Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Балицького Богдана Ігоровича
Прізвище, ім'я, по батькові студента(ки)

на здобуття ступеня вищої освіти Бакалавра

Багатокористувацький ігровий застосунок у жанрі «шутер від першої особи»
Назва теми

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗ. 200242.01.01.ПЗ

Виконав студент IV курсу, група ІПЗ-20-1 _____ Богдан БАЛИЦЬКИЙ
Підпис Ім'я, ПРІЗВИЩЕ

Керівник канд. техн. наук, доцент _____ Галина РАДЕЛЬЧУК
Науковий ступінь, звання Підпис Ім'я, ПРІЗВИЩЕ

Нормоконтролер доцент _____ Галина РАДЕЛЬЧУК
Науковий ступінь, звання Підпис Ім'я, ПРІЗВИЩЕ

До захисту допускаю:
Завідувач кафедри інженерії програмного забезпечення _____ Леонід БЕДРАТЮК
Підпис Ім'я, ПРІЗВИЩЕ

12 червня 2024 р.

Хмельницький 2024

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Інженерії програмного забезпечення

Рівень вищої освіти Перший (бакалаврський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
_____ Л. П. Бедратюк
02 01 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Балицькому Богдану Ігоровичу
Прізвище, ім'я, по батькові студента

1. Тема роботи Багатокористувацький ігровий застосунок у жанрі «шутер від першої особи»

Керівник роботи Радельчук Галина Іванівна, канд. техн. наук, доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 08.01.2024 р. № 6-КП

2. Строк подання студентом роботи на кафедру 01.06.2024 р.
3. Вихідні дані до роботи Матеріали переддипломної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
   Аналіз предметної області та постановка задачі
   Проєктування програмного забепечення
   Програмна реалізація та тестування програмного забезпечення
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
   Три креслення у форматі А3:
   1. Діаграма класів
   2. DFD-діаграма
   3. Діаграма процесів гри

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Радельчук Г. І., доцент	03.06.2024	10.06.2024.
Антиплагіат	Форкун Ю. В., доцент	30.05.24	06.06.24

7. Дата видачі завдання « 02 » січня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1 Дослідження суб'єкта господарювання відповідно до теми КвР; визначення задач та вимог до ПЗ, розроблення технічного завдання	1-й – 3-й тиждень	
2 Проєктування та розроблення загальної архітектури і структури ПЗ, інтерфейсу користувача; вибір засобів реалізації ПЗ	4-й – 8-й тиждень	
3 Програмна реалізація та тестування ПЗ	9-й – 12-й тиждень	
4 Написання тексту пояснювальної записки та розроблення графічних матеріалів	13-й – 14-й тиждень	
Попередній захист КвР	Травень	згідно із графіком
6 Остаточне коригування КвР з урахуванням зауважень керівника; оформлення КвР як документа відповідно до вимог	15-й тиждень	
Отримання супровідних документів (відгуку керівника, рецензії, довідок про перевірку на плагіат); проходження нормоконтролю; брошурування (зшиття) пояснювальної записки	16-й тиждень	
Здача КвР на кафедру; підготовка КвР для розміщення у репозитарії університету; підготовка до захисту та захист КвР	З 01.06.2024	

Студент _____ Богдан БАЛИЦЬКИЙ
Підпис Ініціали, ПРІЗВИЩЕ

Керівник роботи _____ Галина РАДЕЛЬЧУК
Підпис Ініціали, ПРІЗВИЩЕ

АНОТАЦІЯ

Тема кваліфікаційної роботи: Багатокористувацький ігровий застосунок у жанрі «шутер від першої особи».

Автор роботи: Балицький Богдан Ігорович.

Керівник роботи: Радельчук Галина Іванівна.

Пояснювальна записка: 73 с., 20 рис., 2 дод., 32 джерела.

Графічна частина: 3 креслення ф. А3.

Метою даної дипломної роботи є розроблення багатокористувацького ігрового застосунку у жанрі шутер від першої особи, що забезпечить динамічний функціонал, у якому передбачено підключення гравців, динамічну систему руху та зрозумілий інтерфейс.

У кваліфікаційній роботі проведено аналіз предметної області та її інформаційного забезпечення, визначені функціональні вимоги до програмної системи, розроблена загальна архітектура застосунку, сформована структура гри. Для розробки програмного продукту використано мову C++ та ігровий рушій Unreal Engine 5. За допомогою цих засобів розроблено програмне забезпечення, яке має за мету розважити користувача.

10.06.2024
Дата

Підпис

ВІДОМІСТЬ ДОКУМЕНТІВ

№ рядка	Формат	Позначення документа	Найменування документа	К-сть аркушів	№ екз.	Примітка
			Текстові документи			
1	А4	КвРІПЗ.200242.01.01.ПЗ	Пояснювальна записка	73		
2	А4		Завдання на кваліфіка-	1		
			ційну роботу			
3	А4		Анотація	1		
			Графічні документи			
4	А3	КвРІПЗ.200242.01.01.Е8	Діаграма класів	1		
5	А3	КвРІПЗ.200242.01.01.Е8	DFD-діаграма	1		
6	А3	КвРІПЗ.200242.01.01.Е8	Діаграма процесів гри	1		

Змн.	Арк.	№ докум.	Підпис	Дата	КвРІПЗ.200242.01.01.ВД		
Виконав		Балицький Б.І		10.06	Багатокористувацький ігровий застосунок у жанрі «шутер від першої особи» Відомість документів	Літ.	Арк. 1 / Аркушів 1
Керівник		Радельчук Г.І		10.06			
Н. контр.		Радельчук Г. І.		10.06		ХНУ, ІПЗ-20-1	
Зав. каф.		Бедратюк Л.М.		10.06			

ЗМІСТ

					КвРІПЗ.200242.01.01.ПЗ		
Змн.	Арк.	№ докум.	Підпис	Дата			
Виконав		Балицький Б.І.		10.06	Багатокористувацький ігровий застосунок у жанрі «шутер від першої особи» Пояснювальна записка	Літ.	Арк. 4 / Аркушів 73
Керівник		Радельчук Г.І.		10.06			
Рецензент						ХНУ, ІПЗ-20-1	
Н. контр.		Радельчук Г. І.		10.06			
Зав. каф.		Бедратюк Л.А.		10.06			

ВСТУП

У сучасному світі ігрова індустрія є однією з найбільш динамічних та прибуткових галузей, що стрімко розвивається. Ігри стали важливою частиною культури та дозвілля мільйонів людей по всьому світу, значно впливаючи на соціальні, економічні та технологічні аспекти суспільства. Особливо популярними є багатокористувацькі ігрові застосунки, які дозволяють гравцям взаємодіяти один з одним у віртуальному просторі, створюючи унікальні та захоплюючі ігрові світи. Жанр шутерів від першої особи (First-Person Shooter, FPS) займає особливе місце серед таких ігор, завдяки своїй динамічності, реалістичній графіці та високому рівню залученості гравців.

Новизна даної роботи полягає у розробці багатокористувацького ігрового застосунку, що поєднує в собі сучасні технологічні досягнення та інноваційні підходи до геймдизайну. Практична значущість роботи полягає у створенні платформи, яка може бути потеційно використана для подальшого розвитку та комерціалізації, а також у здобутті навичок та знань, що можуть бути застосовані в інших проєктах ігрової індустрії.

Основна мета створення ігрових застосунків полягає в наданні користувачам унікального і захоплюючого ігрового досвіду, що поєднує розважальні, освітні та соціальні аспекти. Ігрові застосунки дозволяють:

Розвага: Забезпечити високий рівень розважального контенту, що дозволяє гравцям відпочивати та насолоджуватися віртуальними пригодами.

Соціальна взаємодія: Створити платформи для спілкування та співпраці між гравцями з різних куточків світу.

Розвиток навичок: Сприяти розвитку когнітивних, моторних та стратегічних навичок.

Освіта: Використовувати ігрові механіки для навчання та тренування в різних галузях знань.

Метою даної дипломної роботи є розроблення багатокористувацького ігрового застосунку у жанрі шутер від першої особи, що забезпечить динамічний функціонал, у якому передбачено підключення гравців, динамічну систему руху та зрозумілий інтерфейс. Для досягнення цієї мети необхідно вирішити низку науково-практичних завдань:

– провести аналіз існуючих рішень у жанрі FPS та виявити переваги і недоліки таких застосунків.

– здійснити постановку задачі та обґрунтувати перелік задач для реалізації застосунку;

– розробити концепцію гри, включаючи геймплейні механіки, інтерфейс користувача та зброю.

– сформувати вимоги до застосунку;

– вибрати відповідні інструменти та технології для реалізації проєкту.

– вибрати засоби шаблони архітектури та окреслити найкращий варіант;

– описати залежності;

– окреслити структуру і функціональне призначення модулів системи, їх тісних взаємозв’язок;

– здійснити декомпозицію різних рівнів;

– виконати програмну реалізацію та провести тестування для оцінки її якості та стабільності.

– оптимізувати мережеву складову гри для забезпечення стабільної роботи при великій кількості одночасних гравців.

Інформаційна база дослідження. Інформаційною базою дослідження є наукові праці вітчизняних та зарубіжних вчених, авторський матеріал.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Змістовний аналіз предметної області, її структурних та функціональних особливостей

Предметна область багатокористувацьких ігрових застосунків у жанрі шутер від першої особи (First-Person Shooter, FPS) [1,2] має свої особливості та вимоги. Розуміння цих аспектів є важливим для успішної розробки програмного забезпечення, яке відповідатиме сучасним стандартам і потребам гравців. Вивчення предметної області проводиться для визначення проблем та невирішених питань з точки зору впровадження інформаційних технологій, автоматизації процесів обробки та передачі інформації тощо.

Жанр FPS є одним з найбільш популярних і розвинутих у сучасній ігровій індустрії. Багатокористувацькі шутери від першої особи відрізняються високим рівнем інтерактивності, динамічним геймплеєм та реалістичною графікою. Основні елементи та процеси в цій предметній області включають:

Ігровий клієнт: Відповідає за відображення графіки, обробку введення від користувача та взаємодію з сервером.

Ігровий сервер: Виконує обробку запитів від клієнтів, координує дії гравців та забезпечує синхронізацію даних у реальному часі.

Мережеві протоколи: використовуються для передачі даних між клієнтами та сервером, забезпечуючи низьку затримку та високу надійність з'єднання в грі.

База даних: Зберігає інформацію про гравців, їхні досягнення, налаштування та інші важливі дані.

Однією з головних рис FPS є перспектива від першої особи, коли гравець бачить світ очима свого персонажа. Це підсилює занурення в ігровий процес і дозволяє відчути себе безпосереднім учасником подій.

Шутери від першої особи відомі своїм швидким і напруженим геймплеєм. Гравець повинен швидко реагувати на ворогів, точно стріляти та ефективно

використовувати доступні ресурси. Це вимагає хороших рефлексів і координації рухів гравця.

FPS пропонують гравцям широкий арсенал зброї, від пістолетів і автоматів до гранат і снайперських гвинтівок. Крім того, гравці можуть використовувати різноманітне спорядження, таке як броня, медичні набори та спеціальні гаджети.

Сучасні FPS часто пропонують широкий вибір налаштувань персонажів, що дозволяє гравцям індивідуалізувати свій зовнішній вигляд, зброю та спорядження. Це додає глибини геймплею та дозволяє гравцям виразити свою унікальність серед інших.

Однією з головних проблем FPS є забезпечення балансу між різними класами персонажів та видами зброї. Це важливо для підтримання чесності та задоволення гравців.

З розвитком технологій графіка в FPS стає все більш реалістичною. Використання сучасних графічних рушіїв дозволяє створювати детально пророблені оточення, реалістичні ефекти освітлення та текстури високої якості.

Боротьба з шахрайством є ще однією серйозною проблемою. Розробники постійно вдосконалюють античит-системи, щоб захистити гру від недобросовісних гравців.

Забезпечення стабільної роботи гри на різних платформах і пристроях вимагає великих зусиль. Розробники мають оптимізувати графіку та інші ресурси, щоб забезпечити плавний ігровий процес.

Незважаючи на значні досягнення у розробці багатокористувацьких FPS, існує низка проблем та викликів, які потребують вирішення:

Оптимізація продуктивності: Забезпечення стабільної роботи гри на різних платформах та пристроях є важливим завданням. Високі вимоги до графіки та фізики вимагають ефективного використання ресурсів системи.

Забезпечення мережевої стабільності: Велика кількість одночасних гравців потребує надійної мережевої інфраструктури. Висока затримка або втрати пакетів можуть негативно вплинути на ігровий досвід.

Баланс геймплею: Підтримка рівноваги між різними класами персонажів та видами зброї є критичною для забезпечення чесної гри. Це вимагає постійного моніторингу та налаштування.

Безпека та античит-системи: Захист від шахрайства та несанкціонованого доступу є важливим аспектом, що забезпечує чесність та захист даних гравців.

На основі аналізу предметної області визначено основну проблему, яка буде вирішена за допомогою майбутнього програмного забезпечення: оптимізація багатокористувацької взаємодії та забезпечення стабільної роботи ігрового застосунку.

Змістовний аналіз предметної області багатокористувацьких ігрових застосунків у жанрі шутер від першої особи дозволив визначити основні структурні та функціональні особливості, а також проблеми, які необхідно вирішити. Використання модельних представлень, таких як IDEF0, допомогло візуалізувати основні процеси та визначити ключові аспекти для подальшої розробки програмного забезпечення. Результати цього аналізу слугуватимуть основою для створення ефективного та стабільного ігрового застосунку, що задовольнить потреби сучасних гравців.

1.2 Аналіз наявного програмно-технічного забезпечення предметної області

Для ефективного розроблення багатокористувацьких шутерів від першої особи необхідно ретельно проаналізувати наявне програмно-технічне забезпечення, яке використовується у цій предметній області. Одним із яскравих прикладів є Team Fortress 2 [3-5] (TF2) – популярний багатокористувацький шутер, розроблений компанією Valve [6]. У цьому розділі розглянемо основні аспекти програмно-технічного забезпечення TF2 (рисунок 1.1), включаючи його архітектуру, ігровий рушій, мережеву інфраструктуру та інші ключові компоненти гри.

Рисунок 1.1 – Team Fortress 2

Архітектура Team Fortress 2 складається з кількох основних компонентів. Ігровий клієнт відповідає за відображення графіки та звуку, обробляє введення від користувача за допомогою клавіатури та миші, а також відправляє команди на сервер та отримує від нього оновлення ігрового стану. Ігровий сервер обробляє запити від клієнтів, координує дії гравців, обробляючи їхні команди та підтримуючи синхронізацію ігрового процесу, а також зберігає дані про ігровий стан, включаючи позиції гравців, їхнє здоров'я, використання зброї тощо.

Мережеві протоколи у Team Fortress 2 використовують UDP протокол для швидкої передачі даних між клієнтом і сервером. Впровадження алгоритмів компенсації затримки забезпечує плавний ігровий процес навіть при високій затримці мережі.

Team Fortress 2, як і інші багатокористувацькі шутери, стикається з кількома технічними викликами. Одним із основних викликів є оптимізація продуктивності. Розробники постійно працюють над оптимізацією гри для різних апаратних конфігурацій, використовуючи оновлення і патчі для покращення продуктивності та стабільності гри.

Балансування геймплею також є важливим аспектом. Регулярне оновлення ігрового контенту допомагає підтримувати баланс між різними класами та видами зброї. Для цього проводяться ігрові тести і аналізується зворотний зв'язок від спільноти гравців, що дозволяє швидко реагувати на будь-які дизбаланси.

Забезпечення мережевої стабільності є ще одним критичним завданням. Використання передових алгоритмів допомагає зменшити затримку та втрати пакетів, що забезпечує плавний ігровий процес. Постійний моніторинг серверів і мережевої інфраструктури дозволяє виявляти та вирішувати проблеми, що виникають, підтримуючи стабільну роботу гри.

Аналіз програмно-технічного забезпечення Team Fortress 2 демонструє, що успіх багатокористувацьких шутерів від першої особи залежить від багатьох факторів, включаючи потужний ігровий рушій, ефективну мережеву інфраструктуру та добре продумані ігрові механіки. Досвід розробників TF2 може бути корисним для створення нових ігор у цьому жанрі, враховуючи їхні рішення щодо оптимізації продуктивності, балансування геймплею та забезпечення мережевої стабільності.

Overwatch [7-9], розроблений компанією Blizzard Entertainment [10], використовує власний ігровий рушій, що забезпечує високу якість графіки та ефективність роботи. Рушій підтримує сучасні графічні технології, такі як динамічні тіні, шейдери та високоякісні текстури, що дозволяє створювати деталізовані ігрові світи. Він також інтегрує передові фізичні моделі, які забезпечують реалістичну взаємодію об'єктів у грі, і розвинену систему анімації, що дозволяє створювати плавні та природні рухи персонажів.

Архітектура Overwatch включає декілька ключових компонентів. Ігровий клієнт відповідає за відображення графіки та звуку, обробляє введення від користувача за допомогою клавіатури та миші, а також надсилає команди на сервер і отримує оновлення ігрового стану. Ігровий сервер обробляє запити від клієнтів, координує дії гравців, обробляючи їхні команди, підтримуючи

синхронізацію ігрового процесу та зберігає дані про ігровий стан, включаючи позиції гравців, їхнє здоров'я, використання зброї та здібностей.

Overwatch (рисунок 1.2) використовує мережеві протоколи, що базуються на UDP, для швидкої передачі даних між клієнтом і сервером. Впровадження алгоритмів компенсації затримки забезпечує плавний ігровий процес навіть при високій затримці мережі. Це дозволяє мінімізувати негативний вплив затримок і забезпечити стабільний геймплей.

Рисунок 1.2 - Overwatch

Одним із основних технічних викликів для Overwatch є оптимізація продуктивності. Розробники Blizzard постійно працюють над покращенням гри для різних апаратних конфігурацій, використовуючи оновлення і патчі для забезпечення високої продуктивності та стабільності гри. Балансування геймплею є ще одним важливим аспектом, який включає регулярне оновлення ігрового контенту для підтримки балансу між різними персонажами та їхніми здібностями. Команда розробників проводить ігрові тести і аналізує зворотний зв'язок від спільноти гравців, що дозволяє швидко реагувати на дисбаланси.

Забезпечення мережевої стабільності є ще одним критичним завданням. Використання передових алгоритмів для зменшення затримки та втрат пакетів допомагає забезпечити плавний ігровий процес. Постійний моніторинг серверів і мережевої інфраструктури дозволяє виявляти та вирішувати проблеми, що виникають, підтримуючи стабільну роботу гри.

Аналіз програмно-технічного забезпечення Overwatch демонструє, що успіх багатокористувацьких шутерів від першої особи залежить від потужного ігрового рушія, ефективної мережевої інфраструктури та добре продуманих ігрових механік. Досвід розробників Overwatch може бути корисним для створення нових ігор у цьому жанрі, враховуючи їхні рішення щодо оптимізації продуктивності, балансування геймплею та забезпечення мережевої стабільності.

1.3 Визначення функціональних та нефункціональних вимог до програмного забезпечення

Визначення вимог до програмного забезпечення є ключовим етапом у процесі розробки будь-якої системи. Для багатокористувацького ігрового застосунку у жанрі шутер від першої особи ці вимоги допомагають сформувати чітке уявлення про те, яким має бути кінцевий продукт, які функції він повинен виконувати і які характеристики він має мати. Вимоги до програмного забезпечення поділяються на функціональні та нефункціональні.

Функціональні вимоги визначають, що система повинна робити, які функції та завдання вона має виконувати. Для багатокористувацького шутера від першої особи основні функціональні вимоги включають:

– гравці повинні мати можливість приєднуватися до існуючих ігор або створювати нові;

– повинна бути реалізована система управління персонажем від першої особи з можливістю стрільби, пересування, взаємодії з об'єктами тощо;

– система повинна обробляти та зберігати дані про ігровий стан у реальному часі, включаючи позиції гравців, їхнє здоров'я, використання зброї та інші події.

Нефункціональні вимоги визначають характеристики системи, які стосуються її якості, продуктивності, надійності тощо. Для багатокористувацького шутера від першої особи основні нефункціональні вимоги включають:

– система повинна забезпечувати стабільну частоту кадрів не менше 60 FPS на рекомендованих апаратних конфігураціях;

– серверна частина повинна обробляти запити від гравців у реальному часі з мінімальною затримкою;

– система повинна бути стійкою до збоїв і забезпечувати безперебійну роботу;

– інтерфейс користувача повинен бути інтуїтивно зрозумілим і простим у використанні.

Визначення функціональних та нефункціональних вимог до багатокористувацького ігрового застосунку у жанрі шутер від першої особи є критично важливим етапом у процесі його розробки. Функціональні вимоги забезпечують розуміння того, які саме завдання і функції повинна виконувати система, тоді як нефункціональні вимоги визначають якісні характеристики програмного забезпечення, такі як продуктивність, надійність, безпека та зручність використання. Врахування цих вимог дозволяє створити стабільний, безпечний та привабливий для користувачів продукт.

Use Case. Діаграма (рисунок 1.3) допомагає описати, як система взаємодіє з різними акторами (користувачами або іншими системами) та які функціональні можливості системи доступні цим акторам.

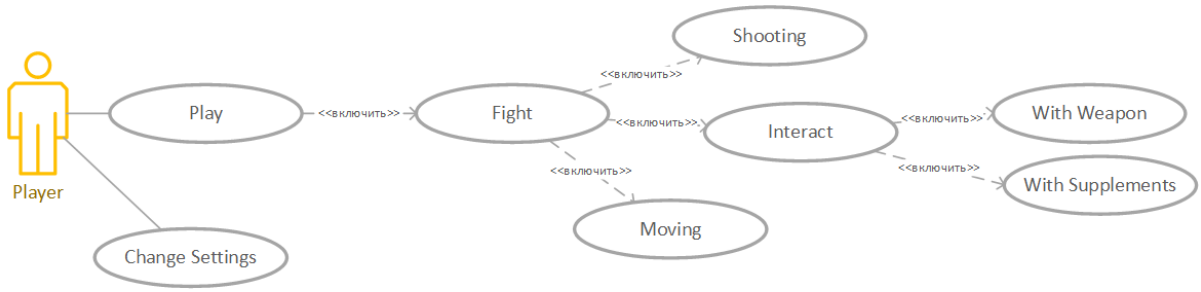


Рисунок 1.3 – Діаграма варіантів використання

У багатокористувацькій грі жанру FPS є лише один користувач, який виконує ролі, представлені на діаграмі. Цей користувач (актор) на діаграмі позначений як «Player». Діаграма варіантів використання (use case diagram) відображає основні дії, які може виконувати гравець у грі.

Гравець є центральною фігурою на діаграмі і виконує основні дії в грі. Перш за все, гравець може почати гру, що є основним варіантом використання, позначеним як «Play». Це охоплює всі дії, пов'язані з участю в ігровому процесі. Окрім цього, гравець має можливість змінювати налаштування гри, наприклад, графічні параметри, звук та управління, що представлено варіантом «Change Settings».

Коли гравець бере участь у грі, він може виконувати бойові дії, що відображено варіантом «Fight». Цей варіант включає взаємодію з різними об'єктами у грі, стрільбу та пересування. Взаємодія з об'єктами деталізована у варіанті «Interact», який включає взаємодію з зброєю та іншими предметами. Дії зі зброєю специфіковані варіантом «With Weapon», що охоплює підбирання, використання та перезарядку зброї. Взаємодія з іншими предметами, такими як аптечки та боєприпаси, відображена у варіанті «With Supplements».

Стрільба є важливою складовою бойових дій і представлена варіантом «Shooting», який деталізує процес стрільби по ворогах або цілях. Пересування гравця у грі, що включає біг, стрибки та хования за укриттями, відображено у варіанті «Moving». Ці дії є невід'ємною частиною бойового процесу і дозволяють гравцеві ефективно виконувати завдання у грі.

Таким чином, діаграма варіантів використання наочно демонструє, як гравець взаємодіє з різними аспектами гри, і як різні дії пов'язані між собою, забезпечуючи комплексний і захоплюючий ігровий процес.

1.4 Висновки. Постановка задачі

У процесі розробки багатокористувацького ігрового застосунку в жанрі шутер від першої особи важливо приділити особливу увагу всебічному аналізу предметної області та чіткому визначенню функціональних і нефункціональних вимог до програмного забезпечення. Підготовка до розробки такого складного програмного продукту передбачає ретельне вивчення існуючих рішень на ринку, їхніх сильних і слабких сторін, а також технологій, що забезпечують високу якість геймплею та користувацького досвіду.

На прикладі відомих багатокористувацьких шутерів, таких як Team Fortress 2 і Overwatch, було детально проаналізовано програмно-технічне забезпечення, яке використовується в цих іграх. Це включало огляд ігрових рушіїв, які забезпечують високоякісну графіку та фізику, архітектурні особливості систем, які координують взаємодію між клієнтами і серверами, а також мережевих протоколів, що гарантують стабільну та швидку передачу даних.

Визначення функціональних вимог дозволяє сформулювати чіткі завдання для розробників, які включають підключення користувачів, управління персонажами та інвентарем, а також реалізацію режиму ігрового процесу. Ці вимоги визначають, що саме повинно робити програмне забезпечення, щоб забезпечити цікавий ігровий досвід.

Нефункціональні вимоги стосуються якості програмного забезпечення. Вони включають продуктивність, масштабованість, надійність, безпеку, зручність використання та сумісність. Забезпечення відповідності цим вимогам

гарантує, що гра буде працювати стабільно та ефективно на різних пристроях, а користувачі отримають задоволення від її використання.

Діаграма варіантів використання, розроблена для багатокористувацької гри жанру FPS, показала, як користувач взаємодіє з різними аспектами гри. Гравець виконує основні дії, такі як участь у грі, зміну налаштувань, бойові дії, взаємодію з об'єктами, стрільбу та пересування. Ця діаграма допомагає розробникам зрозуміти ключові сценарії використання і створити інтуїтивно зрозумілий і привабливий інтерфейс користувача.

Підсумовуючи проведений аналіз та визначення вимог, можна стверджувати, що розроблення багатокористувацького ігрового застосунку у жанрі шутер від першої особи є складним, але захоплюючим завданням. Врахування функціональних і нефункціональних вимог, а також ретельний аналіз існуючих рішень і технологій, дозволяють створити якісний продукт, який буде відповідати очікуванням користувачів і забезпечувати стабільний і цікавий ігровий досвід.

Таким чином, успіх у розробці багатокористувацького шутера значною мірою залежить від ґрунтовної підготовки та вивчення предметної області, чіткого визначення вимог до програмного забезпечення та ефективної реалізації всіх ключових аспектів гри. Це забезпечить конкурентоспроможність продукту на ринку та задоволення гравців від використання нової захоплюючої гри.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Проєктування архітектури програмного забезпечення

У цьому підрозділі розглядається загальна архітектура програмного забезпечення багатокористувацької гри жанру FPS. Основна мета – визначити ключові компоненти та підсистеми, які складають програму, а також описати їх взаємодію та роль у загальній структурі проєкту. Архітектура повинна забезпечувати ефективну роботу програми, високу продуктивність, масштабованість та зручність для користувачів.

Для розробки багатокористувацької гри жанру FPS було обрано клієнт-серверну архітектуру (рисунок 2.1) де один з гравців виступає як хост (сервер), а інші гравці підключаються до нього як клієнти. Цей підхід дозволяє розподілити обчислювальне навантаження між гравцями та забезпечити гнучкість у виборі хосту, що може зменшити затримки та поліпшити якість з'єднання.

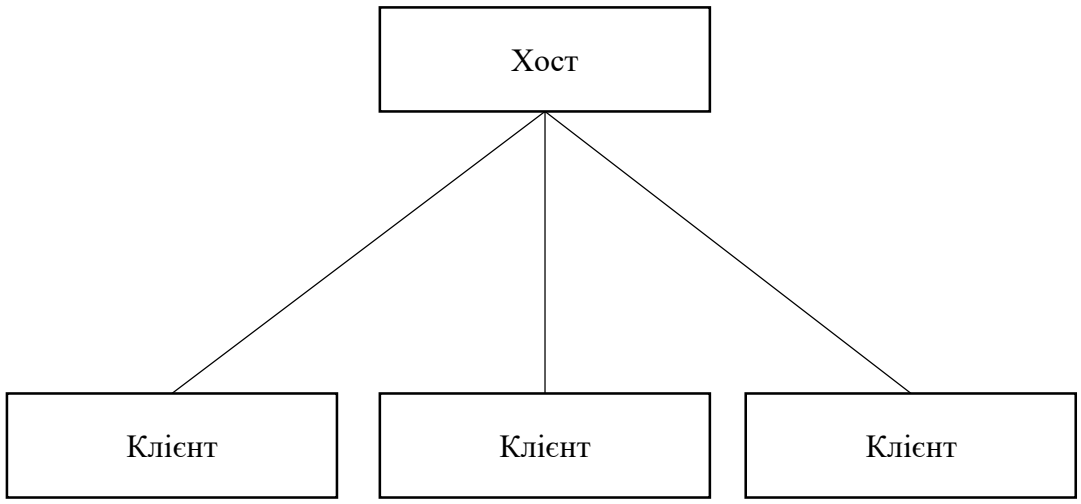


Рисунок 2.1 – Серверна архітектура

Хост - це один з гравців, який виступає як сервер. Він відповідає за координацію дій усіх гравців, обробку ігрової логіки, синхронізацію ігрового

стану, а також за комунікацію з іншими клієнтами. Хост зберігає дані про позиції гравців, їхнє здоров'я, стан зброї та інші важливі параметри.

Клієнти - це інші гравці, які підключаються до хоста. Клієнти відправляють дані про дії гравця (рух, стрільба, взаємодія з об'єктами) на хост та отримують оновлення ігрового стану від хоста. Клієнти відповідають за відображення графіки, обробку введення від користувача (клавіатура, миша, геймпад), та реалізацію локальної фізики.

Мережевий інтерфейс: забезпечує комунікацію між хостом і клієнтами. Використовується протокол UDP для швидкої передачі даних з мінімальною затримкою. Для забезпечення надійності ігрового процесу впроваджуються алгоритми компенсації затримок та обробки втрачених пакетів.

У цій архітектурі ігровий хост отримує дані про дії від кожного клієнта, обробляє ці дії, оновлює загальний ігровий стан і передає оновлені дані назад до всіх клієнтів. Це дозволяє всім учасникам бачити синхронізовану картину ігрового світу.

Клієнт: Відправляє на хост інформацію про дії гравця (рух, стрільба, взаємодія з об'єктами) через мережевий інтерфейс. Отримує від хоста оновлені дані про ігровий стан.

Хост: Обробляє запити від клієнтів, виконує обчислення, що стосуються ігрової логіки (наприклад, зіткнення, фізика), і підтримує синхронізацію ігрового стану між усіма клієнтами. Відправляє оновлення ігрового стану назад до клієнтів.

Централізований контроль: Сервер контролює всі аспекти гри, забезпечуючи централізоване управління ігровим процесом, даними та взаємодією між гравцями. Це дозволяє легко впроваджувати оновлення, патчі та новий контент, оскільки всі зміни можна внести на сервері.

Безпека: Оскільки сервер є єдиною точкою, яка має повний контроль над грою, це дозволяє зменшити ризик шахрайства та зломів. Сервер може

перевіряти всі дії клієнтів на відповідність правилам гри та відфільтровувати некоректні запити.

Синхронізація ігрового стану: Сервер підтримує синхронізацію ігрового стану між всіма клієнтами, забезпечуючи узгодженість ігрового процесу для всіх гравців. Це дозволяє уникнути проблем з розбіжністю ігрових даних між різними клієнтами.

Спрощена архітектура клієнта: гра може бути більш легкою і менш вимогливою до ресурсів.

Залежність від сервера: Якщо сервер виходить з гри або має проблеми з продуктивністю, це впливає на всіх клієнтів. Сервер стає критичною точкою відмови, і будь-яка його несправність може призвести до непрацездатності всієї системи.

Затримки (латенція): Віддаленість клієнтів від сервера може спричиняти затримки в обміні даними, що негативно впливає на ігровий процес, особливо в реалістичних іграх жанру FPS, де кожна мілісекунда має значення.

Обмеження на масштабування: хоча клієнт-серверна архітектура дозволяє масштабування, воно має свої межі. Існують фізичні та технічні обмеження на кількість одночасних підключень, які може обробляти один сервер.

Складність мережевого програмування: розроблення надійного і ефективного мережевого коду є складним завданням, яке потребує високої кваліфікації. Врахування затримок, втрат пакетів, синхронізації ігрового стану та інших мережевих проблем потребує значних зусиль і знань.

Розроблення багатокористувацького шутеру від першої особи (FPS) вимагає обрання архітектурних патернів, які забезпечать ефективність, гнучкість та масштабованість системи. У сучасній індустрії ігрової розробки існує безліч підходів до побудови архітектури, кожен з яких має свої переваги та недоліки. Проте, для FPS гри найбільш доцільним є використання Component-Based Architecture (CBA) та Entity-Component-System (ECS). Розглянемо, чому саме ці патерни є найкращим вибором.

Передусім, варто розглянути специфічні вимоги багатокористувацького шутеру. Такі ігри мають бути динамічними і складними, оскільки ігрові об'єкти мають багато характеристик і поведінкових аспектів, що постійно змінюються. Вони також мають бути масштабованими, щоб підтримувати велику кількість гравців і об'єктів одночасно без втрати продуктивності. Гнучкість і розширюваність є ключовими, оскільки гра повинна дозволяти легке додавання нових функцій і змін наявних без необхідності суттєвих змін у базовій архітектурі. Висока продуктивність є критично важливою для забезпечення плавного ігрового процесу.

Патерн посередника — це шаблон поведінки, який сприяє слабкому зв’язку між об’єктами шляхом інкапсуляції їх взаємодії в об’єкт-посередник. По суті, він визначає об’єкт (посередник), який централізує зв’язок між іншими об’єктами (колегами), таким чином зменшуючи пряму залежність між ними.

Наприклад, коли кілька об’єктів повинні спілкуватися один з одним. Без посередника кожен об’єкт мав би знати про всі інші об’єкти, з якими йому потрібно взаємодіяти, що призвело б до складної мережі залежностей. Ця тісно пов’язана конструкція стає важко підтримувати та розширювати.

Завдяки введенню посередника об’єкти більше не спілкуються безпосередньо один з одним. Замість цього вони спілкуються через посередника, який діє як центральний центр. Це забезпечує кращу інкапсуляцію, оскільки кожен об’єкт повинен знати лише про посередника, а не про інші об’єкти, з якими він взаємодіє.

Патерн посередника сприяє гнучкості та розширюваності програмних систем. Це дозволяє легше додавати або видаляти компоненти, оскільки зміни потрібно вносити лише в посередник, а не поширювати по всій системі. Крім того, це сприяє багаторазовому використанню, оскільки посередник можна повторно використовувати в різних контекстах.

Шаблон інтерпретатора — це поведінковий шаблон проєктування, який використовується для визначення граматики для інтерпретації виразів у мові та

забезпечує спосіб оцінки цих виразів. Це особливо корисно під час роботи з предметно-спеціальними мовами або коли вирази потрібно проаналізувати та виконати.

У шаблоні інтерпретатора зазвичай є набір класів, що представляють різні елементи граматики, такі як нетермінальні та термінальні вирази. Кожен із цих класів реалізує метод «інтерпретації», який обчислює або виконує вираз, який він представляє.

Наприклад, розглянемо простий обчислювач арифметичного виразу. Ви можете мати класи, що представляють різні елементи, наприклад числа, оператори та вирази. Отримавши арифметичний вираз, інтерпретатор аналізує його в синтаксичне дерево, а потім обчислює, рекурсивно інтерпретуючи кожен вузол дерева.

Паттерн інтерпретатора дає змогу відокремити проблеми між граматикою та її інтерпретацією. Це дозволяє визначати складні граматичні правила та легко розширювати або змінювати логіку інтерпретації без зміни самої граматики. Крім того, це полегшує реалізацію нових виразів або мов, надаючи чітку структуру для аналізу та оцінки.

Component-Based Architecture дозволяє створювати складні об'єкти шляхом комбінування простих, незалежних компонентів. У цьому підході об'єкти гри складаються з набору компонентів, кожен з яких додає певну функціональність. Переваги CBA полягають у модульності, адже кожен компонент виконує окрему функцію і може бути змінений незалежно від інших. Це також сприяє повторному використанню коду, оскільки компоненти можуть використовуватися повторно в різних об'єктах, що знижує дуплікацію коду. Легкість розширення досягається через можливість додавання нових функцій шляхом додавання нових компонентів без необхідності змін у вже існуючих об'єктах.

Entity-Component-System (ECS) є еволюцією компонентного підходу, яка розділяє дані та логіку на три окремі частини: Entities, Components, Systems. ECS

має переваги у продуктивності, оскільки чітке розділення даних і логіки дозволяє ефективніше обробляти великі набори даних, оптимізуючи використання кешу і багатопоточності. Гнучкість забезпечується легкістю додавання нових типів поведінки через створення нових систем. Масштабованість досягається ефективним управлінням великою кількістю об'єктів завдяки системам, які обробляють лише релевантні компоненти.

Обидва підходи мають свої переваги і недоліки, проте вони ідеально доповнюють один одного. Комбінація CBA і ECS дозволяє створювати гнучку, модульну і продуктивну архітектуру для багатокористувацького шутеру від першої особи. Модульність та повторне використання компонентів з CBA забезпечують швидку розробку і тестування нових функцій, тоді як ECS забезпечує високу продуктивність завдяки розділенню даних і логіки, дозволяючи ефективно обробляти велике число об'єктів та взаємодій. Обидві архітектури дозволяють легко додавати нові функції та змінювати існуючі, не порушуючи загальної структури коду.

Вибір архітектурних патернів для багатокористувацького шутеру від першої особи є критично важливим для успіху проєкту. Component-Based Architecture та Entity-Component-System забезпечують найкращу комбінацію гнучкості, продуктивності та масштабованості, необхідних для створення складних ігрових систем. Використовуючи ці патерни, розробники можуть створювати ефективні, легко підтримувані та розширювані ігри, які забезпечать захоплюючий досвід для гравців.

2.2 Проєктування модулів

Розроблення багатокористувацького шутера від першої особи вимагає добре структурованої архітектури, яка включає класи гравців, зброї, систему руху, анімацій, мережевий плагін та користувацький інтерфейс. Далі буде розглянуто, як можна організувати цю архітектуру.

Основні компоненти архітектури:

– клас гравця;

– класи зброї;

– система руху;

– система анімацій;

– мережевий плагін;

– користувацький інтерфейс.

Клас гравця відповідає за управління станом гравця, його взаємодію з оточенням, рух, інвентар та стрільбу. Базовий клас гравця, який містить логіку для управління рухом, здоров'ям, взаємодією з оточенням та інвентарем. Гравець має масив зброї, яку гравець може носити.

Класи зброї відповідають за реалізацію різних типів зброї та їх функціональність.

Система руху в стилі Quake повинна підтримувати швидке пересування, стрибки, скольжіння та інші механіки.

Система анімацій відповідає за відтворення анімацій гравця та його взаємодію з оточенням.

Мережевий плагін забезпечує синхронізацію станів між клієнтами та сервером, а також обробку мережевих подій таких як: підключення, знаходження ігрової сесії.

Користувацький інтерфейс (UI) забезпечує відображення інформації гравцю та взаємодію з грою впродовж всіх стадій.

Діаграма (рисунок 2.2) ілюструє процес взаємодії гравця з сервером у багатокористувацькому шутері від першої особи. На ній показано, як різні дії гравця обробляються локально і на сервері для забезпечення синхронізації гри між всіма учасниками.

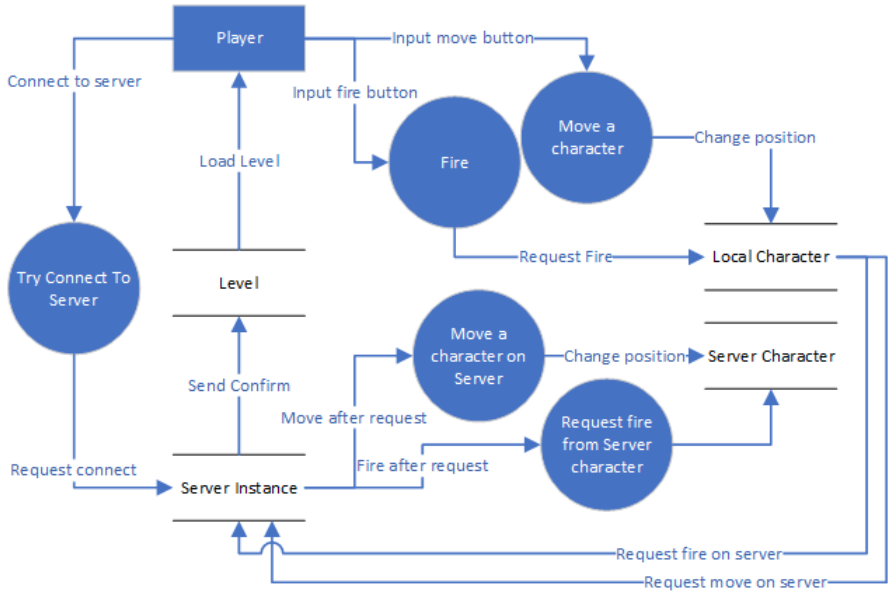


Рисунок 2.2 – DFD-діаграма

Процес починається з того, що гравець намагається підключитися до сервера. Для цього він надсилає запит, який сервер обробляє та підтверджує з'єднання. Після успішного підключення гравець завантажує рівень гри, про що сервер також отримує підтвердження. На цьому етапі гравець готовий до ігрових дій.

Коли гравець вводить команди для руху персонажа, локальний персонаж обробляє ці команди і змінює свою позицію. Одночасно запит на зміну позиції надсилається на сервер, який також обробляє цей запит і змінює позицію персонажа на сервері, забезпечуючи синхронізацію з іншими клієнтами. Цей процес гарантує, що всі гравці бачать актуальне місцезнаходження персонажів у грі.

Коли гравець натискає кнопку стрільби, локальний персонаж обробляє цей запит і виконує стрільбу. Запит на стрільбу також надсилається на сервер, де він обробляється з урахуванням правил гри. Сервер, в свою чергу, повідомляє інших клієнтів про результати стрільби, що включає в себе такі дії, як нанесення шкоди іншим персонажам та відображення візуальних ефектів.

Таким чином, взаємодія гравця з сервером включає кілька етапів: підключення до сервера, завантаження рівня, обробка команд на рух і стрільбу.

Усі ці дії проходять як локально, так і на сервері, що забезпечує коректну синхронізацію станів гри між усіма учасниками. Ця діаграма та опис процесів демонструють, як гравці взаємодіють з сервером у багатокористувацькому шутері, забезпечуючи синхронізацію дій між усіма учасниками гри.

Ця діаграма (рисунок 2.3) зображує процес переміщення в ігровому контексті, демонструючи взаємодію різних компонентів. У центрі діаграми знаходиться процес «Переміщення» (Moving), позначений як вузол A0.

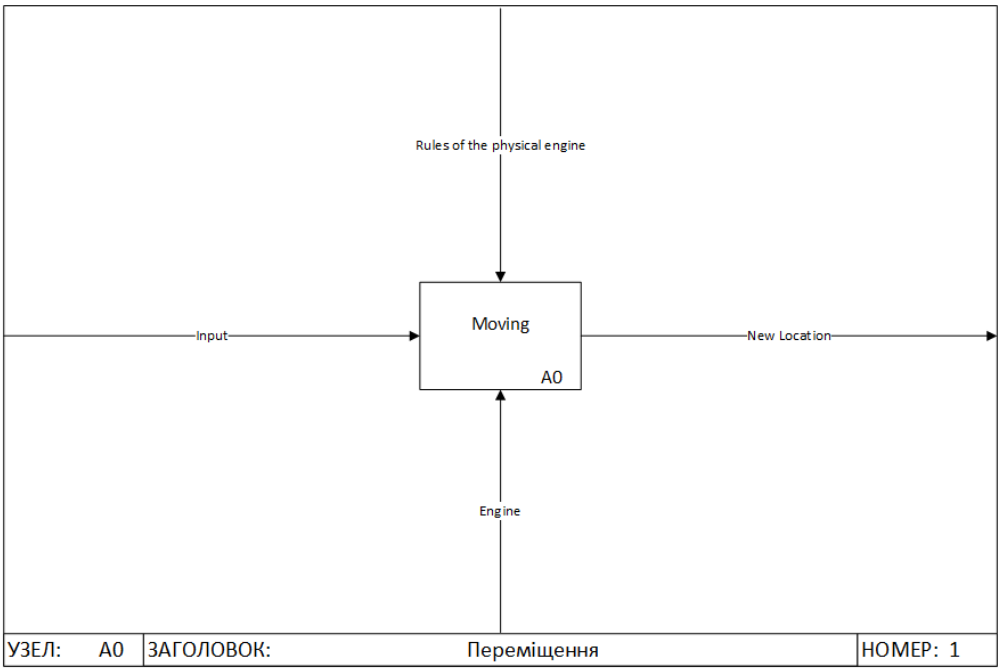


Рисунок 2.3 – Діаграма даних руху

Процес переміщення отримує дані з чотирьох напрямків. Зліва надходять дані введення (Input), які представляють команди користувача, такі як натискання клавіш для руху персонажа. Ці команди є вхідними даними для процесу переміщення, ініціюючи сам процес.

Зверху приходять правила фізичного двигуна (Rules of the physical engine), які визначають фізичні закони, що впливають на рух персонажа. Ці правила забезпечують реалізм у русі, враховуючи такі фактори, як сила тяжіння, тертя і взаємодія з об'єктами у грі.

Знизу надходить двигун (Engine), який представляє рушійну силу, що відповідає за виконання команд користувача відповідно до фізичних правил. Це може бути програмний компонент, який обчислює нові позиції персонажа на основі введених команд і правил фізики.

Справа виходить нове місцезнаходження (New Location), яке є результатом процесу переміщення. Це нова позиція персонажа у просторі гри після обробки введених даних, фізичних правил та обчислень двигуна. Нове місцезнаходження надсилається назад у гру для оновлення позиції персонажа на екрані.

Таким чином, діаграма демонструє, як вхідні команди користувача, фізичні правила і двигун взаємодіють у процесі переміщення, щоб визначити нову позицію персонажа у грі. Кожен компонент відіграє ключову роль у забезпеченні коректного і реалістичного переміщення персонажа у віртуальному середовищі.

Діаграма (рисунок 2.4) ілюструє процес переміщення гравця в багатокористувацькій грі.

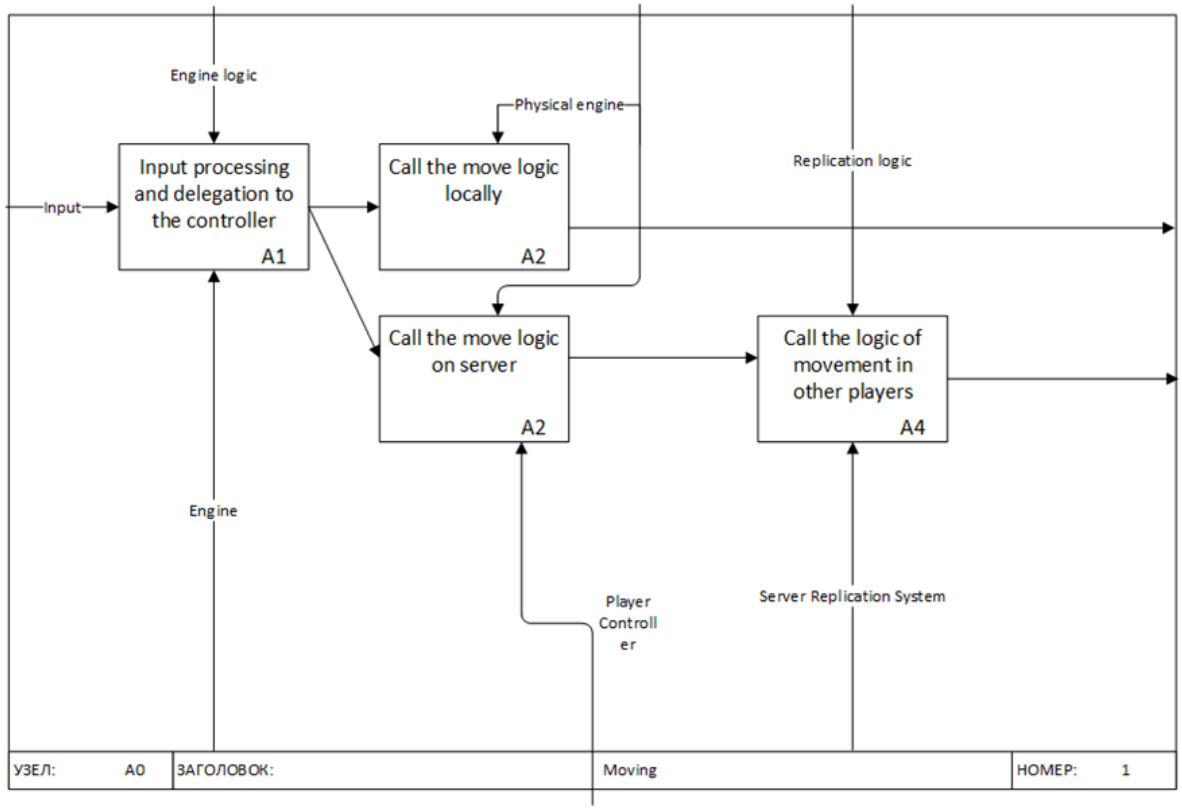


Рисунок 2.4 – Діаграма станів

Демонструє, як обробляється введення користувача та синхронізується рух між клієнтами і сервером. Процес починається з того, що користувач вводить команду, наприклад, натискання клавіші для руху персонажа. Ця команда надходить у блок «Обробка введення та делегування контролеру» (Input processing and delegation to the controller), позначений як A1.

Далі логіка двигуна передає оброблену команду у блок «Виклик логіки переміщення локально» (Call the move logic locally), позначений як A2. Тут обчислюється рух персонажа відповідно до фізичних правил двигуна, що дозволяє локальному клієнту швидко відобразити результат дій гравця.

Паралельно з локальною обробкою, команда передається на сервер у блок «Виклик логіки переміщення на сервері» (Call the move logic on server), також позначений як A2. Сервер обробляє цю команду для забезпечення коректності і відповідності правил гри. Це включає врахування фізичних правил і забезпечення того, що всі дії відповідають встановленим межам гри.

Результати обробки на сервері передаються в блок «Виклик логіки переміщення інших гравців» (Call the logic of movement in other players), позначений як A4. Це дозволяє серверу синхронізувати стан гри між усіма клієнтами, передаючи інформацію про рух персонажа іншим гравцям. Цей процес забезпечується системою реплікації сервера (Server Replication System), яка відповідає за поширення змін у стані гри між всіма підключеними клієнтами.

Таким чином, діаграма демонструє, як введення користувача обробляється локально для швидкого відображення, а також на сервері для забезпечення коректності гри і синхронізації між гравцями. Процес переміщення включає кілька етапів обробки введення, локальної і серверної обробки, а також реплікації змін, щоб всі гравці бачили актуальний стан гри.

Підсумовуючи, виведемо діаграму класів проєкту(рисунок 2.5).

На діаграмі представлено детальний огляд залежностей класів і взаємодії в багатокористувацькій грі шутер від першої особи (FPS), створеній на Unreal Engine. Центральним у цій архітектурі є клас «Персонаж», який представляє

головну сутність, якою керує гравець. Цей клас охоплює різні атрибути та методи, критичні для ігрового процесу, як-от набір зброї (Weapons[] CharacterWeapons) і функції для обробки стрільби зі зброї (Fire()), а також перемикання між зброєю (NextWeapon() і PrevWeapon()).

Клас «Персонаж» інтегрується з кількома ключовими компонентами, кожен з яких керує різними аспектами функціональності персонажа. «Компонент здоров'я» є ключовим для керування здоров'ям персонажа з такими атрибутами, як float MaxHealth і float CurrentHealth, а також методами для встановлення та отримання значень здоров'я (SetHealth() і GetHealth()). Цей компонент гарантує, що стан здоров'я персонажа точно відстежується та оновлюється у відповідь на події в грі.

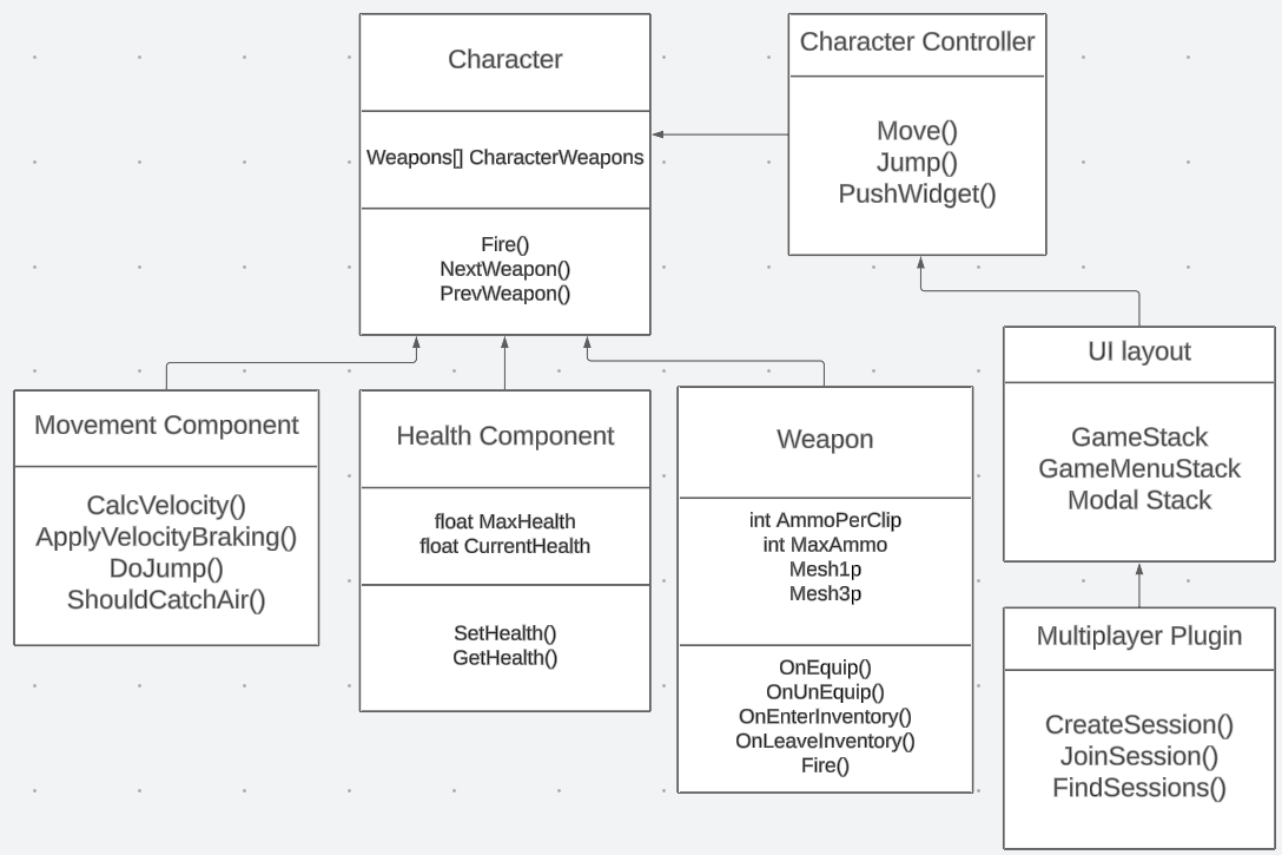


Рисунок 2.5 – Діаграма класів

«Компонент руху» керує механікою руху персонажа. Це включає обчислення швидкості (CalcVelocity()), застосування гальмування швидкості

(ApplyVelocityBraking()), виконання стрибків (DoJump()) і визначення, чи застосовувати силу тертя коли персонаж у повітрі (ShouldCatchAir()). Ці функції спільно керують тим, як персонаж переміщується в ігровому середовищі, забезпечуючи чутливі та динамічні рухи.

Керування зброєю обробляється компонентом «Зброя», який детально описує такі властивості, як int AmmoPerClip, int MaxAmmo та атрибути сітки (Mesh1p для перспектив від першої особи та Mesh3p для перспектив від третьої особи). Компонент містить методи спорядження (OnEquip()), зняття спорядження (OnUnEquip()), входу в інвентар (OnEnterInventory()), виходу з інвентарю (OnLeaveInventory()) і стрільби зі зброї (Fire()). Таке комплексне керування атрибутами та поведінкою зброї має важливе значення для бойової механіки гри.

Клас «Контролер гравця» контролює обробку ввідних даних і керування персонажем за допомогою методів для переміщення (Move()), стрибків (Jump()) і керування взаємодією інтерфейсу користувача (PushWidget()). Цей контролер перетворює дані гравця в дії персонажа, гарантуючи, що гравець може ефективно контролювати персонажа в ігровому світі.

Компонент «UI Layout» відповідає за керування інтерфейсом користувача, включаючи стеки гри (GameStack), стеки меню (GameMenuStack) і модальні стеки (Modal Stack). Це гарантує, що елементи інтерфейсу користувача відображають поточний стан гри, забезпечуючи необхідні відгуки та параметри для гравця. Макет інтерфейсу користувача взаємодіє з контролером персонажа, щоб оновлювати HUD та інші елементи інтерфейсу в режимі реального часу на основі дій гравця та ігрових подій.

Для роботи з багатьма гравцями «Мережевий плагін» керує мережевим ігровим процесом, обробляючи створення сеансу (CreateSession()), приєднання до сеансів (JoinSession()) і пошук сеансів (FindSession()). Цей плагін гарантує, що гра може підтримувати кілька гравців, синхронізуючи дії та стани гри в різних екземплярах клієнта. Він взаємодіє як з контролером персонажів, так і з макетом

інтерфейсу користувача, щоб підтримувати цілісну багатокористувацьку роботу, гарантуючи, що всі гравці бачать узгоджені стани гри та можуть безперешкодно взаємодіяти один з одним.

Підсумовуючи, ця діаграма демонструє добре організовану модульну структуру для багатокористувацької гри FPS на Unreal Engine. Клас персонажа служить основною сутністю, інтегруючись із такими компонентами, як здоров'я, рух і зброя, щоб визначити можливості персонажа. Контролер символів керує введенням і поведінкою, взаємодіючи з макетом інтерфейсу користувача, щоб забезпечити зворотній зв'язок у реальному часі, і плагіном для кількох гравців, щоб забезпечити мережеву синхронізацію. Кожен компонент має окремі обов'язки та взаємодіє через чітко визначені інтерфейси, що полегшує розробку, обслуговування та розширення функцій гри. Ця конструкція забезпечує надійну та масштабовану архітектуру, здатну підтримувати складну ігрову механіку та функції для кількох гравців.

2.3 Проєктування інтерфейсу користувача

Проєктування інтерфейсу користувача (UI) в багатокористувацькому шутері від першої особи є важливою складовою загального дизайну гри. Зокрема, головне меню, яке дозволяє гравцям створювати сесії, приєднуватися до існуючих сесій, налаштовувати гру або виходити, повинно бути інтуїтивно зрозумілим та зручним у використанні. Розміщення меню ліворуч на екрані, з відображенням ігрового рівня на задньому плані, може значно покращити враження користувачів.

Кнопка «Host» - створити сесію (рисунок 2.6): дозволяє гравцю створити нову багатокористувацьку сесію. Натискання цієї кнопки відкриває додаткові параметри для налаштування сесії (вибір карти, кількість гравців тощо).

Рисунок 2.6 – Меню гри

Кнопка «Join» (Підключитись до існуючої сесії): дає можливість підключитися до вже існуючої сесії. Після натискання кнопки з'являється список доступних сесій, до яких можна приєднатися.

Кнопка «Settings» (Налаштування): відкриває меню налаштувань, де користувач може змінити різні параметри гри, такі як графічні налаштування, аудіо, управління та інше.

Кнопка «Exit» (Вихід з гри): закриває гру і повертає користувача на робочий стіл або в головне меню платформи.

Меню розташоване ліворуч на екрані, щоб забезпечити легкий доступ до всіх кнопок. Задній план демонструє рівень гри, створюючи відчуття занурення ще до початку гри.

Верхня панель (рисунок 2.7) з трьома розділами: управління (Controls), відео (Video) та аудіо (Audio). У вкладці «Controls» є налаштування для інвертування вертикальної та горизонтальної осі, а також чутливості миші.

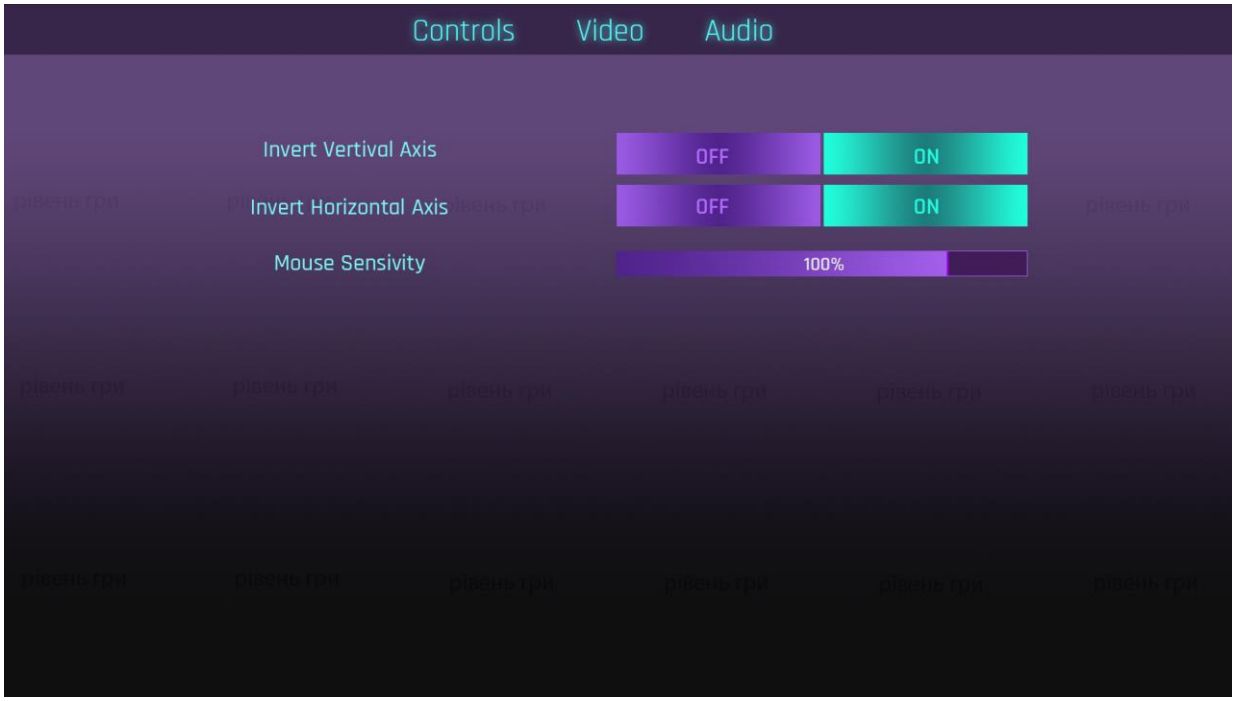


Рисунок 2.7 – Інтерфейс налаштувань

На верхній панелі інтерфейсу розташовані три вкладки: «Controls», «Video» та «Audio». Це дозволяє користувачам швидко перемикатися між різними розділами налаштувань. Після натискання на вкладку «Controls» відкриваються відповідні налаштування для управління.

Верхня панель має бути чітко видимою і розташованою горизонтально у верхній частині екрану. Кожна вкладка оформлена у вигляді кнопки, яка змінює колір або підсвічується при наведенні курсора.

У вкладці «Controls» користувач бачить кілька налаштувань. Першим йде перемикач для інвертування вертикальної осі (Invert Vertical Axis). Цей перемикач дозволяє гравцям змінити напрямок вертикального руху миші. Поряд з ним знаходиться аналогічний перемикач для інвертування горизонтальної осі (Invert Horizontal Axis), що змінює напрямок горизонтального руху миші. Під цими перемикачами розташований повзунок для налаштування чутливості миші (Mouse Sensitivity), який дозволяє гравцям точно налаштувати швидкість руху курсору відповідно до своїх уподобань.

На панелі «Audio» можна змінити гучність звуків гри.

На панелі «Video» є можливість корегувати налаштування графіки такі як: дальність рендеру, відсоток обробки сцени, якість тіней. якість текстур.

На зображенні (рисунок 2.8) показано інтерфейс користувача (UI) відеогри, ймовірно шутера від першої особи. У центрі екрана розташований маленький червоний хрестик, який служить прицілом для стрільби. Внизу екрана знаходиться горизонтальна смуга, що відображає рівень здоров'я гравця, заповнена фіолетовим кольором, з написом «100», що вказує на повний рівень здоров'я.

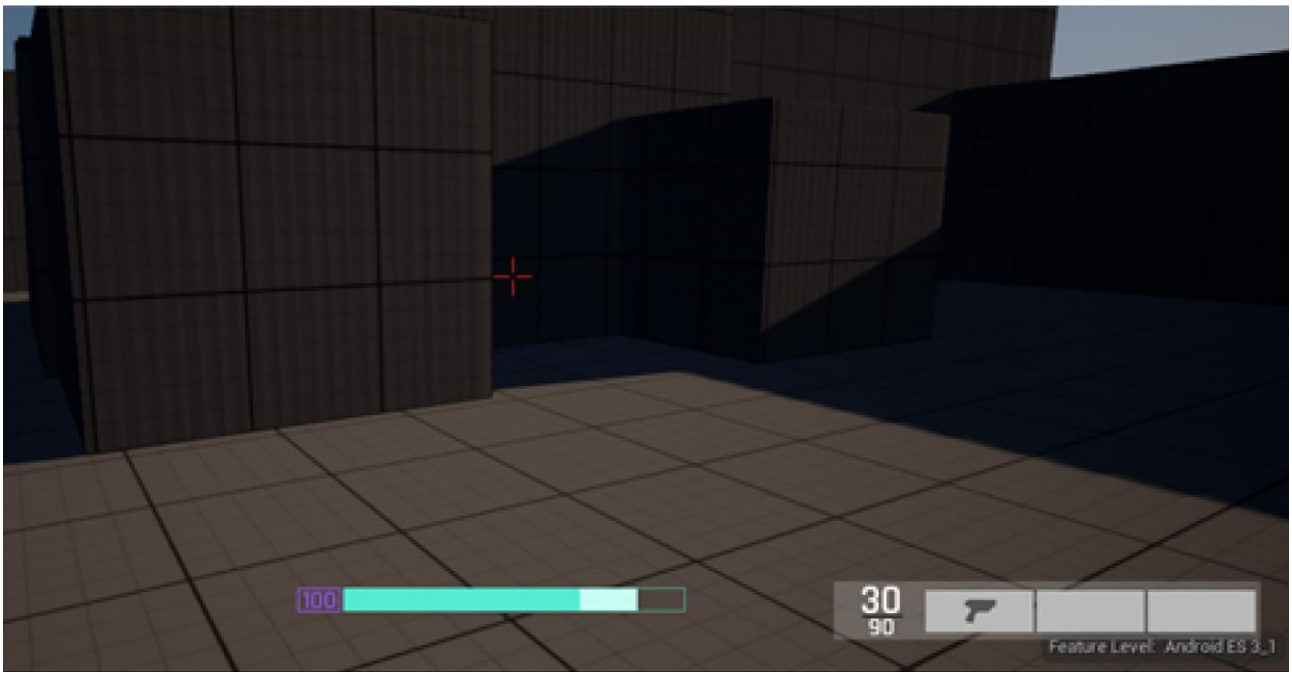


Рисунок 2.8 – Інтерфейс налаштуваннь

У правому нижньому куті екрана відображається інформація про боєприпаси: великим шрифтом вказано число «30» для поточного магазину зброї і «90» для загальної кількості патронів. Поруч з цими числами показано іконку пістолета, що вказує на тип зброї, яку використовує гравець. Панель праворуч знизу також демонструє зброю в інвентарі гравця.

2.4 Аналіз технологій і методів реалізації ПЗ

Розроблення багатокористувацького шутеру від першої особи (FPS) вимагає інтеграції кількох важливих технологій і методів. Процес включає проєктування ігрових механік, реалізацію мережевої архітектури, створення високоякісної графіки та звукового супроводу, а також тестування й оптимізацію. Кожен з цих аспектів потребує ретельного підходу для забезпечення стабільної та захоплюючої гри.

Ітеративна методологія (рисунок 2.9), також відома як ітераційне розроблення або поступове розроблення, — це підхід до розробки програмного забезпечення, коли проєкт розбивається на менші керовані сегменти або ітерації. Кожна ітерація передбачає планування, проєктування, реалізацію та тестування підмножини функцій або функцій. Ітерації повторюються до завершення всього проєкту.

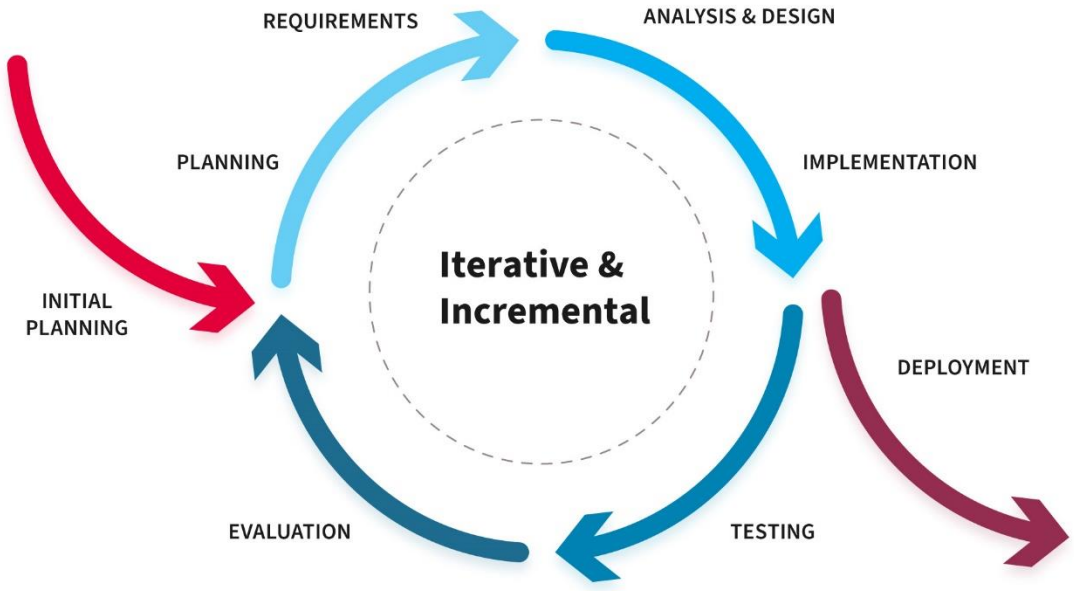


Рисунок 2.9 – Схема ітеративної розробки

Гнучкість: однією з ключових переваг ітераційної методології є її гнучкість. Оскільки проєкт розробляється невеликими кроками, він дозволяє

вносити зміни на кожній ітерації на основі відгуків зацікавлених сторін, ринкових умов або нових вимог. Ця адаптивність допомагає ефективно задовольняти нові потреби.

Раннє надання цінності: надаючи робочі прирости програмного забезпечення наприкінці кожної ітерації, ітераційна методологія забезпечує раннє надання цінності зацікавленим сторонам. Це означає, що користувачі можуть почати отримувати переваги від програмного забезпечення раніше, а не чекати, поки весь проєкт буде завершено. Це також дозволяє ранню перевірку припущень і вимог.

Управління ризиками. Ітеративне розроблення допомагає зменшити ризики проєкту, розбиваючи процес розробки на менші, керовані частини. Ризики визначаються та розглядаються на ранніх стадіях циклу розробки, зменшуючи ймовірність серйозних невдач пізніше. Крім того, ітераційний підхід дозволяє виявляти та виправляти дефекти на ранній стадії, мінімізуючи їхній вплив на загальний проєкт.

Покращена співпраця: Ітеративна методологія сприяє співпраці та комунікації між членами команди та зацікавленими сторонами. Оскільки кожна ітерація передбачає тісну взаємодію між розробниками, тестувальниками та іншими зацікавленими сторонами проєкту, вона сприяє спільному розумінню цілей проєкту, вимог і прогресу. Це середовище для співпраці сприяє швидшому прийняттю рішень і вирішенню проблем.

Цикл зворотного зв’язку: ітеративний підхід заохочує постійний зворотний зв’язок від зацікавлених сторін протягом усього процесу розробки. Зворотній зв’язок, отриманий наприкінці кожної ітерації, допомагає перевірити припущення, уточнити вимоги та визначити пріоритетність функцій для наступних ітерацій. Цей цикл зворотного зв’язку гарантує, що програмне забезпечення, що розробляється, тісно відповідає потребам і очікуванням користувачів.

Легша адаптація до змін: у сучасному динамічному бізнес-середовищі вимоги та пріоритети часто змінюються швидко. Ітеративна методологія дозволяє командам програмного забезпечення ефективніше адаптуватися до цих змін, включаючи їх у наступні ітерації. Така гнучкість дозволяє організаціям залишатися чутливими до ринкових тенденцій і потреб клієнтів, отримуючи таким чином конкурентну перевагу.

Використовуючи ітераційну методологію розробки програмного забезпечення, проєкти можуть виконуватися більш гнучким, адаптивним і спільним способом, що призводить до більш якісних результатів і більшого задоволення зацікавлених сторін.

CryEngine [11, 12], відомий своїми графічними можливостями, часто розглядається як один із потенційних рушіїв для розробки таких ігор. Проте при детальному аналізі можна виявити декілька причин, чому він може не бути найкращим вибором для багатокористувацького FPS.

Перш за все, CryEngine пропонує видатну графіку. Його рендерингова система дозволяє створювати реалістичні візуальні ефекти, високодеталізовані моделі та складне освітлення. Це робить його ідеальним для створення вражаючих ландшафтів і реалістичних середовищ. Однак, графічна перевага CryEngine може стати одночасно і його недоліком. Високі системні вимоги можуть обмежити коло гравців, які зможуть насолоджуватися грою на повну потужність, а також підвищують витрати на розробку і тестування.

Мережева архітектура є ще одним ключовим аспектом. Багатокористувацькі FPS потребують надійної та масштабованої системи, яка забезпечить стабільний зв'язок між гравцями і сервером, мінімальні затримки та захист від шахрайства. CryEngine включає підтримку мережевих функцій, але вона не є такою гнучкою і розвиненою, як у деяких інших рушіїв, таких як Unreal Engine або Unity. Це може ускладнити процес реалізації надійного мультиплеєра і призвести до додаткових витрат на налаштування та підтримку.

Щодо інструментів для розробки та підтримки спільноти, CryEngine поступається своїм конкурентам. Інтерфейс розробки, хоча й потужний, має вищу криву навчання, що може уповільнити команду розробників, особливо якщо вони не мають попереднього досвіду роботи з цим рушієм. Додатково, спільнота розробників CryEngine не така велика і активна, як у випадку з Unreal Engine або Unity, що ускладнює доступ до навчальних матеріалів і технічної підтримки.

Оптимізація продуктивності є критично важливою для будь-якого FPS, особливо багатокористувацького. CryEngine, з його складною графічною системою, може створити проблеми з продуктивністю на менш потужних системах. Це вимагає додаткових зусиль для оптимізації, що може збільшити час і вартість розробки.

Отже, незважаючи на видатні графічні можливості, CryEngine має кілька важливих обмежень, які можуть ускладнити розробку багатокористувацького шутеру від першої особи. Проблеми з мережею, високі системні вимоги, складність інтерфейсу розробки та обмежена підтримка спільноти роблять цей рушій менш привабливим вибором у порівнянні з іншими доступними варіантами, такими як Unreal Engine [13, 14].

Unity служить ігровим рушієм, забезпечуючи механізм візуалізації для створення реалістичної графіки, фізичний рушій для імітації фізики реального світу та середовище сценаріїв для програмування логіки гри за допомогою таких мов, як C#.

Unity має великий магазин активів, де розробники можуть знайти широкий спектр готових ресурсів, включаючи моделі, текстури, анімацію, сценарії та плагіни. Це дозволяє розробникам прискорити розробку, використовуючи готові ресурси та інструменти.

Unity дозволяє розробникам створювати ігри для кількох платформ з одного проєкту. Це означає, що розробники можуть створити гру один раз і

розгорнути її на різних платформах з мінімальними додатковими зусиллями, таким чином охоплюючи ширшу аудиторію.

Unity надає інструменти та функції для оптимізації продуктивності гри, включаючи вбудовані інструменти профілювання, аналітику продуктивності та оптимізацію для конкретної платформи. Це дозволяє розробникам створювати високопродуктивні ігри, які безперебійно працюють на різноманітних пристроях.

Незважаючи на зручний візуальний редактор, Unity має крутий період навчання, особливо для новачків. Освоєння сценарного середовища Unity та розуміння його різноманітних функцій і робочих процесів може зайняти час і зусилля.

Незважаючи на те, що Unity пропонує інструменти оптимізації продуктивності, досягнення оптимальної продуктивності може бути складним завданням, особливо для складних або ресурсомістких ігор. Розробникам може знадобитися додатковий час і зусилля для налаштування та оптимізації продуктивності.

Ключові етапи створення такого ПЗ включають проєктування ігрових механік, розробку мережевої архітектури, створення графіки та звукового супроводу, а також тестування та оптимізацію. Для цього багато студій вибирають сучасні ігрові рушії, такі як Unreal Engine 5.

Проєктування ігрових механік. Це початковий етап, який включає розробку концепції гри, дизайну рівнів, системи зброї, поведінки ворогів і взаємодії гравців. Важливо забезпечити баланс ігрового процесу та цікавий геймплей.

Мережева архітектура. Важливо забезпечити стабільний та надійний мультиплеєр. Це включає розробку клієнт-серверної моделі, синхронізацію ігрових подій, обробку затримок (лагів) і захист від читерства. Для цього використовуються різні протоколи передачі даних (TCP/UDP) і технології, такі як реплікація об’єктів і передбачення руху.

Графіка. Високоякісна графіка є одним з головних аспектів сучасних FPS. Це включає моделювання 3D-об’єктів, текстурування, освітлення, ефекти часток і анімації. Використання фізично коректного рендерингу (PBR) допомагає досягти реалістичних візуальних ефектів.

Звуковий супровід. Реалістичний звук підвищує занурення гравця в гру. Це включає створення звукових ефектів, музичного супроводу та просторового звуку, що дозволяє гравцям орієнтуватися у віртуальному світі за допомогою слуху.

Тестування та оптимізація. Ретельне тестування дозволяє виявити баги і проблеми з продуктивністю. Оптимізація включає зниження навантаження на процесор і графічний процесор, управління пам’яттю та оптимізацію мережевих запитів.

Переваги Unreal Engine 5:

– реалістична графіка (завдяки Lumen і Nanite, UE5 дозволяє створювати надзвичайно реалістичні та детальні візуальні ефекти);

– потужні інструменти розробки. Інтуїтивний редактор і широкий набір інструментів полегшують процес розробки і скорочують час на створення гри;

– велика спільнота і підтримка. Завдяки великій спільноті розробників, доступності великої кількості навчальних матеріалів і активній підтримці з боку Epic Games, розробники можуть швидко знаходити рішення для своїх проблем;

– кросплатформність. UE5 підтримує розробку для різних платформ, включаючи ПК, консолі і мобільні пристрої.

Недоліки Unreal Engine 5:

– для роботи з UE5 потрібні потужні апаратні ресурси, що може бути проблемою для малих студій;

– незважаючи на інтуїтивний інтерфейс, освоєння всіх можливостей UE5 може зайняти значний час, особливо для новачків;

– хоча UE5 безкоштовний для розробки, Epic Games бере роялті за використання рушія після досягнення певного рівня доходу.

Ігри на Unreal Engine розробляються за допомогою мови програмування C++ [15, 16]. Це забезпечує високу продуктивність та гнучкість у створенні складних ігрових механік та систем.

C++ є мовою низького рівня, що дозволяє оптимізувати код для досягнення максимальної продуктивності. Це особливо важливо для ігор, які вимагають швидкої обробки даних та реалістичної графіки.

Розробники можуть безпосередньо керувати апаратними ресурсами, що дозволяє досягати високої швидкості виконання програм.

C++ надає розробникам повний контроль над пам'яттю, що дозволяє ефективно керувати ресурсами та оптимізувати використання пам'яті.

Мова дозволяє використовувати різні парадигми програмування, включаючи об'єктно-орієнтоване програмування, що сприяє організованій структурі коду.

Завдяки підтримці багатьох бібліотек і фреймворків, C++ забезпечує широкий спектр інструментів для реалізації різноманітних ігрових функцій, від фізики та анімації до мережевих можливостей та штучного інтелекту.

Unreal Engine використовує C++ для реалізації основних систем ігрового рушія, що дозволяє розробникам створювати складні ігрові механіки та інтегрувати нові функціональні можливості.

C++ є кросплатформною мовою, що дозволяє розробляти ігри для різних платформ, таких як Windows, macOS, Linux, консолі та мобільні пристрої. Це значно розширює аудиторію та ринок для розроблених ігор.

Таким чином, Unreal Engine 5 є потужним і універсальним інструментом для створення багатокористувацького шутеру від першої особи, що пропонує значні переваги, але також має деякі обмеження, які варто враховувати при виборі рушія.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Програмна реалізація модулів

3.1.1 Розроблення мережевого плагіну

IOnlineSubsystem в Unreal Engine 5 є універсальним інтерфейсом для інтеграції онлайн-сервісів у ваші ігри. Він забезпечує підтримку різних платформ та дозволяє реалізувати багатокористувацькі функції, такі як створення, управління та з'єднання ігрових сесій, обробка досягнень, лідербордів та інших мережевих можливостей.

Завдяки IOnlineSubsystem розробники можуть створювати сесії, які дозволяють гравцям взаємодіяти незалежно від того, чи використовують вони Epic Games Store [17, 18] або Steam [19]. Це означає, що гравці, які придбали гру на різних платформах, можуть грати разом, забезпечуючи міжплатформенний мультиплеєр. Така інтеграція значно розширює аудиторію гри та покращує користувацький досвід.

Інтерфейс IOnlineSubsystem [20, 21] надає доступ до різних модулів, специфічних для кожної платформи, таких як FOnlineSubsystemSteam для Steam і FOnlineSubsystemEpic для Epic Games Store. Це дозволяє налаштовувати параметри сесій, синхронізувати дані гравців та забезпечувати стабільне з'єднання між користувачами на різних платформах.

Розробники можуть використовувати цей інтерфейс для створення лобі, управління запрошеннями в гру та обробки всіх аспектів мережевої взаємодії. Інтеграція з IOnlineSubsystem значно спрощує процес розробки багатокористувацьких ігор, забезпечуючи необхідні інструменти для підтримки складної мережевої архітектури.

Клас UMultiplayerSessionsSubsystem є підсистемою, що розширює функціональність UGameInstanceSubsystem для управління багатокористувацькими сесіями у грі. Цей клас визначає ряд методів і делегатів

для створення, пошуку, приєднання, знищення та запуску сесій, забезпечуючи зручний інтерфейс для роботи з онлайн-сесіями.

Конструктор класу UMultiplayerSessionsSubsystem ініціалізує підсистему та налаштовує основні параметри. Основні публічні методи включають CreateSession, FindSessions, JoinSession, DestroySession та StartSession. Ці методи забезпечують основні дії, необхідні для управління сесіями, такі як створення нової сесії з заданою кількістю підключень та типом матчу, пошук доступних сесій, приєднання до знайдених сесій, знищення існуючих сесій та їх запуск.

Клас також визначає декілька делегатів, які використовуються для повідомлення інших частин коду про завершення різних операцій з сесіями. До них відносяться делегати для повідомлення про завершення створення сесії, пошуку сесій, приєднання до сесії, знищення сесії та запуску сесії. Ці делегати дозволяють іншим класам підписуватися на ці події та виконувати відповідні дії при завершенні відповідних операцій.

Захищені методи включають внутрішні зворотні виклики для обробки подій, які додаються до списку делегатів інтерфейсу сесії. Ці методи включають OnCreateSessionComplete, OnFindSessionComplete, OnJoinSessionComplete, OnDestroySessionComplete та OnStartSessionComplete. Вони використовуються для обробки завершення відповідних операцій та виклику відповідних делегатів.

Приватні змінні класу включають вказівники на інтерфейс сесії (SessionInterface), останні налаштування сесії (LastSessionSettings) та останній пошук сесій (LastSessionSearch). Також визначені делегати для додавання до списку делегатів інтерфейсу сесії та їх обробки. До них відносяться CreateSessionCompleteDelegate, FindSessionsCompleteDelegate, JoinSessionCompleteDelegate, DestroySessionCompleteDelegate та StartSessionCompleteDelegate, а також відповідні обробники (DelegateHandle).

Додатково, клас має прапорець bCreateSessionOnDestroy, який вказує на необхідність створення нової сесії після знищення існуючої, та змінні для збереження інформації про останню кількість підключень

(LastNumPublicConnections) та тип матчу (LastMatchType), що використовуються для створення сесії після знищення.

Цей клас забезпечує комплексне управління багатокористувацькими сесіями, надаючи всі необхідні інструменти для роботи з онлайн-сесіями в рамках гри.

Розглянемо детально функції CreateSession та FindSession, які відіграють ключову роль.

Перевірка валідності SessionInterface:

```
if (!SessionInterface.IsValid()) { return; }
```

На початку методу перевіряється, чи є інтерфейс сесії (SessionInterface) валідним. Якщо він не валідний, метод завершує виконання:

```
auto ExistingSession = SessionInterface->GetNamedSession(SessionName);
	if (ExistingSession != nullptr)
	{
	    bCreateSessionOnDestroy = true;
	    LastNumPublicConnections = NumPublicConnections;
	    LastMatchType = MatchType;
	    SessionInterface->DestroySession(SessionName);
	}
```

Перевіряється, чи існує сесія з заданим ім'ям (SessionName). Якщо така сесія вже існує, встановлюються прапорці для створення нової сесії після знищення існуючої, а також зберігаються параметри нової сесії. Потім існуюча сесія знищується.

Додавання делегата для обробки завершення створення сесії:

```
CreateSessionCompleteDelegateHandle=SessionInterface-
>AddOnCreateSessionCompleteDelegate_Handle(CreateSessionCompleteDelegate);
```

(CreateSessionCompleteDelegate) додається до інтерфейсу сесії. Це дозволяє викликати відповідну функцію, коли створення сесії завершиться.

Налаштування сесії:

```
LastSessionSettings = MakeShareable(new FOnlineSessionSettings());
LastSessionSettings->bIsLANMatch = IOnlineSubsystem::Get()->GetSubsystemName() == "NULL" ? true :
false;
LastSessionSettings->NumPublicConnections = NumPublicConnections;
LastSessionSettings->bAllowJoinInProgress = true;
LastSessionSettings->bAllowJoinViaPresence = true;
LastSessionSettings->bShouldAdvertise = true;
LastSessionSettings->bUsesPresence = true;
LastSessionSettings->bUseLobbiesIfAvailable = true;
LastSessionSettings->Set(FName("MatchType"),                                    MatchType,
EOnlineDataAdvertisementType::ViaOnlineServiceAndPing);
LastSessionSettings->Set(FName("SESSION_NAME"),                          SessionName.ToString(),
EOnlineDataAdvertisementType::ViaOnlineServiceAndPing);
LastSessionSettings->BuildUniqueId = 1;
```

Створюються нові налаштування сесії (LastSessionSettings) і встановлюються різні параметри, такі як кількість підключень, можливість приєднання під час гри, реклама сесії, використання лобі та інші. Параметри MatchType і SessionName також встановлюються за допомогою методу Set.

Створення сесії:

```
const ULocalPlayer* LocalPlayer = GetWorld()->GetFirstLocalPlayerFromController();
if    (!SessionInterface->CreateSession(*LocalPlayer->GetPreferredUniqueNetId(),    SessionName,
*LastSessionSettings))
{
    SessionInterface-
>ClearOnCreateSessionCompleteDelegate_Handle(CreateSessionCompleteDelegateHandle);

    //Broadcast failed
    MultiplayerOnCreateSessionComplete.Broadcast(false);
}
```

Отримується локальний гравець (LocalPlayer) і використовується його унікальний ідентифікатор для створення сесії. Якщо створення сесії не вдається,

делегат очищається, і викликається широкомовний сигнал (Broadcast) з параметром false, що вказує на невдачу створення сесії.

Цей метод забезпечує створення нової багатокористувацької сесії з вказаними параметрами та гарантує, що існуюча сесія буде знищена перед створенням нової, якщо це необхідно. Також він налаштовує всі необхідні параметри сесії та обробляє випадки невдачі при створенні сесії.

Метод UMultiplayerSessionsSubsystem::FindSessions призначений для пошуку доступних багатокористувацьких сесій у грі. Давайте розглянемо роботу цього методу покроково.

Спочатку перевіряється, чи є інтерфейс сесії (SessionInterface) валідним. Якщо інтерфейс не валідний, метод завершує своє виконання, оскільки без валідного інтерфейсу неможливо виконати подальші дії.

Перевірка валідності SessionInterface:

```
if (!SessionInterface.IsValid()) { return; }
```

Далі, створюється делегат для обробки завершення пошуку сесій. Делегат додається до списку делегатів інтерфейсу сесії і зберігається у змінну FindSessionsCompleteDelegateHandle, щоб його можна було видалити зі списку пізніше.

Знаходження сесій:

```
FindSessionsCompleteDelegateHandle = SessionInterface->AddOnFindSessionsComplete
Delegate_Handle(FindSessionsCompleteDelegate);
```

Після цього створюється об'єкт FOnlineSessionSearch, який використовується для зберігання параметрів пошуку сесій. Максимальна кількість результатів пошуку встановлюється в MaxSearchResults, переданий як параметр до методу. Перевіряється, чи використовується локальна мережа (LAN), шляхом порівняння назви підсистеми. Якщо підсистема має ім'я «NULL»,

значить, використовується LAN, і відповідний прапорець встановлюється в true. Додається налаштування пошуку (SEARCH_PRESENCE), яке вказує на те, що потрібно шукати сесії, що використовують присутність (пошук сесій в регіоні гравця).

Налаштування сесій які в пошуку:

```
LastSessionSearch = MakeShareable(new FOnlineSessionSearch());
LastSessionSearch->MaxSearchResults = MaxSearchResults;
LastSessionSearch->bIsLanQuery = IOnlineSubsystem::Get()->GetSubsystemName() == "NULL" ? true : false;
LastSessionSearch->QuerySettings.Set(SEARCH_PRESENCE, true, EOnlineComparisonOp::Equals);
```

Потім отримується локальний гравець (LocalPlayer) із поточного світу. Використовується унікальний ідентифікатор цього гравця для запуску пошуку сесій, викликаючи метод FindSessions інтерфейсу сесії з параметрами локального гравця і налаштуваннями пошуку.

Якщо метод FindSessions не вдається запустити пошук, делегат FindSessionsCompleteDelegateHandle видаляється зі списку делегатів. Потім викликається делегат MultiplayerOnFindSessionsComplete, який сповіщає про завершення пошуку сесій із порожнім масивом результатів і прапорцем false, що вказує на невдачу.

Отримання масиву сесій:

```
const ULocalPlayer* LocalPlayer = GetWorld()->GetFirstLocalPlayerFromController();
if (!SessionInterface->FindSessions(*LocalPlayer->GetPreferredUniqueNetId(), LastSessionSearch.ToSharedRef()))
{
SessionInterface->ClearOnFindSessionsCompleteDelegate_Handle(FindSessionsCompleteDelegateHandle);

MultiplayerOnFindSessionsComplete.Broadcast(TArray<FOnlineSessionSearchResult>(), false);
}
```

Таким чином, метод FindSessions виконує послідовність дій для налаштування параметрів пошуку сесій, запускає процес пошуку і обробляє випадки невдачі, сповіщаючи інші частини коду про результат.

3.1.2 Розроблення системи руху

Система руху в ігрових проєктах, розроблених на платформі Unreal Engine, відіграє ключову роль у створенні реалістичного та захоплюючого геймплею. Вимоги до цієї системи можуть змінюватися в залежності від типу гри, її жанру та концепції, але деякі загальні аспекти можна виокремити для визначення її основних вимог.

По-перше, система руху повинна бути реалістичною та природньою у використанні. Гравець повинен мати відчуття контролю та точності у керуванні персонажем, щоб легко та ефективно взаємодіяти з ігровим середовищем. Це означає, що рух персонажа повинен бути плавним, з відповідними анімаціями та реагуванням на команди гравця без затримок чи неочікуваних перешкод.

Bunny hopping, як технічний термін, відноситься до методу, який часто застосовується у відеоіграх, особливо у шутерах від першої особи (FPS). Ця техніка полягає у швидкому стрибанні або підскакуванні, що робить персонажа важче вразливим і зменшує отримані пошкодження. Вона може забезпечити значну перевагу в ігрових ситуаціях, особливо в конкурентних онлайн-мультиплеєрних сценаріях.

Bunny hopping збільшує швидкість і імпульс гравця, мінімізуючи його вразливість перед вогнем противника [22-24]. Непередбачувані рухи ускладнюють завдання супротивникам прицілитися і вразити персонажа. Основна мета bunny hopping – покращення маневреності та живучості гравця під час ігрових битв та сутичок.

Оволодівши цією технікою, гравці можуть ефективніше переміщатися віртуальними середовищами, ухиляючись від снарядів і долаючи перешкоди. Bunny hopping особливо корисний при захопленні об'єктів, ухилянні від ворогів у вузьких просторах та виході зі складних ситуацій.

Найперший і найдосконаліший метод bunny hopping, що використовує повітряне керування, з'явився у Quake [25, 26], моді Quake III Arena Challenge

ProMode Arena [27] та їх похідних, таких як Warsow і Xonotic. У Half-Life [28-30] (версія 1.1.0.8, випущена у 2001 році) було введено обмеження швидкості, яке обмежує ефективність bunny hop, але ця техніка все ще популярна в багатьох модифікаціях та іграх від Valve. Модель системи руху за якою обчислюється вектор швидкості персонажа можна побачити на рисунку 3.1.

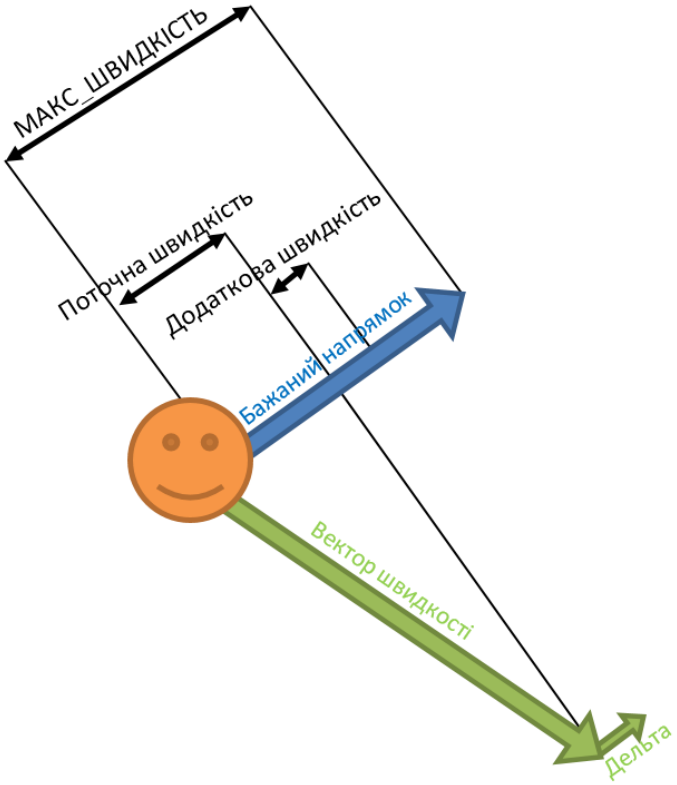


Рисунок 3.1 – Модель руху

Бажаний напрямок використовується, щоб оновлювати напрямок швидкості по горизонталі, вертикальний напрямок швидкості обчислюється за простою гравітаційною моделлю.

Коли персонаж знаходиться на землі – сила тертя впливає на його швидкість. Поточна швидкість – це проекція вектору швидкості на бажаний напрямок. Додаткова швидкість дорівнює різниці між максимальною швидкістю та поточною швидкістю обмежена максимальним прискоренням.

Другий аспект − це оптимізація та продуктивність. Система руху повинна бути оптимізованою для роботи на різних пристроях та платформах,

забезпечуючи стабільний кадровий розподіл та низький рівень затримок навіть у вимогливих ігрових сценах. Це важливо для забезпечення комфортного геймплею та уникнення відчуття «зависання» гри під час активного руху.

Враховуючи ці аспекти, вимоги до системи руху на Unreal Engine можна визначити як динамічність, різноманітність та гнучкість, оптимізація та продуктивність. Вона має забезпечувати гравцям ефективний та захоплюючий ігровий досвід у будь-яких умовах та обставинах.

Логіка руху описана в класі UMBCharacterMovementComponent, проаналізуємо метод який обчислює швидкість.

Метод CalcVelocity в класі UMBCharacterMovementComponent визначає, як змінюється швидкість персонажа протягом часу (DeltaTime), враховуючи такі фактори, як тертя (Friction), прискорення, і стан персонажа (на землі, в повітрі, на сходах тощо).

На початку методу перевіряється, чи є дані для обчислень валідними, чи використовується root motion, чи персонаж є локально керованим, і чи минуло достатньо часу з останнього оновлення. Якщо будь-яка з цих умов не виконується, метод завершиться без змін до швидкості.

Фрикційна сила (Friction) встановлюється на максимальне значення між переданим значенням та нулем, щоб уникнути негативних значень. Далі визначаються максимальне прискорення (MaxAccel) і максимальна швидкість (MaxSpeed).

Якщо прапорець bForceMaxAccel встановлений, прискорення задається на максимальне значення. Це робиться шляхом нормалізації вектора прискорення або використання напрямку швидкості чи вектора вперед компонента, якщо швидкість дуже мала.

Залежно від умов (наприклад, чи персонаж на землі та чи час гальмування минув), застосовується різне тертя та гальмування. Якщо персонаж на землі, застосовується гальмування з використанням тертя та коефіцієнта гальмування (BrakingFriction). Якщо швидкість персонажа перевищує максимальну, але після

гальмування вона стає меншою за максимальну, швидкість коригується, щоб не перевищувати максимум.

Якщо персонаж у рідині, до швидкості застосовується тертя рідини. Значення швидкості обмежуються за осями X і Y, щоб уникнути надмірного прискорення.

При режимі «no clip» (коли персонаж може вільно проходити через об'єкти), швидкість обмежується за допомогою векторів напрямку та прискорення, враховуючи стан бігу персонажа. Якщо персонаж на сходах, рух не змінюється (це місце для майбутньої логіки руху по сходах).

Для ходьби застосовується прискорення, обмежене максимальною швидкістю. Якщо прискорення додає швидкість, швидкість оновлюється з врахуванням тертя поверхні та множника прискорення.

Далі швидкість знову обмежується за осями X і Y. Швидкість оцінюється для можливих змін висоти кроку персонажа: якщо персонаж повільний або на сходах, використовується стандартна висота кроку, в іншому випадку висота кроку зменшується із збільшенням швидкості, щоб уникнути пересування по крутих схилах.

Весь цей процес забезпечує реалістичну і керовану зміну швидкості персонажа в залежності від його стану і взаємодії з оточенням, дозволяючи адаптувати рух для різних ситуацій в грі.

3.1.3 Розроблення зброї

Основні функції стрільби для зброї, такі як керування боєприпасами, перезарядження та реплікація, реалізовано в класі AShooterWeapon.

Зброя переходить у стан стрільби на локальному клієнті та сервері (через виклики RPC). DetermineWeaponState() викликається в StartFire() / StopFire(), який виконує певну логіку, щоб визначити, у якому стані має бути зброя, а потім викликає SetWeaponState(), щоб перевести зброю у відповідний стан. У стані

стрільби локальний клієнт буде неодноразово викликати HandleFiring(), який, у свою чергу, викликає FireWeapon(). Потім він оновлює боєприпаси та викликає ServerHandleFiring(), щоб зробити те саме на сервері. Серверна версія також відповідає за сповіщення віддалених клієнтів про кожен запущений раунд через змінну BurstCounter.

Дії, які виконуються на віддалених клієнтах, є чисто косметичними. Вогонь зі зброї відтворюється за допомогою властивості BurstCounter, щоб віддалені клієнти могли відтворювати анімацію та створювати ефекти – виконувати всі візуальні аспекти стрільби зі зброї.

Миттєве виявлення влучень використовується для швидкострільної зброї (рисунок 3.2), наприклад гвинтівок або лазерних пістолетів.

Рисунок 3.2 – Автоматична зброя

Основна концепція полягає в тому, що коли гравець стріляє зі зброї, перевірка лінії виконується в напрямку, куди зброя спрямована в цей момент, щоб побачити, чи буде щось влучено.

Цей метод забезпечує високу точність і працює з акторами, які не існують на стороні сервера (наприклад, косметичні або відірвані). Локальний клієнт виконує обчислення та інформує сервер про те, що було вражено. Потім сервер перевіряє збіг і, якщо необхідно, повторює його.

У FireWeapon() локальний клієнт виконує відстеження розташування камери, щоб знайти перше блокуюче попадання під перехрестям (прицілом) і передає його в ProcessInstantHit(). Після цього відбувається одна з трьох речей:

– звернення надсилається на сервер для перевірки (ServerNotifyHit()⇒ ProcessInstantHit_Confirmed());

– якщо актор звернення не існує на сервері, звернення обробляється локально (ProcessInstantHit_Confirmed());

– якщо нічого не влучено, сервер отримує сповіщення (ServerNotifyMiss()).

Підтверджені попадання завдають шкоди діючим акторам, створюють сліди та ефекти удару, а також сповіщають віддалених клієнтів, встановлюючи дані про попадання у змінній HitNotify. Пропускає лише створення слідів і встановлення HitNotify для віддалених клієнтів, які шукають зміни HitNotify і виконують ту саму трасування, що й локальний клієнт, створюючи сліди та впливи за потреби.

Реалізація миттєвого влучення також має розкид зброї. Для узгодженості трасування/перевірки локальний клієнт вибирає випадкове початкове число кожного разу, коли виконується FireWeapon(), і передає його в кожен пакет RPC і HitNotify.

Снарядний вогонь використовується для імітації зброї (рисунок 3.3), яка стріляє снарядами, які рухаються повільніше, вибухають під час удару, під впливом сили тяжіння,

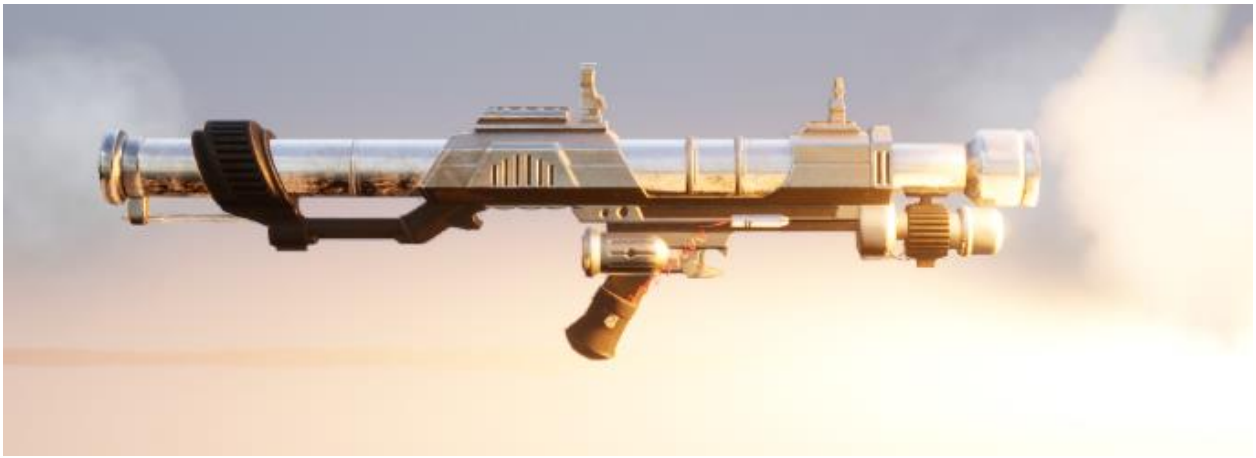


Рисунок 3.3 – Гранатомет

Це випадки, коли результат пострілу зі зброї неможливо визначити в точний момент пострілу, наприклад, запуск гранати. Для цього типу зброї створюється реальний фізичний об’єкт, або снаряд, і направляється в напрямку, куди спрямована зброя. Попадання визначається зіткненням снаряда з іншим об'єктом у світі.

Для снарядового вогню локальний клієнт виконує трасування з камери, щоб перевірити, який Актор знаходиться під прицілом у FireWeapon(), подібно до реалізації миттєвого попадання. Якщо гравець цілиться у щось, він регулює напрямок вогню, щоб влучити в цю точку, і викликає ServerFireProjectile() на сервері, щоб створити снаряд-актора в напрямку, куди була спрямована зброя.

Коли компонент руху снаряда виявляє влучення в сервер, він вибухає, завдаючи шкоди, створюючи ефекти, і відривається від реплікації, щоб сповістити клієнта про цю подію. Потім снаряд вимикає зіткнення, рух і видимість і знищує себе через одну секунду, щоб дати клієнту час для оновлення реплікації.

На клієнтах ефекти вибуху відтворюються за допомогою OnRep_Exploded() .

Інвентар гравця — це масив посилань на AWeapon, що зберігається у властивості Inventory пішака гравця (ACharacter). Озброєна на даний момент зброя копіюється з сервера, і, крім того, ACharacter зберігає свою поточну зброю

локально у властивості CurrentWeapon, що дозволяє знімати попередню зброю, коли споряджається нова зброя.

Коли гравець одягає зброю, відповідна сітка зброї — від першої особи для місцевих, від третьої особи для інших — прикріплюється до персонажа, і на зброї відтворюється анімація. На час анімації зброя переходить у стан спорядження.

3.1.4 Розроблення інтерфейсу

Схема Overall_overlay діє як центральний центр для керування різними рівнями інтерфейсу користувача у вашій грі. Він містить три окремі компоненти UCommonActivatableWidgetStack:

- HUD (Game Stack);
- Game Menu Stack (GameMenuStack);
- Modal Stack (ModalStack).

Кожен стек служить певній меті та контролює різні аспекти інтерфейсу користувача. Розберемо функціональність і роль кожного стека.

HUD (Head-Up Display) — це система відображення інформації, яка проектує дані безпосередньо в поле зору користувача.Призначення: стек HUD (Heads-Up Display) відповідає за відображення елементів інтерфейсу в грі, які завжди видно гравцеві під час гри.

Компоненти: це смужка здоров’я, лічильники боєприпасів, швидкість гравця та інша важлива інформація.

Поведінка: віджети стеку HUD зазвичай неінтерактивні або мінімально інтерактивні, надаючи гравцеві зворотний зв’язок і інформацію в реальному часі, не перериваючи процес гри.

Коли гравець активує ігрове меню (наприклад, натиснувши кнопку меню), GameMenuStack стає видимим.

Схема Overall_overlay із трьома компонентами UCommonActivatableWidgetStack ефективно керує різними рівнями інтерфейсу користувача гри. Стек HUD надає постійну інформацію про гру, GameMenuStack обробляє інтерактивні меню, а ModalStack керує критичними діалогами. Цей багаторівневий підхід гарантує, що користувальницький інтерфейс є організованим, чуйним та інтуїтивно зрозумілим, дозволяючи гравцям зосередитися на грі, маючи при необхідності легкий доступ до меню та важливих діалогових вікон.

Створюємо наступні класи: MBPlayerHUD, MBProgressBar, MBSettingsList, MBSettingsWidget, MBUIButtonBase, UMBMenuWidget. Класи які відповідають за головне меню та меню налаштувань походять від UCommonActivatableWidget, в той час як всі інші походять від UUserWidget.

Клас кнопки:

```
UCLASS()
class MB_API UMBUIButtonBase : public UUserWidget
{
GENERATED_BODY()
public:
UPROPERTY(BlueprintReadWrite, meta = (BindWidget))
UButton* ButtonBase;
protected:
FText* ButtonText;
```

Класс кнопки налічує в собі посилання на саму кнопку, щоб з класів меню була можливість викликати делегат OnClicked та посилання на текст в кнопці.

Клас меню посилається на всі кнопки які знаходяться в головному меню, визначає їх функціональність та відповідає за перемикання між собою та меню налаштувань.

Присвоєння фунції кнопці:

```
if (Exit_Button)
Exit_Button-&gt;ButtonBase-&gt;OnClicked.AddDynamic(this,
&UMBMnuWidget::Exit_Button_OnClicked);
```

Спочатку в конструкторі присвоюємо кожній кнопці її функцію з допомогою делегату OnClicked який мають усі об’єкти класу UButton.

Функціонал кнопок:

```
FString QualityLevel[5] = {TEXT(&quot;Low&quot;),TEXT(&quot;Medium&quot;),
TEXT(&quot;High&quot;), TEXT(&quot;Epic&quot;), TEXT(&quot;Cinematic&quot;) };
int32 CurrentQualityLevel;
```

В заголовку класу створений масив загальних налаштувань графіки та функції для відслідковування поточних налаштувань та подальшого їх встановлення іншим значенням(void SetQualityLevel(int32);void SetQualityLev elText(int32);int32 GetQualityLevel() const;). Подібно віджету меню оголошуються функкції для стрілок перемикання між налаштуваннями графіки та встановлюються в конструкторі.

Головне меню «MACHINE BLASTER» (рисунок 3.4) вирізняється елегантним сучасним дизайном із темним фоном, що створює висококонтрастну футуристичну атмосферу. У лівій частині екрана вертикальна фіолетова панель містить параметри меню. У верхній частині цієї панелі назва гри «MACHINE BLASTER» відображається яскравим неоновим синім футуристичним шрифтом, що чітко виділяє її на фіолетовому тлі.

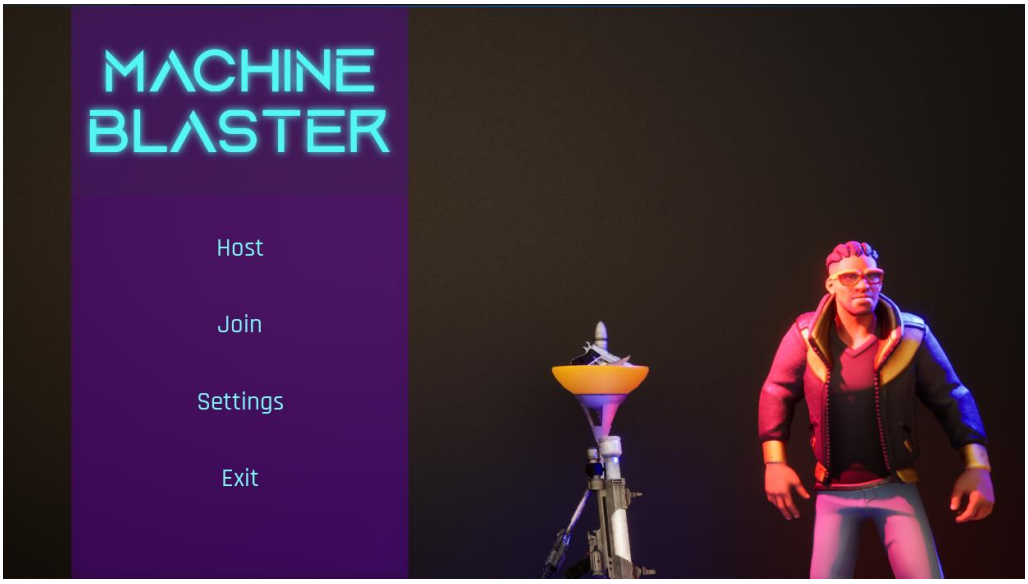


Рисунок 3.4 – Головне меню гри

Під назвою гри опції меню розташовано вертикально чистим білим текстом, що забезпечує просту навігацію. Опції включають «Хост» для початку нової гри або проведення сеансу для кількох гравців, «Приєднатися» для приєднання до існуючої гри, «Налаштування» для налаштування параметрів гри та «Вихід» для закриття гри.

Праворуч від фіолетової панелі модель персонажа додає меню візуального інтересу та індивідуальності. Персонаж, одягнений у сучасний одяг із піджаком та окулярами, стоїть впевнено, навіюючи відчуття готовності та дії. Поруч із персонажем відображається футуристичний гаджет або зброя, що підсилює науково-фантастичну тему гри. Ця комбінація елементів створює захоплюючий і захоплюючий вступ до гри, легко проводячи гравців через їхні варіанти, задаючи тон майбутнім пригодам.

Меню налаштувань у «MACHINE BLASTER» (рисунок 3.5), зокрема вкладка налаштувань відео, зберігає витончену та сучасну естетику гри. Темно-фіолетовий фон створює послідовну та візуально привабливу тему.

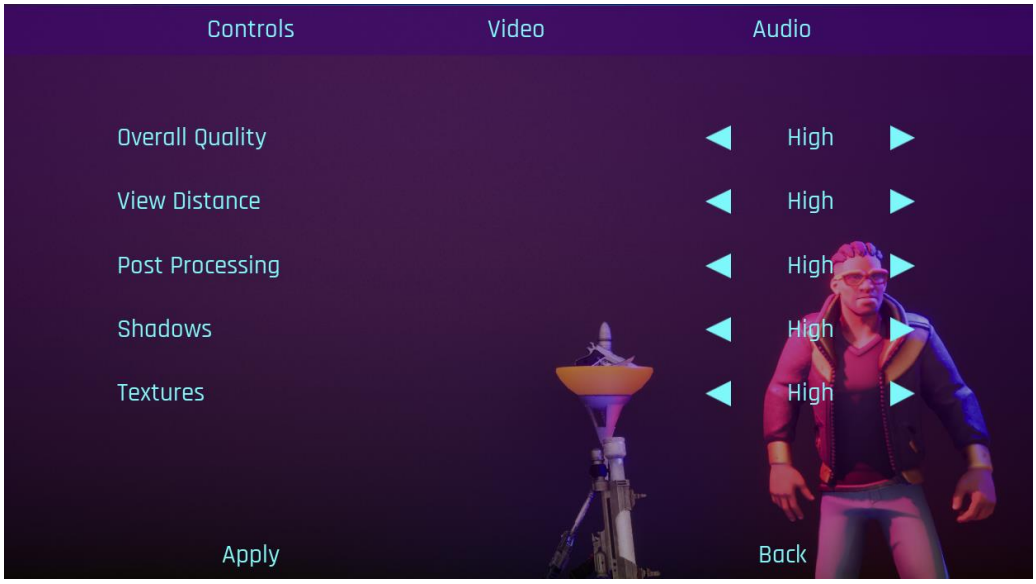


Рисунок 3.5 – Меню налаштувань

Отримуємо об’єкт gameUserSettings який передає налаштування гри та змінює їх. З допомогою раніше описаних функцій передаємо поточні налаштування у шаблон, щоб при натисканні на відео налаштування користувач бачив поточні параметри графіки. Переходи між вкладками реалізовані ідентичним чином як і в головному меню.

Завершальним моментом зберігаємо визначені користувачем налаштування при натисканні кнопки Apply:

```
if (gameUserSettings) {
gameUserSettings->SetViewDistanceQuality(ViewDistance->GetQualityLevel());
gameUserSettings->SetPostProcessingQuality(PostProcessing->GetQualityLevel());
gameUserSettings->SetShadowQuality(Shadows->GetQualityLevel());
gameUserSettings->SetTextureQuality(Textures->GetQualityLevel());
gameUserSettings->SetOverallScalabilityLevel(OverallQuality-
>GetQualityLevel());
UpdateVideoQualityText();
}
gameUserSettings->ApplySettings(false);
```

Після чого можемо бачити результат на ігровому рівні на на фоновому рівні меню.

У верхній частині екрана є горизонтальна навігаційна панель із трьома вкладками з позначками «Елементи керування», «Відео» та «Аудіо». Зараз активна вкладка «Відео».

Основний розділ меню налаштувань розділений на дві області. Ліворуч світло-блакитним текстом представлено список параметрів налаштування відео. Ці параметри включають «Загальну якість», «Відстань перегляду», «Постобробку», «Тіні» та «Текстури». Для кожного параметра праворуч відображені значення, які можна настроїти, для яких наразі встановлено значення «Високий». Маленькі трикутні стрілки поруч із кожним параметром дозволяють легко регулювати.

У нижній частині екрана світло-блакитним текстом є дві опції: ліворуч «Застосувати» для збереження змін, внесених до налаштувань, і «Назад» праворуч для повернення до попереднього меню.

Меню пошуку в лобі (рисунок 3.6) в «MACHINE BLASTER» має простий і зрозумілий дизайн, що відповідає загальній футуристичній темі гри. Фон залишається темним із тонким градієнтом, який додає глибини інтерфейсу.

У верхній частині екрана помітно відображається заголовок «Список сеансів» зеленувато-блакитним шрифтом, який є стандартним в CommonUI для цього проєкту, що вказує на поточний фокус меню.

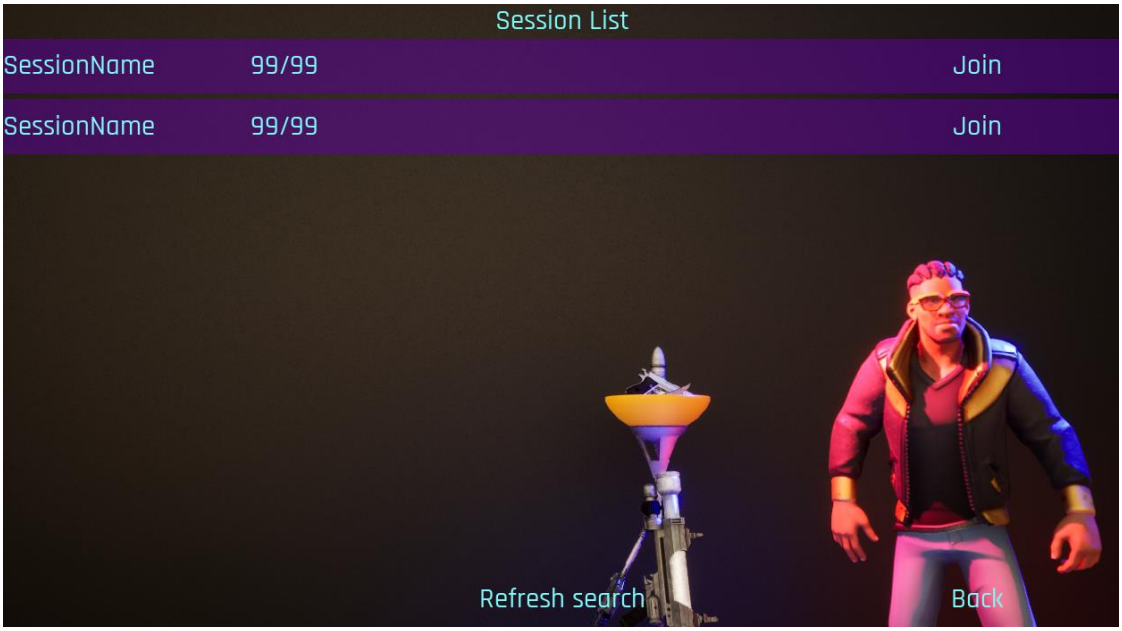


Рисунок 3.6 – Меню пошуку сесій

Кожен запис про сеанс містить назву сеансу, яка відображається ліворуч світло-блакитним текстом, кількість гравців у форматі «99/99», що вказує як поточних гравців, так і максимальну кількість, а також кнопку «Приєднатися» праворуч, яка запрошує гравців приєднатися до сеансу. . Цей макет гарантує, що важлива інформація є легкодоступною та добре організованою.

Ігровий HUD багатокористувацького шутера, який продемонстрований на рисунку 3.7, включає різні елементи. Угорі по центру є таймер, який показує 10:33, поруч із результатами, що вказують 1 для рожевої команди та 3 для синьої команди. Під таймером і оцінками є центральне перехрестя для прицілювання.

У нижньому лівому куті смужка здоров’я показує здоров’я гравця на рівні 100. Поруч із цією смужкою здоров’я є числове відображення, яке вказує «0», що показує швидкість персонажа, відповідно до одиниць рушія.

У нижньому правому куті HUD (рисунок 3.7) відображає кількість боєприпасів гравця з поточною зброєю, яка показує 30 куль у магазині. Поруч з ним є значки іншої доступної зброї: пістолета та автомата, кожна з яких відображається індикатором кількості. Слово «ПОРОЖНІЙ» відображається

поруч із додатковим слотом, що вказує на те, що в цьому слоті зараз немає жодного предмета чи зброї.

Рисунок 3.7 – HUD гравця

Середовище — це футуристична металева кімната з гладкими відбивними поверхнями, а зброю гравця видно в нижній правій частині екрана.

3.2 Вимоги до технічних та програмних засобів

Для забезпечення стабільної та якісної роботи багатокористувацького шутера від першої особи, розробленого на Unreal Engine, необхідно враховувати певні вимоги до апаратного та програмного забезпечення. Відповідність цим вимогам гарантує, що гра буде працювати плавно, без затримок та технічних проблем, що є критично важливим для задоволення користувачів та створення захоплюючого ігрового досвіду. У цьому розділі детально розглядаються необхідні технічні характеристики та програмні засоби, які забезпечують повноцінну та безперебійну функціональність гри.

Для запуску багатокористувацького шутера від першої особи на Unreal Engine на базовому рівні, необхідна така апаратно-програмна конфігурація:

– операційна система: Windows 10 (64-bit) або новіша, macOS 10.14 або новіша;

– процесор: Intel Core i5-4460 або AMD Ryzen 3 1200;

– оперативна пам'ять: 8 GB RAM;

– відеокарта: NVIDIA GeForce GTX 960 або AMD Radeon R9 280 з 2GB відеопам'яті;

– місце на диску: 15 GB вільного місця;

– мережеве підключення: Широкосмуговий інтернет з мінімальною швидкістю 5 Mbps.

Для забезпечення найкращої якості графіки та продуктивності, рекомендується наступна конфігурація:

– операційна система: Windows 10 (64-bit) або новіша, macOS 10.14 або новіша;

– процесор: Intel Core i7-8700 або AMD Ryzen 5 3600;

– оперативна пам'ять: 16 GB RAM;

– відеокарта: NVIDIA GeForce GTX 1070 або AMD Radeon RX 580 з 8GB відеопам'яті;

– місце на диску: SSD з 15 GB вільного місця

– мережеве підключення: Широкосмуговий інтернет з мінімальною швидкістю 25 Mbps.

Для оптимальної роботи гри потрібні наступні програми: Steam або Epic Games Launcher для завантаження, оновлення гри та підключення гравців.

Для забезпечення стабільної роботи гри необхідно підтримувати актуальні версії драйверів:

– драйвери відеокарти: останні версії драйверів від NVIDIA або AMD для оптимальної роботи графіки;

– аудіодрайвери: оновлені драйвери звукової карти для забезпечення якісного звукового супроводу;

– мережеві драйвери: актуальні драйвери для мережевих адаптерів, що забезпечують стабільне інтернет-з'єднання.

Забезпечення відповідності технічним та програмним вимогам є ключовим для стабільної та якісної роботи багатокористувацького шутера від першої особи, розробленого на Unreal Engine. Дотримання зазначених мінімальних та рекомендованих вимог дозволить користувачам насолоджуватися грою без технічних перешкод, забезпечуючи при цьому оптимальний ігровий досвід.

3.3 Тестування програмного забезпечення

3.3.1 Вибір та обґрунтування методів тестування застосунку

Тестування багатокористувацького шутера від першої особи на Unreal Engine потребує ретельного підходу для забезпечення якості та стабільності гри. Було обрано кілька методів на розгляд для тестування різних аспектів гри, зокрема юніт-тестування, інтеграційне тестування, функціональне тестування, навантажувальне тестування та тестування продуктивності.

Інтеграційне тестування є наступним кроком, де перевіряється взаємодія між різними модулями системи. В багатокористувацьких іграх це особливо важливо, оскільки різні компоненти повинні працювати разом без конфліктів. Використовуючи Unreal Automation Framework, було створено сценарії, які симулюють реальні ігрові умови. Це дозволило тестувати взаємодію мережевого коду, синхронізацію гравців та інтерфейс користувача, перевіряючи коректність обміну даними між клієнтами та сервером, а також узгодженість ігрових подій між різними гравцями.

Функціональне тестування перевіряє роботу системи відповідно до вимог технічного завдання. Це включає тестування всіх основних функцій та ігрових

механік. Було використано як ручне, так і автоматизоване тестування для перевірки ключових аспектів гри, таких як стрільба, рух персонажів, взаємодія з об'єктами та користувацький інтерфейс. Чек-листи та тест-кейси, розроблені на основі технічного завдання, забезпечили всебічну перевірку кожної функції. В процесі тестування залучалися як внутрішні тестувальники, так і зовнішні бета-тестери, що дозволило отримати зворотний зв'язок і виявити додаткові дефекти.

Навантажувальне тестування оцінює продуктивність гри під великим навантаженням, що є ключовим для багатокористувацьких ігор. За допомогою інструментів, таких як JMeter та Unreal Engine Server, були створені сценарії, які симулювали велику кількість одночасних користувачів. Це дозволило виміряти продуктивність серверу, включаючи час відгуку та стабільність системи під високим навантаженням, і визначити максимально допустиме число одночасних гравців без втрати продуктивності.

Тестування продуктивності перевіряє ефективність гри на різних апаратних конфігураціях, щоб забезпечити плавний ігровий процес для всіх користувачів. За допомогою Unreal Profiler та інших профайлерів було виміряно ключові показники продуктивності, такі як FPS, використання пам'яті та завантаження процесора. Гра тестувалася на різних системах, включаючи мінімальні та рекомендовані конфігурації, що дозволило виявити вузькі місця в продуктивності та провести оптимізацію коду.

На основі проведеного аналізу різних методів тестування, було прийнято рішення зосередитися на юніт-тестах [31, 32] для перевірки окремих компонентів та функціональному тестуванні модулів для детального аналізу функціональності та інтеграції. Такий підхід дозволяє швидко виявляти та виправляти помилки на ранніх етапах розробки, а також забезпечує детальну перевірку взаємодії між різними частинами системи, що є ключовим для успішного запуску багатокористувацького шутера.

3.3.2 Аналіз результатів тестування

На рисунку 3.8 показано інтерфейс модульного тестування,, із середовища розробки ігор Unreal Unit Test Framework, із різними тестами, націлених на перевірку програмного продукту. Кожний тест має відповідний прапорець, який вказує на його статус вибору, стовпець тривалості, який показує, скільки часу потрібно для виконання кожного тесту, і зелені смужки, що вказують на успішне завершення.

Конкретні тести, перелічені в розділі «GunAmmo» (який містить 50 тестів), включають й ті, які зображені на рисунку 3.8.

Add (7): Цей тест зайняв 0,211 секунди і пов'язаний із додаванням боєприпасів до зброї.

CanReload (6): цей тест, який триває 0,167 секунди, перевіряє, чи може зброя перезарядитися.

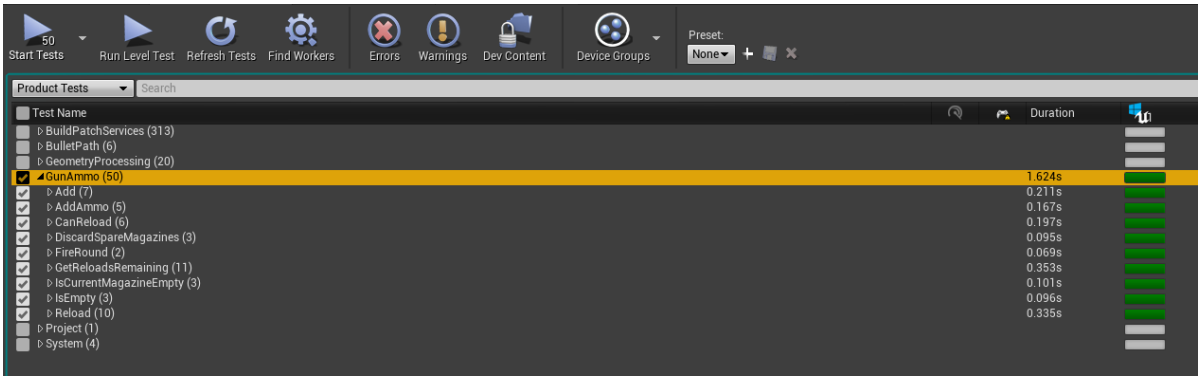


Рисунок 3.8 – Юніт-тести

DiscardSpareMagazines (3): виконується за 0,197 секунди, перевіряє функціональність перезарядки з патронами в магазині.

FireRound (2): цей тест, який займає 0,095 секунди, перевіряє, чи може зброя вистрілити.

GetReloadsRemaining (11): перевірка часу, що залишився до готової перезарядки зброї, займає 0,069 секунди.

IsCurrentMagazineEmpty (3): цей тест перевіряє, чи поточний магазин порожній, і займає 0,353 секунди.

IsEmpty (3): цей швидкий тест, який займає 0,101 секунди, перевіряє, чи патрони відсутні.

Reload (10): цей тест займає 0,096 секунди і перевіряє функцію перезавантаження.

Project (1): тривалість цього тесту становить 0,335 секунд, стосується функцій, пов’язаних зі снарядом.

System (4): ширший системний тест перевіряє загальну інтеграцію та зайняв 0,335 секунди.

Функціональне тестування є критичним етапом у розробці багатокористувацького шутера від першої особи, оскільки воно забезпечує відповідність гри визначеним вимогам та очікуванням користувачів. Цей вид тестування спрямований на перевірку всіх ключових функцій гри, таких як механіки стрільби, система руху, процес підключення гравців та інші важливі аспекти ігрового процесу.

Одним з найважливіших елементів багатокористувацького шутера є система стрільби.

Різноманітні типи зброї: Було проведено ретельне тестування всіх типів зброї, доступних у грі, на предмет точності, шкоди, віддачі, швидкострільності та інших характеристик.

Різні умови стрільби: Тестування проводилося в різних умовах, щоб перевірити поведінку зброї. Це включало стрільбу по статичних та рухомих цілях, стрільбу з близької та далекої відстані, стрільбу з різних положень та з різною швидкістю руху.

Виявлення та виправлення помилок: Виявлені під час тестування помилки, пов'язані зі стрільбою, такі як невідповідність шкоди, візуальні помилки або проблеми з синхронізацією, були ретельно задокументовані та виправлені.

Іншою критичною областю є система руху персонажів.

Різні типи рухів: Було протестовано всі аспекти системи руху, включаючи швидкість ходу, бігу, стрибків, ковзання, лазіння, плавання та інші механіки.

Різні середовища: Тестування проводилося в різних середовищах, щоб перевірити, як система руху взаємодіє з різними типами поверхонь, перешкодами та інтерактивними елементами.

Виявлення та виправлення помилок: Були виявлені та виправлені помилки, пов'язані з системою руху, такі як незграбні анімації, проблеми з фізикою або проблеми з синхронізацією між клієнтами.

Процес підключення гравців до матчу є важливим аспектом для забезпечення безперебійного ігрового досвіду.

Процес підключення: було проведено ретельне тестування процесу підключення гравців до матчу, щоб виміряти час завантаження, синхронізацію даних та інші аспекти.

Різні умови мережі: тестування проводилося в різних мережевих умовах, щоб перевірити стійкість процесу підключення до втрати пакетів, затримки та інших проблем з мережею.

Виявлення та виправлення помилок: Були виявлені та виправлені помилки, пов'язані з підключенням гравців, такі як проблеми з пошуком матчу, помилки приєднання або проблеми з синхронізацією між клієнтами.

Окрім вищезазначених основних аспектів, тестування також охопило й інші аспекти.

Тестування користувацького інтерфейсу: перевірка зручності використання та інтуїтивності інтерфейсу гри.

Тестування продуктивності: Оцінка продуктивності гри на різних конфігураціях обладнання.

Завдяки комплексному підходу до тестування, багатокористувацький шутер від першої особи на Unreal Engine став надійним та якісним продуктом, готовим до запуску і подальшої експлуатації.

ВИСНОВКИ

В сучасному світі ігрова індустрія є однією з найбільш швидко зростаючих та прибуткових галузей, яка стрімко розвивається. Відеоігри стали невід'ємною частиною культури та дозвілля мільйонів людей по всьому світу, значно впливаючи на соціальні, економічні та технологічні аспекти суспільства. Особливо популярними є багатокористувацькі ігри, які дозволяють гравцям взаємодіяти один з одним у віртуальному просторі, створюючи унікальні та захоплюючі світи. Жанр шутерів від першої особи (First-Person Shooter, FPS) займає особливе місце серед таких ігор завдяки своїй динаміці, реалістичній графіці та високому рівню залученості гравців.

Для досягнення мети розробки багатокористувацького шутера від першої особи було необхідно вирішити низку науково-практичних завдань: провести аналіз існуючих рішень у жанрі FPS, розробити концепцію гри, вибрати відповідні інструменти та технології для реалізації проєкту, здійснити тестування та оптимізацію мережевої складової гри.

Проведення тестування було важливим етапом розробки, який включав юніт-тестування, інтеграційне тестування, функціональне тестування, навантажувальне тестування та тестування продуктивності. Виявлені помилки та недоліки були успішно виправлені, що дозволило забезпечити стабільну та якісну роботу гри. Особливу увагу було приділено функціональному тестуванню, що охопило всі ключові аспекти ігрового процесу, такі як механіка стрільби, система руху, процес підключення гравців та інші важливі елементи.

Для забезпечення стабільної та якісної роботи багатокористувацького шутера було визначено основні вимоги до апаратно-програмних засобів. Мінімальні та рекомендовані системні вимоги дозволяють гравцям насолоджуватися грою на різних конфігураціях обладнання, забезпечуючи оптимальний ігровий досвід.

Виконання всіх етапів розробки та тестування призвело до створення продукту, на базі якого можна зробити готову до експлуатації гри.

У результаті аналізу існуючих рішень у жанрі FPS було вивчено та проаналізовано два провідні представники жанру: «Quake» та «Overwatch».

Quake: Було докладно досліджено геймплей та механіки гри, включаючи систему стрільби, руху, а також різноманітність зброї. Проаналізовано особливості інтерфейсу користувача, такі як способи взаємодії з гравцем, інформаційний відображення та контроль гри.

Overwatch: Проведено дослідження головних особливостей гри, включаючи унікальні персонажі з різними навичками та ролі в команді, а також різноманітність карт та геймплейних режимів. Проаналізовано систему зброї, руху та інтерфейсу користувача.

Реалізовано оновлення напрямку швидкості по горизонталі та обчислення вертикального напрямку швидкості за простою гравітаційною моделлю. Це дозволяє персонажеві вільно рухатися в грі, дотримуючись реалістичної фізики.

Реалізовано оновлення напрямку швидкості по горизонталі та обчислення вертикального напрямку швидкості за простою гравітаційною моделлю. Це дозволяє персонажеві вільно рухатися в грі, дотримуючись реалістичної фізики.

Таким чином, система руху на Unreal Engine відповідає визначеним критеріям динамічності, різноманітності, гнучкості, оптимізації та продуктивності. Вона успішно забезпечує гравцям ефективний та захоплюючий ігровий досвід у будь-яких умовах та обставинах.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Analysts: FPS 'Most Attractive' Genre for Publishers. URL: https://www.gamedeveloper.com/latest-news (дата звернення: 08.04.2024)

2. What the strange evolution of the hero shooter tells us about the genre's future. URL: https://www.pcgamer.com/what-the-strange-evolution-of-the-hero-shooter-tells-us-about-the-genres-future/ (дата звернення: 08.04.2024)

3. Сторінка Team Fortress 2 в Steam. URL: https://store.steampowered.com/app/440/Team_Fortress_2/ (дата звернення: 09.04.2024)

4. Офіційна сторінка Team Fortress. URL: https://www.teamfortress.com/ (дата звернення: 09.04.2024)

5. Official Team Fortress Wiki. URL: https://wiki.teamfortress.com/wiki/Main_Page (дата звернення: 09.04.2024)

6. Офіційний сайт Valve. URL: https://www.valvesoftware.com/el/about (дата звернення: 09.04.2024)

7. Офіційний сайт Overwatch. URL: https://overwatch.blizzard.com/ (дата звернення: 09.04.2024)

8. Overwatch Review. URL: http://www.gamespot.com/reviews/overwatch-review/1900-6416439/ (дата звернення: 09.04.2024)

9. Overwatch open beta - everything you need to know. URL: http://www.pcgamesn.com/overwatch/overwatch-beta (дата звернення: 09.04.2024)

10. Офіційний сайт Blizzard. URL: https://www.blizzard.com/ (дата звернення: 09.04.2024)

11. Офіційний сайт CryEngine. URL: https://www.cryengine.com/ (дата звернення: 12.04.2024)

12. . Crytek's Video Game Engine Is Now Free. URL: https://web.archive.org/web/20160510122147/http://kotaku.com/cryteks-video-game-engine-is-now-free-1765078659 (дата звернення: 12.04.2024)

13. Unreal Engine 5 documentation. URL: https://docs.unrealengine.com/5.0/en-US/ (дата звернення: 13.04.2024)

14. Офіційний сайт Unreal Engine. URL: https://www.unrealengine.com/en-US (дата звернення: 13.04.2024)

15. C++ reference. URL: https://en.cppreference.com/w/ (дата звернення: 15.04.2024)

16. Peter Van Weert, Ivor Horton Beginning C++20: From Novice to Professional 6th ed. Edition : Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A, 853p.

17. Офіційний сайт Epic Games. URL: https://www.epicgames.com/site/en-US/home (дата звернення: 16.04.2024)

18. Офіційний твіттер Epic Games. URL: https://twitter.com/EpicGames (дата звернення: 12.04.2024)

19. Офіційний сайт Steam https://store.steampowered.com/ (дата звернення: 16.04.2024)

20. OnlineSubsystem. URL: https://unrealcommunity.wiki/61f1ff3b9c31042d6bb4e46b

21. C++ (Cpp) IOnlineSubsystem Examples. URL: https://cpp.hotexamples.com/examples/-/IOnlineSubsystem/-/cpp-ionlinesubsystem-class-examples.html

22. Bunny Hopping. URL: https://www.devx.com/terms/bunny-hopping/ (дата звернення: 15.02.2024)

23. What is a Bunny Hop. URL: www.computerhope.com (дата звернення: 15.02.2024)

24. Strafe Jumping Madness. URL: http://dk.toastednet.org/strafe_jump.htm

25. Опис ігор серії Quake українською, а також стисла історія id Software. URL: http://www.idsoftware.in.ua/ (дата звернення: 17.02.2024)

26. Quake репозиторій. URL: https://github.com/id-Software/Quake (дата звернення: 17.04.2024)

27. . Quake-III-Arena репозиторій. URL: https://github.com/id-Software/Quake-III-Arena (дата звернення: 16.04.2024)

28. Офіційний сайт Half-Life. URL: https://www.half-life.com/en/halflife (дата звернення: 17.04.2024)

29. Half-Life review. URL: http://www.gamespot.com/reviews/half-life - review/1900-2537398/ (дата звернення: 18.04.2024)

30. Half-Life’s 25th anniversary celebrations have caused a resonance cascade in its Steam player-count, surging to a new all-time high over the weekend. URL: https://www.pcgamer.com/half-lifes-25th-anniversary-celebrations-have-caused-a-resonance-cascade-in-its-steam-player-count-surging-to-a-new-all-time-high-over-the-weekend/ (дата звернення: 18.04.2024)

31. Automation Test Framework. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/automation-system-in-unreal-engine (дата звернення: 28.04.2024)

32. Automation Technical Guide. URL: https://docs.unrealengine.com/4.27/en-US/TestingAndOptimization/Automation/TechnicalGuide/ (дата звернення: 29.04.2024)

ДОДАТОК А

(обов’язковий)

ПРОГРАМНИЙ КОД ОСНОВНИХ МОДУЛІВ

Файл MBMovementComponent:

```
void UMBCharacterMovementComponent::CalcVelocity(float DeltaTime, float Friction, bool bFluid, float BrakingDeceleration)
{
	// UE4-COPY: void UCharacterMovementComponent::CalcVelocity(float DeltaTime, float Friction, bool bFluid, float BrakingDeceleration)

	// Do not update velocity when using root motion or when SimulatedProxy and not simulating root motion - SimulatedProxy are repped their Velocity
	if (!HasValidData() || HasAnimRootMotion() || DeltaTime < MIN_TICK_TIME || (CharacterOwner && CharacterOwner->GetLocalRole() == ROLE_SimulatedProxy && !bWasSimulatingRootMotion))
	{
		return;
	}

	Friction = FMath::Max(0.0f, Friction);
	const float MaxAccel = GetMaxAcceleration();
	float MaxSpeed = GetMaxSpeed();

	// Player doesn't path follow
#if 0
	// Check if path following requested movement
	bool bZeroRequestedAcceleration = true;
	FVector RequestedAcceleration = FVector::ZeroVector;
	float RequestedSpeed = 0.0f;
	if (ApplyRequestedMove(DeltaTime, MaxAccel, MaxSpeed, Friction, BrakingDeceleration, RequestedAcceleration, RequestedSpeed))
	{
		RequestedAcceleration = RequestedAcceleration.GetClampedToMaxSize(MaxAccel);
		bZeroRequestedAcceleration = false;
	}
#endif

	if (bForceMaxAccel)
	{
		// Force acceleration at full speed.
```

```
            // In consideration order for direction: Acceleration, then Velocity, then Pawn's rotation.
            if (Acceleration.SizeSquared() > SMALL_NUMBER)
            {
                Acceleration = Acceleration.GetSafeNormal() * MaxAccel;
            }
            else
            {
                Acceleration = MaxAccel * (Velocity.SizeSquared() < SMALL_NUMBER ? UpdatedComponent->GetForwardVector() : Velocity.GetSafeNormal());
            }

            AnalogInputModifier = 1.0f;
        }

#if 0
        // Path following above didn't care about the analog modifier, but we do for everything else below, so get the fully modified value.
        // Use max of requested speed and max speed if we modified the speed in ApplyRequestedMove above.
        const float MaxInputSpeed = FMath::Max(MaxSpeed * AnalogInputModifier, GetMinAnalogSpeed());
        MaxSpeed = FMath::Max(RequestedSpeed, MaxInputSpeed);
#else
        MaxSpeed = FMath::Max(MaxSpeed * AnalogInputModifier, GetMinAnalogSpeed());
#endif

        // Apply braking or deceleration
        const bool bZeroAcceleration = Acceleration.IsNearlyZero();
        const bool bIsGroundMove = IsMovingOnGround() && bBrakingWindowElapsed;

        // Apply friction
        if (bIsGroundMove)
        {
            const bool bVelocityOverMax = IsExceedingMaxSpeed(MaxSpeed);
            const FVector OldVelocity = Velocity;

            const float ActualBrakingFriction = (bUseSeparateBrakingFriction ? BrakingFriction : Friction) * SurfaceFriction;
            ApplyVelocityBraking(DeltaTime, ActualBrakingFriction, BrakingDeceleration);

            // Don't allow braking to lower us below max speed if we started above it.
            if (bVelocityOverMax && Velocity.SizeSquared() < FMath::Square(MaxSpeed) && FVector::DotProduct(Acceleration, OldVelocity) > 0.0f)
            {
```

```
                Velocity = OldVelocity.GetSafeNormal() * MaxSpeed;
            }
        }

        // Apply fluid friction
        if (bFluid)
        {
            Velocity = Velocity * (1.0f - FMath::Min(Friction * DeltaTime, 1.0f));
        }

        // Limit before
        Velocity.X = FMath::Clamp(Velocity.X, -AxisSpeedLimit, AxisSpeedLimit);
        Velocity.Y = FMath::Clamp(Velocity.Y, -AxisSpeedLimit, AxisSpeedLimit);

        // no clip
        if (bCheatFlying)
        {
            if (bZeroAcceleration)
            {
                Velocity = FVector(0.0f);
            }
            else
            {
                auto LookVec = CharacterOwner->GetControlRotation().Vector();
                auto LookVec2D = CharacterOwner->GetActorForwardVector();
                LookVec2D.Z = 0.0f;
                auto PerpendicularAccel = (LookVec2D | Acceleration) * LookVec2D;
                auto TangentialAccel = Acceleration - PerpendicularAccel;
                auto UnitAcceleration = Acceleration;
                auto Dir = UnitAcceleration.CosineAngle2D(LookVec);
                auto NoClipAccelClamp = MBCharacterOwner->IsSprinting() ? 2.0f *
MaxAcceleration : MaxAcceleration;
                Velocity = (Dir * LookVec * PerpendicularAccel.Size2D() +
TangentialAccel).GetClampedToSize(NoClipAccelClamp, NoClipAccelClamp);
            }
        }
        // ladder movement
        else if (bOnLadder)
        {
        }
        // walk move
        else
        {
            // Apply input acceleration
            if (!bZeroAcceleration)
```

```
		{
			// Clamp acceleration to max speed
			Acceleration = Acceleration.GetClampedToMaxSize2D(MaxSpeed);
			// Find veer
			const FVector AccelDir = Acceleration.GetSafeNormal2D();
			const float Veer = Velocity.X * AccelDir.X + Velocity.Y * AccelDir.Y;
			// Get add speed with air speed cap
			const	float	AddSpeed	=	(bIsGroundMove	?	Acceleration	:
Acceleration.GetClampedToMaxSize2D(AirSpeedCap)).Size2D() - Veer;
			if (AddSpeed > 0.0f)
			{
				// Apply acceleration
				const	float	AccelerationMultiplier	=	bIsGroundMove	?
GroundAccelerationMultiplier : AirAccelerationMultiplier;
				FVector CurrentAcceleration = Acceleration * AccelerationMultiplier *
SurfaceFriction * DeltaTime;
				CurrentAcceleration
CurrentAcceleration.GetClampedToMaxSize2D(AddSpeed);
				Velocity += CurrentAcceleration;
			}
		}

		// No requested accel on player
#if 0
		// Apply additional requested acceleration
		if (!bZeroRequestedAcceleration)
		{
			Velocity += RequestedAcceleration * DeltaTime;
		}
#endif
	}

	// Limit after
	Velocity.X = FMath::Clamp(Velocity.X, -AxisSpeedLimit, AxisSpeedLimit);
	Velocity.Y = FMath::Clamp(Velocity.Y, -AxisSpeedLimit, AxisSpeedLimit);

	const float SpeedSq = Velocity.SizeSquared2D();

	// Dynamic step height code for allowing sliding on a slope when at a high speed
	if (bOnLadder || SpeedSq <= MaxWalkSpeedCrouched * MaxWalkSpeedCrouched)
	{
		// If we're crouching or not sliding, just use max
		MaxStepHeight = DefaultStepHeight;
		SetWalkableFloorZ(DefaultWalkableFloorZ);
	}
```

```
          else
          {
               // Scale step/ramp height down the faster we go
               float Speed = FMath::Sqrt(SpeedSq);
               float SpeedScale = (Speed - SpeedMultMin) / (SpeedMultMax - SpeedMultMin);
               float SpeedMultiplier = FMath::Clamp(SpeedScale, 0.0f, 1.0f);
               SpeedMultiplier *= SpeedMultiplier;
               if (!IsFalling())
               {
                      // If we're on ground, factor in friction.
                      SpeedMultiplier = FMath::Max((1.0f - SurfaceFriction) * SpeedMultiplier,
0.0f);
               }
               MaxStepHeight = FMath::Lerp(DefaultStepHeight, MinStepHeight, SpeedMultiplier);
               SetWalkableFloorZ(FMath::Lerp(DefaultWalkableFloorZ, 0.9848f, SpeedMultiplier));
          }

          // Players don't use RVO avoidance
#if 0
          if (bUseRVOAvoidance)
          {
               CalcAvoidanceVelocity(DeltaTime);
          }
#endif
}
```

Файл MBCharacter:

```
// Copyright Epic Games, Inc. All Rights Reserved.

#include "AMBCharacter.h"
#include "../Weapons/Projectile.h" // ???
#include "Animation/AnimInstance.h"
#include "Camera/CameraComponent.h"
#include "Components/CapsuleComponent.h"
#include "EnhancedInputComponent.h"
#include "EnhancedInputSubsystems.h"
#include "Components/SkeletalMeshComponent.h"
#include "../Components/MBHealthComponent.h"
#include "../Controllers/MBPlayerController.h"
#include "../Components/MBCharacterMovementComponent.h"
#include "Components/TextBlock.h"
#include "Components/WidgetComponent.h"
#include "Launch/Resources/Version.h"
#include "Net/UnrealNetwork.h"
```

```
#include "../Weapons/Weapon.h"
#include "MBFPS/Components/MBCombatComponent.h"

#if ENGINE_MAJOR_VERSION == 5 && ENGINE_MINOR_VERSION >= 1
#include "Engine/DamageEvents.h"
#include "GameFramework/DamageType.h"
#endif
#include "CommonTextBlock.h"

#include "DrawDebugHelpers.h"
#include "Math/Rotator.h"
#include "MBAnimInstance.h"

static TAutoConsoleVariable<int32> CVarAutoBHop(TEXT("move.Pogo"), 1, TEXT("If holding spacebar should make the player jump whenever possible.\n"), ECVF_Default);

static TAutoConsoleVariable<int32> CVarJumpBoost(TEXT("move.JumpBoost"), 1, TEXT("If the player should boost in a movement direction while jumping.\n0 - disables jump boosting entirely\n1 - boosts in the direction of input, even when moving in another direction\n2 - boosts in the direction of input when moving in the same direction\n"), ECVF_Default);

static TAutoConsoleVariable<int32> CVarBunnyhop(TEXT("move.Bunnyhopping"), 0, TEXT("Enable normal bunnyhopping.\n"), ECVF_Default);

FOnCharacterEquipWeaponAMBCharacter::NotifyEquipWeapon;
FOnCharacterUnEquipWeaponAMBCharacter::NotifyUnEquipWeapon;

//////////////////////////////////////////////////////////////////////////
// AMBCharacter

AMBCharacter:: AMBCharacter (const FObjectInitializer& ObjectInitializer)
          :
Super(ObjectInitializer.SetDefaultSubobjectClass<UMBCharacterMovementComponent>(ACharacter::CharacterMovementComponentName))
{

          // Character doesnt have a rifle at start
          bHasRifle = true;
          // Set size for collision capsule
          GetCapsuleComponent()->InitCapsuleSize(30.48f, 38.58f);

          // Set collision settings. We are the invisible player with no 3rd person mesh.
          GetCapsuleComponent()->SetCollisionResponseToChannel(ECC_Camera, ECR_Block);

          // set our turn rates for input
```

```
	BaseTurnRate = 45.0f;
	BaseLookUpRate = 45.0f;

	// Camera eye level
	DefaultBaseEyeHeight = 53.34f;
	BaseEyeHeight = DefaultBaseEyeHeight;
	const float CrouchedHalfHeight = 68.58f / 2.0f;
	CrouchedEyeHeight = 53.34f - CrouchedHalfHeight;

	// Fall Damage Initializations
	// PLAYER_MAX_SAFE_FALL_SPEED
	MinSpeedForFallDamage = 1002.9825f;
	// PLAYER_MIN_BOUNCE_SPEED
	MinLandBounceSpeed = 329.565f;

	CapDamageMomentumZ = 476.25f;

	// Create a CameraComponent
	FirstPersonCameraComponent = CreateDefaultSubobject<UCameraComponent>(TEXT("FirstPersonCamera"));
	FirstPersonCameraComponent->SetupAttachment(GetCapsuleComponent());
	FirstPersonCameraComponent->SetRelativeLocation(FVector(-10.f, 0.f, 60.f)); // Position the camera
	FirstPersonCameraComponent->bUsePawnControlRotation = true;

	// Create a mesh component that will be used when being viewed from a '1st person' view (when controlling this pawn)
	Mesh1P = CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("CharacterMesh1P"));
	Mesh1P->SetOnlyOwnerSee(true);
	Mesh1P->SetupAttachment(RootComponent);
	Mesh1P->bCastDynamicShadow = false;
	Mesh1P->CastShadow = false;
	//Mesh1P->SetRelativeRotation(FRotator(0.9f, -19.19f, 5.2f));
	Mesh1P->SetRelativeLocation(FVector(-30.f, 0.f, -150.f));

	MBMovementComponent = Cast<UMBCharacterMovementComponent>(GetCharacterMovement());
	//CombatComponent = CreateDefaultSubobject<UMBCombatComponent>(TEXT("CombatComponent"));
	//CombatComponent->SetIsReplicated(true);
	HealthComponent = CreateDefaultSubobject<UMBHealthComponent>(TEXT("HealthComponent"));


	PlayerController = Cast<AMBPlayerController>(UGameplayStatics::GetPlayerController(this, 0));
```

```
	//Initialize weapons
	//
	if (!AssaultRifle)
	{
		AssaultRifle = CreateDefaultSubobject<AWeapon>(TEXT("PainKiller"));
		//AssaultRifle->SetupAttachment(FirstPersonCameraComponent);
	}

	bWantsToFire = false;
}

void AMBCharacter::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const
{
	Super::GetLifetimeReplicatedProps(OutLifetimeProps);

	// only to local owner: weapon change requests are locally instigated, other clients
don't need it
	DOREPLIFETIME_CONDITION(AMBCharacter, Inventory, COND_OwnerOnly);

	// everyone
	DOREPLIFETIME(AMBCharacter, CurrentWeapon);
}

void AMBCharacter::PostInitializeComponents()
{
	Super::PostInitializeComponents();

	if (GetLocalRole() == ROLE_Authority)
	{
		//Health = GetMaxHealth();

		// Needs to happen after character is added to repgraph
		GetWorldTimerManager().SetTimerForNextTick(this,                                    &
AMBCharacter::SpawnDefaultInventory);
	}
	//if (CombatComponent)
	//{
	//	CombatComponent->Character = this;
	//}
	//else
	//{
	//	if (GEngine)
	//	{
	//		GEngine->AddOnScreenDebugMessage(
	//			1,
```

```
		//				3.f,
		//				FColor::Blue,
		//				FString("No CombatComponent")

		//		);
		//	}
		//}

}

USkeletalMeshComponent* AMBCharacter::GetPawnMesh() const
{
		return IsFirstPerson() ? Mesh1P : GetMesh();
}

bool AMBCharacter::IsFirstPerson() const
{
		return Controller && Controller->IsLocalPlayerController();
}

void AMBCharacter::BeginPlay()
{
		// Call the base class
		Super::BeginPlay();
		PlayerController->SetInputMode(FInputModeGameOnly());
		UE_LOG(LogTemp, Log, TEXT("Mvt Comp Class = %s"),*GetCharacterMovement()->GetClass()->GetName())
		if (UEnhancedInputLocalPlayerSubsystem* Subsystem = ULocalPlayer::GetSubsystem<UEnhancedInputLocalPlayerSubsystem>(PlayerController->GetLocalPlayer()))
		{
			Subsystem->AddMappingContext(DefaultMappingContext, 0);
		}
		// Max jump time to get to the top of the arc
		MaxJumpTime = -4.0f * GetCharacterMovement()->JumpZVelocity / (3.0f * GetCharacterMovement()->GetGravityZ());

}

void AMBCharacter::Tick(float DeltaSeconds)
{
		Super::Tick(DeltaSeconds);

		if (bDeferJumpStop)
		{
```

```
            bDeferJumpStop = false;
            Super::StopJumping();
        }
        //
}

void AMBCharacter::SetCurrentWeapon(AWeapon* NewWeapon, AWeapon* LastWeapon)
{
        AWeapon* LocalLastWeapon = nullptr;

        if (LastWeapon != NULL)
        {
            LocalLastWeapon = LastWeapon;
        }
        else if (NewWeapon != CurrentWeapon)
        {
            LocalLastWeapon = CurrentWeapon;
        }

        // unequip previous
        if (LocalLastWeapon)
        {
            LocalLastWeapon->OnUnEquip();
        }

        CurrentWeapon = NewWeapon;

        // equip new one
        if (NewWeapon)
        {
            NewWeapon->SetOwningPawn(this);     // Make sure weapon's MyPawn is pointing back
to us. During replication, we can't guarantee APawn::CurrentWeapon will rep after AWeapon::MyPawn!

            NewWeapon->OnEquip(LastWeapon);
        }
}

void AMBCharacter::OnRep_CurrentWeapon(AWeapon* LastWeapon)
{
        SetCurrentWeapon(CurrentWeapon, LastWeapon);
}

void AMBCharacter::SpawnDefaultInventory()
{
        if (GetLocalRole() < ROLE_Authority)
```

```
	{
		return;
	}

	int32 NumWeaponClasses = DefaultInventoryClasses.Num();
	for (int32 i = 0; i < NumWeaponClasses; i++)
	{
		if (DefaultInventoryClasses[i])
		{
			FActorSpawnParameters SpawnInfo;
			SpawnInfo.SpawnCollisionHandlingOverride =
ESpawnActorCollisionHandlingMethod::AlwaysSpawn;
			AWeapon* NewWeapon = GetWorld()-
>SpawnActor<AWeapon>(DefaultInventoryClasses[i], SpawnInfo);
			AddWeapon(NewWeapon);
		}
	}

	// equip first weapon in inventory
	if (Inventory.Num() > 0)
	{
		EquipWeapon(Inventory[0]);
	}
}

void AMBCharacter::DestroyInventory()
{
	if (GetLocalRole() < ROLE_Authority)
	{
		return;
	}

	// remove all weapons from inventory and destroy them
	for (int32 i = Inventory.Num() - 1; i >= 0; i--)
	{
		AWeapon* Weapon = Inventory[i];
		if (Weapon)
		{
			RemoveWeapon(Weapon);
			Weapon->Destroy();
		}
	}
}

bool AMBCharacter::ServerEquipWeapon_Validate(AWeapon* Weapon)
```

```
{
	return true;
}

void AMBCharacter::ServerEquipWeapon_Implementation(AWeapon* Weapon)
{
	EquipWeapon(Weapon);
}

void AMBCharacter::SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent)
{
	Super::SetupPlayerInputComponent(PlayerInputComponent);
	// Set up action bindings
	//
		// Get the Enhanced Input Local Player Subsystem from the Local Player related to our Player Controller.

	if (UEnhancedInputComponent* EnhancedInputComponent = CastChecked<UEnhancedInputComponent>(PlayerInputComponent))
	{
		//Jumping
		EnhancedInputComponent->BindAction(JumpAction, ETriggerEvent::Triggered, this, &AMBCharacter::Jump);
		EnhancedInputComponent->BindAction(JumpAction, ETriggerEvent::Completed, this, &AMBCharacter::StopJumping);

		// Sprinting
		EnhancedInputComponent->BindAction(SprintAction, ETriggerEvent::Started, MBMovementComponent, &UMBCharacterMovementComponent::SprintPressed);
		EnhancedInputComponent->BindAction(SprintAction, ETriggerEvent::Completed, MBMovementComponent, &UMBCharacterMovementComponent::SprintReleased);

		// Crouching... for now
		EnhancedInputComponent->BindAction(CrouchAction, ETriggerEvent::Started, MBMovementComponent, &UMBCharacterMovementComponent::CrouchPressed);
		EnhancedInputComponent->BindAction(CrouchAction, ETriggerEvent::Completed, MBMovementComponent, &UMBCharacterMovementComponent::CrouchReleased);

		//Moving
		EnhancedInputComponent->BindAction(MoveAction, ETriggerEvent::Triggered, this, &AMBCharacter::Move);

		//Looking
		//EnhancedInputComponent->BindAction(LookAction, ETriggerEvent::Started, this, &AMBCharacter::TouchBegin);
```

```
            EnhancedInputComponent->BindAction(LookAction,   ETriggerEvent::Triggered,   this,
&AMBCharacter::LookUp);
            EnhancedInputComponent->BindAction(TurnAction,   ETriggerEvent::Triggered,   this,
&AMBCharacter::Turn);
            EnhancedInputComponent->BindAction(PickUpAction,  ETriggerEvent::Triggered,  this,
&AMBCharacter::PickUpWeapon);

            //Weapon
            EnhancedInputComponent->BindAction(NextWeaponAction,     ETriggerEvent::Triggered,
this, & AMBCharacter::OnNextWeapon);
            EnhancedInputComponent->BindAction(PrevWeaponAction,     ETriggerEvent::Triggered,
this, & AMBCharacter::OnPrevWeapon);
            EnhancedInputComponent->BindAction(FireAction,  ETriggerEvent::Started,  this,  &
AMBCharacter::StartWeaponFire);
            EnhancedInputComponent->BindAction(FireAction,  ETriggerEvent::Completed,  this,  &
AMBCharacter::StopWeaponFire);

            //EnhancedInputComponent->BindAction(OptionMenuAction,   ETriggerEvent::Triggered,
this, & AMBCharacter::OpenOptionMenu);
        }

}

void AMBCharacter::PickUpWeapon(const FInputActionValue& Value)
{
        //if (CombatComponent)
        //{
        //  if (HasAuthority())
        //  {
        //          CombatComponent->EquipWeapon(OverlappingWeapon);
        //  }
        //  else
        //  {
        //          ServerPickUpWeapon();
        //  }
        //}
}

void AMBCharacter::ServerPickUpWeapon_Implementation()
{
        //if (CombatComponent)
        //{
        //  CombatComponent->EquipWeapon(OverlappingWeapon);
        //}
}
```

```
void AMBCharacter::Destroyed()
{
	Super::Destroyed();
	DestroyInventory();
}

void AMBCharacter::AddWeapon(AWeapon* Weapon)
{
	if (Weapon && GetLocalRole() == ROLE_Authority)
	{
		Weapon->OnEnterInventory(this);
		Inventory.AddUnique(Weapon);
	}
}

void AMBCharacter::RemoveWeapon(AWeapon* Weapon)
{
	if (Weapon && GetLocalRole() == ROLE_Authority)
	{
		Weapon->OnLeaveInventory();
		Inventory.RemoveSingle(Weapon);
	}
}

AWeapon* AMBCharacter::FindWeapon(TSubclassOf<AWeapon> WeaponClass)
{
	for (int32 i = 0; i < Inventory.Num(); i++)
	{
		if (Inventory[i] && Inventory[i]->IsA(WeaponClass))
		{
			return Inventory[i];
		}
	}

	return NULL;
}

void AMBCharacter::EquipWeapon(AWeapon* Weapon)
{
	if (Weapon)
	{
		if (GetLocalRole() == ROLE_Authority)
		{
			SetCurrentWeapon(Weapon, CurrentWeapon);
```

```
            }
            else
            {
                ServerEquipWeapon(Weapon);
            }
        }
}

void AMBCharacter::StartWeaponFire()
{
        if (!bWantsToFire)
        {
            bWantsToFire = true;
            if (CurrentWeapon)
            {
                CurrentWeapon->StartFire();
            }
        }
}

void AMBCharacter::StopWeaponFire()
{
        if (bWantsToFire)
        {
            bWantsToFire = false;
            if (CurrentWeapon)
            {
                CurrentWeapon->StopFire();
            }
        }
}

bool AMBCharacter::CanFire() const
{
        return true;//IsAlive();
}

bool AMBCharacter::CanReload() const
{
        return true;
}

FName AMBCharacter::GetWeaponAttachPoint() const
{
        return WeaponAttachPoint;
```

```
}

bool AMBCharacter::IsWeaponEquipped()
{
	return false;//return (CombatComponent && CombatComponent->EquippedWeapon);
}

bool AMBCharacter::IsFiring() const
{
	return bWantsToFire;
}

void AMBCharacter::OnNextWeapon(const FInputActionValue& Value)
{
	//AMBPlayerController* MyPC = Cast<AShooterPlayerController>(Controller);
	UE_LOG(LogTemp, Warning, TEXT("OnNextWeapon"));
	if (PlayerController)// && MyPC->IsGameInputAllowed())
	{
		if (Inventory.Num() >= 2 && (CurrentWeapon == NULL || CurrentWeapon->GetCurrentState() != EWeaponState::Equipping))
		{
			const int32 CurrentWeaponIdx = Inventory.IndexOfByKey(CurrentWeapon);
			AWeapon* NextWeapon = Inventory[(CurrentWeaponIdx + 1) % Inventory.Num()];
			EquipWeapon(NextWeapon);
		}
	}
}

void AMBCharacter::OnPrevWeapon(const FInputActionValue& Value)
{
	//AShooterPlayerController* MyPC = Cast<AShooterPlayerController>(Controller);
	if (PlayerController)// && MyPC->IsGameInputAllowed())
	{
		if (Inventory.Num() >= 2 && (CurrentWeapon == NULL || CurrentWeapon->GetCurrentState() != EWeaponState::Equipping))
		{
			const int32 CurrentWeaponIdx = Inventory.IndexOfByKey(CurrentWeapon);
			AWeapon* PrevWeapon = Inventory[(CurrentWeaponIdx - 1 + Inventory.Num()) % Inventory.Num()];
			EquipWeapon(PrevWeapon);
		}
	}
}
```

```
void AMBCharacter::ApplyDamageMomentum(float DamageTaken, FDamageEvent const& DamageEvent, APawn*
PawnInstigator, AActor* DamageCauser)
{
	UDamageType const* const DmgTypeCDO = DamageEvent.DamageTypeClass-
>GetDefaultObject<UDamageType>();
	if (GetCharacterMovement())
	{
		FVector ImpulseDir;

		if (IsValid(DamageCauser))
		{
			ImpulseDir = (GetActorLocation() - DamageCauser-
>GetActorLocation()).GetSafeNormal();
		}
		else
		{
			FHitResult HitInfo;
			DamageEvent.GetBestHitInfo(this, DamageCauser, HitInfo, ImpulseDir);
		}

		const float SizeFactor = (60.96f * 60.96f * 137.16f) /
(FMath::Square(GetCapsuleComponent()->GetScaledCapsuleRadius() * 2.0f) * GetCapsuleComponent()-
>GetScaledCapsuleHalfHeight() * 2.0f);

		float Magnitude = 1.905f * DamageTaken * SizeFactor * 5.0f;
		Magnitude = FMath::Min(Magnitude, 1905.0f);

		FVector Impulse = ImpulseDir * Magnitude;
		bool const bMassIndependentImpulse = !DmgTypeCDO->bScaleMomentumByMass;
		float MassScale = 1.f;
		if (!bMassIndependentImpulse && GetCharacterMovement()->Mass > SMALL_NUMBER)
		{
			MassScale = 1.f / GetCharacterMovement()->Mass;
		}
		if (CapDamageMomentumZ > 0.f)
		{
			Impulse.Z = FMath::Min(Impulse.Z * MassScale, CapDamageMomentumZ) /
MassScale;
		}

		GetCharacterMovement()->AddImpulse(Impulse, bMassIndependentImpulse);
	}
}

void AMBCharacter::ClearJumpInput(float DeltaTime)
```

```
{
          // Don't clear jump input right away if we're auto hopping or noclipping (holding to go
up), or if we are deferring a jump stop
          if (CVarAutoBHop.GetValueOnGameThread() != 0 || bAutoBunnyhop || GetCharacterMovement()-
>bCheatFlying || bDeferJumpStop)
          {
               return;
          }
          Super::ClearJumpInput(DeltaTime);
}

void AMBCharacter::Move(const FInputActionValue& Value)
{
          if (!FMath::IsNearlyZero(Value.GetMagnitude()))
          {
               // add movement in that direction
               FVector MovementVector = Value.Get<FVector>();
               AddMovementInput(GetActorForwardVector(), MovementVector.Y);
               AddMovementInput(GetActorRightVector(), MovementVector.X);
          }

}

void AMBCharacter::Jump()
{
          if (GetCharacterMovement()->IsFalling())
          {
               bDeferJumpStop = true;
          }

          Super::Jump();
}

void AMBCharacter::OnMovementModeChanged(EMovementMode PrevMovementMode, uint8 PrevCustomMode)
{
          if (!bPressedJump)
          {
               ResetJumpState();
          }

          if (GetCharacterMovement()->IsFalling())
          {
               // Record jump force start time for proxies. Allows us to expire the jump even if
not continually ticking down a timer.
               if (bProxyIsJumpForceApplied)
```

```
            {
                    ProxyJumpForceStartedTime = GetWorld()->GetTimeSeconds();
            }
        }
        else
        {
            JumpCurrentCount = 0;
            JumpKeyHoldTime = 0.0f;
            JumpForceTimeRemaining = 0.0f;
            bWasJumping = false;
        }

        K2_OnMovementModeChanged(PrevMovementMode,        GetCharacterMovement()->MovementMode,
PrevCustomMode, GetCharacterMovement()->CustomMovementMode);
        MovementModeChangedDelegate.Broadcast(this, PrevMovementMode, PrevCustomMode);
}

void AMBCharacter::StopJumping()
{
        if (!bDeferJumpStop)
        {
            Super::StopJumping();
        }
}

void AMBCharacter::OnJumped_Implementation()
{
        const int32 JumpBoost = CVarJumpBoost->GetInt();
        //if (MBMovementComponent->IsOnLadder())
        //{
        //   return;
        //}

        if (!MBMovementComponent)
        {
            UE_LOG(LogTemp,Warning,TEXT("NO MOVEMENT COMPONENT"))
            return;
        }

        if (GetWorld()->GetTimeSeconds() >= LastJumpBoostTime + MaxJumpTime && JumpBoost)
        {
            LastJumpBoostTime = GetWorld()->GetTimeSeconds();
            // Boost forward speed on jump
            FVector Facing = GetActorForwardVector();
            // Use input direction
```

```
		FVector Input = GetCharacterMovement()->GetCurrentAcceleration();
		if (JumpBoost != 1)
		{
			// Only boost input in the direction of current movement axis (prevents ABH).
			Input *= FMath::Max(Input.GetSafeNormal2D() | GetCharacterMovement()->Velocity.GetSafeNormal2D(), 0.0f);
		}
		float ForwardSpeed = Input | Facing;
		// Adjust how much the boost is
		float SpeedBoostPerc = bIsSprinting || bIsCrouched ? 0.1f : 0.5f;
		// How much we are boosting by
		float SpeedAddition = FMath::Abs(ForwardSpeed * SpeedBoostPerc);
		// We can only boost up to this much
		float MaxBoostedSpeed = GetCharacterMovement()->GetMaxSpeed() + GetCharacterMovement()->GetMaxSpeed() * SpeedBoostPerc;
		// Calculate new speed
		float NewSpeed = SpeedAddition + GetMovementComponent()->Velocity.Size2D();
		float SpeedAdditionNoClamp = SpeedAddition;

		// Scale the boost down if we are going over
		if (NewSpeed > MaxBoostedSpeed)
		{
			SpeedAddition -= NewSpeed - MaxBoostedSpeed;
		}

		if (ForwardSpeed < -MBMovementComponent->GetMaxAcceleration() * FMath::Sin(0.6981f))
		{
			// Boost backwards if we're going backwards
			SpeedAddition *= -1.0f;
			SpeedAdditionNoClamp *= -1.0f;
		}

		// Boost our velocity
		FVector JumpBoostedVel = GetMovementComponent()->Velocity + Facing * SpeedAddition;
		float JumpBoostedSizeSq = JumpBoostedVel.SizeSquared2D();
		if (CVarBunnyhop.GetValueOnGameThread() != 0)
		{
			FVector JumpBoostedUnclampVel = GetMovementComponent()->Velocity + Facing * SpeedAdditionNoClamp;
			float JumpBoostedUnclampSizeSq = JumpBoostedUnclampVel.SizeSquared2D();
			if (JumpBoostedUnclampSizeSq > JumpBoostedSizeSq)
			{
				JumpBoostedVel = JumpBoostedUnclampVel;
				JumpBoostedSizeSq = JumpBoostedUnclampSizeSq;
			}
```

```
            }
            if (GetMovementComponent()->Velocity.SizeSquared2D() < JumpBoostedSizeSq)
            {
                    GetMovementComponent()->Velocity = JumpBoostedVel;
            }
        }
}

void AMBCharacter::ToggleNoClip()
{
        MBMovementComponent->ToggleNoClip();
}

// Sample for multiplayer games with a Mesh3P with crouch support
#if 0
void AMBCharacter::OnEndCrouch(float HalfHeightAdjust, float ScaledHalfHeightAdjust)
{
        Super::OnEndCrouch(HalfHeightAdjust, ScaledHalfHeightAdjust);
        const AMBCharacter * DefaultChar = GetDefault< AMBCharacter >(GetClass());
        if (Mesh3P && DefaultChar->Mesh3P)
        {
            FVector MeshRelativeLocation = Mesh3P->GetRelativeLocation();
            MeshRelativeLocation.Z     =     DefaultChar->Mesh3P->GetRelativeLocation().Z     -
ScaledHalfHeightAdjust;
            Mesh3P->SetRelativeLocation(MeshRelativeLocation);
        }
}

void AMBCharacter::OnStartCrouch(float HalfHeightAdjust, float ScaledHalfHeightAdjust)
{
        Super::OnStartCrouch(HalfHeightAdjust, ScaledHalfHeightAdjust);
        const AMBCharacter * DefaultChar = GetDefault< AMBCharacter >(GetClass());
        if (Mesh3P && DefaultChar->Mesh3P)
        {
            FVector MeshRelativeLocation = Mesh3P->GetRelativeLocation();
            MeshRelativeLocation.Z     =     DefaultChar->Mesh3P->GetRelativeLocation().Z     +
ScaledHalfHeightAdjust;
            Mesh3P->SetRelativeLocation(MeshRelativeLocation);
        }
}
#endif

bool AMBCharacter::CanJumpInternal_Implementation() const
{
        // UE-COPY: ACharacter::CanJumpInternal_Implementation()
```

```
	bool bCanJump = GetCharacterMovement() && GetCharacterMovement()->IsJumpAllowed();

	//if (bCanJump)
	//{
	//	// Ensure JumpHoldTime and JumpCount are valid.
		if (!bWasJumping || GetJumpMaxHoldTime() <= 0.0f)
		{
			if (JumpCurrentCount == 0 && GetCharacterMovement()->IsFalling())
			{
				bCanJump = JumpCurrentCount + 1 < JumpMaxCount;
			}
			else
			{
				bCanJump = JumpCurrentCount < JumpMaxCount;
			}
		}
	//	else
	//	{
	//		// Only consider JumpKeyHoldTime as long as:
	//		// A) We are on the ground
	//		// B) The jump limit hasn't been met OR
	//		// C) The jump limit has been met AND we were already jumping
	//		const bool bJumpKeyHeld = (bPressedJump && JumpKeyHoldTime < GetJumpMaxHoldTime());
	//		bCanJump = bJumpKeyHeld &&
	//			(GetCharacterMovement()->IsMovingOnGround() || (JumpCurrentCount < JumpMaxCount) || (bWasJumping && JumpCurrentCount == JumpMaxCount));
	//	}
	//	if (GetCharacterMovement()->IsMovingOnGround())
	//	{
	//		float FloorZ = FVector(0.0f, 0.0f, 1.0f) | GetCharacterMovement()->CurrentFloor.HitResult.ImpactNormal;
	//		float WalkableFloor = GetCharacterMovement()->GetWalkableFloorZ();
	//		bCanJump &= (FloorZ >= WalkableFloor || FMath::IsNearlyEqual(FloorZ, WalkableFloor));
	//	}
	//}

	return bCanJump;
}

void AMBCharacter::Look(const FInputActionValue& Value)
{
	// input is a Vector2D
```

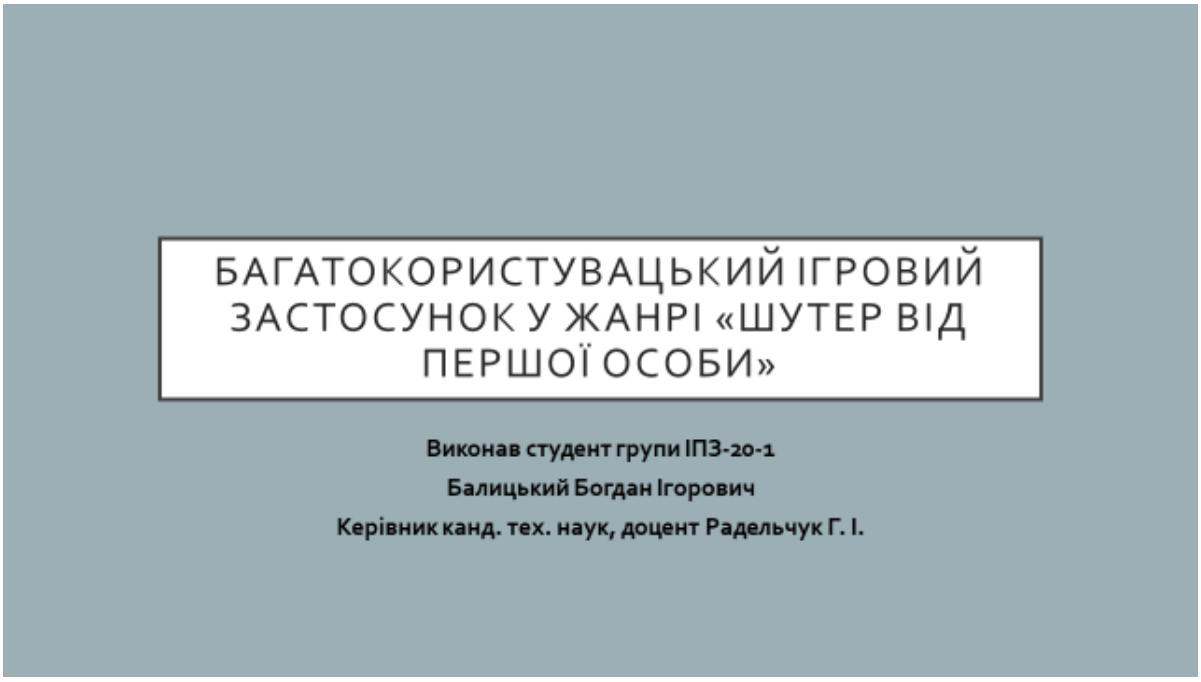
```
          FVector2D LookAxisVector = Value.Get<FVector2D>();

          if (PlayerController != nullptr)
          {
               // add yaw and pitch input to controller
               AddControllerYawInput(LookAxisVector.X);
               AddControllerPitchInput(LookAxisVector.Y);
          }
}

void AMBCharacter::Turn(const FInputActionValue& Value)//bool bIsPure, float Rate)
{
          float Rate = Value.GetMagnitude();
          //if (!bIsPure)
          //{
               Rate = Rate * BaseTurnRate * GetWorld()->GetDeltaSeconds();
          //}

          // calculate delta for this frame from the rate information
          AddControllerYawInput(Rate);
}
void AMBCharacter::LookUp(const FInputActionValue& Value)//bool bIsPure, float Rate)
{
          float    Rate    =    Value.GetMagnitude();//Value.Get<FVector>().GetSafeNormal().Size();
//Value.Get<FVector>().Size();
          /*if (!bIsPure)
          {*/
          Rate = Rate * BaseLookUpRate * GetWorld()->GetDeltaSeconds();
          //}

          // calculate delta for this frame from the rate information
          AddControllerPitchInput(Rate);
}

void AMBCharacter::RecalculateBaseEyeHeight()
{
          const ACharacter* DefaultCharacter = GetClass()->GetDefaultObject<ACharacter>();
          const    float    OldUnscaledHalfHeight    =    DefaultCharacter->GetCapsuleComponent()-
>GetUnscaledCapsuleHalfHeight();
          const float CrouchedHalfHeight = GetCharacterMovement()->GetCrouchedHalfHeight();
          const float FullCrouchDiff = OldUnscaledHalfHeight - CrouchedHalfHeight;
          const UCapsuleComponent* CharacterCapsule = GetCapsuleComponent();
          const         float         CurrentUnscaledHalfHeight         =         CharacterCapsule-
>GetUnscaledCapsuleHalfHeight();
```

```
	const float CurrentAlpha = 1.0f - (CurrentUnscaledHalfHeight - CrouchedHalfHeight) / FullCrouchDiff;
	BaseEyeHeight = FMath::Lerp(DefaultCharacter->BaseEyeHeight, CrouchedEyeHeight, SimpleSpline(CurrentAlpha));
}

bool AMBCharacter::CanCrouch() const
{
	return !GetCharacterMovement()->bCheatFlying && Super::CanCrouch() && !MBMovementComponent->IsOnLadder();
}

void AMBCharacter::TouchBegin(const FInputActionValue& Value)
{
	bool bIsCurrentlyPressed;
	PlayerController->GetInputTouchState(ETouchIndex::Touch1, previousTouchLocation.X, previousTouchLocation.Y, bIsCurrentlyPressed);
}

void AMBCharacter::SetHasRifle(bool bNewHasRifle)
{
	bHasRifle = true; //= bNewHasRifle;
}
```

ДОДАТОК Б

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ



Слайд 1 – Титулка

Слайд 2 – Предметна область та актуальність теми роботи

ПОСТАНОВКА ЗАДАЧІ

Метою кваліфікаційної роботи є розроблення багатокористувацької гри у жанрі шутера від першої особи з динамічним функціоналом, у якому передбачено підключення гравців, динамічну систему руху, зрозумілий інтерфейс.

Для досягнення мети слід вирішити такі задачі:

- провести аналіз існуючих рішень у жанрі FPS;
- сформувати вимоги до застосунку;
- спроектувати архітектуру та структуру застосунку;
- вибрати відповідні інструменти та технології для реалізації проекту;
- виконати програмну реалізацію та тестування застосунку.

Слайд 3 – Постановка задачі

ІСНУЮЧІ РІШЕННЯ

- Quake Champions

- Overwatch

Слайд 4 – Існуючі рішення

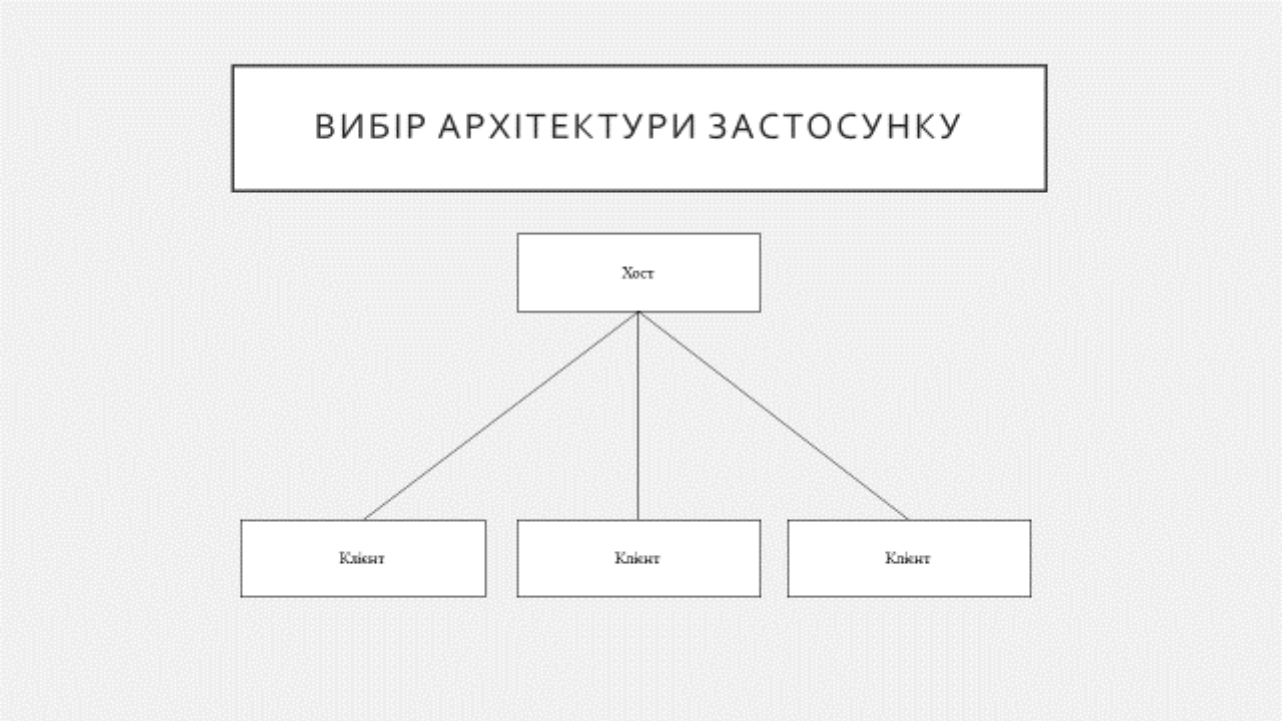
АНАЛІЗ ВИМОГ ДО ЗАСТОСУНКУ

- Гравці повинні мати можливість приєднуватися до існуючих ігор або створювати нові.
- Повинна бути реалізована система управління персонажем від першої особи з можливістю стрільби, пересування, взаємодії з об'єктами.
- Система повинна обробляти та зберігати дані про ігровий стан у реальному часі, включаючи позиції гравців, їхнє здоров'я, використання зброї та інші події.
- Інтерфейс користувача повинен бути інтуїтивно зрозумілим і простим у використанні.

Слайд 5 – Аналіз вимог до застосунку

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ

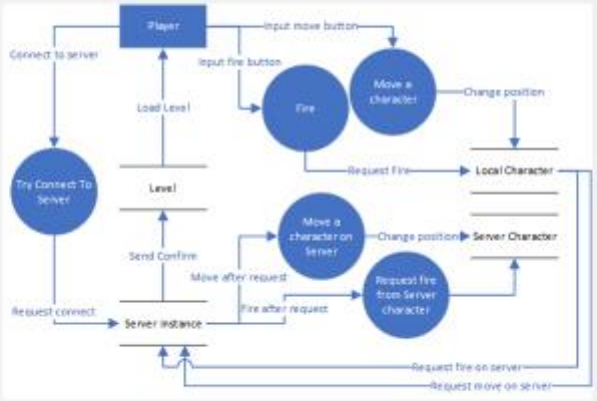
Слайд 6 – Діаграма варіантів використання



Слайд 7 – Вибір архітектури застосунку

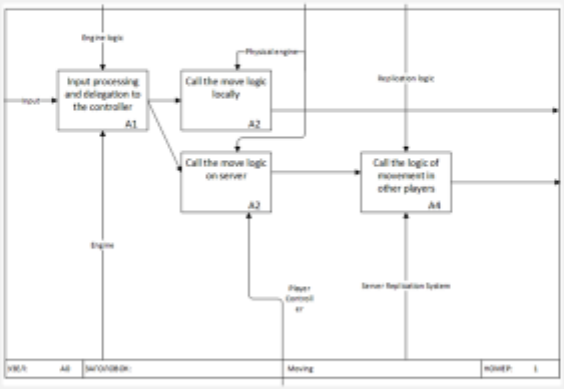
Слайд 8 – Загальний алгоритм гри

ПРОЄКТУВАННЯ:DFD-ДІАГРАМА



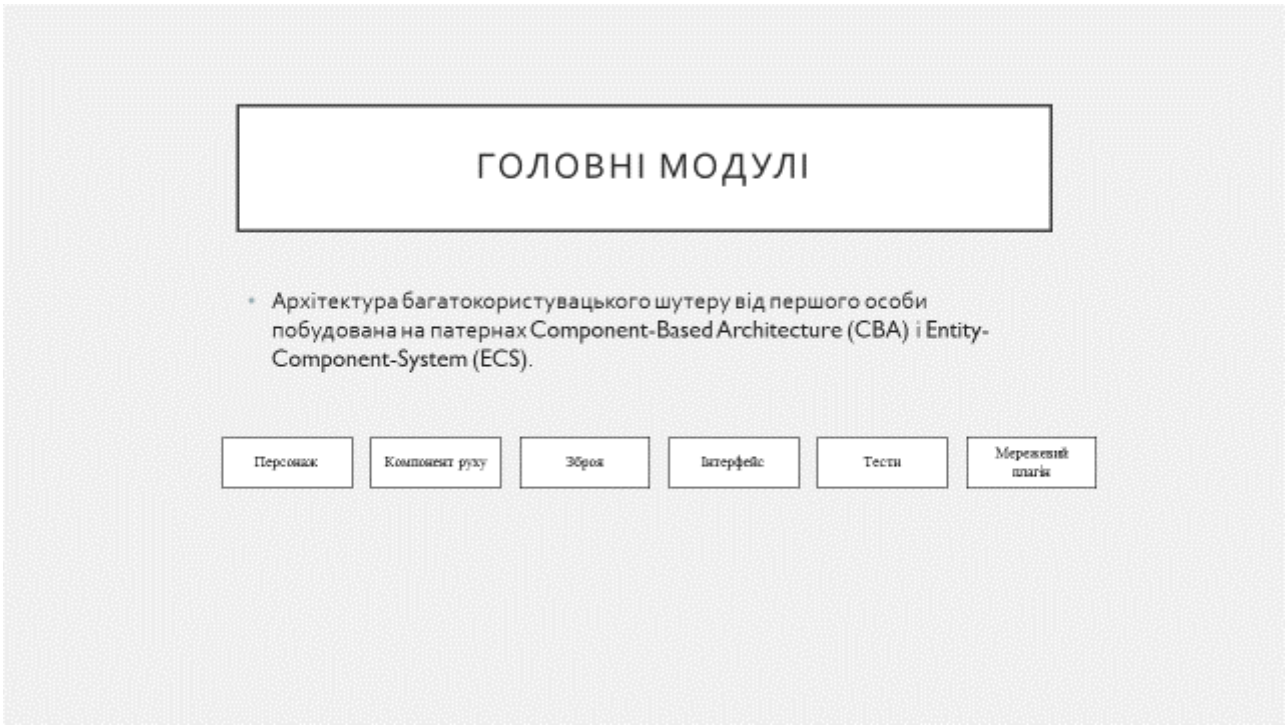
Слайд 9 – DFD-діаграма

ПРОЄКТУВАННЯ:ДІАГРАМА ПРОЦЕСІВ ГРИ



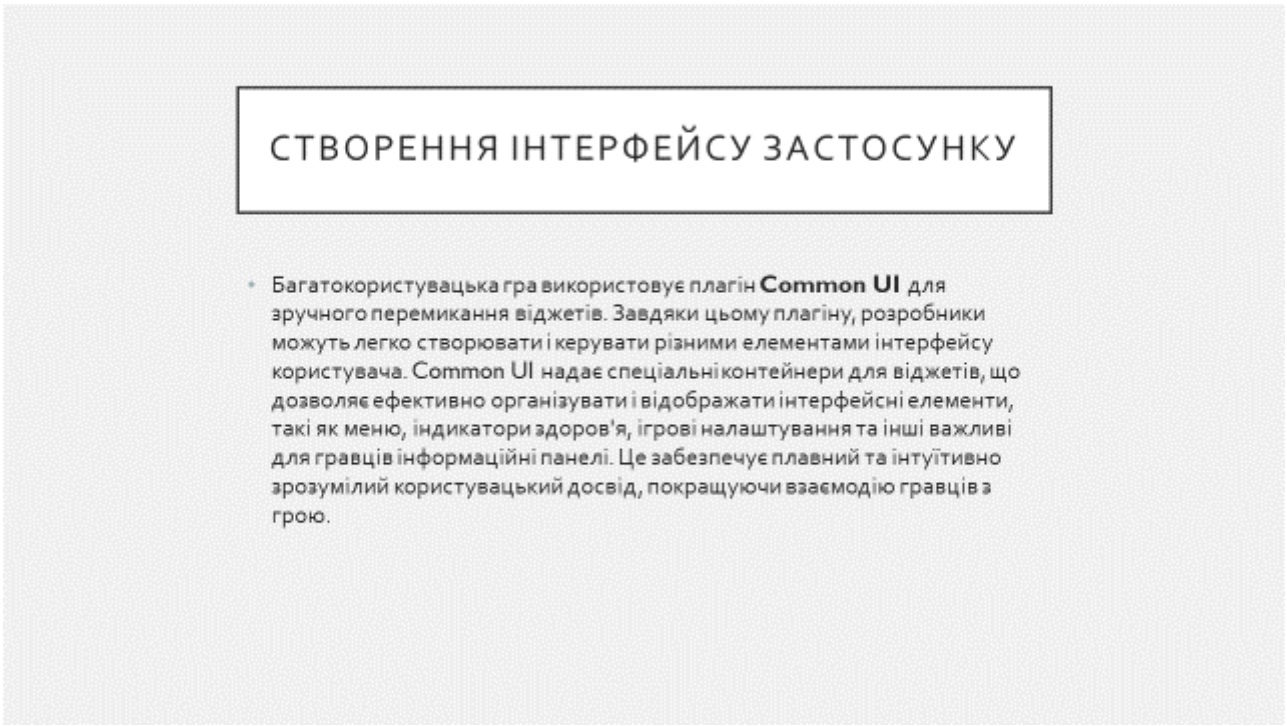
Слайд 10 – Діаграма процесів гри

Слайд 11 – Діаграма класів

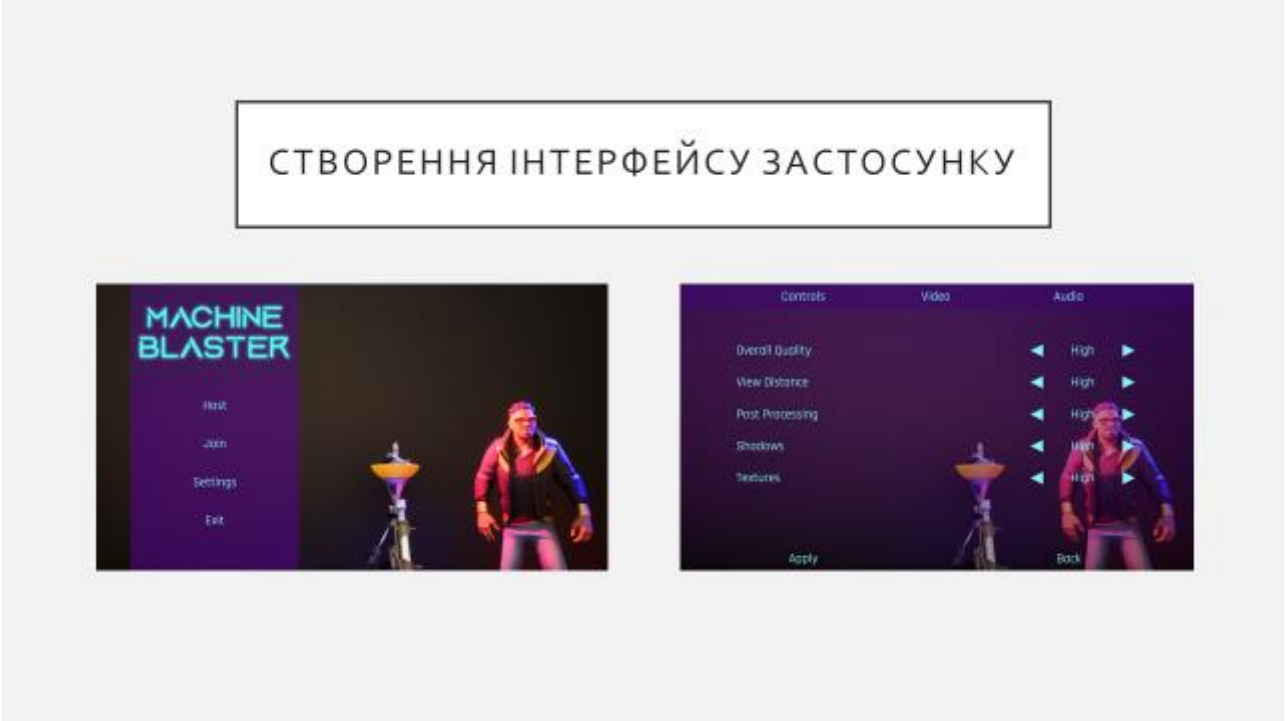


Слайд 12 – Головні модулі

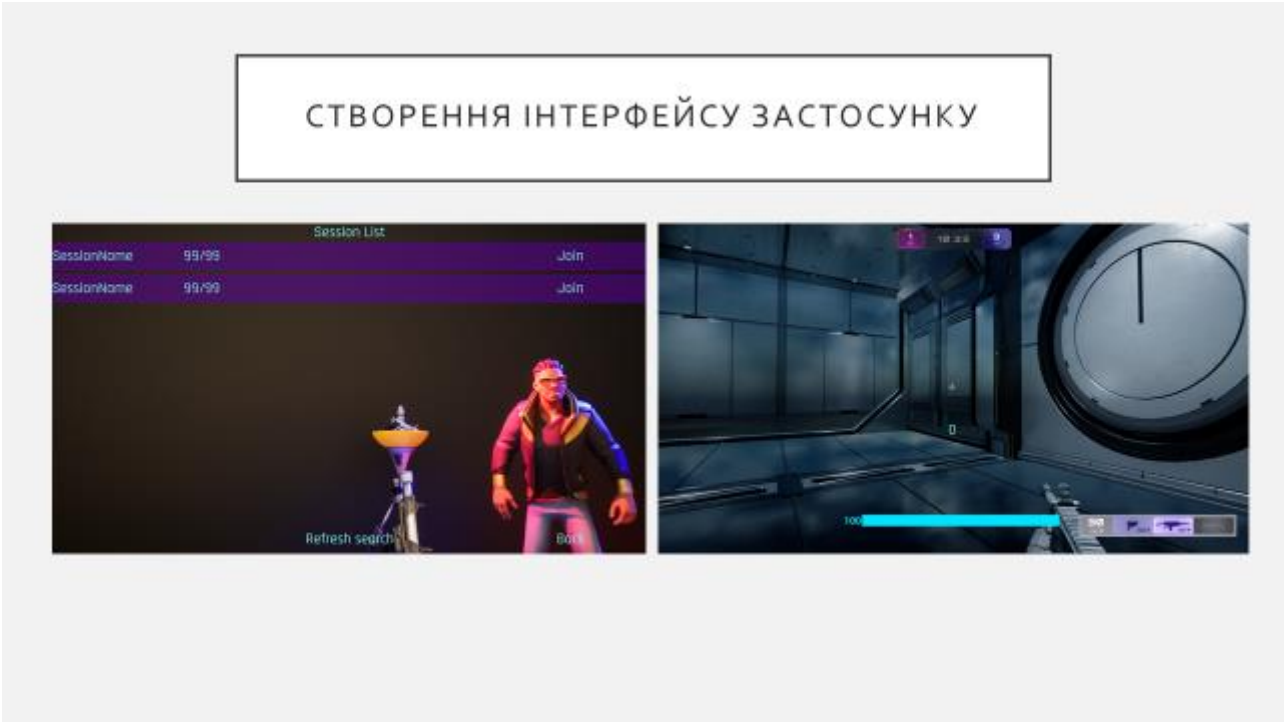
Слайд 13 – Технології реалізації застосунку



Слайд 14 – Створення інтерфейсу застосунку



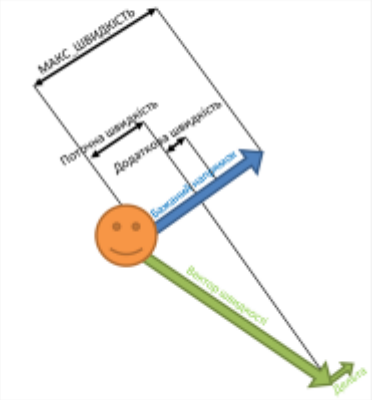
Слайд 15 – Меню, налаштування



Слайд 16 – пошук сесій, HUD

КОМПОНЕНТ РУХУ

- Бажаний напрямок використовується, щоб оновлювати напрямок швидкості по горизонталі, вертикальний напрямок швидкості обчислюється за простою гравітаційною моделлю.
- Коли персонаж знаходиться на землі – сила тертя впливає на його швидкість. Поточна швидкість – це проекція вектору швидкості на бажаний напрямок. Додаткова швидкість дорівнює різниці між максимальною швидкістю та поточною швидкістю обмежена максимальним прискоренням
- Така система руху дозволяє зберігати і нарощувати швидкість за рахунок послідовних стрибків без втрати швидкості між ними. Гравець використовує ритмічні стрибки в поєднанні з обертанням миші та натисканням стрейф-клавіш.



Слайд 17 – Компонент руху

ВОГОНЬ ЗІ ЗБРОЇ МИТТЄВОГО ВЛУЧЕННЯ

- Коли гравець стріляє зі зброї, перевірка лінії виконується в напрямку, куди спрямована зброя в цей момент, щоб побачити, чи буде щось влучено.
- Підтверджені попадання завдають шкоди діючим акторам, створюють сліди та ефекти удару, а також сповіщають віддалених клієнтів, встановлюючи дані про попадання в змінній HitNotify. Промахи лише створюють сліди та встановлюють HitNotify для віддалених клієнтів, які шукають зміни HitNotify і виконують ту саму трасування, що й локальний клієнт, створюючи сліди та впливи за потреби.

Слайд 18 – Вогонь зі зброї миттєвого влучення

ЗБРОЯ З РЕАЛЬНИМИ СНАРЯДАМИ

- Для цього типу зброї створюється реальний фізичний об'єкт, або снаряд, і направляється в напрямку, куди спрямована зброя. Попадання визначається зіткненням снаряда з іншим об'єктом у світі.
- Для снарядового вогню локальний клієнт виконує трасування з камери, щоб перевірити, який Актор знаходиться під прицілом у FireWeapon(), подібно до реалізації миттєвого попадання. Якщо гравець цілиться у щось, він регулює напрямок вогню, щоб влучити в цю точку, і викликає ServerFireProjectile() на сервері, щоб створити снаряд-актора в напрямку, куди була спрямована зброя.

Слайд 19 – Зброя з реальнпими снарядами

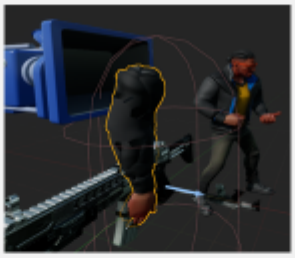
МЕРЕЖЕВИЙ ПЛАГІН

- Компонент був створений на основі IOnlineSubsystem як плагін і дозволяє під'єднувати один до одного користувачів Steam та Epic Games Store, забезпечуючи зручний та універсальний спосіб керування сесіями.
- Надає доступ до інформації про гравця, його друзів.

Слайд 20 – мережевий плагін

ПЕРСОНАЖ

- Інвентар гравця — це масив посилань на AWeapon, що зберігається у властивості Inventory пішака гравця (AMBCharacter). Озброєна на даний момент зброя копіюється з сервера, і, крім того, AMBCharacter зберігає свою поточну зброю локально у властивості CurrentWeapon, що дозволяє знімати попередню зброю, коли споряджається нова зброя.
- Коли гравець одягає зброю на зброї відтворюється анімація.



Слайд 21 – Персонаж

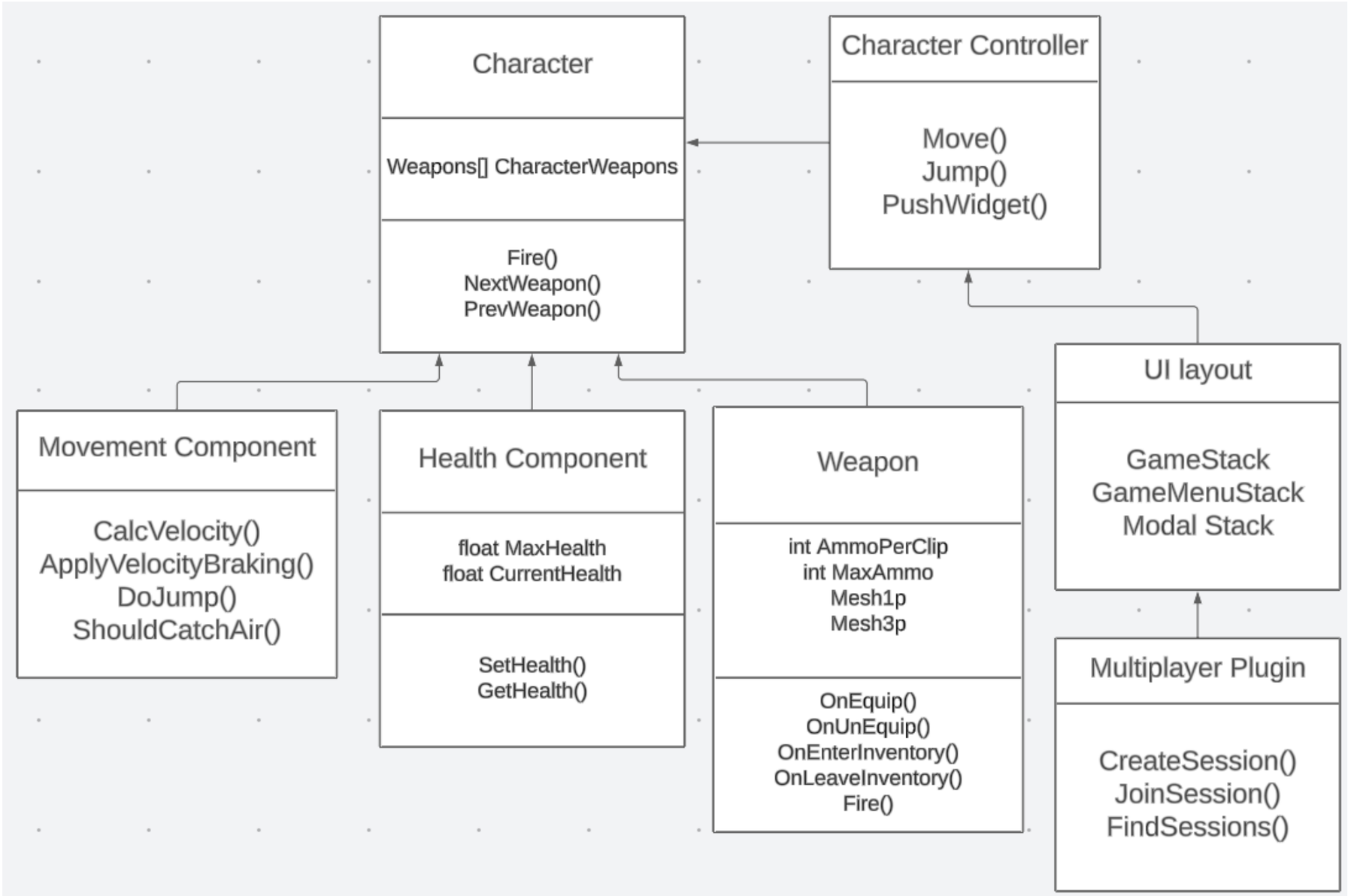
ТЕСТУВАННЯ

- У грі реалізовані тести, які надають інформацію про швидкість підключення до сервера. Ці тести вимірюють час, необхідний для встановлення з'єднання та передачі даних між клієнтом і сервером, а також аналізують стабільність і пропускну здатність з'єднання.
- Крім автоматизованих тестів, проводилось також мануальне тестування, яке включає перевірку функціональності ігрових механік, стабільності роботи при різних умовах мережевого підключення, а також реакції гри на зміну мережевих параметрів.
- Unreal Engine надає можливість встановлювати штучну затримку (ping) для користувача, що дозволяє емулювати умови з високою затримкою.

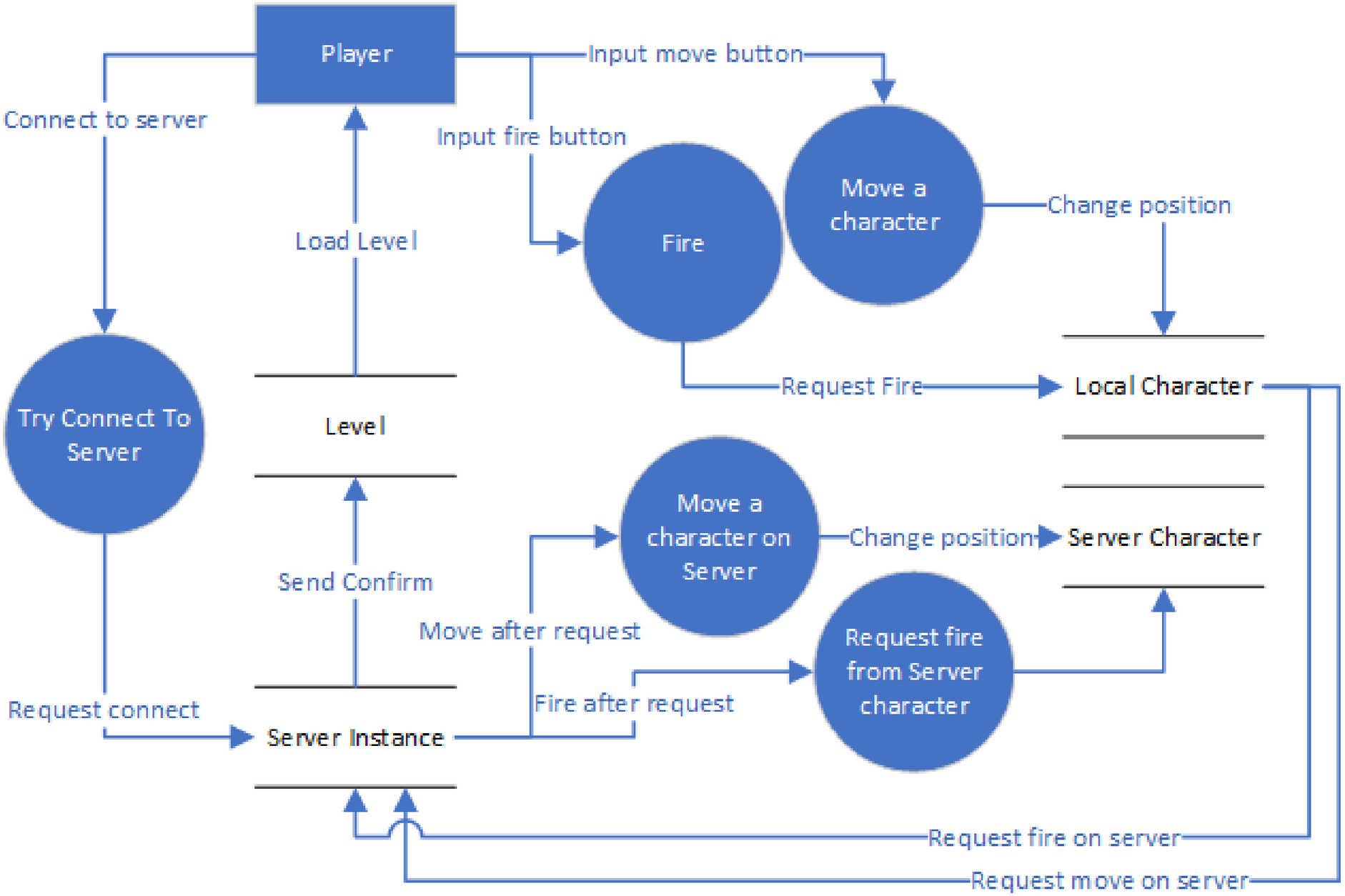
Слайд 22 – Тестування

Слайд 23 – Висновки

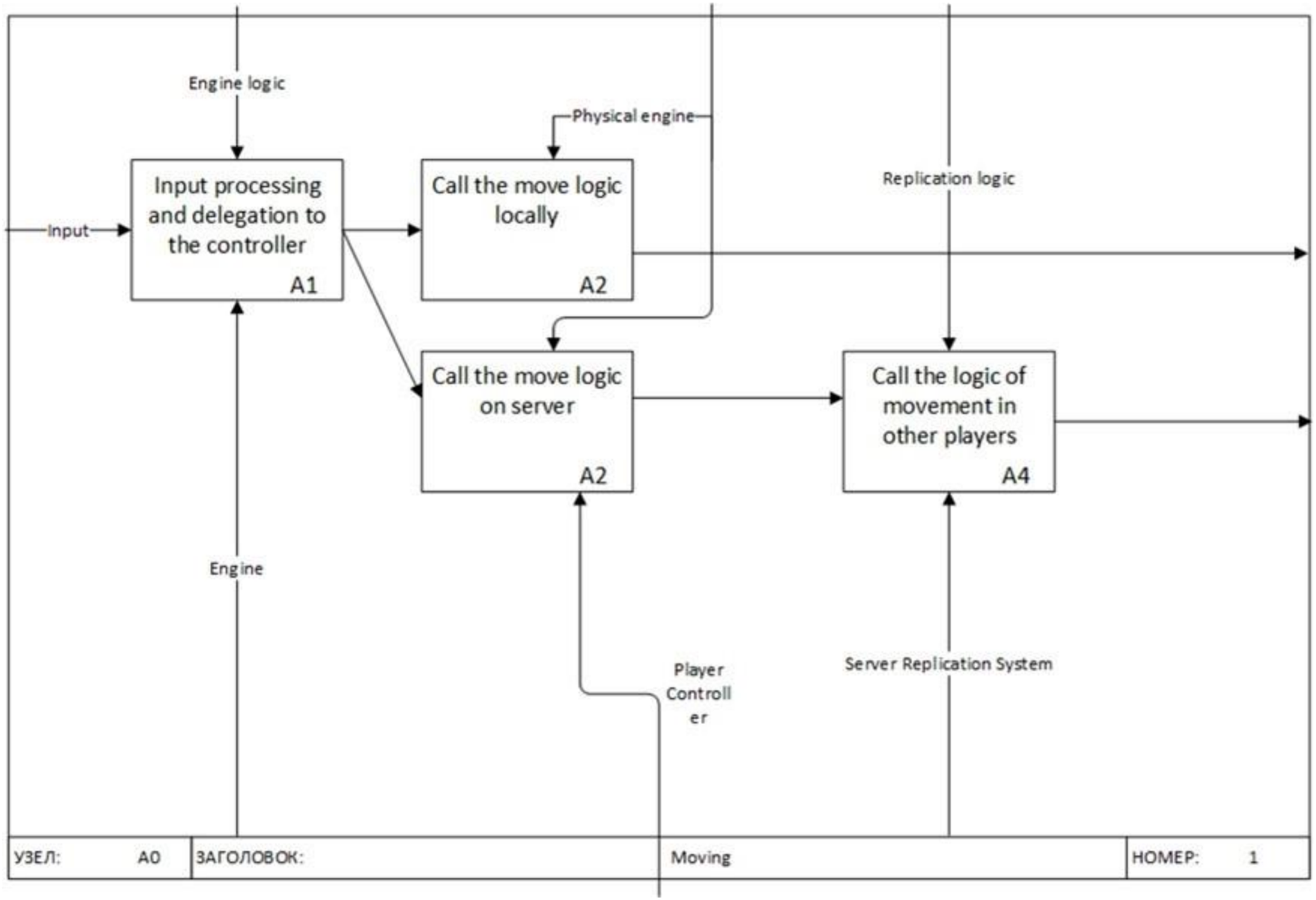
ГРАФІЧНА ЧАСТИНА



					КвРІПЗ.200242.01.01.Е8	
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробив		Балицький Б. І.			Багатокористувацький ігровий застосунок у жанрі «шутер від першої особи» Діаграма класів	Літера / Маса / Масштаб
Керівник		Радельчук Г. І.				
Консульт.						Аркуш 1 / Аркушів 3
Н. Контр.		Радельчук Г. І.				ХНУ, ІПЗ-20-1
Зав. каф.		Бедратюк Л. П.				



					КвРІПЗ.200242.01.01.Е8	
Зм.	*Арк.*	*№ докум.*	*Підпис*	*Дата*	Багатокористувацький ігровий застосунок у жанрі «шутер від першої особи» DFD-діаграма	*Літера* / *Маса* / *Масштаб*
Розробив		*Балицький Б. І.*				
Керівник		*Радельчук Г. І.*				*Аркуш* 2 / *Аркушів* 3
Консульт.						
Н. Контр.		*Радельчук Г. І.*				*ХНУ, ІПЗ-20-1*
Зав. каф.		*Бедратюк Л. П.*				



					КвРІПЗ.200242.01.01.Е8	
Зм.	Арк.	№ докум.	Підпис	Дата	Багатокористувацький ігровий застосунок у жанрі «шутер від першої особи» Діаграма процесів гри	Літера / Маса / Масштаб
Розробив		Балицький Б. І.				
Керівник		Радельчук Г. І.				Аркуш 3 / Аркушів 3
Консульт.						
Н. Контр.		Радельчук Г. І.				ХНУ, ІПЗ-20-1
Зав. каф.		Бедратюк Л. П.				

СУПРОВІДНІ ДОКУМЕНТИ

Завідувачу кафедри інженерії програмного забезпечення проф. Бедратюку Л. П.

здобувача вищої освіти
Балицького Б. І.
Прізвище, ініціали

факультет ІТ, 4 курс, група ІПЗ-20-1

ЗАЯВА

З правилами чинного Положення «Про систему забезпечення академічної доброчесності в Хмельницькому національному університеті», згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлений . Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії Хмельницького національного університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та/або Anti-Plagiarism) і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення текстових збігів у роботах.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

10.06.2024
дата

підпис

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 1.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 11%**

	Документ		Сумарний збіг по Базі Даних	
ID: 129574 Назва: БКР_Багатокористувацький_ігровий_застосунок_у_жанрі_«шутер від першої особи»_Балицький Б. І._Радельчук Г. І. Додано в БД: 2024-06-10 Автора: Балицький Б.І Керівники: Радельчук Г.І , канд. техн. наук, доцент Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	86010	1305	2537 (3%)	33 (3%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

UNICHECK
by Turnitin

Ім'я користувача:
ІПЗ

ID перевірки:
1016346555

Дата перевірки:
11.06.2024 11:02:25 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
11.06.2024 11:56:24 EEST

ID користувача:
100012953

Назва документа: БКР_Багатокористувацький_ігровий_застосунок_у_жанрі_«шутер від першої особи»_Балиць...

Кількість сторінок: 72 Кількість слів: 13954 Кількість символів: 112685 Розмір файлу: 2.03 MB ID файлу: 1016148350

4.61%
Схожість

Найбільша схожість: 1.43% з джерелом з Бібліотеки (ID файлу: 1016121077)

3.59% Джерела з Інтернету 48 ..... Сторінка 74

2.36% Джерела з Бібліотеки 161 ..... Сторінка 76

0.92% Цитат

Цитати 9 ..... Сторінка 77

Не знайдено жодних посилань

0%
Вилучень

Немає вилучених джерел

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ
освітнього ступеня «Бакалавр»

Дипломник Балицький Богдан Ігорович

Тема Багатокористувацький ігровий застосунок у жанрі «шутер від першої особи»

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг кваліфікаційної роботи:

Кількість листів креслень 3; кількість сторінок записки 73

1.Короткий зміст пояснювальної записки та прийнятих рішень У кваліфікаційній роботі було проведено дослідження та проаналізовано предметну область створення багатокористувацьких шутерів від першої особи. Були визначені функціональні та нефункціональні вимоги до гри, які включали у себе механіку геймплею, графічний дизайн та інші важливі аспекти. У процесі аналізу існуючих рішень було розглянуто кілька варіантів серед найкращих представників жанру. Цей аналіз підтвердив актуальність розроблення нового програмного забезпечення. Для реалізації були використані інструменти розроблення, зокрема фреймворк Unreal Engine 5, який забезпечує потужні можливості для створення графіки, фізики, динамічності. Після розроблення застосунку було проведено тестування гри з метою перевірки її функціональності, стабільності та відповідності вимогам. Результати тестування підтвердили, що розроблений застосунок забезпечення працює коректно.

2. Висновок про відповідність роботи поставленому завданню Кваліфікаційна робота виконана відповідно до поставленого завдання та з дотриманням вимог.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки та передових методів роботи У кваліфікаційній роботі була виконана низка завдань, які сприяли створенню програмного продукту – шутера від першої особи, У першому розділі проведено аналіз предметної області та її функціональних особливостей. Розглянуто застосунки-аналоги, а також сформовано вимоги до розроблюваного застосунку. У другому розділі, на основі результатів аналізу та визначених вимог, було створено функціональну архітектуру та структуру застосунку. У третьому розділі описані практичні аспекти розроблення, реалізація модулів застосунку та особливості реалізації цих модулів, які сприяли створенню робочого продукту. Також наведено результати тестування гри, які відповідали за різні її функціональні аспекти. Застосунок створено з використанням актуальних технологій та засобів. Використання Unreal Engine 5 забезпечило ефективну взаємодію гри з користувачем та серверними ресурсами.

4. Позитивні сторони роботи Розроблення гри на платформі Unreal Engine 5 є актуальною та цікавою темою, що відповідає потребам ігрової індустрії, яка швидко розвивається. Тематика роботи має потенціал для подальшого розвитку.

5. Негативні сторони роботи Фокусування на системі руху та мережевому підключенні гравців спричинило зниження якості інших аспектів гри, які залишились на базовому рівні.

6. Оцінка графічного оформлення та пояснювальної записки Графічне оформлення в базовому обсязі описує роботу. Пояснювальна записка оформлена згідно з вимогами стандартів.

7. Відгук про кваліфікаційну роботу в цілому Кваліфікаційна робота заслуговує позитивної оцінки. Матеріал пояснювальної записки структурований, послідовний, що дозволяє в достатній мірі зрозуміти викладений матеріал.

8. Інші зауваження _____

9. Оцінка кваліфікаційної роботи Кваліфікаційна робота виконана в достатньому обсязі, відповідає поставленій задачі та заслуговує на оцінку «задовільно».

РЕЦЕНЗЕНТ (прізвище, ім'я, по-батькові, посада, місце роботи) Мартинюк Валерій Володимирович, доктор технічних наук, професор, зав. кафедри автоматизації, комп'ютерно-інтегрованих технологій та робототехніки ХНУ

"12" 06 2024 р. _____
(підпис)

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатами звіту/звітів перевірки роботи, продукованими програмно-технічним засобом (ами), на наявність текстових збігів:

Назва кваліфікаційної роботи: «Багатокористувацький ігровий застосунок у жанрі «шутер від першої особи»»
Автор: Балицький Богдан Ігорович
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Спеціальність: 121 – Інженерія програмного забезпечення
Науковий керівник: Радельчук Галина Іванівна, кандидат технічних наук, доцент
Після аналізу звіту/звітів зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої й електронної версії роботи	
3	Виявлені запозичення не є академічним плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнуті. Робота може бути допущена до захисту після того, як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Запозичення, виявлені у роботі, є законними і не є плагіатом, оскільки:

1) у тексті кваліфікаційної роботи системою перевірки на плагіат Unicheck виявлено схожість з деякими документами у частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, відомість документів), у структурі змісту, назвах розділів/підрозділів, рамках форм, у назвах та URL-адресах публікацій переліку джерел посилання;

2) в якості запозичень системою Unicheck було зафіксовано деякі послідовності вихідного коду, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;

3) запозичення, виявлені у тексті роботи, є фрагментарними.

Максимальний обсяг запозичень, визначений системою Anti-Plagiarism, складає **1.0**%. Обсяг запозичень, визначений системою Unicheck виявлення збігів ідентичності/схожості, складає **4.61**% і адресується до 439 джерел з Інтернету і 181 джерела з бібліотеки, що, з урахуванням наведених обґрунтувань, відповідає характеру теми і свідчить на користь кваліфікаційної роботи.

Дата 11.06.2024 р.

Завідувач кафедри _____ Леонід БЕДРАТЮК

Гарант освітньої програми _____ Леонід БЕДРАТЮК

Керівник кваліфікаційної роботи _____ Галина РАДЕЛЬЧУК