do Sudoeste a Bahia (UESB).

Área de Concentração: Inteligência Artificial e Jogos

Orientador: Prof. Dr. Fábio Moura Pereira

VITÓRIA DA CONQUISTA- BA
2016

UNIVERSIDADE ESTADUAL DO SUDOESTE DA BAHIA – UESB

IGOR VIEIRA DE SOUZA

AMBIENTE DE SIMULAÇÃO BASEADO NO JOGO SUPER BOMBERMAN PARA ENSINO E PESQUISA EM INTELIGÊNCIA ARTIFICIAL

Monografia apresentada ao Colegiado do Curso de Ciência da Computação (CCCC) como pré-requisito para obtenção do Grau de Bacharel em Ciência da Computação pela Universidade Estadual do Sudoeste a Bahia (UESB).

Área de Concentração: Inteligência Artificial e Jogos

Orientador: Prof. Dr. Fábio Moura Pereira

Vitória da Conquista, ___/___/___

BANCA EXAMINADORA

_____

Profª. Dra. Alzira Ferreira da Silva
Doutora em Engenharia Elétrica pela UFRN
Professor (a) Adjunto da UESB

_____

Prof. Dr. Fábio Moura Pereira
Doutor em Ciência da Computação pela UFPE
Professor Adjunto da UESB
(Orientador)

_____

Prof. Dr. Roque Mendes Prado Trindade
Doutor em Engenharia Elétrica e de Computação pela UFRN
Professor Adjunto da UESB

AGRADECIMENTOS

Agradeço a Deus por ter me concedido a oportunidade de ingressar no curso de Bacharelado em Ciência da Computação, na UESB, e por ter me dado saúde e força para enfrentar cada obstáculo. A todos os meus professores, em específico, a Fábio Moura (o meu orientador) e Alzira Ferreira (minha coorientadora) por todo o apoio, pelos conselhos e pelos ensinamentos que contribuíram muito para a minha formação enquanto profissional de Tecnologia da Informação. Aos meus pais, Eremir Vieira e Reginaldo de Souza, pela força que me deram, pelo grande incentivo, broncas e ensinamentos. A minha irmã, Marianne Vieira, que todos os dias me conferia muito apoio. A minha namorada, Laíse Gonçalves, que sempre esteve ao meu lado me apoiando nos momentos mais difíceis. Aos meus primos Hevérton Sales e Helder Medrado pelo companheirismo e pelo auxílio e também aos meus amigos Helber Marinho, Victor Willer e Leandro Gonçalves. Aos meus colegas de classe e, com certeza, futuros profissionais. E, finalmente, a todas as pessoas que estiveram ao meu lado, me proporcionando um convívio proveitoso e alegre no decorrer da minha graduação e da minha vida.

RESUMO

Um dos principais problemas envolvendo o processo de ensino-aprendizagem na área da Inteligência Artificial (IA), consiste no fato dos discentes possuírem uma compreensão superficial acerca dos algoritmos e técnicas apresentadas em sala de aula, devido a falta de exercitá-los na prática. Diante disto, neste trabalho, fizemos uma breve descrição das principais características de agentes inteligentes e ambientes de simulação, e abordamos como os jogos podem assumir um caráter "sério" e serem utilizados como uma ferramenta para tornar mais atraente o aprendizado dos discentes. Além disso, propomos a criação um sistema que seja fácil de programar e permita aos discentes realizar experimentos práticos com os conteúdos vistos em sala de aula, apresentando a eles os mecanismos necessários para que se preocupem unicamente com a implementação de seus algoritmos e técnicas de IA. Para tanto, desenvolvemos o jogo programável *Bomberman X*, a fim de apresentar um ambiente programável de simulação e testes que possa ser explorado como uma ferramenta para estudo, análise e aplicação de técnicas na área de IA.

Palavras-chave: Inteligência Artificial; Jogos Sérios; Agentes Inteligentes; Ambientes de Simulação; *Bomberman X.*

ABSTRACT

One of the main problems involving the process of teaching and learning in the field of Artificial Intelligence (AI), it is the fact that the students possess a superficial understanding of algorithms and techniques presented in class, due to lack of exercise them in practice. In view of this, in this paper, we made a brief description of the main characteristics of intelligent agents and simulation environments, and we approach how the games can take a "serious" character and be used as a tool to become more attractive the learning of students. In addition, we propose to create a system that is easy to program and enable students carry out practical experiments with the contents seen in the classroom, presenting them with the necessary mechanisms to be concerned solely with the implementation of their algorithms and AI techniques. Therefore, We develop the programmable game Bomberman X, in order to introduce a programmable setting simulation and testing that can be exploited as a tool for study, analysis and application techniques in AI area.

Keywords: Artificial Intelligence, Serious Games, Intelligent Agent, Simulation Environment, Bomberman X.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Os jogos digitais representam, atualmente, um grande mercado econômico com faturamento mundial de bilhões de dólares. Segundo dados apresentados pela consultoria *PricewaterhouseCoopers*[1], no ano de 2010, foram gastos cerca US$ 57 bilhões em jogos e consoles ao redor do mundo. Superando, até mesmo, o faturamento de US$ 31,8 bilhões que o cinema obteve no mesmo ano, tornando-se, dessa forma, o mercado com o maior faturamento na área de entretenimento. Só em 2014, no Brasil, o Grupo de Estudos e Desenvolvimento da Indústria de Games[2] (GEDIGAMES) afirma que este mercado já esteja próximo dos US$ 3 bilhões.

Um dos fatores que colaboraram para o crescimento da importância dos jogos digitais, segundo a GEDIGAMES (2014, p.6), é que atualmente eles não são consumidos apenas por jovens e adultos do sexo masculino, mas por crianças, mulheres e idosos. Outro fator importante, é que, hoje em dia, os jogos digitais vão muito além de somente proporcionar diversão e entretenimento aos jogadores. É cada vez mais comum, vermos jogos que simulam situações práticas do dia a dia com o objetivo de proporcionar treinamento de profissionais ou de conscientizar uma comunidade.

Conforme afirma Liliane S. Machado et al. (2009, p.2), jogos como estes que extrapolam a ideia de entretenimento e buscam oferecer outros conteúdos ao jogador são identificados como *Serious Games* ou Jogos Sérios. Neles, o entretenimento, que é uma característica dos jogos digitais, transforma-se em uma ferramenta para passar o conhecimento de uma determinada área. Tornando, assim, esta passagem do conhecimento mais atraente, pois ao mesmo tempo em que é feita a construção de conceitos também é realizada a estimulação de funções psicomotoras. Isto possibilita que estes jogos possam ser empregados em diversas áreas e com as mais diversas finalidades, ou seja, em atividades educacionais, pesquisas científicas, treinamentos de âmbito corporativo, militar ou médico, campanhas publicitárias ou de conscientização, entre outros. Eles podem ser empregados também na área de Inteligência Artificial, já que, de acordo *Zak Middleton* (2002, p.1), a recente evolução do *hardware* utilizado nos jogos, liberou poder de processamento dos módulos gráficos para outros módulos como, por exemplo, o de inteligência. Possibilitando, dessa forma, desenvolver novas técnicas de Inteligência Artificial e estudar as já existentes.

Além disso, como afirma Victor K. Tatai (2002, p.2), os jogos digitais apresentam-se como um excelente plataforma para teste e validação de novas metodologias e algoritmos, pois possuem a riqueza e a complexidade de ambientes sofisticados, o que os tornam sistemas perfeitos para estudos ligados a área de IA (Inteligência Artificial). Afinal, eles proporcionam ambientes

1 Para mais informações acesse www.pwc.com.
2 Para mais informações sobre acesse www.abragames.org/downloads.html.

controlados que podem simular situações adaptadas do mundo real, tornando possível testar procedimentos que talvez não estejam prontos para um ambiente real.

Levando-se em consideração tais aspectos, o presente projeto visa utilizar o jogo programável *Bomberman* X, a fim de apresentar um ambiente de simulação e testes que possa ser explorado como uma ferramenta para fins acadêmicos (ensino e pesquisa) na área de Inteligência Artificial.

1.1 Motivação

De acordo com Fábio Pereira (2014, p.3), um dos principais problemas relacionados ao processo de ensino e aprendizagem na área de Inteligência Artificial acontece porque os discentes possuem um entendimento superficial do funcionamento de algoritmos e técnicas apresentadas em sala de aula, devido a falta de visualização do funcionamento destes na prática. Outro ponto que o autor aborda, é a inexistência de um ambiente padronizado no qual os discentes possam focar apenas na implementação de algoritmos sem ter que se preocupar em implementar a infraestrutura necessária para executá-los.

Sob esta ótica, a motivação para o desenvolvimento do *Bomberman* X vem da necessidade da criação de um sistema que permita aos discentes realizar experimentos práticos com os assuntos vistos em sala de aula, assim como fornecer a eles um ambiente de fácil programação e entendimento que apresente as ferramentas necessárias para somente se preocupar com a implementação de seus algoritmos e técnicas de IA. Outra questão que visamos resolver com *Bomberman* X, é em como tornar a experiência prática do discente mais agradável e atraente. Tendo isto em vista, buscamos criar este sistema nos moldes de um jogo sério. Dessa forma, este ambiente atenderá tanto as questões de ensino e aprendizagem quanto as envolvendo questões de estimulação, já que os jogos possuem apetrechos visuais e dinâmicos que possibilitam isso.

1.2 Objetivos

Os objetivos deste trabalho resumem-se da seguinte forma:

- Objetivo geral: Propor uma ferramenta para o estudo, análise e aplicação de técnicas de IA em jogos, que possa ser utilizado como um ambiente de ensino/aprendizagem de IA.
- Objetivos específicos:

- Analisar diferentes características de agentes inteligentes e de ambientes de simulação em IA;
- Identificar características de diferentes ambientes de simulação de IA, suas qualidades e limitações;
- Identificar as técnicas mais utilizadas no desenvolvimento de IA e propor quais delas melhor podem ser utilizadas como introdutórias do assunto;
- Modelar e implementar uma ferramenta de simulação a partir de um ambiente conhecido e que permita fácil aprendizado;
- Avaliar a ferramenta desenvolvida neste trabalho através da aplicação de técnicas de IA no desenvolvimento do mecanismo de raciocínio dos agentes presentes no sistema.

2. INTELIGÊNCIA ARTIFICIAL E JOGOS

Neste capítulo, abordaremos em específico o que é a Inteligência Artificial e como ela é utilizada em Jogos Digitais. Além disso, apresentaremos os conceitos de agentes inteligentes, mecanismos de raciocínio, ambientes de simulação, Jogos Sérios e algumas técnicas de tomada de decisão.

2.1 O que é Inteligência Artificial?

A inteligência artificial corresponde a uma área da computação que segundo *Russell* e *Norvig* (2003, p.1) "tenta não só compreender, mas também criar entidades inteligentes.". Tais entidades inteligentes visam simular a execução de tarefas que envolvem o raciocínio e a reflexão que os seres humanos e os animais são capazes de realizar. Entre essas tarefas, estão atividades que para nós, seres humanos, são consideradas triviais e que, até mesmo, lidamos diariamente com muita facilidade. Porém, para os computadores são extremamente complexas e desafiadoras de serem realizadas como, por exemplo, atividades que visam reconhecer faces familiares ou planejar atividades.

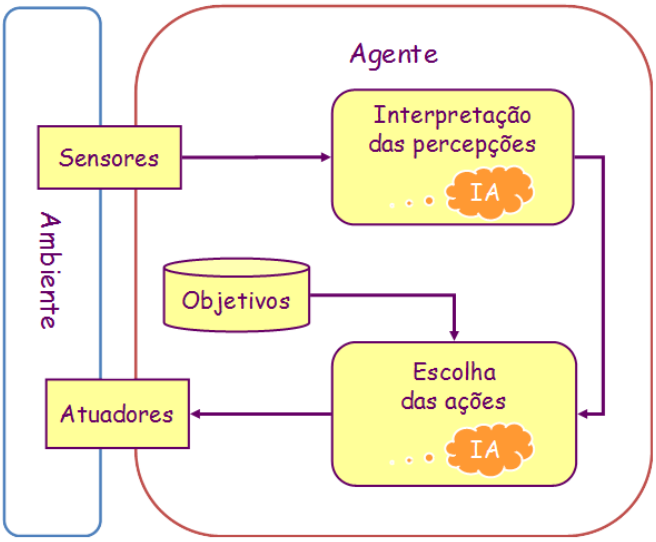
Na área acadêmica, para *Ian Millington* e *John Funge* (2009, p.4), o núcleo da IA é constituído por três segmentos de raciocínio: o filosófico, que busca compreender a natureza do pensamento e da inteligência humana e a partir disso construir um *software* para modelar como o pensamento pode funcionar; o psicológico, que procura entender como funciona os mecanismos do cérebro humano; e o de engenharia, que busca construir algoritmos para realizar tarefas feitas pelos seres humanos. Ainda de acordo esses autores, estes três segmentos e seus desdobramentos são os responsáveis por formar diferentes subcampos que constituem a IA.

Outro ponto que *Ian Millington* e *John Funge* (2009, p.5) abordam é que a IA pode ser dividida e simplificada em três períodos: os primeiros dias, a era simbólica e a era moderna. Os primeiros dias são o período que engloba o surgimento das primeiras questões filosóficas ligadas a IA e a chegada dos primeiros computadores programáveis, assim como o surgimento de alguns dos pioneiros da IA e da área de computação. A era simbólica é o período entre o fim da década 1950 e o início da década 80, no qual os sistemas conhecidos como simbólicos ganharam impulsão. Já a era moderna, engloba o fim da década de 80 e vai até os dias atuais, em que técnicas inspiradas pela biologia e outros sistemas naturais começaram a surgir como, por exemplo, as redes neurais e algoritmos genéticos.

2.2 Agentes Inteligentes

De acordo *Russel e Norving* (2003, p.32) é considerado um agente, qualquer entidade que através do uso de sensores possa perceber o ambiente em que ele está situado e que através de atuadores possa realizar ações que modifiquem este ambiente. Sob esta perspectiva, um agente pode ser qualquer programa computacional desde que receber uma entrada e produzir uma saída sejam considerados respectivamente a sensorear e atuar em um ambiente. Entretanto, nem todo agente pode ser considerado inteligente, dado que a ideia de inteligente está intrinsecamente ligada a ideia de raciocinar. Desta forma, podemos dizer que um agente inteligente é qualquer agente que possa perceber o ambiente a sua volta, raciocinar e, a partir disso, tomar a decisão que melhor o levará à realização do seu objetivo. Um modelo geral de um agente inteligente pode ser visto na figura 1.

Figura 1: Modelo de Agente Inteligente



Fonte: Retirada do slide, Agentes Inteligentes II, Fábio M. Pereira

A taxonomia dos agentes, criada por *Russel e Norving* (2003), está associada ao subconjunto de propriedades que cada um possui. Assim, eles podem ser classificados da seguinte forma:

- **Agente reflexivo** (ou puramente reativo)**:** é o agente cujas regras ou funções são utilizadas associando diretamente a percepção com a ação (condição – ação). Por exemplo, encher balde com água, **Se** balde cheio **Então** parar. Esse tipo agente tende a executar a primeira ação cuja regra se encaixe de acordo com uma percepção dada, tornando-se, dessa

forma, eficiente e compreensível em tomada de decisões simples, mas desvantajoso por não armazenar uma sequência perceptiva e ser pouco autônomo[3].

- **Agente autônomo** (ou reativo baseado em modelo do mundo)**:** este agente se difere do anterior por construir e manter um modelo interno do ambiente exterior. Assim, suas ações não dependem somente da percepção atual, mas também do conjunto de ações e percepções anteriores, tornando possível que percepções semelhantes apresentem ações diferentes. Entretanto, isto também o limita, tornando-o pouco autônomo pelo fato de não possuir objetivos específicos.

- **Agente cognitivo** (ou orientado em objetivos)**:** este agente possui uma flexibilidade maior por adaptar suas escolhas a um resumido conjunto de objetivos dinâmicos. Dessa forma, ele se torna mais autônomo, entretanto, ele se limita a escolher ações que correspondem ao estado imediato (atual) do ambiente, não planejando as próximas ações que devem ser tomadas a fim de prever futuros estados que o ambiente pode apresentar para que ele alcance seus objetivos. Além disso, esse agente também possui dificuldades para lidar com objetivos conflitantes, como, por exemplo, chegar ao destino pelo caminho mais rápido, barato, seguro e bonito.

- **Agente deliberativo** (ou planejador)**:** é um agente no qual as ações tomadas a partir de uma percepção são mediadas não somente pelo estado atual do ambiente como pela previsão de estados futuros que o ambiente possa apresentar devido a uma sequência de ações. Para isso, este agente encadeia regras com a finalidade de construir um plano de múltiplos passos para alcançar seu objetivo. Dessa forma, este agente tem a vantagem de optar por escolher as melhores ações, mas por outro lado, o custo para obter tais ações durante o processo de deliberação pode ser excessivo para ambientes cuja tomada de decisão deva ser feita em tempo real.

- **Agente otimizador** (ou baseado em utilidades)**:** este agente incorpora algum tipo de heurística a seu raciocínio para definir preferências entre os estados do ambiente ou ações, que deseja executar a fim de alcançar um dado objetivo. Deste modo, o agente pode ponderar melhor entre vários objetivos conflitantes de forma a pesá-los de acordo com a probabilidade de alcançá-los. A desvantagem deste agente é que ele é pouco escalável, tornando-se assim, difícil de adaptar à diferentes ambientes, objetos e relacionamentos.

- **Agente adaptativo** (*learning*)**:** este agente possui componentes para análise crítica

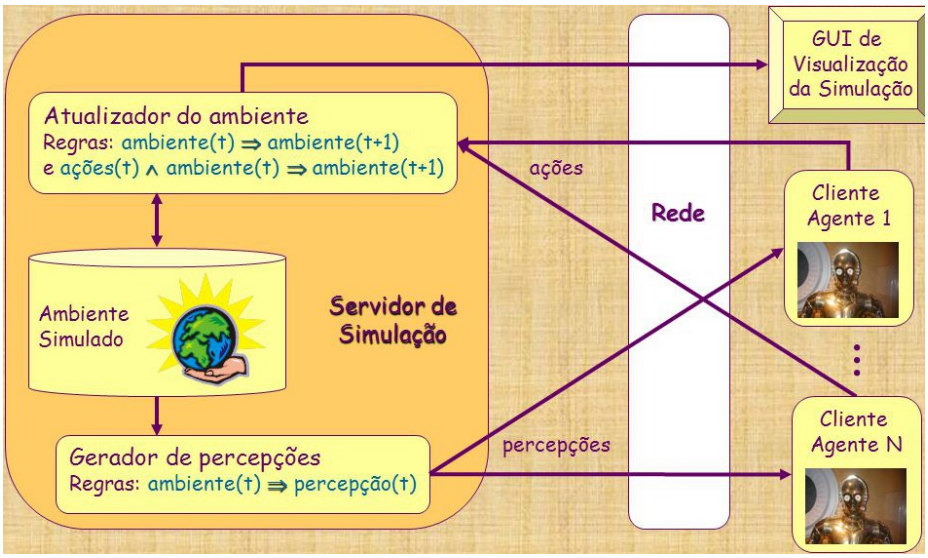
3 Neste trabalho o conceito de autônomo está diretamente ligado à ideia de autonomia de raciocínio, no qual o agente para governar-se utiliza uma base de conhecimento e uma máquina de inferência.

de desempenho e de aprendizagem de conhecimento, tornando possível para ele aprender e se adaptar ao ambiente em que se encontra.

2.3 Ambientes de Simulação

Ambientes de simulação são ambientes controlados que possibilitam portar características ideais e desejáveis do mundo real para um *software*. Nesses ambientes é possível realizar pesquisas nas mais diversas áreas da computação. Na área da inteligência artificial, por exemplo, eles podem ser utilizados como uma ferramenta para testes na criação de agentes autônomos ou na prototipação de projetos que envolvam aprendizado de máquina, uma vez que tais ambientes virtuais possuem o poder de simular os ambientes reais em que tais máquinas ou agentes podem ser empregados. A figura 2 ilustra a arquitetura padrão de um ambiente de simulação. Nesta arquitetura, o gerador de percepções contido no servidor de simulação fornece informações aos agentes. Estas informações permitem aos agentes realizarem ações que atualizam o estado do ambiente, sendo esta atualização passada aos usuários através da interface de visualização da simulação.

Figura 2: Arquitetura de um Ambiente de Simulação



Fonte: Retirada do slide, Agentes Inteligentes II, Fábio M. Pereira

O tipo de ambiente de simulação afeta diretamente o desenvolvimento de um agente que nele será inserido. Conhecer e saber classificar um ambiente de acordo com suas características é uma peça fundamental no projeto de um agente, visto que estas características nortearão os tipos de atuadores e sensores que o agente deverá possuir para interagir com a plataforma. Tendo isso em vista, a classificação de um ambiente segundo *Russel* e *Norving* (2003) pode ser realizada de acordo

com os seguintes critérios:

- **Nível de Acessibilidade**
  - **Totalmente acessíveis:** os sensores do agente percebem o estado completo do ambiente, assim todos aspectos relevantes para tomada de decisão são visíveis ao agente;
  - **Parcialmente acessíveis:** os sensores do agente não conseguem perceber o estado completo do ambiente seja devido a variáveis escondidas ou limitação dos sensores;
  - **Inacessível:** os sensores do agente são incapazes de detectar o estado do ambiente seja devido a ruídos ou por pouca granularidade nos sensores.
- **Determinismo**
  - Um ambiente é classificado como **determinista**, caso o agente possa prever exatamente qual será o próximo estado do ambiente dado a escolha de determinada ação. Caso isso não possa ser feito, seja devido a existência de outros agentes ou outras varáveis inerentes ao ambiente, ele é classificado como **não determinista**.
- **Dinamicidade**
  - Um ambiente pode ser classificado em **sequencial** (**ou estacionário**), se ele não sofrer alterações enquanto o agente escolhe a ação que deseja realizar. Caso contrário, o ambiente pode ser classificado como **concorrente síncrono**, desde que ele mude durante uma ação e outra, mas não durante o raciocínio do agente. Se isto não ocorrer e o ambiente puder mudar a qualquer momento, inclusive durante o raciocínio do agente, o ambiente pode ser classificado como **concorrente assíncrono**,
- **Número de agente**
  - Ambientes que possuem um único agente são classificados como **mono agente**, caso possuam mais de um agente, eles são classificados como **multiagente**. Além disso, a forma como os agentes interagem entre si, disputando ou cooperando para alcançar objetivos em comum dentro do ambiente, também afeta o modo de classificá-lo. Dessa forma, um ambiente pode, além de ser multiagente, também ser classificado como **cooperativo**, **competitivo** e **cooperativo/competitivo**.
- **Natureza Matemática das Grandezas**
  - As grandezas são os valores fornecidos aos sensores, aos atributos dos objetos, às relações e os locais do ambiente, etc. Esses valores classificam o ambiente e podem

variar entre **discretos**, por exemplo binário e nominal, e **contínuos**, por exemplo os valores dentro do conjunto dos reais.

- **Periodicidade**
  - Um ambiente é classificado como **episódico,** caso a experiência do agente no ambiente não seja afetada por ações tomadas por ele anteriormente. Caso contrário, ele é classificado como **não episódico**.
- **Tamanho**
  - Vários fatores podem servir para classificar um ambiente por seu tamanho. O número de agentes, de locais, de relacionamentos e a quantidade de percepções, ações e objetivos do agente são alguns exemplos desses fatores.

Existem diversos exemplos de ambientes de simulação empregados para ensino e pesquisa na área de IA. Alguns dos mais famosos serão citados a seguir.

2.3.1 O Mundo de Wumpus

O Mundo do *Wumpus* (*Russell, Norvirg,* 2004) é um jogo de computador em que um agente deve explorar uma caverna formada por salas conectadas através de passagens. O mapa do jogo é igual a uma matriz, no qual cada quadrado (ou posição da matriz) corresponde a uma sala. O agente sempre inicia no canto inferior esquerdo do mapa e tem como objetivo explorá-lo a procura de uma barra de ouro. Uma vez que ele a encontre, seu objetivo se torna sair da caverna com a barra de ouro em mãos. Para que o agente tenha sucesso em sua missão, ele deve evitar entrar em salas que contenham buracos ou o *Wumpus* (um monstro que devora qualquer um que entre em sua sala) para não morrer.

Um dos grandes pontos positivos do mundo do *Wumpus* é que ele funciona bem como exemplo ilustrativo para vários problemas relacionadas a IA, pois seu ambiente é simples de implementar e de compreender. Alguns dos pontos negativos, é que suas características como: inacessibilidade, ser discreto, não episódico, sequencial e mono agente, são obrigatórias para todas as suas simulações, o que o torna inviável para alguns tipos de estudos como, por exemplo, os relacionados a ambientes multiagentes ou dinâmicos. Além disso, também não há uma diversidade de ações e iterações do agente com o ambiente.

2.3.2 Robocode

O *Robocode* (*Larsen*, 2013) é um jogo de programação cujo objetivo do jogador é programar um robô tanque para batalhar e competir contra outros robôs em uma arena. As batalhas na arena ocorrem em tempo real e podem ser vistas pelos jogadores, permitindo que eles possam ver o desempenho de seus algoritmos em tempo real. Isto torna o aprendizado mais divertido, pois os jogadores competem entre si através de uma batalha para ver qual algoritmo foi melhor elaborado. Um ponto negativo é que apesar de o Robocode ser um ambiente dinâmico, multiagente e com ações em tempo real, a falta de interação dos agentes com o cenário deixa a simulação um pouco monótona e com poucas alternativas a nível de estratégica, pois os agentes só podem fugir ou atacar seus inimigos.

2.3.3 RoboCup

A *RoboCup* (*RoboCup*, 2014) é uma iniciativa internacional cuja finalidade é aumentar a pesquisa na área de robótica e IA. Esta iniciativa oferece uma plataforma para pesquisas em desenvolvimento de robôs virtuais para simulações de futebol e resgate. A simulação de futebol é conhecida como *Soccer Server* e, assim como o esporte de verdade, se trata de um ambiente multiagente e em tempo real em que para alcançar a vitória, um time deve fazer o máximo de gols que puder no time adversário, ao mesmo tempo em que tenta levar o mínimo possível. Para isto, os membros de cada equipe precisam se comportar de maneira rápida, flexível e cooperativamente, levando em considerações situações locais e globais, tornando-se, dessa forma, uma plataforma interessante para estudos voltados as áreas de IA distribuída e pesquisas multiagentes. Porém, um ponto desfavorável deste ambiente é a falta da diversidade de classes de percepções, ações, objetivos e terrenos, e a alta complexidade necessária para o desenvolvimento do mecanismo de raciocínio dos agentes, praticamente impedindo o seu uso em cursos introdutórios de IA.

2.4 Representação do Conhecimento dos Agentes

A representação do conhecimento é um subtópico da IA que estuda como o conhecimento é organizado e processado, assim como que tipo de estruturas de dados um agente inteligente utiliza e que tipo de raciocínio pode ou não ser feito com o conhecimento que ele adquiriu. De acordo com Juha P. Pesonen (2002, p.45), a representação do conhecimento tem um papel vital na IA, porque

ela não só determina em larga escala que tipo de raciocínio pode ser feito com o conhecimento, como também o quão rápido ele pode ser, o quanto de memória ele consome e, até mesmo, o quão ótimo e completo o algoritmo que utiliza este conhecimento pode ser.

Neste trabalho, foram utilizadas duas abordagens para representar o conhecimento dos agentes: a Orientação a Objetos e a Lógica de Primeira Ordem. É importante salientarmos que existem outras formas de representar o conhecimento dos agentes, porém devido ao fato de fugirem do escopo do trabalho optamos por não abordá-las.

2.4.1 Orientação a Objetos

A Orientação a Objetos é um paradigma que de acordo Silva D. L. et al. (2009), citada por Fábio Pereira (2014, p.30), utiliza a ideia de classes e objetos para classificar um domínio conhecido, possibilitando o reúso dos mesmos em outras aplicações. Um dos fatores que contribuem para reusabilidade destas aplicações é a flexibilidade advinda do encapsulamento de dados, que possibilita que estas aplicações sejam divididas em partes menores e mais isoladas possibilitando que modificações sejam feitas em uma parte da aplicação sem que isso afete diretamente a aplicação inteira.

Outro ponto positivo da Orientação a Objetos, de acordo com Juha P. Pesonen (2002, p.53), é que ela permite uma analogia mais realista entre modelos reais e de *software.* Isto possibilita que os componentes contidos em um programa sejam conceituados de forma semelhante a dos existentes no mundo real, permitindo, assim, uma maior compreensão da estrutura e do comportamento apresentado por eles. Nesta representação do conhecimento, uma classe é a definição de um conjunto de entidades que possuam características e comportamentos semelhantes, enquanto um objeto é uma entidade específica pertencente a alguma classe.

Para ficar mais clara esta definição, tomemos como exemplo os animais em uma floresta. As corujas, cobras, onças, etc, contidos nesta floresta são derivações (ou herdeiros) da classe animal e por isso eles possuem características incorporadas a eles que são comuns entre si, mas, ao mesmo tempo, eles possuem suas próprias características individuais que definem sua subclasse. Por exemplo, uma coruja é uma subclasse que deriva da classe animal igualmente a classe onça. Entretanto, a classe coruja possui características e comportamentos que a diferencia da classe onça como: asas, penas e voar, assim como possui características em comum como: dois olhos, duas orelhas e serem hábeis caçadores. Já um exemplo de objeto seria uma coruja específica, ou seja, uma instância da classe coruja que possua todas as características e comportamentos que definem

esta classe como, por exemplo, uma coruja branca, de olhos vermelhos, com penas brancas e bico marrom.

2.4.2 Lógica de Primeira Ordem

A Lógica de Primeira Ordem é um tipo de lógica matemática que pressupõe que o mundo é constituído de objetos e que entre eles existem relações que podem ou não ser válidas. Além disso, os objetos contidos nesta lógica possuem propriedades e funções que os distinguem entre si. Segundo Fábio Pereira (2014, p.32), a Lógica de Primeira Ordem apresenta os critérios necessários para a representação do conhecimento, já que sua linguagem, robusta e expressiva, permite representar o mundo, adequá-lo através de inferências e acréscimos de novos conhecimentos e, sobretudo, devido a sua modularidade consegue conceder uma maior legibilidade, confiabilidade e flexibilidade a base de conhecimento.

2.5 Jogos Sérios

De acordo com Damien Djaouti et al (2011, p.4), embora o atual termo Jogos Sérios tenha começado a aparecer em meados de 2002, muitos jogos com propósitos sérios já haviam sido feitos antes desta data. Segundo este autor, um dos primeiros usos do termo Jogos Sérios, próximo do seu significado conhecido, foi pelo pesquisador Clark Abt em 1970. Este pesquisador tinha como objetivo utilizar os jogos para treinamento e educação. Inclusive, ele é responsável por desenvolver diversos jogos para estas funções, sendo alguns utilizados até mesmo pelas forças armadas americanas. Além disso, de acordo com Liliane S. Machado et al.(2009, p.2), a década de 1980 é onde se localizam temporalmente o surgimento dos Jogos Sérios, com o uso de jogos em formato de simuladores por militares para treinamentos de combate e voo, prática que até hoje é utilizada para o treinamento de soldados e pilotos inexperientes.

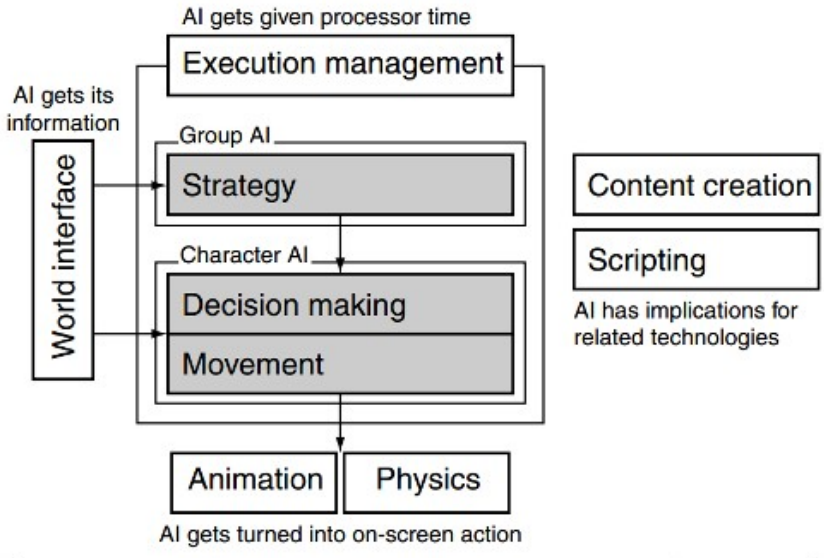
Ao longo dos anos, os Jogos Sérios têm sido feitos com diversos propósitos e por isso uma nomenclatura foi criada para ajudar a classificá-los. Apesar de ainda não estar bem solidificada, esta nomenclatura é bastante aceita por definir bem o propósito de cada jogo, alguns exemplos são: *Advergames*, jogos cuja a finalidade é divulgar algum produto, marca ou instituição; *Edutainment*, jogos que buscam educar através do entretenimento; *Newsgame*, jogos cujo foco é informar sobre alguma reportagem ou evento jornalístico; *Simulation Game*, jogos que permitem aos jogadores adquirirem ou exercitarem diferentes habilidades através da simulação de ambientes específicos

para isso; *Exergames*, jogos que permitem aos jogadores se exercitarem fisicamente.

2.6 Modelo de IA em Jogos

Por nosso ambiente de simulação se tratar de um jogo, optamos por utilizar o modelo de IA para jogos proposto por Ian *Millington* e *John Funge* (2009). É valido ressaltarmos que além deste modelo, exitem outras estruturas que ajudam a compreender como a IA é utilizada em jogos, entretanto, optamos por este devido ao fato dele ter uma definição concisa e de fácil entendimento. Neste modelo, as tarefas da IA são divididas em três níveis: movimento, tomada de decisão e estratégia. Como pode ser visto na figura 3, as primeiras duas seções contém algoritmos que são responsáveis pelo controle individual de cada personagem, enquanto o último contém algoritmos que opera controlando um grupo de personagens. Juntamente a estas seções, há um conjunto de elementos adicionais de infraestrutura envolvendo como a IA interage com o mundo e com a tecnologia utilizada para desenvolver o sistema. Por estes elementos de infraestrutura não fazerem parte do escopo do trabalho como as animações e a física, por exemplo, eles não serão aprofundados aqui.

Figura 3: Modelo de IA para Jogos



Fonte: Retirada do livro, *Artificial Intelligence for Games*, p,32

O modelo acima serve como referência, pois nem todos os jogos precisam de todos os níveis de IA apresentado nele. Por exemplo, jogos de tabuleiros como Xadrez e Dominó requerem somente o nível de estratégia, uma vez que as peças contidas no tabuleiro não precisam fazer suas

próprias decisões e, ainda, não precisam se preocupar em se mover. Diferentemente destes jogos mencionados anteriormente, os jogos de tiro em primeira pessoa (*First-Person Shooter* ou *FPS*) apresentam um ambiente dinâmico em que cada personagem deve tomar decisões rápidas sobre o que fazer em cada situação, sendo assim necessário ter os níveis de movimento e tomada de decisão. Além disso, caso esses personagens façam parte de um mesmo grupo eles terão a necessidade de implementar também o nível de estratégia para que ações tomadas individualmente possam ser coordenadas.

2.6.1 Movimento

As técnicas de movimento se resumem em tornar decisões em movimentos. Muitas vezes, estes movimentos podem funcionar como gatilhos para ativar ações do personagem. Por exemplo, em jogos de tiros, geralmente, um personagem sem armas tende a tentar se aproximar do jogador antes de atacá-lo. Caso o jogador esteja muito longe dele, a tendência é que esse personagem utilize algum algoritmo para se aproximar do jogador para só assim ativar a animação da ação de atacar.

Além desse tipo de técnica, existem outras mais complexas, como as utilizadas pra evitar obstáculos ou as de navegação de mundo. Nelas, após tomada a decisão do que o personagem deve fazer, os algoritmos devem guiar o personagem do ponto de origem até o de destino, escolhendo a melhor rota possível evitando obstáculos que possam surgir durante o processo.

2.6.2 Tomada de decisão

Os algoritmos de tomada de decisão são responsáveis por decidir o que o personagem deve fazer a seguir. Normalmente, cada personagem em um jogo possui um número de ações que ele pode realizar, tais como: atacar, fugir, defender, pular, se esconder e assim por diante. Dessa forma, a função da tomada de decisão é escolher a ação que, apropriadamente, se encaixe a cada situação. Um bom exemplo são os guardas do jogo ***Metal Gear*** [*Konami Digital Entertainment, Inc, 1987*] que ao avistarem o jogador durante sua ronda, param a ação de patrulhar para persegui-lo e atacá-lo.

Normalmente, o nível de complexidade da tomada de decisão pode variar de jogo para jogo. Assim como em alguns jogos, a decisão de atacar um inimigo pode simplesmente significar que o personagem deva partir para cima com o que ele tiver em mãos. Já em outros, a mesma decisão pode significar que o personagem deva iniciar uma cadeia de outras decisões, como derrubar uma mesa no chão, utilizá-la como cobertura para se proteger e flanquear o inimigo ou, até

mesmo, pode afetar que tipo de arma o personagem deve utilizar.

2.6.3 Estratégia

De acordo com *Ian Millington* e *John Funge* (2009, p.10, tradução nossa), “estratégia se refere no geral a uma aproximação utilizada por um grupo de personagens.” Dessa forma, estratégia não está responsável por coordenar o comportamento de um único indivíduo, mas, sim, do grupo ao qual ele faz parte. Pois, cada personagem é livre para tomar suas próprias decisões e fazer suas próprias movimentações, desde que elas estejam de acordo com a estratégia elaborada para o grupo.

Suponhamos que há dois guardas e que um intruso acabou de ser visto por um deles. O algoritmo de estratégia pode orientar o guarda que viu o inimigo a avisar ao outro guarda, o que aconteceu e fazer com que eles decidam cercar o inimigo. Após isso, cada guarda decide por si mesmo qual é a melhor rota que ele deve tomar para cercar o inimigo, deixando ao algoritmo de estratégia a responsabilidade de coordenar a dupla para que eles não tomem a mesma rota. Assim, é importante notar que apesar de cada guarda decidir, individualmente, qual rota tomar, cada um deles afeta diretamente a estratégia tomada pelo algoritmo já que vai ser necessário organizá-los durante suas tomadas de decisões.

2.7 Técnicas para Tomada de Decisão de agentes em jogos

Os algoritmos que compõem as técnicas de tomada de decisão, como foi apresentado no tópico anterior, são responsáveis por estruturar o comportamento individual de cada agente por meio das ações que ele deve realizar, dada uma determinada situação ou objetivo. Estes algoritmos são bastante flexíveis, tanto que alguns podem ser adaptados para que além de coordenar as ações de um único agente possam também coordenar as de um grupo inteiro, tornando, assim, possível a realização de tarefas do nível de estratégia.

Dessa forma, neste tópico, buscamos apresentar algumas das técnicas utilizadas na construção de mecanismos para tomada de decisão em agentes. Dentre estas, estão as máquinas de estados, as árvores de decisão e os sistemas baseados em regras, além de abordarmos também algumas ferramentas matemáticas e de aprendizado.

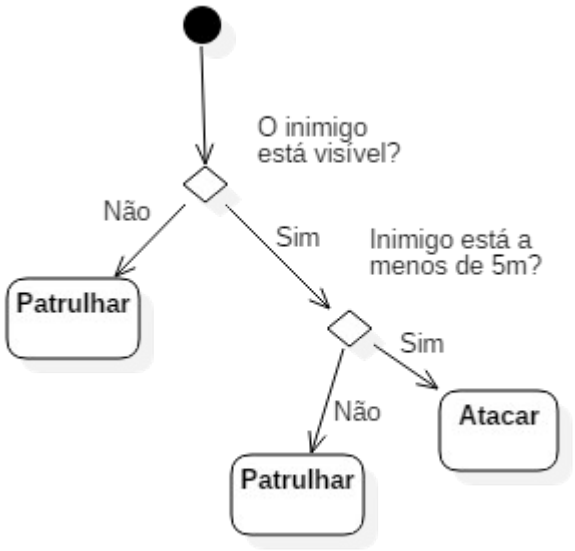
2.7.1 Árvore de Decisão

A árvore de decisão é uma técnica simples, de fácil implementação e entendimento que, apesar de sua simplicidade, pode se tornar bastante sofisticada dependendo de como aplicada. Por ser flexível e fácil de criar, é amplamente utilizada em diversas áreas de tomada de decisão dentro dos jogos, de tal forma que podem ser vista em quase tudo, desde a simples controle de animação e personagens a complexas tarefas de IAs táticas e estratégicas.

Toda árvore de decisão é constituída por pontos de decisão conectados, sendo o primeiro ponto de decisão chamado de raiz e os outros de nós. Dessa forma, para definir uma ação, uma árvore inicia sua busca pela raiz tomando decisões a todo nó por onde passar até que não haja mais decisões a serem tomadas. A forma como uma decisão é tomada a cada nó é por meio da checagem de algum valor. Geralmente, esse valor pode variar de acordo com o tipo de estrutura de dados utilizada para representar o conhecimento do agente como, por exemplo, valores *booleanos*, enumerações ou vetores 2D. A partir do momento em que a árvore decidir que não há mais decisões a serem feitas, ela chegará em uma folha e acoplada a cada folha há uma ação que deve ser tomada pelo agente, encerrando, assim, a busca.

Na maioria das árvores de decisão, uma ação pode aparecer em múltiplas folhas e um nó geralmente possui apenas duas possibilidades de decisão, mas nada impede que possam existir mais de duas possibilidades de decisão por nó. Um exemplo de árvore de decisão pode ser vista na figura 4. Nela, o agente só atacará um inimigo, caso este inimigo esteja visível e a menos de 5 metros dele. Caso contrário ele patrulhará o local.

Figura 4: Árvore de decisão simples



Fonte: Elaborada pelo autor

2.7.2 Máquina de Estados

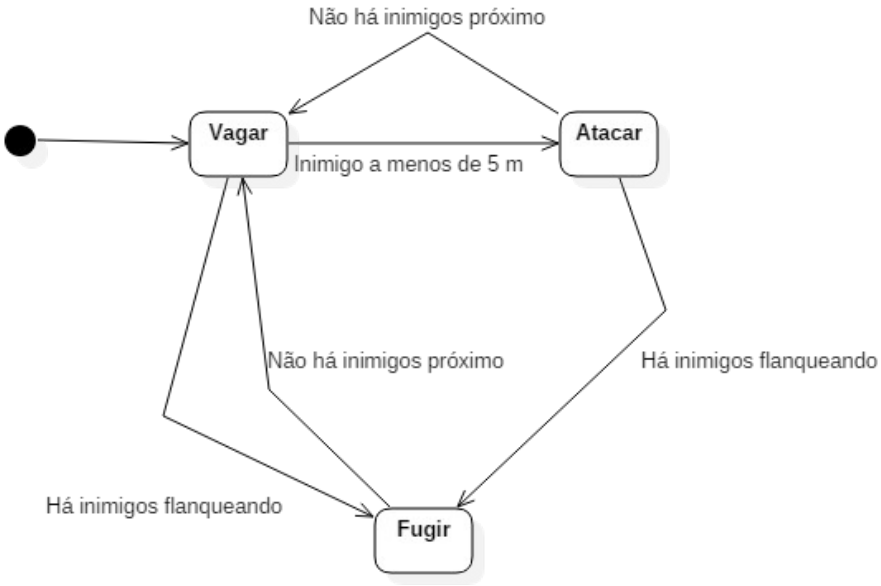
Uma máquina de estados finitos, ou simplesmente máquina de estados, é uma máquina abstrata que é responsável por definir quais os comportamentos que um personagem deve assumir dado algum evento. Estes comportamentos são classificados em estados e são responsáveis por definir quais tipos de ações um personagem deve tomar ao estar neles. Dessa forma, um personagem que está no estado atacar, por exemplo, pode alterar seu estado para fugir caso seja flanqueado, alterando, assim, o seu comportamento.

O fato das máquinas de estados representarem o comportamento de um agente por meio de estados é o que as tornam populares já que isso facilita a compreensão da técnica, tornando mais fácil implementá-la. Tendo isso em vista, o algoritmo funciona da seguinte forma: o agente sempre está dentro de algum estado e este estado é o responsável por determinar as ações que ele deve tomar; Para que o agente mude de estado, é necessário que algum evento, seja ele interno (pertencente ao agente) ou externo (pertencente ao ambiente), ative alguma transição pertencente ao estado em que ele se encontra; Uma vez que isso ocorra, o agente mudará as ações do antigo estado pelas ações do novo estado.

A figura 5 exemplifica uma máquina de estados. Nela, o agente pode atacar, vagar e fugir. É importante notar que cada estado possui suas próprias transições e que cada transição possui um tipo de evento específico para ser ativada. Dessa forma, se o agente está no estado fugir, por exemplo, ele só poderá ir para estado vagar caso não haja inimigos o flanqueando e nunca poderá ir para estado de atacar, pois não existe transição que ligue o estado fugir ao estado atacar, como pode ser visto na figura.

Uma das principais dificuldades em empregar máquinas de estados finitos na construção de mecanismos de tomada de decisão é, segundo *Ian Millington* e *John Funge* (2009, p.318), na modelagem de comportamentos de alarme. Nestes comportamentos, a máquina de estado deve interromper um comportamento normal em resposta a algum evento importante. Para que isso seja feito, é necessário que o estado responsável pelo comportamento de alarme esteja ligado a todos os outros estados contidos na máquina. O que pode ocasionar uma duplicação do número de estados e transições na máquina, tornando a mais dispendiosa de desenvolver e implementar. Para contornar este problema, pode-se utilizar as máquinas de estados hierárquicas.

Figura 5: Exemplo de Máquina de Estados



Fonte: Elaborada pelo Autor

Diferentemente das demais, as máquinas de estados hierárquicas possuem uma estrutura hierárquica entre seus estados que as permitem definir uma prioridade entre os comportamentos que devem ser escolhidos dado um certo evento. Isto é feito a partir da inserção de novas máquinas de estado ao sistema, dessa forma, em vez de combinar todos os comportamentos em uma única máquina de estados é possível separá-la em diversas máquinas mantendo em cada uma delas sua própria lógica, possibilitando aos estados manterem seus comportamentos originais. Como as máquinas são organizadas de forma hierárquica, uma máquina de nível mais baixo na hierarquia é executada somente quando não há nenhum evento ativando uma máquina com o nível mais alto que o seu. Isto garante que sempre que algum evento ative um mecanismo de alarme em uma máquina de nível superior está será executada.

Outro ponto importante, é que a criação dessas novas máquinas diminui a quantidade de transições e estados que precisariam ser implementados. Esta redução ocorre porque os estados que anteriormente precisariam estar ligados diretamente ao estado que possui o comportamento de alarme, agora, estão ligados através da máquina que os possui. Assim, em vez de termos uma transição para cada estado contido na máquina, nós temos uma única transição ligando a máquina ao estado do comportamento de alarme.

2.7.3 Sistemas baseados em regras

Os sistemas baseados em regras são sistemas especialistas que segundo, *Ian Millington* e *John Funge* (2009), vem sendo utilizados por pelo menos 15 anos tanto dentro quanto fora dos jogos. Apesar de apresentarem suas próprias peculiaridades e terem fama de ineficientes, os sistemas baseados em regras são uma abordagem bastante comum, especialmente, porque conseguem lidar com situações nas quais o raciocínio do personagem não pode ser facilmente antecipado por designers e desenvolvedores. Além disso, eles são capazes de simplificar situações que ficariam extremamente complexas ao se utilizar máquinas de estados ou árvores de decisão, como, por exemplo, a representação de comportamentos que apresentam características semelhantes mas que diferem em suas ações.

A estrutura comum de um sistema baseado em regras é dividido em duas partes: um banco de dados contendo o conhecimento disponível para a IA e um conjunto de regras condição-ação. As regras são verificadas pelo banco de dados no intuito de confirmar se suas condições foram atendidas. Regras que tenham suas condições atendidas podem ser selecionadas para serem ativadas, executando, assim, seu componente ação. Para demonstrar o funcionamento dessa estrutura consideremos como exemplo um cachorro virtual.

Neste exemplo nosso cachorro pode fazer os seguintes fenômenos: Se há um osso por perto e ele está com fome então ele irá comê-lo; Se ele está com fome, mas não há ossos perto ele procura por um; Se ele não está com fome, mas está com sono então ele dormirá; E por último, se ele não está com fome e não está com sono, então ele irá andar e latir. Observe que todos os fenômenos que o nosso cachorro pode realizar possuem as ações e suas respectivas condições para serem realizadas, dessa forma fica claro a forma com nosso cachorro deve agir a cada uma das possíveis situações em que ele se encontra.

2.7.4 Técnicas Matemáticas e de Aprendizado

Técnicas matemáticas ou de aprendizado para melhorar o desempenho dos algoritmos e tornar mais realista o raciocínio do agente. A lógica *Fuzzy* é um ótimo exemplo de uma destas técnicas matemáticas, nela o conceito de incerteza é inserido a lógica tradicional. Dessa forma, diferente dos algoritmos que empregam a lógica tradicional em que as condições e decisões só podem ser verdadeiras ou falsas (0's ou 1's), os algoritmos que fazem uso da lógica *Fuzzy* apresentam condições e decisões que possuem graus de verdade e falsidade. Estes graus pode ser medido por valores que estejam entre o intervalo de 0 e 1, em que os valores mais próximos de zero

possuem maior grau de falsidade enquanto os mais próximos de 1 possuem maior grau de verdade. Isto faz com que o comportamento dos agentes tornem-se mais realistas e convincentes pois adicionar uma maior flexibilidade à IA.

Além da lógica *Fuzzy*, existem os algoritmos de *pathfinding* (busca de caminho). Tais algoritmos são responsáveis por calcular uma rota através do mapa da fase para que um personagem possa chega da sua posição atual à posição do seu objetivo. Dentre esses algoritmos estão o A* (lê-se A estrela) e *Dijkstra*. Os referidos algoritmos, segundo André Kishimoto (2004, p.7), utilizam *grids* (grades) criadas no mapa como representação de nós de um grafo e atribuem a cada um desses nós um custo de deslocamento, podendo, assim, formar caminhos por meio de buscas pelo grafo passando por nós que detenham os menores custos. Ambos os algoritmos garantem encontrar o caminho caso ele exista. Entretanto, a diferença entre eles é que o algoritmo A* utiliza de uma heurística para otimizar o desempenho da busca pelo grafo, de forma a garantir que o caminho encontrado será sempre ótimo[4], enquanto o algoritmo de *Dijkstra* requer mais processamento, mas garante sempre encontrar o melhor caminho. Esses algoritmos são utilizados com muita frequência em jogos RTS (Real Time Strategy – Estratégia em tempo real) como os jogos *Age of Empires* [*Ensemble Studios*, 1997] e *StarCraft* [*Blizzard Entertainment*, 1998]. É importante ressaltar que tanto lógica *fuzzy* quanto os algoritmos de *Pathfiding* não são técnicas propriamente dita de tomada de decisão, mas, sim, uma ferramenta que auxilia outras técnicas de tomada de decisão a se tornarem mais eficientes e realistas.

Apensar de ainda serem raros de se encontrar nos jogos comerciais atuais, é muito comum vermos estudos acadêmicos sobre técnicas de aprendizado sendo utilizadas para a tomada de decisão. As redes neurais artificiais são um exemplo disto, nelas são construídos mecanismos similares às redes neurais biológicas, em que neurônios artificiais são conectados entre si seguindo algum padrão de arquitetura. Cada neurônio possui uma entrada e saída cujos comportamentos são determinados por uma função de ativação e a cada conexão de entrada com um outro neurônio há um valor associado que determina a relevância entre a conexão de ambos neurônios. Sendo assim, o resultado obtido pelo processamento de uma rede neural depende da configuração assumida pelos neurônios combinada com seu processo de aprendizado. O que segundo Victor K. Tatai (2002, p.34) torna possível, entre outras coisas, tomadas de decisões complexas, processamento de informações, otimização e aprendizado.

4 A critério de esclarecimento, caminho ótimo é um caminho que apresenta um bom custo-benefício (pode ser em questão de tamanho ou qualquer outro parâmetro que seja utilizado para definir a qualidade do caminho), mas que não necessariamente precisar ser o melhor.

3 Ambiente de Simulação *Bomberman X*

Devido a dificuldade de encontrar uma ferramenta adequada que possibilite aos discentes das disciplinas introdutórias de IA fixarem o conhecimento adquirido em sala de aula, através da aplicação prática de algoritmos e técnicas de IA, desenvolvemos o ambiente de simulação *Bomberman X*. Esta é uma ferramenta *open source* projetada com a finalidade de prover um ambiente de simulação para desenvolvimento e testes de algoritmos relacionados as disciplinas de IA, principalmente, as que envolvem agentes inteligentes, tornando possível aos discentes praticar os conteúdos vistos em sala de aula de uma forma criativa, intuitiva e diversificada.

O projeto do *Bomberman X* ainda está em fase de desenvolvimento e é vinculado ao Laboratório Virtual de Sistemas Inteligentes da Universidade Estadual do Sudoeste da Bahia, coordenado pelo professor Fábio Moura Pereira. Neste projeto, busca-se disponibilizar aos discentes um ambiente de simulação que assemelha-se a um jogo digital no qual os jogadores são responsáveis por programar a IA dos agentes que fazem parte do ambiente. Isto torna possível aos jogadores compararem e competirem entre si através dos seus agentes, possibilitando a eles um aprendizado empírico.

Além disso, para que fosse possível aos discentes aplicar os conhecimentos adquiridos em sala de aula, o sistema do *Bomberman X* foi estruturado para dar suporte desde agentes puramente reflexivos até agentes que utilizem técnicas de tomada de decisão, permitindo ao discente ir gradualmente aumentando o nível do seu algoritmo conforme o conteúdo é ministrado em sala de aula. Isto é possível devido ao sistema fazer chamadas externas ao módulo de IA do agente, conforme será mostrado adiante neste capítulo. Outro ponto importante deste sistema é que ele possui um cenário interativo no qual os agentes devem não só reconhecer o cenário disponibilizado pelo ambiente como também saber utilizá-lo ao seu favor. Isto faz com que os jogadores sejam desafiados de forma dinâmica e inteligente pelo sistema, pois a cada partida podem existir diferentes formas de vencer, tornando-o completamente diferente de sistemas que possuem cenários fixos e não interativos, tais como *Robocode* (*Larsen*, 2013) e *Robocup* (*RoboCup*, 2014), em que o cenário onde a simulação ocorre é um mero plano de fundo para a interação dos agentes. Uma tabela com uma comparação entre os ambientes *Bomberman* X, RoboCup, Robocode e O Mundo do Wumpus pode ser visto na figura 6.

Figura 6: Comparação entre ambientes de simulação

Características	Ambiente			
	Mundo do Wumpus	Rodocode	Rodocup	Bomberman x
Nível de Acessibilidade	Inacessível	Inacessível	Inacessível	Inacessível
Determinismo	Determinista	Não Determinista	Não Determinista	Não Determinista
Dinamicidade	Sequencial	Dinâmico	Assíncrono	Síncrono
Número de Agentes	Monoagente	Multiagente	Multiagente	Multiagente
Natureza Matemática das Grandezas	Discreto	Discreto	Contínuo	Discreto
Periodicidade	Não episódico	Não episódico	Não episódico	Não episódico
Pós	Ambiente simples de implementar e de compreender	Aprendizado divertido; possibilita visualizar o desempenho dos algoritmos em tempo real	Plataforma interessante para estudos voltados as áreas de IA distribuída e pesquisas multiagentes em tempo real	Fácil desenvolvimento do mecanismos de raciocínio do agente; variedade de terrenos, ações e percepções; cenário com elementos interativos
Contras	Simplicidade do ambiente e falta de diversidade das ações e iterações do agente com o ambiente	Falta de interação dos agentes com o cenário e poucas alternativas a nível de estratégia	Falta da diversidade de classes de percepções, ações, objetivos e terrenos, além da alta complexidade para o desenvolvimento do mecanismo de raciocínio dos agentes	Ambiente em fase de desenvolvimento e falta de diversidade de personagens

Fonte: Elaborada pelo autor

A fim de explicar como o sistema funciona, o dividiremos em dois subtópicos: o ambiente de simulação e a arquitetura do sistema. No primeiro, falaremos sobre as regras, os objetivos dos agentes e as entidades que o ambiente de simulação possui. Já no segundo, explicaremos como todo o sistema está estruturado, demonstrando suas classes e estruturas lógicas.

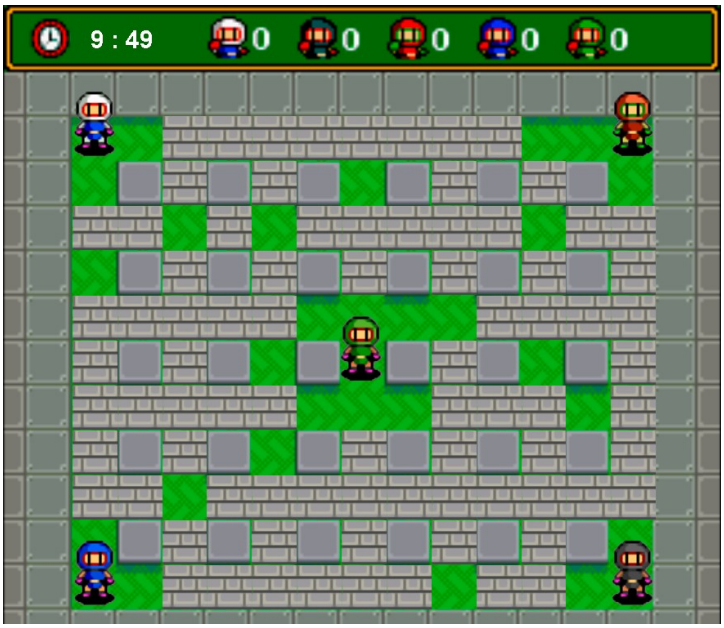
3.1 Ambiente de Simulação

O *Bomberman X* baseia-se na franquia de jogos da série *Super Bomberman* [*Hudson Soft.,* 1993 a 1997], de onde deriva seu nome. Deste modo, o design do ambiente de simulação foi feito com o intuito de assemelhar-se ao máximo com o dos jogos originais, diferenciando-se, apenas, pela ausência de alguns itens e pelo jogador ter que programar o seu personagem em vez de controlá-lo por meio de um *joystick*, não afetando, assim, as regras e a essência do jogo.

Tendo isso em vista, o *Bomberman X* é um ambiente de simulação de estratégia em tempo real baseado em labirinto que, de acordo com a taxonomia de Russel e Norving (2003), pode ser classificado como multiagente, parcialmente acessível, não determinista, dinâmico episódico e discreto. Nele, o jogador possui uma visão de cima do cenário onde pode visualizar os obstáculos e

os outros personagens. O foco principal do jogo é que o jogador consiga programar seu personagem, o *Bomberman*, para sobreviver e eliminar os outros personagens do cenário antes que o tempo esgote. Para conseguir isto, o personagem pode colocar bombas para explodir obstáculos e inimigos e, ainda, movimentar-se para cima, para baixo, para esquerda e para direita desde que não haja obstáculos em seu caminho. Além disso, os personagens podem coletar itens que aparecem aleatoriamente no cenário após a destruição dos obstáculos. Tais itens funcionam como melhorias para o *Bomberman*, como aumentar o número de bombas que ele pode colocar ao mesmo tempo ou aumentar a velocidade do movimento dele. Vejamos a figura 7 que exemplifica um cenário com os personagens do ambiente de simulação:

Figura 7: Exemplo de Cenário do Bomberman X



Fonte: Elaborada pelo autor

Conforme pode ser visto na figura 7, dentro das paredes que cercam o cenário existem dois tipos de obstáculos: os blocos rachados, que são destrutíveis; e os blocos lisos, indestrutíveis. Para que um agente possa destruir um bloco rachado, ele deve colocar uma bomba próxima a ele. Assim, quando a bomba explodir, a chama da explosão colidirá com o bloco e o destruirá. Se devido a destruição do bloco surgir um item, o agente pode pegá-lo andando até a posição em que o item está. Como já foi citado, a melhoria que este item pode causar ao agente irá variar de acordo com o seu tipo. Existem quatro tipos de itens: a chama, os patins, a bomba e o chuta bomba como mostrado na figura 8.

Figura 8: Itens

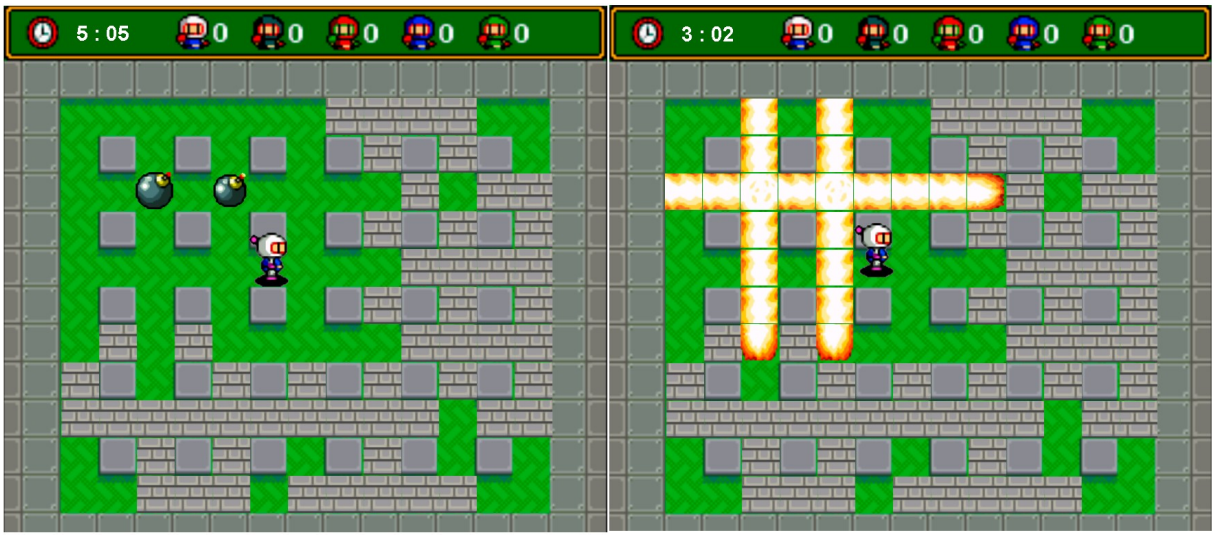
Fonte: Elaborada pelo autor

O item chama aumenta a potência das bombas colocadas pelo personagem, fazendo com que suas explosões alcancem distâncias maiores; o item patins faz com que o personagem se movimente rapidamente; já o item bomba, aumenta o número de bombas que o personagem pode colocar ao mesmo tempo no cenário, sendo que todo personagem inicia a rodada podendo colocar apenas uma bomba por vez. Por fim, temos o item chuta bomba. Para compreender o que este item faz, é importante entender primeiro como as bombas funcionam no ambiente. As bombas são peças fundamentais para o desenrolar da partida porque elas possibilitam os personagens explorarem o cenário e eliminarem seus inimigos. Além disso, elas podem ser utilizadas como obstáculos para impedir a passagem dos personagens, possibilitando os agentes elaborarem vários tipos de jogadas.

Um exemplo simples de um tipo de jogada que pode ser feita utilizando a bomba como obstáculo é tentar prender um personagem entre um bloco e uma bomba, impossibilitando ele de escapar da explosão. Uma vez que isso acontecer, a única forma deste personagem sair desta situação é se ele possuir o item chuta bomba, porque assim será permitido a ele empurrar a bomba para qualquer direção onde não exista um obstáculo colidindo com ela. Uma vez empurrada, a bomba só irá parar de se movimentar caso ela colida com algum obstáculo ou personagem, ou caso ela exploda durante o percurso.

Outra característica importante sobre as bombas são suas explosões. Normalmente, uma bomba após ser colocada, leva alguns segundos até explodir naturalmente, o que possibilita ao personagem fugir do alcance da explosão, evitando, assim, morrer. Um detalhe sobre essas explosões é, que com exceção do *Bomberman*, elas não capazes de atravessar nenhuma entidade. Inclusive, isto permite que, além de explodirem naturalmente, as bombas também possam ser explodidas por meio da colisão com chamas advindas da explosão de outras bombas. Isto possibilita aos personagens alinharem bombas a fim de conseguirem explosões em cadeia que atinjam áreas maiores. Na figura 9 temos um exemplo de uma explosão em cadeia, feita por um personagem.

Figura 9: Explosão em cadeia. Esquerda antes da explosão e direita depois da explosão



Fonte: Elaborada pelo autor

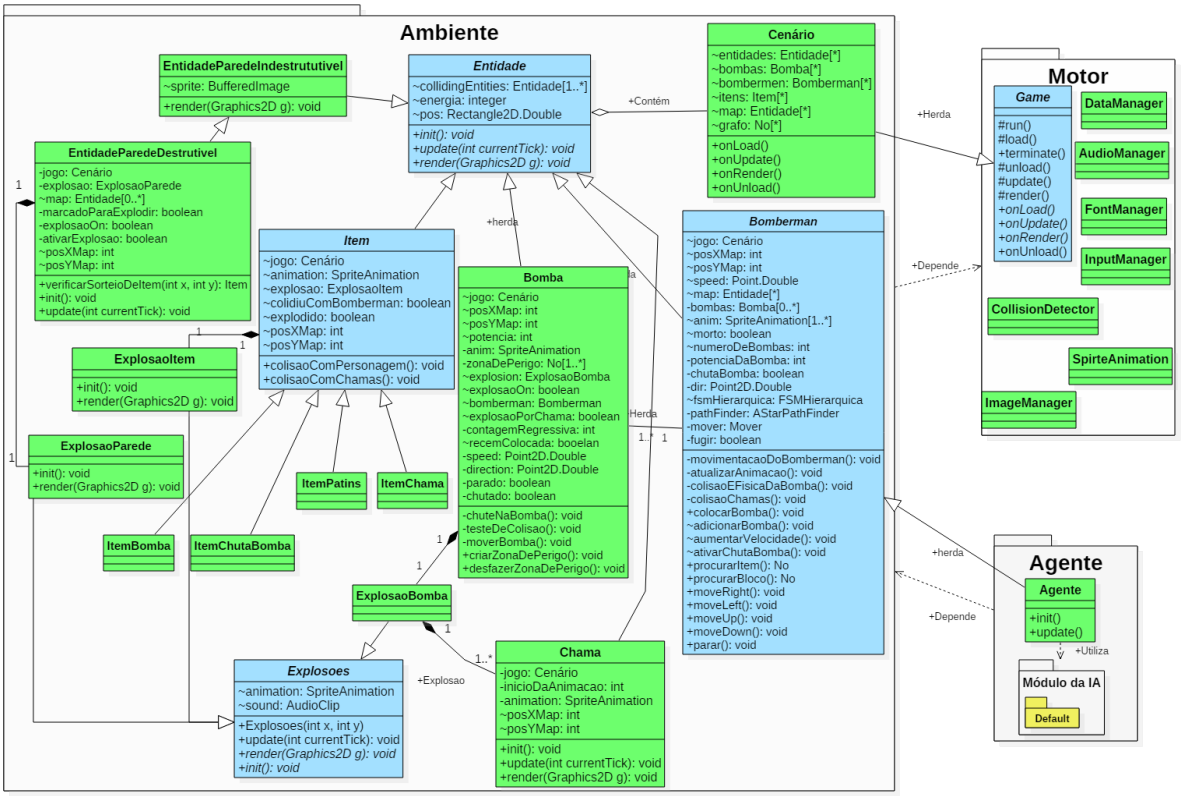
3.2 Arquitetura do Sistema

A arquitetura do *Bomberman* X foi projetada a fim de torná-lo bastante intuitivo e flexível, buscando, com isso, facilitar ao usuário a compreensão do sistema e incentivá-lo a participar do desenvolvimento dele, construindo, assim, uma comunidade ativa de desenvolvedores. Pensando nisto, o desenvolvimento do *Bomberman* X foi feito utilizando a linguagem de programação Java (Harvey M. Deitel & Paul J. Deitel, 2010) que além de ser multiplataforma e orientada a objetos, também possui uma ampla comunidade de usuários, tornando-a uma linguagem de fácil obtenção de material de estudos e informações. Além disso, também utilizamos a IDE (Integrated Development Environment – Ambiente de desenvolvimento integrado) Netbeans[5] 8.0.2 como ferramenta de suporte à programação. Um ponto importante é que por utilizarmos o Java é necessário que o usuário do sistema possua uma JVM[6] (Java Virtual Machine – Máquina Virtual Java) instalada em seu computador para executar o ambiente de simulação.

Outro ponto importante da arquitetura do sistema, como pode ser visto no diagrama de classes apresentado na figura 10, é que ela é organizada em três subsistemas que interagem entre si. Cada subsistema fica responsável por lidar com um tipo de tarefa em específico, solicitando, caso seja necessário, a ajuda de um outro através de uma interface. Isto facilita a compreensão do sistema e torna mais simples a depuração e os testes, além de criar um encapsulamento que permite que as atualizações de métodos e estruturas de um subsistema não afetem diretamente os outros. Os três subsistemas são: o motor, o ambiente e o agente. A seguir, apresentamos cada um dos subsistemas.

5 Para saber mais informações ou para baixar o Netbeans acesse o site netbeans.org
6 Para fazer o *download* do JVM acesse www.java.com/pt_BR/download

Figura 10: Diagrama de Classes do Sistema



Fonte: Elaborada pelo autor

3.2.1 Motor

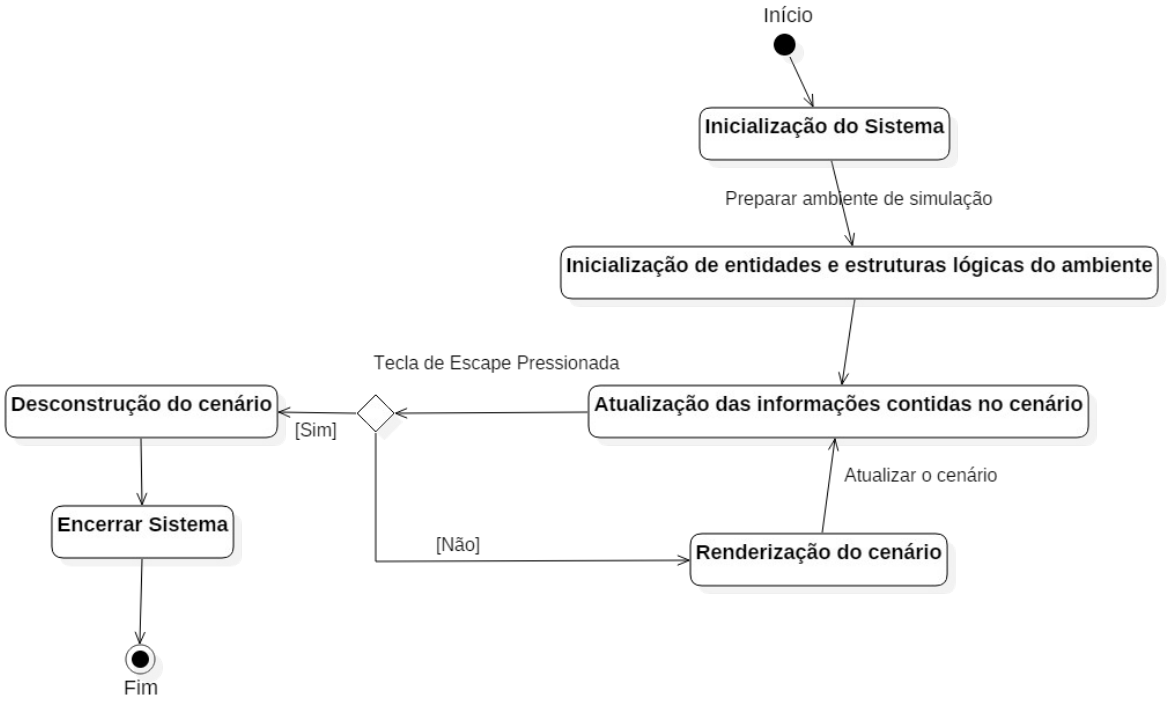
Este subsistema é a base do ambiente de simulação. Seu funcionamento assemelha-se aos motores feitos para jogos e, assim como eles, é responsável por realizar tarefas relacionadas ao processamento de gráficos, sons, detecção de colisão entre entidades, animação, gerência de arquivos e de entrada de dados. É a este que os outros subsistemas farão uma solicitação se por acaso uma entidade precisar ter representada na tela sua movimentação ou caso algum evento acione algum efeito sonoro. Desta forma, o motor é responsável realizar a ligação entre os eventos que ocorrem no ambiente de simulação com o computador.

Dentro do motor, a classe *ImageManager* é responsável por processar gráficos 2D para o ambiente e é utilizada pela classe *SpriteAnimation* para montar as animações das entidades. Já as classes *AudioManager* e *FontManager* são, respectivamente, responsáveis por controlar os efeitos sonoros do ambiente e as fontes utilizadas em textos exibidos na tela. O *InputManager* lida com a entrada de dados advinda do teclado e mouse e o *DataManager* faz o gerenciamento dos dados da simulação, podendo carregá-los ou salvá-los. A classe *CollisionDetector* permite ao sistema

verificar a colisão entre as entidades contidas no cenário. Todas as classes com o nome "*manager*" são responsáveis por gerenciar alguma coisa dentro do sistema. Devido a este fato, elas foram feitas seguindo o padrão de projeto *Singleton,* visando garantir que exista apenas uma instância de cada uma delas, impedindo, dessa forma, que haja mais de um objeto gerenciando um mesmo tipo de dados.

A classe *Game* é uma classe abstrata responsável por gerenciar o sistema. As classes que a implementam podem executar processos de inicialização, atualização, renderização e finalização do sistema. O funcionamento desta classe é bem simples e ocorre todo dentro do método ***run*()**. **N**este método, há uma fase de inicialização do sistema que é feita por meio da chamada do método ***load*()** onde é criado a janela principal do sistema e é inserida a lógica do ambiente. Após o carregamento ser encerrado, o método ***run*()** entra em um laço que só se encerrá ao término da simulação. Dentro deste laço, a cada repetição, ocorre a atualização da lógica inserida durante o ***load*()** e a renderização da tela, sendo a primeira feita pelo método ***update*()** e a segunda pelo método ***render*()**. Ao fim da simulação, o laço é encerrado e o método ***unload*()** é chamado para finalizar o sistema. Isto pode ser visto no diagrama de atividade apresentado na figura 11, no qual todo o sistema é apresentado de forma simplificada.

Figura 11: Diagrama de Atividades do Sistema



Fonte: Elaborada pelo autor

Outro ponto importante, é que dentro dos métodos ***load*()**, ***update*()**, ***render*()** e ***unload*()** há um método abstrato que deve ser implementado pela classe herdeira da classe *Game* para que seja possível a ela inserir seus elementos lógicos e gráficos. Estes métodos são: o ***onLoad*()**, o ***onUpdate*()**, ***onRender*()** e ***onUnload*()** que serão explicados no tópico a seguir.

3.2.2 Ambiente

O ambiente é o subsistema responsável pela construção da estrutura e da lógica do ambiente de simulação, é nele que as regras e as entidades definidas no tópico 3.1 são aplicadas. As classes contidas neste subsistema são basicamente as entidades e o cenário. Dentre elas, a que abordaremos com mais detalhes neste tópico será a classe Cenário, devido ao fato dela implementar a classe abstrata *Game* do subsistema motor e também reunir as entidades e suas interações. Antes de nos aprofundarmos na explicação desta classe, é importante explicarmos qual é a estrutura padrão de uma entidade, assim, ficará mais fácil compreender como funciona o cenário.

Toda entidade precisa implementar um conjunto de métodos abstratos advindos da classe pai Entidade, são eles: ***init*()**, ***update*()** e ***render*()**. O método ***init*()** é onde ocorre a inicialização dos atributos que não foram inicializados no construtor e outros itens lógicos da entidade. Este método só é chamado durante o carregamento de cada partida e nele pode haver, por exemplo, instruções de carregamento de animações da entidade ou de posicionamento da entidade no mapa. Já o método ***update*()** é onde ocorre a atualização dá lógica da entidade, normalmente, a chamada deste método ocorre durante as atualizações do cenário. Por atualizações do Cenário, nós queremos dizer o processo em que o Cenário processa as ações realizadas pelas entidades no ambiente e atualiza seu estado. Desta forma, o método ***update*()**, geralmente, é composto, por exemplo, por atualização de animações, de movimentos de personagens e de testes de colisões. E por último, o método ***render*()** que é responsável por pintar e repintar na tela as animações das entidades.

A classe Cenário é responsável tanto pela construção de todo o ambiente de simulação quanto da comunicação dele com o computador. Além disso, ela também disponibiliza aos agentes todas informações necessárias para que eles se situem sobre o estado atual do ambiente, funcionando, assim, como a classe central de todo sistema. Nela, as entidades contidas no ambiente são armazenadas em um vetor e tem seus métodos ***init*()**, ***update*()** e ***render*()** executados durante momentos distintos da execução do código do Cenário. O Cenário é composto basicamente por quatro métodos abstratos herdados da classe game, são eles: o ***onLoad*()**, o ***onUpdate*()**, o ***onRender*()** e o ***onUnload*().**

O método ***onLoad*()** é onde o cenário é construído, ou seja, onde as entidades são adicionadas e posicionadas no cenário, a cada partida. Por isto, é aqui que os métodos init() das entidades são chamados. Já o método ***onUpdate*()** é o responsável por atualizar o estado do ambiente. Devido a isto, ele é chamado durante todo o desenrolar da simulação, já que é ele que processa as atualizações vindas do método ***update*()** das entidades. O ***onRender*()** é o método que atualiza as imagens contidas na tela. Para fazer isso ele chama o método ***render*()** de todas entidades contidas no ambiente e, assim como o ***onUpdate*()**, é chamado durante todo o desenrolar da simulação. O método ***onUnload*()** faz exatamente o contrário do método ***onLoad*()** e é executado ao final da partida para limpar os dados e varáveis utilizados durante a simulação, desconstruindo, desta forma, o cenário.

3.2.3 Agente

Este é o subsistema encarregado de lidar com o raciocínio do agente e é através dele que o jogador se comunicará com o ambiente. Assim como foi mostrado na figura 10, ele é divido em agente e módulo da IA. Agente é o nome dado para qualquer classe que implemente a classe abstrata *Bomberman* e é partir dele que é feita a interação do personagem com o ambiente. Já o módulo da IA é um pacote em que os jogadores podem adicionar as classes e estruturas lógicas utilizadas para construir o mecanismo de raciocínio do agente. Este módulo permite que o jogador possa desenvolver algoritmos mais arrojados e abrangentes que, devido a sua organização e complexidade, não podem ser desenvolvidos dentro do escopo do agente.

As classes do tipo agente devem ser implementadas pelo jogador e possuem dois métodos da classe pai que devem ser sobre escritos por serem a base de suas estruturas: o ***init*()** e o ***update*()**. O método ***init*()**, como já foi citado no tópico anterior, é responsável por inicializar os atributos e a lógica da entidade. É neste método que o jogador deve inicializar o mecanismo de raciocínio desenvolvido por ele. Já o método ***update*()** é onde este mecanismo de raciocínio do agente é colocado para executar. Além disso, é **obrigatório** que na primeira linha desses métodos seja feita uma chamada para o método pai. Desta forma, na primeira linha de ***init*()** dever ter uma chamada ***super.init*()** e em ***update*()**, também na primeira linha, deve ter uma chamada ***super.update*()**. Isto é feito para que o jogador não precise lidar com o gerenciamento de animações, movimentos e colisões do personagem, tendo, assim, somente que preocupar-se com o mecanismo de raciocínio.

Para a construção do mecanismo de raciocínio do agente o jogador possui uma biblioteca de métodos fornecidos pela classe *Bomberman*. Utilizando dessa biblioteca, o jogador pode obter, por

exemplo, informações sobre a quantidade de inimigos, os objetos contidos no cenário, a quantidade de bombas que o personagem possui e a posição do personagem no cenário. Além de fornecer para o jogador métodos que controlam a movimentação e as ações que o personagem pode realizar. Esta biblioteca de métodos se encontra no apêndice A, onde seus métodos estão devidamente apresentados e explicados. Outro ponto importante é que no módulo da IA, dentro do pacote *Default,* encontra-se o mecanismo de raciocínio desenvolvido como estudo de caso deste trabalho. Nele, há exemplos que demonstram como essa biblioteca pode ser utilizada. Detalhes sobre esse mecanismo de raciocínio serão representados no próximo capítulo.

4 ESTUDO DE CASO

O estudo de caso realizado neste trabalho, fundamenta-se em verificar como o ambiente de simulação *Bomberman* X comporta-se à aplicação de técnicas de IA nos agentes. Para isso foram realizadas as seguintes etapas: 1. Escolha do mecanismo de raciocínio do agente; 2. Modelagem do mecanismo de raciocínio e representação do conhecimento utilizando técnicas de orientação a objetos; 3. Implementação do mecanismo de raciocínio do agente; e 4. Avaliação e testes. A seguir detalhamos o processo de realização de cada uma destas etapas.

4.1 Escolha do Mecanismo de Raciocínio do Agente

Para o mecanismo de raciocínio do agente, foi proposto e implementado uma combinação de máquinas de estados hierárquicas com o algoritmo A*. A utilização da máquina de estados hierárquica visa resolver problemas relacionados à tomada de decisão do agente no ambiente, enquanto o algoritmo A* visa proporcionar assistência ao agente durante os processos de movimentação e tomada de algumas decisões.

Durante o processo de modelagem do raciocínio dos agentes, percebemos que para testar o ambiente de forma mais ampla, precisaríamos implementar uma técnica que permitisse definir o raciocínio que os agentes possuem da forma mais genérica possível, mas que, ao mesmo tempo, o deixasse claro e intuitivo para que o acompanhamento de seus comportamentos e ações pudessem ser feitos. Dadas tais premissas, inicialmente, foi cogitada a utilização de máquinas de estados para o controle dos agentes. Entretanto, durante a modelagem dos comportamentos percebemos que alguns apresentavam características de comportamento de alarme e por isso foi necessário criar níveis de prioridade entre eles. Assim como foi abordado no tópico 2.7.2, os comportamentos de alarme tem como um de seus grandes problemas o aumento no número de transições e estados contidos na máquina. Desta forma, visando evitar este excedente optamos por utilizar uma máquina de estados hierárquica.

Outro fator que colaborou para esta escolha, foi o tipo de agente que queríamos utilizar para testar o ambiente. Em primeira instância, achávamos que utilizar um agente reflexivo para tal papel seria o suficiente, pois, assim como abordado no tópico 2.2, tal agente possui seu conjunto de regras definidas através da associação direta da percepção com a ação, tornando-o compatível com a ótica utilizada pelas transições contidas nas máquinas de estados. Além disso, este tipo de agente é extremamente viável para testar tanto o gerador de percepções quanto a reposta do ambiente. Porém, por ser muito simples, notamos que utilizar somente este tipo agente não seria o suficiente

para testar todo o ambiente, pois não teríamos dimensão de como ele se comportaria ao uso de um tipo de agente mais arrojado. Dado este fato, optamos por criar um agente que fosse reflexivo e que em alguns de seus comportamentos possuíssem características de agente deliberativo. Para tornar isto possível, escolhemos utilizar o algoritmo A* para dar respaldo a algumas tomadas de decisões feitas pelo agente. Nos tópicos seguintes, demonstraremos como o raciocínio do agente foi criado e implementado e quais foram os resultados obtidos com ele.

4.2 Modelagem do Raciocínio e Representação do conhecimento

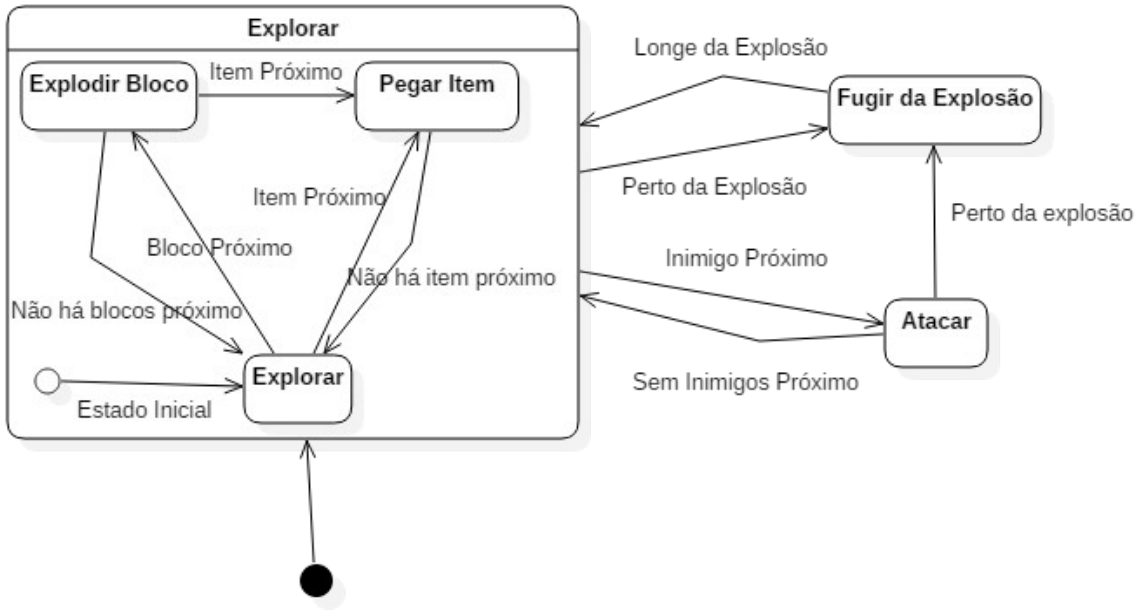
Assim como foi abordado no tópico 3.1, para alcançar a vitória em uma partida, o personagem deve eliminar todos os adversários e ser o único sobrevivente. Tendo isso em vista, para implementação do agente, nós modelamos seu raciocínio em três tipos de comportamentos, são eles: explorar, atacar e fugir. O comportamento de explorar é responsável por fazer o agente vagar pelo cenário a procura de itens para fortalecê-lo. Como os itens aparecem aleatoriamente após a explosão dos blocos espalhados pelo cenário, este comportamento também é responsável por fazer o agente procurar e explodir blocos. Já os comportamentos de atacar e fugir, são mais intuitivos, sendo o foco do primeiro fazer o agente atacar sempre que estiver próximo de um inimigo, colocando uma bomba para explodi-lo. Enquanto o segundo, faz o agente fugir para uma posição segura sempre em que ele estiver no alcance de alguma explosão.

Vagar pelo cenário, explodir blocos e procurar por itens são três objetivos distintos, porém entrelaçados entre si que formam o comportamento de explorar, já que, por exemplo, não existe uma maneira de explodir blocos sem vagar pelo cenário, assim como não há uma maneira de vagar pelo cenário sem explodir blocos, exceto quando o cenário já estiver sem blocos. Dado este fato, resolvemos criar um estado para cada um desses objetivos no intuito de manter intacto as ações e os comportamentos que o agente deve possuir para alcançá-los. Além disso, criamos uma máquina de estados e a encarregamos de gerenciar somente estes estados, tornando possível, com isso, manter a lógica original do comportamento de explorar.

Os comportamentos de atacar e fugir, diferentemente de explorar, foram modelados como estados únicos que possuem suas próprias máquinas de estados, pois seus objetivos são, respectivamente, colocar uma bomba caso o inimigo esteja próximo e, assim, sair do alcance de explosões. Além disso, estes dois comportamentos apresentam características de comportamentos de alarme, porque que ambos estão diretamente ligados às condições para vitória do agente na partida, forçando-os, assim, a terem prioridade maior que o comportamento de explorar. Já que para

vencer, o agente precisa sobreviver, nós optamos por definir o comportamento de fugir com uma prioridade superior ao de atacar, conforme pode ser observado no diagrama de estados[7] da figura 12:

Figura 12: Representação do Raciocínio do Agente



Fonte: Elaborada pelo autor

É importante observar na figura anterior, que todo o diagrama é uma máquina de estados hierárquica e que o ponto preto na figura indica que a máquina de estados hierárquica inicia pela máquina de estados Explorar. Ou seja, a máquina de estado Explorar funciona como o ponto de partida para o raciocínio do agente. Isto é feito porque, inicialmente, todos os personagens surgem cercados por blocos que os impedem de trafegar livremente pelo cenário, sendo impossível que eles saiam de suas posições, caso estes blocos não sejam destruídos. Outro ponto importante, é observar que Explorar, Fugir da Explosão e Atacar são submáquinas contidas dentro da máquina hierárquica e assim como máquinas de estados comuns, elas possuem um estado inicial dentro delas e podem transitar livremente entre seus estados. Contudo, caso algum evento de alarme seja ativado com prioridade maior que as transições contidas na submáquina, será necessário que a máquina hierárquica intervenha na execução da submáquina atual e faça a transição para a submáquina que lidará com o evento, conforme pode ser observado na transição “perto da explosão” demonstrado na figura anterior.

7 No diagrama, optamos por substituir os nomes vagar e procurar itens tanto por questões cosméticas quanto pelas novas definições representarem melhor o comportamento destes estados.

Uma outra situação que a máquina hierárquica também precisa lidar, é em saber para qual estado o agente deve retornar após um comportamento de alarme ser desativado. Por exemplo, caso o estado atual do agente seja Explodir Bloco e ele entre no alcance de uma explosão, a máquina hierárquica observará que esta transição foi ativada e interromperá o estado atual para que o estado Fugir da Explosão possa ser ativado. Após o agente sair do alcance da explosão e o evento de alarme for desativado, note que em vez de retornar diretamente ao estado anterior, Explodir Bloco, o agente retornará à submáquina Explorar. Uma vez que isto aconteça, a submáquina deve verificar se o estado Explodir Bloco foi finalizado antes da transição para o estado Fugir da Explosão ter sido feita. Caso isto tenha ocorrido, a submáquina simplesmente iniciará do estado inicial contido nela. Caso isto não ocorra, ela retomará de onde o estado Explodir Bloco parou.

Agora que explicamos como a máquina hierárquica funciona e como foram modelados seus estados e submáquinas, é necessário abordamos como foi feita a implementação das ações que o agente deve tomar em cada estado. Primeiramente, é importante ressaltarmos que os comportamentos apresentados pelo agente foram divididos em dois tipos: os estados que necessitam que o agente se movimente e os estados que necessitam que o agente coloque bombas. No primeiro grupo, estão os estados que utilizam o algoritmo A* para movimentar o agente pelo cenário, são eles: o Pegar Item, o Explorar e o Fugir da Explosão. Ao entrar nestes estados, algum destino é definido e, imediatamente, é solicitado ao algoritmo A* que escolha uma rota para que o agente consiga chegar a tal local. Já no segundo, estão os estados que utilizam o algoritmo A* para planejar suas ações, são eles: o Explodir Bloco e o Atacar. Por estes estados envolverem colocar uma bomba, isso quer dizer que o agente se colocará no alcance de uma explosão. É importante que antes que essa bomba seja colocada, o agente saiba se há como fugir da explosão e por isso ele utiliza o A*. Note que em ambos os grupos o algoritmo A* é utilizado na busca de caminhos. Contudo, no primeiro grupo, após encontrar um caminho os estados iniciarão a movimentação do personagem. Já no segundo, o fato de encontrar ou não um caminho será utilizado como respaldo para a próxima ação que o agente deve tomar no estado.

Para que o algoritmo A* pudesse ser empregado dessa forma pelos estados, foi necessário criar um nó para cada posição do cenário em que o agente pudesse estar. Após isso, atribuímos custos a cada um desses nós com o objetivo de permitir que o agente sempre procure por rotas com os menores custos possíveis. O custo dado a cada um dos nós, seguiu como critério, o fato de haver ou não obstáculos na posição em que o nó está, por exemplo, caso na posição (3, 5) não haja nenhum bloco ou bomba, o nó que está nesta posição terá seu custo igual a 1. Contudo, caso haja algum obstáculo, o custo para o agente chegar até este nó será igual a infinito. A escolha de infinito

como valor para esta situação, se deve ao fato de que um agente não pode estar na mesma posição que um bloco ou uma bomba[8]. Um outro ponto importante, é que os nós com o custo igual a 1 são aumentados sempre que estiverem no alcance de alguma explosão e só terão seus custos retornados ao valor normal, quando não houver mais explosões em suas posições. Isto é feito para alertar os agentes que utilizar estes nós como rota pode ser perigoso.

Além do custo atribuído aos nós, também foi necessário escolher o tipo de heurística que seria empregada para otimizar a busca de caminhos. Dado isso, optamos pela escolha da Geometria do Táxi para este papel. Está geometria também é conhecida como Distância de *Manhattan* e nela, a distância entre dois pontos é igual a soma das diferenças absoluta de suas coordenadas. Esta escolha se deu devido ao fato de que dentro do cenário, o agente não pode realizar movimentos em diagonal e, como a Distância de *Manhattan* desconsidera movimentos em diagonal, a distância encontrada por esta heurística sempre ficará próxima da distância mínima real que o agente deve percorrer. A fórmula desta heurística pode ser vista a seguir. Nela, dt é a distância entre dois pontos, x1 e x2 são coordenadas no eixo x e y2 e y1 são coordenadas no eixo y.

Figura 13: Distância de *Manhattan*

$$d_t = |x_2 - x_1| + |y_2 - y_1|$$

Fonte: Retirada do site,
http://taxicabgeometry.altervista.org/general/definitions.html

4.3 Implementação

Para facilitar o processo de implementação do mecanismo de raciocínio do agente, nós o dividimos em quatro componentes lógicos: os estados, responsáveis por realizar as ações do agente; as transições, responsáveis por lidar com as percepções geradas pelo ambiente; as máquinas de estado, responsáveis pelo controle do raciocínio do agente; e os algoritmos de movimentação, responsáveis por definir rotas e movimentar o agente pelo cenário.

Para a implementação dos estados, criamos uma classe abstrata chamada State. Nela, há os métodos abstratos entrar no estado, fechar o estado e *update*. O entrar no estado e fechar estado são

8 É importante deixarmos claro que ao afirmamos que um agente não pode estar na mesma posição que uma bomba, estamos nos referindo ao fato de que ao haver uma bomba e um agente em posições diferentes, o agente **nunca** conseguirá alcançar a posição da bomba, a não ser que, a bomba exploda e deixe de existir ou ele a desloque para outra posição. Por isso, está afirmação não se aplica ao fato de o agente poder ficar na mesma posição de uma bomba que ele acabou de colocar.

responsáveis, respectivamente, por inicializar e encerrar o estado. Já o método *update* é responsável por definir o comportamento do agente. Nas transições, assim como nos estados, criamos uma classe abstrata chamada *Transition*. Nela, existem dois métodos abstratos importantes: o verificar transição, que verifica se as condições para a transição ser ativada foram cumpridas e o trocar de máquina, que faz a transição entre as máquinas de estados.

As máquinas de estados foram divididas em duas classes: a máquina de estados hierárquica FSMHierarqueica e a máquina de estado comuns FSM. A FSMHierarquica é responsável por gerenciar as máquinas de estados comuns enquanto elas controlam os comportamentos do agente. Em ambas as máquinas há um método chamado *update*, que é responsável por fazer a atualização do comportamento do agente. Já nos algoritmos de movimentação, nós desenvolvemos a classe AStarPathFinder, onde implementamos o algoritmo A* e o Mover, onde desenvolvemos os métodos utilizados para mover o personagem. Após todo esse processo, nós criamos a classe Agente e adicionamos todos estes componentes.

4.4 Avaliação e Testes

Após implementado o mecanismo de raciocínio do agente, realizamos testes empíricos e exaustivos (força bruta) diretamente no código-fonte, com o objetivo de procurar possíveis erros que pudessem existir no ambiente de simulação. Pelo ambiente de simulação ainda estar em fase de desenvolvimento, os experimentos realizados nele ocorreram sob algumas restrições. A primeira, é que o ambiente da simulação não possui áudio, pois não foram desenvolvidos trilhas e efeitos sonoros para ele ainda. A segunda, é que as partidas só possuem um round, sendo que a simulação sempre deve ser encerrada caso algum agente alcance a vitória ou termine em empate. E por último, é que foi utilizado o mesmo tipo de mecanismo de raciocínio em todos os agentes da simulação e, por isso, não foram realizados testes comparativos entre diferentes tipos de algoritmos de raciocínio.

Levando em consideração tais premissas, no primeiro conjunto de testes, realizarmos a verificação das grades do cenário, da percepção de blocos indestrutíveis e dos movimentos do agente. Para que essa verificação fosse possível, neste primeiro momento, utilizamos um único agente no ambiente e retiramos todos os blocos destrutíveis do cenário. Dessa forma, o agente entraria somente no estado de explorar e navegaria pelo cenário, testando todos os nós contidos nele, evitando as posições em que houvesse blocos indestrutíveis. Após confirmamos que o agente passou por todas as posições e que conseguia reconhecer as posições dos blocos indestrutíveis no

cenário, nós iniciamos o segundo conjunto de testes, em que, aos poucos, fomos inserindo os blocos destrutíveis. Neste teste, quase todas as percepções e ações que o agente possui foram testadas. É importante observar que ao colocarmos os blocos destrutíveis no cenário, a máquina de estados que controla o mecanismo de raciocínio do agente poderá ativar todos os estados contidos na máquina, com exceção do estado Atacar. Dessa forma, pudemos realizar o teste das ações de pegar itens e colocar bombas, além de realizarmos os testes da percepção de blocos destrutíveis, itens, bombas e alcance de explosões. Foi nesta fase também que pudemos testar as colisões do sistema, já que todas as entidades que o agente pode colidir foram adicionadas.

No terceiro conjunto de testes, nós adicionamos os outros agentes para verificarmos como o sistema se comportaria ao lidar com um ambiente com múltiplos agentes. Após a aplicação de todos esses testes, pudemos avaliar que o ambiente de simulação *Bomberman* X apresenta um rendimento satisfatório ao problema que ele visa sanar, ou seja, ele apresenta as ferramentas necessárias que o permite ser utilizado para implementação de algoritmos e técnicas de IA. Isso se deve ao fato, de que tanto o atualizador do ambiente funciona conforme o esperado, respondendo bem as ações tomadas pelo agente, quanto a interface gráfica e o gerador de percepções do ambiente apresentam um rendimento razoável. Entretanto, ainda é necessário uma melhoria nestes dois últimos componentes, pois pudemos notar, não só problemas relacionados a renderização de algumas animações pela interface gráfica, como o de posicionamento dos agentes no cenário pelo gerador de percepções.

5 CONSIDERAÇÕES FINAIS

O objetivo por trás do desenvolvimento do ambiente de simulação *Bomberman* X é fornecer uma ferramenta que seja útil tanto para docentes quanto para discentes na área de IA, permitindo a eles utilizá-lo como um material de apoio aos estudos ligados a estas disciplinas. Sabemos que uma das principais dificuldades no ensino e aprendizado nesta área, vem da falta de oportunidade em poder colocar em prática os algoritmos e técnicas que são vistos em sala de aula. Por isso, quando criamos este ambiente de simulação nos moldes de um jogo digital, esperamos não só disponibilizar aos usuários um ambiente para estudo e aplicações práticas, mas um mecanismo cuja estratégia é motivar os usuários através de um aprendizado agradável e divertido, pois vemos nos jogos, devido a sua riqueza e complexidade, uma excelente plataforma para teste e validação de novas metodologias e algoritmos.

A implementação da máquina de estados hierárquica e do algoritmo A* foram fundamentais para os testes realizados no ambiente de simulação. Por serem simples e intuitivos, esses algoritmos tornaram possível realizarmos testes claros e objetivos que demonstraram um resultado satisfatório do ambiente de simulação. Devido ao prazo para o término do trabalho, não tivemos a oportunidade de implementar outros algoritmos para realizarmos testes comparativos no ambiente de simulação e, por isso, esperamos realizar estes testes em trabalhos futuros. Um outro ponto, é que o *Bomberman* X ainda está em fase de desenvolvimento e que há espaço para grandes melhorias no sistema. Por isso, esperamos futuramente realizar tarefas como: adicionar novos cenários e novas ações aos personagens, melhorar o processo de renderização de animações do sistema, aprimorar o sistema de percepções do ambiente e paralelizar o sistema. Além disso, também como um trabalho futuro, esperamos que após finalizado, possamos testar este ambiente em sala de aula para verificar os benefícios que ele trará para o ensino de IA, aprimorando-o de acordo com os resultados que obtivermos nestes testes.

6 REFERÊNCIAS BIBLIOGRÁFICAS

BNDES, **Relatório Final do Mapeamento da Indústria Brasileira e Global de Jogos Digitais.** Fevereiro/2014.

CROCOMO, Marcio Kassouf. **Um algoritmo evolutivo para aprendizado on-line em jogos eletrônicos**. 2008. Tese de Doutorado. Instituto de Ciências Matemáticas e de Computação.

DIAS-NETO, A. C.. **Casos de Teste: Aprimore seus casos e procedimentos de teste**. Engenharia de Software Magazine, v. 1, p. 1, 2014.

DEITEL, Harvey M.; DEITEL, Paul J. **Java, como programar**. 4ª Edição.Porto Alegre, 2003.

DJAOUTI, Damien et al. *Origins of Serious Games*. Toulouse, França. 2011.

MACHADO, L.S.; MORAES, R.M.; NUNES, F.L.S (2009) **Serious Games para Saúde e Treinamento Imersivo**. *Book Chapter*. In: Fátima L. S. Nunes; Liliane S. Machado; Márcio S. Pinho; Cláudio Kirner. (Org.). Abordagens Práticas de Realidade Virtual e Aumentada. Porto Alegre: SBC, p. 31-60.

KISHIMOTO, André. **Inteligência artificial em jogos eletrônicos.** 2004.

MIDDLETON, Zak. *Case History: The Evolution of Artificial Intelligence in Computer Games.* Stanford, 2002

MILLINGTON, Ian. *Artificial Intelligence for games*/Ian Millington, Jhon Funge. - 2nd ed.p.cm. Includes index.

OSÓRIO, F. S. et al. **Inteligência Artificial para Jogos: Agentes especiais com permissão para matar... e raciocinar**. SBGAMES, 2007.

PEREIRA, F. M. **Laboratório Virtual de Sistemas Inteligentes**. UESB, 2014.

PESONEN, Juha Petteri. *Concepts and object-oriented knowledge representation*. MA Thesis, University of Helsinki, Department of Cognitive Science, fev. 2002.

RIEDER, Rafael; BRANCHER, Jacques Duílio. **Aplicação da Lógica Fuzzy a Jogos Didáticos de Computador-A Experiência do Mercadão GL**. In:CONGRESSO IBEROAMERICANO DE INFORMÁTICA EDUCATIVA. 2004. p. 127-136.

RUSSEL, S., NORVING, P. **Inteligência Artificial.** 2a ed. Elsevier, 2004.

TATAI, Victor Kazuo. **Técnicas de Sistemas Inteligentes Aplicadas ao Desenvolvimento de Jogos de Computador**. Campinas, 2003.

APÊNDICE A – Biblioteca de Métodos Fornecidos ao Agente Pela Classe *Bomberman*

Figura: Métodos fornecidos ao agente pela classe Bomberman

Método	Descrição do método	Tipo de Retorno
moveUp()	Movimenta o personagem para cima no mapa.	Void
moveDown()	Movimenta o personagem para baixo no cenario.	Void
moveLeft()	Movimenta o personagem para esquerda.	Void
moveRight()	Movimenta o personagem para direita.	Void
parar()	Para o movimento do personagem.	Void
colocarBomba()	Faz o personagem colocar uma bomba.	Void
GetPosXMap()	Retorna a posicao do personagem no eixo X.	int
GetPosYMap()	Retorna a posicao do personagem no eiox Y.	int
GetPoteciaDaBomba()	retorna o tamanho da explosao que as bombas do personagem alcacam.	int
GetNumeroDeBombas()	retorna a quantidades de bombas que o personagem pode utilizar.	int
IsChutaBomba()	Retorna verdadeiro, caso o personagem possa chutar bombas.	boolean
MapaDeEntidade()	Retorna uma matriz contendo todas as entidades do mapa.	Entidade[][]
Grafo()	retorna uma matriz com os nos do cenario.	No[][]
BombasNoCenario	retorna um vetor contendo todas as bombas do cenario	ArrayList<Bombas>
ItensNoCenario	retorna um vetor com os itens que estao no cenario.	ArrayList<Itens>
BombermansNoCenario	retorna um vetor com os personagens do cenario.	ArrayList<Bomberman>