ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

Д И П Л О М Н А Р А Б О Т А

Тема: Екшън игра от първо лице във Виртуална Реалност

Дипломант:
Даниел Янев

Научен ръководител:
Виктор Кетипов

С О Ф И Я

2 0 1 9

Мнение на Дипломен Ръководител

Дипломантът Даниел Янев се е справил отлично с разработката на дипломната си работа. Обратната връзка, която му бе предоставена е взета предвид и в резултат на това както архитектурата, така и за използваемостта на продукта са на високо ниво.

Дипломен Ръководител: Виктор Кетипов

.....

Увод

Продуктът представлява игра от първо лице в Виртуална Реалност. Целевата аудитория на продукта е геймъри, които вече притежават комплект за Виртуална Реалност със система за контрол на движение. Такива система в момента на пазара са:

- Oculus Rift (Избран за разработка)
- HTC Vive

Целта на играта е защита на обект (в случая бариера) от врагове. Играта включва 5 вълни от врагове, 5 различни врагове, 3 различни оръжия, система за купуване на оръжия/ъпгрейди. Играчът може да държи едновременно максимум 2 оръжия (по едно във всяка ръка).Играта приключва или когато паднат и двете прегради от едната страна или когато играчът успешно победи всички вълни от врагове.

Основното движение се установява чрез VR комфортен режим (VR Comfort Mode). VR комфортният режим заменя популярното плавно движение на камерата със завъртане на точно определен интервал от градуси.

Движение на базата на ориентация погледа също е популярен избор сред разработчиците, които включват VR комфортен режим в игрите от първо лице, но е по-малко естествено от завъртания на точни интервали.

ПЪРВА ГЛАВА

Технологии за реализация на дипломния проект

1. Основни принципи, технологии и развойни среди, използвани при реализацията.
   1.1. Технологии
      1.1.1. Unreal Engine (Фиг 1.1)

Фиг 1.1 Unreal Engine лого

Unreal Engine е един от най-разпространените публични гейм енджини („game engines“) на пазара. Поддръжката на всички масови платформи заедно с безкомпромисното качество и вече 21 години разработка показват защо това е един от най-популярните и често използвани гейм енджини. Unreal Engine е изцяло написан на C++, като целия код е публично достъпен в Github.

Unreal Engine се грижи „почти“ изцяло за процеса на рендериране, менажирането на памет, създавайки абстракция за програмиста, и още много.

Няма преки пътища към създаването на потапящи се преживявания, които са правдоподобни за човешкия ум. VR изисква сложни сцени, представени при много високи стойности на кадрите. Тъй като Unreal Engine е предназначен за взискателни приложения като AAA игри, създаване на филми и фотореалистични сцени, той отговаря на тези изисквания и осигурява солидна основа за изграждане на продукти за всички VR платформи - от компютри през конзоли до мобилни.

-Wikipedia

Главните предимства при използване на гейм енджин са, че те се грижат за:

- Рендериране
- Управление на памет
- Засичане на колизии и др.

Като бонус Unreal Engine предоставя и невероятно лесно интегриране на Виртуална Реалност.

1.1.2. C++ (Фиг 1.2)

Фиг 1.2 C++ лого

„C++ или CPP („Си Плюс Плюс“) е език за програмиране с общо предназначение. Той има императивни, обектно-ориентирани и общи програмируеми функции, като същевременно осигурява възможности за манипулиране на паметта на ниско ниво.“ – Wikipedia

Unreal Engine имплементира собствен „Garbage Collector“ и ” Unreal Property System (Reflection)“ система базирана изцяло на макрота. Две функции, нужни на един модерен гейм енджин, които C++ не предлага.

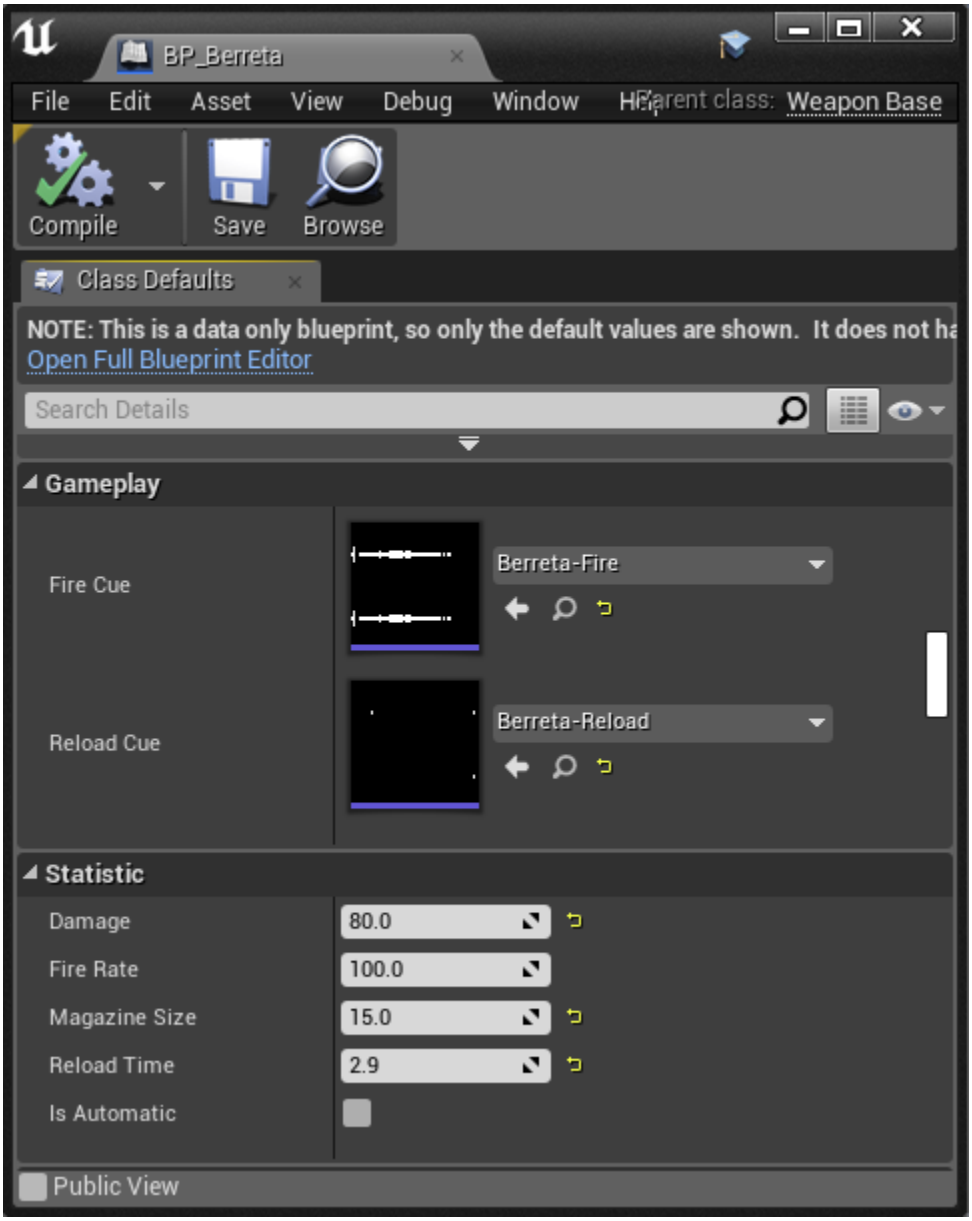
Системата „Garbage Collector” изцяло се грижи за управлението на памет като следи кои обекти се използват и кои вече не се използват. “Modern C++” имплементира няколко „умни поинтери“ („Smart pointers”), но Unreal Engine скрива и това от програмистите като менажира цялата памет.

“Reflection” системата дава способността на една програма да се изследва по време на изпълнение. Това е изключително полезно и е основополагаща технология на Unreal Engine, много системи като детайлни панели в редактора, сериализация, „Garbage Collection”, репликация на мрежата и комуникация „Blueprint“ / C ++, разчитат на тази система. C ++ обаче не поддържа под никаква форма „Reflection” система, така че Unreal има своя собствена система за събиране, обработване и манипулиране на информация за C ++ класове, структури, функции, член променливи и енумерации.

„Reflection” системата не е автоматична. За да бъде „видим“ за нея тип или свойство на един клас трябва да се анотира (Фиг 1.3 и Фиг 1.4). „Unreal Header Tool“ ще събере тази информация, когато компилирате проекта.

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "InteractableComponent.h"
#include "WeaponBase.generated.h"


//Amount of times reload widget is updated
#define UPDATE_TICKS 50

UCLASS(abstract)
class ZOMBIESURVIVALFPS_API AWeaponBase : public AActor
{
	GENERATED_BODY()

public:
	AWeaponBase();

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = Statistic)
	float Damage = 0;

	/** Firerate in rounds per minute */
	UPROPERTY(EditAnywhere, Category = Statistic)
	float FireRate = 100;

	UPROPERTY(EditAnywhere, Category = Statistic)
	float MagazineSize = 0;

	/** In seconds */
	UPROPERTY(EditAnywhere, Category = Statistic)
	float ReloadTime = 0;

	UPROPERTY(EditAnywhere, Category = Statistic)
	bool bIsAutomatic = false;

	/** Sound to play each time gun is fired */
	UPROPERTY(EditAnywhere, NoClear, BlueprintReadWrite, Category = Gameplay)
	class USoundBase* FireCue;

	/** Sound to play each time gun is reloaded */
	UPROPERTY(EditAnywhere, NoClear, BlueprintReadWrite, Category = Gameplay)
	class USoundBase* ReloadCue;

	…

}
```

Фиг 1.3 Демонстрация на „Unreal Property System (Reflection)“ в код



Фиг 1.4 Демонстрация на „Unreal Property System (Reflection)“ в редактора

1.1.2. Oculus API

Всички версии на Unreal 4.10 и по-късно включват вградена поддръжка за Oculus Rift и Samsung Gear VR, включително автоматично стереоскопично визуализиране и проследяване. Поддръжката на Oculus Go е включена в Unreal Engine 4.19 и по-късно. Unreal се грижи за цялата комуникация между софтуера и “Oculus API”, което управлява, както очилата за виртуална реалност, така и контролерите за движение.

1.2.Хардуер

1.2.1. Oculus Rift (Фиг 1.5)

Фиг 1.5 Oculus Rift + Touch bungle

Oculus Rift е платформа за виртуална реалност, разработен от купената от Facebook компания Oculus. Rift има два Pentile OLED дисплея, 1080 × 1200 резолюция всеки, 90 Hz честота на опресняване и 110 ° зрително поле. Очилата за Виртуална Реалност (HMD – Head Mounted Display) притежава и ротационно и позиционно проследяване, както и вградени слушалки, които осигуряват 3D аудио ефект.

1.2.2. Touch bundle (Фиг 1.6)

Фиг 1.6 Oculus Touch bungle

Системата за контрол на движението на Oculus Rift е известна като Oculus Touch. Състои се от двойка преносими устройства, по една за всяка ръка, всяка от които съдържа аналогова стик, три бутона и два тригера (един често използван за хващане, а другия за стрелба). Контролерите са напълно проследени в 3D пространството от системата “Constellation”, така че те могат да бъдат представени във виртуалната среда[i] и всеки контролер разполага със система за откриване на жестове с пръсти, които потребителят може да прави, докато ги държи. Всеки контролер използва една AA батерия.

Контролерите представляват единственият (като изключим движението на HMD) начин на потребителя да взаимодейства със софтуера.

ВТОРА ГЛАВА

Проектиране на структурата на игрови приложения за Виртуална Реалност

1. Функционални изисквания към приложението

1.1. Изисквания към изграждането на логиката

Трябва да бъде изградена основна структура на приложението. Да се направят основни класове, чрез които добавянето на нови функционалности да става бързо и лесно. Да се изгради лесна и логически издържана комуникация между различните части на приложението по „Good practice" конвенцията на Unreal Engine.

1.2. Потребителски интерфейс и итерация с потребителя

Потребителският интерфейс на приложението трябва да бъде максимално опростен и разбираем. Цялата навигация ще трябва да се осъществява само чрез реални 3D обекти в играта, затова всички менюта и бутони трябва да са с достатъчно голям размер, за да не бъде изпитвано затруднение при използването и разчитането им.

1.3. Функционалност

Приложението трябва да има минимум 5 различни вълни от врагове, 5 различни врагове и система за купуване на оръжия.

1.4. Производителност

Приложението не трябва да забива или замръзва без причина и е желателно да поддържа средно 90 кадъра в секунда като минимума не пада под 60. Трябва да се използва метод за движение, който не причинява гадене и

замайване при продължително използване на Виртуалната Реалност. Моменти на насичане, прекъсване или други дефекти не трябва да бъдат забелязвани.

2. Съображения за избор на програмни средства и развойната среда

2.1. Гейм енджин Unreal Engine

За реализацията на приложението беше взет избора да се използва гейм енджин поради многото си предимства при разработка, като например:

- Спестява много време и усилия, като се грижи за паметта, рендериране и дава много абстракции, и вече готови и имплементирани класове, както и инструменти за работа с 3D модели, анимация и т.н.

- Повечето игрални двигатели са междуплатформени, както и предоставят интегрирана поддръжка за Виртуална Реалност.

Игровият двигател, който беше избран е Unreal. Причината е, че е “Source Available” и инструментите му не са непознати. Това многократно ускорява процеса на работа.

2.2. Oculus Rift платформа

Продуктът е насочен към компютърни ентусиасти, по простата причина, че една игра във Виртуална Реалност е много по-изискваща от страна на ресурс. На пазара има много мобилни платформи за Виртуална Реалност, но нито една не успява да предостави достатъчно ресурси, за да е приятно и максимално близко до реалността изживяването.

2.3. GitHub (Фиг 2.1) система за контрол и управление на версиите

Фиг 2.1 GitHub лого

GitHub Inc. е уеб-базирана хостинг услуга за контрол на версиите чрез Git. Git система за контрол е избрана заради вече съществуващия опит с нея.

2.4. Visual Studio 2017 текстов редактор (Фиг 2.2)

Фиг 2.2 Visual Studio лого

Unreal Engine има вградена интеграция с Visual Studio, която улеснява компилиране, менажиране на проекта и търсене в кода на игровия двигател.

2.5. Blender графичен редактор (Фиг 2.3)

Фиг 2.3 Blender лого

Blender е безплатен „Open Source" 3D софтуер, който се използва за създаване на анимационни филми, визуални ефекти, изкуство, 3D печатни модели, интерактивни 3D приложения и видео игри. - Wikipedia

ТРЕТА ГЛАВА
Програмна реализация на проекта

1. Архитектура на приложението
   1.1. Основни класове за всеки Unreal Проект

Архитектурата на Unreal Engine разчита на няколко основни класа за да контролира играта.

1.1.1. Game Mode („Игрови Режим“)

В архитектурата на Unreal игровият режим се грижи за следните неща:

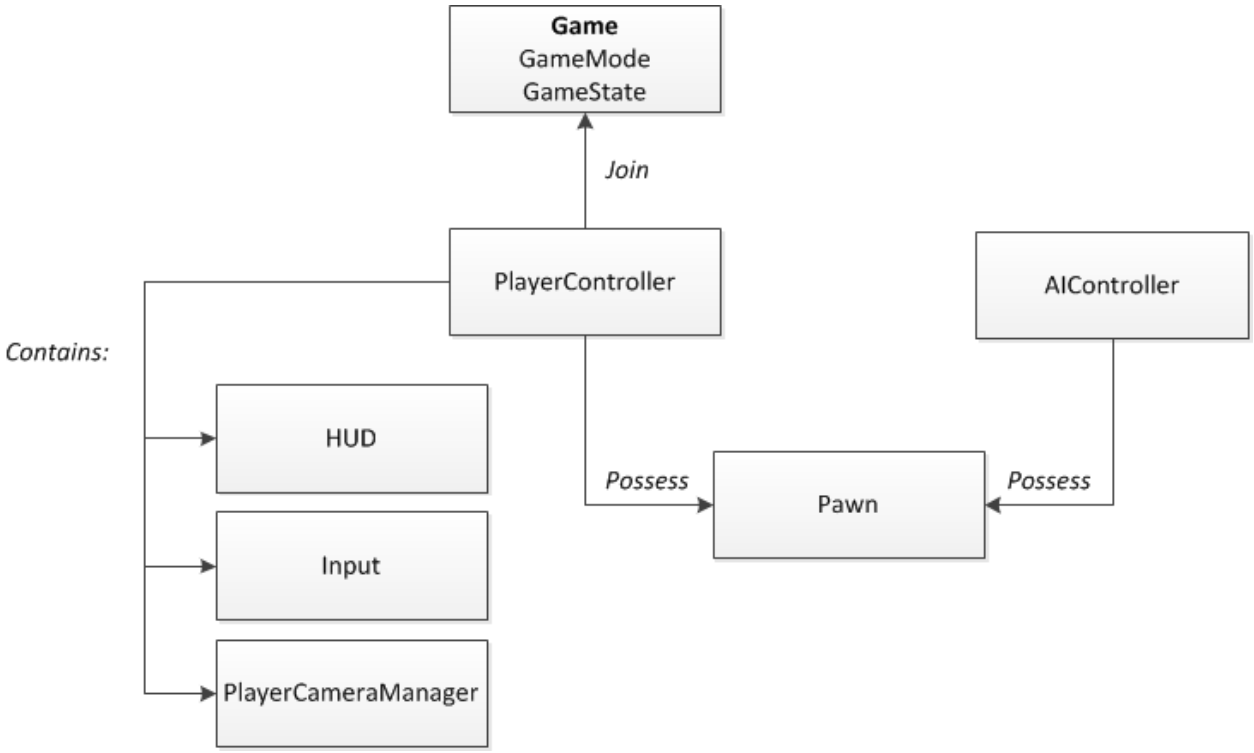
- Броят на присъстващите играчи и зрители, както и максималният брой играчи и зрители;
- Как играчите влизат в играта, които могат да включват правила за избор на места за инстанциране (spawn) и друго поведение;
- Дали играта може да бъде поставена на пауза и правилата на паузата;
- Преходи между нива, включително дали играта трябва да започне кинематографичен монтаж.

В случая игровият режим се грижи и за състоянието на играта, ресурси и стигането на информация за тях до играча, отговаря за инстанцирането и позиционирането на враговете в сцената и пази информация нужна за функционирането на враговете, като техните цели и т.н.

В началото на играта игровия режим запазва референция към всеки обект от тип AZombieBarrier(Виж 1.4.2) в масив *Targets*. (ZombieSurvivalFPSGameMode.cpp/64)

1.1.2. Player Controller („Контролер на играча“) (Фиг 3.1.1.1)

Всяка пионка (“Pawn“, обект който може да бъде контролиран) се контролира или от контролер за играч (Player Controller) или от контролер с изкуствен интелект (AI Controller).



Фиг 3.1.1.1 Структура на архитектурата на Unreal за управление на игра

Тъй като играта се играе от един играч, имаме само една инстанция на Player Controller, който по подразбиране се контролира от играча и е връзката между играча и обекта, който той контролира.

1.2. Разделение на класовете

Всеки тип враг, оръжие, анимационен контролер (Animation Controller)(Виж 1.4.3.3) наследява съответно базов клас, който имплементира общи функционалности.

Функция	Съответстващ клас
Базов клас за враг	AZombieBase
Базов клас за оръжия	AWeaponBase
Базов клас за анимации	UZombieBaseAnimationInstance
Базов клас за “снаряд”	AProjectileBase
Базов клас за UI с един параметър	UOneParamWidget

1.3. Помощни класове

1.3.1. UInteractableComponent

Чрез този клас компонент всеки обект може да взаимодейства с друг в света. Предоставя интерфейси както за прости взаимодействие (като включване, изключване),така и за използване на предмети като оръжия с множество контроли.

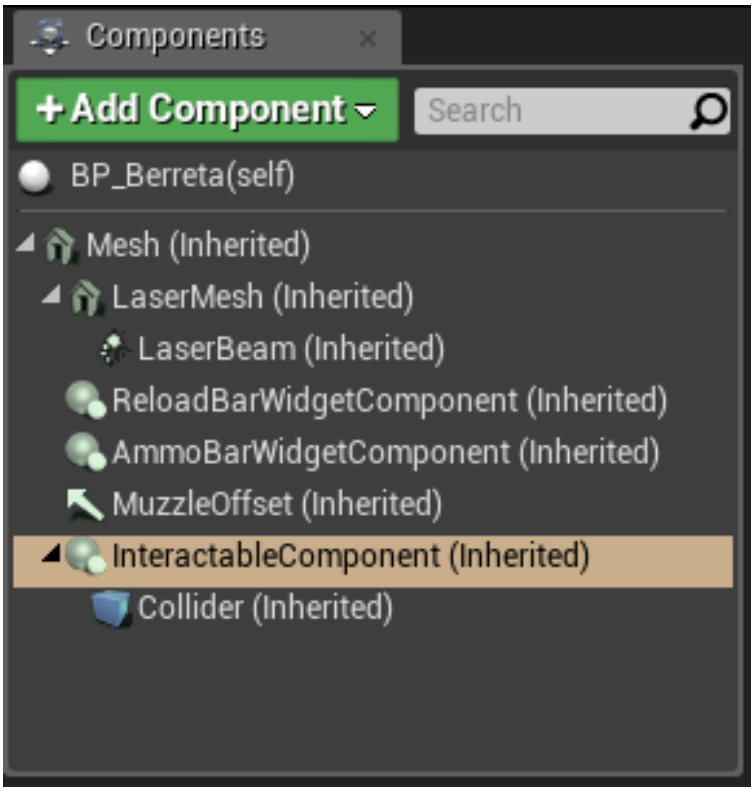
UInteractableComponent класа разчита на така наречените Delegates(делегати), които Unreal Engine поддържа.

Делегатите позволяват да бъдат извиквани член функции на C ++ класове по общ, но безвреден за типа начин (type-safe). Използвайки делегати, може динамично да се свържат с член функция на произволен обект. След това могат да бъдат извикани функции на обекта, през делегата.

Копирането на делегати е напълно безопасно. Те могат да бъдат предавани по стойност, но това обикновено не се препоръчва, тъй като те заделят памет в heap.

Делегатите биват няколко вида - Single-cast, Multi-cast, Events и Dynamic (UObject, serializable).

Всеки обект, който трябва да може да бъде използван или взаимодействан с, трябва да притежава UInteractableComponent в йерархията си от компоненти (Фиг 3.1.3.1) и да е инициализирал желаните методи.



Фиг 3.1.3.1 InteractableComponent в йерархията на компонентите на обект

Комуникацията се осъществява като при създаване на обекта на InteractableComponent-а се подават референции към вече създадени в обекта делегати. (Фиг 3.1.3.2, Фиг 3.1.3.3, Фиг 3.1.3.4)

```
//InteractableComponent.h
DECLARE_DELEGATE(SignatureOnFunction);
DECLARE_DELEGATE_OneParam(SignatureOnBeginGrab, USceneComponent*)
```

Фиг 3.1.3.2 Дефиниция на вече създадените делегатите

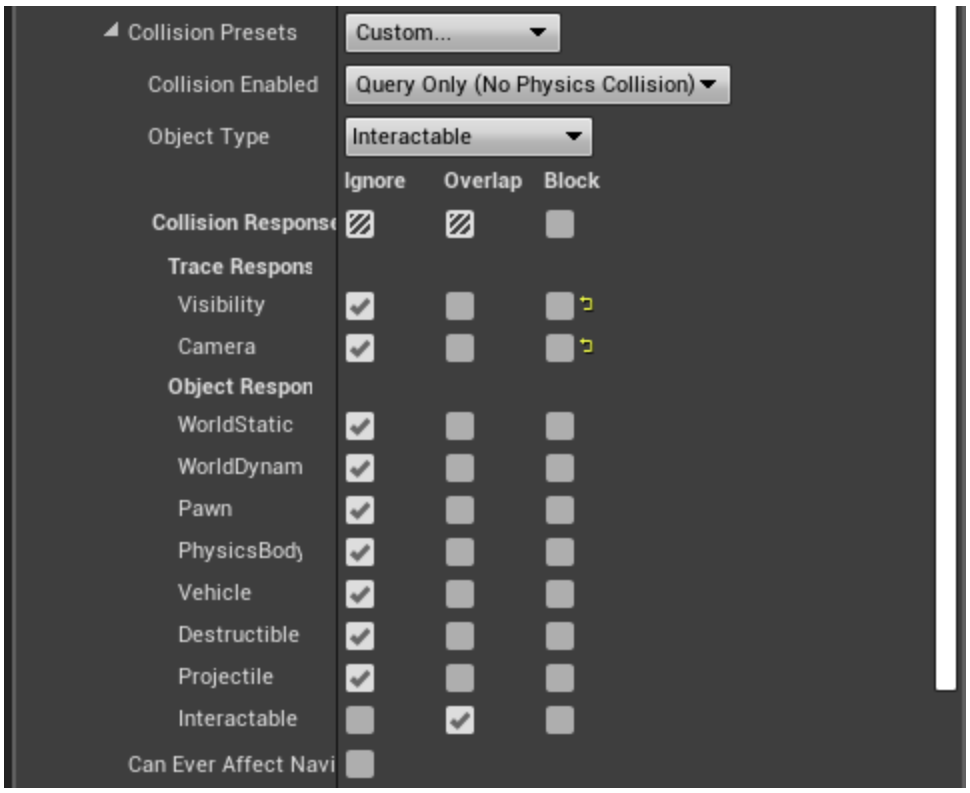
```
//WeaponBase.cpp
#include "InteractableComponent.h"

UCLASS(abstract)
class ZOMBIESURVIVALFPS_API AWeaponBase : public AActor
{
	GENERATED_BODY()

public:
	…
private:

	…
	SignatureOnFunction OnBeginUseDelegate;
	SignatureOnFunction OnEndUseDelegate;
	SignatureOnFunction OnSelectDelegate;
	SignatureOnFunction OnDeselectDelegate;
	SignatureOnBeginGrab OnBeginGrabDelegate;
	SignatureOnFunction OnEndGrabDelegate;
	SignatureOnFunction OnReloadDelegate;

};
```

Фиг 3.1.3.3 Делегати, които се подават по референция на InteractableComponent

```
//WeaponBase.cpp
// Called when the game starts or when spawned
void AWeaponBase::BeginPlay()
{
	Super::BeginPlay();

	…

	OnBeginUseDelegate.BindUFunction(this, TEXT("BeginUse"));
	OnEndUseDelegate.BindUFunction(this, TEXT("EndUse"));
	OnSelectDelegate.BindUFunction(this, TEXT("Select"));
	OnDeselectDelegate.BindUFunction(this, TEXT("Deselect"));
	OnBeginGrabDelegate.BindUFunction(this, TEXT("BeginGrab"));
	OnEndGrabDelegate.BindUFunction(this, TEXT("EndGrab"));
	OnReloadDelegate.BindUFunction(this, TEXT("Reload"));

	if (IsValid(InteractableComponent)) {
		InteractableComponent->InitializeUseDelegates(&OnBeginUseDelegate, &OnEndUseDelegate);
		InteractableComponent->InitializeSelectDelegates(&OnSelectDelegate, &OnDeselectDelegate);
		InteractableComponent->InitializeGrabDelegates(&OnBeginGrabDelegate, &OnEndGrabDelegate);
		InteractableComponent->InitializeActionDelegates(&OnReloadDelegate);
	}
	else {
		UE_LOG(LogTemp, Error, TEXT("Interactable Component is not valid."));
	}
}
```

Фиг 3.1.3.4 Свързване на член функции към съответните делегати и предаване на референция към тях на InteractableComponent

UInteractableComponent има дъщерен компонент UBoxComponent, чиято колизия се използва за разпознаване и управляване на компонента. Колизията е от тип “Interactable” и „отговаря“ само при припокриване с друга колизия от тип „Interactable” (Фиг 3.1.3.5)



Фиг 3.1.3.5 Настройки на колизията на UBoxComponent на UInteractableComponent

UInteractableComponent не имплементира никаква логика. Той е предназначен да бъде контролиран изцяло от външен обект, чрез следните публични методи (всеки извиква метода ExecuteIfBound() на съответния делегат):

- void BeginUse()
- void BeginAction()
- void BeginGrab(USceneComponent * AttachActor)
- void Select()
- void EndUse()
- void EndAction()
- void EndGrab()
- void Deselect()

1.3.2. UOneParamWidget

Този клас се използва като връзка между C++ и Blueprints. Наследява UUserWidget с цел в C++ да се пази референция към инстанция, която наследява този клас и имплементира неговият BlueprintImplementable метод – void UpdateParam(float Value). (Фиг 3.1.3.6)

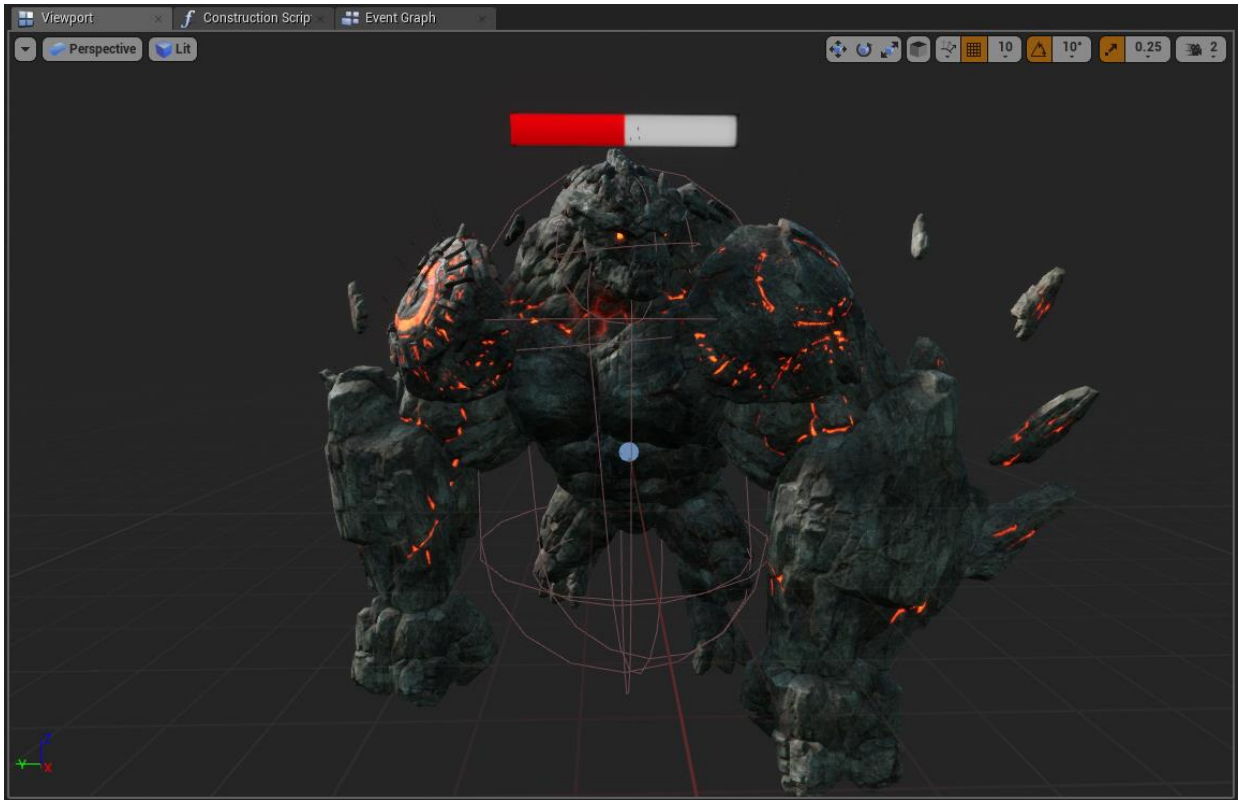
```
// OneParamWidget.h
UCLASS()
class ZOMBIESURVIVALFPS_API UOneParamWidget : public UUserWidget
{
	GENERATED_BODY()

public:

	UFUNCTION(BlueprintImplementableEvent)
	void UpdateParam(float Value);
};
```

Фиг 3.1.3.6 Дефиниция на член функцията UpdateParam

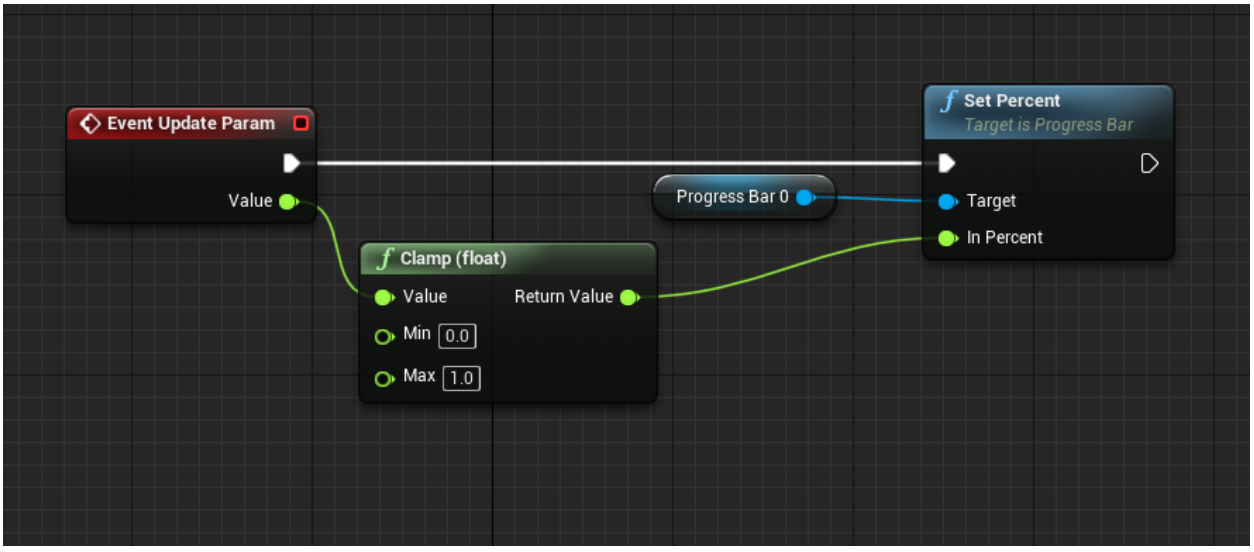
Когато някой клас наследи и имплементира метода UpdateParam, при извикване на функцията от C++ ще бъде извикана имплементацията на дъщерния клас (ако съществува такава).

Класове, които наследяват UOneParamWidget и техните имплементации:

- BP_HeatlhBar (Фиг 3.1.3.7 и Фиг 3.1.3.8)



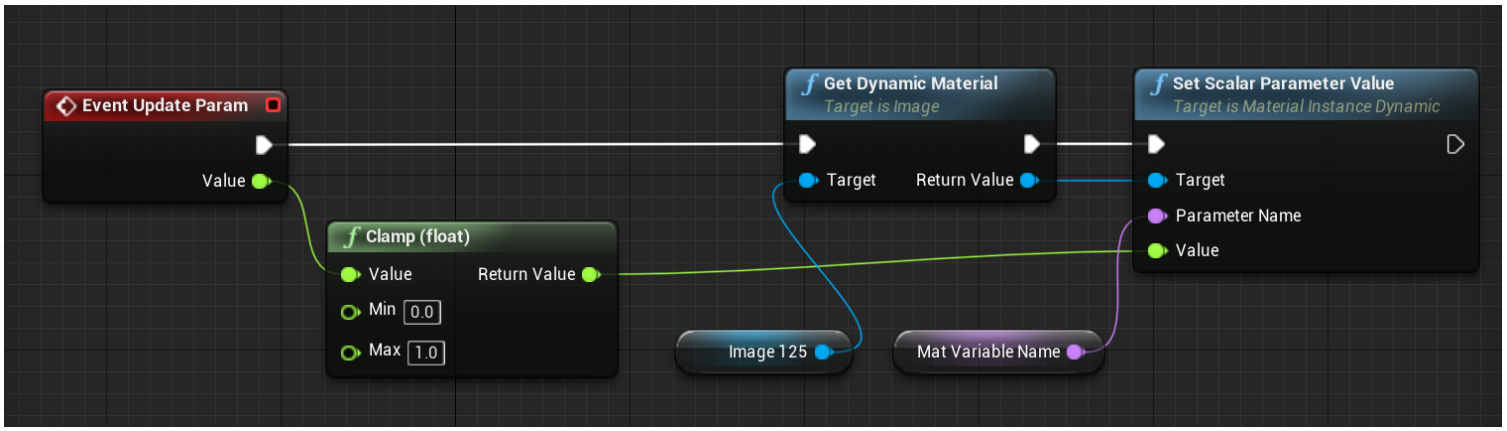
Фиг 3.1.3.7 BP_HealthBar и ZombieRampage в редактора



Фиг 3.1.3.8 Имплементацията на UpdateParam в BP_HealthBar

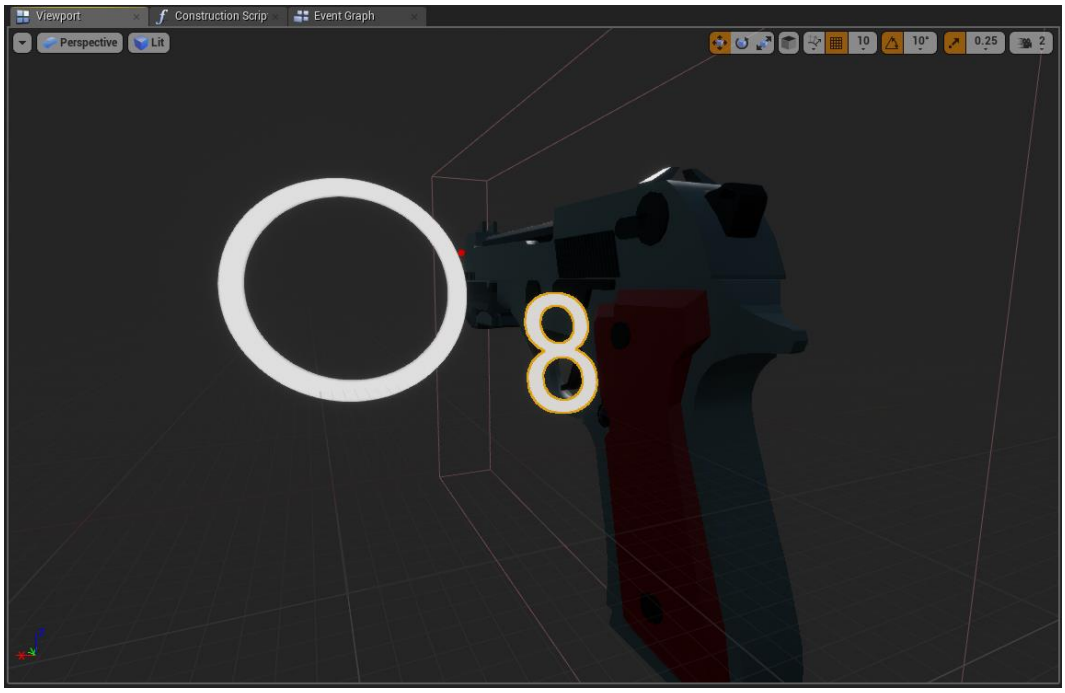
- BP_LoadingBar (Фиг 3.1.3.9 и Фиг 3.1.3.10)

Фиг 3.1.3.9 BP_LoadingBar

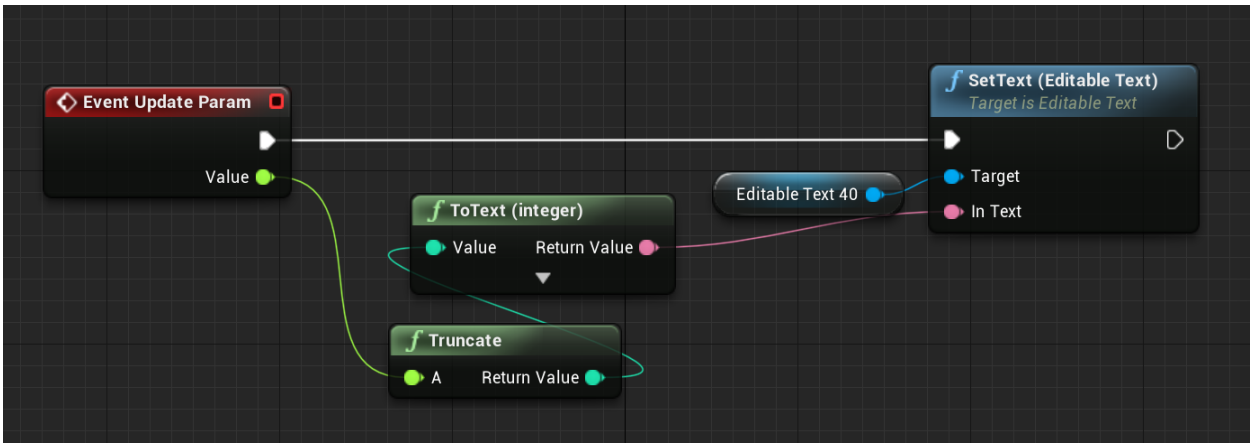


Фиг 3.1.3.10 Имплементация на UpdateParam в BP_LoadingBar

- BP_AmmoBar (Фиг 3.1.3.11 и Фиг 3.1.3.12)



Фиг 3.1.3.11 BP_AmmoBar



Фиг 3.1.3.12 Имплементацията на UpdateParam в BP_AmmoBar

1.3.3. UZombieBaseAnimationInstance

Подобно на UOneParamWidget (Виж 1.3.2) този клас се грижи изцяло за комуникация с Blueprints през C++, като позволява на Blueprint Child класовете да четат член променливи за статуса на обекта и да имплементират логика върху тях.

1.3.3.1. Основни Параметри

За изграждането на логиката на всички анимации са нужни следните параметри:

- bool bIsRunning
  Автоматично се опреснява в функцията NativeUpdateAnimation спрямо скоростта на движение.

- bool bIsAttacking
  OwningPawn прави променливата на true, а дъщерния клас я прави false когато приключи анимацията.

- bool bIsCheering
  OwningPawn прави променливата на true, когато играта приключи с загуба за играча.

- bool bIsDying
  OwningPawn прави променливата на true, когато анимацията приключи дъщерния клас изтрива обекта.

- AActor* Target
  Предава се от OwningPawn, за да може дъщерния клас да извика AZombieBase::DealDamageToTargetActor(*Target*), в определен момент от анимацията.

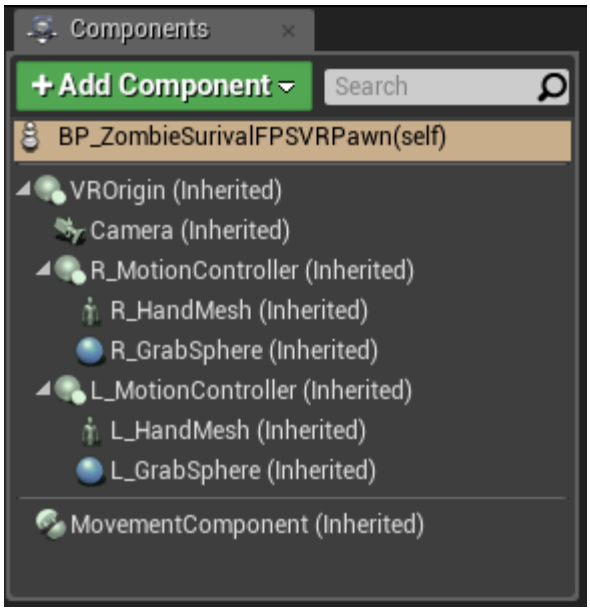
- AZombieBase* OwningPawn
  Дъщерните класове извикват методи чрез тази референция.

1.4. Главни класове на приложението

1.4.1. Главна игрова пионка (Player Pawn)

1.4.1.1. Базов Клас- AZombieSurvivalFPSVRPawn

Този клас се грижи за приемане на информацията от контролера и имплементиране на движение, стрелба, анимации, взаимодействие със средата (валидно за всеки един Pawn). В случая той създава и пази референция към шлема за виртуална реалност и двете устройства за проследяване на ръцете (Motion Controllers) (Виж 1.2.2). За всеки един от тях е закачена в йерархията на класа (Фиг 3.1.4.1) по една сфера за колизия, която да може да взаимодейства с Interactable компонентите.

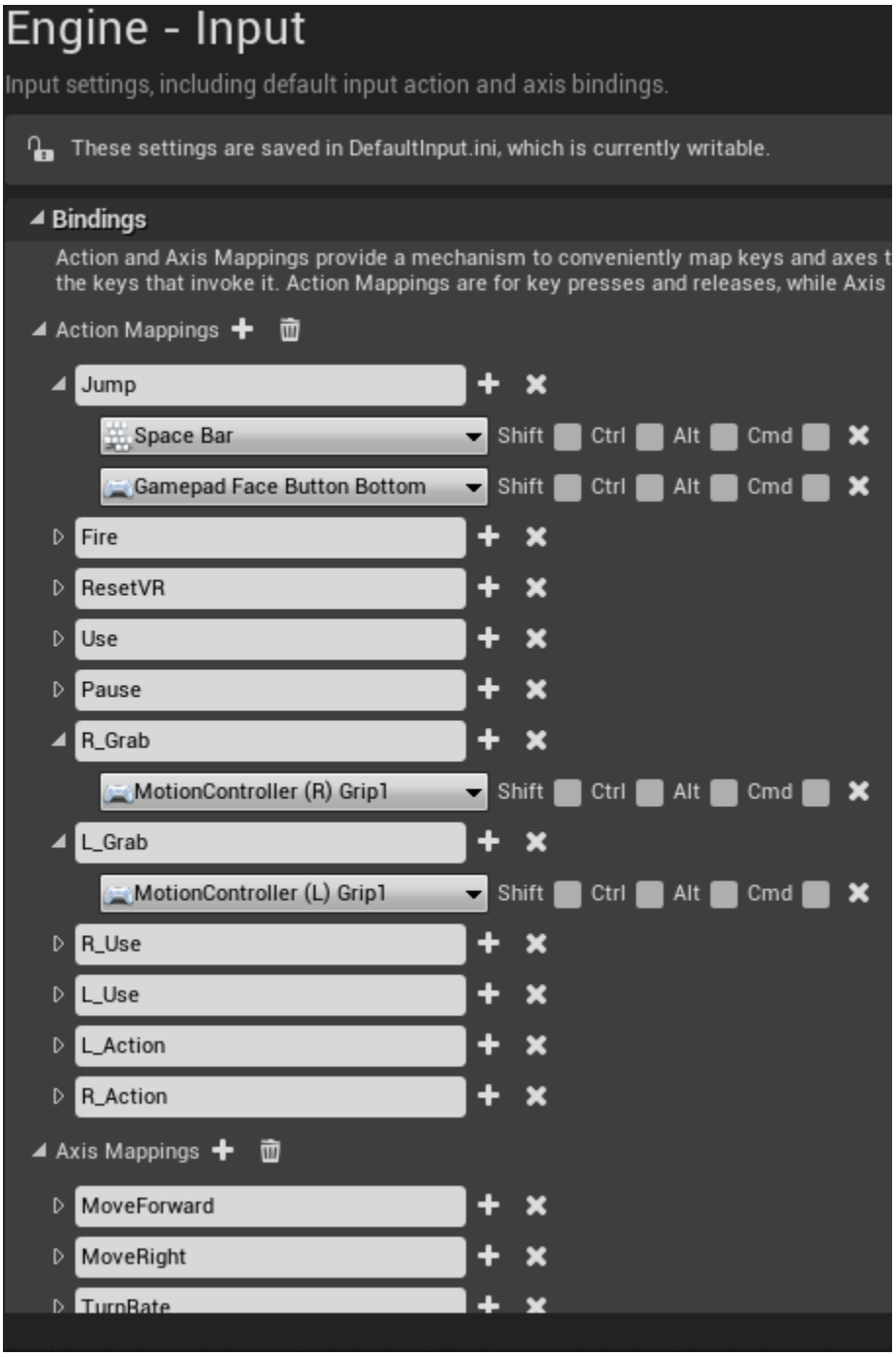


Фиг 3.1.4.1 Йерархията от компоненти на AZombieSurvivalFPSVRPawn

1.4.1.2. Input

В Unreal Engine системата за приемане на информация от играча се състои от две таблици (Фиг 3.1.4.2)

- FInputActionKeyMapping
- FInputAxisKeyMapping



Фиг 3.1.4.2 Конфигуриране на Input Mappings в редактора

След като са конфигурирани всички нужни Input Mappings, могат да бъдат имплементирани в код чрез функцията SetupInputComponent (Фиг 3.1.4.3).

```
// AZombieSurvivalFPSVRPawn.cpp
// Called to bind functionality to input
void AZombieSurivalFPSVRPawn::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
	Super::SetupPlayerInputComponent(PlayerInputComponent);

	// set up gameplay key bindings
	check(PlayerInputComponent);

	PlayerInputComponent->BindAction("ResetVR", IE_Pressed, this, &AZombieSurivalFPSVRPawn::OnResetVR);
	PlayerInputComponent->BindAction("Pause", IE_Pressed, this, &AZombieSurivalFPSVRPawn::Pause)
				.bExecuteWhenPaused = true;

	PlayerInputComponent->BindAction("R_Grab", IE_Pressed, this, &AZombieSurivalFPSVRPawn::R_BeginGrab);
	PlayerInputComponent->BindAction("R_Grab", IE_Released, this, &AZombieSurivalFPSVRPawn::R_EndGrab);
	PlayerInputComponent->BindAction("R_Use", IE_Pressed, this, &AZombieSurivalFPSVRPawn::R_BeginUse);
	PlayerInputComponent->BindAction("R_Use", IE_Released, this, &AZombieSurivalFPSVRPawn::R_EndUse);
	PlayerInputComponent->BindAction("R_Action", IE_Pressed, this, &AZombieSurivalFPSVRPawn::R_BeginAction);
	PlayerInputComponent->BindAction("R_Action", IE_Released, this, &AZombieSurivalFPSVRPawn::R_EndAction);

	PlayerInputComponent->BindAction("L_Grab", IE_Pressed, this, &AZombieSurivalFPSVRPawn::L_BeginGrab);
	PlayerInputComponent->BindAction("L_Grab", IE_Released, this, &AZombieSurivalFPSVRPawn::L_EndGrab);
	PlayerInputComponent->BindAction("L_Use", IE_Pressed, this, &AZombieSurivalFPSVRPawn::L_BeginUse);
	PlayerInputComponent->BindAction("L_Use", IE_Released, this, &AZombieSurivalFPSVRPawn::L_EndUse);
	PlayerInputComponent->BindAction("L_Action", IE_Pressed, this, &AZombieSurivalFPSVRPawn::L_BeginAction);
	PlayerInputComponent->BindAction("L_Action", IE_Released, this, &AZombieSurivalFPSVRPawn::L_EndAction);


	// Bind movement events
	PlayerInputComponent->BindAxis("MoveForward", this, &AZombieSurivalFPSVRPawn::MoveForward);
	PlayerInputComponent->BindAxis("MoveRight", this, &AZombieSurivalFPSVRPawn::MoveRight);
	PlayerInputComponent->BindAxis("Turn", this, &AZombieSurivalFPSVRPawn::Turn);
}
```

Фиг 3.1.4.3 Закачане на Input Mappings към член функции

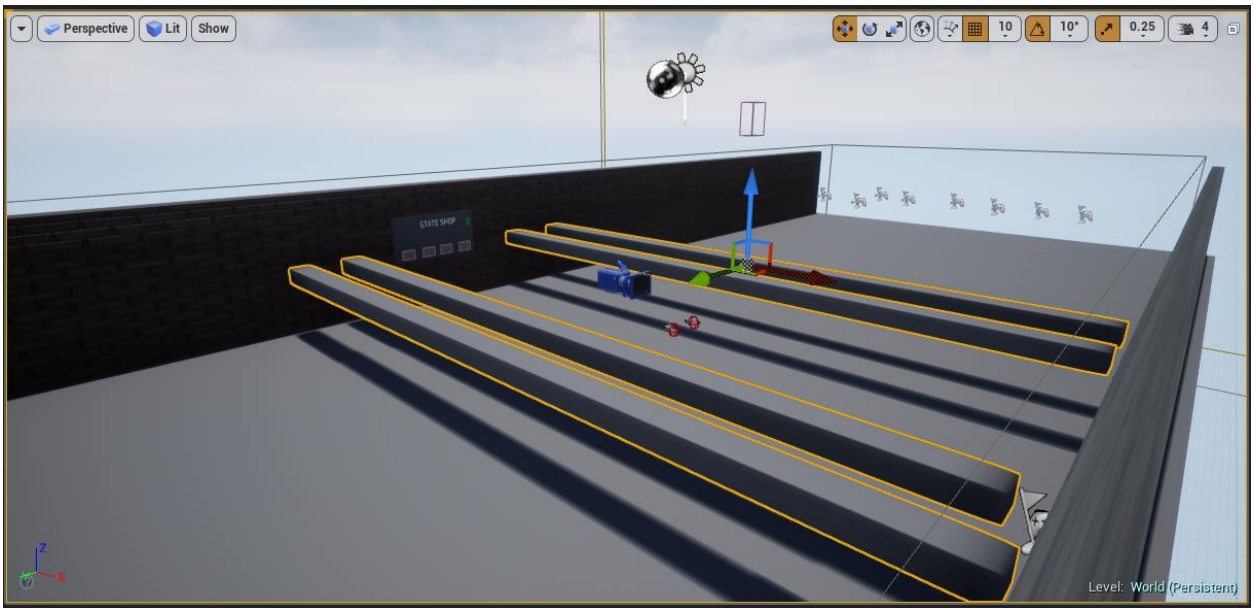
1.4.1.3. Pawn Movement – движение на обекта

За движение във Виртуалната Реалност се използва метод наречен VR Comfort Mode, който се имплементира, чрез резки завъртания на определен градус (45°). Този метод максимално намалява причиняването на гадене и замайване при продължително използване на Виртуалната Реалност.

1.4.2. Бариери

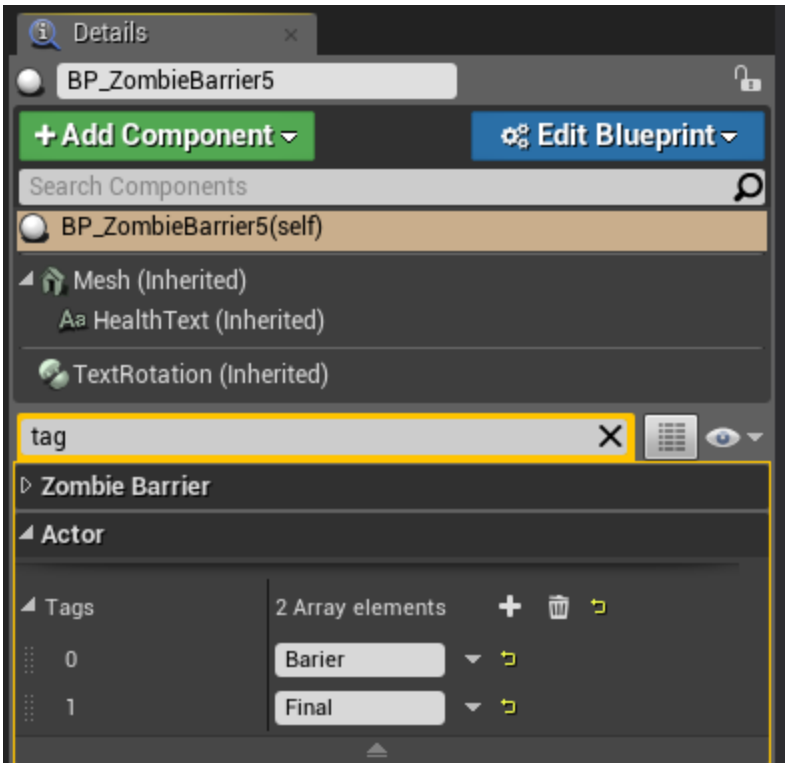
1.4.2.1. Базов Клас- AZombieBarrier

Този клас имплементира мишените, които враговете се опитват да унищожат.(Фиг 3.1.4.4)



Фиг 3.1.4.4 AZombieBarrier инстанции в редактора

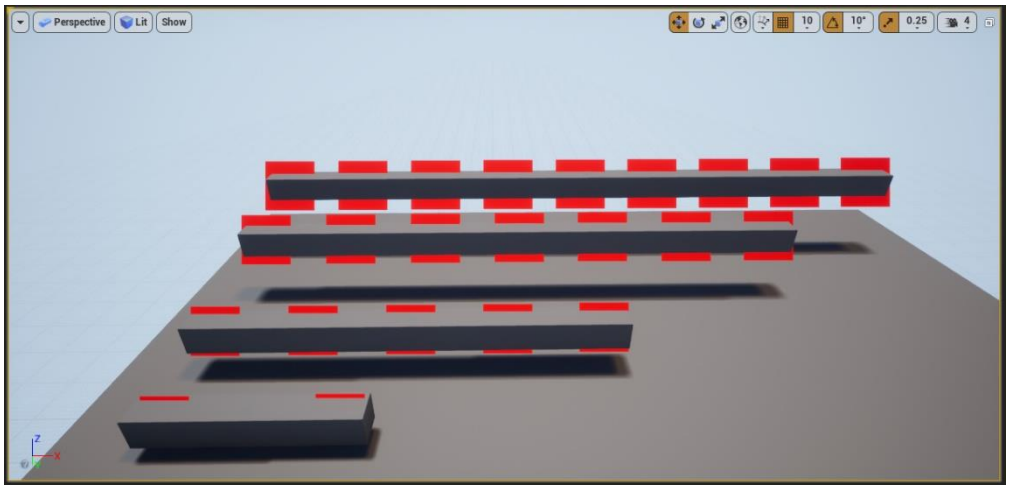
Класа имплементира метода TakeDamage и чрез него губи жизнени точки. Когато те стигнат 0 бариерата изчезва и това събитие се регистрира в игровия режим. Всяка бариера може да притежава Actor Tag – „Final” (Фиг 3.1.4.5) когато бариера с този таг бъде унищожена играта приключва.



Фиг 3.1.4.5 Лист от тагове, който AZombieBarrier притежава

1.4.2.2. Динамично генериране на точки за атака.

Когато зомбито атакува точка, тя не може да бъде само една (pivot локацията на обекта). За това всяка бариера динамично генерира точки по най-дългата и хоризонтална ос (*AZombieBarrier::GenerateTargetPoints).* (Фиг 3.1.4.6)



Фиг 3.1.4.6 Визуална репрезентация на динамично генерирани точки спрямо големината на бариерата

Масива TargetPoints е публичен и бива директно достъпван от AZombieAI (1.4.3.2), за да определи най-подходящата точка за атака.

1.4.3. Врагове

1.4.3.1. Базов Клас – AZombieBase

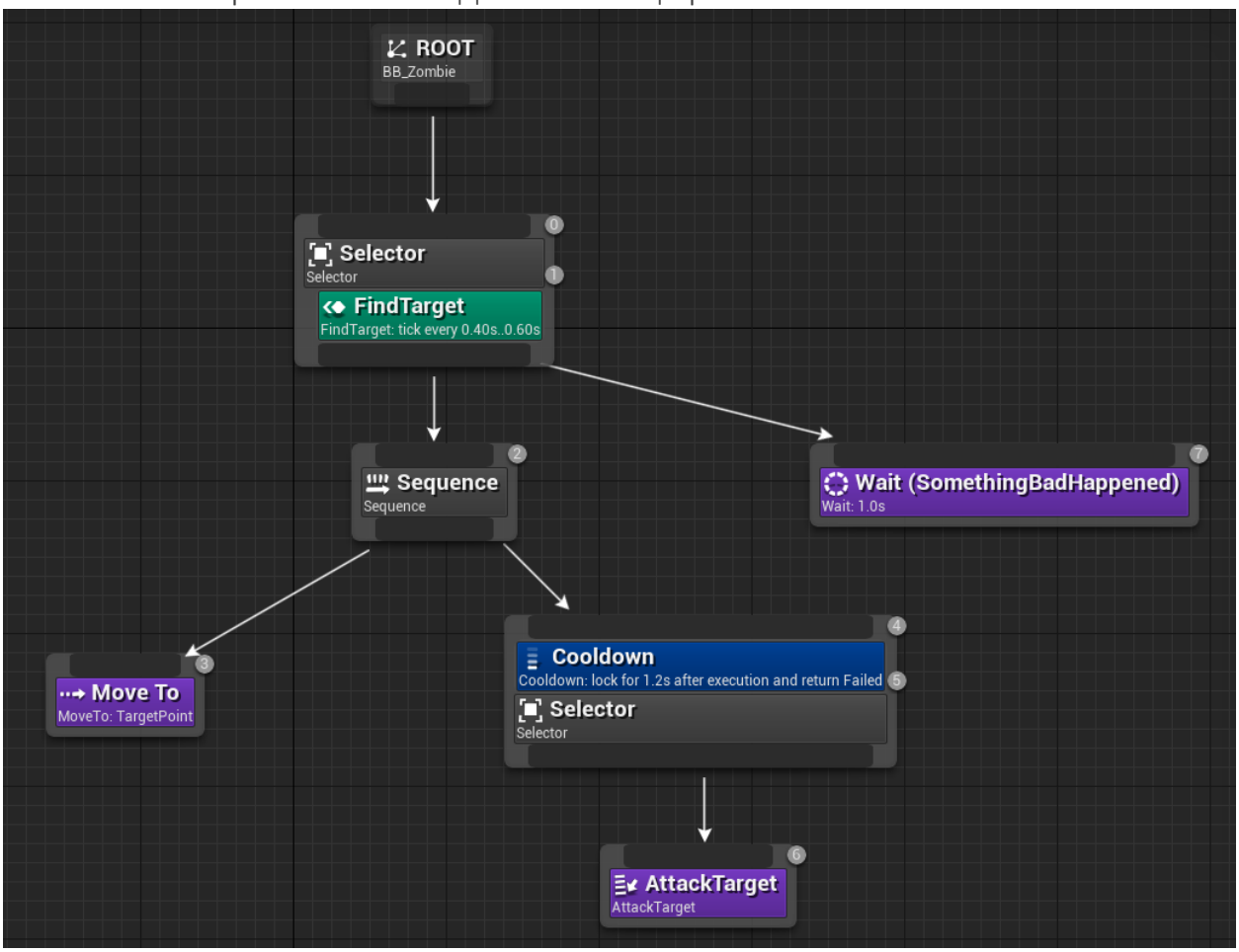
Основният клас, който всеки един враг (зомби) наследява е AZombieBase. Той имплементира част от логиката, но оставя имплементационни детайли като поведение при атака, как зомбито реагира на различни видове щети и колко и как са разположени капсулите за колизия (hitboxes), на наследяващите класове.

1.4.3.2. Изкуствен интелект (AI) – AZombieAI

AZombieAI наследява AAIController и е контролера, който управлява зомбитата. Всяко едно има инстанция на AZombieAI, която се грижи за събирането и менажирането на данните нужни за определяне на поведението.

Цялата логика се диктува от Behavior Tree („дърво“, което изпълнява логика. Често срещано при имплементация на AI) (Фиг 3.1.4.7). Unreal Engine дава множество логически елементи(nodes), с които да постигнем желаната логика както и поддържа имплементиране на нови от програмиста. Всяко зомби има собствена имплементация на “Behavior Tree” като референция към него пази AZombieBase. Изкуствения интелект за този проект е много прост. Единственото нещо което трябва да направят зомбитата е да се придвижат до точката и да изпълнят атака. Това трябва да се повтаря докато не приключи играта. Разликите в имплементацията са единствено колко близо трябва да са до целта и през колко време трябва да изпълнят атака, за да е балансирана играта. Изборът да съществуват N различни имплементации с разлика само параметрите, е че за пълна игра се очаква от зомбитата повече

от просто тичане и изпълняване на една атака, като всяко ново различно поведение е специфично.

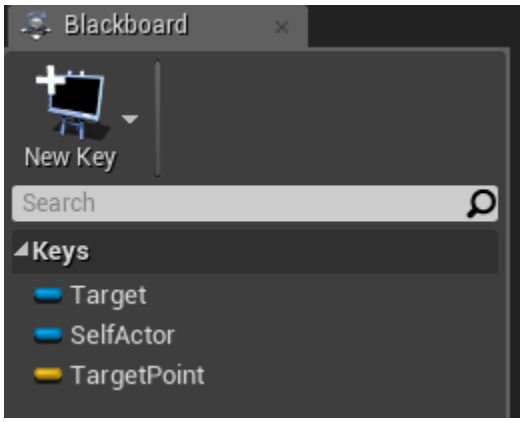


Фиг 3.1.4.7 Имплементацията на BT_Zombie

Добра практика при имплементирането на логика чрез Behavior Trees е да имаме „нещо не е наред“ Node. Така по време на търсене на бъг (debugging) може лесно да видим, ако дървото попадне върху този node. Едно от най-големите предимства на имплементация на изкуствен интелект чрез „дървета“, пред прости if-else блокове, е че търсенето на бъгове и проследяване на логиката става невероятно лесно и програмирането на поведението може да се прави от хора, който не са програмисти.

За да работи едно Behavior Tree с параметри, Unreal Engine имплементира BlackBoard. В йерархията на Behavior Tree съществува референция към инстанция на BlackBoard. В случая

имаме нужда от няколко параметъра (keys), за правилно функциониране на логиката (Фиг 3.1.4.8).



Фиг 3.1.4.8 Лист от ключове в BB_Zombie

- Target – Референция към целта като обект (AZombieBarrier)
- SelfActor
- TargetPoint – Вектор репрезентиращ точка в сцената. (Виж 1.4.2.2)

Използвани Nodes

- FindTarget - UBTTask_AttackTarget

При извикване на функцията BeginPlay, AZombieAI запазва локална константна референция към масива Targets в AZombieSurvivalFPSGameMode(Виж 1.1).

При изпълняването на този node се извиква метода AZombieAI::SetNewTarget, който се грижи за опресняването и поддържането на валидни данни в BlackBoard-а. Функцията SetNewTarget от своя страна извиква метода FindClosestTarget, който обхожда масива Targets(чрез референцията) и намира най-близката цел до обекта и я записва в BlackBoard-а, след това се извика методът FindClosestTargetPoint, който извлича от BlackBoard-а обекта, обхожда през целия масив *TargetPoints* на AZombieBarrier(Виж 1.4.2.2), избира най-близката точка за атака от тях и я записва в BlackBoard-а.

- MoveTo

MoveTo node използва TargetPoint като цел. Той ще позволи продължаване към следващия node само когато върне *Success* (логика на Sequencer), което означава, че текущия SelfActor е в *AcceptableRadius (Параметър на MoveTo Node)* единици от целта.

- AttackTarget – UBTTask_AttackTarget

Имплементацията на ExecuteTask(…) имплементирана от UBTTask_AttackTarget (Фиг 3.1.4.9) ще се извика само когато MoveTo върне *Success.* Това ще извика метода AttackTarget(…) на AZombieAI, който ще стартира атака.

```
//BTTask_AttackTarget.cpp
EBTNodeResult::Type UBTTask_AttackTarget::ExecuteTask(UBehaviorTreeComponent& OwnerComp,
uint8* NodeMemory) {

	AZombieAI * ZombieAI = Cast<AZombieAI>(OwnerComp.GetAIOwner());
	if (!IsValid(ZombieAI))
		return EBTNodeResult::Failed;

	AActor * Target = Cast<AActor>(OwnerComp.GetBlackboardComponent()-
>GetValue<UBlackboardKeyType_Object>(ZombieAI->TargetKeyId));

	if (IsValid(Target)) {
		ZombieAI->AttackTarget(Target);

		return EBTNodeResult::Succeeded;
	}

	return EBTNodeResult::Failed;
}
```

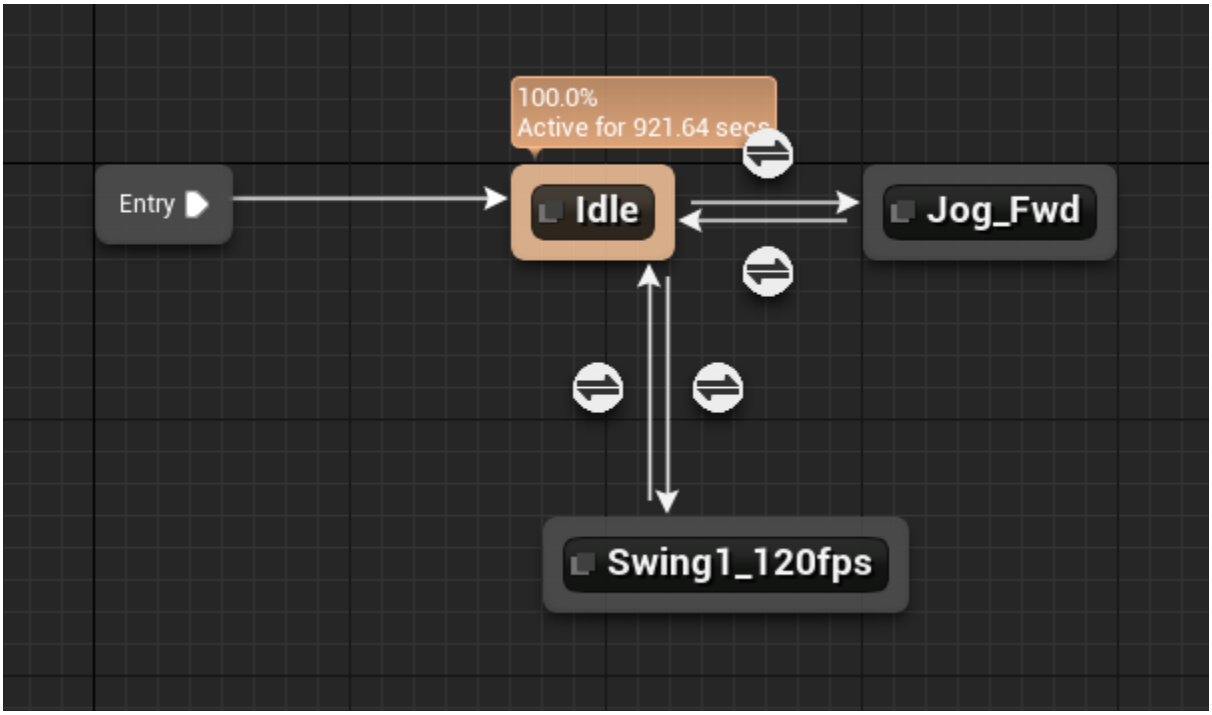
Фиг 3.1.4.9 Имплементация на ExecuteTask в UBTTask_AttackTarget

1.4.3.3. Анимации

Всички модели и анимации на врагове са взети от Unreal Marketplace.

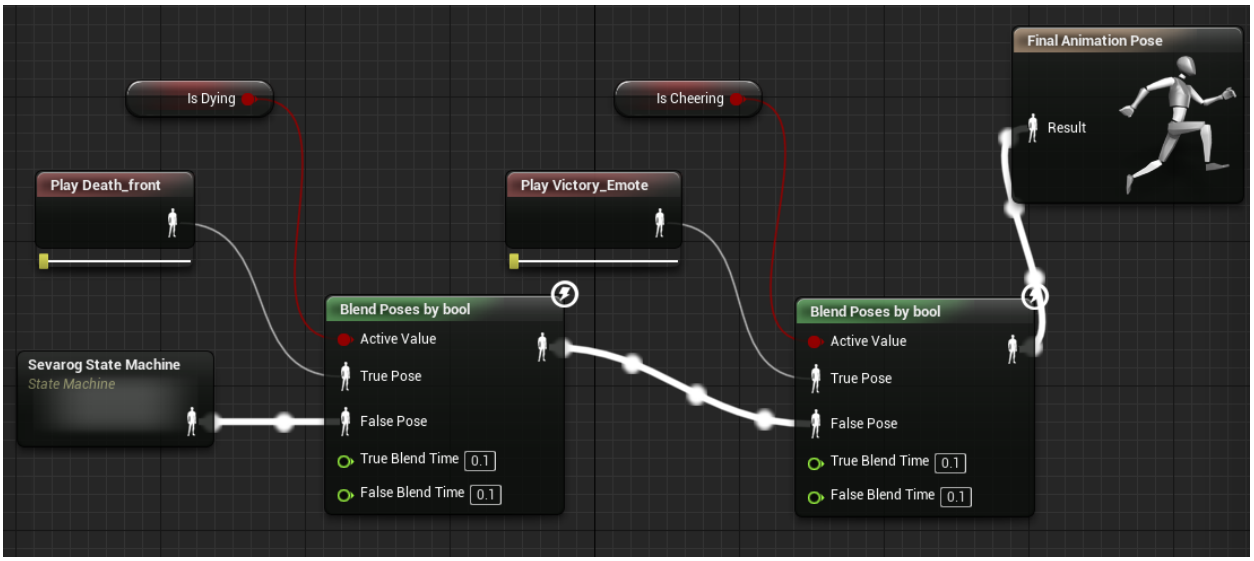
Animation Blueprint е специализиран тип Blueprint, който контролира анимацията на Skeletal Mesh (Комбинация от Mesh + навързани „bones”). В анимационният редактор, може да се извършва смесване на анимация, „костите“ на скелета или имплементиране на логика, която в крайна сметка ще определя крайната поза на анимацията за Skeletal Mesh, която да се използва в кадъра.

Финалната поза за кадъра се определя от Animation Graph, където освен node е State Machine(Фиг 3.1.4.10). Той се грижи за придвижване между различните анимации на база условие.



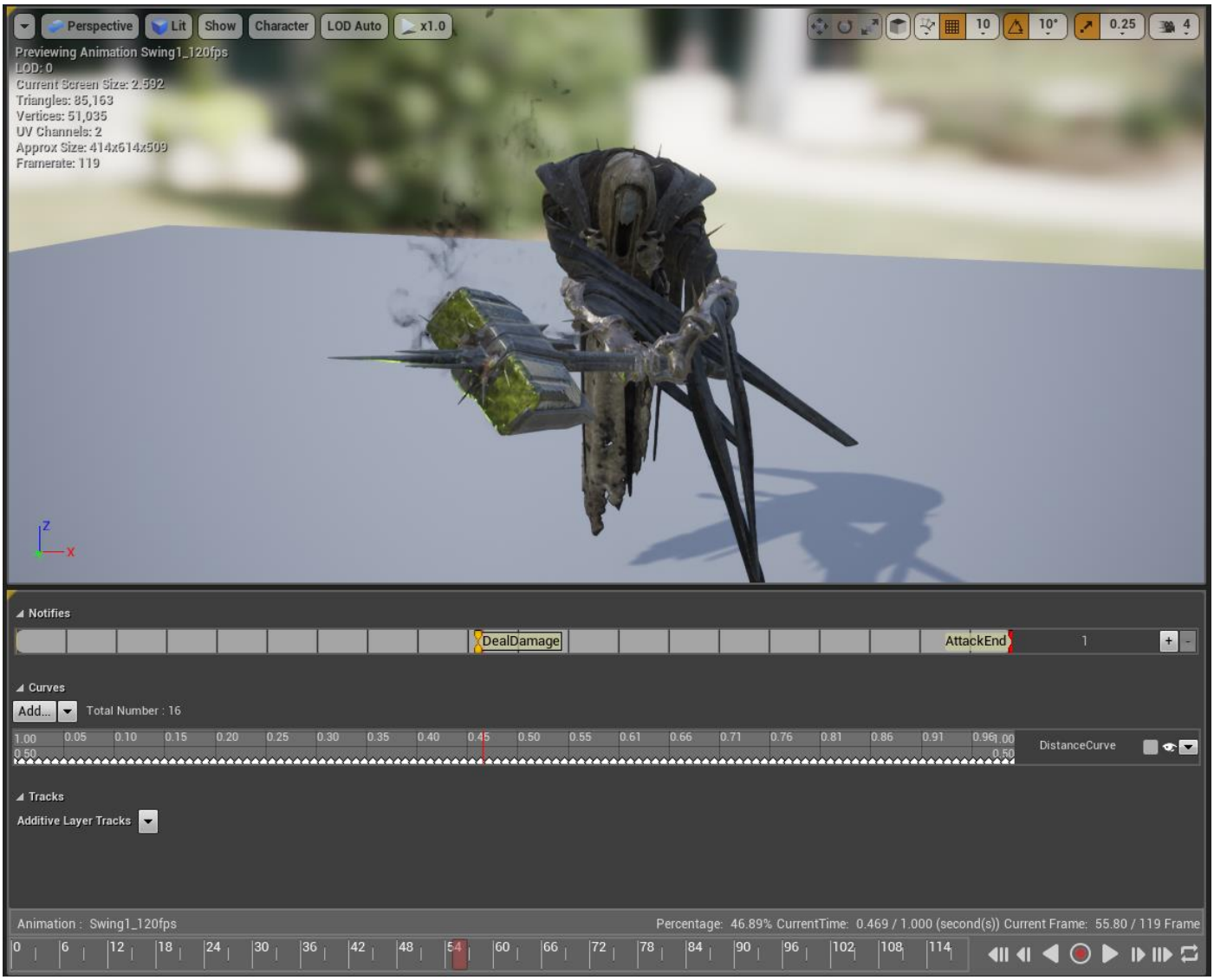
Фиг 3.1.4.10 Имплементация на State Machine в AnimBP_Sevarog

След като бъде определена базова анимация в Animation Graph (Фиг 3.1.4.11) се проверяват няколко финални условия за анимация, след което резултата се подава във финалния node.

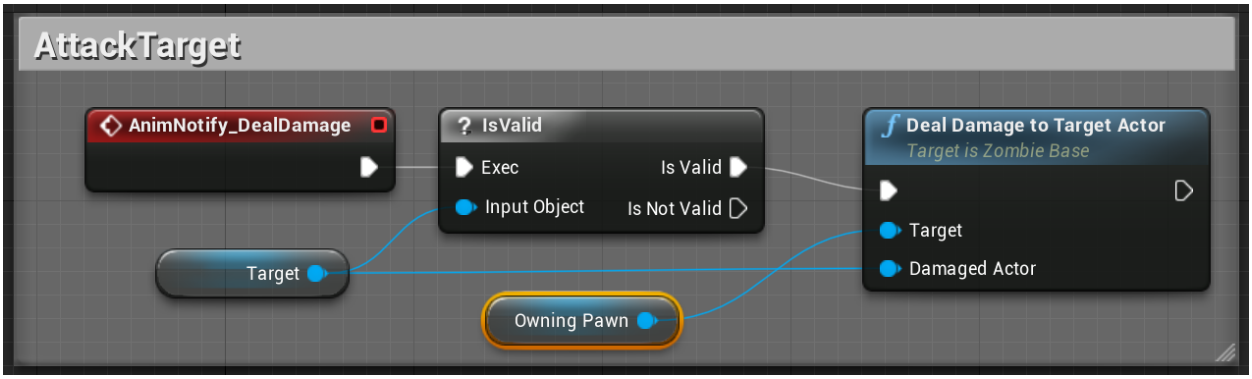


Фиг 3.1.4.11 Animation Graph в AnimBP_Sevarog

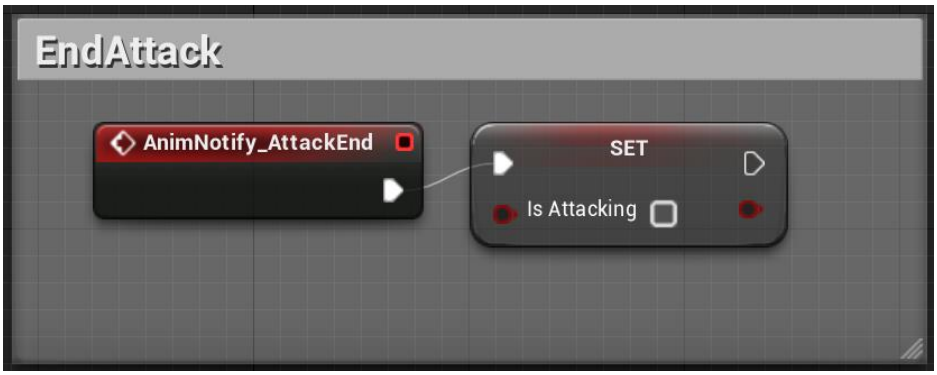
Animation Notifiers (Фиг 3.1.4.12) са специални събития, който могат да бъдат извиквани в точен кадър по време на анимацията. В текущата имплементация се използват с цел нанасяне на щети върху целта (Фиг 3.1.4.13) или известие за приключване на конкретна анимация(Фиг 3.1.4.14, Фиг 3.1.4.15).



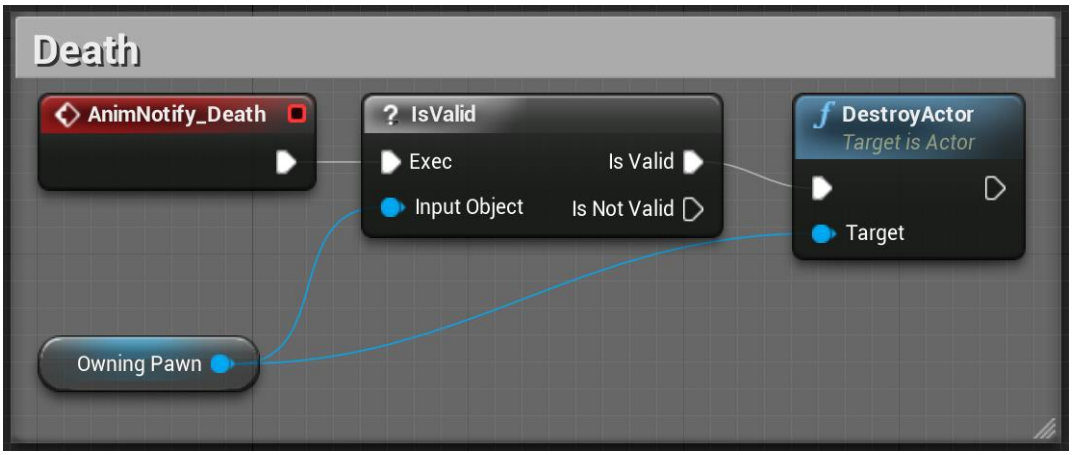
Фиг 3.1.4.12 Custom Anim Notifier Event DealDamage и AttackEnd Event в AnimBP_Sevarog



Фиг 3.1.4.13 Имплементация на AnimNotify_DealDamage



Фиг 3.1.4.14 Имплементация на AnimNotify_AttackEnd



Фиг 3.1.4.15 Имплементация на AnimNotify_Death

Всяка имплементация на Animation Blueprint е индивидуална за всеки враг.

1.4.3.4. Видове врагове

1.4.3.4.1. Sevarog – AZombieCharacter (Фиг 3.1.4.16)

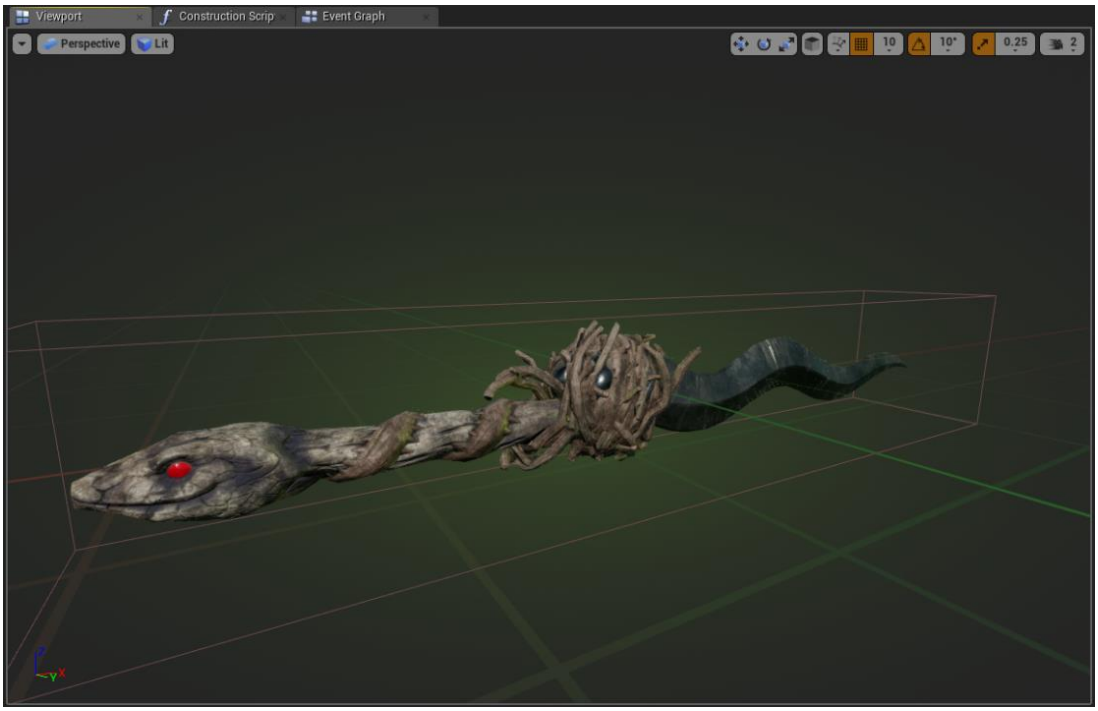
Фиг 3.1.4.16 Paragon Sevarog

- Mele
- Low Health
- Average Speed

1.4.3.4.2. Morigesh – AZombieMorigesh (Фиг 3.1.4.17)

Фиг 3.1.4.17 Paragon Morigesh

- Ranged (Фиг 3.1.4.18)
- Low Health
- High Speed



Фиг 3.1.4.18 BP_ZombieMorigeshDagger Projectile в редактора

1.4.3.4.3. Rampage – AZombieRampage (Фиг 3.1.4.19)

Фиг 3.1.4.19 Paragon Rampage

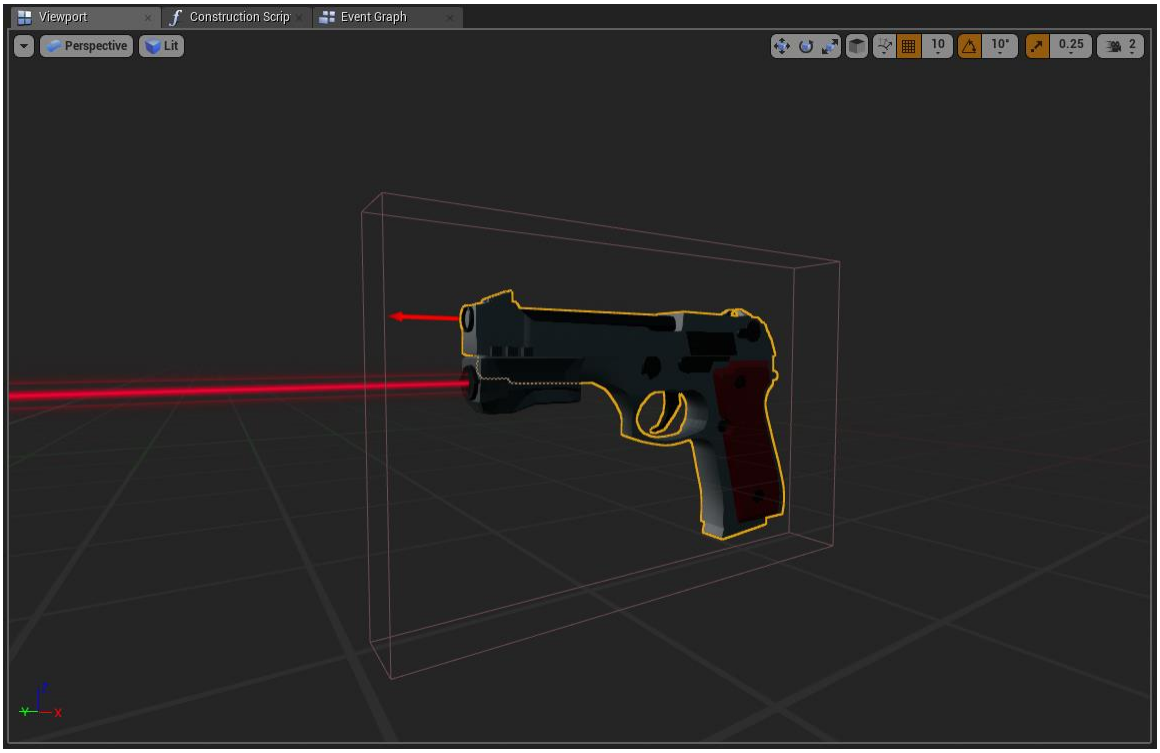
- Mele
- High Health
- Medium Speed

1.4.3.4.4. Grux – AZombieGrux (Фиг 3.1.4.20)

Фиг 3.1.4.20 Paragon Grux

- Mele
- High Health
- Slow Speed

1.4.4. Оръжия (Фиг 3.1.4.21)



Фиг 3.1.4.21 BP_Beretta в „Blueprint" редактора

1.4.4.1. Базов Клас – AWeaponBase

Основният клас, който всички оръжия наследяват чрез "Blueprint" е AWeaponBase. Имплементира стрелба (както автоматична, така и ръчна), зареждане, потребителски интерфейс и предоставя интерфейс за контрол през InteractableComponent. (Виж 1.3.1)

1.4.4.2. Основни параметри (Фиг 3.1.4.22)

Фиг 3.1.4.22 Полета на основните параметри в редактора

- Damage - Щети на удар
- Fire Rate - Брой куршуми в минута (При автоматични оръжия)
- Magazine Size - Максимум брой куршуми в пълнителя
- Reload Time - Време на зареждане в секунди
- Is Automatic - Дали оръжието е автоматично

1.4.4.3. Имплементирана логика

1.4.4.3.1. Презареждане

Презареждане се извиква автоматично при опит за стрелба без патрони в пълнителя и ръчно при натискане на бутона за презареждане.

Тъй като трябва постоянно опресняване на потребителския интерфейс прост таймер, който приключва след *ReloadTime* време не върши работа. Вместо това презареждането се имплементира чрез таймер, който тактува *UPDATE_TICKS* (Фиг 3.1.4.23) пъти през *ReloadTime/UPDATE_TICKS* време.

```
//WeaponBase.h
//Amount of times reload widget is updated
#define UPDATE_TICKS 50
```

Фиг 3.1.4.23 *Дефиниция на UPDATE_TICKS*

След всеки тик на таймера се извиква член функцията TickReload, която приключва презареждането когато отброи *UPDATE_TICKS* пъти, като на всяко извикване опреснява потребителския интерфейс чрез BP_LoadingBar(Виж 1.3.2) (Фиг 3.1.4.24).

Фиг 3.1.4.24 Опресняване на потребителския интерфейс

1.4.4.3.2. Стрелба

Има два основни подхода за имплементиране на стрелба: чрез изстрелване на куршум (Arma, всяка игра в която куршумът се влияе от външни сили) или чрез директен ray-cast (Counter-Strike поредица)

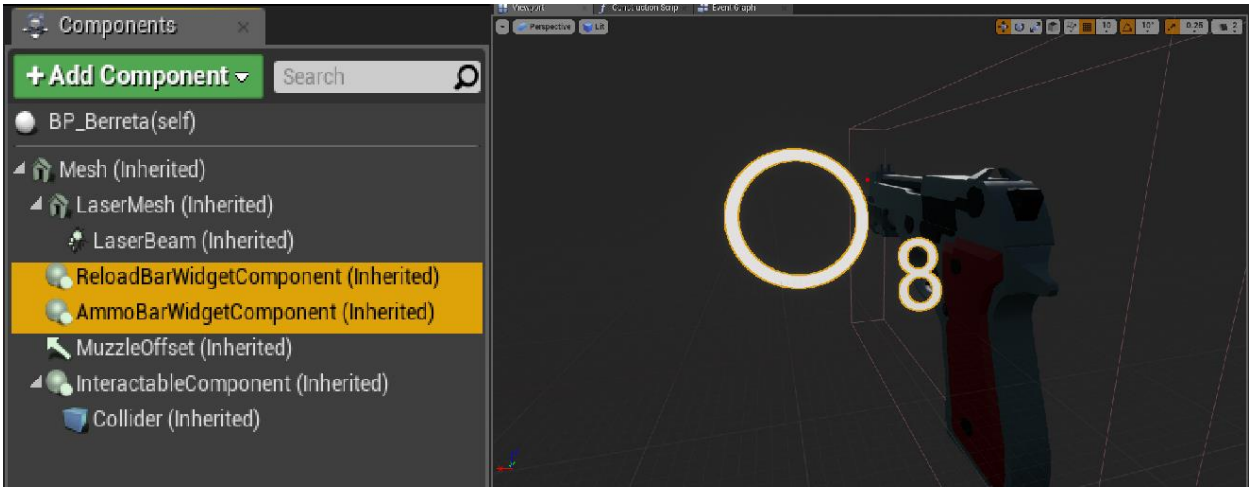
Изстрелването на куршум има едно голямо предимство – реализъм. Куршумът може да бъде повлиян от всякакви фактори, чрез физика на играта (тегло, скорост, гравитация и

т.н.). Но това от своя страна означава обременяване на програмиста и разликата е видима само при по-големи разстояния. Поради тази причина имплементацията на стрелбата в този проект се случва чрез Ray-Casting.

Автоматичната стрелба е имплементирана по подобен начин на презареждането като при натискане на бутона се задейства таймер, който извиква функцията Fire на всеки *60 / FireRate* секунди (FireRate е в куршуми в минута, а таймерите работят в секунди). Таймера се прекратява когато играчът спре да натиска бутона.

1.4.4.3.3. Потребителски интерфейс (UI)

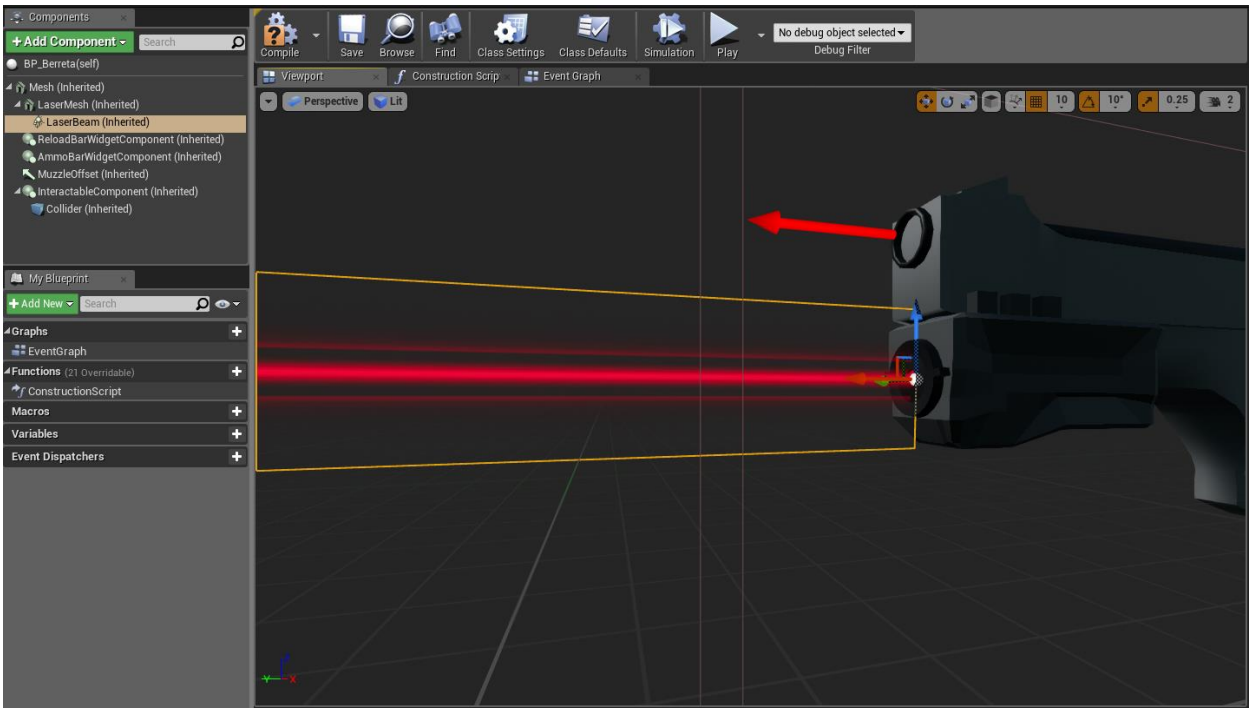
За разлика от типичните компютърни игри, във Виртуална Реалност нямаме типичният потребителски интерфейс върху целия екран („HUD"). Избраният начин за обратна връзка към потребителя е имплементирана чрез 3D Widgets закачени директно за самото оръжие (Фиг 3.1.4.25) (Виж 1.3.2). AWeaponBase се грижи за опресняването на информацията на тези "3D Widgets".



Фиг 3.1.4.25 Потребителски интерфейс на оръжие

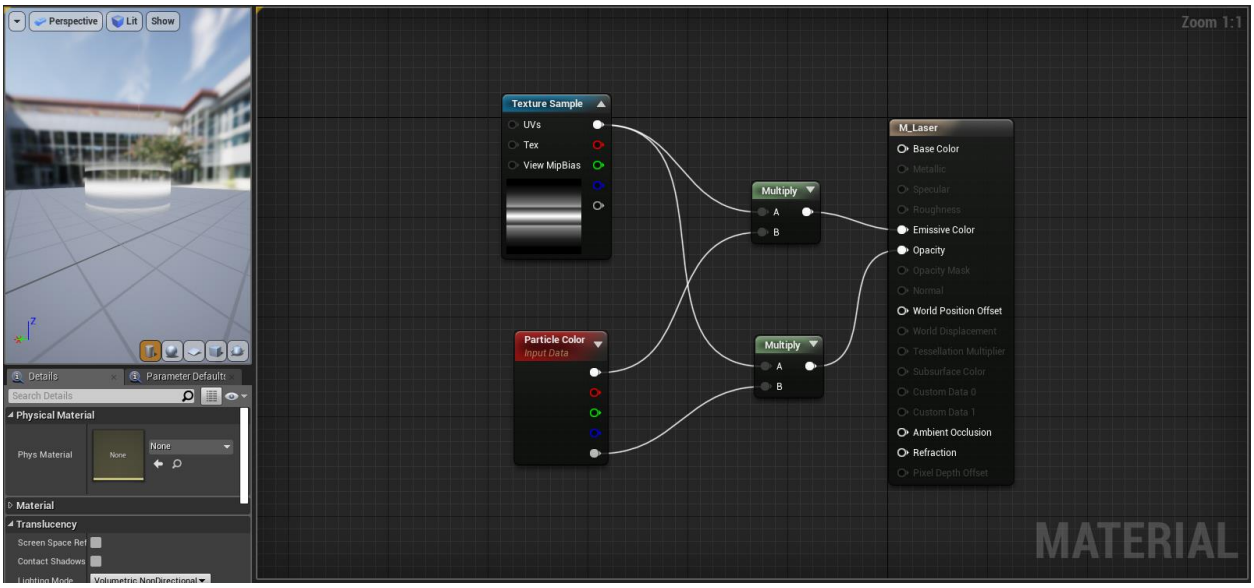
1.4.4.3.4. Лазер

Във Виртуална Реалност удобен начин за прицелване е чрез лазери (Laser Sight). Лазерът (Фиг 3.1.4.26) е имплементиран чрез Cascade Particle Systems в Unreal Engine.

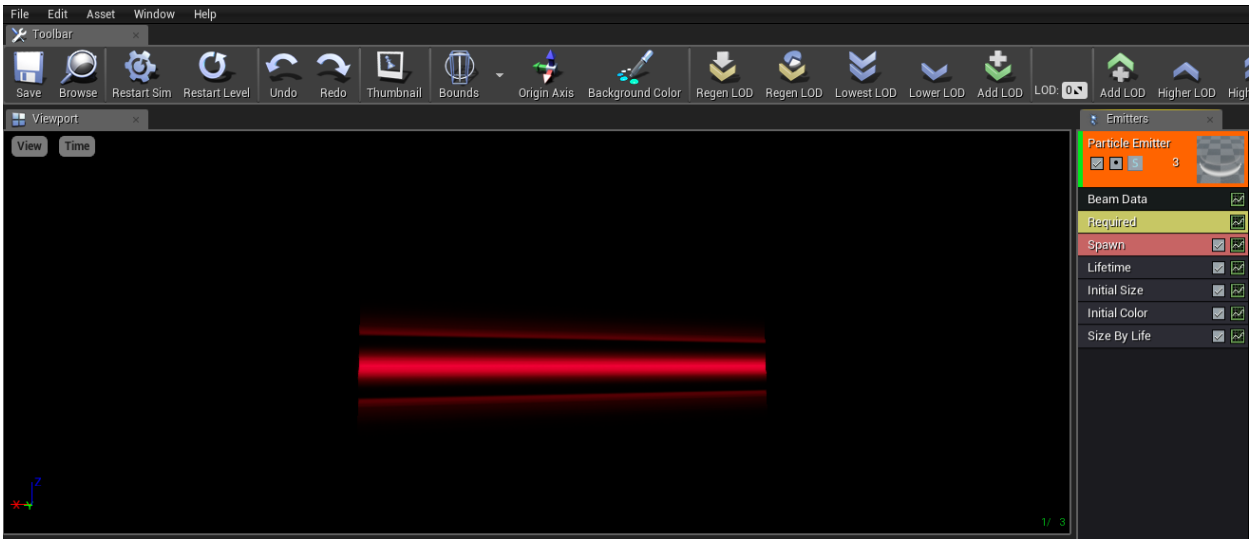


Фиг 3.1.4.26 Потребителски интерфейс на оръжие

За да се постигне ефекта чрез Cascade Particle System е необходим базов материал на лазер – *M_Laser* (Фиг 3.1.4.27). Той лежи в основата на *PS_LaserBeam* (Фиг 3.1.4.28).



Фиг 3.1.4.27 *Имплементация на материал лазер*



Фиг 3.1.4.28 *Имплементация на Particle System лазер*

Управлението на лазера става посредством динамично променяне на параметър на инстанцията на системата за частици (Particle System). Дължината се определя като на всеки кадър се използва ray-cast, за да бъде опредено точното разстояние от точката на „изстрелване“ на лазера до първия обект. (Фиг 3.1.4.29)

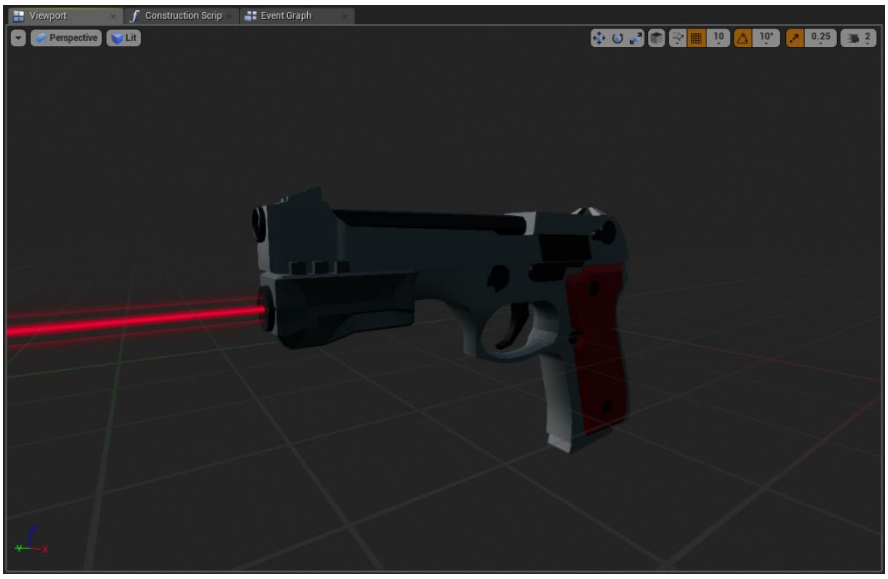
```cpp
//WeaponBase.cpp
float AWeaponBase::CalculateLaserBeamDistance()
{
	FVector Start = LaserBeam->GetComponentLocation();
	FVector Forward = LaserBeam->GetForwardVector();
	//Multiply forward vector by X moves it "forward" x units
	FVector End = Start + (Forward * 3000);

	FCollisionQueryParams Params = FCollisionQueryParams(FName("LaserBeamTrace"));

	FHitResult RV_Hit(ForceInit);

	//call GetWorld() from within an actor extending class
	GetWorld()->LineTraceSingleByChannel(
		RV_Hit,
		Start,
		End,
		ECollisionChannel::ECC_Visibility,
		Params
	);

	if (RV_Hit.IsValidBlockingHit()) {

		return RV_Hit.Distance;
	}

	//Hit nothing
	return 3000.f;
}
```

Фиг 3.1.4.29 Калкулиране на дължината на лазера

1.4.4.3.5. Различни видове оръжия

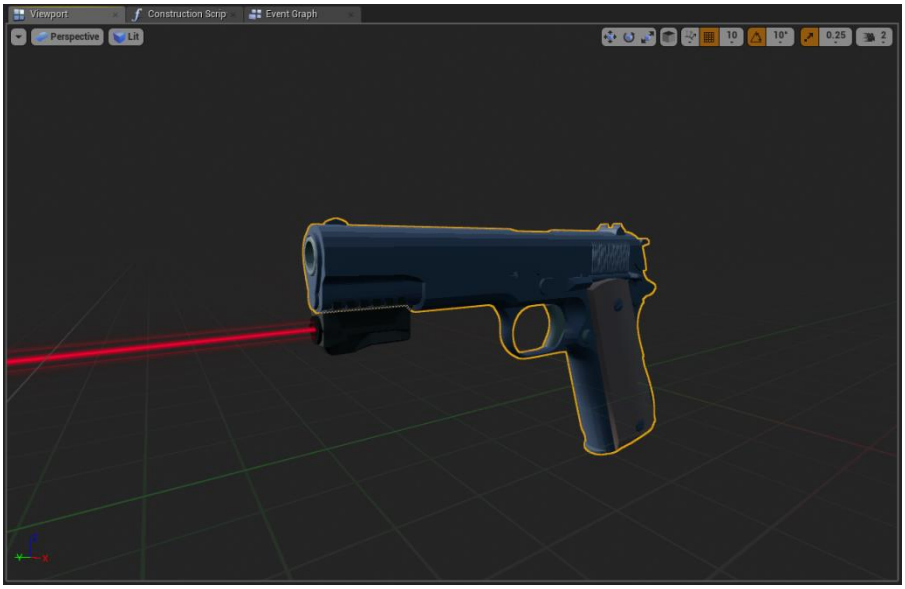
Моделите са свалени готови (Виж Глава 4 /реф/) с малки промени. Параметрите на оръжията са максимално близки до реалните им прототипи. Добавянето на ново оръжие става чрез наследяване на AWeaponBase в Blueprints, и променянето на няколко параметъра.

- Beretta (Фиг 3.1.4.30)



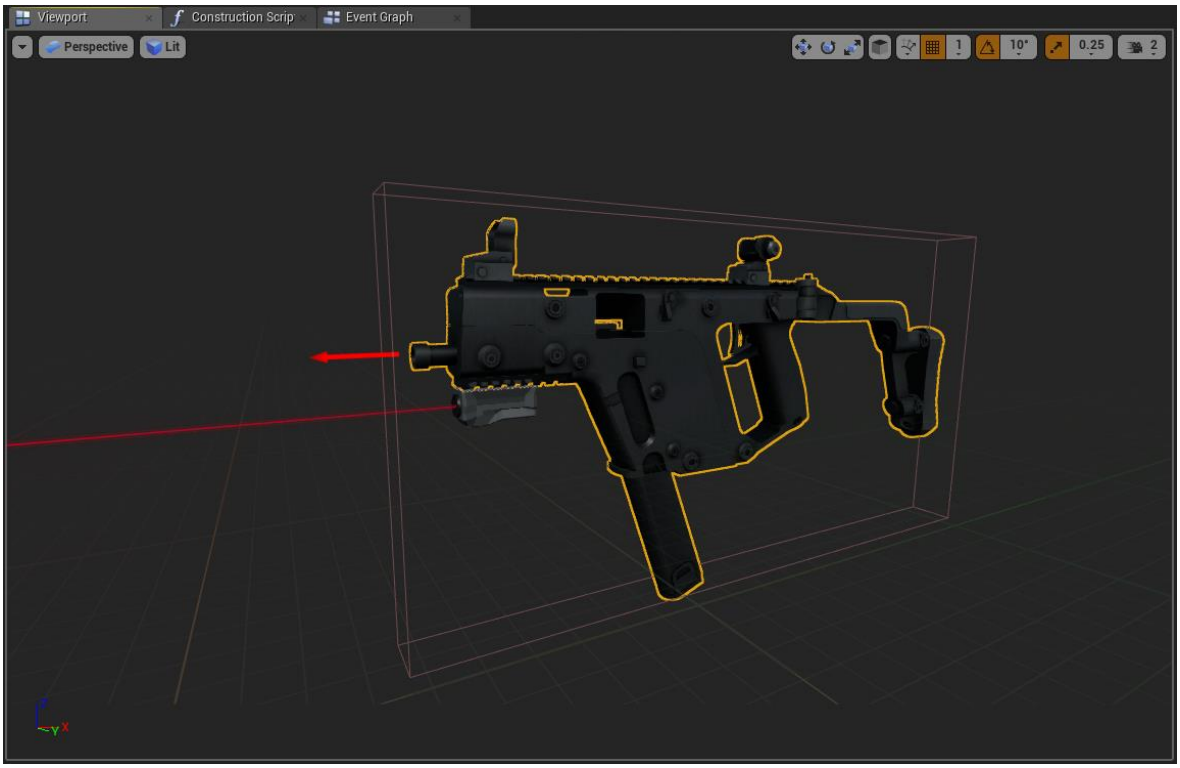
Фиг 3.1.4.30 Оръжие "Beretta" в редактора

- Colt (Фиг 3.1.4.31)



Фиг 3.1.4.31 Оръжие "Colt" в редактора

- **Vector** (Фиг 3.1.4.32)



Фиг 3.1.4.32 Оръжие "Vector" в редактора

1.4.5. Магазин (Фиг 3.1.4.33)

Фиг 3.1.4.33 Магазин за оръжия

1.4.5.1. Базови класове

За имплементацията на магазин за оръжия се използват няколко класа:

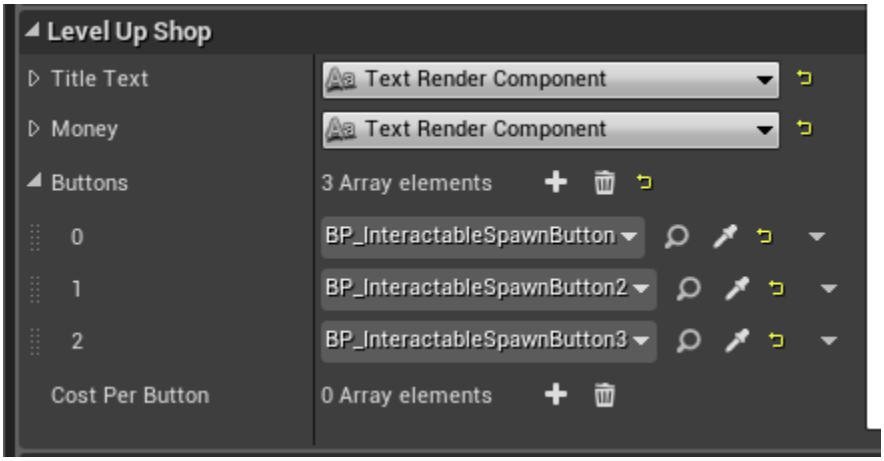
- ALevelUpShop – грижи се за управлението на бутоните и UI

- AInteractableButton – базов клас, който имплементира InteractableComponent, както и комуникацията с ALevelUpShop

- AInteractableSpawnButton – наследява AInteractableButton и имплементира допълнителна логика за менажиране и създаване на инстанции на оръжията.

ALevelUpShop пази в себе си референция към всяка инстанция на AInteractableButton, която менажира. Те се задават ръчно през редактора (Фиг 3.1.4.34). При извикване на метода BeginPlay, на всяка инстанция се извика метода InitializeButton, така AInteractableButton получава референция към делегата, който да извика, когато бъде „купен“ бутона. ALevelUpShop се грижи за статуса на всеки бутон – enum EButtonState *(InteractableButton.h).* Статуса се определя по това дали играчът има достатъчно пари за да закупи артикулът свързан със съответния бутон. Когато бутонът бъде използвам, посредством InteractableComponent (Виж 1.3.1), се извика метода Use, който от своя страна извиква един от делегатите: OnUseUnlockedDelegate, OnUsePurchaseableDelegate, OnUseLockedDelegate. Те са с цел да бъдат имплементирани от наследник (в случая – AInteractableSpawnButton), за да имплементират допълнителна логика. Също така при извикване на метода Use, ако бутонът е в състояние VE_Purchasable, ще бъде извикан и делегата, за да бъде уведомен ALevelUpShop за покупката, след което ще бъде приспадната съответната сума от

ресурсите на играча. AInteractableSpawnButton се грижи съответния обект, свързан с бутона, да бъде инстанциран в света като за целта имплементира и „закача“ функции към предоставените от бащиния клас делегати. Когато бутонът бъде „купен“ или използван при „купено“ (VE_Unlocked) състояние, AInteractableSpawnButton инстанцира нов обект от този тип, като унищожи стария. Добавянето на нов „предмет“ става само чрез създаване на една инстанция на класа AInterctableSpawnButton, закачането ѝ към ALevelUpShop, и нагласянето на параметрите ѝ.



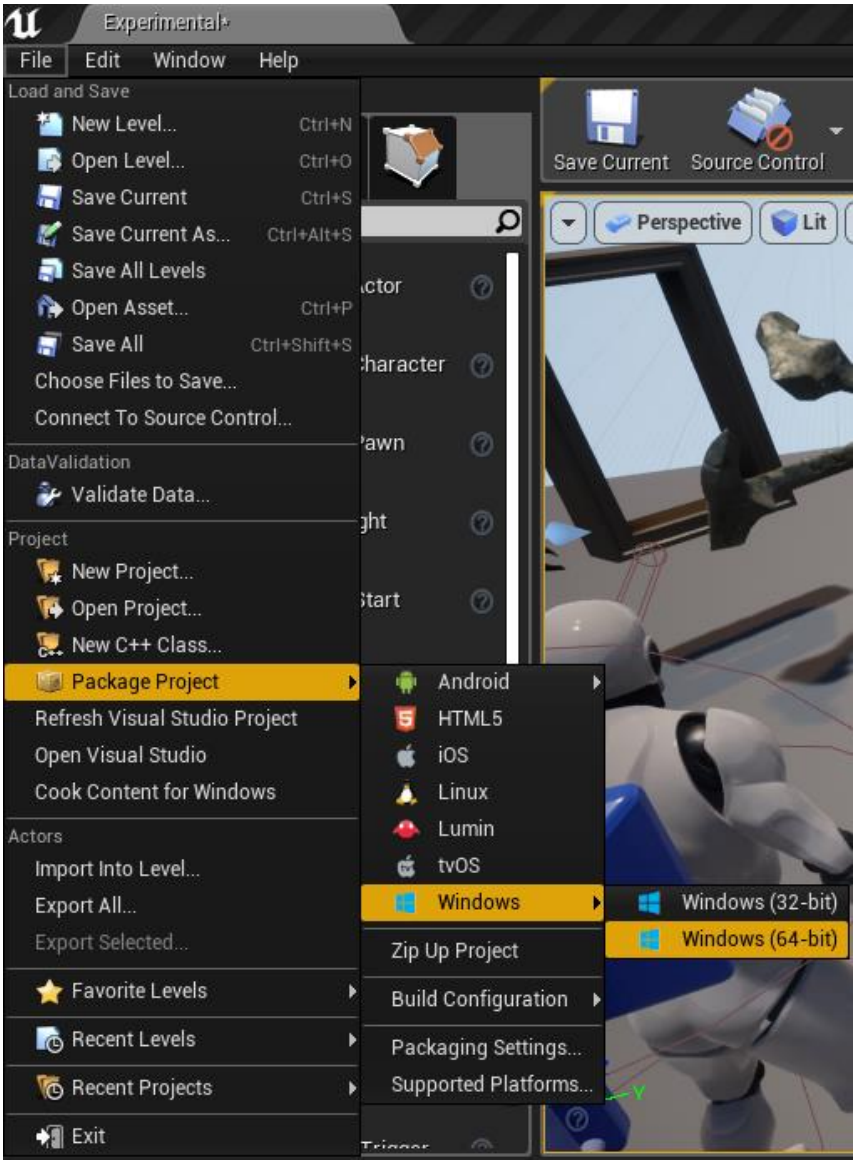
Фиг 3.1.4.33 Ръчно подаване на референции на „бутони“ към управляващия ги ALevelUpShop

ЧЕТВЪРТА ГЛАВА

Наръчник на потребителя

1. Инсталиране

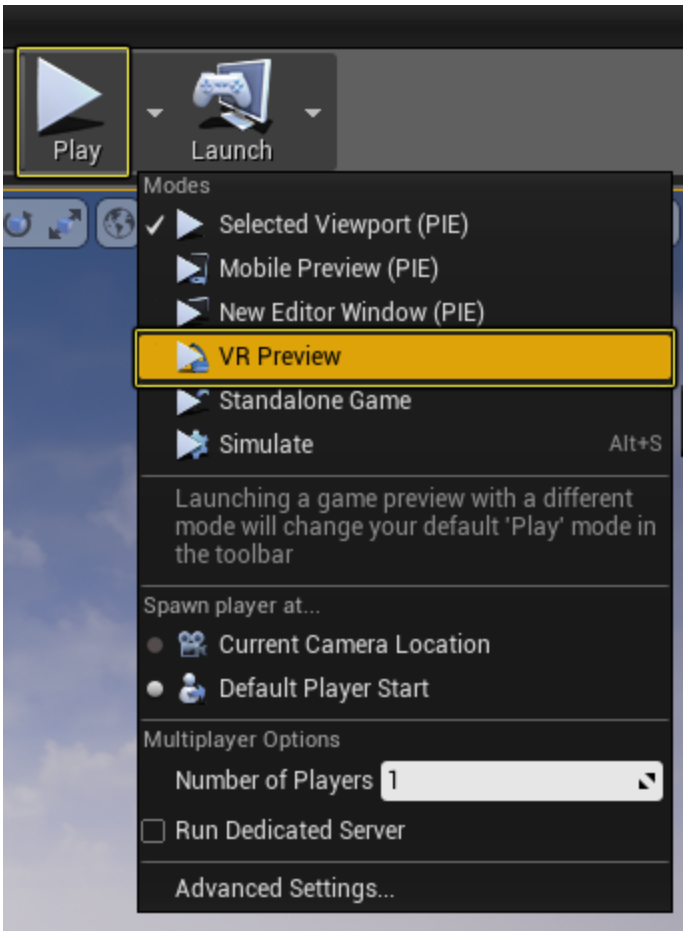
За да се стартира играта е нужен изпълним файл. Този файл се генерира от Unreal Engine(Фиг 4.1), но също така може да бъде копиран вече генериран от друга машина.



Фиг 4.1 Генериране на изпълним файл през редактора

Необходим е и включен и настроен Oculus Rift.

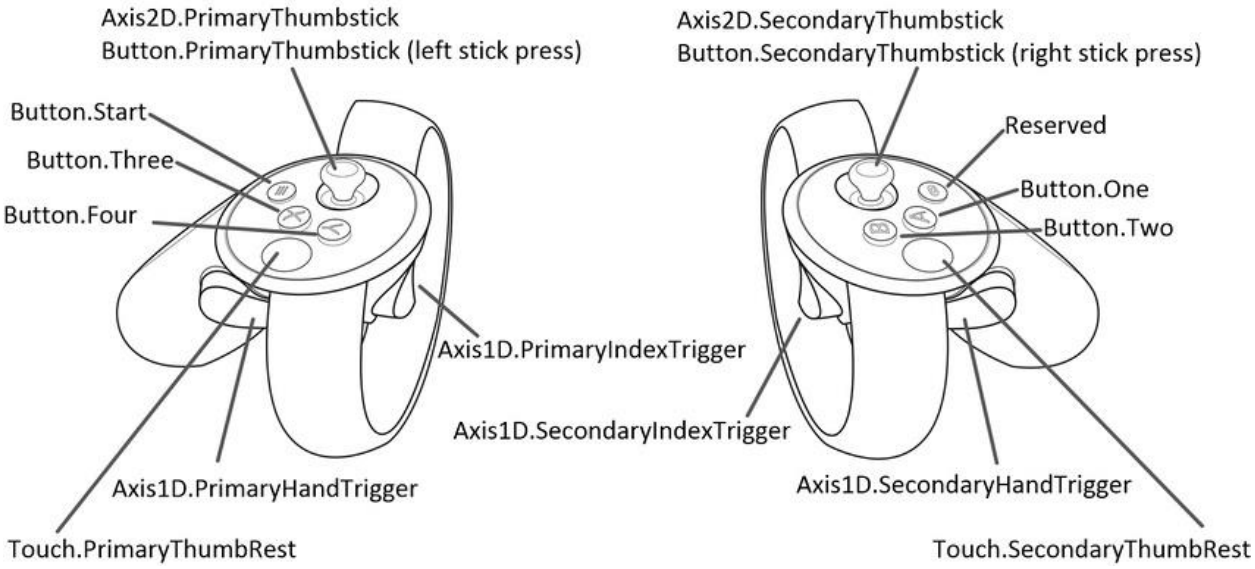
2. Стартиране
    - Стартиране на изпълнимия файл.
    - Отваряне на нивото World, и стартиране играта в редактора (VR Preview) (Фиг 4.2)



Фиг 4.1 VR Preview

3. Използване

    При стартиране в играта играчът бива пуснат на основното ниво след което играта започва. За имена на бутоните виж Фиг. 4.3.



Фиг 4.3 Наименование на копчетата на контролерите

Чрез Primary Thumbstick играчът се движи из сцената. Чрез Secondary Thumbstick се завърта на определен градус в съответната посока. Button One рестартира позицията на камерата. Чрез PrimaryHandTrigger и SecondaryHandTrigger се извикат функциите BeginGrab и съответно при пускане на бутона – EndGrab. Чрез които се взимат обекти в съответната ръка. Използването на обекти става чрез, PrimaryIndexTrigger и SecondaryHandTrigger. Button Two и Button Three изпълняват презареждане, когато се държи оръжие.

Целта на играта е да се защитят бариерите през всички 5 вълни от врагове.

4. Деинсталиране

Изпълнимия файл не пази никакви файлове извън текущата му директория.

ЗАКЛЮЧЕНИЕ

Изпълнение са всички изисквания на дипломната работа чрез максимално придържане към Unreal Engine Coding Standard (1). Играта има функционалност за стрелба и убиване на противници (Виж Глава 3 - 1.4.4.3.2), играе се напълно във Виртуална Реалност посредством Oculus Rift CV1 и Oculus Touch, имплементира четири различни противници (Виж Глава 3 - 1.4.3.4), три различни оръжия (Виж Глава 3 - 1.4.4.3.5) и система за купуване на предмети (Виж Глава 3 - 1.4.5). Играта е написана с цел максимално лесно добавяне на нови оръжия, врагове, предмети в магазина и др.

ИЗПОЛЗВАНА ЛИТЕРАТУРА

Characters:

- Paragon - https://www.unrealengine.com/marketplace/paragon-sevarog
- Morigesh - https://www.unrealengine.com/marketplace/paragon-morigesh
- Grux - https://www.unrealengine.com/marketplace/paragon-grux
- Rampage - https://www.unrealengine.com/marketplace/paragon-rampage

Guns:

- Colt 3D Model - https://www.turbosquid.com/FullPreview/Index.cfm/ID/362695
- Beretta M9 3D Model - https://free3d.com/3d-model/beretta-m9-48306.html
- Laser Sight - https://sketchfab.com/3d-models/laser-sight-model-low-poly-871b3bdcd0f74e9fab3cb054c4efea20
- Reload Cue - http://soundbible.com/414-Eject-Clip-And-Re-Load.html
- Fire Cue - http://soundbible.com/2091-MP5-SMG-9mm.html

Circular Progress Bar for UMG - https://www.tomlooman.com/circular-progress-bar-for-umg/

Wikipedia Quotes - https://www.wikipedia.org/

Unreal Engine Coding Standard (1) - https://docs.unrealengine.com/en-us/Programming/Development/CodingStandard

СЪДЪРЖАНИЕ