МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ЕЛЕКТРОНІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

«До захисту допущено»
В.о. завідувача кафедри

_____Світлана ВАЩЕНКО

____ _____ 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття освітнього ступеня бакалавра

зі спеціальності 122 «Комп'ютерні науки»
_____,

освітньо-професійної програми «Інформаційні технології проектування»

на тему: Ігровий додаток-шутер «Shoot Them Up»
_____

Здобувача групи ____ІТ-91____Сніжка Артема Ярославовича_____
(шифр групи) (прізвище, ім'я, по батькові)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____ ____Артем СНІЖКО____
(підпис) (Ім'я та ПРІЗВИЩЕ здобувача)

Керівник кандидат технічних наук, доцент Наталія ФЕДОТОВА_____
(посада, науковий ступінь, вчене звання, Ім'я та ПРІЗВИЩЕ) (підпис)

Суми-2023

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра інформаційних технологій
Спеціальність 122 «Комп'ютерні науки»
Освітньо-професійна програма «Інформаційні технології проектування»

ЗАТВЕРДЖУЮ
В. о. зав. кафедри ІТ

_____Світлана ВАЩЕНКО
«_____» _____2023 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА СТУДЕНТУ

Сніжко Артему Ярославовичу

1 Тема роботи *ігровий додаток-шутер «Shoot Them Up»,*

керівник роботи *Федотова Наталія Анатоліївна, к.т.н., доцент***,**

затверджені наказом по університету від «*29*» 05 2023 р. №0588-VI

2 Строк подання студентом роботи «7» червня 2023 р.

3 Вхідні дані до роботи правила ігрового процесу, системні вимоги

4 Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) вступ, аналіз предметної області, порівняння аналогів, вибір засобів реалізації, постановка задачі, моделювання програмної реалізації, моделювання варіантів використання, проектування процесного алгоритму, програмна реалізація додатку, тестування та працездатність, висновки, список літературних джерел, додатки.

5 Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) _____

6. Консультанти розділів роботи:

Розділ	Консультант	Підпис, дата	
		Завдання видав	**Завдання прийняв**
Аналіз предметної області	*Федотова Н. А.*		
Постановка задачі	*Федотова Н. А.*		
Моделювання та проектування	*Федотова Н. А.*		

7.Дата видачі завдання_____8 лютого 2023_____

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Оформлення планування робіт	До 19.02.2023	
2	Оформлення технічного завдання	До 28.02.2023	
3	Аналіз предметної області	До 10.03.2023	
4	Проектування та моделювання	До 30.03.2023	
5	Розробка ігрового додатку	До 15.05.2023	
6	Тестування додатку	До 21.05.2023	
7	Оформлення пояснювальної записки	До 06.06.2023	

Студент _____ Артем СНІЖКО
(підпис)

Керівник роботи _____ к.т.н., доц. Наталія ФЕДОТОВА
(підпис)

РЕФЕРАТ

Тема роботи «Розробка ігрового дотатку «Shoot Them Up» »

Пояснювальна записка містить вступ, розділ «Аналіз предметної області», розділ «Моделювання та проектування», розділ «Розробка ігрового додатку», висновки, список літературних джерел та додатки. Вона включає 143 сторінки, 43 рисунка, 21 літературне джерело та 11 таблиць.

У рамках даної роботи проведено аналіз предметної області розробки комп'ютерних ігор і виявлено ряд проблем, що існують у цій галузі. Основною метою роботи є розробка ігрового додатку "Shoot Them Up".

Під час роботи над проектом було проведено дослідження предметної області розробки комп'ютерних ігор. Були виявлені проблеми, з якими стикаються розробники, а також була встановлена необхідність у створенні нового додатку. Для досягнення успіху було проведено аналіз існуючих програмних продуктів, моделей, методів та технологій. На основі проведеного аналізу були обрані засоби реалізації, що найкраще відповідають потребам проекту.

Окремий розділ присвячено моделюванню додатку. Застосована нотація IEDF0 дозволила провести структурно-функціональне моделювання додатку з погляду розробника та користувача. Також було розроблено процесний алгоритм, що забезпечує логіку та послідовність дій у рамках ігрового додатку.

Останній розділ роботи був присвячений програмній реалізації ігрового додатку "Shoot Them Up". З використанням обраних засобів реалізації було створено функціональний ігровий додаток, який задовольняє поставлені вимоги та відповідає специфікації проекту. Процес програмної реалізації включав розробку імплементації графічного інтерфейсу, логіки гри, обробки взаємодії з користувачем та інших необхідних компонентів.

Ключові слова: додаток, гра, Unreal Engine, C++, шутер.

ЗМІСТ

ВСТУП

IT-фахівець це представники багатьох професій, що працюють у галузі інформаційних технологій. Це включає програмістів, розробників, адміністраторів мереж та баз даних, модераторів, спеціалістів з робототехніки, інформаційної безпеки, web-дизайнерів та 3D-аніматорів. При цьому, з поширенням інформаційних технологій у все нові сфери діяльності, з'являються нові професії для IT-фахівців. заданим на сьогоднішній день і думкою багатьох аналітиків, фахівці даної області є вимогливими і будуть потрібні в найближчому майбутньому. Зокрема, сьогодні у світі з'являється багато перспективних ігрових розробок та проектів [**Ошибка! Источник ссылки не найден.**, 2].

Game development (розробка ігор) є комплексним процесом, що включає в себе розробку ігрового процесу (геймплея), створення архітектурного програмного забезпечення (рушія) або використання готових рішень (Unreal Engine, Unity, GameMaker: Studio), створення візуального навантаження гри та його окремих компонентів, створення та додання спецефектів, а також звуковий супровід.

У процесі розробки ігрового додатку дотримання наукових принципів та методів може виявитись корисним. Наприклад, застосування принципів геймдизайну, що базуються на психології гравців, дозволяє створити захоплюючий та цікавий геймплей. Розробка оптимізованого програмного забезпечення дозволяє підвищити продуктивність гри та забезпечити плавну роботу. Крім того, використання методів комп'ютерної графіки та моделювання дозволяє створити реалістичну та привабливу візуальну складову.

Дослідницькі роботи та наукові статті, присвячені геймдеву, вивчають такі аспекти, як інноваційні технології у геймінгу, вплив ігор на гравців та їх поведінку, процеси творчості в геймдеві, розвиток віртуальної реальності та

доповненої реальності в ігровій сфері, аналіз успішних геймдев-компаній та їх стратегій [524, 5, 5].

Метою бакалаврської роботи є розробка ігрового додатку для розвитку моторики, реакції та розваги гравця з сучасною графікою та ігровим процесом.

Задачі які необхідно виконати:

− визначити актуальність роботи, цільову аудиторію та дослідити предметну область;

− провести аналіз аналогів ігрових додатків і виділити їх переваги та недоліки;

− спроектувати модель та структуру додатку;

− обрати технології для розробки гри;

− реалізувати структуру ігрового додатку;

− створити прототип;

− розробити функціонал додатку;

− виконати тестування гри.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд останніх досліджень і публікацій

В даний момент індустрія ігрових додатків — це одні із найпопулярніших сегментів цифрового контенту у світі. На рисунку 1.1 відображено, як кількість доходу від цієї промисловості зростає з кожним роком:

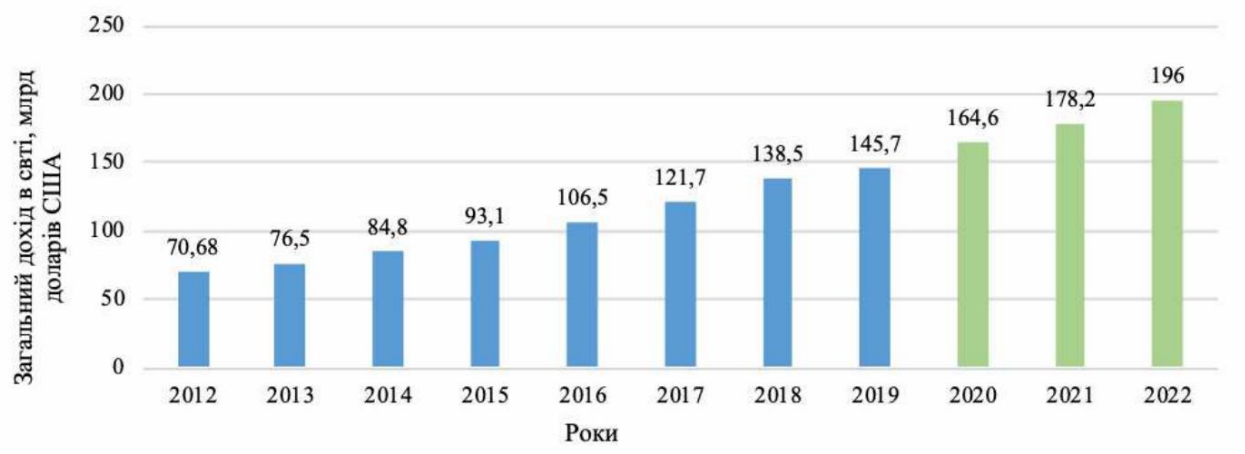


Рисунок 1.1 – Світовий ринок ігор в 2012-2022 роках, млрд. долл. [6]

Ігрові платформи які займають велику частину ринку на даний момент:

• персональні комп'ютери (розробка виконується для різних ОС);

• мобільні пристрої (розробка виконується окремо під Android та IOS);

• консолі (виділяють PlayStation, Xbox, Nintendo).

Перші масові ігрові додатки були тільки для користувачів ПК, але з часом індустрія розвивалась, створювались нові платформи та тепер аудиторія гравців різниться. Так, за даними на 2019 рік [**Ошибка! Источник ссылки не найден.**], майже половини ігрового ринку належить платформі для смартфонів, але більшість все таки на стаціонарних пристроях, таких я персональний комп’ютер чи консоль (рис.1.2).

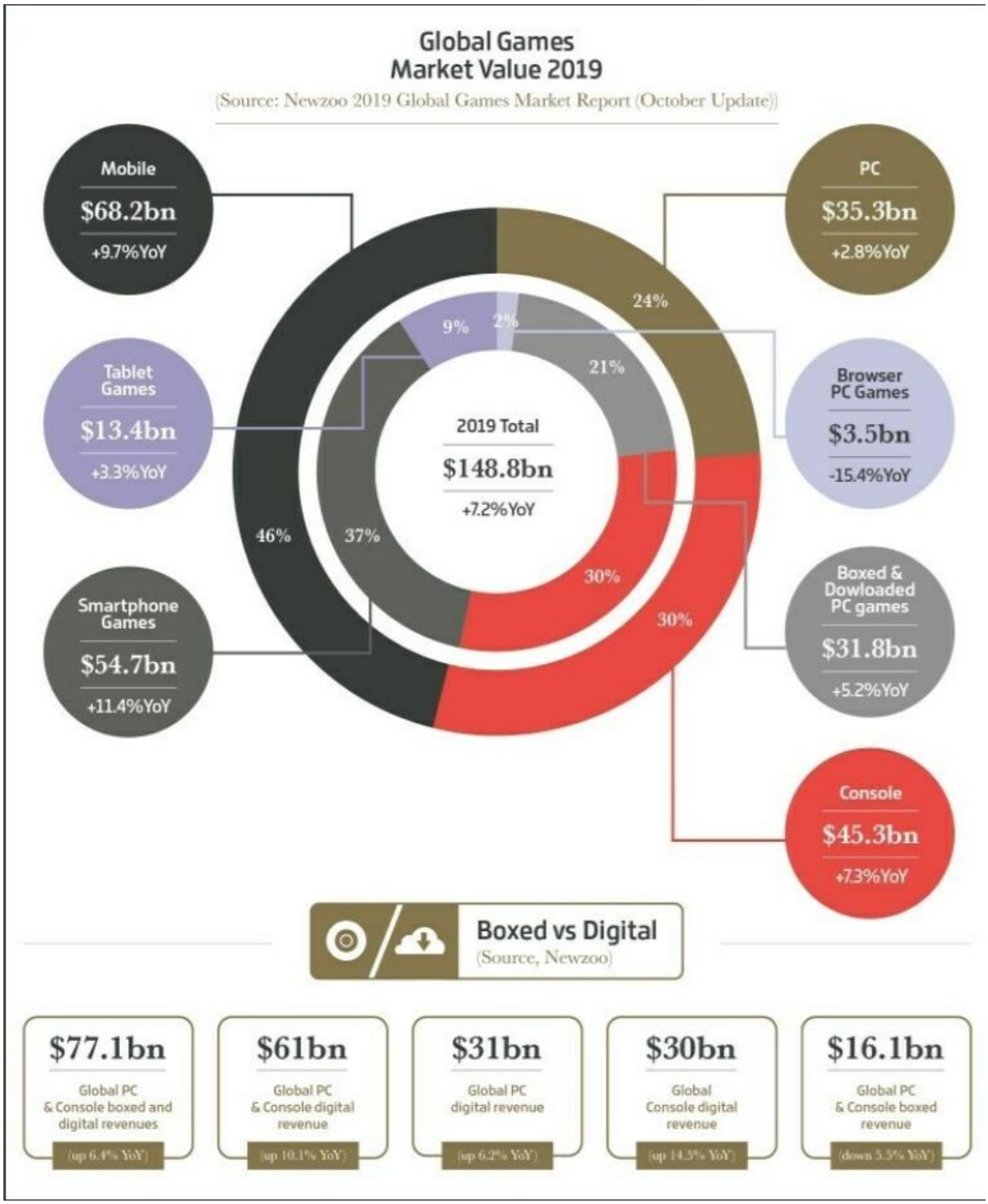


Рисунок 1.2 – Дохід ігрового сегменту на 2019 рік [7]

Так як більшість ринку займають ігри для ПК та консолей, тому було обрано розробку саме під ці платформи, так як розробка відрізняється лише зміною керуючого пристрою у грі та оптимізація під різні види пристроїв.

На рисунку 1.3 відображено популярність операційних систем на території України (статистика 2017 рік):

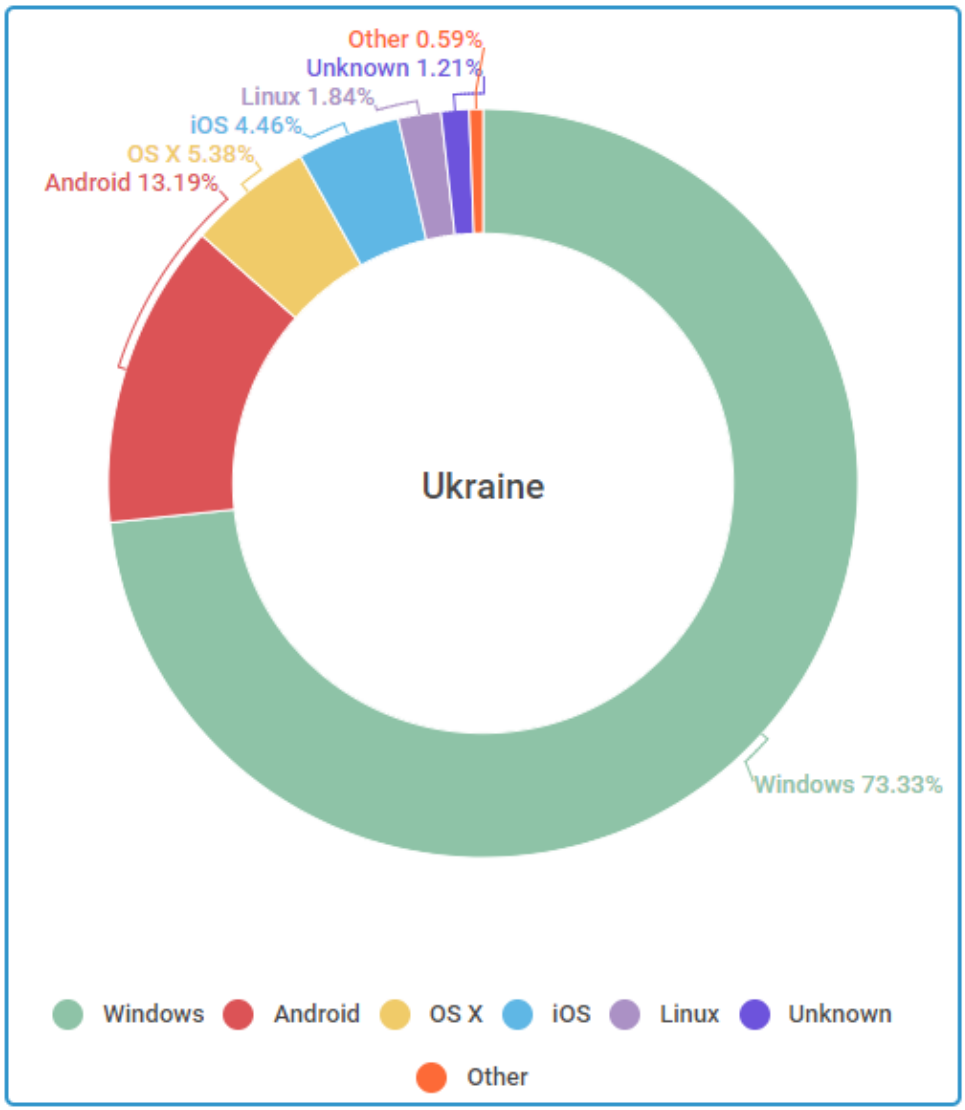


Рисунок 1.3 – Популярність операційних систем для України [8]

Можна зробити висновок, що більшість користувачів персональних комп’ютерів обирають Windows як операційну систему.

Для комп’ютерних ігор виділяють такі найголовніші жанри:

- азартні
- спортивні
- головоломки
- стратегії
- аркади
- 3D-шутер

Шутери є досить популярними щодо інших жанрів з таких причин (рис 1.4) [9]:

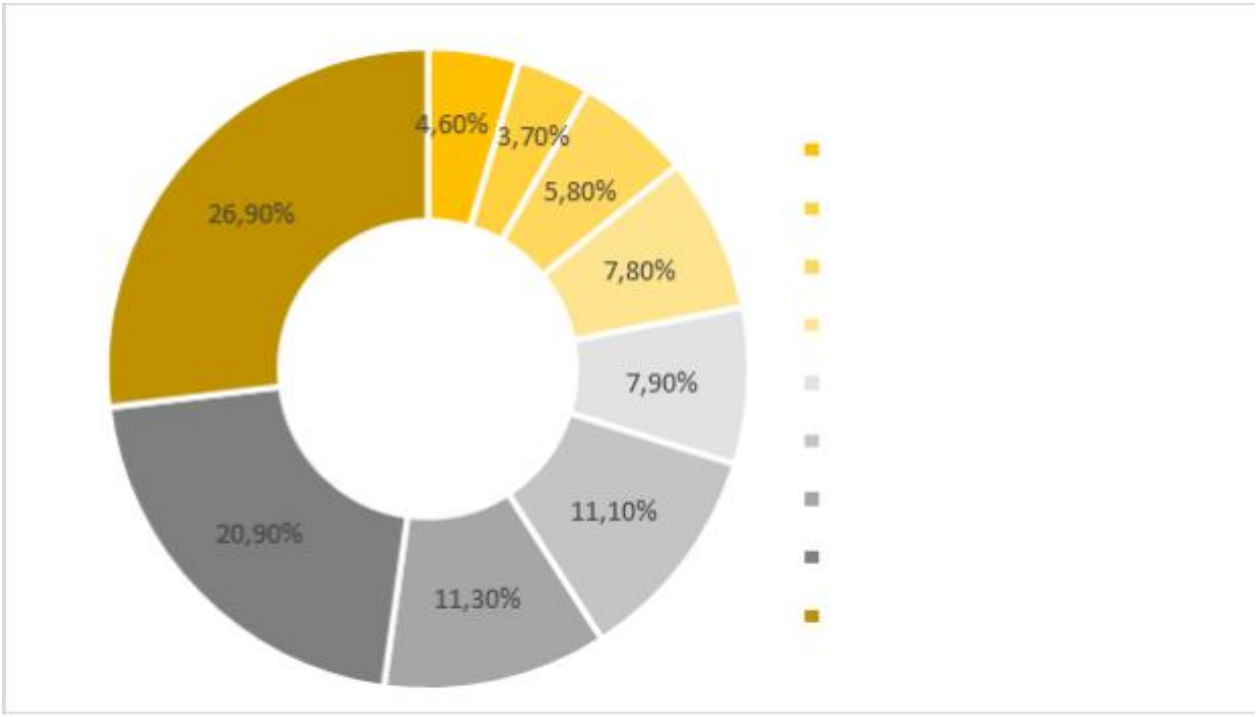


Рисунок 1.4 – Популярність ігрових жанрів, 2018 рік

1. В шутерах простий принцип гри ;
2. Зручні для новачків;
3. Шутери дуже популярні(онлайн гравців);
4. Цікавість стати кращим;
5. Візуальна та звукова атмосфера.

Під час виконання дипломної роботи було проведено аналіз предметної області та встановлено, що є необхідність у створенні додатку, який буде у жанрі «шутер» з сучасною графікою, звуковим наповненням, з цікавим «геймплеєм» та запуском оффлайн за допомогою .exe файлу.

Перелік проблем наведений у таблиці 1.1.

Таблиця 1.1 – Перелік проблем предметної області

№	Назва /Проблема	Зміст	Проблема, що уточнюється	Примітка: роль/ чи інтерес
1	Платний графічний контент	Більшість графічного контенту треба придбати, або наймати художників	Уточнюється та обговорюється з менеджером проекту	Звітуємо команді розробці
2	Довготривалий термін розробки	Більшість компаній по розробці ігор розробляють ігри декілька років	-	Звітуємо менеджеру проекту
3	Велика конкуренція	Не можливість конкурувати з більш популярними схожими проектами	Уточняється з замовником	Звітуємо замовнику
4	Вибір технологій розробки	Більшість технологій для розробки ігр є платними (деякі платні коли компанія заробить певну суму)	Уточняється з замовником, менеджером проекту	Звітуємо замовнику

Щоб вирішити проблеми, що наведені у таблиці вище, для розроблюваного ігрового додатку визначено низку вимог, які представлені у додатку А [Функціональні вимоги].

1.2 Аналіз програмних продуктів-аналогів

На сьогоднішній день існує безліч ігрових програм жанру шутер. Розглянемо найпоширеніші з них та наближених до вихідного ігрового продукту.

Хоч більшість прикладів є мультиплеєрними, але було вирішено додати їх на огляд.

1.2.1 **Unreal Tournament**[10]

Серія Unreal, розроблена компанією Epic Games, здобула велику популярність в ігровій індустрії. Перша гра з цієї серії, випущена в 1998 році, швидко стала культовою. Unreal Tournament (UT), одна з ігор цієї серії, є комп'ютерною грою у жанрі шутера від першої особи. UT відрізняється великим вибором зброї та різноманітними картами для гри. Головною фокусом Unreal Tournament були масові битви з великою кількістю гравців і ця гра стала успішною спробою поліпшити мультиплеєрний режим оригінальної гри Unreal.

Unreal Tournament відомий своїм високорозвиненим штучним інтелектом, який забезпечує складні та викликові противників для гравця. Гра також відрізняється відмінним дизайном рівнів, деякі з яких були взяті з Unreal Tournament 3. Наявність цікавих режимів гри робить Unreal Tournament захоплюючим досвідом для гравців.

У контексті наукового дослідження Unreal Tournament може слугувати прикладом успішної реалізації різноманітних аспектів геймдеву, таких як штучний інтелект, дизайн рівнів та інноваційні режими гри. Дослідження таких успішних ігор допомагає розкрити принципи та методи розробки ігрових додатків, а також сприяє розвитку нових технологій та покращенню якості ігрового досвіду.

Вигляд гри (рис 1.5):

Рисунок 1.5 – Unreal Tournament

1.2.2 **Counter-Strike: Global Offensive**[10]

Counter-Strike: Global Offensive (CS:GO) є багатокористувацькою комп'ютерною грою, спільно розробленою компаніями Valve і Hidden Path Entertainment[10]. Ця гра належить до жанру від першої особи шутерів і розрахована на гру з великою кількістю гравців. Гравці поділяються на дві команди і змагаються один з одним. Основна мета гри полягає в тому, щоб терористи заклали бомбу в одній з декількох точок і уникнули її розмінування, тоді як спецназ має успішно розмінувати її [11, 12, 13].

Counter-Strike: Global Offensive є сучасною грою, яка використовує актуальні шляхи розвитку. Один з важливих аспектів її фінансування полягає в продажу косметичних предметів та скінів для зброї. Це дозволяє грі фінансуватися і забезпечує стабільну підтримку та оновлення. Регулярні оновлення дозволяють грі залишатися свіжою та актуальною, забезпечуючи гравцям нові функції та вдосконалення.

CS:GO славиться своєю механікою стрільби та простотою геймплею, що можна успішно перенести в реалізацію ігрового додатку "Shoot Them Up". Ця

гра надає можливість гравцям насолоджуватися інтенсивними битвами та стратегічними вирішеннями в реальному часі.

Вигляд гри (рис 1.6):

Рисунок 1.6 – Counter-Strike: Global Offensive

1.2.3 **Mass Effect**[14]

Mass Effect— комп'ютерна гра в жанрі рольового бойовика, розроблена студією BioWare і випущена Microsoft Game Studios.

Основні ігри серії є шутером від третьої особи з елементами рольової гри. Присутня система завдань. У кожному з них гравець під час діалогу може вибирати різні варіанти відповіді, які впливають сюжет. Різновидність локацій та противників у цій грі є гарним напрямком зацікавити гравця.

У Mass Effect ігровий процес ведеться від третьої особи, що буде використано у дипломному проекті.

Вигляд гри (рис 1.7):

Рисунок 1.7 – Mass Effect

Таблиця 1.2 – Порівняльна таблиця характеристик ігор у жанрі шутер:

Характеристика / Гра	Unreal Tournament	Counter-Strike: Global Offensive	Mass Effect
Вид від 3-ої особи	-	-	+
Мультиплеєр	+/-	+/-	-
Різні види зброї	+	+	+
Сучасний штучний інтелект	+	+	+
Випадкова віддача зброї	-	+	-
Зміна режиму гри	+	+	-
Графіка	4/10	8/10	9/10
Архітектура	Unreal Engine 1.5 / Unreal Engine 3	Source	Unreal Engine 3/ Frostbite 3
Різні мапи/рівні	+	+	+

За допомогою таблиці 1.2 варто придивлятись на можливості та характеристики, які надають великий вплив на геймплей гри, під час розробки.

Розроблюваний ігровий додаток повинен мати сучасну графіку, різні локації/рівні, цікавий ігровий процес та інтерфейс. З функціональних доповнень варто виділити архітектуру, застарілу графіку у всіх випадках та наявність мультиплеєру.

Порівняння між грою Counter-Strike, Unreal Tournament і Mass Effect:

Жанр гри:

- Counter-Strike: Global Offensive (CS:GO) і Unreal Tournament - обидві гри належать до жанру шутерів від першої особи. Однак, CS:GO більше фокусується на множинних ігрових режимах, таких як командні битви та виконання місій, тоді як Unreal Tournament більше спрямований на швидкі та екшн-орієнтовані мультиплеєрні битви.
- Mass Effect - це гра, що поєднує жанри рольової гри (RPG) та шутера від третьої особи. Гра відзначається наявністю глибокої сюжетної лінії, характерного для RPG, а також активної бойової системи в реальному часі.

Режими гри:

- Counter-Strike: Global Offensive (CS:GO) має різноманітні множинні режими, такі як збирання/розмінування бомби, звільнення заручників або командні смертельні матчі. Гра пропонує ігровий досвід, заснований на стратегічних рішеннях та командній грі.
- Unreal Tournament відомий своїми швидкими та експресивними мультиплеєрними битвами, де гравці змагаються у широкому спектрі режимів, включаючи смертельні матчі, захоплення прапорів та знищення ворожих баз.
- Mass Effect пропонує глибоку сюжетну кампанію, яка зосереджена на взаємодії з персонажами, прийнятті важливих рішень та розвитку персонажа. Гра також має екшн-орієнтовану систему бою, в якій гравці можуть використовувати зброю та бойові навички для перемоги у ворожих сутичках.

Графіка та дизайн:

- Counter-Strike: Global Offensive (CS:GO) використовує деталізовану, реалістичну графіку, що дозволяє створити атмосферу сучасного бойового середовища. Дизайн рівнів і об'єктів відповідає реальним місцям та сценаріям.
- Unreal Tournament має більш насичену, футуристичну графіку. Рівні та персонажі виражені у стилі науково-фантастичного світу, з великою кількістю яскравих кольорів та деталей.
- Mass Effect вражає своєю деталізованою графікою та вражаючими кінематографічними сценами. Гра відрізняється відмінною прорисовкою космічних ландшафтів, ефектів світла та дизайном різноманітних галактичних світів.

Сюжет та глибина гри:

- Counter-Strike: Global Offensive (CS:GO) має просту сюжетну лінію, орієнтовану на командні битви і завдання. Основний акцент робиться на геймплеї та багатошаровій грі.
- Unreal Tournament більше спрямований на безперервну екшн та швидкість гри, і сюжет виступає в менш важливій ролі.
- Mass Effect славиться своєю глибокою сюжетною лінією, історичними розколами та моральними дилемами. Гра надає гравцям велику свободу в прийнятті рішень, які впливають на розвиток сюжету та взаємодію з персонажами.

Унікальні особливості:

- Counter-Strike: Global Offensive (CS:GO) відомий своїми великими турнірами та конкурентними ігровими з магами, де професійні гравці змагаються за призові гроші. Гра також має активну спільноту моддерів, які створюють різноманітні модифікації та додатковий контент для гри.
- Unreal Tournament відрізняється великим вибором зброї та різноманітними картами, що дозволяє гравцям насолоджуватися широким спектром геймплею. Особливу увагу приділено штучному інтелекту, що

забезпечує високий рівень виклику та наснаги в битвах з комп'ютерними супротивниками.

- Mass Effect відрізняється своєю глибиною сюжету та ролевою системою. Гравці можуть приймати рішення, які впливають на хід історії та взаємодію з персонажами. Також у грі є елементи дослідження космосу, розвитку персонажа та прийняття стратегічних рішень у бою.

В цілому, Counter-Strike, Unreal Tournament і Mass Effect представляють різні підходи до геймплею та надають унікальні ігрові враження. CS:GO акцентується на мультиплеєрних битвах та конкурентному геймінгу, Unreal Tournament надає швидкий та експресивний мультиплеєрний досвід, а Mass Effect пропонує глибоку сюжетну лінію і поєднує елементи рольової гри та шутера.

За допомогою вивчення проблем предметної області було сформовано вимоги, що допомогло створити повну постановку задачі. Було проведено порівняння існуючих додатків та детально дослідженні можливостей архітектур та мов програмування та обрано найкращі технології розробки.

При аналізі інших шутерів були виявлені наступні елементи:

• у всіх шутерах застаріла, на 2023 рік, графіка;

• без мультіплеєру ігри також можуть бути досить популярними;

• в деяких іграх присутній сюжет, який потребує час та гарного спеціаліста;

• у всіх розглянутих екземплярах присутній розумний штучний інтелект.

1.3 Вибір та обґрунтування засобів реалізації

Так як програмний додаток є грою, то потрібно розглянути забезпечення націлене на сторення проектів такого характеру. Для вибору програмного забезпечення буде розглянуто два види, а саме ігрові архітектури (рушії) та мову програмування.

Для вибору ігрової архітектури було обрано Unity (3D), CryEngine та Unreal Engine (4):

Unity ─ це ігрова архітектура для створення ігор, здебільшого у 2D та для мобільних пристроїв. Але за допомогою можливості створювати саме 3D, вона є непоганим рішенням у розв'язку поставленої задачі. DirectX і OpenGL хоч і є вже базовими технологіями, але підтримка технологій у архітектурі є плюсом. За допомогою мови програмування C# здійснюється більша частина логіки та скриптів в Unity 3D, також для інтерфейсу використовують JavaScript.

Переваги:

• невеликі витрати ресурсів (порівняно з іншими архітектурами);

• кросплатформність;

• велика кількість інструментів розробки;

• потребує не потужний комп'ютер для роботи.

Недоліки:

• складність створення масштабних ігр;

• платна комерційна ліцензія;

• відсутність підтримки в Unity посилань на зовнішні бібліотеки.

CryEngine ─ архітектура для створення вже меншого жанру ігор, через недостатність інструментів для розробки, але може виділитись гарною графікою. Націлений на AAA-сегмент розробки ігор, основним напрямом якого є створення фотореалістичних шутерів від першої особи. Данний напрямок потребується у вирішення задач проекту, але рушій має велику низку проблем.

Переваги:

• високі показники графіки;

• кросплатформність.

Недоліки:

• регулярні помилки клієнта;

• мала община розробників;

• мізерна кількість учбових матеріалів та документацій;

• спрямовано створення певного жанру ігор;

• потребує великі ресурси ПК від розробника.

Unreal Engine ─ ігровий рушій, який як і попередник використовується для розробки AAA проектів. У більшості випадків використовується як 3D засіб. Починаючи із четвертої версії ─ розповсюджується безкоштовно. Є основним рушієм більшісті сучасних ігор.

Переваги:

• повна безкоштовність;

• кросплатформність;

• графіка;

• великий вибір інструментів;

• велика спільнота користувачів;

• більша за конкурентів кількість учбового матеріалу та документації.

Недоліки:

• високі технічні вимоги;

• складність в освоєнні.

2) мови програмування.

Також варто визначитися - яку мову програмування використовувати. На сьогоднішній день розглянуті архітектури використовую такі мови програмування для розробки:

1. C#

Мова програмування C# легша в освоєнні, через що дозволяє процес розробки буде швидше. C# містить автоматичний сборщик сміття для об’єктів. Для платформ, окрім Windows, мало де розповзюджений, хоч і є кросплатформним. Для невеликих проектів продуктивність C# є невідмінним та йде на рівні з іншими мовами програмування, але при масштабуванні проекту швидкість роботи доволі зменшується, що може стати великою

проблемою у розглянутому проекті. C# має значну кількість бібліотек, як і інші мови, що суттєво полегшує розробку. Також логіка в Unity 3D пишеться на C#, а в CryEngine реалізовуються деякі механіки.

2. C++

Мова програмування C++ є складнішою у освоєнні, але більш швидкою та ефективною. Через свою більш контрольованість є основною мовою програмування розробки ігор. Дана мова програмування реалізує логіку в Unreal Engine та CryEngine.

І C#, і C++ можна використовувати для створення ігор. Однак C++ має краще апаратне забезпечення для керування ПК або сервером. Тому зазвичай це більш придатна мова для розробки ігор.

При порівнянні C# з C++ можна виділити те, що ігри створюються набагато швидше, ніж за допомогою C++. Але ігри, розроблені на C++, як правило, працюють швидше та є більш відшліфованими. C# має мізерний спектр інструментів для роботи, а C++ пропонує повний контроль над кожним аспектом поведінки програми. Для мови C++ є 2 різні архітектури, це в Unreal Engine та CryEngine. CryEngine в порівнянні потребує додаткових знання Lua та C#, має меншу користувацьку підтримку та погану документацію, не поширюється на всі види консолей та ОС, має дуже високу деталізацію. Unreal Engine – є швидшою, має більшу функціональність, оптимізація краща, команда розробників потрібна менша.

Хоча CryEngine має візуальну потужність, Unreal Engine має набагато більш фундаментальний рівень функціональності та більш універсальні параметри для кількох різних графічних параметрів, створена гра не буде базуватися тільки на графіці та візуальних ефектах тому було прийняте рішення використовувати C++ з Unreal Engine, а саме 4 версію.

1.4 Постановка задачі

Метою даного дослідження є розробка ігрового додатку для розвитку моторики, реакції та розваги гравця з сучасною графікою та ігровим процесом. Результатом буде початкову кількість гравців до 10 тис. та додання платних графічних товарів, щоб поступово зібрати суму для розширення компанії.

Таким чином, на цьому етапі для гри «Shoot Them Up» були сформовані наступні вимоги:

• гра буде представлять із себе шутер від 3-ої особи;

• в грі присутній гравець, вороги, зброя, союзники та вороги(залежить від обраного режиму);

• в грі присутня система очок(вбивств та смертей) і компонент здоров'я;

• по закінченню часу всіх раундів або наявності на рівні тільки останнього гравця ― завершення гри;

• сенс гри ― набрати найбільшу кількість очок або стати останнім живим гравцем.

Задачі, які виконуємо для досягнення мети:

− визначення актуальності роботи, цільову аудиторію на яку буде розповсюджено додаток та дослідження предметної області;

− проведення аналізу ігрових додатків, які є подібними за змістом та механікою, та програмним забезпечення, виділяємо їх переваги та недоліки;

− проведення функціонального моделювання ігрового додатку;

− проектування моделі та структури додатку;

− обрання технології для розробки гри;

− реалізацію структури ігрового додатку;

− розроблюємо прототип;

− реалізуємо функціонал додатку;

− виконуємо тестування гри.

Цільовою аудиторією гри є люди віку від 14 років які зацікавленні у веселому, простому та динамічному «геймплеї».

Технічне завдання на розробку продукту у повному обсязі наведено у додатку А.

2 МОДЕЛЮВАННЯ ТА ПРОЕКТУВАННЯ

Так як вже було проведено аналіз предметної області та визначена мета наступним етапом йде моделювання та проектування ігрового додатку.

2.1 Моделювання програмної реалізації

Функціональна модель IDEF0 відображає функції та структуру системи, а також потоки матеріальних об'єктів та інформації, які перетворюються за допомогою цих функцій. За допомогою методології створено діаграми для показу ієрархічного та поетапного розподілу процесів.[15, 16**Ошибка! Источник ссылки не найден.**]

IDEF0 діаграма складається з блоків, які відображають процеси робіт та стрілок, які відображають управління, механізми та дані.

Для діаграми IDEF0 на тему: «Реалізація ігрового додатку-шутер Shoot Them Up», створено параметри для розробника (рис.2.1).

Вхідними даними до створення ігрового додатку є розроблене технічне завдання (Додаток А). Для управління нами використані правила ігрового процесу та системні вимоги. Механізмами які управляють є програмне та апаратне забезпечення, розробник, тестувальник. Вихідними даними є готовий ігровий додаток.

З точки зору користувача (рис. 2.2), вхідними даними є пристрої вводу(клавіатура, маніпулятор миша, джойстік). Для управління - це правила ігрового процесу та системні вимоги. Механізмами є програмне та апаратне забезпечення та користувач. Вихідними даними є результат гри.

Контекстні діаграми ігрового додатку «Shoot Them Up» в нотації IDEF0 представлені на рисунку 2.1-2.2.

Рисунок 2.1 – Контекстна діаграма IDEF0 з погляду розробника

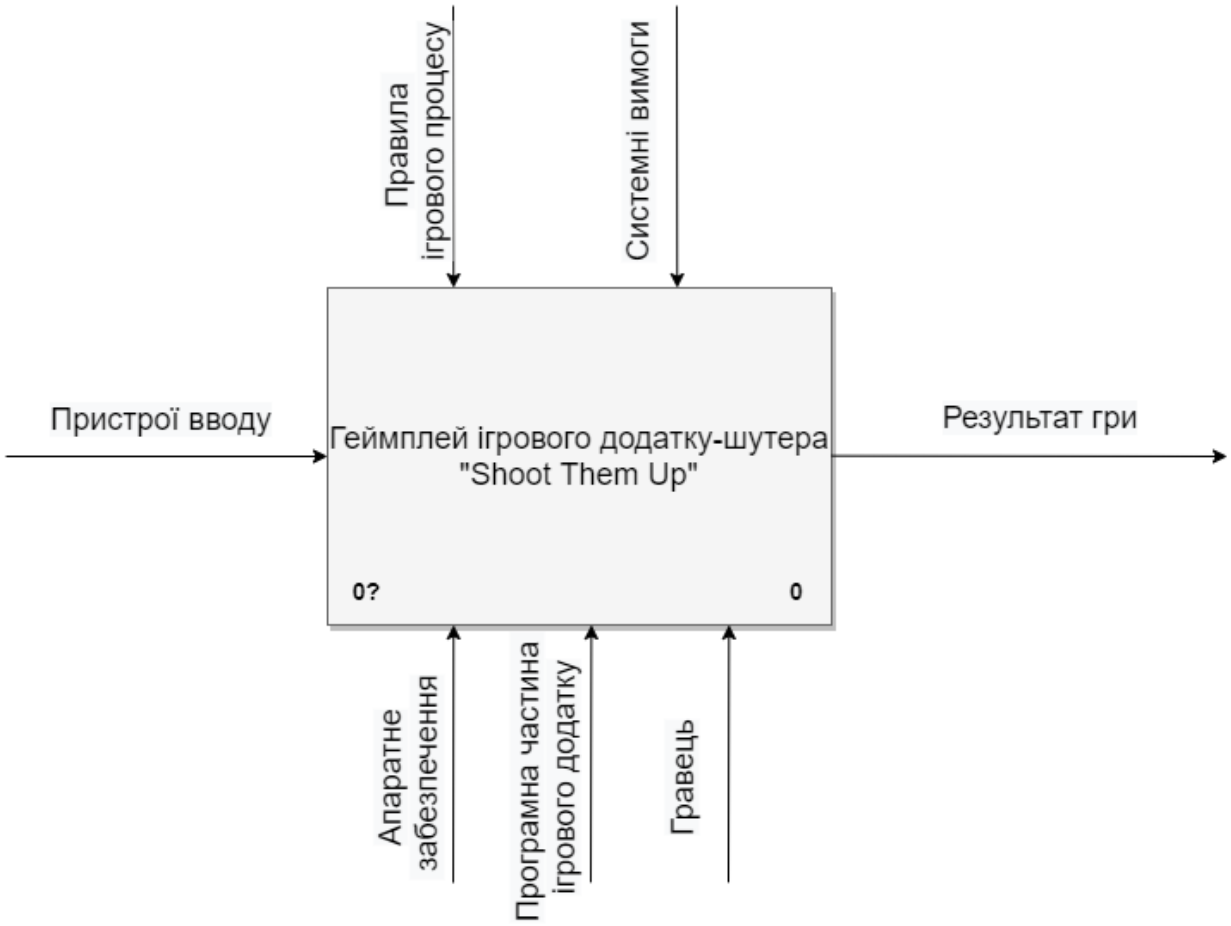


Рисунок 2.2 – Контекстна діаграма IDEF0 з погляду користувача

На рисунку 2.3-2.4 зображена декомпозиція діаграм реалізації ігрового додатку-шутер «Shoot Them Up».

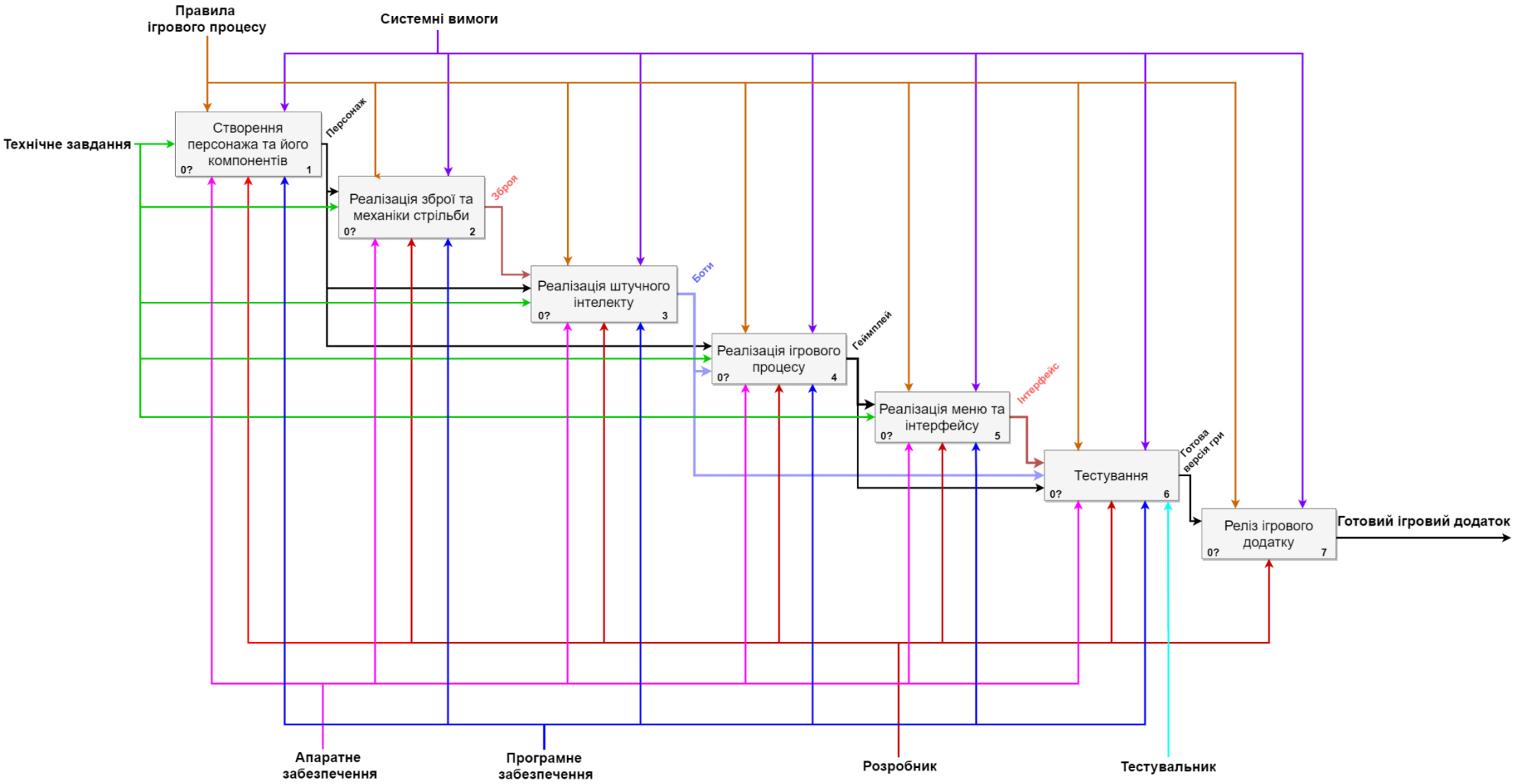


Рисунок 2.3 – Декомпозиція діаграми IDEF0 з погляду розробника

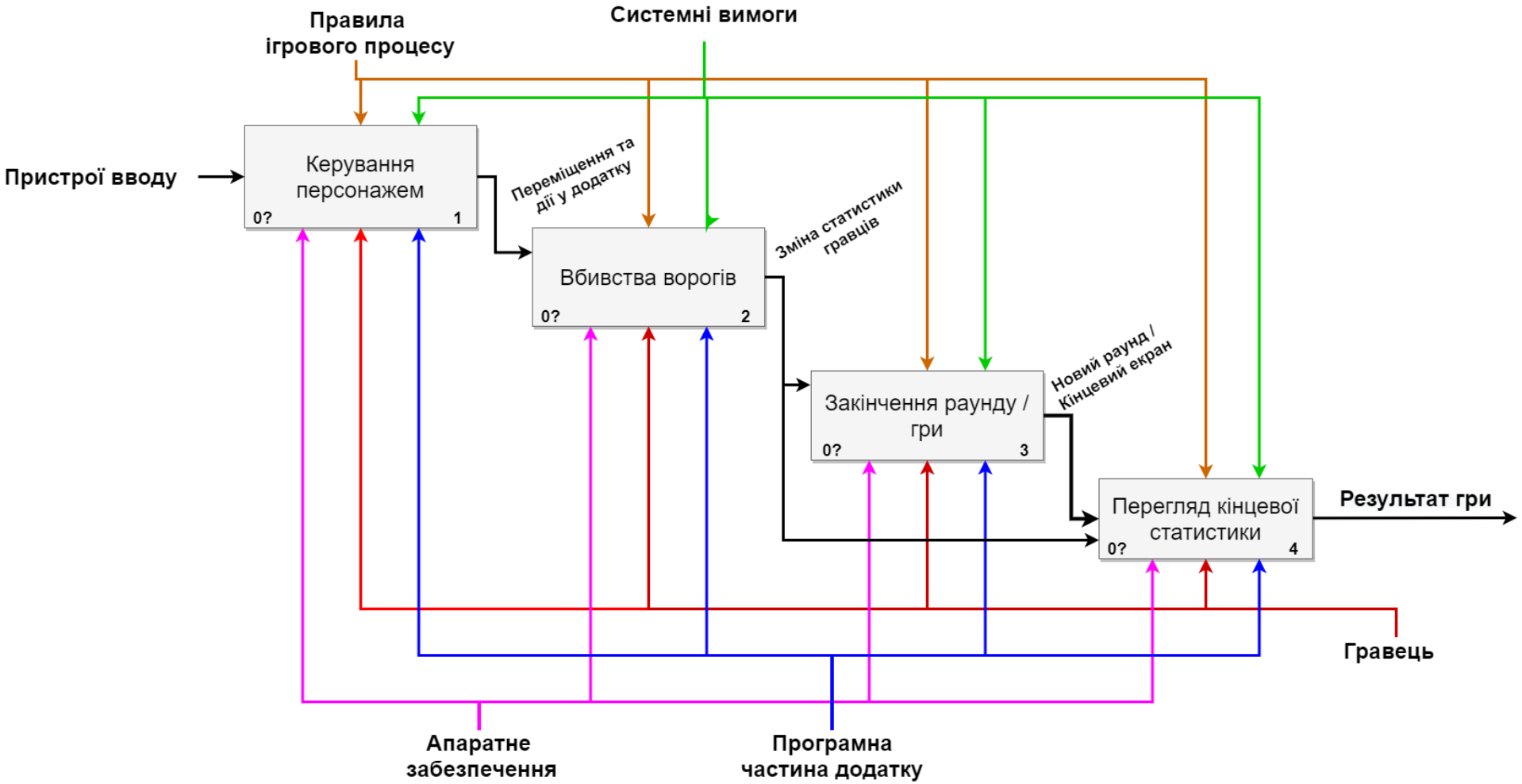


Рисунок 2.4 – Декомпозиція діаграми IDEF0 з погляду користувача

2.2 Моделювання варіантів використання

UML - найпростіша з поведінкових діаграм. Це відображення, яке вказує функціональні особливості програми. Ця діаграма використовується для опису цілей, які переслідує користувач програми. Діаграма відображає як саме програмою потрібно користуватись, та які можливості для користувача вона має.[17]

Використання діаграми варіантів використання дозволяє точно визначити цілі користувача і вимоги до програми. Вона відображає взаємодію між користувачем і програмою, а також взаємодію програми з іншими системами або компонентами. Це дозволяє розробникам чітко зрозуміти потреби користувачів і спрямувати свої зусилля на реалізацію відповідних функцій.

Загалом, діаграма варіантів використання є потужним інструментом для опису функціональних вимог до програмного забезпечення. Вона допомагає замовникам, розробникам і всім зацікавленим сторонам отримати чітке уявлення про те, як програма буде використовуватись і які користувальницькі цілі вона вирішує.

Спираючись на технічне завдання, визначаємо акторів та варіанти використання в системі ігрового додатку.

Актори в системі:

- Користувач / гравець.

Варіанти використання в системі:

- Обирання режиму та рівня гри – вибір режиму гри у головному меню;
- Початок гри – перехід на рівень гри та запуск ігрового процесу;
- Вихід з гри – завершення процесу ігрового додатку;
- Пауза – призупиняє ігровий процес;

- Кінець гри – зупиняє ігровий процес та відображає меню статистики;
- Зміна параметру звуку – керує налаштування гучності додатку;
- Продовження гри – продовження ігрового процесу;
- Вихід до головного меню – перехід на сцену головного меню, завершуючи ігровий процес;
- Перезапуск рівня – перезавантажує рівень та ігровий процес спочатку;
- Перегляд статистики гравців – відображення повної статистики гравців.

На рисунку 2.5 зображена діаграма варіантів використання ігрового додатку-шутер «Shoot Them Up».

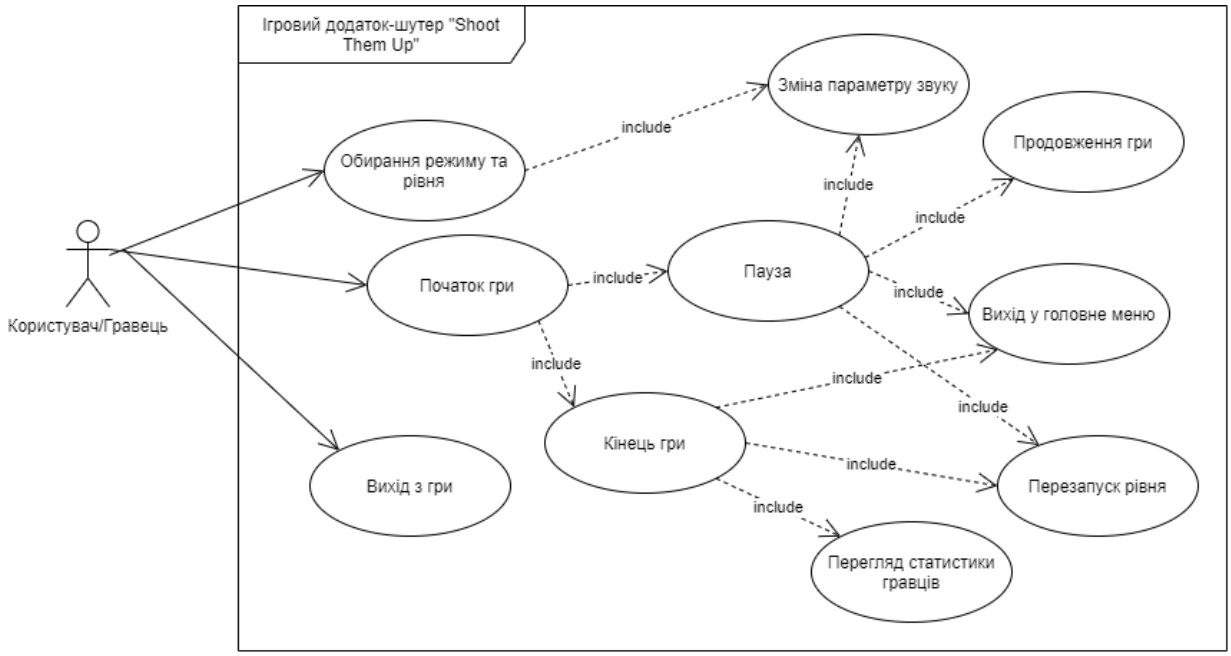


Рисунок 2.5 – Діаграма варіантів використання

2.3 Проектування процесного алгоритму

Для проектування процесного алгоритму використано графічне представлення блок-схемою. Головна мета блок-схеми – зображення послідовності процесів програми у вигляді різних видів фігур та стрілок, які

з’єднують блоки.[18] У блок-схеми обов’язково повинен бути початок (початок роботи програми) та кінець (зупинка всіх процесів).

На рисунку 2.6 зображено блок-схему ігрового додатку-шутер «Shoot Them Up»:

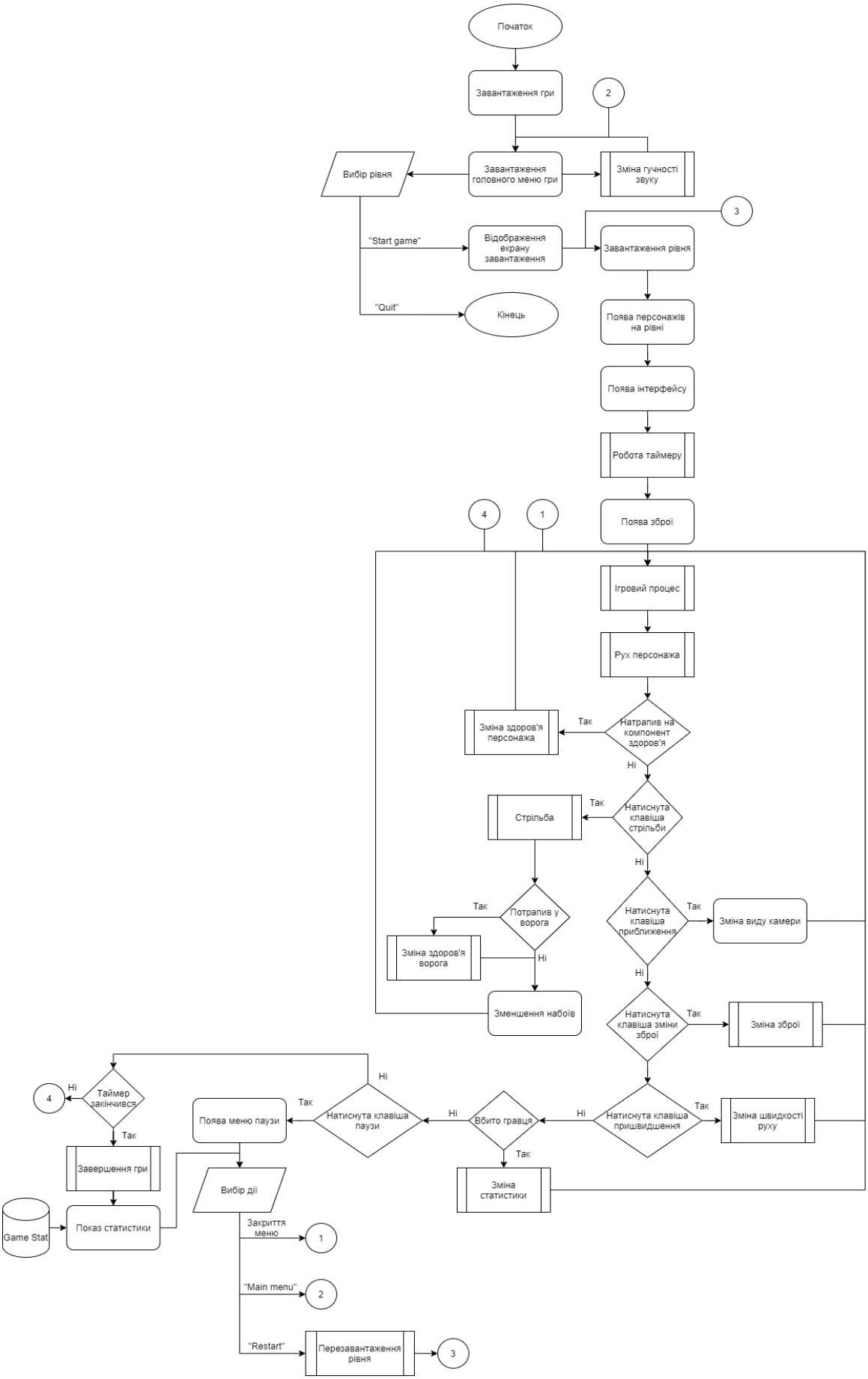


Рисунок 2.6 – Блок-схема ігрового додатку

3 РОЗРОБКА ІГРОВОГО ДОДАТКУ

3.1 Програмна реалізація

3.1.1 Персонажа

Після базових першочергових дій по створенню проекту, обранню шаблона від 3-ої особи та додання стартового контенту, можна переходити до створення персонажа. Потрібно створити базового персонажа спочатку у коді, а вже потім унаслідуватись від нього і створити Blueprint Class. За допомогою зразка шутеру, наданий у Epic Games, зробити міграцію Character Mesh з його Skeletal Mesh, текстурами, фізичними матеріалами та колізіями. Вигляд персонажа (рис. 3.1):

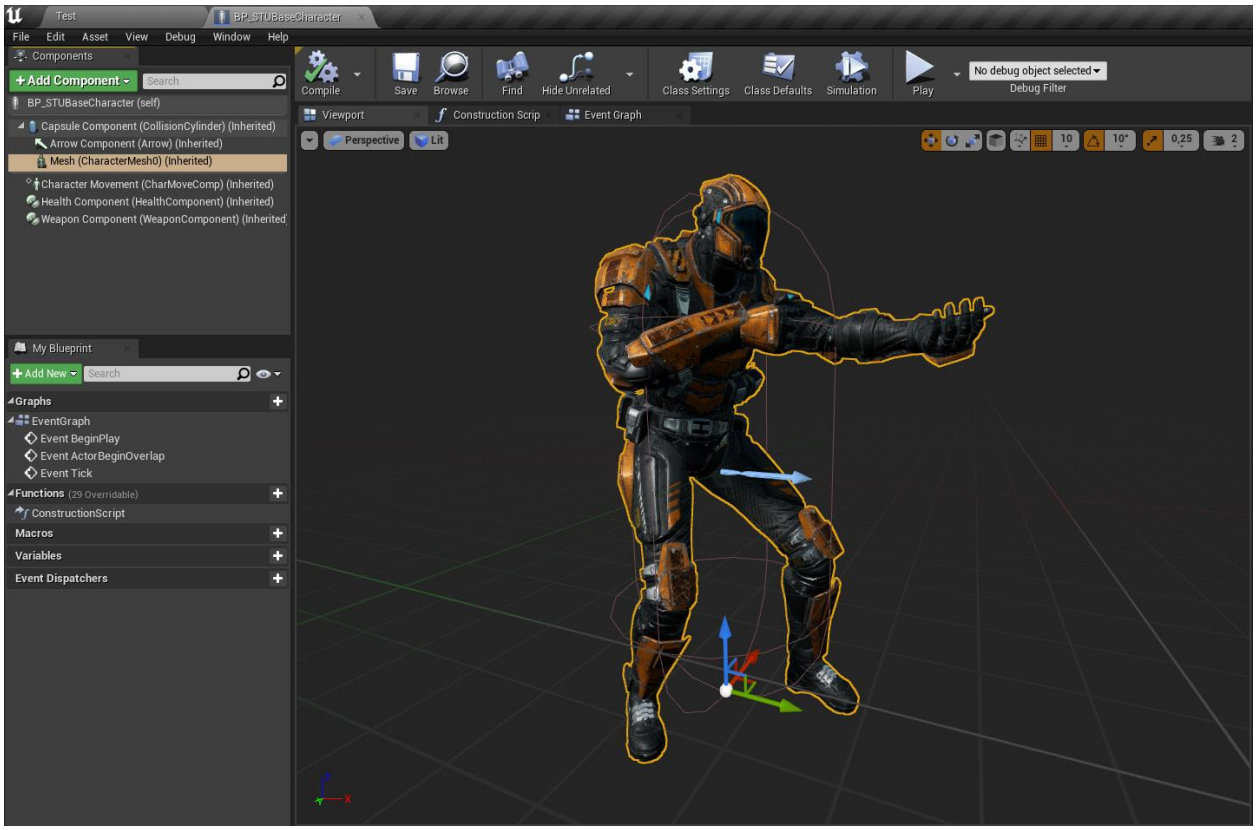


Рисунок 3.1 – Головний персонаж

Далі розпочинається програмна реалізація руху персонажа за допомогою мови C++, для цього потрібно створити Player Controller через який

можлива взаємодія з персонажем та реалізувати функції у Player Character файлі коду, де будуть реалізовані всі функції взаємодії пристроїв вводу з грою, додатково назначити клавіши у налаштуваннях проєкта назначити Action та Axis Mapping. Вигляд Input налаштувань (рис. 3.2):

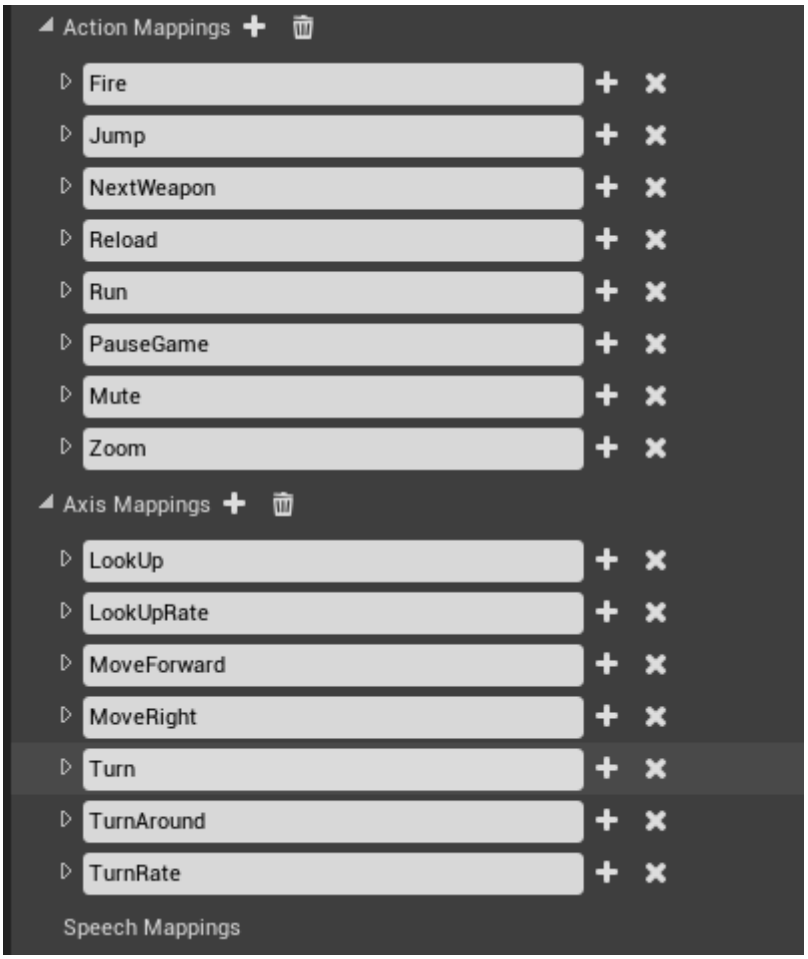


Рисунок 3.2 – Input налаштування

Після створення руху персонажа стає зрозуміло що потрібні анімації, які також містяться у зразці шутеру від Epic Games. У анімаційному блюпринті потрібно створити State Machine з базовими анімаціями пересування (рис. 3.3) та додати його на анімаційний граф, де також додати Aim Offset для коректного відображення погляду персонажа.

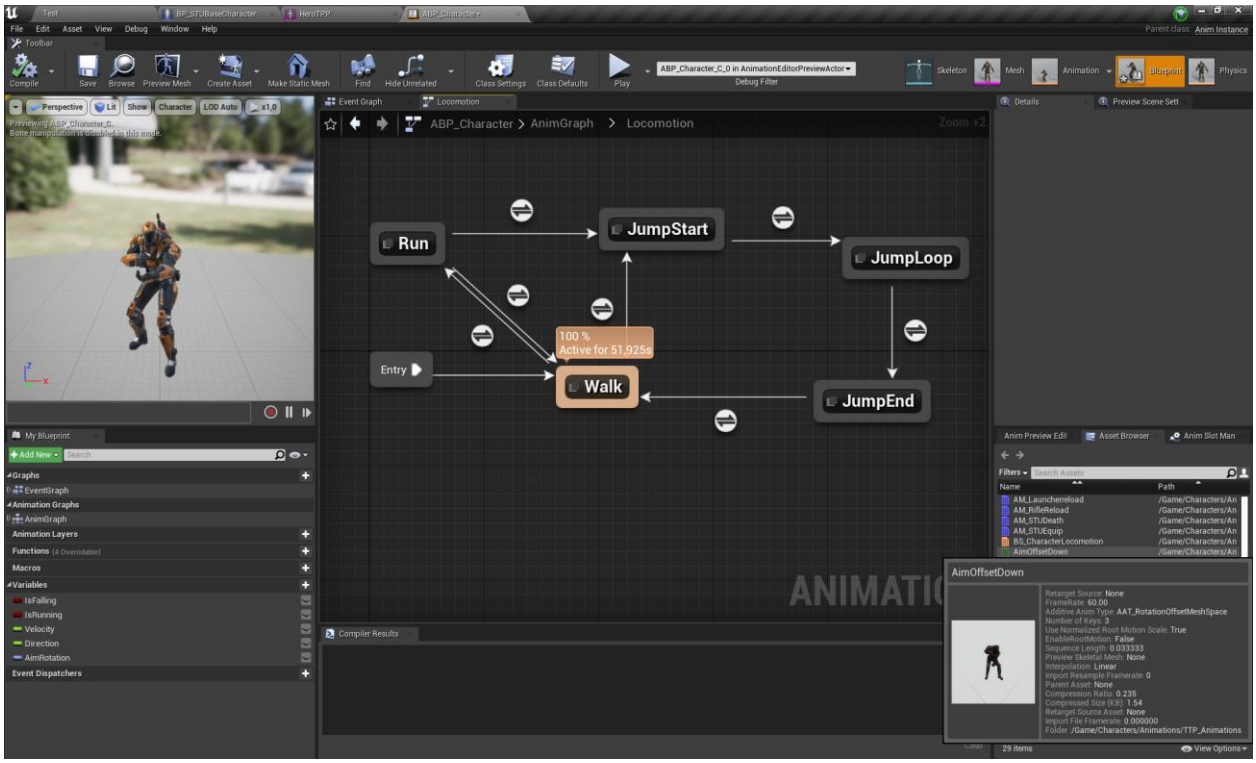


Рисунок 3.3 – State Machine пересування

3.1.2 Здоров’я

Коли всі анімації персонажа встановлені можна почати процес створення компоненту здоров’я гравця. В ньому потрібно створити функції отримання пошкоджень (3 видів), авторегенерації здоров’я, встановлення на певне значення очок здоров’я та смерті персонажа. Також створено Spectator Pawn для вільного руху та перегляду у разі смерті героя.

У компонентів в Unreal Engine немає візуальної частини, вони тільки запрограмовані на певні функції та додаються до блюпринтів.

3.1.3 Зброя

Наступним кроком є створення базованого класу зброї з імпортом Mesh та текстур. Також паралельно розроблено прототип прицілу, для правильного тестування стрільби та подальша його переробка у фінальний вигляд. Створено компонент зброї для повного відслідковування інформація щодо обраної зброї, набоїв та функції, наприклад: почати та зупинити стрільбу, змінити зброю, перезаряджання, приближення камери, спорядити зброю,

ініціалізація та програвання анімацій, багато перевірок для коректного спрацювання основних функцій.

Паралельно з компонентом розроблювалась механіка стрільби базової зброї. Для цієї реалізації було обрано створити віртуальний метод MakeShot та ще додаткові функції зв’язані зі стрільбою, які у подальшому перенесуться у клас певної зброї, або стануть віртуальними. Логіку прорахунку польоту патрона було реалізовано через Line Trace.

Також на цьому етапі було взаємодія з компонентом здоров’я та реалізовані функції Make Damage для різного типу шкоди.

Вигляд першої зброї (рис. 3.4):

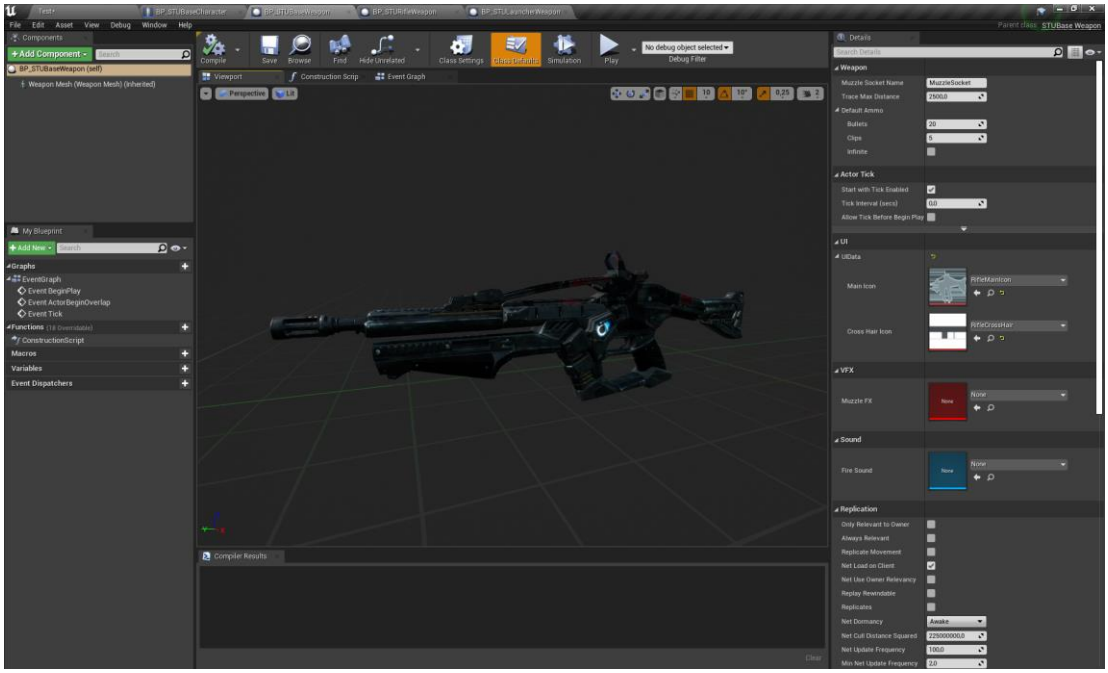


Рисунок 3.4 – Base weapon

Після реалізації першого прототипу зброї потрібно було доповнити анімації зміщення прицілювання для коректного відображення стрільби. Проведено налаштування інверсної кінематики для кістки руки персонажа. Створено таймер та функцію для регулювання кількості пострілів за певний час.

Далі створено два нових класи зброї для реалізацію гвинтівки та базуки (рис 3.6-3.7). Більшість функцій базової зброї було перенесено у клас

гвинтівки, а для базуки було створений додатковий об’єкт ракета. Реалізовано спаун ракети, її рух та логіку при потраплянні у об’єкти. Вигляд снаряду (рис. 3.5):

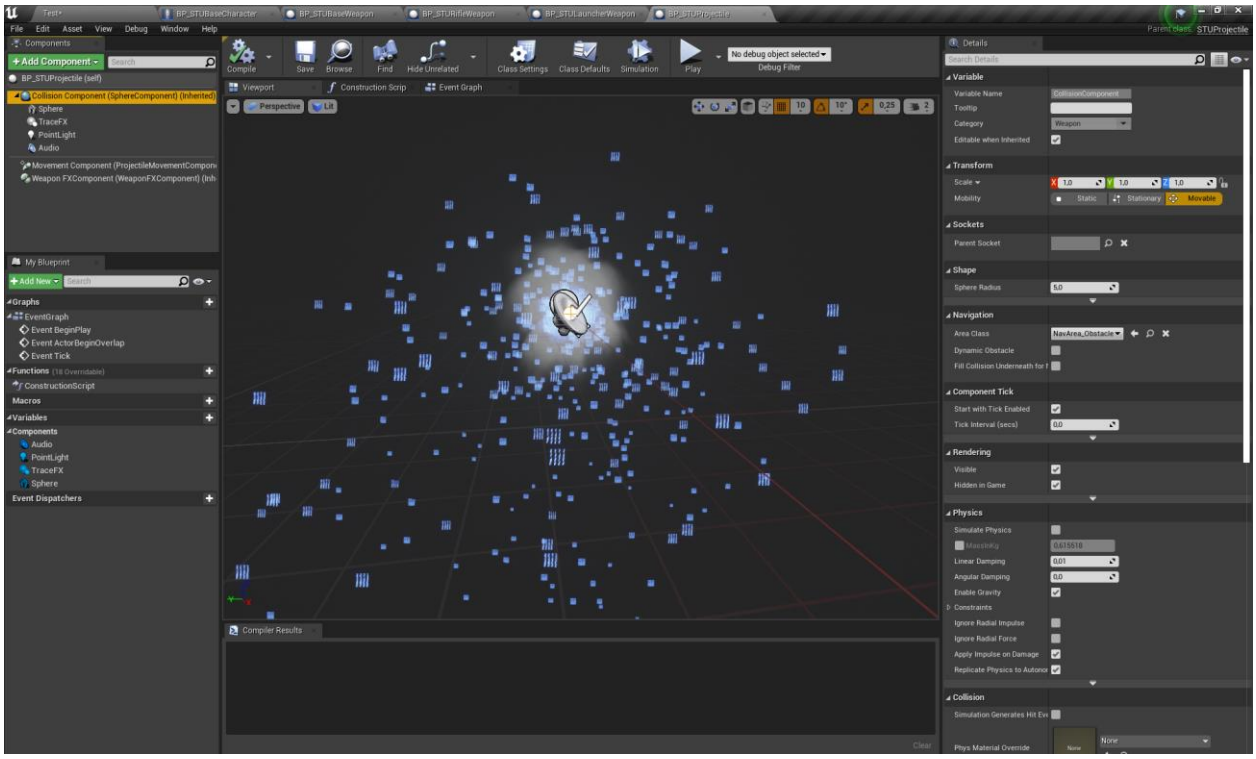


Рисунок 3.5 – Projectile

Після цього будо додано анімації екіпірування для зброї при початку гри та їх зміні, анімації перезарядження зброї різних типів. Додано функції для обмеження стрільби при анімації.

Було розроблено окремий файл для структур які будуть міститися у грі. На даному етапі створено структуру інформації про патрони, про зброю та про UI данні зброї.

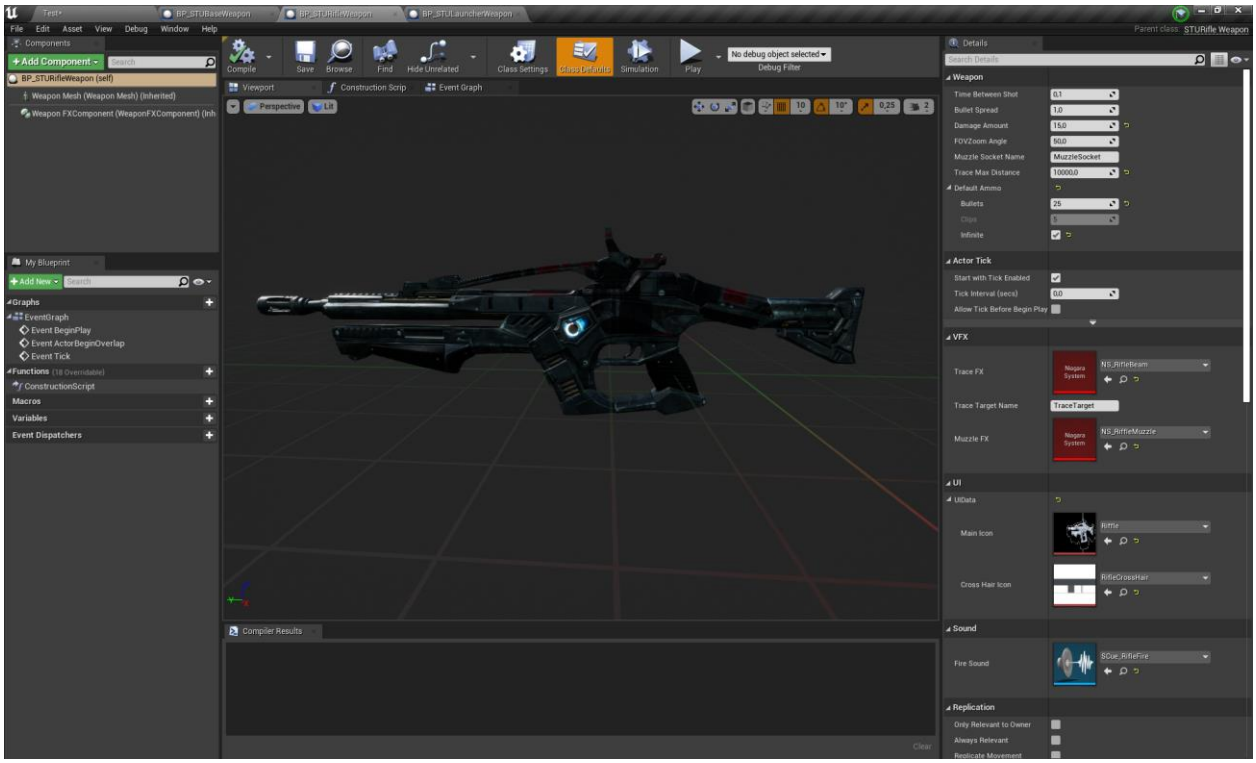


Рисунок 3.6 – Rifle weapon (гвинтівка)

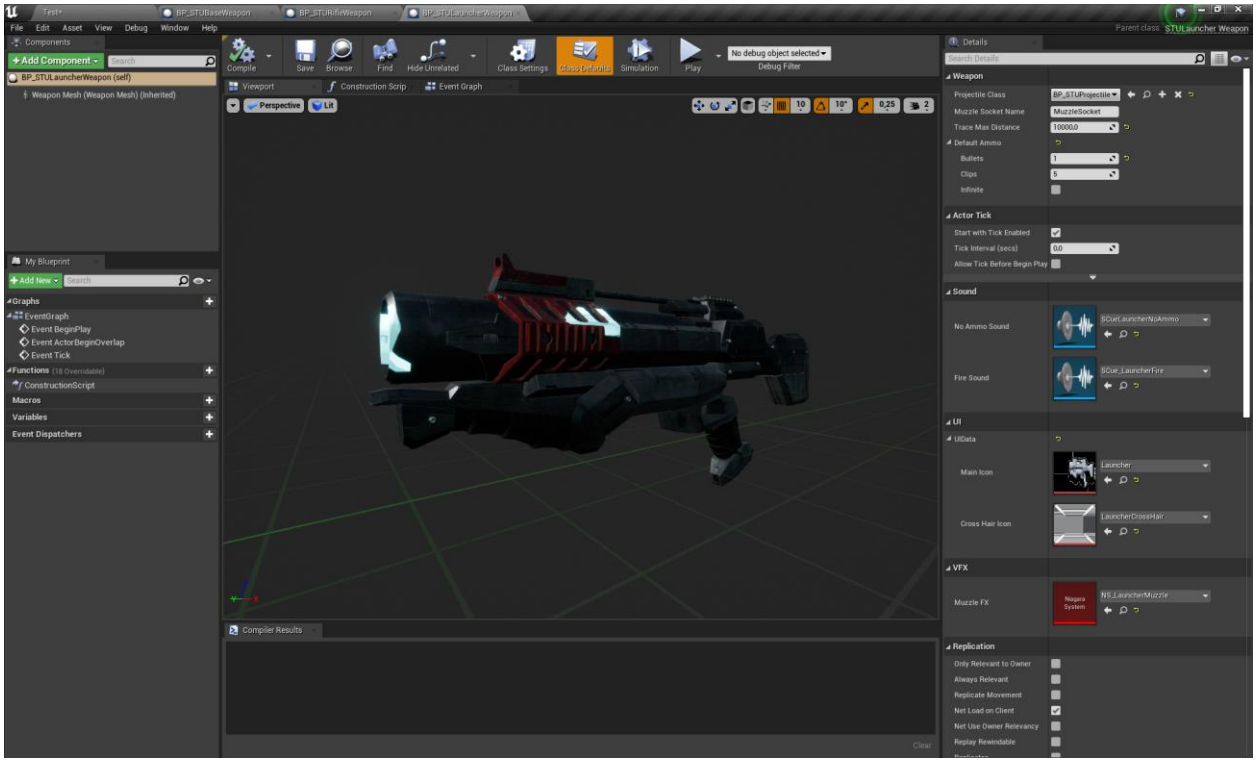


Рисунок 3.6 – Launcher (базука)

Додатково для гвинтівки додано різну кількість пошкодження у потраплянні в певні частини тіла.

3.1.4 Вбудована система інтерфейсу UMG

Unreal Motion Graphics (UMG) — це вбудована система інтерфейсу користувача Unreal Engine, яка використовується для створення таких інтерфейсів, як меню і текстові поля. Інтерфейси користувача, створені за допомогою UMG, складаються з міні-додатків.

В першу чергу при розробці інтерфейсу гравця було розроблено шкалу здоров'я, різні види прицілу для кожної зброї, віджет боєприпасів, заміну інтерфейса у режимі спостерігача. Далі створено UI ефект крові на екрані, при зменшенні здоров'я. Вигляд інтерфейсу користувача (рис. 3.7):

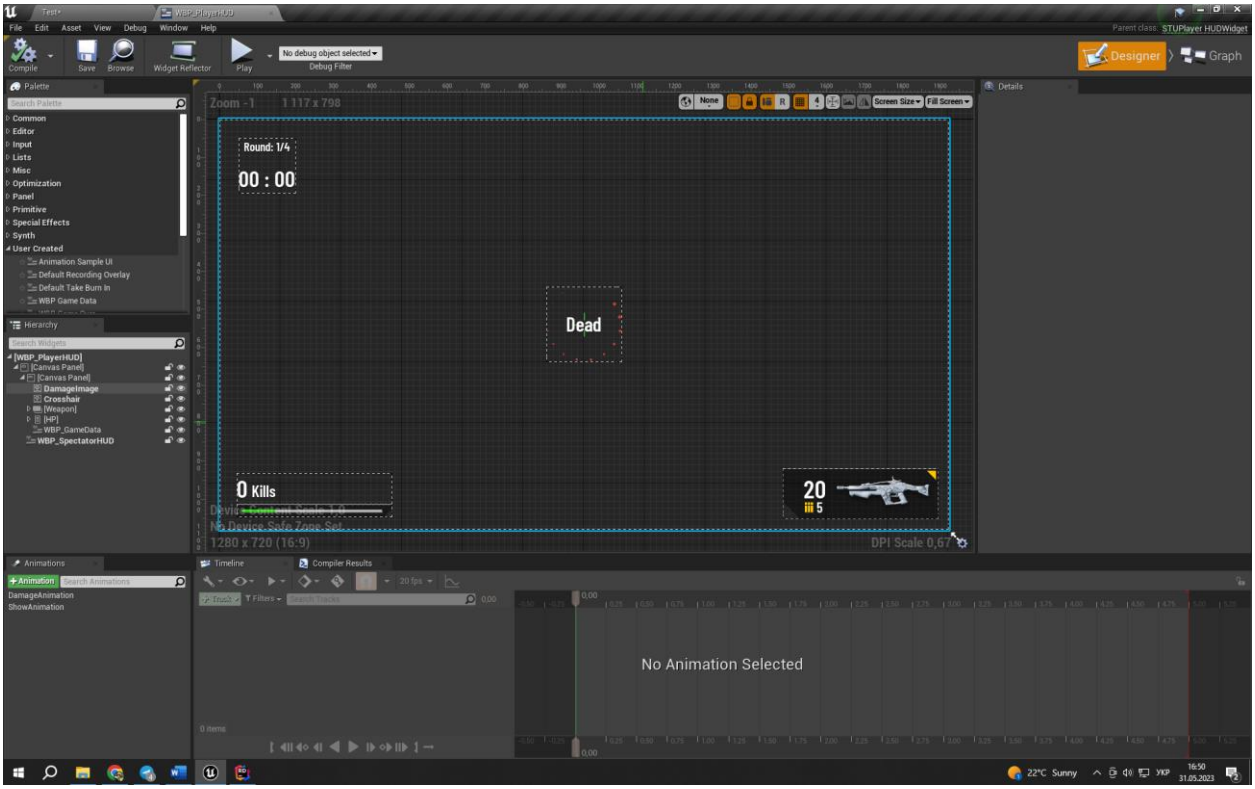


Рисунок 3.7 – Інтерфейс користувача

3.1.5 Pickup об'єкти

Було створено стандартний Pickup об'єкт який реагує на колізію зіткнення з гравцем. Від базового об'єкта створені два види Pickup: здоров'я та додаткових снарядів для Launcher. Реалізовано появу об'єктів через деякий час після підбирання, їх обертання, створено спеціальний матеріал, та Static

Mesh. Запрограмована логіка додавання боєприпасів та здоров'я на певне значення. Вигляд Pickup об'єктів (рис 3.8-3.9):

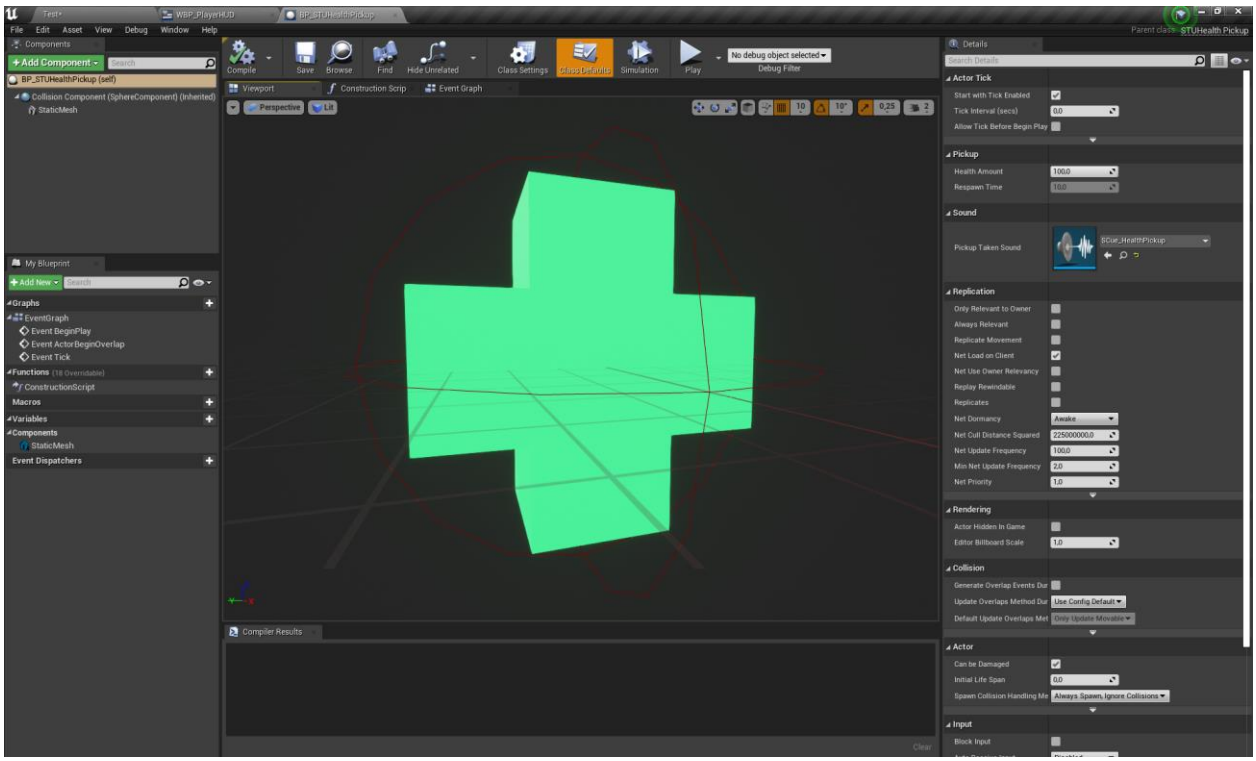


Рисунок 3.8 – Pickup об'єкт здоров'я

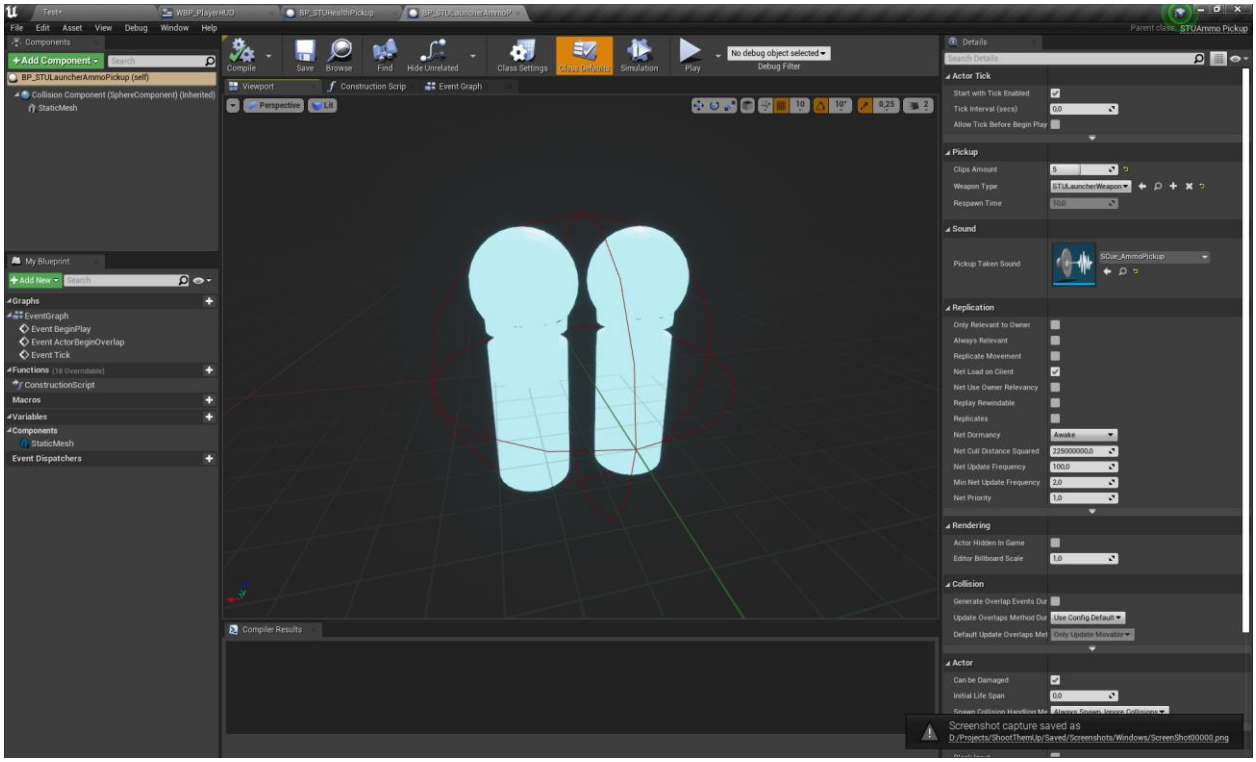


Рисунок 3.9 – Pickup об'єкт снарядів

3.1.6 Візуальні ефекти VFX, Niagara

Niagara – це система частинок візуальних ефектів нового покоління додана у Unreal Engine версії 4.27 та вище.

У цій частині роботи було створено та додано до гри візуальні ефекти пострілів, а саме сліду від потрапляння на текстурах персонажа та об’єктів на рівні, ефектів диму для зброї, шлейфу для звичайного патрона та снаряду базуки, і ефект вибуху. Для сліду від пострілу було додано спеціальні фізичні матеріали накладання. Було замінено анімацію смерті на Ragdoll physics для фізичної симуляції тіла персонажа, для більшої реалістичності. Додано тряску камери при зменшенні здоров’я. Приклад роботи з Niagara системою (рис. 3.10):

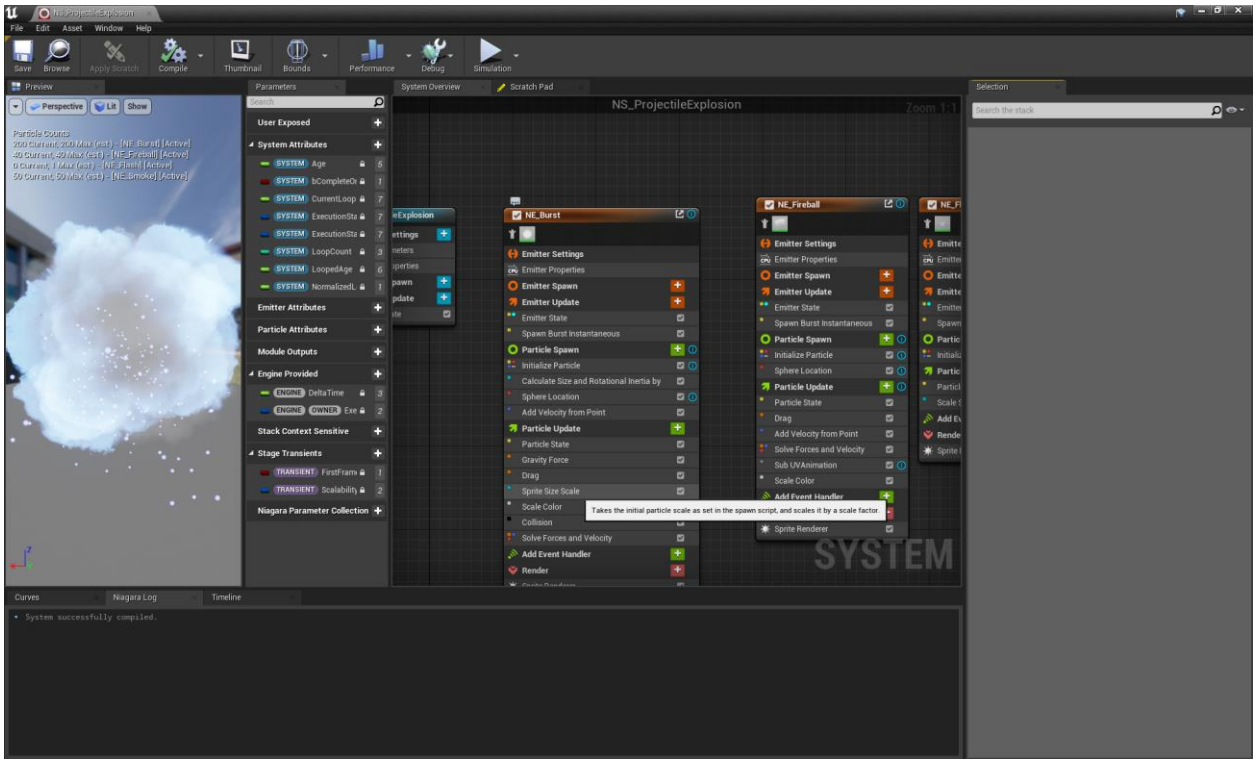


Рисунок 3.10 – Ефект вибуху снаряду базуки

3.1.7 Штучний інтелект AI

Для створення штучного інтелекту противників було створено окремого персонажа, з батьківським класом базового персонажа. Додано на рівень навігаційну сітку для руху та створено дерево поведінки (рис 3.11).

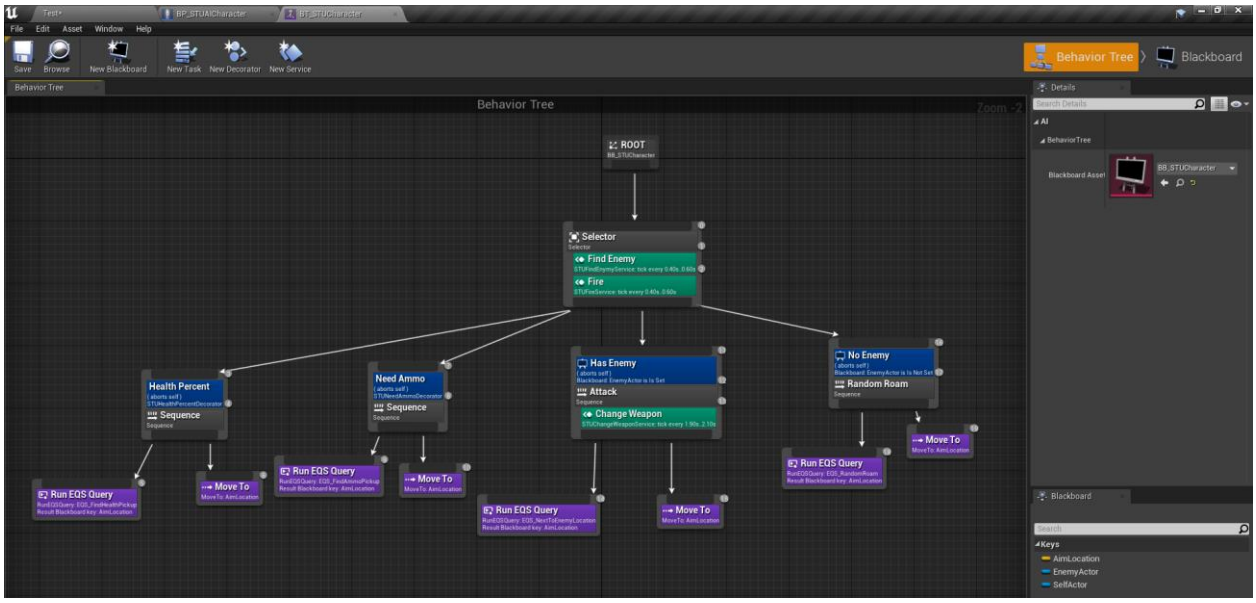


Рисунок 3.11 – Дерево поведінки штучного інтелекту

Для дерева поведінки створені:

- завдання (Tasks), а саме прорахунок локації руху;
- компоненти сприйняття(PerceptionComponent) завдяки яким NPC бачать та реагують на противників та шкоду;
- сервіси находження противника(AI Services), стрільби та зміни зброї;
- системи запитів середовища (EQS), такі як пошук Pickup об’єктів здоров’я та патронів, випадковий рух при баченні противника та його відсутності;
- декоратори для них, які рахують чи потрібно персонажу Pickup об’єкт здоров’я чи снарядів та чи є противник якого можна побачити.

3.1.8 Game Mode

Після створення базового класу Game Mode потрібно створити функції спауну персонажів для коректного початку гри та їх оживлення зі стану наглядача. Також у класі створено функції початку гри та таймер для прорахування часових меж раунду, додатково для цього було створено рестарт

всіх гравців на рівні. Далі створено клас для командної гри, в якому створено функцію розподілу всіх гравців на команди, через це додатково у дереві поведінки NPC гравців, а саме в компоненті сприйняття було прораховано стрільбу тільки по персонажам з іншої команди. Також створено клас для режиму Last Man Standing в якому визначено правила гри, а саме кількість разів переродження дорівнює двом, тобто 3 життя, та закінчення гри при виживанні останнього гравця.

Створено окремий клас для підрахування та зберігання статистики всіх персонажів, для подальшого його використання на кінцевому екрані.

3.1.8 Інтерфейс

Створено нову структуру для зміни стану гри (InProgress, GamePaused, GameOver). Для всього інтерфейсу гри було імпортовано декілька шрифтів. Створено екран паузи з кнопками переходу до меню та перезапуск рівня (рис. 3.12). Також розроблено віджет кінцевого екрану з аналогічними кнопками та виводом кінцевої статистики кожного гравця у таблиці(рис 3.13), окремо для режиму Last Man Standing створено схожий кінцевий екран (рис 3.14). Створено окремій рівень для головного меню, додано на нього джерело світла та дим, накладено поверх віджет з вибором режиму, картинки для режимів було згенеровано за допомогою BlueWillow AI, кнопками старту та виходу з гри (рис 3.15). Паралельно розроблено віджет зміни гучності та додано на окремі екрани меню.

У грі додано UI для відображення поточного стану здоров’я противника при досягненні певної відмітки та при наближенні до цього NPC. До інтерфейсу гравця додано відображення кількості вбивств.

Для всіх віджетів додано анімації затухання та появи.

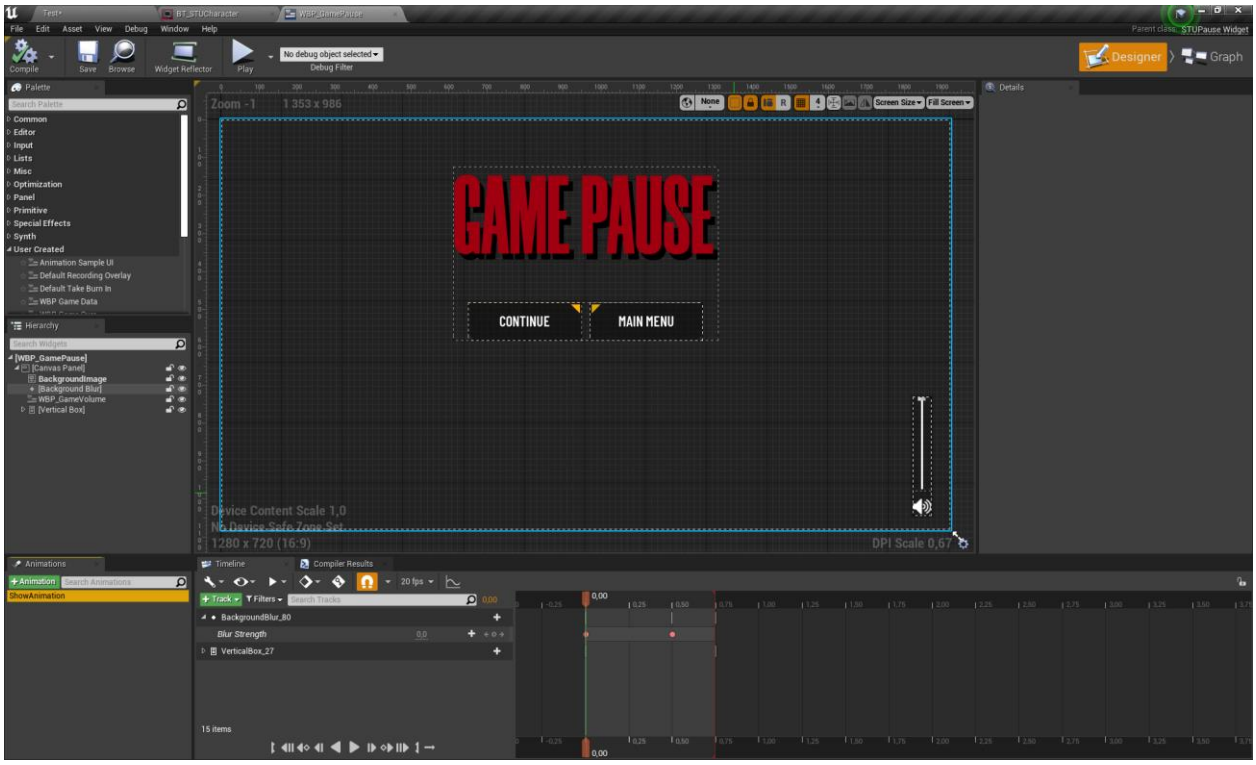


Рисунок 3.12 – Меню паузи

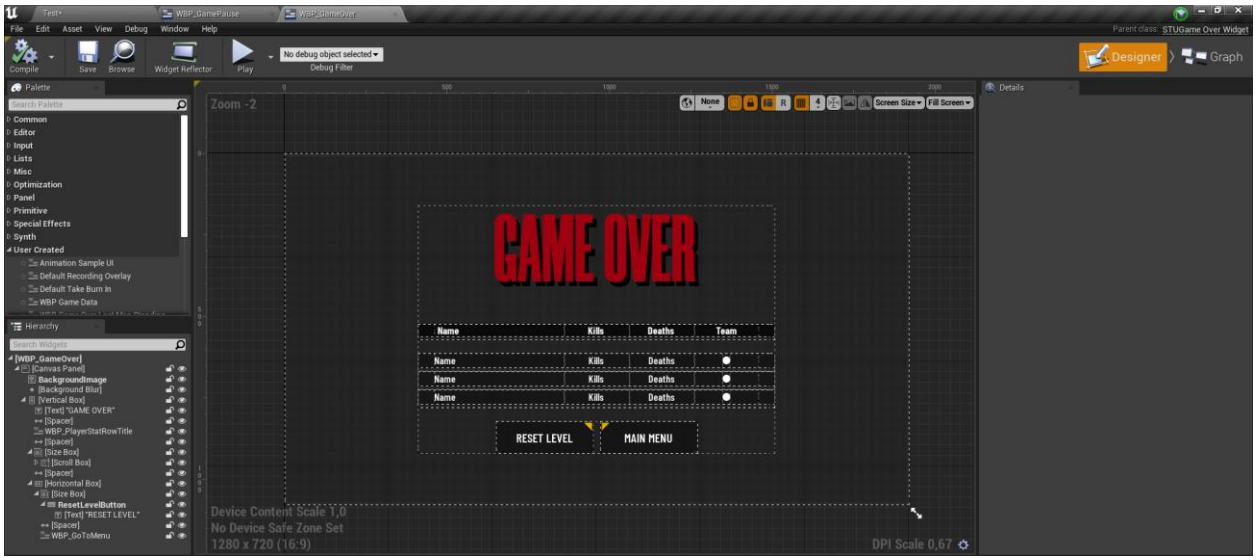


Рисунок 3.13 – Кінцевий екран для Deathmatch та Team Deathmatch

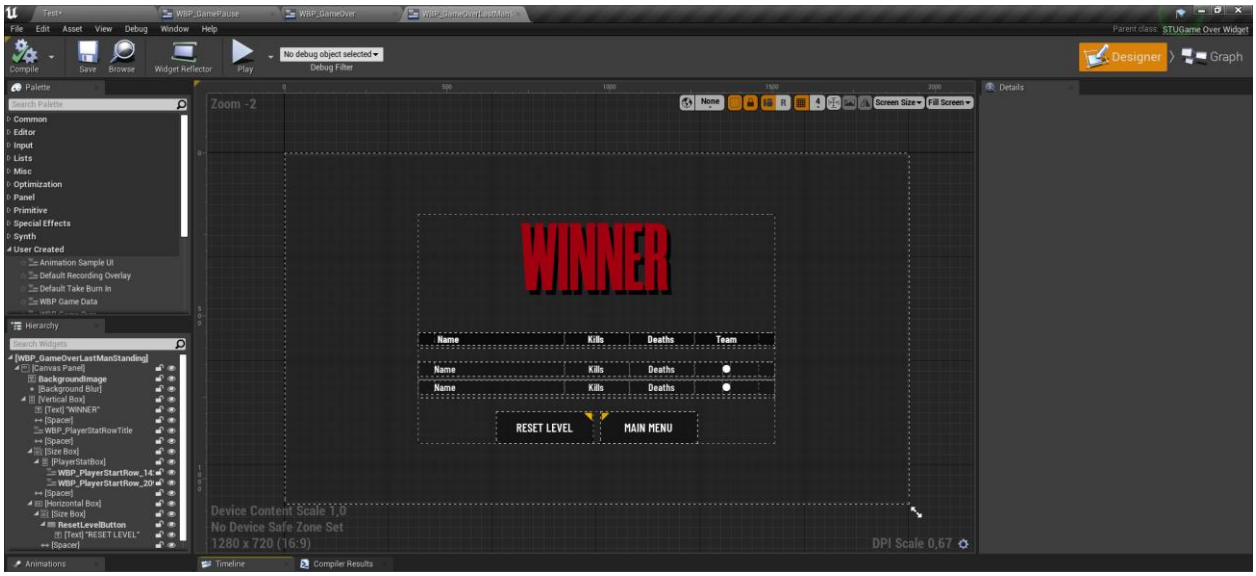


Рисунок 3.14 – Кінцевий екран для Last Man Standing

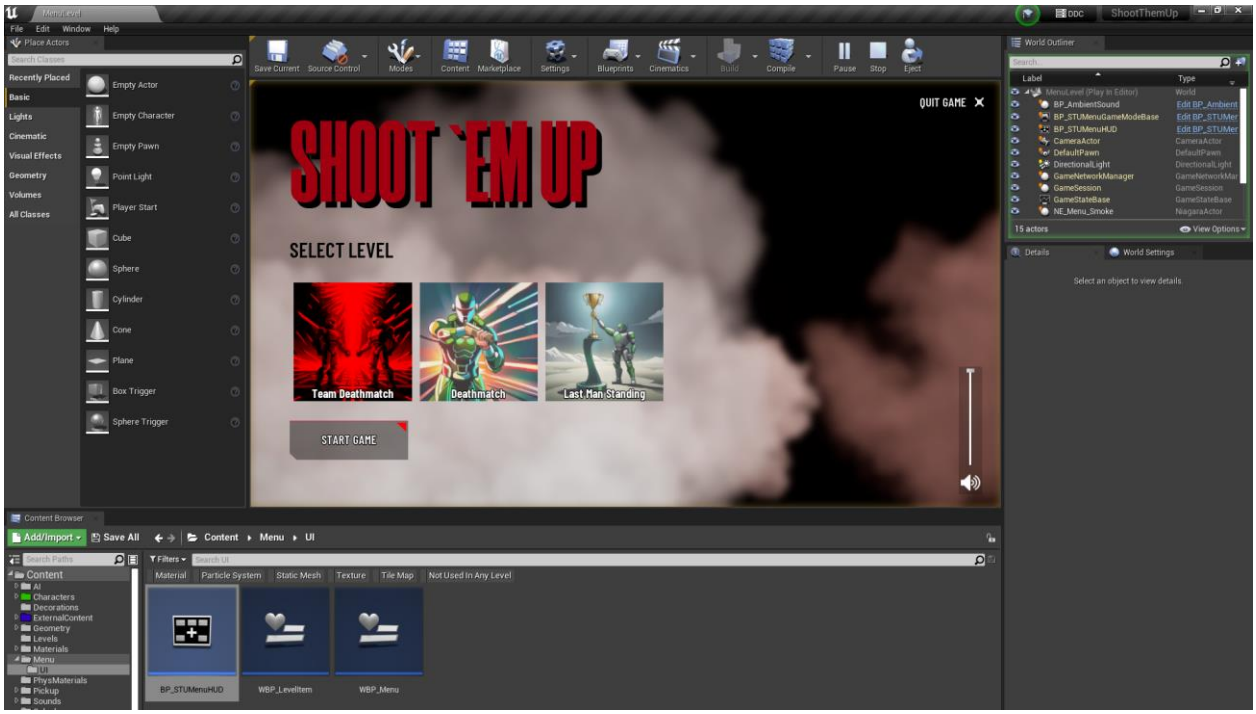


Рисунок 3.15 – Головне меню

3.1.8 Звуки

Імпортовано звуки з шаблону шутера від Epic Games. Додано всі звуки для інтерфейсу гри, фоновий звук для меню та процесу гри, звуки початку та відкривання меню.

Для ігрового процесу гри додані звуки:

- персонажу, такі як кроку, бігу, стрибка та смерті персонажа на лініях анімації;
- зброї, такі як звуки перезарядки, пострілу 2 видів, польоту снаряду та його вибуху, звук потрапляння в персонажа та звичайних об’єктів, звук коли набої вичерпано;
- Pickup акторів, для підняття та звуку використання.

Всі звуки реалізовані за допомогою Sound Attention для просторового звуку.

3.1.8 Рівні

Для всіх режимів створено різні рівні, реалізовані через імпортовані об’єкти та геометричні об’єкти на які накладено текстури з масштабуванням під їх розмір. Для кожного рівня створено різне освітлення, додано точки переродження під кількість персонажів на рівні, розставлено Pickup актори. За допомогою геометричних об’єктів та накладання на них полупрозорої текстури було зроблено стіни рівня через які головний герой не зможе випасти за рівень. На кожний рівень додано навігаційну сітку для штучного інтелекту та сітку для прорахування тіней кожного статичного об’єкту. Вигляд рівнів (рис 3.16-3.18):

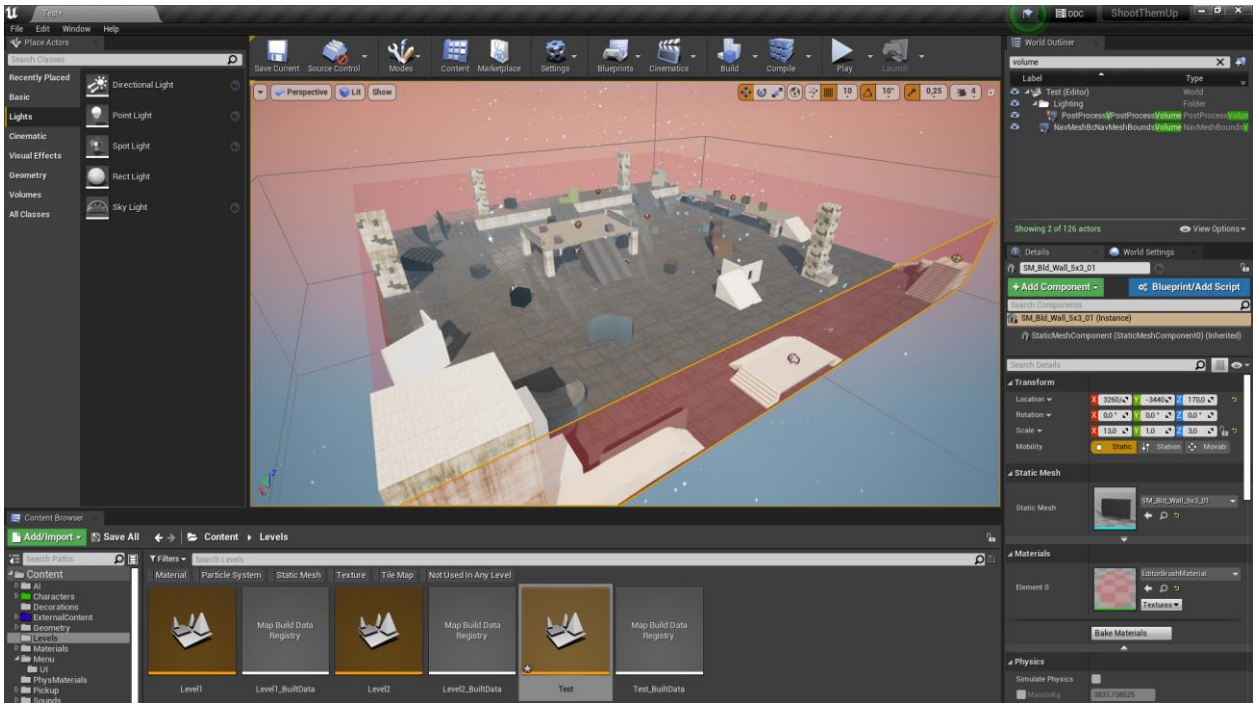


Рисунок 3.16 – Рівень для Team Deathmatch

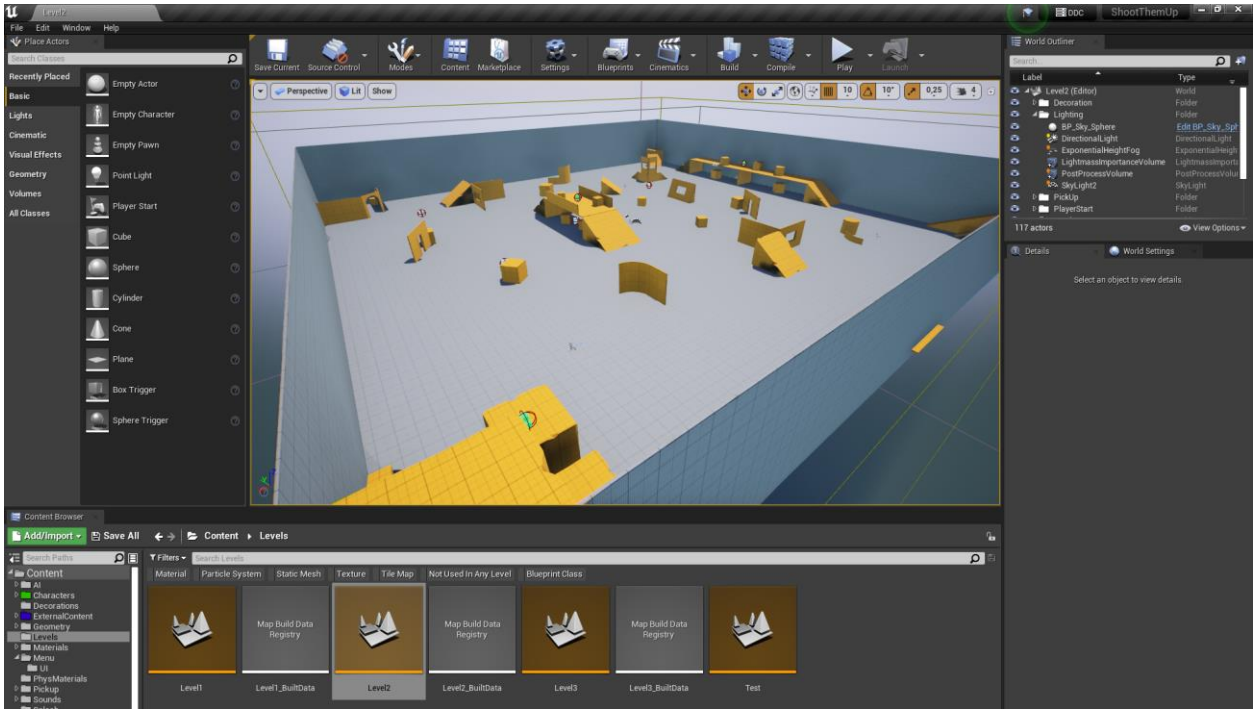


Рисунок 3.17 – Рівень для Deathmatch

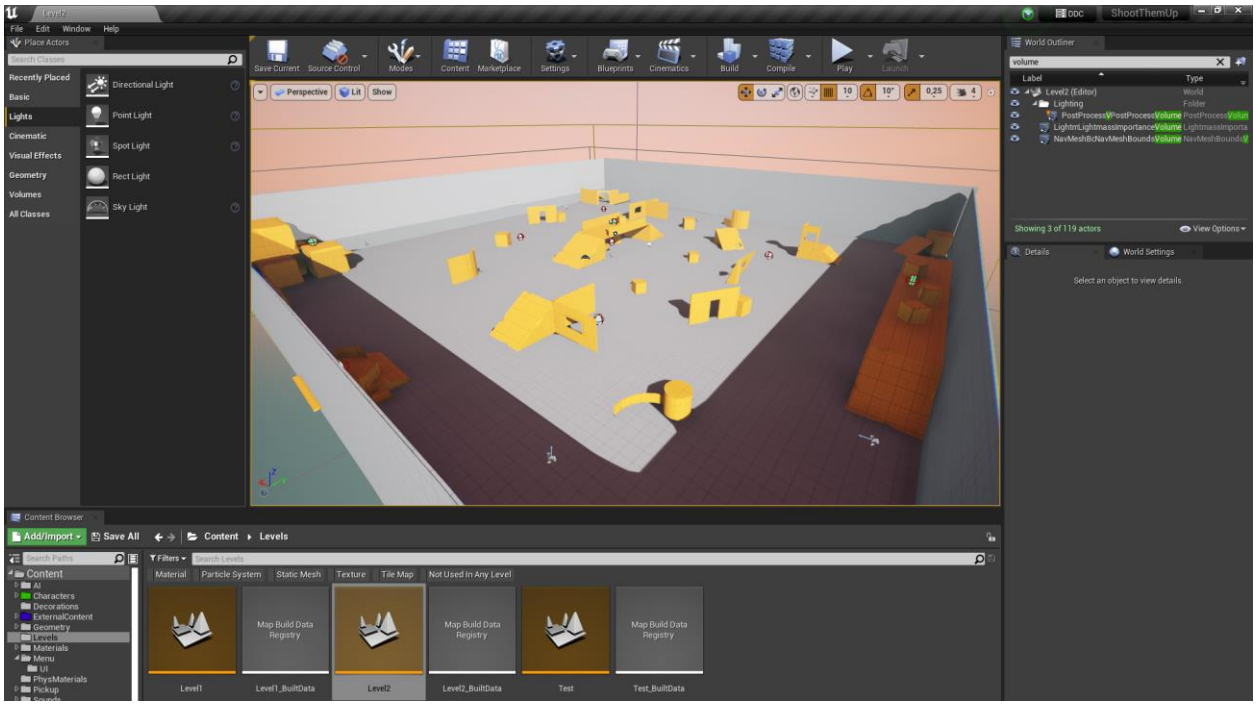


Рисунок 3.18 – Рівень для Last Man Standing

3.2 Тестування та працездатність

Після та під час кожного етапу розробки представлених в програмній реалізації було перевірено та протестовані всі функції які повинні виконуватись. По перше було протестована механіка руху та анімація для руху персонажа, плавний перехід до наступної анімації та підібрані часові обмеження переходу, деякі анімації було зациклено. Для тестування компоненту здоров’я були створені радіальні джерела нанесення шкоди (рис. 3.19). Під час тестування механіки стрільби викликано Debug лінію, для коректного розуміння. Тестування штучного інтелекту проводилось поетапно при доданні нових можливостей поведінки NPC.

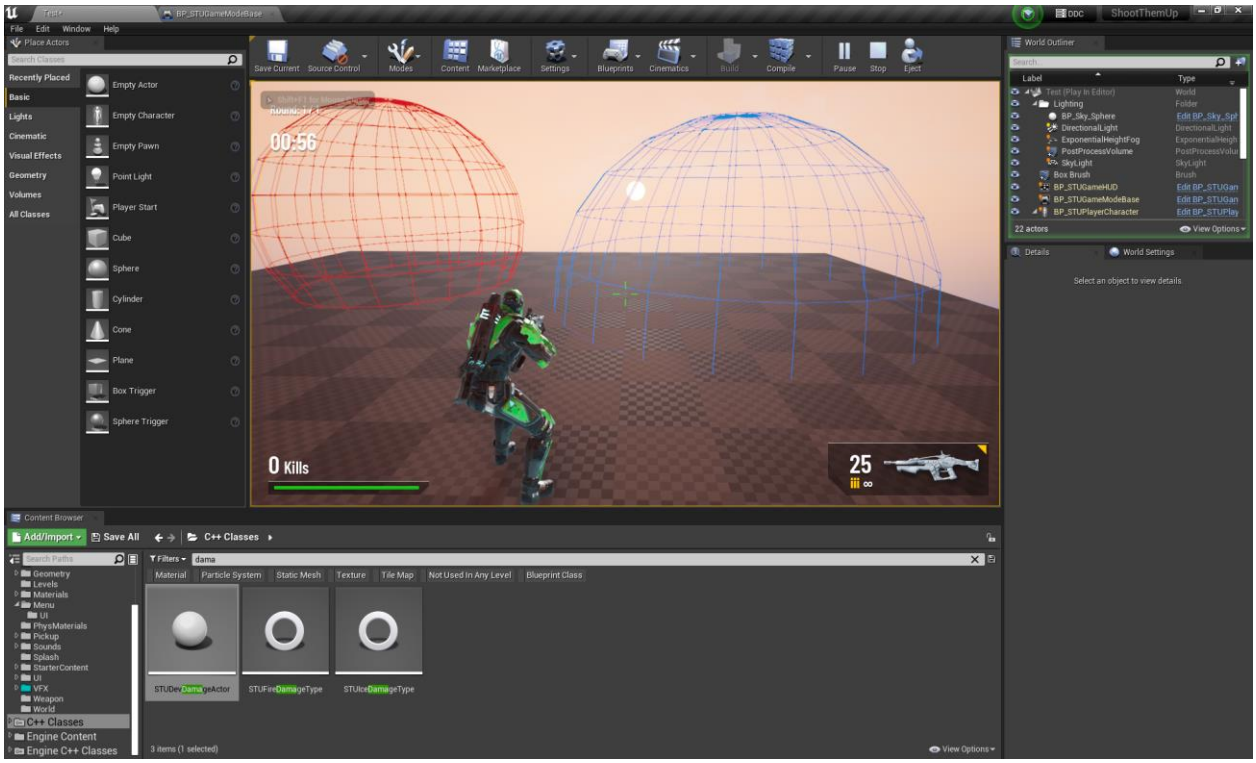


Рисунок 3.19 – Актор для тестування зменшення здоров’я

Проект збирався за допомогою внутрішньої технології пакування для системи Windows (64-bit), конфігурація - транспортування. Після кожного етапу проект пакувався, щоб перевірити останні зміни на працездатність у .exe форматі. При першій збірці у налаштуваннях проекту було встановлено рівень при запуску, на той момент ще не було рівня меню, тому одразу почалась гра. У наступних пакуваннях проєкту додано Splash картинку при загрузці та іконку додатка. Останнім кроком була повна зборка проекту та перевірка всіх можливостей гри.

ВИСНОВКИ

Під час виконання дипломного проекту та для досягнення мети першочергово було проведено огляд статистичних даних популярності та ринку ігрової індустрії. Далі було проведено огляд засобів реалізації для розробки комп'ютерних ігор. Виявлено: ігрова індустрія дуже популярний інформаційний сегмент, ринок постійно збільшується, частка саме комп'ютерних ігор є великою. Тому актуальність створення ігрового додатку жанру шутер для ПК є найпопулярнішим, цей додаток є комерційно корисним.

Наступним кроком був проведений аналіз ігрових додатків схожих за ідеєю та жанром, що дало змогу створити порівняльну таблицю. Виявлено, що необхідно приділити увагу на ігровий процес, вбудований штучний інтелект у рушію, механіки шутера та якість графіки.

Під час проектування та моделювання були розроблені контекстні діаграми та блок-схема процесів, на базі яких створено ігровий додаток жанру шутер. Це дало змогу відслідкувати процеси розробки та можливості гри.

Далі була проведена розробка додатку:

- всіх визначених механік – прискорення персонажа, стрільба, зміна зброї;
- за рахунок вбудованого AI реалізовано обмеження руху персонажа за мапою, реагування на дію стороннього актора, пошук необхідних компонентів;
- створено візуальне та звукове навантаження.

Реалізація гри виконано за допомогою мови програмування C++ та середовищем JetBrains Rider, ігрового рушія Unreal Engine 4 та штучним інтелектом генератора картинок за запитом BlueWillow AI. Тестування проводилось по ходу реалізації механік.

Результатом роботи над дипломним проектом є готовий ігровий додаток «Shoot Them Up» для операційної системи Windows.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. IT-спеціаліст: професія майбутнього. https://business.rayon.in.ua/. [Електронний ресурс] – Доступ до ресурсу: https://business.rayon.in.ua/news/420323-it-spetsialist-profesiya-maybutnogo (дата звернення: 08.02.2023).
2. Де працюють IT спеціалісти? proekt.biz.ua [Електронний ресурс] – Доступ до ресурсу: https://proekt.biz.ua/de-pracyuyut-it-specialisti/ (дата звернення: 08.02.2023).
3. Kultima, A., & Stenros, J. (Eds.). (2018). The Dark Side of Game Play: Controversial Issues in Playful Environments. CRC Press.
4. Salen, K., & Zimmerman, E. (2004). Rules of Play: Game Design Fundamentals. MIT Press.
5. Fullerton, T. (2014). Game Design Workshop: A Playcentric Approach to Creating Innovative Games. CRC Press.
6. Гречко А. В., Захаров Н. В., Фалько М. О. Аналіз динаміки розвитку ринку відеоігор, джерел його фінансування та особливостей монетизації продукції в даній сфері. Ефективна економіка. [Електронний ресурс] – Доступ до ресурсу: http://www.economy.nayka.com.ua/pdf/5_2021/4.pdf (дата звернення: 10.02.2023).
7. Batchelor J. GamesIndustry.biz presents... the year in numbers 2019. gamesindustry. [Електронний ресурс] – Доступ до ресурсу: https://www.gamesindustry.biz/gamesindustry-biz-presents-the-year-in-numbers-2019 (дата звернення: 14.02.2023).
8. Юдин А. Найпопулярніші операційні системи. marketer.ua. [Електронний ресурс] – Доступ до ресурсу: https://marketer.ua/ru/stats-operating-system-2017/ (дата звернення: 16.02.2023).

9. Седих І. А. Індустрія комп'ютерних ігор. Інститут "Центр розвитку". [Електронний ресурс] – Доступ до ресурсу: https://dcenter.hse.ru/data/2020/07/27/1599127653/Индустрия%20компьютерных%20игр-2020.pdf (дата звернення: 19.02.2023).
10. Найкращі шутери на ПК. uaplay.com.ua. [Електронний ресурс] – Доступ до ресурсу: https://uaplay.com.ua/tag/shutery/ (дата звернення: 21.02.2023).
11. Valve Corporation. (2012). Counter-Strike: Global Offensive [Video Game]. Steam.
12. Farina, C., & Tiengo, S. (2017). Play the Game: The Pursuit of Extravagant Aesthetics in Competitive Digital Games. Digital Creativity, 28(2), 121-136.
13. Portnow, J. (2012). Game Design: Counter-Strike and Flow. Extra Credits.
14. J. Ocampo. Mass Effect Review. www.ign.com. [Електронний ресурс] – Доступ до ресурсу: https://www.ign.com/articles/2008/05/27/mass-effect-review (дата звернення: 21.02.2023).
15. Теоретичні відомості про методологію idef0. studfile.net[Електронний ресурс] – Доступ до ресурсу: https://studfile.net/preview/5706328/page:4/ (дата звернення: 12.05.2023).
16. Решетова Н. Е. Чи актуально на сьогодні моделювання в IDEF0? projectimo.ru [Електронний ресурс] – Доступ до ресурсу: http://projectimo.ru/biznes-processy/idef0.html (дата звернення: 15.05.2023)
17. Застосування UML в дипломних роботах dut.edu.ua [Електронний ресурс] – Доступ до ресурсу: https://dut.edu.ua/ua/news-1-626-7758-

zastosuvannya-uml-v-diplomnih-robotah_kafedra-kompyuternih-nauk-ta-informaciynih-tehnologiy (дата звернення: 16.05.2023)

18. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms, 3rd Edition (The MIT Press) 3rd Edition [Електронний ресурс] – Доступ до ресурсу: http://mitpress.mit.edu/9780262046305/introduction-to-algorithms/ (дата звернення: 26.04.2023)
19. Бутхерейт А. В. Основні засади розробки ігор. habr.com. [Електронний ресурс] – Доступ до ресурсу: https://habr.com/ru/articles/188372/ (дата звернення: 24.02.2023).
20. 10 кращих ігрових рушіїв. senfil.net. [Електронний ресурс] – Доступ до ресурсу: https://senfil.net/index.php?newsid=321 (дата звернення: 01.04.2023).
21. Stroustrup, The C++ Programming Language 4th Edition – 2013, 1281p
22. Epic Games. Unreal Engine 4.27 Documentation. docs.unrealengine.com. [Електронний ресурс] – Доступ до ресурсу: https://docs.unrealengine.com/4.27/en-US/ (дата звернення: 11.05.2023).
23. UMG і клавіатура в Unreal. learn.microsoft.com [Електронний ресурс] – Доступ до ресурсу: https://learn.microsoft.com/ru-ru/windows/mixed-reality/develop/unreal/unreal-umg-keyboard (дата звернення: 10.05.2023)

ДОДАТОК А.

МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Технічне завдання на створення ігрового додатку-шутер

«Shoot Them Up»

ПОГОДЖЕНО:

Доцент кафедри комп’ютерних наук

_____ Федотова Н. А.

Студент групи ІТ-91

_____ Сніжко А. Я.

2023

1. Призначення й мета створення ігрового додатку

1.1 Призначення ігрового додатку

Ігровий додаток має реалізувати повне функціонування ігрового процесу та режимів гри, для демонстрації можливостей додатку.

1.2 Мета створення ігрового додатку

Метою даного дослідження є розробка ігрового додатку для розвитку моторики, реакції та розваги гравця з сучасною графікою та ігровим процесом.

1.3 Цільова аудиторія

До цільовою аудиторії гри можна віднести людей віком від 14 років які зацікавленні у веселому, простому та динамічному «геймплеї».

2 Вимоги до ігрового додатку

2.1 Вимоги до ігрового додатку в цілому

2.1.1 Вимоги до структури й функціонування ігрового додатку

Ігровий додаток має бути доступним за допомогою файла завантаження. Ігровий додаток повинен складатися із взаємозалежних розділів із чітко розділеними функціями.

2.1.2 Вимоги до персоналу

Персоналом є технічний розробник, який повинен мати особливі технічні навички з розробки ігрових додатків. Розробники мають підтримувати програмний продукт оновленнями, тому повинні мати навички мови програмування C++ та програми Unreal Engine 4.

2.1.3 Вимоги до збереження інформації

Вся інформація надана в ігровому додатку зберігається тільки у програмному коді, реалізованими мовою програмування C++.

2.1.4 Вимоги до розмежування доступу

Розроблюваний додаток має бути загальнодоступним для завантаження у мережі Інтернет. Права доступу до інформації розмежовані за групами користувачів: технічні розробники та гравці/користувачі. Технічні розробники мають необмежений доступ до даних гри, можливістю їх зміни/видалення/додання.

Користувач ігрового додатку має доступ до гри та всіх її режимів та перегляд псевдоніму користувачів у грі з ним.

Розробники мають доступ до програмного коду, візуального навантаження та всього проекту з можливістю змінювати, додавати чи видаляти їх.

Додаток створюється для використання на платформах Windows 7 (SP1),Windows 8, Windows 10 або Windows 11.

2.2 Структура ігрового додатку

2.2.1 Загальна інформація про структуру ігрового додатку

Структура ігрового додатку – це набір різних рівнів, які пов’язані через головне меню. В головному меню є можливість обрати рівень та режим гри.

2.2.2 Навігація

Основна навігація реалізована в головному меню, меню паузи та екрану завершення гри. На кожному рівні є меню паузи та екран статистики після завершення гри, в яких є кнопка переходу до головного меню гри.

2.2.3 Наповнення ігрового додатку

Наповнення ігрового додатку додатковим контентом можливо лише розробником, безпосередньо через середовище Unreal Engine.

2.2.4 Дизайн та структура додатку

Так, як додаток це шутер, його дизайн виконаний у чорно-червоних кольорах для більш серйозного тону гри. Інтерфейс зроблений у футуристичному стилі, так як персонажі – це кіборги із зброєю майбутнього. Навігація по додатку зроблена максимально просто, щоб новим користувачам було просто зрозуміти додаток.

Рисунок А.1 – Меню гри

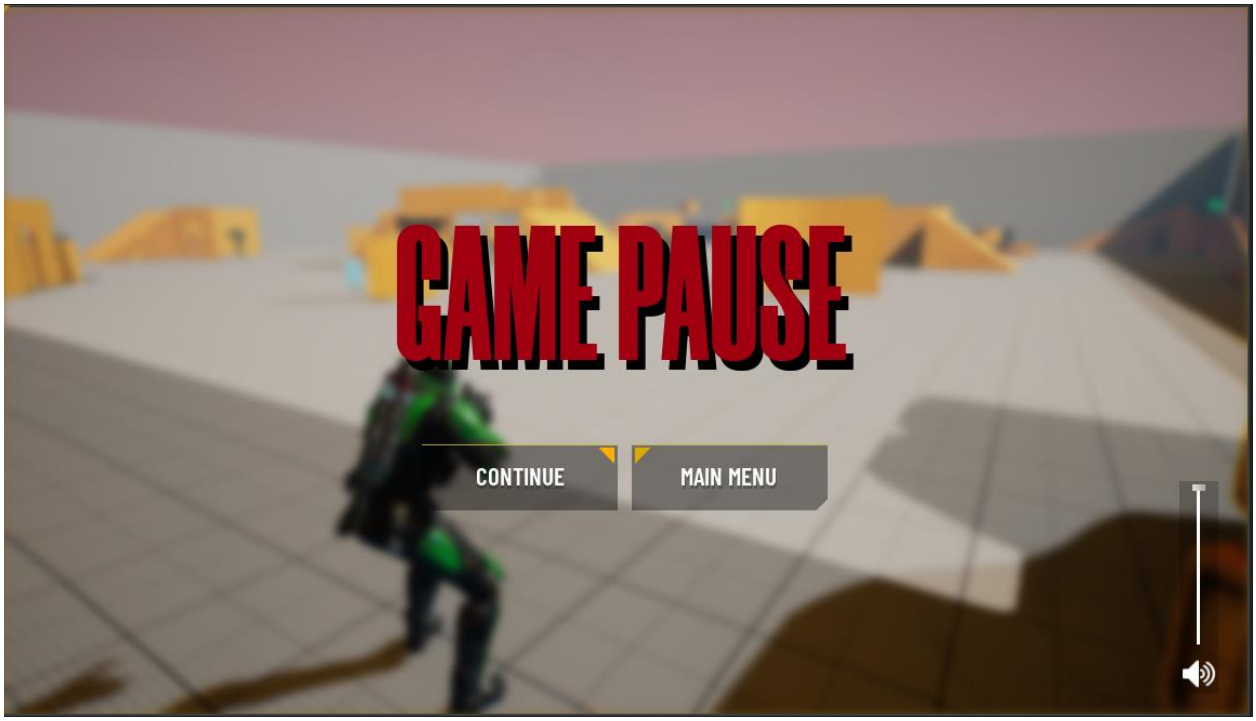


Рисунок А.2 – Меню паузи

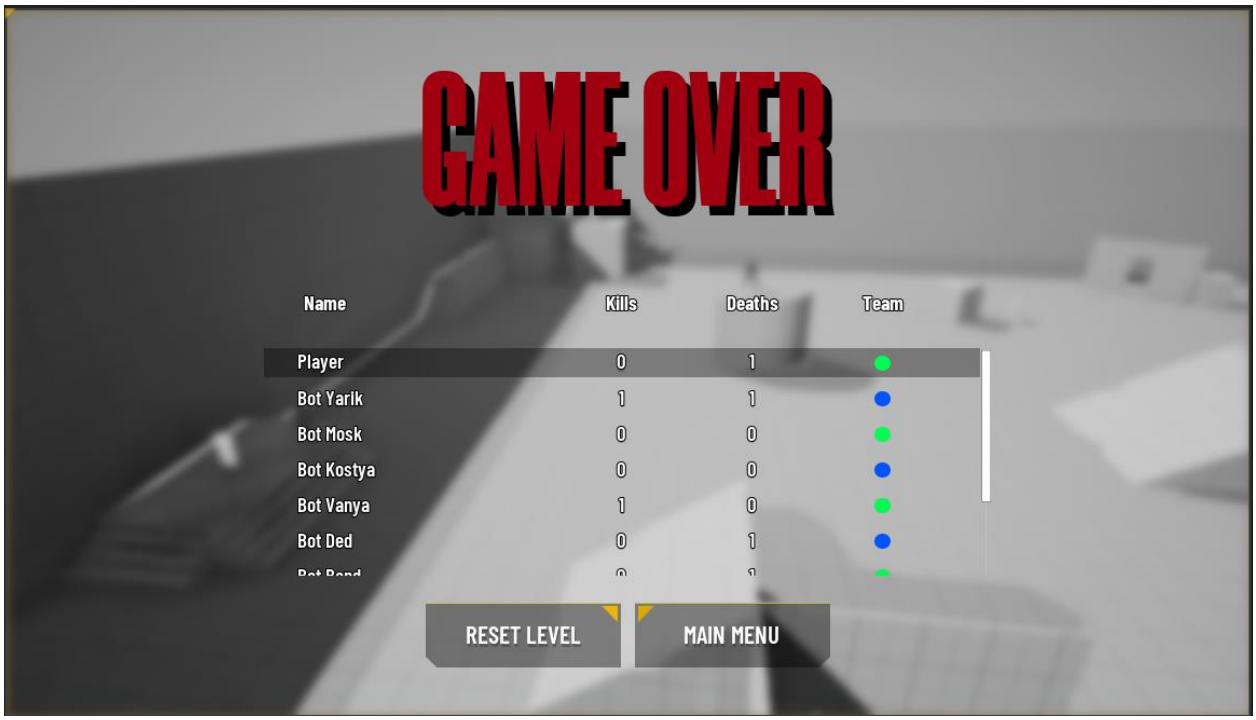


Рисунок А.3 – Екран завершення гри

Рисунок А.4 – Інтерфейс та вигляд гри

2.2.5 Короткий опис головного потоку виконання програми

При запуску додатку ми потрапляємо в головне меню. В головному меню ми можемо обрати рівень та почати грати, зменшити гучність додатку чи вийти з гри. Коли ми починаємо грати, то потрапляємо на рівень та починається ігровий процес. Під час ігрового процесу можемо натиснути клавішу паузи та у меню паузи перезавантажити рівень чи вийти до головного меню. Після завершення часу чи завершення гри бачимо екран завершення гри та також можемо зробити дії як на меню паузи.

2.3 Вимоги до функціонування системи

2.3.1 Потреби користувача

Потреби користувача наведені у таблиці А.1.

Таблиця А.1 – Потреби користувача

ID	Потреби користувача	Джерело
UN-01	Стабільний ігровий процес	Користувач
UN-02	Можливість обирати рівні/рижими гри	Користувач
UN-03	Зміна зброї, лікування та підняття додаткових набоїв	Користувач
UN-04	Наявність звукового супроводу при взаємодії з об’єктами	Користувач
UN-05	Перегляд результатів гри	Користувач
UN-06	Зміна, видалення та додання програмного коду, візуального та звукового навантаження гри	Технічний розробник
UN-07	Зручний та швидкий інтерфейс гри та меню	Користувач

2.3.2 Функціональні вимоги

На основі потреб користувача були визначені такі функціональні вимоги:

- Стабільність гри;
- Наявність зручних вікон меню;
- Можливість зміни режиму у головному меню;
- Можливість зміни зброї, лікування та підняття додаткових боєприпасів;
- Можливість чути взаємодію з об’єктами на рівні та дії інших персонажів на рівні;
- Наявність виводу результатів матчу;

2.3.3 Системні вимоги

Даний розділ визначає, розподіляє та вказує на системні вимоги, визначені розробником. Їх перелік наведений в таблиці А.2.

Таблиця А.2 – Системні вимоги

ID	Системні вимоги	Пріоритет	Опис
SR-01	Стабільний ігровий процес	S	За допомогою оптимізації гри більшій частці користувачів буде краще/плавніше користуватися додатком
SR-02	Перемикач рівнів/рижимів гри	M	Надає доступ до інших розроблений режимів гри
SR-03	Клавіши управління персонажем, меню та зброєю	M	Кожна клавіша відповідає за окрему дію, визначену програмно

Продовження таблиці А.2 – Системні вимоги

SR-04	Звук	S	Відповідає за фонову мелодію, звук при взаємодію свого персонажу чи інших, звуки стрільби та ракет, інтерфейсу
SR-05	Результати гри	S	Відповідає за статистику персонажів на рівні, яка потім виводиться у кінцевому екрані
SR-06	Зміна імені головного персонажа	C	Відповідає за окреме поле у головному меню, де є можливість змінити ім’я персонажа
SR-07	Автоматичне лікування, підняття здоров’я та додаткових набоїв	S	Відповідає за зміну числового значення у компонентах здоров’я та зброї

Умовні позначення в таблиці А.2:

Must have (M) – вимоги, які повинні бути реалізовані в системі;

Should have (S) – вимоги, які мають бути виконані, але вони можуть почекати своєї черги;

Could have (C) – вимоги, які можуть бути реалізовані, але вони не є центральною ціллю проекту.

2.4 Вимоги до видів забезпечення

2.4.1 Вимоги до інформаційного забезпечення

Для створення ігрового додатку використовується:

- Unreal Engine 4.27
- C++
- Environments встановлені у проект

2.4.2 Вимоги до лінгвістичного забезпечення

Ігровий додаток використовує англійську мову.

2.4.3 Вимоги до програмного забезпечення

Для забезпечення стабільної роботи додатку у ПК гравця повинні бути такі мінімальні конфігурації ПК:

ОС: Windows 7 (SP1),Windows 8, Windows 10 або Windows 11

Процесор: Intel Core i7-3770 @3.5 GHz or AMD FX-8350 X8 @ 4 GHz чи потужніший;

Оперативна пам’ять: 8 GB ОЗУ;

Відеокарта: з 2 GB відео пам’яті та підтримкою DirectX 11.

3 Склад і зміст робіт зі створення web-додатку

Послідовність створення ігрового додатку наведена в таблиці А.3.

Таблиця А.3 – Етапи створення ігрового додатку

№	Склад і зміст робіт	Строк розробки
1	Створення проекту, персонажа, додання переміщення та анімацій	3 дні
2	Розробка компоненту здоров’я	2 дні
3	Розробка зброї	8 днів
4	Створення додаткових боєприпасів та здоров’я на рівні	3 дні
5	Створення візуальних ефектів	2 дні
6	Створення штучного інтелекту ворогів та союзників	12 днів
7	Розробка різних ігрових режимів	5 днів
8	Створення інтерфейсу гри	4 дні
9	Наповнення гри звуками	1 день
10	Створення ігрових рівнів	10 днів
11	Beta-тестування	8 днів
12	Alpha-тестування	5 днів
13	Перевірка працездатності	2 дні
14	Написання супровідної документації	2 дні
15	Реліз ігрового додатку	1 день
	Загальна тривалість робіт	68 днів

4 Вимоги до складу й змісту робіт із введення ігрового додатку в експлуатацію

Продукт розробляється ітеративно із урахуванням принципів та технологій уніфікованого процесу розроблення програмного забезпечення. Додаток повинен бути розроблений з використанням мови C++ та архітектури Unreal Engine 4. Повинен мати зручний інтерфейс у меню (головному та меню паузи), цікавий ігровий процес, зручне та швидке створення гри в одиночному режимі.

ДОДАТОК Б.

ПЛАНУВАННЯ РОБІТ

Деталізація мети проекту методом SMART. Продуктом дипломного проекту є ігровий додаток «Shoot Them Up».

Результати деталізації мети даного проекту представлено в таблиці Б.1.

Таблиця Б.1 – Деталізація мети проекту методом SMART

Specific	Розвиток моторики пальців рук, реакції та розваги гравця
Measurable	Користувач може провести матч в цікавому для нього режимі на обраній локації
Achievable	Мета досяжна, є узгоджена тема проекту та навички у роботі з середовищами розробки ігор
Relevant	Додаток дозволить користувачу ознайомитись з грою та її можливостями. Розвиток власних навичок розробника у розробці ігрових додатків
Time-framed	30 травня 2023 року.

Планування змісту структури робіт. Структурна декомпозиція робіт (work breakdown structure, WBS) – це ієрархічна структура робіт, побудована з метою логічного розподілу усіх робіт з виконання проекту і подана у графічному вигляді. Це сукупність декількох рівнів, кожний з яких формується в результаті розподілу роботи поперед-нього рівня на її складові. Елементом найнижчого рівня є група робіт, або так званий робочий пакет (work package). WBS, створення гри «Shoot Them Up» представлений на рис. Б.1

Планування структури організації, для впровадження готового проекту (OBS). Наступним кроком розробки структури проекту є визначення організаційної структури (OBS) проекту. Організаційна структура проекту (OBS) – є графічним відображенням учасників проекту (фізичних та юридичних осіб) та їхніх відповідальних осіб, залучених до реалізації проекту. На верхньому рівні OBS проекту знаходиться керівник та команда управління проектом; на наступному рівні – виконавці. Останнім рівнем OBS-структури є відповідальні особи виконавців. Це не обов'язково повинні бути керівники, а ті співробітники, яким доручено безпосередньо організовувати і відповідати перед виконавцем за виконання конкретного елемента WBS-структури. OBS, Діаграма OBS представлена на рис. Б.2. Список виконавців, що функціонують в проекті знаходиться в табл. Б.2.

Таблиця Б.2 – Виконавці проекту

Роль	**Ім'я**	**Проектна роль**
Розробник	Сніжко А. Я.	Виконує розробку ігрового додатку
Проектувальник	Сніжко А. Я.	Виконує проектування структури ігрового додатку
Тестувальник	Сніжко А. Я.	Відповідає за тестування ігрового додатку
Керівник проекту	Федотова Н. А.	Формує завдання проекту
Менеджер проекту	Сніжко А. Я.	Виконує аналіз даних та відповідає за розподіл ресурсів, стежить за виконанням термінів

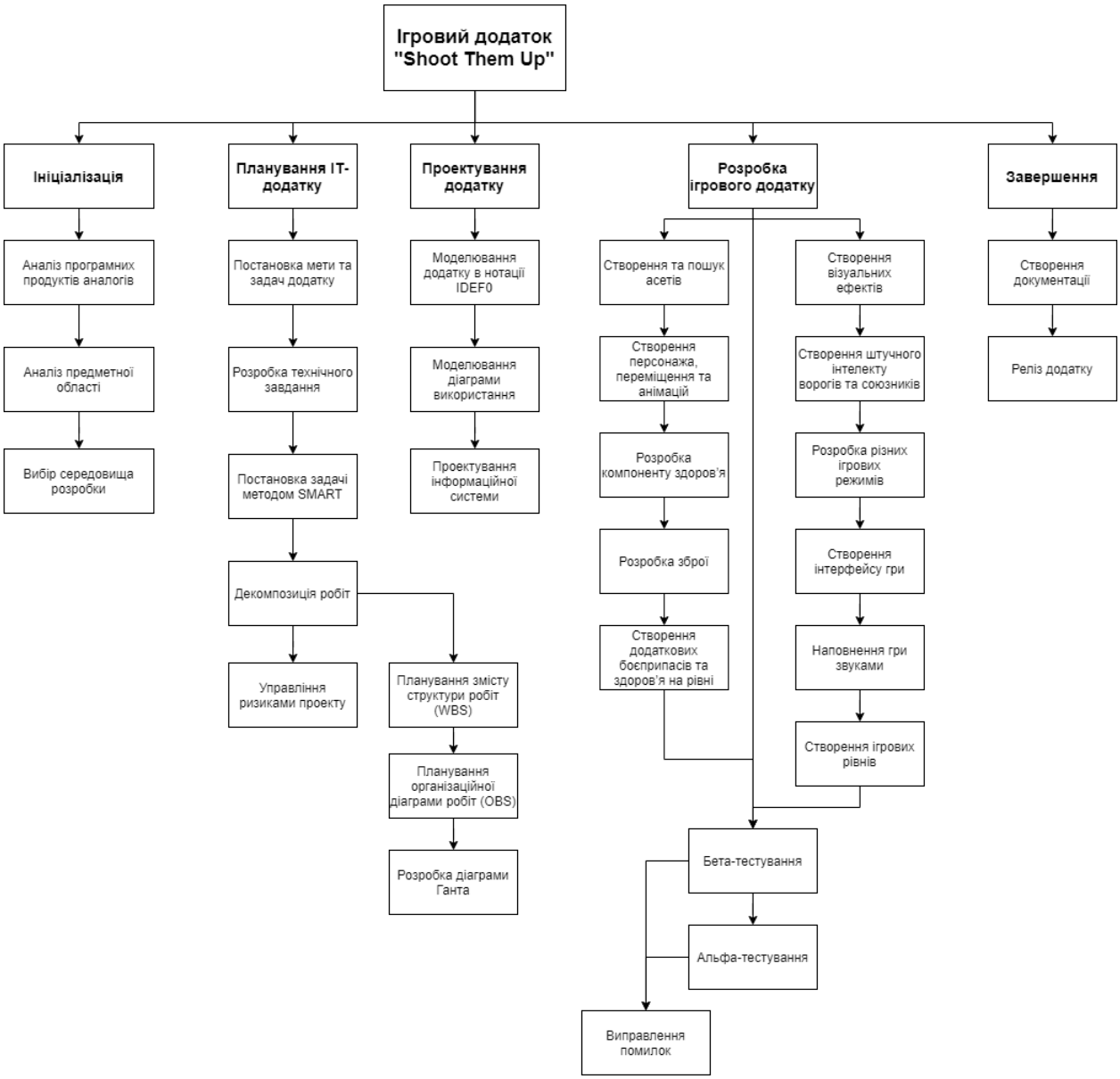


Рисунок Б.1 - WBS-структура проекту

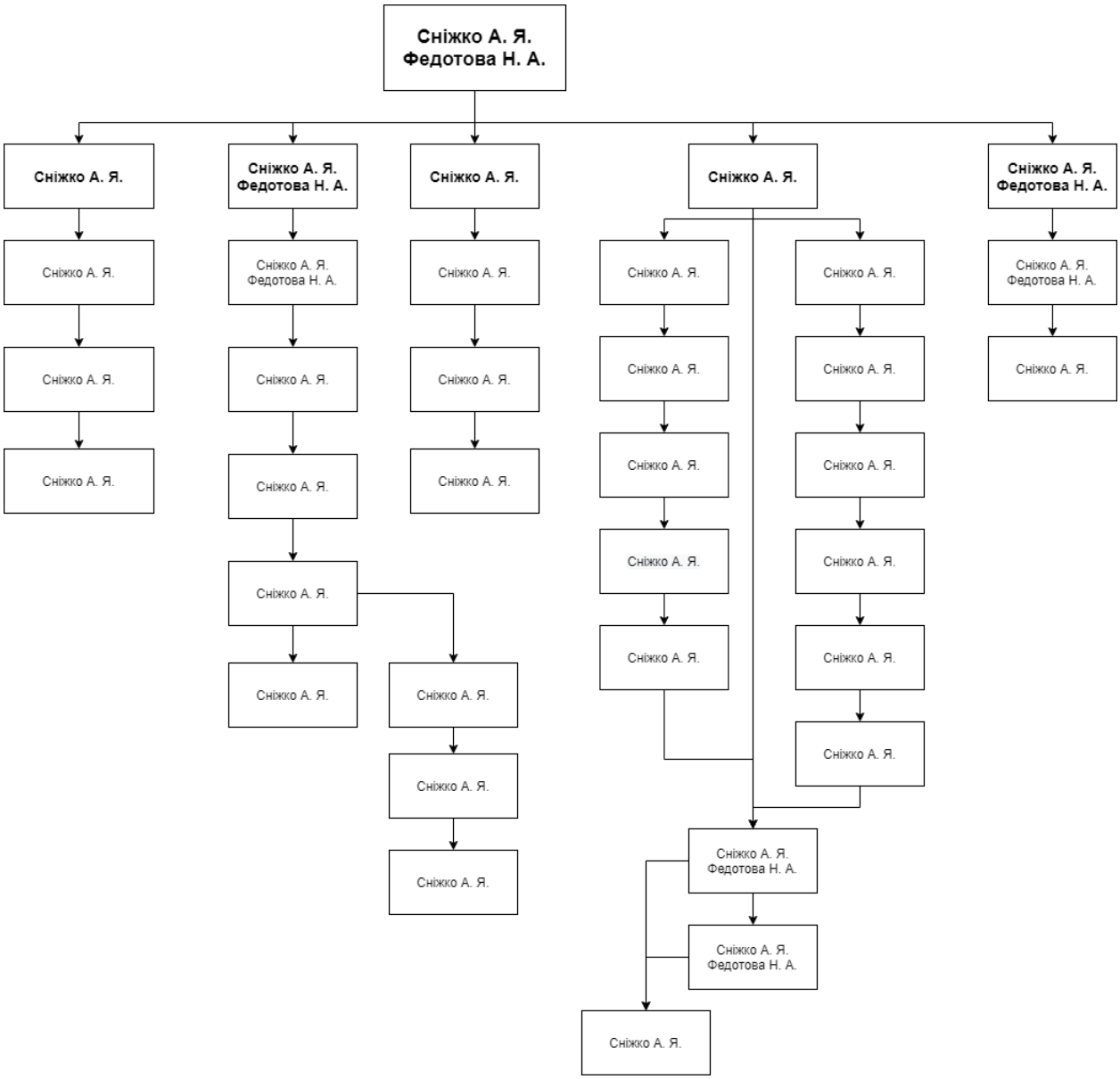


Рисунок Б.2 - OBS-структура проекту

Діаграма Ганта. Діаграма Ганта – це популярний вид діаграми який використовується для планування і контролю виконання проекту. Такий інтерактивний мережевий графік присутній практично у всіх системах управління проектами. На діаграмі відображаються завдання і стадії проекту з урахуванням їх часу виконання. Завдання на діаграмі можуть бути залежними один від одного (наприклад, одна задача може починатися тільки після завершення другого). Крім того, може показуватися відсоток виконання

кожного завдання і відповідальний за її виконання. Діаграма Ганта представлена на рис. Б.3-Б6

Название задачи	Длительнс	Начало	Окончани	Предшественн	Названия ресурсов
Розробка ігрового додатку "Shoot Them Up"	**97 дней**	**Вт 24.01.23**	**Ср 07.06.23**		**Сніжко А. Я.;Федотова Н. А.**
Ініціалізація	**11 дней**	**Вт 24.01.23**	**Вт 07.02.23**		**Сніжко А. Я.**
Аналіз програмних продуктів аналогів	4 дней	Вт 24.01.23	Пт 27.01.23		Сніжко А. Я.
Аналіз предметної області	5 дней	Пн 30.01.23	Пт 03.02.23	3	Сніжко А. Я.
Вибір середовища розробки	2 дней	Пн 06.02.23	Вт 07.02.23	4	Сніжко А. Я.
Планування ІТ-додатку	**6 дней**	**Ср 08.02.23**	**Ср 15.02.23**	**5**	**Сніжко А. Я.**
Постановка мети та задач проекту	2 дней	Ср 08.02.23	Чт 09.02.23		
Розробка технічного завдання	2 дней	Пт 10.02.23	Пн 13.02.23	7	
Моделювання додатку в нотації IDEF0	1 день	Пн 13.02.23	Пн 13.02.23	8	Сніжко А. Я.
Моделювання діаграми використання	1 день	Вт 14.02.23	Вт 14.02.23	9	Сніжко А. Я.
Проектування інформаційної системи	1 день	Ср 15.02.23	Ср 15.02.23	10	Сніжко А. Я.
Розробка ігрового додатку	**71 дней**	**Чт 16.02.23**	**Чт 25.05.23**	**11**	**Сніжко А. Я.**
Створення та пошук асетів	2 дней	Чт 16.02.23	Вт 21.02.23		Сніжко А. Я.
Створення персонажа, переміщення та анімацій	3 дней	Вт 21.02.23	Пт 24.02.23	13	Сніжко А. Я.
Розробка компоненту здоров’я	2 дней	Пт 24.02.23	Вт 28.02.23	14	Сніжко А. Я.
Розробка зброї	8 дней	Вт 28.02.23	Пт 10.03.23	15	Сніжко А. Я.
Створення додаткових боєприпасів та здоров'я на рівні	3 дней	Пт 10.03.23	Ср 15.03.23	16	Сніжко А. Я.

Рисунок Б.3 – Перелік виконаних робіт

Название задачи	Длительнс	Начало	Окончани	Предшественн	Названия ресурсов
Розробка зброї	8 дней	Вт 28.02.23	Пт 10.03.23	15	Сніжко А. Я.
Створення додаткових боєприпасів та здоров'я на рівні	3 дней	Пт 10.03.23	Ср 15.03.23	16	Сніжко А. Я.
Створення візуальних ефектів	2 дней	Ср 15.03.23	Пт 17.03.23	17	Сніжко А. Я.
Створення штучного інтелекту ворогів та союзників	12 дней	Пт 17.03.23	Вт 04.04.23	18	Сніжко А. Я.
Розробка різних ігрових режимів	5 дней	Вт 04.04.23	Вт 11.04.23	19	Сніжко А. Я.
Створення інтерфейсу гри	4 дней	Вт 11.04.23	Пн 17.04.23	20	Сніжко А. Я.
Наповнення гри звуками	1 день	Пн 17.04.23	Вт 18.04.23	21	Сніжко А. Я.
Створення ігрових рівнів	10 дней	Вт 18.04.23	Вт 02.05.23	22	Сніжко А. Я.
Beta-тестування	8 дней	Вт 02.05.23	Пт 12.05.23	23	Сніжко А. Я.
Alpha-тестування	5 дней	Пт 12.05.23	Пт 19.05.23	24	Сніжко А. Я.
Виправлення помилок	4 дней	Пт 19.05.23	Чт 25.05.23	25	Сніжко А. Я.
Завершення	**9 дней**	**Пт 26.05.23**	**Ср 07.06.23**	**26**	**Сніжко А. Я.;Федотова Н. А.**
Створення документації	8 дней	Пт 26.05.23	Вт 06.06.23		Сніжко А. Я.;Федотова Н. А.
Презентація проек	1 день	Ср 07.06.23	Ср 07.06.23	28	Сніжко А. Я.

Рисунок Б.4 – Продовження переліку виконаних робіт

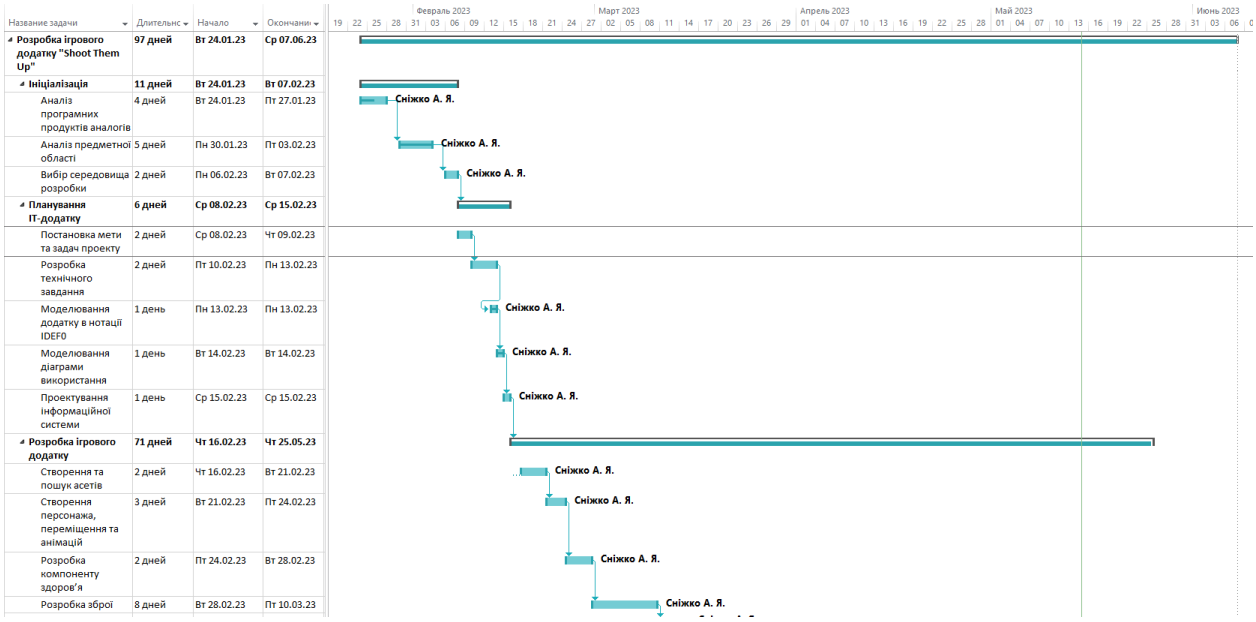
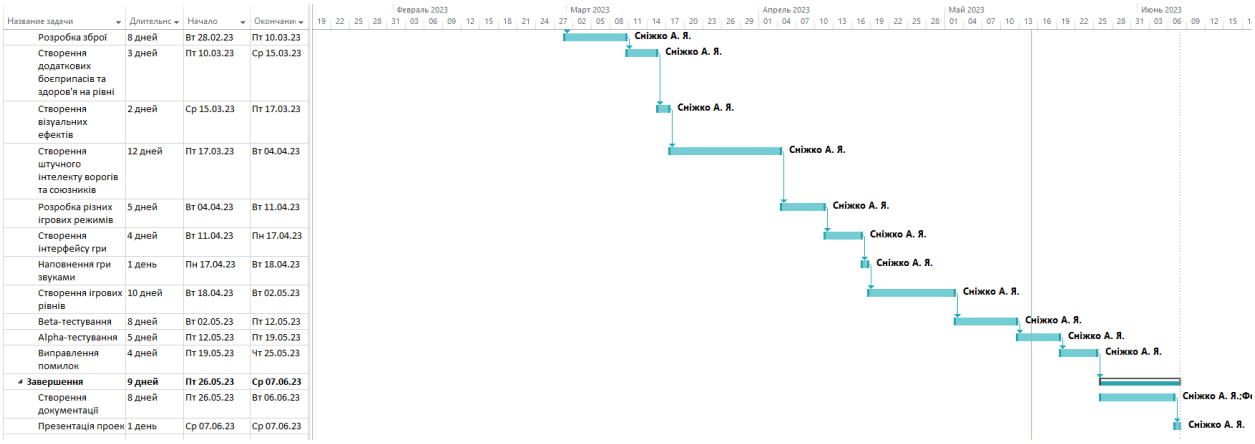


Рисунок Б.5 – Діаграма Ганта



Название задачи	Длительнс	Начало	Окончани
Розробка зброї	8 дней	Вт 28.02.23	Пт 10.03.23
Створення додаткових боєприпасів та здоров'я на рівні	3 дней	Пт 10.03.23	Ср 15.03.23
Створення візуальних ефектів	2 дней	Ср 15.03.23	Пт 17.03.23
Створення штучного інтелекту ворогів та союзників	12 дней	Пт 17.03.23	Вт 04.04.23
Розробка різних ігрових режимів	5 дней	Вт 04.04.23	Вт 11.04.23
Створення інтерфейсу гри	4 дней	Вт 11.04.23	Пн 17.04.23
Наповнення гри звуками	1 день	Пн 17.04.23	Вт 18.04.23
Створення ігрових рівнів	10 дней	Вт 18.04.23	Вт 02.05.23
Beta-тестування	8 дней	Вт 02.05.23	Пт 12.05.23
Alpha-тестування	5 дней	Пт 12.05.23	Пт 19.05.23
Виправлення помилок	4 дней	Пт 19.05.23	Чт 25.05.23
Завершення	**9 дней**	**Пт 26.05.23**	**Ср 07.06.23**
Створення документації	8 дней	Пт 26.05.23	Вт 06.06.23
Презентація проек	1 день	Ср 07.06.23	Ср 07.06.23

Рисунок Б.6 – Продовження діаграми Ганта

Аналіз ризиків. Виконано якісну і кількісну оцінку ризиків роботи. При якісній оцінці визначено проблеми, що потребують швидкого реагування. Така оцінка визначить ступінь важливості і дозволить вибрати спосіб реагування. Кількісна оцінка буде виконана для більш повної ідентифікації ризиків та ступеня їхнього впливу на виконання проекту. Кількісна і якісна оцінка можуть використовуватися окремо або разом, залежно від наявного часу і бюджету, необхідності в кількісній або якісній оцінці. У табл. Б.6 знаходиться класифікація ризиків за показниками ймовірності виникнення ризику та величині втрат.

Далі виконано планування реагування на ризики — це розробка методів і технологій зниження негативного впливу ризиків на проект. Визначено ефективність розробки реагування на проект, визначено чи будуть наслідки впливу ризику на проект позитивними або негативним. Оцінено ризики за показниками, що знаходяться в табл. Б.3. На основі оцінки побудовано матрицю ймовірності виникнення ризиків та впливу ризику, що знаходиться в табл. Б.4.

Таблиця Б.3 − Шкала оцінювання ймовірності виникнення та впливу ризику на виконання проекту

Оцінка	Ймовірність виникнення	Вплив ризику
1	Низька	Низький
2	Середня	Середній
3	Висока	Високий

Таблиця Б.4 - Матриця ймовірності виникнення ризиків та впливу ризику

Вплив ризику / **Ймовірність виконання**	Низький	Середній	Високий
Висока	-	RS_2	-
Середня	RS_7	RS_3, RS_6	-
Низька	-	RS_1, RS_5	RS_4

- зелений колір – прийнятні ризики;
- жовтий колір – виправданні ризики;
- червоний колір – недопустимі ризики.

На підставі отриманого значення індексу ризики класифікують: за рівнем ризику, що знаходиться в табл. Б.4.

Таблиця Б.5 − Шкала оцінювання за рівнем ризику

№	Назва	Межі	Ризики, які входять (номера)
1	Прийнятні	$0{,}005 \leq R \leq 0{,}05$	5, 7
2	Виправдані	$0{,}05 < R \leq 0{,}16$	1, 3, 4, 6
3	Недопустимі	$0{,}16 < R \leq 0{,}72$	2

Таблиця Б.6 – Оцінка ймовірності виникнення, величини витрат та індексу ризику

ID	Статус ризику	Опис ризику	Ймовірність виникнення	Вплив	Ранг ризику	План А	Тип стратегії реагування	План Б
RS_1	Закритий	Непорозуміння між розробниками та замовником	Низька	Середній	0,08	Налагодження відносин. Дотримання ділового етикету спілкування.	Попередження	Аналіз та обговорення проблем
RS_2	Відкритий	Продукти-конкуренти	Висока	Середній	0,24	Створення більших переваг у кінцевого продукта	Попередження	Обговорення з замовниками щодо популяризації продукту
RS_3	Відкритий	Недоліки завдання на розробку	Середня	Середній	0,15	Обговорення всіх видів вимог з замовником. Контроль замовником етапів розробки	Попередження	Окреслення що було зроблено невірно та виправлення помилок
RS_4	Закритий	Низька кваліфікація розробників	Низька	Великий	0,16	Переконуватися у кваліфікації виконавців	Попередження	Найняти інших людей
RS_5	Відкритий	Відпустки/лікарняні працівників	Низька	Середній	0,04	Внесення коректив у графік	Прийняття	
RS_6	Відкритий	Часте внесення змін в ТЗ	Середня	Середній	0,12	Обговорення всіх майбутніх змін та правок.	Попередження	Поступове розроблення всіх правок
RS_7	Закритий	Неоптимальний розподіл часу	Середня	Низький	0,03	Додання робочих годин розробникам	Попередження	Обговорення з замовником щодо додання деякого часу на розробку

ДОДАТОК В.

Програмний код

AI.

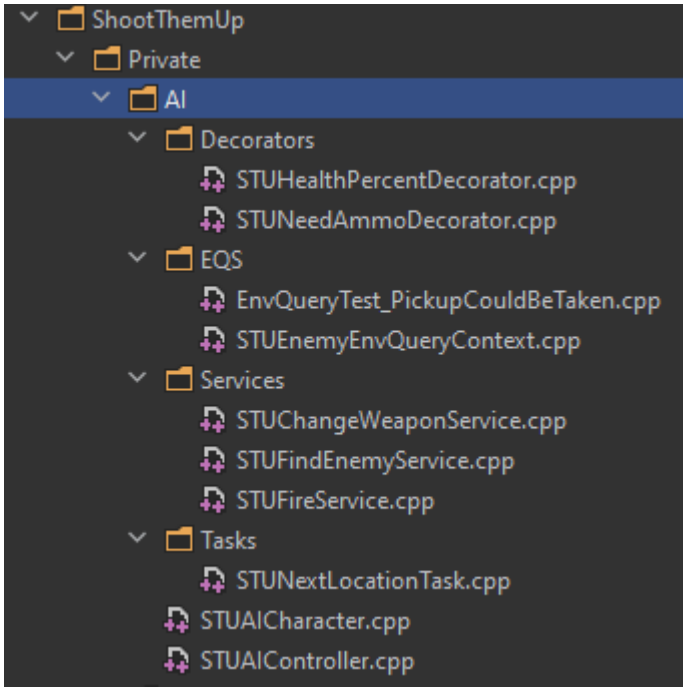


Рисунок В.1 – Файли реалізації штучного інтелекту

STUAICharacter.h:

```
#pragma once

#include "CoreMinimal.h"
#include "BehaviorTree/BehaviorTree.h"
#include "Components/WidgetComponent.h"
#include "Player/STUBaseCharacter.h"
#include "STUAICharacter.generated.h"


UCLASS()
class SHOOTTHEMUP_API ASTUAICharacter : public ASTUBaseCharacter
{
   GENERATED_BODY()

public:
   ASTUAICharacter(const FObjectInitializer& ObjInit);//конструктор класу

   UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="AI")
   UBehaviorTree* BehaviorTreeAsset;

   virtual void Tick(float DeltaSeconds) override;//тік функція

protected:
   UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Components")
   UWidgetComponent* HealthWidgetComponent;//компонент для відображення здоров'я
```

```
   UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="AI")
   float HealthVisibilityDistance = 1000.0f;//дистанція для відображення компоненту

   virtual void BeginPlay() override;
   virtual void OnDeath() override;
   virtual void OnHealthChanged(float Health, float HealthDelta) override;

private:
   void UpdateHealthWidgetVisibility();
};
```

STUAICharacter.cpp:

```
#include "AI/STUAICharacter.h"
#include "STUAIWeaponComponent.h"
#include "STUHealthBarWidget.h"
#include "AI/STUAIController.h"
#include "GameFramework/CharacterMovementComponent.h"

ASTUAICharacter::ASTUAICharacter(const FObjectInitializer& ObjInit):
Super(ObjInit.SetDefaultSubobjectClass<USTUAIWeaponComponent>("WeaponComponent"))
{
   AutoPossessAI = EAutoPossessAI::Disabled;
   AIControllerClass = ASTUAIController::StaticClass();

   bUseControllerRotationYaw = false;
   if(GetCharacterMovement())
   {
      GetCharacterMovement()->bUseControllerDesiredRotation = true;//вмикаємо поворот
      GetCharacterMovement()->RotationRate = FRotator(0.0f, 200.0f, 0.0f);//швидкість повороту
   }
   HealthWidgetComponent = CreateDefaultSubobject<UWidgetComponent>("HealthWidgetComponent");
   HealthWidgetComponent->SetupAttachment(GetRootComponent());//додаю до корня віджет здоров'я
   HealthWidgetComponent->SetWidgetSpace(EWidgetSpace::Screen);
   HealthWidgetComponent->SetDrawAtDesiredSize(true);
}

void ASTUAICharacter::Tick(float DeltaSeconds)
{
   Super::Tick(DeltaSeconds);
   UpdateHealthWidgetVisibility();
}

void ASTUAICharacter::BeginPlay()
{
   Super::BeginPlay();
   check(HealthWidgetComponent);
}

void ASTUAICharacter::OnDeath(){
   Super::OnDeath();

   const auto STUController = Cast<AAIController>(Controller);
   if(STUController && STUController->BrainComponent)
   {
      STUController->BrainComponent->Cleanup();
   }
}

void ASTUAICharacter::OnHealthChanged(float Health, float HealthDelta)
{
   Super::OnHealthChanged(Health, HealthDelta);

   const auto HealthBarWidget = Cast<USTUHealthBarWidget>(HealthWidgetComponent-
>GetUserWidgetObject());
   if(!HealthBarWidget) return;
   HealthBarWidget->SetHealthPercent(HealthComponent->GetHealthPercent());
}

void ASTUAICharacter::UpdateHealthWidgetVisibility()
{
   if(!GetWorld() || !GetWorld()->GetFirstPlayerController() || !GetWorld()-
>GetFirstPlayerController()->GetPawn()) return;
   const auto PlayerLocation = GetWorld()->GetFirstPlayerController()->GetPawn()-
>GetActorLocation();
   const auto Distance = FVector::Distance(PlayerLocation, GetActorLocation());
```

```
   HealthWidgetComponent->SetVisibility(Distance < HealthVisibilityDistance, true);
}
```

STUAIController.h:

```
#pragma once

#include "CoreMinimal.h"
#include "AIController.h"
#include "STUAIPerceptionComponent.h"
#include "STURespawnComponent.h"
#include "STUAIController.generated.h"

/**
 *
 */
UCLASS()
class SHOOTTHEMUP_API ASTUAIController : public AAIController
{
   GENERATED_BODY()

public:
   ASTUAIController();

protected:
   UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category="AI")
   USTUAIPerceptionComponent* STUAIPerceptionComponent;

   UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category="AI")
   USTURespawnComponent* RespawnComponent;

   UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="AI")
   FName FocusOnKeyName = "EnemyActor";

   virtual void OnPossess(APawn* InPawn) override;
   virtual void Tick(float DeltaSeconds) override;

private:
   AActor* GetFocusOnActor() const;
};
```

STUAIController.cpp:

```
#include "AI/STUAIController.h"
#include "AI/STUAICharacter.h"
#include "BehaviorTree/BlackboardComponent.h"

ASTUAIController::ASTUAIController()
{
        STUAIPerceptionComponent =
CreateDefaultSubobject<USTUAIPerceptionComponent>("STUAIPerceptionComponent");
        SetPerceptionComponent(*STUAIPerceptionComponent);
        RespawnComponent = CreateDefaultSubobject<USTURespawnComponent>("RespawnComponent");
        bWantsPlayerState = true;
}

void ASTUAIController::OnPossess(APawn* InPawn)
{
        Super::OnPossess(InPawn);

        const auto STUCharacter = Cast<ASTUAICharacter>(InPawn);
        if(STUCharacter)
        {
                RunBehaviorTree(STUCharacter->BehaviorTreeAsset);
        }
}

void ASTUAIController::Tick(float DeltaSeconds)
{
        Super::Tick(DeltaSeconds);
        const auto AimActor = GetFocusOnActor();
        SetFocus(AimActor);
}

AActor* ASTUAIController::GetFocusOnActor() const
{
        if(!GetBlackboardComponent()) return nullptr;
```

```
        return Cast<AActor>(GetBlackboardComponent()->GetValueAsObject(FocusOnKeyName));
}
```

Декоратори:

STUHealthPercentDecorator.h:

```
#pragma once

#include "CoreMinimal.h"
#include "BehaviorTree/BTDecorator.h"
#include "STUHealthPercentDecorator.generated.h"

/**
 *
 */
UCLASS()
class SHOOTTHEMUP_API USTUHealthPercentDecorator : public UBTDecorator
{
        GENERATED_BODY()

public:
        USTUHealthPercentDecorator();

protected:
        virtual bool CalculateRawConditionValue(UBehaviorTreeComponent& OwnerComp, uint8*
NodeMemory) const override;

        UPROPERTY(EditAnywhere, BlueprintReadWrite,Category="AI")
        float HealthPercent = 0.6f;
};
```

STUHealthPercentDecorator.cpp:

```
#include "AI/Decorators/STUHealthPercentDecorator.h"

#include "AIController.h"
#include "STUHealthComponent.h"
#include "STUUtils.h"

USTUHealthPercentDecorator::USTUHealthPercentDecorator()
{
        NodeName = "Health Percent";
}

bool USTUHealthPercentDecorator::CalculateRawConditionValue(UBehaviorTreeComponent& OwnerComp,
uint8* NodeMemory) const
{
        const auto Controller = OwnerComp.GetAIOwner();
        if(!Controller) return false;

        const auto HealthComponent =
STUUtils::GetSTUPlayerComponent<USTUHealthComponent>(Controller->GetPawn());
        if(!HealthComponent || HealthComponent->IsDead()) return false;
        return HealthComponent->GetHealthPercent() <= HealthPercent;
        //return Super::CalculateRawConditionValue(OwnerComp, NodeMemory);
}
```

STUNeedAmmoDecorator.h:

```
#pragma once

#include "CoreMinimal.h"
#include "STUBaseWeapon.h"
#include "BehaviorTree/BTDecorator.h"
#include "STUNeedAmmoDecorator.generated.h"

/**
 *
 */
UCLASS()
```

```
class SHOOTTHEMUP_API USTUNeedAmmoDecorator : public UBTDecorator
{
	GENERATED_BODY()

public:
	USTUNeedAmmoDecorator();

protected:
	virtual bool CalculateRawConditionValue(UBehaviorTreeComponent& OwnerComp, uint8*
NodeMemory) const override;

	UPROPERTY(EditAnywhere, BlueprintReadWrite,Category="AI")
	TSubclassOf<ASTUBaseWeapon> WeaponType;
};
```

STUNeedAmmoDecorator.cpp:

```
USTUNeedAmmoDecorator::USTUNeedAmmoDecorator()
{
	NodeName = "Need Ammo";
}

bool USTUNeedAmmoDecorator::CalculateRawConditionValue(UBehaviorTreeComponent& OwnerComp, uint8*
NodeMemory) const
{
	const auto Controller = OwnerComp.GetAIOwner();
	if(!Controller) return false;
	const auto WeaponComponent =
STUUtils::GetSTUPlayerComponent<USTUWeaponComponent>(Controller->GetPawn());
	if(!WeaponComponent) return false;
	return WeaponComponent->NeedAmmo(WeaponType);
	//return Super::CalculateRawConditionValue(OwnerComp, NodeMemory);
}
```

EQS:

EnvQueryTest_PickupCouldBeTaken.h:

```
#pragma once

#include "CoreMinimal.h"
#include "EnvironmentQuery/EnvQueryTest.h"
#include "EnvQueryTest_PickupCouldBeTaken.generated.h"

/**
 *
 */
UCLASS()
class SHOOTTHEMUP_API UEnvQueryTest_PickupCouldBeTaken : public UEnvQueryTest
{
	GENERATED_BODY()

public:
	UEnvQueryTest_PickupCouldBeTaken(const FObjectInitializer& ObjectInitializer);

	virtual void RunTest(FEnvQueryInstance& QueryInstance) const override;
};
```

EnvQueryTest_PickupCouldBeTaken.cpp:

```
#include "AI/EQS/EnvQueryTest_PickupCouldBeTaken.h"
#include "EnvironmentQuery/Items/EnvQueryItemType_ActorBase.h"
#include "Pickups/STUBasePickup.h"

UEnvQueryTest_PickupCouldBeTaken::UEnvQueryTest_PickupCouldBeTaken(const FObjectInitializer&
ObjectInitializer) : Super(ObjectInitializer)
{
	Cost = EEnvTestCost::Low;
	ValidItemType = UEnvQueryItemType_ActorBase::StaticClass();
	SetWorkOnFloatValues(false);
}
```

```
void UEnvQueryTest_PickupCouldBeTaken::RunTest(FEnvQueryInstance& QueryInstance) const
{
	const auto DataOwner = QueryInstance.Owner.Get();
	BoolValue.BindData(DataOwner, QueryInstance.QueryID);
	const bool WantsBeTakable = BoolValue.GetValue();
	for (FEnvQueryInstance::ItemIterator It(this, QueryInstance); It; ++It)
	{
		const auto ItemActor = GetItemActor(QueryInstance, It.GetIndex());
		const auto PickupActor = Cast<ASTUBasePickup>(ItemActor);
		if(!PickupActor) continue;

		const auto CouldBeTaken = PickupActor->CouldBeTaken();
		It.SetScore(TestPurpose, FilterType, CouldBeTaken, WantsBeTakable);
	}
	//Super::RunTest(QueryInstance);
}
```

STUEnemyEnvQueryContext.h:

```
#pragma once

#include "CoreMinimal.h"
#include "EnvironmentQuery/EnvQueryContext.h"
#include "STUEnemyEnvQueryContext.generated.h"

/**
 *
 */
UCLASS()
class SHOOTTHEMUP_API USTUEnemyEnvQueryContext : public UEnvQueryContext
{
	GENERATED_BODY()

public:
	virtual void ProvideContext(FEnvQueryInstance& QueryInstance, FEnvQueryContextData&
ContextData) const override;

protected:
	UPROPERTY(EditAnywhere, BlueprintReadWrite,Category="AI")
	FName EnemyActorKeyName = "EnemyActor";
};
```

STUEnemyEnvQueryContext.cpp:

```
#include "AI/EQS/STUEnemyEnvQueryContext.h"
#include "BehaviorTree/BlackboardComponent.h"
#include "Blueprint/AIBlueprintHelperLibrary.h"
#include "EnvironmentQuery/EnvQuery.h"
#include "EnvironmentQuery/EnvQueryTypes.h"
#include "EnvironmentQuery/Items/EnvQueryItemType_Actor.h"
#include "GameFramework/Character.h"


void USTUEnemyEnvQueryContext::ProvideContext(FEnvQueryInstance& QueryInstance,
FEnvQueryContextData& ContextData) const
{
	const auto QueryOwner = Cast<AActor>(QueryInstance.Owner.Get());
	const auto Blackboard = UAIBlueprintHelperLibrary::GetBlackboard(QueryOwner);
	if(!Blackboard) return;
	const auto ContextActor = Blackboard->GetValueAsObject(EnemyActorKeyName);
	UEnvQueryItemType_Actor::SetContextHelper(ContextData, Cast<AActor>(ContextActor));
	//Super::ProvideContext(QueryInstance, ContextData);
}
```

Сервіси:

STUChangeWeaponService.h:

```
#pragma once

#include "CoreMinimal.h"
#include "BehaviorTree/BTService.h"
```

```
#include "STUChangeWeaponService.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUChangeWeaponService : public UBTService
{
	GENERATED_BODY()

public:
	USTUChangeWeaponService();

protected:
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="AI", meta=(ClampMin = "0.0",
ClampMax = "1.0"))
	float Probability = 0.5f;

	virtual void TickNode(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory, float
DeltaSeconds) override;
};
```

STUChangeWeaponService.cpp:

```
#include "AI/Services/STUChangeWeaponService.h"
#include "AIController.h"
#include "STUUtils.h"
#include "STUWeaponComponent.h"

USTUChangeWeaponService::USTUChangeWeaponService()
{
	NodeName = "Change Weapon";
}

void USTUChangeWeaponService::TickNode(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory,
float DeltaSeconds)
{
	const auto Controller = OwnerComp.GetAIOwner();
	if(Controller)
	{
		const auto WeaponComponent =
STUUtils::GetSTUPlayerComponent<USTUWeaponComponent>(Controller->GetPawn());
		if(WeaponComponent && Probability > 0 && FMath::FRand() <= Probability)
		{
			WeaponComponent->NextWeapon();
		}
	}
	Super::TickNode(OwnerComp, NodeMemory, DeltaSeconds);
}
```

STUFindEnemyService.h:

```
#pragma once

#include "CoreMinimal.h"
#include "BehaviorTree/BTService.h"
#include "STUFindEnemyService.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUFindEnemyService : public UBTService
{
	GENERATED_BODY()

public:
	USTUFindEnemyService();

protected:
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="AI")
	FBlackboardKeySelector EnemyActorKey;

	virtual void TickNode(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory, float
DeltaSeconds) override;
};
```

STUFindEnemyService.cpp:

```
#include "AI/Services/STUFindEnemyService.h"

#include "AIController.h"
#include "STUAIPerceptionComponent.h"
```

```
#include "STUUtils.h"
#include "BehaviorTree/BlackboardComponent.h"

USTUFindEnemyService::USTUFindEnemyService()
{
	NodeName = "Find Enemy";
}

void USTUFindEnemyService::TickNode(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory, float
DeltaSeconds)
{
	const auto Blackboard = OwnerComp.GetBlackboardComponent();
	if(Blackboard)
	{
		const auto Controller = OwnerComp.GetAIOwner();
		const auto PerceptionController =
STUUtils::GetSTUPlayerComponent<USTUAIPerceptionComponent>(Controller);
		if(PerceptionController)
		{
			Blackboard->SetValueAsObject(EnemyActorKey.SelectedKeyName,
PerceptionController->GetClosesEnemy());
		}
	}
	Super::TickNode(OwnerComp, NodeMemory, DeltaSeconds);
}
```

STUFireService.h:

```
#pragma once

#include "CoreMinimal.h"
#include "BehaviorTree/BTService.h"
#include "STUFireService.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUFireService : public UBTService
{
	GENERATED_BODY()

protected:
	USTUFireService();

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="AI")
	FBlackboardKeySelector EnemyActorKey;

	virtual void TickNode(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory, float
DeltaSeconds) override;
};
```

STUFireService.cpp:

```
#include "AI/Services/STUFireService.h"

#include "AIController.h"
#include "STUUtils.h"
#include "STUWeaponComponent.h"
#include "BehaviorTree/BlackboardComponent.h"

USTUFireService::USTUFireService()
{
	NodeName = "Fire";
}

void USTUFireService::TickNode(UBehaviorTreeComponent& OwnerComp, uint8* NodeMemory, float
DeltaSeconds)
{
	const auto Controller = OwnerComp.GetAIOwner();
	const auto Blackboard = OwnerComp.GetBlackboardComponent();

	const auto HasAim = Blackboard && Blackboard-
>GetValueAsObject(EnemyActorKey.SelectedKeyName);
	if(Controller)
	{
		const auto WeaponComponent =
STUUtils::GetSTUPlayerComponent<USTUWeaponComponent>(Controller->GetPawn());
		if(WeaponComponent)
		{
```

```
                        HasAim ? WeaponComponent->StartFire() : WeaponComponent->StopFire();
                }
        }
        Super::TickNode(OwnerComp, NodeMemory, DeltaSeconds);
}
```

Tasks:

STUNextLocationTask.h:

```
#pragma once

#include "CoreMinimal.h"
#include "BehaviorTree/BTTaskNode.h"
#include "STUNextLocationTask.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUNextLocationTask : public UBTTaskNode
{
        GENERATED_BODY()

public:
        USTUNextLocationTask();

        virtual EBTNodeResult::Type ExecuteTask(UBehaviorTreeComponent& OwnerComp, uint8*
NodeMemory) override;

protected:
        UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="AI")
        float Radius = 1000.0f;

        UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="AI")
        FBlackboardKeySelector AimLocationKey;

        UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="AI")
        bool SelfCenter = true;

        UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="AI", meta=(EditCondition =
"!SelfCenter"))
        FBlackboardKeySelector CenterActorKey;
};
```

STUNextLocationTask.cpp:

```
#include "AI/Tasks/STUNextLocationTask.h"

#include "AIController.h"
#include "NavigationSystem.h"
#include "BehaviorTree/BlackboardComponent.h"

USTUNextLocationTask::USTUNextLocationTask()
{
        NodeName = "Next Location";
}

EBTNodeResult::Type USTUNextLocationTask::ExecuteTask(UBehaviorTreeComponent& OwnerComp, uint8*
NodeMemory)
{
        const auto Controller = OwnerComp.GetAIOwner();
        const auto Blackboard = OwnerComp.GetBlackboardComponent();
        if(!Controller || !Blackboard) return EBTNodeResult::Failed;
        const auto Pawn = Controller->GetPawn();
        if(!Pawn) return EBTNodeResult::Failed;
        const auto NavSys = UNavigationSystemV1::GetCurrent(Pawn);
        if(!NavSys) return EBTNodeResult::Failed;
        FNavLocation NavLocation;
        auto Location = Pawn->GetActorLocation();
        if(!SelfCenter)
        {
                auto CenterActor = Cast<AActor>(Blackboard-
>GetValueAsObject(CenterActorKey.SelectedKeyName));
                if(!CenterActor) return EBTNodeResult::Failed;
                Location = CenterActor->GetActorLocation();
        }
        const auto Found = NavSys->GetRandomReachablePointInRadius(Location, Radius,
NavLocation);
```

```
        if(!Found) return EBTNodeResult::Failed;
        Blackboard->SetValueAsVector(AimLocationKey.SelectedKeyName, NavLocation.Location);
        return EBTNodeResult::Succeeded;
}
```

Animations.

AnimUtils:

```
#pragma once

class AnimUtils
{
public:

        template<typename T>
        static  T* FindNotifyByClass(UAnimSequenceBase * Animation)
        {
                if(!Animation) return nullptr;
                const auto NotifyEvents = Animation->Notifies;
                for(auto NotifyEvent : NotifyEvents)
                {
                        auto AnimNotify = Cast<T>(NotifyEvent.Notify);
                        if(AnimNotify)
                        {
                                return AnimNotify;
                        }
                }
                return nullptr;
        }
};
```

STUAnimNotify.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Animation/AnimNotifies/AnimNotify.h"
#include "STUAnimNotify.generated.h"

DECLARE_MULTICAST_DELEGATE_OneParam(FOnNotifiedSignature, USkeletalMeshComponent*);
UCLASS()
class SHOOTTHEMUP_API USTUAnimNotify : public UAnimNotify
{
        GENERATED_BODY()

public:
        virtual void Notify(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation)
override;

        FOnNotifiedSignature OnNotified;
};
```

STUAnimNotify.cpp:

```
#include "Animations/STUAnimNotify.h"

void USTUAnimNotify::Notify(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation)
{
        OnNotified.Broadcast(MeshComp);
        Super::Notify(MeshComp, Animation);
}
```

STUEquipFinishedAnimNotify:

```
#pragma once

#include "CoreMinimal.h"
#include "Animations/STUAnimNotify.h"
#include "STUEquipFinishedAnimNotify.generated.h"
```

```
UCLASS()
class SHOOTTHEMUP_API USTUEquipFinishedAnimNotify : public USTUAnimNotify
{
	GENERATED_BODY()

};
```

STUReloadFinishedAnimNotify:

```
#pragma once

#include "CoreMinimal.h"
#include "Animations/STUAnimNotify.h"
#include "STUReloadFinishedAnimNotify.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUReloadFinishedAnimNotify : public USTUAnimNotify
{
	GENERATED_BODY()

};
```

Components.

STUAIPerceptionComponent.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Perception/AIPerceptionComponent.h"
#include "STUAIPerceptionComponent.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUAIPerceptionComponent : public UAIPerceptionComponent
{
	GENERATED_BODY()

public:
	AActor* GetClosesEnemy() const;
};
```

STUAIPerceptionComponent.cpp:

```
#include "Components/STUAIPerceptionComponent.h"
#include "AIController.h"
#include "STUHealthComponent.h"
#include "STUUtils.h"
#include "Perception/AISense_Damage.h"
#include "Perception/AISense_Sight.h"

AActor* USTUAIPerceptionComponent::GetClosesEnemy() const
{
	TArray<AActor*> PercieveActors;
	GetCurrentlyPerceivedActors(UAISense_Sight::StaticClass(), PercieveActors);
	if(PercieveActors.Num() == 0)
	{
		GetCurrentlyPerceivedActors(UAISense_Damage::StaticClass(), PercieveActors);
		if(PercieveActors.Num() == 0)
		{
			return nullptr;
		}
	}

	const auto Controller = Cast<AAIController>(GetOwner());
	if(!Controller) return nullptr;

	const auto Pawn = Controller->GetPawn();
	if(!Pawn) return nullptr;

	float BestDistance = MAX_FLT;
```

```
	AActor* BestPawn = nullptr;
	for(const auto PercieveActor : PercieveActors)
	{
		const auto HealthComponent =
STUUtils::GetSTUPlayerComponent<USTUHealthComponent>(PercieveActor);
		const auto PercievePawn = Cast<APawn>(PercieveActor);
		const auto AreEnemies = PercievePawn && STUUtils::AreEnemies(Controller,
PercievePawn->Controller);
		if(HealthComponent && !HealthComponent->IsDead() && AreEnemies)
		{
			const auto CurrentDistance = (PercieveActor->GetActorLocation() - Pawn-
>GetActorLocation()).Size();
			if(CurrentDistance < BestDistance)
			{
				BestDistance = CurrentDistance;
				BestPawn = PercieveActor;
			}
		}
	}
	return BestPawn;
}
```

STUAIWeaponComponent.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Components/STUWeaponComponent.h"
#include "STUAIWeaponComponent.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUAIWeaponComponent : public USTUWeaponComponent
{
	GENERATED_BODY()

public:
	virtual void StartFire() override;
	virtual void NextWeapon() override;
};
```

STUAIWeaponComponent.cpp:

```
#include "Components/STUAIWeaponComponent.h"

void USTUAIWeaponComponent::StartFire()
{
	if(!CanFire()) return;
	if(CurrentWeapon->IsAmmoEmpty())
	{
		NextWeapon();
	}
	else
	{
		CurrentWeapon->StartFire();
	}
}

void USTUAIWeaponComponent::NextWeapon()
{
	if(!CanEquip()) return;
	int32 NextIndex = (CurrentWeaponIndex + 1) % Weapons.Num();
	while (NextIndex != CurrentWeaponIndex)
	{
		if(!Weapons[NextIndex]->IsAmmoEmpty()) break;
		NextIndex = (NextIndex + 1) % Weapons.Num();
	}
	if(CurrentWeaponIndex != NextIndex)
	{
		CurrentWeaponIndex = NextIndex;
		EquipWeapon(CurrentWeaponIndex);
	}
}
```

STUCharacterMovementComponent.h:

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "STUCharacterMovementComponent.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUCharacterMovementComponent : public UCharacterMovementComponent
{
	GENERATED_BODY()

public:
	virtual float GetMaxSpeed() const override;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Movement", meta=(ClampMin =
"1.5", ClampMax = "10.0"))
	float RunModifier = 2.0f;
};
```

STUCharacterMovementComponent.cpp:

```
#include "Components/STUCharacterMovementComponent.h"
#include "STUBaseCharacter.h"

float USTUCharacterMovementComponent::GetMaxSpeed() const
{
	const float MaxSpeed = Super::GetMaxSpeed();
	const ASTUBaseCharacter* Player = Cast<ASTUBaseCharacter>(GetPawnOwner());
	return Player && Player->IsRunning() ? MaxSpeed * RunModifier : MaxSpeed;
}
```

STUHealthComponent.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Components/ActorComponent.h"
#include "STUCoreTypes.h"
#include "STUHealthComponent.generated.h"

class UCameraShakeBase;

UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class SHOOTTHEMUP_API USTUHealthComponent : public UActorComponent
{
	GENERATED_BODY()

public:
	USTUHealthComponent();

	FOnDeathSignature OnDeath;
	FOnHealthChangedSignature OnHealthChanged;

	UFUNCTION(BlueprintCallable, Category="Health")
	bool IsDead() const { return FMath::IsNearlyZero(Health); }

	UFUNCTION(BlueprintCallable, Category="Health")
	float GetHealthPercent() const{return Health / MaxHealth;}

	float GetHealth() const { return  this->Health; }

	bool TryToAddHealth(float HealthAmount);
	bool IsHealthFull() const;

protected:

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Health", meta=(ClampMin =
"0.0", ClampMax = "1000.0"))
	float MaxHealth = 100.0f;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Heal")
```

```
        bool AutoHeal = true;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Heal", meta=(EditCondition =
"AutoHeal"))
        float HealUpdateTime = 1.0f;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Heal", meta=(EditCondition =
"AutoHeal"))
        float HealDellay = 3.0f;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Heal", meta=(EditCondition =
"AutoHeal"))
        float HealModifier = 5.0f;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX")
        TSubclassOf<UCameraShakeBase> CameraShake;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Health")
        TMap<UPhysicalMaterial*, float> DamageModifiers;

        virtual void BeginPlay() override;

private:
        float Health = 0.0f;
        FTimerHandle HealTimerHandle;

        UFUNCTION()
        void OnTakeAnyDamage(AActor* DamagedActor, float Damage, const class UDamageType*
DamageType, class AController* InstigatedBy, AActor* DamageCauser );
        UFUNCTION()
        void OnTakePointDamage( AActor* DamagedActor, float Damage, class AController*
InstigatedBy, FVector HitLocation, class UPrimitiveComponent* FHitComponent,
                FName BoneName, FVector ShotFromDirection, const class UDamageType* DamageType,
AActor* DamageCauser);
        UFUNCTION()
        void OnTakeRadialDamage(AActor* DamagedActor, float Damage, const class UDamageType*
DamageType, FVector Origin, FHitResult HitInfo,
                class AController* InstigatedBy, AActor* DamageCauser);

        void HealUpdate();
        void SetHealth(float Health);
        void PlayCameraShake();
        void Killed(AController* KillerController);
        void ApplyDamage(float Damage, AController* InstigatedBy);
        float GetPointDamageModifier(AActor* DamagedActor, const FName& BoneName);

        void ReportDamageEvent(float Damage, AController* InstigatedBy);
};
```

STUHealthComponent.cpp:

```
#include "Components/STUHealthComponent.h"
#include "STUGameModeBase.h"
#include "GameFramework/Actor.h"
#include "GameFramework/Character.h"
#include "Perception/AISense_Damage.h"
#include "PhysicalMaterials/PhysicalMaterial.h"


DEFINE_LOG_CATEGORY_STATIC(LogHealthComponent, All, All);

USTUHealthComponent::USTUHealthComponent()
{
        PrimaryComponentTick.bCanEverTick = false;
}

bool USTUHealthComponent::TryToAddHealth(float HealthAmount)
{
        if(IsDead() || IsHealthFull()) return false;
        SetHealth(Health + HealthAmount);
        return true;
}

bool USTUHealthComponent::IsHealthFull() const
{
        return FMath::IsNearlyEqual(Health, MaxHealth);
```

```
}

// Called when the game starts
void USTUHealthComponent::BeginPlay()
{
	Super::BeginPlay();

	check(MaxHealth > 0);
	SetHealth(MaxHealth);

	AActor* ComponentOwner = GetOwner();
	if(ComponentOwner)
	{
		ComponentOwner->OnTakeAnyDamage.AddDynamic(this,
&USTUHealthComponent::OnTakeAnyDamage);
		ComponentOwner->OnTakePointDamage.AddDynamic(this,
&USTUHealthComponent::OnTakePointDamage);
		ComponentOwner->OnTakeRadialDamage.AddDynamic(this,
&USTUHealthComponent::OnTakeRadialDamage);
	}
}

void USTUHealthComponent::OnTakeAnyDamage(AActor* DamagedActor, float Damage, const UDamageType*
DamageType,
	AController* InstigatedBy, AActor* DamageCauser)
{
}

void USTUHealthComponent::OnTakePointDamage(AActor* DamagedActor, float Damage, AController*
InstigatedBy,
	FVector HitLocation, UPrimitiveComponent* FHitComponent, FName BoneName, FVector
ShotFromDirection,
	const UDamageType* DamageType, AActor* DamageCauser)
{
	const auto FinalDamage = Damage * GetPointDamageModifier(DamagedActor, BoneName);
	ApplyDamage(FinalDamage, InstigatedBy);
}

void USTUHealthComponent::OnTakeRadialDamage(AActor* DamagedActor, float Damage, const
UDamageType* DamageType,
	FVector Origin, FHitResult HitInfo, AController* InstigatedBy, AActor* DamageCauser)
{
	ApplyDamage(Damage, InstigatedBy);
}

void USTUHealthComponent::HealUpdate()
{
	SetHealth(Health + HealModifier);


	if(IsHealthFull() && GetWorld())
	{
		GetWorld()->GetTimerManager().ClearTimer(HealTimerHandle);
	}
}

void USTUHealthComponent::SetHealth(float NewHealth)
{
	const auto NextHealth = FMath::Clamp(NewHealth, 0.0f, MaxHealth);
	const auto HealthDelta = NextHealth - Health;
	Health = NextHealth;
	OnHealthChanged.Broadcast(Health, HealthDelta);
}

void USTUHealthComponent::PlayCameraShake()
{
	if(IsDead()) return;
	const auto Player = Cast<APawn>(GetOwner());
	if(!Player) return;
	const auto Controller = Player->GetController<APlayerController>();
	if(!Controller || !Controller->PlayerCameraManager) return;
	Controller->PlayerCameraManager->StartCameraShake(CameraShake);
}

void USTUHealthComponent::Killed(AController* KillerController)
{
	if(!GetWorld()) return;
	const auto GameMode = Cast<ASTUGameModeBase>(GetWorld()->GetAuthGameMode());
```

```
        if(!GameMode) return;

        const auto Player = Cast<APawn>(GetOwner());
        const auto VictimController = Player ? Player->Controller : nullptr;

        GameMode->Killed(KillerController, VictimController);
}

void USTUHealthComponent::ApplyDamage(float Damage, AController* InstigatedBy)
{
        if(Damage <= 0.0f || IsDead() || !GetWorld()) return;

        SetHealth(Health - Damage);


        GetWorld()->GetTimerManager().ClearTimer(HealTimerHandle);

        if(IsDead())
        {
                Killed(InstigatedBy);
                OnDeath.Broadcast();
        }
        else if(AutoHeal)
        {
                GetWorld()->GetTimerManager().SetTimer(HealTimerHandle, this,
&USTUHealthComponent::HealUpdate, HealUpdateTime, true, HealDellay);
        }
        PlayCameraShake();
        ReportDamageEvent(Damage, InstigatedBy);
}

float USTUHealthComponent::GetPointDamageModifier(AActor* DamagedActor, const FName& BoneName)
{
        const auto Character = Cast<ACharacter>(DamagedActor);
        if(!Character || !Character->GetMesh() || !Character->GetMesh()-
>GetBodyInstance(BoneName)) return 1.0f;

        const auto PhysMaterial = Character->GetMesh()->GetBodyInstance(BoneName)-
>GetSimplePhysicalMaterial();
        if(!PhysMaterial || !DamageModifiers.Contains(PhysMaterial)) return 1.0f;
        return DamageModifiers[PhysMaterial];
}

void USTUHealthComponent::ReportDamageEvent(float Damage, AController* InstigatedBy)
{
        if(!InstigatedBy || !InstigatedBy->GetPawn() || !GetOwner()) return;
        UAISense_Damage::ReportDamageEvent(GetWorld(), GetOwner(), InstigatedBy->GetPawn(),
Damage,
                InstigatedBy->GetPawn()->GetActorLocation(), GetOwner()->GetActorLocation());
}
```

STURespawnComponent.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Components/ActorComponent.h"
#include "STURespawnComponent.generated.h"


UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class SHOOTTHEMUP_API USTURespawnComponent : public UActorComponent
{
        GENERATED_BODY()

public:
        // Sets default values for this component's properties
        USTURespawnComponent();

        void Respawn(int32 RespawnTime);

        int32 GetRespawnCountDown() const {return RespawnCountDown; }
        bool IsRespawnInProgress() const;
private:
        FTimerHandle RespawnTimerHandle;
        int32 RespawnCountDown = 0;
```

```
	void RespawnTimerUpdate();
};
```

STURespawnComponent.cpp:

```
#include "Components/STURespawnComponent.h"
#include "STUGameModeBase.h"

USTURespawnComponent::USTURespawnComponent()
{
	PrimaryComponentTick.bCanEverTick = false;

}

void USTURespawnComponent::Respawn(int32 RespawnTime)
{
	if(!GetWorld()) return;
	RespawnCountDown = RespawnTime;
	GetWorld()->GetTimerManager().SetTimer(RespawnTimerHandle, this,
&USTURespawnComponent::RespawnTimerUpdate, 1.0f, true);
}

bool USTURespawnComponent::IsRespawnInProgress() const
{
	return GetWorld() && GetWorld()->GetTimerManager().IsTimerActive(RespawnTimerHandle);
}

void USTURespawnComponent::RespawnTimerUpdate()
{
	if(--RespawnCountDown == 0)
	{
		if(!GetWorld()) return;
		GetWorld()->GetTimerManager().ClearTimer(RespawnTimerHandle);
		const auto GameMode = Cast<ASTUGameModeBase>(GetWorld()->GetAuthGameMode());
		if(!GameMode) return;
		GameMode->RespawnRequest(Cast<AController>(GetOwner()));
	}
}
```

STUWeaponComponent.h:

```
#pragma once

#include "CoreMinimal.h"
#include "STUBaseWeapon.h"
#include "Components/ActorComponent.h"
#include "STUCoreTypes.h"
#include "STUWeaponComponent.generated.h"

class ASTUBaseWeapon;

UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class SHOOTTHEMUP_API USTUWeaponComponent : public UActorComponent
{
	GENERATED_BODY()

public:
	USTUWeaponComponent();
	virtual void StartFire();
	void StopFire();
	virtual void NextWeapon();
	void Reload();

	bool GetCurrentWeaponUIData(FWeaponUIData& UIData);
	bool GetCurrentWeaponAmmoData(FAmmoData& AmmoData);

	bool TryToAddAmmo(TSubclassOf<ASTUBaseWeapon> WeaponType, int32 ClipsAmount);
	bool NeedAmmo(TSubclassOf<ASTUBaseWeapon> WeaponType);

	void Zoom(bool Enabled);

protected:
	virtual void BeginPlay() override;
	virtual void EndPlay(const EEndPlayReason::Type EndPlayReason) override;
```

```
        bool CanFire() const;
        bool CanEquip() const;
        void EquipWeapon(int32 WeaponIndex);

        UPROPERTY(EditDefaultsOnly, Category="Weapon")
        TArray<FWeaponData> WeaponData;

        UPROPERTY(EditDefaultsOnly, Category="Weapon")
        FName WeaponEquipSocketName = "WeaponSocket";

        UPROPERTY(EditDefaultsOnly, Category="Weapon")
        FName WeaponArmorySocketName = "ArmorySocket";

        UPROPERTY(EditDefaultsOnly, Category="Animation")
        UAnimMontage* EquipAnimMontage;

        UPROPERTY()
        ASTUBaseWeapon* CurrentWeapon = nullptr;

        UPROPERTY()
        TArray<ASTUBaseWeapon*> Weapons;

        int32 CurrentWeaponIndex = 0;

private:
        UPROPERTY()
        UAnimMontage *CurrentReloadAnimMontage = nullptr;

        bool EquipAnimInProgress = false;
        bool ReloadAnimInProgress = false;

        void SpawnWeapons();
        void AttachWeaponToSocket(ASTUBaseWeapon* Weapon, USceneComponent* SceneComponent, const
FName& SocketName);

        void PlayAnimMontage(UAnimMontage* Animation);
        void InitAnimations();
        void OnEquipFinished(USkeletalMeshComponent* MeshComponent);
        void OnReloadFinished(USkeletalMeshComponent* MeshComponent);

        bool CanReload() const;

        void OnEmptyClip(ASTUBaseWeapon* AmmoEmptyWeapon);
        void ChangeClip();
};
```

STUWeaponComponent.cpp:

```
#include "Components/STUWeaponComponent.h"
#include "Animations/STUEquipFinishedAnimNotify.h"
#include "Weapon/STUBaseWeapon.h"
#include "GameFramework/Character.h"
#include "Animations/STUReloadFinishedAnimNotify.h"
#include "Animations/AnimUtils.h"
// Sets default values for this component's properties
USTUWeaponComponent::USTUWeaponComponent()
{
        PrimaryComponentTick.bCanEverTick = false;
}

void USTUWeaponComponent::StartFire()
{
        if(!CanFire()) return;
        CurrentWeapon->StartFire();

}

void USTUWeaponComponent::StopFire()
{
        if(!CurrentWeapon) return;
        CurrentWeapon->StopFire();
}

void USTUWeaponComponent::NextWeapon()
{
        if(!CanEquip()) return;
```

```
	CurrentWeaponIndex = (CurrentWeaponIndex + 1) % Weapons.Num();
	EquipWeapon(CurrentWeaponIndex);
}

void USTUWeaponComponent::Reload()
{
	ChangeClip();
}

bool USTUWeaponComponent::GetCurrentWeaponUIData(FWeaponUIData& UIData)
{
	if(CurrentWeapon)
	{
		UIData = CurrentWeapon->GeUIData();
		return true;
	}
	return false;
}

bool USTUWeaponComponent::GetCurrentWeaponAmmoData(FAmmoData& AmmoData)
{
	if(CurrentWeapon)
	{
		AmmoData = CurrentWeapon->GetAmmoData();
		return true;
	}
	return false;
}

bool USTUWeaponComponent::TryToAddAmmo(TSubclassOf<ASTUBaseWeapon> WeaponType, int32 ClipsAmount)
{
	for(const auto Weapon : Weapons)
	{
		if(Weapon && Weapon->IsA(WeaponType))
		{
			return Weapon->TryToAddAmmo(ClipsAmount);
		}
	}
	return false;
}

bool USTUWeaponComponent::NeedAmmo(TSubclassOf<ASTUBaseWeapon> WeaponType)
{
	for(const auto Weapon : Weapons)
	{
		if(Weapon && Weapon->IsA(WeaponType))
		{
			return !Weapon->IsAmmoFull();
		}
	}
	return false;
}

void USTUWeaponComponent::Zoom(bool Enabled)
{
	if(CurrentWeapon)
	{
		CurrentWeapon->Zoom(Enabled);
	}
}

void USTUWeaponComponent::AttachWeaponToSocket(ASTUBaseWeapon* Weapon, USceneComponent*
SceneComponent,
                                               const FName& SocketName)
{
	if(!Weapon || !SceneComponent) return;
	FAttachmentTransformRules AttachmentRules(EAttachmentRule::SnapToTarget, false);
	Weapon->AttachToComponent(SceneComponent, AttachmentRules, SocketName);

}

void USTUWeaponComponent::EquipWeapon(int32 WeaponIndex)
{
	if(WeaponIndex < 0 || WeaponIndex >= Weapons.Num())
	{
		return;
	}
	ACharacter* Character = Cast<ACharacter>(GetOwner());
```

```
	if(!Character) return;
	if(CurrentWeapon)
	{
		CurrentWeapon->Zoom(false);
		CurrentWeapon->StopFire();
		AttachWeaponToSocket(CurrentWeapon, Character->GetMesh(), WeaponArmorySocketName);
	}
	CurrentWeapon = Weapons[WeaponIndex];
	//CurrentReloadAnimMontage = WeaponData[WeaponIndex].ReloadAnimMontage;
	const auto CurrentWeaponData = WeaponData.FindByPredicate([&](const FWeaponData& Data){
		return Data.WeaponClass == CurrentWeapon->GetClass();
	});
	CurrentReloadAnimMontage = CurrentWeaponData ? CurrentWeaponData->ReloadAnimMontage :
nullptr;
	AttachWeaponToSocket(CurrentWeapon, Character->GetMesh(), WeaponEquipSocketName);
	EquipAnimInProgress = true;
	PlayAnimMontage(EquipAnimMontage);
}

void USTUWeaponComponent::PlayAnimMontage(UAnimMontage* Animation)
{
	ACharacter* Character = Cast<ACharacter>(GetOwner());
	if(!Character) return;

	Character->PlayAnimMontage(Animation);
}

void USTUWeaponComponent::InitAnimations()
{
	auto const EquipFinishedNotify =
AnimUtils::FindNotifyByClass<USTUEquipFinishedAnimNotify>(EquipAnimMontage);
	if(EquipFinishedNotify)
	{
		EquipFinishedNotify->OnNotified.AddUObject(this,
&USTUWeaponComponent::OnEquipFinished);
	}
	else
	{
		UE_LOG(LogTemp, Error, TEXT("Anim notify isn`t set"));
		checkNoEntry();
	}
	for (auto OneWeaponData : WeaponData)
	{
		auto ReloadFinishedNotify =
AnimUtils::FindNotifyByClass<USTUReloadFinishedAnimNotify>(OneWeaponData.ReloadAnimMontage);
		if(!ReloadFinishedNotify)
		{
			UE_LOG(LogTemp, Error, TEXT("Anim notify isn`t set"));
			checkNoEntry();
		}
		ReloadFinishedNotify->OnNotified.AddUObject(this,
&USTUWeaponComponent::OnReloadFinished);
	}
}

void USTUWeaponComponent::OnEquipFinished(USkeletalMeshComponent* MeshComponent)
{
	ACharacter* Character = Cast<ACharacter>(GetOwner());
	if(!Character || MeshComponent != Character->GetMesh()) return;
	EquipAnimInProgress = false;

}

void USTUWeaponComponent::OnReloadFinished(USkeletalMeshComponent* MeshComponent)
{
	ACharacter* Character = Cast<ACharacter>(GetOwner());
	if(!Character || MeshComponent != Character->GetMesh()) return;
	ReloadAnimInProgress = false;
}

bool USTUWeaponComponent::CanFire() const
{
	return CurrentWeapon && !EquipAnimInProgress && !ReloadAnimInProgress;
}

bool USTUWeaponComponent::CanEquip() const
{
	return !EquipAnimInProgress && !ReloadAnimInProgress;
```

```
}

bool USTUWeaponComponent::CanReload() const
{
	return CurrentWeapon && !EquipAnimInProgress && !ReloadAnimInProgress && CurrentWeapon-
>CanReload();
}

void USTUWeaponComponent::OnEmptyClip(ASTUBaseWeapon* AmmoEmptyWeapon)
{
	if(!AmmoEmptyWeapon) return;
	if(CurrentWeapon == AmmoEmptyWeapon)
	{
		ChangeClip();
	}
	else
	{
		for(const auto Weapon : Weapons)
		{
			if(Weapon == AmmoEmptyWeapon)
			{
				Weapon->ChangeClip();
			}
		}
	}
}

void USTUWeaponComponent::ChangeClip()
{
	if(!CanReload()) return;
	CurrentWeapon->StopFire();
	CurrentWeapon->ChangeClip();
	ReloadAnimInProgress = true;
	PlayAnimMontage(CurrentReloadAnimMontage);
}


// Called when the game starts
void USTUWeaponComponent::BeginPlay()
{
	Super::BeginPlay();
	checkf(WeaponData.Num() == 2, TEXT("Weapons != 2"))
	CurrentWeaponIndex = 0;
	InitAnimations();
	SpawnWeapons();
	EquipWeapon(CurrentWeaponIndex);
}

void USTUWeaponComponent::EndPlay(const EEndPlayReason::Type EndPlayReason)
{
	CurrentWeapon = nullptr;
	for(auto Weapon : Weapons)
	{
		Weapon->DetachFromActor(FDetachmentTransformRules::KeepWorldTransform);
		Weapon->Destroy();
	}
	Weapons.Empty();
	Super::EndPlay(EndPlayReason);
}

void USTUWeaponComponent::SpawnWeapons()
{
	ACharacter* Character = Cast<ACharacter>(GetOwner());
	if(!Character || !GetWorld()) return;
	for(auto OneWeaponData : WeaponData)
	{
		auto Weapon = GetWorld()->SpawnActor<ASTUBaseWeapon>(OneWeaponData.WeaponClass);
		if(!Weapon) continue;

		Weapon->OnClipEmpty.AddUObject(this, &USTUWeaponComponent::OnEmptyClip);
		Weapon->SetOwner(Character);
		Weapons.Add(Weapon);

		AttachWeaponToSocket(Weapon, Character->GetMesh(), WeaponArmorySocketName);
	}

}
```

Dev.

STUDevDamageActor.h:

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "STUDevDamageActor.generated.h"

UCLASS()
class SHOOTTHEMUP_API ASTUDevDamageActor : public AActor
{
	GENERATED_BODY()

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

	UPROPERTY(VisibleAnywhere, BlueprintReadWrite)
	USceneComponent* SceneComponent;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	float Radius = 300.0f;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	FColor SphereColor = FColor::Red;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	float Damage = 10.0f;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	bool DoFullDamage = false;

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
	TSubclassOf<UDamageType> DamageType;

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

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

};
```

STUDevDamageActor.cpp:

```
#include "Dev/STUDevDamageActor.h"
#include "DrawDebugHelpers.h"
#include "Kismet/GameplayStatics.h"

// Sets default values
ASTUDevDamageActor::ASTUDevDamageActor()
{
	// Set this actor to call Tick() every frame.  You can turn this off to improve
performance if you don't need it.
	PrimaryActorTick.bCanEverTick = true;

	SceneComponent = CreateDefaultSubobject<USceneComponent>("SceneComponent");
	SetRootComponent(SceneComponent);
}

// Called when the game starts or when spawned
void ASTUDevDamageActor::BeginPlay()
{
	Super::BeginPlay();

}

// Called every frame
void ASTUDevDamageActor::Tick(float DeltaTime)
{
```

```
        Super::Tick(DeltaTime);

        DrawDebugSphere(GetWorld(), GetActorLocation(), Radius, 24, SphereColor);
        UGameplayStatics::ApplyRadialDamage(
                GetWorld(),
                Damage,
                GetActorLocation(),
                Radius,
                DamageType,
                {},
                this,
                nullptr,
                DoFullDamage);
}
```

STUFireDamageType.h:

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/DamageType.h"
#include "STUFireDamageType.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUFireDamageType : public UDamageType
{
        GENERATED_BODY()

};
```

STUIceDamageType.h:

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/DamageType.h"
#include "STUIceDamageType.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUIceDamageType : public UDamageType
{
        GENERATED_BODY()

};
```

Menu.

STUMenuGameModeBase.h:

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/GameModeBase.h"
#include "STUMenuGameModeBase.generated.h"

UCLASS()
class SHOOTTHEMUP_API ASTUMenuGameModeBase : public AGameModeBase
{
        GENERATED_BODY()

protected:
        ASTUMenuGameModeBase();
};
```

STUMenuGameModeBase.cpp:

```
#include "Menu/STUMenuGameModeBase.h"
```

```
#include "Menu/STUMenuPlayerController.h"
#include "Menu/UI/STUMenuHUD.h"

ASTUMenuGameModeBase::ASTUMenuGameModeBase()
{
	PlayerControllerClass = ASTUMenuPlayerController::StaticClass();
	HUDClass = ASTUMenuHUD::StaticClass();
}
```

STUMenuPlayerController.h:

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/PlayerController.h"
#include "STUMenuPlayerController.generated.h"

UCLASS()
class SHOOTTHEMUP_API ASTUMenuPlayerController : public APlayerController
{
	GENERATED_BODY()

public:
	virtual void BeginPlay() override;
};
```

STUMenuPlayerController.cpp:

```
#include "Menu/STUMenuPlayerController.h"
#include "STUGameInstance.h"

void ASTUMenuPlayerController::BeginPlay()
{
	Super::BeginPlay();

	SetInputMode(FInputModeUIOnly());
	bShowMouseCursor = true;
	//GetWorld()->GetGameInstance<USTUGameInstance>()->TestString = "Menu";
}
```

UI:

STULevelItemWidget.h:

```
#pragma once

#include "CoreMinimal.h"
#include "STUCoreTypes.h"
#include "Blueprint/UserWidget.h"
#include "Components/Button.h"
#include "Components/Image.h"
#include "Components/TextBlock.h"
#include "STULevelItemWidget.generated.h"

/**
 *
 */
UCLASS()
class SHOOTTHEMUP_API USTULevelItemWidget : public UUserWidget
{
	GENERATED_BODY()

public:
	FOnLevelSelectedSignature OnLevelSelected;

	void SetLevelData(const FLevelData& Data);
	FLevelData GetLevelData() const {return LevelData;}

	void SetSelected(bool IsSelected);
```

```
protected:
	UPROPERTY(meta=(BindWidget))
	UButton* LevelSelectButton;

	UPROPERTY(meta=(BindWidget))
	UTextBlock* LevelNameTextBlock;

	UPROPERTY(meta=(BindWidget))
	UImage* LevelImage;

	UPROPERTY(meta=(BindWidget))
	UImage* FrameImage;

	virtual void NativeOnInitialized() override;

private:
	FLevelData LevelData;

	UFUNCTION()
	void OnLevelItemClicked();

	UFUNCTION()
	void OnLevelItemHovered();

	UFUNCTION()
	void OnLevelItemUnhovered();
};
```

STULevelItemWidget.cpp:

```
#include "Menu/UI/STULevelItemWidget.h"

void USTULevelItemWidget::SetLevelData(const FLevelData& Data)
{
	LevelData = Data;
	if(LevelNameTextBlock)
	{
		LevelNameTextBlock->SetText(FText::FromName(Data.LevelDisplayName));
	}
	if(LevelImage)
	{
		LevelImage->SetBrushFromTexture(Data.LevelThumb);
	}
}

void USTULevelItemWidget::SetSelected(bool IsSelected)
{
	if(LevelImage)
	{
		LevelImage->SetColorAndOpacity(IsSelected ? FLinearColor(1.0f, 0.0f, 0.0f, 1.0f) :
FLinearColor(1.0f, 1.0f, 1.0f, 0.7f));
	}
}

void USTULevelItemWidget::NativeOnInitialized()
{
	Super::NativeOnInitialized();

	if(LevelSelectButton)
	{
		LevelSelectButton->OnClicked.AddDynamic(this,
&USTULevelItemWidget::OnLevelItemClicked);
		LevelSelectButton->OnHovered.AddDynamic(this,
&USTULevelItemWidget::USTULevelItemWidget::OnLevelItemHovered);
		LevelSelectButton->OnUnhovered.AddDynamic(this,
&USTULevelItemWidget::USTULevelItemWidget::OnLevelItemUnhovered);
	}
}

void USTULevelItemWidget::OnLevelItemClicked()
{
	OnLevelSelected.Broadcast(LevelData);
}

void USTULevelItemWidget::OnLevelItemHovered()
{
```

```
        if(FrameImage)
        {
                FrameImage->SetVisibility(ESlateVisibility::Visible);
        }
}

void USTULevelItemWidget::OnLevelItemUnhovered()
{
        if(FrameImage)
        {
                FrameImage->SetVisibility(ESlateVisibility::Hidden);
        }
}
```

STUMenuHUD.h:

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/HUD.h"
#include "STUMenuHUD.generated.h"

UCLASS()
class SHOOTTHEMUP_API ASTUMenuHUD : public AHUD
{
        GENERATED_BODY()

protected:
        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
        TSubclassOf<UUserWidget> MenuWidgetClass;

        virtual void BeginPlay() override;
};
```

STUMenuHUD.cpp:

```
#include "Menu/UI/STUMenuHUD.h"
#include "UI/STUBaseWidget.h"

void ASTUMenuHUD::BeginPlay()
{
        Super::BeginPlay();
        if(MenuWidgetClass)
        {
                const auto MenuWidget = CreateWidget<USTUBaseWidget>(GetWorld(), MenuWidgetClass);
                if(MenuWidget)
                {
                        MenuWidget->AddToViewport();
                        MenuWidget->Show();
                }
        }
}
```

STUMenuWidget.h:

```
#pragma once

#include "CoreMinimal.h"
#include "STUGameInstance.h"
#include "STULevelItemWidget.h"
#include "UI/STUBaseWidget.h"
#include "Components/Button.h"
#include "Components/HorizontalBox.h"
#include "Sound/SoundCue.h"
#include "STUMenuWidget.generated.h"

/**
 *
 */
UCLASS()
class SHOOTTHEMUP_API USTUMenuWidget : public USTUBaseWidget
{
        GENERATED_BODY()
```

```
protected:
	UPROPERTY(meta=(BindWidget))
	UButton* StartGameButton;

	UPROPERTY(meta=(BindWidget))
	UButton* QuitGameButton;

	UPROPERTY(meta=(BindWidget))
	UHorizontalBox* LevelItemBox;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
	TSubclassOf<UUserWidget> LevelItemWidgetClass;

	UPROPERTY(meta=(BindWidgetAnim), Transient)
	UWidgetAnimation* HideAnimation;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Sound")
	USoundCue* StartGameSound;

	virtual void NativeOnInitialized() override;
	virtual void OnAnimationFinished_Implementation(const UWidgetAnimation* Animation)
override;

private:
	UPROPERTY()
	TArray<USTULevelItemWidget*> LevelItemWidgets;
	UFUNCTION()
	void OnStartGame();

	UFUNCTION()
	void OnQuitGame();

	void InitLevelItems();
	void OnLevelSelected(const FLevelData& Data);
	USTUGameInstance* GetGameInstance() const;
};
```

STUMenuWidget.cpp:

```
#include "Menu/UI/STUMenuWidget.h"
#include "STUGameInstance.h"
#include "Kismet/GameplayStatics.h"

void USTUMenuWidget::NativeOnInitialized()
{
	Super::NativeOnInitialized();
	if(StartGameButton)
	{
		StartGameButton->OnClicked.AddDynamic(this, &USTUMenuWidget::OnStartGame);
	}
	if(QuitGameButton)
	{
		QuitGameButton->OnClicked.AddDynamic(this, &USTUMenuWidget::OnQuitGame);
	}
	InitLevelItems();
}

void USTUMenuWidget::OnAnimationFinished_Implementation(const UWidgetAnimation* Animation)
{
	Super::OnAnimationFinished_Implementation(Animation);
	if(Animation != HideAnimation) return;
	const auto STUGameInstance = GetGameInstance();
	if(!STUGameInstance) return;

	UGameplayStatics::OpenLevel(this, STUGameInstance->GetStartupLevel().LevelName);
}

void USTUMenuWidget::OnStartGame()
{
	PlayAnimation(HideAnimation);
	UGameplayStatics::PlaySound2D(GetWorld(), StartGameSound);
}

void USTUMenuWidget::OnQuitGame()
{
```

```
        UKismetSystemLibrary::QuitGame(this, GetOwningPlayer(), EQuitPreference::Quit, true);
}

void USTUMenuWidget::InitLevelItems()
{
        const auto STUGameInstance = GetGameInstance();
        if(!STUGameInstance) return;

        checkf(STUGameInstance->GetLevelsData().Num() != 0, TEXT("Levels data is empty!"))

        if(!LevelItemBox) return;
        LevelItemBox->ClearChildren();

        for(auto LevelData : STUGameInstance->GetLevelsData())
        {
                const auto LevelItemWidget = CreateWidget<USTULevelItemWidget>(GetWorld(),
LevelItemWidgetClass);
                if(!LevelItemWidget) continue;

                LevelItemWidget->SetLevelData(LevelData);
                LevelItemWidget->OnLevelSelected.AddUObject(this,
&USTUMenuWidget::OnLevelSelected);
                LevelItemBox->AddChild(LevelItemWidget);
                LevelItemWidgets.Add(LevelItemWidget);
        }
        if(STUGameInstance->GetStartupLevel().LevelName.IsNone())
        {
                OnLevelSelected(STUGameInstance->GetLevelsData()[0]);
        }
        else
        {
                OnLevelSelected(STUGameInstance->GetStartupLevel());
        }
}

void USTUMenuWidget::OnLevelSelected(const FLevelData& Data)
{
        const auto STUGameInstance = GetGameInstance();
        if(!STUGameInstance) return;

        STUGameInstance->SetStartupLevel(Data);
        for(auto LevelItemWidget : LevelItemWidgets)
        {
                if(LevelItemWidget)
                {
                        const auto IsSelected = Data.LevelName == LevelItemWidget-
>GetLevelData().LevelName;
                        LevelItemWidget->SetSelected(IsSelected);
                }
        }
}

USTUGameInstance* USTUMenuWidget::GetGameInstance() const
{
        if(!GetWorld()) return nullptr;
        return GetWorld()->GetGameInstance<USTUGameInstance>();
}
```

Pickups.

STUBasePickup.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Components/SphereComponent.h"
#include "GameFramework/Actor.h"
#include "Sound/SoundCue.h"
#include "STUBasePickup.generated.h"

UCLASS()
class SHOOTTHEMUP_API ASTUBasePickup : public AActor
{
        GENERATED_BODY()
```

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

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

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

	UPROPERTY(VisibleAnywhere,BlueprintReadWrite, Category="Pickup")
	float RespawnTime = 10.0f;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="Sound")
	USoundCue* PickupTakenSound;

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

private:
	float RotationYaw = 0.0f;
	FTimerHandle RespawnTimerHandle;

	virtual bool GivePickupTo(APawn* PlayerPawn);
	void PickupWasTaken();
	void Respawn();
	void GenerateRotationYaw();
};
```

STUBasePickup.cpp:

```
#include "Pickups/STUBasePickup.h"

#include "Kismet/GameplayStatics.h"

// Sets default values
ASTUBasePickup::ASTUBasePickup()
{
	// Set this actor to call Tick() every frame.  You can turn this off to improve
performance if you don't need it.
	PrimaryActorTick.bCanEverTick = true;

	CollisionComponent = CreateDefaultSubobject<USphereComponent>("SphereComponent");
	CollisionComponent->InitSphereRadius(50.0f);
	CollisionComponent->SetCollisionEnabled(ECollisionEnabled::QueryOnly);
	CollisionComponent->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Overlap);
	SetRootComponent(CollisionComponent);
}

// Called when the game starts or when spawned
void ASTUBasePickup::BeginPlay()
{
	Super::BeginPlay();
	check(CollisionComponent);
	GenerateRotationYaw();
}

void ASTUBasePickup::NotifyActorBeginOverlap(AActor* OtherActor)
{
	Super::NotifyActorBeginOverlap(OtherActor);

	const auto Pawn = Cast<APawn>(OtherActor);
	if(GivePickupTo(Pawn))
	{
		PickupWasTaken();
	}
}

// Called every frame
void ASTUBasePickup::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);
```

```
	AddActorLocalRotation(FRotator(0.0f, RotationYaw, 0.0f));
}

bool ASTUBasePickup::CouldBeTaken() const
{
	return !GetWorldTimerManager().IsTimerActive(RespawnTimerHandle);
}

bool ASTUBasePickup::GivePickupTo(APawn* PlayerPawn)
{
	return false;
}

void ASTUBasePickup::PickupWasTaken()
{
	CollisionComponent->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
	if(GetRootComponent())
	{
		GetRootComponent()->SetVisibility(false, true);
	}
	GetWorldTimerManager().SetTimer(RespawnTimerHandle, this,&ASTUBasePickup::Respawn,
RespawnTime);
	UGameplayStatics::PlaySoundAtLocation(GetWorld(), PickupTakenSound, GetActorLocation());
}

void ASTUBasePickup::Respawn()
{
	GenerateRotationYaw();
	CollisionComponent->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Overlap);
	if(GetRootComponent())
	{
		GetRootComponent()->SetVisibility(true, true);
	}
}

void ASTUBasePickup::GenerateRotationYaw()
{
	const auto Direction = FMath::RandBool() ? 1.0f : -1.0f;
	RotationYaw = FMath::RandRange(1.0f, 2.0f) * Direction;
}
```

STUAmmoPickup.h:

```
#pragma once

#include "CoreMinimal.h"
#include "STUBaseWeapon.h"
#include "Pickups/STUBasePickup.h"
#include "STUAmmoPickup.generated.h"

/**
 *
 */
UCLASS()
class SHOOTTHEMUP_API ASTUAmmoPickup : public ASTUBasePickup
{
	GENERATED_BODY()

protected:
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="Pickup", meta=(ClampMin = "1.0",
ClampMax = "10.0"))
	int32 ClipsAmount = 10;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="Pickup")
	TSubclassOf<ASTUBaseWeapon> WeaponType;

private:
	virtual bool GivePickupTo(APawn* PlayerPawn) override;
};
```

STUAmmoPickup.cpp:

```
#include "Pickups/STUAmmoPickup.h"

#include "STUHealthComponent.h"
```

```
#include "STUUtils.h"
#include "STUWeaponComponent.h"

bool ASTUAmmoPickup::GivePickupTo(APawn* PlayerPawn)
{
	const auto HealthComponent =
STUUtils::GetSTUPlayerComponent<USTUHealthComponent>(PlayerPawn);
	if(!HealthComponent || HealthComponent->IsDead()) return false;

	const auto WeaponComponent =
STUUtils::GetSTUPlayerComponent<USTUWeaponComponent>(PlayerPawn);
	if(!WeaponComponent) return false;

	return WeaponComponent->TryToAddAmmo(WeaponType, ClipsAmount);
}
```

STUHealthPickup.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Pickups/STUBasePickup.h"
#include "STUHealthPickup.generated.h"

/**
 *
 */
UCLASS()
class SHOOTTHEMUP_API ASTUHealthPickup : public ASTUBasePickup
{
	GENERATED_BODY()

protected:
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="Pickup", meta=(ClampMin = "1.0",
ClampMax = "100.0"))
	float HealthAmount = 100.0f;

private:
	virtual bool GivePickupTo(APawn* PlayerPawn) override;
};
```

STUHealthPickup.cpp:

```
#include "Pickups/STUAmmoPickup.h"

#include "STUHealthComponent.h"
#include "STUUtils.h"
#include "STUWeaponComponent.h"

bool ASTUAmmoPickup::GivePickupTo(APawn* PlayerPawn)
{
	const auto HealthComponent =
STUUtils::GetSTUPlayerComponent<USTUHealthComponent>(PlayerPawn);
	if(!HealthComponent || HealthComponent->IsDead()) return false;

	const auto WeaponComponent =
STUUtils::GetSTUPlayerComponent<USTUWeaponComponent>(PlayerPawn);
	if(!WeaponComponent) return false;

	return WeaponComponent->TryToAddAmmo(WeaponType, ClipsAmount);
}
```

Player.

STUBaseCharacter.h:

```
#pragma once

#include "CoreMinimal.h"
#include "STUWeaponComponent.h"
#include "Camera/CameraComponent.h"
#include "Components/STUHealthComponent.h"
```

```
#include "GameFramework/Character.h"
#include "Sound/SoundCue.h"
#include "STUBaseCharacter.generated.h"

UCLASS()
class SHOOTTHEMUP_API ASTUBaseCharacter : public ACharacter
{
	GENERATED_BODY()

public:
	// Sets default values for this character's properties
	ASTUBaseCharacter(const FObjectInitializer& ObjInit);

protected:
	// Called when the game starts or when spawned
	virtual void BeginPlay() override;
	virtual void OnHealthChanged(float Health, float HealthDelta);
	virtual void OnDeath();

	UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category="Components")
	USTUHealthComponent* HealthComponent;

	UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category="Components")
	USTUWeaponComponent* WeaponComponent;

	UPROPERTY(EditAnywhere, Category="Animations")
	UAnimMontage* DeathAnimMontage;

	UPROPERTY(EditDefaultsOnly, Category="Damage")
	FVector2D LandedDamageVelocity = FVector2D(900.0f, 1500.0f);

	UPROPERTY(EditDefaultsOnly, Category="Damage")
	FVector2D LandedDamage = FVector2D(10.0f, 100.0f);

	UPROPERTY(EditDefaultsOnly, Category="Material")
	FName MaterialColorName = "Paint Color";

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Sound")
	USoundCue* DeathSound;

public:
	virtual void Tick(float DeltaTime) override;
	virtual void TurnOff() override;
	virtual void Reset() override;

	UFUNCTION(BlueprintCallable, Category="Movement")
	virtual bool IsRunning() const;

	UFUNCTION(BlueprintCallable, Category="Movement")
	float GetMovementDirection() const;

	void SetPlayerColor(const FLinearColor& Color);

private:
	UFUNCTION()
	void OnGroundLanded(const FHitResult& Hit);

};
```

STUBaseCharacter.cpp:

```
#include "STUBaseCharacter.h"
#include "Components/CapsuleComponent.h"
#include "Components/STUCharacterMovementComponent.h"
#include "Kismet/GameplayStatics.h"

// Sets default values
ASTUBaseCharacter::ASTUBaseCharacter(const FObjectInitializer& ObjInit) :
Super(ObjInit.SetDefaultSubobjectClass<USTUCharacterMovementComponent>(ACharacter::CharacterMovem
entComponentName))
{
	// Set this character to call Tick() every frame.  You can turn this off to improve
performance if you don't need it.
	PrimaryActorTick.bCanEverTick = true;

	HealthComponent = CreateDefaultSubobject<USTUHealthComponent>("HealthComponent");
```

```
	WeaponComponent = CreateDefaultSubobject<USTUWeaponComponent>("WeaponComponent");
}

// Called when the game starts or when spawned
void ASTUBaseCharacter::BeginPlay()
{
	Super::BeginPlay();

	check(HealthComponent);
	check(GetCapsuleComponent());
	check(GetCharacterMovement());
	check(GetMesh());

	OnHealthChanged(HealthComponent->GetHealth(), 0.0f);
	HealthComponent->OnDeath.AddUObject(this, &ASTUBaseCharacter::OnDeath);
	HealthComponent->OnHealthChanged.AddUObject(this, &ASTUBaseCharacter::OnHealthChanged);
	LandedDelegate.AddDynamic(this, &ASTUBaseCharacter::OnGroundLanded);

}

// Called every frame
void ASTUBaseCharacter::Tick(float DeltaTime)
{
	Super::Tick(DeltaTime);
}

void ASTUBaseCharacter::TurnOff()
{
	WeaponComponent->StopFire();
	WeaponComponent->Zoom(false);
	Super::TurnOff();
}

void ASTUBaseCharacter::Reset()
{
	WeaponComponent->StopFire();
	WeaponComponent->Zoom(false);
	Super::Reset();
}

bool ASTUBaseCharacter::IsRunning() const
{
	return false;
}

float ASTUBaseCharacter::GetMovementDirection() const
{
	if(GetVelocity().IsZero()) return 0.0f;
	const auto VelocityNormal = GetVelocity().GetSafeNormal();
	const auto AngleBetween = FMath::Acos(FVector::DotProduct(GetActorForwardVector(),
VelocityNormal));
	const auto CrossProduct = FVector::CrossProduct(GetActorForwardVector(), VelocityNormal);
	const auto Degrees = FMath::RadiansToDegrees(AngleBetween);
	return CrossProduct.IsZero() ? Degrees : Degrees * FMath::Sign(CrossProduct.Z);
}

void ASTUBaseCharacter::SetPlayerColor(const FLinearColor& Color)
{
	const auto MaterialInst = GetMesh()->CreateAndSetMaterialInstanceDynamic(0);
	if(!MaterialInst) return;
	MaterialInst->SetVectorParameterValue(MaterialColorName, Color);
}


void ASTUBaseCharacter::OnDeath()
{
	//PlayAnimMontage(DeathAnimMontage);
	GetCharacterMovement()->DisableMovement();
	SetLifeSpan(5.0f);
	GetCapsuleComponent()->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Ignore);
	WeaponComponent->StopFire();
	WeaponComponent->Zoom(false);

	GetMesh()->SetCollisionEnabled(ECollisionEnabled::QueryAndPhysics);
	GetMesh()->SetSimulatePhysics(true);

	UGameplayStatics::PlaySoundAtLocation(GetWorld(), DeathSound, GetActorLocation());
}
```

```
void ASTUBaseCharacter::OnHealthChanged(float Health, float HealthDelta)
{

}

void ASTUBaseCharacter::OnGroundLanded(const FHitResult& Hit)
{
	const auto FallVelocityZ = -GetVelocity().Z;
	if(FallVelocityZ < LandedDamageVelocity.X) return;
	const auto FinalDamage = FMath::GetMappedRangeValueClamped(LandedDamageVelocity,
LandedDamage, FallVelocityZ);
	TakeDamage(FinalDamage, FDamageEvent{}, nullptr, nullptr);
}
```

STUPlayerCharacter.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Components/SphereComponent.h"
#include "Player/STUBaseCharacter.h"
#include "GameFramework/SpringArmComponent.h"
#include "STUPlayerCharacter.generated.h"

UCLASS()
class SHOOTTHEMUP_API ASTUPlayerCharacter : public ASTUBaseCharacter
{
	GENERATED_BODY()

public:
	ASTUPlayerCharacter(const FObjectInitializer& ObjInit);

protected:
	UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category="Components")
	UCameraComponent* CameraComponent;

	UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category="Components")
	USpringArmComponent* SpringArmComponent;

	UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category="Components")
	USphereComponent* CameraCollisionComponent;

	virtual void OnDeath() override;
	virtual void BeginPlay() override;

public:
	virtual void SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent)
override;

	virtual bool IsRunning() const override;

private:
	bool WantsToRun = false;
	bool IsMovingForward = false;

	void MoveForward(float Amount);
	void MoveRight(float Amount);

	void OnStartRunning();
	void OnStopRunning();

	UFUNCTION()
	void OnCameraCollisionBeginOverlap(UPrimitiveComponent* OverlappedComponent, AActor*
OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const
FHitResult & SweepResult);

	UFUNCTION()
	void OnCameraCollisionEndOverlap(UPrimitiveComponent* OverlappedComponent, AActor*
OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex);

	void CheckCameraOverlap();
};
```

STUPlayerCharacter.cpp:

```
#include "Player/STUPlayerCharacter.h"
#include "Components/CapsuleComponent.h"

ASTUPlayerCharacter::ASTUPlayerCharacter(const FObjectInitializer& ObjInit) : Super(ObjInit)
{
	// Set this character to call Tick() every frame.  You can turn this off to improve
performance if you don't need it.
	PrimaryActorTick.bCanEverTick = true;

	SpringArmComponent = CreateDefaultSubobject<USpringArmComponent>("SpringArmComponent");
	SpringArmComponent->SetupAttachment(GetRootComponent());
	SpringArmComponent->bUsePawnControlRotation = true;
	SpringArmComponent->SocketOffset = FVector(0.0f, 100.0f, 80.0f);

	CameraComponent = CreateDefaultSubobject<UCameraComponent>("CameraComponent");
	CameraComponent->SetupAttachment(SpringArmComponent);

	CameraCollisionComponent =
CreateDefaultSubobject<USphereComponent>("CameraCollisionComponent");
	CameraCollisionComponent->SetupAttachment(CameraComponent);
	CameraCollisionComponent->SetSphereRadius(10.0f);
	CameraCollisionComponent-
>SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Overlap);
}

void ASTUPlayerCharacter::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
	Super::SetupPlayerInputComponent(PlayerInputComponent);

	check(PlayerInputComponent);
	check(WeaponComponent);
	PlayerInputComponent->BindAxis("MoveForward", this, &ASTUPlayerCharacter::MoveForward);
	PlayerInputComponent->BindAxis("MoveRight", this, &ASTUPlayerCharacter::MoveRight);
	PlayerInputComponent->BindAxis("LookUp", this,
&ASTUPlayerCharacter::AddControllerPitchInput);
	PlayerInputComponent->BindAxis("TurnAround", this,
&ASTUPlayerCharacter::AddControllerYawInput);
	PlayerInputComponent->BindAction("Jump", IE_Pressed, this, &ASTUPlayerCharacter::Jump);
	PlayerInputComponent->BindAction("Run", IE_Pressed, this,
&ASTUPlayerCharacter::OnStartRunning);
	PlayerInputComponent->BindAction("Run", IE_Released, this,
&ASTUPlayerCharacter::OnStopRunning);
	PlayerInputComponent->BindAction("Fire", IE_Pressed, WeaponComponent,
&USTUWeaponComponent::StartFire);
	PlayerInputComponent->BindAction("Fire", IE_Released, WeaponComponent,
&USTUWeaponComponent::StopFire);
	PlayerInputComponent->BindAction("NextWeapon", IE_Pressed, WeaponComponent,
&USTUWeaponComponent::NextWeapon);
	PlayerInputComponent->BindAction("Reload", IE_Pressed, WeaponComponent,
&USTUWeaponComponent::Reload);
	DECLARE_DELEGATE_OneParam(FZoomInputSignature, bool);
	PlayerInputComponent->BindAction<FZoomInputSignature>("Zoom", IE_Pressed,
WeaponComponent, &USTUWeaponComponent::Zoom, true);
	PlayerInputComponent->BindAction<FZoomInputSignature>("Zoom", IE_Released,
WeaponComponent, &USTUWeaponComponent::Zoom, false);
}

void ASTUPlayerCharacter::MoveForward(float Amount)
{
	IsMovingForward = Amount > 0.0f;
	if(Amount == 0.0f) return;
	AddMovementInput(GetActorForwardVector(), Amount);
}

void ASTUPlayerCharacter::MoveRight(float Amount)
{
	if(Amount == 0.0f) return;
	AddMovementInput(GetActorRightVector(), Amount);
}

void ASTUPlayerCharacter::OnStartRunning()
{
	WantsToRun = true;
}

void ASTUPlayerCharacter::OnStopRunning()
{
	WantsToRun = false;
```

```
}

void ASTUPlayerCharacter::OnCameraCollisionBeginOverlap(UPrimitiveComponent* OverlappedComponent,
AActor* OtherActor,
	UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult&
SweepResult)
{
	CheckCameraOverlap();
}

void ASTUPlayerCharacter::OnCameraCollisionEndOverlap(UPrimitiveComponent* OverlappedComponent,
AActor* OtherActor,
	UPrimitiveComponent* OtherComp, int32 OtherBodyIndex)
{
	CheckCameraOverlap();
}

void ASTUPlayerCharacter::CheckCameraOverlap()
{
	const auto HideMesh = CameraCollisionComponent-
>IsOverlappingComponent(GetCapsuleComponent());
	GetMesh()->SetOwnerNoSee(HideMesh);

	TArray<USceneComponent*> MeshChildren;
	GetMesh()->GetChildrenComponents(true, MeshChildren);

	for (auto MeshChild : MeshChildren)
	{
		const auto MeshChildGeometry = Cast<UPrimitiveComponent>(MeshChild);
		if(MeshChildGeometry)
		{
			MeshChildGeometry->SetOwnerNoSee(HideMesh);
		}
	}
}

bool ASTUPlayerCharacter::IsRunning() const
{
	return WantsToRun && IsMovingForward && !GetVelocity().IsZero();
}

void ASTUPlayerCharacter::OnDeath()
{
	Super::OnDeath();
	if(Controller)
	{
		Controller->ChangeState(NAME_Spectating);
	}
}

void ASTUPlayerCharacter::BeginPlay()
{
	Super::BeginPlay();
	check(CameraCollisionComponent);
	CameraCollisionComponent->OnComponentBeginOverlap.AddDynamic(this,
&ASTUPlayerCharacter::OnCameraCollisionBeginOverlap);
	CameraCollisionComponent->OnComponentEndOverlap.AddDynamic(this,
&ASTUPlayerCharacter::OnCameraCollisionEndOverlap);
}
```

STUPlayerController.h:

```
#pragma once

#include "CoreMinimal.h"
#include "STUCoreTypes.h"
#include "STURespawnComponent.h"
#include "GameFramework/PlayerController.h"
#include "STUPlayerController.generated.h"

/**
 *
 */
UCLASS()
class SHOOTTHEMUP_API ASTUPlayerController : public APlayerController
{
```

```
	GENERATED_BODY()

public:
	ASTUPlayerController();

protected:
	UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category="AI")
	USTURespawnComponent* RespawnComponent;

	virtual void BeginPlay() override;
	virtual void OnPossess(APawn* InPawn) override;
	virtual void SetupInputComponent() override;

private:
	void OnPauseGame();
	void OnMatchStateChanged(ESTUMatchState State);
	void OnMuteSound();
};
```

STUPlayerController.cpp:

```
#include "STUPlayerController.h"
#include "STUGameInstance.h"
#include "STUGameModeBase.h"
#include "GameFramework/GameModeBase.h"

ASTUPlayerController::ASTUPlayerController()
{
	RespawnComponent = CreateDefaultSubobject<USTURespawnComponent>("RespawnComponent");
}

void ASTUPlayerController::BeginPlay()
{
	Super::BeginPlay();
	if(GetWorld())
	{
		const auto GameMode = Cast<ASTUGameModeBase>(GetWorld()->GetAuthGameMode());
		if(GameMode)
		{
			GameMode->OnMatchStateChanged.AddUObject(this,
&ASTUPlayerController::OnMatchStateChanged);
		}
	}
}

void ASTUPlayerController::OnPossess(APawn* InPawn)
{
	Super::OnPossess(InPawn);
	OnNewPawn.Broadcast(InPawn);
}

void ASTUPlayerController::SetupInputComponent()
{
	Super::SetupInputComponent();
	if(!InputComponent) return;

	InputComponent->BindAction("PauseGame", IE_Pressed, this,
&ASTUPlayerController::OnPauseGame);
	InputComponent->BindAction("Mute", IE_Pressed, this, &ASTUPlayerController::OnMuteSound);
}

void ASTUPlayerController::OnPauseGame()
{
	if(!GetWorld() || !GetWorld()->GetAuthGameMode()) return;
	GetWorld()->GetAuthGameMode()->SetPause(this);
}

void ASTUPlayerController::OnMatchStateChanged(ESTUMatchState State)
{
	if(State == ESTUMatchState::InProgress)
	{
		SetInputMode(FInputModeGameOnly());
		bShowMouseCursor = false;
	}
	else
	{
```

```
                SetInputMode(FInputModeUIOnly());
                bShowMouseCursor = true;
        }
}

void ASTUPlayerController::OnMuteSound()
{
        if(!GetWorld()) return;
        const auto STUGameInstance = GetWorld()->GetGameInstance<USTUGameInstance>();
        if(!STUGameInstance) return;
        STUGameInstance->ToggleVolume();
}
```

STUPlayerState.h:

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/PlayerState.h"
#include "STUPlayerState.generated.h"

UCLASS()
class SHOOTTHEMUP_API ASTUPlayerState : public APlayerState
{
        GENERATED_BODY()

public:

        void SetTeamID(int32 ID){ TeamID = ID;}
        int32 GetTeamID() const { return TeamID;}

        void SetTeamColor(const FLinearColor& Color){ TeamColor = Color;}
        FLinearColor GetTeamColor() const { return TeamColor;}

        void AddKill(){KillsNum++;}
        int32 GetKillsNum() const {return KillsNum;}

        void AddDeath(){DeathsNum++;}
        int32 GetDeathsNum() const {return DeathsNum;}

        void SetInvisible(bool NewValue){Invisible = NewValue;};
        bool GetInvisible() const {return Invisible;};
        void LogInfo();

private:
        int32 TeamID;
        FLinearColor TeamColor;
        int32 KillsNum = 0;
        int32 DeathsNum = 0;
        bool Invisible = false;
};
```

STUPlayerState.cpp:

```
#include "Player/STUPlayerState.h"

void ASTUPlayerState::LogInfo()
{
        UE_LOG(LogTemp, Warning, TEXT("TeamID: %i, Kills: %i, Deaths: %i"), TeamID, KillsNum,
DeathsNum);
}
```

Sound.

STUSoundFuncLib.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Kismet/BlueprintFunctionLibrary.h"
#include "STUSoundFuncLib.generated.h"
```

```
UCLASS()
class SHOOTTHEMUP_API USTUSoundFuncLib : public UBlueprintFunctionLibrary
{
	GENERATED_BODY()

public:
	UFUNCTION(BlueprintCallable)
	static void SetSoundClassVolume(USoundClass* SoundClass, float Volume);

	UFUNCTION(BlueprintCallable)
	static void ToggleSoundClassVolume(USoundClass* SoundClass);
};
```

STUSoundFuncLib.cpp:

```
#include "Sound/STUSoundFuncLib.h"

void USTUSoundFuncLib::SetSoundClassVolume(USoundClass* SoundClass, float Volume)
{
	if(!SoundClass) return;

	SoundClass->Properties.Volume = FMath::Clamp(Volume, 0.0f, 1.0f);
}

void USTUSoundFuncLib::ToggleSoundClassVolume(USoundClass* SoundClass)
{
	if(!SoundClass) return;

	const auto NextVolume = SoundClass->Properties.Volume > 0.0f ? 0.0f : 1.0f;
	SetSoundClassVolume(SoundClass, NextVolume);
}
```

UI.

STUBaseWidget.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Blueprint/UserWidget.h"
#include "STUBaseWidget.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUBaseWidget : public UUserWidget
{
	GENERATED_BODY()

public:
	void Show();

protected:
	UPROPERTY(meta=(BindWidgetAnim), Transient)
	UWidgetAnimation* ShowAnimation;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Sound")
	USoundCue* OpenSound;
};
```

STUBaseWidget.cpp:

```
#include "UI/STUBaseWidget.h"
#include "Kismet/GameplayStatics.h"

void USTUBaseWidget::Show()
{
	PlayAnimation(ShowAnimation);
	UGameplayStatics::PlaySound2D(GetWorld(), OpenSound);
}
```

STUGameDataWidget.h:

```
#pragma once

#include "CoreMinimal.h"
#include "STUGameModeBase.h"
#include "STUPlayerState.h"
#include "Blueprint/UserWidget.h"
#include "STUGameDataWidget.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUGameDataWidget : public UUserWidget
{
	GENERATED_BODY()

public:
	UFUNCTION(BlueprintCallable, Category="UI")
	int32 GetCurrentRoundNum() const;

	UFUNCTION(BlueprintCallable, Category="UI")
	int32 GetTotalRoundNum() const;

	UFUNCTION(BlueprintCallable, Category="UI")
	int32 GetRoundSecondsRemaining() const;

private:
	ASTUGameModeBase* GetSTUGameMode() const;
	ASTUPlayerState* GetSTUPlayerState() const;
};
```

STUGameDataWidget.cpp:

```
#include "UI/STUGameDataWidget.h"

int32 USTUGameDataWidget::GetCurrentRoundNum() const
{
	const auto GameMode = GetSTUGameMode();
	return GameMode ? GameMode->GetCurrentRoundNum() : 0;
}

int32 USTUGameDataWidget::GetTotalRoundNum() const
{
	const auto GameMode = GetSTUGameMode();
	return GameMode ? GameMode->GetGameData().RoundsNum : 0;
}

int32 USTUGameDataWidget::GetRoundSecondsRemaining() const
{
	const auto GameMode = GetSTUGameMode();
	return GameMode ? GameMode->GetRoundSecondsRemaining() : 0;
}

ASTUGameModeBase* USTUGameDataWidget::GetSTUGameMode() const
{
	return GetWorld() ? Cast<ASTUGameModeBase>(GetWorld()->GetAuthGameMode()) : nullptr;
}

ASTUPlayerState* USTUGameDataWidget::GetSTUPlayerState() const
{
	return GetOwningPlayer() ? Cast<ASTUPlayerState>(GetOwningPlayer()->PlayerState) :
nullptr;
}
```

STUGameHUD.h:

```
#pragma once

#include "CoreMinimal.h"
#include "STUBaseWidget.h"
#include "STUCoreTypes.h"
#include "GameFramework/HUD.h"
#include "STUGameHUD.generated.h"

UCLASS()
class SHOOTTHEMUP_API ASTUGameHUD : public AHUD
{
```

```
	GENERATED_BODY()

	virtual void DrawHUD() override;

protected:
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
	TSubclassOf<UUserWidget> PlayerHUDWidgetClass;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
	TSubclassOf<UUserWidget> PauseWidgetClass;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
	TSubclassOf<UUserWidget> GameOverWidgetClass;

	virtual void BeginPlay() override;
private:
	UPROPERTY()
	TMap<ESTUMatchState, USTUBaseWidget*> GameWidgets;

	UPROPERTY()
	USTUBaseWidget* CurrentWidget = nullptr;

	void DrawCrossHair();
	void OnMatchStateChanged(ESTUMatchState State);
};
```

STUGameHUD.cpp:

```
#include "UI/STUGameHUD.h"
#include "STUGameModeBase.h"
#include "UI/STUBaseWidget.h"
#include "Engine/Canvas.h"

void ASTUGameHUD::DrawHUD()
{
	Super::DrawHUD();
	//DrawCrossHair();
}

void ASTUGameHUD::BeginPlay()
{
	Super::BeginPlay();
	GameWidgets.Add(ESTUMatchState::InProgress, CreateWidget<USTUBaseWidget>(GetWorld(),
PlayerHUDWidgetClass));
	GameWidgets.Add(ESTUMatchState::Pause, CreateWidget<USTUBaseWidget>(GetWorld(),
PauseWidgetClass));
	GameWidgets.Add(ESTUMatchState::GameOver, CreateWidget<USTUBaseWidget>(GetWorld(),
GameOverWidgetClass));


	for(auto GameWidgetPair : GameWidgets)
	{
		const auto GameWidget = GameWidgetPair.Value;
		if(!GameWidget) continue;
		GameWidget->AddToViewport();
		GameWidget->SetVisibility(ESlateVisibility::Hidden);
	}
	if(GetWorld())
	{
		const auto GameMode = Cast<ASTUGameModeBase>(GetWorld()->GetAuthGameMode());
		if(GameMode)
		{
			GameMode->OnMatchStateChanged.AddUObject(this,
&ASTUGameHUD::OnMatchStateChanged);
		}
	}
}

void ASTUGameHUD::DrawCrossHair()
{
	const TInterval<float> Center((Canvas->SizeX)/2, (Canvas->SizeY)/2);

	const float HalfLineSize = 10.0f;
	const float LineThickness = 2.0f;
```

```
        const FLinearColor LineColor = FLinearColor::Green;

        DrawLine(Center.Min - HalfLineSize, Center.Max, Center.Min + HalfLineSize, Center.Max,
LineColor, LineThickness);
        DrawLine(Center.Min, Center.Max - HalfLineSize, Center.Min, Center.Max + HalfLineSize,
LineColor, LineThickness);
}

void ASTUGameHUD::OnMatchStateChanged(ESTUMatchState State)
{
        if(CurrentWidget)
        {
                CurrentWidget->SetVisibility(ESlateVisibility::Hidden);
        }
        if(GameWidgets.Contains(State))
        {
                CurrentWidget = GameWidgets[State];
        }
        if(CurrentWidget)
        {
                CurrentWidget->SetVisibility(ESlateVisibility::Visible);
                CurrentWidget->Show();
        }
        UE_LOG(LogTemp, Warning, TEXT("State changed: %s"), *UEnum::GetValueAsString(State));
}
```

STUGameOverWidget.h:

```
#pragma once

#include "CoreMinimal.h"
#include "STUCoreTypes.h"
#include "STUGameModeBase.h"
#include "UI/STUBaseWidget.h"
#include "Components/Button.h"
#include "Components/VerticalBox.h"
#include "STUGameOverWidget.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUGameOverWidget : public USTUBaseWidget
{
        GENERATED_BODY()

protected:
        UPROPERTY(meta=(BindWidget))
        UVerticalBox* PlayerStatBox;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
        TSubclassOf<UUserWidget> PlayerStatRowWidgetClass;

        UPROPERTY(meta=(BindWidget))
        UButton* ResetLevelButton;

        virtual void NativeOnInitialized() override;

private:
        void OnMatchStateChanged(ESTUMatchState State);
        void UpdatePlayerStat();

        UFUNCTION()
        void OnResetLevel();
};
```

STUGameOverWidget.cpp:

```
#include "UI/STUGameOverWidget.h"
#include "STUGameModeBase.h"
#include "STUPlayerState.h"
#include "STUPlayerStatRowWidget.h"
#include "STUUtils.h"
#include "Kismet/GameplayStatics.h"

void USTUGameOverWidget::NativeOnInitialized()
{
        Super::NativeOnInitialized();
        if(GetWorld())
```

```
	{
		const auto GameMode = Cast<ASTUGameModeBase>(GetWorld()->GetAuthGameMode());
		if(GameMode)
		{
			GameMode->OnMatchStateChanged.AddUObject(this,
&USTUGameOverWidget::OnMatchStateChanged);
		}
	}
	if(ResetLevelButton)
	{
		ResetLevelButton->OnClicked.AddDynamic(this, &USTUGameOverWidget::OnResetLevel);
	}
}

void USTUGameOverWidget::OnMatchStateChanged(ESTUMatchState State)
{
	if(State == ESTUMatchState::GameOver)
	{
		UpdatePlayerStat();
	}
}

void USTUGameOverWidget::UpdatePlayerStat()
{
	if(!GetWorld() || !PlayerStatBox) return;
	PlayerStatBox->ClearChildren();
	for (auto It  = GetWorld()->GetControllerIterator(); It; ++It)
	{
		const auto Controller = It->Get();
		if(!Controller) continue;
		const auto PlayerState = Cast<ASTUPlayerState>(Controller->PlayerState);
		if(!PlayerState) continue;
		const auto PlayerStatRowWidget = CreateWidget<USTUPlayerStatRowWidget>(GetWorld(),
PlayerStatRowWidgetClass);
		if(!PlayerStatRowWidget) continue;
		PlayerStatRowWidget->SetPlayerName(FText::FromString(PlayerState-
>GetPlayerName()));
		PlayerStatRowWidget->SetKills(STUUtils::TextFromInt(PlayerState->GetKillsNum()));
		PlayerStatRowWidget->SetDeaths(STUUtils::TextFromInt(PlayerState-
>GetDeathsNum()));
		PlayerStatRowWidget->SetTeam(STUUtils::TextFromInt(PlayerState->GetTeamID()));
		PlayerStatRowWidget->SetPlayerIndicateVisibility(Controller-
>IsPlayerController());
		PlayerStatRowWidget->SetTeamColor(PlayerState->GetTeamColor());
		if(!PlayerState->GetInvisible())
		{
			PlayerStatBox->AddChild(PlayerStatRowWidget);
		}
	}
}

void USTUGameOverWidget::OnResetLevel()
{
	const FString CurrentLevelName = UGameplayStatics::GetCurrentLevelName(this);
	UGameplayStatics::OpenLevel(this, FName(CurrentLevelName));
}
```

STUGoToMenuWidget.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Blueprint/UserWidget.h"
#include "Components/Button.h"
#include "STUGoToMenuWidget.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUGoToMenuWidget : public UUserWidget
{
	GENERATED_BODY()

protected:
	UPROPERTY(meta=(BindWidget))
	UButton* GoToMenuButton;

	virtual void NativeOnInitialized() override;
```

```
private:
	UFUNCTION()
	void OnGoToMenu();
};
```

STUGoToMenuWidget.cpp:

```
#include "UI/STUGoToMenuWidget.h"
#include "STUGameInstance.h"
#include "Kismet/GameplayStatics.h"

void USTUGoToMenuWidget::NativeOnInitialized()
{
	Super::NativeOnInitialized();
	if(GoToMenuButton)
	{
		GoToMenuButton->OnClicked.AddDynamic(this, &USTUGoToMenuWidget::OnGoToMenu);
	}
}

void USTUGoToMenuWidget::OnGoToMenu()
{
	if(!GetWorld()) return;
	const auto STUGameInstance = GetWorld()->GetGameInstance<USTUGameInstance>();
	if(!STUGameInstance) return;
	if(STUGameInstance->GetMenuLevelName().IsNone())
	{
		UE_LOG(LogTemp, Error, TEXT("Menu Name is none!"))
		return;
	}
	const FName StartUpLevelName = "Test";
	UGameplayStatics::OpenLevel(this, STUGameInstance->GetMenuLevelName());
}
```

STUHealthBarWidget.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Blueprint/UserWidget.h"
#include "Components/ProgressBar.h"
#include "STUHealthBarWidget.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUHealthBarWidget : public UUserWidget
{
	GENERATED_BODY()

public:
	void SetHealthPercent(float Percent);

protected:
	UPROPERTY(meta=(BindWidget))
	UProgressBar* HealthProgressBar;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
	float PercentVisibleThreshold = 0.6f;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
	float PercentColorThreshold = 0.3f;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
	FLinearColor GoodColor = FLinearColor::Yellow;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
	FLinearColor BadColor = FLinearColor::Red;
};
```

STUHealthBarWidget.cpp:

```
#include "UI/STUHealthBarWidget.h"

void USTUHealthBarWidget::SetHealthPercent(float Percent)
{
```

```
        if(!HealthProgressBar) return;

        const auto HealthBarVisibility = (Percent > PercentVisibleThreshold ||
FMath::IsNearlyZero(Percent)) ? ESlateVisibility::Hidden : ESlateVisibility::Visible;
        HealthProgressBar->SetVisibility(HealthBarVisibility);

        const auto HealthBarColor = Percent > PercentColorThreshold ? GoodColor : BadColor;
        HealthProgressBar->SetFillColorAndOpacity(HealthBarColor);

        HealthProgressBar->SetPercent(Percent);
}
```

STUPauseWidget.h:

```
#pragma once

#include "CoreMinimal.h"
#include "UI/STUBaseWidget.h"
#include "Components/Button.h"
#include "STUPauseWidget.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUPauseWidget : public USTUBaseWidget
{
        GENERATED_BODY()

protected:
        UPROPERTY(meta=(BindWidget))
        UButton* ClearPauseButton;

        virtual void NativeOnInitialized() override;

private:
        UFUNCTION()
        void OnClearPause();
};
```

STUPauseWidget.cpp:

```
#include "UI/STUPauseWidget.h"
#include "GameFramework/GameModeBase.h"

void USTUPauseWidget::NativeOnInitialized()
{
        Super::NativeOnInitialized();
        if(ClearPauseButton)
        {
                ClearPauseButton->OnClicked.AddDynamic(this, &USTUPauseWidget::OnClearPause);
        }
}

void USTUPauseWidget::OnClearPause()
{
        if(!GetWorld() || !GetWorld()->GetAuthGameMode()) return;
        GetWorld()->GetAuthGameMode()->ClearPause();
}
```

STUPlayerHUDWidget.h:

```
#pragma once

#include "CoreMinimal.h"
#include "STUCoreTypes.h"
#include "Components/ProgressBar.h"
#include "UI/STUBaseWidget.h"
#include "STUPlayerHUDWidget.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUPlayerHUDWidget : public USTUBaseWidget
{
        GENERATED_BODY()

public:
        UFUNCTION(BlueprintCallable, Category="UI")
        float GetHealthPercent() const;
```

```
        UFUNCTION(BlueprintCallable, Category="UI")
        bool GetCurrentWeaponUIData(FWeaponUIData& UIData) const;

        UFUNCTION(BlueprintCallable, Category="UI")
        bool GetCurrentWeaponAmmoData(FAmmoData& AmmoData) const;

        UFUNCTION(BlueprintCallable, Category="UI")
        bool IsPlayerAlive() const;

        UFUNCTION(BlueprintCallable, Category="UI")
        bool IsPlayerSpectating() const;

        UFUNCTION(BlueprintImplementableEvent, Category="UI")
        void OnTakeDamage();

        UFUNCTION(BlueprintCallable, Category="UI")
        int32 GetKillsNum() const;

        UFUNCTION(BlueprintCallable, Category="UI")
        FString FormatBullets(int32 BulletsNum) const;

protected:
        UPROPERTY(meta=(BindWidget))
        UProgressBar* HealthProgressBar;

        UPROPERTY(meta=(BindWidgetAnim), Transient)
        UWidgetAnimation* DamageAnimation;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
        float PercentColorThreshold = 0.6f;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
        float PercentColorRisk = 0.3f;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
        FLinearColor GoodColor = FLinearColor::Yellow;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
        FLinearColor BadColor = FLinearColor::Red;

        virtual void NativeOnInitialized() override;

private:
        void OnHealthChanged(float Health, float HealthDelta);
        void OnNewPawn(APawn* Pawn);
        void UpdateHealthBar();
};
```

STUPlayerHUDWidget.cpp:

```
#include "UI/STUPlayerHUDWidget.h"
#include "STUHealthComponent.h"
#include "STUUtils.h"
#include "STUWeaponComponent.h"

float USTUPlayerHUDWidget::GetHealthPercent() const
{
        const auto HealthComponent =
STUUtils::GetSTUPlayerComponent<USTUHealthComponent>(GetOwningPlayerPawn());
        if(!HealthComponent) return 0.0f;
        return HealthComponent->GetHealthPercent();
}

bool USTUPlayerHUDWidget::GetCurrentWeaponUIData(FWeaponUIData& UIData) const
{
        const auto WeaponComponent =
STUUtils::GetSTUPlayerComponent<USTUWeaponComponent>(GetOwningPlayerPawn());
        if(!WeaponComponent) return false;
        return WeaponComponent->GetCurrentWeaponUIData(UIData);
}

bool USTUPlayerHUDWidget::GetCurrentWeaponAmmoData(FAmmoData& AmmoData) const
{
        const auto WeaponComponent =
STUUtils::GetSTUPlayerComponent<USTUWeaponComponent>(GetOwningPlayerPawn());
        if(!WeaponComponent) return false;
        return WeaponComponent->GetCurrentWeaponAmmoData(AmmoData);
```

```
}

bool USTUPlayerHUDWidget::IsPlayerAlive() const
{
	const auto HealthComponent =
STUUtils::GetSTUPlayerComponent<USTUHealthComponent>(GetOwningPlayerPawn());
	return HealthComponent && !HealthComponent->IsDead();
}

bool USTUPlayerHUDWidget::IsPlayerSpectating() const
{
	const auto Controller = GetOwningPlayer();
	return Controller && Controller->GetStateName() == NAME_Spectating;
}

int32 USTUPlayerHUDWidget::GetKillsNum() const
{
	const auto Controller = GetOwningPlayer();
	if(!Controller) return 0;
	const auto PlayerState = Cast<ASTUPlayerState>(Controller->PlayerState);
	return PlayerState ? PlayerState->GetKillsNum() : 0;
}

FString USTUPlayerHUDWidget::FormatBullets(int32 BulletsNum) const
{
	const int32 MaxLen = 2;
	const TCHAR PrefixSymbol = '0';
	auto BulletsStr = FString::FromInt(BulletsNum);
	const auto SymbolsNumToAdd = MaxLen - BulletsStr.Len();

	if(SymbolsNumToAdd > 0)
	{
		BulletsStr = FString::ChrN(SymbolsNumToAdd, PrefixSymbol).Append(BulletsStr);
	}
	return BulletsStr;
}

void USTUPlayerHUDWidget::NativeOnInitialized()
{
	Super::NativeOnInitialized();
	if(GetOwningPlayer())
	{
		GetOwningPlayer()->GetOnNewPawnNotifier().AddUObject(this,
&USTUPlayerHUDWidget::OnNewPawn);
		OnNewPawn(GetOwningPlayerPawn());
	}
}

void USTUPlayerHUDWidget::OnHealthChanged(float Health, float HealthDelta)
{
	if(HealthDelta < 0.0f)
	{
		OnTakeDamage();

		if(!IsAnimationPlaying(DamageAnimation))
		{
			PlayAnimation(DamageAnimation);
		}
	}
	UpdateHealthBar();
}

void USTUPlayerHUDWidget::OnNewPawn(APawn* Pawn)
{
	const auto HealthComponent = STUUtils::GetSTUPlayerComponent<USTUHealthComponent>(Pawn);
	if(HealthComponent && !HealthComponent->OnHealthChanged.IsBoundToObject(this))
	{
		HealthComponent->OnHealthChanged.AddUObject(this,
&USTUPlayerHUDWidget::OnHealthChanged);
	}
	UpdateHealthBar();
}

void USTUPlayerHUDWidget::UpdateHealthBar()
{
	if(HealthProgressBar)
	{
```

```
            HealthProgressBar->SetFillColorAndOpacity(GetHealthPercent() <
PercentColorThreshold ? (GetHealthPercent() < PercentColorRisk ? BadColor : GoodColor) :
FLinearColor::Green);
        }
}
```

STUPlayerStatRowWidget.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Blueprint/UserWidget.h"
#include "Components/Image.h"
#include "Components/TextBlock.h"
#include "STUPlayerStatRowWidget.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUPlayerStatRowWidget : public UUserWidget
{
        GENERATED_BODY()

protected:
        UPROPERTY(meta=(BindWidget))
        UTextBlock* PlayerNameTextBlock;

        UPROPERTY(meta=(BindWidget))
        UTextBlock* KillsTextBlock;

        UPROPERTY(meta=(BindWidget))
        UTextBlock* DeathsTextBlock;

        UPROPERTY(meta=(BindWidget))
        UTextBlock* TeamTextBlock;

        UPROPERTY(meta=(BindWidget))
        UImage* PlayerIndicateImage;

        UPROPERTY(meta=(BindWidget))
        UImage* TeamImage;

public:
        void SetPlayerName(const FText& Text);
        void SetKills(const FText& Text);
        void SetDeaths(const FText& Text);
        void SetTeam(const FText& Text);
        void SetPlayerIndicateVisibility(bool Visible);
        void SetTeamColor(const FLinearColor& Color);
};
```

STUPlayerStatRowWidget.cpp:

```
#include "UI/STUPlayerStatRowWidget.h"

void USTUPlayerStatRowWidget::SetPlayerName(const FText& Text)
{
        if(!PlayerNameTextBlock) return;
        PlayerNameTextBlock->SetText(Text);
}

void USTUPlayerStatRowWidget::SetKills(const FText& Text)
{
        if(!KillsTextBlock) return;
        KillsTextBlock->SetText(Text);
}

void USTUPlayerStatRowWidget::SetDeaths(const FText& Text)
{
        if(!DeathsTextBlock) return;
        DeathsTextBlock->SetText(Text);
}

void USTUPlayerStatRowWidget::SetTeam(const FText& Text)
{
        if(!TeamTextBlock) return;
        TeamTextBlock->SetText(Text);
}
```

```
void USTUPlayerStatRowWidget::SetPlayerIndicateVisibility(bool Visible)
{
	if(!PlayerIndicateImage) return;
	PlayerIndicateImage->SetVisibility(Visible ? ESlateVisibility::Visible :
ESlateVisibility::Hidden);
}

void USTUPlayerStatRowWidget::SetTeamColor(const FLinearColor& Color)
{
	if(!TeamImage) return;
	TeamImage->SetColorAndOpacity(Color);
}
```

STUSpectatorWidget.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Blueprint/UserWidget.h"
#include "STUSpectatorWidget.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUSpectatorWidget : public UUserWidget
{
	GENERATED_BODY()

public:
	UFUNCTION(BlueprintCallable, Category="UI")
	bool GetRespawnTime(int32& CountDownTime) const;
};
```

STUSpectatorWidget.cpp:

```
#include "UI/STUSpectatorWidget.h"
#include "STURespawnComponent.h"
#include "STUUtils.h"

bool USTUSpectatorWidget::GetRespawnTime(int32& CountDownTime) const
{
	const auto RespawnComponent =
STUUtils::GetSTUPlayerComponent<USTURespawnComponent>(GetOwningPlayer());
	if(!RespawnComponent || !RespawnComponent->IsRespawnInProgress()) return false;
	CountDownTime = RespawnComponent->GetRespawnCountDown();
	return true;
}
```

Weapon.

STUBaseWeapon.h:

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "STUCoreTypes.h"
#include "Sound/SoundCue.h"
#include "STUBaseWeapon.generated.h"

class USkeletalMeshComponent;
class UNiagaraSystem;
class UNiagaraComponent;

UCLASS()
class SHOOTTHEMUP_API ASTUBaseWeapon : public AActor
{
	GENERATED_BODY()

public:
	ASTUBaseWeapon();

	FOnClipEmptySignature OnClipEmpty;

	virtual void StartFire();
```

```
        virtual void StopFire();

        void ChangeClip();
        bool CanReload() const;

        FWeaponUIData GeUIData() const { return UIData; }
        FAmmoData GetAmmoData() const {return CurrentAmmo;}

        bool TryToAddAmmo(int32 ClipsAmount);
        bool IsAmmoEmpty() const;
        bool IsAmmoFull() const;

        virtual void Zoom(bool Enabled) {};

protected:
        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Components")
        USkeletalMeshComponent* WeaponMesh;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
        FName MuzzleSocketName = "MuzzleSocket";

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
        float TraceMaxDistance = 2500.0f;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
        FAmmoData DefaultAmmo{20, 5, false};

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
        FWeaponUIData UIData;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX")
        UNiagaraSystem* MuzzleFX;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Sound")
        USoundCue* FireSound;

        // Called when the game starts or when spawned
        virtual void BeginPlay() override;
        virtual void MakeShot();
        virtual bool GetTraceData(FVector& TraceStart, FVector& TraceEnd) const;
        bool GetPlayerViewPoint(FVector& ViewLocation, FRotator& ViewRotation) const;
        FVector GetMuzzleWorldLocation() const;
        void MakeHit(FHitResult& HitResult, const FVector& TraceStart, const FVector& TraceEnd);
        void DecreaseAmmo();
        bool IsClipEmpty() const;

        UNiagaraComponent* SpawnMuzzleFX();

private:
        FAmmoData CurrentAmmo;
};
```

STUBaseWeapon.cpp:

```
#include "Weapon/STUBaseWeapon.h"
#include "DrawDebugHelpers.h"
#include "NiagaraFunctionLibrary.h"
#include "GameFramework/Character.h"
#include "Pickups/STUAmmoPickup.h"

ASTUBaseWeapon::ASTUBaseWeapon()
{
        PrimaryActorTick.bCanEverTick = false;
        WeaponMesh = CreateDefaultSubobject<USkeletalMeshComponent>("Weapon Mesh");
        SetRootComponent(WeaponMesh);
}

void ASTUBaseWeapon::BeginPlay()
{
        Super::BeginPlay();
        check(WeaponMesh);
        checkf(DefaultAmmo.Bullets > 0, TEXT("Bullets <= 0!"));
        checkf(DefaultAmmo.Clips > 0, TEXT("Clips <= 0!"));
        CurrentAmmo = DefaultAmmo;
}

void ASTUBaseWeapon::StartFire()
```

```
{}

void ASTUBaseWeapon::StopFire()
{}

void ASTUBaseWeapon::MakeShot()
{}

bool ASTUBaseWeapon::GetPlayerViewPoint(FVector& ViewLocation, FRotator& ViewRotation) const
{
	const auto STUCharacter = Cast<ACharacter>(GetOwner());
	if(!STUCharacter) return false;
	if(STUCharacter->IsPlayerControlled())
	{
		const auto Controller = STUCharacter->GetController<APlayerController>();
		if(!Controller) return false;
		Controller->GetPlayerViewPoint(ViewLocation, ViewRotation);
	}
	else
	{
		ViewLocation = GetMuzzleWorldLocation();
		ViewRotation = WeaponMesh->GetSocketRotation(MuzzleSocketName);
	}
	return true;
}

FVector ASTUBaseWeapon::GetMuzzleWorldLocation() const
{
	return WeaponMesh->GetSocketLocation(MuzzleSocketName);
}

bool ASTUBaseWeapon::GetTraceData(FVector& TraceStart, FVector& TraceEnd) const
{
	FVector ViewLocation;
	FRotator ViewRotation;
	if(!GetPlayerViewPoint(ViewLocation, ViewRotation)) return false;

	TraceStart = ViewLocation;
	const FVector ShootDirection = ViewRotation.Vector();
	TraceEnd = TraceStart + ShootDirection * TraceMaxDistance;
	return true;
}

void ASTUBaseWeapon::MakeHit(FHitResult& HitResult, const FVector& TraceStart, const FVector&
TraceEnd)
{
	if(!GetWorld()) return;
	FCollisionQueryParams CollisionParams;
	CollisionParams.AddIgnoredActor(GetOwner());
	CollisionParams.bReturnPhysicalMaterial = true;
	GetWorld()->LineTraceSingleByChannel(HitResult, TraceStart, TraceEnd,
ECollisionChannel::ECC_Visibility, CollisionParams, FCollisionResponseParams());
}

void ASTUBaseWeapon::DecreaseAmmo()
{
	if(CurrentAmmo.Bullets == 0)
	{
		return;
	}
	CurrentAmmo.Bullets--;

	if(IsClipEmpty() && !IsAmmoEmpty())
	{
		StopFire();
		OnClipEmpty.Broadcast(this);
	}
}

bool ASTUBaseWeapon::IsAmmoEmpty() const
{
	return !CurrentAmmo.Infinite && CurrentAmmo.Clips == 0 && IsClipEmpty();
}

bool ASTUBaseWeapon::IsClipEmpty() const
{
	return CurrentAmmo.Bullets == 0;
}
```

```
bool ASTUBaseWeapon::IsAmmoFull() const
{
	return CurrentAmmo.Clips == DefaultAmmo.Clips && CurrentAmmo.Bullets ==
DefaultAmmo.Bullets;
}

UNiagaraComponent* ASTUBaseWeapon::SpawnMuzzleFX()
{
	return  UNiagaraFunctionLibrary::SpawnSystemAttached(
		MuzzleFX,
		WeaponMesh,
		MuzzleSocketName,
		FVector::ZeroVector,
		FRotator::ZeroRotator,
		EAttachLocation::SnapToTarget,
		true);
}

void ASTUBaseWeapon::ChangeClip()
{
	if (!CurrentAmmo.Infinite)
	{
		if(CurrentAmmo.Clips == 0)
		{
			return;
		}
		CurrentAmmo.Clips--;
	}
	CurrentAmmo.Bullets = DefaultAmmo.Bullets;
}

bool ASTUBaseWeapon::CanReload() const
{
	return CurrentAmmo.Bullets < DefaultAmmo.Bullets && CurrentAmmo.Clips > 0;
}

bool ASTUBaseWeapon::TryToAddAmmo(int32 ClipsAmount)
{
	if(CurrentAmmo.Infinite || IsAmmoFull() || ClipsAmount <= 0) return false;
	if(IsAmmoEmpty())
	{
		CurrentAmmo.Clips = FMath::Clamp(ClipsAmount, 0, DefaultAmmo.Clips + 1);
		OnClipEmpty.Broadcast(this);
	}
	else if(CurrentAmmo.Clips < DefaultAmmo.Clips)
	{
		const auto NextClipsAmount = CurrentAmmo.Clips + ClipsAmount;
		if(DefaultAmmo.Clips - NextClipsAmount >= 0)
		{
			CurrentAmmo.Clips = NextClipsAmount;
		}
		else
		{
			CurrentAmmo.Clips = DefaultAmmo.Clips;
			CurrentAmmo.Bullets = DefaultAmmo.Bullets;
		}
	}
	else
	{
		CurrentAmmo.Bullets = DefaultAmmo.Bullets;
	}
	return true;
}
```

STULauncherWeapon.h:

```
#pragma once

#include "CoreMinimal.h"
#include "STUProjectile.h"
#include "Weapon/STUBaseWeapon.h"
#include "STULauncherWeapon.generated.h"
UCLASS()
class SHOOTTHEMUP_API ASTULauncherWeapon : public ASTUBaseWeapon
{
```

```
	GENERATED_BODY()

public:
	virtual void StartFire() override;

protected:
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
	TSubclassOf<ASTUProjectile> ProjectileClass;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Sound")
	USoundCue* NoAmmoSound;

	virtual void MakeShot() override;
};
```

STULauncherWeapon.cpp:

```
#include "Weapon/STULauncherWeapon.h"
#include "Kismet/GameplayStatics.h"

void ASTULauncherWeapon::StartFire()
{
	MakeShot();
}

void ASTULauncherWeapon::MakeShot()
{
	if(!GetWorld()){
		StopFire();
		return;
	}
	if(IsAmmoEmpty())
	{
		UGameplayStatics::PlaySoundAtLocation(GetWorld(), NoAmmoSound,
GetActorLocation());
		StopFire();
		return;
	}
	FVector TraceStart, TraceEnd;
	if(!GetTraceData(TraceStart, TraceEnd)){
		StopFire();
		return;
	}
	FHitResult HitResult;
	MakeHit(HitResult, TraceStart, TraceEnd);

	const FVector EndPoint = HitResult.bBlockingHit ? HitResult.ImpactPoint : TraceEnd;
	const FVector Direction = (EndPoint - GetMuzzleWorldLocation()).GetSafeNormal();

	const FTransform SpawnTransform(FRotator::ZeroRotator, GetMuzzleWorldLocation());
	ASTUProjectile* Projectile = GetWorld()-
>SpawnActorDeferred<ASTUProjectile>(ProjectileClass, SpawnTransform);
	if(Projectile)
	{
		Projectile->SetShotDirection(Direction);
		Projectile->SetOwner(GetOwner());
		Projectile->FinishSpawning(SpawnTransform);
	}
	DecreaseAmmo();
	SpawnMuzzleFX();
	UGameplayStatics::SpawnSoundAttached(FireSound, WeaponMesh, MuzzleSocketName);
}
```

STUProjectile.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Components/SphereComponent.h"
#include "GameFramework/Actor.h"
#include "GameFramework/ProjectileMovementComponent.h"
#include "STUProjectile.generated.h"

class USTUWeaponFXComponent;
UCLASS()
class SHOOTTHEMUP_API ASTUProjectile : public AActor
{
```

```
	GENERATED_BODY()

public:
	ASTUProjectile();

	void SetShotDirection(const FVector& Direction)
	{
		ShotDirection = Direction;
	}

protected:
	UPROPERTY(VisibleAnywhere, Category="Weapon")
	USphereComponent* CollisionComponent;

	UPROPERTY(VisibleAnywhere, Category="Weapon")
	UProjectileMovementComponent* MovementComponent;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
	float DamageRadius = 200.0f;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
	float DamageAmount = 50.0f;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
	bool DoFullDamage = false;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
	float LifeSeconds = 5.0f;

	UPROPERTY(VisibleAnywhere, Category="VFX")
	USTUWeaponFXComponent* WeaponFXComponent;

	virtual void BeginPlay() override;

private:
	FVector ShotDirection;

	UFUNCTION()
	void OnProjectileHit(UPrimitiveComponent* HitComponent, AActor* OtherActor,
UPrimitiveComponent* OtherComp, FVector NormalImpulse, const FHitResult& Hit );

	AController* GetController() const;
};
```

STUProjectile.cpp:

```
#include "Weapon/STUProjectile.h"
#include "DrawDebugHelpers.h"
#include "Components/STUWeaponFXComponent.h"
#include "Kismet/GameplayStatics.h"

// Sets default values
ASTUProjectile::ASTUProjectile()
{
	PrimaryActorTick.bCanEverTick = false;

	CollisionComponent = CreateDefaultSubobject<USphereComponent>("SphereComponent");
	CollisionComponent->InitSphereRadius(5.0f);
	CollisionComponent->SetCollisionEnabled(ECollisionEnabled::QueryOnly);
	CollisionComponent->SetCollisionResponseToAllChannels(ECollisionResponse::ECR_Block);
	CollisionComponent->bReturnMaterialOnMove = true;
	SetRootComponent(CollisionComponent);

	MovementComponent =
CreateDefaultSubobject<UProjectileMovementComponent>("ProjectileMovementComponent");
	MovementComponent->InitialSpeed = 2000.0f;
	MovementComponent->ProjectileGravityScale = 0.0f;

	WeaponFXComponent = CreateDefaultSubobject<USTUWeaponFXComponent>("WeaponFXComponent");
}

void ASTUProjectile::BeginPlay()
{
	Super::BeginPlay();
	check(MovementComponent);
	check(CollisionComponent);
	check(WeaponFXComponent);
```

```
	MovementComponent->Velocity = ShotDirection * MovementComponent->InitialSpeed;
	CollisionComponent->IgnoreActorWhenMoving(GetOwner(), true);
	CollisionComponent->OnComponentHit.AddDynamic(this, &ASTUProjectile::OnProjectileHit);
	SetLifeSpan(LifeSeconds);
}

void ASTUProjectile::OnProjectileHit(UPrimitiveComponent* HitComponent, AActor* OtherActor,
	UPrimitiveComponent* OtherComp, FVector NormalImpulse, const FHitResult& Hit)
{
	if(!GetWorld()) return;
	MovementComponent->StopMovementImmediately();

	UGameplayStatics::ApplyRadialDamage(
		GetWorld(),
		DamageAmount,
		GetActorLocation(),
		DamageRadius,
		UDamageType::StaticClass(),
		{GetOwner()},
		this,
		GetController(),
		DoFullDamage);

	//DrawDebugSphere(GetWorld(), GetActorLocation(), DamageRadius, 20, FColor::Red, false,
3.0f);
	WeaponFXComponent->PlayImpactFX(Hit);
	Destroy();
}

AController* ASTUProjectile::GetController() const
{
	const auto Pawn = Cast<APawn>(GetOwner());
	return Pawn ? Pawn->GetController() : nullptr;
}
```

STURifleWeapon.h:

```
#pragma once

#include "CoreMinimal.h"
#include "Components/STUWeaponFXComponent.h"
#include "Weapon/STUBaseWeapon.h"
#include "NiagaraComponent.h"
#include "NiagaraSystem.h"
#include "STURifleWeapon.generated.h"

UCLASS()
class SHOOTTHEMUP_API ASTURifleWeapon : public ASTUBaseWeapon
{
	GENERATED_BODY()

public:
	ASTURifleWeapon();
	virtual void BeginPlay() override;
	virtual void StartFire() override;
	virtual void StopFire() override;
	virtual void Zoom(bool Enabled) override;

protected:
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
	float TimeBetweenShot = 0.1f;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
	float BulletSpread = 1.0f;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
	float DamageAmount = 10.0f;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX")
	UNiagaraSystem* TraceFX;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX")
	FString TraceTargetName = "TraceTarget";

	UPROPERTY(VisibleAnywhere, Category="VFX")
	USTUWeaponFXComponent* WeaponFXComponent;
```

```
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
	float FOVZoomAngle = 50.0f;

	virtual void MakeShot() override;
	virtual bool GetTraceData(FVector& TraceStart, FVector& TraceEnd) const override;
private:
	FTimerHandle ShotTimerHandle;

	UPROPERTY()
	UNiagaraComponent* MuzzleFXComponent;

	UPROPERTY()
	UAudioComponent* FireAudioComponent;

	float DefaultCameraFOV = 90.0f;

	void MakeDamage(const FHitResult& HitResult);
	void InitFX();
	void SetFXActive(bool IsActive);
	void SpawnTraceFX(const FVector& TraceStart, const FVector& TraceEnd);
	AController* GetController() const;
};
```

STURifleWeapon.cpp:

```
#include "Weapon/STURifleWeapon.h"
#include "DrawDebugHelpers.h"
#include "NiagaraFunctionLibrary.h"
#include "Components/AudioComponent.h"
#include "Components/STUWeaponFXComponent.h"
#include "GameFramework/Character.h"
#include "Kismet/GameplayStatics.h"

ASTURifleWeapon::ASTURifleWeapon()
{
	WeaponFXComponent = CreateDefaultSubobject<USTUWeaponFXComponent>("WeaponFXComponent");
}

void ASTURifleWeapon::BeginPlay()
{
	Super::BeginPlay();
	check(WeaponFXComponent);
}

void ASTURifleWeapon::StartFire()
{
	InitFX();
	GetWorldTimerManager().SetTimer(ShotTimerHandle, this, &ASTURifleWeapon::MakeShot,
TimeBetweenShot, true);
	MakeShot();
}

void ASTURifleWeapon::StopFire()
{
	GetWorldTimerManager().ClearTimer(ShotTimerHandle);
	SetFXActive(false);
}

void ASTURifleWeapon::Zoom(bool Enabled)
{
	Super::Zoom(Enabled);
	const auto Controller = Cast<APlayerController>(GetController());
	if(!Controller || !Controller->PlayerCameraManager) return;
	if(Enabled)
	{
		DefaultCameraFOV = Controller->PlayerCameraManager->GetFOVAngle();
	}
	Controller->PlayerCameraManager->SetFOV(Enabled ? FOVZoomAngle : DefaultCameraFOV);
}


void ASTURifleWeapon::MakeShot()
{
	if(!GetWorld() || IsAmmoEmpty()){
		StopFire();
		return;
	}
```

```
        FVector TraceStart, TraceEnd;
        if(!GetTraceData(TraceStart, TraceEnd)) {
                StopFire();
                return;
        }
        FHitResult HitResult;
        MakeHit(HitResult, TraceStart, TraceEnd);

        FVector TraceFXEnd = TraceEnd;
        if(HitResult.bBlockingHit)
        {
                TraceFXEnd = HitResult.ImpactPoint;
                MakeDamage(HitResult);

                WeaponFXComponent->PlayImpactFX(HitResult);
        }
        SpawnTraceFX(GetMuzzleWorldLocation(), TraceFXEnd);
        DecreaseAmmo();
}

bool ASTURifleWeapon::GetTraceData(FVector& TraceStart, FVector& TraceEnd) const
{
        FVector ViewLocation;
        FRotator ViewRotation;
        if(!GetPlayerViewPoint(ViewLocation, ViewRotation)) return false;

        TraceStart = ViewLocation;
        const auto HalfRad = FMath::DegreesToRadians(BulletSpread);
        const FVector ShootDirection = FMath::VRandCone(ViewRotation.Vector(), HalfRad);
        TraceEnd = TraceStart + ShootDirection * TraceMaxDistance;
        return true;
}

void ASTURifleWeapon::MakeDamage(const FHitResult& HitResult)
{
        const auto DamagedActor = HitResult.GetActor();
        if(!DamagedActor) return;
        FPointDamageEvent PointDamageEvent;
        PointDamageEvent.HitInfo = HitResult;
        DamagedActor->TakeDamage(DamageAmount, PointDamageEvent, GetController(), this);
}

void ASTURifleWeapon::InitFX()
{
        if(!MuzzleFXComponent)
        {
                MuzzleFXComponent = SpawnMuzzleFX();
        }
        if(!FireAudioComponent)
        {
                FireAudioComponent = UGameplayStatics::SpawnSoundAttached(FireSound, WeaponMesh,
MuzzleSocketName);
        }
        SetFXActive(true);
}

void ASTURifleWeapon::SetFXActive(bool IsActive)
{
        if(MuzzleFXComponent)
        {
                MuzzleFXComponent->SetPaused(!IsActive);
                MuzzleFXComponent->SetVisibility(IsActive, true);
        }
        if(FireAudioComponent)
        {
                IsActive ? FireAudioComponent->Play() : FireAudioComponent->Stop();
        }
}

void ASTURifleWeapon::SpawnTraceFX(const FVector& TraceStart, const FVector& TraceEnd)
{
        const auto TraceFXComponent = UNiagaraFunctionLibrary::SpawnSystemAtLocation(GetWorld(),
TraceFX, TraceStart);
        if(TraceFXComponent)
        {
                TraceFXComponent->SetNiagaraVariableVec3(TraceTargetName, TraceEnd);
        }
}
```

```
AController* ASTURifleWeapon::GetController() const
{
	const auto Pawn = Cast<APawn>(GetOwner());
	return Pawn ? Pawn->GetController() : nullptr;
}
```

STUWeaponFXComponent.h:

```
#pragma once

#include "CoreMinimal.h"
#include "STUCoreTypes.h"
#include "Components/ActorComponent.h"
#include "STUWeaponFXComponent.generated.h"

class UNiagaraSystem;
class UPhysicalMaterial;
UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class SHOOTTHEMUP_API USTUWeaponFXComponent : public UActorComponent
{
	GENERATED_BODY()

public:
	// Sets default values for this component's properties
	USTUWeaponFXComponent();

	void PlayImpactFX(const FHitResult& Hit);

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

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX")
	FImpactData DefaultImpactData;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX")
	TMap<UPhysicalMaterial*, FImpactData> ImpactDataMap;

public:
	// Called every frame
	virtual void TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction) override;
};
```

STUWeaponFXComponent.cpp:

```
#include "Weapon/Components/STUWeaponFXComponent.h"
#include "NiagaraComponentPool.h"
#include "NiagaraFunctionLibrary.h"
#include "Components/DecalComponent.h"
#include "Kismet/GameplayStatics.h"

USTUWeaponFXComponent::USTUWeaponFXComponent()
{
	PrimaryComponentTick.bCanEverTick = true;
}

void USTUWeaponFXComponent::PlayImpactFX(const FHitResult& Hit)
{
	auto ImpactData = DefaultImpactData;
	if(Hit.PhysMaterial.IsValid())
	{
		const auto PhysMat = Hit.PhysMaterial.Get();
		if(ImpactDataMap.Contains(PhysMat))
		{
			ImpactData = ImpactDataMap[PhysMat];
		}
	}
	//Niagara
	UNiagaraFunctionLibrary::SpawnSystemAtLocation(this, ImpactData.NiagaraEffect,
Hit.ImpactPoint, Hit.ImpactNormal.Rotation());
	//Decal
	auto DecalComponent = UGameplayStatics::SpawnDecalAtLocation(
		GetWorld(),
		ImpactData.DecalData.Material,
		ImpactData.DecalData.Size,
```

```
                Hit.ImpactPoint,
                Hit.ImpactNormal.Rotation());
        if(DecalComponent)
        {
                DecalComponent->SetFadeOut(ImpactData.DecalData.LifeTime,
ImpactData.DecalData.FadeOutTime);
        }
        //Sound
        UGameplayStatics::PlaySoundAtLocation(GetWorld(), ImpactData.Sound, Hit.ImpactPoint);
}

void USTUWeaponFXComponent::BeginPlay()
{
        Super::BeginPlay();
}

void USTUWeaponFXComponent::TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction)
{
        Super::TickComponent(DeltaTime, TickType, ThisTickFunction);
}
```

Інше:

STUUtils:

```
#pragma once
#include "STUPlayerState.h"

class STUUtils
{
public:
        template <typename T>
        static T* GetSTUPlayerComponent(AActor* PlayerPawn)
        {
                if(!PlayerPawn) return nullptr;
                const auto Component = PlayerPawn->GetComponentByClass(T::StaticClass());
                return Cast<T>(Component);
        }

        bool static AreEnemies(AController* Controller1, AController* Controller2)
        {
                if(!Controller1 || !Controller2 || Controller1 == Controller2) return false;

                const auto PlayerState1 = Cast<ASTUPlayerState>(Controller1->PlayerState);
                const auto PlayerState2 = Cast<ASTUPlayerState>(Controller2->PlayerState);
                return PlayerState1 && PlayerState2 && PlayerState1->GetTeamID() != PlayerState2-
>GetTeamID();
        }

        static FText TextFromInt(int32 Number)
        {
                return FText::FromString(FString::FromInt(Number));
        }
};
```

STUCoreTypes:

```
#pragma once

#include "STUCoreTypes.generated.h"
//Weapon
class ASTUBaseWeapon;
DECLARE_MULTICAST_DELEGATE_OneParam(FOnClipEmptySignature, ASTUBaseWeapon*)

USTRUCT(BlueprintType)
struct FAmmoData{
        GENERATED_USTRUCT_BODY()

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
        int32 Bullets;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon", meta=(EditCondition =
"!Infinite"))
```

```
        int32 Clips;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
        bool Infinite;
};

USTRUCT(BlueprintType)
struct FWeaponData
{
        GENERATED_USTRUCT_BODY();

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
        TSubclassOf<ASTUBaseWeapon> WeaponClass;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Weapon")
        UAnimMontage* ReloadAnimMontage = nullptr;
};

USTRUCT(BlueprintType)
struct FWeaponUIData
{
        GENERATED_USTRUCT_BODY();

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
        UTexture2D* MainIcon = nullptr;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="UI")
        UTexture2D* CrossHairIcon = nullptr;

};

//Health
DECLARE_MULTICAST_DELEGATE(FOnDeathSignature);
DECLARE_MULTICAST_DELEGATE_TwoParams(FOnHealthChangedSignature, float, float);

//VFX
class UNiagaraSystem;
class USoundCue;

USTRUCT(BlueprintType)
struct FDecalData
{
        GENERATED_USTRUCT_BODY();

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX")
        UMaterialInterface* Material = nullptr;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX")
        FVector Size = FVector(10.0f);

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX")
        float LifeTime = 5.0f;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX")
        float FadeOutTime = 0.5f;
};

USTRUCT(BlueprintType)
struct FImpactData
{
        GENERATED_USTRUCT_BODY();

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX")
        UNiagaraSystem* NiagaraEffect = nullptr;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX")
        FDecalData DecalData;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Sound")
        USoundCue* Sound = nullptr;
};

USTRUCT(BlueprintType)
struct FGameData
{
        GENERATED_USTRUCT_BODY();
```

```
	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Game", meta=(ClampMin = "1",
ClampMax = "100"))
	int32 PlayersNum = 2;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX", meta=(ClampMin = "1",
ClampMax = "100"))
	FDecalData DecalData;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX", meta=(ClampMin = "1",
ClampMax = "10"))
	int32 RoundsNum = 4;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX", meta=(ClampMin = "5",
ClampMax = "300"))
	int32 RoundTime = 60; //in seconds

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
	FLinearColor DefaultTeamColor = FLinearColor::Red;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite)
	TArray<FLinearColor> TeamColors;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="VFX", meta=(ClampMin = "3",
ClampMax = "20"))
	int32 RespawnTime = 5; //in seconds
};

UENUM(BlueprintType)
enum class ESTUMatchState: uint8
{
	WaitingToStart = 0,
	InProgress,
	Pause,
	GameOver
};

DECLARE_MULTICAST_DELEGATE_OneParam(FOnMatchStateChangedSignature, ESTUMatchState);

USTRUCT(BlueprintType)
struct FLevelData
{
	GENERATED_USTRUCT_BODY();

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Game")
	FName LevelName = NAME_None;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Game")
	FName LevelDisplayName = NAME_None;

	UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Game")
	UTexture2D* LevelThumb = nullptr;
};

DECLARE_MULTICAST_DELEGATE_OneParam(FOnLevelSelectedSignature, const FLevelData&);
```

STUGameModeBase.h:

```
#pragma once

#include "CoreMinimal.h"
#include "AIController.h"
#include "STUCoreTypes.h"
#include "GameFramework/GameModeBase.h"
#include "STUGameModeBase.generated.h"

UCLASS()
class SHOOTTHEMUP_API ASTUGameModeBase : public AGameModeBase
{
	GENERATED_BODY()

public:
	ASTUGameModeBase();

	FOnMatchStateChangedSignature OnMatchStateChanged;

	virtual void StartPlay() override;
```

```
        virtual UClass* GetDefaultPawnClassForController_Implementation(AController*
InController) override;

        void Killed(AController* KillerController, AController* VictimController);

        FGameData GetGameData() const {return GameData;}
        int32 GetCurrentRoundNum() const {return  CurrentRound;}
        int32 GetRoundSecondsRemaining() const {return  RoundCountDown;}

        void RespawnRequest(AController* Controller);

        virtual bool SetPause(APlayerController* PC, FCanUnpause CanUnpauseDelegate =
FCanUnpause()) override;
        virtual bool ClearPause() override;

protected:
        UPROPERTY(EditDefaultsOnly, Category="Game")
        TSubclassOf<AAIController> AIControllerClass;

        UPROPERTY(EditDefaultsOnly, Category="Game")
        TSubclassOf<APawn> AIPawnClass;

        UPROPERTY(EditDefaultsOnly, Category="Game")
        FGameData GameData;

        UPROPERTY(EditAnywhere, Category="Game")
        int32 MinRoundTimeForRespawn = 10;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Game")
        int32 TeamsNum = 2;

        UPROPERTY(EditDefaultsOnly, BlueprintReadWrite, Category="Game")
        bool LastManStandingGameMode = false;

private:
        void SpawnBots();
        void StartRound();
        void GameTimerUpdate();
        void ResetPlayers();
        void ResetOnePlayer(AController* Controller);
        void CreateTeamsInfo();
        FLinearColor DetermineColorByTeamID(int32 TeamID) const;
        void SetPlayerColor(AController* Controller);

        int32 CurrentRound = 1;
        int32 RoundCountDown = 0;
        FTimerHandle GameRoundTimerHandle;
        TArray<FString> BotNames = {"Bot Yarik", "Bot Mosk", "Bot Kostya", "Bot Vanya", "Bot
Ded", "Bot Bond", "Bot Kas", "Bot Sanya", "Bot Anton"};

        void LogPlayerInfo();
        void StartRespawn(AController* Controller);
        void LastManStandingGameOver();
        void GameOver();

        ESTUMatchState MatchState = ESTUMatchState::WaitingToStart;
        void SetMatchState(ESTUMatchState State);

        void StopAllFire();
};
```

STUGameModeBase.cpp:

```
#include "STUGameModeBase.h"

#include "EngineUtils.h"
#include "STUBaseCharacter.h"
#include "STUGameHUD.h"
#include "STUGameInstance.h"
#include "STUPlayerController.h"
#include "STUPlayerState.h"
#include "STURespawnComponent.h"
#include "STUUtils.h"

ASTUGameModeBase::ASTUGameModeBase()
{
```

```
	DefaultPawnClass = ASTUBaseCharacter::StaticClass();
	PlayerControllerClass = ASTUPlayerController::StaticClass();
	HUDClass = ASTUGameHUD::StaticClass();
	PlayerStateClass = ASTUPlayerState::StaticClass();
}

void ASTUGameModeBase::StartPlay()
{
	Super::StartPlay();
	//UE_LOG(LogTemp, Warning, TEXT("%s"), *GetWorld()->GetGameInstance<USTUGameInstance>()-
>TestString)
	SpawnBots();
	CreateTeamsInfo();
	CurrentRound = 1;
	StartRound();
	SetMatchState(ESTUMatchState::InProgress);
}

UClass* ASTUGameModeBase::GetDefaultPawnClassForController_Implementation(AController*
InController)
{
	if(InController && InController->IsA<AAIController>())
	{
		return AIPawnClass;
	}
	return Super::GetDefaultPawnClassForController_Implementation(InController);
}

void ASTUGameModeBase::Killed(AController* KillerController, AController* VictimController)
{
	const auto KillerPlayerState = KillerController ? Cast<ASTUPlayerState>(KillerController-
>PlayerState) : nullptr;
	const auto VictimPlayerState = VictimController ? Cast<ASTUPlayerState>(VictimController-
>PlayerState) : nullptr;
	if(KillerPlayerState)
	{
		KillerPlayerState->AddKill();
	}
	if(VictimPlayerState)
	{
		VictimPlayerState->AddDeath();
	}
	StartRespawn(VictimController);
}

void ASTUGameModeBase::RespawnRequest(AController* Controller)
{
	ResetOnePlayer(Controller);
}

bool ASTUGameModeBase::SetPause(APlayerController* PC, FCanUnpause CanUnpauseDelegate)
{
	const auto PauseSet = Super::SetPause(PC, CanUnpauseDelegate);
	if(PauseSet)
	{
		StopAllFire();
		SetMatchState(ESTUMatchState::Pause);
	}
	return PauseSet;
}

bool ASTUGameModeBase::ClearPause()
{
	const auto PauseCleared = Super::ClearPause();
	if(PauseCleared)
	{
		SetMatchState(ESTUMatchState::InProgress);
	}
	return PauseCleared;
}

void ASTUGameModeBase::SpawnBots()
{
	if(!GetWorld()) return;

	for(int32 i = 0; i < GameData.PlayersNum - 1; ++i)
	{
		FActorSpawnParameters SpawnInfo;
```

```
                SpawnInfo.SpawnCollisionHandlingOverride =
ESpawnActorCollisionHandlingMethod::AlwaysSpawn;
                const auto STUAIController = GetWorld()-
>SpawnActor<AAIController>(AIControllerClass, SpawnInfo);
                RestartPlayer(STUAIController);
        }
}

void ASTUGameModeBase::StartRound()
{
        RoundCountDown = GameData.RoundTime;
        GetWorldTimerManager().SetTimer(GameRoundTimerHandle, this,
&ASTUGameModeBase::GameTimerUpdate, 1.0f, true);
}

void ASTUGameModeBase::GameTimerUpdate()
{
        //UE_LOG(LogTemp, Warning, TEXT("Time: %i / Round: %i / %i"), RoundCountDown,
CurrentRound, GameData.RoundsNum);
        if(--RoundCountDown == 0)
        {
                GetWorldTimerManager().ClearTimer(GameRoundTimerHandle);
                if(CurrentRound + 1 <= GameData.RoundsNum)
                {
                        ++CurrentRound;
                        ResetPlayers();
                        StartRound();
                }
                else
                {
                        GameOver();
                }
        }
        if(LastManStandingGameMode)
        {
                if(GetWorld()->GetNumControllers() == 1)
                {
                        GameOver();
                }
                else if(GetWorld()->GetNumControllers() == 2)
                {
                        LastManStandingGameOver();
                }
        }
}

void ASTUGameModeBase::ResetPlayers()
{
        if(!GetWorld()) return;
        for(auto It = GetWorld()->GetControllerIterator(); It; ++It)
        {
                ResetOnePlayer(It->Get());
        }
}

void ASTUGameModeBase::ResetOnePlayer(AController* Controller)
{
        const auto PlayerState = Cast<ASTUPlayerState>(Controller->PlayerState);
        if(!LastManStandingGameMode || (LastManStandingGameMode && PlayerState->GetDeathsNum() <
3))
        {
                if(Controller && Controller->GetPawn())
                {
                        Controller->GetPawn()->Reset();
                }
                RestartPlayer(Controller);
                SetPlayerColor(Controller);
        }
        else if(LastManStandingGameMode && PlayerState->GetTeamID() != 1 && PlayerState-
>GetDeathsNum() == 3)
        {
                Controller->Destroy();
        }
}

void ASTUGameModeBase::CreateTeamsInfo()
{
        if(!GetWorld()) return;
```

```
	int32 TeamID = 1;
	for(auto It = GetWorld()->GetControllerIterator(); It; ++It)
	{
		const auto Controller = It->Get();
		if(!Controller) continue;
		const auto PlayerState = Cast<ASTUPlayerState>(Controller->PlayerState);
		if(!PlayerState) continue;
		PlayerState->SetTeamID(TeamID);
		PlayerState->SetTeamColor(DetermineColorByTeamID(TeamID));
		PlayerState->SetPlayerName(Controller->IsPlayerController() ? "Player" :
BotNames[It.GetIndex() - 1]);
		SetPlayerColor(Controller);
		if(TeamID + 1 > TeamsNum) TeamID = 1;
		else
		{
			TeamID += 1;
		}
	}
}

FLinearColor ASTUGameModeBase::DetermineColorByTeamID(int32 TeamID) const
{
	if(TeamID - 1 < GameData.TeamColors.Num())
	{
		return GameData.TeamColors[TeamID-1];
	}
	UE_LOG(LogTemp, Warning, TEXT("No color!"))
	return GameData.DefaultTeamColor;
}

void ASTUGameModeBase::SetPlayerColor(AController* Controller)
{
	if(!Controller) return;
	const auto Character = Cast<ASTUBaseCharacter>(Controller->GetPawn());
	if(!Character) return;

	const auto PlayerState = Cast<ASTUPlayerState>(Controller->PlayerState);
	if(!PlayerState) return;
	Character->SetPlayerColor(PlayerState->GetTeamColor());
}

void ASTUGameModeBase::LogPlayerInfo()
{
	if(!GetWorld()) return;
	for(auto It = GetWorld()->GetControllerIterator(); It; ++It)
	{
		const auto Controller = It->Get();
		if(!Controller) continue;
		const auto PlayerState = Cast<ASTUPlayerState>(Controller->PlayerState);
		if(!PlayerState) continue;
		PlayerState->LogInfo();
	}
}

void ASTUGameModeBase::StartRespawn(AController* Controller)
{
	const auto RespawnAvailable = RoundCountDown > MinRoundTimeForRespawn +
GameData.RespawnTime;
	if(!RespawnAvailable) return;
	const auto RespawnComponent =
STUUtils::GetSTUPlayerComponent<USTURespawnComponent>(Controller);
	if(!RespawnComponent) return;
	RespawnComponent->Respawn(GameData.RespawnTime);
}

void ASTUGameModeBase::LastManStandingGameOver()
{
	if(!GetWorld()) return;
	for(auto It = GetWorld()->GetControllerIterator(); It; ++It)
	{
		const auto Controller = It->Get();
		if(!Controller) continue;
		const auto PlayerState = Cast<ASTUPlayerState>(Controller->PlayerState);
		if(!PlayerState) continue;
		if(PlayerState->GetDeathsNum() >= 3)
		{
			if(!Controller->IsPlayerController())
			{
```

```
                        Controller->Destroy();
                }
                else
                {
                        PlayerState->SetInvisible(true);
                }
                GameOver();
        }
    }
}

void ASTUGameModeBase::GameOver()
{
    //UE_LOG(LogTemp, Warning, TEXT("GAME OVER!"))
    //LogPlayerInfo();
    for(auto Pawn : TActorRange<APawn>(GetWorld()))
    {
        if(Pawn)
        {
            Pawn->TurnOff();
            Pawn->DisableInput(nullptr);
        }
    }
    SetMatchState(ESTUMatchState::GameOver);
}

void ASTUGameModeBase::SetMatchState(ESTUMatchState State)
{
    if(MatchState == State) return;
    MatchState = State;
    OnMatchStateChanged.Broadcast(MatchState);
}

void ASTUGameModeBase::StopAllFire()
{
    for(auto Pawn : TActorRange<APawn>(GetWorld()))
    {
        const auto WeaponComponent =
STUUtils::GetSTUPlayerComponent<USTUWeaponComponent>(Pawn);
        if(!WeaponComponent) continue;

        WeaponComponent->StopFire();
        WeaponComponent->Zoom(false);
    }
}
```

STUGameInstance.h:

```
#pragma once

#include "CoreMinimal.h"
#include "STUCoreTypes.h"
#include "Engine/GameInstance.h"
#include "STUGameInstance.generated.h"

UCLASS()
class SHOOTTHEMUP_API USTUGameInstance : public UGameInstance
{
    GENERATED_BODY()

public:
    FLevelData GetStartupLevel() const { return StartupLevel;}
    void SetStartupLevel(const FLevelData& Data) { StartupLevel = Data;}

    TArray<FLevelData> GetLevelsData() const {return  LevelsData;}

    FName GetMenuLevelName() const { return MenuLevelName;}

    void ToggleVolume();

protected:
    UPROPERTY(EditDefaultsOnly, Category="Game", meta=(ToolTip = "Level names must be
unique!"))
    TArray<FLevelData> LevelsData;

    UPROPERTY(EditDefaultsOnly, Category="Game")
    FName MenuLevelName = NAME_None;
```

```
        UPROPERTY(EditDefaultsOnly, Category="Sound")
        USoundClass* MasterSoundClass;

private:
        FLevelData StartupLevel;
};
```

STUGameInstance.cpp:

```
#include "STUGameInstance.h"
#include "Sound/STUSoundFuncLib.h"

void USTUGameInstance::ToggleVolume()
{
        USTUSoundFuncLib::ToggleSoundClassVolume(MasterSoundClass);
}
```