

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:

«Розробка кор-механік та дизайну рівнів для гри в жанрах
Metroidvania і Soulslike»

Виконав студент групи КН-20-1:
Кірій Даніла Олегович

Керівник роботи:
МАРЧУК Галина Вікторівна

Рецензент: Левківський В.Л.

Житомир-2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК
СПЕЦІАЛЬНІСТЬ 122 «Комп'ютерні науки»

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: КІРІЙ Даніла Олегович
Керівник роботи: МАРЧУК Галина Вікторівна
Тема роботи: «Розробка кор-механік та дизайну рівнів для гри в жанрах Metroidvania і Soulslike»,
затверджена наказом закладу вищої освіти від «25» січня 2024 р., №22/с
Термін здачі закінченої роботи: «06» червня 2024 р.
Вихідні дані роботи (зазначається предмет і об'єкт дослідження): ігровий рушій Unity та його можливості для реалізації ігрових механік із використанням патернів програмування.

Консультанти з кваліфікаційної роботи із зазначенням розділів, що їх стосуються

Розділ	Консультант	Дата	
		Завдання видав	Завдання прийняв
1	Марчук Г.В.	11.11.2023	11.11.2023
2	Марчук Г.В.	13.01.2024	13.01.2024
3	Марчук Г.В.	30.03.2024	30.03.2024

Календарний план

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Постановка задачі. Пошук, огляд, аналіз доступного інструментарію та аналогічного ПЗ.	11.11.2023-15.12.2023	Виконано
2	Створення схем та структури додатку.	13.01.2024-20.02.2024	Виконано
3	Розробка програмної частини.	25.02.2024-19.05.2024	Виконано
4	Тестування.	21.05.2024-03.06.2024	Виконано
5	Попередній захист.	06.06.2024	Виконано

Здобувач вищої освіти Даніла КІРІЙ
Керівник Галина МАРЧУК

РЕФЕРАТ

Кваліфікаційна робота складається з програмного додатку гри та пояснювальної записки

Пояснювальна записка до кваліфікаційної роботи складається з реферату, вступу, основної частини, загального висновку, списку використаної літератури та додатку.

Текстова частина викладена на 62 сторінках друкованого тексту.

Пояснювальна записка має 19 сторінок додатків. Список використаних джерел містить 20 найменувань, і займає 1 сторінку.

В роботі наведено 17 рисунків та 1 таблицю. Загальний обсяг роботи-85 сторінок.

Ключові слова: GAME DESIGN, METROIDVANIA, SOULSLIKE, UNITY, 2D, LEVEL DESIGN, STATE MASHINE.

					КН.КР.Б-122-24-ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата	«Розробка кор-механік та дизайну рівнів для гри в жанрах Metroidvania і Soulslike» Пояснювальна записка	Літ.	Арк.	Аркушів
Розроб.		Кірій Д.О.						
Керівник		Марчук Г.В.					2	85
Реценз.		Левківський В.Л.				Житомирська політехніка, група КН-20-1		
Н. Контр.								
Затверд.		Граф М.С.						

ABSTRACT

The graduation thesis of the bachelor consists of a software application of the game and an explanatory note.

An explanatory note to the final thesis consists of an abstract, introduction, main part, general conclusion, a list of used literature and an appendix.

The text part is laid out on 62 pages of printed text.

The explanatory note has 19 pages of annexes. The list of used sources contains 20 names and occupies 1 pages.

The work contains 17 figures and 1 table. The total volume of work is 85 pages.

Keywords: GAME, METROIDVANIA, SOULSLIKE, UNITY, 2D, PLATFORMER, STATE MASHINE, URP, GAME INTELLIGENCE.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		4

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	6
ВСТУП	7
РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ	9
1.1 Постановка задачі	9
1.2 Аналіз існуючого програмного забезпечення за тематикою кваліфікаційної роботи	10
1.3 Концепт-документ	16
1.4 Аналіз та вимоги до програмного забезпечення	17
Висновки до першого розділу	19
РОЗДІЛ 2. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	20
2.1 Розробка архітектури механік в ігровому додатку	20
2.2 Розробка головного персонажа	25
2.3 Реалізація здібностей героя та зброї	35
2.4 Створення системи чармів	48
Висновки до другого розділу	53
РОЗДІЛ 3. ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО ТЕСТУВАННЯ	55
3.1 Порядок встановлення та налаштування системи	55
3.2 Опис дизайну рівнів	56
3.3 Тестування роботи механік	59
Висновки до третього розділу	61
ВИСНОВКИ	62
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	63
ДОДАТКИ	65

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ПК	Персональних комп'ютер
ПЗ	Програмне забезпечення
RPG	Role-Playing Game - це жанр гри, у якому гравці виконують ролі вигаданих персонажів і проходять різноманітні сюжетні лінії
UML	Unified Modeling Language - уніфікована мова моделювання, використовується у парадигмі об'єктно-орієнтованого програмування

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		6

ВСТУП

Актуальність теми: На сьогоднішній день відеоігри займають дуже важливе місце в культурному та економічному житті. Ця індустрія є більш прибутковою, ніж кінематограф і музична індустрія. За даними Newzoo, світовий ринок відеоігор досягне приблизно 200 мільярдів доларів США у 2023 році і має тенденцію до зростання. Відеоігри стали не лише формою розваг, але й способом соціалізації, навчання та навіть мистецтвом. Жанри Metroidvania (ігри, що поєднують у собі елементи платформера та рольових ігор з нелінійним дослідженням світу) та Soulslike (ігри, натхненні серією *Dark Souls* з високою складністю та глибокою бойовою системою) набирають популярності останніми роками, і це можна пояснити унікальними ігровими механіками, глибиною геймплею та складністю. Давайте розглянемо їх ближче.

Metroidvania - це жанр платформер з відкритим світом і нелінійною прогресією. Гра представляє досить великі та взаємопов'язані локації, де гравець знаходить корисні предмети та здібності, які дозволяють отримати доступ до нових областей. Привабливість полягає у вільному дослідженні, інтерактивності та глибокій історії, яка спонукає до повторного проходження та відкриття нових секретів.

Soulslike - це піджанр екшн-ігор з високою складністю, суворими покараннями за помилки, стратегічними боями та глибоким сюжетом. Більшість ігор цього жанру отримали свою назву від серії *Dark Souls*, створеної компанією From Software. Привабливість цього жанру полягає у складності, яка змушує гравців бути точними та уважними, залишаючись стратегічними. Це дає гравцеві відчуття досягнення та винагороди за подолання складних проблем.

Metroidvania поєднує в собі найкращі елементи жанру з Soulslike для створення синергетичного ігрового досвіду. Поєднання підходу відкритого світу Metroidvania з нелінійною прогресією та складною бойовою механікою Soulslike дає дуже потужний та захоплюючий ігровий досвід. Рівень складності

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		7

та тактичні бої Soulslike поєднуються з постійним дослідженням світу та відкриттям нових здібностей у Metroidvania.

Враховуючи стрімко зростаючу популярність ігор в цьому жанрі, можна сміливо сказати, що обрана тема є актуальною та розроблена гра має всі шанси отримати увагу від гравців.

Мета кваліфікаційної роботи полягає у аналізі та розробці кор-механік і дизайну рівнів для відегри у жанрах Metroidvania та Soulslike. Це включає детальне вивчення ігрових механік, що забезпечують глибокий і захоплюючий ігровий досвід, а також розробку ефективних методів дизайну рівнів, які інтегрують ці механіки та сприяють створенню цілісного ігрового світу.

Об'єктом дослідження є технології реалізації ігрових механік із використанням патернів програмування.

Предметом дослідження є унікальні ігрові механіки, які формують геймплей відповідних жанрів, та методи дизайну рівнів, спрямовані на оптимізацію використання цих механік та створення іммерсивного ігрового середовища.

В рамках теми кваліфікаційної роботи було опубліковано тези:

Марчук Г.В., Кірій Д.О. Механіки та дизайн рівнів в жанрах «Metroidvania» та «Souls-Like». Комп'ютерні технології: інновації, проблеми, рішення: Тези VI Всеукраїнської науково-технічної конференції, 29-30 листопада 2023 року. Житомир : «Житомирська політехніка», 2023. С.192-194.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		8

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ

1.1 Постановка задачі

Метою розробки гри в жанрах Metroidvania та Soulslike є створення захоплюючого ігрового досвіду [1], що забезпечує глибоку іммерсію та виклик для гравців. Для досягнення цієї мети потрібно розробити та налаштувати комплекс кор-механік та дизайн рівнів, керування персонажем і його анімації [2], використовуючи патерн програмування «state machine» [3, 4].

Для досягнення цієї мети потрібно виконати наступні завдання:

1. Розробка головного персонажа:
 - Розробка керування головним персонажем.
 - Налаштування анімацій.
 - Імплементация патерну «state machine».
2. Розробка дизайну рівнів та локацій:
 - Створення деталізованих локацій, які сприяють експлорації та дослідженню.
 - Встановлення логічного зв'язку між різними локаціями та рівнями.
3. Аналіз та тестування:
 - Аналіз геймплейного досвіду для впевненості в його відповідності жанрам Metroidvania та Soulslike.
4. Розробка системи геймплейних механік:
 - Розробка та імплементация унікальних здібностей героя, які взаємодіють з оточенням.
 - Розробка стилів зброї та унікальних здібностей для кожного з них
 - Реалізація амулетів (чармів) з унікальними ефектами, які покращують геймплей.
 - Розробка системи талантів, що дозволяє гравцеві розвивати персонажа.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		9

1.2 Аналіз існуючого програмного забезпечення за тематикою кваліфікаційної роботи

Дослідження подібних до кваліфікаційної роботи проєктів, а також виявлення сильних і слабких сторін дозволить визначити найкращі практики та методи, які можуть бути застосовані під час розробки власного проєкту, а також виявити потенційні можливості для інновацій і вдосконалень. У ході аналізу буде розглянуто кілька популярних представників жанру, а також проведено порівняння їх функціональних можливостей та ефективності у вирішенні поставлених завдань.

1. The Last Faith (рис. 1.1)



Рисунок 1.1 - The Last Faith

The Last Faith [5] - це 2D-екшн RPG з елементами Roguelike (ігри, що відрізняються процедурною генерацією рівнів, перманентною смертю та покроковим геймплеєм), розроблена і видана Addictive Games. Гра вийшла 30 вересня 2023 року на ПК, Nintendo Switch, PlayStation 4, PlayStation 5.

Сюжет The Last Faith розгортається у похмурому та жорстокому світі, де людство веде відчайдушну боротьбу з ордами монстрів. Гравець втілюється у ролі Аркана (рис. 1.2), воїна, який шукає спосіб врятувати світ від загибелі.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		10



Рисунок 1.2 - Головний герой гри The Last Faith

Сильні сторони:

- Жорстокі та захоплюючі бої. The Last Faith пропонує динамічні бої, які постійно тримають вас у напрузі.
- Атмосферний та мальовничий світ. Світ The Last Faith - це темне та жорстоке місце, яке наповнене цікавими та небезпечними місцями.
- Глибока та емоційна історія. The Last Faith розповідає історію про втрату, самопожертву та пошук сенсу життя.
- Унікальна система крафту. Система крафту The Last Faith дозволяє вам створювати потужну зброю та броню, щоб стати сильнішим.

Недоліки:

- Повторюваний геймплей. The Last Faith може здатися дещо повторюваною, особливо після кількох годин гри.
- Відсутність інновацій. The Last Faith багато в чому схожа на інші ігри жанру roguelike, і не пропонує багато нового.
- Деякі технічні проблеми. Деякі гравці повідомляли про технічні проблеми, такі як збої та вильоти.

2. Blasphemous 2 (рис. 1.3)

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		11



Рисунок 1.3 - Blasphemous 2

Blasphemous 2 [6] - це продовження жорстокої та атмосферної 2D-метроїдванії Blasphemous, розробленої The Game Kitchen. Гра вийшла 24 серпня 2023 року на ПК, Nintendo Switch, PlayStation 4, PlayStation 5, Xbox One та Xbox Series X/S.

Сюжет Blasphemous 2 розгортається у світі, охопленому Чудом - руйнівною силою, що карає за гріхи. Гравець знову втілюється у ролі Каяника (рис. 1.4), проклятого воїна, який прагне розірвати вічне прокляття.



Рисунок 1.4 - Головний герой гри Blasphemous

Сильні сторони:

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		12

– Жорстокий та атмосферний світ. Cvstodia - це жорстокий та безжальний світ, який вражає своєю деталізацією та глибиною. Кожен куточок наповнений історією та небезпекою, що робить дослідження захоплюючим.

– Складні та динамічні бої. Бойова система Blasphemous 2 - це виклик, який потребує майстерності та винахідливості. Вам доведеться вивчати рухи ворогів, використовувати різноманітну зброю та здібності, щоб перемогти.

– Глибока та емоційна історія. Історія Каяника, проклятого воїна, який прагне розірвати вічне прокляття, захоплює та зворушує. Гра досліджує теми віри, прощення та самопожертви.

– Вдосконалена графіка та звук: Blasphemous 2 має мальовничу 2D-графіку, яка поєднує в собі елементи готики та фентезі. Саундтрек доповнює атмосферу гри та підкреслює її емоційні моменти.

Недоліки:

– Висока складність. Blasphemous 2 - це дуже складна гра, яка може здатися не всім. Вам доведеться багато вчитися та помилятися, щоб прогресувати.

– Відсутність інновацій. Blasphemous 2 багато в чому схожа на свою попередницю, і не пропонує багато нового в жанрі.

– Деякі технічні проблеми. Деякі гравці повідомляли про технічні проблеми, такі як збої та вильоти.

3. Moonscars (рис. 1.5)

Moonscars [7] - це 2D гра жанру Soulslike, розроблена Black Mermaid та видана Focus Home Interactive. Гра вийшла 29 вересня 2022 року на ПК, PlayStation 4 та PlayStation 5.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		13



Рисунок 1.5 - Moonscars

Сюжет Moonscars розгортається у похмурому та жорстокому світі Грандмар, де гравець втілюється у ролі Ембер (рис. 1.6) воїни, яка шукає ліки від таємничої хвороби, що перетворює людей на монстрів.

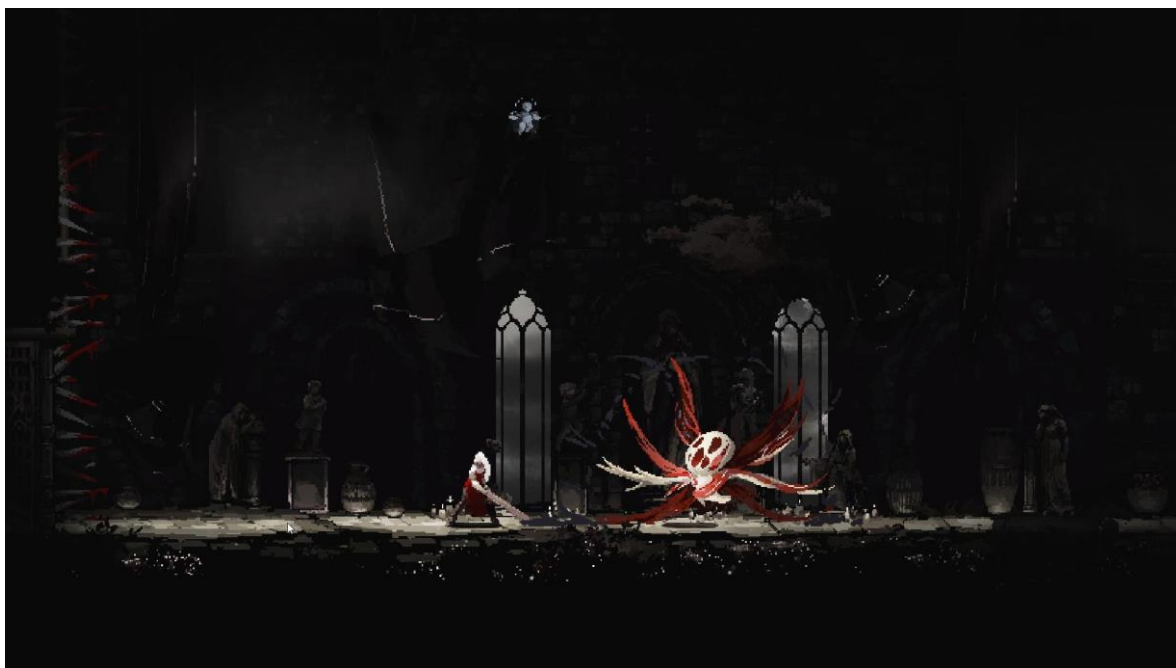


Рисунок 1.6 - Головна героїня гри Moonscars

Сильні сторони:

- Жорстокий та захоплюючий геймплей. Moonscars пропонує жорстокі та складні бої, які змусять вас відчувати себе на межі своїх сил.
- Атмосферний та мальовничий світ. Грандмар - це темний та жорстокий світ, який наповнений цікавими та небезпечними місцями.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		14

– Глибока та емоційна історія. Moonscars розповідає історію про втрату, самопожертву та пошук сенсу життя.

– Унікальний візуальний стиль. Moonscars має мальовничий 2D-арт, який поєднує в собі елементи фентезі та готики.

Недоліки:

– Висока складність. Moonscars - це дуже складна гра, яка може здатися не всім.

– Відсутність інновацій. Moonscars багато в чому схожа на інші ігри жанру Soulslike, і не пропонує багато нового.

– Деякі технічні проблеми. Деякі гравці повідомляли про технічні проблеми, такі як збої та вильоти.

Унікальні кор-механіки у Metroidvania та Soulslike

1. The Last Faith:

– Процедурно генерується світ. Кожне проходження генерується випадково, що робить гру свіжою та непередбачуваною. Ви ніколи не знаєте, що вас чекає за наступним поворотом.

– Система перків. Ви можете знаходити та використовувати перки, щоб покращити свої характеристики та надати собі нові здібності. Це дає вам більше можливостей для персоналізації вашого персонажа.

– Бойова система з акцентом на позиціонування. У боях важливо правильно позиціонувати себе, щоб ухилятися від атак та завдавати ударів з максимальним ефектом

2. Blasphemous 2:

– Три різні види зброї. Меч, кинджал та спеллс, кожен з яких має свої сильні та слабкі сторони. Ви можете перемикатися між ними під час бою, щоб створювати комбінації та адаптуватися до різних ситуацій.

– Унікальні здібності. Завдяки системі молитов ви можете розблокувати різні здібності, такі як телепортація, парирування та удари з більшою силою. Ці здібності роблять бої більш динамічними та різноманітними.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		15

— Здібності для попадання у наступні локації. Деякі здібності, такі як подвійний стрибок та плавання, дозволяють вам досліджувати нові області, які раніше були недоступні. Це додає новий рівень глибини дослідженню світу.

3. Moonscars:

— Система ризику та винагороди. Ви можете використовувати свій "Glimpse" (блиск), щоб уповільнити час під час бою. Це дає вам можливість завдати більше шкоди або ухилитися від атак, але за це ви втрачаєте частину свого здоров'я.

— Система крафту. Ви можете збирати ресурси та використовувати їх для створення нової зброї, броні та предметів. Це дає вам більше контролю над своїм персонажем та його характеристиками.

— Трансформації. У міру проходження гри ви можете розблокувати нові форми, які дають вам нові здібності та можливості. Це робить геймплей більш різноманітним та цікавим.

Проаналізовані 2D-ігри жанрів Metroidvania та Soulslike пропонують жорстокий та атмосферний геймплей, який сподобається шанувальникам складних і захоплюючих ігор. Усі проаналізовані ігри мають свої унікальні ігрові механіки, які роблять їх свіжими та оригінальними, а глибокі історії та мальовничі світи тримають гравців у напрузі протягом багатьох годин.

1.3 Концепт-документ

Назва проекту: «Dethrone»

Унікальна пропозиція: Надати користувачу можливість отримання власного унікального ігрового досвіду, досліджуючи та вступаючи в різноманітні битви у світі гри. Створення виклику для гравців різного рівня. Занурення в атмосферу темного-фентезі. Глибоке історія світу гри, унікальна система геймплею, експериментальний підхід до жанру Metroidvania.

Жанр і цільова аудиторія:

Жанр-Metroidvania & Soulslike.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		16

Цільова аудиторія: Фанати таких ігрових жанрів як Metroidvania, Soulslike. Підлітки (18 +) і старше. Також ті хто прагне виклику і занурення в історію.

Платформа: PC

Сюжет: У світі, охопленому вічною війною, де життя не має ціни, існує легенда про Ловця душ. Це могутній воїн, який володіє мільйонами душ загиблих ворогів. Кожна душа - це не просто трофей, а ключ до неймовірної сили. Ви граєте за Ловця душ, який збирається розгадати таємницю свого походження та зупинити жорстокий конфлікт, що роздирає світ.

Геймплей:

– Збирайте душі. Протягом історії ви будете битися з різними ворогами, від звичайних солдатів до могутніх босів. Перемагаючи їх, ви будете збирати їхні душі, додаючи їх до своєї колекції.

– Прокачка душ. Не всі душі однакові. Існують вороги з найсильнішими душами, якщо ви їх отримаєте, вони стануть ключем до вашої сили. Вивчайте їхні здібності та використовуйте їх, щоб покращити свої бойові навички, отримати доступ до нових атак та здібностей.

– Механіка безсмертя. Завдяки мільйонам душ у вашій колекції ви володієте практично безсмертям. Коли ви гинете в бою, одна з звичайних душ з вашої колекції згорає, а ви воскресаете біля останнього місця збереження.

– Різноманіття атак та здібностей. Використовуйте силу душ, щоб вивчити унікальні атаки та здібності, засновані на бойовому стилі та особистості кожного з ваших переможених ворогів. Це дозволяє створювати персоналізований стиль гри та підбирати стратегії під різних ворогів.

– Розгадуйте таємниці. Досліджуйте світ, шукайте підказки та розкрийте таємницю свого походження. Дізнайтеся про Ловців душ з минулого, про силу душ та про те, яку роль ви граєте в цій вічній війні.

1.4 Аналіз та вимоги до програмного забезпечення

Вибір ігрового рушія

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		17

У процесі вибору інструментальних засобів для розробки ігрового проекту було звернуто увагу на ряд ключових аспектів, які визначили вибір ігрового рушія Unity (рис. 1.7).



Рисунок 1.7 - Unity

1. Мова програмування C#. Один з найбільш суттєвих аргументів у виборі Unity [8, 9] - це можливість використання мови програмування C#, з якою я маю значний досвід і відмінно володію. Це забезпечує швидку і ефективну розробку, зменшуючи час, необхідний для вивчення нових мов або підходів.

2. Зручність використання. Unity відомий своєю дружньою та інтуїтивно зрозумілою інтерфейсною системою, що полегшує процес розробки і дозволяє зосередитися на творчому процесі, не втрачаючи часу на вирішення технічних проблем.

3. Доступність літератури та ресурсів. Unity є одним з найпопулярніших ігрових рушіїв у світі, тому до нього доступна велика кількість літератури, онлайн-курсів, відеоуроків та форумів, де можна знайти відповіді на багато питань та проблем.

4. Підтримка 2D розробки. Unity має потужні засоби для розробки як 3D, так і 2D ігор. З огляду на те, що метою проекту є створення 2D інді гри, Unity ідеально підходить для досягнення цієї мети.

Враховуючи вищезазначені переваги, вибір Unity являє собою обґрунтований крок для успішної реалізації поставленої задачі з розробки ігрового проекту.

Вибір інтегрованого середовища розробки

Для написання коду було обрано Visual Studio (рис. 1.8) з наступних причин:

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		18



Рисунок 1.8 - Visual Studio

1. Інтеграція з Unity. Visual Studio [10] надає максимальну зручність у роботі з Unity завдяки глибокій інтеграції з ігровим рушієм, що полегшує відлагодження, автоматичне доповнення коду та інші корисні функції.

2. Підтримка мови програмування C#. Visual Studio є одним із найпопулярніших інтегрованих середовищ розробки для мови програмування C#. Враховуючи, що Unity використовує C# для розробки, Visual Studio є ідеальним вибором для написання коду.

3. Розширені можливості. Visual Studio пропонує розширені можливості редагування коду, підтримку Git та інші інструменти, що сприяють зручній та продуктивній розробці ігрового проекту.

Об'єднуючи Unity та Visual Studio, ми отримуємо потужну комбінацію інструментів, що допоможе ефективно втілити поставлені цілі в реальність та забезпечити успішну реалізацію проекту.

Висновки до першого розділу

Було визначено мету розробки та поставлено завдання. Було успішно проаналізовано популярні ігри у схожих жанрах, їхні сильні сторони та недоліки. Було описано механіки для подальшої реалізації.

Створено концепт документ, у якому описано унікальні особливості гри, жанр та цільову аудиторію.

Було обґрунтовано вибір ігрового рушія, а саме Unity та Visual Studio, як середовища для програмування.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		19

РОЗДІЛ 2. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Розробка архітектури механік в ігровому додатку

Діаграма варіантів використання [11] (Use Case Diagram) (рис. 2.1) зображує різні види взаємодії між гравцем (Player) та ігровим додатком. У центрі знаходиться сутність «Гравець», від якої розгалужуються декілька варіантів використання, позначених овалами.

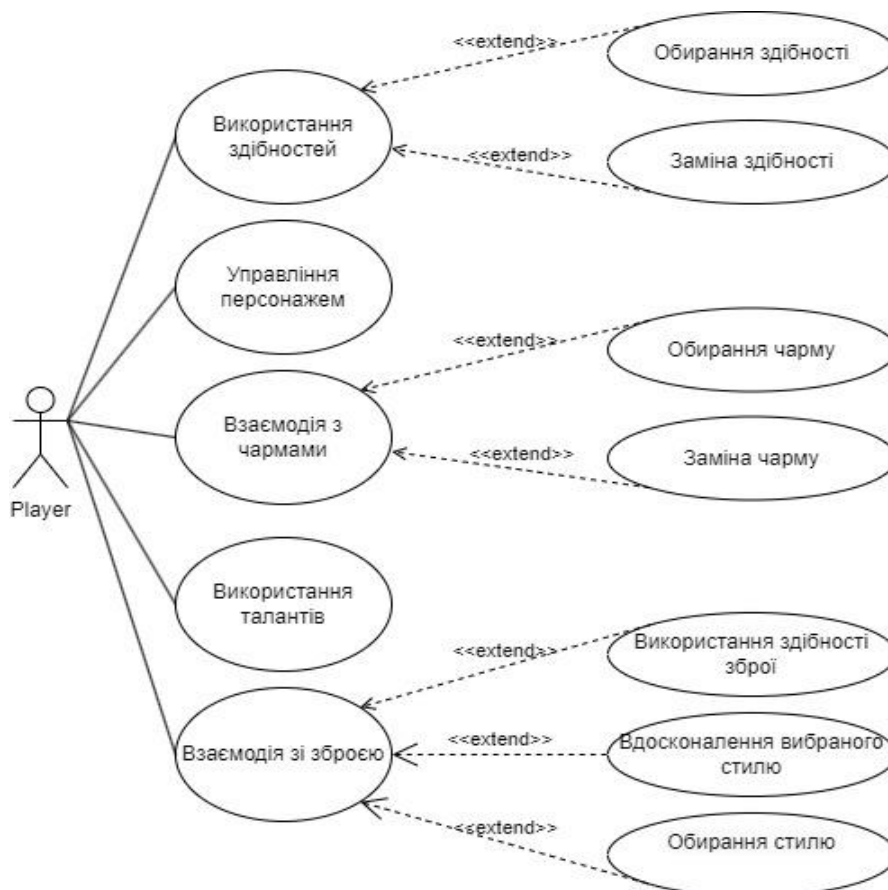


Рисунок 2.1 - Діаграма використання

Прецендент "Використання здібностей" вказує на те, що гравець може вибирати та змінювати доступні йому здібності, «Управління персонажем» передбачає можливість гравця вибирати або змінювати чарм свого персонажа, «Взаємодія з чармами» дозволяє гравцеві змінювати чарм під час гри, «Використання талантів» свідчить про можливість застосовувати спеціальні здібності, пов'язані із зброєю, «Взаємодія зі зброєю» вказує на те, що гравець може обирати певний стиль зброї та вибирати його для використання у грі. Загалом, діаграма описує основні способи взаємодії гравця з ігровим

середовищем, такі як вибір та використання здібностей, талантів, чармів, зброї та стилів.

Діаграма активності [12] зображує послідовність можливих дій та рішень у грі. Вона починається з початкового стану "Почати гру" і далі розгалужується на різні варіанти розвитку подій (рис. 2.2).

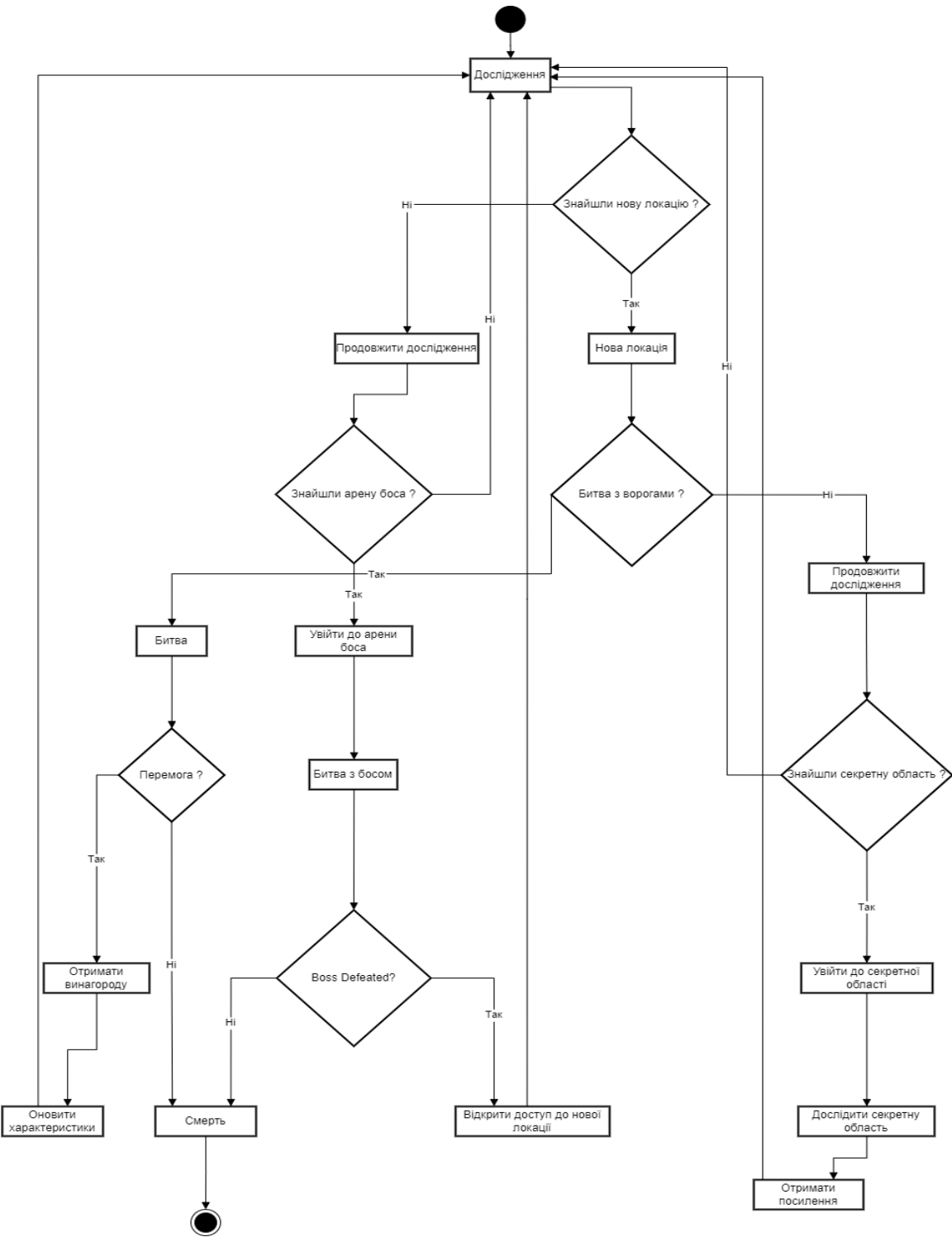


Рисунок 2.2 - Діаграма активності

Діаграма класів (рис. 2.3) відображає структуру класів у грі з елементами рольового бою.

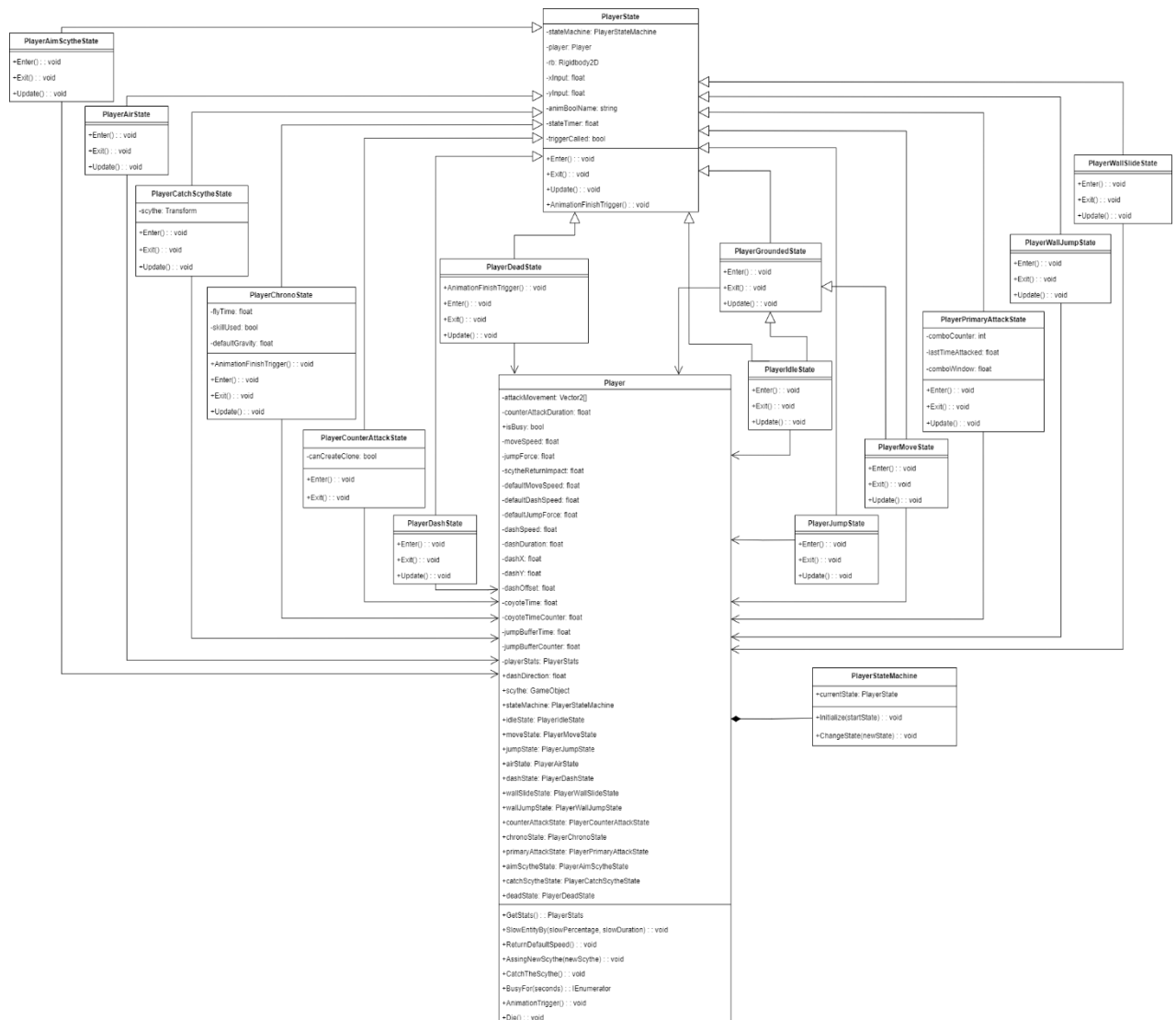


Рисунок 2.3 - Діаграма класів

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				22
Змн.	Арк.	№ докум.	Підпис	Дата		

Центральним класом є Player, який має численні стани, такі як PlayerIdleState, PlayerMoveState, PlayerJumpState тощо. Ці стани успадковуються від базового класу PlayerState. Клас Player також має посилення на об'єкти класів PlayerStateMachine та PlayerStats. Останній клас відповідає за управління здоров'ям, манною та ультою гравця, а також успадковується від абстрактного класу CharacterStats.

Діаграма також показує відношення між класами, такі як композиція (Player має PlayerStateMachine), агрегація (PlayerStats має посилення на Player та BarsController) та успадкування (PlayerStats успадковується від CharacterStats). Примітки на діаграмі пояснюють призначення деяких методів у класі PlayerStats.

Ця діаграма класів надає візуальне уявлення про основні класи та їх відношення у грі, що допомагає зрозуміти загальну структуру програмного забезпечення.

Діаграма компонентів (рис. 2.4) показує [14] різні модулі та їхні взаємозв'язки в рамках ігрового додатку "Dethrone". У центрі діаграми розташований компонент "Геймдизайн гри "Dethrone"", який є головним компонентом гри.

Цей головний компонент пов'язаний з такими компонентами:

1. "Сюжет гри" - компонент, що відповідає за сюжетну лінію та історію гри.
2. "Модуль чармів" - компонент, що відповідає за систему чармів (додаткових предметів або навичок для персонажа).
3. "Модуль здібностей" - компонент, що керує здібностями персонажа.
4. "Модуль стилів зброї" - компонент, що відповідає за різні стилі зброї, які може використовувати персонаж.
5. "Модуль керування персонажем" - компонент, що відповідає за управління персонажем, його анімації та стани.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		23

6. "Модуль сцен та локацій" - компонент, що відповідає за сцени та локації в грі.

Компонент "Модуль керування персонажем" має підкомпоненти "Налаштування анімацій" та "Налаштування станів персонажа", які безпосередньо пов'язані з ним.

Діаграма показує взаємозв'язки між різними компонентами, які складають функціональність ігрового додатку "Dethrone".

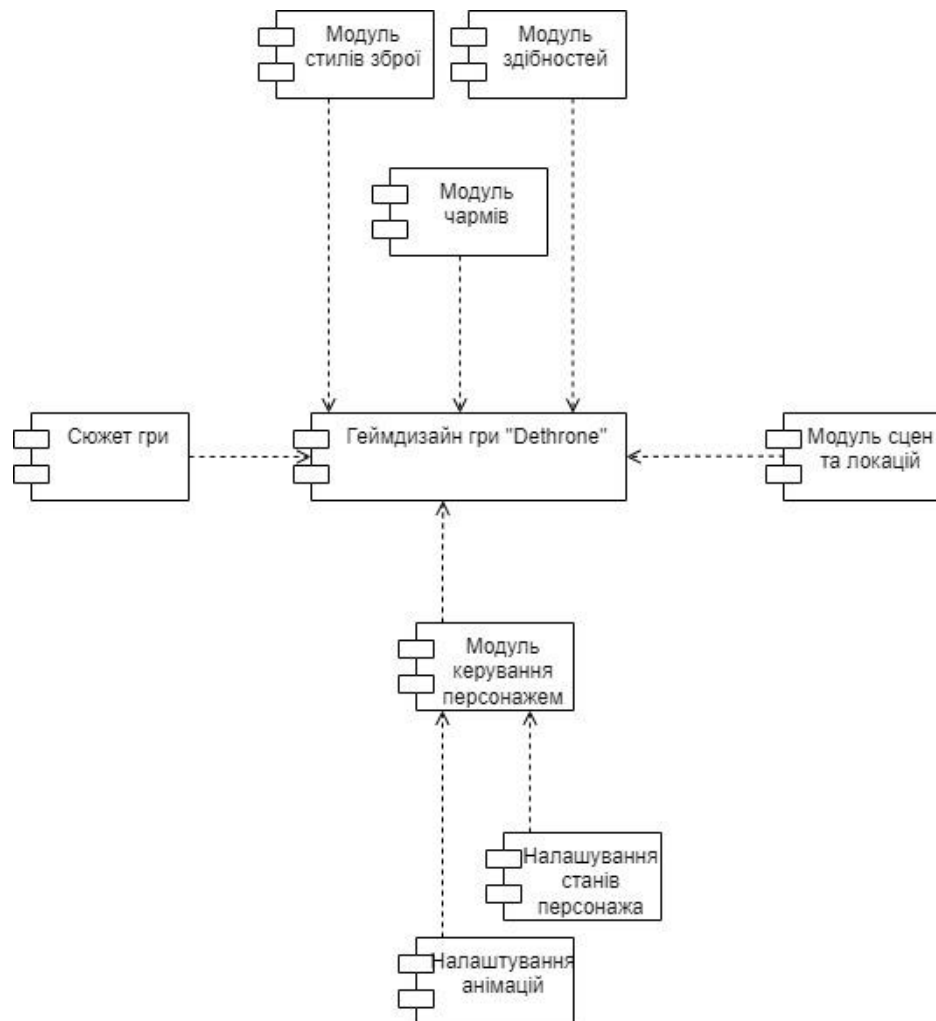


Рисунок 2.4 - Діаграма компонентів

Діаграма розгортання (рис. 2.5) демонструє [15], як різні компоненти гри розгортаються на різних вузлах (пристроях).

Центральним вузлом є "Engine", який представляє ігровий рушій Unity. На цьому вузлі розгортається компонент Unity, який є ядром ігрового рушія.

Інший вузол - "Developer's PC" (комп'ютер розробника). На ньому розгортаються такі компоненти:

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		24

- "Game Assets" (ігрові ресурси, такі як моделі, текстури, звуки тощо)
- "Source control" (система контролю версій вихідного коду)

Вузол "Player's PC" (комп'ютер гравця) є кінцевим пунктом, де фактично запускається і працює гра. На ньому розгортаються:

- "Game Executable" (виконуваний файл гри)
- "Save data" (дані збереження гри)

Діаграма показує зв'язки між різними вузлами під час розробки та виконання гри. Розробник працює на своєму комп'ютері, використовуючи ігровий рушій Unity та ресурси гри. Потім виконуваний файл гри та необхідні дані розгортаються на комп'ютері гравця для запуску та гри.

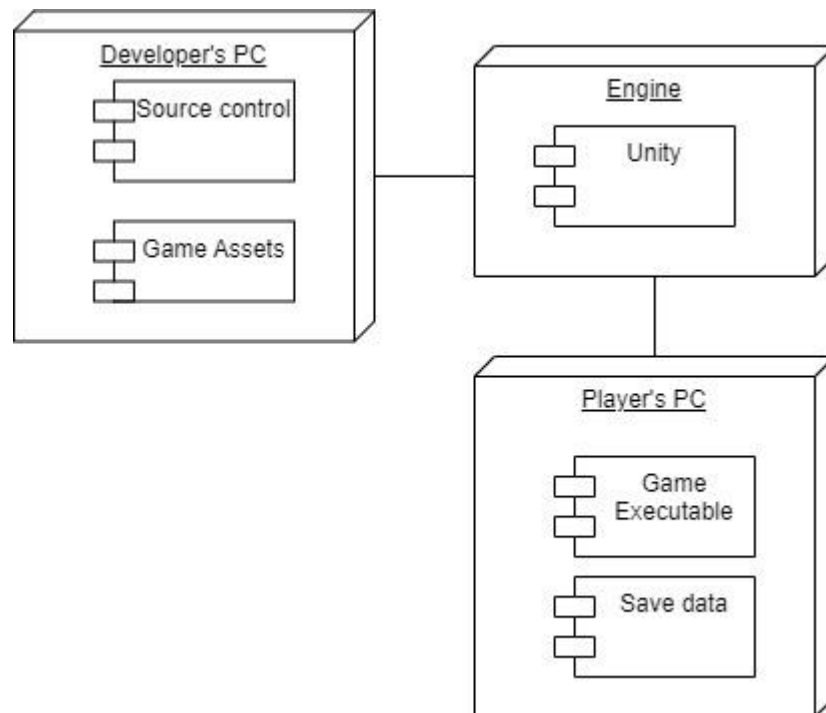


Рисунок 2.5 - Діаграма розгортання

2.2 Розробка головного персонажа

Розробка головного персонажа [16] є однією з ключових складових створення успішної гри в жанрах Metroidvania та Soulslike. Головний персонаж є центральним елементом, з яким взаємодіє гравець, тому його проектування вимагає детального опрацювання як візуальних, так і ігрових аспектів. В даному розділі ми розглянемо процес розробки головного персонажа,

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		25

включаючи налаштування керування, реалізацію state machine, систему характеристик персонажа та налаштування аніматора.

Першим кроком у розробці головного персонажа є визначення його основних характеристик та можливостей. Враховуючи унікальні особливості жанрів Metroidvania та Soulslike, головний персонаж повинен володіти широким набором рухів, а також можливістю покращення своїх характеристик протягом гри.

Наступним кроком є розробка системи керування, яка забезпечить плавний і інтуїтивно зрозумілий геймплей. Важливим аспектом є створення state machine, яка дозволяє легко управляти різними станами персонажа, такими як біг, стрибок, атака, блокування та інші. Кожен стейт повинен бути чітко визначений і мати свої умови переходу, що забезпечить плавний і логічний ігровий процес.

Таким чином, у цьому розділі ми детально розглянемо кожен з цих аспектів, що дозволить створити добре продуманого та цікавого головного персонажа для гри в жанрах Metroidvania та Soulslike.

Характеристики персонажа

У розробці головного персонажа для гри в жанрах Metroidvania та Soulslike важливо враховувати різноманітні характеристики, які визначають його можливості та взаємодію з ігровим світом. Для управління цими характеристиками використовується спеціальний клас *Stat.cs*, який ми також детально розглянемо.

Клас *Stat* відповідає за визначення та управління значеннями характеристик персонажа.

```
[System.Serializable]
public class Stat
{
    // Базове значення характеристики
    [SerializeField] private float baseValue;

    // Список модифікаторів, які додаються до базового
    значення
    public List<float> modifiers;
```

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		26

```

        // Метод отримання поточного значення характеристики з
        // урахуванням всіх модифікаторів
        public float GetValue()
        {
            float finalValue = baseValue;

            // Додаємо всі модифікатори до базового значення
            foreach (float modifier in modifiers)
            {
                finalValue += modifier;
            }

            return finalValue;
        }

        // Метод встановлення базового значення характеристики
        public void SetDefaultValue(float _value)
        {
            baseValue = _value;
        }

        // Метод додавання модифікатора до характеристики
        public void AddModifier(float _modifier)
        {
            modifiers.Add(_modifier);
        }

        // Метод видалення модифікатора з характеристики
        public void RemoveModifier(float _modifier)
        {
            modifiers.Remove(_modifier);
        }

        // Метод видалення всіх модифікаторів з характеристики
        public void RemoveAllModifiers()
        {
            modifiers.Clear();
        }
    }

```

Пояснення коду:

- Базове значення (baseValue): Це початкове значення характеристики без урахування додаткових бонусів чи модифікаторів.
- Модифікатори (modifiers): Це список додаткових значень, які додаються до базового значення характеристики. Модифікатори можуть змінюватися під впливом різних факторів гри, таких як предмети, бафи або дебафи.
- Метод GetValue(): Цей метод повертає поточне значення характеристики з урахуванням всіх модифікаторів.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		27

- Метод `SetDefaultValue(float _value)`: Цей метод дозволяє встановити нове базове значення характеристики.
- Метод `AddModifier(float _modifier)`: Цей метод додає новий модифікатор до списку.
- Метод `RemoveModifier(float _modifier)`: Цей метод видаляє модифікатор зі списку.
- Метод `RemoveAllModifiers()`: Цей метод видаляє всі модифікатори зі списку, повертаючи характеристику до її базового значення.

Клас `Stat` забезпечує гнучкість та зручність в управлінні характеристиками персонажа, дозволяючи легко додавати, видаляти та змінювати їх значення в залежності від ситуації в грі.

У нашій грі ми використовуємо систему характеристик для управління атрибутами персонажів. Клас `CharacterStats` відповідає за основні характеристики та їхнє керування. Цей клас визначає різноманітні параметри персонажа, такі як атака, захист, мана та інші. Давайте розглянемо основні категорії характеристик та їхнє призначення.

1. Основні характеристики (Major stats)

Основні характеристики визначають базові здібності персонажа, впливаючи на його ефективність у бою та виживання. До них належать:

- Сила (`strength`): Збільшує фізичну шкоду та силу критичних ударів.
- Спритність (`agility`): Збільшує шанс ухилення від атак та шанс критичних ударів.
- Інтелект (`intelligence`): Збільшує магічну шкоду та магічний опір.
- Вітальність (`vitality`): Збільшує максимальний рівень здоров'я.

2. Атакуючі характеристики (Offensive stats)

Атакуючі характеристики визначають силу атак персонажа та його здатність завдавати критичні удари:

- Шкода (`damage`): Базова величина шкоди, яку завдає персонаж.
- Шанс критичного удару (`critChance`): Визначає ймовірність завдати критичний удар.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		28

- Сила критичного удару (critPower): Визначає множник шкоди при критичному ударі.

- Магічне посилення (magicAmplify): Збільшує ефективність магічних атак (початкове значення 150%).

Метод DoDamage відповідає за завдання шкоди цільовому персонажу, враховуючи різні фактори, такі як шанс ухилення, критичні удари та броня цілі.

```
public virtual void DoDamage(CharacterStats _targetStats)
{
    if (TargetCanAvoidAttack(_targetStats))
        return;

    float totalDamage = damage.GetValue() +
strength.GetValue();

    if (CanCrit())
    {
        totalDamage = CalculateCriticalDamage(totalDamage);
    }

    totalDamage = CheckTargetArmor(_targetStats,
totalDamage);
    _targetStats.TakeDamage(totalDamage);
    onDamageDealt?.Invoke(totalDamage);
}
```

У класі PlayerStats цей метод перевизначено для додавання додаткових ефектів, таких як збільшення мани та ультимативної енергії після завдання шкоди.

```
public override void DoDamage(CharacterStats _targetStats)
{
    base.DoDamage(_targetStats);
    IncreaseMana(5);
    IncreaseUlt(10);
    allBars.SetMana(currentMana, maxMana.GetValue());
    allBars.SetUlt(currentUlt, maxUlt.GetValue());
}
```

Метод CanCrit відповідає за перевірку нанесення критичної шкоди по ворогу.

```
private bool CanCrit()
{
    float totalCriticalChance = critChance.GetValue() +
agility.GetValue();

    if (Random.Range(0, 100) <= totalCriticalChance)
    {
        return true;
    }
}
```

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		29

```

    }
    return false;
}

```

В методі CalculateCriticalDamage ми обраховуємо вихідну критичну шкоду.

```

private float CalculateCriticalDamage(float _damage)
{
    float totalCritPower = (critPower.GetValue() +
    strength.GetValue()) * .01f;
    float critDamage = _damage * totalCritPower;

    return Mathf.RoundToInt(critDamage);
}

```

3. Захисні характеристики (Defensive stats)

Захисні характеристики забезпечують виживання персонажа, знижуючи отриману шкоду та підвищуючи здатність уникати атак:

- Максимальне здоров'я (maxHealth): Визначає максимальну кількість здоров'я персонажа.
- Броня (armor): Зменшує шкоду, яку отримує персонаж від фізичних атак.
- Ухилення (evasion): Визначає ймовірність уникнути атаки.

Метод TakeDamage обробляє отримання шкоди персонажем, враховуючи можливі зниження шкоди та виклики відповідних подій. Він також відповідає за візуальні ефекти, пов'язані з отриманням шкоди, та перевіряє, чи персонаж ще живий після отримання шкоди.

```

public virtual void TakeDamage(float _damage)
{
    float reducedDamage = _damage * (1f -
    damageReceivedReductionPercentage / 100f);
    DecreaseHealthBy(reducedDamage);

    GetComponent<Entity>().DamageImpact();
    fx.StartCoroutine("FlashFX");

    if (currentHealth < 0 && !isDead)
        Die();
    OnDamageReceived?.Invoke(_damage);
}

```

Коли говориться про "події" в Unity, мається на увазі патерн програмування, відомий як "Спостерігач" (Observer Pattern) [17]. Цей патерн

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		30

дозволяє одному об'єкту (видавцеві) повідомляти інші об'єкти (передплатників) про зміни свого стану. У випадку з методом TakeDamage, подія OnDamageReceived дозволяє іншим об'єктам передплатитися на отримання повідомлення про те, що персонаж отримав шкоду. Це дозволяє динамічно реагувати на події в грі, наприклад, оновлювати інтерфейс користувача або запускати додаткові ефекти.

```
public event System.Action<float> OnDamageReceived;
```

У класі PlayerStats цей метод перевизначено для додаткової перевірки на невразливість під час ривку та оновлення інтерфейсу користувача.

```
public override void TakeDamage(float _damage)
{
    base.TakeDamage(_damage);
    if (isInvincible)
    {
        return;
    }
    allBars.SetHealth(currentHealth - _damage,
        GetMaxHealthValue());
}
```

4. Характеристики мани (Mana stats)

Характеристики мани визначають здатність персонажа використовувати магічні здібності:

- Максимальна мана (maxMana): Визначає максимальний рівень мани персонажа.
- Швидкість регенерації мани (manaRegenRate): Визначає швидкість відновлення мани.

Методи роботи з манною обробляють регенерацію, збільшення та зменшення мани. У класі PlayerStats вони перевизначені для оновлення інтерфейсу користувача.

```
public virtual void RegenerateMana()
{
    currentMana += manaRegenRate.GetValue() * Time.deltaTime;
    currentMana = Mathf.Clamp(currentMana, 0f,
        maxMana.GetValue());
}

public override void RegenerateMana()
```

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		31

```
{
    base.RegenerateMana();
    allBars.SetMana(currentMana, maxMana.GetValue());
}
```

5. Характеристики ультимативної здібності (Ult stats)

Характеристики ультимативної здібності визначають можливість використання особливих потужних здібностей. Максимальна ультра (maxUlt) визначає максимальний рівень ультимативної енергії. Ці характеристики та їх зміни безпосередньо впливають на ігровий процес, роблячи його більш динамічним та цікавим для гравців.

Керування персонажем, патерн State Machine

Керування персонажем у 2D іграх є важливою складовою, що впливає на ігровий процес та загальне враження від гри. Від точності і чутливості управління залежить, наскільки комфортно гравець може виконувати дії та маневри в ігровому середовищі. Основні елементи керування в 2D іграх включають:

- Рух. Переміщення персонажа по горизонталі (вліво/вправо) та вертикалі (вгору/вниз), що може бути реалізовано за допомогою клавіатури, геймпада або сенсорного управління.
- Стрибки. Додавання можливості стрибати, що включає розрахунок висоти стрибка, гравітації та приземлення.
- Атаки. Взаємодія з ворогами або об'єктами через атаки, що можуть бути ближніми або дистанційними.
- Спеціальні дії. Використання здібностей або скілів, що можуть бути унікальними для кожного персонажа.

Патерн State Machine (машина станів) — це архітектурний шаблон, який дозволяє моделювати поведінку об'єктів на основі їх поточного стану. Кожен стан відповідає за певну поведінку об'єкта, а переходи між станами здійснюються на основі певних умов або подій.

Переваги використання State Machine в іграх:

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		32

1. Модульність: Легке додавання або зміна станів без впливу на інші частини коду.
2. Зрозумілість: Чітка структура, яка робить код легшим для розуміння та підтримки.
3. Гнучкість: Можливість легко керувати складною поведінкою персонажів та об'єктів.
4. Тестованість: Легше тестування окремих станів та переходів між ними.

У грі існує багато унікальних станів персонажа, далі переглянемо кожен з них:

1. Idle.

Стан `PlayerIdleState` відповідає за бездіяльний стан персонажа, коли той не рухається та не виконує жодних дій. Його головна функція - перехід до стану руху, якщо гравець натисне відповідні клавіші для руху. Також, він нулює швидкість персонажа при вході у стан.

2. Move.

Стан `PlayerMoveState` відповідає за рух персонажа в горизонтальному напрямку. Він встановлює швидкість руху персонажа та визначає перехід до стану спокою, якщо гравець припинив натискання кнопок руху.

3. Jump.

Стан `PlayerJumpState` відповідає за стрибок персонажа. При вході в цей стан, персонаж отримує початкову швидкість стрибка вгору. Якщо вертикальна швидкість персонажа стає від'ємною, то він переходить у стан повітря.

4. Aim.

Стан `PlayerAimScytheState` відповідає за направлення кидка сокири персонажа. Гравець може маніпулювати мишкою для визначення напрямку кидка. Перехід до стану спокою відбувається при натисканні відповідної клавіші.

5. Air.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		33

Стан PlayerAirState відповідає за рух персонажа у повітрі. Він визначає перехід до стану спокою при досягненні землі або при наявності стіни. Якщо гравець має вміння стіни, то він може перейти до стану ковзання по стіні.

6. CatchScythe.

Стан PlayerCatchScytheState відповідає за повернення сокири до персонажа після кидка. Перехід до стану спокою відбувається після успішного повернення сокири.

7. Chrono.

Стан PlayerChronoState відповідає за використання спеціального навички "Chrono". Персонаж може короткий час літати у повітрі, а потім повернутися на землю. Перехід до стану повітря відбувається після завершення дії навички.

8. CrimsonCarnage.

Стан PlayerCrimsonCarnageState відповідає за активацію особливого навички "Crimson Carnage", яка наносить кровоточивий удар. Перехід до стану спокою відбувається після успішного завершення удару.

9. Dash.

Стан PlayerDashState відповідає за виконання руху "Dash". Під час дашу персонаж може пересуватися швидше і проходити через ворогів. Перехід до стану спокою відбувається після завершення дашу.

10. Dead.

Стан PlayerDeadState відповідає за смерть персонажа. Перехід до стану спокою відбувається після відновлення життя гравцем.

11. Grounded.

Стан PlayerGroundedState відповідає за поведінку персонажа, коли він знаходиться на землі. Він визначає перехід до інших станів в залежності від введення гравця та його можливостей.

12. Attack.

Стан PlayerPrimaryAttackState відповідає за основну атаку персонажа. Перехід до стану спокою відбувається після завершення атаки або проміжку часу між атаками для формування комбо.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		34

13. WallJump.

відповідає за стрибок від стіни. Перехід до стану повітря відбувається після певного часу або при дотику до землі.

14. WallSlide.

Стан `PlayerWallSlideState` відповідає за ковзання персонажа по стіні. Перехід до стану спокою або повітря відбувається в залежності від дій гравця та його розташування.

2.3 Реалізація здібностей героя та зброї

Розвиток персонажа та його бойові навички здатні значно збагатити геймплей та зробити гру більш цікавою для гравця.

У цьому розділі ми дослідимо, як працюють здібності героя в грі «Dethrone», а також розглянемо реалізацію трьох різних стилів зброї, кожен з яких має унікальні активні та пасивні здібності. Ми розглянемо реалізацію методів для їх активації та використання. Крім того, буде розглянуто підвищення рівня пасивних здібностей та їх вплив на героя в грі.

Стилі зброї

У грі присутні три стилі зброї: червоний, синій та зелений. Кожен стиль має свою унікальну активну здібність та дві пасивні, які можна прокачати.

Клас `WeaponStyle` є базовим для всіх стилів зброї. Він містить віртуальні методи для активації та деактивації ефектів, загальних для всіх стилів зброї.

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

public class WeaponStyle : MonoBehaviour
{
    public virtual void ActivateFirstUpgrade()
    {
    }

    public virtual void ActivateSecondUpgrade()
    {
    }

    public virtual void ActivateThirdUpgrade()
```

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		35

```
{
}
```

```
public virtual void DeactivateEffect()
{
```

Червоний стиль розрахований на гравців, які шукають активні сутички з ворогами, полюбляють завдавати велику шкоду та постійно відновлювати своє здоров'я. Саме тому було розроблено клас RedWeaponStyle, який успадковує клас WeaponStyle та реалізує конкретні методи для активації різних рівнів прокачки червоного стилю зброї.

Розглянемо логіку методів цього класу:

```
public override void ActivateFirstUpgrade()
{
    isBaseActive = true;
}

public override void ActivateSecondUpgrade()
{
    isFirstActive = true;

    playerStats.maxHealth.AddModifier(maxHealthIncrease);
    playerStats.onDamageDealt += ApplyVampirism;
}

public override void ActivateThirdUpgrade()
{
    isSecondActive = true;
}
```

Ці методи відповідають за активацію відповідного рівня прокачки стилю. Оскільки більшість ефектів не можна один раз додати гравцю, то треба постійно перевіряти рівень прокачки стилю та чи користується гравець саме им стилем. Все це зручно робити в методі Update()

```
private void Update()
{
    if (isSecondActive && bleedDuration > 0 &&
    isBleeding)
    {
        bleedDuration -= Time.deltaTime;
        damageTimer += Time.deltaTime;
        if (damageTimer >= 1f / bleedTickPerSecond)
        {
            damageTimer = 0f;
            crimsonCarnage.BleedEffect(bleedingDamage);
        }
    }
}
```

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		36

```

        if(bleedDuration == 0)
        {
            isBleeding = false;
        }
    }
    if (Input.GetKeyUp(KeyCode.F) && isBaseActive &&
playerStats.currentUlt == playerStats.maxUlt.GetValue())
    {
        bleedDuration = originalBleedDuration;
        crimsonCarnage.ActivateCrimsonCarnage();
        isBleeding = true;
    }
}

```

Для реалізації ефекта вампіризму від атак було використано патерн «спостерігач» реалізуємо цей ефект при досягненні другого рівня прокачки в червоному стилі. Тепер при нанесенні шкоди гравець буде отримувати ефект вампіризму.

```

private void ApplyVampirism(float damageDealt)
{
    float healAmount = damageDealt *
(vampirismPercentage / 100f);
    playerStats.IncreaseHealthBy(healAmount);

playerStats.allBars.SetHealth(playerStats.currentHealth,
playerStats.maxHealth.GetValue());
}
}

```

У кожного стилю є своя активна здібність у червоного - "Crimson Carnage". Також існує однойменний клас, який відповідає за активацію спеціальної атаки.

```

public void ActivateCrimsonCarnage()
{

playerStats.DecreaseUlt(playerStats.maxUlt.GetValue());
    Collider2D[] colliders =
Physics2D.OverlapCircleAll(transform.position, damageRadius);
    foreach (Collider2D collider in colliders)
    {
        if (collider.CompareTag("Enemy"))
        {
            EnemyStats enemyStats =
collider.GetComponent<EnemyStats>();
            if (enemyStats != null)
            {
                enemyStats.TakeDamage(damage + damage *
playerStats.magicAmplify.GetValue() / 100);

```

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		37

```

    }
    }
}

```

Метод `ActivateCrimsonCarnage()` виконує розподіл шкоди по всім ворогам, які знаходяться у визначеному радіусі навколо героя.

```

public void BleedEffect(float bleedDamage)
{
    Collider2D[] colliders =
Physics2D.OverlapCircleAll(transform.position, damageRadius);
    foreach (Collider2D collider in colliders)
    {
        if (collider.CompareTag("Enemy"))
        {
            EnemyStats enemyStats =
collider.GetComponent<EnemyStats>();
            if (enemyStats != null)
            {
                enemyStats.TakeDamage(bleedDamage);
            }
        }
    }
}

```

Метод `BleedEffect()` виконує нанесення ефекту кровотечі ворогам, він стане активним, коли досягне останнього рівня прокачки червоного стилю.

Коли гравець захоче змінити стиль, ми повинні деактивувати усі ефекти поточного стилю, для цього існує метод `DeactivateEffect()`.

```

public override void DeactivateEffect()
{
    base.DeactivateEffect();
    if (isFirstActive)
    {
        playerStats.maxHealth.RemoveModifier(maxHealthIncrease);
        playerStats.onDamageDealt -= ApplyVampirism;
    }

    isBaseActive = false;
    isFirstActive = false;
    isSecondActive = false;
}

```

Наступний стиль-зелений. Цей стиль створено для гравців, які прагнуть швидкого геймплею та взаємодії зі своїми копіями. Клас `GreenWeaponStyle` відповідає за реалізацію зеленого стилю зброї. Він також успадковує базовий клас `WeaponStyle` та реалізує методи для кожного рівня прокачки:

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		38

ActivateFirstUpgrade() активує перший рівень прокачки стилю та надає унікальну здібність «Miracle Mirage» яка дозволяє створювати копію себе на певний проміжок часу. Копія повторює усі дії оригіналу та наносить шкоду ворогам.

```
public override void ActivateFirstUpgrade()
{
    isBaseActive = true;
}
```

Для реалізації активної здібності зеленого стилю було написано скрипт MiracleMirage у якому реалізовано метод для створення копії персонажа.

```
public void ActivateMirage()
{
    playerStats.DecreaseUlt(playerStats.maxUlt.GetValue());
    if (mirageInstance == null)
    {
        mirageInstance = Instantiate(miragePrefab,
        transform.position, Quaternion.identity);
        Destroy(mirageInstance, mirageDuration);
    }
}
```

Метод ActivateSecondUpgrade() викликається для активації другого рівня прокачки, він збільшує швидкість руху гравця та зменшує час відновлення ривка.

```
public override void ActivateSecondUpgrade()
{
    isFirstActive = true;
    player.moveSpeed += player.moveSpeed *
    moveSpeedIncreasePercent/100;

    SkillManager.instance.dash.ReduceCooldown(reduceCooldown);
}
```

Метод ActivateThirdUpgrade() активує третій рівень прокачки зеленого стилю зброї, він дозволяє гравцеві створювати свого клона на старті ривка.

```
public override void ActivateThirdUpgrade()
{
    isSecondActive = true;

    SkillManager.instance.clone.CanCreateCloneOnStart(true);
}
```

Деактивація відбувається ідентично до усіх стилів.

Останній стиль-синій. Стиль дозволяє гравцеві поліпшити свої магічні здібності та використовувати ману та інтелект набагато частіше, ніж це роблять гравці з іншим типом геймплею.

Клас відповідає BlueWeaponStyle за реалізацію синього стилю зброї. Він успадковує базовий клас WeaponStyle та реалізує методи для кожного рівня прокачки:

ActivateFirstUpgrade() активує перший рівень прокачки стилю та надає унікальну здібність «Celestial Nexus» активація якої створює поруч з героєм магічну сферу, яка постійно лікує гравця або ж наносить шкоду ворогам, які знаходяться в радіусі дії здібності. Сферу можна рухати, якщо вдарити по ній звичайною атакою.

```
public override void ActivateFirstUpgrade()
{
    isBaseActive = true;
}
```

Для реалізації активної здібності синього стилю було написано скрипт CelestialNexus у якому реалізовано методи для перевірки колізій та переміщення магічної сфери.

```
private void CheckCollisions(){
    Collider2D[] colliders =
Physics2D.OverlapCircleAll(transform.position, 2f);

    foreach (Collider2D collider in colliders){
        if (collider.CompareTag("Player"))
        {
            playerStats.IncreaseHealthBy(healthRestore);
        }
        else if (collider.CompareTag("Enemy")){
            EnemyStats enemyStats =
collider.GetComponent<EnemyStats>();
            if (enemyStats != null){

enemyStats.TakeDamageWithoutKnockback(damage + damage *
playerStats.magicAmplify.GetValue()/100);
            }
        }
    }
}

public void MoveNexus(){
    Player player = PlayerManager.instance.player;
    float attackRadius =
playerStats.GetComponent<Entity>().GetAttackCheckRadius();
```

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		40

```

        if (player.stateMachine.currentState ==
player.primaryAttackState &&
            Physics2D.OverlapCircle(transform.position,
attackRadius, LayerMask.GetMask("Player"))) {
            Vector2 playerDirection =
player.transform.right;
            float distance =
Vector2.Distance(player.transform.position, transform.position);

            float adjustedForce = moveForce / distance;

            rb.AddForce(playerDirection.normalized *
adjustedForce, ForceMode2D.Impulse);
            Debug.Log("A DA");
        }
    }
}

```

Створення сфери здійснюється завдяки методу `ActivateCelestialNexus()`.

```

public void ActivateCelestialNexus() {
    if (celestialNexusPrefab != null) {

playerStats.DecreaseUlt(playerStats.maxUlt.GetValue());
        PlayerManager playerManager =
PlayerManager.instance;
        if (playerManager != null &&
playerManager.player != null) {
            celestialNexusInstance =
Instantiate(celestialNexusPrefab,
playerManager.player.transform.position +
playerManager.player.transform.right * 2f, Quaternion.identity);
            Destroy(celestialNexusInstance,
timeToDestroy);
        }
        else {
            Debug.LogError("PlayerManager or player
reference is missing!");
        }
    }
    else {
        Debug.LogError("Celestial Nexus prefab is not
assigned!");
    }
}
}

```

Метод `ActivateSecondUpgrade()` викликається для активації другого рівня прокачки, він дозволяє з часом повністю регенерувати ману гравця та збільшує її максимальне значення.

```

public override void ActivateSecondUpgrade()
{

```

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		41

```

        isFirstActive = true;

        playerStats.maxMana.AddModifier(maxManaIncrease);
playerStats.manaRegenRate.AddModifier(manaRegenerationRate);

        playerStats.allBars.SetMana(playerStats.currentMana,
playerStats.maxMana.GetValue());
    }

```

Третій рівень прокачки синього стилю надає унікальну пасивну здібність, яка з плином часу створює біля гравця маленькі магичні сфери, кожна з яких посилює шкоду від активної здібності «Celestial Nexus».

Створено корутину AddSphere(), яка додає сферу кожні декілька секунд.

```

private IEnumerator AddSphere()
{
    if (currentOrbCount < maxOrbCount)
    {
        isAddingSphere = true;
        yield return new
WaitForSeconds(timeToSpawnSphere);
        currentOrbCount++;
        sphereGUI.FillImage(currentOrbCount - 1);

playerStats.magicAmplify.AddModifier(magicDamageIncreasePerOrb);
        isAddingSphere = false;
    }
}

```

Метод RemoveAllSpheres() прибирає усі сфери при отриманні будь якої шкоди.

```

private void RemoveAllSpheres(float remove)
{
    currentOrbCount = 0;
    playerStats.magicAmplify.RemoveAllModifiers();
    for (int i = 0; i < sphereGUI.images.Length; i++)
    {
        sphereGUI.ClearImage(i);
    }
}

```

Здібності гравця

У грі передбачено дві основні здібності, які гравець отримує після перемоги над босами. Кожна з цих здібностей наслідуються від базового класу Skill, що дозволяє стандартизувати їх реалізацію та полегшує їх інтеграцію в гру.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		42

Клас Skill забезпечує основну структуру для всіх здібностей у грі. Він містить основні параметри та методи, які можуть бути використані або перевизначені у похідних класах.

```

public virtual bool CanUseSkill()
{
    if(cooldownTimer < 0)
    {
        UseSkill(); // Використання навички
        cooldownTimer = cooldown; // Встановлення часу
перезарядки
        return true;
    }
    return false;
}

public virtual void UseSkill()
{
    // Реалізація використання навички буде у похідних
класах
}

```

1. Метод CanUseSkill() - перевіряє, чи може навичка бути використана в даний момент. Якщо час перезарядки закінчився, викликає метод UseSkill() і встановлює cooldownTimer на значення cooldown.

2. Метод UseSkill() - метод-заглушка, який буде перевизначений у похідних класах для реалізації конкретних дій навички.

```

protected virtual Transform FindClosestEnemy(Transform
_checkTransform)
{
    Collider2D[] colliders =
Physics2D.OverlapCircleAll(_checkTransform.position, 25);

    float closestDistance = Mathf.Infinity;
    Transform closestEnemy = null;

    foreach (var hit in colliders)
    {
        if (hit.GetComponent<Enemy>() != null)
        {
            float distanceToEnemy =
Vector2.Distance(_checkTransform.position,
hit.transform.position);

            if (distanceToEnemy < closestDistance)
            {
                closestDistance = distanceToEnemy;
            }
        }
    }
}

```

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		43

```

        closestEnemy = hit.transform;
    }
}

return closestEnemy;
}
}

```

Метод FindClosestEnemy(Transform _checkTransform) - знаходить найближчого ворога до переданого трансформу, що дозволяє навичці взаємодіяти з ворогами в межах радіусу.

Клас Skill є основою для створення конкретних здібностей гравця, забезпечуючи базову функціональність і дозволяючи легко додавати нові здібності через успадкування. Далі ми розглянемо конкретні реалізації здібностей, наслідуючи цей базовий клас.

Одна з двох здібностей, яку може використовувати гравець - це хроносфера. Ця здібність дозволяє гравцю створити чорну сферу, яка зупиняє час для ворогів в певному радіусі. Після кількох секунд герой зникає, залишаючи лише сферу, в якій з'являються проекції гравця, що наносять удари ворогам. По закінченню дії сфери, час для ворогів відновлюється.

Клас ChronoSkill наслідується від базового класу Skill і реалізує специфічну логіку для здібності хроносфери.

Основні елементи класу ChronoSkill:

1. Метод Start() - ініціалізує посилання на дані досягнень.
2. Метод CanUseSkill() - перевіряє можливість використання навички, викликаючи метод з базового класу.
3. Метод UseSkill() - реалізує логіку створення хроносфери, викликає метод налаштування для контролера хроносфери та оновлює досягнення.
4. Метод SkillCompleted() - перевіряє, чи завершилася дія навички.
5. Методи SetMaxSize(float _size) і GetMaxSize() - методи для встановлення та отримання максимального розміру хроносфери.

Клас ChronofistSkillController відповідає за управління хроносферою та взаємодією з ворогами всередині неї.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		44

Основні елементи класу ChronofistSkillController:

1. Метод SetupChrono() - налаштовує параметри хроносфери.
2. Метод Update() - відповідає за оновлення стану хроносфери, контроль часу та атаки клонів.
3. Метод ReleaseCloneAttack() - викликає атаку клонів.
4. Метод CloneAttackLogic() - логіка атак клонів, з випадковим вибором ворогів.
5. Метод FinishChronoAbility() - завершує дію хроносфери та знищує гарячі клавіші.
6. Методи DestroyHotKeys(), OnTriggerEnter2D(), OnTriggerExit2D(), CreateHotKey(), AddEnemyToList() - логіка для взаємодії з ворогами та створення гарячих клавіш.

Клас ChronofistHotkeyController відповідає за створення гарячих клавіш, які гравець натискає для нанесення ударів проекціями.

```
public class ChronofistHotkeyController : MonoBehaviour
{
    private SpriteRenderer sr;
    private KeyCode myHotKey;
    private TextMeshProUGUI myText;

    private Transform myEnemy;
    private ChronofistSkillController chrono;

    public void SetupHotKey(KeyCode _myNewHotKey, Transform
_myEnemy, ChronofistSkillController _myChrono){
        sr = GetComponent<SpriteRenderer>();
        myText = GetComponentInChildren<TextMeshProUGUI>();

        myEnemy = _myEnemy;
        chrono= _myChrono;

        myHotKey = _myNewHotKey;
        myText.text = _myNewHotKey.ToString();
    }

    private void Update(){
        if (Input.GetKeyDown(myHotKey))
        {
            chrono.AddEnemyToList(myEnemy);

            myText.color = Color.clear;
            sr.color = Color.clear;
        }
    }
}
```

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		45

```

    }
}
}

```

Основні елементи класу ChronofistHotkeyController:

1. Метод SetupHotKey() - налаштовує гарячу клавішу та її взаємодію з ворогом.
2. Метод Update() - відстежує натискання гарячої клавіші та додає ворога до списку цілей.

Клас ScytheThrowSkill відповідає за здібність кидання ножа різних типів. Гравець може обрати один із чотирьох типів ножів: звичайний, відскакуючий, пробивний або обертовий. Цей клас забезпечує основну логіку для запуску ножів, управління їх поведінкою та відображення напрямку кидка.

Основні елементи класу ScytheThrowSkill

1. SetupGravity() - встановлює значення гравітації для ножа в залежності від його типу.
2. Update() - оновлює напрямок прицілювання та позиції точок прицілювання, коли гравець натискає клавішу R.
3. CreateScythe() - створює новий ніж відповідно до поточного типу ножа, налаштовує його властивості та активує його поведінку.
4. AimDirection() - обчислює напрямок прицілювання від гравця до курсора миші.
5. DotsActive(bool isActive) - включає або вимикає точки прицілювання.
6. GenerateDots() - генерує об'єкти точок прицілювання.
7. DotsPosition(float t) - обчислює позицію кожної точки прицілювання в залежності від часу польоту ножа та гравітації.

Клас ScytheSkillController забезпечує управління поведінкою ножа після його кидання. Він включає логіку для різних типів ножів, а також обробку зіткнень з ворогами.

1. DestroyMe() - знищує об'єкт ножа через певний час.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		46

2. SetupScythe(Vector2 direction, float gravityScale, Player playerInstance, float freezeTime, float returnSpeedValue) - налаштовує основні параметри ножа після його створення.
3. SetupBounce(bool isBouncingValue, int bounceAmountValue, float bounceSpeedValue) - налаштовує параметри для відскакуючого ножа.
4. SetupPierce(int pierceAmountValue) - налаштовує параметри для пробивного ножа.
5. SetupSpin(bool isSpinningValue, float maxTravelDistanceValue, float spinDurationValue, float hitCooldownValue) - налаштовує параметри для обертового ножа.
6. ReturnScythe()-активує повернення ножа до гравця.
7. Update() - оновлює стан ножа, включаючи логіку обертання, повернення та відскакування.
8. SpinLogic() - логіка обертання ножа та обробка пошкоджень ворогам.
9. StopWhenSpinning() - зупиняє рух ножа при досягненні максимального дистанції обертання.
10. BounceLogic() - логіка відскакування ножа між ворогами.
11. OnTriggerEnter2D(Collider2D collision) - обробка зіткнень ножа з ворогами та іншими об'єктами.
12. ScytheThrowSkillDamage(Enemy enemy) - застосовує пошкодження ворогу та заморожує його на певний час.
13. SetupTargetsForBounce(Collider2D collision) - налаштовує цілі для відскакуючого ножа при зіткненні.
14. StuckInto(Collider2D collision) - логіка зупинки ножа при зіткненні з об'єктом, якщо ніж не має можливості пробивати або обертатися далі.

2.4 Створення системи чармів

Система чармів є важливою складовою механіки гри, яка дозволяє гравцям адаптувати свого персонажа під свій стиль гри та підвищувати його ефективність у боях. Чарми представляють собою спеціальні предмети, які надають персонажу різноманітні бонуси або нові здібності. Це можуть бути покращення бойових навичок, збільшення захисту, підвищення здоров'я, або навіть нові унікальні здібності, які можуть кардинально змінити ігровий процес.

Впровадження системи чармів надає гравцям можливість експериментувати з різними комбінаціями, створюючи унікальні тактики для проходження гри. Вона додає елемент стратегічного планування, оскільки гравець повинен обирати чарми, які найкраще підходять для конкретних викликів, що стоять перед ним у грі.

Розробка системи чармів включає кілька ключових аспектів:

1. Створення базової логіки для чармів: Це включає визначення, які саме бонуси або здібності надаватиме кожен чарм і як вони будуть впливати на персонажа.

2. Балансування чармів: Важливо забезпечити, щоб жоден з чармів не був занадто сильним або занадто слабким. Всі чарми повинні бути корисними в певних ситуаціях, що стимулюватиме гравців використовувати різні комбінації.

Далі ми детально розглянемо кожен чарм, який доступний у грі, і проаналізуємо, які бонуси та можливості він надає персонажу. Це допоможе краще зрозуміти, як саме вони впливають на ігровий процес і як гравець може використовувати їх для досягнення своїх цілей у грі.

Абстрактний клас Charm

Усі чарми наслідуються від абстрактного класу Charm, що реалізує шаблон проектування "Шаблонний метод" (Template Method). Цей шаблон дозволяє визначити основну структуру алгоритму, тоді як підкласи можуть

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		48

реалізовувати або перевизначати окремі етапи алгоритму без зміни його загальної структури.

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

public abstract class Charm : MonoBehaviour
{
    [SerializeField] protected PlayerStats playerStats;

    public virtual void ActivateEffect()
    {
        CharmEffect();
    }

    public virtual void DeactivateEffect()
    {
    }

    public abstract void CharmEffect();
}
```

Клас Charm має такі методи:

- ActivateEffect() - активує ефект чарма. Викликає метод CharmEffect(), який реалізується у підкласах.
- DeactivateEffect() - деактивує ефект чарма. Може бути перевизначений у підкласах.
- CharmEffect() - абстрактний метод, який визначає основний ефект чарма. Його реалізують підкласи.

Опис чармів

У грі існує 14 чармів, серед яких є унікальні та звичайні. Тільки один унікальний чарм може бути у використанні одночасно. Розглянемо декілька з них:

1. Aegis Amulet - зменшує отримуваний персонажем урон на певний відсоток, надаючи додатковий захист під час інтенсивних боїв. Також він відображає невелику частину отриманого урону назад ворогам. Це дозволяє гравцеві зменшити вхідний урон і одночасно наносити додатковий урон ворогам, роблячи бій більш ефективним і цікавим.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		49

Чарм AegisAmuletCharm реалізує свої ефекти через методи ActivateEffect(), DeactivateEffect() та CharmEffect().

CharmEffect(). Цей метод додає значення damageReductionPercentage до відсотка зменшення урону персонажа.

```
public override void CharmEffect()  
{  
    playerStats.damageReceivedReductionPercentage +=  
    damageReductionPercentage;  
}
```

ActivateEffect(). Викликає CharmEffect(), активуючи зменшення урону. Підписується на подію OnDamageReceived, щоб відображати урон ворогу.

```
public override void ActivateEffect()  
{  
    base.ActivateEffect();  
    playerStats.OnDamageReceived += ReflectDamageAndEnemy;  
}
```

DeactivateEffect(). Зменшує відсоток зменшення урону на величину damageReductionPercentage. Відписується від події OnDamageReceived.

```
public override void DeactivateEffect()  
{  
    playerStats.damageReceivedReductionPercentage -=  
    damageReductionPercentage;  
    base.DeactivateEffect();  
    playerStats.OnDamageReceived -= ReflectDamageAndEnemy;  
}
```

2. Fate's Gambit - закликає сили випадковості на допомогу гравцеві, підвищуючи їх критичні удари і надаючи додаткові ефекти в бою. Це чарм збільшує шанси на критичний удар і його силу, а також надає випадкові бонуси до швидкості, урону та захисту під час бою.

Чарм MulticastCharm реалізує свої ефекти через методи ActivateEffect(), DeactivateEffect() та CharmEffect(). CharmEffect(). Цей метод підписується на подію OnEnemyTakeDamage, щоб перевіряти критичні удари.

```
public override void CharmEffect()  
{  
    EnemyStats.OnEnemyTakeDamage += CheckCritical;  
}
```

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		50

ActivateEffect(). Викликає CharmEffect(), активуючи збільшення шансів і сили критичних ударів. Додає модифікатори до шансів і сили критичних ударів.

```
public override void ActivateEffect()
{
    base.ActivateEffect();

    playerStats.critChance.AddModifier(criticalChanceIncrease);

    playerStats.critPower.AddModifier(criticalPowerIncrease);
}
```

DeactivateEffect(). Знімає модифікатори шансів і сили критичних ударів, відписується від події OnEnemyTakeDamage, а також деактивує активні бонуси Fortune та Fate Shield.

```
public override void DeactivateEffect()
{
    base.DeactivateEffect();

    playerStats.critChance.RemoveModifier(criticalChanceIncrease);

    playerStats.critPower.RemoveModifier(criticalPowerIncrease);
    EnemyStats.OnEnemyTakeDamage -= CheckCritical;
    DisableFortune();
    DisableFateShield();
}
```

CheckCritical(EnemyStats enemyStats). Викликається при кожному нанесенні урону ворогам, перевіряючи, чи є урон критичним. Якщо критичний урон успішно виконано, з певною ймовірністю активуються бонуси Fortune або Fate Shield.

```
private void CheckCritical(EnemyStats enemyStats)
{
    if (Random.value * 100 < criticalChanceIncrease)
    {
        if (!isFortuneActive && Random.value <= 0.5)
        {
            isFortuneActive = true;
            player.moveSpeed += fortuneSpeedBonus;

            playerStats.damage.AddModifier(fortuneDamageBonus);
            Invoke("DisableFortune", fortuneDuration);
        }
        if (!isFateShieldActive && Random.value > 0.5)
        {
            isFateShieldActive = true;
            playerStats.armor.AddModifier(fateShieldBonus);
        }
    }
}
```

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		51

```

        Invoke("DisableFateShield", fateShieldDuration);
    }
}

```

DisableFortune(). Деактивує бонус Fortune, повертаючи швидкість руху і урон до початкових значень.

```

private void DisableFortune()
{
    isFortuneActive = false;
    player.moveSpeed = originalSpeed;
    playerStats.damage.RemoveModifier(fortuneDamageBonus);
}

```

DisableFateShield(). Деактивує бонус Fate Shield, повертаючи показник захисту до початкового значення.

```

private void DisableFateShield()
{
    isFateShieldActive = false;
    playerStats.armor.RemoveModifier(fateShieldBonus);
}

```

3. Vampiric Embrace Charm - значно збільшує максимальне здоров'я гравця, підвищує вампіризм і зменшує отримуваний урон. Однак гравець не може використовувати фляги для зцілення, коли цей чарм активний. Це дозволяє гравцеві мати більше здоров'я і відновлювати його за рахунок нанесеного ворогам урону, що робить його більш витривалим у бою.

Чарм VampiricEmbrace реалізує свої ефекти через методи ActivateEffect(), DeactivateEffect() та CharmEffect().

CharmEffect(). Цей метод збільшує максимальне здоров'я персонажа на певний відсоток і додає значення зменшення урону.

```

public override void CharmEffect()
{
    hpAdded = playerStats.maxHealth.GetValue() *
(maxHPIncreasePercentage / 100f);
    playerStats.maxHealth.AddModifier(hpAdded);
    playerStats.allBars.UpdateBars();

    playerStats.damageReceivedReductionPercentage +=
damageReductionPercentage;
}

```

ActivateEffect(). Викликає CharmEffect(), активуючи збільшення максимального здоров'я та зменшення урону. Підписується на подію onDamageDealt, щоб застосовувати вампіризм. Деактивує можливість використовувати фляги.

```
public override void ActivateEffect()
{
    base.ActivateEffect();
    playerStats.onDamageDealt += ApplyVampirism;
    healthFlask.CanUseFlasks(false);
    flaskGUI.Deactivate();
    Debug.Log("HP ADDED:" + hpAdded);
}
```

DeactivateEffect(). Відписується від події onDamageDealt, знімає модифікатори збільшення максимального здоров'я та зменшення урону, а також дозволяє використовувати фляги.

```
public override void DeactivateEffect()
{
    playerStats.onDamageDealt -= ApplyVampirism;

    playerStats.maxHealth.RemoveModifier(hpAdded);
    healthFlask.CanUseFlasks(true);
    flaskGUI.Activate();

    playerStats.damageReceivedReductionPercentage -=
    damageReductionPercentage;
    Debug.Log("HP REMOVED:" + hpAdded);
}
```

ApplyVampirism(float damageDealt). Викликається при нанесенні урону ворогам, відновлюючи здоров'я гравця на певний відсоток від нанесеного урону.

```
private void ApplyVampirism(float damageDealt)
{
    float healAmount = damageDealt * (vampirismPercentage /
    100f);
    playerStats.IncreaseHealthBy(healAmount);
    playerStats.allBars.SetHealth(playerStats.currentHealth,
    playerStats.maxHealth.GetValue());
}
```

Висновки до другого розділу

Було здійснено комплексний підхід до розробки програмного забезпечення для ігрового додатку. Розглянуто та реалізовано ключові аспекти

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		53

архітектури та функціоналу гри, що забезпечили її ефективність та ігрову привабливість.

Було створено детальні діаграми варіантів використання, активності, класів, компонентів та розгортання, що дозволили чітко визначити структуру та взаємодію між різними елементами гри. Це сприяло забезпеченню високої якості коду та полегшило процес подальшої розробки та підтримки гри.

Було створено головного персонажа гри з урахуванням вимог ігрового процесу та естетичних параметрів. Реалізовано систему фізичних параметрів та взаємодії з оточенням, що забезпечило природний ігровий досвід для користувача.

Розроблено та інтегровано систему здібностей героя та зброї, яка використовує патерн проектування "Стратегія" для забезпечення гнучкості та розширюваності. Це дозволило легко додавати нові здібності та стилі зброї, а також налаштовувати їхню поведінку без значних змін у основному коді.

Було реалізовано систему чармів, яка дозволяє гравцеві покращувати свого персонажа та адаптувати його до різних стилів гри. Чарми наслідуються від абстрактного класу "Charm", що реалізує патерн проектування "Шаблонний метод". Це забезпечило можливість легко створювати нові чарми з різноманітними ефектами та впливами на ігровий процес.

В цілому, при розробці програмного забезпечення для ігрового додатку було продемонстровано ефективне використання патернів проектування та сучасних підходів до розробки, що дозволило створити гнучку та масштабовану архітектуру гри. Це забезпечило високу якість програмного продукту та задоволення вимог користувачів.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		54

РОЗДІЛ 3. ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО ТЕСТУВАННЯ

3.1 Порядок встановлення та налаштування системи

Гра "Dethrone" була розроблена на платформі Unity і доступна для завантаження на itch.io. У цьому підрозділі ми розглянемо вимоги до комп'ютера, процес встановлення та налаштування гри, щоб забезпечити її коректну роботу на вашому пристрої.

Вимоги до комп'ютера

Щоб грати в "Dethrone", комп'ютер повинен відповідати наступним мінімальним системним вимогам:

Мінімальні системні вимоги:

- Операційна система: Windows 10 (64-bit) або новіше
- Процесор: Intel Core 2 Duo E5200
- Оперативна пам'ять: 4 GB RAM
- Відеокарта: GeForce GTX 1030 (2GB)
- DirectX: Версія 11
- Місце на диску: 5 GB доступного місця
- Звукова карта: Будь-яка

Рекомендовані системні вимоги:

- Операційна система: Windows 10 (64-bit) або новіше
- Процесор: Intel Core i5
- Оперативна пам'ять: 8 GB RAM
- Відеокарта: GeForce GTX 1060
- DirectX: Версія 11
- Місце на диску: 5 GB доступного місця
- Звукова карта: Будь-яка

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		55

3.2 Опис дизайну рівнів

Левел дизайн [18] - це процес створення, планування та налаштування рівнів у відеоіграх. Він включає в себе створення архітектури, установлення об'єктів, розміщення ворогів, пасток, прихованих предметів, пазлів та інших ігрових елементів на рівні таким чином, щоб забезпечити гравцеві цікавий та захоплюючий досвід гри.

Левел дизайн важливий, оскільки він впливає на емоційну та інтелектуальну взаємодію гравця з грою. Добре спроектований рівень може стимулювати гравця вивчати ігровий світ, розв'язувати головоломки, боротися з ворогами та досягати цілей, що зробить гру більш захоплюючою та задовольняючою.

У Metroidvania жанрі левел дизайн використовується для створення великих, інтерактивних та нелінійних ігрових світів [19], які гравець може вивчати. Деякі особливості, які використовуються в левел дизайні Metroidvania ігор, включають:

1. Системи ключів та дверей: Деякі області можуть бути заблоковані до тих пір, поки гравець не знайде відповідний ключ або не вирішить певну головоломку.

2. Сховані області та секрети: Рівні можуть містити сховані проходи, секретні кімнати або приховані предмети, які заохочують гравця вивчати кожний куточок ігрового світу.

3. Різноманітність пасток та ворогів: Вороги та пастки можуть бути розміщені таким чином, щоб створити складні ситуації, які вимагають від гравця стратегічного планування та уміння.

У грі наявні усі техніки для реалізації хорошого левел дизайну. На (рис. 3.1) та (рис. 3.2) можна побачити як різні локації поєднуються між собою завдяки системі ключів та дверей. Спочатку гравець дійде до двері і вона буде зачинена, щоб її відкрити потрібно дістатися наступної локації та відкрити двері з іншої сторони. Така система створена для зручності переміщення між локаціями та задля збереження часу гравця.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		56



Рисунок 3.1 - Перехід з підземелля замку, який відкриває двері в іншій локації



Рисунок 3.2 - Двері, які відкриваються тільки зі сторони підземелля

Секретні проходи та локації створенні для збереження зацікавленості гравця та щоб додати бажання досліджувати локації. У данному випадку було створенно підземельну локацію (рис. 3.3), у якій знаходиться велика кількість

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		57

корисних речей: чарми, предмети, переходи у інші локації та цікаві вороги. Щоб потрапити до цієї локації треба знайти відповідний ключ. Якщо гравець випадково попаде у дану локацію, то не зможе її пройти, бо вона спеціально зроблена так, що гравець нічого у ній не бачить. Якщо гравець зацікавиться цим місцем, він знайде спеціального торговця на карті, який має у своєму арсеналі цікавий предмет-магічну лампу. Придбавши її гравець зможе бачити у темряві, тому з точки зору геймдизайну, ми вважаємо гравця розумним та поважаємо його винахідливість. Гра не робить прями підказки, що треба пійти з точки «а» до точки «б», а лише дає невеликі натяки на це, гравець сам вирішує, що йому робити.



Рисунок 3.3 - Секретна локація

У кожній метроїдванії мають бути пастки та вороги, які створюють цікаві та складні ситуації у грі. Гравець має бути зосередженим та приймати правильні рішення, щоб продвинути далі. У даному випадку показано (рис. 3.4) момент в грі, де спочатку гравцю треба вбити ворога, який постійно заповільнює його, а далі вже проходити секцію з шипами. Таким чином гравець відчуває вплив своїх рішень на ігровий процес.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		58

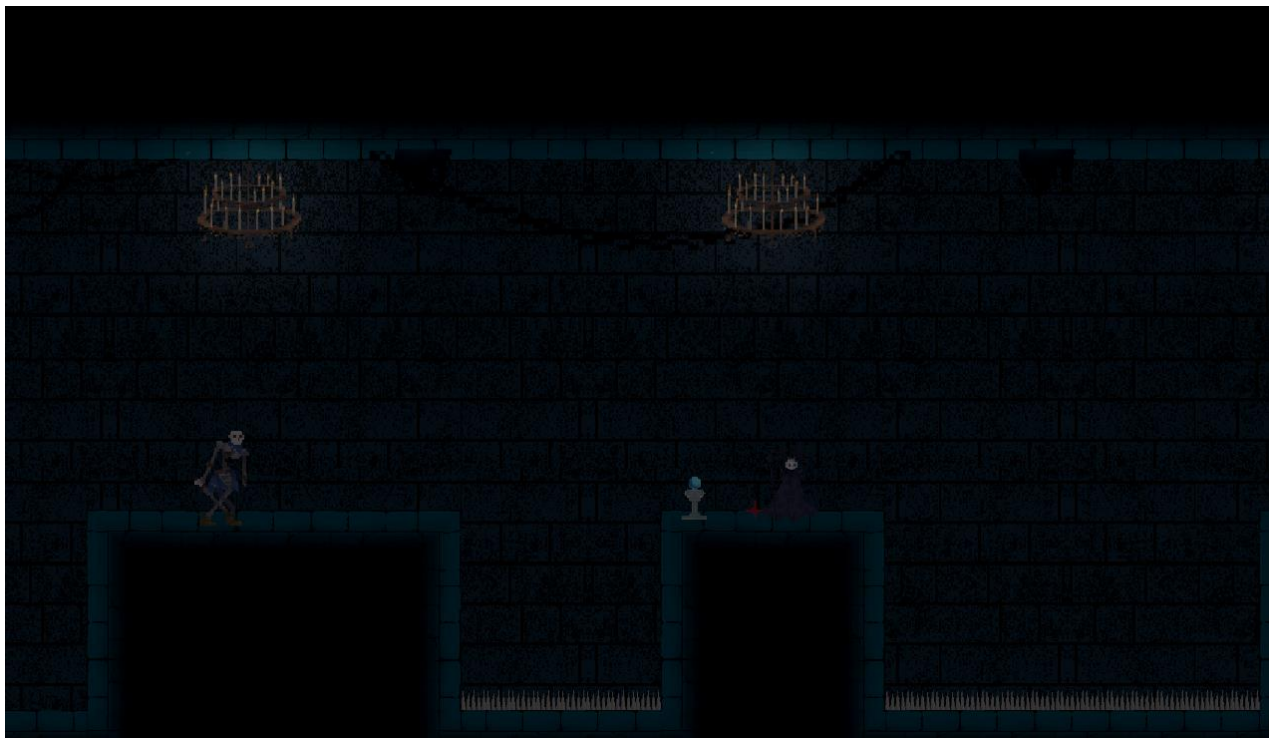


Рисунок 3.4 - Пастки з ворогами

3.3 Тестування роботи механік

Тестування ігрового процесу [20] є важливою частиною розробки гри, оскільки воно дозволяє впевнитись у коректній роботі всіх механік, функцій та компонентів гри. Основна мета тестування полягає у виявленні та виправленні помилок, підвищенні якості гри, а також в перевірці відповідності гри вимогам та очікуванням користувачів.

Основні задачі тестування:

1. Перевірка функціональності. Ми перевіряємо, чи працюють всі механіки, здібності героя, зброя і інші компоненти гри згідно з їх задумкою.
2. Виявлення помилок. Тестування допомагає виявити та виправити будь-які помилки, що можуть виникнути під час гри, такі як програмні баги, невідповідності у діях героя та інше.
3. Перевірка на відповідність вимогам. Ми перевіряємо, чи відповідає гра встановленим системним вимогам та очікуванням користувачів щодо геймплею та графічної якості.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		59

4. Підвищення якості гри. Тестування допомагає забезпечити високу якість гри шляхом виявлення та виправлення потенційних проблем ще на ранніх стадіях розробки.

При тестуванні треба враховувати не лише технічні аспекти, а ще й як різні функції гри відповідають заданому геймдизайну. Отримані результати були приведені нижче (табл. 3.1).

Таблиця 3.1

Тестування ПЗ

Об'єкт тестування	Тест	Очікувані результати	Реальні результати
Рух головного персонажа	Спроба керування гравцем	Плавний рух гравця	Плавний рух гравця
Взаємодія зі зброєю	Використання зброї	Вдалий удар по ворогу, нанесення шкоди	Вдалий удар по ворогу, нанесення шкоди
Взаємодія зі здібностями героя	Активація здібностей	Відтворення ефекту	Відтворення ефекту
Система чармів	Надягання чармів	Активація ефектів чармів	Активація ефектів чармів
Стилі зброї	Випробування різних типів зброї та їх вплив на геймплей	Різноманітність та варіативність в стилях зброї	Наявність різних стилів зброї та їх вплив на геймплей
Вплив на різні стилі гри	Випробування гри з різними стилями геймплею (агресивний, оборонний, балансований)	Відчуття різниці у геймплеї залежно від обраного стилю	Виявлення відмінностей у геймплеї при виборі різних стилів

Взаємодія між різними здібностями та стилями зброї	Використання комбінації здібностей героя	Виникнення нових стратегій та можливостей в геймплеї	Виявлення взаємодії між різними здібностями та їх вплив на гру
--	--	--	--

Висновки до третього розділу

У даному підрозділі було розглянуто порядок встановлення та налаштування системи, де виявили мінімальні та рекомендовані системні вимоги для успішного запуску гри "Dethrone".

Також було досліджено дизайн рівнів гри, який демонструє використання різноманітних технік, характерних для жанру Metroidvania, таких як система ключів та дверей, приховані області, секрети, пастки та вороги для створення цікавого та захопливого досвіду гри.

Крім того, наголошено на важливості тестування ігрового процесу для перевірки коректної роботи механік, функцій та компонентів, виявлення та виправлення помилок, а також забезпечення відповідності гри вимогам та очікуванням користувачів.

Проведене тестування в грі "Dethrone" підтвердило належну роботу основних механік, таких як рух головного персонажа, взаємодія зі зброєю та здібностями, система чармів, різноманітність стилів зброї. Також було виявлено вплив різних стилів геймплею та взаємодію між здібностями й стилями зброї, що відкриває нові стратегії та можливості.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було розроблено оригінальну комп'ютерну гру жанру Metroidvania під назвою "Dethrone". Успішно визначено мету, завдання та проаналізовано популярні ігри в даному жанрі. Сформовано концепт документ з описом унікальних особливостей гри, жанру та цільової аудиторії. Обґрунтовано вибір ігрового рушія Unity та середовища програмування Visual Studio.

Здійснено комплексний підхід до розробки програмного забезпечення ігрового додатку. Створено детальні діаграми використання, активності, класів, компонентів та розгортання, що забезпечило високу якість коду та полегшило процес розробки й підтримки гри. Реалізовано головного персонажа з системою фізичних параметрів та взаємодії з оточенням для цікавого ігрового досвіду.

Розроблено та інтегровано систему здібностей героя для гнучкості й розширюваності. Впроваджено систему чармів на основі патерну "Шаблонний метод" для покращення персонажа та адаптації до різних стилів гри.

Визначено мінімальні та рекомендовані системні вимоги для успішного запуску гри. Проаналізовано дизайн рівнів з використанням характерних для жанру Metroidvania технік для створення цікавого ігрового досвіду. Наголошено на важливості тестування для перевірки коректної роботи механік, функцій, компонентів та виявлення помилок.

Проведене тестування підтвердило належну роботу основних механік гри, таких як рух персонажа, взаємодія зі зброєю, здібностями, система чармів і різноманітність стилів зброї. Виявлено вплив різних стилів геймплею та взаємодію між здібностями й стилями зброї, що відкриває нові стратегії та можливості для гравця.

Таким чином, в результаті виконання кваліфікаційної роботи було успішно розроблено якісний та оригінальний ігровий додаток у жанрі Metroidvania з використанням сучасних підходів та технологій програмування, що відповідає поставленим вимогам та очікуванням користувачів.

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		62

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. How to create your own Metroidvania [Електронний ресурс] - Режим доступу до ресурсу: <https://dreamnoid.com/articles/how-to-create-your-own-metroidvania>
2. Introduction to 2D Animation [Електронний ресурс] - Режим доступу до ресурсу: <https://docs.unity3d.com/Packages/com.unity.2d.animation@10.1/manual/index.html>
3. Understanding State Machines [Електронний ресурс] - Режим доступу до ресурсу: <https://www.freecodecamp.org/news/state-machines-basics-of-computer-science-d42855debc66/>
4. What is a State Machine? [Електронний ресурс] - Режим доступу до ресурсу: <https://www.digikey.com/es/maker/tutorials/2023/what-is-a-state-machine>
5. «The Last Faith» - Kumi Souls Games [Електронний ресурс] - Режим доступу до ресурсу: <https://www.thelastfaithgame.com>
6. «Blasphemous 2» - The Game Kitchen [Електронний ресурс] - Режим доступу до ресурсу: <https://www.blasphemous2game.com>
7. «Moonscars» - Black Mermaid [Електронний ресурс] - Режим доступу до ресурсу: <https://www.humblegames.com/games/moonscars/>
8. Why is Unity your Best Bet for Game Development in 2024 [Електронний ресурс] - Режим доступу до ресурсу: <https://300mind.studio/blog/unity-for-game-development/>
9. Unity Documentation [Електронний ресурс] - Режим доступу до ресурсу: <https://docs.unity.com>
10. What is Visual Studio? [Електронний ресурс] - Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022>
11. What is Use Case Diagram? [Електронний ресурс] - Режим доступу до ресурсу: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/>

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		63

12. Create a UML activity diagram - [Електронний ресурс]-Режим доступу до ресурсу: <https://support.microsoft.com/en-gb/office/create-a-uml-activity-diagram-19745dae-2872-4455-a906-13b736f01685>
13. What is Class Diagram? [Електронний ресурс] - Режим доступу до ресурсу: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/>
14. What is Component Diagram? [Електронний ресурс] - Режим доступу до ресурсу: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-component-diagram/>
15. What is Deployment Diagram? [Електронний ресурс] - Режим доступу до ресурсу: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-deployment-diagram/>
16. HOW TO CREATE A VIDEO GAME CHARACTER: KEY PRINCIPLES [Електронний ресурс] - Режим доступу до ресурсу: <https://room8studio.com/news/how-to-create-a-video-game-character-key-principles/>
17. Create modular and maintainable code with the observer pattern [Електронний ресурс] - Режим доступу до ресурсу: <https://learn.unity.com/tutorial/65de086fedbc2a06ac2aca58>
18. How to Design Great Metroidvania Levels [Електронний ресурс] - Режим доступу до ресурсу: <https://www.youtube.com/watch?v=bAHXYfP38CA>
19. How to design a great Metroidvania map [Електронний ресурс]-Режим доступу до ресурсу: <https://www.pcgamer.com/how-to-design-a-great-metroidvania-map/>
20. Basic Tips and Strategies [Електронний ресурс] - Режим доступу до ресурсу: <https://starloopstudios.com/game-testing-101-tips-and-strategies/>

ДОДАТКИ

		Кірій Д.О.			КН.КР.Б-122-24-ПЗ	Арк.
		Марчук Г.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		65

Стилі зброї

WeaponStyle.cs:

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

public class WeaponStyle : MonoBehaviour
{
    public virtual void ActivateFirstUpgrade()
    {
    }

    public virtual void ActivateSecondUpgrade()
    {
    }

    public virtual void ActivateThirdUpgrade()
    {
    }

    public virtual void DeactivateEffect()
    {
    }
}
```

RedWeaponStyle.cs:

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

public class RedWeaponStyle : WeaponStyle
{
    [Header("Base Upgrade")]
    [SerializeField] public CrimsonCarnage crimsonCarnage;
    [Header("First Upgrade")]
    [SerializeField] private float maxHealthIncrease;
    [SerializeField] private float vampirismPercentage;
    [Header("Second Upgrade")]
    [SerializeField] public float bleedingDamage;
    [SerializeField] public float bleedTickPerSecond;
    [SerializeField] public float bleedDuration; public float
originalBleedDuration;
    private float damageTimer = 0f;
    public bool isBleeding;
```

```

public bool isBaseActive;
public bool isFirstActive;
public bool isSecondActive;

private PlayerStats playerStats;
private Player player;
private void Start()
{
    playerStats = FindObjectOfType<PlayerStats>();
    player = PlayerManager.instance.player;
    originalBleedDuration = bleedDuration;
}
private void Update()
{
    if (isSecondActive && bleedDuration > 0 && isBleeding)
    {
        bleedDuration -= Time.deltaTime;
        damageTimer += Time.deltaTime;
        if (damageTimer >= 1f / bleedTickPerSecond) //
        {
            damageTimer = 0f;
            crimsonCarnage.BleedEffect(bleedingDamage);
        }
        if(bleedDuration == 0)
        {
            isBleeding = false;
        }
    }
}

public override void ActivateFirstUpgrade()
{
    isBaseActive = true;
}

public override void ActivateSecondUpgrade()
{
    isFirstActive = true;
    playerStats.maxHealth.AddModifier(maxHealthIncrease);
    playerStats.onDamageDealt += ApplyVampirism;
}

public override void ActivateThirdUpgrade()
{
    isSecondActive = true;
}

public override void DeactivateEffect()
{
    base.DeactivateEffect();
    if (isFirstActive)
    {

```

```

playerStats.maxHealth.RemoveModifier(maxHealthIncrease);
    playerStats.onDamageDealt -= ApplyVampirism;
}

    isBaseActive = false;
    isFirstActive = false;
    isSecondActive = false;

}
private void ApplyVampirism(float damageDealt)
{
    float healAmount = damageDealt * (vampirismPercentage /
100f);
    playerStats.IncreaseHealthBy(healAmount);
    playerStats.allBars.SetHealth(playerStats.currentHealth,
playerStats.maxHealth.GetValue());
}
}

```

GreenWeaponStyle.cs:

```

using System.Collections;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using UnityEngine;

public class GreenWeaponStyle : WeaponStyle
{
    [Header("Base Upgrade")]
    [Header("First Upgrade")]
    [SerializeField] private float moveSpeedIncreasePercent;
    [SerializeField] private float reduceCooldown;

    private bool isBaseActive;
    private bool isFirstActive;
    private bool isSecondActive;

    [SerializeField] private MirageMirage miracleMirage;
    private Player player;
    private PlayerStats playerStats;
    private float originalSpeed;
    private void Start()
    {
        player = FindObjectOfType<Player>();
        playerStats = FindObjectOfType<PlayerStats>();
        originalSpeed = player.moveSpeed;
    }
    private void Update()
    {
        if
(Input.GetKeyUp(InputSettings.Instance.GetKeyForAction(InputSetti
ngs.Instance.Button_ability)) && isBaseActive &&
playerStats.currentUlt == playerStats.maxUlt.GetValue())

```

```

        {
            miracleMirage.ActivateMirage();
        }
    }
    public override void ActivateFirstUpgrade()
    {
        isBaseActive = true;
    }

    public override void ActivateSecondUpgrade()
    {
        isFirstActive = true;
        player.moveSpeed += player.moveSpeed *
moveSpeedIncreasePercent/100;

SkillManager.instance.dash.ReduceCooldown(reduceCooldown);
    }

    public override void ActivateThirdUpgrade()
    {
        isSecondActive = true;
        SkillManager.instance.clone.CanCreateCloneOnStart(true);
    }
    public override void DeactivateEffect()
    {
        base.DeactivateEffect();
        if(isFirstActive)
        {
            player.moveSpeed = originalSpeed;
            SkillManager.instance.dash.SetDefaultCooldown();
        }
        if (isSecondActive)
        {

SkillManager.instance.clone.CanCreateCloneOnStart(false);
        }

        isBaseActive = false;
        isFirstActive = false;
        isSecondActive = false;
    }
}

```

BlueWeaponStyle.cs:

```

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

public class BlueWeaponStyle : WeaponStyle
{
    [Header("Base Upgrade")]
    public GameObject celestialNexusPrefab; // İöâôää íáúâêòà ñ
ñêöèïòîî CelestialNexus

```

```

private GameObject celestialNexusInstance;
[SerializeField] private float timeToDestroy;
[Header("First Upgrade")]
[SerializeField] private float maxManaIncrease;
[SerializeField] private float manaRegenerationRate;
[Header("Second Upgrade")]
[SerializeField] private float magicDamageIncreasePerOrb;
[SerializeField] private int maxOrbCount = 3;
[SerializeField] private int currentOrbCount;
[SerializeField] private float timeToSpawnSphere;
private bool isAddingSphere = false;

private bool isBaseActive;
private bool isFirstActive;
private bool isSecondActive;

private PlayerStats playerStats;
private SphereGUI sphereGUI;
private void Start()
{
    playerStats = FindObjectOfType<PlayerStats>();
    sphereGUI = FindObjectOfType<SphereGUI>();
}
private void Update()
{
    if
(Input.GetKeyUp(InputSettings.Instance.GetKeyForAction(InputSetti
ngs.Instance.Button_ability)) && isBaseActive &&
playerStats.currentUlt == playerStats.maxUlt.GetValue())
    {
        ActivateCelestialNexus();
    }
    if (isSecondActive && currentOrbCount < maxOrbCount &&
!isAddingSphere)
    {
        StartCoroutine(AddSphere());
    }
}

private void RemoveAllSpheres(float remove)
{
    currentOrbCount = 0;
    playerStats.magicAmplify.RemoveAllModifiers();
    for (int i = 0; i < sphereGUI.images.Length; i++)
    {
        sphereGUI.ClearImage(i);
    }
}

private IEnumerator AddSphere()
{
    if (currentOrbCount < maxOrbCount)

```

```

        {
            isAddingSphere = true;
            yield return new WaitForSeconds(timeToSpawnSphere);
            currentOrbCount++;
            sphereGUI.FillImage(currentOrbCount - 1);

playerStats.magicAmplify.AddModifier(magicDamageIncreasePerOrb);
            isAddingSphere = false;
        }
    }

    public override void ActivateFirstUpgrade()
    {
        isBaseActive = true;
    }

    public override void ActivateSecondUpgrade()
    {
        isFirstActive = true;
        playerStats.maxMana.AddModifier(maxManaIncrease);

playerStats.manaRegenRate.AddModifier(manaRegenerationRate);
        playerStats.allBars.SetMana(playerStats.currentMana,
playerStats.maxMana.GetValue());
    }

    public override void ActivateThirdUpgrade()
    {
        isSecondActive = true;
        sphereGUI.Activate();
        playerStats.OnDamageReceived += RemoveAllSpheres;
    }

    public override void DeactivateEffect()
    {
        base.DeactivateEffect();
        if (isFirstActive)
        {
            playerStats.maxMana.RemoveModifier(maxManaIncrease);

playerStats.manaRegenRate.RemoveModifier(manaRegenerationRate);
            playerStats.allBars.SetMana(playerStats.currentMana,
playerStats.maxMana.GetValue());
        }
        if(isSecondActive)
        {
            RemoveAllSpheres(0);
            sphereGUI.Deactivate();
        }

        isBaseActive = false;
        isFirstActive = false;
        isSecondActive = false;
    }

```

```

    }

    public void ActivateCelestialNexus()
    {
        if (celestialNexusPrefab != null)
        {
            playerStats.DecreaseUlt(playerStats.maxUlt.GetValue());
            // ĩîëó÷àâî ññûëêó íà èãðîêà èç PlayerManager
            PlayerManager playerManager = PlayerManager.instance;
            if (playerManager != null && playerManager.player !=
null)
            {
                celestialNexusInstance =
Instantiate(celestialNexusPrefab,
playerManager.player.transform.position +
playerManager.player.transform.right * 2f, Quaternion.identity);
                Destroy(celestialNexusInstance, timeToDestroy);
            }
            else
            {
                Debug.LogError("PlayerManager or player reference
is missing!");
            }
        }
        else
        {
            Debug.LogError("Celestial Nexus prefab is not
assigned!");
        }
    }
}

```

Чарми

Charm.cs:

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

public abstract class Charm : MonoBehaviour
{
    [SerializeField] protected PlayerStats playerStats;

    public virtual void ActivateEffect()
    {
        CharmEffect();
    }
    public virtual void DeactivateEffect()
    {
    }

    public abstract void CharmEffect();
}
```

AegisAmuletCharm.cs:

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

public class AegisAmuletCharm : Charm
{
    [SerializeField] private float damageReductionPercentage;
    [SerializeField] private float damageReflectionPercentage;

    private CharacterStats enemyInflictingDamage;

    public override void CharmEffect()
    {
        playerStats.damageReceivedReductionPercentage +=
        damageReductionPercentage;
    }

    public override void ActivateEffect()
    {
        base.ActivateEffect();
        playerStats.OnDamageReceived += ReflectDamageAndEnemy;
    }

    public override void DeactivateEffect()
    {
    }
}
```

```

        playerStats.damageReceivedReductionPercentage -=
damageReductionPercentage;
        base.DeactivateEffect();
        playerStats.OnDamageReceived -= ReflectDamageAndEnemy;
    }

    private void ReflectDamageAndEnemy(float damage)
    {
        enemyInflictingDamage = GetNearestEnemy();

        float reflectedDamage = damage *
(damageReflectionPercentage / 100f);

        if (enemyInflictingDamage != null)
        {
            enemyInflictingDamage.TakeDamage(reflectedDamage);
        }
    }

    private CharacterStats GetNearestEnemy()
    {
        Collider2D hit =
Physics2D.OverlapCircle(playerStats.transform.position, 10f,
LayerMask.GetMask("Enemy"));

        if (hit != null && hit.GetComponent<EnemyStats>() !=
null)
        {
            Debug.Log("Found nearest enemy: " +
hit.gameObject.name);
            return hit.GetComponent<EnemyStats>();
        }

        Debug.Log("No nearest enemy found!");
        return null;
    }
}

```

ArcaneResonanceOrbCharm.cs:

```

using JetBrains.Annotations;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ArcaneResonanceOrbCharm : Charm
{
    [SerializeField] private float radiusIncreasePercentage;

    private float originalMaxSize;

    public override void ActivateEffect()
    {

```

```

        originalMaxSize =
SkillManager.instance.chrono.GetMaxSize();

        base.ActivateEffect();
    }

    public override void DeactivateEffect()
    {
        SkillManager.instance.chrono.SetMaxSize(originalMaxSize);

        base.DeactivateEffect();
    }

    public override void CharmEffect()
    {
        float newSize =
SkillManager.instance.chrono.GetMaxSize();
        newSize *= (1 + radiusIncreasePercentage / 100f);
        SkillManager.instance.chrono.SetMaxSize(newSize);
    }
}

```

BersekresRageCharm.cs:

```

using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using UnityEngine;
using Debug = UnityEngine.Debug;

public class BerserkersRageCharm : Charm
{
    [SerializeField] private float damageIncreasePercentage;

    private float originalDamage;
    private float damageAdded;
    private bool effectApplied = false;
    private bool isActive = false;

    private void Update()
    {
        if (isActive)
        {
            if (playerStats.currentHealth <
playerStats.maxHealth.GetValue() && !effectApplied)
            {
                CharmEffect();
            }
            else if (playerStats.currentHealth >=
playerStats.maxHealth.GetValue() && effectApplied)
            {
                playerStats.damage.RemoveModifier(damageAdded);
                effectApplied = false;
            }
        }
    }
}

```

```

    }
}

public override void ActivateEffect()
{
    originalDamage = playerStats.damage.GetValue();
    CharmEffect();

    isActive = true;
}

public override void DeactivateEffect()
{
    if (effectApplied)
    {
        playerStats.damage.RemoveModifier(damageAdded);
        effectApplied = false;
    }

    isActive = false;
}

public override void CharmEffect()
{
    damageAdded = originalDamage * (damageIncreasePercentage
/ 100f);
    playerStats.damage.AddModifier(damageAdded);
    effectApplied = true;
}
}

```

VampiricEmbraceCharm.cs:

```

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

public class VampiricEmbraceCharm : Charm
{
    public float maxHPIncreasePercentage;
    public float vampirismPercentage;
    public float damageReductionPercentage;
    private float hpAdded;

    private FlaskGUI flaskGUI;
    private HealthFlask healthFlask;

    public void Start()
    {
        flaskGUI = FindObjectOfType<FlaskGUI>();
        healthFlask = FindObjectOfType<HealthFlask>();
    }

    public override void ActivateEffect()

```

```

{
    base.ActivateEffect();
    playerStats.onDamageDealt += ApplyVampirism;
    healthFlask.CanUseFlasks(false);
    flaskGUI.Deactivate();
    Debug.Log("HP ADDED:" + hpAdded);
}
public override void DeactivateEffect()
{
    playerStats.onDamageDealt -= ApplyVampirism;

    playerStats.maxHealth.RemoveModifier(hpAdded);
    healthFlask.CanUseFlasks(true);
    flaskGUI.Activate();

    playerStats.damageReceivedReductionPercentage -=
damageReductionPercentage;
    Debug.Log("HP REMOVED:" + hpAdded);
}

public override void CharmEffect()
{
    hpAdded = playerStats.maxHealth.GetValue() *
(maxHPIncreasePercentage / 100f);
    playerStats.maxHealth.AddModifier(hpAdded);
    playerStats.allBars.UpdateBars();

    playerStats.damageReceivedReductionPercentage +=
damageReductionPercentage;
}
private void ApplyVampirism(float damageDealt)
{
    float healAmount = damageDealt * (vampirismPercentage /
100f);
    playerStats.IncreaseHealthBy(healAmount);
    playerStats.allBars.SetHealth(playerStats.currentHealth,
playerStats.maxHealth.GetValue());
}
}

```

Зібності героя

Skill.cs:

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

public class Skill : MonoBehaviour
{
    [SerializeField] protected float cooldown;
    protected float cooldownTimer;
    [SerializeField] private float manaCost;
    public float ManaCost => manaCost;
    protected Player player;

    protected virtual void Update()
    {
        cooldownTimer -= Time.deltaTime;
    }
    protected virtual void Start()
    {
        player = PlayerManager.instance.player;
    }

    public virtual bool CanUseSkill()
    {
        {
            if(cooldownTimer < 0)
            {
                UseSkill();
                cooldownTimer = cooldown;
                return true;
            }
            return false;
        }
    }
    public virtual void UseSkill()
    {
    }

    protected virtual Transform FindClosestEnemy(Transform
_checkTransform)
    {
        Collider2D[] colliders =
Physics2D.OverlapCircleAll(_checkTransform.position, 25);

        float closestDistance = Mathf.Infinity;
        Transform closestEnemy = null;

        foreach (var hit in colliders)
        {
```

```

        if (hit.GetComponent<Enemy>() != null)
        {
            float distanceToEnemy =
Vector2.Distance(_checkTransform.position,
hit.transform.position);

            if (distanceToEnemy < closestDistance)
            {
                closestDistance = distanceToEnemy;
                closestEnemy = hit.transform;
            }
        }
    }

    return closestEnemy;
}
}

```

ChronoSkill.cs:

```

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

public class ChronoSkill : Skill
{
    [SerializeField] private float cloneCooldown;
    [SerializeField] private int amountOfAttacks;
    [SerializeField] private float chronoDuration;
    [Space]
    [SerializeField] private GameObject chronoPrefab;
    [SerializeField] private float maxSize;
    [SerializeField] private float growSpeed;
    [SerializeField] private float shrinkSpeed;

    ChronofistSkillController currentChrono;

    private string saveKey2 = "achivementsSave";
    private SaveData.AchivementsSave data2;
    protected override void Start()
    {
        base.Start();
        data2 =
SaveManager.Load<SaveData.AchivementsSave>(saveKey2);
    }
    public override bool CanUseSkill()
    {
        return base.CanUseSkill();
    }

    public override void UseSkill()
    {

```

```

        base.UseSkill();

        GameObject newChrono = Instantiate(chronoPrefab,
player.transform.position, Quaternion.identity);

        currentChrono =
newChrono.GetComponent<ChronofistSkillController>();

        currentChrono.SetupChrono(maxSize, growSpeed,
shrinkSpeed, amountOfAttacks, cloneCooldown, chronoDuration);
        if (!data2._isOwned[5])
        {
            MainMenu.instance.setAchievementOwned(5);
        }
    }

    protected override void Update()
    {
        base.Update();
    }

    public bool SkillCompleted()
    {
        if (!currentChrono)
            return false;

        if (currentChrono.playerCanExitState)
        {
            currentChrono = null;
            return true;
        }

        return false;
    }

    public void SetMaxSize(float _size)
    {
        maxSize = _size;
    }

    public float GetMaxSize()
    {
        return maxSize;
    }
}

```

CloneSkill.cs:

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.ExceptionServices;
using UnityEngine;

public class CloneSkill : Skill
{

```

```

[Header("Clone info")]
[SerializeField] private GameObject clonePrefab;
[SerializeField] private float cloneDuration;
[Space]
[SerializeField] private bool canAttack;

[SerializeField] private bool creatCloneOnDashStart;
[SerializeField] private bool createCloneOnDashOver;
[SerializeField] private bool canCreateCloneOnCounterAttack;
[Header("Clone can duplicate")]
[SerializeField] private bool canDuplicateClone;
[SerializeField] private float chanceToDuplicate;

public void CreateClone(Transform _clonePosition, Vector3
_offset)
{
    GameObject newClone = Instantiate(clonePrefab);

    newClone.GetComponent<CloneSkillController>().SetupClone(_clonePo
sition, cloneDuration, canAttack, _offset,
FindClosestEnemy(newClone.transform), canDuplicateClone,
chanceToDuplicate, player);
}

public void CreateCloneOnDashStart()
{
    if (creatCloneOnDashStart)
        CreateClone(player.transform, Vector3.zero);
}

public void CreateCloneOnDashOver()
{
    if (createCloneOnDashOver)
        CreateClone(player.transform, Vector3.zero);
}

public void CreateCloneOnCounterAttack(Transform
_enemyTransform)
{
    if (canCreateCloneOnCounterAttack)
        StartCoroutine(CreateCloneWithDelay(_enemyTransform,
new Vector3(2 * player.facingDirection, 0)));
}

private IEnumerator CreateCloneWithDelay(Transform
_trasnform, Vector3 _offset)
{
    yield return new WaitForSeconds(.4f);
    CreateClone(_trasnform, _offset);
}

```

```

    }
    public void CanCreateCloneOnStart(bool canCreateClone)
    {
        creatCloneOnDashStart = canCreateClone;
    }
}

```

ScytheThrowSkill.cs:

```

using System;
using UnityEngine;

public enum ScytheType
{
    Regular,
    Bounce,
    Pierce,
    Spin
}

public class ScytheThrowSkill : Skill
{
    public ScytheType scytheType = ScytheType.Regular;

    [Header("Bounce info")]
    [SerializeField] private int BounceAmount;
    [SerializeField] private float bounceGravity;
    [SerializeField] private float bounceSpeed;

    [Header("Pierce info")]
    [SerializeField] private int pierceAmount;
    [SerializeField] private float pierceGravity;

    [Header("Spin info")]
    [SerializeField] private float hitCooldown = 0.2f;
    [SerializeField] private float maxTravelDistance = 5;
    [SerializeField] private float spinDuration = 2;
    [SerializeField] private float spinGravity = 1;

    [Header("Skill info")]
    [SerializeField] private GameObject scythePrefab;
    [SerializeField] private Vector2 launchForce;
    [SerializeField] private float scytheGravity;
    [SerializeField] private float freezeTimeDuration;
    [SerializeField] private float returnSpeed;

    private Vector2 finalDirection;

    [Header("Aim dots")]
    [SerializeField] private int numberOfDots;
    [SerializeField] private float spaceBetweenDots;
    [SerializeField] private GameObject dotsPrefab;
}

```

```

[SerializeField] private Transform dotsParent;

private GameObject[] dots;

private string saveKey2 = "achivementsSave";
private SaveData.AchivementsSave data2;
protected override void Start()
{
    base.Start();

    GenerateDots();

    SetupGravity();
    data2 =
SaveManager.Load<SaveData.AchivementsSave>(saveKey2);
}

private void SetupGravity()
{
    if (scytheType == ScytheType.Bounce)
        scytheGravity = bounceGravity;
    else if(scytheType == ScytheType.Pierce)
        scytheGravity = pierceGravity;
    else if(scytheType == ScytheType.Spin)
        scytheGravity = spinGravity;
}

protected override void Update()
{
    if
(Input.GetKeyUp(InputSettings.Instance.GetKeyForAction(InputSetti
ngs.Instance.Button_ult)))
        finalDirection = new
Vector2(AimDirection().normalized.x * launchForce.x,
AimDirection().normalized.y * launchForce.y);

    if
(Input.GetKey(InputSettings.Instance.GetKeyForAction(InputSetting
s.Instance.Button_ult)))
    {
        for (int i = 0; i < dots.Length; i++)
        {
            dots[i].transform.position = DotsPosition(i *
spaceBetweenDots);
        }
    }
}

public void CreateScythe()
{
    if (!data2._isOwned[5])
    {
        MainMenu.instance.setAchivementOwned(5);
    }
}

```

```

    }

    GameObject newScythe = Instantiate(scythePrefab,
player.transform.position, transform.rotation);
    ScytheSkillController newScytheScript =
newScythe.GetComponent<ScytheSkillController>();

    if (scytheType == ScytheType.Bounce)
        newScytheScript.SetupBounce(true, BounceAmount,
bounceSpeed);
    else if (scytheType == ScytheType.Pierce)
        newScytheScript.SetupPierce(pierceAmount);
    else if (scytheType == ScytheType.Spin)
        newScytheScript.SetupSpin(true, maxTravelDistance,
spinDuration, hitCooldown);

    newScytheScript.SetupScythe(finalDirection,
scytheGravity, player, freezeTimeDuration, returnSpeed);

    player.AssingNewScythe(newScythe);

    DotsActive(false);
}
#region Aim region
public Vector2 AimDirection()
{
    Vector2 playerPosition = player.transform.position;
    Vector2 mousePosition =
Camera.main.ScreenToWorldPoint(Input.mousePosition);
    Vector2 direction = mousePosition - playerPosition;

    return direction;
}
public void DotsActive(bool _isActive)
{
    for (int i = 0; i < dots.Length; i++)
    {
        dots[i].SetActive(_isActive);
    }
}

private void GenerateDots()
{
    dots = new GameObject[numberOfDots];
    for (int i = 0; i < numberOfDots; i++)
    {
        dots[i] = Instantiate(dotsPrefab,
player.transform.position, Quaternion.identity, dotsParent);
        dots[i].SetActive(false);
    }
}

private Vector2 DotsPosition(float t)

```

```
{
    Vector2 position = (Vector2)player.transform.position +
new Vector2(
    AimDirection().normalized.x * launchForce.x,
    AimDirection().normalized.y * launchForce.y) * t + .5f
* (Physics2D.gravity * scytheGravity) * (t * t);

    return position;

}
#endregion
}
```