**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**Національний університет «Острозька академія»**

**Економічний факультет**

**Кафедра економіко-математичного моделювання та інформаційних технологій**

# КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

на тему: «**Розробка гри в жанрі RogueLike на ігровому рушії Unity »**

**Виконав:** студент 4 курсу, групи КН-42
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп’ютерні науки
освітньо-професійної програми «Комп’ютерні науки»
Самков Олександр Сергійович

**Керівник:** Шатний С.В.
кандидат технічних наук, доцент кафедри ЕММІТ

**Рецензент: Рецензент:** *кандидат технічних наук, доцент, доцент кафедри прикладної математики та кібербезпеки Донецького національного університету імені Василя Стуса*

***Загоруйко Любов Василівна***

***РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ***

Завідувач кафедри економіко-математичного моделювання та інформаційних технологій ______________ (проф., д.е.н. Кривицька О.Р.)

Протокол № 11 від « 30 » травня 2024 р.

Острог, 2024

Міністерство освіти і науки України

Національний університет «Острозька академія»

Факультет: *економічний*

Кафедра: *економіко-математичного моделювання та інформаційних технологій*

Спеціальність: 122 Комп'ютерні науки

Освітньо-професійна програма: Комп'ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри

___________ Ольга КРИВИЦЬКА

«____» ____________20__ р.

ЗАВДАННЯ

**на кваліфікаційну роботу студента**

Самкова Олександра Сергійовича

*1. Тема роботи:* “Розробка гри в жанрі RogueLike на рушії Unity”

*Керівник роботи:* Шатний Сергій В’ячеславович, кандидат технічних наук, доцент кафедри ЕММІТ

*Затверджено наказом ректора НаУОА від від 03.11.2023 р., № 98.*

*2. Термін здачі студентом закінченої роботи:* 31 травня 2024 року.

*3. Вихідні дані до роботи: Unity, C#, Visual Studio Code*

*4. Перелік завдань, які належить виконати: обрати асети(*спеціальний файл, який використовується в Unity, програмі для створення відеоігор) *для застосування, створити концепт карти, створити механіку та анімації до руху, створити механіки бойової системи, створення взаємодії з предметами, додавання аудіосупроводу, створення локацій та наповнення, створення міні-карти, створення ворогів, створення штучного інтелекту для ворогів, створення системи збереження, створення можливості розвитку головного героя, створення шкали здоров’я, створення звукових ефектів, створення меню.*

*5. Перелік графічного матеріалу:* рисунки, таблиці, діаграм

*6. Консультанти розділів роботи:*

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | Завдання видав | Завдання прийняв |
| 1 | Шатний С.В. | 01.12.2023 | 01.12.2023 |
| 2 | Шатний С.В. | 01.12.2023 | 01.12.2023 |
| 3 | Шатний С.В. | 01.12.2023 | 01.12.2023 |

*7. Дата видачі завдання:*

## КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів кваліфікаційної роботи | Строк виконання етапів | Примітка |
|---|---|---|---|
| 1 | Затвердження теми проєкту | до 31.10.2023 | |
| 2 | Постановка технічного завдання | до 01.12.2023 | |
| 3 | Створення концепту гри | до 10.12.2023 | |
| 4 | Створення базового функціоналу | до 12.12.2023 | |
| 5 | Створення базової структури. | до 12.12.2023 | |
| 6 | Підключення необхідних бібліотек | до 01.03.2024 | |
| 7 | Верстка елементів клієнтської частини. | до 01.03.2024 | |
| 8 | Створення Меню | до 01.03.2024 | |
| 9 | Оновлення дизайну певних компонентів. | до 01.03.2024 | |
| 10 | Додавання засобів тестування до проєкту. | до 20.03.2024 | |
| 11 | Тестування гри | до 20.03.2024 | |
| 12 | Створення полоси здоров'я | до 20.03.2024 | |
| 13 | Додавання курсорів | до 01.04.2024 | |
| 14 | Тестування гри | до 01.05.2024 | |
| 15 | Виправлення помилок | до 20.04.2024 | |
| 16 | Попередній захист кваліфікаційної роботи | до 31.05.2024 | |
| 17 | Здача кваліфікаційної роботи на кафедру | до 31.05.2024 | |

**Студент:** ________ Олександр САМКОВ

**Керівник кваліфікаційної роботи:** ________ Сергій ШАТНИЙ

# АНОТАЦІЯ
## кваліфікаційної роботи
## на здобуття освітнього ступеня бакалавра

***Тема:*** Розробка гри в жанрі RogueLike на рушії Unity

***Автор:*** *<u>Самков Олександр Сергійович</u>*

***Науковий керівник:*** *Шатний С.В.,кандидат технічних наук, доцент кафедри ЕММІТ*

***Захищена «………»………………… 2024 року.***

***Пояснювальна записка до кваліфікаційної роботи:***

***Ключові слова:*** Rogue-like, Unity, гра, процедурна генерація

***Короткий зміст праці:***

*У цій кваліфікаційній роботі було розроблено Rogue-Like гру на рушії Unity із реалізацією процедурної генерації. У прогресі розробки було використано такі засоби як рушій Unity, засіб для написання коду Visual Studio, магазин асетів Unity Asset Store та мову програмування C#.*

*Метою було створення гри для любителів жанру Rogue-Like та для новачків на рушії Unity. В звіті показано різні види механік та як вони створювались для цікавого геймплею та залучення гравців.*

***SUMMARY***

*Keywords: Rogue-like, Unity, game, procedural generation*

*In this qualifying work, a Rogue-Like game was developed on the Unity engine with the implementation of procedural generation. In the development progress, such tools as the Unity engine, the tool for writing Visual Studio code, the asset store Unity Asset Store and the C# programming language were used.*

*The goal was to create a game for fans of the Rogue-Like genre and for beginners on the Unity engine. The report shows different types of mechanics and how they were created for interesting gameplay and player engagement.*

______________

Зміст

ВСТУП 6
**РОЗДІЛ 1 ЗАГАЛЬНІ ПОЛОЖЕННЯ 9**
1.1 Аналіз жанру RogueLike та основні методи розробки" 9
1.2 Платформа для розробки 11
1.3 Задачі дослідження: 15
1.4 Процедурна генерація. Типи, методи реалізації, патерни. 17
1.5 Постановка завдання кінцевого продукту 18
Висновки до розділу 1 18
**РОЗДІЛ 2 ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ 19**
2.1 Аналіз предметної області 19
2.2 Архітектура проєкту **21**
Висновки до розділу 2 25
**РОЗДІЛ 3 Програмне та технічне забезпечення 26**
3.1 Засоби розробки 26
3.2 Вимоги до технічного та програмного забезпечення 26
3.3 Програмна реалізація 27
Висновки до розділу 3 52
**ВИСНОВКИ 54**
**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 55**
**ДОДАТКИ 58**

# ВСТУП

**Актуальність** роботи підтверджується тим що, сучасному світі галузь розробки ігор постійно збільшується та розвивається, привертаючи увагу як досвідчених розробників, так і любителів. Вже тривалий час, розробка ігор в стилі RogueLike на платформі Unity стає особливо актуальною та захоплюючою галуззю, та не втрачає своєї популярності та аудиторії, навпаки аудиторія - зростає.

Вибір теми "Розробка гри на рушії Unity в стилі RogueLike" обумовлений не лише моєю жагою до геймдизайну, але й актуальністю цього напрямку. В такому інноваційному середовищі, як Unity, розробка гри в стилі RogueLike відкриває безмежні можливості для технічного та творчого розвитку. Сполучення процедурної генерації, випадковості та унікальних геймплейних елементів дозволяє створювати захоплюючі та непередбачувані відчуття для гравців. Вивчення цієї теми дозволить розкрити та закріпити знання та нові можливості Unity для розробки, які об'єднують в собі випадковість, стратегію та елементи розвитку персонажів.

**Метою** роботи є: Розробка та вдосконалення методів процедурної генерації контенту в ігровому рушії Unity з метою створення динамічних та унікальних ігрових світів. Основна мета полягає в підвищенні варіативності ігрового досвіду, забезпеченні високого рівня цікавості гравців та оптимізації використання ресурсів гри. Дослідження передбачає аналіз існуючих підходів до процедурної генерації, визначення їхніх переваг та недоліків, а також розробку нових стратегій, специфічно адаптованих до особливостей інтеграції з Unity.

Додатковими завданнями є вивчення оптимальних методів інтеграції з іншими аспектами геймдизайну, такими як сценарії, рівні та механіки гри. Також важливим етапом дослідження є вдосконалення алгоритмів генерації з урахуванням потреб різних жанрів ігор.

Також, варто врахувати важливість створення захоплюючого інтерфейсу, розробки цікавих сценаріїв гри та використання сучасних технічних рішень для оптимізації роботи гри. Усе це дасть можливість зануритися у фундаментальні

аспекти створення RogueLike ігор, розглядаючи їх з поглибленим аналізом та практичними прикладами відновлення.

Звертаючи увагу на вплив ігор в стилі RogueLike на індустрію розваг та взаємодію з гравцями, ми вивчимо важливість інноваційних підходів та творчості в розробці героїчних ігор. У результаті дослідження ми будемо мати можливість не лише розуміти технічні аспекти розробки RogueLike гри на Unity, але й визначити їх вплив на геймдизайн, взаємодію гравця та змістовність гри в цілому.

**Предметом дослідження**: є розробка процедурної генерації в ігровому рушії Unity з фокусом на створення гри у стилі RogueLike. Дослідження охоплює генерацію рівнів, об'єктів, характеристик персонажів, а також взаємодії між цими елементами з метою створення унікальних та цікавих геймплейних сценаріїв. Вивчення оптимальних підходів до реалізації процедурної генерації в контексті RogueLike гри, включаючи аналіз інших ігор у цьому жанрі та ідентифікацію ключових елементів.

**Об'єктом дослідження:**є технології: розробка гри в жанрі Rogue-Like, ігровий рушій Unity, редактор Adobe Illustrator, мова програмування C#

**Методами дослідження** визначено перелік завдань, що виступатимуть у ролі плану виконання роботи:

1. ***Вивчення матеріалів та існуючих підходів:***
   - Провести аналіз матеріалів по розробці та існуючих ігор у жанрі RogueLike, звертаючи увагу на методи та підходи до процедурної генерації контенту.
2. ***Вивчення можливостей Unity для процедурної генерації:***
   - Ознайомитися з інструментами та можливостями, які надає Unity для реалізації процедурної генерації, зокрема генерації рівнів та об'єктів.
3. ***Розробка архітектури генератора:***
   - Створити архітектуру для генератора, визначивши основні компоненти та їх взаємодії, зокрема алгоритми генерації рівнів, об'єктів та характеристик персонажів.

4. ***Реалізація алгоритмів генерації:***
   - Розробити та реалізувати алгоритми для процедурної генерації різноманітних елементів гри, забезпечуючи їхню взаємодію та взаємодію з гравцем.
5. ***Оптимізація та адаптація відповідно RogueLike стилістики:***
   - Впровадження оптимізації, специфічної стилістики для RogueLike гри, такої як створення лабіринтів, розташування об'єктів та розміщення ворогів з метою створення характерної геймплейної динаміки.
6. ***Валідація та тестування:***
   - Провести систематичне тестування розроблених алгоритмів генерації, перевірити відповідність створеного контенту вимогам RogueLike жанру та забезпечити баланс гри.
7. ***Інтеграція з геймплейними елементами:***
   - Забезпечити ефективну інтеграцію з іншими геймплейними елементами, такими як системи управління гравцем, бойові механіки та системи прогресу, для створення повноцінного RogueLike геймплею.
8. ***Аналіз результатів.***

# РОЗДІЛ 1
# ЗАГАЛЬНІ ПОЛОЖЕННЯ

## 1.1 Аналіз жанру RogueLike та основні методи розробки"

У відповідь на зростаючу цікавість гравців до відповідальних ігор та збільшення популярності незалежних розробників, постає проблема створення нових rogue-like ігор, які б забезпечували високий рівень задоволення гравців. Ця проблема виникає з вимогою до ігрового контенту, який пропонує не лише захоплюючий геймплей, але й унікальний досвід відтворення, що дозволяє кожній грі бути неповторною. Із врахуванням швидкого темпу розвитку індустрії важливо виявляти нові підходи та механіки, які б збагатили жанр rogue-like і привернули нових гравців. Таким чином, існує потреба у вивченні інноваційних методів розробки ігор, спрямованих на створення нових rogue-like ігор, що задовольняють сучасні потреби гравців і відповідають їх високим очікуванням.

**Основними елементами гри у жанрі RougueLike є:**

Випадкова генерація рівнів (процедурна генерація): при кожній новій грі рівні, мапи та інші елементи гри генеруються випадковим чином. Це створює сценарії, які є унікальними та непередбачуваними для кожного гравця.

Перманентна смерть, також відома як Permadeath: одна з найбільш помітних характеристик RogueLike. Перманентна смерть означає, що коли персонаж помирає, гравець повинен розпочати гру знову. Це підвищує рівень виклику в грі, а також спонукає до роздумів про стратегію.

Покроковий геймплей: більшість RogueLike-ігор грають послідовно, з кожним гравцем і ворожим ходом. Це робить гру більш стратегічною, а гравцеві дає час подумати про те, як вони будуть діяти.

ASCII-графіка: Більшість RogueLike-ігор використовують ASCII-символи, щоб відобразити персонажів, предмети та навколишнє середовище. Це надає жанру особливий візуальний стиль і є одним з його особливостей.

Система персонажів і розвиток (система персонажів і прогресії): зазвичай гравець керує персонажем, який має особливі властивості та навички. Система рівнів, набір навичок, придбання нового обладунку, може відображатись прогрес персонажа.

Текстові діалоги (Text-Based Interaction): У RogueLike-іграх гравці зазвичай отримують інформацію через текстові описи та діалоги. Це може включати описи навколишнього середовища, розмови з некерованими персонажами (NPC) та інші інтерактивні елементи.

## Конкуренти

- "Dead Cells"

«Dead Cells» — це RogueVanilla, яка використовує елементи RogueLike і метроїдванію. Проходячи через створені рівні, гравець може розвивати свого персонажа та боротися з ворогами в режимі реального часу. (Метроїдваній - піджанр пригодницького бойовика з набором елементів ігрової механіки та ігровим процесом, подібним до серій ***Metroid* *і* *Castlevania*)**

Розробник: Motion Twin

- "Hades"

Опис: "Хадес" - це RogueLike з акцентом на історії. Гравець потрапляє в Ад, де він грає за сина бога Грецького пантеону. Збираючи сили, він має намагатися уникнути ворогів, які виникають на шляху.

Розробник: Supergiant Games

- "FTL: Faster Than Light"

Опис: «FTL» — це космічний RogueLike, де гравець керує космічним кораблем, щоб вижити в небезпечному космосі. Відмінною рисою гри є її стратегічна складність і різноманітність випадкових подій.

Розробник: Subset Games.

- "Enter the Gungeon"

Опис: «Enter the Gungeon» — це булет-хелл RogueLike, де гравець вирушає в печеру, повну загадкових предметів і небезпечних ворогів. Ступінь виклику та швидкість є ключовими характеристиками гри.

Розробник: Dodge Roll

## 1.2 Платформа для розробки

Інструмент для розробки відеоігор і застосунків **Unity -** це повноцінний рушій, багатоплатформний інструмент, спрямований на створення 2D та 3D застосунків в одному редакторі.

На веб-сайті Unity доступні різні тарифні плани рушія, які відрізняються рівнем підтримки та вартістю.Кожна з версій цього програмного забезпечення надає користувачеві нові можливості, які не були доступні в базовій версії.

## Серед переваг ігрового рушія Unity можна виділити:

- Простий редактор і інструментарій: навіть новачки в розробці застосунку можуть освоїти основи за пару днів. Багато ресурсів, форумів і відеоуроки на YouTube мають відповіді на ваші запитання. Навіть студенти можуть створити додаток на Unity.
- Сучасна графіка, яка може конкурувати з більш дорогими рушіями Хоча Unity не має таких можливостей, як UnrealEngine, вона може працювати з освітленням і стандартним набором постпроцессингових ефектів, відомим як SSAO.
- Unity Game Launcher поширюється умовно безкоштовно. Розширення підписки не вимагає додаткових витрат. Знижки на ліцензії зазвичай становлять 20% щороку.
- Значна спільнота розробників
- Багато вільних додатків.

  - Внутрішній магазин готових рішень, де можна купити готові звуки, текстури та фрагменти коду
  - Можливість робити фотореалістичну графіку.
- Створені програми можна легко імпортувати за допомогою розробки на Unity. ОС Windows, Linux, OS X, Android, iOS, на консолі PlayStation, Xbox, Nintendo, на VR- і ARпристрої.
- **Мультиплатформність**: Unity дозволяє розробникам створювати ігри для різних платформ, включаючи комп'ютери, консолі, мобільні пристрої, віртуальну реальність тощо. Unity є актуальним інструментом для розробників, які хочуть розповсюдити свої ігри на широку аудиторію.
- **Спільнота:** У Unity є велика спільнота розробників, тому можна отримати підтримку, допомогу та отримати доступ до багатьох ресурсів, таких як документація, онлайн-курси та форуми.
- **Зручний інтерфейс:** Unity має інтуїтивний інтерфейс, що дозволяє легко створювати та редагувати ігрові об'єкти та сцени.
- **Графічний потенціал**: Unity підтримує розробку графічного контенту, включаючи 2D та 3D графіку, а також різні спеціальні ефекти.
- **Фізика та анімація:** Unity надає інструменти для створення реалістичних фізичних ефектів та анімацій об'єктів.
- **Можливості програмування:** Розробка ігор в Unity відбувається з використанням мови програмування C#, що дає розробникам широкий простір для програмування ігрової логіки.

**Основи аспекти ігрового рушія Unity при розробки ігор на**

Unity містить декілька ключових деталей, що дозволяють розробникам створювати ігри та інтерактивні додатки. Далі можна прочитати та зрозуміти основні деталі, та загалом, як працює ігровий рушій Unity

**Ігрові об'єкти та сцени:**

- **Ігрові об'єкти:** Unity працює з концепцією "ігрових об'єктів", які є основними елементами вашої гри. Це може бути герой, ворог, вежа, предмет, та інше. Кожен об'єкт містить компоненти, що визначають його властивості та поведінку.
- **Сцени**: Сцена в Unity являє собою віртуальний ігровий світ, де ігрові об'єкти розміщуються, взаємодіють та відображаються. Розробники можуть створювати сцени для різних рівнів гри.

**Компоненти та скрипти:**

- **Компоненти**: Компоненти - це будівельні блоки ігрових об'єктів, які визначають їхню поведінку та властивості. Unity має декілька вбудованих компонентів, таких як Transform, Collider, Rigidbody та інші, а також дає розробникам створювати власні компоненти.
- **Скрипти**: Розробники використовують скрипти для програмування логіки та поведінки ігрових об'єктів. Скрипти дають можливість створювати ігровий інтелект, керувати анімацією, обробляти дії гравця та багато чого іншого.

**Фізика та анімація:**

- Unity має вбудовану систему фізики, яка дозволяє симулювати рух та зіткнення об'єктів в грі, що робить світ гри реалістичнішим та дозволяє створювати фізичні головоломки та ефекти.
- Unity також підтримує анімацію об'єктів, де розробники можуть створювати рухи та анімацію для персонажів, об'єктів та інших ігрових елементів.

**Мультимедіа та ресурси:**

- Unity дозволяє імпортувати графіку, звуки, відео та інші мультимедійні ресурси для використання в грі. Розробники мають можливість налаштувати освітлення, матеріали та текстури для досягнення бажаного стилю гри.
- Засоби ресурсів Unity дозволяють організовувати та керувати активами, що спрощує роботу з ресурсами гри.

**Віртуальна камера:**

- В Unity можна налаштовувати камери для відображення гри з різних ракурсів та кутів огляду. Це дозволяє створювати різноманітні ефекти камери, включаючи зум, повороти та зміну фокуса.

**Взаємодія та фізика зв'язків:**

- Unity надає засоби для взаємодії ігрових об'єктів, включаючи детектори зіткнень та події, які спрацьовують при певних діях гравця чи ігрових об'єктів.

Unity містить потужний рендеринг, що дає можливість створювати візуально захопливі ігри з різними стилями графіки.

## Серед недоліків ігрового рушія Unity можна виділити:

- Розробка додатків на Unity вимагає навичок програмування
- Продукт досить об’ємний завдяки великій кількості вбудованих компонентів. Це може стати проблемою, оскільки користувачі не люблять завантажувати великі додатки. Крім того, в деяких країнах, люди використовують недорогі, слабкі пристрої, які не можуть підтримувати цей застосунок.
- Розробники не можуть отримати доступ до вихідного коду своїх власних програм. Чекайте, поки інженери Unity самі зроблять це. Навіть при купівлі ліцензії вам не нададуть початкових кодів.
- Немає інтеграції з зовнішніми сервісами та бібліотеками, такими як Facebook, тому розробники повинні налаштовувати це вручну.
- Неможливість додати сторонню фізику до рушія.

## 1.3 Задачі дослідження:

1. **Технічні аспекти:**

- Розробка та реалізація архітектури гри на Unity.
- Вивчення можливостей Unity для створення процедурно генерованих рівнів.
- Оптимізація гри для роботи на різних платформах.

2. **Геймдизайн:**

- Дослідження ключових елементів RogueLike ігор, таких як випадковість, невизначеність та перманентна смерть персонажа.
- Розробка імплементації системи процедурної генерації рівнів та взаємодії з об'єктами гри.
- Вивчення впливу різних механік гри на геймплей та взаємодію гравців.

3. **Взаємодія з гравцями:**

- Аналіз впливу соціальної взаємодії в грі на гравцівський досвід.
- Вивчення можливостей для многогравцевих режимів в грах в стилі RogueLike.
- Розробка та тестування механік гри, спрямованих на покращення взаємодії з аудиторією.

4. **Інновації та творчість:**

- Аналіз історії та еволюції RogueLike ігор.
- Розробка новаторських геймплейних елементів або механік, які можуть відзначити гру.
- Дослідження впливу використання віртуальної реальності або інших технологій на геймплей RogueLike ігор.

5. **Вивчення аудиторії:**

- Аналіз цільової аудиторії для гри в стилі RogueLike.
- Вивчення попередніх успішних проектів та їх вплив на гравців.
- Створення та проведення опитувань для збору думок та пропозицій гравців.

У цьому розділі ми розглянули ключові технічні аспекти, пов'язані з розробкою гри на Unity. Спершу ми звернули увагу на розробку та реалізацію архітектури гри, а також вивчили можливості Unity для створення процедурно генерованих рівнів. Важливим елементом технічної частини є оптимізація гри для роботи на різних платформах. Ці аспекти формують основу технічної реалізації проекту, забезпечуючи стабільність та продуктивність гри.

***Геймдизайн***

Одним із критичних аспектів для успішної розробки гри є геймдизайн який є критично важливим для успіху гри. Було досліджено ключові елементи RogueLike ігор, зокрема випадковість, невизначеність та перманентну смерть персонажа. Важливим аспектом є розробка імплементації системи процедурної генерації рівнів та взаємодії з об'єктами гри. Також ми вивчали вплив різних механік на геймплей та взаємодію гравців. Ці елементи геймдизайну значно впливають на загальний досвід гравців, формуючи унікальність кожного проходження гри.

***Взаємодія з гравцями***

Наступною важливою темою є взаємодія з гравцями. Ми аналізували вплив соціальної взаємодії в грі на гравцівський досвід та вивчали можливості для багатокористувацьких режимів у RogueLike іграх. Розробка та тестування механік, спрямованих на покращення взаємодії з аудиторією, дозволяє створити більш захоплюючий та інтерактивний ігровий процес.

***Інновації та творчість***

Інновації та творчість є важливими для створення унікальної гри. Ми проаналізували історію та еволюцію RogueLike ігор, розробили новаторські геймплейні елементи та механіки, які можуть відзначити гру серед конкурентів. Дослідження впливу використання віртуальної реальності або інших технологій на геймплей RogueLike ігор допомагає вивести гру на новий рівень.

***Вивчення аудиторії***

Розуміння цільової аудиторії є ключовим для успіху гри. Ми аналізували цільову аудиторію для RogueLike ігор, вивчали попередні успішні проекти та їх вплив на гравців. Створення та проведення опитувань дозволило зібрати думки та пропозиції гравців, що є важливим для подальшого розвитку гри.

У попередніх розділах було неодноразово згадано про процедурну генерацію як важливий елемент розробки ігор. Тепер настав час детальніше зупинитися на цій темі. Процедурна генерація грає ключову роль у створенні унікальних і неповторних ігрових рівнів, забезпечуючи різноманітність і повторюваність ігрового процесу. В цьому розділі ми розглянемо різні типи процедурної генерації, методи їх реалізації та поширені патерни, які використовуються для створення цікавих ігрових світів.

## 1.4 Процедурна генерація. Типи, методи реалізації, патерни.

**Існують різні типи процедурної генерації:**

1) ***Генерація контенту:***

Генерація місць: Створення ландшафту, рівнів або ігрових світів.

Генерація об'єктів: Створення об'єктів в середовищі, таких як рослини, структури або різноманітні декорації.

2) ***Генерація текстур:***

Генерація поверхонь: Автоматичне створення текстур для об'єктів чи ландшафту.

3) ***Генерація правил:***

Генерація правил поведінки: Визначення взаємодії об'єктів або сутностей на основі певних правил.

**Існують певні інструменти для реалізації процедурної генерації в Unity:**

1) ***Unity's Procedural Generation API:***

Unity надає API для процедурної генерації, що включає класи та методи для створення геометричних форм, текстур та інших аспектів ігрового світу.

*2) Noise Functions:*

Використання функцій шуму, таких як Perlin або Simplex noise, для створення випадкових, але неперіодичних значень.

*3) Asset Graphs:*

Використання графічних інструментів, таких як Unity's Shader Graph або власні Asset Graphs, для визначення правил генерації об'єктів та текстур.

**Патерни проєктування для реалізації процедурної генерації в Unity:**

*1) Factory Method:*

Використання фабричних методів для динамічного створення об'єктів залежно від умов та потреб генерації.

*2) Observer Pattern:*

Застосування патерну спостерігача для слідкування за подіями та змінами під час генерації для внесення коректив або взаємодії з іншими компонентами гри.

*3) Decorator Pattern:*

Використання патерну декоратора для динамічного додавання нових функціональностей чи атрибутів до згенерованих об'єктів.

## 1.5 Постановка завдання кінцевого продукту

Ця гра буде цікава для гравців різного віку та для гравців яким цікавий жанр Rogue-Like. Гравці можуть випробувати свої сили в проходженні захоплюючої гри з різноманітними механіками. Гра доступна на PC для Windows.

Для створення гри була використана мова програмування C# та рушій Unity. Концепт та фінальна версія була створення інструментом Unity. Гра розробляється для PC оскільки більшість гравців які зацікавлені в цьому жанрі більшість ігрового часу грають тільки на PC. Завдяки широкому функціоналу гра має залучити різноманітних гравців, незалежно від віку.

## Висновки до розділу 1

У цьому розділі було описано проблематику теми та описано популярність та нюанси ігрового жанру rouge-like. Було проаналізовано основних конкурентів та представлено їх. Розказано про те, що таке процедурна генерація.

# РОЗДІЛ 2

# ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

У цьому розділі ми розглянемо інформаційне та математичне забезпечення, яке є основою проєкту. Детальне опрацювання математичних алгоритмів і моделей є ключовим для забезпечення ефективного функціонування гри. Було проаналізувано різні підходи до алгоритмів процедурної генерації рівнів та методів оптимізації ігрового процесу. Це забезпечує стійку та надійну базу для подальшої розробки ігрових механік та архітектури проєкту.

***Аналіз предметної області***

Переходячи до наступного розділу, важливо звернути увагу на аналіз предметної області. Дослідження цієї області дозволяє зрозуміти контекст та специфіку гри, визначити ключові аспекти, які необхідно врахувати при розробці. Аналіз предметної області включає вивчення існуючих рішень, аналіз ринку та потреб гравців. Це допомагає сформувати чітке уявлення про вимоги до проекту та визначити основні напрямки його розвитку.

## 2.1 Аналіз предметної області

Сьогодні існує множина ігор різних жанрів, і кожен розробник прагне зацікавити користувачів своєю творчістю. У сфері розробки ігор конкуренція надзвичайно висока, і для вирізнення серед інших необхідно створювати щось унікальне й захоплююче, спроможне привернути увагу геймерів. Для досягнення комерційного успіху гри потрібно пройти численні організаційні етапи розробки. Серед них: визначення концепції гри, управління маркетингом і продажами, привертання гравців, розробка технічних аспектів, розподіл бюджету, тестування перед випуском та розклад розробки. Рольові ігри привертають гравців різного віку, які бажають погрузитися у запам'ятовуючуся історію. Однак для створення захоплюючого сюжету необхідно розробити зручний геймплей, що не підірве репутацію гри взагалі.

Для вирішення цієї проблеми необхідно створити гру з таким геймплеєм:

1. Головному герою потрібно знайти вихід із підземелля.
2. На кожному шляхубудуть зустрічатися вороги різного типу.
3. Для подолання ворогів можна підібрати різну зброю та вирушити в подорож.
4. У головного персонажа та відповідно у ворогів є здоров’я, тому задля того щоб добратися до виходу, потрібно звертати увагу на здоров’я та планувати подальші дії.
5. Завжди можна полікуватись, якщо пощастить знайти аптечку.

### 2.1.1 Характерні риси жанру

Одним з ключових аспектів у розробці гри є визначення основних механік. Основні механіки формують основу ігрового процесу, визначають правила взаємодії гравця з ігровим світом. Важливо детально опрацювати кожну механіку, щоб забезпечити гармонійний і збалансований геймплей. Це включає в себе розробку системи процедурної генерації рівнів, визначення параметрів персонажів та їх взаємодію з об'єктами гри.

Планування успішного ігрового проєкту неможливе без аналізу ключових механік та особливостей геймплею. Дослідження механік відомих ігор у даному жанрі дозволяє краще зрозуміти, яким чином вони привертають гравців. Різні вікові групи мають власні уподобання, тому важливо забалансувати геймплей за допомогою потрібних механік та особливостей.

Успішна гра повинна містити різноманітні механіки для забезпечення виживання головного героя та задоволення гравця.

***Основні завдання та механіки для прогресу в грі включають:***

Використання зброї: гравець має можливість використовувати різні типи зброї, які можна знайти на карті або здобути через час у певних точках.

Лікування для виживання: гравець може полікувати своє здоров'я, підвищивши свої шанси на успішне проходження.

Можливість збереження: гравець може зберігати свій прогрес, щоб продовжувати гру без страху втрати прогресу. У випадку смерті гравець повертається до останньої точки збереження.

***Ідея гри***

Гравець опиняється в абсурдному та складному світі, де головний герой є загартованим воїном-мисливцем, що подорожує через величезне підземелля. Основна мета полягає в тому, щоб перемогти всіх ворогів та вийти з лабіринту.

***Що гравець може робити***

Гравець відправляється в захоплюючу подорож, перетинаючи безліч кімнат заплутаного лабіринту. Кожна кімната пропонує унікальний досвід та можливості, адже сам лабіринт створює атмосферу напруги та невизначеності. У цьому захоплюючому світі гравець має унікальну можливість знаходити різноманітну зброю та предмети, які стануть його надійними союзниками у боротьбі з темрявою. Від розкішної зброї до корисних артефактів, кожен етап подорожі пропонує нові виклики та нагороди, роблячи кожну кімнату унікальною ареною для пригод та випробувань.

***Мета гри***

Гравець повинен битися з різними ворогами та босами, які захопили лабіринт. Він може використовувати різні види зброї. Перемога допомагає гравцеві здобувати ключові предмети, що дозволяють просуватися глибше в лабіринт і зустрічати більш потужних ворогів**.**

## 2.2 Архітектура проєкту

Після визначення основних механік слід перейти до розробки архітектури проекту. Архітектура визначає структуру програмного забезпечення, організацію модулів та взаємодію між ними. Це критично важливо для забезпечення масштабованості, надійності та продуктивності гри. Архітектура проекту повинна враховувати всі аспекти гри, від процедурної генерації рівнів до оптимізації ігрового процесу.

Для створення архітектури була створена ігрова діаграма з використанням різних пунктів гри, які також використані як папки для скриптів. Кожен із пунктів відповідає за відповідну функціональність і залежить від іншого пункта.
Для правильного планування гри необхідно створити архітектурну складову для кращого розуміння, які етапи повинні виконуватись в першу чергу, а які не можуть продовжуватись без інших. Також потрібно розуміти середовище, потрібне для успішного виконання завдання.

**Для чого потрібна архітектура проєкту та діаграма послідовності?**

**Архітектура проекту** в контексті розробки комп'ютерної гри є важливим аспектом, що визначає структуру, взаємозв'язки та організацію компонентів програмного забезпечення. Вона визначає принципи, які керують створенням, розширенням та підтриманням гри. Архітектурна концепція дозволяє розробникам вирішувати проблеми, такі як модульність, розширюваність та зручність управління кодом.

**Архітектура проєкту у розробці гри на Unity допомагає:**

1) ***Модульність:***

Розділити функціональність гри на логічні модулі для полегшення розробки, тестування та підтримки.

2) ***Розширюваність:***

Забезпечити можливість динамічного розширення гри шляхом додавання нових функцій чи модулів.

3) ***Зручність управління:***

Забезпечити зручні та легко зрозумілі засоби управління та взаємодії між компонентами. (Рис. 2.1).

**Діаграма послідовності:**

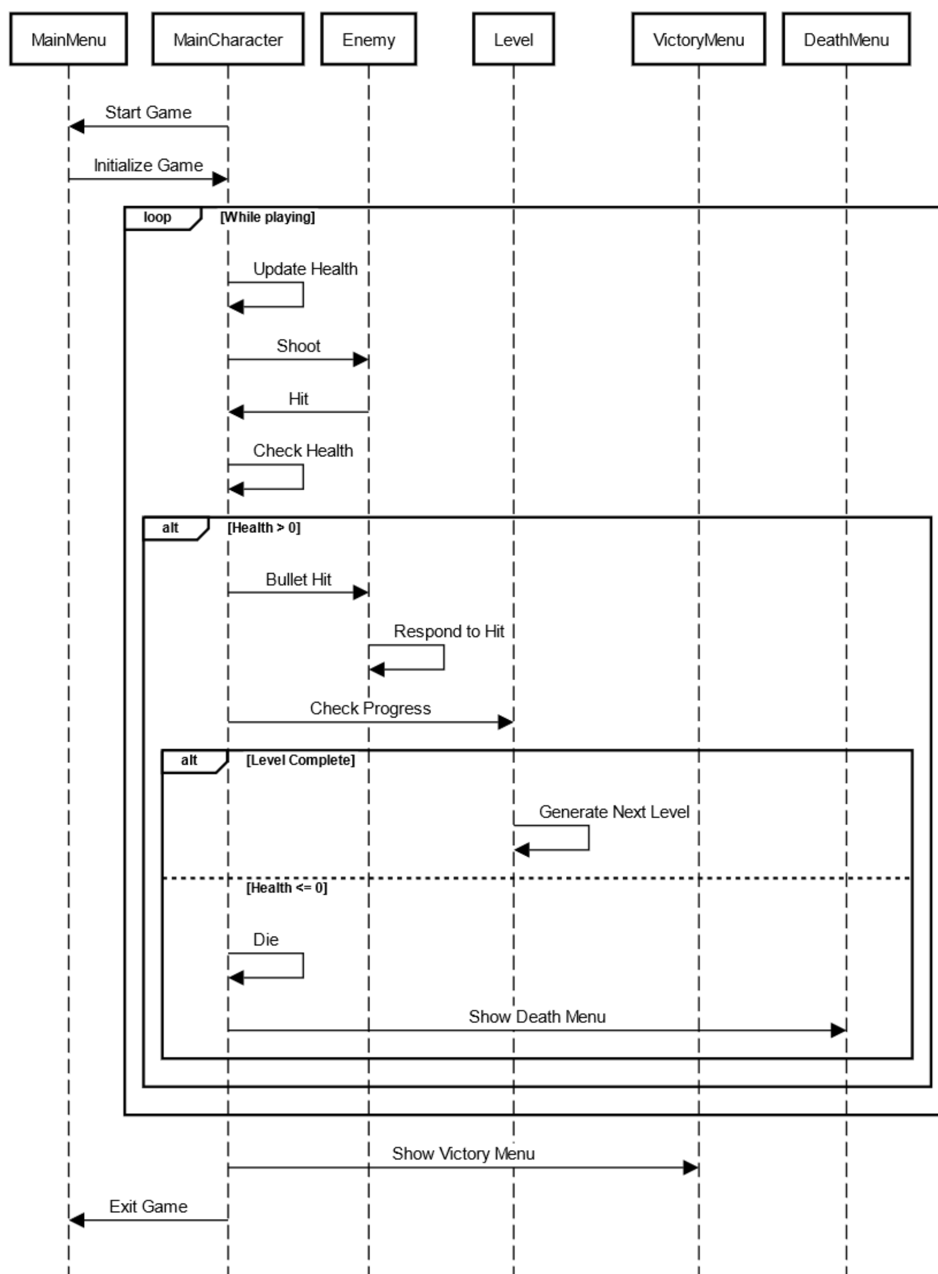


Рис. 2.1. Діагріма послідовності

Діаграма послідовності це графічний представник взаємодій між об'єктами в часі та порядку їх виконання. У контексті розробки гри на Unity, вона ілюструє взаємодію різних систем, скриптів та компонентів під час виконання певної функціональності. Діаграма послідовності сприяє розумінню та оптимізації взаємодій, допомагає визначити потреби в обміні інформацією та контролю між об'єктами. Це інструмент, який дозволяє:

***1) Чітко визначати послідовність подій:***

Дозволяє розробникам точно визначити порядок виконання дій та подій в межах взаємодії між об'єктами.

***2) Оптимізує використання ресурсів:***

Допомагає уникнути зайвого використання ресурсів шляхом ефективного обміну інформацією та виконання лише необхідних дій.

***3) Полегшує відлагодження та аналіз:***

Забезпечує зручність відлагодження, оскільки ілюструє потік виконання програми під час конкретної взаємодії.

## Процедурна генерація рівнів

Процедурна генерація рівнів — це метод, який використовує псевдо-випадковість і алгоритми для створення різноманітних рівнів замість традиційного ручного розроблення рівнів.

**Загальна структура процедурної генерації:**

Створення базового рівня - створюється основний шаблон або «текстурний» рівень, який може містити елементи, такі як підлога, перешкоди, об'єкти, тощо.

Визначення правил генерації – визначаються правила для генерації рівня. Це може включати такі фактори: розташування локацій, коридорів, розміри локацій, розміщення об'єктів та рівень складності локації.

Використання алгоритмів генерації - Розробник використовує алгоритми генерації, такі як клітинний автомат або алгоритм «Комерсанта», щоб створити рівень відповідно до певних правил. Цей алгоритм створює логічні рівні за допомогою псевдо-випадковості та ітерації.

Тестування та налаштування - після генерації проводиться тестування та вдосконалення, щоб переконатися, що рівні, які були створені, відповідають вимогам гри та забезпечують чудовий геймплей.

Якщо детальніше, то процедура випадкової генерації являє собою замкнений процес(цикл) автоматизованоо створення нових локацій із використанням випадкових елементів які обмежуються генерацією можливості. У контексті даної курсової роботи, генерація рівнів представляє створення 15 різних кімнат із виходами зображених як коридори, завдяки яким можна перейти від одної локації до іншої. Процес генерації відбувається завдяки створеному об'єкту в сцені, так як при запуску гри всі об'єкти генеруються, то відповідно тоді й запускається скрипт який генерує випадкові локації.

Кожна локація генерується випадково згідно правил що надають відповідність та логіку структури локації. Не менш важливо є розташування виходів, для правильної генерації, самі виходи(коридори) генеруються випадково обов'язково з врахуванням правил генерації щоб цей процес відбувався безперебійно. Загальний процес включає використання скриптів для створення пустої місцевості та додавання до неї префабів, які представляють різні об'єкти, ворогів та інші елементи гри. Цей метод динамічної генерації рівнів використовується для забезпечення відповідності та оптимальності.

Процедура випадкової генерації працює як замкнений цикл і автоматично створює нові місця з випадковими елементами, які обмежуються параметрами генерації. У контексті цієї курсової роботи генерація рівнів передбачає створення п'ятнадцяти окремих кімнат із виходами, які є коридорами, що дозволяють переміщатися між різними місцями. У процесі генерації всі об'єкти створюються під час запуску гри, викликаючи відповідний скрипт для генерації випадкових локацій.

**Висновки до розділу 2**

В цьому розділі було розглянуто проблематику створення рольової гри, та які складові потрібно для залучення гравців, проаналізовано основні механіки гри, для правильного планування та описано чому це важливо. Описано всі головні етапи гри, які відповідають за відповідний функціонал та в поєднанні доповнюють загальне враження від гри.

# РОЗДІЛ 3

# Програмне та технічне забезпечення

## 3.1 Засоби розробки

Засобами розробки є : Unity, Unity Asset Store, Visual Studio Code.

Засоби розробки включають технічну документацію Unity, яка дозволяє працювати з інтерфейсом і бібліотеками під час створення продуктів. У програмі розробки Unity є велика кількість інструментів, які можна використовувати для створення додатків і додавання унікальних під'єднуваних модулів.

Unity - один із найпопулярніших та найкращих інструментів для створення ігор, різноманітних додатків та дизайну середовища.

Цей рушій також є потужним графічним інструментом, оскільки він підтримує різноманітні ефекти, такі як освітлення, тіні, текстури та шейдери. Для розробників важливою перевагою є наявність вбудованої системи фізики та системи анімації. Система анімації містить інструменти для створення складних анімацій для персонажів, об'єктів та інших елементів гри. Завдяки вбудованій системі фізики розробник може створювати реалістичні фізичні об'єкти, такі як гравітація, колізії та симуляція руху.

Кожна сцена містить багато об'єктів, а пусті об'єкти — це об'єкти, які не мають ігрової моделі. Сценарії будуть взаємодіяти з певними наборами елементів кожного об'єкта. Кожен із цих елементів має свою назву, (Unity дозволяє називати різні об'єкти подібними назвами), теги та шари, на яких вони будуть відображатися. Логіка гри визначається створенням окремих сценаріїв для кожного елемента та додаванням цих сценаріїв до відповідного об'єкта. Кожен окремий сценарій має два стандартних методи, це: start і update(початок та кінець події).В цих методах описуються основні дії та явища, які відбудуться в результаті постійного виклику.

## 3.2 Вимоги до технічного та програмного забезпечення

Для розробки гри було визначено такі вимоги до програмного забезпечення:

1. Unity -версія 2022.3.7f1.
2. C# - мова програмування.
3. Microsoft Visual Studio Code - середовище для написання коду.
4. Unity Asset Store - магазин ассетів, інструментів та бібліотек.

Вимоги до технічного забезпечення:

1. Процесор: AMD Ryzen 5 3600
2. Відеокарта: NVIDIA GeForce GTX 1650
3. Операційна система: Windows 10/11 64-розрядна
4. Оперативна пам’ять: 16 ГБ

### 3.3 Програмна реалізація

Розробка гри в жанрі RogueLike на рушії Unity потребує ретельного планування та детального програмування кожного аспекту гри. Після визначення основних механік та загальної структури гри, настає етап програмної реалізації, який є одним з найважливіших етапів створення гри. У цьому розділі буде детально розглянуто ключові аспекти програмування гри, які забезпечують її функціональність, динаміку та інтерактивність. Програмна реалізація включає в себе декілька важливих елементів, кожен з яких відіграє значну роль у створенні цілісного ігрового досвіду. Спершу ми розглянемо процес прискорення героя, який забезпечує плавний та динамічний рух персонажа в ігровому світі. Потім перейдемо до механік руйнування об’єктів, що додають інтерактивності та реалізму до ігрового процесу.

Після цього буде описано взаємодію з предметами, яка дозволяє гравцеві збирати, використовувати та маніпулювати різними об'єктами в грі. Важливим аспектом є також міні-карта, що допомагає гравцеві орієнтуватися у складному процедурно згенерованому світі гри. Не менш важливою є тема маніпулювання головним персонажем, яка включає в себе налаштування анімацій, управління та реакції персонажа на дії гравця. На завершення буде описаноінтерфейс та механіки гри, що об’єднують усі елементи гри в єдиний, зрозумілий та зручний для користувача інтерфейс.

Усі ці аспекти разом утворюють основу програмної реалізації гри. Вони забезпечують не тільки технічну функціональність, але й створюють захопливий ігровий процес, який відповідає високим стандартам сучасних ігор жанру Rogue-Like.

### 3.3.1 Прискорення героя

Для реалізації цієї механіки було написано скрипт для **Player Controller**, відповідно визначивши швидкість "руху" прискорення і час цього "руху" та встановивши проміжок між яким можна робити цю дію (рис. 3.3.1.1)

- Активна швидкість руху (**activeMoveSpeed**):

Це значення визначає, наскільки швидко герой рухається по екрану у "звичайному" режимі, коли не виконується жодна спеціальна дія, така як стрибок.

- Швидкість стрибка (**dashSpeed**):

Це значення визначає, наскільки швидко герой може переміщуватися під час спеціального руху, який називається стрибком. Воно встановлюється на 8 пікселів за одиницю часу.

- Тривалість стрибка (**dashLength**):

Це значення вказує, як довго триватиме стрибок героя. Воно встановлене на 0.5 секунди, щоб надати короткочасний, але ефективний стрибок.

- Час охолодження стрибка (**dashCooldown**):

Це значення показує, скільки часу герой повинен почекати перед тим, як зможе виконати наступний стрибок. Це важливо, щоб уникнути можливих зловживань або перенасичення гри. Воно встановлене на 1 секунду.

- Лічильник часу стрибка (**dashCounter**) і лічильник часу охолодження (**dashCoolCounter**):

Ці дві змінні використовуються для відстеження часу, що залишився до закінчення стрибка або до того моменту, коли герой зможе виконати наступний стрибок після завершення попереднього.

Загалом, механіка прискорення дозволяє ефективно керувати рухом героя у грі, забезпечуючи при цьому ігровий процес з певною динамікою та балансом.(Рис.

```
private float activeMoveSpeed;
public float dashSpeed = 8f, dashLength = .5f, dashCooldown = 1f;
private float dashCounter, dashCoolCounter;
```

3.1)**.**

**Рис. 3.1. Механіка прискорення**

### 3.3.2 Руйнування об’єктів

Для цієї механіки було використано простий скрипт при якому об’єкти руйнються лише коли гравець використовує прискорення для того щоб влучити в об’єкт і коли він його торкається, об’єкт зникає. Скрипт відповідає за реакцію на зіткнення об'єкта з тригерним колайдером. Якщо зіткнення відбувається з гравцем і гравець має ще можливість прискорення, то об'єкт, що має цей колайдер, буде видалено зі сцени гри (Рис 3.2).

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == "Player")
    {
        if (PlayerController.instance.dashCounter > 0)
        {
            Destroy(gameObject);
        }
    }
}
```

Рис 3.2. Знищення об’єктів

Цей скрипт є обробником подій взаємодії колайдера у двовимірному просторі.

- *private void OnTriggerEnter2D(Collider2D other)*

Це метод, який викликається, коли об'єкт потрапляє в тригерний **Collider**. В якості аргументу **other** передається **Collider**, з яким взаємодіє даний об'єкт.

- *if (other.tag == "Player"):*

Це умова перевіряє, чи тег (tag) об'єкта, з яким взаємодіємо, рівний "Player". Теги використовуються для ідентифікації типу об'єкта.

- *if (PlayerController.instance.dashCounter > 0):*

Ця умова перевіряє, чи значення **dashCounter** у класі **PlayerController** більше 0. Використано статичний екземпляр класу **PlayerController**, щоб отримати доступ до методів та змінних цього класу.

- *Destroy(gameObject):*

Ця команда видаляє поточний об'єкт зі сцени гри. У конкретному контексті, якщо тригерний **Collider** зіштовхується з гравцем і гравець має ще можливість прискорення *(dashCounter > 0)*, то об'єкт, до якого прикріплений цей **Collider**, буде знищено.

### 3.3.3 Взаємодія з предметами

Взаємодіяти у грі можна лише з двома аптечками, одна зеленого кольору – лікує на 1 очко, друга синього кольору – лікує на 3 очка здоров'я. Для реалізації було використано префаб аптечки та використано скрипт який дозволяє підбирати аптечку тільки гравці та відновлювати здоров'я на одне очко здоров'я, також в момент коли гравець підбирає аптечку, вона знищується (Рис 3.3).

```
private void OnTriggerEnter2D(Collider2D other)
{
    if(other.tag == "Player")
    {
        PlayerHealthController.instance.HealPlayer(healAmount);

        Destroy(gameObject);
    }
```

Рис. 3.3. Лікування героя

Для того щоб встановити кількість здоров'я яке гравець буде відновлювати коли бере аптечку, було визначено параметри для CurrentHealth (здоров'я гравця на момент),

такі якщо здоров'я гравця дорівнює максимуму, тоді він не може полікуватись, у випадках коли здоров’я менше максимуму – він лікується (рис. 3.4).

Рис. 3.4. Встановлення відновленого здоров’я

```
public void HealPlayer(int healAmount)
{
    currentHealth += healAmount;
    if(currentHealth > maxHealth)
    {
        currentHealth = maxHealth;
    }
```

### 3.3.4 Міні-карта

Міні-карта і звичайна карта реалізована за допомогою створення і прив’язки другорядної камери (Map Camera) до основної (Main Camera). Працює це все дуже простим методом, міні-карта є дублікатом всієї карти і цей дублікат зменшений в розмірі у 50 разів, також має такі властивості: напівпрозорий, відображає лише контури локації і коридори між локаціями, також точку на міні-карті якою є гравець, все решта (коробки, вороги, постріли, предмети і т.д) не відображається. Для того щоб ускладнити процес самої гри, було реалізовано так, що у room.cs у нас є **public GameObject mapHider**: Це об'єкт, який приховує карту або її частину. Карта працює через **ui unity**, вона виводить на екран вгорі праворуч лейаут кімнати Тобто коли гравець переходить на іншу локацію – вона стає видимою(значення true). Відповідно до кожної із 14 локацій, присвоєно **MapHider.** (Рис 3.5).

Велика мапа реалізована за такою ж схемою і за допомогою простого скрипта, який активовує та відображає на екрані **BigMap,** тільки в тому випадку, коли користувач натискає відповідну клавішу на клавіатурі. Також коли користувач відкриває велику мапу – час зупиняєтсья.

Рис 3.5. Міні-карта

### 3.3.5 Маніпулювання головним персонажем

Налаштування системи обробки подій визначає використання введення. Система введення Unity надає високорівневий підхід, який дозволяє взаємодіяти з джойстиками, мишами, клавішами та іншими введеними пристроями. Обробка подій вводу регулює реакцію гри на дії гравця, такі як рух, стрибки та інші команди.

Фізична модель виконує технічну частину руху персонажа. **Rigidbody2D** дозволяє керувати динамікою руху, враховуючи фізичні закони. Щоб покращити геймплей і реалістичність, анімаційні переходи та параметри анімації регулюють, як персонаж реагує на зміни стану, такі як хідба, біг або стрибок.

Реалізація зіткнень і фізичних взаємодій є важливими елементами ефективного керівництва персонажем. Colider2D та Rigidbody2D забезпечують точне моделювання об’єктів і їхню поведінку. Це дозволяє виконувати рухи правильно.

Завдяки гнучкості управління персонажем можна динамічно змінювати його параметри відповідно до умов геймплея. Наприклад, можливість змінювати характеристики руху, обмежувати стрибки чи запускати режими (наприклад, режими

бігу чи боротьби) розширює можливості гравця та покращує гнучкість геймплею (Рис. 3.6).

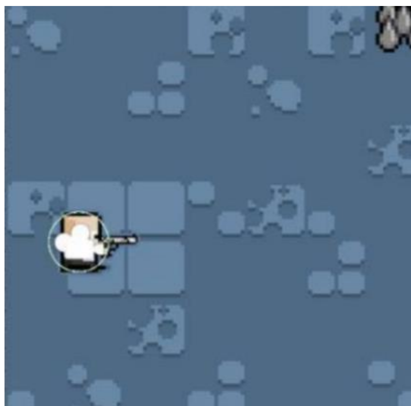

Рис. 3.6. Модель головного героя

### 3.3.6 Інтерфейс

Для створення меню взаємодії в Unity, використовується об'єкт Canvas. Canvas можна назвати "полотном" для розміщення та організації різних елементів інтерфейсу гри. У об’єкті Canvas є кнопки, які додаються за допомогою об'єкта "Button", відповідно коли гравець натискає на таку кнопку, викликається певна подія чи функція, що визначається за допомогою "Unity Event". Це дозволяє визначити, які конкретні дії повинні відбутися при взаємодії гравця з цими елементами, наприклад, коли гравець натисканіє кнопку "Start Game", "Exit" тощо. Висновок тому, що Canvas виступає як простір, де розташовуються кнопки та інші елементи для створення інтерфейсу гри. За допомогою Unity Events ми можемо точно визначити події гри при взаємодії гравця з цими елементами інтерфейсу. (Рис. 3.7; Рис. 3.8)

**Декілька прикладів інтерфейсу**

Рис. 3.7. Головне меню

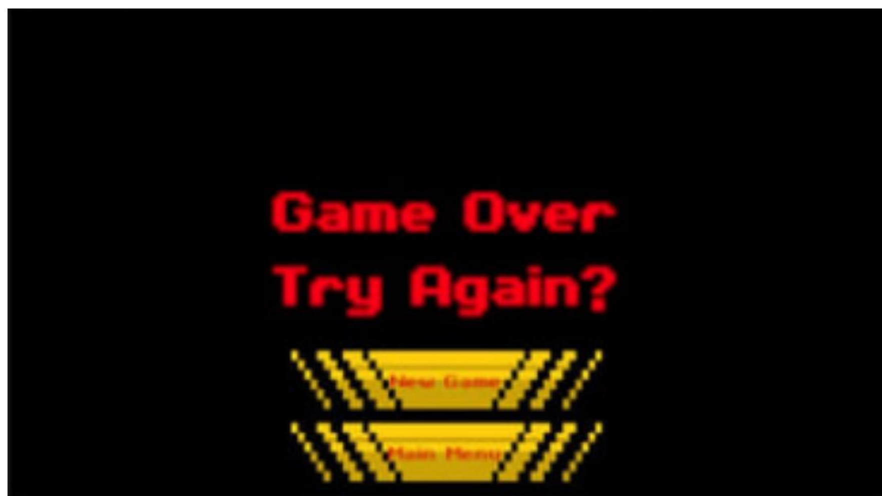


Рис. 3.8. Меню поразки

### 3.3.7 Механіки

- Контролер камери (Рис. 3.9).

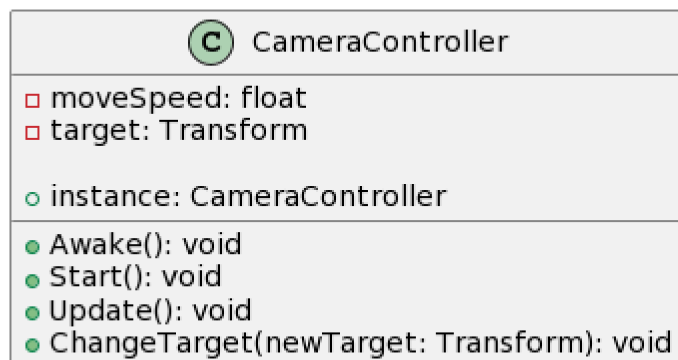


Рис 3.9. діаграма контролеру камери

***Опис діаграми:***

**CameraController:** Представляє клас CameraController.

- **Атрибути:**
  - **- moveSpeed: float**: Змінна для зберігання швидкості руху камери.
  - **- target: Transform**: Змінна для зберігання об'єкта, за яким камера буде слідкувати.
  - **• instance: CameraController**: Змінна для представлення єдиного екземпляру класу (реалізація патерну Singleton).
- **Методи:**
  - + **Awake(): void**: Метод, який викликається при ініціалізації об'єкта та виконує підготовчі операції.
  - + **Start(): void**: Метод, який викликається під час початкового фрейму та виконує початкові налаштування.
  - + **Update(): void**: Метод, який викликається на кожному кадрі та відповідає за оновлення стану камери.
  - + **ChangeTarget(newTarget: Transform): void**: Метод для зміни цілі камери.

- Контролер маніпуляції героєм (Рис. 3.10).

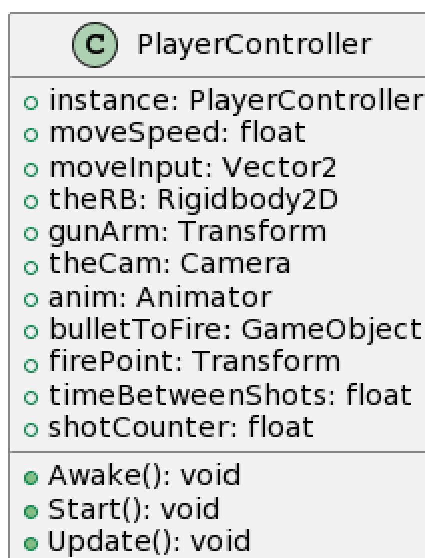


Рис. 3.10. Діаграма контролера маніпуляції героєм

***Опис діаграми:***

**PlayerController:** Представляє клас PlayerController.

- **Атрибути:**
  - **• instance: PlayerController**: Змінна для представлення єдиного екземпляру класу (реалізація патерну Singleton).
  - **- moveSpeed: float**: Змінна для зберігання швидкості руху гравця.
  - **- moveInput: Vector2**: Змінна для зберігання вектору вводу руху гравця.
  - **- theRB: Rigidbody2D**: Змінна для представлення компонента Rigidbody2D.
  - **- gunArm: Transform**: Змінна для представлення трансформації гравцевої руки зі зброєю.
  - **- theCam: Camera**: Змінна для представлення компонента Camera.
  - **- anim: Animator**: Змінна для представлення компонента Animator.
  - **- bulletToFire: GameObject**: Змінна для представлення об'єкта снаряду.
  - **- firePoint: Transform**: Змінна для представлення точки вистрілу.
  - **- timeBetweenShots: float**: Змінна для визначення часу між пострілами.
  - **- shotCounter: float**: Змінна для лічильника часу до наступного пострілу.
- **Методи:**
  - + **Awake(): void**: Метод, який викликається при створенні об'єкта та виконує налаштування до початку гри.
  - + **Start(): void**: Метод, який викликається при старті гри та виконує початкові налаштування.
  - + **Update(): void**: Метод, який викликається на кожному кадрі та відповідає за оновлення стану гравця.

- Контролер завдання шкоди (Рис. 3.11).

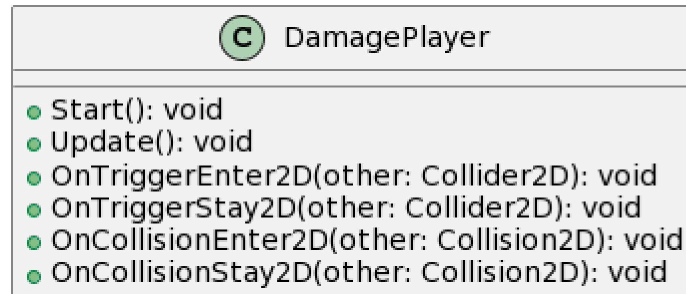


Рис. 3.11. діаграма завдання шкоди

***Опис діаграми:***

**DamagePlayer:** Представляє клас DamagePlayer.

- **Методи:**
  - + **Start(): void**: Метод, який викликається при ініціалізації об'єкта та виконує підготовчі операції.
  - + **Update(): void**: Метод, який викликається на кожному кадрі та відповідає за оновлення стану об'єкта.
  - + **OnTriggerEnter2D(other: Collider2D): void**: Метод, що викликається при зіткненні об'єкта з іншим колайдером.
  - + **OnTriggerStay2D(other: Collider2D): void**: Метод, що викликається, якщо об'єкт залишається в колайдері після зіткнення.
  - + **OnCollisionEnter2D(other: Collision2D): void**: Метод, що викликається при зіткненні об'єкта з іншим об'єктом.
  - + **OnCollisionStay2D(other: Collision2D): void**: Метод, що викликається, якщо об'єкт залишається в колізії після зіткнення.

- Контролер здоров’я героя (Рис. 3.12).

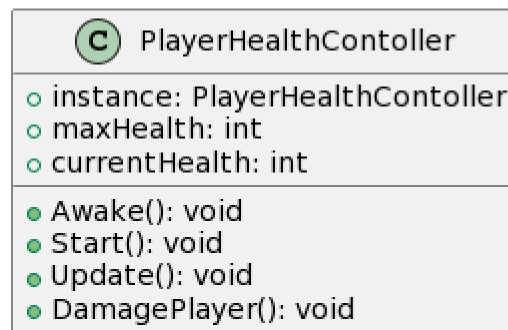


Рис. 3.12. діаграма контролеру здоров’я героя

***Опис діаграми:***

**PlayerHealthContoller:** Представляє клас PlayerHealthContoller.

- **Атрибути:**
  - **instance: PlayerHealthContoller**: Змінна для представлення єдиного екземпляру класу (реалізація патерну Singleton).
  - **maxHealth: int**: Змінна для визначення максимального здоров'я гравця.
  - **currentHealth: int**: Змінна для визначення поточного здоров'я гравця.
- **Методи:**
  - **Awake(): void**: Метод, який викликається при створенні об'єкта до початку гри та виконує підготовчі операції.
  - **Start(): void**: Метод, який викликається при старті гри та виконує початкові налаштування.
  - **Update(): void**: Метод, який викликається на кожному кадрі та відповідає за оновлення стану здоров'я гравця.
  - **DamagePlayer(): void**: Метод, який викликається для зменшення здоров'я гравця при отриманні пошкодження.

- Контролер куль героя (Рис. 3.13).

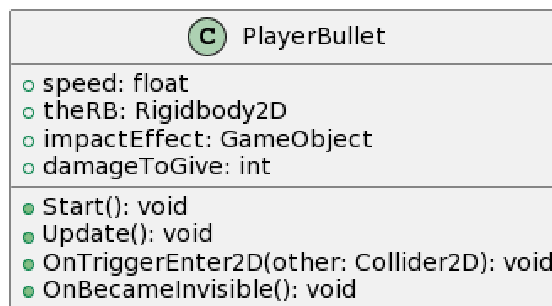


Рис. 3.13. Діаграма контролеру куль героя

***Опис діаграми:***

**PlayerBullet:** Представляє клас PlayerBullet.

- **Атрибути:**
    - **speed: float**: Змінна для зберігання швидкості снаряду.
    - **theRB: Rigidbody2D**: Змінна для представлення компонента Rigidbody2D.
    - **impactEffect: GameObject**: Змінна для представлення об'єкта ефекту при попаданні снаряду.
    - **damageToGive: int**: Змінна для визначення кількості шкоди, яку снаряд завдає.
- **Методи:**
    - **Start(): void**: Метод, який викликається при створенні об'єкта та виконує налаштування до початку гри.
    - **Update(): void**: Метод, який викликається на кожному кадрі та відповідає за оновлення стану снаряду.
    - **OnTriggerEnter2D(other: Collider2D): void**: Метод, який викликається при зіткненні об'єкта з іншим колайдером.
    - **OnBecameInvisible(): void**: Метод, який викликається, коли об'єкт стає невидимим в ігровому просторі.

- Контролер ворожих куль (Рис. 3.14).

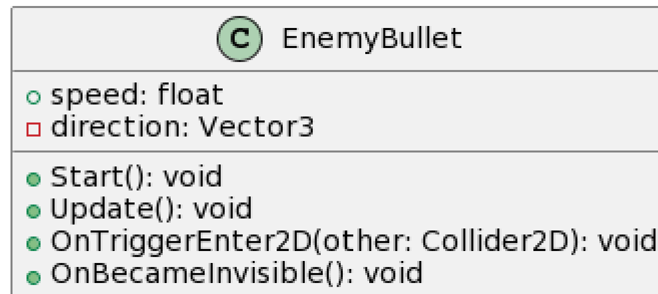


Рис. 3.14. Діаграма контролеру ворожих куль

***Опис діаграми:***

**EnemyBullet:** Представляє клас EnemyBullet.

**Атрибути:**

- **- speed: float**: Змінна для зберігання швидкості кулі.
- **- direction: Vector3**: Змінна для зберігання вектору напрямку руху снаряду.

**Методи:**

- + **Start(): void**: Метод, який викликається при старті об'єкта та виконує необхідні налаштування.
- + **Update(): void**: Метод, який викликається на кожному кадрі та відповідає за оновлення стану об'єкта.
- + **OnTriggerEnter2D(other: Collider2D): void**: Метод, який викликається при зіткненні об'єкта з іншим колайдером.
- + **OnBecameInvisible(): void**: Метод, який викликається, коли об'єкт стає невидимим в ігровому просторі

- Контролер руху ворогів (Рис. 3.15).

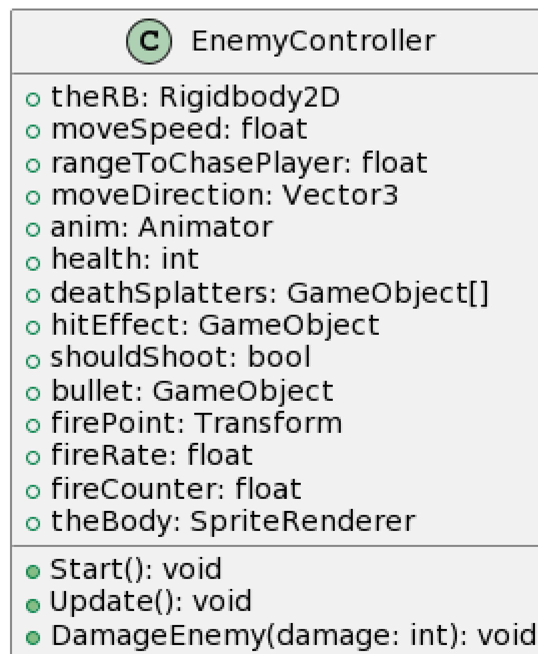


Рис. 3.15. Діаграма контролеру руху ворогів

***Опис діаграми:***

**EnemyController:** Представляє клас EnemyController.

- **Атрибути:**
  - **theRB: Rigidbody2D**: Змінна для представлення компонента Rigidbody2D.
  - **moveSpeed: float**: Змінна для зберігання швидкості руху ворога.
  - **rangeToChasePlayer: float**: Змінна для визначення дальності, на якій ворог почне переслідувати гравця.
  - **moveDirection: Vector3**: Змінна для зберігання вектору напрямку руху.
  - **anim: Animator**: Змінна для представлення компонента Animator.
  - **health: int**: Змінна для визначення здоров'я ворога.
  - **deathSplatters: GameObject[]**: Змінна для зберігання різноманітних об'єктів для ефекту при смерті ворога.

- **hitEffect: GameObject**: Змінна для зберігання об'єкту для ефекту попадання.
- **shouldShoot: bool**: Змінна для визначення можливості ворога стріляти.
- **bullet: GameObject**: Змінна для представлення об'єкта снаряду.
- **firePoint: Transform**: Змінна для представлення точки вистрілу.
- **fireRate: float**: Змінна для визначення швидкості стрільби ворога.
- **fireCounter: float**: Змінна для лічильника для стрільби.
- **theBody: SpriteRenderer**: Змінна для представлення компонента SpriteRenderer.

- **Методи:**
  - **Start(): void**: Метод, який викликається при ініціалізації об'єкта та виконує підготовчі операції.
  - **Update(): void**: Метод, який викликається на кожному кадрі та відповідає за оновлення стану об'єкта.
  - **DamageEnemy(damage: int): void**: Метод для обробки пошкодження ворога.

- Контролер наповнення локації (Рис. 3.16).

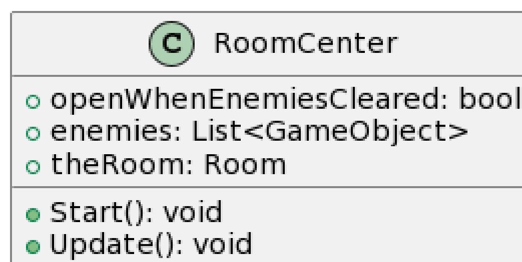


Рис. 3.16. Діаграма контролеру наповнення локації

***Опис діаграми:***

**RoomCenter:** Представляє клас RoomCenter.

- **Атрибути:**

- **openWhenEnemiesCleared: bool**: Змінна для визначення можливості відкриття кімнати після поразки ворогів.
- **enemies: List<GameObject>**: Змінна для представлення списку ворожих об'єктів у кімнаті.
- **theRoom: Room**: Змінна для представлення об'єкта кімнати.

- **Методи:**
  - **Start(): void**: Метод, який викликається при ініціалізації об'єкта та виконує підготовчі операції.
  - **Update(): void**: Метод, який викликається на кожному кадрі та відповідає за оновлення стану об'єкта

- Контролер генерації локації (Рис. 3.17).

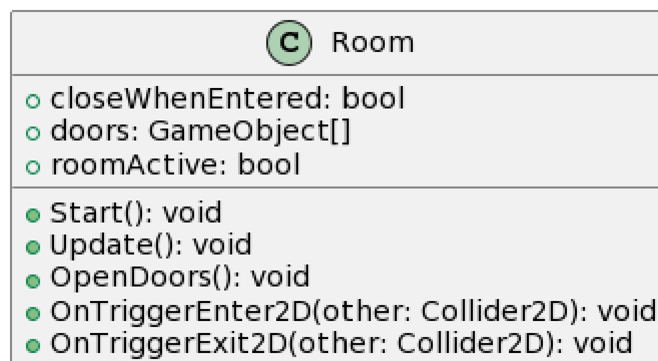


Рис. 3.17. Діаграма контролеру генерації локації

***Опис діаграми:***

**Room:** Представляє клас Room.

- **Атрибути:**
  - **- closeWhenEntered: bool**: Змінна для визначення закриття кімнати при вході в неї.
  - **- doors: GameObject[]**: Змінна для представлення масиву дверей кімнати.
  - **- roomActive: bool**: Змінна для визначення активності кімнати.

- **Методи:**
  - + **Start(): void**: Метод, який викликається при ініціалізації об'єкта та виконує підготовчі операції.
  - + **Update(): void**: Метод, який викликається на кожному кадрі та відповідає за оновлення стану об'єкта.
  - + **OpenDoors(): void**: Метод для відкриття дверей кімнати.
  - + **OnTriggerEnter2D(other: Collider2D): void**: Метод, який викликається, коли інший об'єкт зіткнувся з колайдером входу в кімнату.
  - + **OnTriggerExit2D(other: Collider2D): void**: Метод, який викликається, коли інший об'єкт покинув зону входу в кімнату.
- Контролер екрану перемоги (Рис. 3.18).

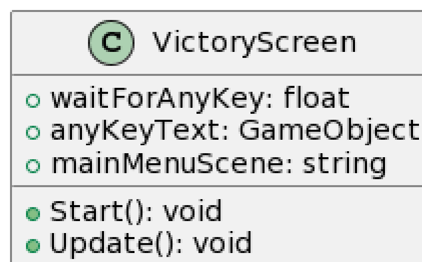


Рис. 3.18. Діаграма контролеру екрану перемоги

***Опис діаграми:***

**VictoryScreen:** Представляє клас VictoryScreen.

- **Атрибути:**
  - **waitForAnyKey: float**: Змінна для визначення часу очікування натискання будь-якої клавіші після відображення екрану перемоги.
  - **anyKeyText: GameObject**: Змінна для представлення текстового об'єкта, який вказує користувачеві на можливість натискання будь-якої клавіші.
  - **mainMenuScene: string**: Змінна для визначення назви головного меню, до якого користувач буде перенаправлений після виграшу.

- **Методи:**
  - **Start(): void**: Метод, який викликається при ініціалізації об'єкта та виконує підготовчі операції.
  - **Update(): void**: Метод, який викликається на кожному кадрі та відповідає за оновлення стану об'єкта.
- Контролер виходу з рівня (Рис. 3.19).

Рис. 3.19. Діаграма контролеру виходу з рівня

***Опис діаграми:***

**LevelExit:** Представляє клас LevelExit.

- **Атрибути:**
  - **- levelToLoad: string**: Змінна для визначення назви рівня, який буде завантажений після того, як гравець пройде вихід.
- **Методи:**
  - + **Start(): void**: Метод, який викликається при ініціалізації об'єкта та виконує підготовчі операції.
  - + **Update(): void**: Метод, який викликається на кожному кадрі та відповідає за оновлення стану об'єкта.
  - + **OnTriggerEnter2D(other: Collider2D): void**: Метод, який викликається, коли інший об'єкт зіткнувся з колайдером виходу.

- Контролер головного меню (Рис. 3.20).

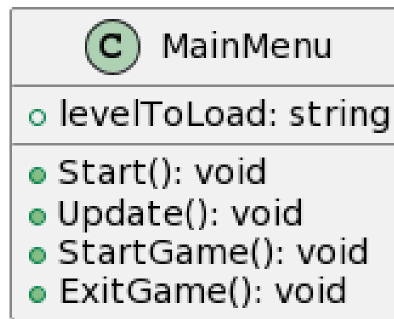


Рис. 3.20. Діаграма контролеру головного меню

***Опис діаграми:***

**MainMenu:** Представляє клас MainMenu.

- **Атрибути:**
    - **- levelToLoad: string**: Змінна для визначення назви рівня, який буде завантажений після вибору гравцем "Start Game".
- **Методи:**
    - + **Start(): void**: Метод, який викликається при ініціалізації об'єкта та виконує підготовчі операції.
    - + **Update(): void**: Метод, який викликається на кожному кадрі та відповідає за оновлення стану об'єкта.
    - + **StartGame(): void**: Метод для початку нової гри.
    - + **ExitGame(): void**: Метод для виходу з гри.

- Контролер користувацького інтерфейсу Рис. 3.21).

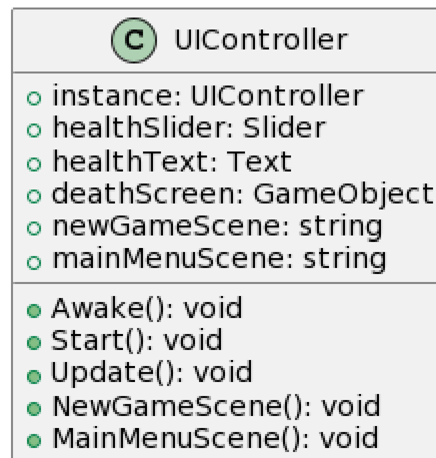


Рис. 3.21. Діаграма контролеру UI

***Опис діаграми:***

**UIController:** Представляє клас UIController.

- **Атрибути:**
  - • **instance: UIController**: Змінна для представлення єдиного екземпляру класу (реалізація патерну Singleton).
  - • **healthSlider: Slider**: Змінна для представлення компонента слайдера, який відображає здоров'я гравця.
  - • **healthText: Text**: Змінна для представлення компонента тексту, який відображає кількість здоров'я гравця.
  - • **deathScreen: GameObject**: Змінна для представлення об'єкта екрану смерті.
  - • **newGameScene: string**: Змінна для визначення назви сцени нової гри.
  - • **mainMenuScene: string**: Змінна для визначення назви головного меню.
- **Методи:**
  - • **Awake(): void**: Метод, який викликається при створенні об'єкта до початку гри та виконує підготовчі операції.

- • **Start(): void**: Метод, який викликається при старті гри та виконує початкові налаштування.
- • **Update(): void**: Метод, який викликається на кожному кадрі та відповідає за оновлення стану інтерфейсу користувача.
- • **NewGameScene(): void**: Метод, який викликається для переходу до нової гри.
- • **MainMenuScene(): void**: Метод, який викликається для переходу до головного меню.

- Контролер процедурної генерації (Рис. 3.22).

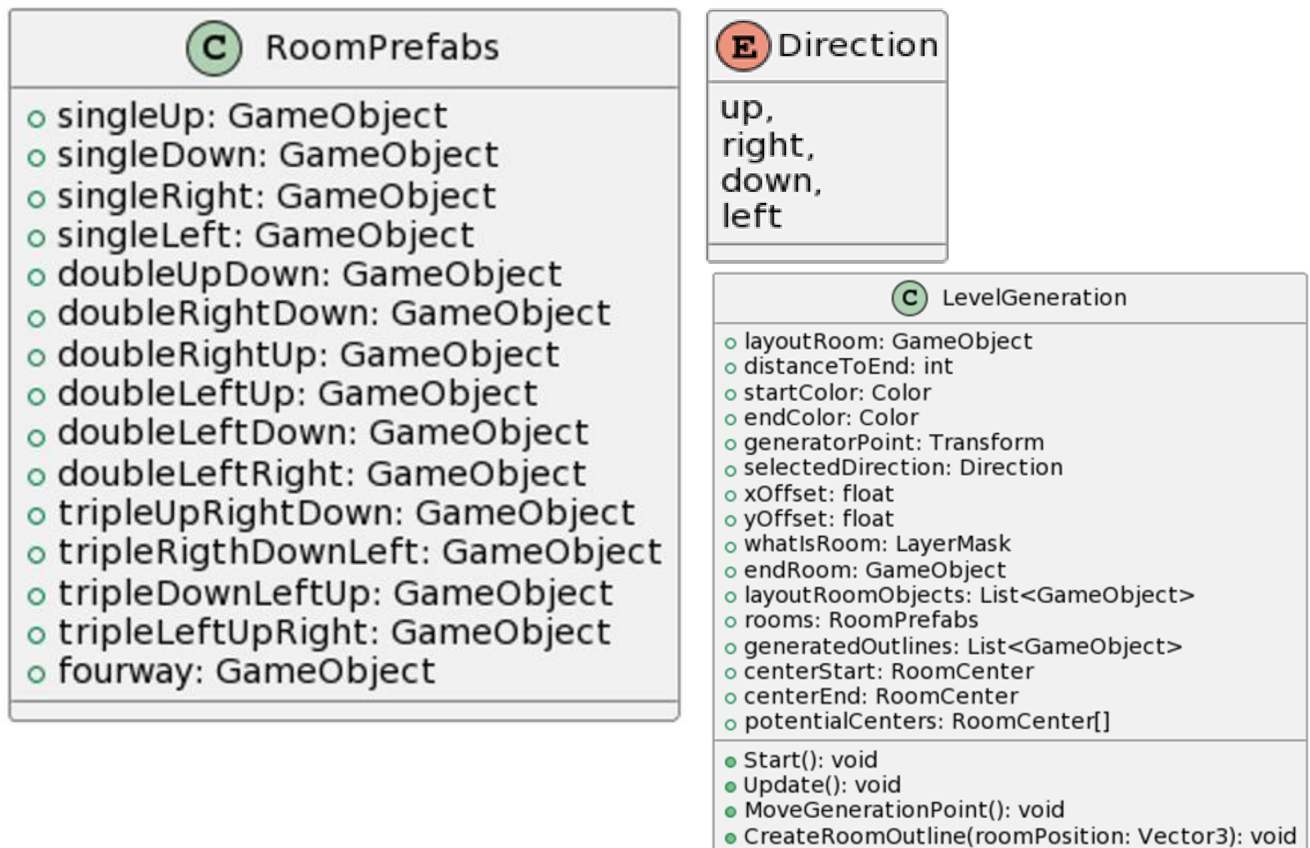

Рис. 3.22. Діаграма контролеру процедурної генерації

**Опис діаграми:**

Клас LevelGeneration відповідає за процес створення рівнів у грі. Спочатку він визначає кілька ключових параметрів, таких як відстань до кінця рівня, початковий та кінцевий кольори для кімнат, і точка генерації, з якої починається побудова рівня. Для створення кожної нової кімнати клас використовує зміщення по осях X і Y, а також обраний напрямок, який визначає, куди буде додана наступна кімната.

Процес генерації починається з методу Start(), який викликається на початку гри. Цей метод ініціалізує генерацію рівня, визначаючи початкові параметри та викликаючи методи для створення кімнат. На кожному кадрі гри виконується метод Update(), який слідкує за станом генерації та оновлює її, якщо це необхідно. Клас також містить метод MoveGenerationPoint(), який відповідає за переміщення точки генерації до нової позиції на основі обраного напрямку та зміщення. Коли точка генерації переміщується, метод CreateRoomOutline(roomPosition: Vector3) створює контур для нової кімнати в заданій позиції. Це дозволяє поступово будувати рівень, додаючи одну кімнату за іншою.

Для створення різних типів кімнат LevelGeneration використовує клас RoomPrefabs, який містить шаблони для кімнат різних конфігурацій: одиночних кімнат, що розташовуються в одному напрямку (вгору, вниз, вправо, вліво), подвійних кімнат, що з'єднуються двома напрямками, трійних кімнат і навіть чотиристоронніх кімнат. Це дозволяє створювати складні та різноманітні рівні з різними шляхами та роздоріжжями. Напрямки генерації визначаються Direction, яке включає чотири можливі напрямки: вгору, вправо, вниз і вліво. Це перерахування допомагає обирати напрямок для переміщення точки генерації та створення нової кімнати.

Отже, логіка роботи класу LevelGeneration полягає у послідовному створенні рівня шляхом додавання нових кімнат у заданому напрямку від стартової точки до кінцевої кімнати. Використовуючи різні шаблони кімнат із класу RoomPrefabs та

дотримуючись обраних напрямків, клас створює різноманітні та цікаві рівні для гравців.

- Вибір персонажа (Рис. 3.23).

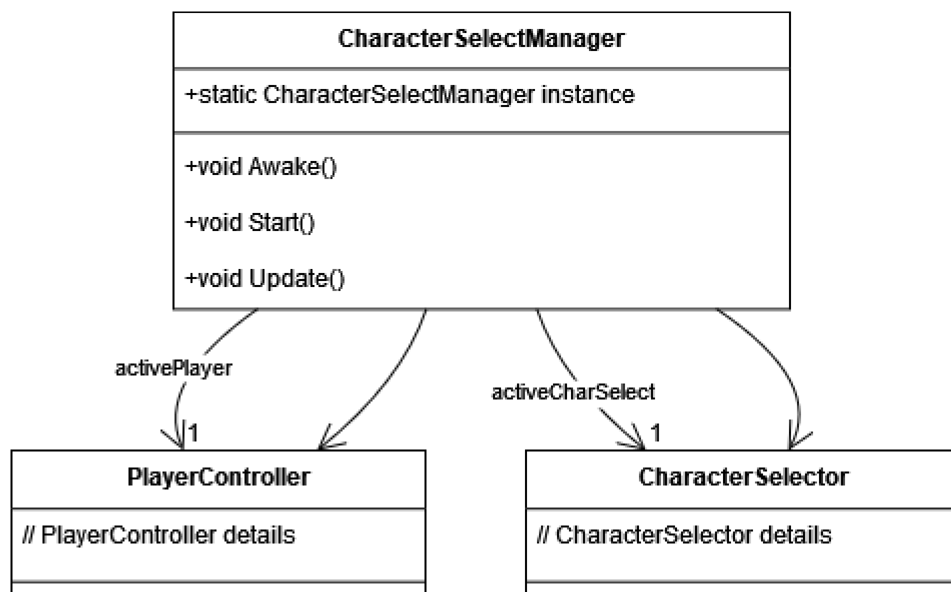


Рис. 3.23. Діаграма вибору персонажа

**Опис діаграми:**

Ця діаграма відображає взаємодію між об'єктами та класами в системі вибору персонажа в грі. Основним елементом є клас CharacterSelectManager, який відповідає за управління процесом вибору персонажа. Він містить два важливі атрибути: activePlayer, який представляє активного гравця, та activeCharSelect, який представляє вибраного персонажа.

Клас CharacterSelectManager взаємодіє з двома іншими класами: PlayerController та CharacterSelector. PlayerController керує діями активного гравця. Це означає, що коли гравець вибирає персонажа, PlayerController відповідає за збереження інформації про вибір гравця, що відображено через атрибут activePlayer.

З іншого боку, CharacterSelector забезпечує інтерфейс для вибору персонажа. Він взаємодіє з CharacterSelectManager для відображення доступних персонажів та обробки вибору гравця. Атрибут activeCharSelect вказує на поточного вибраного персонажа в процесі вибору.

Таким чином, логіка роботи системи полягає в тому, що гравець використовує інтерфейс CharacterSelector для вибору персонажа. CharacterSelectManager контролює цей процес, зберігаючи інформацію про активного гравця та вибраного персонажа. Взаємодія між PlayerController та CharacterSelectManager забезпечує, що вибір персонажа коректно зберігається та використовується в подальшій грі.

- Ігровий магазин (Рис. 3.24, Рис. 3.25).

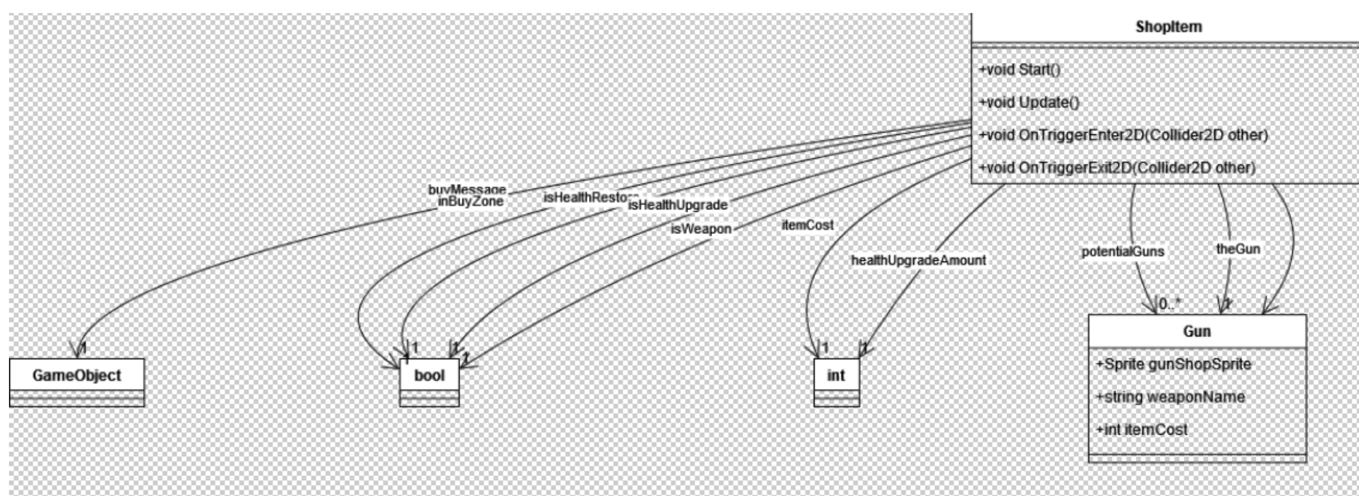


Рис. 3.24. Діаграма ігрового магазину

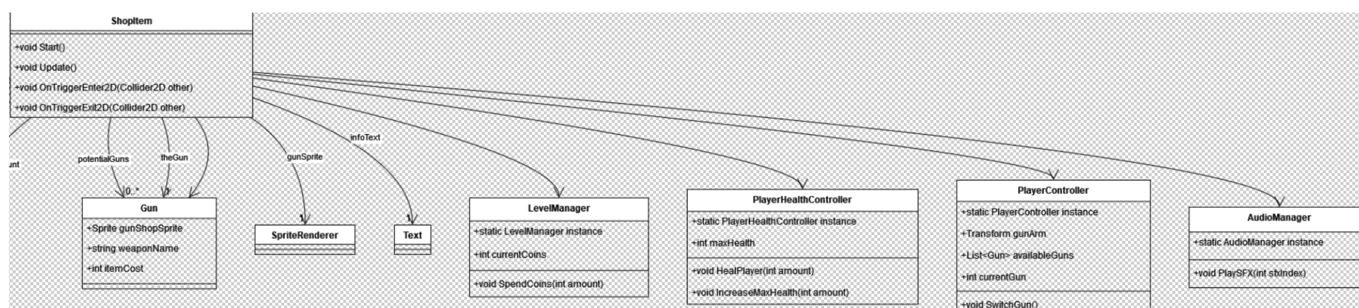


Рис. 3.25. Діаграма ігрового магазину

**Опис діаграми:**

Діаграма демонструє, як різні об'єкти та класи взаємодіють у системі гри для управління предметами в магазині та їх впливом на гравця.

Центральним об'єктом є клас ShopItem, який представляє предмет магазину. Цей клас має різні атрибути, які визначають властивості та функції предмета. Наприклад, є атрибути, що вказують на те, чи предмет відновлює здоров'я гравця, чи покращує його, а також чи є предмет зброєю. Вартість предмета також визначається тут.

Коли гравець потрапляє в зону покупки, це визначається атрибутом inBuyZone. Якщо гравець вирішує придбати предмет, з'являється відповідне повідомлення. У випадку, якщо предмет є зброєю, зображення зброї відображається за допомогою gunSprite, а сама зброя визначається об'єктом theGun, який може бути вибраний з масиву можливих гармат.

Клас ShopItem взаємодіє з кількома іншими компонентами гри для забезпечення різних аспектів ігрового процесу. LevelManager контролює різні аспекти рівнів гри, включаючи доступність та розташування предметів. PlayerHealthController керує здоров'ям гравця, що важливо для предметів, які відновлюють або покращують здоров'я. PlayerController відповідає за дії гравця, такі як переміщення та взаємодія з предметами. Нарешті, AudioManager забезпечує відповідні звукові ефекти при покупці або використанні предметів.

Загалом, логіка роботи системи полягає в тому, що гравець може взаємодіяти з предметами в магазині, які мають різні функції та вплив на ігровий процес. Ці предмети інтегруються з іншими компонентами гри, забезпечуючи комплексний ігровий досвід.

## Висновки до розділу 3

В цьому розділі було описано які засоби розробки було використано для створення гри та які системні та технічні вимоги потрібні були використані протягом роботи над грою.

Також було описано як гра була реалізована, описуючи такі етапи розробки: Прискорення героя, руйнування об’єктів, взаємодія з предметами, міні-карта,

Керування головним персонажем, Інтерфейс та інші механіки.

# ВИСНОВКИ

У процесі розробки гри в жанрі Rogue-Like було використано комплексний підхід, який включав кілька методів для досягнення поставленої мети. Спочатку був проведений ретельний аналіз конкурентів на ринку ігрових продуктів. Застосування методів конкурентного аналізу дозволило виявити основні тенденції ринку, різниці у ігрових ресурсах та визначити потреби користувачів. Було визначено, які рішення вже використовуються конкурентами для задоволення потреб гравців. Основними результатами цього етапу стали визначення характеристик продукту, такі як генерація процедурних рівнів, інтуїтивний інтерфейс, підтримка багатокористувацького режиму та баланс складності.

Після цього був проведений аналіз користувачів з метою визначення цільової аудиторії ігрових продуктів, їх потреб, проблем, мотивацій та побажань. Використання методів користувацького дослідження дозволило виявити основні проблеми та потреби потенційних гравців. Основними результатами цього етапу стали основна потреба аудиторії, яка підкреслила необхідність ігор з можливістю грати знову, цікавими механіками та можливістю персоналізації ігрового досвіду.

На основі цих досліджень було проведено ряд досліджень, спрямованих на збір та організацію даних для створення власного продукту. Використання методів формування технічного завдання, створення моделі проєкту, дослідження візуальних практик ігрових продуктів та розробки інформаційної архітектури дозволило сформулювати гіпотези щодо поліпшення користувацького досвіду.

Отже, результати дослідження показали, що використані рішення, такі як оптимізація ігрових елементів та адаптація до потреб гравців, є актуальними та оптимальними для сучасних ігор жанру Rogue-Like. Ці рішення дозволяють підвищити користувацький досвід, ефективно задовольняти потреби аудиторії та забезпечувати високу якість ігрових продуктів.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. App Diagrams. [Електронний ресурс] URL: https://app.diagrams.net/(дата звернення 01.02.2024)
2. Unity asset Store. [Електронний ресурс] URL:https://assetstore.unity.com (дата звернення 01.02.2024)
3. Visual studio. [Електронний ресурс] URL:https://visualstudio.microsoft.com (дата звернення 03.02.2024)
4. Jira Atlassian. [Електронний ресурс] URL:https://jira.atlassian.com (дата звернення 03.02.2024)
5. Unity. Unity User Manual 2022.3 [Електронний ресурс] URL:https://docs.unity3d.com/Manual/index.html (дата звернення 02.12.2023)
6. Creating and Using Scripts. [Електронний ресурс] URL:https://docs.unity3d.com/Manual/ScriptingSection.html (дата звернення 03.12.2023)
7. Character controlling in 2d game. [Електронний ресурс] URL:https://www.youtube.com/watch?v=lcw6nuc2uaU (дата звернення 04.03.2024)
8. Shooting system. [Електронний ресурс] URL: https://www.youtube.com/watch?v=ApLXuMeSVOI&ab_channel=MuddyWolf (дата звернення 05.03.2024)
9. Enemies that deal damage. [Електронний ресурс] URL:https://www.youtube.com/watch?v=KF3EVjOhN4c (дата звернення 06.02.2024)
10. Heatlh and damage. [Електронний ресурс] URL: https://www.youtube.com/watch?v=aoZqeG7rqV0&ab_channel=CanWithCode (дата звернення 04.03.2024)
11. Dashing. [Електронний ресурс] URL:https://www.youtube.com/watch?v=Do4LdlKB0bY&ab_channel=TacticalProgrammer (дата звернення 06.03.2024)
12. Pickups and healing. [Електронний ресурс] URL: https://www.youtube.com/watch?v=UPAWy4C-2mQ&ab_channel=Bobbville (дата звернення 06.03.2024)
13. Music and sounds. [Електронний ресурс] URL: https://www.youtube.com/watch?v=N8whM1GjH4w (дата звернення 06.02.2024)

14. Creating rooms. [Електронний ресурс] URL:

https://www.youtube.com/watch?v=sH69PdAxaPA&ab_channel=HardwareHaquer%28AustinStewart%29 (дата звернення 07.02.2024)

15. Entering and leaving rooms. [Електронний ресурс] URL:

https://www.youtube.com/watch?v=iwD7H0X8E7M (дата звернення 08.02.2024)

16. Victory screen. [Електронний ресурс] URL:

https://medium.com/nerd-for-tech/creating-a-win-screen-in-unity-da902fe540f3 (дата звернення 09.02.2024)

17. Death screen. [Електронний ресурс] URL:

https://forum.unity.com/threads/how-can-i-make-a-death-screen-for-my-player.1167551/ (дата звернення 10.02.2024)

18. Pause screen. [Електронний ресурс] URL:

https://medium.com/nerd-for-tech/creating-the-pause-menu-in-unity-e48277abdf8e (дата звернення 10.02.2024)

19. Menu system. [Електронний ресурс] URL:

https://gamedevacademy.org/unity-start-menu-tutorial/ (дата звернення 10.02.2024)

20. Procedural generation. [Електронний ресурс] URL:

https://www.youtube.com/watch?v=-QOCX6SVFsk&list=PLcRSafycjWFenI87z7uZHFv6cUG2Tzu9v&ab_channel=SunnyValleyStudio (дата звернення 11.02.2024)

21. Procedural generation. [Електронний ресурс] URL:

https://gamedevacademy.org/complete-guide-to-procedural-level-generation-in-unity-part-1/ (дата звернення 11.02.2024)

22. Procedural generation. [Електронний ресурс] URL:

https://www.reddit.com/r/proceduralgeneration/comments/mh2ow5/any_advice_on_procedural_map_generation_for_a_2d/ (дата звернення 11.02.2024)

23. Money system. [Електронний ресурс] URL:

https://www.youtube.com/watch?v=qmCOt5Lzyu8 (дата звернення 12.02.2024)

24. Collecting coins. [Електронний ресурс] URL:https://forum.unity.com/threads/how-to-collect-coin-in-unity.949107/ (дата звернення 12.02.2024)

25. Shop system. [Електронний ресурс] URL:

https://medium.com/geekculture/a-simple-shop-system-for-a-unity-2d-game-part-3-game-mechanics-5bd674a33287 (дата звернення 12.02.2024)

26. Mini map. Електронний ресурс] URL:https://www.youtube.com/watch?v=TkegkmRbrN0 (дата звернення 13.02.2024)

27. Big map. Електронний ресурс] URL:https://www.youtube.com/watch?v=Pejo_U_7cZo (дата звернення 13.02.2024)

28. Level Tracking. Електронний ресурс] URL:https://forum.unity.com/threads/how-do-you-make-an-object-follow-your-exact-movement-but-delayed.512787/ (дата звернення 13.02.2024)

29. Multiple characters to choose. Електронний ресурс] URL:https://forum.unity.com/threads/change-multiple-character-sprites.1444294/ (дата звернення 14.02.2024)

30. Final bosses. Електронний ресурс] URL:https://www.youtube.com/watch?v=moe9eLdtEKo (дата звернення 14.02.2024)

31. GitHub project.

Електронний ресурс] URL:https://github.com/samkov01/dyploma (дата звернення 15.02.2024)

# ДОДАТКИ

## Додаток А

***CharacterSelectManager.cs*** *- клас керує вибором персонажа, зберігає активного гравця та вибір персонажа, реалізуючи Singleton (шаблон, який забезпечує наявність лише одного екземпляра класу та надає до нього глобальний доступ.) патерн для єдиного екземпляра.*

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

public class CharacterSelectManager : MonoBehaviour
{
    public static CharacterSelectManager instance;
    public PlayerController activePlayer;
    public CharacterSelector activeCharSelect;
    private void Awake()
    {
        instance = this;
    }
    // Start is called before the first frame update
    void Start()
    {
    }
    // Update is called once per frame
    void Update()
    {
    }
}
```

**Додаток Б**

***CharacterSelector.cs*** *- клас керує вибором персонажа, перевіряє, чи розблокований персонаж, обробляє заміну персонажа при натисканні клавіші "E", і показує або приховує повідомлення про вибір персонажа при вході/виході гравця із зони.*

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

public class CharacterSelector : MonoBehaviour
{
    private bool canSelect;
    public GameObject message;
    public PlayerController playerToSpawn;
    public bool shouldUnlock;

    // Start is called before the first frame update
    void Start()
    {
        if (shouldUnlock)
        {
            if (PlayerPrefs.HasKey(playerToSpawn.name))
            {
                if (PlayerPrefs.GetInt(playerToSpawn.name) == 1)
                {
                    gameObject.SetActive(true);
                }
                else
                {
                    gameObject.SetActive(false);
                }
            }
            else
            {
                gameObject.SetActive(false);
            }
        }
    }

    // Update is called once per frame
    void Update()
```

```
    {
        if(canSelect)
        {
            if(Input.GetKeyDown(KeyCode.E))
            {
                Vector3 playerPos = PlayerController.instance.transform.position;

                Destroy(PlayerController.instance.gameObject);

                PlayerController newPlayer = Instantiate(playerToSpawn, playerPos,
playerToSpawn.transform.rotation);
                PlayerController.instance = newPlayer;

                gameObject.SetActive(false);

                CameraController.instance.target = newPlayer.transform;

                CharacterSelectManager.instance.activePlayer = newPlayer;
                CharacterSelectManager.instance.activeCharSelect.gameObject.SetActive(true);
                CharacterSelectManager.instance.activeCharSelect = this;
            }
        }
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        if(other.tag == "Player")
        {
            canSelect = true;
            message.SetActive(true);
        }
    }

    private void OnTriggerExit2D(Collider2D other)
    {
        if (other.tag == "Player")
        {
            canSelect = false;
            message.SetActive(false);
        }
    }
}
```

**Додаток В**

***ShopItem.cs** - клас обробляє логіку купівлі предметів у магазині, перевіряє наявність достатньої кількості монет, виконує купівлю предмета (відновлення здоров'я, підвищення здоров'я або зброї) та відповідним чином оновлює стан гри.*

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

public class ShopItem : MonoBehaviour
{
    public GameObject buyMessage;
    private bool inBuyZone;
    public bool isHealthRestore, isHealthUpgrade, isWeapon;
    public int itemCost;
    public int healthUpgradeAmount;
    public Gun[] potentialGuns;
    private Gun theGun;
    public SpriteRenderer gunSprite;
    public Text infoText;

    // Start is called before the first frame update
    void Start()
    {
        if(isWeapon)
        {
            int selectedGun = Random.Range(0, potentialGuns.Length);
            theGun = potentialGuns[selectedGun];

            gunSprite.sprite = theGun.gunShopSprite;
            infoText.text = theGun.weaponName + "\n - " + theGun.itemCost + " Gold - ";
            itemCost = theGun.itemCost;
        }
    }

    // Update is called once per frame
    void Update()
    {
        if(inBuyZone)
        {
            if(Input.GetKeyDown(KeyCode.E))
```

```
        {
            if(LevelManager.instance.currentCoins >= itemCost)
            {
                LevelManager.instance.SpendCoins(itemCost);

                if(isHealthRestore)
                {

PlayerHealthController.instance.HealPlayer(PlayerHealthController.instance.maxHealth);
                }

                if(isHealthUpgrade)
                {

PlayerHealthController.instance.IncreaseMaxHealth(healthUpgradeAmount);
                }

                if(isWeapon)
                {
                    Gun gunClone = Instantiate(theGun);
                    gunClone.transform.parent = PlayerController.instance.gunArm;
                    gunClone.transform.position = PlayerController.instance.gunArm.position;
                    gunClone.transform.localRotation = Quaternion.Euler(Vector3.zero);
                    gunClone.transform.localScale = Vector3.one;

                    PlayerController.instance.availableGuns.Add(gunClone);
                    PlayerController.instance.currentGun =
PlayerController.instance.availableGuns.Count - 1;
                    PlayerController.instance.SwitchGun();
                }
                gameObject.SetActive(false);
                inBuyZone = false;

                AudioManager.instance.PlaySFX(18);
            } else
            {
                AudioManager.instance.PlaySFX(19);
            }
        }
    }
  }
  private void OnTriggerEnter2D(Collider2D other)
```

```
    {
        if(other.tag == "Player")
        {
            buyMessage.SetActive(true);

            inBuyZone = true;
        }
    }
    private void OnTriggerExit2D(Collider2D other)
    {
        if (other.tag == "Player")
        {
            buyMessage.SetActive(false);

            inBuyZone = false;
        }
    }
}
```