Inteligência Artificial para Jogos: Trabalho Prático 1 - Máquina de Estados Finitos

Eduardo Rodrigues da Silva[1], Everton Perez Vaszelewski[2], Giovana Goulart Jardim[2], Tailine Wornath[3]

[1]International Universities Laureate (UniRitter) – Campus Iguatemi
Av. João Wallig, 1800 – Passo d'Areia, Porto Alegre, RS – 91340-000 – Brazil

[2]International Universities Laureate (UniRitter) – Campus FAPA
Av. Manoel Elias, 2001 – Passo das Pedras, Porto Alegre, RS – 91240-261 – Brazil

[3]International Universities Laureate (UniRitter) – Campus Canoas
Rua Santos Dumont, 888 – Niterói, Canoas, RS – 92120-110 – Brazil

{erodrigues680, everton.vaszelewski, giovanagoulart70, taywornath}@gmail.com

Abstract. *This article presents the different methods of applying artificial intelligence in games, bringing the specific example and illustration of the Finite State Machine (FSM) algorithm, which is used until today by games recognized worldwide. It also contemplates the development of a game prototype, applying the Hunt x Hunter problem, with the purpose of exercising the modeling and implementation of finite state machines, and was used in the C ++ programming language for development.*

Resumo. *Este artigo apresenta os diferentes métodos de aplicação de inteligência artificial em jogos, trazendo a exemplificação e ilustração específica do algoritmo Finite State Machine (FSM), a qual é utilizado até hoje por jogos reconhecidos mundialmente. Também contempla o desenvolvimento de um protótipo de jogo, aplicando o problema de Caça x Caçador, com o propósito de exercitar a modelagem e a implementação de máquina de estados finitos, e foi utilizada a linguagem de programação C++ para o desenvolvimento.*

1. Algoritmos de Inteligência Artificial para Jogos

A primeira inteligência artificial aplicada em jogos pôde ser vista em 1979, no jogo PacMan, onde através de Controlling Non-Player Characters (NPCs), que nada mais é do que um personagem não jogável, foi utilizado o algoritmo Finite State Machine (FSM), também conhecido como máquina de estados. Neste processo, foram identificados os três diferentes comportamentos que poderiam acontecer durante o jogo, e assim, foram programadas as ações que seriam executadas automaticamente pela máquina, trazendo assim, a ideia de que estes jogadores possuem uma inteligência própria, garantindo com que a experiência de usuário alcançasse um outro patamar na época.

Atualmente existem diferentes metodologias de inteligência artificial que são aplicadas em jogos, mas o algoritmo FMS ainda é o mais utilizado por aquelas empresas que desejam garantir que o usuário entenda os padrões comportamentais destes NPCs, além de permitir que o jogador possa vencer a máquina eventualmente. Do outro lado das opções finitas da implementação da Inteligência Artificial (IA), está o método realizado pelo algoritmo Monte Carlo Search Tree (MCST), o qual permite que a máquina aprenda a analisar e calcular a ação que seria a mais adequada, a cada jogada.

Através da Figura 2, apresentada abaixo, podemos visualizar a demonstração de uma análise realizada pelo algoritmo, através do formato de árvore. Neste cenário, são infinitas as possibilidades que podem ser geradas e que consequentemente, a cada resultado, será criado mais um status para ser estudado pela máquina e assim sucessivamente, até que com o passar do tempo, a máquina deixará o usuário sem opções e ganhará dele.

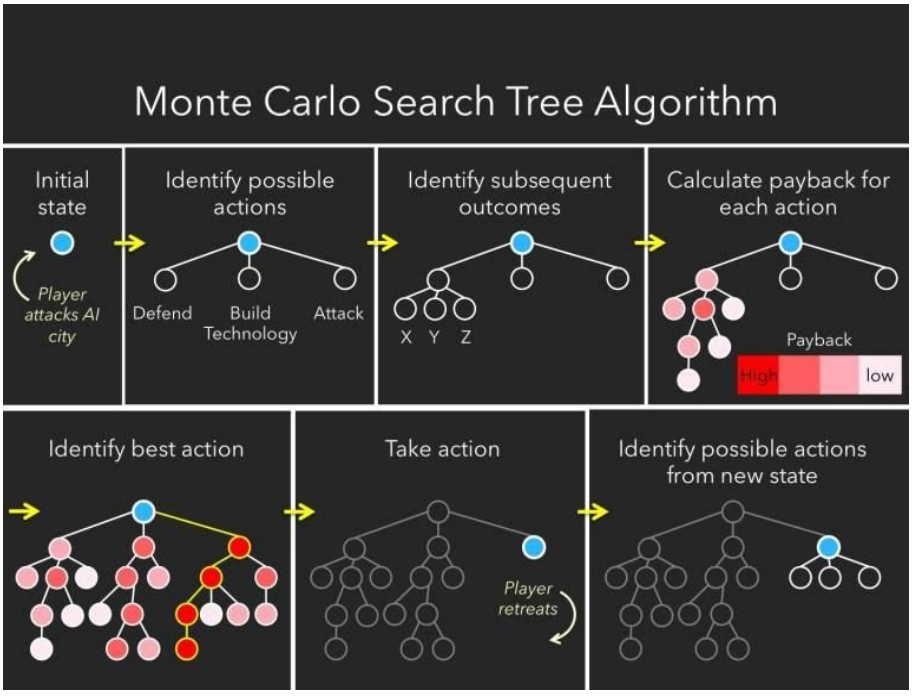

Figure 2. Demonstração simplificada do Monte Carlo Search Tree Algorithm.

Dentro dos métodos de aplicação da inteligência artificial em jogos, ainda pode-se citar o Real-Time Strategy (RTS), que consistem em uma estratégia de tempo real, onde a máquina acaba tirando proveito diante do jogador, pois ela é capaz de identificar e programar diversas tarefas ao mesmo tempo, enquanto que nós humanos, devemos fazer isso de forma manual. Este é um exemplo de técnica semelhante às duas anteriormente citadas, mas em que a capacidade da IA foi deliberadamente reduzida, com a finalidade de manter o interesse e melhorar a experiência do jogador.

Imagine um cenário de jogo onde você esteja tentando manter sua aldeia a salvo, prevendo e tomando ações contra ataques por parte da máquina. Ao mesmo tempo, a máquina consegue identificar todas as vulnerabilidades e padrões de defesa adotados e consequentemente, também saberá exatamente onde e quando atacar a aldeia. Neste formato, é possível que você irá perder rapidamente para a máquina e com isso, você também perderá o interesse pelo jogo, pois terá a impressão de que não existe nenhuma chance de derrotar a máquina. É através desta análise, que muitos jogos ainda hoje, optam por manter técnicas que fomentam o perfil de jogador ganhador, a fim de continuar com novas versões e consequentemente, manter-se no mercado.

2. Simulação do Jogo: Caça x Caçador

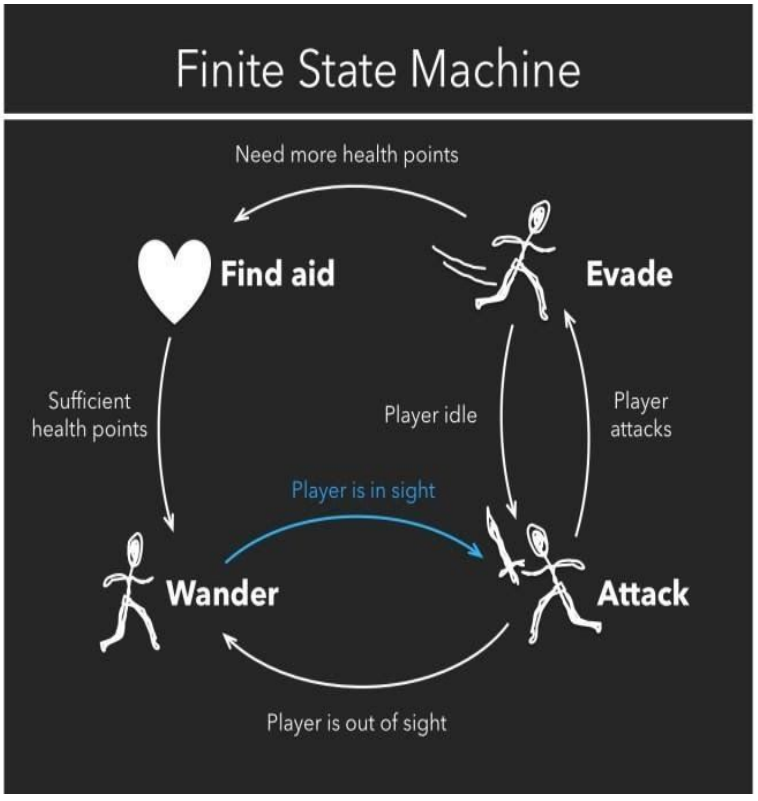

Figure 3. Método Finite State Machine.

Todo objeto de estudo sempre apresentará ao menos uma referência à sua origens e com os jogos, isso não é diferente. Tomando como base o exemplo utilizado na Figura 3, é possível identificar o comportamento adotado pelo método Finite State Machine, muito utilizado em jogos até hoje. Neste cenário, são analisados os estados de dois personagens: a caça e o caçador. Na imagem, é possível ainda identificar os seguintes status que são posteriormente implementados no jogo, que é Vaguear (perambular), Escapar e Atacar, e também ainda, poderá analisar dois possíveis estados adicionais, os quais poderão ser implementados, caso desejar.

A programação do algoritmo de uma forma geral, pode ser vista de acordo com seus respectivos status. No estado Vagar, quando uma caça for avistada, a máquina é programada para que o caçador realize o ataque. No estado de Atacar, se a caça for programada para revidar, o caçador poderá escapar ou então, se o caçador perder caça de vista e não representar mais perigo, ele poderá voltar ao estado de vagar. Já no estado de Escapar, são possíveis duas ações, o estado bônus que pode ser implementado, onde caso existir com risco de vida para o caçador, ele irá em busca de ajuda para se recuperar, enquanto que no outro status, se uma caça for avistada neste processo de fuga, o caçador poderá atacá-lo, visto que não representa um risco muito alto. Para finalizar o círculo desta representação, ainda é necessário analisar o último estado bônus que podem ser implementado, que é o Procurar ajuda, onde a partir do momento em que o caçador atingir os pontos de vida suficientes, ele poderá voltar a vagar e assim, seguir o fluxo conforme o novo status gerado.

2.1. O Desafio

Para realizar a simulação deste jogo, foi apresentado o propósito de exercitar a modelagem e a implementação de máquinas de estados finitos, desenvolvendo um protótipo de jogo, aplicando o problema de Caça x Caçador.

Para tal prática, algumas condições foram apresentadas e deveriam ser respeitadas para um resultado final satisfatório, as quais podem ser vistas a seguir.

A matriz da simulação deverá ser em uma matriz de células de tamanho 30 x 30. Cada célula poderá ter 3 status: vazia, preenchida por um Caçador ou preenchida por Caças. Logo na execução da simulação, deve ser gerado entre 5 a 10 caças e 1 caçador, todos posicionados aleatoriamente. O caçador não pode ocupar ao mesmo tempo uma célula que tenha uma caça. A cada turno, as caças e o caçador devem se movimentar aleatoriamente para uma das oito posições vizinhas. As caças devem preferir ocupar células mais distantes do caçador.

O caçador pode além de mover-se, pode girar. O caçador deve possuir um campo de visão de 5 células em todas as direções. Se houver uma caça no campo de visão, deve persegui-la, e se o caçador ficar em uma célula adjacente à caça, pode atacar. Uma vez que a caça é atacada, ela deve ser removida da matriz. Obviamente se não houver caça no campo de visão, deve ficar vagueando pela matriz até alguma aparecer. A simulação finaliza quando não houver mais caças na matriz. E ao final, deve exibir a quantidade de turnos necessários para capturar todas as caças.

2.2. Diagrama de Estados

Para determinar o comportamento dos personagens do jogo, comumente são utilizados os Diagramas de Estados. Através deles, é possível identificar todos os estados em que um jogador poderá passar durante a execução do jogo, e é a partir dele, que a análise de desenvolvimento do projeto é iniciada.

Podemos observar na Figura 4, apresentada abaixo, que o jogo é composto dos estados Vagando, Persegue a Caça e Ataca a Caça. Vejamos a seguir, o comportamento do personagem Caçador em cada um dos estados programados para o jogo e suas transições.

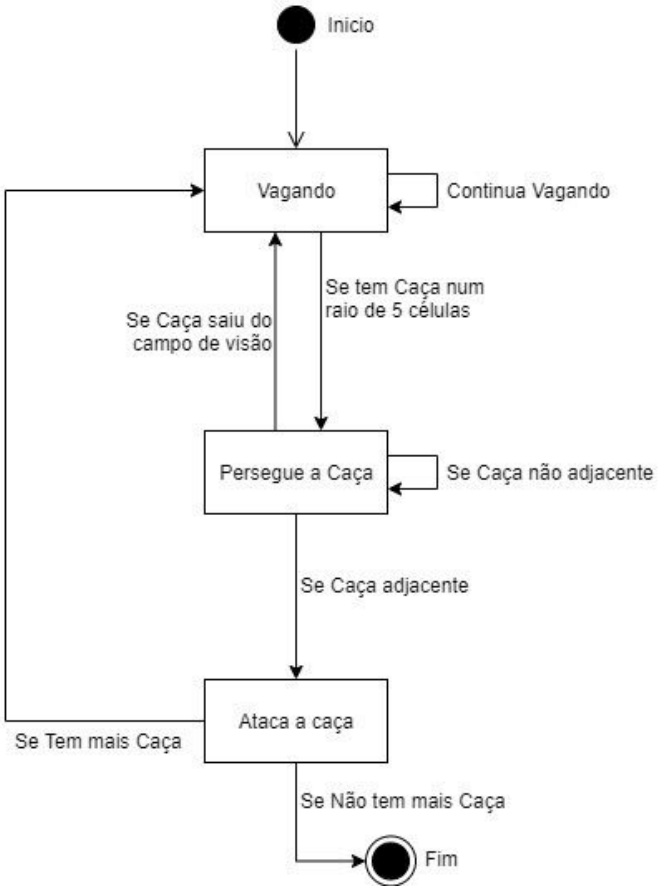

Figure 4. Diagrama de Estados.

No estado Vagando são possíveis duas ações: continuar vagando se não houver caça visível ou perseguir a caça no caso em que a caça estiver visível, sendo considerado um raio de 5 células da matriz ao seu redor.

No estado Persegue a Caça são possíveis três ações: se a caça não estiver adjacente à posição atual do caçador, o caçador não irá atacar e permanecerá perseguindo até que a caça seja adjacente, onde trocará para o estado de Atacar a Caça. Neste estado ainda, poderá ocorrer o caso da caça sair do campo de visão do caçador, onde então, o caçador retornará para o estado Vagando.

No estado Ataca a Caça, são possíveis duas ações: caso existir outra caça viva no jogo, ele não será finalizado e o caçador voltará para o status de Vagando. Quando todas as caças forem encontradas e eliminadas, neste momento então, a simulação do jogo será finalizada.

2.3. Máquina de Estados Finitos

Outra forma de representação muito utilizada para jogos é a máquina de estados finitos. Através dela, delimitam-se todos e somente os estados possíveis durante a execução do jogo.

Podemos observar na Figura 5 apresentada abaixo, que a Máquina de Estados Finitos se assemelha muito com o Diagrama de Estados. O estado inicial é Vagando e o estado final é Ataca a Caça, pois no momento que houver somente uma caça na simulação, o último ato do Caçador será atacar a caça.

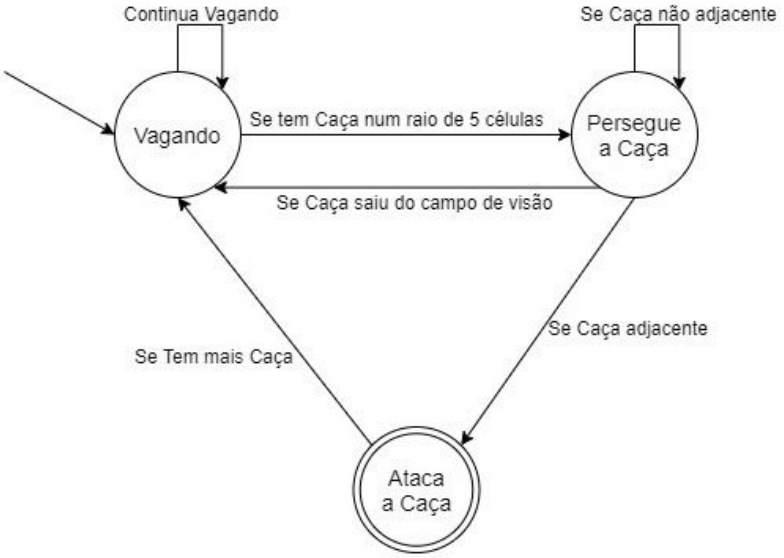

Figure 5. Máquina de Estados Finitos.

2.4. Implementação

A simulação do jogo foi desenvolvida utilizando a linguagem de programação C++, a qual permite utilização da linguagem imperativa, orientada a objetos e genérica.

Foram criadas duas STRUCTs, sendo uma para a caça, contendo sua posição e um booleano para indicá-la como viva e a outra struct para o caçador, com sua posição e indicando seu alcance de visão. O alcance foi definido com um raio de 5 células em sua volta, para que assim, o caçador possa identificar e perseguir a caça quando alguma delas for avistada no seu campo de visão, e não ocorrer a possibilidade de ter "pontos cegos".

Foi utilizada a criação da estrutura da matriz, inicializando tanto as linhas, quanto as colunas, com o valor = 30, indicando assim, que será gerada uma matriz final de 30 por 30 e um total de 900 posições. Outra variável importante a ser considerada na simulação é a quantidade de possibilidades que o caçador poderá se movimentar, onde consideramos que existiriam 8 posições ao seu redor e que poderiam ser utilizadas, como por exemplo, andar para cima, andar para cima na vertical superior esquerda, na vertical superior direita e assim por diante.

Conforme apresentado no desafio, a simulação do jogo deveria conter entre 5 e 10 caças. Por padrão, utilizamos 5 caças, as quais são geradas através de um vetor e representadas na matriz através dos números 1, 2, 3, 4 e 5.

A simulação começa na função Main(), em que é exibido um menu para o usuário escolher a opção desejada. Logo, a função Simulacao() é chamada, e lá é chamada a função CriaMatriz(), que realiza a montagem da matriz previamente parametrizada, utilizando-se de um laço de repetição FOR, a função CriaPosicoes(), que realiza a geração de números aleatórios utilizando o SRAND e as respectivas posições são salvas na STRUCT do caçador e das caças. Para que as caças não ocupem o mesmo lugar do caçador, foi utilizado o laço de repetição WHILE. São sorteadas posições de 0 a 29.

Um while foi implementado para executar enquanto houver caças, então, chama as funções Varredura(), que busca caças num raio de 5 células para todos os lados, formando uma matriz de 11x11, chama a função MovimentaCaca(), que faz a caça andar pela matriz aleatoriamente. Essas duas últimas funções mencionadas executam enquanto acharem uma caça, e quando achadas, imprimem a matriz já sem a caça que foi encontrada, e segue a execução.

Existem também as funções Atacar() e Cacar(). Atacar() que realizam o estado do caçador atacar a matriz e remover da matriz. Neste momento, em alguns casos, não todos, ocorre um pequeno problema, as vezes em um turno o caçador está ao lado da caça, no turno seguinte ambos somem da matriz, e no próximo turno o caçador retorna para a matriz normalmente e a caça não (pois foi atacada). Neste caso não foi encontrada uma solução a tempo. Já a função Cacar() é o ato do Caçador perseguir a caça que está no seu campo de visão.

3. Links de Acesso

3.1. GitHub

O projeto realizado e apresentado neste artigo, assim como o arquivo ReadMe (Leia-me) com as instruções e orientações de execução, está disponível no seguinte link:

https://github.com/edu-rodrigues/IA_Hunt_c-

3.2. YouTube

O vídeo com a execução e explicação do projeto realizado e apresentado neste artigo está disponível no seguinte link:

https://youtu.be/fhZBV7q-dR0

References

Boulic, R. and Renault, O. (1991) “3D Hierarchies for Animation”, In: New Trends in Animation and Visualization, Edited by Nadia Magnenat-Thalmann and Daniel Thalmann, John Wiley & Sons ltd., England.

Lou, Harbing. (2017) “AI in Video Games: Toward a More Intelligent Game”, http://sitn.hms.harvard.edu/flash/2017/ai-video-games-toward-intelligent-game/, Setembro.

Wikipedia. (2020) “Monte Carlo tree search”, https://en.wikipedia.org/wiki/Monte_Carlo_tree_search, Outubro.

Wikipedia. (2020) “Real-time strategy”, https://en.wikipedia.org/wiki/Real-time_strategy, Outubro.

Wikipedia. (2020) “Finite-state machine”, https://en.wikipedia.org/wiki/Finite-state_machine, Outubro.

Wikipedia. (2020) “Non-player character”, https://en.wikipedia.org/wiki/Non-player_character, Outubro.