

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 122 «Комп'ютерні науки»
(освітня програма «Комп'ютерна графіка та розробка ігор»)
на тему:
«Ігровий додаток в жанрі "Метроїдванія"»

Виконав студент групи КН-20-1
ЛЕВИЦЬКИЙ Андрій Андрійович

Керівник роботи:
ЛЕВКІВСЬКИЙ Віталій Леонідович

Рецензент:
ГРАФ Марина Сергіївна

ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук
Марина ГРАФ
«25» січня 2024 р.

ЗАВДАННЯ

на кваліфікаційну роботу освітнього ступеня «бакалавр»
за спеціальністю 122 «Комп'ютерні науки»
(освітня програма «Комп'ютерна графіка та розробка ігор»)

Здобувач вищої освіти: **ЛЕВИЦЬКИЙ Андрій Андрійович**

Керівник роботи: **ЛЕВКІВСЬКИЙ Віталій Леонідович**

Тема роботи: **«Ігровий додаток в жанрі "Метроїдванія"»**,
затверджена наказом закладу вищої освіти від «25» січня 2024 р., №22/с

Термін здачі закінченої роботи: «15» червня 2024 р.

Вихідні дані роботи: об'єктом дослідження є процес розробки ігрових додатків жанру «Метроїдванія», патерни та принципи об'єктно-орієнтованого програмування для ігор, інструментарій для розробки елементів додатку. Предмет дослідження - аналіз обраного жанру та існуючих аналогів, визначення ігрового циклу, розробка програмної та візуальної частини, об'єднання елементів в один додаток, тестування отриманого додатку.

Консультанти кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Дата	
		Завдання видав	Завдання прийняв

Календарний план

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Постановка задачі. Пошук, огляд та аналіз аналогічних розробок. Формулювання технічного завдання. Опрацювання літературних джерел.	25.01.24 – 27.02.24	Виконано
2	Проектування структури додатку.	28.02.24 – 30.03.24	Виконано
3	Написання програмного коду.	01.04.24 – 15.05.24	Виконано
4	Тестування додатку.	16.05.24 – 25.05.24	Виконано
5	Оформлення пояснювальної записки.	26.05.24 – 06.06.24	Виконано
6	Попередній захист.	07.06.2024	Виконано

Здобувач вищої освіти **Андрій ЛЕВИЦЬКИЙ**
Керівник **Віталій ЛЕВКІВСЬКИЙ**

РЕФЕРАТ

Кваліфікаційна робота складається з ігрового додатку та пояснювальної записки.

Пояснювальна записка до кваліфікаційної роботи складається з вступу, основної частини, висновків, списку використаних джерел та додатків.

Текстова частина викладена на 80 сторінках друкованого тексту. Пояснювальна записка має 18 сторінок додатків. Список використаних джерел містить 28 найменувань, і займає 3 сторінки.

В роботі наведено 58 рисунків. Загальний обсяг роботи – 98 сторінок.

Ключові слова: ГРА, METROIDVANIA, UNITY, 2D, ПЛАТФОРМЕР, UI, ASEPRITE, ГЕЙМДИЗАЙН, ПАТЕРН.

					КН.КР.Б – 122 – 24 – ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата	«Ігровий додаток в жанрі "Метроїдванія"» Пояснювальна записка	Літ.	Арк.	Аркушів
Розроб.		А. А. ЛЕВИЦЬКИЙ						
Керівник		В. Л. ЛЕВКІВСЬКИЙ					3	98
Реценз.		М. С. ГРАФ				Житомирська політехніка, група КН-20-1		
Н. Контр.								
Затверд.		М. С. ГРАФ						

ABSTRACT

The qualification work consists of a game application and an explanatory note.

The explanatory note to the qualification work consists of an introduction, main part, conclusion, a reference list and annexes.

The text part is laid out on 80 pages of printed text.

The explanatory note has 18 pages of annexes. The reference list contains 28 names and occupies 3 pages.

The work contains 58 figures. The total volume of work is 98 pages.

Keywords: GAME, METROIDVANIA, UNITY, 2D, PLATFORMER, UI, ASEPRITE, GAME-DESIGNE, PATTERN.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		4

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	7
ВСТУП	8
РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ.....	10
1.1. Постановка задачі.....	10
1.2. Аналіз існуючого програмного забезпечення за тематикою кваліфікаційної роботи	12
1.3. Концепт-документ (Concept Document).....	14
1.4. Обґрунтування вибору інструментальних засобів	15
Висновки до першого розділу.....	18
РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	19
2.1. Проектування архітектури ігрового додатку	19
2.2 Розробка програмної частини елементів ігрового циклу.....	23
2.2.1. Програмна реалізація персонажа та ворогів	24
2.2.2. Програмна реалізація зброї для сутностей	38
2.2.3. Програмна реалізація об'єктів на локації	43
2.3. Розробка програмної частини логіки ігрового додатку	48
2.3.1. Програмна реалізація станів ігрового циклу.....	48
2.3.2. Програмна реалізація системи збереження	50
2.4. Реалізація графічних елементів, анімацій та інтерфейсу користувача ..	54
Висновки до другого розділу	62
РОЗДІЛ 3. ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО ТЕСТУВАННЯ	64

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		5

3.1. Порядок встановлення та налаштування системи	64
3.2. Інтерфейс та порядок роботи	64
3.3. Тестування роботи додатку	70
Висновки до третього розділу	73
ВИСНОВКИ	75
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	77
ДОДАТКИ	80

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

GUI – Graphic User Interface – графічний інтерфейс, елементи за допомогою яких користувач взаємодіє з програмою, отримує данні;

ЛКМ – Ліва клавіша миші;

ООП – об’єктно орієнтоване програмування;

ПЗ – Програмне забезпечення;

ПК – Персональний комп’ютер;

ПКМ – Права клавіша миші.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						7
Змн.	Арк.	№ докум.	Підпис	Дата		

ВСТУП

Актуальність теми. Ігрова індустрія в наш час – це великий розділ ПЗ, котрий має широке застосування та значний дохід. Ігрові додатки мають різне призначення: зазвичай розважальне або те, що кидає виклик користувачам, однак певні представники подібного ПЗ використовуються для навчання, симуляції різного виду моделей та дослідження ходів розвитку подій. Враховуючи їх широкий спектр використання, ігрові додатки можна вважати корисними для суспільства.

Розвиток ігрової індустрії дає можливість створювати більш якісні додатки та надавати мотивацію вдосконалювати людські технології. Для цього потрібні відповідні фахівці, котрі матимуть змогу їх створювати. Так як освітня програма спеціальності 122 «Комп’ютерні науки» передбачає вивчення подібної галузі, котра є однією з впливових на ринку ПЗ, актуальність обраної теми є беззаперечною.

Жанр «Метроїдванія» був закладений в основу додатка через його популярність серед розробників в ігровій індустрії та гравців по всьому світу. Він є достатньо комплексним для реалізації та потребує обширний стек інструментів та технологій, тому обраний жанр є гарною практикою для розробки подібного ПЗ та можливістю представити власні ідеї та сюжети в подібній формі.

Мета кваліфікаційної роботи. Створити ігровий додаток в жанрі «Метроїдванія» з необхідним функціоналом та набором елементів.

Об’єктом дослідження є процес розробки ігрових додатків жанру «Метроїдванія», патерни та принципи об’єктно-орієнтованого програмування для ігор, інструментарій для розробки елементів додатку.

Предмет дослідження. Аналіз обраного жанру та існуючих аналогів, визначення ігрового циклу, розробка програмної та візуальної частини, об’єднання елементів в один додаток, тестування отриманого додатку.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		8

В рамках розробки кваліфікаційної роботи були опубліковані наукові праці:

Левицький А. А., Левківський В. Л. Патерн state як засіб для структуризації програмного коду. Тези VI Всеукраїнської науково-технічної конференції "Комп'ютерні технології: інновації, проблеми, рішення", 29-30 листопада 2023 р. Житомир: Житомирська політехніка, 2023. С. 247-248.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						9
Змн.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ

1.1. Постановка задачі

Обрана тема кваліфікаційної роботи – «Ігровий додаток в жанрі "Метроїдванія"». Це передбачає створення ігрового додатку, котрий буде мати характерні риси обраного жанру. Перед постановкою задачі потрібно дослідити жанр.

Жанр «Метроїдванія» (Metroidvania) [1] виник у результаті поєднання елементів двох видатних ігрових серій "Metroid" і "Castlevania". Його часто визначають як платформер (частіше всього в 2D просторі з видом збоку) з відкритим світом, в якому гравцеві потрібно досліджувати ігрову мапу, котра має декілька шляхів, по яким можна рухатись.

У таких іграх часто використовується нелінійний підхід до проходження рівнів, що вимагає стратегічного мислення та вміння використовувати механіки гри та її мапи для подолання перешкод. Під час проходження таких ігор гравець відкриває нові області та знаходить секрети за допомогою здібностей, які можна отримати протягом гри. Предмети слугують ключем для подолання перешкод, вирішення головоломок та переходу на недоступні до цього локації мапи. Така концепція заохочує гравця до дослідження задля отримання предметів, знаходження нових шляхів або отримання нових відомостей про ігровий світ.

З описаними вище критеріями жанру можна сформулювати частину задач, що стосуються ігрового циклу ігрового додатку:

- створення персонажа, котрий керується гравцем та має цікаві вміння для дослідження світу;
- локації, які матимуть основний шлях проходження та розгалуження з перешкодами, після проходження яких можна отримати нагороду;
- предмети, що відкривають або покращують здібності персонажа, відновлюють певний параметр (наприклад, здоров'я) або дають відомості про ігровий світ;

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		10

- вороги, проти який боротиметься персонаж.

Також для того, щоб додаток відповідав стандартам сьогоденних ігор, буде додано ще декілька елементів:

- 1) збереження ігрових сеансів для легкого повернення до гри через певний проміжок часу;
- 2) додавання декілька мов до гри;
- 3) налаштування гри під свій лад;
- 4) користувацький інтерфейс (UI) із зручним функціоналом:
 - головне меню додатку;
 - пауза під час ігрового циклу;
 - меню персонажа для керування предметами та параметрами.

Ігровий додаток матиме назву «Interim» та ігровий цикл представлятиме собою 2D платформер з видом збоку.

Графічним стилем гри та елементів стане 2D піксель-арт, приклад якого представлений на рисунку 1.1. В останні роки він набирає популярність серед ігор, створеними незалежними командами або одинаками-ентузіастами, адже він легко реалізовується та виглядає доречно як для комп'ютерних ігор. Те ж саме можна сказати і про анімацію.

Детальніше про функціонал гри буде описано в пункті 1.3.

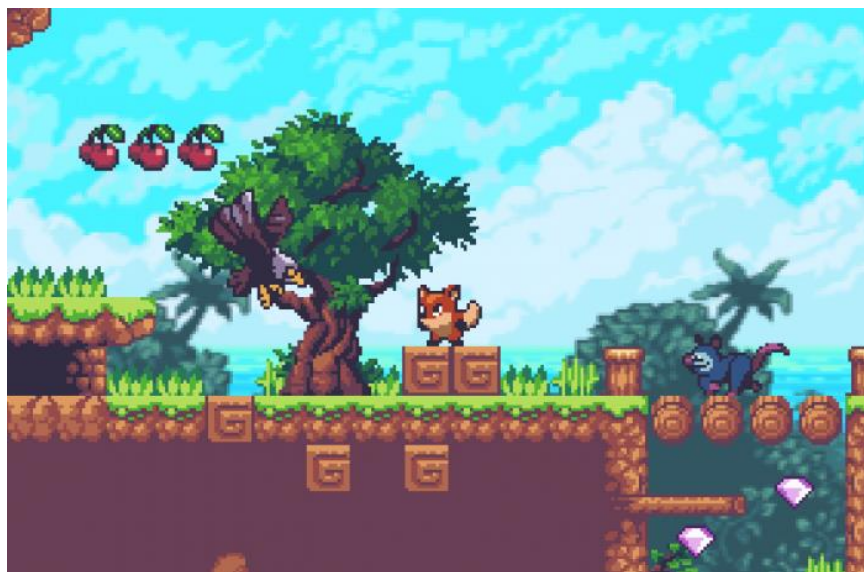


Рисунок 1.1 - Приклад графічного стилю в грі

1.2. Аналіз існуючого програмного забезпечення за тематикою кваліфікаційної роботи

Під час дослідження існуючих аналогів жанру «Метроїдванія» було розглянуто дві гри, що наявно представляють стандарти жанру в наш час: «Hollow Knight» та «Blasphemous».

1) Hollow Knight [2] (рис. 1.2).



Рисунок 1.2 - Скріншот ігрового циклу «Hollow Knight»

Розроблена студією Team Cherry. Гра відзначається великим, детально прописаним та розставленим різними елементами світом, що наповнений цікавими місцями та заселений незвичайними персонажами. Має прості, але вдалі ігрові механіки, вражаюче візуальне оформлення та атмосферний музичний супровід.

Головний герой отримує різні здібності та предмети, які він може використовувати для дослідження ігрової мапи та боротися із ворогами. Здібності базуються на русі персонажа (наприклад, ривок). Також персонаж може отримувати додаткові види атак, що можуть згодитись в бою.

Особливістю даної гри являються амулети – предмети, які екіпірує персонаж в обмеженій за обсягом певної сили кількості, які змінюють різні параметри персонажа або додають цікаві можливості або стани. Хоч і зброя тут одна, але саме амулети надають грі певної реграбельності – здатності гри надавати гравцям інший ігровий досвід за допомогою власних механік.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		12

Однак гравці виділяють недоліки в грі: частина амулетів гри незбалансовані та персонаж має певні проблеми з рухами, що в окремих ділянках гри є незручним у використанні.

2) Blasphemous [3] (рис. 1.3).

Ще один цікавий представник жанру Метроїдванія, розроблений іспанським студійним колективом The Game Kitchen. Гра відзначається темною та містичною атмосферою, що переповнена релігійними мотивами та візуальною реалізацією, що добре запам'ятовується.

Геймплей поєднує в собі елементи бойової системи з розгадуванням головоломок та дослідженням великого та світу гри. Головний герой володіє здібностями, які він використовує для боротьби зі злом та розкриття таємниць своєї долі. "Blasphemous" вражає якістю анімації та відповідністю міфологічному стилю, а також складним, але захоплюючим ігровим процесом.

Однак в грі виділяють малу різноманітність можливостей та предметів, котрі покращуватимуть їх, та певну одноманітність в ігровому циклі.

З вищеописаних ігор обраного жанру можна зробити висновок, що гравцям більше подобаються ігри з простими, але різноманітними та цікаво реалізованими механіками, предметами, цікавими рухами персонажа та проробленою ігровою мапою та сюжетом. Певні особливості представлених ігор були враховані для поточного ігрового додатку в розділі 1.1.

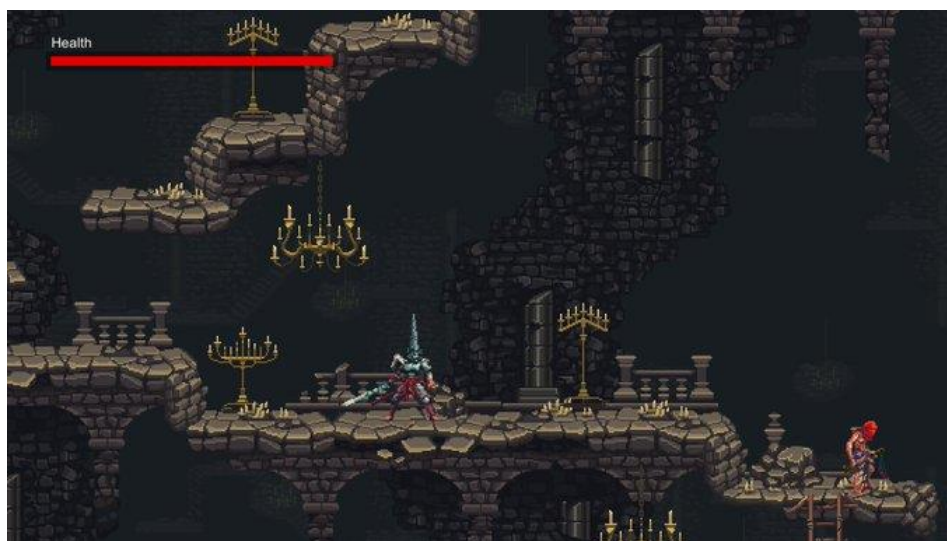


Рисунок 1.3 - Скріншот ігрового процесу гри «Blasphemous»

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		13

Для розроблюваної гри потрібно взяти до прикладу простоту механік та реалізацію ігрової мапи. Також для більшого різноманіття буде додано декілька видів зброї та можливість їх комбінувати, щоб надати гравцям кращий ігровий досвід.

1.3. Концепт-документ (Concept Document)

Ігровий додаток, котрий розроблятиметься для кваліфікаційної роботи, матиме назву «Interim». Гра відноситься до жанру «2D платформер» з видом збоку, піджанр - «Метроїдванія».

Складність – мідкор.

Сеттінг – технічний сюрреалізм. Графічний стиль – 2D піксель-арт.

Гра націлена переважно на англомовних та україномовних гравців 16-25 років, котрі люблять названі вище графічний стиль та жанр, а також які цікавляться інді – іграми від невеликих незалежних студій.

Додаток розроблятиметься на ПК з незначними обчислювальними можливостями. Для запуску необхідна дискретна відеокарта або один з передових вбудований графічний чіпів в процесор машини.

Користувачу (надалі - гравець) випадає можливість рухатись за головну героїню гри по локаціям, взаємодіяти з об'єктами, збирати предмети та боротись проти ворогів. Певні об'єкти мають можливість рухатись, активується та руйнуватись від дій гравця.

В користувача буде інвентар, в якому будуть відображені параметри персонажа на даному етапі гри та всі зібрані предмети. Деякі предмети екіпіруються на персонажа та дають нові здібності або покращують старі, а певні – застосовуються з певним ефектом та втрачаються. Всі предмети поділяються на зброю, аксесуари, їжу та сюжетні предмети.

Особливості гри:

- різноманітні здібності гравця, котрі даються з початку гри;
- нетипова здібність персонажа притягуватись до об'єктів на нитці, що робитиме ігровий процес більш захоплюючим для гравця;

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		14

- предмети матимуть унікальні ефекти, що заохочуватиме гравців досліджувати ігрові локації;
- GUI, що під час ігрового циклу буде прив'язаний до гравця, а не відокремленим вікном ігрового процесу.

1.4. Обґрунтування вибору інструментальних засобів

При розробці ігрового додатку спочатку потрібно обрати ігровий рушій, на якому будуть розроблюватись функціонал елементів гри, взаємозв'язок між різними елементами та сцени, на яких і будуть відбуватись дії ігрового циклу.

Для виконання завдань кваліфікаційної роботи доречно підходить рушій Unity [4], інтерфейс якого представлений на рисунку 1.4, котрий має необхідні інструменти для створення вищеназваних пунктів. Також їм користується багато розробників та людей, що цікавляться сферою, а тому в рушія є користувацька підтримка та в просторі Всесвітньої павутини можна знайти вирішення більшості задач. Це робить даний засіб фундаментом на початку роботи.

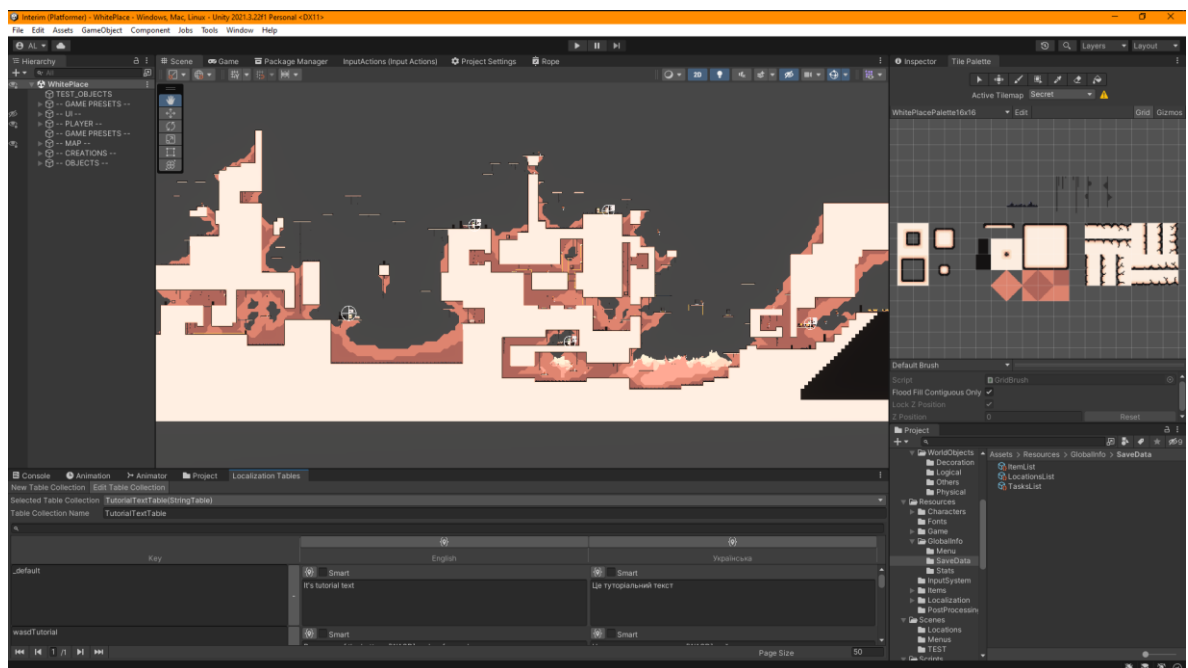


Рисунок 1.3 - Інтерфейс ігрового рушія Unity

Для написання програмного коду обраний ігровий рушій використовує об'єктно-орієнтовану мову програмування C# [5]. Дана мова має строгу

статичну типізацію, підтримує поліморфізм, перевантаження операторів, вказівники на функції-члени класів, події, властивості та інше. Функціоналу достатньо щоб реалізовувати патерни проектування, котрі будуть вкрай необхідні для ігрового додатку.

Середовище розробки Rider від GetBrains [6], інтерфейс якого представлений на рисунку 1.5, дозволить зручно писати необхідний код. Дане середовище має підтримку рушія Unity, що полегшує роботу з його бібліотеками за допомогою підказок, а також аналізує використання полів, методів та іншого на поточній сцені рушія для можливого дебагінгу – процесу виправлення програмного коду, котрий працює з похибкою від початково очікуваних результатів.

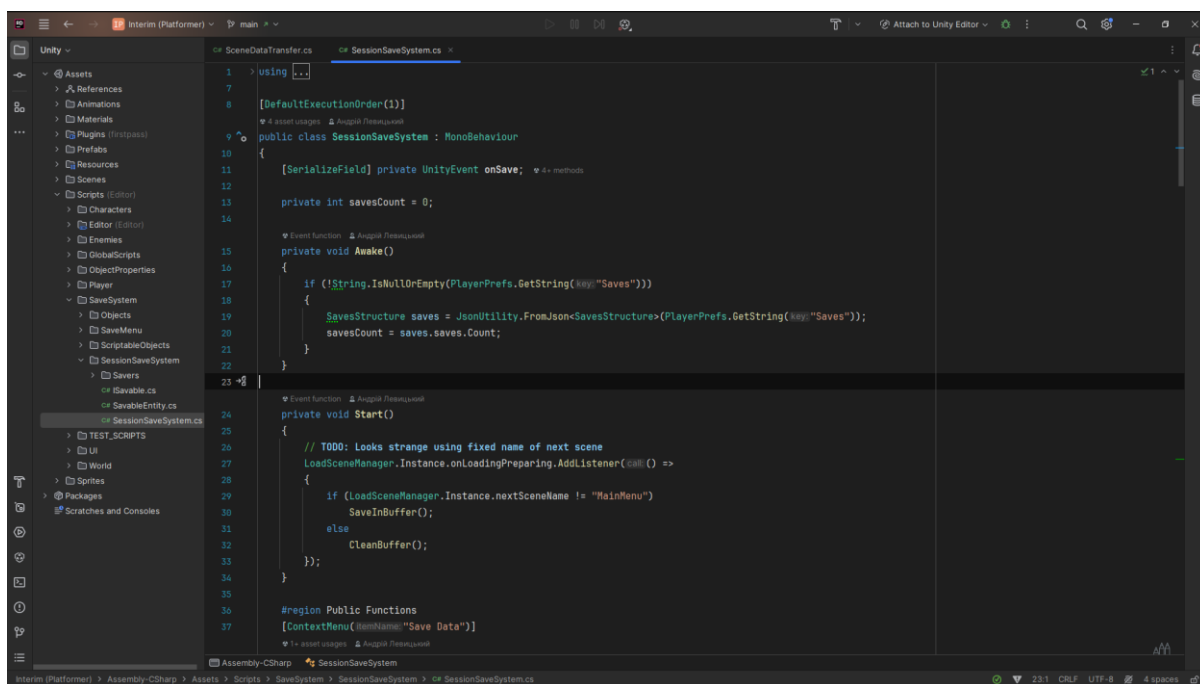


Рисунок 1.4 - Інтерфейс середовища розробки Rider

Для збереження ресурсів ігрового додатку та фіксування їх змін по мірі розробки використовуватиметься веб сервіс спільного керування версіями GitHub, а точніше його зручна версія для персональних комп'ютерів GitHub Desktop [7], інтерфейс якого представлений на рисунку 1.5. Він дозволяє зберігати прогрес розробки в будь-який момент часу з можливістю переглянути минулі версії та їх зміни. При некоректній роботі поточної версії програма дає можливість повернутися до минулої, що робить розробку безпечнішим.

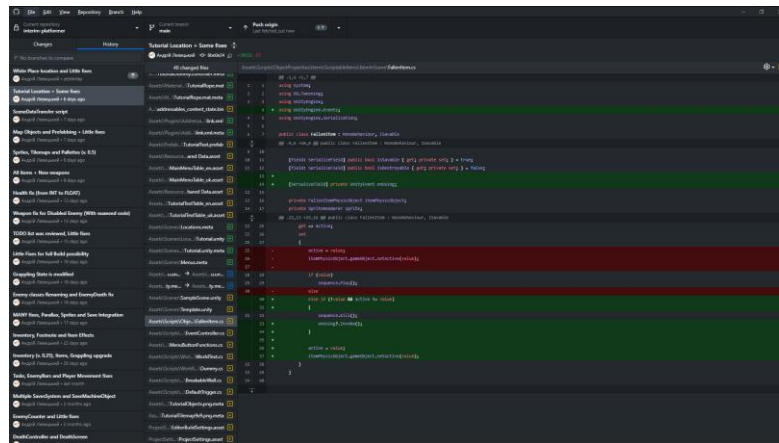


Рисунок 1.5 - Інтерфейс програми спільного керування версіями GitHub Desktop

Так як графічна частина ігрового додатку виконуватиметься в стилі «піксель-арт», буде використано спеціальний графічний редактор – Aseprite [8], що представлений на рисунку 1.6. Його інструменти розроблені для малювання на малорозмірних полотнах без ефекту згладжування, як це роблять багатфункціональні редактори.

Також він має відкритий початковий код, що дозволяє його користувачам розробляти надбудови, які додають новий функціонал або покращують наявний. Для кращої творчої роботи над графічними елементами гри було додано надбудову PixelLab [9], котра використовує штучний інтелект для генерування піксельних зображень за запитами.

Надбудова є експериментальною, тому згенеровані зображення будуть оброблені вручну, також частина графіки та елементи графічного інтерфейсу програми будуть виконані без застосування штучного інтелекту.

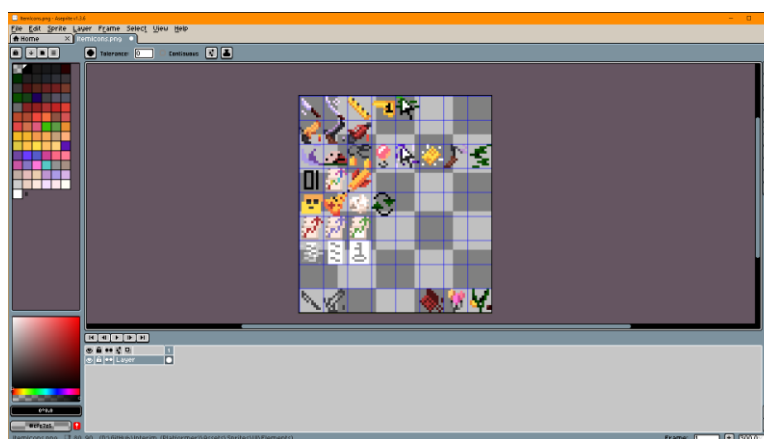


Рисунок 1.6 - Інтерфейс графічного редактора Aseprite

Висновки до першого розділу

В розділі було визначено ряд вимог, які будуть виконані під час розробки кваліфікаційного проекту.

Для додатку було обрано жанр «Метроїдванія» та вказано основні його особливості: персонаж з різноманіттям можливостей, котрі отримуються впродовж всієї гри, локації з секретами, перешкодами та шляхами, що відкриватимуться з розвитком персонажа, предмети з впливом на персонажа та вороги, що також є складовою перешкод. Щоб додаток відповідав ігровим тенденціям сьогодення, було вказано збереження, інтерфейс, локалізація тексту та налаштування як його необхідні складові.

Було проаналізовано два аналога-представника обраного жанру. Гра «Hollow Knight» є гарним прикладом жанру та відзначається простотою вмінь персонажа, проробленим світом, бійками та широким різноманіттям спеціальних предметів для вмінь, однак персонаж гри має певні недоліки в рухах. В іншому аналогу під назвою «Blasphemous» також є пророблений світ, бійки, детальні анімації та сюжет, однак відсутня різноманітність в ігровому циклі та предметах, котрі даватимуть цікаві ефекти.

На основі отриманих даних був сформований концепт-документ, який також описує особливості додатку: можливість притягуватись персонажем до стін за допомогою здібностей, більшість предметів матимуть вплив на персонажа, а також GUI в ігровому циклі з'являтиметься навколо персонажа. Гра отримала назву «Interim», матиме візуальний стиль 2D піксель-арт та розроблятиметься для ПК.

Було визначено стек технологій, який використовуватиметься для розробки. Сам проект кваліфікаційної роботи розроблятиметься на ігровому рушії Unity з використанням мови програмування C#. Програмний код буде реалізовуватись в середовищі розробки Rider. Для керування версіями використовуватиметься GitHub Desktop. Графічні елементи додатку будуть виконані в спеціальному графічному редакторі Aseprite, частина з яких створюватимуться частково за допомогою штучного інтелекту.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						18
Змн.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1. Проектування архітектури ігрового додатку

Перед тим як розпочати реалізацію поставлених в минулому розділі вимог, слід виконати проектування архітектури та алгоритмів роботи ігрового додатку щоб мати точне представлення в якому напрямку потрібно виконувати розробку.

Для початку почнемо з діаграми використання, котра візуалізує функціональності системи та взаємодії між її компонентами та зовнішніми чинниками (у випадку додатку - гравцями). Враховуючи описані в минулому розділі дані можна сформуванати представлену на рисунку 2.1 діаграму.



Рисунок 2.1 - Діаграма використання ігрового додатку

Отримана діаграма дає зрозуміти, що певні елементи відносяться лише до ігрового циклу, а інші можуть бути виконані як при циклі, так і в початковому меню перед самою грою, тому зробимо висновок, що в грі буде два стани: головне меню та ігровий цикл.

Далі доречним буде змодельовати дві діаграми діяльності для кожної зі станів гри. Діаграма діяльності візуалізує та описує послідовності дій, що відбуватимуться в системі або в процесі виконання конкретної функції. Вона дозволяє моделювати бізнес-процеси, логіку алгоритмів та потік управління в програмних системах.

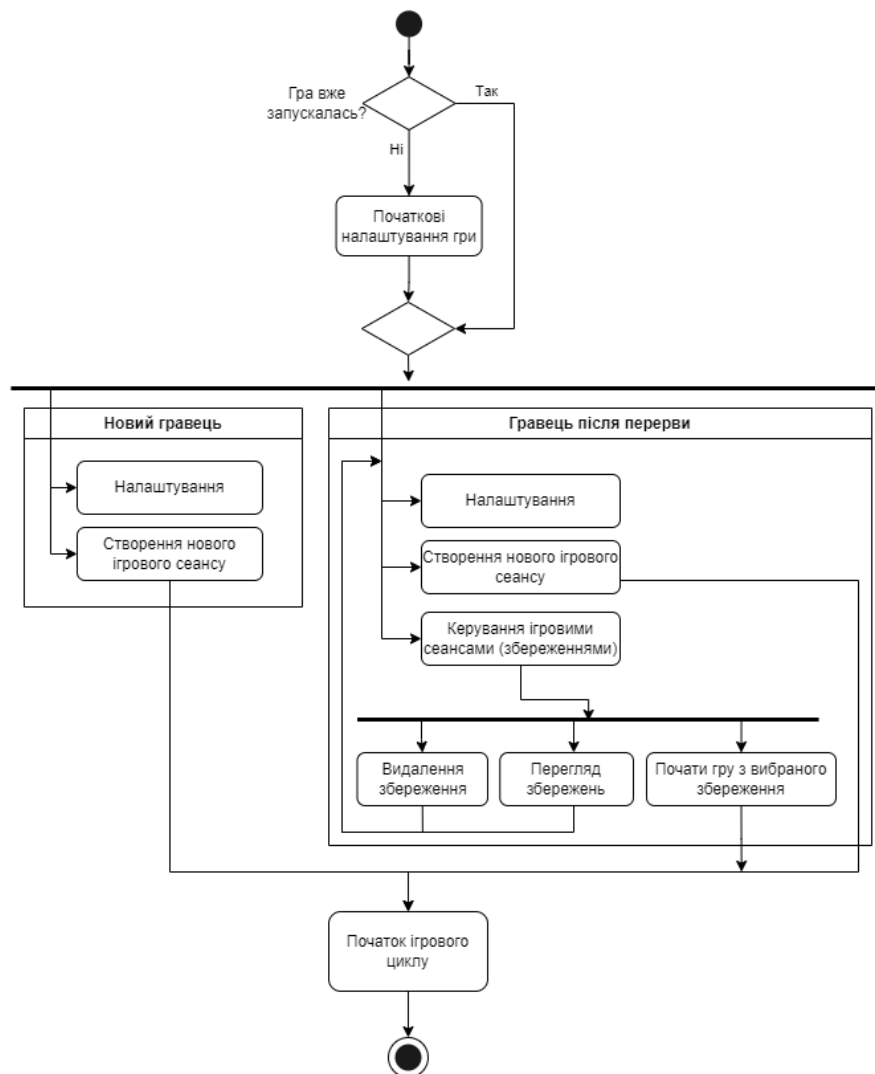


Рисунок 2.2 - Діаграма діяльності гравця при головному меню

Діаграма діяльності для головного меню (рис. 2.2) точніше описує функціонал однойменного меню гри, з яким взаємодіятимуть гравці. Також в ньому відображено можливі дії гравців, що запустили гру вперше, та гравців, що продовжують грати після певної перерви.

Подібна діаграма для ігрового циклу (рис. 2.3) дозволяє детальніше ознайомитись з тим, які дії доступні гравцям під час гри. Певні пункти дій внесені в окремі логічні пункти, щоб було зрозуміліше розбирати функціонал того чи іншого вибору.

Як можна побачити, до діаграми додана робота зі збереженнями. Це стосується не лише того, що гравець має можливість зберігатись в окремо відведених точках на мапі, а й зручному виборі збереження гравцем під час самого ігрового циклу.

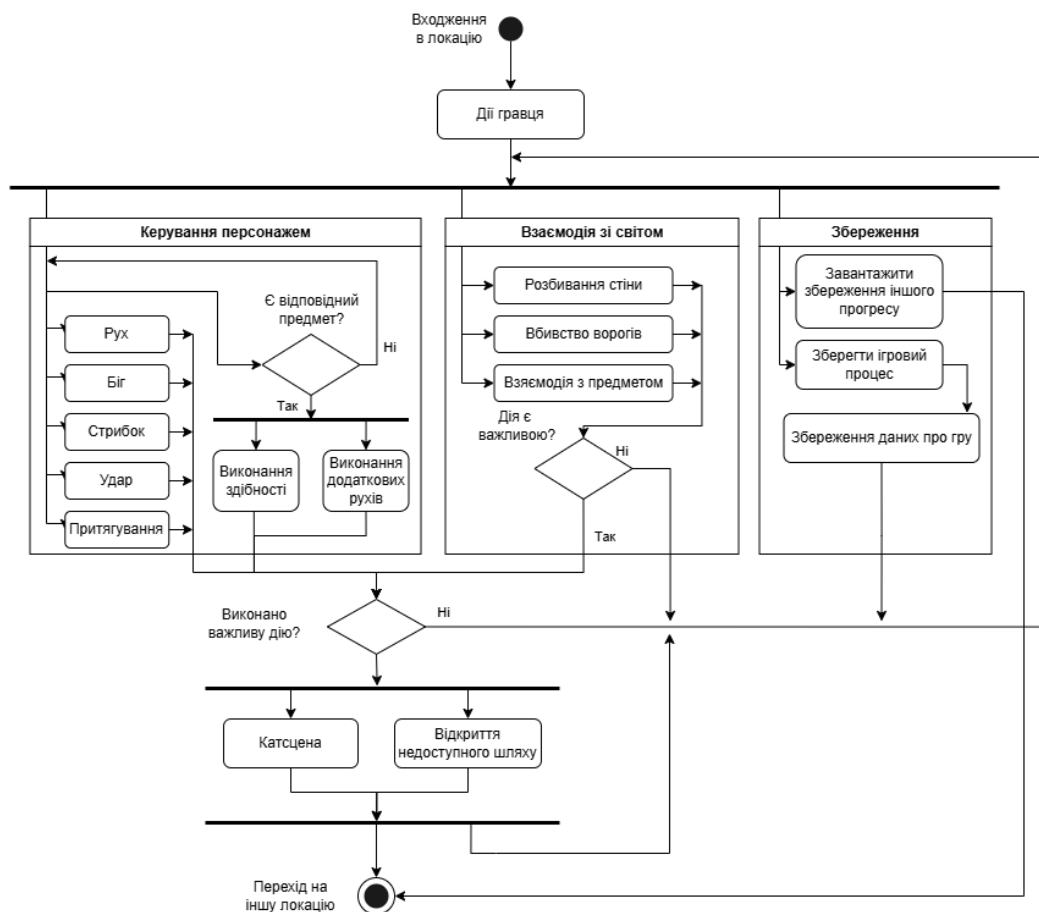


Рисунок 2.3 - Діаграма діяльності гравця під час ігрового циклу

Також змодельуємо діаграму розгортання ігрового додатку (рис. 2.4) для повного представлення того, з якими ресурсами потрібно працювати під час розробки. Це дозволить підготувати дані про можливості їх взаємодії за допомогою ігрового рушія, а також ознайомитись з кожним задіяним ресурсом.

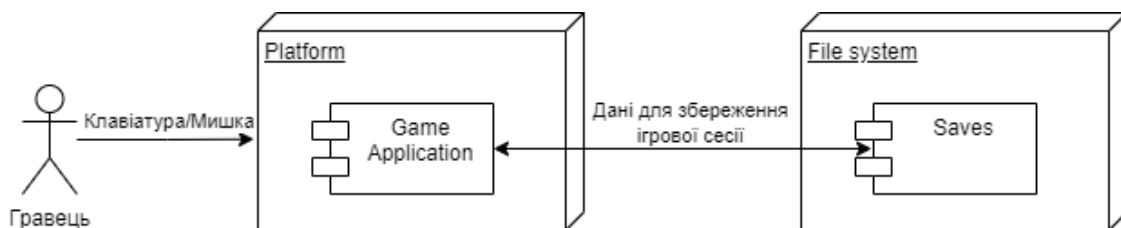


Рисунок 2.4 - Діаграма розгортання ігрового додатку

Діаграма дає зрозуміти, що під час розробки необхідно врахувати роботу ігрового додатка з файловою системою гри. Це зумовлено тим, що кожне збереження буде представлене як окремий файл, до якого записуватимуться необхідні дані для його завантаження.

Зі змодельованих раніше діаграм можна скласти список функцій, котрі потрібно реалізувати в ігровому додатку під час розробки.

Головне меню матиме наступні функції:

- початок нового ігрового сеансу;
- завантаження останнього збереженого сеансу, якщо такий є;
- перегляд існуючих збережень та вибір необхідного для продовження минулого бажаного сеансу;
- налаштування:
 - зміна розмірів вікна;
 - перемикання повноекранного режиму;
 - зміна мови;
 - скидання прогресу гри;
- вихід з гри.

Ігровий цикл буде складатись з наступних елементів та функцій:

- персонаж та керування ним:
 - рух вправо та вліво;
 - стрибок;
 - ривок вправо та вліво;
 - ковзання по стіні;
 - закріплювання на стіну;
 - стрибок від стіни;
 - притягування до поверхонь;
 - використання зброї на праву та ліву кнопку миші;
- вороги та алгоритм їх роботи:
 - рух вправо та вліво (наземні вороги);
 - рух в повітрі в різні боки (надземні вороги);
 - відслідковування персонажа в полі зору;
 - виконання атаки на персонажа;
- фізичні об'єкти на локації:
 - локація, складена з тайлів – графічних частинок, що формують цільне зображення;
 - блок, що рухається від дій гравця;

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						22
Змн.	Арк.	№ докум.	Підпис	Дата		

- елементи, що активується від дій гравця;
- елементи, що є цілями активації попередніх;
- шипи, що моментально вбивають ворогів та персонажа;
- логічні об'єкти на локації:
 - області, що виконують певні дії при вході та виході з неї;
 - кинуті предмети, які можна підібрати;
 - точки збереження прогресу ігрового сеансу;
- предмети:
 - можна екіпірувати (зброя та аксесуари);
 - можна застосувати;
 - відкриті лише для перегляду опису;
- пауза:
 - налаштування ігрового додатку;
 - перегляд та вибір наявних збережень;
 - вихід до головного меню;
- меню персонажа:
 - перегляд параметрів гравця;
 - перегляд інвентарю та предметів в ньому;
 - взаємодія з предметами;

Наступні пункти будуть описувати хід розробки, реалізацію представлених раніше пунктів та програмні рішення, такі як.

2.2 Розробка програмної частини елементів ігрового циклу

Розробку першочергово потрібно почати з головних елементів ігрового циклу, з котрих і будуть сформовані наступні частини додатку.

Такими елементами вважатимуться:

- персонаж;
- вороги;
- зброя;
- предмети взаємодії;

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						23
Змн.	Арк.	№ докум.	Підпис	Дата		

- логічні об'єкти на локації.

Персонаж та вороги будуть мати однакову програмну реалізацію, тому вони розглядатимуться в одному пункті. Зброя також буде реалізована однаково для ворожих сутностей та сутності, якою керує гравець, однак вона винесена в окремий пункт, адже його структура є широкою, що буде представлено далі по тексту.

Предмети взаємодії та логічні об'єкти на локації також будуть об'єднані в один пункт, адже їх реалізація не є складною, враховуючи наявний функціонал рушія Unity.

2.2.1. Програмна реалізація персонажа та ворогів

З переліку можливостей персонажа та ворога, що описані в пункті 2.1, можна зробити висновок, що реалізовувати весь функціонал в єдиний клас є дуже недоцільним, особливо на довготривалу перспективу, тому буде використано поведінковий патерн проектування Стан [10].

Діаграма класів реалізації описаного патерна представлена на рисунку 2.5. На ньому знаходяться всі основні класи, які необхідно реалізувати для початку розробки функціоналу.

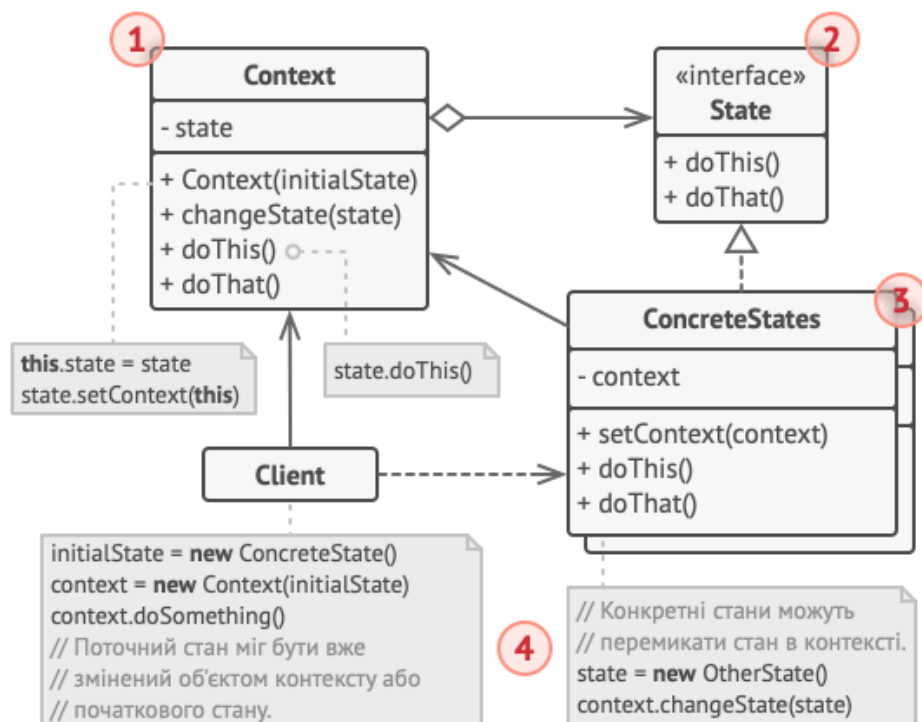


Рисунок 2.5 - Діаграма класів реалізації патерна Стан

Обраний патерн використовується для реалізації кінцевого автомату станів (state machine) в програмному коді. Головна ідея полягає в тому, щоб дозволити об'єкту змінювати свою поведінку за допомогою перемикання власних станів між собою. Такий об'єкт виглядає так, ніби він змінив свій клас.

В програмному коді це реалізовано за допомогою класів State та StateMachine.

Перший клас є узагальненим станом – це абстрактний клас, який визначає спільні характеристики для всіх існуючих станів для персонажа та ворогів. Також є подібні стани, які унаслідують представлений клас, але який є батьківським для певної групи станів щоб запобігти повторенню коду. Програмний код класу представлений нижче.

```
using UnityEngine;

public abstract class State
{
    protected StateMachine stateMachine;
    protected StatsController stats;

    protected AnimatorController animatorController;

    protected bool isAnimationFinished;
    protected bool isExitingState;

    protected float startTime;

    public virtual void Enter()
    {
        DoChecks();
        startTime = Time.time;
        isAnimationFinished = false;
        isExitingState = false;
    }

    public virtual void Exit()
    {
        isExitingState = true;
    }

    public virtual void LogicUpdate() { }

    public virtual void PhysicsUpdate()
    {
        DoChecks();
    }
}
```

```

public virtual void DoChecks() { }

public virtual void AnimationTrigger() { }

public virtual void AnimationFinishTrigger() =>
isAnimationFinished = true;
}

```

Клас StateMachine – винесена логіка роботи зі станами. Через нього відбувається заміна одного стану іншим з необхідною підготовкою та є можливість отримати поточний та минулий стан.

Для його роботи потрібно створити поля станів та поле власне машини станів в класі, виконати ініціалізацію його роботи, відправивши йому початковий стан, та вписати виконання фізичного та логічного оновлення поточного стану, що зберігається в класі, в подібні функції класу сутності.

Сам клас представлений нижче.

```

using System;

[Serializable]
public class StateMachine
{
    public State CurrentState { get; private set; }
    public State PreviousState { get; private set; }

    public string stateName;

    public void Initialize(State state)
    {
        CurrentState = state;
        CurrentState.Enter();
    }

    public void ChangeState(State state)
    {
        CurrentState?.Exit();
        PreviousState = CurrentState;
        CurrentState = state;
        CurrentState.Enter();

        stateName =
CurrentState.GetType().ToString();
    }
}

```

З представленням того, як виглядає та працює патерн Стан, можна описати схему роботи всіх реалізованих станів для персонажа та ворогів.

Стани здібностей відповідають за короткочасне виконання певної дії під час введення необхідного сигналу від гравця, які можна виконувати в будь-який момент часу, якщо є можливість їх задіяти та якщо ситуація відповідає описаним умовам. В минулих описаних групах також було зчитування введень гравця, однак в такому випадку стан залишався одним і тим ж самим, в той час як здібності виконуються лише мить або малий відведений час. Наприклад, при виконанні стрибка на об'єкт персонажа прикладується певна сила, котра направляє його вгору, і для цього потрібен лише проміжок між двома кадрами, а для виконання удару персонаж переходить до цілої системи, котра відповідає за зброю, під час чого стан виконується до тих пір, поки система не закінчить свою роботу. Система зброї буде описана згодом в наступному пункті розділу.

Для кожної групи створений власний узагальнюючий стан, в котрому описаний функціонал, що є однаковим для кожного стану групи, а також перевірки, які потрібні для визначення моменту, в який потрібно перемкнути стан на інший.

Однак це не стосується стану в повітрі. Він не відноситься ні до однієї з груп, тому він є громіздким через перевірки, але його роль в схемі є суттєвою. Навіть під час падіння персонажа з висоти вмикається цей стан. Для здібностей це також стосується, адже вони можуть виконуватись під час польоту.

Приклад реалізованого батьківського узагальненого класу для наземних станів та один з їх представників, що виконує рух гравця вправо та вліво, представлено в додатках А та Б відповідно.

Кожен зі станів персонажа зчитує сигнали, які вводить гравець, тому доречно їх всі винести в окремий клас.

Таким класом слугує `PlayerInputHandler`, котрий представлений в додатку В. Він отримує всі необхідні сигнали та конвертує їх в публічні поля типу `bool` або `float`, які, в свою чергу, зчитуються станами для власної роботи.

Однак цей клас скоріше відповідає патерну Адаптер [11] – структурному патерну проектування, що дає змогу об'єктам з несумісними інтерфейсами працювати разом. В цьому випадку клас отримує сигнали, обробляє їх та в

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		28

зручному вигляді дає станам персонажа. Але ще не було розібрано де власне беруться сигнали.

Для зручного розподілення сигналів з миші та клавіатури в конкретні найменування було використано надбудову рушія Unity під назвою Input System. В його налаштуваннях можна вказати декілька сигналів з різних популярних способів введення в ПК, такі як миша, клавіатура, геймпад та подібні, в єдине поле, якому можна змінювати тип даних: двовимірний вектор, дробове значення, значення типу bool та інші доступні.

Список можливих введень в грі представлений на рисунках 2.7 та 2.8.

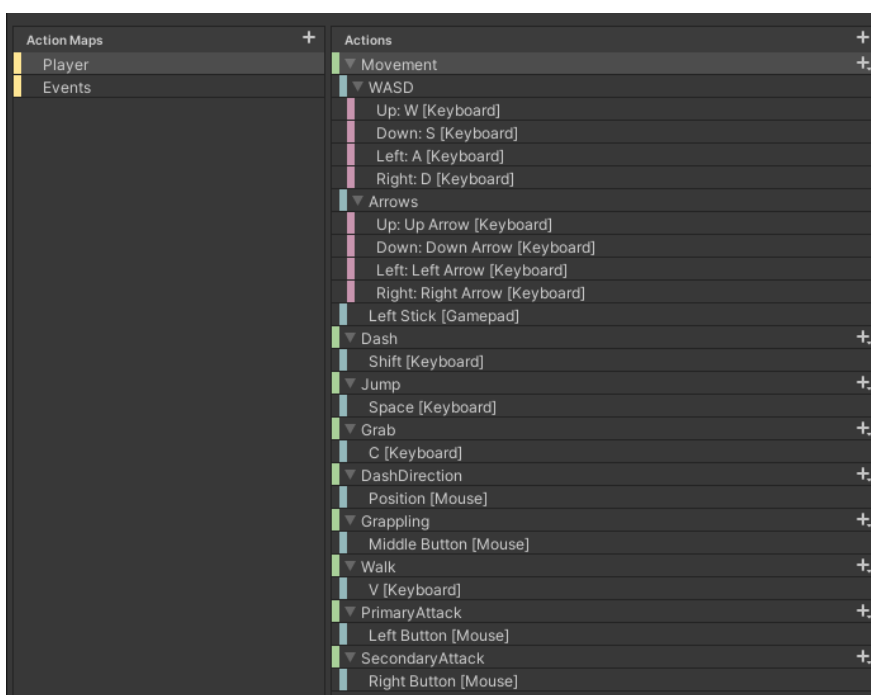


Рисунок 2.7 - Список найменувань груп сигналів для керування гравцем

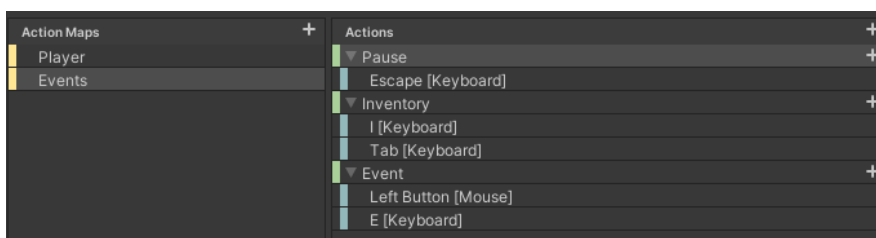


Рисунок 2.8 - Список найменувань груп сигналів для виклику меню

Для коректної роботи сигналів розробник підписує методи класу PlayerInputHandler до відповідних найменувань та обробляє отримані сигнали в описаний раніше вигляд. Це є зручнішим, аніж реалізація зчитування сигналів з надбудови напряму станом персонажа.

Далі перейдемо до систем поменше, а точніше підсистеми сутностей, які є не менш важливими для персонажа та ворогів. Серед них є:

- контролер параметрів;
- контролер здоров'я;
- клас визначення зіткнень;
- приймач відкидування;
- контролер руху;
- інвентар (лише для персонажа).

Деякі підсистеми були зроблені спеціально для роботи окремої структури роботи зброї, яка притаманна як персонажу, так і ворогам, однак вони розглядатимуться в наступному пункті розділу.

Перед початком розбору кожного із систем розглянемо їх взаємодію із основними класами сутностей. Для роботи з ними потрібно зберігати посилання на їх екземпляри в різних полях класу, а це є доволі громіздким рішенням.

Для запобігання цьому використано структурний патерн проектування Фасад [12], який надає простий інтерфейс до складної системи класів, бібліотеки або фреймворку. В цьому випадку клас буде зберігати список класів та повертати потрібний клас за допомогою універсального методу [13], тобто потрібно вказати точний клас щоб його отримати з класу.

Таким фасадом стане клас CoreSystem, лістинг якого представлений в додатку В, а класи вказаних вище підсистем будуть унаслідувати спеціального батька CoreComponent, що представлений нижче.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CoreComponent : MonoBehaviour, ILogicUpdate
{
    protected CoreSystem core;

    protected virtual void Awake()
    {
        core = transform.parent.GetComponent<CoreSystem>();
    }
}
```

```

        if (core == null)
            Debug.LogError("There is no Core on the parent");

        core.AddComponent(this);
    }

    public virtual void LogicUpdate() { }
}

```

Для отримання всіх підсистем фасад отримує їх з окремих дочірніх об'єктів. Приклад реалізації подібної системи можна побачити на рисунку 2.9.

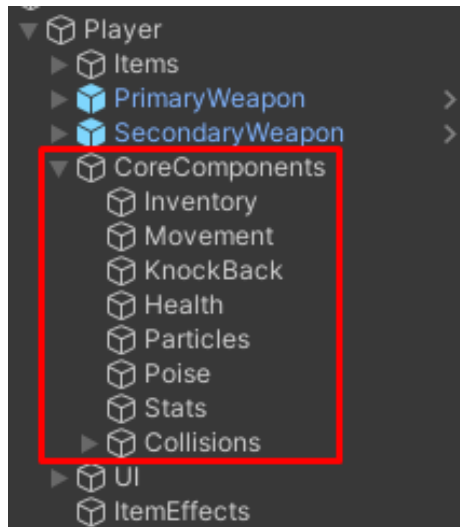


Рисунок 2.9 - Приклад реалізації фасаду та дочірніх підсистем на сцені

В класі CoreSystem (додаток В) можна побачити поле Controller, що має інтерфейс ICharacterController. Воно зберігає клас сутності, до якого він доданий, а інтерфейс реалізує всі необхідні методи, якими можуть скористатись підсистеми. Лістинг інтерфейсу представлений нижче.

```

public interface ICharacterController
{
    void StartDamage();
    void FinishDamage();
    void DoDeath();
    bool IsDead();
}

```

Після розбору взаємозв'язку між підсистемами та сутностями можна перейти до розбору кожного з них.

Першочергово слід розглянути підсистему параметрів. Для неї буде використано структурний патерн проектування Декоратор [14], що дає змогу динамічно додавати об'єктам нову функціональність, загортаючи їх у корисні

«обгортки». Діаграма класів для обраного патерна представлено на рисунку 2.10.

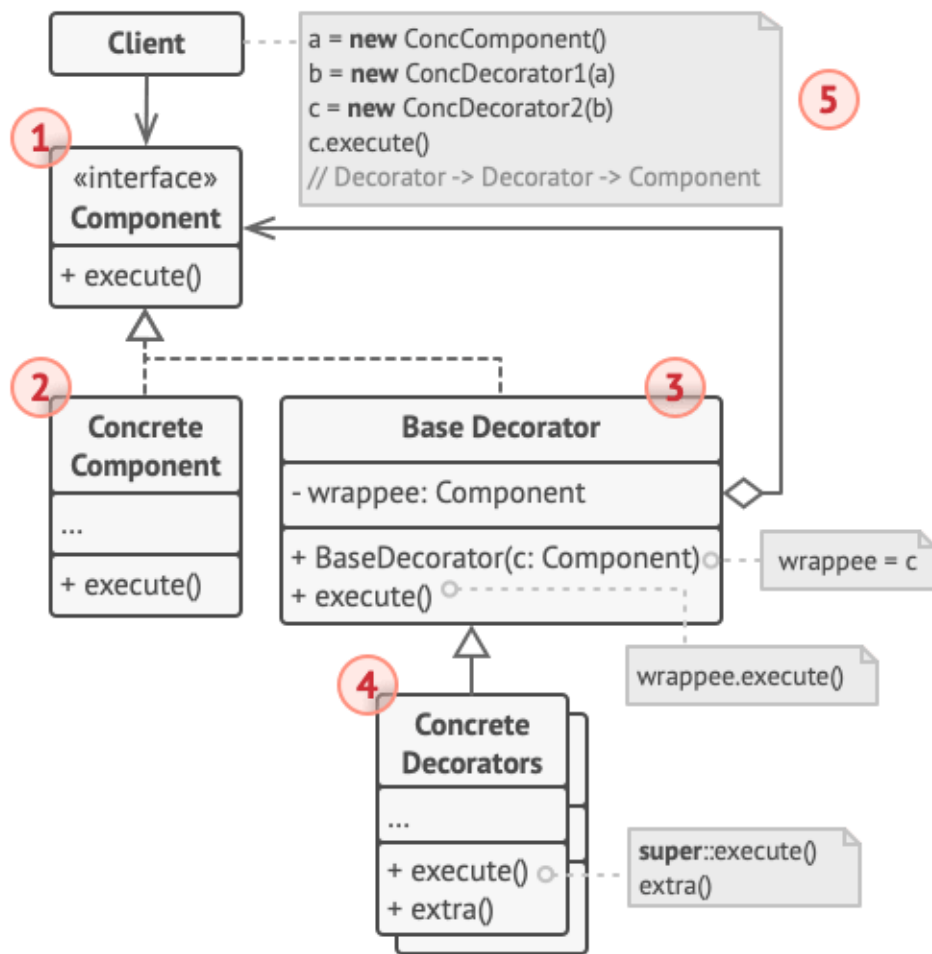


Рисунок 2.10 - Діаграма класів для патерну проектування Декоратор.

У випадку додатка це буде певне поле класу, яке зберігатиме поточні параметри сутності. Воно буде мати спеціальну структуру даних, що описуватиме всі поля, які можуть змінюватись. При накладанні на сутність ефекту від удару, використання предмета або інших дії, який в собі має подібне поле зі значеннями, які змінюватимуться в оригінальних параметрах, підсистема отримує це поле та додає до поточних параметрів як «обгортку».

Клас підсистеми має назву StatsController та представлений в додатку Г.

Структура даних параметрів називається CharacterStats, лістинг якого знаходиться в додатку Д, однак без коду перевизначення операторів додавання, віднімання, множення та ділення, тому що вони вказують як будуть діяти оператори на поля двох структур. Також там висвітлена структура даних

StatsModifier, до якого, окрім параметрів, додане поле, що вказує його дію як «обгортку» на параметри: примножувати поля чи додавати до полів значення.

Нижче представлений власне базовий клас StatsDecorator, що й виконує додавання та застосування «обгортки» до параметрів, та інтерфейс IStatsProvider, який реалізовується базовим класом.

В додатку Е представлено клас ModifiedStats, що унаслідкує базовий клас. В ньому зберігається список всіх «обгортки», котрі були додані до сутності, та при отриманні параметрів застосовує їх до параметрів.

```
public interface IStatsProvider
{
    CharacterStats GetStats();
    abstract void AddModifier(StatsModifier modifier);
    abstract void RemoveModifier(StatsModifier modifier);
}

public abstract class StatsDecorator : IStatsProvider
{
    protected readonly IStatsProvider _wrappedEntity;

    protected StatsDecorator(IStatsProvider wrappedEntity)
    {
        _wrappedEntity = wrappedEntity;
    }

    public CharacterStats GetStats()
    {
        return GetStatsInternal();
    }

    protected abstract CharacterStats GetStatsInternal();
    public abstract void AddModifier(StatsModifier
modifier);
    public abstract void RemoveModifier(StatsModifier
modifier);
}
```

Далі розглянемо контролер здоров'я. Ця підсистема зберігає поточне та максимальне значення життя сутності та має методи зміни цього значення.

Однак описувати функціонал цього значення в середині класу підсистеми є не кращою ідеєю, адже в подальшому подібна модель поведінки значення може пригодитись, наприклад, для керування значенням оглушення сутності ударом. Для кращого розуміння роботи значення наведено наступний список властивостей:

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						33
Змн.	Арк.	№ докум.	Підпис	Дата		

- значення не може бути від'ємним;
- значення не може перевищувати максимальне значення;
- відбувається перевірка значення кожен раз, як воно змінюється;
- при від'ємному значенні або максимальному викликатиметься подія, на яку зможуть підписуватись класи;

Для контролю такою логікою буде створено окремий клас під назвою Stat, що виконує всі вищевказані властивості, зберігає максимальне значення та керує поточним. Лістинг коду наведено нижче.

```
using System;
using UnityEngine;

[Serializable]
public class Stat
{
    public event Action OnCurrentValueZero;
    [SerializeField] private float maxValue;
    private float currentValue;

    public float MaxValue
    {
        get => maxValue;
        set
        {
            maxValue = value;
            if (currentValue > value)
                currentValue = maxValue;
        }
    }
    public float CurrentValue
    {
        get => currentValue;
        private set
        {
            currentValue = Mathf.Clamp(value, 0f, MaxValue);
            if (currentValue <= 0f)
                OnCurrentValueZero?.Invoke();
        }
    }
    public void Init() => CurrentValue = MaxValue;
    public void Increase(float amount) => CurrentValue += amount;
    public void Decrease(float amount) => CurrentValue -= amount;
    public void SetMaxValue(float amount) => MaxValue = amount;
}
```

Клас підсистеми здоров'я має назву HealthController та представлена в додатку Ж. Вона виконує ініціалізацію вищеописаних класів, змінює значення життя в ньому, виконує події на різні дії зі здоров'ям, керує системою часток та робить смерть при настанні нульового значення. Також підсистема керує системою шкали здоров'я, відправляючи йому дані, однак вона розглядатиметься в пункті про GUI.

Далі на черзі йде проста підсистема перевірки зіткнень, яка перевіряє факт зіткнення сутності з землею, стіною або стелею. Вона визначає зіткнення за допомогою відслідковування потрібного елемента на певній дистанції або в межах конкретного радіусу.

Для точнішої перевірки присвоюватимуться точки для кожного виду перевірки відносно центра сутності. Від них будуть виконуватись аналізи факту зіткнень.

Клас підсистеми має назву CollisionSenses. Отримання від неї факту зіткнення відбувається через зчитування полів типу bool, які в свою чергу повертають результат аналізу зіткнення за допомогою функціоналу класу Physics2D [15], а точніше функцій OverlapCircle() та Raycast().

Наступними розглядатимуться підсистеми руху та відкидування для пояснення для яких цілей був створена ще одна програмна реалізація переміщення гравця.

Підсистема відкидування відповідає за приймання від снаряду або різних ударів зброєю ефект, який відкидає сутність в бік. Для його роботи він блокує систему станів сутності, не даючи йому шанс на контрольований рух, та прикладає на нього силу, яку зброя передає у вигляді двовимірного вектору.

Реалізація підсистеми є клас KnockBackController і в розділ додатків кваліфікаційної роботи також не виноситься. Однак слід вказати, що він реалізує інтерфейс IKnockBackable, який вказує на необхідність мати метод KnockBack(). Таке рішення буде пояснене під час розбору зброї в грі.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		35

```
using UnityEngine;

public interface IKnockBackable
{
    void KnockBack(Vector2 angle, float strength, int
direction);
}
```

Для прикладання потрібної сили на сутність з різних підсистем, в тому числі з приймача відкидань, або класів, які не пов'язані з системою станів, і був створений контролер руху. Він описує основні способи прикладання сили та використовується у випадках, коли система станів є заблокованою в даний момент часу. Таке стається якраз під час відкидання сутності та, забігаючи наперед, використання зброї.

Назва класу підсистеми – MovementSet. Вся робота підсистеми зводиться до збереження поточної прикладеної сили, зміни сили на іншу та повороту сутності, а також роботі з класом Rigidbody2D [16], а точніше з його полем velocity, в яке і записується прикладена сила.

І на останок, розгляд підсистеми інвентарю персонажа. Серед сутностей він може бути лише у персонажа, яким керує гравець. Інвентар зберігає всі дані про підібрані предмети та предмети, що були екіпіровані на персонажа.

Предмети мають власний клас ScriptableItem, що представлений нижче, який має батьківський клас ScriptableObject [17], тобто екземпляри класу можна створювати поза грою в окремі файли. Кожен об'єкт класу має назву, опис, малюнок та параметри, які можна застосовувати як модифікатори в підсистемі, а також список об'єктів типу ItemEffect, але про них згодом.

```
using System.Collections.Generic;
using UnityEngine;

public class ScriptableItem : ScriptableObject
{
    public string itemName;
    public ItemType type;
    public string description;
    public List<string> additional;
    public List<ItemEffect> initializers;
    public Sprite icon;
    public Color backgroundColor;
    public StatsModifier stats;
}
```

За реалізацію підсистеми інвентарю відповідає клас InventoryController, який представлений в додатку II. Його функціонал полягає в додаванні та зберіганні предметів, екіпірування тих предметів, що являють собою зброю або аксесуар, а також застосування і видалення предметів, що відносяться до класу їжі. Також в інвентарі можуть бути додаткові предмети, які слугують лише інформацією для гравця.

При екіпіруванні або застосуванні предметів інвентар передає модифікатор параметра та передає його в підсистему параметрів, який виконує патерн Декоратор, як вже це було описано раніше.

Однак нещодавно було згадано список об'єктів типу ItemEffect. Це батьківський клас для створення різного виду ефектів для сутності, як от додавання додаткового модифікатора параметрам, відновлення здоров'я та інше. Вони зберігаються як окремі файли налаштованих об'єктів – префаби [18], які в подальшому можна використати для їх додавання до окремого дочірнього об'єкта сутності. Ефекти можуть бути як постійні, так і тимчасові, котрі видаляються з часом.

Ці ефекти додає інвентар, однак за допомогою класу ItemEffectSpawner, який виконує ініціалізацію об'єкта з ефектом та зберігає його в список, який можна отримати за відповідним ключем – предметом, який цей ефект і наклав. Якщо ефект тимчасовий, він видаляється зі списку, до якого він належить. Якщо предмет в інвентарі знятий з екіпірування, всі ефекти, що з ним пов'язані, видаляються.

Описані раніше підсистеми так чи інакше використовуються класом сутності або самостійно використовують його методи для реалізації їх призначення.

Для персонажа власний клас несе назву Player. В ньому виконується ініціалізація всіх можливих станів та машина станів, яка виконує перемикання між ними. Окрім цього клас має власні можливості прикладати на персонажа силу та перевіряти певні зіткнення. Це зроблено через те, що підсистеми, які виконують подібні дії, зроблені для відокремленої роботи із сутністю зі

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		37

сторонніх джерел, які не пов'язані зі станами, та мають властивість вимикатись в разі зникнення потреби.

Вороги мають подібну систему, однак через їх різноманітність цей клас є абстрактним, який узагальнює методи, що можуть виконуватись в їх роботі. В конкретних класах, скажімо, наземних ворогів, вже визначаються власні стани, які також є похідними від абстрактних, а також методи обрахунку гравітації, способу прикладання сили на ворога та перевірок зіткнень, адже для наземних ворогів та летючих логіка поведінки є різною.

Також для ворогів існує власний тип даних EnemyData, що унаслідкує ScriptableObject. В ньому визначені параметри поведінки: відстань для відслідковування персонажа та нападу на нього, час на виконання певних дій та власне атаки, що мають свій тип даних - ComboComponentData.

Ворог під час атаки може використати власну зброю, зробити ривок на гравця зі шкодою або різко змінити своє положення. Це відбувається за допомогою окремої машини станів всередині стану атаки, яка виконує відповідні до доданих у ворога комбінації атак стани. Вони мають свій клас EnemyComboState, що витягує дані з ComboComponentData, якщо такі є, та виконує по ним роботу.

2.2.2. Програмна реалізація зброї для сутностей

Зброя працює однаковою чином для персонажа та ворогів, однак її реалізація є доволі комплексною, тому її реалізація винесена в окремий пункт.

Зброя може бути різною: від холодної до вогнепальної. Кожна зброя має власні особливості, які потрібно реалізовувати:

- шкода:
 - значення шкоди;
 - площа нанесення шкоди;
 - відкидування;
- віддача від використання зброєю;
- відмальовування спрайтів:

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						38
Змн.	Арк.	№ докум.	Підпис	Дата		

- спрайти сутності;
- спрайти зброї на сутності;
- спрайти снарядів (якщо потрібні);
- створення снарядів зі своїми параметрами.

Для запобігання повторюваного коду в кожному відокремленому класі зброї було прийнято рішення створити систему, до якої можна буде додавати самотужки необхідні ефекти в меню рушія. Таким чином можна в будь-який момент часу створювати нові види зброї зі своїми унікальними ефектами.

Зброя реалізуватиметься окремим об'єктом, що матиме компонент класу Weapon. Він відповідає за керування їх діями, а також номеру атаки, який наразі виконується. З лістингом коду можна ознайомитись в додатку К.

Клас не має власного функціоналу, який реалізує дії конкретної зброї та не зберігає напряму поля необхідних комбінацій атак. Це вхідна точка, яка вже запускає всі інші механізми, передбачені системою.

Система зброї виконує свої дії відокремлено від станів об'єктів. Об'єкт, що має на меті використати зброю має стан атаки, але в ньому вимикаються основні анімації, що вказано на рисунку 2.11, та майже припиняється робота скрипту, окрім машини станів. Натомість зброя вмикає окремо підготовлені анімації персонажа під час атаки та скрипти, що забезпечують рух та реагування об'єкта на зараз.

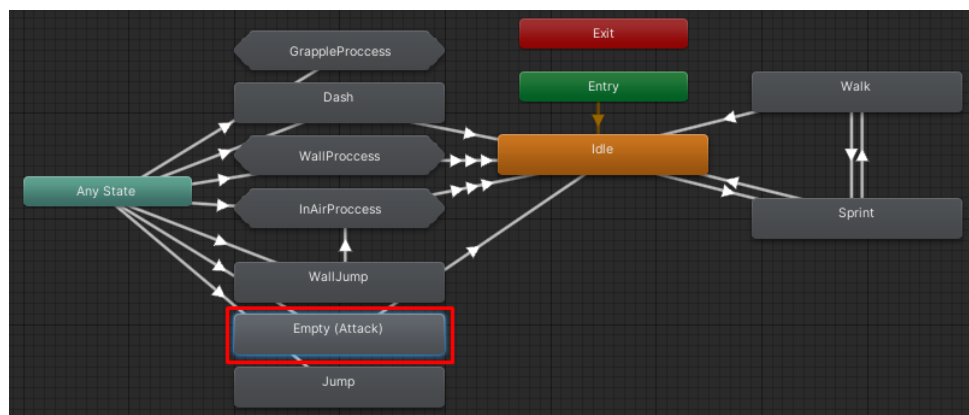


Рисунок 2.11 - Схема аніматора персонажа та виділена порожня анімація атаки

Спрацьовування компонентів системи відбувається під час анімацій, в яких за допомогою тригерів виставляються місця анімації, в яких повинні

виконуватись потрібні дії. Наприклад, гравець під час використання одного з виду холодної зброї рухається вперед, і це зумовлено двома тригерами початку та кінця руху. Реалізація представлена на рисунку 2.12, а всі можливі тригери для анімацій на рисунку 2.13.

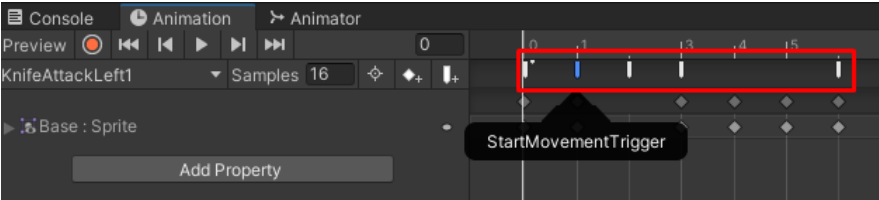


Рисунок 2.12 - Тригери в анімації атаки персонажа холодною зброєю

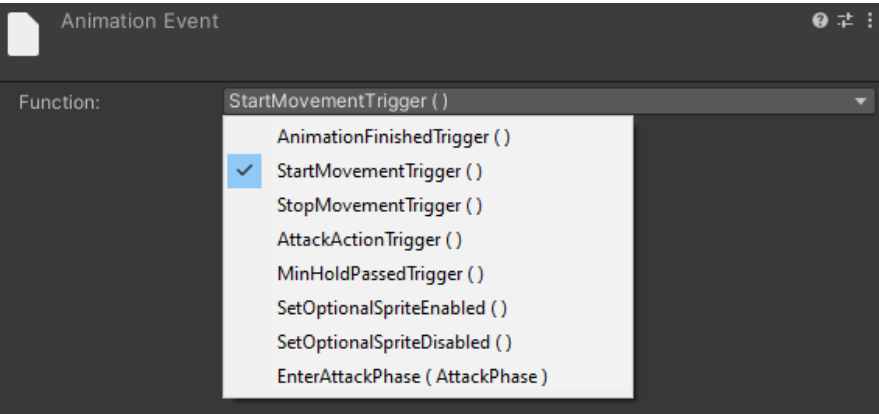


Рисунок 2.13 - Список існуючих стригерів для анімацій атаки зброєю

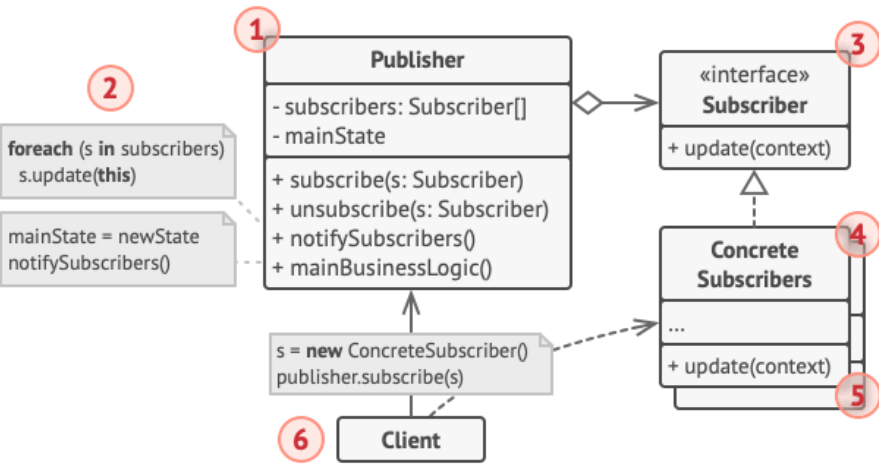


Рисунок 2.14 - Діаграма класів патерна Спостерігач

Обробляє тригери та виконує необхідні дії клас `WeaponAnimationEventHandler`, який реалізує поведінковий патерн проектування Спостерігач [19] – патерн, який створює механізм підписки, що дає змогу одним об'єктам стежити й реагувати на події, які виконуються в інших об'єктах. У випадку класу, який розглядається, компоненти зброї

підписують свої методи до його подій, які при спрацьовуванні тригера їх викликає.

Діаграма класів патерна представлена на рисунку 2.14, що розміщений на минулій сторінці.

Компонентів зброї реалізована велика кількість, в чому можна переконатись поглянувши на рисунок 2.15, і кожен з них відповідає за свою дію, яку може виконувати зброя, однак розробник сам визначає який компонент потрібно додати зброї для досягнення необхідного ефекту.

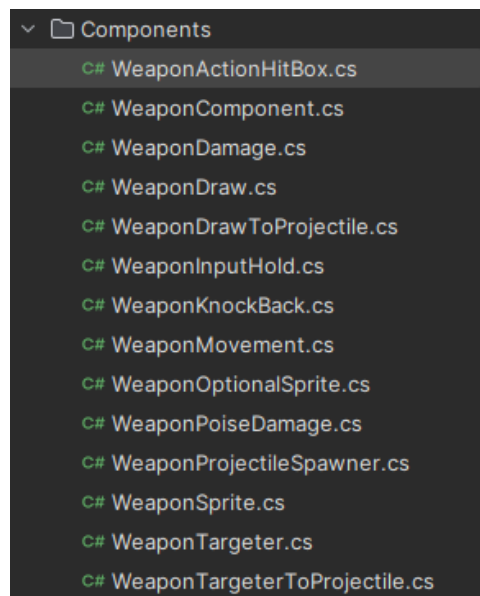


Рисунок 2.15 - Список всіх реалізованих компонентів зброї

Компоненти можна поділити на такі категорії:

- візуальні – працюють з відображенням анімацій або додатковими спрайтами;
- діючі – виконують дії, що пов'язані із заданням шкоди: власне шкода, накладання ефекту, відкидування тощо;
- рухові – рухають об'єкт, що керує зброєю;
- аналітичні – перевіряють дані для подальшої реалізації дій зброї. Наприклад, перевірка сигналу атаки, аналіз об'єктів, що попали у відповідну область тощо;
- снарядні – виконують запуск та обробку снарядів, якими володіє зброя.

Однак їх не потрібно додавати вручну, адже це робить клас WeaponGenerator (додаток Л), який на базі отриманих даних зброї додає необхідні компоненти та вписує в них дані, що необхідні для їх роботи.

Окремо можна розглянути снарядні компоненти, адже для них створюються окремі об'єкти снарядів, які виконують свою функціональність. Для того, щоб цих об'єктів не було занадто багато на мапі було використано твірний шаблон проектування Пул об'єктів [20].

Снаряди реалізовані за допомогою класу Projectile, однак, як і зі зброєю, він має компоненти, що унаслідують клас ProjectileComponent та які виконують свою функцію: частина з них перегукуються з компонентами зброї, як от шкода, відслідковування зіткнення, відкидування тощо, так і особливі, які відповідають за поворот снаряду, його гравітацію, застрягання в об'єкт, самонаведення тощо. Вони додаються до об'єкта, який зберігатиметься як префаб, та там і налаштовуються.

Однак один з компонентів зброї, який відповідає за створення снарядів, описує дані, які потрібно присвоїти деяким компонентам. Також він створює клас типу ObjectPools, який зберігає пули об'єктів всіх використовуваних типів снарядів та керує ними, передаючи снаряди для створення. Якщо снаряд вважається недійсним, він деактивується і якщо через час новий снаряд створюється, цей ж об'єкт повторно використовується вже з новими параметрами. Для гравців це буде виглядати як новий снаряд.

Дані зброї зберігаються в WeaponData, що реалізує батьківський клас ScriptableObject. Він має дані про кількість наявних атак та аніматор, який буде програвати анімації, а також, найголовніше, список всіх рухів, які може виконувати об'єкт під час атак. Список складається з елементів типу WeaponComponentData, який, в свою чергу, зберігає список даних з типом AttackData в кількості реалізованих атак.

Як можна було помітити, описані компоненти мають власний тип даних, який потрібно додавати до інспектора ігрового рушія, однак в його стандартному вигляді не передбачено додавання кнопкою різних типів даних,

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		42

тому для цього було написано клас `WeaponDataEditor`, що видозмінює інспектор для таких потреб. Він вбудовує різні кнопки, що додають всі можливі елементи до списків, та дає можливість модифікувати назви елементів списку для кращого розуміння параметрів. Вигляд інспектора з модифікатором є на рисунку 2.16, що розміщений нижче.

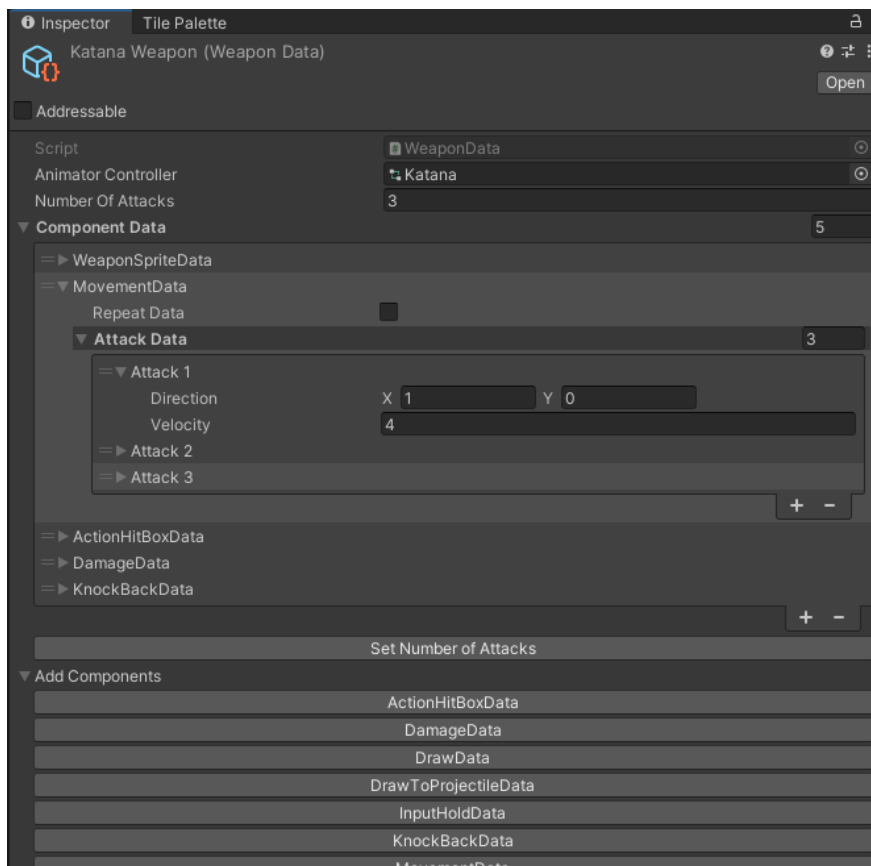


Рисунок 2.16 - Вигляд класу `WeaponData` в інспекторі рушія з модифікацією (неповний вигляд)

2.2.3. Програмна реалізація об'єктів на локації

Одними ворогами та персонажем ігровий цикл не завершується. Для різноманіття дій гравця на локації були створені фізичні та логічні об'єкти, які можуть взаємодіяти або працювати поодиночі.

Насамперед потрібно зазначити, що в ігровому рушії Unity реалізація поведінкового патерна Спостерігач зводиться до створення в класі поля типу `UnityEvent`, порожнього або з вказанням конкретних параметрів, та підписання до нього необхідного методу з об'єктів, використовуючи інспектор рушія. В

рамках розділу ця реалізація взаємодії є типовою для всіх об'єктів, що розглядатимуться.

Об'єкти, що є фізичними на локації, є наступними:

- рухомий блок;
- гострі шипи;
- стіна, що розбивається;
- манекен;
- кнопка;
- лампа;
- ворота;
- машина предметів.

З вказаних вище об'єктів рухомий блок та гострі шипи не мають програмних точок взаємодії з собою, однак вони впливають на інші об'єкти або сутності.

Рухомий об'єкт по своїй суті простий квадрат з графікою, котрий має фізичне тіло та колайдер. Він не має ніякого власноруч написаного програмного коду, який би описував його поведінку, однак його можуть штовхати вороги та персонаж, а також з його допомогою активуються наземні об'єкти, на які потрібно встати. Таким об'єктом є кнопка, але про це згодом.

Гострі шипи – карта тайлів, на яких розміщені спрайти шипів. До нього додано спеціальний колайдер, який приймає форму виставлених спрайтів. Їх поведінка полягає в тому, що при зіткненні із сутністю вони передають їх підсистемі життів нанести максимальної шкоди, тим самим вони призводять сутність до загибелі. Клас, що виконує цей функціонал, представлений нижче.

```
using UnityEngine;

public class Spike : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D i) =>
    i.gameObject.GetComponent<HealthController>()?.Damage(999);
}
```

Такою ж картою тайлів є стіна, що розбивається. Точніше вона має декілька карт, де однією закритий простір, прохід до якого закрито стіною, а всі

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						44
Змн.	Арк.	№ докум.	Підпис	Дата		

інші відповідають за стадії розбиття стіни. На них доданий відповідний колайдер як в попереднього об'єкта, який реагує на зброю та снаряди, від яких стадії починають змінюватись до тих пір, поки їх кількість не завершиться. В такому разі прохід відкриється.

Манекен – об'єкт, який потрібен для відслідковування ударів від зброї або снарядів. Відслідкувати удар можна візуально: залежно від сторони, з якої по ньому вдарили, він починає колихатись як м'ячик у відповідну сторону. Зміна спрайтів виконана за допомогою аніматора.

Кнопка є однією з ключових об'єктів для створення та вирішення механізмів. Його клас `ActivationButton` перевіряє чи не знаходиться на ньому об'єкт необхідного шару та при виконанні умови змінює свій спрайт на натиснутий та виконує подію з поля типу `UnityEvent`, на який можна підписатись. Якщо необхідних об'єктів на ньому не виявлено, а кнопка активована, вона виконує подію деактивації.

Також вона має можливість зберігати список об'єктів, які потрібно активувати та деактивувати автоматично. Якщо звичайно розробнику потрібно підписувати два методи в різні події, які від великої їх кількості стають незручними для додавання інших, то цей метод аналізує всі додані до списку елементи та підписує необхідні методи до потрібних подій. Такі об'єкти мають батьківський клас `LogicalActionableObject`, який визначає однакову назву для активації та деактивації. Такий вид підписування об'єктів до подій буде зустрічатись в подальшому.

Наприклад, так працює наступний об'єкт – лампа. Якщо її зламати за допомогою зброї або снаряду під час її світіння, виконується подія активації. Якщо лампа не світиться, зламати її не можна.

Ворота – один з цілей активації кнопки або розбиття лампи. В звичайному режимі ворота зачинені, однак якщо виконати активацію, вони відкриваються. Також можна виставити режиму зворотної активації – вони відчинені до того моменту, поки об'єкт, до події якого він підписаний, не виконає її.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		45

Таким ж чином працює машина предметів, однак вона лише активуються один раз та під час цього видає з себе об'єкт кинутий предмет, однак про нього згодом.

На рисунку 2.17 представлено вигляд фізичних об'єктів, які між собою пов'язані за вищеописаними алгоритмами. Спочатку гравець штовхає рухомий об'єкт, що позначений номером 1, на ворота під номером 3. Ці ворота активуються від розбиття лампи за номером 2. Далі рухомий об'єкт падає вниз, де персонаж переміщує його на кнопку, що позначена номером 4. В результаті ворота під номером 5 відчиняються та персонаж отримує відкритий шлях вперед.

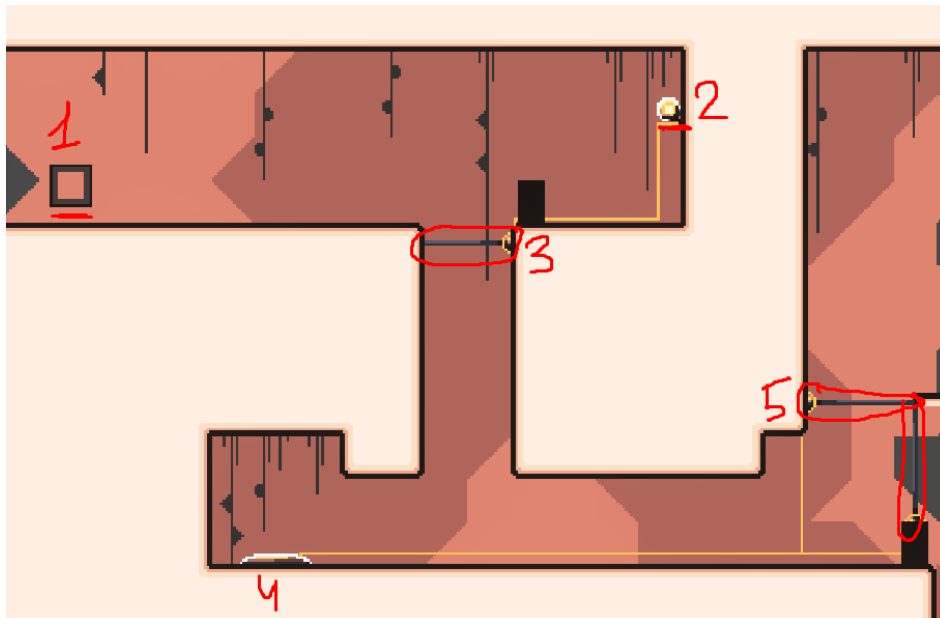


Рисунок 2.17 - Вигляд фізичних об'єктів на локації (з нумерацією)

Можна звернути увагу на яскраві оранжеві лінії. Вони зазвичай вказують шлях від об'єкта активації до об'єкта, що роблять активним. Таким чином гравець буде знати де знаходиться об'єкт-ціль механізму.

Також на локаціях є логічні об'єкти, які фізично не діють на сутність, однак відслідковують його або дії гравця мишкою для виконання потрібних задач. Їх перелік представлено нижче:

- тригер;
- подія на сцені;
- текст.

Тригер – універсальний невидимий об’єкт на локації. Його функція – відслідковувати, частіше всього, персонажа або інші потрібні об’єкти, та якщо він їх відслідкував в конкретних межах, виконує подію. У випадку виходу об’єкта з меж, виконується інша подія.

Тригер використовується у багатьох випадках: для переходу між сценами, вмиканні або вимиканні об’єктів на локації, додавання досягнення тощо. В собі він має лише клас DefaultTrigger та колайдер, який і відслідковує об’єкти. З лістингом класу можна ознайомитись в додатку М.

Подія на сцені працює подібним чином, тільки замість колайдера використовується радіус дії, та він діє лише на персонажа. Гравець заходить на відповідну область, після чого з’являється індикатор. Якщо на нього навести, висвітлиться інформація про подію, яка тут знаходиться, а якщо на нього натиснути, відбудеться програмна подія, на яку підписували різні об’єкти. Відслідковування миші також відбувається за допомогою виміру відстані між центром об’єкта події та курсора миші.

Зовнішній вигляд події на локації можна побачити на рисунку 2.18 сторінкою нижче. Великим кругом представлено область, на якій відслідковуватиметься персонаж, а кругом поменше – курсор мишки. Цятка та рамка з текстом – індикатор та інформація відповідно.

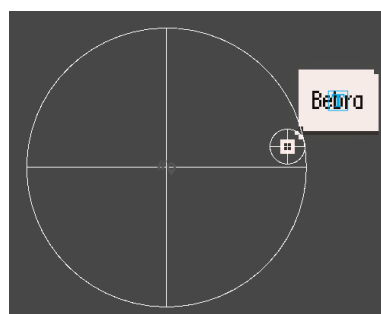


Рисунок 2.18 - Вигляд фізичних об’єктів на локації (з нумерацією)

Останнім за списком йде текст. Він з’являється на сцені за певних умов та має зміст. Він не є елементом GUI та знаходиться на одному місці на локації.

Насправді існує ще один об’єкт, який не згадувався – машина збереження, однак вона буде розглядатись разом з системою збереження в іншому пункті розділу.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		47

2.3. Розробка програмної частини логіки ігрового додатку

В минулому пункті розділу 2 було описано реалізацію елементів ігрового циклу ігрового додатку, однак для повноцінного вигляду цього недостатньо. В цьому пункті буде розглянуто логічну організацію.

Для початку потрібно розділити елементи, що формують логіку, на певні групи. Таких груп є дві та вони представлені в переліку нижче;

- Стани ігрового циклу;
- Система збереження.

Далі в підпунктах буде розписано їх призначення та програмна реалізація.

2.3.1. Програмна реалізація станів ігрового циклу

Стани ігрового циклу – група елементів ігрового додатка, які зберігають її поточний стан для контролювання процесів, що можуть виконуватись.

Для прикладу, гравець поставив гру на паузу. В такому разі він не буде мати можливості відкривати інвентар або керувати персонажем до тих пір, поки він вийде з цього стану. Або під час переходу до наступної локації відбуваються певні дії перед закриттям поточної сцени та на початку наступної.

Серед таких елементів є зберігачі та спостерігачі станів, а також ті, що відповідають за завантаження ігрової сцени. Перед їх розглядом потрібно зазначити, що вони реалізують поведінковий патерн Спостерігач подібним до пункту 2.2.3 чином.

Контролери завантаження ігрової сцени відповідають за перемикання ігрових сцен та виконання необхідних дій на їх початку та кінці. Представників таких контролерів є два.

Перший з них – клас GameListeners. Це класичний спостерігач, що викликає подію на початку сцени та в кінці її існування. До нього зазвичай підписуються об'єкти, яким необхідно виконати свої дії в момент, коли відбулось повне завантаження необхідних ресурсів гри.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		48

Другий представник – клас LoadSceneManager, що реалізує породжувальний патерн проектування Одинак [21] щоб інші об’єкти мали до нього вільний доступ, так як він є єдиним на сцені. Цього ціль – викликати подію перед початком завантаження наступної сцени та заміною поточної сцени наступною. Для подібних дій клас використовує асинхронний метод SceneManager.LoadSceneAsync(), котрий на фоні завантажує наступну сцену, однак не робить її поточною до тих пір, поки йому про це не повідомлять. Таким чином відбувається як виклик вищевказаних подій, так і програвання анімації завантаження наступної сцени.

Далі буде розглянуто зберігачі та спостерігачі станів. Насправді це один клас під назвою GameStates, котрий має визначення всіх необхідних станів гри, а саме:

- стан завантаженого ігрового циклу;
- стан катсцени;
- стан відкритого інвентаря;
- стан відкритої паузи;
- стан смерті;
- стан бою з вказанням кількості ворогів, що атакують.

Клас має подію на випадок зміни значення станів, на які підписуються всі охочі об’єкти. Однак є одна відмінність роботи класу від того, що було продемонстровано до цього моменту – елемент не має власного об’єкта на сцені.

Клас GameStates унаслідуює спеціальний клас SingletonScriptableObject. Він є модифікацією класу ScriptableObject, котрий реалізує в ньому патерн Одинак. Він зберігає стани гри протягом всієї гри, що спрощує їх перенесення на наступну сцену, та до нього є простий доступ зі всієї програмної частини додатку для підписання методів до його подій, а також змін станів, якщо об’єкт відповідає за нього.

Окремо слід винести стан бою, адже він може викликати як подію при зміні кількості ворогів, так і окрему подію в момент появи першого ворога або зникнення останнього.

2.3.2. Програмна реалізація системи збереження

Для кращого ігрового досвіду для гравців для ігрового додатку було реалізовано систему збереження. Він дозволяє гравцям через певні точки на локаціях зберігати прогрес гри, який дає можливість після перерви повернути гравця в той момент, на якому він зупинився. Також він дає можливість зберігати прогрес в декілька файлів, які гравець зможе використати в разі потреби.

В додатку це буде реалізовано ланцюгом взаємодії програмного коду, формуючи ієрархічну структуру. На вершині ієрархії – контролер, який отримує всі існуючі об'єкти, котрі потребують збереження, та отримує з них дані для запису у файл. Таким ж чином він з файлу може передати потрібні дані кожному об'єкту. Вони ж, в свою чергу, відокремлюють ці дані для кожного компонента, якщо це необхідно, та відправляють їм для відновлення збереженого стану.

Кожен компонент, що потребує збереження даних, реалізуватиме інтерфейс ISavable, лістинг якого наведено нижче.

```
using UnityEngine;

public interface ISavable
{
    object GetData();
    void RestoreData(object data);
}
```

Інколи об'єктам характерно мати декілька подібних компонентів. Також вони потребують певного маркування для того, щоб отримати точно потрібний набір даних, тому для цього буде реалізовано клас SavableEntity (лістинг наведено нижче), що генеруватиме кожному об'єкту ідентифікатор і, якщо гравцеві потрібно зберегтись, зчитуватиме із всіх компонентів об'єкта дані для

збереження, та відправить їх далі по ієрархії. Таким ж чином відбуватиметься запис даних в компоненти при завантаженні гри.

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SavableEntity : MonoBehaviour
{
    [SerializeField] private string id = string.Empty;

    public string Id => id;

    [ContextMenu("Generate Id")]
    private void GenerateId() => id =
        Guid.NewGuid().ToString();

    public object GetSavedData()
    {
        var data = new Dictionary<string, object>();

        foreach (var savable in GetComponents<ISavable>())
        {
            data[savable.GetType().ToString()] =
                savable.GetData();
        }

        return data;
    }

    public void RestoreSavedData(object data)
    {
        var dataDictionary = (Dictionary<string,
            object>)data;

        foreach (var savable in GetComponents<ISavable>())
        {
            string typeName = savable.GetType().ToString();

            if (dataDictionary.TryGetValue(typeName, out
                object value))
            {
                savable.RestoreData(value);
            }
        }
    }
}
```

Слід зазначити, що автоматично згенерований ідентифікатор для таких об'єктів є великим та неінтуїтивним, через це отримати дані, які для них

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						51
Змн.	Арк.	№ докум.	Підпис	Дата		

призначені, через сторонній код не вийде. Але якщо є така необхідність, можна вписати власний ідентифікатор і по ньому з файлу отримати необхідне поле, однак потрібно забезпечити неповторність ідентифікаторів, інакше будуть проблеми в додатку.

Таким об'єктом частіше всього є персонаж. Також далі по тексту буде згадано ще декілька подібних за ситуацією об'єкта.

Отримані дані відправляються до головного класу SessionSaveSystem, що відповідає за зчитування та запис даних в файли. Лістинг класу наведений в додатку М.

Він має на меті коригувати всі збереження окремо один від одного, щоб гравець мав можливість зберігати декілька прогресій гри та вибирати їх в подальшому через спеціальне меню. Для того щоб дізнатись скільки збережень існує та як їх файли називаються було створено структуру даних SavesStructure, в якому ці дані записані.

Однак вона, як і певні параметри, як наприклад налаштування якості графіки додатку або обраної мови, зберігається за допомогою вбудованої системи збереження PlayerPrefs [22]. Вона є зручною для запису невеликих даних, але є недоречним збереження всіх даних в одне місце, адже зчитування та пошук потрібних даних в громіздкому файлі є більш вимогливим процесом аніж відкриття окремого файлу, який повністю заповнений необхідною наразі даними, тому ця система не використовується для збереження прогресу гравця.

Для вибору збереження за допомогою GUI в меню гри або в паузи використовується клас SavesMenuSelector. Він зчитує дані зі структури SavesStructure, використовуючи вищеописану систему збереження, та генерує кнопки. В ці кнопки передаються дані для відображення необхідної гравцеві інформацією та сценою, де він зупинився та яку потрібно завантажити. Однак реалізація кнопок та самого меню буде описана в наступному пункті розділу про GUI.

Однак дані, які потрібні кнопкам, беруться з окремого об'єкта на локації, який під час збереження передає цю інформацію. Він має вручну описаний

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		52

ідентифікатор щоб просто отримати з файлу необхідні дані. Його реалізація виконана в класі SessionInfoManager (додаток П). Для збереження потрібних даних використовується структура даних SessionInfoData, лістинг якого представлено в наступній сторінці.

```
using System;
using System.Collections.Generic;

[Serializable]
public struct SessionInfoData
{
    public string sceneName;
    public string locationKey;
    public double time;
    public List<string> completedTasks;
}
```

В ньому зберігаються наступні речі:

- назва сцени;
- назва місця на локації;
- поточний час, що витрачений гравцем на цьому моменті гри;
- список виконаних завдань.

Назва місця на локації та нові виконані завдання додаються за допомогою тригерів, що описані в пункті 2.2.3, або іншими подібними діями. Назва сцени береться напряму з додатку, а витрачений час через об'єкт з класом-компонентом GameTimeCounter, що реалізує патерн Одинак.

Даний клас отримує необхідне поле з часом в структурі даних про інформацію збереження та відслідковує час, який гравець проводить в сеансі. Якщо гравець зберігає дані, з класу приймається поточний витрачений час та записується в структуру даних. У випадку переходу до іншої сцени клас зберігає значення як буфер до системи збереження в PlayerPrefs та отримує його у разі завантаження наступної сцени.

Насправді подібний алгоритм збереження буферу відбувається для всіх даних поточного збереження, адже при зміні сцен всі дані очищаються. В такому разі SessionSaveSystem зберігає всі дані зі сцени до PlayerPrefs та отримує з нього ці дані в наступній сцені. Алгоритм відновлення збережених даних для об'єктів працює так, що дані використовуються лише тоді, коли

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		53

потрібний об'єкт існує на сцені, тому помилок від лишніх даних, що призначені іншим сценам, не буде.

Також слід вказати дані, які використовуються для відображення інформації на кнопках, адже структура даних інформації про збереження містить лише дані типу string або float, хоча для кнопок потрібна картинка і відсоток проходження по кількості виконаних досягнень. Для цього використовуються класи, що мають батьківський клас SingletonScriptableObject, який описувався раніше в кінці пункту 2.3.1. В таких об'єктах можуть зберігатись наступні речі:

- список існуючих місць на локаціях (клас LocationsListScriptable);
- список досягнень в ігровому циклі (клас TasksListScriptable);
- список предметів (клас SaveScriptableItemList).

З перших двох списків кнопка отримує дані про те, який текст використовувати, яке зображення відмалювати та який відсоток від всіх досягнень гравець отримав.

Останній список створений через те, що зберегти предмети інвентаря у вигляді об'єктів типу SctiptableObject у файл неможливо, тому було вирішено зберігати лише унікальні назви предметів та по ним отримувати потрібні предмети з об'єкта.

2.4. Реалізація графічних елементів, анімацій та інтерфейсу користувача

Графічна складова ігрових додатків є ключовою у формуванні впізнаваного стилю, як і оформлення GUI та його функціоналу для гравців, тому цим темам буде виділено окремий пункт в розділі.

Для початку потрібно розібратись з роботою анімацій, адже це основна річ, до якої зазвичай буде писатись програмний код. Він згаданий не просто так, адже за допомогою нього будуть реалізовуватись більшість анімацій інтерфейсу за допомогою певної надбудови ігрового рушія.

Перший спосіб – використання вбудованої в рушій систему анімацій [23], котра працює на основі патерна Стан. Розробник створює файли анімації для

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		54

кожного об’єкта та за допомогою контролера Animator встановлює між ними переходи. В ньому можна виставляти певні параметри різних типів: int, float, bool або просто сигнал – за допомогою яких можна керувати поточні стани анімації через код.

Таким прикладом є система анімацій для сутностей, що представлена в класі AnimatorController. За допомогою нумератора ActionStateValues класи сутностей через власні стани посилають сигнали того, яку анімацію та з яким параметром програвати. Також цей клас перетворює відіслані дані в потрібний для контролера Animator вигляд та має окремі методи для встановлення руху і стрибка, так як від числових значень, що передаються, анімація для програвання змінюється.

Приклад вигляду схеми анімацій та параметрів можна побачити на рисунку 2.19.

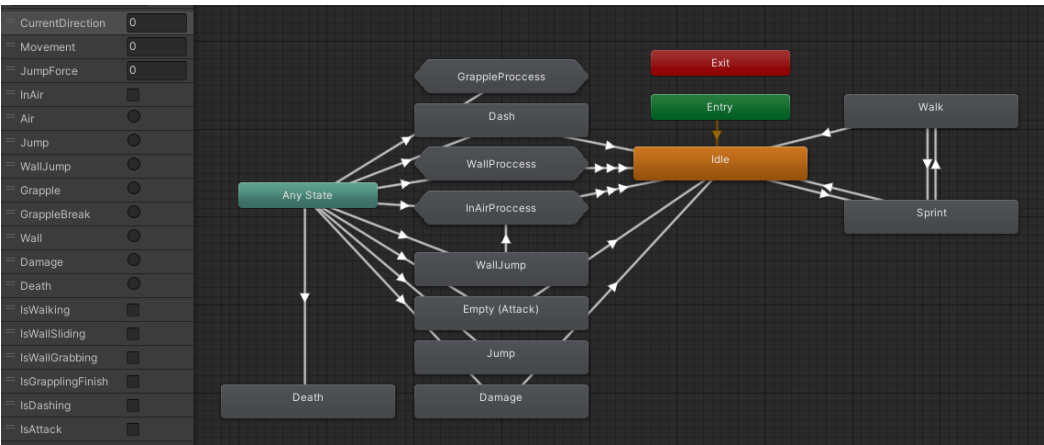


Рисунок 2.19 - Схема анімацій для гравця

Також для анімацій, котрі залежать від числових значень, мають власні схеми зі станами анімації. Це створено для того, щоб не засмічувати основну схему анімацій великою кількістю елементів. Приклад таких схем представлено на рисунках 2.20 та 2.21.

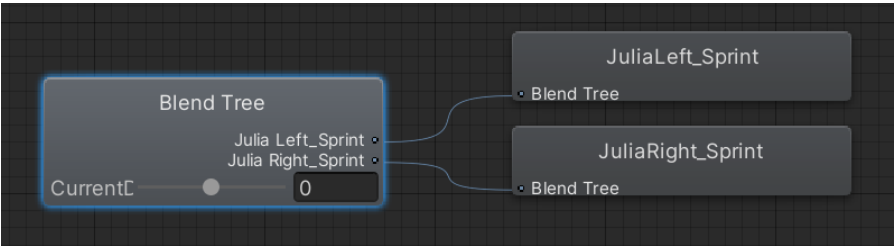


Рисунок 2.20 - Схема анімації бігу персонажа

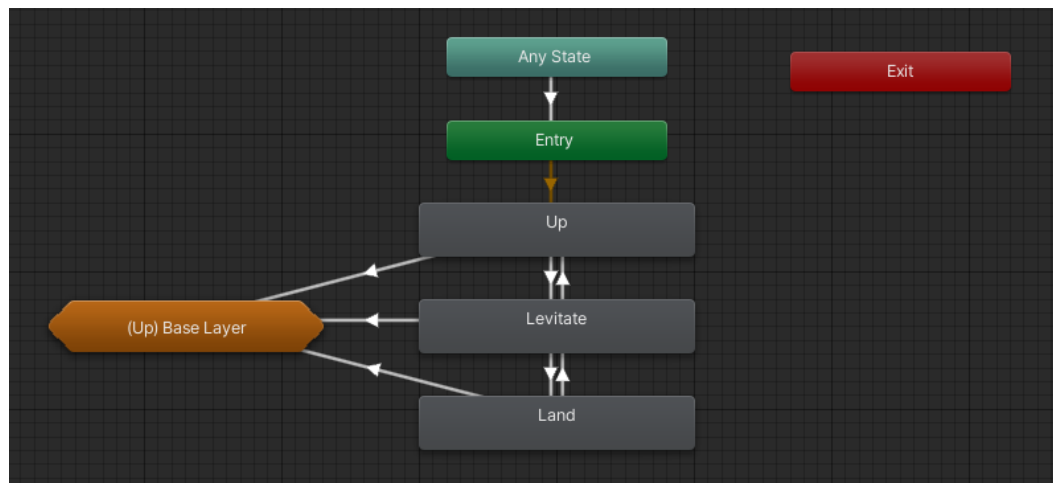


Рисунок 2.21 - Схема анімацій персонажа в повітрі

Другий спосіб реалізувати анімації – використання надбудови DOTween [24], котра через код самотужки створює анімації для об’єктів та дії над ними. В ній можна використовувати як вбудовані, так і власноруч створені методи для різних дій. Але її потрібно використовувати лише в тих моментах, коли анімація не прив’язана до постійних змін, як от наприклад рух сутностей або зміна їх спрайтів, так як сам процес програвання анімацій неконтрольований зсередини і може бути лише зупиненим без повернення на початкове положення.

Однак для незначних елементів, як от GUI, це найзручніший та найшвидший спосіб анімації з великою кількістю різноманіття. Найчастіше використовувались вбудовані методи затухання спрайтів, їх переміщення від початкової точки та тряска у випадку взаємодії.

Для інтерфейсу надбудова стала у нагоді в таких ситуаціях:

- ефекти появи та зникнення елементів інтерфейсу;
- зміна положення та розміру елементів;
- реалізація та зміна сторінок інтерфейсу;
- ефекти взаємодії з елементами.

Для появи та зникнення елементів використовувалось затухання за допомогою методу DOFade(), в якому вказується значення прозорості та кількість секунд на виконання дії. Таким чином виконуються більшість вбудованих методів надбудови DOTween.

Для зміни положення елементів використовувались методи DOAnchorPosX() або DOAnchorPosY() в залежності від напрямлення руху елемента по координаті X або Y відповідно. Вони є аналогом методу DOMove() для інтерфейсу, що використовується для руху звичайних об'єктів. У випадку зміни розмірів елементів існує метод DOScale().

Реалізація сторінок інтерфейсу в додатку виконана за допомогою списку елементів, які потрібно перемикає, та способу їх розміщення: заміщення на місце попереднього елемента з ефектом затухання або рух одного об'єкта, в якому по черзі розміщені елементи-сторінки, який рухають на певне значення, що відповідає потрібній сторінці.

Ефекти для взаємодії з елементами виконані названими раніше методами в залежності від цілі ефекту, а також не згаданим методом DoShakeAnchorPos(), який «трясе» об'єкт певний проміжок часу.

Елементи інтерфейсу поділені на ті, що застосовуються для головного меню, та елементи ігрового циклу. Перелік всіх елементів наступний:

- текст;
- кнопка:
 - звичайна;
 - кнопка розділу;
 - предметна (для взаємодії з предметами інвентаря);
 - збереження (для вибору збереження зі списку);
- чек-бокс;
- слайдер;
- список варіантів;
- сторінки;
- виноска (для деталей предмета).

Всім елементам можна призначити свої функції та скласти з них GUI, однак потрібно зупинитись на текстах, адже вони реалізовані спеціальним чином.

Перш за все вони відображають свій текст за допомогою надбудови TextMeshPro [25]. Він відповідає за створення мапи символів, яка буде виконувати їх коректне відмальовування на екрані. Це стане у нагоді у випадку інтеграції в додаток графічного стилю «піксель-арт», але про це згодом.

В грі додано дві мови, між якими можна перемикатись, однак це потрібно певним чином керувати. Для цього було використано ще одну надбудову під назвою Localization [26], котра дозволяє змінювати будь-які параметри типу string на ті, що визначені в таблицях – спеціальних структурах даних, що зберігають ключ та їх значення для кожної мови. Таблиць можна створювати безліч та ними можна керувати з окремого вікна рушія.

В ігровому додатку реалізовано українську та англійську мови. Приклад збереження даних про значення для кожної з мов зображено на рисунку 2.22.

New Table Collection Edit Table Collection		
Selected Table Collection ItemsDataTable(StringTable)		
Table Collection Name ItemsDataTable		
Key	English	Українська
bigBalloonArtifact	Big balloon	Велика кулька
bigBalloonArtifactDescription	It has great upward thrust. Probably, a large number of them can lift the whole house up	Має велику тягу наверх. Напевно велика їх кількість підніме в повітря цілий будинок

Рисунок 2.22 - Збереження тексту для кожної мови в списках для предметів

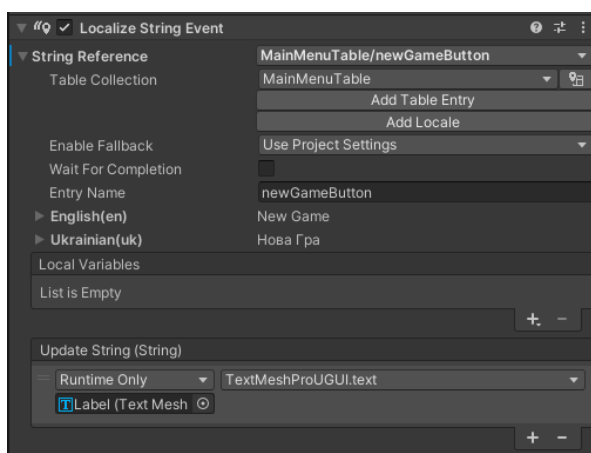


Рисунок 2.23 – Компонент LocalizeStringEvent для кнопки «Продовжити»

З визначеного ключа отримує та назначає потрібне значення компонент LocalizeStringEvent. Якщо мову в додатку було змінено, він отримує нове

значення на записує його в текст інтерфейсу. На рисунку 2.23 вказано приклад налаштування компонента для кнопки «Продовжити». Для спрощеної роботи з таблицями в ньому є можливість створити новий ключ та значення до нього.

Всі описані вище елементи утворюють різні меню, котрі виконують свою функцію. Їх список містить такі пункти:

- головне меню, яке зустрічає гравець під час запуску гри;
- меню налаштування, в якому відбуваються налаштування;
- список збережень, котре відображає всі збереження, які в ньому можна вибрати для завантаження;
- пауза, яка дозволяє гравцеві перейти до головного меню, відкрити меню налаштувань та список завантаження;
- інвентар, через який гравець переглядає предмети та взаємодіє з ними, а також меню дає відомості про параметри персонажа.

Вигляд цих меню та порядок їх використання будуть описані в пунктах наступного розділу.

Як вже зазначалось в пункті 1.1, в ігровому додатку буде використано графіку в стилі «піксель-арт». В ньому будуть виконані всі елементи, які є в додатку. До цього списку включаються: карти спрайтів анімацій персонажа та ворогів, спрайти елементів ігрового циклу, індикація, елементи GUI, а також шрифт, який був підібраний спеціально для підтримання коректної роботи компонента камери PixelPerfectCamera [27].

Цей шрифт має назву Bulgaria Glorious Cyr та був взятий з безкоштовної збірки шрифтів з додатковою підтримкою українських символів UKRFonts [28]. його вигляд в ігровому додатку представлено на рисунку 2.24.



Рисунок 2.24 - Вигляд «піксельного» шрифту в грі

Спрайти анімацій для всіх об'єктів об'єднані в один файл, який розділяється інструментами ігрового рушія. Рисунки з прикладами анімаційних спрайтів для гравця, одного ворога та елемента декору можна розміщені нижче. За допомогою них і формуються анімації, про які був наданий опис на початку поточного пункту.



Рисунок 2.25 - Одне із зображень зі спрайтами анімацій персонажа вліво



Рисунок 2.26 - Зображення зі спрайтами анімацій наземного ворога



Рисунок 2.27 - Зображення зі спрайтами анімацій манекена

Звичайні спрайти зберігаються також об'єднаними у зображення по групам. Приклад одного з таких зображення представлено на рисунку 2.28.



Рисунок 2.28 - Зображення зі спрайтами елементів локації Білий Простір

Мапи тайлів мають окреме зображення, з якого в ігровому рушії виділяються всі тайли для малювання локації. З прикладом такого зображення можна ознайомитись на рисунку 2.29.



Рисунок 2.29 - Зображення мапи тайлів локації Білий Простір

Для локацій також існують спрайти, котрі виконують декоративну ціль щоб заповнити порожнечу задньому плані. Для них описаний спеціальний компонент BackgroundParallax, який реалізовує ефект паралаксу в залежності від положення персонажа на локації. Один з елементів заднього плану можна побачити на рисунку 2.30. Пікселі на ньому помінити складно, адже він має великі як для «піксельного» зображення розміри.



Рисунок 2.30 - Спрайт одного з елементів заднього плану локації

Всі елементи GUI також знаходяться на одному зображенні. Може здатись, що вони малі, однак в ігровому рушії є можливість розтягувати спрайт вказуючи кутові області, котрі потрібно зберегти в початковому вигляді. На рисунку 2.31 можна побачити це зображення, однак оранжевого фону на зображенні немає.



Рисунок 2.31 - Зображення зі спрайтами елементів інтерфейсу

Також є два зображення зі всіма іконками предметів та параметрів або кнопок. Вони зображені на рисунках 2.32 та 2.33 відповідно.



Рисунок 2.32 - Зображення з іконками предметів



Рисунок 2.33 - Зображення з іконками для параметрів та кнопок

На останок потрібно продемонструвати який вид формують описані вище спрайти для локацій (рисунок 2.34) та інтерфейсу (рисунок 2.35).

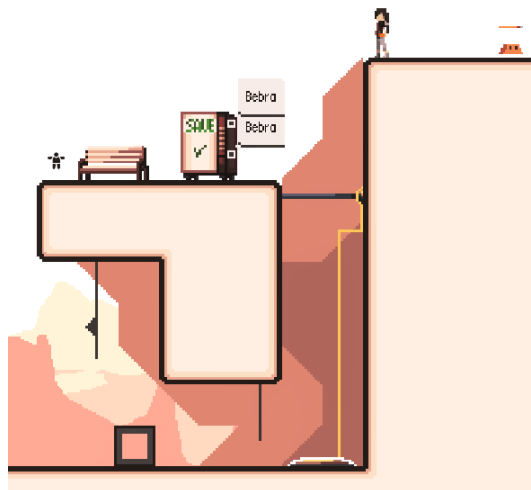


Рисунок 2.34 - Фрагмент локації Білий Простір



Рисунок 2.35 - Вигляд інвентаря під час ігрового процесу

Висновки до другого розділу

В розділі було спроектовано алгоритми роботи ігрового додатку і реалізовано програмну частину елементів GUI та ігрового циклу на основі об'єктно-орієнтованого підходу та патернів проектування.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		62

Для кращого представлення подальшої роботи було складено діаграми використання, діяльності та розгортання, на яких вказані доступні гравцю дії в грі та те, що виконуватиме додаток під час їх виконання. На їх основі було складено список елементів, котрі потрібно реалізувати для персонажа, ворогів, ігрового світу, GUI та внутрішньої логіки додатку.

Персонажу та ворогам було реалізовано однакову систему поведінки, яка базується на патерні Стан. Для керування компонентами, таких як система здоров'я, рухів, зіткнень тощо, існує єдина точка взаємодії на базі патерн Фасад. Система параметрів спроектована за допомогою патерна Декоратор. Система зброї для сутностей є однаковою та може використовуватись для створення нових предметів того ж типу з унікальними можливостями.

Елементи ігрового циклу, такі як наземна кнопка, ворота тощо, реалізовувались із застосуванням патерну Спостерігач. Вони мають можливість формувати окремі системи на локаціях для взаємодії з персонажем.

Стани ігрового циклу реалізовані одним класом із використанням патерна Одинак та використовуються для коректної роботи додатка під час ігрового циклу. Система збереження забезпечує збереження даних потрібних об'єктів на мапі для їх подальшого відновлення за потреби гравця.

Спрайти для персонажа, ворога, локацій та інтерфейсу були додані в карти спрайтів та додані в додаток для подальшого використання. Карти спрайтів реалізовувались для групування різних предметів з спільною рисою в одне зображення та кадрів анімацій одного предмета.

Були спроектовані елементи інтерфейсу, такі як кнопка, чек-бокс, список тощо, з яких вже формуватимуться всі меню GUI. Для їх відображення використовуються спрайти.

Анімації керуються програмним кодом. У випадку створення анімацій елементів GUI використовувалась окрема надбудова рушія, котра легко створює для них поведінку під час клацання, наведення тощо.

В результаті було отримано готовий ігровий продукт для кваліфікаційної роботи, який виконує необхідні умови.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		63

РОЗДІЛ 3. ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО ТЕСТУВАННЯ

3.1. Порядок встановлення та налаштування системи

Ігровий додаток в жанрі "Метроїдванія", який створено в результаті виконання кваліфікаційної роботи має назву «Interim». Він розроблений для користувачів ПК на базі операційних систем Windows 10, Windows 11 або іншої версії лінійки Windows для ПК, які наразі є підтримуваними корпорацією Microsoft.

Мінімальні системні вимоги для запуску додатку:

- процесор: Intel Core 2 Duo E5200 чи еквівалентний;
- оперативна пам'ять: 4GB;
- відеокарта: GeForce 9800GTX+ (1GB) чи еквівалентна;
- місце на диску: 200MB;
- DirectX: версія 10.

Отримати додаток можна через сервіс цифрової дистрибуції ігрових продуктів Steam через офіційний додаток для ПК. Для нього буде існувати як посилання, так і пошук через сервіс за допомогою назви або жанру. На сторінці гри потрібно натиснути кнопку, що відповідає за встановлення, після чого вищезгаданий додаток почне завантаження гри на ПК. Після цього з'явиться ярлик на робочому столі. Для запуску необхідно натиснути ЛКМ по ньому два рази.

3.2. Інтерфейс та порядок роботи

При запуску ігрового додатка відкривається головне меню, яке зображене на рисунку 3.1. На ньому знаходяться кнопки, які відповідають за запуск гри, перехід на різні сторінки меню: налаштування та список збережень – а також вихід з додатка до робочого столу. Слід зазначити, що гравець, який вперше запустив гру та не зберігав свій прогрес в грі, не має доступу до меню збережень, тому відповідна кнопка темніша за інші.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ док.ум.	Підпис	Дата		64

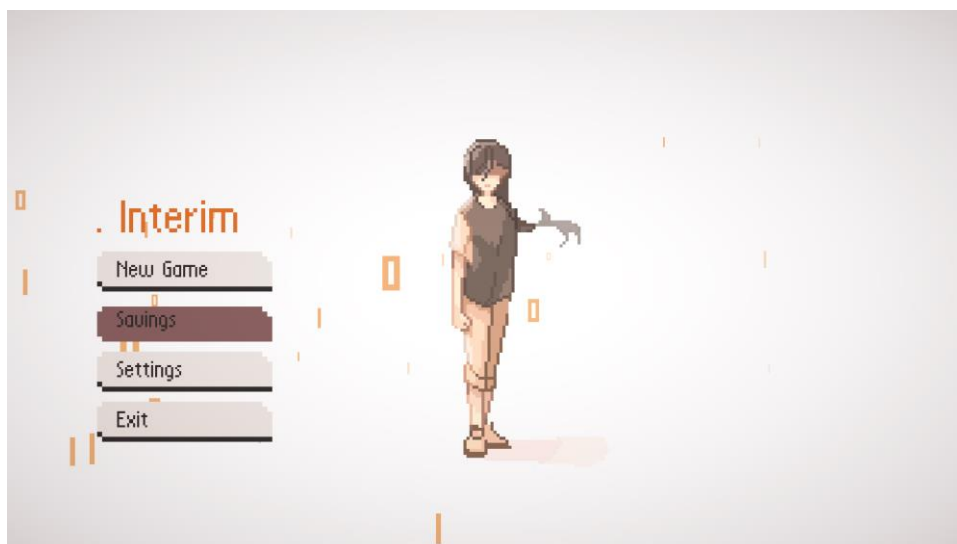


Рисунок 3.1 - Головне меню ігрового додатка

Для нових гравців всі налаштування є доступними, тому вони можуть відкрити меню налаштувань, де є параметри якості та мови, щоб підлаштувати додаток під свій лад. Саме меню представлено на рисунку 3.2. На ньому також є розділ «Інше», де можна в майбутньому скинути досягнення та подивитись ігрові титри.

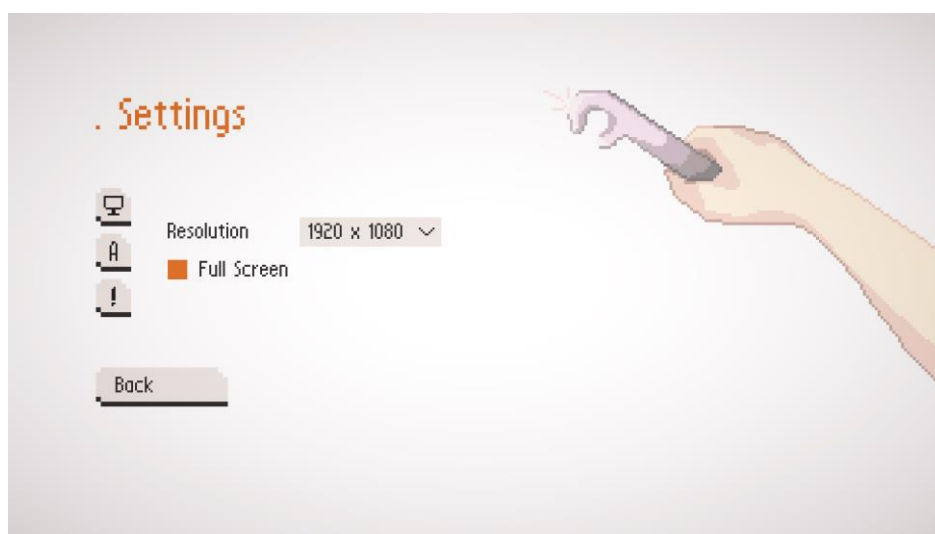


Рисунок 3.2 - Меню налаштувань

Якщо гравець має намір закрити додаток, з'явиться вікно сповіщення, яке запросить підтвердити свою дію. Подібне вікно може з'явитись якщо гравець під час ігрового циклу хоче вийти до головного меню або запустити збереження, а також в момент скидання досягнень. Приклад такого вікна представлено нижче.

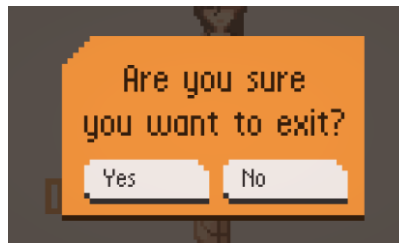


Рисунок 3.3 - Вікно сповіщення

В момент зміни сцен в додатку працює система завчасного завантаження сцен, під час якої з'являється відповідний екран. Приклад відображено на рисунку нижче.

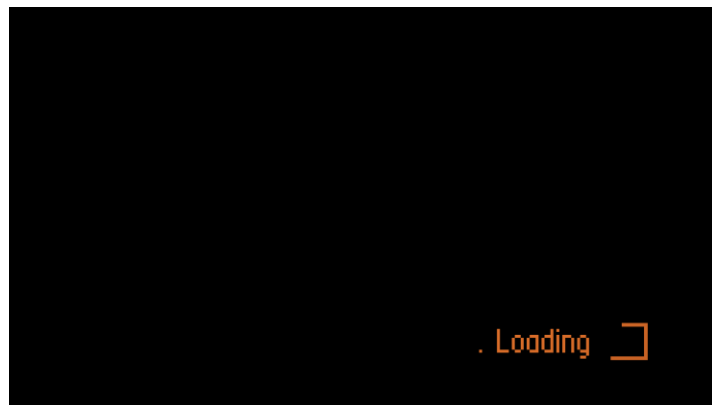


Рисунок 3.4 - Вікно завантаження

Перша локація, на яку потрапляє гравець вперше в додатку, є навчальною. На ньому гравця вчать основним механікам керування персонажем та взаємодії з елементами локацій за допомогою тексту на задньому плані та практичних прикладів. Фрагмент локації представлений на рисунку 3.5.

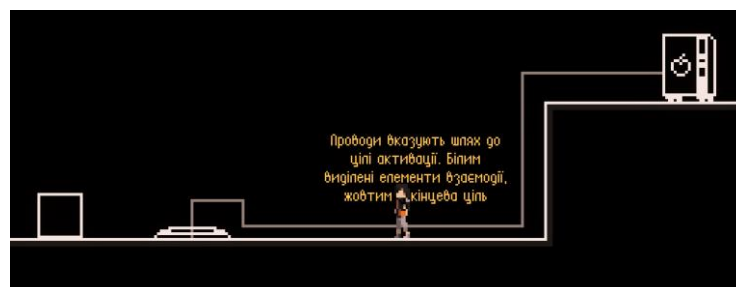


Рисунок 3.5 - Фрагмент навчальної локації

Список можливостей гравця наступний:

- бігати (кнопки WASD або стрілки);
- стрибати (пробіл);
- робити ривок (клавіша Shift);
- притягуватись до об'єктів (кнопка колеса миші);

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		66

- активувати кнопки та події на локації (ЛКМ або клавіша E);
- відкривати паузу (клавіша Escape);
- відкривати інвентар (клавіша I або Tab);
- використовувати зброю (ЛКМ та ПКМ).

З вказаним вище набором можливостей гравець може проходити перешкоди, які з'являтимуться впродовж гри.

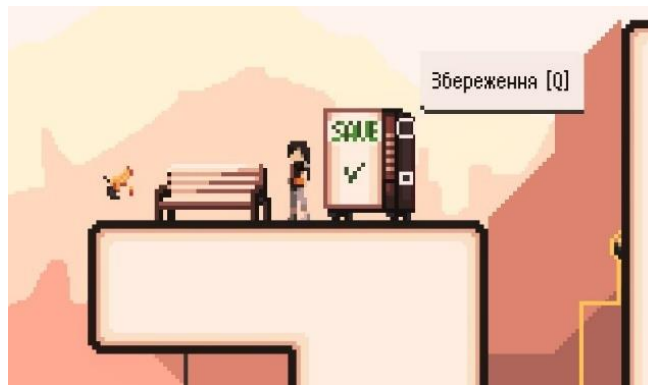


Рисунок 3.6 - Фрагмент локації Білий Простір

Після проходження навчального рівня гравець переходить до локації Білий Простір, з якої і починається основний ігровий цикл. В грі реалізовано дві локації: Білий простір та Бінарний міст

Фрагмент локації можна побачити на рисунку 3.6. Також на ньому зображено об'єкт збереження та кнопки, за допомогою яких гравець вибирає як саме зберегтись на цьому моменті. Також з лівої сторони можна помітити предмет, який персонаж може підібрати.

Для перегляду та застосування предметів гравець може відкрити інвентар. За допомогою нього можна переглянути параметри персонажа (рис. 3.7), всі наявні та одягнені предмети за категоріями (рис. 3.8), а також екіпірувати або зняти один з них.



Рисунок 3.7 - Основна сторінка меню інвентаря

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		67

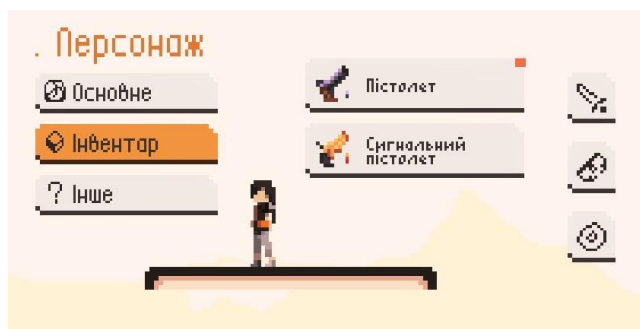


Рисунок 3.8 - Сторінка списку зброї інвентаря

Якщо навести курсор миші на предмет, можна отримати його опис, параметри та особливості. Приклад відображено на рисунку 3.9.



Рисунок 3.9 - Опис предмета «Шматочок піци»

Якщо гравець зберіг свій прогрес, в головному меню відкривається можливість продовжити гру з останнього збереження та переглянути меню збережень, в якому гравець може запустити те, яке йому наразі необхідно. Вигляд меню представлено нижче.

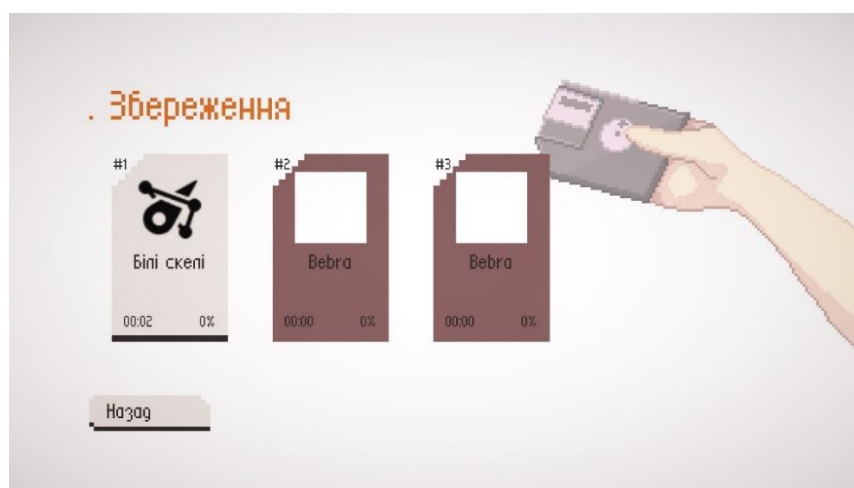


Рисунок 3.10 - Меню збережень

Таке ж меню можна побачити під час ігрового циклу, відкривши паузу. Це зроблено для зручності користування системою збережень. Побачити як виглядає ця версія збережень можна на рисунку 3.11.



Рисунок 3.11 - Список збережень в меню паузи

Через паузу можна також налаштувати додаток (рис. 3.12) таким же чином, як і в головному меню.

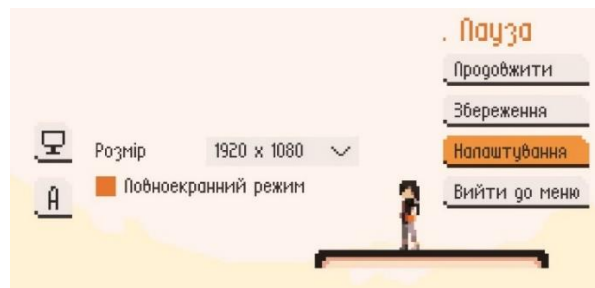


Рисунок 3.12 - Налаштування в меню паузи

Гравець не тільки рухається по локації, проходячи механізми та перешкоди, а й б'ється з ворогами (рис. 3.13). Персонаж, як і вороги, мають шкалу здоров'я, яка може змінюватись в залежності від того, отримує сутність шкоду чи зцілення. В цьому можна переконатись за допомогою рисунка 3.14.

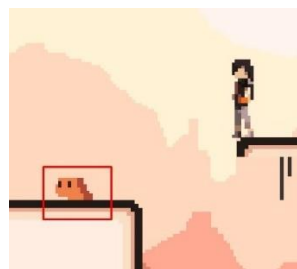


Рисунок 3.13 - Приклад ворога в грі



Рисунок 3.14 - Приклад неповної шкали здоров'я у персонажа

3.3. Тестування роботи додатку

Тестування ігрового додатку включає в себе низку областей, які відповідають за власний функціонал в ньому, однак не руйнує схему роботи іншої функціональності. Їх список складається з:

- інтерфейс;
- збереження;
- ігровий цикл;
- текстова складова.

Остання область перевіряє роботу тексту в грі та правильність викладення інформації, включаючи помилки в словах та граматику.

Для тестування було задіяно декілька осіб-гравців, які грали в додаток з точки зору власного досвіду та описували помилки або незручності які в них виникали. У більшості був досвід в іграх жанру «Метроїдванія», тому це також дало змогу виявити певні проблеми, які стосуються саме цього жанру.

Потрібно зазначити, що в кваліфікаційній роботі представлений додаток без описаних нижче помилок.

Інтерфейс тестується в першу чергу, адже під час першого запуску гравець знайомиться з грою через головне меню. До роботи інтерфейсу нарікань у тестувальників не було. Однак через звичку відкривати паузу для перегляду інвентарю було вирішено додати до паузи кнопку Персонаж. Це спростить керування додатком.

Далі протестуємо тестування збереження. Ця область додатку не мала нарікань зі сторони тестувальників.

Ігровий цикл є основною складовою гри, тому до нього було привернуто більше всього уваги під час тестування. В цій складовій було виявлено найбільше помилок, однак вони не були критичними для гри. Також були дані поради щодо певних елементів ігрового процесу, які заважали або уповільнювали його.

Список виявлених моментів складається з наступних пунктів:

- незаплановане ковзання персонажа по землі;

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						70
Змн.	Арк.	№ докум.	Підпис	Дата		

- неконтрольований рух персонажа:
 - під час прицілювання та стрільби;
 - при відкритому меню в ігровому циклі;
- довге скидання комбо ударів зброєю;
- незручне керування.

Незаплановане ковзання виникало під час зупинки на динамічних об'єктах, як от рухомі блоки, або руху персонажа на похиліні поверхні. Хоча останні для додатку окремо не створювались, однак для кнопки, що є елементом ігрового циклу, був доданий спеціальний колайдер (рис. 3.15), що дозволяє легко заштовхувати на нього рухомі предмети.

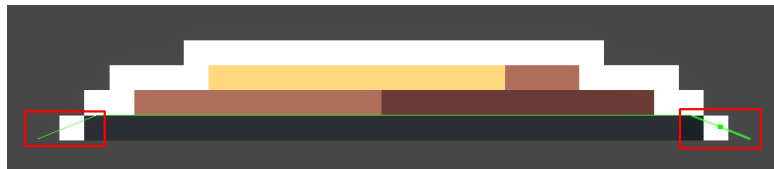


Рисунок 3.15 - Нахилені грані колайдера кнопки

Така поведінка ставалась в результаті відсутності тертя персонажа. Ця проблема була вирішена фізичним матеріалом, котрий мав певне значення тертя (рис. 3.16). Після його налаштування подібної помилки не відбувалось.

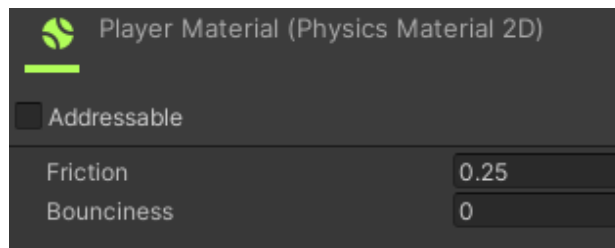


Рисунок 3.16 - Фізичний матеріал з вказаною силою тертя

Таке рішення виправило рух персонажа під час прицілювання та стрільби, однак для цього також було додано скидання прикладених сил на персонажа. Це зумовлено тим, що під час пострілу зброєю в повітрі персонаж летів з прикладеною раніше силою, яку не можна було контролювати. Тепер ж гравець буде знати, що при пострілі персонаж затримується в повітрі без руху, і це є більш справедливим по відношенню до нього. Побачити зміну анімації зброї дальнього ураження можна на рисунку нижче.

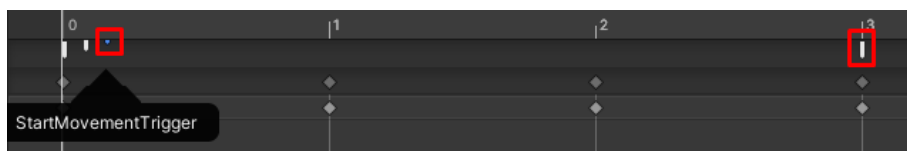


Рисунок 3.17 - Тригери початку та кінця руху гравця для дальньої зброї

Неконтрольований рух при відкритому меню в ігровому циклі є доволі значущим, адже гравець може при русі по землі або повітрі відкрити паузу або інвентар, під час чого персонаж може йти далі. Це призведе до незапланованих для гравця наслідків.

При дослідженні програмного коду виявилось, що клас `PlayerInputHandler`, що отримує та обробляє сигнали від гравця для керування персонажем, під час зміни станів гри в `GameStates` не очищає значення сигналів руху, через що стани персонажа гадають, що гравець все ще рухається. Це було виправлено присвоєнням нульового значення для відповідних полів.

Довге скидання номера поточного удару зброї є не помилкою, а скоріше неврахуванням доречності такого часу на відновлення першого удару. Тестувальники помітили, що удари можна зберігати на декілька секунд та використовувати для свого блага, що не є справедливим вже для гри. Однак значення затримки було наперед винесено в інспектор рушія, тому замість трьох секунд було виставлено значення в 0.2 секунди (рис. 3.18).

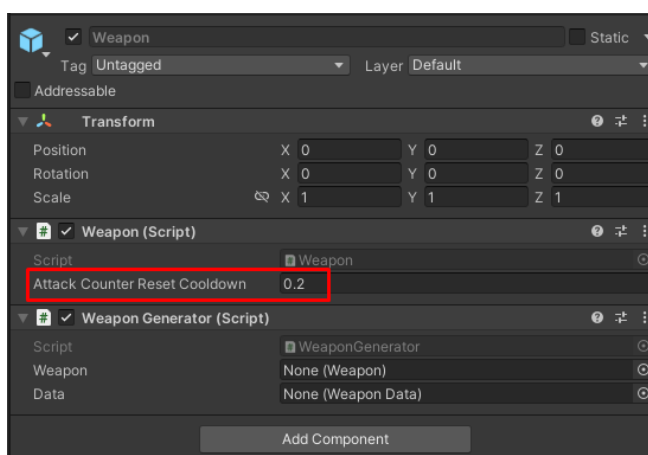


Рисунок 3.18 - Префаб зброї зі зміненим значенням

Керування – це аспект, котрий є суб'єктивним, адже кожен гравець має власні звички в ігрових додатках, однак керування гри було незручним для кожного з певних причин: розміщення декількох важливих кнопок для лише

одного пальця, далеко розміщені кнопки, які змушують використовувати руку з комп'ютерної миші тощо. Представлені в кваліфікаційній роботі кнопки взаємодії гравця з додатком є зміненими завдяки проведеному тестуванню.

Остання область – текстова складова. В ній тестувальники виявили частіше всього помилки у написанні слів, що було виправлено в подальшому. Однак вони також виявили помилку у відображенні опису предмета Скакалка, що відображена на рисунку 3.18. Це було зумовлено тим, що в таблицях локалізацій були пропущені ключі та текст для цього предмета. Наразі цієї помилки в додатку немає.



Рисунок 3.19 – Помилка відображення відомостей про предмет Скакалка

Висновки до третього розділу

У третьому розділі описано готовий ігровий додаток для кваліфікаційної роботи.

На початку було сформовано інструкцію як отримати через сервіс Steam, встановити та запустити ігровий додаток, а також представлено мінімальні системні вимоги, які потрібні для його запуску на ПК на основі операційної системи Windows. Далі було складено алгоритм можливих дій гравця в додатку.

В грі було реалізовано три ігрові локації: навчальна, Білий простір та Бінарний міст. Перша в переліку відповідає за демонстрацію гравцю всіх можливих умінь персонажа та механік гри. Всі інші є основними та мають цікаве наповнення, що залучає гравця досліджувати локації.

Для GUI були створені такі меню: головне, яке дає можливість відкривати інші сторінки інтерфейсу та запускати гру, меню налаштувань та його

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		73

підпункти для коригування гри під свій лад, меню збережень для вибору необхідної збереженої сесії в разі потреби, пауза та спрощені меню налаштувань та збережень в ньому, що відкривається під час ігрового циклу, а також інвентар з підпунктами. Також в ньому реалізовані вікна завантаження, смерті та попередження перед виконанням певної дії.

В кінці розділу проведено тестування додатку із залученням кількох людей, котрі оцінили роботу додатка та вказали на проблеми, які були виявлені та в подальшому виправлені. Серед них були: проблеми зі станами ігрового циклу під час паузи, непередбачуване ковзання персонажа під час взаємодії з кнопкою на локації та використанні вогнепальної зброї, довге збереження комбо зброї, а також певні помилки в тексті та відображенні локалізації.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						74
Змн.	Арк.	№ докум.	Підпис	Дата		

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було отримано робочий ігровий додаток в жанрі «Метроїдванія».

На початку було визначено ряд вимог, що відповідають жанру, а саме особливості персонажа, локацій, предметів та ворогів. Також було вказано збереження, інтерфейс, локалізація та налаштування як необхідні елементи сучасної гри подібного виду. З двох проаналізованих аналогів було винесено проблеми, які будуть вирішені в додатку.

Був сформований концепт-документ, який описує особливості додатку: можливість притягуватись до стін гравцем, різноманітність предметів та GUI, який з'являється біля персонажа. Гра отримала назву «Interim», візуальний стиль 2D піксель-арт, платформа – ПК, локалізація – українська та англійська мови.

Стек технологій обирався на основі визначених вимог. При створенні було використано ігровий рушій Unity та мова програмування C# як основні інструменти проектування. Для графічних елементів в обраному стилі використовувався спеціальний графічний редактор Aseprite.

Для реалізації персонажа, ворогів, предметів, елементів локацій, GUI та внутрішньої логіки гри, такі як система збережень та стани гри, використовувався об'єктно-орієнтований підхід та патерни проектування: одинак, стан, спостерігач, декоратор, фасад, адаптер та об'єктний пул.

Персонажу та ворогам було реалізовано однакову систему поведінки, зброї та взаємодії для відповідності принципам ООП. Елементи ігрового циклу, такі як кнопка, ворота тощо, створені з можливістю формувати системи на локаціях для взаємодії з персонажем. Стани ігрового циклу спроектовані для коректної роботи додатка. В програмі забезпечено збереження даних об'єктів для їх подальшого відновлення за потреби гравця.

Спрайти для персонажа, ворогів, предметів, елементів локацій та GUI були намальовані та додані в карти спрайтів для оптимізації додатка. Вони можуть зберігати зображення предметів за спільними рисами та кадри анімацій.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		75

Були спроектовані елементи інтерфейсу, які в подальшому сформували меню GUI, та збережені значення тексту в кожному елементі для доданих мов. Анімації для об'єктів керуються програмним кодом. Для GUI анімації реалізовані за допомогою можливостей окремої надбудови рушія.

В додатку були реалізовані три локації: навчальна, Білий простір та Бінарний міст. Перша відповідає за навчання гравця механікам гри. Всі інші мають цікаве наповнення та залучають гравця до їх дослідження.

Для GUI були створені такі меню: головне, меню налаштувань та його підпункти, меню збережень, пауза та спрощені меню налаштувань та збережень в ньому, а також інвентар з підпунктами. Також реалізовані вікна завантаження, смерті та попередження перед виконанням певної дії.

Було проведено тестування додатку з залученням кількох людей, виявлені помилки були виправлені, та враховані поради щодо організації ігрового процесу.

Було складено інструкцію отримання ігрового додатку та його встановлення, мінімальні вимоги до системи, а також алгоритм користування додатком.

Реалізована гра готова до використання. У подальшому гра може бути розширена шляхом впровадження нових механік, функцій та локацій, а також розширення наявних елементів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bahar. What is Metroidvania? And the 10 Best Metroidvania Games – 2023. [Електронний ресурс]. – Режим доступу: <https://bahrr.medium.com/what-is-metroidvania-and-best-10-metroidvania-game-f986265ff726> (дата звернення 29.10.2023).
2. Team Cherry. Hollow Knight – 2017. [Електронний ресурс]. – Режим доступу: <https://www.hollowknight.com/> (дата звернення 02.11.2023).
3. The Game Kitchen. Blasphemous – 2019. (сторінка на Steam) [Електронний ресурс]. – Режим доступу: <https://store.steampowered.com/app/774361/Blasphemous/> (дата звернення 04.11.2023).
4. Unity Technologies. Unity – 2021. [Електронний ресурс]. – Режим доступу: <https://unity.com/> (дата звернення 06.11.2023).
5. Microsoft Learn. C# language documentation – 2023. [Електронний ресурс]. – Режим доступу: <https://learn.microsoft.com/en-us/dotnet/csharp/> (дата звернення 08.11.2023).
6. JetBrains. Rider – 2023. [Електронний ресурс]. – Режим доступу: <https://www.jetbrains.com/rider/> (дата звернення 13.11.2023).
7. GitHub. GitHub Desktop – 2023. [Електронний ресурс]. – Режим доступу: <https://desktop.github.com/> (дата звернення 15.11.2023).
8. Igarra Studio. Aseprite – 2023. [Електронний ресурс]. – Режим доступу: <https://aseprite.org/> (дата звернення 16.11.2023).
9. PixelLab Docs – 2024. [Електронний ресурс]. – Режим доступу: <https://www.pixellab.ai/docs> (дата звернення 23.03.2024).
10. Refactoring.guru. State – 2024. [Електронний ресурс]. – Режим доступу: <https://refactoring.guru/uk/design-patterns/state> (дата звернення 06.01.2024).
11. Refactoring.guru. Adapter – 2024. [Електронний ресурс]. – Режим доступу: <https://refactoring.guru/uk/design-patterns/adapter> (дата звернення 07.01.2024).

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		77

12. Refactoring.guru. Facade – 2024. [Електронний ресурс]. – Режим доступу: <https://refactoring.guru/uk/design-patterns/facade> (дата звернення 14.01.2024).
13. Microsoft Learn. Generic classes and methods – 2024. [Електронний ресурс]. – Режим доступу: <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics> (дата звернення 24.01.2024).
14. Refactoring.guru. Decorator – 2024. [Електронний ресурс]. – Режим доступу: <https://refactoring.guru/uk/design-patterns/decorator> (дата звернення 10.01.2024).
15. Unity Documentation. Physics2D – 2021. [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/ScriptReference/Physics2D.html> (дата звернення 06.01.2024).
16. Unity Documentation. Rigidbody2D – 2021. [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/Manual/rigidbody2D.html> (дата звернення 06.01.2024).
17. Unity Documentation. ScriptableObject – 2021. [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/Manual/class-ScriptableObject.html> (дата звернення 27.01.2024).
18. Unity Documentation. Prefabs – 2021. [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/Manual/Prefabs.html> (дата звернення 13.01.2024).
19. Refactoring.guru. Observer – 2024. [Електронний ресурс]. – Режим доступу: <https://refactoring.guru/uk/design-patterns/observer> (дата звернення 01.02.2024).
20. Astemir Eleev. Object Pool – 2023. [Електронний ресурс]. – Режим доступу: <https://github.com/eleev/swift-design-patterns/blob/main/Common%20Design%20Patterns/Creational/ObjectPool/ObjectPool.md> (дата звернення 04.02.2024).

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		78

21. Refactoring.guru. Singleton – 2024. [Електронний ресурс]. – Режим доступу: <https://refactoring.guru/uk/design-patterns/singleton> (дата звернення 06.01.2024).

22. Unity Documentation. PlayerPrefs – 2021. [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/ScriptReference/PlayerPrefs.html> (дата звернення 16.03.2024).

23. Unity Documentation. Animation – 2019. [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/2019.4/Documentation/Manual/AnimationSection.html> (дата звернення 21.02.2024).

24. Demigiant. DOTween Documentation – 2024. [Електронний ресурс]. – Режим доступу: <https://dotween.demigiant.com/documentation.php> (дата звернення 13.04.2024).

25. Unity Documentation. TextMeshPro – 2021. [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/Manual/com.unity.textmeshpro.html> (дата звернення 09.04.2024).

26. Unity Packages. Localization – 2021. [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/Packages/com.unity.localization@1.0/manual/index.html> (дата звернення 21.04.2024).

27. Unity Packages. 2D Pixel Perfect – 2023. [Електронний ресурс]. – Режим доступу: <https://docs.unity3d.com/Packages/com.unity.2d.pixel-perfect@1.0/manual/index.html> (дата звернення 11.03.2024).

28. UKRFonts – 2024. [Електронний ресурс]. – Режим доступу: <https://ukrfonts.com/> (дата звернення 06.04.2024).

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						79
Змн.	Арк.	№ докум.	Підпис	Дата		

ДОДАТКИ

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		80

Лістинг батьківського узагальненого класу для наземних станів

PlayerGroundedState

```
using UnityEngine;

public class PlayerGroundedState : PlayerState
{
    protected Vector2 movement;
    protected int xInput;

    private bool _isDash;
    private bool _isWalk;

    private bool _isJump;
    private bool _isGrab;

    private bool _isGrappling;

    private bool _isGrounded;
    private bool _isTouchingWall;

    public PlayerGroundedState(
        Player player, StateMachine stateMachine)
        : base(player, stateMachine)
    {
    }

    public override void DoChecks()
    {
        base.DoChecks();
        if (stateMachine.CurrentState == core.DirectionState)
        {
            stateMachine.ChangeState(core.DirectionState);
            return;
        }

        _isGrounded = core.CheckIfGrounded();
        _isTouchingWall = core.CheckIfTouchingWall();
    }

    public override void Enter()
    {
        base.Enter();

        core.JumpState.ResetAmountOfJumpsLeft();
        core.DashState.ResetDashPosibility();
    }

    public override void Exit()
    {
        base.Exit();
    }

    public override void LogicUpdate()
    {
        base.LogicUpdate();

        xInput = core.InputHandler.NormalizedInputX;
        core.CheckIfShouldFlip(xInput);

        animatorController.SetMovement(xInput);

        _isWalk = core.InputHandler.IsWalk;
        _isDash = core.InputHandler.IsDash;
        _isJump = core.InputHandler.IsJump;
        _isGrab = core.InputHandler.IsGrab;
        _isGrappling = core.InputHandler.IsGrappling;
    }
}
```

```

        if (core.InputHandler.IsAttackInput[(int)CombatInputs.Primary])
        {
            stateMachine.ChangeState(core.PrimaryAttackState);
        }
        else if (core.InputHandler.IsAttackInput[(int)CombatInputs.Secondary])
        {
            stateMachine.ChangeState(core.SecondaryAttackState);
        }
        else if (_isJump && core.JumpState.CheckJumpPosibility())
        {
            core.InputHandler.UseJumpInput();
            stateMachine.ChangeState(core.JumpState);
        }
        else if (!_isGrounded)
        {
            core.InAirState.StartCoyoteTime();
            stateMachine.ChangeState(core.InAirState);
        }
        else if (_isTouchingWall && _isGrab)
        {
            stateMachine.ChangeState(core.WallGrabState);
        }
        else if (_isDash && core.DashState.CheckDashPosibility())
        {
            stateMachine.ChangeState(core.DashState);
        }
        else if (_isGrappling)
        {
            stateMachine.ChangeState(core.GrapplingState);
        }
        else if (_isWalk)
        {
            stateMachine.ChangeState(core.WalkState);
        }
    }

    public override void PhysicsUpdate()
    {
        base.PhysicsUpdate();
    }
}

```

Додаток Б

Лістинг наземного стану персонажа PlayerMoveState

```

public class PlayerMoveState : PlayerGroundedState
{
    public PlayerMoveState(
        Player player, StateMachine stateMachine)
        : base(player, stateMachine)
    {
    }

    public override void LogicUpdate()
    {
        base.LogicUpdate();

        core.SetVelocityX(stats.CurrentStats.sprintSpeed * xInput);

        if (xInput == 0f && !isExitingState)
        {
            stateMachine.ChangeState(core.IdleState);
        }
    }
}

```

Лістинг фасаду підсистем CoreSystem

```
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class CoreSystem : MonoBehaviour
{
    public ICharacterController Controller { get; private set; }

    private readonly List<CoreComponent> CoreComponents = new List<CoreComponent>();

    private void Awake()
    {
        Controller = transform.parent.GetComponent<ICharacterController>();

        if (Controller == null)
        {
            Controller =
transform.parent.parent.GetComponent<ICharacterController>();
        }
    }

    public void Update()
    {
        foreach (CoreComponent component in CoreComponents)
        {
            component.LogicUpdate();
        }
    }

    public void AddComponent(CoreComponent component)
    {
        if (!CoreComponents.Contains(component))
        {
            CoreComponents.Add(component);
        }
    }

    public T GetCoreComponent<T>() where T : CoreComponent
    {
        var comp = CoreComponents.OfType<T>().FirstOrDefault();

        if (comp)
            return comp;

        comp = GetComponentInChildren<T>();

        if (comp)
            return comp;

        Debug.LogWarning($"{typeof(T)} not found on {transform.parent.name}");
        return null;
    }

    public T GetCoreComponent<T>(ref T value) where T : CoreComponent
    {
        value = GetCoreComponent<T>();
        return value;
    }
}
```

Лістинг підсистеми параметрів сутності StatsController

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine.Events;
using UnityEngine;
using UnityEngine.Serialization;

[DefaultExecutionOrder(0)]
public class StatsController : CoreComponent
{
    [Header("Stats")]
    [SerializeField] private CharacterStats _defaultStats;
    [SerializeField] private GlobalStats _defaultGlobalStats;

    [field: SerializeField, Header("Events")] public UnityEvent OnStatsChanging {
        get; private set; }

    private GlobalStats _currentGlobalStats;

    private IStatsProvider _currentStats;
    private ModifiedStats _mainStats;

    public CharacterStats DefaultStats => _defaultStats;
    public CharacterStats CurrentStats => _currentStats.GetStats();
    public CharacterStats MainStats => _mainStats.GetStats();
    public GlobalStats GlobalStats => _currentGlobalStats;

    protected override void Awake()
    {
        base.Awake();

        _currentStats = new DefaultStats(_defaultStats);
        _mainStats = new ModifiedStats(_currentStats);

        _currentStats = _mainStats;
        _currentGlobalStats = _defaultGlobalStats;
    }

    public void ChangeStats(IStatsProvider modification)
    {
        _currentStats = modification;
        OnStatsChanging.Invoke();
    }

    public void AddStateModifier(StatsModifier modifier)
    {
        _currentStats.AddModifier(modifier);
        OnStatsChanging.Invoke();
    }

    public void RemoveStateModifier(StatsModifier modifier)
    {
        _currentStats.RemoveModifier(modifier);
        OnStatsChanging.Invoke();
    }

    public void ReturnDefaultGlobalStats() => _currentGlobalStats =
        _defaultGlobalStats;
}
```

Лістинг структури даних параметрів сутності CharacterStats та модифікатора структури даних StatsModifier

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[Serializable]
public struct CharacterStats
{
    [Header("Movement")]
    public float moveSpeed;
    public float sprintSpeed;

    [Space(3)]

    [Header("Health")]
    public float healthPoints;

    [Space(3)]

    [Header("Jump")]
    public int amountOfJumps;
    public float jumpHeight;

    [Space(3)]

    [Header("Dash")]
    public int amountOfDashes;
    public float dashTime;
    public float dashCooldown;
    public float dashStrength;

    [Space(3)]

    [Header("Grappling")]
    public float grapplingMaxDistance;

    [Space(3)]

    [Header("Stamina")]
    public int stamina;
    public float staminaCooldown;

    [Space(3)]

    [Header("Damage")]
    public float damage;
}

[Serializable]
public struct StatsModifier
{
    [Header("Stats")]
    public CharacterStats stats;

    [Header("Modifier Type")]
    public StatsModifierType modifierType;
}
```

Лістинг класу ModifiedStats

```

using System.Collections.Generic;

public class ModifiedStats : StatsDecorator
{
    private List<StatsModifier> _modifiersList = new List<StatsModifier>();

    public ModifiedStats(IStatsProvider wrappedEntity) : base(wrappedEntity)
    {
    }

    public override void AddModifier(StatsModifier modifier)
    {
        _modifiersList.Add(modifier);
    }

    public override void RemoveModifier(StatsModifier modifier)
    {
        _modifiersList.Remove(modifier);
    }

    protected override CharacterStats GetStatsInternal()
    {
        CharacterStats result = _wrappedEntity.GetStats();

        List<StatsModifier> add = _modifiersList.FindAll(x => x.modificatorType ==
StatsModifierType.Add);
        List<StatsModifier> multiply = _modifiersList.FindAll(x => x.modificatorType
== StatsModifierType.Multiply);

        foreach (StatsModifier modifier in multiply)
        {
            result *= modifier.stats;
        }

        foreach (StatsModifier modifier in add)
        {
            result += modifier.stats;
        }

        return result;
    }
}

```

Лістинг класу HealthController

```

using System;
using UnityEngine;
using UnityEngine.Events;

[DefaultExecutionOrder(1)]
public class HealthController : CoreComponent, IDamagable
{
    [Header("Bar")]
    [SerializeField] private BarController bar;

    [Header("Effects")]
    [SerializeField] private GameObject healParticles;
    [SerializeField] private GameObject damageParticles;

    [Header("Events")]
    [SerializeField] private UnityEvent onHeal;
}

```

```

[SerializeField] private UnityEvent onDamage;
[SerializeField] private UnityEvent onDeath;
[field: SerializeField] public UnityEvent OnHealthChanging { get; private set; }

private StatsController stats;
private ParticleController particleController;

public Stat Health { get; private set; }

protected override void Awake()
{
    base.Awake();

    stats = core.GetCoreComponent<StatsController>();
    stats.OnStatsChanging.AddListener(UpdateStats);
    particleController = core.GetComponent<ParticleController>();

    Health = new Stat();

    UpdateStats();
    bar.ResetValues();
    Health.Init();

    Health.OnCurrentValueZero += DoDeath;
}

public void Damage(float damage)
{
    if (damageParticles)
        particleController.StartParticles(damageParticles);

    Health.Decrease(MathF.Round(damage, 1));

    bar.CurrentValue = Health.CurrentValue;

    onDamage?.Invoke();
    OnHealthChanging?.Invoke();
}

public void GetHeal(float value)
{
    Health.Increase(MathF.Round(value, 1));

    bar.CurrentValue = Health.CurrentValue;

    if (!Health.CurrentValue.Equals(Health.MaxValue))
    {
        if (healParticles)
            particleController.StartParticles(healParticles);

        onHeal?.Invoke();
        OnHealthChanging?.Invoke();
    }
}

public void DoDeath()
{
    Health.OnCurrentValueZero -= DoDeath;

    core.Controller.DoDeath();

    onDeath?.Invoke();
}

public void UpdateStats()
{
    Health.SetMaxValue(stats.CurrentStats.healthPoints);
    bar.MaxValue = stats.CurrentStats.healthPoints;
}

private void OnDestroy()

```

```

    {
        stats.OnStatsChanging.RemoveListener(UpdateStats);
    }
}

```

Додаток И

Лістинг класу InventoryController

```

using System;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using UnityEngine.Events;

public class InventoryController : CoreComponent
{
    [SerializeField] private int maxWeapons;
    [SerializeField] private int maxAccessories;

    [field: SerializeField, Space(5)] public List<ScriptableItem> Items { get;
private set; }

    [field: SerializeField, Space(5)] public List<WeaponScriptableItem>
EquippedWeapons { get; private set; }
    [field: SerializeField] public List<ScriptableItem> EquippedArtifacts { get;
private set; }

    [field: SerializeField, Space(5)] public UnityEvent OnInventoryChange { get;
private set; }

    private ItemEffectSpawner effectSpawner;
    private WeaponSwitcher weaponSwitcher;

    protected void Start()
    {
        Initialize();
    }

    protected void Initialize()
    {
        effectSpawner =
transform.parent.parent.GetComponentInChildren<ItemEffectSpawner>();
        weaponSwitcher = GetComponent<WeaponSwitcher>();

        if (EquippedWeapons == null)
            EquippedWeapons = new();
        else
            foreach (var weapon in EquippedWeapons)
                effectSpawner.SpawnItemEffects((ScriptableItem) weapon);

        if (EquippedArtifacts == null)
            EquippedArtifacts = new();
        else
            foreach (var artifact in EquippedArtifacts)
            {
                AddCharacterStatsModifier(artifact);
                effectSpawner.SpawnItemEffects(artifact);
            }

        weaponSwitcher.SwitchWeapon(EquippedWeapons);

        OnInventoryChange.Invoke();
    }

    public void AddItem(ScriptableItem item)
    {
        Items.Add(item);
        OnInventoryChange?.Invoke();
    }
}

```

```

    }

    public bool AddUniqueItem(ScriptableItem item)
    {
        if (!Items.Contains(item))
        {
            AddItem(item);
            return true;
        }

        return false;
    }

    public bool CheckItem(ScriptableItem item) =>
        Items.Contains(item) || CheckEquippedItem(item);

    public bool CheckEquippedItem(ScriptableItem item) =>
        (item.type == ItemType.Weapon &&
        EquippedWeapons.Contains((WeaponScriptableItem) item))
        || EquippedArtifacts.Contains(item);

    public bool RemoveItem(ScriptableItem item)
    {
        if (Items.Contains(item))
        {
            Items.Remove(item);
            OnInventoryChange?.Invoke();

            if (item.type == ItemType.Food)
                effectSpawner.SpawnItemEffects(item);

            return true;
        }

        return false;
    }

    public bool EquipItem(ScriptableItem item)
    {
        if (!Items.Contains(item))
            return false;

        if (CheckEquippedItem(item))
            return false;

        if (item.type == ItemType.Artifacts && EquippedArtifacts.Count <
maxAccessories)
        {
            AddCharacterStatsModifier(item);
            EquippedArtifacts.Add(item);
        }
        else if (item.type == ItemType.Weapon && EquippedWeapons.Count < maxWeapons)
        {
            EquippedWeapons.Add((WeaponScriptableItem) item);
            weaponSwitcher.SwitchWeapon(EquippedWeapons);
        }
        else
            return false;

        effectSpawner.SpawnItemEffects(item);

        Items.Remove(item);

        OnInventoryChange?.Invoke();
        return true;
    }

    public bool UnequipItem(ScriptableItem item)
    {
        if (!CheckEquippedItem(item))
            return false;

```

```

        if (item.type == ItemType.Artifacts)
        {
            EquippedArtifacts.Remove(item);
            RemoveCharacterStatsModifier(item);
        }
        else if (item.type == ItemType.Weapon)
        {
            EquippedWeapons.Remove((WeaponScriptableItem) item);
            weaponSwitcher.SwitchWeapon(EquippedWeapons);
        }

        effectSpawner.RemoveItemEffects(item);

        Items.Add(item);

        OnInventoryChange?.Invoke();
        return true;
    }

    private void AddCharacterStatsModifier(ScriptableItem item) =>
        core.GetCoreComponent<StatsController>().AddStateModifier(item.stats);

    private void RemoveCharacterStatsModifier(ScriptableItem item) =>
        core.GetCoreComponent<StatsController>().RemoveStateModifier(item.stats);
}

```

Додаток К

Лістинг класу Weapon

```

using System;
using UnityEngine;

public class Weapon : MonoBehaviour
{
    public WeaponData Data { get; private set; }
    [SerializeField] private float attackCounterResetCooldown;

    public float AttackStartTime { get; private set; }

    public int CurrentAttackCounter
    {
        get => currentAttackCounter;
        private set => currentAttackCounter = value >= Data.NumberOfAttacks ? 0 :
value;
    }

    public bool CurrentInput
    {
        get => currentInput;
        set
        {
            if (currentInput != value)
            {
                currentInput = value;
                OnCurrentInputChange?.Invoke(currentInput);
            }
        }
    }

    public Vector2 CurrentDirection { get; set; }

    public event Action<bool> OnCurrentInputChange;
    public event Action OnEnter;
    public event Action OnExit;

    private Animator anim;
    private WeaponAnimationEventHandler eventHandler;
    public GameObject BaseGameObject { get; private set; }
}

```

```

public GameObject WeaponSpriteGameObject { get; private set; }

public WeaponAnimationEventHandler EventHandler
{
    get
    {
        if (!initDone)
            GetDependencies();

        return eventHandler;
    }
    private set => eventHandler = value;
}

public CoreSystem Core { get; private set; }

private int currentAttackCounter;

private ActionTimer attackCounterResetTimer;

private bool currentInput;
private bool initDone;
public bool CanEnterAttack { get; private set; }

private void Awake()
{
    GetDependencies();

    attackCounterResetTimer = new ActionTimer(attackCounterResetCooldown);
}

private void Update()
{
    attackCounterResetTimer.Tick();
}

public void Enter()
{
    //print($"{transform.name} enter");

    AttackStartTime = Time.time;

    attackCounterResetTimer.StopTimer();

    int facingDirection = Core.GetCoreComponent<MovementSet>().FacingDirection;

    anim.SetBool("active", true);
    anim.SetInteger("counter", CurrentAttackCounter);
    anim.SetFloat("direction", facingDirection);

    if (facingDirection > 0)
        WeaponSpriteGameObject.GetComponent<SpriteRenderer>().flipX = true;
    else
        WeaponSpriteGameObject.GetComponent<SpriteRenderer>().flipX = false;

    OnEnter?.Invoke();
}

private void GetDependencies()
{
    if (initDone)
        return;

    BaseGameObject = transform.Find("Base").gameObject;
    WeaponSpriteGameObject = transform.Find("WeaponSprite").gameObject;
    anim = BaseGameObject.GetComponent<Animator>();

    EventHandler = BaseGameObject.GetComponent<WeaponAnimationEventHandler>();

    initDone = true;
}

```

```

    public void SetCore(CoreSystem core)
    {
        Core = core;
    }

    public void SetData(WeaponData data)
    {
        Data = data;

        if (Data != null)
            return;

        ResetAttackCounter();
    }

    private void Exit()
    {
        anim.SetBool("active", false);

        CurrentAttackCounter++;
        attackCounterResetTimer.StartTimer();

        OnExit?.Invoke();
    }

    private void ResetAttackCounter() => CurrentAttackCounter = 0;
    public void SetCanEnterAttack(bool value) => CanEnterAttack = value;

    private void OnEnable()
    {
        EventHandler.OnFinish += Exit;
        attackCounterResetTimer.OnTimerDone += ResetAttackCounter;
    }

    private void OnDisable()
    {
        EventHandler.OnFinish -= Exit;
        attackCounterResetTimer.OnTimerDone -= ResetAttackCounter;
    }
}

```

Додаток Л

Лістинг класу WeaponGenerator

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class WeaponGenerator : MonoBehaviour
{
    public event Action OnWeaponGenerating;

    [SerializeField] private Weapon weapon;
    [SerializeField] private WeaponData data;

    private List<WeaponComponent> componentsAlreadyOnWeapon = new List<WeaponComponent>();
    private List<WeaponComponent> componentsAddedToWeapon = new List<WeaponComponent>();
    private List<Type> componentDependencies = new List<Type>();

    private Animator anim;

    private void Start()
    {
        anim = GetComponentInChildren<Animator>();
        GenerateWeapon(data);
    }
}

```

```

[ContextMenu("Test Generate")]
private void TestGeneration()
{
    GenerateWeapon(data);
}

public void GenerateNewWeapon(WeaponData data)
{
    this.data = data;
    GenerateWeapon(data);
}

public void GenerateWeapon(WeaponData data)
{
    OnWeaponGenerating?.Invoke();

    weapon.SetData(data);

    if (data is null)
    {
        weapon.SetCanEnterAttack(false);
        return;
    }

    componentsAlreadyOnWeapon.Clear();
    componentsAddedToWeapon.Clear();
    componentDependencies.Clear();

    componentsAlreadyOnWeapon = GetComponents<WeaponComponent>().ToList();
    componentDependencies = data.GetAllDependencies();

    foreach (var dependency in componentDependencies)
    {
        if (componentsAddedToWeapon.FirstOrDefault(component => component.GetType() ==
dependency)
            continue;

        var weaponComponent =
            componentsAlreadyOnWeapon.FirstOrDefault(component => component.GetType() ==
dependency);

        if (weaponComponent == null)
        {
            weaponComponent = gameObject.AddComponent(dependency) as WeaponComponent;
        }

        weaponComponent.Init();

        componentsAddedToWeapon.Add(weaponComponent);
    }

    var componentsToRemove = componentsAlreadyOnWeapon.Except(componentsAddedToWeapon);

    foreach (var component in componentsToRemove)
    {
        Destroy(component);
    }

    anim.runtimeAnimatorController = data.AnimatorController;

    weapon.SetCanEnterAttack(true);
}
}

```

Лістинг класу DefaultTrigger

```

using System;
using UnityEngine;
using UnityEngine.Events;

public class DefaultTrigger : MonoBehaviour
{
    [SerializeField] private LayerMask triggerLayers;
    [SerializeField] private UnityEvent onTriggerEnter;
    [SerializeField] private UnityEvent onTriggerExit;

    [SerializeField] private bool isOnce = false;
    private bool isUsed = false;
    private bool isActive = false;

    private void OnTriggerEnter2D(Collider2D collider)
    {
        if (CheckLayerInLayerMask(collider.gameObject.layer) && !isActive)
        {
            if (isOnce && isUsed)
                return;

            isActive = true;
            onTriggerEnter?.Invoke();
            isUsed = true;
            return;
        }
    }

    private void OnTriggerExit2D(Collider2D collider)
    {
        if (CheckLayerInLayerMask(collider.gameObject.layer) && isActive)
        {
            if (isOnce && isUsed)
                return;
            isActive = false;
            onTriggerExit?.Invoke();
            return;
        }
    }

    private bool CheckLayerInLayerMask(int layer) => (triggerLayers.value & 1 <<
layer) > 0;
}

```

Лістинг класу SessionSaveSystem

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using UnityEngine;
using UnityEngine.Events;

[DefaultExecutionOrder(1)]
public class SessionSaveSystem : MonoBehaviour
{
    [SerializeField] private UnityEvent onSave;

    private int savesCount = 0;

    private void Awake()
    {

```

```

        if (!String.IsNullOrEmpty(PlayerPrefs.GetString("Saves")))
        {
            SavesStructure saves =
                JsonUtility.FromJson<SavesStructure>(PlayerPrefs.GetString("Saves"));
            savesCount = saves.saves.Count;
        }
    }

    private void Start()
    {
        // TODO: Looks strange using fixed name of next scene
        LoadSceneManager.Instance.onLoadingPreparing.AddListener(() =>
        {
            if (LoadSceneManager.Instance.nextSceneName != "MainMenu")
                SaveInBuffer();
            else
                CleanBuffer();
        });
    }

    #region Public Functions
    [ContextMenu("Save Data")]
    public void Save()
    {
        var data = LoadFile(GetExistingSavePath());
        GetFromBuffer(data);
        GetData(data);
        SaveFile(data, GetExistingSavePath());

        CleanBuffer();

        onSave?.Invoke();
    }

    [ContextMenu("Save Data To New")]
    public void SaveToNew()
    {
        var data = LoadFile(GetExistingSavePath());
        GetFromBuffer(data);
        GetData(data);
        SaveFile(data, GetNewSavePath());

        CleanBuffer();

        onSave?.Invoke();
    }

    [ContextMenu("Load Data")]
    public void Load()
    {
        var data = LoadFile(GetExistingSavePath());
        GetFromBuffer(data);
        RestoreData(data);
    }
    #endregion

    #region Data Exluatation
    private void SaveFile(object state, string path)
    {
        using (var stream = File.Open(path, FileMode.Create))
        {
            var formatter = new BinaryFormatter();
            formatter.Serialize(stream, state);
        }
    }

    private Dictionary<string, object> LoadFile(string path)
    {
        if (!File.Exists(path))
        {
            return new Dictionary<string, object>();
        }
    }

```

```

    }

    using (FileStream stream = File.Open(path, FileMode.Open))
    {
        var formatter = new BinaryFormatter();
        return (Dictionary<string, object>)formatter.Deserialize(stream);
    }
}

private void GetData(Dictionary<string, object> data)
{
    foreach (var savable in FindObjectsOfType<SavableEntity>())
    {
        data[savable.Id] = savable.GetSavedData();
    }
}

private void RestoreData(Dictionary<string, object> data)
{
    foreach (var savable in FindObjectsOfType<SavableEntity>())
    {
        if (data.TryGetValue(savable.Id, out object value))
        {
            savable.RestoreSavedData(value);
        }
    }
}
}
#endregion

#region Buffer Exluatation

public void SaveInBuffer()
{
    var data = new Dictionary<string, object>();

    if (PlayerPrefs.HasKey("SaveBuffer"))
        data = JsonUtility.FromJson<Dictionary<string, object>>(PlayerPrefs.GetString("SaveBuffer"));

    GetData(data);
    PlayerPrefs.SetString("SaveBuffer", JsonUtility.ToJson(data));
}

private void GetFromBuffer(Dictionary<string, object> data)
{
    if (PlayerPrefs.HasKey("SaveBuffer"))
    {
        var bufferData = JsonUtility.FromJson<Dictionary<string, object>>(PlayerPrefs.GetString("SaveBuffer"));
        foreach (var key in bufferData.Keys)
            data[key] = bufferData[key];
    }
}

private void CleanBuffer()
{
    PlayerPrefs.DeleteKey("SaveBuffer");
}
}
#endregion

#region Path Generation Function
private string GetNewSavePath()
{
    string path;

    savesCount++;

    SavesStructure saves =
    JsonUtility.FromJson<SavesStructure>(PlayerPrefs.GetString("Saves"));
    saves.currentSave = $"Save{savesCount}";
    saves.saves.Insert(0, saves.currentSave);
}

```

```

        PlayerPrefs.SetString("Saves", JsonUtility.ToJson(saves));

        path = $"{saves.savePath}{saves.currentSave}.save";

        return path;
    }

    private string GetExistingSavePath()
    {
        string path;

        try
        {
            SavesStructure saves =
                JsonUtility.FromJson<SavesStructure>(PlayerPrefs.GetString("Saves"));

            if (!String.IsNullOrEmpty(saves.currentSave))
            {
                saves.saves.Remove(saves.currentSave);
                saves.saves.Insert(0, saves.currentSave);
                path = $"{saves.savePath}{saves.currentSave}.save";
                PlayerPrefs.SetString("Saves", JsonUtility.ToJson(saves));
            }
            else
            {
                path = GetNewSavePath();
                Debug.Log("Save file is not find");
            }
        }
        catch
        {
            SavesStructure saves = new SavesStructure
            {
                currentSave = "",
                savePath = $"{Application.persistentDataPath}/",
                saves = new List<string>()
            };

            PlayerPrefs.SetString("Saves", JsonUtility.ToJson(saves));
            path = GetNewSavePath();

            Debug.Log("Save file is not find with error");
        }

        return path;
    }
    #endregion

    private void OnApplicationQuit()
    {
        CleanBuffer();
    }
}

[Serializable]
public struct SavesStructure
{
    public string currentSave;
    public string savePath;
    [SerializeReference] public List<string> saves;
}

```

Лістинг класу SessionInfoManager

```
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class SessionInfoManager : MonoBehaviour, ISavable
{
    public static SessionInfoManager Instance { get; private set; }

    public SessionInfoData InfoData => infoData;

    private SessionInfoData infoData;

    public void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        else
        {
            Destroy(this);
        }
    }

    public object GetData()
    {
        if (infoData.locationKey == null)
            infoData.locationKey = "whitePlace";

        if (infoData.completedTasks == null)
            infoData.completedTasks = new List<string>();

        infoData.sceneName = SceneManager.GetActiveScene().name;
        infoData.time = GameTimeCounter.Instance.GameTime;
        return infoData;
    }

    public void RestoreData(object data)
    {
        infoData = (SessionInfoData)data;
    }

    public void MarkNewCompletedTasks(string name)
    {
        if (infoData.completedTasks == null)
            infoData.completedTasks = new List<string>();

        if (!infoData.completedTasks.Contains(name))
            infoData.completedTasks.Add(name);
    }

    public void SetNewLocationKey(string key)
    {
        infoData.locationKey = key;
    }
}
```