

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 122 «Комп'ютерні науки»
(освітня програма «Комп'ютерна графіка та розробка ігор»)
на тему:
«Гра з елементами roguelite»

Виконав студент групи КН-20-1

БОМБЛЮК Максим Олегович

Керівник роботи:

КУЗЬМЕНКО Олександр Вікторович

Рецензет:

ГРАФ Марина Сергіївна

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

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

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

Здобувач вищої освіти: БОМБЛЮК Максим Олегович
Керівник роботи: КУЗЬМЕНКО Олександр Вікторович
Тема роботи: «Гра з елементами roguelite»
затверджена наказом закладу вищої освіти від «25» січня 2024 р., №22/с
Термін здачі закінченої роботи: «18» червня 2024 р.
Вихідні дані роботи (зазначається предмет і об'єкт дослідження): Предметом дослідження є "roguelite" в ігровій індустрії та його ключові концепції. Об'єктом дослідження є технічні та дизайнерські аспекти розробки гри з елементами roguelite.

Консультанти з випускної кваліфікаційної роботи із зазначенням розділів, що їх стосуються

Розділ	Консультант	Дата	
		Завдання видав	Завдання прийняв
1	Олександр КУЗЬМЕНКО	25.03.2024	11.04.2024
2	Олександр КУЗЬМЕНКО	12.04.2024	14.05.2024
3	Олександр КУЗЬМЕНКО	15.05.2024	22.05.2024

Календарний план

№ з/п	Назва етапів випускної кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Постановка задачі. Пошук, огляд та аналіз аналогічних розробок. Формулювання завдання. Опрацювання літературних джерел	01.03.2024 - 14.03.2024	Виконав
2	Проектування структури	15.03.2024 - 23.03.2024	Виконав
3	Написання програмного коду	24.03.2024 - 26.05.2024	Виконав
4	Тестування системи	27.05.2024 - 30.06.2024	Виконав
5	Оформлення пояснювальної записки	31.06.2024 - 4.06.2024	Виконав
6	Попередній захист	05.06.2024	Виконав

Здобувач вищої освіти Максим БОМБЛЮК
Керівник Олександр КУЗЬМЕНКО

РЕФЕРАТ

Випускна кваліфікаційна робота складається з програмного додатку гри та пояснювальної записки.

В роботі було сформульовано основні принципи та підходи до проектування гри з елементами roguelite. Проведено огляд і порівняння існуючих аналогів, виявлено основні відмінності. Здійснено проектування і розробку гри з елементами roguelite. На даний час, проєкт знаходиться в альфа версії і буде вдосконалюватися в майбутньому.

Пояснювальна записка до випускної кваліфікаційної роботи складається з вступу, трьох розділів, висновків, списку використаної літератури та додатку.

Текстова частина викладена на 67 сторінках друкованого тексту.

Пояснювальна записка має 8 сторінок додатків. Список використаних джерел містить 11 найменувань і займає 1 сторінку. В роботі наведено 56 рисунків та 2 таблиці. Загальний обсяг роботи – 77 сторінок.

Ключові слова: UNITY, ГРА, ROGUELITE, AUTO BATTLER, ROGUELIKE, ПЕРМАНЕНТНИЙ, 2D, 3D, ПІКСЕЛЬНИЙ, СПРАЙТ, ІНТЕРФЕЙС, ЗДІБНІСТЬ, СТАТ, ЮНІТ, ВОРОГ, ПІДЗЕМЕЛЛЯ, МЕХАНІКА, СТЕЙТ, СТЕЙТ МАШИНА, СИСТЕМА ПАРТІКЛІВ, РЕІГРАБЕЛЬНІСТЬ.

					КН.КР.Б – 122 – 24 – ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата	«Гра з елементами roguelite» Пояснювальна записка	Літ.	Арк.	Аркушів
Розроб.		М.О. БОМБЛЮК						
Керівник		О.В. КУЗЬМЕНКО					3	77
Рецензент		М.С. ГРАФ.				Житомирська політехніка, Група КН-20-1		
Н. контр.								
Зав. каф.		М.С. ГРАФ						

ABSTRACT

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

The work formulated the main principles and approaches to designing a game with roguelite elements. A review and comparison of existing analogues was carried out, the main differences were identified. Completed the design and development of a game with roguelite elements. Currently, the project is in the alpha version and will be improved in the future.

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

The text part is laid out on 68 pages of printed text.

The explanatory note has 8 pages of annexes. The list of used sources contains 11 names and occupies 1 pages. The work contains 56 figures and 2 tables. The total volume of work is 77 pages.

Keywords: UNITY, GAME, ROGUELITE, AUTO BATTLER, ROGUELIKE, PERMANENT, 2D, 3D, PIXEL, SPRITE, INTERFACE, ABILITY, STAT, UNIT, ENEMY, DUNGEON, MECHANIC, STATE, STATE MACHINE, PARTICLE SYSTEM, REPLAYABILITY.

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

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ.....	6
ВСТУП.....	7
РОЗДІЛ 1 АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ.....	8
1.1 Постановка задачі	8
1.2 Аналіз існуючого програмного забезпечення за тематикою випускної кваліфікаційної роботи	10
1.3 Концепт-документ	13
1.4 Обґрунтування вибору інструментальних засобів	15
Висновки до першого розділу	15
РОЗДІЛ 2 ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	16
2.1 Проектування гри.....	16
2.2 Розробка гри	19
2.3 Створення візуальних елементів	45
Висновки до другого розділу	54
РОЗДІЛ 3 ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО ТЕСТУВАННЯ.....	55
3.1 Порядок встановлення та налаштування системи.....	55
3.2 Інтерфейс та порядок роботи з програмним застосунком	57
3.3 Тестування роботи програмного забезпечення	63
Висновки до третього розділу	66
ВИСНОВКИ	67
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	68
ДОДАТКИ	69

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

WFC - Wave Function Collapse

RPG - Role-Playing Game

RTS - Real-Time Strategy

ЛКМ - Ліва Кнопка Миші

ПКМ - Права Кнопка Миші

Юніт (англ. unit) - базова одиниця війська; персонаж, що належить до команди гравця.

Стат (англ. stat) - це певна характеристика чи властивість персонажа, наприклад: сила, спритність чи інтелект.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						6
Змн.	Арк.	№ докум.	Підпис	Дата		

ВСТУП

Актуальність теми. У сучасному інформаційному суспільстві розвиток ігор та ігрової індустрії є невід'ємною частиною технологічного прогресу. Комп'ютерні, мобільні та консольні ігри є одним з найпоширеніших видів розваг в наші часи. В 2023 році прибуток ігрової індустрії в 2 рази перевищив кіноіндустрію та в 8 разів музичну індустрію. Це зумовлено технологічним розвитком, збільшенням кількості гравців та використанням нових моделей монетизації. Тому актуальність ігор досить висока.

Одним із цікавих напрямків у світі відеоігор є roguelite ігри, які поєднують в собі елементи класичних roguelike ігор з низкою новаторських рішень. Ці ігри відзначаються непередбачуваністю геймплею, де кожна нова спроба пройти гру стає унікальним викликом для гравця. Гра з елементами roguelite не лише забезпечує безліч різноманітних сценаріїв, але й створює унікальні умови виживання, де вірність стратегії та вміння адаптуватися до змін стають вирішальними факторами успіху.

В ході даної кваліфікаційної роботи буде проведено аналіз основних концепцій roguelite, технічних та дизайнерських аспектів створення гри з елементами roguelite. Результати дослідження слугуватимуть підставою для розробки гри, яка демонструватиме основні ідеї та концепції, розглянуті в ході аналізу.

Метою випускної кваліфікаційної роботи є розробка гри з елементами roguelite, вивчення та впровадження основних концепції roguelite.

Предметом дослідження є “roguelite” в ігровій індустрії та його ключові концепції.

Об'єктом дослідження є технічні та дизайнерські аспекти розробки гри з елементами roguelite.

В рамках теми кваліфікаційної роботи було опубліковано тези:

Бомблюк М.О., Кузьменко О.В. WAVE FUNCTION COLLAPSE В РОЗРОБЦІ ІГОР. VI Всеукраїнська науково-технічна конференція. Комп'ютерні технології: інновації, проблеми, рішення - 29-30 листопада 2023р. м. Житомир.

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

РОЗДІЛ 1 АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ

1.1 Постановка задачі

Розроблювана в ході кваліфікаційної роботи гра (надалі “гра”) повинна мати елементи roguelite. Для правильної постановки задачі спочатку розглянемо, що таке “roguelite” і які його визначні характеристики.

За походженням “roguelite” є спрощеною формою жанру “roguelike” [1]. Жанр “roguelike” (like з англ. подібний, подібний до Rogue) започаткувала гра Rogue 1980 року (див. рис. 1.1), визначними особливостями якої є випадкове, процедурне створення рівнів, покроковий ігровий процес, збирання спорядження чи предметів, плиткова (тайлова) або ASCII-графіка, покращення персонажу під час гри та перманентна смерть персонажа.

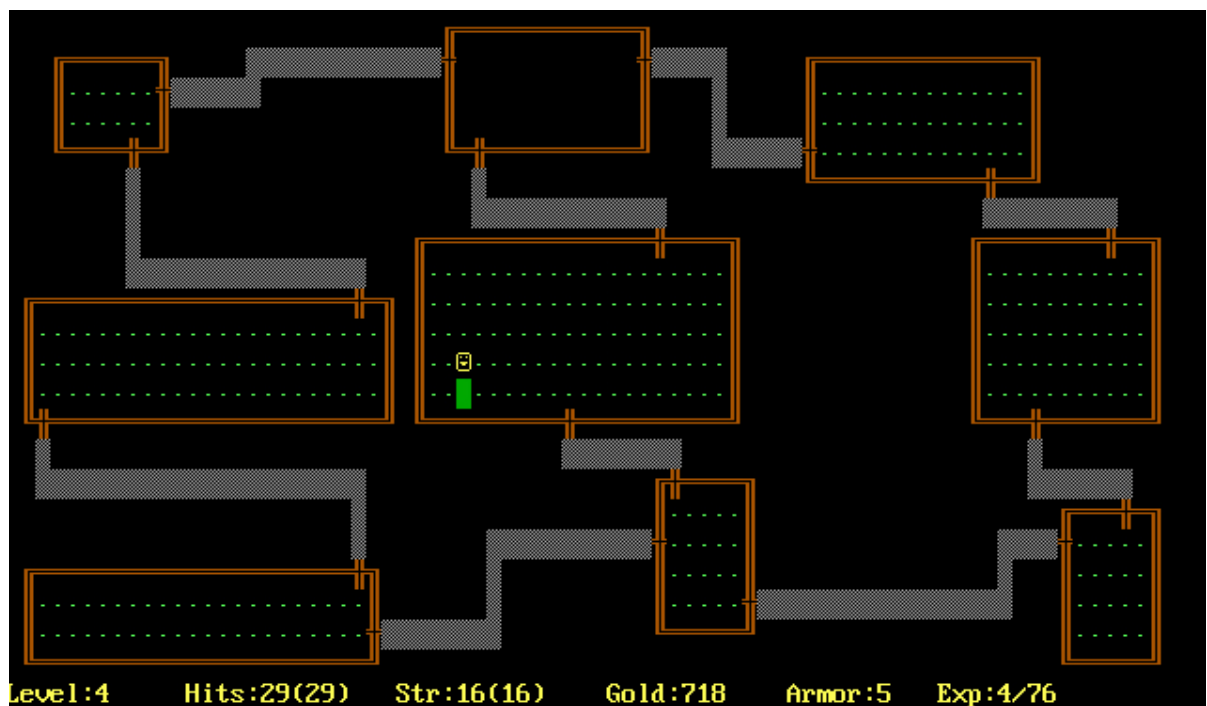


Рис. 1.1. Скріншот гри Rogue (1980).

“Roguelite” з’явився коли розробники почали брати певні механіки чи методи з roguelike ігор і застосовувати в різноманітних, не схожих на себе жанрах: рольових, карткових, стратегічних іграх, головоломках, симуляторах, шутерах чи платформерах. Тому “roguelite” - це не жанр, а скоріше набір методів, який є додатком до жанру гри.

Елементи roguelite, які будуть використані у грі:

1. Випадкова, процедурна генерація підземелля
2. Покращення персонажів під часи гри
3. Перманентна смерть (втрата усього прогресу у випадку поразки)

Основним жанром гри є піджанр стратегічних ігор - “auto battler” [2]. В іграх цього жанру гравець збирає собі команду з різних юнітів і відправляє їх битися проти таких самих команд. Юніти б’ються автоматично і гравець не може напряду втручатися в процес бою. Суть auto battler ігор полягає в збиранні збалансованої команди, стратегічному розміщенні юнітів, і комбінуванні їх здібностей для отримання переваги.



Рис. 1.2. Загальна ідея гри

Отже, потрібно розробити такий функціонал:

1. Стейт машина (State machine) для юнітів та ворогів та такі стейти:
 - 1.1 Рух
 - 1.2 Переслідування
 - 1.3 Атака
 - 1.4 Смерть
2. Можливість додавання здібностей чи особливостей юнітам та ворогам

3. Система підземель:

3.1 Процедурна генерація підземель за допомогою алгоритму Wave Function Collapse

3.2 Наповнення підземелля ворогами та нагородами

3.3 Переміщення по підземеллю

3.4 Система поверхів з наростаючою складністю

4. Зручний користувацький інтерфейс:

4.1 Карта підземелля

4.2 Інвентар для знайдених предметів

4.3 Панель інформації про юніта / ворога

4.4. Меню

Відповідно потрібно створити такі візуальні елементи:

1. 3D моделі для героїв та ворогів

2. 3D моделі для оточення

3. 2D спрайти для інтерфейсів

4. Візуальні ефекти

1.2 Аналіз існуючого програмного забезпечення за тематикою випускної кваліфікаційної роботи

Усі auto battler ігри в якомусь сенсі мають roguelite елементи, адже вони мають спільні характеристики: випадковість, перманентна смерть та висока реіграбельність (англ. replayability - наскільки варто перепроходити гру декілька разів). Auto battler ігри зазвичай мають в своїй основі магазин де можна придбати випадкових юнітів чи предмети за ігрову валюту. Велике різноманіття юнітів та можливість їх повторень у магазині майже гарантує, що гравець за одну гру не побачить усіх можливих юнітів. Це додає грі реіграбельності та постійно ставить перед гравцем вибір яких юнітів краще додавати до своєї команди.

Однак, жодна з наразі популярних auto battler ігор не реалізує фундаментальний елемент roguelite - дослідження процедурно згенерованих підземель. До того ж у сучасних auto battler іграх команда гравця обмежена в 5-10

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

юнітів, але в розроблюваній грі максимальна кількість дорівнює 25. Це зміщує фокус з індивідуальної сили кожного юніта на загальну композицію команди, що є досить незвичним для auto battler ігор.

Великим натхненням для гри є опціональна мінігра Fort Condor [3] з культової японської RPG 1998 року Final Fantasy VII. В цій мінігрі гравцю треба виставляти юнітів на карті для захисту форта Кондор, звідки і назва. Кожен ворог та юніт мають свої сильні та слабкі сторони і гравцю потрібно постійно адаптуватися та віддавати накази своїм юнітам задля перемоги. Fort Condor схожа на гру жанру “tower defense” де гравець може рухати свої вежі.



Рис. 1.3. Скріншот мінігри Fort Condor, Final Fantasy VII (1998).

Fort Condor дуже повільна та незручна в керуванні, тому початковою ідеєю для гри було відтворити Fort Condor з сучасним геймплеем та керуванням схожим до RTS ігор. Але після кількох прототипів та більш детального аналізу геймплею було визначено, що недоліки мінігри є невід’ємної її частиною. Швидке та ефективне керування юнітами дозволяло гравцю маніпулювати рухами монстрів і знищувати їх без будь-якого опору.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		11

В результаті вийшла нецікава RTS гра і через недостатність досвіду в RTS іграх було вирішено змінити геймплей на auto battler, при цьому зберігаючи дух Fort Condor. Гравець і досі відбиватиметься від поступових хвиль ворогів розставляючи юнітів на карті, але бої будуть повністю автоматичні та обмежені кімнатами підземелля.

Крім того, 3D моделі для гри будуть виконані схожим чином до Fort Condor: низькополігональні абстрактні моделі, що складаються переважно з пірамід та простих текстур. Це допоможе скоротити витрати часу на 3D моделювання та зберегти класичний стиль Fort Condor.



Рис. 1.4. Бої в Fort Condor.

Для дизайну auto battler елементів основним натхненням є гра Teamfight Tactics - одна з найпопулярніших auto battler ігор. З Teamfight Tactics буде взято систему підготовки до бою.

Перед боєм гравець має можливість розмістити своїх юнітів на дошці і це розміщення грає дуже важливу роль. Ті юніти які стоять попереду стануть першою ціллю ворогів, тому наперед потрібно ставити юнітів з високими

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

захисними характеристиками. Відповідно, юніти які мають високі атакуючі характеристики ставляться ззаду, подалі від небезпеки. Враховуючи це, а також унікальні здібності кожного юніта, гравець повинен підбирати правильне розташування та стратегію для перемоги.



Рис. 1.5. Скріншот гри Teamfight Tactics.

1.3 Концепт-документ

Назва: Condor Dungeon

Жанр: roguelite auto battler

Платформа: Windows

Кількість гравців: однокористувацька

Цільова аудиторія:

Гравці віком 16-30 років, які люблять інді ігри та жанри roguelite, auto battler чи RPG.

Основні особливості гри:

Унікальне поєднання auto battler боїв та дослідження процедурно згенерованих підземель.

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

Геймплей:

Гравцю доведеться досліджувати процедурно згенероване підземелля наповнене ворогами та винагородами. Битви з ворогами відбуваються автоматично, завдання гравця - зібрати збалансовану, синергічну команду та стратегічно розставити своїх юнітів на дошці перед боєм. Гравець також може використовувати предмети, знайдені в своїх пригодах, для отримання переваги в бою, наприклад: зілля зцілення, тимчасові покращення юнітів чи вибухівка, що завдає шкоди ворогам. За перемогу над ворогами гравець отримує золото, яке може витратити на покращення своїх юнітів.

Гравець буде натрапляти на клітки з ув'язненими юнітами, які можна відчинити використавши ключі. Ключі можна знайти досліджуючи підземелля або отримати як нагороду після бою. Кожен юніт та ворог матимуть унікальні здібності, які будуть спонукати гравця шукати комбінації для юнітів та контрзаходи для ворогів.

Як тільки гравець переможе усіх ворогів на поверсі, він матиме змогу спуститися на наступний поверх і гра закінчується перемогою коли гравець досягає останнього поверху підземелля. Вороги з кожним поверхом будуть ставати сильнішими та збільшувати свою численність, так само як гравець покращує та набирає собі в команду юнітів. Гра закінчується поразкою якщо всі юніти гравця гинуть в бою, при цьому втрачається весь набутий прогрес (перманентна смерть).

Сюжет:

Гра сфокусована більше на глибокий стратегічний геймплей, ніж на розповідання історії, тому сюжет гри досить простий, але він у будь-якому разі доповнює світ та робить гру більш живою.

Події гри розгортаються в класичному фентезійному королівстві, де чудовиська з'являються з древніх підземель та починають знищувати людські міста. Гравець керує групою воїнів, які спускаються в підземелля з метою звільнити ув'язнених товаришів. Чим глибше вони спускаються, тим сильнішими та потворнішими стають монстри.

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

1.4 Обґрунтування вибору інструментальних засобів

Для розробки гри було обрано ігровий рушій Unity та мову програмування C#, оскільки це найпопулярніші та найперспективніші методи власноруч розробити гру. Також буде використовуватися git та Github для контролю версій.

Гра буде виконана в 3D, але також буде включати багато 2D елементів інтерфейсу, такі як карта підземелля, інформація про юніта, тощо.

2D спрайти для інтерфейсу та покадрові анімації будуть виконані в середовищі Aseprite, адже це растровий редактор створений виключно для піксельної графіки.

Для елементів інтерфейсу буде створено власний піксельний шрифт за допомогою онлайн ресурсу fontstruct.com [4]. Це дозволить забезпечити однаковий стиль для інтерфейсу та тексту.

3D моделі та скелетні анімації будуть створюватися в 3D редакторі Blockbench, який є зручним для моделювання низькополігональних моделей та має вбудовану можливість малювати піксельні текстури відразу на моделі. Blockbench є значно простішим у використанні ніж інші 3D редактори, що пришвидшить роботу над моделями

Висновки до першого розділу

В ході виконання першого розділу було визначено основні задачі та вимоги до гри. Також був проведений детальний аналіз аналогічних ігор, їх відмінних та спільних елементів. Розроблений концепт документ встановив основні напрямки та рамки для майбутньої роботи. Було визначено оптимальні інструменти для розробки та створення візуалу гри, що посприяє ефективнішій розробці та якіснішому результату.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		15

РОЗДІЛ 2 ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Проектування гри

Розробка гри це досить складний процес, тому варто спочатку здійснити проектування використовуючи мову моделювання UML (Unified Modeling Language). UML діаграми допомагають візуалізувати складні системи, що полегшує їх розуміння та розробку.

Спочатку розглянемо як гравець може взаємодіяти з грою. Для цього зручно використати діаграму варіантів використання (англ. Use case diagram) (див. рис. 2.1).

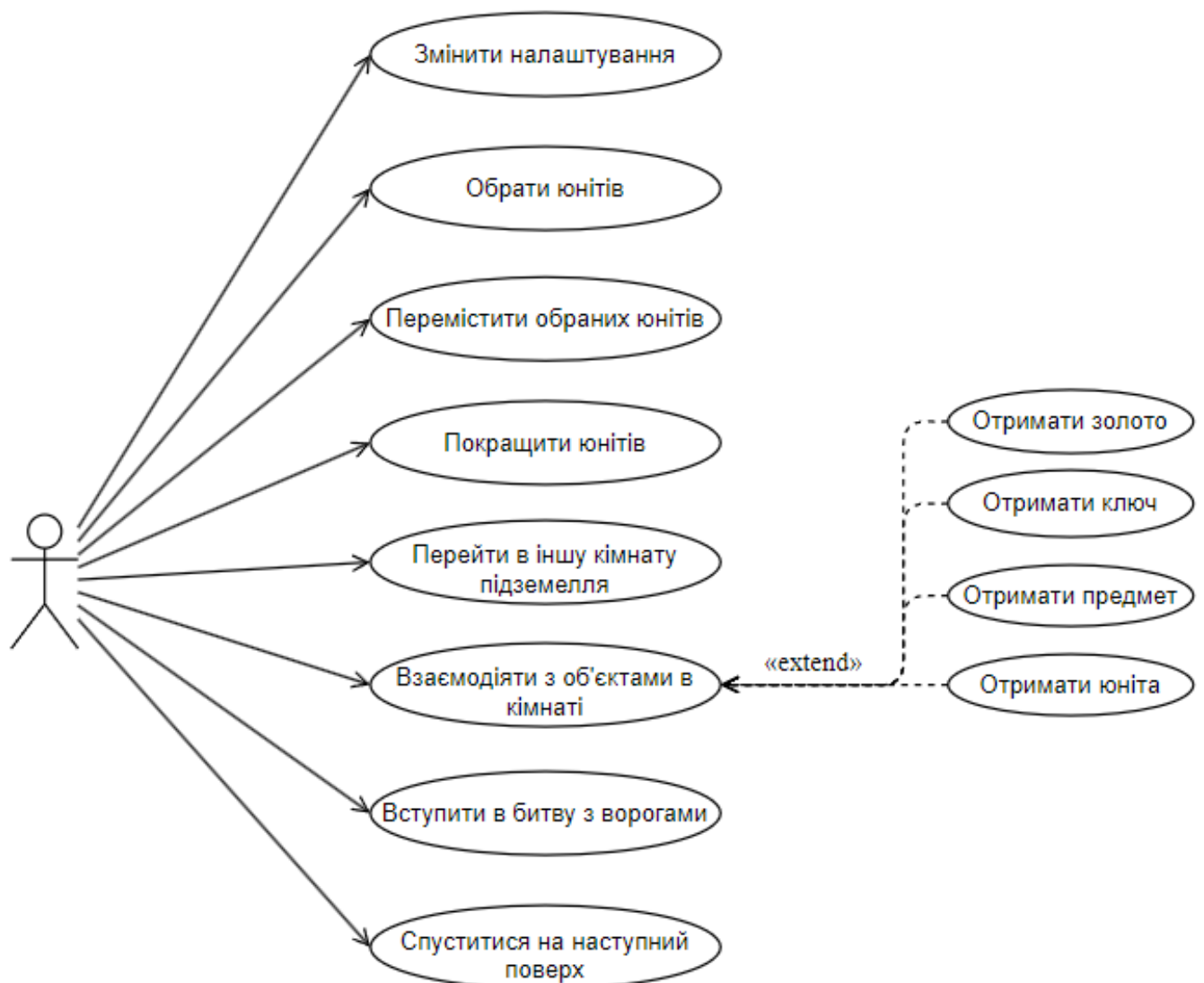


Рис. 2.1. Діаграма варіантів використання.

Варто зазначити, що деякі варіанти використання не завжди доступні. Наприклад, не завжди може бути об'єкт в кімнаті або не завжди є вороги, з якими можна вступити в битву. В будь-якому разі, моделювання цієї діаграми допомогло визначити вимоги до системи з точки зору користувача.

Щоб забезпечити ефективніший процес розробки варто детально розглянути ігровий цикл гри. Для цього використаємо діаграму активності (англ. activity diagram) та покроково опишемо усі можливі дії у грі (див. рис. 2.2).

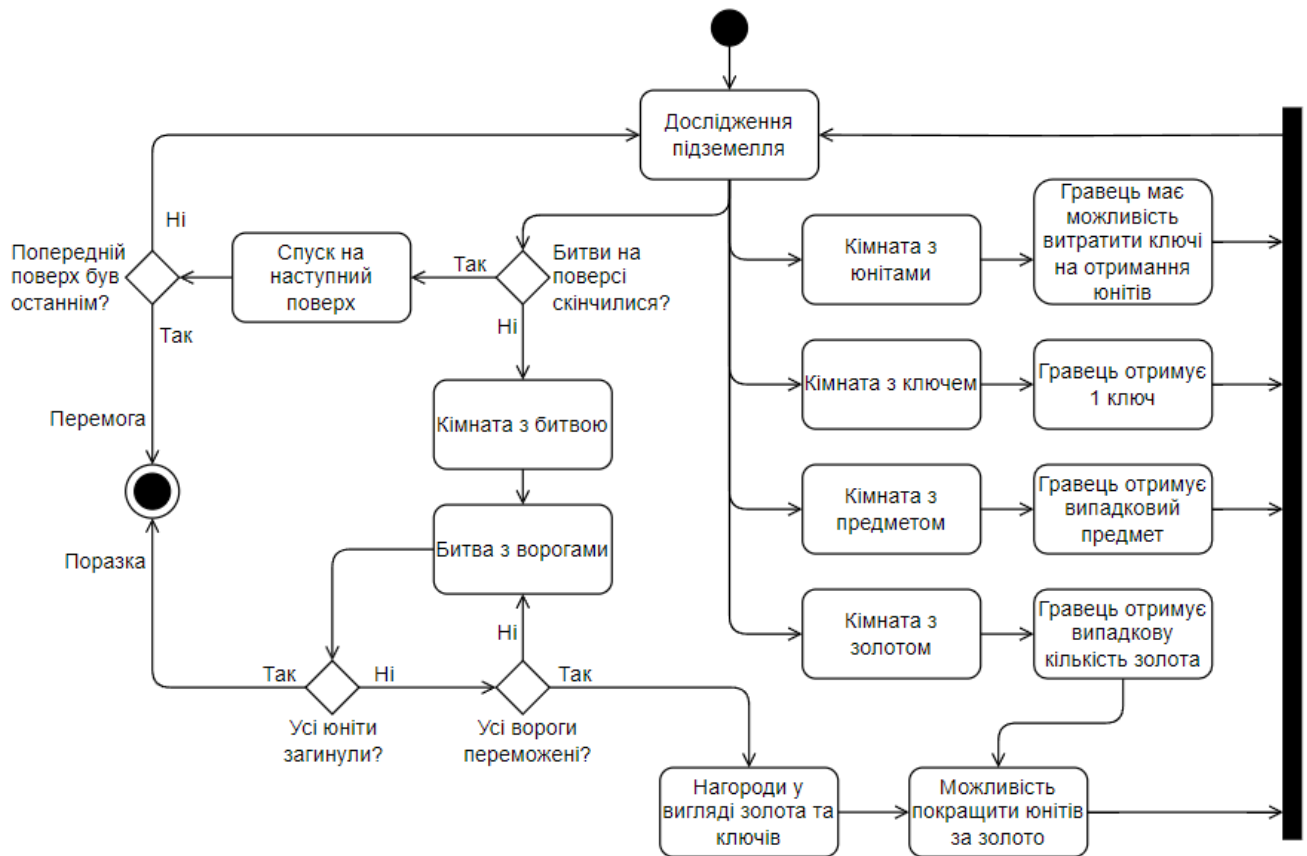


Рис. 2.2. Діаграма активності ігрового циклу.

Дослідження підземелля є основною діяльністю в грі, тому багато активностей в кінці повертаються до дослідження. Під час дослідження гравець обирає яку кімнату досліджувати, проте не завжди усі типи кімнат будуть присутні в підземеллі. Дія “Можливість покращити юнітів” доступна в будь-який момент гри, але логічно це робити відразу після отримання золота. Крім того, не обов’язково спускатися на наступний поверх відразу після перемоги над останнім ворогом, гравець матиме можливість дослідити підземелля до кінця, якщо потрібно.

Найважливішою та найскладнішою частиною розробки є розробка юнітів. Створимо діаграму класів (англ. Class diagram) для визначення оптимальної структури та взаємодії різних елементів юніта (див. рис. 2.3).

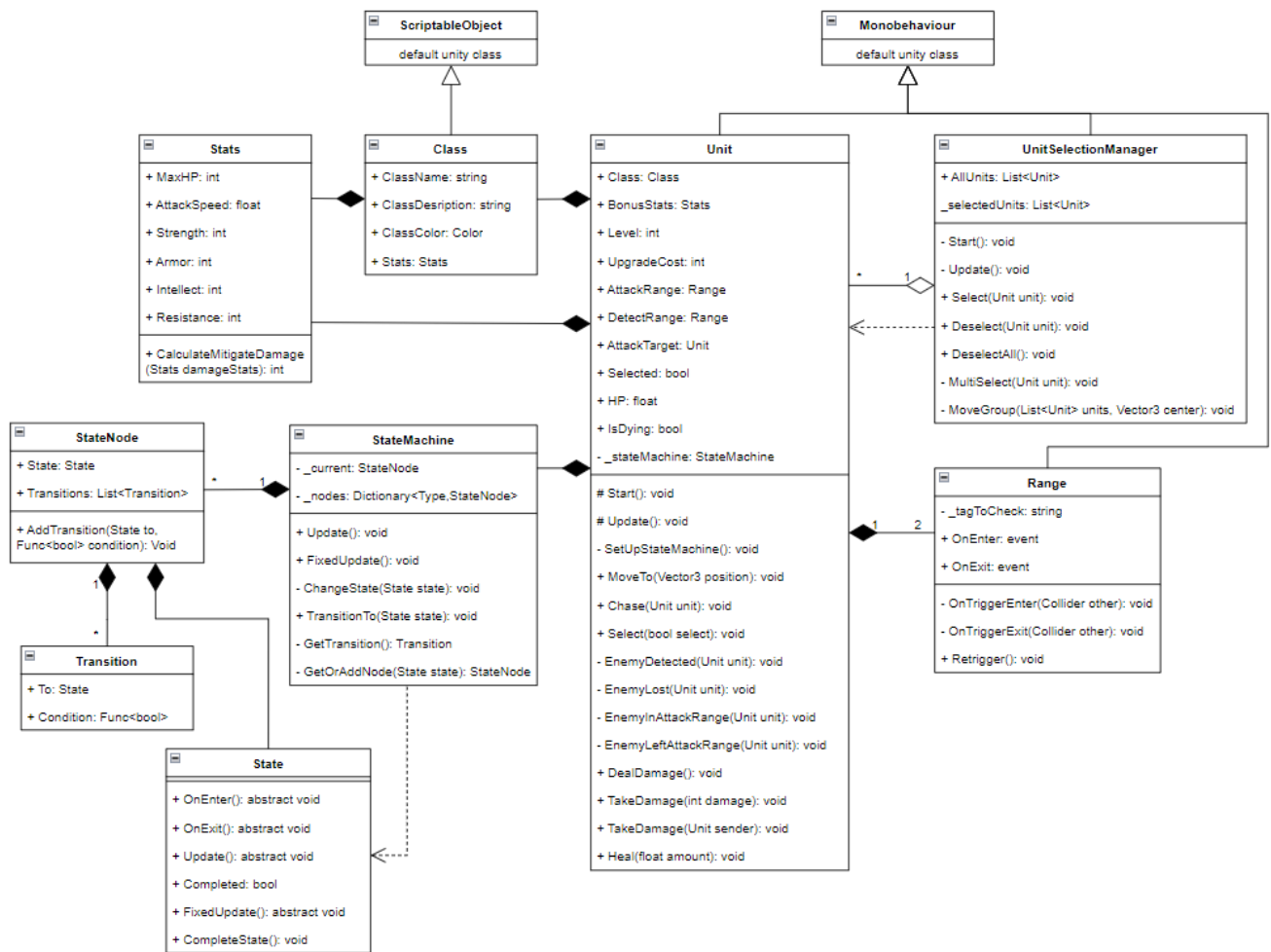


Рис. 2.3. Діаграма класів для роботи юнітів.

Деякі класи наслідують ScriptableObject та MonoBehaviour класи - це стандартні класи Unity які забезпечують взаємодію з рушієм. ScriptableObject - зручний спосіб зберігати дані у вигляді об'єктів, які не потребують прив'язки до сцени. MonoBehaviour дає змогу додати розроблений клас до об'єкту на сцені та взаємодіяти з ним.

Для зв'язків між класами часто використовується композиція (англ. composition), тому що ці класи є частиною класу Unit та не будуть функціонувати без нього. Натомість між класом Unit та UnitSelectionManager використовується агрегація, адже юніт може функціонувати незалежно від UnitSelectionManager.

Класи зображені на рисунку 2.3 будуть детальніше розглянуті в розділі 2.2.

2.2 Розробка гри

Етап 1 - State Machine.

Для розробки юнітів варто використати паттерн State Machine [4], тому що він чітко і логічно організує їх поведінку. Використання стейт машини дозволяє легко додавати нові стейти або змінювати існуючі, не впливаючи на інші частини коду. Крім того, стейт машини допомагають керувати складністю проекту, розбиваючи поведінку юнітів на окремі, зрозумілі частини.

Для початку створимо клас State. Майбутні стейти будуть наслідувати та по-своєму реалізовувати методи класу State, тому він є абстрактним.

Лістинг класу State:

```
public abstract class State
{
    public bool Completed;
    public abstract void OnEnter();
    public abstract void Update();
    public abstract void FixedUpdate();
    public abstract void OnExit();

    public void CompleteState() {
        Completed = true;
    }
    public void Enter() {
        Completed = false;
        OnEnter();
    }
    public void Exit() {
        OnExit();
    }
}
```

Тепер потрібно зробити можливість переходу між стейтами. Створимо клас Transition, який зберігатиме потрібну інформацію про перехід та клас StateNode, який зберігатиме перелік переходів для кожного стейту.

Лістинг класу Transition:

```
public class Transition
{
    public State To { get; }
    public Func<bool> Condition { get; }
    public Transition(State to, Func<bool> condition)
    {
        To = to;
    }
}
```

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						19
Змн.	Арк.	№ докум.	Підпис	Дата		

```

        Condition = condition;
    }
}

```

Ця реалізація є досить універсальною, адже можна передати будь-яку умову переходу завдяки Func<bool>. Також було додано поле Completed в клас стейту, що дозволить завершити стейт зсередини та перейти в наступний стейт.

Лістинг класу StateNode:

```

private class StateNode
{
    public State State;
    public List<Transition> Transitions;

    public StateNode(State state)
    {
        State = state;
        Transitions = new List<Transition>();
    }

    public void AddTransition(State to, Func<bool> condition)
    {
        Transitions.Add(new Transition(to, condition));
    }
}

```

Лістинг класу StateMachine:

```

public class StateMachine
{
    private StateNode _current;
    private Dictionary<Type, StateNode> _nodes = new();

    public void Update()
    {
        var transition = GetTransition();
        if (transition != null)
            ChangeState(transition.To);

        _current.State?.Update();
    }

    public void FixedUpdate()
    {
        _current.State?.FixedUpdate();
    }

    private void ChangeState(State state)
    {
        if (state == _current.State) return;
    }
}

```

```

        var previousState = _current.State;
        PreviousStateName = previousState.GetType().ToString();
        var nextState = _nodes[state.GetType()].State;

        previousState?.Exit();
        nextState?.Enter();
        _current = _nodes[state.GetType()];
    }
    private Transition GetTransition()
    {
        foreach (var transition in _current.Transitions)
            if (transition.Condition())
                return transition;

        return null;
    }
    public void AddTransition(State from, State to, Func<bool> condi-
tion)
    {
        GetOrAddNode(from).AddTransition(GetOrAddNode(to).State, con-
dition);
    }

    private StateNode GetOrAddNode(State state)
    {
        var node = _nodes.GetValueOrDefault(state.GetType());

        if (node == null)
        {
            node = new StateNode(state);
            _nodes.Add(state.GetType(), node);
        }

        return node;
    }
    public void TransitionTo(State state)
    {
        if(_current==null)
        {
            _current = _nodes[state.GetType()];
            _current.State?.Enter();
            return;
        }

        ChangeState(state);
    }
}

```

При ініціалізації, до стейт машини будуть додаватися переходи між стейтами використовуючи AddTransition(). AddTransition() спробує отримати Stat-

eNode для відповідного стейту, а якщо такого немає - створить його. Далі в методі Update() викликаємо GetTransition() для отримання переходу, умови якого були виконані, і виконуємо цей перехід. Метод TransitionTo() буде використано-вуватися коли негайно потрібно перейти в певний стейт та не важливо, який стейт наразі активний.

Юніти та вороги матимуть однакову поведінку та стейти, відмінність буде в налаштуваннях радіусів атаки, де буде зазначатися, на кого реагувати: юніта чи ворога. Розглянути як саме юніти та вороги будуть себе поводити буде зручно на діаграмі активності (див. рис. 2.4).

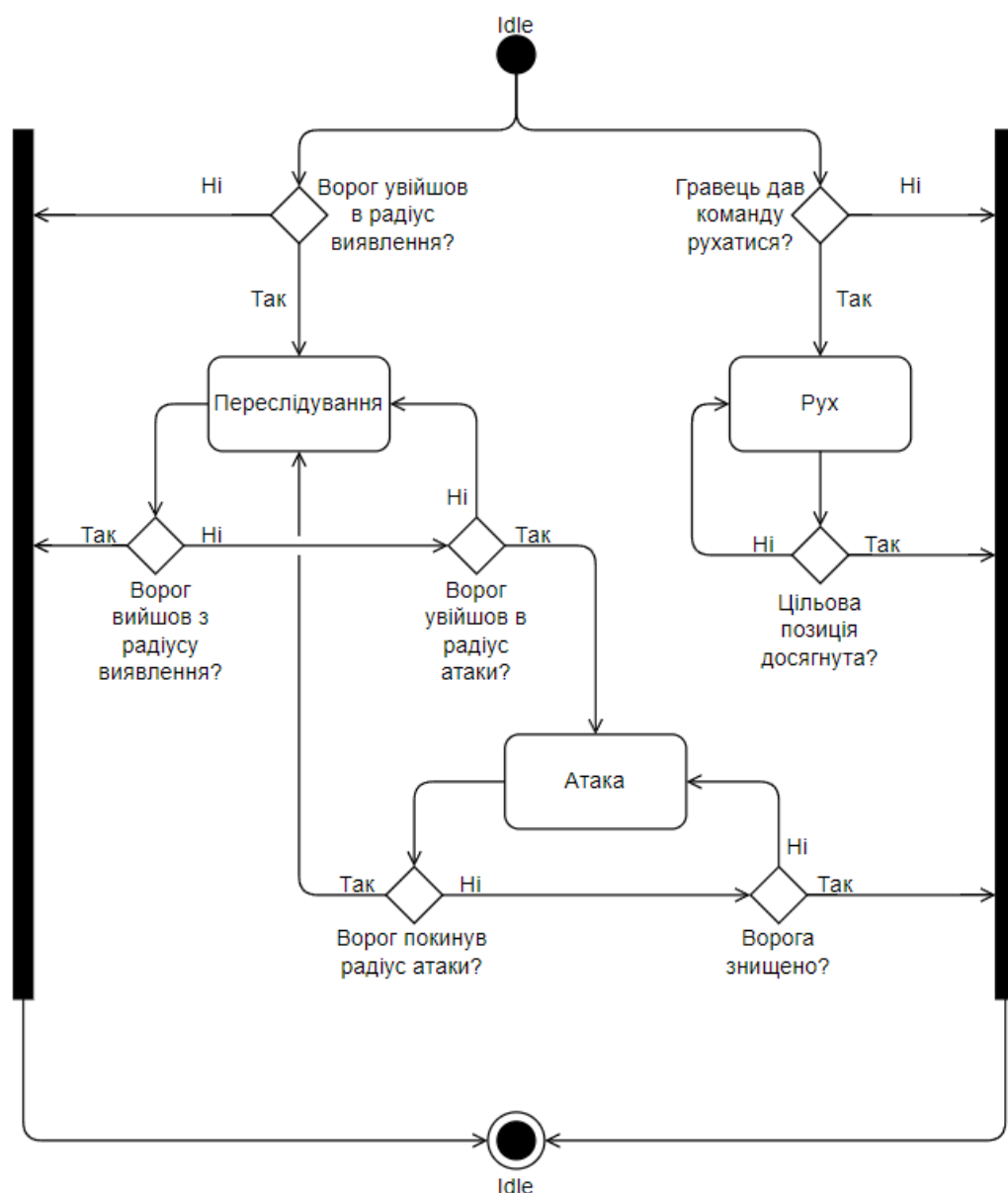


Рис. 2.4. Діаграма активності поведінки юніта.

Хоча стейт машина не має початку і кінця, кожен стейт починається або закінчується в Idle стейті (стані спокою), тому будемо вважати, що початок та кінець - це Idle стейт. Розгалуження “Гравець дав команду рухатися?” для ворогів завжди має результат “Ні”, адже гравець не має керувати ворогами. Окрім стейтів зазначених на діаграмі, ще є стейт смерті, в який переходить юніт або ворог, за допомогою методу TransitionTo(), коли вмирає. Лістинг усіх стейтів доступний в додатку А.

Етап 2 - Підготовка до розробки юнітів

Для реалізації радіусу виявлення та атаки створимо клас Range. Використовуючи OnTriggerEnter() та OnTriggerExit(), будуть надсилатися сигнали про вхід чи вихід з радіусу. Поле _tagToCheck дозволить в інспекторі Unity зазначити на які об’єкти реагувати. Юніти та вороги будуть розрізнятися по тегу: “FriendlyUnit” для юнітів гравця та “EnemyUnit” для ворогів. Для передачі сигналів про вхід та вихід було створено делегат RangeEvent(), що передає юніта, який увійшов чи вийшов з радіусу.

Лістинг класу Range:

```
public class Range : MonoBehaviour
{
    [SerializeField] private string _tagToCheck;
    private SphereCollider _collider;

    public delegate void RangeEvent(Unit unit);
    public event RangeEvent OnEnter;
    public event RangeEvent OnExit;

    private void Start()
    {
        _collider = GetComponent<SphereCollider>();
    }
    private void OnTriggerEnter(Collider other)
    {
        if (!other.gameObject.CompareTag(_tagToCheck)) return;

        OnEnter?.Invoke(other.GetComponent<Unit>());
    }
    private void OnTriggerExit(Collider other)
    {
        if (!other.gameObject.CompareTag(_tagToCheck)) return;
    }
}
```

```

        OnExit?.Invoke(other.GetComponent<Unit>());
    }
    public void ReTrigger()
    {
        _collider.enabled = false;
        Util.DelayOneFrame(()=> _collider.enabled = true);
    }
}

```

Рух юнітів буде реалізований за допомогою бібліотеки NavMesh [5], яка включає в себе алгоритми для пошуку найкоротшого шляху, оминаючи перешкоди. При команді рухатися юніт буде йти в місце куди клікнув гравець, при переслідуванні юніт буде йти до позиції ворога. На об'єкт юніта додамо компонент NavMeshAgent (див. рис. 2.5), який буде взаємодіяти з компонентом NavMeshSurface на сцені.

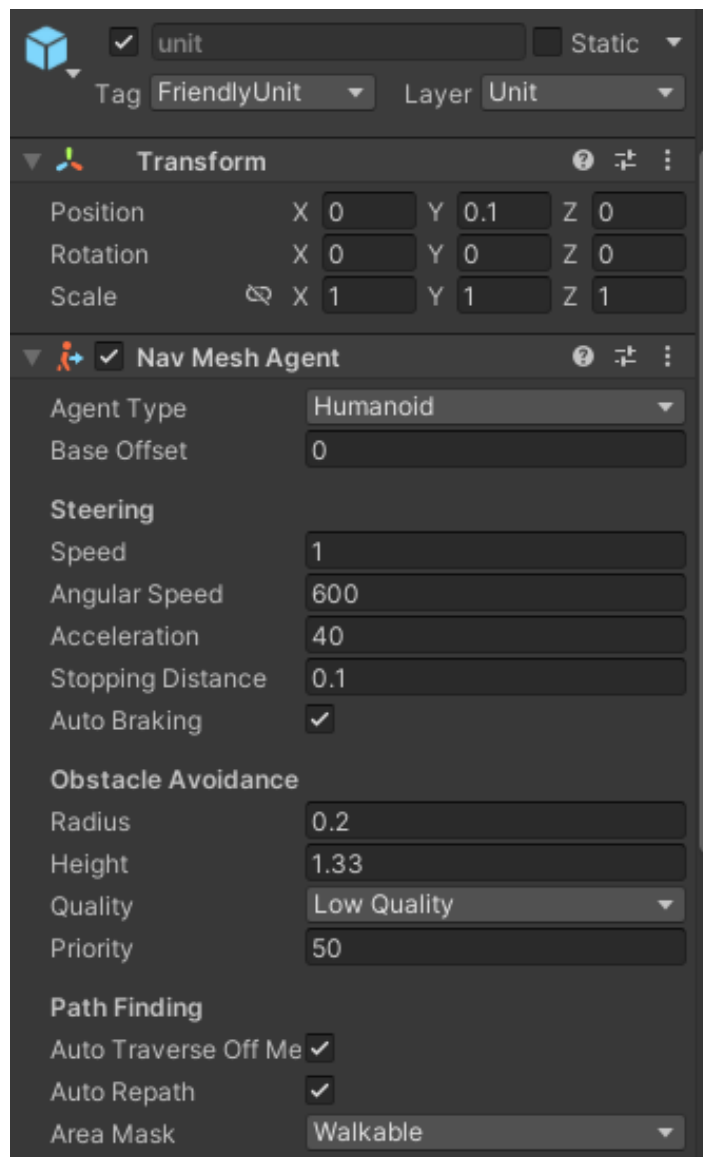


Рис. 2.5. Налаштування NavMeshAgent для юніта.

Етап 3 - Клас Unit

Тепер створимо клас Unit, який об'єднає весь вище розроблений функціонал. Оскільки клас Unit є досить об'ємним, розглянемо лише основні його методи. Повний лістинг класу Unit доступний на репозиторії [11].

Спочатку створимо методи для керування юнітом:

```
public virtual void Select(bool selected = true)
{
    Selected = selected;
    _visuals.ShowUiElements(selected);
    _visuals.BounceSelect();
}

public void MoveTo(Vector3 position)
{
    if (IsDying) return;

    _nav.SetDestination(position);
    _visuals.BounceMarker();

    AssignedPosition = position;
    AttackTarget = null;

    _stateMachine.TransitionTo(_moveState);
}

public void Chase(Unit unit)
{
    if (unit == null) return;
    AttackTarget = unit;

    AttackRange.ReTrigger();

    _visuals.TargetMarker();
    _stateMachine.TransitionTo(_chaseState);
}
```

Щоб гравець міг рухати лише обраних юнітів, а не всіх одночасно, було додано метод “Select()”. Далі реалізуємо реакції на вхід та вихід з радіусу виявлення та атаки:

```
private void EnemyDetected(Unit unit)
{
    if (unit.IsDying) return;
    if (AttackTarget == null)
    {
        Chase(unit);
    }
}
```

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						25
Змн.	Арк.	№ докум.	Підпис	Дата		

```

    }
}
private void EnemyLost(Unit unit)
{
    if (AttackTarget == unit)
    {
        AttackTarget = null;
    }
}
private void EnemyInAttackRange(Unit unit)
{
    if (unit.IsDying) return;
    if (AttackTarget != unit) return;

    _stateMachine.TransitionTo(_attackState);
}
private void EnemyLeftAttackRange(Unit unit)
{
    if (AttackTarget != unit) return;

    Chase(unit);
}

```

Юніт та вороги мають отримувати шкоду від атак чи оточення. Створимо метод “TakeDamage(int damage)”, який буде завдавати вказану цілочисельну шкоду юніту. Кожен юніт буде мати певний набір атакуючих та захисних характеристик, тому коли один юніт атакує іншого, викликатиметься TakeDamage(Unit sender) - перевантаження TakeDamage(int damage) методу. Це дозволить взяти потрібні характеристики відразу від нападника та обрахувати отриману шкоду.

```

public void TakeDamage(int trueDamage)
{
    if (IsDying) return;
    HP -= trueDamage;
    _visuals.DamageImpact(_animator.transform);
    _visuals.Flash();
    _visuals.ShakeHealthbar();

    if (HP <= 0)
    {
        HP = 0;
        AttackTarget = null;
        IsDying = true;
        _stateMachine.TransitionTo(_unitDeathState);
    }
}

```

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						26
Змн.	Арк.	№ докум.	Підпис	Дата		

```

public void TakeDamage(Unit sender)
{
    if (IsDying) return;

    int damage = Stats.CalculateMitigatedDamage(sender.Stats);
    TakeDamage(damage);

    if (_stateMachine.CurrentStateName==nameof(UnitIdle) && At-
tackTarget == null)
    {
        Chase(sender);
        AttackRange.ReTrigger();
    }
}

```

Також, якщо юніт знаходиться в стані спокою та отримує шкоду від ворога, юніт почне переслідувати та атакувати цього ворога.

Етап 4 - Стати юнітів

В лістингу методу TakeDamage(Unit sender) для обрахунку шкоди використовується метод CalculateMitigatedDamage() класу Stats. Клас Stats відповідає за усі статистичні дані юнітів та має деякі допоміжні функції.

Перед тим як розглядати статистичні дані юнітів, варто роз'яснити типи шкоди. У грі буде три типи шкоди: фізична, магічна та чиста шкода (англ. true damage). Відповідно фізична шкода блокується фізичним захистом, магічна - магічним захистом. Чиста шкода ігнорує усі захисти та влучає відразу по здоров'ю юніта.

Кожен юніт матиме 6 статів і всі статистичні дані окрім швидкості атаки це цілочисельні значення (див. рис. 2.6).

-  Vitality (Життєва сила) - максимальне здоров'я юніта
-  AttackSpeed (Швидкість атаки) - скільки атак юніт виконує за секунду
-  Strenght (Сила) - кількість фізичної шкоди, яку завдає юніт
-  Armor (Обладунки) - значення фізичного захисту
-  Intellect (Інтелект) - кількість магічної шкоди, яку завдає юніт
-  Resistance (Опір) - значення магічного захисту

Рис. 2.6. Перелік статів та їх умовні позначення.

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						27
Змн.	Арк.	№ докум.	Підпис	Дата		

Інтелект також може застосовуватися для деяких ефектів здібностей юнітів, тому обраховуємо завдану магичну шкоду тільки якщо сила нападника дорівнює 0.

Лістинг класу Stats:

```
public class Stats
{
    public int MaxHP;
    public float AttackSpeed = 1;

    public int Strength;
    public int Armor;

    public int Intellect;
    public int Resistance;

    public int CalculateMitigatedDamage(Stats damageStats)
    {
        int damage;
        if(damageStats.Strength==0)
        {
            damage = damageStats.Intellect - Resistance;
        }
        else
        {
            damage = damageStats.Strength - Armor;
        }
        return damage <= 0 ? 0 : damage;
    }
    public static Stats operator +(Stats a, Stats b)
    => new Stats(a.MaxHP + b.MaxHP,
        a.AttackSpeed + b.AttackSpeed,
        a.Strength + b.Strength,
        a.Armor + b.Armor,
        a.Intellect + b.Intellect,
        a.Resistance + b.Resistance);
}
```

В клас Stats також було додано перевантаження оператора “+”, що дозволить зручно додавати статистичні дані. Це знадобиться для реалізації покращень юніта.

Базові статистичні дані юніта зберігаються в ScriptableObject класі Class, який також відповідає за назву та опис юніта. Class в ігровій термінології - це архетип чи професія персонажа, наприклад: воїн, чаклун чи лучник. Може бути безліч персонажів, які відносяться до одного класу, і кожен з них має відповідні статистичні дані та здібності свого класу.

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

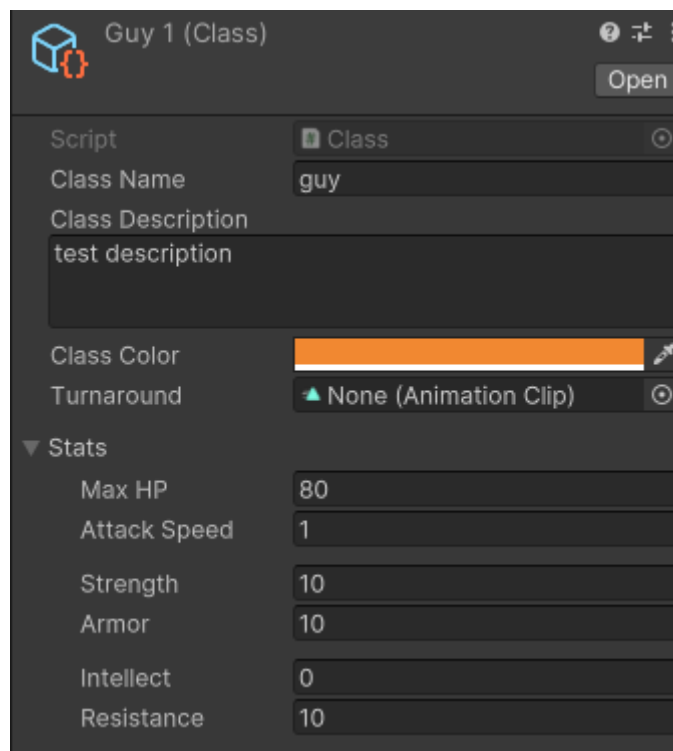


Рис. 2.7. Приклад класу юніта.

Крім того, юніт має поле BonusStats, яке відповідає за отримані покращення. Коли юніт буде отримувати покращення для певного стату, значення буде додаватися в BonusStats, аби не змінювати базові статистичні дані, спільні для усіх юнітів цього класу. Тоді, для того щоб отримати поточні статистичні дані юніта, потрібно додати його базові та бонусні статистичні дані.

Етап 5 - Керування юнітами

З завершеним функціоналом юнітів можна приступити до розробки керування ними. Спочатку розглянемо як саме керування буде в грі:

1. При натисканні ЛКМ на юніта, обирається лише цей юніт;
2. При натисканні ЛКМ з зажатою кнопкою “Shift” на юніта, цей юніт додається або прибирається з обраних;
3. При натисканні ЛКМ з зажатою кнопкою “Ctrl” на юніта, обираються усі юніти такого самого класу;
4. При натисканні ЛКМ на землю, всі юніти прибираються з обраних;
5. При натисканні ПКМ на землю, обрані юніти почнуть рухатися в натиснуту позицію.

Отже, створимо клас UnitSelectionManager, повний лістинг якого доступний на репозиторії [11]. Методи Select(), SelectClass(), Deselect() та DeselectAll() відповідають за обирання юнітів:

```
public void Select(Unit unit)
{
    if (_selectedUnits.Contains(unit)) return;

    if (unit.IsEnemy) DeselectAll();
    else DeselectEnemy();

    _selectedUnits.Add(unit);
    unit.Select(true);
}
public void SelectClass(string name)
{
    DeselectAll();
    foreach (var unit in AllUnits)
    {
        if (unit.Class.ClassName == name)
            Select(unit);
    }
}
public void Deselect(Unit unit)
{
    unit.Select(false);
    _selectedUnits.Remove(unit);
}
public void DeselectAll()
{
    foreach (var unit in _selectedUnits)
    {
        if (unit == null) continue;
        unit.Select(false);
    }
    _selectedUnits.Clear();
}
```

Метод MultiSelect() реалізує логіку для цих методів:

```
private void MultiSelect(Unit unit)
{
    if (Input.GetKey(KeyCode.LeftControl))
    {
        DeselectAll();
        SelectClass(unit.Class.ClassName);
        return;
    }

    if (Input.GetKey(KeyCode.LeftShift))
```

```

    {
        if(unit.Selected)
        {
            Deselect(unit);
            return;
        }
    }
    else DeselectAll();

    Select(unit);
}

```

Рух забезпечує метод MoveGroup(). Щоб всі обрані юніти не скупчувалися в одну точку, використовуючи метод GetCenter() обраховується центральна позиція поточної групи юнітів. Далі, для кожного юніта обраховується відступ від центру і додається до нового центру. Таким чином юніти зберігають свій стрій і переміщуються в вказану точку.

```

private void MoveGroup(List<Unit> units,Vector3 center)
{
    Vector3 oldCenter = GetCenter(units);

    foreach (var unit in units)
    {
        if (unit.IsEnemy) continue;
        Vector3 offset = unit.transform.position - oldCenter;

        unit.MoveTo(center + offset);
    }
}

private Vector3 GetCenter(List<Unit> list)
{
    var totalX = 0f;
    var totalY = 0f;
    var totalZ = 0f;

    foreach (var obj in list)
    {
        totalX += obj.transform.position.x;
        totalY += obj.transform.position.y;
        totalZ += obj.transform.position.z;
    }

    return new Vector3(totalX, totalY, totalZ) / list.Count;
}

```

Тепер в методі Update() зчитуємо натискання гравця та використовуючи Physics.Raycast() дізнаємося чи гравець натиснув на юніта або землю.

```

private void Update()
{
    var ray = _camera.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit;

    if (Input.GetMouseButtonDown(0))
    {
        if (Physics.Raycast(ray, out hit, Mathf.Infinity, Layer-
Mask.Mask("Unit")))
            MultiSelect(hit.collider.GetComponent<Unit>());
        else
            DeselectAll();
    }
    if (Input.GetMouseButtonDown(1) && _selectedUnits.Count > 0)
    {
        if (Physics.Raycast(ray, out hit, Mathf.Infinity, Layer-
Mask.Mask("Ground")))
            MoveGroup(_selectedUnits, hit.point);
    }
}

```

Етап 6 - Юніти та вороги

Юнітів та ворогів можна поділити на 3 типи:

1. Юніти ближнього бою, які мають малий радіус атаки;
2. Юніти дальнього бою, які вистрілюють снаряди, що завдають шкоди;
3. Магічні юніти, які завдають шкоди по області заклинаннями.

Для базової версії гри створимо 3 юніти та 3 ворога, кожен зі своїми стататами та здібностями (див. табл. 2.1).

Таблиця 2.1

		Юніти			Вороги		
Назва		Knight	Ranger	Wizard	Goblin	Undead	Cyclops
Тип		Ближній	Дальній	Магічний	Ближній	Магічний	Ближній
Стати	Життєва сила	90	55	60	40	80	300
	Швид. атаки	0.57	0.67	0.60	0.77	0.50	0.55
	Сила	10	20	0	18	0	30
	Обладунки	15	3	2	6	0	0
	Інтелект	0	15	20	0	16	0
	Опір	5	3	10	5	15	0
Здібність		За кожного іншого Knight в команді отримує +2 сили та +1 обладунки	Має шанс завдати подвійної шкоди, який дорівнює Інтелекту	Кожну третю атаку завдає шкоду по великій області перед собою	Має 30% шанс ігнорувати отриману шкоду	Завдає шкоду цілі та всім юнітам навколо неї	Відновлює 8% відсутнього здоров'я щосекунди

Для кожної здібності створений окремий клас, що наслідує клас Ability. Клас Ability зберігає всю потрібну інформацію про здібність та має віртуальний методи, який дочірні класи можуть використовувати для свого функціоналу.

Етап 7 - Object pool

Розглянемо детальніше як працюють юніти дальнього бою. Кожен постріл створює об'єкт снаряда, який летить у свою ціль. Коли снаряд досягає цілі він самознищується.

Нехай ми маємо 10 юнітів дальнього бою в команді - це постійне створення та знищення 10 об'єктів. До того ж, можна покращувати швидкість атаки і тоді інтервал між пострілами стане менший. Це може значно вплинути на продуктивність гри, тому варто використати паттерн Object pool [6].

Object Pool (пул об'єктів)- це патерн проектування, який використовується для ефективного створення та повторного використання об'єктів. Використовуючи Object pool ми зможемо деактивувати снаряди, які вже влучили в ціль, та активувати їх знову при пострілі. Таким чином буде достатньо заздалегідь створити декілька екземплярів снаряду та використовувати їх повторно. Якщо потрібно виконати постріл, а деактивованих снарядів не вистачає, буде створюватися новий та додаватися до пулу.

Лістинг класу ObjectPool:

```
public class ObjectPool<T>
{
    public List<T> List { get; private set; }
    private Func<T, bool> _predicate;
    private Func<T> _create;
    private Action<T> _enable;
    private Action<T> _disable;

    public ObjectPool(Func<T, bool> predicate, Func<T> create, Action<T> enable, Action<T> disable)
    {
        List = new List<T>();
        _predicate = predicate;
        _create = create;
        _enable = enable;
        _disable = disable;
    }
}
```

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

```

public void AddDefault(T obj)
{
    List.Add(obj);
    _disable(obj);
}

public T GetObject()
{
    foreach (T obj in List)
    {
        if (_predicate(obj))
        {
            _enable(obj);

            return obj;
        }
    }
    var newObj = _create();
    List.Add(newObj);

    return newObj;
}
}

```

Ця реалізація є максимально універсальною, адже ми не задаємо як саме створюються, деактивуються та активуються об'єкти пулу. Окрім того, використання дженеріків (generic) [7] дозволить працювати з будь-якими об'єктами, не тільки снарядами юнітів. Розглянемо детальніше кожен Func та Action:

1. Func<T, bool> _predicate - приймає параметром об'єкт пулу та повертає булеве значення. Це умова для активації об'єкту;
2. Func<T> _create - повертає створений об'єкт пулу;
3. Action<T> _enable - приймає об'єкт пулу та активує його;
4. Action<T> _disable - приймає об'єкт пулу та деактивує його.

Реалізація пулу об'єктів на прикладі снарядів юніта:

```

_pool = new ObjectPool<Projectile> (
    (obj) => !obj.gameObject.activeSelf,
    () => Instantiate(_projectile, transform),
    (obj) => obj.gameObject.SetActive(true),
    (obj) => obj.gameObject.SetActive(false)
);

_pool.AddDefault(GetComponentInChildren<Projectile>());

```

Етап 8 - Генерація підземелля

Перед тим як перейти до генерації підземелля, розглянемо, що саме має генеруватися. Підземелля складається з кімнат, що розміщені в комірках на сітці. Кожна кімната має проходи, що ведуть в неї, нумеровані від 0 до 3 за годинниковою стрілкою (див. рис. 2.6). Це зручно, адже тоді ми можемо отримати протилежний прохід просто додавши 2. Відповідно, при генерації проходи мають бути об'єднані між собою, не має бути проходів, які ведуть у стіну.

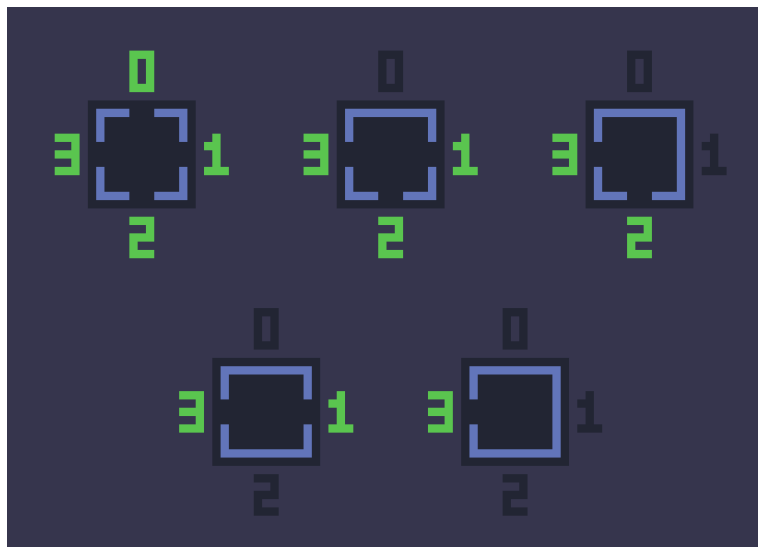


Рис. 2.8. Можливі кімнати.

На рисунку 2.6 зображено які кімнати будуть використовуватися для генерації підземель. Обертаючи їх, ми отримуємо 15 можливих варіантів для кімнати. Розмір підземелля буде обмежений сіткою 8x7 комірок. Приклад побудованого підземелля зображений на рисунку 2.7.

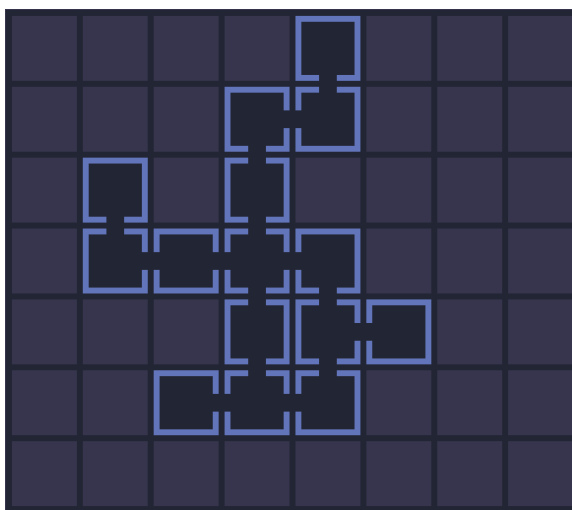


Рис. 2.9. Приклад підземелля.

Для процедурної генерації підземель буде використано алгоритм Wave Function Collapse [8]. Вхідними даними для WFC є набір усіх можливих станів комірки та певні правила. Розглянемо як працює цей алгоритм на діаграмі активності (див. рис. 2.8).

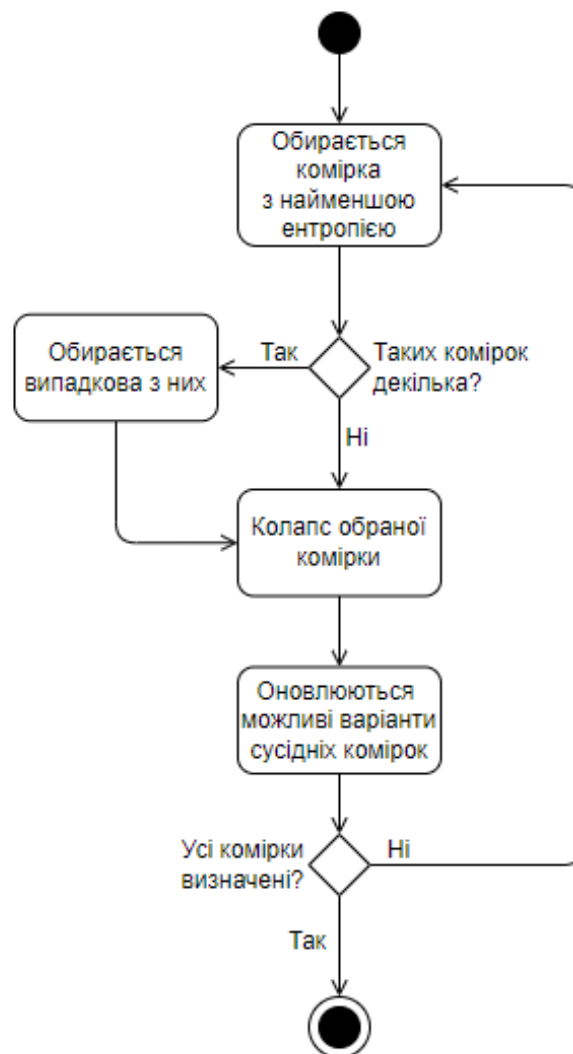


Рис. 2.10. Діаграма активності алгоритму WFC.

На початку алгоритму усі комірки можуть набути будь-який стан з заданого на вході набору. Кількість можливих варіантів для комірки також називається ентропією (англ. entropy). В процесі виконання алгоритму комірки поступово отримують визначений стан - це називається колапсом (англ. collapse). Під часу колапсу для комірки випадково обирається один з доступних їй варіантів і цей стан надалі не змінюється. Сусідні комірки оновлюють свої можливі варіанти відповідно до заданих правил та комірки, що зазнала колапсу.

В результаті WFC ми зможемо отримати випадкове, але структуроване підземелля, що додасть реіграбельності грі та зробить кожне нове проходження несхожим на попереднє.

Почнемо з класу для комірки MapCell. За визначення комірки відповідає метод Collapse(). Щоб мати трохи більше контролю над генерацією, кожна можлива кімната матиме певну ймовірність, що її оберуть. Тоді для вибору випадкової кімнати потрібно використати метод зважених випадкових елементів (англ. weighted random choices). Обраховуємо загальну суму усіх ймовірностей та випадково обираємо число від 1 до значення загальної суми. Далі в циклі перебираємо усі можливі кімнати та додаємо їх ймовірності. Кожну ітерацію циклу перевіряємо, чи є обране число меншим за набрану суму, і якщо так, останній доданий елемент стає результатом.

```
public void Collapse()
{
    int totalWeight = Options.Sum(option => option.Probability);
    int randomNumber = Random.Range(1, totalWeight + 1);

    int cumulativeWeight = 0;
    foreach (var option in Options)
    {
        cumulativeWeight += option.Probability;
        if (randomNumber <= cumulativeWeight)
        {
            SetTile(option);
            break;
        }
    }
}

public void SetTile(MapTile tile)
{
    Tile = tile;

    _tileImage.sprite = tile.Sprite;
    _tileImage.transform.DORotate(new Vector3(0,0,-tile.Angle),0);
    IsCollapsed = true;
    Options.Clear();
}
```

Далі розробимо метод для оновлення можливих кімнат комірки. Потрібно врахувати, що проходи можуть бути суміжні лише з проходами, а стіни - лише з стінами.

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

```

public void UpdateOptions(int direction, bool passage)
{
    if (IsCollapsed) return;

    int neededDirection = (direction + 2) % 4;

    var list = new List<MapTile>();
    foreach (MapTile tile in Options)
    {
        if (passage && tile.Passages.Contains(neededDirection))
            list.Add(tile);
        if (!passage && !tile.Passages.Contains(neededDirection))
            list.Add(tile);
    }
    Options = list;
}

```

Тепер в класі MapController реалізуємо сам алгоритм WFC. Повний лістинг класів MapController та MapCell доступний на репозиторії [11]. Спочатку потрібен метод для пошуку комірки з найменшою кількістю можливих кімнат.

```

private MapCell GetLowestEntropyCell()
{
    int min = 99;
    List<MapCell> minCells = new List<MapCell>();

    foreach (var cell in _allCells.Values)
    {
        if (cell.Options.Count < min)
        {
            min = cell.Options.Count;

            minCells.Clear();
            minCells.Add(cell);
        }
        else if (cell.Options.Count == min)
        {
            minCells.Add(cell);
        }
    }
    int rand = Random.Range(0, minCells.Count);
    return minCells.Count == 0 ? null : minCells[rand];
}

```

Також потрібен спосіб отримувати сусідні комірки, для цього створимо метод GetNeighbors(). Позиції комірок зберігаються у вигляді “(int,int)”, де Item1 це X, а Item2 це Y. Також, тернарними операціями врахуємо можливість розміщення комірок на краю сітки.

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

```

public List<MapCell> GetNeighbors((int, int) pos)
{
    List<MapCell> cells = new List<MapCell>
    {
        pos.Item2!=0 ? _allCells[(pos.Item1, pos.Item2 - 1)] : null,
        pos.Item1!=7 ? _allCells[(pos.Item1 + 1, pos.Item2)] : null,
        pos.Item2!=6 ? _allCells[(pos.Item1, pos.Item2 + 1)] : null,
        pos.Item1!=0 ? _allCells[(pos.Item1 - 1, pos.Item2)] : null
    };

    return cells;
}

```

Результат методу GetNeighbors() - це список, який має 4 елементи. Кожен елемент відповідає за певний напрямок, як згадувалося раніше. Таким чином, щоб отримати сусіда, наприклад, справа від комірки, достатньо вказати номер напрямку: масив GetNeighbors(cell) з індексом 1.

Використаємо розроблені методи для генерації підземелля в методі GenerateMap(). Якщо метод GetLowestEntropyCell() повернув “null”, це означає, що підземелля не має більше вільних проходів, в такому випадку генеруємо підземелля спочатку.

```

public void GenerateMap()
{
    for (int j = 0; j < _roomLimit+1; j++)
    {
        var cell = GetLowestEntropyCell();
        if (cell is null)
        {
            ResetMap();
            GenerateMap();
            return;
        }

        cell.Collapse();

        var neighbours = GetNeighbors(cell.Position);
        for (int i = 0; i < neighbours.Count; i++)
        {
            if (neighbours[i] is null) continue;

            neighbours[i].UpdateOptions(i,
cell.Tile.Passages.Contains(i));
        }
    }
}

```

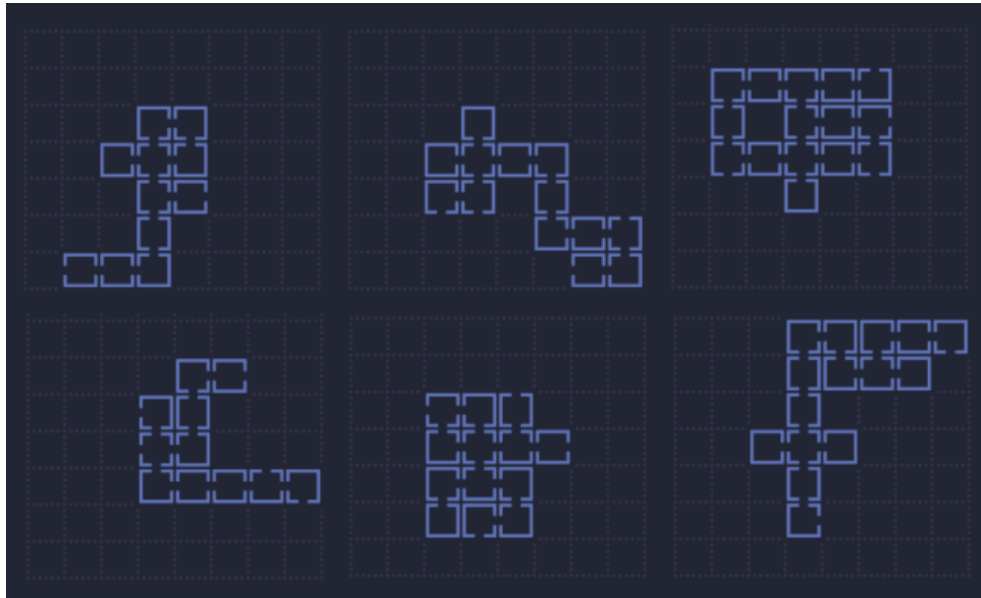


Рис. 2.11. Результати генерації.

Генерація працює коректно, але лишаються відкриті проходи. Додамо поля `IsClosed` (bool) та `_closedPassages` (int) в клас `MapCell` для відслідковування незакритих кімнат. Якщо `_closedPassages` дорівнює кількості проходів кімнати, `IsClosed` стає true. Далі створимо метод `CloseOpenPassages()` в класі `MapController` для закриття усіх незакритих кімнат. Для того щоб закрити кімнату потрібно аналізувати сусідні комірки та підібрати нову кімнату, де усі проходи будуть закриті.

```
private void CloseOpenPassages()
{
    var openCells = _allCells.Where((x) => x.IsCollapsed &&
!x.IsClosed);
    foreach (var cell in openCells)
    {
        var neighbors = GetNeighbors(cell.Position);

        List<int> neededPassages = new List<int>();
        for (int i = 0; i < neighbors.Count; i++)
        {
            if (neighbors[i] is not null && neigh-
bors[i].HasPassageFor(i)) neededPassages.Add(i);
        }
        var tile = _tiles.Find((x) =>
            x.Passages.SequenceEqual(neededPassages)
        );
        cell.SetTile(tile);
    }
}
```

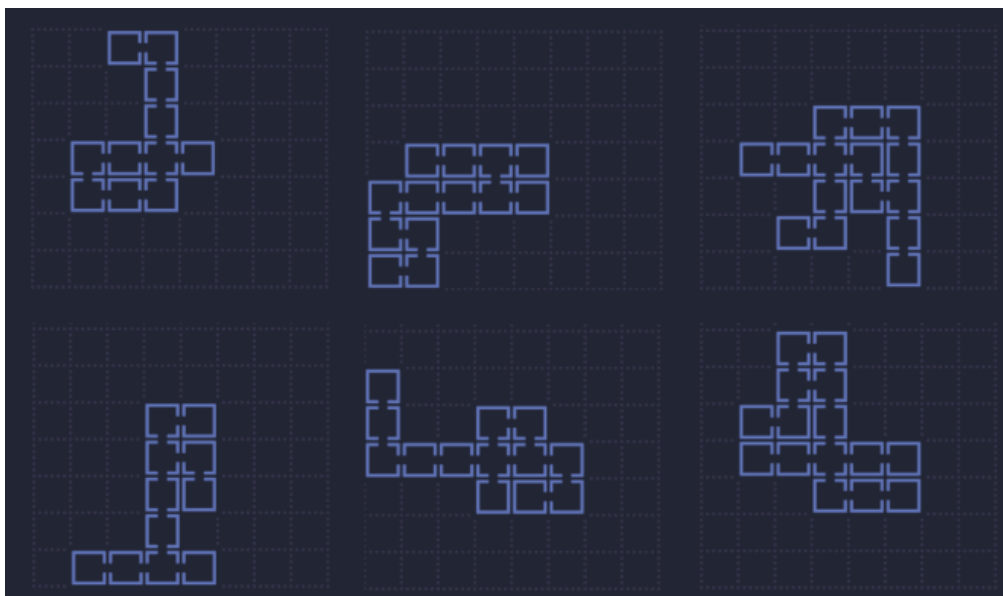


Рис. 2.12. Результати генерації з закритими проходами.

Тепер потрібно наповнювати згенеровані кімнати. Як було зазначено в першому розділі, підземелля буде поділено на поверхи. На першому поверсі буде 10 кімнат. Коли гравець спускається на поверх нище, кількість кімнат буде збільшуватися на 2. На рисунку 2.13 зображені усі можливі кімнати та їх умовні позначки.

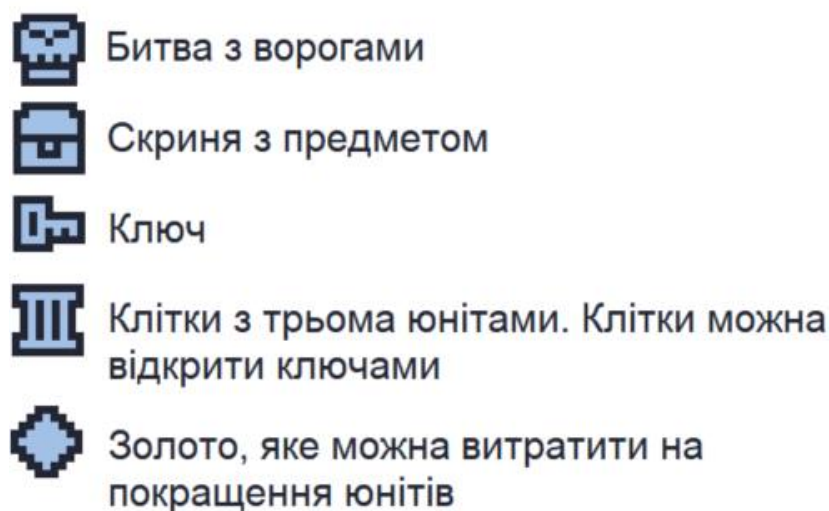


Рис. 2.13. Кімнати та їх умовні позначки.

На кожному поверсі половина кімнат матимуть битви з ворогами, 2 кімнати завжди будуть з клітками, а решта матимуть випадкові нагороди (скриня, ключ чи золото). В базовій версії гри буде 3 поверхи і оскільки максимальний розмір команди гравця - 25, розмістивши по 2 кімнати з клітками на кожному поверсі, гравець матиме змогу зібрати повну команду. В кожній кімнаті з клітками один

ключ дається безкоштовно, а решту треба шукати в підземеллі та отримувати за перемоги в битвах.

Метод для наповнення кімнат досить простий:

```
private void FillRooms()
{
    var cells = _allCells.Values.Where((x) =>
x.IsCollapsed).OrderBy(x => Random.value).ToList();

    for (int i = 0; i < cells.Count; i++)
    {
        if (i < cells.Count / 2)
        {
            if (i < 2)
                cells[i].SetRoom(_rooms[1]/*Cages*/);
            else
                cells[i].SetRoom(_rooms[Random.Range(2, 5)]);
        }
        else cells[i].SetRoom(_rooms[0]/*Battle*/);
    }
}
```

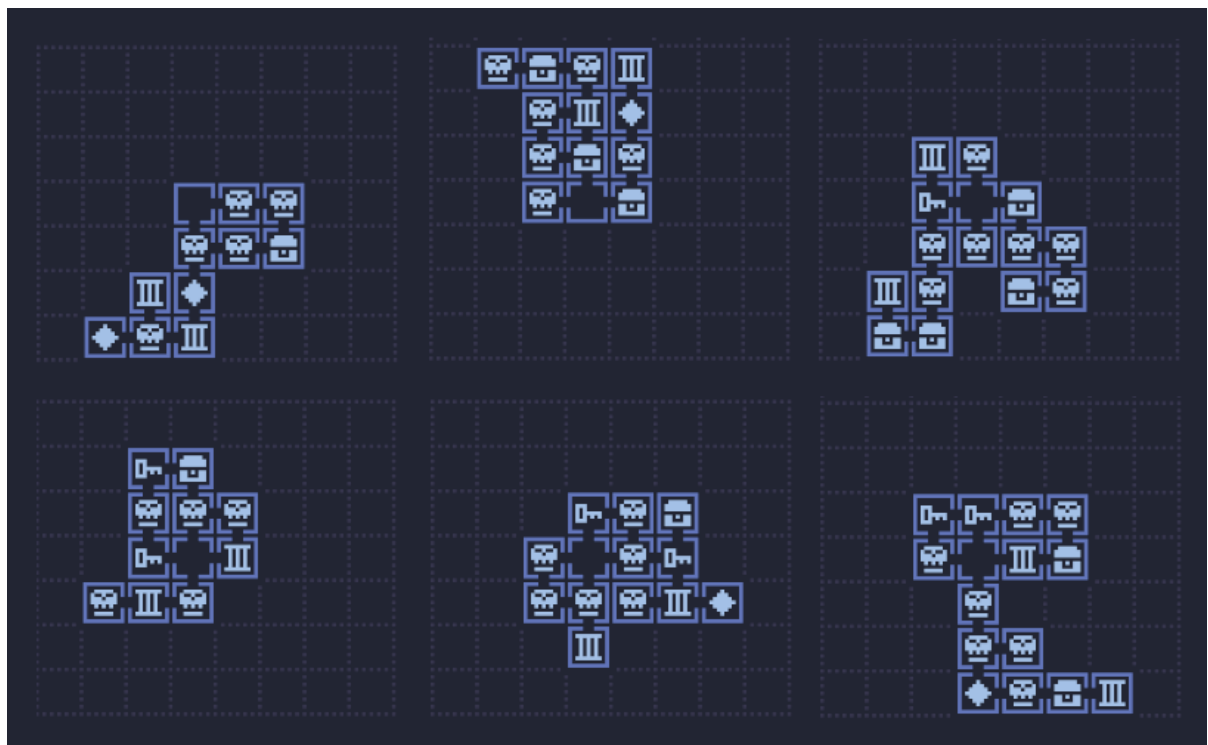


Рис. 2.14. Результати генерації з наповненими кімнатами

Список кімнат `_rooms` заповнюється в інспекторі (див. рис. 2.15). Кожна кімната окрім битви має об'єкт, який створюється при генерації підземелля. Створюються вони неактивними і активуються коли гравець заходить у відповідну кімнату.

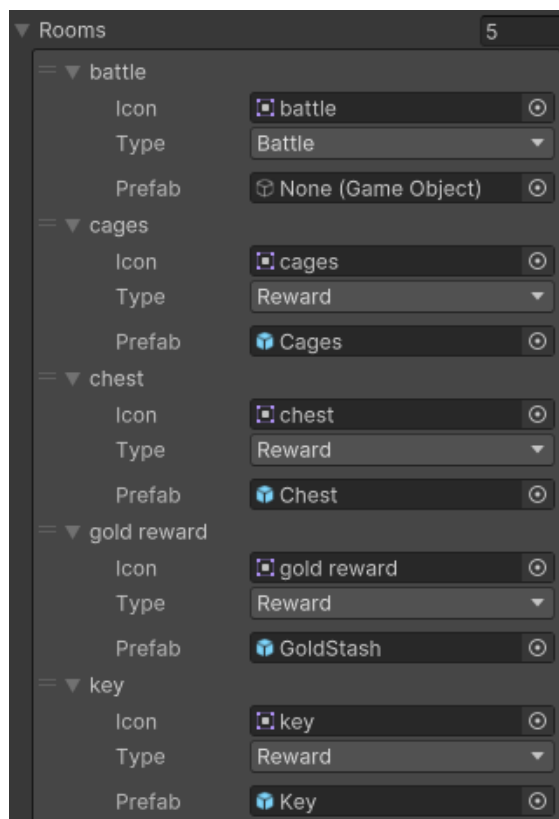


Рис. 2.15. Список кімнат в інспекторі.

Наприклад, коли гравець перейде в кімнату з золотом, він матиме змогу натиснути на об'єкт GoldStash, який додає випадкову кількість золота до гаманця. В скринях гравець може отримати предмет, що належить до однієї з категорій:

1. Зілля зцілення;
2. Тимчасові покращення статів для юнітів;
3. Тимчасові зниження статів для ворогів;
4. Вибухівка, що завдає чистої шкоди ворогам;
5. Спорядження з постійними покращеннями статів для юнітів.

Етап 9 - Битви

Кімнати з битвами працюють дещо інакше. Коли гравець потрапляє в таку кімнату, він не може її покинути доки не переможе в битві. Червоними стрілочками вказується звідки будуть виходити вороги (обирається випадково для кожної битви) і гравець має змогу стратегічно розмістити своїх юнітів. Коли гравець готовий битися, він має натиснути на кнопку “Start Battle”. Коли починається битва, з певним інтервалом з'являються вороги. Якщо гравець

перемагає усіх ворогів, він отримує золото та ключі. Якщо всі юніти гравця гинуть - гра закінчується поразкою.

Вороги для битви також підбираються випадково. Кожен ворог має рівень небезпеки: Goblin - 10, Undead - 20, Cyclops - 50. Битви теж мають рівень небезпеки, але він слугує як сума небезпеки усіх ворогів цієї битви. Наприклад, ми маємо битву з небезпекою 40, тоді є лише 3 комбінації ворогів для неї:

1. Goblin, Goblin, Goblin, Goblin;
2. Goblin, Goblin, Undead;
3. Undead, Undead.

Рівень небезпеки битв та інші параметри, пов'язані з битвами, вказуються в інспекторі для кожного поверху (див. рис. 2.16).

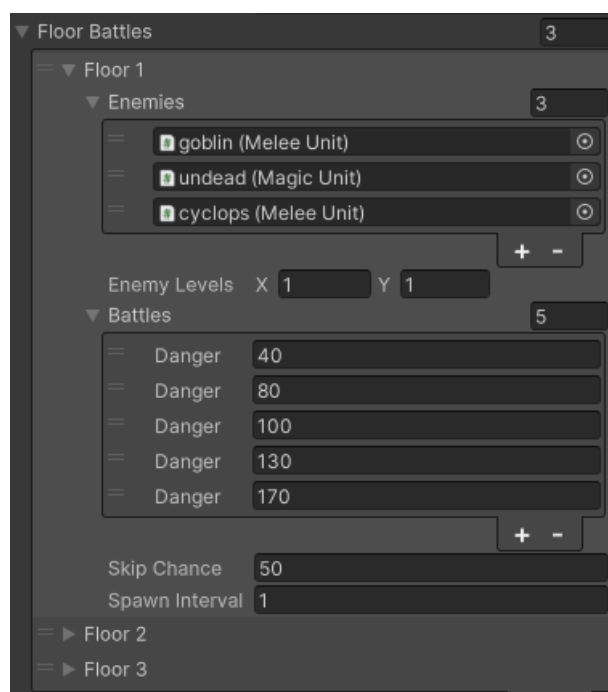


Рис. 2.16. Налаштування битв на поверсі.

Enemies - це перелік ворогів, яких можна зустріти на поверсі.

Enemy Levels - це Vector2, який відповідає за мінімальний (X) та максимальний (Y) рівень ворогів на поверсі. Гравець піднімає рівень своїм юнітам, покращує їх стати, а вороги, за кожен рівень отримують випадкові покращення.

Battles - це послідовність битв, а точніше їх рівень небезпеки. Неважливо, в які саме кімнати битв заходить гравець, рівень небезпеки обирається послідовно.

Це гарантує, що гравець випадково не натрапить на битву з небезпекою 170 на самому початку.

Spawn Interval - це інтервал в секундах між появою ворогів. Skip Chance - це ймовірність того, що поява ворога відкладеться на ще один інтервал. Це дозволяє керувати складністю битв, адже чим менше Spawn Interval та Skip Chance, тим складніша сутичка з ворогами.

Лістинг класу BattleController, який реалізує весь функціонал битв, доступний в додатку Б.

2.3 Створення візуальних елементів

Етап 1 - Інтерфейс

Інтерфейс гри буде релізований використовуючи вбудовані елементи інтерфейсу Unity, такі як: Canvas, TextMeshPro, Button та інші. Візуально інтерфейс матиме піксельний стиль та власний шрифт, висотою 6 пікселів (див. рис. 2.17). Шрифт також включає символи для відображення статів та предметів.



Рис. 2.17. Розроблений шрифт для гри.

Інтерфейс гри складається з декількох вікон, розглянемо кожне з них детально. Список юнітів (див. рис. 2.18) зручно відображає усіх юнітів в команді гравця та дозволяє обирати юнітів.

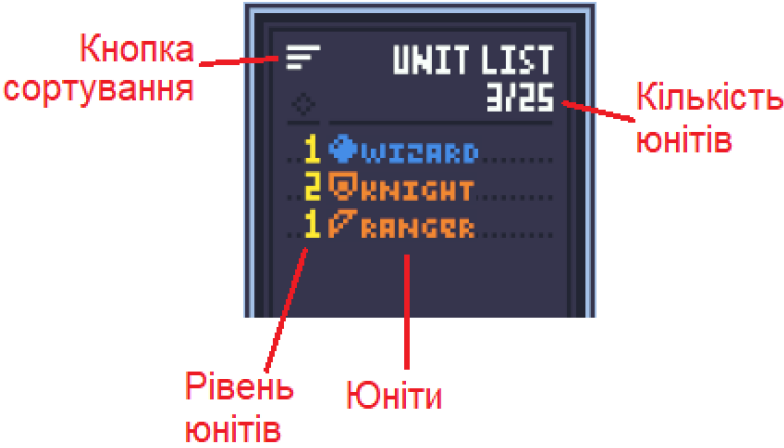


Рис. 2.18. Список юнітів.

Інформація про юніта (див. рис. 2.19) відображається коли обраний лише один юніт. В цьому вікні відображається все, що потрібно знати про юніта, а також можна покращити стати натиснувши на відповідні кнопки. Здібність юніта відображається іконкою і при наведенні з'являється підказка (англ. tooltip), де розписано ефект здібності (див. рис. 2.20).

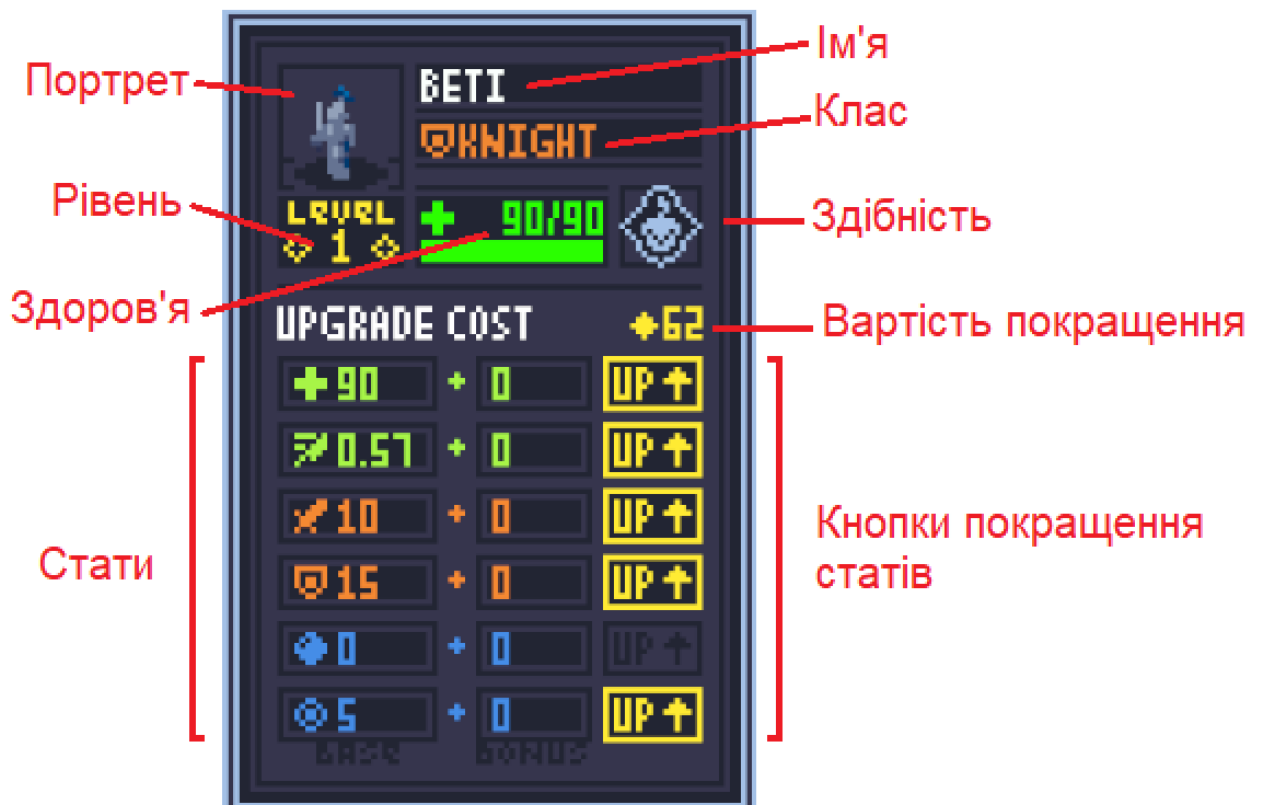


Рис. 2.19. Інформація про юніта.



Рис. 2.20. Ефект здібності

Таким самим чином працює інвентар для предметів - при наведенні на предмет відображається його ефект (див. рис. 2.21). Інвентар вміщає максимум 3 предмета, тому гравцю доведеться ретельно обирати, які предмети брати з собою.

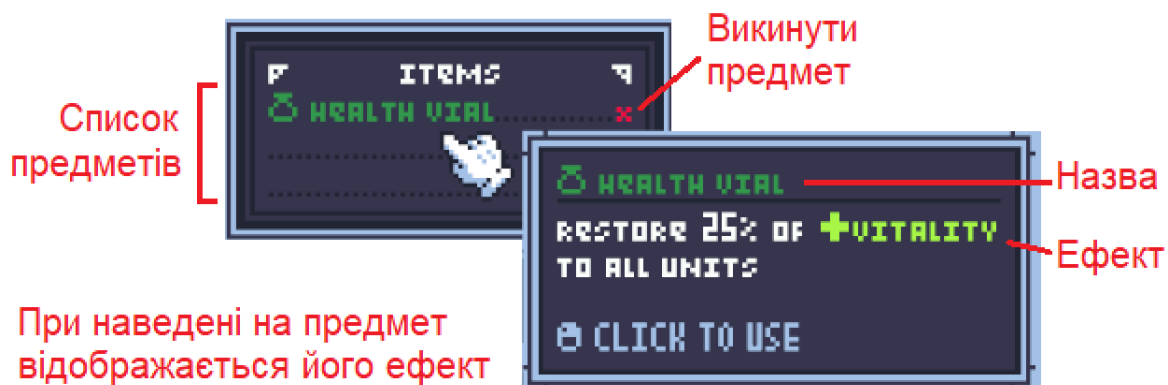


Рис. 2.21. Інвентар для предметів.

Останнім вікном інтерфейсу є мапа підземелля (див. рис. 2.22). Натискаючи на кімнати мапи гравець переміщується у відповідну кімнату. Кнопка спуску на наступний поверх активується коли усі вороги на поверсі переможені і лічильник переможених ворогів відображається біля кнопки.

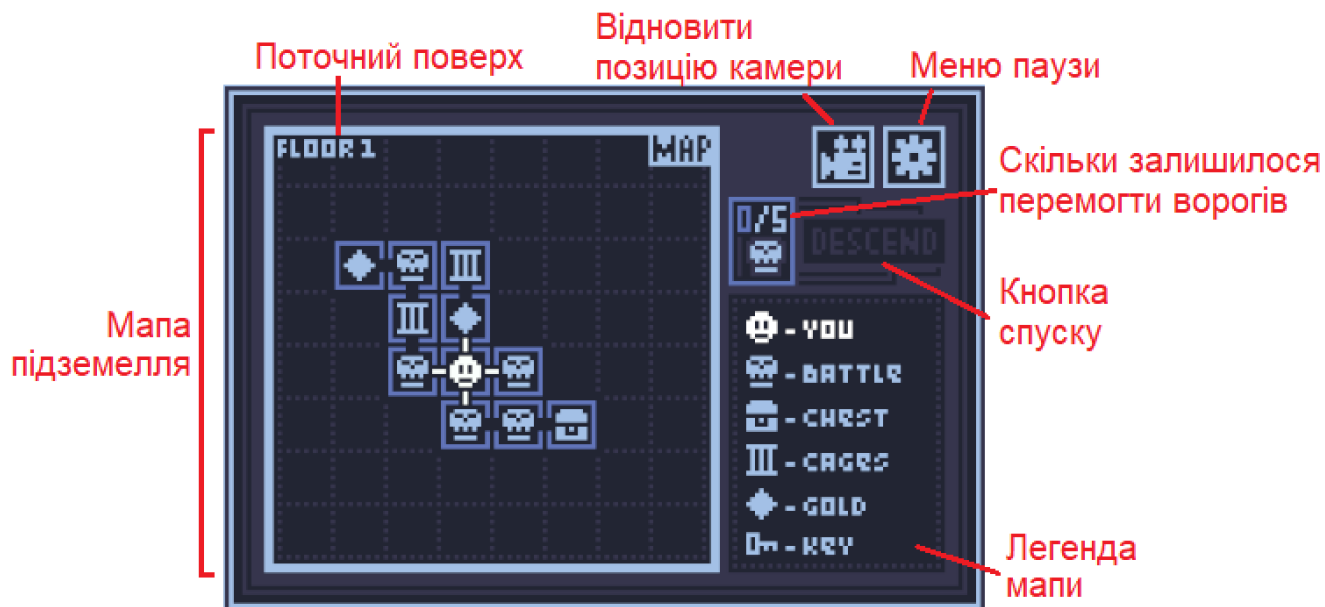


Рис. 2.22. Мапа підземелля.

Поточні золото та ключі відображаються збоку від основного інтерфейсу (див. рис. 2.23).



Рис. 2.23. Золото та ключі.

Змн.	Арк.	№ докум.	Підпис	Дата

Після запуску гри гравець потрапляє в головне меню (див. рис. 2.24), звідки він може розпочати гру або змінити налаштування. На фоні також відображається ігрова дошка та назва гри. В налаштуваннях гравець може змінити відображення гри, використовуючи “PIXELIZATION” та “BACKGROUND” слайдери. Налаштування гучності створене на майбутнє, коли в гру додасться музика та звукові ефекти. Натиснувши на кнопку меню паузи (див. рис. 2.22) відкривається це саме меню.

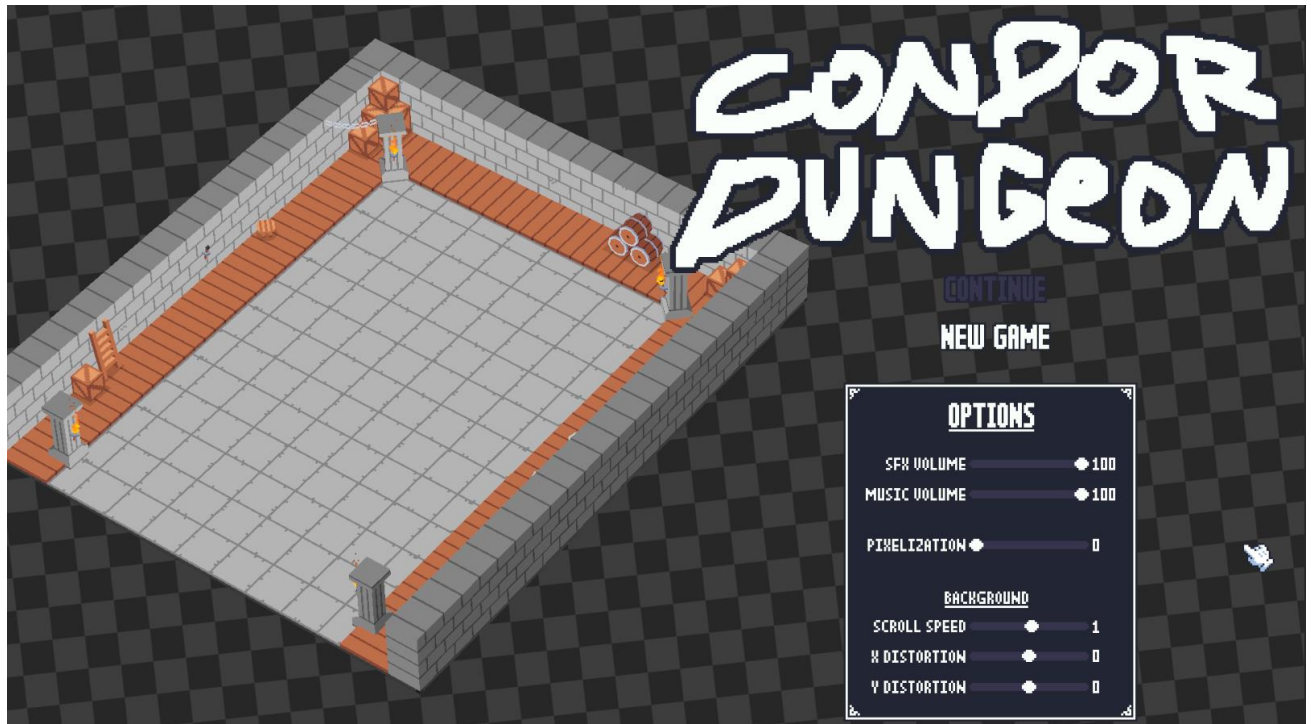


Рис. 2.24. Головне меню

Етап 2 - 3D моделі та анімації

Як вже згадувалося в першому розділі, 3D моделі будуть виконані в стилі Fort Condor - низькополігональні абстрактні моделі, які складаються переважно з пірамід та мають розфарбовані грані. Освітлення в грі буде відсутнє, а тіні будуть імітуватися на самих моделях, що також доповнить ретро стиль. Для анімації створених моделей буде використана кісткова анімація. Вся робота над 3D моделями буде виконана в 3D редакторі Blockbench.

Розглянемо процес створення моделі та анімацій для юніта. Спочатку з простих фігур створюємо саму модель (див. рис. 2.25). Для цього переважно

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

використовується переміщення вертексів (англ. vertex), аби деформувати прості фігури в складніші.

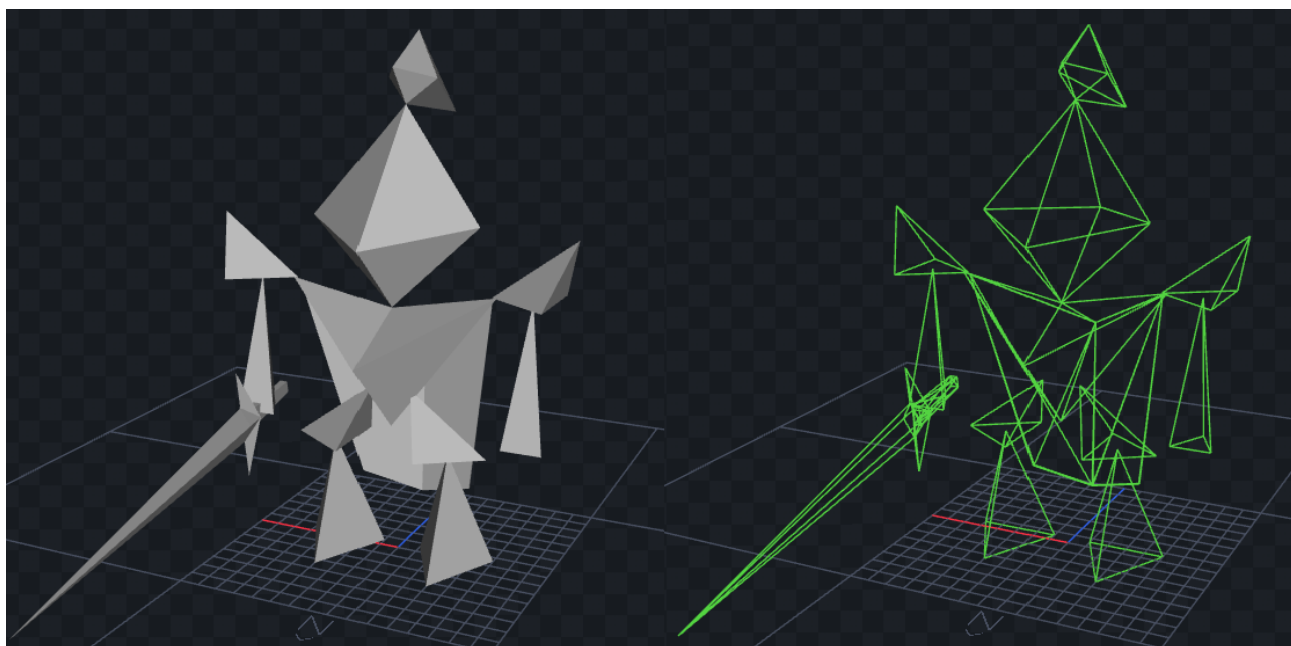


Рис. 2.25. Створена модель

Далі, модель розфарбовується в режимі Paint (див. рис. 2.26). В Blockbench цей процес досить простий, адже можна малювати текстури відразу на моделі.

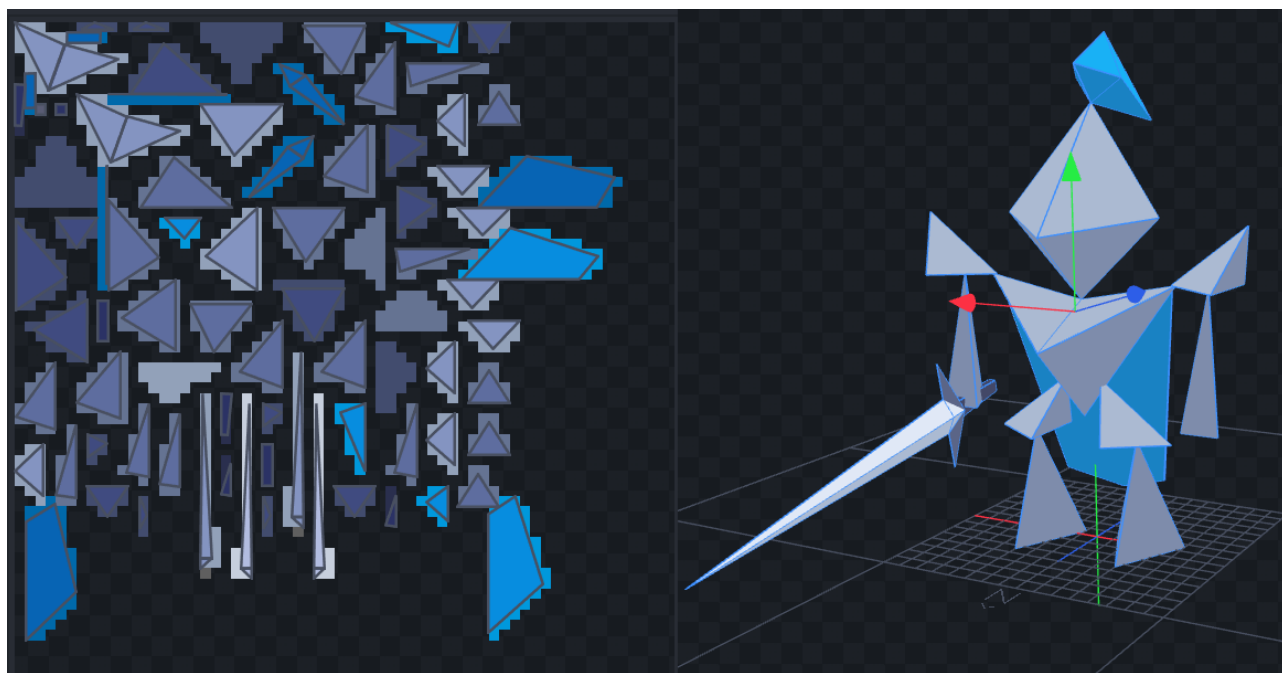


Рис. 2.26. Розфарбована текстура моделі.

Для анімування моделі потрібно поділити її на кістки. В Blockbench кістки - це групи об'єктів, які мають певну точку обертання (англ. pivot). У режимі анімації можна додавати ключові пози для кісток на шкалі часу (англ. timeline), а

Змн.	Арк.	№ докум.	Підпис	Дата

рух між ними відтворюється автоматично (див. рис. 2.27). Кожен юніт та ворог мають 4 анімації: спокій, ходьба, атака та смерть.

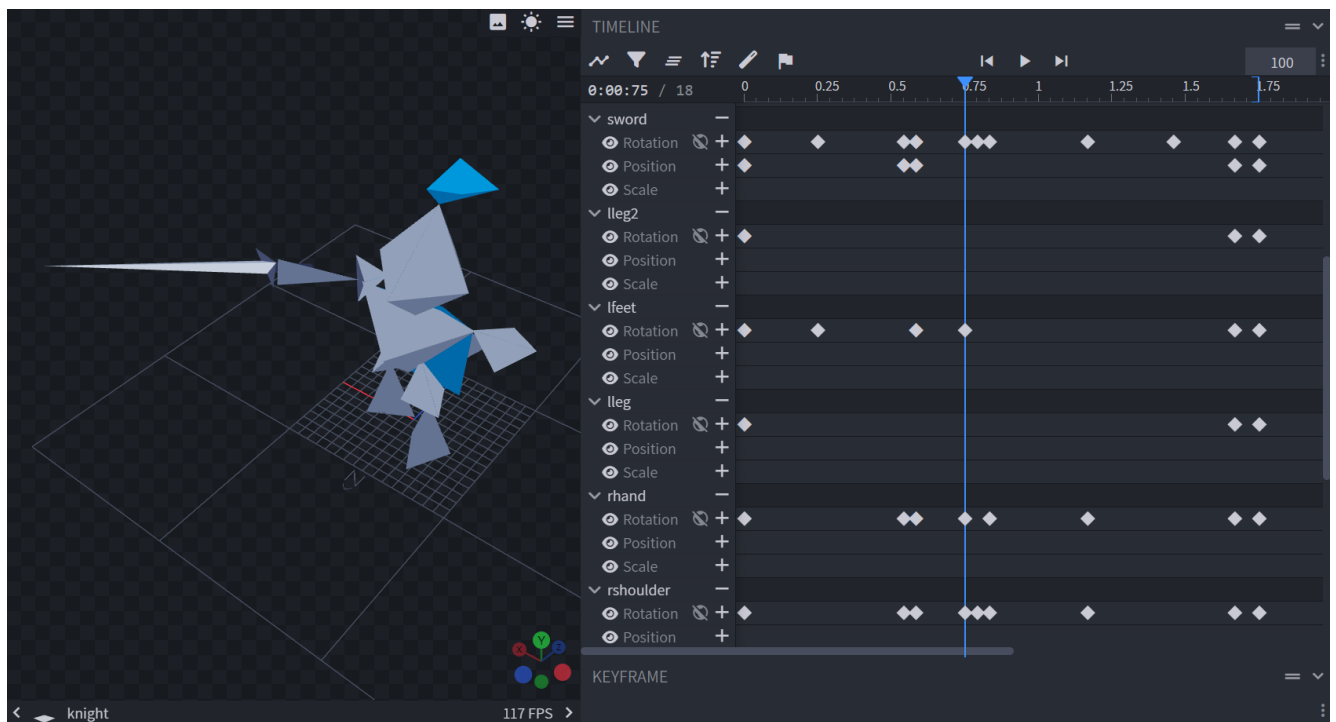


Рис. 2.27. Кісткове анімування моделі.

Імпортування моделей в Unity відбувається у форматі glTF (Graphics Library Transmission Format), який зручно об'єднує модель, текстури та анімації в одному файлі. Для роботи з glTF файлами в Unity потрібен пакунок “Unity glTFast” [9]. Він автоматично конвертує увесь вміст glTF файлу в зрозумілий для Unity формат (див. рис. 2.28).

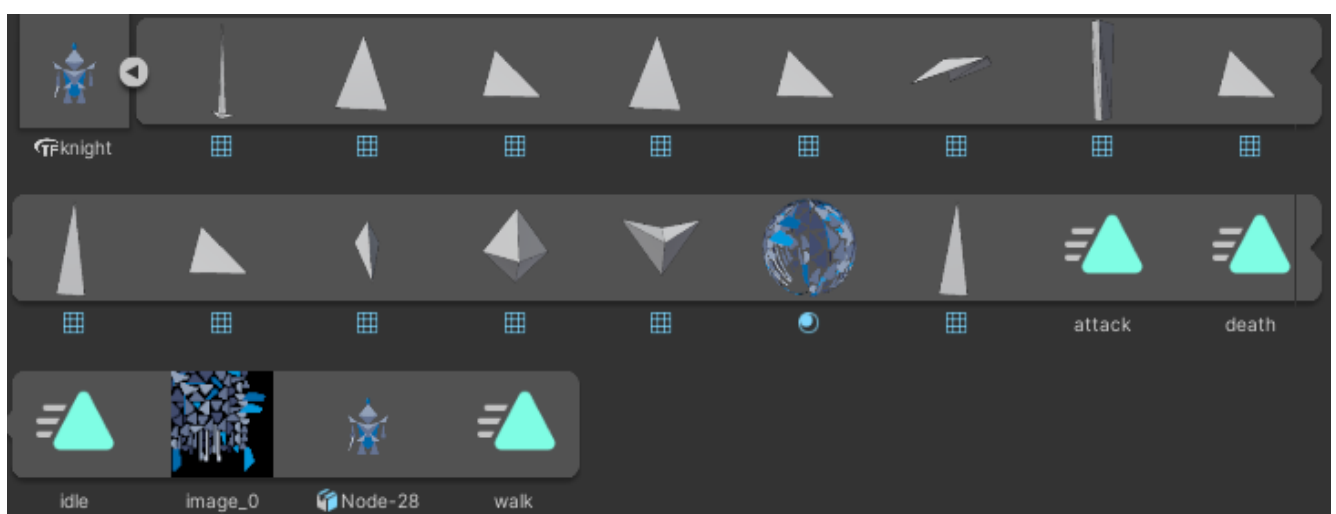
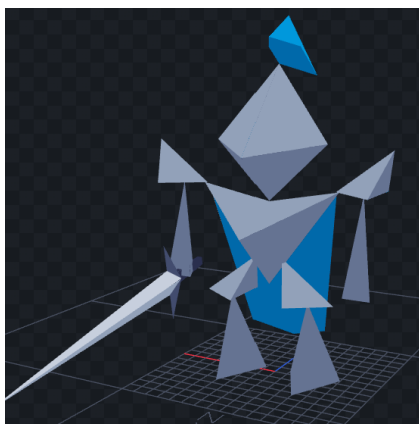
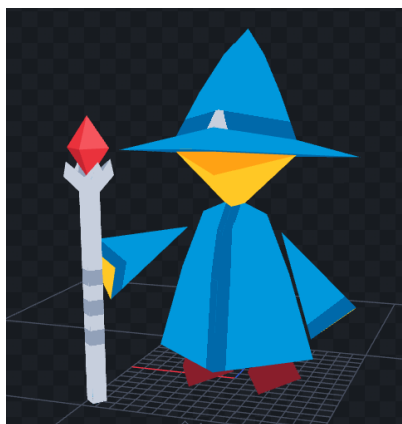


Рис. 2.28. Імпорт моделі в Unity.

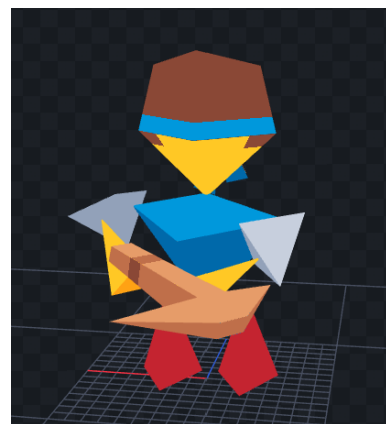
Тепер повторимо цей процес для усіх юнітів та ворогів.



а)

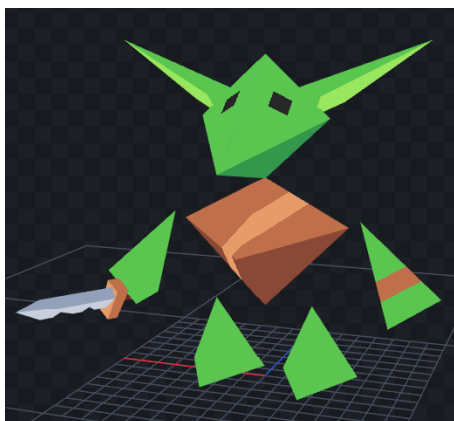


б)



в)

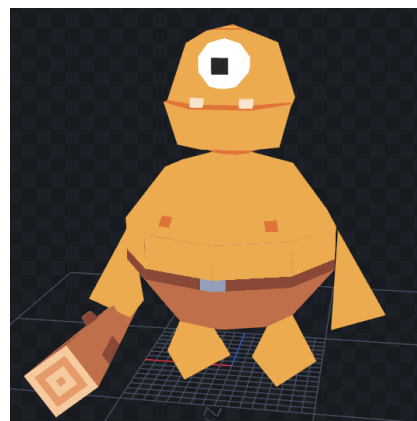
Рис. 2.29. Моделі юнітів: а) Knight; б) Wizard; в) Ranger.



а)



б)



в)

Рис. 2.30. Моделі ворогів: а) Goblin; б) Undead; в) Cyclops.

Ігрова дошка, де будуть розміщатися юніти, буде змодельована схожим чином. Вона має виглядати як кімната підземелля, тому зробимо стіни з цегли, а підлогу з кам'яних плит та дошок (див. рис. 2.31). Використовуючи 3 відтінки сірого створимо ілюзію, що кімната освітлена всередині.

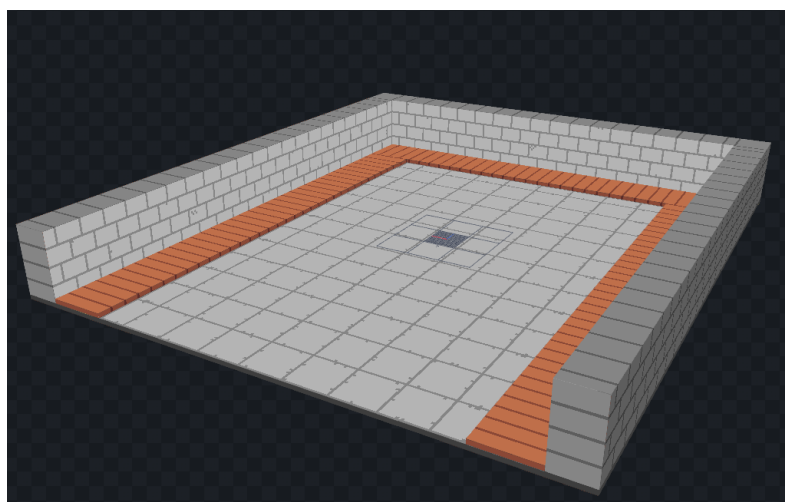


Рис. 2.31. Модель ігрової дошки.

Змн.	Арк.	№ докум.	Підпис	Дата

КН.КР.Б – 122 – 24 – ПЗ

Арк.

51

Щоб кімната не була такою пустою, додамо кілька декорацій (див. рис. 2.32). Юніти матимуть змогу рухатися лише в області з кам'яними плитами, тому можна розмістити усі декорації на дошках, де вони не будуть заважати (див. рис. 2.33).

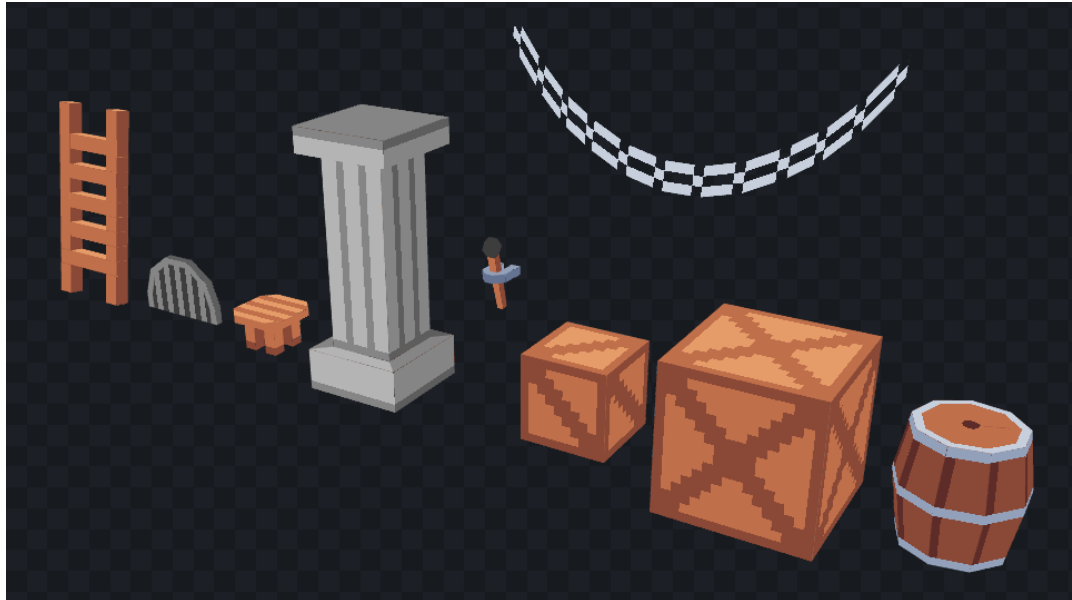


Рис. 2.32. Створені декорації.

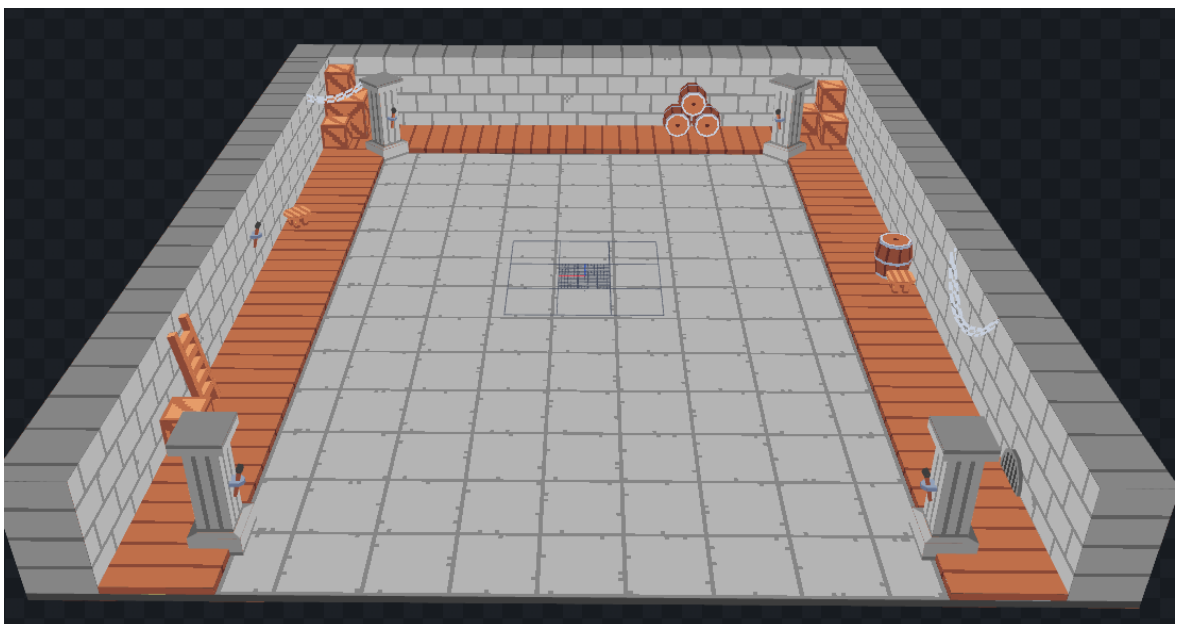


Рис. 2.33. Ігрова дошка наповнена декораціями

Щоб зробити дошку менш статичною, додамо систему партіклів в Unity [10], яка імітує вогонь, до кожного факелу (див. рис. 2.34). Систему партіклів також можна додати Undead ворогу, щоб зробити його дизайн цікавішим (див. рис. 2.35)



Рис. 2.34. Система партіклів для факелів

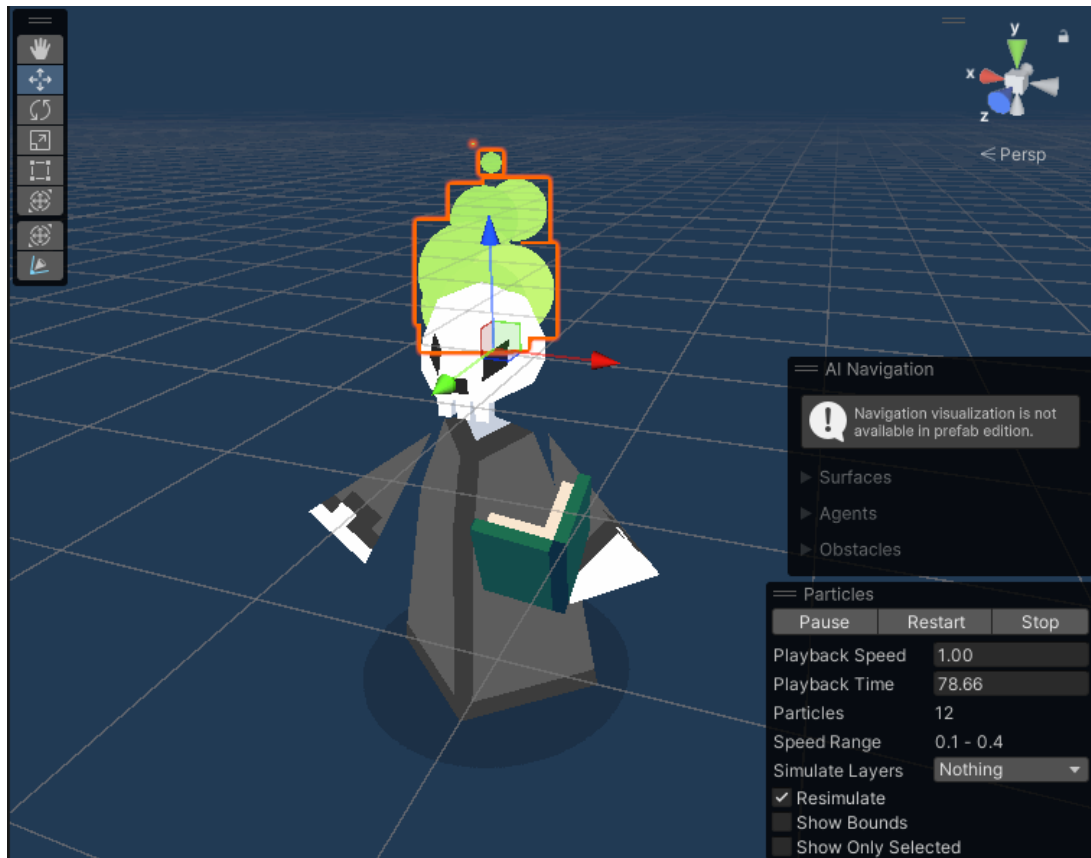
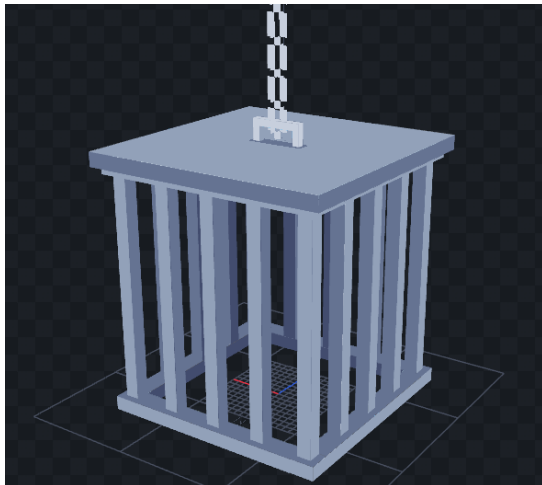


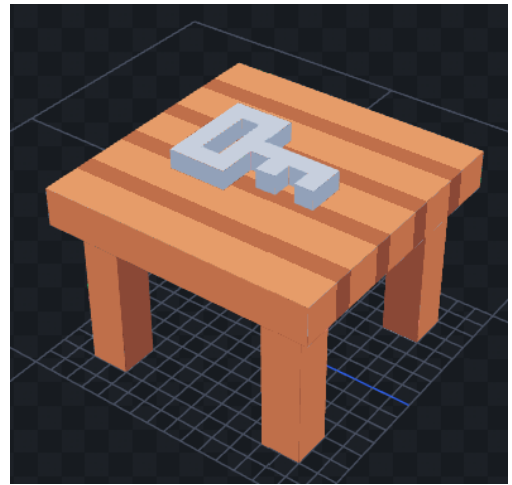
Рис. 2.35. Система партіклів для Undead ворогів.

Змн.	Арк.	№ докум.	Підпис	Дата

Останні 3D моделі, які залишилося створити - це об'єкти з нагородами, які гравець буде знаходити в підземеллі (див рис. 2.36).



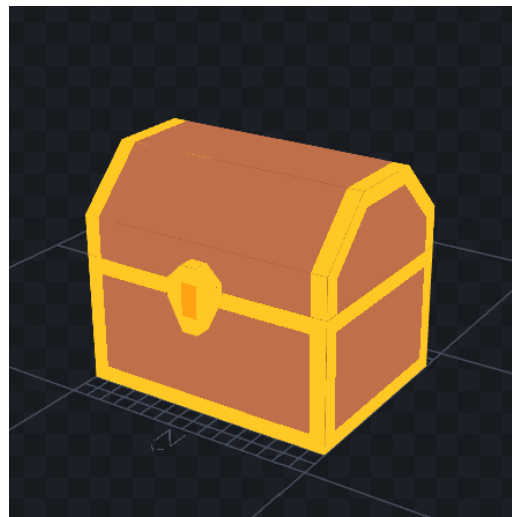
а)



б)



в)



г)

Рис. 2.36. Об'єкти для кімнат підземелля: а) клітка; б) ключ; в) золото; г) скриня.

Висновки до другого розділу

В ході виконання другого розділу було розроблено UML моделі систем та основний функціонал гри, а саме: юніти та керування ними; генерація та наповнення підземелля. Було досліджено та імплементовано патерни та алгоритми, такі як: State machine, Object pool та Wave function collapse.

Крім того, було створено візуальну частину гри, включаючи користувацький інтерфейс, 3D моделі для юнітів і оточення, 3D кісткові анімації та візуальні ефекти.

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

РОЗДІЛ 3 ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО ТЕСТУВАННЯ

3.1 Порядок встановлення та налаштування системи

Гра буде опублікована на вебсайті itch.io. Itch.io - це місце де кожен може безкоштовно публікувати та поширювати свої ігри. Хоча itch.io підтримує продаж ігор, Condor Dungeon буде розміщено в безкоштовному доступі, адже гра наразі в альфа версії. Потрапивши на сторінку гри через жанр або обрані теги, гравець матиме змогу завантажити архів з грою (див. рис. 3.1).

Condor Dungeon

[More information](#) ^

Updated 7 minutes ago

Status [Prototype](#)

Platforms [Windows](#)

Author [macseem](#)

Genre [Role Playing](#)

Tags [Dungeon Crawler](#), [Indie](#), [Low-poly](#), [Roguelite](#), [Tactical](#)

Download

Download

Condor Dungeon.zip 29 MB

Comments

Write your comment...



Рис. 3.1. Сторінка гри на itch.io.

Далі гравець має розпакувати архів в зручне місце на диску та запустити виконуваний файл “Condor Dungeon.exe” (див. рис. 3.2).

Ім'я	Дата змінення	Тип	Розмір
Condor Dungeon_Data	27.05.2024 17:40	Папка файлів	
MonoBleedingEdge	27.05.2024 17:40	Папка файлів	
Condor Dungeon.exe	27.05.2024 17:50	Застосунок	651 КБ
UnityCrashHandler64.exe	27.05.2024 17:40	Застосунок	1 089 КБ
UnityPlayer.dll	27.05.2024 17:40	Розширення заст...	30 196 КБ

Рис. 3.2. Вміст архіву.

Unity робить поширення ігор досить простим, збираючи усе потрібне в один виконуваний файл. Unity також автоматично обмежує максимальну кількість кадрів в секунду (англ. frames per second) до частоти оновлення монітору завдяки технології вертикальної синхронізації (англ. VSync). Це забезпечує плавний хід гри та запобігає використанню більшої кількості ресурсів, ніж потрібно. Налаштування використані для збірки гри зображені на рисунку 3.3.

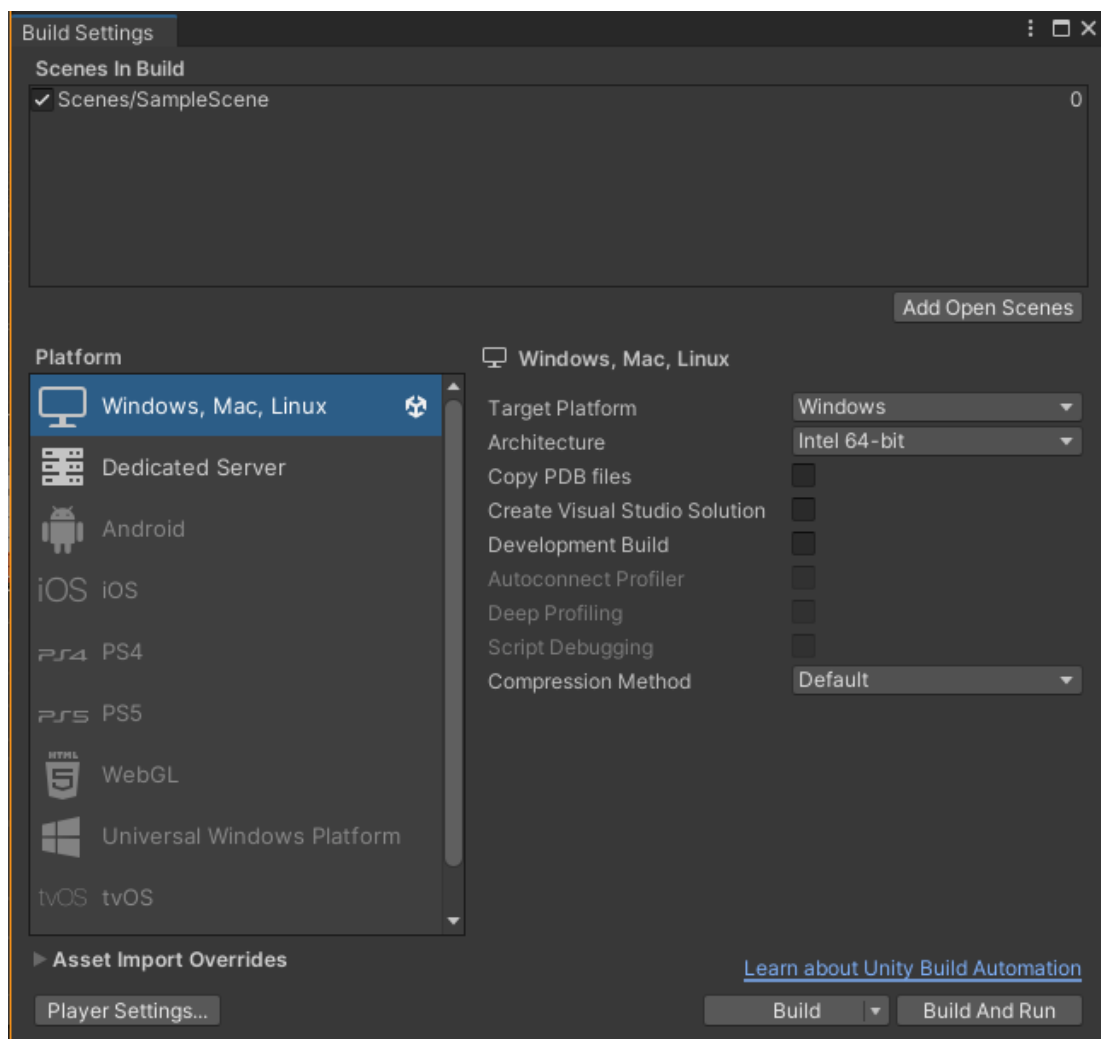


Рис. 3.3. Збірка гри в Unity.

Мінімальні системні вимоги:

1. Операційна система: Windows 7 x64;
2. Процесор: Intel Core i3;
3. Оперативна пам'ять: 1 ГБ;
4. Відеокарта: сумісна з OpenGL 2.1, інтегрована графіка
5. Місце на диску: 100 МБ вільного місця

3.2 Інтерфейс та порядок роботи з програмним застосунком

Коли гравець запускає гру, він потрапляє у головне меню, як зображено на рисунку 2.24. Натиснувши на “NEW GAME” гра розпочнеться та гравця зустріне тьюторіал (див. рис. 3.4). Тьюторіал поступово пояснює гравцю механіки гри та як взаємодіяти з інтерфейсом.



Рис. 3.4. Тьюторіал.

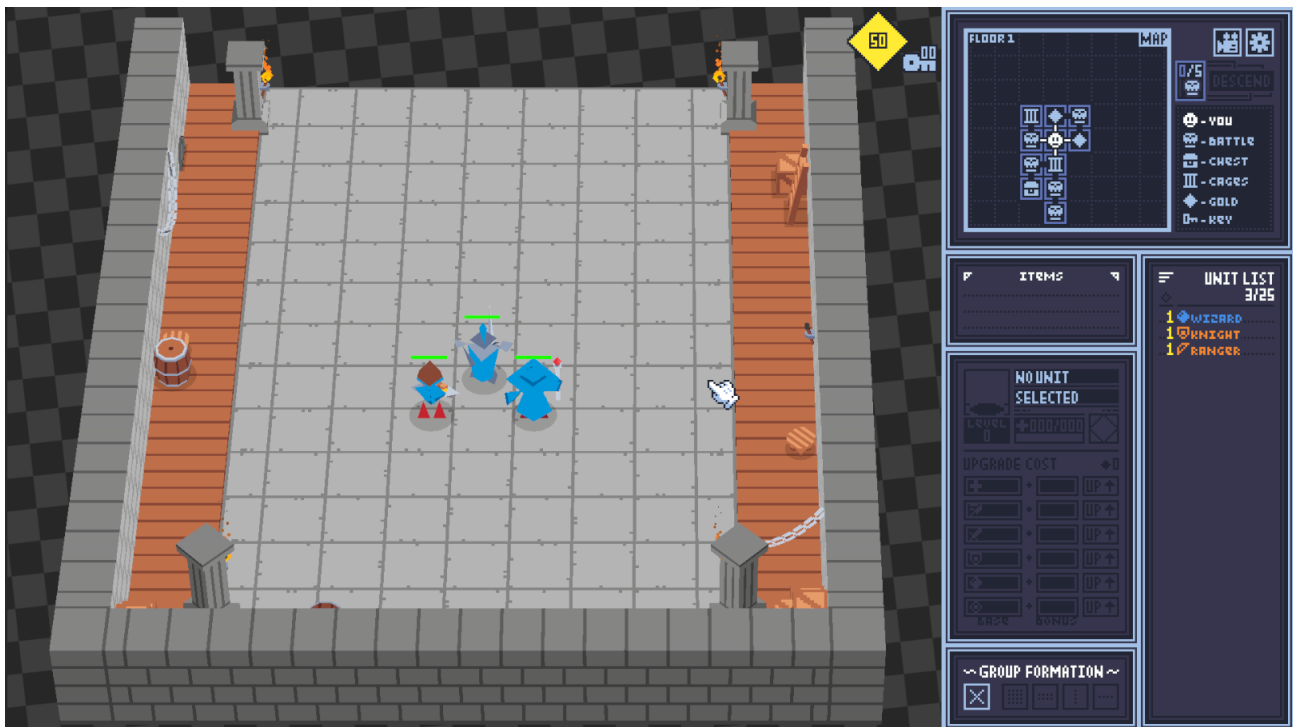


Рис. 3.5. Початок гри.

Завершивши тьюторіал гравець отримує повний контроль над грою та має змогу переміститися у суміжні кімнати (див. рис. 3.5). На початку гравцю дається стандартний набір юнітів - Knight, Ranger та Wizard.

Припустимо, що гравець вирішує зібрати золото, яке знаходиться в сусідніх кімнатах (див. рис. 3.5). Коли гравець увійде в кімнату на дошці з'явиться відповідний об'єкт, натиснувши на який, гравець отримає випадкову кількість золота (див. рис. 3.6).

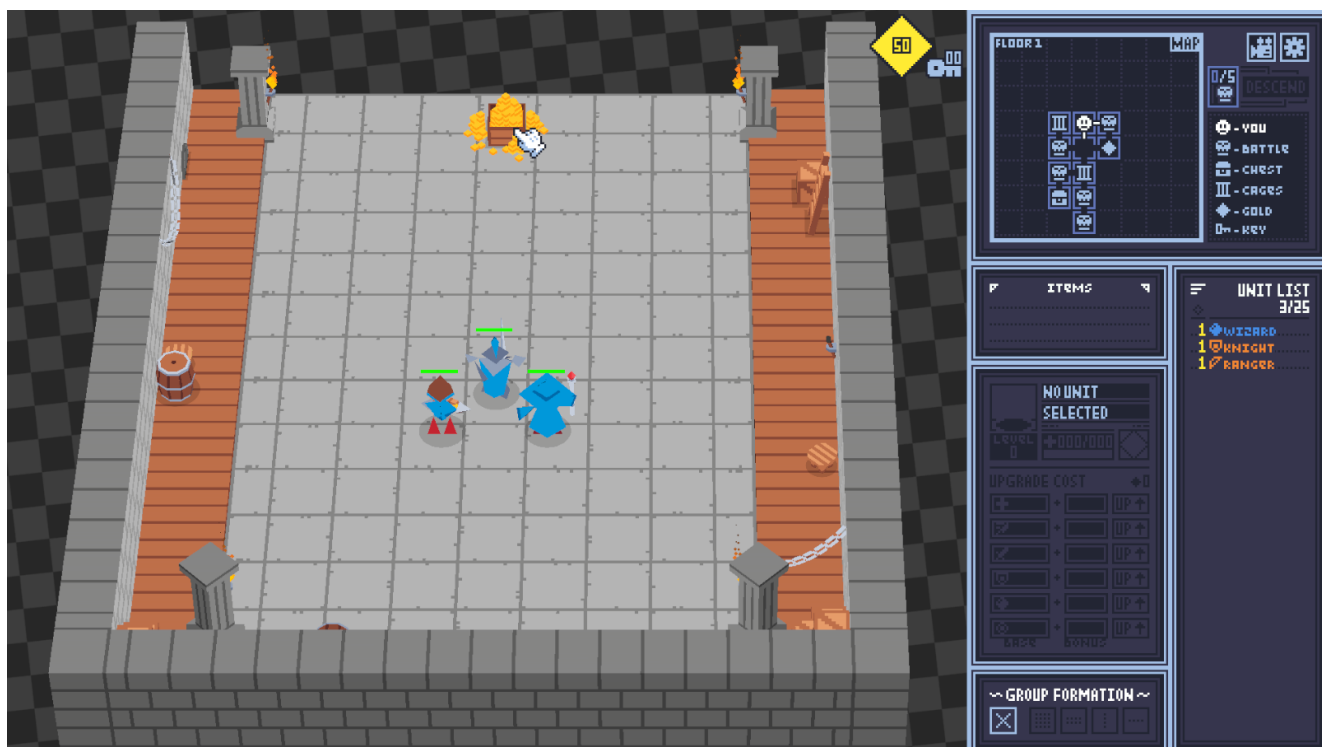


Рис. 3.6. Кімната з золотом.

Використовуючи отримане золото, гравець має змогу покращити своїх юнітів. Кожен наступний рівень коштує дорожче за попередній, тому припустимо, що гравець покращив життєву силу юніту Knight та швидкість атаки юніту Ranger (див. рис. 3.7).

Крім того, гравець має доступ до кліток, де він може отримати нових юнітів. Гравець може навестися курсором на кожну клітку щоб побачити, який юніт в ній знаходиться (див. рис. 3.8). Оскільки в кожній кімнаті з клітками дається безкоштовно 1 ключ, припустимо, що гравець обрав відкрити клітку з юнітом класу Ranger. Далі гравець може увійти тільки в кімнату з битвою, але зібравши золото та додаткового юніта перемога гарантована.



Рис. 3.7. Покращені юніти.



Рис. 3.8. Клітки з юнітами.

Припустимо, що гравець обирає вступити в битву, яка перешкоджає доступ до ще однієї кімнати з клітками, аби було легше впоратися з наступними битвами. Коли гравець входить в кімнату з битвою, мапа блокується сповіщенням про обмеження переміщення по підземеллю. Крім того, з'являється кнопка

“START BATTLE” та червоними стрілочками відмічається звідки будуть виходити вороги (див. рис. 3.9).

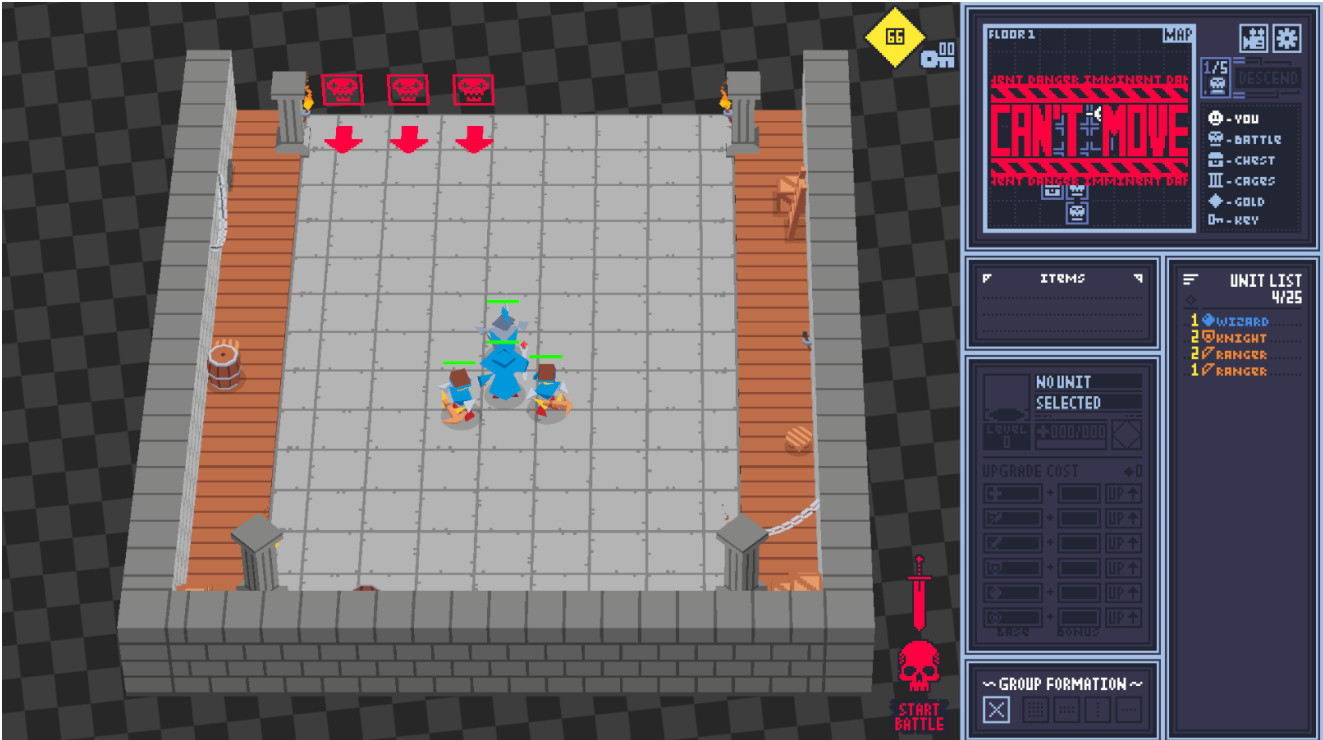


Рис. 3.9. Початок битви.

Розпочавши битву, з вказаних точок починають виходити вороги (див. рис. 3.10). Юніти зазнали значної шкоди, але в результаті перемогли.



Рис 3.10. Битва.

За перемогу гравець отримав 75 золота (див. рис. 3.11), які він також витратить на покращення. Тепер прохід до кліток вільний та гравець додає собі в команду юніта класу Knight.



Рис. 3.11. Нагорода після битви.

Нехай далі гравець вирішує перемогти ворогів, які перешкоджають шляху до скрині з предметом. За перемогу гравець отримує два ключі. Ймовірність отримання одного ключа після битви дорівнює 10%, а двох ключів - 3%. В скрині гравець отримує “Health vial”, який дозволить відновити 25% здоров'я усім юнітам під час битви (див. рис. 3.12).

Витративши золото на покращення та ключі на отримання юнітів, гравець не має більше вибору, окрім як перемогти решту ворогів на поверсі. Остання битва на поверсі - найскладніша, тому довелося використати “Health vial”, аби перемогти усіх ворогів. Коли на поверсі не лишається ворогів, кнопка спуску на наступний поверх стає активною (див. рис. 3.13).

На наступних поверхах гравець повторює ці самі дії: дослідження підземелля, збирання предметів, отримання юнітів, покращення юнітів та битви з ворогами. Всього в грі 3 поверхи і якщо гравець спускається нижче третього поверху, гра закінчується перемогою (див. рис. 3.14).

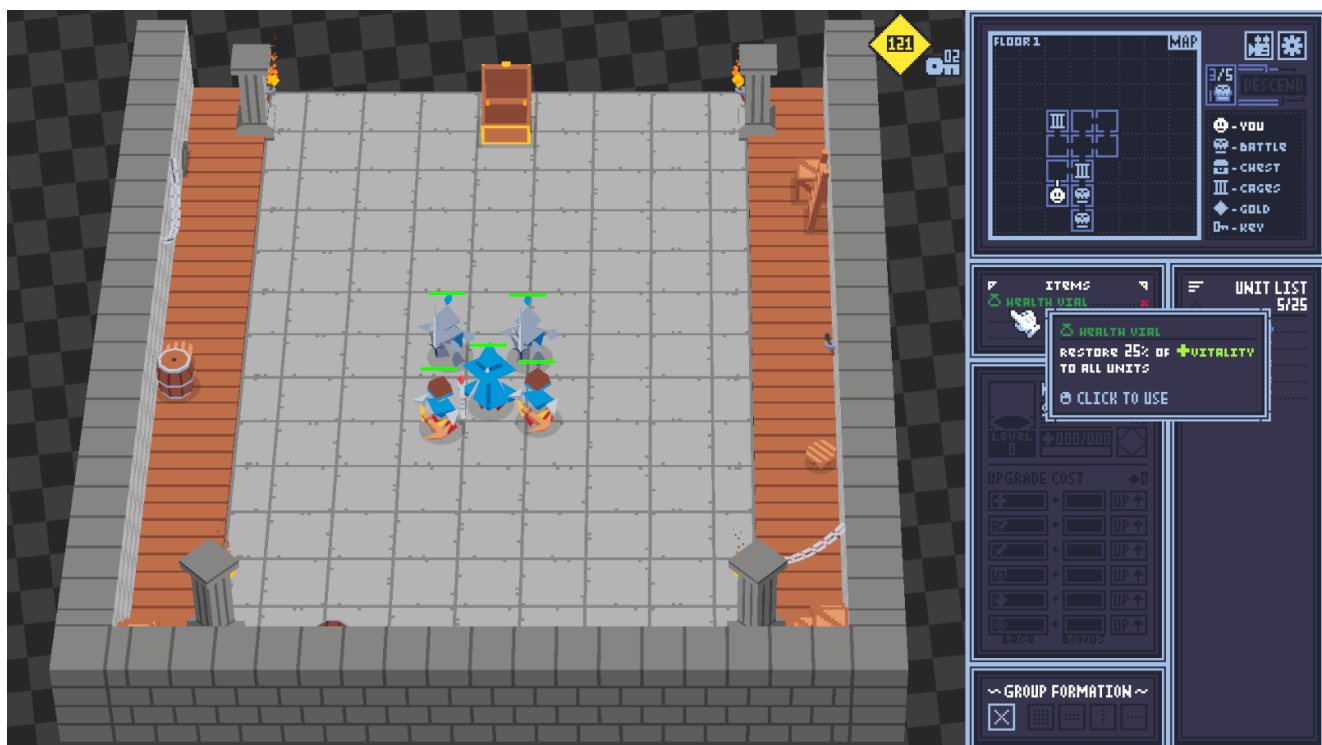


Рис. 3.12. Отриманий предмет.

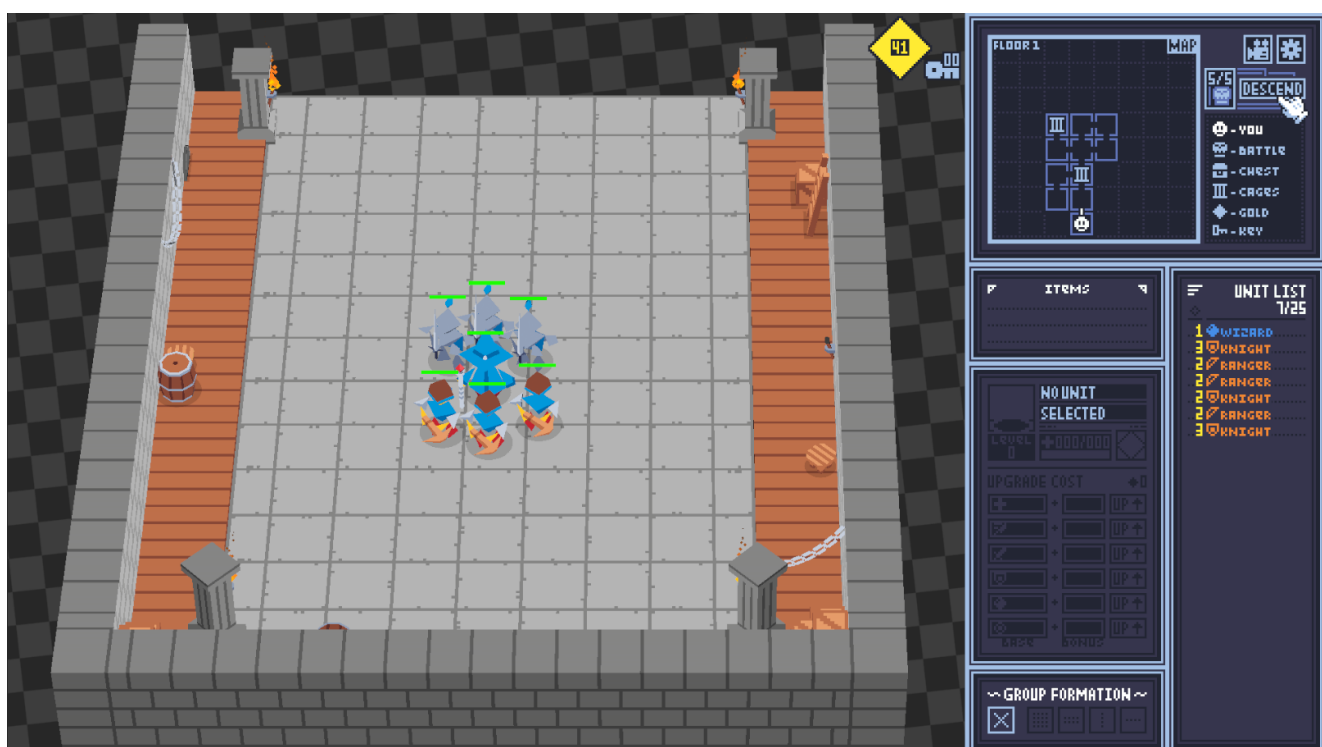


Рис. 3.13. Можливість спуститися нижче.

Якщо усі юніти гравця гинуть в бою - гра закінчується поразкою (див. рис. 3.15) та весь набутий прогрес втрачається. В будь-якому випадку гравець має можливість відразу розпочати нову гру, натиснувши на кнопку в меню. Це саме меню доступне натиснувши на кнопку з шестернею.

Змн.	Арк.	№ докум.	Підпис	Дата



Рис. 3.14. Перемога в грі.



Рис. 3.15. Поразка в грі.

3.3 Тестування роботи програмного забезпечення

Гра має багато функціоналу, який потребує тестування, тому для зручності представимо виконані тести у вигляді таблиці (див. табл. 3.1).

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

Таблиця 3.1

Тест	Очікуваний результат	Отриманий результат
Головне меню	Усі кнопки та слайдери коректно виконують свої функції	Як і очікувалося
Туторіал	Переходи між частинами туторіалу працюють коректно, туторіал можна пропустити, текст туторіалу зрозумілий та не має помилок.	Як і очікувалося
Обирання юнітів	1. При натисканні ЛКМ на юніта, обирається лише цей юніт; 2. При натисканні ЛКМ з зажатою кнопкою “Shift” на юніта, цей юніт додається або прибирається з обраних; 3. При натисканні ЛКМ з зажатою кнопкою “Ctrl” на юніта, обираються усі юніти такого самого класу; 4. При натисканні ЛКМ на землю, всі юніти прибираються з обраних.	Як і очікувалося
Рух та переслідування юнітів	1. Юніти рухаються в позицію відносно натиснутої точки; 2. Юніти не блокують рух один одного; 3. Якщо юніта здвигає з місця інший юніт, він повертається на місце; 4. Юніти коректно переслідують свою ціль та зупиняються, якщо ціль було втрачено.	Як і очікувалося
Атаки, здібності та отримання шкоди	1. Юніти атакують свою ціль, завдають коректну кількість шкоди та змінюють ціль, якщо поточну ціль переможено. 2. Ефекти отримання шкоди працюють коректно, юніти отримують коректну кількість шкоди та вмирають, якщо закінчується здоров’я 3. Здібності юнітів працюють справно. 4. Заклинання магічних юнітів працюють справно.	Як і очікувалося

Генерація підземелля	1. Підземелля генерується коректно; 2. Закриваються відкриті проходи; 3. Кімнати обираються з певним шансом; 4. Кімнати наповнюються нагородами або ворогами; 5. Якщо генерацію неможливо завершити, генерація починається спочатку.	Як і очікувалося
Об'єкти підземелля	1. Гравець отримує коректну кількість золота та ключів з відповідних об'єктів; 2. Юніти в клітках неактивні та активуються після відкриття клітки; 3. Скриня надає предмети, кожен предмет має ймовірність випадіння.	Як і очікувалося
Битви	1. Можливість рухатися по підземеллю деактивована протягом битви; 2. Коректно підбираються вороги для битви; 3. Коректно відображаються місця звідки виходять вороги; 4. Вороги отримують правильну кількість покращень; 5. Битва закінчується коли всіх ворогів переможено; 6. За перемогу надається коректна кількість золота та ключів.	Як і очікувалося
Вікно інформації про юніта	1. Вся інформація відображається коректно; 2. Кнопки покращення працюють коректно; 3. При наведенні на статі відображається інформація про них; 4. При наведенні на здібність відображається її ефект.	Як і очікувалося
Список юнітів	Коректно відображає усіх юнітів команди та їх рівні, дозволяє обирати юнітів так само як в 3D просторі.	Як і очікувалося

Вікно мапи	1. Підземелля відображається коректно; 2. Переміщатися можна лише у суміжні кімнати, які мають прохід; 3. При переході в кімнату активуються її об'єкти та при виході деактивуються; 4. Відображається правильна кількість переможених ворогів; 5. Кнопка спуску стає активною тільки коли було переможено усіх ворогів.	Як і очікувалося
Вікно предметів	1. Предмети додаються до списку; 2. Відображається ефект предмету при наведенні на нього; 3. Предмет викидується, якщо натиснути на хрестик біля нього; 4. Предмети коректно використовуються та активують свій ефект; 5. Предмет не додається, якщо список повний.	Як і очікувалося

Висновки до третього розділу

В ході виконання третього розділу було визначено порядок встановлення та мінімальні системні вимоги для гри. Також, було розглянуто порядок роботи гри, від початку до кінця, та продемонстровано функціонал гри на прикладі уявного гравця. Було успішно проведено тестування всіх аспектів гри та підтверджено, що гра відповідає заданим вимогам.

ВИСНОВКИ

В ході виконання випускної кваліфікаційної роботи було розроблену гру з елементами roguelite, вивчено та впроваджено основні концепції roguelite.

У першому розділі було ознайомлено з поняттями roguelite та auto battler, визначено функціонал та перелік механік для гри, її тематику, візуальний вигляд та стиль, обрано інструменти для розробки та створення графіки.

У другому розділі було здійснено проектування та розробку функціоналу гри, а саме: State machine для керування юнітами, Object pool для повторного використання об'єктів, Wave function collapse для процедурної генерації підземель та інші.

Окрім цього, було створено усі візуальні елементи гри, такі як: 2D спрайти та анімації для інтерфейсу, власний піксельний шрифт, 3D моделі та кісткові анімації для юнітів та оточення, візуальні ефекти.

У третьому розділі було протестовано весь розроблений функціонал, описано порядок роботи з грою та можливі взаємодії в ній. Визначено системні вимоги та порядок встановлення і налаштування гри.

Як результат було отримано базову альфа версію гри, яка відповідає усім заданим вимогам. Функціонал гри розроблявся універсальним та гнучким для забезпечення ефективної подальшої розробки.

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

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The Difference Between Roguelike & Roguelite Games [Електронне джерело] - Режим доступу: <https://www.destructoid.com/the-difference-between-roguelike-and-roguelite-games/>
2. Understanding what auto battler games are - Gaming Street [Електронне джерело] - Режим доступу: <https://gamingstreet.com/auto-battler-games-explained/>
3. Condor War | Final Fantasy Wiki - Fandom [Електронне джерело] - Режим доступу: https://finalfantasy.fandom.com/wiki/Condor_War
4. State · Design Patterns Revisited · Game Programming Patterns [Електронне джерело] - Режим доступу: <https://gameprogrammingpatterns.com/state.html>
5. Unity NavMesh - Unity Learn [Електронне джерело] - Режим доступу: <https://learn.unity.com/tutorial/unity-navmesh>
6. Object Pool · Optimization Patterns · Game Programming Patterns [Електронне джерело] - Режим доступу: <https://gameprogrammingpatterns.com/object-pool.html>
7. Using Generics In C# [Електронне джерело] - Режим доступу: <https://www.c-sharpcorner.com/UploadFile/84c85b/using-generics-with-C-Sharp/>
8. Wave Function Collapse Explained - BorisTheBrave.Com [Електронне джерело] - Режим доступу: <https://www.boristhebrave.com/2020/04/13/wave-function-collapse-explained/>
9. Unity glTFast Documentation [Електронне джерело] - Режим доступу: <https://docs.unity3d.com/Packages/com.unity.cloud.gltfast@5.2/manual/index.html>
10. Introduction to Particle Systems [Електронне джерело] - Режим доступу: <https://learn.unity.com/tutorial/introduction-to-particle-systems#>
11. 6macseem9/fort-condor - Github репозиторій [Електронне джерело] - Режим доступу: <https://github.com/6macseem9/condor-dungeon>

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						68
Змн.	Арк.	№ докум.	Підпис	Дата		

ДОДАТКИ

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						69
Змн.	Арк.	№ докум.	Підпис	Дата		

Класи, що відповідають за стейти юніта

```

public class UnitState : State
{
    protected Unit _unit;
    protected Animator _animator;
    protected NavMeshAgent _nav;
    protected Transform _transform;
    protected Vector3 _lookTarget;

    public UnitState(Unit unit, Animator animator, NavMeshAgent nav)
    {
        _animator = animator;
        _nav = nav;
        _transform = unit.transform;
        _unit = unit;
    }

    public override void FixedUpdate()
    {
        Vector3 lookAtPos = _lookTarget - _transform.position;
        Quaternion newRotation = Quaternion.LookRotation(lookAtPos, _transform.up);
        _transform.rotation = Quaternion.Slerp(_transform.rotation, newRotation, 0.1f);
    }

    public override void OnEnter()
    {
    }

    public override void OnExit()
    {
    }

    public override void Update()
    {
    }
}

public class UnitIdle : UnitState
{
    private SpriteRenderer _actionMarker;
    public UnitIdle(Unit unit, Animator animator, NavMeshAgent nav, SpriteRenderer action-
Marker) : base(unit, animator, nav)
    {
        _actionMarker = actionMarker;
    }
    public override void OnEnter()
    {
        _actionMarker.enabled = false;
        _animator.CrossFade("idle", 0.4f);

        _nav.avoidancePriority = 50;
        _nav.obstacleAvoidanceType = ObstacleAvoidanceType.LowQualityObstacleAvoidance;

        //if (_unit.HoldPosition) _unit.AttackTarget = null; //????
    }
    public override void FixedUpdate()
    {
    }
}

```

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

```

public class UnitMove : UnitState
{
    private SpriteRenderer _actionMarker;
    private bool _startedPath;

    public UnitMove(Unit unit, Animator animator, NavMeshAgent nav, SpriteRenderer action-
Marker) : base(unit, animator, nav)
    {
        _actionMarker = actionMarker;
    }
    public override void OnEnter()
    {
        _animator.Play("walk");

        Util.Delay(0.04f, () => _startedPath = true);

        _nav.avoidancePriority = 49;
        _nav.obstacleAvoidanceType = ObstacleAvoidanceType.NoObstacleAvoidance;
    }
    public override void Update()
    {
        _actionMarker.transform.position = _nav.destination;

        if (_startedPath && _nav.velocity == Vector3.zero)
        {
            CompleteState();
        }
    }
    public override void FixedUpdate()
    {
        if (_nav.steeringTarget == _transform.position) return;
        _lookTarget = _nav.steeringTarget;
        base.FixedUpdate();
    }
    public override void OnExit()
    {
        _startedPath = false;
    }
}

public class UnitChase : UnitState
{
    private SpriteRenderer _actionMarker;
    private Range _attackRange;

    public UnitChase(Unit unit, Animator animator, NavMeshAgent nav, SpriteRenderer action-
Marker, Range attackRange) : base(unit, animator, nav)
    {
        _actionMarker = actionMarker;
        _attackRange = attackRange;
    }
    public override void OnEnter()
    {
        _animator.Play("walk");

        _nav.avoidancePriority = 49;
        _nav.obstacleAvoidanceType = ObstacleAvoidanceType.NoObstacleAvoidance;
    }
    public override void Update()
    {
        if (_unit.AttackTarget == null) return;

        _actionMarker.transform.position = _unit.AttackTarget.transform.position;
    }
}

```

```

        var enemy = _unit.AttackTarget.transform;
        var dir = _transform.position - enemy.position;
        var dest = enemy.position + dir.normalized * (_attackRange.Radius + 0.3f);
        _nav.SetDestination(dest);
    }
    public override void FixedUpdate()
    {
        if (_nav.steeringTarget == _transform.position) return;
        _lookTarget = _nav.steeringTarget;
        base.FixedUpdate();
    }
}

public class UnitAttack : UnitState
{
    private SpriteRenderer _actionMarker;
    public UnitAttack(Unit unit, Animator animator, NavMeshAgent nav, SpriteRenderer action-
Marker) : base(unit, animator, nav)
    {
        _actionMarker = actionMarker;
    }
    public override void OnEnter()
    {
        _animator.CrossFade("attack", 0.4f);

        _nav.SetDestination(_transform.position);
    }
    public override void Update()
    {
        _actionMarker.transform.position = _unit.AttackTarget.transform.position;
    }
    public override void FixedUpdate()
    {
        if (_unit.AttackTarget == null) return;
        _lookTarget = _unit.AttackTarget.transform.position;
        base.FixedUpdate();
    }
}

public class UnitDeath : UnitState
{
    private Collider _collider;
    private GameObject _uiElements;
    private GameObject _attackRange;
    private GameObject _detectRange;
    private Healthbar _healthbar;
    private SpriteRenderer _actionMarker;

    public UnitDeath(Unit unit, Animator animator, NavMeshAgent nav, SpriteRenderer action-
Marker, Collider collider, GameObject uiElements, GameObject attackRange, GameObject de-
tectRange, Healthbar healthbar) : base(unit, animator, nav)
    {
        _collider = collider;
        _uiElements = uiElements;
        _attackRange = attackRange;
        _detectRange = detectRange;
        _healthbar = healthbar;
        _actionMarker = actionMarker;
    }

    public override void OnEnter()
    {

```

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						72
Змн.	Арк.	№ докум.	Підпис	Дата		

```

        _animator.Play("death");

        _collider.enabled = false;
        _attackRange.SetActive(false);
        _detectRange.SetActive(false);
        _healthbar.FadeOut();

        if(_unit.IsEnemy)
        {
            _nav.enabled = false;
            _collider.enabled = false;
            _uiElements.SetActive(false);
            _actionMarker.transform.DOKill();

            Util.Delay(0.05f, () => WaitForAnimFinish());
            return;
        }

        UnitSelectionManager.Instance.UnitDied();
    }
    private void WaitForAnimFinish()
    {
        var animDuration = _animator.GetCurrentAnimatorStateInfo(0).length;
        _transform.DOMoveY(-2, 2).SetEase(Ease.InExpo).SetDelay(animDuration).onComplete = ()
=>
        {
            BattleController.Instance.DecreaseEnemyCount();
            MonoBehaviour.Destroy(_unit.gameObject);
        };
    }
    public override void FixedUpdate()
    {
    }
}

```

Лістинг класу BattleController

```

public class BattleController : MonoBehaviour
{
    public static BattleController Instance;

    [SerializeField] private FloorBattle[] _floorBattles;
    [Space(5)]
    [SerializeField] private Vector3[] _spawnPositions;
    [Space(5)]
    [SerializeField] private CanvasGroup _startButtonGroup;
    [SerializeField] private RectTransform _startButton;
    [SerializeField] private RectTransform _skull;
    [SerializeField] private RectTransform _sword;
    [Space(5)]
    [SerializeField] private BattleIntroAndResults _battleIntro;

    public int CurrentBattle { get; set; }
    public bool InCombat { get; private set; }
    public int MaxFloors { get { return _floorBattles.Length; } }
    private FloorBattle _floor { get {
        return _floorBattles[MapController.Instance.CurrentFloor-1];
    } }

    private Spawner[] _spawners;
    private int _enemyCount;
    private Tweener _loop;

    private void Start()
    {
        _spawners = GetComponentsInChildren<Spawner>();

        ShowSpawners(false);
        _startButton.GetComponentInChildren<Button>().AddPressAnimation();
        _sword.GetComponentInChildren<Button>().AddPressAnimation();
    }

    private void RandomSpawn(Unit unit)
    {
        var spawner = _spawners.RandomChoice();

        var instance = Instantiate(unit, spawner.SpawnPoint, Quaternion.Euler(0,180,0));
        instance.Spawn();

        Util.DelayOneFrame(() =>
        {
            int levels = Random.Range(_floor.EnemyLevels.x-1, _floor.EnemyLevels.y);
            instance.GetComponent<Enemy>().LevelUp(levels);
            instance.FullHeal();
        });
    }

    private void RandomizeSpawnPositions()
    {
        List<int> used = new List<int>();
        int rand;

        foreach (var spawner in _spawners)
        {
            do

```

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

```

        {
            rand = Random.Range(0, _spawnPositions.Length);
        } while (used.Contains(rand));

        spawner.transform.localPosition = _spawnPositions[rand];
        spawner.ShowIcon(true);
        used.Add(rand);
    }
}
private void ShowSpawners(bool show)
{
    foreach (Spawner spawner in _spawners)
    {
        spawner.gameObject.SetActive(show);
        if (show == false) spawner.ShowIcon(true);
    }
}
public void InitializeBattle()
{
    _battleIntro.Intro();

    RandomizeSpawnPositions();
    _startButtonGroup.gameObject.SetActive(true);
    _startButtonGroup.blocksRaycasts = true;
    _startButtonGroup.interactable = true;

    ShowSpawners(true);
}
public void StartBattle()
{
    InCombat = true;
    StartButtonAnimation();
    UnitSelectionManager.Instance.PauseUnitControl(true);

    foreach (var spawner in _spawners)
        spawner.ShowIcon(false);

    var danger = _floor.Battles[CurrentBattle];
    var enemies = ChooseEnemiesForBattle(danger).OrderBy(x=>Random.value).ToList();
    _enemyCount = enemies.Count;

    var interval = _floor.SpawnInterval;
    var skip = _floor.SkipChance;

    var step = -1;
    var skips = 0;
    _loop = Util.Repeat(interval, -1, () => { });
    _loop.onStepComplete = () =>
    {
        if(step!=0 && skips<2 && Random.Range(1,101) <= skip)
        {
            skips++;
            return;
        }
        step++;
        skips = 0;
        RandomSpawn(enemies[step]);

        if(step == enemies.Count-1)
        {
            ShowSpawners(false);
            _loop.Kill();
        }
    };
};

```

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

```

    }
    public List<Unit> ChooseEnemiesForBattle(int danger)
    {
        List<Unit> enemies = new List<Unit>();
        int remainingDanger = danger;

        while (remainingDanger > 0)
        {
            Unit enemy = GetRandomValidMonster(remainingDanger);
            enemies.Add(enemy);
            remainingDanger -= (enemy.Class as EnemyClass).DangerLevel;
        }

        return enemies;
    }

    private Unit GetRandomValidMonster(int dangerLevel)
    {
        List<Unit> validMonsters = _floor.Enemies.Where(x => (x.Class as Enemy-
Class).DangerLevel <= dangerLevel).ToList();
        return validMonsters.RandomChoice();
    }

    #region Victory and UI
    public void DecreaseEnemyCount()
    {
        _enemyCount--;
        if(_enemyCount==0)
        {
            Victory();
        }
    }

    private void Victory()
    {
        InCombat = false;
        MapController.Instance.ClearCurrentRoom();
        CurrentBattle++;

        UnitSelectionManager.Instance.FullHeal();
        UnitSelectionManager.Instance.ReturnUnitsToPositions();

        _battleIntro.Victory(OnRegainControl: ()=>
        {
            MapController.Instance.SetCanMove(true);
            UnitSelectionManager.Instance.PauseUnitControl(false);
            HideStartBattleButton();
            MapController.Instance.UpdateBattleCount(add: 1);
        });
    }

    private void StartButtonAnimation()
    {
        _sword.DOAnchorPosY(38.25f, 0.5f).SetEase(Ease.InBack).onComplete=()=>
        {
            _skull.DOAnchorPosY(3.3f, 0.3f).SetEase(Ease.OutBack,0.8f);
            _startButton.DOAnchorPosY(-24f, 0.3f).SetEase(Ease.OutCirc);
            _sword.DOAnchorPosY(17.35f, 0.3f).SetEase(Ease.OutBack);
        };
    }

    public void HideStartBattleButton()

```

					КН.КР.Б – 122 – 24 – ПЗ	Арк.
						76
Змн.	Арк.	№ докум.	Підпис	Дата		

```

{
    _startButtonGroup.DOFade(0, 0.7f).SetEase(Ease.Flash, 15, 1)
        .onComplete = () =>
        {
            _startButtonGroup.gameObject.SetActive(false);
            _startButtonGroup.DOFade(1, 0);
            _skull.DOAnchorPosY(24.33f, 0);
            _startButton.DOAnchorPosY(0f, 0);
            _sword.DOAnchorPosY(59.25f, 0);
        }
};

}

public (int,int) GetBattleReward()
{
    var danger = _floor.Battles[CurrentBattle-1];
    var tenPercent = (int)(danger * 0.1f);
    var gold = Random.Range(danger - tenPercent, danger + tenPercent);
    var keys = Random.Range(1, 101) <= 10 ? Random.Range(1, 101) <= 30 ? 2 : 1 : 0;

    return (gold, keys);
}

public void ResetEverything()
{
    HideStartBattleButton();
    MapController.Instance.SetCanMove(true);
    UnitSelectionManager.Instance.PauseUnitControl(false);
    InCombat = false;

    ShowSpawners(false);
    _loop.Kill();

    CurrentBattle = 0;
}
#endregion
}
public class FloorBattle
{
    public Unit[] Enemies;
    public Vector2Int EnemyLevels;
    public int[] Battles;
    public int SkipChance;
    public float SpawnInterval;
}

```

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