Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента	*Семенова Микити Сергійовича*
	(ПІБ)
академічної групи	*122М-22-4*
	(шифр)
спеціальності	*122 Комп'ютерні науки*
	(код і назва спеціальності)
освітньої програми	*«Комп'ютерні науки»*
	(назва освітньої програми)
на тему:	*Дослідження ігрових можливостей гравців різного віку та розробка ігри з урахуванням вікових особливостей*

Семенов Микита Сергійович

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
розділів кваліфікаційної роботи	*доц. Приходченко С.Д.*			
спеціальний	*доц. Приходченко С.Д.*			
Рецензент				
Нормоконтролер	*доц. Гуліна І.Г.*			

Дніпро
2023

Міністерство освіти і науки України
Національний технічний університет «Дніпровська політехніка»

ЗАТВЕРДЖЕНО:

Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

_____ М. О. Алексєєв
(підпис) (прізвище, ініціали)

« » ___ 20 23 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності ____ *122 Комп'ютерні науки*
(код і назва спеціальності)

студенту *122м-22-4* *Семенову Микиті Сергійовичу*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи Дослідження ігрових можливостей гравців різного віку та розробка гри з урахуванням вікових особливостей

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 09.10.2023 р. № 1227-с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – ігрові можливості та уподобання гравців різного віку.

Предмет досліджень – методи розробки гри, враховуючи вікові особливості гравців.

Методи дослідження: аналіз вікових груп геймерів та їх вплив на геймдизайн, дослідження попередніх ігрових досвідів різних вікових груп у контексті їх впливу на вибір ігор та їх ефективне сприйняття.

Мета роботи – створення гри, яка оптимально відповідає інтересам та потребам гравців різного віку.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Новизна запропонованих рішень визначається розробкою гри, яка враховує вікові особливості гравців. Цей підхід спрямований на створення геймплейних функцій, графічного оформлення та інших елементів гри, що відповідають інтересам та потребам гравців різних вікових категорій.

Практична цінність результатів полягає у створенні інноваційних рішень для розробки гри, що враховує вікові аспекти.

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Розробка методів спрямованих на адаптацію гри для різних вікових груп користувачів. Це включає аналіз та дослідження інтересів, уподобань та здібностей гравців різного віку для створення оптимальних умов гри для кожної категорії.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз джерел та постановка задачі	08.10.2023 - 30.10.2023
Вибір платформи та необхідних матеріалів для розробки	01.11.2023 - 15.11.2023
Розробка та тестування гри з урахуванням можливостей різних вікових категорій	15.11.2023 - 01.12.2023

Завдання видав _____ *Приходченко С.Д.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Семенов М.С.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *09.10.2023 р.*

Термін подання кваліфікаційної роботи до ЕК *20.01.2024*

РЕФЕРАТ

Пояснювальна записка: 105с., 109 рис., 1 табл., 4 додатка, 23 джерел.

Об’єкт досліджень – ігрові можливості та уподобання гравців різного віку.

Предмет досліджень – методи розробки гри, враховуючи вікові особливості гравців.

Мета роботи – створення гри, яка оптимально відповідає інтересам та потребам гравців різного віку.

Методи дослідження: аналіз вікових груп геймерів та їх вплив на геймдизайн, дослідження попередніх ігрових досвідів різних вікових груп у контексті їх впливу на вибір ігор та їх ефективне сприйняття.

Новизна запропонованих рішень визначається розробкою гри, яка враховує вікові особливості гравців. Цей підхід спрямований на створення геймплейних функцій, графічного оформлення та інших елементів гри, що відповідають інтересам та потребам гравців різних вікових категорій.

Практична цінність результатів полягає у створенні інноваційних рішень для розробки гри, що враховує вікові аспекти

Список ключових слів: C#, UNITY, ГЕЙМІНГ, ВІКОВІ ГРУПИ, ЖАНРИ ІГРОВИХ ДОДАТКІВ, КОМФОРТНИЙ ГЕЙМПЛЕЙ, ІГРОВИЙ ДОДАТОК, ГРА

ABSTRACT

Explanatory note: 105 pages., 109 figures., 1 tables., 4 appendices, 23 sources.

The object of research is gaming capabilities and preferences of players of different ages.

The subject of research is methods of game development, taking into account the age characteristics of players.

Research methods: analysis of age groups of gamers and their influence on game design, research of previous gaming experiences of different age groups in the context of their influence on the choice of games and their effective perception.

The goal of the work is to create a game that optimally meets the interests and needs of players of all ages.

The novelty of the proposed solutions is determined by the development of the game, which takes into account the age characteristics of the players. This approach is aimed at creating gameplay features, graphics and other game elements that meet the interests and needs of players of different age categories.

The practical value of the results lies in the creation of innovative solutions for game development that takes into account age aspects

List of key words: C#, UNITY, GAMING, AGE GROUPS, GAMING APP GENRES, COMFORTABLE GAMEPLAY, GAMING APP, GAME

ЗМІСТ

ВСТУП

У сучасному світі ігрова індустрія є однією з динамічніших галузей, яка надихає мільйони людей по всьому світу. З ростом популярності комп'ютерних ігор усе більше геймерів різних вікових категорій шукають захопливий, цікавий та відповідний своїм потребам інтерактивний досвід.

Розробка ігрового продукту, спрямованого на різноманітні смаки геймерів, передбачає створення захоплюючого та естетично привабливого ігрового середовища. Завдяки зростаючому впливу інноваційних технологій, геймери все більше цінують високу якість графіки, захоплюючий геймплей та можливості для індивідуалізації ігрового процесу.

Проект, присвячений розробці гри, націлений на задоволення потреб та очікувань різних груп користувачів. Основні аспекти включають в себе створення захоплюючих сценаріїв, інноваційні механіки гри, які дають гравцям відчуття унікальності та емоційність ігрового процесу. Важливими факторами успіху є не лише геймплей, а й візуальне оформлення, аудіо-супровід та можливості адаптації до потреб різних географічних та вікових груп користувачів.

Цей проект надає можливість дослідження методів та стратегій розробки ігор, а також використання передових технологій для створення ігрового вмісту, що задовольняє різноманітні потреби геймерів, та є спробою реалізувати концепції, що привертають увагу та створюють цікавий інтерактивний світ для геймерської аудиторії.

РОЗДІЛ 1
АНАЛІЗ ВІКОВИХ ТА ІНДИВІДУАЛЬНИХ ВЛАСТИВОСТЕЙ ГРАВЦІВ ДЛЯ ОПТИМІЗАЦІЇ ГЕЙМПЛЕЮ

1.1 Загальні відомості

В сучасному світі відеоігри виявляються важливою складовою культурного та розважального простору. Статистика говорить нам[1], що на 2023 рік світова кількість геймерів сягає вражаючої позначки у 3,24 мільярди осіб. Прогнозується, що ця цифра продовжить зростати, досягаючи 3,32 мільярди гравців у 2024 році. Вартість відеоігрового ринку сягає $197,11 мільярдів у 2023 році, що свідчить про його значущий внесок у глобальну економіку. Досить вражаюче, що понад половини гравців підписані на хоча б один сервіс із підтримкою гри.

Середній вік гравця становить 35 років, що демонструє велике розмаїття гравців у різних вікових категоріях. Європа є важливим регіоном у світовому геймінгу, де понад половина населення віддає перевагу відеоіграм. Значна кількість гравців – понад 1,8 мільярдів осіб – використовують ПК для гри, що свідчить про популярність комп'ютерного геймінгу у світі.

Також у таблиці 1.1 зазначені показники дорослих гравців у США за період з 2018 по 2022 рік, а також співвідношення цього числа до загальної кількості гравців у США.

Важливо відзначити, що демографічний склад гравців різноманітний. Приблизно 48% гравців у США – жінки, що свідчить про рівноправність у гёмінгу між чоловіками та жінками. Азія є найбільшим регіоном світу з 1,6 мільярдами гравців. Розподіл гравців за віком показує, що середній вік американського гравця становить 33 роки, у європейців – 31,3 року, а в австралійців – 34 роки. Що стосується професійного кіберспорту, то середній вік його учасників становить 24 роки, але професійні гравці в гру "Overwatch" часто завершують[2] кар'єру уже у 23 роки.

Таблиця 1.1 – Показники дорослих гравців у США

Рік	Кількість дорослих гравців, млн	Порівняно з усіма гравцями, %
2018	125.1	70
2019	165.91	75
2020	169.06	79
2021	181.6	80
2022	163.3	76

Майже половина гравців у США та Європі – жінки. Щодо платформ, на яких грають гравці у світі, то налічується 1,8 мільярда ПК-гравців, 151,4 мільйонів власників консолі PlayStation та 63,7 мільйонів гравців Xbox.

1.2 Типи гравців та їх вплив на геймплей

Зрозуміти різноманітність типів гравців є важливо складовою розробки відмінної гри. Кожен тим гравця має свої унікальні вподобання, що варіюються від жанрів до стилю гри[3].

Ось деякі основні категорії:

1. Казуальні гравці;
2. Спортивні гравці;
3. Пригодницькі гравці;
4. Співробітники;
5. Експлоратори;
6. Tryhards;
7. Хардкорні гравці.

Казуальні гравці – це ті, хто обирає грати в ігри в розрізі вільного часу або для розваги, не приділяючи грі багато часу або уваги. Вони можуть віддають перевагу простим ігровим механікам, легкому навчанню та швидкому

отриманню задоволення від гри. Казуальні гравці можуть вибирати аркади, головоломки або прості симулятори.

Спортивні гравці віддають перевагу змаганню та вдосконаленню навичок. Вони часто грають в ігри, де можна конкурувати з іншими гравцями, надаючи перевагу онлайн-іграм або многокористувацьким режимам. Спортивні гравці можуть вибирати файтинги, спортивні симулятори чи онлайн-гонки.

Пригодницькі гравці шукають захоплюючі сюжети та зацікавлюються глибокими історіями в іграх. Вони часто вибирають ігри, які пропонують широкий світ для дослідження та можливість взаємодіяти з навколишнім середовищем. Пригодницькі гравці можуть обирати екшн-пригоди, RPG чи ігри з відкритим світом.

Співробітники насолоджуються грою, де спільна гра команди дуже важлива. Вони шукають можливість грати разом з друзями чи іншими гравцями для досягнення спільних цілей. Співробітники можуть вибирати кооперативні ігри, многокористувацькі онлайн-гри чи ігри з режимами співпраці.

Експлоратори насолоджуються дослідженням великих, деталізованих світів і відкриттям нових можливостей. Вони шукають ігри, де важливо відкривати нові локації, вивчати невідомі території та збирати різноманітні об'єкти. Експлоратори можуть вибирати гри з відкритим світом, пригодницькі ігри або симулятори.

Tryhards - це гравці, які завжди намагаються досягти максимальних результатів у грі. Вони приділяють велику увагу оптимізації своєї гри, вивчають стратегії та техніки, щоб стати кращими у своїй грі. Їх головна мета - перемога та виявлення найвищого рівня вмінь[4].

Хардкорні геймери - це ті, хто вкладає надзвичайно багато часу і зусиль у гру. Вони зацікавлені у глибокому дослідженні гри, вивчаючи її механіку, аналізуючи стратегії та докладаючи зусиль для досягнення високого рівня майстерності. Ці гравці можуть брати участь у змаганнях, спільнотах та інших аспектах геймінгової культури, прагнучи стати експертами у своїй грі.

1.3 Популярні жанри серед гравців

Геймінгова індустрія пропонує неймовірну різноманітність жанрів, що виходять за рамки стандартних шаблонів. Кожен жанр відзначається власним неповторним підходом до геймплею та вміщує в собі велику кількість субжанрів[5]. Ця многогранність створює унікальні можливості для задоволення вподобань гравців різних вікових та інтересів груп. Розподіл популярності жанрів серед гравців на 2023 рік[6] демонструється на рисунку 1.1.

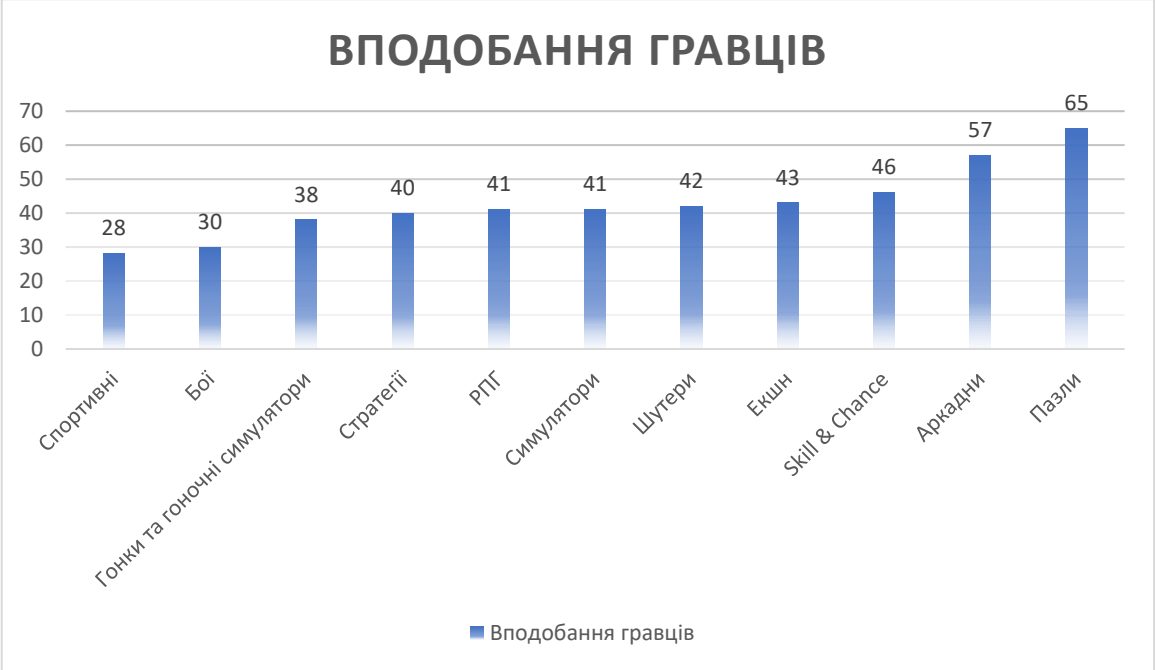

Рисунок 1.1 Популярні жанри станом на 2023 рік

Поглиблене дослідження різноманітних жанрів у геймінгу має важливе значення для кращого розуміння їхнього впливу та особливостей. Згідно з розподілом популярності серед гравців, основні жанри включають[7]:

– Екшн

Жанр екшн є одним із найдинамічніших та найбільш популярних серед гравців. Він визначається своєю активністю та інтенсивністю геймплею, де гравцеві потрібно швидко реагувати на події в грі. Часті елементи включають стрілянину, бійки та швидкі та енергійні події, які створюють атмосферу

адреналіну та виклику. Гравці, які шукають постійну динаміку та готові до найрізноманітніших викликів, часто вдаються до цього жанру. Гарним прикладом гри у цьому жанрі є “Devil May Cry”, на Рисунку 1.2 демонструється геймплей цієї гри.

Рисунок 1.2 Геймплей “Devil May Cry”

– Шутер

Жанр шутер вимагає від гравця вміння ефективно маневрувати та стріляти, найчастіше з різних видів вогнепальної зброї. Від уміння точно прицілюватися та уникати снарядів супротивника залежить успіх у грі. Цей жанр може бути представлений у вигляді військових конфліктів, науково-фантастичних битв чи арен для бою, кожен з яких надає власний унікальний досвід гравцям. Гарним прикладом гри у цьому жанрі є “Call of Duty”, на Рисунку 1.3 демонструється геймплей цієї гри.

Рисунок 1.3 Геймплей “Call of Duty”

– Симулятор

Жанр симулятор надає гравцям можливість відчути себе в ролі різних персонажів чи професіоналів, імітуючи різні аспекти реального життя. Головна мета симулятора - створити найбільш автентичне та вірогідне відтворення реальності. Це може включати водіння різних видів транспорту, польоти на літаках, фермерську роботу та багато інше. Від докладності та реалізму симуляції залежить задоволення гравця від гри. Гарним прикладом гри у цьому жанрі є “Microsoft Flight Simulator”, на Рисунку 1.4 демонструється геймплей цієї гри.

Рисунок 1.4 Геймплей “Microsoft Flight Simulator”

– РПГ

Жанр рольових ігор надає можливість гравцеві втілитися у вигаданого персонажа та впливати на хід подій у віртуальному світі. Однією з ключових особливостей є розвиток персонажа, який може здійснювати різні вибори та вирішення. Гравець має можливість формувати власну унікальну історію та впливати на хід подій у відповідності до своїх виборів. Жанр RPG відкриває широкі можливості для іммерсивного ігрового досвіду. Гарним прикладом гри у цьому жанрі є “The Witcher 3: Wild Hunt”, на Рисунку 1.5 демонструється геймплей цієї гри.

Рисунок 1.5 Геймплей “The Witcher 3: Wild Hunt”

– Стратегія

Жанр стратегія вимагає від гравця глибокого аналізу, планування та вирішення стратегічних завдань. Гравець повинен управляти різними аспектами гри, враховуючи поточну ситуацію та передбачаючи можливі наслідки своїх дій. Цей жанр особливо приваблює тих, хто вміє довго грати та приймати складні рішення в умовах великої відповідальності. Гарним прикладом гри у цьому жанрі є “Civilization VI”, на Рисунку 1.6 демонструється геймплей цієї гри.

Рисунок 1.6 Геймплей “Civilization VI”

– Skill & Chance

Жанр Skill & Chance поєднує в собі елементи випадку та вміння гравця. Це може включати в себе елементи стратегії та тактики, а також елементи випадковості, які додають грі несподіваності та варіативності. Гравець повинен розвивати власні навички та вміти адаптуватися до непередбачуваних обставин. Гарним прикладом гри у цьому жанрі є “Poker Online”, на Рисунку 1.7 демонструється геймплей цієї гри.

Рисунок 1.7 Геймплей “Poker Online”

– Аркада

Жанр аркади відзначається швидкістю, реакцією та простотою геймплею. Гравцеві пропонується проста, але захоплююча мета, яку необхідно досягти шляхом швидкого реагування та вправного управління. Жанр аркади найчастіше реалізується у формі коротких, інтенсивних сесій гри. Гарним прикладом гри у цьому жанрі є “Pac-Man”, на Рисунку 1.8 демонструється геймплей цієї гри.

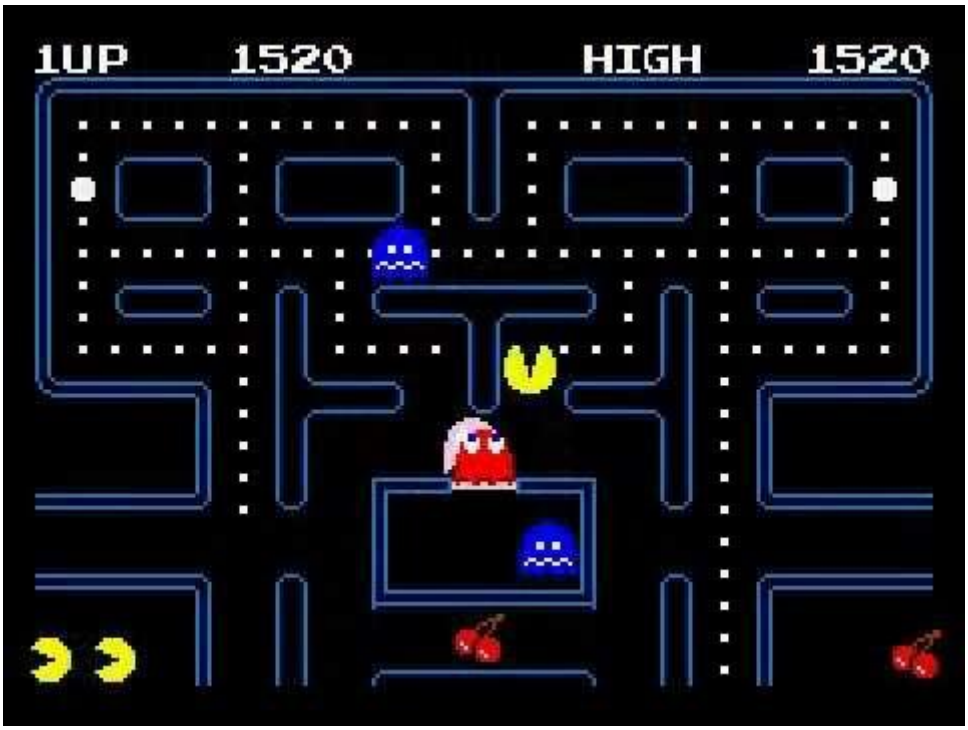

Рисунок 1.8 Геймплей “Pac-Man”

– Пазли

Жанр пазли вимагає від гравця логічного мислення та розв'язання складних завдань. Головною метою є розгадати головоломки чи вирішити завдання, які часто базуються на логіці, спостережливості та кмітливості. Гравці, які віддають перевагу аналітичному мисленню, зазвичай вибирають ігри цього жанру. Гарним прикладом гри у цьому жанрі є “Tetris”, на Рисунку 1.9 демонструється геймплей цієї гри.

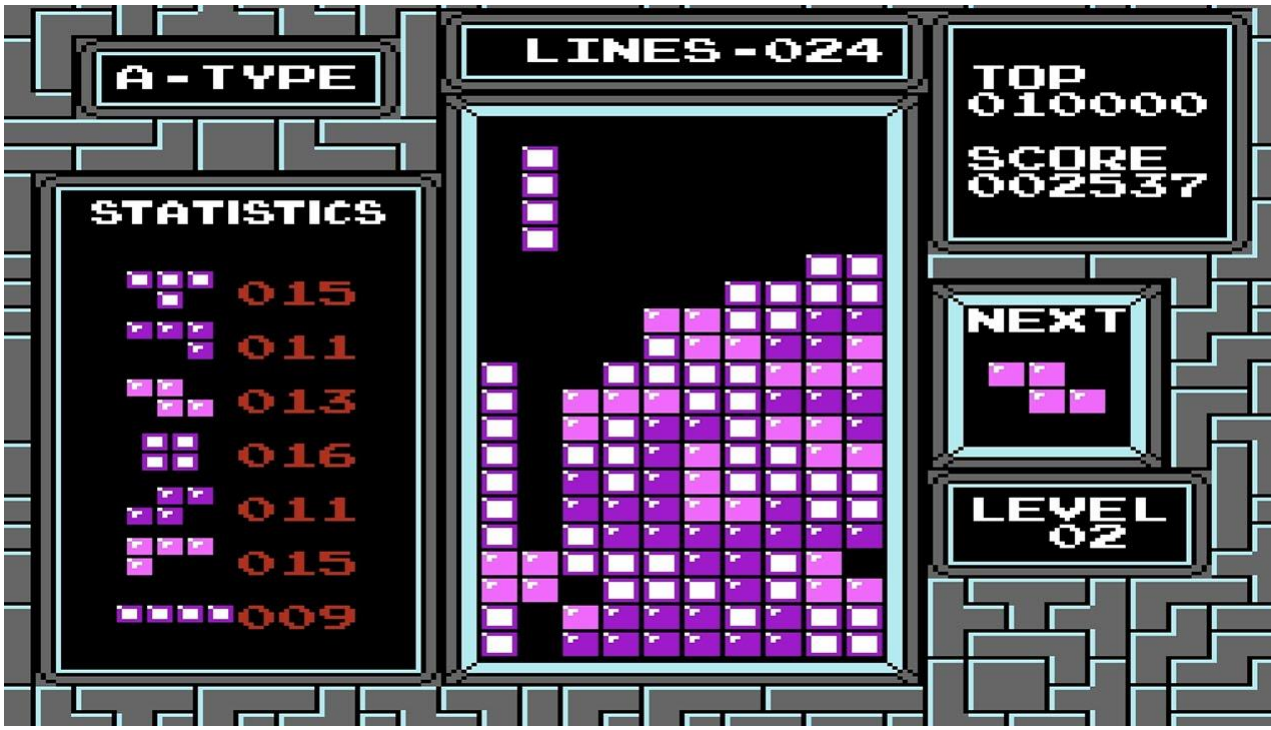

Рисунок 1.9 Геймплей “Tetris”

– Бої

Бойові ігри надають можливість гравцеві випробувати свої навички в різних бойових сценаріях. Гравець стає учасником битв, де вирішальними є навички управління персонажем, правильний вибір тактики та точність у виконанні бойових прийомів. Гарним прикладом гри у цьому жанрі є “Street Fighter”, на Рисунку 1.10 демонструється геймплей цієї гри.

Рисунок 1.10 Геймплей “Street Fighter”

– Гонки

Гравці, які захоплюються швидкістю та адреналіном, часто обирають гонки. Цей жанр дозволяє змагатися на віртуальних трасах, відчуваючи неймовірну швидкість та контролюючи різноманітні види транспорту. Гонки можуть включати як реалістичні симуляції, так і аркадні стилизовані гонки, кожна з яких надає свій унікальний досвід гри. Гарним прикладом гри у цьому жанрі є “Forza Horizon”, на Рисунку 1.11 демонструється геймплей цієї гри.

Рисунок 1.11 Геймплей “Forza Horizon”

– Спортивні

Спортивні ігри надають гравцям можливість відчути себе на справжньому спортивному полі. Цей жанр включає в себе різноманітні види спорту, такі як футбол, баскетбол, теніс, гольф та інші. Гравці можуть вибирати команди, управляти гравцями та брати участь у реалістичних симуляціях або аркадних інтерпретаціях різних видів спорту. Гарним прикладом гри у цьому жанрі є “FIFA”, на Рисунку 1.12 демонструється геймплей цієї гри.

Рисунок 1.12 Геймплей “FIFA”

1.4 Аналіз вікових категорій гравців та їх вподобання

Геймінгова громада представлена різноманітним спектром вікових груп, кожна з яких вносить свій внесок у розвиток і ромаїття індустрії відеоігор. Згідно статистики на 2023 рік[8], розподіл гравців за віком та їх вподобання виглядає наступним чином:

– Діти та підлітки (до 18 років) становлять приблизно 20% гравців. Вони часто обирають ігри з веселим та аркадним геймплеєм, а також інтерактивні пригоди. Ця група цінує можливість розвивати навички та відкривати для себе нові можливості у віртуальному світі.

– Молодь (18-25 років) складає близько 30% аудиторії геймерів. Ця група активно зацікавлена у іграх з екшн-компонентом, онлайн іграх та рольових іграх. Молодь цінує можливість відчути адреналін та випробувати свої сили у віртуальних пригодах.

– Люди середнього віку (25-35 років) охоплюють близько 15% гравців. Ця група часто обирає ігри з розгорнутим сюжетом, стратегіями та іграми на логіку. Вони цінують можливість приймати важливі рішення та вирішувати складні завдання.

– Споживачі старшого віку (35+ років) складають приблизно 10% аудиторії. Вони, як правило, шукають ігри, які дозволяють розслабитися та розважитися, такі як карточні ігри, головоломки та симулятори. Ця група цінує можливість провести час з користю та насолодитися спокоєм у віртуальному середовищі.

Враховуючи цей аналіз, для оптимізації геймплею та забезпечення приємного досвіду для кожної вікової групи гравців, було обрано аркадний жанр гри з додатковим жанром у вигляді пазлів. Ці формати гри надають можливість гравцям різних вікових категорій насолоджуватися грою в комфортних умовах.

1.6 Порівняння ігрових механік у різних жанрах аркад

Для ефективної розробки гри, яка найкраще відповідає потребам гравців, необхідно провести аналіз вже існуючих популярних ігор у жанрі аркади. Кожен жанр аркад відзначається унікальними особливостями та основними механіками, що впливають на взаємодію гравця з грою та визначають її загальний геймплей. Цей аналіз надасть цінний контекст та дозволить визначити оптимальні рішення для створення найбільш комфортного геймплею. Для розгляду було прийнято наступні варіанти ігор у жанрі аркади:

Гра “Vampire Survivors” пропонує гравцям виживання на арені, геймплей цієї гри показано на рисунку 1.13. У цій грі, гравці уособлюють вампірів, які повинні відбивати безперервні атаки і залишатися в живих.

Рисунок 1.13 Геймплей “Vampire Survivors”

У грі “Vampire Survivors” важливим аспектом є вибір оптимальної стратегії та уникання атак супротивників. Гравцям надається можливість обирати серед різноманітних персонажів які показані на рисунку 1.14. Деяких персонажів потрібно відкривати, а також між ігровими сесіями можна прокачувати деякі характеристики та параметри які додаються для кожного з персонажів що показано на рисунку 1.15. Кожен персонаж володіє власними унікальними початковими навичками та характеристиками.

Рисунок 1.14 Вибір персонажів

Рисунок 1.15 Прокачка навичок та характеристик персонажів

Гра відзначається динамічним геймплеєм та надає можливість випробувати різні стратегії виживання. Гравець може отримати насолоду від

швидкого та захоплюючого геймплею, де кожне рішення має вагоме значення. Головний герой збирає різнокольорові кристали, які додають йому досвід. Коли герой піднімає рівень, гравцеві надається можливість вибрати додатковий предмет для свого персонажа. Крім того, герой отримує гроші за вбивство супротивників. У грі працює таймер: після відповідних ключових точок у часі хвилі супротивників стають все важчі, що дозволяє гравцеві перевірити свої сили та ефективність своїх виборів у предметах.

“Celeste” являє собою аркадний платформер, який відзначається високим рівнем складності та цікавим геймплеєм що показано на рисунку 1.16. Найбільш важливою особливістю є інтуїтивна система управління та точність рухів, яка є вирішальною для успішного проходження рівнів, це пов’язано з тим, що у гравця є лише одна спроба від початку екрану до кінця пройти рівень без шкоди. У випадку отримання героєм шкоди, рівень починається з контрольної точки.

Рисунок 1.16 Геймплей “Celeste”

Також, “Celeste” вирізняється особливим візуальним стилем, цікаві анімаційні рішення та ефективне використання кольорів для передачі настрою і атмосфери гри що можна побачити на рисунку 1.17

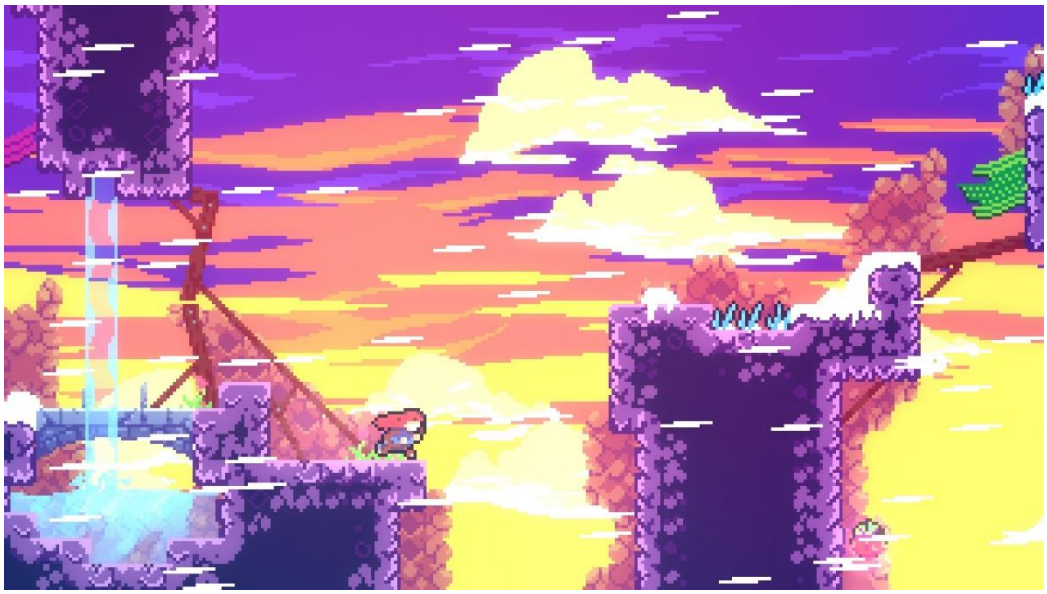

Рисунок 1.17 Використання кольорів для передачі настрою

На цьому рисунку демонструються теплі кольори, які відображають зміну атмосфери гри протягом подорожі гравця. Починаючи з нічних сцен та холодних відтінків на початку подорожі, кольорова гама поступово переходить у тепліші та більш різноманітні відтінки, особливо, коли герой піднімається все вище на гору. Цей ефект вдало підкреслює динаміку гри та перетворюється у важливий елемент візуального наративу.

Гравцям також надається можливість збирати різні предмети, які розсіяні по всіх її розділах, приклад такого предмету демонструється на рисунку 1.18.

Рисунок 1.18 Приклад колекційних предметів у “Celeste”

Кожен з них має свою унікальну ціль, деякі можна знайти лише в певних частинах гри, надаючи гравцям можливість взяти на себе додатковий виклик.

Гра “Hollow Knight” є представником аркадної пригодної інтерактивної гри, в якій події розгортаються у вигаданому підземному королівстві під назвою Галлонест. Гравець управляє безіменним Лицарем, схожим на комаху, який досліджує глибини підземного світу.

Головною бойовою зброєю Лицаря слугує “Кіготь”, який використовується для бою і для взаємодії з навколишнім середовищем. У більшості областей гри гравець зіштовхується з ворогами та іншими створіннями. Ближній бій передбачає використання “Кігтя” для атаки ворогів з невеликої відстані, рисунок 1.19 демонструє атаки з використанням “Кігтя”. Гравець може навчитися заклинанням, що дозволяє здійснювати удари на відстані, на рисунку 1.20 демонструється використання заклинання. Переможені вороги залишають за собою валюту під назвою “Гео”.

Рисунок 1.19 Ближній бій у “Hollow Knight”

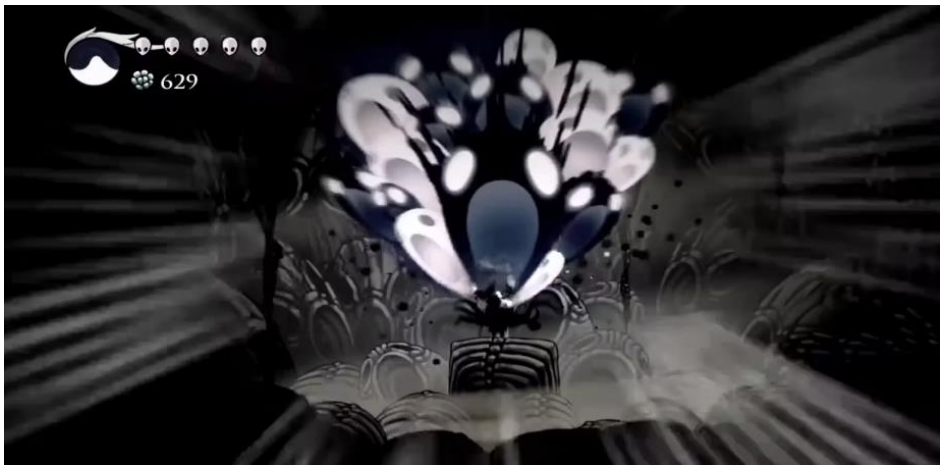

Рисунок 1.20 Приклад заклинання у “Hollow Knight”

Лицар починає з обмеженою кількістю життєвих балів, представлених у вигляді масок, на рисунку 1.21. Коли маски втрачені, Лицар вмирає, і на місці його смерті з’являється ворожий Відбиток. Гравець втрачає всі гео та може зберігати менше душі.

Рисунок 1.21 Маски життєвих балів

У грі існують лавки (рисунок 1.22), які служать як точки збереження та можливість змінити спеціальні навички, екран з навичками демонструється на рисунку 1.23.

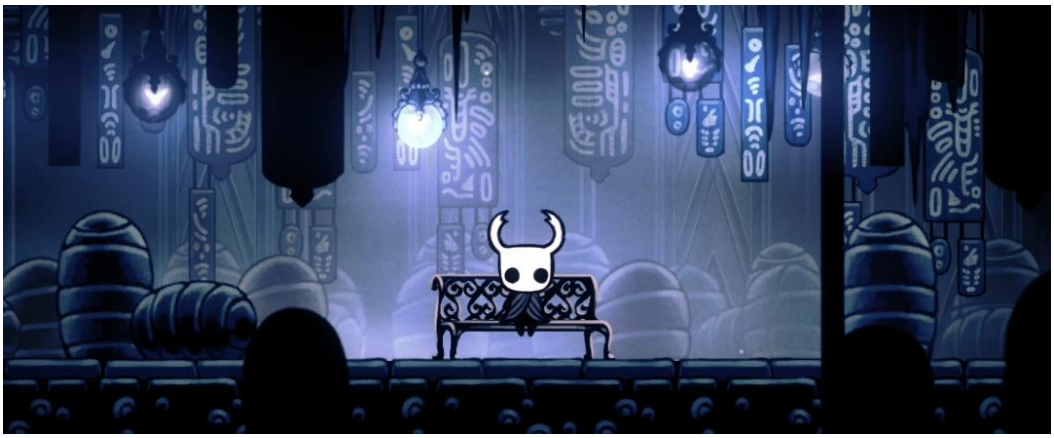

Рисунок 1.22 Лавки для збереження

Рисунок 1.23 Навички у “Hollow Knight”

Спочатку гравець може використовувати душу тільки для відновлення життя, але з розвитком гри гравець розблоковує та збирає атакуючі заклинання, які використовують душу. У грі присутні додаткові посудини для душі, які дозволяють накопичувати більше душі.

Загалом гра вирізняється глибоким геймплеєм, який включає тактичний бій, дослідження інтерактивного світу та вирішення різноманітних завдань для досягнення успіху в грі.

Розглядаючи різноманітні варіанти аркадних ігор, таких як "Vampire Survivors" з фокусом на стратегії і виживанні, "Celeste" з його високим рівнем складності та точністю управління, і "Hollow Knight" з акцентом на дослідженні та тактиці бою, стає очевидним, що кожна з них пропонує унікальні геймплейні особливості.

У світлі цього аналізу стає важливим узяти найкращі аспекти кожної гри і використовувати їх як натхнення для розробки нового проекту. Врахування цих унікальних особливостей дозволить створити захопливий та неповторний геймплей, який приверне увагу та запам'ятається гравцям.

1.7 Розгляд активності гравців та їх сприйняття гри

Для розуміння поведінки гравців та їх сприйняття гри необхідно ретельно проаналізувати загальну активність у кожній з ігор. Цей аналіз надасть більш глибоке розуміння того, як гравці взаємодіють з кожним проектом та які елементи виявилися особливо привабливими для аудиторії.

Для більш детального аналізу гри “Vampire Survivors”, розглянемо важливі показники та аспекти цієї гри, що допоможуть зрозуміти її популярність, геймплей та вплив на гравців. З цими даними буде можливо краще зрозуміти, чому “Vampire Survivors” відзначається серед інших ігор і які особливості цієї гри роблять її такою привабливою для гравців.

Гра “Vampire Survivors” привернула увагу гравців з моменту свого виходу. На 6 січня 2022 року було зафіксовано[9] 8700 гравців, і з тих пір спостерігався стійкий ріст активності. На 20 січня 2022 року ця цифра зросла до 51400 активних гравців, а на 3 лютого 2022 року досягла 71100. Апогей популярності припав на 17 лютого 2022 року, коли одночасно грали 77000 гравців. Проте, після цього спостерігається послаблення ігрової активності. Наприклад, на 17 березня 2022 року активних гравців стало 44200, а на 26 травня 2022 року – лише 29900. Такий спад продовжувався, і на 29 вересня 2022 року активна база гравців

скоротилася до 9600. У наш час, станом на 31 жовтня 2023 року, за останній час в грі було зафіксовано лише 7600 активних гравців. Більш наглядний вигляд активності гравців демонструється на рисунку 1.24.

Рисунок 1.24 Одночасна кількість гравців у “Vampire Survivors”

Також, за даними, отриманими за останні три місяців, можна побачити коливання активності гравців. Наприкінці серпня спостерігався різкий зріст, коли кількість одночасних гравців збільшилася з 8958 до 18734. Проте у наступні тижні ця цифра зменшилася до 13775. Далі, у вересні, відбулася подальша зміна, коли кількість гравців скоротилася до 11583. У жовтні також відбулася відмітна динаміка, коли кількість гравців знову зросла до 12127, але потім знову зменшилася до 9852. Візуалізацію кількості одночасно активних гравців за останні три місяці можна побачити на рисунку 1.25.

Рисунок 1.25 Одночасна кількість гравців за останні 3 місяці

Загальний аналіз активності гравців за останні три місяці свідчить про те, що гра зберігає певний рівень популярності, проте вона також піддатлива коливанням у кількості одночасних гравців. Це може бути пов'язане з різними факторами, такими як оновлення гри, акції та події, які пропонує розробник.

Гра "Vampire Survivors" утримує вражаючий рейтинг на платформі Steam, де 97.40% користувачів висловлюють позитивну думку про неї, і це базується на 212,107 користувацьких відгуків. Це підкреслює високу оцінку, яку отримала гра від гравців. Користувачі відзначають декілька ключових аспектів, які роблять геймплей привабливим. Наприклад, простота управління та абсолютна можливість фокусування на ігровому процесі[10].

Далі, для більш глибокого аналізу гри "Celeste", потрібно також розглянути ключові показники та характеристики цієї гри, що допоможуть розібратися у її популярності та впливу на гравців. Ці дані дадуть можливість краще зрозуміти, чому "Celeste" виділяється серед інших ігор і які особливості роблять її привабливою для широкого кола гравців.

У період з вересня 2018 року до березня 2019 року гра "Celeste" показала зріст популярності. Кількість активних гравців зросла з 210 у вересні 2018 до

1973 у січні 2019[11]. Цей стрімкий ріст може бути спричинений позитивними відгуками та високою оцінкою гравців.

Проте, в наступні місяці весни 2019 року, спостерігається спад активності гравців, що може бути пов’язано з виходом нових ігор на ринку або іншими конкурентними чинниками. Загальний графік гравців з 2018 до 2019 року показано на рисунку 1.26.

Рисунок 1.26 Графік активних гравців з 2018 до 2019 рік

Після цього, у лютому 2023 року, гра “Celeste” знову привернула увагу гравців, що може бути пов’язано з виходом нових ігор на ринку або іншими конкурентними чинниками.

Після цього, у лютому 2023 року, гра “Celeste” знову привернула увагу гравців, показавши стрімкий ріст активності з 1598 у лютому до 4177 у березні, Графічно відображені дані можна побачити на рисунку 1.27.

Рисунок 1.27 Стрімкий ріст гравців “Celeste”

Цей ріст може бути зв’язаний з оновленнями гри, або можливо, з новим контентом який привернув інтерес гравців.

Гра отримала велику кількість позитивних відгуків, що свідчить про її популярність серед аудиторії, зараз гра має 96.06% рейтингу, який базується на 81,543 позитивних рецензій. Узагальнюючи відгуки[12], можна відзначити, що гра отримала високі оцінки від гравців та критиків. Гравці високо оцінюють гру за високий рівень викликів та задоволення від подолання труднощів, а також за глибокий та емоційно насичену історію.

Розглянемо основні аспекти та показники гри "Hollow Knight", які допомагають зрозуміти причини її високої популярності та впливу на гравців. Аналіз цих даних надасть можливість краще оцінити унікальність та привабливість цієї гри для геймерського співтовариства.

За аналізом даних активних гравців у грі “Hollow Knight” можна відзначити вражаючий ріст популярності протягом всього часу існування гри. Навіть зі скромних початків у 2017 році з 2094 одночасно активних гравців, “Hollow Knight” швидко набув визнання та популярності[13]. Початковий графік гравців можна побачити на рисунку 1.28.

Рисунок 1.28 Початкова тенденція гравців у “Hollow Knight”

Період з лютого по липень 2022 року виділяється особливою активністю гравців, коли кількість одночасно активних гравців досягла вражаючих показників, перевищуючи 20 тисяч гравців. Графік активності гравців у період з лютого по липень демонструється на рисунку 1.29.

Рисунок 1.29 Найбільша активність гравців за весь час

Розглянувши дані активних гравців у грі "Hollow Knight", особливо вражає той факт, що популярність гри не лише не втратила свою актуальність після релізу в 2017 році, а й продовжує стрімко рости, зберігаючи стабільну активність гравців і сьогодні. Це свідчить про надзвичайну привабливість та високу якість гри, яка завоювала серця геймерів і змогла утримати їх у своєму віртуальному світі протягом тривалого періоду.

Розглянувши думки користувачів, слід зазначити, що "Hollow Knight" викликає захоплення своєю неймовірною атмосферою та відмінним геймплеєм. Гравці особливо цінують багатий світ гри та якість бойової системи. Зазначається, що вона поєднує в собі різноманітність та значущість дослідження. Деякі, проте, відмічають, що початковий етап гри може виявитися важким через відсутність підказок[14].

1.8 Висновки за розділом

В розділі надано загальний огляд популярних жанрів відеоігор, типів гравців та вікових категорій гравців. Результатом порівняння популярності жанрів серед гравців різних вікових категорій було визначено, що оптимальним вибором для розробки гри є жанр “аркади” з субжанром “пазли”, також після ретельного розгляду кожної гри, виявлено деякі ключові аспекти, які найбільше приваблюють гравців. Вони включають в себе швидкий геймплей, різноманітну та захоплюючу бойову систему, виклик до розвитку навичок, а також простоту управління. Крім того, важливо відзначити наявність різних викликів та босів, які надають гравцеві можливість відчути напруження та випробувати свої сили у різних ситуаціях. Ці аспекти стають ключовими факторами, що впливають на популярність та привабливість кожної з гри.

РОЗДІЛ 2
ПІДГОТОВКА ДО РОЗРОБКИ

2.1 Вибір платформи для розробки

У передовому світі розробки ігор вибір ігрового двигуна має ключове значення для успішної реалізації концепції гри. У даному контексті, аналіз трьох провідних платформ - Godot, Unity та Unreal Engine - є необхідним кроком для обґрунтованого вибору найефективнішого інструменту для розробки проекту. Кожна з них володіє власним спектром можливостей та особливостей, призначених для відповідних категорій розробників. В даному аналізі ми розглянемо кожну платформу у деталях, з метою визначення оптимального вибору для нашого конкретного проекту.

– Godot

Godot – це відмінний вибір для тих, хто шукає ефективну та безкоштовну платформу для розробки відеоігор [15]. Його інтегрована розробка 2D та 3D графіки дозволяє швидко приступити до розробки, зображення інтерфейсу демонструється на рисунку 2.1.

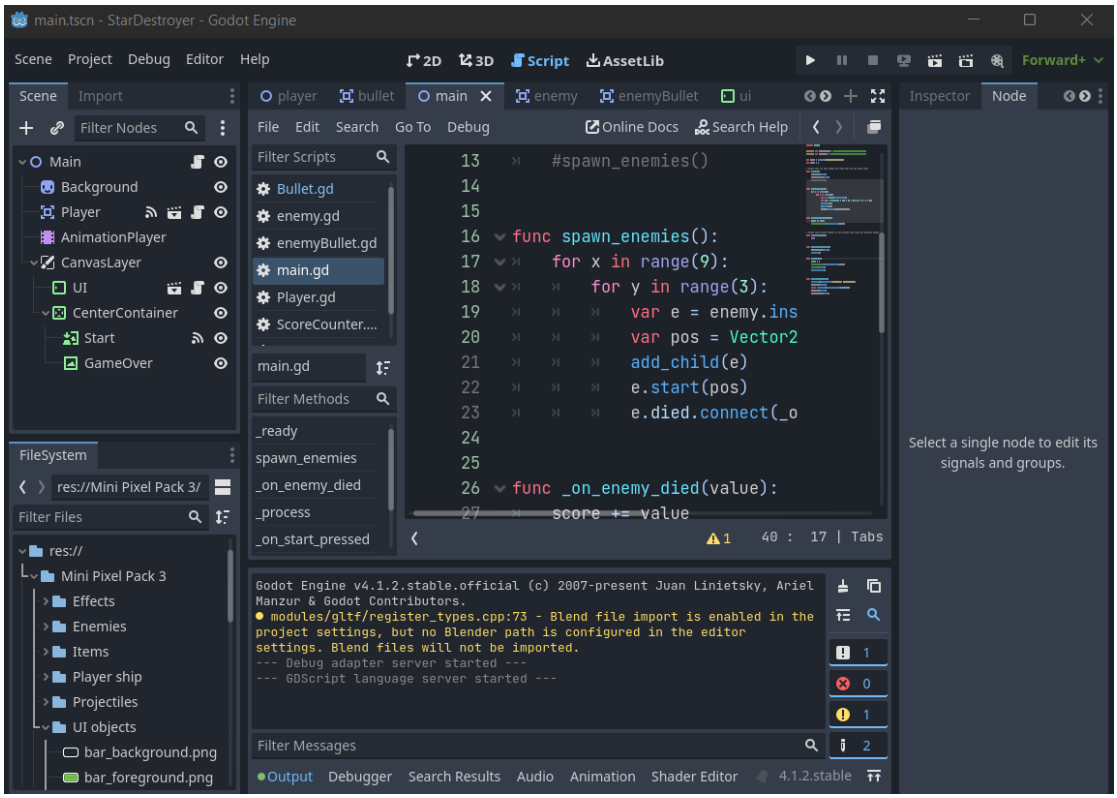

Рисунок 2.1 Інтерфейс Godot

Важливо відзначити, що Godot дозволяє розробляти гри для різних платформ, включаючи мобільні пристрої та ПК.

Ще однією перевагою Godot є його низький поріг входження. Розробники можуть швидко освоїти цей ігровий двигун, що особливо корисно для невеликих студій та незалежних розробників. Однак, у порівнянні з Unity та Unreal Engine, Godot може мати обмежену кількість готових ресурсів та меншу підтримку спільноти.

– Unity

Unity – це досить універсальний ігровий двигун, який надає велику кількість інструментів для розробки різноманітних ігор, інтерфейс платформи демонструється на рисунку 2.2.

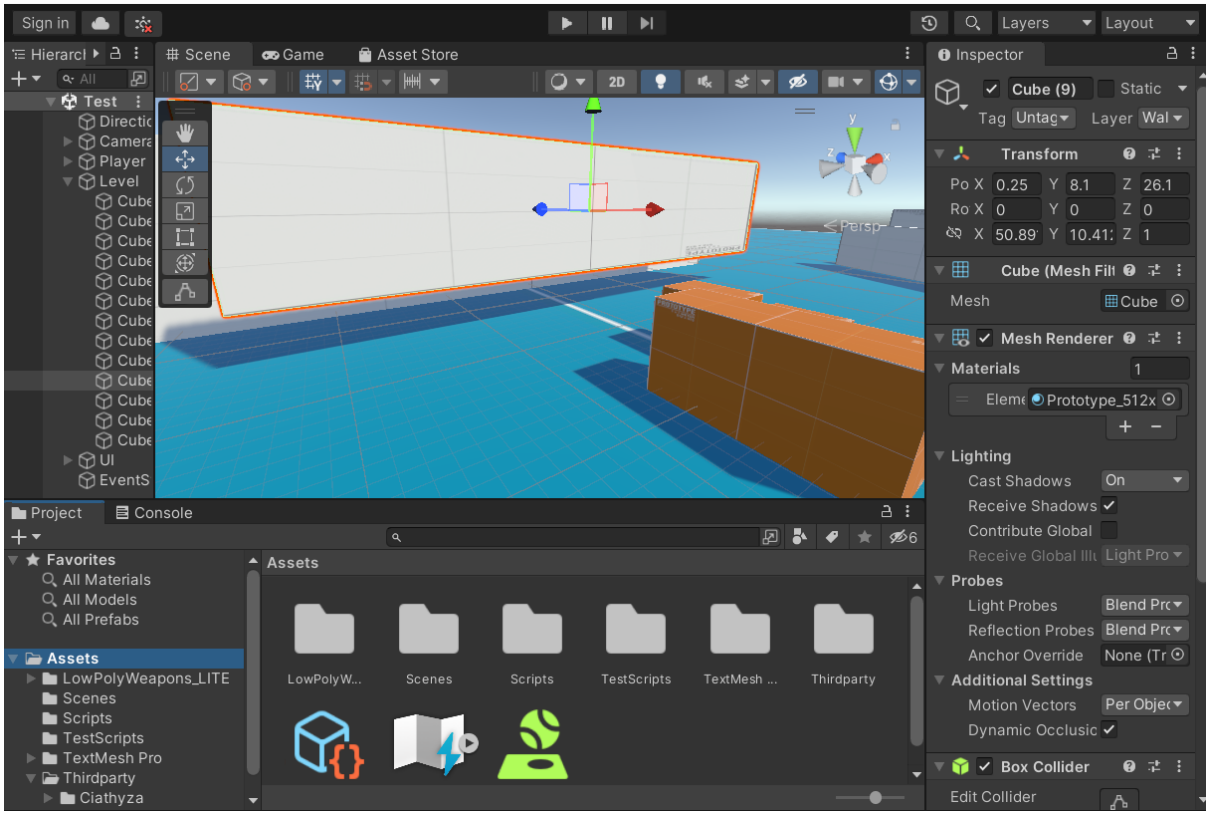

Рисунок 2.2 Інтерфейс Unity

Однією з його сильних сторін є велика кількість готових ресурсів та підтримка активної спільноти, готові ресурси можуть бути як платні так і безкоштовні, що можна побачити на рисунку 2.3.

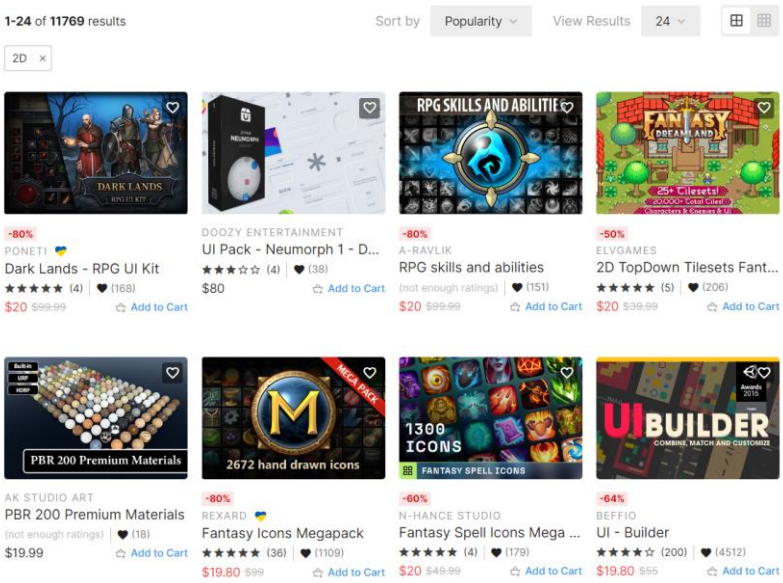

Рисунок 2.3 Asset Store на платформі Unity

Unity надає можливість розробляти ігри для різних платформ, включаючи мобільні пристрої, ПК та консолі[16].

Проте, при роботі зі складними проектами, Unity може генерувати велику кількість коду, що потребує певних навичок та досвіду. Деякі продуктивні функції можуть вимагати встановлення додаткових платних розширень.

– Unreal Engine

Unreal Engine славиться своєю потужною графічною системою та можливостями розробки вражаючих 3D ігор, графічний інтерфейс Unreal Engine демонструється на рисунку 2.4.

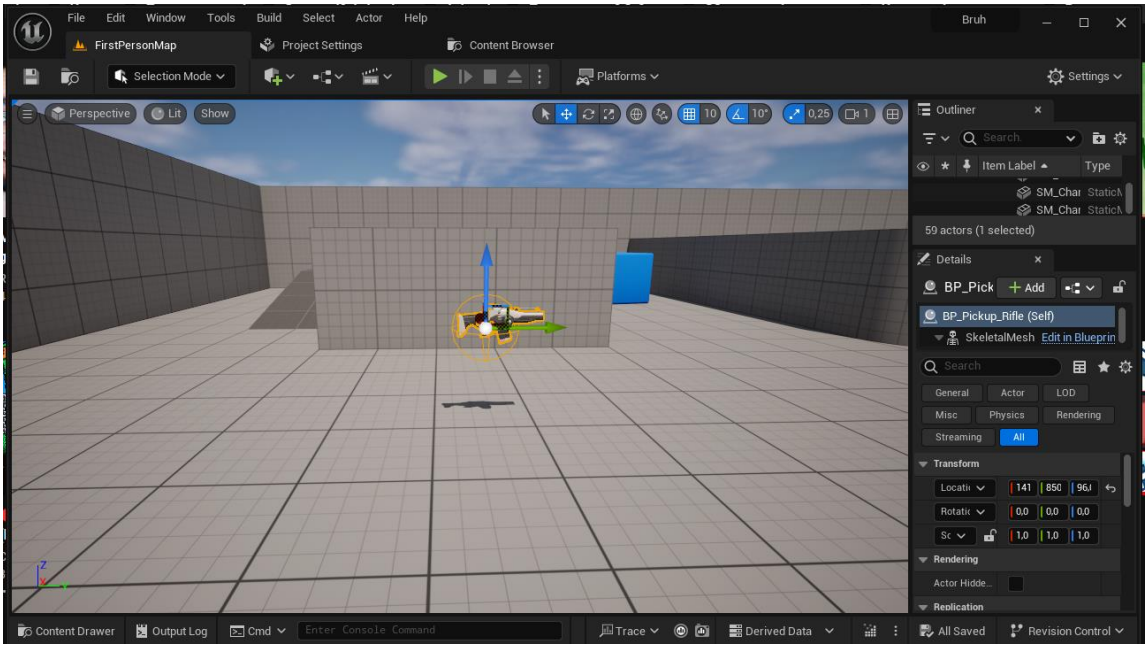

Рисунок 2.4 Графічний інтерфейс Unreal Engine 5

Цей двигун надає широкий спектр готових ресурсів приклад готових ресурсів для Unreal Engine демонструється на рисунку 2.5, та інструментів для створення великих та деталізованих світів[17].

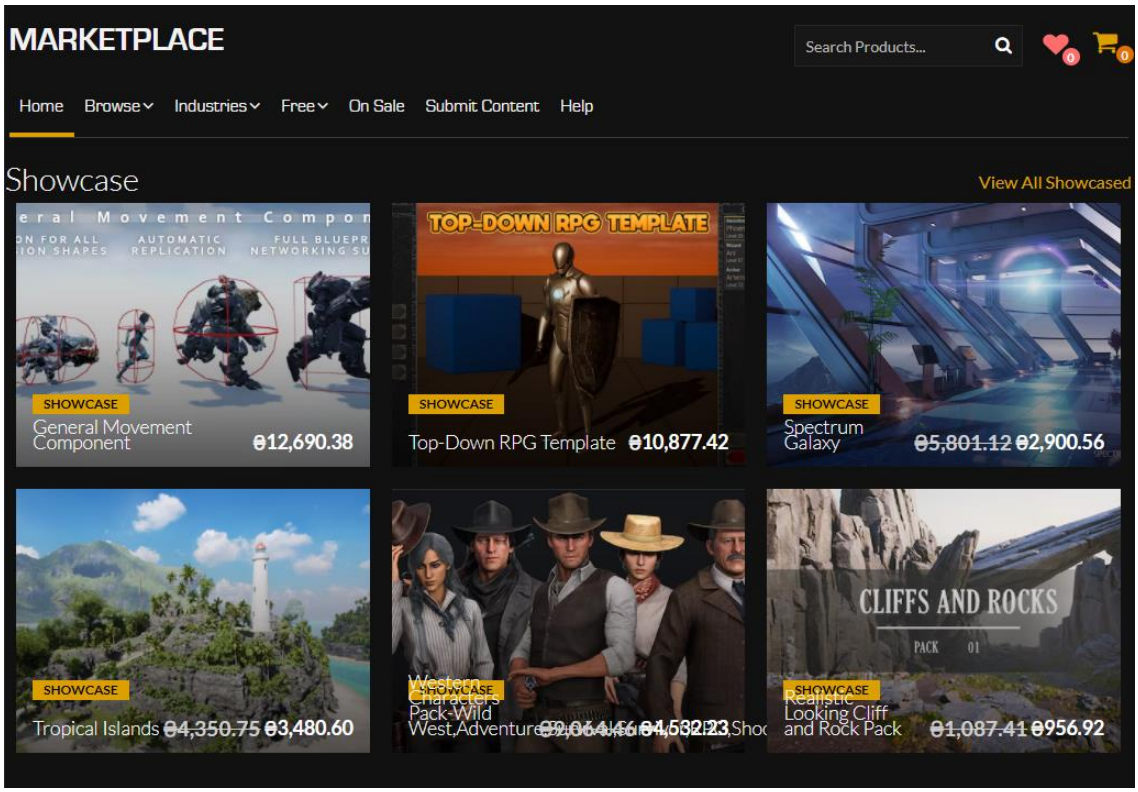

Рисунок 2.5 Готові ресурси у Unreal Engine

Окрім того, Unreal Engine дозволяє використовувати мову програмування C++, що особливо важливо для деяких розробників.

Проте, Unreal Engine може виявитися складним для освоєння, особливо для тих, хто тільки починає свій шлях у розробці ігор. Робота з ним може вимагати більше часу та ресурсів для досягнення конкретних ідей.

Враховуючи всі аспекти, найкращим вибором для розробки нашої гри виявився ігровий двигун Unity. Його потужність, гнучкість та підтримка спільноти дозволять нам ефективно реалізувати ідеї та створити високоякісний продукт. Крім того, широкі можливості для розробки для різних платформ роблять Unity оптимальним вибором для нашого проекту.

2.2 Визначення візуального стилю гри

Візуальний стиль є важливим елементом гри, який визначає її естетичний вигляд та вплив на гравців. Обраний візуальний стиль може суттєво вплинути на

успішність гри та сформувати її унікальну ідентичність в очах аудиторії. В цьому розділі буде проаналізовано процес визначення візуального стилю для нашого проекту, а також важливість вибору та редагування відповідних графічних ресурсів. Додатково, буде оцінено вплив візуального оформлення на тематику гри та інші аспекти проекту. Також, з метою ефективного використання часу, вирішено скористатися готовими ассетами, що надаються користувачами на відповідних платформах[18].

Задля визначення ключового візуального стилю гри, розглянемо кілька популярних ассетів, які можуть виступити основним елементом візуального оформлення проекту.

– Mossy Cavern

Ассет-пак “Mossy Cavern” представляє собою комплекс графічний та анімаційних ресурсів, спроектованих для використання у 2D side-scroller платформерах[19]. Він пропонує розробникам широкий спектр елементів для створення середовища моховитої печери. Приклад готового рівня з використанням ресурсів цього ассет-паку демонструється на рисунку 2.6.

Рисунок 2.6 Готовий рівень з ассетів Mossy Cavern

Головні характеристики ассет-паку включають у себе:

1. Пликтова графіка, готова до автоматичного мапінгу, з кожною плиткою розміром 512x512 пікселів. Це надає можливість швидкого та зручного розташування елементів у грі.

2. Базовий персонаж із руховими анімаціями, що включають “Спокій”, “Хід”, “Стрибок” та “Ривок”. Це дозволяє відтворити рухи персонажа.

3. Декорації, що додають деталізацію та унікальність оточенню: рослини, шипи, невеликі пагорби, невеликі дерева, коріння та великі колони. Ці елементи можуть бути використані для оформлення фону та попереднього плану в грі.

4. Анімації для 13 рослин, що надають руху та життєвості сценам.

5. Два простих ворога: два різновиди слизи з анімацією.

Також цей ассет-пак розповсюджується за ліценцією V0.1 що дозволяє використання ассет-паку як у безкоштовних, так і в комерційних проектах з можливістю модифікації.

– Pixel Game Art

Ассет-пак “Pixel Game Art” являє собою повний набір графічних ресурсів загальнодоступного користування[20]. Він включає всі необхідні компоненти для створення прототипу гри “з коробки”:

1. Паралаксний фон
2. 6 анімацій для гравця
3. 3 різних анімованих ворогів
4. Елементи та анімації ефектів
5. Файли у форматах PNG, PSD, Tiled, атласи
6. 5 типів будинків
7. 4 типів дерев
8. Шаблон проекту для Godot 4

Приклад рівня з використанням цього ассет-паку демонструється на рисунку 2.7.

Рисунок 2.7 Готовий рівень з ассетів “Pixel Game Art”

Всі ресурси ассет-паку “Pixel Game Art” розповсюджуються за умовами ліцензії Creative Commons Zero (CC0). Це дозволяє вільно розповсюджувати, змінювати, адаптувати та використовувати їх у будь-якому середовищі та форматі, навіть у комерційних проектах.

– Pirate Adventure Essentials

Ассет-пак “Pirate Adventure Essentials” містить повний набір ресурсів для створення пригодницької гри про піратів (Гравець, вороги, пастки, об’єкти, предмети, рівні, інтерфейс).

Також з офіційного сайту розповсюдження цього ассет-паку[21], відомо що:

1. У пакеті містяться файли у форматі Aseprite (редаговані)
2. Усі зображення в форматі PNG
3. Анімації надходять окремими зображеннями
4. Тайлсети представлені у вигляді спрайтових аркушів
5. Анімації працюють зі швидкістю 10 FPS або 100 MS

Приклад готового рівня з використанням цього ассет-паку демонструється на рисунку 2.8.

Рисунок 2.8 Готовий рівень з ассетів “Pirate Adventures Essentials”

Цей ассет пак також поширюється за ліцензією Creative Commons Zero (CC0), що дозволяє вільно розповсюджувати, модифікувати та адаптувати контент цього ассет паку у будь-якому середовищі та форматі.

З усіх наведених ассет-паків, Pirate Adventure Essentials виявився найбільш привабливим і комплексним рішенням для розробки гри. Його наповненість всім необхідним для створення пригодницької відеогри, включаючи гравця, ворогів, пастки, об'єкти, предмети, рівні та інтерфейс, робить його чудовим вибором. Тематика ассет-паку дозволяє створити захопливий світ піратських пригод, що привертає увагу гравців. Таким чином, прийнято рішення використовувати Pirate Adventure Essentials для подальшого розвитку гри.

2.3 Звуковий дизайн та музичний супровід

Звук та музика є критичними елементами для створення затягуючого іммерсивного досвіду гри. Вони додають динаміку, підкреслюють ключові події та дозволяють гравцеві відчути атмосферу віртуального світу. Для нашого проекту необхідно ретельно підібрати звукові елементи та музичний супровід, щоб забезпечити максимальну іммерсію та зручність для гравця.

Для загальних елементів гри, важливо врахувати наступне:

1. Звук підбору предметів: Звук, що відтворюється при зборі відповідних предметів, має бути добре впізнаваним та задовольняти очікування гравця.
2. Звук смерті: Цей звук повинен передавати важливість події та надати відчуття наслідків дій гравця.
3. Звук шкоди: Звук, що відтворюється при отриманні гравцем ушкоджень, повинен бути настільки інтенсивним, щоб привернути увагу, але не надто налякати.
4. Звук стрибку: Цей звук повинен передати енергію та динаміку при кожному стрибку гравця.
5. Звук ближнього та дальнього бою: Особливу увагу слід приділити звуковому супроводу бойових дій, щоб вони були насиченими та динамічними, але не надто навіювали агресію.

Також, для елементів користувацького інтерфейсу потрібно врахувати наступне:

1. Звуки діалогів у трьох станах:

– Відкриття діалогу
– Закриття діалогу
– Друк діалогу

Ці звуки мають бути чіткими та легко розпізнаваними, щоб спростити навігацію.

2. Звуки навігації по інтерфейсу:

– Звуки натискання на кнопки та інші звуки навігації допоможуть гравцеві відчути відгук на свої дії та взаємодію з інтерфейсом гри.

А також, не менш важливим є музичний супровід під час гри, фонова музика відіграє ключову роль у створенні атмосфери та підсиленні загального настрою гри. Слід враховувати, що вибір музичного супроводу вимагає врахування синергії з візуальним стилем гри та здатності створити навколишню атмосферу для гравця.

Для досягнення більш виразних та унікальних звукових ефектів гри, було вирішено використовувати утиліту ChipTone. Цей безкоштовний інструмент призначений для генерації звукових ефектів, переважно для використання в іграх, але може бути використаний у будь-яких інших проектах[22]. Окрім того, утиліта пропонує можливості семплювання та створення музичних послідовностей. Для прикладу на рисунку 2.10 демонструється налаштування для звуку отримання шкоди.

Рисунок 2.10 Налаштування для звуку отримання шкоди

Для ефективності вибору фонової музики було вирішено скористатися відомими ресурсами для вільних ресурсів. Один із найбільших репозиторіїв, який надає широкий вибір медіа-елементів для використання в різних проектах, включаючи розробку гри, є OpenGameArt. Цей ресурс визнаний великим співтовариством та надає можливість розробникам знаходити якісну та безкоштовну музику для використання в їхніх творчих твореннях[23]. Приклад музики демонструється на рисунок 2.11.

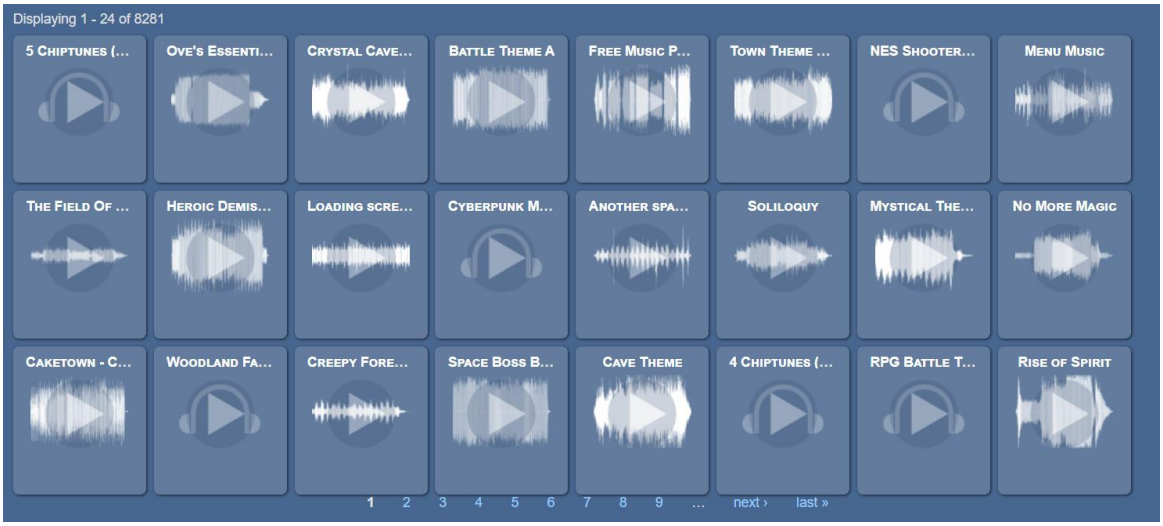

Рисунок 2.11 Секція з музикою на OpenGameArt

У результаті детального аналізу та вибору, була обрана музична композиція, яка найкращим чином відображає естетичний стиль та атмосферу гри, що допомагає поглибити іммерсію гравця.

2.4 Висновки за розділом

У цьому розділі було обрано ігровий двигун Unity для подальшої розробки гри та визначено візуальний стиль для гри, використовуючи вільні асети. Були створені звукові ефекти для покращення іммерсії геймплею та обрано фонову музику для створення потрібної атмосфери.

РОЗДІЛ 3
РОЗРОБКА ФУНКЦІОНАЛУ

3.1 Додавання ассетів та проста анімація

З метою полегшення та прискорення процесу додавання ассетів до розроблюваної гри, використовується пресет, створений на основі одного з вже вибраних ассетів. Цей підхід дозволяє ефективно інтегрувати різноманітні графічні та аудіо компоненти до проекту, забезпечуючи швидший розвиток гри. Готовий пресет демонструється на рисунку 3.1.

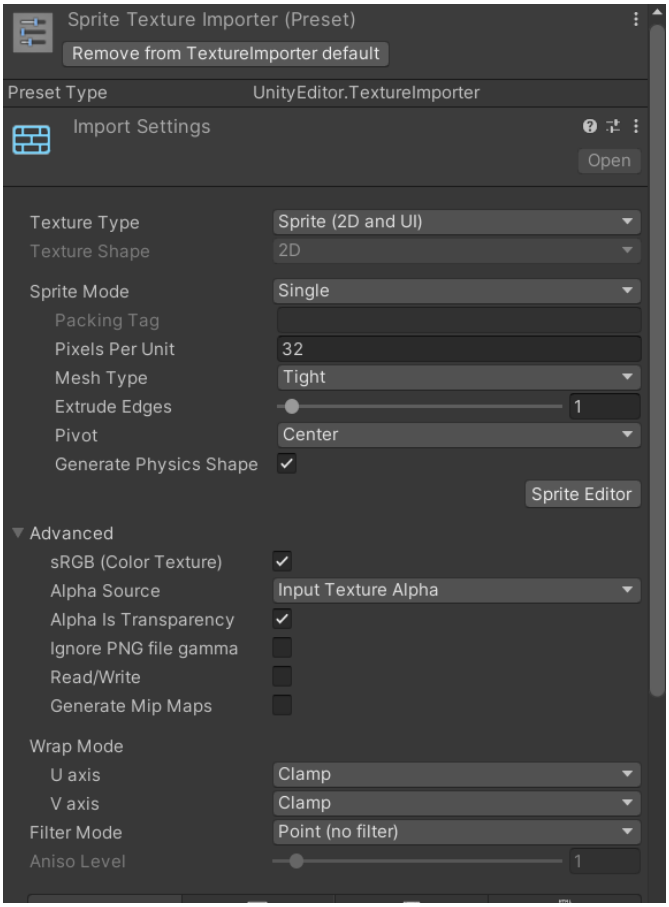

Рисунок 3.1 Готовий пресет для імпортованих ассетів

І також для його використання було створено новий профіль для пресетів, що демонструється на рисунку 3.2.

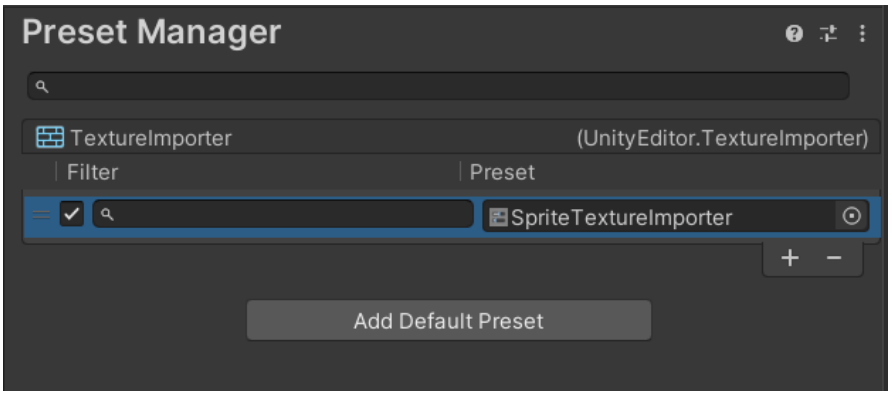

Рисунок 3.2. Створений профіль для пресету

На рисунку, що демонструє налаштування пресету для імпортованих зображень, видно використання Sprite Mode "Single". Це конфігурування дозволяє імпортувати всі ассети як одне зображення, що особливо корисно при роботі з неанімованими елементами. Однак, якщо використовуються ассети, які містять анімації, як показано на рисунку 3.3 або мають у одному рисунку багато елементів потрібно змінити налаштування Sprite Mode на "Multiple". Це забезпечить коректне імпортування анімованих та багатоелементних зображень та їх подальше використання у грі.

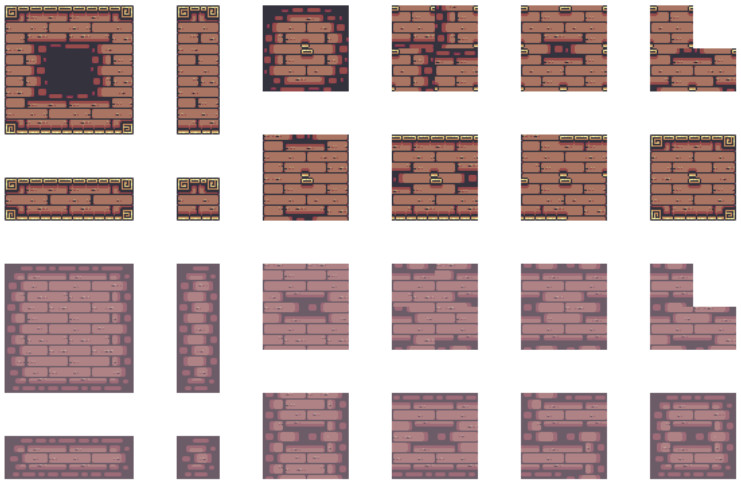

Рисунок 3.3 Рисунок без використання Multiple

Також на рисунку 3.4 показано отриманий результат зображення з використанням налаштування Multiple.

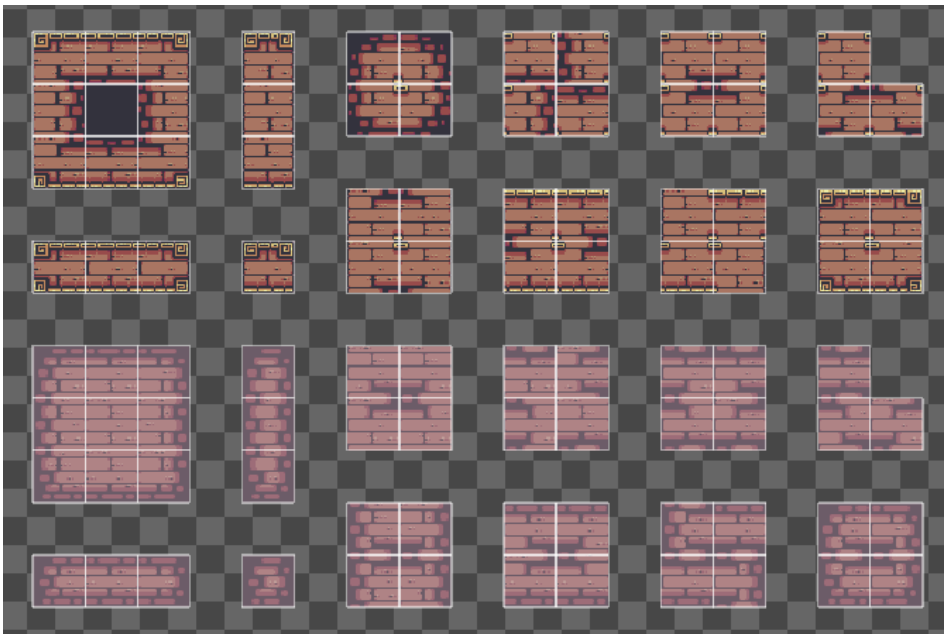

Рисунок 3.4 Отриманий результат після встановлення Multiple

З метою оптимізації розробки гри, створено Unity-скрипт для простого керування анімацією об'єктів. Цей скрипт надає можливості налаштування, такі як частота кадрів, циклічне відтворення та автоматичний перехід до наступного кліпу. Використання цього скрипту з простою анімацією демонструється на рисунку 3.5.

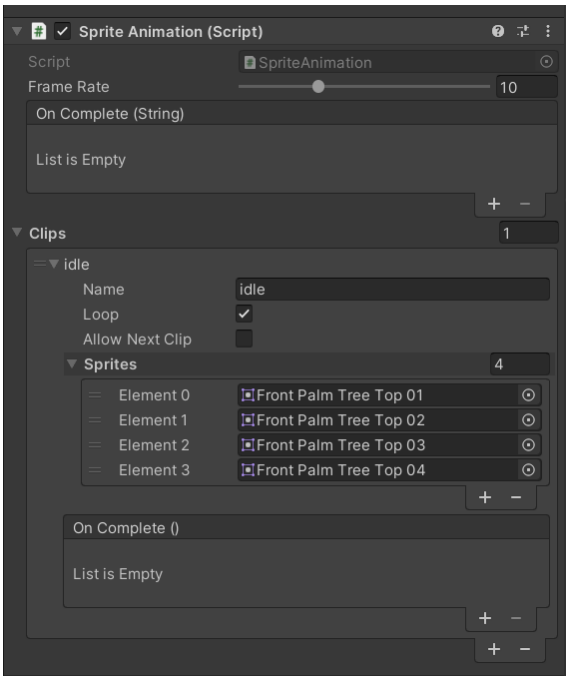

Рисунок 3.5 Використання скрипту для простої анімації

Основні характеристики скрипту включають:

1. Частота кадрів (Framerate): Визначає кількість кадрів на секунду для анімації.
2. Циклічне відтворення: Дозволяє налаштовувати анімації на безперервне циклічне відтворення.
3. Дозвіл на наступний кліп: Автоматичний перехід до наступного кліпу після завершення поточного.
4. Оптимізація роботи: Скрипт вимикає анімації, коли об’єкт вимкнутий, щоб ефективно використовувати ресурси системи.

3.2 Функціонал гравця

Одним із фундаментальних елементів будь-якої відеогри є спосіб, яким гравець взаємодіє з ігровим світом. При проектуванні цієї взаємодії, виникає необхідність розробки функціоналу пересування гравця та створення певних механік, що дозволяють йому рухатися в ігровому світі. Засоби, що використовуються для цієї мети, впливають на реалізацію гравцем різних дій та рухів у ігровому світі.

Забезпечення високоякісного та зручного руху є важливим завданням для створення іммерсивного геймплею та задоволення потреб гравців. Для досягнення цього руху, програмні реалізації відображаються через ряд методів, які включають в себе маніпулювання фізичними характеристиками героя та управління його діями.

Для реалізації базового пересування в грі було прийнято рішення використати функціонал Unity, зокрема пакет Input Action Package. Цей пакет надає можливість швидко та ефективно налаштовувати управління діями персонажа, використовуючи вбудовані методи, які відповідають за обробку вхідних команд користувача. На рисунку 3.6 демонструється створений ассет Input Action.

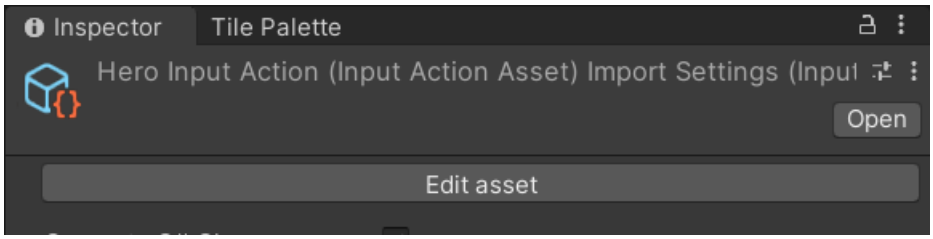

Рисунок 3.6. Створений Input Action

У процесі розробки було створено Action Map з назвою "Hero". Ця мапа була використана для визначення специфічних дій, які гравець може виконати для керування персонажем. В межах цієї мапи були створені різні Actions, які пов'язані з основними рухами та взаємодією персонажа з ігровим світом. налаштовані дії демонструються на рисунку 3.7.

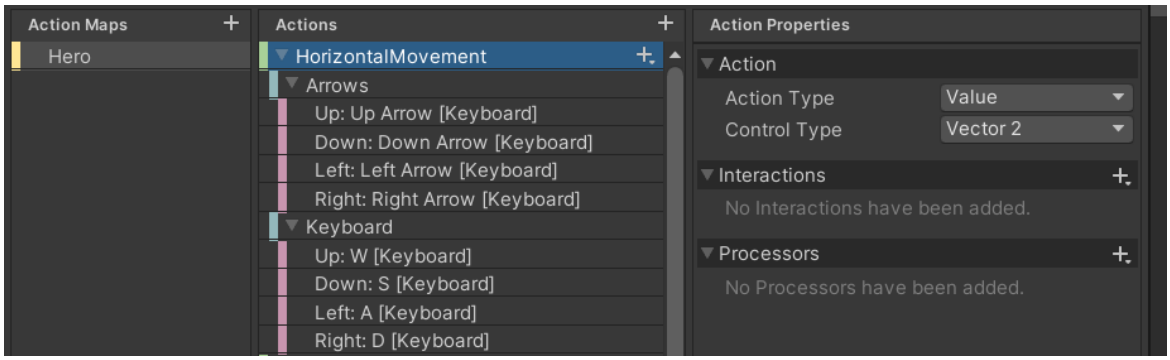

Рисунок 3.7 Створені дії для пересування

Для об’єднання створених дій для пересування та контролю за персонажем було реалізовано спеціальні методи. В класі HeroInputReader створено відповідні дії для керування персонажем. Наприклад, метод OnHorizontalMovement обробляє горизонтальні переміщення, використовуючи вхідні дані щодо напрямку руху гравця та передаючи цю інформацію об’єкту героя для подальшої обробки.

У методі Update класу Hero відбувається регулювання фізичних параметрів героя, особливо при контакті зі стіною. Цей метод відповідає за зміну гравітації, а також активує відповідні анімації при зіткненні з перешкодами.

Крім того, методи CalculateYVelocity та CalculateJumpVelocity у класі Hero відповідають за розрахунок швидкості руху вгору під час стрибків у залежності від стану гравця (чи знаходиться він на землі чи на стіні). Ця функціональність створює більш гнучке та реалістичне управління персонажем.

На рисунку 3.8 демонструється додані герою Events які відповідають за керування.

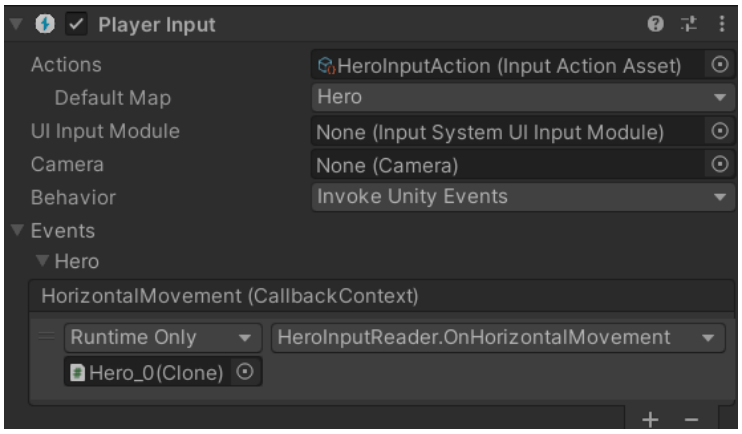

Рисунок 3.8 Event для контролю пересування

Для базової взаємодії персонажа з навколишнім середовищем розроблено скрипт ColliderCheck, що реагує на об’єкти різних шарів у грі. Цей скрипт перевіряє торкання об’єкта на якому він існує з об’єктами певних шарів, надаючи можливість виявлення та реакції на зіткнення з ними.

У методі Awake() він ініціює пошук колайдера внутрішніх об’єктів свого батьківського елементу. Під час виклику методу OnTriggerStay2D(Collider2D other) скрипт визначає, чи знаходиться батьківський елемент у стані торкання з об’єктами певних шарів. Метод OnTriggerExit2D(Collider2D other) викликається при виході персонажа з торкання об’єктів, що належать до певних шарів. На рисунку 3.9 показано налаштування для об’єкта персонажа GroundCheck, що перевіряє чи торкається персонаж шару під назвою Ground.

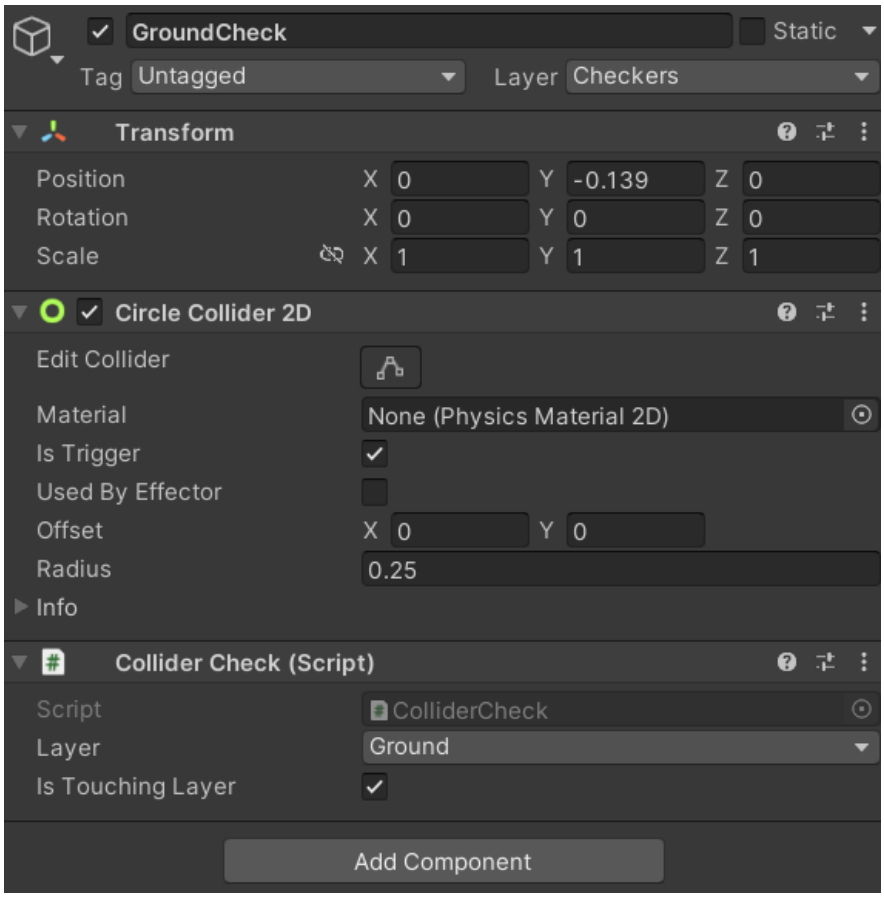

Рисунок 3.9 Налаштування для об’єкта GroundCheck

Використання подібних об’єктів до GroundCheck та відповідно WallCheck дозволяють додати можливість персонажу стрибати та виконувати відповідні дії при зіткненні зі стіною.

Для більш різноманітного геймплею створено модель здоров’я гравця. Клас HealthComponent відповідає за управлінням здоров’я. Основні функції та можливості цього компонента зроблені наступним чином:

Поля класу:

– _health: Це поле зберігає поточний рівень здоров’я гравця.

– _onDamage, _onHeal, _onDie: Події UnityEvent, які викликаються під час зменшення здоров’я, лікування чи смерті відповідно.

– _onChange: Подія UnityEvent, яка відбувається при зміні рівня здоров’я, передаючи нове значення здоров’я.

Властивості:

– Health: Повертає поточний рівень здоров’я гравця.

– Immune: Повертає об’єкт типу Lock, який використовується для захисту гравця від додаткових пошкоджень під час певних станів.

Методи:

- ModifyHealth(int healthDelta): Змінює рівень здоров'я гравця на певне значення healthDelta. Викликає відповідні події у випадках зменшення, збільшення або смерті гравця.

– SetHealth(int health): Встановлює значення здоров'я гравця.

– Методи життєвого циклу:

– OnDestroy(): При видаленні об'єкта забирає всі підписки на подію смерті _onDie.

– UpdateHealth(): [Тільки для редактора Unity] Викликає подію зміни здоров'я для оновлення візуального відображення значення здоров'я.

А також зроблено додаткову внутрішню дію, а саме метод DropItem, він викликається для випадкового скидання предмету у випадку отримання гравцем шкоди.

Для більш гнучкої невразливості гравця від пошкоджень створено клас ImmuneAfterHit, що надає користувачеві невразливість протягом певного періоду після отримання ушкодження. Основні елементи цього класу:

Поля класу:

– _immuneTime: Час невразливості після отримання ушкодження.

– _health: Елемент компоненту здоров’я HealthComponent.

– _trash: Об’єкт CompositeDisposable для управління підписками на події

Методи:

– Awake(): Встановлює посилання на HealthComponent та підписується на подію отримання ушкодження (_onDamage).

– OnDamage(): Запускає процес надання невразливості після отримання ушкодження, викликаючи метод MakeImmune().

– MakeImmune(): Корутина, що надає невразливість на певний час після отримання ушкодження.

– TryStop(): Перевіряє і, за потреби, зупиняє корутину.

– OnDestroy(): Зупиняє корутину і звільняє ресурси під час видалення об'єкта.

На рисунку 3.10 показано налаштування для компоненту ImmuneAfterHit.

Рисунок 3.10 Налаштування для компоненту ImmuneAfterHit

Щоб взаємодіяти з оточуючим світом у грі створено скрипт для визначення та реагування на колізії об’єктів. EnterCollisionComponent є скриптом, що визначає реакцію об’єкта на колізії у 2D просторі. Він призначений для виконання певних дій під час зіткнення з іншими об’єктами, які мають певний тег. Відповідний метод викликається при зіткненні об’єкта з іншими об’єктами. Він перевіряє, чи має зіткнений об’єкт вказаний тег, у разі співпадіння тегів викликається відповідна подія. Приклад використання даного скрипту демонструється на рисунку 3.11.

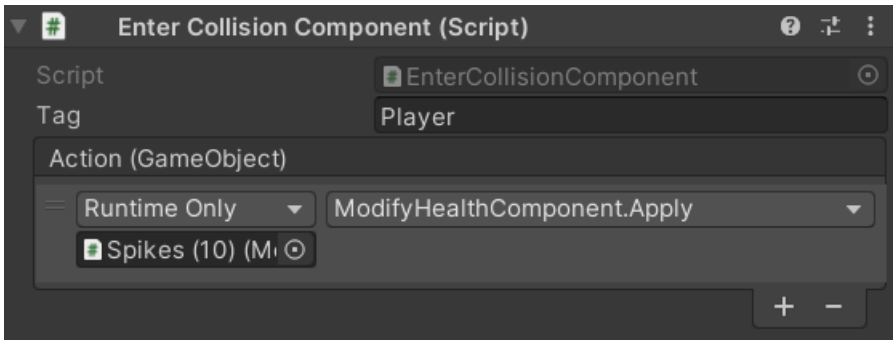

Рисунок 3.11 Використання компоненту EnterCollisionComponent

Відповідно до наданого прикладу, також було створено компонент ModifyHealthComponent, який дозволяє модифікувати здоров’я об’єкта у грі. Цей компонент здатний змінювати поточний рівень здоров’я цільового об’єкта, використовуючи значення зміни здоров’я. На рисунку 3.12 демонструється налаштування цього компоненту щоб наносилась шкода здоров’ю.

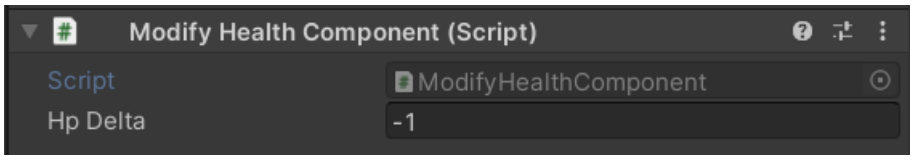

Рисунок 3.12 Налаштування компоненту для нанесення шкоди

За допомогою методу SetDelta(int delta) встановлюється значення зміни здоров’я, яке буде застосовано до цільового об’єкта, а метод Apply(GameObject target) застосовує зміну здоров’я до цільового об’єкта у грі. Він отримує доступ до компонента здоров’я цільового об’єкта, перевіряє його наявність і, якщо знайдено, застосовує задане значення зміни здоров’я за допомогою методу ModifyHealth з компонента здоров’я.

3.3 Інвентар та зберігання даних

Щоб розширити функціонал гри створено компонент EnterTriggerComponent, цей компонент визначає взаємодію об’єктів, що потрапляють у тригерну зону. Цей компонент має параметри, такі як _tag (мітка) та _layer(шар), які використовуються для визначення, які об’єкти можуть взаємодіяти з тригером. Метод OnTriggerEnter2D викликається, коли інший об’єкт потрапляє в тригерну зону цього об’єкта. У цьому методі спершу перевіряється, чи об'єкт, який потрапив в зону тригера, знаходиться у встановленому шарі (заданий у _layer). Потім перевіряється, чи відповідає тег цього об'єкта вказаному _tag (якщо він заданий). Якщо обидва умови виконуються, викликається подія _action, яка може мати взаємодію з об'єктом,

що увійшов у зону тригера. Для прикладу на рисунку 3.13 демонструється використання цього компоненту для додавання монети до інвентарю персонажа.

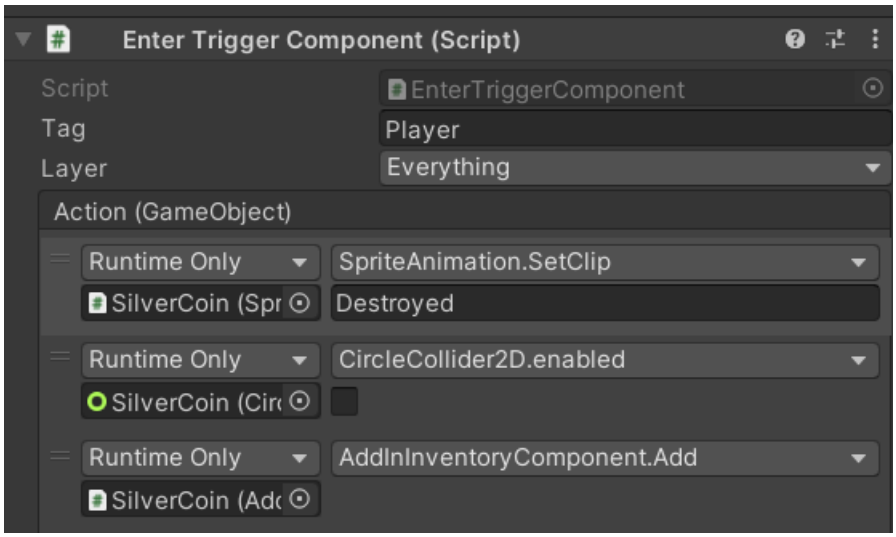

Рисунок 3.13 Використання компоненту EnterTriggetComponent для монети

Відповідно, для використання функціоналу інвентаря було створено код, що включає клас DefRepository<TDefType>. Цей клас є типом ScriptableObject, що має загальний функціонал для зберігання різноманітних предметів у грі. Він забезпечує зберігання колекції предметів у вигляді масиву типу TDefType, де TDefType реалізує інтерфейс IHaveId.

Клас ItemRepository виступає як певна реалізація DefRepository для конкретних предметів (тип ItemDef). Клас ItemDef є структурою, що містить опис окремого предмету. Кожен екземпляр ItemDef має унікальний ідентифікатор (Id), зображення (Icon) та набір тегів (_tags), які дозволяють класифікувати предмети за різними параметрами.

Метод Get(string id) в DefRepository дозволяє швидко знаходити предмет за його унікальним ідентифікатором, тоді як All повертає масив усіх предметів.

Також варто зазначити, що в ItemsRepository присутній метод ItemsForEditor, доступний лише у режимі редактора Unity. Цей метод дозволяє

зручно переглядати та відстежувати колекцію предметів під час розробки гри в середовищі редактора. На рисунку 3.15 показано приклад колекції предметів.

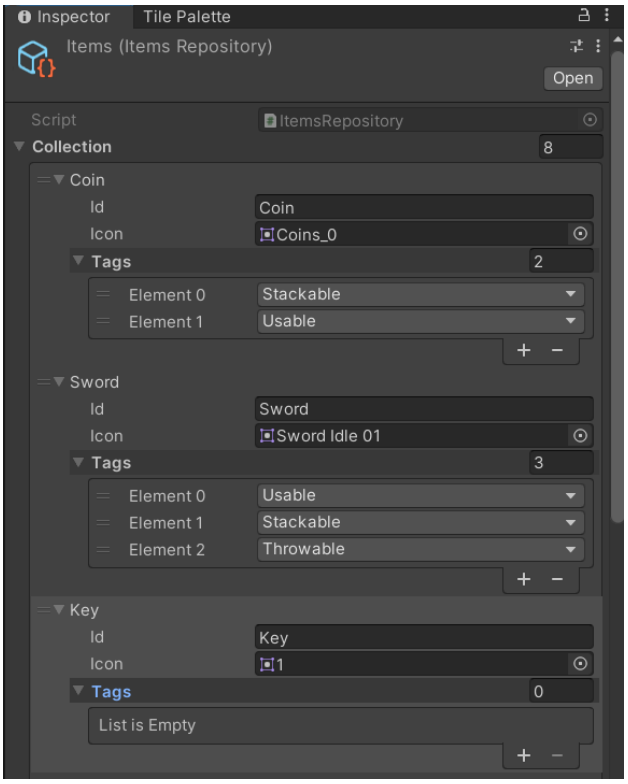

Рисунок 3.15 Колекція предметів та їх характеристики

Цей функціонал надає основу для організації та зберігання предметів у грі, даючи зручний доступ до їхніх характеристик та властивостей.

Для можливості управління інвентарем персонажа створено клас InventoryData, цей клас містить ряд методів для додавання, видалення та перевірки наявності предметів у інвентарі.

Метод Add відповідає за додавання предметів у інвентар. Він перевіряє, чи можна додати вказану кількість предметів, і додає їх до інвентаря з урахуванням їх типу (складається або ні).

Метод Remove видаляє предмети з інвентаря відповідно до вказаної кількості. Якщо предмет складається, кількість зменшується; якщо ні, видаляється необхідна кількість окремих предметів.

Метод IsEnough перевіряє, чи є достатня кількість певних предметів у інвентарі, з урахуванням їх кількості та типу. Це важливий метод для перевірки наявності необхідних предметів перед виконанням певних дій у грі.

InventoryItemData - це структура, яка містить інформацію про окремий предмет у форматі Id (ідентифікатор) та Value (кількість).

Для додавання предметів до інвентарю створено клас AddInInventoryComponent, який є компонентом Unity MonoBehaviour. В ньому є два поля:

1. _id: Це поле містить ідентифікатор предмета, який буде доданий до інвентаря.
2. _count: Це поле визначає кількість одиниць предмета, які будуть додані до інвентаря.

Метод Add(GameObject go) виконує додавання предмета до інвентаря. Він отримує доступ до інтерфейсу ICanAddInInventory об’єкта героя, щоб передати методу AddInInventory необхідний ідентифікатор та кількість предметів. Якщо об’єкт героя підтримує додавання предметів до інвентаря (ICanAddInInventory), метод AddInInventory буде викликаний з відповідними аргументами (_id та _count), які були встановлені для компонента AddInInventoryComponent. На рисунку 3.16 демонструється використання цього компоненту.

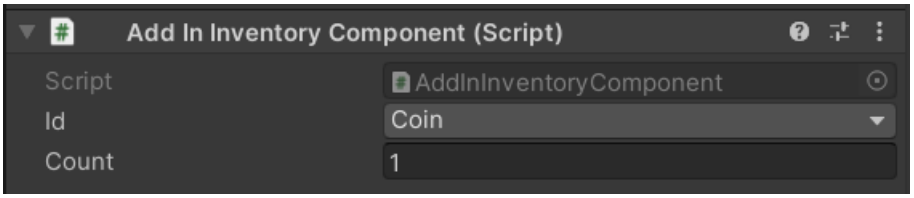

Рисунок 3.16 Приклад додавання предмету до інвентарю

PlayerData створено для ефективного керування даними гравця, включаючи їх збереження та управління. Однією з ключових можливостей PlayerData є його здатність працювати з інвентарем гравця. В класі реалізовані методи, що дозволяють зберігати і використовувати дані про предмети, отримані

гравцем. При цьому забезпечується можливість керування цими даними з метою оптимізації геймплею та збереження стану гри.

Для оптимізації ресурсів системи створено клас ActionDisposable, що є реалізацією інтерфейсу IDisposable. При створенні об’єкта цього класу передається делегат onDispose, який буде викликаний при виклику методу Dispose. Це забезпечує звільнення ресурсів або виконання певних операцій після використання об'єкта ActionDisposable.

Також створено QuickInventoryModel, що відображає модель швидкого інвентаря. У конструкторі він отримує посилання на PlayerData і заповнює інвентар виключно предметами, які можна використовувати (Usable). Також, він підписується на подію зміни інвентаря гравця для відслідковування змін у власному інвентарі. Методи класу дозволяють підписуватися на зміни, вибирати наступний предмет, а також вивільняти ресурси, які використовуються для відстеження змін у QuickInventoryModel.

Клас GameSession відображає та керує ігровою сесією в цілому. Однією з його важливих властивостей є обробка та зберігання даних гравця. Для системи інвентарю, спеціальні методи відповідають за зберігання та завантаження даних гравця. Під час запуску гри, GameSession ініціалізує основні параметри гравця, зберігає його стан та починає нову сесію. Основні методи що працюють з інвентарем наступні:

1. Метод Awake – запускається при старті сцени та перевіряє, чи існує вже активна сесія гри. Якщо така сесія вже існує, вона активується, і поточний об’єкт GameSession знищується. В іншому випадку, він створює збереження та ініціалізує моделі для подальшого використання.

2. Метод InitModels – ініціалізує моделі для гравця. Це включає створення об’єкту QuickInventoryModel і збереження його у QuickInventory, після чого він додається до списку _trash.

3. Метод Save – створює клон поточних даних гравця для подальшого збереження.

4. Метод LoadLastSave – відновлює останній збережений стан гри, використовуючи клон даних, який був створений під час збереження. Він замінює поточні дані гри на збережені дані та ініціалізує моделі для подальшого використання.

5. Метод GetExistsSession – шукає наявну сесію гри серед усіх об’єктів GameSession, щоб переконатися, чи вже існує активна гра. Він повертає наявну сесію гри або null якщо іншої сесії не знайдено.

3.4 Механіка ворогів та бойовий аспект гри

Механіка бою та взаємодії з ворогами відіграє ключову роль у створенні цікавого геймплею. Елементи взаємодії з ворогами в грі створюють особливу атмосферу та викликають інтерес до стратегічного планування та розв’язання завдань.

У процесі розробки виникла потреба в універсальному класі, що дозволяє створювати та управляти параметрами для різних сутностей. Для розширення масштабу проекту та оптимізації управління параметрами сутностей в грі був реалізований клас Creature. Цей клас надає базові параметри та функції, що є спільними для всіх створінь у грі. Клас Creature виступає в якості основи, від якої успадковуються класи, такі як Hero та вороги, забезпечуючи їм базовий набір функцій, таких як пересування, обробка отримання шкоди, здійснення атаки. Приклад використання класу Creature демонструється на рисунку 3.17.

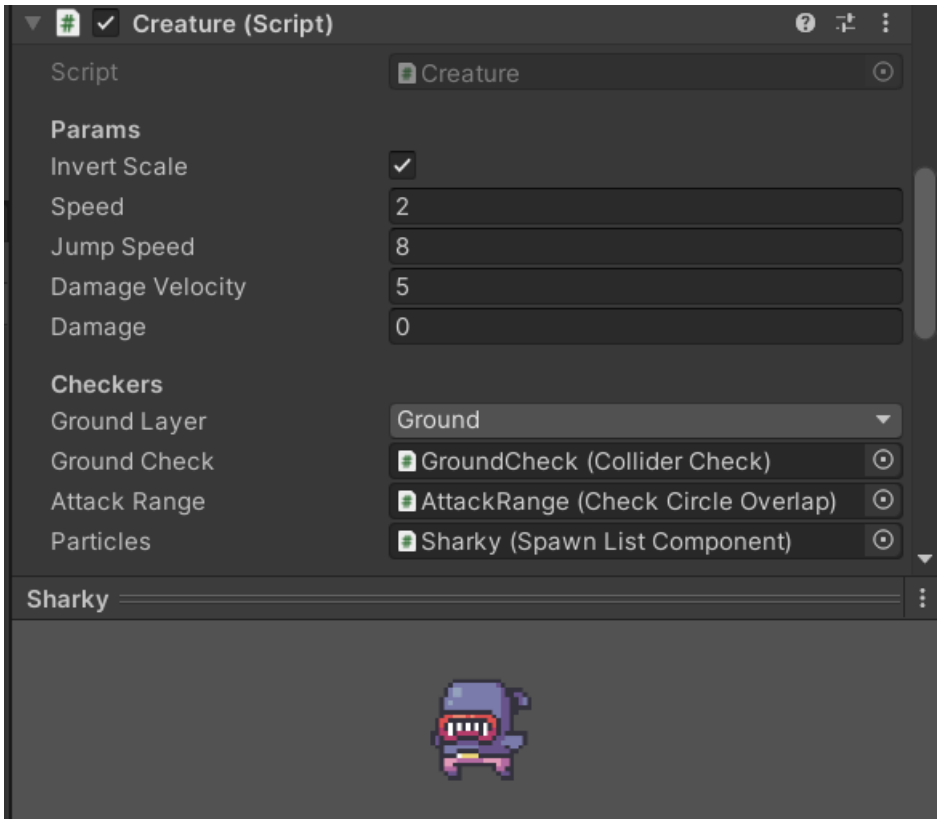

Рисунок 3.17 Використання класу Creature для ворога

Важливою особливістю класу Creature є можливість перезапису методів, що дозволяє успадкованим класам персоналізувати та адаптувати ці функції під конкретні потреби героїв чи ворогів у грі.

Динамічна поведінка та взаємодія ворогів з гравцем є основою для іммерсивності та глибини геймплею, підвищуючи рівень викликів та зацікавленості. Для цього створено клас MobAI, що відіграє значущу роль у наданні ворогам інтелекту та поведінкових сценаріїв, що дозволяє їм виявляти розумні реакції та стратегії взаємодії з гравцем. Цей клас динамічно реагує на події у грі, контролюючи перехід ворогів між різними станами та діями.

Механіка MobAI заснована на використанні корутин, що надає можливість ворогам виявляти складне та варіативне поведінкове забарвлення. Вона дозволяє імітувати різні реакції та стратегії ворогів, від виявлення героя до взаємодії та атаки. На рисунку 3.18 показано використання MobAI для ворога.

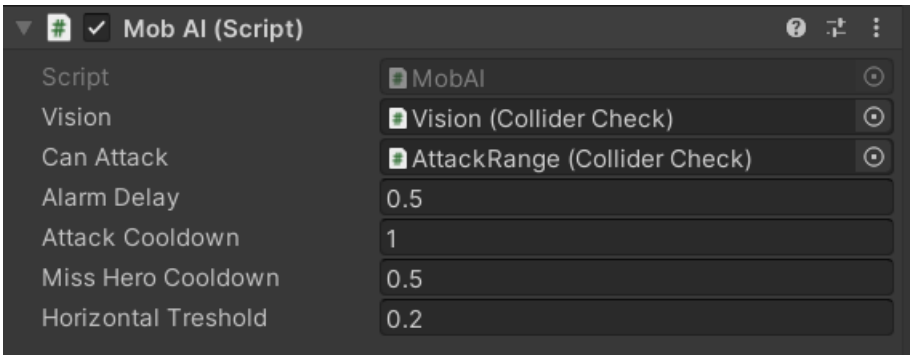

Рисунок 3.18 Використання MobAI для ворога

Він використовує об’єкт Vision що в собі має реалізацію ColliderCheck та EnterTriggerComponent, що дозволяє ворогу шукати гравця та мати відповідне поле зору, яке можна побачити на рисунку 3.19.

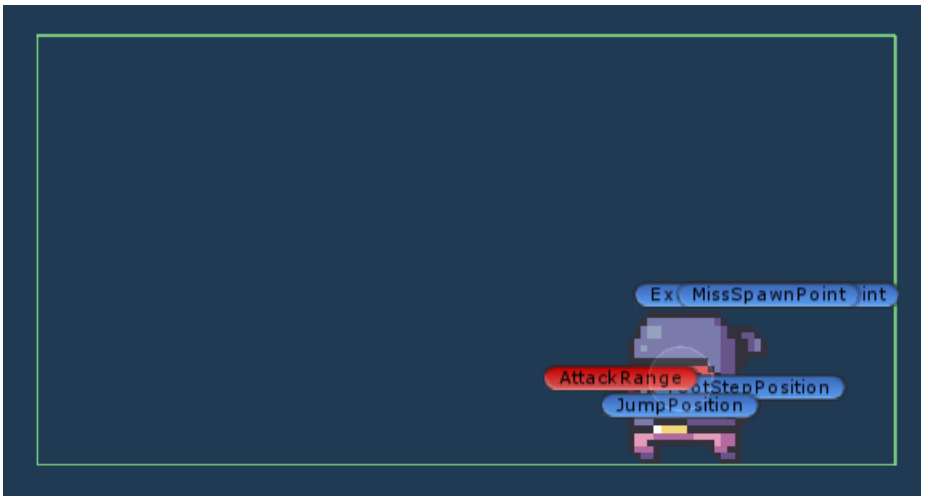

Рисунок 3.19 Поле зору ворога

Налаштування параметрів для об’єкта Vision демонструються на рисунку 3.20.

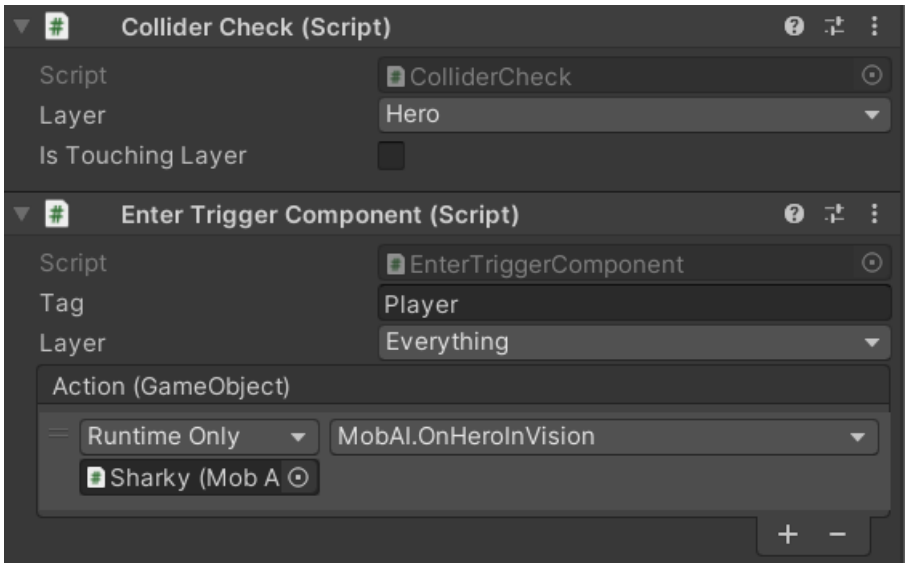

Рисунок 3.20 Налаштування компонентів для об’єкта Vision

Реалізація такого механізму AI впливає на загальний досвід гри, створюючи ситуації та різноманітні стратегії взаємодії, що робить ігровий процес більш цікавим для гравців.

У створенні ігрових ворогів значну увагу приділяють їхній автономній поведінці, зокрема пересуванню та навігації в оточуючому середовищі. Механіка автоматичного пересування ворога при відсутності героя в полі зору забезпечує підтримку ігрового процесу, дозволяючи ворогами самостійно досліджувати ігровий світ.

Сценарії автоматичного руху ворогів були реалізовані за допомогою скрипту PlatformPatrol, який включає в себе два LineCheck. Перший LineCheck призначений для перевірки наявності поверхні (землі), а другий – для перевірки перешкод (об’єктів, що можуть перешкоджати руху). На рисунку 3.21 демонструються ці два LineCheck.

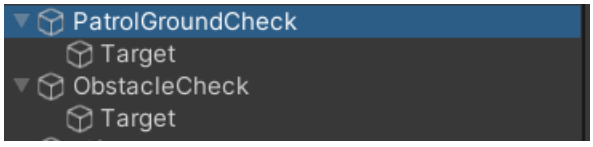

Рисунок 3.21 Використання LineCheck для перевірок шарів

Також на рисунку 3.22 можна побачити приклад налаштування для LineCheck що перевіряє перешкоди та землю.

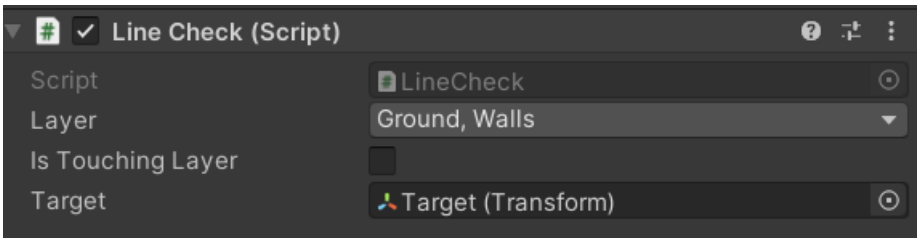

Рисунок 3.22 Налаштування LineCheck для перевірки перешкод та землі

Ці перевірки дозволяють ворогам автоматично пристосовуватись та пересуватись вздовж наявних платформ.

PlatformPatrol є абстрактним класом, який реалізує загальну логіку пересування та успадковується від сценарію руху Patrol. Клас виконує основну функцію керування рухом ворога, орієнтуючись на результати перевірок ліній та спрямовуючи ворога відповідно до цих умов.

Також створена додаткова реалізація патрулювання – PointPatrol. Цей скрипт дозволяє значно розширити функціонал патрулювання та надає можливості для більш гнучкого управління маршрутами руху ворогів у грі.

PointPatrol базується на використанні набору точок, які визначають маршрут патрулювання. Кожна точка визначає місце, до якого ворог повинен дійти, щоб перейти до наступної точки патрулювання.

Основний метод DoPatrol реалізовано таким чином, що він перевіряє, чи вже досягнута поточна точка. Після досягнення поточної точки, ворог переходить до наступної, утворюючи замкнутий маршрут патрулювання. Визначення напрямку руху відбувається через обчислення вектора до наступної точки і встановлення цього напрямку до об’єкта Creature.

PointPatrol додає нові можливості для гнучкого управління траєкторією руху ворогів, дозволяючи встановлювати різні зони для патрулювання та індивідуально налаштовувати маршрути для кожного ворога.

Окрім ворогів що пересуваються також розроблено й стаціонарних ворогів. Перший це Seashell що має можливості атаки на відстані та ближнього бою. Цей ворог обладнаний AI, який спроектований для виконання дій в залежності від обставин навколишнього середовища та відстані до гравця.

SeashellTrapAI виявляє наявність гравця у своїй області видимості, використовуючи LayerCheck для цього. Якщо гравець опиняється у зоні видимості, ворог спробує визначити найбільш відповідну стратегію атаки:

– Ближній бій: Якщо гравець знаходиться у зоні ближнього бою, ворог має можливість виконати атаку ближнього діапазону. Це досягається за допомогою Cooldown та LayerCheck для перевірки можливості атаки.

– Дальній бій: Якщо гравець знаходиться у зоні дальнього бою, і атака ближнього діапазону недоступна, ворог має можливість виконати атаку на відстані. Тут також використовується Cooldown для керування часом між атаками.

Кожна атака викликає реакцію, включаючи проведення самої атаки, здійснення перевірок і спавн відповідних об’єктів.

SeashellTrapAI створено таким чином, щоб забезпечити ефективну та реалістичну реакцію на присутність гравця та виконання різних типів атак в залежності від ситуації. На рисунку 3.23 демонструється налаштування для SeashellTrapAI.

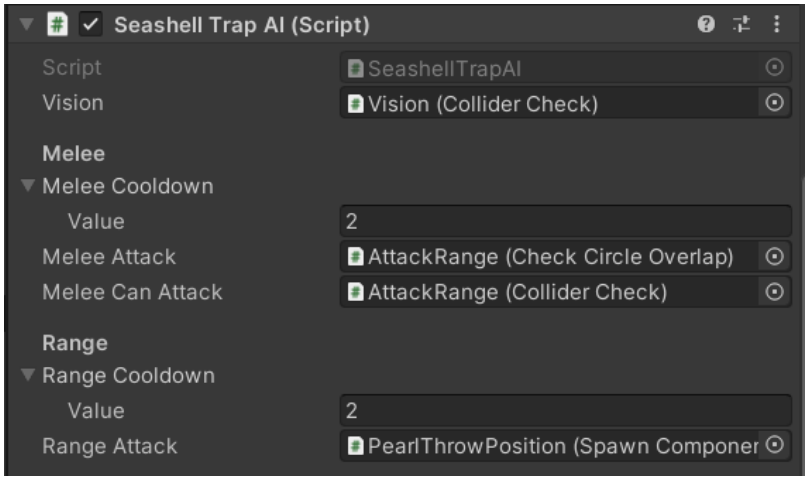

Рисунок 3.23 Налаштування для SeashellTrapAI

Для спавну відповідних об’єктів використовується скрипт SpawnComponent, він дозволяє створення нових об’єктів у грі. Він визначає місце, де буде розташовано створений об’єкт та сам об’єкт, який потрібно створити. Метод Spawn створює новий об’єкт у вказаній позиції, використовуючи обраний до цього префаб. Метод SetPrefab дозволяє змінити об’єкт, який буде створено за допомогою _prefab.

Клас Coolown використовується для управління часовими інтервалами. Він має властивість Value, яка представляє тривалість цього інтервалу. Метод Reset встановлює момент часу, коли інтервал закінчиться, додаючи значення _value до поточного часу. Властивість RemainingTime повертає час, що залишився до кінця інтервалу, обчислюючи різницю між моментом часу, визначеним методом Reset, та поточним часом. І властивість IsReady вказує на те, чи минув вже встановлений інтервал часу, повертаючи true, якщо час інтервалу пройшов.

Для усіх снарядів у грі створено BaseProjectile, це базовий клас для всіх снарядів, він має поля для швидкості _speed та флагу _invertX, який визначає можливість зміни напрямку руху по осі X. Цей клас також містить посилання на Rigidbody2D для керування фізикою об’єкту та змінну Direction, яка визначає напрямок руху залежно від масштабу об’єкту. У методі Start, спочатку обчислюється напрямок руху залежно від установленого _invertX та масштабу об’єкту, після чого отримується посилання на Rigidbody2D компонента.

Але для снаряду який запускає Seashell було розширено базовий функціонал снарядів і створено SinusoidalProjectile. Він наслідує функціонал BaseProjectile та додає особливості руху у формі синусоїди. Крім умовних полів для частоти _frequency та амплітуди _amplitude синусоїдального руху, він також містить змінні для збереження початкового значення Y-координати та часу _originalY та _time відпоівдно. У методі Start, спочатку викликається метод базового класу Start для ініціалізації базових параметрів, після чого зберігається початкове значення Y-координати. У методі FixedUpdate кожен кадр визначається нова позиція снаряду, враховуючи рух уперед з врахуванням _speed

та синусоїдальний рух вздовж Y-координати за допомогою Mathf.Sin(). Кожного кадру час _time збільшується на Time.fixedDeltaTime, що впливає на форму синусоїди. На рисунку 3.24 демонструється налаштування для SinusoidalProjectile.

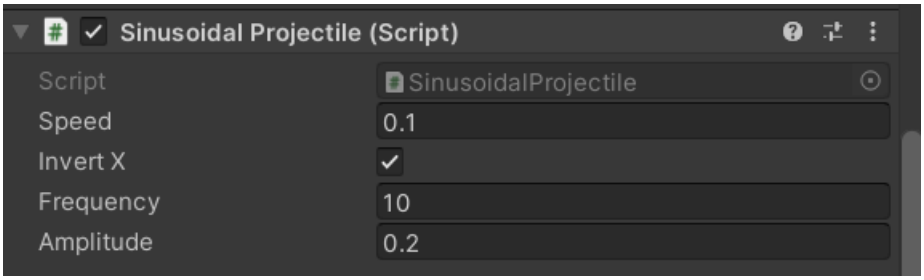

Рисунок 3.24 Налаштування для SinusoidalProjectile

Задля оптимізації системних ресурсів було створено два компонента RestoreStateComponent та DestroyObjectComponent.

Компонент RestoreStateComponent призначений для відновлення стану об’єкта під час завантаження рівня в грі. У нього є поле _id, яке ідентифікує об’єкт. Під час запуску він намагається отримати посилання на об’єкт GameSession в грі. Викликається метод RestoreState, передаючи йому ідентифікатор об’єкта _id. Якщо метод повертає true, це означає, що об’єкт має бути знищений, тому він викликає Destroy(gameObject) для знищення самого себе. На рисунку 3.25 показано приклад налаштування для компоненту RestoreStateComponent.

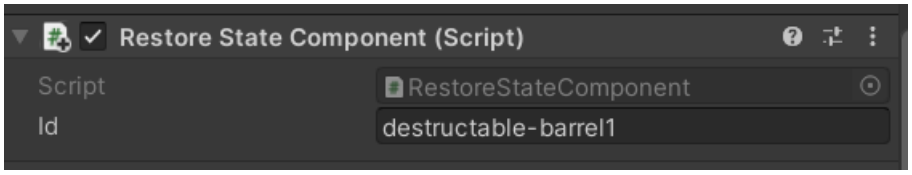

Рисунок 3.25 Приклад налаштування для компоненту RestoreStateComponent

Компонент DestroyObjectComponent призначений для знищення об’єкту в грі. Він містить посилання на об’єкт, який має бути знищений, зберігає його в _objectToDestroy. Також містить посилання на RestoreStateComponent у полі _state. При виклику методу DestroyObject, він використовує Destroy(_objectToDestroy) для знищення цього об’єкту в грі. Якщо _state не дорівнює null, він отримує посилання на об’єкт GameSession і викликає StoreState(_state.Id) для збереження стану _state, щоб у разі необхідності його можна було відновити в майбутньому. Приклад для об’єкту що може відновлюватись в майбутньому демонструється на рисунку 3.26, а для того що не може при наступному завантаженні рівня відновитись на 3.27.

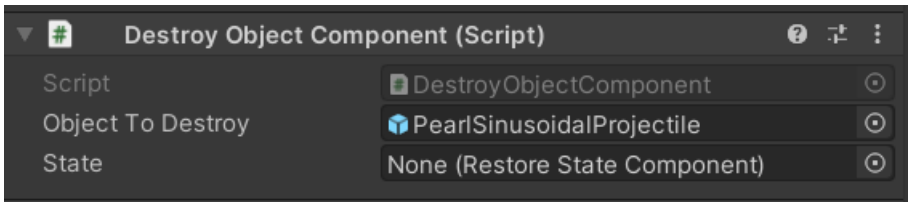

Рисунок 3.26 Налаштування для об’єкта що може відновлюватись

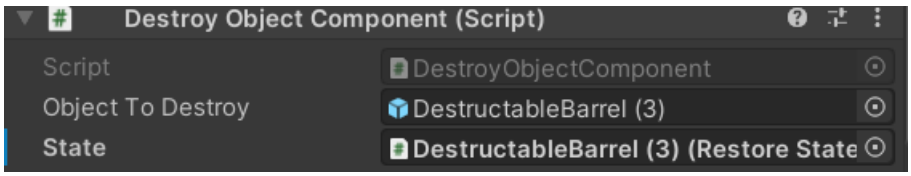

Рисунок 3.27 – Налаштування для об’єкта що не повинен відновлюватись

Відповідно для використання створених об’єктів для снарядів використовується логіка що демонструється на рисунку 3.28.

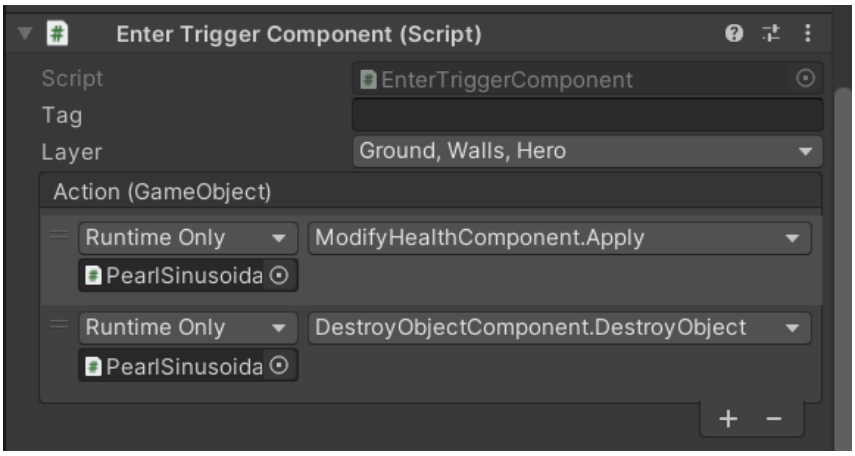

Рисунок 3.28 Приклад логіки снарядів

Спираючись на концепцію Seashell, був створений TotemTower – інший ворог дальнього бою, реалізований як складний комплекс, що має кілька шарів. TotemTower складається з декількох TotemHead, які вистрілюють послідовно. Кожна частина TotemHead цілком віддільна, і один TotemHead може вистрілювати незалежно від іншого. На рисунку 3.29 показано вигляд TotemTower та на рисунку 3.30 показано структуру TotemTower.

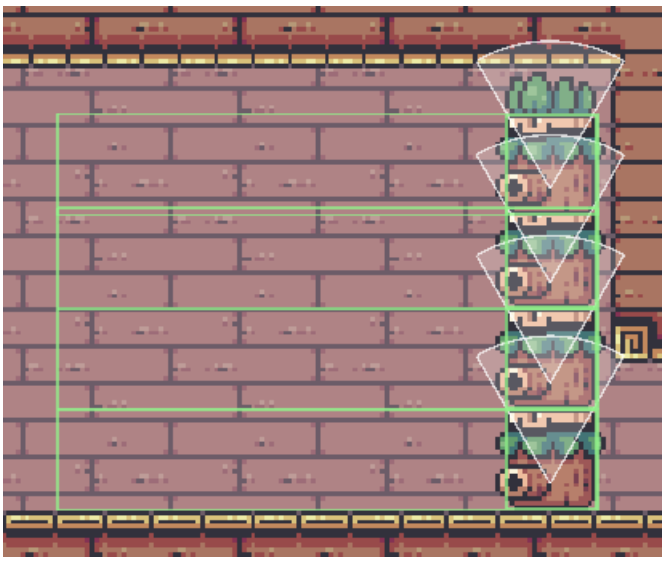

Рисунок 3.29 Вигляд TotemTower

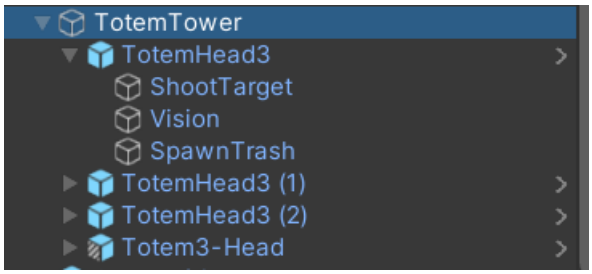

Рисунок 3.30 Структура TotemTower

Скрипт TotemTower контролює роботу цих TotemHead. При запуску гри, кожен TotemHead, який входить у склад TotemTower, вимкнений. Крім того, налаштовані прослуховувачі подій, які реагують на знищення кожного TotemHead. Якщо TotemHead знищується, він видаляється зі списку діючих і зменшує індекс поточного TotemHead. Налаштування скрипту TotemTower демонструється на рисунку 3.31.

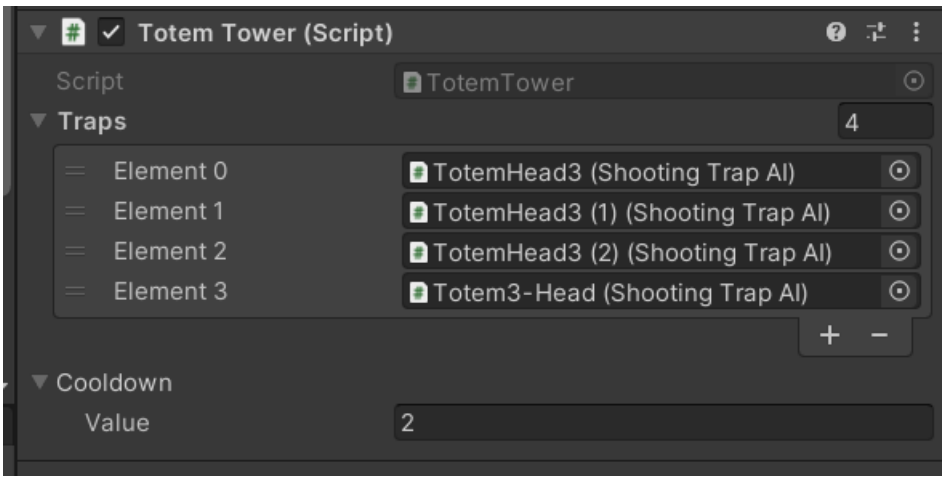

Рисунок 3.31 Налаштування скрипту TotemTower

Також важливо відмітити, що TotemTower контролює свою активність в залежності від наявності активних TotemHead. Коли всі TotemHead були знищені, TotemTower вимикається і, в кінцевому результаті, також видаляється з гри, забезпечуючи оптимізацію роботи гри.

Поведінка TotemTower в основному полягає в перевірці наявності цілей для атаки. Кожен TotemHead може вести вогонь, якщо встановлений час відпочинку _cooldown є готовим, при цьому вистрілює один TotemHead, після

чого переходить до наступного в залежності від збереження активних TotemHead.

З можливостей ближнього бою ворогів реалізовано ближній бій для головного героя. За допомогою скрипту CheckCircleOverlap, що представляє собою компонент для перевірки перетину об’єктів у формі кола, використовується у головного героя для атаки в ближньому бою. За допомогою методу Check, компонент виконує перевірку об’єктів, що перетинаються з його зоною, з врахуванням тегів та шарів. Цей компонент дозволяє головному герою виявити наявність ворогів у певній зоні. На рисунку 3.32 демонструється зона ближньої атаки героя.

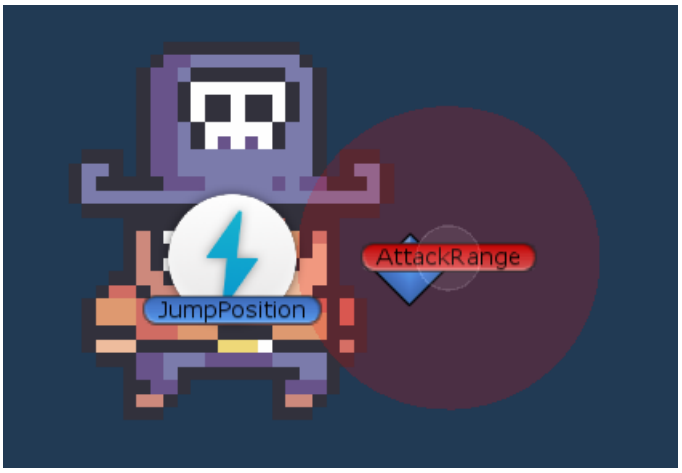

Рисунок 3.32 Зона ближньої атаки героя

У героя встановлено радіус атаки рівним 0.35, і він може атакувати об’єкти з тегом “Enemy” за допомогою методу Attack. Цей метод виконується при спробі атаки гравцем і базується на результатах, отриманих від компоненту CheckCircleOverlap.

Крім ближнього бою, головний герой може кидати об’єкти за допомогою ThrowSpawnPosition. Він може кидати предмети зі свого інвентарю, які мають тег ItemTag.Throwable. Для цього гравець використовує методи CanThrow, який перевіряє можливість кидання об’єктів, та PerformThrowing, що запускає процес кидання.

Метод PerformThrowing перевіряє готовність до кидання та активує подію кидання через Animator, після чого виконується метод OnDoThrow, який кидає об’єкт із інвентаря. При цьому герой перевіряє кількість доступних об’єктів для кидка. На рисунку 3.33 демонструється встановлений Animation Event.

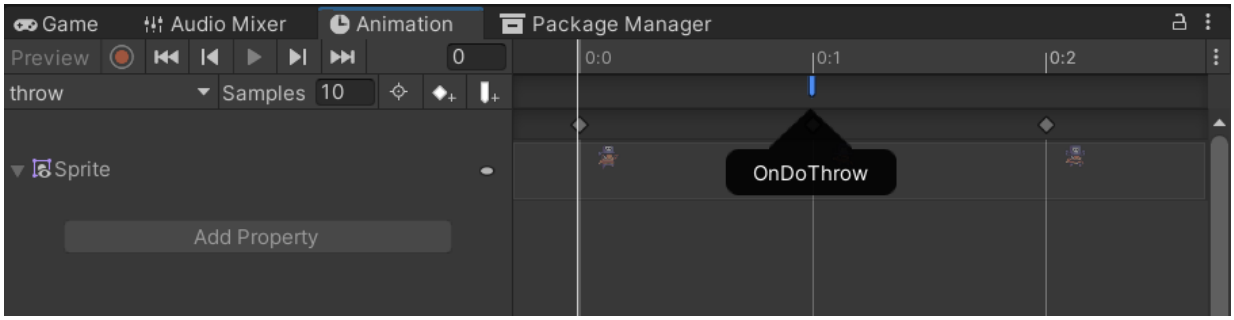

Рисунок 3.33 Встановлений Animation Event для кидка

Метод ThrowAndRemoveFromInventory відповідає за саме кидання об’єкта з інвентарю. Він встановлює параметри кидання та видаляє кинутий об’єкт з інвентарю героя. Це дозволяє гравцю використовувати об’єкти з інвентарю для дальньої атаки. Усі об’єкти гравець може кидати встановлені у Throwable Repository, це можна побачити на рисунку 3.34.

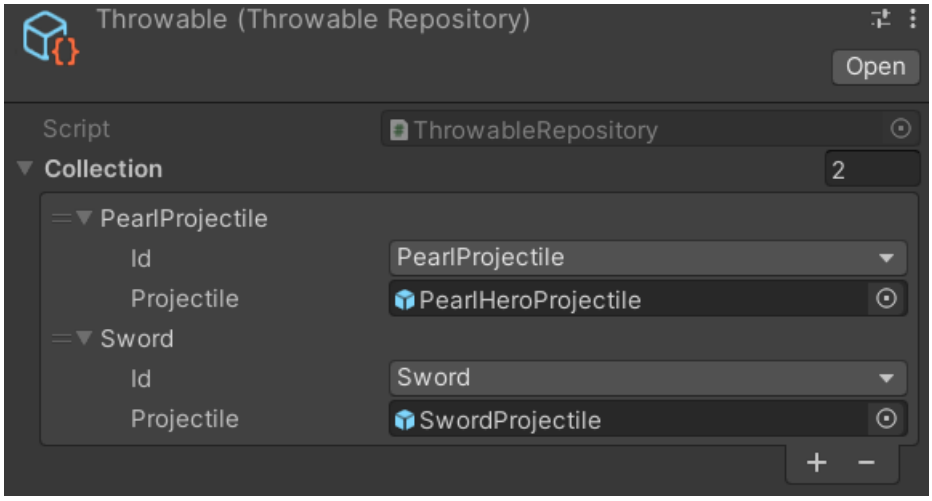

Рисунок 3.34 Предмети у Throwable Repository

Завдяки цим механікам гравець має можливості взаємодії з різними типами ворогів відповідними механіками ближнього та дальнього бою. Одночасно гравець може кидати різні предмети на ворогів, розширюючи свої можливості в дальньому бою.

3.5 Інтерфейс та меню гри

У сучасних комп’ютерних іграх, інтерфейс та меню є важливими складовими частинами геймплею, які не лише забезпечують зручність користування, але й сприяють загальній імерсії гравця у світ гри. Цей аспект ігрового досвіду необхідний для навігації, налаштування, доступу до різноманітних функцій та взаємодії з ігровим середовищем.

В головному меню гравці зазвичай знаходять опції, що дозволяють їм керувати основними параметрами гри. Це включає в себе доступ до різноманітних функцій, таких як:

– Старт гри: Розпочати або продовжити гру

– Налаштування: Налаштування графіки, звуку, управління та інші параметри, що дозволяють користувачам налаштовувати гру на свій смак

– Мова інтерфейсу: Вибір мови для тексту та інтерфейсу гри, щоб забезпечити комфортний досвід гравця незалежно від мовних уподобань

– Інші параметри: Можливість змінювати налаштування, такі як рівень складності, наявність вібрації, режим екрану тощо.

Для головного меню було обрано наступний функціонал:

1. Початок гри
2. Налаштування
3. Зміна мови
4. Вихід з гри

На рисунку 3.35 демонструється створене головне меню з відповідним функціоналом.

Рисунок 3.35 Головне меню у грі

Щоб зробити інтерфейс більш інтерактивним та цікавим, було вирішено створити анімації для різних вікон. За допомогою загального для кожного вікна скрипту AnimatedWindow, що відповідає за анімаційні ефекти відкриття та закриття вікон. Він містить логіку відображення анімації. Коли вікно показується, відтворюється анімація відкриття, а при закритті – анімація закриття, після чого вікно автоматично знищується.

Скрипт MainMenuWindow, у свою чергу, є частиною головного меню та розширює функціонал AnimatedWindow. Він містить методи, які відповідають за взаємодію з користувачем під час відкриття інших вікон: вікна налаштувань, вибору мови та виходу з гри. Кожен з цих методів викликає відповідні вікна для налаштування та керування грою. Крім того, він містить логіку завершення гри та переходу на новий рівень, а також закриття вікон за допомогою анімаційного ефекту. На рисунку 3.36 демонструється приклад анімації для функції закриття вікна.

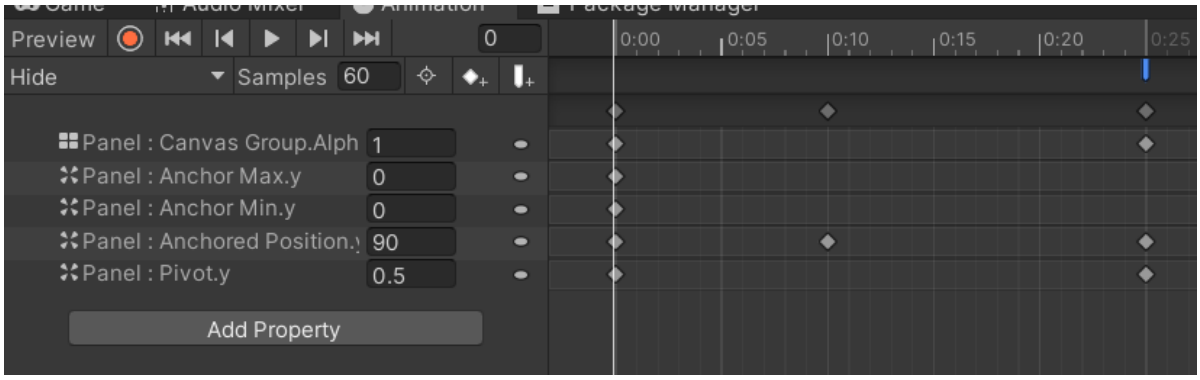

Рисунок 3.36 Анімація закриття вікна

Кожна кнопка у MainMenuWindow встановлена у пустий елемент “VerticalGroup”, який використовує компонент Vertical Layout Group. Це дозволяє організовувати кнопки у вертикальну групу, забезпечуючи рівномірне розміщення елементів. Використання цієї групи полегшує управління та розміщення кнопок у головному меню, забезпечуючи зручний та організований інтерфейс для користувача.

Для кожної кнопки створено клас CustomButton, він розширює функціонал стандартного класу Button і використовується для керування станами кнопки. Кожен елемент має два об’єкти: _norma та _pressed, які представляють відповідно звичайний та натиснутий стан кнопки. При зміні стану кнопки (SelectionState), метод DoStateTransition визначає, який стан відображати. Коли кнопка не натиснута (SelectionState не є Pressed), відображається _normal. Коли кнопка натиснута (SelectionState – Pressed), відображається _pressed. Це дозволяє змінювати зовнішній вигляд кнопки відповідно до її стану, надаючи візуальний зворотний зв’язок користувачеві при взаємодії з кнопкою. Налаштування для класу CustomButton демонструється на рисунку 3.37.

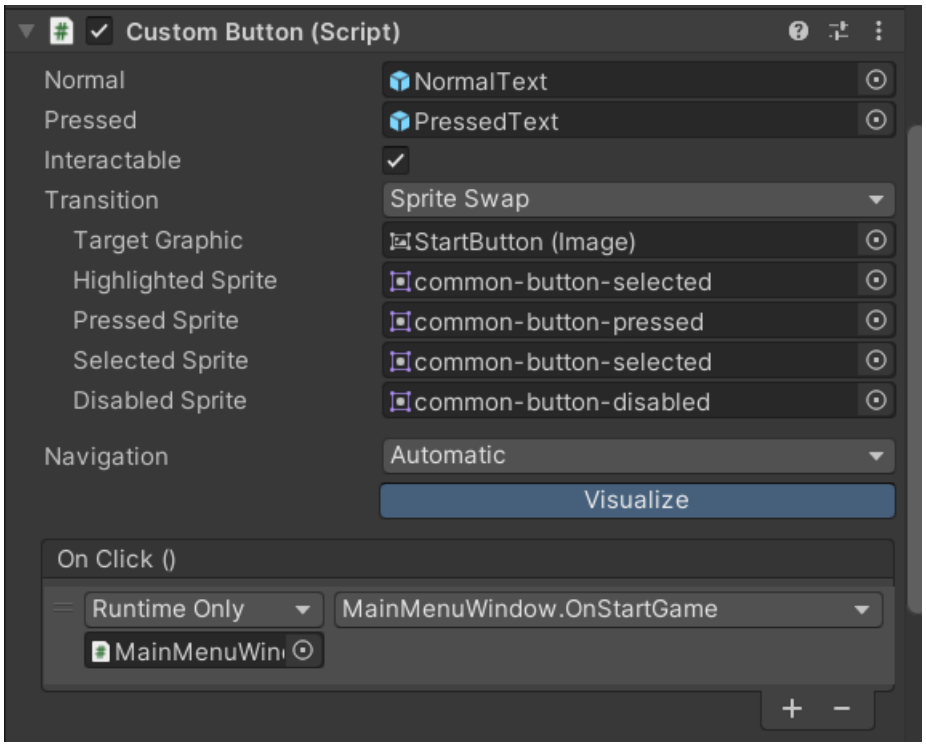

Рисунок 3.37 Налаштування класу CustomButton

За допомогою компонента Image для кожної кнопки встановлено різноманітні кольори, щоб додати змогу користувачеві одразу бачити відповідні кнопки.

Для завантаження рівнів також створено клас LevelLoader, він відповідає за завантаження рівнів у грі з використанням анімації для покращення користувацького досвіду. При створенні гри він завантажує спеціальний рівень LevelLoader, який використовується для відображення анімації переходу між рівнями. Клас зберігається в пам’яті протягом усієї гри.

Метод LoadLevel викликає анімацію завантаження, при цьому включається відповідний анімаційний ефект, після чого виконується очікування протягом визначеного часу. Після закінчення цього інтервалу рівень завантажується методом SceneManager.LoadScene(sceneName), анімація вимикається, і користувач переходить до нового рівня.

Цей підхід дозволяє використовувати анімацію під час завантаження рівнів, підвищуючи естетику інтерфейсу та забезпечуючи користувачеві час для сприйняття візуальних змін у грі під час переходу між ними.

Для керування інтерфейсом користувача створено клас HudController, загальний HUD розміщується в окремій сцені. Цей контролер відповідає за відображення та оновлення певних компонентів HUD.

Початкові дані та стан інтерфейсу оновлюються залежно від змін у грі. Наприклад відображення життя персонажа контролюється ProgressBarWidget _healthBar, який відображає рівень здоров'я гравця.

Події та зміни у грі відслідковуються для оновлення відповідних компонентів HUD. Наприклад, зміна рівня здоров'я миттєво відображається у візуальних елементах інтерфейсу. Крім цього, доступність деяких вікон у грі, таких як меню налаштувань, також контролюється через цей контролер.

Як було вище описано, відображення здоров'я контролюється компонентом ProgressBarWidget. Рівень здоров'я відображено у вигляді прогрес-бару на екрані. Цей простий скрипт використовує об'єкт типу "Image", щоб керувати відображенням прогресу на сцені гри. На рисунку 3.38 демонструється рівень здоров'я гравця в інтерфейсі.

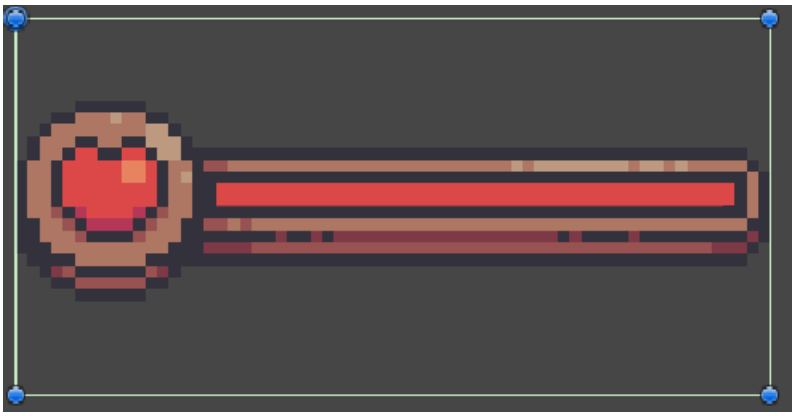

Рисунок 3.38 Рівень здоров'я гравця

Метод SetProgress(float progress) встановлює заповненість прогрес-бару відповідно до значення progress. Заповненість бару визначається значенням від 0 до 1, де 0 відповідає повній втраті здоров'я, а 1 – повному здоров'ю.

Для візуалізації використання предмету з інвентарю створено клас QuickInventoryController, цей клас відповідає за відображення інвентарю гравця

в грі. Він використовує клас DataGroup, який допомагає взаємодіяти зі списком предметів та їхніми візуальними представленнями.

Кожен предмет у швидкому інвентарі представляється об’єктом типу InventoryItemWidget, який налаштовується на відображення значення, іконки та виділення обраного предмета. Клас DataGroup відповідає за керування групою таких візуальних представлень, оновлення їхніх даних та активацію/деактивацію в залежності від кількості елементів у швидкому інвентарі.

InventoryItemWidget реалізує інтерфейс IItemRenderer, що дозволяє встановлювати дані для кожного предмета в швидкому інвентарі, такі як іконка, значення та виділення обраного предмета. Він реагує на зміни вибраного предмета, відображаючи/приховуючи відповідні позначення. На рисунку 3.39 показано приклад швидкого інвентарю та предметів в ньому.

Рисунок 3.39 Швидкий інвентар з предметами

Для доступу до меню під час гри було зроблено кнопку, яка використовує функцію OnSettings(), в класі HudController. Вона викликає функцію CreateWindow з параметром, який вказує шлях до вікна у ресурсах гри.

Клас InGameMenuWindow є реалізацією вікна, що відображається при натисканні кнопки для відкриття меню у грі. На рисунку 3.40 показано відкрите меню під час гри.

Рисунок 3.40 Відкрите меню під час гри

При старті вікна, він зупиняє час у грі, щоб призупинити гру під час відкриття меню. Крім налаштувань, вікно також містить кнопку “Вихід”, яка викликає перехід до головного меню гри, знищуючи об’єкт GameSession і відновлює роботу часу в грі при знищенні вікна.

3.6 Звуки та музичний супровід

Звукові ефекти були налаштовані відповідно їх використання в грі. Відповідні налаштування використовуються для простих звуків та музики в грі. На рисунку 3.41 демонструється приклад налаштування для музики в грі, та на рисунку 3.42 показано приклад налаштування для звуків в грі.

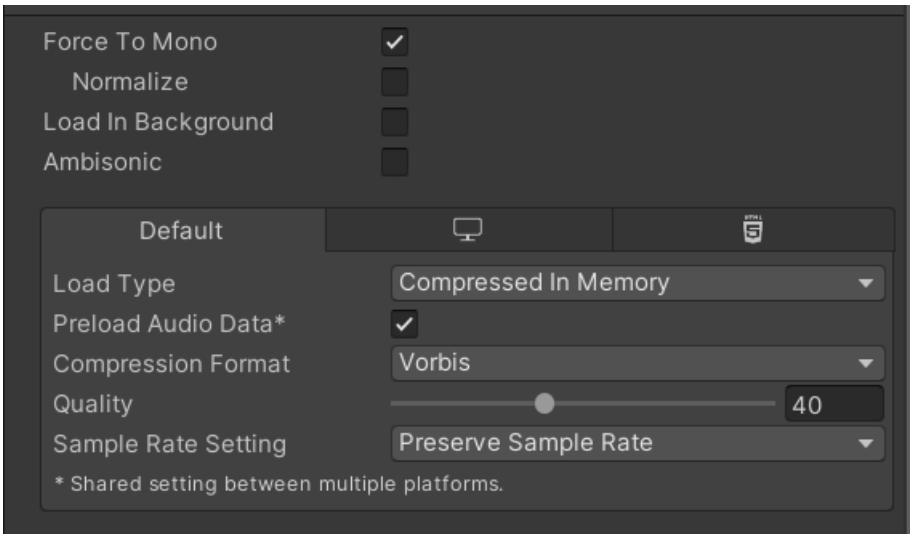

Рисунок 3.41 Приклад налаштування для музики

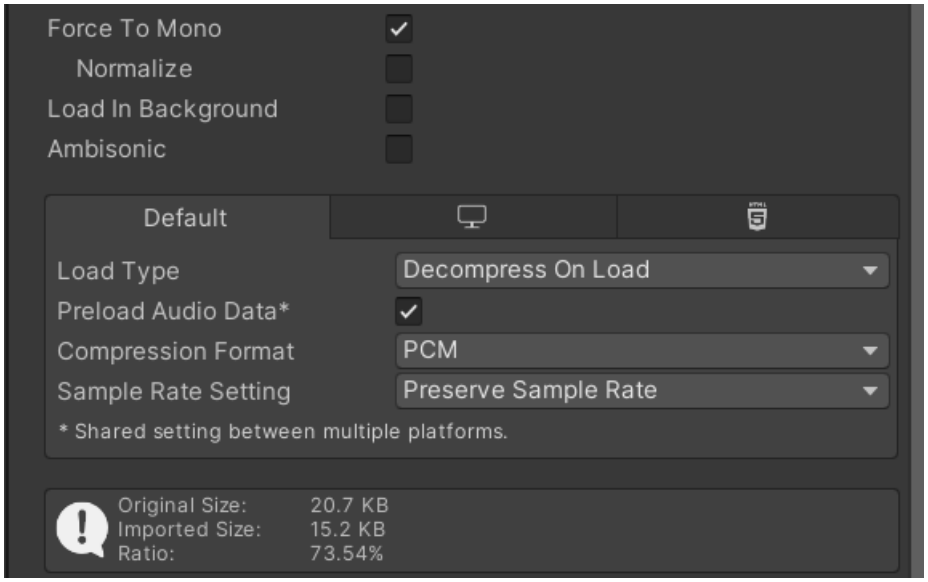

Рисунок 3.42 Приклад налаштування для звуків

Щоб налаштувати звукові ефекти для музики в грі та звукових ефектів використано AudioMixer. Налаштування для музики та звуків у грі показано на рисунках 3.43 та 3.44 відповідно.

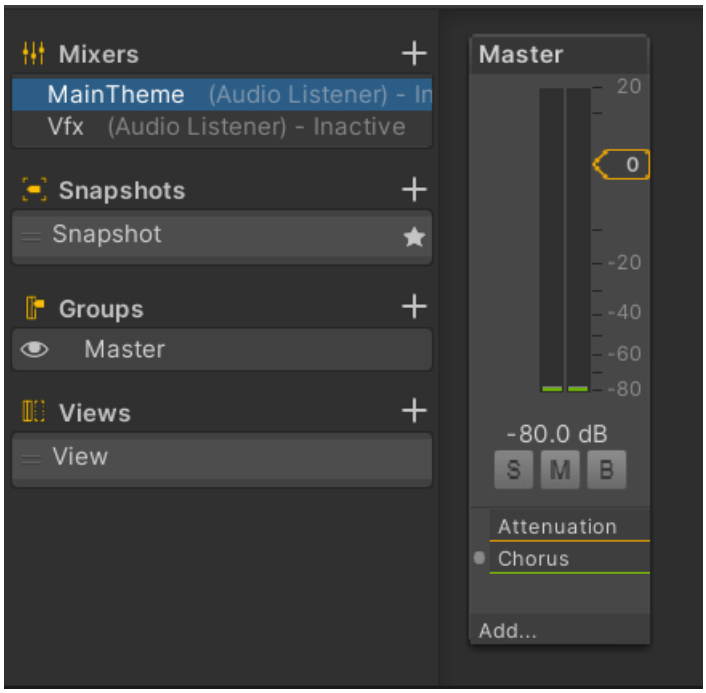

Рисунок 3.43 AudioMixer для музики

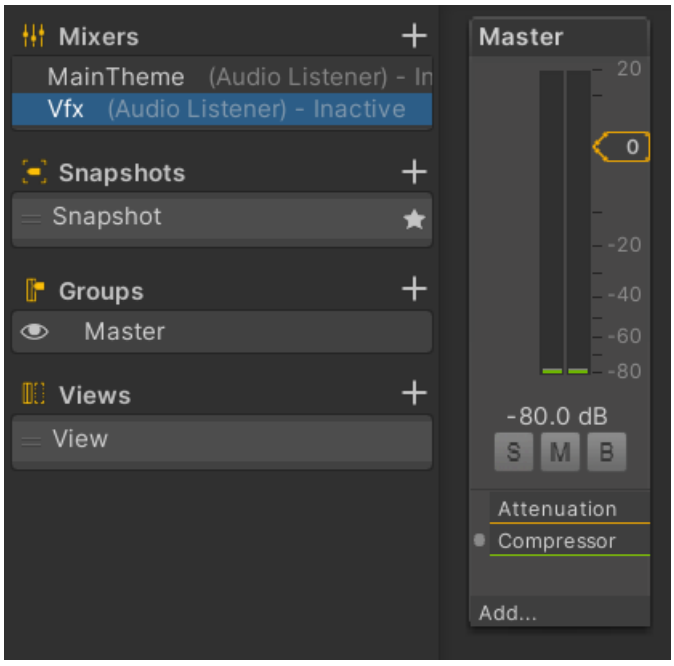

Рисунок 3.44 AudioMixer для звуків

Відповідно, щоб мати можливість програвати звуки та музику створено SfxAudioSource та MusicAudioSource на які було встановлено компонент Audio Source. На рисунках 3.45 та 3.46 показано налаштування для звуків та музики.

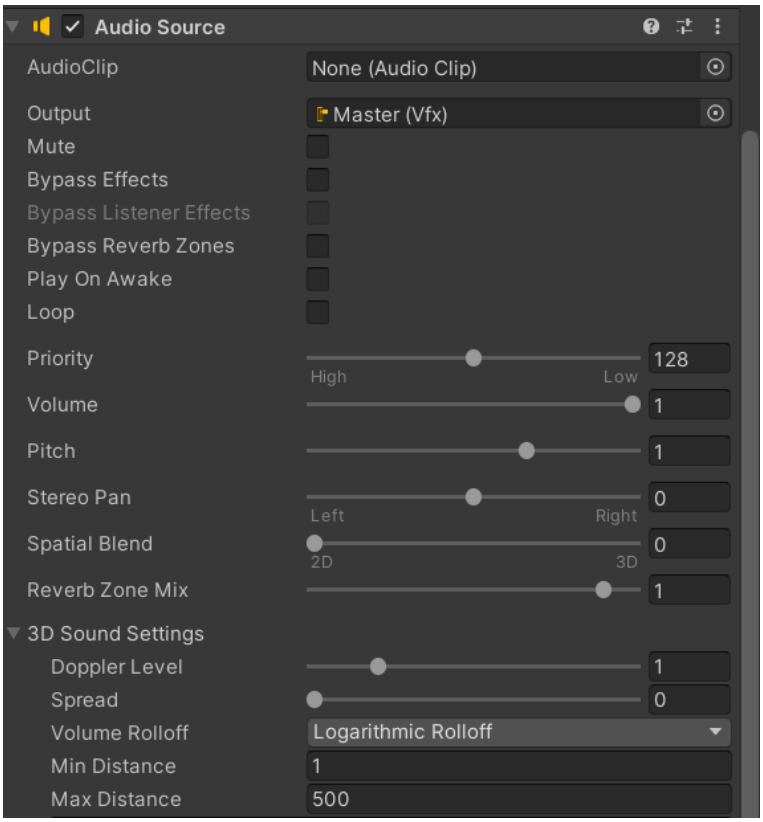

Рисунок 3.45 Налаштування Audio Source для звуків

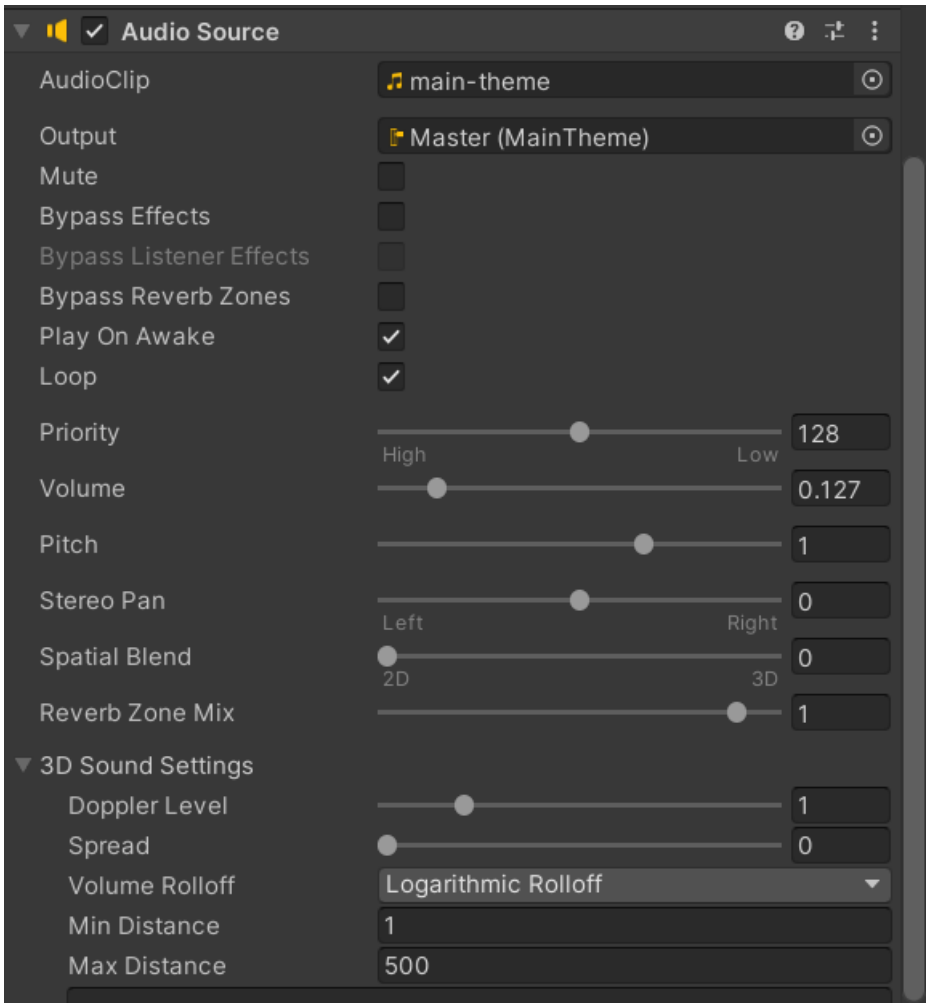

Рисунок 3.46 Налаштування Audio Source для музики

З налаштувань можна зрозуміти що музика у грі програється циклічно.

Створено клас AudioUtils, що містить методи для отримання аудіо джерел в ігровому середовищі. Він містить метод FindSfxSource, який використовується для пошуку і повернення компоненту AudioSource, призначеного для відтворення звукових ефектів в грі.

Цей метод використовує статичний метод FindWithTag, який шукає об’єкт в ієрархії сцени за тегом, який визначений у константі SfxSourceTag. Після знаходження об’єкта з відповідним тегом, викликається GetComponent<AudioSource>(), щоб отримати компонент AudioSource, який і використовується для відтворення звуків у грі.

Важливо враховувати, що метод FindWithTag повинен знаходити об’єкт з тим самим тегом, інакше виклик GetComponent<AudioSource>() може повернути

помилку, якщо об’єкт з вказаним тегом не знайдено або у ньому не міститься компоненту AudioSource.

Для програвання простих ефектів в грі створено клас PlaySFXSound, він використовується для подій, які відбуваються відразу або мають маленьку тривалість, наприклад коли гравець підбирає монету або інший предмет.

Основна функція Play цього класу викликається відповідно до подій у грі. Клас містить посилання на AudioClip – це аудіофайл, який містить звук, який треба відтворити. Коли метод Play викликається, він перевіряє, чи існує вже аудіо джерело (AudioSource). Якщо такого джерела немає, воно знаходиться чи створюється нове джерело для відтворення звуків.

Після отримання або створення джерела звуку, метод PlayOneShot використовується для програвання конкретного звукового ефекту _clip один раз.

Цей підхід здебільшого використовується для коротких звукових ефектів, що не потребують управління часом відтворення, таких як звуки підбору предметів.

Щоб програти звуки для персонажів та головного героя створено компонент PlaySoundComponent. Він містить масив звуків _sounds, який складається з об’єктів AudioData. Кожен AudioData представляє звуковий ефект, пов’язаний з конкретним ідентифікатором.

Метод Play(string id) відтворює звуковий ефект за його унікальним ідентифікатором id. Він перебирає всі звукові дані у масиві _sounds і, якщо знаходиться звук з відповідним ідентифікатором, відтворює його за допомогою PlayOneShot в аудіо джерелі _source. На рисунку 3.47 демонструється приклад використання цього компоненту.

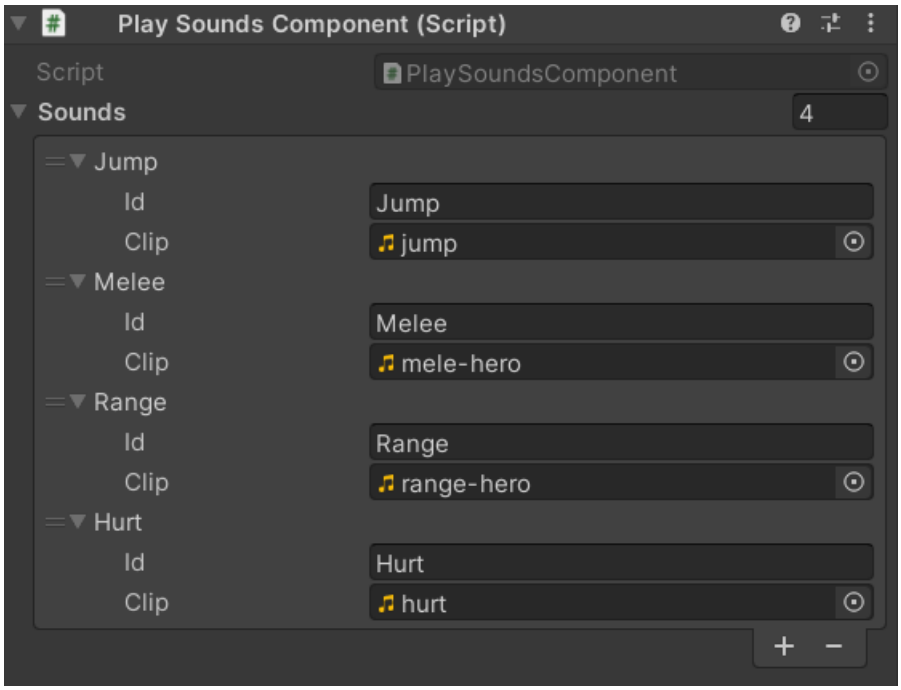

Рисунок 3.47 Приклад використання PlaySoundComponent

Щоб мати можливість регулювання рівню гучності звуків та музики було створено компонент AudioSettingsWidget. Цей компонент використовується для керування слайдером, що регулює рівень гучності, і текстовим полем, де відображується поточний рівень гучності. Він підписується на зміни гучності за допомогою паттерну спостерігача і взаємодіє з моделлю даних FloatObservableProperty, щоб відобразити та змінити значення гучності.

Також створено компонент AudioSettingsComponent, що відповідає за налаштування гучності програвача звуків та музики. Компонент також використовує FloatObservableProperty для зберігання та сповіщення про зміни рівня гучності. При старті він знаходить прив’язаний AudioSource та встановлює йому поточне значення гучності відповідно до обраного режиму (музика чи звукові ефекти).

Для зберігання рівня гучності музики та звуків у грі створено компонент GameSettings. Цей клас дозволяє доступатися до FloatObservableProperty для музики та звуків. Він завантажує дані про налаштування з ресурсів, якщо вони не були завантажені раніше, і має можливість валідувати та змінювати значення гучності.

Відповідно, для того щоб користувач мав змогу легко та зручно налаштовувати гучність створено візуальне меню для налаштування, вигляд цього вікна показано на рисунку 3.48.

Рисунок 3.48 Вигляд вікна налаштувань

За відображення цього вікна відповідає SettingsWindow. Під час старту вікна він встановлює моделі даних FloatObservableProperty для керування компонентами AudioSettingsWidget відповідно до обраних параметрів музики та звуків у GameSettings.

3.7 НПС та локалізація

У багатьох сучасних іграх існує потреба в НПС, які створюють іммерсивний світ та розширюють геймплейні можливості. НПС відіграють ключову роль у відтворенні життєвого середовища гри, надаючи гравцеві можливість взаємодіяти з віртуальним світом.

Для взаємодії гравця з об’єктами ігрового світу створено компонент InteractableComponent. Цей компонент містить подію Unity, яка спрацьовує при

взаємодії з об’єктом. Метод Interact() викликає цю подію, дозволяючи виконати певну дію, передбачену для об’єкта.

Зона взаємодії героя з об’єктами створена за допомогою вже існуючого компоненту CheckCircleOverlap, яка в цьому випадку використовується для визначення, чи перебуває герой у зоні взаємодії з об’єктами, які мають тег “Interactable”. Цей компонент відслідковує наявність об’єктів з вказаним тегом в певній області, і якщо об’єкт перебуває у зоні, він викликає метод Interact() для спрацювання відповідної події в цьому об’єкті.

Створено два типи вікон для відображення діалогів:

1. Одне вікно, що призначене для простих повідомлень, що можуть з’являтися у центрі екрану, на рисунку 3.49 показано приклад такого повідомлення.

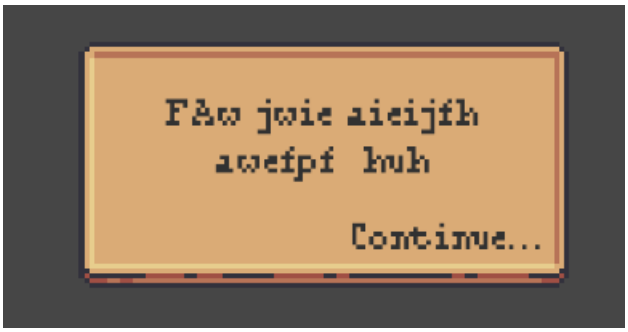

Рисунок 3.49 Просте повідомлення

2. Стилізовані вікна, призначені для реалізації справжніх діалогів між об’єктами, що включає зображення, розміщення тексту та інші елементи оформлення, на рисунку 3.50 показано приклад такого повідомлення.

Рисунок 3.50 Стилізоване вікно діалогу

Також створено структури для управління діалоговими даними у грі. DialogData є серіалізованою структурою, що містить масив речень (Sentence) і визначає тип діалогу (DialogType).

Sentence також є серіалізованою структурою та містить рядок _value, зображення _icon та сторону _side. Value повертає рядок, що містить текст речення, Icon повертає зображення, а Side вказує на сторону, з якої відображається повідомлення (ліва або права сторона).

За управління діалоговими вінками в грі відповідає клас DialogBoxController. Він має параметри контейнера, аніматора, швидкості тексту, звукові ефекти та контент діалогів.

Метод ShowDialogue відображає діалогове вікно з переданими даними. Текст діалогу виводиться посимвольно з звуковим ефектом “типографського” написання тексту.

Методи OnSkip та OnContinue викликаються при пропуску або продовженні діалогу відповідно. При завершенні діалогу, вікно приховується, і виконується подія onComplete, яка передана під час виклику ShowDialogue.

Клас DialogueDef є нащадком ScriptableObject і призначений для ефективного управління окремими діалогами в грі. Маючи атрибут CreateAssetMenu, він може створювати нові екземпляри у редакторі Unity через контекстне меню.

Цей клас містить поле _data типу DialogData, яке зберігає дані діалогу. Воно використовується для отримання доступу до даних конкретного діалогу через властивість Data.

Його основна функція полягає у зручному зберіганні та керуванні даними про діалоги, дозволяючи легко створювати та редагувати ці дані безпосередньо в редакторі Unity. На рисунку 3.51 показано приклади зберігання діалогів в грі де кожний DialogueDef це окремий діалог.

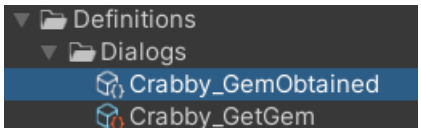

Рисунок 3.51 Зберігання діалогів в грі

Для отримання від гравця необхідних предметів створено компонент RequireItemComponent. Він призначений для створення взаємодії з ігровими об’єктами, які потребують певних предметів для активації або досягнення певного стану.

Цей компонент має поле _required, яке містить масив об’єктів типу InventoryItemData. Кожен з цих об’єктів представляє вимоги до певних предметів для виконання певної дії.

Метод Check перевіряє наявність необхідних предметів у інвентарі гравця. Якщо всі вимоги виконані, виконується подія _onSuccess, яка активує певний функціонал. Якщо ж не виконані, спрацьовує подія _onFail.

На рисунку 3.52 показано приклад використання компоненту RequireItemComponent з використанням окремих діалогів (DialogueDef).

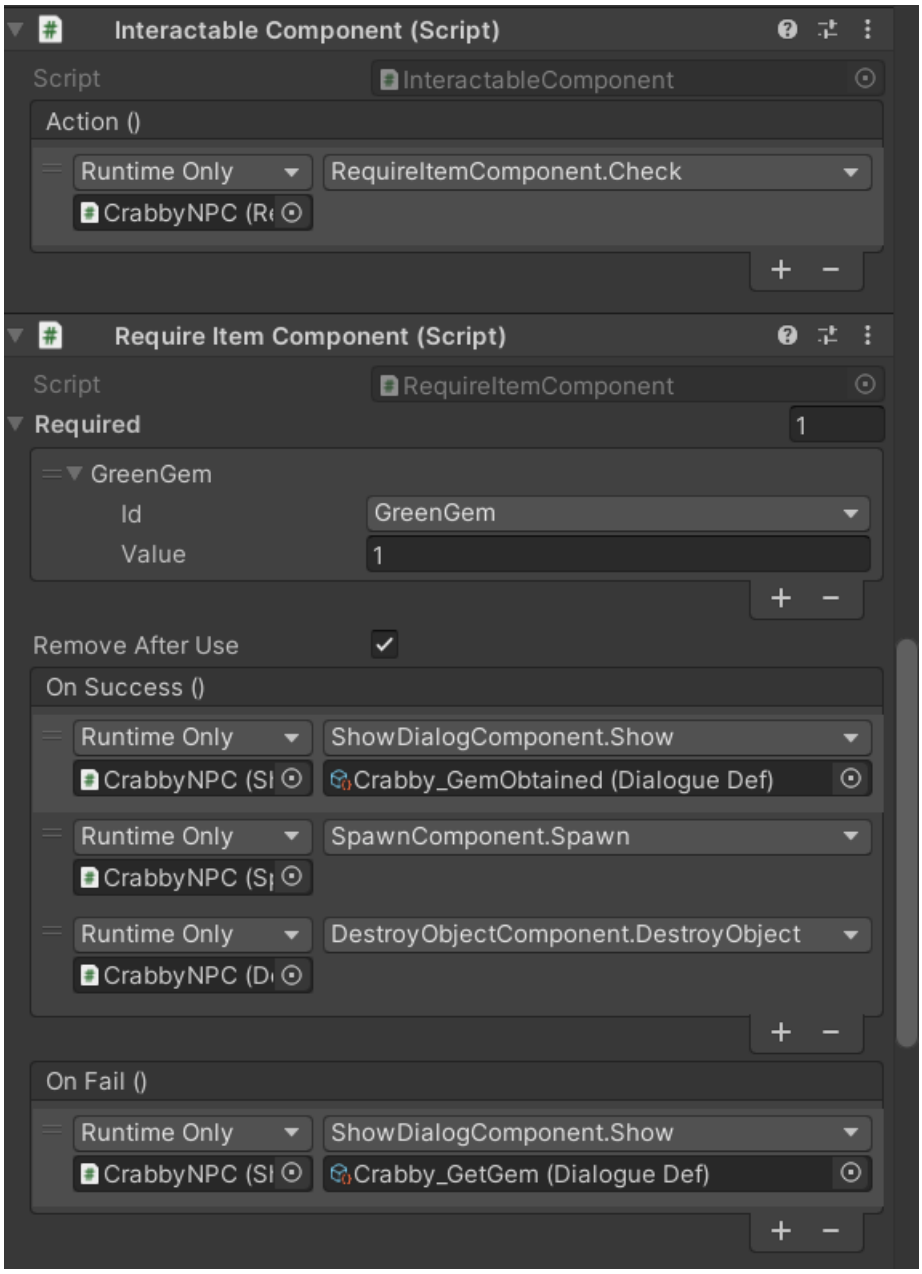

Рисунок 3.52 Використання окремих діалогів

Завдяки цьому компоненту гра стає більш інтерактивною, оскільки дозволяє створювати умови для активації різноманітних функцій або подій у грі, в залежності від наявності певних предметів у гравця.

Компонент ShowDialogComponent призначений для показу діалогів у грі та управління контролером діалогових вікон. Він надає можливість відтворювати різні типи діалогів у відповідності до вибраного режиму Mode

Компонент має поля:

– _mode: визначає режим, у якому показується діалог (з прив’язкою до даних чи зовнішніми).

– _bound: дані діалогу, прив’язані безпосередньо до компонента.

– _external: зовнішній віджет діалогу, який можна передати в якості параметра.

Метод Show відповідає за відображення діалогу на основі вибраного режиму та відповідно до наявних даних. Він викликає контролер вікон діалогу (DialogBoxController) та показує діалог.

Метод FindDialogController знаходить контролер діалогових вікон в залежності від типу діалогу (DialogType). Після цього повертає контролер для подальшого використання.

Цей компонент забезпечує зручний механізм відтворення діалогів у грі, дозволяючи вибирати тип діалогу та керувати процесом їх відображення.

На рисунках 3.53 та 3.54 показано приклад використання компоненту ShowDialogComponent для різних варіантів діалогу.

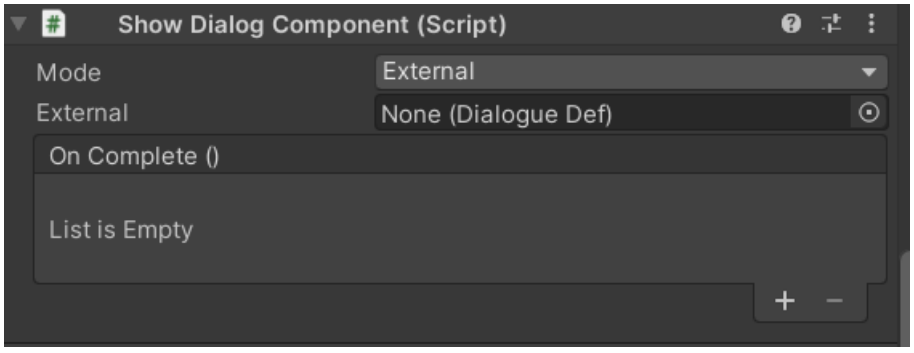

Рисунок 3.53 ShowDialogComponent для простого діалогу

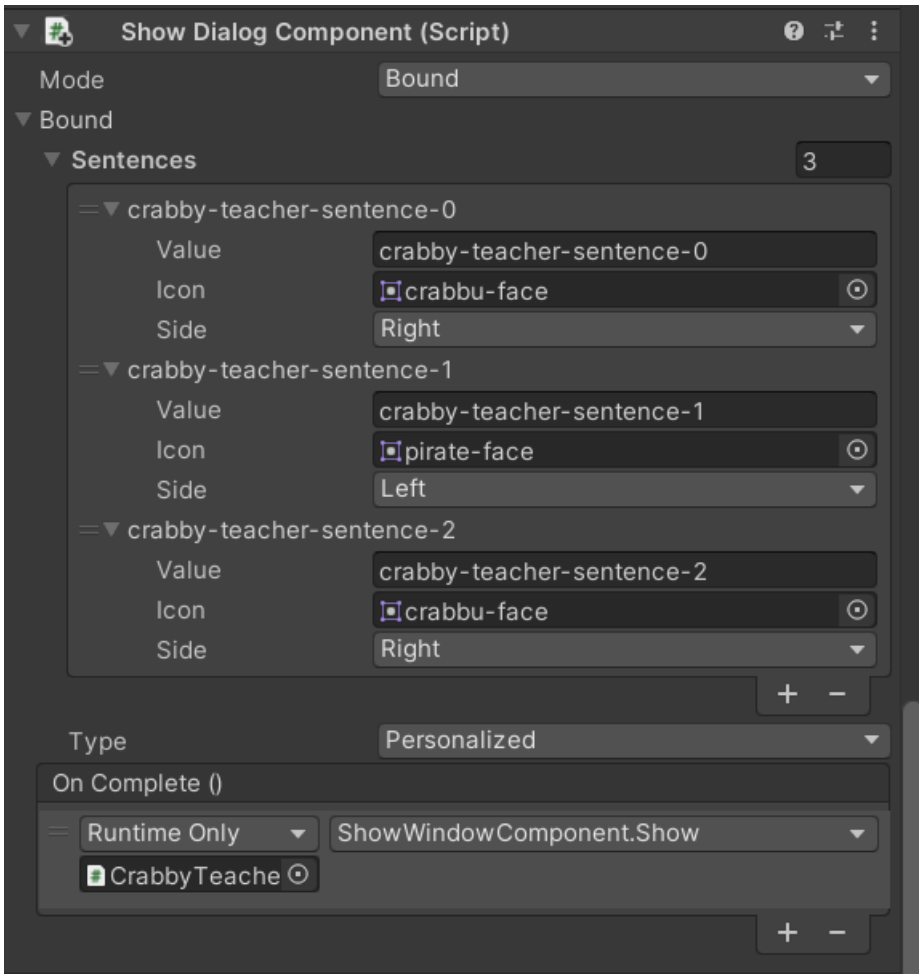

Рисунок 3.54 ShowDialogComponent для персоналізованого діалогу

Для локалізації в грі створено систему, яка дозволяє локалізовувати будь-який текст за допомогою ключа локалізації. Створений клас LocaleDef є ключовим компонентом для управління локалізацією текстових даних у грі. Він дозволяє легко та ефективно створювати та редагувати словники ключ-значення для локалізації тексту в грі. Ці ключ-значення використовуються для заміни оригінального тексту на мову яку обрав користувач.

У шаблоні усіх можливих мов є можливість завантажити дані локалізації з різних джерел:

1. З визначеного URL (відповідної Google-таблиці)
2. З файлу локалізації у форматі “tsv”

На рисунках 3.55 та 3.56 показано приклад даних з Google-таблиці та з файлу відповідно.

1	menu_title	menu
2	button_start	start
3	button_option	options
4	button_exit	exit
5	selected_language	Selected language is {0}
6	menu_pause	pause
7	button_restart	restart
8	button_back	back
9	button_locales	languages
10	Health	Health
11	Speed	Speed
12	Range Damage	Range Damage
13	level_up	Upgrade
14	shield-info	Use shield for self protection
15	crabby-teacher-sentence-0	I can teach you!
16	crabby-teacher-sentence-1	Good!
17	crabby-teacher-sentence-2	Let's go
18	crabby-gem-dialogue-0	I need a green gem

Рисунок 3.55 Приклад даних для локалізації з Google-таблиць

```
1   menu_title  menu
2   button_start    start
3   button_option   options
4   button_exit exit
5   selected_language   Selected language is {0}
6   menu_pause  pause
7   button_restart  restart
8   button_back back
9   button_locales  languages
10  Health  Health
11  Speed   Speed
12  Range Damage    Range Damage
13  level_up    Upgrade
```

Рисунок 3.56 Приклад даних для локалізації з файлу

Основні функції LocaleDef:

1. GetData: цей метод створює словник, використовуючи вже наявні значення, які можна відредагувати у редакторі.

2. LoadLocale: Метод завантажує дані локалізації з вказаного URL за допомогою UnityWebRequest, щоб отримати оновлені дані з мережі.

3. UpdateLocaleFromFile: Ця функція дозволяє легко оновлювати дані локалізації з файлу прямо у редакторі.

4. OnDataLoader та ParseData: Ці методи обробляють завантажені дані з URL або файлу розбиваючи їх на окремі рядки та заповнюючи список _localeItems.

5. Клас LocaleItem: Це вкладений клас, який містить ключ та відповідне йому значення для локалізації. Кожен LocaleItem представляє пару ключ-значення, що використовується для перекладу тексту у відповідні мови.

На рисунку 3.57 демонструється приклад створеної бази даних ключ-значення для локалізації.

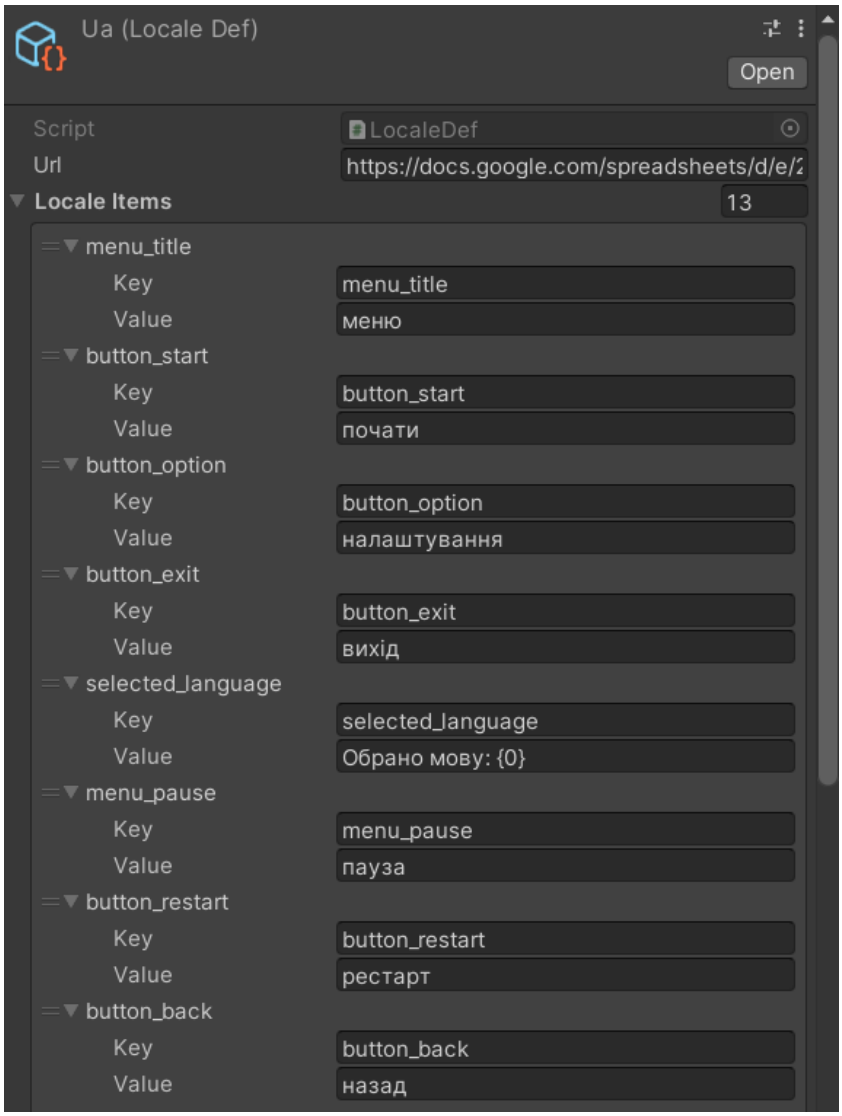

Рисунок 3.57 Приклад створеної бази даних для локалізації

Для вибору файлу локалізації створено відповідне вікно у головному меню гри. За управління вікна локалізації відповідає LocalizationWindow, він демонструє список доступних локалізацій для вибору користувачем. Під час старту встановлює список підтримуваних локалізацій (_supportedLocales) та створює список LocaleInfo для кожної локалізації. Використовується DataGroup для заповнення вмісту вікна.

Для відображення окремого елемента у вікні локалізації для кожної мови використовується LocaleItemWidget, він показує прапор та індикатор обраної локалізації та переключається відповідно до обраної мови.

Та для керування завантаженням та вибором локалізації відповідає LocalizationManager. Він має можливість завантажувати та зберігати тексти локалізації для конкретної мови. Метод Localize використовується для отримання локалізованого тексту за його ключем.

Основна логіка полягає у виборі локалізації за допомогою менеджера локалізації, який завантажує відповідні тексти для вибраної мови, а LocaleItemWidget відображає ці тексти та дозволяє вибрати іншу мову для локалізації. На рисунку 3.58 показано вигляд вікна локалізації.

Рисунок 3.58 Вигляд вікна локалізації

Щоб локалізувати окремі елементи використовуються наступні елементи:

– LocalizeImage

Цей клас відповідає за зміну зображення (прапора обраної мови) відповідно до обраної локалізації. Він отримує список IconId, який містить посилання на різні зображення для кожної мови. Коли змінюється локалізація, викликається метод Localize, який встановлює зображення на основі ключа локалізації. На рисунку 3.59 показано використання LocalizeImage.

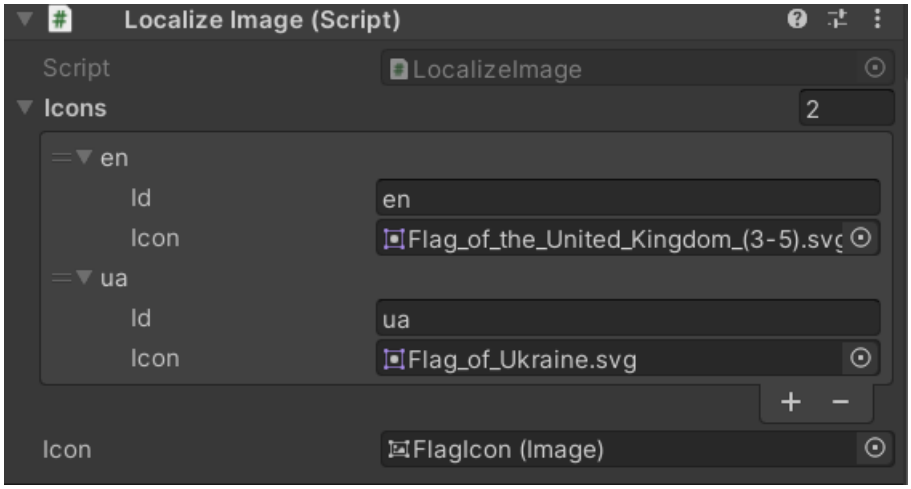

Рисунок 3.59 Використання LocalizeImage для прапорів

– LocalizeText

Цей клас призначений для локалізації текстових об’єктів. Він має поле для ключа локалізації _key, який визначає, який саме текст має бути локалізований. Під час зміни локалізації викликається метод Localize, який отримує локалізований текст з допомогою LocalizationManager та встановлює його у текстовому полі. Також є опція _capitalize, яка визначає, чи потрібно перетворити текст на великі літери. Приклад використання локалізації на об’єкті демонструється на рисунку 3.60.

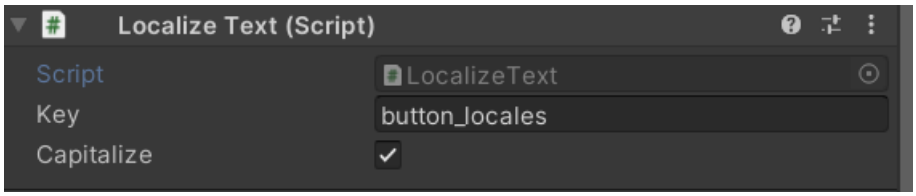

Рисунок 3.60 Використання LocalizeText

– AbstractLocalizeComponent

Абстрактний клас AbstractLocalizeComponent слугує базовим для LocalizeImage та LocalizeText. Він встановлює підписку на подію зміни локалізації OnLocaleChanged у LocalizationManager під час запуску об’єкту і відписується при знищенні об’єкту. Метод Localize в цьому класі обов’язково потрібно перевизначити у всіх його нащадках для реалізації локалізації відповідно до конкретної логіки.

– LocalizationExtensions

Цей клас відповідає за розширення Localize для тексту. Він дозволяє легко отримувати локалізований текст з допомогою рядкового ключа.

Це розширення визначає метод Localize, який приймає рядок key. Коли цей метод викликається на рядку з коду програми, він використовує LocalizationManager для пошуку локалізованого тексту за заданим ключем. Це дозволяє користувачам отримувати локалізований текст, викликаючи метод Localize безпосередньо на рядку, який потрібно локалізувати.

3.8 Покращення персонажа та тимчасові ефекти

Щоб користувач мав можливості покращення персонажа було розроблено два механізми покращення, а саме: Активні навички та покращення базових характеристик.

Для зберігання параметрів активних навичок створено PerkRepository. Він має у собі список навичок (PerkDef), кожна з яких має ідентифікатор, іконку, інформацію, ціну, та час відновлення.

Щоб користувач мав змогу бачити свою активну навичку створено CurrentPerkWidget, цей компонент відображає обрану активну навичку гравця у режимі гри. Він використовує зображення та прогрес-бар, щоб показати обрану навичку та її час відновлення. На рисунку 3.61 показано активну навичку героя.

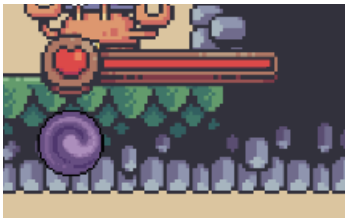

Рисунок 3.61 Демонстрація обраної активної навички

Також створено PerkWidget, цей компонент інтерфейсу, показує навички, доступні для прокачування(придбання). Він містить зображення, показує чи є навичка заблокованою або обраною, і реагує на вибір навички гравцем. На рисунку 3.62 показано вигляд вікна для придбання та вибору активних навичок.

Рисунок 3.62 Вигляд вікна для придбання та вибору навичок

За вікно управлінням навичками відповідає скрипт ManagePerksWindow, він показує список доступних навичок для прокачування. Воно показує інформацію про навичку, ціну, статус (куплено/використано), та дозволяє

купувати або використовувати навички. На рисунках 3.63 та 3.64 показано вигляд меню після придбання навички та активації відповідно.

Рисунок 3.63 Меню після придбання навички

Рисунок 3.64 Меню після вибору активної навички

Щоб показувати предмети створено скрипт ItemWidget. Цей компонент використовується для демонстрації предметів у вікні прокачування навичок. Він показує зображення предмету та його кількість.

За аналогією активних навичок було створено покращення характеристик персонажа.

Для представлення характеристик героя створено скрипт StatDef. Кожна характеристика має унікальний ідентифікатор, назву, іконку та рівні, на які можна її покращити. Кожен рівень має вартість та значення. На рисунку 3.65 показано приклад представлення характеристик.

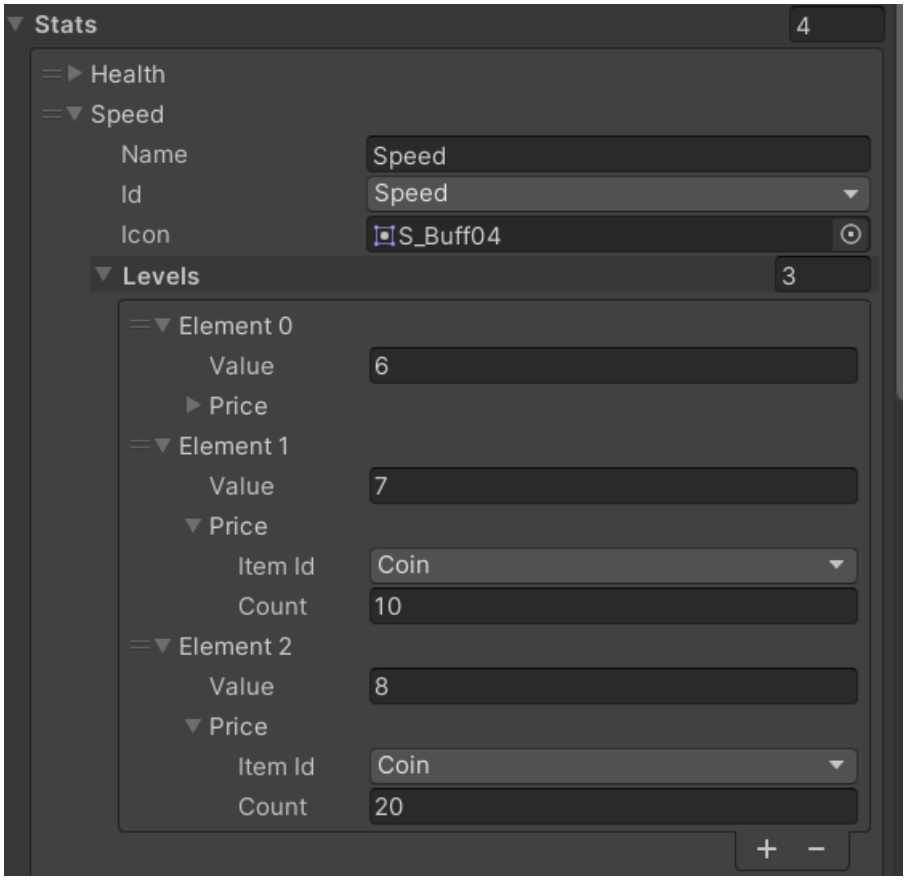

Рисунок 3.65 Представлення характеристик

Щоб відображати відповідні характеристики гравця у вікні прокачки характеристик створено StatWidget. Він показує значення поточного рівня характеристики, можливе покращення наступного рівня та вартість покращення. Також, цей компонент реагує на вибір характеристики гравцем.

Для управління вікна характеристик створено PlayerStatsWindow. Воно показує список усіх можливих характеристик, які можна прокачати, відображає їхні значення та можливості для покращення. Крім того, воно відображає вартість покращення обраної характеристики та дозволяє гравцеві покращувати

обрані характеристики, якщо персонаж має достатньо ресурсів. На рисунку 3.66 показано вигляд вікна для покращення характеристик.

Рисунок 3.66 Вікно для покращення характеристик

За управління характеристиками гравця виступає StatsModel. Він має методи для прокачки характеристик, отримання значення та рівня характеристики, а також обробників подій для покращення та змін в характеристиках. Крім того, цей клас слугує для сповіщення інших компонентів про зміни в характеристиках гравця.

Ці компоненти разом утворюють систему покращення характеристик героя у грі. Гравець може переглядати різні характеристики, обирати та покращувати їх, а система відстежує ці зміни та взаємодіє з іншими частинами гри.

Також реалізовано можливість використовувати різні еліксири в грі. Для цього використовується клас PotionRepository, який використовується для зберігання та керування еліксирами. Він є репозиторієм, де зберігаються всі доступні еліксири у грі. На рисунку 3.67 показано вигляд репозиторія для еліксирів.

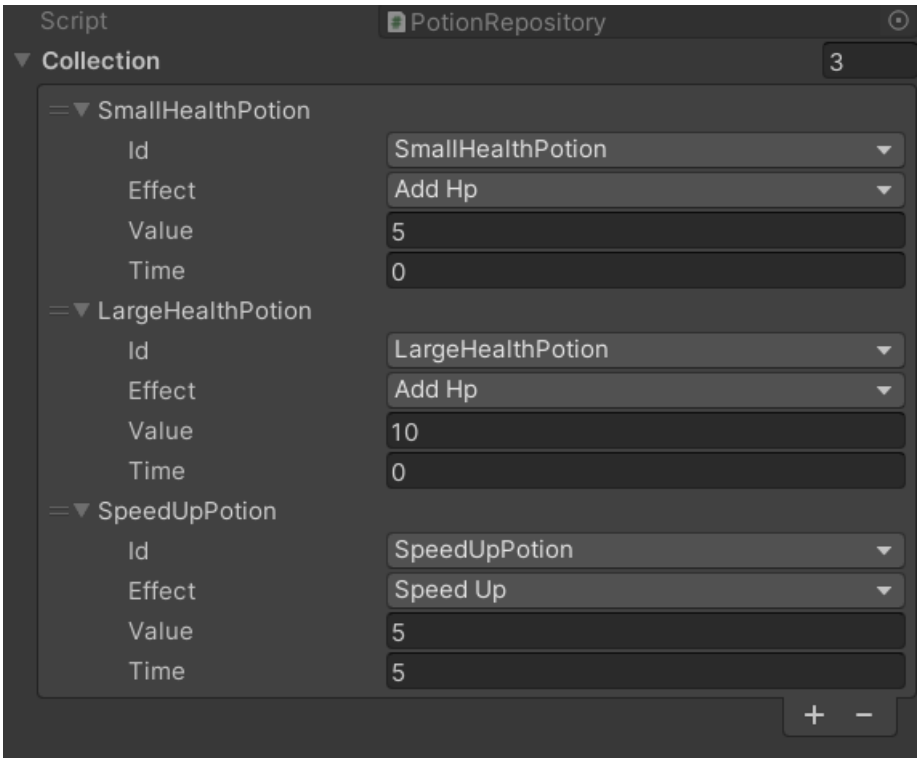

Рисунок 3.67 Вигляд репозиторія для еліксирів

Також, створено структуру PotionDef, що містить інформацію про окремий еліксир. Кожен еліксир має унікальний ідентифікатор, тип ефекту, числове значення ефекту та час дії еліксиру. Типи ефектів можуть включати додавання здоров’я та прискорення, а параметр Value та Time відповідають числовим значенням ефекту та тривалості еліксиру.

3.9 Покращення досвіду користувача та закінчення гри

Щоб покращити загальний досвід користувача від гри та поліпшенню взаємодії з користувачем розроблено відповідні аспекти гри.

Для початку було прийнято рішення про додавання до гри візуальних ефектів за допомогою URP (Universal Renderer Pipeline). Оновлення асетів до URP надає можливість взаємодії зі світлом, що створює більш реалістичну та іммерсивну графіку. Основні переваги URP полягають у підтримці новітніх

функцій світла та обробці матеріалів, що дають можливість досягти візуальних ефектів, встановлений URP показано на рисунку 3.68.

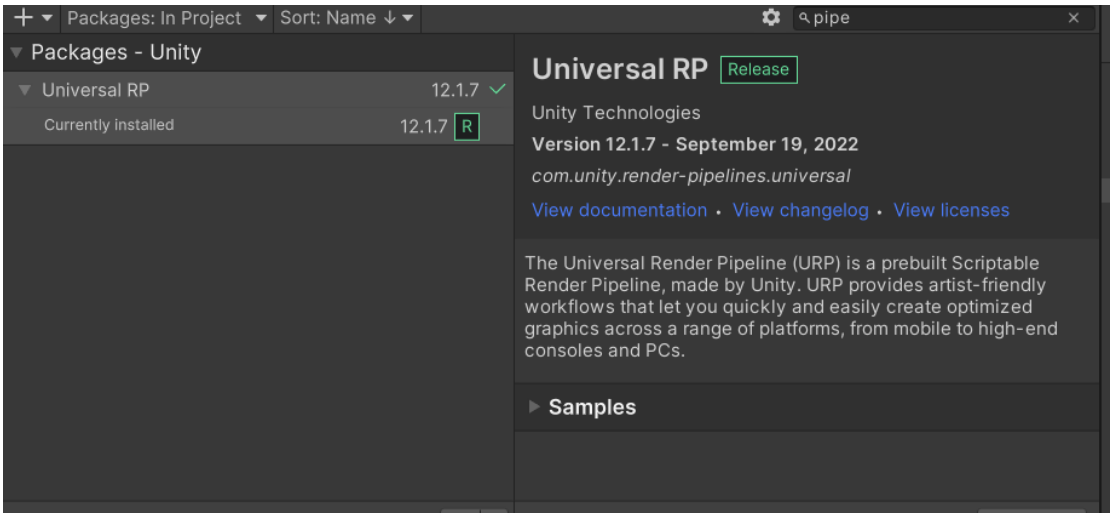

Рисунок 3.68 Встановлений URP

Також налаштовано Renderer2D Data та Universal Renderer Pipeline Asset, налаштування для показані на рисунках 3.69 та 3.70 відповідно.

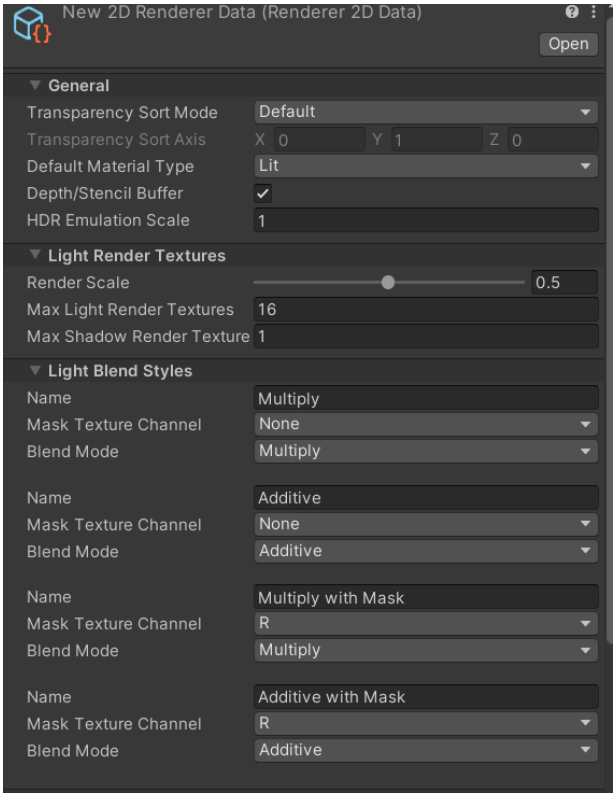

Рисунок 3.69 Налаштування для Renderer2D Data

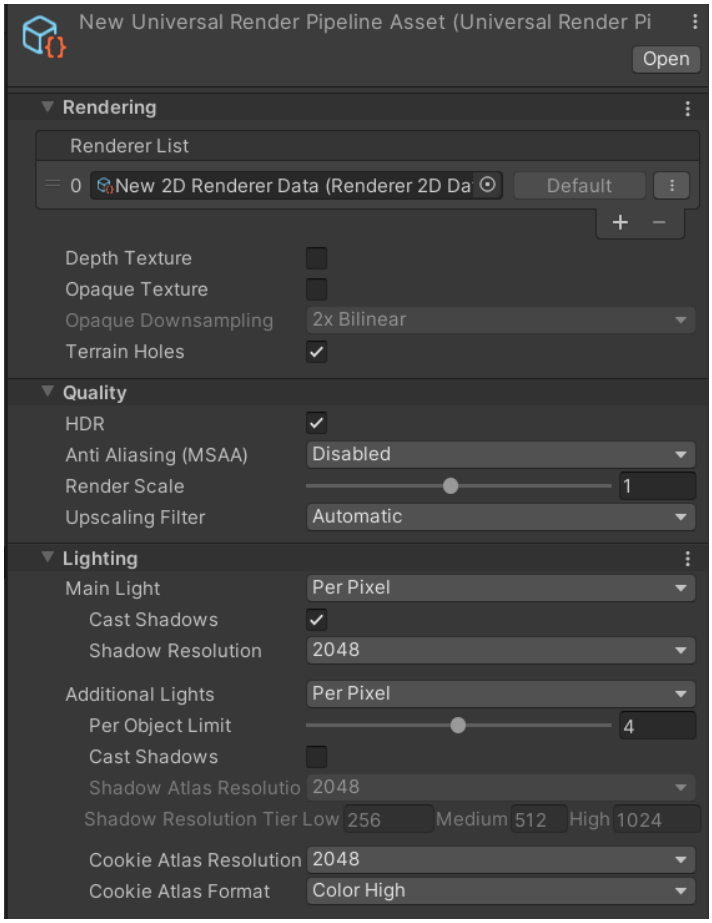

Рисунок 3.70 Налаштування для Universal Renderer Pipeline Asset

За допомогою URP було створено темний рівень, що є одним із способів додати різноманіття та візуальну динаміку у гру. Темний рівень надає атмосферу напруги та нові виклики гравцю, наприклад залучення до прихованих об’єктів у місцях з обмеженою видимістю.

Також, для НПС було додано ефект emission, відповідно, створено маску та шейдер для цього ефекту, маска та шейдер показані на рисунках 3.71 та 3.72.

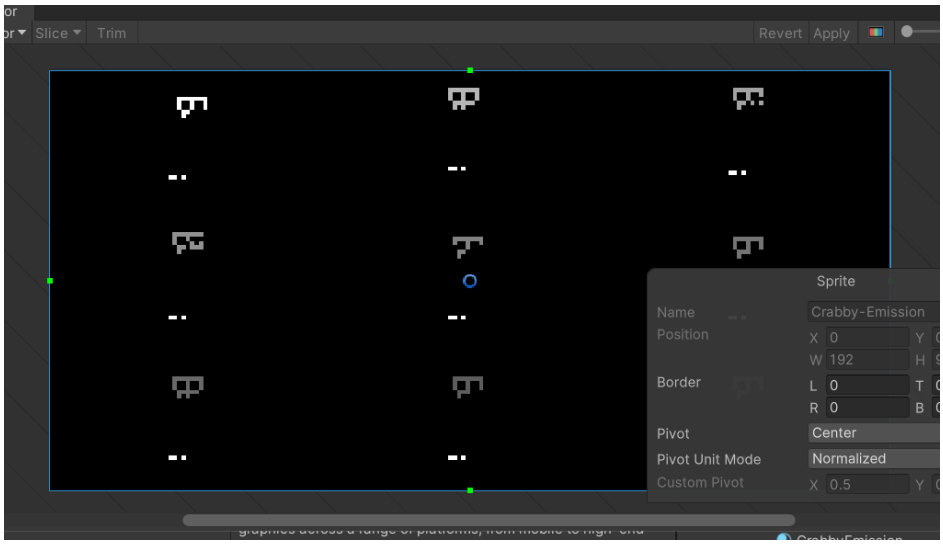

Рисунок 3.71 Маска для ефекту emission

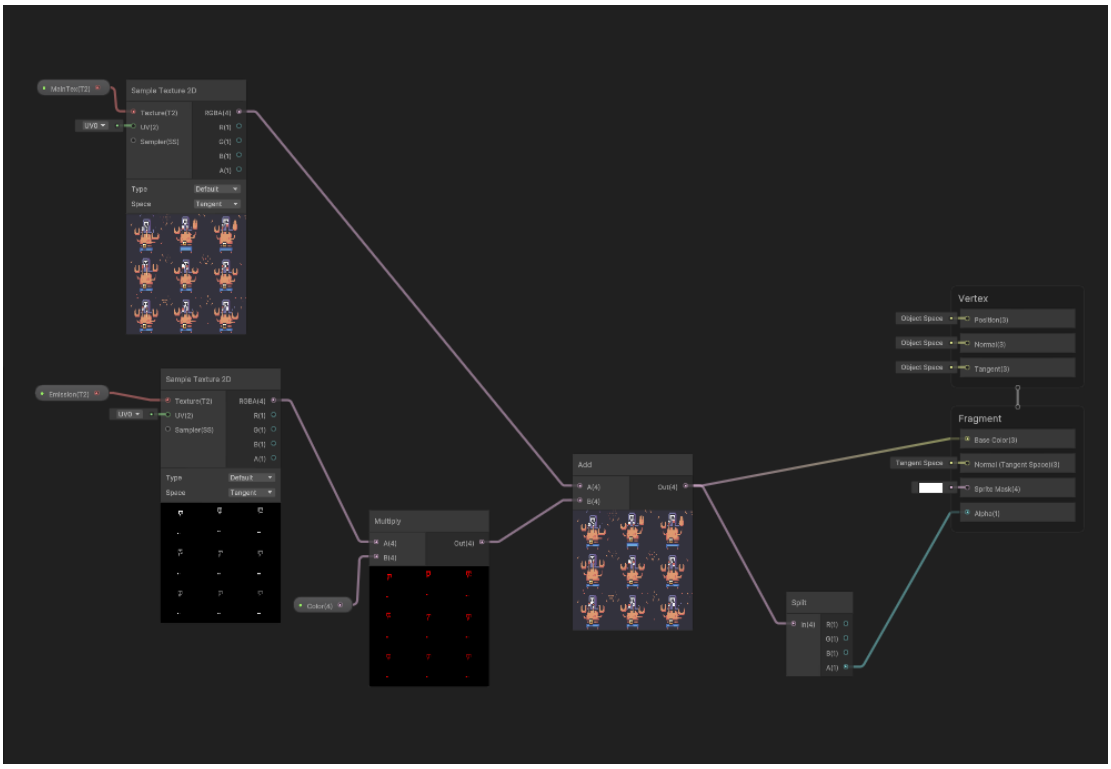

Рисунок 3.72 Створений шейдер для ефекту emission

За допомогою створеного шейдеру отримано матеріал та названо його “Crabby Emission”, що використовується для темного рівня для НПС щоб він мав ефект emission.

Типи світла відіграють ключову роль у візуальному відтворенні гри, особливо рівнів де світло відіграє значну роль, вони визначають характеристики освітлення та ефекті, які створюють атмосферу та реалістичність оточення.

У грі було створено три джерела світла:

1. Вікно: Вікно має декілька джерел світла, перше це SpotLight від самого вікна, що має радіус, а також SpriteLight що використовується щоб створити промінь світла від вікна.
2. Лампа: Вона має вогонь, що використовує тип світла SpotLight.
3. Загальне світло: Загальне світло має тип GlobalLight.

Також створено різні пост-ефекти для окремих сцен, що дозволяють змінювати візуальність гри.

1. Cave Volume Profile що використовує перевантаження Color Adjustments;
2. Global Volume Profile що має перевантаження Bloom;
3. Boss Dialog Profile що має перевантаження Bloom, Vignette та Film Grain.

Ці пост-ефекти додають глибину та настрій до візуального досвіду гравця, підсилюючи емоційні аспекти та створюючи неповторну атмосферу у різних сценах гри.

За встановлення пост-ефектів відповідає скрипт SetPostEffectProfile. Основна його функція це швидкий та зручний спосіб встановлення вказаного пост-ефекту на глобальний об’єкт, який контролює візуальні ефекти у всій сцені гри.

Щоб під час відповідних діалогів користувач не міг пересувати персонажа, створено скрипт InputEnableComponent, що відповідає за активацію/деактивацію ведення кнопок керування персонажем.

Для зміни кольору обраних джерел світла використовується створений компонент ChangeLightsComponent. Це допомагає під час відповідних дій змінити колір від світла в сцені.

Під час діалогу з босом зроблено анімацію для появи стану здоров’я ворога, за появу цього стану відповідає компонент HealthAnimationGlue. Вигляд стану здоров’я боса показано на рисунку 3.73.

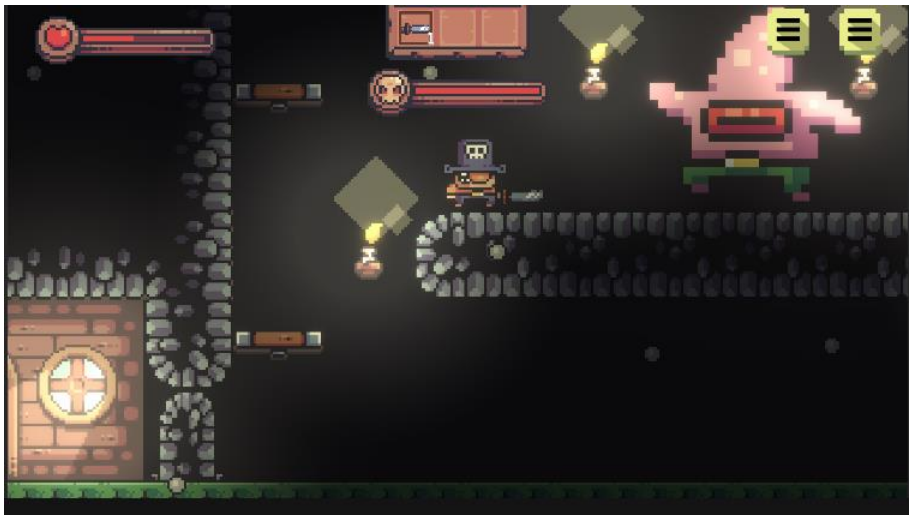

Рисунок 3.73 Стан здоров’я боса

Отримуючи посилання на HealthComponent боса та Animator, компонент підписується на зміни здоров'я (_onChange) через _hp._onChange.Subscribe(OnHealthChanged). Коли змінюється здоров'я боса, метод OnHealthChanged викликається та оновлює значення анімації для відображення стану здоров'я боса через _animator.SetInteger(Health, health), де Health - це ідентифікатор для параметра анімації. Загальний перехід анімацій для боса показано на рисунках 3.74 та 3.75.

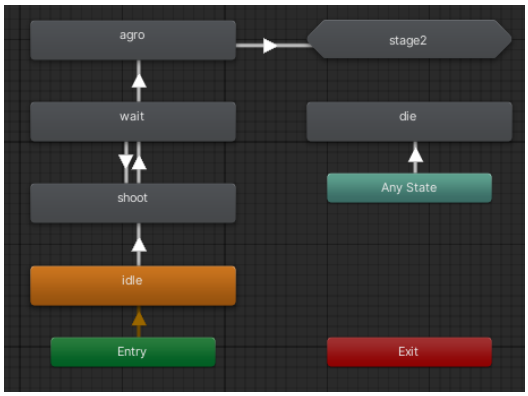

Рисунок 3.74 Перша стадія анімацій боса

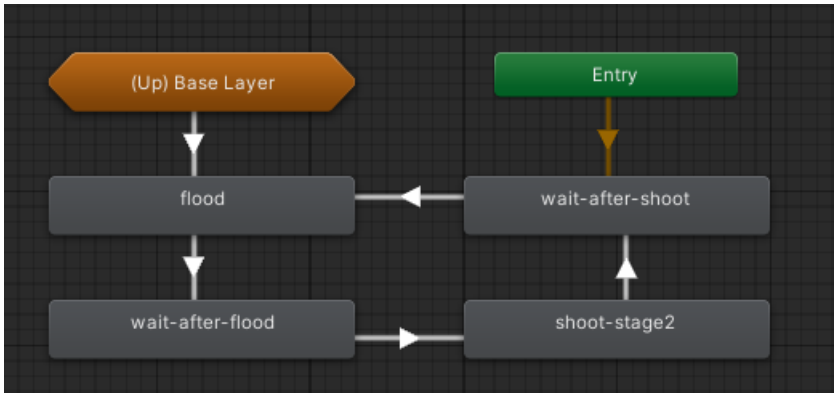

Рисунок 3.75 Друга стадія анімацій боса

Для бою з босом створено компоненти BossShootState та BossNextStageState що є реалізаціями StateMachineBehaviour. Вони використовуються для керування станами анімаційного стану боса під час бою.

Клас BossShootState викликається під час входу в конкретний анімаційний стан, який відповідає стрільбі боса. У методі OnStateEnter цього класу відбувається запуск вистрілів. В цьому випадку, використовується компонент CircularProjectileSpawner, який має метод LaunchProjectiles(), щоб запустити снаряди або проектіли боса.

Клас BossNextStageState викликається при виході з певного анімаційного стану, що відповідає переходу боса на наступний етап бою. У методі OnStateExit цього класу відбувається зміна етапу боса, наприклад, збільшення значення Stage, а також виклик методу SetColor() у компоненті ChangeLightsComponent, який, очевидно, відповідає за зміну кольору світла або освітлення на наступному етапі бою.

Обидва ці класи наслідують від StateMachineBehaviour, що дозволяє їм бути використаними в анімаційному стані боса для керування його поведінкою та діями в різних моментах бою.

Для більшого різноманіття бою з босом створено також клас BossFloodState та FloodStateController, це базове представлення атаки “затоплення” певної області.

Клас BossFloodState є реалізацією StateMachineBehaviour в Unity. У методі OnStateEnter відбувається виклик функції StartFlooding() у компоненті FloodController, що спричиняє початок процесу "затоплення".

Клас FloodController є компонентом, що контролює процес "затоплення". У ньому є Animator _floodAnimator, який керує анімацією затоплення, і float _floodTime, який визначає тривалість цього процесу.

Метод StartFlooding() запускає анімацію затоплення, а саме встановлює в Animator параметр IsFlooding у стан true. Цей метод викликається з BossFloodState для активації атаки.

IEnumerator Animate() відповідає за анімаційний процес. У ньому спочатку встановлюється параметр IsFlooding у true для запуску анімації, після чого очікується задана кількість часу (_floodTime) і параметр знову встановлюється у false, що завершує анімаційний ефект затоплення.

3.10 Висновки за розділом

У цьому розділі створено повну програмну частину для ігрового проєкту. Реалізовано систему навичок та управління статами персонажа, включено світлові ефекти для різних сцен, налаштовано управління користувача та додано компоненти для управління подіями під час боїв та інших сценаріїв у грі. Все це допомагає створити цікавий та динамічний ігровий досвід з різноманітними можливостями взаємодії гравця з ігровим середовищем.

ВИСНОВКИ

В роботі розроблено прототип гри для задоволення різних вікових категорій гравців.

В роботі розглянуто найпопулярніші жанри ігор станом на 2023 рік та популярні представники цих жанрів.

Зроблено аналіз сучасних популярних ігор які отримали найбільшу кількість позитивних рецензій від гравців у обраному жанрі та їх механік.

Обрано середовище та мову для реалізації гри, визначено візуальний стиль та звуковий супровід гри.

Розроблено та налагоджено інтерактивні системи, візуальні анімації, системи покращення персонажа, різні типи ворогів, пастки, механіки тимчасових ефектів. Впроваджено компоненти для аудіовізуальної частини, зокрема системи звукових ефектів та механіки тимчасових аудіо-візуальних ефектів. Створено систему локалізації. Реалізовано та оптимізовано світлові ефекти.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. 50+ Gamer Demographics Facts in 2023. [Електронний ресурс] – 2023. - Режим доступу до ресурсу: https://headphonesaddict.com/gamer-demographics-statistics/
2. Gamer Demographics: Facts and Stats About the Most Popular Hobby in the World [Електронний ресурс] - 2023 - Режим доступу до ресурсу: https://dataprot.net/statistics/gamer-demographics/
3. Billieux, J., Thorens, G., Khazaal, Y., Zullino, D., Achab, S., & Van der Linden, M. (2015). Problematic involvement in online games: A cluster analytic approach. Computers in Human Behavior, 43, 242–250 с.
4. Uncovering the Top 5 Types of Gamers in Today's Gaming Community [Електронний ресурс] – 2023. - Режим доступу до ресурсу: https://ganknow.com/blog/types-of-gamers/
5. What's the most popular gaming genre in 2023? [Електронний ресурс] - 2023 - Режим доступу до ресурсу: https://pinglestudio.com/blog/industry-news/whats-the-most-popular-gaming-genre-in-2023
6. Most popular genres played regularly according to video gamers in the United States in 2022 [Електронний ресурс] - 2022 - Режим доступу до ресурсу: https://www.statista.com/statistics/246766/favorite-video-game-genres-in-the-us/
7. Meet the 29 Top Video Game Genres of 2023 [Електронний ресурс] – 2023 - Режим доступу до ресурсу: https://plarium.com/en/blog/video-game-genres/
8. How Many Gamers Are There? (New 2023 Statistics) [Електронний ресурс] - 2023 - Режим доступу до ресурсу: https://explodingtopics.com/blog/number-of-gamers
9. Vampire Survivors Steam Charts [Електронний ресурс] - 2023 - Режим доступу до ресурсу: https://steamdb.info/app/1794680/charts/
10. Review: VAMPIRE SURVIVORS is So Incredibly Good You Should Go Play It Now [Електронний ресурс] - 2023 - Режим доступу до ресурсу:

https://geektyrant.com/news/review-vampire-survivors-is-so-incredibly-good-you-should-go-play-it-now

11. Celeste Steam Charts [Електронний ресурс] – 2023 - Режим доступу до ресурсу: https://steamdb.info/app/504230/charts/
12. Celeste - Metacritic [Електронний ресурс] – 2023 - Режим доступу до ресурсу: https://www.metacritic.com/game/celeste/
13. Hollow Knight Steam Charts [Електронний ресурс] – 2023 - Режим доступу до ресурсу: https://steamdb.info/app/367520/charts/
14. Hollow Knight - Metacritic [Електронний ресурс] - 2023 - Режим доступу до ресурсу: https://www.metacritic.com/game/hollow-knight/
15. Why Godot is right for you [Електронний ресурс] - 2023 - Режим доступу до ресурсу: https://godotengine.org/features/
16. UNITY ENGINE [Електронний ресурс] - 2023 - Режим доступу до ресурсу: https://unity.com/products/unity-engine
17. The most powerful real-time 3D creation tool [Електронний ресурс] – 2023 - Режим доступу до ресурсу: https://www.unrealengine.com/en-US
18. Top Game assets tagget 2D [Електронний ресурс] - 2023 - Режим доступу до ресурсу: https://itch.io/game-assets/tag-2d
19. Mossy Cavern by Maaot [Електронний ресурс] - 2023 - Режим доступу до ресурсу: https://maaot.itch.io/mossy-cavern
20. Sunny Land – Pixel Game Art [Електронний ресурс] – 2023 - Режим доступу до ресурсу: https://ansimuz.itch.io/sunny-land-pixel-game-art
21. Treasure Hunters [Електронний ресурс] – 2023 - Режим доступу до ресурсу: https://pixelfrog-assets.itch.io/treasure-hunters
22. ChipTone [Електронний ресурс] - 2021 - Режим доступу до ресурсу: https://sfbgames.itch.io/chiptone
23. OpenGameArt [Електронний ресурс] - 2023 - Режим доступу до ресурсу: https://opengameart.org/

Додаток А

Лістинг класу GameSession

```
namespace Model
{
   public class GameSession : MonoBehaviour
   {
      [SerializeField] private PlayerData _data;
      [SerializeField] private string _defaultCheckpoint;
      public PlayerData Data => _data;
      private PlayerData _save;

      private readonly CompositeDisposable _trash = new CompositeDisposable();
      public QuickInventoryModel QuickInventory { get; private set; }
      public PerksModel PerksModel { get; private set; }
      public StatsModel StatsModel { get; private set; }
      private readonly List<string> _checkpoints = new List<string>();

      private void Awake()
      {

         var existsSession = GetExistsSession();
         if (existsSession != null)
         {
            existsSession.StartSession(_defaultCheckpoint);
            Destroy(gameObject);
         }
         else
         {
            Save();
            InitModels();
            DontDestroyOnLoad(this);
            StartSession(_defaultCheckpoint);
         }
      }

      private void StartSession(string defaultCheckPoint)
      {
         SetChecked(defaultCheckPoint);

         LoadHud();
         SpawnHero();
      }

      private void SpawnHero()
      {
         var checkpoints = FindObjectsOfType<CheckPointComponent>();
         var lastCheckPoint = _checkpoints.Last();
         foreach (var checkPoint in checkpoints)
         {
            if (checkPoint.Id == lastCheckPoint)
```

```
        {
            checkPoint.SpawnHero();
            break;
        }
    }
}

private void InitModels()
{
    QuickInventory = new QuickInventoryModel(Data);
    _trash.Retain(QuickInventory);

    PerksModel = new PerksModel(_data);
    _trash.Retain(PerksModel);

    StatsModel = new StatsModel(_data);
    _trash.Retain(StatsModel);

    _data.Hp.Value = (int) StatsModel.GetValue(StatId.Hp);
}

private void LoadHud()
{
    SceneManager.LoadScene("Hud", LoadSceneMode.Additive);
}

public void Save()
{
    _save = _data.Clone();
}

public void LoadLastSave()
{
    _data = _save.Clone();

    _trash.Dispose();
    InitModels();
}

public bool IsChecked(string id)
{
    return _checkpoints.Contains(id);
}

public void SetChecked(string id)
{
    if (!_checkpoints.Contains(id))
    {
        Save();
        _checkpoints.Add(id);
    }
}
```

```
        private GameSession GetExistsSession()
        {
            var sessions = FindObjectsOfType<GameSession>();

            foreach (var gameSession in sessions)
            {
                if (gameSession != this)
                    return gameSession;
            }
            return null;
        }

        private void OnDestroy()
        {
            _trash.Dispose();
        }

        private List<string> _removedItems = new List<string>();

        public bool RestoreState(string id)
        {
            return _removedItems.Contains(id);
        }

        public void StoreState(string id)
        {
            if (!_removedItems.Contains(id))
                _removedItems.Add(id);
        }
    }
}
```

Додаток Б

Лістинг класу Creature

```
namespace Creatures
{
   public class Creature : MonoBehaviour
   {
      [Header("Params")] [SerializeField] private bool _invertScale;
      [SerializeField] private float _speed;
      [SerializeField] protected float _jumpSpeed;
      [SerializeField] private float _damageVelocity;
      [SerializeField] private int _damage;

      [Header("Checkers")]
      [SerializeField] protected LayerMask _groundLayer;
      [SerializeField] private LayerCheck _groundCheck;
      [SerializeField] private CheckCircleOverlap _attackRange;
      [SerializeField] protected SpawnListComponent _particles;

      protected Vector2 Direction;
      protected Rigidbody2D Rigidbody;
      protected Animator Animator;
      protected PlaySoundsComponent Sounds;
      protected bool IsGrounded;
      private bool _isJumping;

      private static readonly int IsGroundKey = Animator.StringToHash("is-ground");
      private static readonly int IsRunningKey = Animator.StringToHash("is-running");
      private static readonly int VerticalVelocity = Animator.StringToHash("vertical-velocity");
      private static readonly int Hit = Animator.StringToHash("hit");
      private static readonly int AttackKey = Animator.StringToHash("attack");

      protected virtual void Awake()
      {
         Rigidbody = GetComponent<Rigidbody2D>();
         Animator = GetComponent<Animator>();
         Sounds = GetComponent<PlaySoundsComponent>();
      }

      public void SetDirection(Vector2 direction)
      {
         Direction = direction;
      }

      protected virtual void Update()
      {
         IsGrounded = _groundCheck.IsTouchingLayer;
      }
```

```
private void FixedUpdate()
{
    var xVelocity = CalculateXVelocity();
    var yVelocity = CalculateYVelocity();
    Rigidbody.velocity = new Vector2(xVelocity, yVelocity);


    Animator.SetBool(IsGroundKey, IsGrounded);
    Animator.SetFloat(VerticalVelocity, Rigidbody.velocity.y);
    Animator.SetBool(IsRunningKey, Direction.x != 0);

    UpdateSpriteDirection(Direction);
}

protected virtual float CalculateXVelocity()
{
    return Direction.x * CalculateSpeed();
}

protected virtual float CalculateSpeed()
{
    return _speed;
}

protected virtual float CalculateYVelocity()
{
    var yVelocity = Rigidbody.velocity.y;
    var isJumpPressing = Direction.y > 0;

    if (IsGrounded)
    {
        _isJumping = false;
    }
    if (isJumpPressing)
    {
        _isJumping = true;

        var isFalling = Rigidbody.velocity.y <= 0.001f;
        yVelocity = isFalling ? CalculateJumpVelocity(yVelocity) : yVelocity;
    }
    else if (Rigidbody.velocity.y > 0 && _isJumping)
    {
        yVelocity *= 0.5f; //Eq to bottom line
        //_rigidbody.velocity = new Vector2(_rigidbody.velocity.x, _rigidbody.velocity.y * 0.5f);
    }

    return yVelocity;
```

```
    }

    protected virtual float CalculateJumpVelocity(float yVelocity)
    {

        if (IsGrounded)
        {
            yVelocity = _jumpSpeed;
            DoJumpVfx();
        }

        return yVelocity;
    }
protected void DoJumpVfx()
    {
        _particles.Spawn("Jump");
        Sounds.Play("Jump");
    }

    public void UpdateSpriteDirection(Vector2 direction)
    {
        var multiplier = _invertScale ? -1 : 1;
        if (direction.x > 0)
        {
            transform.localScale = new Vector3(multiplier, 1, 1);
        }
        else if (direction.x < 0)
        {
            transform.localScale = new Vector3(-1 * multiplier, 1, 1);
        }
    }

    public virtual void TakeDamage()
    {
        _isJumping = false;
        Animator.SetTrigger(Hit);
        Rigidbody.velocity = new Vector2(Rigidbody.velocity.x, _damageVelocity);

    }

    public virtual void Attack()
    {
        Animator.SetTrigger(AttackKey);
        Sounds.Play("Melee");
    }

    public void OnDoAttack()
    {
```

```
            _attackRange.Check();
            _particles.Spawn("Slash");
        }
    }
}
```

Додаток В

Лістинг класу Hero

```
namespace Creatures.Hero
{
   public class Hero : Creature, ICanAddInInventory
   {
      [SerializeField] private CheckCircleOverlap _interactionCheck;
      [SerializeField] private LayerCheck _wallCheck;

      [SerializeField] private float _slamDownVelocity;
      [SerializeField] private Cooldown _throwCooldown;
      [SerializeField] private AnimatorController _armed;
      [SerializeField] private AnimatorController _unarmed;

      [Header("Super Throw")] [SerializeField]
      private Cooldown _superThrowCooldown;
      [SerializeField] private int _superThrowParticles;
      [SerializeField] private float _superThrowDelay;

      [SerializeField] private ProbabilityDropComponent _hitDrop;
      [SerializeField] private SpawnComponent _throwSpawner;
      [SerializeField] private HeroShield _shield;
      [SerializeField] private HeroFlashlight _flashlight;

      private static readonly int ThrowKey = Animator.StringToHash("throw");
      private static readonly int IsOnWall = Animator.StringToHash("is-on-wall");

      private bool _allowDoubleJump;
      private bool _isOnWall;
      private bool _superThrow;

      private GameSession _session;
      private HealthComponent _health;
      private CameraShakeEffect _cameraShake;
      private float _defaultGravityScale;

      private const string SwordId = "Sword";
      private int CoinsCount => _session.Data.Inventory.Count("Coin");
      private int SwordCount => _session.Data.Inventory.Count(SwordId);

      private string SelectedItemId => _session.QuickInventory.SelectedItem.Id;

      private bool CanThrow
      {
         get
         {
            if (SelectedItemId == SwordId)
```

```
            return SwordCount > 1;

        var def = DefsFacade.I.Items.Get(SelectedItemId);
        return def.HasTag(ItemTag.Throwable);
    }
}


protected override void Awake()
{
    base.Awake();
    _defaultGravityScale = Rigidbody.gravityScale;
}

private void Start()
{
    _cameraShake = FindObjectOfType<CameraShakeEffect>();
    _session = FindObjectOfType<GameSession>();
    _health = GetComponent<HealthComponent>();
    _session.Data.Inventory.OnChanged += OnInventoryChanged;
    _session.StatsModel.OnUpgraded += OnHeroUpgraded;

    _health.SetHealth(_session.Data.Hp.Value);
    UpdateHeroWeapon();
}

private void OnHeroUpgraded(StatId statId)
{
    switch (statId)
    {
        case StatId.Hp:
            var health = (int) _session.StatsModel.GetValue(statId);
            _session.Data.Hp.Value = health;
            _health.SetHealth(health);
            break;
    }
}

private void OnDestroy()
{
    _session.Data.Inventory.OnChanged -= OnInventoryChanged;
}

private void OnInventoryChanged(string id, int value)
{
    if (id == SwordId)
        UpdateHeroWeapon();
}
```

```
    public void OnHealthChanged(int currentHealth)
    {
        _session.Data.Hp.Value = currentHealth;
    }

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

        var moveToSameDirection = Direction.x * transform.lossyScale.x > 0;
        if (_wallCheck.IsTouchingLayer && moveToSameDirection)
        {
            _isOnWall = true;
            Rigidbody.gravityScale = 0;
        }
        else
        {
            _isOnWall = false;
            Rigidbody.gravityScale = _defaultGravityScale;
        }

        Animator.SetBool(IsOnWall, _isOnWall);
    }

    protected override float CalculateYVelocity()
    {
        var isJumpPressing = Direction.y > 0;

        if (IsGrounded || _isOnWall)
        {
            _allowDoubleJump = true;
        }

        if (!isJumpPressing && _isOnWall)
        {
            return 0f;
        }

        return base.CalculateYVelocity();

    }

    protected override float CalculateJumpVelocity(float yVelocity)
    {
        if (!IsGrounded && _allowDoubleJump && _session.PerksModel.IsDoubleJumpSupported
&& !_isOnWall)
        {
```

```
            _session.PerksModel.Cooldown.Reset();
            _allowDoubleJump = false;
            DoJumpVfx();
            return _jumpSpeed;
        }

        return base.CalculateJumpVelocity(yVelocity);
    }

    public void AddInInventory(string id, int value)
    {
        _session.Data.Inventory.Add(id, value);
    }

    public override void TakeDamage()
    {
        base.TakeDamage();
        _cameraShake?.Shake();
        if (CoinsCount > 0)
        {
            SpawnCoins();
        }
    }
    public void Interact()
    {
        _interactionCheck.Check();
    }

    public void ChangeDoubleJump()
    {
        _allowDoubleJump = true;
    }

    public void SpawnCoins()
    {
        var numCoinsToDispose = Mathf.Min(CoinsCount, 5);
        _session.Data.Inventory.Remove("Coin", numCoinsToDispose);

        _hitDrop.SetCount(numCoinsToDispose);
        _hitDrop.CalculateDrop();
    }

    private void OnCollisionEnter2D(Collision2D other)
    {
        if (other.gameObject.IsInLayer(_groundLayer))
        {
            var contact = other.contacts[0];
            if (contact.relativeVelocity.y >= _slamDownVelocity)
```

```
        {
            _particles.Spawn("SlamDown");
        }
    }
}

public override void Attack()
{
    if (SwordCount <= 0) return;
    base.Attack();
    //_attack1Particles.Spawn();

}

private void UpdateHeroWeapon()
{
    // _animator.runtimeAnimatorController = _session.Data.IsArmed ? _armed : _unarmed;
    if (SwordCount > 0)
    {
        Animator.runtimeAnimatorController = _armed;
    }
    else
    {
        Animator.runtimeAnimatorController = _unarmed;
    }
}

public void OnDoThrow()
{
    if (_superThrow && _session.PerksModel.IsSuperThrowSupported)
    {
        var throwableCount = _session.Data.Inventory.Count(SelectedItemId);
        var possibleCount = SelectedItemId == SwordId ? throwableCount - 1 : throwableCount;

        var numOfThrows = Mathf.Min(_superThrowParticles, possibleCount);
        _session.PerksModel.Cooldown.Reset();
        StartCoroutine(DoSuperThrow(numOfThrows));
    }
    else
    {
        ThrowAndRemoveFromInventory();
    }
    _superThrow = false;
}

private IEnumerator DoSuperThrow(int numOfThrows)
{
    for (int i = 0; i < numOfThrows; i++)
```

```
    {
        ThrowAndRemoveFromInventory();
        yield return new WaitForSeconds(_superThrowDelay);
    }
}

private void ThrowAndRemoveFromInventory()
{
    Sounds.Play("Range");

    var throwableId = _session.QuickInventory.SelectedItem.Id;
    var throwableDef = DefsFacade.I.Throwable.Get(throwableId);
    _throwSpawner.SetPrefab(throwableDef.Projectile);
    var instance = _throwSpawner.SpawnInstance();
    ApplyRangeDamageStat(instance);
    // _particles.Spawn("Throw");
    _session.Data.Inventory.Remove(throwableId, 1);
}

private void ApplyRangeDamageStat(GameObject projectile)
{
    var hpModify = projectile.GetComponent<ModifyHealthComponent>();
    var damageValue = (int) _session.StatsModel.GetValue(StatId.RangeDamage);
    damageValue = ModifyDamageByCrit(damageValue);
    hpModify.SetDelta(-damageValue);
}

private int ModifyDamageByCrit(int damage)
{
    var critChance = _session.StatsModel.GetValue(StatId.CriticalDamage);
    if (Random.value * 100 <= critChance)
    {
        return damage * 2;
    }

    return damage;
}

public void StartThrowing()
{
    _superThrowCooldown.Reset();
}

public void UseInventory()
{
    if (IsSelectedItem(ItemTag.Throwable))
        PerformThrowing();
    else if (IsSelectedItem(ItemTag.Potion))
```

```
        UsePotion();
    }

    private void UsePotion()
    {
        var potion = DefsFacade.I.Potions.Get(SelectedItemId);

        switch (potion.Effect)
        {
            case Effect.AddHp:
                _session.Data.Hp.Value += (int)potion.Value;
                break;
            case Effect.SpeedUp:
                _speedUpCooldown.Value = _speedUpCooldown.RemainingTime + potion.Time;
                _additionalSpeed = Mathf.Max(potion.Value, _additionalSpeed);
                _speedUpCooldown.Reset();
                break;
        }

        _session.Data.Inventory.Remove(potion.Id, 1);
    }

    private readonly Cooldown _speedUpCooldown = new Cooldown();
    private float _additionalSpeed;

    protected override float CalculateSpeed()
    {
        if (_speedUpCooldown.IsReady)
            _additionalSpeed = 0f;

        var defaultSpeed = _session.StatsModel.GetValue(StatId.Speed);
        return defaultSpeed + _additionalSpeed;
    }

    private bool IsSelectedItem(ItemTag tag)
    {
        return _session.QuickInventory.SelectedDef.HasTag(tag);
    }

    private void PerformThrowing()
    {
        if (!_throwCooldown.IsReady || !CanThrow)
        {
            return;
        }

        if (_superThrowCooldown.IsReady) _superThrow = true;
```

```
            Animator.SetTrigger(ThrowKey);
            _throwCooldown.Reset();
        }

        public void NextItem()
        {
            _session.QuickInventory.SetNextItem();
        }

        public void DropDown()
        {
            var endPosition = transform.position + new Vector3(0, -1);
            var hit = Physics2D.Linecast(transform.position, endPosition, _groundLayer);
            if (hit.collider == null) return;
            var component = hit.collider.GetComponent<TmpDisableColliderComponent>();
            if (component == null) return;
            component.DisableCollider();
        }

        public void UsePerk()
        {
            if (_session.PerksModel.IsShieldSupported)
            {
                _shield.Use();
                _session.PerksModel.Cooldown.Reset();
            }
        }

        public void ToggleFlashlight()
        {
            var isActive = _flashlight.gameObject.activeSelf;
            _flashlight.gameObject.SetActive(!isActive);
        }
    }
}
```

Додаток Г

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ДНІПРОВСЬКА ПОЛІТЕХНІКА»

Факультет інформаційних технологій
Кафедра програмного забезпечення комп'ютерних систем

ВІДГУК

Наукового керівника Приходченко С. Д., доцента, кафедри ПЗКС
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

На кваліфікаційну роботу

студента Семенова Микити Сергійовича
(прізвище, ім'я, по батькові)

курсу IV групи 122м-22-4
спеціальності 122 Комп'ютерні науки
на тему Дослідження ігрових можливостей гравців різного віку та розробка гри з урахуванням вікових особливостей

Актуальність теми Актуальність теми визначається високим попитом на ігрові продукти, які враховують індивідуальні вікові особливості гравців різних категорій

Мета досліджень підвищення ефективності розробки ігор для задоволення більшої кількості різноманітних гравців.

Коротка характеристика розділів роботи У першому розділі проведено аналіз предметної області, де досліджується популярні жанри серед гравців різних вікових категорій. У другому розділі проведено порівняння платформ для розробки гри, обрано візуальну та музичну складову для розробки. Третій розділ описує розроблений ігровий додаток, його механіки та загальний функціонал додатку.

Практичне значення роботи Полягають у вдосконаленні використання ігрових технологій шляхом відбору найбільш оптимальних умов для задоволення різного вікового спектру гравців.

Зауваження та недоліки Наявні певні недоліки в оформлені роботи, частина інформації розкрита недостатньо

Висновки та оцінка Кваліфікаційна робота заслуговує оцінки «добре», виконавець заслуговує на присвоєння відповідної кваліфікації.

Науковий керівник Приходченко Сергій Дмитрович, професор, каф. ПЗКС
(прізвище, ім'я, по батькові, посада, місце роботи)

«_13_»___грудня___2022 р. _____
(підпис)

Додаток Д

РЕЦЕНЗІЯ
на кваліфікаційну роботу

студента Семенова Микити Сергійовича
(прізвище, ім'я, по батькові)
курсу IV групи 122м-22-4
кафедри програмного забезпечення комп'ютерних систем
спеціальності 122 Комп'ютерні науки
Тема роботи Дослідження ігрових можливостей гравців різного віку та розробка гри з урахуванням вікових особливостей

Стисла характеристика розділів роботи Перший розділ надає інформацію про предметну область та її поточний стан. В другому розділі проведена підготовка до розробки гри, обрано платформу для розробки, візуальну та музичну складову. Третій розділ містить інформацію про розроблений функціонал та механіки гри.

Пропозиції, внесені студентом, рівень їх наукового обґрунтування
Дана робота, написана згідно стандартів, використовуючи корректну термінологію. Висновки з роботи є адекватними, виведеними з дослідження проведеного протягом її написання

Практичне значення роботи полягає у створенні новітніх рішень для розробки гри, що враховує вікові аспекти гравців.

Якість оформлення роботи робота виконана у відповідності до вимог оформлення кваліфікаційних робіт магістерського рівня і відповідає поставленій задачі.
Недоліки в роботі у роботі недостатньо розкриті глибинні принципи, що відповідають за спосіб роботи кожної конкретної технології AR.
Загальний висновок отримані результати є закінченою науково-дослідною роботою і мають практичну цінність, що підтверджує здатність Семенова

Микити Сергійовича до самостійного ведення наукової роботи та вміння з проектування та розробки програмного забезпечення. Кваліфікаційна робота заслуговує оцінки "добре", а Семенов М.С. – присвоєння відповідної кваліфікації.

(підготовленість студента до самостійної роботи як спеціаліста)

Оцінка магістерської роботи "добре"

Рецензент _____

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання, посада, місце роботи)

«___»_____20__ р. _____

(підпис)