

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

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

« Гра в жанрі "зомбі шутер" »

Виконав студент групи ЗПЗк-22-1
НИКОНЕНКО Дмитро Олександрович

Консультант:
ЄФРЕМОВ Юрій Миколайович

Керівник роботи:
ЛЕВЧЕНКО Антон Юрійович

Рецензент:
ВЛАСЕНКО Олег Васильович

Житомир – 2024

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

«ЗАТВЕРДЖУЮ»
зав. кафедри інженерії
програмного забезпечення
Тетяна ВАКАЛЮК
«26» лютого 2024 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **НИКОНЕНКО Дмитро Олександрович**
Керівник роботи: **ЛЕВЧЕНКО Антон Юрійович**
Тема роботи: «Гра в жанрі "зомбі шутер"», затверджена Наказом закладу вищої освіти від «23» лютого 2024 р., №75/с
Вихідні дані для роботи: гра в жанрі "зомбі шутер"
Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Левченко А.Ю. Єфремов Ю.М.	02.03.2024р.	02.03.2024р.
2	Левченко А.Ю. Єфремов Ю.М.	04.04.2024р.	04.04.2024р.
3	Левченко А.Ю. Єфремов Ю.М.	02.05.2024р.	02.05.2024р.

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра складається з комп'ютерної гри та пояснювальної записки. Пояснювальна записка до випускної роботи містить 53 сторінки, 12 ілюстрацій, 1 додаток та 15 використаних джерел.

Випускна робота бакалавра - це розробка захоплюючої гри у жанрі "шутер зомбі", де гравці мають можливість боротися з віртуальними зомбі у напруженій і атмосферній грі. Цей проект включає створення програмного продукту, що дозволяє геймерам насолоджуватися різноманітними геймплейними можливостями, включаючи стрілянину, виживання та виконання завдань у вигаданому апокаліптичному світі.

У роботі я визначив ключові завдання для розробки гри, провів аналіз існуючих ігрових рішень у цьому жанрі, включивши сценарії гри та загальну структуру програми. Особливу увагу було приділено оптимізації графічних ефектів та перетворень для підвищення продуктивності гри та комфортного геймплею. Крім того, в роботі розглянуті варіанти вирішення завдань з оптимізації для підвищення швидкості обробки графіки та покращення загального враження від гри.

КЛЮЧОВІ СЛОВА: КОМП'ЮТЕРНА ГРА, ЮНІТІ, СКРИПТ, АССЕТ, ГЕЙМПЛЕЙ, ІНТЕРФЕЙС.

					ІПЗ.КР.Б – 121 – 24 - ПЗ		
Змн.	Арк.	№ докум.	Підпис	Дата			
Розроб.		Никоненко Д.О.			Гра в жанрі "зомбі шутер"	Літ.	Арк.
Перевір.		Єфремов Ю.М.,					3
Керівник		Левченков А. Ю				Аркушів	
Н. контр.		Тичина О.П.				53	
Зав. каф.		Вакалюк Т.А.				Житомирська політехніка, група ЗПЗк-22-1	

ABSTRACT

A bachelor's thesis consists of a computer game and an explanatory note. The explanatory note to the final thesis contains 53 pages, 12 illustrations, 1 appendix and 15 references.

This bachelor's thesis is about developing an exciting zombie shooter game where players have the opportunity to fight virtual thieves in an intense and atmospheric game. This project involves the creation of a software product that allows gamers to enjoy a variety of gameplay features, including shooting and survival in a fictional apocalyptic world.

In my work, I identified key tasks for the game development, analyzed existing game solutions in this genre, including game scenarios and the overall structure of the application. Particular attention was paid to optimizing graphic effects and transformations to improve game performance and comfortable gameplay. In addition, the paper considers options for solving optimization problems to increase graphics processing speed and improve the overall game experience.

KEYWORDS: COMPUTER GAME, UNITY, SCRIPT, ASSET, GAMEPLAY, INTERFACE.

					ПЗ.КР.Б – 121 – 24 - ПЗ	Арк.
						4

ЗМІСТ

РЕФЕРАТ	3
ABSTRACT	4
ЗМІСТ.....	5
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ	6
ВСТУП	7
РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ.....	9
1.1 Постановка задачі.....	9
1.2 Аналіз аналогів програмного продукту	10
1.3 Вибір інструменту для розробки гри “Зомбі шутер”	12
1.4 Обґрунтування вибору інструментальних засобів для розробки	13
Висновки до першого розділу.....	14
РОЗДІЛ 2. ПРОЕКТУВАННЯ РОЗРОБКИ ГРИ	15
2.1 Визначення варіантів використання	15
2.2. Проектування та реалізація алгоритмів роботи гри.....	16
Висновки до другого розділу	39
РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК ВИКОРИСТАННЯ ГРИ В ЖАНРІ “ЗОМБІ ШУТЕР”	40
3.1. Опис структури ігрового рівня	40
3.2 Структура інтерфейсу гри	41
3.3 Тестування продуктивності гри	49
Висновки до третього розділу	50
ВИСНОВКИ.....	51
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	52
ДОДАТКИ	54

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

Комп'ютерна гра — це інтерактивний цифровий продукт

Unity — інструмент для розробки ігор

Скрипти — це фрагменти коду, які визначають, як поведуться об'єкти в грі.

Асети — це всі ресурси, які використовуються для створення ігрового світу.

Геймплей — те, що робить гру захоплюючою та цікавою.

Інтерфейс — це набір елементів, з якими взаємодіє гравець під час гри.

Колізія — зіткнення двох або більше об'єктів у грі.

Коллайдери — це компоненти, які додаються до об'єктів у Unity для визначення їх фізичних меж.

ВСТУП

Актуальність теми. Розробка гри у жанрі "шутер зомбі" залишається актуальною темою в індустрії відеоігор з кожним роком. Цей жанр поєднує в собі елементи стрілянини, виживання та хоррору, що приваблює широке коло гравців. Зомбі завжди залишаються популярними противниками у відеоіграх завдяки своїй непередбачуваності та напруженій атмосфері, яку вони створюють.

Останні технологічні досягнення у галузі графіки, штучного інтелекту та фізичної моделювання дозволяють розробникам створювати більш реалістичні та захоплюючі ігрові світи з вражаючими зомбі та навколишніми об'єктами. Крім того, розвиток віртуальної реальності та інших інтерактивних технологій відкриває нові можливості для іммерсивного геймплею у жанрі "шутер зомбі".

З огляду на постійний попит гравців на ігри з зомбі, розробка гри у цьому жанрі є актуальною та перспективною для відданих геймерів та студій розробників відеоігор.

Мета даної кваліфікаційної роботи полягає у створенні захоплюючого та незабутнього ігрового досвіду для гравців. Гра повинна надати можливість насолоджуватися напруженою атмосферою, екшеном та стріляниною, які характерні для цього жанру. Крім того, вона повинна пропонувати цікавий сюжет, різноманітні механіки гри та викликати у гравців почуття відповідальності за власну виживання та стратегічного мислення для подолання викликів, що стоять перед ними. В цілому, мета полягає у створенні гри, яка буде цікавою та захоплюючою для широкого кола гравців у цьому популярному жанрі.

Встановлена мета обумовлює наступні завдання:

- створення напруженої атмосфери і геймплею, які зацікавлять гравців із першої миті;
- відтворення характерних елементів шутерів і зомбі-апокаліпсисів, таких як стрілянина та виживання у ворожому середовищі;
- впровадження різноманітних механік гри, щоб забезпечити різноманітність і гнучкість геймплею;
- створення відчуття відповідальності за виживання героя та розробка

стратегій для подолання викликів гри.

Об'єктом дослідження є процес створення гри додавання ассетів створення та створення скриптів.

Предметом дослідження є використання інструментарію розробки програмного забезпечення мовою програмування C# (Unity) для програмної гри. В роботі було використано методи об'єктно-орієнтованого програмування та дизайну інтерфейсів. Також використання інтеграції з Unity таких як Visual Studio забезпечує безшовну інтеграцію з Unity, що дозволяє розробникам писати і відлагоджувати код безпосередньо в середовищі Unity. Також Visual Studio має один з найкращих відлагоджувачів, що значно полегшує процес виявлення і виправлення помилок у коді. Visual Studio надає підказки, автодоповнення та інші інструменти, які допомагають писати код швидше і без помилок.

Практичне значення отриманих результатів: створена гра з можливостями керувати головним персонажем, з організованим головним меню також з доволі цікавою локацією.

Структура пояснювальної записки. складається з комп'ютерної гри та пояснювальної записки. Пояснювальна записка до випускної роботи містить 53 сторінку, 12 ілюстрацій, 1 додаток та 15 використаних джерел.

РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

1.1 Постановка задачі

Головною метою розробки гри є виживання в апокаліпсисі, знищуючи зомбі та виконуючи завдання для рятування головного героя. Гравці керують персонажем, який опинився в центрі зомбі-апокаліпсису. Вони повинні вижити, використовуючи зброю, ресурси та стратегії, щоб протистояти хвилям зомбі та виконувати завдання для рятування тих хто залишився в живих.

Для реалізації поставленого завдання, основними етапами є:

- Аналіз проекту та збір вимог:
- проведення ретельного аналізу практики використання подібного жанру ігор;
- визначення основних характеристик і функціональних можливостей гри;
- аналіз аналогів програмного продукту;
- Вибір та інтеграція технологій:
- дослідження та вибір платформи розробки;
- інтеграція стандартних інструментів та фреймворків, на яких буде працювати редактор;
- впровадження розгортання програмного продукту на будь-яку Windows-машину;
- Розробка та впровадження системи:
- розробка інтуїтивно зрозумілого та зручного інтерфейсу, який включає налаштування, меню, та інші елементи інтерфейсу;
- реалізація основних функцій: такі як локація, фізика переміщення персонажа, фізика бойових можливостей;
- реалізація алгоритмів: рух груп “зомбі”, фізика атаки;
- Тестування та забезпечення якості:
- проведення ретельного тестування, щоб гарантувати, що програмне рішення працює за призначенням і відповідає вимогам;
- впровадження процесів забезпечення якості для виявлення та вирішення

будь-яких проблем або помилок, які можуть виникнути на етапах розробки та тестування.

1.2 Аналіз аналогів програмного продукту

Існує багато аналогів ігор данного жанру кожна по своєму унікальна та має своє механіки, своє бачення графіки та інше. У цьому розділі ми проведемо аналіз двох відомих ігор: Alien Shooter, Dead Island.

Alien Shooter. Ця гра відома своїми швидкими та динамічними боями з великою кількістю ворогів на екрані одночасно. Вона має доволі примітивну графіку та механіки. Геймплей наведена на (рис.1.1).



Рис 1.1 – Ігровий процес Alien Shooter

Переваги:

- швидкий темп гри;
- велика різноманітність зброї;
- декілька типів ворогів.

Недоліки:

- немає сюжетної лінії;
- монотонні місії.

Dead Island — це популярна екшн-RPG гра, де гравцям доводиться виживати на тропічному острові, охопленому зомбі-апокаліпсисом. Вона відразу захопила

увагу геймерів своїм драматичним трейлером, який став вірусним завдяки своїй емоційності та кінематографічній якості. Геймплей наведено на (рис.1.2 та 1.3).



Рис.1.2 – Ілюстровано ігровий процес



Рис.1.3 – Проілюстровано якість локації та кількість персонажів для взаємодії

Переваги:

- відкритий світ;
- великий вибір зброї;
- можливість взаємодіяти з іншими персонажами.

Недоліки:

- можливі проблеми з провалами під мапу;

- можливі не реєстрування шкоди по ворогу;
- повторювання завдань.

Нижче представлено порівняльну таблицю проектів (табл.1.1).

Таблиця 1.1

Таблиця порівняння проектів

	Alien shooter	Dead island	Zombie game
Має графіку на вищому рівні	-	+	-
Взаємодія з персонажами	-	+	-
Операційна системи	Windows, android	Windows	Android
Доступність	+	-	+
Фізика	-	+	+
Швидкість та продуктивність	+	+	+

Беручи до уваги всі аспекти, ми можемо ретельно проаналізувати та включити найкращі функції та можливості Alien shooter, Dead Island у наш власний продукт. Такий інтегрований підхід гарантує, що наш продукт не лише задовольняє різноманітні потреби користувачів, але й використовуємо сильні сторони цих визнаних ігор.

1.3 Вибір інструменту для розробки гри “Зомбі шутер”

Для створення даної гри використовується мова програмування C# так як вона добре інтегрується з іншими технологіями та фреймворками, такими як Unity (популярний інструмент для розробки ігор), що полегшує роботу з графікою, звуком та фізикою у грі. Мова програмування C# є високопродуктивною, що дозволяє створювати ігри з високим рівнем деталізації та складності без втрати швидкодії.

Unity - це одна з найпопулярніших інтегрованих середовищ для розробки ігор. Вона надає розробникам зручний інтерфейс та потужні інструменти для створення ігор для різних платформ, включаючи ПК, консолі, мобільні пристрої.

Основні переваги використання Unity для розробки ігор:

- Кросплатформеність: Unity дозволяє створювати ігри для різних платформ, використовуючи один і той же код. Це дозволяє розробникам ефективно використовувати ресурси та швидко розгортати гру на різних пристроях.
- Графіка та анімація: Unity має потужний двигун для рендерингу, що дозволяє створювати вражаючі візуальні ефекти та реалістичну анімацію. Вона підтримує використання шейдерів та спеціальних ефектів для досягнення бажаного вигляду гри.
- Фізика: Unity має вбудовану систему фізики, яка дозволяє моделювати реалістичну поведінку об'єктів у грі. Це дозволяє створювати ігри з реалістичною фізикою та взаємодією об'єктів.
- Швидкість розробки: Unity надає інструменти для швидкої розробки прототипів та ігор. Це дозволяє розробникам швидко перевіряти ідеї та вносити зміни у гру.

1.4 Обґрунтування вибору інструментальних засобів для розробки

Використання мови програмування C# та середовища розробки Unity для створення ігор має кілька переваг:

- Простота вивчення: C# є однією з найпопулярніших мов програмування, що використовуються для розробки ігор. Вона має чистий синтаксис та велику кількість документації, що робить її досить легкою для вивчення, навіть для початківців.
- Кросплатформеність: Unity дозволяє створювати ігри для різних платформах, включаючи ПК, консолі, мобільні пристрої та веб. Використання C# дозволяє розробникам писати один і той же код, який працюватиме на різних платформах, що зменшує затрати на розробку та підтримку.
- Потужність інструментів: Unity надає розробникам широкий спектр інструментів для розробки ігор, включаючи графіку, анімацію, фізику та штучний інтелект. Використання C# дозволяє легко і ефективно використовувати ці інструменти для створення складних та цікавих ігор.

- Швидкість розробки: Використання C# та Unity дозволяє розробникам швидко створювати прототипи та ігри завдяки широкому спектру готових рішень та інструментів. Це дозволяє прискорити процес розробки та випуску гри на ринок.

- Активна спільнота та підтримка: Unity має велику та активну спільноту розробників, яка постійно доповнюється новими ресурсами та підтримкою. Це дозволяє швидко знаходити відповіді на питання та вирішувати проблеми під час розробки.

Visual Studio:

- Інтеграція з Unity: Visual Studio забезпечує безшовну інтеграцію з Unity, що дозволяє розробникам писати і відлагоджувати код безпосередньо в середовищі Unity.

- Потужний відлагоджувач: Visual Studio має один з найкращих відлагоджувачів, що значно полегшує процес виявлення і виправлення помилок у коді.

- Інтелектуальна підтримка коду: Visual Studio надає підказки, автодоповнення та інші інструменти, які допомагають писати код швидше і без помилок.

Висновки до першого розділу

У першому розділі роботи було описано постановку задачі, виокремлюючи мету розробки гри.

Проведено глибокий аналіз аналогів програмного продукту, зосередившись на двох відомих іграх, а саме: Alien shooter, Dead Island. Цей аналіз висвітлив особливості, функціональність, переваги та недоліки кожної гри, що дозволило краще зрозуміти існуючі рішення на ринку, щоб розробити максимально зручний і корисний продукт.

Крім того, розділ заглибився у важливий аспект вибору архітектури для розробки гри. Було розглянуто середовище для розробки ігор Обґрунтування вибору інструментів та вимог до апаратного забезпечення базувалося на їх сумісності, продуктивності та кращих галузевих практиках.

Було описано обране середовище розробки на основі використання мови C#.

РОЗДІЛ 2. ПРОЕКТУВАННЯ РОЗРОБКИ ГРИ

2.1 Визначення варіантів використання

При розробці зомбі шутера на Unity необхідно враховувати різні аспекти, щоб забезпечити ефективну та зручну структуру коду, а також оптимальну продуктивність гри. У цьому розділі розглянемо основні варіанти використання гри та об'єктно-орієнтовану структуру, яка буде застосована для розробки проекту.

Отже, коли ми говоримо про визначення варіантів використання та об'єктно-орієнтовану структуру, ми насправді маємо на увазі процес планування та розробки програмного забезпечення, яке буде спрямоване на вирішення конкретних завдань.

По суті, це означає, що ми спочатку ретельно аналізуємо потреби користувачів та вимоги до системи. Нам треба зрозуміти, що саме має робити наше програмне забезпечення і які вимоги воно повинно задовольняти.

Потім ми розглядаємо можливі способи побудови програми, використовуючи об'єктно-орієнтований підхід. Це означає, що ми дивимось, як ми можемо організувати програму навколо окремих об'єктів, які взаємодіють один з одним. Це дозволяє нам створювати більш модульні та ефективні програми.

Далі ми вибираємо конкретні варіанти використання програмного забезпечення. Це можуть бути різні сценарії використання, які відповідають потребам користувачів. Наприклад, для веб-додатків це можуть бути сценарії авторизації користувачів або обробки запитів.

Після цього ми переходимо до фактичної реалізації обраного підходу, використовуючи об'єктно-орієнтовану структуру. Ми створюємо необхідні класи, методи та інші елементи програми з урахуванням нашого планування.

Завершальним кроком є тестування та перевірка програми, щоб переконатися, що вона працює як потрібно та відповідає всім вимогам. Це допомагає нам переконатися, що обрані варіанти використання та структура програми є ефективними та надійними.

Чітке визначення варіантів використання та створення об'єктно-орієнтованої структури є важливими аспектами успішної розробки зомбі шутера на Unity. Визначення варіантів використання дозволяє зрозуміти, які функції необхідно

реалізувати в грі, що допомагає зосередити процес розробки на ключових аспектах. Об'єктно-орієнтована структура надає проекту модульність і гнучкість, що значно полегшує його підтримку та розширення в майбутньому.

Структурування гри за допомогою визначених класів, таких як гравець, зомбі, рівень, зброя та інтерфейс користувача, робить код зрозумілим та логічним. Це дозволяє розробникам легко додавати нові функції або змінювати існуючі компоненти з мінімальними зусиллями.

Таким чином, правильне визначення варіантів використання та об'єктно-орієнтована структура є основою для створення якісного, функціонального та легкого в підтримці програмного продукту. Ці фактори дозволяють команді розробників зосередитися на поліпшенні геймплею та користувацького досвіду, що є ключовими елементами успіху гри.

2.2. Проектування та реалізація алгоритмів роботи гри

Після завантаження головного меню гри користувач може вибрати наступні дії: Старт, Меню налаштування, Кнопка виходу з гри, Профіль гравця.

Після натиску кнопки старт:

- Перед гравцем з'явиться меню вибора рівнів;
- Вибір рівнів буде зроблений в текстовому варіанті в стовбчик.

Так як це мобільний пристрій кнопка виходу буде добавлена для більш легкого виходу з гри.

В меню налаштувань гравцеві будуть доступні налаштування якості картинки та тіней для більш комфортної кількості кадрів під різні пристрої. Доступними будуть наступні рівні зображення:

- Низька;
- Середня;
- Висока;
- Ультра.

```
public void OnButtonClickUltra()
```

```
{
```

```
    SetGraphicsLevel("Ultra"); // Встановлюємо рівень Ultra
```

```

OutlineLow.SetActive(true);
OutlineLowSet.SetActive(false);
OutlineMedium.SetActive(true);
OutlineMediumSet.SetActive(false);
OutlineHigh.SetActive(true);
OutlineHighSet.SetActive(false);
OutlineUltra.SetActive(false);
OutlineUltraSet.SetActive(true);
}

private void SetGraphicsLevel(string graphicsLevel)
{
    // Зберігаємо рівень налаштувань графіки
    PlayerPrefs.SetString("GraphicsLevel", graphicsLevel);
    PlayerPrefs.Save();
    // Застосовуємо налаштування графіки
    ApplyGraphicsSettings(graphicsLevel);
}

private void ApplyGraphicsSettings(string graphicsLevel)
{
    // Виконуємо зміни графіки відповідно до вибраного рівня
    switch (graphicsLevel)
    {
        case "Low":
            QualitySettings.SetQualityLevel(1, true);
            break;
        case "Medium":
            QualitySettings.SetQualityLevel(2, true);
            break;
        case "High":
            QualitySettings.SetQualityLevel(3, true); // Високі налаштування
    }
}

```

```

        break;
    case "Ultra":
        QualitySettings.SetQualityLevel(5, true); // Найвищі налаштування
        break;
    default:
        Debug.LogWarning("Unsupported graphics level");
        break;
    }
}

```

Цей код відповідає за керування налаштуваннями графіки в грі Unity. Він дозволяє змінювати рівні графіки (низький, середній, високий, ультра) і зберігати вибрані налаштування для майбутніх запусків гри. Ось основні функції цього коду:

Оголошення змінних:

- Визначаються публічні змінні для різних об'єктів, які представляють різні рівні графіки (OutlineLow, OutlineMedium, OutlineHigh, OutlineUltra) та їхні набори (OutlineLowSet, OutlineMediumSet, OutlineHighSet, OutlineUltraSet).

Метод `Start`:

- При запуску гри перевіряється, чи існують збережені налаштування графіки у `PlayerPrefs`. Якщо ні, за замовчуванням встановлюються середні налаштування.
- Активуються відповідні об'єкти для вибраного рівня графіки.
- Застосовуються налаштування графіки.

Методи для кнопок:

- `OnButtonClickLow()`, `OnButtonClickMedium()`, `OnButtonClickHigh()`, `OnButtonClickUltra()`: Ці методи викликаються при натисканні відповідних кнопок в інтерфейсі користувача. Вони встановлюють новий рівень графіки, активують/деактивують відповідні об'єкти та зберігають вибрані налаштування.

Метод `SetGraphicsLevel`:

- Зберігає вибраний рівень графіки у `PlayerPrefs` і застосовує його.

Метод `ApplyGraphicsSettings`:

- Встановлює рівень якості графіки у налаштуваннях Unity ('QualitySettings') відповідно до вибраного рівня.

Основна ідея полягає в тому, щоб забезпечити можливість користувачеві вибирати рівень графіки, який найкраще відповідає можливостям його пристрою, та зберігати цей вибір для майбутніх запусків гри.

Також буде стоступне ввімкнення та вимкнення тіней.

```
public void DisableShadows()
{
    shadowsEnabled = false;
    // Зберігаємо стан тіней
    PlayerPrefs.SetInt("ShadowsEnabled", 0);
    PlayerPrefs.Save();
    // Застосовуємо налаштування тіней
    ApplyShadowSettings(false);
    OutlineShadowON.SetActive(true);
    OutlineShadowsONSel.SetActive(false);
    OutlineShadowOFF.SetActive(false);
    OutlineShadowsOFFSel.SetActive(true);
}

private void ApplyShadowSettings(bool enableShadows)
{
    // Увімкнути або вимкнути тіні відповідно до вибраного стану
    if (enableShadows)
    {
        QualitySettings.shadows = ShadowQuality.All;
    }
    else
    {
        QualitySettings.shadows = ShadowQuality.Disable;
    }
}
```

}

Цей код відповідає за керування тінями в грі Unity. Він дозволяє вмикати або вимикати тіні і зберігати цей стан між сесіями гри. Ось основні функції цього коду:

1. Оголошення змінних:

- ``onButton`` і ``offButton``: Кнопки для ввімкнення та вимкнення тіней.
- ``shadowsEnabled``: Булева змінна, яка зберігає стан тіней (включені або вимкнені).
- ``OutlineShadowON``, ``OutlineShadowsONSel``, ``OutlineShadowOFF``, ``OutlineShadowsOFFSel``: Об'єкти, які представляють різні стани тіней у UI (увімкнені або вимкнені).

Метод ``Start``:

- При запуску гри перевіряється, чи існують збережені налаштування тіней у ``PlayerPrefs``. Якщо налаштування існують, встановлюється відповідний стан тіней.
- Залежно від збереженого стану тіней, активуються відповідні об'єкти UI.
- Застосовуються збережені налаштування тіней.
- Встановлюються обробники подій для кнопок ``onButton`` і ``offButton``.

Методи для кнопок:

- ``EnableShadows()``: Викликається при натисканні кнопки для увімкнення тіней. Зберігає новий стан тіней, застосовує налаштування тіней та оновлює об'єкти UI.
- ``DisableShadows()``: Викликається при натисканні кнопки для вимкнення тіней. Зберігає новий стан тіней, застосовує налаштування тіней та оновлює об'єкти UI.

Метод ``ApplyShadowSettings``:

- Встановлює якість тіней у налаштуваннях Unity (``QualitySettings``) відповідно до вибраного стану тіней (увімкнені або вимкнені).

Основна ідея цього коду полягає в тому, щоб надати користувачеві можливість увімкнути або вимкнути тіні в грі і зберегти цей вибір для майбутніх запусків гри.

В профілі гравця буде доступна наступна інформація:

- Статистика за рівень
- Ім'я гравця (по стандарту при першому запуску гри гравцеві буде

надано початкове ім'я "Player # від 1 до 1000") //змінна імені буде статичною

```
using System.Collections;
```

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

```
using UnityEngine;
```

```
using TMPro;
```

```
public class NickName : MonoBehaviour
```

```
{ [Header("Text")]
```

```
    public TMP_Text PlayerName;
```

```
    public static string playerName;
```

```
    #region program
```

```
    void Start()
```

```
    {
```

```
        if(!PlayerPrefs.HasKey("PlayerNameReal"))
```

```
        {
```

```
            playerName = "Player #" + Random.Range(1, 1000).ToString();
```

```
        }
```

```
    else
```

```
    {
```

```
        playerName = PlayerPrefs.GetString("PlayerNameReal");
```

```
    }
```

```
    PlayerName.text = playerName.ToString();
```

```
}
```

```
private void Update()
```

```
{
```

```
    PlayerName.text = playerName.ToString();
```

```
    PlayerPrefs.SetString("PlayerNameReal", playerName);
```

```
}
```

```
#endregion
}
```

Призначення коду:

Цей скрипт призначений для роботи з ім'ям гравця в грі, створений на Unity. Він виконує такі основні функції:

- Встановлює початкове ім'я гравця при запуску гри.
- Зберігає ім'я гравця в локальному сховищі (PlayerPrefs), щоб воно зберігалось між сесіями гри.
- Постійно оновлює текстове поле на екрані, щоб відображати поточне ім'я гравця.

Цей скрипт може бути корисним для ігор, де потрібно відображати ім'я гравця, наприклад, у багатокористувацьких режимах або для персоналізації досвіду гри.

- Гравець зможе змінити своє ім'я на власне яке забажає.

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

public class ChangeNickName : MonoBehaviour
{
    public TMP_InputField NickNameNew;
    public TMP_Text text;
    public TMP_Text Error;
    string newName;
    #region program
    void Start()
    {
        PlayerPrefs.GetString("PlayerNameReal", NickName.playerName);
        newName = NickName.playerName;
        text.text = NickName.playerName;
    }
}
```

```

}
#endregion
#region buttons
public void OnButtonClickOk()
{
    string EnterName = NickNameNew.text;
    if (EnterName.Length > 32 && EnterName.Length > 0)
    {
        Error.text = "Max name length 32";
        StartCoroutine(HideResultText(2.5f));
        return;
    }
    else if(EnterName.Length <= 0)
    {
        Error.text = "Min name length 1";
        StartCoroutine(HideResultText(2.5f));
        return;
    }
    else if(EnterName.Length < 33 && EnterName.Length > 0)
    {
        newName = EnterName.ToString();
        NickName.playerName = newName;
        text.text = EnterName.ToString();
        PlayerPrefs.SetString("PlayerNameReal", NickName.playerName);
    }
}
#endregion
#region coroutine
private IEnumerator HideResultText(float delay)
{

```

```

        field return new WaitForSeconds(delay);
        Error.text = "";
    }
#endregion
}

```

Призначення коду:

Цей скрипт дозволяє гравцям змінювати своє ім'я в грі, перевіряючи його на відповідність заданим обмеженням довжини. Якщо ім'я відповідає умовам, воно зберігається і відображається на екрані. Якщо ім'я не відповідає умовам, відображається відповідне повідомлення про помилку на кілька секунд.

В статистиці за рівень буде доступна така інформація:

- Кількість знешкоджених ворожих цілей;
- Тип знешкоджених цілей;
- Кількість програвів гравця на поточному рівні і в цілому по всіх рівнях разом;
- Кількість використаних аптечок на поточному рівні і в цілому по всіх рівнях разом.

Опис функціоналу мінікарти:

Мінікарта в данній грі потрібна для того аби гравець міг швидко продумати та оцінити свої дії наперед. На мінікарті доступні наступні дані:

- 2D вигляд рельєфу зверху;
- Зелена мітка гравця завдяки якій гравець може зрозуміти де він знаходиться;
- Червона мітка ворогів завдяки якій гравець може зрозуміти як далеко від нього ворожий персонаж;
- Місце розташування аптечки для того аби гравець міг зрозуміти де знаходиться аптечка.

Для мінікарти використовується власна камера та власні теги для таких об'єктів як маркери для ворогів, гравця та аптечки. Дані теги слугують для того аби вони не були видимими на основній камери.

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class MiniMap : MonoBehaviour
{
    [SerializeField] private Transform _player;
    // Update is called once per frame
    void LateUpdate()
    {
        Vector3 newPosition = _player.position;
        newPosition.y = transform.position.y;
        transform.position = newPosition;
        //transform.rotation = Quaternion.Euler(90f, _player.eulerAngles.y, 90f);
    }
}

```

Призначення коду:

Цей скрипт реалізує просту функцію мінікарти, яка слідує за позицією гравця в горизонтальній площині. Висота мінікарти залишається фіксованою, щоб уникнути зміщень, які можуть виникати через зміну висоти гравця (наприклад, підйоми на пагорби або стрибки).

Закоментований код:

```
//transform.rotation = Quaternion.Euler(90f, _player.eulerAngles.y, 90f);
```

Цей рядок закоментовано, але він вказує на можливість обертання мінікарти відповідно до обертання гравця. Якщо цей рядок раскоментувати, то мінікарта буде також обертатися відповідно до кута обертання гравця по осі у.

Цей скрипт можна розширити для додаткової функціональності, наприклад, для відображення різних маркерів на мінікарті, масштабування мінікарти або обертання мінікарти разом з гравцем.

Опис функціоналу аптечки:

Аптечка потрібна для того аби підняти рівень свого здоров'я якщо його

бракує. Також вона трішки полегшує геймплей гри, так як гравець використавши аптечку не втрачає надію що він ото от програє.

Аптечка має свій власний варіант префабу який включає:

- Box collider аби зчитувати коли гравець ввійшов в колізію;
- Tag “First Aid” аби зчитувало що гравець ввійшов саме в потрібну нам колізію.

Головний код використання аптечки міститься в скрипті Player який прикріплений до гравця. Як саме працює даний шматок коду:

Коли гравець ввійшов в Box Collider аптечки і Tag колізії “First Aid”, то з’являється кнопка для того аби використати аптечку. Коли гравець вийшов з Box Collider аптечки і Tag колізії “First Aid”, то кнопка для того аби використати аптечку зникає.

За те чи потрібно гравцеві добавляти здоров’я чи ні теж контролює свій власний шматочок коду. Він працює по такому принципу:

```
public void OnButtonClickAid()
{
    if(health <= Health && firstAid <= 50 && firstAid > 0)
    {
        if(gameObject.CompareTag("Aptechka"))
        {
            Destroy(gameObject);
        }
        health += firstAid;
        firstAid -= firstAid;
        button.SetActive(false);
        AidUsed = 1;
    }
    else if(health <= Health && firstAid > 50)
    {
        health += 50;
    }
}
```

```

        firstAid -= 50;
        Destroy(FirstAidObject);
        button.SetActive(false);
        AidUsed = 1;
    }
    healthbar.UpdateHealthbar(maxHealth, health);
}

```

Призначення коду:

Цей метод відповідає за логіку використання аптечки, що збільшує здоров'я гравця, а також оновлює індикатор здоров'я. Він перевіряє кількість аптечки і здоров'я гравця перед тим, як виконати дії, щоб забезпечити коректне використання ресурсів і оновлення інтерфейсу.

- Оголошено приватну статичну змінну скільки отримано шкоди.

[Header("Health")]

```

private int healthCount;
public static int maxHealth;
private static int health;
public static int firstAid;

```

Пояснення змін.

Перевірки та логіка в OnButtonClickAid:

- Перевіряє, чи поточне здоров'я health менше або дорівнює максимальному здоров'ю maxHealth.
- Використовує аптечку для збільшення здоров'я гравця, коректно змінюючи значення firstAid та health.
- Знищує об'єкт аптечки, коли вона використовується.
- Оновлює індикатор здоров'я за допомогою методу UpdateHealthbar.

Оновлення індикатора здоров'я:

- healthbar.UpdateHealthbar(maxHealth, health); - оновлює відображення здоров'я гравця на екрані, використовуючи значення maxHealth і health.

Ці змінні дозволяють зберігати інформацію про стан здоров'я гравця і

кількість аптечок, що забезпечує коректну роботу механізму відновлення здоров'я в грі.

- При натиску на кнопку “use (використати)” йде вже перевірка якщо шкода менше 50% то аптечка додає всього стільки здоров'я, скільки було отримано шкоди (після чого віднімає статичну зміну шкоди аби значення дорівнювало 0).

- Якщо отримано шкоди 50% або більше то аптечка додає всього 50% здоров'я і запускається скрипт спавна.

- Якщо шкоди зовсім не отримано аптечка не використовується.

- Після використання аптечки кнопка “use (використати)” зникає та зникає об'єкт аптечки аби не засмічувати дерево об'єктів (Hierarchy) та спрацьовує скрипт спавна аптечки.

Опис функціоналу спавну.

Для спавна в мене використано два різні скрипти, один під ворогів а другий під аптечки. Скрипт під спавн ворогів працює так:

При старті гри спавниться вказана кількість ворогів наприклад 6 коли гравець їх знешкодив то спавняються нові. Кількість ворогів які появляються ви вказуємо в налаштуваннях спавну в інспекторі. Для спавну нам потрібно пусті об'єкти в якості спавнпоінтів ці об'єкти, та префаб копію якого потрібно заспавнити.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;
public class ZombieSpawner : MonoBehaviour
{
    [Header("Prefab")]
    public GameObject zombiePrefab;
    [Header("Count of spawned zombie")]
    [SerializeField] private int numberOfZombie;
    public static int killedZombie;
    [Header("Spawnpoint")]
```

```

public Transform[] spawnPoints;
void Start()
{
    killedZombie = 0;
    SpawnZombie();
}
void SpawnZombie()
{
    for (int i = 0; i < numberOfZombie; i++)
    {
        int randomSpawnIndex = Random.Range(0, spawnPoints.Length);
        Transform spawnPoint = spawnPoints[randomSpawnIndex];
        Instantiate(zombiePrefab, spawnPoint.position, spawnPoint.rotation);
    }
}
void Update()
{
    if (killedZombie >= 6)
    {
        killedZombie = 0;
        SpawnZombie();
    }
}

```

Призначення коду.

Цей скрипт відповідає за:

- Спавн початкової групи зомбі на початку гри.
- Відстеження кількості знищених зомбі.
- Повторний спавн зомбі після того, як певна кількість зомбі буде знищена.

Можливі покращення:

1. Параметрична зміна кількості зомбі для повторного спавну.
 - Ви можете зробити параметр для кількості зомбі, які потрібно знищити перед повторним спавном.
2. Збільшення складності з часом.
 - Наприклад, збільшення кількості зомбі, що спавняться, кожного разу.
3. Використання подій Unity.
 - Використання Unity Events для більшого контролю над подіями в грі, такими як знищення зомбі та спавн нових зомбі.

Скрипт спавну аптечки працює по такому ж принципу але на відміну від ворогів де їх потрібно знешкодити, аби аптечка появилась потрібно використати її і розпочнеться відлік коли заспауниться нова наприклад через 30 секунд, після чого вона з'явиться на мінікарті та рівні.

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

public class FirstAidSpawner : MonoBehaviour
{
    [Header("Prefab")]
    public GameObject firstAidPrefab;
    [Header("Count of spawned aid")]
    [SerializeField] private int numberOfFirstAid;
    [Header("Spawnpoint")]
    public Transform[] spawnPoints;
    [Header("Respawn time")]
    public float respawnTime = 10f; // Час респауну в секундах
    void Start()
    {
        Player.AidUsed = 0;
    }
}
```

```

        SpawnAid();
    }
    void SpawnAid()
    {
        for (int i = 0; i < numberOfFirstAid; i++)
        {
            int randomSpawnIndex = Random.Range(0, spawnPoints.Length);
            Transform spawnPoint = spawnPoints[randomSpawnIndex];
            Instantiate(firstAidPrefab, spawnPoint.position, spawnPoint.rotation);
        }
    }
    void Update()
    {
        if (Player.AidUsed == 1)
        {
            GameObject[] firstAidInstances =
GameObject.FindGameObjectsWithTag("Aptechka");
            // Check if there are any instances
            if (firstAidInstances.Length > 0)
            {
                // Get the first instance and destroy it
                Destroy(firstAidInstances[0]);
                StartCoroutine(RespawnAid());
            }
            Player.AidUsed = 0;
            IEnumerator RespawnAid()
            {
                // Wait for the specified respawn time
                yield return new WaitForSeconds(respawnTime);
                // Respawn the first aid

```

```

        SpawnAid();
    }
}
}
}

```

Призначення коду.

Цей скрипт забезпечує:

- Спавн початкової кількості аптечок на початку гри.
- Відстеження використання аптечок гравцем.
- Респавн нових аптечок через заданий час після того, як аптечка була використана.

Можливі покращення:

1. Додавання більшої гнучкості.
 - Додавання можливості налаштувати різні параметри (кількість аптечок, час респавну) через інспектор Unity.
2. Збільшення складності:
 - Додавання логіки для зміни часу респавну або кількості аптечок в залежності від прогресу гравця в грі.
3. Використання подій Unity:
 - Використання подій Unity для більш гнучкого управління подіями в грі, такими як використання аптечки та респавн нових аптечок.

Опис функціоналу камер.

На рівні присутні дві камери одна для мінікарти друга для персонажа.

Камера, яка призначена для персонажа має скрипт, який допомагає їй слідкувати за гравцем.

```

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

public class PlayerCamera : MonoBehaviour
{

```

```

Player player;
Vector3 offset;
// Start is called before the first frame update
void Start()
{
    player = FindObjectOfType<Player>();
    offset = transform.position - player.transform.position;
}
// Update is called once per frame
void LateUpdate()
{
    transform.position = player.transform.position + offset;
}
}

```

Призначення коду.

Цей скрипт забезпечує:

- Камера постійно слідує за гравцем, підтримуючи фіксовану відстань від нього.
- Використання методу LateUpdate гарантує, що камера оновлюється після всіх інших оновлень, що мінімізує ривки в русі камери та гравця.

Можливі покращення:

1.Додаткові налаштування камери:

- Додати інтерфейс для налаштування відступу камери через інспектор Unity.

2.Плавне слідування:

- Використовувати інтерполяцію (Vector3.Lerp або Vector3.SmoothDamp) для більш плавного руху камери.

3.Обмеження руху камери:

- Додати обмеження для руху камери, щоб уникнути переміщення за межі ігрового світу.

- А камера що для мінікарти призначена аби проектувати зображення зверху для того аби гравець знав де конкретно він знаходиться.

- Для цих камер є також налаштування, основні налаштування це слої які вони можуть рендити наприклад те що потрібно аби показувалось на камері гравця знаходиться під тегом “Not for Minimap”, а те що для мінікарти “Minimap”.

Опис функціоналу хелсбару

Над гравцем та ворогами я зробив хелсбар який відображає рівень здоров'я. Хелсбар знаходиться в 3Д просторі над персонажами. Також для більш ефектного та красивого візуала я додавив з AssetStore DOTween плагін. Цей плагін робить ефект пошкодження більш красивим а саме коли гравець або ворог отримує хіт то хелсбар при відніманні здоров'я програє анімацію жовтої лінії, яка появляється лише на короткий проміжок часу.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using DG.Tweening;
public class HelthBar : MonoBehaviour
{
    private Slider _helthBarSlider;
    [SerializeField] private Slider damageEffectSlider;
    private Quaternion _startRotation;
    private void Awake()
    {
        _startRotation = transform.rotation;
        _helthBarSlider = GetComponent<Slider>();
    }
    private void LateUpdate()
    {
        transform.rotation = _startRotation;
```

```

    }
    public void UpdateHealthbar(int maxHealth, int currentHealth)
    {
        _helthBarSlider.maxValue = maxHealth;
        _helthBarSlider.minValue = 0;
        _helthBarSlider.value = currentHealth;
        damageEffectSlider.maxValue = maxHealth;
        DamageEffectAnimation(currentHealth);
    }
    private void DamageEffectAnimation(int targetValue)
    {
        damageEffectSlider.DOKill();
        damageEffectSlider.DOValue(targetValue, 0.5f);
    }
}

```

Призначення коду.

Цей скрипт забезпечує:

- Відображення поточного рівня здоров'я гравця.
- Анімацію зниження рівня здоров'я для покращеного візуального ефекту.
- Утримання здоров'я бар в одній і тій же позиції незалежно від руху та обертання об'єкта.

Можливі покращення:

1. Додаткові візуальні ефекти:
 - Додавання різних кольорів для різних рівнів здоров'я (наприклад, зелений для високого здоров'я, жовтий для середнього та червоний для низького).
2. Додаткові анімації:
 - Анімації для відновлення здоров'я (якщо в грі є механізм відновлення).

3. Оптимізація:

- Зменшення частоти викликів LateUpdate, якщо здоров'я бар не змінюється часто.

Опис функціоналу локалізації:

Для більш зручного і зрозумілого користування інтерфейсом в грі буде доступно декілька мов Англійська та Українська.

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

public class LocalizationManager : MonoBehaviour
{
    public void Ukrainian()
    {
        string language = "Ukrainian";
        PlayerPrefs.SetString("Language", language);
    }
    public void English()
    {
        string language = "English";
        PlayerPrefs.SetString("Language", language);
    }
}
```

Даний код відповідає за зміну мови в грі Unity і зберігання вибраної мови в налаштуваннях користувача. Нижче наведено докладні пояснення основних функцій цього коду:

1. Метод Ukrainian:
 - Встановлює рядок "Ukrainian" як вибрану мову.
 - Зберігає цей рядок у PlayerPrefs з ключем "Language".
2. Метод English:

- Встановлює рядок "English" як вибрану мову.
- Зберігає цей рядок у PlayerPrefs з ключем "Language".

3. Використання PlayerPrefs:

- PlayerPrefs є зручним способом зберігання і доступу до простих даних, таких як налаштування користувача, між сесіями гри.

- SetString зберігає рядкове значення за заданим ключем (у цьому випадку, ключ - "Language").

4. Недостатній функціонал:

- У цьому коді немає функціоналу для застосування обраної мови до інтерфейсу гри. Для повноцінної локалізації потрібно додати механізм, який буде використовувати збережене значення мови і відповідно змінювати текст у грі.

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

public class TextLanguage : MonoBehaviour
{
    public static string language;
    private TMP_Text textMesh;
    public string textUkrainian;
    public string textEnglish;
    // Start is called before the first frame update
    void Start()
    {
        textMesh = GetComponent<TMP_Text>();
        language = PlayerPrefs.GetString("Language");
        if (language == "" || language == "English")
        {
            textMesh.text = textEnglish;
        }
    }
}
```

```

    }
    else if (language == "Ukrainian")
    {
        textMesh.text = textUkrainian;
    }
}
// Update is called once per frame
void Update()
{
    language = PlayerPrefs.GetString("Language");
    if (language == "" || language == "English")
    {
        textMesh.text = textEnglish;
    }
    else if (language == "Ukrainian")
    {
        textMesh.text = textUkrainian;
    }
}
}

```

Цей код забезпечує динамічну зміну тексту у відповідності до обраної мови в грі Unity. Використовується TextMeshPro для відображення тексту. Код перевіряє збережену мову у PlayerPrefs і змінює текстовий компонент відповідно. Ось більш детальний розгляд функціоналу та пропозиції щодо покращення:

1. Оголошення змінних:
 - language: Статична змінна, що зберігає вибрану мову.
 - textMesh: Компонент TMP_Text, що використовується для відображення тексту.
 - textUkrainian та textEnglish: Рядки, що зберігають текст для відповідних мов.

2. Метод Start:

- Ініціалізується компонент TMP_Text.
- Зчитується збережена мова із PlayerPrefs.
- Залежно від збереженої мови, встановлюється відповідний текст.

3. Метод Update:

- Кожного кадру зчитується мова із PlayerPrefs і відповідно змінюється текст.
- Це може бути неефективним, оскільки перевірка та встановлення тексту в кожному кадрі є надмірним. Рекомендується використовувати події або окремий метод для оновлення тексту при зміні мови.

Висновки до другого розділу

У другому розділі роботи було детально описано процес проектування гри. На початку ми визначили варіанти використання гри та об'єктно-орієнтовану структуру, що дозволило нам зрозуміти, як саме буде працювати наше програмне забезпечення. Ми ретельно проаналізували потреби користувачів та вимоги до системи, що допомогло нам створити план розробки гри, який враховує всі необхідні функції та можливості.

Проектування та реалізація алгоритмів роботи гри також відіграє ключову роль. Ми розробили чіткі сценарії взаємодії користувача з грою, від завантаження головного меню до вибору рівнів та налаштувань якості зображення. Це забезпечує зручність та гнучкість у використанні гри на різних пристроях.

Таким чином, цей розділ закладає фундамент для розробки гри, забезпечуючи її функціональність, ефективність та зручність для користувачів.

Тестування проводиться для виявлення та усунення можливих помилок, забезпечуючи надійність та стабільність роботи гри. Ми також запланували проведення стрес-тестів для оцінки продуктивності гри під час високих навантажень.

Тестування та перевірка програми на завершальному етапі гарантують, що данні рішення відповідають усім вимогам та забезпечують надійність та стабільність роботи гри.

РОЗДІЛ 3. інтерфейс та порядок використання гри в жанрі “зомбі шутер”

3.1. Опис структури ігрового рівня

Структура рівня є одним із найважливіших аспектів при розробці зомбі шутера на Unity, оскільки від неї залежить загальний геймплей, взаємодія гравця з оточенням та рівень занурення в гру. В цьому розділі розглянемо основні компоненти структури рівня, які необхідно враховувати при розробці, та їх взаємодію.

Наш персонаж з’являється на рівні, основною його задачею та метою є знешкодити всіх ворогів аби мати можливість відкрити наступний рівень. Але пройти перший рівень не буде так легко так як для його завершення потрібно знешкодити першу хвилю ворогів в кількості 25 зомбі, після чого буде мінібос якого потрібно знешкодити аби з’явилась наступна хвиля ворогів, яких знову потрібно знешкодити в кількості 25 аби з’явився фінальний бос в якого вдвічі більше здоров’я, а ніж в попереднього мінібоса. Після його знешкодження, гравцеві буде виведено інформацію щодо завершення рівня.

Після чого гравцеві буде надано доступ до наступного рівня де задачею буде знешкодити 10 ворогів, але це вже буде важче адже незважаючи що вони будуть повільніші вони будуть завдавати вдвічі більше шкоди, після чого з’явиться 2 мінібоса, після їх знешкодження гравцеві так як і в попередньому рівні буде виведено інформацію про його завершення, та надано доступ до наступного рівня.

Також для більш комфортного орієнтуванню по рівню є присутня невелика мінікарта в правому верхньому куті екрана.

Основні компоненти структури рівня

1. Локація:

- Будівлі та споруди: Включають різні об’єкти, такі як будинки, магазини, гаражі та інші споруди, де гравець може ховатися або шукати ресурси.
- Природні об’єкти: Ліси, річки, гори та інші природні елементи, що додають реалізму і складності рівню.

2. Перешкоди та укриття:

- Перешкоди: Стіни, барикади, транспортні засоби та інші об'єкти, що обмежують пересування гравця та зомбі.

- Укриття: Об'єкти, за якими гравець може ховатися від зомбі та вести продумувати план проходження.

3. Спавн:

- Спавн гравця: Місце, де гравець починає рівень.

- Спавн зомбі: Місця, звідки з'являються зомбі під час гри. Важливо розмістити ці точки так, щоб забезпечити постійний виклик для гравця.

Вороги :

- Зомбі: Різні типи зомбі з різними характеристиками (швидкі, повільні, сильні, слабкі).

3.2 Структура інтерфейсу гри

Так як в данному проєкті головне меню є важливим елементом інтерфейсу який надає можливість почати проходити гру. У головному меню часто є розділ налаштувань, де гравці можуть змінювати різні параметри гри, такі як графічні налаштування, мову та інші параметри. Це дозволяє гравцям персоналізувати свій ігровий досвід. Також в головне меню є розділ зі статистикою гравця, досягненнями, та іншими даними, які можуть бути цікавими для гравців. Меню зазвичай містить опцію для виходу з гри, що дозволяє гравцям безпечно завершити ігровий сеанс. Відповідно головне меню є першою взаємодією гравця з грою, і воно встановлює тон та атмосферу гри. Хороший дизайн головного меню може створити позитивне перше враження. Відповідно повинно бути інтуїтивно зрозумілим і легко доступним меню для того щоб було легкодоступним та зрозумілим для гравця.

зазвичай містить опцію для виходу з гри, що дозволяє гравцям безпечно завершити ігровий сеанс. Відповідно головне меню є першою взаємодією гравця з грою, і воно встановлює тон та атмосферу гри. Хороший дизайн головного меню може створити позитивне перше враження. Відповідно повинно бути інтуїтивно зрозумілим і легко доступним меню для того щоб було легкодоступним та зрозумілим для гравця.

Для перегляду розробленого меню слід перейти за (рис. 3.1).

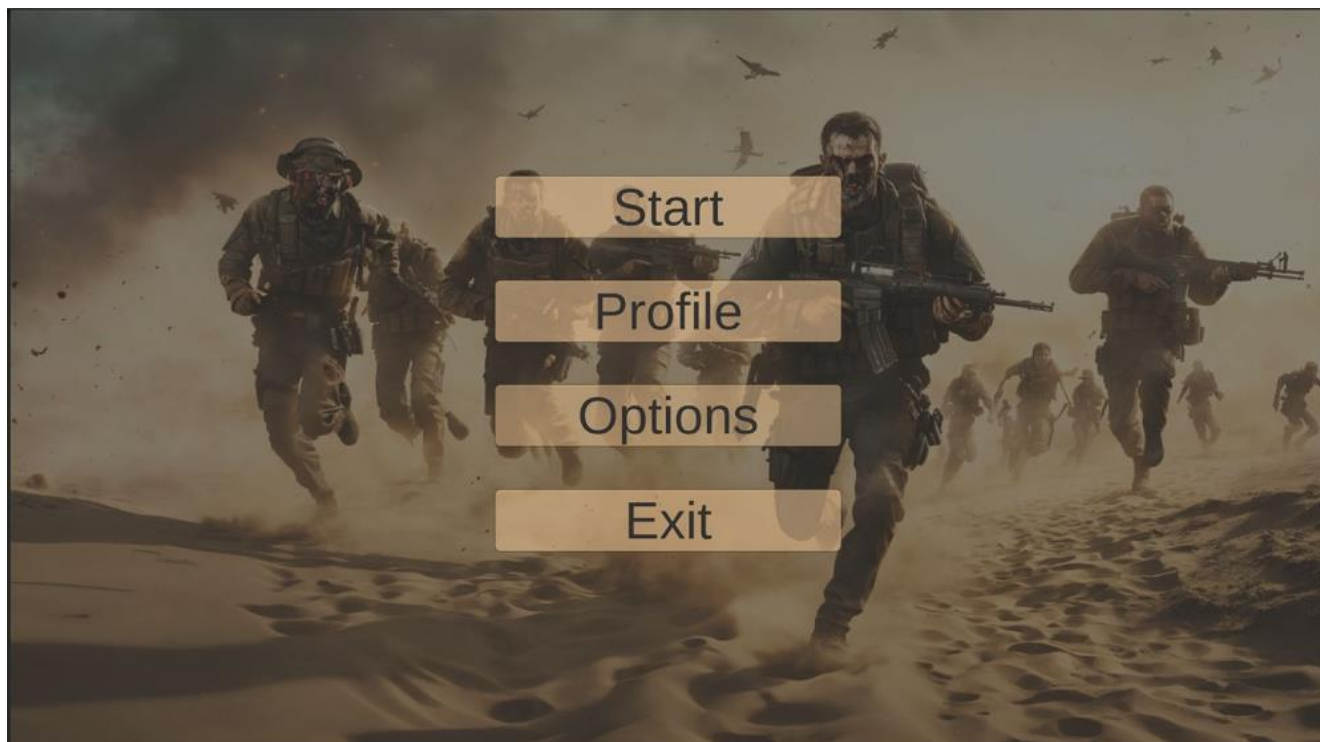


Рис. 3.1 – Головне меню гри

Вигляд меню графіки було розроблено для комфортного налаштування графіки для апаратного забезпечення гравця відповідно до можливостей. Такі налаштування зазвичай поділяються на категорії: низькі, середні, високі та ультра, для перегляду даного меню слід перейти до (рис. 3.2 та рис. 3.3).

Ось детальний опис кожної категорії.

Низькі налаштування графіки:

Основні характеристики:

- Текстури: Низька роздільна здатність текстур, що може робити об'єкти менш деталізованими та розмитими.
- Тіні: Прості або відсутні тіні, що зменшує навантаження на GPU.
- Антиаліасинг: Зазвичай вимкнено або налаштовано на найнижчий рівень, що може призводити до "зазубрених" країв об'єктів.
- Ефекти постобробки: Відсутні або мінімальні, наприклад, відсутність ефектів розмиття руху або глибини різкості.
- Дистанція промальовування: Обмежена, через що віддалені об'єкти можуть з'являтися раптово, коли гравець наближається до них.

Середні налаштування графіки.

Основні характеристики:

- Текстури: Середня роздільна здатність текстур, що покращує деталізацію без значного навантаження на систему.
- Тіні: Відтворення базових тіней з середньою деталізацією.
- Антиаліасинг: Низький або середній рівень антиаліасингу, що згладжує краї об'єктів, зменшуючи ефект "зазубрення".
- Ефекти постобробки: Мінімальні або помірні ефекти, такі як легке розмиття руху та основна глибина різкості.
- Дистанція промальовування: Середня, що забезпечує більш плавний вигляд віддалених об'єктів.

Високі налаштування графіки.

Основні характеристики:

- Текстури: Висока роздільна здатність текстур, що забезпечує високу деталізацію та чіткість об'єктів.
- Тіні: Високоякісні тіні з високою деталізацією та точністю.
- Антиаліасинг: Середній або високий рівень антиаліасингу, що значно згладжує краї об'єктів.
- Ефекти постобробки: Повний набір ефектів, включаючи розмиття руху, глибину різкості, об'ємне освітлення та інші візуальні покращення.
- Дистанція промальовування: Висока, що дозволяє бачити об'єкти далеко на горизонті без появи.

Ультра налаштування графіки.

Основні характеристики:

- Текстури: Найвища роздільна здатність текстур, що забезпечує максимальну деталізацію та чіткість.
- Тіні: Тіні з максимальною деталізацією, включаючи м'які тіні та додаткові ефекти освітлення.
- Антиаліасинг: Максимальний рівень антиаліасингу, що забезпечує найгладкіші краї об'єктів.

- Ефекти постобробки: Усі можливі ефекти, включаючи розширені версії розмиття руху, глибини різкості, об'ємного освітлення, реалістичних відблисків, відбиттів та інших візуальних покращень.

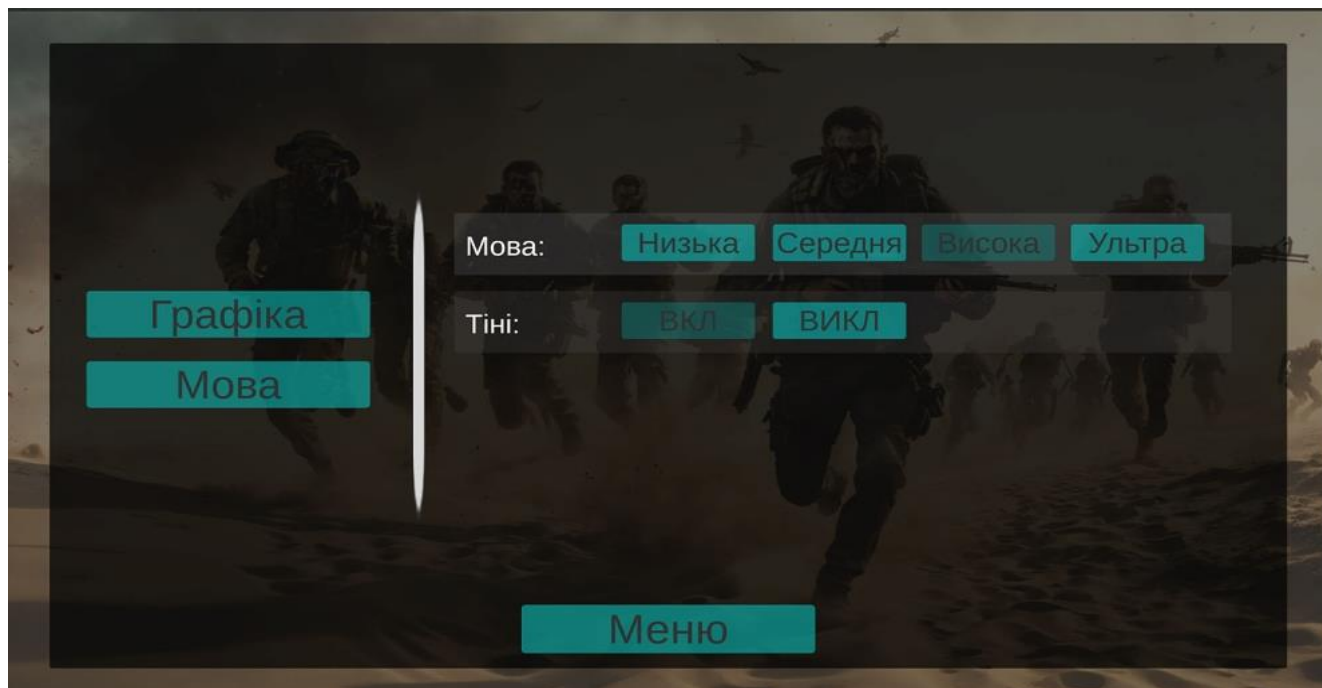


Рис. 3.2 – Вигляд меню графіки

Варіант меню графіки використовуючи англійську локалізацію

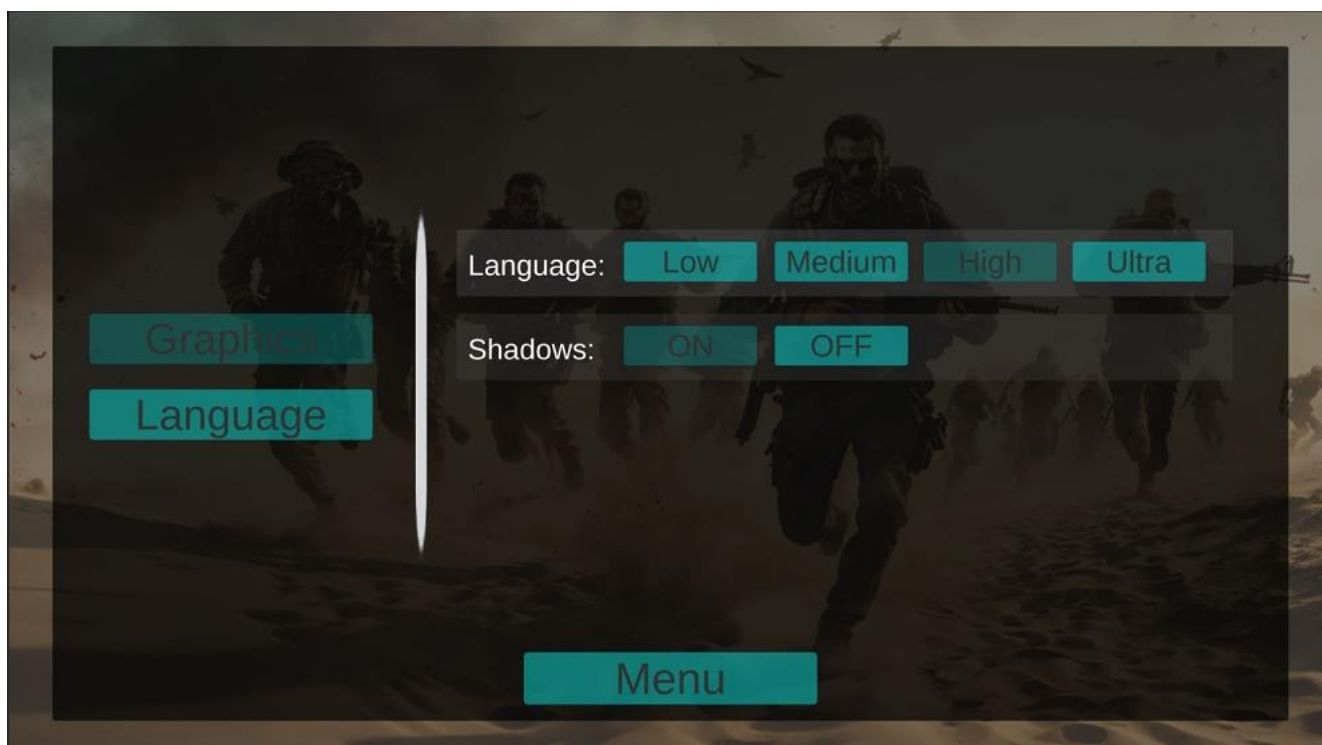


Рис. 3.3 – Меню графіки з використанням англійської локалізації

Локалізація гри – це процес адаптації відеогри до різних мов та культурних

контекстів. Це важливий аспект розробки, оскільки допомагає зробити гру доступною для ширшої аудиторії.

Українська локалізація:

- Забезпечує можливість грати в гру рідною мовою, що підвищує комфорт та задоволення від гри.
- Підвищує розуміння сюжету, діалогів, інструкцій та інших елементів гри.
- Допомагає молодим гравцям вивчати рідну мову, розвиваючи їх мовні навички через інтерактивні медіа.
- Відображає культурні особливості та специфіку, що робить гру більш привабливою для місцевих гравців.

Для перегляду української локалізації слід перейти до (рис. 3.4).

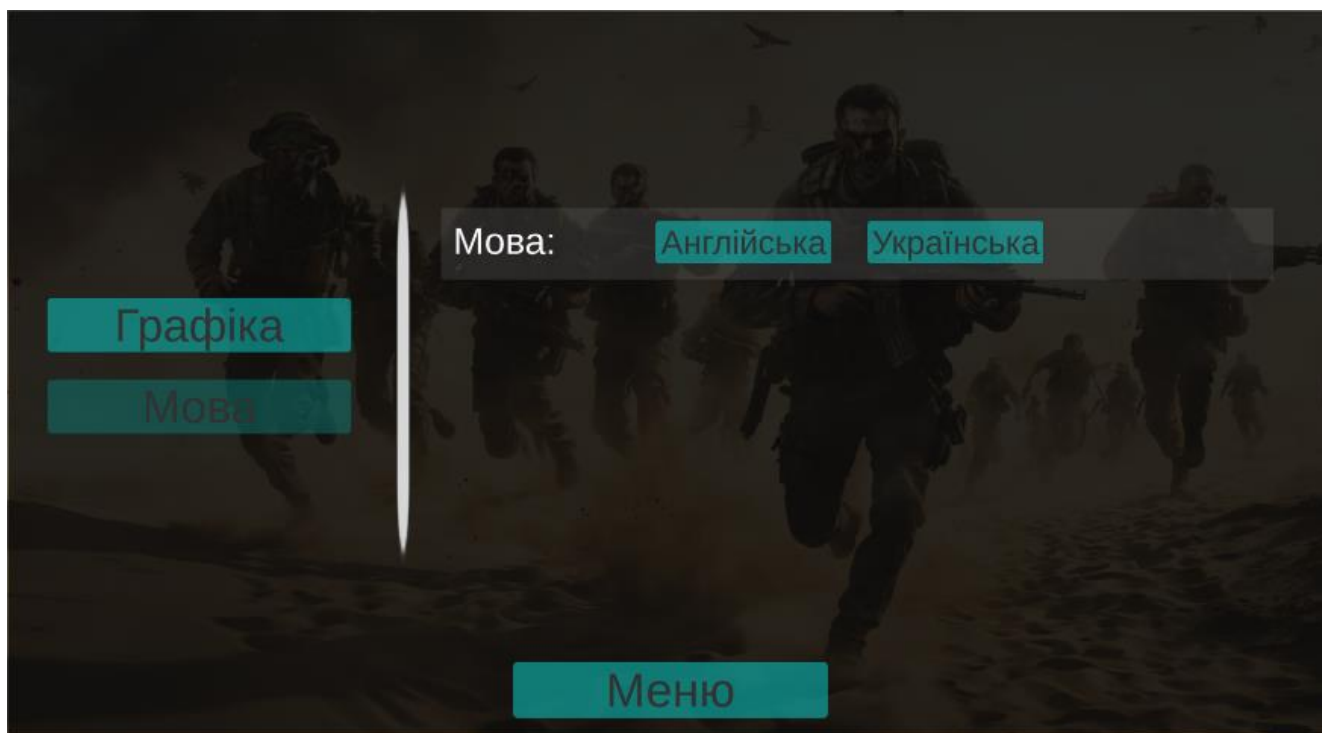


Рис. 3.4 - Вибрана мова локалізації українська

Англійська локалізація:

- Англійська мова є міжнародною мовою, яку розуміють у багатьох країнах світу.
- Збільшує доступність гри для міжнародної аудиторії.

- Полегшує спілкування між гравцями з різних країн, сприяючи створенню глобальних ігрових співтовариств.
- Робить гру доступною для англомовних оглядачів, критиків та стрімерів, що підвищує її видимість та популярність.
- Сприяє вивченню англійської мови, особливо серед молоді, яка часто грає в ігри як на рідній, так і на іноземній мовах.
- Локалізація дозволяє враховувати культурні особливості та специфіку, що робить гру більш привабливою та прийнятною для місцевих гравців.

Українська та англійська локалізація є важливими для розширення аудиторії гри та покращення користувацького досвіду. Вони забезпечують доступність гри для різних груп гравців, підтримують розвиток національних мов і культур, а також сприяють глобальному поширенню гри. Для перегляду англійської локалізації слід перейти до (рис. 3.5).

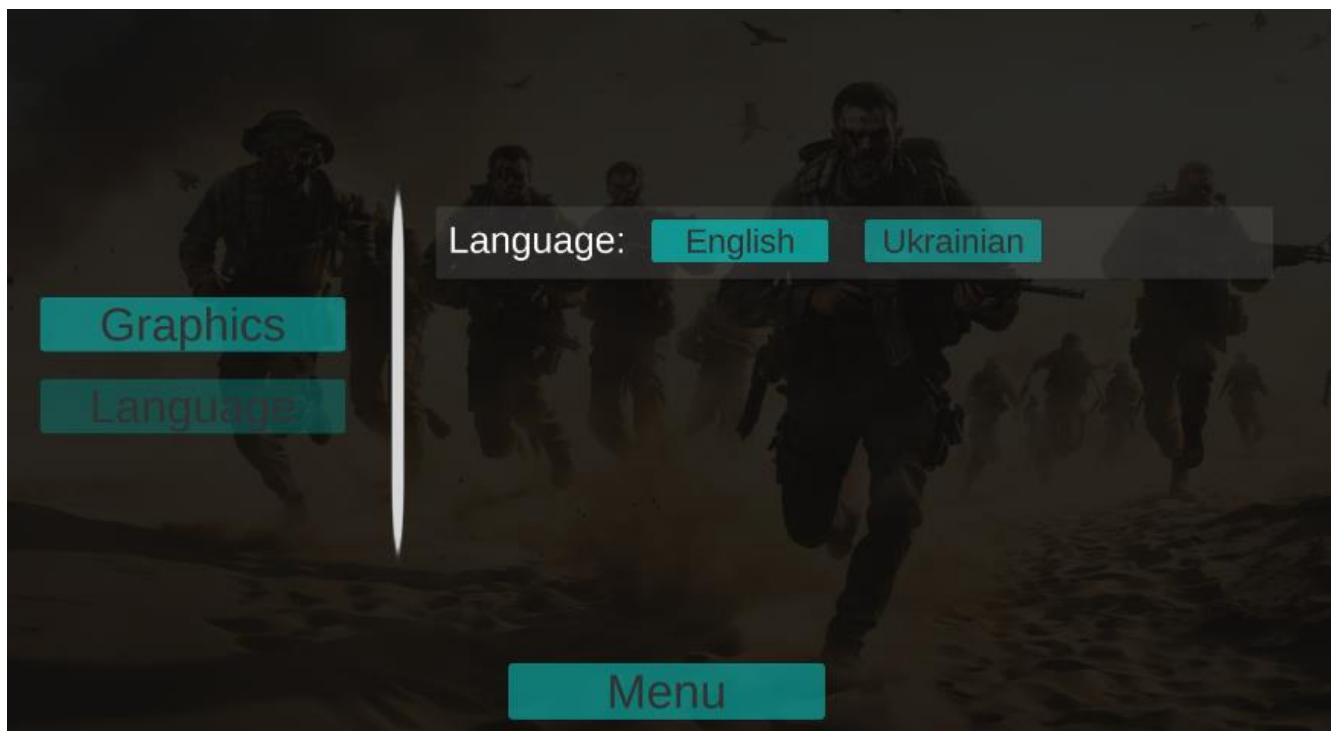


Рис. 3.5 – Вигляд англійської локалізації

Створення локації:

- Локація допомагає створити атмосферу гри та забезпечують гравцю відчуття занурення в ігровий світ. Правильно створена локація може перенести

гравця у фантастичний світ постапокаліптичного міста. Вона задає тон і настрій гри, впливаючи на емоції гравця.

- Кожна локація може запропонувати унікальні геймплейні можливості та механіки. Наприклад, міські локації можуть бути наповнені NPC, з якими можна взаємодіяти, тоді як лісові або пустельні області можуть містити більше природних перешкод і ворогів. Локації створюють різноманітність у геймплеї, роблячи його більш цікавим і захоплюючим.

- Красиві та детально опрацьовані локації приносять естетичне задоволення. Вони можуть бути художніми шедеврами, які приємно розглядати. Естетика локацій також допомагає створити унікальний візуальний стиль гри, роблячи її впізнаваною та запам'ятовуваною.

- Локації у відеоіграх виконують багато важливих функцій, від створення атмосфери та просування сюжету до забезпечення геймплейних можливостей та соціальних взаємодій. Вони допомагають гравцям зануритися в ігровий світ, роблячи його більш реалістичним, захоплюючим і різноманітним.

Для перегляду реалізованої локації слід переглянути (рис. 3.6 та рис. 3.7).



Рис. 3.6 – Реалізована локація

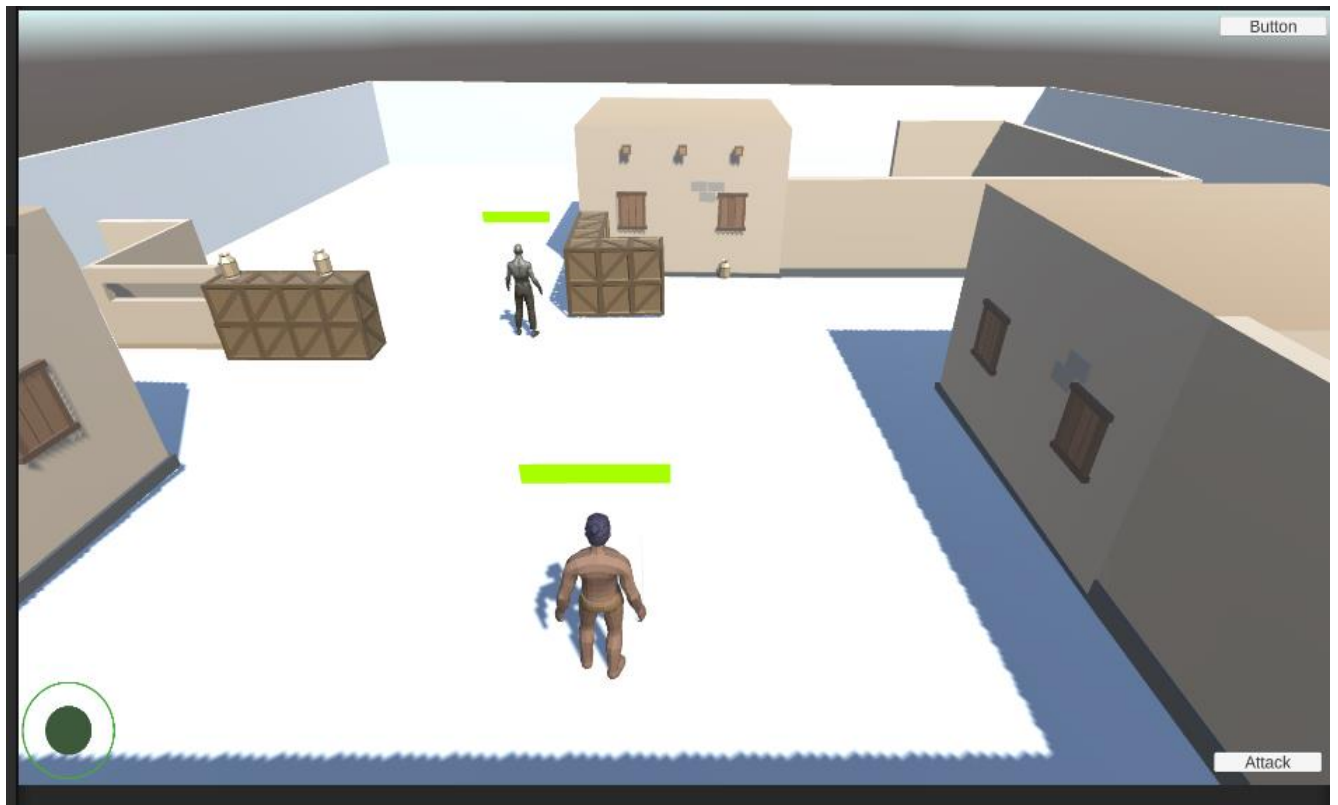


Рис. 3.7- Ілюстрація іншої частини локації

3.3 Тестування продуктивності гри

Unity Profiler - це вбудований інструмент, який дозволяє розробникам відстежувати різні аспекти продуктивності гри в реальному часі.

Етапи тестування продуктивності:

1. Підготовка:

- Визначення апаратних конфігурацій та платформ, на яких буде проводитись тестування (наприклад, різні моделі ПК, консолей, мобільних пристроїв).
- Налаштування середовища тестування, включаючи встановлення необхідного програмного забезпечення та інструментів для моніторингу продуктивності.

2. Тестування:

- FPS (кадри в секунду):
- Вимірювання кількості кадрів, які гра може відобразити за секунду. Це один із головних показників плавності геймплею.
- Завантаження ресурсів:

- Перевірка швидкості завантаження текстур, моделей, звуків та інших ресурсів гри.
- Моніторинг використання ресурсів:
- Відстеження використання процесора , графічного процесора , оперативної пам'яті та інших ресурсів системи під час гри.
- Тестування різних налаштувань графіки:
- Перевірка роботи гри на різних рівнях налаштувань графіки (низькі, середні, високі, ультра), щоб оцінити їх вплив на продуктивність.
- Навантажувальні тести:
- Проведення тестів під високим навантаженням, наприклад, у сценах з великою кількістю об'єктів або інтенсивною графікою, щоб оцінити, як гра витримує стресові умови.

3. Аналіз результатів:

- Збір та аналіз даних, отриманих під час тестування, щоб визначити слабкі місця і можливі проблеми.
- Визначення областей, які потребують оптимізації або виправлення.

Для перегляду процесу тестування вбудованим інструмент для моніторингу продуктивності ігор, створених на Unity. Дозволяє відстежувати використання CPU, GPU, пам'яті та інші важливі метрики. (рис. 3.8 та рис. 3.9). Використання цих методів допоможе виявити та усунути проблеми з продуктивністю, що забезпечить плавну та стабільну роботу данної гри.



Рис. 3.8 – Процес тестування гри

Нижче наведено тестування продуктивності при активному режимі навантаження



Рис. 3.9 – Ілюстровано тестування при навантаженнях

Висновки до третього розділу

У третьому розділі роботи було докладно описано структуру рівня, включаючи його реалізацію та вимоги. Було визначено основні елементи рівня, такі як карти, локації, перешкоди, укриття, точки спавну, ресурси, предмети, вороги та NPC. Також було розглянуто взаємодію цих компонентів, що дозволяє створити захопливий ігровий процес, який підтримує високий рівень зацікавленості гравців. Особлива увага приділялась навігації та інтерактивності, що забезпечують гравцю можливість легко орієнтуватися в просторі та взаємодіяти з оточенням.

Було розглянуто інтерфейс гри, який включає в себе реалізацію меню, графіку та локалізацію. Детально описано, як створити інтуїтивно зрозуміле меню, що дозволяє гравцям легко налаштовувати параметри гри, обирати рівні та керувати своїм профілем. Графічний дизайн був описаний як важливий аспект, який впливає на загальне враження від гри, включаючи стилістику, кольорову гаму та візуальні ефекти. Локалізація ж забезпечує доступність гри для ширшої аудиторії, дозволяючи перекладати її на різні мови без втрати якості контенту.

Висвітлено процес тестування гри при активному та пасивному навантаженні. Для тестування використовувалися вбудовані інструменти Unity, що забезпечило точність та ефективність перевірок.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи бакалавра було спроектовано та розроблено гру в жанрі екшен, проведено аналіз предметної галузі та порівняно з існуючими аналогами програмного продукту. На основі аналізу було визначено функціональні вимоги до гри.

Під час проектування визначено движок який буде використовуватись для розробки гри - це Unity. Також для написання скриптів використовувалась Visual studio code. Відповідно було здійснено опис механіки гри та супутнього функціоналу.

Під час реалізації гри було розроблено механіку руху персонажу, аптечки, мінікарти, спавну, функціонал камери, за допомогою якої відбувається показ локації, реалізація хелсбару, локалізації та меню.

Гру було протестовано на вбудованому інструментарії, який зміг показати навантаження на систему та продуктивність проекту також було протестовано швидкість завантаження гри.

Отже, розроблена гра відповідає поставленим вимогам та досягнутим цілям. Гра швидко запускається та реалізовую всі пропрацьовані аспекти на вищому рівні.

Результати тестування підтвердили ефективність та надійність розробленого функціоналу гри. Програмний продукт готовий до подальшого використання та відповідає до потреб користувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Learning Pathways [Електронний ресурс] – Режим доступу до ресурсу: <https://learn.unity.com/u/66412701edbc2a246b105422?tab=activity>.
2. low poly assets [Електронний ресурс] – Режим доступу до ресурсу: <https://assetstore.unity.com/>.
3. Unity Learn. \"Physics in Unity\" [Електронний ресурс] – Режим доступу до ресурсу: <https://learn.unity.com/course/physics>.
4. Microsoft. \"C# Fundamentals for Absolute Beginners\" [Електронний ресурс] – Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/shows/c-fundamentals-for-absolute-beginners/>.
5. Ткаченко Т. Просунуті техніки рендерингу на С# у Unity / Т. Ткаченко. // Журнал розробки ігор. – 2022. – №1. – С. 50–64.
6. Сидоренко Г. Створення та використання користувацьких компонентів у Unity на С# / Г. Сидоренко. // Міжнародний журнал розробки ігор. – 2018. – №4. – С. 77–89.
7. Ковальчук О. Використання паралелізму в Unity для підвищення продуктивності ігор / О. Ковальчук. // Журнал комп'ютерних наук і технологій. – 2020. – №5. – С. 90–102.
8. Петров А. Оцінка продуктивності коду на С# у Unity / А. Петров. // Журнал розробки ігор та досліджень. – 2019. – №3. – С. 45–56.
9. Animation in Unity [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unity3d.com/Manual/AnimationSection.html>.
10. User interface (UI) [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unity3d.com/Manual/UIToolkits.html>.
11. Built-in 3D Physics [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unity3d.com/Manual/PhysicsOverview.html>.
12. Animator Controllers [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unity3d.com/Manual/AnimatorControllers.html>.
13. Playables API [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.unity3d.com/Manual/Playables.html>.

14. Unity architecture [Электронный ресурс] – Режим доступа до ресурсу: <https://docs.unity3d.com/Manual/unity-architecture.html>.

15. Important Classes [Электронный ресурс] – Режим доступа до ресурсу: <https://docs.unity3d.com/Manual/ScriptingImportantClasses.html>.

ДОДАТКИ

					ПЗ.КР.Б – 121 – 24 - ПЗ	Арк.
						54

```

Player.cs

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

public class Player : MonoBehaviour, IDamageable
{
    [Header("MainSettings")]
    public float moveSpeed;
    public Joystick joy;
    private Rigidbody rb;
    Animator animatorObject;
    private Vector3 moveVector;
    private Coroutine stopAttakCoroutine;
    public Animator animator;

    [Header("AttakSettings")]
    [SerializeField] private float stopAttak;

    [Header("Events")]
    public UnityEvent offScript;
    public UnityEvent Sound;
    public UnityEvent actTrigger, deactTrigger;

    [Header("Health")]
    private int healthCount;
    public static int maxHealth;

```

```

private int health;

public static int firstAid;

[Header("HealthBar")]
[SerializeField] private HelthBar healthbar;

[Header("First Aid Object")]
public GameObject FirstAidObject;

[Header("GameObject")]
public GameObject button;

public static int AidUsed;

public static bool Dead = false;

#region programs
void Start()
{
    PlayerPrefs.GetInt("UpgradeLvlTest4", UpgradeHp.saveUpgradeLvl);
    rb = GetComponent<Rigidbody>();
    rb.freezeRotation = true;
    animatorObject = GetComponentInChildren<Animator>();
    healthCount = UpgradeHp.maxhealth;
    maxHealth = healthCount;
    Health = maxHealth;
    Debug.Log(health.ToString());
}

void FixedUpdate()

```

```

{
    MovePlayer();
}

#endregion

#region PlayerMove
private void MovePlayer()
{
    if(Dead == false)
    {
        float horizontalInput = joy.Horizontal;
        float verticalInput = joy.Vertical;
        Vector3 dir = new Vector3(horizontalInput, 0, verticalInput).normalized;

        // Використовуйте Rigidbody.MovePosition() для плавного руху з
        фіксованою частотою
        rb.MovePosition(transform.position + dir * moveSpeed *
Time.fixedDeltaTime);

        if(dir != Vector3.zero)
        {
            Quaternion toRotation = Quaternion.LookRotation(dir, Vector3.up);
            transform.rotation = Quaternion.RotateTowards(transform.rotation,
toRotation, 750* Time.fixedDeltaTime);
        }

        moveVector.x = joy.Horizontal;
        moveVector.z = joy.Vertical;
    }
}

```

```

float joystickYPosition = joy.Vertical;
float joystickXPosition = joy.Horizontal;

if (joystickYPosition != 0 || joystickXPosition != 0)
{
    animatorObject.SetBool("Speed", true);
}
else
{
    animatorObject.SetBool("Speed", false);
}
}
else{

}
}
#endregion

#region health
public int Health
{
    get { return health; }
    set
    {
        health = value;

        if(health <= 0)
        {
            health = 0;
            Die();
        }
    }
}

```

```

        }
        healthbar.UpdateHealthbar(maxHealth, health);
    }
}

public int firstAidHelth
{
    get {return firstAid; }
    set {firstAid = value;}
}

public virtual void TakeDamage(int damageValue)
{
    Health -= damageValue;
}

#endregion

#region buttons
public void OnButtonClickAttack()
{
    Sound.Invoke();

    animatorObject.SetBool("Attak", true);

    if (stopAttakCoroutine != null)
    {
        StopCoroutine(stopAttakCoroutine);
    }

    StartCoroutine(StartAttackAnimation());
}

```

```

    }

    public void OnButtonClickAid()
    {
        if(health <= Health && firstAid <= 50 && firstAid > 0)
        {
            health += firstAid;
            firstAid -= firstAid;
            Destroy(FirstAidObject);
            button.SetActive(false);
            AidUsed = 1;
        }
        else if(health <= Health && firstAid > 50)
        {
            health += 50;
            firstAid -= 50;
            Destroy(FirstAidObject);
            button.SetActive(false);
            AidUsed = 1;
        }
        healthbar.UpdateHealthbar(maxHealth, health);

    }

    #endregion

    #region coroutines
    IEnumerator StartAttackAnimation()
    {
        yield return new WaitForSeconds(0.1f);
    }

```

```

    animatorObject.SetBool("Attak", true);

    if (stopAttakCoroutine != null)
    {
        StopCoroutine(stopAttakCoroutine);
    }

    if(actTrigger != null)
    {
        actTrigger.Invoke();
    }
    stopAttakCoroutine = StartCoroutine(stopAt());
}

IEnumerator stopAt()
{
    while (true)
    {
        yield return new WaitForSeconds(stopAttak);

        animatorObject.SetBool("Attak", false);
        stopAttakCoroutine = null;

        if(deactTrigger != null)
        {
            deactTrigger.Invoke();
        }
    }
}

```

```

#endregion

#region Die
public virtual void Die()
{
    Dead = true;
    Enemy.playerChasing = false;
    animator.SetBool("Attak", false);
    animatorObject.SetBool("Death", true);

    if(offScript != null)
    {
        offScript.Invoke();
    }
}
#endregion
}

Enemy.cs
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;
using UnityEngine.Events;

public class Enemy : MonoBehaviour, IDamageable
{
    [Header("MainSettings")]
    public static bool playerChasing;
    private Transform playerTransform;
    private NavMeshAgent navMeshAgent;

```

```
public float detectionRange = 10f;
```

```
[Header("Events")]
```

```
public UnityEvent Destroy;
```

```
[Header("Animation")]
```

```
Animator animator;
```

```
[Header("Health")]
```

```
[SerializeField] private int maxHealth;
```

```
private int health;
```

```
[Header("HealthBar")]
```

```
[SerializeField] private HelthBar healthbar;
```

```
[Header("Money")]
```

```
[SerializeField] private int Money;
```

```
[Header("TypyOfZombie")]
```

```
public string TypeOfZombie;
```

```
public string classOfZombie;
```

```
private int notuseble;
```

```
#region program
```

```
void Start()
```

```
{
```

```
    /*KilledZombie.KilledZombieType_1 = PlayerPrefs.GetInt("Type_1");
```

```
    KilledZombie.KilledZombieType_2 = PlayerPrefs.GetInt("Type_2");
```

```
    KilledZombie.KilledZombieType_3 = PlayerPrefs.GetInt("Type_3");*/
```

```

        PlayerPrefs.GetInt("coinstest", MoneyCount._money);
        Health = maxHealth;
    }

    private void Awake()
    {
        navMeshAgent = GetComponent<NavMeshAgent>();
        playerTransform = FindObjectOfType<Player>().transform;
        animator = GetComponentInChildren<Animator>();
    }

    private void FixedUpdate()
    {
        float distanceToPlayer = Vector3.Distance(transform.position,
        playerTransform.transform.position);

        if(distanceToPlayer <= detectionRange)
        {
            playerChasing = true;
        }
        else
        {
            playerChasing = false;
        }

        if(playerChasing == true)
        {
            navMeshAgent.destination = playerTransform.position;
        }
    }

```

```

    }
#endregion

#region health
public int Health
{
    get { return health; }
    set
    {
        health = value;

        if(health <= 0)
        {
            health = 0;
            Die();
        }
        healthbar.UpdateHealthbar(maxHealth, health);
    }
}
#endregion

#region damage
public virtual void TakeDamage(int damageValue)
{
    Health -= damageValue;
}
#endregion

#region die
public virtual void Die()

```

```

{
    //ZombieSpawner.killedZombie += 1;

    MoneyCount._money += Money;
    PlayerPrefs.SetInt("coinstest", MoneyCount._money);

    playerChasing = false;
    animator.SetBool("Death", true);

    if(Destroy != null)
    {
        Destroy.Invoke();
    }

    /*if(TypeOfZombie == "Type_1")
    {
        KilledZombie.KilledZombieType_1 += 1;
        PlayerPrefs.SetInt("Type_1", KilledZombie.KilledZombieType_1);
    }
    else if(TypeOfZombie == "Type_2")
    {
        KilledZombie.KilledZombieType_2 += 1;
        PlayerPrefs.SetInt("Type_2", KilledZombie.KilledZombieType_2);
    }
    else if(TypeOfZombie == "Type_3")
    {
        KilledZombie.KilledZombieType_3 += 1;
        PlayerPrefs.SetInt("Type_3", KilledZombie.KilledZombieType_3);
    }*/
}

```

```

#endregion

#region triggers
void OnCollisionEnter(Collision collision)
{
    if(collision.gameObject.CompareTag("Player"))
    {
        playerChasing = false;
        animator.SetBool("Attak", true);
    }
}

void OnCollisionExit(Collision collision)
{
    if(collision.gameObject.CompareTag("Player"))
    {
        playerChasing = true;
        animator.SetBool("Attak", false);
    }
}
#endregion

```

```

public int firstAidHelth
{
    get {return notuseble;}

    set {}
}

```

Movements animation

```

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

public class MovementAnimator : MonoBehaviour
{
    NavMeshAgent navMeshAgent;
    Animator animator;

    // Start is called before the first frame update
    void Start()
    {
        navMeshAgent = GetComponent<NavMeshAgent>();
        animator = GetComponentInChildren<Animator>();
    }

    // Update is called once per frame
    void Update()
    {
        animator.SetFloat("Speed", navMeshAgent.velocity.magnitude);
    }
}

Gun damage

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

public class GunDamage : MonoBehaviour

```

```

{
    [Header("Damage")]
    public int damageG;

    #region program
    private void OnTriggerEnter(Collider other)
    {
        if(other.CompareTag("Zombie"))
        {
            Debug.Log("Zombie hite");
            other.GetComponent<IDamageable>().TakeDamage(damageG);
        }
    }
    #endregion
}

Playercamera.cs
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerCamera : MonoBehaviour
{
    Player player;
    Vector3 offset;

    // Start is called before the first frame update
    void Start()
    {
        player = FindObjectOfType<Player>();
        offset = transform.position - player.transform.position;
    }
}

```

```

    }

    // Update is called once per frame
    void LateUpdate()
    {
        transform.position = player.transform.position + offset;
    }
}

```

Healthbar.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using DG.Tweening;

public class HelthBar : MonoBehaviour
{

    private Slider _helthBarSlider;
    [SerializeField] private Slider damageEffectSlider;
    private Quaternion _startRotation;

    private void Awake()
    {
        _startRotation = transform.rotation;
        _helthBarSlider = GetComponent<Slider>();
    }

    private void LateUpdate()

```

```

        transform.rotation = _startRotation;
    }

    public void UpdateHealthbar(int maxHealth, int currentHealth)
    {
        _helthBarSlider.maxValue = maxHealth;
        _helthBarSlider.minValue = 0;
        _helthBarSlider.value = currentHealth;

        damageEffectSlider.maxValue = maxHealth;
        DamageEffectAnimation(currentHealth);
    }

    private void DamageEffectAnimation(int targetValue)
    {
        damageEffectSlider.DOKill();
        damageEffectSlider.DOValue(targetValue, 0.5f);
    }
}

```