Aplicação de *Behavior Tree* para IA de *NPCs* inimigos em jogos de *survival horror*

João Victor Pacheco Ferreira[1], André Kishimoto[1]

[1]Faculdade de Computação e Informática (FCI)
Universidade Presbiteriana Mackenzie – São Paulo, SP – Brasil

joaopachecoferreira03@gmail.com andre.kishimoto@mackenzie.br

Resumo *Este artigo tem como objetivo analisar e aplicar os conceitos de Behavior Tree na implementação de personagens não controlados pelo jogador (Non-Playable Characters ou NPCs) em jogos do gênero survival horror. Como principal referência para este estudo, foi usado o jogo Alien: Isolation (2014), que emprega Inteligência Artificial (IA) baseada em Behavior Trees para criar comportamentos dinâmicos e adaptativos nos inimigos. O uso dessa técnica permite que o sistema de IA analise as ações dos jogadores e responda de forma eficiente através de uma estrutura hierárquica de tomada de decisão.*

Palavras-chave: *Behavior Trees, Inteligência Artificial, Game AI, Personagens Não Jogáveis, Survival Horror*

Abstract. *This paper aims to analyze and apply the concepts of Behavior Trees in the implementation of non-playable characters (NPCs) in survival horror games. The main reference for this study is the game Alien: Isolation (2014), which employs Artificial Intelligence (AI) based on Behavior Trees to create dynamic and adaptive behaviors in enemy NPCs. This technique enables the AI system to analyze player actions and respond efficiently through a hierarchical decision-making structure.*

Keywords: *Behavior Trees, Artificial Intelligence, Game AI, Non-Playable Characters, Survival Horror*

1. INTRODUÇÃO

A indústria de jogos digitais tem crescido exponencialmente ao longo dos anos. Segundo o *Insight Report Videogames: market data & analysis* (STATISTA, 2023), ela movimentou cerca de US$197 bilhões em 2022, apresentando alta em relação aos US$184 bilhões registrados no ano de 2021, seguindo as suas projeções com a crescente de em média 12,1% ao ano.

Um exemplo desse crescimento é o gênero *survival horror*, que teve seu auge nos anos 90 com títulos icônicos como *Resident Evil* (1996) e *Silent Hill* (1999). Em 2024, houve o retorno do clássico *Alone in the Dark*, considerado o "pai do *survival horror*". O relançamento trouxe uma atmosfera sombria com gráficos inovadores, além de contar

com um elenco de peso, incluindo os atores David Harbour e Jodie (CONCEIÇÃO, 2023).

A implementação de *Behavior Trees* começou a ganhar destaque com *Halo 2* lançado em 2004, que trouxe avanços significativos para a comunidade de desenvolvimento de jogos. O jogo introduziu comportamentos complexos e realistas para NPCs (*Non-Playable Characters*) aliados e inimigos, revolucionando o design de jogos de tiro em primeira pessoa. Esse avanço foi essencial para criar desafios mais inteligentes, o que incentivou a adoção das *Behavior Trees* para aprimorar a jogabilidade e a Inteligência Artificial (IA) dos jogos, sendo que os desenvolvedores precisam lidar com o crescente problema da complexidade na criação de inimigos (ISLA, 2005).

Dez anos depois, *Alien: Isolation* (2014) aperfeiçoou essa técnica, utilizando uma versão mais sofisticada e moderna das *Behavior Trees*. Nesse jogo, a IA do inimigo reage e se adapta dinamicamente ao comportamento do jogador, elevando o nível de desafio e imersão (THOMPSON, 2020).

No estudo intitulado *"Tuning Stressful Experience in Virtual Reality Games"* (BRAMBILLA et al., 2023), os autores analisam como o nível de estresse dos jogadores aumenta em ambientes virtuais imersivos com NPCs perseguidores. O experimento utilizou uma Máquina de Estados Finitos (*Finite State Machine* ou FSM), na qual o NPC seguia um conjunto de estados predefinidos para perseguir o jogador.

Com base nos avanços observados em *Halo 2* e *Alien: Isolation*, onde as *Behavior Trees* foram aplicadas para criar comportamentos mais adaptativos e dinâmicos, este artigo busca verificar se o uso de *Behavior Trees* em NPCs poderia elevar significativamente o nível de imersão, estresse e dinamismo em experiências de jogo, considerando também os resultados do estudo de Brambilla et al. (2023).

2. REFERENCIAL TEÓRICO

De acordo com Bourg (2004), uma das técnicas mais usadas para a aplicação de IA em NPCs é a FSM. A popularidade das FSMs se deve à sua simplicidade e praticidade; muitos NPCs têm uma implementação relativamente pequena, com apenas alguns estados, onde cada estado representa um comportamento específico ou uma condição. Em uma FSM, apenas um estado é considerado ativo por vez, e os estados são conectados por transições que realizam mudanças com base em condições predefinidas.

Uma variação da FSM, conhecida como Máquinas de Estados Finitos Hierárquicas (*Hierarchical Finite State Machine* ou HFSM), oferece uma solução para a complexidade de FSMs grandes, permitindo que um estado contenha outras máquinas de estados finitos. Essa abordagem modulariza a estrutura, reduzindo a complexidade ao agrupar relacionamentos sob um único estado que gerencia seus próprios subestados. Por exemplo, um personagem de guarda pode ter um estado de alto nível denominado “Combate”, que inclui estados subordinados como “Engajar”, “Desviar” e “Recuar” (MILLINGTON, 2020).

Na produção do jogo *The Last of Us* (2013), foi adotada uma abordagem modular com FSM para implementar a IA, como descreve Mark Botta: *“Queríamos que os infectados parecessem fundamentalmente diferentes dos caçadores. Enquanto os caçadores*

trabalham em grupo e se comunicam com palavras e gestos, os infectados transmitem uma sensação de caos e alienação" (BOTTA, 2015, p. 33, tradução nossa).

A facilidade de manipulação e implementação das FSMs as torna adequadas para situações em que as respostas são previsíveis e, além disso, apresentam bom desempenho. No entanto, à medida que as FSMs aumentam em tamanho, com mais estados e transições, podem se tornar difíceis de gerenciar, levando à chamada "explosão de estados", conforme menciona Rabin (2014).

Outra possibilidade de implementar IA para NPCs é conhecida como GOAP (*Goal Oriented Action Planning*). Ao invés dos personagens apenas interagirem e reagirem ao ambiente, eles passam a tomar decisões autônomas para atingir determinados objetivos. Ao avaliar o estado atual e o objetivo final, o agente busca nas séries de opções de ações qual é a melhor para atingir o seu propósito. O agente vai reavaliando suas opções na medida que completa cada ação, baseando-se nas consequências de suas ações passadas para tomar as decisões futuras. Portanto, o comportamento dos personagens acaba sendo dinâmico e as possibilidades dentro do jogo se tornam flexíveis. No entanto, o custo computacional desta ferramenta é alto, principalmente em jogos com muitos agentes. Além disso, uma configuração malfeita pode gerar reações inesperadas dos personagens (KLOOSTER, 2024).

Uma alternativa às FSMs para implementação de IA é o uso da técnica de *Behavior Trees*. *Behavior Trees* (BTs) são usadas para a criação de comportamentos complexos, onde o sistema de árvore, uma estrutura hierárquica, organiza as ações e decisões de um agente. Essa estrutura de árvore é composta por nós que controlam a execução de tarefas, permitindo a modularidade e o reuso de comportamentos, e inclui:

- **Nós Controladores**: Definem a lógica de execução e controlam os nós filhos, determinando a ordem em que devem ser executados.
- **Nós Folha**: Representam as ações ou condições que a IA pode realizar ou avaliar.
- **Nós Decoradores**: Modificam o comportamento dos nós filhos, como inverter um resultado ou repetir sua execução.

Essa organização modular e hierárquica permite que as BTs sejam flexíveis e adaptáveis, facilitando a implementação de comportamentos dinâmicos em agentes de IA.

O uso de *Behavior Trees* permite uma modularidade que facilita a adição, alteração ou remoção de comportamentos. Sua estrutura hierárquica, na qual os nós representam comportamentos específicos, possibilita a agrupamento em sequências ou seletores. Cada ação, condição ou controlador funciona de forma independente e pode ser facilmente conectado ou desconectado da árvore principal. Dessa forma, diversas ações podem ser criadas, como afirmam Alex J. Champandard e Philip Dunstan: *"Condições modulares com responsabilidades simples (únicas) são combinadas para formar outras complexas"* (CHAMPARD; DUNSTAN, 2015, p. 91, tradução nossa).

Certas combinações de nós podem gerar ações distintas devido à estrutura de execução de uma BT. A árvore inicia em um nó raiz e avalia cada nó sequencialmente, onde os seletores e sequências determinam quais nós filhos serão avaliados ou executados. Esse processo cria um fluxo lógico que reage a mudanças no ambiente, permitindo que a IA se adapte dinamicamente.

Um exemplo prático do conceito pode ser ilustrado por um inimigo tentando acessar uma sala onde o jogador se encontra. Utilizando três estágios na árvore, observa-se como é possível criar, expandir e incrementar a estrutura, que pode ser tão simples ou complexa quanto necessário, com nós compostos, como ilustrado na Figura 1.

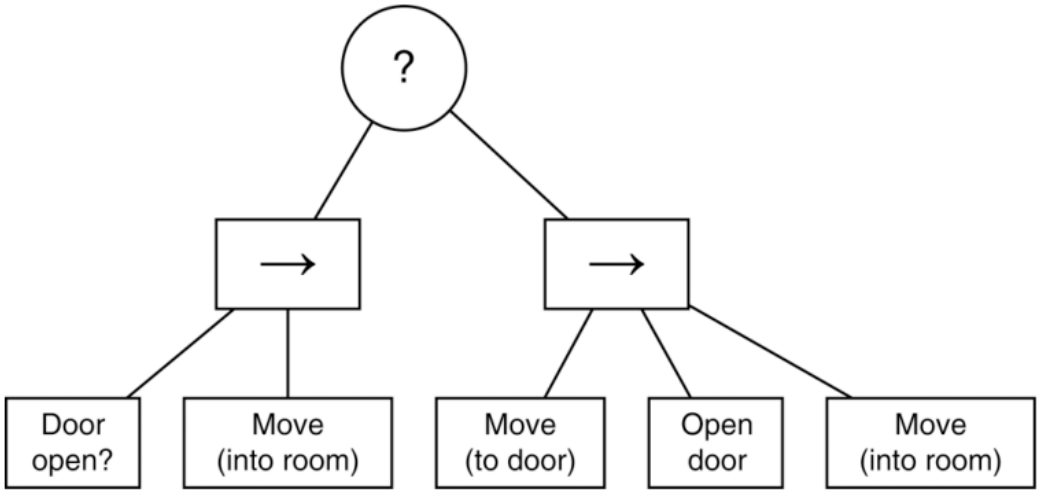


Figura 1: Behavior Tree com nós compostos (Fonte: MILLINGTON, 2020).

A Figura 1 apresenta uma *Behavior Tree* com seletores que permitem escolhas: o NPC pode primeiro verificar se a porta está aberta, usando uma tarefa de condição para então entrar na sala. Em um segundo caso, o NPC se desloca até a porta, realiza uma animação para abri-la e, então, entra na sala (MILLINGTON, 2020).

Já na Figura 2, observa-se uma *Behavior Tree* composta por uma única tarefa: a movimentação do NPC para entrar em uma sala. Esse tipo de configuração simples pode levar o NPC a realizar ações sem inteligência, executando movimentos “burros”.

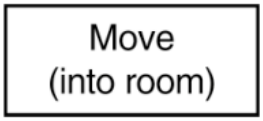


Figura 2: Behavior Tree simples, com apenas um nó (Fonte: MILLINGTON, 2020).

Um exemplo do uso de *Behavior Trees* pode ser encontrado no jogo *Final Fantasy XV* (2016), que utilizou a ferramenta *AI Graph Tool* como a *engine* escolhida para a produção das decisões dos personagens. De acordo com Miyake (2017), a implementação permitiu a reutilização de todos os nós, de modo que um nó de uma FSM poderia ser utilizado também em uma *Behavior Tree*, embora a execução fosse diferente em cada caso.

Para viabilizar essa abordagem, ambas as estruturas (FSM e *Behavior Tree*) foram executadas simultaneamente. Ambas as arquiteturas realizam o início, o término e até mesmo a atualização de estados juntas. A principal diferença entre elas reside no motivo

pelo qual um nó é interrompido: para a *Behavior Tree*, o nó termina quando uma condição interna é atendida; já na FSM, a interrupção ocorre devido a uma condição externa. Se os nós forem projetados com esses componentes, será possível integrar as duas arquiteturas de forma eficaz (MIYAKE, 2017).

No jogo *Alien: Isolation*, o uso de *Behavior Tree* permite apresentar comportamentos inteligentes que simulam aprendizado do inimigo Xenomorfo. Por exemplo, se o Xenomorfo for ferido com fogo pelo jogador, ele "lembrará" dessa experiência de dor e tentará atacar o jogador de maneiras diferentes, adotando novas estratégias com base nas interações anteriores. A criação do NPC Xenomorfo envolve duas estruturas que operam simultaneamente. Uma delas é o *Director AI*, semelhante ao que existe no jogo *Left 4 Dead*, de 2009, que comunica ao Xenomorfo a localização geral do jogador, sem, contudo, direcioná-lo para uma ação específica. Assim, as decisões do Xenomorfo são guiadas pela *Behavior Tree*. Mesmo sabendo onde o jogador está, o NPC possui uma gama de escolhas para persegui-lo, seja de forma silenciosa ou ruidosa, esperando que o jogador cometa um erro para que possa capturá-lo (THOMPSON, 2020).

Outro exemplo do uso de *Behavior Trees* é encontrado no jogo *Resident Evil 2 Remake* (2020), onde a IA do personagem Mr. X (ou *"Nemesis"*) é composta por uma série de decisões organizadas em sub-árvores, que se ativam de acordo com as situações enfrentadas durante o jogo. Embora similar ao Xenomorfo em alguns aspectos, a tecnologia usada para Mr. X permite que ele seja "posicionado" próximo ao jogador quando necessário, evitando que o NPC se perca em um mapa grande e vazio. Isso contribui para uma experiência de perseguição mais intensa e dinâmica. Assim como em *Alien: Isolation*, o personagem Mr. X também utiliza um *feedback* sonoro para rastrear o jogador, mantendo sempre uma noção de sua localização, mas sem instruções diretas sobre quais ações tomar.

3. METODOLOGIA

Com os estudos realizados durante o período da pesquisa, fica evidente a importância do desenvolvimento de uma *Behavior Tree* para jogos do gênero *survival horror*. A experiência de imersão proporcionada no jogo *Alien: Isolation* destaca a relevância de uma IA aplicada a NPCs do tipo *"Nemesis"*, cuja função é perseguir o jogador e tornar a experiência a mais imersiva e impactante possível.

Para a aplicação deste projeto, foi criado uma demonstração que empregou os conceitos estudados nos artigos e textos apresentados, utilizando o método de pesquisa qualitativa e o levantamento de informações sobre o desenvolvimento de *Behavior Trees* para o gênero *survival horror*.

A demonstração foi desenvolvida na *game engine* Unity, incorporando todos os conceitos aprendidos e aplicando-os na prática, de forma que o resultado seja funcional e aplicável tanto para o jogador quanto para o desenvolvedor. Muitos dos princípios usados foram baseados na estrutura de *Behavior Trees* descrita no livro *AI for Games* (MILLINGTON, 2020). Tal estrutura permite a utilização de tarefas básicas: a CPU recebe uma tarefa para executar e, ao concluir, retorna um código de status indicando sucesso ou falha, o que pode ser representado por um valor *booleano* nesta etapa.

4. RESULTADOS E DISCUSSÃO

O projeto desenvolvido neste trabalho possui um NPC cujo comportamento é definido por uma *Behavior Tree* com três funções: "Procura" (o NPC fica andando em um perímetro pré-estabelecido, procurando pelo jogador), "Caça" (o NPC começa a perseguir o jogador) e "Espreita" (quando o jogador não é encontrado, o NPC se esconde em locais definidos como esconderijo), como ilustrado na Figura 3.

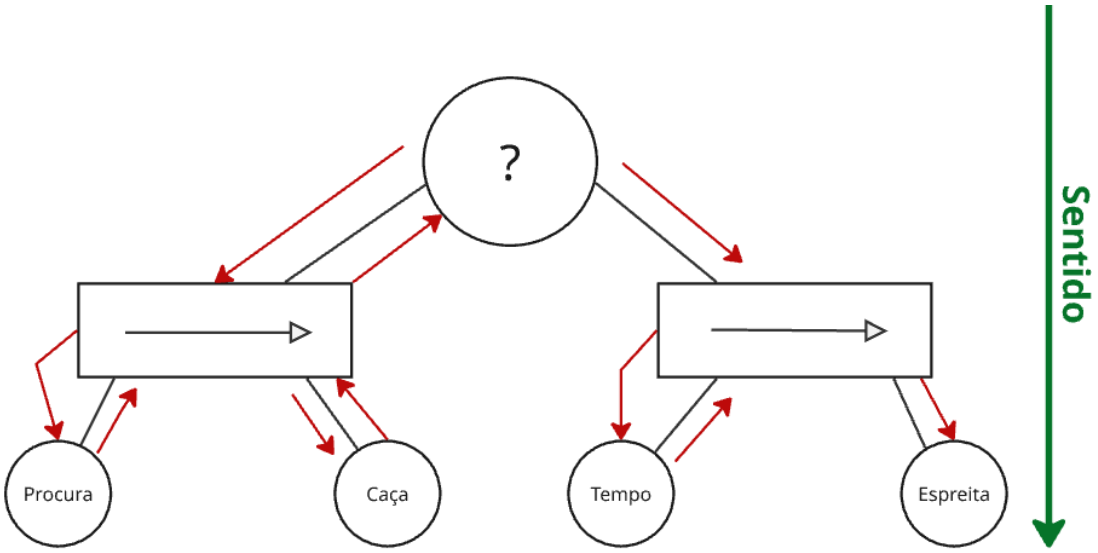


Figura 3: Behavior Tree que define o comportamento do NPC do projeto (Fonte: elaborado pelo autor).

A *Behavior Tree* possui dois nós seletores como raiz para os outros nós. O primeiro seletor possui dois filhos, os nós de procura e caça. Caso um deles falhe, a árvore avança para o próximo seletor, que contém um temporizador que define uma pausa de 10 segundos antes de alternar para o nó de espreita.

O sistema foi construído utilizando um dos sistemas internos da Unity conhecido como *NavMesh*, que gera um grafo de arestas para a navegação do NPC, e o componente *RayCast*, usado para simular a linha de visão do NPC.

Durante a avaliação de cada nó da *Behavior Tree*, um raio (*RayCast*) é traçado entre a posição do NPC e do jogador. Caso ocorra uma colisão do raio com o jogador, sem existir nenhuma colisão intermediária, o NPC entende que está "enxergando" o jogador e o nó retorna um valor de sucesso para a árvore.

A *Behavior Tree* foi construída começando pela raiz (*rootNode*), que recebe um nó do tipo seletor com memória (*MemSelectorNode*), como apresentado na Figura 4. A memória, neste caso, é usada para que o nó consiga lembrar qual nó está processando, a fim de evitar ciclos (*loops*) em algum de seus nós filhos.

```
rootNode = new MemSelectorNode(new List<Node>
{
    new SequenceNode(new List<Node>
    {
        new SearchPlayerNode(transform, agent, player),
        new ChasePlayerNode(transform, player, agent)
    }),
    new SequenceNode(new List<Node>
    {
        new TimerNode(1f),
        new StalkPlayerNode(transform, player, agent, hidingSpots)
    })
});
```

Figura 4: Behavior Tree construída em código (Fonte: elaborado pelo autor).

O nó seletor avalia os nós filhos na hierarquia em que a subárvore foi construída e grava o índice do filho que está sendo avaliado no momento (estado *NodeState.Running*), como apresentado da Figura 5.

O nó filho "memorizado" continua sendo avaliado até que termine com sucesso ou falha. Em caso de sucesso, o nó filho retorna sucesso ao nó pai (seletor). Em caso de falha, o nó seletor avança e avalia o próximo nó filho.

```
    if (runningChild >= 0)
    {
        NodeState state = children[runningChild].Evaluate();
        if (state == NodeState.Running) return NodeState.Running;
        if (state == NodeState.Success)
        {
            runningChild = -1;
            return NodeState.Success;
        }
        runningChild++;
    }

    for (int i = runningChild < 0 ? 0 : runningChild; i < children.Count; i++)
    {
        NodeState state = children[i].Evaluate();
        if (state == NodeState.Running)
        {
            runningChild = i;
            return NodeState.Running;
        }
        if (state == NodeState.Success)
        {
            runningChild = -1;
            return NodeState.Success;
        }
    }

    runningChild = -1;
    return NodeState.Failure;
```

Figura 5: Nó seletor com memória (Fonte: elaborado pelo autor).

Somente depois que todos os filhos tiverem falhado é que o *MemSelectorNode* considera seu próprio estado como falha e limpa a sua memória. Com essa estratégia,

garante-se que, uma vez que um ramo da árvore seja iniciado, ele não seja interrompido e reiniciado a cada momento do jogo.

O nó de sequência (*SequenceNode*), cujo código é apresentado na Figura 6, é utilizado para gerar um dinamismo na seleção dos nós presentes em sua estrutura, como, por exemplo, quando o nó de busca pelo jogador (*SearchPlayerNode)* é ativado como sucesso, ele segue para o próximo nó do seletor (*ChasePlayerNode*, na Figura 4).

```
using System.Collections.Generic;
// Criação de um nó de sequência executa as ações em ordem e falha se alguma delas falhar.
4 referências
public class SequenceNode : Node
{
    private List<Node> children = new List<Node>();

    3 referências
    public SequenceNode(List<Node> nodes)
    {
        children = nodes;
    }

    8 referências
    public override NodeState Evaluate()
    {
        foreach (var child in children)
        {
            NodeState state = child.Evaluate();
            //Verificação dos estados
            if (state == NodeState.Running) return NodeState.Running;
            if (state == NodeState.Failure) return NodeState.Failure;
        }
        return NodeState.Success;
    }
}
```

Figura 6: Nó de sequência (Fonte: elaborado pelo autor).

Na construção desse nó, é realizado a verificação dos estados presentes para a execução dos nós presentes e ter uma reação quando o nó que está vigente falha para poder avançar em um outro nó que está em sua formação.

Um dos nós usados na *Behavior Tree* do NPC é o *SearchPlayerNode*, que é responsável por movimentar o NPC e, ao mesmo tempo, monitorar se o jogador está em seu campo de visão ou não. A implementação dessa tarefa é mostrada na Figura 7.

```
public override NodeState Evaluate()
{
    if (!hasStarted)
    {
        startTime = Time.time;
        hasStarted = true;
        timer = 0f;
        isMoving = false;
        Debug.Log(" [Search] Iniciando busca (10s)...");
    }

    float elapsedTime = Time.time - startTime;
    float distanceToPlayer = Vector3.Distance(npc.position, player.position);

    if (PlayerInSight() && distanceToPlayer < maxChaseDistance)
    {
        Debug.Log(" [Search] Jogador encontrado! → Success");
        hasStarted = false;
        return NodeState.Success;
    }
```

Figura 7: Nó de busca (Fonte: elaborado pelo autor).

Internamente, o nó *SearchPlayerNode* utiliza dois mecanismos de temporização para coordenar o comportamento de busca (Figura 8). O primeiro envolve as variáveis *timer* e *wanderTimer*, que definem com que frequência o NPC escolhe um novo destino aleatório: a cada chamada de *Evaluate(),* acumula-se *timer += Time.deltaTime*; quando esse valor ultrapassa *wanderTimer* ou o agente alcança seu destino anterior, gera-se uma direção aleatória dentro de um raio (*wanderRadius*), ajusta-se essa posição ao *NavMesh* com *NavMesh.SamplePosition* e chama-se *agent.SetDestination()*, reiniciano o *timer* e sinalizando com *isMoving = true*.

```
if (elapsedTime >= maxSearchTime)
{
    Debug.LogWarning(" [Search] Tempo de busca (10s) esgotado → Failure");
    hasStarted = false;
    return NodeState.Failure;
}

timer += Time.deltaTime;
if (!agent.pathPending
    && (agent.remainingDistance < agent.stoppingDistance
        || !isMoving
        || timer >= wanderTimer))
{
    Vector3 randomDir = Random.insideUnitSphere * wanderRadius + npc.position;
    if (NavMesh.SamplePosition(randomDir, out NavMeshHit hit, wanderRadius, NavMesh.AllAreas))
    {
        agent.SetDestination(hit.position);
        isMoving = true;
        timer = 0f;
        Debug.Log($" [Search] Indo para ponto: {hit.position}");
    }
}

if (agent.remainingDistance <= agent.stoppingDistance)
    isMoving = false;

return NodeState.Running;
```

Figura 8: Contador e movimentação do nó de busca (Fonte: elaborado pelo autor).

Em paralelo, um segundo temporizador garante que a busca não se estenda indefinidamente: nas primeiras linhas de *Evaluate()*, quando *hasStarted* é falso, registra-se *startTime = Time.time*, marca-se *hasStarted = true* e a variável *timer* é zerada, iniciando a contagem total (Figura 7).

A cada quadro, calcula-se *elapsedTime = Time.time – startTime*; se o método *PlayerInSight()* retornar verdadeiro e a distância até o jogador for menor que *maxChaseDistance*, o nó reinicia *hasStarted* e retorna *Success*, acionando imediatamente o ramo de perseguição. Se, por outro lado, *elapsedTime* atinge *maxSearchTime* (10 segundos) sem detectar o jogador, o nó registra uma falha e retorna *Failure*, permitindo a transição para o modo de espreita. Dessa forma, a implementação combina exploração randômica, controlada por um temporizador curto, com um limite global de busca, assegurando comportamento previsível, modular e fácil de ajustar.

Em cada chamada do método *Evaluate*(), o nó *SearchPlayerNode* calcula quanto tempo já se passou desde o início (Figura 7). Se, a qualquer momento, o raio de visão

detectar o jogador dentro do limite de distância pré-estabelecido, o nó imediatamente retorna sucesso e vai para o nó de Caça (*ChaseNode*).

O nó *ChaseNode* transforma a condição "jogador detectado" em movimento de perseguição, mantendo referências à posição do NPC e do jogador e ao *NavMesh* do NPC.

```
8 referências
public override NodeState Evaluate()
{
    agent.isStopped = false;
    agent.speed = 4f;

    float distance = Vector3.Distance(npc.position, player.position);

    if (!PlayerInSight() || distance > maxChaseDistance)
    {
        Debug.Log("Jogador fora de alcance! Parando perseguição...");
        return NodeState.Failure;
    }

    if (!agent.pathPending)
    {
        agent.SetDestination(player.position);
        Debug.Log("Perseguindo o jogador.");
    }

    return NodeState.Running;
}
```

Figura 9: Nó de caça (Fonte: elaborado pelo autor).

Em cada chamada ao método *Evaluate()* (Figura 9), *ChasePlayerNode* inicializa o *NavMeshAgent* do NPC com *agent.isStopped = false* e redefine seu parâmetro de movimentação *agent.speed* para um valor adequado à perseguição, garantindo agilidade ao agente.

Na sequência, calcula-se a distância até o jogador por meio *de Vector3.Distance(npc.position, player.position)* e invoca-se o método *PlayerInSight()*, que dispara um *raycast* dos "olhos" do *NPC* até o peito do jogador para confirmar se a linha de visão do NPC não está bloqueada. Caso o jogador não esteja visível ou ultrapasse o limite definido em *maxChaseDistance*, o nó retorna *NodeState.Failure*, sinalizando à *Behavior Tree* que a tentativa de perseguição falhou e que deve transitar para outro comportamento.

Por fim, é verificado se o *NavMeshAgent* já terminou de calcular o caminho anterior, avaliando a propriedade *agent.pathPending*. Caso não haja cálculo pendente, chama-se *agent.SetDestination(player.position)*, definindo a posição atual do jogador como novo destino do agente e garantindo que o *NPC* continue se movendo em sua direção.

O método retorna *NodeState.Running*, sinalizando à *Behavior Tree* que o nó de perseguição permanece ativo e não foi concluído, de modo que continuará sendo executado até que outras condições de sucesso ou falha sejam atendidas.

O último nó presente na *Behavior Tree* é o nó de espreita (*StalkPlayerNode*), usado para o NPC encontrar um esconderijo, se deslocar até o esconderijo e aguardar o jogador se aproximar, para então disparar a perseguição novamente.

A escolha do melhor esconderijo é feita avaliando todos os pontos de esconderijo disponíveis no ambiente, sendo que o ponto escolhido é aquele cuja distância em relação ao jogador seja a menor.

Ao chegar no esconderijo, o NPC fica em estado de espera. Durante essa espera, a busca não é reiniciada e nem há mudança de comportamento, sendo que apenas a aproximação do jogador aciona o critério de sucesso de execução do nó (Figura 10).

```
public override NodeState Evaluate()
{
    if (PlayerNearby())
    {
        Debug.Log("Jogador detectado próximo! Perseguindo.");
        agent.isStopped = false;
        isAtSpot = false;
        return NodeState.Success;
    }

    if (!isAtSpot)
    {
        Transform bestSpot = FindBestHidingSpot();
        if (bestSpot != null)
        {
            agent.SetDestination(bestSpot.position);
            if (Vector3.Distance(npc.position, bestSpot.position) <= agent.stoppingDistance + 0.5f)
            {
                agent.isStopped = true;
                isAtSpot = true;

            }

            return NodeState.Running;
        }
        else
        {
            return NodeState.Failure;
        }
    }
    return NodeState.Running;
}
```

Figura10: Nó de espreitar (Fonte: elaborado pelo autor).

Inicialmente, o nó de espreita *StalkPlayerNode* chama *PlayerNearby()*, que calcula a distância entre o NPC e o jogador. Caso a distância seja menor ou igual a um valor configurado no projeto (*detectionRange*), o retorno *NodeState.Success* sinaliza à *Behavior Tree* que ela deve retornar ao ramo de perseguição.

Caso contrário, verifica-se a flag *isAtSpot:* se o NPC ainda não estiver no esconderijo, *FindBestHidingSpot()* é chamado para selecionar o ponto de esconderijo mais próximo do jogador (*bestSpot*), respeitando uma distância mínima de segurança. Uma vez encontrado o esconderijo, define-se a posição de destino do NPC e *isAtSpot* passa a armazenar *true* quando o NPC chega ao esconderijo, garantindo que ele permaneça no local. Caso não exista nenhum esconderijo válido, o nó retorna *NodeState.Failure*, permitindo à *Behavior Tree* tentar outro comportamento.

Durante toda a movimentação ou enquanto aguarda no esconderijo, o nó retorna *NodeState.Running*, mantendo ativo o ramo de espreita até que ocorra uma nova detecção ou *timeout* em outro nível da árvore.

Para a construção do ambiente do projeto, foram utilizados dois *assets* (artefatos) gratuitos obtidos na *Unity Store*, um para o mapa e um para a representação visual do NPC. Optou-se por usar *assets* com visual neutro (Figura 11), a fim de mostrar que o projeto pode ser aplicado tanto em jogos de horror quanto de outros gêneros.

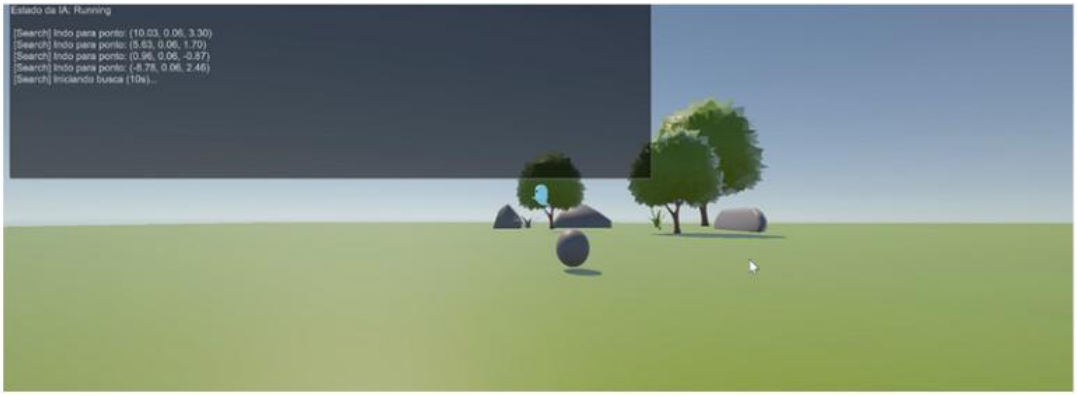
Figura 11: Captura de tela do projeto (Fonte: elaborado pelo autor).

A escolha da ambientação para este protótipo seguiu uma linha minimalista e ao mesmo tempo sugestiva, combinando modelos de baixo polígono com uma paleta de cores suaves que reforçam tanto a legibilidade do cenário quanto a atmosfera de leve tensão.

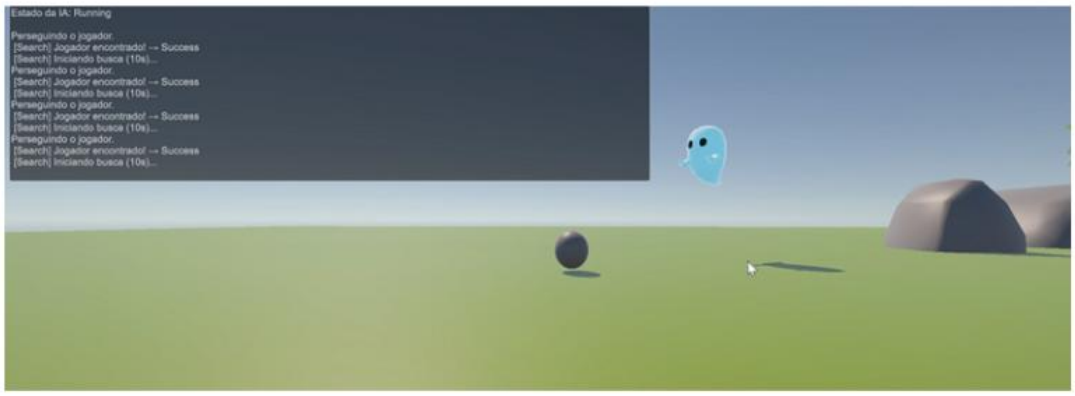
As rochas do ambiente não são apenas elementos estéticos: elas foram marcadas no *NavMesh* como pontos de *hiding spot*, ou seja, locais onde o fantasma pode se ocultar quando entra no modo de espreita. Esse posicionamento foi planejado para oferecer ao jogador pistas visuais sutis, como as sombras projetadas, e, ao mesmo tempo, permitir que o NPC utilize a malha de navegação para buscar abrigo de forma crível, aproximando-se das superfícies rochosas e "desaparecendo" atrás delas.

O uso de rochas como esconderijos reforça a ideia de que o ambiente é parte ativa da mecânica de jogo: ao perder o rastro do jogador, o fantasma não se restringe a um ponto fixo, mas desloca-se até o objeto mais próximo que ofereça cobertura visual. Essa dinâmica instiga o jogador a permanecer atento não apenas à movimentação do próprio adversário, mas também ao *layout* do ambiente, avaliando rotas de fuga e pontos de visibilidade.

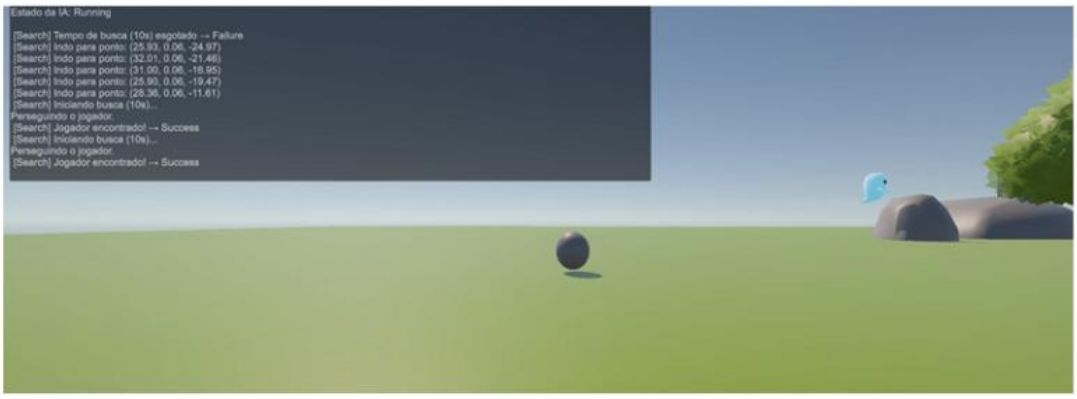
A Figura 12 apresenta momentos diferentes do jogo, com as ações do NPC.



(a)



(b)



(c)

Figura 12: Telas do projeto em diferentes momentos (Fonte: elaborado pelo autor).

Em (a), o jogador está fora do campo de visão do NPC, que entra em modo procura. Em (b), o NPC entra no modo de perseguição ao avistar o jogador, parando de seguilo somente quando ele não está mais no campo de visão do NPC. Por fim, o NPC se esconde no local mais próximo esperando que o jogador retorne e fica no modo espreita (c).

5.CONCLUSÃO

Em conclusão, a implementação de *Behavior Trees* no Unity demonstrou-se uma abordagem robusta e flexível para controlar o comportamento de NPCs em um jogo de *survival horror*. Ao combinar um *NavMeshAgent* para navegação com *raycasts* que simulam a linha de visão, criamos um NPC capaz de alternar entre três estados principais – busca aleatória, perseguição reativa e espreita estratégica – de forma modular e extensível.

A utilização de um nó seletor com memória garantiu que, uma vez iniciado um ramo de comportamento, o NPC ali permanecesse até que ele se completasse, antes de retornar à busca inicial, resultando em uma inteligência mais previsível e imersiva. Nos testes realizados, o fantasma procurou o jogador por até dez segundos, perseguiu-o assim que avistado e, ao falhar na detecção, deslocou-se para um esconderijo onde aguardou até o jogador se aproximar, comportamento esse que elevou o desafio e a sensação de presença no ambiente de jogo.

Para trabalhos futuros, sugere-se aplicar este mesmo modelo de *Behavior Tree* em diferentes gêneros (como *stealth*/furtividade, tiro em primeira pessoa, RPG ou plataforma), avaliando a eficácia de seus nós em contextos diversos. Além disso, recomenda-se o desenvolvimento de métricas quantitativas, como o tempo médio de detecção e a taxa de sucesso em perseguições, de modo a medir objetivamente o impacto da IA na experiência do jogador.

A adoção de heurísticas mais sofisticadas para seleção de esconderijos e *waypoints* dinâmicos pode tornar o comportamento ainda mais natural, enquanto técnicas de aprendizado de máquina, como reforço, poderiam ajustar automaticamente parâmetros críticos (alcance de visão, duração de busca), tornando a árvore de comportamento adaptativa ao estilo de jogo. Dessa forma, as *Behavior Trees* mostradas aqui não apenas estabelecem uma base sólida para IA de NPCs, mas também abrem caminho para ampliações que tornem jogos cada vez mais envolventes e desafiadores.

REFERÊNCIAS BIBLOGRÁFICAS

BOURG, D. M.; SEEMANN, G. AI for Game Developers. 1. ed. Sebastopol: O'Reilly Media, 2004.

BRAMBILLA, S. et al. Tuning stressful experience in virtual reality games. Proceedings of the 15th Biannual Conference of the Italian SIGCHI Chapter. Anais...New York, NY, USA: ACM, 2023. Disponível em: https://dl.acm.org/doi/pdf/10.1145/3605390.3605412 Acesso em: 11 de novembro de 2024.

CHAMPANDARD, A. J.; DUNSTAN, P. The behavior tree starter kit. Disponível em: <https://www.gameaipro.com/GameAIPro/GameAIPro_Chapter06_The_Behavior_Tree_Starter_Kit.pdf>. Acesso em: 11 novembro de 2024.

CONCEIÇÃO, V. Como o survival horror que conhecemos hoje deve aos clássicos dos anos 90 e 5 games obrigatórios para todo fã do gênero. Disponível em: https://br.ign.com/games/112177/feature/como-o-survival-horror-que-conhecemos-hoje-deve-aos-classicos-dos-anos-90-e-5-games-obrigatorios-par. Acesso em: 11 de novembro de 2024.

ISLA, D. GDC 2005 proceeding: Handling complexity in the Halo 2 AI. Disponível em: https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai. Acesso em: 11 de novembro de 2024.

KLOOSTER, P. What is Goap? | GOAP. Disponível em: <https://goap.crashkonijn.com/readme/theory>. Acesso em: 13 de junho de 2025.

MILLINGTON, I. AI for Games, 3. ed. London: CRC Press, 2020.

MIYAKE, Y. et al. A Character Decision-Making System for FINAL FANTASY XV by Combining Behavior Trees and State Machines. Disponível em: https://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter11_A_Character_Decision-Making_System_for_FINAL_FANTASY_XV_by_Combining_Behavior_Trees_and_State_Machines.pdf. Acesso em: 11 de novembro de 2024.

STATISTA. Video Games: market data & analysis. Disponível em: https://www.statista.com/study/39310/video-games. Acesso em: 11 de novembro de 2024.

THOMPSON, T. Revisiting the AI of Alien: Isolation. Disponível em https://www.gamedeveloper.com/design/revisiting-the-ai-of-alien-isolation. Acesso em: 02 de novembro de 2024.