ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

Д И П Л О М Н А Р А Б О Т А

Тема: Pacboy – 3D игра, разработена на Unreal Engine

Дипломант:
Ивайло Христов

Научен ръководител:
Виктор Кетипов

С О Ф И Я

2 0 1 5

Увод

Видео игрите са най-разпространените средства за развлечение в модерния свят. През 50-те години на 20-ти век, видео игрите бележат своето начало на конзоли като Nintendo Family, Atari 2600 и Sega Mega Drive, а днес можем да се насладим на преживяванията, които предлагат на модерни конзоли от следващо поколение като *Xbox 360*, *Xbox One*, *PlayStation 1/2/3*, *Nintendo Wii*, както и най-бързо развиваща се платформа за игри - персоналният компютър.

Компютърните игри непрекъснато се развиват, благодарение на бързото развитие на хардуерните компоненти. Техният напредък прави възможно достигането на нови висини в компютърната графика, играното във виртуална реалност[1], както и създаването на така наречения esport[2].

В началото игрите не са се различавали по жанр, но са били иновативни и уникални за своето време. В днешно време съществуват най-различни жанрове игри - симулатори, шутъри, екшън, ужаси, ролеви, стратегии, както и многобройни поджанрове.

Целта на дипломната работа е да бъде създадена *Pacboy* - триизмерна компютърна игра от трето лице от жанр шутър, с бърз игров ход и възможност за игра в мрежа.

[1] http://en.wikipedia.org/wiki/Virtual_reality
[2] http://en.wikipedia.org/wiki/Electronic_sports

Глава 1 Игрови жанрове. Third person shooter. Енджини за разработка на игри

1.1 Игрови жанрове

Игровите жанрове започват своето начало още с появата на игрите. Жанрът на една игра описва нейната същност и какъв ще бъде главния елемент в сюжета ѝ. Чрез жанра играч може да придобие представа за играта преди да я е играл. С развитието на игрите са се развили и видовете жанрове, защото познатите до сега вече не са успявали да опишат дадения продукт достатъчно добре.

Игровите жанрове са много и често се комбинират помежду си – *Amnesia: The Dark Descent* (3D, сървайвал, ужаси). Комбинацията между жанрове служи за по-добро описание на играта и по този начин може да привлече повече играчи, които биха искали да я пробват.

Жанрът е важна част от играта, защото чрез него играта е представена пред света с една дума.

1.2 Third person shooter[1]

1.2.1 Характеристика на жанра

3D тип игри, които през последни години придобиват все повече популярност, особено при конзолите. Комбинира стрелящия елемент от шутърите от първо лице със скачанията и бързите движения от 3D платформърите и имплементира проста система за бой от близки разстояния типични за *Beat 'em up* игри. Шутърите от трето лице почти винаги включват и опция, която да помага при прицелване, тъй като прицелването от камера в трето лице е трудно. Повечето включват и изглед от първо лице, което позволява точна стрелба и възможност за вглеждане в детайли в обстановката, които биха били трудни за откриване чрез нормалния изглед. Игрите от този жанр могат да бъдат както разказвачи на история така и арена шутъри, които наблягат повече на геймплея, от колкото на изживяването.

[1] http://en.wikipedia.org/wiki/Third-person_shooter

1.2.2 Third person shooter (TPS)

- ***Resident Evil 5***

Resident Evil 5, позната в Япония като *Biohazard 5*, е екшън шутър от трето лице разработена и публикувана от *Capcom* през 2009. Играта е създадена за *PlayStation 3*, *Xbox 360* и *Microsoft Windows*. При разработката е използвано улавяне на движение за клипчета в играта, правейки движенията на героя да изглеждат по-реалистични.

Историята на играта се развива около Крис Редфиълд и Шева Аломар и тяхното разследването на терористка заплаха в Киджуджу (измислен регион в Африка). Още в начален стадий в историята на играта Крис разбира, че ще трябва да се изправи срещу своето минало в лицето на свой стар враг Албърт Уескър.

Играчът поема контрол над Крис (или Шева след завършване на играта поне веднъж), а при мрежова игра втория играч поема контрол над Шева. При самостоятелна игра Шева е управлявана от изкуствен интелект в играта.

Фиг 1.1. Играчът контролира Крис от изглед *над рамото*. Кръв и амуниции са разположени в долния десен ъгъл на екрата, изглед на картата в горния десен.

Играчите могат да използват няколко оръжия като пистолети, пушки, автомати, снайпери, гранатомети и също така могат да използват умения за ръкопашен бой срещу противници. Нараняването на противник често ги кара да залитат. Ако това се случи, когато играчът е на близко разстояние от противника се появява опция за специална

ръкопашна атака като ъперкът. Много от клипчетата в играта включват *quick time events*[1], които представляват приканването на играча да извърши някакво действие в със интервал от време за да продължи.

Както в *Resident Evil 4*, играчите не могат да тичат и да стрелят едновременно, но имат опцията да надграждат оръжията си използвайки пари и съкровища намерени по време на играта и да се лекуват с билки. Преди всяка мисия играча трябва да избере лимитирани на брой средства, които може да използва по време на следващата мисия.

Фиг 1.2. Играчите могат да надграждат и разменят предмети.

- ***Gears of War***

Gears of War е военна фантастика от тип шутър от трето лице разработена от *Epic Games* на *Unreal Engine 3* и публикувана от *Microsoft Game Studios* за *Xbox 360* през 2006. Година по-късно е пусната и версия за *Microsoft Windows*, включваща бонус нива и нов multiplayer режим чрез функционалността на *Games for Windows - Live*.

Историята ни съсредоточава върху войската от Отделение Делта, докато те правят последен, отчаян опит да спасят оцелелите човешки обитатели на измислената планета Сера от безмилостните и видимо неудържими подземни врагове познати като Орда Локуст.

[1] http://en.wikipedia.org/wiki/Quick_time_event

Играчът поема контрол над героя Маркъс Феникс (но може да играе и като Доминик Сантяго), бивш затворник и военно закален войник. Втори играч може да се включи по всяко време за да помага при изтребването на чудовищата.

Играта се основава над използването на прикритие и тактическа стрелба за прогресиране по сценария.

Фиг 1.3. Играчът се прикрива от стрелба

В играта има набор от оръжия, но основно се използва така наречения Lancer - автомат, който има прикачена резачка на дулото, която може да се използва при ръкопашен бой. Играчът разполага едновременно с две основни оръжия, гранати и пистолет. Когато на играча е нанесена щета, червен зъб репрезентиращ щетата се показва бледо на екрана, ставайки все по-наситен в зависимост от щетата. Играчът може да намери подслон и да изчака кръвта му да се възстанови, но ако поеме прекалено много щети ще стане недееспособен, имайки нужда от друг играч за да бъде възстановен.

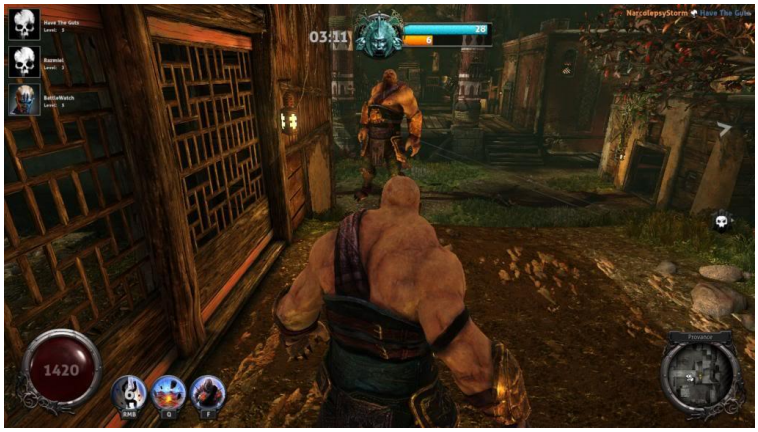
Фиг 1.4. Играчът, екзекутиран от чудовище след понасяне на прекалено много щети

- ***Nosgoth***

Nosgoth е безплатна multiplayer екшън-приключенска игра, разработена от *Psyonix* и публикувана от *Square Enix* за *Microsoft Windows* през 2015.

Тя е spin-off[1] на *Legacy of Kain*[2] игрите и заема място в тяхната едноименна измислена вселена.

Играта използва player versus player[3] система, в която всеки мач се състои от два рунда. Отборите са разделени на две раси: вампири, чийто бойна система е проектирана по модел в hack and slash игри;



Фиг 1.5. Играч от отбор вампири

и хора, чийто стил на игра е шутър от трето лице. След края на рунд, отборите сменят местата си и управляват другата раса и отборът с най-много асимилирани точки накрая на мача печели.

Фиг 1.6. Играч от отбор хора

[1] http://en.wikipedia.org/wiki/Spin-off_(media)
[2] http://en.wikipedia.org/wiki/Legacy_of_Kain
[3] http://en.wikipedia.org/wiki/Player_versus_player

Nosgoth е типичен пример за арена игра. Неангажираща, разделена на мачове, които продължават винаги еднакво (или приблизително еднакво) време и наблягаща повече на геймплей аспекта. Legacy of Kain вселената е огромна и има много история, но в тази игра тя е там просто заради идеята и е оставена на заден план.

1.3 Енджини за разработка на игри[1]

Гейм Енджин представлява софтуерен фреймуърк, чрез който може да бъде създадена игра. Той е разделен на различни модули, всеки, от които извършва специфична задача. Основното, което един енджин може предоставя включва рисуване на 2D и 3D обекти, физика в играта, детектиране на колизии, управляване на звуци, анимации, изкуствен интелект, мрежови опции и скриптиране, представляващо добавяне на допълнително държание и съдържание в играта без да е нужно прекомпилиране на ядрото. Големите фирми за разработка на игри често си правят свой собствен енджин за отделните игри, който да има специфична функционалност за продукта им. По-известните общи гейм енджини са *Unity*, *Unreal Development Kit*, *CryEngine*, *Unreal Engine 4*.

1.3.1 Unity[2]

Гейм енджинът *Unity* предлага множество функционалности и сравнително лесен за научаване интерфейс. Основен плюс е cross-platform интеграцията, което означава, че игри могат да бъдат лесно портнати към *Android*, *iOS*, *Windows Phone 8* и *BlackBerry*, правейки го чудесен енджин за разработка на мобилни игри. Освен това има възможност за разработка на игри за *PlayStation 3*, *Xbox 360*, *Wii U*, *Microsoft Windows* и *Web browsers*.

Фиг 1.7. Мобилна игра разработена на Unity

[1] http://en.wikipedia.org/wiki/Game_engine
[2] http://en.wikipedia.org/wiki/Unity_%28game_engine%29

Въпреки че енджина предоставя възможност за вкарване на асети направени от почти всяко приложение за моделиране, страда от липса на редактиращи опции вътре в самия редактор. *Unity* няма истинска функционалност за моделиране или строене освен няколко примитивни фигури, но пък се хвали с огромна библиотека от асети, които могат да бъдат изтеглени безплатно или закупени.

1.3.2 Unreal Development Kit (UDK)[1]

UDK е безплатната версия на *Unreal Engine 3*, разработена от Epic Games, която дава достъп до този мощен гейм енджин, който и днес се използва за създаването на AAA[2] игри, като *Gears of War*. Може да се използва за разработка на мобилни игри, както и за стандартните конзоли и настолни компютри. За разлика от *Unity*, има мощни инструменти за модифициране на нива директно в гейм енджина.

Фиг 1.8. Разлика в графичните възможности м/у Unreal Engine 1, 2 и 3

Unreal Engine отначало е бил разработен за създаване на шутъри от първо лице (специфично за *Unreal Tournament* игрите), но се използва и за много други жанрове като ролеви игри. *Unreal Engine 3* използва скриптовия език UnrealScript, който е обектно-ориентиран език подобен на Java и C++.

[1] http://en.wikipedia.org/wiki/Unreal_Engine#Unreal_Development_Kit
[2] http://en.wikipedia.org/wiki/AAA_(game_industry)

1.3.3 CryEngine[1]

CryEngine е изключително мощен енджин разработен от фирмата *Crytek*, представен за първи път при играта *Far Cry*. Създаден е да се използва при разработката на игри за настолен компютър и конзоли, включващи *PlayStation 4* и *Xbox One*.

Фиг 1.9. Реалистична обстановка в игра разработена на CryEngine

Графичните възможности на *CryEngine* надвишават тези на *Unity* и *UDK*, но са на нивото с тези на *Unreal Engine 4*. С реалистичните си светлини, физики, системи за анимация и много други енджинът предлага всичко нужно за създаването на AAA игра. Подобно на *UDK*, *CryEngine* има интуитивна и силна функционалност за редактиране на нива в самия гейм енджин.

1.3.4 Unreal Engine 4 (UE4)[2]

Unreal Engine 4 е най-новият енджин разработен от *Epic Games* и наследник на *UDK*. *UE4* има изумителни графични способности като динамични светлини, нова система за частици, която поддържа до 1 милион частици в сцена едновременно. *UE4* използва *DirectX 10-12* (при разработка на игри за *Microsoft Windows*, *Xbox One*, *Windows RT*), *OpenGL* (при разработка на игри за *OS X*, *Linux*, *PlayStation 4*, *iOS*, *Android*, *Ouya*, *Windows XP*), и JavaScript/*WebGL* (при разработка на игри за *HTML5 Web browsers*).

[1] http://en.wikipedia.org/wiki/CryEngine
[2] http://en.wikipedia.org/wiki/Unreal_Engine#Unreal_Engine_4

Фиг 1.10. Реалистична сцена в Unreal Engine 4

UE4 загърбва UnrealScript и го заменя с C++, а visual scripting системата в *UE3* (*Kismet*), е заменена с интуитивните блупринти. Минусът на енджина е, че не могат да се разработват игри за конзоли от старо поколение, което означава, че поддържа игри само за настолни компютри, *PlayStation 4*, *Xbox One* и мобилни устройства.

Глава 2 Изисквания към играта. Избор на средства за разработка. Основна структура на UE4 и играта.

2.1 Изисквания към играта

- Да се създаде триизмерна шутър от трето лице игра, която да може да се играе в Multiplayer
- Да има меню, чрез което играчите да могат да се свържат в мрежа
- Да има различни оръжия, с които играчите да стрелят
- Геймплеят да бъде бърз – играча да може да спринтира, отскача от стени
- Възможност играчите да се убиват и съживяват
- Нормална мрежова среда на игра – анимациите, проектилите, движенията и подобни да бъдат репликирани и виждани от всички в реално време

2.2 История

След като Пакман умря единственото, което спираше духовете от превземането на света, беше Пакбой. За съжаление духовете ставаха все по-силни и след като Пакбой осъзна, че няма да може да се справи сам, откраднa технологията им за съживяване, жертвайки живота си.

След този инцидент е създадена военната програма Pacboy. Всеки, който има желание може да се включи в благородната мисия да спре духовете от превземането на света.

В момента програмата Pacboy е в стадии на подготовка. Използвайки технологията, за която Пакбой се жертва, участници в програмата минават през симулации на реални битки без да се притесняват за загубата на живота си.

Включи се и ти докато програмата е още в подготвителен стадии и помогни да спасим света!

2.3 Избор на средства за разработка

За разработка на играта е използван Unreal Engine 4.

В Unreal Engine 4, скриптирането се случва чрез C++ или чрез visual scripting системата им (Blueprints). Придобитите знания по C++ в 11. клас улесниха значително писането на проекта и в крайна сметка използването на друг език за програмиране би довело до известни затруднения и отделяне на време за научаването и упражнението му. Блупринтите дават възможност за бързо създаване на прототип, на който могат да бъдат нанасяни промени веднага. Бонус е и блупринт дебъгера, до който има достъп директно през едитора.

Основна причина за избора на UE4, е че Epic Games, създателите на енджина, имат огромен опит в правенето на гейм енджини и факта, че предната версия на енджина се използва и днес, показва, че правят енджини, които да оцеляват дълго. Научаването на UE4, вместо UE3 е по-смислено тъй като UE4 като нова версия ще се наложи в бъдеще и познанията ми по него биха помогнали за бъдещи проекти.

Като бонус UE4 предоставя Marketplace секция, с множество безплатни асети, които биха могли да се използват при създаването на прототип или цяла игра.

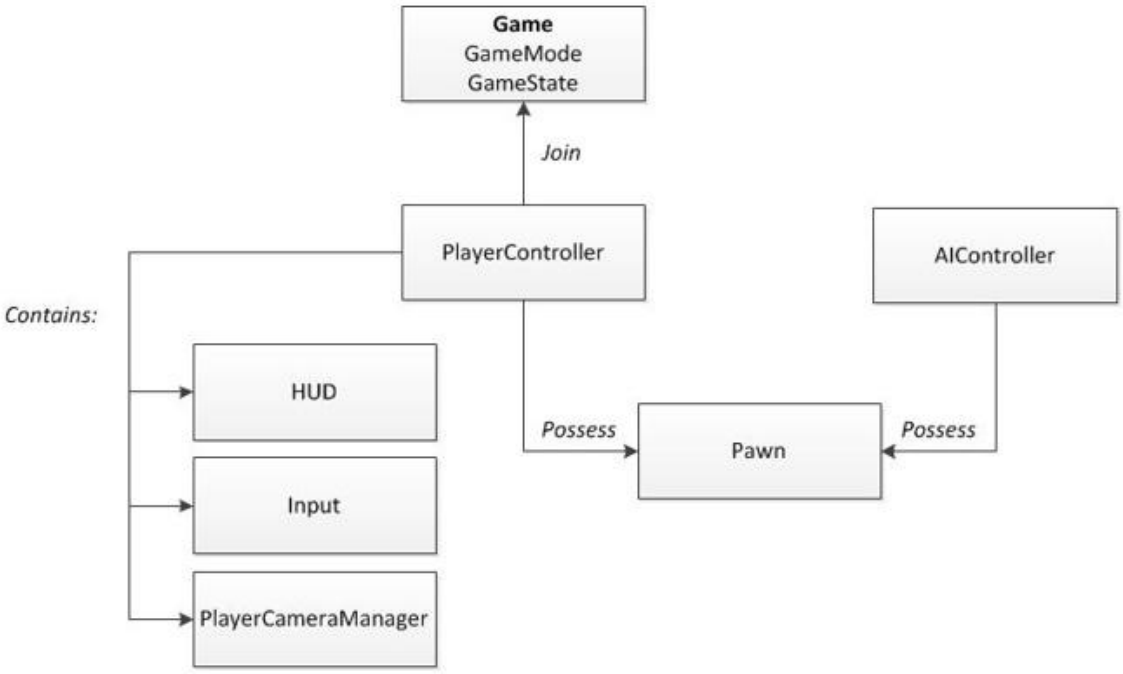
IDE за програмиране на UE4, под Windows операционна система по подразбиране е Microsoft Visual Studio. Visual Studio предлага чудесни инструменти за рефакториране поддържащи преименуване, извличане на методи и интерфейси както и изключително добър дебъгер. Като бонус Visual Studio има безплатна версия за ученици и студенти.

2.4 Основни класове в UE4 фреймуорка

UE4 фреймуорка предоставя набор от класове, които създават основна структура и функционалност на играта. Чрез наследяването и надграждането на тези класове, програмистите добавят нуждата им, допълнителна функционалност за съответната игра.

Тези класове "крият" детайлите от ниско ниво на енджина и предоставят функционалности, които хора с 20+ години опит са разработвали. Чрез тях програмистите могат да се съсредоточат над това, което е наистина важно: създаването

на играта и не трябва да се занимават с тривиални неща като вход от потребителя, основни движения и други.



Фиг 2.1. Основни класове предоставени от UE4 и тяхното стандартно взаимодействие

2.4.1 GameFramework/Actor

Обекти от типа Actor представляват основната единица в играта. Обикновено съдържат поне един ActorComponent в себе си. Имат удобни опции за репликиране в мрежова среда.

Не съдържат в себе си информация относно позиция и ротация (root компонента на обекта пази тази информация). Създават се в света чрез SpawnActor() метода. Трабва да бъдат унищожавани изрично чрез Destroy() метода, тъй като garbage collector-а няма да ги изчисти автоматично.

2.4.2 Components/ActorComponent

Обекти от типа ActorComponent съдържат преизползваема функционалност на някой Actor. Тази функционалност може да бъде засичане на колизия, управление на движението, изпълняване на звук и много други. Всъщност всичко, с което играчът взаимодейства или вижда е от тип ActorComponent. До ActorComponent имаме достъп и през блупринти.

ActorComponent, който има информация за местоположение, ротация и размер се нарича SceneComponent, а такъв който може да бъде нарисуван PrimitiveComponent. Други подтипове на ActorComponent са AudioComponent, ArrowComponent, InputComponent, LightComponent, MeshComponent, ParticleSystemComponent и много други.

2.4.3 Components/PrimitiveComponent

С обект от тип PrimitiveComponent можеш да се сблъскаш, при което се извикват различни евенти като:

- Hit - при блъскането в стена
- Begin/EndOverlap - при влизането/излизането от някакво пространство
- Begin/EndCursorOver - при наличието на мишка на екрана, когато тя е преместена през Actor
- Clicked - при наличието на мишка на екрана, когато Actor е кликнат

2.4.4 GameFramework/Pawn

Pawn обектите представляват базовия клас на всички Actors в света, които могат да бъдат контролирани от играч или изкуствен интелект. Pawn обикновено е физическата репрезентация на играч или изкуствен интелект. Това не означава само, че Pawn определя как изглежда Actor-а визуално, но и как взаимодейства със света и по специфично как се сблъсква с други обекти. В някои игри той може да няма видим визуален изглед вътре в игровия свят, но въпреки това Pawn представлява неговото физическо местоположение, ротация и други.

2.4.5 GameFramework/Controller

Обектите от тип Controller са физически несъществуващи единици, които могат да овладеят обекти от тип Pawn, за да контролират неговите действия. Овладяват Pawn чрез метода Possess() и освобождават Pawn чрез метода Unpossess().

По подразбиране Controller и Pawn имат "един към един" връзка, което означава, че един Controller контролира само един Pawn в даден момент. Pawn създаден по време на игра не е автоматично овладян от Controller.

Controller обектите могат да съществуват и след като техния Pawn е унищожен. В зависимост дали управляващия е човек или изкуствен интелект контролера бива PlayerController и AIController.

2.4.6 GameFramework/PlayerController

PlayerController представлява интерфейс за играчите към Pawn. Грижи се за докосвания, кликове и евенти от клавиатурата. Чрез него може да се показва и скрива мишката. Използва се в менюта, чат и други.

2.4.7 GameFramework/Character

Character класът надгражда Pawn с CharacterMovementComponent (грижещ се за движението на героя), CapsuleComponent (грижещ се за засичането на сблъсъци) и SkeletalMesh (визуалната репрезентация на героя). Character обектите са играчи, които могат до ходят, тичат, скачат, летят и плуват по света.

2.4.8 GameFramework//HUD

HUD (съкратено от Head-Up Display) е класът отговорен за потребителския интерфейс в играта.

2.4.9 GameFramework/GameMode

Класът GameMode имплементира правилата на режима на игра. Чрез него се задават класовете по подразбиране за Pawn, Controller, HUD и подобни. Текущия GameMode обект може да бъде достъпен навсякъде чрез GetGameMode().

GameMode обектите съществуват само на сървъра, тъй като той има права да определя какво се случва в играта. Клиентите в играта получават информация за състоянието на играта чрез друг клас на име GameState. Може да съществува само един GameMode в даден момент.

2.5 Основен алгоритъм на играта.

При стартиране на играта пред играчът се показва меню с бутони Host, Join и Quit и червено поле за въвеждане на IP Address.

При избиране на опция Host играчът преминава в Lobby екран и започва да "слуша" за други връзки, което означава, че друг играч може да се свърже към тази мрежа. В Lobby екрана има два бутона Start и Quit. При натискане на бутона Start играчът заедно с всички свързани играчи се преместват в предварително зададена карта за игра.

При въвеждането на адрес в червеното поле и натискането на бутона Join, играчът прави опит да се свърже към този адрес и при успех преминава към екран Lobby, където има само бутон Quit. Играчът трябва да изчака Host-а да натисне Start, при което е преместен към предварително зададена карта.

При започването на играта за всеки играч се създава Контролер, който приема неговите команди от клавиатура и мишка и ги изпраща към други класове, които функционират с тях. Контролера е постоянен и не се променя до края на играта или загуба на връзка с играча. След Контролер се създават герой, който играча управлява и потребителски интерфейс.

Режима на игра представлява Free for All, което означава, че всеки ти е противник и играеш само за себе си. Чрез потребителски интерфейс играча вижда своята кръв и енергия, както и текущия си резултат. При умиране играчът се съживява след кратък период.

Глава 3 Реализация на играта

3.1 Описване на начина на реализация

За реализация на играта е нужно следното:

- Създаване или взимане на готови модели и анимации, които да бъдат ползвани в играта
- Писане на скриптове в комбинация от C++ и блупринти за добавяне на съдържание на играта

В играта са използвани безплатни модели и анимации от *Unreal Engine 4 Marketplace*. Моделите идват със скелет и заедно с анимациите са готови за ползване.

3.2 Основна информация при програмиране със C++ на UE4

При дефиниране на променливи и функции UE4 ни предоставя възможност да използваме така наречените макроси, които ни позваляват да въведем допълнително поведение на полетата чрез спецификатори. Записват се над дефиницията на полето и биват:

- UPROPERTY() – за променливи
  - VisibleDefaultsOnly – достъпни са само началните стойности на променливата само за четене
  - VisibleAnywhere – променливата е видима нявсякъде, но не може да бъде редактирана
  - EditDefaultsOnly – може да се зададе друга начална стойност на променливата
  - EditAnywhere – променливата може да бъде променяна навсякъде
  - BlueprintReadOnly – променливата е достъпна само за четене в блупринти
  - BlueprintReadWrite – променливата е достъпна за промяна в блупринти
  - Replicated – променливата е обозначена за репликиране – когато сървъра ѝ промени стойността, тя се променя и локално при клиентите
  - Category – определя категорията на променливата видима в блупринта

- UFUNCTION() – за функции
    - BlueprintCallable – функцията е достъпна за извикване от блупринти
    - BlueprintImplementableEvent – функцията може да бъде имплементирана в блупринти
    - Client – функцията ще бъде изпълнена локално на клиент ако е извикана от сървър
    - Server – функцията ще опита да бъде изпълнена на сървър ако е извикана от клиент
    - WithValidation – обозначава, че ще се дефинира допълнителна функция, която да разрешава/забранява изпълнението на ф-я на сървър
    - NetMulticast – функцията се изпълнява на всички клиенти ако е извикана от сървър (изпълнява се и локално на сървъра)
    - Reliable – обозначава, че функцията зависима от сървъра ще бъде изпълнена със сигурност в даден момент дори и при мрежови проблеми
    - Unreliable – обозначава, че при натоварен мрежов трафик функцията може да не се изпълни
    - Category – определя категорията на функцията видима в блупринти

3.3 Класове в приложението

3.3.1 Интерфейсът IDamageableObject

Всеки клас имплементиращ този интерфейс показва, че може да му бъде нанесена щета.

```
IDamageableObject.h
public:

	virtual void TakeDamage(float Damage, const FHitResult& Hit, AController*
EventInstigator) = 0;
```

3.3.2 Класът MainPlayerController

Този клас ще бъде основния Controller в играта след меню екраните. Целта му е да запази информация относно броя убийства и умирания на играча. Тази информция се записва в контролера, а не в Pawn класа, тъй като той може да бъде унищожен, а контролерът е постоянен до загуба на връзка.

```
MainPlayerController.h
public:
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Replicated, Category = "Gameplay")
	int32 Kills;
```

```
UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Replicated, Category = "Gameplay")
	int32 Deaths;
```

- Kills – броят убийства над други играчи
- Deaths – брой умирания

```
MainPlayerController.cpp
void AMainPlayerController::GetLifetimeReplicatedProps(TArray< FLifetimeProperty > &
OutLifetimeProps) const
{
	Super::GetLifetimeReplicatedProps(OutLifetimeProps);

	DOREPLIFETIME(AMainPlayerController, Kills);
	DOREPLIFETIME(AMainPlayerController, Deaths);
}
```

Във функцията GetLifetimeReplicatedProps определяме кои са променливите, които искаме да бъдат репликирани по време на изпълнение.

3.3.3 Класът ProjectileBase

Този клас ще представлява основата на проектил – може да бъде изстрелян от оръжие и да нанася щети при сблъскване с други играчи.

```
ProjectileBase.h
public:

	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "Projectile")
	float Damage;

	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Projectile")
	USphereComponent* CollisionComponent;

	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Movement")
	UProjectileMovementComponent* ProjectileMovement;

	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Projectile")
	AController* Shooter;

	UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = "Projectile")
	UStaticMeshComponent* ProjectileMesh;

	AProjectileBase(const FObjectInitializer& ObjectInitializer);

	UFUNCTION(BlueprintImplementableEvent, Category = "Projectile")
	void OnImpact(AActor* OtherActor, UPrimitiveComponent* OtherComp);

protected:

	UFUNCTION()
	virtual void OnHit(AActor* OtherActor, UPrimitiveComponent* OtherComp, FVector
```

```
NormalImpulse, const FHitResult& Hit);
```

- Damage – щетите, които проектилът нанася при сблъсък.
- CollisionComponent – компонент, отговарящ за колизии.
- ProjectileMovement – компонент, отговарящ за движението на проектила.
- Shooter – контролерът на играча, изстрелял този проектил.
- ProjectileMesh – визуалната репрезентация на проектила.
- OnImpact() – функция, която се извиква вътрешно при сблъсък. Имплементира се в блупринти. Служи за добавяне на ефект при сблъсък.
- OnHit() – функция, която се извиква при сблъсък.

```
ProjectileBase.cpp
AProjectileBase::AProjectileBase(const FObjectInitializer& OI)
	: Super(OI)
{
	CollisionComponent = OI.CreateDefaultSubobject<USphereComponent>(
		this,
		FName(TEXT("CollisionComponent"))
	);
	CollisionComponent->InitSphereRadius(2.5f);
	CollisionComponent->OnComponentHit.AddDynamic(this, &AProjectileBase::OnHit);

	RootComponent = CollisionComponent;

	ProjectileMovement = OI.CreateDefaultSubobject<UProjectileMovementComponent>(
		this,
		FName(TEXT("ProjectileMovementComponent"))
	);

	ProjectileMesh = OI.CreateDefaultSubobject<UStaticMeshComponent>(
		this,
		FName(TEXT("ProjectileMesh"))
	);
	ProjectileMesh->AttachTo(RootComponent);

	InitialLifeSpan = 10.f;
}
```

В конструктора на класа създаваме обекти за CollisionComponent, ProjectileMovement и ProjectileMesh. Инициализираме CollisionComponent като сфера с радиус 2,5, след което задаваме при засичане на сблъсък да извиква функцията OnHit() дефинирана в класа ни. Определяме CollisionComponent като root на обекта и прикрепяме меша към него, и

задаваме живота на проектила да бъде 10 секунди, което означава, че ако проектилът не се сблъска с нищо в рамките на 10 секунди ще извика метода Destroy(). Оставяме задаването на стойности на проектила като щети, скорост и избирането на меш на проектила за блупринт наследници на този клас.

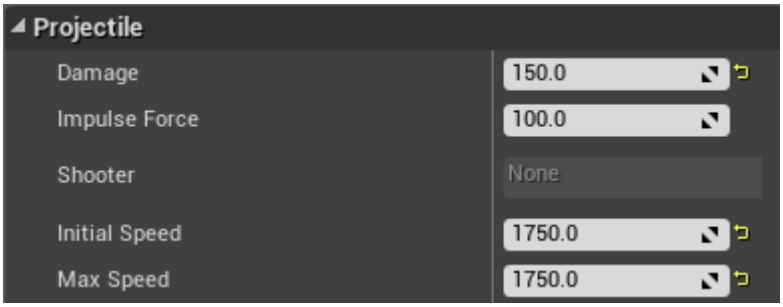
```
ProjectileBase.cpp
void AProjectileBase::OnHit(AActor* OtherActor, UPrimitiveComponent* OtherComp, FVector
NormalImpulse, const FHitResult& Hit)
{
	AProjectileBase* OtherProj = Cast<AProjectileBase>(OtherActor);
	if (OtherProj != NULL)
	{
		return;
	}

	IDamageableObject* DamageableObject = Cast<IDamageableObject>(OtherActor);
	if (DamageableObject != NULL)
	{
		APawn* Pawn = Cast<APawn>(OtherActor);
		if (Pawn != NULL)
		{
			if (Pawn->GetController() == Shooter)
			{
				return;
			}
		}

		DamageableObject->TakeDamage(Damage, Hit, Shooter);
	}

	OnImpact(OtherActor, OtherComp);
	Destroy();
}
```

При сблъсък проверяваме другия обект дали също е от тип ProjectileBase и ако е, го игнорираме (не искаме проектил да експлодира при сблъсък с друг проектил). След което проверяваме дали обектът е имплементирал интерфейса IDamageableObject и ако го имплементира проверяваме дали е Pawn (дали е играч). Ако е Pawn проверяваме дали играчът не се е сблъскъл със собствения си проектил и ако е така го игнорираме. Ако е друг играч извикваме функцията TakeDamage, която му нанася щети. Ако всичко е

минало извикваме OnImpact() за да се изпълнят ефектите при сблъсък имплементирани в блупринти и Destroy() за унищожаване на проектила.

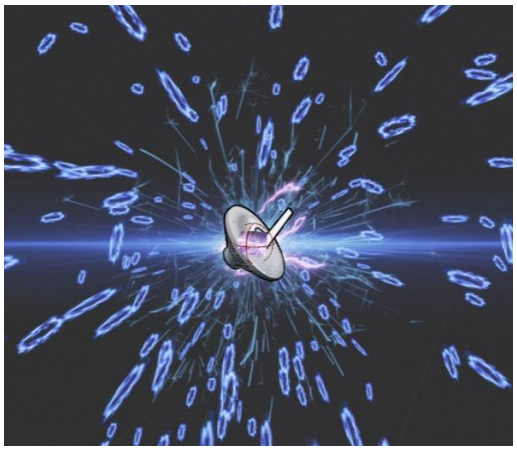
- **BP_RocketLauncher_Projectile**

Този блупринт клас наследява ProjectileBase и имплементира поведение на проектил на гранатомет.



Фиг 3.1. Задаване на стойности на обектите от класа BP_RocketLauncher_Projectile

Освен това е избран меш на проектила и са добавени допълнителни компоненти – AudioComponent за звук и ParticleSystemComponent, представляващ система от частици като и двата компонента следват проектила.



Фиг 3.2. Симулация на проектила с новите компоненти

В евент графата на блупринта се имплементира поведение на OnImpact() функцията. При извикването и взима ротацията на проектила и завърта Pitch, което представлява ротацията на елемент по Y оста “напред-назад”, с 180°, за да може да обърне ефекта наобратно срещу стената и създава емитер на текущата позиция на проектила с променената ротация придружен от звук на експлозия. След което се извиква функцията ApplyRadialDamage() с изходна точка текущата позиция на проектила и радиус 375, която ще опита да нанесе щети на всички Actor обекти, които се намират в радиуса на сферата.

3.3.4 Класът Weapon

Класът Weapon е основата на далекобойните оръжия в играта. Чрез него може да създадем различни оръжия бързо чрез блупринти.

```
Weapon.h
UENUM(BlueprintType)
namespace EWeaponType
{
	enum Type
	{
		Melee,
		Pistol,
		Rifle,
		Sniper,
		RocketLauncher
	};
}

UENUM(BlueprintType)
namespace EWeaponShootingType
{
	enum Type
	{
		Instant,
		Projectile
	};
}
```

Енумерациите EWeaponType и EWeaponShootingType представляват съответно типа оръжие и типа на стрелба на оръжието. Различните типове оръжия имат различна функционалност. Имплементирани са чрез блупринти. От типа на стрелба на оръжието се разбира дали оръжието изстрелва проектил или нанася моментални щети.

```
Weapon.h
public:

	UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "Weapon")
	TEnumAsByte<EWeaponType::Type> WeaponType;

	UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Weapon")
	TEnumAsByte<EWeaponShootingType::Type> ShootingType;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Weapon")
	float Damage;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Ammo")
```

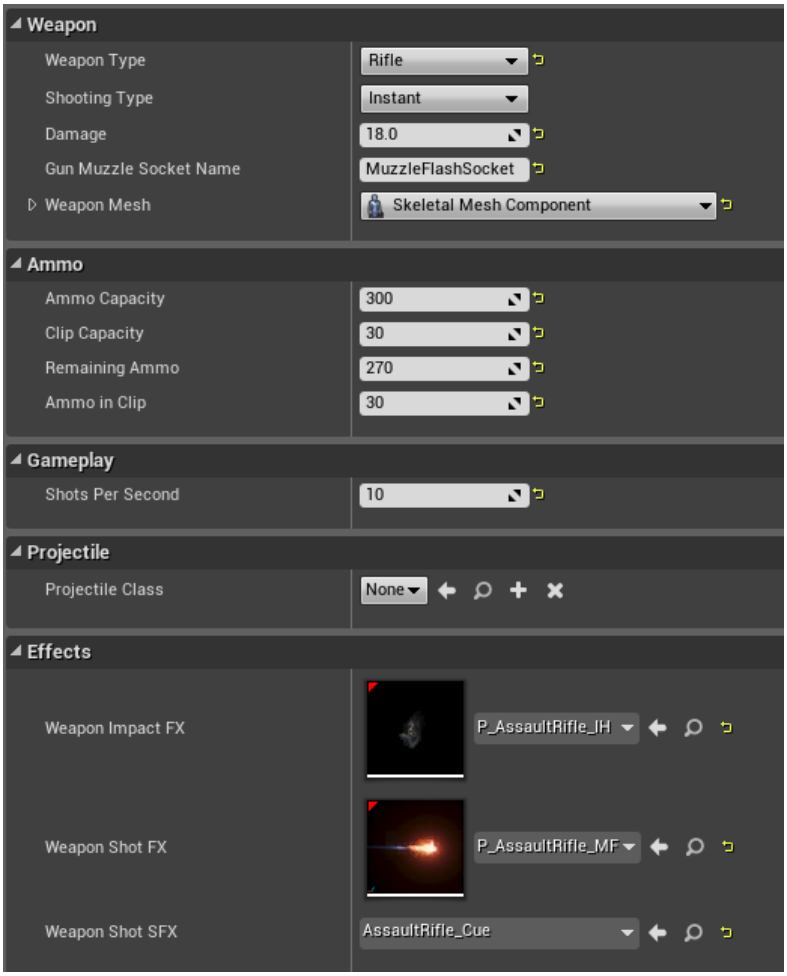
```
int32 AmmoCapacity;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Ammo")
int32 ClipCapacity;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Ammo")
int32 RemainingAmmo;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Ammo")
int32 AmmoInClip;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Weapon")
FName GunMuzzleSocketName;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Gameplay")
int32 ShotsPerSecond;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Projectile")
TSubclassOf<AProjectileBase> ProjectileClass;

UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category = "Weapon")
USkeletalMeshComponent* WeaponMesh;

UPROPERTY(EditDefaultsOnly, Category = "Effects")
UParticleSystem* WeaponImpactFX;

UPROPERTY(EditDefaultsOnly, Category = "Effects")
UParticleSystem* WeaponShotFX;

UPROPERTY(EditDefaultsOnly, Category = "Effects")
USoundBase* WeaponShotSFX;

AWeapon(const FObjectInitializer& ObjectInitializer);

void Init(const AWeapon& Weapon);

UFUNCTION(BlueprintCallable, Category = "Weapon")
void Reload();
```

- Damage – щетите, които оръжието нанася при удар (използва се, когато оръжието е с Instant тип стрелба)
- AmmoCapacity – общ брой патрони, които оръжието може да носи
- ClipCapacity – общ брой патрони, които оръжието може да има в пълнителя
- RemainingAmmo – оставащи патрони
- AmmoInClip – оставащи патрони в пълнителя
- GunMuzzleSocketName – името на гнездото на дулото на оръжието (използва се за начална позиция на изстрелите и за ефекти при стрелба)
- ShotsPerSecond – брой изстрели на секунда при автоматична стрелба (задържане на бутона за стрелба)

- ProjectileClass – класът на проектила, който оръжието изстрелва (използва се, когато оръжието е с Projectile тип стрелба)
- WeaponMesh – визуалната репрезентация на оръжието
- WeaponImpactFX – ефектът при сблъсък на изстрел (използва се, когато оръжието е с Instant тип стрелба)
- WeaponShotFX – ефектът при изстрелване
- WeaponShotSFX – звуковият ефект при изстрелване
- Init() – инициализира променливите на класа като копира стойностите им от друго вече създадено оръжие
- Reload() – презарежда оръжието

➢ BP_Rifle

BP_Rifle наследява Weapon класа и имплементира поведение на автомат.

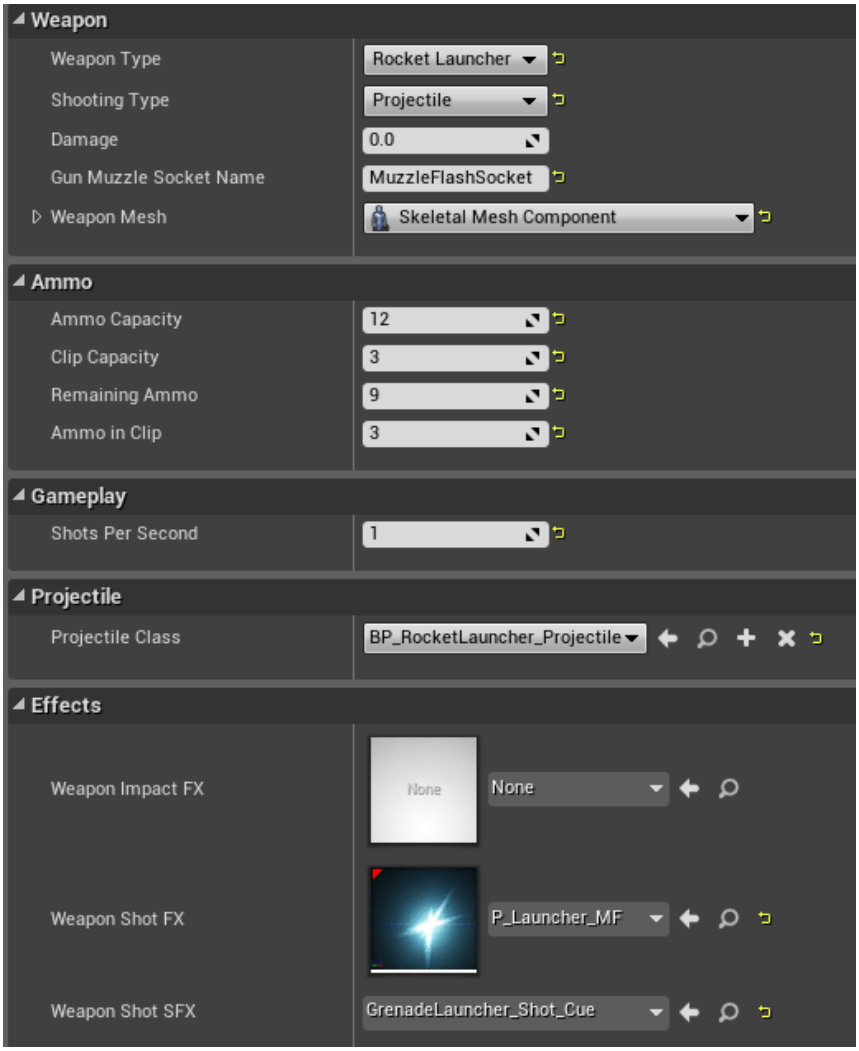


Фиг 3.3. Зададени стойности на класа BP_Rifle

Класът има Instant тип стрелба, което означава, че трябва да му бъдат зададени стойности на щети и ефект при сблъсък на изстрел. За ProjectileClass не избираме нищо, тъй като оръжието не изстрелва проектили.

- **BP_RocketLauncher**

Подобно на BP_Rifle, BP_RocketLauncher наследява Weapon класа, но имплементира поведение на гранатомет.



Фиг 3.4. Зададени стойности на класа BP_RocketLauncher

Класът има Projectile тип стрелба, което означава, че трябва да му бъде зададен клас на проектил, който да изстрелва. Damage и WeaponImpactFX остават със стойности по подразбиране тъй като класът на проектила отговаря за тях.

3.3.5 Класът CharacterBase

CharacterBase наследява Character и е класът, който седи в основата на героя в играта. Той се грижи за това как героят да се движи, стреля, скача, както и да репликира всичко това на свързаните клиенти, като изпълнява извиканите функции на сървърна част или от сървъра ги извиква на всички клиенти. Класът имплементира интерфейса IDamageableObject, което означава че може да поема щети. В класа е дефинирана и структурата DetectWallResult, която съдържа информация за докоснати стени при опит за отскок. Най-важните променливи на класа са:

- JogSpeed – скоростта, с която героятсе движи в нормално състояние
- SprintSpeed – скоростта, с която геротя се движи докато спринтира
- AimSpeed – скоростта, с която героят се движи докато се прицелва
- Health – текущата кръв на героя
- HealthCapacity – максималната кръв на героя
- Energy – текущата енергия на героя (използва се за спринтиране и отскачане от стени)
- EnergyCapacity – максималната енергия на героя
- bIsSprinting – индикира дали героят спринтира
- bIsAiming – индикира дали героят се прицелва
- bIsFiring – индикира дали героят стреля
- bIsDead – индикира дали героят е мъртъв
- SprintEnergy – енергията нужна за спринтиране
- WallJumpEnergy – енергията нужна за отскок от стена
- Fallen – индикира дали героят е умрял от падане извън картата
- Rifle – автомата на героя
- RocketLauncher – гранатомета на героя
- EquippedWeapon – оръжието, което героя използва в момента

В TakeDamage() функцията нанясаме подадената щета на героя и пускаме ефект за удар. След което имаме проверка дали кръвта на героя е свършила и ако е така индикираме,

че е мъртъв. Добавяме едно умиране към контролера на този клас и едно убийство ако съществува реален убиец (когато героят е паднал от картата няма убиец). Задаваме два таймера, които да извикат съответно функцията Respawn_Player след 5 секунди, която да откачи контролера от тялото на героя, да създаде ново тяло и да закачи контролера за него и DespawnBody след 15 секунди, която извиква функцията Destroy() на всички компоненти в героя за да може да изчезне от света. Override-ваме функцията ReceiveAnyDamage(), която се извиква от наследения Pawn клас при поемане на друг вид щета (като ApplyRadialDamage() от проектила на гранатомета), да извиква имплементацията на TakeDamage().

Jump() функцията се извиква при всеки опит за скок. Първо има проверка дали героят е в състояние на падане и ако не е така (героят е на земята) извикваме Jump() функцията на класа родител, която имплементира обикновен скок. Ако героят вече е във въздуха извикваме DetectWall(), която връща променлива от тип DetectWallResult с информация за стени около героя. Ако сме в близост до стена проверяваме дали имаме нужната енергия за отскок и ако е така проверяваме дали сме ударили стена в дясната, лявата или предната/задната част на героя. В зависимост от коя страна сме имали сблъсък извикваме наследения метод LaunchCharacter за да изстреляме героя в определена посока. За дясна и лява страна използваме ускорение чрез GetActorRightVector() на героя. Аналогично за предна и задна страна използваме GetActorForwardVector(). За ускорение нагоре по Z оста използваме GetActorUpVector().

```
CharacterBase.cpp
ACharacterBase::DetectWallResult ACharacterBase::DetectWall()
{
	DetectWallResult Result{ false, false, false };

	const FVector TraceStart = this->GetActorLocation();
	const FCollisionQueryParams QueryParams = FCollisionQueryParams(
		TraceTag,
		true,
		this
	);
	const FCollisionObjectQueryParams OQP = FCollisionObjectQueryParams(
		ECC_TO_BITFIELD(ECC_WorldStatic) | ECC_TO_BITFIELD(ECC_WorldDynamic)
	);

	FHitResult SphereHitResult();
	const FVector SphereTraceEnd = (this->GetActorUpVector() * 100) + TraceStart;

	bool WallFound = this->GetWorld()->SweepSingle(SphereHitResult, TraceStart,
```

```
SphereTraceEnd, FQuat(), FCollisionShape::MakeSphere(60.f), QueryParams, OQP);

	if (!WallFound)
	{
		return Result;
	}
	Result.HitWall = true;

	// right side
	FHitResult RightSideHitResult();
	const FVector RightSideTraceEnd = (this->GetActorRightVector() * 100) +
TraceStart;

	bool RightSideHit = this->GetWorld()->LineTraceSingle(RightSideHitResult,
TraceStart, RightSideTraceEnd, QueryParams, OQP);

	if (RightSideHit)
	{
		Result.HitSide = true;
		Result.RightSideHit = true;

		return Result;
	}

	// left side
	FHitResult LeftSideHitResult();
	const FVector LeftSideTraceEnd = (this->GetActorRightVector() * -100) +
TraceStart;

	bool LeftSideHit = this->GetWorld()->LineTraceSingle(LeftSideHitResult,
TraceStart, LeftSideTraceEnd, QueryParams, OQP);

	if (LeftSideHit)
	{
		Result.HitSide = true;
		Result.RightSideHit = false;

		return Result;
	}

	return Result;
}
```

DetectWall() се извиква при опит за скок, когато героят вече е във въздуха. Функцията проверява за близки стени и връща резултата. Начална точка на лъчите, които пускаме да проверят за други обекти е текущата позиция на героя. Като допълнителни настройки на лъчите отбелязваме да игнорират този актьор и да търсят сблъсък само с обекти от тип WorldStatic и WorldDynamic. Използваме функцията SweepSingle, която пуска лъчи във сфера около дадена позиция с цел да разберем дали изобщо има стена в близост до героя ни с радиус 60. Ако открием стена правим две нови проверки с единичен лъч за стена вдясно и вляво от героя. Накрая връщаме получените резултати в структура от тип DetectWallResult. Записаната информация в структурата е:

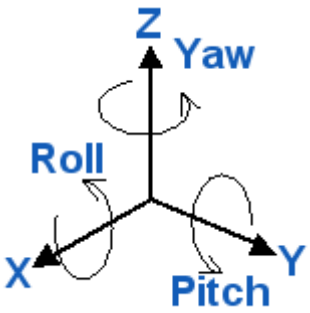
- HitWall – дали е докосната стена
- HitSide – дали е докосната стена в дясната или лявата част
- RightSideHit – дали е докосната стена в дясната част

Чрез тях във функцията Jump() можем да изстреляме героя в правилна посока.

В BeginPlay() инициализираме променливите Health и Energy с максималните им стойности зададени в HealthCapacity и EnergyCapacity, задаваме скоростта на играча да е стойността запазена в JogSpeed и създаваме оръжията Rifle и RocketLauncher на героя, чиито класове определяме в блупринт наследник на героя. Задаваме и таймер, който да извиква функцията UpdateEnergy(), която добавя или взима енергия в зависимост дали героя спринтира, 10 пъти на всяка секунда.

```
CharacterBase.cpp
void ACharacterBase::Move(float AxisValue, EAxis::Type Axis)
{
	if (this->bIsDead)
	{
		return;
	}

	if (this->GetController() != NULL && AxisValue != 0.f)
	{
		// Find out which way is forward
		const FRotator Rotation = this->GetController()->GetControlRotation();
		const FRotator YawRotation = FRotator(0.f, Rotation.Yaw, 0.f);

		// Get forward or right vector
		const FVector Direction = FRotationMatrix(YawRotation).GetUnitAxis(Axis);

		this->AddMovementInput(Direction, AxisValue);
	}
}
```

Във функцията Move() добавяме движение на героя по оста, която му е подадена. X оста за движение напред и назад и Y оста за движение надясно и наляво. Проверяваме дали имаме скорост на движение и ако е така търсим посоката на героя ни чрез ротацията на контролера. Интересуваме се само от Yaw завъртането (Фиг 3.5.), тъй като то представлява завъртане “наляво-надясно” и чрез него определяме посоката и добавяме движение по подадената ос.



Фиг 3.5. Наименования на ротациите по различните оси

FireStart() е методът, който се извиква при опит за стрелба. В начаоло има проверка дали това е първия изстрел и дали трябва да се изчака преди стреляне. Тази проверка е нужна, за да могат оръжията да функционират правилно при автоматична стрелба (задържане на бутона за стреляне). След нея има проверка и за наличие на патрони и при успешно преминаване се индикира, че героят стреля с променливата bIsFiring, извиква се OnFire() метода, който се имплементира в наследници и представлява реалното стреляне и се включва таймер, който извиква OnFire() на всеки 1/ShotsPerSecond секунди.

3.3.6 Класът MainCharacter

MainCharacter наследява CharacterBase и добавя камери, които да следват героя и да се грижат за изгледа при прицелване или в близост до стена. Кои методи да се извикват при подадена информация от контролера се задават в този клас във функцията SetupPlayerInputComponent(), а самите бутони се записват в DefaultInput.ini в /Config папката на проекта.

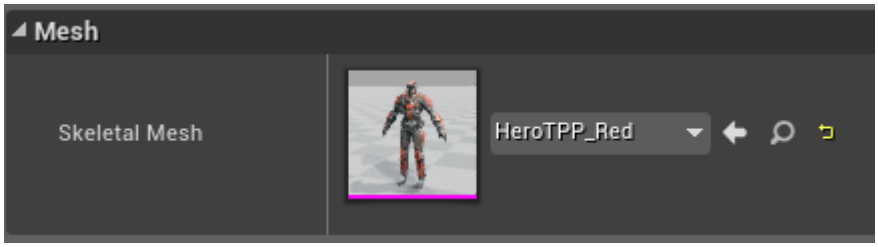
```
MainCharacter.cpp
void AMainCharacter::SetupPlayerInputComponent(class UInputComponent* InputComponent)
{
	Super::SetupPlayerInputComponent(InputComponent);

	if (InputComponent != NULL)
	{
		InputComponent->BindAxis("MoveForward", this,
&AMainCharacter::MoveForward);
		InputComponent->BindAxis("MoveRight", this, &AMainCharacter::MoveRight);

		InputComponent->BindAxis("Turn", this, &AMainCharacter::Turn);
		InputComponent->BindAxis("LookUp", this, &AMainCharacter::LookUp);

		InputComponent->BindAction("Sprint", IE_Pressed, this,
&AMainCharacter::SprintStart);
		InputComponent->BindAction("Sprint", IE_Released, this,
&AMainCharacter::SprintStop);
```

```
		InputComponent->BindAction("Aim", IE_Pressed, this,
&AMainCharacter::AimStart);
		InputComponent->BindAction("Aim", IE_Released, this,
&AMainCharacter::AimStop);

		InputComponent->BindAction("Fire", IE_Pressed, this,
&AMainCharacter::FireStart_Key);
		InputComponent->BindAction("Fire", IE_Released, this,
&AMainCharacter::FireStop);

		InputComponent->BindAction("Reload", IE_Pressed, this,
&AMainCharacter::ReloadStart);

		InputComponent->BindAction("SwapToRifle", IE_Pressed, this,
&AMainCharacter::SwapToRifle);
		InputComponent->BindAction("SwapToRocketLauncher", IE_Pressed, this,
&AMainCharacter::SwapToRocketLauncher);

		InputComponent->BindAction("Jump", IE_Pressed, this,
&AMainCharacter::Jump);
	}
}
```

В този клас се имплементира и OnFire() метода наследен от CharacterBase. В него след проверки нужни за функционалността на автоматичната стрелба взимаме позицията на дулото на текущото оръжие (EquippedWeapon) и в зависимост дали оръжието има стрелба от тип Instant или Projectile съответно пускаме лъч или създаваме проектил. Ако стрелбата на оръжието е от тип Instant проверяваме дали лъчът се е сблъскъл с друг герой и ако е така нанасяме съответната щета. И в двата случая след изстрел намаляваме броя на патроните в пълнителя.

- **BP_MainCharacter**

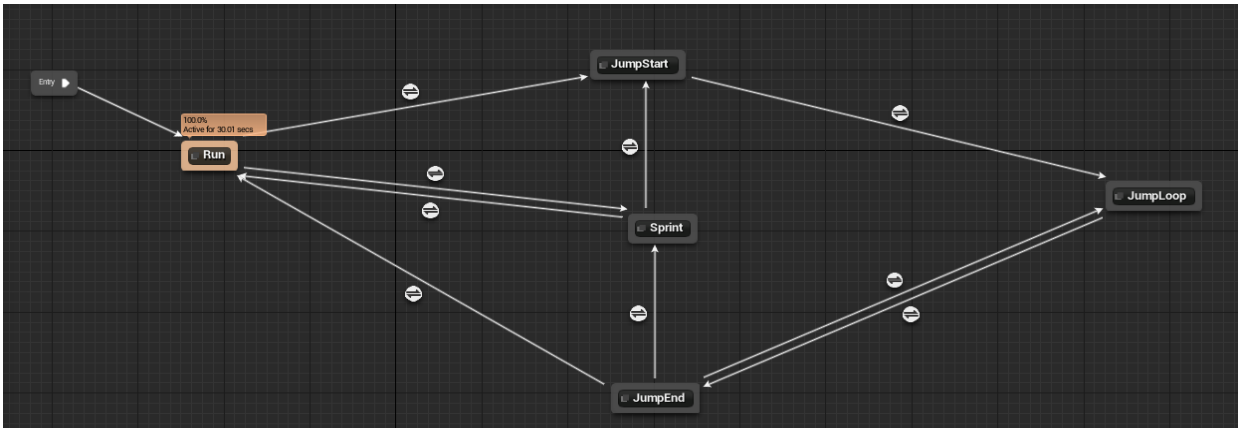
Този блупринт клас наследява CharacterBase и задава стойности на променливите и компонентите в класа. В евент графата на класа се инициализира потребителски интерфейс, добавя се в изгледа на играча и на героя се задава меш оцветен в друг цвят локално за да се различава от противниците.



Фиг 3.6. Задаване на меш на героя

- **HeroTPP_AnimBlueprint**

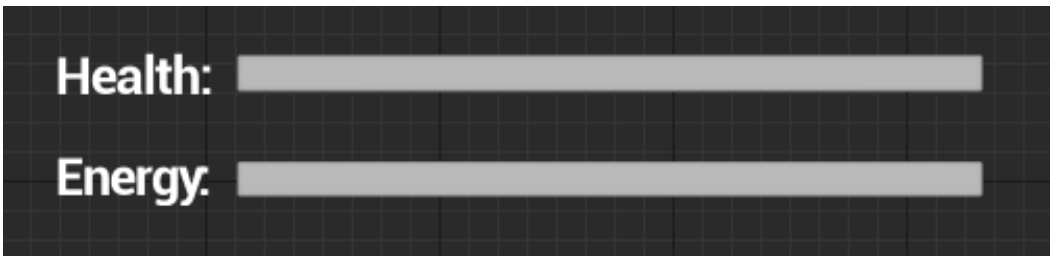
В този блупринт клас и имплементирана цялата логика за анимациите. В евент графата при всеки Update, което представлява всеки кадър се задават стойности взети от героят, на когото принадлежи тази анимация, чрез булевите променливи за състоянието на героя като bIsFiring, bIsSprinting, bIsReloading и други. С тези стойности работи State Machine-а на анимациите. Чрез него героят има различни състояния (с различни анимации към тях) и изпълнявайки дадени условия, проверявани чрез зададените стойности в Update() функцията, може да премине от едно състояние в друго. Този модел ни улеснява като премахва нуждата от допълнителни проверки от кое състояние може да се премине в друго.



Фиг 3.7. State Machine-а на анимациите на героя

- **HUD_GameMode**

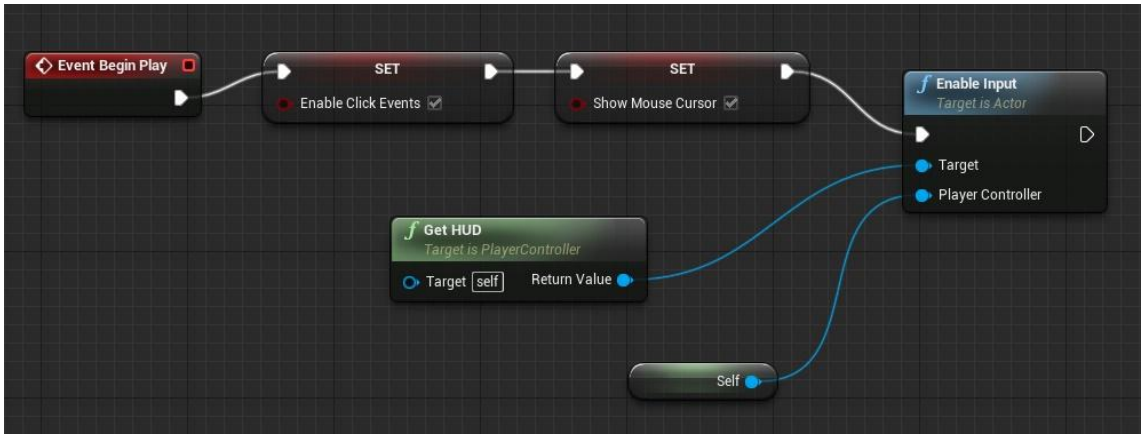
HUD_GameMode представлява блупринт класа, в който е имплементиран потребителския интерфейс по време на игра. Той е минималистичен – представлява само един мерник, който се вижда само при прицелване и два бара, които репрезентират кръвта и енергията. Логиката на самото рисуване се случва в евент графата на класа.



Фиг 3.8. Барове репрезентиращи кръвта и енергията на героя

3.3.7 Blueprint класът PC_MainMenu

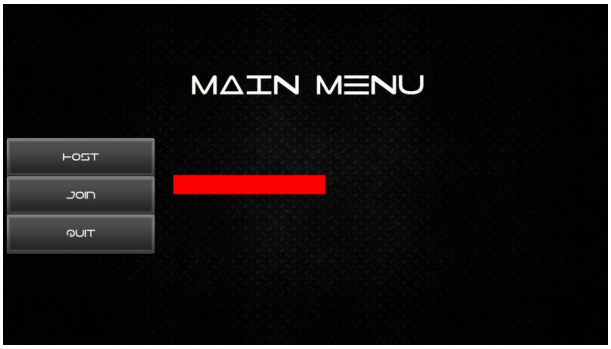
Този клас представлява контролера използван преди започване на игра – в главното меню и в стаята за изчакване. Позволява само евенти от мишката и показва курсора на екрана.



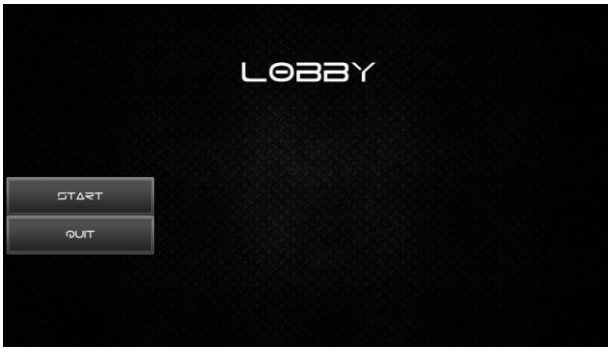
Фиг 3.9. Евент графата на PC_MainMenu

3.3.8 Blueprint класът HUD_MainMenu

В този клас се случва самото рисуване на потребителския интерфейс в менюто. В евент графата в зависимост дали играча се намира в главното меню или в стаята за изчакване се рисуват различни бутони.



Фиг 3.10. Изглед на главното меню



Фиг 3.11. Изглед на стаята за изчакване

Глава 4 Ръководство за потребителя

4.1 Изискване към хардуера

Минималните изисквания към компютърната система са:

- Дисково пространство – поне 1 GB
- Процесор – 1.5 GHz или по-добър
- Оперативна памет – 512 MB
- Налични клавиатура и мишка

4.2 Изискване към софтуера

Като софтуерна платформа изискванията са Windows 7 или по-нов. Като допълнение може да е необходимо инсталирането на DirectX.

4.3 Инсталиране

За Windows операционна система инсталирането се състои в разархивиране на архива с файлове. Всички нужни библиотеки са заедно с пакета.

4.4 Стартиране на играта

Играта се стартира от изпълнимия файл Pacboy.exe.

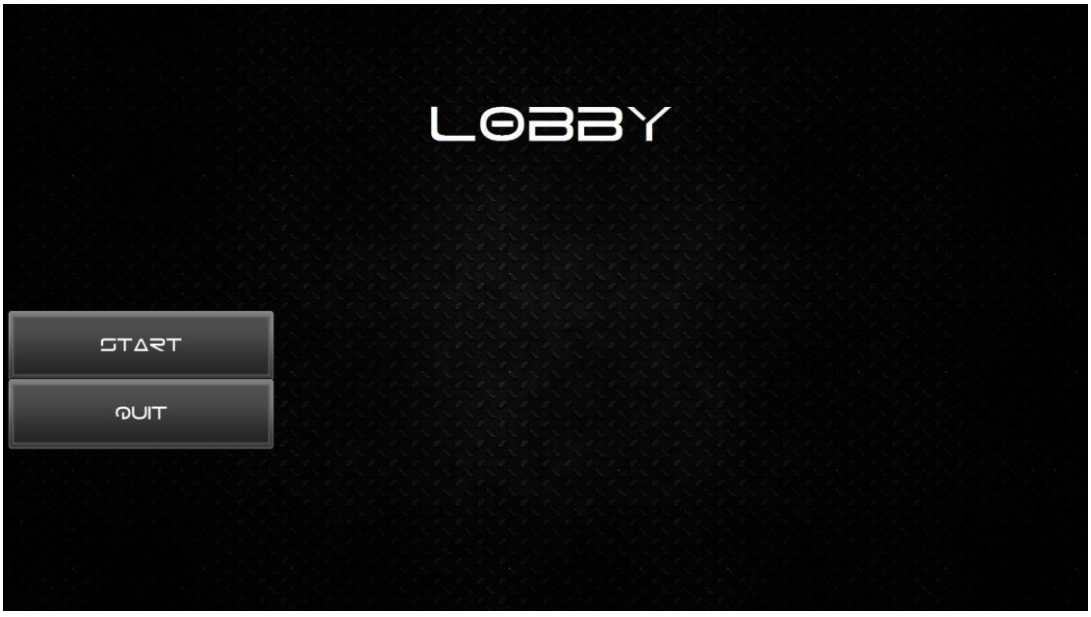
Фиг 4.1. Главното меню на играта

В главното меню потребителят може да избере една от три опции:

- Host – играчът създава стая, в която други играчи могат да се свържат чрез неговия IP адрес. За успешно свързване играчът Host трябва да подсигури порта 7777 като отворен. При създаване на стая играчът преминава в екрана Lobby (стая за изчакване) и остава в нея докато не напусне или на натисне новопоявилият се бутон Start. Всички играчи свързани в стаята в момента преминават в играта заедно с играча Host при стартиране.

- Join – играчът трябва да въведе IP адреса, към който иска да се свърже в червеното поле вдясно от бутоните, и да натисне бутона Join. При успешна връзка играчът ще бъде преместен в стаята за изчакване заедно с потребителя Host и трябва да изчака да бъда стартирана играта

- Quit – изход от играта



Фиг 4.2. Стаята за изчакване на Host играч

4.5 Управление

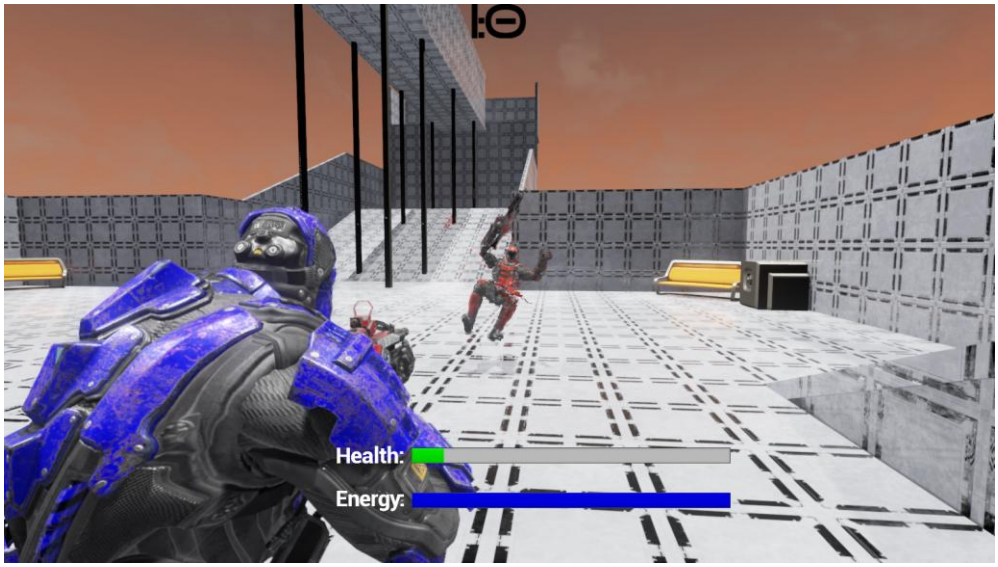
Контролите за управление са:

- W – Напред
- S – Назад
- A – Наляво

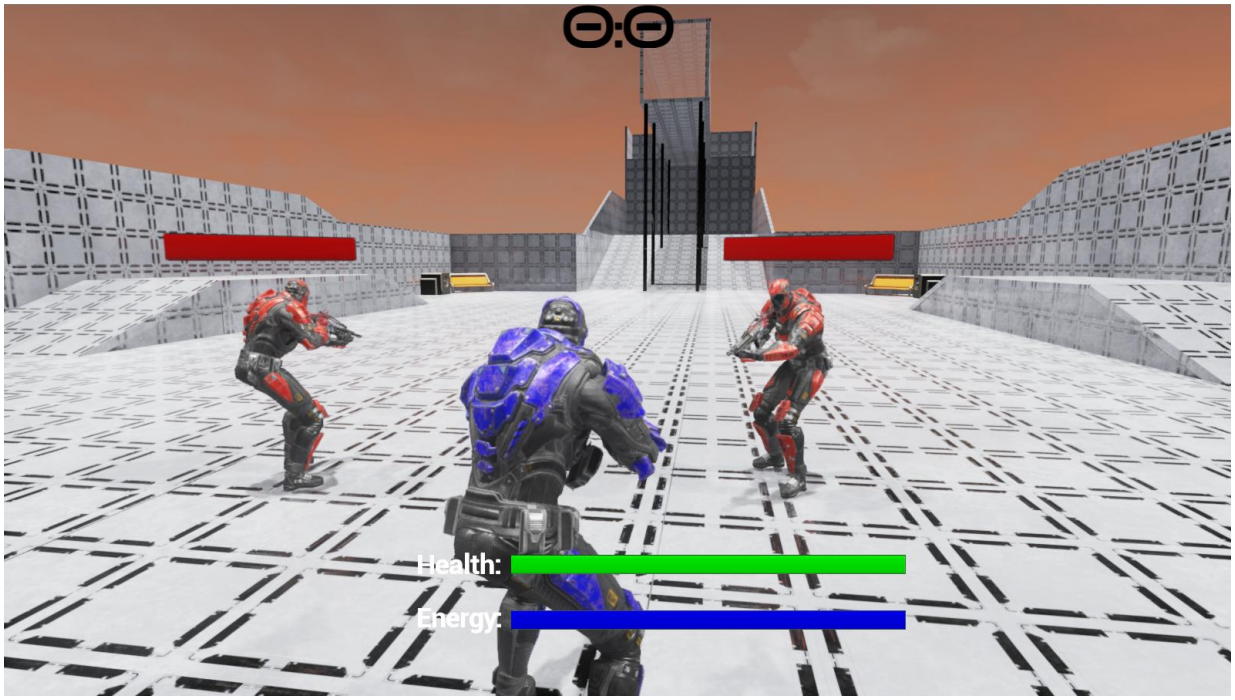
- D – Надясно
- Q – Отскок вляво
- E – Отскок вдясно
- 3 – Превключване към автомат
- 5 – Превключване към гранатомет
- Мишка – Въртене на камерата
- Десен бутон на мишка – Прицелване
- Ляв бутон на мишка – Стрелба

4.6 Screenshots

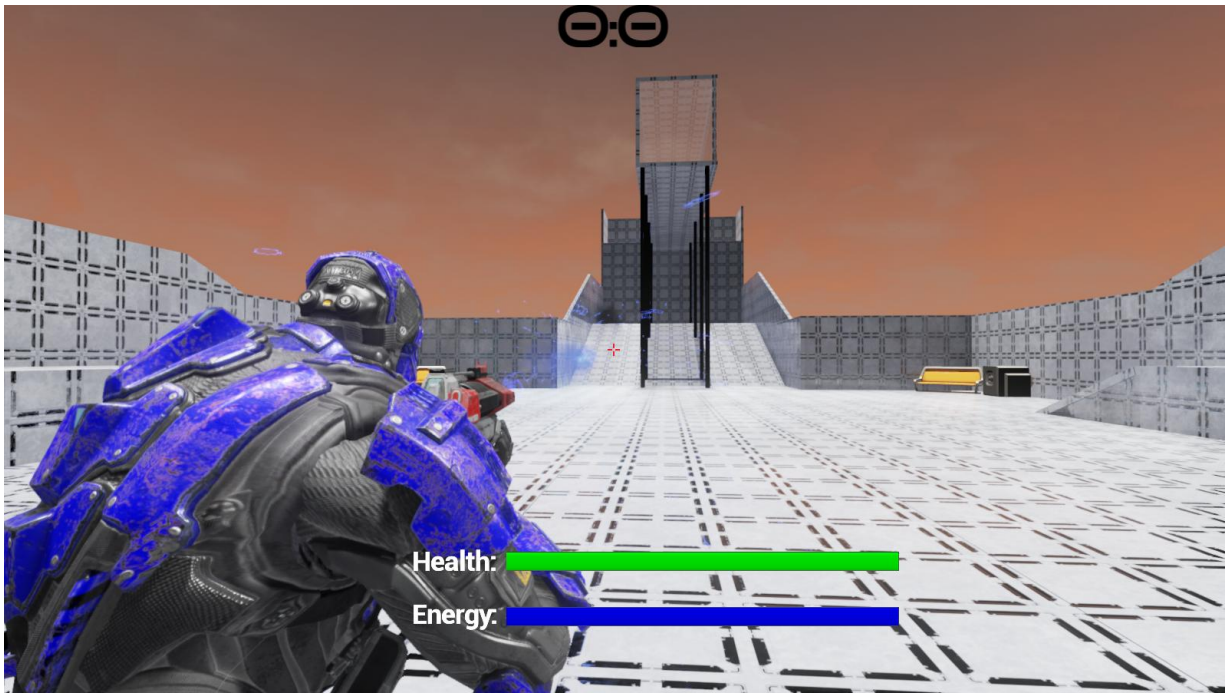
Фиг 4.3. Противникът е уцелен и кърви



Фиг 4.4. Противникът умирайки пада назад



Фиг 4.5. Плаващите барове, показващи кръвта на опонентите винаги се завъртат спрямо играча



Фиг 4.6. Изстрелян проектил от тип BP_RocketLauncher_Projectile

4.7 Hints

- Проектилът на гранатомета е бавен, но за сметка на това при директен сблъсък убива с 1 удар

- Проектилът на гранатомета създава експлозия, която може да ви помогне в “напечени” ситуации, но може и да нанесе щети на вас
- Чрез отскок вляво и вдясно можете да избегнете проектил, който е близко до вас
- Докато сте във въздуха от отскачане от стени е по-трудно да ви оцелят
- Внимавайте да не паднете извън картата, защото умирате

Заключение

С развитието на технологиите се развиват и интерактивните забавления. Тази еволюция на видео игрите ще направи играенето на игри много по-естествено преживяване. Същевременно игрите ще станат по-голяма част от реалния свят. Някои игри в много държави вече са приети за официален спорт. В днешно време вече се развиват игри, контролирани от движения.

Успешно бе създадена триизмерна шутър игра от трето лице, която може да достави часове забавление с приятели или случайни хора онлайн. С бързия си режим на игра, ефекти и звуци, играта поддържа интереса на играча.

Играта има развити основни механики, които предоставят възможността да се играе възможно най-бързо. Могат да се добавят нови карти, нови оръжия, умения на героите, меню с опции, екран за зареждане и нови режими на игра наблягащи на други аспекти като тактика и отборна игра. Може да се добави и single-player режим, за да може играта да предоставя забавно изживяване и без нужда от интернет връзка или други хора.

Използвана литература и ресурси

- Bjarne Stroustrup. The C++ Programming Language (4th edition)
- Robert Nystrom. Game Programming Patterns
- Mike McShaffry, David Graham. Game Coding Complete, Fourth Edition
- http://gameprogrammingpatterns.com/
- https://docs.unrealengine.com/latest/INT/
- https://docs.unrealengine.com/latest/INT/Videos/index.html

Съдържание