

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**  
**Львівський національний університет імені Івана Франка**  
**Факультет електроніки та комп'ютерних технологій**  
**Кафедра радіоелектронних і комп'ютерних систем**

Допустити до захисту  
Завідувач кафедри  
Проф. Оленич І.Б.  
«\_\_\_\_\_»\_\_\_\_2025 р.

**Кваліфікаційна робота**

**Бакалавр**  
(освітній ступінь)

**Розробка комп'ютерної гри в жанрі rogue-like**

Виконав:

студент IV курсу групи ФєП-42  
спеціальності 121 – Інженерія  
програмного забезпечення

Б.М. Головчак

Науковий керівник:

ас. В.Т. Гура

« \_\_\_\_ »\_\_\_\_2025 р.

Рецензент:

\_\_\_\_Асист. Мисюк І.В.

Львів 2025

## **АНОТАЦІЯ**

Ця бакалаврська робота описує весь процес розробки Rogue-like гри в жанрі 3D-шутера з видом зверху з використанням ігрового рушія та середовища Unity. Основною метою цього дослідження є вивчення виробництва ігор з акцентом на випадкову генерацію всього ігрового процесу та його реіграбельність. У роботі описано ключові етапи розробки ігор цього жанру, а саме: створення головного героя, розробка механіки маніпуляції персонажем та бойової механіки, а також принципи та розробка процедурної генерації рівнів та випадкового розміщення об'єктів. Значна увага також приділяється розробці штучного інтелекту ворогів, калібруванню ускладнюючих систем та реалізації візуальних ефектів. Основні виклики в розробці передбачають балансування складності гри, оптимізацію процедурної генерації та адаптацію гри до гравця. У цьому контексті використання Unity дозволило мені ефективно вирішити ці питання завдяки багатофункціональним інструментам, таким як NaviMesh та Cinemachine. Отримані результати продемонстрували можливості середовища Unity для виробництва ігор та довели, що жанр Rogue-like є цілком життєздатним у сучасному світі. Практична значущість проекту полягає у розвитку навичок розробки ігор та підготовці їх до подальшої комерціалізації.

## **ANNOTATION**

This bachelor's thesis describes the entire process of developing a Rogue-like game in the genre of 3D top-down shooter using the Unity game engine and environment. The main goal of this research is to study game production with an emphasis on random generation of the entire gameplay and its replayability. This work describes the key stages of game development in this genre, namely: creating the main character, developing the mechanics of character manipulation and combat mechanics, as well as the principles and development of procedural generation of levels and random placement of objects. Considerable attention is also paid to the development of enemy artificial intelligence, calibration of complicating systems, and implementation of visual effects. The main challenges in development include balancing the complexity of the game, optimizing procedural generation and

adapting the game to the player. In this context, using Unity allowed me to effectively solve these issues owing to multifunctional tools such as NaviMesh and Cinemachine. The results demonstrated the capabilities of the Unity environment for game production and proved that the Rogue-like genre is quite viable in the modern world. The practical significance of the project lies in the development of game development skills and their preparation for further commercialization.

## ЗМІСТ

ВСТУП .....	5
РОЗДІЛ 1 МЕТОДОЛОГІЧНІ ЗАСАДИ ТА СУЧАСНИЙ СТАН РОЗРОБКИ ІГОР .....	7
1.1 Загальні відомості про основну тему роботи.....	7
1.1.1 Визначення жанру Rogue-like: історія виникнення та ключові особливості. ....	7
1.1.2. Сучасний стан індустрії відеоігор і місце Rogue-like у ній. ....	7
1.1.3. Популярні приклади Rogue-like ігор: аналіз їх механік і успіху. ....	9
1.1.4. Роль процедурної генерації, нелінійності та інших особливостей у формуванні унікального ігрового досвіду. ....	11
1.1.5. Вплив технологій розробки (Unity, Unreal Engine) на жанр. ....	12
1.2 Алгоритми досягнення поставленої мети .....	13
1.2.1 Процедурна генерація рівнів .....	13
1.2.2 Механіка постійної смерті.....	14
1.2.3 Балансування складності та варіативність геймплею .....	15
РОЗДІЛ 2 СЕРЕДОВИЩЕ РОЗРОБКИ .....	17
2.1 Середовище розробки – ігровий двигун Unity.....	17
2.2 Мова програмування - C#.....	19
2.3 Середовище розробки коду – Visual Studio .....	20
РОЗДІЛ 3 ПРАКТИЧНА РЕАЛІЗАЦІЯ МЕТИ БАКАЛАВРСЬКОЇ РОБОТИ	23
3.1 Створення проекту та налаштування сцени в Unity.....	23
3.2 Створення головного героя, його керування та характеристики.....	25
3.3 Створення генерації карти, її взаємодії з головним героєм та відображення на екрані .....	35
3.4 Створення Бомби .....	59
3.5 Система здоров'я.....	63
3.6 Противники та алгоритм їх поведінки.....	67
ВИСНОВОК .....	79
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ .....	80

## ВСТУП

Ігрова індустрія - одна з найбільш динамічних і перспективних галузей сучасних технологій, що поєднує в собі творчість, програмування та інженерні рішення. Жанр Rogue-like є одним з найпопулярніших жанрів інді-ігор завдяки своїй унікальній ігровій механіці, такій як повністю випадково згенеровані рівні, постійна смерть протагоніста та неймовірна варіативність ігрового процесу. Ці особливості роблять Rogue-like ігри дуже популярними серед людей, які люблять складні виклики та глибокий ігровий досвід. Навчившись розробляти ігри цього жанру, можливо не тільки вивчити всі важливі аспекти цього жанру, але й вдосконалити свої техніки та навички створення ігор, використовуючи новітні інструменти розробки, такі як ігровий двигун Unity. З огляду на зростаючий інтерес до цього ігрового жанру, ця тема залишається важливою як в контексті розвитку ігрової індустрії в цілому, так і з точки зору набуття нових знань, практичних і теоретичних навичок. Метою даної бакалаврської роботи є створення прототипу гри з використанням новітніх технологій та інструментів розробки, що відповідають особливостям жанру. Основними частинами, які будуть реалізовані, є процедурна генерація рівнів, яка щоразу створює унікальний ігровий досвід, механіка постійної смерті, яка пропонує гравцям новий виклик і змушує їх мислити більш стратегічно та нелінійний геймплей, який заохочує гравців досліджувати локації та цілі карти. Додатково до технічних елементів дослідження, будуть також розглянуті творчі аспекти, такі як дизайн персонажів, анімація, естетика та складність балансу. Об'єктом цього дослідження є процес створення відеогри в жанрі "Rogue-like" і охоплює всю розробку від її концепції та дизайну до технічної реалізації. Предметом дослідження є технічні та концептуальні елементи розробки ігор у цьому жанрі. Сюди входить використання процедурної генерації для випадкової побудови рівнів, програмування основних ігрових механік та створення відчуття складності. Окрім того, досліджується використання сучасних інструментів розробки. Новизна даної роботи полягає в поєднанні сучасних методів процедурної генерації. Результати цього дослідження можуть бути використані для подальших розробок в освітніх цілях, а також для комерційних проєктів, спрямованих на створення конкурентоспроможних ігор. Також передбачається, що створені ігри можуть бути використані як основа для впровадження та розвитку нових функцій,

таких як: розширення навичок та механік персонажів, включення кооперативних ігрових режимів та підтримка інших платформ, до прикладу консолі та смартфони. Крім того, процедурна генерація може бути використана для інших жанрів, таких як платформні ігри. Жанр Rogue-like користується неймовірним попитом серед геймерів завдяки своїй складності, високій реіграбельності та унікальності кожної ігрової сесії та ігрового процесу. На ринку існує величезний попит на інді-ігри в цьому жанрі, що полегшує та заохочує створення нових продуктів. Вибір ігрового двигуна Unity в якості основного інструменту розробки обумовлений його гнучкістю, підтримкою великої кількості платформ і дуже активною спільнотою розробників, що дозволяє легко створювати нові ідеї і прискорювати робочі процеси. Ігрова індустрія продовжує стрімко розвиватися, охоплюючи тисячі людей щодня. Особливе місце в цій індустрії займають інді-ігри, які завжди експериментують і проявляють власний творчий потенціал. Жанр Rogue-like - лише один із прикладів того, як невеликий проект може вплинути на весь світ. Такі відомі проекти, як «The Binding of Isaac», «Hades», «Dead Cells», «Risk of Rain 2», «Cult of the Lamb» та інші, є популярними та конкурентоспроможними продуктами з великою увагою до деталей, оригінальності та вміння адаптувати класичні концепції до сучасних стандартів. Дослідження цього процесу має як практичну, так і академічну цінності. Це дослідження може розширити знання про алгоритми, інтеграцію художніх і творчих елементів у іграх та розвиток ігрової індустрії на загал. Застосування цих знань полегшить підготовку майбутніх фахівців у цій галузі та дозволить українським розробникам конкурувати на міжнародній арені. Обрана тема є вкрай важлива та матиме значний вплив на розвиток української індустрії розробки ігор. Дослідження дозволить виявити всі основні аспекти та виклики у створенні Rogue-like ігор, які можуть бути використані як початківцями, так і досвідченими розробниками. Також дослідження сприятиме розвитку творчого мислення та використання сучасних інструментів розробки.

## **РОЗДІЛ 1 МЕТОДОЛОГІЧНІ ЗАСАДИ ТА СУЧАСНИЙ СТАН РОЗРОБКИ ІГОР**

### **1.1 Загальні відомості про основну тему роботи**

#### **1.1.1 Визначення жанру Rogue-like: історія виникнення та ключові особливості.**

Жанр Rogue-like є одним з найбільш унікальних і впізнаваних жанрів в індустрії відеоігор. Вважається, що жанр бере свій початок з відеогри Rogue 1980 року, створеної Майклом Тоєм, Гленом Віхтманом та Кеном Арнольдом [10]. Основи жанру були закладені завдяки інноваційним підходам до ігрового дизайну того часу. Наприклад, випадкова генерація підземель забезпечувала унікальність кожного проходження, а постійна смерть робила гру складною та непередбачуваною. Завдяки елементам стратегічного мислення та глибокої несподіванки, Rogue завоювала серця багатьох гравців того часу. З часом цей жанр набув деяких вкрай характерних елементів, які є актуальними і сьогодні. Процедурна генерація (кожне підземелля або карта генеруються випадковим чином), постійна смерть (після смерті персонажа гравцеві доводиться починати всю гру з початку), нелінійний геймплей (гравець завжди вільний у виборі власної стратегії та підходу до проходження), унікальність кожного проходження, що тільки підкреслює неповторний характер гри. Всі сучасні розробники Rogue-like ігор використовують такі елементи, як процедурна генерація, постійна смерть і нелінійність, щоб створювати ігри, які сподобаються широкій аудиторії. Загалом, ігри цього жанру приваблюють людей своїм високим рівнем складності, але вона не є надто складною, і кожне проходження дає гравцеві відчуття виконаного завдання. Унікальна атмосфера і постійна складність Rogue-like ігор створюється поєднанням постійної смерті та обмежених ресурсів. Це змушує гравця ретельно обдумувати і аналізувати майбутні дії та приймати правильні рішення в умовах обмеженого часу.

#### **1.1.2. Сучасний стан індустрії відеоігор і місце Rogue-like у ній.**

Індустрія відео ігор-одна з найбільших і найдинамічніших галузей розваг у сучасному світі, яка щороку зростає завдяки технічному прогресу та інтернет технологіям. Сучасні ігри охоплюють дивовижний спектр жанрів і платформ: від невеликих інді-ігор для мобільних пристроїв до дуже великих і

складних AAA-проектів з реалістичною графікою для консолей і ПК. У 2024 році ринок відео ігор досяг понад 200 мільярдів доларів США завдяки розвитку PlayStation 5, Xbox Series X|S, хмарних сервісів (GeForce NOW, Xbox Cloud Gaming) і портативних консолей (Steam Deck, Nintendo Switch, ROG Ally), а кількість гравців у всьому світі становитиме понад 3 мільярди людей [12]. Розвиток цих ігрових платформ, сприяв широкій доступності відеоігор для всіх груп користувачів. Крім того, стрімінгові платформи, такі як, Twitch, Kick, YouTube також користуються великою популярністю та мають значний вплив на розвиток аудиторії та ігрової спільноти в цілому.

На тлі стрімкого розвитку індустрії відеоігор жанр Rogue-like здобув дуже високу репутацію і стає все більш популярним серед геймерів. Завдяки своїй оригінальності цей жанр став одним з найпопулярніших не тільки серед інді-розробників, а й серед великих студій, які випускають проекти з неймовірним рівнем комерційного успіху.

Одна з головних причин, чому геймери віддають перевагу Rogue-like іграм - це їхня реіграбельність. Завдяки процедурній генерації кожна ігрова сесія або проходження гри є унікальними і завжди відчуються по-новому, незалежно від того, скільки разів ви в їх проходите. Користувачам подобається стикатися з новими та непередбачуваними сценаріями, з якими вони раніше не мали справи, і адаптуватися до них. Гравцям також імпонує високий рівень складності, який змушує їх більше працювати і думати над тим, щоб досягти успіху в грі.

Жанр Rogue-like стає знаменитішим завдяки своїй простоті використання. Процедурна генерація рівнів автоматично генерує частини гри, зменшуючи ручну роботу при створенні контенту і дозволяючи їм зосередитися на розробці механік та балансу. Багато ігор цього жанру здобувають прихильність гравців усіх рівнів вмінь, до прикладу: «The Binding of Isaac», «Dead Cells», «Darkest Dungeon», «Enter the Gungeon», «Hades», «Cult of the Lamb» та «Balatro» - зразки успішних проектів інді-розробки, які знайшли своїх гравців та стали впізнаваними серед ігрової спільноти.

Жанр Rogue-like сприяє створенню численних спільнот геймерів, яких об'єднує жага до випробувань та обговорення нових стратегій. Спільний інтерес цих спільнот також проявляється у створенні модифікацій ігор у вигляді модів та фан-арту. Існування жанру Rogue-like та подібних йому ігор,



таких як The Binding of Isaac та Balatro, доводить, що для залучення тисяч гравців щодня необхідна не лише вражаюча графіка, а й незвичайний та неповторний контент. Підтвердженням цього слугує факт що, 3.8 років тому в The Binding of Isaac грали понад 70 000 гравців водночас, а зараз в неї грають більше ніж 32 000 гравців по всьому світу одночасно (див Рис.1.1). Таким чином, жанр Rogue-like доводить свою значущість в сучасному світі та індустрії відеоігор [2].

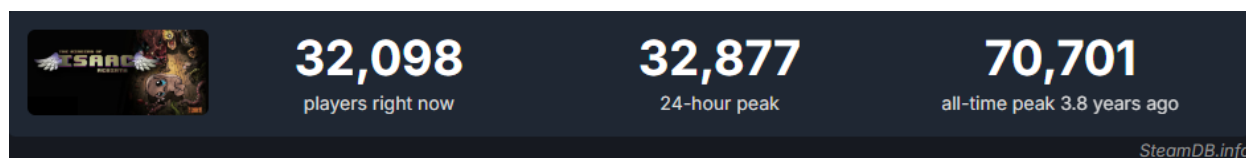


Рис.1.1 Кількість одночасних гравців у грі “The Binding of Isaac”

### 1.1.3. Популярні приклади Rogue-like ігор: аналіз їх механік і успіху.

Жанр Rogue-like зарекомендував себе завдяки своїм неповторним механікам, неймовірній реіграбельності та здатності викликати емоції у гравців. Ми розглянемо деякі з найбільш культових ігор цього жанру та проаналізуємо їхні основні механіки, причини успіху та вплив на жанр.

#### **The Binding of Isaac**

##### *Основні механіки:*

1. випадково згенеровані кімнати та рівні роблять кожне проходження унікальним;
2. велика кількість предметів (719), які взаємодіють між собою в певних ситуаціях, до прикладу, Spirit sword та Technology, які разом створюють лазерний меч з відомої серії фільмів Star Wars (див. рис. 1.2);

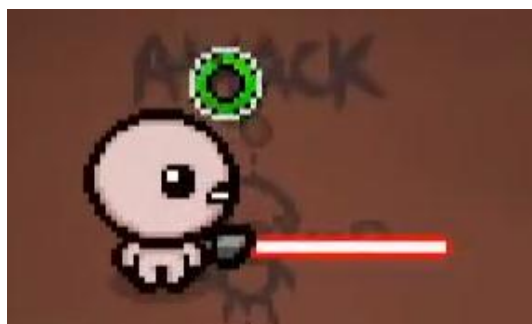


Рис. 1.2. Комбінація Spirit Sword та Technology

3. постійна смерть підвищує вагомість кожного предмета та процес проходження гри;

4. проста, але водночас достатньо складна бойова система, щоб провокувати гравця.

*Причини успіху:*

1. унікальний візуальний стиль і глибока сюжетна лінія з біблійними мотивами;

2. величезна кількість контенту (боси, персонажі, рівні, вороги, секрети, завдання);

3. багаторічна підтримка розробника — Едмунда Макміллена.

**Hades**

*Основні механіки:*

1. процедурно згенеровані рівні з неповторними «поверхами»;

2. після поразки гравці приносять ресурси в ігровий штаб, щоб покращити свого персонажа;

3. глибокий сюжет, заснований на грецькій міфології, з батьком Загреуса у вигляді Хейдса;

4. різноманітна зброя та стилі бою, щоб не давати користувачам занудьгувати.

*Причини успіху:*

1. видатна анімація та графічна робота, що відповідає рівню AAA-компаній та їхніх розробок;

2. кожен другорядний персонаж має власну історію та місію;

3. гра добре збалансована, достатньо складна для гравців, які хочуть випробувати себе, і достатньо проста для новачків;

4. нагороди та номінації, такі як «Гра року 2020», «Найкращий наратив» та «Найкраща інді-гра» [15].

Успіх цих відеоігор підтверджує, що Rogue-like ігри мають потенціал для процвітання та залучення аудиторії. Популярність The Binding of Isaac and Hades надихнула тисячі інді-розробників на створення подібних ігор та на створення унікальних проектів. Таким чином, аналіз цих популярних прикладів показує, що цей жанр залишається одним з найперспективніших у індустрії відеоігор

#### **1.1.4. Роль процедурної генерації, нелінійності та інших особливостей у формуванні унікального ігрового досвіду.**

Ігри в стилі Rogue залишаються актуальними завдяки певним особливостям, які підтримують незрівнянність кожної гри, однією з котрих є процедурна генерація, яка доповнює нелінійність гри. Процедурна генерація — це технологія, яка використовує алгоритми в реальному часі для генерації ігрових карт і місцевості. Саме цей вид генерації дозволяє гравцям щоразу стикатися з новими комбінаціями, розташуванням кімнат, ворогами та предметами, завдяки чому підтримує інтерес до гри. Вона також економить використання сил та ресурсів на самостійне створення величезної кількості рівнів, оскільки існує алгоритм, який може зробити це замість вас.

Нелінійність у Rogue-like іграх представлена свободою вибору гравця. Гравці можуть вибирати, які рівні досліджувати, які ресурси використовувати і як саме ці рівні досліджувати. Нелінійність дозволяє пристосувати гру до стилю гравця. Наприклад, в Hades гравці можуть обирати здібності та зброю відповідно до своїх уподобань. Навіть якщо гра має простий сюжет, нелінійність жанру дозволяє гравцям обирати власний шлях до кінця, що створює особливу винятковість. Саме завдяки цьому гравці будуть повертатися до гри щоразу, щоб спробувати нові ідеї та тактики.

Серед інших унікальних механік — постійна смерть (з англ. Permadeath), система, де смерть персонажа означає повернення його на початок гри, що, також, заохочує шукати нові стратегії і ретельно продумувати подальші дії. Permadeath робить кожну спробу важливою, а кожне проходження — визначним досягненням.

У багатьох сучасних Rogue-like іграх також присутня прогресія персонажа. Наприклад, з кожною смертю гравці отримують певні ресурси для покращення свого персонажа або лігва. Ця прогресія компенсує біль і розчарування, які може спричинити смерть персонажа. Rogue-like ігри видатні своєю складністю. Це приваблює певний вид гравців, яким подобається долати серйозні випробування. Високий рівень складності сприяє швидкому розвитку навичок, стратегічного мислення та наполегливості. Процедурна генерація, нелінійність і постійна смерть — основні елементи, які формують жанр Rogue-like. Ці механіки розвивають індустрію в цілому та впливають на створення

нових ігор. Ігри цього жанру стали невід'ємною частиною сучасної ігрової культури.

#### **1.1.5. Вплив технологій розробки (Unity, Unreal Engine) на жанр.**

Сучасні ігрові двигуни, такі як Unity та Unreal Engine, мали значний вплив на розвиток ігрової індустрії та ігор в стилі Rogue, надаючи розробникам інструменти для створення складних та візуально захоплюючих ігор. Ці рушії відкрили нові можливості для інтеграції всіх передових технологій у процес розробки, спростили доступ до створення своїх проектів для інді-розробників та допомогли знизити бар'єри для входження в індустрію розробки ігор.

Гнучкість, простота використання та універсальність Unity сприяють його популярності серед інді-розробників. Підтримка процедурної генерації в Unity дозволяє розробникам легко вбудовувати алгоритми процедурної генерації. Це особливо важливо для жанру Rogue-like. Репозиторій ресурсів Unity Asset Store також дає розробникам можливість зосередитися на розробці механік і процедурної генерації [17]. Прикладом успішного використання Unity у жанрі Rogue-like є гра Dead Cells, яка поєднує процедурну генерацію з плавною анімацією.

Unreal Engine вирізняється своєю здатністю забезпечувати фотореалістичну графіку та реалістичну фізику. Рушій в першу чергу використовується для створення AAA проектів, але також підходить для Rogue-like ігор. За допомогою двигуна можна створювати атмосферні та деталізовані світи. У жанрі Rogue-like його можна використовувати для створення унікальних візуальних стилів. До прикладу, інтерактивні об'єкти на рівні можна використовувати для створення непередбачуваних ситуацій

Прикладом успішної гри в жанрі Rogue-like з використанням Unreal Engine є Risk of Rain 2 [1]. Гра використовує рушій Unreal Engine для створення величезних рівнів з чудовою графікою та унікальним ігровим процесом. Він не тільки спрощує розробку ігор, але й стимулює розвитку творчості розробників і дозволяє створювати захопливі проекти, які відповідають сучасним сподіванням. Завдяки цим інструментам жанр Rogue-like продовжує процвітати.

## 1.2 Алгоритми досягнення поставленої мети

### 1.2.1 Процедурна генерація рівнів

Процедурна генерація карт є важливою складовою жанру Rogue-like, що дозволяє гравцям створювати інший ігровий світ кожного разу, коли вони намагаються розпочати гру. Такий підхід не тільки додає грі різноманітності та складності, але й допомагає підтримувати інтерес гравців завдяки непередбачуваності кожного рівня.

Процедурна генерація в основному базується на алгоритмах, які генерують ігровий контент (карти, рівні, об'єкти, ворогів, тощо) в реальному часі або автоматично перед початком ігрової сесії. Такий підхід зменшує кількість ручної роботи та економить час і ресурси, оскільки розробникам не потрібно створювати кожен рівень власноруч. Це також може створити варіативність в ігровому процесі, оскільки кожне проходження в грі залишається цікавим завдяки постійним змінам карти. Існує кілька основних і поширених алгоритмів процедурної генерації карт в іграх:

1. кліткові автомати, метод в якому використовується сітка клітинок, кожна з яких може бути активною або неактивною. Алгоритм ітераціями змінює стан цих клітинок згідно заданих правил, поки не буде створено карту. До прикладу гра, яка використовує клітковий автомат є Unexplored [11];

2. алгоритм розділення простору – це метод, який поділяє простір на менші частини, а потім об'єднує їх коридорами. До прикладу використання цього алгоритму, Dungeons of Dredmor [8];

3. графові алгоритми, в цьому алгоритмі рівні генеруються як графи, де вузли представляють кімнати, а ребра – з'єднання між ними, гра яка це використовує – Slay the Spire [9];

4. шум Перліна, це метод в якому передбачається створення високо деталізованих структур за допомогою математичних функцій, що імітують природу. Гра яка використовує такий метод – Minecraft [14].

Хоча процедурна генерація має багато переваг, вона також має і підводні камені, такі як контроль якості, нелогічні або занадто прості рівні, балансування, необхідність забезпечення однакового рівня складності для всіх гравців, унікальність та надмірне використання одних і тих самих об'єктів.

Сучасні ігрові двигуни, такі як Unity та Unreal Engine, використовують потужні інструменти для реалізації процедурної генерації. Unity використовує власний API та сторонні плагіни, а Unreal Engine дозволяє створювати алгоритми без програмування, використовуючи Blueprints. Процедурна генерація рівнів є одним з найважливіших інструментів у розробці Rogue-like ігор, що допомагає підвищити інтерес та зацікавленість гравців. Ефективне використання процедурної генерації — одне з головних завдань розробників ігор цього жанру.

### **1.2.2 Механіка постійної смерті**

Механізм постійної смерті (з англ. Permadeath) є одним з ключових складових Rogue-like ігор і відіграє важливу роль у формуванні ігрового досвіду. Ця механіка передбачає повну втрату прогресу в разі загибелі персонажа, що в свою чергу надає особливого значення кожній спробі проходження, кожному рішенню та діям гравця.

*Основними аспектами постійної смерті є:*

1. повна втрата прогресу. В разі смерті персонажа всі предмети, рівні, характеристики втрачаються та гра починається спочатку, що забезпечує значущість кожної спроби.
2. емоційний вплив. Кожна перемога приносить сильне емоційне задоволення досягненням, у той час як смерть змушує все починати спочатку.
3. Стимул реіграбельності, оскільки після кожної смерті гра починається спочатку, це мотивує гравця постійно пробувати нові підходи та стратегії для проходження того чи іншого сценарію.

Механіка постійної смерті бере свій початок ще з оригінальної гри «Rogue». Ця концепція була використана для створення складного та захопливого ігрового процесу. Пізніше ця механіка поступово еволюціонувала і стала основою для багатьох, якщо не всіх, ігор цього жанру. У минулому ця механіка використовувалася через обмеження тогочасного обладнання, яке не мало достатньо пам'яті для зберігання всієї інформації. В сучасних іграх Permadeath є свідомим дизайнерським рішенням для підвищення складності та цікавості гри.

Permadeath є ключовим елементом усіх Rogue-like ігор і має значний вплив на цей жанр. Вимагаючи від гравців стратегічного мислення та

ретельного аналізу кожної дії, щоб не починати все спочатку, механіка постійної смерті стимулює реіграбельність і робить кожне успішне проходження незабутнім. Її вдала реалізація є викликом для будь-якого розробника, але при цьому сприяє створенню унікального продукту, який здатний захоплювати аудиторію на сотні годин.

### **1.2.3 Балансування складності та варіативність геймплею**

Баланс складності та варіативний геймплей є ключовими елементами в розробці Rogue-like ігор, котрі безпосередньо впливають на відчуття задоволення гравців, мотивацію до повторного проходження та загальний успіх гри. Ці елементи спрямовані на створення ігор, які є одночасно складними і цікавими, а також забезпечують різноманітний і захоплюючий досвід для гравця.

Баланс рівня складності забезпечує відповідне відчуття виклику, коли гра легка і складна водночас, для створення відчуття напруги. Гравці повинні відчувати, що їхній прогрес залежить від власних навичок і поведінки, а не від абсолютно випадкового елементу, над яким вони не мають контролю.

У деяких іграх у стилі Rogue-like використовується динамічний баланс складності в залежності від прогресу гравця. Наприклад, у The Binding of Isaac рівень складності зростає відповідно до кількості здобутих предметів і ресурсів. Розробники завжди намагаються проектувати свої ігри таким чином, щоб ігровий світ ставав дедалі складнішим, але при цьому залишав гравцеві можливості для адаптації. Тестування є важливим кроком у визначенні складності гри. Спостереження за тим, як різні гравці взаємодіють з грою, допомагає виявити і вирішити проблеми різного виду.

Нові механіки слід вводити поступово і давати гравцям змогу вивчити основи, перш ніж вони зіткнуться з більш складними механіками. Наприклад, перший рівень є простіший і поблажливіший, тоді як останній рівень є напрочуд складним і непередбачуваним. Різноманітність ігрового процесу гарантує, що кожна ігрова сесія буде унікальною, і досягається завдяки процедурній генерації, випадковому розподілу ресурсів та нелінійному геймплею.

Різноманітність забезпечується додаванням багатьох видів ворогів різних типів та поведінки, широким спектром зброї та предметів для

постійного вибору та зміни стилів гри, а також системою розвитку персонажа, яка дозволяє гравцям розвиватися після кожної смерті.

Дуже важливо вміти справлятися з такими проблемами, як випадкова несправедливість, моменти в яких прогрес гравця та його здоров'я залежить не від навичок самого гравця, а від випадкової послідовності подій, збереження мотивації гравця, оскільки якщо гра буде занадто складною, це може відштовхнути менш досвідчених гравців, а якщо буде занадто легкою, можна втратити любителів хардкору. Балансування складності та забезпечення різноманітності в ігровому процесі є ключовими елементами успішної Rogue-like гри. Вони створюють виклики і стимул для гравців повертатися до гри знову і знову. Успішна реалізація цих елементів може створити гру, яка приваблює гравців із різним рівнем навичок і залишається актуальною в довгостроковій перспективі.



## РОЗДІЛ 2 СЕРЕДОВИЩЕ РОЗРОБКИ

### 2.1 Середовище розробки – ігровий двигун Unity

Unity було обрано для розробки 3D Rogue-like гри завдяки поєднанню виняткової продуктивності, гнучких інструментів візуалізації та процедурної генерації, широкої мультиплатформеної підтримки, зручного редактора, потужної архітектури скриптування на C# і доступності для невеликих команд. Усе це робить Unity ідеальним середовищем для створення масштабних, гнучких та ефективних проєктів у жанрі Rogue-like. Одним із вирішальних моментів встала потужна продуктивність рушія, особливо ефективна обробка фізики та оптимізоване відображення. Unity гарантує безперебійну роботу великої кількості об'єктів завдяки гнучким налаштуванням фізичних симуляцій, а сучасна графіка, зокрема Universal Render Pipeline (URP), дозволяє створювати насичені 3D-середовища з мінімальним навантаженням на GPU. URP надає такі можливості, як **Deferred+** рендеринг та Variable Rate Shading, що особливо актуально в іграх Rogue-like, де на екрані одночасно може бути багато противників, джерел світла та ефектів [13]. Unity також надає широкі можливості для процедурної генерації, що є базовим елементом ігрової механіки в більшості Rogue-like ігор.

Завдяки використанню скриптів на C# та підтримці технології Data-Oriented Technology Stack (DOTS), розробники здатні реалізовувати доволі комплексні алгоритми генерації рівнів, створення ворогів, предметів і різних подій з високим рівнем продуктивності. Системи C# Job System і Burst Compiler забезпечують ефективне розпаралелювання обчислень, що значно збільшує швидкість роботи навіть у сценах, де налічуються тисячі активних об'єктів.

Мультиплатформеність Unity — ще одна важлива перевага. Двигун підтримує понад 25 цільових платформ, включаючи Windows, macOS, Linux, Android, iOS, а також сучасні ігрові консолі (PlayStation, Xbox, Nintendo Switch). Це дає змогу без додаткових витрат і суттєвих змін у коді адаптувати гру для різних пристроїв, охоплюючи ширшу аудиторію гравців. Для AR/VR-платформ також доступні готові SDK-пакети (ARKit, ARCore, Oculus, HoloLens), що розширює потенціал гри в майбутньому. Наявність Unity Asset Store, який нараховує десятки тисяч готових розв'язків, суттєво прискорює

процес розробки. В магазині є 3D-моделі, анімаційні пакети, вже готові UI-набори. Завдяки цьому розробники мають можливість сконцентруватися на унікальній ігровій логіці, а не на створенні базового контенту. До деяких пакетів додається документація, скрипти та інтеграційні приклади, що особливо корисно для швидкого старту.

Ще одним важливим чинником є гнучкість Unity Editor. Його можна налаштувати відповідно до потреб команди: створювати користувацькі інспектори, макроси, автоматизовані інструменти для генерації рівнів чи перевірки сцен. Prefab-система дозволяє централізовано оновлювати всі екземпляри об'єктів, що особливо корисно при частих ітераціях та оновленнях контенту. Сцени й елементи гри легко перетворюються на багаторазові шаблони, що підтримують варіації, спадкування й інкапсуляцію властивостей. Скриптова архітектура Unity базується на C#, який підтримує як класичний об'єктно-орієнтований підхід, так і сучасні підходи, включно з використанням ScriptableObjects, які спрощують зберігання даних про ворогів, зброю, властивості кімнат чи рівнів. Такі об'єкти не потребують дублювання даних у кожній сцені й дають змогу легко масштабувати систему без порушення цілісності гри.

Крім того, Unity Personal є безкоштовною для індивідуальних розробників і невеликих студій з річним доходом до певного ліміту, надаючи повний доступ до рушія без обмеження функціоналу. У разі зростання проекту доступні версії Unity Plus і Pro з додатковими можливостями: пріоритетною техпідтримкою, Unity Analytics, Cloud Build, інтеграцією із сервісами CI/CD тощо [3]. Окрім цього, Unity оснащений функціями, на кшталт Occlusion Culling, що автоматично прибирає з рендеру об'єкти, приховані іншими. Це миттєво зменшує обсяг зайвої роботи. Така оптимізація критична для густонаселених рівнів в іграх типу Rogue like, де може бути велика кількість ворогів та інтерактивних елементів одночасно. LOD Group дозволяє замінювати детальні моделі менш складними, коли камера віддаляється. Це знижує кількість полігонів у сцені, майже не впливаючи на візуальну складову. Додатково, можливо налаштувати плавні переходи, з використанням Cross fade, щоб уникнути різкого оновлення моделей.

## 2.2 Мова програмування - C#

C# з'явився в 2000 році як частина .NET Framework і вже через два роки став міжнародним стандартом ECMA-334, закріпивши свій статус універсальної мови програмування з сильною статичною типізацією та підтримкою багатьох парадигм [18]. Початкові версії C# 1.0–2.0 принесли базові об'єктно-орієнтовані механізми та generics, після чого з релізом C# 3.0 з'явилися LINQ і анонімні типи, що значно спростило роботу з колекціями даних. Реформа .NET Core у 2016 році надала мові крос-платформене середовище виконання, дозволивши розгортати програми на Windows, Linux, macOS і мобільних платформах. Unity використовує C# 9 у профілі .NET Standard 2.1 (з частковою підтримкою C# 8 і 9) та інтегрує Roslyn-аналітики для перевірки якості коду, а для високопродуктивних обчислень пропонує Job System і Burst Compiler, а також Data-Oriented Technology Stack (DOTS) з архітектурою ECS. C# розроблявся під керівництвом Андерса Хейлсберга в команді Microsoft і вперше широко поширився в липні 2000 року разом із релізом .NET Framework. У грудні 2002 року специфікація C# 1.0 була прийнята як стандарт ECMA-334, а в 2003 році мова набула статусу ISO/IEC 23270, що гарантувало відкритість специфікації та захист від патентних претензій. Початкові випуски C# 1.0–2.0 (2002–2005) забезпечили базові OOP-концепції, події та generics для розробки типобезпечного коду без накладних витрат на кастинг. З релізом C# 3.0 (листопад 2007) були додані Language-Integrated Query (LINQ) і анонімні типи, що дозволили використовувати функціональні підходи при роботі з колекціями та запитами даних. У червні 2016 року світ побачив .NET Core 1.0 — повністю відкритий, модульний та крос-платформенний фреймворк. Він дав першому поколінню .NET змогу функціонувати не лише на Windows, а й на Linux та macOS. Завдяки реалізації .NET Standard 2.1 у .NET Core та наступних версіях (.NET 5/6/7), програми, написані на C#, можна компілювати в єдину бібліотеку для різних платформ, не переписуючи код. Починаючи з Unity 2021.3, двигун цілковито підтримує .NET Standard 2.1 та можливості C# 8.0, з частковою підтримкою C# 9.0. Це відкриває доступ до використання записних типів (records), pattern matching та інших передових функцій мови. Unity інтегрувала C# Job System та Burst Compiler, які відкривають можливості створення безпечних паралельних завдань і їх компіляції у високопродуктивний

машинний код на основі LLVM, демонструючи продуктивність, майже аналогічну нативному C++. Data-Oriented Technology Stack (DOTS) включає Entity Component System (ECS), що підвищує кеш-локальність даних і дає змогу опрацьовувати тисячі об'єктів з високою частотою кадрів, поєднуючись із Job System та Burst для масштабних ігор. Unity підтримує Roslyn analyzers та source generators для перевірки стилю й якості коду безпосередньо в IDE (Visual Studio, Rider), дозволяючи налаштовувати правила аналізу через файли ruleset і вбудовані пакети.

Використовуючи історичний шлях C#, його актуальні можливості, а також інтеграцію з Unity через .NET Standard, Job System, Burst, DOTS і Roslyn аналітики, стає реальним поєднання продуктивності та гнучкості при створенні масштабних 3D Rogue like ігор.

### **2.3 Середовище розробки коду – Visual Studio**

Visual Studio — це всебічне інтегроване середовище розробки (IDE) від Microsoft, яке створене, щоб об'єднати всі етапи створення програмного забезпечення: від написання коду, його налагодження до тестування та розгортання готових програм [16]. Завдяки підтримці найрізноманітніших мов програмування — .NET/C#, C++, Python, JavaScript, тощо — і можливості розширення функціоналу через офіційний Marketplace, Visual Studio задовольняє потреби як індивідуальних розробників (Community edition), так і великих команд (Professional та Enterprise editions). Останні оновлення додають глибоку інтеграцію з AI-асистентами (зокрема GitHub Copilot), котрі оптимізують швидкодію середовища та розширюють підтримку хмарних робочих процесів для безперервної розробки застосунків.

Історія Visual Studio починається у 1997 році, коли світ побачила версія Visual Studio 97 (під кодовою назвою Boston). Саме вона вперше поєднала в одному інтерфейсі Developer Studio інструменти для Visual Basic, Visual C++ та Visual FoxPro. Цей крок став фундаментом для майбутньої інтеграції різних середовищ розробки в єдиний пакет, значно полегшивши створення мультиплатформних рішень та зробивши їх більш доступними. З приходом “.NET-ери”, платформа зазнала значних перетворень: Visual Studio .NET 2002 (Rainier) відкрила підтримку .NET Framework 1.0, а до випуску VS 2008 (Orcas) система вже мала версію .NET Framework 3.5, яка включала в себе вбудовану

роботу з ASP.NET та єдину модель проектів MSBuild. У версії 2005 (Whidbey) з'явилися generics та покращене налагодження, а VS 2008 додала LINQ, анонімні типи та багатопоточність на рівні мови, що суттєво розширило можливості розробників при роботі з колекціями даних та паралельними алгоритмами. Перехід на новий рівень інтерфейсу відбувся з Visual Studio 2010 (Dev10): IDE перейшло на оболонку WPF, що відкрило шлях до гнучкішого та більш динамічного дизайну вікон, покращило підтримку мультимоніторних конфігурацій і дало змогу створювати кастомізовані розширення з багатшими UI-елементами. У релізах 2012–2013 років було впроваджено синтаксис C# 5 із ключовими словами `async/await`, що спростило написання асинхронного коду, а також значно покращено інструменти для веб-розробки. Далі, в 2015 році (Dev14), з'явилася підтримка .NET Core 1.0, а у VS 2017 (Dev15) — абсолютно новий інсталятор із можливістю вибіркового встановлення лише тих компонентів, які потрібні для конкретного проекту. Нова ера почалася з Visual Studio 2022 (Dev17) — першої 64-бітної версії IDE, яка подолати обмеження в 4 ГБ адресованої пам'яті та відкрила можливість працювати з надвеликими проектами без ризику “out-of-memory” помилок. Кожне значуще оновлення (17.x) приносить нових AI-помічників, зокрема інструменти на основі машинного навчання для аналізу коду, поліпшення загальної продуктивності середовища та ще глибшу інтеграцію з хмарними сервісами Azure для безперервної розробки.

Серцем Visual Studio є інтелектуальний текстовий редактор із контекстним синтаксичним підсвічуванням, системою шаблонів (snippets) і потужним автозавершенням IntelliSense, що дозволяє миттєво отримувати інформацію про типи, методи та параметри у коді C#, C++, JavaScript, Python тощо. Навігаційні інструменти — Go To Definition, Find All References, CodeLens — гарантують швидкий перехід між ключовими місцями в проекті, а можливість створювати власні фрагменти коду значно прискорює написання повторюваних конструкцій. Для забезпечення якості програмних рішень Visual Studio інтегрує Test Explorer, який підтримує MSTest, NUnit, xUnit та інші поширені фреймворки. Користувачі можуть запускати тести, групувати їх за категоріями, переглядати детальні звіти про успішні та неуспішні випадки без виходу з IDE. Вбудована підтримка паралельного виконання тестів і збірки

показників покриття коду дозволяє одержувати швидкий зворотний зв'язок про стабільність і якість змін в рамках CI/CD-процесів.

## РОЗДІЛ 3 ПРАКТИЧНА РЕАЛІЗАЦІЯ МЕТИ БАКАЛАВРСЬКОЇ РОБОТИ

### 3.1 Створення проекту та налаштування сцени в Unity

Для розробки гри в стилі 3D Rogue-like потрібно для початку створити сам ігровий проект для цього потрібно завантажити Unity з офіційного сайту та запустити Unity Hub (див. рис. 3.1.).

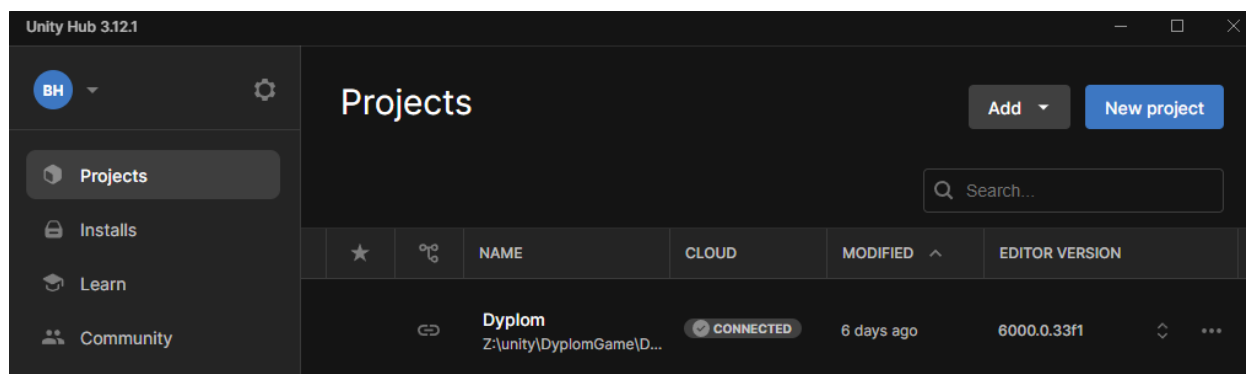


Рис 3.1. Unity Hub v3.12.1

Опісля, можна створити новий проект за допомогою кнопки New Project, де потрібно вибрати 3D Universal ядро за допомогою якого ми і будем реалізовувати проект подібного типу (див. рис. 3.2). Також, не завадить увімкнути Unity Version Control, що дозволить зберігати різні версії своєї гри без використання GitHub.

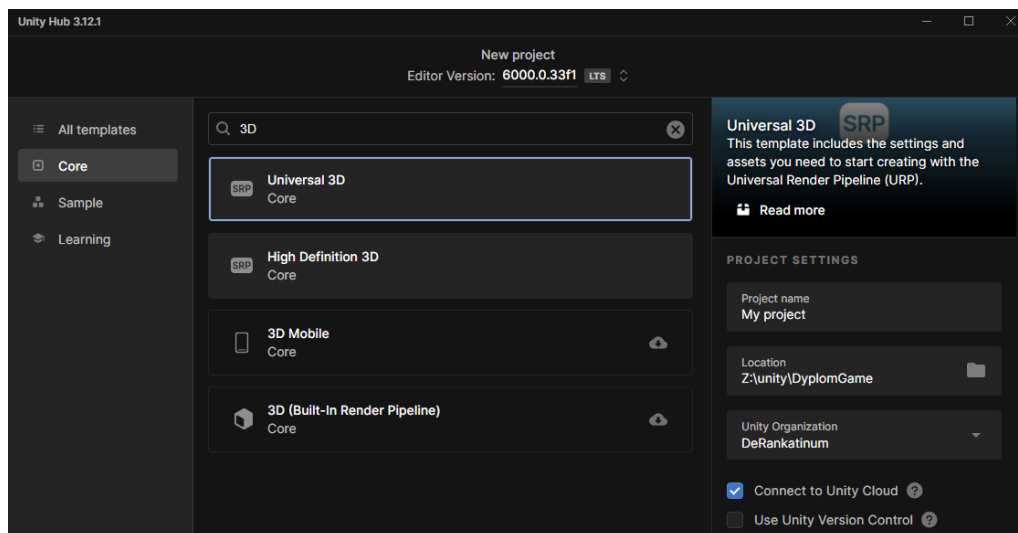


Рис 3.2. Екран створення проекту

Після того як ви оберете назву проекту та виберете правильне ядро можна створити сам проект і після тривалого завантаження вам відкриється порожній проект на екрані з пустою сценою, місце де буде створюватись практично уся гра (див рис 3.3).

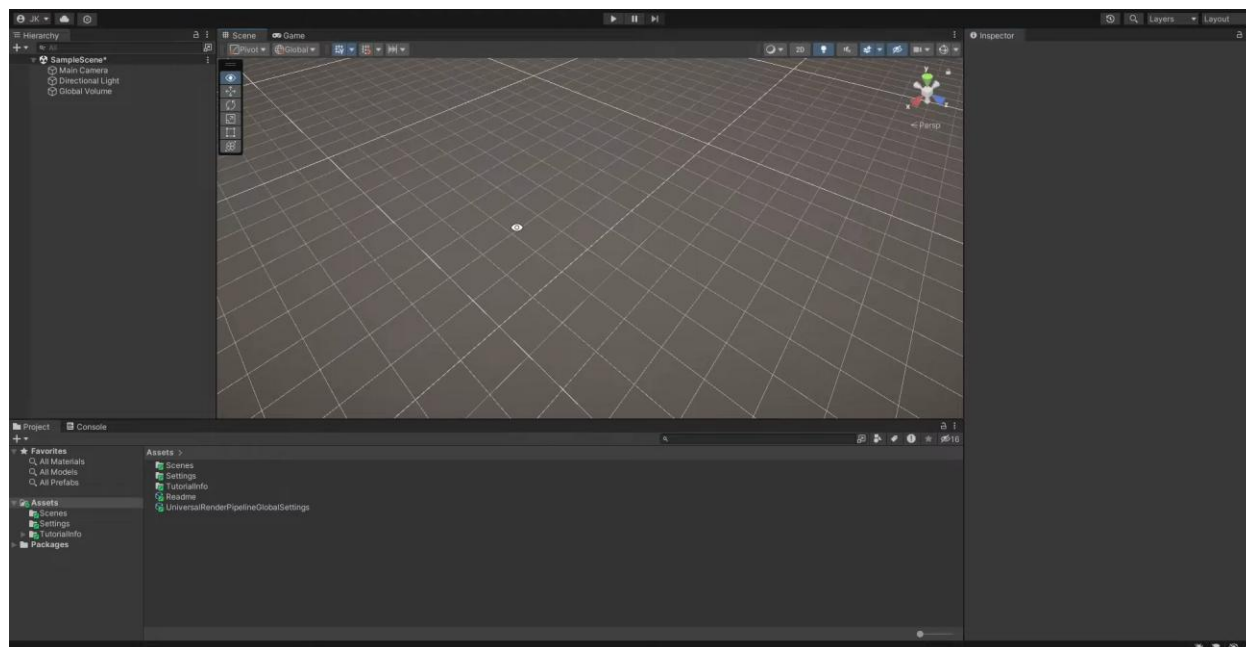


Рис 3.3. Пуста сцена після створення проекту

Це все що потрібно для початкового налаштування сцени та проекту.



### 3.2 Створення головного героя, його керування та характеристики

Для початку роботи зі створення головного героя варто створити підлогу, на якій ваш персонаж буде стояти, для цього потрібно натиснути на порожнє місце правою кнопкою миші та вибрати 3D objects >> Plane та поставити її місце розташування 0 0 0 (див. рис. 3.4).

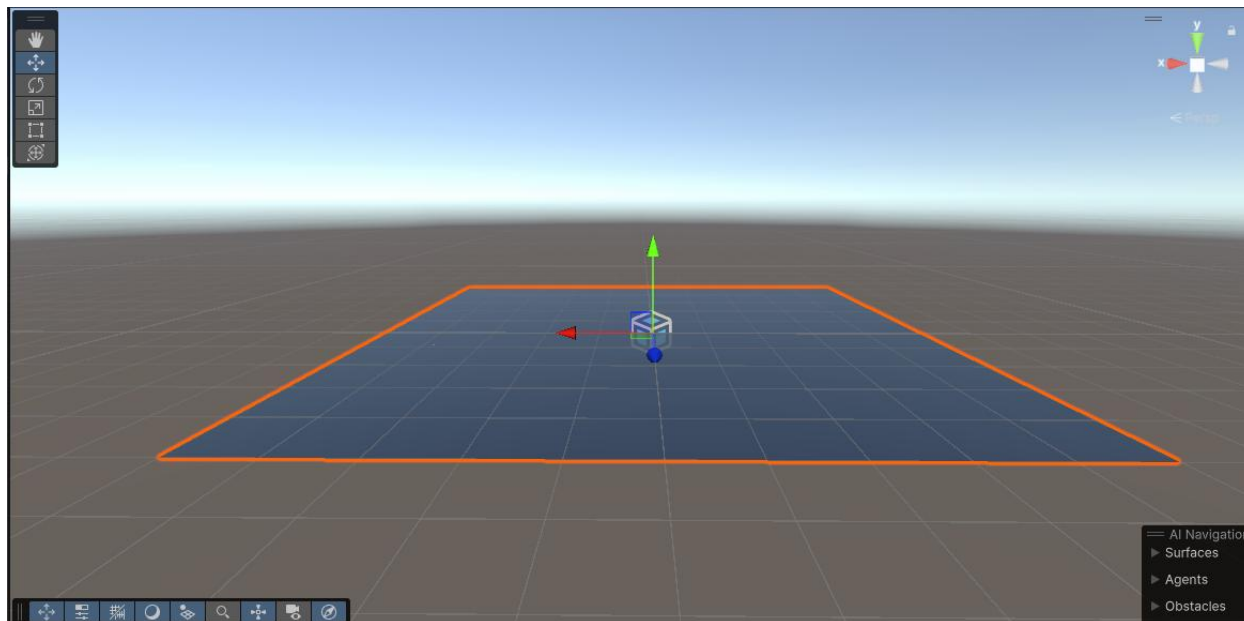


Рис 3.4. Пуста підлога (Plane)

Для імпортування вашої моделі головного героя бажано створити папку Models у вікні з файлами проекту для зручності навігації у майбутньому (див. рис 3.5.).

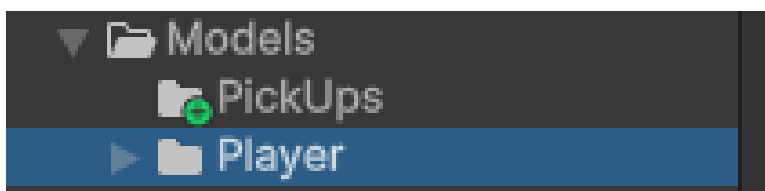


Рис 3.5. Створена папка Models та Папка Player

Згодом можна просто перетягнути файл з моделлю персонажа в дану папку. Коли файл із вашим персонажем з'явиться у цій папці, можна просто

перетягнути його у сцену і ваша модель з'явиться на місці вашого курсора. У моєму випадку головний герой має ось такий вигляд (див. рис. 3.6.).

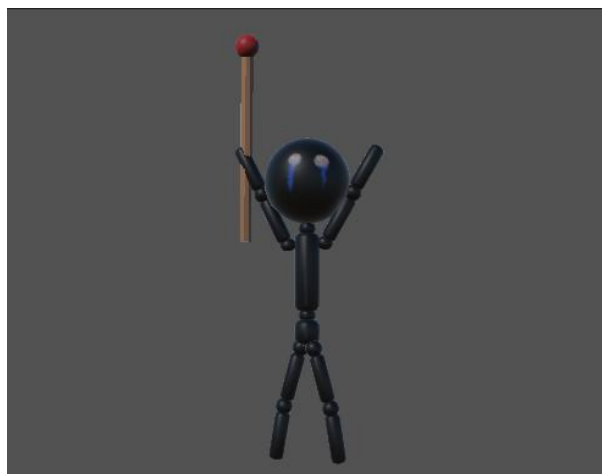


Рис 3.6. Модель головного героя

У мого персонажа разом з моделлю є ще деякі потрібні анімації, до прикладу: ходьба, стрільба, смерть, тощо. Для того щоб ці анімації можна було використати, потрібно створити аватар з даної моделі. Для цього потрібно вибрати модель героя у вашій папці, перейти у вкладку Rig та в полі Avatar Definition натиснути Create from this model, що створить аватар персонажа із вашої 3D моделі (див. рис. 3.7.).

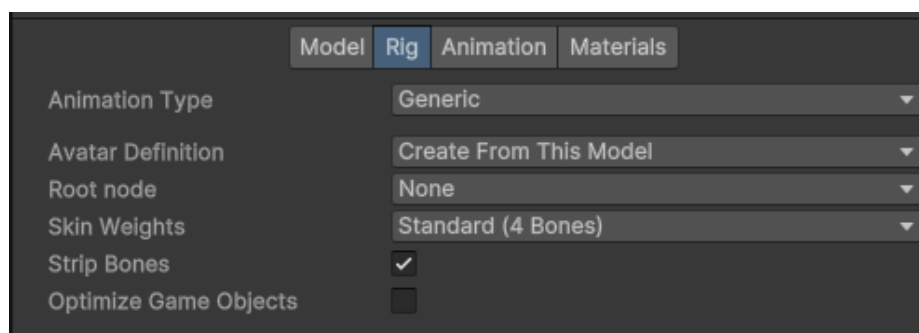


Рис 3.7. Створення аватару

Для того щоб додати свою першу анімацію потрібно вибрати вашого головного героя в меню Sample Scene та додати на нього Аніматор (див. рис. 3.8.).

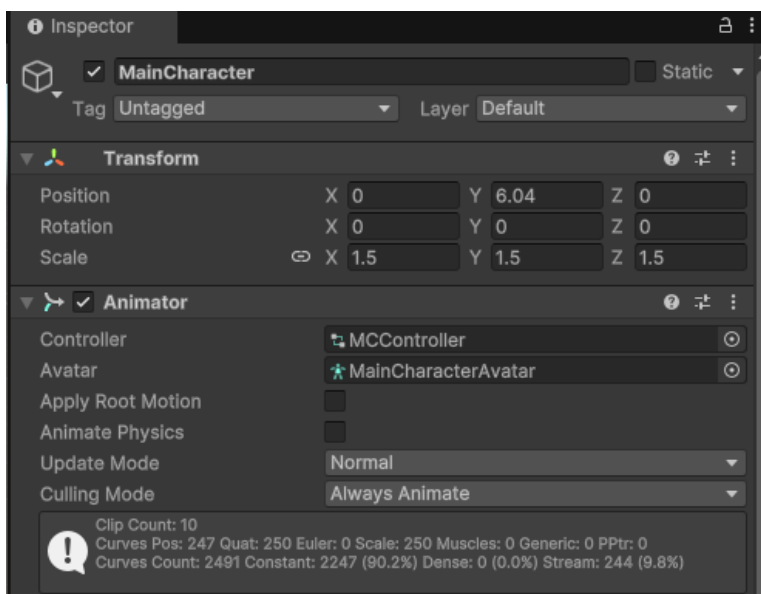


Рис 3.8. Меню інспектора разом із доданим аніматором

Для відтворення анімації потрібно перемістити створений аватар у поле Avatar, а також потрібно створити контролер для анімацій. Щоб досягнути цього результату, в папці з вашою моделлю натисніть права кнопка миші >> Create >> Animator Controller та дати йому назву на свій розсуд, в моїй ситуації анімаційний контролер називається MCController, що означає Main Character Controller. Цей контролер потрібно перенести в поле Controller у меню головного героя. По тому, потрібно відкрити сам контролер анімацій (див. рис. 3.9.).

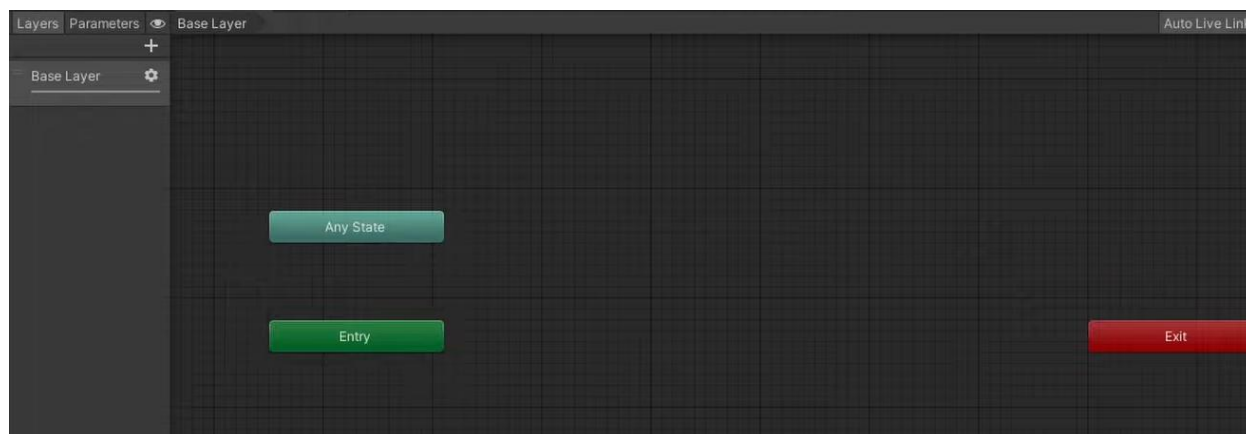


Рис. 3.9. Меню пустого контролера анімацій

У меню контролера потрібно створити новий стан, для цього потрібно натиснути правою кнопкою миші в доступному місці >> Create State >> Empty. Цей пустий стан персонажа вкрай важливо назвати правильно та запам'ятати цю назву, оскільки ми ще будемо повертатися до цього стану при написанні коду. Я назвав цей стан IdleAnim та додав до нього анімацію Idle (див. рис. 3.10.), яка буде відтворюватися, якщо персонаж нічого не робить, вона циклічна, тому буде відтворюватись безкінечно. Це буде, так званий, Default State, який буде відтворюватись одразу при запуску гри, тому він ідеально підходить для анімації, де ви ще нічого не робите.

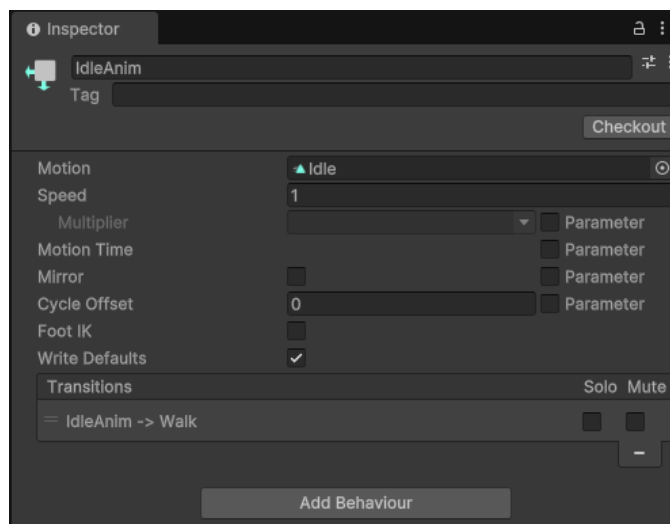


Рис 3.10 Стан IdleAnim

Оскільки ми ще не налаштували камеру при запуску гри, аби перевірити чи працює анімація, камера буде невідомо де, саме тому нам потрібно її перемістити в правильне місце, для цього потрібно її вибрати в полі Sample Scene. Я перевернув її по осі X на 90 градусів та поставив її позицію 0 70 0, для того щоб було видно головного героя (див. рис. 3.11.). Важливо! Якщо ви зміните щось під час запуску гри, зміни не збережуться.

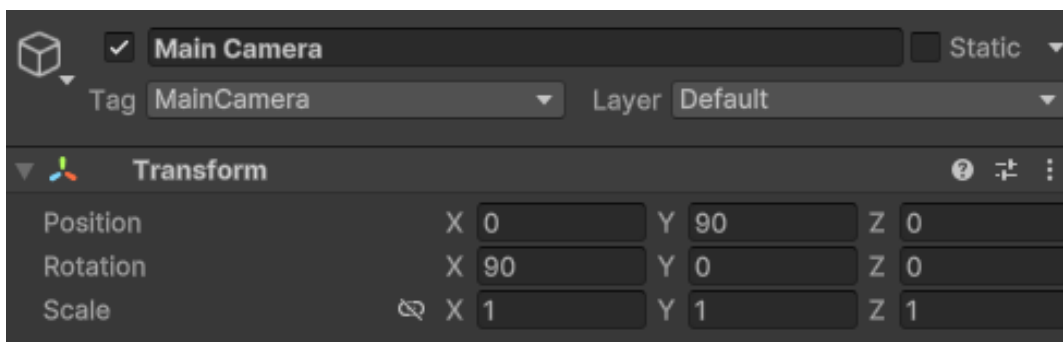


Рис 3.11 Розташування камери

Для керування головним героєм, потрібно додати Character Controller в меню інспектора та створити новий C# скрипт, свій скрипт я назвав PlayerMovement (див. рис. 3.12.).

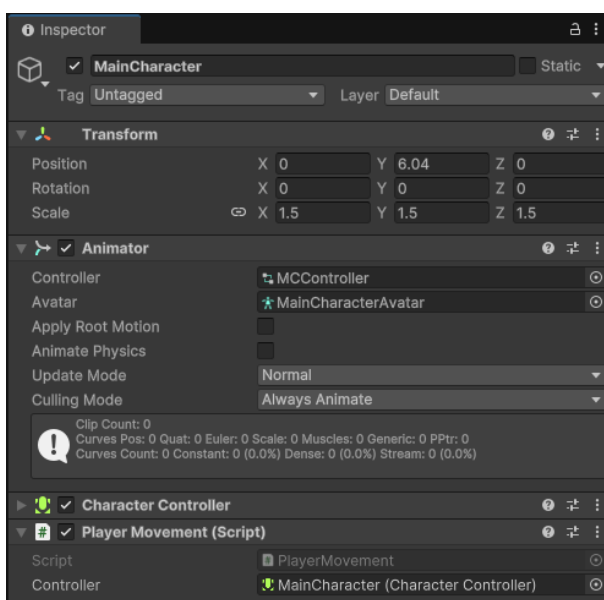


Рис 3.12. Створені Player Controller та скрипт Player Movement

Для зберігання скриптів бажано створити папку Scripts, у папці Assets, куди ми і збережемо наш новий C# скрипт. Далі потрібно його відкрити за допомогою зручної для вас IDE, для мене це Visual Studio. У цьому скрипті нам потрібно:

1. додати character controller;
2. зчитати ввід із клавіатури;
3. рухати персонажа в напрямку натиснутої кнопки;

4. відтворити анімацію руху персонажа;
5. пересунути камеру за головним героєм;
6. повернути персонажа лицем до курсору.

Для виконання пунктів 1-2 потрібно написати такий код:

```
public CharacterController Controller;
float h = Input.GetAxisRaw("Horizontal");
float v = Input.GetAxisRaw("Vertical");
```

Для створення руху головного героя потрібно модифікувати наш скрипт в такий манері:

```
float downmove = 0;
if(transform.position.y < 4.1){
    downmove = -Controller.transform.position.y + 4.1f; }
Vector3 Move = new Vector3(h, downmove, v) * Player.MovementSpeed;
Controller.Move(Move);
```

Цей код дозволить пересувати нашого головного героя відповідно його швидкості - значення яке ми будемо зберігати в статичному класі Player.

Для виконання пункту 4, потрібно відкрити раніше створений контролер анімацій та створити у ньому новий стан для ходьби, я назвав його Walk та додав перехід між анімацією Idle та Walk (див. рис. 3.13.). Також потрібно створити значення float з назвою MS та додати її як множник до цієї анімації, це потрібно для того щоб швидкість анімації відповідала швидкості пересування головного героя.

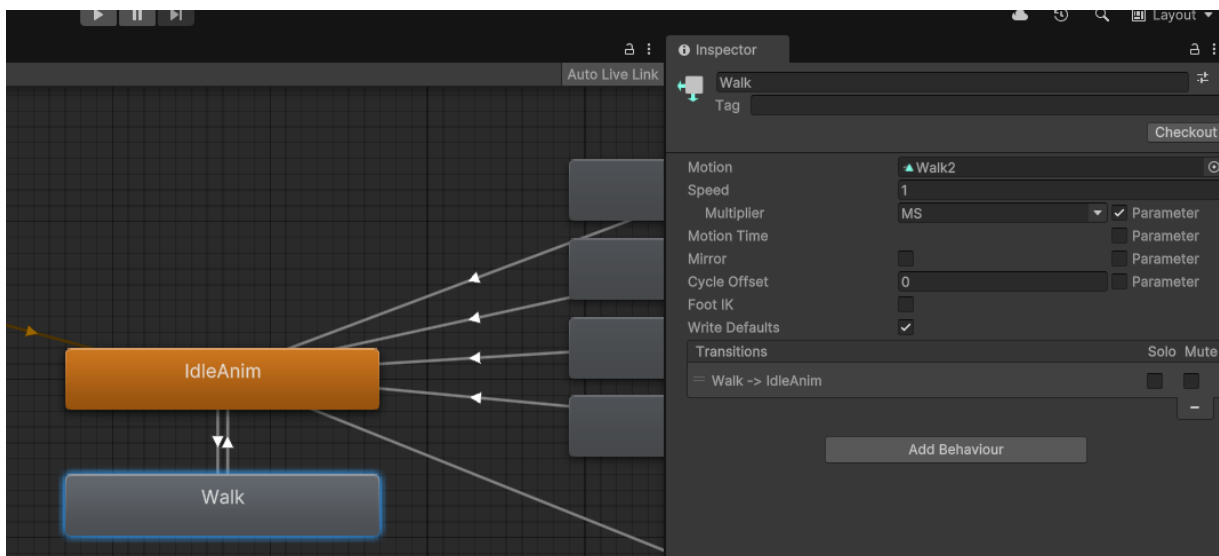


Рис 3.13. Стан Walk у вікні контролера анімацій

Для того аби відтворювати цю анімацію потрібно створити код із таким функціоналом:

```
if((h != 0 || v != 0) && (Player.State == "IdleAnim" || Player.State == "HoldingBomb")){
    Player.Animator.SetFloat("MS", 1);}
else{
    Player.Animator.SetFloat("MS", 0);}
```

Цей код дозволить запускати анімацію Idle, коли значення руху дорівнюють 0.

Аби камера пересувалась разом із персонажем, всередині нашого скрипту потрібно встановити стандартне положення камери та пересувати її відповідно до позиції головного героя:

```
Vector3 targetposition = transform.position + new Vector3(0, 70, 0);
Vector3 newPosition =
Vector3.MoveTowards(Player.MainCamera.transform.position, targetposition, 500f *
Time.deltaTime);
Player.MainCamera.transform.position = newPosition;
```

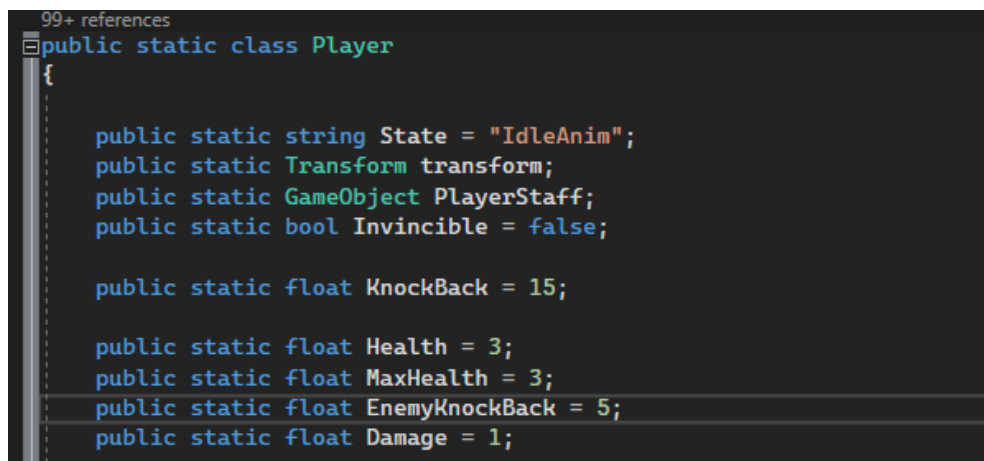
Щоб повернути нашого персонажа лицем до курсора у пункті номер 6, потрібно використати функцію ScreenPointToRay, яка допоможе визначити місцезнаходження курсора на даний момент та потрібно створити уявну площину, на якій ми будемо визначати місце курсора. Ми відправлятимемо

уявний промінь в те місце, де знаходиться курсор, та повертатимемо головного героя відповідно до цього значення. Ось такий код я написав для виконання даної задачі:

```
Plane playerplane = new Plane(Vector3.up, transform.position);
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
float hitdist;
if (playerplane.Raycast(ray, out hitdist))
{
    Vector3 targetpoint = ray.GetPoint(hitdist);
    Quaternion targetrotation = Quaternion.LookRotation(targetpoint -
transform.position);
    transform.rotation = (Quaternion.Slerp(transform.rotation,
targetrotation, Time.deltaTime * 50f));}
```

Це все, що потрібно, для того аби створити керування персонажем в Unity.

Для правильного, ефективного та зручного зберігання характеристик головного героя варто для зручності створити статичний клас в окремому файлі, я назвав його Player. У такий клас можна записувати всі можливі значення які потрібні головному герою, до прикладу, швидкість руху, шкода, аніматор, і так далі (див рис 3.14).



```
99+ references
public static class Player
{
    public static string State = "IdleAnim";
    public static Transform transform;
    public static GameObject PlayerStaff;
    public static bool Invincible = false;

    public static float KnockBack = 15;

    public static float Health = 3;
    public static float MaxHealth = 3;
    public static float EnemyKnockBack = 5;
    public static float Damage = 1;
```

Рис 3.14. Статичний клас Player

Також важливо додати гравцю можливість атакувати, вам потрібно створити новий скрипт PlayerAttack та додати його на нашого головного героя в меню інспектора. У самому скрипті потрібно очікувати ввід лівої кнопки



миші, яка стандартно називається Fire1 та після отримання даного вводу змінити стан головного героя на атаку, відтворити відповідну анімацію та створити кулю, яка буде розбиватись об об'єкти, такі як стіни або противники.

Для запуску потрібної нам анімації, варто додати її до аніматор контролера, дати їм відповідну назву та додати перехід після їх закінчення до стандартного стану (див. рис. 3.15).

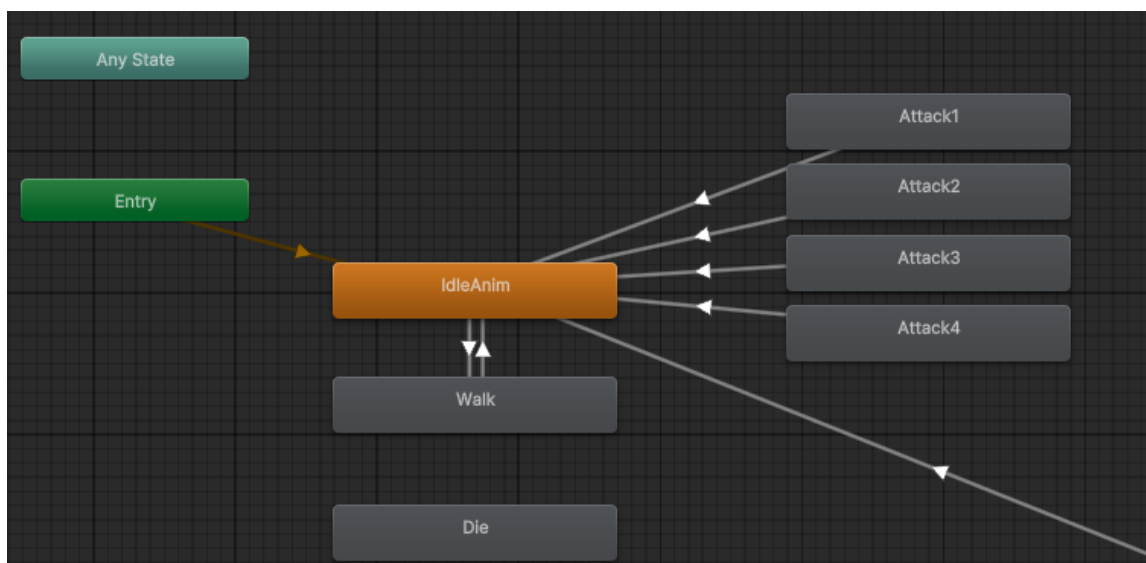


Рис 3.15 Контролер анімації з уже доданими анімаціями атаки

Для того аби зробити так, щоб персонаж вмів стріляти, потрібно створити об'єкт, який головний герой створить при натисканні лівої кнопки миші. Для досягнення цього, потрібно сторонньо створити модель кулі та її вибух, для цього можна використати будь-який редактор для створення 3D моделей, по типу Blender, у моєму випадку ось так виглядає куля [5] (див. рис. 3.16).

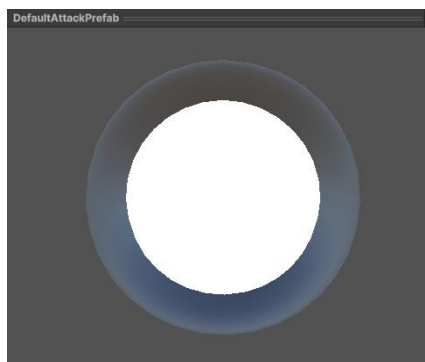


Рис. 3.16. 3D модель кулі

Для перевірки чи буде працювати наш скрипт та наші атаки, варто зробити стіну у нашій сцені, це доволі просто, натиснувши праву кнопку миші можна створити 3D об'єкт стіна та розтягнути її так, як потрібно (див. рис. 3.17.).

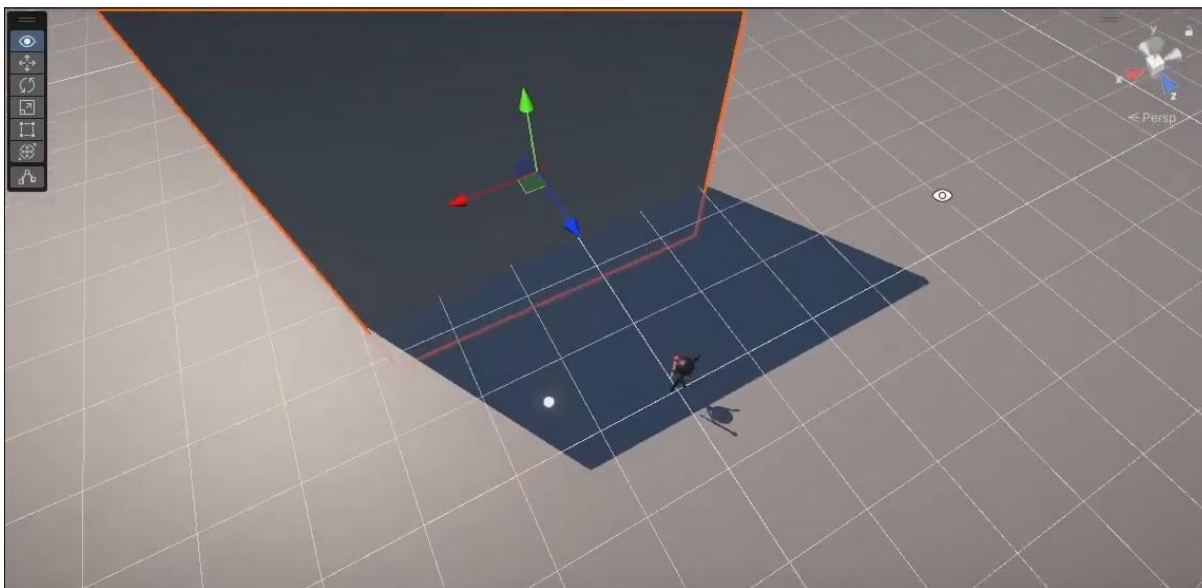


Рис. 3.17. Стіна для тестування скрипту

Також перед написанням скрипта варто додати наш префаб атаки у вільний слот в інспекторі (див. рис. 3.18.), які з'являться після додавання значень Attack та AttackExplosion у нашому статичному класі Player.

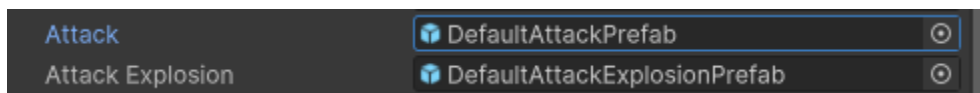


Рис. 3.18. Доданий префаб у відповідний слот в інспекторі

Далі, можна приступити до написання свого скрипта, який у мене має ось такий вигляд:

```
public class PlayerAttack : MonoBehaviour{
    public void AttackEnd(){
        Player.State = "IdleAnim"}
    public void Attack(){
        GameObject bullet = Instantiate(Player.Attack, transform.position +
transform.forward * 5, Player.Attack.transform.rotation);
        bullet.GetComponent<Rigidbody>().linearVelocity = transform.forward * 65;
        GameObject.Destroy(bullet, 4);}
    void Update(){
```

```

if (Input.GetButton("Fire1") && Player.State == "IdleAnim"){
    Player.State = "Attacking";
    string AttackNumber = Random.Range(1,5).ToString();
    Player.animator.Play("Attack" + AttackNumber);}

```

Якщо все було правильно підключено та написано, у вас повинен бути робочий скрипт, який дозволяє вашому персонажу атакувати, але кулі персонажа ще не будуть пропадати при ударі об стіну, для цього нам потрібно створити ще один скрипт, який я назвав Spell та прикріпити його в інспекторі до самої кулі. У цьому скрипті нам потрібно створювати AttackExplosion та видаляти кулю при ударі об об'єкти. Для цього існує дуже корисна функція в Unity, яка називається OnCollisionEnter. Цей скрипт наразі буде дуже коротким, але ми його будемо модифікувати в майбутньому, при створенні перших противників. Ось такий вигляд має скрипт Spell:

```

private void OnCollisionEnter(Collision collision){
    GameObject.Destroy(Instantiate(Player.AttackExplosion,
    transform.position, Quaternion.identity), 3);
    GameObject.Destroy(gameObject); }

```

При правильному виконанні, ваш персонаж після натискання лівої кнопки миші повинен створювати кулю, яка летить в напрямку погляду головного героя та повинна підриватись при ударі об створену вами стіну (див. рис. 3.19).

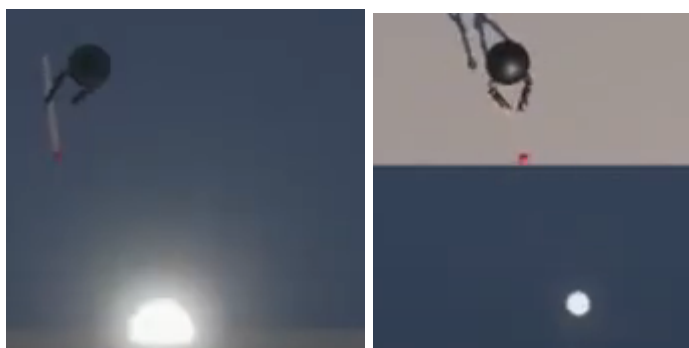


Рис. 3.19. Вибух кулі при ударі об стіну та сама куля при натисканні Fire1

### 3.3 Створення генерації карти, її взаємодії з головним героєм та відображення на екрані

Для створення карти потрібно спочатку зрозуміти алгоритм її створення, а саме береться поле 4X4 (це значення можна змінити), початкова кімната позначається як 0,0 та береться ліва сторона, з 50% шансом зліва створиться

кімната, якщо ж вона дійсно створилася, ми переходимо в цю кімнату і проводимо ту ж саму дію по створенню кімнати, але тепер вже перевіряємо всі 4 сторони, а також, перевіряємо чи вже існує кімната поруч, оскільки кімнати не повинні існувати одна біля одразу біля іншої, і цей алгоритм потрібно продовжити доти, доки не закінчатся сторони та створиться певна кількість кімнат або доки уся карта не заповниться повністю. Також, потрібно буде створити додатковий алгоритм для генерації, так званих, особливих кімнат, до прикладу: Секретна кімната, Кімната з предметом та Бос кімната. Для реалізації такої генерації карти нам потрібно створити інтерфейс користувача з певними іконками (див. рис. 3.20.) та розташуванням.

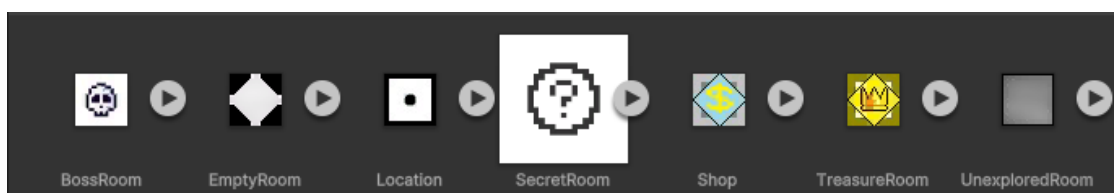


Рис. 3.20 Іконки, які будуть використані для карти

Для створення візуальної частини карти потрібно перевести редактор Unity в 2D режим натиснувши відповідну клавішу зверху панелі. Далі потрібно створити UI Panel, яка автоматично створиться всередині Canvas. Також, потрібно встановити, щоб розширення Canvas розширювалось або зменшувалось разом із екраном. Щоб досягнути цього результату, всередині Canvas потрібно встановити UI Scale mode >> Scale With Screen Size (див. рис. 3.21.).

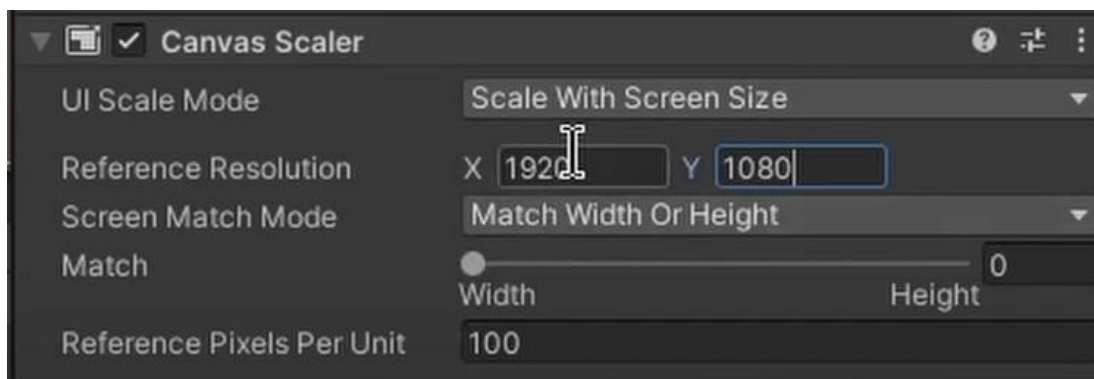


Рис. 3.21. Canvas Scaler

Далі потрібно перенести нашу створену панель в правий верхній кут та встановити її розміри, в моєму випадку це буде 500 на 500 пікселів:

```
public static float Height = 500;
public static float Width = 500;
```

Це і буде нашим полем для візуалізації карти, згодом ми вимкнемо фон для кращого візуального сприйняття, але наразі для того аби було видно прогрес ми залишимо його включеним. Далі потрібно створити UI Image всередині нашої панелі та встановити її розмір 50 на 50, або 1\10 розміру панелі, такого розміру буде візуалізуватись кімната на карті.

Для створення генерації карти потрібно, для початку, створити новий статичний скрипт в якому ми будемо зберігати усі дані про генерацію рівнів, так само як ми це робили із персонажем в пункті 3.2. Свій скрипт я назвав Level, такий вигляд він буде мати в коді public static class Level. Для початку достатньо зберігати лише іконки, розмір карти, її обмеження, тощо:

```
public static float Height = 500;
public static float Width = 500;
public static float Scale = 1f;
public static float IconScale = .07f;
public static float padding = .005f;
public static float RoomGenerationChance = .5f;
public static int RoomLimit = 4;
public static Sprite TreasureRoomIcon;
public static Sprite BossRoomIcon;
public static Sprite ShopRoomIcon;
public static Sprite UnexploredRoomIcon;
public static Sprite DefaultRoomIcon;
public static Sprite CurrentRoomIcon;
public static Sprite SecretRoomIcon;
```

Далі, власне, потрібно створити скрипт в якому і буде відбуватися уся магія генерації карти, свій скрипт я назвав GenerateLevel. Створений скрипт потрібно додати до попередньо створеного Canvas (див. рис. 3.22).

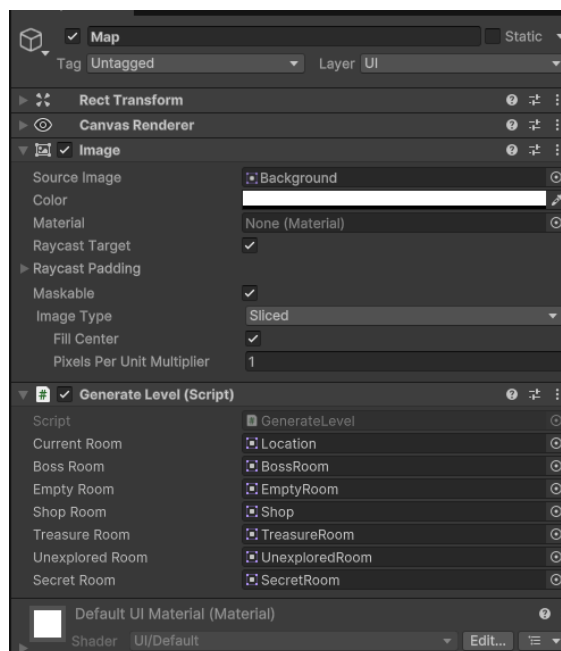


Рис. 3.22 Доданий скрипт до Canvas

Після того як ми його додали до нашого Canvas, можна приступити до написання самого скрипта. Перше що потрібно зробити це задекларувати наші іконки для кімнат в самому скрипті та додати їх до нього:

```
public Sprite CurrentRoom;
public Sprite BossRoom;
public Sprite EmptyRoom;
public Sprite ShopRoom;
public Sprite TreasureRoom;
public Sprite UnexploredRoom;
public Sprite SecretRoom;
```

Для того аби відображувати ці іконки на карті потрібно створити метод, який буде відповідальний за це. Цей метод бере номер кімнати, яка буде створена і, відповідно, створює ігровий об'єкт (картинку), який в даний момент потрібен в правильному місці та додає цю кімнату в список кімнат. Цей метод на даному етапі має такий вигляд:

```
void DrawRoomOnMap(Room R) {
    string TileName = "MapTile";
    if (R.RoomNumber == 1) TileName = "BossRoomTile";
    if (R.RoomNumber == 2) TileName = "ShopRoomTile";
    if (R.RoomNumber == 3) TileName = "TreasureRoomTile";
    GameObject MapTile = new GameObject(TileName);
    Image RoomImage = MapTile.AddComponent<Image>();
```

```

RoomImage.sprite = R.RoomSprite;
R.RoomImage = RoomImage;
RectTransform rectTransform = RoomImage.GetComponent<RectTransform>();
rectTransform.sizeDelta = new Vector2(Level.Height, Level.Width) *
Level.IconScale;
rectTransform.position = R.Location * (Level.IconScale * Level.Height *
Level.Scale + (Level.padding * Level.Height * Level.Scale));
RoomImage.transform.SetParent(transform, false);
Level.Rooms.Add(R);
Debug.Log("Drawing Room: " + R.RoomNumber + " at location: " + R.Location);}

```

Після написання цього методу та використання його в методі Start у вас при запуску повинна показатись ваша іконка стартової кімнати, або ту, що ви створили (див. рис. 3.23).

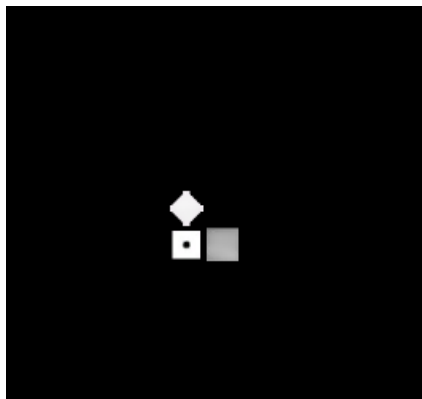


Рис. 3.23 Створена іконка на карті

Далі потрібно працювати над створенням кімнат у всіх напрямках, для початку потрібно встановити змінну для випадково шансу генерації карти, в моєму випадку це:

```
public static float RoomGenerationChance = .5f;
```

Згодом, потрібно створити перевірку чи випадкове число більше за наше число випадкової генерації та створити кімнату в даному напрямку, таких перевірок буде 4 для кожної з 4 сторін. Ось так це повинно виглядати:

```

//left
if(Random.value > Level.RoomGenerationChance){
    Room newRoom = new Room();
    newRoom.Location = new Vector2(-1, 0);
    newRoom.RoomSprite = Level.DefaultRoomIcon;
    newRoom.RoomNumber = RandomRoomNumber();
    if (!CheckIfExists(newRoom.Location)){

```

```

        if (!CheckIfRoomAroundExists(newRoom.Location, "Right"))
            Generate(newRoom);}}
//right
if (Random.value > Level.RoomGenerationChance) {
    Room newRoom = new Room();
    newRoom.Location = new Vector2(1, 0);
    newRoom.RoomSprite = Level.DefaultRoomIcon;
    newRoom.RoomNumber = RandomRoomNumber();
    if (!CheckIfExists(newRoom.Location)){
        if (!CheckIfRoomAroundExists(newRoom.Location, "Left"))
            Generate(newRoom); }}
//up
if (Random.value > Level.RoomGenerationChance){
    Room newRoom = new Room();
    newRoom.Location = new Vector2(0, 1);
    newRoom.RoomSprite = Level.DefaultRoomIcon;
    newRoom.RoomNumber = RandomRoomNumber();
    if (!CheckIfExists(newRoom.Location)){
        if (!CheckIfRoomAroundExists(newRoom.Location, "Down"))
            Generate(newRoom);
    }}
//down
if (Random.value > Level.RoomGenerationChance){
    Room newRoom = new Room();
    newRoom.Location = new Vector2(0, -1);
    newRoom.RoomSprite = Level.DefaultRoomIcon;
    newRoom.RoomNumber = RandomRoomNumber();
    if (!CheckIfExists(newRoom.Location))
        {if (!CheckIfRoomAroundExists(newRoom.Location, "Up"))
            Generate(newRoom);}}

```

Для того аби цей метод працював, також, потрібно створити два методи для перевірки, 1 для перевірки чи така кімната вже існує, щоб не створити її заново та другий метод для того аби перевірити чи уже є дотична до неї кімната. Ці методи я назвав CheckIfRoomAroundExists та CheckIfExists :

```

bool CheckIfExists(Vector2 v){
    return (Level.Rooms.Exists(x => x.Location == v));}
bool CheckIfRoomAroundExists(Vector2 v, string direction){
    switch(direction){
        case "Right":{
            //checking Down,up,left
            if(Level.Rooms.Exists(x => x.Location == new Vector2(v.x -1,v.y)) ||
            Level.Rooms.Exists(x => x.Location == new Vector2(v.x, v.y -1)) ||
            Level.Rooms.Exists(x => x.Location == new Vector2(v.x, v.y +1)))
                return true;
            break;}
        case "Left":{
            //checking Down,Right,Up

```



```

        if (Level.Rooms.Exists(x => x.Location == new Vector2(v.x + 1, v.y)) ||
            Level.Rooms.Exists(x => x.Location == new Vector2(v.x, v.y - 1)) ||
            Level.Rooms.Exists(x => x.Location == new Vector2(v.x, v.y + 1)))
            return true;
        break;}
    case "Up":{
        //checking Right,down,left
        if (Level.Rooms.Exists(x => x.Location == new Vector2(v.x - 1, v.y)) ||
            Level.Rooms.Exists(x => x.Location == new Vector2(v.x + 1, v.y)) ||
            Level.Rooms.Exists(x => x.Location == new Vector2(v.x, v.y - 1)))
            return true;
        break;}
    case "Down":{
        //checking right,up,left
        if (Level.Rooms.Exists(x => x.Location == new Vector2(v.x - 1, v.y)) ||
            Level.Rooms.Exists(x => x.Location == new Vector2(v.x + 1, v.y)) ||
            Level.Rooms.Exists(x => x.Location == new Vector2(v.x, v.y + 1)))
            return true;
        break;} }

```

Проте, ці методи створять лише 1 кімнату, а нам потрібно наповнити ними всю карту, для того щоб це виправити потрібно створити метод для генерації кімнат, я його назвав Generate. Цей метод буде постійно створювати нові кімнати доти, доки не стикнеться з обмеженням по кімнатах, з іншою кімнатою збоку або не випаде достатнє число для створення цієї ж кімнати. Метод Generate складається з чотирьох блоків для кожної зі сторін, так як і наш попередній метод. Такий вигляд буде мати один із 4 блоків:

```

//left
if (Random.value > Level.RoomGenerationChance)
{
    Room newRoom = new Room();
    newRoom.Location = new Vector2(-1, 0) + room.Location;
    newRoom.RoomSprite = Level.DefaultRoomIcon;
    newRoom.RoomNumber = RandomRoomNumber();

    if (!CheckIfExists(newRoom.Location))
    {
        if(!CheckIfRoomAroundExists(newRoom.Location, "Right"))
        {
            if (Mathf.Abs(newRoom.Location.x) < Level.RoomLimit &&
                Mathf.Abs(newRoom.Location.y) < Level.RoomLimit)
                Generate(newRoom);
        }
    }
}

```

Цей блок потрібно аналогічно повторити для правої, нижньої та верхньої сторін. Також, для того щоб цей метод правильно працював, нам потрібно

створити новий клас Room в якому ми будемо зберігати весь список кімнат та їх значення:

```
public class Room{
    public static List<Room> Rooms = new List<Room>();
    public int RoomNumber = 6;
    public Vector2 Location;
    public Image RoomImage;
    public Sprite RoomSprite;
    public bool Revealed;
    public bool Explored;
    public bool Cleared = true;
    public string RoomType = "Rooms";}
```

При дотриманні всіх вимог та правильно написаному коді до даного кроку, повинно вийти щось на кшталт цього (див. рис. 3.24.), проте у вас не повинно бути особливих іконок.



Рис. 3.24. Результат генерації карти без обмеження

Для того аби перевірити генерацію візуальної частини карти я додав метод для повторної генерації карти при натисканні кнопки О:

```
if(Input.GetKey(KeyCode.P) && !regen)
{Regenerate();}
void Regenerate() {
    regen = true;
    oopsie = 0;
    Level.Rooms.Clear();
    Invoke(nameof(StopRegen), 1);
    for (int i = transform.childCount - 1; i >= 0; i--)
    {
        Transform child = transform.GetChild(i);
        Destroy(child.gameObject);}
    Level.Rooms.Clear();
```

```
Start();}
```

Далі потрібно створити першу особливу кімнату, а саме бос кімнату. Для цього потрібно взяти найдальшу кімнату зі списку кімнат, це можна визначити додавши X та Y кімнат без знаку та створити на цьому місці нову кімнату з певною іконкою. Щоб виконати таку операцію потрібно створити новий метод, у моєму разі це GenerateBossRoom, також потрібно перевірити чи не конфліктує бос кімната із ніякою іншою кімнатою, тому ми додамо до цього методу наші вище згадані перевірки:

```
private void GenerateBossRoom(){
    float Max = 0;
    Vector2 FurthestRoom = Vector2.zero;
    foreach(Room R in Level.Rooms){
        if (Mathf.Abs(R.Location.x) + Mathf.Abs(R.Location.y) >= Max){
            Max = Mathf.Abs(R.Location.x) + Mathf.Abs(R.Location.y);
            FurthestRoom = R.Location;} }
    Room BossRoom = new Room();
    BossRoom.RoomSprite = Level.BossRoomIcon;
    BossRoom.RoomNumber = 1;
    //Left
    if (!CheckIfExists(FurthestRoom + new Vector2(-1, 0))){
        if (!CheckIfRoomAroundExists(FurthestRoom + new Vector2(-1, 0), "Right")){
            BossRoom.Location = FurthestRoom + new Vector2(-1, 0);}}
    //Right
    else if (!CheckIfExists(FurthestRoom + new Vector2(1, 0))){
        if (!CheckIfRoomAroundExists(FurthestRoom + new Vector2(1, 0), "Left")){
            BossRoom.Location = FurthestRoom + new Vector2(1, 0);} }
    //Up
    else if (!CheckIfExists(FurthestRoom + new Vector2(0, 1))){
        if (!CheckIfRoomAroundExists(FurthestRoom + new Vector2(0, 1), "Down")){
            BossRoom.Location = FurthestRoom + new Vector2(0, 1);} }
    //Down
    else if (!CheckIfExists(FurthestRoom + new Vector2(0, -1))){
        if (!CheckIfRoomAroundExists(FurthestRoom + new Vector2(0, -1), "Up")){
            BossRoom.Location = FurthestRoom + new Vector2(0, -1);}}
    DrawRoomOnMap(BossRoom);}
```

За умови правильного виконання цього алгоритму, у вас повинна створюватись бос кімната в найдальшій точці карти, ще й з правильною іконкою, яку ми встановили викликавши BossRoom.RoomSprite = Level.BossRoomIcon;(див. рис. 3.25)

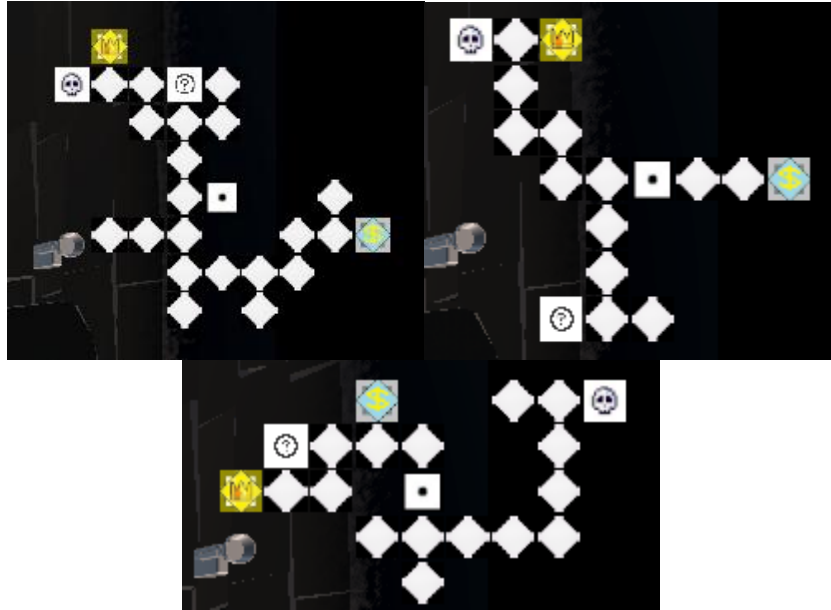


Рис. 3.25 Різні випадки створення бос кімнати

Далі по списку в нас генерація всіх інших особливих кімнат, а саме: магазин, предметна кімната та секретна кімната. Для створення таких кімнат на карті потрібно взяти зі списку випадкову кімнату, яка немає дотичних до неї кімнат та перетворити її на кімнату з відповідним номером та картинкою. Єдиним виключенням з правил буде секретна кімната, яка може знаходитись будь-де. Для того щоб дістати випадковий номер кімнати потрібно спочатку отримати випадкове число кімнати, для цього був написаний такий метод, який повертає число від 6 до кількості створених префабів кімнат, які ми ще пізніше зробимо:

```
int RandomRoomNumber(){
    return Random.Range(6, GameObject.Find("Rooms").transform.childCount);}
```

Також, для створення спеціальної кімнати, потрібно написати метод, який буде брати весь список кімнат, які у нас є, та перетасує їх, для цього я написав такий метод і назвав його `ShuffleList`. Цей метод є оригінально написаний Рональдом Фішером та Френком Єйтсом і саме тому називається тасування Фішера-Єйтса:

```
void ShuffleList<T>( List<T> list){
    int n = list.Count;
    System.Random rng = new System.Random();
    while (n > 1) {
        n--;
        int k = rng.Next(n + 1);
```

```

T value = list[k];
list[k] = list[n];
list[n] = value;}}

```

Після того як ми отримаємо список випадкових кімнат, ми можемо приступити до написання методу генерації спеціальних кімнат. Для початку, потрібно створити копію нашого списку та заповнити його нашим випадковим списком і зробити уже використану нами перевірку на наявність сусідніх кімнат. Окрім цього, потрібно створити перевірку, адже якщо номер числа більший ніж 6, то не потрібно створювати цю кімнату, як особливу. Після пройденої перевірки потрібно намалювати відповідну кімнату на карті з відповідною іконкою. Цей метод має ось такий вигляд:

```

private bool GenerateSpecialRoom(Sprite MapIcon, int RoomNumber ) {
    List<Room> ShuffledList = new List<Room>(Level.Rooms);
    ShuffleList(ShuffledList);
    Room SpecialRoom = new Room();
    SpecialRoom.RoomSprite = MapIcon;
    SpecialRoom.RoomNumber = RoomNumber;
    bool FoundAvailibleLocation = false;
    foreach (Room R in ShuffledList){
        Vector2 SpecialRoomLocation = R.Location;
        if (R.RoomNumber < 6) continue;
        //Left
        if (!CheckIfExists(SpecialRoomLocation + new Vector2(-1, 0))){
            if (!CheckIfRoomAroundExists(SpecialRoomLocation + new Vector2(-1, 0), "Right")){
                SpecialRoom.Location = SpecialRoomLocation + new Vector2(-1, 0);
                FoundAvailibleLocation = true;}}
        //Right
        else if (!CheckIfExists(SpecialRoomLocation + new Vector2(1, 0))){
            if (!CheckIfRoomAroundExists(SpecialRoomLocation + new Vector2(1, 0), "Left")){
                SpecialRoom.Location = SpecialRoomLocation + new Vector2(1, 0);
                FoundAvailibleLocation = true;}}
        //Up
        else if (!CheckIfExists(SpecialRoomLocation + new Vector2(0, 1))){
            if (!CheckIfRoomAroundExists(SpecialRoomLocation + new Vector2(0, 1), "Down")){
                SpecialRoom.Location = SpecialRoomLocation + new Vector2(0, 1);
                FoundAvailibleLocation = true;}}
        //Down
        else if (!CheckIfExists(SpecialRoomLocation + new Vector2(0, -1))){
            if (!CheckIfRoomAroundExists(SpecialRoomLocation + new Vector2(0, -1), "Up")){
                SpecialRoom.Location = SpecialRoomLocation + new Vector2(0, -1);
                FoundAvailibleLocation = true;
            }
        }
        if (FoundAvailibleLocation){
            DrawRoomOnMap(SpecialRoom);
            return true;} }
    return false;}

```

Опісля потрібно використати створений нами метод в методі Start, проте на карті завжди повинна бути 1 магазин, 1 кімната з предметом, хоча б 1 секретна кімната та кімната з босом, саме тому нам потрібно додати таку перевірку в методі Start. Оскільки наш метод GenerateSpecialRoom повертає bool, то ми можемо використати це для нашої перевірки. Цю перевірку я реалізував ось так:

```
bool treasure = GenerateSpecialRoom(Level.TreasureRoomIcon, 3);
bool shop = GenerateSpecialRoom(Level.ShopRoomIcon, 2);
bool secret = GenerateSeretRoom();

if (!treasure || !shop || !secret)
{
    if (matrixes > 15) return;
    Regenerate();
}
```

Таким чином, якщо не буде створено хоча б якоїсь 1 кімнати з перелічених, карта перествориться і буде перестворюватися, доки не буде існувати усіх потрібних нам кімнат. Тепер ваша карта повинна виглядати повноцінною (див. рис. 3.26), хоча на разі лише візуально.

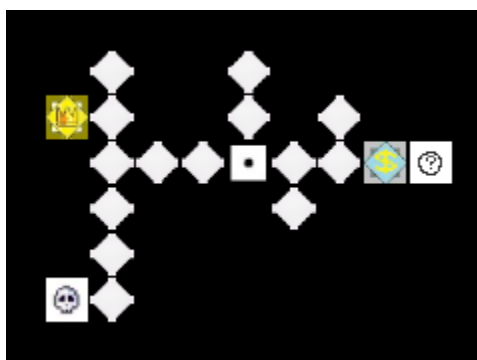


Рис. 3.26 Повноцінна карта

Далі потрібно створити самі кімнати та пересування головного героя між ними. Для досягнення такого результату варто створити прототип кімнати використовуючи стандартні фігури Unity або створити свою модель кімнати та дверей, для прикладу ось є кімната зроблена із простих фігур (див. рис. 3.27).

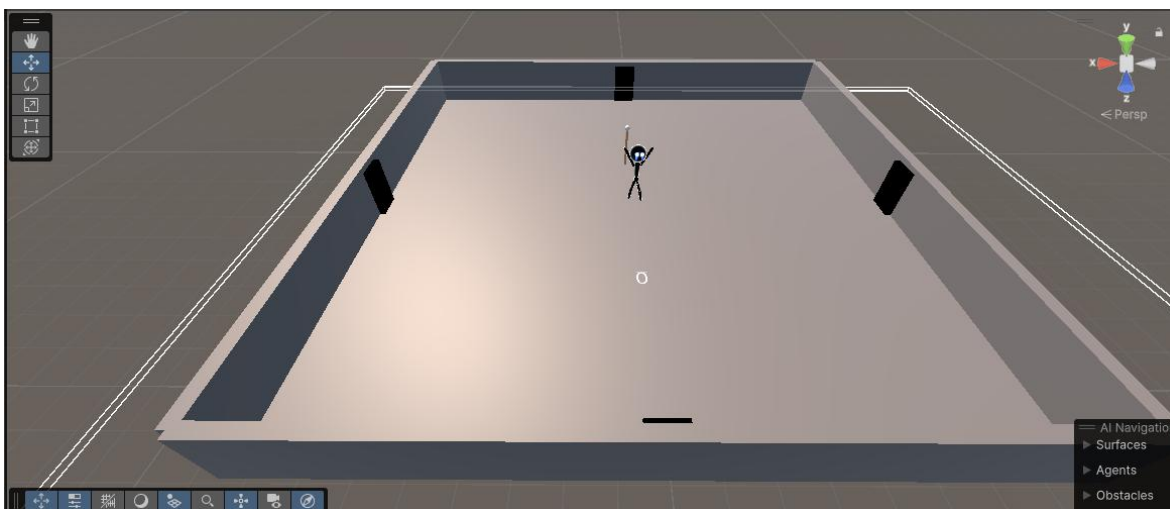


Рис. 3.27 Прототип кімнати створеної із 4 стін та 4 чорних кубів.

Аби полегшити керування цим всім, варто створити папку ігрового об'єкту в нашому редакторі сцени та назвати її 6, продублювати цю кімнату комбінацією Ctrl + D та перейменувати копії цифрами від 0 до 9 (див. рис. 3.28).

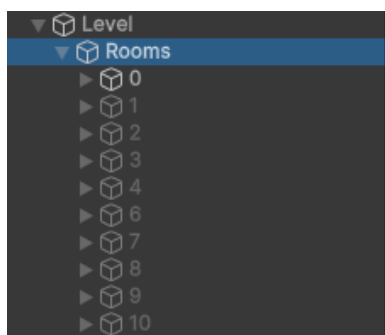


Рис. 3.28 Ієрархія кімнат

Також кожен із 4 чорних квадратів, які будуть дверима варто назвати TopDoor, Bottom Door і тд. І всередині ігрового об'єкта 6 створити ігровий об'єкт Doors та перемістити всі наші двері туди для читабельності. Далі можна приступити до створення скрипту переміщення персонажа в іншу кімнату, для цього потрібно створити новий C# скрипт та додати його до персонажа, свій скрипт я назвав ChangeRooms. У цьому скрипті потрібно перевіряти чи головний герой не торкається однієї із дверей, і якщо він їх торкається, переміщати його відповідно, для цього я використав OnCollisionHit :

```
private void OnCollisionHit(ColliderHit hit)
{
```

```

if (changeroomcd || hit.gameObject.layer == LayerMask.NameToLayer("Floor") ||
Player.CurrentRoom.Cleared != true)
{
    return;
}else{
    changeroomcd=true;
    Invoke(nameof(EndChangeRoomCD), Level.RoomChangeTime);}
if(hit.gameObject.name == "LeftDoor")
{
CheckDoor(new Vector2(-1, 0), "RightDoor", new Vector3(-RoomSpawnOffset, 0, 0));
}
else if (hit.gameObject.name == "RightDoor")
{
CheckDoor(new Vector2(1, 0), "LeftDoor", new Vector3(RoomSpawnOffset, 0, 0));
}
else if (hit.gameObject.name == "TopDoor")
{
CheckDoor(new Vector2(0, 1), "BottomDoor", new Vector3(0, 0, RoomSpawnOffset));
}
else if (hit.gameObject.name == "BottomDoor")
{
CheckDoor(new Vector2(0, -1), "TopDoor", new Vector3(0, 0, -RoomSpawnOffset));
}}

```

Цей скрипт, при ударі об одну із дверей, переміщатиме персонажа до інших дверей, де повинен бути персонаж, вимикати кімнату з якої ми вийшли та вмикати кімнату в якій ми повинні опинитись. Але цей скрипт не буде ще поки нікуди нас переміщати. Для цього ми створимо метод CheckDoor, який буде нас переміщати та перевіряти які двері повинні існувати, а які ні, оскільки не у всіх кімнатах є всі 4 проходи. Ось як виглядатиме цей метод:

```

void CheckDoor(Vector2 NewLocation, string Direction, Vector3 RoomOffset)
{
    //where is the character
    Vector2 Location = Player.CurrentRoom.Location;
    //Where is the character moving
    Location = Location + NewLocation;
    if (Level.Rooms.Exists(x => x.Location == Location))
    {
        Room R = Level.Rooms.First(x => x.Location == Location);
        //Disabling the room that charater was in
        Transform TT = GameObject.Find(Player.CurrentRoom.RoomType.ToString()).transform;
        TT.Find(Player.CurrentRoom.RoomNumber.ToString()).gameObject.SetActive(false);
        //Get the new room and set it active
        Transform T = GameObject.Find(R.RoomType.ToString()).transform;
        GameObject NewRoom = T.Find(R.RoomNumber.ToString()).gameObject;
        NewRoom.SetActive(true);
        //Move the character model to the correct door
        Player.Controller.enabled = false;
    }
}

```



```

transform.position =
NewRoom.transform.Find("Doors").transform.Find(Direction).position+RoomOffset;
    Player.Controller.enabled = true;
    ChangeRoomIcon(Player.CurrentRoom, R);
    Player.CurrentRoom = R;
    EnableDoors(R);
    Player.CurrentRoom.Explored = true;
    RevealRooms(R);
    RedrawRevealedRooms();

```

Також потрібен метод, який буде вмикати потрібні нам двері, а саме Enable Doors, в цьому випадку, ось цей метод:

```

void EnableDoors(Room R)
{
    Transform T = Rooms.Find(R.RoomNumber.ToString());
    Transform Doors = T.Find("Doors");
    for(int i = 0; i < Doors.childCount; i++)
    {
        Doors.GetChild(i).gameObject.SetActive(false);
    }
    //Check What doors should exist
    //Left
    {
        Vector2 NewPosition = R.Location + new Vector2(-1, 0);
        if (Level.Rooms.Exists(x => x.Location == NewPosition)){
            if (Level.Rooms.First( x => x.Location == NewPosition).RoomNumber == 4 &&
            !Xcheck(T))
                { if (Level.SecretRoomExploded){
                    GameObject GO = Doors.Find("LeftDoor").gameObject;
                    GameObject SecretDoor = Instantiate(Level.SecretRoomDoor, GO.transform.position,
                    Quaternion.Euler(0, -90, 0), Doors);
                    SecretDoor.name = "LeftDoor";
                }else {
                    GameObject GO = Doors.Find("LeftDoor").gameObject;
                    Instantiate(Level.XMark, GO.transform.position,Quaternion.Euler(0, -90, 0), T);
                }
            }
        }else{
            Doors.Find("LeftDoor").gameObject.SetActive(true);
        }
    }
    //Up
    {
        Vector2 NewPosition = R.Location + new Vector2(0, 1);
        if (Level.Rooms.Exists(x => x.Location == NewPosition))
        {
            if (Level.Rooms.First(x => x.Location == NewPosition).RoomNumber ==
            4 && !Xcheck(T))
                {if (Level.SecretRoomExploded){
                    GameObject GO = Doors.Find("TopDoor").gameObject;

```

```

GameObject SecretDoor = Instantiate(Level.SecretRoomDoor,
GO.transform.position,Quaternion.Euler(0, 0, 0), Doors);
    SecretDoor.name = "TopDoor";
    }else{
        GameObject GO = Doors.Find("TopDoor").gameObject;
        Instantiate(Level.XMark, GO.transform.position,Quaternion.Euler(0, 0, 0), T);
    }}
    else{
        Doors.Find("TopDoor").gameObject.SetActive(true);
    }}}
//Down
{
    Vector2 NewPosition = R.Location + new Vector2(0, -1);
    if (Level.Rooms.Exists(x => x.Location == NewPosition)){
        if (Level.Rooms.First(x => x.Location == NewPosition).RoomNumber ==
4 && !Xcheck(T))
            {if (Level.SecretRoomExploded){
                GameObject GO = Doors.Find("BottomDoor").gameObject;
                GameObject SecretDoor = Instantiate(Level.SecretRoomDoor, GO.transform.position,
Quaternion.Euler(0, 180, 0), Doors);
                SecretDoor.name = "BottomDoor";
            }else{
                GameObject GO = Doors.Find("BottomDoor").gameObject;
                Instantiate(Level.XMark, GO.transform.position,Quaternion.Euler(0, 180, 0), T);
            }
            }else{
                Doors.Find("BottomDoor").gameObject.SetActive(true);
            }}}
//Right
{
    Vector2 NewPosition = R.Location + new Vector2(1, 0);
    if (Level.Rooms.Exists(x => x.Location == NewPosition))
    {
        if (Level.Rooms.First(x => x.Location == NewPosition).RoomNumber ==
4 && !Xcheck(T))
            {if (Level.SecretRoomExploded)
                {
                    GameObject GO = Doors.Find("RightDoor").gameObject;
                    GameObject SecretDoor = Instantiate(Level.SecretRoomDoor, GO.transform.position,
Quaternion.Euler(0, 90, 0), Doors);
                    SecretDoor.name = "RightDoor";
                }else{
                    GameObject GO = Doors.Find("RightDoor").gameObject;
                    Instantiate(Level.XMark, GO.transform.position, Quaternion.Euler(0, 90, 0), T);
                }}
            }else{
                Doors.Find("RightDoor").gameObject.SetActive(true);
            }}}
}

```

Окрім цього потрібно створити метод для зміни іконки, який знаходиться в методі `CheckDoor`. Він буде брати кімнату, в якій є наш персонаж, та міняти її іконку:

```
void ChangeRoomIcon(Room CurrentRoom, Room NewRoom)
{
    CurrentRoom.RoomImage.sprite = previousImage;
    previousImage = NewRoom.RoomImage.sprite;
    NewRoom.RoomImage.sprite = Level.CurrentRoomIcon;}

```

Також аби цей скрипт правильно працював потрібно поставити його останнім у черзі запуску, це можна зробити в настройках проекту. Якщо все було створено та написано правильно, ми зможемо ходити з кімнати в кімнату та з'являтимуться лише потрібні нам двері (див. рис. 3.29).

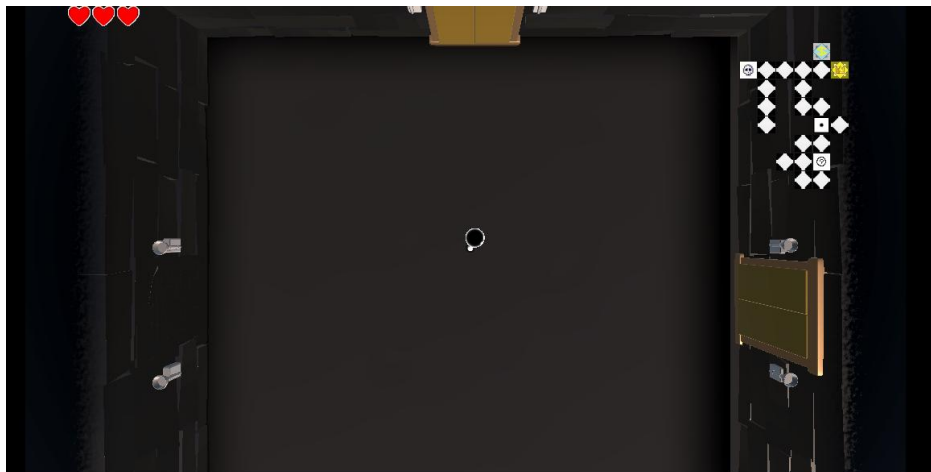


Рис. 3.29 Вимкнені непотрібні двері

Тепер у вас є повністю робоча карта. Після її створення потрібно зробити так, щоб гравець не бачив всю карту одразу, а лише кімнати, які розташовані біля тих, в яких він вже був. Для цього нам потрібно в статичному класі `Level` створити змінну `Revealed` та `Explored`, а в скрипті `ChangeRooms` створити метод `RevealRooms`, який ми будемо використовувати при пересуванні головного героя. Ось так має виглядати цей метод:

```
public static void RevealRooms(Room R){
    foreach (Room room in Level.Rooms){
        //left
        if(room.Location == R.Location + new Vector2(-1 ,0) && room.RoomNumber != 4){
            room.Revealed = true;
        }
    }
}

```

```

        //right
        if (room.Location == R.Location + new Vector2(1, 0) && room.RoomNumber != 4){
            room.Revealed = true;
        }
        //down
        if (room.Location == R.Location + new Vector2(0, -1) && room.RoomNumber != 4){
            room.Revealed = true;
        }
        //up
        if (room.Location == R.Location + new Vector2(0, 1) && room.RoomNumber != 4){
            room.Revealed = true;
        }
    }
}
}

```

Якщо все було виконано правильно у вас повинна бути така карта (див. рис. 3.30).

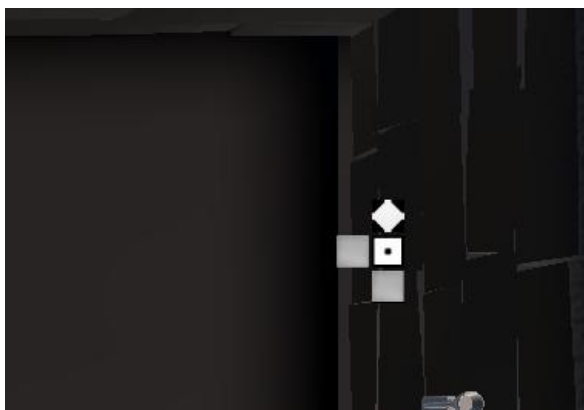


Рис. 3.30 Кімнати, в яких ще не був головний герой, та інші кімнати не показуються

Це все, що стосується генерації карти та її дослідження героєм. Далі нам потрібно розробити секретну кімнату, оскільки вона не підпорядковується стандартним правилам генерації та може знаходитися де завгодно. Також, для неї потрібно створити окремі двері, які потрібно підірвати, аби дістатись до них. Ми будемо створювати секретну кімнату, яка буде виглядати ось так (див. рис. 3.31).

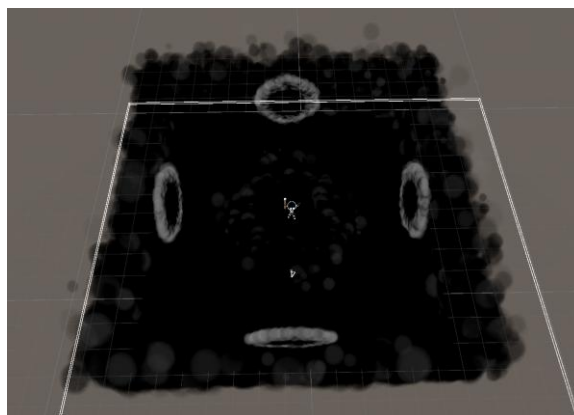


Рис. 3.31. Секретна кімната

Для того щоб створити таку кімнату потрібно створити візуальний ефект для стін та дверей та додати до них колізію. Цей ефект для стін повинен виглядати ось так (див. рис. 3.32):

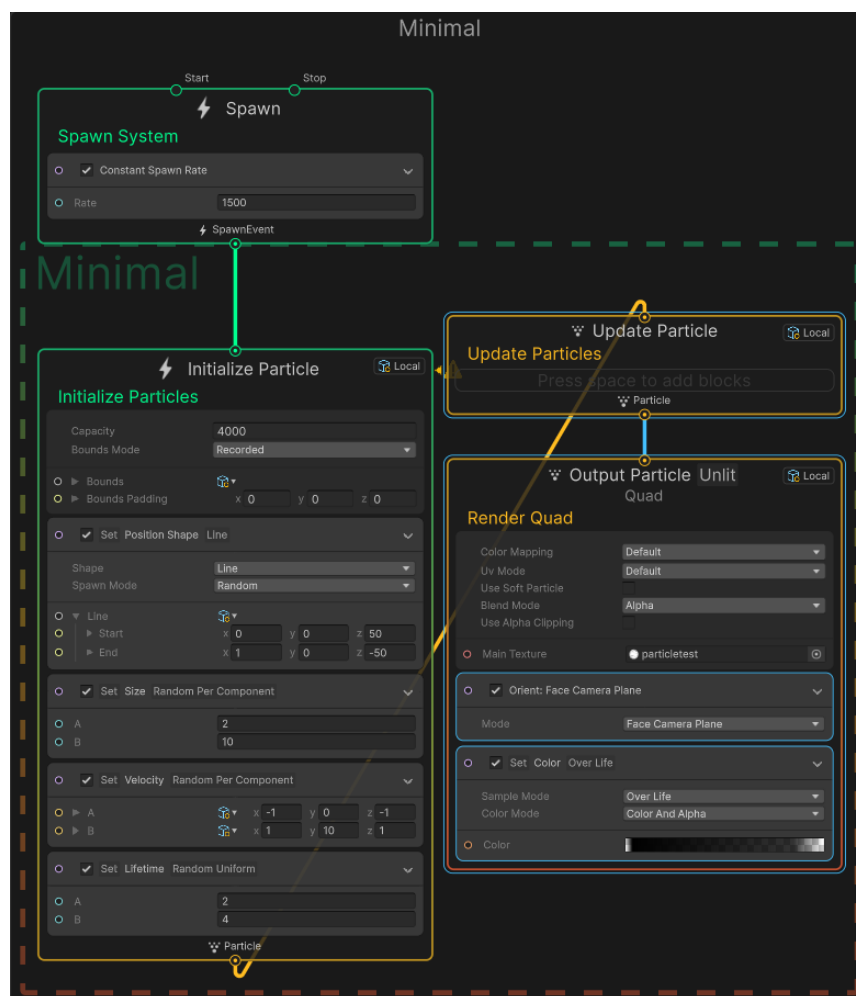


Рис. 3.32 Ефект для стін

Окрім цього, потрібно створити подібний ефект для дверей (див. рис. 3.33). До цих дверей потрібно ще додати VoxCollider та перейменувати їх відповідно, щоб вони працювали.

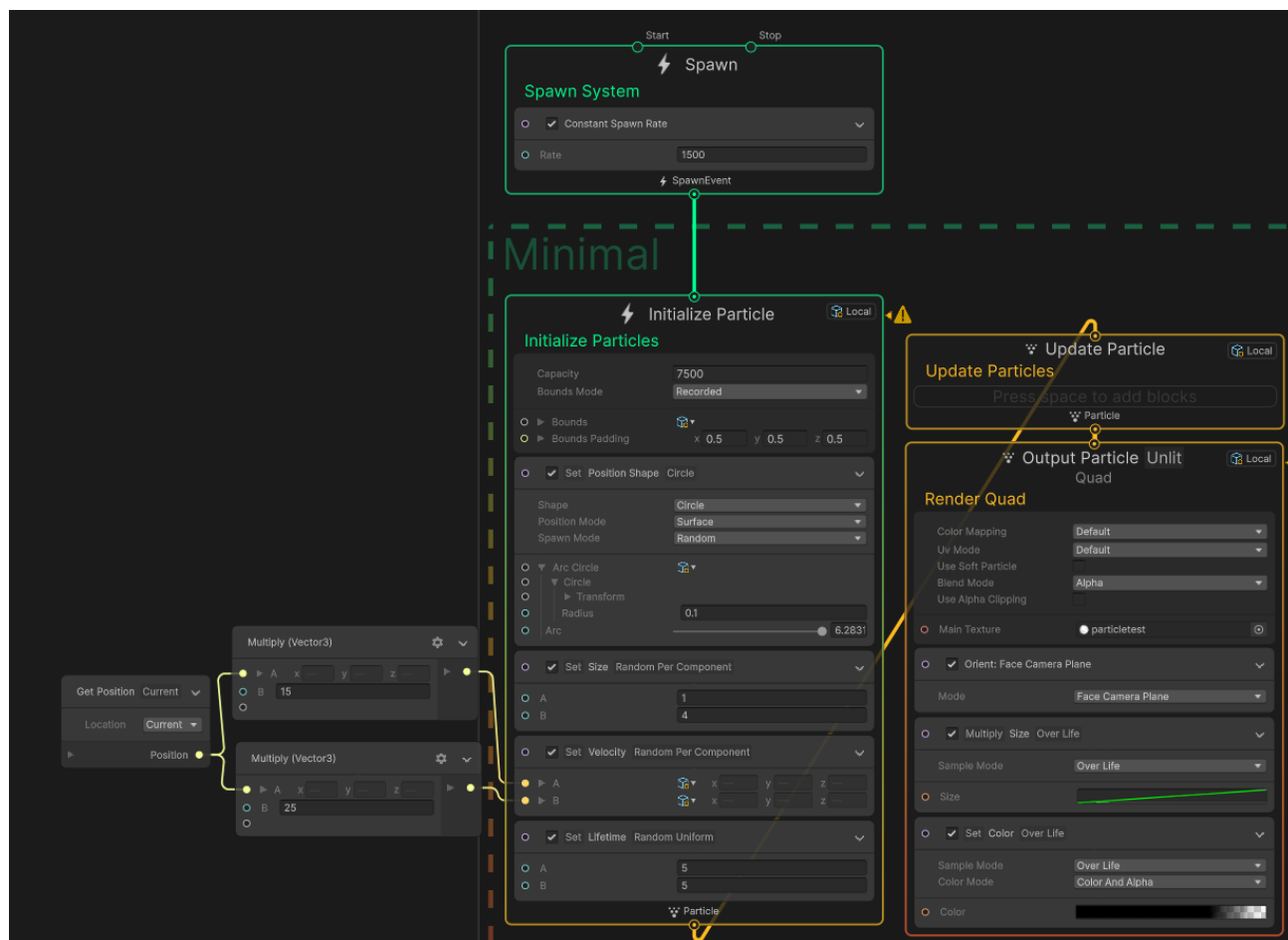


Рис. 3.33 Ефект дверей

Опісля, потрібно створити X, який буде відображатись на стіні, що потрібно підірвати. Для цього в мене є картинка X (див. рис. 3.34), яку я перетворив на декаль, також використовуючи візуальні ефекти (див. рис. 3.35).

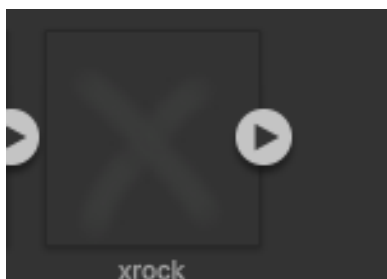


Рис. 3.34 Картинка X

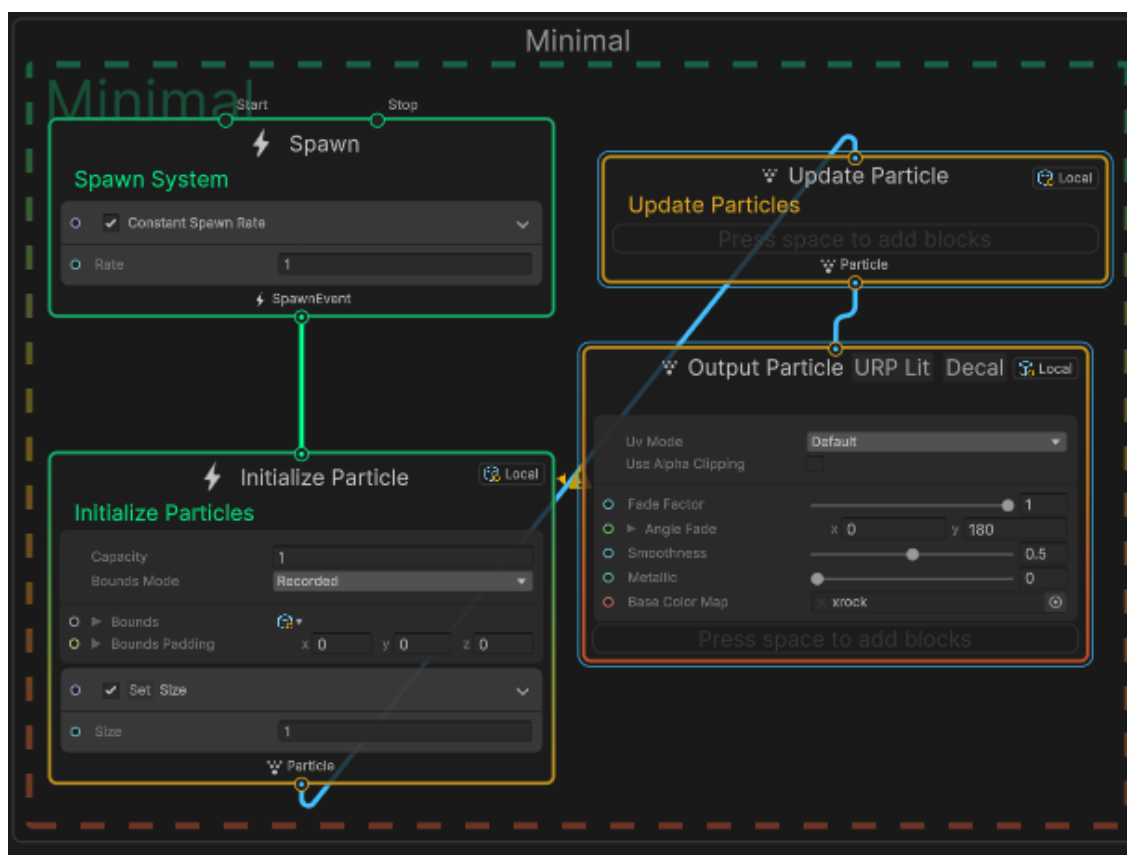


Рис. 3.35 Візуальний ефект для декалі

Згодом, потрібно зробити так, щоб цей X з'являвся в правильному місці та при попаданні, поки що кулі, створював вибух та портал, який переноситиме головного героя в секретну кімнату. Для реалізації цього потрібно створити спочатку візуальний ефект вибуху (див. рис. 3.36).

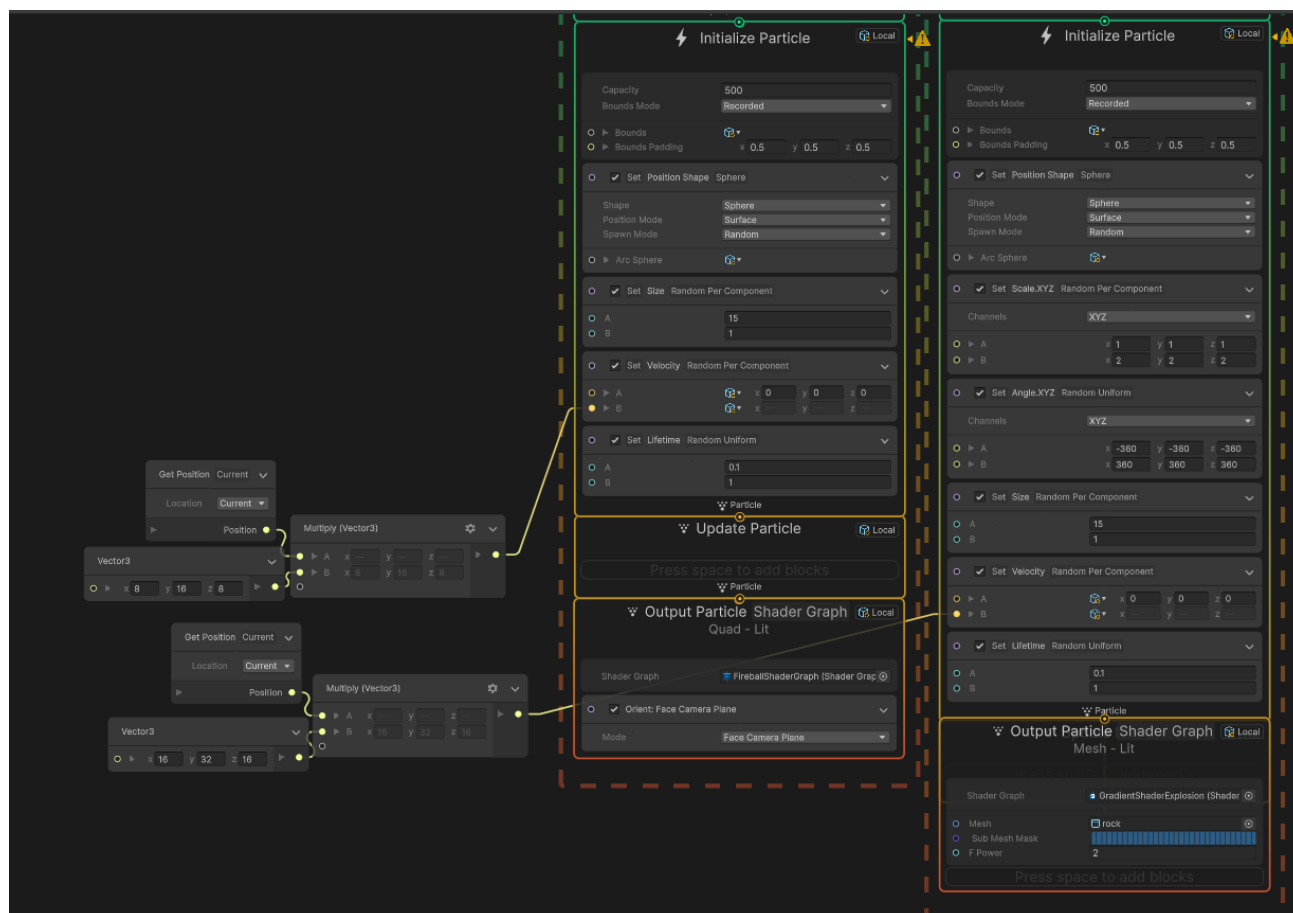


Рис. 3.36 Візуальний ефект вибуху

Далі, для створення секретної кімнати потрібно створити особливий метод `GenerateSecretRoom`, який відповідатиме за створення секретної кімнати згідно потрібних нам правил:

```
private bool GenerateSecretRoom(){
    List<Room> ShuffledList = new List<Room>(Level.Rooms);
    ShuffleList(ShuffledList);
    foreach (Room R in ShuffledList){
        if (Mathf.Abs(R.Location.x) > 3 || Mathf.Abs(R.Location.y) > 3 ||
R.Location == Vector2.zero)
        {continue;}
        Vector2[] directions = { new Vector2(-1, 0), new Vector2(1, 0), new
Vector2(0, 1), new Vector2(0, -1) };
        foreach (Vector2 direction in directions){
            Vector2 newLocation = R.Location + direction;
            //Check if room already exists at new location
            if(!Level.Rooms.Exists(x => x.Location == newLocation)){
                if (Mathf.Abs(newLocation.x) > 1 || Mathf.Abs(newLocation.y) > 1){
                    CreateNewRoom(newLocation);
                    return true;}}}}}
```



```

    return false;}
void CreateNewRoom(Vector2 location){
    Room SecretR = new Room{
        Location = location,
        RoomSprite = Level.SecretRoomIcon,
        Explored = false,
        Revealed = false,
        RoomNumber = 4};
    DrawRoomOnMap(SecretR); }

```

Цей скрипт бере випадкову кімнату, перевіряє чи є місце для секретної кімнати та створює її, якщо є вільне місце та воно не контактує з кімнатою 0,0. Далі, потрібно створити префаб із нашої картинки, це можна зробити перетягнувши наш X із сцени в папку. Також, оскільки це секретна кімната, нам потрібно не показувати її, якщо головний герой в сусідній кімнаті, тому до методу `RevealRooms` потрібно додати додаткову перевірку у вигляді : `&& room.RoomNumber != 4`. Окрім цього, в статичному класі `Player` потрібно створити змінні для зберігання нашого X, вибуху та дверей. Після цього можна створити додаткову перевірку на наявність секретної кімнати, і якщо кімната знаходиться там, створити на місці дверей нашу декаль X. Ця перевірка повинна мати такий вигляд:

```

//Left
{
    Vector2 NewPosition = R.Location + new Vector2(-1, 0);
    if (Level.Rooms.Exists(x => x.Location == NewPosition)){
if (Level.Rooms.First( x => x.Location == NewPosition).RoomNumber == 4 &&
!Xcheck(T))
    {
        if (Level.SecretRoomExploded){
            GameObject GO = Doors.Find("LeftDoor").gameObject;
GameObject SecretDoor = Instantiate(Level.SecretRoomDoor, GO.transform.position,
Quaternion.Euler(0, -90, 0), Doors);
            SecretDoor.name = "LeftDoor"; }
        else{
            GameObject GO = Doors.Find("LeftDoor").gameObject;
Instantiate(Level.XMark, GO.transform.position,Quaternion.Euler(0, -90, 0), T);
        }}else {
            Doors.Find("LeftDoor").gameObject.SetActive(true);
        }}
}

```

Ця перевірка дозволить нам створити X в тому місці, де повинні бути двері до секретної кімнати. Якщо все було виконано правильно, у вас повинен з'явитися X в тому місці, де є секретна кімната (див. рис. 3.37).



Рис. 3.37 X на стіні

Також, після того як ми підірвемо цей X, повинні утворитися нові двері, які ми створили раніше. Щоб отримати бажаного результату, в скрипті з бомбою, яку ми створимо пізніше, потрібно додати ось такі лінійки:

```
if (b.name.Contains("XDecal")){
    Level.SecretRoomExploded = true;
    GameObject.Destroy(Instantiate(Level.SecretRoomExplosion, b.transform.position,
    Quaternion.identity), 5f);
    GameObject SecretDoor = Instantiate(Level.SecretRoomDoor, b.transform.position,
    b.transform.rotation, T);
```

За правильного виконання роботи у вас повинна бути робоча секретна кімната, яку не видно поки її не відкрито і розташування якої видає X на стіні, а також, X створює вибух та двері на місці вибуху (див. рис. 3.38).

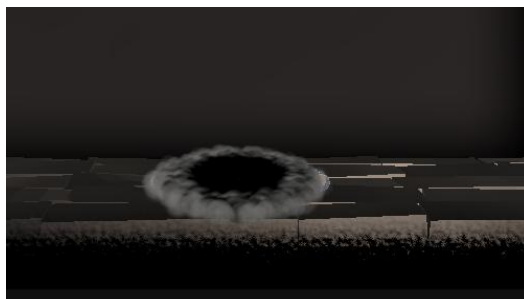


Рис. 3.38 Створені двері при попаданні бомби

Це все, що стосується генерації карти та секретної кімнати. У нас є створена карта візуально, ми можемо переміщатися з однієї кімнати в іншу, намалювати X на стіні, де є секретна кімната, а також цей X можна підірвати та створити на його місці двері в цю кімнату. Окрім цього, у нас є можливість досліджувати карту, яку ще не видно. Далі, нам потрібно буде лише створити саму бомбу, якою ми будемо підрипати X на стіні, противників та систему здоров'я.



використовувати для створення бомби в руках нашого головного героя. Після цього, нам потрібно створити 2 скрипти для роботи із цією моделлю та додати один із них до головного героя, а інший до бомби. Скрипт для головного героя буде BombScript, а для бомби – BombExplosionScript. У скрипті BombScript нам потрібно при натиснутій клавіші G змінювати стан головного героя на “тримає бомбу”, забороняти йому атакувати, виключати його палицю, змінювати анімацію на “тримає бомбу”. Також при стані “тримає бомбу”, якщо гравець натисне ліву кнопку миші, він повинен кинути цю бомбу вперед, а при повторному натиску G персонаж повинен поставити цю бомбу на землю і вона вибухне через 10 секунд. Для всього вищеперерахованого був написаний скрипт який виглядає, ось так:

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

public class BombScript : MonoBehaviour{
    public void EndBomb(){
        Player.State = "IdleAnim";
        Player.PlayerStaff.SetActive(true);}
    bool AbleToDropBomb = false;
    public void WaitForBombDrop(){
        AbleToDropBomb=true;}
    public IEnumerator BombCountdown(float seconds, GameObject bomb){
        bool bombdropped = false;
        float minspeed = 1;
        float maxspeed = 50;
        float TTime = 0;
        for (float i = 0; i<= seconds*100; i++){
            if (bomb == null){
                bombdropped = true;
                yield break;}
            Material M1 = bomb.GetComponent<MeshRenderer>().materials[0];
            Material M2 = bomb.GetComponent<MeshRenderer>().materials[1];
            TTime += Time.deltaTime;
            float angle = TTime * Mathf.Lerp(minspeed, maxspeed, i / seconds / 100);
            float SinValue = Mathf.Sin(angle);
            M1.SetFloat("_FlashSpeed", SinValue);
            M2.SetFloat("_FlashSpeed", SinValue);
            bomb.transform.Find("Timer").GetComponent<TextMeshPro>().fontSize = 24 + (SinValue * 5);

            if (i % 100 == 0){
                bomb.transform.Find("Timer").GetComponent<TextMeshPro>().text = (seconds - (i / 100)).ToString();}
            yield return new WaitForSeconds(0.01f);}
        GameObject.Destroy(Instantiate(Player.BombExplosion,
            bomb.transform.position, Quaternion.identity), 3);
```

```

Player.Bombs.Remove(bomb);
GameObject.Destroy(bomb);
if (!bombdropped) {
    Player.State = "IdleAnim";
    Player.animator.Play("IdleAnim");
    Player.PlayerStaff.SetActive(true);
    AbleToDropBomb = false;}}

void Update(){
    if (Input.GetButtonUp("Bomb") && Player.State == "IdleAnim"){
        Player.State = "HoldingBomb";
        Player.PlayerStaff.SetActive(false);
        Player.animator.Play("HoldingBomb");
        Player.Bombs.AddLast(Instantiate(Player.Bomb, transform.position + new Vector3(0, 7,
        0),Player.Bomb.transform.rotation,transform));
        StartCoroutine(BombCountdown(10,Player.Bombs.Last.Value));
        Invoke(nameof(WaitForBombDrop), 0.1f);}
    if (Input.GetButtonUp("Bomb") && Player.State == "HoldingBomb" &&
    AbleToDropBomb){
        Player.State = "IdleAnim";
        Player.animator.Play("IdleAnim");
        AbleToDropBomb = false;
        GameObject GO = Player.Bombs.Last.Value;
        Destroy(GO.GetComponent<BombExplosionScript>());
        GO.transform.parent = null;
        GO.transform.position = new Vector3(GO.transform.position.x, 45,
        GO.transform.position.z);
        GO.GetComponent<Rigidbody>().constraints = RigidbodyConstraints.None;
        GO.GetComponent<Rigidbody>().freezeRotation = true;
        Invoke(nameof(EndBomb), 0.1f);}
    if (Input.GetButtonUp("Fire1") && Player.State == "HoldingBomb"){
        Player.State = "ThrowingBomb";
        Player.animator.Play("ThrowBomb");
        StopCoroutine(BombCountdown(10, Player.Bombs.Last.Value));
        AbleToDropBomb = false;
        GameObject GO = Player.Bombs.Last.Value;
        GameObject.Destroy(Player.Bombs.Last.Value.transform.Find("Timer").gameObject);
        GO.transform.parent = null;
        GO.GetComponent<Rigidbody>().constraints = RigidbodyConstraints.None;
        GO.GetComponent<Rigidbody>().freezeRotation = true;
        GO.GetComponent<Rigidbody>().linearVelocity = (transform.forward * 70) + new
        Vector3(0, 30, 0);
        Invoke(nameof(EndBomb), 0.1f);}}}}

```

Також, для вибуху бомби нам потрібно створити новий візуальний ефект, який буде створюватись при її вибуху. Для цього можна використати Particle System, в якій нам потрібно включити Emission, змінити форму на сферу та додати швидкість (див. рис. 3.41).

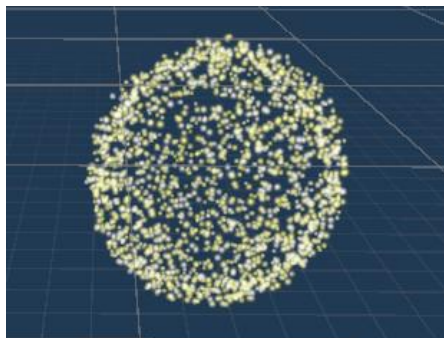


Рис. 3.41 Вибух бомби

Після того як ми створили вибух бомби, можна перейти до написання BombExplosionScript, який буде створювати цей вибух при завершенні таймера або при ударі об стіну. А також, при ударі об X будуть створюватись особливі секретні двері, які ми створили раніше. Ось такий вигляд має цей скрипт:

```
public class BombExplosionScript : MonoBehaviour{
    private void OnCollisionEnter(Collision collision){
        if (collision.collider.name != "MainCharacter" && Player.State != "HoldingBomb"){
            GameObject.Destroy(
                Instantiate(Player.BombExplosion, transform.position, Quaternion.identity), 3);
            Collider[] BombHitObjects = Physics.OverlapSphere(transform.position, 15);
            foreach (Collider b in BombHitObjects) {
                if (b.name.Contains("XDecal")){
                    Level.SecretRoomExploded = true;
                    GameObject.Destroy(Instantiate(Level.SecretRoomExplosion, b.transform.position,
                        Quaternion.identity), 5f);
                    Transform T =
                        GameObject.Find("Rooms").transform.Find(Player.CurrentRoom.RoomNumber.ToString()).Find("Doors").transform;
                    GameObject SecretDoor = Instantiate(Level.SecretRoomDoor, b.transform.position,
                        b.transform.rotation, T);
                    string direction = (b.transform.position.x > 10) ? "RightDoor":
                                        (b.transform.position.x < -10) ? "LeftDoor":
                                        (b.transform.position.z > 10) ? "TopDoor" :
                                        (b.transform.position.z < -10) ? "BottomDoor" :
                                        "Center";
                    SecretDoor.name = direction;
                    GameObject.Destroy(b.gameObject);}}
            Player.Bombs.Remove(gameObject);
            GameObject.Destroy(gameObject);}}}
```

Тепер у нас є повністю створена та робоча бомба, від візуальної до технічної частини. Цю бомбу можна кинути, залишити на землі з таймером, по закінченню таймера або при ударі об стіну вона вибухне.

### 3.5 Система здоров'я

Здоров'я зазвичай відображається звичайними сердечками, тому я вирішив зробити його саме таким. Для цього нам знадобляться три базові картинки (див. рис. 3.42). Також, у майбутньому можна буде створити додаткові види сердець.

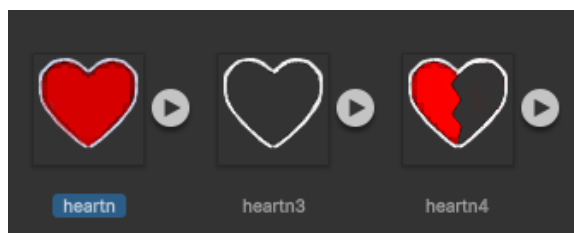


Рис. 3.42 Повне, пів та пусте серце

Далі, для створення системи здоров'я потрібно створити панель, так само як ми це робили під час створення карти. Для цього потрібно перейти в режим 2D та створити панель в списку редактора, прикріпити її зліва зверху та поставити розміри 500 на 1000. Ця панель повинна вміщати 20 сердець і має виглядати ось так (див. рис. 3.43):

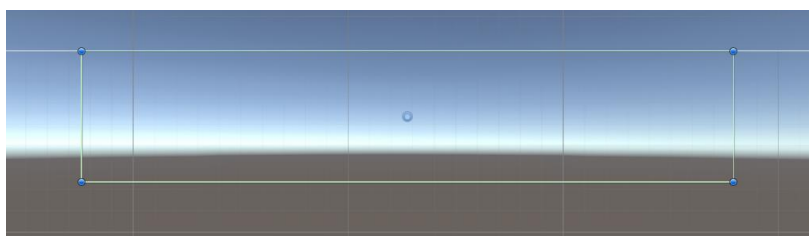


Рис. 3.43 Панель для сердець

Далі всередині панелі, яку ми щойно створили, назвемо її Hearts, потрібно створити ще одну панель для уявлення серця. Цю панель, також, потрібно закріпити зліва зверху та встановити розміри 50 на 50. Далі потрібно перейти до створення скрипта, який буде співпрацювати із нашою панеллю. Створивши наш скрипт, я його назвав HeartScript, потрібно задекларувати три змінні для кожного із виду сердець та додати їх у вільні слоти (див. рис. 3.43). Ці слоти з'являться, коли скрипт буде додано до головного героя.

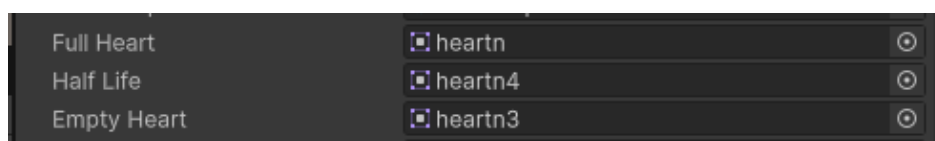


Рис. 3.44 Вільні слоти для картинок сердець

Після цього, нам потрібно написати статичний метод, який дозволить нам намалювати серця на екрані. Цей метод повинен прийняти те, яке серце намалювати, та скільки, тому цей метод буде виглядати так:

```
public static void DrawHeart(Sprite Type, int num){
    GameObject Heart = new GameObject("Heart");
    Image HeartImage = Heart.AddComponent<Image>();
    HeartImage.sprite = Type;
    RectTransform rectTransform = Heart.GetComponent<RectTransform>();
    rectTransform.sizeDelta = new
Vector2(Player.HeartPanel.GetComponent<RectTransform>().sizeDelta.x / 10,
Player.HeartPanel.GetComponent<RectTransform>().sizeDelta.x / 10);
    Animator animator = Heart.AddComponent<Animator>();
    animator.runtimeAnimatorController = Player.HeartAnimator;
    float XPos = 0;
    float YPos = -5;
    if (num <= 9){
        XPos = num * Player.HeartPanel.GetComponent<RectTransform>().sizeDelta.x / 10;}
    else{
XPos = (num - 10) * Player.HeartPanel.GetComponent<RectTransform>().sizeDelta.x /
10;
        YPos = -5 - Player.HeartPanel.GetComponent<RectTransform>().sizeDelta.x / 10;
        rectTransform.position = new Vector2(XPos, YPos);
        rectTransform.pivot = new Vector2(0f, 1f);
        rectTransform.anchorMin = new Vector2(0, 1);
        rectTransform.anchorMax = new Vector2(0, 1);
        HeartImage.transform.SetParent(Player.HeartPanel.transform, false);}
```

Цей метод буде малювати 10 сердець в одному ряду, а кожне наступне серце після 10 – в другому. Далі, нам потрібно створити декілька нових значень в класі Player, а саме MaxHealth та CurrentHealth. Ці дані дозволять нам правильно намалювати кількість сердець. Після цього, нам потрібно написати метод, який відобразить відповідну кількість сердець до наших значень максимального та поточного здоров'я:

```
public static void DrawHearts(){
    for (int i = 0; i <= Player.Health - 1; i++){
        DrawHeart(Player.FullHeart, i);}
    if (Player.Health % 1 != 0){
        DrawHeart(Player.HalfLife, (int)Player.Health);}
    for (int i = (int)Player.Health; i <= Player.MaxHealth - 1; ++i){
        DrawHeart(Player.EmptyHeart, (int)i);}}
```

Тепер, якщо всі методи та скрипти були створені правильно можна запуснути гру і у вашого персонажа буде вірно відображатися здоров'я (див.



рис. 3.45), в моєму випадку у мене максимальне здоров'я стоїть 3 та поточне здоров'я теж.



Рис. 3.45 Здоров'я головного героя

Опісля, нам потрібно модифікувати наш скрипт так, щоб наш головний герой міг отримувати шкоду, та при досяганні поточного здоров'я до 0 він помирав. Також, варто створити панель, яка буде з'являтися при отриманні шкоди, оскільки вона допоможе з сприйняттям ситуації. Для того, щоб це все зробити, потрібно створити анімацію серця, де серце збільшується та пропадає, створити ударну панель та написати метод, в якому буде оновлюватися кількість здоров'я нашого героя відповідно до отриманої шкоди. Для створення панелі потрібно просто перейти в 2D режим та створити її, далі перефарбувати її в червоний та зменшити прозорість приблизно до 40% (див. рис. 3.46).

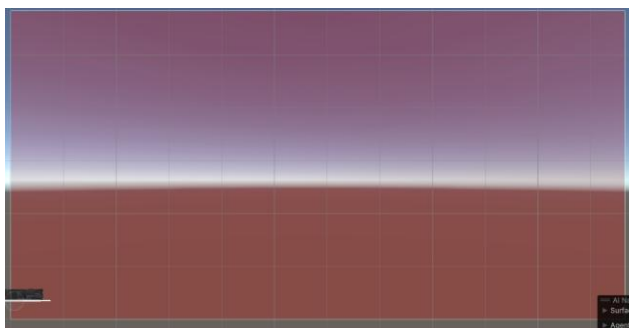


Рис. 3.46 Червона панель

Аби наш головний герой міг отримувати шкоду, потрібно написати статичний метод TakeDamage. У цьому методі потрібно обраховувати скільки повинно залишатися здоров'я після удару та показувати червону панель десь на 1 або 0.5 секунди після отримання шкоди. Цей метод доволі простий та повинен виглядати ось так:

```
public static void TakeDamage(float damage) {
    if (!Player.Invincible){
        Player.Invincible = true;
        CoroutineManager.Instance.StartCoroutine(Uninvincible());
        for (int i = 1; i <= damage; ++i){
```

```

        (Instantiate(Player.HeartPanel.transform.GetChild((int)Player.Health - i)
, Player.HeartPanel.transform)).GetComponent<Animator>().Play("HeartAnim");
Player.HeartPanel.transform.GetChild((int)Player.Health -
i).GetComponent<Image>().sprite = Player.EmptyHeart;}
    Player.Health -= damage;

```

Також, по отриманню шкоди потрібно викликати метод для перерахування кількості сердець. Цей метод теж простий, він і буде видаляти потрібну кількість сердець та показувати нашу червону панель. У мене він називається WaitAndRedrawHearts:

```

public static IEnumerator WaitAndRedrawHearts(){
    Player.DamagePanel.SetActive(true);
    yield return new WaitForSeconds(.1f);
    Player.DamagePanel.SetActive(false);
    yield return new WaitForSeconds(.4f);
    for (int i = 0; i < Player.HeartPanel.transform.childCount; ++i){
        GameObject.Destroy(Player.HeartPanel.transform.GetChild(i).gameObject);}
    DrawHearts();}

```

Окрім цього, коли у персонажа закінчиться поточне здоров'я, буде викликаний метод Die, який видалить усі скрипти з персонажа, знищить усі панелі та відтворить анімацію смерті, саме так я реалізував цей метод:

```

public static IEnumerator Die(){
    Player.DamagePanel.SetActive(true);
    yield return new WaitForSeconds(0f);
    for (int i = 0; i < Player.HeartPanel.transform.childCount; ++i){
        GameObject.Destroy(Player.HeartPanel.transform.GetChild(i).gameObject);}
    Player.animator.Play("Die");
    Player.State = "Dead";
    Player.Invincible = true;
    Destroy(Player.transform.GetComponent<PlayerMovement>());
    Destroy(Player.transform.GetComponent<PlayerAttack>());
    Destroy(Player.transform.GetComponent<ChangeRooms>());
    Destroy(Player.transform.GetComponent<BombScript>());
    Player.DamagePanel.SetActive(false);}

```

Це все, що потрібно для створення повноцінної системи здоров'я в іграх жанру Rogue-like. Якщо все у вас вийшло, то при отриманні шкоди буде програватися анімація та оновлюватися кількість здоров'я (див. рис. 3.47). Для перевірки, можна створити кнопку при натисканні якої, буде отримана шкода.



Рис. 3.47 Отримання шкоди

### 3.6 Противники та алгоритм їх поведінки

Для створення перших противників нам потрібно спочатку розробити алгоритм, за яким вони будуть це робити. У моєму випадку я вирішив зробити це за допомогою алгоритму A\*, який є одним із найпопулярніших методів для реалізації штучного інтелекту в іграх [4]. Також, альтернативою могло бути використання NavMesh Agent, яке є вбудоване у сам двигун Unity. Для створення A\* нам потрібно створити декілька скриптів, а саме: Grid, Path, PathFindingRequest, Line, Heap, PathFinding, Unit та EnemyDamageScript. Тож A\* працює таким чином, що він ділить усю ділянку, яку ми визначимо на квадратики певної величини, яку ми також можемо пізніше визначити. Окрім цього, цей алгоритм бере точку А та точку Б, обраховує якими квадратами краще йти так, щоб дістатись до точки Б якомога швидше. Для того, аби створити цю, так звану, сітку квадратиків, потрібно написати скрипт Grid, який і буде обраховувати нашу територію. Цей скрипт повинен мати такий вигляд:

```
public bool DisplayGridGizmos;
public LayerMask unwalkableMask;
LayerMask walkableMask;
public Vector2 gridWorldSize;
public float nodeRadius;
public Node[,] grid;
public TerrainType[] WalkableRegions;
public int obstacleProximityPenalty = 10;
Dictionary<int, int> walkableRegionsDictionary = new Dictionary<int, int>();
int penaltyMin = int.MaxValue;
int penaltyMax = int.MinValue;
float nodeDiameter;
int gridSizeX, gridSizeY;
private void Awake(){
    nodeDiameter = nodeRadius * 2;
    gridSizeX = Mathf.RoundToInt(gridWorldSize.x / nodeDiameter);
    gridSizeY = Mathf.RoundToInt(gridWorldSize.y / nodeDiameter);
    foreach(TerrainType region in WalkableRegions){
        walkableMask.value = walkableMask | region.terrainMask.value;
        walkableRegionsDictionary.Add((int)Mathf.Log(region.terrainMask.value, 2), region.terrainPenalty);
    }
    CreateGrid();}
```

```

public int MaxSize{
    get { return gridSizeX*gridSizeY; }}
void CreateGrid(){
    grid = new Node[gridSizeX, gridSizeY];
    Vector3 WorldBottomLeft = transform.position - Vector3.right *
gridWorldSize.x / 2 - Vector3.forward * gridWorldSize.y / 2;
    for (int x = 0; x < gridSizeX; x++){
        for (int y = 0; y < gridSizeY; y++){
            Vector3 worldPoint = WorldBottomLeft + Vector3.right * (x * nodeDiameter +
nodeRadius) + Vector3.forward * (y * nodeDiameter + nodeRadius);
            bool walkable = !(Physics.CheckSphere(worldPoint, nodeRadius, unwalkableMask));
            int movementPenalty = 0;
            Ray ray = new Ray(worldPoint + Vector3.up * 50, Vector3.down);
            RaycastHit hit;
            if (Physics.Raycast(ray, out hit, 100, walkableMask)){
                walkableRegionsDictionary.TryGetValue(hit.collider.gameObject.layer, out
movementPenalty);}
            if (!walkable){
                movementPenalty += obstacleProximityPenalty;}
            grid[x, y] = new Node(walkable, worldPoint, x, y, movementPenalty);}
        BlurPenaltyMap(5);}
    void BlurPenaltyMap(int blurSize){
        int kernelSize = blurSize * 2 + 1;
        int kernelExtents = (kernelSize - 1) / 2;
        int[,] penaltiesHorizontalPass = new int[gridSizeX, gridSizeY];
        int[,] penaltiesVerticalPass = new int[gridSizeX, gridSizeY];
        for(int y = 0; y< gridSizeY; y++){
            for(int x = -kernelExtents; x<= kernelExtents; x++){
                int sampleX = Mathf.Clamp(x, 0, kernelExtents);
                penaltiesHorizontalPass[0, y] += grid[sampleX, y].movementPenalty;}
            for(int x = 1; x < gridSizeX; x++){
                int removeIndex = Mathf.Clamp(x - kernelExtents - 1,0,gridSizeX);
                int addIndex = Mathf.Clamp(x + kernelExtents,0,gridSizeX-1);
                penaltiesHorizontalPass[x, y] = penaltiesHorizontalPass[x - 1, y] -
grid[removeIndex, y].movementPenalty + grid[addIndex, y].movementPenalty;
            }}
        for (int x = 0; x < gridSizeX; x++){
            for (int y = -kernelExtents; y <= kernelExtents; y++){
                int sampleY = Mathf.Clamp(y, 0, kernelExtents);
                penaltiesVerticalPass[x, 0] += penaltiesHorizontalPass[x, sampleY];} int
blurredPenalty = Mathf.RoundToInt((float)penaltiesVerticalPass[x, 0] / (kernelSize
* kernelSize));
                grid[x, 0].movementPenalty = blurredPenalty;
            for (int y = 1; y < gridSizeY; y++){
                int removeIndex = Mathf.Clamp(y - kernelExtents - 1, 0, gridSizeY);
                int addIndex = Mathf.Clamp(y + kernelExtents, 0, gridSizeY - 1);

                penaltiesVerticalPass[x, y] = penaltiesVerticalPass[x, y - 1] -
penaltiesHorizontalPass[x, removeIndex] + penaltiesHorizontalPass[x, addIndex];
                blurredPenalty = Mathf.RoundToInt((float)penaltiesVerticalPass[x,
y] / (kernelSize * kernelSize));
                grid[x, y].movementPenalty = blurredPenalty;
            }
        }
    }
}

```

```

        if (blurredPenalty > penaltyMax){
            penaltyMax = blurredPenalty;}
        if(blurredPenalty < penaltyMin){
            penaltyMin = blurredPenalty;}} }
public List<Node> GetNeighbors(Node node){
    List<Node> neighbors = new List<Node>();
    for(int x = -1; x<= 1; x++){
        for (int y = -1; y <= 1; y++){
            if (x == 0 && y == 0)
                continue;
            int checkX = node.gridX + x;
            int checkY = node.gridY + y;
            if(checkX >= 0 && checkX < gridSizeX && checkY >= 0 && checkY < gridSizeY){
                neighbors.Add(grid[checkX, checkY]);}}}
    return neighbors;}
public Node NodeFromWorldPoint (Vector3 worldPosition){
    float percentX = (worldPosition.x + gridWorldSize.x / 2) / gridWorldSize.x;
    float percentY = (worldPosition.z + gridWorldSize.y / 2) / gridWorldSize.y;
    percentX = Mathf.Clamp01(percentX);
    percentY = Mathf.Clamp01(percentY);
    int x = Mathf.RoundToInt((gridSizeX - 1) * percentX);
    int y = Mathf.RoundToInt((gridSizeY - 1) * percentY);
    return grid[x, y];}
private void OnDrawGizmos(){
    Gizmos.DrawWireCube(transform.position, new Vector3(gridWorldSize.x, 1,
gridWorldSize.y));
    if (grid != null && DisplayGridGizmos){
        foreach (Node n in grid){
            Gizmos.color = Color.Lerp(Color.white, Color.black,
Mathf.InverseLerp(penaltyMin, penaltyMax, n.movementPenalty));
            Gizmos.color = (n.Walkable) ? Gizmos.color : Color.red;
            Gizmos.DrawCube(n.WorldPosition, Vector3.one * (nodeDiameter));}}}}
[System.Serializable]
public class TerrainType{
    public LayerMask terrainMask;
    public int terrainPenalty;}
public class Node : IHeapItem<Node>{
    public bool Walkable;
    public Vector3 WorldPosition;
    public int gCost;
    public int hCost;
    public Node parent;
    int heapIndex;
    public int gridX;
    public int gridY;
    public int movementPenalty;
    public Node(bool _walkable, Vector3 _worldPosition, int _gridX, int _gridY, int
_penalty){
        Walkable = _walkable;
        WorldPosition = _worldPosition;
        gridX = _gridX;
        gridY = _gridY;

```



```

        if (!openSet.Contains(neighbor)){
            openSet.Add(neighbor);}
        else { openSet.UpdateItem(neighbor); }}}}
    if(pathSuccess){
        waypoints = RetracePath(startNode, targetNode);
        pathSuccess = waypoints.Length > 0;}
    callback(new PathResult(waypoints, pathSuccess, request.callback));}
Vector3[] RetracePath(Node startNode, Node endNode){
    List<Node> path = new List<Node>();
    Node currentNode = endNode;
    while(currentNode != startNode){
        path.Add(currentNode);
        currentNode = currentNode.parent;}
    Vector3[] waypoints = SimplifyPath(path);
    Array.Reverse(waypoints);
    return waypoints;    }

Vector3[] SimplifyPath(List<Node> path){
    List<Vector3> waypoints = new List<Vector3>();
    Vector2 directionOld = Vector2.zero;
    for(int i = 1;i<path.Count;i++){
        Vector2 directionNew = new Vector2(path[i - 1].gridX - path[i].gridX,
path[i - 1].gridY - path[i].gridY);
        if(directionNew != directionOld){
            waypoints.Add(path[i].WorldPosition);}
        directionOld = directionNew;}
    return waypoints.ToArray();}
int GetDistance(Node nodeA, Node nodeB){
    int dstX = Mathf.Abs(nodeA.gridX - nodeB.gridX);
    int dstY = Mathf.Abs(nodeA.gridY - nodeB.gridY);
    if(dstX > dstY){
        return 14 *dstY+ 10* (dstX-dstY);}
    return 14 * dstX + 10 * (dstY - dstX);}}
public class PathRequestManager : MonoBehaviour{
    Queue<PathResult> results = new Queue<PathResult>();
    static PathRequestManager instance;
    PathFinding pathfinding;
    private void Awake(){
        instance = this;
        pathfinding = GetComponent<PathFinding>();}
    private void Update(){
        if (results.Count > 0){
            int itemsInQueue = results.Count;
            lock (results){
                for(int i =0 ; i < itemsInQueue; i++){
                    PathResult result = results.Dequeue();
                    result.callback(result.path, result.success);}}}}
    public static void RequestPath(PathRequest request){
        ThreadStart threadStart = delegate{
            instance.pathfinding.FindPath(request, instance.FinishedProcessingPath); };
        threadStart.Invoke();}

```

```

    public void FinishedProcessingPath(PathResult result){
        lock (results){
            results.Enqueue(result);}}    }
public struct PathResult{
    public Vector3[] path;
    public bool success;
    public Action<Vector3[], bool> callback;
    public PathResult(Vector3[] path, bool success, Action<Vector3[], bool>
callback){
        this.path = path;
        this.success = success;
        this.callback = callback;}}
public struct PathRequest{
    public Vector3 pathStart;
    public Vector3 pathEnd;
    public Action<Vector3[], bool> callback;
    public PathRequest(Vector3 _start, Vector3 _end, Action<Vector3[], bool>
_callback){
        pathStart = _start;
        pathEnd = _end;
        callback = _callback;}}

```

Окрім цього, потрібно написати скрипт Unit, який буде використовуватися для усіх створених противників, нижче наведений цей скрипт:

```

public class Unit : EnemyDamageScript{
    Grid g;
    public bool DrawPath = false;
    const float minPathUpdateTime = .2f;
    const float pathUpdateMoveThreshold = .5f;
    [Header("0: Chase Player\n" +
        "1: Random Path")]
    public int MonsterMovementType = 0;
    public float Health = 3;
    public GameObject EnemyDeathEffect;
    public Transform target;
    public float speed = 5f;
    public float turnDst = 5f;
    public float TurnSpeed = 3;
    Path path;
    [HideInInspector] public Rigidbody rb;
    protected override void Start(){
        base.Start();
        rb = GetComponent<Rigidbody>();
        g = GameObject.Find("A*").GetComponent<Grid>();
        switch (MonsterMovementType){
            case 0: StartCoroutine(UpdatePath()); break;
            case 1: StartCoroutine(UpdatePathRandom()); break;}}
    public bool HitPlayer = false;

```



```

public void GivePlayerASecond(){
    HitPlayer = false;}
public void OnPathFound(Vector3[] waypoints, bool pathSuccessful){
    if (!pathSuccessful) return;
    path = new Path(waypoints, transform.position, turnDst);
    StopCoroutine("FollowPath");
    StartCoroutine("FollowPath");}
IEnumerator UpdatePathRandom(){
    if (Time.timeSinceLevelLoad < .3f){
        yield return new WaitForSeconds(.3f);}
    while (true){
        yield return new WaitForSeconds(.2f);
        Node curN = g.NodeFromWorldPoint(transform.position);
        int max = g.grid.GetLength(0) - 1;
        int newX = Mathf.Clamp(curN.gridX + Random.Range(-25, 25), 3, max - 3);
        int newY = Mathf.Clamp(curN.gridY + Random.Range(-25, 25), 3, max - 3);
        int iterationNumber = 0;
        while (!g.grid[newX, newY].Walkable){
            iterationNumber++;
            if (iterationNumber > 10) break;
            newX = Mathf.Clamp(curN.gridX + Random.Range(-25, 25), 3, max - 3);
            newY = Mathf.Clamp(curN.gridY + Random.Range(-25, 25), 3, max - 3);
            curN = g.grid[newX, newY];}
        Vector3 RPos = g.grid[newX, newY].WorldPosition;
        PathRequestManager.RequestPath(new PathRequest(transform.position, RPos,
OnPathFound));}}
IEnumerator UpdatePath(){
    if (Time.timeSinceLevelLoad < .3f){
        yield return new WaitForSeconds(.1f);}
    PathRequestManager.RequestPath(new PathRequest(transform.position,
target.position, OnPathFound));
    float sqrMoveThreshold = pathUpdateMoveThreshold * pathUpdateMoveThreshold;
    Vector3 targetPosOld = target.position;
    float counter = 0;
    while (true){
        yield return new WaitForSeconds(minPathUpdateTime);
        ++counter;
        if ((target.position - targetPosOld).sqrMagnitude > sqrMoveThreshold ||
counter > 10){
            PathRequestManager.RequestPath(new PathRequest(transform.position,
target.position, OnPathFound));
            targetPosOld = target.position;
            counter = 0;}}}}
public virtual void FollowPathAction(Path path, int pathIndex){
    Quaternion targetRotation =
Quaternion.LookRotation(path.lookPoints[pathIndex] - transform.position);
    transform.rotation = Quaternion.Lerp(transform.rotation, targetRotation,
Time.deltaTime * TurnSpeed);
    transform.Translate(Vector3.forward * Time.deltaTime * speed, Space.Self);}
IEnumerator FollowPath(){
    bool followingPath = true;
    int pathIndex = 0;

```

```

while (followingPath){
    Vector2 pos2D = new Vector2(transform.position.x, transform.position.z);
    while (path.turnBoundaries[pathIndex].HasCrossedLine(pos2D)){
        if (pathIndex == path.finishLineIndex){
            followingPath = false;
            break;}
        else{
            pathIndex++;}}
    if (followingPath){
        FollowPathAction(path, pathIndex);}
    yield return null;}}
public void OnDrawGizmos(){
    if (path != null && DrawPath){
        path.DrawWithGizmos();}}}

```

Наступним кроком, щоб імплементувати це все у вашій грі потрібно, створити порожній ігровий об'єкт A\* та додати на нього скрипти: Grid, PathFinding, PathFindingRequest. По тому, варто вибрати розміри нашої сітки та окремих квадратиків. Що менше число, то більша точність, але гірша продуктивність, тому найкраще лишити її 1, обрати, що потрібно уникати, та по чому можна ходити. Для того щоб противники оминали перешкоду, потрібно створити шар Obstacle та додати його на предмет, який потрібно оминати. У результаті ми отримаємо налаштований скрипт (див. рис. 3.48).

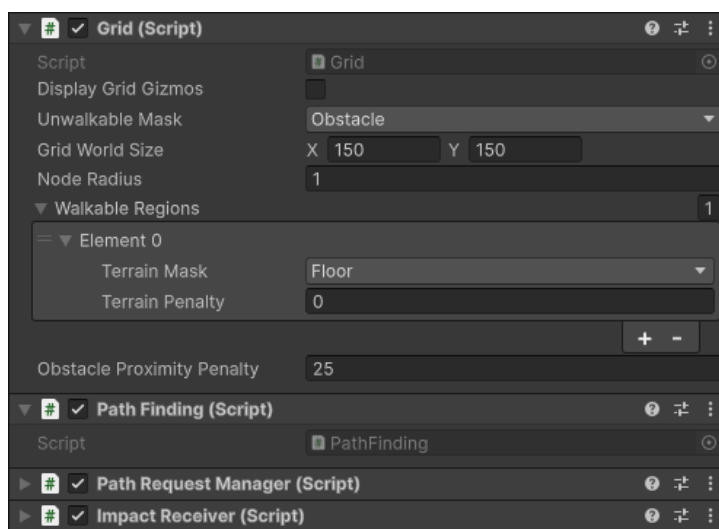


Рис. 3.48 Налаштування A\*

Якщо при запуск все було налаштовано вірно, у вас повинна відображатися сітка, яка виділяє всі перешкоди червоним (див. рис. 3.49).

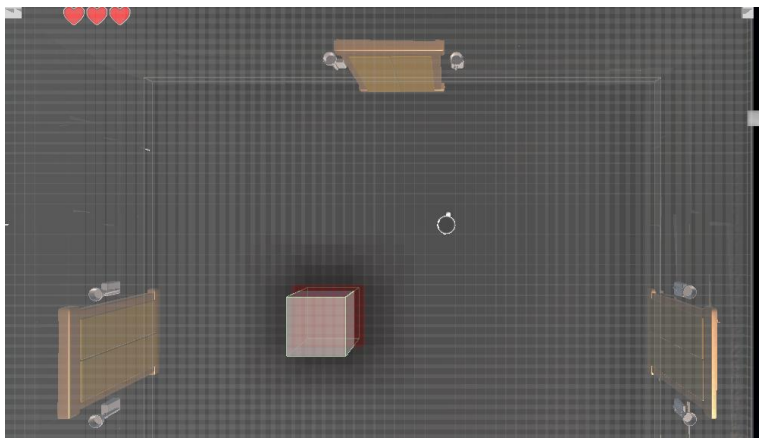


Рис. 3.49 Сітка A\*

Після налаштування A\*, можна приступити до написання першого противника. Першим кроком буде додавання його моделі до гри та створення скрипта для вашого противника. У моєму разі це слимак – зелений, круглий м'ячик, який, наче слимак, бігає за головним героєм (див. рис. 3.49).

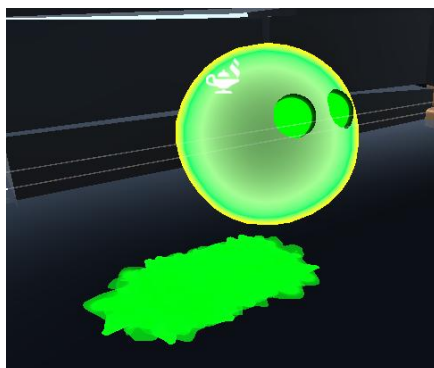


Рис 3.50 Сликак

Скрипт повинен наслідуватися від Unit, оскільки в ньому є все, що потрібно для того, аби наш слимак або будь-який інший майбутній противник міг слідувати за головним героєм. У цьому скрипті потрібно переписати FollowPathAction, щоб наш слимак мав свою відповідну поведінку, швидкість і решту характеристик. Сликак це противник в якого 3 здоров'я та котрий прямує до головного героя ривками. Якщо той торкнеться головного героя, то нанесе йому 1 одиницю шкоди. Ось так повинен виглядати скрипт для слимака:

```
public class SlimeScript : Unit{
    private float timeCounter = 0.0f;
    public float oscillationSpeed = 1.0f;
```

```

public float minSpeed = 20;
public float maxSpeed = 500;
public float minScale = 1.5f;
public float maxScale = 2.5f;
override public void FollowPathAction(Path path, int pathIndex) {
    Quaternion targetRotation =
Quaternion.LookRotation(path.lookPoints[pathIndex] - transform.position);
    transform.rotation = Quaternion.Lerp(transform.rotation, targetRotation,
Time.deltaTime * TurnSpeed);
    float sineValue = Mathf.Sin(timeCounter * oscillationSpeed);
    float CurrentSpeed = Mathf.Lerp(minSpeed, maxSpeed, (sineValue + 1.0f) / 2.0f);
    speed = CurrentSpeed;
    float scale = Mathf.Lerp(minScale, maxScale, (sineValue + 1.0f) / 2.0f);
transform.localScale = new Vector3(Mathf.Clamp(scale, minScale, minScale + .5f),
minScale, scale);
    rb.linearVelocity = transform.forward * Time.deltaTime * CurrentSpeed * 200;
    timeCounter += Time.deltaTime;}
private void OnCollisionEnter(Collision collision){
    if(collision.collider.name == "MainCharacter"){
        ImpactReceiver.AddImpactOnGameObject(Player.transform.gameObject,
(Player.transform.position - transform.position) * Player.KnockBack);
        HeartScript.TakeDamage(1);}}}

```

Після створення скрипта можна розмістити декілька слимаків в одній із кімнат, в папці Enemies, і готово, у вас є перший противник (див. рис. 3.51).

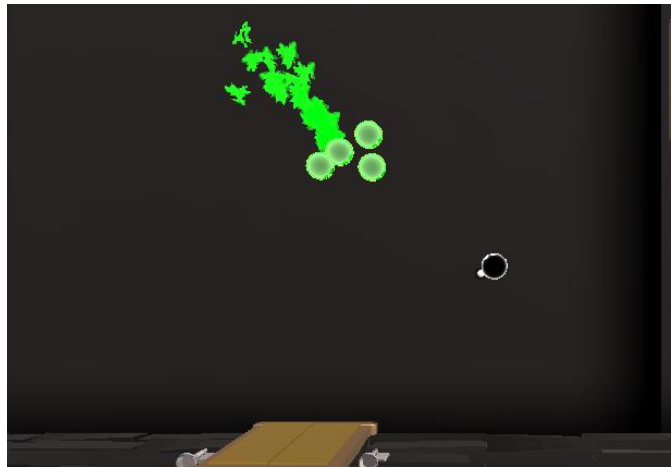


Рис. 3.51 Декілька слимаків в одній кімнаті

За тим самим принципом можна створювати безліч противників різних за поведінкою, до прикладу робота, який буде, як навіжений, бігати за головним героєм [7] (див. рис. 3.52).

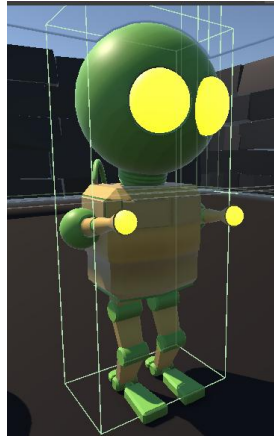


Рис. 3.52 Робот

Створення робота є абсолютно ідентичним до створення слимака, робот всього лише потребує нового скрипта. У цьому скрипті треба прописати, як він буде поводитися, і як наслідок, ви отримаєте повноцінного противника, який, як навіжений, буде бігати за головним героєм при вході до кімнати. А також, варто тільки додати декілька роботів або комбінацій противників у кімнату, і у вас трапиться бійка. Оскільки ці противники наслідують усе з класу Unit, вони отримуватимуть шкоду при отриманні удару від нашого головного героя та будуть помирати від трьох ударів. Нижче наведений скрипт для робота:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class RobotScript : Unit{
    override public void FollowPathAction(Path path, int pathIndex){
        if (HitPlayer) return;
        Quaternion targetRotation =
Quaternion.LookRotation(path.lookPoints[pathIndex] - transform.position);
        Quaternion newTR = Quaternion.Euler(0, targetRotation.eulerAngles.y, 0);
        transform.rotation = Quaternion.Slerp(transform.rotation, newTR,
Time.deltaTime * TurnSpeed);
        Vector3 translation = transform.forward * speed * Time.deltaTime;
        rb.MovePosition(rb.position + translation);
        rb.linearVelocity = Vector3.zero;
    }
    private void OnCollisionEnter(Collision collision){
        if (collision.collider.name == "MainCharacter"){
            HitPlayer = true;
            ImpactReceiver.AddImpactOnGameObject(Player.transform.gameObject,
(Player.transform.position - transform.position) * Player.KnockBack);
            HeartScript.TakeDamage(1);
            Invoke(nameof(GivePlayerASecond), .2f);}
        if (collision.collider.name.Contains("Robot")){
collision.collider.transform.GetComponent<Rigidbody>().AddForce((collision.collider.
```

```

transform.position - transform.position) * Player.EnemyKnockBack,
ForceMode.Impulse);
    Unit U;
    collision.collider.TryGetComponent<Unit>(out U);
    U.HitPlayer = true;
    StartCoroutine(ResetHitPlayer(U));}}
private IEnumerator ResetHitPlayer(Unit U){
    yield return new WaitForSeconds(0.2f);
    U.HitPlayer = false;}}}

```

Це, власне, все що стосується створення противників, їхнього налаштування та поведінки. Тепер ви можете зайти в гру, дійти до кімнати, де ви розташували ворогів та поспробувати їх подолати (див. рис. 3.53).



Рис. 3.53 Отримання шкоди противником

## ВИСНОВОК

У цьому дослідженні було створено гру у жанрі Rogue-like, яка об'єднує в собі елементи процедурної генерації, бою з різними противниками та дослідження карти. Увага приділялася не лише візуальному аспекту гри, але й розробці алгоритмів, що забезпечують глибину геймплею та адаптивність оточення до дій гравця. Були використані алгоритми процедурної генерації карти, що дозволяють створювати унікальні рівні під час кожної нової гри. Логіка генерації кімнат, розміщення ключових об'єктів, таких як кімната із предметом, вороги та секретні кімнати, була ретельно пророблена.

Особливу увагу було приділено моделюванню поведінки противників за допомогою алгоритму  $A^*$ . Це дозволило продуктивно та ефективно налаштувати реакцію штучного інтелекту на рухи гравця по кімнаті. Шляхом експериментування було оптимізовано параметри алгоритму для досягнення оптимальної складності гри та її продуктивності. Гра містить систему здоров'я для гравця та ворогів з можливістю завдавати шкоду та помирати. Система бою включає як ближні, так і віддалені бої, а також механіку використання бомб для відкриття секретних кімнат.

У роботі подано детальний опис всіх етапів створення проекту, починаючи від концепції, закінчуючи реалізацією ключових функцій у Unity. Окрім цього, наведено численні ілюстрації, скріншоти та фрагменти коду з поясненнями, які можуть бути корисними для майбутніх проектів у цьому жанрі. Структурований підхід до розробки дозволяє краще розуміти логіку рішень та їх реалізацію.

Підсумовуючи, ця робота показує актуальність жанру Rogue-like, вказує на те, що саме робить його привабливим для широкої аудиторії, а також доводить, що цей жанр сповнений інновацій та має простір для розвитку, тим самим збільшуючи свою аудиторію.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Історія гри Risk of Rain 2: [Електронний ресурс] – Режим доступу: [https://uk.wikipedia.org/wiki/Risk\\_of\\_Rain\\_2](https://uk.wikipedia.org/wiki/Risk_of_Rain_2)
2. Статистика у Steam Charts: [Електронний ресурс] – Режим доступу: <https://steamdb.info/app/250900/charts/>
3. Версії Unity: [Електронний ресурс] – Режим доступу: <https://unity.com/products>
4. Створення свого алгоритму А\*: [Електронний ресурс] – Режим доступу: [https://www.youtube.com/playlist?list=PLFt\\_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW](https://www.youtube.com/playlist?list=PLFt_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW)
5. Модель кулі: [Електронний ресурс] – Режим доступу: <https://www.dropbox.com/scl/fi/dqpkttpmjy8t4mwy13j5s/DefaultAttackExplosion.zip?rlkey=hq38o3pscj5isxq0d748ok9kb&e=1&dl=0>
6. Модель бомби: [Електронний ресурс] – Режим доступу: <https://www.dropbox.com/scl/fi/f0iuvo24t1pnzb3cbyheh/Bombz.zip?rlkey=aewz336l0q7uy898iplgeqwim&e=1&dl=0>
7. Модель робота: [Електронний ресурс] – Режим доступу: <https://www.dropbox.com/scl/fi/x0de2tj1r1k1d7koapyl/Robot.2024.05.25.fbx?rlkey=2wby9ixs8xp1aesuefd2u4hfb&e=1&st=qyev8ged&dl=0>
8. Basic BSP Dungeon generation: [Електронний ресурс] – Режим доступу: [https://www.roguebasin.com/index.php/Basic\\_BSP\\_Dungeon\\_generation](https://www.roguebasin.com/index.php/Basic_BSP_Dungeon_generation)
9. Bazzaz M., Cooper S. Analysis of Uncertainty in Procedural Maps in Slay the Spire / M. Bazzaz, S. Cooper. – Northeastern University, Boston, Massachusetts, USA. – 2025.
10. Complexity, Controversy, Creativity: The History of the Roguelike Genre: [Електронний ресурс] – Режим доступу: <https://podcasts.ceu.edu/content/history-roguelike-genre>
11. Dungeon Generation in Unexplored: [Електронний ресурс] – Режим доступу: <https://www.boristhebrave.com/2021/04/10/dungeon-generation-in-unexplored>
12. Newzoo: Game market will hit \$200B in 2024: [Електронний ресурс] – Режим доступу: <https://venturebeat.com/games/newzoo-game-market-will-hit-200b-in-2024>
13. Optimize Renderer Features with Variable Rate Shading in Unity 6.1: [Електронний ресурс] – Режим доступу:



<https://discussions.unity.com/t/optimize-renderer-features-with-variable-rate-shading-in-unity-6-1/1605893>

14. Perlin Noise: Implementation, Procedural Generation, and Simplex Noise: [Электронный ресурс] – Режим доступа: <https://garagefarm.net/blog/perlin-noise-implementation-procedural-generation-and-simplex-noise>
15. The Game Awards 2020: [Электронный ресурс] – Режим доступа: <https://thegameawards.com/rewind/year-2020>
16. The Glorious (and Painful) History of Visual Studio: [Электронный ресурс] – Режим доступа: <https://levelup.gitconnected.com/the-glorious-and-painful-history-of-visual-studio-7a182ece05f7>
17. Unity asset store: [Электронный ресурс] – Режим доступа: <https://support.unity.com/hc/en-us/articles/210142503-What-is-the-Unity-Asset-Store-and-how-do-I-purchase-Assets#:~:text=The%20Unity%20Asset%20Store%20is%20home%20to%20a%20growing%20library,%2C%20tutorials%2C%20and%20Extension%20Assets>
18. Valerii Filatov History of C#: versions, .NET, Unity, Blazor, and MAUI: [Электронный ресурс] – Режим доступа: <https://pvs-studio.com/en/blog/posts/csharp/1248/>