ESTRATÉGIAS DE IMPLEMENTAÇÃO DE JOGOS DE AÇÃO COM INTELIGÊNCIA ARTIFICIAL NO GAME ENGINE UNREAL

Henrique Rachti Assumpção (IC) e Prof. Dr. Luciano Silva (Orientador)

Apoio: PIBIC CNPq

RESUMO

Jogos do gênero ação oferecem diversos desafios computacionais para implementação, principalmente quando associados a mecanismos de Inteligência Artificial (IA). Este trabalho apresenta algumas estratégias de implementação de jogos no *game engine* UNREAL, com requisitos de Inteligência Artificial. Foram realizadas diversas implementações de IA utilizando o sistema de *scripting* visual BLUPRINT.

Palavras-chave: Jogos de Ação, Inteligência Artificial, Jogos Digitais.

ABSTRACT

Action games offer several computational challenges for implementation, especially when associated with Artificial Intelligence (AI) mechanisms. This work presents some strategies of action gamse implementation in the game engine UNREAL, with requirements of Artificial Intelligence. Several IA implementations were performed using the BLUPRINT visual scripting system.

Keywords: Action Games, Artificial Inteligence, Digital Games.

1. INTRODUÇÃO

O visual e a física dos jogos mais atuais estão cada vez mais semelhantes com a realidade, e um dos principais responsáveis por esse grande avanço são as *Game Engines,* que facilitam drasticamente o trabalho dos projetistas e desenvolvedores, apresentando uma ambiente visual não muito complexo para sua utilização e bibliotecas com diversas funcionalidades, que não obrigam que os trabalhos sejam iniciados do zero e que recursos complexos sejam mais simples de serem utilizados, além de não obrigar os desenvolvedores a reprogramarem um jogo para que ele seja lançado em uma plataforma diferente da original, ou em diferentes plataformas simultaneamente.

O primeiro jogo lançado a partir de uma *Game Engine* foi *Driller*, em 1987, produzido através da *Freescape Engine*, o primeiro motor gráfico 3D da história, criada pela Incentive Software. Na época, o termo *Game Engine* nem existia, e ele só foi aparecer após a popularização dos jogos em primeira pessoa, *Wolfenstein 3D*, *Doom* e *Quake*, sendo que diversos trabalhos, posteriormente, utilizaram a *engine* desses jogos, por conta dos seus altos potenciais.

Dos anos 80 para o atual momento da indústria de jogos, as *Game Engines* avançaram muito, e hoje uma das mais utilizadas e avançadas *engines* é a Unreal Engine, lançado em sua primeira versão originalmente em 1998 pela Epic Games, e que atualmente está em sua quarta geração, lançada em 2014, sempre sendo atualizada com novas funcionalidades.

Este trabalho propõe o estudo na implementação de jogos na Unreal Engine 4, tendo como base a produção de um jogo do gênero ação. Partindo do conhecimento das suas principais funcionalidades, dos recursos que a Unreal Engine 4 pode oferecer e, principalmente, do sistema de scripting visual *Blueprint*, este que possibilita desenvolver a lógica de um jogo inteiro sem programar uma linha de código em C++, serão então apresentados os componentes principais presentes em jogos do gênero ação, sua base lógica e como é desenvolvido da Unreal Engine 4 através do sistema *Blueprint*.

2. REFERENCIAL TEÓRICO

2.1 A engine Unreal

Desenvolvido pela Epic Games, a Unreal Engine é um motor com grande possibilidade de portabilidade entre diferentes plataformas, e atualmente é uma das ferramentas mais utilizada pelos desenvolvedores de jogos. Estando no momento em sua quarta geração, o conjunto de ferramentas da Unreal Engine foi construído com o intuito de trazer maior

facilidade no desenvolvimento de jogos, havendo a capacidade de trabalhar com uma ampla série de gêneros distintos de jogos para diferentes plataformas.

2.2 **Visão geral das ferramentas da engine**

O conjunto de ferramentas da Unreal Engine é bastante avançado, sendo que além de renderizar, ela contém um vasto conjunto de APIs e mecanismos desenvolvidos para a criação de físicas baseadas no mundo real, e também, editores de animações faciais, inteligência artificial, folhagem, iluminação, entre muitas outras funcionalidades. Na última versão da *engine*, a Unreal Engine 4, comparada a suas versões anteriores, foi incluído o operador de scripting visual *Blueprint*, sucessor do *Kismet* da Unreal Engine 3, e o sistema de *Voxel Cone Tracing,* permitindo utilizar iluminação global ampliada, não obrigando a prévia renderização da iluminação do jogo, além de o motor ter sido construído para funcionar perfeitamente nos consoles da oitava geração, PCs, sistemas Android com o processador Tegra K1, iOS, Mac OS X e HTML5

2.2 Simulação da Física dos jogos

Um dos aspectos mais importantes dos jogos é a simulação da física por trás dele, sendo responsável por trazer realidade e imersão ao *gameplay* do jogador, e para isso, na Unreal Engine é utilizado a engine de física PhysX 3.3, lançado pela NVIDIA, sendo possível realizar detecções de colisão extremadamente precisas e tornar as interações físicas entre os objetos do jogo mais realistas.

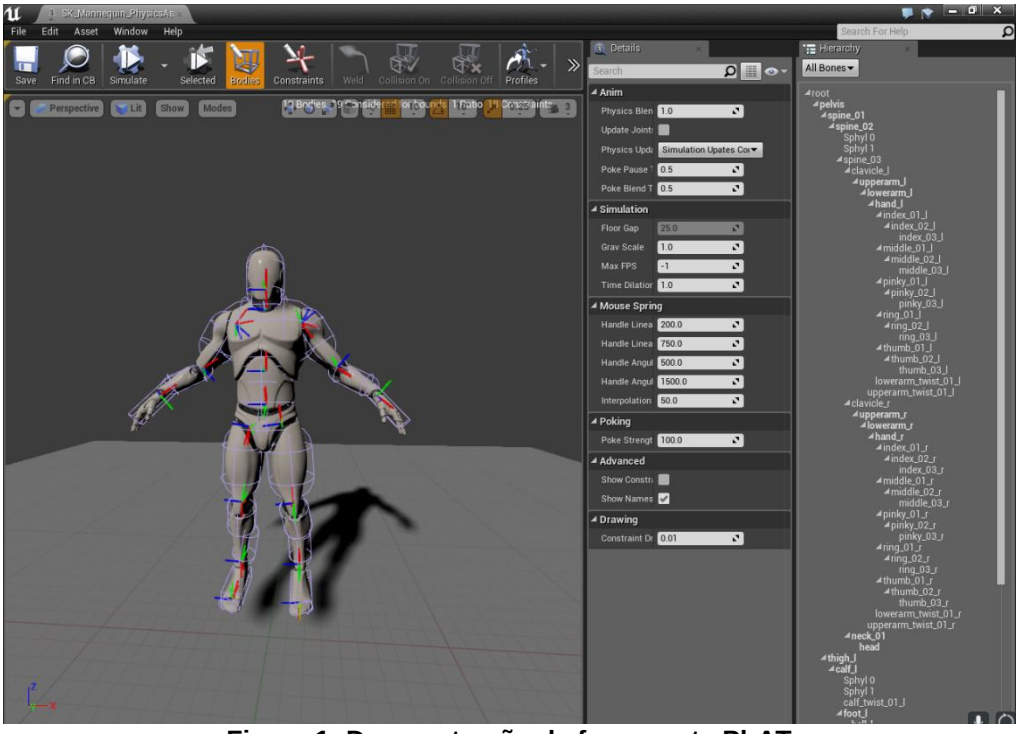

Figura 1: Demonstração da ferramenta PhAT

E utilizando a ferramenta **Physics Asset Tool** (**PhAT**), é possível modificar ainda mais a física dos corpos dos atores utilizados nos jogos, podendo manipular separadamente cada parte de sua constituição física através de **Skeletal Meshes**, recurso que também é muito útil no desenvolvimento das animações dos atores.

2.3 Sistema de Renderização

Na Unreal Engine 4 é utilizado o novo pipeline DirectX 11 para seu sistema de renderização, o que possibilita a utilização de:

- Iluminação Global: Com esse recurso, a iluminação dos cenários tem aspectos mais próximos da realidade, sendo que ele simula juntamente a uma fonte de luz, os seus subsequentes reflexos emitidos pela superfície de outros objetos, mesmo que eles não sejam reflexivos.
- Deferred Shading: Esse mecanismo dissocia a iluminação da renderização da geometria dos objetos, considerando que em *Render Targets,* denominados GBuffers, são armazenados os atributos dos materiais dos objetos na primeira passagem de um *deferred shader*, e a cada vez que há uma passagem de luz sobre eles, ao invés de redesenhar a geometria dos objetos para calcular os atributos, é apenas necessário acessar os dados das GBuffers.
- Pós-processamento: Os efeitos implantados por esse artifício possibilitam trabalhar com a aparência geral da cena antes de renderizar os objetos 3D na tela
- Simulação de partículas na GPU: Utilizando a GPU ao invés da CPU na simulação das partículas, torna a eficiência e frequência de elementos na tela muito maior, passando de centenas para milhares de partículas em um único frame.

2.4 Sistema de scripting visual Blueprint

O sistema de *scripting* visual *Blueprint* é uma interface baseada em nós que possibilita a construção da *gameplay* do jogo dentro do Unreal Editor, de modo bem flexível e poderoso, possibilitando que designers utilizem conceitos e ferramentas que eram de conhecimento apenas dos programadores, no desenvolvimento de seus jogos, além de ser possível depurar o projeto enquanto ele está em execução, o que facilita a alteração de elementos básicos no jogo, não sendo mais necessário ficar compilando o cenário por um período, para efetuar a alteração desejada, reduzindo assim, o tempo de iteração.

2.5 Graph: a base de implementação do Blueprint

A implementação no Blueprint é baseada em redes de nós conectadas, denominadas como *Graphs*, que durante a execução, segue o fluxo dessas conexões de nós. Em cada

graph é possível iniciar eventos, chamar funções para a realização de determinadas ações, transferir a execução à outro *graphs*, e iniciar *scripts* de eventos, e em todos os casos, a sua realização depende dos eventos ocorridos na execução do jogo.

2.6 Os *Event Graphs*

O tipo mais comum de *Graph* utilizado é o *Event Graph*, que a partir de determinados eventos que ocorrem no jogo, desencadeia a execução de uma sequência de nós, sendo então, um método utilizado para adicionar funcionalidades aos jogos.

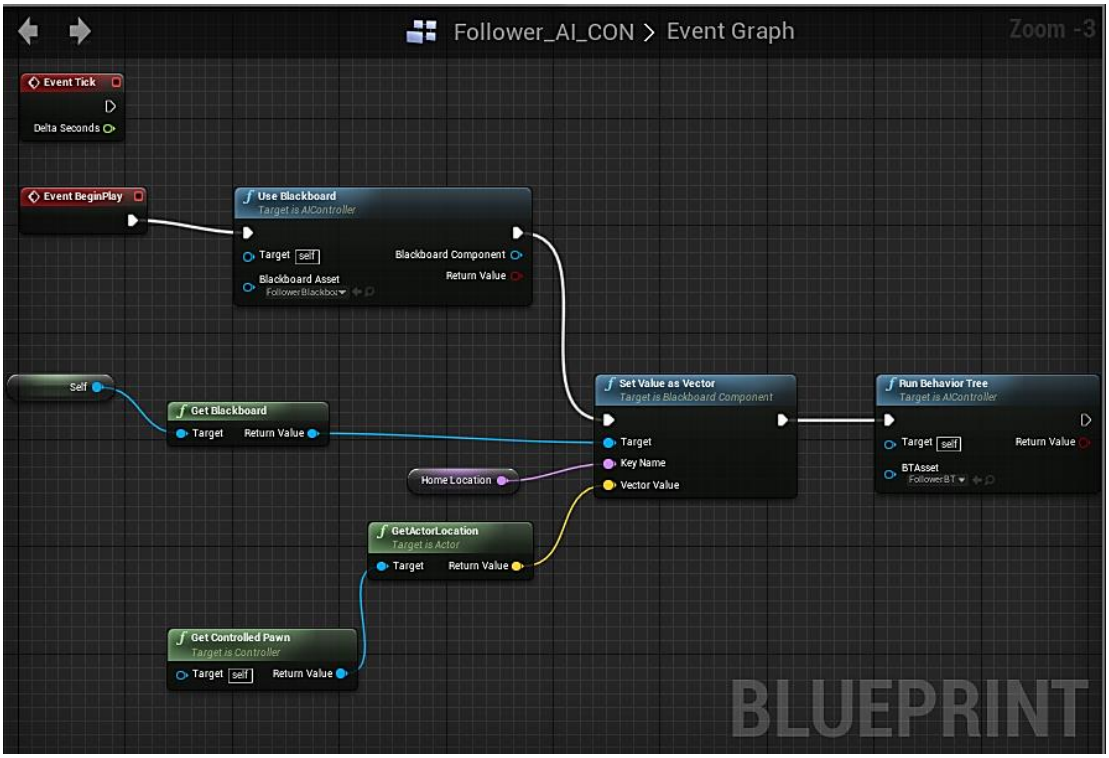

Figura 2: Exemplo de fragmento de um Event Graph

2.7 As Functions

Assim como na programação de certas linguagens de programação, como o C++, em que podemos criar funções que podem ser chamadas por outras funções ou pela execução principal do programa, existem as *Functions* que são *graphs* que podem ser chamados para serem executados, sendo eles do mesmo *Blueprint*, ou de um *Blueprint* externo. As *functions* podem ter pinos que recebem dados de entrada, e outros que apresentam dados de saída. Além disso, o sistema *Blueprint* concede variadas *functions* já implementadas, o que facilita o trabalho dos desenvolvedores.

2.8 Behavior Tree: desenvolvendo a inteligência artificial do jogo

A inteligência artificial dos atores é um componente muito importante para que em um jogo se tenha maior realismo e uma jogabilidade atraente e desafiadora, dependendo do estilo

de jogo. E para isso, na Unreal Engine é possível trabalhar com *Behaviour Trees*, que através de árvores de decisões relativa a atores específicos, é capaz de fazer com que os mesmos executam determinadas ações, ou alterações em sua constituição, de acordo com os eventos do jogo.

Partindo de um ponto inicial, denominado como *Root*, existirá uma conexão de três diferentes tipos de nós, do tipo *Composite*, sendo eles: os *Selectors*, em que causa a execução de seus nós-filho, da esquerda para direita, até que um deles obtenha sucesso de execução; os *Sequences*, em que executa seus nós-filhos, da esquerda para direita, até que um deles falhe sua execução; e os *Simple Parallel*, que executa uma única tarefa principal de uma *Behaviour Tree*.

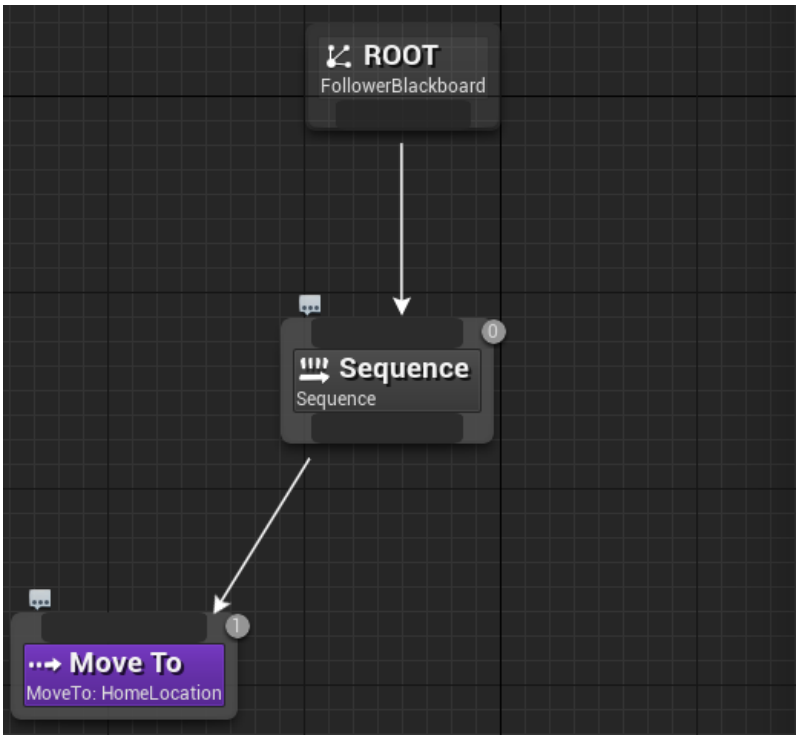

Figura 3: Amostra de um Behavior Tree com a utilização de um Composite do tipo Sequence

Vale ressaltar, que em todos os *Composites*, o início da execução dos nós-filho das mesmas, depende das condições estabelecidas neles, e essas condições são estabelecidas pelos *Decorators.* Na verificação, além de considerar os valores atribuídos internamente em seu *Event Graph*, podem também conferir os valores armazenados na memória da *Behaviour Tree*, esta que é designada como *Blackboard*, um importante componente que possibilita com que todos os nós da *Behavior Tree*, tenham conhecimento dos atributos mais significativos da arvore de decisão.

Figura 4: Decorator agregado a um Composite

Além disso, antes da execução dos filhos de um *Composite*, é possível atualizar ou verificar os dados da *Blackboard,* em *Blueprints* anexados aos *Composites*, esses que são denominados como *Services.* Considerando o jogo já em execução, muitos dos atributos da *Blackboard* devem ter seus valores modificados, e essa é a responsabilidade dos *Services*, que fica anexada nos *Composite*, no mesmo local que os *Decorators.*

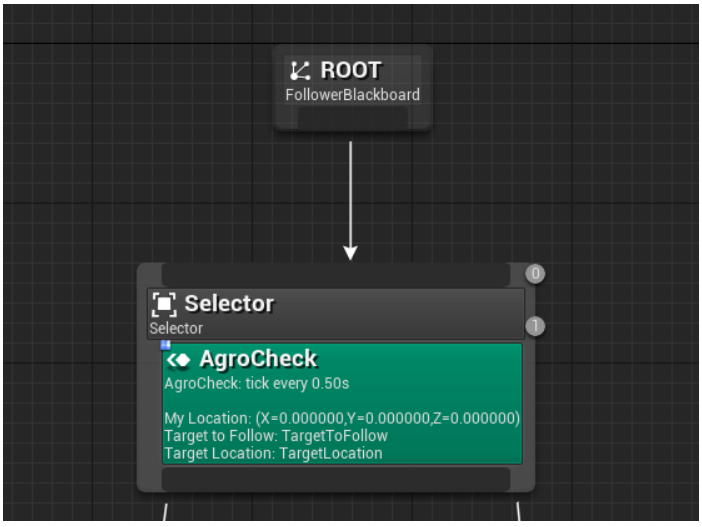

Figura 5: Demonstração de um Service anexado a um Composite

Em uma *Behavior Tree*, os filhos dos *Composites* podem ser outros *Composites,* ou também, podem ser a execução de *Blueprints,* intitulados como *Tasks*, que realizam ações com os atores que estão tendo sua inteligência artificial trabalhada pela *Behavior Tree.*

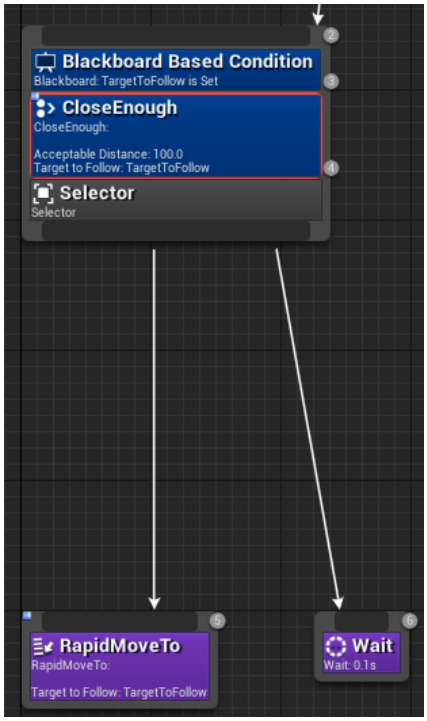

Figura 6: Amostras de Tasks ligadas a um Composite

3. Metodologia

Sendo considerado um dos gêneros de jogos mais populares entre os jogadores, os jogos de ação, que muitas vezes possui também elementos de jogos de aventura, abrange aqueles onde há um personagem principal que dentro de um cenário, confronta com inimigos, completa objetivos e recolhe itens. E esse estilo de jogo, assim como todos os outros estilos, compreende alguns aspectos comuns entre eles, e através das ferramentas da Unreal Engine, existe maneiras de desenvolver essas funcionalidades.

3.1 Patrulhamento de um NPC pelo cenário

Um dos aspectos mais frequentes em jogos de ação, é a presença do combate com NPCs inimigos, e esse confronto com o *Player Character* ocorre quando ambos se encontram, ou quando o Player Character esta no campo de visão do NPC inimigo. Entretanto, enquanto o inimigo não é confrontado, eles costumam patrulhar pelo cenário ou por um caminho especifico.

Figura 7: Execução do patrulhamento de um NPC

Para inserir a funcionalidade de patrulhamento na inteligência artificial de um NPC, na Unreal Engine, é feita através do desenvolvimento de uma *Behaviour Tree.* Nessa árvore de decisão, através de um nó *Sequence*, devem ser executados três *tasks*, sendo que dois deles não precisam ser implementados.

O primeiro deles pode ser de dois tipos diferentes e deve ser implementado, sendo que: caso o NPC deva seguir um caminho especifico, nesse *task* ele deverá alterar o valor de dois atributos da *Blackboard* que está ligado a *Behavior Tree* em questão, e esse dados são as localizações, em vetor, dos destinos de ida e volta do NPC pelo cenário; e caso esse ator deva seguir uma caminho randômico, basta que ao invés de dois atributos de destino na *Blackboard*, se tenha um único, e ele deverá receba valores aleatórios gerados pela *function GetRandomReachablePointinRadius,* em um raio de distância definido.

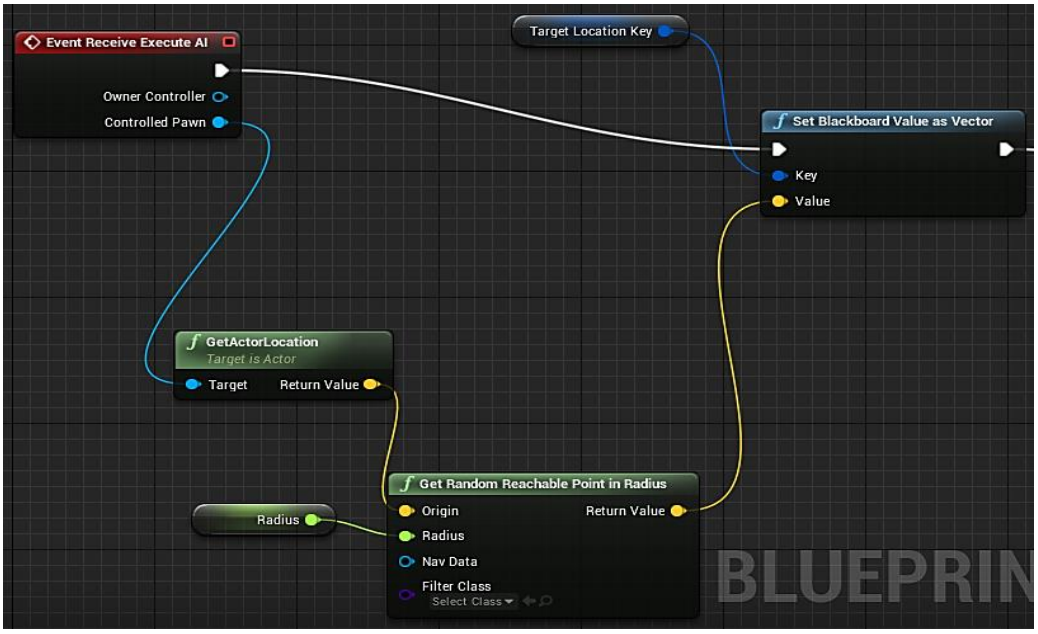

Figura 9: Amostra da utilização da function GetRandomReachablePointinRadius

Após a alteração do atributo de destino que foi criado e alocado na *Blackboard*, haverá a execução de duas *tasks* já presentes na plataforma para utilização: a *MoveTo*, que recebendo uma localização em vetor, será responsável por movimentar o NPC até o ponto estabelecido; e, em seguida, poderá ser utilizado a task *Wait*, já implementada na *engine*, que irá paralisar a execução da *Behavior Tree* por um tempo determinado, e que nesse caso, fará com que o ator fique no mesmo local por esse período, o que pode ser utilizado como uma simulação de que o NPC está analisando o seu campo de visão.

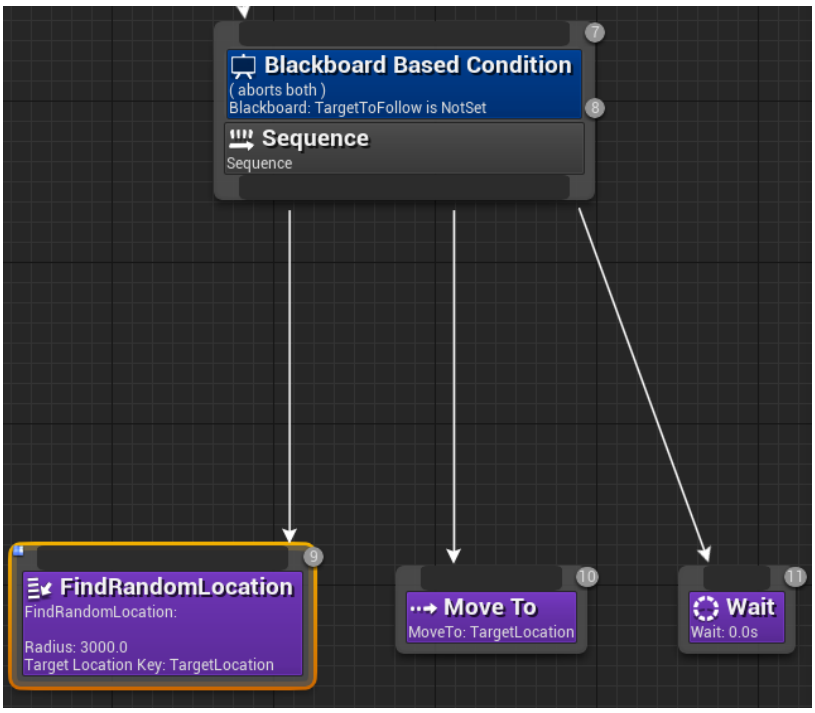

Figura 10: Fragmento do Behavior Tree que executa o patrulhamento do NPC

3.2 Detecção e Perseguição de um NPC ao personagem controlado pelo jogador

A partir do momento que o *Player Character* entra no campo de visão de um NPC inimigo, é comum, em jogos de ação, que esse adversário se movimente em direção ao personagem controlado pelo jogador, até alcançá-lo, para atacá-lo, ou também, se aproxima o bastante, para poder enfrentá-lo à distância. Além disso, existem ainda jogos em que o *Player Character* possui um NPC companheiro, e nesse caso, esse ator também perseguirá o *Player Character* a todo momento.

Figura 11: Execução da perseguição de um NPC ao Player Character

E para incluir essa funcionalidade de perseguição a um NPC inimigo, pode ser utilizado a mesma *Behavior Tree* que adicionou a funcionalidade de patrulhamento. Antes da execução do *Sequence*, que ronda o ator pelo cenário, deve ser adicionado outro *composite* ligado a ele, do tipo *Selector*, e nele é preciso ter um *Service* que será responsável por checar se o *Player Character* está no campo de visão do NPC, sendo que ao final da sua execução, deverá atualizar duas variáveis da *Blackboard*: uma que receberá a referência do *Player Character* e outra que receberá um vetor com a localização do mesmo. Deve-se considerar que, caso o *Player Character* não esteja no campo de visão do NPC inimigo, a variável que deve armazenar a referência do *Player Character* ficará vazio, e assim, a perseguição não será executada.

Para o desenvolvimento desse *decorator*, pode-se utilizar algumas *functions* já presentes no sistema *Blueprint*, que trazem um resultado bastante eficaz. A primeira delas é a *MultisphereTraceofObjects*, que utilizando: uma localização inicial, em vetor, que no caso é a posição do próprio NPC; um ponto final em vetor, que seria a posição do NPC somado a uma distância definida no seu eixo Z; um vetor com os tipos de objetos que se tem interesse em buscar, que nesse caso são do tipo *Pawn*; um vetor com a referência dos atores que devem ser ignorados na busca, sendo então os outros NPC inimigos; e um valor estabelecido para raio. Através desses atributos, essa *function* executa um “radar” de círculos, com o valor de raio determinado, em volta do NPC, sendo que o ponto de busca desse radar se inicia na localização do NPC e termina no ponto final estabelecido. E como resultado da busca desse radar, é gerado um vetor com a referência de todos os objetos que foram identificados nessa busca, sendo ignorados aqueles que também são

NPC inimigos ou que não seja do tipo *Pawn,* ou seja, caso haja apenas um *Player Character* no cenário, esse vetor só terá a referência dele.

Figura 12: Representação gráfica do campo de busca gerado pelo MultisphereTraceofObject, e a detecção do Player Character, causando uma mudança na sua coloração

Após ser executada a *function MultisphereTraceofObjects*, deverá ser feito uma verificação se o *Player Character* está no campo de visão do *NPC*, e para isso será utilizado um conceito denominado como *Field of View*, em que se realiza um cálculo para estabelecer qual o campo de visão de um ator. Nesse cálculo, em primeiro lugar, é necessário obter o valor do eixo X da posição do *NPC*, e também, calcular o valor da distância entre o *NPC* e o *Player Character* encontrado, em seguida, é preciso normalizar os dois valores obtidos, fazendo com que seu comprimento fique unitário, mas mantendo a mesma direção, e assim, possibilitar o cálculo do produto escalar de ambos os valores. Essa operação resultará no cosseno do ângulo entre os dois vetores, e calculando o arco do cosseno desse valor, será obtido o ângulo entre a frente do NPC e a posição do Player Character. Com esse valor, basta então verificar se ele é superior a 60º, ou outro valor que se deseja estabelecer como ângulo do campo de visão do NPC..

Se o valor não superar o ângulo de campo de visão estabelecido, então a variável da *Blackboard* que armazena a referência de ator será anulada, caso contrário, haverá a execução de mais uma *function*, com o valor de referência do *Player Character* encontrado. Como há possibilidade de que o personagem controlado pelo jogador esteja atrás de algum objeto que impediria o NPC de avistá-lo, através da *function LineTraceByChannel*, é lançado uma linha em direção do *Player Character*, e como retorno, esse *function* oferece a referência do primeiro objeto que se colidiu com a linha. Através desse dado, é averiguado se a referência do ator encontrado é a do *Player Character*, sendo que em caso positivo, é alterado duas variáveis da *Blackboard,* sendo

elas: o atributo que armazena a referência do ator que o NPC deve seguir, o que seria a referência do *Player Character*; e outra variável, com o vetor da localização dele naquele momento que foi executado o *Event Graph*.

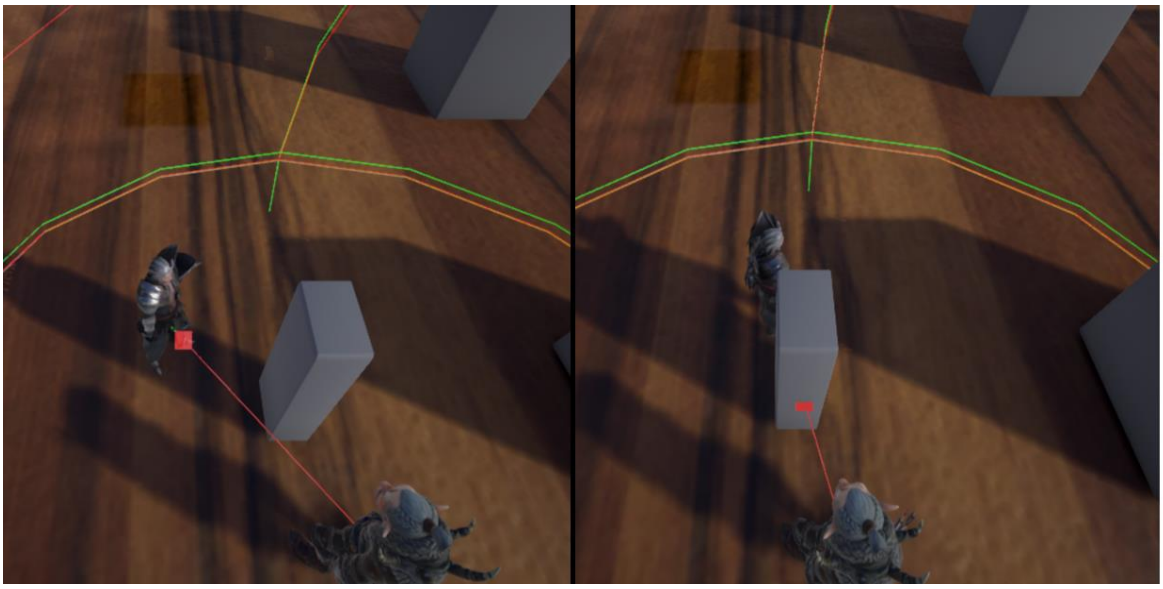

Figura 13: Representação gráfica das linhas geradas pelo LineTraceByChannel, para a detecção de obstaculo, a frente do Player Character

A partir desse *Service* será possível checar se o *Player Character* está no campo de visão do NPC inimigo, e então executar uma condicional se ele deve, ou não, começar a perseguir o *Player Character*. Para isto, o *selector* em que esse *service* está instalado, deverá ter um selector e um sequence instalado nele, de modo que ambos deverão ter um *decorator* denominado como *Blackboard Based Condition*. Ele irá verificar se a variável da *Blackboard,* que referencia um ator que o NPC deve seguir, está nula, sendo que em caso positivo, será executado o *sequence* responsável pelo patrulhamento do *NPC*, e caso contrário, haverá a execução do *selector* encarregado de fazer com que o *NPC* siga o *Player Character*. Além de ter esse *decorator* condicional, esse *selector* também deverá ter outro que irá verificar se o NPC já está perto o suficiente do *Player Character*. Após a verificação dos dois *decorators* instalados nele, esse *selector* poderá executar um *task* muito semelhante com o *MoveTo*, entretanto, ao invés de ter como atributo, uma localização de destino, ele irá receber o *Pawn* do *Player Character*, para que a perseguição seja constante.

4. Resultados e Análise

Um dos elementos mais comumente presente em jogos de ação é o combate com inimigos, e para isso é necessário adicionar funcionalidades a *gameplay* do *Player Character*, e suas devidas animações, que simularam a realização de um ataque.

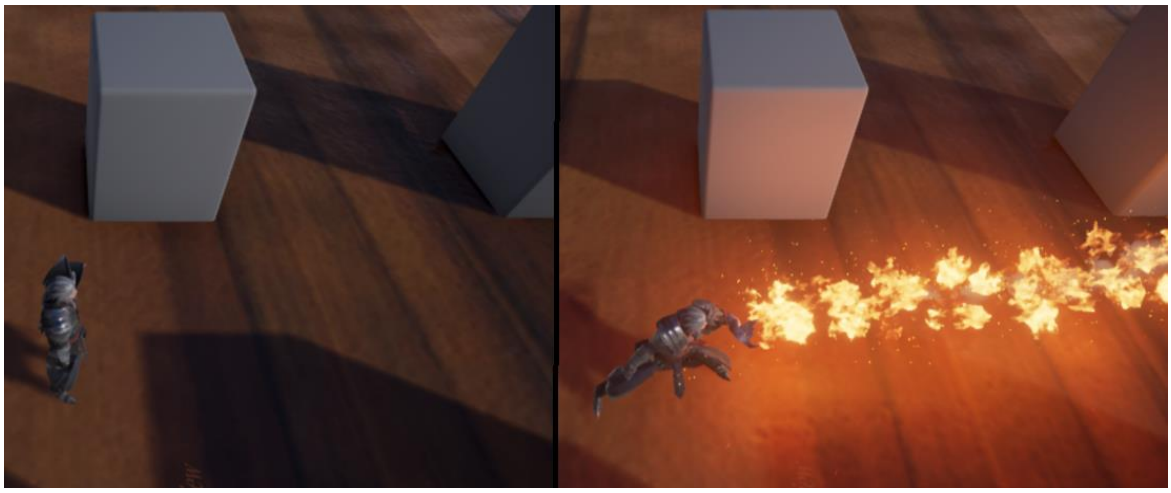

Figura 14: Amostra da execução de um ataque

Para a implementação desse recurso, é preciso, em primeiro lugar, alterar o *Blueprint* responsável por configurar os comandos do jogador. Nele, é possível incluir um *event* que irá iniciar uma conexão de nós, quando o comando do controlador, correspondente ao *event,* ser pressionado. Com essa conexão de nós, será verificado uma variável booleana que notifica se um ataque está sendo executado, e em caso negativo, irá imobilizar NPC, caso ele esteja em estado de movimentação, e alterará essa variável para *true*, o que poderá executar também um *spawn* de um ator, como uma bola de fogo, por exemplo para simular o ataque, ou efeito do mesmo, se for necessário.

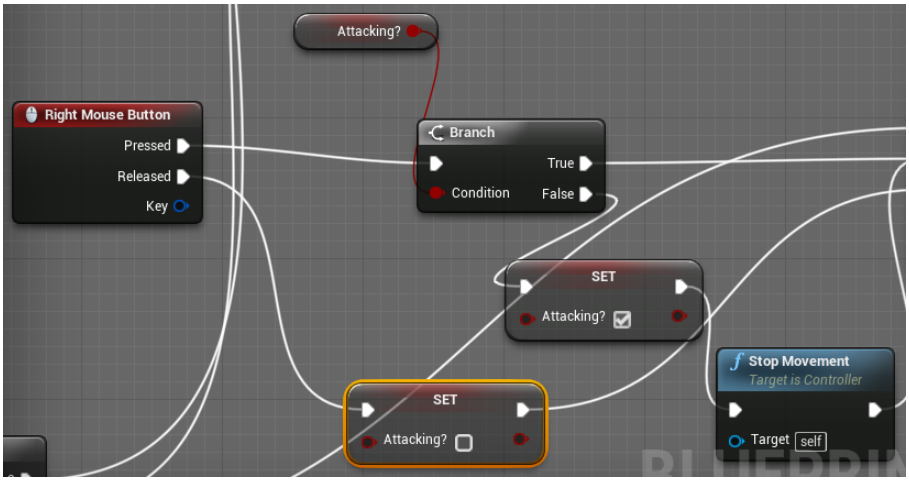

Figura 15: Exemplo de fragmento de um Event Graph que inicia a execução de um ataque através do botão direito do mouse

E no Event Graph correspondente a animação do Player Character, será criado e verificado, a cada execução, uma outra variável booleana que representa se um ataque deve, ou não, ter sua execução iniciada, e caso o valor seja false, será importado o atributo que representa se um ataque está em execução do Blueprint responsável por configurar os comandos do jogador, e verificado seu valor. Caso esse atributo esteja com valor true, a variável booleana que corresponde se um ataque deve, ou não, ter sua execução iniciada, receberá o valor true, e após o tempo necessário para a execução da animação passar, através de uma function de delay, ela receberá um valor false.

Além dos *Event Graph*, há também outros tipos de *Graph* que é possível trabalhar, sendo um deles, o *Anim Graph*, que será o graph necessário para implantar a animação de ataque. Nele é possível atribuir as variáveis booleanas que informam se uma animação deve ser executada, e nesse caso, deverá ser incluído a certificação de que a variável booleana que corresponde se um ataque deve ser executado está com valor *true*, considerando que desse modo, ele irá iniciar a animação de ataque do *Player Character*.

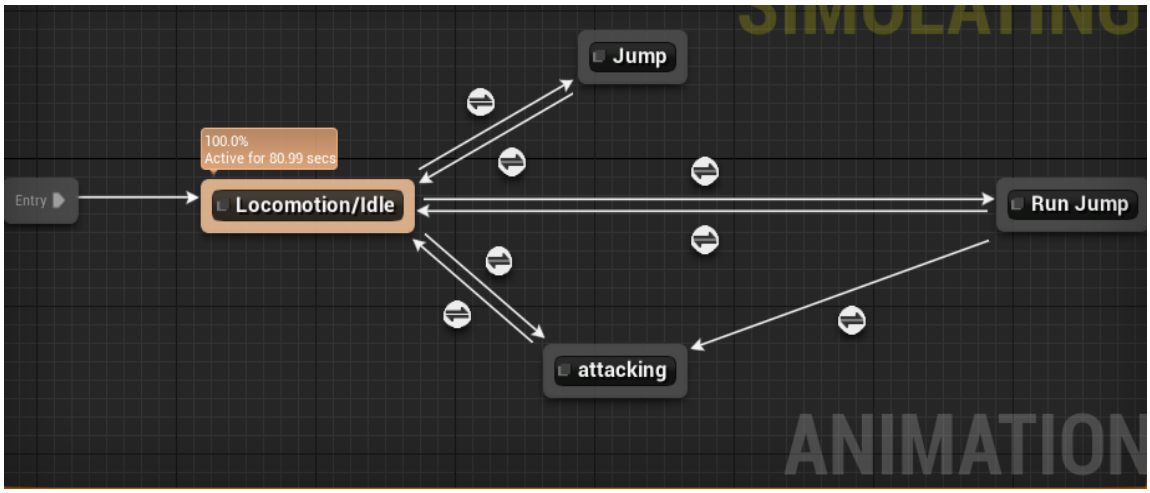

Figura 16: Exemplo de Anim Graph

5 Conclusões e Trabalhos Futuros

Atualmente, a Unreal Engine, em sua quarta geração, é uma das mais sofisticadas e avançadas *engines* presente no mercado de games, e agora que sua utilização, sem pretensões comerciais, pode ser feita de forma gratuita, está possibilitando que um público maior possa estuda-la, e utilizar recursos avançadas no desenvolvimento de seus projetos.

Ao final desse trabalho, foram alcançados os elementos básicos presentes em jogos do gênero ação, tanto na parte da inteligência artificial de NPCs, como na configuração do controlador e na animação do *Player Character*. O sistema de *scripting Blueprint* apresenta uma dificuldade inferior da programação em C++, que é a utilizada pela Unreal Engine,

apresentando muitas *functions* já implementadas e seu aspecto visual, o que facilita muito na inserção de avançadas funcionalidades em projetos de jogos. E por conta da possibilidade de depuração, a correção de erros é simplificada.

E para trabalhar com a inteligência artificial e animação dos atores, as *Behavior Trees* possibilitam com que se implemente e visualize mais facilmente, o modo como ela irá executar, tirar a complexidade que seria, caso fosse totalmente implementada em C++.

Considerando que o diferencial de um jogo em relação aos outros, do mesmo gênero, é o seu aspecto peculiar, poder trabalhar com os recursos mais avançados presentes na indústria atual de jogos torna isso viável. Por isso, para dar continuidade a esse trabalho realizado, existem três possibilidade: a criação de diferentes *levels*, que alternariam entre eles, pelos NPC inimigos presentes e suas dificuldades de confronto, o que tornaria necessário trabalhar mais na inteligência artificial dos mesmos, adicionando variadas funcionalidades na *Behavior Tree* deles; com o desenvolvimento de um mundo aberto, com muitos elemento, dentre eles, NPCs inimigos e obstáculos para o *Player Character* ultrapassar, o que traria desafios na área do processamento do ambiente e implementação da modelagem do cenário, além de ter que trabalhar na inteligência artificial dos NPCs; e trabalhar na implementação de dispositivos de realidade virtual, como por exemplo, um óculos VR e controladores baseados na movimentação do jogador.

6 REFERÊNCIAS

80 Level, Unreal Engine 4 Stylized Rendering Workflow. Disponível em: <https://80.lv/articles/unreal-engine-4-stylized-rendering-workflow/>. Acesso em 10.Junho.17.

Ação, RPG, Plataforma? Saiba mais sobre os gêneros de jogos e quais games se classificam nos mais conhecidos - Parte 1 Disponível em: <http://www.nintendoblast.com.br/2012/04/gamedev-conheca-os-tipos-de-jogos.html> Acesso em 15 Junho 17.

Applications of the Vector Dot Product for Game Programming. . Disponível em: <https://hackernoon.com/applications-of-the-vector-dot-product-for-game-programming-12443ac91f16> Acesso em 29 Junho 17.

Carnal, Benjamin. Unreal Engine 4.X By Example. Packt Publishing , Julho, 2016

Creative Bloq, 25 tips for Unreal Engine 4. Disponível em: <http://www.creativebloq.com/3d/25-tips-unreal-engine-4-71621196>. Acesso em 12. Junho. 17.

Emperore, Katax. Unreal Engine Physics Essentials. Packt Publishing , Setembro, 2015

Linear algebra for game developers ~ part 2. Disponível em: <http://blog.wolfire.com/2009/07/linear-algebra-for-game-developers-part-2/> Acesso em 25 Junho 17.

Nvidia Developer, Unreal Engine.. Disponível em:<https://developer.nvidia.com/unrealengine>. Acesso em 10.Fevereiro.17.

Orfeasel, Creating a basic Patrol AI. Diponível em: <http://orfeasel.com/creating-a-basic-patrol-ai/>Acesso em 20 Junho 17.

PhysX Info, PhysX SDK 3.x. Disponível em: <http://physxinfo.com/wiki/PhysX_SDK_3.x>. Acesso em 03. Junho. 17.

Slideshare, Behavior Tree in Unreal engine 4. Disponível em: <https://pt.slideshare.net/JaeWanPark2/behavior-tree-in-unreal-engine-4>. Acesso em 13. Junho. 17.

Tom Looman, Getting started with Unreal Engine 4. Disponível em: <http://www.tomlooman.com/getting-started-with-unreal-engine-4//>. Acesso em 14. Fevereiro. 17.

Unreal Engine, BluePrints Visual Scripting,. Disponível em: <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/index.html>. Acesso em 19. Fevereiro. 17.

Victor Phellipe, Iluminação Global. Disponível em: <https://victorphellipe.wordpress.com/2010/11/13/iluminacao-global-2/>. Acesso em 12. Junho. 17.

Contatos: henrique.rachti@gmail.com e e-mail orientador: luciano.silva@mackenzie.br