SOCCER WARFARE

Mestrado em Engenharia Informática – Computação Móvel

Rui Miguel dos Santos Fernandes

Leiria, setembro de 2021

SOCCER WARFARE

Mestrado em Engenharia Informática – Computação Móvel

Rui Miguel dos Santos Fernandes

Project Report under the supervision of Professor Gustavo Miguel Jorge dos Reis

Leiria, setembro de 2021

Agradecimentos

Gostaria de agradecer ao professor Doutor Gustavo Reis por ter aceite orientar me neste projeto autoproposto e por me ter orientado ao longo do desenvolvimento até à entrega. Ao professor Filipe Gonçalves por ter ajudado com a parte visual do projeto, ao professor Ricardo Antunes por ter feito críticas construtivas, e também ao meu agregado familiar por me ter apoiado.

Originalidade e Direitos de Autor

Este projeto é original, realizado apenas para este propósito. Todos os autores cujo os estudos e publicações foram utilizadas são devidamente reconhecidos.

Abstract

Currently, video games are the highest revenue generators from entertainment, with two big examples being Fortnite (developed in Unreal Engine 4) and Grand Theft Auto V. This income model is adopted by mobile games very often.

This project consists on the development of a videogame using Unreal Engine 4 (UE4) and achieving network multiplayer synchronization. This game can simply be described as a soccer game with guns and abilities, featuring local play and network multiplayer play. The implemented game also features artificial intelligence for the non-player controlled units. It is programmed in C++ and Blueprints, which is the visual scripting language of UE4. Its basic elements are: *nodes*, *events*, *functions*, *parameters* and others.

The biggest hurdles while developing this project were to learn UE4 and how to synchronize everything properly in the networking part. It started off as a search of the inner workings of UE4, followed by adding simple components and finally more complicated ones. First, adding the characters, the ball, the camera, animations, abilities, the Artificial Intelligence (AI), the game rules and then, network multiplayer and local multiplayer. Most objects were created with a C++ base class, then a Blueprint being created inheriting the corresponding class, so that the structure was made in C++ and only the visual part and defaults were to be adjusted for the corresponding Blueprint. All goals set for this project were completed.

In conclusion, much was learned during the development of this project, in regards to how Unreal Engine 4 works, what is needed to develop a game using the engine and also, additional knowledge in game development.

Keywords: Unreal Engine 4, Video games, Blueprints, Artificial Language, Network Multiplayer, C++

Resumo

Atualmente, os videojogos são os maiores geradores de receitas de entretenimento, com dois grandes exemplos sendo o Fortnite (desenvolvido em Unreal Engine 4), e o Grand Theft Auto V. Ambos apresentam compras dentro do jogo, este modelo de rendimento é adotado muito frequentemente por jogos mobile.

Este projeto consiste no desenvolvimento de um jogo utilizando o Unreal Engine 4 (UE4) e alcançar sincronização em rede. O jogo elaborado pode ser simplesmente descrito como um jogo de futebol com armas e habilidades, contendo multijogador local e multijogador em rede. O jogo implementado também apresenta inteligência artificial para as unidades não controladas pelo jogador. É programado em C++ e Blueprints que se trata da linguagem de Visual Scripting do UE4. Os elementos básicos desta são: *nodes*, *events*, *functions*, *parameters*, entre outros.

Os maiores obstáculos durante o desenvolvimento deste projeto foram aprender UE4 e como sincronizar tudo corretamente na parte da rede. O projeto começou como uma pesquisa sobre o funcionamento interno do UE4, e seguido da adição de componentes simples e depois dos mais complicados. Adicionando primeiro os personagens, a bola, a câmara, animações, habilidades, a Inteligência Artificial (IA), as regras de jogo e finalmente multijogador em rede e multijogador local. A maioria dos objetos foram criados tendo como base uma classe C++, sendo posteriormente criada uma blueprint que herda da classe correspondente em C++, de forma a que haja uma estrutura base feita em C++ e seja só necessária a alteração do aspeto visual e os valores *default* no lado da Blueprint correspondente. Todos os objetivos definidos neste projeto foram concluídos.

Em conclusão, muito foi aprendido durante o desenvolvimento deste projeto, no que diz respeito ao funcionamento do Unreal Engine 4, ao necessário para executar um jogo no Engine e conhecimento adicional na área do desenvolvimento dos videojogos.

Palavras-chave: Unreal Engine 4, Video games, Blueprints, Artificial Language, Network Multiplayer, C++

Índice

Lista de Figuras

Lista de Tabelas

Lista de Abreviações e Acrónimos

AI/IA	Artificial Intelligence/Inteligência Artificial
BT	Behavior Tree
ESTG	School of Technology and Management
EQS	Environment Query System
FPS	Frames Per Second
GDD	Game Design Document
HUD	Heads Up Display
HZ	Hertz
RPC	Remote Procedure Call
TF	Trust Factor
UE4	Unreal Engine 4
UI	User Interface
VSync	Vertical Sync

1. Introdução

Este projeto realiza-se no âmbito da disciplina de projeto do Mestrado de Computação Móvel em Engenharia Informática. Foi auto proposto com o objetivo de aprender a desenvolver um jogo no motor de jogo Unreal Engine 4, sendo um dos motores de jogos mais utilizados na indústria dos videojogos.

Este projeto trata-se de desenvolver um jogo multiobjectivo nas condições de vitória realizado totalmente em Unreal Engine 4. É um jogo no formato 1 contra 1, que utiliza inteligência artificial. Este jogo assemelha-se a um jogo de futebol, com a adição de armas e habilidades especiais: 2 equipas compostas por 6 elementos competem pela vitória enquanto lidam com a existência de 2 unidades adicionadas no campo de jogo: o árbitro e o jardineiro. Cada jogador pode tomar posse de qualquer unidade da sua equipa a qualquer momento. As restantes são controladas por inteligência artificial.

A temática em que este projeto se enquadra é relevante pois tenho interesse particular no desenvolvimento de jogos, inteligência artificial e sincronização em rede. Atualmente os videojogos são o meio mais lucrativo no que toca a entretenimento. Os prémios de eSports estão cada vez a ficar mais altos e cada vez a ganhar mais popularidade.

Os objetivos principais deste projeto são:

- Aprendizagem do funcionamento básico do UE4;
- Execução do Game Design Document desenvolvido previamente;
- Aprendizagem em sincronização de jogos multijogador em rede.

O desenvolvimento e aprendizagem foram feitas empiricamente com suporte de documentação.

Os testes de elementos adicionados e interações foram testados em jogo, jogando o jogo, em multijogador local e multijogador em rede. Esta abordagem é conhecida como playtesting, tendo o benefício de descoberta de bugs e problemas de jogabilidade.

Este relatório tem a seguinte estrutura: no capítulo 2 é abordado o Game Design Document que irá declarar as especificações do jogo; no capítulo 3 é abordada a arquitetura base e funcionamento do Unreal Engine 4; no capítulo 4 é abordada a implementação do projeto; no capítulo 5 é explicada a inteligência artificial; no capítulo 6 e 7 é abordada a implementação do multijogador local e em rede; e no capítulo 8 é abordado o Niagara Particle System.

2. Resumo do GDD

No desenvolvimento de jogos é comum a realização do Game Design Document (GDD), este é feito pelo Game Designer. O GDD trata-se de um documento altamente descritivo com as especificações do jogo. Contém história, personagens, regras de jogo, mecânicas, concept art e qualquer outra informação necessária para a realização da ideia.

2.1. Gênero do jogo

Este jogo trata-se de um jogo de futebol 6v6 (2 equipas) com habilidades, para 2 jogadores em 1vs1 com limite de tempo de 5 minutos.

Os jogadores podem controlar um dos seis elementos da sua equipa à vez, podendo alternar o controlo entre estes. As unidades, enquanto não controladas pelo jogador, são controlados pela inteligência artificial.

2.2. Objetivos do jogo

O jogo tem um objetivo principal: marcar mais pontos que o adversário até ao final do tempo. Caso a bola seja destruída, ou caso o árbitro morra, o objetivo do jogo muda para manter vivos mais elementos de equipa que o adversário até o final do tempo.

2.3. Personagens

O jogo tem dois tipos de personagens: as personagens jogáveis e as não jogáveis.

2.3.1. Personagens Jogáveis

Estas personagens podem movimentar-se, saltar, chutar, disparar e utilizar a sua habilidade especial. Possuem 30 balas, 100 pontos de vida e 100 pontos de energia, com regeneração passiva:

- 1 de vida por segundo;
- 1 de energia por segundo se a bola estiver na posse da equipa.

Um elemento da equipa, ao utilizar a sua habilidade especial, são-lhe consumidos pontos de energia. Existem 4 *roles* (papel do personagem na equipa) com as suas habilidades, sendo as seguintes:

- DPS – Bola de fogo, consome 25 de energia, causa 25 de dano aos inimigos;
- Mage – Feixe, consome 50 de energia, causa 40 de dano aos inimigos, dura 1 segundo;
- Healer – Bomba de cura, consome 50 de energia, causa 30 de cura aos aliados em proximidade (cura também envenenamento), dura 1.6 segundos;
- Tank – Barreira, consome 5 de energia, dura 5 segundos, possui 15 pontos de vida.

2.3.2. Não Jogáveis

Estas personagens não podem ser controladas pelos jogadores.

Árbitro

Unidade controlável por inteligência artificial, pode movimentar-se e disparar, irá seguir a bola e retaliará a ataques, possui 200 pontos de vida e 100 balas.

Jardineiro

Unidade controlável por inteligência artificial, pode movimentar-se e disparar, irá seguir a bola e retaliará a ataques, possui 200 pontos de vida e 250 balas.

Após morrer causará uma explosão em área que mata instantaneamente quaisquer unidades na proximidade, irá envenenar os restantes, causa 1% da vida máxima em dano por segundo.

2.4.Itens

Este jogo tem vários itens de jogabilidade e de objetivos do jogo.

2.4.1. Bola

A bola pode ser controlada por elementos das equipas, possui 400 pontos de vida, 2 estados:

- Neutro (Branco)
- Equipa (Vermelho/Azul) – fornece 1 ponto de energia por segundo à equipa

O estado é mudado quando um elemento de equipa toca na bola.

2.4.2. Áreas de marcação

Existem 2 áreas de marcação, uma para cada equipa, a bola se estiver sobre a posse da equipa oposta e tocar na área de marcação (não há autogolos), adicionará 1 ponto à equipa e fará com que a bola regresse à posição inicial após 1 segundo.

2.4.3. Balas e caixa de balas

As balas causam 20 de dano. As unidades, após morrerem, largam uma caixa de balas que contém o número das suas balas restantes. Esta caixa pode ser apanhada por outras unidades.

2.5.Mapa

O mapa é de formato retangular com 2 áreas de marcação.

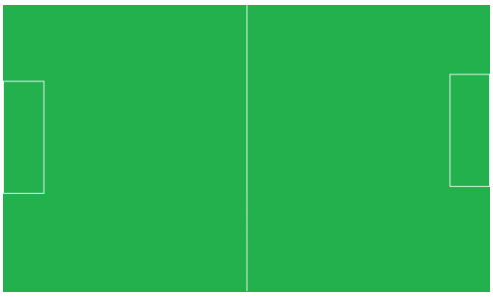

Figura 1 - Esboço do campo

2.5.1. Câmara

A câmara tem vista aérea sobre o campo, segue a bola e, quando esta for destruída, tentará seguir a maioria dos jogadores.

2.6. Controlos

Para o controlo das unidades existem os seguintes controlos:

Figura 2 - Controlos

2.7. Fluxo do multijogador

Os jogadores podem jogar localmente, ou em rede através da plataforma Steam[1].

[1] Steam – Plataforma online de distribuição de videojogos

2.7.1. Fluxo local

No menu principal, um dos jogadores carrega no botão de jogo local e procede à cena de escolha de *roles*.

Ambos os jogadores podem interagir e escolher os *roles* da sua equipa, esta fase tem duração de 15 segundos.

Passado o tempo, começará o jogo e quando terminado, demonstrará o resultado por 10 segundos e regressará ao menu principal.

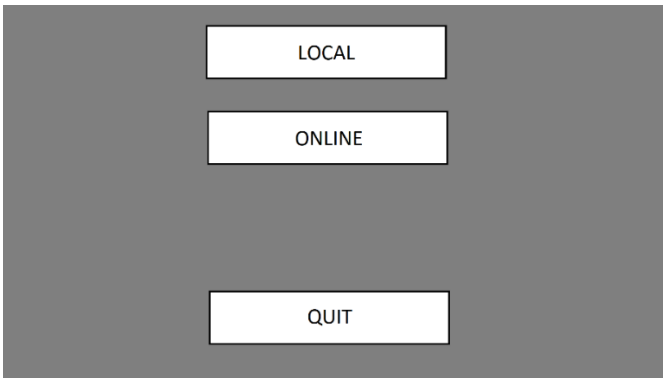

Figura 3 - Esboço do menu principal

2.7.2. Fluxo online

Para este fluxo é necessária a plataforma Steam (plataforma online de distribuição de videojogos por via digital).

No menu principal, um dos jogadores carrega no botão de jogo em rede, cria uma sessão e procede à cena de escolha de *roles*. O segundo jogador, carrega no botão de jogo em rede, junta-se à sessão e procede à cena de escolha de *roles*.

Estando ambos os jogadores na mesma cena, inicia-se o temporizador, estes podem interagir e escolher os *roles* da sua equipa, esta fase tem duração de 15 segundos.

Passado o tempo, começará o jogo e quando terminado, demonstrará o resultado por 10 segundos e regressará ao menu principal.

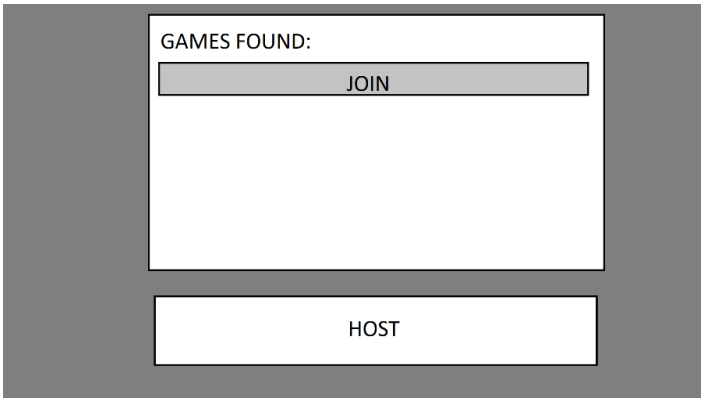

Figura 4 - Esboço do menu do multijogador online

3. Unreal Engine 4

O Unreal Engine é um motor de jogo desenvolvido pela Epic Games, com a sua primeira demonstração em 1998 no jogo de tiros em primeira pessoa Unreal.

O Unreal Engine está atualmente na sua 5ª versão, pouco depois do ínicio da 9ª geração de consolas. Para além de jogos de qualquer género pode também ser utilizado para realização cinematográfica, como é o caso da série “The Mandalorian”.

Existem algumas alternativas de motores de jogo disponíveis, sendo o Unreal Engine, o Unity, o Cry Engine, o Lumberyard e o Godot os mais populares. Foi escolhido o Unreal Engine 4 para o desenvolvimento por ser o mais completo em termos de inteligência artificial, compatibilidade de código multijogador em rede com multijogador local e blueprints.

3.1. Arquitetura Base

Para desenvolvimento de jogos, o Unreal Engine disponibiliza algumas classes base para facilitar o desenvolvimento. Estas podem ser herdadas por blueprints ou por outras classes C++.

A Gameplay Framework é constituida pelas seguintes classes base (herdam da classe Actor):

- `Pawn;`
- `Character;`
- `Controller;`
- `PlayerController;`
- `AIController;`
- `Camera.`

Para gestão de jogo, as classes base utilizadas são as seguintes:

- `GameMode;`
- `GameState;`
- `PlayerState.`

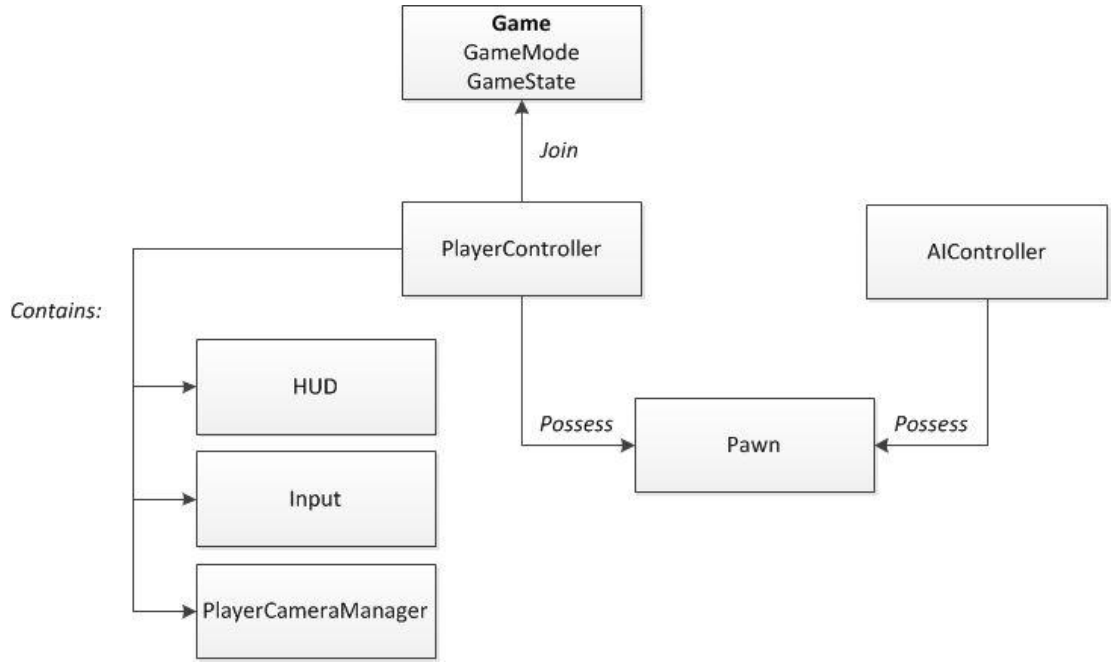

Figura 5 – Relações de classes da Gameplay Framework

Actor

Os objetos da classe `Actor` herdam da classe Object e podem ser colocados no nível. Não podem ser possuídos (isto será abordado no capítulo de inteligência artificial).

Imaginemos um jogo de plataforma: as moedas a colecionar podem herdar da classe `Actor`.

Pawn

Os objetos da classe `Pawn` herdam da classe Actor com a principal diferença que podem ser possuídos por um *controller*. Geralmente utilizado para unidades controladas por AI, ou utilizado para o personagem controlado pelo humano.

Character

Os objetos da classe Character herdam da classe `Pawn`. Já vem com o movimento e colisões preparadas. Pode ser considerado como *template* de personagem para não ter de ser reinventado.

Este objeto para além de conter a funcionalidade da classe `Pawn`, contém também *Character Movement*, *Skeletal Mesh* e *Capsule Component*, implementação básica de networking e modelos de input. Destina-se para personagens orientados verticalmente que podem andar, correr, saltar, agachar, voar e nadar. O componente *Skeletal Mesh* está responsável por adicionar animações a meshs com esqueleto humanoide.

Controller e PlayerController

A classe `Controller` tem como finalidade possuir *Pawns*.

A classe `PlayerController` serve de interface entre o jogador e o *Pawn*. Herda da classe `Controller`. O `PlayerController` implementa funcionalidades ao receber *input* do jogador e traduzindo-o para ações que o *Pawn* executará.

O `PlayerController` possui um `ViewTarget`, ou seja, a câmara de jogo para o utilizador que está sobre o domínio do `PlayerController`. Ao possuir um `Pawn` este irá alterar o seu `ViewTarget` para a câmara presente no `Pawn`. Caso esta se não exista, será criada uma câmara temporária para ser utilizada pelo `PlayerController`.

3.2.Blueprints

O Unreal Engine dispõe de uma grande ferramenta de desenvolvimento: as Blueprints. Estas permitem o desenvolvimento de jogos inteiros sem ser necessária a escrita de uma linha única de código. São ótimas para protótipos ou para utilizadores sem conhecimentos da linguagem C++, devido à sua facilidade de leitura, visualização e utilização de linguagem corrente.

Uma Blueprint é comparável a uma classe: pode conter métodos e atributos. O fluxo de um método está representado por linhas de fluxo, sendo fácil seguir o seu fluxo de execução.

Existe também o Material Editor que não é blueprint necessariamente, mas é bastante semelhante em termos de nós e ligações, este serve para a criação de materiais (*shaders).*

3.2.1. Eventos

Os eventos são nós que são chamados por código de jogo, e que irão executar ações após serem disparados (triggered).

Diferenças entre Eventos e Funções

Funções:

- Não podem manipular tempo;
- Podem conter variáveis locais;
- Podem retornar valores.

Eventos:

- Podem ser replicados (multijogador, será explicado no capítulo dedicado à rede);
- São assíncronos.

A figura seguinte ilustra um exemplo simples de um evento, este incrementa a variável counter.

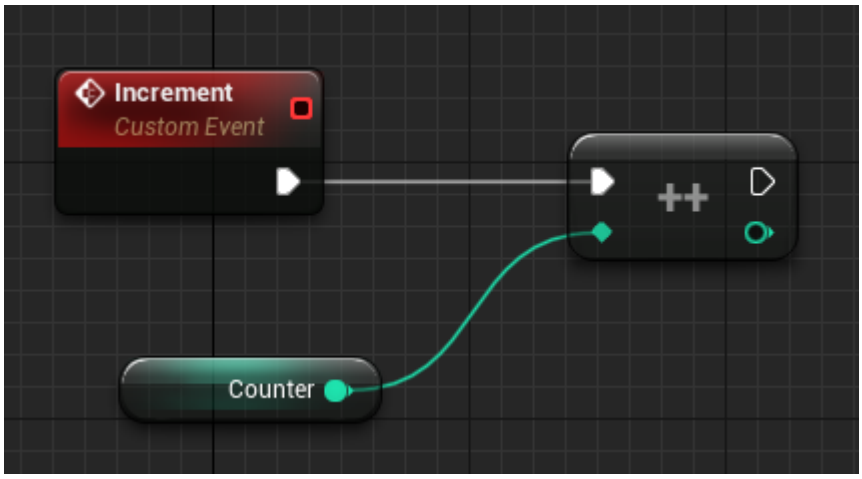

Figura 6 - Evento que incrementa uma variável

A figura abaixo ilustra um exemplo de função que recebe 2 floats e devolve o resultado da sua multiplicação.

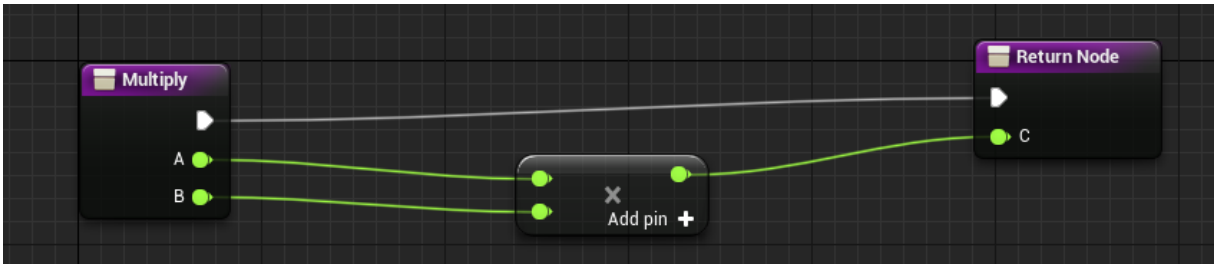

Figura 7 - Função que multiplica 2 floats

A figura abaixo ilustra a função IsClosestToBall, que verifica se um determinado jogador (Player) é o mais próximo da bola. Esta função retorna um valor booleano.

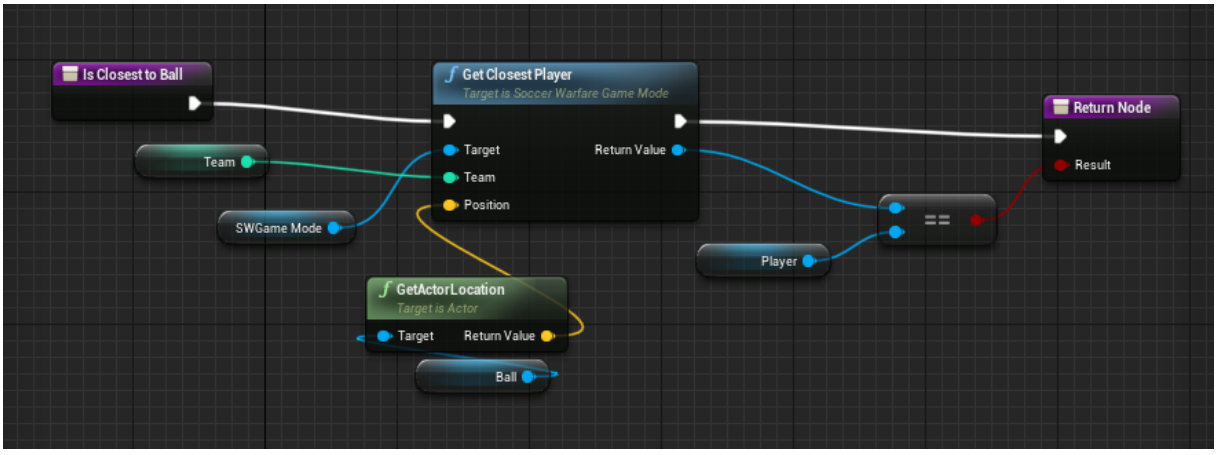

Figura 8 - Função a retornar um booleano

3.2.2. Objetos Base

Os atores são constituídos por componentes. Seguindo a analogia de um carro, este é composto pelas rodas, volante, motor, portas, chassis, e outros componentes.

Pode se verificar na figura abaixo um objeto, a BallMesh que contém o componente filho CameraArm e esse contém o componente filho Camera.

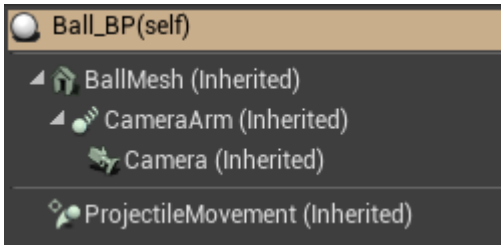

Figura 9 - Ball Blueprint

O componente BallMesh é uma *Static Mesh*, este componente é um modelo ou malha 3D.

O componente CameraArm é um *Spring Arm Component*. Este componente serve para controlar automaticamente como a câmara lida com situações em que ficaria obstruída (por exemplo se surgir um objeto entre a câmara e a bola).

```
BallMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("BallMesh"));
RootComponent = BallMesh;
CameraArm = CreateDefaultSubobject<USpringArmComponent>("CameraSpringArm");
CameraArm->SetupAttachment(RootComponent);
Camera = CreateDefaultSubobject<UCameraComponent>("Camera");
Camera->SetupAttachment(CameraArm);

ProjectileMovement = CreateDefaultSubobject<UProjectileMovementComponent>("Projectile Movement");
```

Figura 10 - Alteração do RootComponent na classe da bola

3.2.3. Gestão de Estado de Jogo

A gestão do jogo é efetuada através das classes descritas em seguida.

GameMode

Esta classe é responsável por lidar com a lógica do estado de jogo, regras de jogo, número de jogadores e transição entre níveis.

Neste jogo foi criada a classe `SoccerWarfareGameMode` com o objetivo de interagir com a classe `SoccerWarfareGameState` e iniciar e terminar o jogo. A classe `SoccerWarfareGameMode` informa a classe `SoccerWarfareGameState` sempre que algum ponto é marcado, alguma unidade morre, a bola é destruída, e sempre que há alteração tempo restante do jogo e na gestão de posse de jogadores.

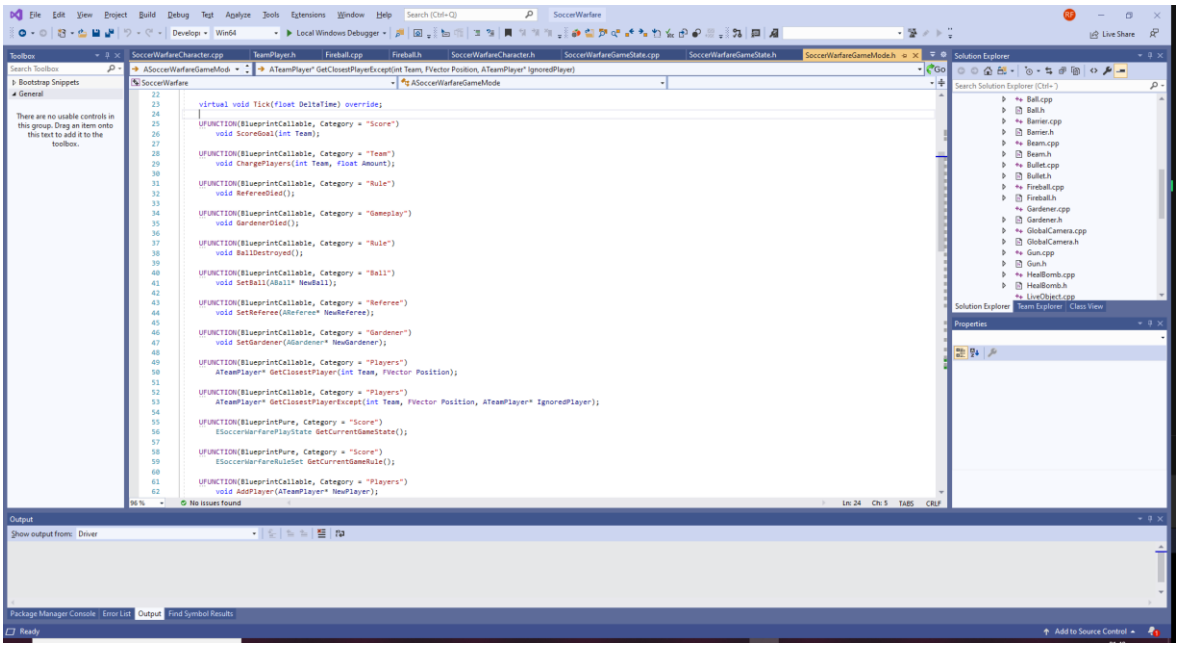

Figura 11 - Header do SoccerWarfareGameMode

Foi depois criada uma blueprint que herda desta classe para lidar com o login multijogador, receber os *roles* selecionados e associá-los aos jogadores.

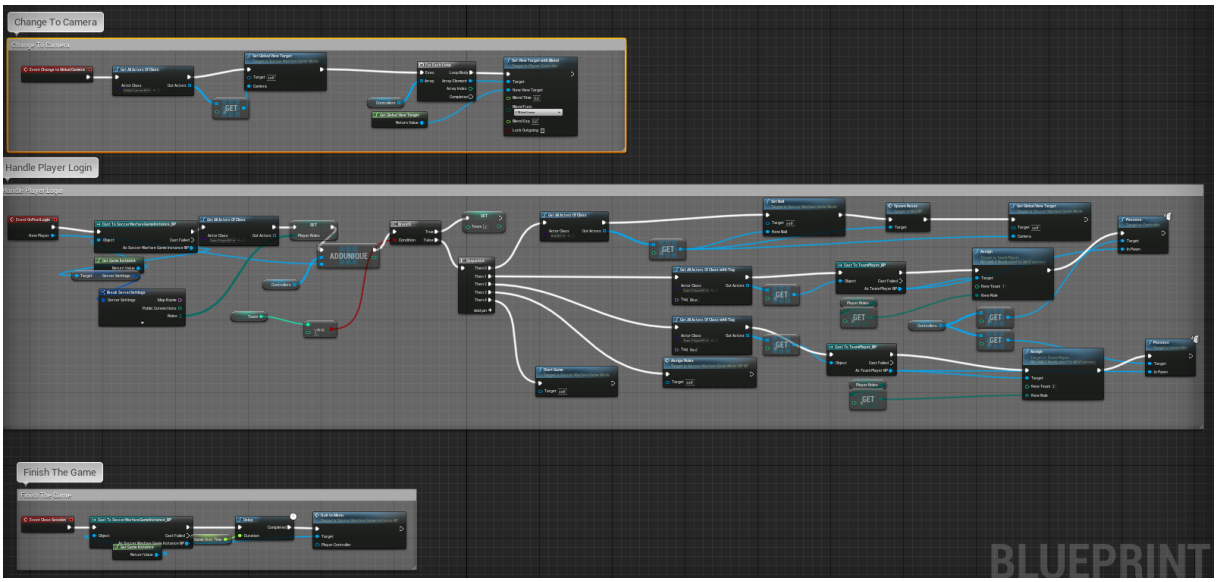

Figura 12 - Blueprint do SoccerWarfareGameMode

GameState

Esta classe armazena, a duração que o jogo está a decorrer, quando cada jogador entrou no jogo e o estado de jogo.

A classe `SoccerWarfareGameState` foi criada para armazenar o tempo de jogo, referências de unidades, bola, jogadores possuídos por humanos e métodos para a UI poder apresentar os dados. Foi depois criada a blueprint que herda desta classe para permitir a seleção deste GameState na blueprint de GameMode.

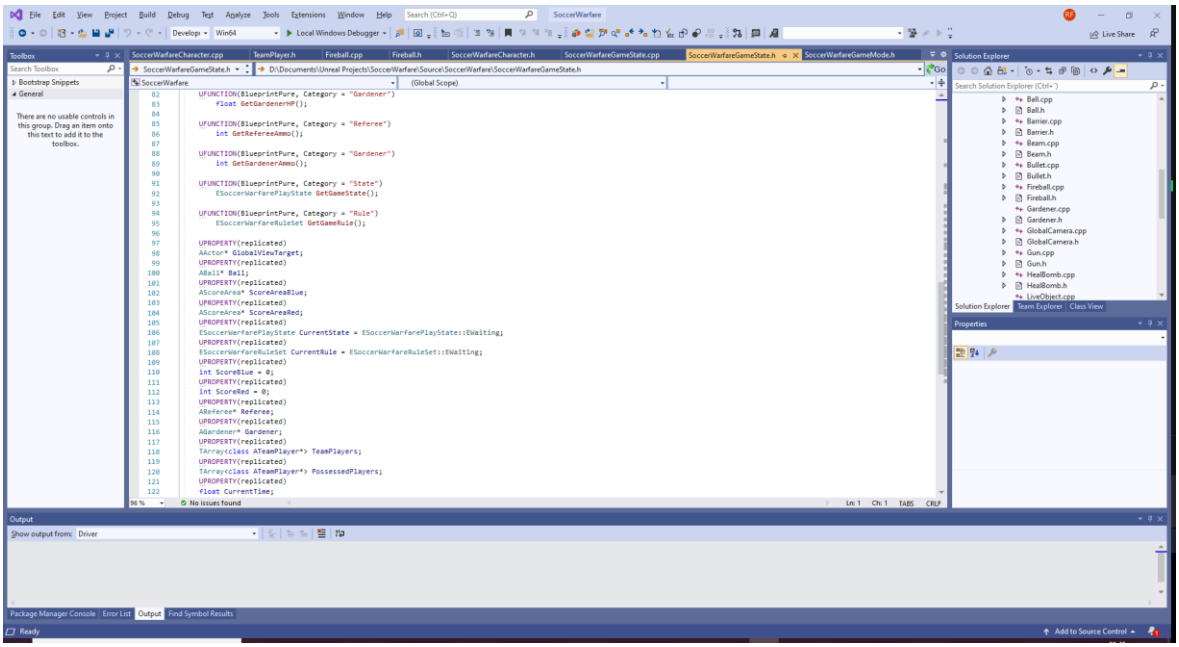

Figura 13 - Header do SoccerWarfareGameState

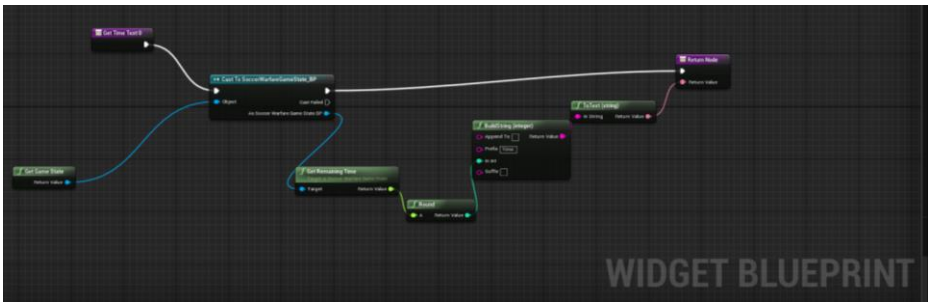

Figura 14 - Função binding do tempo restante

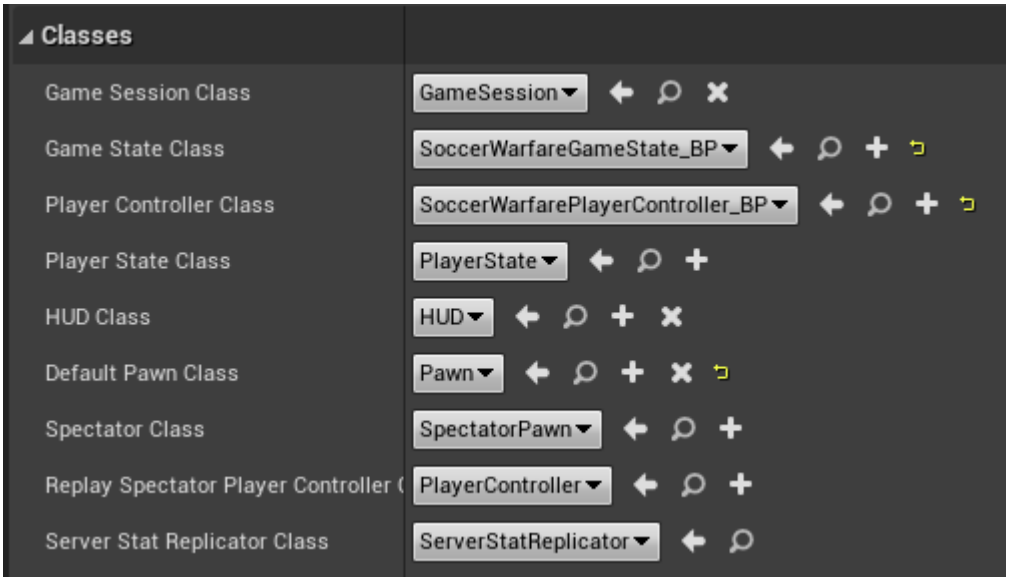

Figura 15 - Classes e Blueprints do SoccerWarfareGameMode

GameInstance

Esta classe é persistente durante a execução do jogo inteira. Isto é: não é apagada com a transição de níveis, o que a torna útil para armazenamento de informação.

Foi criada a blueprint `SoccerWarfareGameInstance` que herda desta classe, para a criação de sessões, entrada e procura destas. Esta classe armazena os *roles* escolhidos do mapa do menu principal.

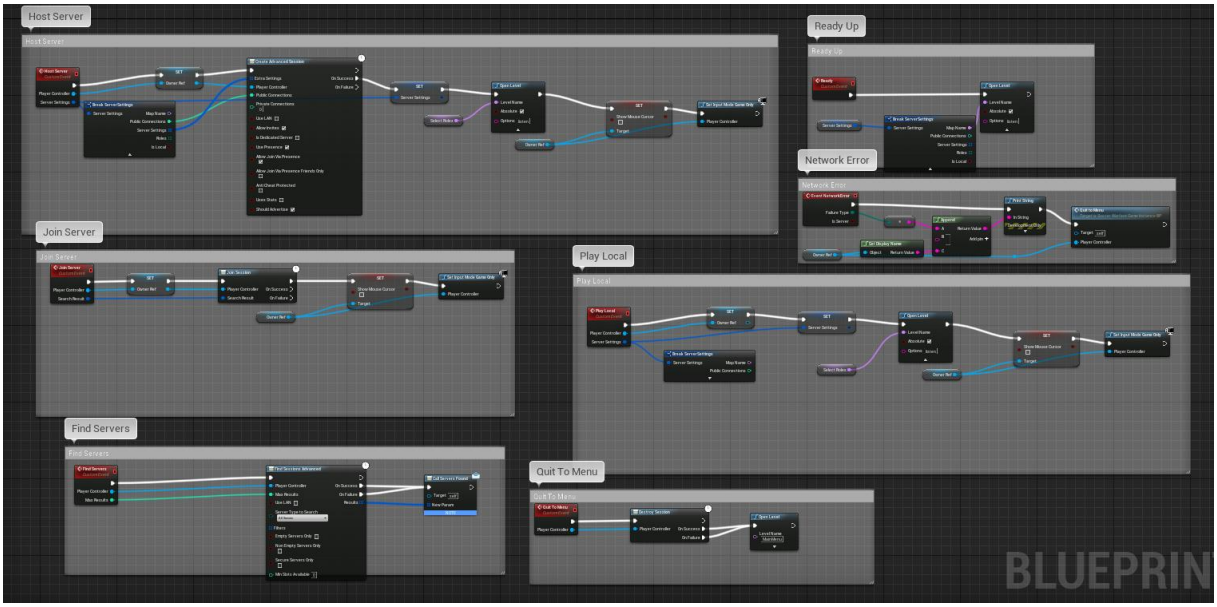

Figura 16 - Blueprint SoccerWarfareGameInstance

3.3.Macros C++ do Unreal Engine

Para o desenvolvimento de projetos C++ em Unreal Engine, ao criar uma classe, são gerados 2 ficheiros pertencentes a essa classe, o ficheiro de extensão “.h” e o ficheiro de extensão “.cpp”. O ficheiro header (.h) irá conter as declarações de funções e atributos dessa classe com os seus níveis de acessibilidade (public, protected, private) e o outro ficheiro (.cpp) irá conter a implementação das funções declaradas na header file correspondente. Isto é comum a qualquer tipo de projeto C++.

Para declarar uma propriedade, é possível acrescentar atributos especiais específicos com a macro UPROPERTY, de forma a que esta possa ser acedida por Blueprints. Nessa macro pode-se declarar esse atributo como replicado (que será explicado num futuro capítulo), adicionar uma tag (visualização no Editor), o seu nível de acessibilidade e muito mais.

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Ball", meta = (allowprivateaccess = true))
    UStaticMeshComponent* BallMesh;
```

Figura 17 - Macro de propriedade que será acedida pela blueprint

```
UPROPERTY(replicated)
AActor* GlobalViewTarget;
```

Figura 18 - Macro de propriedade replicada

Ao declarar uma função, para que as blueprints a possam chamar, ou se for necessário replicá-la, é necessária a utilização da macro UFUNCTION.

Para que as blueprints possam chamar a função, utiliza-se o atributo especial BlueprintPure ou BlueprintCallable. O BlueprintPure adequa-se a funções *GET* pois não recebem nó de execução. O BlueprintCallable adequa-se a qualquer função executável não utilizada em rede.

```
UFUNCTION(BlueprintPure, Category = "Score")
    ESoccerWarfareRuleSet GetCurrentGameRule();

UFUNCTION(BlueprintCallable, Category = "Players")
    void AddPlayer(ATeamPlayer* NewPlayer);
```

Figura 19 - Macros de funções

Figura 20 - Funções s visíveis na blueprint

O código C++ também pode executar eventos numa Blueprint.

```
UFUNCTION(BlueprintImplementableEvent, Category = "Power")
    void PowerChangeEffect_Event();
```

Figura 21 - Declaração de um evento de blueprint

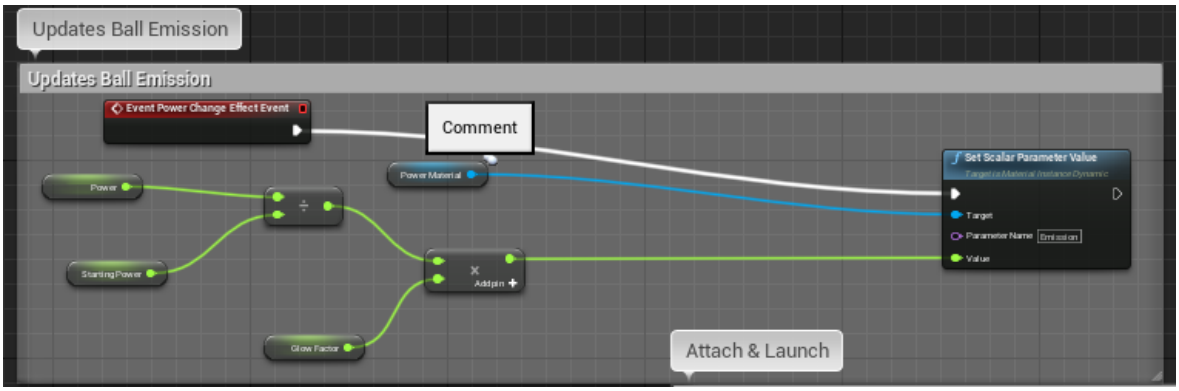

Figura 22 - Evento blueprint

Para funções executadas em rede é necessário criar uma função com o mesmo nome com "_Implementation" acrescentado no fim. Não será criada a função normal no ficheiro de código, mas sim a função Implementation.

```
UFUNCTION(NetMulticast, Reliable, Category = "Color")
    void ColorChangeEffect();
    void ColorChangeEffect_Implementation();
UFUNCTION(Client, Unreliable, Category = "Damage")
    void DamageReaction();
    void DamageReaction_Implementation();
```

Figura 23 - Funções replicadas

```
void ABall::PowerChangeEffect_Implementation()
{
    PowerChangeEffect_Event();
}
```

Figura 24 - Implementação da função replicada

Há muitas mais propriedades que podem ser utilizadas nas macros que não são referenciadas neste documento, tais como:

- Client;
- Reliable;
- Unreliable;
- Server;
- Exec;
- BlueprintCosmetic.

4. Implementação

O jogo para ser jogado irá necessitar de uma área de jogo. Isto é: um nível. Onde serão colocadas as áreas de marcação, os jogadores, o árbitro, o jardineiro e a bola.

4.1. Criação do Nível

Para o nível de jogo principal começou-se por colocar o chão, barreiras, áreas de marcação, adicionar os 12 jogadores, o árbitro, o jardineiro e a bola.

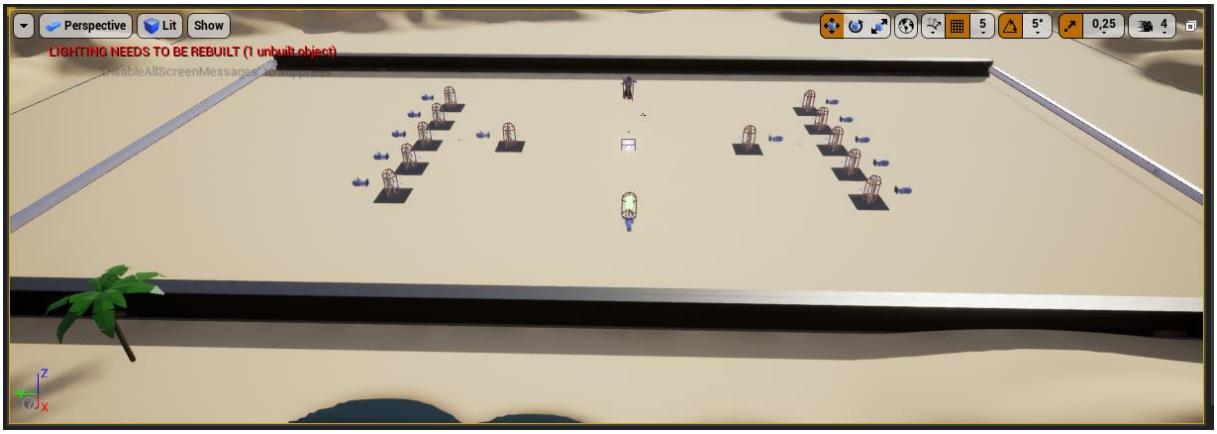

Figura 25 - Nível do jogo

Após o estudo sobre a arquitetura do Unreal Engine 4, deu-se início ao projeto a partir de um template que consiste num jogo de terceira pessoa. Tendo em mente a arquitetura e o GDD (anexo A). Foram criadas várias classes para cada tipo de unidade controlável e não controlável. Estas classes herdam da classe `SoccerWarfareCharacter`. Esta classe contém os atributos comuns a todas as unidades.

Após criadas as classes, foram criadas blueprints que herdam dessas classes para facilitar a edição de static meshes e outro código que será necessário testar ou influencie com materiais, rede, colisões e outros atores.

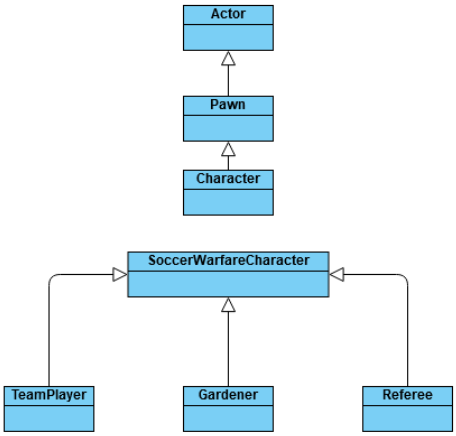

Figura 26 - Diagrama de classes

Após ter preparado os atributos necessários tendo em mente o GDD, deu-se a procura das animações para aplicar para cada habilidade especial e movimentações necessárias.

Para obter algumas animações foi utilizada a plataforma Mixamo. Esta trata-se de uma biblioteca de animações customizáveis para humanóides. Também pode ser utilizada para *auto rigging*, que trata de criar a estrutura óssea (bones) de um modelo 3D, para que neste possam ser aplicadas animações.

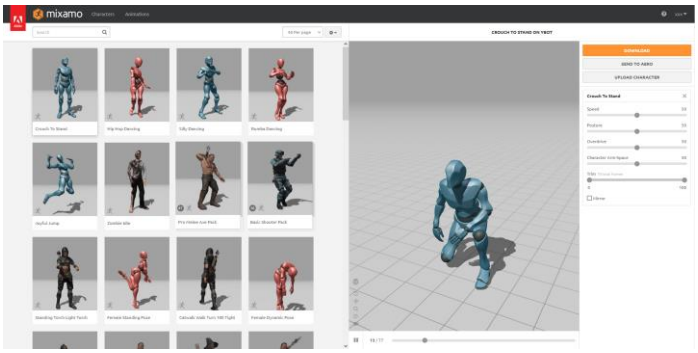

Figura 27 - Mixamo

O Unreal Engine fornece já uma ferramenta chamada de Animation Montage (montagem). Esta pode ser contituida por faixas, secções, clips de animações e triggers.

Foi necessário fazer o retarget das animações de Ybot Mixamo para ThirdPersonCharacter. Foram criadas montagens que e irão conter um trigger que será executado pela animation blueprint do character.

Após ter as animações completas, foi criado código que seria executado após as animações para executar as habilidades com lógica de jogo definida no GDD. Para além disso foram criados parâmetros que serão editados na blueprint.

Foram então criadas classes que herdam da classe Actor presente no UE4 visto que que são objetos de jogo, tais como: áreas de marcação, bolas, balas, caixas de balas, bolas de fogo, feixes de energia, paredes e esferas de cura.

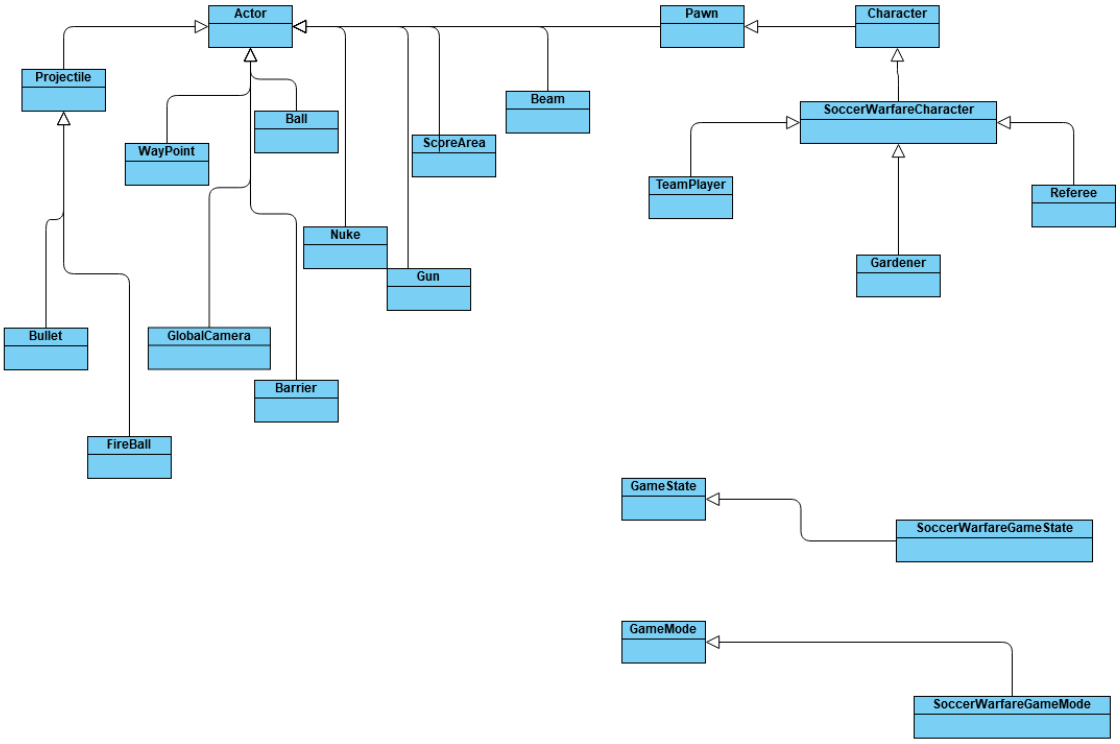

Figura 28 - Diagrama de classes completo

Para cada classe foi criada uma blueprint que conterá a sua lógica de colisão e replicação.

Após ter sido criada a lógica para cada classe, foi criada a classe `SoccerWarfareGameMode` que herda da classe `GameMode` e a classe `SoccerWarfareGameState` que herda da classe `GameState`, o *gamestate* guarda e fornece dados, o *gamemode* interage com os dados e altera-os quando necessário, tais como: regras de jogo, estado de jogo e pontuações.

Figura 29 - Objetos de jogo

Foram criados controladores para as entidades controladas pela inteligência artificial. Estes controladores herdam da classe `DetourAIController`, para evitar *gridlock* entre *Pawns*.

Por sua vez, esta classe herda da classe `AIController`, que por sua vez herda da classe `Controller`.

O AI Controller irá fazer uso da NavMesh e irá auxiliar a Behavior Tree ao alterar dados da BlackBoard utilizada pela mesma (estes conceitos irão ser explicados no capítulo de inteligência artificial).

Após ter o funcionamento do jogo completo, procedeu-se à parte de sincronização em rede, começou-se por marcar blueprints, os seus atributos e components como replicáveis (isto será explicado no capítulo dedicado ao multijogador em rede), alteração das classes do ficheiro header para declarar métodos e propriedades como replicáveis, feito isto e testado no editor com a definição de 2 jogadores, notou-se que estava praticamente tudo a ser replicado.

Figura 30 - Executar o jogo em 2 instâncias

No entanto, os inputs de disparo, chuto e habilidades especiais não estavam a ser replicados, para isto teve-se de criar 2 eventos “Run On Server” (será também explicado no capítulo dedicado ao multijogador em rede) que são executados no servidor sempre que o botão de input seja pressionado e largado. Estes eventos tratam de despoletar um evento multicast para informar todos os clientes que a ação correspondente foi realizada ou parada dependendo do evento inicial.

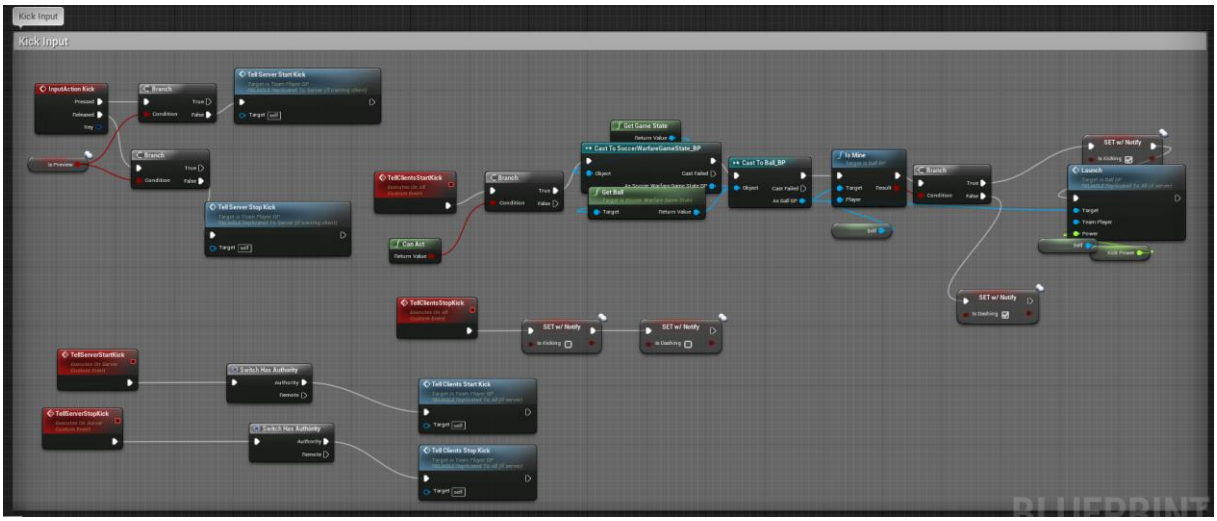

Figura 31 – Kick e Dash replicados

4.2.Inputs

Qualquer jogo que requer interação do jogador, necessita de alguma interface para que a interação seja possível. Na maioria dos casos, esta é feita por rato, teclado ou comandos.

De acordo com o GDD utilizado e tendo em conta as ações disponíveis para o jogador, foi utilizado um input de eixo para movimentação (Axis Mapping), pois permite uma rotação mais suave. Para as restantes ações foram inputs booleanos (Action Mapping), já que as ações são executadas por um aperto de botão único. Para isto será o chuto, disparo, habilidade especial, salto, mudança de jogador.

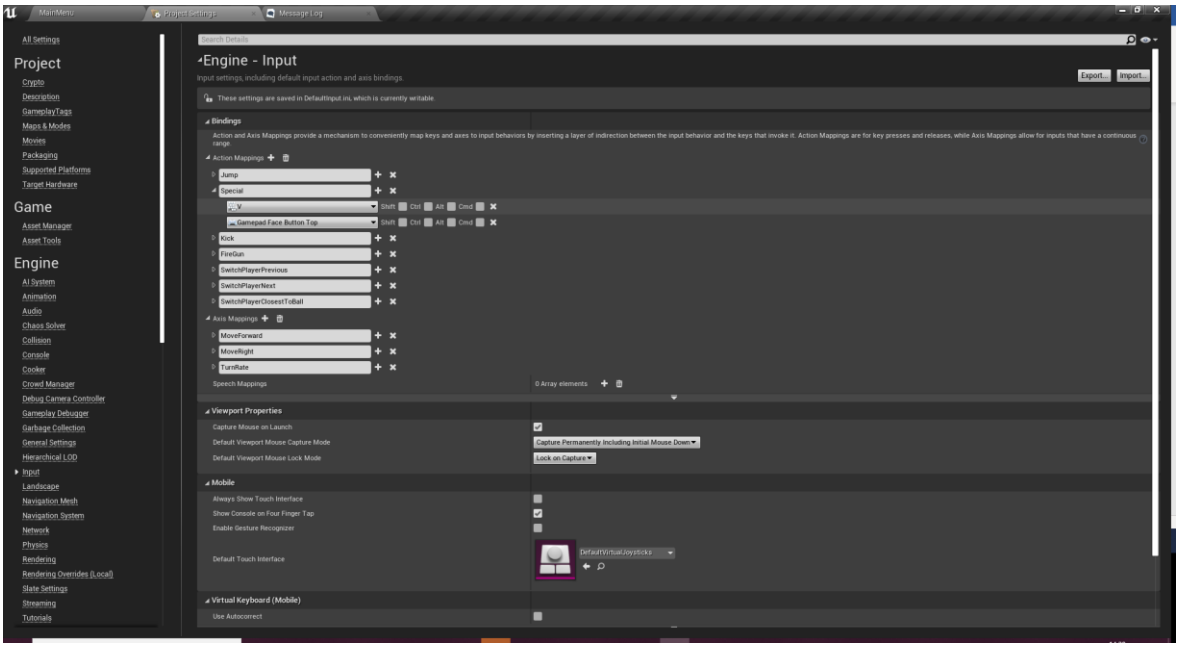

Figura 32 - Menu de definição e mapeamento de controlos

4.3.Humanoide e animações

Para as unidades foi decidida a utilização de *Assets* do *Marketplace* do Unreal Engine. De onde foram descarregados alguns modelos de personagens. Seguidamente nas Blueprints das unidades foram aplicadas essas novas *skeletal meshes* e *blueprints* de animação correspondentes às mesmas. No entanto, quando o seu movimento é executado a partir de um controlador de Inteligência Artificial, não reproduz a animação de andamento porque a animação é reproduzida à base de aceleração e não velocidade. Para resolver este problema, foi ativada a opção "Use Acceleration for Paths" no componente CharacterMovement.

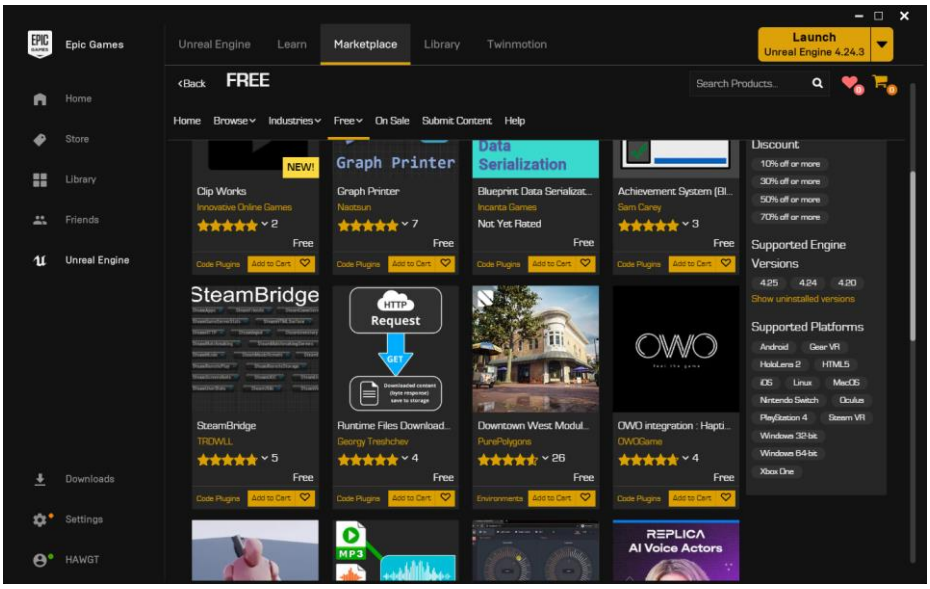

Figura 33 - Marketplace

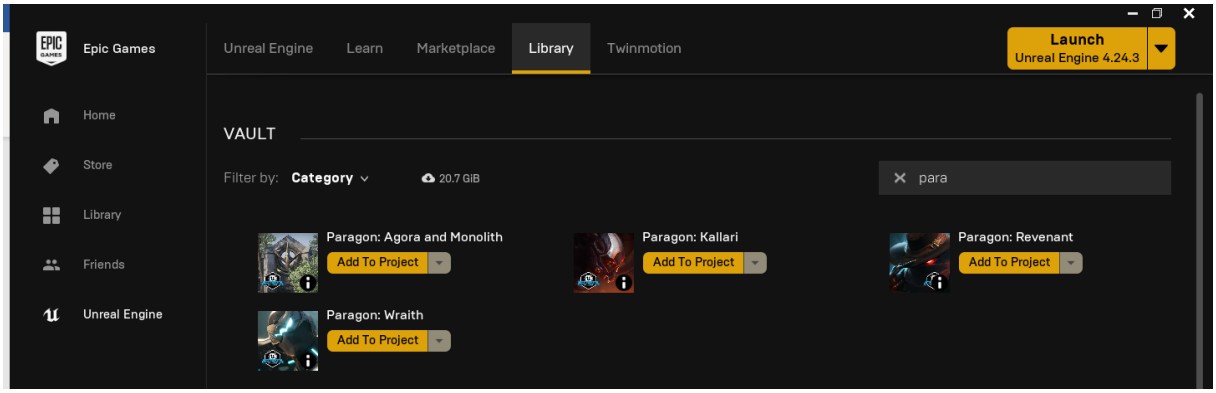

Figura 34 - Assets transferidos

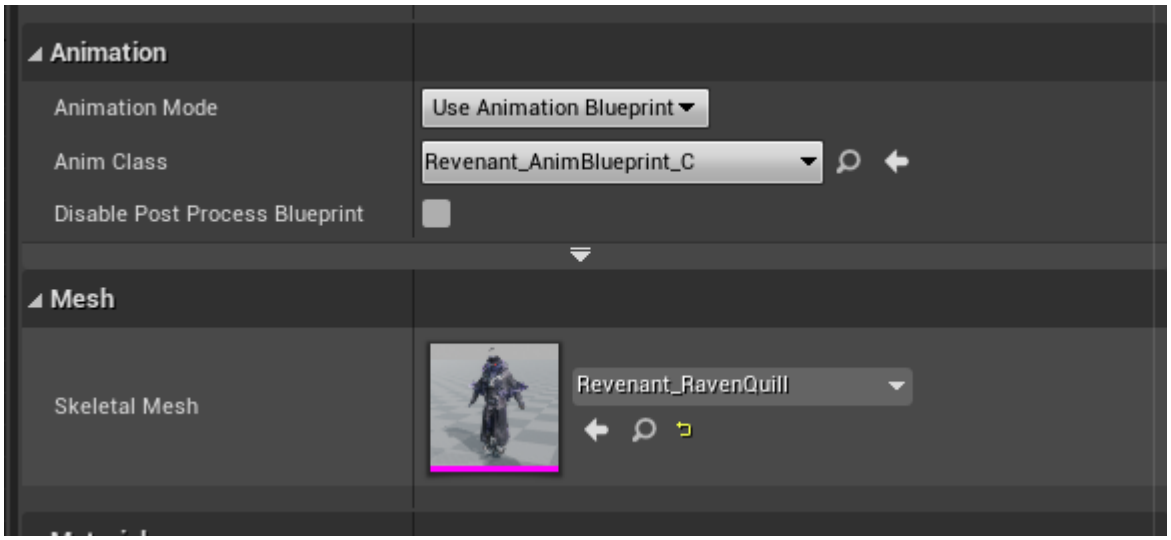

Figura 35 - Mesh & Animation Blueprint

Figura 36- Opção Use Acceleration for Paths

4.4.Câmera de jogo

O tipo de câmara utilizada neste jogo foi a vista aérea, devido a importância de serem visíveis todos os elementos essenciais do jogo. Inicialmente todos os Pawns têm uma câmara interna que o Player Controller utiliza. Caso exista outra câmara além da interna, o Player Controller irá usar esta. A habilidade de se poder de trocar de jogador e manter a câmara aérea foi, por si só, um desafio já que isto altera o comportamento de um Player Controller base. Foram testadas várias alternativas, no entanto eram inconsistentes no lado multijogador em rede. Por esta razão, foi criada uma blueprint que herda do Player Controller.

Para alcançar isto tudo e garantir a sintonia com o GDD, foram necessárias 2 câmaras: uma que segue a bola e outra câmara para quando a bola for destruída ou o árbitro morra. Esta passará a ser a câmara principal. Colocar isto a funcionar no multijogador em rede também foi um grande desafio, assim como a posse de jogador.

```
void ASoccerWarfareGameMode::PossessNext(int Team, int Direction)
{
    ASoccerWarfareGameState* SWGameState = Cast<ASoccerWarfareGameState>(GameState);
    if (!SWGameState) return;
    TArray<ATeamPlayer*> AlivePlayers = GetAlivePlayers(Team);
    int iIndex = 0;
    int length = AlivePlayers.Num();

    ATeamPlayer* PossessedPlayer = GetPossessedPlayer(Team);
    ATeamPlayer* ToPossess = PossessedPlayer;

    if (PossessedPlayer == nullptr) return;

    for (ATeamPlayer* TPlayer : AlivePlayers) {
        if (TPlayer == PossessedPlayer) break;
        iIndex++;
    }

    iIndex += Direction;
    if (iIndex < 0) iIndex = length - 1;
    if (iIndex >= length) iIndex = 0;
    ToPossess = AlivePlayers[iIndex];

    AController* OldController = PossessedPlayer->Controller;
    AController* NewController = ToPossess->Controller;
    APlayerController* OController = Cast<APlayerController>(NewController);
    APlayerController* PController = Cast<APlayerController>(OldController);

    if (!PController) return;

    if (ToPossess == nullptr) return;

    if (ToPossess == PossessedPlayer) return;

    if (OldController == nullptr) return;

    if (NewController != nullptr) NewController->Destroy();

    PossessedPlayer->AIControl();
    OldController->Possess(ToPossess);
}
```

Figura 37 - Código de alternação de posse entre jogadores

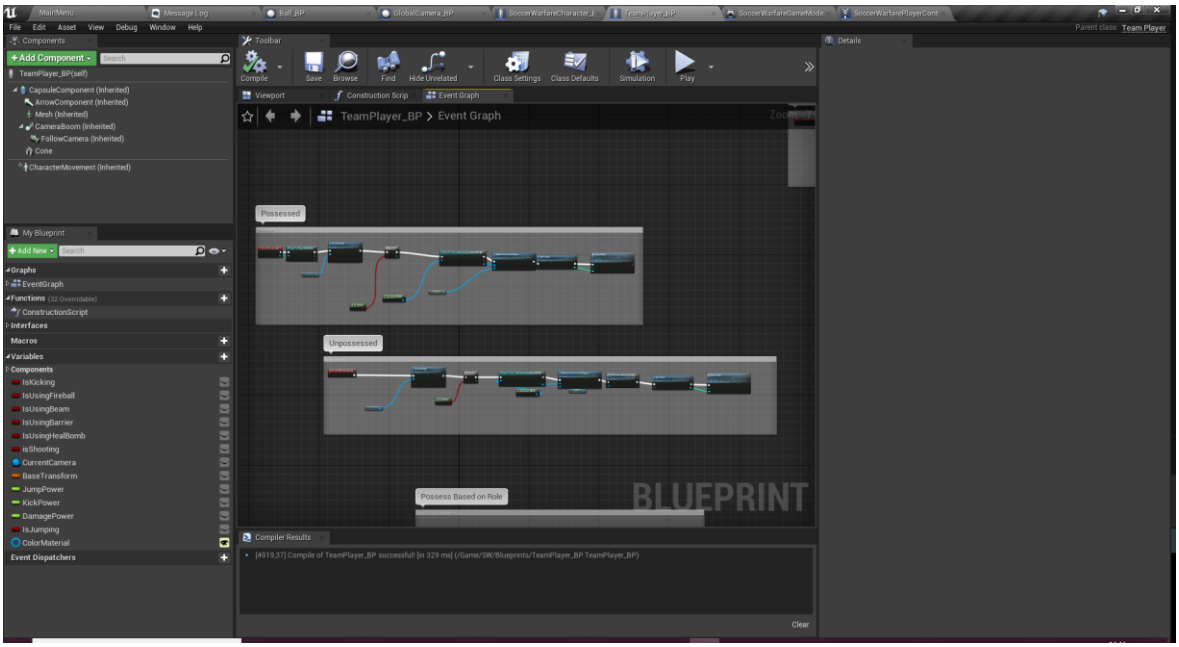

Figura 38 - Eventos de alteração de posse na blueprint do jogador

Neste projeto foi criada a blueprint `SoccerWarfarePlayerController`, esta herda da classe `PlayerController`. Foi criada para refazer o funcionamento base do `PlayerController`, que ao possuir um `Pawn/Character` irá mudar o seu ViewPort para a camara deste. Caso não exista câmara, então irá criar uma. Este comportamento não é o desejado para este tipo de jogo. Deseja-se aqui uma mudança de jogador sem a alteração de camâra.

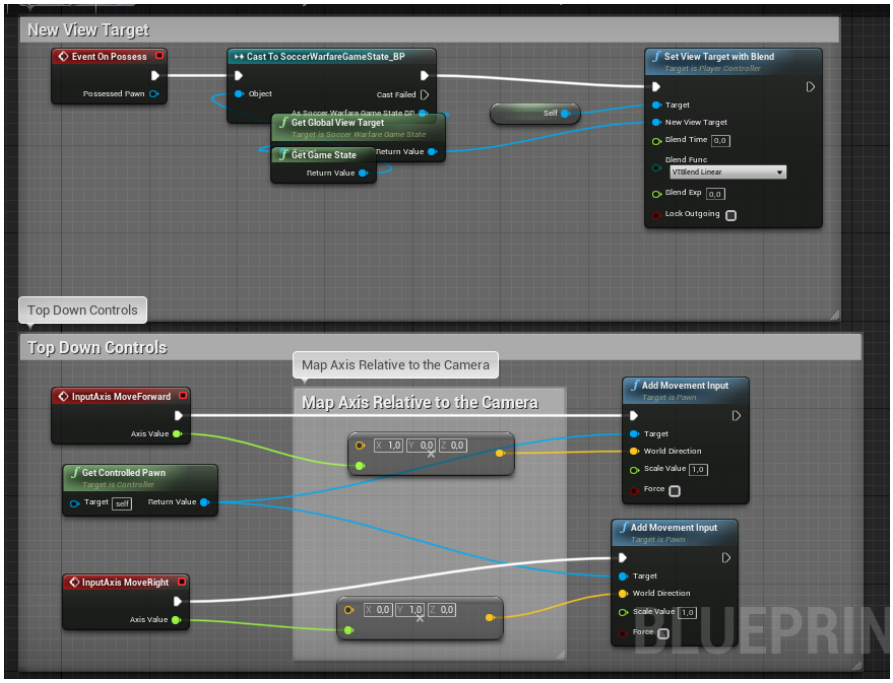

Figura 39 - Blueprint SoccerWarfarePlayerController

Para o efeito criou-se um *event handler* para o evento “OnPossess” na blueprint `SoccerWarfarePlayerController`, que irá definir o ViewTarget para o ViewTarget

global que está presente no SoccerWarfareGameState, este pode ser a câmara da bola ou a câmara secundária.

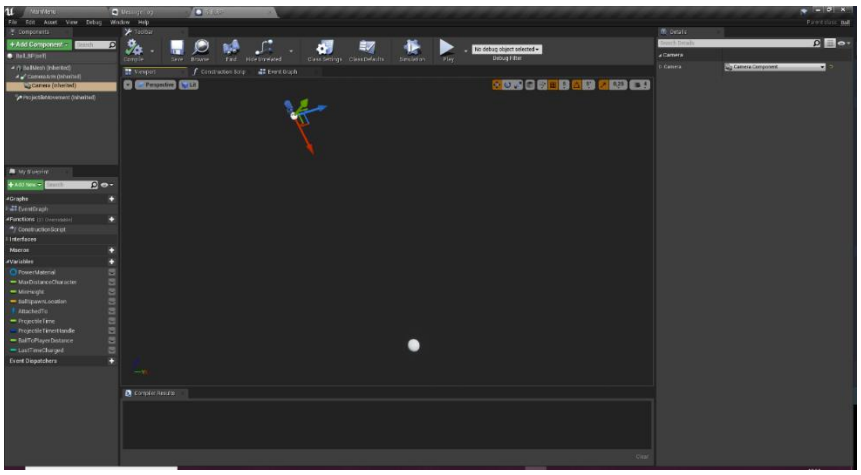

Figura 40 - Câmara que segue a bola

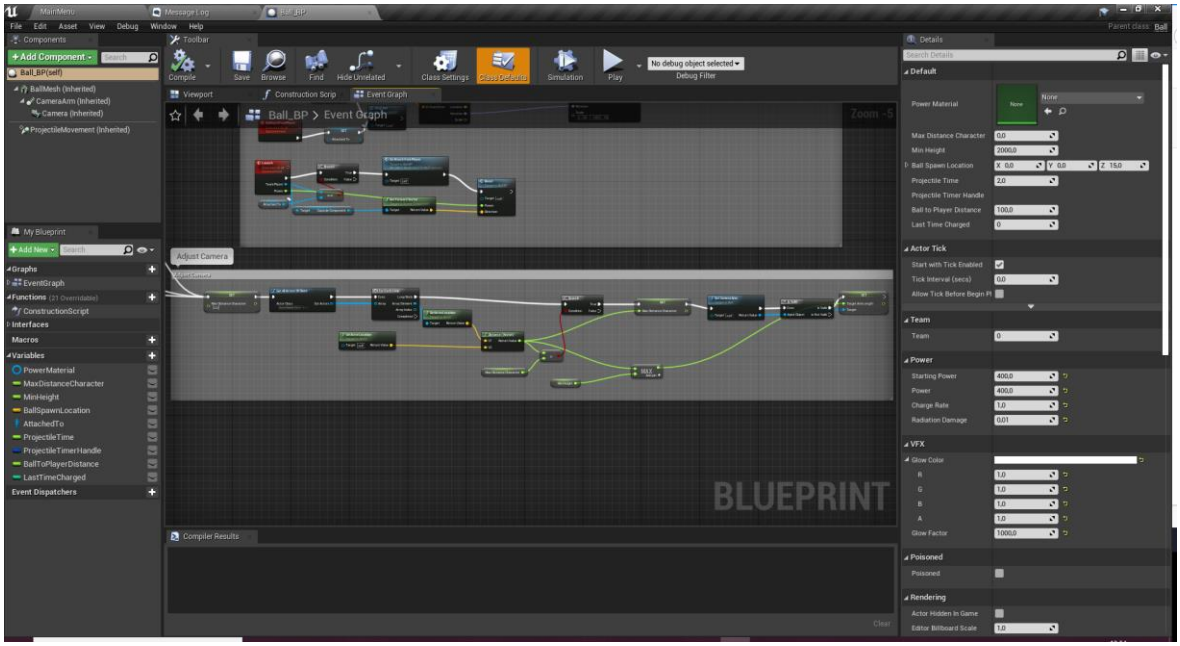

Figura 41 - Câmara altera a sua distância dependendo das posições dos jogadores

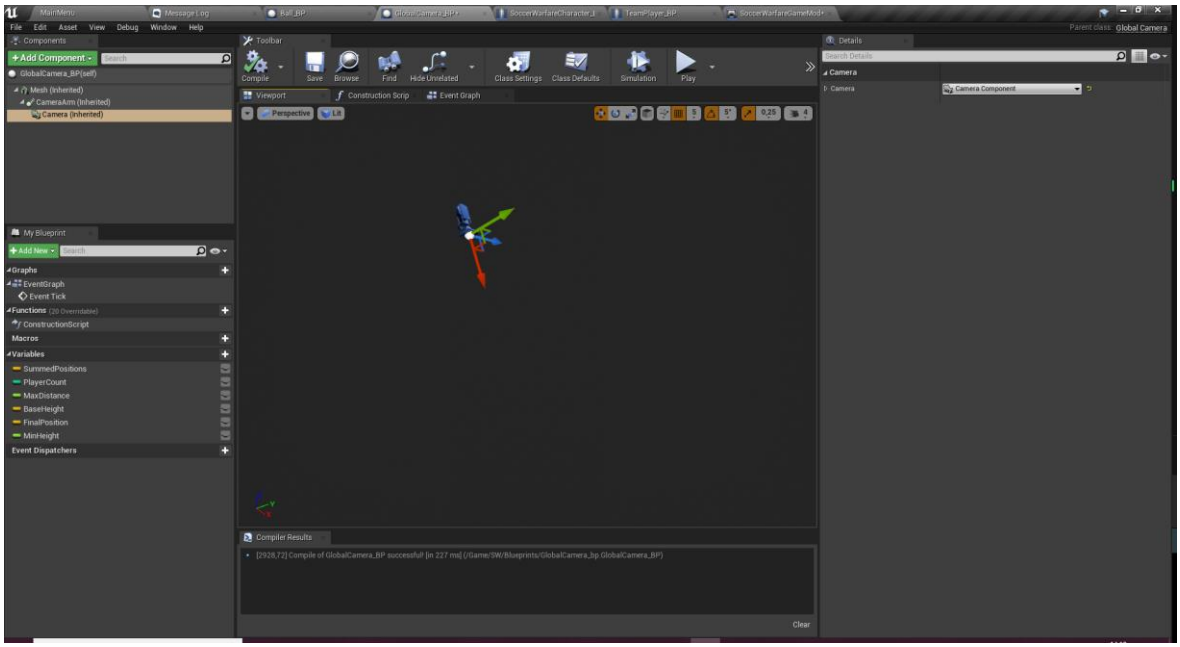

Figura 42 - Câmara secundária

O fluxo funciona da seguinte maneira: o SoccerWarfareGameMode define a referência da bola e a sua câmara como câmara global no SoccerWarfareGameState. Sempre que o PlayerController possuir uma unidade, este vai buscar a referência da câmara global ao SoccerWarfareGameState e definir o seu view target para esta referência. Assim que a bola é destruída ou árbitro morre, um evento é chamado para atualizar a câmara global para a referência da outra câmara referida anteriormente. Este evento também força a mudança de câmara em todos os Player Controllers já que não estão a possuir uma nova unidade.

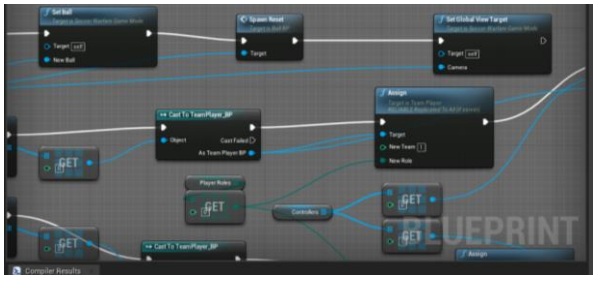

Figura 43 - Definição do view target global e bola

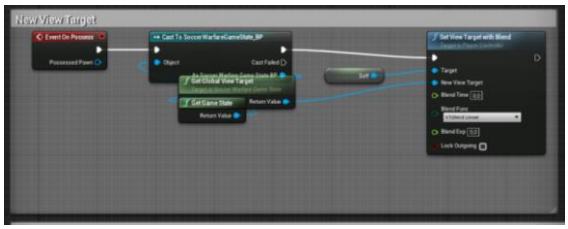

Figura 44 - Utilização da câmara global na tomada de posso pelo controller

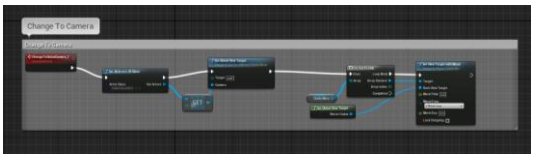

Figura 45 - Mudança para a câmara secundária

4.5. Objetos de jogo

Esta secção irá explicar o funcionamento de cada objeto de jogo.

4.5.1. Bola

A bola é um objeto que herda da classe `Actor` e tem uma Mesh, uma câmara, pontos de energia e ainda uma propriedade que contém a última equipa que lhe tomou posse. Após criar o ficheiro cpp e header, criou-se uma blueprint que herda esta classe. É nesta blueprint onde se configura a Mesh desejada, e adiciona-se lógica adicional e depois coloca-se a bola (blueprint) no nível.

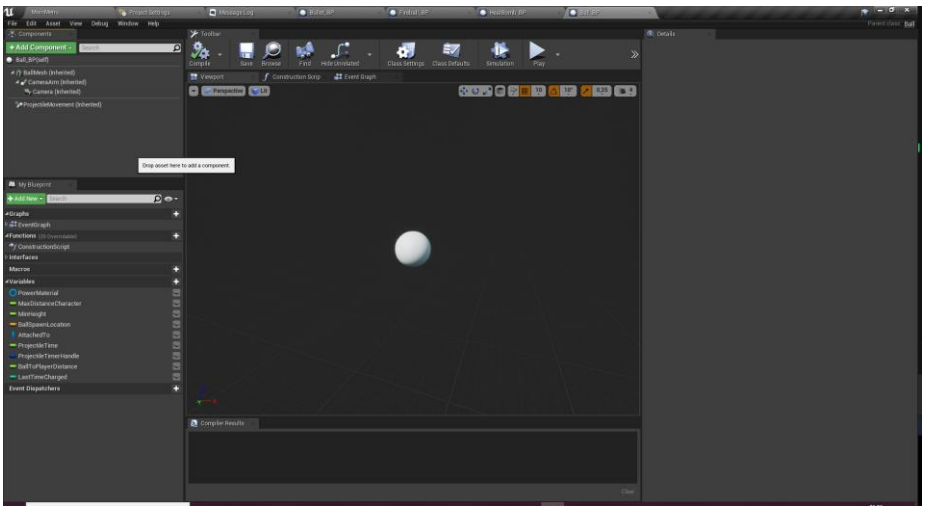

Figura 46 - Blueprint bola

A bola ao entrar em contato com um jogador por colisão irá-se agarrar a este até que o jogador receba dano ou a lance com o chuto, o lance é feito utilizando o projectile movement component. Esta irá mudar de cor de acordo com a última equipa que lhe tocou, e irá ainda fornecer energia a todos os elementos dessa mesma equipa.

A bola ao levar dano, perde pontos de energia. Se este valor chegar a 0 ou inferior, a bola irá ser destruída mudando o objetivo de jogo para o modo de sobrevivência.

A bola ao entrar em contacto com uma área de marcação irá fazer uma comparação, a última equipa que lhe tocou e a equipa da área de marcação, se forem diferentes, a bola será transportada para a posição inicial e um ponto será marcado para a última equipa que tocou na bola, isto é informado ao GameMode que irá atualizar o GameState, onde a UI irá buscar a pontuação.

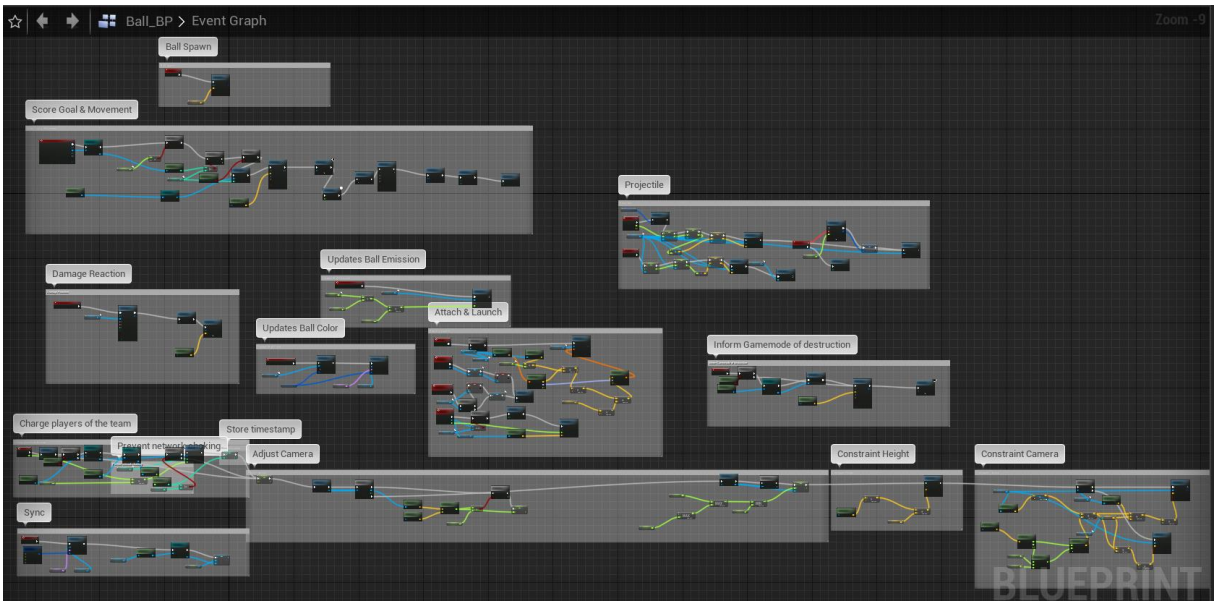

Figura 47 - Event Graph da blueprint da bola

Tabela 1 – Atributos da bola

Energia Inicial	400/400
Taxa de Regeneração de Energia aos Jogadores da Equipa/s	1

Colisões

As colisões são detetadas por colliders. Estes componentes detetam a colisão de outros objetos com estes, quer seja por impacto ou por sobreposição. Pode ser configurado o raio, a forma, o tamanho e o tipo de colisão (este pode tornar o objeto sólido ou não).

As meshes e colliders já vêm preparadas com eventos de colisão, tais como:

- `OnComponentHit` – Destinado a "toques", como por exemplo, balas;
- `OnComponentBeginOverlap` – Destinado a sobreposições sendo que pelo menos 1 objeto interviniente não seja sólido, utilizado no Beam visto que não impede a movimentação, apenas causa o dano;
- `OnComponentEndOverlap` – Destinado a sobreposições sendo que pelo menos 1 objeto interviniente não seja sólido, este evento é executado assim que a sobreposição termine, não está a ser utilizado atualmente pois não existe nenhuma mecânica que o necessite.

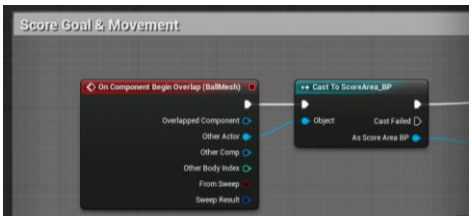

Figura 48 - Colisão da bola com a área de marcação

Materiais Dinâmicos

Os materiais são o que define a aparência dos objetos, como por exemplo: a cor, a textura, o brilho, a transparência, a superfície, entre outros. Quando a luz atinge a superfície de um objeto, o material é utilizado para calcular como a luz interage com a superfície do objeto.

A bola, como muda de cor, irá utilizar um material dinâmico. Este será criado no Construction Script da Blueprint. O Construction Script de uma Blueprint assemelha-se a um construtor de classe. É extremamente poderoso para ações como fazer rastreios no mundo, criando objetos que se adaptam ao meio presente, como por exemplo: um relvado gerado de acordo com o espaço livre presente.

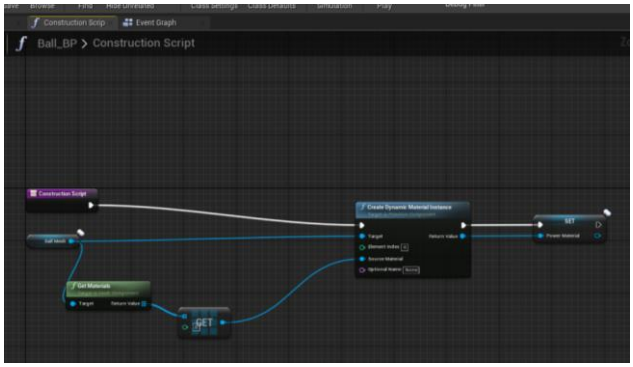

Figura 49 - Construction Script da bola

O material criado para a bola recebe um parâmetro de cor, logo será instanciado um material dinâmico que usará esse material como fonte, onde em execução será acedido e modificado o parâmetro de cor.

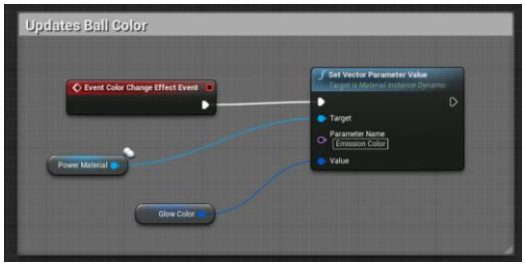

Figura 50 - Alteração de parâmetro do material dinâmico

4.5.2. Zona de Marcação

A blueprint zona de marcação herda da classe `ScoreArea` que por sua vez herda da classe `Actor`. Apenas contém uma mesh, uma propriedade que indica a equipa que pertence e outra com a cor. Também utiliza um material dinâmico.

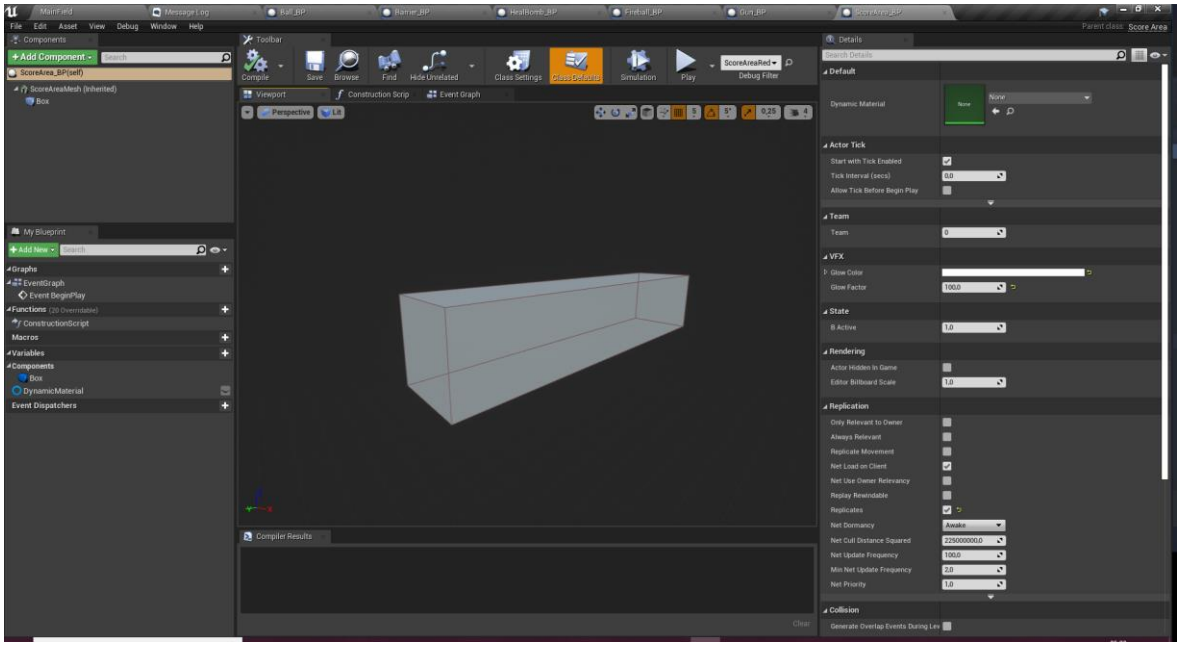

Figura 51 - Blueprint da zona de marcação

4.6. Interface Gráfica

Para criar interfaces gráficas e menus no Unreal Engine foi utilizado o Unreal Motion Graphics (UMG).

SoccerWarfare.Build.cs | SoccerWarfare.cpp | SoccerWarfare.h | SoccerWarfare.Target.cs | SoccerWarfareGameState.h | Ball.cpp | Ball.h | TeamPlayer.h

Miscellaneous Files | SoccerWarfare | SoccerWarfare(ReadOnlyTargetRules Target)

```
1   // Copyright 1998-2019 Epic Games, Inc. All Rights Reserved.
2
3   using UnrealBuildTool;
4
5   using UnrealBuildTool;
6
7   public class SoccerWarfare : ModuleRules
8   {
9       public SoccerWarfare(ReadOnlyTargetRules Target) : base(Target)
10      {
11          PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;
12
13          PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore", "HeadMountedDisplay", "UMG", "OnlineSubsystem", "
14
15          DynamicallyLoadedModuleNames.Add("OnlineSubsystemSteam");
16
17          PrivateDependencyModuleNames.AddRange(new string[] { "Slate", "SlateCore" });
18
19          PrivateDependencyModuleNames.Add("OnlineSubsystem");
20      }
21  }
22
```

Figura 52 - Definições da Build

4.6.1. Menu Principal

Para o menu principal foram criados 3 botões: multijogador local e em rede, e um botão sair.

Foi criado um mapa vazio onde o menu seria aplicado para poupança e otimização e depois foi alterada a blueprint para mostrar este menu sempre que o nível seja carregado.

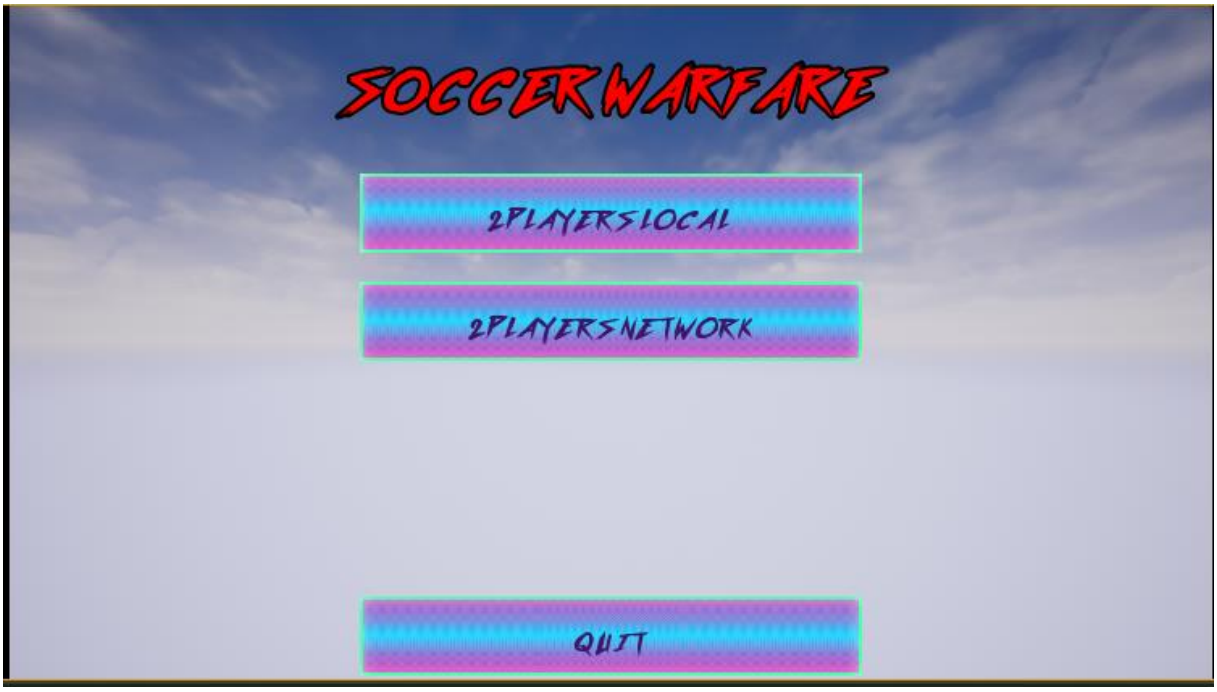

Figura 53 - UI Menu Principal

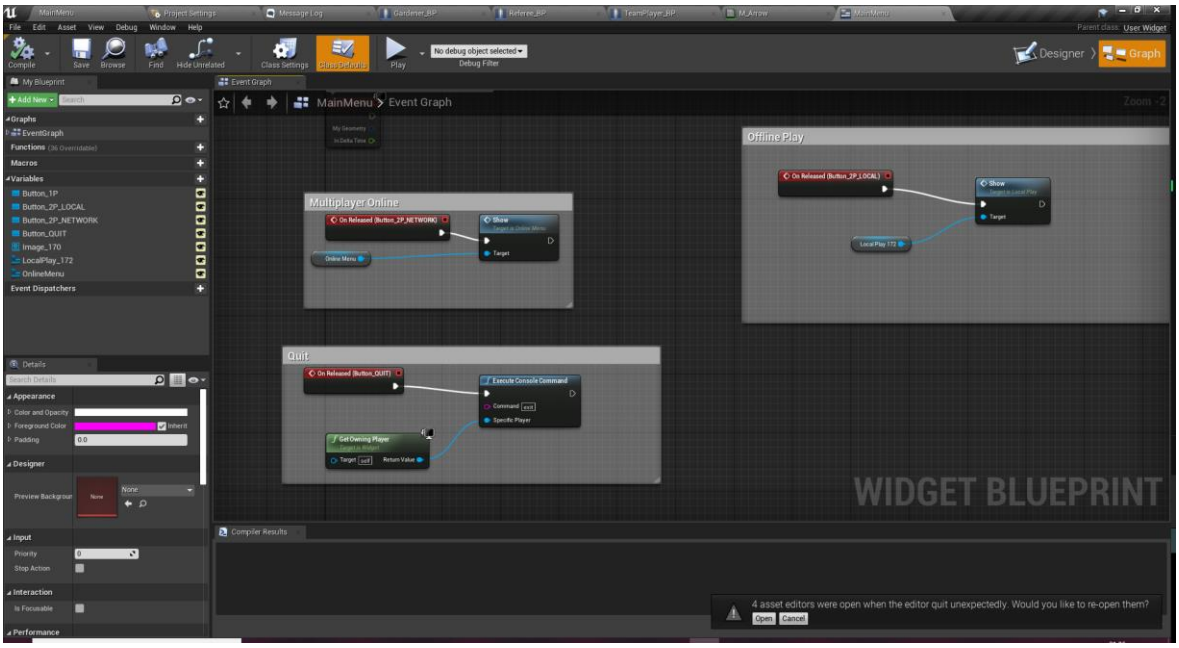

Figura 54 - Funcionalidade dos botões do menu principal

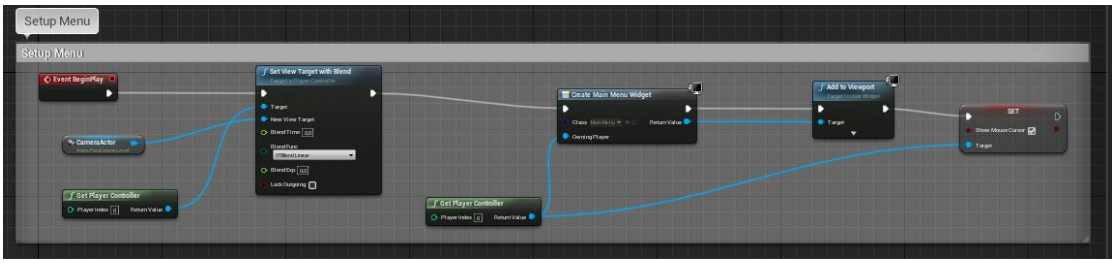

Figura 55 - Setup do menu

4.6.2. Game HUD

Neste jogo foi criado um painel de informação que mostra a vida, energia, munições, *role* do jogador possuído atualmente, tempo, pontuação, objetivo, vida da bola e informação do árbitro e jardineiro. Para tal, foi necessário criar o HUD com a formatação e locais onde a informação será mostrada.

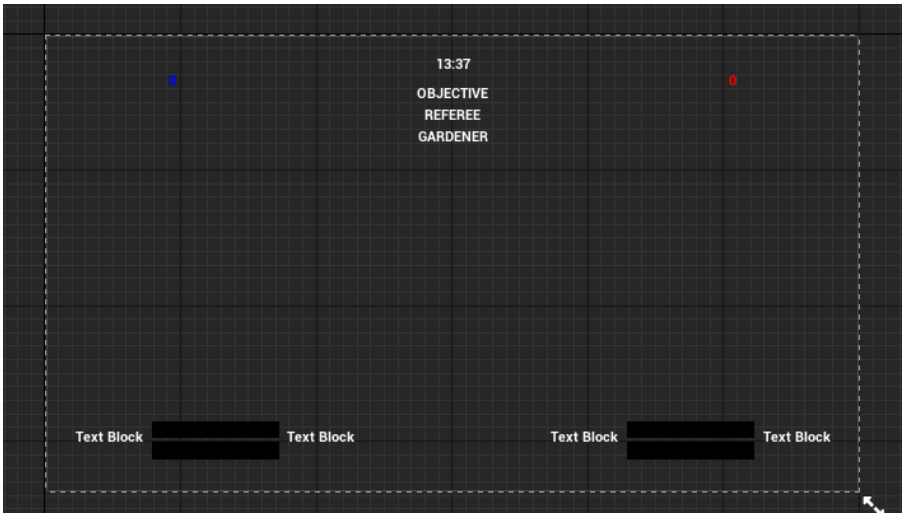

Figura 56 – Painel de informação de jogo

Note que os dados são dinâmicos, logo é preciso arranjar maneira de alterar os dados sempre que necessário. Para isto criaram-se bindings que são funções blueprint capazes de alterar texto e outros atributos do objeto selecionado.

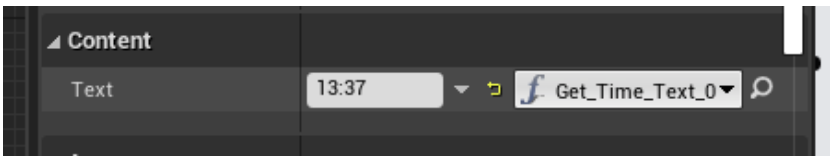

Figura 57 - Definição de uma bind

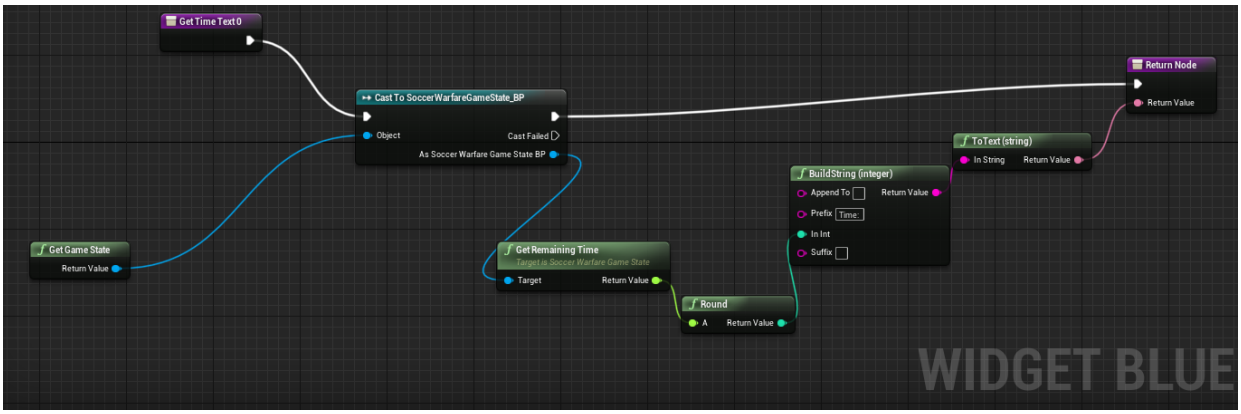

Figura 58 - Função bind criada

Esta função demonstrada acima, recebe o valor de tempo restante da classe `SoccerWarfareGameState` e retorna o valor em forma de string para o texto da textbox `Time`. Faz-se isto para os restantes elementos do HUD.

4.7. Unidades Controláveis

Neste jogo, as diferentes unidades têm comportamentos comuns, logo foi criada uma hierarquia na implementação destas.

4.7.1. Unidade Base

Esta unidade é a que está no topo hierárquico no que diz respeito às unidades, a classe `SoccerWarfareCharacter` implementa o comportamento base e propriedades das outras unidades.

As propriedades principais da unidade base são:

- Energia, energia máxima e taxa de regeneração de energia por segundo;
- Balas;
- Cooldown atual;
- Equipa;
- Estado de envenenamento por radiação;
- Cor (utilizada para o material dinâmico).

Figura 59 - Jogo a decorrer

Os métodos base interagem com as propriedades principais, como a receção de dano e cura. Para além desses métodos, também está implementada a habilidade de disparo e a movimentação do humanoide.

A unidade base é composta por um Capsule Component, este componente trata da colisão com objetos e é possível observá-lo na figura abaixo, onde está representado com um tom de rosa-escuro. Contém o componente Mesh que se trata do próprio modelo 3D do humanoide, este modelo 3D está associado à Animation Blueprint onde estão as animações do humanoide. O Arrow Component serve para indicar para onde o objeto está virado, este não é visível durante o jogo. Contém uma câmara e o componente CharacterMovement que armazena dados tais como a velocidade de movimentação, salto, etc.

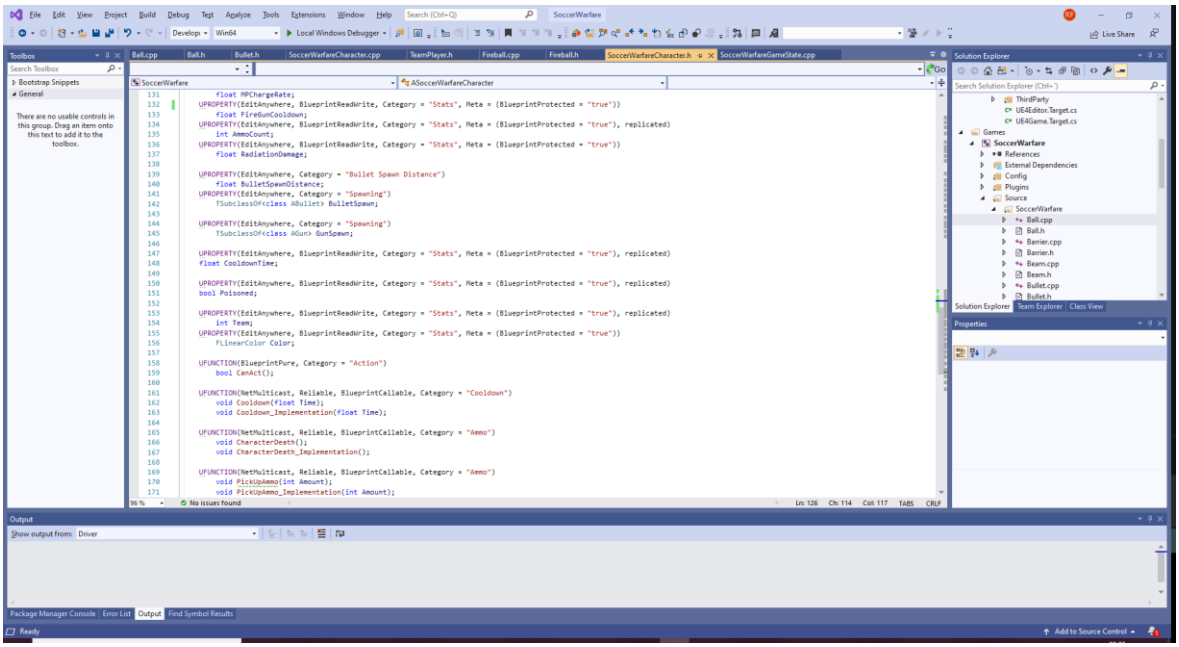

Figura 60 - Métodos e atributos da unidade base

4.7.2. Jogador

Esta unidade, herda da classe `SoccerWarfareCharacter`, acrescentando a funcionalidade de utilização de habilidades, uma propriedade nova que representa o *role* que afetará a habilidade especial.

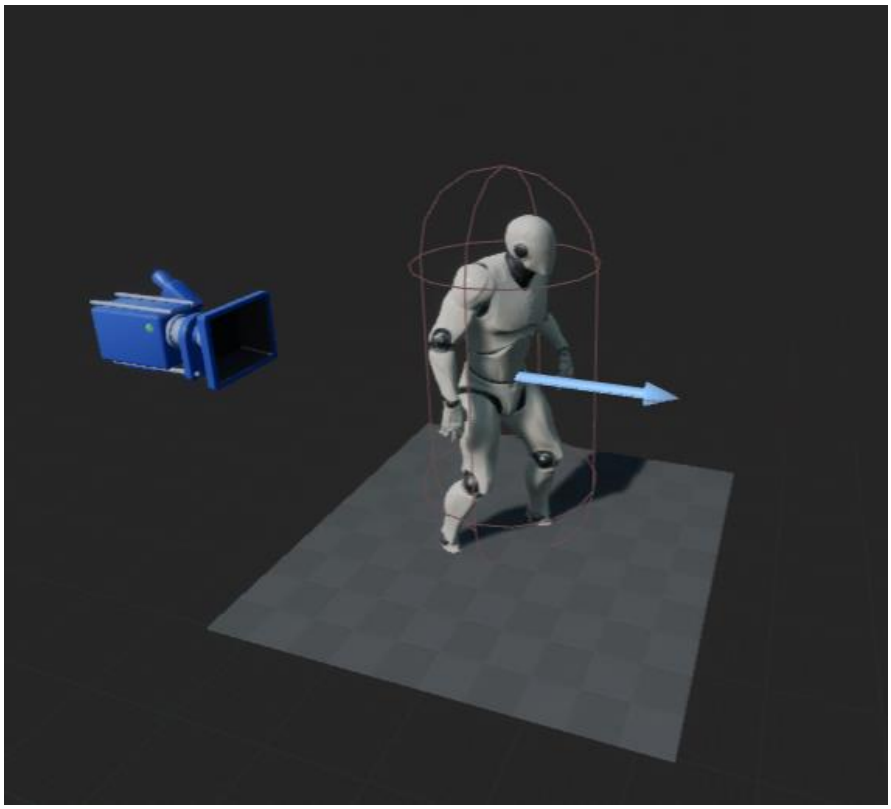

Figura 61 – Jogador sem *role* atribuído

Tabela 2 - Atributos do jogador

Vida	100/100
Energia	100/100
Taxa de Regeneração de Vida	1/s
Taxa de Regeneração de Energia	1/s
Número de Balas	30

Esta unidade pode ser controlada por um humano ou por inteligência artificial. Pertence a uma equipa e tem um *role*, o seu papel na equipa, dependendo deste irá ter a sua habilidade especial específica.

O jogador tem atributos de vida, energia e velocidade. A vida e energia são regeneradas passivamente ao longo do tempo. Pode saltar, disparar, utilizar o chuto e a sua habilidade especial de *role*, que terá um custo de energia. Todas as ações que não sejam de movimentação ou salto ativam um cooldown de ação, este é um tempo de espera necessário para ser permitida a realização de outra ação que não seja de movimentação ou salto, cada ação tem o seu cooldown. Quando morre, larga as suas munições no chão.

A blueprint é semelhante à blueprint base em termos de componentes, contendo adicionalmente o componente Cone que fica visível se a unidade estiver a ser controlada por um jogador humano. Este componente trata-se de uma Static Mesh com um material aplicado.

No entanto contém várias variáveis para indicar o estado de habilidades, estas estas estão marcadas como RepNotify (que será explicado no capítulo de rede) que por sua vez irá executar a montagem de animação e o método respetivo da habilidade.

Processo de Criação de Habilidade

As habilidades, sendo funcionalidade adicional ao Character, terão de ser implementadas. O primeiro passo para criar uma habilidade consiste em criar uma classe C++ que herda da classe Actor ou Projectile, dependendo do tipo de habilidade. Nesta classe são definidos os atributos que esta habilidade terá.

```
rfare                                                    (Global Scope)
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Projectile.h"
#include "Bullet.generated.h"

UCLASS()
class SOCCERWARFARE_API ABullet : public AProjectile
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    ABullet();



protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Stats", Meta = (BlueprintProtected = "true"))
        float Damage;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "VFX", Meta = (BlueprintProtected = "true"))
        float GlowValue;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "VFX", Meta = (BlueprintProtected = "true"))
        FLinearColor GlowColor;

    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Bullet", meta = (allowprivateaccess = true))
        UStaticMeshComponent* BulletMesh;
    UPROPERTY(EditAnywhere, Category = "Components")
        class UProjectileMovementComponent* ProjectileMovement;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

};
```

Figura 62 - Header da classe Fire Ball

Criada a lógica base dos métodos se utilizados, procede-se à criação da blueprint que herda desta classe. Aí define-se a static mesh, colisão, transform e a lógica que utiliza componentes da própria blueprint e de outras blueprints.

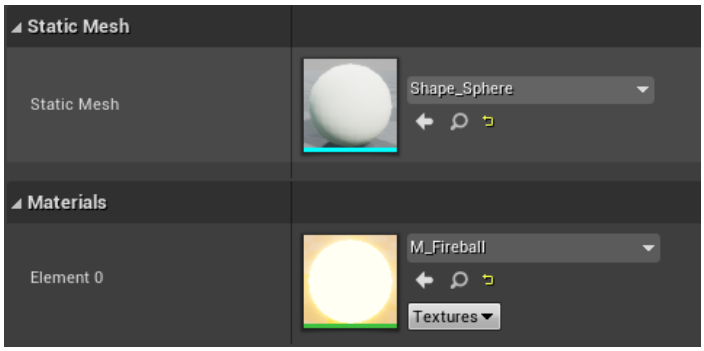

Figura 63 - Static Mesh e Material da Fire Ball

Ao aplicar-se a static mesh, a aba Materials abre-se permitindo a aplicação de um ou mais materiais dependendo da mesh escolhida. Um material controla o aspeto visual onde este for aplicado. Pode ser considerado a tinta que é aplicada num objeto. Pode ser definido o tipo de superfície do material, o seu brilho, transparência e muito mais. Em termos mais técnicos, quando a luz atinge a superfície do material, este calcula como a luz interage com ele. Utilizando imagens (texturas), expressões matemáticas, valores constantes e valores de parâmetro.

Optando por um aspeto visual mais limpo, a maioria dos materiais não utilizam texturas.

Após a criação de um material, podem ser alterados vários parâmetros e propriedades. Depois este pode ser aplicado na mesh da blueprint criada anteriormente.

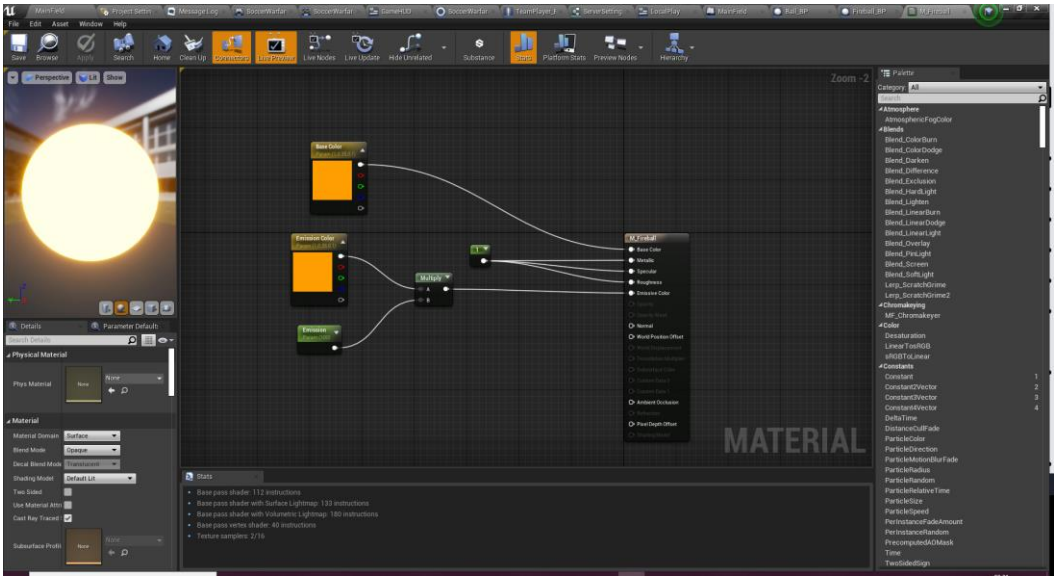

Figura 64 - Material da Fireball

No próximo passo, seguiu-se a implementação da habilidade no jogador. Criaram-se propriedades na classe C++ do jogador para receber o Asset (Blueprint) da nova habilidade. Feito isto, adiciona-se a parte lógica para instanciar a habilidade.

```
protected:

    UPROPERTY(EditAnywhere, Category = "Spawning")
        TSubclassOf<class AFireball> FireballSpawn;
    UPROPERTY(EditAnywhere, Category = "Spawning")
        TSubclassOf<class ABeam> BeamSpawn;
    UPROPERTY(EditAnywhere, Category = "Spawning")
        TSubclassOf<class ABarrier> BarrierSpawn;
    UPROPERTY(EditAnywhere, Category = "Spawning")
        TSubclassOf<class AHealBomb> HealBombSpawn;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Stats", Meta = (BlueprintProtected = "true"))
        float KickCooldown;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Stats", Meta = (BlueprintProtected = "true"))
        float FireballCooldown;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Stats", Meta = (BlueprintProtected = "true"))
        float FireballCost;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Stats", Meta = (BlueprintProtected = "Spawning"))
        float FireballSpawnDistance;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Stats", Meta = (BlueprintProtected = "Spawning"))
        float BeamSpawnDistance;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Stats", Meta = (BlueprintProtected = "true"))
        float BeamCooldown;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Stats", Meta = (BlueprintProtected = "true"))
        float BeamCost;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Stats", Meta = (BlueprintProtected = "true"))
        float HealBombCooldown;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Stats", Meta = (BlueprintProtected = "true"))
        float HealBombCost;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Stats", Meta = (BlueprintProtected = "true"))
        float BarrierCooldown;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Stats", Meta = (BlueprintProtected = "true"))
        float BarrierCost;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Stats", Meta = (BlueprintProtected = "Spawning"))
        float BarrierSpawnDistance;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Stats", Meta = (BlueprintProtected = "true"), replicated)
```

Figura 65 - Atributos a serem alterados em blueprint

```cpp
void ATeamPlayer::Fireball()
{
    if (!CanAct() || MP < FireballCost) return;
    if (FireballSpawn != nullptr)
    {
        UWorld* const World = GetWorld();
        if (World)
        {
            FActorSpawnParameters SpawnParams;
            SpawnParams.Owner = this;
            SpawnParams.Instigator = Instigator;
            FVector SpawnLocation = GetActorLocation() + FVector(0.f, 0.f, -50.f) + (GetActorForwardVector() * FireballSpawnDistance);
            FRotator SpawnRotation;
            SpawnRotation = GetActorRotation();
            AFireball* const SpawnedFireball = World->SpawnActor<AFireball>(FireballSpawn, SpawnLocation, SpawnRotation, SpawnParams);
            MP -= FireballCost;
            Cooldown(FireballCooldown);
        }
    }
}
```

Figura 66 - Instanciação da Fire Ball

```cpp
void ATeamPlayer::Special_Implementation()
{
    switch (TeamRole) {
    case ETeamPlayerRole::EDPS:
        Fireball();
        break;
    case ETeamPlayerRole::EHEALER:
        HealBomb();
        break;
    case ETeamPlayerRole::EMAGE:
        Beam();
        break;
    case ETeamPlayerRole::ETANK:
        Barrier();
        break;
    default:
        break;
    }
}
```

Figura 67 - Método da habilidade especial

Falta apenas acrescentar o input para esta habilidade e uma animação. Procedeu-se à blueprint e criação de uma variável do tipo booleano e o input handler, tendo em consideração a replicação para o multijogador em rede. A variável booleana será utilizada como RepNotify (o que será explicado no capítulo de rede) que quando atualizada irá executar uma função na blueprint.

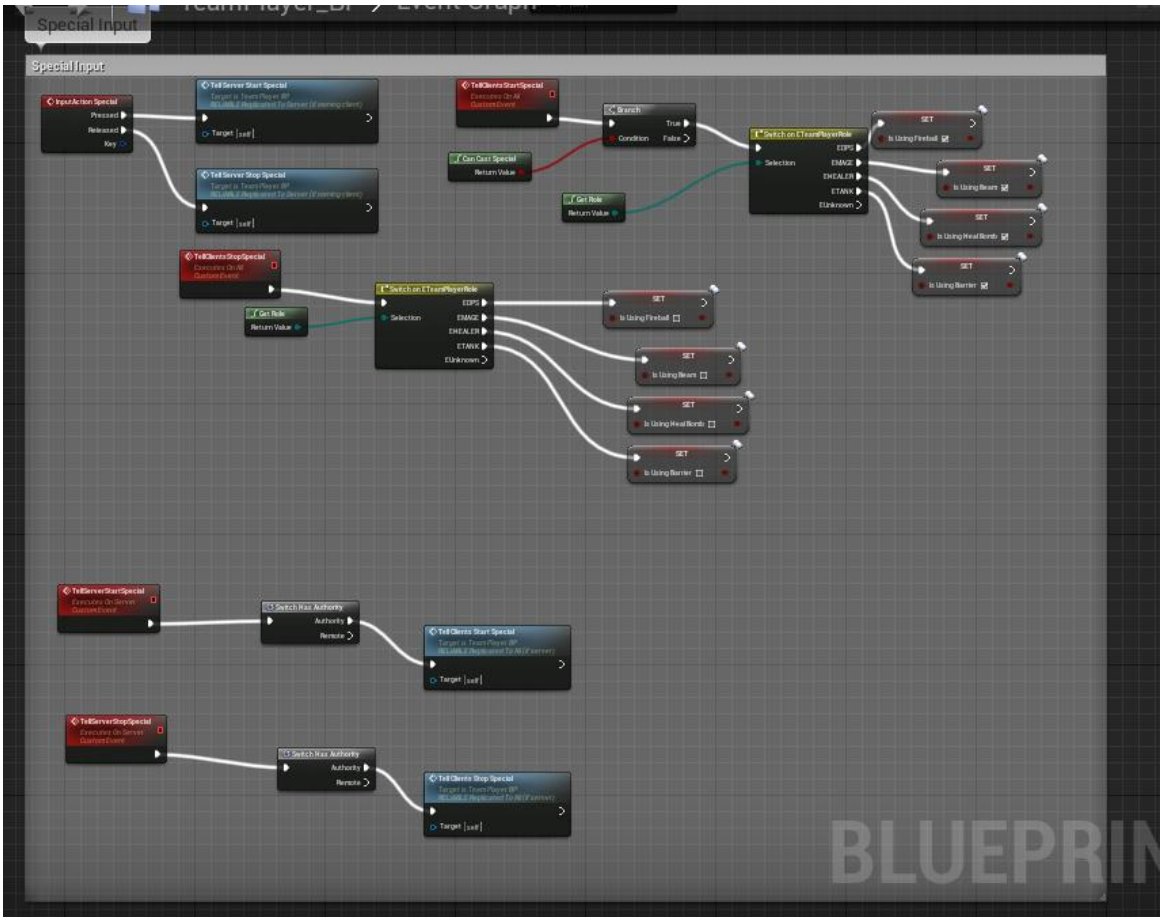

Figura 68 - Input da habilidade especial

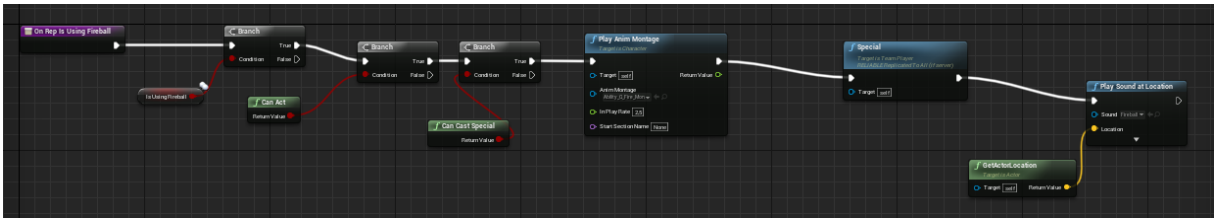

Figura 69 - Função RepNotify da Fireball, irá reproduzir uma animação do humanoide e assim que concluída irá instanciar a Fireball

Feito isto, o processo está concluído. Contudo, se a AI tiver de utilizar esta habilidade terá de ser criada uma task que será executada a partir da behavior tree (será explicado no capítulo de Inteligência Artificial). Neste caso, como se trata de uma habilidade especial de *role*, já tinha sido criada esta task anteriormente que utiliza a habilidade especial do seu *role*.

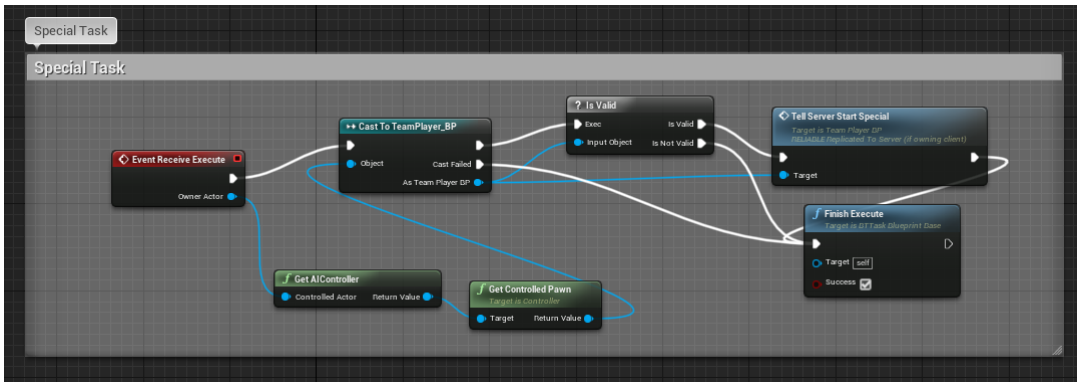

Figura 70 - Behavior Tree Task da habilidade especial

Disparo

A blueprint herda da classe `Bullet` que por sua vez herda da classe `Projectile`, pois trata-se de um projétil.

Esta habilidade é universal, todos as unidades do jogo a podem utilizar. Isto consome as balas e serve para causar dano a unidades da equipa oposta. Trata-se de uma mesh com colisão e projectile movement com uma velocidade. A bala é destruída assim que embate com um objeto sólido.

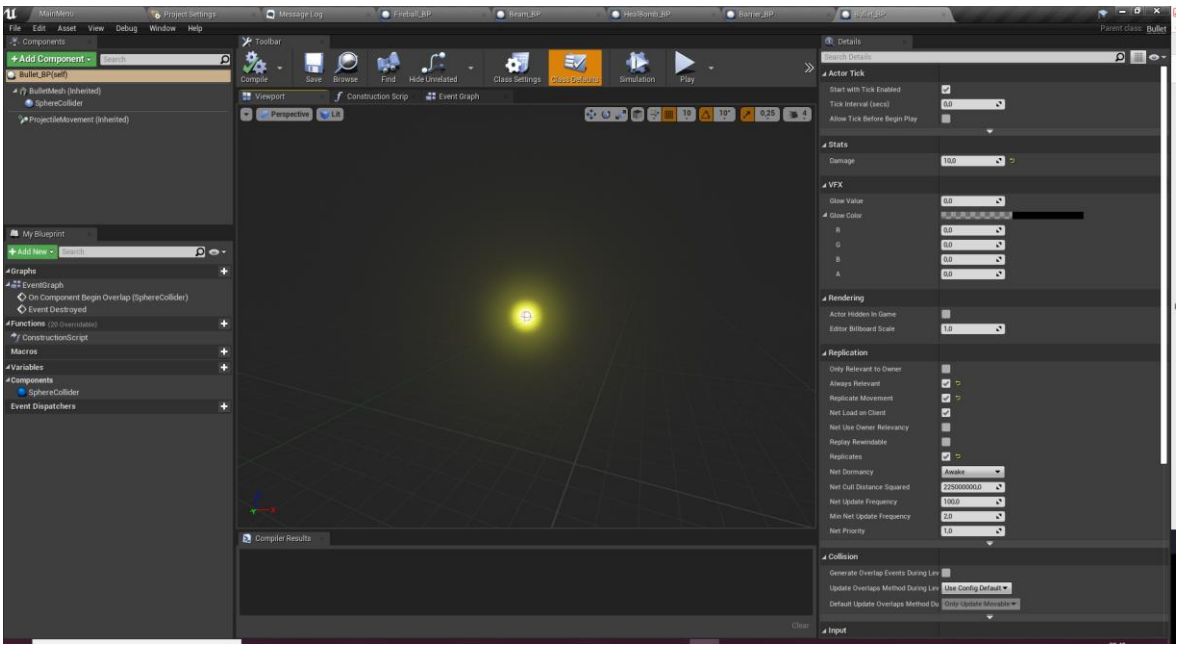

Figura 71 - Blueprint da bala

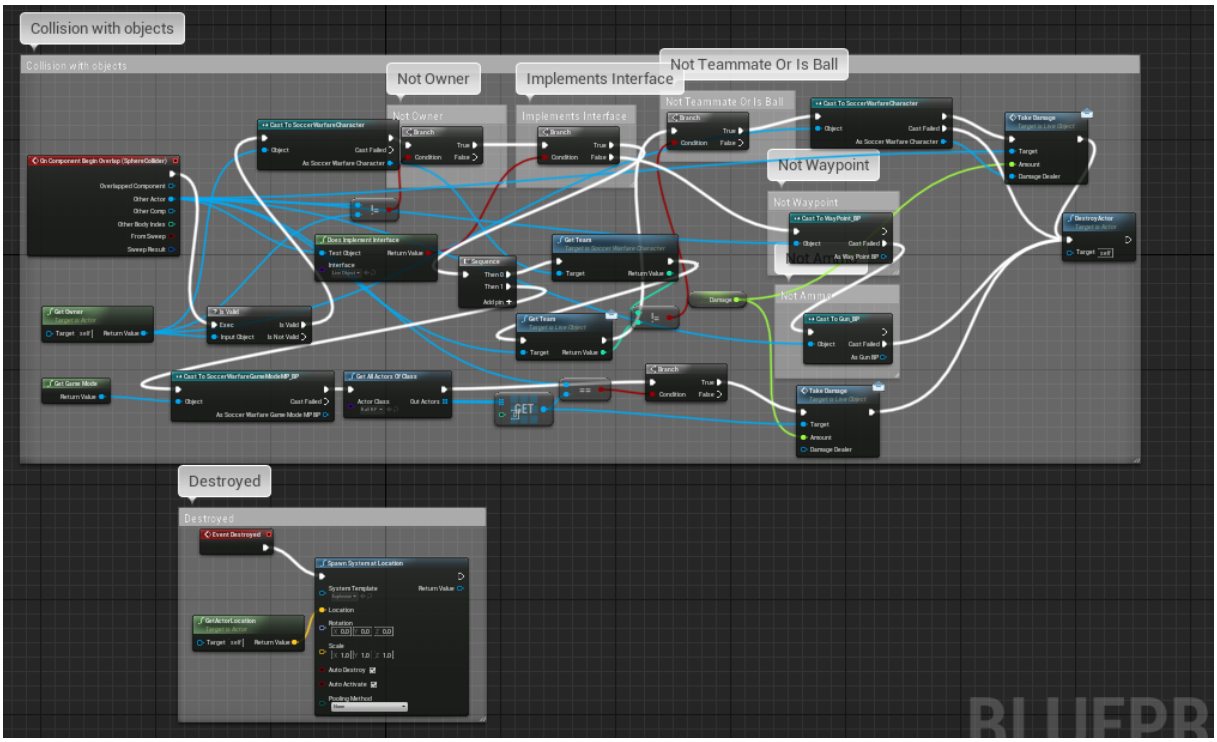

Figura 72 - Interação da bala com unidades

Tabela 3 - Atributos do disparo

Cooldown	1 segundo
Dano	20

Caixa de balas

A blueprint herda da classe `Gun`, essa classe herda da classe `Actor`. Esta blueprint apenas contém uma static mesh. A classe contém o número de balas armazenado. Ao entrar em contato com uma unidade, soma a quantidade de balas ao número de balas dessa unidade.

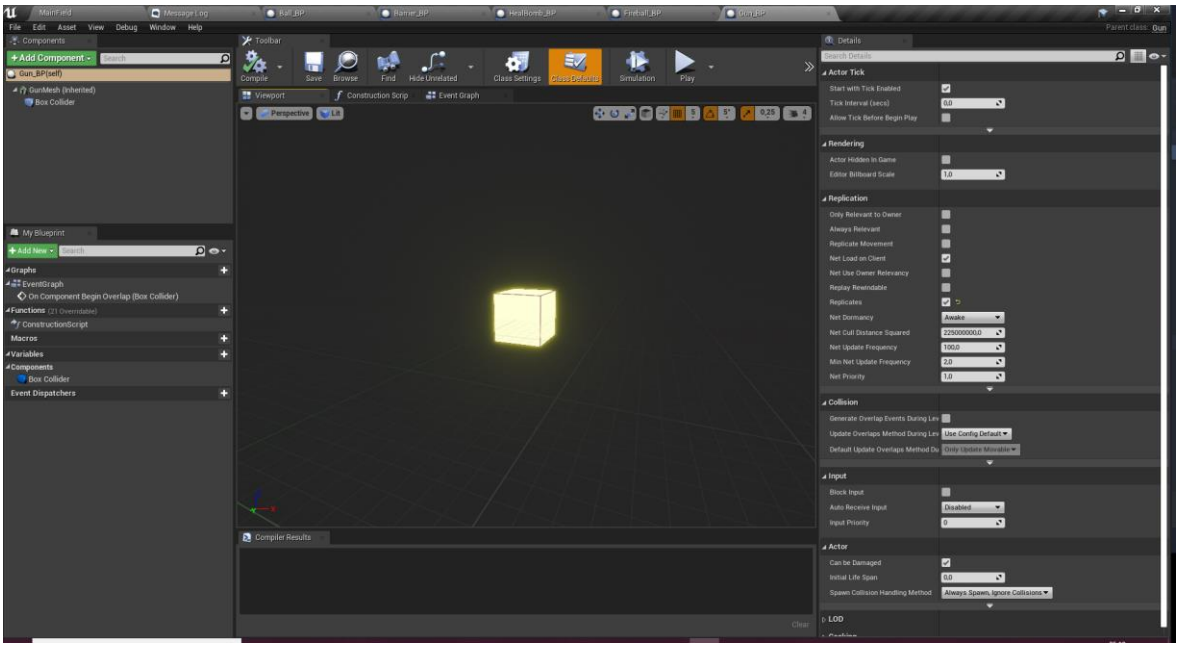

Figura 73 - Caixa de balas

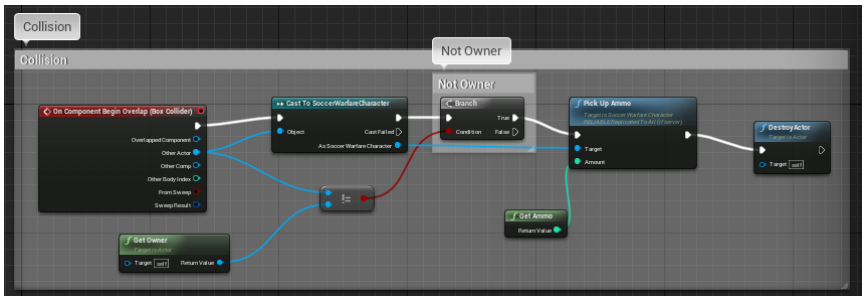

Figura 74 - Interação a da caixa de balas com unidades

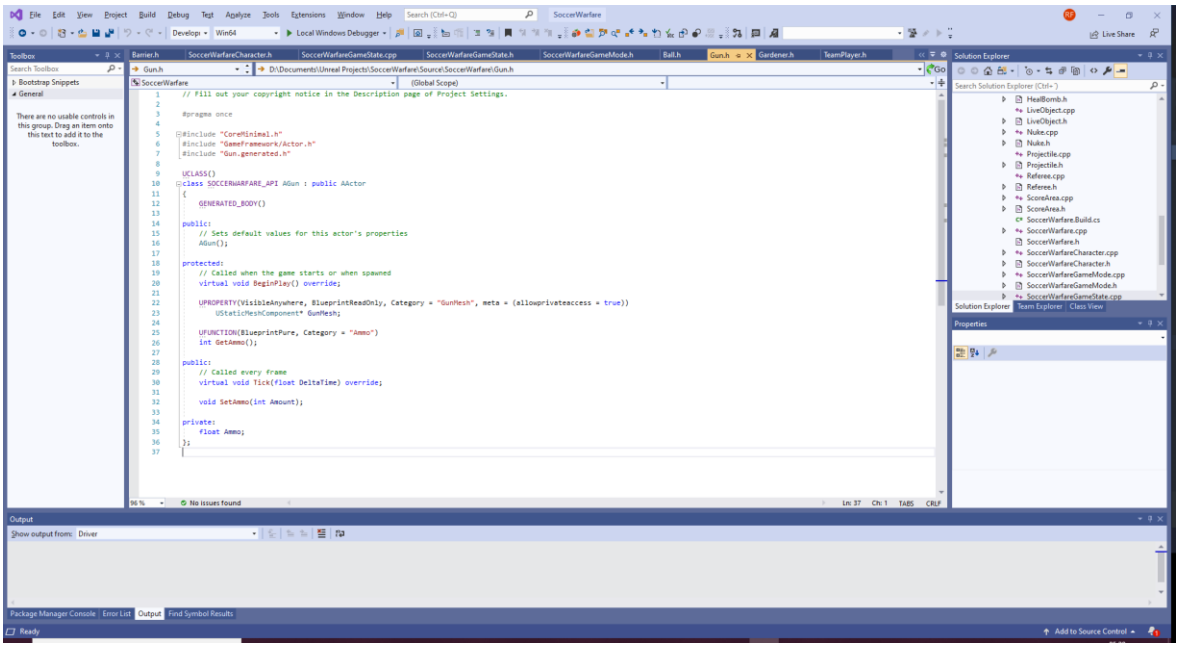

Figura 75 - Ficheiro header da classe da caixa de balas

Kick (Chuto) / Dash

Esta habilidade é comum para todos os jogadores. Serve apenas ou para lançar a bola ou para teletransportar-se para a direção que está virado. Apenas é possível teletransportar se não se possuir a bola.

Tabela 4 - Atributos do Kick / Dash

Cooldown	1 segundo

Fire Ball

Semelhante ao disparo, mas apenas está disponível para jogadores com o *role* DPS. Tem a velocidade de projétil mais lenta que uma bala, mas tem um maior volume, incluindo na sua colisão. Esta habilidade tem custo nos pontos de energia, ao invés de custo a consumir balas. A Fireball é destruída assim que embate com um objeto sólido. A classe `Fireball` herda da classe `Projectile`.

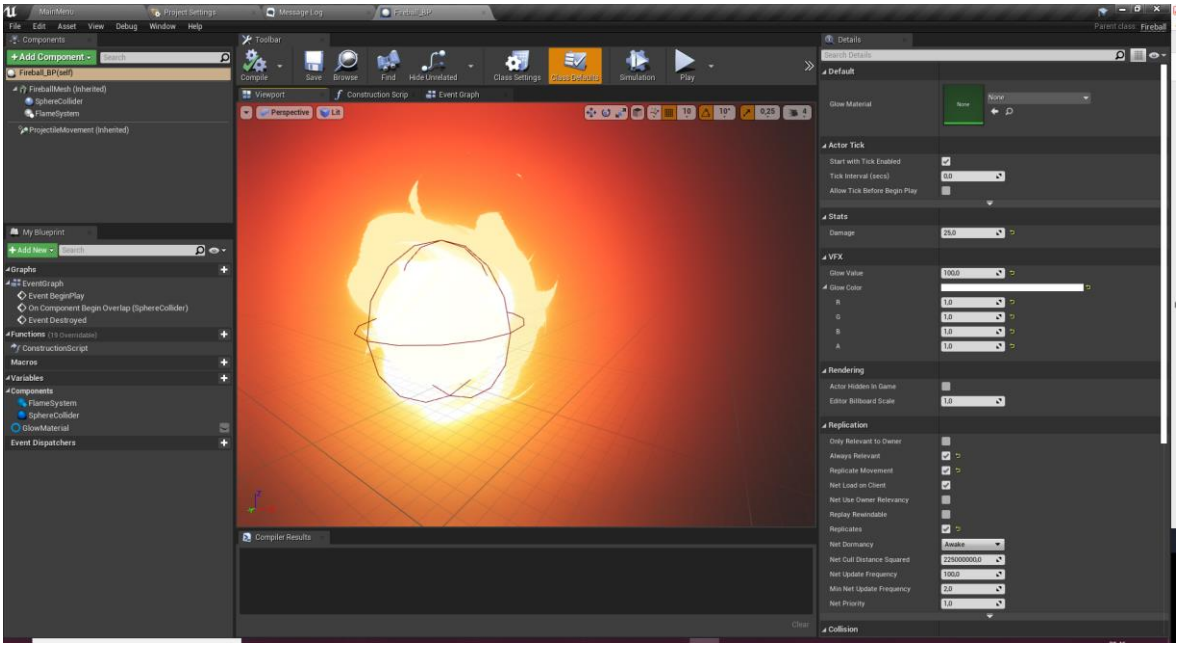

Figura 76 - Fire Ball

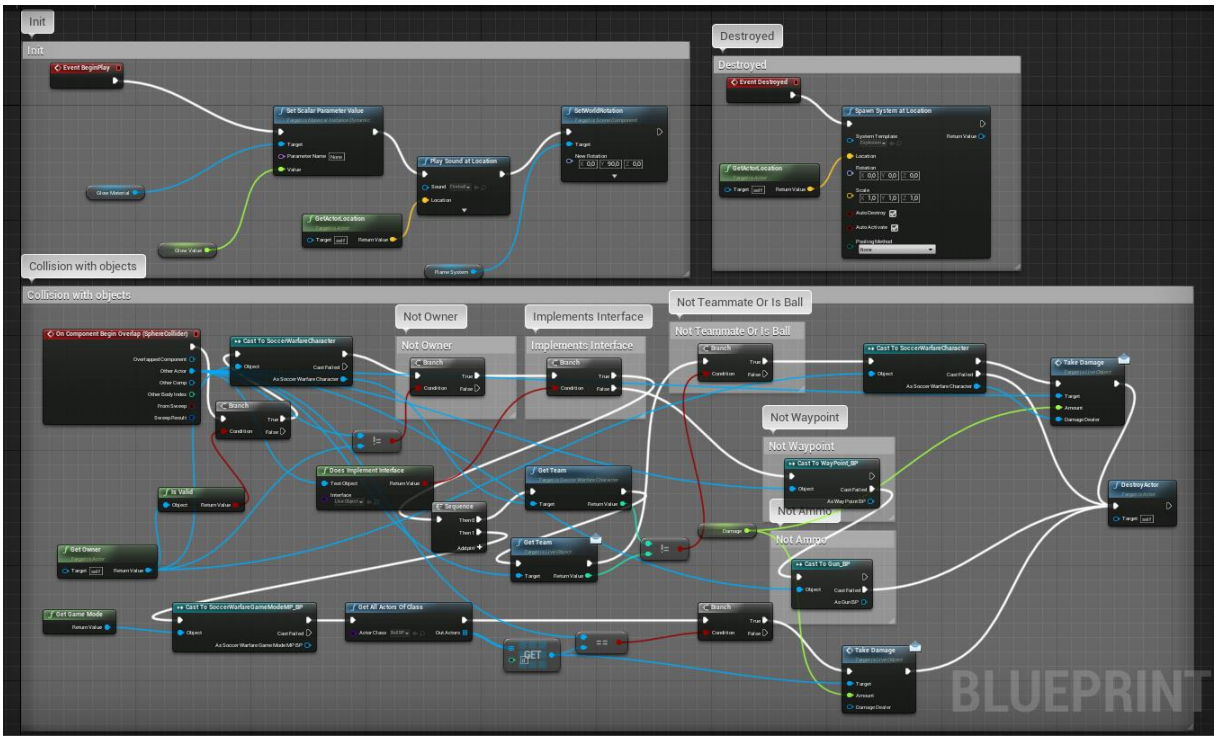

Figura 77 - Interação da fire ball com unidades

Tabela 5 - Atributos da Fire Ball

Dano	25
Cooldown	1 segundo
Custo de Energia	25

Beam

Esta habilidade causa dano uma vez a todas as unidades inimigas que se encontrem dentro do volume de colisão, está disponível para jogadores com o *role* Mage. A classe `Beam` herda da classe `Actor`.

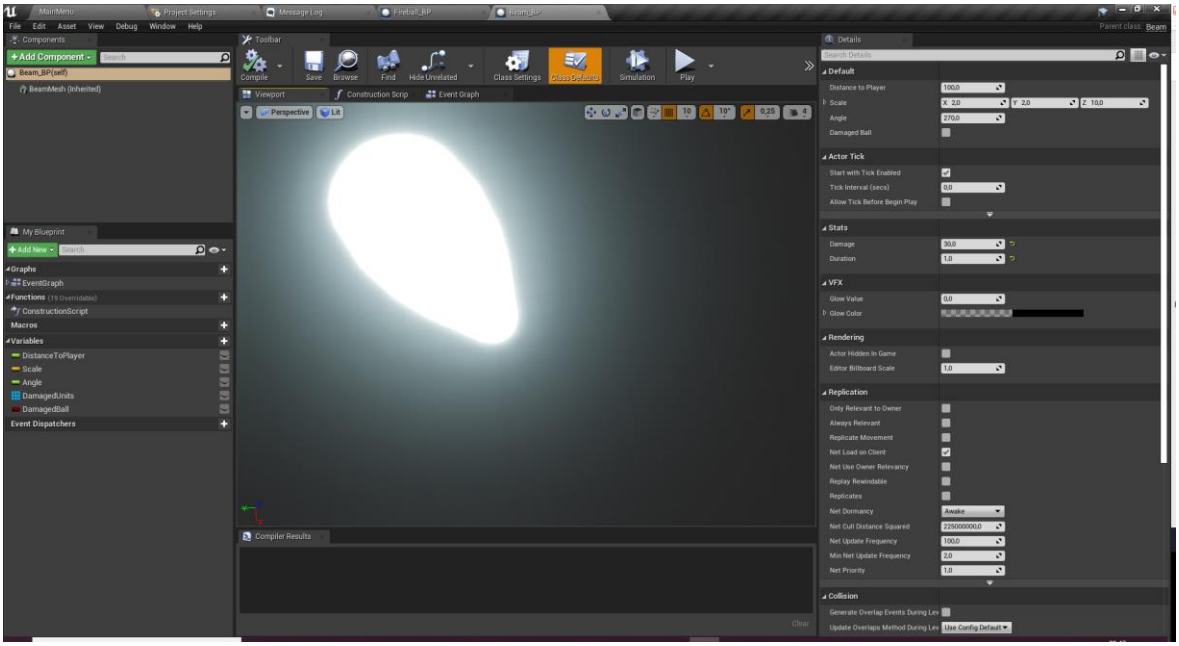

Figura 78 – Beam

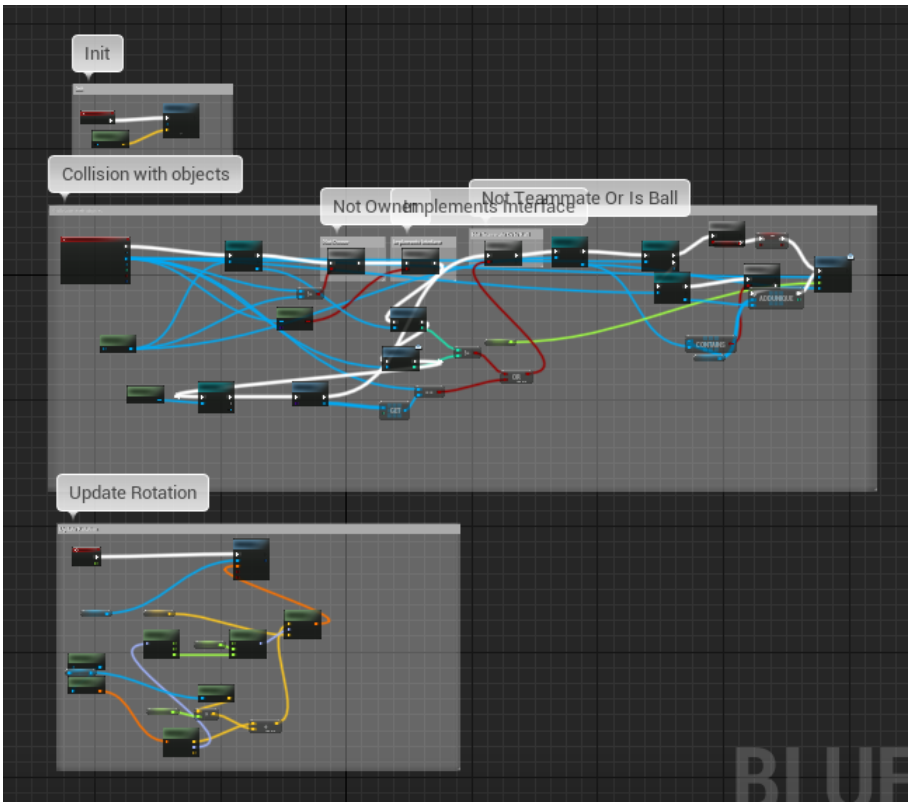

Figura 79 - Interação do beam com unidades

Tabela 6 - Atributos do Beam

Dano	40
Cooldown	5 segundos
Custo de Energia	50

Heal Bomb

Esta habilidade cura uma vez todas as unidades da equipa que se encontrem dentro da esfera de colisão. Está disponível para jogadores com o *role* Healer. A classe `HealBomb` herda da classe `Actor`.

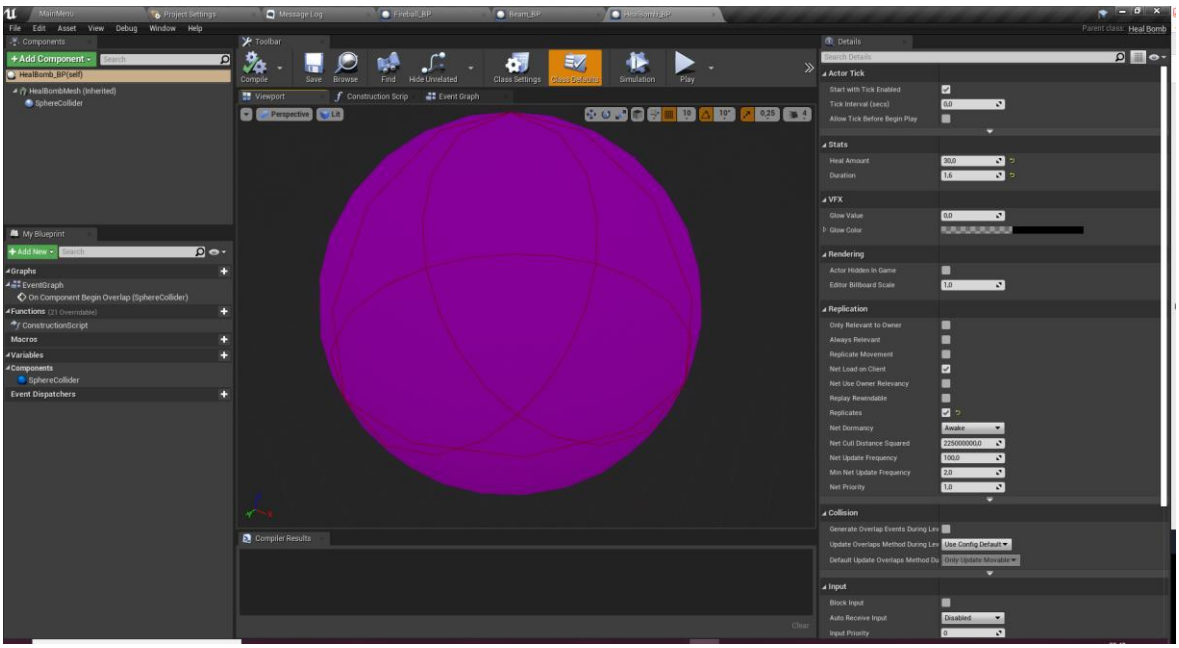

Figura 80 – Heal bomb

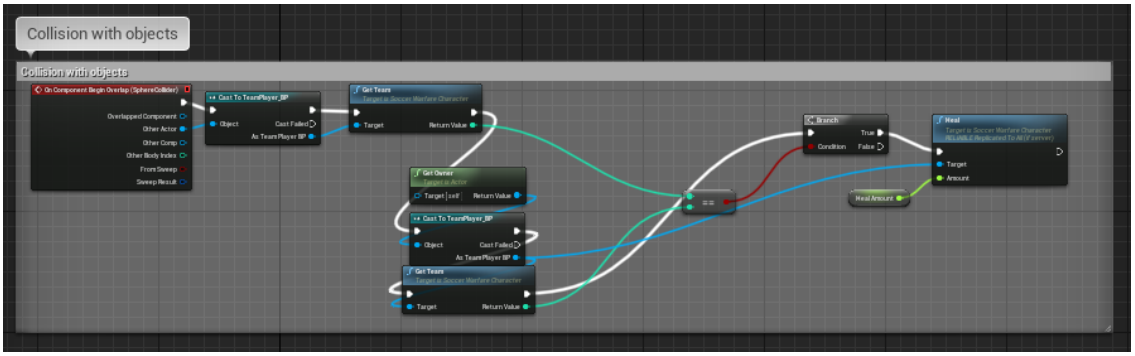

Figura 81 - Interação da heal bomb com jogadores

Tabela 7 - Atributos da Heal Bomb

Cura	30
Cooldown	2 segundos
Custo de Energia	50

Barrier

Esta barreira serve para destruir todos os projéteis, está disponível para jogadores com o *role* Tank. A classe `Barrier` herda da classe `Actor`.

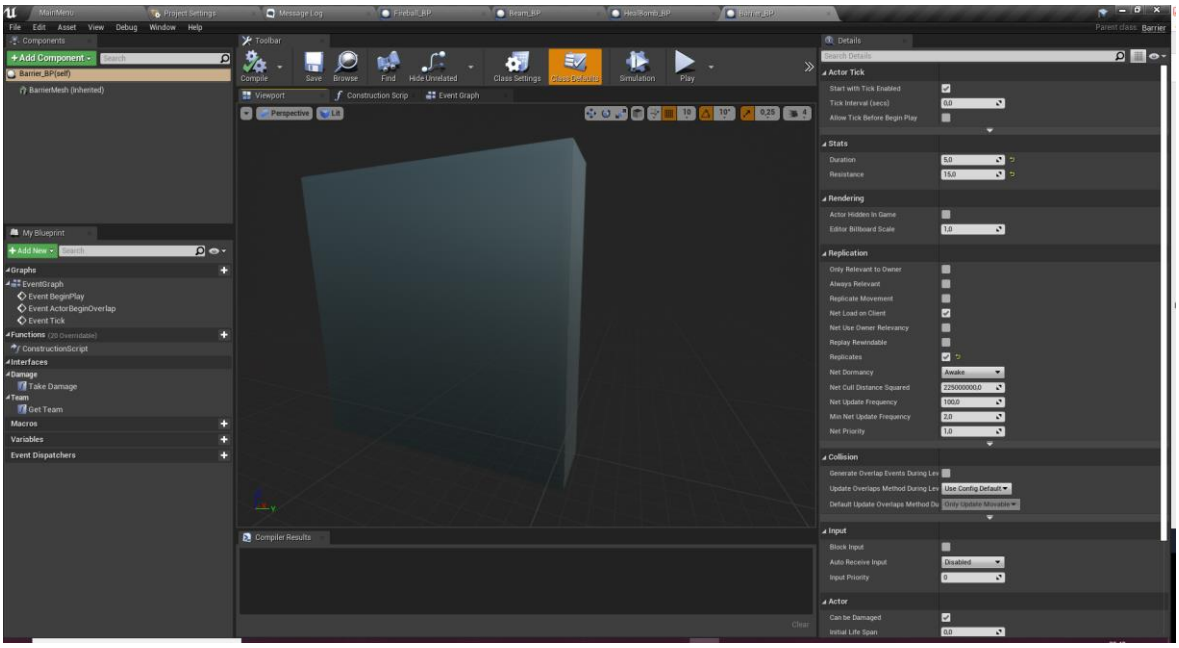

Figura 82 - Barrier

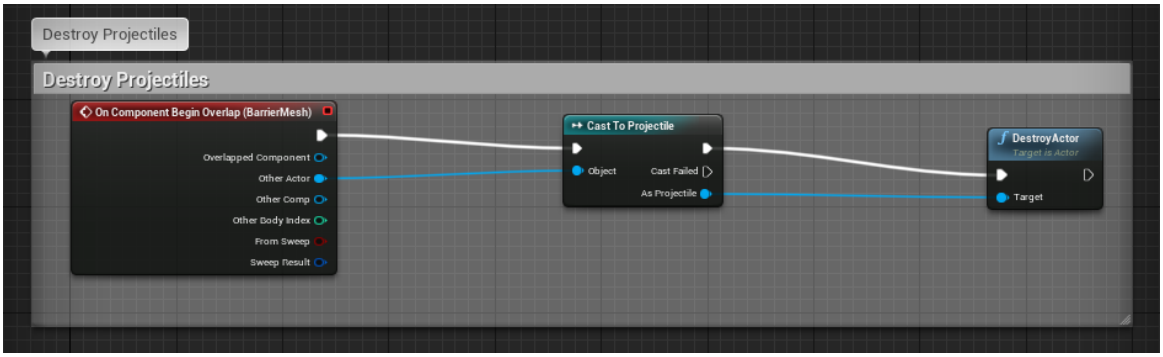

Figura 83 - Interação da barrier com projéteis

Tabela 8 - Atributos da Barrier

Duração	5 segundos

Cooldown	1 segundo
Custo de Energia	5

4.7.3. Árbitro

Esta unidade é controlada por inteligência artificial. Persegue a bola até uma certa distância. Irá retaliar se for atacado. Quando morto, o objetivo de jogo muda para sobrevivência.

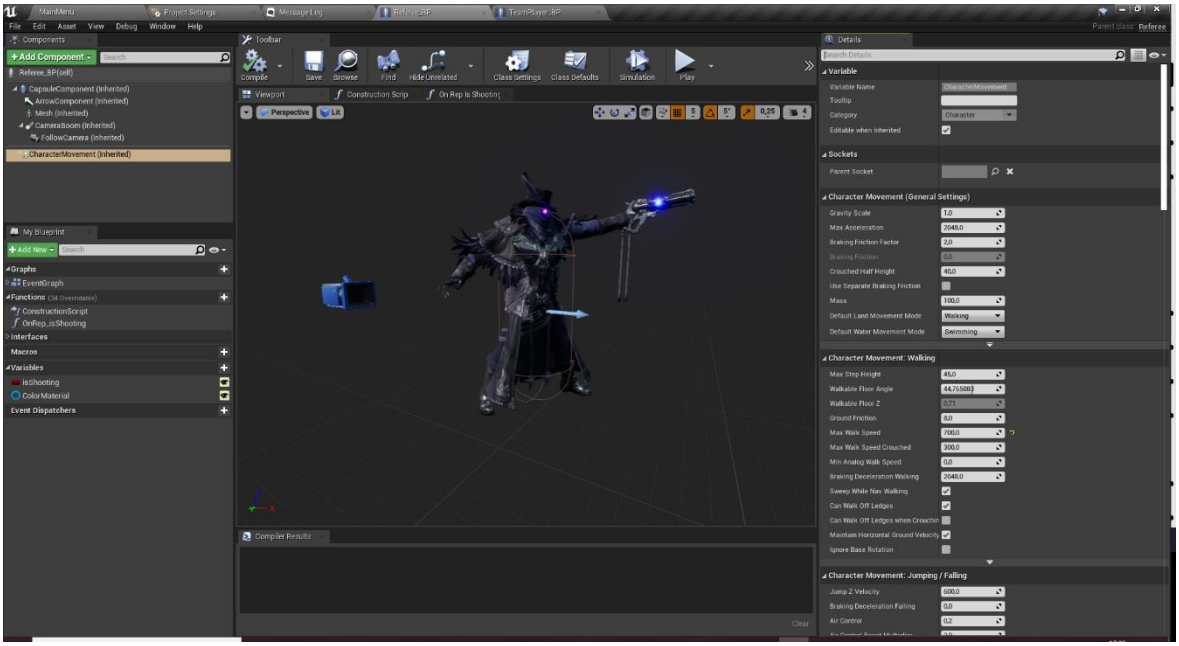

Figura 84 - Árbitro

Tabela 9 - Atributos do árbitro

Vida	200/200
Balas	100

4.7.4. Jardineiro

Esta unidade é controlada por inteligência artificial. Persegue a bola até uma certa distância. Retalia se for atacado.

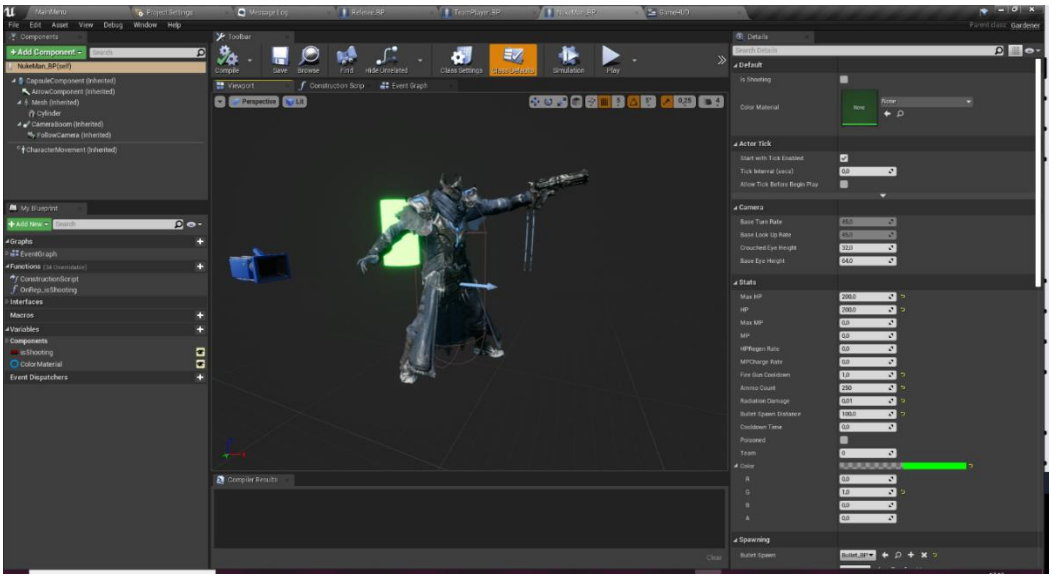

Figura 85 – Jardineiro

Tabela 10 - Atributos do Jardineiro

Vida	200/200
Balas	250

Nuke

Habilidade passiva do jardineiro, esta é ativa assim que este morre, mata todas as unidades e destrói a bola se estiver dentro da sua esfera de colisão e desativa a cura passiva para todas as unidades. Causa dano ao longo do tempo a estas e à bola. A habilidade Heal Bomb reverte o efeito desta habilidade.

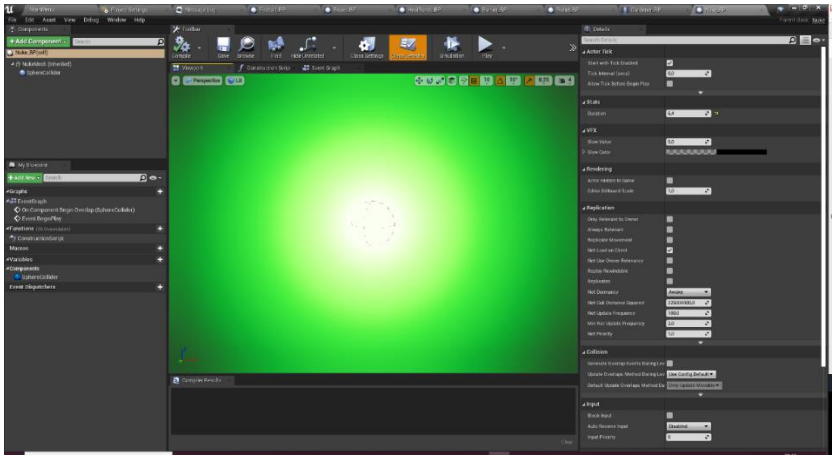

Tabela 11 - Atributos do Nuke

Dano de Radiação	1% da vida máxima/energia da bola por segundo

Figura 86 - Nuke

5. Inteligência Artificial

A inteligência artificial escolhida foi a behavior tree (que será explicado mais à frente), cada jogador de equipa que não seja controlado por um humano irá ser possuído por um controller que herda da classe `DetourAIController` que ativará uma árvore de decisão auxiliada por uma blackboard e uma Nav Mesh (Grelha de Navegação). A blackboard irá conter variáveis que poderão ser alteradas pelo controller. A árvore de decisão contém condições baseadas na blackboard para executar tarefas.

As behavior trees foram escolhidas para a implementação de inteligência artificial devido à sua facilidade suportando uma grande quantidade de alterações feitas na jogabilidade ao longo do projeto, permitindo o controlo total e tornando possível testar sempre as alterações efetuadas. No caso de utilizar uma rede neuronal seria necessário treinar a mesma e, qualquer alteração no jogo requeria um novo treino ou até uma nova rede neuronal. O número de variáveis em cada jogo é demasiado grande, o que requer muito tempo de treino, não sendo de todo prático. Podem ocorrer casos em que a rede fique overfitted para algumas situações e underfit para outras. Isto é uma das razões pela qual se utiliza behavior trees na maioria dos jogos.

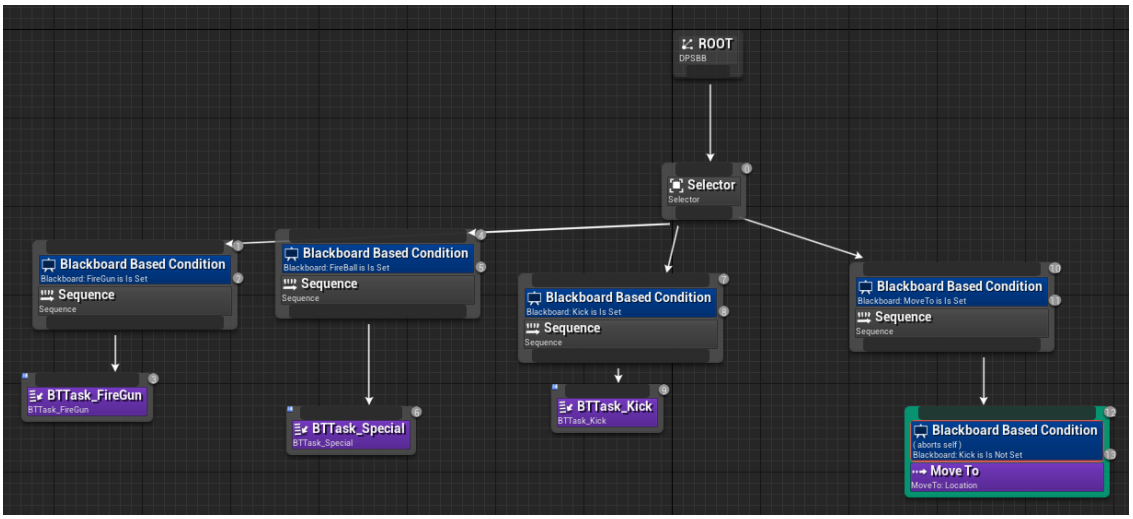

Figura 87 - Behavior Tree

A inteligência artificial necessita de uma NavMesh para a navegação do Pawn quando ordenado o seu movimento para uma determinada posição. Foi arrastada uma para o nível, esta está representada a verde.

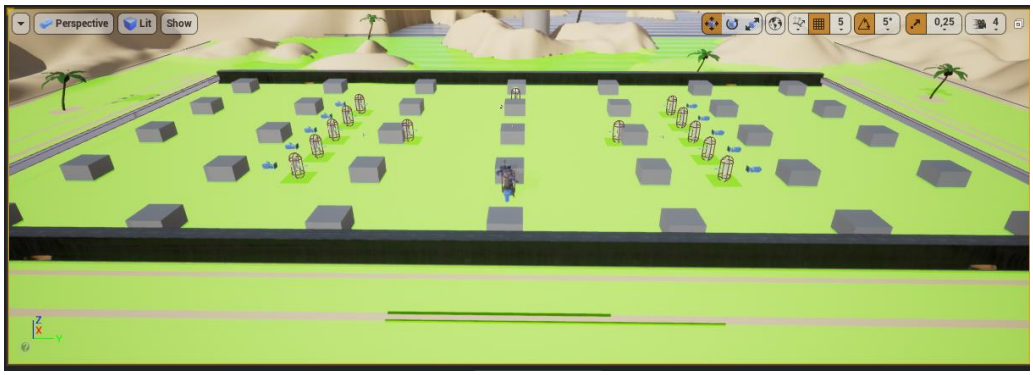

Figura 88 - NavMesh do nível

5.1.Cérebro Humano

O cérebro humano é um órgão muito complexo e não é conhecido o seu funcionamento totalmente. O cérebro humano é composto por 100 biliões de neurónios em média, cada neurónio pode estar ligado a 10000 neurónios em média, tornando o número de sinapses aproximadamente entre 100 triliões a 1000 triliões. Os neurónios são a unidade fundamental do tecido do sistema nervoso. Um neurónio orgânico tem 3 componentes: o axónio, dendrites e a soma. As dendrites são ramificações, o axónio trata-se de um ramo isolado maior. As dendrites ligam-se a axónios de neurónios através de junções denominadas de sinapses.

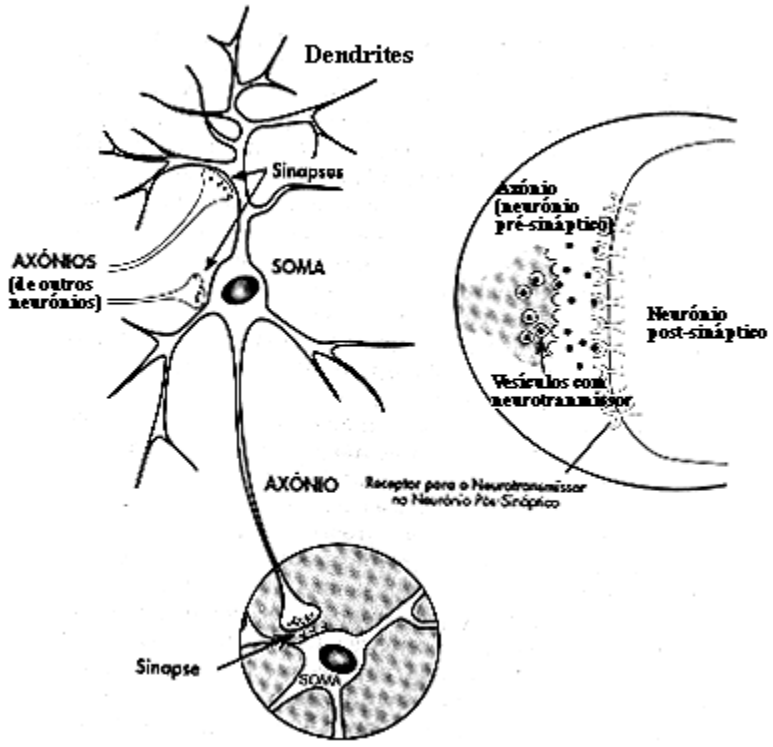

Figura 89 - Neurónio

Os sinais propagam-se entre neurónios através de uma reação química complexa que faz as sinapses produzirem substâncias químicas que entram nos neurónios através das dendrites. Isto altera a voltagem do neurónio. Se a voltagem exceder um determinado limite, um impulso elétrico será enviado para o axónio que o irá espalhar pelas suas ramificações, transmitindo sinais elétricos para outros neurónios.

Foi observado que as conexões mais utilizadas tornam-se mais fortes e que neurónios podem formar novas conexões com outros neurónios, pensa-se que estes mecanismos levam a aprendizagem.

5.2.Redes Neuronais (Neural Networks)

As redes neuronais tentam emular algumas características do cérebro em termos de receção de dados, ações possíveis de acordo com os dados de entrada e aprendizagem. Consistem em neurónios artificiais interligados onde a sua funcionalidade é semelhante a um neurónio biológico. Estes comunicam entre si enviando sinais através de um grande número de ligações, ocorrendo depois o processamento de informação. Cada neurónio tem ligações de input (entrada) e output (saída), alguns dos neurónios são fronteiras, ou seja, uns recebem dados para a rede neuronal, e outros enviam os dados de saída da rede.

Cada ligação tem um peso associado, estes pesos são a "memória" da rede neuronal, os pesos são modificados durante o processo de treino ou de aprendizagem.

Na criação de uma rede neuronal é decidido o tipo de neurónios e a sua quantidade. Cada neurónio recebe sinais das conexões de input, calcula o valor de ativação que depois é enviado para as conexões de output. O cálculo desse valor é feito em 2 fases, a soma dos inputs multiplicados individualmente pelo seu peso (valor decimal de 0 a 1) e depois esse valor é utilizado na função de ativação.

Uma maneira de treinar a rede, é calcular a diferença entre o output alvo e o output previsto, com esta margem de erro é depois ajustado o peso das ligações associadas, feito isto repetidamente, a rede acabará por atingir uma margem de erro muito pequena.

Existe um problema que pode ocorrer se o treino da rede não for diversificado, a rede ficará muito apta para certas situações (overfit) e muito pouco apta para outras (underfit). Um exemplo que se poderia aplicar neste jogo seria, a rede apenas jogar contra um jogador que apenas seguisse a bola e tentasse pontuar, logo a rede passada n iterações chegaria a estar muito otimizada para lidar com essa situação, mas no entanto se encontrasse um jogador mais

agressivo que tentava jogar para o objetivo de sobrevivência, a rede, sem treinamento nenhum para esta situação, não teria como reagir se a bola fosse destruída.

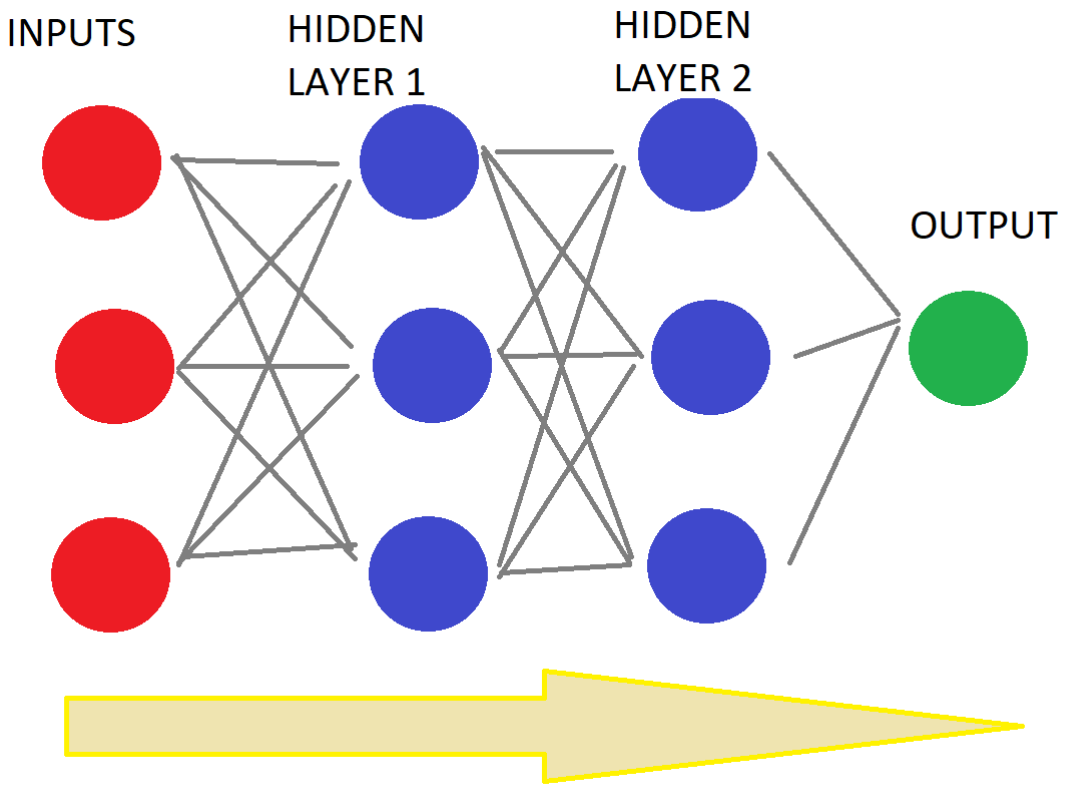

Figura 90 – Exemplo Rede neuronal de 2 camadas

Pela razão referida acima, não é prática a utilização de redes neuronais em jogos se não existir uma infraestrutura capaz de obter dados de jogo e jogadores e depois processar tudo e realizar treinos a partir desses dados. Estas infraestruturas requerem uma capacidade enorme de armazenamento e processamento, muitos cores e threads, ram, placas gráficas, nomeadamente Nvidia devido à enorme quantidade de CUDA cores (estes são muito utilizados na área de machine learning, não só para jogos como também para a ciência).

A Valve neste momento utiliza machine learning no seu projeto interno VACNet, que tem como objetivo detetar jogadores não legítimos e submetê-los ao sistema de tribunal do Counter-Strike Global Offensive. O Overwatch, para além da VACNet também é utilizado para o Trust Factor (TF), o fator de confiança dos jogadores, utilizado no matchmaking para encontrar jogos para jogadores com TF semelhante.

5.3.NavMesh

Uma Nav Mesh Bounds Volume, é um volume responsável por encontrar superfícies onde a AI se poderá deslocar do seu local ao destino, esta pode cobrir um determinado volume, usa-se este volume em zonas onde a AI se pode deslocar.

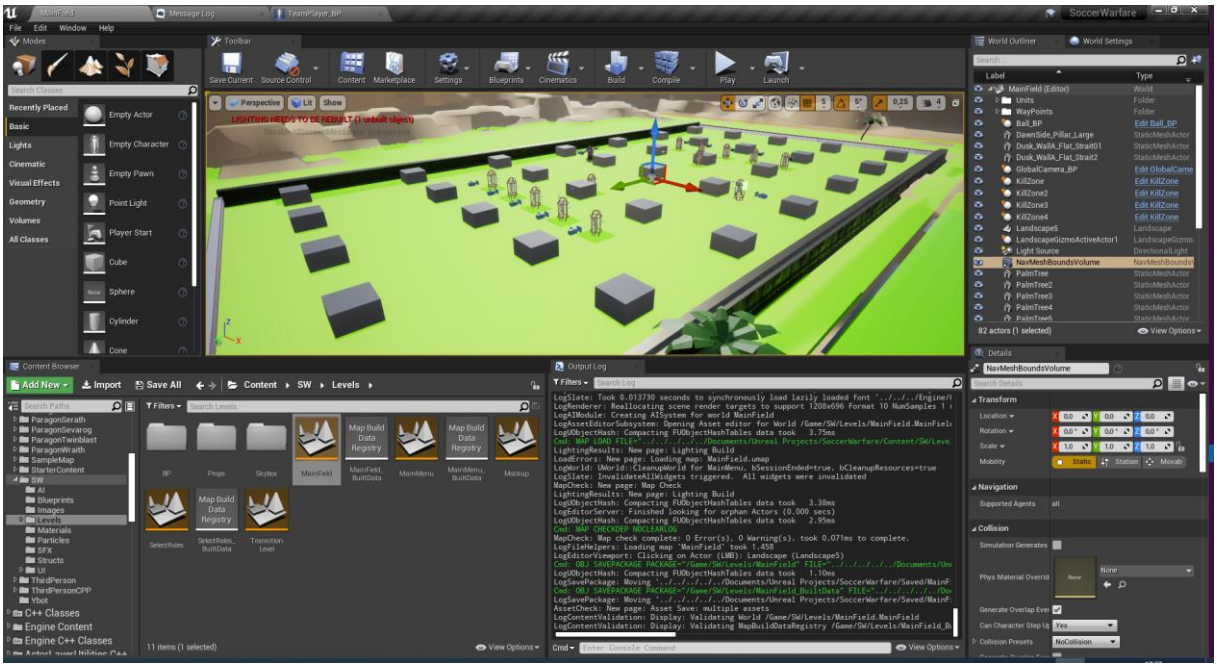

Figura 91 - NavMeshBoundsVolume

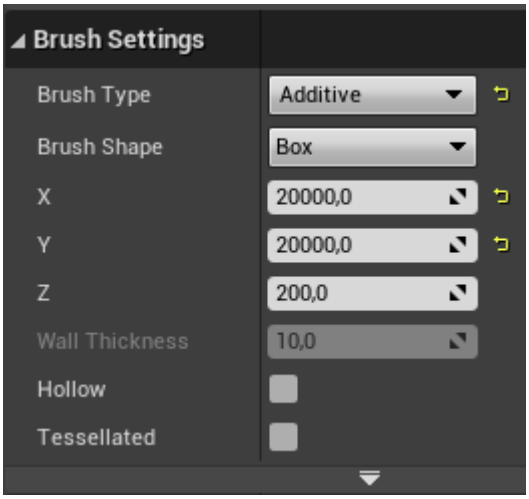

Figura 92 - Definições da NavMesh

5.4.Blackboard

No sentido tradicional a Blackboard é uma base de conhecimento comum que é iterativamente atualizada por um grupo diverso de fontes de conhecimento especialistas, começando com a especificação de um problema e terminando com a solução. Cada fonte de conhecimento atualiza a blackboard com a solução parcial quando os seus requisitos internos alinham com o estado da blackboard.

Uma Blackboard no Unreal Engine pode ser considerada como um dicionário com pares chave-valor públicos. Os seus dados podem ser alterados e acedidos, por múltiplos AI

Controllers. Podem ser utilizados em conjunto com ou sem Behavior Tree e também com ou sem blackboard.

Neste projeto são utilizadas pois há condições necessárias para executar qualquer tarefa na Behavior Tree. Como estas condições não são de raiz, são condições para comportamentos específicos dos personagens, é necessário um local para as armazenar e a Behavior Tree para ir buscar valores personalizados necessita de uma Blackboard.

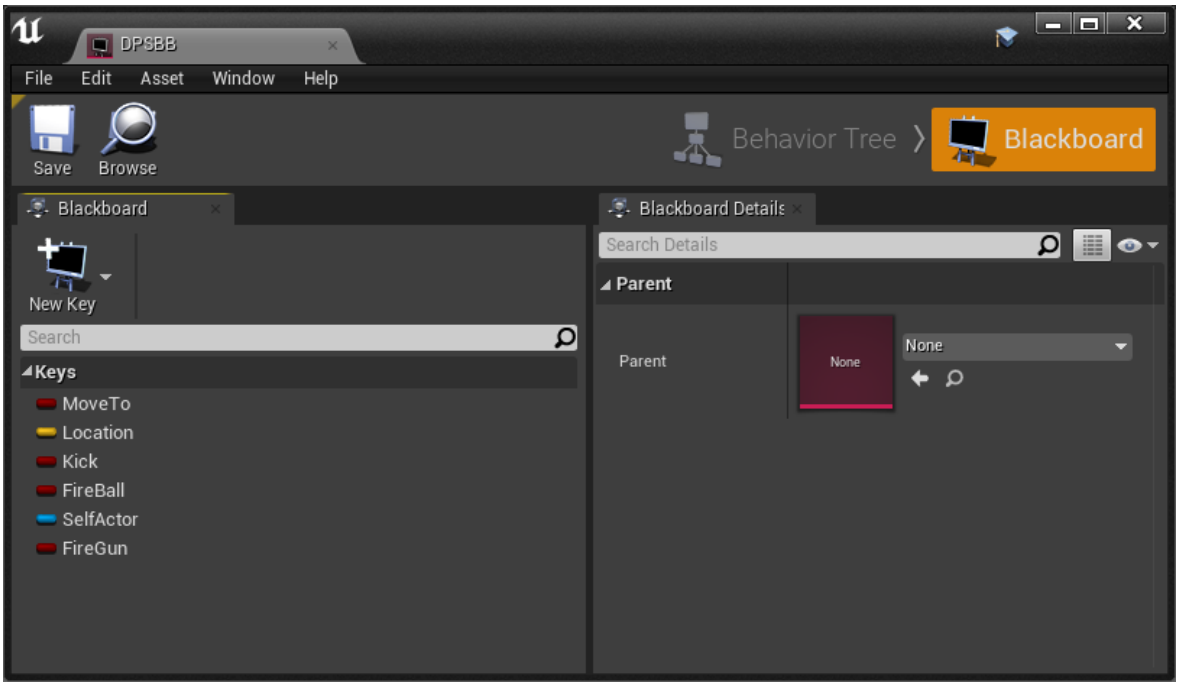

Figura 93 - Blackboard

5.5.Behavior Tree

Uma behavior tree, pode ser caraterizada por uma árvore com uma raiz e com ramos que podem ligar a outros ramos. Estes ramos eventualmente chegarão ao seu destino: a folha. Esta, não liga a nada. São muito utilizadas em jogos, robótica e sistemas de controlos. Uma árvore de decisão é executada de cima a baixo, esquerda para a direita.

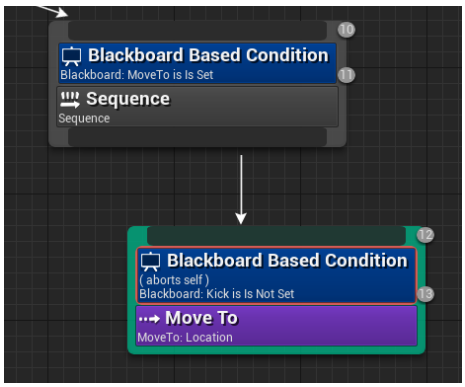

Figura 94 - Task numa behavior Tree

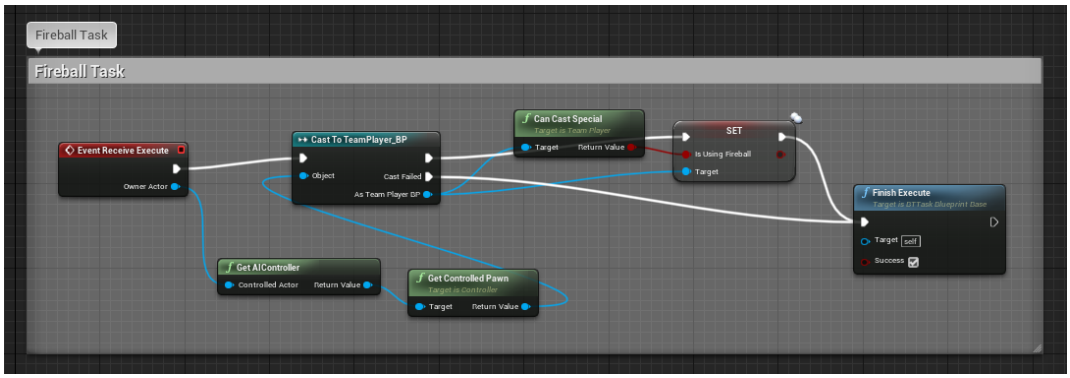

Figura 95 - Task de behavior tree

5.5.1. Behavior Trees em Unreal Engine

As Behavior Trees em Unreal Engine funcionam à base de eventos invés de estar ativamente a cada tick a verificar valores, escuta passivamente por eventos que podem ativar mudanças na árvore.

No modelo tradicional da behavior tree, as condicionais são Tasks que retornam sucesso ou falha. Já no UE4, as condicionais estão presentes na forma de Decorator.

5.5.2. Nodes

Root – Apenas existe 1 por behavior tree, é o ponto de início, apenas tem uma ligação, não se pode ligar Decorators nem Services a este nó.

Figura 96 - BT Root

Composite – Estes são blocos de construção para os ramos da behavior tree, podem conter Decorators e Serviços.

Selector (Composite) – Executa os nós filho de esquerda para a direita, se um nó filho devolver o valor de sucesso o Selector para.

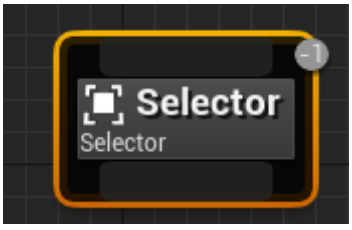

Figura 97 - BT Selector

Sequence (Composite) – Executa os nós filho de esquerda para a direita, se um nó filho devolver o valor de fracasso a Sequence para.

Figura 98 - BT Sequence

Simple Parallel (Composite) – Permite uma Task principal única para ser executada em paralelo com uma árvore inteira abaixo. Quando a Task principal termina, a definição em Finish Mode determina se o nó deve terminar imediatamente abortando o segundo nó, ou se deve aguardar que a segunda Task termine. Este nó tem 2 saídas.

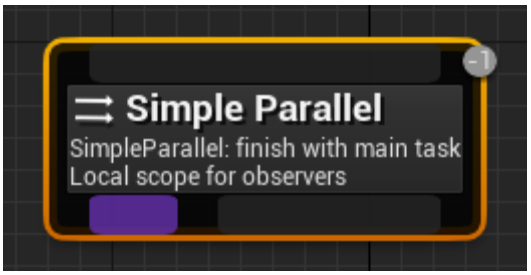

Figura 99 - BT Simple Parallel

5.5.3. Decorators

Decorator – É uma condicional, pode utilizar valores da blackboard. Este é executado dentro de um nó Composite ou Task.

Os Decorators possuem a propriedade Observer Abort:

- None – Não aborta;
- Self – Aborta próprio e qualquer ligação abaixo;
- Lower Priority – Aborta qualquer nó à direita deste;
- Both – Faz ambos o Self e o Lower Priority.

5.5.4. Task

Task – Executa uma tarefa (saltar, mover, etc.) e retorna o resultado de sucesso ou falha. Estes são as folhas da árvore, significa que não têm saídas.

5.5.5. Services

Services – Estão associados a um nó Composite, este pode registar-se para receber um callback. Executarão desde que o seu ramo esteja a ser executado, podem ser utilizados para fazer verificações e atualizações à Blackboard.

5.6.AI Controller e Detour AI Controller

Para alcançar controlo sobre um Pawn é necessária a utilização de um Controller. Neste caso para a AI será o AI Controller. Como há vários Pawns AI no mesmo local a perseguir o mesmo objeto, utilizou-se o Detour AI Controller como classe base, pois este funciona em harmonia com os outros AI Pawns, evitando os obstáculos. Cada personagem e cada *role* recebeu o seu Detour AI Controller, Blackboard e Behavior Tree. Cada Blueprint que herda da classe `DetourAIController` está responsável por ler o estado de jogo, posição da bola, vida e outros atributos, atualiza a Blackboard que por sua vez altera o fluxo da Behavior Tree de acordo com as condições definidas para as tasks.

Figura 100 – DPS *role* Detour AI Controller

O fluxo de AI Possession é o seguinte: o GameMode aplica o role aos jogadores e instância um Detour AI Controller específico dependendo do *role* para cada jogador. Este Detour AI Controller possuí o jogador.

O Controller irá utilizar a Blackboard que lhe pertence e irá executar a Behavior Tree que lhe pertence. Este verifica o estado do jogo, como o objetivo principal, localização da bola, área de marcação inimiga, localização de unidades, etc. Vai alterando os valores da blackboard conforme a situação, como para onde se irá deslocar, se irá utilizar a sua habilidade especial, disparar ou irá chutar.

Quando ocorre a troca de jogador, o jogador que deixa de ser possuído pelo humano, irá verificar o *role* que lhe foi atribuído pelo GameMode e instanciar o Detour AI Controller desse *role*, replicando o que o GameMode faz.

5.7.EQS e AI Perception

O Environment Query System (EQS) é uma ferramenta do sistema de inteligência artificial do UE4. Esta obtém dados do meio presente (por exemplo, o nível). O EQS pode ser questionado pelas EQS Queries. Estas podem ser chamadas por behavior trees. As EQS Queries podem ser constituídas por:

- Generators – Produzem posições ou atores, estes serão testados e pesados;
- Contexts – São utilizados como pontos de referência para testes e Generators.

Em exemplo de caso de uso seria: um jogo de escondidas. Onde é necessário encontrar o local mais próximo e que não seja visível por quem está à procura.

A AI Perception é um componente, que utiliza os sentidos, semelhante a um humano: visão, sentido e audição que servirão como estímulos. É possível alterar os seus parâmetros como a distância, raio de visão, etc. Os estímulos podem disparar eventos. Com isto, por exemplo, os jogadores podem detetar um ruído e mover-se para a localização de onde veio o som.

6. Networking em Unreal Engine e a Steam

O Unreal Engine 4 já vem preparado para a ocasião de desenvolvimento de jogos multijogador em rede, o que facilita imenso visto que não é necessária a implementação de envio e receção de pacotes de rede UDP e TCP. A carga protocolar já está toda implementada, logo os programadores apenas se têm de preocupar com o que vão replicar e otimizações.

Os objetos e variáveis que são partilhados entre clientes e servidores através da rede são considerados replicados. Os dados são enviados do servidor para os clientes. É possível definir o tipo de replicação, sendo:

- Replicated – Replicação normal;
- RepNotify – O cliente executa uma função se o valor replicado for alterado pelo servidor.

6.1. Tags de replicação

Existem 3 tags principais de replicação de funções para utilizar nas macros, estes são:

- Server – Função executada apenas no servidor (dedicado ou o dono da sessão). Destina-se principalmente à lógica de jogabilidade no que toca especialmente a verificações ou a gestão de utilizadores. Neste jogo é utilizada por exemplo, para verificar se o jogador tem energia suficiente para utilizar uma habilidade;
- Client – Função executada no cliente, destina-se a elementos que o cliente pode simular, não críticos, por exemplo, efeitos de partículas;
- NetMulticast – Função executada em todos os clientes conectados à sessão, tem muitos casos de uso. Alguns exemplos são: enviar uma mensagem a todos os jogadores, informar todos os jogadores que um jogador utilizou uma certa habilidade e muito mais.

Também é possível indicar se a função usará obrigatoriamente um canal fiável ou poderá utilizar canais não fiáveis se necessário, utilizando a tag adicional Reliable ou Unreliable.

6.2. Advanced Sessions Plugin

Após se ter decidido o que replicar, foi necessário arranjar maneira de criar sessões e juntar a sessões. Foi aí que o plugin Advanced Sessions entrou. O plugin suporta integração com a Steam - a plataforma de jogos de PC mais popular e utilizada de momento. É possível criar conta nesta plataforma gratuitamente, logo é o local ideal para testar o multijogador em rede.

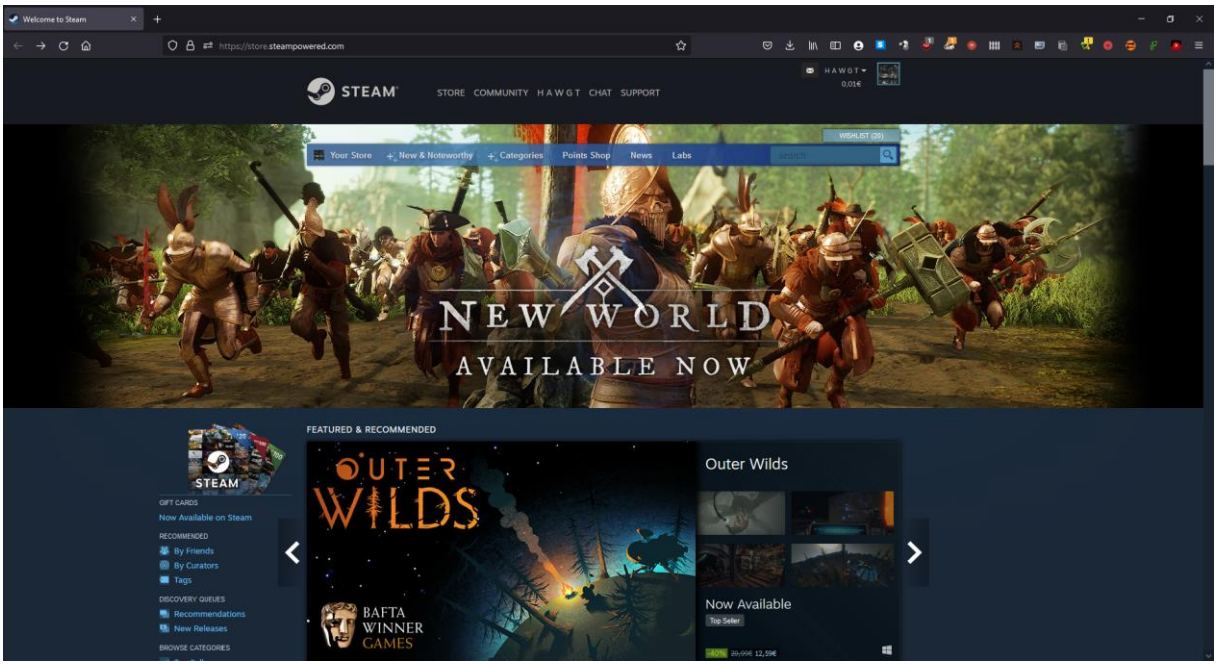

Figura 101 - Página da plataforma Steam

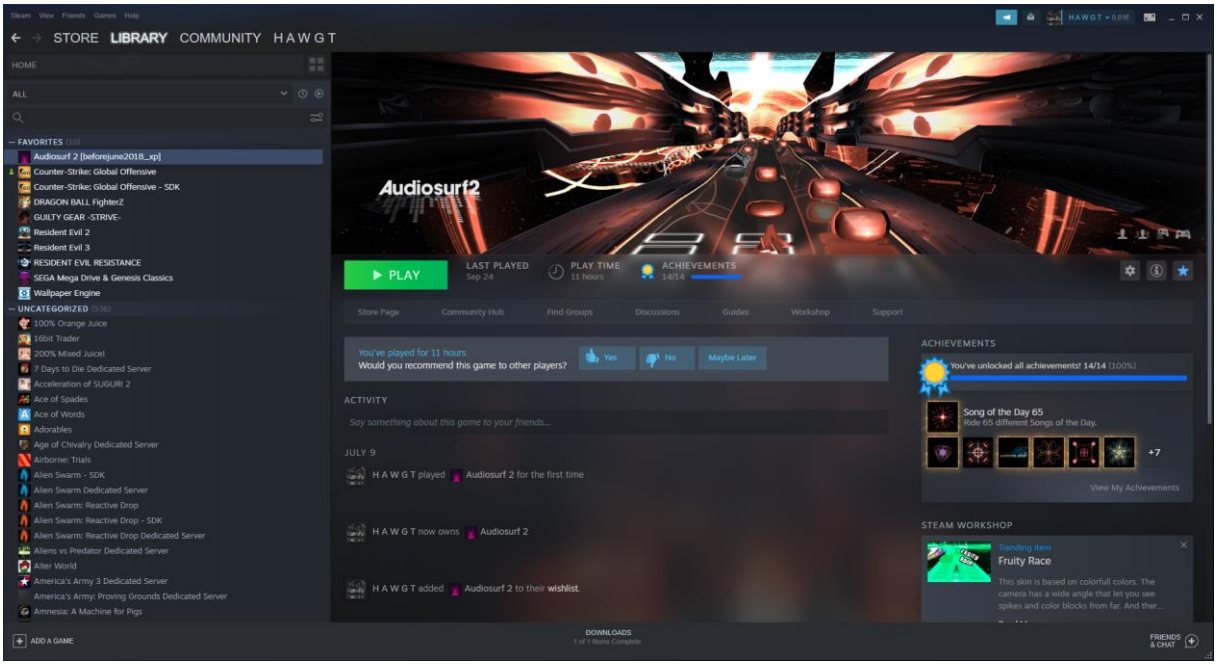

Figura 102 - Plataforma Steam

Após o download do plugin, colocam-se os conteúdos extraídos na pasta Plugins, que se encontra na pasta raiz do projeto. Prossegue-se à adição da configuração no ficheiro "DefaultEngine.ini", que está na pasta Config na pasta raiz do projeto. Prossegue-se agora à edição da Build para utilizar Networking e Steamworks.

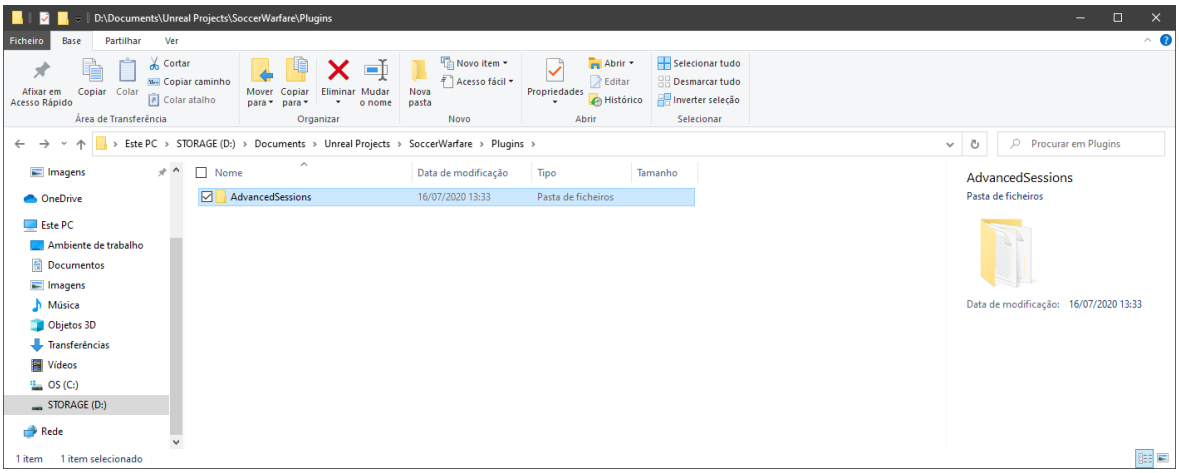

Figura 103 - Pasta do plugin

A opção que terá foco aqui será a "SteamDevAppId". Todos os jogos e aplicações na plataforma Steam contêm um id que a Valve (empresa que possuí a Steam) atribui. Esta permite que os utilizadores utilizem o id de teste 480. No entanto é possível utilizar qualquer outro id, desde que o utilizador possua a aplicação ou jogo com esse id na sua biblioteca Steam. Para a publicação do jogo na plataforma substitui-se o id por aquele que foi atribuído pela Valve.

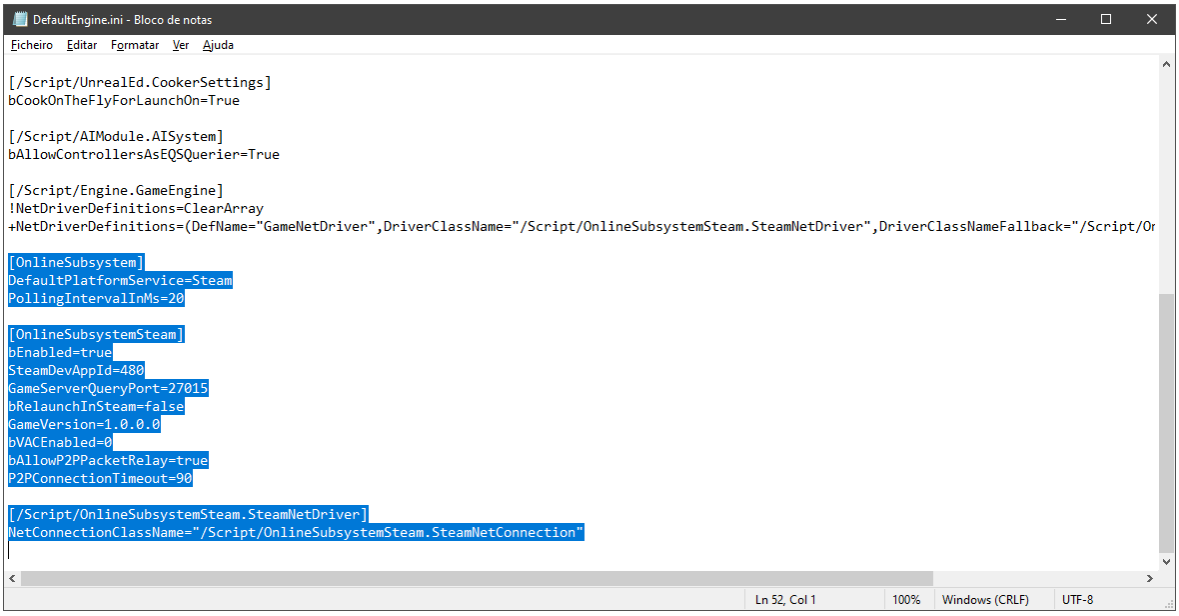

Figura 104 - Ficheiro de configuração

Editado o ficheiro de configuração, prossegue-se à declaração da sua utilização como dependência do projeto: processo semelhante para acrescentar a dependência da interface Gráfica.

SoccerWarfare.Target.cs | SoccerWarfareGameState.h | Ball.cpp | Ball.h | TeamPlayer.h

Miscellaneous Files | SoccerWarfareTarget

```
1   // Copyright 1998-2019 Epic Games, Inc. All Rights Reserved.
2
3   using UnrealBuildTool;
4   using System.Collections.Generic;
5
6   public class SoccerWarfareTarget : TargetRules
7   {
8       public SoccerWarfareTarget(TargetInfo Target) : base(Target)
9       {
10          bUsesSteam = true;
11          Type = TargetType.Game;
12          DefaultBuildSettings = BuildSettingsVersion.V2;
13          ExtraModuleNames.Add("SoccerWarfare");
14      }
15  }
16
```

Figura 105 - Ficheiro Target do projeto

SoccerWarfare.cpp | SoccerWarfare.h | SoccerWarfare.Target.cs | SoccerWarfareGa

SoccerWarfare | (Global Scope)

```
1   // Copyright 1998-2019 Epic Games, Inc. All Rights Reserved.
2
3   #pragma once
4
5   #include "CoreMinimal.h"
6   #include "Engine.h"
7   #include "Net/UnrealNetwork.h"
8   #include "Networking.h"
9   #include "Online.h"
```

Figura 106 - Includes necessários para utilização do netwroking em UE4

SoccerWarfare.Build.cs | SoccerWarfare.cpp | SoccerWarfare.h | SoccerWarfare.Target.cs | SoccerWarfareGameState.h | Ball.cpp | Ball.h | TeamPlayer.h | SoccerWarfareGameMode.h

Miscellaneous Files | SoccerWarfare | SoccerWarfare(ReadOnlyTargetRules Target)

```
1   // Copyright 1998-2019 Epic Games, Inc. All Rights Reserved.
2
3   using UnrealBuildTool;
4
5   using UnrealBuildTool;
6
7   public class SoccerWarfare : ModuleRules
8   {
9       public SoccerWarfare(ReadOnlyTargetRules Target) : base(Target)
10      {
11          PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;
12
13          PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore", "HeadMountedDisplay", "UMG", "OnlineSubsystem", "OnlineSubsystemUtils", "Steamworks", "Networking" });
14
15          DynamicallyLoadedModuleNames.Add("OnlineSubsystemSteam");
16
17          PrivateDependencyModuleNames.AddRange(new string[] { "Slate", "SlateCore" });
18
19          PrivateDependencyModuleNames.Add("OnlineSubsystem");
20      }
21  }
22
```

Figura 107 - Ficheiro Build com utilização dos módulos da Steam, Networking e o plugin

Compilado o projeto, agora pode-se finalmente dar início à implementação do Host & Join através do Steamworks, começando pelo mais básico.

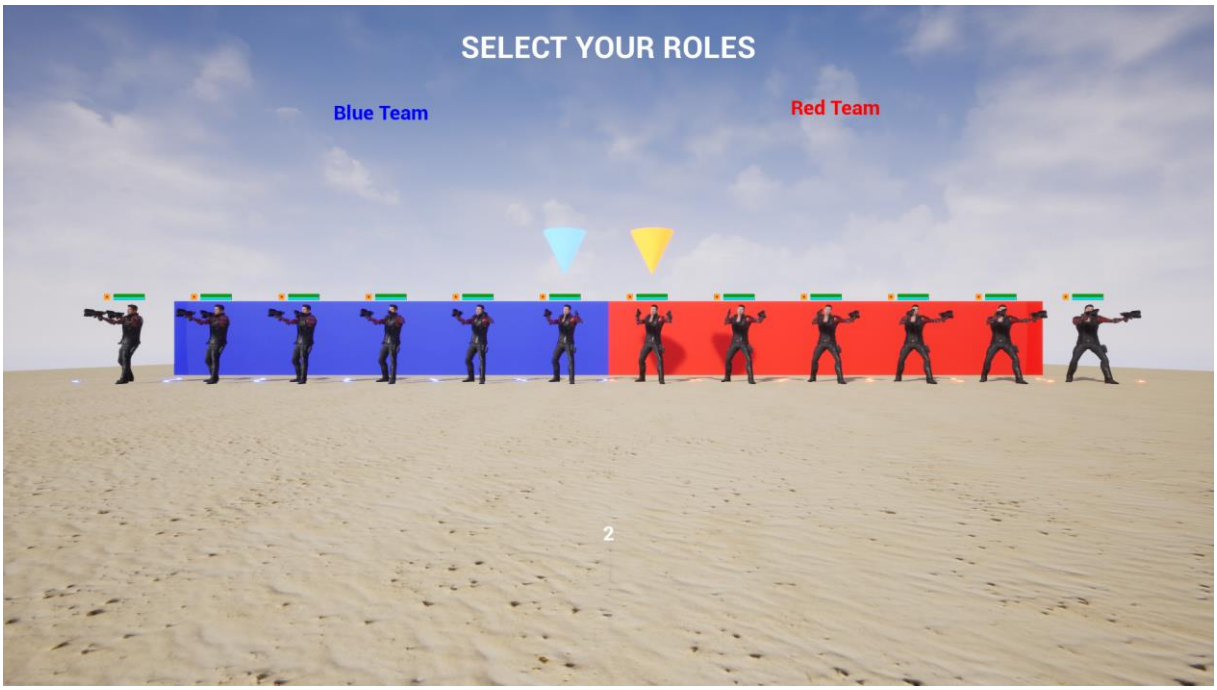

Figura 108 – Seleção de roles

A classe `SoccerWarfareGameInstance` servirá para juntar a sessões e criar sessões já que esta não é destruída na mudança de nível. Isto é importante já que o menu está presente num nível vazio, o qual é diferente do nível de jogo.

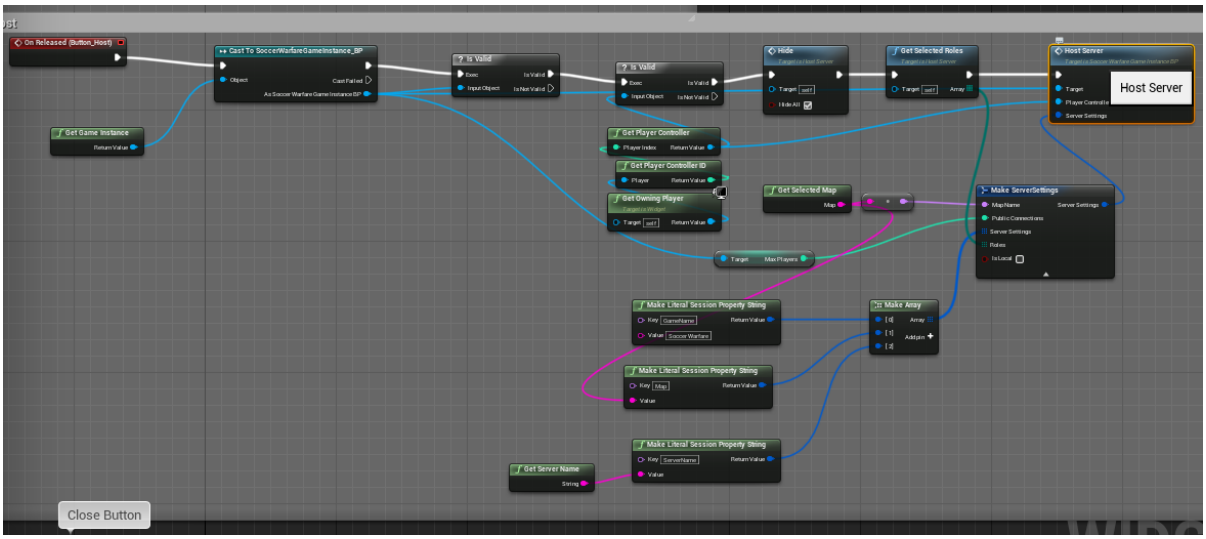

Figura 109 – Criação de sessão

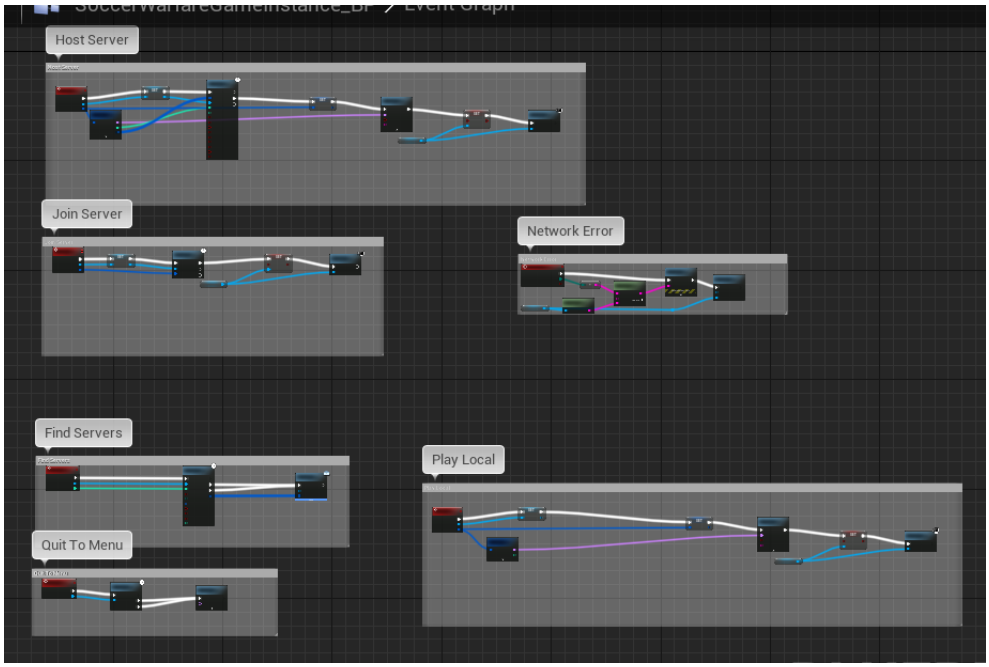

Figura 110 – Blueprint da SoccerWarfareGameInstance

Para encontrar sessões foi criado um *Server Browser*, este faz uma query à Steam de jogos com o AppId da aplicação igual e a chave “GameName” com o valor “Soccer Warfare”.

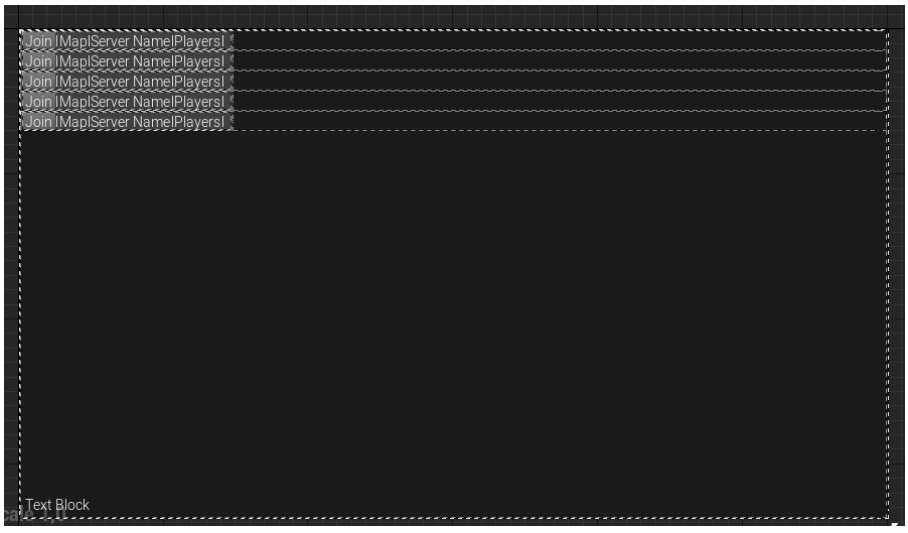

Figura 111 – Lista dinâmica de sessões

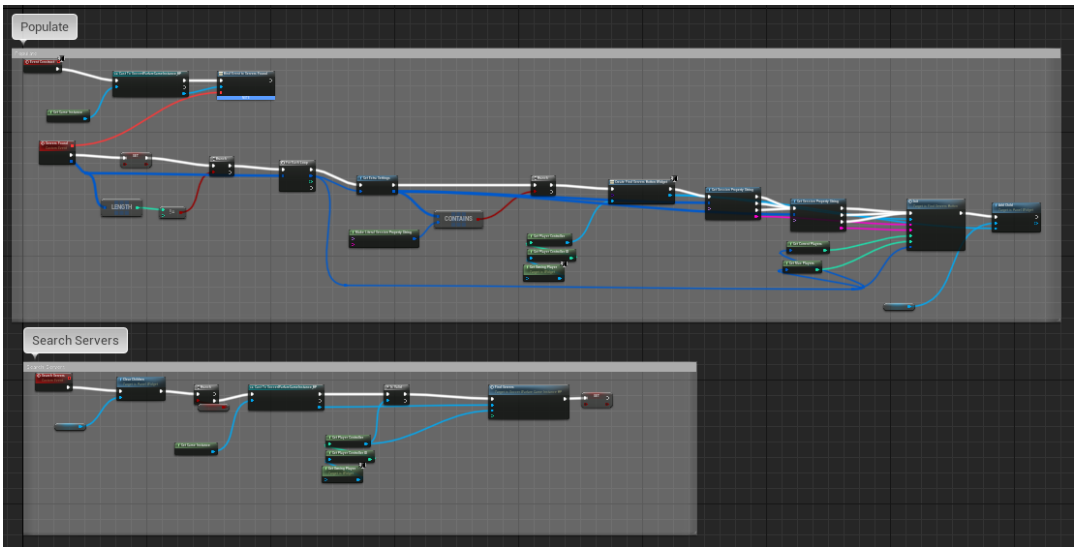

Figura 112 - Procura e preenchimento da lista de sessões

7. Adição do multijogador

Este capítulo vai explorar alguns dos passos tomados para a replicação do multijogador em rede e ajustes realizados para o multijogador local.

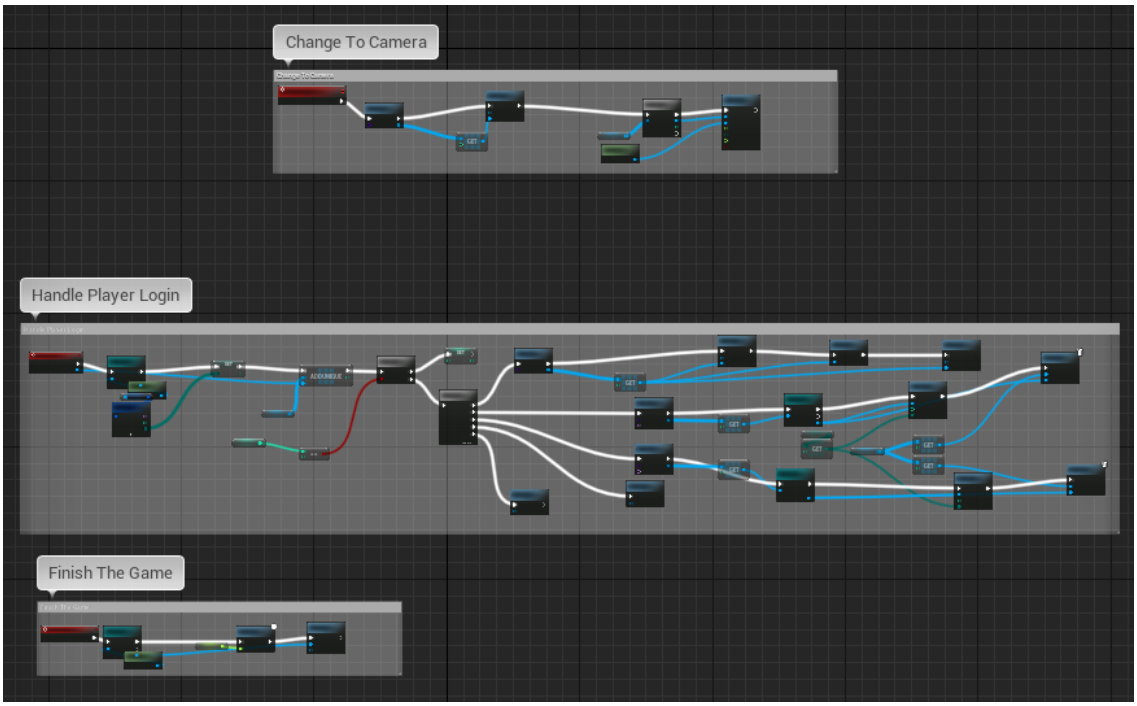

Figura 113 – Blueprint do SoccerWarfareGameMode

7.1. Multijogador em Rede

Após criar todos os menus necessários para se poder criar e juntar a servidores, falta decidir os elementos cruciais que necessitam de ser replicados constantemente pois o próprio nível já existe no jogador que se junta ao jogo.

Há coisas que são apenas serverside que nunca vão ser replicadas para o cliente, como as classes que herdam do GameMode, AIController. O único Controller replicado será o `PlayerController` do cliente.

Os elementos cruciais ao jogo, são:

- Propriedades do `GameState`
- Bola
- Jogadores
- Árbitro
- Jardineiro
- Tiros

- Caixa de balas
- Fire Ball
- Barrier
- Heal Bomb
- Beam

Para replicar blueprints ativa-se a opção *Replicates* na raiz da blueprint e também nos componentes que se deseja replicar. Em príncipio isto não requer mais ajustes. Como se deseja que as físicas sejam Server Authoritive, para atingir o máximo de fidelidade na jogabilidade, a opção “Replicate Physics to Autonomous Proxy” tem que ser ativada em todas as Meshs de blueprints replicadas neste jogo em específico, já que não há meshs que pertencem a blueprints replicadas que não é pretendida a sua replicação. Noutros jogos podem haver blueprints com meshs que a física é calculada apenas no client em elementos que não alterem a jogabilidade para otimização de rede.

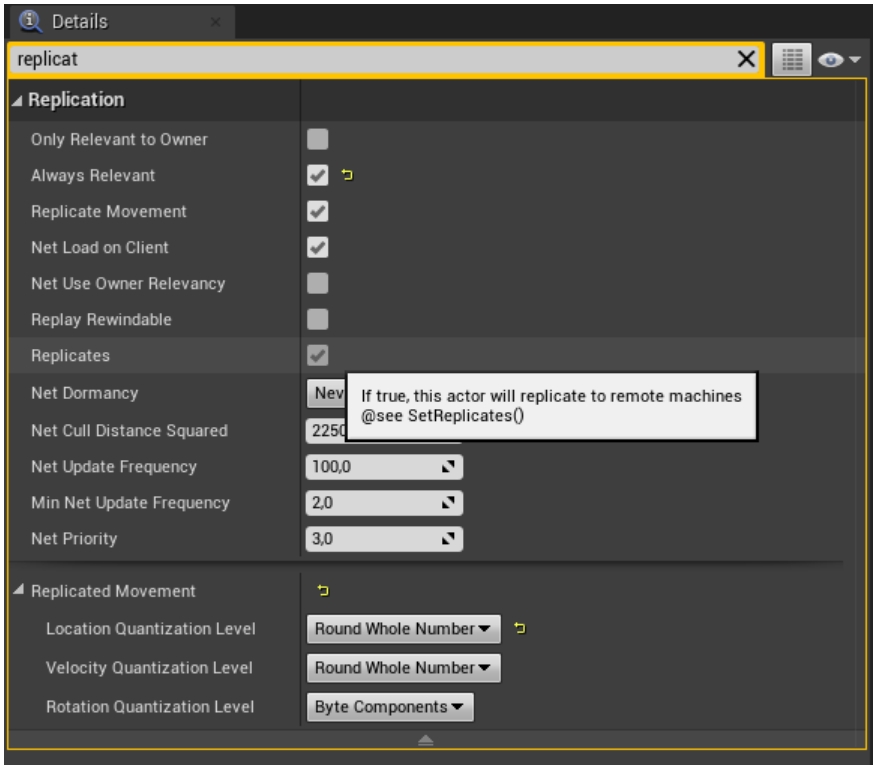

Figura 114 - Opções de replicação na blueprint

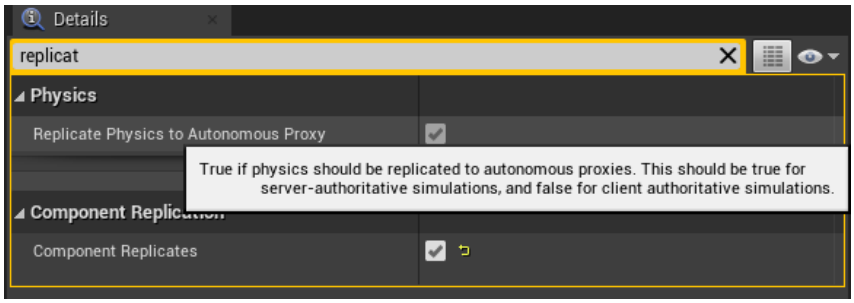

Figura 115 - Replicação de físicas

Na replicação de ações de jogadores, o utilizador quando tenta executar uma ação, informa o servidor. O servidor irá verificar o estado e se é possível executar a ação. Caso seja possível, irá enviar a todos os clientes a mensagem que esse utilizador executou a ação.

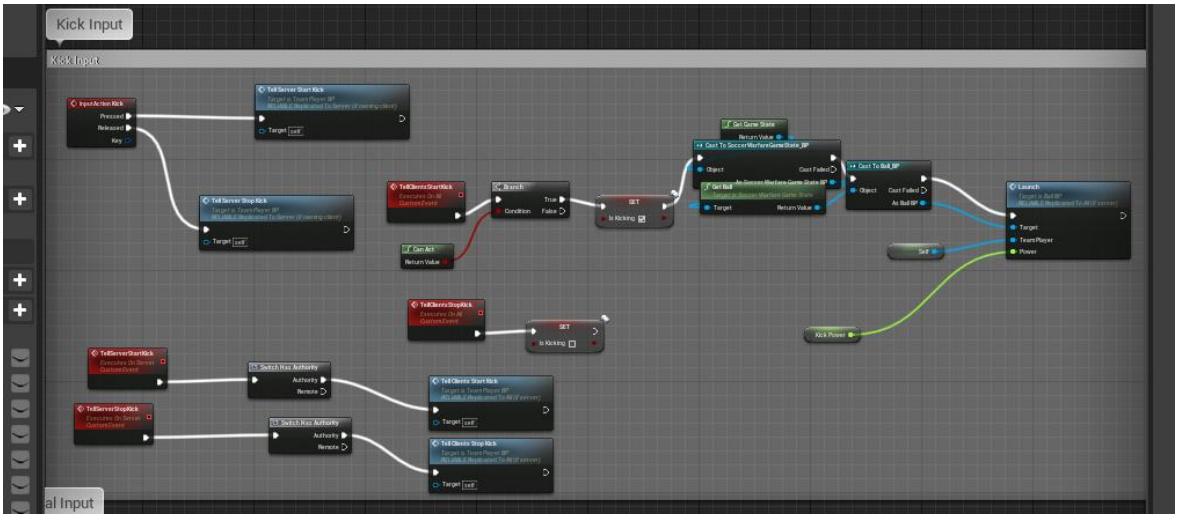

Figura 116 - Input do Chuto

Há métodos que necessitam ser replicados, nomeadamente os métodos que instanciam objetos no jogo, nomeadamente habilidades: troca de posse de jogador, mudança de cor da bola,

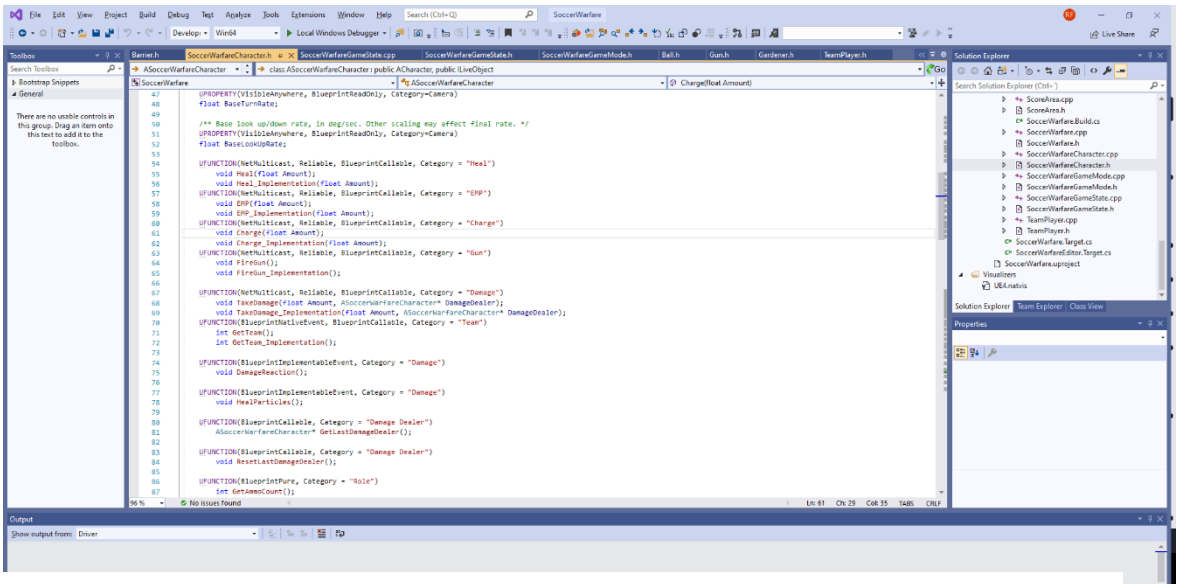

Figura 117 - Replicação de métodos da classe SoccerWarfareCharacter

dano, cura, etc. Para isto, acrescenta-se a tag dependo do tipo de replicação à macro dos métodos a serem replicados.

Para replicação de atributos o processo é semelhante: adiciona-se a tag replicated à macro da propriedade. Isto requer a declaração de uma função no ficheiro de código da classe e a chamada ao método SetReplicates com o único parâmetro com o valor verdadeiro.

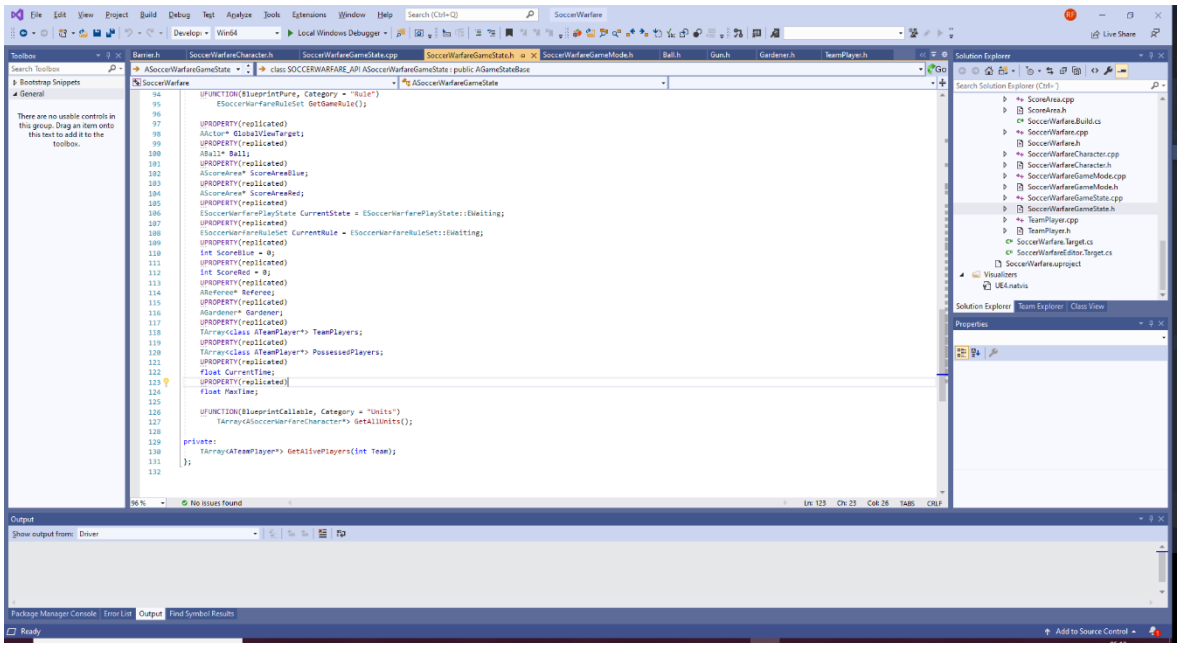

Figura 118 - Replicação de propriedades da classe SoccerWarfareGameState

```
ASoccerWarfareGameState::ASoccerWarfareGameState() {
    SetReplicates(true);
}
```

Figura 119 – SetReplicates no SoccerWarfareGameState

A função que tem de ser criada contém todas propriedades que foram declaradas como replicadas.

```
void ASoccerWarfareGameState::GetLifetimeReplicatedProps(TArray< FLifetimeProperty >& OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);
    DOREPLIFETIME(ASoccerWarfareGameState, GlobalViewTarget);
    DOREPLIFETIME(ASoccerWarfareGameState, Ball);
    DOREPLIFETIME(ASoccerWarfareGameState, ScoreAreaBlue);
    DOREPLIFETIME(ASoccerWarfareGameState, ScoreAreaRed);
    DOREPLIFETIME(ASoccerWarfareGameState, CurrentState);
    DOREPLIFETIME(ASoccerWarfareGameState, CurrentRule);
    DOREPLIFETIME(ASoccerWarfareGameState, ScoreBlue);
    DOREPLIFETIME(ASoccerWarfareGameState, ScoreRed);
    DOREPLIFETIME(ASoccerWarfareGameState, Referee);
    DOREPLIFETIME(ASoccerWarfareGameState, Gardener);
    DOREPLIFETIME(ASoccerWarfareGameState, TeamPlayers);
    DOREPLIFETIME(ASoccerWarfareGameState, PossessedPlayers);
    DOREPLIFETIME(ASoccerWarfareGameState, CurrentTime);
    DOREPLIFETIME(ASoccerWarfareGameState, MaxTime);
}
```

Figura 120 - Atributos a serem replicados

Após definir todos os métodos que necessitam ser replicados: RPCs, blueprints e componentes de blueprints, propriedades e atributos, procedeu-se ao playtesting. Inicialmente parecia estar tudo bem até algum jogador tocar na bola, aí começava a aparecer a mensagem na consola:

"LogNetPlayerMovement: Warning: CreateSavedMove: Hit Limit of 96 saved moves (timing out or very bad ping?)"

O que se estava a passar foi que as posições de todos os objetos já não eram replicadas, mas sim o cliente fazia a sua própria simulação das físicas ignorando a replicação do servidor criando dessincronização sendo objetos com posições diferentes no servidor e no cliente. Isto é um problema devido à natureza do motor de física não sendo determinístico (utiliza o PhysX tal como o Unity, para ser determinístico, a simulação teria que avançar apenas por intervalos de tempo fixos e todas as interações teriam que seguir sempre a mesma ordem), após a desativação das físicas no cliente, os objetos ficavam parados após o toque na bola.

Após alguma pesquisa na documentação Unreal Engine, foi encontrado o Network Profiler, uma ferramenta de gravação e visualização de tráfego de rede no jogo utilizada na consola do editor do UE4 enquanto o jogo está em execução para gravação de uma sessão.

Os comandos da consola para o Network Profiler são os seguintes:

- netprofile – alterna entre ativação e desativação
- netprofile enable – ativação
- netprofile disable – desativação

Após feita a gravação é possível encontrá-la na diretoria: "<Pasta do projeto Unreal Engine>\Saved\Profiling". Para visualizar a gravação basta abrir o NetworkProfiler.exe que está na diretoria: "<Pasta do UE>\Engine\Binaries\DotNET". Aberto o programa carrega-se uma gravação, aí foi possível descobrir a causa do problema.

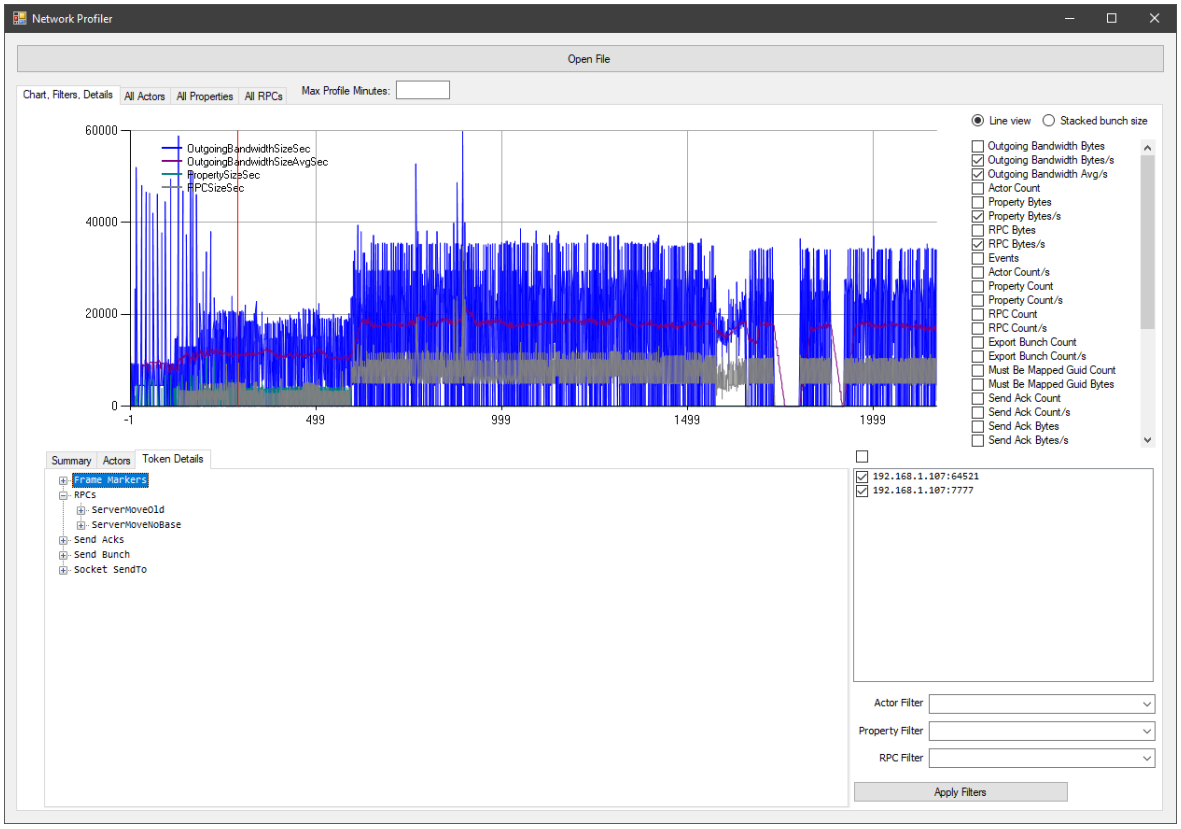

Figura 121 – Network Profiler – Gráfico I

Neste gráfico o eixo x representa o tempo, e o eixo y a quantidade, é possível clicar no gráfico a dado momento (fica uma linha vermelha a marcar o local) e ver as RPC (chamadas a funções replicadas) que foram feitas, ao andar um pouco na linha do tempo, verifica-se um aumento considerável no envio de dados, observando de mais perto, descobre-se que uma RPC estava a ser chamada várias vezes.

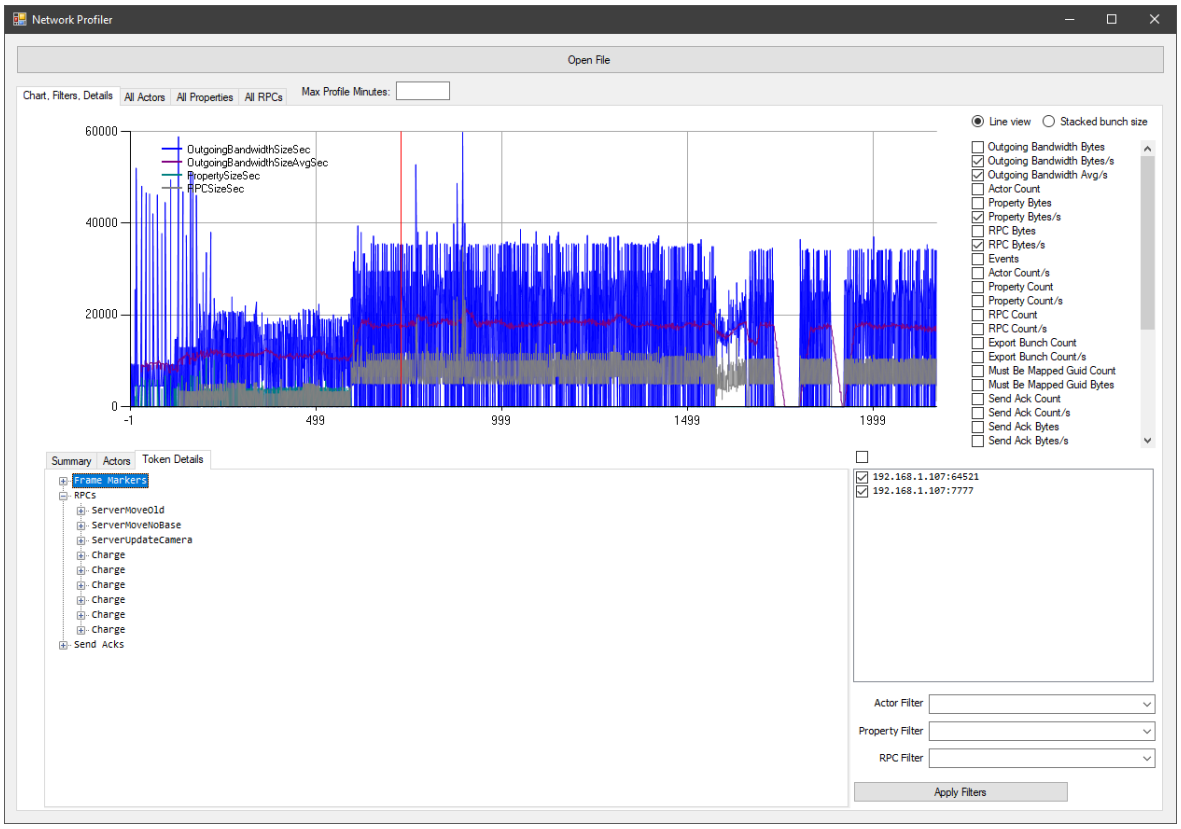

Figura 122 - Network Profiler - RPC a ser chamada várias vezes no mesmo pacote

Total Size (KBytes)	Count	Average Size (Bytes)	Average Size (Bits)	Time (ms)	Average Time (ms)	RPC
61.2	8802	7.1	57.0	0.00	0.0000	Charge
8.5	682	12.8	102.2	0.00	0.0000	ServerUpdateCamera
16.0	616	26.5	212.2	0.00	0.0000	ServerMoveNoBase
6.1	516	12.1	97.1	0.00	0.0000	ServerMoveOld
0.4	62	6.9	55.0	0.00	0.0000	SetTeam
1.3	62	20.9	167.0	0.00	0.0000	SetColor
0.3	62	4.9	38.8	0.00	0.0000	AttachToPlayer
1.9	46	42.9	343.3	0.00	0.0000	ServerMoveDualNoBase
0.1	13	7.1	57.0	0.00	0.0000	ClientAckGoodMove
0.1	7	12.8	102.0	0.00	0.0000	ClientSetViewTarget
0.1	4	36.0	288.0	0.00	0.0000	ClientAdjustPosition
0.0	2	4.1	33.0	0.00	0.0000	ClientRestart
0.0	2	4.3	34.0	0.00	0.0000	ServerAcknowledgePossession
0.0	1	4.1	33.0	0.00	0.0000	ClientSetHUD
0.0	1	3.0	24.0	0.00	0.0000	ClientVoiceHandshakeComplete
0.0	1	3.3	26.0	0.00	0.0000	ClientEnableNetworkVoice
0.0	1	7.3	58.0	0.00	0.0000	ClientCapBandwidth
0.0	1	5.8	46.0	0.00	0.0000	ClientSetRotation
0.0	1	3.1	25.0	0.00	0.0000	ServerShortTimeout

Figura 123 - Todas as RPCs replicadas presentes no gráfico I

Esta RPC está a ser executada na blueprint da Bola (Ball_BP), inspecionando a blueprint, observa-se que está chamada esta a ser feita por tick. O que está a causar o entupimento na ligação, foi logo executada a correção para apenas fazer esta chamada de segundo a segundo.

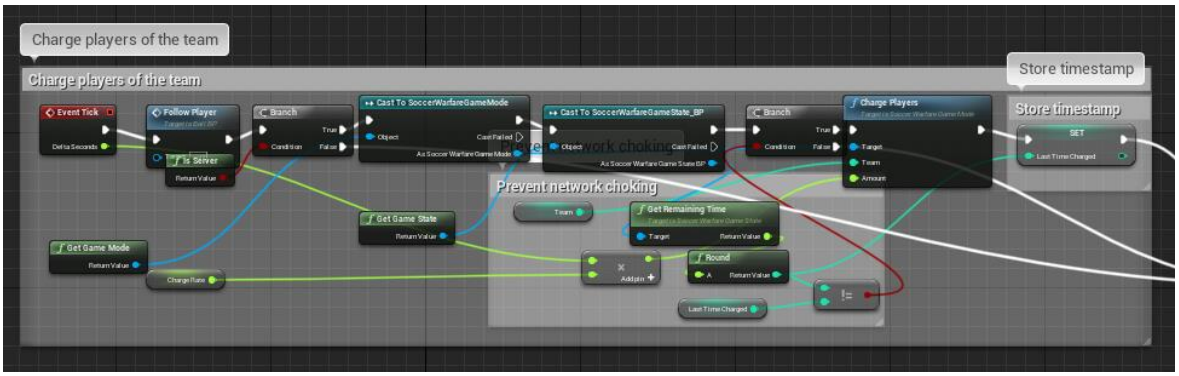

Figura 124 – Chamada à RPC ChargePlayers com verificação de execução

```
void ASoccerWarfareGameMode::ChargePlayers(int Team, float Amount)
{
    ASoccerWarfareGameState* SWGameState = Cast<ASoccerWarfareGameState>(GameState);
    if (!SWGameState) return;
    for (ATeamPlayer* TPlayer : SWGameState->TeamPlayers) {
        if (TPlayer->GetTeam() == Team && TPlayer->GetHP() > 0) TPlayer->Charge(Amount);
    }
}
```

Figura 125 - Método RPC ChargePlayers

```
    UFUNCTION(NetMulticast, Reliable, BlueprintCallable, Category = "Charge")
        void Charge(float Amount);
        void Charge_Implementation(float Amount);
```

Figura 126 - Declaração da RPC Charge

Sendo feita a correção, ao testar, observou-se que o problema tinha sido resolvido e foi feita uma nova gravação para comparação.

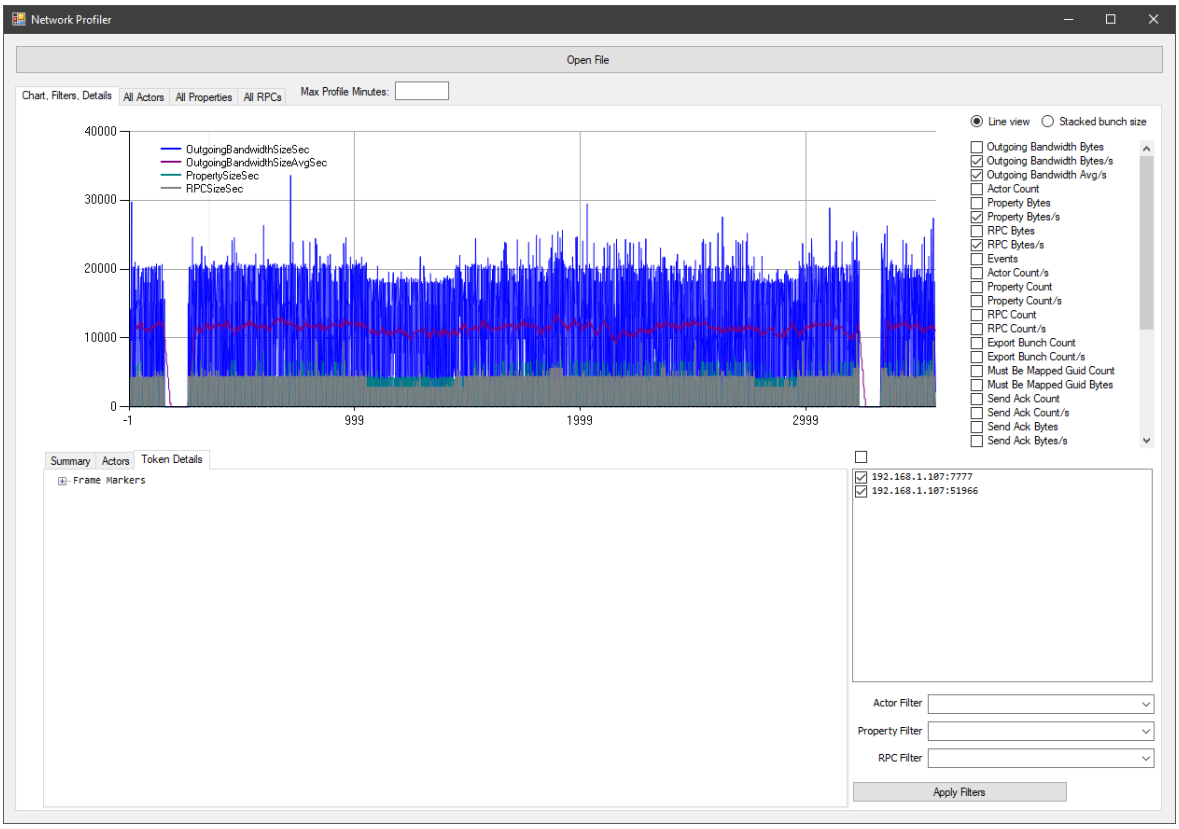

Figura 127 - Network Profiler - Gráfico II

Total Size (KBytes)	Count	Average Size (Bytes)	Average Size (Bits)	Time (ms)	Average Time (ms)	RPC
25.3	1079	24.0	192.0	0.00	0.0000	ServerMoveNoBase
12.2	942	13.2	105.8	0.00	0.0000	ServerUpdateCamera
1.0	144	7.1	57.0	0.00	0.0000	Charge
0.8	110	7.1	57.0	0.00	0.0000	ClientAckGoodMove
2.2	67	33.9	271.0	0.00	0.0000	ServerMoveDualNoBase
0.1	27	3.0	24.0	0.00	0.0000	Kick
0.1	10	6.1	48.6	0.00	0.0000	SetTeam
0.2	10	20.9	167.0	0.00	0.0000	SetColor
0.0	8	5.0	40.0	0.00	0.0000	AttachToPlayer
0.0	2	2.9	23.0	0.00	0.0000	UnAttachFromPlayer

Figura 128 - Todas as RPCs replicadas presentes no gráfico II

A RPC Charge já não está a ser executada com tanta frequência, nota-se que esta gravação tem uma duração maior, mas um número menor de chamadas RPC.

7.2.Multijogador Local

Apesar da adição de multijogador local ser mais fácil de implementar que o multijogador em rede, foi desenvolvido em 2º lugar, pois é possível correr multijogador local com código e lógica de rede, no entanto o contrário não é possível, logo o código foi desenvolvido primeiramente para multijogador em rede para depois apenas se ter de fazer ajustes pequenos para assegurar o funcionamento do multijogador local. O evento *OnPostLogin* funciona exatamente da mesma maneira localmente como em rede.

Para o multijogador funcionar localmente, acrescentou-se um atributo booleano na Struct *ServerSettings* que indicará se o jogo é local ou não. Isto vai determinar se o jogo tem que criar um `SoccerWarfarePlayerController` para o segundo jogador.

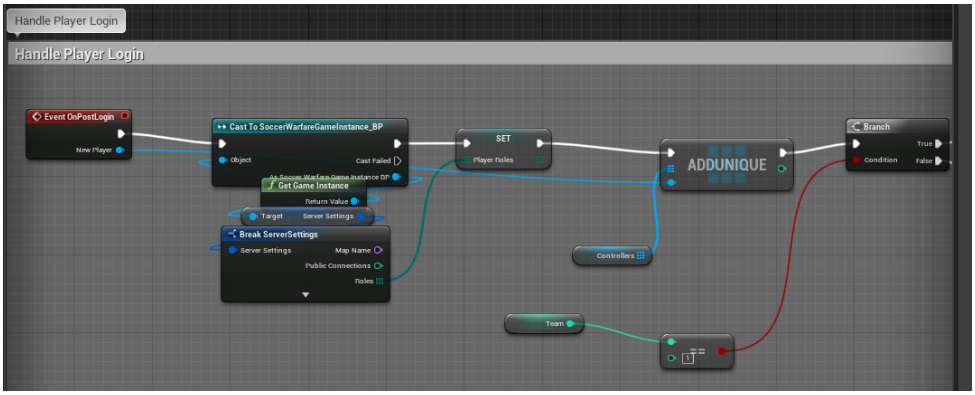

Figura 129 - Evento OnPostLogin

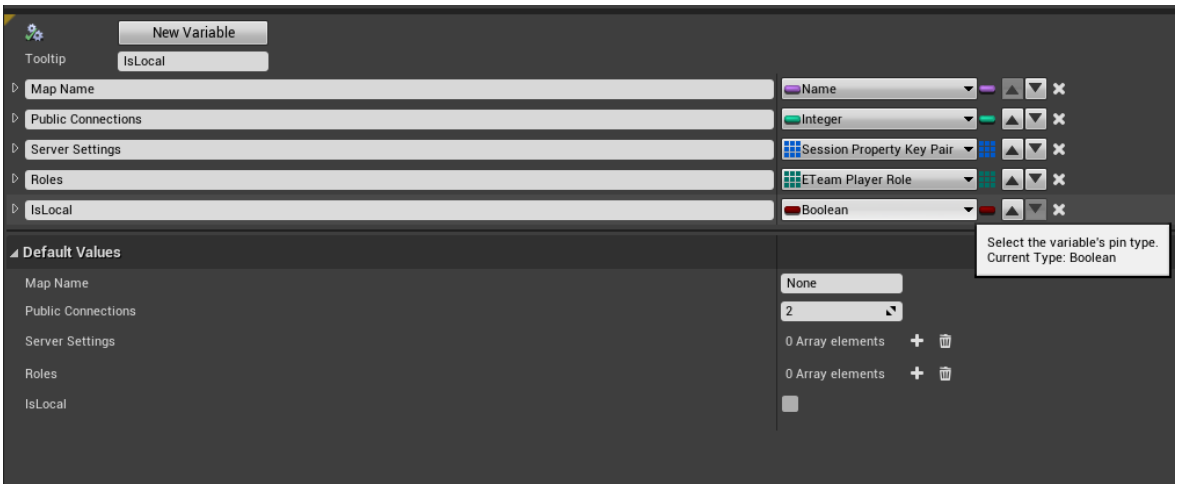

Figura 130 - ServerSettings Struct

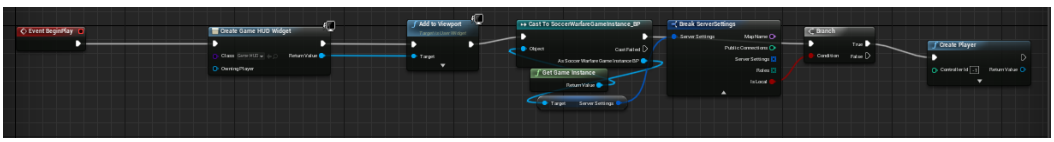

Figura 131 - Blueprint do nível principal com verificação se o jogo é local para a criação de um segundo Player

8. Niagara Particle System

O Unreal Engine de raiz contém 2 motores de sistemas de partículas, sendo estes o Cascade e o Niagara. O Niagara é o mais recente e o que terá mais utilização na próxima geração de jogos. Logo, foi decidida a sua utilização já que o conhecimento do seu funcionamento será muito útil.

O Niagara System permite a utilização de vários emitters dentro dele, o Niagara System também suporta funções e scripting para efeitos visuais mais complexos.

Para aprender o funcionamento do Niagara, procedeu-se a criação de um Niagara Emitter de partículas básico para representar fogo. Para isto criou-se uma imagem que foi convertida para Material que iria ser a textura base para cada partícula gerada no sistema de partículas.

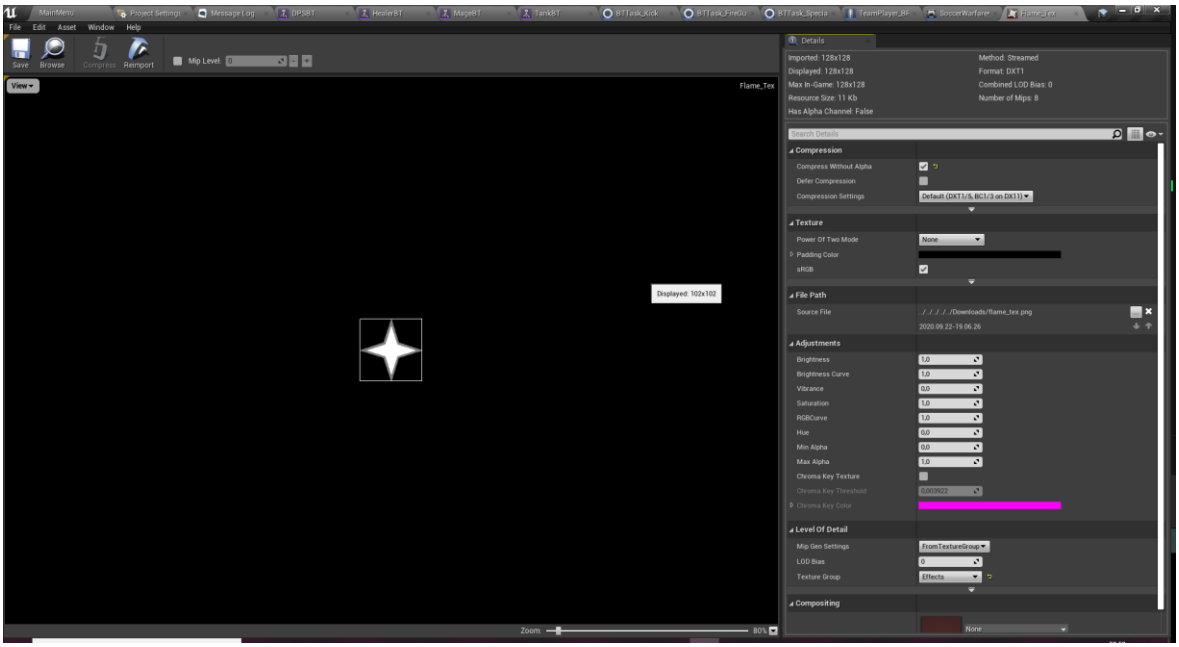

Figura 132 - Textura utilizada

No Emitter pode-se alterar as diferentes propriedades, nomeadamente coordenadas, velocidade, StaticMesh e Material. A razão de se ter utilizado uma textura monocromática foi porque se desejou que a origem tivesse uma cor diferente de quando estivesse para ser apagada, começando num branco, seguido de amarelo, depois vermelho e por fim preto. A escala das partículas também foi definida para aumentar ao longo do seu tempo de vida, esta curva foi definida nas definições do emitter.

Figura 133 - Definições do emitter I

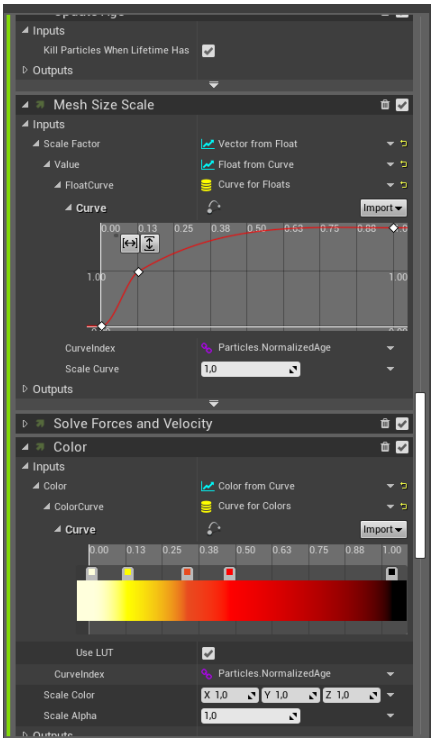

Figura 134 - Definições do emitter II

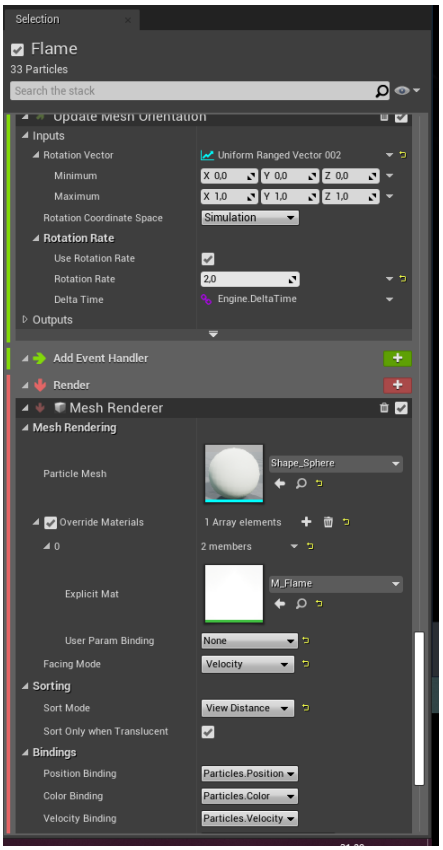

Figura 135 - Definições do Emitter III

Com o emitter acabado, foi criado um Niagara System que utilizará o emitter criado anteriormente, este Niagara System será depois instanciado em runtime sempre que desejado.

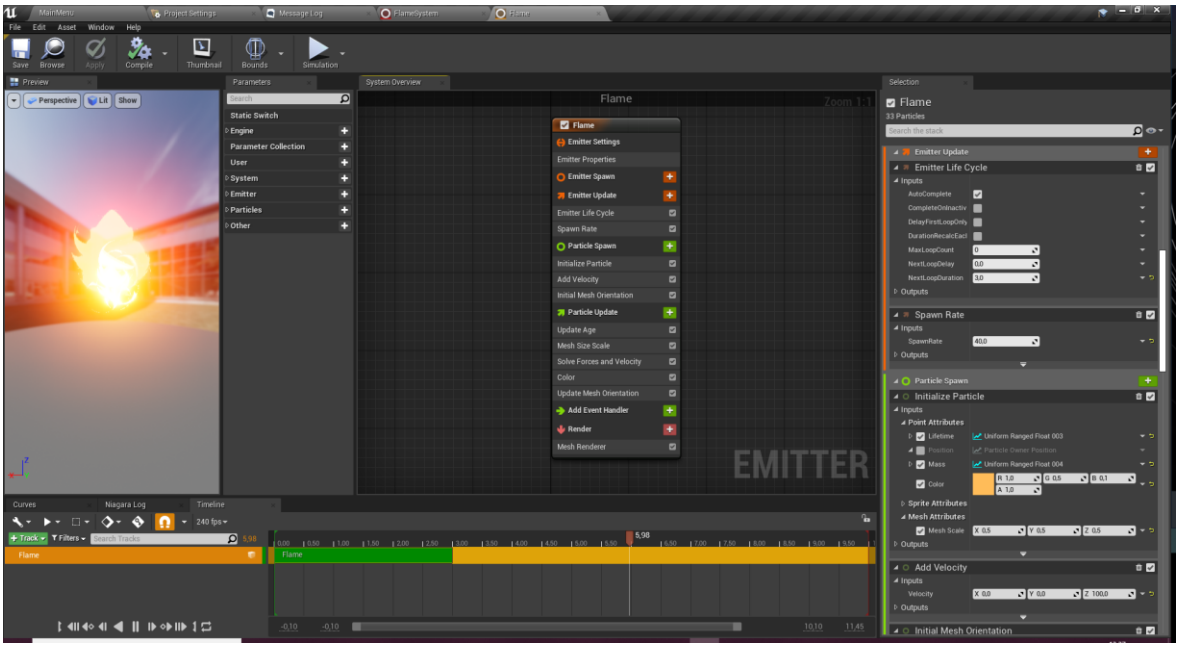

Figura 136 - Efeito de partículas de chama concluído

9. Publicação

Obviamente que o jogo não será lançado em formato de Editor, será uma release standalone. Por este motivo, o multijogador em rede não funcionará da mesma maneira pois nunca irá existir apenas uma sessão. É necessário um gestor de sessões, maneira de criar sessões, procurá-las e juntar-se a elas.

Para jogadores de jogos em computador, a grande maioria concorda que a Steam é a plataforma superior, ambos em termos de quantidade de jogadores e quantidade de jogos. Para programadores isto é ótimo pois é muito provável que haja muita documentação para integração de jogos com a plataforma.

A Steam fornece a Steamworks que é um kit de ferramentas e serviços que ajudam os desenvolvedores de jogos e publicadores a construir os seus jogos e fazer uso da plataforma. Uma das funcionalidades é o matchmaking e lobbies. O matchmaking serve para encontrar sessões automaticamente a partir de determinados parâmetros. Para este jogo inicialmente vai ser utilizado o sistema de lobbies pois é muito mais prático o seu teste durante o desenvolvimento.

Após alguma pesquisa, descobriu-se que existe um plugin chamado de Advanced Sessions que tem a funcionalidade de integração com a Steam. Instalou-se o plugin e alterou-se o ficheiro de configuração para permitir a sua utilização com a Steam. Depois disso criou-se interface gráfica para se poder criar sessões, procurá-las e juntar-se a elas.

Tendo o multijogador em rede a funcionar, fez-se umas alterações mínimas para o multijogador local funcionar.

10. Conclusão

Este trabalho destinou-se a desenvolver um projeto no âmbito da disciplina de projeto do Mestrado de Computação Móvel em Engenharia Informática, com objetivo de desenvolver competências na área de desenvolvimento de videojogos, na área de inteligência artificial, o Unreal Engine 4 e de aprofundar conhecimentos em lógica de rede.

Nos capítulos anteriores abordaram-se as partes consideradas relevantes no desenvolvimento deste projeto, desde o GDD em si, a arquitetura do UE4, criação do nível, implementação das mecânicas, inteligência artificial, o multijogador, até ao aspeto visual (Niagara Particle System).

Os aspetos que mais marcaram este projeto foram: a implementação de mecânicas de jogo, a câmara, inteligência artificial e multijogador em rede. Durante o seu desenvolvimento surgiram alguns problemas, tais como: dessincronização no multijogador em rede, comportamento não desejado pelo *PlayerController*, animações que não eram reproduzidas pela IA. Todos estes problemas foram solucionados, e a sua resolução foi descrita neste documento.

Em suma, todos os objetivos foram concluídos, desde a aprendizagem do UE4, o desenvolvimento de um jogo, a partir de um GDD em UE4, com o multijogador em rede.

Para trabalho futuro, sugere-se melhorias na interface gráfica, inclusão de novas animações, novos modelos para as unidades, mais mapas e algoritmos para suavizar a sincronização em situações de rede menos favoráveis, melhorias na inteligência artificial (tilização de serviços nas behavior trees para alterações de dados na blackboard), matchmaking automático, mais jogadores por jogo, substituição de materiais dinâmicos por instâncias de materiais (materiais iguais, mas com parâmetros diferentes), inclusão de músicas, mais sons, efeitos especiais e AI Auto Balancer (atualização de equílibrio e ajuste de valores por inteligência artificial, tais como o dano de habilidades).

Webgrafia e Bibliografia

Epic Games – Getting Started – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/GettingStarted/index.html

Epic Games – Third Person Template – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Resources/Templates/ThirdPerson/index.html

Epic Games – Blueprints – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html

Epic Games – Programming – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Programming/index.html

Epic Games – Gameplay Framework Quick Reference – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Gameplay/Framework/QuickReference/index.html

Epic Games – Character Movement – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Gameplay/HowTo/CharacterMovement/Blueprints/index.html

Epic Games – Game Mode – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Gameplay/Framework/GameMode/index.html

Epic Games – Controller – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Gameplay/Framework/Controller/index.html

Epic Games – AI Controller – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Gameplay/Framework/Controller/AIController/index.html

Epic Games – User Construction Script – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Engine/Blueprints/UserGuide/UserConstructionScript/index.html

Epic Games – Player Controller – Consultado em: 30/09/2021 – Disponível em:

https://docs.unrealengine.com/en-US/Gameplay/Framework/Controller/PlayerController/index.html

Epic Games – Niagara – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Engine/Niagara/Overview/index.html

Epic Games – Blackboard Documentation – Consultado em: 30/09/2021 – Disponível em: https://forums.unrealengine.com/development-discussion/blueprint-visual-scripting/2138-blackboard-documentation

Epic Games – Nav Mesh – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Resources/ContentExamples/NavMesh/index.html

Epic Games – Behavior Trees – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/BehaviorTrees/index.html

DevSquad – Project Setup - #1 Unreal Engine 4 Steam Multiplayer Essentials – Consultado em: 30/09/2021 – Disponível em: https://www.youtube.com/watch?v=d4gQSXALyHU

Split Hare Games – How to Make a Stylized Fire Effect in UE4 - Niagara Tutorial – Consultado em: 30/09/2021 – Disponível em: https://www.youtube.com/watch?v=mQen2jrxrgg

Wikipedia – Blackboard System – Consultado em: 30/09/2021 – Disponível em: https://en.wikipedia.org/wiki/Blackboard_system

Cobb Dev – Retargeting Mixamo Animations | #04 Let's Create Breath of the Wild in UE4 – Consultado em: 30/09/2021 – Disponível em: https://www.youtube.com/watch?v=SbC9rIRgu-A

Valve – Steamworks – Consultado em: 30/09/2021 – Disponível em: https://partner.steamgames.com/doc/home

Wikipedia – Game Design Document – Consultado em: 30/09/2021 – Disponível em: https://en.wikipedia.org/wiki/Game_design_document

Wikibooks – Blender 3D: Noob to Pro/What is a Mesh? – Consultado em: 30/09/2021 – Disponível em: https://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro/What_is_a_Mesh%3F

Epic Games – RPCs – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Gameplay/Networking/Actors/RPCs/index.html

Epic Games – EQS – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/EQS/index.html

Epic Games – AI Perception – Consultado em: 30/09/2021 – Disponível em: https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/AIPerception/index.html

Martin T. Hagan – Neural Network Design

Pixel Sapiens – Functions vs Events in Unreal Engine 4 – Consultado em: 30/09/2021 – Disponível em: https://pixelsapiens.com/functions-vs-events-in-unreal-engine-4/

Steve Streeting – Using Bullets for physics in UE4 – Consultado em: 30/09/2021 – Disponível em: https://www.stevestreeting.com/2020/07/26/using-bullet-for-physics-in-ue4/

Glossário

Blueprint – Sistema de scripting visual do Unreal Engine 4.

Mesh (modelação) – Coleção de vértices, arestas e faces que descrevem a forma de um objeto 3D.

Material - Controla o aspeto visual onde este for aplicado, este calcula como a luz interage com ele.

RPC – Funções chamadas localmente, mas executadas remotamente noutra máquina.

Tick – Iteração.

Transform – Coordenadas, rotação e escala de um objeto.

Anexos

Anexo A: GDD – Soccer Warfare