UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO ACADÊMICO DE INFORMÁTICA

TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

LUCAS GABRIEL GONSALVES

UM ESTUDO DE APLICAÇÃO DE INTELIGÊNCIA ARTIFICIAL EM JOGOS

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2018

LUCAS GABRIEL GONSALVES

UM ESTUDO DE APLICAÇÃO DE INTELIGÊNCIA ARTIFICIAL EM JOGOS

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, do Departamento de Informática da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. André Koscianski

PONTA GROSSA

2018

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Ponta Grossa
Diretoria de Graduação e Educação Profissional
Departamento Acadêmico de Informática
Tecnologia em Análise e Desenvolvimento de Sistemas

TERMO DE APROVAÇÃO

UM ESTUDO DE APLICAÇÃO DE INTELIGÊNCIA ARTIFICIAL EM JOGOS

por

LUCAS GABRIEL GONSALVES

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 20 de novembro de 2018 como requisito parcial para a obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Prof. Dr. André Koscianski
Orientador(a)

_____
Prof. Dr. André Pinz Borges
Membro titular

_____
Prof. Dr. Richard Duarte Ribeiro
Membro titular

_____
Profª. Dra Helyane Bronoski Borges
Responsável pelo Trabalho de Conclusão de Curso

_____
Prof. Dr. André Pinz Borges
Coordenador do curso

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

Dedico este trabalho à minha família e meus amigos, pela capacidade de acreditar em mim e motivar-me nessa jornada.

AGRADECIMENTOS

Agradeço ao meu orientador Prof. Dr. André Koscianski, pela sabedoria com que me guiou nesta trajetória.

Aos meus preciosos amigos que tornaram essa jornada divertida e me ajudaram a passar por momentos difíceis.

Gostaria de deixar registrado também, o meu reconhecimento à minha família, pois acredito que sem o apoio deles seria muito difícil vencer esse desafio.

Enfim, agradeço a todos aqueles que de alguma forma me ajudaram na realização desse trabalho.

RESUMO

GONSALVES, Lucas Gabriel. **Um estudo de aplicação de Inteligência Artificial em Jogos**. 2018. 61f. Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2018.

Jogos eletrônicos começaram a se popularizar no início da década de 1970. Houve uma grande evolução desde então. Motores de jogos, *Engines* em inglês, são fundamentais para o desenvolvimento dos mesmos, visto que buscam agilizar o desenvolvimento dos jogos, auxiliando os desenvolvedores. O conceito de Inteligência Artificial, IA, existe a milhares de anos, mas foi a partir da década de 50 que o termo surgiu e, desde então, se tornou uma importante área de estudo, sendo utilizada nas indústrias, literatura, jogos, etc. No mundo dos jogos, a IA é utilizada para desafiar e entreter os jogadores. Existem diversas técnicas de IA e algumas dessas serão abordadas nesse trabalho como Máquina de Estados Finitos, Comportamento de Direções e Algoritmo Colônia de Formigas, ACF. Um jogo, que utiliza um sistema de movimentação em matriz, foi desenvolvido utilizando duas técnicas de IA: Máquina de Estados Finitos e Algoritmo Colônia de Formigas, com algumas adaptações. Após ajustes de parâmetros como intensidade de feromônio e taxa de evaporação, obteve-se um comportamento satisfatório, com o jogo rodando em uma matriz aberta, sem obstáculos e também conseguindo replicar o experimento das pontes duplas do algoritmo ACF.

Palavras-chave: Jogos Eletrônicos. Motores de Jogos. Inteligência Artificial. Máquina de Estados Finitos. Algoritmo Colônia de Formigas.

ABSTRACT

GONSALVES, Lucas Gabriel. **A study of application of artificial intelligence in games**. 2018. 61p. Course Completion Work (Technology in Analysis and Development of System) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2018.

Electronic games began to become popular in the early 1970s. There has been a great evolution since then. “Motores de Jogos”, Engines in English, are fundamental for the development of games, since they look for to accelerate the development of the games, aiding the developers. The concept of Artificial Intelligence, AI, exists for thousands of years, but it was from the 50's that the term came into being and has since become an important area of study, being used in industries, literature, games, etc. In the gaming world, AI is used to challenge and entertain players. There are several AI techniques and some of these will be approached in this work as Finite State Machine, Steering Behaviors and Ant Colony Algorithm, ACA. A game, which uses a grid based movement system, was developed using two AI techniques: Finite State Machine and Ant Colony Algorithm, with some adaptations. After adjustment of parameters such as pheromone intensity and evaporation rate, a satisfactory behavior was obtained, with the game running in an open, unobstructed grid, and also able to replicate the double bridge experiment of the ACA algorithm.

Keywords: Electronic Games. Game Engines. Artificial Intelligence. Finite State Machine. Ant Colony Algorithm.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Depois de brinquedos eletrônicos e máquinas como fliperama, surgiram jogos construídos em computadores. A partir daí a evolução desse setor foi contínua (KENT, 2001).

No início os jogos eram criados em *hardwares* especializados. Um jogo eletrônico era considerado um brinquedo comum. Atualmente os jogos são uma indústria multi-bilionária, rivalizando mercados como o de filmes de Hollywood e esses jogos são criados utilizando *engines*, ou motores de jogos em português, *softwares* criados com a ideia de facilitar o desenvolvimento de jogos (GREGORY, 2009).

O conceito de Inteligência Artificial, por sua vez, é muito novo, sendo um dos campos científicos mais recentes: o termo surgiu em 1956, proposto por John McCarthy, entretanto suas bases surgiram há muito tempo. Diversos campos ajudaram no desenvolvimento da Inteligência Artificial, entre eles estão filosofia, linguística, psicologia e biologia (RUSSELL; NORVIG, 2013).

O uso da IA está cada vez mais comum, seja com máquinas como robôs que facilitam o trabalho humano, ou com *softwares* em tarefas como reconhecimento de imagens, mineração de dados e outras aplicações (KISHIMOTO, 2004).

Dentro de jogos eletrônicos, a Inteligência Artificial pode ser útil de diversas formas, tornando esses aplicativos mais interessantes e divertidos. Isso pode ser ilustrado por meio de dois exemplos. Planet Coaster, um jogo de parque de diversões lançado em novembro de 2016 que atrai os jogadores pelo seu bom uso de Inteligência Artificial. Os personagens do jogo que visitam o parque demonstram reações muito humanas. Eles riem, se assustam, ficam com enjoo e se divertem com os brinquedos do parque[1]. F.E.A.R é um outro exemplo de boa aplicação de Inteligência Artificial. Os personagens controlados pela máquina impressionam. Os inimigos são capazes de se esconder atrás de objetos, se espalhar em grupos para

[1] SAMPAIO, Henrique. 2016. Acesso em: 23 Junho 2017, disponível em: < http://overloadr.com.br/especiais/2016/11/planet-coaster-evoca-aquela-sensacao-deliciosa-de-jogar-rollercoaster-tycoon-pela-primeira-vez/>

cercar o jogador e atacar e defender de maneira eficiente. O uso de IA em F.E.A.R torna o jogo mais desafiador e toda a experiência com o jogo mais satisfatória [2].

Técnicas de IA empregadas em jogos encontram aplicação também em outras áreas, como algoritmos de otimização (RUSSELL; NORVIG, 2013). A matriz curricular atual do curso de Análise e Desenvolvimento de Sistemas da UTFPR aborda parcialmente este campo de estudos, com a disciplina de 'Sistemas de Apoio a Decisão' que aplica técnicas como algoritmos genéticos para buscar padrões em dados. Assim, uma das motivações para a realização desse trabalho é obter uma visão geral sobre Inteligência Artificial, tema que possui grande importância no mundo dos jogos. Além disso, busca-se adquirir conhecimento também na área de desenvolvimento de jogos.

O trabalho inicia por estudar alguns algoritmos e técnicas de I.A., como Comportamento de Direções, Máquina de Estados Finitos e Algoritmo Colônia de Formigas, utilizadas em jogos eletrônicos, abordando essas técnicas e descrevendo em quais universos (gêneros, como jogos de corrida e aventura) de jogos e em quais situações são aplicados.

A partir dos estudos sobre Inteligência Artificial, foi implementado duas técnicas, Algoritmo Colônia de Formigas e Máquinas de Estados Finitos, em um jogo com movimentação em *grid*, uma grade ou matriz, por onde as formigas se movem. O *software* utiliza de IA para controlar as formigas que buscam comida pelo cenário e levam a comida até o formigueiro, depositando feromônio, substância química volátil utilizada por formigas reais para se comunicarem, no caminho, criando uma trilha de feromônio que se fortalece quanto mais formigas passam por ela. O papel do jogador é de atrapalhar as formigas, colocando pedras em seu caminho.

Espera-se que o conhecimento gerado através desse trabalho possa ser útil para outros alunos que pretendem realizar um trabalho sobre Inteligência Artificial e também para programadores que desejam utilizar IA em seus jogos, servindo como uma referência inicial para as técnicas e como elas são utilizadas.

[2] LOHMANN JÚNIOR, Fred. 2011. Acesso em: 23 Junho 2017, disponível em: < http://www.redegeek.com.br/2011/01/12/retro-review-f-e-a-r/>

1.1 OBJETIVOS

Nesta seção, serão apresentados o objetivo geral e os objetivos específicos deste trabalho.

1.1.1 Objetivo Geral

O objetivo geral é estudar algumas técnicas de IA e aplicar uma ou mais técnicas na implementação de um jogo de computador selecionado para o trabalho.

1.1.2 Objetivos Específicos

- Aplicar técnicas de Inteligência Artificial em um jogo, analisando os resultados e calibrar os parâmetros para se obter o comportamento desejado;
- Implementar uma ou miais técnicas de IA dentro de um jogo;
- Analisar os resultados obtidos, documentando os resultados obtidos com diferentes valores de parâmetros.

1.2 ORGANIZAÇÃO DO TRABALHO

Este Trabalho de Conclusão de Curso está dividido da seguinte maneira:

O Capítulo 2, Fundamentação Teórica, abordará de uma maneira geral todos os principais assuntos relacionados a este trabalho como: Jogos Eletrônicos, Motores de Jogos e Inteligência Artificial.

O Capítulo 3, Desenvolvimento, abordará assuntos como: motivo da escolha do jogo e das técnicas de IA escolhidas e como essas técnicas foram implementadas.

O Capítulo 4, Conclusão, abordará os resultados obtidos na realização do trabalho, as dificuldades encontradas e futuras versões do jogo.

2 FUNDAMENTAÇÃO TEÓRICA

Desde a década de 60 os jogos eletrônicos vêm se desenvolvendo e conquistando as pessoas, na forma dos chamados ‘fliperamas’, que inicialmente eram eletromecânicos. A partir da década de 70 começaram a surgir jogos utilizando computadores, como exemplificado pela empresa Atari (JÖRNMARK et al., 2005). A partir da década de 80, com a vinda dos jogos de fliperama para os videogames pessoais, as vendas de videogames pessoais começam a se popularizar nos Estados Unidos. Em 1984 a Nintendo lança o videogame Famicon (Nintendinho no Brasil) no Japão. Em 1986 a Nintendo começa a vender o Famicon no mundo todo. A década de 90 é marcada com o lançamento dos super famosos videogames Super Famicon (Super Nintendo no Brasil) , Playstation 1. No ano 2000 é lançado o campeão de vendas Playstation 2[3] no mercado japonês (KENT, 2001). No ano 2006 a Sony lança o videogame Playstation 3 e em 2011 o Playstation 4[4].

O salário médio de um desenvolvedor nessa indústria pode chegar a 10 mil dólares mensais [5]; assim, um jogo envolvendo 5 pessoas durante um ano e meio atingiria a faixa de 1 milhão de dólares. O desenvolvimento de jogos como GTA V chega a custar 100 milhões de dólares[6]. No Brasil em 2017 o número de jogadores era de 66.3 milhões e são esperados 75.7 milhões de jogadores até o fim de 2018. A movimentação financeira era de 1.3 bilhões de dólares em 2017 e espera-se que o valor alcance 1.5 bilhões de dólares até o fim de 2018[7].

Para melhorar a experiência do jogador, desenvolvedores começaram a aplicar técnicas de Inteligência Artificial em seus jogos, tornando-os mais realistas e desafiadores como o sistema “driving avatar” ou simplesmente Drivatar, utilizado na

[3] JORDÃO, Fabrio, 2015. Acesso em: 21 de novembro de 2018, disponível em: <https://www.tecmundo.com.br/video-game-e-jogos/88357-conheca-15-video-games-vendidos-historia-video.htm>

[4] JUNIOR, Milton, 2018. Acesso em: 21 de novembro de 2018, disponível em: <https://www.tricurioso.com/2018/07/28/playstation-conheca-a-sua-historia/>

[5] SCHREIER, Jason, 2017. Acesso em: 11 de setembro de 2018, disponível em: <https://kotaku.com/why-video-games-cost-so-much-to-make-1818508211>

[6] SINCLAIR BRENDAN, 2013. Acesso em: 21 de novembro de 2018, disponível em: <https://www.gamesindustry.biz/articles/2013-02-01-gta-v-dev-costs-over-USD137-million-says-analyst>

[7] DINO, 2018. Acesso em 25 de novembro de 2018, disponível em: <https://exame.abril.com.br/negocios/dino/o-crescimento-da-industria-de-games-no-brasil/>

série de jogos Forza, onde uma IA aprende a pilotar como o jogador e enfrenta outros jogadores no modo online do jogo[8].

A próxima seção abordará um panorama da história dos jogos eletrônicos e da aplicação de Inteligência Artificial no desenvolvimento de jogos.

2.1 JOGOS ELETRÔNICOS

A história dos jogos pode ser dividida entre as décadas de 70 até a atualidade, e também pelo uso de diferentes tecnologias utilizadas. O primeiro jogo eletrônico interativo criado na história foi o Spacewar (figura 1), onde 2 jogadores se enfrentavam utilizando duas naves espaciais diferentes. O jogo foi criado em 1961 por um estudante do MIT (Instituto de Tecnologia de Massachusetts), Steve Russell (KENT, 2001).

Figura 1 - Spacewar

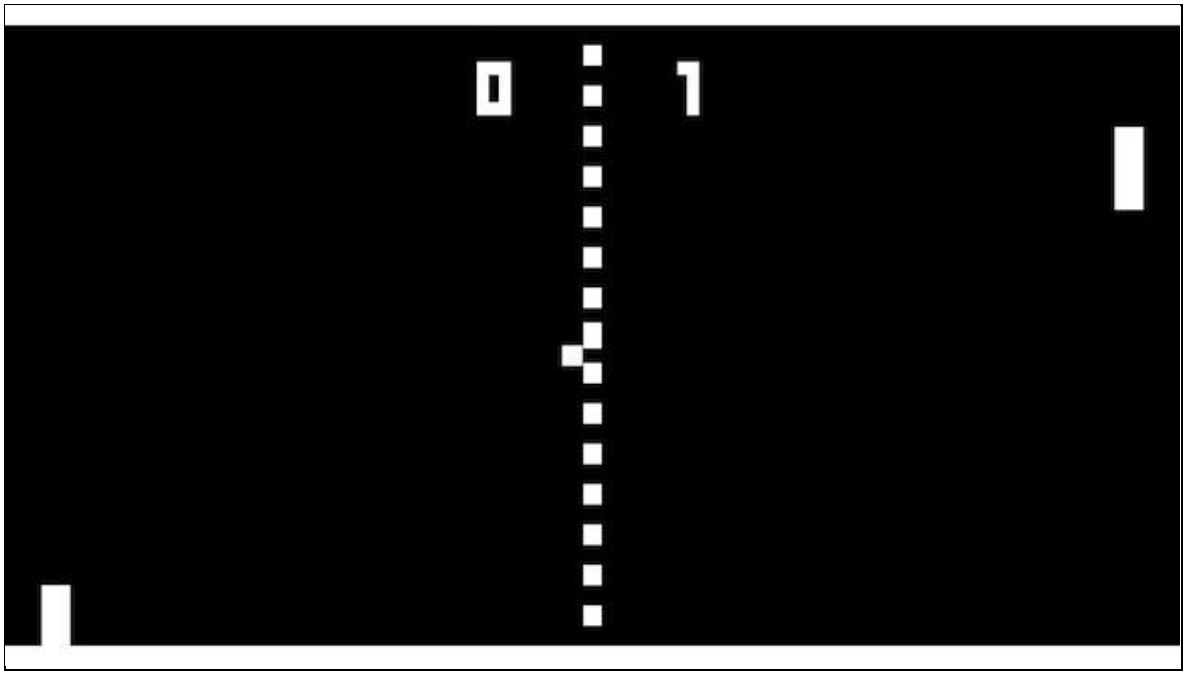
Fonte: Spacewar (2014)

Em 1970, Nolan Bushnell, começou a trabalhar em uma versão de fliperama do jogo Spacewar, chamado de Computer Space. Em 1971 a empresa Nutting Associates compra o jogo Computer Space de Bushnell e o contrata para trabalhar. Em seguida coloca no mercado a primeira máquina de fliperama (KENT, 2001).

Em 1972 Bushnell abre sua própria empresa, a Atari. O engenheiro da Atari, Al Alcorn, cria o famoso jogo Pong (KENT, 2001) (figura 2).

[8] Equipe Xbox, 2014. Acesso em: 21 de novembro de 2018, disponível em: <https://news.xbox.com/en-us/2014/09/30/games-forza-horizon-2-drivatars/>

Figura 2 - Famoso jogo da Atari, Pong



Fonte: Pong (2016)

Até o fim da década de 70, várias outras empresas entraram no mercado de jogos eletrônicos, como Taito, Midway (que viria a lançar em 1978 o famoso jogo Space Invaders) e Capcom (KENT, 2001).

Durante a década de 80 os jogos de fliperama estavam em seu auge com o lançamento de Pac Man, o jogo de fliperama mais famoso do mundo, Defender e Donkey Kong, ao mesmo tempo em que foram lançados os primeiros videogames 8-bit: Famicon (também conhecido como NES ou Nintendinho no Brasil), da Nintendo e Master System, da SEGA (KENT, 2001).

A década de 90 foi marcada pelo lançamento dos videogames de 16-bits: Super Famicon (Super Nintendo ou SNES) e Sega Genesis. Em 1994 a Sony lança o Playstation e a SEGA lança o SEGA Saturn (videogames 32-bit) ambos no Japão. Em 1995 ambas as empresas lançaram seus videogames nos Estados Unidos (KENT, 2001).

No final da década de 90 e começo dos anos 2000, a Sony e a Nintendo anunciam seus videogames de 128-bits, Playstation 2 e GameCube, respectivamente. A Microsoft também entra no mercado de jogos com seu videogame Xbox (KENT, 2001).

Em 2005 a Microsoft anuncia seu novo videogame, Xbox 360 e em 2006 a Sony anuncia o Playstation 3. Em 2013 a Sony anuncia o Playstation 4 e a Microsoft anuncia o Xbox One.

O *hardware* desses equipamentos evoluiu ao longo do tempo. As GPUs (*Graphical Processing Units*) são provavelmente os componentes mais significativos, atingindo hoje em dia a casa de várias centenas de núcleos paralelos em uma única placa (NIKKOLS, 2010).

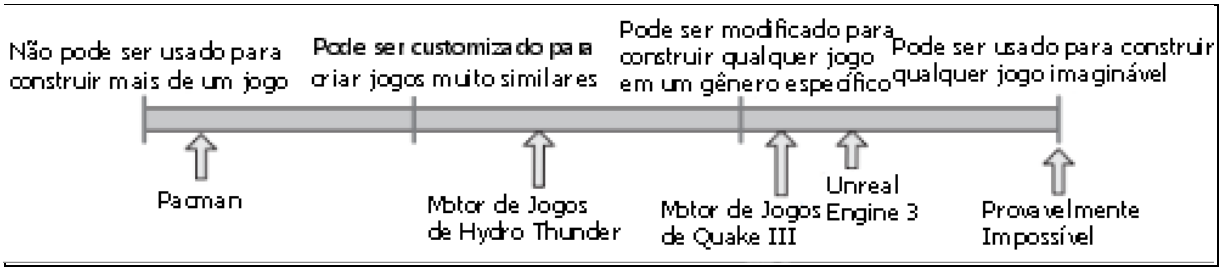
2.2 MOTORES DE JOGOS

O primeiro jogo eletrônico foi construído inteiramente em *hardware* especializado. Com o passar dos anos o desenvolvimento de microprocessadores tornou possível uma nova abordagem. Hoje em dia os jogos são executados em computadores e videogames pessoais e o uso desses *softwares* impulsionou o desenvolvimento de uma grande variedade de jogos (GREGORY, 2009).

Motores de jogos, ou *engines* em inglês, podem ser resumidos como bibliotecas que contém o código para várias funções básicas de um jogo, como tratar o *hardware* gráfico, receber entradas do usuário, cuidar de renderização (CLUA, 2005).

O termo “motor de jogos” surgiu na década de 90 se referindo a jogos de tiro em primeira pessoa como o Doom. Doom foi desenvolvido com uma divisão bem definida de seus principais componentes. Seu sistema de renderização 3D, sistema de detecção de colisão e sistema de áudio ficavam claramente divididos dos recursos de arte, fases e regras do mundo. A importância da separação desses elementos ficou evidente quando desenvolvedores começaram a recriar esses jogos, criando novas armas, novos inimigos, novos cenários, tudo isso sem modificar os elementos principais do jogo. Isso marcou o nascimento das comunidades de modificação dos jogos. Desenvolvedores usavam ferramentas disponibilizadas pelos criadores do jogo para modificar vários elementos e criar jogos novos com base no jogo original (GREGORY, 2009). A figura 3 mostra a evolução nos motores de jogos e sua reusabilidade.

Figura 3 - Reusabilidade de um motor de jogos



Fonte: Adaptado de Gregory (2009)

Jogos antigos, anteriores a Pacman, não podiam ser reaproveitados. Os primeiros motores permitiam a customização para criar jogos muito similares. Hydro Thunder foi um de corrida de barcos do videogame Sega Dreamcast, lançado em 1999. Quake III foi um jogo de tiro em primeira pessoa lançado no final de 1999. Unreal Engine 3 é um motor de jogos lançado em 2005.

Engines atuais, como Unreal, Unity e Godot, permitem a criação de jogos de gêneros diferentes, entretanto, quanto maior for a variedade de gêneros que o motor permite criar, inferior é a otimização para esses gêneros (GREGORY, 2009).

As próximas seções irão abordar um pouco sobre os três motores mencionados.

2.2.1 Unity

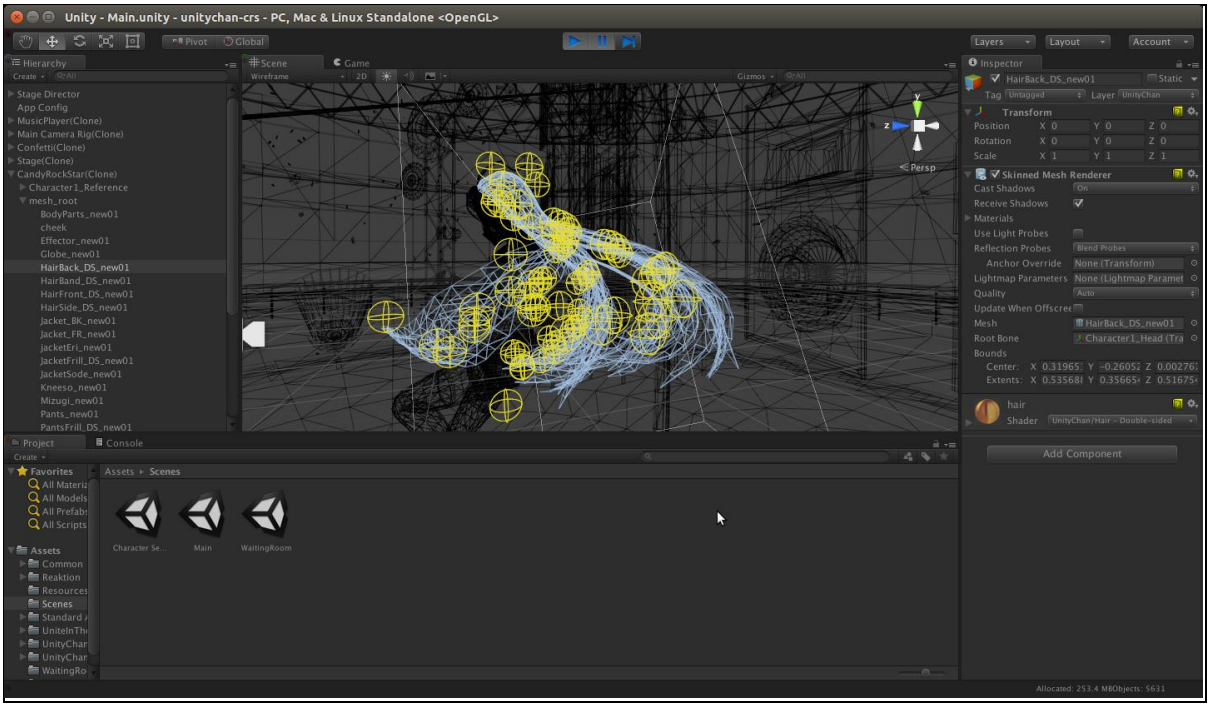
Ferramenta gratuita, de código fechado, ou seja, o usuário não pode editar a estrutura interna do motor. Ao alcançar um faturamento de 100 mil dólares por ano, o usuário precisa pagar uma mensalidade de 35 dólares. A versão gratuita não conta com suporte da Unity[9].

O motor Unity permite o desenvolvimento de jogos 3D e 2D. O desenvolvedor tem a possibilidade de programar com C# e Javascript.

A figura 4 mostra a interface de desenvolvimento do motor Unity.

[9] Acesso em: 25 de novembro de 2018, disponível em: <https://store.unity.com/>

Figura 4 - Interface do motor Unity



Fonte: Interface Unity (2015)

Alguns exemplos de jogos feitos com a Unity:

- Endless Space 2;
- Enter the Gungeon;
- Ori and the Blind Forest;
- Superhot;
- Cities Skylines.

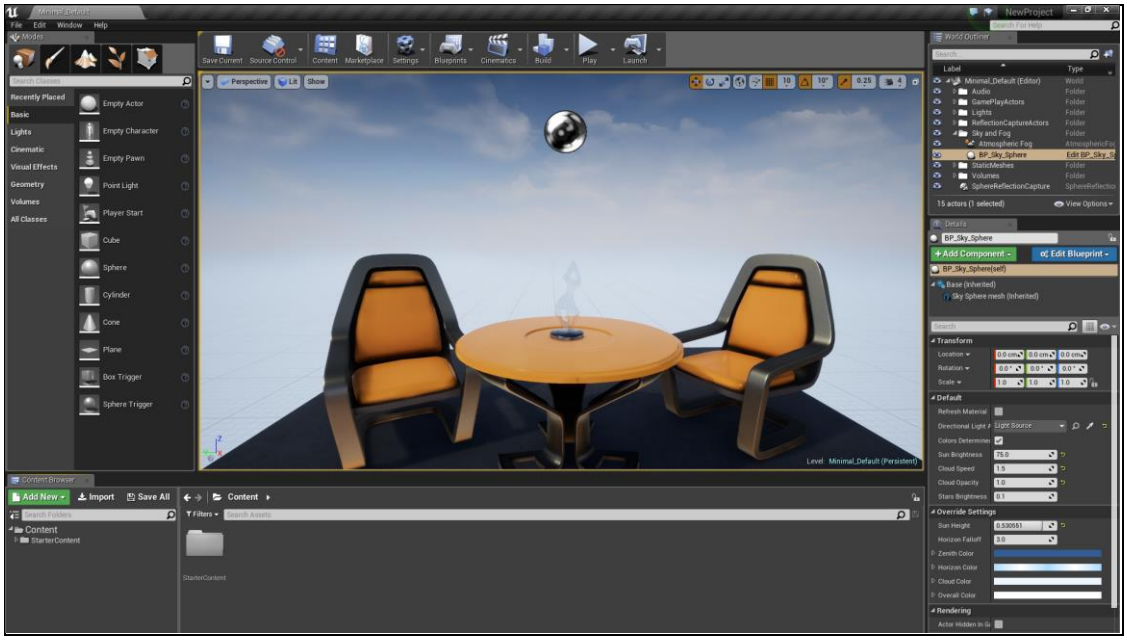
2.2.2 Unreal

Ferramenta inteiramente grátis de desenvolvimento de jogos 2D e 3D de código fechado. O desenvolvedor não possui limitações, entretanto, caso o jogo desenvolvido na Unreal consiga arrecadar 3 mil dólares por jogo por trimestre, o usuário da ferramenta deve pagar uma taxa de ganhos (5%) à EpicGames, desenvolvedora da Unreal[10].

[10] Acesso em: 25 de novembro de 2018, disponível em: <https://www.unrealengine.com/en-US/release>

A figura 5 mostra a interface de desenvolvimento do motor gráfico Unreal Engine.

Figura 5 - Interface do motor Unreal Engine



Fonte: Interface Unreal (2018)

Alguns exemplos de jogos feitos na Unreal Engine:

- Borderlands 2;
- Spec Ops: The Line;
- Batman Arkham City;
- Rime;
- Tekken 7;
- EVE: Valkyrie.

2.2.3 Godot

Godot é um programa de desenvolvimento de jogos 2D e 3D, completamente livre e de código aberto (Open Source) desenvolvido pela comunidade distribuído sob a licença MIT. Os jogos desenvolvidos no Godot podem ser exportados para Linux, Mac e Windows, assim como para dispositivos móveis como Android e iOS e plataformas baseadas em HTML5 (GODOT, 2017).

A ferramenta vem com diversos projetos de jogos, códigos demonstrativos de jogos 2D mostrando movimentação de personagens e algumas funcionalidades da ferramenta, para que o desenvolvedor possa estudar e aprender como a ferramenta funciona.

A figura 6 mostra a interface de desenvolvimento do motor gráfico Godot.

Figura 6 - Interface do motor Godot

Fonte: Interface Godot (2016)

Alguns exemplos de jogos feitos com Godot:

- Towns of the dead;
- RPG in a box;
- Dolphin Island 2.

Godot foi a ferramenta escolhida para o desenvolvimento do jogo neste trabalho por uma série de vantagens:

- Ferramenta sob a licença MIT, grátis e de código aberto. O motor Unity possui uma versão grátis, entretanto com algumas limitações como uma tela inicial não customizável. O motor Unreal tem a vantagem de não possuir limitações, entretanto quando o jogo atinge certa arrecadação – três mil dólares em um trimestre, o desenvolvedor deve pagar uma taxa para os criadores do motor[11];

[11] Acesso em: 11 de setembro de 2018, disponível em: <https://www.unrealengine.com/faq>

- Possui 2D dedicado. Unreal e Unity simulam um desenvolvimento 2D;
- Um projeto vazio no Unreal pode ter 200 MB enquanto o Godot constrói projetos com cerca de 20 MB.

2.3 VISÃO GERAL DO MOTOR GODOT

As próximas seções abordarão o funcionamento do sistema de *Nodes* e *Scenes*, partes fundamentais na criação de um jogo utilizando Godot, e sua linguagem: GDScript.

2.3.1 *Nodes*

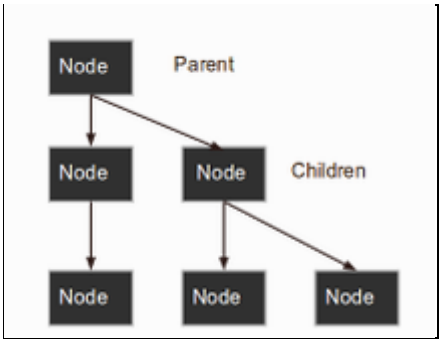
Um *node*, ou nó, é um elemento básico na criação de jogos no Godot. Pode ser entendido como um container para objetos, como: um personagem do jogo; uma parede ou obstáculo; uma fonte de som.

Um *node* tem as seguintes características:

- Possui um nome;
- Possui propriedades editáveis;
- Pode receber um chamado de processo a cada *frame*;
- Pode ser estendido para ter mais funções;
- Pode ser adicionado a outros *nodes* como filhos.

Quando organizados dessa maneira cria-se uma árvore de nódulos conforme figura 7.

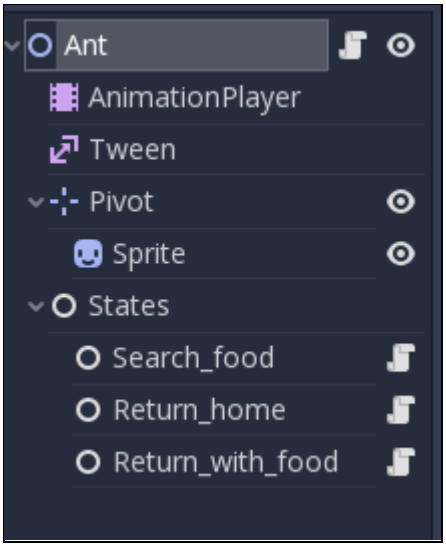
Figura 7 - Sistema de *nodes* do Godot



Fonte: Nós e Cenas no Godot (2018)

Um exemplo disso seria associar um player de animação a um personagem, conforme elemento AnimationPlayer mostrado na figura 8, que é um elemento de animação associado ao nó Ant.

Figura 8 - *Player* de animações da formiga



Fonte: Autoria Própria

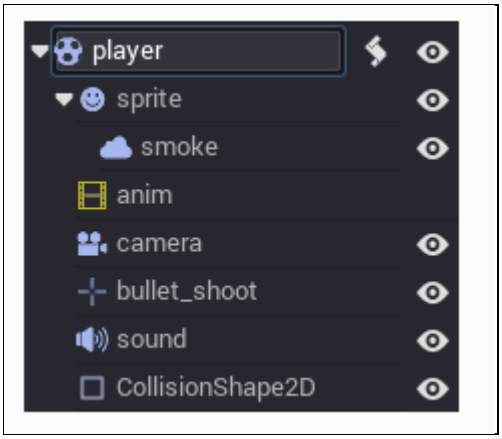
2.3.2 *Scenes*

Uma cena, conforme Figura 9, é um grupo de nós organizados de maneira hierárquica. Possui as seguintes características:

- Uma cena sempre tem um nó raiz;
- Cenas podem ser gravadas e carregadas em disco;
- Cenas podem ser instanciadas. É possível criar uma versão base, uma classe, de uma cena para carregá-la quantas vezes for necessário;
- Executar um jogo significa rodar uma cena;
- Podem existir diversas cenas em um projeto, mas para o projeto rodar é necessário selecionar uma cena principal para o jogo executar.

A figura 9 mostra a estrutura de cenas presente no Godot.

Figura 9 - Sistema de scenes no Godot



Fonte: Nós e Cenas no Godot (2018)

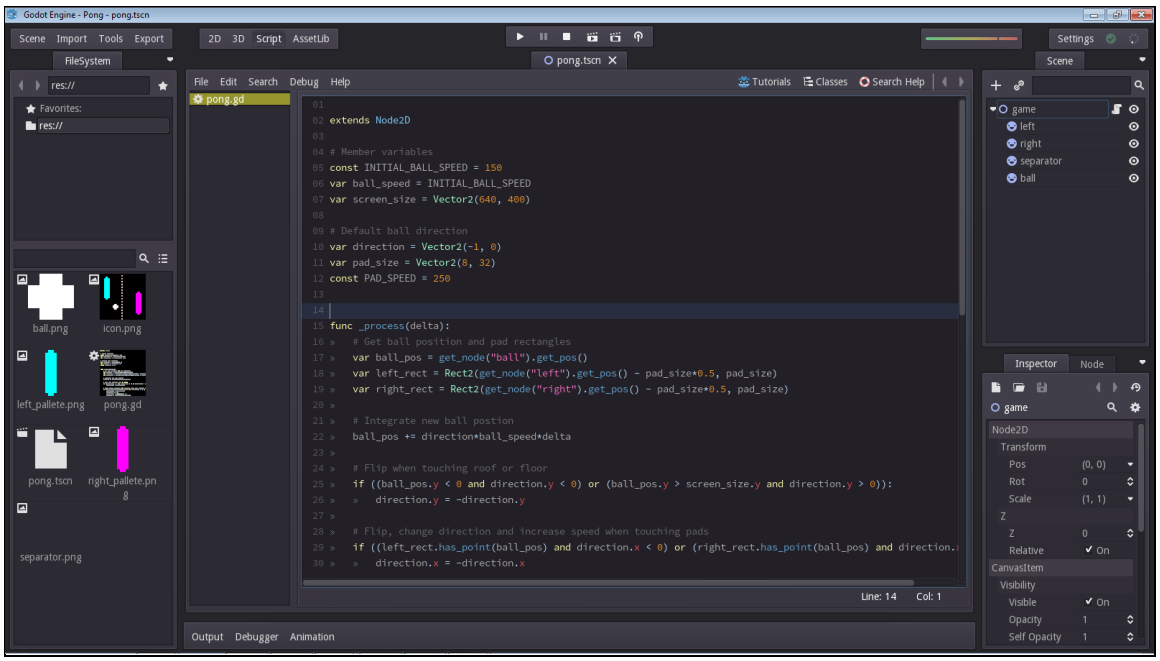
2.3.3 GDScript

A linguagem de *script* do Godot é o GDScript[12]. Está é uma linguagem baseada no Pyton, com sintaxe parecida, conforme mostrado na figura 10. Os objetivos do GDScript são:

- Manter o código simples, familiar e de fácil aprendizagem;
- Manter o código legível e seguro de erros.

[12] Acesso em: 12 de setembro de 2018, disponível em: <http://docs.godotengine.org/en/stable/learning/step_by_step/scripting.html>

Figura 10 - Codificando no Godot



Fonte: Autoria Própria

Como é uma linguagem interpretada (sem compilação) há um impacto no desempenho, entretanto, os códigos e funções mais pesados, como apresentação de gráficos e testes de colisão, já são escritos em C++ no motor. Caso seja necessário ter maior desempenho em algum algoritmo específico do jogo, é possível escrever código em C++, substituindo uma classe GDScript por uma classe C++, sem afetar o resto do código.

2.4 INTELIGÊNCIA ARTIFICIAL

O termo Inteligência Artificial foi criado em 1956, por John McCarthy, durante um encontro de dois meses em Dartmouth onde 10 pesquisadores, entre eles Trenchard More, de Princeton, Arthur Samuel, da IMB, e Ray Solomonoff e Oliver Selfridge, do MIT, se reuniram para discutir sobre Inteligência Artificial. O estudo se basearia na ideia de que cada aspecto da aprendizagem ou qualquer outra característica da inteligência pode ser descrita de maneira tão precisa que seria possível criar uma máquina para simulá-la (RUSSELL; NORVIG, 2013).

A ideia de Inteligência Artificial é discutida há milênios. Aristóteles (384-322 a.C.) desenvolveu um sistema de silogismos que permitiriam gerar conclusões

mecanicamente, dadas as premissas iniciais. Seus estudos deram início ao campo chamado de lógica, um dos principais campos da IA (RUSSELL; NORVIG, 2013).

Não existe uma definição exata para Inteligência Artificial. O quadro 1 mostra oito definições de I.A.

Quadro 1 - Oito definições de Inteligência Artificial, organizadas em quatro categorias

Pensando como um humano	**Pensando racionalmente**
"O novo e interessante esforço para fazer os computadores pensarem (...) *máquinas com mentes*, no sentido total e literal." (Haugeland, 1985) "[Automatização de] atividades que associamos ao pensamento humano, atividades como a tomada de decisões, a resolução de problemas, o aprendizado..." (Bellman, 1978)	"O estudo das faculdades mentais pelo uso de modelos computacionais." (Charniak e McDermott, 1985) "O estudo das computações que tornam possível perceber, raciocinar e agir." (Winston, 1992)
Agindo como seres humanos	**Agindo racionalmente**
"A arte de criar máquinas que executam funções que exigem inteligência quando executadas por pessoas." (Kurzweil, 1990) "O estudo de como os computadores podem fazer tarefas que hoje são melhor desempenhadas pelas pessoas." (Rich and Knight, 1991)	"Inteligência Computacional é o estudo do projeto de agentes inteligentes." (Poole *et al*., 1998) "AI... está relacionada a um desempenho inteligente de artefatos." (Nilsson, 1998)

Fonte: Russell; Norvig (2013)

As definições mostradas no quadro 1 se relacionam a processos de pensamento e raciocínio, as partes de baixo se relacionam ao comportamento. As definições do lado esquerdo definem a inteligência do agente comparando-o a um humano, já as definições do lado direito mede o sucesso do agente comparando-o a um conceito ideal de inteligência, a racionalidade. Um agente é racional se ele "faz a coisa certa", dado o que ele sabe (RUSSELL; NORVIG, 2013).

A Inteligência Artificial pode ser dividida em dois campos: IA forte, onde uma máquina tenta imitar os processos de pensamento de um ser humano e IA fraca, onde uma máquina se torna muito boa em fazer uma atividade específica como dirigir ou realizar uma cirurgia (BUCKLAND, 2004). A comunidade acadêmica busca fazer com que máquinas inteligentes realizem processos complexos que somente um ser humano é capaz de realizar.

2.4.1 O Teste De Turing

O teste de Turing, proposto por Alan Turing (1950), foi projetado para definir se uma máquina é inteligente. O teste consiste em uma conversação entre uma máquina e uma pessoa. A máquina passa no teste se conseguir enganar a pessoa, ou seja, fazer com que a pessoa acredite estar conversando com outro ser humano, por pelo menos 30% do tempo (RUSSELL; NORVIG, 2013).

Com o passar do tempo, desenvolvedores de IA deixaram de tentar criar agentes que conseguissem passar no teste de Turing e começaram a focar seus esforços em criar agentes capazes de realizar, de maneira ótima, tarefas específicas. Exemplificando esse raciocínio, o desafio do “voo artificial” teve sucesso quando pesquisadores pararam de tentar imitar um pássaro e começaram a estudar aerodinâmica. O objetivo de um avião não é enganar outros pombos, mas sim realizar de maneira eficiente uma tarefa (RUSSELL; NORVIG, 2013).

2.5 INTELIGÊNCIA ARTIFICIAL NOS JOGOS

Jogos de computador possuem requisitos de *hardware* e de tempo de execução. Assim, a IA utilizada nos jogos deve se adequar a esses requisitos (BUCKLAND, 2004).

Muitos jogos de computador utilizam alguma técnica de IA, geralmente buscando dar mais variabilidade de ações e evitando situações repetitivas (MORAIS 2009). A IA em um jogo deve ser subótima, ou seja, não devem vencer o jogador o tempo todo, ela deve ser capaz de desafiar o jogador, fazer com que ele se sinta inteligente e superior, uma boa IA deve perder mais do que ganha. Esse conceito é contrário ao conceito da IA em outras aplicações como na indústria, onde a máquina deve desempenhar perfeitamente o seu papel (BUCKLAND, 2004).

As próximas seções irão apresentar algumas técnicas de IA que podem ser implementadas em jogos de computador.

2.5.1 Máquina de estados finitos

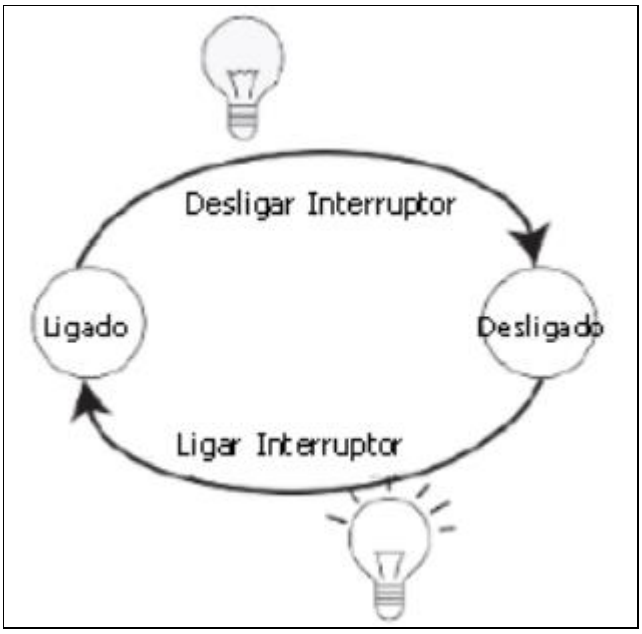
Máquinas de estados finitos estão nos jogos desde os primórdios dos jogos eletrônicos. Por muitos anos essa técnica tem sido escolhida pelos desenvolvedores para atribuir a ilusão de inteligência aos jogos (BUCKLAND, 2004). Uma máquina de estados finitos pode ser parte de uma solução de IA mais complexa, como máquinas hierárquicas (PARR, RUSSELL, 1998).

Desenvolver uma máquina de estados finitas é rápido e simples, é fácil de debugar, necessita de pouco processamento, é intuitivo e flexível (BUCKLAND, 2004).

Uma máquina de estados finitos pode ser descrita como um dispositivo, ou um modelo de dispositivo, que possui um número finito de estados em que pode estar a qualquer momento e que pode operar com entrada de dados para realizar ações (BUCKLAND, 2004).

Um interruptor de uma lâmpada é uma máquina de estados finitos (figura 11). Ela possui dois possíveis estados, ligado e desligado, e só pode ter um estado por vez. Ao mover o interruptor para a posição de ligar, é permitido a passagem de energia, o que acende a lâmpada, essa é a ação gerada a partir da entrada de dados – ligar o interruptor, movendo o interruptor para o outro lado é cortada a passagem de energia e a lâmpada apaga.

Figura 11 - Um interruptor de uma lâmpada é uma máquina de estados finitos

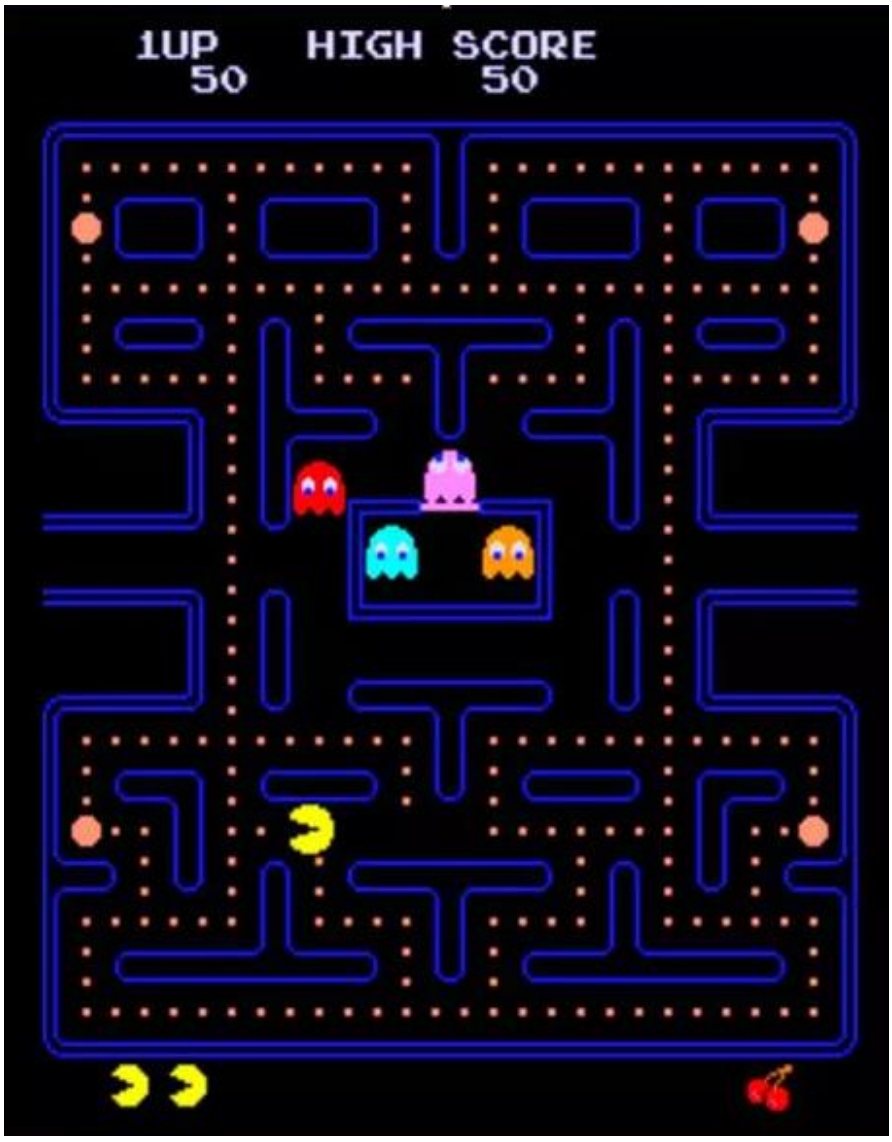


Fonte: Adaptado de Buckland (2004)

Uma máquina de estados em um jogo eletrônico é mais complexa do que um interruptor de energia, mas funciona segundo os mesmos princípios.

No jogo Pacman, conforme figura 12, os fantasmas que perseguem o jogador são máquinas de estados finitos. Todos os fantasmas possuem um estado Evitar, que é implementado de maneira igual para todos, e cada fantasma possui um estado Perseguir, que é implementado de maneira diferente para cada fantasma. O jogador comer uma pílula de poder é a ação necessária para mudar o estado dos fantasmas de Perseguir para Evitar. Um temporizador é a entrada de dados necessária para a mudança do estado Evitar para o estado Perseguir (BUCKLAND, 2004).

Figura 12 - Jogo Pacman



Fonte: Pacman (2018)

Personagens controlados pelo computador, NPC (Non-Playable Character), em jogos de Estratégia em Tempo Real (RTS, Real Time Strategy) como Warcraft também são implementados como máquinas de estados finitos. Eles possuem estados como MoverParaPosição, Patrulhar e SigaOCaminho.

Existem diversas maneiras de se implementar uma máquina de estados finitos. Uma dessas maneiras é utilizando if-then, entretanto essa pode não ser uma boa maneira caso o projeto cresça e mais estados sejam implementados o código se torna confuso. Uma melhor maneira de implementar uma máquina de estados finitos é utilizando uma tabela de transição de estados, conforme quadro 2.

Quadro 2 - Tabela de transição de estados

Current State	Condition	State Transition
Runaway	Safe	Patrol
Attack	WeakerThanEnemy	RunAway
Patrol	Threatened AND StrongerThanEnemy	Attack
Patrol	Threatened AND WeakerThanEnemy	RunAway

Fonte: Buckland (2004)

Utilizando uma tabela de transição de estados é possível fazer com que o personagem mude o estado de acordo com o estímulo recebido. Essa abordagem é mais flexível que a que utiliza if-then, caso se deseje inserir um novo estado, basta programá-lo (como um novo objeto) e inseri-lo na tabela e reorganizar as condições. Um exemplo disso seria um robô aspirador de pó automático. Ele teria estados como ProcurarSujeira e LimparSujeira e a tabela de estados é verificada a cada instante. Se, enquanto o robô anda pelo local, ele encontrar sujeira através de algum sensor, ele muda o estado para LimparSujeira, assim que o sensor não detectar mais sujeira, a tabela é verificada e muda o estado de LimparSujeira para o estado ProcurarSujeira novamente. Na primeira maneira de implementação, caso deseje-se inserir novos estados, como por exemplo um estado RecarregarEnergia, onde o robô vai até a tomada e começa a recarregar automaticamente, seria preciso colocar mais condições de teste. É preciso cuidado para o código não ficar caótico (BUCKLAND, 2004).

Uma terceira abordagem é a de regras embutidas. Essa abordagem permite que cada estado seja capaz de acionar uma mudança de estado automaticamente. Essa abordagem tem a vantagem de não precisar utilizar uma tabela. As condições ficam dentro de cada estado, dessa forma, você poderia programar o robô para verificar seus níveis de energia, enquanto no estado de ProcurarSujeira, e se necessário, trocar para o estado RecarregarEnergia. A segunda e terceira abordagens permitem uma organização melhor do código e menos trabalho para inserir novos estados (BUCKLAND, 2004).

2.5.2 Agentes

Existem muitas definições diferentes para agente e discuti-las foge do escopo deste trabalho. Será utilizada aqui a definição de agente segundo Buckland (2004), podemos aplicar no desenvolvimento de jogos: um agente autônomo é um sistema situado em um ambiente, capaz de perceber esse ambiente e agir de acordo com seus objetivos pré-estabelecidos.

A ideia de agente se encaixa com a de personagem dentro de um videogame. Se um personagem em um jogo encontra um obstáculo, ele seria capaz de reagir a isso de uma maneira apropriada. Por exemplo, se um carro de corrida encontra um veículo parado na pista, ou uma parede em seu caminho, ele deveria modificar seu curso para desviar do obstáculo. Outro exemplo seria um atacante que persegue o jogador dentro de um labirinto. Esses agentes agem “por si próprios”, de acordo com o que acontece ao longo do jogo.

De acordo com Buckland (2004), a movimentação de um agente autônomo em um jogo pode ser dividida em 3 partes:

- Seleção de ação: O agente seleciona qual objetivo completar, como “ir para posição A” e “fazer isso, depois aquilo”;
- Direção: O agente escolhe a trajetória que irá seguir para atingir seu objetivo. O comportamento de direções é a implementação dessa parte;
- Locomoção: Essa é a parte de movimentação do agente. Se é dado o comando de “ir para o norte” para um camelo, um carro e um avião, os três vão se mover de maneira diferente, mas todos têm o mesmo objetivo. Cada agente pode implementar uma locomoção um pouco diferente.

2.5.3 Comportamento de direções

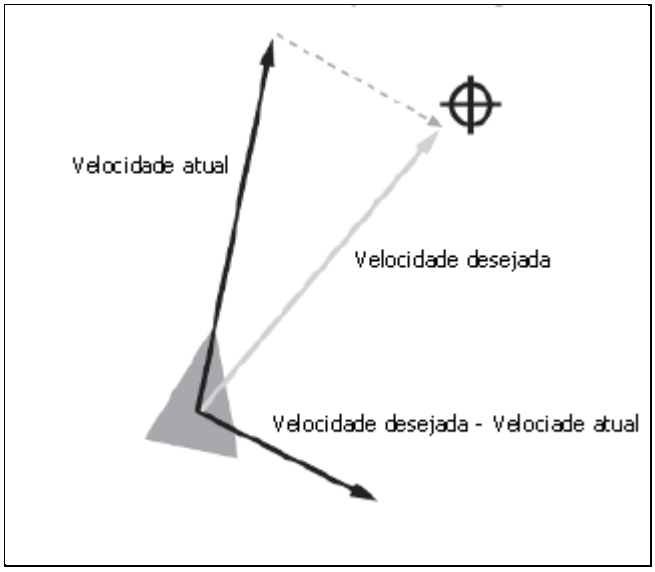
A movimentação de personagens é uma parte importante de um jogo de computador. Exemplificando, se um personagem patrulha uma porta apenas fazendo um movimento de ida e volta, isso pode tornar o cenário cansativo e entediante. Se esse personagem incorporar alguma aleatoriedade de direção, atrairá mais a atenção. E se o personagem interagir com o jogador, por exemplo perseguindo-o quando estiver em uma linha de visão, passará a proporcionar desafio ao usuário.

Existem diversas técnicas básicas para controlar movimentação de um agente dentro de um jogo. Embora simples, essas técnicas mostram como programar o comportamento de um agente em função do que acontece com outros elementos do jogo, como agentes inimigos ou alvos.

2.5.3.1 *Seek* (Procurar)

O comportamento Procurar pode ser implementado calculando uma força que direciona um agente para uma posição alvo. Isso é feito calculando a diferença de posição entre alvo e agente e ajustando a velocidade desejada, conforme figura 13.

Figura 13 - Calculando os vetores do comportamento Procurar



Fonte: Adaptado de Buckland (2004)

O ponto marcado é o alvo, ao subtrair a velocidade desejada pela velocidade atual obtém-se uma diferença que deve ser aplicada ao atual movimento do personagem.

“Procurar” é um comportamento muito útil e várias outras técnicas de comportamento de direção farão uso dela.

2.5.3.2 *Flee* (Fuga)

O comportamento Fugir é oposto de Procurar. Ao invés de produzir uma força para levar o agente até um determinado ponto, Fugir cria um vetor apontando para a direção contrária.

O código e a lógica de implementação são os mesmos do comportamento Seek.

2.5.3.3 *Arrive* (Chegar)

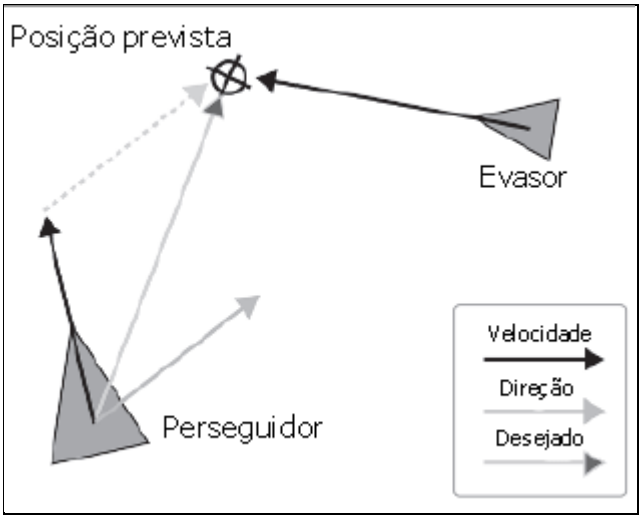
O comportamento Procurar é muito útil para fazer um agente alcançar uma posição, entretanto não há desaceleração. O comportamento Chegar define o comportamento de um agente de maneira que haja desaceleração para atingir o objetivo. O comportamento Chegar leva em conta a desaceleração do agente para calcular em quanto tempo o agente chegará ao objetivo.

Isto pode ser implementado calculando a distância até o objetivo e reduzindo a velocidade de maneira inversamente proporcional à essa distância.

2.5.3.4 *Pursuit* (Perseguição)

Perseguir é uma técnica usada para interceptar um alvo em movimento. Caso fosse escolhido Procurar como técnica, o desenvolvedor não passaria a impressão de inteligência pois o personagem estaria o tempo todo visando a posição atual do alvo; dependendo da movimentação essa estratégia pode nunca ter sucesso. Utilizando perseguir, o personagem estabelece como alvo uma posição futura do objeto sendo perseguido, conforme figura 14.

Figura 14 - Prevendo posição do alvo



Fonte: Adaptado de Buckland (2004)

Uma das dificuldades de se programar essa técnica é saber quanto tempo no futuro o agente deve prever para poder alcançar o seu alvo.

2.5.3.5 *Evade* (Desviar)

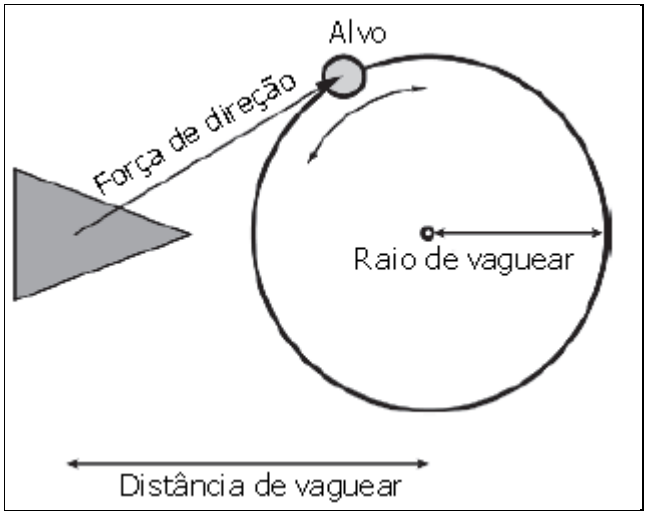
O comportamento Evitar é o oposto de Perseguir. Ao invés de ir em direção ao alvo, o agente deve prever a direção de perseguição e evitá-la.

2.5.3.6 *Wander* (Vaguear)

O comportamento Vaguear é muito importante para passar a jogadores uma impressão de inteligência. Esse comportamento é utilizado para fazer com que o agente ande de maneira aleatória pelo ambiente, mas de forma convincente.

Uma maneira inicial de se programar isso seria calculando direções aleatórias para mover o agente, mas isso criaria um movimento “tremido”. Uma aproximação melhor é projetar um círculo na frente do agente e fazer ele se mover pelo perímetro do círculo, mostrado na figura 15.

Figura 15 - Movendo alvo pelo perímetro de um círculo



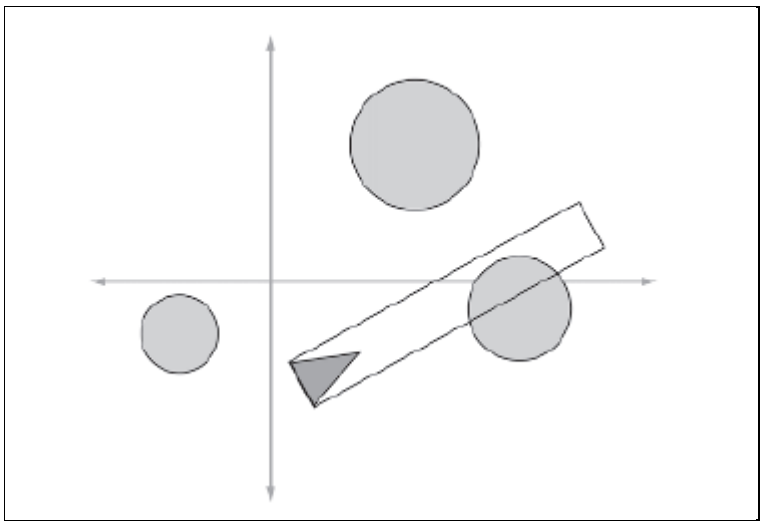
Fonte: Adaptado de Buckland (2004)

Utilizar essa técnica permite muitas possibilidades de movimentos. É possível fazer o agente se mover de maneira suave ou de maneira violenta, isso só depende do tamanho do círculo.

2.5.3.7 *Obstacle Avoidance* (Evasão de Obstáculo)

Evasão de Obstáculos é um comportamento que direciona um agente para evitar obstáculos em seu caminho. Um obstáculo pode ser representado por um círculo; o objetivo do comportamento é criar uma caixa de detecção a partir do personagem e, se algum objeto entrar nessa área, direcioná-lo para evitar a colisão com o objeto, conforme figura 16.

Figura 16 - Caixa de detecção



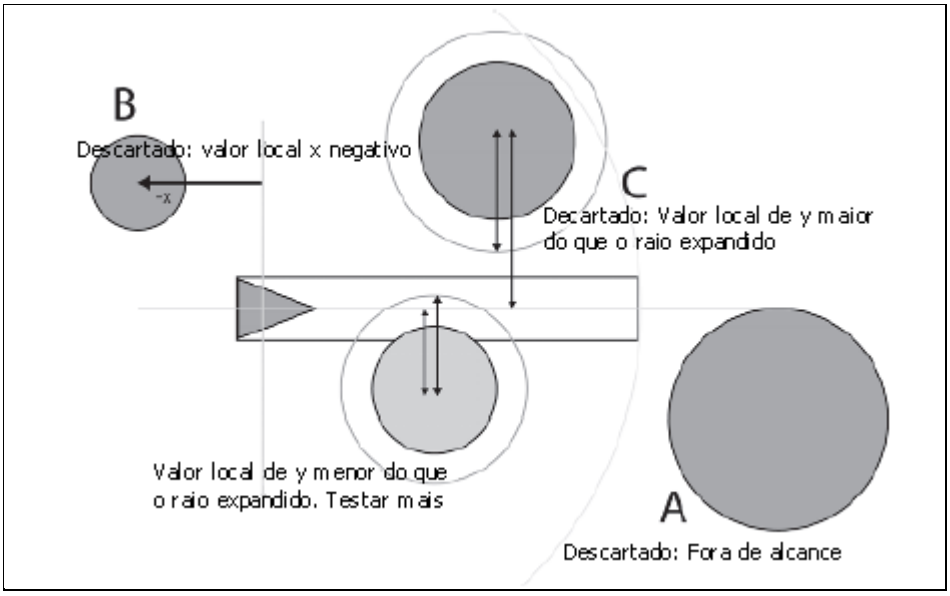
Fonte: Buckland (2004)

A largura do retângulo é igual ao círculo de detecção projetado em volta do personagem e o comprimento é igual à velocidade do agente. Quanto mais rápido o personagem estiver se movendo, maior deve ser a caixa de detecção dele.

Checar a interseção com outros objetos pode ser dividido em quatro etapas:

- O personagem deve considerar apenas os objetos dentro do alcance de sua caixa de detecção. O algoritmo deve inicialmente iterar entre todos os objetos do ambiente e marcar os objetos dentro desse alcance;
- Calcular a posição dos objetos em relação ao eixo ao longo do qual o personagem se move. Qualquer objeto que tenha posição x negativa é descartado;
- Verificar se algum desses objetos está dentro da caixa de detecção. O raio de detecção dos objetos deve ser expandido pela metade da largura da caixa de detecção do agente. Se o valor y do objeto for maior que o tamanho da caixa, pode ser descartado;
- Finalmente, deve ser encontrado o ponto de interseção mais próximo do agente, conforme figura 17.

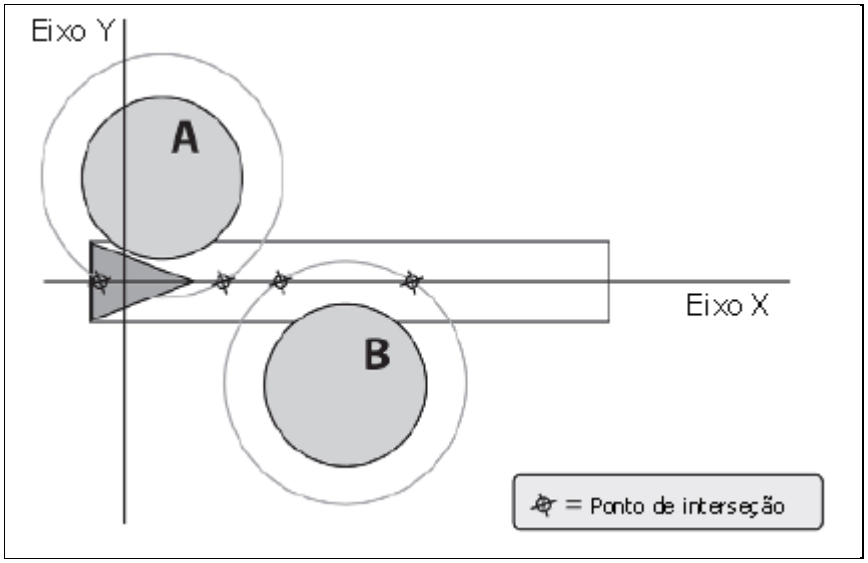
Figura 17 - Interseção no eixo y



Fonte: Adaptado de Buckland (2004)

Depois que são encontrados os pontos de interseção no eixo x, é necessário calcular a força para direcionar o personagem de maneira a evitar o toque no objeto , mostrado na figura 18.

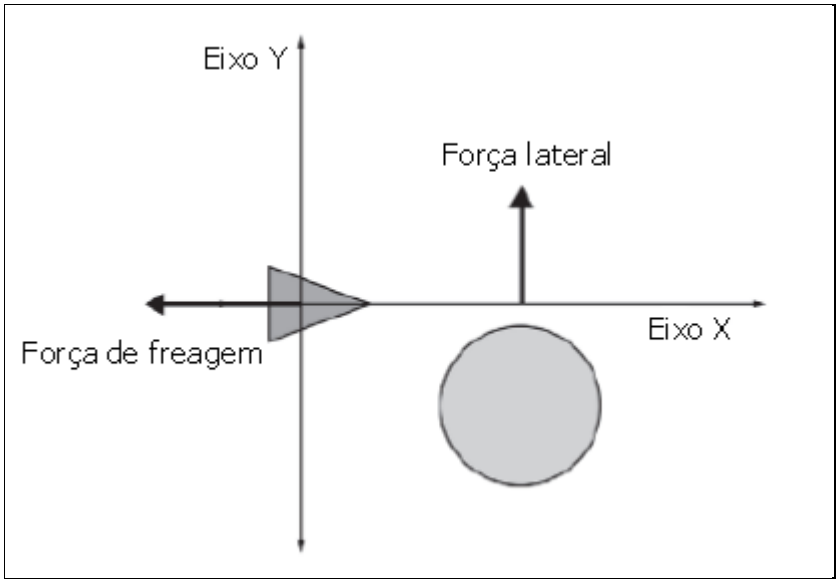
Figura 18 - Teste de ponto de interseção



Fonte: Adaptado de Buckland (2004)

São calculadas duas forças, uma força para frear o objeto e outra para mover o objeto para longe do obstáculo, conforme figura 19.

Figura 19 - Calculando força para evitar colisão

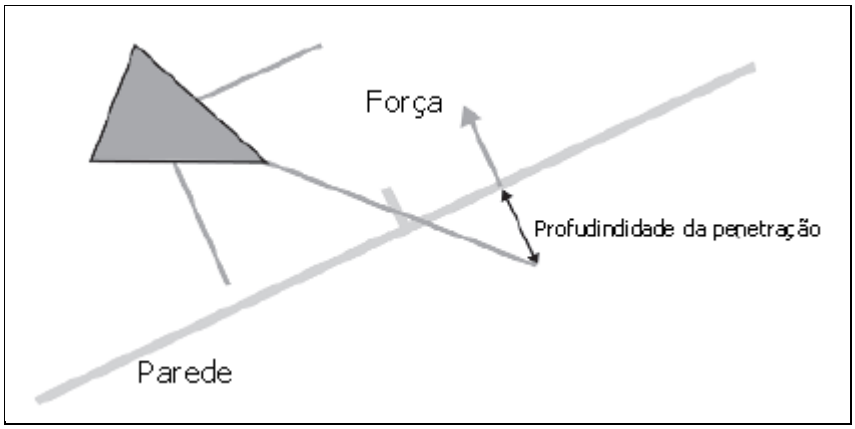


Fonte: Adaptado de Buckland (2004)

2.5.3.8 *Wall Avoidance* (Evasão de paredes)

Evasão de paredes é um comportamento que direciona o personagem para evitar a colisão com alguma parede. Essa técnica pode utilizar três referências, na forma de vetores. Quando um desses vetores encosta em uma parede, cálculos podem determinar a direção para levar o personagem para longe da parede, conforme figura 20.

Figura 20 - Sensores de evasão de paredes



Fonte: Adaptado de Buckland (2004)

A magnitude do vetor de desvio é proporcional a quanto um dos vetores penetrou na parede.

2.5.4 Algoritmo Colônia de Formigas.

O Algoritmo Colônia de Formigas é inspirado no comportamento de formigas reais (MULATI, 2013). Esse algoritmo já foi utilizado em jogos de computador (MOCHOLI et al., 2010, EMILIO et al., 2010).

O pesquisador Marco Dorigo observou que as formigas tendem a seguir o caminho mais curto entre uma fonte de alimento e a colônia. Isso acontece graças a uma substância chamada feromônio, que permite uma comunicação indireta entre as formigas. Quando uma formiga encontra alimento ela retorna para a colônia e no caminho, libera feromônio, criando uma trilha. Quando outras formigas entram em contato com essa substância, elas tendem a seguir a trilha, liberando ainda mais feromônio, fortalecendo-a.

O feromônio é uma substância volátil e, portanto, acontece uma evaporação natural. A evaporação do feromônio permite que trilhas mais curtas e, portanto, mais eficazes, prevaleçam sobre as trilhas mais longas, visto que mais formigas passarão por esse caminho em relação à caminhos mais demorados até a fonte de alimento. Esse processo resulta em um comportamento auto catalítico[13], processo onde ocorre uma reação sem a necessidade de um catalisador, ou seja, o fato da formiga seguir a trilha e depositar feromônio é o que faz com que outras formigas sigam a trilha depositando mais feromônio, reforçando a trilha e o ciclo se repete.

2.5.4.1 Os Experimentos Das Pontes Duplas.

Como explicado por Dorigo e Stützle (DORIGO; STÜTZLE, 2004), foram realizados diversos experimentos para estudar o comportamento das formigas de depositar feromônio e seguir a trilha que se formava. Uma série de experimentos em específico realizado por Deneubourg et al. chama a atenção. Esse experimento utilizou formigas da espécie I. humilis que deveriam atravessar uma ponte para alcançar uma fonte de comida.

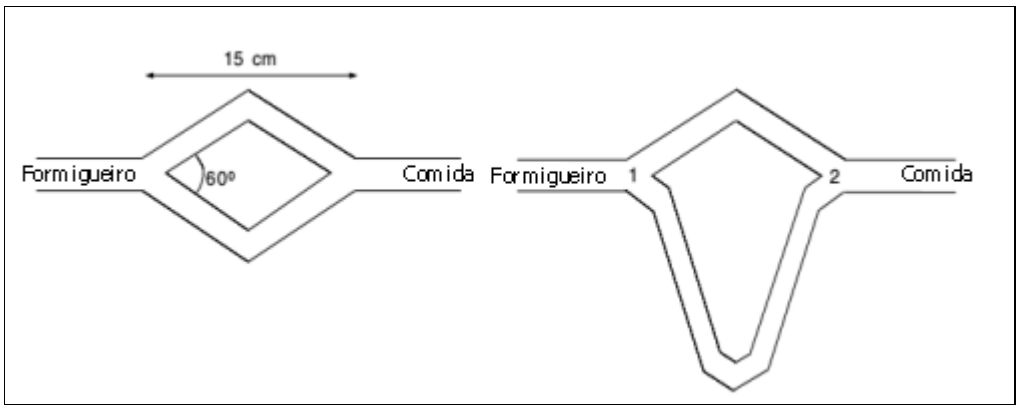
[13] Acesso em: 21 de setembro de 2018, disponível em: <https://www.dicionarioinformal.com.br/diferenca-entre/autocat%C3%A1lise/rea%C3%A7%C3%A3o/>

O primeiro desses experimentos foi realizado utilizando uma ponte com dois caminhos de mesma distância para uma fonte de comida. Inicialmente as formigas escolhem um caminho aleatoriamente, entretanto com o passar do tempo elas tendem a escolher apenas um caminho. Isso acontece pois quando o teste inicia não há feromônio em nenhum dos caminhos, logo as formigas escolhem de maneira aleatória. Entretanto um dos caminhos será escolhido por um número maior de formigas aleatoriamente. Como as formigas depositam feromônio no caminho até a comida e no caminho de volta até o formigueiro, esse caminho conterá uma trilha de feromônio mais forte, fazendo com que mais formigas sigam esse caminho, fortalecendo ainda mais a trilha até que todas as formigas sigam apenas esse caminho.

O segundo experimento utiliza um caminho curto e um caminho com o dobro do tamanho do primeiro. No início as formigas escolherão aleatoriamente entre os dois caminhos. Pode-se assumir que metade das formigas escolham o caminho curto e metade escolham o caminho longo. As formigas que escolheram o caminho curto chegarão antes na fonte de comida e ao ter de escolher o caminho de volta tenderão a escolher o caminho mais curto por este já ter uma trilha de feromônio. Quando as formigas que escolheram o caminho longo chegarem na fonta de comida, no caminho de volta também tenderão a escolher o caminho curto. Dessa forma, após algum tempo todas as formigas escolherão o caminho curto.

A figura 21 mostra como foram os experimentos das pontes duplas. Na imagem da esquerda, dois caminhos de mesma distância separam o formigueiro da fonte de comida. Na imagem da direita há um caminho com o dobro do tamanho do outro.

Figura 21 - Experimentos das pontes duplas



Fonte: Adaptado de Dorigo; Stützle (2004)

Essa observação se tornou a inspiração para o Algoritmo Colônia de Formigas.

3 DESENVOLVIMENTO

As próximas seções abordarão o desenvolvimento do trabalho.

3.1 ANT HILL

Inicialmente planejava-se desenvolver uma Inteligência Artificial tendo como base um jogo simples, de código aberto, chamado Ant Hill. O jogo está disponível para download em https://cowthing.itch.io/ant-hill.

Ant Hill foi desenvolvido por um desenvolvedor ou equipe de desenvolvimento identificado como Cowthing, utilizando o motor gráfico Godot, publicado em 24 de abril de 2017.

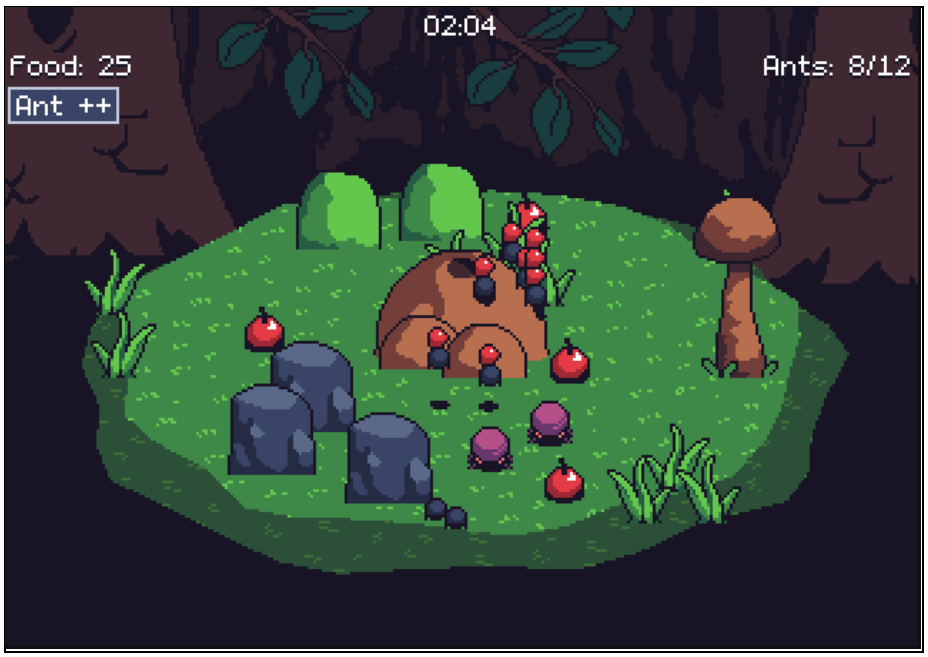
O objetivo do jogo é bastante simples, o jogador deve sobreviver o máximo de tempo possível controlando uma pequena colônia de formigas. O jogador deve colher comida e, com esta, criar mais formigas. A cada 10 segundos algumas frutas aparecem pelo cenário e o jogador deve coletá-las. Entretanto também há inimigos que tentarão matar as formigas do jogador

O jogo funciona através do clique do mouse. Ao clicar, o jogador envia uma mensagem para todas as formigas irem até aquele local. Ao entrar em contato com a comida, a formiga pega a comida e leva até o formigueiro. Cada comida tem uma quantidade de pontos de vida limitada, valor que representa quantas vezes um determinado personagem pode receber dano antes de morrer ou sumir, muito comum nos jogos eletrônicos, e cada formiga que entra em contato com ela remove 1 ponto de vida. As formigas e os inimigos também possuem uma pequena quantia de vida, portanto é possível vencer os inimigos ao atacá-los, fazendo com que a formiga entre em contato com o inimigo, algumas vezes. O jogador pode utilizar dois pontos de comida para criar uma nova formiga.

O jogo acaba caso todas as formigas do jogador sejam derrotadas e o jogador não possua mais pontos para criar outras formigas.

A figura 22 mostra a tela do jogo Ant Hill, desenvolvido por Cowthing. Na figura pode-se ver as formigas, pequenas com coloração escura, fugindo dos inimigos maiores, de coloração avermelhada e carregando as maças para o formigueiro.

Figura 22 - Jogo Ant Hill



Fonte: Ant Hill (2018)

A partir desse cenário analisou-se possibilidades para aplicar técnicas de IA. Isso levou ao estudo do algoritmo de colônia de formigas.

No original, o jogador controla as formigas que lutam com inimigos. Para uma versão modificada do *software*, pensou-se em um controle automático para as formigas e colocar o próprio jogador na posição de inimigo, tentando evitar o avanço das formigas.

Após algumas análises, chegou-se a conclusão que a implantação das técnicas de IA nesse jogo necessitaria de muitos ajustes na física do jogo, visto que a movimentação dos personagens não é precisa, os personagens “deslizam” pelo cenário. Para contornar esse problema decidiu-se realizar a implantação das técnicas em um jogo mais simples, com movimentação em *grid*, uma matriz, pois os esforços para alteração na física do Ant Hill somariam mais tempo do que a implantação da IA.

3.2 ANT GAME - NOVA VERSÃO DO JOGO

Após algumas buscas, encontrou-se um código demonstrativo de movimentação em *grid* no Godot, desenvolvido por Nathan Lovato, um criador de

conteúdos online que grava vídeos para a plataforma Youtube sobre desenvolvimento de jogos em seu canal, GDquest[14].

O código demonstrativo está disponível para *download* em https://github.com/GDquest/Godot-engine-tutorial-demos/tree/master/2018/06-09-grid-based-movement.

Esse código é apenas um exemplo que mostra o funcionamento dos *grids* no motor Godot. Essa versão foi escolhida pois o funcionamento do movimento em *grid* se encaixaria melhor com a implementação dos algoritmos escolhidos: Máquina de estados finitos e Algoritmo Colônia de Formiga.

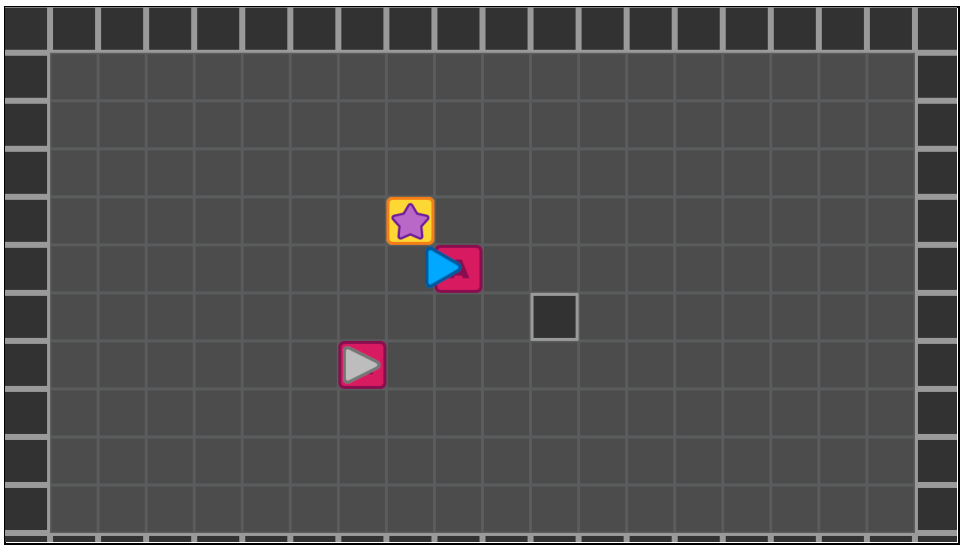
Nessa versão o jogador pode se mover pelo *grid* utilizando as setas do *mouse* ou as teclas W, A, S e D. O personagem, representado por uma seta azul, consegue se mover na vertical, horizontal e na diagonal pelo *grid*. Ao tocar em um obstáculo, é tocado a animação de colisão, impedindo o jogador de entrar dentro das paredes. A mesma animação é tocada ao encostar em outro peão no cenário. Ao passar por cima de um objeto que tem o formato de uma estrela, o objeto desaparece.

Como se trata de um código demonstrativo, não há objetivos.

A figura 23 mostra o funcionamento do código demonstrativo do *grid*. O quadrado vermelho presente embaixo dos atores (setas) e o quadrado amarelo embaixo da estrela roxa representam a posição dos objetos referentes ao *grid*. As setas e a estrela são objetos, *nodes*, instanciados no cenário.

[14] GDQuest. Acesso em: 01 de dezembro de 2018, disponível em: <https://www.youtube.com/watch?v=9laHKHYNyXc>

Figura 23 - Demonstração de *Grid*



Fonte: Autoria Própria

Para o desenvolvimento desse trabalho, foram implementadas as seguintes técnicas de IA: Máquina de Estados Finitos e Algoritmo Colônia de Formiga.

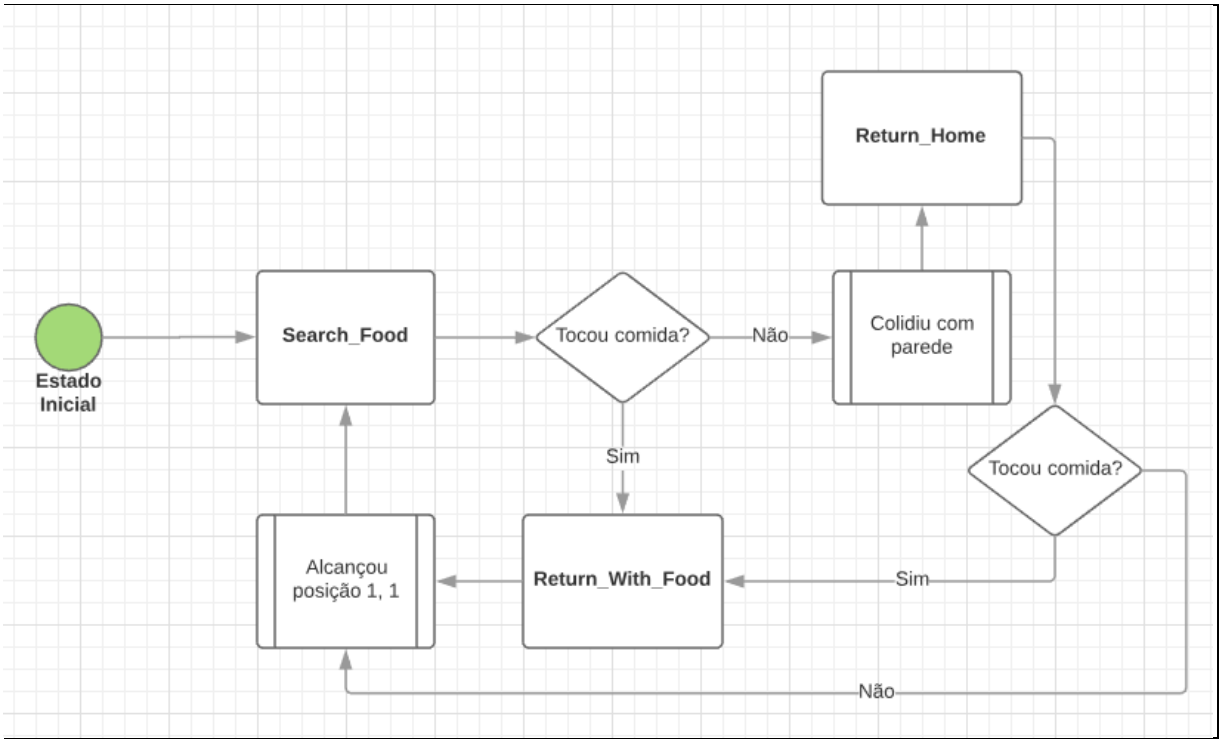
Os personagens formigas serão controlados por uma máquina de estados simples que, em certos momentos, ativa o algoritmo de colônia de formigas.

A máquina de estados apresentada na figura 24 funciona da seguinte maneira:

- As formigas terão três estados e iniciarão na primeira posição do *grid* (1, 1);
- As formigas poderão andar em um total de 6 direções, como representado na figura 25;
- Como se trata de um *grid*, a posição à direita da formiga, chamaremos de posição 1, a posição à direita e abaixo da formiga, posição 2, a posição abaixo da formiga, posição 3. A formiga poderá se mover para essas direções enquanto estiver no estado Search_Food (Procurar comida, em português);
- Já nos estados Return_Home (Retornar para o formigueiro, em português) e Return_With_Food (Retornar com comida, em português), a formiga poderá se mover para esquerda, o que chamaremos de posição 4, à esquerda e acima da formiga, posição 5, e para cima, posição 6;
- Quando a formiga tocar uma das paredes, isso é, qualquer posição que seja 10 no eixo x ou eixo y, ela ativará o estado Return_Home;
- Ao tocar a posição 1, 1 no *grid*, a formiga entrará no estado Search_Food novamente;

- Ao tocar em uma comida, a formiga entrará no estado Return_With_Food, estando no estado Search_Food ou Return_Home;
- No estado Return_With_Food, a formiga terá o comportamento parecido com o estado Return_Home, entretanto a formiga depositará feromônio no caminho;
- Em todos os estados a formiga considerará a quantia de feromônio nos nódulos possíveis, determinados pelo estado atual, para decidir em qual nódulo ela se moverá;

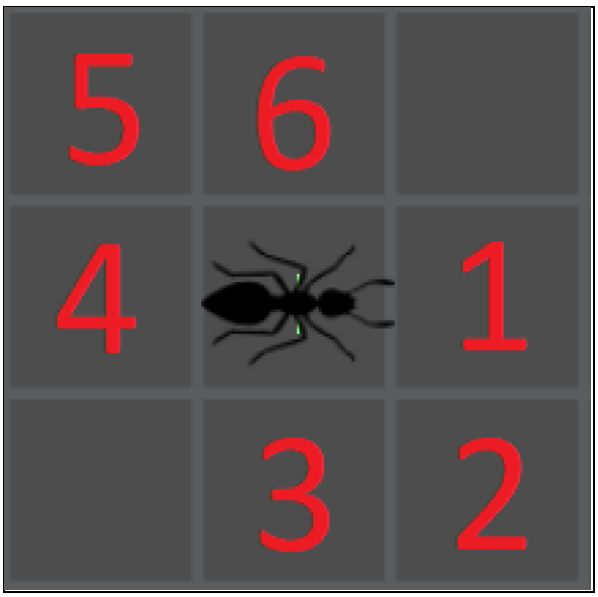
Figura 24 - Representação da máquina de estados finitos



Fonte: Autoria própria

A figura 25 mostra as seis possíveis direções que a formiga pode seguir. A limitação dos movimentos da formiga simplificaram o desenvolvimento das técnicas de IA usadas e ajudaram a não alongar demasiadamente o cronograma do projeto.

Figura 25 - Possíveis posições de movimento



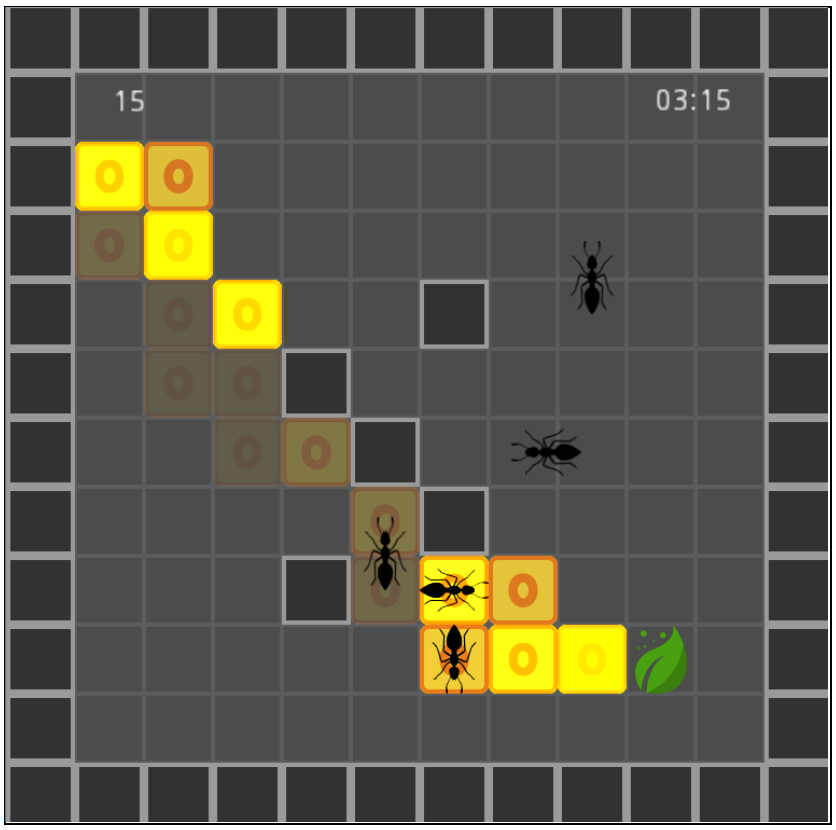
Fonte: Autoria Própria

A interação com o jogador se dará através do clique do *mouse*. O jogador terá de acumular pontos, que são ganhos com o passar do tempo, para então gastar esses pontos para colocar um objeto no caminho das formigas.

O objetivo é manter as formigas longe da comida, colocando pedras no meio do caminho para atrapalhá-las. Quando as formigas tocarem na comida um determinado número de vezes, o jogo acabará e a pontuação será mostrada. Quanto mais tempo o jogador conseguir manter as formigas longe da comida, mais pontos ele conseguirá.

Como mostrado na figura 26, na tela haverá uma matriz com 100 posições contendo 10 posições no eixo x e 10 posições no eixo y. Os pontos serão mostrados no canto superior esquerdo e o tempo total de jogo no canto superior esquerdo. O jogador poderá gastar alguns pontos acumulados para colocar um obstáculo no caminho das formigas para atrasá-las. O comportamento esperado é que as formigas contornem o objeto e voltem a criar uma trilha de feromônio até a comida.

Figura 26 - Tela do jogo



Fonte: Autoria Própria

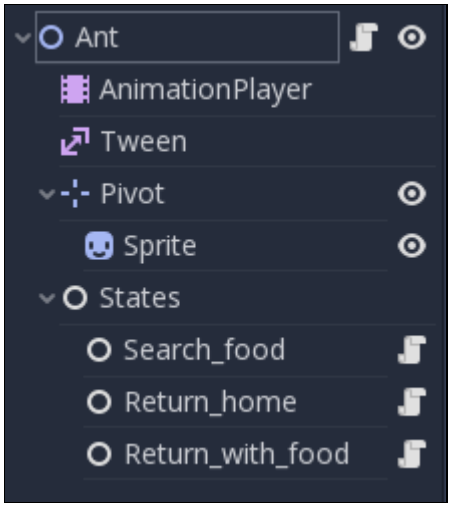
3.3 FUNCIONAMENTO DOS ALGORITMOS

As próximas seções abordarão o funcionamento dos algoritmos implantados, máquina de estados finitos e algoritmo colônia de formigas.

3.3.1 Máquina De Estados Finitos

Cada formiga possui uma máquina de estados finitos que possui três possíveis estados, como mostrado na figura 27.

Figura 27 - *Scene* da Formiga contendo os estados



Fonte: Autoria Própria

O nódulo States é usado para agrupar os três possíveis estados: Search_food, Return_home e Return_with_food. A máquina de estados que controla a interação dos estados é implementado como um script pertencente à formiga.

A cada ciclo do jogo é chamado o método update de um dos estados. O estado inicial da formiga é Search_Food.

No estado Search_Food a formiga busca uma fonte de comida, como explicado na seção 3.2 Nova Versão Do Jogo.

A formiga mantém em um dicionário, uma espécie de matriz com um identificador único, as posições do *grid* na direção 1, 2 e 3, o tipo de objeto que está naquela posição, que pode estar vazio, conter uma pedra, comida ou feromônio, a

quantia feromônio presente naquela posição e a porcentagem de chance da formiga ir para aquele local que é calculado usando a quantia de feromônio presente no local.

A formiga sai do estado Search_Food caso alguma das posições próximas a ela contenha comida.

Na linha 35 (figura 28) é verificado se alguma das posições arredores da formiga contém comida, caso sim, então o estado atual da formiga muda para Return_With_Food.

Na linha 38 é verificado se a posição x ou y da formiga é igual a 10, caso sim, então o estado atual da formiga muda para Return_Home.

Figura 28 - Trecho de código do estado search_food

```
29 func update(host, delta, grid):
30     timer += delta
31     if timer >= timer_timeout:
32         timer = timer_timeout
33         host.update_surround()
34         update_movable_pos(host)
35         if search_pos.pos1[1] == 3 or search_pos.pos2[1] == 3 or search_pos.pos3[1] == 3:
36             return "return_with_food"
37         var pawn_pos = grid.get_pawn_position(host)
38         if pawn_pos.x == 10 or pawn_pos.y == 10:
39             return "return_home"
40         var input_direction =  calc_direction(host)
41         if not input_direction:
42             return
43         host.update_look_direction(input_direction)
```

Fonte: Autoria própria

Na linha 36 (figura 29) é verificado se a formiga está na primeira posição do *grid*, caso sim, então o estado muda para Search_Food.

Na linha 39 é verificado se alguma das posições arredores da formiga contém comida, caso sim, então o estado muda para Return_With_Food.

No estado Return_With_Food a formiga volta para o estado Search_Food quando alcança a primeira posição, como no estado Return_Home.

Figura 29 - Trecho de código do estado return_home

```
func update(host, delta, grid):
    timer += delta
    if timer >= timer_timeout:
        timer = timer_timeout
        host.update_surround()
        pawn_pos = grid.get_pawn_position(host)
        if pawn_pos.x == 1 and pawn_pos.y == 1:
            timer = 0
            return "search_food"
        if search_pos.pos1[1] == 3 or search_pos.pos2[1] == 3 or search_pos.pos3[1] == 3:
            return "return_with_food"
        var input_direction =  calc_direction(host)
        if not input_direction:
            timer = 0
            return
        host.update_look_direction(input_direction)
```

Fonte: Autoria Própria

3.3.2 Algoritmo Colônia De Formigas

O comportamento do algoritmo colônia de formigas está implantado em todos os estados. Em todos os estados as formigas tendem a seguir o caminho com maior concentração de feromônio e no estado return_with_food a formiga deposita feromônio no caminho de volta ao formigueiro.

Antes da formiga se mover para uma nova posição no estado return_with_food, ela deposita feromônio na sua posição, conforme mostrado no código presente na figura 30.

Figura 30 - Trecho de código do *grid*, responsável por criar feromônio

```
func spawn_pheromone(pawn_position): #Valor ja vem transformado pro grid
    var cell_target_type = get_cellv(pawn_position)
    match cell_target_type:
        EMPTY:
            var pheromone = pheromone_base.instance()
            add_child(pheromone)
            pheromone.connect("pheromone_evaporated", self, "clear_pheromone")
            pheromone.set_position(map_to_world(pawn_position) + cell_size / 2)
            set_cellv(pawn_position, PHEROMONE)
        PHEROMONE:
            increase_pheromone(pawn_position)
```

Fonte: Autoria Própria

O *grid* verifica se há algo na posição da formiga, caso seja um espaço vazio, então é criado uma instância do objeto feromônio, caso haja feromônio naquela posição, então é aumentado a quantidade de feromônio.

A figura 31 mostra o processo de evaporação do feromônio. A cada ciclo do temporizador, que é definido como uma variável do objeto, é diminuído o valor da vida do feromônio. Quando a vida do feromônio chega a zero, então ele é evaporado e desaparece.

Figura 31 - Trecho de código do feromônio

```
27 func _process(delta):
28     timer += delta
29     if timer >= timer_timeout:
30         timer = timer_timeout
31         pheromone_health -= pheromone_evap_rate
32         modulate.a = pheromone_health / 100.0
33         pheromone_qty = pheromone_health / taxa_pheromone
34 
35         timer = 0
36 
37     if pheromone_health <= 0:
38         evaporate()
```

Fonte: Autoria Própria

A figura 32 mostra o trecho de código que realiza o cálculo que é realizado pela formiga para escolher uma rota.

Figura 32 - Cálculo para escolha de rota

```
47 func calc_direction(host):
48     randomize()
49     search_pos.pos1[3] = search_pos.pos1[2] / (search_pos.pos1[2] + search_pos.pos2[2] + search_pos.pos3[2])
50     search_pos.pos2[3] = search_pos.pos2[2] / (search_pos.pos1[2] + search_pos.pos2[2] + search_pos.pos3[2])
51     search_pos.pos3[3] = search_pos.pos3[2] / (search_pos.pos1[2] + search_pos.pos2[2] + search_pos.pos3[2])
52     randomize()
```

Fonte: Autoria Própria

A formiga pode se mover para três posições diferentes em cada estado e cada uma dessas posições possui uma certa quantia de feromônio que por padrão é 0.1, portanto se há três posições livres o cálculo ficaria 0.1 / 0.3 para cada uma das posições representando 33% de chance de ir para qualquer local.

O feromônio possui duas variáveis importantes para o cálculo que são a vida e a taxa de feromônio. A vida dividida pela taxa de feromônio representa a quantidade

de feromônio em determinada posição do *grid*, portanto, quanto menor for o valor da taxa de feromônio, maior será a concentração de feromônio.

Por padrão a vida do feromônio inicia com 20 e a taxa de feromônio em 100, portanto quando uma formiga deposita um feromônio a taxa de feromônio do local passa de 0.1 para 0.2 e o cálculo seria 0.2 / 0.4, logo a formiga teria 50% de chance de seguir uma trilha que possua um feromônio com 20 de vida.

A cada segundo o feromônio perde 2 pontos de vida e portanto a chance da formiga seguir essa direção diminui conforme o tempo.

3.4 CALIBRAÇÃO

Como exemplificado na seção 3.3.2 Algoritmo Colônia de Formigas, o algoritmo colônia de formigas precisa de alguns parâmetros, como quantidade de feromônio depositada, taxa de evaporação e quantidade de feromônio por nódulo, obtido pelo cálculo da vida e a taxa de feromônio, responsável pela representação visual e pelo cálculo realizado pela formiga para decidir qual rota seguir.

A modificação dessas variáveis e alteração de algumas regras podem impactar fortemente nos resultados obtidos.

Foram realizados testes onde as formigas depositam feromônio apenas quando estão voltando com comida e testes onde a formiga deposita feromônio enquanto está no estado Search_Food. Percebeu-se que, enquanto em um cenário aberto, sem um caminho pré-definido, na maioria dos casos é favorável a técnica de depositar feromônio apenas no caminho de volta para evitar que as formigas se percam e formem um caminho para um local onde não haja comida.

Entretanto, ao criar um cenário parecido com o experimento das pontes duplas, com duas rotas pré-definidas onde uma é mais curta que a outra, as formigas se saem melhor depositando feromônio no caminho e na volta, pois quando as formigas que estão no caminho curto chegam na comida, elas já possuem um caminho favorável para retornar. Entretanto, pelo fato de utilizar 5 formigas ao mesmo tempo, poderia ocorrer de 3 ou 4 formigas irem pelo caminho longo, o que faria a trilha de feromônio predominar na rota longa.

Para prevenir isso, fora criada uma nova condição para as formigas, onde elas depositariam menos feromônio a medida que andassem, ou seja, a formiga que está

na rota curta chegaria na comida ainda depositando uma quantia considerável de feromônio, entretanto as formigas que seguissem pela rota longa estariam depositando uma quantia muito baixa de feromônio. Dessa forma, o caminho mais curto seria mais tentador para uma formiga que estivesse retornando. Para realizar esse teste a taxa de evaporação do feromônio deveria aumentar, para diminuir a quantia de feromônio no início da rota. Porém, por mais que essa solução se mostrasse superior ao teste anterior, ainda não possuía um resultado satisfatório pois caso 3 ou 4 formigas seguissem a rota longa, depositariam uma quantia muito alta de feromônio no início da rota longa, fazendo com que a evaporação elevada não fosse suficiente para diminuir a quantia de feromônio na rota longa e tornar a rota curta mais atrativa.

A figura 33 mostra uma formiga que já alcançou a fonte de comida e deve retornar para o formigueiro.

Figura 33 - Formiga escolhendo entre as rotas

Fonte: Autoria Própria

Nesse teste ela deposita feromônio no caminho de ida até a comida, por conta disso ela tende a escolher o caminho por qual ela veio por já possuir uma trilha de feromônio.

Quando a segunda formiga alcançar a fonte de comida ela terá de escolher entre o caminho que ela fez, que possui uma quantia menor de feromônio por ser mais

longo, ou pelo caminho mais curto que possui uma quantia maior de feromônio depositado pela primeira formiga no caminho de ida e volta.

Por se tratar de um algoritmo onde os valores das variáveis e a alteração de algumas regras podem mudar consideravelmente os resultados, são necessários diversos testes para se chegar em uma solução ótima.

4 CONCLUSÕES

Para desenvolver um jogo são necessárias diversas habilidades diferentes dentro de uma equipe, desde artistas de imagens e sons a roteiristas. O desenvolvimento de jogos difere-se do desenvolvimento de outros *softwares* mais comuns por ser focado em entreter milhares de jogadores ao invés de atender às necessidades corporativas.

A Inteligência Artificial é muito importante e explorada pelos jogos para torná-los mais dinâmicos, desafiadores e divertidos. IA pode ser usada de diferentes maneiras nos jogos, seja para pilotar personagens para tornar o jogo mais realista, controlar um time de futebol, pilotos de corrida ou para controlar o comportamento de algumas formigas.

Uma das deficiências da ferramenta Godot, utilizado na elaboração desse trabalho, é a falta de documentação, o que dificulta a procura por alguns assuntos específicos, como o funcionamento do *grid*. Uma quantidade considerável de tempo foi aplicado no entendimento desse componente e na interação dele com os objetos do jogo.

Alguns problemas foram identificados na adaptação do jogo Ant Hill que necessitaria de um grande esforço para a modificação de sua física. As formigas e os inimigos não possuem movimentação precisa, o que dificultaria a implantação e testes dos algoritmos. A solução para isso foi encontrar um jogo base que funcionasse com movimentação em uma matriz, mais precisa, para implantação das duas técnicas escolhidas.

A aplicação do Algoritmo Colônia de Formigas, sem tratamento, traria problemas com *loops* na movimentação das formigas e, por conta disso, a movimentação das formigas foi limitada para apenas três possíveis movimentos por estado. Outra solução seria colocar memória nas formigas, fazendo com que não seguissem um caminho que entrasse em *loop*.

Para alcançar um comportamento satisfatório da IA, diversas variáveis são modificadas até encontrar o comportamento desejado. Ao realizar alguns testes replicando o experimento das pontes duplas, percebeu-se que ao permitir as formigas liberarem feromônio no caminho de ida e de volta da comida ao formigueiro, obteve-se um comportamento mais satisfatório, seguindo com maior frequência o caminho

curto. Entretanto para obter-se esse resultado também fez-se necessário fazer com que as formigas depositassem menos feromônio a medida que andassem para evitar que o caminho longo se fortalecesse mais que o curto por conta da probabilidade de mais formigas seguirem inicialmente o caminho longo.

Para futuras versões deste jogo desenvolvido neste trabalho podem ser inseridos novas habilidades para o jogador, como poder limpar toda a trilha de feromônio por um custo mais alto nos pontos. Mais testes podem ser realizados com a matriz aberta e com a matriz replicando o experimento das pontes duplas, a fim de obter um comportamento mais satisfatório. Também poderão ser incluídos novos estados para a formiga para que ela possa tomar diferentes decisões perante algumas situações como por exemplo poder andar em uma direção oposta à permitida pelo estado atual, como forma de desviar de um obstáculo.

Com a inserção de inimigos o comportamento das formigas teria de ser adaptado. Poderia haver a possibilidade de ajuste da dificuldade, o que mexeria com os parâmetros do feromônio, diminuindo a quantidade depositada por formiga ou então diminuindo a influência do feromônio na decisão da formiga em escolher uma rota. Também poderia ser aumentado o tamanho do *grid* que atualmente possui 100 posições, em 10x10. Com a inserção de novas fontes de comida espalhadas poderia-se verificar o comportamento das formigas assim como aumentando o desafio imposto ao jogador, visto que o mesmo terá de bloquear possíveis rotas para outras fontes de comida.

REFERÊNCIAS

Ant Hill, **Ant Hill**. Disponível em: <https://cowthing.itch.io/ant-hill>. Acesso em: 03 dez. 2018.

BUCKLAND, Mat. **Programming game ai by example**. Sudbury: Wordware, 2004. 495 p. ISBN 9781556220784.

CLUA, Esteban Walter Gonzalez; BITTENCOURT, João Ricardo. **Desenvolvimento de jogos 3D: Concepção, Design e Programação**. Disponível em: <http://www.ic.uff.br/~esteban/files/Desenvolvimento%20de%20jogos%203D.pdf> Acesso em: 05 out. 18.

DORIGO, Marco; STÜTZLE, Thomas. **Ant Colony Optimization**. Londres: The MIT Press, 2004.

MARTIN, Emilio; MARTINEZ, Moises; RECIO, Gustavo; SAEZ, Yago. Pac-mAnt: Optimization based on ant colonies applied to developing an agent for Ms. Pac-Man. In **Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games** 2010 Aug 18 (pp. 458-464). IEEE.

Godot Engine. Disponível em: <http://docs.godotengine.org/en/stable/>. Acesso em: 15 Junho 2017.

Godot vs Unreal Engine 4. Disponível em: <https://www.slant.co/versus/1068/5128/~godot_vs_unreal-engine-4>. Acesso em: 25 Junho 2017.

GREGORY, Jason. **Game Engine Architecture**. Wellesley: A K Peters/CRC Press, 2009. E-book. ISBN 13: 978-1-4398-6526-2. Disponível em: <http://www.latexstudio.net/wp-content/uploads/2014/12/Game_Engine_Architecture-en.pdf> Acesso em: 25 Junho 2017.

Interface Godot, **Godot aims for mainstream**. Disponível em: <https://godotengine.org/article/godot-aims-mainstream>. Acesso em: 03 dez. 2018.

Interface Unity, **Unity Editor on Linux**. Disponível em: <https://blogs.unity3d.com/pt/2015/07/01/the-state-of-unity-on-linux/linux-editor-screenshot2/>. Acesso em: 03 dez. 2018.

Interface Unreal, **Tools and Editors**. Disponível em: <https://docs.unrealengine.com/en-US/GettingStarted/SubEditors>. Acesso em: 03 dez. 2018.

JÖRMMARK, Jan; AXELSSON, Ann-Sofie; ERNKVIST, Mirko. Wherever hardware, there'll be games: The evolution of hardware and shifting industrial leadership in the gaming industry. In **Proceedings of the Digital Games Research Association International Conference (DiGRA)** 2005, Vancouver, Canada 2005.

KENT, Steven L. **The Ultimate History of Video Games: From Pong to Pokémon and Beyond- The Story Behind the Craze That Touched Our Lives and Changed the World**. Nova Iorque: Three Rivers Press, 2001.

KISHIMOTO, André. **Inteligência Artifical em Jogos Eletrônicos**. 2004. Disponível em: <http://www.programadoresdejogos.com/trab_academicos/vandre_kishimoto.pdf>. Acesso em: 18 de Set. 2018.

MOCHOLI, Jose A. et al. An emotionally biased ant colony algorithm for pathfinding in games. **Expert Systems with Applications**, v. 37, n. 7, p. 4921-4927, 2010.

MORAIS, Felipe Castanheira et SILVA. Desenvolvimento de jogos eletrônicos. **e-xacta**, 2009, vol. 2, no 2.

NICKOLLS, John, DALLY, William. The GPU computing era. **IEEE micro**. 2010 Mar;30(2).

Nós e Cenas no Godot. **Scenes and Nodes**. Disponível em: <http://docs.godotengine.org/en/stable/learning/step_by_step/scenes_and_nodes.html>. Acesso em: 03 dez. 2018.

Pacman. **O maior ícone dos jogos eletrônicos**. Disponível em: <http://www.museudocomputador.org.br/historia-pacman>. Acesso em: 03 dez. 2018.

PARR, Ronald; RUSSELL, Stuart. Reinforcement learning with hierarchies of machines. In **Advances in neural information processing systems.** 1998 (pp. 1043-1049).

PONG, **Conheça Pong, o primeiro videogame lucrativo da história**. Disponível em: <https://www.techtudo.com.br/noticias/noticia/2016/03/conheca-pong-o-primeiro-videogame-lucrativo-da-historia.html>. Acesso em: 03 dez. 2018.

RUSSELL, Stuart; NORVIG, Peter. **Inteligência Artificial: Uma abordagem moderna**. 3. ed. Rio de Janeiro: Elsevier, 2013.

SPACEWAR, **Spacewar!: A Spin Around the Sun**. Disponível em: <http://thedoteaters.com/?cat=741>. Acesso em: 03 dez. 2018.