МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
КАФЕДРА КОМП’ЮТЕРНИХ НАУК

ВИПУСКНА РОБОТА

на тему:
«Інформаційна система підтримки відеогри з процедурною генерацією рівнів на Unity»

Завідувач
випускаючої кафедри **А. С. Довбиш**

Керівник роботи **О. А. Шовкопляс**

Студент групи ІН-72 **В. О. Фоменко**

СУМИ 2021

Сумський державний університет

(назва вузу)

Факультет Еліт Кафедра Комп'ютерних наук
Спеціальність «Комп'ютерні науки»

Затверджую___________

Зав. кафедри Довбиш А. С.

«_____» ____________20___ р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТУ

Фоменко Вячеславу Олександровичу
(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) Інформаційна система підтримки відеогри з процедурною генерацією рівнів на Unity
затверджую наказом по університету від «____» ________________ 20 ___ р. № _____________

2. Термін здачі студентом закінченого проєкту (роботи) ______________________________

3. Вхідні дані до проєкту (роботи)____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити) 1) Інформаційний огляд. Процедурна генерація, прикладі використання, методи Лабіринти: різновиди, категорії класифікації. Лабіринти: алгоритми генерації та знаходження шляху. Постановка задачі 2) Інформаційний огляд програм для вирішення задачі. Середовище розробки Unity Мова програмування C# Жанр гри її особливості 3) Програмна реалізація Опис алгоритма генерації та опис вигляду програми. Результати

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти до проєкту (роботи), із значенням розділів проєкту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання ____

Керівник ____________________
(підпис)

Завдання прийняв до виконання ____________________
(підпис)

РЕФЕРАТ

Записка: 64 стор., 35 рис., 1 додаток, 21 джерел.

Об’єкт дослідження – комп’ютерні методи створення ігор із процедурною генерацією рівнів.

Мета роботи – на основі методів процедурної генерації створити 3 d гру використовуючи платформу Unity, реалізувати управління персонажем від третьої особи, розробити користувацький інтерфейс, сцени з головоломками та босом, розробити моделі персонажів.

Методи дослідження – методи процедурної генерації , зокрема рекурсивний метод створення рівня та пошуку шляху.

Результати – створена 3d гра з процедурною генерацією рівнів використовуючи рекурсивний метод генерації рівнів та пошуку шляху. Створені моделі героїв та комірку рівня так звану базову, на основі якої і буде побудований рівень. Створені рівні головоломок та рівень з босом. Доданий музичний супровід і користувацький інтерфейс.

ПРОЦЕДУРНА ГЕНЕРАЦІЯ ЛАБІРИНТИ РЕКУРСИВНИЙ МЕТОД UNITY C# 3D ГОЛОВОЛОМКИ РОГАЛИК

Зміст

ВСТУП 6
1 ІНФОРМАЦІЙНИЙ ОГЛЯД.. 8
1.1 Процедурна генерація, прикладі використання, методи................ 8
1.2 Лабіринти: різновиди, категорії класифікації. 14
1.3 Лабіринти: алгоритми генерації та знаходження шляху................ 22
1.4 Постановка задачі... 25
2 ІНФОРМАЦІЙНИЙ ОГЛЯД ПРОГРАМ ДЛЯ ВИРІШЕННЯ ЗАДАЧІ.... 26
2.1 Середовище розробки Unity 26
2.2 Мова програмування C# 27
2.3 Жанр гри її особливості 27
3 ПРОГРАМНА РЕАЛІЗАЦІЯ 29
3.1 Опис алгоритма генерації та опис вигляду програми 29
3.2 Інтерфейс та основні його можливості 31
3.3 Основний контент гри: Моделі персонажів, Генерація рівня, Головоломки, Рівень з босом, Музичний супровід. 32
ВИСНОВКИ... 40
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 41
ДОДАТОК А ЛІСТИНГ МОДУЛІВ ЕКСПЕРТНОЇ СИСТЕМИ 44

ВСТУП

Ігрова індустрія існує вже декілька десятиліть, вона значно еволюціонувала з перших днів комп’ютерів та перших приставок. Дні піксельної графікі та відсталого звуку вже давно позаду, так як ігри стали реалістичнішими як ніколи раніше. Вона настільки розвилась, що кожен рік компанії, які розробляють відеоігри проводять заходи на яких презентують нові ігри та нові технології які будуть в нових проектах. Однією з най популярніших є E3(ELECTRONIC ENTERTAINMENT EXPO), проходить вона у Лос – Анджелесі і в 2019 році її відвідало приблизно 66 тис. людей. Разом з розвитком інформаційних технологій ігри продовжують еволюціонувати забезпечуючи розвагу не тільки дітям, а і дорослим. Так однією із головних інновацій в ігровій індустрії стала інтеграція віртуальної та доповненої реальності в ігри, які в останній час стали дуже швидко розвиватися і використовуватись у різних сферах життя. Наявність тензерних або як їх ще називають RTX ядер у новому поколінні відеокарт забезпечує реалістичне освітлення в іграх [1]. Витрати на розробку також збільшився до прикладу бюджет гри Red Dead Redemption 2 від Rock Stars склав 256 мільйонів доларів. Зараз ігрова індустрія оцінюється в 159 млрд. доларів, до прикладу ще вісім років тому ця цифра була 70.6 млрд, прогнозується, що в 2023 році капіталізація буде складати 200 млрд. це дає змогу зрозуміти що індустрія не стоїть на місці, а і є дуже прибутковою. Індустрія відеоігор надзвичайно велика. Вона більша, ніж індустрії кіно та музики разом узяті, хоча їй не приділяють такої уваги. В світі нараховують приблизно 2.9 млрд гравців. Це більше ніж 26% від населення усього світу [2]. Така кількість гравців забезпечується великою кількістю платформ на які випускають ігри: консолі, персональні комп’ютери, портативні консолі, мобільні телефони тощо і великою різноманітністю жанрів серед яких кожен зможе знайти щось для себе. Одними з най популярніших ігор є ігри з онлайн складовою, що в час

пандемії є гарним способом проведення часу та комунікації з незнайомими людьми. Ігри з процедурною генерацією користуються особливою увагою, бо вони завжди привносять щось нове в гру і роблять кожне проходження унікальним. Саме тому темою роботи була обрана гра з процедурною генерацією рівнів.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Процедурна генерація, прикладі використання, методи

На сьогоднішній день комп’ютерна гра вимагає декількох матеріалів для складання складної та великої сцени, що несе великі витрати на розробку їхнього контенту. Генерація цифрового вмісту – це підмножина алгоритмів, вона є ключовою складовою сучасних відеоігор, що формується набором алгоритмів для створення рівнів, карт, персонажів, зброї тощо, забезпечуючи зменшення витрат на виробництво ігор.

Процедурна генерація – це процес створення контенту автоматично, а не самостійно. Даний напрямок досліджень не є новим у галузі комп’ютерних наук. Процедурна генерація виникла в 1975 році, коли Бенуа Мандельброт визначив фрактальний об'єкт, який описує подібні повторювані або подібні математичні моделі. Ця ідея використовувалась в декількох додатках, таких як створення текстур, міст [3], лісів на основі L-систем, місцевостей [4], відеоігор та інших областей. Зазвичай у відеоіграх цей процес називається динамічною генерацією вмісту.

Потужність генерації дозволяє інтегрувати декілька дослідницьких областей для вирішення проблем. Наприклад, у 2013 році Genevaux et al. [5] визначити новий спосіб формування місцевості на основі гідрології. Їх алгоритм приймає за вхід контур місцевості та деякі річки, задані користувачем. Потім алгоритм візьме вхідну інформацію і почне генерувати повну дренажну річкову мережу, що призведе до повного формування річок, які йдуть від джерел до витоків, з очікуваним коливанням висоти. Після створення річок вони починають працювати на решті з рельєфом, використовуючи різні будівельні блоки та модифікуючи рельєф відповідно до річок, щоб загальний результат карти був топологічно правильним.

Гарним прикладом гри де вся карта гри створюється процедурно є гра Minecraft. Випадковим чином створюється географічні та геологічні об’єкти на мапі, при першому заході в гру світ генерується не до кінця. Вона продовжуватиме генеруватися по мірі подорожування по мапі [6]. Основою генерації є шуми Перліна (рис. 1.1) – це алгоритм за допомогою якого генеруються об'ємні ландшафти. За допомогою генератора створюється мапа шумів (мапа висот).

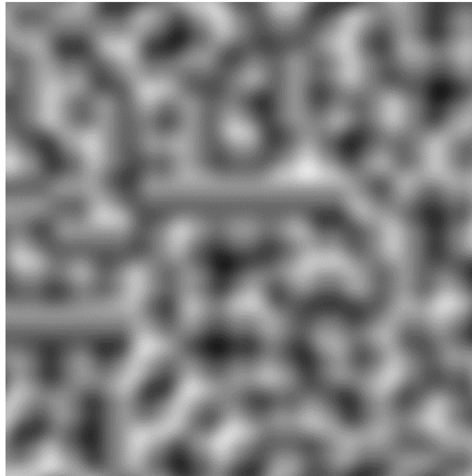

Рис. 1.1. Шум Перліна

Там, де зображення більш світле – місцеві є вищою, а, де темніше – нижчою. Але отримана карта є не істотно обривчастою, через те що великі значення можуть бути поряд з малими. Цей недолік виправляється за допомогою інтерполяції – знаходять проміжні значення, після чого генеруються гори та низини, додаються елементи по типу дерев, річок тощо. Небо на відміну від ландшафту є однаковим і ніколи не змінюється [7]. В залежності від того в якій локації (біомі) ви знаходитесь може йти: дощ, сніг або нічого як в пустелі. В залежності від зерна (сіда) – значення, яке являє собою набір символів і являється основою при генерації кожного світу [8]; буде залежати настання першого дощу, на відміну від першої дати, дата наступної зміни погоди є сталою і складає проміжок 0,5 – 7,5 днів. Від так званого зерна залежить не тільки погода. А і згенерований світ, якщо для

генерації наступної карти ввести ті самі символи, то згенерований світ буде ідентичним до попередньо згенерованого світу з тим самим набором символів [9]. Згенерований світ можна побачити на рисунку 1.2.

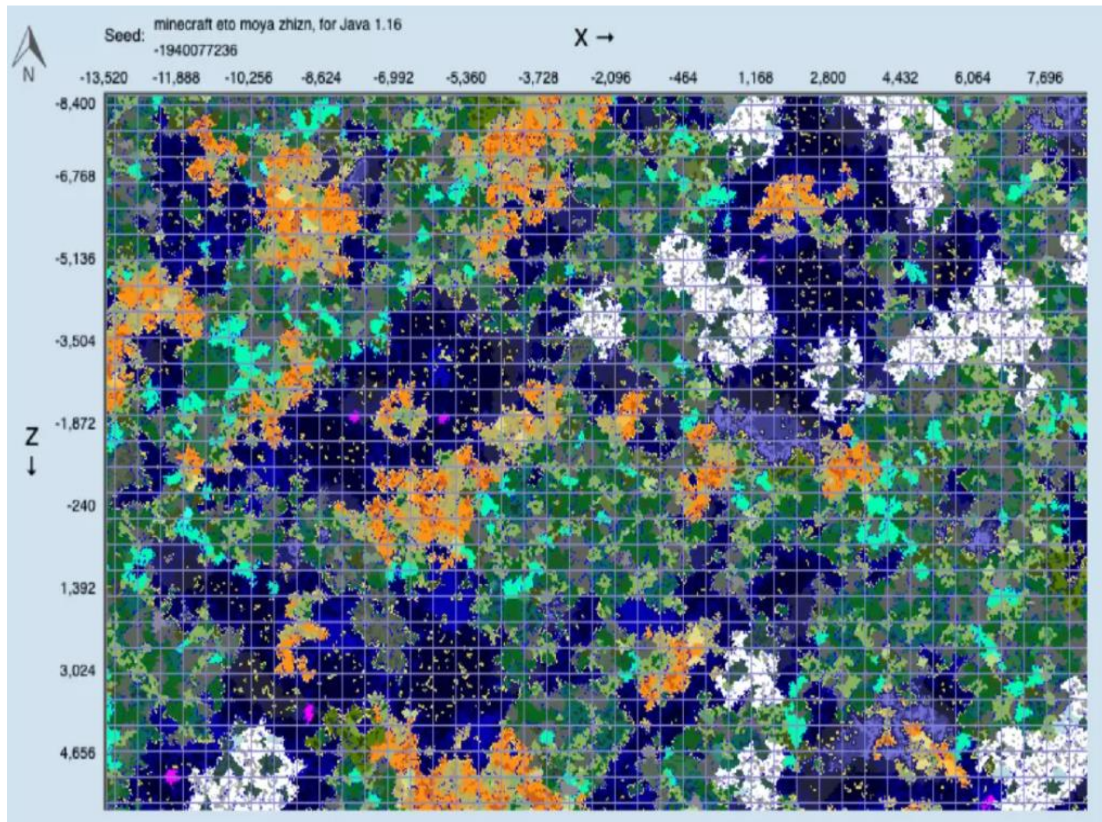

Рис. 1.2. Вигляд згенерованого світу в Minecraft зверху

Одним із головних критеріїв до згенерованого контенту є те що він не повинен відрізнятися від контенту створеного своїми руками. Однією з найпоширенішою задач з якою справляються алгоритми це створення рівнів, кімнат. За допомогою алгоритмів можна не лише створювати об'єкти, а і задавати їх властивості та поведінку, розташування, випадкові події, музику тощо. Дуже важливою складовою процедурної генерації є в іграх те, де користувач повинен не один раз з початку починати гру, щоб процес проходження гри не став монотонним, було вирішено кожний раз генерувати новий рівень: змінювати розташування кімнат, ворогів, скарбів, ігрових об’єктів з якими взаємодіє користувач. Існують різні методи для генерації.

Метод двійкового розбиття простору

Він базується на так званих BSP деревах (дерева двійкового розбиття простору). Вхідними даними є розмір рівня та мінімальний розмір кімнати. Уся площина береться як один лист BSP дерева і він випадково ділиться на дві частини, які стануть листами у наступній ітерації. Ці дії виконуються рекурсивно допоки розмір листа не стане менше заданого мінімального порога [10]. Цей алгоритм зображений на малюнку 1.3.

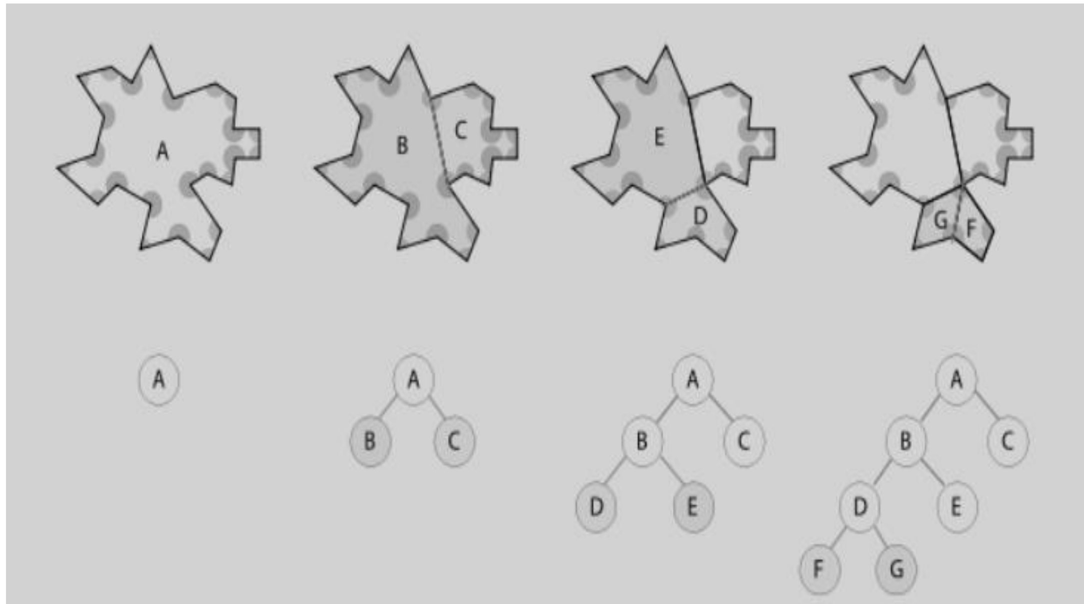

Рис. 1.3. Схема алгоритма двійкового розбиття

Метод «Ходи п’яниці»

Метод «Хода п’яниці» – один із різновидів методу випадкової ходьби, отримав таку назву за хаотичний візерунок, який утворюється в результаті роботи даного алгоритму. Основа алгоритму – це переміщення курсора, який зафарбовує області кімнат рухаючись в випадковому направленні. Для покращення візерунка рівня та їх більшої натуральності, для кожного можливого напрямку курсора виставляють вагові коефіцієнти для того, щоб він зміщувався до центра, а не до кордонів рівня. Також на наступній ітерації виставляють більші вагові коефіцієнти для попереднього напрямку, таким чином між кімнатами з’являються більш вираженні кордони (рис. 1.4). Даний алгоритм гарно підходить для створення рівнів в печерах або джунглях [11].

Рис. 1.4. Приклад роботи алгоритма «Ходи п’яниці»

Метод Клітинних Автоматів

Клітинні Автомати – масив клітинок які створюють періодичну решітку з заданими правилами переходу, які визначають стан клітинки в наступний момент часу, через стан клітинки, яка знаходиться на відстані не більше деякого в даний момент часу. Спочатку кожна точка об’являється стіною або підлогою, що називається стандартним станом автомата. Кожна клітинка має вісім з нею суміжних сусідів клітинок, точка залишається стіною, якщо вона була нею і має чотири сусіда точок, стає підлогою, якщо була стіною але не має п’ять сусідів стін. Через декілька ітерацій формуються кімнати. У результаті виконання алгоритму можуть утворитися відділені області, які потрібно перевірити власноруч або за допомогою іншого алгоритму. У результаті отримаємо дуже істотній ландшафт, схожий на вигляд печери з гори [12]. Алгоритм зображений на малюнку 1.5.

Рис. 1.5. Приклад роботи алгоритму Клітинних Автоматів

Також гарним прикладом використання процедурної генерації в іграх є “No man`s sky”. У цій грі, окрім мапи усі події, вороги, зброя генерується процедурно. Ігри з генерацією користуються великою популярністю серед гравців і дуже часто отримують вищі нагороди в сфері відеоігор та підтримку гравців. Гра Star Citizen отримала 250 мільйонів доларів за що отримала звання найдорожчої гри в історії, розробка гри ще продовжується і кожен день її бюджет стає все більшим і більшим [13]. Загалом алгоритми генерації дуже розповсюджені використовуються не тільки як основна «фіча» в іграх. У грі Crusader Kings 2, генерація використовується для побудови складних родинних зв’язків та інтриг в старовинних монарших сім’ях середньовіччя завдяки генерації особливостей персонажів. У грі Shadow of Mordor генеруються взаємозв’язки ворогів між собою, що робить їх унікальними і схожими на людей. Нещодавно проект Georgia Tech зміг дозволити генерувати прості інтерактивні сюжетні лінії. Відома нейронна мережа ANGELINA, створена для того щоб створювати цілі ігри, не тільки окремі елементи, а механіки використовуючи мовні команди [14].

Вигляд вищезазначених ігор показано на рисунку 1.6.

а б

Рис. 1.6. Вигляд ігор з процедурною генерацією: Minecraft (а), Shadow of Mordor (б)

1.2 Лабіринти: різновиди, категорії класифікації

За основу для рівнів гри були обрані лабіринти. Лабіринт – це деяка структура, яка складається із заплутаних частинок, які ведуть до фінішу або у глухий кут. Загалом характеризація лабіринтів відбувається за сімома класифікаціями: топологією, теселяцією, маршрутизацією, текстуризацією, пріоритетностю, розміром, гіперрозміром.

Топологія: цей клас описує геометрію простору лабіринту, існують такі типи:

Звичайний тип – це звичайний лабіринт в евклідовому просторі. Евклідовий простір – простір, властивості якого описується аксіомами евклідової геометрії. Також в даному контексті розуміється, що мова йде про трьох мірний простір. Іноді для опису топології використовують поняття “Plainiar”, він використовується для опису лабіринтів з незвичайними з’єднаннями країв.

З’єднана ліва та права та верхня і нижня сторона.

Теселяція: використовується для класифікації геометрії окремих комірок із яких складається лабіринт. Найбільш цікавим видами є:

Ортогональний – в якому він має вигляд прямокутної сітки, комірки мають проходи і претинаються під кутом в 90 градусів. Іноді зустрічається назва гамма-лабіринт(рис 1.7).

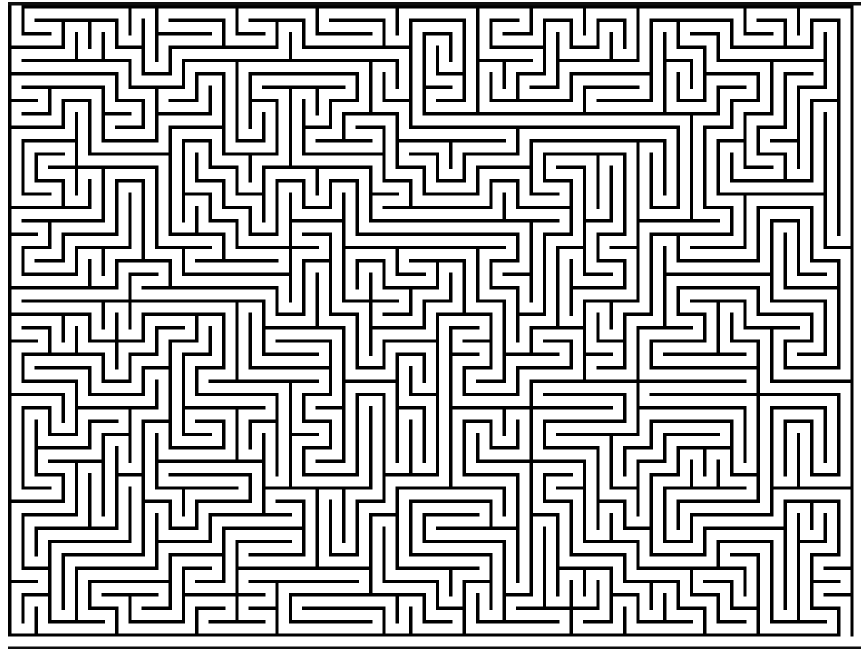

Рис. 1.7. Вигляд лабіринту з ортогональною тесиляцією

Дельта – складається зі з’єднаних трикутників, кожна комірка має до трьох з’єднань(рис. 1.8).

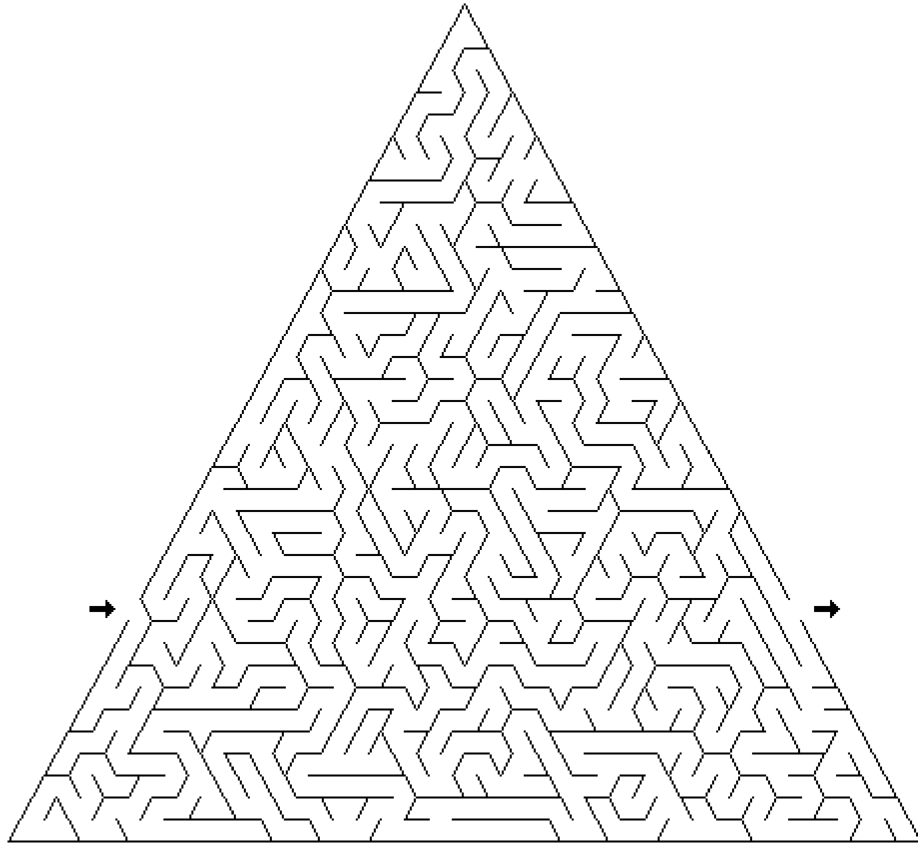

Рис. 1.8. Вигляд лабіринту з дельта тесиляцією

Сігма - складаються з шестикутників і мають до шести проходів.

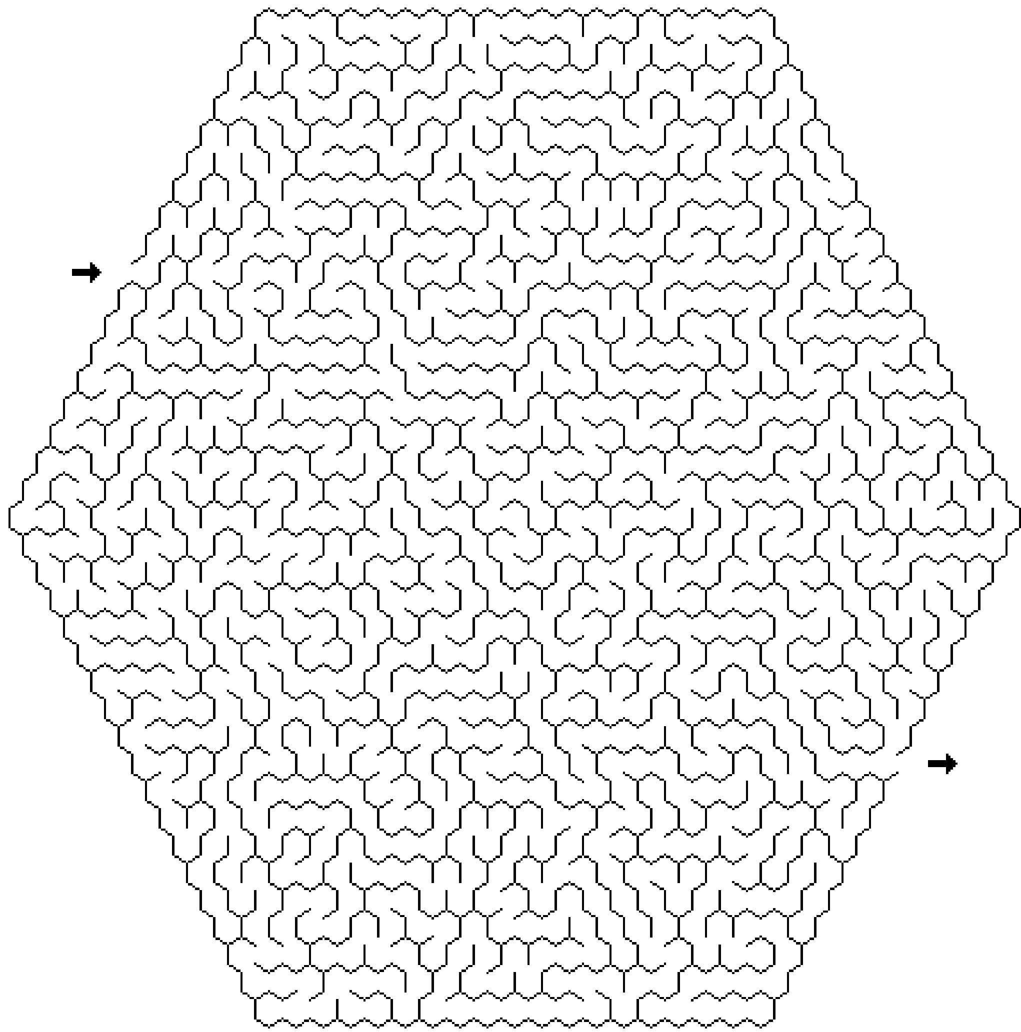

Рис. 1.9. Вигляд лабіринта з сігма тесиляцією

Тета – лабіринти з такою теселяцією характеризуються тим, що складаються з концентричних кіл проходів, де початок або кінець знаходяться по центру, а другий - на зовнішньому краю. Комірки мають до чотирьох проходів але є можливість, що їх буде більше завдяки кількості комірок на зовнішніх кільцях (рис 1.10).

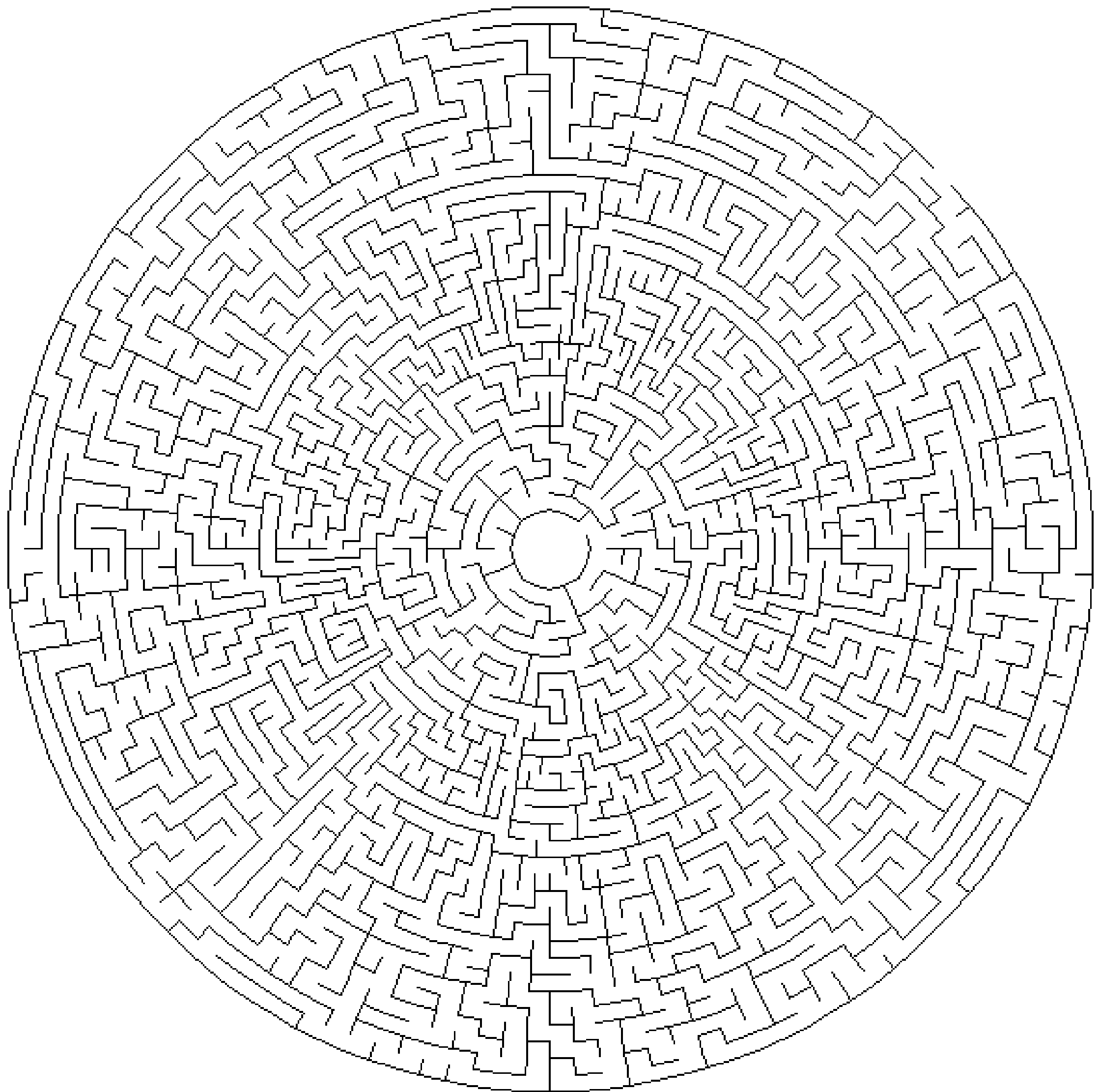

Рис. 1.10. Вигляд лабіринту з тета тесиляцією

Омега – має відношення майже до будь-якого лабіринту зі сталою неортогональною теселяцією. Усі вище перераховані лабіринти відносяться до даного типу лабіринтів, як і безліч інших лабіринтів, наприклад ті, що складаються з прямокутних трикутників. Приклади таких лабіринтів надані вище.

Фрактальний – лабіринт який складається з менших лабіринтів. У кожній комірці такого лабіринту знаходяться інші лабіринти. Рекурсивний лабіринт даного типу буде мати нескінченну кількість однакових лабіринтів ,

які копіюють одне одного, тим самим створюючи нескінченний лабіринт (рис. 1.11).

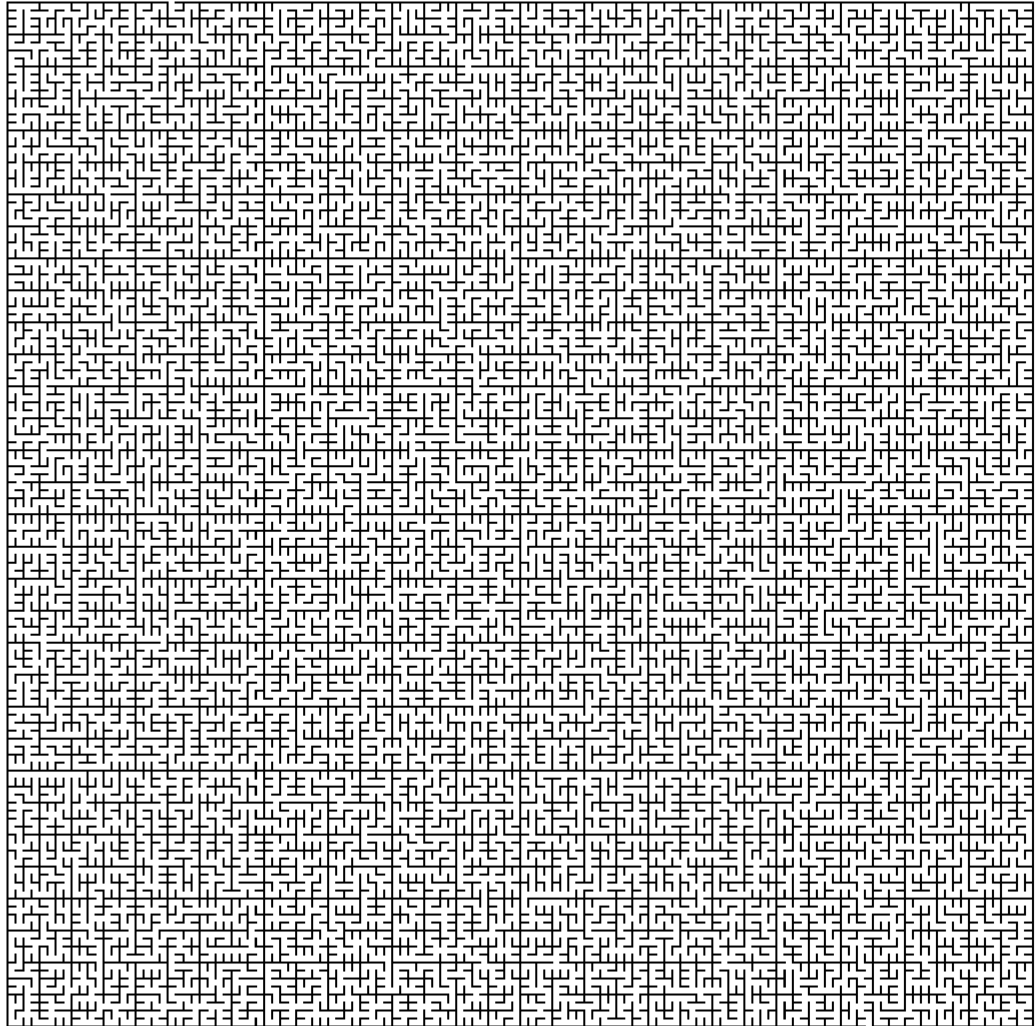

Рис. 1.11. Вигляд лабіринту з тета тесиляцією

Маршрутизація: даний аспект пов’язаний з геометрією лабіринту приклади якої були наведені вище.

Ідеальний – це лабіринт у якому не має петель і недосяжних областей. Лабіринт має лише одне рішення, гарним прикладом такого лабіринту є ортогональний лабіринт.

Розріджений – це лабіринт який в якому немає проходу через кожну комірку. через це виникають області через , які неможливо пройти. При додаванні стін використовуючи дану концепції. Можна отримати нерівномірний лабіринт з дуже великими проходами і кімнатами (рис 1.12).

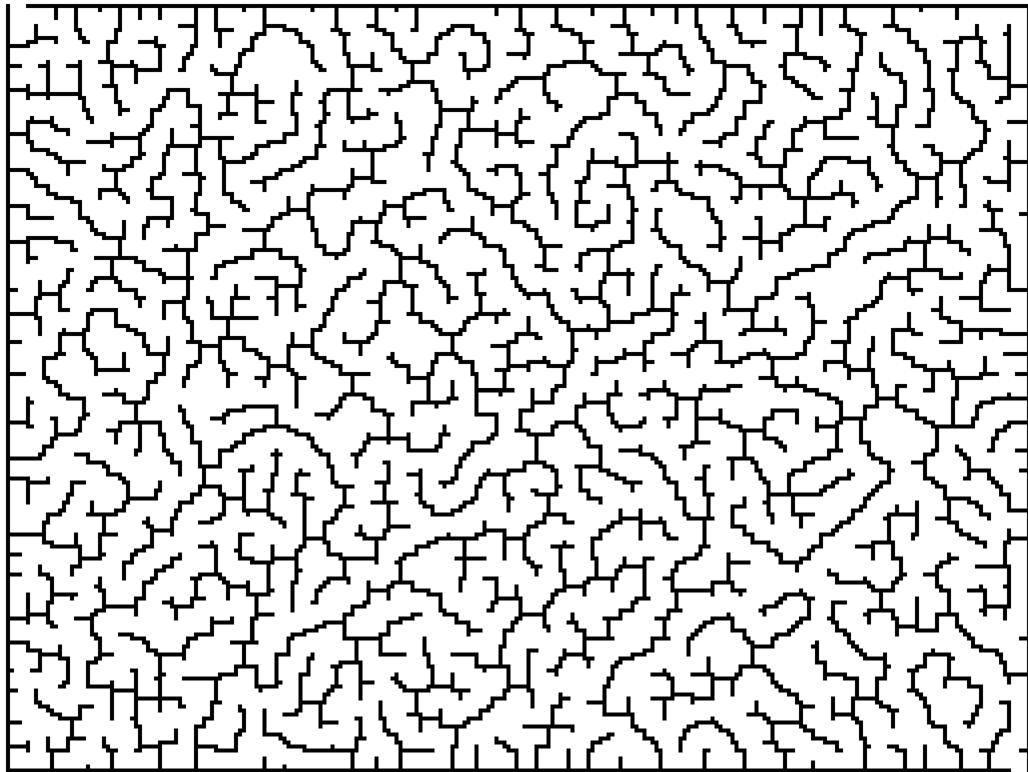

Рис. 1.12. Вигляд лабіринту з тета тесиляцією

Частково плетеним лабіринтом називають лабіринт у якому зустрічаються петлі і глухі кути.

Текстура: ця класифікація описує оформлення проходів при різній маршрутизації і геометрії.

Однорідність – алгоритм який генерує усі можливі лабіринти з рівною ймовірністю (рис. 1.13).

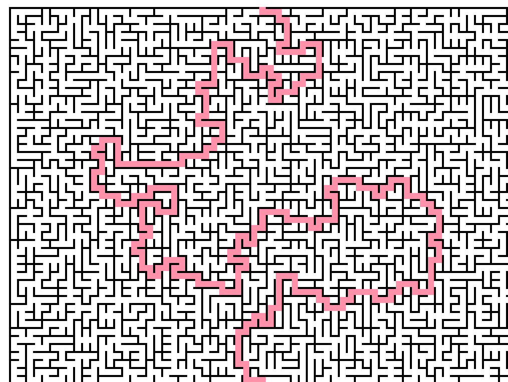

Рис. 1.13. Вигляд однорідного лабіринту

Симетрія – лабіринти, які мають симетричні проходи , наприклад оберти навколо центра (рис. 1.14).

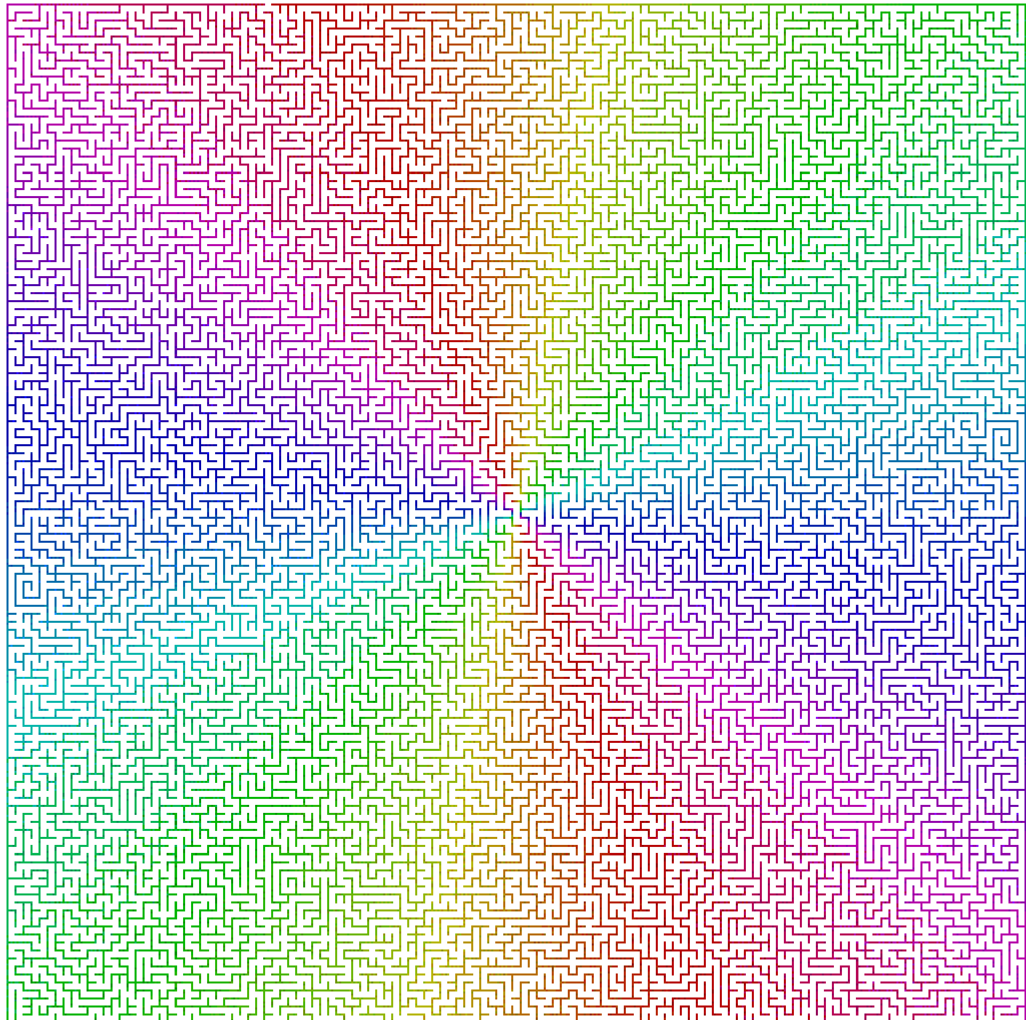

Рис. 1.14. Приклад симетрії у лабіринті

Текучість – це поняття вказує на те, що при створенні лабіринта алгоритм буде шукати і працювати з сусідніми комірками і так допоки алгоритм не згенерує лабіринт, тим самим він буде моделювати річку з її течією саме тому дана характеристика і отримала свою назву.

Пріоритет: цю класифікацію можна поділити на дві складові якими вона і характеризується, а саме: створення стін і проходів між ними тим самим утворюючи комірки лабіринта.

Деякі алгоритми можуть одночасно додавати стіни і вирізати проходи. Шаблоном такого лабіринта називають зображення, яке за найменшу кількість кроків стане лабіринтом. Лабіринт вигляду пересічних спіралей легше усього реалізувати за допомогою шаблона.

1.3 Лабіринти: алгоритми генерації та знаходження шляху

Існує дуже багато алгоритмів для створення лабіринтів. Одними з найпопулярніших є алгоритми для створення так званих ідеальних лабіринтів.

Алгоритм Краскала – створює мінімальне сполучне дерево. Він вирізає сегменти проходів випадковим чином і як результат створюється ідеальний лабіринт. Для його роботи використовується обсяг пам'яті еквівалентний до розміру лабіринту. Ми ідентифікуємо кожну клітинку за унікальним ідентифікатором, а потім вирізаємо всі краї у довільному порядку. Якщо комірки з обох боків мають різний ідентифікатор очистіть , то видаляємо стіну та призначаємо усім клітинкам з одного боку однаковий ідентифікатор, такий що мають клітинки з іншого боку. Якщо кімнати по обидва боки стіни мають однакові ідентифікатори, між ними вже є шлях, тому стіну можна залишити, щоб запобігти появі петель. Даний алгоритм має погану текучість (рис 1.15).

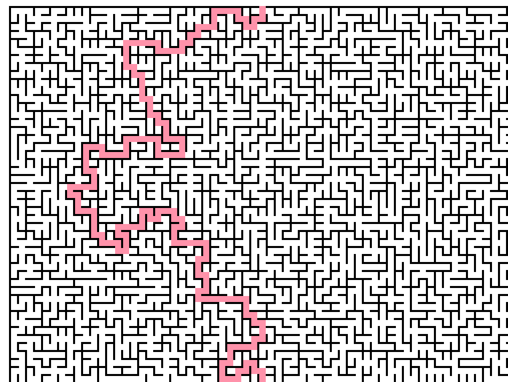

Рис. 1.15. Результат алгоритму Краскала

Алгоритм Пріма – цей алгоритм створює мінімальне зв'язне дерево, обробляюче випадкову вагу ребер. Як і вище наведеному алгоритмі пам'ять яка потрібна для виконання алгоритма пропорційна розміру згенерованого

лабіринту. Не має значення з якої вершини починається виконання, лабіринт обирається ребро з найменшою вагою з'єднуючим лабіринт до точки, яка ще не знаходиться у ньому, після чого вона приєднується до нього. Виконання закінчується після того яка закінчуються ребра, які відповідають вище приведеним вимогам (1.16).

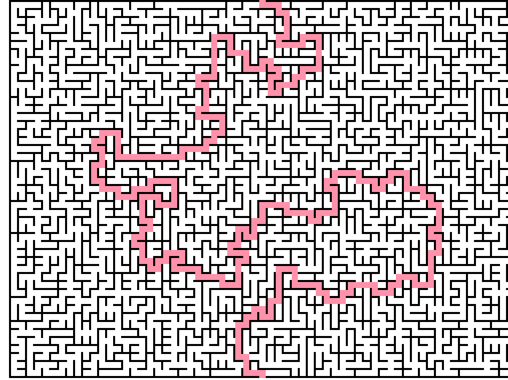

Рис. 1.16. Результат алгоритму Пірса

Алгоритм «Вирощування лісу» – він починається зі всіх комірок, випадковим чином обираються клітинки, які заносяться до списку нових. Потім декілька комірок переходять зі списку нових до списку активних. Обираємо клітинку зі списку активних і робимо в ній прохід до сусідньої нествореної клітинки із списку нових, додаємо її до списку активних і об'єднуємо множини двох клітинок (рис. 1.17). Виконуємо ці дії до кінця [15].

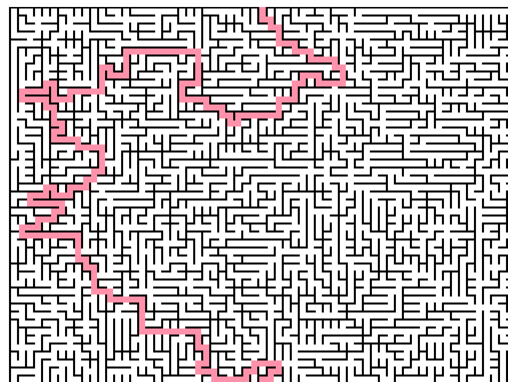

Рис. 1.17. Результат алгоритму «Вирощування лісу»

Іншою важливою частиною лабіринтів є їх рішення.

Найпопулярнішим і найлегшим є алгоритм за яким вам потрібно йти уздовж стіни. По досягненню розходження завжди повертаєте на право або навпаки на ліво. В реальному житті потрібно покласти руку на праву або ліву стіну і виконувати вище зазначені дії. Але цей метод не є ідеальним бо рішення зазвичай не є оптимальним (рис. 1.18).

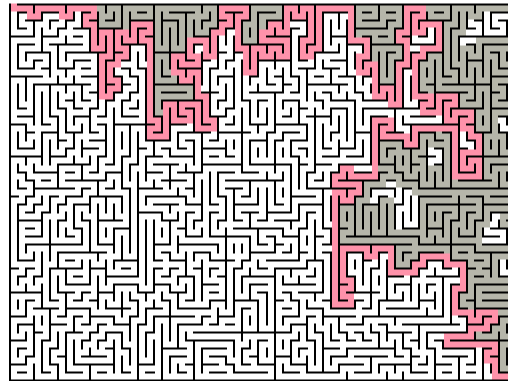

Рис. 1.18. Результат алгоритму знаходження шляху слідуванням уздовж стіни

Алгоритм Пледжа – модифікований алгоритм слідування уздовж стіни з можливістю перестрибувати між «островами» для тих лабіринтів де неможливо знайти рішення методом слідування уздовж стіни (рис. 1.19).

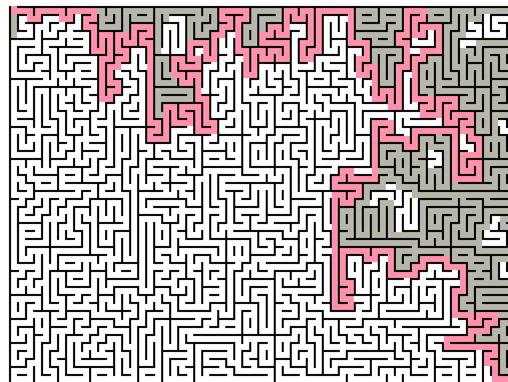

Рис. 1.19. Результат алгоритму знаходження шляху алгоритмом Пледжа

Метод Ланцюгів – цей метод розбиває лабіринт на менші лабіринти і в кожному з них знаходить шлях (рис. 1.20). Для виконання цього алгоритму головне знати початок і кінець лабіринту. Цей алгоритм є одним з найкращих

за оптимальністю і часом виконання, але якщо точка виходу з лабіринту не відомо, то його не можливо використати [16].

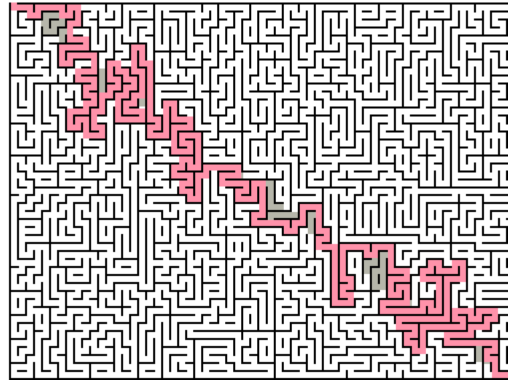

Рис. 1.20. Результат алгоритму знаходження шляху методом Ланцюгів

1.4 Постановка задачі

На основі отриманої інформації я вирішив за допомогою Рекурсивного метода створити гру з процедурною генерацією рівнів, використовуючи середовище для створення ігор Unity, яка дозволить:

- Створити користувацький інтерфейс та додати музичний супровід для гравця;
- Реалізувати процедурну генерацію лабіринтів за допомогою рекурсивного метода;
- Зробити пересування героя по лабіринту за допомогою клавіатури;
- Реалізувати пошук найкоротшого шляху для виходу з лабіринту;
- Реалізувати рівні з головоломками;
- Реалізувати поведінку для боса на деяких рівнях;
- Імпортувати моделі персонажів та створити комірку лабіринту з якої він буде генеруватися.

Детальніше про методи вирішення і реалізацію буде написано у наступному розділі.

2 ІНФОРМАЦІЙНИЙ ОГЛЯД ПРОГРАМ ДЛЯ ВИРІШЕННЯ ЗАДАЧІ

2.1 Середовище розробки Unity

Серед всіх можливих додатків для вирішення вище назвоної задачі була обрана платформа Unity. Unity – це міжплатформене середовище для створення 2d, 3d ігор та VR, AR додатків та ігор. Вона є однією з найпопулярніших платформ, яка допомагає розробникам вирішувати купу завдань. Вона була розроблена американською компанією Unity Technologies. Випуск Unity відбувся в 2005 році і з того часу йде постійний розвиток [17]. До головних переваг Unity, які дуже сильно вирізняють її на фоні конкурентів є:

- Можливість запустити свою програму на будь-якій із існуючих платформ і для цього не треба все переписувати знову завдяки мові програмування C#;
- Можливість створювати ігри з гарною графікою завдяки вбудованим функціям, які дозволяють без проблем створювати вражаючі візуальні ефекти;
- Простота редактора. Під простотою розуміється те, що в інтерфейсі програми дуже легко розібратися;
- Наявність гарної офіційної документації;
- Чудово підходить для візуалізації 3d та 2d сцен;
- Наявність гарної спільноти;
- Наявність Asset Store, де можна знайти готові моделі персонажів, текстури, скрипти та таке інше;
- Можливість працювати в команді завдяки Unity Teams;
- Можливість створити гру без коду;

Але серед усіх переваг unity має свої недоліки, а саме:

- Неможливість завантажити шаблон, все потрібно робити з нуля;

− Наявність ліцензій, цей факт дуже сильно ускладнює ваше життя якщо ви хочете монетизувати ваш продукт;

− Складність в редагуванні ландшафту гри, точніше відсутність необхідних інструментів, які вже є в інших редакторах;

− Наявність застарілої середи для написання програмного коду Mono, але цей недолік дуже швидко можливо виправити встановивши Visual Studio та обравши його середою за замовчуванням у налаштуваннях [18].

Зважаючи на усі недоліки та переваги, в роботі обрана саме ця платформа, бо мова програмування C# для більшості задач є зручною. Та і легкість у використанні Unity, про яку багато говорять, привертає до себе увагу.

2.2 Мова програмування C#

C # - об'єктно-орієнтована мова програмування зі статичною типізацією. Розроблена в 1998-2001 роках групою інженерів компанії Microsoft під керівництвом Андерса Хейлсберг і Скотта Вільтаумота як мову розробки додатків для платформи Microsoft .NET Framework. C# повністю задовольняє вимоги до проєкта. Є дуже гарною для розробки програм для операційної системі Windows. Також синтаксис мові дуже схожий на синтаксис Java, що в свою чергу є плюсом, але як на мене ті речі, які відрізняються від Java дуже гарно виділяють C#, бо робить його більш гнучким та і підтримка мови більш активна і оновлення виходять дуже часто. Сама мова швидко розвивається, в інтернеті є багато статей, які допомагають вирішувати питання, які виникають при написанні [19].

2.3 Жанр гри її особливості

Створена гра відносить до жару ігор, який називається Roughlike або як його ще називають ”Рогалик”. Характерними особливостями класичного roguelike є рівні які генеруються випадковим чином, пошаговість і

незворотність смерті героя - в разі його загибелі гравець не може завантажити гру і повинен почати її з початку. Даний жанр зародився ще сорок років тому і зараз є одним з найпопулярніших. Найвідомішими іграми цього жанру є: Hades, Binding of Isaac [20]. Її можна побачити на малюнку 2.21

Рис. 2.21. Гра Binding of Isaac

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

Для програмної реалізації як вже було зазначено вище використовувалось середовище Unity та мова програмування C#, яке є високоефективним інструментом для реалізації 3d ігор з процедурною генерацією.

Програмна реалізація містить в собі:

- Зробити пересування героя по лабіринту за допомогою клавіатури;
- Реалізувати пошук найкоротшого шляху для виходу з лабіринту;
- Реалізувати рівні з головоломками;
- Реалізувати поведінку для боса на деяких рівнях;
- Імпортувати моделі персонажів та створити комірку лабіринту з якої він буде генеруватися.

Результатом проведеної роботи є програмний продукт з назвою “Dungeon”, опис якого подано у наступних пунктах.

3.1 Опис алгоритма генерації та опис вигляду програми

Для реалізації генерації був обраний рекурсивний метод генерації. Алгоритм метода полягає:

1. Задаємо розмір лабіринту.
2. З префабу генеруємо лабіринт без проходів.
3. Починаємо з початкової клітинки.
4. Рухаємося у випадковому напрямку до глухого кута після чого відходимо назад до моменту, коли знову можна буде видалити стіну і т. д. до стартової клітинки
5. Для кожної клітинки задаємо дистанцію від фінішу.
6. У найвіддаленішій клітинці від старту видаляємо одну стінку для виходу вона і буде фінальною.

7. Для створення підказки: починаючи від фінішу шукаємо клітинку у якої дистанція до старту буде на один менша ніж у попередньої.

Алгоритм за яким відбувається відображення та знаходження найкоротшого шляху від фінішу до фіналу також називається рекурсивним і він наданий у пункті 7 вище описаного алгоритму. Перед розробкою програмного продукту була розроблена use-case діаграма та структури програми. Ці діаграми зображені на малюнках 3.22 та 3.23.

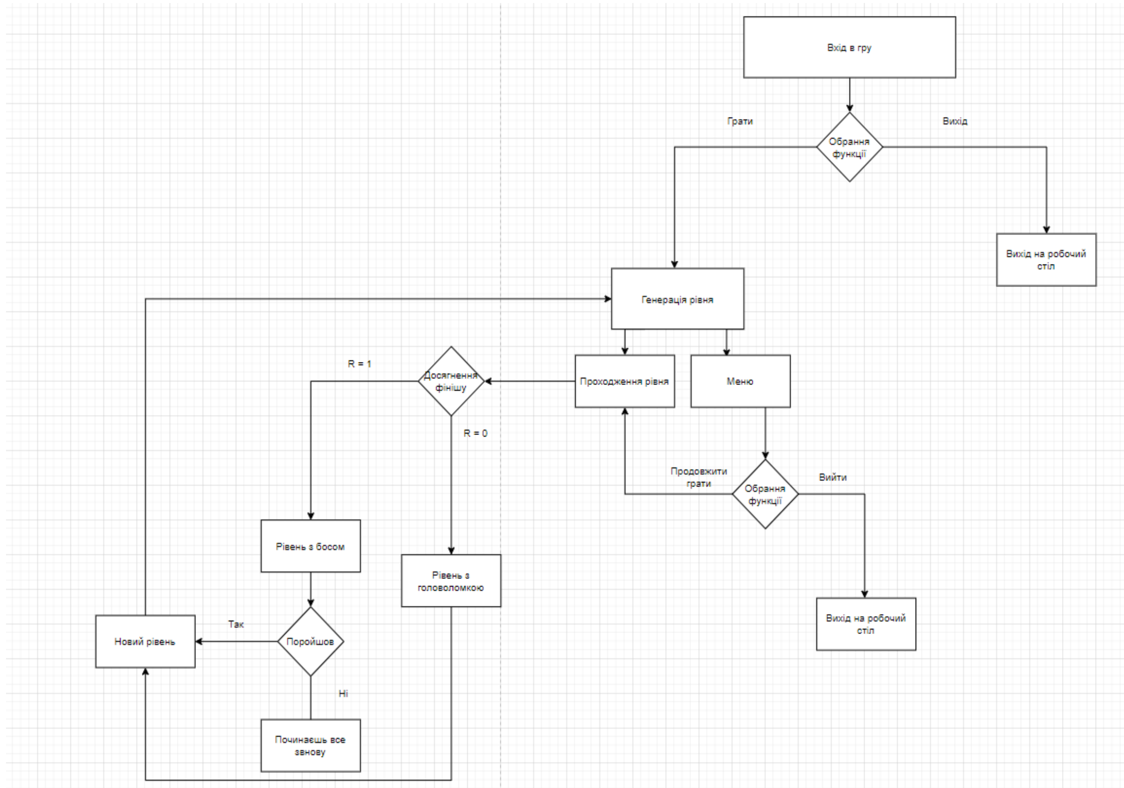

Рис. 3.22. Структурна діаграма проєкту

Та use-case діаграма яка включає в себе агента в ролі якого виступає користувач тобто гравець (рис. 3.23).

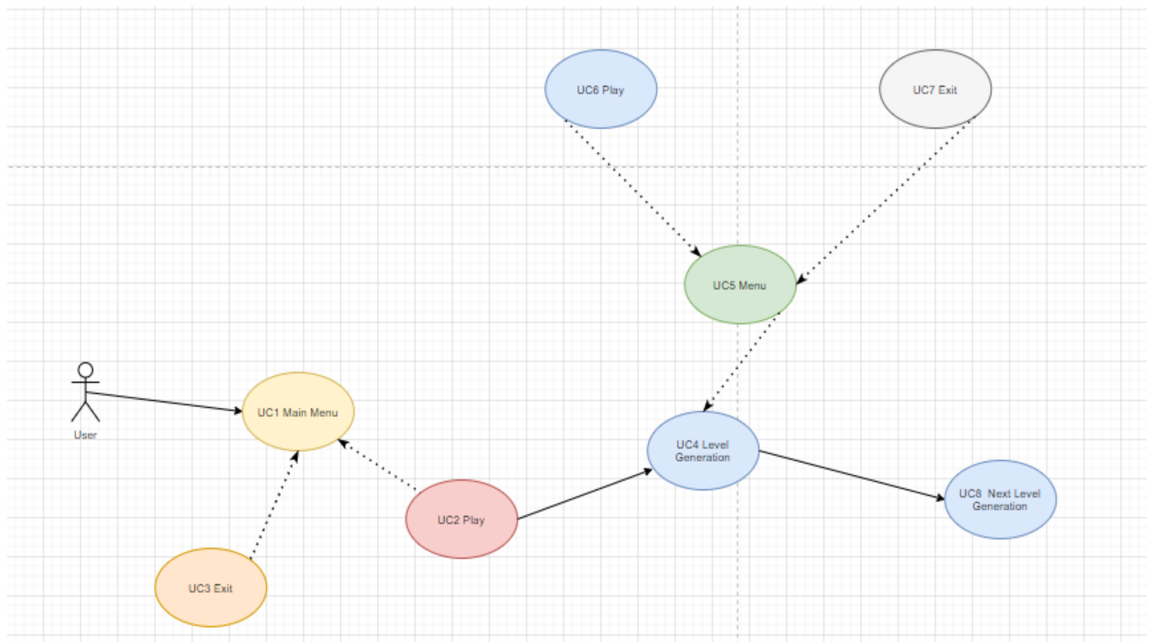

Рис. 3.23. Use-case діаграма

3.2 Інтерфейс та основні його можливості

Загальний вигляд вікна системи “Dungeon” представлений на рисунку 3..

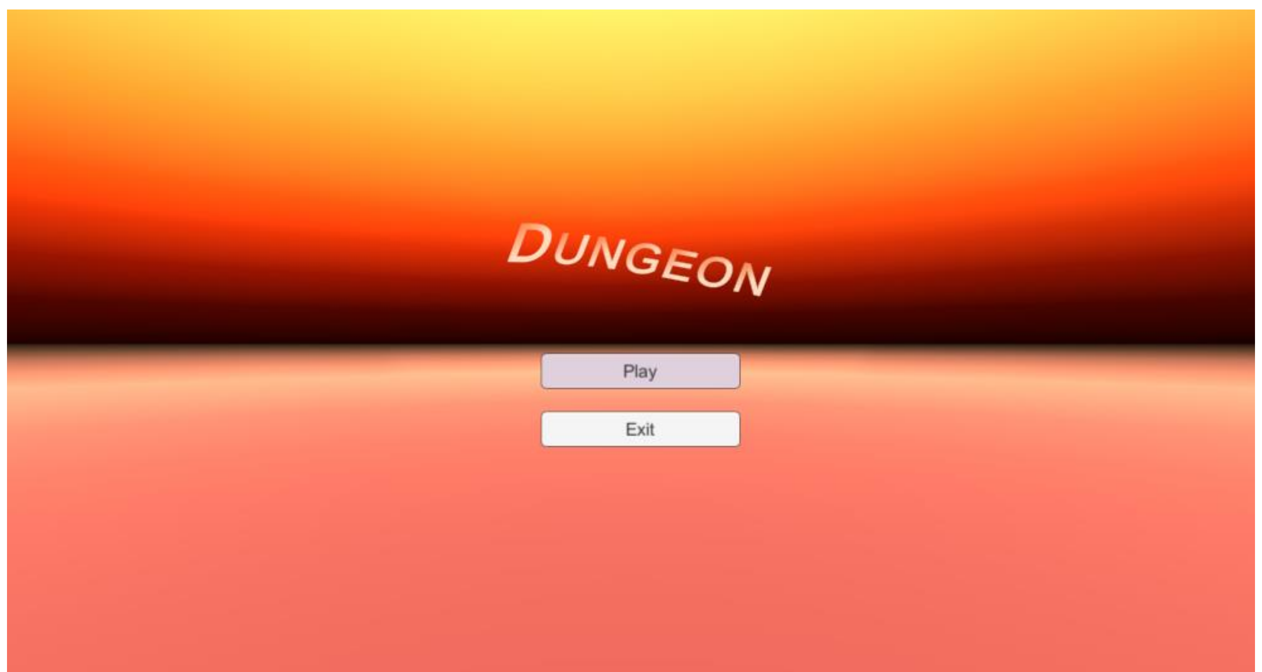

Рис. 3.24. Загальний вигляд головного меню програми

Основні елементи головного меню програми – дві кнопки та анімована назва, де кнопка “Play” відповідає за початок гри, “Exit” для виходу з гри .

Рис. 3.25. Вигляд меню паузи гри

Основні елементи меню паузи програми – дві кнопки та назва, де кнопка “Space” відповідає за відновлення гри, “Esc” для виходу з гри. Для виклику даного меню треба натиснути кнопку “Escape”. Вони зображені на малюнку 3.25

3.3 Основний контент гри: Моделі персонажів, Генерація рівня, Головоломки, Рівень з босом, Музичний супровід.

В даній роботі присутні дві моделі героїв, та ще були створені моделі для головоломок але вони являють собою звичайний примітив куба. Дві моделі це: модель головного героя та боса (рис. 3.26 та 3.27). Для створення моделей персонажів використовувались засоби для створення об’єктів самої Unity та програма Blender. Анімування персонажів виконувалось в Unity, а самі анімації були взяті з сайта mixamo.com

Рис. 3.26. Модель головного героя

Рис. 3.27. Модель боса

Для анімування персонажів були створені аніматори в які були додані анімації, які повинні виконувати герої, а саме: анімацію бездії, бігу, стрибку, удару. Вони мають такий вигляд рис. 3.28:

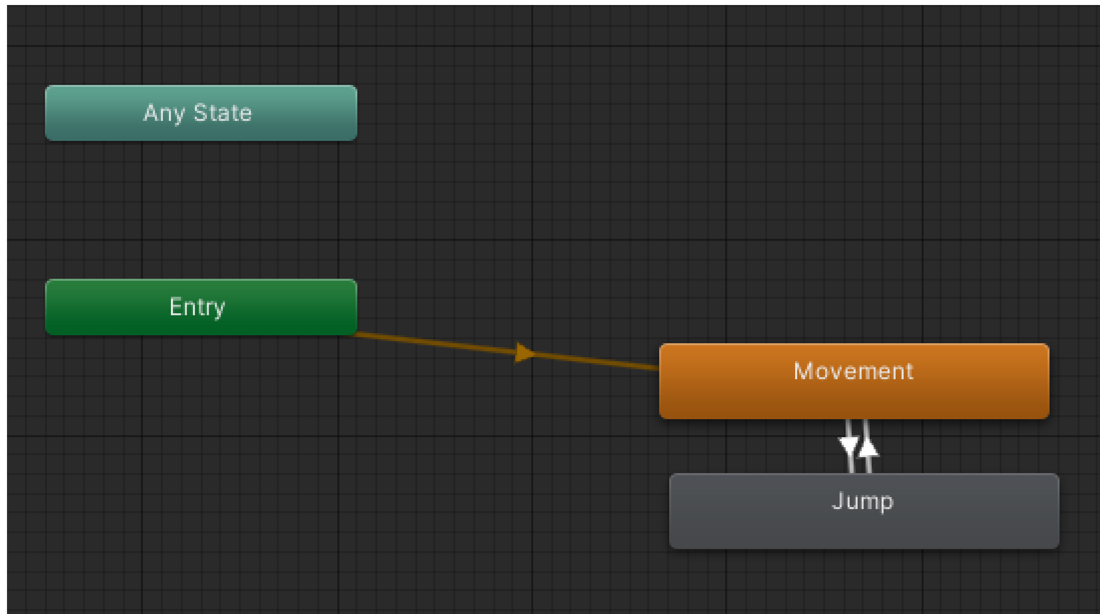

Рис. 3.28. Аніматор для головного героя

Цей аніматор включає в себе три стани, це початковий, стан руху, який містить в собі анімації ходьби, бігу та бездії, та стан для стрибку рис. 3.29.

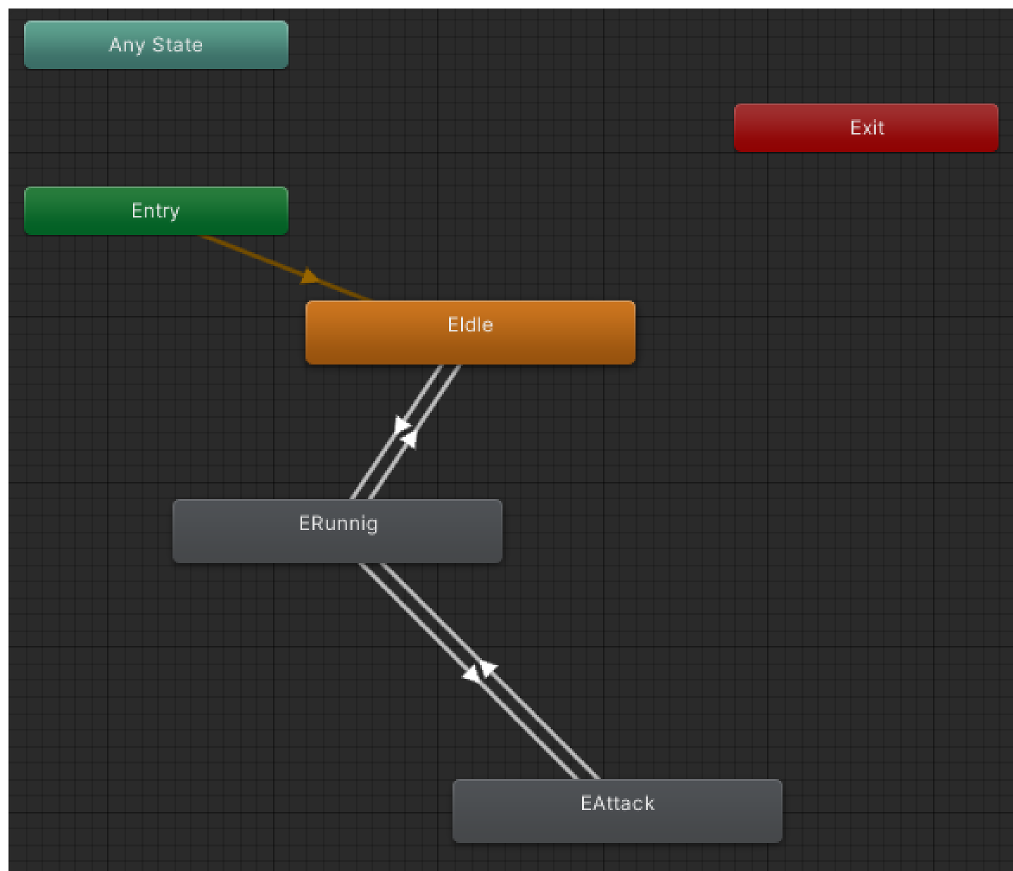

Рис. 3.29. Аніматор для боса

Оскільки бос має більше анімацій ніж головний герой, то аніматор складається з п’яти станів. Та має анімації: анімації ходьби, бігу та бездії та удару. Правила поведінки для боса прописані в скрипті Enemy.cs в лістингу програми. А для зміни анімації героя за рахунок руху героя та натисканню тих чи інших клавіш(файли CameraFollow.cs, MovementCharacteristics.cs, ThirdPersonMovement.cs). Також для створення штучного інтелекта для боса була використана вбудована бібліотека UnityEngine.AI. Та за допомогою навігатора в Unity створена площина де бос може переміщуватись (рис. 3.30).

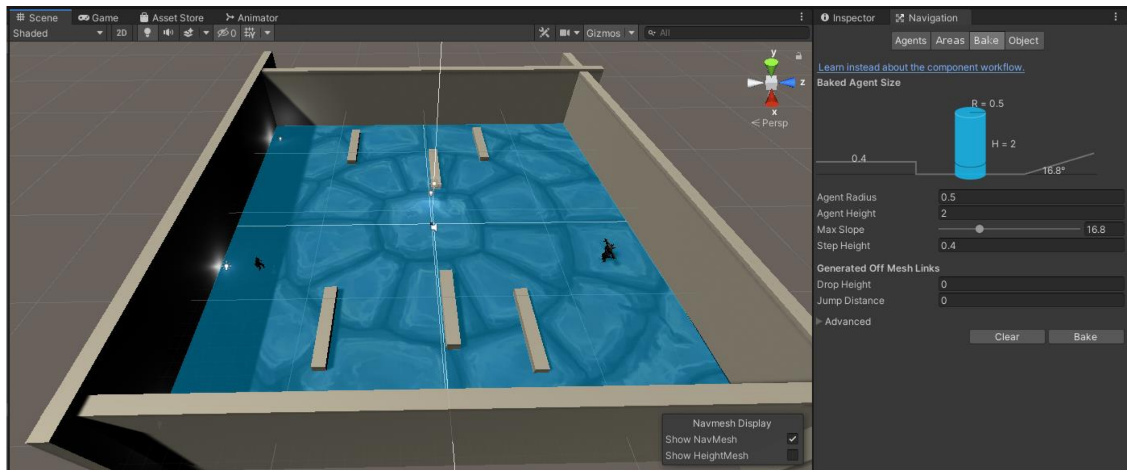

Рис. 3.30. Вигляд навігатора та сцени переміщення для противника

Для реалізації розробленого алгоритму використані такі відомі прийоми програмування – робота з масивами, циклами, обробка подій тощо. Алгоритм генерації описаний у файлі MazeGenerator.cs. а пошуку шляху у HintRenderer.cs, для їх розміщення все описано у файлі MazeSpawner.cs. Їх можна побачити у лістингу програми. Результат на рис. 3.31.

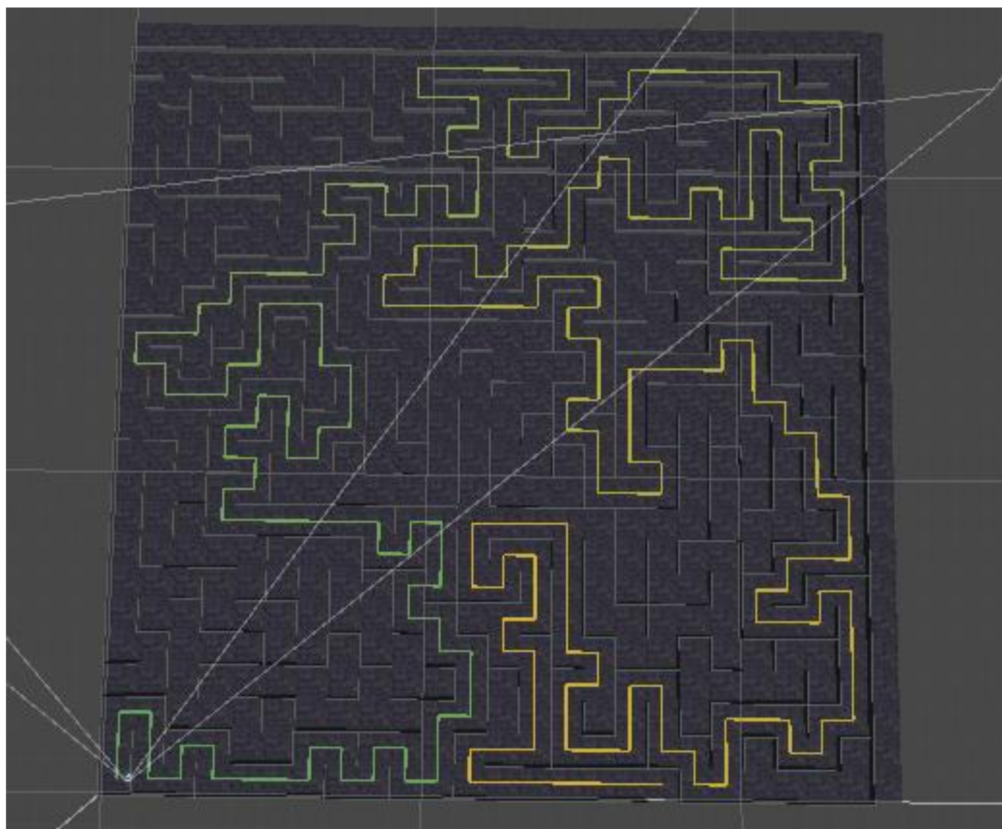

Рис. 3.31. Вигляд згенерованого лабіринта

Загалом гравець бачить перед собою таку картину(рис. 3.32).

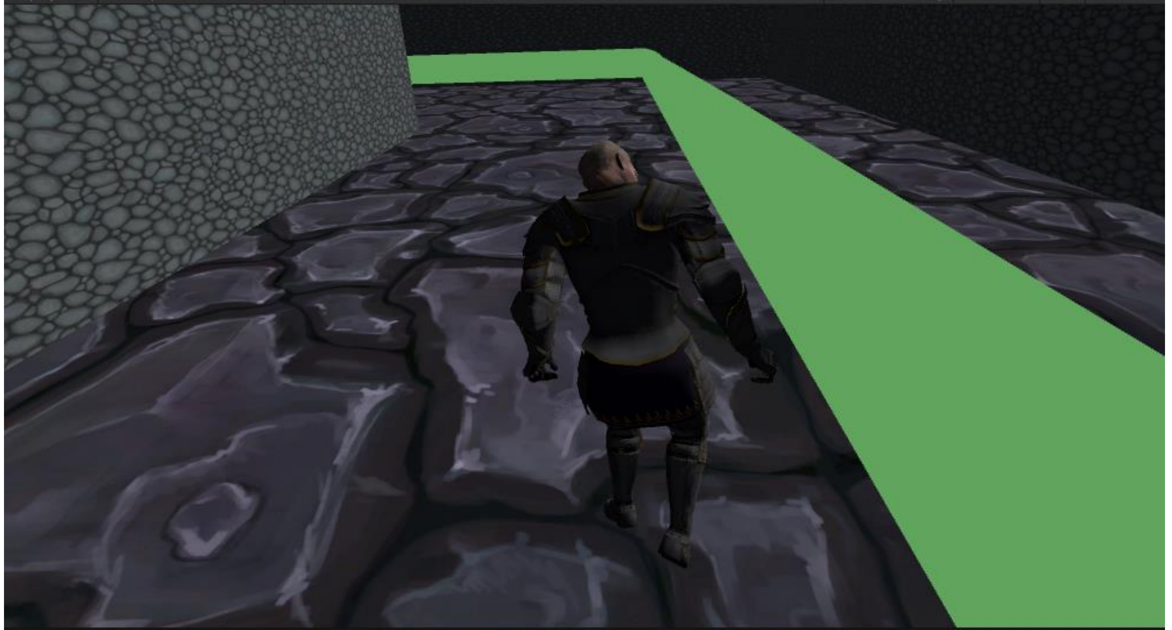

Рис. 3.32. Вигляд сцени для гравця

Для проходження рівня гри вам потрібно дійти до кінця лабіринту після чого ви переходите на рівень з головоломкою або босом. На рівні з босом вам потрібно протягом хвилини намагатися вижити проти нього після чого виз можете перейти на новий рівень (рис. 3.33).

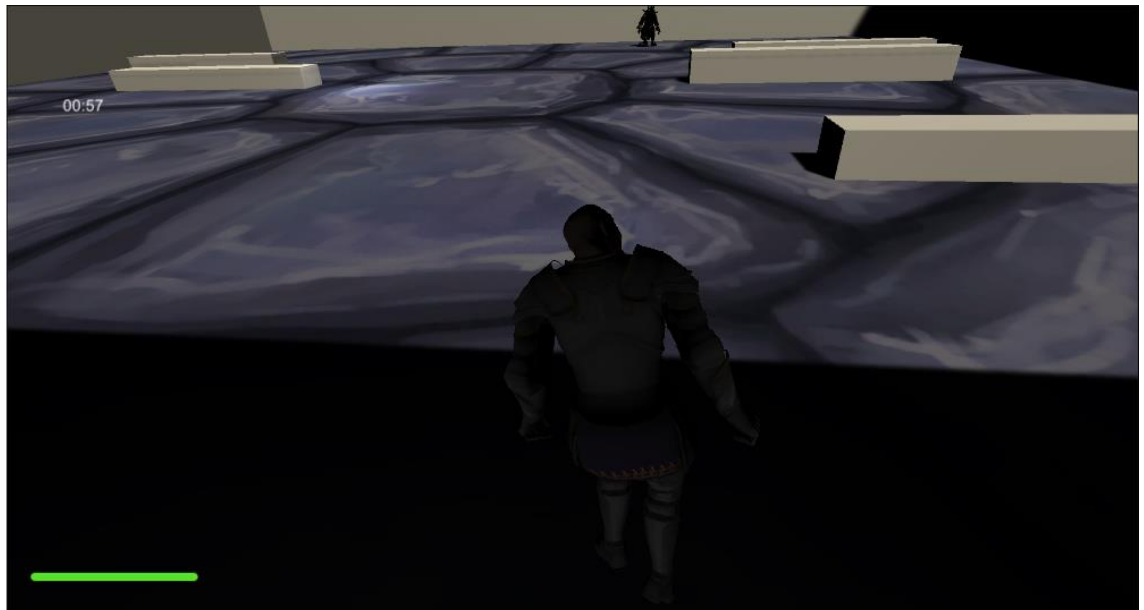

Рис. 3.33. Вигляд сцени з босом

Як ви бачите, герой має рівень здоров'я і якщо він втратить все здоров'я то гру треба буде починати знову (рис. 3.34).

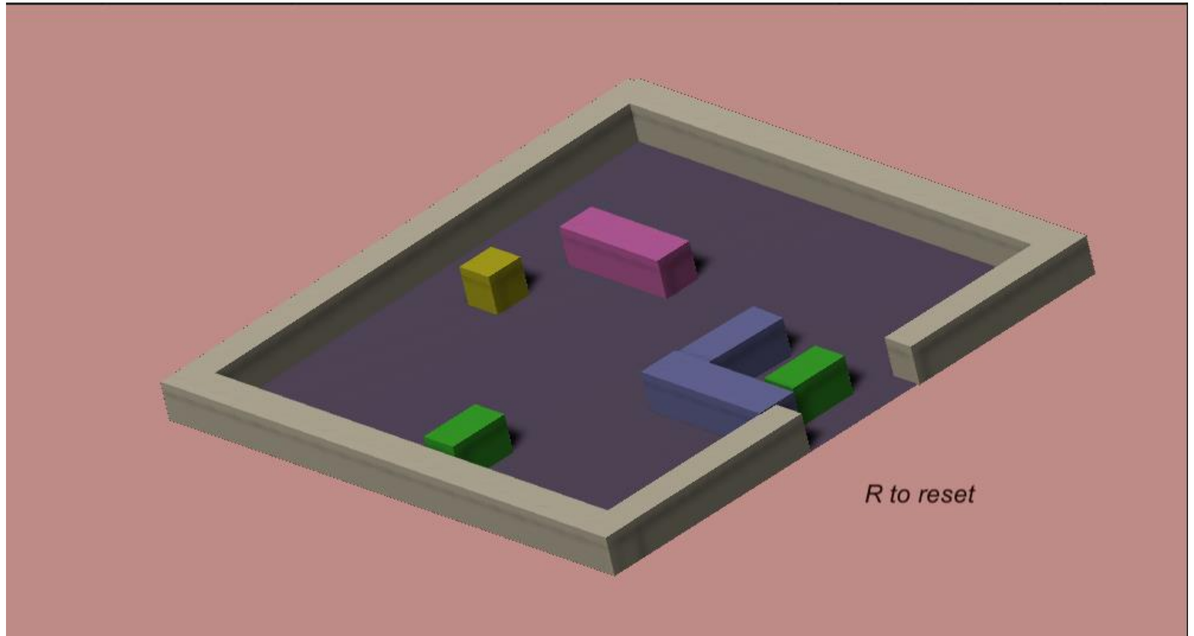

Рис. 3.34. Вигляд сцени з головоломкою

Для стилізації цього рівня була імпортована бібліотека для пост-обробки, щоб покращити вигляд сцени. Також для анімації тексту в головному меню використовувалась бібліотека TextMeshPro(файл WoblingText.cs). Для проходження цього рівня потрібно вирішити не складну головоломку після чого вас перекине на наступний рівень. Також якщо виникнуть якісь труднощі з проходженням головоломки ви може почати вирішувати її знов. Для покращення ігрового процесу для гри був доданий музичний супровід. Був створений ігровий об'єкт на який був доданий компонент Audio Source. Він зображений на малюнку 3.35.

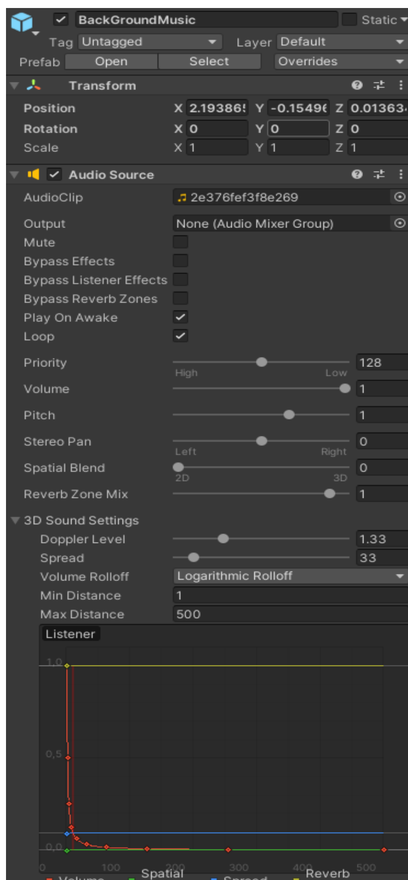

Рис. 3.35. Вигляд об’єкта для додавання музики

Загалом було додано чотири музикальні композиції. Одна для меню, одна для рівня з лабіринтом і так само для інших рівнів.

ВИСНОВКИ

У кваліфікаційній бакалаврській роботі проаналізовані основні методи та алгоритми процедурної генерації рівнів в даному випадку лабіринтів. Були розглянуті приклади ігор з використанням даної технології та з таким жанром який був використаний для проєкту. Після чого було проведено ознайомлення та порівняння інструментів для вирішення даної проблеми. Після зваження усіх мінусів та плюсів цих програм була обрана середа Unity та мова програмування C#. Потім були сформовані головні вимоги до продукту та розроблена назва гри, після чього для більш точного розуміння поставленої задачі були розроблені: Use-case та декомпозиційна діаграма. Після чого був обраний метод генерації яким став рекурсивний метод. Після чого була розпочата розробка продукту: створені моделі героїв, анімації. Написання програмного коду, створення користувацького інтерфейсу, додавання музики та все інше, що було описано у попередньому розділі. Після виконання цього всього були виконані всі вимоги, які ставились перед проектом. Після цього були прибрані баги, які зміг знайти я та товариші яким я показав даний проект. Основні результати роботи представлені на міжнародній науково-технічній конференції студентів та молодих вчених «Інформатика, математика, автоматика» (ІМА – 2021) (Суми, 2021 р.) [21]. Рецензенти відзначили вигляд проєкту та складність створення ігор. Також у подальшому даний проєкт може бути модифікований за допомогою додання рівнів складності, онлайн складової або бойової системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. A. BEATTIE, “How the Video Game Industry Is Changing.” https://www.investopedia.com/articles/investing/053115/how-video-game-industry-changing.asp (accessed May 21, 2021).
2. WePc, “Video Game Industry Statistics, Trends and Data In 2021.” https://www.wepc.com/news/video-game-statistics/ (accessed May 21, 2021).
3. Y. I. H. Parish and P. Müller, “Procedural modeling of cities,” in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2001*, 2001, pp. 301–308, doi: 10.1145/383259.383292.
4. P. Prusinkiewicz, L. Mündermann, R. Karwowski, and B. Lane, “The use of positional information in the modeling of plants,” in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2001*, 2001, pp. 289–300, doi: 10.1145/383259.383291.
5. J. D. Génevaux, É. Galin, É. Guérin, A. Peytavie, and B. Beneš, “Terrain generation using procedural models based on hydrology,” *ACM Trans. Graph.*, vol. 32, no. 4, Jul. 2013, doi: 10.1145/2461912.2461996.
6. “Генерация карты — Официальная Minecraft Wiki.” https://minecraft.fandom.com/ru/wiki/Генерация_карты (accessed May 23, 2021).
7. A. Kaleev, “Как Minecraft устроена за кадром.” https://dtf.ru/s/minecraft/202337-kak-minecraft-ustroena-za-kadrom (accessed May 23, 2021).
8. “Зерно (генерирование мира) — Официальная Minecraft Wiki.” https://minecraft.fandom.com/ru/wiki/Зерно_(генерирование_мира) (accessed May 23, 2021).
9. “Погода — Официальная Minecraft Wiki.” https://minecraft.fandom.com/ru/wiki/Погода (accessed May 23, 2021).

10. T. Hely, “Использование BSP-деревьев для создания игровых карт / Хабр.” https://habr.com/ru/post/332832/ (accessed May 23, 2021).
11. “Drunken Master cave generation.” https://forums.roguetemple.com//index.php?topic=4128.0. (accessed May 23, 2021).
12. M. Cook, “Generate Random Cave Levels Using Cellular Automata.” https://gamedevelopment.tutsplus.com/tutorials/generate-random-cave-levels-using-cellular-automata--gamedev-9664 (accessed May 23, 2021).
13. Владимир Скрипин, “Сборы на разработку космического симулятора Star Citizen превысили $250 млн - ITC.ua.” https://itc.ua/news/sbory-na-razrabotku-kosmicheskogo-simulyatora-star-citizen-prevysili-250-mln/ (accessed May 24, 2021).
14. Ragequit, “7 примеров использования процедурной генерации в играх, о которых полезно знать всем разработчикам / Блог компании ua-hosting.company / Хабр.” https://habr.com/ru/company/ua-hosting/blog/275195/ (accessed May 24, 2021).
15. RussDragon, “Классические алгоритмы генерации лабиринтов. Часть 2: погружение в случайность / Хабр.” https://habr.com/ru/post/321210/ (accessed May 26, 2021).
16. PatientZero, “Лабиринты: классификация, генерирование, поиск решений / Хабр.” https://habr.com/ru/post/445378/ (accessed May 03, 2021).
17. A. Sinicki, “What is Unity? Everything you need to know - Android Authority.” https://www.androidauthority.com/what-is-unity-1131558/ (accessed May 27, 2021).
18. “A Unity Review: Pros and Cons | CitrusBits.” https://www.citrusbits.com/a-unity-review-pros-and-cons/ (accessed May 27, 2021).
19. Lysis, “Pros and Cons of Using C# as Your Backend Programming Language.” https://agilites.com/pros-and-cons-of-using-c-as-your-backend-programming-language.html (accessed May 27, 2021).

20. R. C. MOSS, “ASCII art + permadeath: The history of roguelike games | Ars Technica.” https://arstechnica.com/gaming/2020/03/ascii-art-permadeath-the-history-of-roguelike-games/ (accessed May 27, 2021).

21. Фоменко В. О. Відеогра з процедурною генерацією рівнів на Unity //Матеріали та програма міжнародної науково-технічної конференції студентів та молодих вчених «Інформатика, математика, автоматика» (ІМА – 2021) (Суми, 2021 р.), м. Суми, Україна. - Суми, 2021. - С. 58-59 - https://elitconference.sumdu.edu.ua/proceedings/.

ДОДАТОК А

ЛІСТИНГ МОДУЛІВ ЕКСПЕРТНОЇ СИСТЕМИ

Файл cell.cs

```
using UnityEngine;

public class Cell : MonoBehaviour
{
    public GameObject WallLeft;
    public GameObject WallBottom;
    public GameObject Floor;
}
```

Файл Activator.cs

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

public class Activator : MonoBehaviour
{
    public GameObject[] firstGroup;
    public GameObject[] secondGroup;
    public Activator button;
    public Material normal;
    public Material transponent;
    public bool canPush;


    private void OnTriggerEnter(Collider other)
    {
        if (canPush)
        {
            if (other.CompareTag("Cube") ||
other.CompareTag("Player"))
            {
                foreach (GameObject first in firstGroup)
                {
                    first.GetComponent<Renderer>().material =
normal;
                    first.GetComponent<Collider>().isTrigger
= true;
                }
                foreach (GameObject second in secondGroup)
                {
                    second.GetComponent<Renderer>().material
= transponent;
```

```
                    second.GetComponent<Collider>().isTrigger
= false;
                }
                GetComponent<Renderer>().material =
transponent;
                button.GetComponent<Renderer>().material =
normal;
                button.canPush = true;
            }
        }
    }
}
```

Файл Damage.cs

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

[RequireComponent(typeof(AudioSource))]
public class Damage : MonoBehaviour
{
    public AudioClip myClip;
    private AudioSource mySource;

    private void Start()
    {
        mySource = GetComponent<AudioSource>();
    }

    void OnTriggerEnter (Collider myCollider)
    {
        if (myCollider.tag == "Player")
        {

myCollider.GetComponent<HealthLevel>().levelHealth -= 10 ;
            mySource.PlayOneShot(myClip);
        }
    }
}
```

Файл HealthLevel.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
public class HealthLevel : MonoBehaviour
{
```

```
    public int levelHealth = 100;
    public Slider mySlider;
    public Image myImage;

    void Update()
    {
        mySlider.value = levelHealth;

        if (levelHealth < 10)
        {
            myImage.enabled = false;
        }
        else
        {
            myImage.enabled = true;
        }
    }
}
```

Файл HintRenderer.cs

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



    public class HintRenderer : MonoBehaviour
    {
        public MazeSpawner MazeSpawner; //ссылка на лабиринт

        private LineRenderer componentLineRenderer;

        private void Start()
        {
            componentLineRenderer =
GetComponent<LineRenderer>();
            //Instantiate(nextLevelSpot, new Vector3(x, y,
5));
        }

        public void DrawPath()
        {
            Maze maze = MazeSpawner.maze;
            int x = maze.finishPosition.x;
            int y = maze.finishPosition.y;
            List<Vector3> positions = new List<Vector3>();

            Debug.Log("Points2" +" "+maze.finishPosition.x +"
"+ maze.finishPosition.y);
            while ((x != 0 || y != 0) && positions.Count <
10000)
```

```
            {
                positions.Add(new Vector3(x * 
MazeSpawner.CellSize.x, y * MazeSpawner.CellSize.y, y * 
MazeSpawner.CellSize.z));

                MazeGeneratorCell currentCell = maze.cells[x, 
y];

                if (x > 0 &&
                    !currentCell.WallLeft &&
                    maze.cells[x - 1, y].DistanceFromStart < 
currentCell.DistanceFromStart)
                {
                    x--;
                }
                else if (y > 0 &&
                    !currentCell.WallBottom &&
                    maze.cells[x, y - 1].DistanceFromStart < 
currentCell.DistanceFromStart)
                {
                    y--;
                }
                else if (x < maze.cells.GetLength(0) - 1 &&
                    !maze.cells[x + 1, y].WallLeft &&
                    maze.cells[x + 1, y].DistanceFromStart < 
currentCell.DistanceFromStart)
                {
                    x++;
                }
                else if (y < maze.cells.GetLength(1) - 1 &&
                    !maze.cells[x, y + 1].WallBottom &&
                    maze.cells[x, y + 1].DistanceFromStart < 
currentCell.DistanceFromStart)
                {
                    y++;
                }
            }
            positions.Add(Vector3.zero);
            componentLineRenderer.positionCount = 
positions.Count;

componentLineRenderer.SetPositions(positions.ToArray());
        }

 }
```

Файл Maze.cs

```
using UnityEngine;

public class Maze
```

```
{
    public MazeGeneratorCell[,] cells;
    public Vector2Int finishPosition; //точка выхода из
лабиринта
    public int zPosition = 5;
}

public class MazeGeneratorCell
{
    public int X;
    public int Y;

    public bool WallLeft = true;
    public bool WallBottom = true;

    public bool Visited = false;
    public int DistanceFromStart;
}
```

Файл MazeGenerator.cs

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

public class MazeGenerator : MonoBehaviour
{
    //public GameObject nextLevelSpot;
    //Размеры лабиринта
    public int Width = 50;
    public int Height = 50;

    public Maze GenerateMaze()
    {
        MazeGeneratorCell[,] cells = new
MazeGeneratorCell[Width, Height];

        for (int x = 0; x < cells.GetLength(0); x++)
        {
            for (int y = 0; y < cells.GetLength(1); y++)
            {
                cells[x, y] = new MazeGeneratorCell {X = x, Y
= y};
            }
        }
        //Удаление лишних стен
        for (int x = 0; x < cells.GetLength(0); x++)
        {
            cells[x, Height - 1].WallLeft = false;
        }

        for (int y = 0; y < cells.GetLength(1); y++)
```

```
        {
            cells[Width - 1, y].WallBottom = false;
        }

        RemoveWallsWithBacktracker(cells);

        Maze maze = new Maze();

        maze.cells = cells;
        maze.finishPosition = PlaceMazeExit(cells);

        return maze;
    }

    private void
RemoveWallsWithBacktracker(MazeGeneratorCell[,] maze)
    {
        MazeGeneratorCell current = maze[0, 0]; //текущая
клетка(начало 0, 0) потом двигаемся в рандомном направлении
до тупика после чего откатываемся назад до момента когда
снова можно убрать стену и т. д до стартовой ячейки;
        current.Visited = true; // посещение клетки
        current.DistanceFromStart = 0; // дистанция от старта

        Stack<MazeGeneratorCell> stack = new
Stack<MazeGeneratorCell>(); // Перемещение по клеткам
        do
        {
            List<MazeGeneratorCell> unvisitedNeighbours = new
List<MazeGeneratorCell>();

            int x = current.X;
            int y = current.Y;

            if (x > 0 && !maze[x - 1, y].Visited)
unvisitedNeighbours.Add(maze[x - 1, y]);
            if (y > 0 && !maze[x, y - 1].Visited)
unvisitedNeighbours.Add(maze[x, y - 1]);
            if (x < Width - 2 && !maze[x + 1, y].Visited)
unvisitedNeighbours.Add(maze[x + 1, y]);
            if (y < Height - 2 && !maze[x, y + 1].Visited)
unvisitedNeighbours.Add(maze[x, y + 1]);

            if (unvisitedNeighbours.Count > 0)
            {
                MazeGeneratorCell chosen =
unvisitedNeighbours[UnityEngine.Random.Range(0,
unvisitedNeighbours.Count)];
                RemoveWall(current, chosen);

                chosen.Visited = true;
                stack.Push(chosen);
```

```
                chosen.DistanceFromStart =
current.DistanceFromStart + 1;
                current = chosen;
            }
            else
            {
                current = stack.Pop();
            }
        } while (stack.Count > 0);
    }

    private void RemoveWall(MazeGeneratorCell a,
MazeGeneratorCell b)
    {
        if (a.X == b.X)
        {
            if (a.Y > b.Y) a.WallBottom = false;
            else b.WallBottom = false;
        }
        else
        {
            if (a.X > b.X) a.WallLeft = false;
            else b.WallLeft = false;
        }
    }

    private Vector2Int PlaceMazeExit(MazeGeneratorCell[,]
maze)
    {

        MazeGeneratorCell furthest = maze[0, 0];
        //Самая далекая ячейка от старта удачляется
        for (int x = 0; x < maze.GetLength(0); x++)
        {
            if (maze[x, Height - 2].DistanceFromStart >
furthest.DistanceFromStart) furthest = maze[x, Height - 2];
            if (maze[x, 0].DistanceFromStart >
furthest.DistanceFromStart) furthest = maze[x, 0];
        }

        for (int y = 0; y < maze.GetLength(1); y++)
        {
            if (maze[Width - 2, y].DistanceFromStart >
furthest.DistanceFromStart) furthest = maze[Width - 2, y];
            if (maze[0, y].DistanceFromStart >
furthest.DistanceFromStart) furthest = maze[0, y];
        }

        //В зависимости от нахождения ячейки финала удяляется
стена для выхода
        if (furthest.X == 0) furthest.WallLeft = false;
```

```
        else if (furthest.Y == 0) furthest.WallBottom = 
false;
        else if (furthest.X == Width - 2) maze[furthest.X + 
1, furthest.Y].WallLeft = false;
        else if (furthest.Y == Height - 2) maze[furthest.X, 
furthest.Y + 1].WallBottom = false;

        return new Vector2Int(furthest.X, furthest.Y); 
//Позиция финиша
    }
}
```

Файл MazeSpawner.cs

```
using UnityEngine;
using UnityEngine.AI;

public class MazeSpawner : MonoBehaviour
{
    public Cell CellPrefab;
    public Vector3 CellSize = new Vector3(1,1,0); //Размер 
одной ячейки x y
    public HintRenderer HintRenderer;
    public GameObject PahtObj;
    public GameObject LevelExit;
    private NavMeshSurface[] navMeshSurfaces;

    public Maze maze;

    private void Start()
    {
        MazeGenerator generator = new MazeGenerator();

        maze = generator.GenerateMaze();

        for (int x = 0; x < maze.cells.GetLength(0); x++)
        {
            for (int y = 0; y < maze.cells.GetLength(1); y++)
            {
                Cell c = Instantiate(CellPrefab, new 
Vector3(x * CellSize.x, y * CellSize.y, y * CellSize.z), 
Quaternion.identity);
                c.WallLeft.SetActive(maze.cells[x, 
y].WallLeft);
                c.WallBottom.SetActive(maze.cells[x, 
y].WallBottom);
                //nextLevelSpot.transform.position = 
            }
        }
        Instantiate(LevelExit, new 
Vector3(maze.finishPosition.x * CellSize.x, 
maze.finishPosition.y * CellSize.y, CellSize.z), 
Quaternion.identity);
```

```
        Debug.Log("Points" +" "+maze.finishPosition.x +" "+
maze.finishPosition.y);

    }

    private void Update()
    {
        if (Input.GetKey(KeyCode.F))
        {
            HintRenderer.DrawPath();
            PahtObj.SetActive(true);
        }
        else if (Input.GetKey(KeyCode.G))
        {
            PahtObj.SetActive(false);
        }
    }

}
```

Файл NavMeshBaker.cs

```
using System.Collections;
using System.Collections.Generic;
using System.Threading.Tasks;
using UnityEngine;
using UnityEngine.AI;

public class NavMeshBaker : MonoBehaviour
{
    [SerializeField] NavMeshSurface[] navMeshSurfaces;

    async void Start()
    {
            await Task.Delay(2000);

            for (int i = 0; i < navMeshSurfaces.Length; i++)
            {
                navMeshSurfaces[i].BuildNavMesh();
            }
    }
}
```

Файл RandomCarck.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;
```

```
public class RandomCrack : MonoBehaviour
{

    private void OnTriggerEnter(Collider col)
    {

        if (col.tag == "Player")
        {
            if (Random.Range(0, 1) == 0) {

                SceneManager.LoadScene("InProgress");

            } else {
                //Application.LoadLevel(level);
                SceneManager.LoadScene("Crack" +
Random.Range(1, 8));
            }
        }
    }
}
```

Файл Timer.cs

```
using UnityEngine;
using UnityEngine.UI;

public class Timer : MonoBehaviour
{
    public float timeRemaining = 10;
    public bool timerIsRunning = false;
    public Text timeText;
    public GameObject nxtLevel;

    private void Start()
    {
        timerIsRunning = true;
    }

    void Update()
    {
        if (timerIsRunning)
        {
            if (timeRemaining > 0)
            {
                timeRemaining -= Time.deltaTime;
                DisplayTime(timeRemaining);
            }
            else
            {
                nxtLevel.SetActive(true);
```

```
                timeRemaining = 0;
                timerIsRunning = false;
            }
        }
    }

    void DisplayTime(float timeToDisplay)
    {
        timeToDisplay += 1;

        float minutes = Mathf.FloorToInt(timeToDisplay / 60);
        float seconds = Mathf.FloorToInt(timeToDisplay % 60);

        timeText.text = string.Format("{0:00}:{1:00}",
minutes, seconds);
    }
}
```

Файл TriggerLoadLevel.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class TriggerLoadLevel : MonoBehaviour
{

    private void OnTriggerEnter(Collider col)
    {
        if(col.tag == "Player")
        {
            SceneManager.LoadScene("Maze3D");
        }
    }
}
```

Файл CameraFollow.cs

```
using UnityEngine;

public class CameraFollow : MonoBehaviour
{
    [SerializeField] [Range(1f, 5f)] private float
_angularSpeed = 1f;

    [SerializeField] private Transform _target;

    private float _angleY;

    private void Start()
    {
```

```
            _angleY = transform.rotation.y;
        }

        private void Update()
        {
            if (Input.GetKey(KeyCode.Z)) _angleY -=
    _angularSpeed;
            if (Input.GetKey(KeyCode.X)) _angleY +=
    _angularSpeed;

            transform.position = _target.transform.position;
            transform.rotation = Quaternion.Euler(0, _angleY, 0);
        }
    }
```

Файл MovementCharacteristics.cs

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

namespace Dungeon
{
    [CreateAssetMenu(fileName = "Characteristics", menuName =
"Movement/MovementCharacteristics", order = 1)]
    public class MovementCharacteristics : ScriptableObject
    {
        [SerializeField] private bool _visibleCursor = false;

        [SerializeField] private float _movementSpeed = 1f;

        [SerializeField] private float _runSpeed = 1.5f;
        [SerializeField] private float _angularSpeed = 150f;

        [SerializeField] private float _gravity = 15f;

        [SerializeField] private float _jumpForce = 7f;

        public bool VisibleCursor => _visibleCursor;

        public float MovementSpeed => _movementSpeed;

        public float RunSpeed => _runSpeed;

        public float AngularSpeed => _angularSpeed;

        public float Gravity => _gravity / 10f;

        public float JumpForce => _jumpForce / 100f;
    }
```

```
}
```

Файл ThirdPersonmovement.cs

```
using UnityEngine;


namespace Dungeon
{
    [RequireComponent(typeof(CharacterController))]
    [RequireComponent(typeof(Animator))]
    public class ThirdPersonMovement : MonoBehaviour
    {
        [SerializeField] private Transform _camera;
        [SerializeField] private MovementCharacteristics
_characteristics;

        private float _vertical, _horizontal, _run;

        private readonly string STR_VERTICAL = "Vertical";
        private readonly string STR_HORIZONTAL =
"Horizontal";
        private readonly string STR_RUN = "Run";
        private readonly string STR_JUMP = "Jump";

        private const float DISTANCE_OFFSET_CAMERA = 5f;

        private CharacterController _controller;
        private Animator _animator;

        private Vector3 _direction;
        private Quaternion _look;

        private Vector3 TargetRotate => _camera.forward *
DISTANCE_OFFSET_CAMERA;
        private bool Idle => _horizontal == 0.0f && _vertical
== 0.0f;


        private void Start()
        {
            _controller =
GetComponent<CharacterController>();
            _animator = GetComponent<Animator>();

            Cursor.visible = _characteristics.VisibleCursor;
        }

        private void Update()
        {
            Movement();
```

```
            Rotate();
        }

        private void Movement()
        {
            if (_controller.isGrounded)
            {
                _horizontal = Input.GetAxis(STR_HORIZONTAL);
                _vertical = Input.GetAxis(STR_VERTICAL);
                _run = Input.GetAxis(STR_RUN);

                _direction =
transform.TransformDirection(_horizontal, 0,
_vertical).normalized;

                PlayAnimation();
                Jump();
            }

            _direction.y -= _characteristics.Gravity *
Time.deltaTime;

            float speed = _run * _characteristics.RunSpeed +
_characteristics.MovementSpeed;
            Vector3 dir = _direction * speed *
Time.deltaTime;

            dir.y = _direction.y;

            _controller.Move(dir);
        }

        private void Rotate()
        {
            if (Idle) return;

            Vector3 target = TargetRotate;
            target.y = 0;

            _look = Quaternion.LookRotation(target);

            float speed = _characteristics.AngularSpeed *
Time.deltaTime;

            transform.rotation =
Quaternion.RotateTowards(transform.rotation, _look, speed);
        }

        private void Jump()
        {
            if (Input.GetButtonDown(STR_JUMP))
            {
```

```
                _animator.SetTrigger(STR_JUMP);
                _direction.y += _characteristics.JumpForce;
            }
        }

        private void PlayAnimation()
        {
            float horizontal = _run * _horizontal +
_horizontal;
            float vertical = _run * _vertical + _vertical;

            _animator.SetFloat(STR_VERTICAL, vertical);
            _animator.SetFloat(STR_HORIZONTAL, horizontal);
        }
    }
}
```

Файл Enemy.cs

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

[RequireComponent (typeof(NavMeshAgent))]
public class Enemy : MonoBehaviour
{
    private NavMeshAgent myAgent;
    private Animator myAnimator;
    private float distance;
    public Transform target;
    public Transform parking;

    void Start()
    {
        myAgent = GetComponent<NavMeshAgent>();
        myAnimator = GetComponent<Animator>();
    }

    void Update()
    {
        distance = Vector3.Distance(target.position,
transform.position);
        //if (distance > 20) // & transform.position !=
parking.position)
        //{
        //    transform.LookAt(target.position);
        //    myAgent.enabled = false;
        //    //myAnimator.Play("EIdle");
        //    myAnimator.SetBool("RunBool", true);
        //    myAgent.SetDestination(parking.position);
        //    myAnimator.SetBool("IdleBool", false);
        //}
```

```
        if (distance > 10) //& transform.position ==
parking.position
        {
            transform.LookAt(target.position);
            myAgent.enabled = false;
            //myAnimator.Play("EIdle");
            myAnimator.SetBool("IdleBool", true);
            myAnimator.SetBool("RunBool", false);
        }

        if (distance < 10 & distance > 1.5f)
        {
            myAgent.enabled = true;
            myAgent.SetDestination(target.position);
            myAnimator.SetBool("IdleBool", false);
            myAnimator.SetBool("RunBool", true);
            myAnimator.SetBool("AttackBool", false);
        }

        if (distance <= 1.5f)
        {
            myAgent.enabled = false;
            myAnimator.SetBool("RunBool", false);
            myAnimator.SetBool("AttackBool", true);
        }
    }


}
```

Файл ExitGame.cs

```
using UnityEngine;
using UnityEngine.UI;

public class ExitGame : MonoBehaviour
{
    public Button exitButton;

    void Start()
    {
        exitButton.onClick.AddListener(TaskOnClick);
    }

    void TaskOnClick()
    {
        Application.Quit();
    }
}
```

Файл GameEsc.cs

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

public class GameEsc : MonoBehaviour
{
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            Application.Quit();
        }
    }
}
```

Файл GameOver.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameOver : MonoBehaviour
{
    void Update()
    {
        if (GetComponent<HealthLevel>().levelHealth <= 10)
        {
            SceneManager.LoadScene("EndScr");

        }
    }
}
```

Файл Pause.cs

```
using UnityEngine;

public class Pause : MonoBehaviour
{
    public GameObject pausePanel;

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            pausePanel.SetActive(true);
            Time.timeScale = 0;
        }
```

```
        if (Input.GetKeyDown(KeyCode.Space))
        {
            pausePanel.SetActive(false);
            Time.timeScale = 1;
        }
    }
}
```

Файл PlayGame.cs

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

public class PlayGame : MonoBehaviour
{
    public string sceneName;
    public Button playButton;

    void Start()
    {
        playButton.onClick.AddListener(TaskOnClick);
    }

    void TaskOnClick()
    {
        SceneManager.LoadScene(sceneName);

    }

}
```

Файл CubePlayer.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class CubePlayer : MonoBehaviour
{
    [SerializeField] KeyCode keyOne;
    [SerializeField] KeyCode keyTwo;
    [SerializeField] Vector3 movementDiraction;



    private void FixedUpdate()
    {
```

```
        if(Input.GetKey(keyOne))
        {
            GetComponent<Rigidbody>().velocity +=
movementDiraction;
        }

        if (Input.GetKey(keyTwo))
        {
            GetComponent<Rigidbody>().velocity -=
movementDiraction;
        }

        if (Input.GetKey(KeyCode.R))
        {

SceneManager.LoadScene(SceneManager.GetActiveScene().buildInd
ex);
        }


}
    private void OnTriggerEnter(Collider other)
    {
        if(this.CompareTag("Player") &&
other.CompareTag("Finish"))
        {

SceneManager.LoadScene(SceneManager.GetActiveScene().buildInd
ex + 1);
        }
        if (this.CompareTag("Cube") &&
other.CompareTag("Cube"))
        {
            foreach (Activator button in
FindObjectsOfType<Activator>())
            {
                button.canPush = false;
            }
        }
    }

    private void OnTriggerExit(Collider other)
    {
        if (this.CompareTag("Cube") &&
other.CompareTag("Cube"))
        {
            foreach (Activator button in
FindObjectsOfType<Activator>())
            {
                button.canPush = true;
            }
        }
```

```
    }

}
```

Файл WoblingText.cs

```
using UnityEngine;
using TMPro;

public class WoblingText : MonoBehaviour
{
    public TMP_Text textComponent;

    void Update()
    {
        textComponent.ForceMeshUpdate();

        var textInfo = textComponent.textInfo;

        for (int i = 0; i < textInfo.characterCount; ++i)
        {
            var charInfo = textInfo.characterInfo[i];

            if (!charInfo.isVisible)
            {
                continue;
            }

            var verts =
textInfo.meshInfo[charInfo.materialReferenceIndex].vertices;

            for(int j = 0; j < 4; ++j)
            {
                var orig = verts[charInfo.vertexIndex + j];
                verts[charInfo.vertexIndex + j] = orig + new
Vector3(0, Mathf.Sin(Time.time * 2f + orig.x * 0.01f) * 20f,
0);
            }
        }
        for (int i = 0; i < textInfo.meshInfo.Length; ++i)
        {
            var meshInfo = textInfo.meshInfo[i];
            meshInfo.mesh.vertices = meshInfo.vertices;
            textComponent.UpdateGeometry(meshInfo.mesh, i);
        }
    }
}
```

Файл OnTheGround.cs

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

public class OnTheGround : MonoBehaviour
{
    [SerializeField] GameObject Player;

    void Update()
    {
        if (Player.transform.position.y <= -5)
        {
            if (Random.Range(0, 1) == 0)
            {
                SceneManager.LoadScene("InProgress");
            }
            else if (Random.Range(0, 1) == 1)
            {

                //Application.LoadLevel(level);
                SceneManager.LoadScene("Crack" +
Random.Range(1, 8));
            }
        }
    }
}
```