МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«До захисту допущено»
В.о. завідувача кафедри
Ігор ШЕЛЕХОВ
_____
(підпис)
« » червня 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавр

зі спеціальності 122 - Комп'ютерних наук,
освітньо-професійної програми «Інформатика»
на тему: «Візуальне та програмне забезпечення ігрової системи жанру 2D RPG у стилістиці піксельної графіки»
здобувачки групи ІН-01 Ашенафі Марії Робі

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____ Марія АШЕНАФІ
(підпис)

Керівник,
старший викладач
кафедри комп'ютерних наук,
к.т.н., доцент Борис КУЗІКОВ _____
(підпис)

Суми – 2024

Сумський державний університет

Кафедра комп'ютерних наук
«Затверджую»
В.о. завідувача кафедри
_____ Ігор ШЕЛЕХОВ
(підпис)

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавра

зі спеціальності 122 - Комп'ютерних наук, освітньо-професійної програми «Інформатика»
здобувача групи ІН-01 Ашенафі Марії Робі

1. Тема роботи: «Візуальне та програмне забезпечення ігрової системи жанру 2D RPG у стилістиці піксельної графіки» затверджую наказом по СумДУ від *«22» квітня 2024 р. № 0414-VI*
2. Термін здачі здобувачем кваліфікаційної роботи *до 29 травня 2024 року*
3. Вхідні дані до кваліфікаційної роботи
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити) *1) Аналіз проблеми предметної області, постановка й формування завдань дослідження. 2) Проектування ігровго додатку та ігрової системи. 3) Розробка візуального забезпечення ігрової системи. 5) Розробка програмної реалізації ігрової системи. 4) Аналіз результатів.*
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «____» _____ 20 ___ р.
Завдання прийняв до виконання _____ Керівник _____
(підпис) (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	*Аналіз проблеми предметної області, постановка й формування завдань дослідження*		
2	*Проектування ігрового додатку та ігрової системи*		
3	*Розробка візуального забезпечення ігрової системи*		
4	*Розробка програмної реалізації ігрової системи*		
5	*Аналіз отриманих результатів*		
6	*Оформлення пояснювальної записки до кваліфікаційної роботи*		

Здобувач вищої освіти _____ Керівник _____
(підпис) (підпис)

АНОТАЦІЯ

Записка: 59 стор., 32 рис., 2 табл., 3 додатки, 27 використаних джерел.

Об’єкт дослідження – відеогра, як складний програмний продукт, що об’єднує в собі різноманітні сучасні технології та мистецькі течії.

Предмет дослідження – теоретичні та практичні аспекти процесу розробки візуального та програмного забезпечення ігрової системи жанру 2D RPG у стилістиці піксельної графіки.

Мета роботи – створення комплексного ігрового забезпечення на основі ігрового рушія Unity, що поєднує у собі візуальну та програмну складову.

Методи дослідження – аналіз аналогів та ігрового ринку, порівняння різних методів та підходів до розробки 2D RPG, розробка та тестування прототипів гри.

Результат – візуальне та програмне забезпечення ігрової системи Moonrise жанру 2D RPG у стилістиці піксельної графіки; звіт із детальним описом процесу розробки ігор, що може бути використаний як навчальні матеріали.

Ключові слова: відеогра, RPG, game development, 2D гра, pixel art, Unity, C#.

ЗМІСТ

ВСТУП 5

1 АНАЛІТИЧНИЙ ОГЛЯД 7

1.1 Знайомство із поняттям відеогри 7

1.2 Огляд жанру 2D RPG 7

1.3 Порівняльний аналіз знакових 2D RPG 11

1.4 Перспективи розвитку ігрової індустрії у світі та в Україні ... 13

1.5 Постановка задачі 14

2 ПРОЄКТУВАННЯ ТА ДИЗАЙН 15

2.1 Проектування гри 15

2.2 Графіка та візуальна складова 18

2.3 Unity як платформа для розробки 2D ігор 21

3 РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ 23

3.1 Створення та налаштування проекту 23

3.2 Створення сцен та візуальне наповнення 24

3.3 Програмне забезпечення 29

3.4 UI та основні меню гри 39

ВИСНОВКИ 46

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 47

ДОДАТОК А 50

ДОДАТОК Б 53

ДОДАТОК В 56

ВСТУП

Актуальність. У сучасному світі з розвитком технологій все більше аспектів людської діяльності переходять у цифрові формати та віртуальну реальність. Розваги не є винятком. Серед різноманітних сервісів можна обрати як приклад відеоігри, які перетворилися на потужну індустрію та стали звичним споживчим продуктом у житті людини.

Комп'ютерні ігри – це складні програмні продукти, які поєднують у собі досягнення різноманітних сучасних технологій. Ігри мають величезний культурний, соціальний та економічний вплив на суспільство та технології. Останніми роками популярність ретро-ігор значно зросла, збільшуючи попит на 2D піксельні ігри. Тому дослідження процесу створення візуального та програмного забезпечення 2D піксельних RPG на платформі Unity є актуальним і має важливе практичне значення як для розвитку ігрової індустрії, так і для підготовки фахівців з розробки ігор.

Об'єктом дослідження роботи є відеогра, як складний програмний продукт, що об'єднує в собі різноманітні сучасні технології та мистецькі течії. **Предмет дослідження** – теоретичні та практичні аспекти процесу розробки візуального та програмного забезпечення ігрової системи жанру 2D RPG у стилістиці піксельної графіки.

Методами дослідження є аналіз аналогів та ігрового ринку, порівняння різних методів та підходів до розробки 2D RPG, розробка та тестування прототипів гри.

Мета та завдання роботи полягають у вивченні процесу розробки відеоігор, історії їх виникнення та поширення; аналіз впливу комп'ютерних ігор на технологічний і соціальний розвиток, а також дослідження типи технологій і програмного забезпечення, що використовуються при розробці відеоігор; отримати кінцевий готовий програмний проект, що становить собою відеогру із двомірною піксельною графікою.

Практична значимість. Результати дослідження можуть бути використані для створення навчальних програм та курсів, які допоможуть новачкам гейм-деву отримати базові знання у розробці ігор. Дослідження також сприяє популяризації українських ігор на міжнародній арені та може підвищити імідж України як країни з високим рівнем розвитку ігрової індустрії.

Структура. Кваліфікаційна дипломна робота складається з аналітичного огляду, постановки задачі, вибору програмних засобів для реалізації поставленої мети, проектування візуального та програмного забезпечення, реалізації та тестування гри, висновків, списку використаних джерел у роботі та додатків.

1 АНАЛІТИЧНИЙ ОГЛЯД

1.1 Знайомство із поняттям відеогри

Відеогра – це розвага, у яку грають завдяки аудіовізуальному апарату і яка може базуватися на історії [11]. Вона зазвичай складається з програмного забезпечення, яке генерує зображення та звук, а також з апаратного забезпечення, яке відображає ці зображення та звук на екрані або іншому пристрої.

Існує безліч різних типів відеоігор, які можна класифікувати за різними критеріями, такими як жанр, платформа, аудиторія тощо. Деякі з найпопулярніших жанрів відеоігор включають екшн, пригоди, рольові, стратегії, симулятори, головоломки та багатокористувацькі онлайн-ігри.

Відеоігри можуть бути як розважальними, так і навчальними. Вони використовуються для розвитку навичок, таких як координація рук і очей, вирішення проблем та критичне мислення. Чимала кількість відеоігор створена для навчання людей новій інформації та навичкам, або для симуляції реальних ситуацій.

1.2 Огляд жанру 2D RPG

1.2.1 Історія розвитку жанру

Role-playing game (RPG; рольова гра) – це термін, який охоплює низку форм і стилів ігор, які певним чином передбачають створення, представлення та розвиток персонажів, які взаємодіють у вигаданому світі за системою структурованих правил.

Перші 2D RPG виникли як додаткові елементи в іграх інших жанрів. З часом вони стали окремими іграми, які мали свої характерні особливості. Загальними рисами 2D RPG були відкритий світ, система розвитку персонажа (наприклад, збільшення рівня персонажа, здібностей, отримання нового обладнання тощо), а також розгалужена сюжетна лінія, яка дозволяла гравцеві взаємодіяти з ігровим світом та впливати на його розвиток.

Провідною передумовою для виникнення жанру 2D RPG була поява комп'ютерних технологій, які дозволили створювати імітовані ігрові світи зі зручним інтерфейсом для користувачів. Розвиток графіки, звуку та обробки даних дало змогу реалізувати багато ідей, які раніше були неможливими в комп'ютерних іграх.

Початкові 2D RPG, які з'явилися в 80-х роках, використовували прості піксельні спрайти та тайлові мапи. Графіка була обмеженою технологіями того часу, але це не заважало створювати захоплюючі ігри з глибоким сюжетом та складними системами бойової механіки та розвитку персонажів. Одними із культових навіть до сьогодення іграми на той момент стали такі, як The Legend of Zelda (1986) та Final Fantasy (1987).

З появою 3D-графіки в кінці 90-х років популярність 2D RPG почала спадати. Однак жанр не зник зовсім. Сучасна еволюція 2D RPG відзначається поєднанням класичного стилю з новаторськими рішеннями у геймдизайні та візуальному виконанні. Інді-розробники активно використовують художній стиль та ретро-естетику, поєднуючи їх з сучасними тенденціями геймдизайну.

1.2.2 Аналіз візуальних та програмних аспектів

2D RPG мають ряд особливостей, які відрізняють їх від інших жанрів відеоігор:

- Вид збоку: 2D RPG зазвичай використовують вид збоку, щоб зобразити світ гри та персонажів. Це дає гравцям чітке уявлення про те, що відбувається на екрані, і полегшує дослідження світу.
- Класична система бою: 2D RPG зазвичай використовують покрокову систему бою, де гравці та вороги роблять ходи по черзі. Це дає гравцям час спланувати свої дії та використовувати стратегію для перемоги.
- Розгадування головоломок: 2D RPG часто містять головоломки, які гравцям потрібно вирішити, щоб просунутися в грі. Це може бути все, від простих логічних завдань до складних лабіринтів.

- Нелінійні історії: 2D RPG часто пропонують нелінійні історії, де гравці можуть приймати рішення, які впливають на сюжет. Це дає гравцям відчуття контролю над історією та робить кожне проходження гри унікальним.

Хоча здебільшого у RPG частіше використовується класична покрокова система бою, рольові ігри не обмежені тільки нею [20]. Система бою відіграє ключову роль у залученні гравця до ігрового процесу.

- Покрокова система бою: базується на чергуванні ходів гравця та противника, роблячи акцент на стратегічному плануванні дій. Прикладом такої системи є перші частини Final Fantasy;
- система бою в режимі реального часу: динамічні бої, що вимагають швидкої реакції та спритності гравця. Найпопулярніший варіант системи бою у сучасних іграх;
- змішані системи бою: поєднують елементи покрокового та реального часу, надаючи гравцю більше контролю над діями;
- автоатака з елементами стратегії: гравець віддає накази персонажам, але бої відбуваються автоматично з можливістю застосування спеціальних здібностей. Прикладом слугує Pokemon;
- системи бою, засновані на навичках: успіх бою залежить від оволодіння персонажами різноманітними навичками та їх грамотного застосування.

Багато ігор використовують комбінації різних типів систем бою. Це робить геймплей більш різноманітним та цікавим для гравців. Важливо забезпечити збалансованість системи бою, щоб усі сторони (гравець, противники) мали рівні шанси на перемогу, уникаючи ситуації переважання однієї сторони.

User interface (UI; інтерфейс користувача) – це елементи гри, де інформація передається між користувачем і комп’ютером. UI включає у себе як інструменти візуального відображення на екрані, так і будь-які фізичні контролери, що використовуються для керування грою [12].

Ефективний UI чітко й інтуїтивно доносить інформацію, необхідну для прийняття рішень, управління персонажами та навігації по ігровому світу, не захаращуючи екран і не відволікаючи від ігрового процесу. Окрім загальних принципів, UI у RPG також має жанрові особливості. Наприклад, існують три основних напрямки RPG, які мають характерні риси користувацького інтерфейсу.

JRPG (Japan role-playing game) – японська рольова гра. Хоча JRPG дійсно має коріння в японській ігровій індустрії сучасні JRPG не є суто японськими продуктами, а радше міжнародними творами. Зазвичай використовують покрокову бойову систему з акцентом на стратегії та командній роботі [18, 20].

UI цього жанру часто використовує стилізовані елементи, такі як аніме-стилістика, химерні шрифти та яскраві кольори. Меню часто розділені на багато вкладок, щоб вмістити велику кількість інформації, необхідної в JRPG: персонажі, знаряддя, навички, елементи інвентарю тощо. UI бою зазвичай використовує систему меню, де гравець вибирає дії з списку.

WRPG (Western role-playing game) – жанр RPG, що зародився у Північній Америці та Європі. Має багато особливостей, котрі лягли в основу більшості сучасних рольових ігор [20].

Користувацький інтерфейс, як і сам жанр, часто використовує більш реалістичний дизайн, з чіткими шрифтами, приглушеними кольорами та текстурами, що імітують реальні матеріали. UI зазвичай більш функціональний, з акцентом на надання інформації та виконання дій швидко та ефективно. Часто додається панель швидкого доступу, де гравець може розмістити часто використовувані дії та предмети, а також активно використовуються контекстні меню, що з'являються при наведенні курсора на певні елементи інтерфейсу.

CRPG (Computer role-playing game) – рольова гра, що використовує текстовий інтерфейс. Головна особливість CRPG – занурення в атмосферу завдяки текстовому опису [20].

CRPG використовують командний рядок, де гравець вводить команди для виконання дій. Інвентар також описаний за допомогою тексту. Часто присутня карта, яка відображає поточне місце розташування гравця та доступні маршрути.

Окрім жанрових особливостей, UI у RPG також може залежати від платформи. Наприклад, UI мобільних RPG часто буде більш простим та оптимізованим для сенсорного екрану, тоді як UI ПК-RPG може бути більш складним та деталізованим [15].

1.3 Порівняльний аналіз знакових 2D RPG

2D RPG ігри є популярним жанром, що пропонує гравцям захоплюючі історії, динамічний геймплей та дослідження віртуальних світів. Brain Breaker, Metroid та Dead Cells – це три приклади 2D RPG, які здобули визнання завдяки своїм унікальним характеристикам. Їхній огляд представлено у таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика Brain Breaker, Metroid, Dead Cells

Характеристика	**Brain Breaker**	**Metroid**	**Dead Cells**
Мова програмування, ігровий рушій	C++, власний ігровий рушій	Assembly, власний ігровий рушій	C++, Unity
Кольорова палітра та стилізація	Темна похмура кольорова палітра для підкреслення таємниці та напруги; реалістична стилізація	Яскрава, барвиста кольорова палітра, що створює відчуття пригод та досліджень; у стилі мультиків	Брудна та тьмяна кольорова палітра, похмура атмосфера; коміксна стилізація

Характеристика	**Brain Breaker**	**Metroid**	**Dead Cells**
Спрайти та анімації	Реалістичні 3D-моделі та текстури	2D-графіка; покадрова анімація	Векторна графіка та програмний рендеринг; скелетна анімація
Система бою	Покрокова	У режимі реального часу	У режимі реального часу з елементами roguelike
Система прокачування	Заснована на досвіді	Заснована на предметах та здібностях, що дозволяє гравцеві отримувати нові можливості та досліджувати нові території	Заснована на зброї та екіпіровці, що робить акцент на експериментах та пошуку кращого обладнання
Ігровий світ	Нелінійний світ, який можна досліджувати в будь-якому порядку	Відкритий світ	Відкритий світ з елементами roguelike

Порівняльна таблиця демонструє різноманіття підходів до дизайну 2D RPG. Кожна з цих ігор пропонує унікальний досвід у жанрі 2D RPG. Brain Breaker зосереджується на атмосфері та стратегії, Metroid на дослідженні та здібностях, а Dead Cells на екшені та roguelike елементах. Розробники можуть обирати з різних варіантів мови програмування, ігрового рушія, візуального стилю, системи бою, системи прокачування та структури світу, щоб створити унікальні та захоплюючі ігрові досвіди.

Важливо зазначити, що це лише деякі з аспектів, які слід враховувати при розробці 2D RPG. Існує безліч інших факторів, які можуть вплинути на дизайн гри, таких як сюжет, персонажі, музика та звукові ефекти.

1.4 Перспективи розвитку ігрової індустрії у світі та в Україні

Індустрія комп’ютерних ігор є однією з найдинамічніших та найшвидше зростаючих галузей у світі. Індустрія комп'ютерних ігор має значний вплив на економіку, культуру та суспільство. Вона створює робочі місця, стимулює інновації та надає розваги мільйонам людей у всьому світі. За оцінками аналітичної компанії Newzoo, яка є лідером у відеоігрових та геймерських даних, у 2024 році ігровий ринок зріс на 1,6%, що досягає позначки у $185,6 мільярдів [19].

Українська ігрова індустрія також активно зростає протягом останніх років. У 2020 році під час пандемії COVID-19 через різке обмеження контакту між людьми, зросло значення комп’ютера не тільки для зв’язку та роботи, а й для розваг. За результатами досліджень фірми NielsenIQ ринок відеоігор в Україні за 2020 рік виріс більш ніж на 20% [3].

Український ринок відеоігор стає все більш цікавим для світових ігрових монополістів. Експерти відзначають стрімке зростання інтересу до українських розробників та потенціалу нашого ринку. Зокрема, акцентується увага на тому, що партнери України вже визнають нашу країну як центр креативних ідей. Яскравим прикладом цього є співпраця з компанією Ubisoft, що посідає 12 місце у світовому рейтингу розробників відеоігор та яка має розробників в Україні [2].

Розвиток ігрової індустрії в Україні стає не лише джерелом розваг, але й потужним інструментом для модернізації економіки та посилення міжнародних зв'язків. Ця сфера відкриває нові горизонти для розвитку інвестиційного, фондового та банківського секторів країни. Інвестиції в гейм-індустрію України не лише сприятимуть її власному розвитку, але й стануть рушієм модернізації економіки загалом, роблячи країну більш конкурентоспроможною на світовому ринку.

1.5 Постановка задачі

Основною метою даного дослідження є розробка ігрового додатку «Moonrise» жанру двовимірної рольової гри для комп’ютерної платформи Windows у стилістиці піксельної графіки на базі ігрового рушія Unity.

Таким чином, було сформовано наступні вимоги для гри:

- гра відповідає жанру RPG;
- у грі присутній гравець, вороги, NPC, зброя;
- простота управління;
- наявність якісного звукового супроводу;
- дизайн відповідний темі гри;
- інтуїтивний UI;
- сенс гри – дізнатися історію головного героя та розібратися у загадках ігрового світу.

Для досягнення мети проекту необхідно виконати наступні задачі:

- визначити актуальность роботи, цільову аудиторію та дослідження предметної області;
- створити концепцію відповідно до ідеї;
- розробити ігрові механіки;
- створити дизайн гри;
- підібрати аудіоресурси, які будуть використані всередині гри;
- розробити анімації;
- обрати технологію для розробки;
- створити програмні скрипти, що відповідають ігровим механікам прототипу;
- виконати білд гри відповідно до платформи релізу;
- протестувати ігровий додаток.

2 ПРОЄКТУВАННЯ ТА ДИЗАЙН

2.1 Проектування гри

2.1.1 Концепція

Створення відеоігор, подібно до будь-якого творчого процесу, розпочинається з ідеї. Ця ідея може виникнути спонтанно або ж бути плодом зусиль спеціально залучених фахівців. Її основою може стати захоплююча історія, унікальний ігровий процес або ж інноваційна система взаємодії гравця з віртуальним світом.

Навіть за відсутності унікального ігрового процесу або вражаючої графіки, захоплюючий сюжет може занурити гравців у віртуальний світ. Добре пророблений сюжет може створити емоційний зв'язок між гравцем і грою, зробити гру запам'ятовуваною. У результаті роботи було скомпоновано наступний сюжет.

Історія відбувається у світі, де таємні наймані спеціалісти вберігають спокій звичайних людей полюючи на різноманітну нечисть. Всі вони працюють на одну організацію, отримуючи завдання від різномісних операторів.

Сюжет починається, коли гравець прибуває у провулок, де на нього вже чекає Кіт. Від Кота герой отримує завдання на сьогодні – зловити невелику нечисть-аномалію, що порушила спокій міста. Це завдання включає в себе вирушення на місце останнього спалаху активності монстра, поблизу провулку.

Гравцеві дають вказівки, що для роботи буде достатньо спеціального пістолету, який він вже має. Також, було б непогано, якщо гравець по дорозі до кінцевої цілі прибере декілька дрібних сутностей, які утворилися як наслідок активності основного монстра.

Пройшовши всі сцени до останньої, гравець опиняється у залі церкви, де бачить вже знищену аномалію. Його цікавить, хто саме це зробив і з якою метою, та вирішує розібратися із дивною ситуацією.

Маючи короткий сюжет, що покладений в основу гри, є можливість проаналізувати ключові події та інструменти. Проте в ході роботи над дипломним проектом виникла необхідність переглянути та частково відмовитися від деяких ідей. Це було зумовлено обмеженими часовими ресурсами та браком знань, необхідних для реалізації певних задумів.

Зокрема, на початковому етапі розробки розглядалась механіка звичайних атак та зарядженої ultimate атаки для битви з фінальною аномалією. Проте, з метою оптимізації процесу та пришвидшення розробки, було прийнято рішення представити проект у вигляді демоверсії. Це дозволило сконцентрувати увагу на ключових аспектах гри та закласти фундамент для її подальшого розвитку.

2.1.2 Ігрові механіки

Ігровий процес проекту ґрунтується на наступних механіках:

1. Лінійний прогрес: гравець проходить рівні послідовно, один за одним, без можливості повернення на попередні. Це створює відчуття динамічного розвитку подій та постійного руху вперед.
2. Перероджeння: після смерті персонажа гравець з'являється на початку гри. Ця механіка додає грі елемент складності та змушує гравця ретельно продумувати свої дії.
3. Стрільба: основним способом атаки є стрільба. Гравець може стріляти в різні напрямки, використовуючи ліву клавішу миші.

Лінійний прогрес у поєднанні з механікою перероджeння роблять гру більш динамічною та жорсткою, змушуючи гравця постійно вчитися та вдосконалюватися. Стрільба як основний спосіб атаки робить геймплей простим та зрозумілим.

2.1.3 Моделювання варіантів використання

Моделювання варіантів використання є важливим етапом проектування програмного забезпечення, що дозволяє чітко і візуально представити функціональні можливості системи та взаємодію з нею користувачів. UML (Unified Modeling Language) – це уніфікована мова моделювання, що

використовується для візуального опису та проектування систем, в основному програмних [16].

Спираючись на технічне завдання, визначаємо актора та варіанти використання в системі ігрового додатку «Схід місяця».

Актори в системі:

- гравець-користувач.

Варіанти використання в системі:

- почати гру: перехід на першу сцену гри із запуском ігрового процесу;
- налаштування звукових ефектів: змінює гучність звукового наповнення;
- налаштування музики: змінює гучність музичного супроводу;
- вихід із гри: вихід з ігрового додатку із завершенням роботи додатку;
- пауза: зупинення ігрового процесу із збереженням поточного прогресу;
- вихід до головного меню: завершення ігрового процесу та повернення на сцену головного меню.

UML-діаграму варіантів використання ігрового додатку зображено на рисунку 2.1.

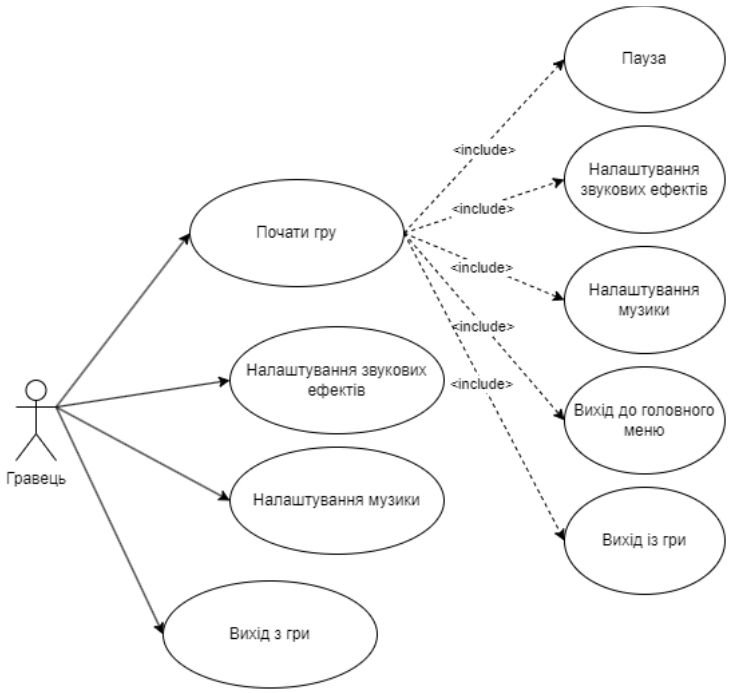

Рисунок 2.1 – Діаграма варіантів використання UML

2.2 Графіка та візуальна складова

2.2.1 Технічні особливості

Піксельарт (pixel art) – це техніка створення цифрових зображень, де кожен піксель на екрані має чітко визначене значення та колір. Ця техніка часто використовується в ретро-іграх, 2D-ілюстраціях та анімаціях завдяки своїй простоті, візуальній чарівності та можливості створювати чіткі та виразні зображення з обмеженою палітрою кольорів [17].

Піксельарт, який з'явився при зародженні цифрової графіки (рис. 2.2), пройшов шлях від технологічної необхідності до самостійного жанру (рис. 2.3). Незважаючи на стрімкий розвиток графічних технологій, він не втрачає актуальності. На сьогодні піксельарт визначає унікальний стиль та атмосферу через обмежену кольорову палітру, блокову анімацію та виразність. Піксельні зображення можуть здатися примітивними з першого погляду, проте у сучасності такий спосіб зображення виділяється у окремий вид мистецтва.

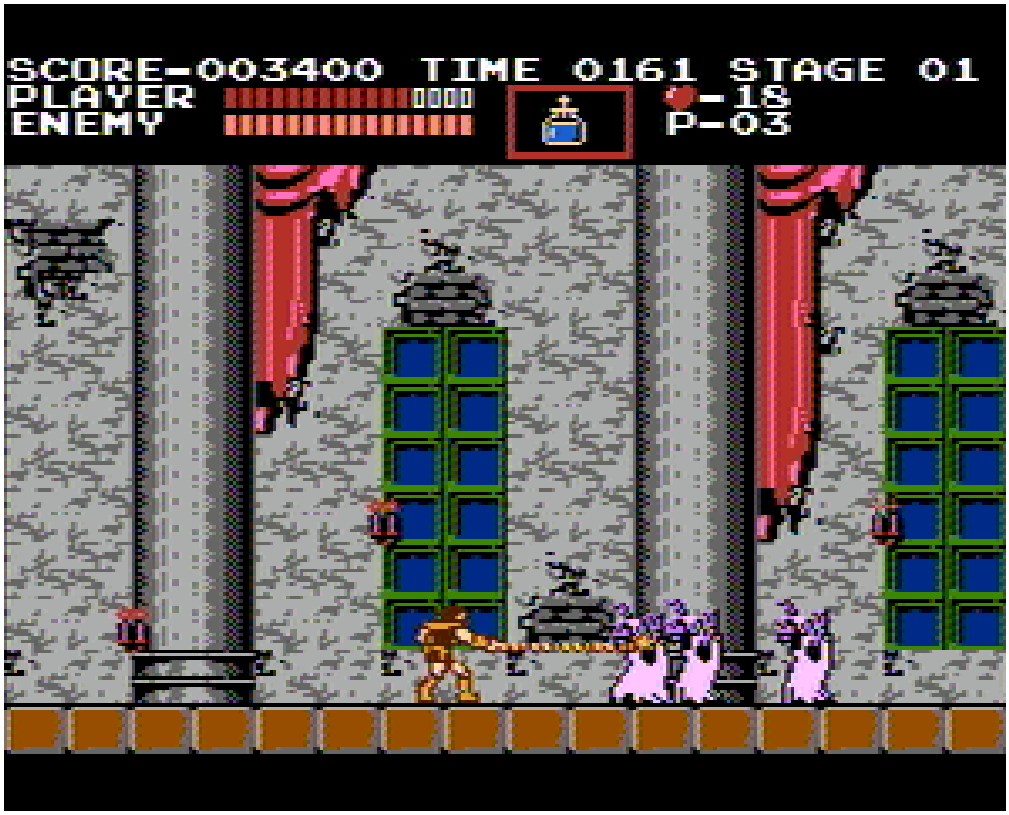

Рисунок 2.2 – Castlevania для платформи NES, 1986.

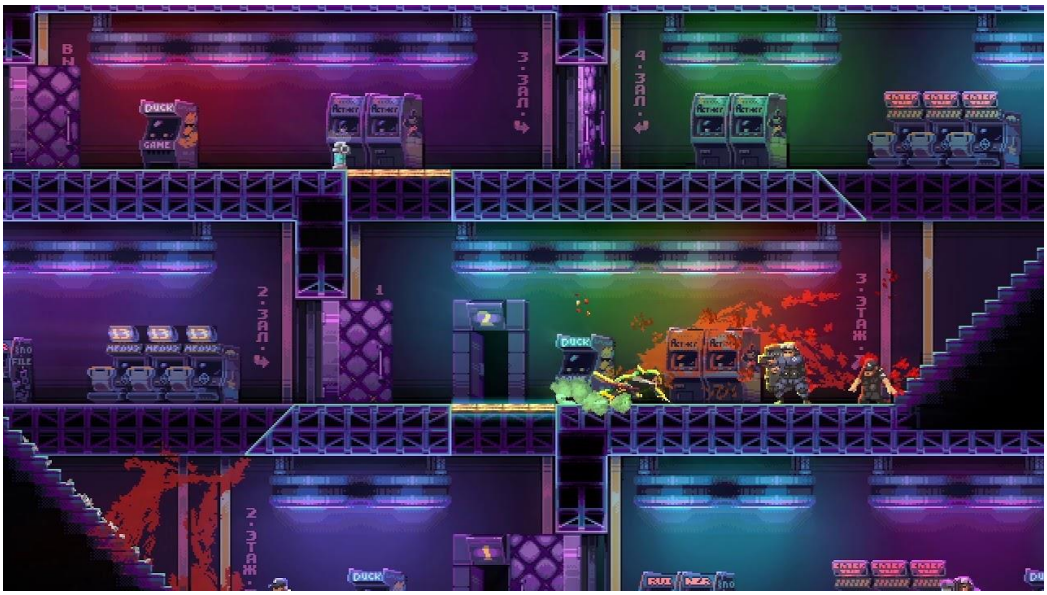

Рисунок 2.3 – кросплатформенна Katana ZERO, 2019

Тайлсети – це набори графічних елементів, які використовуються для створення більших ігрових середовищ. Кожен елемент тайлсету, званий тайлом, зазвичай являє собою квадратне або прямокутне зображення, яке повторюється або комбінується з іншими тайлами для створення різноманітних фонів, об'єктів та текстур [17]. Розмір блоку тайлсету для даного проекту обрано 16 пікселів на 16 пікселів, що дозволяє створювати чіткі та деталізовані зображення при збереженні економії пам'яті.

Спрайт – це графічний об'єкт, який використовується для представлення персонажів, ігрових елементів або інших візуальних компонентів у відеоіграх та інших цифрових проектах. Спрайти можуть бути анімованими, що дозволяє їм рухатися та змінюватися з часом. Так як базовий блок тайлсету складає 16x16 пікселів, відповідно спрайти об’єктів також дотримуються такої розмірної сітки.

2.2.2 Візуальна складова

Маючи напрацювання попередніх пунктів, є можливість почати роботу над візуальною частиною гри. Ця частина має три складових:

- персонажі;
- локації;
- елементи UI.

Як зазначено вище, обрано піксельне 2D оформлення. Для зручної роботи із піксельною графікою та анімацією використовується програма Aseprite. Вона надає функціонал для малювання, анімації та коректного експорту файлів з урахуванням піксельного розширення.

Важливою компонентою в розробці піксельних ігор є tileset – набір піксельних спрайтів (графічних об'єктів), які використовуються для створення графічних об'єктів та фонів у грі. Tileset визначає вигляд та структуру гри, особливо щодо фонів, мап, об'єктів та середовища. Вона має чітко розраховану сітку або координатну систему, що робить розташування об'єктів на мапі простим та систематизованим. За основну для сітки взято квадратний блок на 16 пікселів. Відштовхуючись від обраної сітки tileset, створено основні візуальні елементи:

- персонажі: гравець (рис. 2.4), Кіт, монстр зубоходець.
- локації: провулок, узвіз, підземний тунель, церква.
- елементи UI: діалогове вікно, головне меню, меню паузи, меню Game Over.

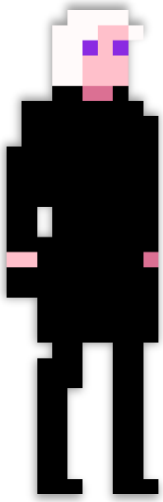

Рисунок 2.4 – Дизайн гравця

Також до персонажів було створено їх ігрові анімації. Таким чином гравець має анімації очікування, ходіння, використання пістолету, смерті; Кіт – дві анімації для розмови; зубоходець – анімацію очікування, руху, нападу та смерті. Гравець отримав невеликі додаткові анімації для катсцен.

2.3 Unity як платформа для розробки 2D ігор

2.3.1 Переваги та недоліки використання Unity

Unity – це міжплатформове середовище розробки ПО та комп`ютерних ігор [6], розроблене Unity Technologies. Він надає повний набір інструментів і функцій для розробки ігор, включаючи можливості 2D і 3D. Unity відомий своєю простотою використання, гнучкістю та підтримкою між платформами, що робить його популярним вибором серед розробників усіх рівнів кваліфікації.

Однією з ключових сильних сторін Unity є його інтуїтивно зрозумілий інтерфейс, який дозволяє розробникам створювати ігри за допомогою візуального редактора без потреби у великих знаннях програмування [6]. Візуальний редактор дозволяє розробникам з легкістю проектувати ігрові сцени, розміщувати об’єкти та створювати анімацію, роблячи процес розробки гри більш доступним для новачків.

Unity також пропонує надійний API сценаріїв, який дозволяє розробникам налаштовувати свої ігри за допомогою C# або UnityScript (мова, схожа на JavaScript). Цей API сценаріїв надає розробникам гнучкість для створення складних ігрових механізмів і функцій, а також інтеграції сторонніх плагінів і ресурсів для покращення своїх ігор.

Окрім простоти використання та гнучкості, Unity відомий своєю міжплатформною підтримкою. Розробники можуть розгортати ігри на різноманітних платформах, включаючи ПК, консолі, мобільні пристрої та Інтернет, без необхідності значного перероблення кодової бази [4]. Ця крос-платформна сумісність дозволяє охопити ширшу аудиторію, що робить Unity популярним вибором для розробників, які прагнуть максимізувати охоплення своїх ігор.

Хоча Unity пропонує безкоштовну версію, яка підходить для початківців та невеликих проектів, для розробки складніших ігор або програмного забезпечення необхідні платні версії. Їх ціна може бути досить високою, особливо для інді-розробників та невеликих команд [6].

Unity – це закрита платформа, що не надає доступу до вихідного коду програми. Така політика робить користувачів залежними від розвитку рушія та оновлень. Закритий базовий код може обмежувати розробників при створенні складних проектів. Також доводиться оновлювати Unity до останніх версій, адже кожна версія може містити помилки у певних аспектах рушія або не відповідати потребам проекту. Через це виникає ще одна незручність: оновлення Unity може призвести до проблем із сумісністю проектів, створених у різних версіях.

2.3.2 Основні інструменти та можливості Unity для 2D розробки

Хоча Unity у першу чергу напрямлений на 3D проекти, він надає повний набір функцій і інструментів, спеціально розроблених для розробки 2D-ігор. Ці функції задовольняють різні аспекти розробки 2D-ігор, включаючи графіку, фізику, анімацію та аудіо.

Система 2D-графіки Unity включає конвеєр 2D-рендерінгу, оптимізований для 2D-графіки, що дозволяє розробникам створювати візуально привабливі 2D-спрайти та тайлсети. Як було зазначено у пункті 2.2, невід'ємною складовою при розробці є спрайти. Unity надає інструменти для керування 2D-спрайтами, включаючи аркуші спрайтів і атласи спрайтів. Механізм також підтримує атласи текстур, які можуть допомогти оптимізувати використання пам'яті та підвищити продуктивність візуалізації.

Система анімації Unity підтримує 2D-анімацію. Unity пропонує два основних способи створення 2D анімації: покадрову та анімацію скелета. Unity надає інструменти для імпорту спрайтів до таймлайну анімації, налаштування їх порядку та швидкості відтворення, а також для додавання ефектів, таких як переходи та змішування. Для скелетної анімації усередині рушія вбудовано редактор, який дозволяє створювати та редагувати скелетну анімацію.

Механізм двовимірної фізики Unity забезпечує реалістичне фізичне моделювання об'єктів у двовимірному світі. За допомогою різних компонентів об'єктам у грі додаються властивості, такі як маса, тертя, та швидкість, які впливають на його рух у віртуальному світі.

3 РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

3.1 Створення та налаштування проекту

Для розробки 2D-гри в Unity було створено проект з використанням вбудованого шаблону 2D (Build-In Render Pipeline) (рис. 3.1). Програма автоматично створює кореневу папку проекту та основу ієрархії.

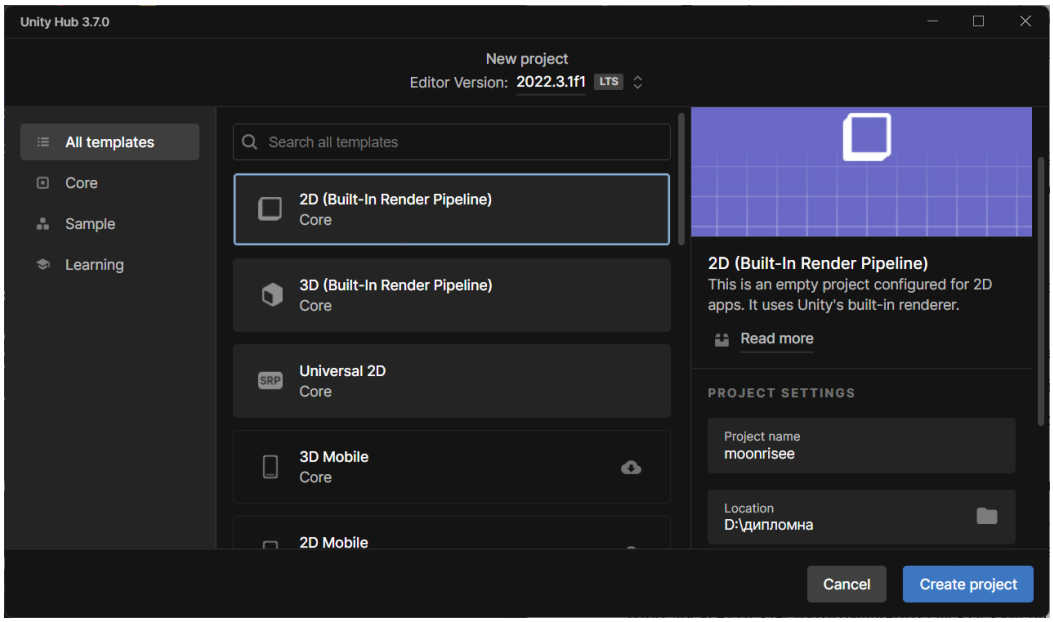

Рисунок 3.1 – Створення проекту Unity

Після створення проекту було зроблено основні налаштування. Всі вказані нижче розширення встановлено через вбудований магазин Unity – Asset Store.

Для розширення можливостей текстового оформлення в проекті було встановлено пакет Text Mesh Pro. Цей пакет надає гнучкість та контроль над текстовими елементами, що робить їх більш чіткими та естетично привабливими.

Для покращення візуальних характеристик та продуктивності гри використовується рендеринг-пайплайн URP (Universal Render Pipeline). Компонент URP Lights з цього пайплайну використовується для додавання освітлення до ігрових сцен.

Щоб досягнути кінематографічного ефекту камери та отримати більше зручних налаштувань для неї, у проект додано розширення Cinemachine Camera. Цей пакет дозволяє створювати динамічні та кінематографічні камери,

які автоматично слідують за об'єктами, реагують на події та плавно переходять між різними точками зору.

3.2 Створення сцен та візуальне наповнення

3.2.1 Додавання фону та об’єктів

Після додавання тайлсету (рис 3.2) до гри можна створювати фони сцен за допомогою інструментів Tile Palette. В ієрархії грі створюється новий об’єкт, що відповідає за шар сітки (рис. 3.3). Саме на цьому об’єкті можна малювати за допомогою пензлика та обраного блоку текстури. Для кращого візуального рішення та функціональності слід використовувати декілька об’єктів-шарів сіток.

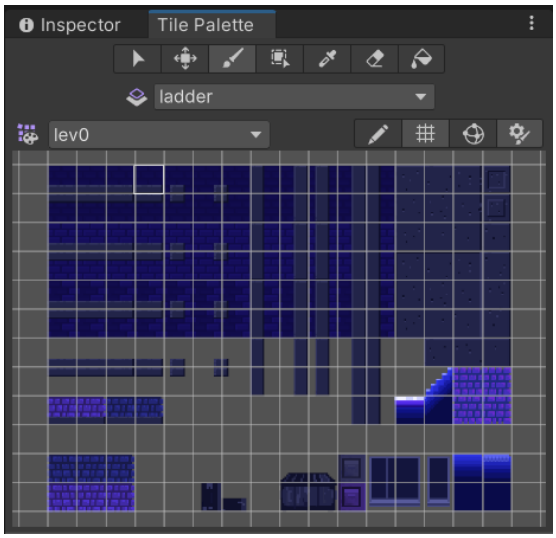

Рисунок 3.2 – Тайлсет для першої сцени гри

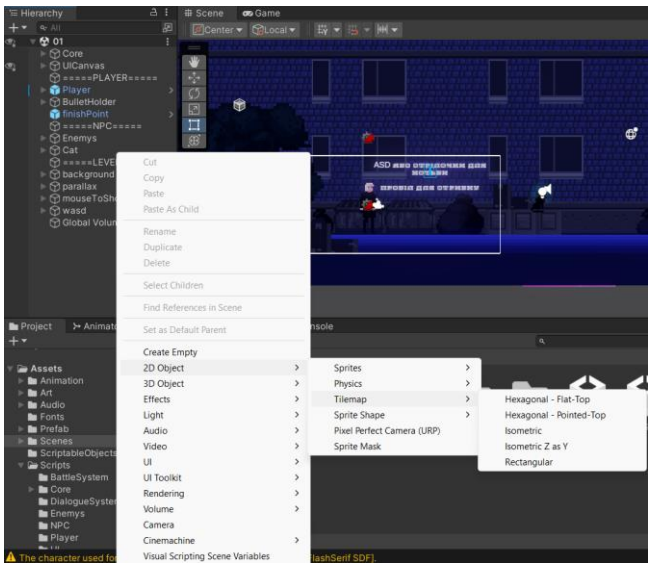

Рисунок 3.3 – Створення нового шару сітки.

Таким чином до всіх сцен створено візуальне наповнення фону. Шар із землею виокремлено окремо та додано колайдери Composite Collider 2D, Tilemap Collider 2D та Rigibody2D. А також створено окремий шар сортування Sorting Layer із назвою Ground, щоб надалі у скрипті руху персонажа встановити колізію із землею. Таким чином забезпечено положення гравця на поверхні землі та прості фізичні властивості.

Окремими ігровими об’єктами у ієрархії додано різні декоративні елементи. Вони розташовані на різній відстані від гравця, щоб створити ефект глибини плоскої сцени. Також деякі об’єкти виконують функції меж сцени, щоб гравець не виходив із неї. Наприклад, на першій сцені з лівого крайнього боку розташовано огорожу (рис. 3.4). До об’єкту додано Box Collider 2D, котрий не пропускає гравця крізь себе.

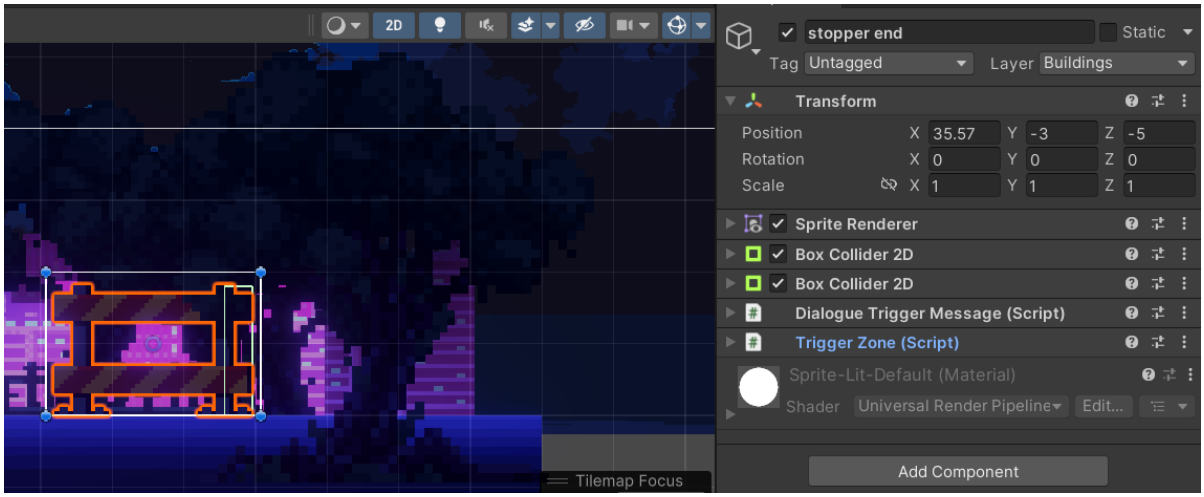

Рисунок 3.4 – Огорожа однієї з меж першої сцени

Для частин фону, які не підлягають тайлсетам, створено звичайний ігровий об’єкт, до якого додано Sprite Renderer. Завдяки налаштуванню Draw Mode → Tiled, зображення спрайту відзеркалюється повторно за вказаними значеннями довжини та висоти (рис. 3.5). Таким чином створено ефект «нескінченного фону». Такі елементи об’єднані під один батьківський об’єкт з подальшою метою на додавання ефекту паралаксу.

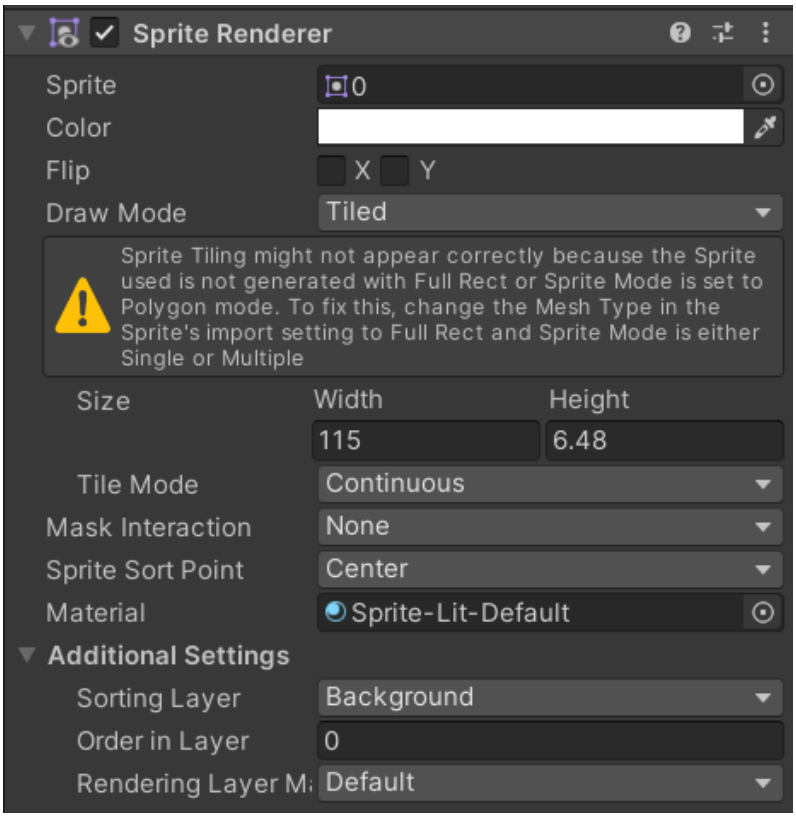

Рисунок 3.5 – Налаштування елементу фону для паралаксу

3.2.2 Спрайти гравця та NPC

Робота над спрайтами гравця та усіх NPC є такою ж, як і робота над окремими об’єктами фону, що не визначаються через тайлсет. У ієрархії сцени створюється новий ігровий об’єкт, до якого додається налаштування Sprite Renderer та у ньому обирається необхідне зображення.

Для використання піксельних зображень слід провести спочатку певні налаштування. Так як вихідне зображення завжди має дуже мале розширення, у середині гри спрайти виглядають розмитими. Основними пунктами налаштування є мод фільтрації (Filter Mode) та стиснення (Compression) (рис. 3.6). На рисунку 3.7 показано різницю між неналаштованим та налаштованим зображенням спрайту.

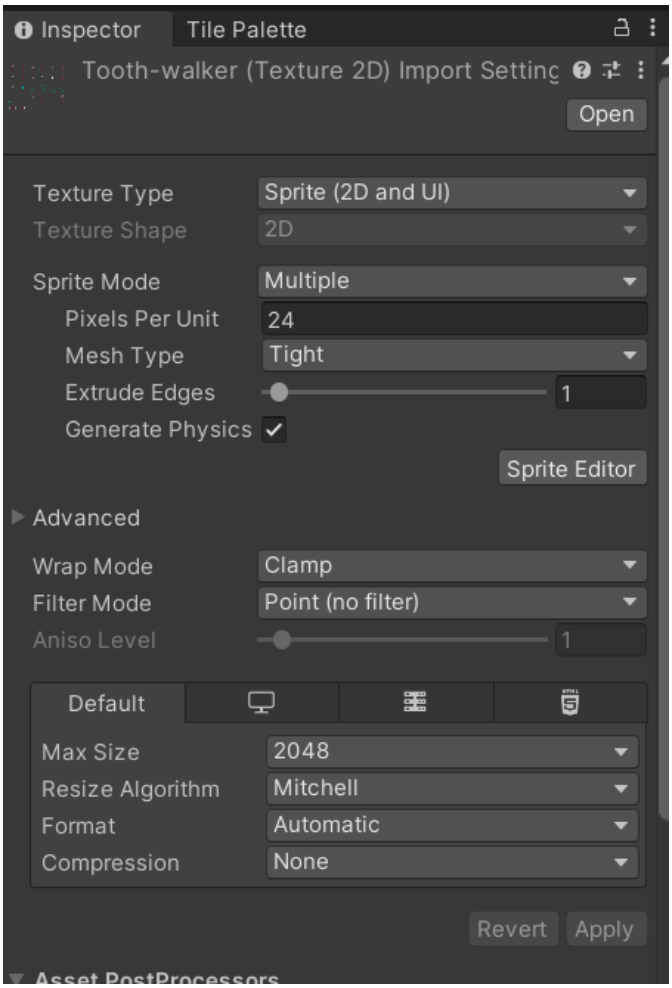

Рисунок 3.6 – Налаштування файлу спрайту

Рисунок 3.7 – Неналаштоване та налаштоване зображення

Деякі ігрові об’єкти, які повторюються від сцени до сцени, мають префаби (prefab). Система Prefab Unity дозволяє створювати, налаштовувати та зберігати ігровий об’єкт разом із усіма його компонентами, значеннями властивостей і дочірніми ігровими об’єктами як багаторазовий ресурс [3]. Це

дозволяє створити об’єкт один раз та використовувати його шаблон будь-де у грі скільки завгодно разів. Саме на ранньому етапі створення основних ігрових об’єктів, гравця та NPC-ворогів додано їх префаби. Надалі всі налаштування над цими об’єктами проводитимуться у вікні редагування префабу, щоб зміни додавалися до усіх екземплярів у грі.

3.2.3 Анімації

Піксельні анімації для проектів зберігаються у вигляді листів, що мають покадрові зображення (рис. 3.8). Кожна частина-кадр розташована у однакових межах одна від одної.

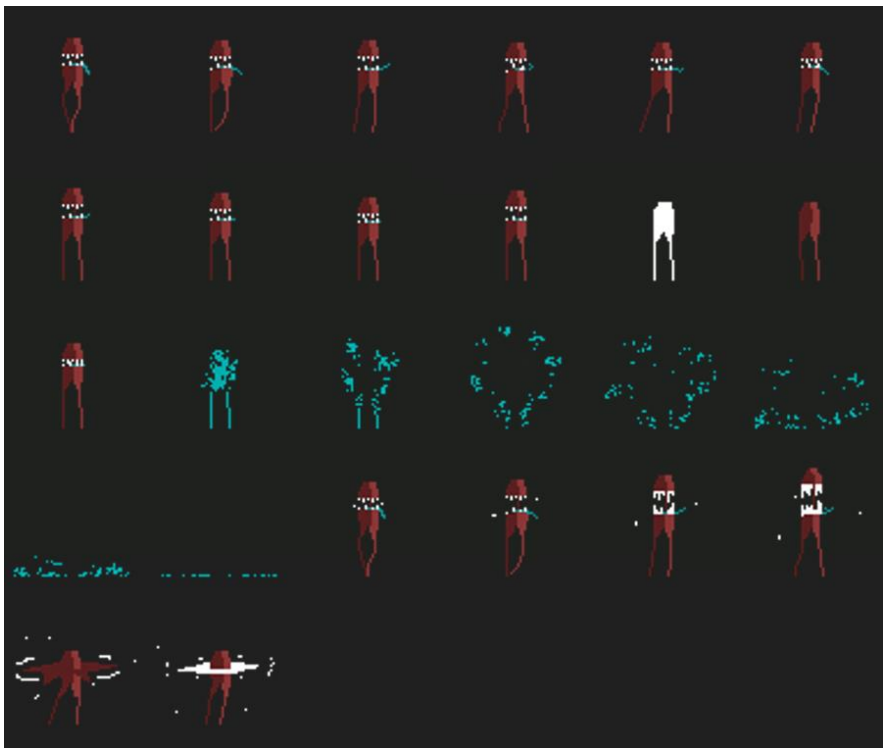

Рисунок 3.8 – Лист анімації монстра зубоходця

Для таких листів важливим налаштуванням у Unity, окрім вище вказаних, є Sprite Mode → Multiple. Завдяки ньому та його налаштуванням «розрізаються» на частини листи анімацій та автоматично конвертуються у окремі кадри всередині проекту.

Для збору та роботи анімації спершу необхідно створити Animation Controller. Через контроллер надалі налаштовується логіка програвання кожної анімації об’єкту та надається доступ до виконання та контролю цієї логіки через скрипти.

Маючи контроллер, у панелі Animation створюються анімації об’єкту. На таймлайн додаються зображення раніше обрізаних кадрів, виставляються налаштування типу кількості кадрів у секунду, додавання ігрових подій тощо. Усі маніпуляції із анімацією зберігаються в окрему папку Assets → Animation.

3.3 Програмне забезпечення

3.3.1 Рух гравця

Імплементація руху персонажа потребує розробки скрипта, що відповідає за динамічні параметри та керує його поведінкою, включаючи взаємодію з оточенням. Скрипт, який відповідає за це, називається PlayerMovement.cs.

Скрипт визначає необхідні змінні, які можна налаштувати через інспектор (рис 3.9). Змінні зберігають значення швидкості руху, сили стрибка та початковий напрямок, в якому дивиться персонаж; посилаються на компоненти Rigidbody2D, Animator та інші об'єкти, необхідні для роботи скрипта. Також скрипт містить аудіозапис для звуку стрибка.

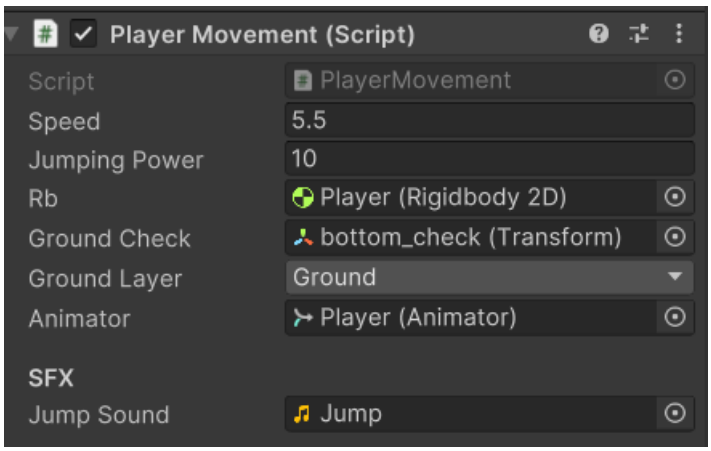

Рисунок 3.9 – Налаштування PlayerMovement.cs

Функція Update() виконується кожен кадр. Вона перевіряє, чи активний діалог або повідомлення за допомогою класів DialogueManager та DialogueManagerMessage. Якщо діалог активний, рух персонажа та анімація зупиняються. Інакше скрипт отримує горизонтальне введення від користувача (ліворуч чи праворуч) та встановлює значення для анімації швидкості персонажа. Скрипт також перевіряє, чи натиснута кнопка стрибка, і якщо так,

та персонаж знаходиться на землі, відтворюється звук стрибка та персонажа підкидає вгору. Ще одна перевірка відбувається, коли гравець відпускає кнопку стрибка. Якщо персонаж летить вгору, швидкість його підйому зменшується вдвічі, роблячи стрибки більш реалістичними. Наприкінці функції Update() викликається функція Flip() для перевертання персонажа відповідно до траєкторії руху.

Скрипт також містить функцію canAttack(), яка перевіряє, чи персонаж може атакувати. Атака можлива лише тоді, коли персонаж не рухається горизонтально (horizontal == 0) та знаходиться на землі (IsGrounded()). На додачу, скрипт реагує на зіткнення з об'єктами тегами "Hole" та "Ladder" у функціях OnCollisionEnter2D та OnCollisionExit2D, які є елементами переходу між сценами.

3.3.2 Динамічна камера

Система динамічної камери в проекті складається з трьох основних компонентів: наслідування гравця, паралакс та зміна зуму. Кожен компонент використовує власні механізми та скрипти для створення візуально динамічної та захоплюючої камери. Для досягнення потрібного ефекту в грі використовується система Cinemachine Virtual Camera.

Наслідування гравця є одним із стандартних параметрів Cinemachine Camera. Достатньо просто перетягнути об’єкт, за яким необхідно слідкувати, у поле «Follow» (рис. 3.10), і камера буде зосереджена тільки на ньому.

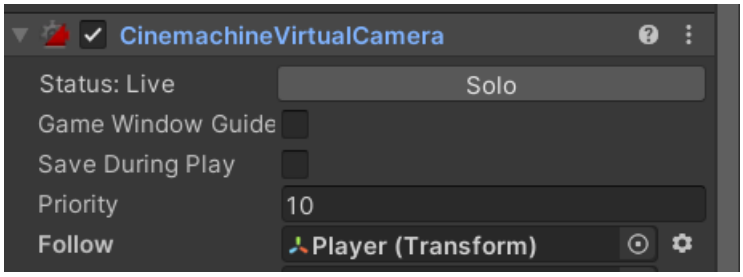

Рис 3.10 – Наслідування об’єкту в Cinemachine Camera

Для створення ефекту **паралаксу** використовується система ParallaxCamera. Ця система відстежує рух камери та переміщує фонові шари з різними швидкостями, створюючи ілюзію глибини. Швидкість переміщення

кожного шару визначається його "коефіцієнтом паралаксу". Чим вище значення, тим швидше шар буде рухатися, створюючи відчуття, що він знаходиться ближче до гравця.

До системи паралаксу відноситься три скрипти. Скрипт ParallaxCamera.cs застосовується безпосередньо до об’єкту Main Camera. Скрипт постійно відстежує позицію основної камери в грі. Коли камера переміщується, скрипт викликає подію onCameraTranslate. Ця подія передає значення дельти (різниця між поточною та попередньою позиціями) всім зареєстрованим об'єктам. Реєструються об’єкти за допомогою скрипту ParallaxBackground.cs, котрий додається до батьківського об’єкту усіх шарів, що підлягають паралаксу.

Скрипт ParallaxLayer.cs додається до кожного дочірнього об’єкту. У інспекторі можна налаштувати значення коефіцієнту паралаксу, який відповідає за швидкість переміщення шару (рис. 3.11). Скрипт використовує метод Move() для переміщення фонового шару в горизонтальному напрямку. Швидкість переміщення визначається коефіцієнтом паралаксу шару та дельтою, отриманою від скрипта ParallaxCamera.

Рисунок 3.11 – Налаштування ParallaxLayer.cs

Зміна **зуму** камери використовується для підкреслення важливих моментів у грі. Для цього також використовується система Cinemachine Virtual Camera та додатково об’єкт, що має у собі тригер-колайдер та визначає межі зміни зуму. Коли гравець входить в зону дії тригер-колайдера, пріоритет «віддаленої» камери збільшується, що призводить до візуального віддалення від гравця; коли гравець покидає зону колайдера, пріоритет змінюється на користь «приближеної» камери, через що візуально приближуємося до гравця.

Скрипт CameraManager.cs є центральним елементом системи керування кількома камерами Cinemachine. Він відповідає зберігання, відстеження та перемикання активних камер Cinemachine Virtual Camera.

Для реєстрації чи зняття реєстрації камери використовується CameraRegister.cs. При активації об'єкта, до якого приєднаний скрипт, автоматично реєструється в CameraManager.cs.

Щоб контролювати коректну зміну пріоритетів камер, до об’єкту із тригер-колайдером, що визначає межі «віддаленої» камери, додано універсальний скрипт TriggerZone.cs. Він відповідає за виявлення зіткнень об’єкту з певною областю та виконання відповідних дій на основі цих зіткнень. У випадку системи динамічного зуму, необхідно перевіряти зіткнення об’єкту гравця із тегом «Player» із тригер-колайдером. На вході у тригер-колайдер налаштовано зміну пріоритету на віддалену, при виході – на приближену. Налаштування цього скрипту показані на рисунку 3.12.

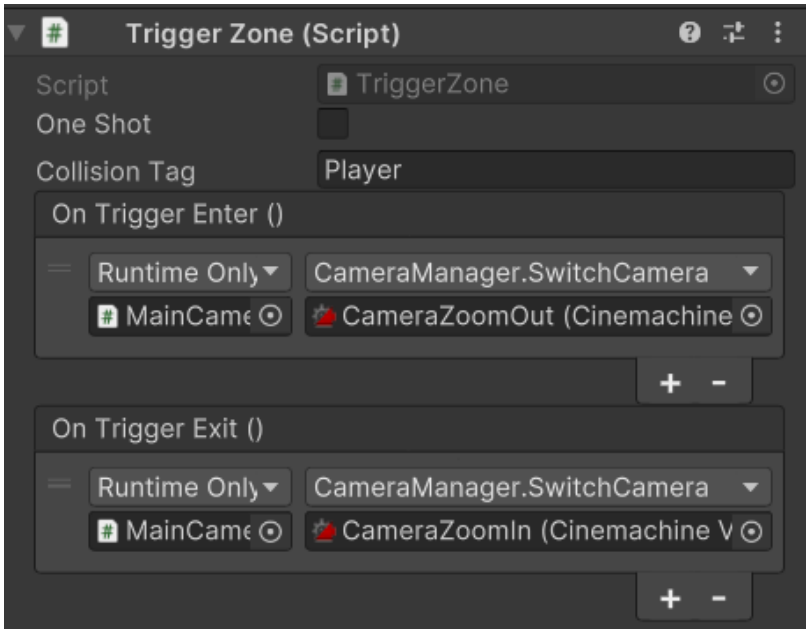

Рисунок 3.12 – Налаштування TrigerZone.cs для системи зміни зуму камер

3.3.3 Система здоров’я

У грі реалізована система здоров'я, яка враховує пошкодження, смерть та респавн (перероджження) гравця, а також монстрів. Система має наступні основні характеристики:

- змінне значення здоров'я: для кожного об'єкта (гравця та монстрів) встановлюється стартове значення здоров'я, яке можна змінювати в інспекторі Unity відповідно до потреб;
- пошкодження: об'єкти можуть отримувати пошкодження, що призводить до зменшення їхнього поточного значення здоров'я;
- смерть: коли поточне значення здоров'я об'єкта досягає 0, він помирає;
- респавн: гравець має можливість відроджуватися після смерті, починаючи з початку сцени з повним запасом здоров'я.
- збереження поточного рівня здоров'я: значення здоров'я гравця зберігається між сценами та коректно відображається в UI.
- візуалізація: система здоров'я візуалізується за допомогою UI та відповідних анімацій об’єктів.

Таким чином система здоров’я базується на скрипті Health.cs, який відповідає за більшу частину вище перечислених характеристик. Скрипт додається до будь-якого об’єкта в ігровій ієрархії, якому необхідно надати можливість долучитися до системи здоров’я. Завдяки скрипту Health.cs налаштовуються індивідуальні для об’єкта значення стартового (максимального) рівня здоров’я, налаштування для анімацій отримання ушкодження та смерті, налаштування супроводжуючих звуків (рис. 3.13).

Для гравця у скрипті є декілька додаткових налаштувань (рис. 3.14). Усі вони мають додаткову перевірку на унікальний тег «Player», який має лише ігровий об’єкт гравця. Така перевірка забезпечує виконання коду лише якщо вони стосуються гравця. Наприклад, коли гравець досягає рівня здоров’я нуль та його ігровий об’єкт вимикається, у методі, що відповідає за смерть додатково вмикається екран Game Over.

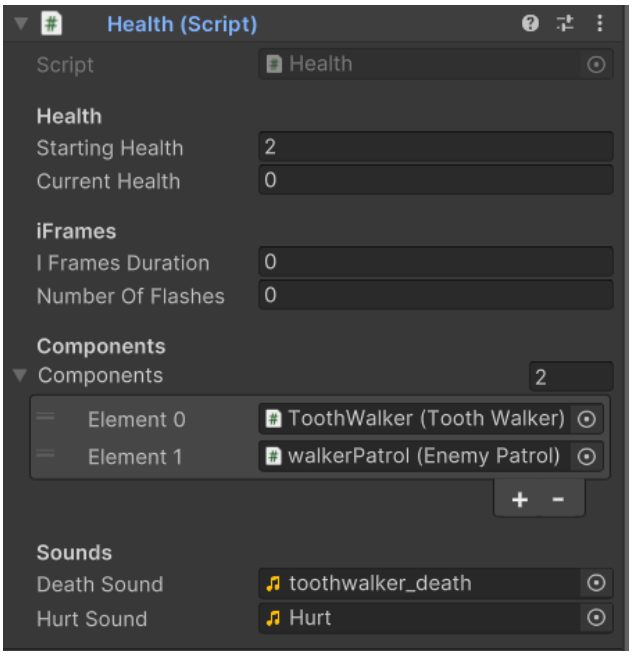

Рисунок 3.13 – Налаштування скрипту Health.cs для монстрів

Рисунок 3.14 – Налаштування скрипту Health.cs для гравця

Дані поточного здоров’я не передаються скриптом Health.cs між сценами, адже скрипт передбачений як універсальний для будь-якого типу об’єктів у грі. Через це, щоб тримати значення поточного здоров’я гравця протягом усіх змін сцен, існує невеликий скрипт HealthManager.cs. Він містить у собі DontDestroyOnLoad(gameObject) – метод Unity, який використовується для запобігання знищенню об'єкта під час завантаження нової сцени. Таким чином, при кожній зміні поточного здоров’я об’єкту із тегом «Player» зміни із

Health.cs передаються до змінної playerHealth, де вони зберігаються. При завантаженні нової сцени ці дані підтягуються до Health.cs для об’єкту із тегом «Player». За замовчуванням значення змінної playerHealth встановлюється у скрипті як максимальне, що забезпечує коректне виведення інформації на самому початку гри.

Для візуалізації рівня здоров’я гравця у грі присутній об’єкт HealthBar. Він є складовою UI ієрархії. HealthBar відображений як image-об’єкт, ряд із 10 однакових сердечок. Завдяки налаштуванню Image Type → Filled та Fill Amount можна змінювати кількість відображуваних сердечок.

Щоб візуалізація працювала коректно, до об’єкту додано відповідний скрипт HealthBar. Через інспектор до нього додано дані про здоров’я об’єкту, чий рівень потрібно відобразити, а також image-об’єкти HealthBar (рис.3.15). Скрипт має всього два невеликі методи, які змінюють налаштування Fill Amount відповідно до рівня поточного здоров’я.

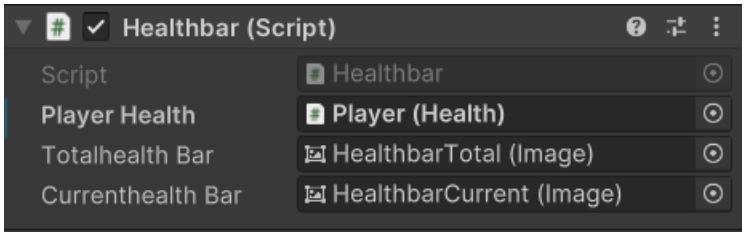

Рисунок 3.15 – Налаштування скрипту HealthBar.cs

3.3.4 Система бою гравця

У грі гравець має змогу атакувати противників за допомогою стрільби. Для цього до гравця додано скрипт PlayerAttack.cs. Через інспектор налаштовано основні дані скрипта: встановлено час відновлення атаки, аудіо ресурс для звука вистрілу, точка об’єкту гравця, з якої має йти вистріл та масив ігрових об’єктів типу пуль (рис. 3.16).

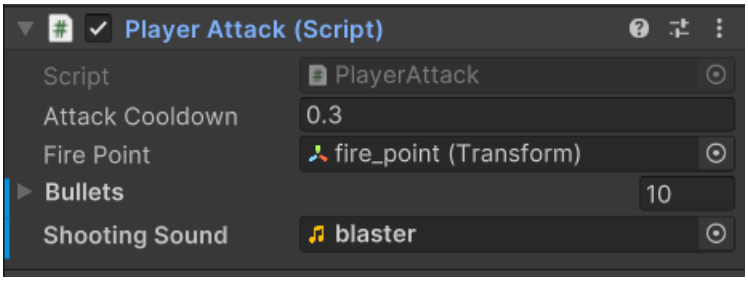

Рисунок 3.16 – Налаштування PlayerAttack.cs

Гравець виконує атаку лише тоді, коли він не рухається та не знаходиться у стрибку. Умова, через яку виконується атака – було натиснено ліву кнопку миші та час відновлення досяг нуля.

Існує два основних методи для створення пуль при вистрілі: створити й знищити (instantiate & destroy) та об'єднання об'єктів (object pooling). Різниця між методами представлена у таблиці 3.1.

Таблиця 3.1 – Порівняння методів створення об'єктів при атаці

Instantiate & destroy	**Object pooling**
- Створюється новий об'єкт (пуля) при кожному пострілі; - об'єкт знищується, коли вдаряє ціль; - легше реалізувати, але дуже впливає на продуктивність.	- Декілька об'єктів (пуль) створені завчасно; - об'єкт деактивується, коли вдаряє в ціль і чекає поки його використають знову; - рекомендується використовувати, коли створюється велика кількість об'єктів.

Для зручності у проекті було використано метод об'єднання об'єктів. Для цього створено шаблон пулі, яка містить у собі скрипт Projectile.cs. Він відповідає за поведінку та життєвий цикл пулі: швидкість її переміщення (можна налаштувати в інспекторі), напрямок переміщення, час життя, анімацію та чи вдарила пуля необхідний об'єкт (рис. 3.17).

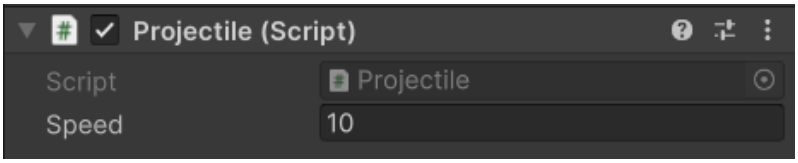

Рисунок 3.17 – Налаштування Projectile.cs

Шаблон об'єкту пулі продубльовано дев'ять разів, а отже в ігровій ієрархії знаходиться 10 пуль. Вони додані до масиву пуль у скрипті PlayerAttack.cs, через що гравець отримує до них доступ. При атаці гравцем

використовується найближча вільна у масиві пуля. Якщо пуля не вдаряє по цілі та покидає сцену, вона деактивовується автоматично через указаний час життя.

3.3.5 Система бою ворогів

Вороги, як було зазначено на етапі проектування, представляють із себе монстрів із великими зубами. При розробці механік гри було обрано наступні характеристики ворожих NPC:

- монстри мають одну стандартну атаку, яка забирає одне сердечко у гравця за раз;
- коли монстри не атакують, вони ходять за певним маршрутом по сцені;
- якщо гравець проходить крізь монстра, гравець отримує урон у вигляді одного сердечка, навіть якщо монстр не атакував;

Попередньо створено ігровий об’єкт, який містить у собі налаштовані анімації монстра. Також до об’єкту додано фізичні межі у вигляді Box Collider 2D із прапорцем «Is Trigger» (рис. 3.18), для того, щоб відслідковувати знаходження гравця у полі зору монстра.

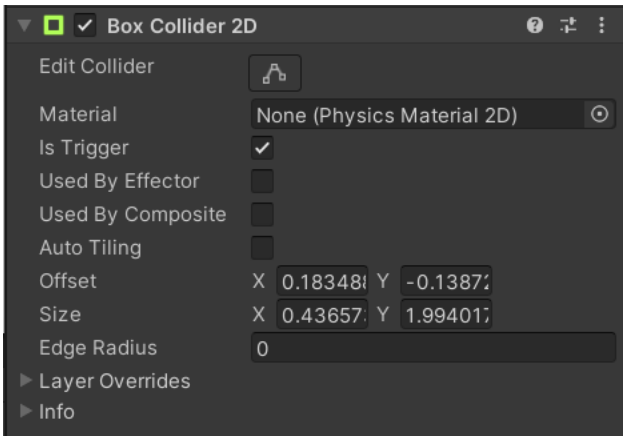

Рис 3.18 – Вікно компонента колізії монстра

Для логіки атаки монстра використано скрипт ToothWalker.cs. Скрипт має методи для нанесення атаки на гравця, відображення анімації атаки, перевірку на наявність гравця у полі зору, а також невеликий метод для відображення меж поля зору, щоб було зручніше орієнтуватися при налаштуванні. Через інспектор налаштовано основні дані, які необхідні для роботи скрипта: час перезарядки атаки, кількість пошкодження, що наносить

монстр, налаштування для атаки у полі зору та вказано шар із гравцем і аудіо кліп, із яким наноситься атака (рис 3.19).

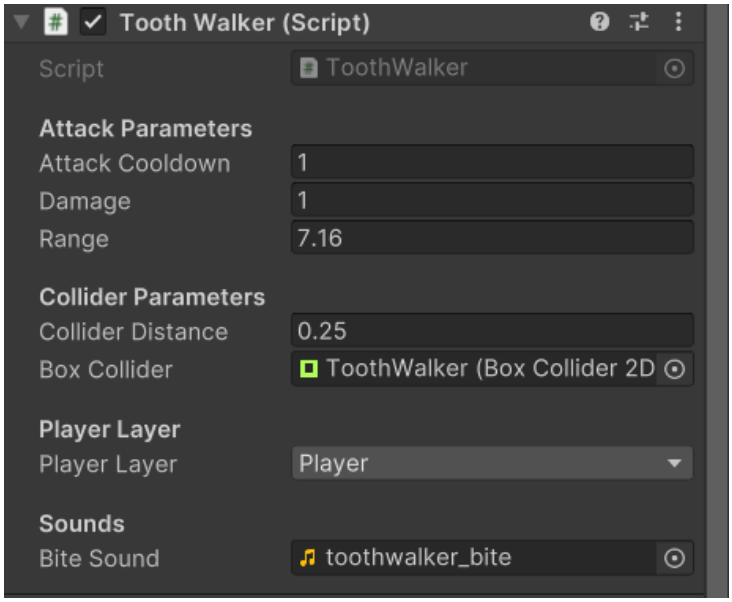

Рисунок 3.19 – Налаштування скрипту ToothWalker.cs

Для реалізації нанесення пошкоджень гравцю, коли той проходить крізь монстра, використано скрипт EnemyDamage.cs. Скрипт містить всього один метод, що перевіряє колізію поточного ігрового об’єкта із об’єктом, що має тег «Player». Якщо колізія є – виконується метод TakeDamage() із скрипту Health.cs, що належить об’єкту із тегом. У інспекторі налаштовується кількість шкоди, що віднімається під час виконання скрипту EnemyDamage.cs (рис. 3.20).

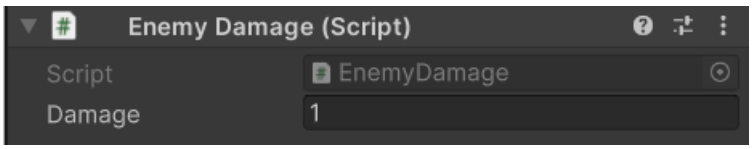

Рисунок 3.20 – Налаштування EnemyDamage.cs

Щоб NPC-вороги не стояли на місці, створено логіку патруля. Завдяки ній монстри періодично ходять з точки до точки, трохи затримуються на них, та йдуть назад.

Ігрові об’єкти лівої та правої точок виставлені на бажаній відстані та не мають у собі ніяких додаткових компонентів. У ієрархії точки знаходяться на одному рівні із об’єктом монстра. Усі три елементи угруповані під один

батьківський елемент WalkerPatrol. Саме батьківський елемент містить скрипт EnemyPatrol.cs, що відповідає за логіку патрулю.

Скрипт має наступні кастомізовані методи:

- Update(): перевіряє, на координатах якої точки (лівої чи правої) знаходиться наразі монстр та вказує, у якому напрямку рухатися йому надалі;
- DirectionChange(): відповідає за період очікування монстра на точках;
- MoveInDirection(): онулює таймер очікування на точці та змінює позицію монстра до протилежної точки.

Усі основні змінні винесені у скрипті у Serialize Field, завдяки чому їх можна налаштувати через інспектор, не зважаючи на privat налаштування (рис 3.21).

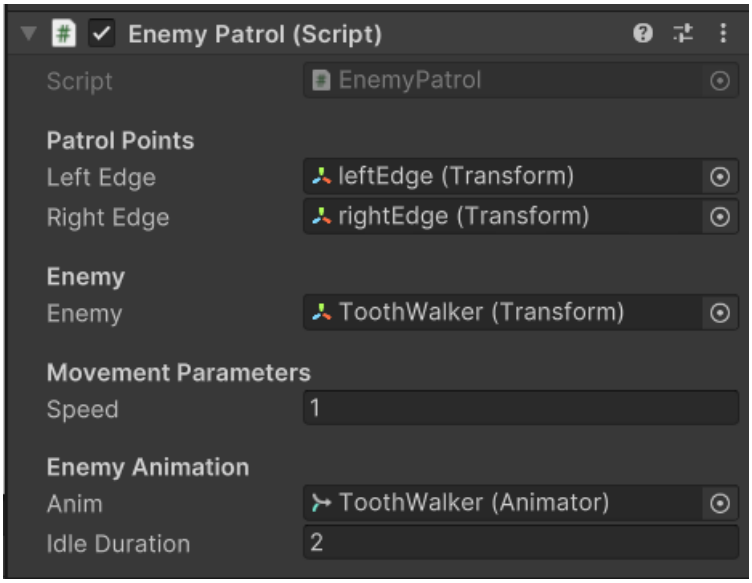

Рисунок 3.21 – Налаштування скрипту EnemyPatrol.cs

3.4 UI та основні меню гри

3.4.1 Система діалогів та повідомлень

Для створення будь-яких UI елементів на сцені необхідно мати в ієрархії грі Canvas об’єкт. Він відповідає за коректну розміртку елементів на кінцевому екрані гри, враховуючи будь-яке розширення екрану. Усі наступні елементи UI мають розміщуватися як дочірні Canvas.

Основна інформація у грі подається за допомогою діалогів та повідомлень. Це дві окремі системи із абсолютно ідентичними основними налаштуваннями. Їх головні різниці полягають у розміщенні діалогового вікна в розмітці ігрового вікна та аудіокліпах, які відповідають за звук напису тексту.

Діалогова система призначена для відображення спілкування між гравцем та/або ігровими персонажами. Система повідомлень відповідає за вивід допоміжної інформації та думок гравця.

Розглянемо приклад цих систем на системі діалогів. Створено UI шаблон діалогового вікна (рис. 3.22), яке збережено як префаб. Вікно містить фон, зображення для розміщення аватару того, хто говорить та текстове вікно.

Рисунок 3.22 – UI діалогового вікна

До шаблону додано скрипт DialogueManager.cs, що має наступні методи:

- Awake(): при завантажені сцени вимикає усі активні діалогові вікна;
- Update(): коли діалогове вікно активне та натискається клавіша Enter, запускається метод відображення наступного рядка;
- StartDialogue(): запускає переданий у метод діалог, вмикає вікно діалогу та відображує перший рядок;
- DisplayNextDialogueLine(): відповідає за відображення наступного рядка діалогу в черзі. Він перевіряє, чи немає більше рядків; якщо черга порожня, метод викликає EndDialogue(), щоб завершити діалог; в іншому випадку він отримує поточний рядок із початку черги та викликає метод для посимвольного виведення рядка;
- TypeSentence(): виводить переданий рядок по символах із програванням звуку виведення;
- EndDialogue(): завершує діалог та вимикає вікно діалогу.

Для того, щоб додати діалог, необхідно до об’єкту, з яким має бути діалог, додати скрипт DailogueTrigger.cs. Цей скрипт містить у собі модель даних для діалогу, таких як аватар персонажу, масив рядків діалогу, а також безпосередньо метод, що викликає початок діалогу та передає масив рядків. Усі налаштування додаються через інспектор (рис 3.23).

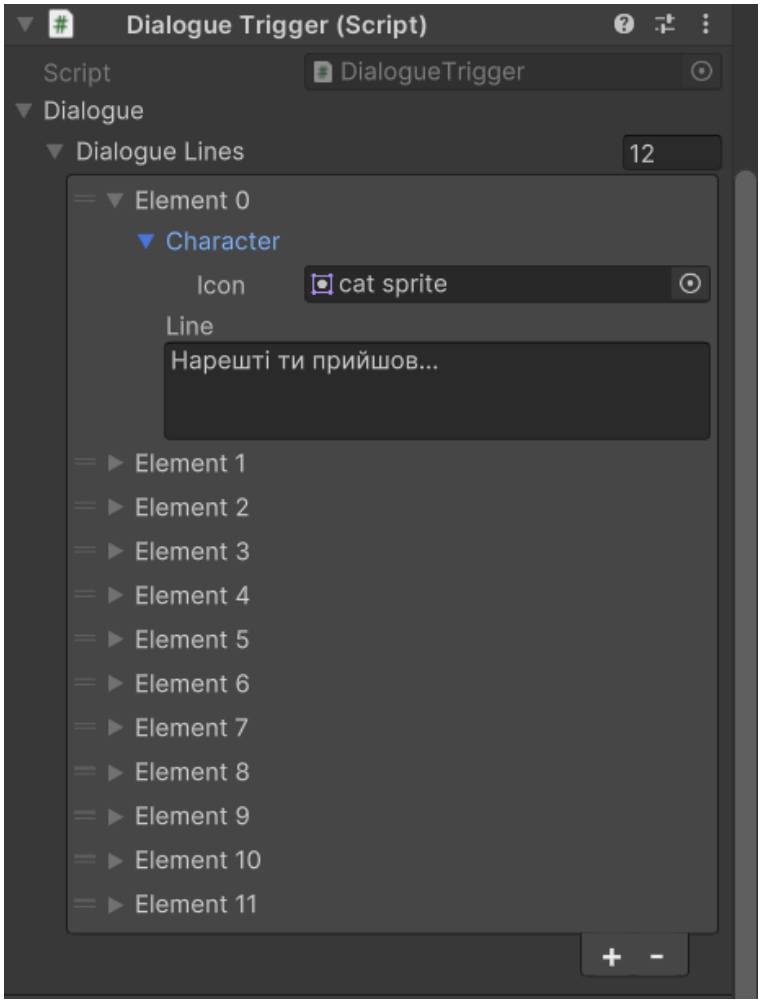

Рисунок 3.23 – Налаштування DialogueTrigger.cs

Для запуску діалогу використовується скрипт TriggerZone.cs, що був описаний у пункті про динамічну камеру. Скрипт додається до того ж об’єкту, що і DialogueTrigger.cs, та налаштовується відповідно до потреб. Приклад налаштування скрипту зображено на рисунку 3.24.

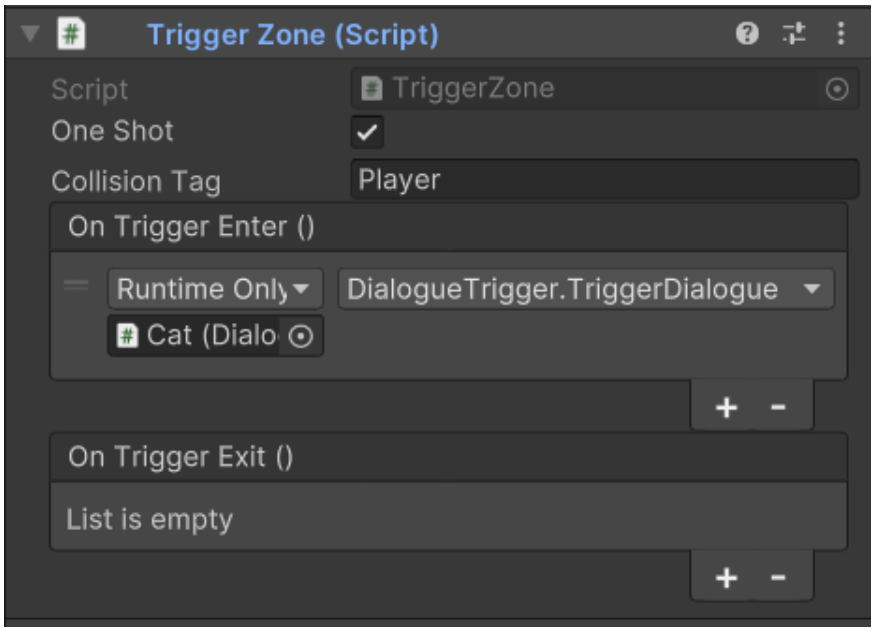

Рисунок 3.24 – Налаштування TriggerZone.cs для використання діалогу

На рисунку зображено налаштування діалогу на початку гри з Котом. Діалог починається автоматично, коли гравець підходить до Кота. Програти діалог можна лише раз – на це вказує прапорець One Shot.

3.4.2 Головне меню

Головне меню зроблене як окрема сцена. Вона виставлена у збірці під індексом 0, тобто є першою при запуску.

Меню містить назву гри, кнопки старту, налаштування рівня звукових ефектів та музики, виходу із гри. Назва створена як ігровий об’єкт із Text Mesh Pro. Кнопки, окрім Text Mesh Pro, також мають Button налаштування (рис. 3.25). На фон додано декілька зображень.

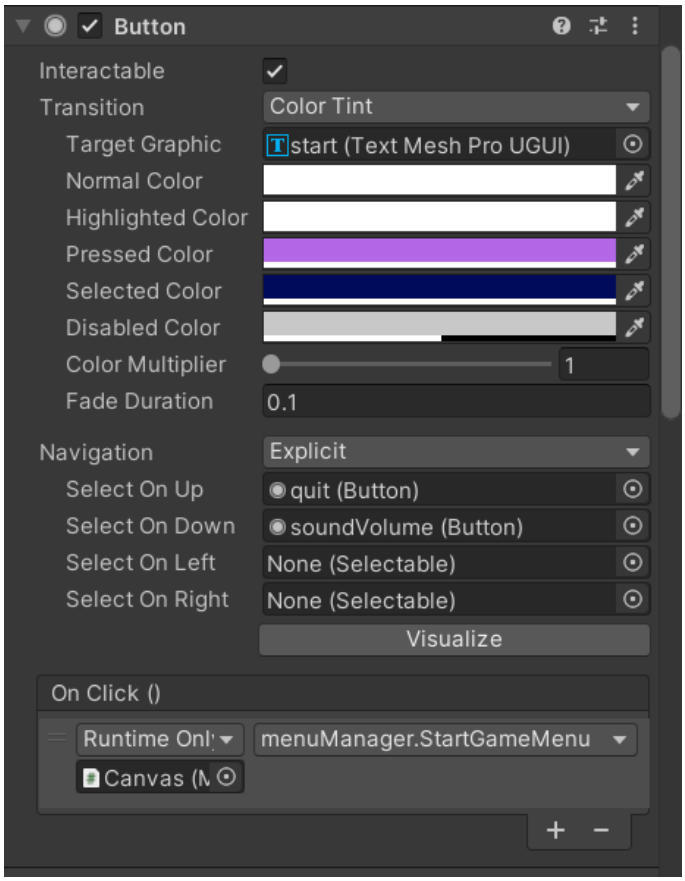

Рисунок 3.25 – Налаштування кнопки Start

На Canvas об’єкт додано скрипт MenuManager.cs. Він містить методи, що визначають роботу кожної кнопки.

3.4.3 Меню паузи та закінчення гри

Меню паузи та закінчення гри додаються як префаби до кожної сцени гри. Вони досить схожі між собою, але мають дещо різні кнопки та умови появи.

Меню закінчення гри з’являється, коли гравець помирає. Інакше кажучи, коли здоров’я гравця досягає нуля, активується відображення вікна закінчення гри.

Меню містить напівпрозорий фон, заголовок-назву, кнопки та стрілку, яка вказує, яка саме зараз обрана опція. Для координації роботи як меню закінчення, та і меню паузи, на їх батьківський об’єкт Canvas додано скрипт UIManager.cs. Він вимикає активні меню при запуску сцени, активує або деактивує меню паузи, якщо натиснута кнопка Escape та має методи для

кожної кнопки в обидвох меню. Через інспектор налаштовано основні змінні UIManager'а (рис. 3.26).

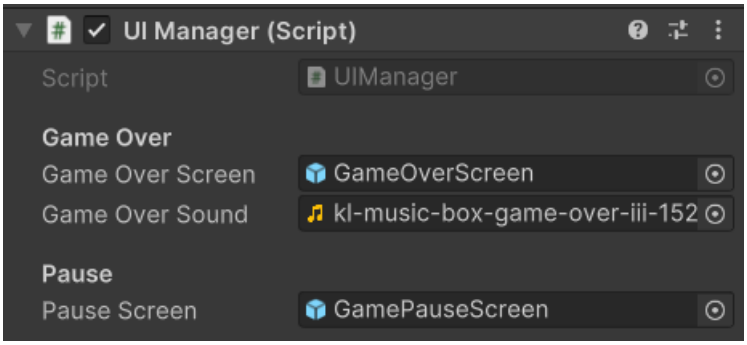

Рис.3.26 – Налаштування UIManager.cs

Стрілка, що відповідає за вказування поточної опції, має скрипт SelectionArrow.cs. Він створює систему вибору, де гравець може використовувати клавіші клавіатури (W, S, або стрілки вгору/вниз) для переміщення стрілок вибору між кнопками. Коли користувач натискає Enter або E, запускається функція, пов'язана з поточним вибраним елементом, імітуючи натискання кнопки. Налаштування в інспекторі для цього скрипта наведено на рисунку 3.27.

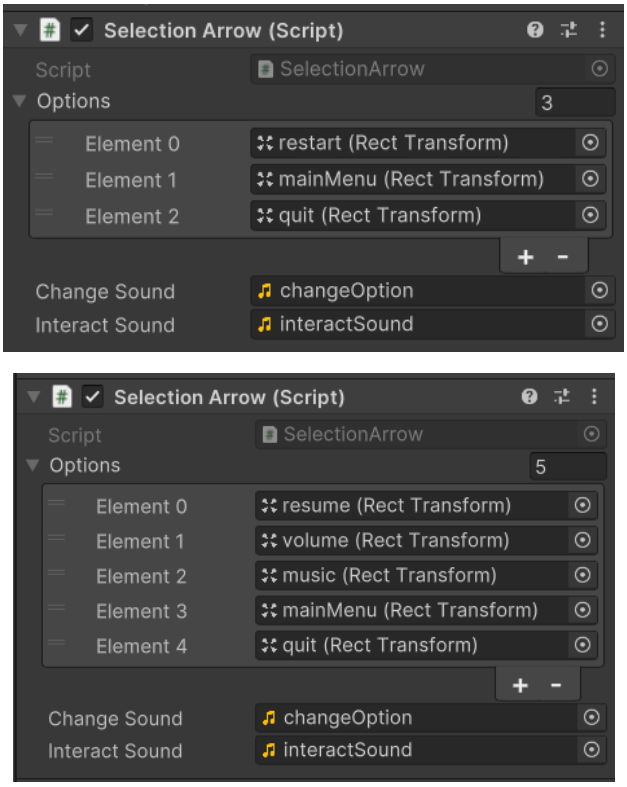

Рисунок 3.27 – Налаштування стрілки вибору для меню закінчення гри та меню паузи

Усі кнопки у обох меню мають налаштування Button. У методі OnClick() викликається скрипт UIManager.cs та відповідний до кнопки метод із цього скрипта. На рисунку 3.28 зображено налаштування Button для кнопки «продовжити» у меню паузи.

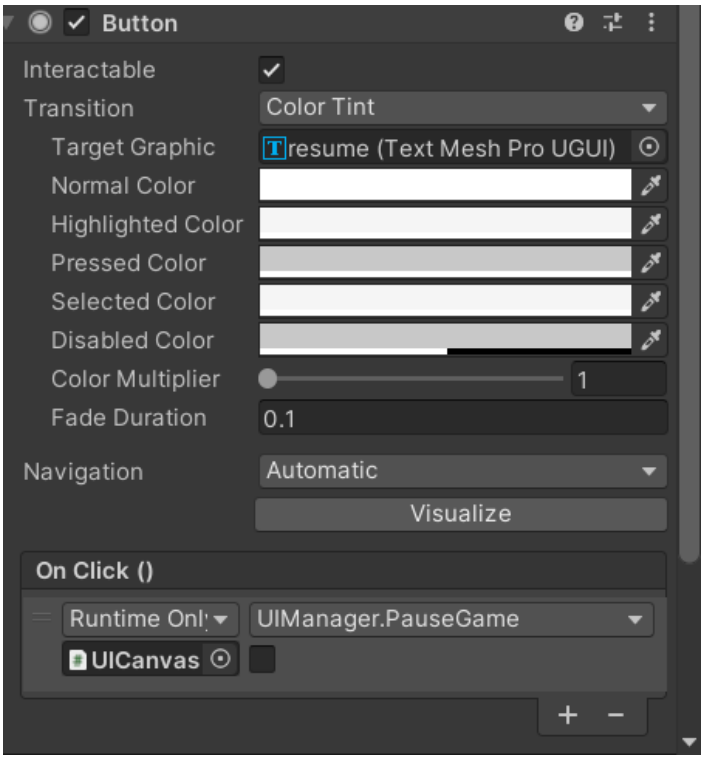

Рис. 3.28 – Кнопка «Продовжити» у меню паузи

ВИСНОВКИ

У роботі було проведено аналіз жанру RPG, порівняльний аналіз знакових ігор у цьому жанрі та висвітлення перспектив розвитку ігрової індустрії як у світі, так і в Україні. Було сформоване чітке технічне завдання та функціональні вимоги до нього.

Під час етапу проектування, на основі попереднього аналізу, проведено роботу над концепцією гри. Вона включала в себе розробку сюжету, основних ігрових механік, створення UML діаграми варіантів використання ігрового додатку, а також дизайн візуальної складової, що відповідає розробці ергономічного користувацького інтерфейсу, спрайтів та тайлсетів, що підходять тематиці.

Функціональний прототип ігрового додатку реалізовано за допомогою мови програмування C# та ігрового рушія Unity. При розробці візуальної складової використовувалися програми Aseprite та Clip Studio Paint. Було скомпоновано чотири сцени: головне меню та три ігрові сцени. Створено меню паузи та закінчення гри.

Як результат, було досягнуто поставлену мету – розроблено візуальне та програмне забезпечення ігрової системи жанру двовимірної рольової гри «Moonrise» для операційної системи Windows у стилістиці піксельної графіки. Гра відповідає всім поставленим вимогам, включаючи присутність гравця, ворогів, NPC, зброї, простоту управління, якісний звуковий супровід та відповідний дизайн. Гру протестовано та перевірено на працездатність.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Беловолченко А. Як працювати з ігровими механіками й чому не всі з них вдалі: поради геймдизайнерів. GamedevDou. URL: https://gamedev.dou.ua/articles/how-to-work-with-game-mechanics/
2. Імерсивні технології в освіті : ЗБ. МАТЕРІАЛІВ, м. Київ, 28 жовт. 2022 р. Київ, 2022. С. 63–66.
3. Комп'ютерні ігри та мультимедіа як інноваційний підхід до комунікаційї / Матеріали 1 Всеукраїнської науково-технічної конференції молодих вчених, аспірантів та студентів. Одеса, 25-26 березня 2021 р. - Одеса, Видавництво ОНАХТ, 2021 р. - 98 с.
4. Попов А., Дробот Д. СПРОЩЕННЯ РОЗРОБКИ КРОСПЛАТФОРМНИХ ІГОР НА UNITY 3D. Проблеми інформатизації : Тези доп., м. Харків. Харків, 2020. С. 83.
5. Ракович В. Аналіз засобів розробки ігор для навчання майбутніх інженерів-програмістів. Ukrainian journal of educational studies and information technology. 2017. Т. 5, № 2. С. 32–36.
6. Рисований М. Переваги та недоліки розробки ігор та пз на unity. VI Міжнародна науково-практична конференція "Інформаційна безпека та комп’ютерні технології" : Тези конф., м. Кропивницький, 20 квіт. 2023 р. Кропивницький, 2023. С. 62.
7. Шолудько О., Данилишин Р. Аспекти економічного розвитку України у сфері ігрової індустрії в умовах глобалізації. Економіко-соціальні відносини в галузі фізичної культури та сфері обслуговування : Тези доп., м. Львів, 21 верес. 2022 р. Львів, 2022. С. 58.
8. Adams E. Fundamentals of Game Design : навчальний посібник. 3rd ed. New Riders, 2014. 576 p.
9. Arenas D. L., Viduani A., Araujo R. B. Therapeutic use of role-playing game (RPG) in mental health: a scoping review. Simulation & gaming. 2022. 53(3), P., 285-311. URL: https://doi.org/10.1177/10468781211073720

10. EPIC Game Dev. Unity 2D - RPG tutorial 2024 - part 09 battle system design, 2023. YouTube. URL: https://www.youtube.com/watch?v=bl0OYtqaIWY
11. Esposito N. A short and simple definition of what a videogame is. Changing views: worlds in play : Матеріали конференції. 2005.
12. Fricker H. Game user-interface guidelines: creating a set of usability design guidelines for the FPS game user-interface : Masters Thesis. 2012. 119 p.
13. Heikkinen O. Hi-Bit pixel graphics – the new era of pixel art : bachelor's thesis. Pirkanmaa, 2021. 37 p.
14. Hocking J. Unity in Action: Multiplatform Game Developing in C#. 3rd ed. Manning Publications Company, 2022. 416 p.
15. Iantorno M., Consalvo M. Background checks: disentangling class, race, and gender in CRPG character creators. Games and culture. 2023. P. 979-1003. URL: https://doi.org/10.1177/15554120221150342
16. Jonsson J. Game development – using UML class diagram : Praktiskt examensarbete. Stockholm, 2012. 23 p.
17. Kylmäaho N. Pixel Graphics in Indie Games : bachelor's thesis. Pirkanmaa, 2019. 53 p.
18. Mallindine J.D. Ghost in the cartridge: nostalgia and the construction of the JRPG genre. Gamevironments. 2016. No. 5. P. 80–103.
19. Newzoo. Newzoo. URL: https://newzoo.com/.
20. Stenström C. D. Gameplay design for role-playing battle systems : Master of Science Thesis. Gothenburg, 2012. 179 p.
21. The effects of graphical fidelity on player experience / K. M. Gerling et al. International conference, Tampere, Finland, 1–4 October 2013. New York, New York, USA, 2013. URL: https://doi.org/10.1145/2523429.2523473
22. Unity Technologies. FPS mod: create enemy patrol paths (NEW). Unity Learn. URL: https://learn.unity.com/tutorial/fps-mod-create-enemy-patrol-paths
23. Unity Technologies. 2D Game Kit - Unity Learn. Unity Learn. URL: https://learn.unity.com/project/2d-game-kit

24. Unity Technologies. Introducing health. Unity Learn. URL: https://learn.unity.com/tutorial/introducing-health
25. Unity Technologies. Prefabs. Unity Manual. URL: https://docs.unity3d.com/Manual/Prefabs.html
26. Unity - Manual: 2D game development. Unity Documentation. URL: https://docs.unity3d.com/Manual/Unity2D.html
27. Zufri T., Ardani N. A. Reseach on the application of pixel art in game character design. IJVCDC (indonesian journal of visual culture, design, and cinema). 2023. Vol. 2, no. 1. P. 100–106. URL: https://doi.org/10.21512/ijvcdc.v2i1.8238

ДОДАТОК А

Скрипт пересування гравця PlayerMovement.cs

```
using UnityEngine;
using System;

public class PlayerMovement : MonoBehaviour
{
    private float horizontal = 0f;
    [SerializeField] private float speed = 8f;
    [SerializeField] private float jumpingPower = 16f;
    private bool isFacingRight = true;

    private Collider2D platformCollider;

    [SerializeField] private Rigidbody2D rb;
    [SerializeField] private Transform groundCheck;
    [SerializeField] private LayerMask groundLayer;
    [SerializeField] private Animator animator;

    [Header ("SFX")]
    [SerializeField] private AudioClip jumpSound;



    private void Awake() {
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
    }

    void Update() {

        if (DialogueManager.Instance.isDialogueActive){
            horizontal = 0f;
            animator.SetFloat("Speed", 0f);
            return;
        }

        if (DialogueManagerMessage.Instance.isMessageActive){
            horizontal = 0f;
            animator.SetFloat("Speed", 0f);
            return;
        }

        horizontal = Input.GetAxisRaw("Horizontal");
        animator.SetFloat("Speed", Mathf.Abs(horizontal));

        if (Input.GetButtonDown("Jump") && IsGrounded())
        {
```

```
            SoundManager.instance.PlaySound(jumpSound);
            rb.velocity = new Vector2(rb.velocity.x, jumpingPower);
        }

        if (Input.GetButtonUp("Jump") && rb.velocity.y > 0f)
        {
            rb.velocity = new Vector2(rb.velocity.x, rb.velocity.y * 0.5f);
        }

        Flip();
    }

    private void FixedUpdate()
    {
        rb.velocity = new Vector2(horizontal * speed, rb.velocity.y);
    }

    private bool IsGrounded()
    {
        return Physics2D.OverlapCircle(groundCheck.position, 0.2f, groundLayer);
    }

    private void Flip()
    {
        if (isFacingRight && horizontal < 0f || !isFacingRight && horizontal >
0f)
        {
            isFacingRight = !isFacingRight;
            Vector3 localScale = transform.localScale;
            localScale.x *= -1f;
            transform.localScale = localScale;
        }
    }

    public bool canAttack() {
        return horizontal == 0 && IsGrounded();
    }

    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Hole"))
        {
            Physics2D.IgnoreLayerCollision(10, gameObject.layer, true);
        }

        if (collision.gameObject.CompareTag("Ladder"))
        {

        }
    }
```

```
    void OnCollisionExit2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Hole"))
        {
            Physics2D.IgnoreLayerCollision(10, gameObject.layer, false);
        }
    }
}
```

ДОДАТОК Б

Скрипти діалогової системи

DialogueManager.cs

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

public class DialogueManager : MonoBehaviour
{
    public static DialogueManager Instance;
    [SerializeField] private Image characterIcon;
    [SerializeField] private TextMeshProUGUI dialogueArea;

    private Queue<DialogueLine> lines;

    public bool isDialogueActive = false;

    [SerializeField] private float typingSpeed = 0.2f;

    [SerializeField] private AudioClip sound;


    private void Awake()
    {
        if (Instance == null)
            Instance = this;

        lines = new Queue<DialogueLine>();

        gameObject.SetActive(false);
    }

    private void Update()
    {
        if (isDialogueActive && Input.GetKeyDown(KeyCode.Return))
        {
            DisplayNextDialogueLine();
        }
    }

    public void StartDialogue(Dialogue dialogue)
    {
        isDialogueActive = true;
        gameObject.SetActive(true);

        lines.Clear();
```

```
        foreach (DialogueLine dialogueLine in dialogue.dialogueLines)
        {
            lines.Enqueue(dialogueLine);
        }

        DisplayNextDialogueLine();
    }

    public void DisplayNextDialogueLine()
    {
        if (lines.Count == 0)
        {
            EndDialogue();
            return;
        }

        DialogueLine currentLine = lines.Dequeue();

        characterIcon.sprite = currentLine.character.icon;

        StopAllCoroutines();

        StartCoroutine(TypeSentence(currentLine));
    }

    IEnumerator TypeSentence(DialogueLine dialogueLine)
    {
        dialogueArea.text = "";
        foreach (char letter in dialogueLine.line.ToCharArray())
        {
            dialogueArea.text += letter;
            SoundManager.instance.PlaySound(sound);
            yield return new WaitForSeconds(typingSpeed);
        }
    }

    void EndDialogue()
    {
        isDialogueActive = false;
        gameObject.SetActive(false);
    }
}
```

DialogueTrigger.cs

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

```
[System.Serializable]
public class DialogueCharacter
{
    public Sprite icon;
}

[System.Serializable]
public class DialogueLine
{
    public DialogueCharacter character;
    [TextArea(3, 10)]
    public string line;
}

[System.Serializable]
public class Dialogue
{
    public List<DialogueLine> dialogueLines = new List<DialogueLine>();
}

public class DialogueTrigger : MonoBehaviour
{
    public Dialogue dialogue;

    public void TriggerDialogue()
    {
        DialogueManager.Instance.StartDialogue(dialogue);
    }
}
```

ДОДАТОК В

Скрипти ворогів

ToothWalker.cs

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

public class ToothWalker : MonoBehaviour
{
    [Header ("Attack Parameters")]
    [SerializeField] private float attackCooldown;
    [SerializeField] private int damage;
    [SerializeField] private float range;

    [Header ("Collider Parameters")]
    [SerializeField] private float colliderDistance;
    [SerializeField] private BoxCollider2D boxCollider;

    [Header ("Player Layer")]
    [SerializeField] private LayerMask playerLayer;
    private float cooldownTimer = Mathf.Infinity;

    [Header ("Sounds")]
    [SerializeField] private AudioClip biteSound;

    //References
    private Animator anim;
    private Health playerHealth;
    private EnemyPatrol enemyPatrol;

    private void Awake() {
        anim = GetComponent<Animator>();
        enemyPatrol = GetComponentInParent<EnemyPatrol>();
    }

    void Update()
    {
        cooldownTimer += Time.deltaTime;

        if (PlayerInSight()) {
            if (cooldownTimer >= attackCooldown){
                SoundManager.instance.PlaySound(biteSound);
                cooldownTimer = 0;
                anim.SetTrigger("toothWalkerAttack");
            }
        }

        if(enemyPatrol != null)
```

```
            enemyPatrol.enabled = !PlayerInSight();
    }

    private bool PlayerInSight(){
        RaycastHit2D hit = Physics2D.BoxCast(boxCollider.bounds.center +
transform.right * range * transform.localScale.x * colliderDistance,
                                             new
Vector3(boxCollider.bounds.size.x * range, boxCollider.bounds.size.y,
boxCollider.bounds.size.z),
                                             0, Vector2.left, 0, playerLayer);

        if(hit.collider != null){
            playerHealth = hit.transform.GetComponent<Health>();
        }

        return hit.collider != null;
    }

    private void OnDrawGizmos() {
        Gizmos.color = Color.red;
        Gizmos.DrawWireCube(boxCollider.bounds.center + transform.right * range *
transform.localScale.x * colliderDistance,
                            new Vector3(boxCollider.bounds.size.x * range,
boxCollider.bounds.size.y, boxCollider.bounds.size.z));
    }

    private void DamagePlayer() {

        if (PlayerInSight()){
            playerHealth.TakeDamage(damage);
        }
    }
}
```

EnemyPatrol.cs

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

public class EnemyPatrol : MonoBehaviour
{
    [Header ("Patrol Points")]
    [SerializeField] private Transform leftEdge;
    [SerializeField] private Transform rightEdge;

    [Header ("Enemy")]
    [SerializeField] private Transform enemy;
```

```
[Header ("Movement Parameters")]
[SerializeField] private float speed;
private Vector3 initScale;
private bool movingLeft;

[Header ("Enemy Animation")]
[SerializeField] private Animator anim;
[SerializeField] private float idleDuration;
private float idleTimer;

private void Awake() {
    initScale = enemy.localScale;
}

private void OnDisable() {
    anim.SetBool("moving", false);
}

private void Update() {

    if(movingLeft){

        if(enemy.position.x >= leftEdge.position.x)
            MoveInDirection(-1);
        else
            DirectionChange();

    } else {

        if(enemy.position.x <= rightEdge.position.x)
            MoveInDirection(1);
        else
            DirectionChange();
    }
}

private void DirectionChange(){
    anim.SetBool("moving", false);
    idleTimer += Time.deltaTime;

    if (idleTimer > idleDuration)
        movingLeft = !movingLeft;
}

private void MoveInDirection(int _direction) {
    idleTimer = 0;
    anim.SetBool("moving", true);

    enemy.localScale = new Vector3(Mathf.Abs(initScale.x) * _direction,
                                   initScale.y, initScale.z);
```

```
        enemy.position = new Vector3(enemy.position.x + Time.deltaTime *
_direction * speed,
                                            enemy.position.y, enemy.position.z);
    }
}
```