КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теоретичної кібернетики

Кваліфікаційна робота
на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки

на тему:

Розроблення гри-квесту у тривимірному просторі з використанням засобів Unity

Виконав студент 4-го курсу
Сергій НАЗАРЧУК _____
(підпис)

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Тетяна КАРНАУХ _____
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри теоретичної кібернетики

«____»_____2023 р., протокол № ____

Завідувач кафедри

Юрій КРАК _____

(підпис)

Київ - 2023

РЕФЕРАТ

Обсяг роботи 59 сторінок, 27 ілюстрацій, 9 джерел посилань, 7 додатків.

ВІДЕОГРА, ГЕЙМДИЗАЙН, UNITY ENGINE, C#, VISUAL STUDIO, ГЕЙМПЛЕЙ, PREFAB, MONOBEHAVIOR, ГОЛОВОЛОМКИ.

Об’єктом роботи є створення гри-квесту за допомогою Unity Engine. Предметом роботи є реалізація розв’язних головоломок, які мають можливість бути згенерованими випадково.

Метою роботи є розроблення та програмна реалізація гри-квесту, а також підбір необхідних засобів та технологій.

Інструменти розроблення: мова програмування C# у середовищі розробки Microsoft Visual Studio, двигун Unity Engine.

Результати роботи: створено гру-квест, при цьому спроєктовано три головоломки («Кольорові кнопки», «Відбиваючі лазери», «Лабіринт») та спосіб їх інтеграції у сценарій гри; спроєктовані головоломки досліджено та для них реалізовано автоматичну генерацію завдань.

Розроблене програмне забезпечення може використовуватись для додавання головоломок та інших запрограмованих екземплярів у інші проекти. Наявна можливість додавання проекту у різні ігрові магазини по типу Steam або Google Play.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

LINQ (Language Integrated Query) – запити, інтегровані в мову

XML (eXtensible Markup Language) – розширювана мова розмітки

ВСТУП

Оцінка сучасного стану об’єкта дослідження або розробки. Ігрова індустрія сьогодні – це бізнес світового масштабу, який на відміну від кінематографа акумулюється не тільки навколо «Голівуду» в Америці. Масштабні ігрові студії є і в Франції (Ubisoft, Focus Entertainment), Швеції (EA Games, Paradox Interactive), Англії (Deep Silver), Японії (Nintendo, Sony, Sega, Bandai Namco, Koei Tecmo, Capcom) ще в десятках країн, та навіть в Україні (GSC Game World, 4AGames).

За свою 80-літню історію відеоігри перетворилися із простих ігор у гральних автоматах по типу Pong (1972) на проєкти, в які вкладають сотні мільйонів доларів, наприклад Red Dead Redemption 2 (2018) та Cyberpunk 2077 (2020).

І на відміну від Голівуду, де звичайний режисер не може показати свій фільм у світі, поки не отримаєш правильних знайомств або не будеш працювати з масштабними кіностудіями, ігрова індустрія дає шанс кожній охочій до геймдизайну людині створити свій проект, який сподобається мільйонам гравців, як от Ender Lilies (2021), Factorio (2016) та Barotrauma (2019).

Ще, на відміну від інших наук комп’ютерної розробки ігрова індустрія включає у себе багато різних напрямів програмування. Будь то штучний інтелект ворогів у шутері, або використання штучних нейронних мереж для просунутих технологій підвищення частоти кадрів, за допомогою домальовування текстур та шейдерів в реальному часі, для збереження красивої та динамічної графіки нового покоління.

Також, наприклад, при технології Motion Matching [1] заздалегідь записані анімації використовуються в якості змінних, для надання рухам героя більшої плавності та реалізму. Ця технологія потребує в рази більше записаних анімацій, та знання того, як краще побудувати зв’язок між двома анімаціями так, щоб

головний герой виглядав більш автентично.

Тому індустрія відеоігор є настільки популярною у сучасному світі, через перетин багатьох допоміжних індустрій, будь то геймдизайн, ілюстрація, кінематограф або музика.

Актуальність роботи та підстави для її виконання. Не дивлячись на масштаб індустрії відеоігор, класичних ігор-квестів, в яких необхідно проходити різноманітні рівні, вирішуючи цікаві головоломки, не так багато. Створення квесту-головоломки може стати успішним через малу кількість представників цього жанру в сучасному світі.

Мета і завдання роботи. Розробити гру-квест з тривимірною візуалізацією з використанням засобів Unity. Гра має містити різні головоломки.

Для виконання мети, необхідно виконати таке.

1. Ознайомитись з двигуном Unity та його внутрішніми бібліотеками (зокрема класом Monobehavior)
2. Спроєктувати головоломки зі зручними елементами взаємодії для їх вирішення.
3. Спроєктувати гру в цілому, зокрема наповнення її головоломками.
4. Реалізувати розроблену логіку з використанням засобів Unity.

Об’єкт, методи й засоби розроблення. Об'єктом розроблення є гра-квест та її складові головоломки. Для розроблення програмного продукту використовувався інтерактивний та інкрементний підхід та середовище Microsoft Visual Studio. Засобами реалізації є мова програмування C#, двигун Unity Engine, зокрема MonoBehavior та інші внутрішні бібліотеки і класи Unity.

Можливі сфери застосування. Розроблені в межах виконання цього проєкту патерни та префаби можуть бути використані для створення інших ігор. Сформований під час роботи над задачею підхід до головоломок та геймдизайн може використовуватись в інших проектах, що будуються з використанням Unity. Проєкт у робочому вигляді можна викласти на різних платформах типу Steam або Google Play.

1 КОМП'ЮТЕРНІ ІГРИ В ЖАНРІ «КВЕСТ»

Ігрова індустрія нараховує велику кількість різноманітних жанрів створених, щоб догодити якомога більшій кількості гравців, враховуючи їх вподобання. До того в гри може не бути суттєвого визначення за жанром, тому багато розробників комбінують жанри як їм заманеться та створюють щось нове для індустрії. Наприклад українці з компанії GSC Game World [2] при розробці "S.T.A.L.K.E.R." успішно скомбінували два різні жанри та створили свою унікальну серію ігор.

В ігровій індустрії є ще багато інших популярних жанрів. турецькі розробники з компанії TaleWorlds [3] при розробці серії ігор "Mount and Blade" успішно схрестили RPG, стратегію на глобальній карті та симулятор життя у середньовіччі – і з цього теж вийшла нова унікальна серія відеоігор.

Багато сучасних розробників не опираються на створення гри тільки відповідного жанру, а схрещують різні, іноді на перший погляд несумісні, жанри, додають свої ідеї, свій почерк та вдосконалюють свої старі напрацювання для створення нового продукту.

Одним з популярних жанрів є ігри в жанрі "квест", який бачиться таким, що може комбінуватись з іншими, а тому потенційно цікавий для наробки методів та прийомів розроблення.

1.1 Коротка характеристика жанру «Квест»

Ігри-квести – це один із найстаріших жанрів ігрової індустрії. Якщо спробувати розшифрувати назву жанру, то це будуть слова «шукати» та «розшукувати», проте частіше як визначення жанру використовують «пригодницька гра». Але до того ж він один із самих найрізноманітніших жанрів:

до складу жанру входять такі піджанри, як Metroidvania, текстові квести, візуальні новели, інтерактивне кіно та платформери.

1.2 Перші квести, розроблені в 70-х та 80-х

У витоків жанру квестів знаходяться **текстові квести**, популярні в середині 80-х років минулого сторіччя. Прикладами хітів тих років є PlanetFall (1977), The Hobbit (1982), Questprobe: Featuring Human Torch and the Thing (1985) та The Lurking Horror (1987). Ігровий процес в них зводився до відповідей на запитання, які ставить гра. У кінці 70-х текстові квести виглядали, як простий текстовий документ в консолі, який сьогодні може запрограмувати кожен, з поправкою на якість написаного сценарію. У 80-х у текстові квести почали додавати піксельні картинки, які вже допомагали мозку домалювати картинку того, що відбувається в історії.

1.3 Еволюція жанру квестів в 90-х

Наступна еволюція жанру сталась в кінці 80-х - на початку 90-х з приходом “Point-and-click” квестів. Тоді звичайні картинки еволюціонували в повноцінні інтерактивні об'єкти – локації. Тепер герой мав свою текстуру, переміщався по місцям за допомогою наперед записаних анімацій та міг реагувати на навколишнє середовище. Пізніше жанр квестів із 90-х отримав назву “Point-and-click”, що в перекладі означало буквально «вказуй та натискай». Ці квести були одними з перших ігор, які були запрограмовані на керування ігровим процесом за допомогою миші. Тут головоломки вже стали більш комплексними, ніж загадки та питання із 80-х, тут вони стали прив’язані більше до предметів.

Приклад головоломки (з гри The Legend of Kyrandia, Book One (1992) [4]): віз в який запряжений осел перешкоджає шлях. Що робити? В сусідній локації «сад» ви знайшли яблуко. Беремо яблуко і підманюємо осла. Він відтягує віз. Прохід відкритий.

Приклади “Point-and-click” квестів: Call of Cthulhu: Shadow of the Comet (1993), I Have No Mouth, and I Must Scream (1995), серія The Legend of Kyrandia (1992-1994).

1.4 Кінець 90-х у жанрі та зародження піджанру платформерів

У 90-х у жанрі квестів стались зміни не тільки через створення піджанру “Point-and-click” квестів. У кінці десятиліття японець Шіґеру Міямото, працюючи у Nintendo, розроблює Super Mario 64 (1996) [5]. Це була одна з перших розроблених відеоігор у жанрі платформер – що дослівно перекладається, як гра у якій треба переміщатись, або стрибати по платформах. Тепер в інструментарії головоломок додалося переміщення по локаціям. Super Mario 64 взагалі стала однією з перших відеоігор з повним контролем персонажа в тривимірному просторі. Головоломки тут у більшості своєму складалися з проблем переміщення.

Приклад (з гри Super Mario 64 (1996)): треба залізти на платформу з рівнем висоти 3, коли ми на висоті 1. Стрибок переміщає нас на один рівень вгору. Треба пошукати предмети, які нам допоможуть на локації. Ми знайшли дерев’яну коробку. Штовхаємо її до платформи, де нам потрібно залізти. Стаємо на коробку – тепер наш рівень висоти 2, і стрибаємо, тим самим залазячи на потрібну платформу.

Приклади: Super Mario 64 (1996), Sonic Adventure (1998), The Legend of Zelda: Ocarina of Time (1998).

1.5 Жанр квестів у 2000-х та спад популярності жанру

У 2000-х почали свій бурхливий розвиток інші жанри відеоігор, такі як: RPG (Role Play Game), FPS (First Person Shooter), Slasher, RTS (Real Time Strategy) та багато інших. Тому класичні квести відійшли на другий план. Але їх механіки та ранішні напрацювання взяли за основу, та почали додавати у вищеназвані жанри у різних пропорціях, створюючи нові та унікальні проєкти, та цілі піджанри.

Так напрацювання квестів використовували британці з Core Design у серії відеоігор Tomb Raider (1996-2018) та американці з Naughty Dog [6] у серії Uncharted (2007-2016).

З проектів 2000-х, які цілком базувались на розв'язанні різноманітних задач або головоломок можна виділити дилогію відеоігор Portal (2008-2011) та малобюджетні ігри по типу Super Meat Boy (2010).

1.6 2010-ті для жанру та створення піджанру «Інтерактивне кіно»

У 2010-х зі зльотом інді-відеоігор (малобюджетні проєкти, бюджет яких не перевищує 5-10 млн. доларів). Так, у цей період почали з'являтися ігри, побудовані виключно на головоломках, такі як: The Witness (2016), The Unravel (2016), Brothers: A Tale of Two Sons (2013), Inside (2016).

Крім того, квести "Point-and-click" у 2010-х еволюціонували в піджанр «Інтерактивного кіно». Визначальні ознаки цього жанру: режисована за правилами дорогого кіно більша половина гри, ігровий процес часто опирається на QTE (Quick Time Event) та на вибір у діалогах, який міняє сюжет в залежності від вибору (ця особливість взята з текстових квестів 70-х та 80-х). Ще важливою частиною ігрового процесу є невеличкі локації, в яких ти вирішуєш деякі

невеличкі головоломки. Ось приклад цілих студій побудованих на жанрі «Інтерактивного кіно»: Telltale [7] з своїми серіями відеоігор: The Walking Dead (2012-2018), The Wolf Among Us (2013-2014), Game of Thrones (2014-2015). Слід зазначити, що розробники з Telltale обрали «серіальну» форму розвитку своїх проєктів, тобто вони випускали по 10-15 епізодів своїх ігор у рік, але значно меншого масштабу, ніж у інших студій. Ігри цієї студії мають графіку, дуже наближену до стилю коміксів та графічних романів, динамічну та яскраву анімацію, сюжети часто взяті з графічних новел.

2 ВИКОРИСТОВУВАНІ ПРОГРАМНІ ЗАСОБИ ТА ТЕХНОЛОГІЇ

2.1 Мова програмування C#

Розроблена Андерсом Гейлсбергом, Пітером Гольде та Скотом Вілтамутом у 2001 році мова програмування C# є об'єктно-орієнтованою мовою програмування зі статичною типізацією.

C# належить до сімейства мов із C-подібним синтаксисом, і їх синтаксис найбільш близький до C++ і Java. Мова підтримує поліморфізм, перевантаження операторів (включаючи явні та неявні оператори перетворення типу), делегати, властивості, події, змінні, атрибути, узагальнені типи та методи, ітератори, анонімні функції, LINQ, винятки, формат XML коментарів.

2.2 Знайомство з Unity

Unity – це багатофункціональний рушій для розробки відеоігор різних жанрів та різноманітних застосунків. Unity є багатоплатформенний засобом, що підтримує розробку на Mac, Windows, Linux, Xbox One, Playstation 4, Nintendo Switch, iOS, Android. Також розробники рушію Unity Technologies регулярно оновлюють його та додають підтримку таких технологій як DirectX та OpenGL [8].

Вибір мови програмування C# для розробки проекту пояснюється можливістю розробки в Unity тільки на мові C#. Хоч рушій і написаний на C++, підтримка програмування скриптів на будь-якій іншій мові, крім C# неможлива.

Unity має велику популярність серед розробників-початківців. Але це не означає, що крупнобюджетних проєктів на Unity не програмують. Приклади

дорогих релізів з використанням Unity: Cuphead (2017), Firewatch (2014), Genshin Impact (2020).

2.3 Інтерфейс та створення скриптів у Unity

Інтерфейс у Unity його сильна частина. Він легко налаштовується під себе при бажанні. При розробленні програмісти часто використовують середовища розробки, типу Visual Studio (який і використовувався для програмування проєкту). Слід зазначити, що Microsoft Visual Studio має засоби для розроблення під Unity. Усі нові зміни для проекту миттєво додаються у середовище, а при зміні скриптів, під час спроби переходу у вікно Unity, починається недовге завантаження змін у проєкт.

Кожен проєкт в Unity має свою колекцію ассетів (assets). Ассети – це усі сцени, префаби (prefabs), скрипти, звуки та інші збережені в проєкті речі.

Проєкт в Unity поділяється на сцени. Сцена – це збережена конфігурація тривимірних об’єктів, включаючи розташування та просунуті налаштування об’єктів.

Для створення скрипту треба тільки викликати контекстне меню правою кнопкою миші, та натиснути на кнопку “Create” -> “C# Script”.

Якщо ми створили скрипт через Unity, то наш щойно створений клас буде нащадком класу Monobehavior, створяться методи «за замовчуванням» Start та Update, тіло яких буде порожнім

Клас Monobehavior – це клас з стандартної бібліотеки Unity. Він потрібен, щоб ми могли інтегрувати скрипти в різні об’єкти на сцені.

Метод Start в свою чергу ж відповідає за стан об’єкта в момент наступного кадру після ініціалізації сцени. В Unity ще є стандартний метод з назвою Awake. Він вже відповідає за стан об’єкта в момент самої ініціалізації сцени. У ньому часто задають значення змінних у момент початку роботи програми або

початкову ініціалізацію масивів.

Метод Update у свою чергу виконується кожен кадр і не зупиняється, поки не знищити об’єкт, до якого умовно прикріплений скрипт. Тут зазвичай знаходиться основна реалізація функціоналу: переміщення, взаємодії тощо. Є ще метод FixedUpdate, який на відміну від Update спрацьовує не кожен кадр, а через фіксовану кількість часу. Тобто, якщо частота Update залежить від оптимізації або від завантаженості ресурсів комп’ютера, то FixedUpdate буде спрацьовувати на двох різних комп’ютерах однаково, незважаючи на залізо.

Є ще методи по типу OnCollisionEnter, які спрацьовують в тому випадку, коли якийсь об’єкт, що має колізію (collision – в Unity це компонент деякого об’єкту, який малює контур фігури в тривимірному просторі, та може ловити дотики цього контура з іншими об’єктами, для подальшої взаємодії), вступає в просторову взаємодію з колізією об’єкта, що теж має даний скрипт. Метод OnCollisionExit спрацьовує при виході із колізії об’єкта.

Ще є деякі стандартні методи Unity, які часто використовуються. Одне з найчастіше використовуваних перетворень (transform) дозволяє програмісту змінювати розташування, обертання та масштаб об’єкта.

Щоб об’єкт взаємодіяв зі скриптом, цей скрипт треба прикріпити до панелі “Inspector” об’єкта.

2.4 Префаби у Unity

Одним з використовуваних в Unity компонентів є префаб.

Префаб (prefab) – зразок об’єкта, який має свої унікальні характеристики. Також можна сказати, що префаб є екземпляром унікальної конфігурації об’єкта, яка може бути повторно редагована.

Префаби можна додавати в нові сцени та на них можна посилатися програмно.

Щоб створити префаб, його треба зберегти в колекції ассетів.

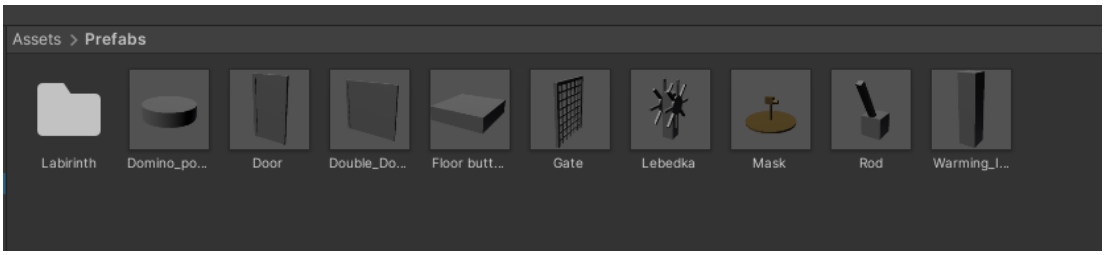


Рисунок 2.1 – Набір префабів у колекції ассетів

3 СЦЕНАРІЙ РОЗРОБЛЮВАНОЇ ГРИ-КВЕСТУ

У розроблюваній грі-квесті один головний герой має переміщуватись по послідовності локацій у тривимірному просторі, розв'язуючи розміщені в ньому головоломки.

Далі розглянемо типи головоломок, що підтримуються, та алгоритми генерації логічної частини разом з їх загальними характеристиками.

3.1 Головоломка «Кольорові кнопки»

Розроблена у рамках цього проекту головоломка «кольорові кнопки» виглядає, як розфарбована у два кольори квадратна підлога розмірів 3x3, поділена на квадрати (рис. 3.1). Насправді ж на підлозі лежать кольорові кнопки, які при натисканні на них міняють колір на протилежний.

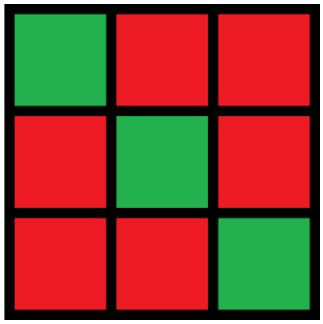


Рисунок 3.1 – Схематичний вигляд головоломки «кольорові кнопки»

У початковий момент часу гравець знаходиться у визначеній клітині підлоги. Далі він може пересуватись підлогою, переходячи на сусідню клітину по горизонталі або вертикалі. Коли гравець наступає на клітину, вона змінює свій колір на протилежний.

Головоломка вважається повністю розв'язаною, коли вся підлога виявиться одного фіксованого кольору.

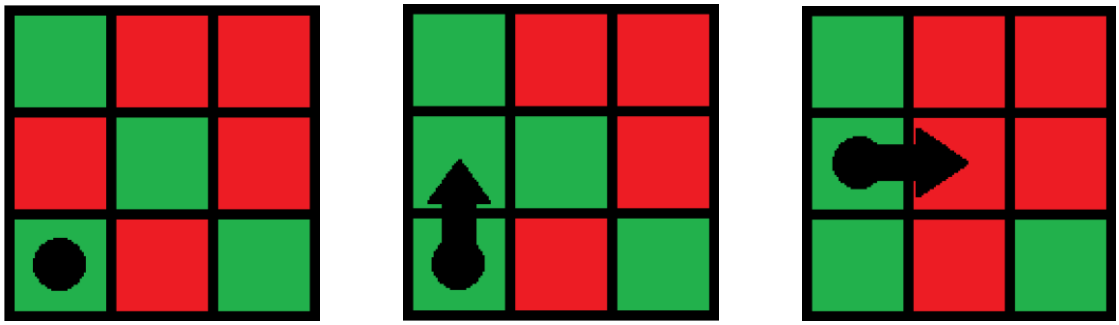


Рисунок 3.2 – Принцип дії головоломки «кольорові кнопки»

Генерація різних варіантів цієї головоломки може бути виконана шляхом присвоєння випадкових кольорів різним кнопкам підлоги та випадковим вибором позиції гравця.

Під конфігурацією цієї головоломки будемо мати на увазі розфарбування підлоги та позицію гравця. Заключними будемо вважати конфігурації, в яких головоломка є повністю розв'язаною.

Без обмеження загальності можна вважати, що кольори кодуються числами 0 та 1, а в заключній конфігурації усі квадрати підлоги мають бути кольору 1.

Оцінимо можливу кількість початкових конфігурацій цієї головоломки для підлоги розмірів $n \times n$.

Без урахування симетрій та поворотів маємо, що підлогу можна розфарбувати 2^{n^2} способами, а далі ще існує n^2 способів розмістити на ній гравця. Отже, для цієї головоломки загалом існує $2^{n^2} \cdot n^2$ різних початкових конфігурацій, що для $n = 3$ становить 4608 формально різних конфігурацій.

Якщо підійти до цієї задачі з позицій того, що деякі початкові конфігурації підлоги можуть бути отримані за допомогою поворотів чи симетрії, тобто будуть мати однакові стратегії розв'язання з точністю до повороту чи симетрії, а тому можуть викликати відчуття дежавю в заядлих гравців і їм буде не дуже цікаво,

то оцінимо скільки можна отримати різних варіантів розфарбування підлоги, якщо розфарбування, що отримуються однією операцією повороту чи відзеркалювання, будемо вважати однаковими.

Маємо: підлога може не змінюватись при відзеркалюванні відносно горизонтальної середини, вертикальної середини та двох діагоналей, а також при поворотах на 90, 180 та 270 градусів. Тому один спосіб розфарбування підлоги шляхом відзеркалень та поворотів може дати 8 формально різних розфарбувань підлоги. За різних початкових розташувань гравця на підлозі іноді можуть отримуватись головоломки, що мають однакові стратегії розв'язання з точністю до симетрії чи повороту. Наприклад, коли розфарбування є симетричним відносно горизонталі, то для кожного початкового розташування гравця симетрична позиція буде еквівалентна з точки зору стратегії. Тому, дуже сильно загрублюючи оцінку, будемо вважати, що початкова позиція гравця знаходиться в трикутнику, що займає 1/8 площі підлоги та утворений діагоналлю, вертикальною серединою та його сторонами (рис. 3.3).

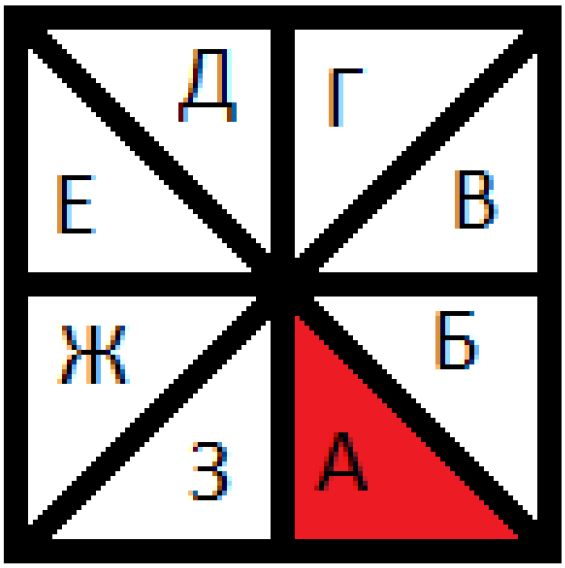


Рисунок 3.3 – Діапазон можливих початкових станів, при врахуванні симетрії, та поворотів

Тоді початкова позиція може приймати $\frac{k(k+1)}{2}$ значення, де $k = \left[\frac{n-1}{2}\right] + 1$.

Отже, загалом маємо, що існує $2^{n^2-3} \cdot \frac{k(k+1)}{2}$ розрізнюваних гравцем

початкових конфігурацій. Для $n = 3$ отримуємо дуже грубу оцінку в $2^{n^2-3} \cdot 3 = 132$ розрізнювані гравцем конфігурації, розраховану для вельми симетричної підлоги. Більш точна оцінка буде більшою, оскільки таких симетричних підлог не дуже багато.

Оцінимо розв'язуваність головоломки. Тут я буду доводити той факт, що ця головоломка розв’язувана при будь-якій конфігурації та при будь-якій формі.

Припустимо ми отримали задачу, яку на перший погляд не зрозуміло, як розв’язувати (рис. 3.4).

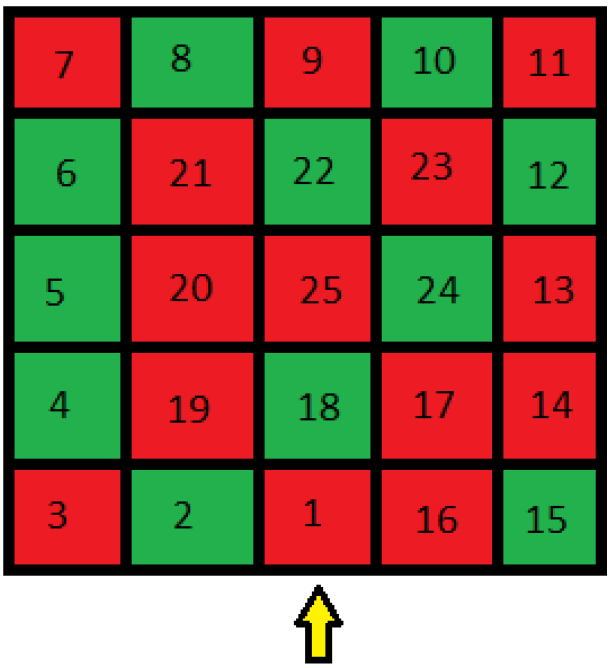


Рисунок 3.4 – Постановка більш складної головоломки

Почнемо з умовного позначення ланцюжка, за яким нам буде легше пересуватись. Намалюємо числа на клітинках для більшої зручності. Тепер в нас вималювався чіткий ланцюг (1, 2, 3, …, 25), який обходить по спіралі обходить кожну кнопку. Таким чином буде легше зображувати перехід від клітинки до клітинки, бо їх без втрати сенсу головоломки можна умовно поставити в лінію за порядком, нічого у вирішенні не зміниться.

Після креслення ланцюгу можна перейти до більш локальної постановки задачі. Якщо ми станемо на клітинку «1», то вона стане зеленою, так як задача

вважається вирішеною при зафарбуванні усіх клітинок у зелений, тоді можемо залишити клітинку «1» у спокої. Далі рухаємось у клітинку «2», оскільки вона вже була зеленою, то вона при наступі стає червоною, розглядається ситуація (рис. 3.5):

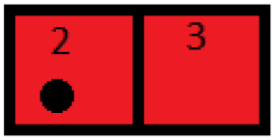
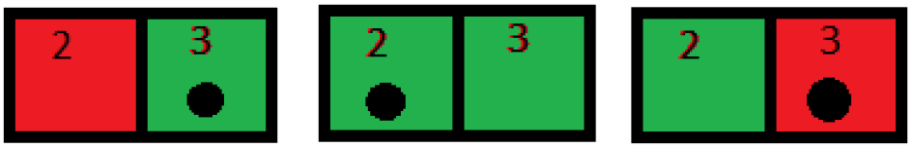


Рисунок 3.5 – Ситуація з двома червоними кнопками у головоломці

З цієї ситуації (рис. 3.5) можна легко вийти використовуючи всього одну (наступну) клітинку як допоміжну (рис. 3.6).



а) б) в)

Рисунок 3.6 – Вирішення ситуації з рис. 3.5:
а – крок 1; б – крок 2; в – крок 3

Далі помічаємо, що ми змогли, як перейти до наступної клітинки («3»), так і зафарбувати дану («2»). Далі слід вилучити з рисунку вершину «2», бо вона вже потрібного нам кольору, а додати справа вершину «4» (рис. 3.7).

Рисунок 3.7 – Ситуація з поточною червоною та наступною зеленою кнопками у головоломці

Насправді хоч ситуація на рис. 3.7 відрізняється від ситуації на рис. 3.5, але вирішуємо ми її однаковими рухами:

а) б) в)

Рисунок 3.8 – Вирішення ситуації з рис. 3.7:
а – крок 1; б – крок 2; в – крок 3

По завершенню ситуації з клітинками «3» та «4» ми зробили «3» зеленою та перейшли до «4», але оскільки по завершенню і вона стала зеленою, то можем одразу переходити до ситуації з клітинками «5» та «6» (рис. 3.9).

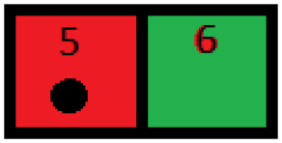


Рисунок 3.9 – Ситуація з клітинками «5» та «6»

Але можна побачити марну трату часу та вирішення ситуації з рис. 3.9, бо вона ідентична до ситуації з рис. 3.8. Недовго виводячи відповідь, можна вивести повний розв’язок таких головоломок:

1. Побудувати ланцюг, що буде проходити усі кнопки без виключення (наприклад, по спіралі чи змійкою).
2. Починаючи з першого елементу створеного ланцюга перетворювати усі клітинки на зелені по черзі за принципом: якщо клітинка зелена – йдемо далі по ланцюгу, якщо клітинка червона – то беремо наступну кнопку у ланцюжку як допоміжну і вирішуємо згідно описаних на рис. 3.5 та рис. 3.7 ситуацій.

А якщо в нас головоломка не нагадує квадрат, повністю набитий кнопками? Тут наша головоломка теж буде розв’язна. За приклад візьмемо зразок головоломки з рис. 3.10:

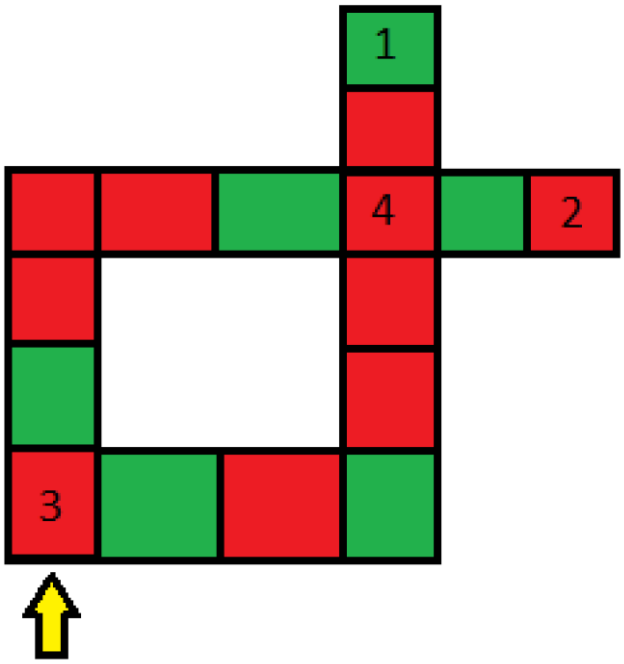


Рисунок 3.10 – Приклад головоломки, де складно побудували умовний ланцюг

Ситуація з рис. 3.10 на перший погляд нерозв’язна, якщо вирішувати її алгоритмом розв’язку, який я навів вище. Тут треба прокрокувати до клітинки «4». Дійшовши до клітинки «4», треба піти в один з тупиків, нехай це буде кнопка «1». Тепер стоячи на кнопці «1», ми повинні побудувати умовний ланцюг від клітинки «1» до клітинки «4». Побудувавши ланцюг ми можемо почати по ньому рухатись згідно виведеного вище розв’язку. Таким чином в нас будуть зелені кнопки від кнопки «1» до «4». Тепер таким само чином ми можемо перетворити усі кнопки від «2» до «4» на зелені. Після попередній операції в нас залишається незафарбованим квадрат з пустою серединою. Його можна зафарбувати у зелений колір таким чином: переходимо у клітинку «3» та будуємо ланцюг по периметру проходячи через клітинку «4». Потім починаємо проходити ланцюг згідно розв’язання – головоломка вирішилася.

Тому можна додати у остаточний алгоритм розв’язання додати важливу правку:

1. Побудувати ланцюг, що буде проходити усі кнопки без виключення,

або розбити задачу на будування декількох ланцюгів, які у купі будуть покривати усі кнопки і вирішувати задачу поетапно.

2. Починаючи з першого елементу створеного ланцюга перетворювати усі клітинки на зелені по черзі за принципом: якщо клітинка зелена – йдемо далі по ланцюгу, якщо клітинка червона – то беремо наступну кнопку у ланцюжку як допоміжну і вирішуємо згідно описаних на рис. 3.5 та рис. 3.7 ситуацій.

3.2 Головоломка «Відбиваючі лазери»

В якості ідеї для головоломки з відбиваючими елементами під назвою «Відбиваючі лазери» було взято головоломку з променем сонця з ігор Witcher 3: Wild Hunt (2015) та Hogwarts: Legacy (2023). [9]

Головоломка «Відбиваючі лазери» складається з елементів трьох типів: стін, лазерів та теплових елементів, які працюють за такими правилами:

а) активний лазер можна повертати;

б) активний лазер може випромінювати промінь світла;

в) якщо промінь світла потрапляє в стіну, то він гасне;

г) якщо промінь світла потрапляє в інший лазер, то той лазер активується, а лазер, який випустив промінь, деактивується;

д) якщо промінь світла попадає в тепловий елемент, то головоломка вважається вирішеною.

Конфігурація локації, кількість та розташування в ній стін та лазерів, а також розташування теплового елементу, потенціально можуть бути довільними. У поточній реалізації за основу для генерації екземпляра головоломки взято схему, в якій:

— локація має форму квадрата, поділеного на клітини;

— лазери розставлено по периметру кімнати у центрах відповідних клітин;

— тепловий елемент є квадратним та знаходиться в середині кімнати;

— стіни розставлено між лазерами перпендикулярно відповідним сторонам кімнати, а також у центрі кімнати, залишаючи невелику щілину для доступу до теплового елементу.

Приклад можливого облаштування локації наведено на рис. 3.11. Стіни позначено рисками, кола позначають лазери, активний лазер позначено зеленим, тепловий елемент позначено червоним.

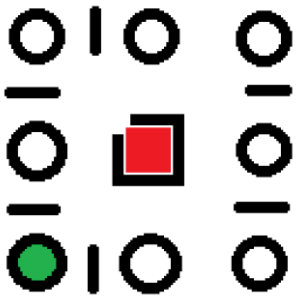


Рисунок 3.11 – Схематичний вигляд головоломки «Відбиваючі лазери»

За потрапляння променю в стіну, він гасне.

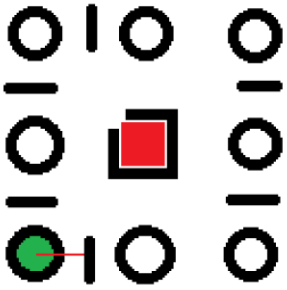
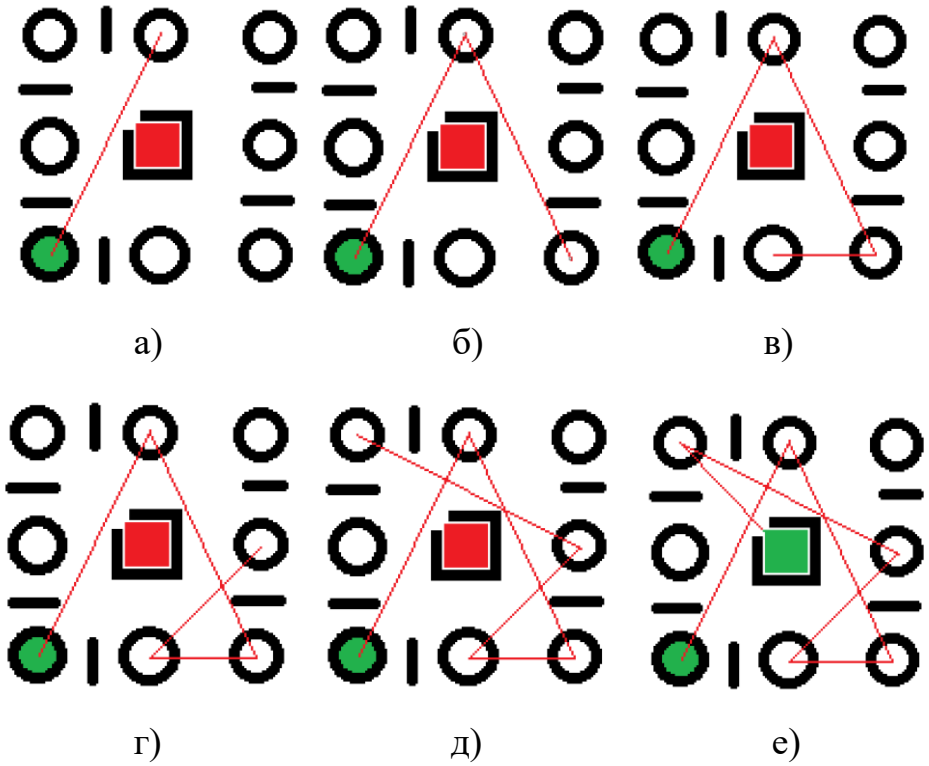


Рисунок 3.12 – Випущений промінь стикнувся зі стіною

На наступному рисунку схематично показано можливі кроки (послідовність випущених лазерами променів) розв'язання прикладу цієї головоломки.



а) б) в)

г) д) е)

Рисунок 3.13 – Розв'язок екземпляру головоломки «Відбиваючі лазери»: а – крок 1; б – крок 2; в – крок 3; г – крок 4; д – крок 5; е – крок 6

Алгоритм генерації головоломки «Відбиваючі лазери» можна описати таким чином:

Нехай кімната має розміри $n \times n$, n непарне, лазери розставляємо тільки у цілочислових клітинах уздовж стін, а тепловий елемент у центральній клітині. Тоді кількість лазерів k можна подати формулою $k = 4n - 4$.

Максимальна можлива кількість стін, що визначаються як сторони клітин (крім стін зовнішнього контуру кімнати) у такому випадку дорівнює $2n^2 - 2n$.

Тепер для генерування головоломки, котру буде хоч якась складність розв’язувати, треба додати перешкоди. Спочатку слід знайти ланцюжок з «переможної» комбінації для головоломки, ми її будемо вважати єдиним правильним розв’язком та по ньому будувати перешкоди так, щоб вони не заважали «правильному розв’язку».

Спробуємо знайти кількість правильних розв’язків, додавши значення для складності головоломки, наприклад: для легких головоломок кількість елементів

у розв’язку буде становити 10% від кількості лазерів, для головоломки середньої складності – 15%, для складних – 20%.

Тепер перекладаємо на значення: $p_{лег} = [\frac{k}{10}]$, $p_{сер} = [\frac{3k}{20}]$, $p_{скл} = [\frac{k}{5}]$.

Кількість унікальних розв’язків позначимо як r = A_k^p, бо при перестановках одних і тих же елементів картинка розв’язку буде різна.

Зазначимо, що деякі такі розв'язки будуть проходити через центральну клітину і утикатись у стіну, а тому розв'язками не будуть. Трохи округлюючи, можна стверджувати, що за фіксованого початку розв'язку вибір $(i + 1)$-го лазеру після вибору перших i гарантовано можна здійснити $(n - i - 1)$ способом, а тому якнайменш $k \cdot A_{n-1}^{p-1}$ розв'язків з p лазерів існуватиме.

Далі треба додати перешкоди, тут можна теж зробити вибір складності: для легких головоломок кількість перешкод буде становити 10% від кількості місць для перешкод, для головоломки середньої складності – 15%, для складних – 20%.

Місця перешкод, це кожне вільне місце між двома елементами, які стоять поруч. Оскільки елементів у нас всього n^2, то кількість місць для перешкод позначимо через t = $2n^2 - 2n$.

Тепер перекладаймо на значення позначивши кількість перешкод на головоломці через m: $m_{лег} = [\frac{t}{10}]$, $m_{сер} = [\frac{3t}{20}]$, $m_{скл} = [\frac{t}{5}]$.

Кількість унікальних комбінацій розставлення перешкод по головоломці буде: q = C_t^m.

Тепер ми можемо вивести загальну формулу для кількості унікальних головоломок (деякі серед них можуть не бути розв'язуваними) просто перемноживши q та r:

$$Res = C_t^m \times A_k^p$$

$$Res = C_{2n^2-2n}^m \times A_{4n-4}^p$$

Тепер виведемо для кожної складності:

$Res_{лег} = C_{2n^2-2n}^{[\frac{2n^2-2n}{10}]} \times A_{4n-4}^{[\frac{4n-4}{10}]}$

$\text{Res}_{\text{сер}} = C_{2n^2-2n}^{[\frac{3(2n^2-2n)}{20}]} \times A_{4n-4}^{[\frac{3(4n-4)}{20}]}$

$\text{Res}_{\text{скл}} = C_{2n^2-2n}^{[\frac{2n^2-2n}{5}]} \times A_{4n-4}^{[\frac{4n-4}{5}]}$

Щоб дати нарешті, хоч якесь представлення, порахуймо різні рівні складності для головоломки зі стороною 3:

Легких = 528

Середніх = 27 720

Складних = 1 552 320

Припустимо, що розставлено всі можливі стіни. Відрізок, що сполучає вершини цілочислової сітки, що розташовані на відстані x по горизонталі та y по вертикалі, може перетинати не більше $x + y \leq 2n - 2$ стін. Тоді маршрут, що з'єднує $p = p_1(4n - 4)$ лазерів, може перетинати не більше $(2n - 2)p$ стін. Саме ці стіни не можна встановлювати, щоб маршрут залишався розв'язком головоломки. Тоді вільними для встановлення є якнайменш $2n^2 - 2n - (2n - 2)p$ стін. Якщо треба розставити $t = t_1 m = t_1(2n^2 - 2n)$ перешкод, то це можна зробити за умови:

$$2n^2 - 2n - (2n - 2)p_1(4n - 4) \geq t_1(2n^2 - 2n)$$

Звідки

$$n^2(1 - 4p_1 - t_1) - n(1 - t_1) + (4n - 2)p_1 \geq 0$$

Якщо p_1=50%, t_1=50%, то за великих n наявність розв'язків для кожного маршруту дещо сумнівна. Через це найбільше значення складності для обох показників не перевищує 20%.

Слід зазначити, що алгоритм генерації описаний таким чином, що згенерована головоломка буде завжди розв'язною, оскільки кожного разу додаючи перешкоду перевіряємо на розв'язність.

3.3 Головоломка «Лабіринт»

По суті головоломка «Лабіринт» є класичною головоломкою, суть якої полягає в тім, щоб подолати згенерований набір перешкод та знайти вихід з лабіринту.

Нехай лабіринт — прямокутник розмірів $n \times m$ розбито на nm одиничних квадратів лініями, паралельними його сторонам. Побудуємо граф клітин, множиною вершин якого є центри утворених квадратів. Дві вершини суміжні, якщо відповідні квадрати мають спільну сторону. Виберемо дві клітини уздовж сторін квадрату: це буде вхід та вихід у лабіринт, уздовж решти одиничних квадратів та зовнішнім світом відразу розставимо стіни. Розв'язком лабіринту буде найкоротший маршрут, що сполучає вхід та вихід, тоді він буде простим ланцюгом побудованого графа. Щоб гарантувати єдиність розв'язку лабіринту, достатньо зробити так, щоб з кожної вершину в кожну існував єдиний маршрут, що є ланцюгом. Таку властивість мають дерева. Якщо розглянути кістякове дерево, то воно і буде задавати лабіринт: якщо ребро початкового графа не входить у кістякове дерево, то тоді між відповідними клітинами поставимо стіну.

Отже, задача зводиться до побудови кістякового дерева. Її можна реалізувати різними способами. Для уникнення доволі специфічних кістяків, які утворюються алгоритмами пошуку в глибину та ширину, можна адаптувати алгоритм Прима.

Алгоритм Прима працює так. Маємо масив точок, які потрібно зв’язати. Обираємо початкову точку. Далі починаємо додавати ребра найменшої ваги між точками, які ми відвідали, та точками, які збираємось відвідати.

Для нашої задачі можна трохи спростити правила:

- Оскільки точки в нас розташовані рівновіддалено, тому знаходження найменшої відстані між двома вершинами не має сенсу, слід перевіряти доступність сусідніх точок зверху, знизу, зліва, справа.

- З вже відвіданих точок при приєднанні нових можна ігнорувати точки, що не мають невідвіданих сусідів.
- У той самий час можна з ще невідвіданих точок ігнорувати ті, що не мають відвіданих сусідів і додавати до потенційних кандидатів на сусіди при наявності вже відвіданого сусіда.

Вхід алгоритму: набір незв’язаних точок.

Вихід алгоритму: зв’язний граф.

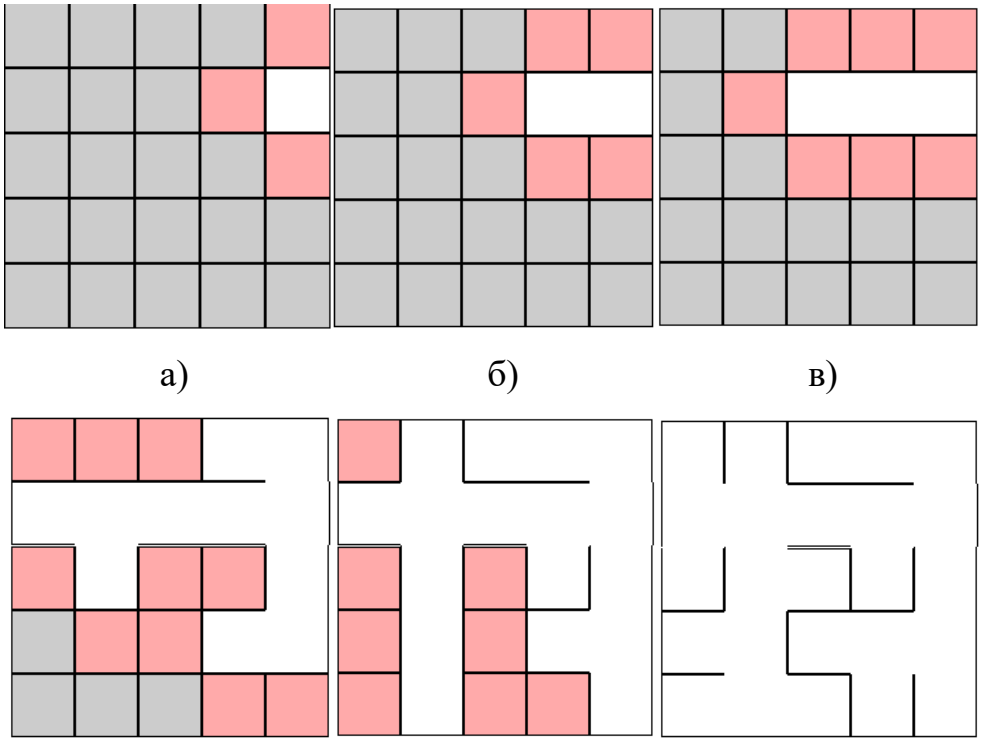
У процесі роботи використовуються два масиви точок: Explored - масив відвіданих точок, та Current - масив невідвіданих точок, що мають відвіданих сусідів).

Початкова ініціалізація: в Explored ми запишемо будь-яку початкову точку (нехай це буде (0, 0)), а от в Current ми ставимо прилежні, до початкової точки (тут це буде (0,1) та (1,0)).

Алгоритм:

1. Обираємо випадкову вершину А з Current.
2. Обираємо випадкову вершину В з Explored, що є сусідом до А.
3. Будуємо ребро від А до В (додаємо в масив B.next вершину А).
4. Видаляємо А з Current і додаємо Explored.
5. Додаємо в Current усі унікальні прилежні вершини до А.
6. Повторюємо з 1-го кроку поки не відвідаємо усі вершини.
7. Генеруємо заготовку під лабіринт: кожній вершині відповідає своя закрита комірка (Рис 4.19-4.24).
8. Маючи зв’язний граф, починаємо проходити усі вершини і видаляти стіни у місцях, де вершини з’єднані ребрами.

Розв’язність лабіринту перевіряється дуже легко: у кінці роботи алгоритму ми маємо зв’язний ациклічний граф, тобто з будь якої вершини можна дістатись у будь-яку іншу. Ациклічність дозволяє зробити висновок, що шлях від будь-якої точки А у будь-яку точку Б, якщо не наступати на одні й ті ж точки більше одного разу, буде єдиним.



а) б) в)

г) д) е)

Рисунок 3.15 – Створення головоломки «лабіринт»:
а – крок 1; б – крок 2; в – крок 3; г – крок t; д – крок k; е – останній крок

Тому ми можемо просто поставити вхід та вихід лабіринту у межах нашого лабіринту і він буде розв’язний.

Тепер спробуймо розглянути кількість унікальних лабіринтів при розмірі лабіринту $n \times n$ клітинок.

Граф, для якого будуємо кістяк має $2n(n-1)$ ребер та n^2 вершин. У кістяк має увійти $n^2 - 1$ ребро. Кількість різних способів вибрати $n^2 - 1$ ребро дорівнює $C_{2n(n-1)}^{n^2-1}$. Але ж не кожен вибір ребер дасть кістяк, тому ця оцінка є дуже грубою верхньою межею. Зокрема, для $n = 3$ ця величина дорівнює 495.

Нехай лабіринт квадратний з стороною $n = 2k$. Тоді відповідний граф містить k^2 незалежних простих циклів довжини 4. Для утворення кістяка з кожного такого циклу треба вилучити принаймні одне ребро, що можна зробити 4 способами. Тому дуже груба оцінка дає, що кількість різних кістяків, а отже і лабіринтів, буде не менша за 4^{k^2}.

Міркування можна узагальнити на випадок непарної сторони лабіринту: гарантовано, що різних розв'язних лабіринтів розмірів $n \times n$ (без урахування вибору точки входу та точки виходу) буде не менше 4^{k^2}, де $k = \left[\frac{n}{2}\right]$.

4 АРХІТЕКТУРА ТА РЕАЛІЗАЦІЯ РОЗРОБЛЮВАНОГО ДОДАТКУ

4.1 Короткий опис сценарію та архітектури додатку

Розроблюваний в рамках кваліфікаційної роботи проект являє собою гру в жанрі квест-головоломка. Гра полягає у вирішенні різноманітних головоломок: пошуку найбільш зручного розв’язання чи розв'язання взагалі.

Архітектура розроблюваної гри в більшості будується на об’єктах взаємодії. Діаграма класів наявних у грі взаємодій знаходиться у (додатку Ж). Послідовність локацій з головоломками будується таким чином, що за успішного проходження головоломки, вона "відкриває двері" до наступного випробування. Тобто головоломки мають якийсь об’єкт типу Вхід (Entry) як змінну, яку вони повинні відкрити при проходженні. При створенні головоломки ми повинні обрати один з нащадків класу Вхід та поставити у відповідність до головоломки.

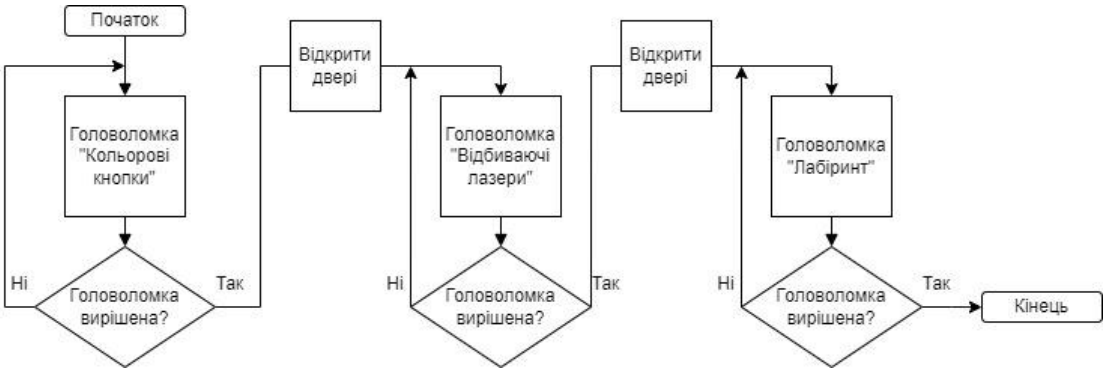


Рисунок 4.1 – Блок схема послідовності вирішення головоломок

4.2 Головний герой

Гра має вигляд від першої особи. Через це детальної моделі в головного героя немає. Головний герой керується через клас Player_Controller. Реалізація

камери ж знаходиться у класі Camera_Controller.

Клас Player_Controller реалізує пересування персонажа. В залежності від натиснутих клавіш програма ставить направлення для подальшого пересування, та персонаж починає рухатись. При чому реалізований рух не тільки у 4-х напрямах, а у восьми. І через те, що вектор направлення «вперед» (1,0,0) за довжиною менше за вектор направлення «вперед і вправо» (1,0,1) додана нормалізація вектору, для того, щоб персонаж не прискорювався рухаючись по діагоналі.

У свою чергу у класі Camera_Controller реалізовані повороти камери, в залежності від рухів миші гравця. Для достатнього огляду гравцю достатньо осей X та Y. Оскільки вісь X відповідає за повороти вправо та наліво, її обмежувати немає сенсу. А от вісь Y обмежити доведеться, бо якщо це не зробити, голова головного героя зможе крутитись наче колесо, а це не відповідає діям звичайної людини. Тому було додано обмеження (-80;+80) для вісь Y, щоб камера зупинялася, при погляді на небо, та при спробі детально роздивитись підлогу під ногами.

4.3 Прості об’єкти взаємодії

Прості взаємодії у грі поділяються на два класи:

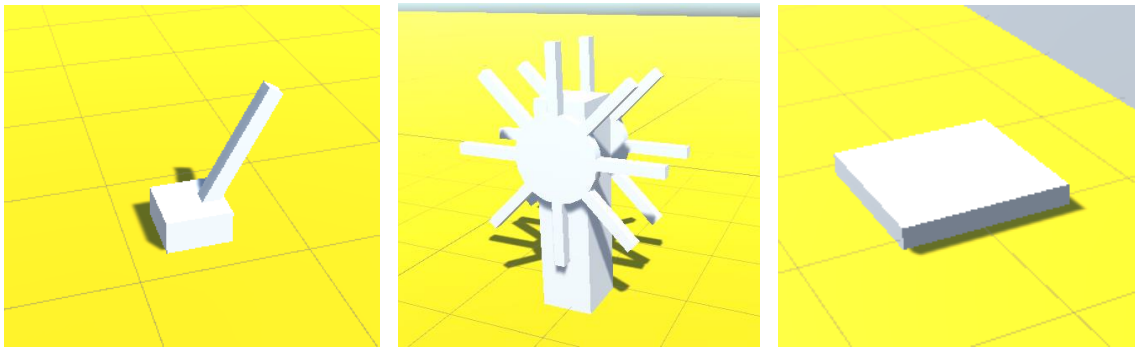
- Входи відносяться до класу Entry_controller. Потомками класу Entry_controller є двері (Door_controller), подвійні двері (Double_Door_controller), ворота (Gate_controller).

```
public abstract class Entry_controller : MonoBehaviour
    {
        public abstract void Open();
        public abstract void Close();
    }
```

Входи мають два абстрактні методи: відкриття (Open) та закриття (Close). Наприклад у класу двері (Door_controller) у методі відкриття ручка разом з основною частиною дверей обертаються на 90 градусів і зупиняються. На закриття ж вони обертаються до початкового стану і зупиняються.

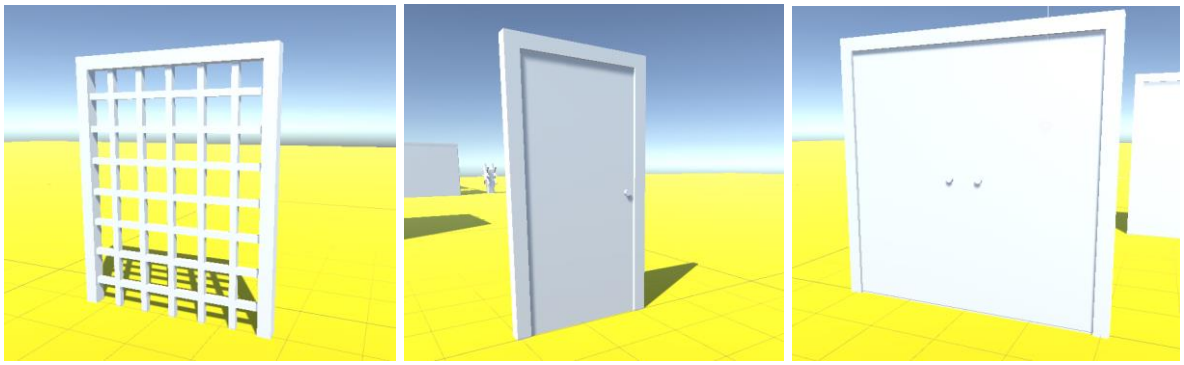
Відкривачі відносяться до класу Unlockers_controller. Потомками класу Unlockers_controller є рушій (Rod_Controller), лебідка (Lebedka_Controller), кнопка (Flour_Color_button). Також як діти класу Unlockers_controller виступають головоломки: Кольорові кнопки (Button_Puzzle), Відбиваючі лазери (Lazer_Puzzle) та Лабіринт (Maze_Puzzle), але їх опис буде в наступних розділах.

Відкривачі мають всього одну логічну змінну Is_activated, яку повинні змінювати при активації. Наприклад лебідка (Lebedka_Controller) при взаємодії швидко обертається до стану активації, а потім в залежності від зазначеного часу, рухається у початковий стан, при досягненні якого стає неактивним.



а) б) в)

Рисунок 4.2 – Відкривачі: а – рушій; б – лебідка; в – кнопка



а) б) в)

Рисунок 4.3 – Входи: г: ворота; д: двері; ж: подвійні двері

Клас Entry_and_Unlock ставить у відповідність одному входу один відкривач.

Цей клас відповідає, щоб кожен кадр перевірявся відкривач на активність, щоб відкрити, або закрити вхід відповідно.

4.4 Реалізація головоломки «Кольорові кнопки»

За керування цією головоломкою відповідає клас Button_Puzzle (Дод. А). Він відповідає за перевірку кольорів усіх кнопок і у випадку, коли всі кнопки мають зелений колір, відкриває двері до наступного випробування.

Слід додати, що як об’єкти взаємодії тут виступають звичайні кнопки, запрограмовані у класі Flour_Color_button. Він відповідає за зміну кольору кнопки на протилежний (червоний <–> зелений). Також він відповідає за незначну зміну позиції кнопки, приспускаючи її трохи вниз. Таким чином з’являється ефект натискання на кнопку.

Також слід зазначити, що клас Button_Puzzle ще відповідає за генерацію випадкового кольору для кожної кнопки, тим самим генеруючи велику кількість різних варіантів головоломки.

4.5 Реалізація головоломки «Відбиваючі лазери»

Цю головоломку запрограмовано у класі Lazer_Puzzle (Дод. Б) та классах Lazer_Controller, Lazer_Ineract_Items, Warming_Items.

Клас Lazer_Puzzle крім усього відповідає за початкову конфігурацію. Тим більше, на сцені у Unity можна своїми руками додавати та вилучати лазери, таким чином роблячи свою унікальну головоломку.

Клас Lazer_Ineract_Items (Дод. В) е батьківським для Lazer_Controller (Дод. Г) та Warming_Items (Дод. Д). Lazer_Ineract_Items має два абстрактних методи, активація та дезактивація.

Для Lazer_Controller активація полягає у випусканні лазера перед собою з подальшими варіантами взаємодії лазера з навколишнім середовищем. Кожен кадр лазер збільшується, та перевіряє об’єкт перед собою:

Якщо там такий самий лазер (Lazer_Controller), то він викликає метод активації у нього, і наступний лазер теж починає рухатись.

Якщо ж там тепловий елемент (Warming_Items), то він викликає метод активації у нього, тепловий елемент міняє колір та позначається активованим при повному зафарбуванні чскраво червоним.

Якщо ж ми зтикнулися не з потомком класу Lazer_Ineract_Items, то лазер просто зупиняється.

Lazer_Puzzle будучи нащадком класу Unlockers_controller об’єднує усіх нащадків класу Lazer_Ineract_Items та при активації усіх теплових елементів відкриває двері до наступної головоломки.

Генерація різних варіантів головоломки теж реалізоване у класі Lazer_Puzzle.

4.6 Реалізація головоломки «Лабіринт»

Головоломка «лабіринт» полягає у тому, щоб подолати згенерований набір перешкод та знайти вихід. Ця головоломка запрограмована у класі Maze_Controller (Дод. Е).

Клас Maze_Controller створює з префабів Maze_module та Wall_module лабіринт. Вони розміщуються так, щоб кожна клітина у лабіринті буде закрита з усіх сторін. Далі будуємо зв’язний граф жадібним алгоритмом схожим на алгоритм Прима. А потім проходячи по зв’язному графу починаємо видаляти тільки стіни префабів Maze_module ігноруючи елементи Wall_module і видаляємо усі стіни, які перетинаються з ребрами графа.

4.7 Огляд наявних результатів гри

Цей розділ присвячений огляду спроектованих головоломок та як це виглядає у внутрішньому середовищі Unity.

На рис. 4.4 показано головоломку «Кольорові кнопки» у варіанті 3×3. Вхід у неї знаходиться тільки з одної клітинки.

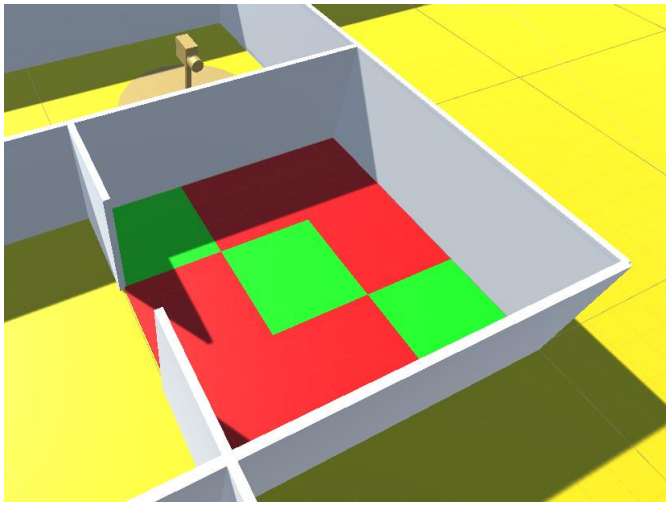


Рис. 4.4 – Вигляд зверху головоломки «Кольорові кнопки» (варіант 3×3)

Далі можна навести приклад з головоломкою «Кольорові кнопки» у варіанті 4 × 4 (рис. 4.5).

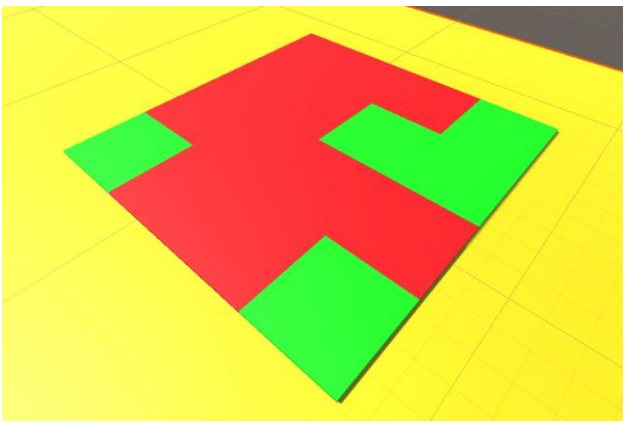
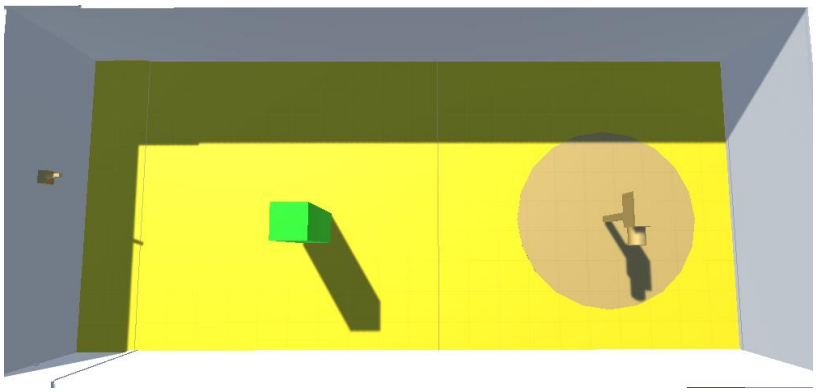
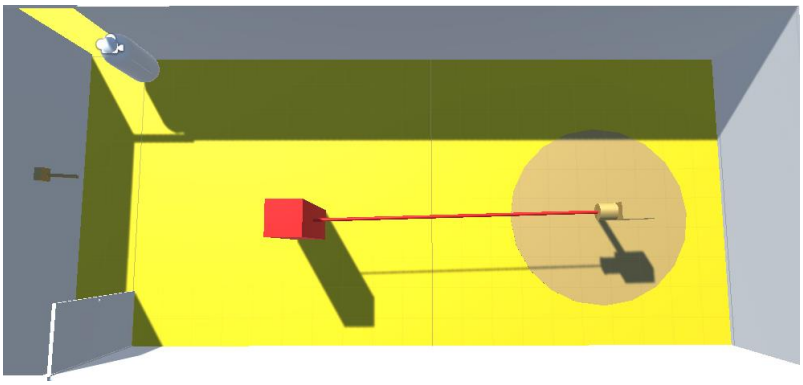


Рисунок 4.5 – Вигляд зверху головоломки «Кольорові кнопки» (варіант 4 × 4)

Тепер перейдемо до головоломки «Відбиваючі лазери». Було спроєктовано простенький навчальний рівень для розуміння роботи головоломки (рис. 4.6).



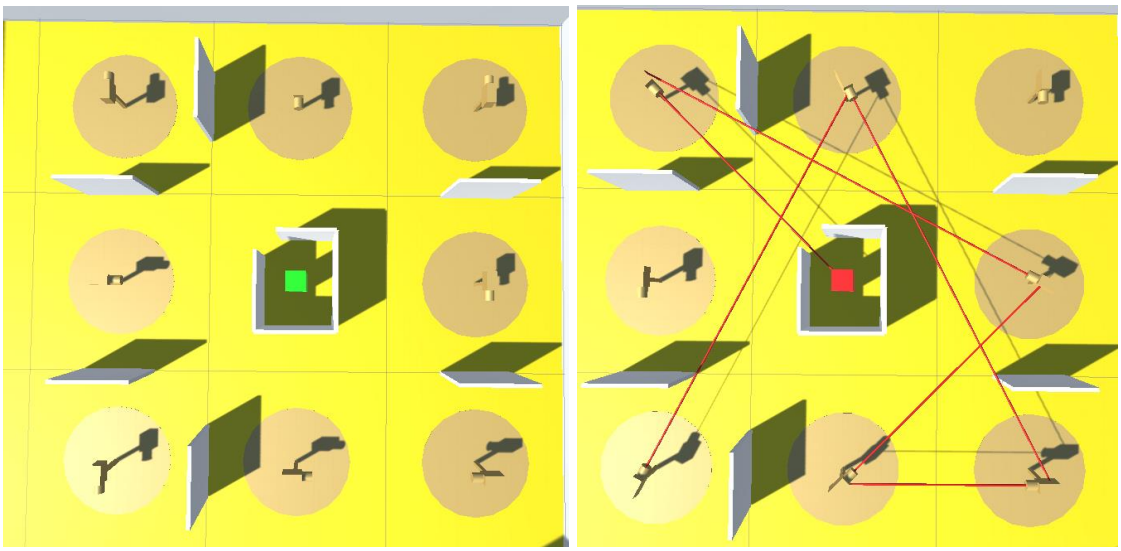
а)



б)

Рисунок 4.6 – Вигляд зверху головоломки «Відбиваючі лазери» (навчальний варіант): а – невирішена; б – вирішена

Далі слід показати вигляд вже повноцінної версії головоломки «Відбиваючі лазери». Кількість лазерів буде 8, один тепловий елемент (рис 4.7).



а) б)

Рисунок 4.7 – Вигляд зверху головоломки «Відбиваючі лазери» (варіант 3 × 3): а – невирішена; б – вирішена

Згенеруємо до головоломки у прикладі на рис. 4.7 інші перешкоди (рис. 4.8).

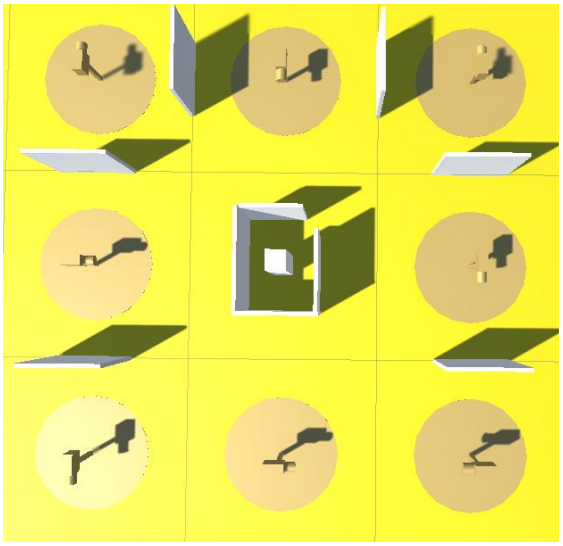


Рисунок 4.8 – Вигляд згори на головоломку «Відбиваючі лазери» в іншому варіанті (3 × 3)

Тепер перейдемо до вигляду головоломки «Лабіринт». Згенеруємо

лабіринт у вигляді 5 × 5 клітин (Рис. 4.9).

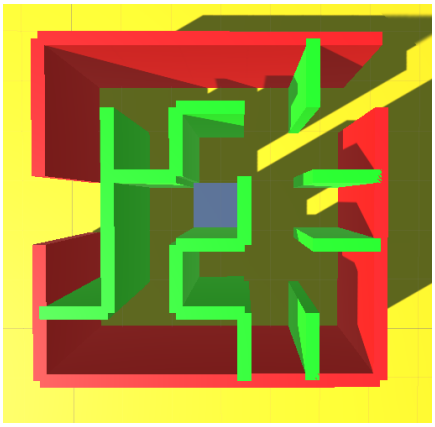
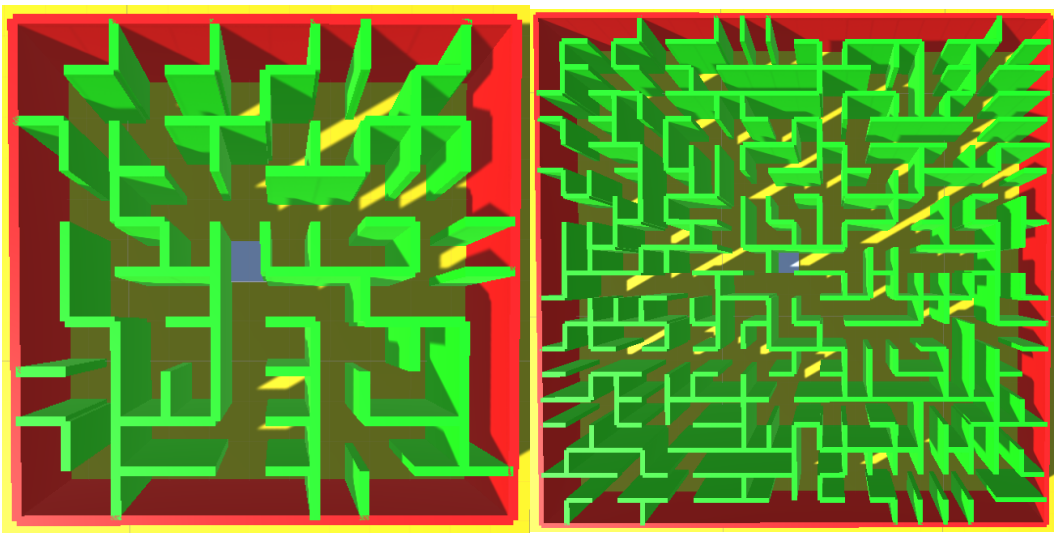


Рисунок 4.9 – Вигляд зверху головоломки «Лабіринт» (варіант 5 × 5)

Лабіринт на рис. 4.9 не виглядає достатньо складним, та це можна пояснити невеликою кількістю клітинок. Спробуймо згенерувати лабіринти у варіантах 10 × 10 та 20 × 20.



а) б)

Рисунок 4.10 – Вигляд зверху головоломки «Лабіринт»:
а – варіант 10 × 10; б – варіант 20 × 20

Варіанти головоломки на 10 × 10 та 20 × 20 вже виглядають більш складно та більш придатні для того, щоб вставити у гру ніж варіант 5 × 5.

Тепер, після огляду кожної головоломки слід вкінці показати як виглядає увесь рівень цілком. Він зображений на рис. 4.11.

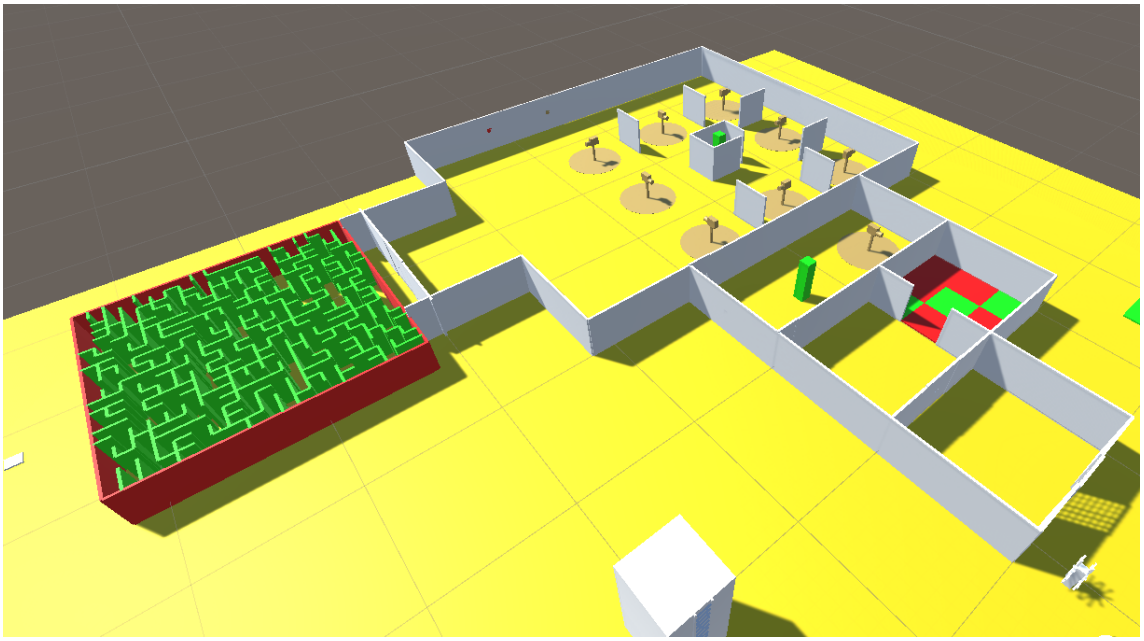


Рисунок 4.11 – Вигляд рівня з головоломками цілком

ВИСНОВКИ

Під час виконання бакалаврської роботи було розроблено гру в жанрі квест, зокрема було запрограмоване керування персонажа від першої особи та переміщення персонажа по віртуальному світу.

Була створена велика кількість екземплярів об’єктів взаємодії, що використовувалися у подальшому проєктуванні головоломок.

При проєктуванні головоломки «Кольорові кнопки» була розроблена ідея, про проходження головоломки за найменшу кількість ходів.

Під час проєктування головоломки «Відбиваючі лазери» було реалізовано головоломку з лазерами, що повинні попасти на тепловий елемент, відбиваючись один від одного. За приклад головоломки була взята реалізація головоломки з відеоігор Witcher 3: Wild Hunt (2015) та Hogwarts: Legacy (2023).

При проєктуванні головоломки «Лабіринт» було обрано та адаптовано алгоритм Прима для генерації лабіринту.

Для розроблених для гри головоломок було виконано грубі оцінки скільки різних варіантів завдань можна згенерувати та досліджено розв'язність.

Можливості для покращення гри можуть бути наступними:

- Додати випадкову генерацію ланцюгів з головоломками з постійним нарощуванням складності.
- Додати нові шаблони головоломок.
- Розробити навчальний рівень для швидкого ознайомлення з грою.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. O3DE: motion matching in O3DE, a data-driven animation technique [Електронний ресурс] // Open 3D Engine. – Режим доступу: https://www.o3de.org/blog/posts/blog-motionmatching/#:~:text=Motion%20matching%20is%20a%20way,the%20animation%20for%20the%20character (дата звернення: 01.06.2023). – Назва з екрана.

2. GSC game world [Електронний ресурс] // GSC Game World. – Режим доступу: https://www.gsc-game.com/ (дата звернення: 01.06.2023). – Назва з екрана.

3. Home - TaleWorlds Entertainment [Електронний ресурс] // Home - TaleWorlds Entertainment. – Режим доступу: https://www.taleworlds.com/. (дата звернення: 01.06.2023). – Назва з екрана.

4. Westwood studios [Електронний ресурс] // Command & Conquer Communications Center. – Режим доступу: https://cnc-comm.com/westwood-studios (дата звернення: 01.06.2023). – Назва з екрана.

5. Super mario 64 [Електронний ресурс] // Nintendo of Europe GmbH. – Режим доступу: https://www.nintendo.co.uk/Games/Nintendo-64/Super-Mario-64-269745.html (дата звернення: 01.06.2023). – Назва з екрана.

6. Naughty dog [Електронний ресурс] // Naughty Dog. – Режим доступу: https://www.naughtydog.com/ (дата звернення: 01.06.2023). – Назва з екрана.

7. Telltale games [Електронний ресурс] // Telltale Games. – Режим доступу: https://telltale.com/ (дата звернення: 01.06.2023). – Назва з екрана.

8. Unity real-time development platform | 3D, 2D, VR & AR engine [Електронний ресурс] // Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine. – Режим доступу: https://unity.com/ (дата звернення: 01.06.2023). – Назва з екрана.

9. Carr J. Hogwarts legacy - all puzzle types and solutions [Електронний

ресурс] / James Carr // GameSpot. – Режим доступу: https://www.gamespot.com/articles/hogwarts-legacy-all-puzzle-types-and-solutions/1100-6511440/ (дата звернення: 01.06.2023). – Назва з екрана.

ДОДАТОК А
КОД КЛАСУ BUTTON_PUZZLE

```
public class Button_Puzzle : Unlockers_controller
{
    private Flour_Color_button[] Puzzle_Buttons = new
Flour_Color_button[9];
    void Start()
    {
        Is_activated = false;
        for (int i = 0; i < 9; i++)
        {
            Puzzle_Buttons[i] =
transform.GetChild(i).GetComponent<Flour_Color_button>();
        }
    }

    void Update()
    {
        if (Is_Green())
        {
            Is_activated = true;
        }
    }

    private bool Is_Green()
    {
        foreach (var f in Puzzle_Buttons)
        {
            if (f.GetComponent<Renderer>().material.color == Color.red)
                return false;
        }
        return true;    }    }
```

ДОДАТОК Б
КОД КЛАСУ LAZER_PUZZLE

```
public class Lazer_Puzzle : Unlockers_controller
{
    Warming_Items W_I;
    void Start()
    {
        Is_activated = false;
        for (int i = 0; i < transform.childCount; i++)
        {
            if (transform.GetChild(i).GetComponent<Warming_Items>())
            {
                W_I =
transform.GetChild(i).GetComponent<Warming_Items>();
            }
        }
    }

    void Update()
    {
        if (W_I.IS_ACTIVATED == true)
        {
            Is_activated = true;
        }
        if (W_I.IS_ACTIVATED == false)
        {
            Is_activated = false;
        }
    }
}
```

ДОДАТОК В
КОД КЛАСУ LAZER_INERACT_ITEMS

```
namespace Assets.Scripts
{
    public abstract class Lazer_Ineract_Items : MonoBehaviour
    {
        public abstract void Activate();
        public abstract void Disactivate();
    }
}
```

ДОДАТОК Г
КОД КЛАСУ LAZER_CONTROLLER

```
public class Lazer_Controller : Lazer_Ineract_Items
{
    [SerializeField] private Unlockers_controller _unlocker;
    private GameObject Lazer;
    private Mask_Follow_Contloller Mask;
    private bool Is_Ready_To_Unlock = false;
    private bool Is_Unlock = false;
    private bool Is_Into = false;
    private float _cur_lazer_dist = 0.1f;
    [SerializeField] private float speed = 1;
    [SerializeField] private float _max_lazer_dist = 5;
    void Start()
    {
        if (_unlocker == null)
        {
            Is_Ready_To_Unlock = true;
        }
        Mask = transform.GetComponent<Mask_Follow_Contloller>();
        Lazer = transform.Find("Lazer").gameObject;
        Lazer.GetComponent<Renderer>().material.color = Color.red;
        Lazer.transform.position += 1 * transform.forward;
    }

    void Update()
    {
        if (Is_Unlock == true || _unlocker?.Is_activated == true)
        {
            Mask.Lock();
            if ((Is_Into == false) && (_cur_lazer_dist <
_max_lazer_dist))
```

```
            {
                _cur_lazer_dist += Time.deltaTime * speed;
            }
            if (Physics.Raycast(Lazer.transform.position,
transform.forward * _cur_lazer_dist,
                out var hit, _cur_lazer_dist, -1,
QueryTriggerInteraction.Ignore))
            {
                Is_Into = true;
                if (transform.name != hit.transform.name)
                {
                    var NextLazer =
hit.transform.gameObject.GetComponent<Lazer_Ineract_Items>();
                    NextLazer?.Activate();
                }
            }
        }
        else
        {
            if (Physics.Raycast(Lazer.transform.position,
transform.forward * _cur_lazer_dist,
                out var hit, _cur_lazer_dist, -1,
QueryTriggerInteraction.Ignore))
            {
                if (transform.name != hit.transform.name)
                {
                    var NextLazer =
hit.transform.gameObject.GetComponent<Lazer_Ineract_Items>();
                    NextLazer?.Disactivate();
                }
            }
            Is_Into = false;
            Mask.Unlock();
            _cur_lazer_dist = 0.1f;
```

```
        }
        Lazer.transform.position = transform.position + _cur_lazer_dist *
transform.forward;
        Lazer.transform.localScale = new Vector3(0.1f, _cur_lazer_dist,
0.1f);
    }

    public override void Activate()
    {
        if (Is_Ready_To_Unlock == true)
        {
            Is_Unlock = true;
        }
    }

    public override void Disactivate()
    {
        if (Is_Ready_To_Unlock == true)
        {
            Is_Unlock = false;
        }
    }
}
```

ДОДАТОК Д
КОД КЛАСУ WARMING_ITEMS

```
public class Warming_Items : Lazer_Ineract_Items
{
    public bool IS_ACTIVATED { get; protected set; }
    private float cur_color = 1;
    [SerializeField] private float speed = 0.2f;
    private bool Is_Activated = false;
    private Material R_material;
    void Start()
    {
        IS_ACTIVATED = false;
        R_material = transform.GetComponent<Renderer>().material;
        R_material.color = new Color(1 - cur_color, cur_color, 0);
    }

    void Update()
    {
        if (Is_Activated == true && cur_color > 0)
        {
            cur_color -= Time.deltaTime * speed;
        }
        else if (Is_Activated == false && cur_color < 1)
        {
            cur_color += Time.deltaTime * speed;
        }
        R_material.color = new Color(1 - cur_color, cur_color, 0);
        if (cur_color <= 0)
        {
            IS_ACTIVATED = true;
        }
        else
```

```
        {
            IS_ACTIVATED = false;
        }
    }

    public override void Activate()
    {
        Is_Activated = true;
    }

    public override void Disactivate()
    {
        Is_Activated = false;
    }
}
```

ДОДАТОК Е
КОД КЛАСУ MAZE_CONTROLLER

```
public class Maze_Controller : MonoBehaviour
{
    [SerializeField] private Maze_module_flag Maze_Prefab;
    [SerializeField] private GameObject Wall_Prefab;
    [SerializeField] private int Lenght_ofLab = 5;
    private List<List<(float, float)>> World_Map = new List<List<(float,
float)>>();
    private float Height = 2;
    void Start()
    {
        Height = Maze_Prefab.gameObject.transform.localScale.z;
        Initialize_World_Map();
        Physics.SyncTransforms();
        Prims_Alg();
    }

    void Initialize_World_Map()
    {
        (float, float) start_cell = (-(((Lenght_ofLab - 1) / 2) *
Height), -(((Lenght_ofLab - 1) / 2) * Height));
        (float, float) cur_cell = start_cell;
        for (int i = 0; i < Lenght_ofLab; i++)
        {
            cur_cell.Item2 = start_cell.Item2;
            List<(float, float)> Fake_Map = new List<(float, float)>();
            for (int j = 0; j < Lenght_ofLab; j++)
            {
                Fake_Map.Add(cur_cell);
                cur_cell.Item2 += Height;
            }
```

```
            World_Map.Add(Fake_Map);
            cur_cell.Item1 += Height;
        }
        for (int i = 0; i < Lenght_ofLab - 1; i++)
        {
            for (int j = 0; j < Lenght_ofLab; j++)
            {
                var instance = Instantiate(Maze_Prefab);
                instance.transform.position = transform.position + new
Vector3(
                    World_Map[i][j].Item1 + Height / 2,
                    1.5f,
                    World_Map[i][j].Item2);
            }
        }
        for (int i = 0; i < Lenght_ofLab; i++)
        {
            for (int j = 0; j < Lenght_ofLab - 1; j++)
            {
                var instance = Instantiate(Maze_Prefab);
                instance.transform.position = transform.position + new
Vector3(
                    World_Map[i][j].Item1,
                    1.5f,
                    World_Map[i][j].Item2 + Height / 2);
                instance.transform.Rotate(0, 90, 0);
            }
        }
        for (int i = 0; i < Lenght_ofLab; i++)
        {
            var instance = Instantiate(Wall_Prefab);
            instance.transform.position = transform.position + new
Vector3(
                World_Map[i][0].Item1,
```

```
                1.5f,
                World_Map[i][0].Item2 - Height / 2);
            instance.transform.Rotate(0, 90, 0);
            instance = Instantiate(Wall_Prefab);
            instance.transform.position = transform.position + new
Vector3(
                World_Map[i][Lenght_ofLab - 1].Item1,
                1.5f,
                World_Map[i][Lenght_ofLab - 1].Item2 + Height / 2);
            instance.transform.Rotate(0, 90, 0);
        }
        for (int j = 0; j < Lenght_ofLab; j++)
        {
            var instance = Instantiate(Wall_Prefab);
            instance.transform.position = transform.position + new
Vector3(
                World_Map[0][j].Item1 - Height / 2,
                1.5f,
                World_Map[0][j].Item2);
            instance = Instantiate(Wall_Prefab);
            instance.transform.position = transform.position + new
Vector3(
                World_Map[Lenght_ofLab - 1][j].Item1 + Height / 2,
                1.5f,
                World_Map[Lenght_ofLab - 1][j].Item2);
        }
    }

    void Prims_Alg()
    {
        List<(int, int)> Explored = new List<(int, int)>();
        List<(int, int)> Current = new List<(int, int)>();
        Explored.Add((0, 0));
        Current.Add((1, 0));
```

```
        Current.Add((0, 1));
        while (Current.Count != 0)
        {
            int rand_c = Random.Range(0, Current.Count);
            (int, int) cur_el = Current[rand_c];
            Explored.Add(cur_el);
            Current.Remove(cur_el);
            List<Vector3> L_dir = new List<Vector3>(){ new Vector3(0, 0,
1),
                new Vector3(0, 0, -1),
                new Vector3(1, 0, 0),
                new Vector3(-1, 0, 0)};
            List<Vector3> L_in = new List<Vector3>();
            for (int i = 0; i < L_dir.Count; i++)
            {
                if (Physics.Raycast(new Vector3(transform.position.x +
World_Map[cur_el.Item1][cur_el.Item2].Item1, 2, transform.position.z +
                    World_Map[cur_el.Item1][cur_el.Item2].Item2),
                    L_dir[i] * Height,
                    out var hit, Height))
                {
                    if (hit.transform.GetComponent<Maze_module_flag>())
                    {
                        if (Explored.Contains(((int)(cur_el.Item1 +
L_dir[i].x), (int)(cur_el.Item2 + L_dir[i].z))))
                        {
                            L_in.Add(L_dir[i]);
                        }
                        else
                        {
                            if (!Current.Contains(((int)(cur_el.Item1 +
L_dir[i].x), (int)(cur_el.Item2 + L_dir[i].z))))
                            {
                                Current.Add(((int)(cur_el.Item1 +
```

```
L_dir[i].x), (int)(cur_el.Item2 + L_dir[i].z)));
                                }
                            }
                        }
                    }
                }
                rand_c = Random.Range(0, L_in.Count);
                if (Physics.Raycast(new Vector3(transform.position.x +
World_Map[cur_el.Item1][cur_el.Item2].Item1, 2, transform.position.z +
                        World_Map[cur_el.Item1][cur_el.Item2].Item2),
                        L_in[rand_c] * Height,
                        out var hita, Height))
                {
                    if (hita.transform.GetComponent<Maze_module_flag>())
                    {
                        Destroy(hita.transform.gameObject);
                    }
                }
            }
        }
    }
}
```

ДОДАТОК Ж

ДІАГРАМА КЛАСІВ НАЯВНИХ У ГРІ ВЗАЄМОДІЙ