

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалавр

(ступінь вищої освіти)

на тему РОЗРОБКА ІГРОВОГО ЗАСТОСУНКУ У ЖАНРІ ROGUELIKE
НА ПЛАТФОРМІ UNITY
DEVELOPMENT OF A ROGUELIKE-GENRE GAME APPLICATION ON THE
UNITY PLATFORM

Виконав(ла): студент(ка) 4 курсу, групи КНТ-229

Спеціальності 122 Комп'ютерні науки

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Комп'ютерні науки

СВІТЛАК А.Ю

(ПРИЗВИЩЕ та ініціали)

Керівник ДУБРОВІН В.І.

(ПРИЗВИЩЕ та ініціали)

Рецензент КАЗУРОВА А.Є

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти бакалавр
Спеціальність 122 Комп'ютерні науки
(код і найменування)

Освітня програма (спеціалізація) Комп'ютерні науки
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
“ ” 2023 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

СВИТЛАКА Артема Юрійовича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка ігрового застосунку у жанрі Roguelike на платформі Unity. Development of a Roguelike-Genre Game Application on the Unity Platform

керівник проєкту (роботи) к.т.н., професор, ДУБРОВІН Валерій Іванович,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 04 ” квітня 2023 року № 87

2. Строк подання студентом проєкту (роботи) 01 червня 2023 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Етапи розробки гри. 3. Основі рішення щодо програмної реалізації гри.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів) _____

Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-3 Основна частина	ДУБРОВІН В.І., професор		
Нормоконтроль	ЛИПОВЕЦЬ М.В., асистент		

7. Дата видачі завдання “ 4 ” квітня 2023 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Етапи розробки гри.	2-3 тижні	Розділ 2
4	Основні рішення щодо програмної реалізації гри.	4-5 тижні	Розділ 3
5	Оформлення пояснювальної записки та документів.	6 тиждень	Додатки
6	Нормоконтроль та рецензування.	7 тиждень	
7	Захист роботи.	8 тиждень	

Студент(ка)

(підпис) Артем СВІТЛАК
(Імя ПРИЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Валерій ДУБРОВІН
(Імя ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра:
98 с., 5 табл., 41 рис., 3 дод., 18 джерел.

C#, UNITY, ІНДІ ГРА, СКРИПТ, ІГРОВИЙ РУШІЙ.

Об'єкт дослідження – процес розробки ігрового додатку у жанрі Roguelike.

Предмет дослідження – розробка ігрового застосунку у жанрі Roguelike на платформі Unity.

Мета роботи – розробка ігрового застосунку у жанрі Roguelike на платформі Unity із наступними функціями:

- можливість взаємодії через ігрового персонажа;
- декілька видів меню;
- супротивники із штучним інтелектом;
- взаємодія між сценами;
- випадкова генерація рівнів.

Матеріали, методи та технічні засоби: мова C#, середовище розробки комп'ютерних додатків Unity, середовище розробки Visual Studio.

Результати. Розроблено ігровий додаток у жанрі Roguelike на платформі Unity для комп'ютерній із операційною системою Windows.

Висновки. Розроблено ігровий додаток для розваг.

Галузь використання – комп'ютерні ігри.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 98 pages, 5 tables, 41 figures, 3 appendixes, 18 sources.

C#, UNITY, INDIE GAME, SCRIPT, GAME ENGINE.

The object of research is the process of developing a Roguelike-Genre game application.

The subject of the research is development of a Roguelike-Genre game application on the unity platform.

The purpose of this work is develop a Roguelike-Genre game on the Unity platform with the following features:

- player interaction;
- multiple types of menus;
- AI-controlled opponents;
- scene interactions;
- random level generation.

Materials, methods and technical tools: C# programming language, Unity development environment, Visual Studio development environment.

Results. A Roguelike-Genre game application was developed on the Unity platform for computers.

Conclusions. An entertainment-oriented game application has been developed.

Scope of use – computer games.

ЗМІСТ

	С.
Перелік скорочень та умовних позначок	8
Вступ	9
1 Аналіз предметної області	10
1.1 Актуальність проблеми	10
1.2 Команда розробників	10
1.2.1 Геймдизайнери	11
1.2.2 Художники	12
1.2.3 Програмісти	13
1.2.4 Звукорежисери	15
1.3 Висновки за розділом 1	16
2 Етапи розробки гри	17
2.1 Розробка гри	17
2.2 Концепція	17
2.3 Прототипування	18
2.4 Вертикальний зріз	24
2.5 Виробництво	25
2.6 Тестування	25
2.7 Підготовка до релізу	25
2.8 Оперування	26
2.9 Висновки за розділом 2	26
3 Основні рішення щодо програмної реалізації гри	27
3.1 Вибір мови програмування, ігрового рушія та жанру гри	27
3.2 Модулі Unity та схематична структура гри	32
3.3 Сцена 1	37
3.4 Сцена 2	40
3.5 Сцена 3	44
3.5 Висновки за розділом 3	49
Висновки	51

Перелік джерел посилань.....	52
Додаток А Текст програми	54
Додаток Б Слайди презентації.....	76
Додаток В Посилання на відео з ігровим процесом.....	96

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

DevOps- Development and Operations;

DLC- Downloadable Content;

RnD- Research and Development;

UI- User Interface.

ВСТУП

Комп'ютерні ігри у наш час мають велике значення на суспільство. Створено величезна кількість ігор і разом із цим величезна кількість компаній які або розробляють ці ігри, або створюють додатки які допомагають при розробці.

Розробкою комп'ютерних ігор може займатися як одна людина, так і невелика команда інди розробників або і фірми які за десятиліття створення ігор мають величезній досвід у цій справі. Фірми можуть спеціалізуватись на виробництві ігор для персональних комп'ютерів, ігрових приставок планшетних комп'ютерів. Розробка може фінансуватися і більшою фірмою - видавцем. Фірма-видавець може брати на себе витрати розробки та займається поширенням та продажами ігор. Команда розробників у цьому випадку бере на себе роль тимчасових робітників. Використання фірми-видавця у цьому випадку має великий ризик оскільки видавцю цікаве лише можливість заробити, а також вони мають право втручатись у процес розробки гри. Протилежним підходом є розробка, коли фірма самостійно розробляє і поширює гру. В цьому випадку команда розробників може створювати ігри більших масштабів оскільки у розробку здебільшого не втручаються. Також можлива розробка яку спонсорують самі гравці. Це можливо за допомогою таких сайтів як Kickstarter.

Про складність розробки ігор говорити складно. Є ігри які були створені за рекордно короткий строк і ігри на які були затрачені роки або десятиріччя. Це показує що гра не повинна бути AAA- проєктом щоб сподобатися гравцям.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Актуальність проблеми

Розробка найбільш великобюджетних ігор може коштувати десятки мільйонів доларів США, причому останні два десятиліття бюджети зростають, як і команди розробників і терміни розробки. Так, наприкінці дев'яностих гру для консолі PlayStation для покупця могла зробити команда з десяти осіб на рік, для PlayStation 2 була необхідна команда з 30-50 осіб і два роки розробки, до 2012 року мова йшла про команди з понад 100 розробників і термін близько трьох років.

З ціною зростання розробки щоб гра окупилася виробнику потрібно продати набагато більше копій. Для AAA- проєктів це зазвичай вдається оскільки великий термін розробки гри підігріває зацікавленість якщо час від часу впливає інформація про стадію розробки або можливості гри. Можуть бути і негативні наслідки якщо про гру на всій стадії розробки нічого не відомо. В таких випадках покупець не знає що очікувати від гри і з цим буде чекати час за який розробник може сказати про неуспішність продукції.

Це створює очевидну проблему. Гравець повинен або чекати декілька років на гру і не факто що вона буде цікавою.

Цю ж проблему можуть вирішити інді розробники. Вони можуть створювати ігри з мінімальними можливостями які будуть цікавими для гравців

1.2 Команда розробників

Кількість розробників у команді варіюється від розміру гри. Це може бути як одна людина яка розробляє невеличкий інді проєкт, так і команда із сотні чи більше розробників для великобюджетних AAA ігор.

У команда із великою кількістю людей головний є менеджер проєкту. Менеджер проєкту слідкує за тим, щоб процес розробки гри був протікав

гладко, віхи виконувались, ризики прогнозувалися, а члени команди робили те, що мають. Вони часто є центром спілкування між розробниками, командами дизайнерів та керівниками. Менеджери проєктів виключно організовані та повинні мати відмінні комунікативні навички та навички роботи з людьми.

Розробники діляться на чотири категорії- програмісти, ігрові дизайнери, художники та звукорежисери.

Геймдизайнери мають основну роль в розробці. Вони займаються продумуванням сеттингу гри, його історією. Створюють ескізи для художників. Також створюють правила гри [1]-[2].

Художник створює те, що бачать гравці коли вони грають. Все починається з концепт-арту, перш ніж перейти до контенту, який з'явиться у грі, включаючи тіло, анімацію, тощо. Художник може бути як 2D- чи 3D-орієнтованим. При розробці 3D-гри у команду набирають декілька художників, однак при розробці 2D- гри може вистачити і одного. До художників відносяться: 2Д художник, 3Д художник, аніматор, моушн дизайнер, арт директор, технічний дизайнер, кінематографічний дизайнер, художник по матеріалам.

Програмісти займаються написанням коду, створення ігрового движка або розробкою програмного забезпечення. До програмістів відносяться: програміст геймплею, програміст штучного інтелекту, програміст анімацій, програміст ігрового інтерфейсу, мережевий програміст [2].

1.2.1 Геймдизайнери

До геймдизайнерів відносяться:

- концепт художник- ця людина зарисовує свої ідеї, по ескізам котрих художники створюють персонажів та оточення, а програмісти все це програмують;

- ігровий сценарист- він придумує сюжет, історію, пише сценарій, діалоги, описує персонажів;
- наративний дизайнер- ціль цієї людини створити оповідання через ігровий процес. Це різні щоденники які можна знайти під час гри, записки, аудіокасети, підказки, пасхалки, відсилки чи просто сцени для донесення міркувань;
- дизайнер рівнів- якщо це одиночна гра, то левел дизайнер намагається створити гарне оточення і робить рух по рівню зрозумілим для гравця, там званий візуальна мова, наприклад підсвічування важливих об'єктів. Якщо це мультиплеєрні ігри, то левел дизайнер робить баланс геометрії рівнів, наприклад щоб у однієї команди не було переваги над іншою у вигляді зручної позиції, щіли для прострілювання чи найкоротший шлях до захвату точки;
- системний дизайнер- він працює над ігровим балансом, опрацьовує правила гри. Приклад балансу: танк А має HP-20 та DMG- 50, танк Б має HP- 100 та DMG- 10 [2].

1.2.2 Художники

Художник створює те, що бачать гравці коли вони грають. Все починається з концепт-арту, перш ніж перейти до контенту, який з'явиться у грі, включаючи тіло, анімацію, тощо. Художник може бути як 2D- чи 3D-орієнтованим. При розробці 3D-гри у команду набирають декілька художників, однак при розробці 2D- гри може вистачити і одного. До художників відносяться:

- 2D художник- рисують концепт арти, такі арти пізніше віддаються 3D художникам. Ще створюють іконки інтерфейсу, спрайти, текстури і також можуть працювати із скелетною анімацією яка витіснила кадрову;
- 3D художник- він створює 3D моделі по концепціям, прикріплює до них скелет якщо це персонаж, може настроїти UV сітку, після чого ця модель

переходить до аніматора. 3D художники також можуть поділятися на organic modeling та hard-surface modeling. Organic modeling це персонажі, тварини чи рослини. Hard-surface modeling об'єкти із твердою поверхнею, транспорт, будинки;

- художник по матеріалам- окрім просто розфарбовування персонажів він має це робити гарно. Розуміти фізику світла, де краще накладати затемнення та потертості;

- кінематографічний аніматор- окрім створення руху він створює сцену та розставляє світло, після чого це все рендерить;

- моушн дизайнер- об'єднує у собі художника який все рисує та аніматора. Ще він робить кат сцени чи рекламу;

- технічний дизайнер- це чотири в одному. Ця людина вміє працювати в 2D 3D, все це анімувати. Також він вміє створювати спецефекти;

- арт директор- головний художник, який також слідкує за якістю [2].

1.2.3 Програмісти

Програмісти займаються написанням коду, створення ігрового рушія або розробкою програмного забезпечення. До програмістів відносяться:

- програміст штучного інтелекту- у його задачі входять алгоритми пошуку шляху та робота з деревом поведінки. Зазвичай необхідний інструментарій вже вмонтований в рушій;

- програміст анімацій- вони працюють з inverse kinematic. ІК це тип планування руху. При скелетної анімації в режимі ІК дочірній вузол при руху впливає на всі значення положення і орієнтацію батьківського сегменту. Є ще другий режим- forward kinematic. Тут навпаки, при руху батьківського сегменту, це впливає на дочірні сегменти та вузли. Також програмісти анімацій працюють з animation blending, це змішування різних анімацій;

- UI програміст- програміст користувальницького інтерфейсу. Також можна зустріти аббревіатуру GUI. Художник малює кнопки і іконки, а

програміст надає їм функціональність. Це також реалізується інструментами рушія.

Усіх вищеперелічених програмістів можна віднести до front-end програмістів. До back-end програмістів можна віднести network programmer- вони працюють із серверною частиною, з тим що знаходиться за кулісами. Ці програмісти працюють з мережевим кодом. В мультиплеєрних іграх є два види з'єднання. Dedicated server- він же виділений сервер. Тут сервер запускається на окремій продуктивній машині. Коли один гравець виконує дію, наприклад рухається по карті, його координати відправляються на сервер, там оброблюються і відправляються іншим гравцям. Другий тип це client listen server. Тут у ролі серверу виступає інший гравець. Такого гравця називають хостом. Тобто гравець і грає в гру, і тримає сервер на своєму пристрої. В разі втрати сигналу с хостом відбувається hosting migration. Сервер запускається у іншого гравця.

Ще існують навколоігрові програмісти. Такі програмісти займаються створення ігрового магазину, підбір гравців по рангу і рейтинги.

Мережеві програмісти часто використовують мову програмування Python. Із-за того що він також являється серверною мовою програмування. Також існують:

- Tools програмісти- це ті, хто створює інструментарій. Такі інструменти зменшують час розробки гри;

- DevOps програмісти- це спеціаліст по інформаційно-технічному обслуговуванню. Він синхронізує всі етапи створення гри. Загружає гру в магазини. Якщо гра мережева, то знаходить хостинг, все це налаштовує. Також пише лаунчер якщо він потрібен і працює з оновленнями;

- RnD програміст- такі люди які досліджують все що відбувається навкруги і намагаються створити те, що ніхто не створював. Зазвичай вимоги до таких людей високі, аж до магістратури;

- програмісти рушіїв- крупні студії віддають перевагу використуванню власного рушія. Подібно Frostbite чи id Tech. Їх код

прихований від громадськості. Такі рушії називаються пропрієтарними. Ну а для інді розробників доступні open source рушії такі як Unity, Godot чи Unreal engine;

- спеціалісти по рендеру- вони пишуть шейдери. Відблиски, тіні, випуклості, це все шейдери.

Fullstack розробник має працювати як із візуальною частиною, так із серверною. Частіше за все цей термін використовується у веб розробці.

В середній компанії на одного програміста приходить три-чотири художника [2].

1.2.4 Звукорежисери

Також для гри важливе звукове супроводження оскільки це є важливим елементом для придання атмосфери грі. Це може базуватися на звуках, діалогах та піснях для гри. До звукорежисерах відносяться:

- саунд дизайнер- актори дубляжа, люди які проговорюють репліки та діалоги прописані сценаристом;

- саунд дизайнер- його професія перетинається із звуковим інженером. В геймдеві саунд дизайнер і звукорежисер це одне і теж. В його обов'язки входить пошук звукових ефектів, або він також може синтезувати свої за допомогою спеціальної програми, чи записати. Наприклад, якщо команді потрібні звуки пострілу, то команда виїжджає на полігон із записуючим обладнанням;

- аудіо програміст- пише скрипт який виконує такі функції як гучність звуку, накладання ефектів та робить можливим запуск таких ефектів як постріл чи рух. Також може створювати звукові інструменти для саунд дизайнера. Він несе відповідальність за тестування та налагодження аудіо коду;

- композитор- в його задачі входить створення музики чи саундтреку до гри. Це завжди позаштатний співробітник. В залежності від розміру

проекту у звуковий відділ ще можуть входити продюсери і художники по спец ефектам [2].

1.3 Висновки за розділом 1

Команда розробників може складатися як із одного або декількох людей для інди гри, так і з 50 чи більше. Для найбільш великобюджетних ігор кількість розробників може доходити до декількох сотень. Також можуть найматися зовнішні працівники для аутсорсингу.

Команда розробників складається із геймдизайнерів, програмістів, художників і звукорежисерів. Керує ними менеджер проєкту.

2 ЕТАПИ РОЗРОБКИ ГРИ

2.1 Розробка гри

Розробка гри являється багатоетапним процесом. Цими етапами являються концепція, прототипування, вертикальний зріз, створення контенту, тестування, підготовка до релізу і оперування [3].

2.2 Концепція

На першому етапі команда на чолі з геймдизайнером збираються разом та проводять брейншторм. Розробники придумують гру.

Спочатку позначають жанр, він стане головним вектором в розробці. Вибирається сеттинг гри, коли і де буде відбуватися гра. Обговорюється художнє направлення гри, її стиль графіки. Всі фінальні думки заносяться в три документи.

Перший - концепт документ. Це коротке та ємне описування гри. Стандартні пункти цього документи такі: назва гри, основна ідея, жанр, аудиторія, особливості гри, сюжет гри, механіки, описування ігрового світу та персонажів, на які платформа гра орієнтована, системні вимоги, строки та бюджет розробки та які технології будуть використовуватися.

Другий - vision документ. В цьому документі описується не сам ігровий процес, а то, що хочеться отримати в підсумку, як кінцевий бізнес продукт. В цей документ записується: повний опис гри, цільова аудиторія, дослідження ринку, конкуренти, стиль та арти, бізнес модель, як буде монетизуватись гра. Цей документ розрахований більше для розробників.

Третій - feature list. Список особливостей: на які деталі буде більше акцентуватись, на графіку, унікальні механіки чи щось ще. Також сюди входить унікальність, унікальні елементи гри які будуть виділять її на фоні конкурентів, приваблювати гравців та збільшувати продажі.

Ці три документа взаємозв'язані між собою і формують загальний

дизайн документа- dice doc. До dice doc може в ходити описування штучного інтелекту, інтерфейсу, звукового оточення та рівнів.

Дизайн документ створюють геймдизайнери і зазвичай це не документи, а модульні схеми побудовані на вікі рушії. Прописуються технічні завдання.

Є ще pitch документ, це стисла версія концепт документа із списку фіч, сценарію, історії гри. Робота над цим може початися не одразу.

Одразу на ранній стадії розробки вибирається ігровий рушій. Це може бути Unity, Unreal Engine, Godot, Cry Engine. Як правило, у великих студіях вже є власна пропрієтарний рушій, а якщо немає пишуть з нуля, але зараз так ніхто не робить.

Також збирається референції, фотографії із реального життя, ігор, фільмів чи кіно. Ці концепти представляють продюсерам і інвесторам. Далі з референції концепт художнику стає зрозуміло що від нього потребується. І ось, сформувавши достатньо інформації концепт художники приступають к створенню ескізів і ілюстрацій [3].

2.3 Прототипування

Ось і розпочинається розробка. Звіряючись з раніше створеними документами, тези та концептами розробники приступають до прототипуванню гри.

Художники продовжують працювати над концептами і потім поступово розпочинають створювати ігрові об'єкти. Це персонажі, транспорт, рослини, будинки, невеликі проб об'єкти та спецефекти.

Тим часом дизайнери рівнів починають виконувати свою роботу-створюють рівні.

Концепт арти передаються 3д художникам і вони починають свою роботу. 3д моделювання можна поділити на два основних напрямки: hard surface та organic surface.

Hard surface це об'єкти з твердою поверхнею. Об'єкт для наповнення світу, бочки, пляшки, коробки і інший непотріб. Також будинки, їх можуть створювати із окремих модулів, як конструктор. Якщо це дуже складні об'єкти, то прибігають к процедурному моделюванню. Використовуються програми на кшталт гудини. Задається що, де і скільки генерувати. Також транспорт, складається він із 3д моделі до котрої причіпляється скелет. Цей skeletal mesh загружається в ігровий рушій і вже там перетворюється а автомобіль. Зазвичай для цього є спеціальний інструментарій який дозволяє налаштувати керування, фізику, налаштувати підвіску і інші характеристики транспорту. Літаючий транспорт створюється за таким же принципом. Пошкодження на транспорті буває статичними, заздалегідь створеними, чи адаптивними. Тут частини кузова не замінюються на пошкодженні, а деформуються по полігонам, реагуючи на колізію іншого об'єкта.

Organic surface це не рукотворні об'єкти, тобто природа. Деревя можуть генеруватися в такій програмі як Speedtree. Також к органіці відносяться істоти і персонажі.

Також ігровий контент може закупатись на стоках. Є чудова бібліотека- Megascans. Ще можна відсканувати об'єкт за допомогою оптичних чи лазерних сканерів.

Для створення персонажа спершу вибирається яким буде він: тварина, людина чи монстр. Далі ці референси збираються в мудборди. Це композиція із референсів на одну тематику. Художник рисує концепт, передає 3д модельєру. Далі модельєр за допомогою скульптингу створює highpoly модель персонажа використовуючи програму Zbrush. Скульптинг це як цифрова ліпка із пластиліну. Щоб моделювати було легше скульптинг ділять на 3 або 4 рідів. Перший рід- створення болванки персонажа, начерку який називається блокаут. Просто кажучи анатомії та силуету. Також це може називатись драфт. Другий рід- внутрішнє наповнення та великі деталі. 3 рід- маленькі деталі та нюанси анатомії. І ще може бути четвертий рід- наддеталізація. Це можуть бути пори чи морщини. Далі модельєр робить

ретопологію сітки. В підсумку із highpoly виходить lowpoly модель. Вона не така вимоглива до продуктивності. Кількість полігонів на highpoly може обчислюватись мільйонами. Після ретопології кількість полігонів зменшується до 20000. Середнє обмеження на проєкт 20000-80000. В підсумку створюється дві моделі- highpoly та lowpoly. Highpoly можна використовувати для прирендерів, запікання та cinematic роликів, а lowpoly йде в саму гру. Для ретопології підійде будь-який 3D пакет: Blender, Maya, 3D Max.

Після цього створюється UV карта для майбутніх текстур. Програмам для текстурування та ігровим рушіям потрібна розгортка. Це координатна сітка яка вказує де яка текстура на моделі буде відображатись. 3D модель розвертається у площину. Розгортку можна робити як вручну, так і автоматично.

Наступний етап- запікання. На цьому етапі деталізацію highpoly моделі можна запекти в карту нормалей. Це така текстура об'єму, також її називають бамп мап. це плоска ілюзія об'єму, як 3D відкритка. Також можна запекти ambient occlusion. Це затінення у стиках отримане від м'якого глобального освітлення. Легше говорячи запікання- це перенос візуальної інформації із highpoly до lowpoly у виді текстур.

Після створення персонажа його необхідно розфарбувати. Це може зробити як сам 3D модельєр так і художник. Зазвичай для цього використовують Substance painter. Усі відблиски, потертості це різні текстури котрі називають мапами. Нормалі також можна створювати на етапі текстурування. Ці мапи об'єднуються в загальний матеріал. Надалі цей матеріал буде накладений на UV карту.

Наступний етап- підготовка до анімації. Для цього персонажа переносять у інший 3D пакет. Це може бути 3ds Max чи Maya. Для майбутньої скелетної анімації розпочинається процес створення скелету- ріггинг. В одному персонажі в залежності від складності може бути від 20 до 100 кісток і навіть більше, якщо потрібна правдоподібна лицева анімація. Далі готовий скелет потрібно прикріпити до тулуба, цей процес називається скіннінг.

Після цього персонаж переходить в руки аніматора. Анімація розпочинається із розкадровки. Використовуються анімації по ключовим кадрам: перший це початкове положення, а другий це кінцеве. Кадри між ними автоматично плавно підстроюються. Таким чином складно створити анімацію людини тому для цього використовується спеціальний костюм motion capture. Це костюм на котрому розташовується велика кількість датчиків і далі рух цих датчиків фіксується камерами. Чотири основні анімації які створюють аніматори: бездіяльність, хода, атака та смерть. Щодо анімації лица, то існують різні способи її створення. Один із них це ріг на основі морфів, це ще називається blend shape. Створюється кілька копій однієї голови і далі на кожній змінюється емоція. Наприклад у одній емоції відкритий рот а у іншій закритий. Таким чином це дві ключові емоції і геометрія на проміжних кадрах автоматично підстроюється за допомогою математичної інтерполяції. Тут мінус у тому, що даний тип ріггінгу не можна використовувати на інших персонажах. Набір таких копій ключових лиць називають ключовими позами. Інший метод, це ріггінг на основі кісток. Його можна використовувати на великій кількості персонажів. Створення того, що може управляти анімаціями називається ріггінгом. Далі для зручності можна додати контролери. У кожній частини лица є свої унікальні анімації і щоб створити унікальну потрібно лише скомбінувати потрібні варіанти. Це лише способи анімації, однак потрібні і самі анімації, тобто дані. Це можна зробити вручну. Інший спосіб це маркерна система захвату лицевої анімації- performance capture.

У завершені готового персонажа імпортують у рушій і там вже розпочинається робота по його настройці. Потрібно прив'язати анімації к діям, прив'язати персонажа до штучного інтелекту, додати діалоги.

Всі ці етапи створення персонажа виконуються послідовно. В деяких студіях цей pipeline може бути трохи змінений.

Етапи створення ігрового світу. Перший це підбір референсів. Це фотографії для натхнення, або команда може виїхати в потрібну локацію і робить знімки. Більшість рівнів розпочинається з прототипу нарисованому на

папері. Складається список локацій та відмічаються івенти, де скільки ворогів буде розташовуватись, які події прив'язані до точки та інше.

Для ігор з одиночної компанією може бути три типи схем створення рівнів. Перший- лінійний, тут гравець рухається із точки А в точку Б. Другий- кластерний, багато відгалужень від центральних хабів. Третій- павутина, багато сплєтених між собою шляхів, це дає свободу гравцю. В мультиплеєрних іграх дизайнер працює над балансом рівня виключаючи перевагу однієї команди над іншою.

Після прототипування на папері дизайнер разом із геймдизайнером формує метрики, це шаблони розмірності для проектування рівнів.

Визначившись з метриками дизайнер рівнів починає створювати блокаuti, заодно розставляє просте освітлення. Блокаут- це скелет карти що складається із примітивної геометрії. Такий блокаут називається грей бокс чи дизайнер блокаут. На даному макеті перевіряється ігровий процес без візуальних складових. Далі ці примітиви будуть замінені на гарні 3д моделі. Якщо потрібен ландшафт, то він також може бути створений завдяки інструментам всередині рушія. Чи його можна імпортувати із інших пакетів. Далі на рівень додаються тригери, це невидимі зони які активують події. Також додається середовище в якій може орієнтуватися штучний інтелект. Додаються точки появи персонажів, спавнери. Однак більш точне налаштування робиться лише на фінальній стадії

Після компоновання прототипу рівня, грей бокса, проводиться тест ігрового процесу і виправляються недоліки.

Друга стадія прототипування рівня це вайт бокс. Він же арт блокаут. В ньому художники проробляють художню частину рівня. Роблять начерки. На цьому етапі підключаються художники по оточенню та концепт художники. Художники по оточенню працюють над візуальною частиною рівня. Вони заповнюють ігровий світ тимчасовими моделями placeholder. На рівень додається ігрова логіка. Художники по оточенню можуть використовувати kit, набір асетів.

Далі наступає третя, фінальна стадія завершення. Placeholder замінюють на фінальні 3д моделі. Художник по оточенню створює середні та маленькі об'єкти і їх текстурює. Художник по рівням наносить текстури на ландшафт і додає декалі та розставляє те, що зробили художники по оточенню, заповнює світ. Зелень розмазується по ландшафту інструментами рушія та настроює світло.

Наративний дизайнер додає композиції, підказки, записки.

Вода також створюється за допомогою інструментів. Самий простий варіант це плоска поверхня з текстурою. Більш складний варіант це тривимірний об'єм з змінюваною геометрією поверхні. Створюється це завдяки шейдерам. Зазвичай вода це тривимірний об'єм. Коли гравець попадає у неї змінюється фізична модель та постобробка.

На рівнях для мультиплеєру позбавляються візуального шуму щоб гравці краще бачили один одного, тому невеликі об'єкти на кшталт сміття здвигається до стінки

Освітленням також може займатися окрема людина- художник світла. Світло може заздалегідь розраховуватись і запікатись, або може бути динамічним працюючим в реальному часі. Із відблиском також як із освітленням.

Небо це просто велика сфера із текстурою.

Коли рівень готовий його заморожують. З цього часу забороняється додавати невеликі об'єкти, видаляються непотрібні асети. В завершені роблять фінальний тест. Проводиться оптимізація продуктивності. Для цього використовується профайлинг. Це інформація про об'єкти, скільки вони споживають ресурсів. Налаштовуються lod, це коли при віддалені камери від гравця зменшується кількість полігонів. Це потрібно для оптимізації.

Оточуючий світ потрібно також розділяти для оптимізації. Один із старих способів це підвантаження рівнів. Коли гравець проходить один рівень, завантажується інший. Другий спосіб це підвантаження рівня в режимі реального часу. Один великий світ розділяється на фрагменти, чанки і

загружаються лиш ті котрі поблизу гравця. Щоб скрити вдалі недостатні чанки розробники йдуть на хитрість як туман. Або на occlusion culling. При рендерінгу прибирається все, що не попадає у поле зору камери. Ще можна створити вузькі місця. Коли гравець протискується крізь них підвантажується наступний рівень і відвантажується попередній.

Аудіо програміст підключає звуки, діалоги, музику.

Коли фінальний контент потрапляє в гру його повинно зробити грабельним, тобто запрограмувати. Зазвичай програмний алгоритм розпочинається із опису того, що потрібно зробити і потім створюється блок схема. Скомпонував блок схему програмісти приступають к роботі. Зазвичай в ігрових рушіях використовують мови програмування C# та C++ разом із надбудовою до рушія. До основного мови підключаються бібліотеки на кшталт Direct X, OpenGL чи Vulkan та шейдерні бібліотеки для ефектів. Через те що потрібно дуже багато програмувати процедурним методом не зручний, тому розробники ігор використовують методи об'єктно-орієнтованого програмування.

Щоб зробити розробку гри ще просте використовується візуальне програмування в рамках самого рушія. Тут замість коду маніпулюють блок схемами котрі називаються вузли, ноди.

Для мобільних ігор можуть використовувати мови Java чи Swift. Для браузерних використовують Html, PHP та Java Script с додатковими бібліотеками [3].

2.4 Вертикальний зріз

Коли основний контент готовий розробники формують вертикальний зріз гри. Це частина гри без самої гри. Розробники заглядають у майбутнє плануючи як буде виглядати гра.

Пара строчок коду, рівнів та персонажів стане майбутнім каркасом усієї гри. Вертикальний зріз, він же демоверсія, може виконуватись на етапі концепції. Вертикальний зріз демонструють інвесторам [3].

2.5 Виробництво

Виробництво гри. Це найбільш складна і тривала стадія розробки. Це наповнення контентом. Створюється все більше рівнів, 3д персонажів, налаштовується фізика, штучний інтелект. Для платформера штучний інтелект може бути більш глупим, для стратегії в реальному часі потрібен бути більш розумний, для мультиплеєрів взагалі не потрібен, хіба що для ботів. Саунд дизайнер додає звуки.

Наступний крок це додавання графічного користувацького інтерфейсу. Візуальна частина інтерфейсу створюється в будь якому 2д редакторі і далі компонується і програмується в рушії [3].

2.6 Тестування

Коли гра готова наступає стадія тестування. Головна задача тестувальників не шукати помилки у коді, а перевірити чи можна пройти гру до кінця та виявляти що заважає основному процесу гри [3].

2.7 Підготовка до релізу

Коли гра готова можна провести закритий бета-тест а далі і відкритий. Поки дороблюється гра видавець теж не сидить на місці і займається піаром гри. Випускається ігровий трейлер, водночас проєкт може взяти участь у виставках. Заставки гри можуть робити як у вигляді кат сцен, так і у вигляді сінематик роликів [3].

2.8 Оперування

Фінальний це оперування. Це може виконуватись на стороні видавця. Видавець локалізує гру, переводить на різні мови, рекламує її та займається продажами забираючи приблизно 30% прибутку.

В підсумку гра доходить до гравця. Потім вже розробник створює патчі та DLC [3].

2.9 Висновки за розділом 2

У цьому розділі було розглянуто етапи розробки гри. Існує 7 етапів при розробці ігор: концепція, прототипування, вертикальний зріз, виробництво, тестування, підготовка до релізу та оперування.

На етапі концепції придумують гру, її жанр, стиль суть гри та інше. Прототипування це початок розробки гри. Вертикальний зріз являє собою перший концеп гри який показують інвесторам і він буде кістяком для майбутньої гри. При виробництві продовжують розробляти гри. При тестування тестувальники перевіряють чи можливо пройти гру та прибирають деталі або фіксять баги які заважають її пройти. При Підготові до релізу проводять спочатку закритий, а після відкритий бета-тест, також видавець гри починає піарити гру. І останній етап це оперування під час якого вже проводиться випуск та продаж гри. Також розробники продовжують доглядати за грою виправляючи деякі баги та випускаючи DLC та патчі.

3 ОСНОВНІ РІШЕННЯ ЩОДО ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ГРИ

3.1 Вибір мови програмування, ігрового рушія та жанру гри

При розробці ігрового застосунку було використано мову програмування C#.

C# це один із найпопулярніших мов програмування для розробки ігор. Це мова високого рівня яка пропонує багато переваг для розробки ігор, включаючи спрощений синтаксис, надійну бібліотеку класів і ефективне керування пам'яті.

C і альтернативна мови програмування для розробки ігор такі як C++. Ця мова програмування більш складні в опануванні, однак вона більш швидка та ефективна. Ця мова також має велику бібліотеку класів, однак вона потребує ручного виділення та звільнення пам'яті для об'єктів до та після завершення завдання. Також обидві мови підходять для кросплатформеності, однак в C# Windows більш пріоритетний [4].

Це дві найпопулярні мови для програмування, однак для розробки ігор використовується набагато більше.

Обґрунтування вибору мови програмування для розробки ігор можна побачити в таблиці 3.1.

Таблиця 3.1 – Обґрунтування вибору мови програмування для розробки ігрових додатків

Критерії порівняння мов програмування	Мова програмування	
	C#	C++
Синтаксис	+	—
Продуктивність	Менша ресурсоемність	Більша швидкодія
Кросплатформеність	+—	+
Швидкість розробки	+	—
Графічний інтерфейс	+—	+

Із таблиці 3.1 можна дійти висновку що мова програмування C# більше підходить для розробників інді ігор. C++ більше підходить для розробки великобюджетних ігор, які потребують більшої продуктивності та більш гарної графіки.

Окрім мови програмування ще потрібно вибрати ігрову платформу на якій буде створена гра. Ігрові платформи допомагають у розробці оскільки надають готові модулі та інструменти, які допомагають зосередитись на розробці ігор не витрачаючи час на написання коду з нуля.

Існує багато ігрових рушіїв. Найбільш популярними є Unity та Unreal Engine. Різниця між цими двома платформами велика, від ціни до способу програмування. Основу різницю для створення 2D ігор між цими двома платформами можна побачити у таблиці 3.2.

Таблиця 3.2 – Різниця між платформами Unity та Unreal Engine для створення 2D ігор

Критерій порівняння ігрових рушіїв	Ігровий рушій	
	Unity	Unreal Engine
Мова програмування	C#, JavaScript	C++, Blueprint
Графічний інтерфейс	+	+-
Візуальний редактор	+	+-
Фізика	+	-
Анімації	+	+-
Активне співтовариство	+	-
Ціна	Безкоштовна, якщо розробник заробив менше 100 тис. доларів за рік	Безкоштовна, якщо розробник заробив менше 3 тис. доларів за 4 місяці

З результатів порівняння можна дійти висновку, що Unity краще підходить для створення 2D ігор. Однак не треба вважати Unreal Engine

гіршою за Unity, оскільки Unreal Engine має більш розвинений інструментарій для розробки 3D ігор.

В Unreal Engine краще розвинена візуалізація та графіка, та за рахунок цього анімації виходять кращі ніж в Unity. Однак це лише для 3D ігор. Ці компоненти просто не підходять для 2D ігор оскільки ця платформа і створювалась із упором у графіку. І також ці компоненти складніше опанувати. А Unity більш простий та зручний використанні і опанувати розробнику-початківцю ці компоненти буде легше.

Також щодо співтовариства. Хоча і обидві платформи мають своїх розробників, однак в цьому плані перемагає Unity оскільки він більш популярний у інди розробників.

І останнє про що можна поговорити це ціна. Unity безкоштовна платформа якщо за рік гра заробила менше 100 тис. доларів. UE має право на 5% роялті якщо гра заробила менше 3 тис. доларів за 4 місяці [5].

Мною було вирішено вибрати ігрову платформу Unity через вищеперелічені особливості.

Тепер потрібно вибрати жанр гри. Їх існує велика кількість жанрів, однак найпопулярнішими із них для 2D гри можна виділити три: Roguelike, Metroidvania та платформери. В таблиці 3.3 можна побачити основні аспекти порівнянь цих жанрів.

Таблиця 3.3 – Різниця між жанрами гри

Критерії порівняння жанрів	Ігрові жанри		
	Roguelike	Metroidvania	Платформер
1	2	3	4
Ігровий процес	Збір предметів, поступова прогресія, постійна смерть, випадковість	Нелінійна прогресія, оволодіння новими здібностями, відкриття областей	Перешкоди, рух та стрибки, платформи, збір предметів

Продовження таблиці 3.3

1	2	3	4
Експлорація	Дослідження випадкових локацій	Відкриття нових областей, предметів, пошук секретів, повернення до попередніх локацій з новими здібностями	Пересування по рівням, збирання предметів, часто неможливість повернення у минулий рівень
Прогресія	Отримання нових предметів, повернення на початок після смерті, поступове поліпшення гравця	Отримання нових навичок та здібностей, розблокування нових областей	Пройходження рівнів зі збільшуваним рівнем складності, вдосконалення навичок
Система смерті	Перманентна смерть, втрата прогресу, початок із початку гри	Втрата частини прогресу після смерті, збереження ключових елементів	Обмежене число життів, початок рівня після смерті
Постійність гри	Кожна гра- унікальний досвід	Контрольований прогрес, можливість повернутись у попередню локацію	покращення навичок після кожного перепроходження

Із вищеперелічених критеріїв можна побачити різницю між цими жанрами, однак вони і багато в чому схожі. У всіх трьох жанрах є схожі або однакові елементи та механіки без яких неможливо створити гру. Можливо є спеціальні види переміщення такі як додатковий стрибок або dash. Також зазвичай у жанрах Roguelike Metroidvania існують локації із платформерною складовою. Однак найголовніша різниця між цими жанрами це складність при розробці.

В таблиці 3.4 можна побачити різницю при розробці гри.

Таблиця 3.4 – Різниця при створенні гри

Критерії порівняння при розробці	Ігрові жанри		
	Roguelike	Metroidvania	Платформер
Створення ігрових локацій	Велика кількість кімнат які з'єднані між собою	Великий продуманий світ із своєю історією	Невеликі локації, велика кількість цих локацій
Програмування	Випадкова генерація рівнів, предмети або навички які будуть відкриватися після кожного проходження	Велика кількість механік які будуть відкриватися з проходженням гри	Потрібні лише базові елементи цього жанру
Арт-дизайн	Піксельна графіка, піксельна анімація здібностей та ефектів	Деталізовані арти областей та персонажів, анімації здібностей та ефектів	Розробка яскравого арту рівнів, персонажів, взаємодії з оточенням

При розробці метроїдванія буде найскладнішою для розробки, оскільки вона потрібна бути і з гарною графікою, з продуманим сюжетом та з великою кількістю механік.

Платформер буде найлегшою при створенні оскільки у цьому жанрі потрібно лише створити велику кількість складних локацій та реалізувати декілька механік для руху.

А ось із Roguelike знаходиться десь по середині. По першу, більшість представників жанру використовує піксельну графіку, яку легше створити. Створення скрипту для випадкової генерації рівнів може бути складною, однак достатньо створити генерацію один раз і після цього копіювати його на кожен рівень змінюючи кімнати. Сюжет може бути мінімальним, лише для того щоб був. Також головне те, що грати у представників цього жанру завжди цікаво оскільки кожна нова гри не щось нове.

Із-за всіх вищеперелічених критеріїв я вирішив вибрати жанр Roguelike. Взагалі, жанр Roguelike описати словами складно оскільки не існує єдиного чіткого визначення цього жанру. Цей жанр можна просто описати як подорож по рандомізованим підземеллям майже або без сюжету, боротьба із противниками, нескінченно повторюючи цей процес [6].

Найбільш узагальненим визначенням можна назвати Берлінську інтерпретацію. Ця інтерпретація містить 12 принципів, однак головними можна назвати декілька. Це будуть перманента смерть, випадкова генерація рівнів, відображення характеристик числами, управління ресурсами та hack'n'slash [7].

3.2 Модулі Unity та схематична структура гри

Перш за все поговоримо про модулі які є в Unity. Вони можуть як бути вбудовані в ігрову платформу, так і їх потрібно буде завантажити із Package Manager. В Package Manager знаходяться всі пакети та модулі які були створені розробниками ігрової платформи. Також можна завантажити

сторонні модулі які були створені сторонніми розробниками. Такі модулі можна знайти на сайті Unity Asset Store. Також вони можуть бути як безкоштовні, так і продаватись за якусь ціну.

В таблиці 3.5 можна побачити всі модулі які будуть використовуватись у моєму проекті.

Таблиця 3.5 – Модулі

№	Назва модуля	Компоненти
1	Unity Editor	Редактор
		Інспектор
		Панель проєкту
		Інші редакторські інструменти
2	Physics	Фізичний рушій
		Зіткнення
		Гравітація
		Динаміка об'єктів
3	Scripting	Розробка ігрової логіка за допомогою скриптів
		Підтримка мов програмування
4	Graphics	Графічний рушій
		Рендеринг у реальному часі
		Шейдери
		Ефекти освітлення та пост-процесингу
5	Animations	Анімаційні стани
		Параметри анімації
		Переходи між станами

Продовження таблиці 3.5

№	Назва модуля	Компоненти
6	Audio	Відтворення звуків
		Керування гучністю
		Просторовий звук
		Аудіо ефекти
7	UI	Створення елементів користувальницького інтерфейсу
		Кнопки
		Текстові поля
		Панель
		Меню
		Інші елементи
8	Cinemachine	Різні види камер

Модуль Unity Editor надає сцену, яка знаходиться посередині і в ній ми розставляємо елементи, інспектор який знаходиться в правій частині екрану і в якому ми налаштовуємо змінні, властивості і тому подібне та панель проєкту в нижній частині екрану та який відповідає за збереження всіх файлів, скриптів і тому подібне.

Модуль Physics надає нам фізичний рушій разом із твердими тілами для персонажів, об'єктів та мапи.

Модуль Scripting дозволяє розробляти ігрову логіку за допомогою скриптів.

Модуль Graphics дасть нам графічний рушій та можливість рендерингу шейдерів, освітлень та ефектів.

Модуль Animations дозволяє створювати анімації та налаштовувати їх.

Модуль Audio дозволяє взаємодіяти із музикою та звуками.

Модуль UI дозволяє створювати користувальницький інтерфейс.

Модуль Cinemachine був завантажений із Package Manager, він дасть нам можливість налаштовувати камеру без необхідності створювати скрипт.

Із модулями закінчили, тепер поговоримо про гру. Roguelike буде реалізовуватись за допомогою випадкової генерації рівнів. Розробка гри буде реалізовуватись за допомогою сцен. Сцени містять об'єкти гри. Вони використовуються для створення меню та окремих рівнів. Кожну сцену можна вважати окремим ігровим рівнем [8].

В моєму випадку будуть три сцени. Перша сцена це головне меню. Друга сцена відповідає за початкову локацію та третя сцена це ігрова локація із супротивниками. Також будуть реалізовані скрипти які дозволять змінювати сцену (рис. 3.1).

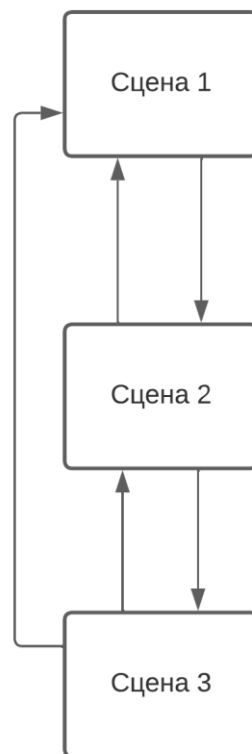


Рисунок 3.1 – Схема переміщення по сценам

Також, окрім сцен ще будуть елементи інтерфейсу такі як меню і вони будуть відрізнятися у кожній сцені. На рисунку 3.2 можна побачити структуру гри, яка допомагає зрозуміти взаємодію між сценами та інтерфейсами.

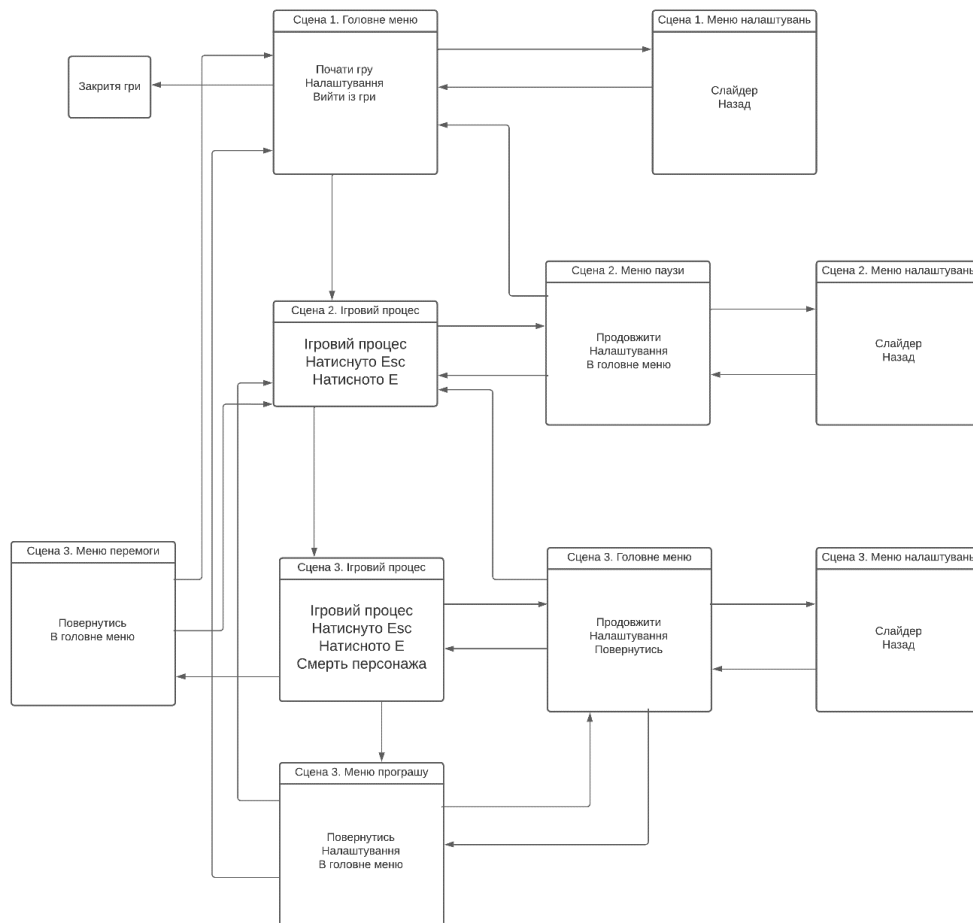


Рисунок 3.2 – Діаграма структура гри

Гра починається із першої сцени. Із неї можна перейти в меню налаштувань, почати гру або вийти із гри при натисканні відповідної кнопки. Із меню налаштувань можна повернутись у попереднє меню аналогічно у всіх сценах.

В 2 сцені при натисканні на Esc включиться меню в якому можна продовжити гру, зайти в меню налаштувань чи повернутись у нульову сцену. Якщо ж під час ігрового процесу натиснути на E біля дверей, то буде перехід у 3 сцену.

І остання сцена. У неї також є меню паузи в якій є кнопка продовження, налаштування та кнопка повернення яка вмикає 1 сцену. Також у цій сцени є вороги і якщо від них померти, ти ввімкнеться меню програшу із якого можна повернутись у 2 сцену, ввімкнути меню налаштувань або повернутись у 1 сцену. Також якщо перемогти боса та встати біля дверей, то при натисканні E

з'явиться меню перемоги із якого можна повернутись у 2 сцену при натисканні кнопки повернення або у 1 сцену при натисканні кнопки повернення в меню.

3.3 Сцена 1

Перша сцена це меню. Меню створюється за допомогою модуля UI. Спочатку потрібно створити об'єкт canvas із цього модуля який буде зберігати всі компоненти цього модуля. Далі у canvas додаємо задній фон за допомогою об'єкта який містить компонент image. Тепер створимо два пустих об'єкта який перейменуємо задля комфорту та вже в них будемо створювати початковий інтерфейс. Перший інтерфейс буде початковим меню та другий буде меню налаштувань. Ці два інтерфейси будуть знаходитись у всіх сценах, однак початкове меню буде трохи змінено під кожну сцену.

В першому меню є 3 кнопки. Перша кнопка відповідає за початок гри (рис 3.3). Друга кнопка це перехід в меню налаштувань. Остання кнопка це вихід із гри.

Друге меню це меню налаштувань яке містить повзунок який дозволяє регулювати гучність музики та звуків (рис. 3.4). Також є кнопка повернення у головне меню.



Рисунок 3.3 – Головне меню

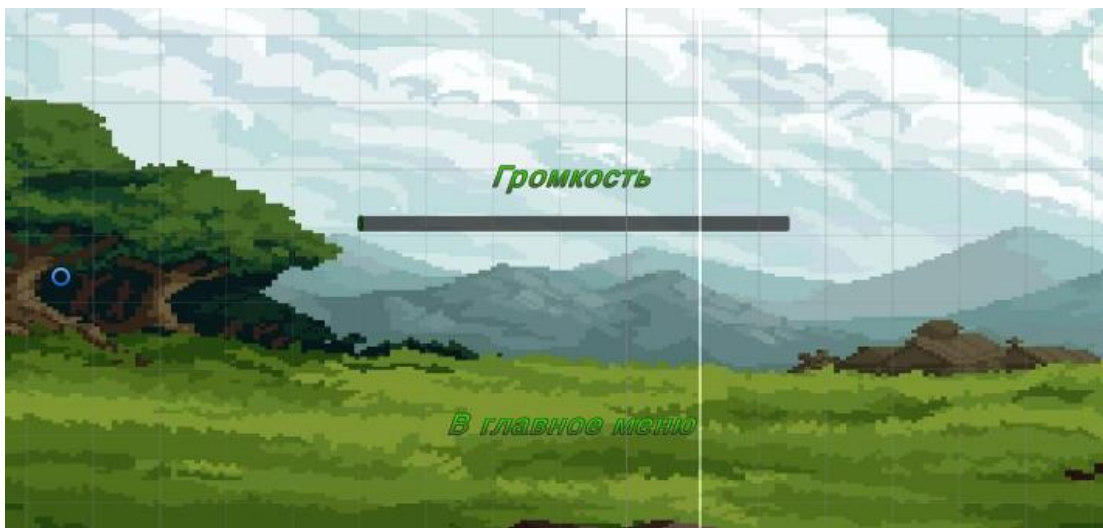


Рисунок 3.4 – Меню налаштувань

Взаємодію із кнопками можна реалізувати двома способами. Перший спосіб це створення скрипту. Кнопка “почати гру” створена за допомогою скрипту і вона змінює першу сцену на другу [9].

Другий спосіб дозволяє робити активним іншу кнопку лише за допомогою активації кнопки в “GameObject”. В цьому способі потрібно додати до “On click()” інший об’єкт та виставити GameObject.SetActive true. Тепер при натисканні на кнопку буде активуватись кнопка яку ви додали. В обох методах потрібно активувати інший об’єкт та деактивувати об’єкт який налаштовуєте (рис. 3.5).

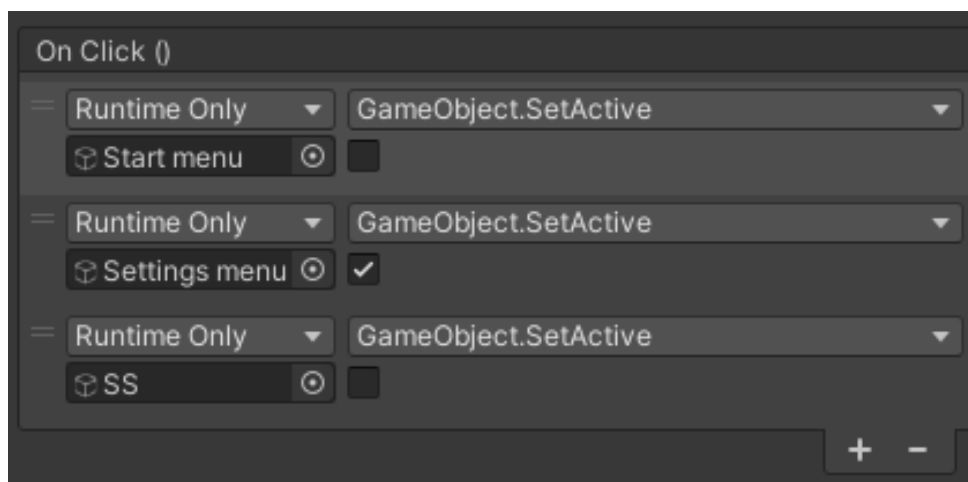


Рисунок 3.5 – Метод взаємодії з кнопками без скрипту

На кожній кнопці і в меню є музика і звуки. Щоб їх додати потрібно із створити об'єкти та із модуля Audio додати до них Audio Source та налаштувати скрипт. Для звуків потрібно створити скрипт. В скрипті потрібно лише два методи. Перший це програвання звуку при наведенні на кнопку. Другий метод буде програванням звуку при натисканні на кнопку. Створюємо різні звуки та додаємо скрипт до кнопки налаштовуючи ці звуки. Для цього до змінних додаємо об'єкту із компонентом Audio Source. Далі додаємо компонент event trigger та налаштовуємо його (рис. 3.6).

Музика програвється без скрипту, потрібно лише в Audio Source поставити галочку на Play on Awake.

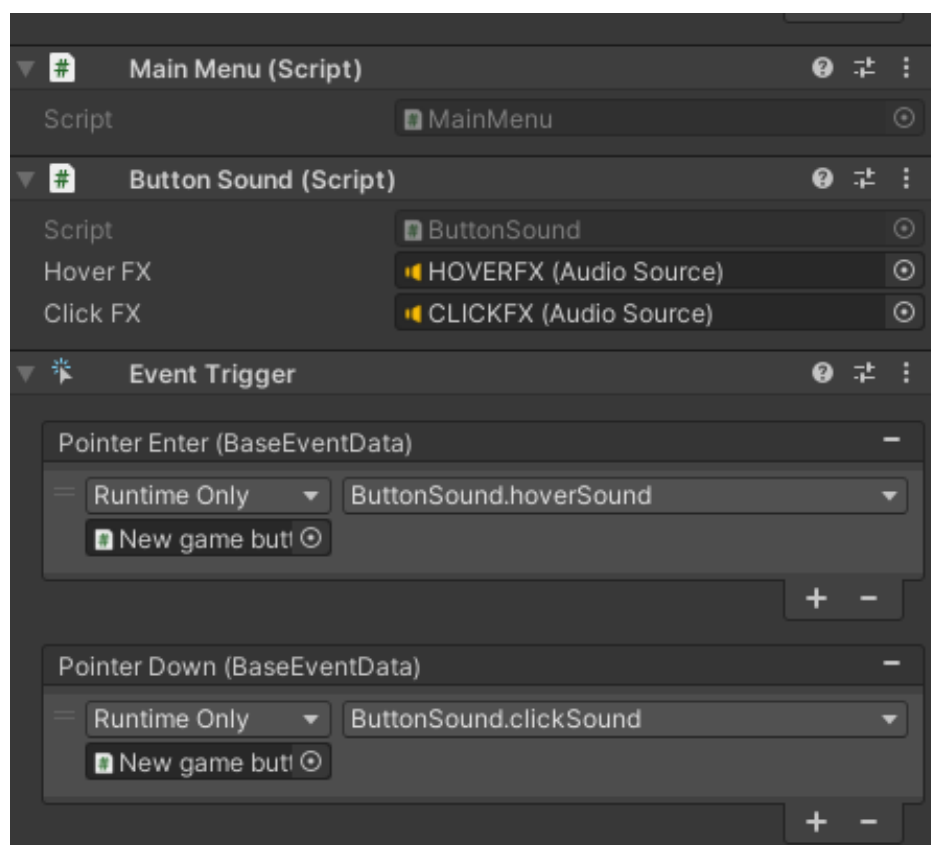


Рисунок 3.6 – Налаштування звуків

Також є скрипт завантаження наступного рівня. Він асинхронно завантажує іншу сцену та переходить до неї коли вона майже завантажиться. Це потрібно із-за того, що спочатку активується наступна сцена і після цього

вона починає завантажуватись. В цьому випадку можна додати екран завантаження (рис. 3.7).

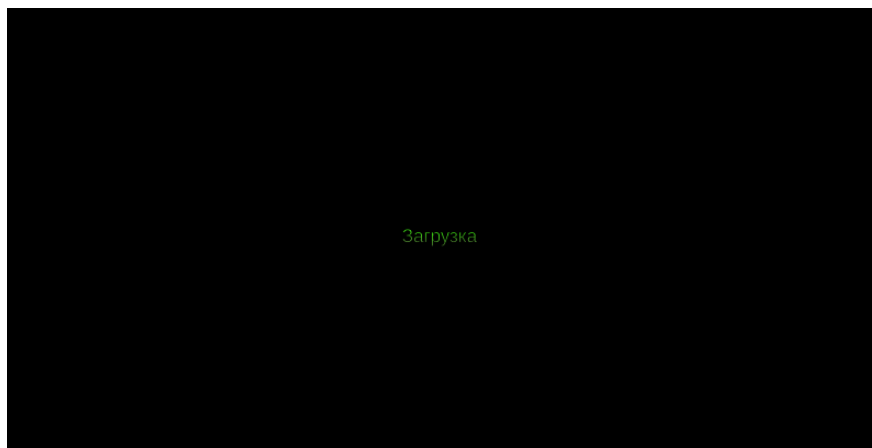


Рисунок 3.7 – Завантаження наступного рівня

Структуру цієї сцени можна побачити на рисунку 3.8.

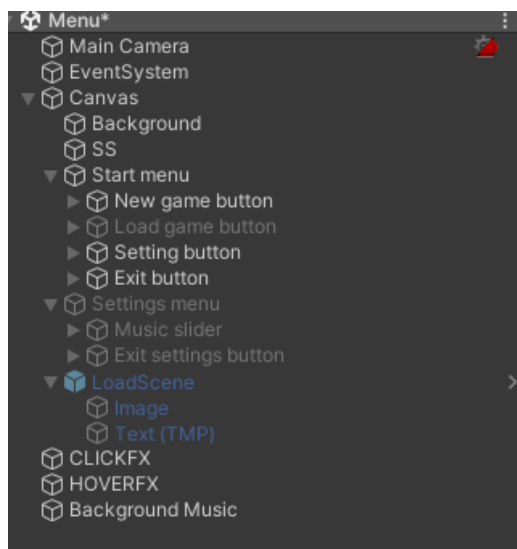


Рисунок 3.8 – Завантаження наступного рівня

3.4 Сцена 2

Друга сцена буде початковою локацією. Зазвичай в таких локаціях розташовані неігрові персонажі в яких можна придбати предмети. Також зазвичай в цій локації можна зберігати процес гри.

Спочатку потрібно створити ігрового персонажа. Для цього потрібно створити пустий об'єкт, додати до нього потрібний колайдер та фізичне тіло. Після цього потрібно зробити так, щоб камера рухалась вслід за персонажем. Для цього можна створити скрипт чи завантажити розширення із Package Manager яке дозволить налаштовувати камеру без скрипту. Це розширення називається Cinemachine. Спочатку потрібно завантажити це розширення і імпортувати його у проєкт. І із нього потрібно дістати 2D camera, у Follow камери вставляємо нашого персонажа [10].

Далі створюємо скрипт яким буде головним для персонажа. В цьому скрипті дамо можливість руху для персонажа, анімації та інші елементи ігрового процесу [11]-[12].

Рух створити нескладно, однак може бути проблема зі стрибками оскільки для Unity вертикальний і горизонтальний рух однакові. Щоб створити нормальний стрибок для персонажа спочатку потрібно додати мітку до ніг та зробити перевірку у скрипті: якщо ця мітка контактує із землею персонаж може стрибати, якщо ж ця мітка не взаємодіє з землею, гра буде вважати що персонаж знаходиться в повітрі. Для цього для землі потрібно створити новий шар і позначити це в скрипті. Тепер персонаж буде стрибати лише один раз (рис. 3.9).

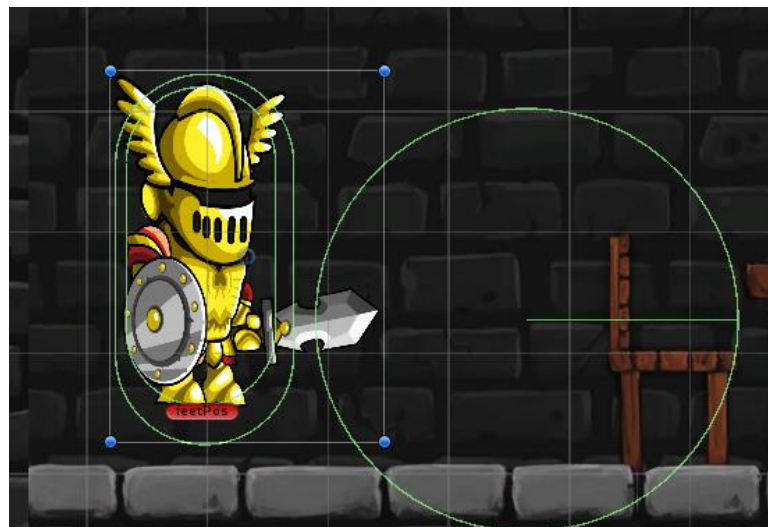


Рисунок 3.9 – Головний герой із об'єктами для розпізнавання землі та областю для атаки

Також потрібні анімації. Для цього спочатку в аніматорі створюємо анімації, прикріплюємо аніматор до персонажа та налаштовуємо у скрипті [13].

Тепер потрібно додати до героя здоров'я. У персонажа фіксовані 5 одиниць здоров'я, супротивник має одиницю атаки. Скрипт для атаки по герою додаємо до супротивника і він буде таким: якщо колізія героя потрапляє в невеличку область біля супротивника, то у героя віднімається здоров'я. Якщо у героя здоров'я буде дорівнювати нулю, то герой вмирає і запускається меню програшу.

Також створимо healthbar для героя за допомогою слайдеру та скрипту який буде змінювати слайдер із відніманням здоров'я [14].

Останнє що потрібно для персонажа це можливість наносити пошкодження супротивникам. Для цього додаємо новий об'єкт до персонажа із областю та вимкнемо його. Далі створимо скрипт який буде віднімати у супротивників здоров'я якщо супротивник увійде у цю зону. Далі потрібно налаштувати це під користувача. Потрібно буде зробити так, щоб ця область вмикалась на мить коли гравець натискає якусь кнопку (рис. 3.9).

Також у героя потрібні бути звуки. Для цього створимо два об'єкта і додаємо до них Audio Source. Один звук буде для руху а інший для атаки.

Для звуку атаки додаємо у методі атаки swordSwing.Play(). Тепер якщо відбудеться атака то програветься звук. Із рухом буде складніше оскільки якщо додати звук до функції Update він буде постійно програватися оскільки цей метод оновлюється 1 раз за кадр. Для цього створимо нову функцію де буде затримка перед повторним програванням звуку і додаємо його до руху [15].

Тепер довікілля. Для створення ігрової локації потрібно створити об'єкт та додати до нього колізію. Тепер на цьому об'єкті може знаходитись персонаж. Також за бажанням можна додати спрайт до цього об'єкту, однак із великою долею вірогідності потрібно буде вручну змінювати колізію. Також створюємо об'єкт без колайдера для заднього фону(рис. 3.10). Останнє що знадобиться це розставити пріоритети у Sprite Renderer, а саме у Order in

Layer. Для заднього у фона пріоритет буде 0, для об'єктами із колайдерами 1 та для героя 2.



Рисунок 3.10 – Початкова локація

Інтерфейс у грі створюємо так само як і в 1 сцені, однак спочатку цю цей інтерфейс вимкнемо. Далі при натисканні на кнопку ставимо гру на паузу та вмикаємо інтерфейс за допомогою скрипту. Щоб поставити гру на паузу потрібно просто змінити течію часу в грі за допомогою `Time.timeScale=0f`, але потрібно буде відновлювати її змінивши нуль на одиницю при продовженні гри. Також в такому меню потрібно додати вихід в головне меню просто зробивши перехід на нульову сцену в скрипті (рис. 3.11) [16].



Рисунок 3.11 – Меню паузи

Останнє, що потрібно зробити в цій сцені це перехід в іншу сцену. Перехід реалізований за допомогою дверей до якої доданий скрипт. Він спочатку перевіряє чи знаходиться гравець поблизу. Якщо знаходиться то при натисканні на якусь кнопку він асинхронно запускає іншу сцену та вмикає екран завантаження (рис. 3.12).



Рисунок 3.12 – Завантаження наступної сцени

3.5 Сцена 3

Остання і головна сцена. У ній буде реалізовано Roguelike складову гри. Також тут будуть супротивники та інші складові ігрового процесу.

Спочатку створимо ігрову локацію із випадковою генерацією. Для цього створюємо початкову локацію в якій з'являється герой та додаємо із чотирьох сторін та у центрі кімнати мітки (рис. 3.13).

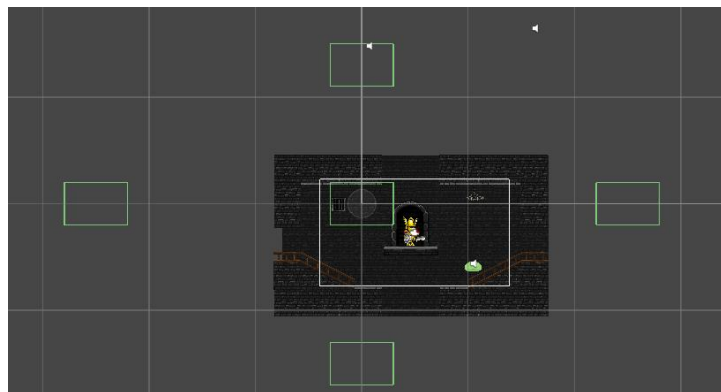


Рисунок 3.13 – Початкова кімната із мітками

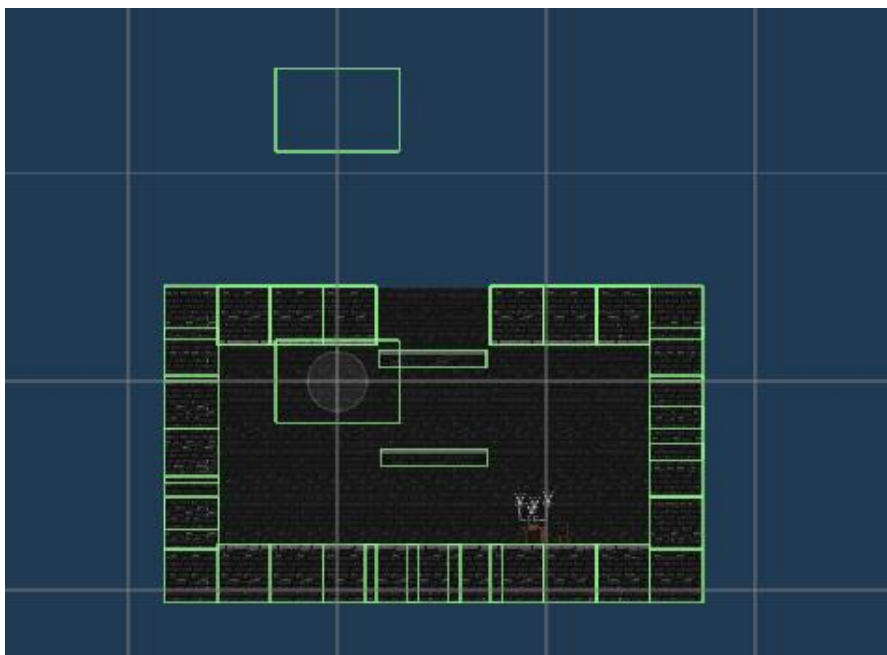


Рисунок 3.14 – Кімната із одним входом

Тепер створимо саму генерацію рівнів. Для цього нам потрібні два скрипта. В першому скрипті буде перелік кімнат. Створимо новий об'єкт і додаємо до нього цей скрипт із переліком кімнат. Якщо додати декілька однакових кімнат то шанс появи цієї кімнати виросте (рис. 3.15).

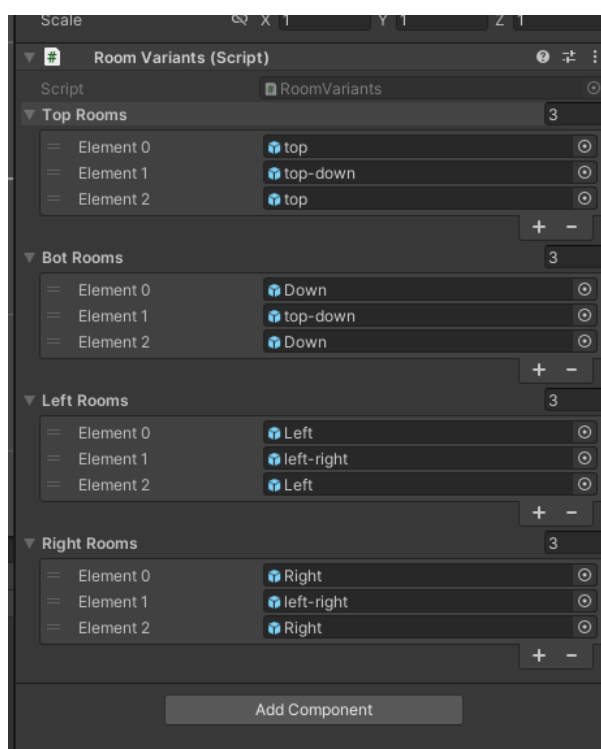


Рисунок 3.15 – Перелік кімнат

Другий скрипт буде генерувати розташування кімнат. Для цього на кожну мітку додаємо її розташування та за допомогою двох строчок коду додаємо кімнати.

```
rand = Random.Range(0, variants.TopRooms.Length);
Instantiate(variants.TopRooms[rand], transform.position,
variants.TopRooms[rand].transform.rotation);
```

Перша строчка коду випадково вибирає кімнату та друга розташовує їх. Це робимо для всіх чотирьох сторін.

Останнє що потрібно буде зробити це додати перевірку яка буде видаляти кімнати якщо вони були створені на місці іншої. Для цього і була створена мітка посередині. Тепер за бажанням робимо скільки завгодно кімнат та розставляємо у першому скрипті. Якщо більше вільних міток не буде то і кімнати створюватись не будуть [17]-[18].

Алгоритм створення кімнат можна побачити на рисунку 3.16

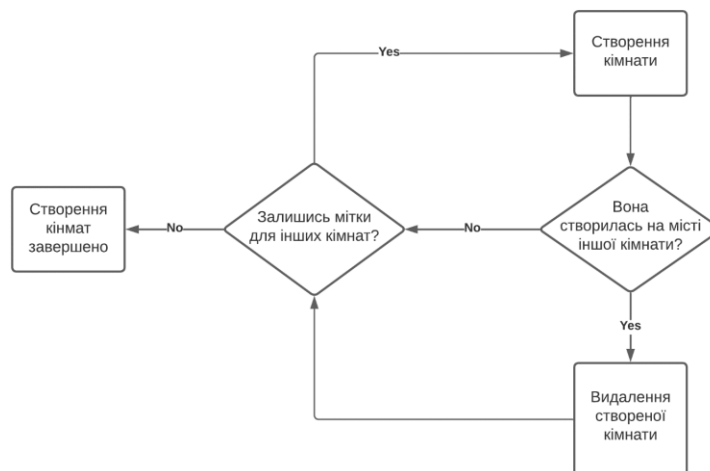


Рисунок 3.16 – Алгоритм створення кімнат

Тепер платформи. Для них теж потрібен буде невеликий скрипт. Створимо скрипт який буде давати можливість падати крізь нього щоб рухатись між кімнатами. Для цього створимо перевірку чи знаходиться герой на платформі. Якщо так, то при натисканні на задовільну кнопку герой не буде взаємодіяти із колізією платформи. Для цього потрібно буде створити 2 шара, для героя та для платформи. Також потрібно буде повернути колізію через

задовільний час. Тепер дамо можливість проходити знизу вверху по платформі. Для цього додаємо до платформи `platform effector`. За замовчуванням він вже буде налаштований так, щоб можна було так рухатись.

Тепер супротивник. Він буде реалізований у вигляді слайма. Більшість скрипту для слайма однакова із скриптом героя. Різниця буде у русі. Слайм може рухатись лише стрибками. Тож потрібно буде знайти героя та дати рух до нього. Для цього потрібно буде знайти його за тегом та передати інформацію до `transform` перемінної. Це потрібно тому що ми будемо шукати координати героя і в `GameObject` цього немає. Тепер робимо самі стрибки та затримку перед кожним стрибком. Тепер рух для слайму готовий. Також за бажанням можна додати звуки.

Здоров'я слайму. Створимо новий скрипт. Додаємо здоров'я та якщо у слайма здоров'я буде дорівнювати нулю то активуємо анімацію смерті та видаляємо об'єкт через пів секунди. Якщо ж здоров'я більше нуля, то при отриманні пошкодження він буде трохи відлітати. Також потрібно створити якийсь індикатор того, що слайм отримав пошкодження. Для цього створимо новий матеріал. Він буде повністю червоним. Тепер якщо слайм отримає пошкодження то він буде замінювати основний матеріал слайма на пів секунди. Після буде повертатись звичайний матеріал. Далі в скрипті атаки героя будемо шукати сам скрипт здоров'я.

```
if (collision.GetComponent<EnemyHealth>())
{
    HandleCollision(collision.GetComponent<EnemyHealth>());
}
```

Тепер, якщо у якогось об'єкту буде скрипт здоров'я, то скрипт атаки героя зафіксує це і буде з ним взаємодіяти.

І останній скрипт для слайма це буде можливість наносити пошкодження герою. Для цього через наслідування візьмемо метод здоров'я у героя та якщо герой взаємодіє із областю, то у нього зменшується здоров'я (рис. 3.17).

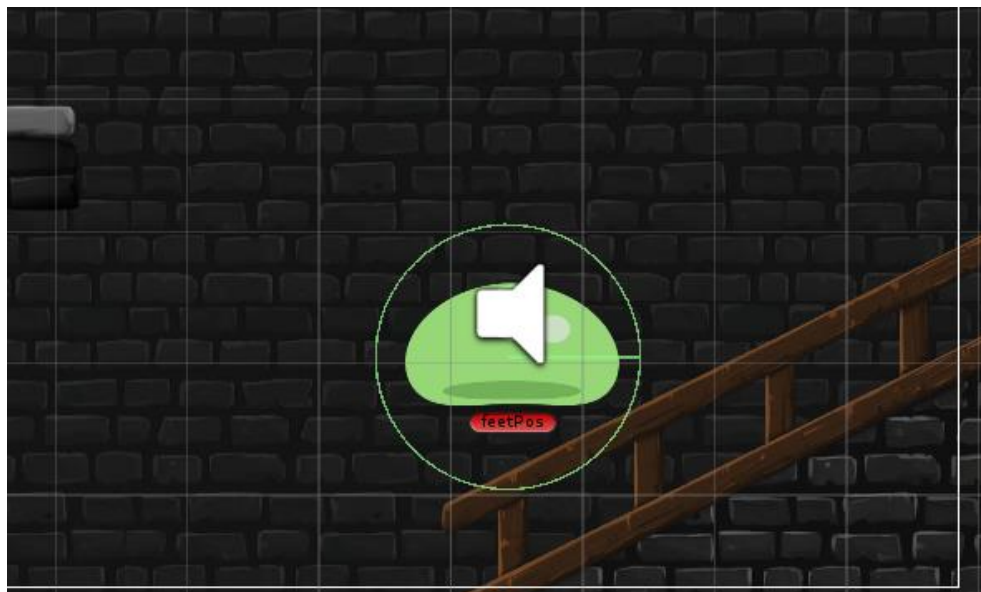


Рисунок 3.17 – Область атаки слайма

Інтерфейс робимо як і в попередній сцені трохи змінивши його.

Останнє що притаманне Roguelike грі це битва із босом. Для цього створимо невеликий скрипт. Якщо він активується, то з'являються два об'єкта та бос. Об'єкти потрібні для того щоб неможливо було утекти від боса. Для самого боса додаємо healthbar та трохи змінимо спрайт (див. рис. 3.18).



Рисунок 3.18 – Бос та активовані об'єкти

І останнє що потрібно буде зробити це додати скрипт кінця гри. Для цього в кінці кімнати є двері які працюють як у другий сцені але він буде показувати меню кінця гри (рис. 3.19).

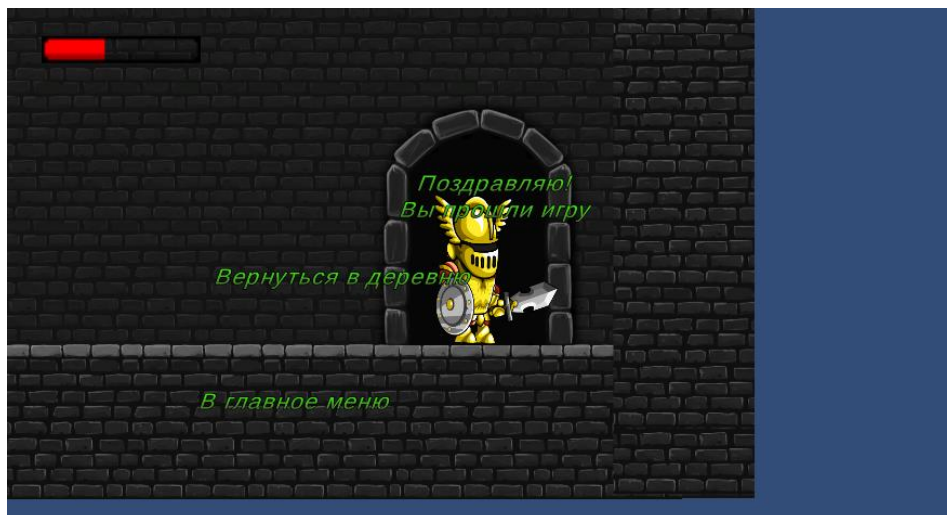


Рисунок 3.19 – Кінець гри

Це був останній скрипт. Єдине що залишилось це натиснути Build and Run та чекати завершення. Тепер гра повністю зібрана і знаходиться у окремій папці. Після цього можна закрити Unity та запустити гру.

3.5 Висновки за розділом 3

У даному розділі було розглянуто створення самої гри. Було показано створення та налаштування інтерфейсу, музики, героя, мапи та супротивників.

Для головного героя та супротивника потрібно додати мітку біля ніг і налаштувати скрипт щоб при взаємодії цієї мітки з землею вважалось, що вони стоять на землі. Це потрібно для стрибків та зміни спрайтів. Також для твердих поверхонь потрібно додати шар який і буде індикатором того, що вони є землею.

Для створення локації додати колайдер та спрайт на задовільний об'єкт, а для створення випадкової генерації рівнів потрібно створити два

скрипта. Перший скрипт буде контейнером для префабів кімнат. Другий буде розставляти ці кімнати.

Інтерфейс створюється у всіх сценах. Він створюється додаванням `EventSystem` яка додає у поле розробки область у якій вже потрібно розставити елементи інтерфейсу. Сам інтерфейс створений за допомогою скриптів, однак є і кнопки які налаштовані за допомогою рушія.

ВИСНОВКИ

В ході виконання дипломної кваліфікаційної роботи бакалавра було проаналізовано та досліджено всі професії, які існують при розробці гри, процес розробки комп'ютерних ігор та створили власну комп'ютерну гру у жанрі Roguelike.

Визначено, що у наш час існують 4 категорії розробників: програмісти, художники та геймдизайнери, яких наймають компанії для розробки ігор та звукорежисери яких наймають на фрілансі. Визначено, що існують 7 етапів розробки гри: концепція, прототипування, вертикальний зріз, виробництво, тестування, підготовка до релізу та оперування.

Сформульовано вимоги комп'ютерної гри.

Для реалізації комп'ютерної ігри обрано мову ігрову платформу Unity, яка має весь потрібний функціонал для розробки гри та легша у навчанні для новачків.

Для створення комп'ютерної гри обрано середовище розробки Visual Studio та мову програмування C#.

Розроблено схематичну структуру гри та саму гру у жанрі Roguelike. Гра має основний функціонал представників жанру: 2D графіка, hack'n'slash, випадкова генерація рівнів, можливість завершити гру у будь який момент часу.

Виконано тестування комп'ютерної гри. За результатами тестування комп'ютерної гри у жанрі Roguelike на платформі Unity виявлено, що програма відповідає технічному завданню, функціонує коректно та дозволяє завершити гру.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Кто такой геймдизайнер в игровой индустрии. Кар'єра в геймдеві [Електрон. ресурс]. – Режим доступу: <https://gamedev.dou.ua/articles/game-designer-in-gamedev/>.
2. Разработка игр| Какие профессии существуют в геймдеве- что делают разработчики игр [Видео ресурс]. – Режим доступа: https://www.youtube.com/watch?v=q_N6ijrV5uw.
3. Как делают игры | Все этапы создания игр – подробно [Видео ресурс]. – Режим доступа: https://www.youtube.com/watch?v=ZVYpRti_7pM.
4. C# Vs C++: Difference between C# and C++ [Electronic resource] – Access mode: <https://www.interviewbit.com/blog/c-sharp-vs-cpp/>.
5. Unity vs Unreal Engine: Game engine comparison guide for 2023 [Electronic resource]. – Access mode: <https://www.evercast.us/blog/unity-vs-unreal-engine>.
6. История жанра Roguelike: от Rogue до Binding of Isaac [Электрон. ресурс]. – Режим доступа: <https://habr.com/ru/articles/493890/>.
7. Berlin interpretation [Electronic resource]. – Access mode: http://www.roguebasin.com/index.php/Berlin_Interpretation.
8. Scenes [Электрон. ресурс]. – Режим доступа: <https://docs.unity3d.com/ru/530/Manual/CreatingScenes.html>.
9. Как создать внутриигровое меню в Unity [Электрон. ресурс]. – Режим доступа: <https://habr.com/ru/articles/346370/>.
10. 2D Camera in Unity (Cinemachine Tutorial) [Electronic resource]. – Access mode: <https://www.youtube.com/watch?v=2jTY11Am0Ig>.

11. Как сделать 2D платформер на Unity| #1 [Видео ресурс].
– Режим доступа:
<https://www.youtube.com/watch?v=sX0hhZT9wDg>.
12. Как сделать 2D платформер на Unity #2| анимации и слежение
камеры [Видео ресурс]. – Режим доступа :
<https://www.youtube.com/watch?v=qHFsgJVyO2Q>.
13. 2D Animation in Unity(Tutorial) [Video resource]. – Access mode:
<https://www.youtube.com/watch?v=hkaysu1Z-N8>.
14. How to make a HEALTH BAR in Unity! [Video resource].
– Access mode:
https://www.youtube.com/watch?v=BLfNP4Sc_iA.
15. Sound Effects & Background Music | Build a 2D Platformer Game in
Unity [Video resource]. – Access mode:
<https://www.youtube.com/watch?v=J77CMuAwVDY>.
16. 6 Minute PAUSE MENU Unity Tutorial [Video resource].
– Access mode:
<https://www.youtube.com/watch?v=9dYDBomQpBQ>.
17. RANDOM DUNGEON GENERATOR - EASY UNITY TUTORIAL -
#1 [Video resource]. – Access mode:
<https://www.youtube.com/watch?v=qAf9axsyijY>.
18. RANDOM DUNGEON GENERATOR - EASY UNITY TUTORIAL -
#2 [Video resource]. – Access mode:
https://www.youtube.com/watch?v=eR74EjkA_4s.

ДОДАТОК А
Текст програми

A.1 Текст скрипты ButtonSound.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class ButtonSound : MonoBehaviour
{
    public AudioSource hoverFX;
    public AudioSource clickFX;
    public void hoverSound()
    {
        hoverFX.Play();
    }

    public void clickSound()
    {
        clickFX.Play();
    }
}
```

A.2 Текст скрипты EndGame.cs

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

public class EndGame : MonoBehaviour
{
    public LoadScreen ld;
    [SerializeField] public bool isInRange;
    [SerializeField] private KeyCode interactKey;
    [SerializeField] private GameObject endGameMenu;
    public void ExittheDungeon()
    {
        ld.Loading();
    }
    void Update()
    {
```

```

        if (isInRange)
        {
            if (Input.GetKeyDown(interactKey))
            {
                endGameMenu.SetActive(true);
                Time.timeScale = 0f;
            }
        }
    }
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            isInRange = true;
        }
    }
    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            isInRange = false;
        }
    }
}

```

A.3 Текст скрипту EnemyHealth.cs

```

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

public class EnemyHealth : MonoBehaviour
{
    [SerializeField] private int healthAmount=20;
    [SerializeField] private float invulnerabilityTime = .2f;
    [SerializeField] public Animator anim;
    [SerializeField] public Transform slime;
    [SerializeField] public Transform knight;
    public AudioSource HitSound;
    private Rigidbody2D rb;
    private Vector3 direction;
    private bool hit;
    private int currentHealth;
}

```

```

private Material matBlink;
private Material matDefault;
private SpriteRenderer sr;
public Health healthBoss;
[SerializeField] public bool bossBar = false;
private void Start()
{
    GameObject Findplayer = GameObject.FindWithTag("Player");
    knight = Findplayer.transform;
    currentHealth = healthAmount;
    if (bossBar) healthBoss.SetMaxHealth(healthAmount);
    sr = GetComponent<SpriteRenderer>();
    rb = GetComponent<Rigidbody2D>();
    matDefault = sr.material;
    matBlink = Resources.Load("EnemyBlink", typeof(Material)) as
Material;
}
public void Damage(int amount)
{
    if (!hit && currentHealth > 0)
    {
        HitSound.Play();
        hit = true;
        currentHealth -= amount;
        if (bossBar) healthBoss.SetHealth(currentHealth);
        sr.material = matBlink;

        if (knight.position.x > slime.position.x)
        {
            rb.AddForce(new Vector2(-50, 30));
        }
        else
        {
            rb.AddForce(new Vector2(50, 30));
        }
    }
    if (currentHealth <= 0)
    {
        anim.SetBool("Dead", true);
        Destroy(gameObject, 0.5f);
    }
    else
    {
        Invoke("ResetMaterial", .2f);
    }
}

```

```

        StartCoroutine(TurnOffHit());
    }
}
void ResetMaterial()
{
    sr.material = matDefault;
}
private IEnumerator TurnOffHit()
{
    yield return new WaitForSeconds(invulnerabilityTime);
    hit = false;
}
}

```

A.4 Текст скрипту GreenSlime.cs

```

using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Diagnostics;
using System.Security.Cryptography;
using UnityEngine;
using System.Threading;
using System.Threading.Tasks;

public class GreenSlime : MonoBehaviour
{
    [SerializeField] public Transform Player;
    [SerializeField] public float slimeSpeed;
    [SerializeField] public float jumpForce;
    [SerializeField] public float checkRadius;
    [SerializeField] public Transform feetPos;
    [SerializeField] public LayerMask whatIsGround;
    [SerializeField] public Animator anim;
    private Rigidbody2D rb;
    [SerializeField] public Transform Slime;
    public AudioSource jumpSound;
    public bool isRange;
    private bool IsGrounded;
    public float delay = 1;
    float timer;
    void Awake()
    {

```

```

        rb = GetComponent<Rigidbody2D>();
        GameObject Findplayer = GameObject.FindWithTag("Player");
        Player = Findplayer.transform;
    }

    public void Update()
    {
        timer = timer + Time.deltaTime;
        if (isRange == true)
        {
            Jump();
        }
    }

    void Jump()
    {
        IsGrounded = Physics2D.OverlapCircle(feetPos.position, checkRadius,
whatIsGround);
        int direction;
        if (Player.position.x > Slime.position.x)
        {
            direction = 1;
        }
        else
        {
            direction = -1;
        }
        if (IsGrounded == true )
        {
            if (timer > delay)
            {
                Playsound();
                rb.velocity = (new Vector2(slimeSpeed * direction, jumpForce));
                timer = 0;
            }
        }
    }

    void Playsound()
    {
        jumpSound.Play();
    }

```

```
}
```

A.5 Текст скрипта Health.cs

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

public class Health : MonoBehaviour
{
    public Slider slider;

    public void SetMaxHealth(int health)
    {
        slider.maxValue = health;
        slider.value=health;
    }
    public void SetHealth(int health)
    {
        slider.value = health;
    }
}
```

A.6 Interactable.cs

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

public class Interactable : MonoBehaviour
{
    public LoadScreen ld;
    [SerializeField] public bool isInRange;
    [SerializeField] public KeyCode interactKey;
    [SerializeField] public UnityEvent interactAction;
    public void EnterTheDungeon()
    {
        ld.Loading();
    }
}
```

```

void Update()
{
    if (isInRange)
    {
        if (Input.GetKeyDown(interactKey))
        {
            EnterTheDungeon();
        }
    }
}
private void OnTriggerEnter2D (Collider2D collision)
{
    if (collision.gameObject.CompareTag("Player"))
    {
        isInRange = true;
    }
}
private void OnTriggerExit2D(Collider2D collision)
{
    if (collision.gameObject.CompareTag("Player"))
    {
        isInRange = false;
    }
}
}

```

A.7 Текст скрипта **JumpDown.cs**

```

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

public class JumpDown : MonoBehaviour
{
    private bool isOnPlatform = true;
    [SerializeField] public KeyCode jumpDown;
    void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            isOnPlatform = true;
        }
    }
}

```

```

void OnTriggerExit2D(Collider2D collision)
{
    if (collision.gameObject.CompareTag("Player"))
    {
        isOnPlatform = false;
    }
}

void Update()
{
    if (isOnPlatform && Input.GetKeyDown(jumpDown))
    {
        Physics2D.IgnoreLayerCollision(9, 10, true);
        Invoke("EnableColl", 0.3f);
    }
}

void EnableColl()
{
    Physics2D.IgnoreLayerCollision(9, 10, false);
}
}

```

A.8 Текст скрипты Knight.cs

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.ComponentModel;
using System.Security.Cryptography;
using System.Threading;
using UnityEngine;

public class Knight : MonoBehaviour
{
    [SerializeField] public float speed;
    [SerializeField] public float jumpForce;
    [SerializeField] public float jumpStartTime;
    private float jumpTime;
    private float Move;
    public Rigidbody2D rb;
    private CapsuleCollider2D capsule;
}

```

```

public bool IsGrounded;
private bool isJump;
[SerializeField] public Animator anim;
[SerializeField] public float checkRadius;
[SerializeField] public Transform feetPos;
[SerializeField] public LayerMask whatIsGround;
[SerializeField] public LayerMask whatIsPlatform;
[SerializeField] public Transform player;
[SerializeField] public KeyCode attackKey;
[SerializeField] public GameObject attackArea;
[SerializeField] public GameObject deadMenu;
[SerializeField] public AudioSource step;
[SerializeField] public AudioSource swordSwing;
public float stepInterval;
private Material matBlink;
private Material matDefault;
private SpriteRenderer sr;
private float lastStepTime;
public bool attackCooldown;
public float nextAttackTime;
public float attackRate;
public int maxHealth = 5;
public int currentHealth;
public Health health;
public bool dead;
void Start()
{
    sr = GetComponent<SpriteRenderer>();
    currentHealth = maxHealth;
    health.SetMaxHealth(maxHealth);
    matDefault = sr.material;
    matBlink = Resources.Load("KnightBlink", typeof(Material)) as
Material;
}
void Awake()
{
    rb = GetComponent<Rigidbody2D>();
}

void Update()
{
    if (dead)
    {
        anim.SetBool("IsDead", dead);
        deadMenu.SetActive(true);
    }
}

```

```

    }
    else
    {
        Move = Input.GetAxisRaw("Horizontal");

        if (Move < 0)
            transform.eulerAngles = new Vector3(0, 180, 0);
        if (Move > 0)
            transform.eulerAngles = new Vector3(0, 0, 0);
        anim.SetFloat("Speed", Math.Abs(Move));
        if(Math.Abs(Move)>0 && IsGrounded==true) PlayFootstep();
        rb.velocity = new Vector2(Move * speed, rb.velocity.y);
        Jump();
        anim.SetBool("Jumping", !IsGrounded);
        if (Time.time >= nextAttackTime)
        {
            attackArea.SetActive(false);
            if (Input.GetKeyDown(attackKey))
            {
                Attack();
                nextAttackTime = Time.time + 1f / attackRate;
            }
        }
    }

}

}

void ResetMaterial()
{
    sr.material = matDefault;
}

public void PlayFootstep()
{
    if (Time.time - lastStepTime > stepInterval)
    {
        step.Play();
        lastStepTime = Time.time;
    }
}

public void TakeDamage(int damage)
{
    currentHealth -= damage;
    health.SetHealth(currentHealth);
}

```

```

        sr.material = matBlink;
        if (currentHealth <= 0)
        {
            dead = true;
        }
        else
        {
            Invoke("ResetMaterial", .2f);
        }
    }
    void Attack()
    {
        swordSwing.Play();
        anim.SetTrigger("Attack");
        attackCooldown = true;
        attackArea.SetActive(attackCooldown);
    }
    void Jump()
    {
        IsGrounded = Physics2D.OverlapCircle(feetPos.position, checkRadius,
        whatIsGround | whatIsPlatform);

        if (IsGrounded == true && Input.GetButtonDown("Jump"))
        {
            isJump = true;
            jumpTime = jumpStartTime;
            rb.velocity = Vector2.up * jumpForce;
        }
        if (Input.GetButton("Jump") && isJump == true)
        {
            if (jumpTime > 0)
            {
                rb.velocity = Vector2.up * jumpForce;
                jumpTime -= Time.deltaTime;
            }
            else
            {
                isJump = false;
            }
        }
        if (Input.GetButtonUp("Jump"))
        {
            isJump = false;

```

```

    }
}
}

```

A.9 Текст скрипты LoadScreen.cs

```

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

public class LoadScreen : MonoBehaviour
{
    public GameObject loadingScreen;

    public void Loading()
    {
        loadingScreen.SetActive(true);
        StartCoroutine(LoadAsync());
    }

    IEnumerator LoadAsync()
    {
        AsyncOperation loadAsync =
SceneManager.LoadSceneAsync(SceneManager.GetActiveScene().buildIndex + 1);
        loadAsync.allowSceneActivation = false;
        while (!loadAsync.isDone)
        {
            if(loadAsync.progress >=.9f && !loadAsync.allowSceneActivation)
            {
                yield return new WaitForSeconds(1.5f);
                loadAsync.allowSceneActivation = true;
            }
            yield return null;
        }
    }
}

```

A.10 Текст скрипту MainMenu.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
public class MainMenu : MonoBehaviour
{
    public void PlayGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
    }
    public void Village()
    {
        SceneManager.LoadScene(1);
        Time.timeScale = 1f;
    }
    public void ExitGame()
    {
        Debug.Log("Closed");
        Application.Quit();
    }
}

```

A.11 Текст скрипту MeleeAttack.cs

```

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

public class MeleeAttack : MonoBehaviour
{
    [SerializeField] private int damageAmount=100;
    private Rigidbody2D rb;

    private void Start()
    {
        rb = GetComponentInParent<Rigidbody2D>();
    }
}

```

```

    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.GetComponent<EnemyHealth>())
        {
            HandleCollision(collision.GetComponent<EnemyHealth>());
        }
    }

    private void HandleCollision(EnemyHealth objHealth)
    {
        objHealth.Damage(damageAmount);
    }
}

```

A.12 Текст скрипта **MusicManager.cs**

```

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

public class MusicManager : MonoBehaviour
{
    [SerializeField] Slider volumeSlider;
    void Start()
    {
        if (!PlayerPrefs.HasKey("MusicVolume"))
        {
            PlayerPrefs.SetFloat("MusicVolume", 1);
            Load();
        }
        else
        {
            Load();
        }
    }
}

```

```

public void ChangeVolume()
{
    AudioListener.volume = volumeSlider.value;
    Save();
}
public void Load()
{
    volumeSlider.value = PlayerPrefs.GetFloat("MusicVolume");
}
private void Save()
{
    PlayerPrefs.SetFloat("MusicVolume", volumeSlider.value);
}
}

```

A.13 Текст скрипту PauseMenu.cs

```

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

public class PauseMenu : MonoBehaviour
{
    [SerializeField] public KeyCode PauseInteract;
    [SerializeField] public GameObject MenuPause;
    [SerializeField] public GameObject MenuSettings;
    [SerializeField] private bool isPaused;
    void Update()
    {
        if (Input.GetKeyDown(PauseInteract))
        {
            if (isPaused)
            {
                ResumeGame();
            }
            else
            {
                PauseGame();
            }
        }
    }
    public void PauseGame()

```

```

    {
        MenuPause.SetActive(true);
        Time.timeScale = 0f;
        isPaused = true;
    }
    public void ResumeGame()
    {
        MenuPause.SetActive(false);
        MenuSettings.SetActive(false);
        Time.timeScale = 1f;
        isPaused = false;
    }
    public void Settings()
    {
        MenuSettings.SetActive(true);
        Time.timeScale = 0f;
        isPaused = true;
    }
    public void Continue()
    {
        MenuPause.SetActive(false);
        Time.timeScale = 1f;
        isPaused = false;
    }
    public void Exit()
    {
        SceneManager.LoadScene(0);
        Time.timeScale = 1f;
        isPaused = false;
    }
    public void ReturnToVilage()
    {
        SceneManager.LoadScene(1);
        Time.timeScale = 1f;
        isPaused = false;
    }

    public void ExtiGame()
    {
        Application.Quit();
    }
}

```

A.14 Текст скрипту PlayerAttack.cs

```

using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Security.Cryptography;
using UnityEngine;

public class PlayerAttack : MonoBehaviour
{
    [SerializeField] public KeyCode attackKey;
    [SerializeField] public Transform attackPoint;
    public float attackRange;
    public LayerMask enemyLayer;
    [SerializeField] public Animator animator;
    public int attackDamage = 1;
    public float attackRate = 2f;
    float nextAttackTime = 1f;
    public bool isAttack;
    private float timer;
    [SerializeField] private GameObject AttackArea;
    void Update()
    {
        if (Time.time >= nextAttackTime)
        {
            AttackArea.SetActive(false);
            if (Input.GetKeyDown(attackKey))
            {
                Attack();

                nextAttackTime = Time.time + 1f / attackRate;
            }
        }
    }
    void Attack()
    {
        animator.SetTrigger("Attack");
        AttackArea.SetActive(true);
    }
}

```

A.15 Текст скрипту RoomSpawner.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class RoomSpawner : MonoBehaviour
{
    private RoomVariants variants;
    private int rand;
    private bool spawned = false;
    public Direction direction;
    public enum Direction
    {
        Top,
        Bot,
        Left,
        Right,
        None
    }
    private void Start()
    {
        variants =
GameObject.FindGameObjectWithTag("Rooms").GetComponent<RoomVariants>(
);
        Invoke("Spawn", 0.1f);
    }
    public void Spawn()
    {
        if (spawned==false)
        {
            if (direction == Direction.Top)
            {
                rand = Random.Range(0, variants.TopRooms.Length);
                Instantiate(variants.TopRooms[rand], transform.position,
variants.TopRooms[rand].transform.rotation);
            }
            if (direction == Direction.Bot)
            {
                rand = Random.Range(0, variants.BotRooms.Length);
                Instantiate(variants.BotRooms[rand], transform.position,
variants.BotRooms[rand].transform.rotation);
            }
            if (direction == Direction.Right)

```

```

        {
            rand = Random.Range(0, variants.RightRooms.Length);
            Instantiate(variants.RightRooms[rand], transform.position,
variants.RightRooms[rand].transform.rotation);
        }
        if (direction == Direction.Left)
        {
            rand = Random.Range(0, variants.LeftRooms.Length);
            Instantiate(variants.LeftRooms[rand], transform.position,
variants.LeftRooms[rand].transform.rotation);
        }
        spawned = true;
    }

}

void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("RoomPoint")) &&
other.GetComponent<RoomSpawner>().spawned==true)
    {
        Destroy(gameObject);
    }
}
}

```

A.16 Текст скрипта RoomVariants.cs

```

using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Security.Cryptography;
using UnityEngine;

public class RoomVariants : MonoBehaviour
{
    [SerializeField] public GameObject[] TopRooms;
    [SerializeField] public GameObject[] BotRooms;
    [SerializeField] public GameObject[] LeftRooms;
    [SerializeField] public GameObject[] RightRooms;
}

```

A.17 Текст скрипту SlimeDamage.cs

```

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

public class SlimeDamage : MonoBehaviour
{
    public Knight knight;
    void Awake()
    {
        knight
GameObject.FindWithTag("Player").GetComponent<Knight>();
    }
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "Player")
        {
            knight.TakeDamage(1);
        }
    }
}

```

=

A.18 Текст скрипту SlimeTargetRange.cs

```

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

public class SlimeTargetRange : MonoBehaviour
{
    public GreenSlime slime;
    void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            slime.isRange = true;
        }
    }
}

```

A.19 Текст скрипту WallAppear.cs

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

public class WallAppear : MonoBehaviour
{
    public GameObject objectToSpawn;
    public GameObject objectToSpawntwo;
    public GameObject Boss;
    public bool isactive = false;
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            isactive = true;
            objectToSpawn.SetActive(true);
            objectToSpawntwo.SetActive(true);
            Boss.SetActive(true);
        }
    }
}
```

ДОДАТОК Б
Слайди презентації

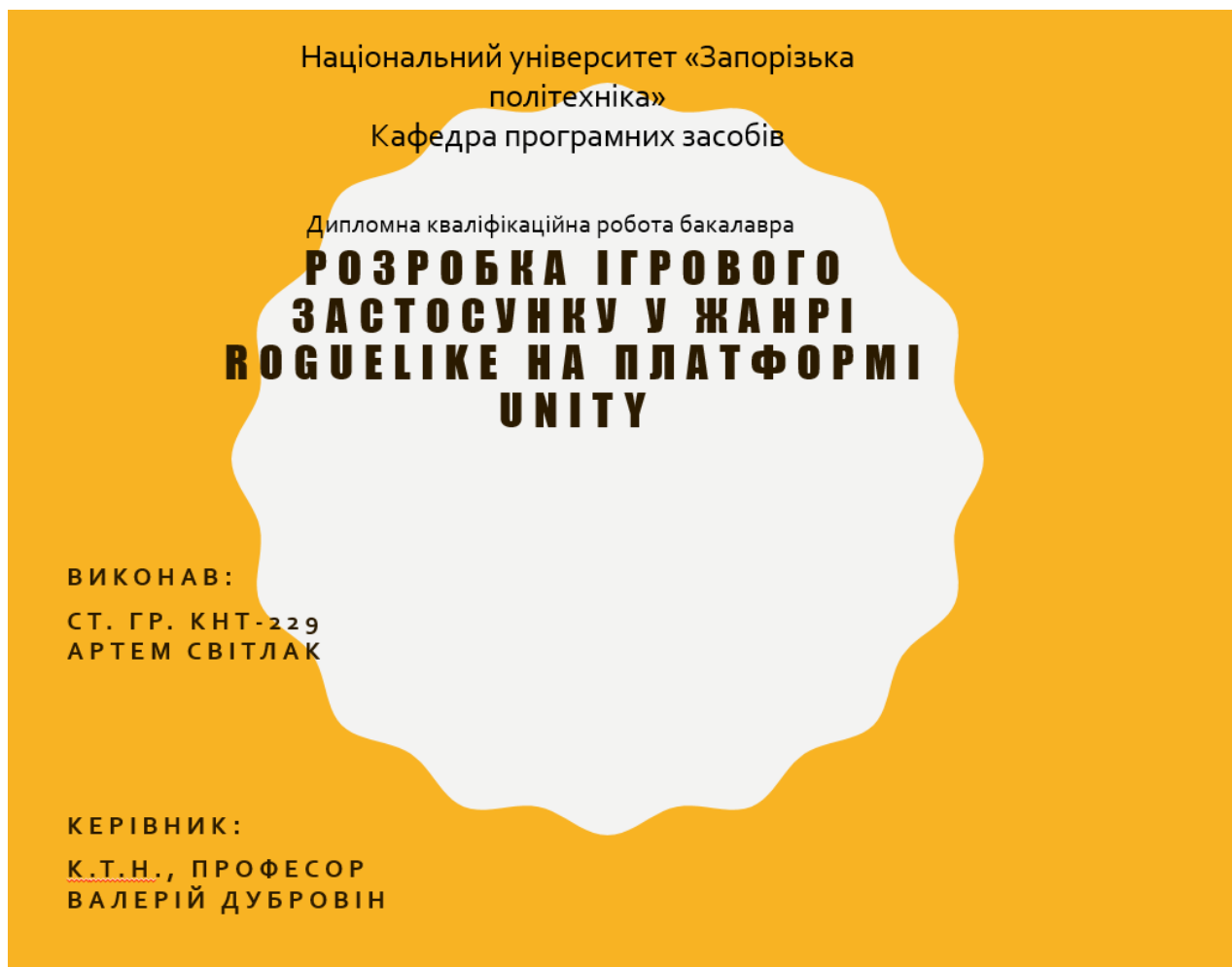


Рисунок Б.1 – Слайд 1

ОБ'ЄКТ, ПРЕДМЕТ, МЕТА ТА ЗАДАЧА РОБОТИ

Об'єкт дослідження – процес розробки ігрового додатку у жанрі Roguelike.

Предмет дослідження – Розробка ігрового застосунку у жанрі Roguelike на платформі Unity.

Мета роботи – Розробка ігрового застосунку у жанрі Roguelike на платформі Unity із наступними функціями:

- можливість взаємодії через ігрового персонажа;
- декілька видів меню;
- супротивника із штучним інтелектом;
- взаємодія між сценами;
- випадкова генерація рівнів.

ВИБІР ЖАНРУ ГРИ

Критерії порівняння при розробці	Ігрові жанри		
	Roguelike	<u>Metroidvania</u>	<u>Платформер</u>
Створення ігрових локацій	Велика кількість кімнат які з'єднані між собою	Великий і продуманий світ із своєю історією	Невеликі локації, велика кількість цих локацій
Програмування	Випадкова генерація рівнів, предмети або навички які будуть відкриватися після кожного проходження	Велика кількість механік які будуть відкриватися з проходженням гри	Потрібні лише базові елементи цього жанру
Артодизайн	Піксельна графіка, піксельна анімація здібностей та ефектів	Деталізовані <u>арти</u> областей та персонажів, анімації здібностей та ефектів	Розробка яскравого <u>арту</u> рівнів, персонажів, взаємодії з оточенням

3

Рисунок Б.3 – Слайд 3

ВИБІР МОВИ ПРОГРАМУВАННЯ

Критерії порівнянь	Мова програмування	
	C#	C++
Синтаксис	+	–
Продуктивність	Менша <u>ресурсоємність</u>	Більша швидкодія
<u>Кросплатформеність</u>	+-	+
Швидкість розробки	+	-
Графічний інтерфейс	+-	+

Рисунок Б.4 – Слайд 4

ВИБІР ПЛАТФОРМИ

Критерії порівнянь	Ігровий рушій	
	Unity	Unreal Engine
Мова програмування	C#, <u>Javascript</u>	C++, Blueprint
Графічний інтерфейс	Простий та зручний у використанні	Більш потужний, але складніший у використанні
Візуальний редактор	+-	+
Анімації	+	-
Активне співтовариство	+-	+
Ціна	Безкоштовна якщо гра за рік заробила менше 100 тис. доларів	Безкоштовна, якщо гра за квартал заробила менше 3 тис. доларів

5

Рисунок Б.5 – Слайд 5

ВИКОРИСТАНІ МОДУЛІ

№	Назва модуля	Застосування
1	Unity Editor	Створення та налаштування гри
2	Physics	Фізичний рушій
3	Scripting	Налагодження <u>скриптів</u>
4	Graphics	Графічний рушій
5	Animations	Анімаційний рушій
6	Audio	Звуковий рушій
7	UI	Створення інтерфейсів
8	<u>Cinemachine</u>	Різні види камер

Рисунок Б.6 – Слайд 6

КОНЦЕПЦІЯ ГРИ

- у грі є 2 локації. 1 локація буде проміжною і у ній буде з'являтися головний герой після завершення забігу, 2 локація буде ігровою, вона буде містити супротивників та випадкову генерацію рівнів;
- будуть декілька видів меню. Головне меню та меню налаштувань у 1 сцені, меню паузи для 2 сцени, меню паузи для 3 сцени і меню смерті та перемоги. Також меню налаштувань будуть у всіх сценах;
- персонаж, із можливістю руху, атаки та отриманням пошкоджень від супротивників;
- вороги, які також здатні атакувати та отримувати пошкодження, також вони будуть переслідувати головного героя за скриптом. Також буде бос;
- можливість завершити гру виходом, смертю чи перемогою.

Рисунок Б.7 – Слайд 7

СЦЕНИ

Сцени містять об'єкти гри. Вони використовуються для створення меню та окремих рівнів. Кожну сцену можна вважати окремим ігровим рівнем.

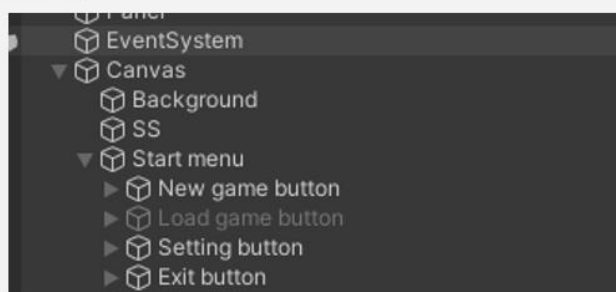
У грі будуть 3 сцени:

- початкове меню;
- проміжна локація;
- ігрова локація.

Рисунок Б.8 – Слайд 8

ІНТЕРФЕЙС

Для створення інтерфейсу потрібен Canvas та EventSystem.



9

Рисунок Б.9 – Слайд 9

МУЗИКА ТА ЗВУКИ

Створюється за допомогою Audio Source.

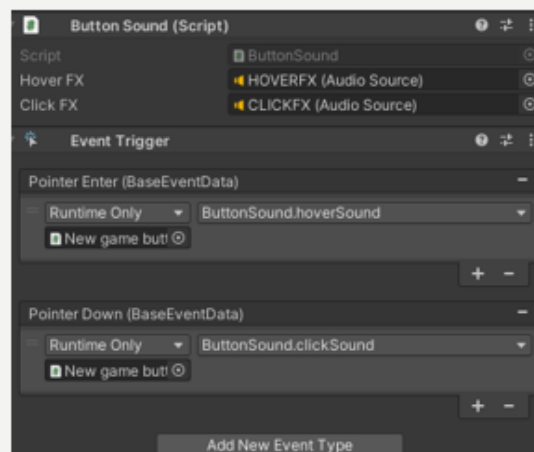
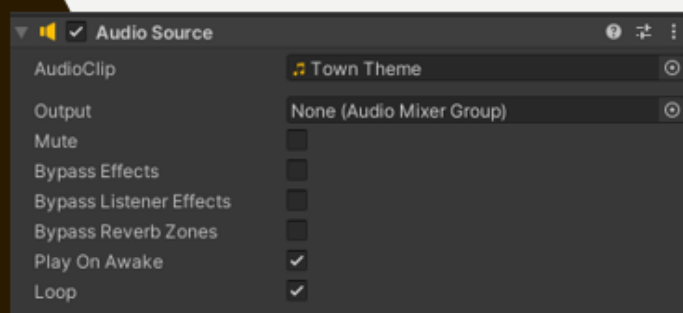
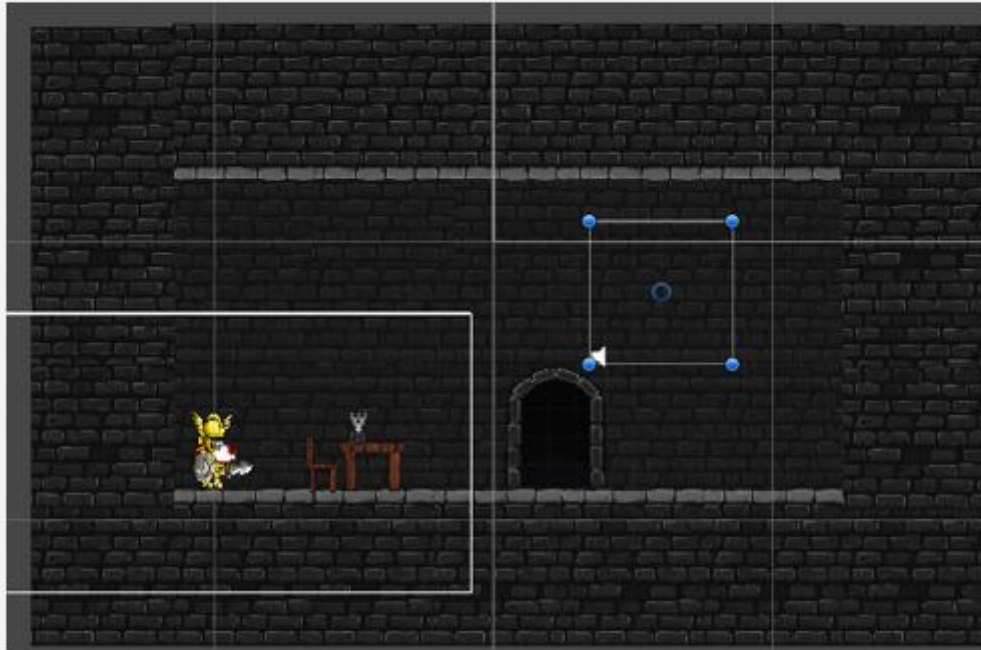


Рисунок Б.10 – Слайд 10

ІГРОВА ЛОКАЦІЯ

Ігрова локація складається із твердих об'єктів такі як підлога та заднього фону. Для заднього фону потрібен лише спрайти, а для твердих поверхонь ще потрібно додати колайдер.

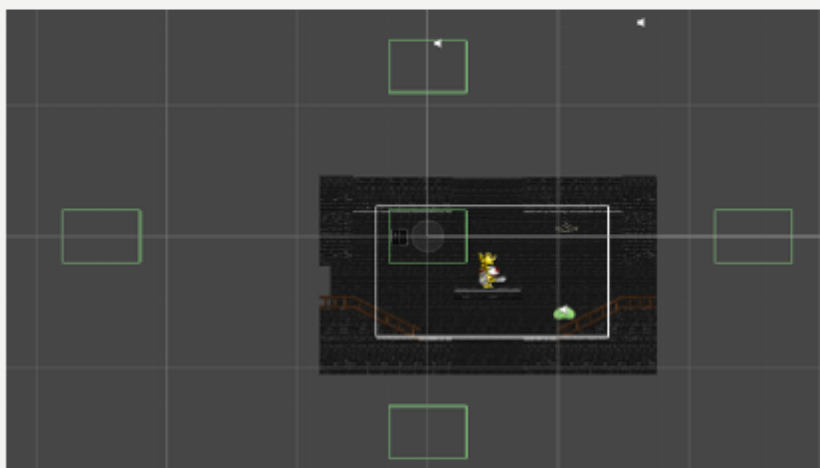


11

Рисунок Б.11 – Слайд 11

ВИПАДКОВА ГЕНЕРАЦІЯ РІВНІВ

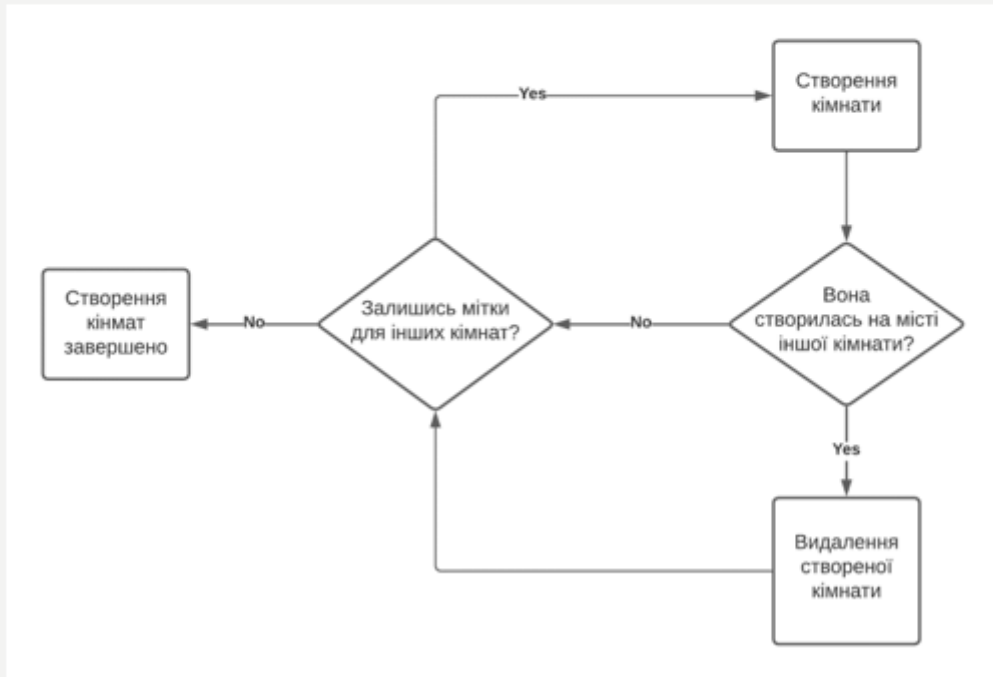
Для випадкової генерації рівнів потрібно спочатку створити кімнату із мітками з усіх сторін та зробити префаб цієї кімнати. Після цього створити довільно кількість кімнат. Мітка повинна знаходитись на виході із кімнати.



12

Рисунок Б.12 – Слайд 12

АЛГОРИТМ РОБОТИ ВИПАДКОВОЇ ГЕНЕРАЦІЇ РІВНІВ

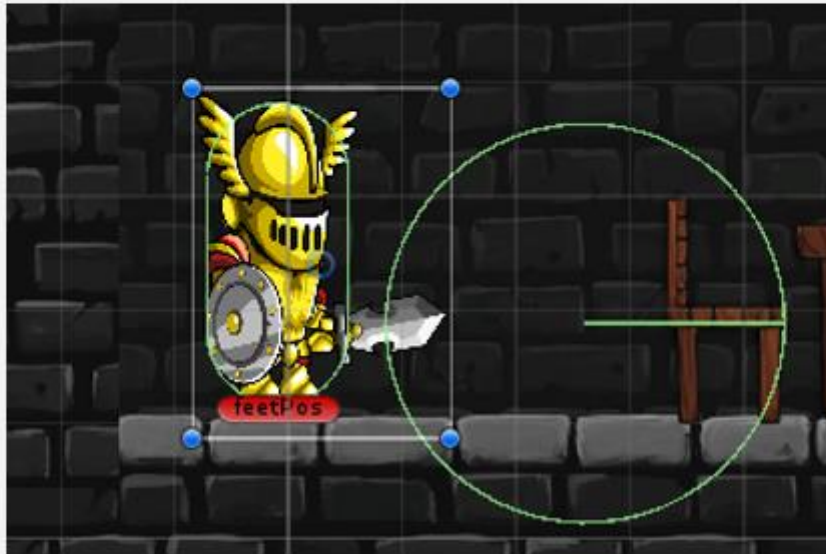


13

Рисунок Б.13 – Слайд 13

НАЛАШТУВАННЯ ПЕРСОНАЖА

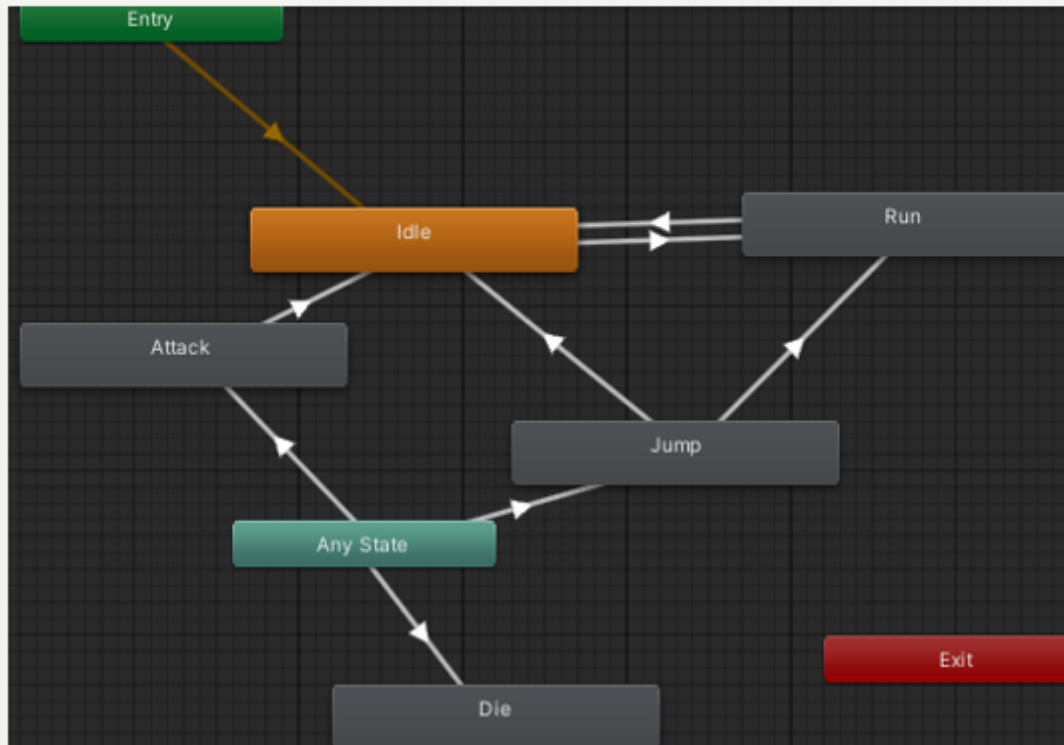
Окрім спрайта та колайдера у нього буде Rigidbody. Також потрібно створити тег та шар "Player", мітку біля ніг, зону для атаки та стеження за об'єктом. Також потрібно створити анімації.



14

Рисунок Б.14 – Слайд 14

АНІМАЦІЇ ПЕРСОНАЖА



15

Рисунок Б.15 – Слайд 15

СЛАЙМ

Створення слайму більш-менш схоже на створення героя, однак рухатися він буде лише коли герой потрапить до триггеру руху.

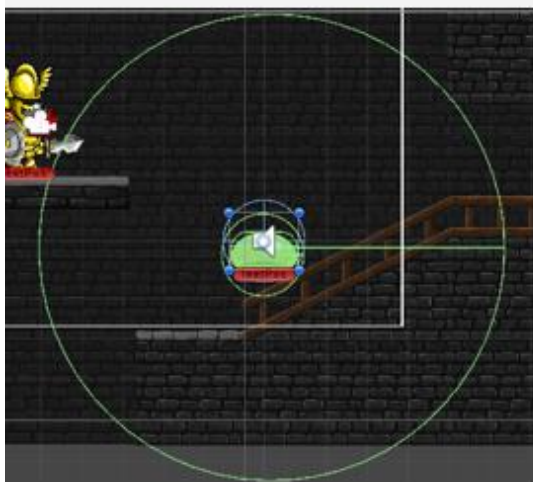


Рисунок Б.16 – Слайд 16

СТРУКТУРА ВЗАЄМОДІЇ У ГРІ

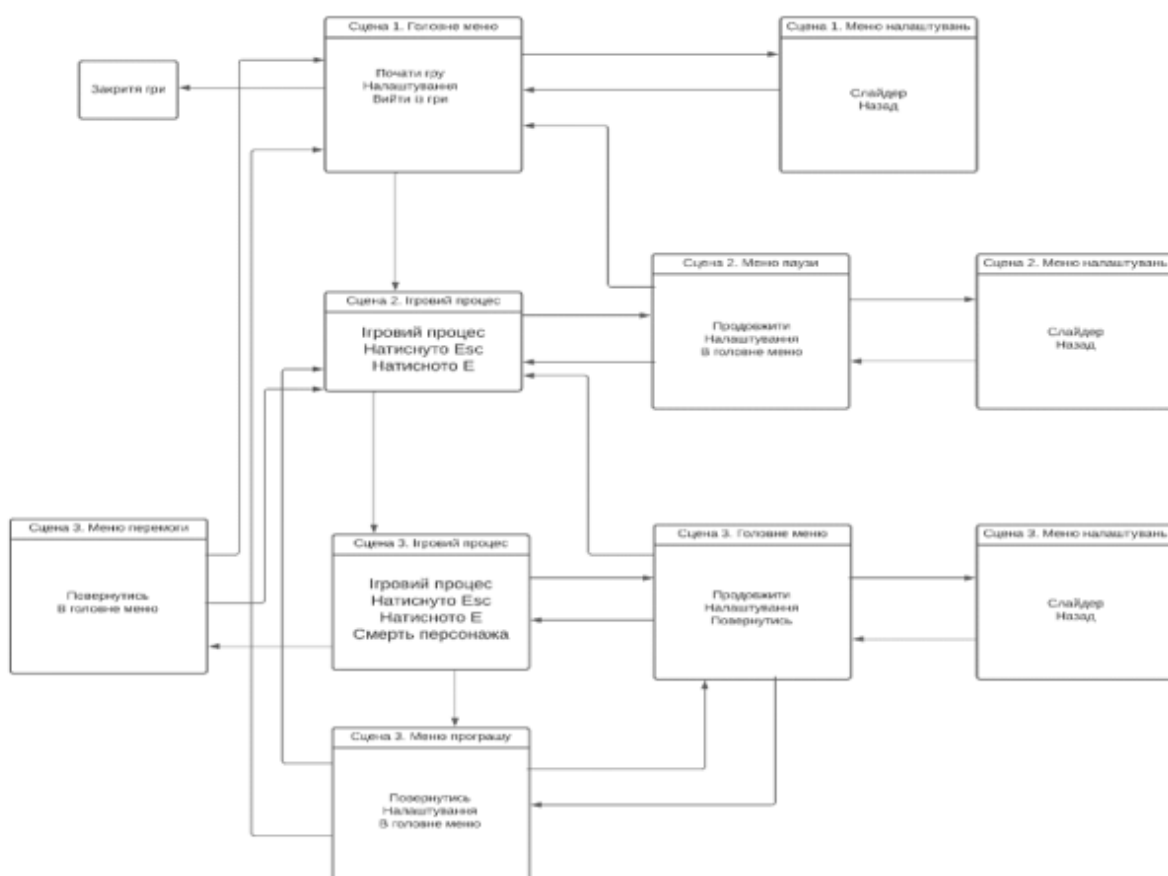
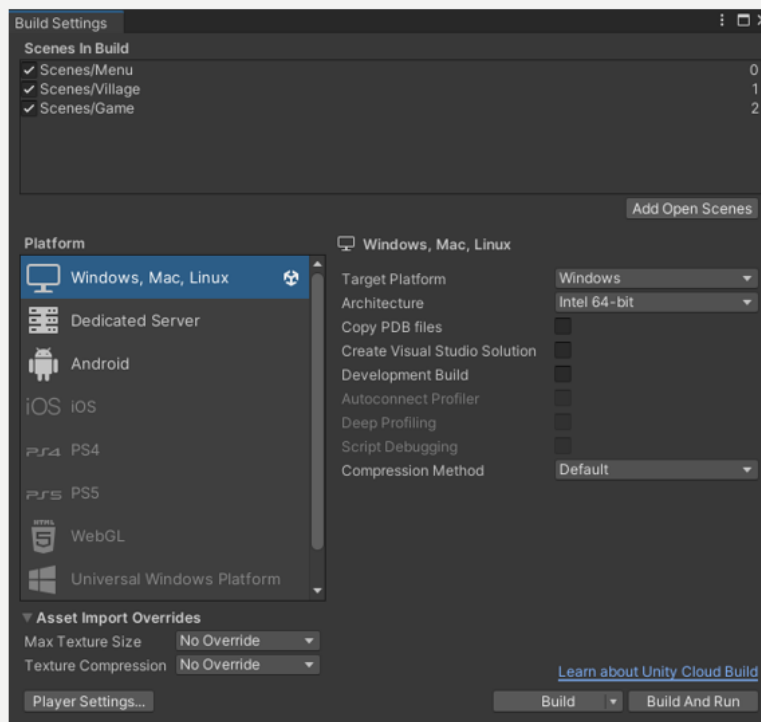


Рисунок Б.17 – Слайд 17

ЗАВЕРШЕННЯ РОЗРОБКИ ГРИ



18

Рисунок Б.18 – Слайд 18

ВИСНОВКИ

В ході виконання даної дипломної кваліфікаційної роботи було проаналізовано та досліджено процес розробки гри у жанрі Unity на платформі розробки.

Гра має наступні функції:

- 2 локації, початкова та ігрова;
- декілька видів меню. Початкову меню, меню налаштувань, меню паузи смерті та перемоги;
- персонаж, із можливістю руху, атаки та смерті. Також він має звук та анімації;
- вороги, які мають те саме, що й персонаж. але рухаються за допомогою штучного інтелекту.

Посилання на відео:

<https://www.youtube.com/watch?v=j2aPsVsNhDY>

Рисунок Б.19 – Слайд 19

ДОДАТОК В

Посилання на відео з ігровим процесом

Посилання- <https://www.youtube.com/watch?v=j2aPsVsNhDY>.



Рисунок В.1 – 8 секунда відео

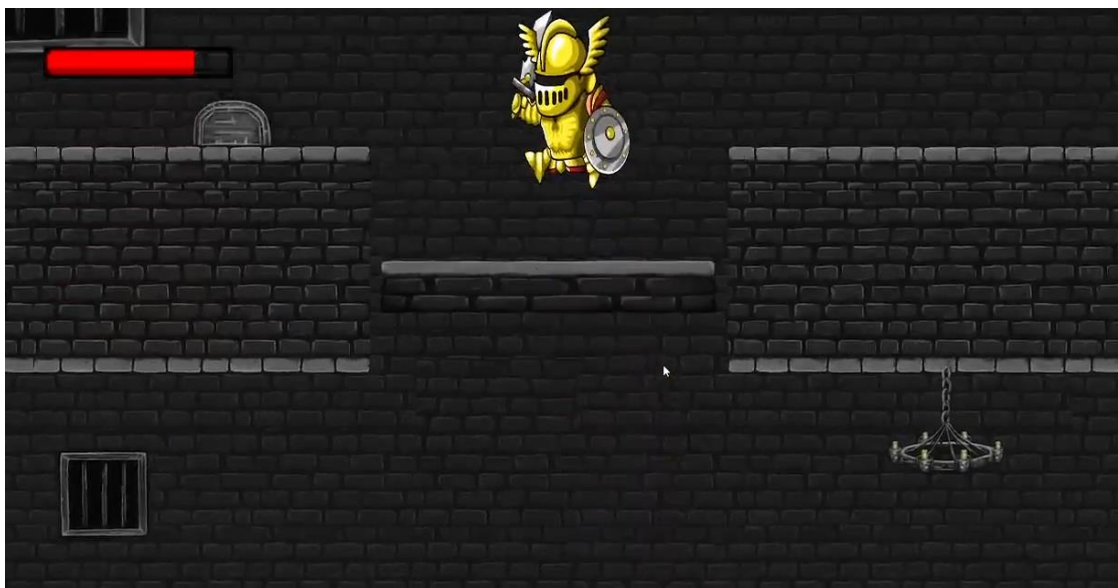


Рисунок В.2 – 30 секунда відео



Рисунок В.3 – 55 секунда відео