Хмельницький національний університет

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Шугалюка Андрія Ігоровича

Прізвище, ім'я, по батькові студента(ки)

на здобуття ступеня вищої освіти Бакалавра

Ігровий застосунок у жанрі «Action-RPG»

Назва теми

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗ. 200257.01.24.ПЗ

Виконав студент IV курсу, група ІПЗ-20-1 _____ Андрій ШУГАЛЮК

Підпис Ім'я, ПРІЗВИЩЕ

Керівник канд. техн. наук, доцент _____ Галина РАДЕЛЬЧУК

Науковий ступінь, звання Підпис Ім'я, ПРІЗВИЩЕ

Нормоконтролер доцент _____ Галина РАДЕЛЬЧУК

Посада Підпис Ім'я, ПРІЗВИЩЕ

До захисту допускаю:

Завідувач кафедри інженерії програмного забезпечення _____ Леонід БЕДРАТЮК

Підпис Ім'я, ПРІЗВИЩЕ

11 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

Шугалюку Андрію Ігоровичу
Прізвище, ім'я, по батькові студента

1. Тема роботи Ігровий застосунок у жанрі «Action-RPG»

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) Дослідження предметної області та постановка задач, проєктування програмного забезпечення, програмна реалізація та тестування застосунку.

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Три креслення
1.Діаграма варіантів використання
2. Діаграма зв'язків модулів
3. Діаграма класів

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 Ознайомлення з тематикою дипломного проєктування, визначення та узгодження індивідуальної теми кваліфікаційної роботи (КвР)	01.12– 31.12.2023	
2 Збір матеріалу за темою КвР; дослідження предметної області, в якій планується використання програмного забезпечення (ПЗ), визначення задач та вимог.	01.01 – 20.02.2024	
3 Проєктування програмного забезпечення	21.02 – 20.03 2024	
4 Програмна реалізація з використанням відповідних засобів розроблення. Тестування ПЗ	21.03 – 30.04.2024	
5 Написання вступу, загальних висновків, оформлення переліку джерел посилання та додатків. Оформлення пояснювальної записки КвР згідно вимог	01.05 – 25.05.2024	
6 Попередній захист КвР	Травень	Згідно графіка
7 Перевірка КвР на плагіат, нормоконтроль, отримання відгуків, рецензій та інших супровідних документів. Брошурування (зшиття) пояснювальної записки.	26.05 – 30.05.2024	
8 Здача КвР на кафедру; підготовка КвР для розміщення у репозитарії ХНУ; підготовка до захисту та захист КвР	з 01.06.2024	

Студент _____ Андрій ШУГАЛЮК
Підпис Ім'я, ПРІЗВИЩЕ

Керівник роботи _____ Галина РАДЕЛЬЧУК
Підпис Ім'я, ПРІЗВИЩЕ

АНОТАЦІЯ

Тема кваліфікаційної роботи: Ігровий застосунок у жанрі «Action-RPG».

Автор роботи: Шугалюк Андрій Ігорович

Керівник роботи: Радельчук Галина Іванівна

Пояснювальна записка: 74 с., 61 рис., 2 табл., 4 дод., 30 джерел.

Графічна частина: 3 креслення.

ІГРОВИЙ ЗАСТОСУНОК, ARPG, C++, UNREAL ENGINE.

Метою роботи є розроблення ігрового застосунку в жанрі A-RPG, який буде відповідати актуальним технічним вимогам та методикам ігрової індустрії.

У кваліфікаційній роботі проведено аналіз предметної області та її інформаційного забезпечення, визначені вимоги до ігрових застосунків у загальному та, зокрема, до ігор у жанрі Action-RPG, розроблена детальна архітектура застосунку, спроєктовано структуру модулів та інтерфейс користувача.

Для реалізації ігрового застосунку використано мову програмування C++, візуальну скриптову мову Blueprints та фреймворк і інструментарій Unreal Engine.

У результаті проєктування здійснена програмна реалізація ігрового застосунку, що пропонує захопливий ігровий досвід і може слугувати базовим модулем для команд розробників при створенні якісних продуктів подібного жанру.

10.06.2024
Дата

Підпис

ВІДОМІСТЬ ДОКУМЕНТІВ

№ рядка	Формат	Позначення документа	Найменування документа	К-сть аркушів	№ екз.	Примітка
			Текстові документи			
1	А4	КвРІПЗ.200257.01.24.ПЗ	Пояснювальна записка	74		
2	А4		Завдання на кваліфіка-	1		
			ційну роботу			
3	А4		Анотація	1		
			Графічні документи			
4	А3	КвРІПЗ. 200257.01.24.Е8	Діаграма варіантів	1		
			використання			
5	А3	КвРІПЗ.200257.01.24.Е8	Діаграма зв’язків	1		
			модулів			
6	А3	КвРІПЗ.200257.01.24.Е8	Діаграма класів	1		

Змн.	Арк.	№ докум.	Підпис	Дата	КвРІПЗ.200257.01.24.ВД		
Виконав		Шугалюк А. І.		10.06	Ігровий застосунок у жанрі «Action-RPG» Відомість документів	Літ.	Арк. 1 / Акрушів 1
Керівник.		Радельчук Г. І.		10.06			
Рецензент						ХНУ, ІПЗ-20-1	
Н. Контр.		Радельчук Г.І.		10.06			
Зав. каф.		Бедратюк Л.П.		10.06			

ЗМІСТ

ВСТУП

Створення ігрових застосунків є однією з найбільш динамічних та перспективних галузей сучасної індустрії програмного забезпечення. З кожним роком цей сегмент ринку лише посилюється, зумовлений зростанням популярності відеоігор серед різних груп користувачів, починаючи від дітей та підлітків і закінчуючи дорослими. Тенденції розвитку ігрової індустрії включають постійне вдосконалення технологій, розширення функціоналу, впровадження віртуальної та доповненої реальності, а також зростання популярності онлайн-геймінгу та мобільних платформ. З урахуванням цих тенденцій, розроблення ігрових додатків стає не лише захоплюючим творчим інженерним процесом, але й ключовим напрямком в сфері створення програмного забезпечення, який відкриває безліч можливостей для інновацій та успіху.

Особливо актуальної є створення ігрового застосунку в жанрі A-RPG (action role-playing game) в контексті сучасних технологічних та культурних тенденцій. Жанр A-RPG привертає увагу гравців своєю можливістю взаємодії з великим ігровим світом, сюжетними лініями та елементами імерсії. Останні розвитки в галузі графіки, штучного інтелекту та фізики відкривають нові можливості для створення ще більш захоплюючого ігрового досвіду, де гравці можуть відчути себе часткою фантастичного світу. Такі ігрові додатки стають платформою для творчості, спілкування та взаємодії та вимагаючи від розробників поєднання технічних знань із здатністю створювати захоплюючі сюжети та ігрові механіки. Цей підхід робить A-RPG ігри цікавим та перспективним напрямком в індустрії створення ігор, що не втрачає своєї актуальності уже багато років.

Важливим фактором є зменшення порогу входу в розроблення ігрових застосунків для індивідуальних розробників. Сучасні технології роблять індивідуальне розроблення ігрових застосунків більш доступною та ефективною, ніж коли-небудь. Готові ігрові рушії, такі як Unity, Unreal Engine та Godot, надають широкі можливості для створення ігор будь-якого жанру, включаючи A-RPG.

Онлайн-ресурси та спільноти забезпечують необхідну підтримку та поради від інших розробників, а готові ресурси, такі як графічні та звукові активи, спрощують процес створення ігрового контенту.

Саме тому розроблення ігрового застосунку в жанрі Action-RPG є актуальною, адже якісне проєктування і реалізація подібного проєкту може поглибити мої знання з програмування, графічного дизайну, UI дизайну та інших технічних аспектів створення програмного забезпечення та в наслідку свідчити про високу кваліфікацію розробника, що сильно підвищує шанси працевлаштування у світові компанії, а також може бути відправним пунктом для створення особистого високоякісного комерційного ігрового проєкту, який зможе надавати користувачам цікавий і насичений ігровий досвід, як з технічної так і з візуальної точки зору.

Метою кваліфікаційної роботи є розроблення ігрового застосунку в жанрі «Action-RPG», який буде відповідати актуальним технічним вимогам та методикам ігрової індустрії.

Для успішного досягнення мети роботи необхідно розв'язати наступні завдання проєктування:

− провести аналіз предметної області розроблення ігрових застосунків та ігор жанру A-RPG зокрема;

− розглянути існуючі конкурентні продукти та проаналізувати прийняті розробниками рішення;

− сформувати вимоги до технічного шару, ігрових механік та візуального оформлення ігрового застосунку;

− визначитись з архітектурним підходом та потрібними патернами;

− провести детальне проєктування модулів та їх інтерфейсів;

− проаналізувати та обрати технології і методи реалізації застосунку;

− реалізувати програмні модулі;

− виконати візуальну частину проєкту;

− провести тестування застосунку;

− проаналізувати результат роботи.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Змістовий аналіз предметної області

Ігрова індустрія є однією з найбільш динамічних і швидко зростаючих галузей розваг у сучасному світі. За останні десятиліття вона перетворилася на справжній промисловий гігант, який генерує мільярди доларів доходів щорічно та привертає увагу мільйонів гравців з усього світу [1]. Ігрова індустрія не лише змінила спосіб, яким ми відпочиваємо та розважаємося, а й стала важливим культурним феноменом, що впливає на суспільство, економіку та технологічний прогрес у загальному.

Ігри стали не лише засобом розваги, але й формою мистецтва, важливим засобом комунікації та способом розвитку креативності та критичного мислення. Разом із технологічними досягненнями в галузі комп'ютерної графіки, штучного інтелекту, віртуальної реальності та інших технологій, ігри стали набагато реалістичнішими, інтенсивнішими та захоплюючішими. Багато великих компаній, таких як Electronic Arts, Ubisoft, Activision Blizzard, і Sony Interactive Entertainment, CD Project Red (рисунок 1.1) активно діють на ринку ігрової індустрії, випускаючи десятки ігор і постійно ведучи бій за увагу гравців.

Рисунок 1.1 – Найбільші ігрові студії сьогодення

Ігрова індустрія є неймовірно широкою, а створення ігрових застосунків включає багато нюансів і особливостей, які потрібно врахувати на початку роботи над проєктом. В першу чергу потрібно проаналізувати реалізацію ігрового застосунку, як окрему предметну під область проєкту.

Проєктування та реалізація ігрового застосунку є складним та багатоаспектним процесом, що вимагає ретельного підходу та розгляду різних аспектів [12]. Починається цей процес з визначення базової концепції, розробники визначають жанр, аудиторію і механіки геймплею, атмосферу та сюжетну лінію. Важливим є визначитись в якому візуальному стилі буде виконуватись проєкт, щоб розуміти, які технічні вимоги до ігрового рушія будуть стояти. Встановлення чітких цілей та завдань визначає напрямок реалізації ігрового застосунку.

Наступним етапом розроблення є створення ігрового рушія та необхідних інструментів або аналізують, які готові технологічні рішення можна вибрати для свого проєкту, які дозволять реалізувати задуману концепцію гри і повному обсязі. Це включає в себе роботу з програмним забезпеченням для графіки, анімації, фізики, штучного інтелекту та інші аспекти технічної реалізації гри [15]. Ігрові движки, як і інші програмні системи, зазвичай мають багаторівневу структуру. Зазвичай вищі рівні залежать від нижніх, але не навпаки. Залежність нижнього рівня від верхнього створює циклічну залежність, яку варто уникати в будь-якій програмній системі. Такі залежності створюють небажані зв'язки між компонентами, що робить програмне забезпечення нестабільним і утруднює повторне використання коду. Це особливо важливо для великомасштабних систем, таких як ігрові движки.

Для реалізації якісного ігрового застосунку від рушій повинен реалізувати такі функціональні шари і модулі:

- шар взаємодії з операційною системою;
- рівень незалежності від платформи (потоки, типи даних, таймери, файлова система, графічний API (application programming interface));
- основні системи (ініціалізація, виділення пам’яті, математичні бібліотеки,

конфігурування, завантаження файлів, профілювання);

− управління ресурсами (3д-моделями, текстурами, скелетами, рівнями, звуками, ефектами);

− модуль загального рендерингу;

− модуль фізики і зіткнення об’єктів;

− модуль взаємодії з HID (human interface device);

− модуль відтворення скелетної анімації;

− система відтворення візуальних ефектів;

− система загального геймплею;

− шар специфічних систем геймплею (Gameplay Framework).

На основі системи ігрового рушія створюються ігрові механіки, які вважаються доцільними та встановлюється візуальний вигляд проєкту.

Після створення базового контенту розробники переходять до етапу тестування та відлагодження. Гра проходить технічне тестування, балансування геймплею та збір фідбеку від тестерів для виявлення та виправлення помилок та недоліків. Наступним кроком є оптимізація гри для різних платформ та випуск. Розробники працюють над покращенням продуктивності та ефективності гри на різних пристроях, після чого гра готова до випуску через цифрові магазини.

Весь цей процес допомагає розробникам створити якісний та цікавий ігровий застосунок, який задовольнить потреби та очікування гравців, а також забезпечить успіх на ринку відеоігор.

Після випуску гри розробники продовжують підтримувати та оновлювати її. Це включає в себе випуск нового контенту, виправлення помилок та вдосконалення геймплею на основі отриманих відгуків від гравців.

Після визначенням проблем, які потрібно буде вирішувати при реалізації ігрового застосунку в загальному, потрібно проаналізувати головну властивість ігрових застосунків – жанр, як окрему і основну предметну область проєкту.

Як тема роботи було обрано ігровий застосунок у жанрі Actio-RPG. В свою чергу жанр Action-RPG (Action Role-Playing Game) є одним із найпопулярніших і

захоплюючих в світі відеоігор [14]. Він поєднує в собі найкращі аспекти екшену та рольової гри, створюючи унікальний геймплей, який пропонує гравцям яскраві враження та безліч емоцій. Основна особливість Action-RPG полягає в тому, що гравець активно бере участь у боях, виконує складні маневри та використовує різноманітні уміння, в той час як він також розвиває свого персонажа та взаємодіє з ігровим світом.

У цьому жанрі великий акцент приділяється динамічному бою в реальному часі, який вимагає від гравця швидкісної реакції та стратегічного мислення. Бійці можуть використовувати різноманітну зброю, магію або спеціальні атаки для перемоги в боях з різноманітними ворогами. При цьому, розвиток персонажа відбувається через отримання досвіду, підвищення рівня, розвиток навичок та вдосконалення екіпіровки, що дозволяє гравцю створити унікального індивідуального персонажа, адаптованого до його власного стилю гри.

Звичайно, Action-RPG також відомі своїми глибокими і захоплюючими сюжетами та квестами [9]. Гравці можуть поглибитися у великі та живописні ігрові світи, де кожне рішення та дія може мати великий вплив на подальший хід подій. Часто ці ігри пропонують різноманітні альтернативні закінчення та варіанти розвитку подій, що стимулює гравців до більш глибокого вивчення ігрового світу та його персонажів.

У світі відеоігор, Action-RPG відомі своїми незабутніми пригодами, захоплюючими боями та глибокими персонажами, що робить їх одними з найпопулярніших та найбільш улюбленими серед гравців усього світу.

Перед тим, як визначитись хто кінцевий користувач і які в нього вимоги важливо, визначитись з сетингом, адже це впливає на визначення аудиторії.

Сетинг гри – це визначальний фон та контекст, в якому відбуваються події гри. Він включає в себе місце дії, час, атмосферу, соціокультурний та історичний контекст, а також всі інші аспекти оточення, що створюють образний світ гри.

Сетинг гри може впливати на багато аспектів геймплейних механік та враження від гри – чи то це містичний світ фентезі зі своїми загадковими

створіннями та чарівною аурою, чи то темний і містичний світ готики, сповнений таємниць та тривожності. Ця атмосфера впливає на емоційний стан гравців та їхні враження від гри. Вигляд та характеристики персонажів гри також залежать від сетингу ігри, наприклад, у фентезі можуть бути присутні різноманітні магічні істоти та персонажі, тоді як у готичних іграх переважають темні, містичні та ексцентричні персонажі.

Сетинг може впливати на геймплей гри, визначаючи доступні механіки та можливості для гравців. Наприклад, у фентезі гравці можуть володіти магічними здібностями або використовувати фантастичні зброї, тоді як у готичних іграх можуть бути акценти на розв'язання загадок або використання додаткових можливостей персонажів.

Звернувшись до статистики [3], які сетинги найчастіше використовуються (рисунок 1.2), у проєкті було вирішено обрати сетинг темного фентезі.

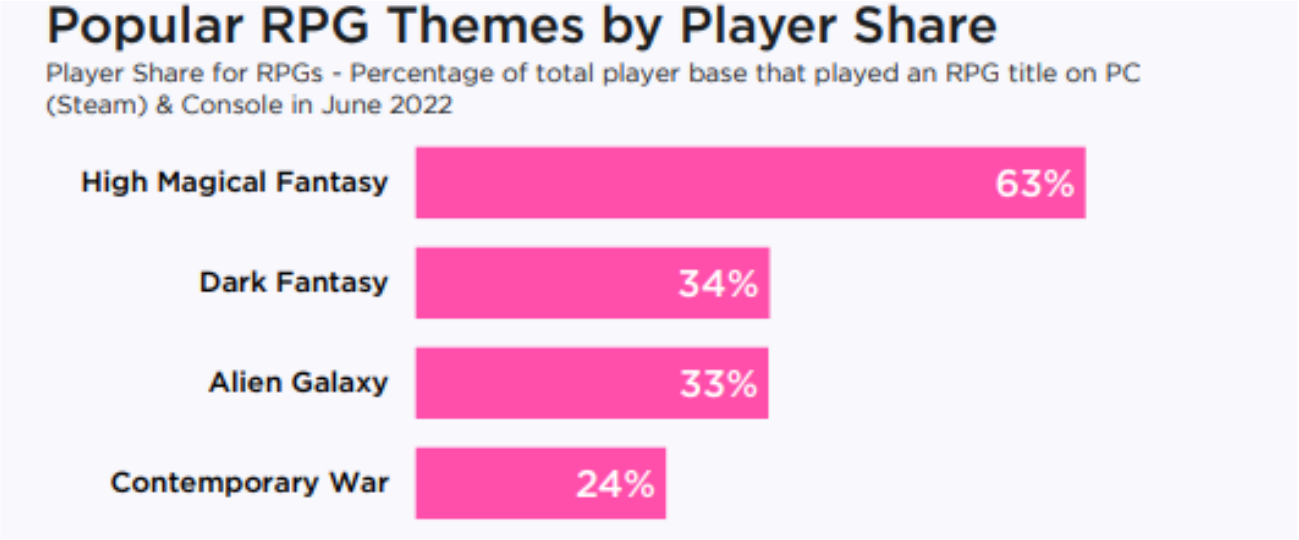


Рисунок 1.2 – Статистика найпопулярніших сетингів для RPG

Сетинг «дарк фентезі» створює містичну, темну та тривожну атмосферу, що може зробити гру більш напруженою та емоційно насиченою. Гравці можуть відчувати неймовірну атмосферу загрози, таємниць та небезпеки.

Провівши аналіз предметної області можна визначити, яких кінцевих

користувачів має зацікавити фінальний продукт і що вони очікуватимуть від проєкту в першу чергу.

Звернувшись до статистичних даних [2], які показують типову аудиторію Action-RPG ігор, можна так зазначити характеристики користувачів (рисунок 1.3):

– переважно це чоловіки, 58% проти 41% жінок;

– вік зазвичай в діапазоні 10-30, це 59% користувачів;

– платформа користувачів переважно ПК 50% але і консолі складають значну частку 35%;

– важка складність є важливим фактором для 50% користувачів і ще 30% вважають що вона повинна бути помірна;

– для половини гравців важлива сюжетна складова ігри.

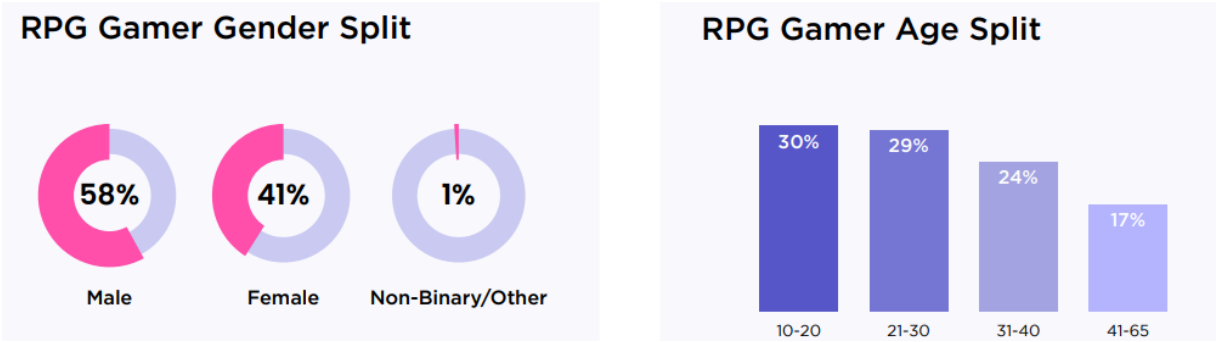


Рисунок 1.3 – Розподіл гравців в RPG за віком і статтю

Проаналізувавши предметну область можна визначити, що зазвичай кінцеві користувачі очікують від ігри в жанрі Action-RPG в сетингові темного фентезі.

Головні потреби, які повинен задовольняти продукт:

– бойова система повинна бути розвинутою, вимагати від гравця майстерності, стратегії та тактичного мислення, а також пропонувати різноманіття бойових стилів та можливостей;

– вороги повинні бути небезпечними, унікальними за дизайном та поведінкою, а також кидати виклик гравцям різними тактиками та здібностями;

– гравці повинні відчувати, що їхні дії мають значення, а персонаж стає сильнішим, досвідченішим, отримує нові можливості та знання;

− візуально світ гри повинен бути темним, похмурим, візуально вражаючим, з унікальним дизайном локацій, детально прописаними NPC та відчуттям небезпеки і загадковості;

− технічний стан ігри має бути високим, забезпечивши динамічний ігровий досвід без переривань і проблем з продуктивністю поєднуючи близький до реалістичного графічний стиль.

В результаті проведеного аналізу можна підсумувати: реалізація ігрового рушія повинна забезпечити модулі для створення приємного та стабільного ігрового досвіду, ігрові механіки обов'язково повинні включати динамічну бойову систему, систему прокачки ігрового персонажу, яка включає унікальні здібності, розвинуту логіку поведінки ворогів та захоплюючу нормативну складову. В художньому плані при створені ігрових об'єктів повинен бути дотриманий темний фантастичний стиль.

1.2 Аналіз наявного програмно-технічного забезпечення предметної області

Для якісного аналізу існуючого ігрового забезпечення у жанрі Action-RPG, необхідно врахувати специфічні особливості цього жанру ігор. Action-RPG відомий своєю динамікою, швидкими боями, глибокою системою розвитку персонажів та часто відкритими світами для дослідження. При аналізі наявного програмного забезпечення для Action-RPG, в першу чергу потрібно зосередитись на оцінці його геймплею, механік бою, системи прокачки, а також візуальної та звукової складових. Потрібно оцінити, наскільки добре гра передає динаміку бою, чи належним чином збалансована система прогресування персонажів, і які можливості щодо налаштування світу та геймплею пропонує гравцям. Аналізуючи існуючі програмні рішення для Action-RPG, можна простежити за останніми тенденціями в галузі та спробувати ідентифікувати ключові пункти для подальшого формування вимог до ігрового досвіду для мого застосунку.

До розгляду було взято ігри із списку хітів та найкраще оцінених в живому рейтингу на платформі Steam [5-6].

Першою розглянемо God of War (2018) [6]. God of War (рисунок 1.4) – це Action-RPG, яка переносить гравців у світ скандинавської міфології, де вони беруть на себе роль Кратоса, бога війни, і його сина Атрея. Гра відзначається динамічним та захоплюючим геймплеєм, який поєднує в собі бойові механіки з елементами рольової гри з цікавим сюжетом. Гра вражає своєю красою та деталізацією. Світ гри ретельно пророблений, а персонажі та вороги мають вражаючий дизайн. Застосована камера з віддачею від третьої особи додає іммерсії і підкреслює епічність бойових сцен та велич ігрового світу.

Рисунок 1.4 – Знімок екрану з грою God of War

Однією з ключових механік гри є бойова система (рисунок 1.5), яка включає в себе різноманітні комбінації атак, захист та унікальні способи бою, що дозволяють гравцеві відчути себе могутнім богом. Крім того, гра має глибоку систему прокачки персонажа (рисунок 1.6), що дозволяє гравцям покращувати навички Кратоса та вдосконалювати його зброю та здібності. Нормативна складова надає користувачу активно взаємодіяти з NPC (Non-player character), а сюжет супроводжується постійними високоякісними ігровими сценами.

Рисунок 1.5 – Знімок екрану з ігровим боєм у God of War

Рисунок 1.6 – Знімок екрану з деревом покращень в God of War

Далі проаналізуємо NieR: Automata [7]. NieR: Automata (рисунок 1.7) – це Action-RPG, розроблена студією PlatinumGames і видана Square Enix. Гра була випущена в 2017 році для консолей PlayStation 4, Xbox One і персональних комп'ютерів. Вона відзначається своєю унікальною атмосферою, захоплюючим геймплеєм та глибоким сюжетом. "NieR: Automata" пропонує глибокий та емоційно заряджений сюжет, який досліджує теми штучного інтелекту та людської природи.

Рисунок 1.7 – Знімок екрану з грою NieR: Automata

Гравці відправляються в містичний світ, окутаний таємницями та небезпеками, що створює унікальну атмосферу та приваблює до глибокого осмислення. Один з найсильніших аспектів гри – це її вражаючий саундтрек, який підкреслює емоційну силу історії та створює неповторну атмосферу. Звукові ефекти та голосове акторське гра також на високому рівні, поглиблюючи імерсію гравця в ігровий світ.

Однією з ключових механік гри є бойова система (рисунок 1.8), яка поєднує в собі швидкі й динамічні бої з використанням різноманітної зброї та унікальних здібностей персонажів. Гравці можуть контролювати різних персонажів і використовувати різноманітні комбінації атак для перемоги над ворогами, система покращень дає змогу «на ходу» змінювати здібності на будь-які добуті раніше, та комбінування різні види зброї. Нормативний дизайн надає користувачу взаємодіяти з великою кількістю побічних персонажів, виконувати їх завдання та знаходити секретні документи, які допомагають краще розкрити сюжет, іноді звичайні діалоги змінюються текстовою новело занурюючи в музику і емоції.

Рисунок 1.8 – Знімок екрану з ігровим боєм у NieR: Automata

Ще одним цікавим екземпляром для розгляду є Elden Ring [8]. ELDEN RING (рисунок 1.9) – це Action-RPG, розроблена спільно компанією FromSoftware та письменником Джорджем Р. Р. Мартіном. Гра вийшла у 2022 році для консолей PlayStation 4, PlayStation 5, Xbox One, Xbox Series X/S та персональних комп'ютерів. Вона відзначається своїм великим відкритим світом, унікальною атмосферою та глибокою системою геймплею.

Рисунок 1.9 – Знімок екрану з грою ELDEN RING

Бойова система в грі ELDEN RING є однією з ключових складових геймплею і відображає основні принципи, характерні для ігор від FromSoftware. Вона поєднує в собі динамічність, стратегічність та глибину, що вимагає від гравця уваги до деталей та швидкого реагування.

У ELDEN RING гравці мають доступ до різноманітної зброї, включаючи мечі, сокири, луки та магічні заклинання. Кожен вид зброї має свої унікальні характеристики та стиль бою, що дозволяє гравцям вибирати оптимальну стратегію для подолання ворогів. Битви в ELDEN RING (рисунок 1.10) відзначаються швидкістю та напруженістю, що вимагає від гравця швидкого реагування на зміни в ситуації та ефективного використання доступних інструментів та здібностей [10].

Рисунок 1.10 – Знімок екрану ігровим боєм у ELDEN RING

Кожен ворог та бос в грі має свої власні унікальні характеристики та атаки, що вимагає від гравця адаптації та створення стратегій бою для кожного конкретного ворога. Деякі боси можуть мати особливі слабкі місця або фази бою, які гравці повинні виявити та використати.

Історія світу і його жителів подається через необов'язкові місії, які проходять через всю основну історію та через опис предметів, які гравець може знайти у світі.

Підсумовуючи розгляд ігор «God of War», «NieR: Automata» та «ELDEN RING», можна виділити кілька спільних характеристик, які є типовими для багатьох сучасних Action-RPG:

− всі ці ігри відзначаються складними та глибокими бойовими системами, що поєднують у собі динамічність, стратегічність та різноманіття у виконанні атак та захисних дій;

− кожна гра із розглянутих має систему прокачки, що дозволяє гравцям вдосконалювати навички, збільшувати статистику та отримувати нові здібності та обладунки для персонажів;

− більшість ігор мають великі відкриті світи, які гравці можуть вільно досліджувати, зустрічаючи різноманітні локації, персонажів та завдання;

− всі ці ігри відзначаються наявністю глибоких сюжетів та атмосферою, яка дозволяє гравцям відчути емоційну зв'язаність з персонажами та подіями в грі;

− кожна з ігор має свої унікальні механіки та атмосферу, що робить їх унікальними та цікавими для гравців.

Отже можна зробити висновок, що ці визначені спільні характеристики роблять ігри особливими представниками жанру Action-RPG та забезпечують їм успіх серед широкої аудиторії гравців.

1.3 Визначення вимог до ігрового застосунку

Провівши аналіз предметної області і існуючі рішення можна прийти до висновку, що потрібно розробити свій ігровий застосунок добавивши нові ідеї в ігрові механіки поєднавши їх з уже існуючими, перевіреними механіками жанру.

Розроблятись застосунок буде для платформи Windows 10. Було вирішено вибрати головною особливістю проєкту зосередження бойової системи і системи прокачки на магічних здібностях не пов’язаних холодною зброєю.

Функціональні вимоги базуються на характеристиці предметної області і

розгляді існуючих рішень та складаються з таких пунктів [8]:

− ігровий світ буде представлений у вигляді простої 3Д симуляції;

− ігровий світ буде складатись з лінійно поєднаних локацій;

− ігровий світ буде наповнений ворогами з якими потрібно буде боротись і нормативними елементами;

− ігрова камера буде від третього лиця;

− ігровий персонаж матиме змогу ходити по рівням у будь-якому напрямку;

− ігровий персонаж зможе підстрибнути і зробити короткий ривок;

− ігровий персонаж зможе атакувати ворогів використовуючи базову атаку героя і атакуючі здібності;

− ігровий персонаж зможе взаємодіяти з нормативними елементами;

− для боротьби з ворогами ігровий персонаж зможе використовувати, бойову магію, захисну магію і лікувальну магію;

− для відкриття нової магії і покращення характеристик гравцю потрібно буде піднімати свій рівень за рахунок знищення ворогів;

− після підняття рівню гравцю будуть надаватись нової здібності або покращуватись уже існуючі;

− ігровий інтерфейс буде надавати інформацію про стан персонажу, його рівень і вибрані здібності та надавати можливість зберегтись і налаштувати гру;

− інтерактивні суб’єкти представлені у вигляді об’єктів на рівні, підійшовши до яких, гравець зможе запустити взаємодію з ним;

− інтерактивні NPC при взаємодії ведуть текстовий діалог з гравцем;

− штучний інтелект для ворогів повинен забезпечувати автоматизовану поведінку, уміти атакувати, захищатись та шукати гравця.

Окремо потрібно виділити функціональні вимоги до ігрового рушія:

− надається можливість зчитування і опрацювання введеня з клавіатури, миші і геймпаду;

− забезпечується рендер ігрових сцен та зміна налаштувань якості;

− імплементований менеджмент ресурсів проєкту;

− доступна перевірка зіткнень і фізична симуляція;

− наявна система збереження і завантаження станів та налаштувань гри;

− реалізована система програмування і відтворення скелетних анімацій;

− впроваджена система рендеру ігрового інтерфейсу;

− реалізований система подій і сповіщень;

− забезпечується можливість відтворення звуків;

− надається система відтворення візуальних ефектів;

− розроблено базовий ігровий фреймворк.

Нефункціональні вимоги повинні забезпечити приємний досвід взаємодії з застосунком, який не буде заважати отримувати задоволення від ігрових механік і сюжету, тому вони будуть складатись з таких пунктів:

− гра повинна працювати без серйозних затримок на цільовій платформі;

− гра повинна мати інтуїтивно зрозумілі та зручні елементи керування;

− повинна бути забезпечена висока якість звукового супроводу, включаючи музику, звуки оточення та голосові ефекти та реалізована можливість налаштування рівня гучності.

На основі аналізу ми можем визначити, що головним і єдиним актором при взаємодії с застосунком буде сам гравець «Player», головною ціллю використання ним системи є процес ігри. Процес ігри складається з таких варіантів використання доступних користувачу:

a) дослідження:

1) переміщення по світу;

2) інтерактив зі світом:

− інтеракція з NPC;

− інтеракція з предметами;

b) бій:

1) переміщення;

2) атака, яка складається із:

− звичайної атаки;

– атаки здібністю;

с) розвиток:

1) отримання досвіду;

2) прокачки здібностей.

В результаті аналізу була розроблена UML-діаграма [11] варіантів використання, яка зображена на рисунку 1.11.

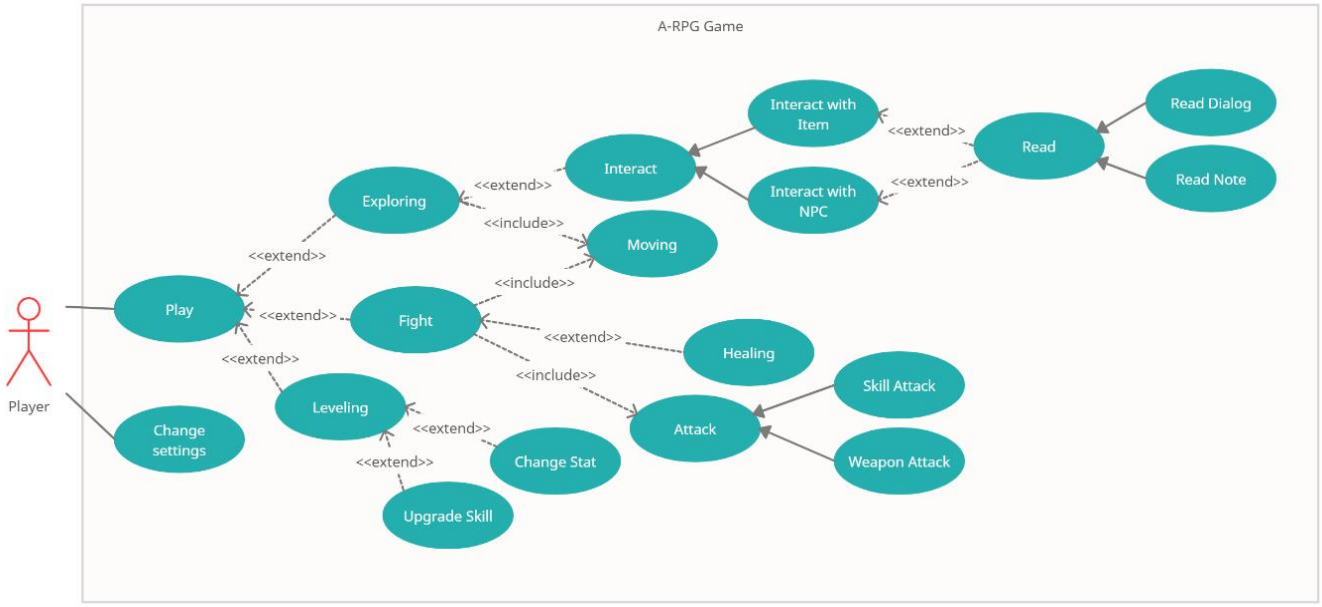


Рисунок 1.11 – UML-діаграма варіантів використання

1.4 Висновки. Постановка задачі

У цьому розділі було розглянуто предметну область створення ігрових застосунків в загальному і зокрема ігор жанру A-RPG. Було проаналізовано основні процеси і етапи створення ігор, яку задачу вони вирішують і яка мета в створені ще однієї ігри. Для визначення вимог до застосунку було проаналізовано жанрові особливості застосунків цього жанру, розглянуто популярні, для ігор цього жанру, сетинги та обрано сетинг для художньої частини мого проєкту. Також, для повного розуміння, було досліджено статистичні дані, які описують аудиторію ігор цього

жанру і визначають, що саме цінять користувачі ігрових застосунків жанру A-RPG. З метою виокремити виграшні і цікаві механіки, які пропонують існуючі застосунки, було розглянуто найпопулярніших представників на платформі Steam.

Повний аналіз предметної області дав змогу сформувати функціональні та нефункціональні вимоги до ігрового застосунку, зокрема до ігрового рушія та ігрового процесу. Ці дані можна підсумувати у вигляді технічного завдання, яке остаточно затвердить вимоги і рамки проєкту, що покращить подальші етапи роботи над застосунком.

Отримавши вичерпну функціональну і технічну специфікацію під час роботи над розділом, можна поставити задачі, які мають бути реалізовуватись в подальшому під час виконання кваліфікаційної роботи.

Перелік поставлених задач:

- провести аналіз існуючих архітектурних підходів і патернів проєктування, які зазвичай використовуються, та на основі цього визначитись з тими, які будуть використані для мого застосунку;
- описати декомпозицію проєкту, зв’язки між варіантами декомпозиції та інтерфейси модулів;
- розробити макетне представлення інтерфейсу користувача;
- провести аналіз та вибрати технології для реалізації спроєктованої архітектури, які будуть найефективнішим варіантом для реалізації;
- розробити основні модулі ігрових механік;
- реалізувати логіку анімаційних сценаріїв;
- створити базові візуальні ресурси;
- знайти та імплементувати потрібні візуальні та звукові ефекти;
- створити та запрограмувати інтерфейс користувача;
- визначитись з методами тестування та реалізувати їх;
- провести тестування застосунку та проаналізувати результати;
- зробити висновки на основі результатів кваліфікаційної роботи.

2 ПРОЄКТУВАННЯ ІГРОВОГО ЗАСТОСУНКУ

2.1 Вибір типу архітектури та шаблонів проєктування

В першу чергу потрібно розглянути, які типи архітектур використовуються для ядра ігрових застосунків, їх ігрових рушіїв, адже зазвичай вони розробляються, як окрема система. Архітектура ігрових рушіїв може значно варіюватися в залежності від цілей [16], для яких вони призначені, та специфічних вимог розробників. Нижче наведено опис декількох поширених архітектур, які використовуються в сучасних ігрових рушіях.

Монолітна архітектура це підхід, при якому всі компоненти ігрового рушія інтегровані в одне велике, тісно пов'язане ціле. Це означає, що рендеринг, фізика, обробка введення, управління ігровою логікою та інші системи розроблені як частини одного великого коду. Така архітектура часто використовується в старих рушіях або в простих іграх. Основною перевагою є простота в реалізації та тестуванні, оскільки всі частини системи можуть безпосередньо взаємодіяти одна з одною. Недоліком є те, що така архітектура погано масштабується і важко підтримується при збільшенні розміру проєкту, а також ускладнює паралельну реалізацію різних компонентів.

Модульна архітектура є найпопулярнішою архітектурою для рушіїв загального призначення. Це підхід розбиває рушій на окремі, незалежні модулі, кожен з яких відповідає за певну функціональність. Наприклад, може бути окремий модуль для рендерингу, фізики, обробки аудіо, управління введенням та інші. Кожен модуль може розроблятися, тестуватися та вдосконалюватися незалежно від інших модулів. Зазвичай модулі додатково об’єднуються в шари, які відповідають за різні рівні функціоналу. Цей підхід значно полегшує підтримку та розширення рушія. Unreal Engine 4 і 5 є прикладами рушіїв з модульною архітектурою. Приклад такої архітектури зображено на рисунку А.1 у додатку А. Модульність дозволяє швидко інтегрувати нові технології або оновлювати окремі частини рушія без необхідності переглядати весь код.

Архітектура ECS (Entity Component System) розділяє дані та поведінку на три основні частини: сутності (entities), компоненти (components) та системи (systems). Сутності є унікальними ідентифікаторами, які поєднують різні компоненти. Компоненти містять дані, а системи визначають логіку та обробляють компоненти. Цей підхід дозволяє легко додавати або видаляти поведінку, просто додаючи або видаляючи компоненти з сутностей. ECS забезпечує високу гнучкість та ефективність, особливо при роботі з великою кількістю об'єктів, які мають різноманітні поведінки. Unity, один з популярних ігрових рушіїв, поступово інтегрує ECS в свою архітектуру.

Аналізуючи розглянуті архітектурні підходи, для мого проєкту, було вирішено, що модульна архітектура надає безліч переваг, які сприяють створенню гнучких, масштабованих та легко підтримуваних ігрових проєктів. Вона дозволяє ефективно розподілити ресурси, підвищує продуктивність роботи та забезпечує високу якість кінцевого продукту. Завдяки цим перевагам, модульна архітектура є оптимальним вибором для розробників сьогодні.

Після вибору бажаної архітектури ігрового рушія, потрібно визначитись з архітектурними підходами, які будуть використовуватись для проєктування і реалізації ігрового модулю, який включатиме саму ігрову логіку.

Існує кілька типів архітектур, які часто використовуються в розроблені ігор. Вибір архітектури залежить від складності гри, платформи, для якої вона розробляється, та вподобань колективу розробників. Для кращого розуміння далі розглянуто кілька популярних архітектурних методів [9], що використовуються під час проєктування ігор.

Монолітна архітектура – ця архітектура передбачає наявність одного виконуваного або бінарного файлу, який містить весь логіку гри, відображення, звук та інші компоненти. Вона підходить для невеликих ігор або прототипів, де простота важливіша за масштабованість.

Шарова архітектура (рисунок 2.1) це архітектура [22] у якій код гри організований у шари, кожен з яких відповідає за певний аспект гри. Наприклад,

можуть бути шари для обробки введення, керування ігровими об’єктами, керування анімацією. Цей підхід поліпшує модульність і спрощує підтримку та тестування.

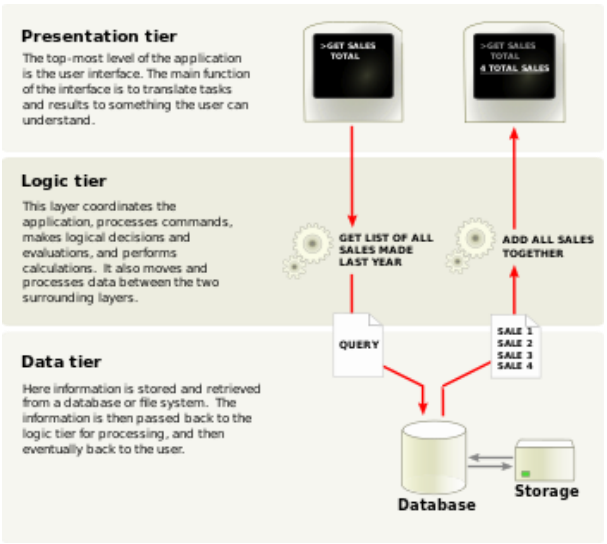


Рисунок 2.1 – Приклад шарової архітектури

Entity-Component-System [23] (рисунок 2.2) – це підхід, зорієнтований на дані, який розділяє сутності гри на окремі компоненти та системи, що обробляють ці компоненти. сутності представлені наборами компонентів, а системи працюють з певними типами компонентів. ECS відома своєю ефективністю та масштабованістю, тому її широко використовують у багатьох ігрових рушіях коли вимагається обробка великої кількості ігрових об’єктів одночасно.

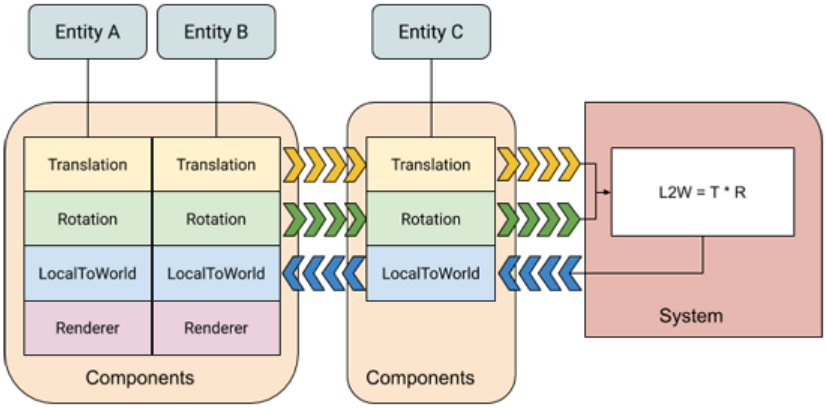


Рисунок 2.2 – Приклад ECS архітектури

Клієнт-серверна архітектура використовується в багатокористувацьких іграх, де логіка гри розподіляється між клієнтом (пристроєм гравця) та сервером. Клієнт обробляє введення користувача та відображення, тоді як сервер керує синхронізацією стану гри, обчисленнями фізики та іншими необхідними завданнями. Ця архітектура дозволяє реалізувати онлайн-функціональність багатокористувацьких ігор.

Data-Driven підхід означає, що вміст та поведінка гри зберігаються в файловій формі, а не прописуються в коді. Цей підхід дозволяє дизайнерам та художникам легко вносити зміни без модифікації базового коду. Він часто використовується для зручної роботи в редакторі.

Компонентна архітектурний (рисунок 2.3) підхід, що передбачає розділення функціональності гри на компоненти, що можуть бути прикріплені до об'єктів гри. Кожен компонент відповідає за конкретний аспект поведінки або властивостей об'єкта. Наприклад, можуть бути компоненти для фізики, штучного інтелекту, анімації, здоров'я персонажа. Це дає гнучкість і можливість з легкістю комбінувати компоненти для створення різноманітних типів персонажів і систем гри.

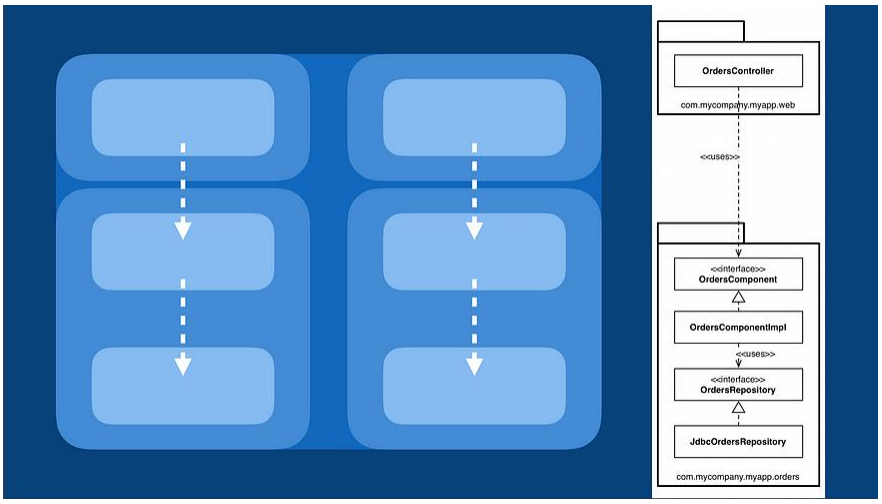


Рисунок 2.3 – Приклад Component-Based Architecture

Подійно-орієнтований підхід також є популярною в реалізації ігор (рисунок 2.4). В цій архітектурі компоненти системи взаємодіють між собою шляхом обміну

подіями. Основною ідеєю є те, що компоненти відправляють та приймають події для спілкування між собою, а не безпосередньо викликають методи один одного. Коли відбувається певна подія, відправник повідомляє про це, і компоненти, які підписалися на цю подію, можуть відреагувати на неї.

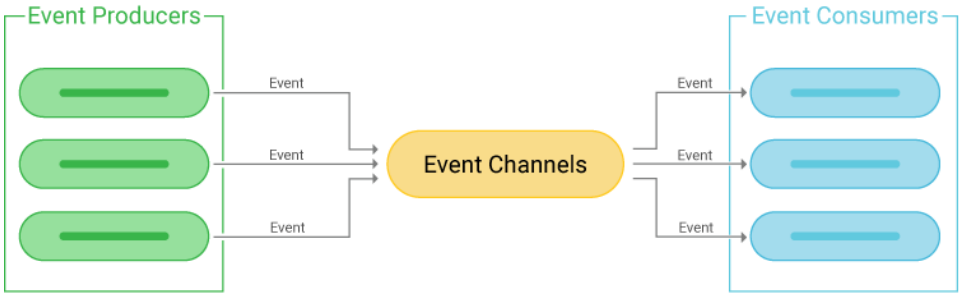


Рисунок 2.4 – Приклад Event-driven architecture

Але використання тільки одного з підходів до архітектури гри сьогодні використовується рідко адже в залежності від специфіки проєкту тільки один із підходів не здатний реалізувати всі потреби від системи, тому частіше використовують змішані архітектури. Наведемо декілька прикладів змішаних архітектур, які можуть бути використані в реалізації ігор.

ECS (Entity-Component-System) з подійно-орієнтованою архітектурою. Використання ECS архітектури спільно з подійно-орієнтованою архітектурою може бути потужним комбінацією. ECS відповідає за організацію компонентів та систем, а подійна система дозволяє компонентам обмінюватися подіями та спілкуватися між собою. Це може забезпечити ефективну та гнучку систему розроблення ігор.

MVC (Model-View-Controller) з подійно-орієнтованою архітектурою. Комбінування архітектури MVC з подійно-орієнтованою архітектурою може бути використано в створені ігор. MVC дозволяє розділити логіку, відображення та взаємодію з користувачем, тоді як подійна архітектура може бути використана для взаємодії між компонентами, сповіщення про зміни та реагування на події гри.

Компонентно-орієнтована архітектура, поєднана з подійно-орієнтованою парадигмою [25] – є підходом до створення програмного забезпечення, який поєднує переваги компонентної архітектури та подійної моделі. Кожен компонент має конкретну функціональність та внутрішній стан, і він може бути доданий або видалений з системи за потребою. Компоненти можуть створювати події, на які інші компоненти можуть підписатися та реагувати на них.

Змішана архітектура може бути корисною для певних типів ігор або для розв'язання конкретних проблем, дозволяючи поєднати переваги різних архітектурних підходів і створити більш гнучку, універсальну та ефективну систему створення ігрових проєктів.

Розглянувши типи архітектури, які можна використати при реалізації застосунку потрібно визначити, які із них можуть бути використані для проєктування і реалізації мого ігрового застосунку.

Відсічемо варіанти які будуть надмірними для цього проєкту, а саме клієнт-серверний підхід, так як гра не планується як мережева в майбутньому.

Через те що архітектура має відповідати таким нефункціональним вимогам, як легка розширюваність і модульність, поганим підходом буде монолітна архітектура, так як вона сильно збільшує зв’язаність та ускладнює кодову базу, що призведе до неможливості швидко вносити зміни і перевикористовувати код, а ці властивості архітектури дуже важливі для ігор [24].

Такі архітектурні підходи, як ECS можуть задовільнити потреби але потрібно звернути увагу на те, що цей підхід в першу чергу підходить для реалізації симуляцій різного типу, через свою орієнтованість на роботу з великою кількістю однотипних об’єктів. Аналізуючи функціональні вимоги, стає зрозуміло, що цей підхід буде надмірним. Натомість, краще звернути увагу на простішу версію ECS, а саме компонентний підхід, який полягає в тому, щоб розділити логіку програми на незалежні компоненти, які можна додавати, видаляти та змінювати в залежності від потреб, такі компоненти на відміну від тих що використовуються в ECS будуть зберігати в собі, як і дані, так і логіку для роботи з ними. Фреймворк Unreal Engine

побудований з орієнтацією на такий архітектурний стиль. Недоліками цього підходу є менша оптимізація і трохи більша зв’язаність, але дивлячись на масштаб проєкту цей підхід буде самим оптимальним.

Також корисними було б використання подійно-орієнтованого підходу так, як гра це набір об’єктів, які постійно реагують на дії користувача, але використання цього підходу з ООП створить проблеми зв’язаності.

Слідуючи з цього аналізу було зроблено висновок, що для даного проєкту найоптимальнішим рішення є вибір компонентної архітектури з подійно-орієнтованим підходом. Така змішана архітектура дасть змогу швидко реалізовувати функціонал гри, який легко буде змінювати та додавати за рахунок використання компонентної архітектури, а подійно-орієнтований підхід дасть змогу легко реалізовувати взаємодію об’єктів [26]. Поєднання цих двох підходів дозволить створити гнучку, легко змінювану та розширювану систему. Компоненти можуть взаємодіяти між собою шляхом створення та обробки подій, що забезпечує слабку залежність та дозволяє гнучко змінювати поведінку системи.

В результаті аналізу існуючих архітектурних підходів та підходів архітектурою проєкту було визначено змішану архітектуру, яка включає компонентно-орієнтований та подійно-орієнтований підхід.

2.2 Опис декомпозиції

2.2.1 Загальна декомпозиція

Після того як було обрано архітектуру потрібно провести декомпозицію проєкту за основними напрямками [27]. Декомпозиція архітектури проєкту, в свою чергу – це процес розбиття великого і складного проєкту на менші, більш керовані компоненти для полегшення управління та розуміння структури проєкту.

Почати декомпозицію було вирішено з загальної, це дасть поверхневу картину складових проєкту, що в свою чергу полегшить всі інші етапи

декомпозиції. Загальна декомпозиція як і декомпозиція всіх рівнів проводиться на основі повних функціональних і нефункціональних вимог. Її ціллю є визначення основних сутностей проєкту та опису їх характеристик.

Загальними сутностями було визначено такі ігрові об’єкти і системи, як :

− фізичний, графічний та анімаційні системи, які будуть обробляти логіку, анімації і рендер об’єктів ігри; цей модуль буде основою рушія;

− ігровий цикл – система, яка буде в циклі приймати введення, після чого відправляти до об’єктів ігри введення та обновляти стан ігри, після чого на основі стану рендерити зображення; цей модуль також є частиною рушія;

− екземпляр ігри – об’єкт який має бути найвищим в ієрархії ігрових модулів та існуватиме в одному екземплярі, буде використовуватись для керування логікою зміни рівнів та збереження проміжних даних між рівнями;

− рівень – об’єкт, який буде зберігати в собі всіх акторів і завантажувати їх при запиті та керувати загальною логіку рівня;

− ігровий режим – об’єкт який зберігає дані про контроллер і ігрового персонажа, а також зможе реалізувати функціонал пов'язаний із режимом гри, наприклад респавном гравця, ворогів та стану гри в загальному;

− контроллер гравця – система, яка буде обробляти команди гравця і передавати їх поточному ігровому персонажу;

− головний персонаж – ігровий об’єкт, яким має керувати гравець, він має переміщуватись по карті, атакувати ворогів звичайною атакою і здібностями, отримувати пошкодження і лікуватись, отримувати досвід і підвищувати свій рівень та взаємодіяти з нейтральними ігровими об’єктами;

− ворожий персонаж – ігровий об’єкт, який має керуватись системою AI, він має переміщуватись по карті, атакувати ігрового персонажа атаками і здібностями, отримувати пошкодження і лікуватись;

− нейтральний персонаж, який має реагувати на взаємодію з ігровим персонажем у вигляді діалогу чи видачі нагороди;

− система інтерфейсу, яка має створювати відтворювати і приховувати елементи інтерфейсу, такі як: головне і ігрове меню, ігровий інтерфейс, вікно діалогу та екран завантажень;

− система збереження і завантаження, яка має зберігати весь ігровий прогрес і завантажувати його при запиті.

2.2.2 Модульна декомпозиція

Після того як була проведена загальна декомпозиція, можна провести модульну декомпозицію. Модульна декомпозиція – це процес розбиття складного проєкту на менші, самостійні модулі або компоненти. Під час модульної декомпозиції система або проєкт розбивається на окремі модулі, які виконують конкретні функції або мають чітко визначені обов'язки.

Першими модулями повинні бути керуючі модулі, які дадуть можливість керувати логікою гри на різних рівнях. Потрібно модуль, який буде відповідати за екземпляр гри і буде займатись збереженням загальних даних і завантаженням рівнів. Ігрові правила будуть відслідковуватись в модулі ігрового режиму, який дасть можливість змінювати правила, контролер гравця і самого ігрового персонажа. Важливим модулем буде контроллер гравця, він буде обробляти ввід гравця, після чого передавати його ігровому персонажу та давати можливість, при потребі легко змінити персонажа.

Ігровий персонаж і ворожий персонаж мають підтримувати бойову логіку, ця логіка є спільною, як для ігрового, так і ворожого персонажа. Така логіка може знадобитись і для інших об’єктів в інших проєктах тому правильно буде виділити її в окремий модуль компонент, який можна буде розробляти окремо і легко переносити. Модуль бойової системи повинен забезпечити можливість проводити звичайні атаки і атакувати здібностями.

Як ігровий персонаж так і ворожий повинен мати здоров’я, яке можна буде

зменшувати чи відновлювати, а при падінні значення до мінімуму обробляти цю подію. Для цієї задачі потрібно виділити ще один модуль компонент, який буде відповідати за цей функціонал. Він також має бути автономним і з легкістю впроваджуватись в інших частинах проєкту чи інших проєктах. Цим модулем буде модуль стану здоров'я.

Ще одною важливою частиною функціоналу є переміщення, його використовує як і ігровий персонаж так і ворожі, також ця функція може повторно використовуватись для інших ігрових об'єктів. Цей модуль також буде окремим компонентом і він повинен надавати функціонал який буде реалізувати переміщення об'єкта по рівню.

Під час ігрового процесу головний ігровий персонаж має отримувати досвід за різні дії (бої, проходження рівня, виконання завдання), який буде підвищувати рівень і відкривати нові можливості. Функціонал, який буде забезпечувати процес прокачування, та збереження статистики гравця потрібно також виділити в окремий модуль компонент прогресу.

Для взаємодії з нейтральними персонажами потрібно реалізувати функцію діалогу, яка буде відображати в інтерфейсі репліки і перемикатись по ним, та давати можливість обробляти завершення репліки чи діалогу. Цей функціонал потрібно виділити в модуль компонент діалогів.

Головним об'єктом в проєкті є ігровий персонаж і його функціонал полягає в використанні інших модулів різними способами, тому його також можна виділити в окремий модуль ігрового персонажа, в собі він буде агрегувати всі потрібні йому функціональні компоненти. Так само функціонал ворогів потрібно виділити в окремий модуль ворожого персонажа.

Для взаємодії з гравцем через елементи інтерфейсу потрібно відділити функціонал, який буде обробляти запити на відображення потрібних елементів інтерфейсу, в окремий модуль взаємодії з інтерфейсом.

Для збереження і завантаження гри в контрольних точках та при запиті гравця потрібно реалізувати функціонал збереження і завантаження стану ігрового

світу та виділити його в модуль завантаження і збереження.

Весь функціонал, в результаті, було розділено на модулі, які відповідають за конкретний функціонал і є повністю самостійними, це значить що модульна декомпозиція була проведена успішно. Результатом цього етапу декомпозиції став визначений набір модулей проєкту, а саме:

- модуль екземпляру гри;
- модуль ігрових правил;
- модуль збереження і завантаження;
- модуль контроллера персонажа;
- модуль контроллера ворога;
- модуль компонент бойової системи;
- модуль компонент стану здоров’я;
- модуль компонент переміщення;
- модуль компонент прогресу;
- модуль компонент діалогів;
- модуль ігрового персонажа;
- модуль ворожого персонажа;
- модуль нейтрального персонажа;
- модуль інтерфейсу користувача.

2.2.3 Декомпозиція моделі переходів станів

Декомпозиція моделі переходів станів є процесом розбиття складної моделі на менші та більш керовані компоненти для полегшення аналізу, реалізації та управління становою логікою системи. Цей підхід дозволяє розділити поведінку системи на частини, що можуть бути аналізовані та модифіковані незалежно одна від одної, особливо вона важлива при реалізації з використанням анімацій для персонажів, адже логіка анімаційних сценаріїв керується перевіркою стану об’єкту.

Декомпозиція моделі переходів станів включає ієрархічну структуру станів. Це означає, що стани можуть бути груповані в більші стани, які представляють вищий рівень абстракції. Це полегшує управління складними моделями, оскільки вони можуть бути розбиті на менші та більш зрозумілі підгрупи.

Для проєктування архітектури розроблюваного проєкту було не менш важливо виділити стани визначених моделей. Так найцікавішим суб'єктом для розгляду моделі станів є ігровий персонаж, так як він постійно переходить від однієї дії до іншої, змінюючи свій стан.

Стани в яких може знаходитись ігровий персонаж такі:

- стоїть очікуючи;
- веде діалог;
- стоїть досліджуючи;
- біжить;
- атакує звичайною атакою;
- атакує здібністю;
- сфокусований но ворогові;
- робить стрибок.

Ще одним важливим об'єктом розгляду моделі станів є ворожий персонаж, він може знаходитись в таких станах:

- патрулює в пошуках ворога;
- очікує ворога;
- атакує ворога.

Нейтральний персонаж може знаходитись тільки в двох станах:

- очікує діалогу;
- веде діалог.

Вікна ігрового інтерфейсу також повинні мати можливість зміни їх стану відображення для уникнення зайвого створення об'єктів:

- відображається;
- прихований;

− активний;

− не активний.

Розуміння цих станів дозволяє забезпечити правильні і плавні переходи, уникнути непередбачуваної поведінки і помилок. Це особливо важливо для забезпечення хорошого користувацького досвіду, коли дії гравця повинні мати логічні і зрозумілі наслідки.

2.3 Опис залежностей

Опис залежностей є важливим етапом при проєктуванні системи або програмного продукту. Він допомагає розібратися в тому, які компоненти або модулі системи взаємодіють між собою та залежать один від одного. У загальному, опис залежностей є потужним інструментом при проєктуванні системи, який допомагає зрозуміти та управляти взаємозалежностями між компонентами, забезпечувати розширюваність та підтримку системи.

Першим етапом став опис між модульних залежностей. Опис міжмодульних залежностей може бути представлений у вигляді діаграм або списків, які вказують на взаємозалежності між модулями та напрямок комунікації між ними. Для побудови такої діаграми було описано зв’язки визначених раніше модулів:

− екземпляр гри звертається до рівнів гри та ігрового рушія;

− рівень звертається до всіх об’єктів поміщених в рівень та ігрового рушія;

− ігровий режим звертається до контролера та модуля ігрового персонажа та керованих ним загальних ігрових об’єктів;

− контроллер звертається до модуля ігрового персонажа;

− ігровий персонажа може звертатись до компонента бойової системи, компонента стану здоров’я, прогресу та компоненту переміщення;

− нейтральний персонаж взаємодіє з компонентом діалогів та ігровим персонажем, а також може звертатись до ігрового режиму;

− компонент бойової системи взаємодіє з ворожим персонажем і ігровим персонажем та може звертатись до ігрового режиму;

− компонент діалогів звертається до модуля UI;

− модуль збереження і завантаження звертається до модуля ігрового персонажа, ворожого і нейтрального.

На основі аналізу була побудована діаграма міжмодульних зв’язків зображена на рисунку А.2 у додатку А, яка наглядно відобразила зв’язки модулів і представлена далі.

Опис зв’язків проведений в цьому підпункті дасть можливість оптимально провести детальне проєктування модулів та даних в наступному пункті за рахунок кращого розуміння зв’язків модулів і даних.

2.4 Опис інтерфейсу модулів

Інтерфейси компонентів повинні забезпечити можливість отримання даних зовнішнім модулям, при цьому інкапсулювавши приватні дані компонентів. Так для компоненту стан здоров’я було визначено що оптимальний інтерфейс має складатись з двох публічних методів GetHelth та SetHelth, що дасть можливість користувачам компоненту, отримати поточне здоров’я або змінити його, без ризику вказання невалідних даних. Для компоненту прогресу інтерфейс складатиметься з валідуючих методів для отримання і встановлення поточного рівня досвіду і тільки отримання рівня гравця – GetExp, SetExp та GetLvl, дані про максимальний рівень досвіду на рівні скриваються. Інтерфейс для компоненту бойової системи матиме подібний вигляд надаючи можливість отримати поточну силу атаки. Інтерфейс компоненту переміщення має надати можливість отримати і змінити масу, швидкість та вектор прискорення, а також дізнатись чи об’єкт переміщається за допомогою методу IsMove, який на основі даних визначить чи об’єкт знаходить в русі на даний момент. Модуль діалогів має інтерфейс, який надає можливість

звернутись до нього з запитом розпочати діалог. Модуль UI повинен підтримувати інтерфейс, який дасть змогу вивести віджет з переданою фразою діалогу DrowDialog(text). Інтерфейси ігрового персонажа мають забезпечити можливість звернення до них з метою зменшити або збільшити їх поточне здоров’я.

В результаті було визначено і описано інтерфейси модулів проєкту, загальне визначення, яких, зображено в таблиці (таблиця 2.1).

Таблиця 2.1 – Модулі та їх інтерфейси

Модуль:	Інтерфейс
HealthComponent	GetHealth(), SetHealth(), IsDead()
LevelComponent	GetExp(), SetExp(), GetLvl()
FightComponent	GetDamage(), Attack(), SpellAttak(), IsAttack().
MovementComponent	GetMass(), SetMass(), GetGrav(), SetGrav(), GetAccel(), SetAccel(), AddAccel(), IsMove()
Character	GetComponentByClass()
GameController	MoveTo(), Jump(), Dush(), Attack(), SpellAttack(), GetCharacter(), SetCharacter
GameMod	GetGameController(), SetGameController(), GetCharacter(), SetCharacter().
DialogComponent	StartDialog(), StartNextSentence(), EndDialog().
UISystem	AddWidgetByClass(), RemoveWidget(), HideWidget().

2.5 Проєктування інтерфейсу користувача

На цьому етапі потрібно розглянути, які функції повинен підтримувати інтерфейс, визначитись з екранами, які потрібно розробити та створити відповідні wireframes (макети), на яких буде зображено загальну структуру інтерфейсу.

Аналізуючи функціональні вимоги до застосунку можна виділити дві основні групи екранів, які будуть потрібні:

− меню – екрани, що дадуть можливість розпочати гру, вибрати налаштування графіки та звуку, переглянути вивчені здібності персонажа;

− HUD (Head-Up Display) – екрани, що будуть виводитись поверх основного ігрового процесу, та надавати інформацію про поточний стан різних компонентів, або виводити важливі повідомлення.

Меню гри буде складатись з таких екранів:

− головне меню (кнопка переходу до ігри, переходу до налаштувань та виходу з ігрового застосунку);

− меню налаштувань (два вікна налаштувань (графіки, звуку), 2 кнопки для перемикання відображення між ними);

− вікно здібностей (дерево відкритих здібностей).

HUD буде складатись з таких елементів:

− основний ігровий екран (стрічка (шкала) здоров’я, мітка вибраної стихії,, кількість досвіду і рівень);

− динамічний вікно інтеракції (елемент з кнопкою для взаємодії та надпис, що дає зрозуміти з чим є можливість взаємодіяти);

− динамічне вікно діалогу (елемент з текстом діалогу);

− стрічка здоров’я ворога (елемент прив’язаний до кожного ворогу).

Розуміючи структуру та визначившись із елементами, які будуть складати екрани інтерфейсу користувача, потрібно створити макети екранів, щоб розуміти і мати можливість проаналізувати, як ці елементи будуть розміщуватись відносно екрану і один одного, що дасть змогу швидко реалізувати їх в подальшому.

Аналізуючи інтерфейс ігрових додатків, наявних в магазинах, було визначено, яке розміщення елементів основного ігрового екрану ефективне. Готовий макет зображено на рисунку А.3 в додатку А. Показники персонажа розміщують зверху зліва і там розміщують елементи, які дають зрозуміти стан гравця з інших поглядів.

Головне меню – це екран для переходу, на ньому розміщують зазвичай тільки кнопки, які запускають гру або відкривають функціональні вікна, наприклад вікно налаштування, ще вони дають можливість вийти з гри. Для заповнення простору часто додається фон зі сценою гри. На основі цього було створено макет головного меню зображений на рисунку А.4 у додатку А.

Екран ігрових здібностей буде відображати ряд здібностей і в залежності від рівню активність маркера здібності буде змінюватись. Макет зображено на рисунку А.5 у додатку А.

Також потрібно визначити розміщення та вигляд в загальному для динамічних елементів діалогу і взаємодії з об’єктом. Зазвичай елемент, який повідомляє про можливість взаємодії розміщується нижче середини екрану, а вікно діалогу в нижній частині екрану. Макет на якому зображено ігровий інтерфейс з одразу двома активними елементами зображений на рисунку А.6 у додатку А.

Реалізувавши всі важливі макети інтерфейсу, буде можливо створити відповідні елементи інтерфейсу під час реалізації мого ігрового застосунку.

2.6 Детальне проєктування модулів

Визначення основних модулів було проведено в попередньому розділі, було визначено самі модулі, їх функціонал та розглянуто в загальному їх зв’язки. Першим етапом детально проєктування модулів стане уточнення модулів, як об’єктів і повний опис зв’язків, який дасть можливість краще зрозуміти структуру проєкту і продовжити ефективну реалізацію архітектури.

Найвище в ієрархії стоїть об’єкт GameInstanse. Цей об’єкт існує в єдиному екземплярі для кожного екземпляру програми. Він повинен містити в собі всі рівні проєкту у вигляді масиву об’єктів Level. Така асоціація визначається, як залежність одного до багатьох. Об’єкт Level в свою чергу зберігає в собі масив всіх ігрових об’єктів, які будуть потрібні на ньому. Він асоціюється до Character, як залежність одного до багатьох, а до GameMod шляхом композиції. Об’єкт GameMod, в свою чергу, композиційно пов'язаний з PlayerController і UI System, та агрегує поточний PlayerCharacter. В PlayerController зберігається екземпляр PlayerCharacter, що визначає їх зв'язок, як композицію. Базовий Character містить в собі MoveComponent і пов'язаний з ним композицією. Від Character наслідуються PlayerCharacter, EnemyCharacter та NPCCharacter. Як PlayerCharacter так і EnemyCharacter відоносяться композицією до HealthComponent та FightComponent, PlayerCharacter ще додатково містить LevelComponent. Також на цьому етапі було виділено додатковий об’єкт Spell, який буде представленням магічної здібності і містити дані пов’язані з її пошкодженням і дальністю дії. FightComponent буде залежати від Spell, як два до багатьох.

На основі проведеної деталізації зв’язків між об’єктами, для кращої наглядності, було побудовано діаграму зв’язків об’єктів, яка зображена на рисунку А.7 у додатку А. Діаграма зв’язків об’єктів допомагає забезпечити, щоб всі компоненти працювали разом узгоджено, полегшуючи спільну роботу над проєктом і сприяючи створенню коду, який легко розширювати. Вона також є корисним інструментом, що дозволяює чітко пояснити складні взаємодії та залежності між об'єктами.

Найкращим методом опису взаємодії об’єктів є діаграми послідовні UML. Зазвичай діаграма послідовності визначає один сценарій [28]. На діаграмі показані екземпляри об'єктів та повідомлення, якими обмінюються об'єкти у межах одного прецеденту. Ці діаграми дають детальне розуміння того, як модулі і компоненти взаємодіють один з одним.

Першим сценарієм, для опису було визначено вхід в гру. Під час цього

сценарію GameInstance завантажує рівень після чого рівень завантажує всі об’єкти з диску і створює динамічно, в тому числі UISystem, після чого завдяки реалізації патерну Observer відправляє системі повідомлення про те що потрібно завантажити інтерфейс ігрового меню, далі система обробляє все введення в інтерфейс, якщо було натиснуто кнопку переходу до ігрового рівня, поточний рівень знищується і завантажується новий рівень. Діаграма яка наглядно зображує послідовність зображена далі (рисунок 2.5).

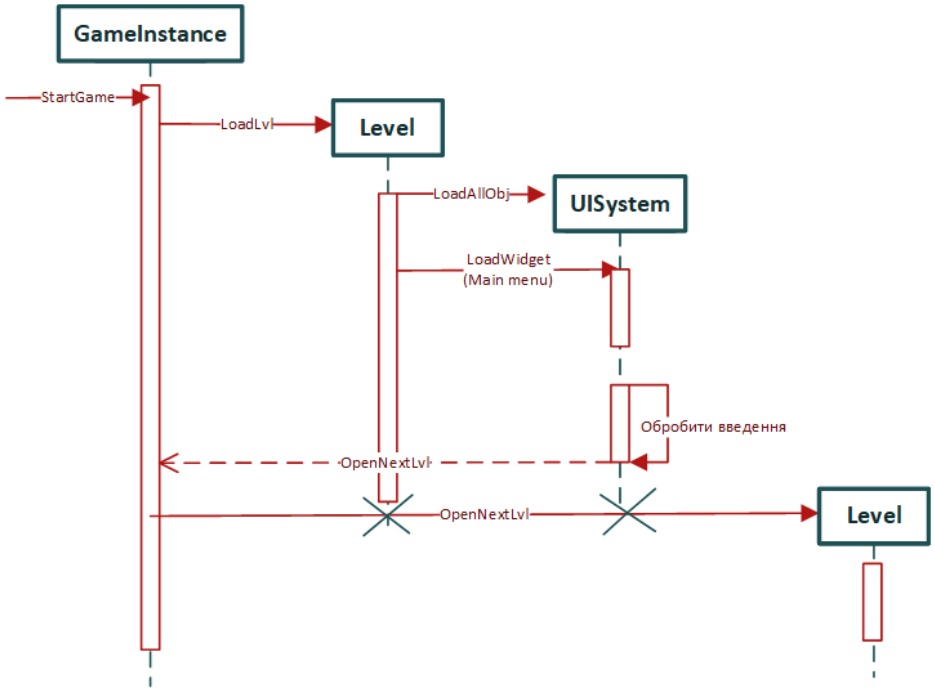


Рисунок 2.5 – Діаграма послідовності для завантаження головного меню

Ще одним сценарієм який важливо розглянути буде нанесення пошкодження ігровому персонажу. Під час цього сценарію завдяки реалізації патерну Observer до ігрового персонажа приходить повідомлення, що нанесено урон, на це повідомлення реагує компонент здоров’я, змінюючи стан здоров’я, якщо здоров’я більше 1, то про це повідомляється система інтерфейсу, яка змінює шкалу здоров’я, якщо здоров’я мене або рівно 0, то повертається гравцю повідомлення про смерть. Діаграма послідовності що відображає взаємодію цих об’єктів (рисунок 2.6) була побудована на основі цієї деталізації взаємодії.

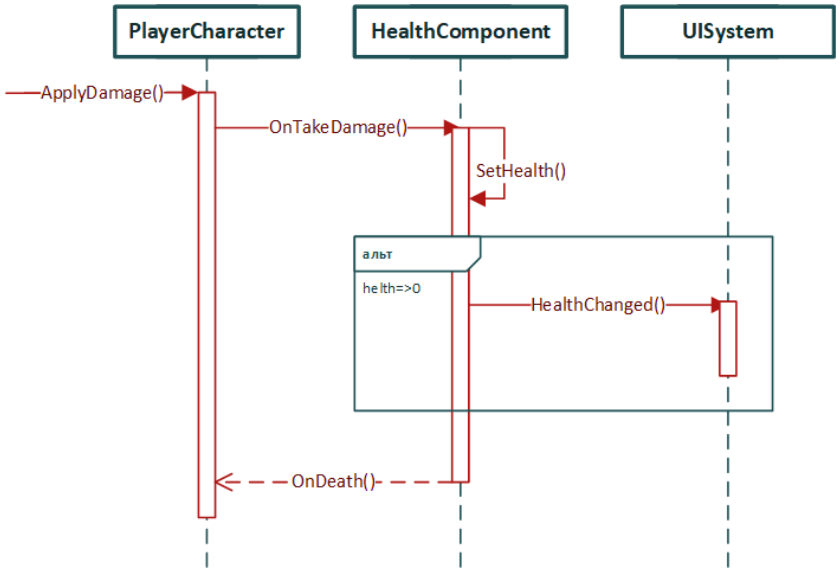


Рисунок 2.6 – Діаграма послідовності для сценарію отримання пошкодження

Важливим сценарієм під час якого відбувається взаємодія між багатьма об’єктами є вистріл магією. Під час цього сценарію до GameController, приходить сповіщення про те що була натиснута клавіша, після чого контроллер завдяки реалізації патерну Observer повідомляє про це PlayerCharacter. Персонаж обробляє це повідомлення після чого він повідомляє про це FightComponent та зміню свій стан на атакуючого, про що дізнаєтья анімаційний сценарій і застосовує відповідну анімацію. FightComponent обробляє це повідомлення таким чином: отримує поточне розміщення персонажа гравця, через інтерфейс цього об’єкту і створює біля нього новий об’єкт Spell, якому задає розміщення, швидкість, напрям та пошкодження, які він повинен буде нанести. Після створення об’єкт використовуючи логіку переміщення здібність переміщується, та при виникненні події удару намагається нанести пошкодження, відтворює візуальний і звуковий ефект і знищується. Створена діаграма зображена на рисунку А.8 у додатку 8.

Подібним чином відбувається пересування гравця за рахунок звернення котроллера до MoveComponent через PlayerCharacter. При натиску на кнопку ходьби про це повідомляється PlayerController, який в залежності від клавіші, що була натиснута, повідомляє про це PlayerCharacter, він в свою чергу робить всю потрібну зовнішню логіку після чого повідомляє про переміщення

MoveComponent, який уже відтворює логіку переміщення, та змінює розміщення персонажа. Додатковов відбувається змінна стану персонажу на «біжить», це оновлює анімаційний скрипт, що веде до зміни анімації в залежності від швидкості та напрямку прискорення компоненту MoveComponent. Діаграма послідовності, яка відображає те, як ці об’єкти взаємодіють також була створена і зображена на рисунку А.9 у додатку А. Ця діаграма наглядно відтворює, як зазвичай користувач починає взаємодіяти з компонентами головного персонажа.

Побудовані діаграми послідовності дадуть краще розуміння всіх основних механік, а також наглядно покажуть, як і які об’єкти спілкують з іншими під час виконання основних сценаріїв.

Для того, щоб краще розуміти внутрішню роботу класів корисними буде створення діаграм станів об’єктів. Діаграми станів (state machine diagrams) – це відома технологія опису поведінки системи [29].

Головна діаграма станів, яка нас цікавить це переходи станів PlayerCharacter. Він за замовчування знаходиться в стані стоїть, після натиску клавіш його стан змінюється на ходить, якщо натиснуті клавіші переміщення, стрибає, якщо натиснуто клавішу стрибку або атакує при натиснутій атаці. Якщо персонаж отримає урон, який буде смертельним тоді стан зміниться тимчасово на помирає і після цього об’єкт видалиться. Діграма станів побудована на основі детального опису має такий вигляд (рисунок 2.7).

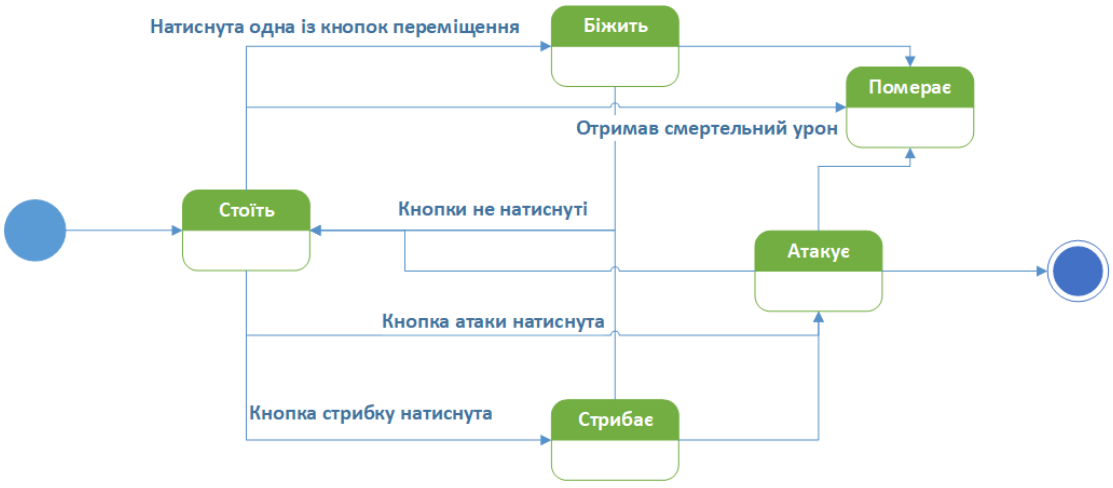


Рисунок 2.7 – Діаграма станів PlayerCharacter

Об’єкт NPC також потребує опису його внутрішніх станів. Неігровий персонаж може перебувати або в очікуванні, або патрулі, якщо Player Character підійде до NPC і натисне кнопку взаємодії, NPC перейде в стан розмови і активує діалог. Після того, як діалог буде завершено завершиться і автомат об’єкта і його буде знищено. Діаграма станів NPC зображена на рисунку А.10 у додатку А.

Ще одним об’єктом який змінює стани є Spell. Його діаграма доволі проста адже немає розгалужень. Spell створюється після чого переходить в стан польоту і знаходиться в ньому до поки не зіткнеться з об’єктом, після зіткнення стан зміниться на стоїть, після чого стан буде змінено вибухає. Далі об’єкт знищується. Діаграма для цього об’єкта (рисунок 2.8) буде доволі проста.

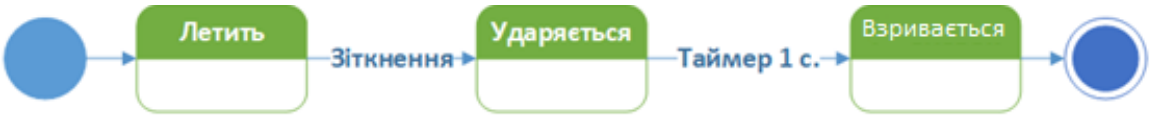


Рисунок 2.8 – Діаграма станів Spell

Створення цих діаграм дасть змогу, також швидко реалізувати анімацій скрипти, які зазвичай будуються, як варіації діаграм станів, state machine.

Завершальним етапом детального опису поведінки буде створення блок-схем важливих методів об’єктів, розуміння внутрішньої роботи яких може бути не очевидним або потребувати специфічних вимог. Блок-схема алгоритму зображає послідовність блоків, з'єднаних між собою стрілками, які вказують послідовність виконання і зв'язок між блоками. Всередині блоків записується їх короткий зміст.

Для детального зображення алгоритму на блок схемі, було вирішено обрати методи, які займаються прорахунком статистики гравця такої, як показник здоров’я, досвід та рівень.

Алгоритм який змінює здоров’я полягає в тому що на вхід передається пошкодження, пошкодження зберігається, після від поточного здоров’я віднімається урон і перевіряється чому дорівнює значення поточного здоров’я, якщо воно більше максимального то встановлюється максимальне, якщо менше то

нуль, а якщо в границях то алгоритм іде до наступного кроку, перевіряється чи це був додатній урон (пошкодження) чи від'ємний (лікування), в залежності від відповіді відправляється сповіщення всім слухачам. Блок-схема цього алгоритму має такий вигляд (рисунок 2.9).

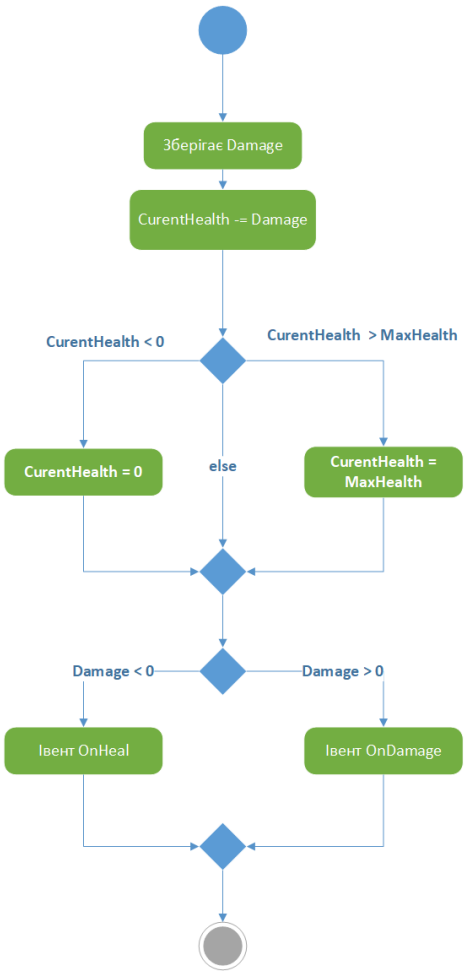


Рисунок 2.9 – Діаграма діяльності алгоритму зміни здоров'я

Останнім етапом проєктування архітектури стало створення діаграми класів, яка уже детально відобразила зв'язки класів проєкту, дані кожного із класів та інтерфейси цих класів. [30] Ця діаграма будувалась на основі проведеної декомпозиції, в якій було визначено модулі, що стали класами, для визначення аргументів класу було використано результати декомпозиції даних проведені в другому розділі, для визначення публічних методів був використаний опис інтерфейсів. Ця діаграма дасть можливість, в комбінації з описаними раніше пове-

дінковими діаграмами, перейти до реалізації ігрового застосунку, дотримуючись всіх вимог. Готова діаграма зображена на рисунку А.12 у додатку А.

2.7 Аналіз та вибір технологій і методів реалізації застосунку

Створення ігрового застосунку є комплексним процесом, що вимагає детального планування та обґрунтованого вибору технологій і методів. Від правильного вибору залежить не тільки якість кінцевого продукту, але й ефективність розроблення, дотримання строків та бюджетних обмежень. У цьому розділі буде проведено аналіз доступних технологій та методів, які можна використовувати для створення гри в жанрі Action-RPG. Буде розглянуто різні ігрові рушії, методології розроблення, інструменти та підходи, щоб обрати найоптимальніші для мого реалізації ігрового. Основна мета цього розділу – забезпечити технічну основу, яка дозволить створити високоякісний ігровий продукт, відповідний усім вимогам та очікуванням. Аналіз врахує сучасні тенденції в індустрії ігор, досвід успішних проєктів, а також визначені вимоги, специфіку жанру і цільову аудиторії.

Найважливішим рішенням для мого застосунку є вибір ігрового рушія, на базі якого буде відбуватись реалізація. Різні рушії мають свої особливості та функціональні можливості, які визначають, які ефекти, анімації, фізичні властивості та базові ігрові можливості можна легко реалізувати. Рівень продуктивності ігрового рушія визначає, наскільки швидко можна створювати, тестувати та відлагоджувати гру, а також як ефективно вона працюватиме на різних пристроях. Деякі рушії можуть краще масштабуватися для створення великих та складних світів, в той час, як інші можуть бути оптимізовані для швидкого створення менших проєктів. Існування активної спільноти розробників, широкий вибір документації, туторіалів та ресурсів для навчання може значно полегшити процес створення і розв’язання проблем тому цей критерій також потрібно оцінити.

На сьогоднішній день нішу універсальних, безкоштовних для невеликих проєктів та розвинутих рушіїв займають два гіганти Unity та Unreal Engine.

Unity є популярним ігровим рушієм, який здобув визнання завдяки своїй простоті використання та широкому спектру інструментів. Для початківців та розробників з обмеженим досвідом, Unity може бути привабливим варіантом, оскільки він надає зручний інтерфейс та швидкий спосіб створення ігор. Мова програмування C#, яка використовується в Unity, є досить простою для вивчення та розуміння, що може полегшити процес розроблення для тих, хто має попередній досвід у програмуванні.

Проте, Unity може мати деякі обмеження щодо продуктивності та графічних можливостей порівняно з Unreal Engine. Для складних та графічно вимогливих проєктів, Unity може виявитися менш підходящим варіантом. Також слід враховувати, що хоча Unity підтримує C++, більшість реалізації відбувається за допомогою C#, що може вимагати адаптації для тих, хто звик до роботи з C++.

Unreal Engine – це потужний та високопродуктивний ігровий движок, розроблений компанією Epic Games. Він використовується для створення ігор різних жанрів, від інді-проєктів до AAA-тайтлів. Unreal Engine відомий своєю потужністю та гнучкістю, що дозволяє розробникам створювати якісні та інноваційні ігри для різних платформ.

Його потужний графічний движок дозволяє створювати вражаючі візуальні ефекти та реалістичні ігрові світи. Багатофункціональний інструментарій, який включає в себе редактор матеріалів, систему фізики та штучного інтелекту, дозволяє розробникам реалізувати широкий спектр ідей та концепцій у своїх проєктах.

Однією з головних переваг Unreal Engine є його вбудована мережева підтримка, яка дозволяє створювати онлайн ігри та ігри з мультиплеєрним режимом без значних труднощів. Це робить Unreal Engine привабливим вибором для створення ігор, які передбачають співпрацю через Інтернет або локальну мережу.

Крім того, Unreal Engine має активну та розгалужену спільноту розробників, що надає доступ до безлічі ресурсів, включаючи документацію, уроки та сторінки

підтримки. Це дозволяє розробникам швидко вирішувати проблеми та ділитися знаннями з іншими. Така підтримка сприяє розвитку відкритого обміну інформацією та сприяє швидкій адаптації до нових технологій та тенденцій у світі ігор.

Звертаючи увагу на зручну документацію по усім складовим рушія, доступність великої кількості редакторів з коробки, неймовірно активну спільноту розробників і магазин ресурсів для ігор де часто роздають безкоштовні високоякісні продукти та вищий рівень знання C++ в порівнянні з C# було вирішено розробляти ігровий застосунок на цьому рушії.

Наступним етапом є вибір інструментів для створення ресурсів для гри, а саме IDE для написання коду, редактори анімацій та графічний редактор.

Unreal Engine підтримує кілька різних інтегрованих середовищ розроблення (IDE) для написання коду, які будуть розглянуті далі.

Visual Studio (рисунок 2.10) – це одне з найпопулярніших середовищ розроблення для роботи з мовою програмування C++. Unreal Engine має вбудовану підтримку Visual Studio, що дозволяє зручно розробляти та відлагоджувати код.

Рисунок 2.10 – IDE Visual Studio 2022

Visual Studio Code (рисунок 2.11) – це легкий та потужний текстовий редактор, який також має підтримку мови програмування C++. Він може бути використаний для роботи з Unreal Engine, але для повноцінного використання може знадобитися налаштування деяких додаткових розширень.

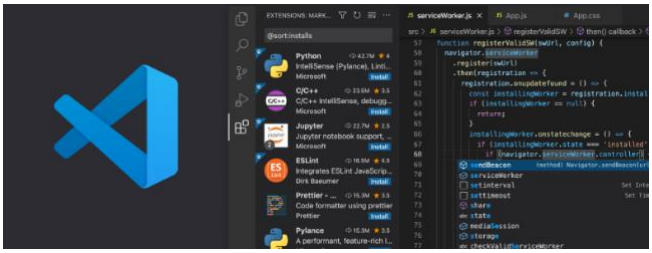


Рисунок 2.11 – IDE Visual Studio Code

JetBrains Rider (рисунок 2.12) – це інтегроване середовище, що спеціалізується на роботі з мовою програмування C#. Від версії 2021.1 також підтримує роботу на Unreal Engine за допомогою розширення Unreal Engine.

Рисунок 2.12 – IDE JetBrains Rider

Хоча всі ці редактори підтримують реалізацію для Unreal Engine, найбільша інтеграція реалізована з VS 2022 і саме це середовище розроблення надає найкращі інструменти для налагодження та профілювання.

Для створення і редагування анімацій будуть використовуватись зручні внутрішні інструменти, як-от:

− Control Rig – це набір контрольних елементів, які використовуються для керування рухом та поведінкою об'єкта можна використовувати контрольні точки, кістки трансформації та інші елементи для створення анімації;

− Anim Blueprint – це спеціальний тип Blueprint в Unreal Engine, що дозволяє створювати складні анімації та управляти рухом персонажів та об'єктів за допомогою візуального програмування.

Для редагування та створення графічних ресурсів, як-от елементи

інтерфейсу, буде використовуватись Adobe Photoshop, як найзручніший та найбільш освоєний інструмент для створення подібних ресурсів.

Останнім потрібно розглянути, які підходи до розроблення використовуються під час реалізації ігрових застосунків і обрати відповідний.

У створені ігрових застосунків часто використовуються різні методології та підходи, але два найпоширеніші – це каскадна та ітеративна (рисунок 2.13).

Каскадна методологія (Waterfall) – це традиційний підхід, в якому розроблення поділяється на послідовні етапи, такі як аналіз, проєктування, реалізація, тестування та впровадження. Кожен етап повністю завершується перед переходом до наступного. Цей підхід може бути корисним для проєктів з чіткою визначеною специфікацією, але може бути менш гнучким у випадках, коли вимоги змінюються під час розроблення.

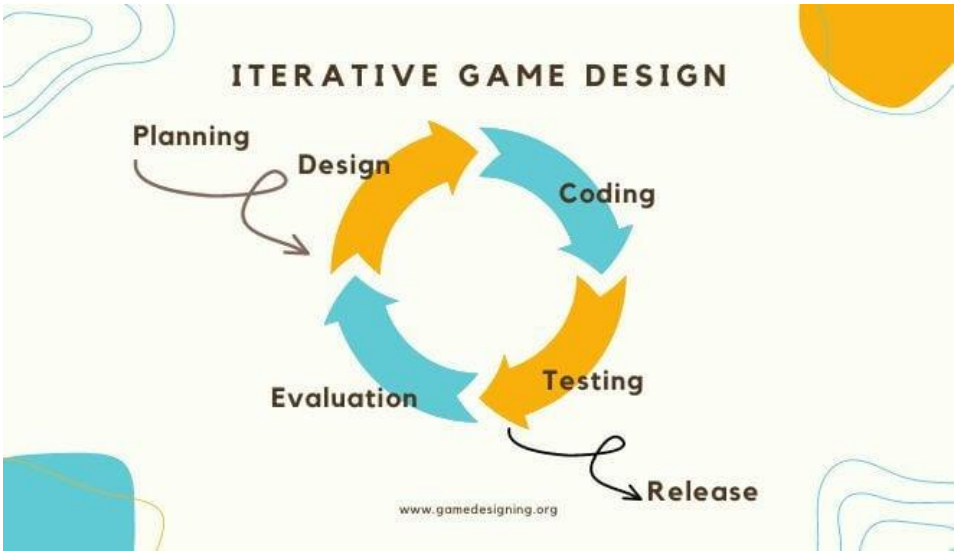


Рисунок 2.13 – Схема ітеративного процесу розроблення

У створені ігор частіше за все використовується ітеративний або гібридний підхід [18], який комбінує в собі елементи каскадного та ітеративного підходів. Такий гібридний підхід дозволяє поєднати переваги обох методологій, забезпечуючи якість та гнучкість розроблення, тому було вирішено обрати цю модель життєвого циклу для реалізації мого застосунку.

2.8 Висновки

Пройшовши всі етапи проєктування потрібно оцінити результати проведених робіт та зробити висновок. Під час проєктування було проведено аналіз існуючих архітектурних рішень, порівняння переваг і недоліків та на основі вимог і проведеного аналізу обрано архітектурний підхід для проєкту, а саме компоненту архітектуру змішану з подійно-орієнтованою архітектуро. Далі було проведено декомпозицію різних рівнів та опис зв’язків. Спочатку було в загальному визначено головні сутності проєкту. На основі визначених сутностей і архітектурного підходу було декомпозовано модулі. В наступних підпунктах було декомпозовано дані модулів та розглянуто їх зв’язки та інтерфейси.

Опираючись на модульну діаграму та визначені інтерфейси модулів було деталізовано діаграму класів, яка відобразила повну структуру архітектури проєкту. Для деталізації внутрішніх процесів окремих класів було розглянуто стани важливих об’єктів та побудовано діаграми зміни станів, для ігрового персонажа, нейтрального персонажа і магії. Критично важливі методи було детально описано і побудовано блок-схеми алгоритмів. Заключним етапом стало детальне проєктування класів. На основі проведеної декомпозиції модулів, даних та опису інтерфейсів модулів, було сформовано структуру класів, після чого, використавши проведений аналіз зв’язків об’єктів, було визначено зв’язки класів, які було відображені на діаграмі класів.

Результатом цього розділу стала повна архітектурна документація, яка включає, як структурний опис і діаграми архітектури, так і поведінкові діаграми. Ця документація дасть можливість реалізувати всі архітектурні рішення коректно, а імплементувати нові ідеї швидко.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ІГРОВОГО ЗАСТОСУНКУ

3.1 Особливості програмної реалізації на Unreal Engine

Для того, щоб розпочати програмну реалізацію модулів застосунку важливо розглянути структуру рушія Unreal Engine і особливості роботи з ним. Так першим етапом буде розгляд особливостей програмування для Unreal Engine. Основні особливості, про які потрібно знати будуть розглянуті далі.

Програмування на C++ в UE подібне до стандартного C++, використовуючи класи, функції та змінні. Вони визначаються за допомогою стандартного синтаксису C++. Кожен клас визначає шаблон для нового об’єкта або актора, який можна додатково інкапсулювати за допомогою Unreal Engine Reflection System. Unreal Engine Reflection System інкапсулює ваші класи різними макросами, які забезпечують функціональність рушія та редактора. При програмуванні за допомогою Unreal Engine можна мати стандартні класи, функції та змінні C++.

Базовим класом для об’єктів в Unreal є UObject. Кожен клас визначає шаблон для нового актора або об’єкта. Ви можете використовувати UCLASS макрос для позначення похідних класів UObject, щоб система обробки UObject знала їх.

Класи можуть містити структури. Структури – це структури даних, які допомагають організовувати пов’язані властивості членів і керувати ними. Структури можна визначити самостійно за допомогою USTRUCT() макросу.

Інтерфейси надають функції та додаткову ігрову поведінку, яку можна реалізувати в кількох або різних класах. Специфікатори метаданих контролюють, як класи, інтерфейси, структури, переліки, функції чи властивості взаємодіють із різними аспектами механізму та редактора. Кожен тип структури даних або члена має власний список специфікаторів метаданих.

Макроси UFUNCTION і UPROPERTY повідомляють UE про нові класи, функції та змінні. Ці макроси є сміттям, яке збирає рушій. Вказуючи макроси, ви можете редагувати та відображати їх у Unreal Editor.

Ігровий фреймворк Unreal Engine [19] – це набір класів, який надає вам модульну основу, на якій можна будувати свій ігровий процес. Ви можете вибирати елементи, які підходять для вашої гри, знаючи, що ці класи створені для роботи та доповнення один одного.

Основні класи (рисунок 3.1) ігрового процесу включають функції для представлення гравців, союзників і ворогів, а також для керування цими аватарами за допомогою введення гравцем або логіки ШІ. Також є класи зі створення хедз-ап дисплеїв і камер для гравців. Нарешті, класи ігрового процесу, такі як GameMode, GameState і PlayerState, встановлюють правила гри та відстежують прогрес гри та гравців. Для коректної реалізації спроєктованих модулів важливо розглянути цю уже існуючу архітектуру ігрових модулів Unreal Engine.

Екземпляр гри створюється під час запуску рушія та залишається активним, поки рушій не вимкнеться. Це клас менеджера, який фізично не присутній у грі, а існує для відстеження даних і запуску коду. Екземпляр гри не реплікується та існує незалежно на сервері та всіх підключених клієнтах у мережевій багатокористувацькій грі. Все, що ви хочете зберігати між завантаженнями рівнів, має зберігатися в екземплярі гри. Екземпляр гри також діє як менеджер для будь-якої кількості підсистем екземпляра гри, які створюються та знищуються екземпляром гри та існують протягом того самого часу, що й сам екземпляр гри.

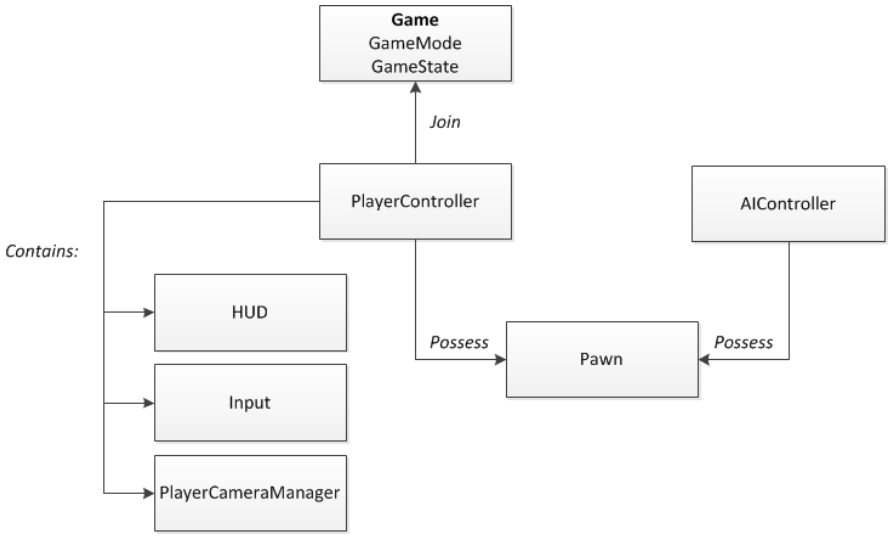


Рисунок 3.1 – Блок-схема зв’язків основних класів

Ігровий режим – це серверний клас менеджера, успадкований від класу акторів. Оскільки цей клас створюється під час завантаження рівня, він не є постійним між рівнями. Ігровий режим є першим актором, який запускається після завантаження рівня, і його можна встановити для кожної карти. Ігровий режим існує в центрі ігрового фреймворку, керуючи загальними правилами та структурою сеансу ігрового процесу та створюючи екземпляри інших акторів фреймворку після створення. Перші два – стан гри та стан гравця.

Стан гри та стан гравця – це нефізичні актори, призначені для відстеження стану гри та гравців у ній відповідно. Ці класи копіюють інформацію про свій стан між авторитетним сервером і всіма підключеними клієнтами в мережевій багатокористувацькій грі.

Ігровий режим породжує гравців, коли вони приєднуються до гри. Гравець в основному складається з контролера та пішака. Класи контролерів керують логікою, яка диктує дії гравця в ігровому світі. Існує два типи класів контролерів в UE, які широко використовуються: контролер гравця та контролер ворога. Клас контролера гравця – це клас менеджера, який може обробляти вхідні дані від людини, відображати інформацію хедз-ап і мати фізичне представлення в грі. Клас контролера штучного інтелекту – це клас менеджера, який володіє фізичними представленнями в грі та диктує свої дії за допомогою штучного інтелекту UE, включаючи: дерева поведінки, дерева станів, навігацію тощо.

Клас пішака складається з фізичного прояву гравця в ігровому світі. Клас пішака такий же важливий для створення гравця, як і клас контролера. Контролери володіють пішаками та керують пішаком для виконання дій у грі.

Клас персонажа – це підклас, похідний від пішака, який будується на основі класу пішака за замовчуванням із багатшими функціями компонентів, зокрема: компонент руху персонажа, компонент скелетної сітки та капсули колізії.

Проаналізувавши основні особливості роботи з фреймворком і рушієм є змога перейти до реалізації мого ігрового застосунку.

3.2 Програмна реалізація модулів

Типовим планом реалізації модулів для ігрового застосунку жанру Action-RPG є розроблення в такому порядку:

- весь функціонал головного персонажа;
- реалізація ворожих NPC;
- реалізації інтерактивних об’єктів;
- реалізація загального функціоналу;
- реалізація інтерфейсу.

Опираючись на це першими було вирішено створювати клас PlayerCharacter. Звертаючись до діаграма, розроблених в другому розділі, цей клас повинен володіти трьома основними компонентами: переміщення, бойових дій, здоров’я і мани та взаємодії. Для цього в клас Unreal Engine Character було добавлено посилання на відповідні компоненти (рисунок 3.2).

Для коректної зміни здоров’я також зберігається максимальне значення, що надає змогу, при запиті на змінну кількості здоров’я, верифікувати змінну здоров’я і не допускати значень більше максимуму і менше нуля. Також цей компонент відправляє подію, при змінні здоров’я і коли здоров’я рівне нулю, що значить смерть. В додатку Б можна знайти код функції, яка верифікує змінну здоров’я.

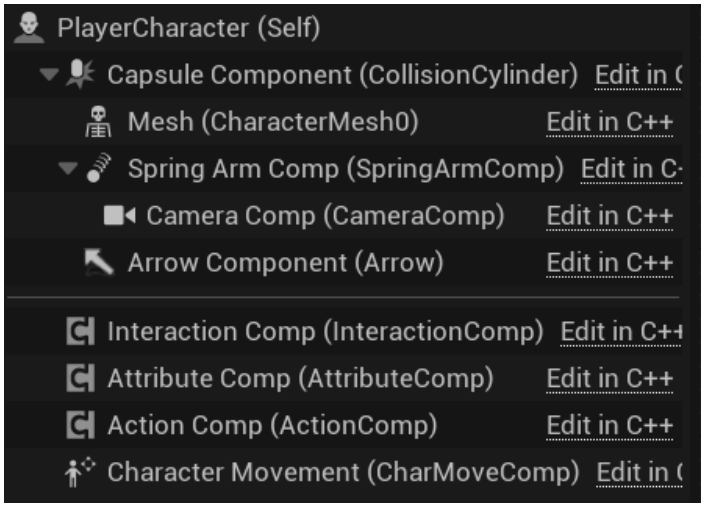


Рисунок 3.2 – Набір компонентів з яких складається PlayerCharacter

Компонент взаємодії займається знаходженням всіх об’єктів, які знаходяться близько перед гравцем і виведенням відповідних елементів інтерфейсу, якщо відповідні об’єкти мають можливість відповідати на взаємодію з гравцем. За цю дію відповідає функція FindBestInteractable(), яка викликається в кожному кадрі, коли камера змінювала положення. Функція виконує зіткнення колізії для знаходження найближчого об'єкта, придатного для взаємодії, оновлює віджет для взаємодії і надає візуалізацію за необхідності.

Введення в Unreal Engine можна приймати за допомогою системи Enhanced Input. Для проєктів Unreal Engine 5 (UE5), які вимагають більш розширених функцій введення, як-от складна обробка вводу або переналаштування керування під час виконання.

Створення відповідного функціоналу складається з добавлення Inpute Action об’єктів, які визначають що відбувається і які параметри приймаються і віддаються та створення контексту, де відповідні дії назначаються на потрібні клавіші, за потреби можуть бути визначені модифікатори. Список реалізованих об’єктів Inpute Action зображений на рисунку А.13 у додатку А.

Створений Inpute Mapping Context назначаються в контролері і може бути за потреби змінений, якщо логіка вводу змінюється в якийсь момент. У застосунку достатньо буде одного контексту.

Для передачі вводу від рушія до персонажу, було вирішено використовувати проміжний клас PlayerController. Це забезпечить модульність персонажів, в будь-який момент є можливість змінити персонажа. В загальному функціонал контролера просто прив’язує функції IMoveCharacter до контексту вводу. Код реалізованого класу можна побачити в додатку Б.

Також потрібно створити скрипт анімації [20], який буде визначати поведінку анімації на основі стану ігрового персонажа. Основна функція анімаційного скрипту, це Update Animation. Вони отримавши об’єкт персонажу, зберігає дані для подальших алгоритмів. Реалізовувався скрипт у Animation Blueprint (рисунок 3.3).

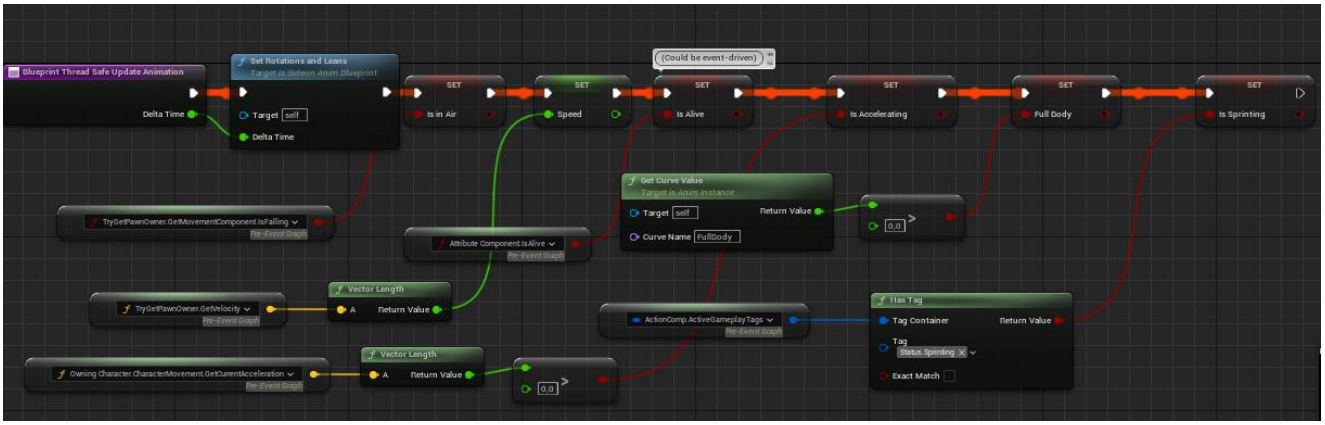


Рисунок 3.3 – Скрипт оновлення даних анімації

Головна функція Animation Blueprint, це генерація анімаційного кадру, пози в які має знаходитись модель персонажу. Для визначення кадру використовують алгоритми на основі машин станів, та функцій змішування позицій.

Першою було створену машину станів (рисунок 3.4) для руху по землі (Ground Locomotion) для переходу між очікування без руху (Idle) та анімацією початку руху (JogStart), бігу (Run) і завершення руху (JogStop).

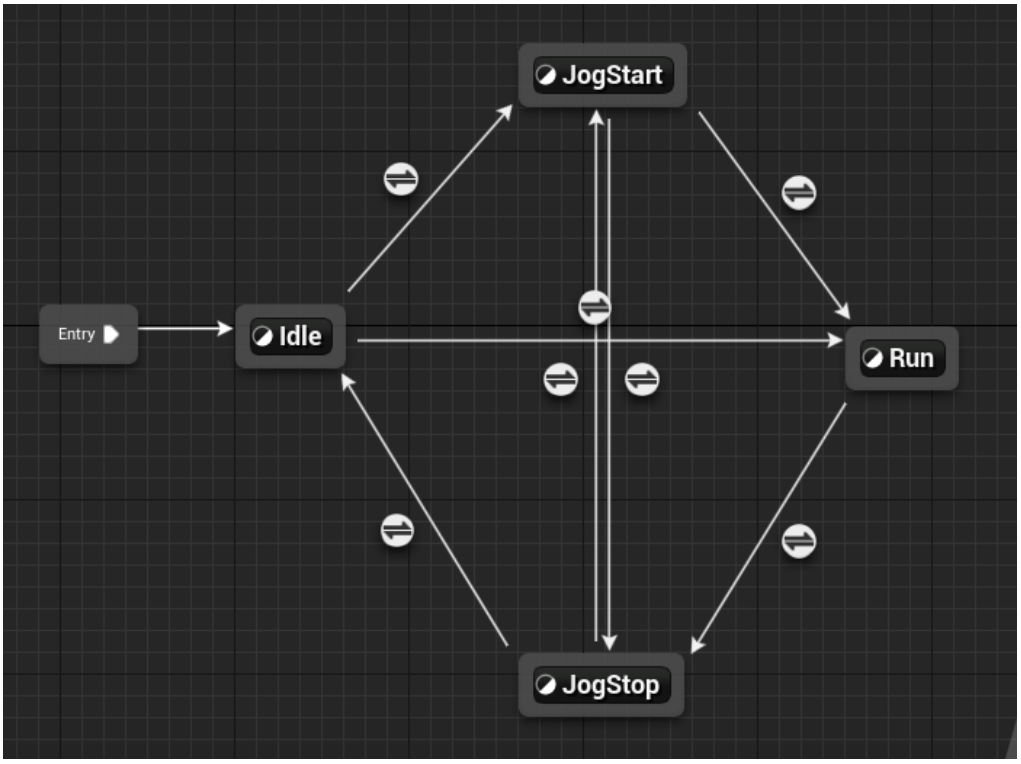


Рисунок 3.4 – Машина станів «Ground Locomotion»

Наступною було реалізовано машину додаткових анімацій переміщення (рисунок 3.5), яка перезаписує поточну анімацію в залежності від того чи стрибає гравець та чи знаходиться в стані. Окремо виділені анімації входу і виходу із станів.

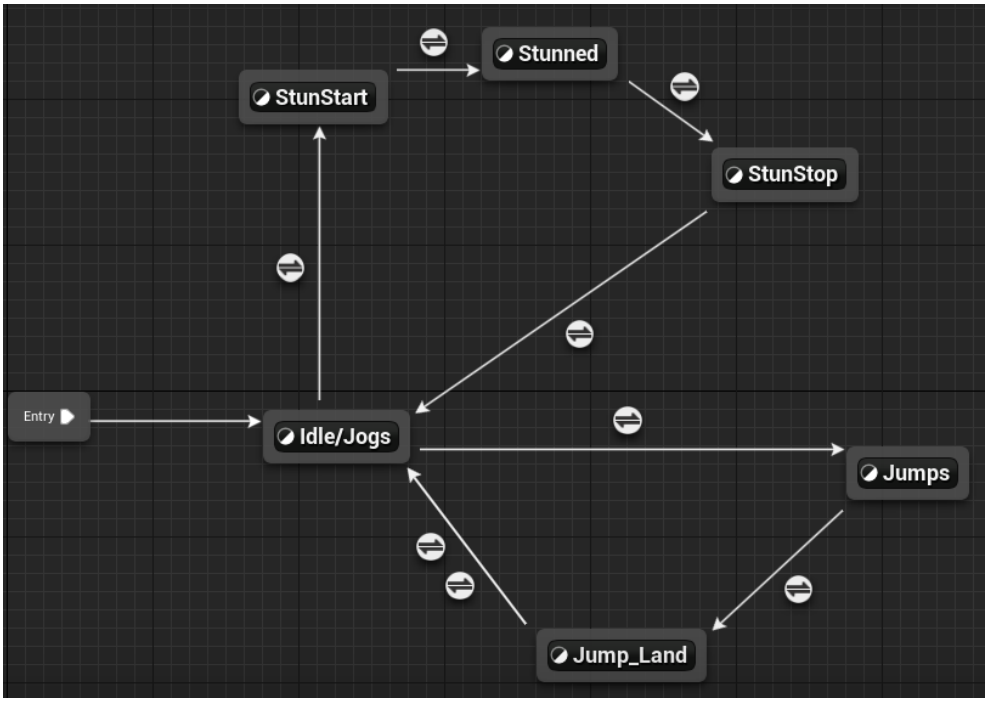


Рисунок 3.5 – Машина станів загального переміщення

До отриманої анімації після цих двох машин станів додається анімація прицілювання (AimOffset) на основі кута нахилу камери.

Після цього отримана анімація може бути змішана з анімаційними монтажами, які потрібно відтворити тільки на верхній частині тіла (рисунок 3.6).

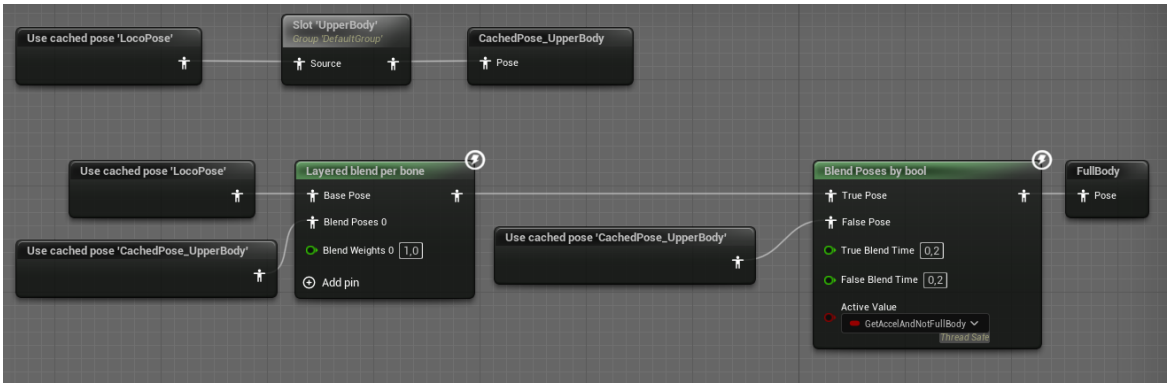


Рисунок 3.6 – Змішування з анімаціями верхньої частини тіла

Наступним етапом було створення противників NPC. Для цього першим етапом було створення класу AIController, який буде керувати поведінкою окремих ворогів. Для створення цього класу було використано клас, який надає Unreal Engine. Цей клас включає специфічний, для роботи з AI, функціонал. Для керування поведінкою NPC в Unreal Engine використовуються поведінкові дерева (Behavior Tree), це спеціальні об’єкти в яких можна запрограмувати алгоритм поведінки NPC, за допомогою спеціальної візуальної скриптової мови. Так, як передбачається один сценарій поведінки ворогів, все що буде робити контроллер це запускати встановлене поведінкове дерево (рисунок 3.7).

```
void ASAIController::BeginPlay()
{
    Super::BeginPlay();

    if (ensureMsgf(BehaviorTree, TEXT("Behavior Tree is nullptr! Please assign BehaviorTree in your AI Controller.")))
    {
        RunBehaviorTree(BehaviorTree);
    }
}
```

Рисунок 3.7 – Код AIController

Далі було створено сам алгоритм behavior tree для ворога (рисунок 3.8). В загальному NPC патрулює, якщо замічає гравця починає наближатись до нього і проводити атаку тричі, також ворожий NPC слідкує за станом свого здоров’я.

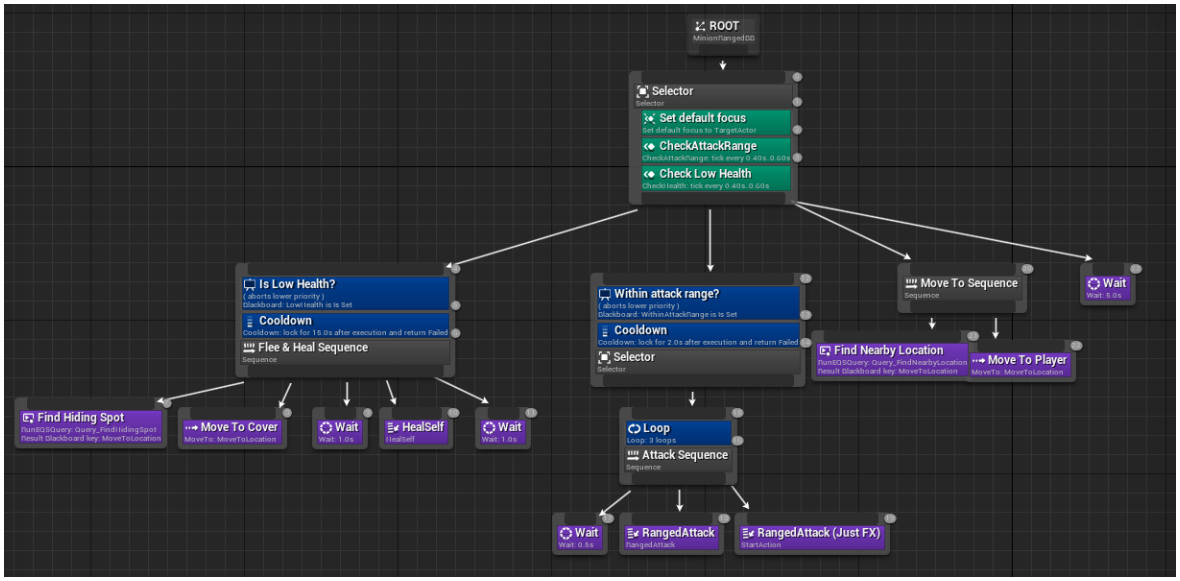


Рисунок 3.8 – Поведінкове дерево ворога

Після створення персонажів потрібно було реалізувати клас ігрового режиму. В першу чергу тут буде проводитись керування елементами інтерфейсу, через цей клас будуть створюватись і приховуватись екрани. Також цей клас буде слідкувати за створенням і знищенням персонажів і приймати відповідні дії, як реакцію на це.

Останнім класом, який потрібно було реалізувати був клас екземпляру гри. Він надає функціонал для збереження і завантаження стану гри.

Повний код програми наведено у додатку Б.

3.3 Реалізація інтерфейсу користувача

В Unreal Engine вбудований редактор для створення елементів інтерфейсу користувача UMG UI Designer. Цей інструмент надає смогу швидко створювати та програмувати весь потрібний інтерфейс, надаючи набір готових загальних елементів. При потребі можна створити потрібні елементи самостійно.

Головним об’єктом інтерфейсу являється віджет. Цей об’єкт може представляти собою, як весь екран так і його елемент. Тому для реалізації інтерфейсу потрібно на основі визначених макетів раніше.

Для реалізації головного ігрового екрану потрібно створити елементи, які відображають здоров’я, досвіду і валюти. Для реалізації цих елементі використовують уже створені елементи тексту та панелей рівня.

Таким ж чином створюються елементи досвіду та валюти. Далі ці елементи об’єднуються у один віджет – ігровий екран. Для цього всі елементи встановлюються на відповідні місця та налаштовується вирівнювання. Також в цьому віджеті створюються функції за допомогою яких можна змінювати відповідні показники при потребі. В результаті був створений віджет, який зображений на рисунку А.14 додатку А.

Далі було реалізовано екран паузи, на якому появляються дві кнопки для продовження гри та переходу в головне меню. Результат на рисунку 3.9.

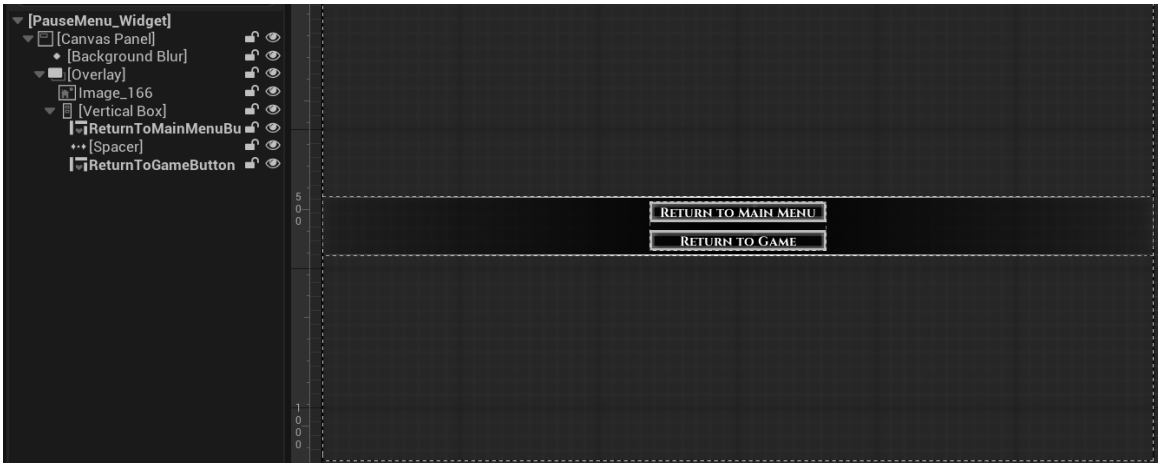


Рисунок 3.9 – Віджет екрану паузи

Також було створено екран головного меню (рисунок 3.10), який складається з кнопок для переходу до ігрового процесу, відкриття меню налаштувань та виходу з ігри. Також для розмиття фону добавляється елемент розмиття.

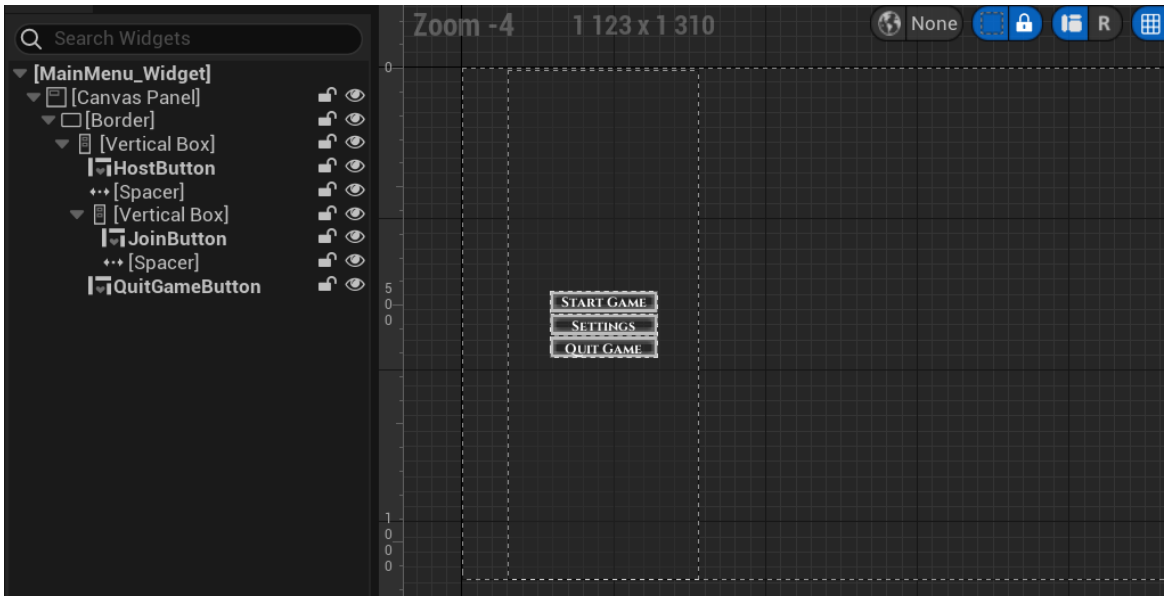


Рисунок 3.10 – Віджет головного меню

Після реалізації всіх елементів інтерфейсу потрібно було в класі ігрового режиму добавити логіку, яка буде створювати елементи інтерфейсу і відображати або приховувати відповідні елементи при запиті.

На цьому реалізація всіх частин ігрового застосунку була завершена.

3.4 Вимоги до технічних та програмних засобів

Визначення системних вимог є важливим етапом. Вони допомагають гарантувати, що гра працюватиме належним чином на обраному обладнанні, надають гравцям чітке уявлення про те, чи зможе їх комп'ютер запустити гру і сприяють кращому плануванню та оптимізації під час розроблення.

Головними характеристиками системи, до яких вставляються вимоги, це тип процесора і операційної системи, продуктивність процесора та кількість оперативної пам’яті, вільний дисковий простір та продуктивність відеоадаптера і наявність підтримки ним потрібних технологій.

Аналізуючи розроблений застосунок було визначено мінімальні та рекомендовані системні вимоги (таблиця 3.1).

Таблиця 3.1 – Системі вимоги застосунку

	Мінімальні	Рекомендовані
Розрядність процесора і ОС	x64	
ОС	Windows 10/11	
Процесор	AMD Ryzen 5 2400G, Intel Core i3-10105F	Intel Core I5-8400, AMD Ryzen 3 3300X
ОЗУ	6 Гб	8 Гб
Відеокарт	Nvidia GeForce GTX 1060 3 Гб, AMD Radeon RX 580 4 GB	NVIDIA GeForce GTX 1070 8 GB, AMD Radeon RX Vega 56 8 GB
DirectX	12	
Місце на диску	1 Гб	

3.5 Тестування ігрового застосунку

3.5.1 Аналіз методів тестування ігрових застосунків

Тестування ігрових застосунків, особливо в жанрі Action-RPG, є важливим етапом, який забезпечує високу якість кінцевого продукту. Використання Unreal Engine для створення ігрових механік вимагає ретельного тестування для виявлення і виправлення потенційних проблем. Тестування гри в Unreal Engine [17] включає кілька методів, кожен з яких має свою специфіку і важливість для загальної стабільності і продуктивності гри і підтримується одразу рушієм.

Модульне тестування (Unit Testing) забезпечує перевірку окремих функцій і компонентів на рівні коду, що дозволяє ізолювати і виправити помилки в ранніх етапах. Це особливо важливо для складних систем, таких як управління персонажами та бойова механіка, де невеликі помилки можуть призвести до серйозних проблем у геймплеї.

Інтеграційне тестування (Integration Testing) дозволяє переконатися, що всі компоненти гри правильно взаємодіють між собою, що особливо важливо для великих проєктів з численними підсистемами.

Для забезпечення оптимальної продуктивності та плавного геймплею використовуються методи тестування продуктивності (Performance Testing). За допомогою таких інструментів, як Unreal Insights, розробники аналізують використання ресурсів і оптимізують гру для різних апаратних конфігурацій.

Автоматизоване тестування (Automated Testing) значно прискорює процес перевірки шляхом запуску набору тестів при кожному збиранні проєкту, що дозволяє виявляти проблеми на ранніх етапах.

Ігрове тестування (Playtesting) залучає реальних гравців для оцінки ігрового процесу і отримання зворотного зв'язку, що допомагає виявити проблеми, які могли залишитися непоміченими під час внутрішнього тестування.

Аналізуючи масштаб, вимоги до застосунку та складність модулів було прийнято рішення використати три головні методи для тестування мого ігрового

застосунку: модульне тестування важливих класів і функцій, тестування продуктивності, а також у фіналі ігрове тестування. Цей набір тестів забезпечить достатньо високу якість фінального проєкту та не буде перенавантажувати проєкт.

3.5.2 Аналіз результатів тестування

Першим етапом тестування було написання модульних тестів для критично важливих класів ігрового застосунку. Важливо забезпечити постійність роботи модулів, які змінюють стан гри, особливо стан персонажів.

Було вирішено, що створювати модульні тести потрібно для таких класів:

- AttributeComponent;
- HealthPotion;
- Projectile.

Тести компоненту атрибутів верифікують коректність зміни здоров'я персонажа, провіряючи чи не було порушено алгоритм роботи компоненту. Для цього класу було написано пять тестів (рисунок 3.11).

Рисунок 3.11 – Результат unit-тестів для компоненту атрибутів

Для інтерактивного компоненту лікувального зілля було створено unit-тести створення предмету і використання. Для базового класу магічного снаряду написані тести для перевірки його створення та зіткнення з персонажем. Результати виконання тестів для цих класів зображені на рисунку 3.12.

Рисунок 3.12 – Результати unit-тестів для лікувального засобу і снаряду

Після завершення реалізації unit-тестів потрібно було провести тестування продуктивності ігрового застосунку. Для цього на платформі характеристики якої відповідають мінімальним було запущено систему профілювання Unreal Insight [21], яка записує точно використання ресурсів комп'ютера для кожного кадру та дозволяє зручно аналізувати ці графіки. Під час запису відтворюються найбільш навантажені сценарії. Для проведення тестування було записано сценарій з боєм проти трьох гравців. Графік часу кадру на рисунку 3.13.

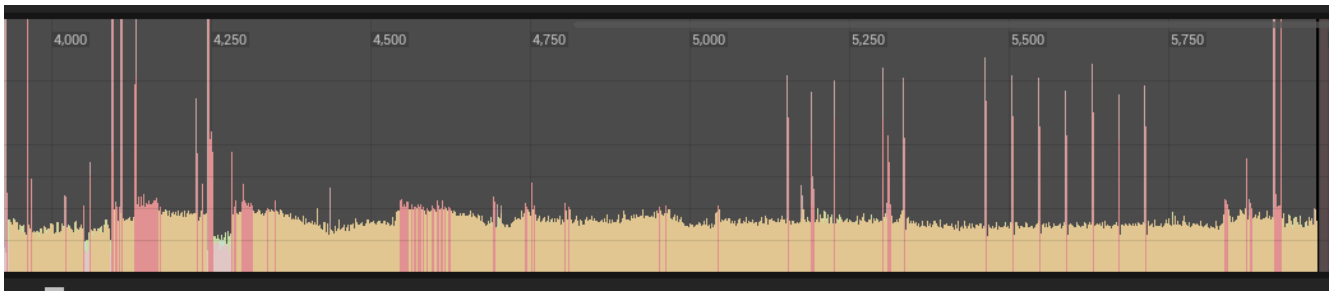


Рисунок 3.13 – Графік часу кадру запису профілювання

Аналізуючи графік можна визначити що середній час кадру 70-80 мс, що відповідає 30-40 FPS (frame per second). Цей показник відповідає вимогам для продуктивності на мінімальних системних вимогах. Також на графіку можна замітити невеликі ділянки з падінням продуктивності до 15 FPS. Для того, щоб визначити, що викликає пониження продуктивності можна розглянути детально, що в цих кадрах займає час. Особливо важливо визначити чи немає пониження продуктивності саме в ігровому потоці, так як весь функціонал, який був розроблено працює в цьому потоці. Також можна розглянути потік рендерингу.

Аналізуючи графік часу кадру (рисунок 3.14), було виявлено, що зменшення продуктивності пов'язане тільки з рендером інтерфейсу користувача і пов'язане з специфікою роботи рушія, тому оптимізоване не може бути. Аналіз інших кадрів також не виявив потенційних проблем продуктивності, які б можна було вирішити не погіршуючи або зменшуючи проєкт і його візуальну частину.

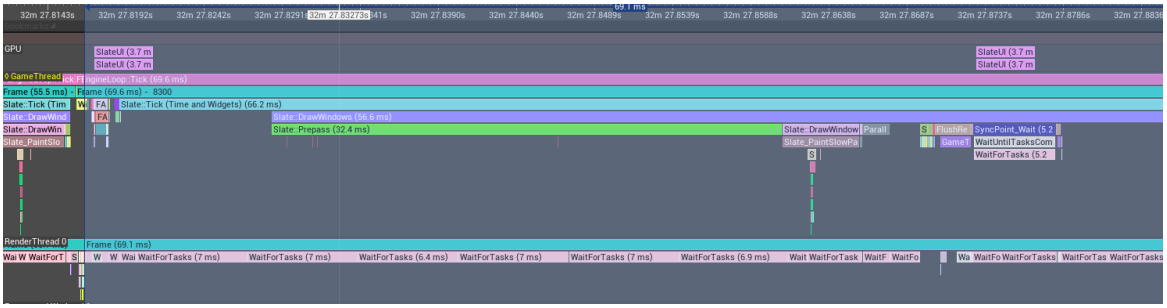


Рисунок 3.14 – Графік роботи потоків для навантаженого кадру

Останнім етапом тестування було проведення ручного ігрового тестування. Це тестування проводилось за допомогою відтворення типових ігрових сценаріїв та на основі емпіричного досвіду визначалась оцінка якості ігрового процесу.

Було протестовано такі сценарії:

- переміщення по рівню;
- використання здібностей;
- бій з ворогами (рисунок 3.15);
- взаємодія з інтерактивними об’єктами.

Рисунок 3.15 – Процес тестування бою

Ігрове тестування визначило, що досвід, який надає застосунок відповідає всі визначеним раніше вимогам і не виявило ніяких проблем пов’язаних з роботою розробленого ігрового застосунка. На основі цього етапу було створено керівництво користувача, яке знаходиться в додатку В.

3.6 Висновки

Для реалізації ігрового застосунку було проведено багато комплексних етапів роботи, які базувались на результатах проєктування системи.

Першим важливим етапом початку реалізації застосунку було ознайомлення с ігрових рушієм, його архітектурою, інструментарієм, документацією і важливими підходами до розроблення для обраного рушія. Вивчення цього питання дозволило ефективно проводити реалізацію функціоналу в подальшому.

Після вивчення рушія було реалізовано спроєктовані раніше модулі, використовуючи мову C++ та Blueprint. На C++ реалізовувались базові та класи функціонал яких бажано приховувати для зручності роботи команди. В Blueprint класів головна функція була в надані зручного інтерфейсу для у рушії.

Окремим етапом роботи було створення скриптів анімацій та поведінкових дерев за допомогою візуальних редакторів рушія.

Створення елементів інтерфейсу за допомогою UMG редактора та програмування його функціоналу було завершальною роботою.

Разом з створення модулів відбувалось створення unit-тестів для класів, що дало змогу перевірити якість модулів після завершення всіх складових. Також для верифікації якості застосунку було проведено тестування продуктивності та ігрове тестування, які визначили достатню якість проєкту.

Результатом цього розділу став завершений та протестований ігровий застосунок у жанрі «Actio-RPG».

ВИСНОВКИ

Завершивши виконання кваліфікаційної роботи на тему «Ігровий застосунок у жанрі «Action-RPG», можна зробити такі висновки.

Виконання розпочалось з дослідження предметної області та постановки задач. Спочатку було визначено рівень актуальності теми, проаналізовано, з яких етапів складається створення ігрового застосунку в загальному і у цьому жанрі та досліджено існуючі ігрові застосунки цього жанру. Це показало високу актуальність теми, пов’язану з ростом глобального ринку, надало можливість розглянути процеси, які типові для області до якої належить робота та дозволило краще розуміти критично важливі завдання, які стоять перед застосунком цього жанру.

В результаті виконання аналізу предметної області було сформовано вимоги до застосунку, побудовано діаграму варіантів використання та поставлено задачі для проєктування і реалізації.

Після визначення вимог і задач було проведено проєктування ігрового застосунку. Першим важливим етапом був розгляд архітектурних підходів, які часто використовують під час реалізації ігрових застосунків на різних рівнях системи. На основі аналізу було обрано поєднання компонентного і подійно-орієнтованого підходу, що забезпечило модульність і гнучкість системи.

На основі обраного архітектурного підходу було спроєктовано модулі для ігрового застосунку та побудовано низку діаграм:

- діаграма міжмодульних зв’язків;
- діаграми послідовності;
- діаграми станів;
- діаграми діяльності;
- діаграма класів.

Після цього було визначено вимоги для інтерфейсу користувача та створено макети всіх потрібних екранів.

Наявність цих діаграм та макетів надала можливість краще розуміти

композицію системи та ефективно почати реалізацію системи.

В кінці проєктування було обрано технології та методи реалізації. Як рушій було обрано Unreal Engine 5.4, через його високу гнучкість, продуктивність та підтримку. Як середовище розроблення було вибрано Microsoft Visual Studio 2022 по причині його високої інтеграції з Unreal Engine. Як метод було обрано ітеративний, через високу потребу у швидкому впроваджені змін у проєкті.

Після завершення проєктування перед початком реалізації було розглянуто особливості процесу реалізації ігрових застосунків за допомогою Unreal Engine. Створення C++ класів системи на основі фреймворку рушія було першим етапом реалізації. Після цього було створено Blueprint класи обгортки, які дали можливість гнучко взаємодіяти з об’єктами класів в рушії. Використовуючи інструментарій рушія було реалізовано скрипти анімації та екрани ігрового інтерфейсу та об’єднано з реалізованими програмними модулями.

Останнім етапом було проведення тестування системи. Було визначено, що основними методами тестування для ігрових застосунків є unit-тести, ігрове тестування та тестування продуктивності через їх високу ефективність, по цій причині ці методи були обрані для тестування мого ігрового застосунку. Результати тестування визначили достатню якість розробленого застосунку.

Отже, можна стверджувати, що мета кваліфікаційної роботи, яка полягала у створені ігрового застосунку в жанрі Action-RPG, була досягнута. Ігровий застосунок успішно розроблений, пройшов тестування, підтвердивши свою відповідність встановленим технічним вимогам і методикам ігрової індустрії. Застосунок забезпечує відповідний ігровий процес, високий рівень графіки та стабільну продуктивність. Таким чином, розроблений ігровий застосунок відповідає сучасним стандартам та вимогам індустрії ігрових розробок. Результати кваліфікаційної роботи підтверджують її успішне завершення і відповідність поставленій меті та завданням.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. GamesIndustry.biz presents... The Year In Numbers 2023. URL: https://www.gamesindustry.biz/gamesindustrybiz-presents-the-year-in-number-2023 (дата звернення: 18.02.2024).
2. Video game industry – Statistics & Facts. URL: https://www.statista.com/topics/868/video-games/#topicOverview (дата звернення: (18.02.2024).
3. Role-Playing Games (RPGs) Revealed | Newzoo Game Genre Report. URL: https://newzoo.com/resources/trend-reports/role-playing-games-rpg-revealed-newzoo-game-genre-report (дата звернення: 19.02.2024).
4. Steam. Хіти продажу Action-RPG. URL: https://store.steampowered.com/category/rpg_action/?flavor=contenthub_topsellers (дата звернення: 20.02.2023).
5. Steam. Найвищі оцінки Action-RPG. URL: https://store.steampowered.com/category/action/?tab=3&facets13268=8%3A26%2C9%3A2&flavor=contenthub_toprated (дата звернення: 20.02.2024).
6. Steam. Сторінка God of War. URL: https://store.steampowered.com/app/1593500/God_of_War (дата звернення: 20.02.23).
7. Steam. Сторінка NieR: Automata. URL: https://store.steampowered.com/app/524220/NieRAutomata/ (дата звернення: 20.02.24).
8. Steam. Сторінка ELDEN RING. URL: https://store.steampowered.com/app/1245620/ELDEN_RING/ (дата звернення: 20.02.24).
9. The Evolution of Action Role-Playing Games: From Humble Beginnings to Modern Innovations. URL: https://medium.com/@johnthekrakentv/the-evolution-of-action-role-playing-games-from-humble-beginnings-to-modern-innovations-8b8da78f1bdc (дата звернення: 19.02.24).

10. How FromSoft took the action-RPG to a new level by putting faith in Dark Souls' players. URL: https://www.gamesradar.com/how-fromsoft-took-the-action-rpg-to-a-new-level-byputting-faith-in-dark-souls-players/ (дата звернення: 20.02.2024).

11. What is Use Case Diagram? URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-usecase-diagram/ (дата звернення: 19.02.2024).

12. THE SEVEN STAGES OF GAME DEVELOPMENT. URL: https://gamemaker.io/ru/blog/stages-of-game-development (дата звернення: 20.02.2024).

13. Unreal Engine 5.3 Documentation. URL: https://docs.unrealengine.com/5.3/en-US/ (дата звернення: 20.02.2024).

14. What is an Action RPG? URL: https://howtomakeanrpg.com/a/what-is-an-action-rpg.html (дата звернення: 20.02.2024).

15. Game Development Technologies and Trends for 2023. URL: https://www.chetu.com/blogs/gaming/game-development-trends.php (дата звернення: 19.02.2024).

16. Gregory J. Game Engine Architecture, Third Edition USA: CRC Press, 2018. 68 с.

17. Creating Functional Tests with the Automation System. URL. https://www.orfeasel.com/functional-tests/ (дата звернення: 04.04.2024).

18. What Is Iterative Game Design? URL. https://www.gamedesigning.org/learn/iterative/ (дата звернення: 15.03.2024).

19. Gameplay Framework. URL. https://dev.epicgames.com/documentation/en-us/unreal-engine/gameplay-framework-in-unreal-engine (дата звернення: 05.03.2024).

20. Animation Blueprints. URL. https://dev.epicgames.com/documentation/en-us/unreal-engine/animation-blueprints-in-unreal-engine (дата звернення: 08.03.2024).

21. Unreal Insights. URL. https://dev.epicgames.com/documentation/ru-ru/unreal-engine/unreal-insights-in-unreal-engine (дата звернення: 08.03.2024).

22. Joshi U. Patterns of Distributed Systems/ Unmesh Joshi. India: Addison-

Wesley Professional, 2023. 34 p.

23. Richards M. Fundamentals of Software Architecture: An Engineering Approach/ Mark Richards, Neal Ford. USA: O'Reilly Media, 2020. 59 p.

24. Khononov V. Learning Domain-Driven Design: Aligning Software Architecture / Vlad Khononov. USA: O'Reilly Media, 2021. 78 p.

25. Brown S. Software Architecture for Developers/ Simon Brown. USA: Leanpub, 2022. 75 p.

26. Shrivastava S. Solutions Architect's Handbook: Kick-start your solutions architect career by learning architecture design principles and strategies/ Saurabh Shrivastava. USA: Packt Publishing, 2020. 331 p.

27. Ford N. Software Architecture: The Hard Parts/ Neal Ford, Mark Richards, Pramod Sadalage. USA: O'Reilly Media, 2021. 129 p.

28. What is Sequence Diagram? URL. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/ (дата звернення: 10.04.2024).

29. What is State Machine Diagram? URL. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-state-machine-diagram/ (дата звернення: 10.04.2024).

30. What is Class Diagram? URL. https://visual-paradigm.com/what-is-class-diagram/(дата звернення: 10.04.2024).

ДОДАТОК А

(обов’язковий)

ГРАФІЧНІ МАТЕРІАЛИ

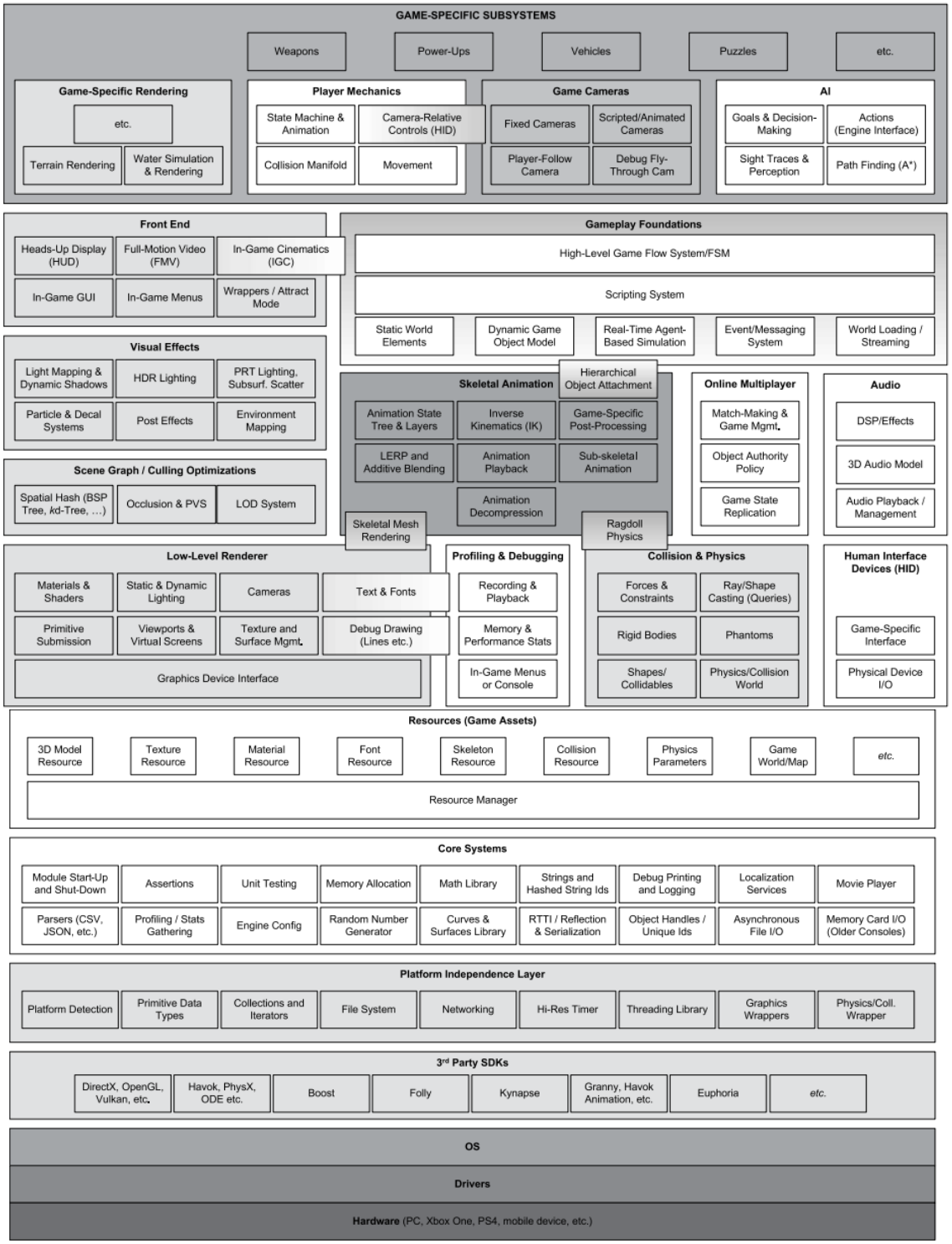


Рисунок А.1 – Типова реалізація модульної архітектури ігрового рушія

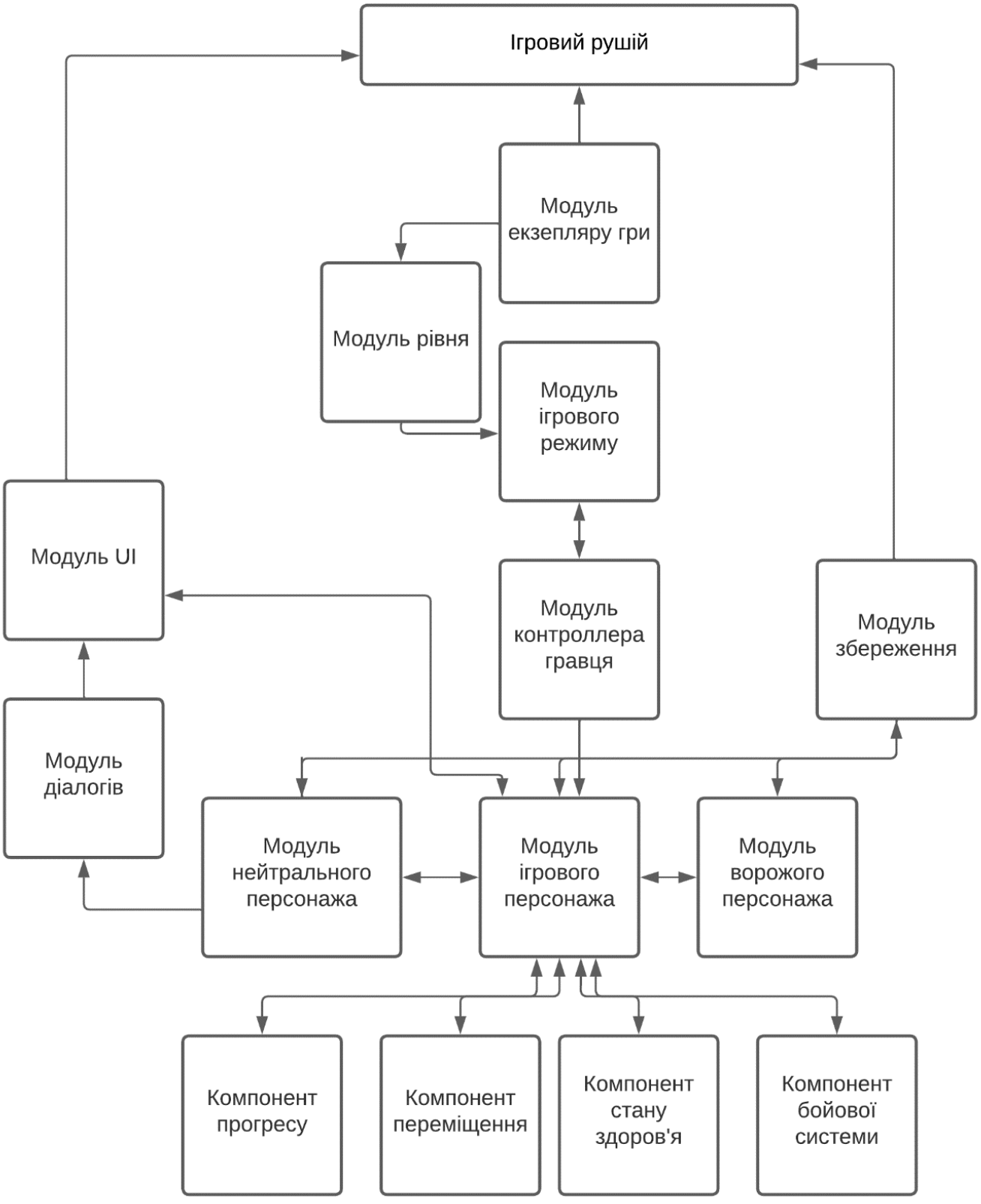


Рисунок А.2 – Діаграма міжмодульних зв’язків

Рисунок А.3 – Макет ігрового екрану

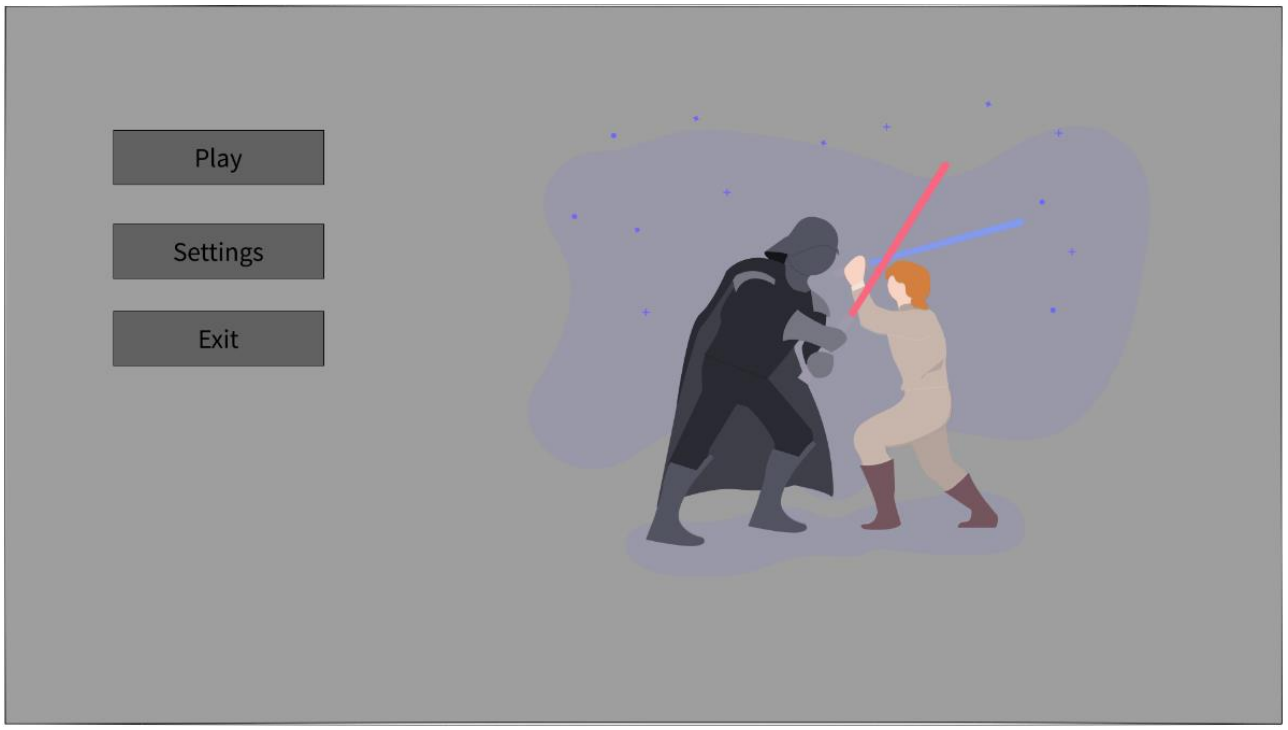


Рисунок А.4 – Макет головного меню

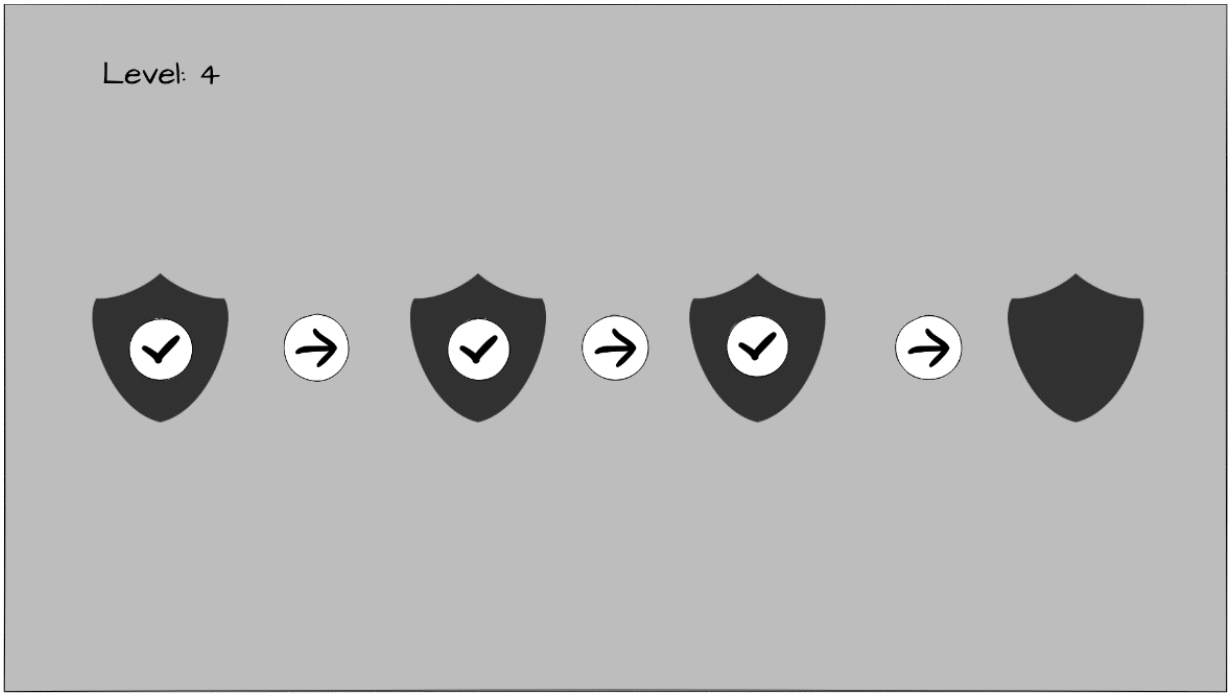


Рисунок А.5 – Макет екрану здібностей

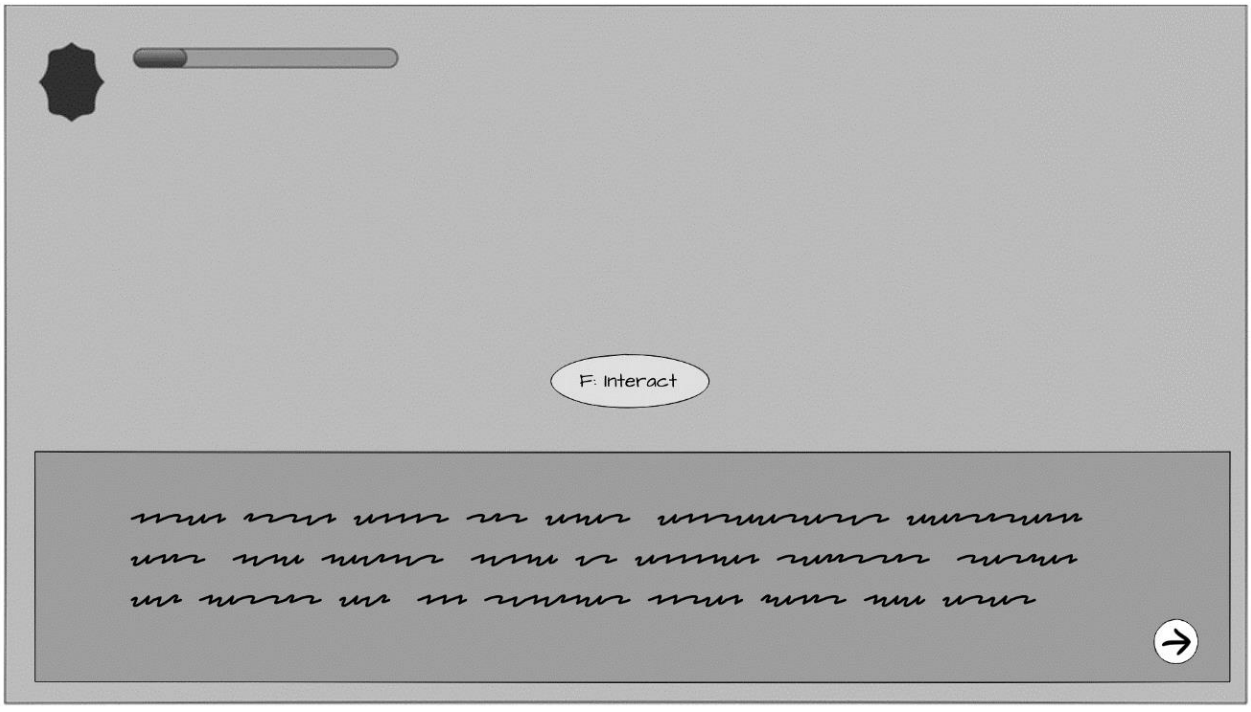


Рисунок А.6 – Макет екрану ігрового інтерфейсу

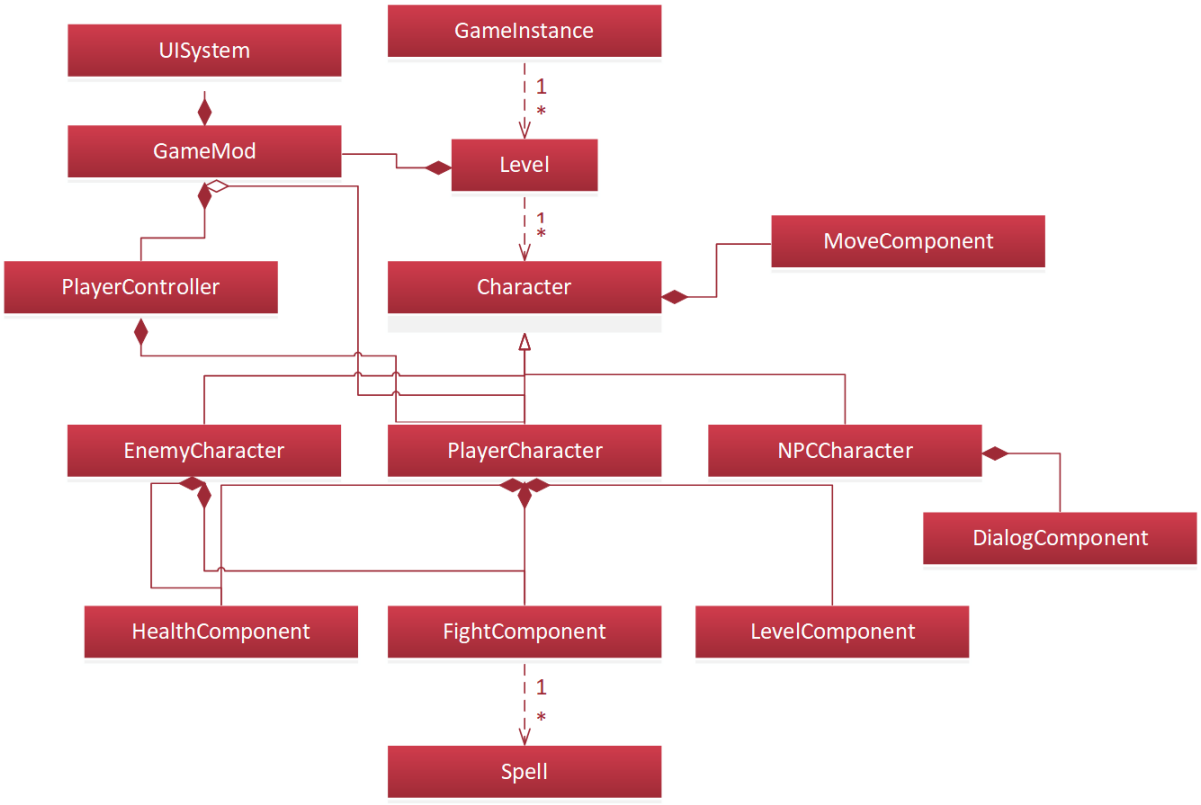


Рисунок А.7 – Діаграма об’єктів

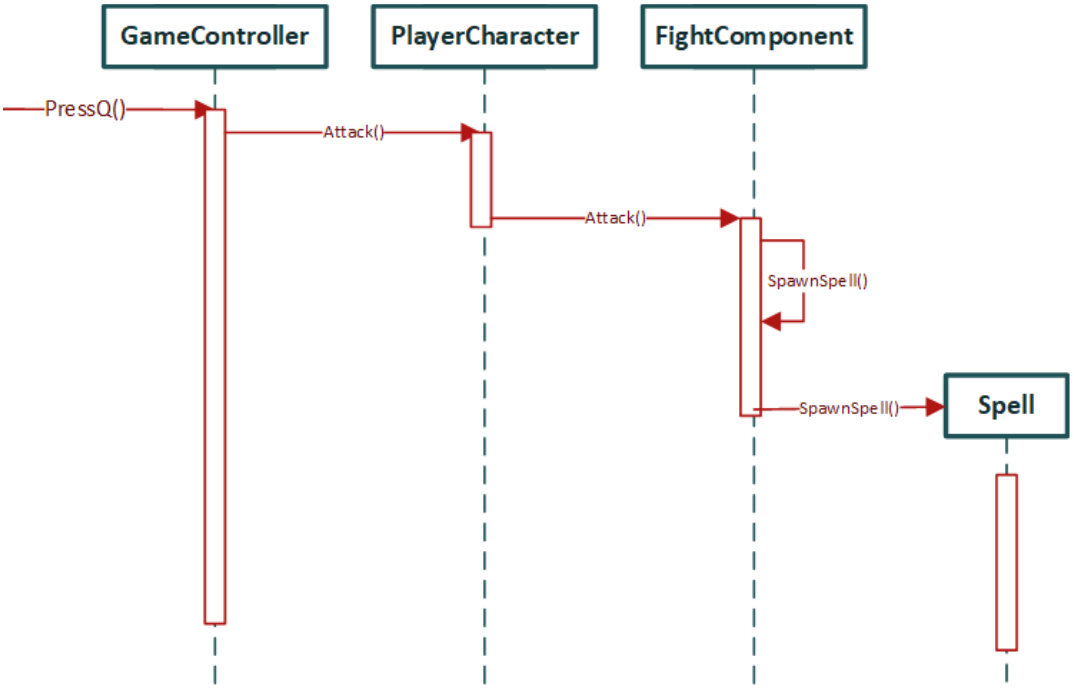


Рисунок А.8 – Діаграма потоків для атаки здібністю

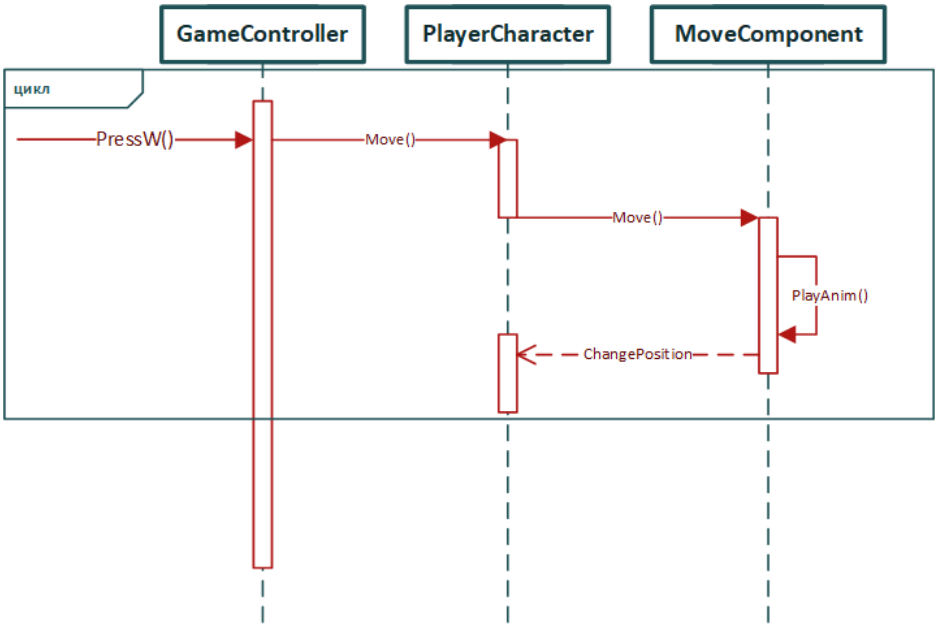


Рисунок А.9 – Діаграма потоків для переміщення вперед

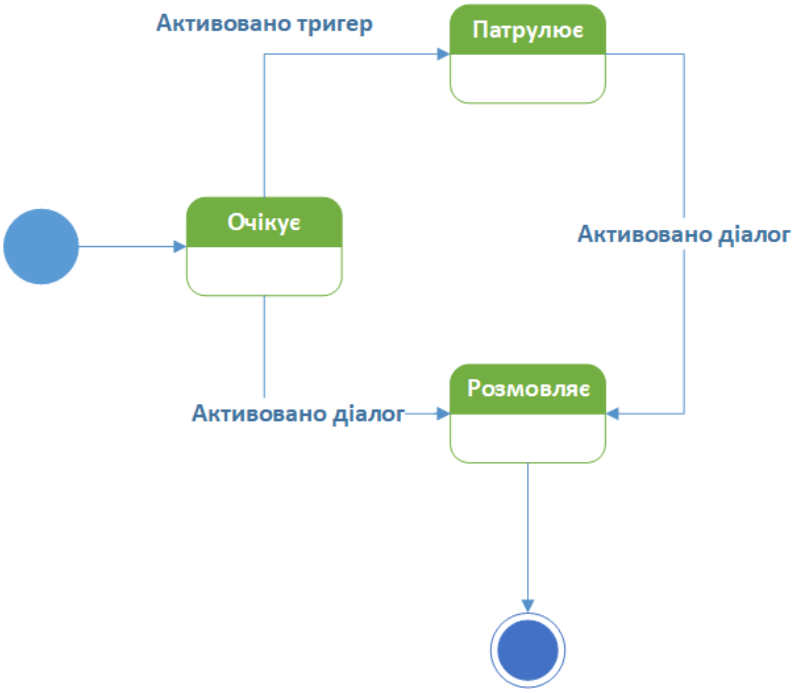


Рисунок А.10 – Діаграма станів логіки ворога

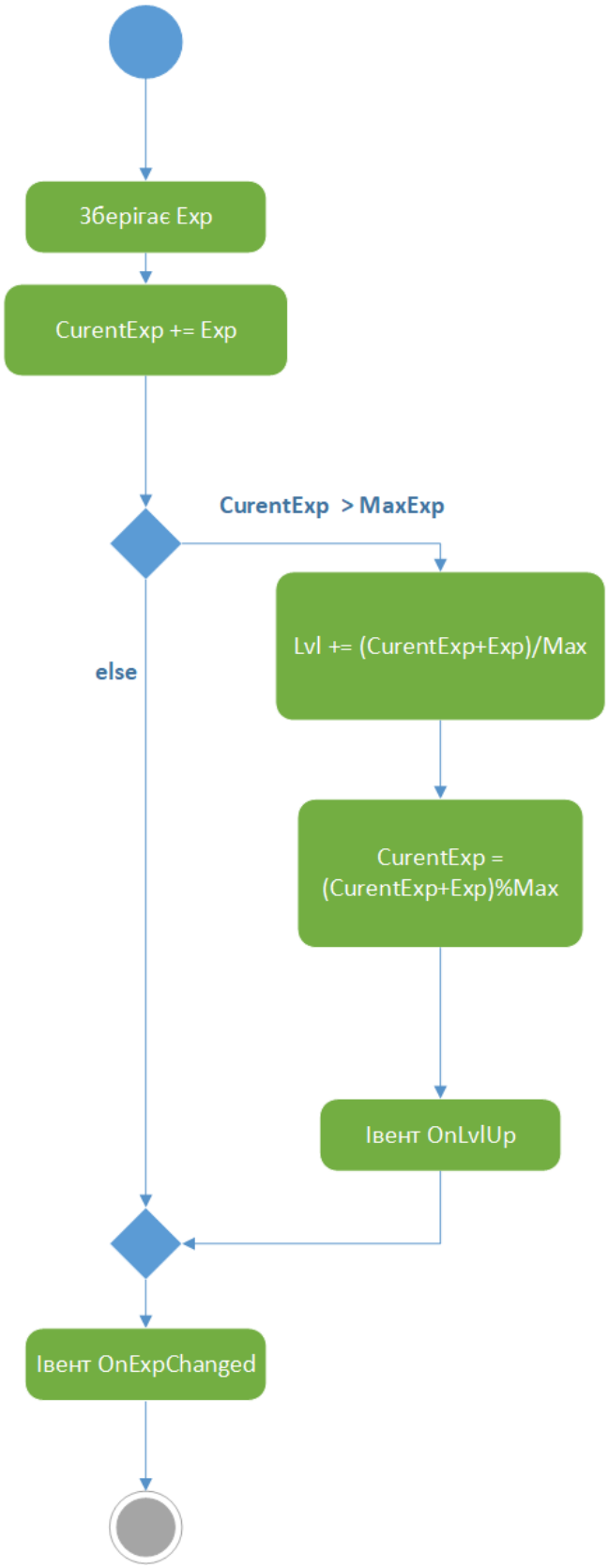


Рисунок А.11 – Діаграма активності для зміни досвіду

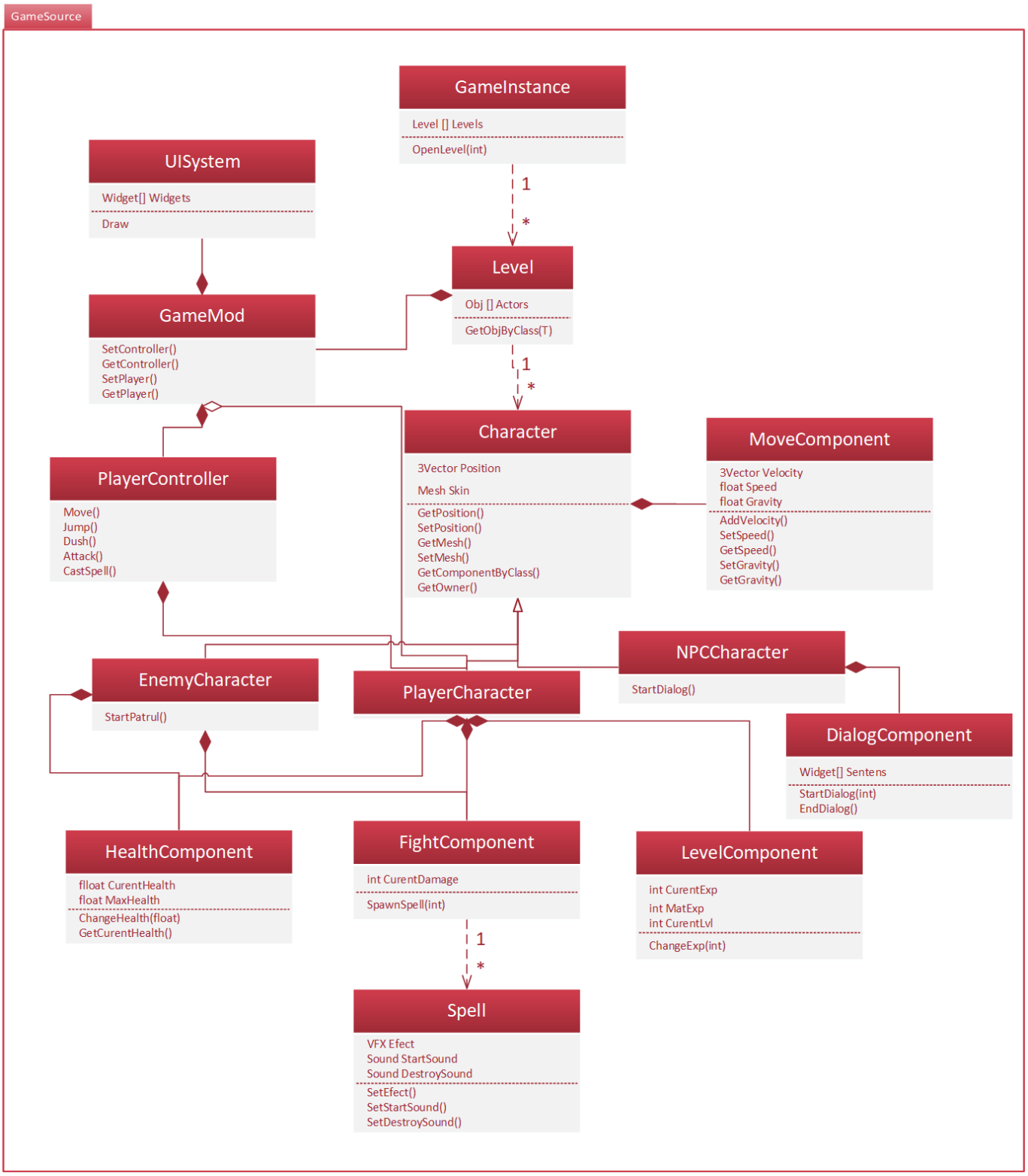


Рисунок А.12 – Діаграма класів

Рисунок А.13 – Набір реалізованих inpute action

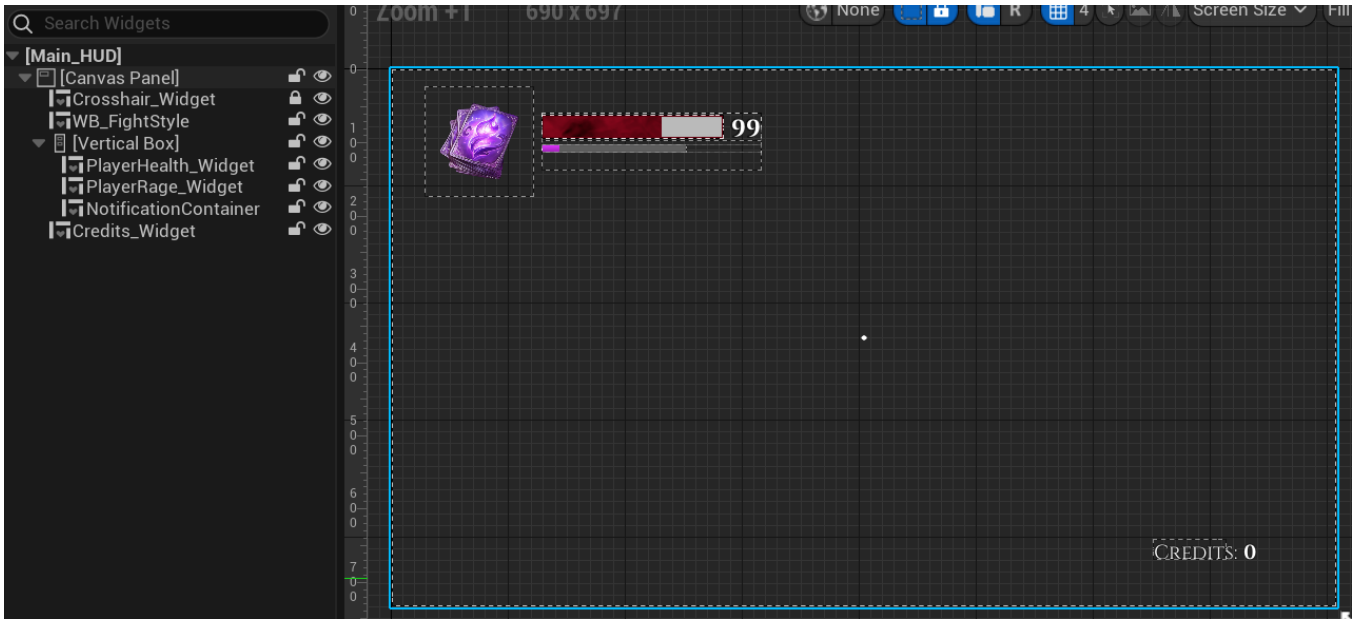


Рисунок А.14 – Реалізація ігрового екрану

ДОДАТОК Б

(обов’язковий)

ПРОГРАМНИЙ КОД ОСНОВНИХ МОДУЛІВ

Б.1 Код персонажа SCharacter

```
#include "SCharacter.h"
#include "GameFramework/SpringArmComponent.h"
#include "Camera/CameraComponent.h"
#include "DrawDebugHelpers.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "SInteractionComponent.h"
#include "SAttributeComponent.h"
#include "SActionComponent.h"
#include "Components/CapsuleComponent.h"
#include "ActionRoguelike.h"
#include "Logging/StructuredLog.h"
#include "SharedGameplayTags.h"
// Enhanced Input
#include "EnhancedInputComponent.h"
#include "EnhancedInputSubsystems.h"
#include "SPlayerController.h"
#include UE_INLINE_GENERATED_CPP_BY_NAME(SCharacter)
// Sets default values
ASCharacter::ASCharacter()
{
	PrimaryActorTick.bCanEverTick = true;

	SpringArmComp = CreateDefaultSubobject<USpringArmComponent>("SpringArmComp");
	SpringArmComp->bUsePawnControlRotation = true;
	SpringArmComp->SetupAttachment(RootComponent);
	SpringArmComp->SetUsingAbsoluteRotation(true);

	CameraComp = CreateDefaultSubobject<UCameraComponent>("CameraComp");
	CameraComp->SetupAttachment(SpringArmComp);

	InteractionComp                                                    =
CreateDefaultSubobject<USInteractionComponent>("InteractionComp");

	AttributeComp                                                      =
CreateDefaultSubobject<USAttributeComponent>("AttributeComp");

	ActionComp = CreateDefaultSubobject<USActionComponent>("ActionComp");

	GetCharacterMovement()->bOrientRotationToMovement = true;
	bUseControllerRotationYaw = false;

	// Skip performing overlap queries on the Physics Asset after animation (7
queries in case of our Gideon mesh)
	GetMesh()->bUpdateOverlapsOnAnimationFinalize = false;

	// Enabled on mesh to react to incoming projectiles
```

```
	GetMesh()->SetGenerateOverlapEvents(true);

	GetCapsuleComponent()->SetGenerateOverlapEvents(false);

	//TimeToHitParamName = "TimeToHit";
	HitFlash_CustomPrimitiveIndex = 0;
}
void ASCharacter::PostInitializeComponents()
{
	Super::PostInitializeComponents();

	AttributeComp->OnHealthChanged.AddDynamic(this,
&ASCharacter::OnHealthChanged);
}

// Called to bind functionality to input
void ASCharacter::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
	Super::SetupPlayerInputComponent(PlayerInputComponent);

	const APlayerController* PC = GetController<APlayerController>();
	const ULocalPlayer* LP = PC->GetLocalPlayer();

	UEnhancedInputLocalPlayerSubsystem*	Subsystem	=	LP-
>GetSubsystem<UEnhancedInputLocalPlayerSubsystem>();
	check(Subsystem);

	Subsystem->ClearAllMappings();

	// Add mappings for our game, more complex games may have multiple Contexts
that are added/removed at runtime
	Subsystem->AddMappingContext(DefaultInputMapping, 0);

	// New Enhanced Input system
	UEnhancedInputComponent*	InputComp	=
CastChecked<UEnhancedInputComponent>(PlayerInputComponent);

	// General
	InputComp->BindAction(Input_Move,	ETriggerEvent::Triggered,	this,
&ASCharacter::Move);
	InputComp->BindAction(Input_Jump,	ETriggerEvent::Triggered,	this,
&ACharacter::Jump);
	InputComp->BindAction(Input_Interact,	ETriggerEvent::Triggered,	this,
&ASCharacter::PrimaryInteract);

	// Sprint while key is held
	InputComp->BindAction(Input_Sprint,	ETriggerEvent::Started,	this,
&ASCharacter::SprintStart);
	InputComp->BindAction(Input_Sprint,	ETriggerEvent::Completed,	this,
&ASCharacter::SprintStop);

	// MKB
	InputComp->BindAction(Input_LookMouse,	ETriggerEvent::Triggered,	this,
&ASCharacter::LookMouse);
	// Gamepad
	InputComp->BindAction(Input_LookStick,	ETriggerEvent::Triggered,	this,
&ASCharacter::LookStick);

	// Abilities
	InputComp->BindAction(Input_PrimaryAttack,	ETriggerEvent::Triggered,	this,
&ASCharacter::PrimaryAttack);
```

```
	InputComp->BindAction(Input_SecondaryAttack, ETriggerEvent::Triggered, this, &ASCharacter::BlackHoleAttack);
	InputComp->BindAction(Input_Dash, ETriggerEvent::Triggered, this, &ASCharacter::Dash);

	// Special
	InputComp->BindAction(Input_PrimaryAttack, ETriggerEvent::Triggered, this, &ASCharacter::PrimaryAttack);
}

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

	FindCrosshairTarget();
}
void ASCharacter::FindCrosshairTarget()
{
	// Ignore if not using GamePad
	ASPlayerController* PC = GetController<ASPlayerController>();

	if (PC == nullptr || !PC->IsUsingGamepad())
	{
		bHasPawnTarget = false;
		return;
	}

	FVector EyeLocation;
	FRotator EyeRotation;
	GetActorEyesViewPoint(EyeLocation, EyeRotation);

	const float AimAssistDistance = 5000.f;
	const FVector TraceEnd = EyeLocation + (EyeRotation.Vector() * AimAssistDistance);

	FCollisionQueryParams Params;
	Params.AddIgnoredActor(this);

	FCollisionShape Shape;
	Shape.SetSphere(50.f);

	// Called next frame when the trace has completed
	FTraceDelegate Delegate = FTraceDelegate::CreateUObject(this, &ASCharacter::CrosshairTraceComplete);

	TraceHandle = GetWorld()->AsyncSweepByChannel(EAsyncTraceType::Single, EyeLocation, TraceEnd, FQuat::Identity, ECC_Pawn, Shape, Params, FCollisionResponseParams::DefaultResponseParam, &Delegate);
}
void ASCharacter::CrosshairTraceComplete(const FTraceHandle& InTraceHandle, FTraceDatum& InTraceDatum)
{
	// at most expect one hit
	if (InTraceDatum.OutHits.IsValidIndex(0))
	{
		FHitResult Hit = InTraceDatum.OutHits[0];
		// Figure out if dealing with a Pawn, may want aim assist on other 'things', which requires a different check
		bHasPawnTarget = Hit.IsValidBlockingHit() && Hit.GetActor()->IsA(APawn::StaticClass());
```

```
	}
}


void ASCharacter::Move(const FInputActionInstance& Instance)
{
	FRotator ControlRot = GetControlRotation();
	ControlRot.Pitch = 0.0f;
	ControlRot.Roll = 0.0f;
	// Get value from input (combined value from WASD keys or single Gamepad stick)
and convert to Vector (x,y)
	const FVector2D AxisValue = Instance.GetValue().Get<FVector2D>();
	// Move forward/back
	AddMovementInput(ControlRot.Vector(), AxisValue.Y);
	// Move Right/Left strafe
	const FVector RightVector =
FRotationMatrix(ControlRot).GetScaledAxis(EAxis::Y);
	AddMovementInput(RightVector, AxisValue.X);

}

void ASCharacter::LookMouse(const FInputActionValue& InputValue)
{
	const FVector2D Value = InputValue.Get<FVector2D>();

	AddControllerYawInput(Value.X);
	AddControllerPitchInput(Value.Y);
}
void ASCharacter::LookStick(const FInputActionValue& InputValue)
{
	FVector2D Value = InputValue.Get<FVector2D>();
	bool XNegative = Value.X < 0.f;
	bool YNegative = Value.Y < 0.f;
	static const float LookYawRate = 100.0f;
	static const float LookPitchRate = 50.0f;
	Value = Value * Value;
	if (XNegative)
	{
		Value.X *= -1.f;
	}
	if (YNegative)
	{
		Value.Y *= -1.f;
	}
	float RateMultiplier = 1.0f;
	if (bHasPawnTarget)
	{
		RateMultiplier = 0.5f;
	}
	AddControllerYawInput(Value.X * (LookYawRate * RateMultiplier) * GetWorld()-
>GetDeltaSeconds());
	AddControllerPitchInput(Value.Y * (LookPitchRate * RateMultiplier) *
GetWorld()->GetDeltaSeconds());
}
void ASCharacter::SprintStart()
{
	ActionComp->StartActionByName(this, SharedGameplayTags::Action_Sprint);
}
void ASCharacter::SprintStop()
{
```

```
	ActionComp->StopActionByName(this, SharedGameplayTags::Action_Sprint);
}
void ASCharacter::PrimaryAttack()
{
	ActionComp->StartActionByName(this,
SharedGameplayTags::Action_PrimaryAttack);
}
void ASCharacter::BlackHoleAttack()
{
	ActionComp->StartActionByName(this, SharedGameplayTags::Action_Blackhole);
}
void ASCharacter::Dash()
{
	ActionComp->StartActionByName(this, SharedGameplayTags::Action_Dash);
}
void ASCharacter::PrimaryInteract()
{
	InteractionComp->PrimaryInteract();
}
void ASCharacter::HealSelf(float Amount)
{
	AttributeComp->ApplyHealthChange(this, Amount);
}
void  ASCharacter::OnHealthChanged(AActor*  InstigatorActor,  USAttributeComponent*
OwningComp, float NewHealth, float Delta)
{
	// Damaged
	if (Delta < 0.0f)
	{
		GetMesh()->SetCustomPrimitiveDataFloat(HitFlash_CustomPrimitiveIndex,
GetWorld()->TimeSeconds);
		const float RageDelta = FMath::Abs(Delta);
		AttributeComp->ApplyRage(InstigatorActor, RageDelta);
	}
	if (NewHealth <= 0.0f && Delta < 0.0f)
	{
		APlayerController* PC = GetController<ASPlayerController>();
		DisableInput(PC);
		SetLifeSpan(5.0f);
	}
}
FVector ASCharacter::GetPawnViewLocation() const
{
	return CameraComp->GetComponentLocation();
}
```

Б.2 Код контроллера SPlayerController

```
#include "SPlayerController.h"
#include "Blueprint/UserWidget.h"
#include "Kismet/GameplayStatics.h"

#include UE_INLINE_GENERATED_CPP_BY_NAME(SPlayerController)



void ASPlayerController::TogglePauseMenu()
 {
	if (PauseMenuInstance && PauseMenuInstance->IsInViewport())
```

```
	{
		PauseMenuInstance->RemoveFromParent();
		PauseMenuInstance = nullptr;

		bShowMouseCursor = false;
		SetInputMode(FInputModeGameOnly());

		// Single-player only
		if (GetWorld()->IsNetMode(NM_Standalone))
		{
			UGameplayStatics::SetGamePaused(this, false);
		}

		return;
	}

	PauseMenuInstance = CreateWidget<UUserWidget>(this, PauseMenuClass);
	if (PauseMenuInstance)
	{
		PauseMenuInstance->AddToViewport(100);

		bShowMouseCursor = true;
		SetInputMode(FInputModeUIOnly());

		// Single-player only
		if (GetWorld()->IsNetMode(NM_Standalone))
		{
			UGameplayStatics::SetGamePaused(this, true);
		}
	}
}


void ASPlayerController::SetupInputComponent()
{
	Super::SetupInputComponent();

	// @todo: replace with Enhanced Input
	InputComponent->BindAction("PauseMenu",		IE_Pressed,		this,
&ASPlayerController::TogglePauseMenu);

	// Keeping as 'old' input for now until we figure out how to do this easily
in Enhanced input
	InputComponent->BindAction("AnyKey",		IE_Pressed,		this,
&ASPlayerController::AnyKeyInput);
}


void ASPlayerController::AnyKeyInput(FKey PressedKey)
{
	bIsUsingGamepad = PressedKey.IsGamepadKey();
}


void ASPlayerController::SetPawn(APawn* InPawn)
{
	Super::SetPawn(InPawn);

	OnPawnChanged.Broadcast(InPawn);
}
```

```
void ASPlayerController::BeginPlayingState()
{
	BlueprintBeginPlayingState();
}


void ASPlayerController::OnRep_PlayerState()
{
	Super::OnRep_PlayerState();

	OnPlayerStateReceived.Broadcast(PlayerState);
}
```

Б.3 Код снаряду SMagicProjectile

```
#include "SMagicProjectile.h"
#include "Components/SphereComponent.h"
#include "SGameplayFunctionLibrary.h"
#include "SActionComponent.h"
#include "Components/SProjectileMovementComponent.h"
#include "SActionEffect.h"

#include UE_INLINE_GENERATED_CPP_BY_NAME(SMagicProjectile)

// NOTE: With SparseDataClass feature in use, some properties are replaced with
"GetXXX()" which is generated automatically by UHT.
// Example: DamageAmount becomes GetDamageAmount() without this function visible
in our own header.

ASMagicProjectile::ASMagicProjectile()
{
	SphereComp->SetSphereRadius(20.0f);
	InitialLifeSpan = 10.0f;
}


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

	// More consistent to bind here compared to Constructor which may fail to
bind if Blueprint was created before adding this binding (or when using hotreload)
	// PostInitializeComponent is the preferred way of binding any events.
	SphereComp->OnComponentBeginOverlap.AddDynamic(this,
&ASMagicProjectile::OnActorOverlap);
}


void ASMagicProjectile::OnActorOverlap(UPrimitiveComponent* OverlappedComponent,
AActor* OtherActor, UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool
bFromSweep, const FHitResult& SweepResult)
{
	if (OtherActor && OtherActor != GetInstigator())
	{
		// Parry Ability (GameplayTag Example)
		USActionComponent* ActionComp = OtherActor-
>FindComponentByClass<USActionComponent>();
```

```
            if (ActionComp && ActionComp-
>ActiveGameplayTags.HasTag(GetParryTag()))
            {
                  MoveComp->Velocity = -MoveComp->Velocity;

                  SetInstigator(Cast<APawn>(OtherActor));
                  return;
            }

            // Apply Damage & Impulse
            if (USGameplayFunctionLibrary::ApplyDirectionalDamage(GetInstigator(),
OtherActor, GetDamageAmount(), SweepResult))
            {
                  // We only explode if the target can be damaged, it ignores
anything it Overlaps that it cannot Damage (it requires an AttributeComponent on
the target)
                  Explode();

                  if (ActionComp && GetBurningActionClass() && HasAuthority())
                  {
                        ActionComp->AddAction(GetInstigator(),
GetBurningActionClass());
                  }
            }
      }
}


#if WITH_EDITOR
// Only required to convert existing properties already stored in Blueprints into
the 'new' system
void ASMagicProjectile::MoveDataToSparseClassDataStruct() const
{
      // make sure we don't overwrite the sparse data if it has been saved already
      const UBlueprintGeneratedClass* BPClass =
Cast<UBlueprintGeneratedClass>(GetClass());
      if (BPClass == nullptr || BPClass->bIsSparseClassDataSerializable == true)
      {
            return;
      }

      Super::MoveDataToSparseClassDataStruct();

#if WITH_EDITORONLY_DATA
      // Unreal Header Tool (UHT) will create GetMySparseClassData automatically.
      FMagicProjectileSparseData* SparseClassData =
GetMagicProjectileSparseData();

      // Modify these lines to include all Sparse Class Data properties.
      SparseClassData->DamageAmount = DamageAmount_DEPRECATED;
      SparseClassData->ParryTag = ParryTag_DEPRECATED;
      SparseClassData->BurningActionClass = BurningActionClass_DEPRECATED;
#endif // WITH_EDITORONLY_DATA
}
#endif
```

Б.4 Код ігрового режиму SGameModeBase

```
#include "SGameModeBase.h"
#include "EnvironmentQuery/EnvQueryManager.h"
```

```
#include "EnvironmentQuery/EnvQueryTypes.h"
#include "AI/SAICharacter.h"
#include "SAttributeComponent.h"
#include "EngineUtils.h"
#include "DrawDebugHelpers.h"
#include "SCharacter.h"
#include "SPlayerState.h"
#include "Kismet/GameplayStatics.h"
#include "GameFramework/GameStateBase.h"
#include "SMonsterData.h"
#include "../ActionRoguelike.h"
#include "SActionComponent.h"
#include "SSaveGameSubsystem.h"
#include "Engine/AssetManager.h"
#include "Subsystems/SActorPoolingSubsystem.h"

#include UE_INLINE_GENERATED_CPP_BY_NAME(SGameModeBase)


static TAutoConsoleVariable<bool> CVarSpawnBots(TEXT("game.SpawnBots"), true,
TEXT("Enable spawning of bots via timer."), ECVF_Cheat);



ASGameModeBase::ASGameModeBase()
{
	SpawnTimerInterval = 2.0f;
	CreditsPerKill = 20;
	CooldownTimeBetweenFailures = 8.0f;

	DesiredPowerupCount = 10;
	RequiredPowerupDistance = 2000;
	InitialSpawnCredit = 50;

	// We start spawning as the player walks on a button instead for convenient
testing w/o bots.
	bAutoStartBotSpawning = false;

	bAutoRespawnPlayer = false;

	PlayerStateClass = ASPlayerState::StaticClass();
}


void ASGameModeBase::InitGame(const FString& MapName, const FString& Options,
FString& ErrorMessage)
{
	Super::InitGame(MapName, Options, ErrorMessage);

	// (Save/Load logic moved into new SaveGameSubsystem)
	USSaveGameSubsystem* SG = GetGameInstance()-
>GetSubsystem<USSaveGameSubsystem>();

	// Optional slot name (Falls back to slot specified in SaveGameSettings
class/INI otherwise)
	FString SelectedSaveSlot = UGameplayStatics::ParseOption(Options,
"SaveGame");
	SG->LoadSaveGame(SelectedSaveSlot);
}
void ASGameModeBase::StartPlay()
{
```

```
	Super::StartPlay();

	AvailableSpawnCredit = InitialSpawnCredit;

	if (bAutoStartBotSpawning)
	{
		StartSpawningBots();
	}

	// Make sure we have assigned at least one power-up class
	if (ensure(PowerupClasses.Num() > 0))
	{
		// Skip the Blueprint wrapper and use the direct C++ option which the
Wrapper uses as well
		FEnvQueryRequest Request(PowerupSpawnQuery, this);
		Request.Execute(EEnvQueryRunMode::AllMatching,                    this,
&ASGameModeBase::OnPowerupSpawnQueryCompleted);
	}

	// We run the prime logic after the BeginPlay call to avoid accidentally
running that on stored/primed actors
	RequestPrimedActors();
}


void ASGameModeBase::RequestPrimedActors()
{
	USActorPoolingSubsystem*         PoolingSystem         =         GetWorld()-
>GetSubsystem<USActorPoolingSubsystem>();
	for (auto& Entry : ActorPoolClasses)
	{
		PoolingSystem->PrimeActorPool(Entry.Key, Entry.Value);
	}
}
void     ASGameModeBase::HandleStartingNewPlayer_Implementation(APlayerController*
NewPlayer)
{
	// Calling Before Super:: so we set variables before 'beginplayingstate' is
called in PlayerController (which is where we instantiate UI)
	USSaveGameSubsystem*            SG            =            GetGameInstance()-
>GetSubsystem<USSaveGameSubsystem>();
	SG->HandleStartingNewPlayer(NewPlayer);

	Super::HandleStartingNewPlayer_Implementation(NewPlayer);

	// Now we're ready to override spawn location
	// Alternatively we could override core spawn location to use store locations
immediately (skipping the whole 'find player start' logic)
	SG->OverrideSpawnTransform(NewPlayer);
}
void ASGameModeBase::KillAll()
{
	for (ASAICharacter* Bot : TActorRange<ASAICharacter>(GetWorld()))
	{
		USAttributeComponent*                  AttributeComp                  =
USAttributeComponent::GetAttributes(Bot);
		if (ensure(AttributeComp) && AttributeComp->IsAlive())
		{
			AttributeComp->Kill(this);
		}
	}
```

```
}


void ASGameModeBase::StartSpawningBots()
{
	if (TimerHandle_SpawnBots.IsValid())
	{
		// Already spawning bots.
		return;
	}

	// Continuous timer to spawn in more bots.
	// Actual amount of bots and whether its allowed to spawn determined by spawn
logic later in the chain...
	GetWorldTimerManager().SetTimer(TimerHandle_SpawnBots,                  this,
&ASGameModeBase::SpawnBotTimerElapsed, SpawnTimerInterval, true);
}

void ASGameModeBase::SpawnBotTimerElapsed()
{
	if (!CVarSpawnBots.GetValueOnGameThread())
	{
		return;
	}

	// Give points to spend
	if (SpawnCreditCurve)
	{
		AvailableSpawnCredit   +=   SpawnCreditCurve->GetFloatValue(GetWorld()-
>TimeSeconds);
	}

	if (CooldownBotSpawnUntil > GetWorld()->TimeSeconds)
	{
		// Still cooling down
		return;
	}

	LogOnScreen(this,    FString::Printf(TEXT("Available    SpawnCredits:    %f"),
AvailableSpawnCredit));

	// Count alive bots before spawning
	int32 NrOfAliveBots = 0;
	// TActorRange simplifies the code compared to TActorIterator<T>
	for (ASAICharacter* Bot : TActorRange<ASAICharacter>(GetWorld()))
	{
		USAttributeComponent*                   AttributeComp                   =
USAttributeComponent::GetAttributes(Bot);
		if (ensure(AttributeComp) && AttributeComp->IsAlive())
		{
			NrOfAliveBots++;
		}
	}

	UE_LOGFMT(LogGame, Log, "Found {number} alive bots.", NrOfAliveBots);

	const float MaxBotCount = 10.0f;
	if (NrOfAliveBots >= MaxBotCount)
	{
		UE_LOGFMT(LogGame, Log, "At maximum bot capacity. Skipping bot spawn.");
		return;
```

```
	}

	if (MonsterTable)
	{
		// Reset before selecting new row
		SelectedMonsterRow = nullptr;

		TArray<FMonsterInfoRow*> Rows;
		MonsterTable->GetAllRows("", Rows);

		// Get total weight
		float TotalWeight = 0;
		for (FMonsterInfoRow* Entry : Rows)
		{
			TotalWeight += Entry->Weight;
		}

		// Random number within total random
		int32 RandomWeight = FMath::RandRange(0.0f, TotalWeight);

		//Reset
		TotalWeight = 0;

		// Get monster based on random weight
		for (FMonsterInfoRow* Entry : Rows)
		{
			TotalWeight += Entry->Weight;

			if (RandomWeight <= TotalWeight)
			{
				SelectedMonsterRow = Entry;
				break;
			}
		}

		if (SelectedMonsterRow && SelectedMonsterRow->SpawnCost >=
AvailableSpawnCredit)
		{
			// Too expensive to spawn, try again soon
			CooldownBotSpawnUntil = GetWorld()->TimeSeconds +
CooldownTimeBetweenFailures;

			LogOnScreen(this, FString::Printf(TEXT("Cooling down until: %f"),
CooldownBotSpawnUntil), FColor::Red);
			return;
		}
	}

	UE_LOG(LogGame, Log, TEXT("Spawning New Bot"));

	// Skip the Blueprint wrapper and use the direct C++ option which the Wrapper
uses as well
	FEnvQueryRequest Request(SpawnBotQuery, this);
	Request.Execute(EEnvQueryRunMode::RandomBest5Pct, this,
&ASGameModeBase::OnBotSpawnQueryCompleted);
}


void ASGameModeBase::OnBotSpawnQueryCompleted(TSharedPtr<FEnvQueryResult> Result)
{
	FEnvQueryResult* QueryResult = Result.Get();
```

```
	if (!QueryResult->IsSuccessful())
	{
		UE_LOGFMT(LogGame, Warning, "Spawn bot EQS Query Failed!");
		return;
	}

	// Retrieve all possible locations that passed the query
	TArray<FVector> Locations;
	QueryResult->GetAllAsLocations(Locations);

	if (Locations.IsValidIndex(0) && MonsterTable)
	{
		UAssetManager& Manager = UAssetManager::Get();

		// Apply spawn cost
		AvailableSpawnCredit -= SelectedMonsterRow->SpawnCost;

		FPrimaryAssetId MonsterId = SelectedMonsterRow->MonsterId;

		TArray<FName> Bundles;
		FStreamableDelegate Delegate = FStreamableDelegate::CreateUObject(this,
&ASGameModeBase::OnMonsterLoaded, MonsterId, Locations[0]);
		Manager.LoadPrimaryAsset(MonsterId, Bundles, Delegate);
	}
}
void	ASGameModeBase::OnMonsterLoaded(FPrimaryAssetId	LoadedId,	FVector
SpawnLocation)
{
	//LogOnScreen(this, "Finished loading.", FColor::Green);

	UAssetManager& Manager = UAssetManager::Get();

	USMonsterData*	MonsterData	=
CastChecked<USMonsterData>(Manager.GetPrimaryAssetObject(LoadedId));

	AActor*	NewBot	=	GetWorld()->SpawnActor<AActor>(MonsterData->MonsterClass,
SpawnLocation, FRotator::ZeroRotator);
	// Spawn might fail if colliding with environment
	if (NewBot)
	{
		LogOnScreen(this,	FString::Printf(TEXT("Spawned	enemy:	%s	(%s)"),
*GetNameSafe(NewBot), *GetNameSafe(MonsterData)));

		// Grant special actions, buffs etc.
		USActionComponent*	ActionComp	=	NewBot-
>FindComponentByClass<USActionComponent>();
		check(ActionComp);

		for (TSubclassOf<USAction> ActionClass : MonsterData->Actions)
		{
			ActionComp->AddAction(NewBot, ActionClass);
		}
	}
}
void	ASGameModeBase::OnPowerupSpawnQueryCompleted(TSharedPtr<FEnvQueryResult>
Result)
{
	FEnvQueryResult* QueryResult = Result.Get();
	if (!QueryResult->IsSuccessful())
	{
		UE_LOGFMT(LogGame, Warning, "Spawn bot EQS Query Failed!");
```

```
			return;
		}

		// Retrieve all possible locations that passed the query
		TArray<FVector> Locations;
		QueryResult->GetAllAsLocations(Locations);

		// Keep used locations to easily check distance between points
		TArray<FVector> UsedLocations;

		int32 SpawnCounter = 0;
		// Break out if we reached the desired count or if we have no more potential
positions remaining
		while (SpawnCounter < DesiredPowerupCount && Locations.Num() > 0)
		{
			// Pick a random location from remaining points.
			int32 RandomLocationIndex = FMath::RandRange(0, Locations.Num() - 1);

			FVector PickedLocation = Locations[RandomLocationIndex];
			// Remove to avoid picking again
			Locations.RemoveAt(RandomLocationIndex);

			// Check minimum distance requirement
			bool bValidLocation = true;
			for (FVector OtherLocation : UsedLocations)
			{
				float DistanceTo = (PickedLocation - OtherLocation).Size();

				if (DistanceTo < RequiredPowerupDistance)
				{
					// Show skipped locations due to distance
					//DrawDebugSphere(GetWorld(),  PickedLocation,  50.0f,  20,
FColor::Red, false, 10.0f);

					// too close, skip to next attempt
					bValidLocation = false;
					break;
				}
			}

			// Failed the distance test
			if (!bValidLocation)
			{
				continue;
			}

			// Pick a random powerup-class
			int32 RandomClassIndex = FMath::RandRange(0, PowerupClasses.Num() - 1);
			TSubclassOf<AActor>                RandomPowerupClass                =
PowerupClasses[RandomClassIndex];

			GetWorld()->SpawnActor<AActor>(RandomPowerupClass,      PickedLocation,
FRotator::ZeroRotator);

			// Keep for distance checks
			UsedLocations.Add(PickedLocation);
			SpawnCounter++;
		}
}
void ASGameModeBase::RespawnPlayerElapsed(AController* Controller)
{
```

```
	if (ensure(Controller))
	{
		Controller->UnPossess();

		RestartPlayer(Controller);
	}
}
void ASGameModeBase::OnActorKilled(AActor* VictimActor, AActor* Killer)
{
	UE_LOGFMT(LogGame, Log, "OnActorKilled: Victim: {victim}, Killer: {killer}",
GetNameSafe(VictimActor), GetNameSafe(Killer));

	// Handle Player death
	ASCharacter* Player = Cast<ASCharacter>(VictimActor);
	if (Player)
	{
		// Auto-respawn
		if (bAutoRespawnPlayer)
		{
			FTimerHandle TimerHandle_RespawnDelay;
			FTimerDelegate Delegate;
			Delegate.BindUFunction(this,   "RespawnPlayerElapsed",   Player-
>GetController());

			float RespawnDelay = 2.0f;
			GetWorldTimerManager().SetTimer(TimerHandle_RespawnDelay,
Delegate, RespawnDelay, false);
		}

		// Store time if it was better than previous record
		ASPlayerState* PS = Player->GetPlayerState<ASPlayerState>();
		if (PS)
		{
			PS->UpdatePersonalRecord(GetWorld()->TimeSeconds);
		}

		USSaveGameSubsystem*          SG          =          GetGameInstance()-
>GetSubsystem<USSaveGameSubsystem>();
		// Immediately auto save on death
		SG->WriteSaveGame();
	}

	// Give Credits for kill
	APawn* KillerPawn = Cast<APawn>(Killer);
	// Don't credit kills of self
	if (KillerPawn && KillerPawn != VictimActor)
	{
		// Only Players will have a 'PlayerState' instance, bots have nullptr
		ASPlayerState* PS = KillerPawn->GetPlayerState<ASPlayerState>();
		if (PS)
		{
			PS->AddCredits(CreditsPerKill);
		}
	}
}
```

ДОДАТОК В
(обов’язковий)

КЕРІВНИЦТВО КОРИСТУВАЧА

Правила ігри:

Ви керуєте головним персонажем (магом у синьому). У персонажа є рівень здоров’я та досвіду. Досвід накопичується за перемогу над ворогом, досвід можна витрачати на використання ультимативної здібності (чорної діри). Здоров’я втрачається після вдалої атаки ворога, відновити його можна за допомогою лікувального зілля, якщо підійти до нього і натиснути кнопку взаємодії.

Вороги (у білій броні) розміщенні по рівню. Вони завжди шукають гравця і якщо побачать почнуть переслідувати і атакувати.

Головна ціль перемогти всіх ворогів і дібратись до кінця рівня.

Призначення клавіш:

W,S,A,D – іти вперед, назад, вліво, вправо;

Space – стрибок;

ЛКМ – звичайна атака;

ПКМ – використати здібність;

E – взаємодіяти з предметом;

F – стрейф/переміщення.

ДОДАТОК Г

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

Рисунок В.1 – Слайд 1

Рисунок В.2 – Слайд 2

Рисунок В.3 – Слайд 3

Рисунок В.4 – Слайд 4

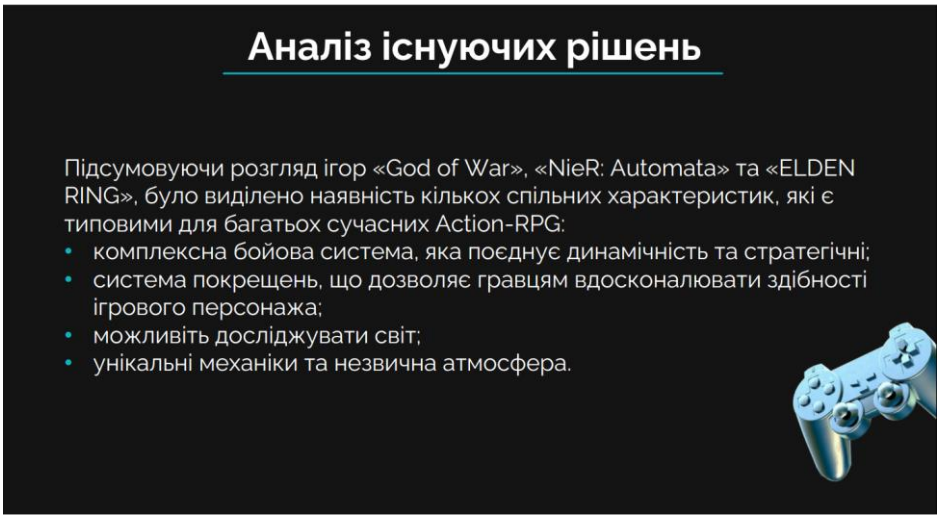


Рисунок В.5 – Слайд 5

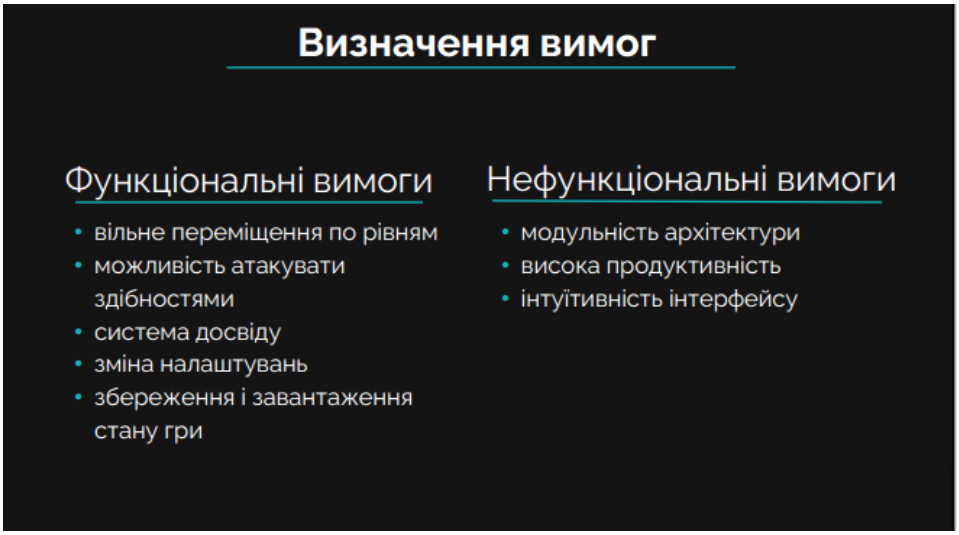


Рисунок В.6 – Слайд 6

Рисунок В.7 – Слайд 7

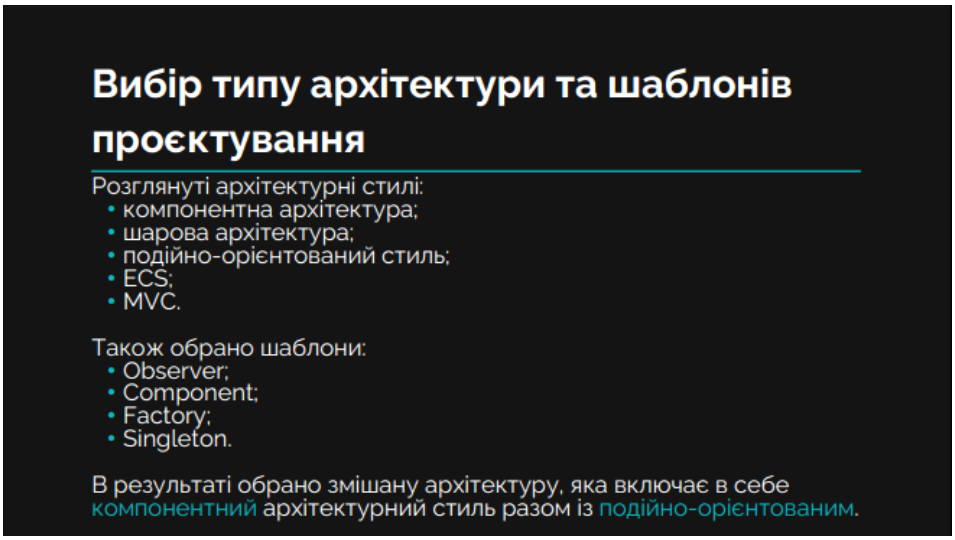


Рисунок В.8 – Слайд 8

Рисунок В.9 – Слайд 9

Модуль:	Інтерфейс
HealthComponent	GetHealth(), SetHealth(), IsDead()
LevelComponent	GetExp(), SetExp(), GetLvl()
FightComponent	GetDamage(), Attack(), SpellAttak(), IsAttack().
MovementComponent	GetMass(), SetMass(), GetGrav(), SetGrav(), GetAccel(), SetAccel(), AddAccel(), IsMove()
Character	GetComponentByClass()
GameController	MoveTo(), Jump(), Dush(), Attack(), SpellAttack(), GetCharacter(), SetCharacter
GameMod	GetGameController(), SetGameController(), GetCharacter(), SetCharacter().
DialogComponent	StartDialog(), StartNextSentence(), EndDialog().
UISystem	AddWidgetByClass(), RemoveWidget(), HideWidget().

Рисунок В.10 – Слайд 10

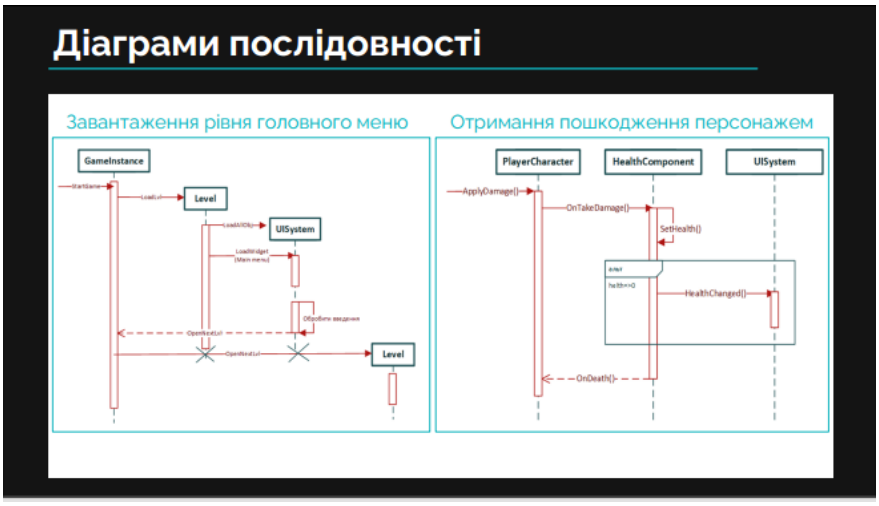


Рисунок В.11 – Слайд 11

Рисунок В.12 – Слайд 12

Рисунок В.13 – Слайд 13

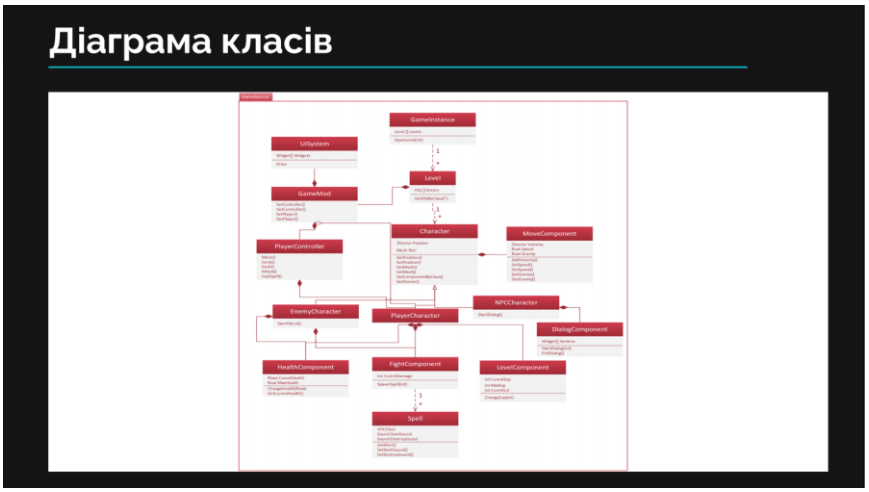


Рисунок В.14 – Слайд 14

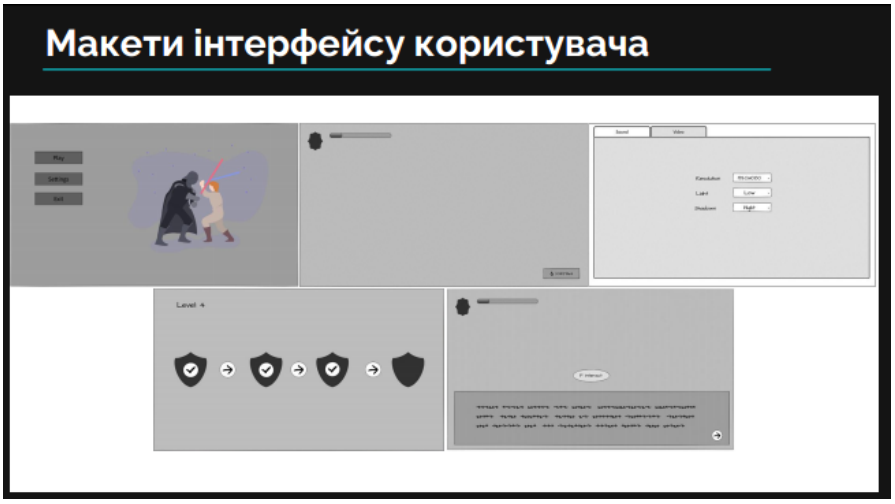


Рисунок В.15 – Слайд 15

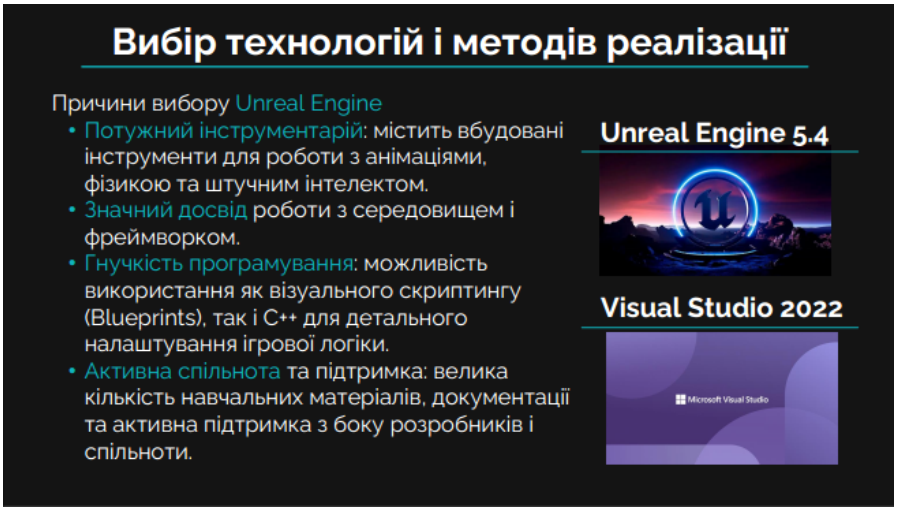


Рисунок В.16 – Слайд 16

Рисунок В.17 – Слайд 17

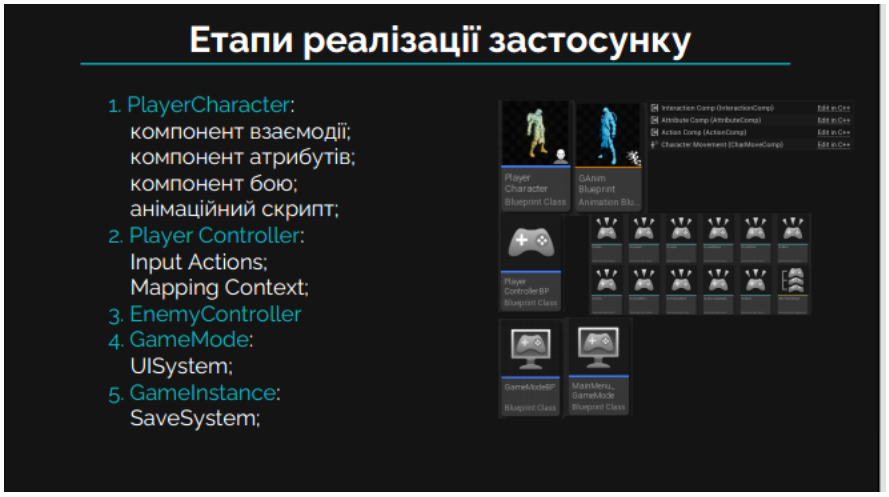


Рисунок В.18 – Слайд 18

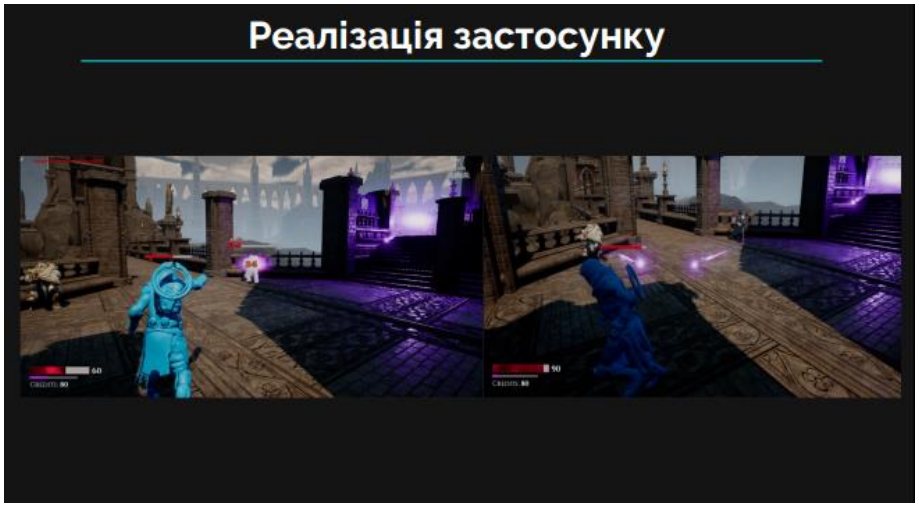


Рисунок В.19 – Слайд 19

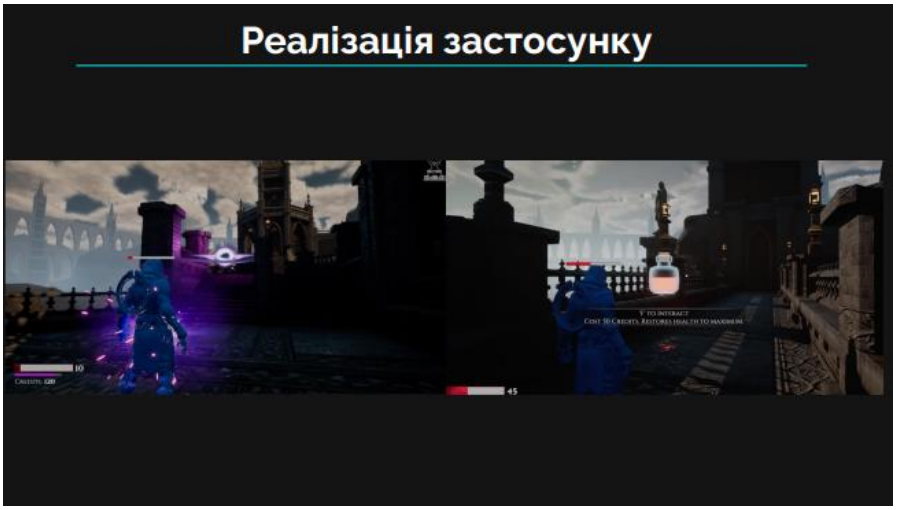


Рисунок В.20 – Слайд 20

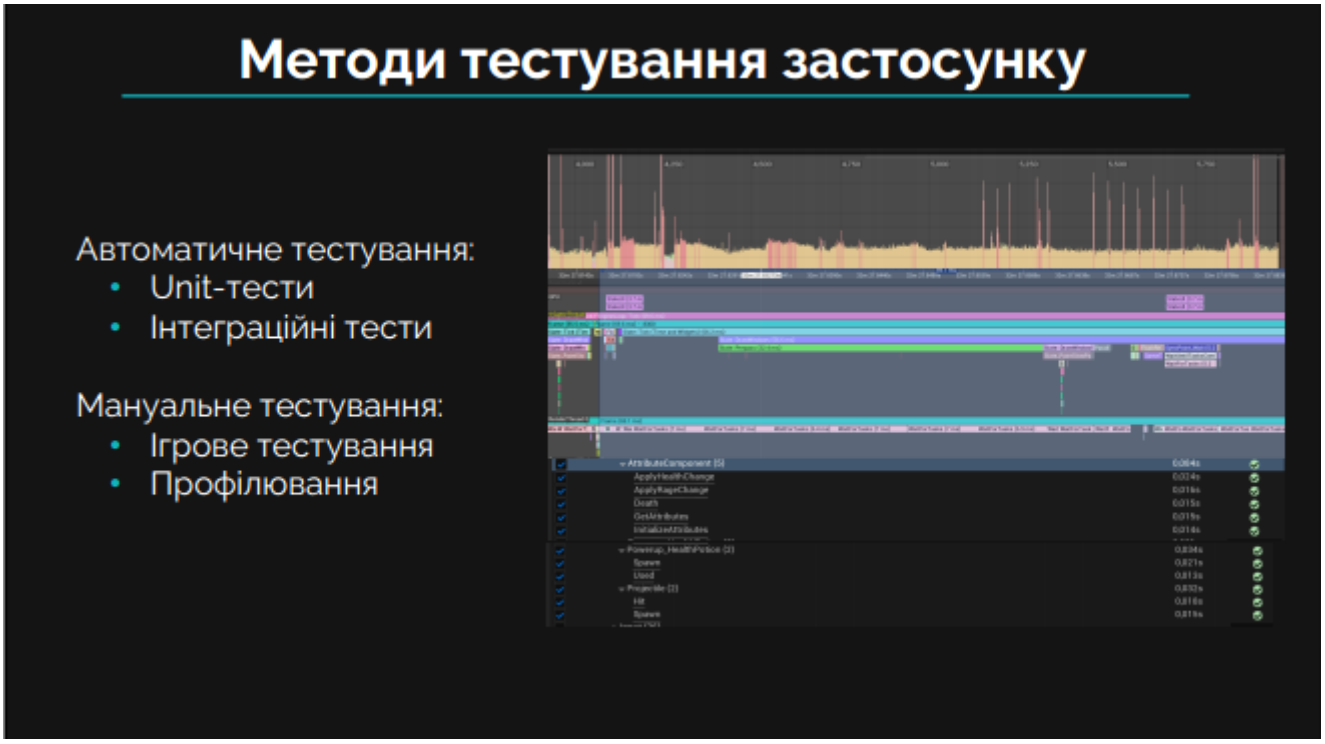


Рисунок В.21 – Слайд 21

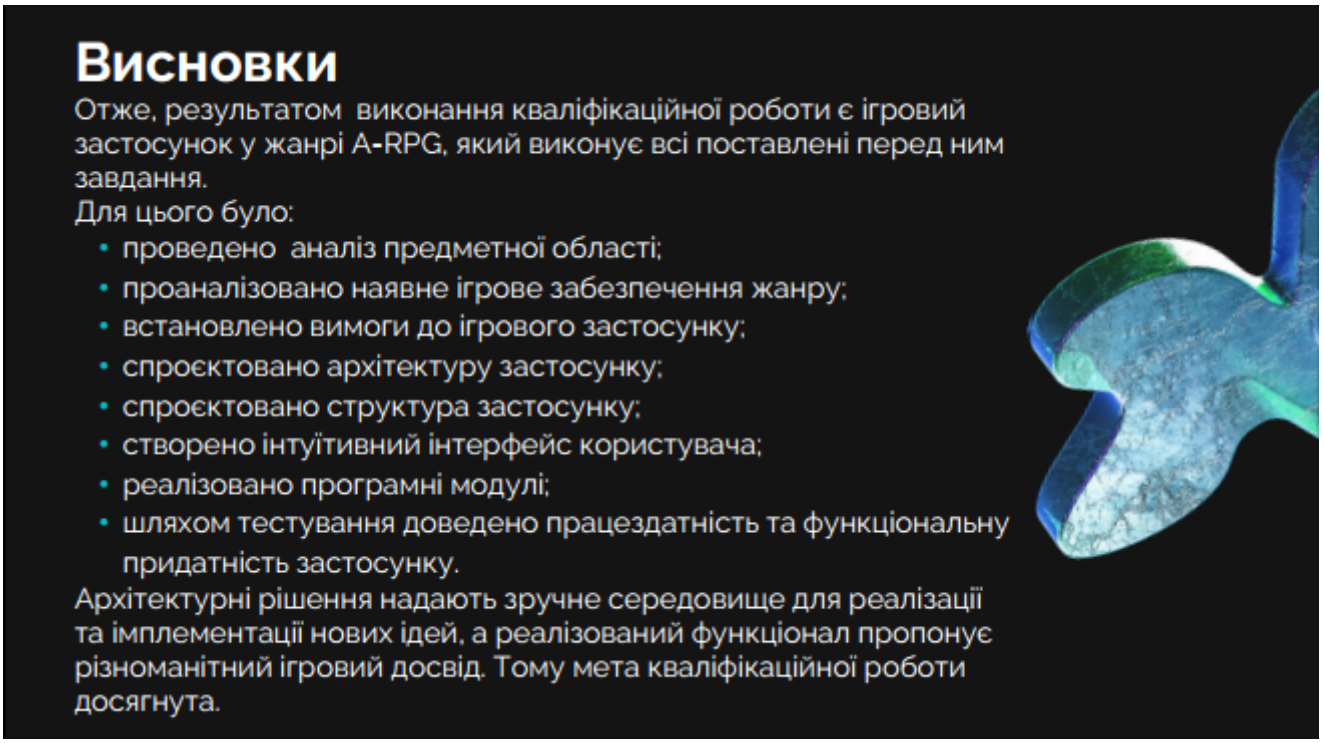
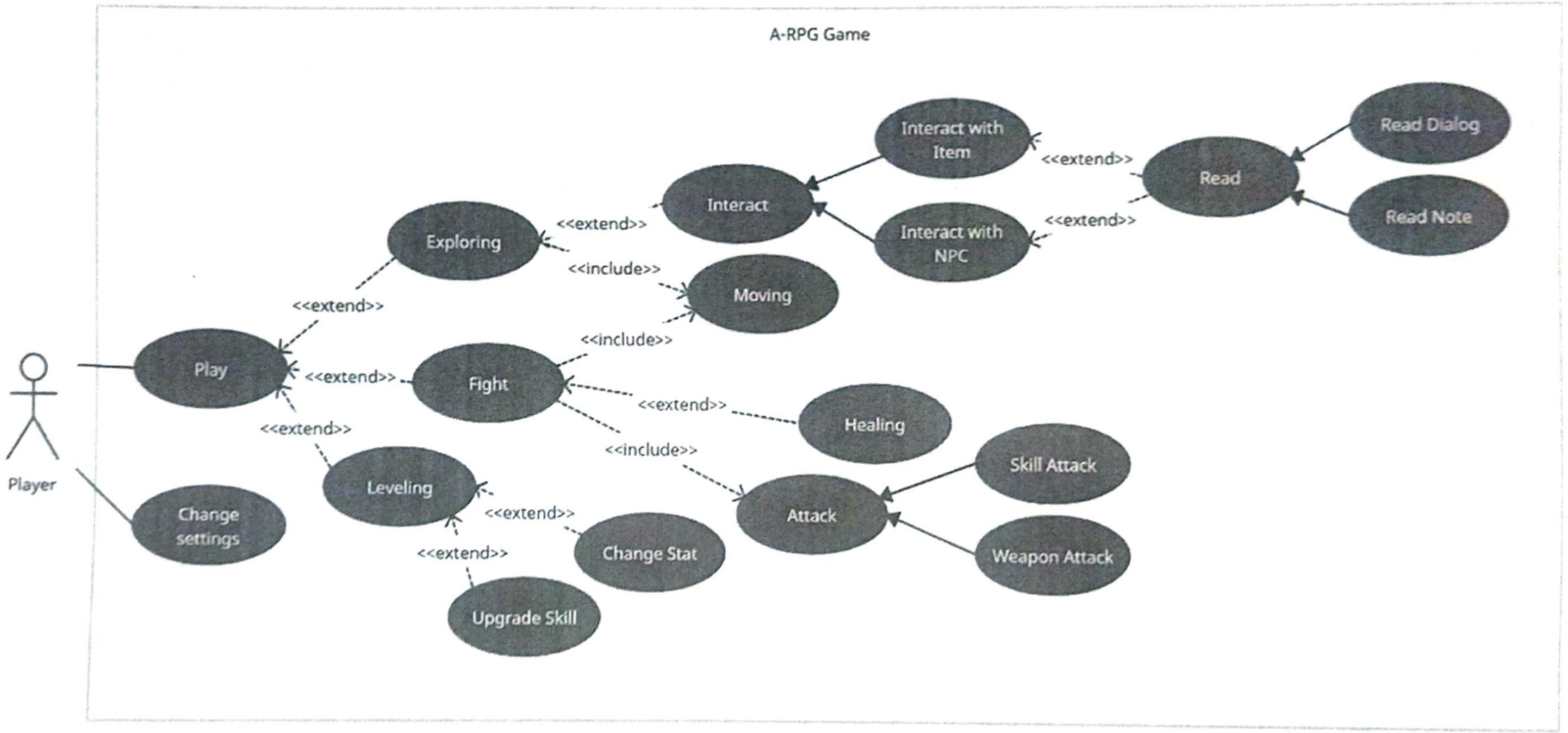
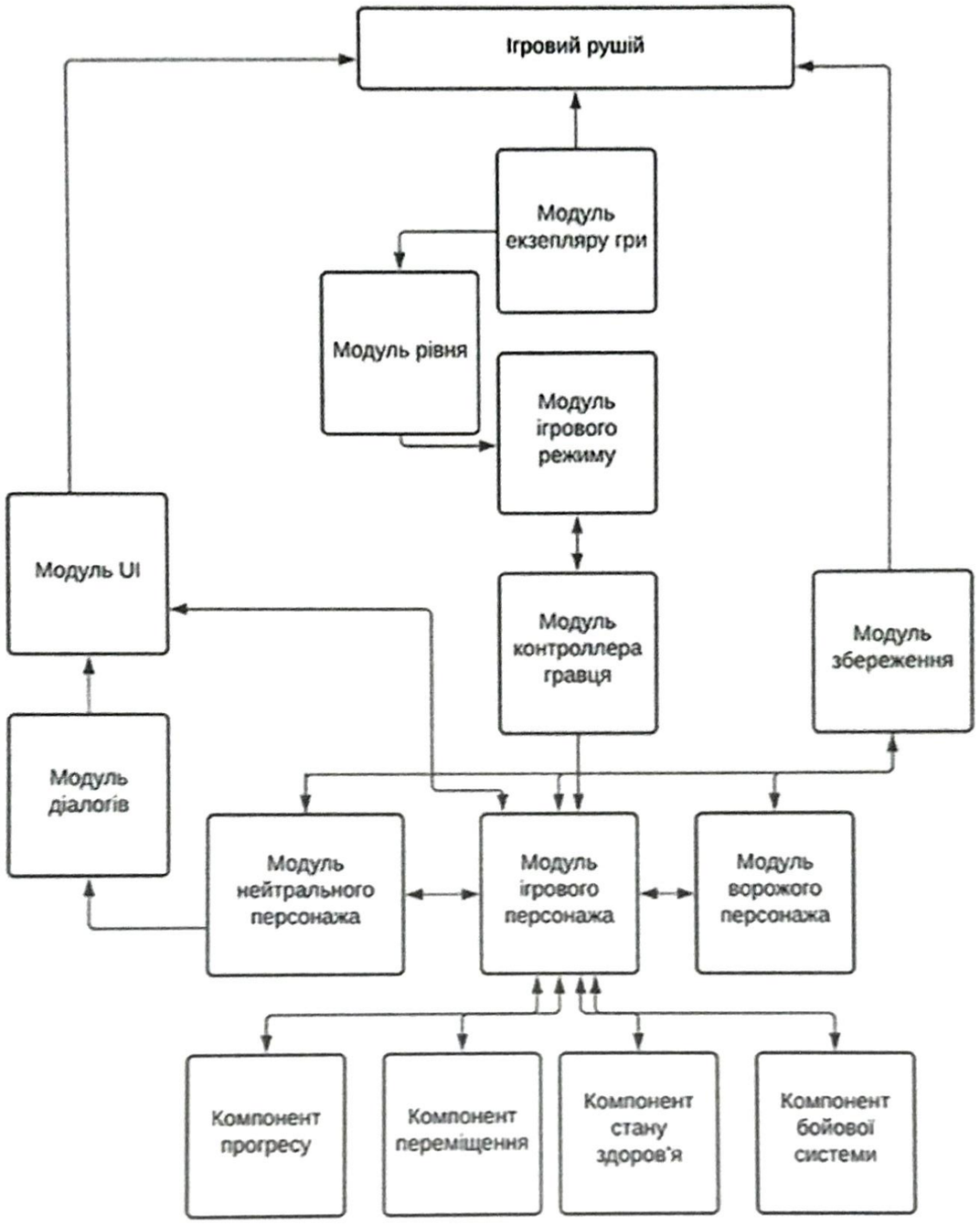


Рисунок В.22 – Слайд 22

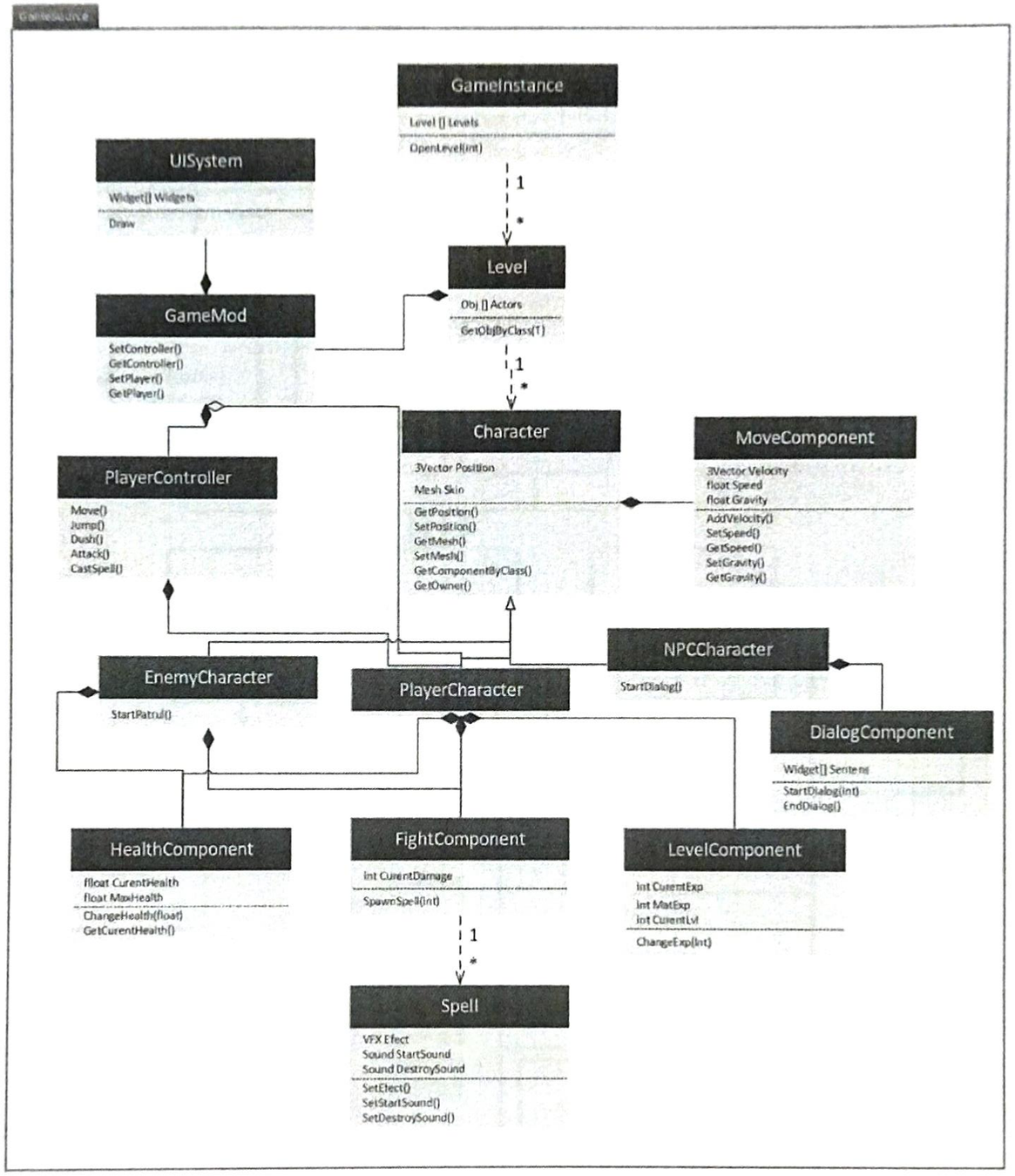
ГРАФІЧНІ МАТЕРІАЛИ



					КвРІПЗ.200257.01.24.Е8	
Зм.	Арк.	№ докум.	Підпис	Дата	Діаграма варіантів використання	Літера / Маса / Масштаб
Розробив		Шугалюк А.І.		10.06		
Керівник		Радельчук Г.І.		10.06		Аркуш 1 / Аркушів 3
Консульт.						
Н. Контр.		Радельчук Г.І.		10.06		ХНУ, ІПЗ-20-1
Зав. каф.		Бедратюк Л.П.		10.06		



					КвРІПЗ.200257.01.24.Е8	
Зм.	Арк.	№ докум.	Підпис	Дата	Діаграма зв'язків модулів	Літера / Маса / Масштаб
Розробив		Шугалюк А.І.		10.06		
Керівник		Радельчук Г.І.		10.06		
Консульт.						Аркуш 2 / Аркушів 3
Н. Контр.		Радельчук Г.І.		10.06	ХНУ, ІПЗ-20-1	
Зав. каф.		Бедратюк Л.П.		10.06		



					КвРІПЗ.200257.01.24.Е8	
Зм.	Арк.	№ докум.	Підпис	Дата	Діаграма класів	Літера / Маса / Масштаб
Розробив		Шугалюк А.І.		10.06		
Керівник		Радельчук Г.І.		10.06		Аркуш 3 / Аркушів 3
Консульт.						
Н. Контр.		Радельчук Г.І.		10.06		ХНУ, ІПЗ-20-1
Зав. каф		Бедратюк Л.Г.		10.06		

СУПРОВІДНІ ДОКУМЕНТИ

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

здобувача вищої освіти
Шугалюк А. 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.

04.06.2024
дата

підпис

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 2.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 10%**

ID: 129572 Назва: БКР_Ігровий_застосунок_у_жанрі_Action-RPG_Шугалюк А.І._Радельчук Г.І. Додано в БД: 2024-06-10 Автора: Шугалюк А. І. Керівники: Радельчук Г. І., канд. техн. наук, доцент Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	86593	1312	2901 (3%)	42 (3%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

UNICHECK
by Turnitin

Ім'я користувача:
ІПЗ

ID перевірки:
1016342734

Дата перевірки:
11.06.2024 06:00:57 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
11.06.2024 09:49:21 EEST

ID користувача:
100012953

Назва документа: **БКР_Ігровий_застосунок_у_жанрі_Action-RPG_Шугалюк А.І._Радельчук Г.І**

Кількість сторінок: **72** Кількість слів: **14434** Кількість символів: **112546** Розмір файлу: **2.63 MB** ID файлу: **1016144110**

4.91%
Схожість

Найбільша схожість: **1.46%** з джерелом з Бібліотеки (ID файлу: **1015128148**)

3.09% Джерела з Інтернету	488	Сторінка 74
2.92% Джерела з Бібліотеки	170	Сторінка 76

0.06% Цитат

Цитати	1	Сторінка 77

Не знайдено жодних посилань

0%
Вилучень

Немає вилучених джерел

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ
освітнього ступеня «Бакалавр»

Дипломник Шугалюк Андрій Ігорович

Тема Ігровий застосунок у жанрі «Action-RPG»

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг кваліфікаційної роботи:

Кількість листів креслень 3; кількість сторінок записки 74

1.Короткий зміст пояснювальної записки та прийнятих рішень
У кваліфікаційній роботі досліджено і проаналізовано предметну область розроблення ігрових застосунків у жанрі Action-RPG. Визначено всі функціональні та нефункціональні вимоги до гри. Проведено детальний аналіз існуючих ігор на ринку, розглянуто їхні переваги та недоліки, що підтвердило актуальність створення нового ігрового застосунку. Обрано відповідні інструменти для реалізації проєкту на базі Unreal Engine, на основі яких було створено ігровий застосунок. Проведено тестування гри, результати якого засвідчили, що розроблений ігровий застосунок функціонує коректно і готовий до впровадження.

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

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки та передових методів роботи У вступі доведено актуальність теми, визначено мету та завдання кваліфікаційної роботи. У першому розділі проведено аналіз предметної області, розглянуто існуючі рішення та визначені функціональні і нефункціональні вимоги до розроблюваного ігрового застосунку. У другому розділі проведено аналіз сучасних архітектурних підходів, розглянуто їх переваги і недоліки та визначено, що система буде відповідати компонентному та обєктно-орієнтованому підходу. У третьому розділі розглянуто всі технологічні особливості для написання коду та виконано практичне розроблення програмних модулів і описано їх особливості, в результаті чого створено ігровий застосунок. Також у цьому розділі виконано unit-тестування, ігрове тестування і тестування продуктивності застосунку та проведено його у відповідності до функціональних вимог. У результаті тестування було підтверджено коректну роботу ігрового застосунку та його функціональну придатність.

4. Позитивні сторони роботи Тема кваліфікаційної роботи через постійне зростання ринку є актуальною сьогодні. Прийняті архітектурні та технологічні рішення відповідають актуальним вимогам ігрової індустрії. Для реалізації застосунку були використані сучасні технологічні рішення.

5. Негативні сторони роботи Реалізовано базовий функціонал притаманний жанру, відсутній інвентар гравця, відкритий світ, система квестів.

6. Оцінка графічного оформлення та пояснювальної записки Графічне оформлення виконано відповідно до теми кваліфікаційної роботи та подано у вигляді діаграм і рисунків. Пояснювальна записка оформлена згідно вимог чинних стандартів.

7. Відгук про кваліфікаційну роботу в цілому Робота виконана на високому рівні та заслуговує на позитивну оцінку. Пояснювальна записка чітко структурована, логічна, зрозуміла та лаконічна. Це робить викладений матеріал доступним та легко засвоюваним у рамках тематики проєктування. Графічне оформлення доповнює текст, даючи можливість наочно візуалізувати деталі проєктування застосунку.

8. Інші зауваження

9. Оцінка кваліфікаційної роботи Кваліфікаційна робота виконана у повному обсязі, відповідає поставленій задачі та заслуговує на оцінку «добре».

РЕЦЕНЗЕНТ Мартинюк Валерій Володимирович, доктор технічних наук, професор, зав. кафедри автоматизації, комп'ютерно-інтегрованих технологій та робототехніки ХНУ

"10" червня 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: «Ігровий застосунок у жанрі «Action-RPG»»
Автор: Шуталюк Андрій 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, у структурі змісту, назвах розділів/підрозділів, рамках форм, у назвах та URL-адресах публікацій переліку джерел посилання;

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

Максимальний обсяг запозичень, визначений системою Anti-Plagiarism, складає **2.0%**. Обсяг запозичень, визначений системою Unicheck виявлення збігів ідентичності/схожості, складає **4.91%** і адресується до 488 джерел з Інтернету і 170 джерел 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