

Міністерство освіти та науки України
Національний університет "Запорізька політехніка"

кафедра програмних засобів

Пояснювальна записка
з дисципліни «Об'єктно-орієнтоване програмування»
до курсової роботи на тему
«Створення десктопної гри на «Лабіринт» на платформі Unity»

Виконав
ст. гр. КНТз-127сп

Кряжков Д.О.

Прийняв
професор
доцент .
доцент

Табунщик Г.В.
Миронова Н.О.
Каплієнко Т.І

Запоріжжя, 2020

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЗАПОРІЗЬКА ПОЛІТЕХНІКА"

Кафедра програмних засобів

Дисципліна Об'єктно-орієнтоване програмування

Спеціальність Інженерія програмного забезпечення

Курс 2 Група КНТз-127сн Семестр III

ЗАВДАННЯ

на курсовий проект (роботу) студентів

Кряжкову Дмитру Олександровичу

1. Тема проекту (роботи): Створення десктопної гри на «Лабіринт» на платформі Unity

2. Термін здачі студентом закінченого проекту (роботи): 24 квітня 2020

3. Вихідні дані до проекту: Реалізувати гру «Лабіринт» на платформі Unity з процедурною генерацією лабіринту і підказками для гравця
вхідні дані: гра «Лабіринт»
вихідні дані: Гра повинна генерувати новий лабіринт з кожним новим рівнем або с початком гри; у грі повинні бути підказки найшвидшого шляху для гравця; прості елементи користувацького інтерфейсу та основних меню;

4. ЗМІСТ розрахунково-пояснювальної записки (перелік питань, що їх належить розробити): 1 Аналіз предметної області

2 Аналіз програмних засобів

3 Основні рішення з реалізації компонентів системи

4 Посібник програміста

5 Інструкція користувача

Висновки,

Додаток А Текст програми,

Додаток Б Інтерфейс програми,

7. Перелік графічного матеріалу: Слайди презентації

8. Дата видачі завдання: 24 березня 2020

КАЛЕНДАРНИЙ ПЛАН

№ пор.	Назва етапів курсового проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1.	Аналіз індивідуального завдання	1-2 тиждень	
2.	Аналіз програмних засобів, що будуть використовуватись в роботі.	3-4 тиждень	
3.	Аналіз структур даних, що необхідно використати в курсовій роботі.	4-5 тиждень	
4.	Вивчення можливостей програмної реалізації структур даних та інтерфейсу користувача.	5-6 тиждень	
5.	Оформлення відповідних пунктів пояснювальної записки	4 тиждень	розділи 1,2 ПЗ
6.	Проміжний контроль	8 тиждень	
7.	Аналіз вимог до апаратних засобів	9 тиждень	
8.	Розробка програмного забезпечення	10-15 тиждень	
9.	Оформлення, відповідних пунктів пояснювальної записки.	9-16 тиждень	розділи 3,4,5 ПЗ
10.	Захист курсової роботи.	17 тиждень	

Студент _____

(підпис)

Керівник _____

(підпис)

(прізвище, ім'я, по батькові)

“ ____ ” _____ 2020 р.

ЗМІСТ

РЕФЕРАТ.....	6
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ,.....	7
СКОРОЧЕНЬ І ТЕРМІНІВ.....	7
ВСТУП.....	8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ.....	9
1.1 Огляд існуючих методів вирішення завдання.....	9
1.2 Огляд існуючих програмних засобів, що вирішують аналогічні завдання.....	16
1.2.2 Flow Free: мобільна гра-головоломка з механікою, яка схожа на проходження лабіринту.....	17
1.3 Постановка завдання до роботи.....	18
2 АНАЛІЗ ПРОГРАМНИХ ЗАСОБІВ	19
2.1 Огляд особливостей рушія гри	19
2.2 Топ-5 кращих ігор створених на Unity:.....	20
2.2 Огляд особливостей мови програмування.....	23
2.3 Огляд можливостей бібліотеки класів .NET.....	25
2.4 Огляд шаблонних класів бібліотеки класів .NET, що використовується в роботі	26
2.5 Огляд можливостей Scripting API у Unity Engine.	32
2.6 Висновки з розділу.....	33
3 ОСНОВНІ РІШЕННЯ З РЕАЛІЗАЦІЇ КОМПОНЕНТІВ СИСТЕМИ.....	34
3.1 Основні рішення щодо уявлення даних системи	34
3.2 Основні розроблені алгоритми	41
3.3 Основні рішення щодо модульного уявлення гри	52
3.4 Особливості реалізації системи	53
3.5 Висновки з розділу.....	55
4 ПОСІБНИК ПРОГРАМІСТА.....	56
4.1 Призначення та умови застосування програми.....	56
4.2 Характеристики гри	56

4.3 Звертання до гри.....	56
5 ІНСТРУКЦІЯ КОРИСТУВАЧА	58
5.1 Призначення програми	58
5.2 Умови виконання гри.....	58
5.3 Як запустити програму	58
5.4 Виконання програми.....	58
ВИСНОВКИ	61
ПЕРЕЛІК ПОСИЛАНЬ.....	62
ДОДАТОК А Текст програми	63
ДОДАТОК Б Інтерфейс гри	73

РЕФЕРАТ

ПЗ: 74стор., 26 рис., 10 табл., 2 додатки, 7 джерел.

Метою даного курсового проекту є розробка гри з використанням алгоритмів процедурної генерації лабіринтів.

Було виконано аналіз предметної області, розглянуто аналогічні існуючі методи та програмні засоби для вирішення завдання.

Для реалізації програмного продукту використовувалася мова C# та багатоплатформний інструмент Unity 2019.3.7f1. У якості середовища розробки та налагодження коду використовувався текстовий редактор Visual Studio Code та IDE Visual Studio Community 2019.

При розробці гри були розроблені такі класи та описані їх методи: MenuControl, CameraController, Cell, HintRenderer, Maze, MazeGeneratorCell, MazeGenerator, MazeSpawner, PlayerMovement, Timer.

У процесі виконання курсового проекту були розглянуті особливості редактору Unity, основні функціональні можливості, а також Scripting API. Також при виконанні проекту були розглянуті основні можливості, синтаксис та шаблонні класи мови C#.

Під час створення проекту також були розглянуті можливості створення внутрішнього ігрового користувацького інтерфейсу (GUI) за допомогою таких інструментів як: Canvas, Button, Text, Panel та інших.

Основна увага у курсовому проекті приділяється використанню алгоритму для процедурної генерації лабіринту, а саме Recursive backtracker та гнучким умовам для масштабування проекту при подальшій розробці.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ООП — об'єктно-орієнтоване програмування;

ПЗ — програмне забезпечення;

Graphical User Interface, GUI — графічний програмний інтерфейс.

Application Programming Interface, API — прикладний програмний інтерфейс.

Integrated development environment, IDE — інтегроване середовище розробки.

Artificial intelligence, AI — штучний інтелект.

ВСТУП

Метою даного курсового проекту є розробка гри «Лабіринт» з використанням алгоритмів процедурної генерації клітинок та стін лабіринту. Також однією з умов розробки стала наявність підказок найшвидшого шляху для гравців.

Дана гра є однією з ігор категорії time-killer'ів, умови гри дуже складні, кожного разу генерується новий лабіринт, що є превентивною мірою заучування шляхів. Рівень важкості підвищується за рахунок того, що у гравця є тільки 60 секунд на те, щоб вийти з лабіринту.

В процесі виконання роботи було виконано аналіз та огляд існуючих рішень, таких як мобільна гра *Mazes & More* — класична головоломка з лабіринтами, веселими налаштуваннями та сюрпризами. Мінімальна двовимірна графіка створює атмосферу класичної ретро-гри, а нові режими гри підтримують інтерес користувача (а іноді й лякають гравця). Гра *Flow Free* описує себе як просту головоломку, яка викликає звикання.

Актуальність роботи полягає в тому, що ніша ігор-головоломок ще зовсім не заповнена, а користувачам кортить кинути виклик своїй кмітливості.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд існуючих методів вирішення завдання

Гра-лабіринт побудована таким шляхом, щоб кожен зміг перевірити свої розумові навички та інтуїцію. Для того щоб гравець не нудьгував в, розроблений під час виконання курсового проекту, грі передбачена процедурна генерація лабіринту за допомогою алгоритму побудови.

Існує дуже багато видів алгоритмів та форм лабіринтів. Для курсового проекту було обрано стандартну квадратну або двомірну форму та алгоритм побудови Recursive backtracker.

Для огляду вирішення завдання треба звернутися до класифікації, способів генерації та знаходження рішень для лабіринтів.

Лабіринти в цілому (а отже і алгоритми для їх створення) можна розбити за сімома різними класифікаціями: розмірності, топології, теселяції, маршрутизації, текстури та пріоритету. Лабіринт може використовувати по одному елементу з кожного класу в будь-якому поєднанні.

Розмірність: клас розмірності по суті визначає, скільки вимірювань в просторі заповнює лабіринт. Існують наступні типи:

- 1) двомірні: більшість лабіринтів, як паперових, так і реальних, мають цю розмірність, тобто ми завжди можемо відобразити план лабіринту на аркуші паперу і рухатися ним, не перетинаючи ніяких інших коридорів лабіринту.
- 2) тривимірні: тривимірний лабіринт має кілька рівнів; в ньому проходи можуть крім чотирьох сторін світу опускатися вниз і підніматися вгору. 3D-лабіринт часто візуалізують як масив з 2D-рівнів з позначками сходів «вгору» і «вниз».
- 3) переплетення: лабіринти з переплетеннями — це, по суті двомірні (або, точніше 2,5-мірні) лабіринти, в яких, проте, проходи можуть накладатися один на одного. При відображенні зазвичай цілком очевидно, де знаходяться тупики та як один прохід знаходиться над іншим. Лабіринти з реального світу, в яких є мости, що з'єднують одну частину лабіринту з іншого, частково є переплетеннями.

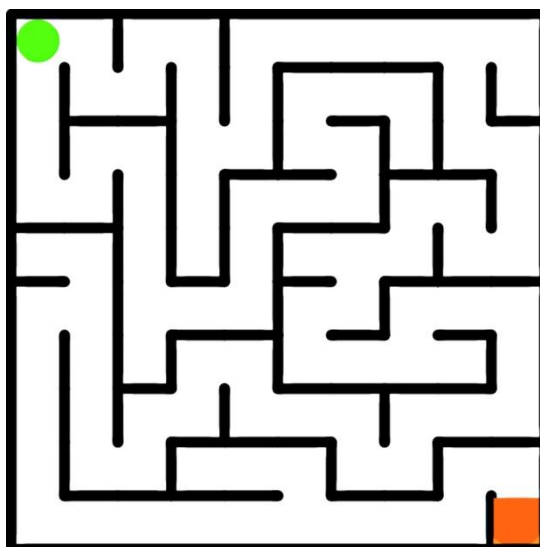


Рисунок 1.1 – Приклад двомірного лабіринту

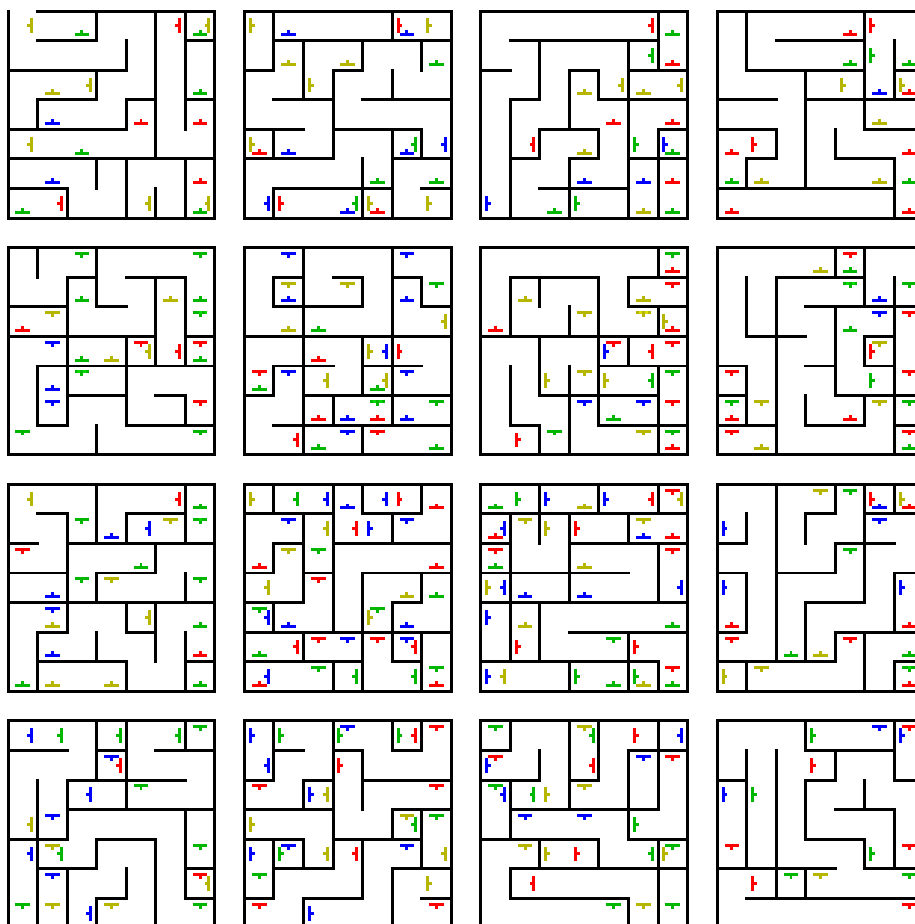


Рисунок 1.2 – Приклад тривимірного лабіринту

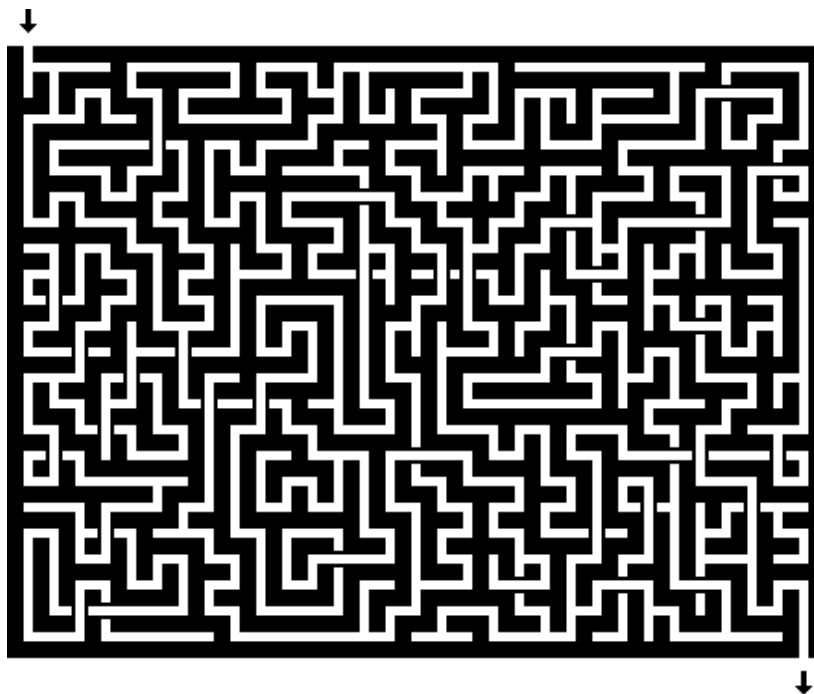


Рисунок 1.3 – Приклад лабіринту з переплетінням

У різних видів лабіринтів є своя топологія: клас топології описує геометрію простору лабіринту, в якому той існує як ціле. Є такі типи:

- 1) звичайний: це стандартний лабіринт в евклідовому просторі. Приклад звичайної топології можна побачити на рисунку 1.1.
- 2) planair: термін «planair» описує будь-лабіринт з незвичайною топологією. Зазвичай це означає, що края лабіринту з'єднані незвичним чином. Приклади: лабіринти на поверхні куба, лабіринти на поверхні стрічки Мебіуса і лабіринти, еквівалентні тим, що знаходяться на торі, де попарно з'єднані ліва і права, верхня і нижня сторони.

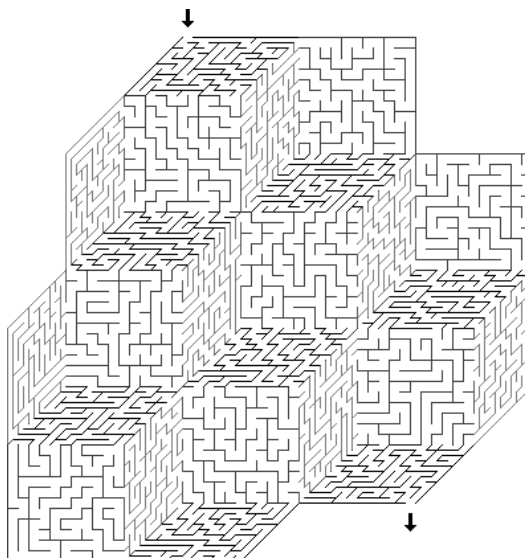


Рисунок 1.4 – Приклад топології Planair

Також види лабіринтів може відрізнати теселяція. Теселяція — класифікація геометрії окремих осередків, з яких складається лабіринт.

Існуючі типи:

- 1) ортогональний: це стандартна прямокутна сітка, в якій осередки мають проходи, що перетинаються під прямими кутами. В контексті теселяції їх також можна назвати гамма-лабіринтами. Приклад ортогональної теселяції можна побачити на рисунку 1.1.
- 2) дельта: дельта-лабіринти складаються із сполучених трикутників, у кожного осередку може бути до трьох з'єднаних з нею проходів.
- 3) сігма: сігма-лабіринти складені зі сполучених шестикутників; у кожного осередку може бути до шести проходів.
- 4) тета: тета-лабіринти складаються з концентричних кіл проходів, в яких початок або кінець знаходиться в центрі, а інший — на зовнішньому краї. Осередки зазвичай мають чотири можливі з'єднання проходів, але їх може бути і більше завдяки більшій кількості осередків в зовнішніх кільцях проходів.
- 5) іпсилон: іпсилон-лабіринти складаються із сполучених восьмикутників або квадратів, в них кожна клітинка може мати до восьми або чотирьох проходів.

- 6) дзета: дзета-лабіринт розташований на прямокутній сітці, тільки на додаток до горизонтальних і вертикальних проходів допускаються діагональні проходи під кутом 45 градусів.
- 7) омега: термін «омега» відноситься практично до будь-якого лабіринту з постійною неортогональною теселяцією. Дельта-, сигма-, тета-і іпсилон-лабіринти відносяться до цього типу, як і багато інших схем, які можна придумати, наприклад, лабіринт, що складається з пар прямокутних трикутників.
- 8) скаск: скаск-лабіринт — це аморфний лабіринт без постійної теселяції, в якому стіни і проходи розташовані під випадковими кутами.
- 9) фрактальний: фрактальний лабіринт — це лабіринт, складений з менших лабіринтів. Фрактальний лабіринт з вкладених осередків — це лабіринт, в кожному осередку якого розміщені інші лабіринти, і цей процес може повторюватися кілька разів. Нескінченно рекурсивний фрактальний лабіринт — це справжній фрактал, в якому вміст лабіринту копіює себе, створюючи по суті нескінченно великий лабіринт.

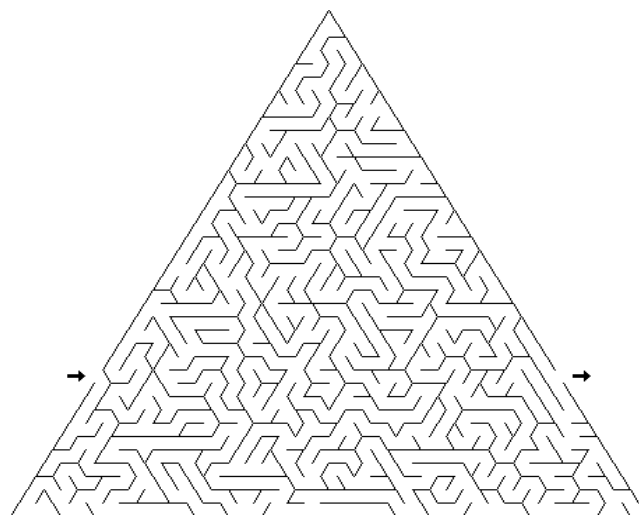


Рисунок 1.5 – Приклад Дельта теселяції

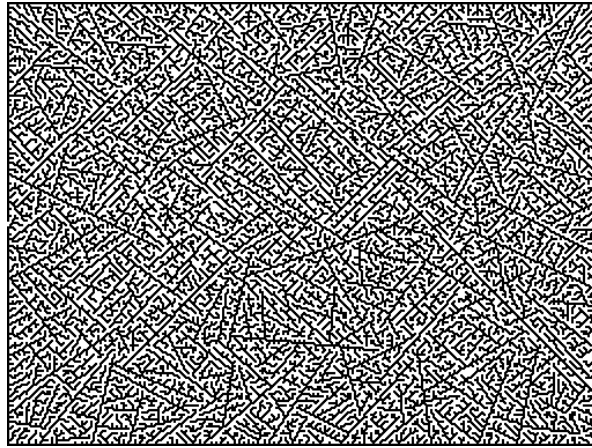


Рисунок 1.6 – Приклад stack-лабіринту

Для створення ідеальних лабіринтів є безліч алгоритмів зі своїми недоліками та перевагами. Серед них найвідоміші:

- 1) Recursive backtracker: він у чомусь схожий на метод вирішення лабіринтів recursive backtracker, і вимагає стеку, обсяг якого може доходити до розмірів лабіринту. При вирізанні він поводить себе максимально жадібно, і завжди вирізає прохід в нествореній частині, якщо вона існує поруч з поточним осередком. Кожен раз, коли ми подорожуємо до нового осередку, записуємо попередній осередок в стек. Якщо поруч з поточною позицією немає нестворених осередків, то витягаємо зі стека попередню позицію. Лабіринт завершений, коли в стеці більше нічого не залишається. Це призводить до створення лабіринтів з максимальним показником плинності, тупиків менше, але вони довші, а рішення зазвичай виявляється дуже довгим і звивистим. При правильній реалізації він виконується швидко (швидше працюють тільки дуже спеціалізовані алгоритми). Recursive backtracking не може працювати з додаванням стін, тому що зазвичай призводить до шляхів вирішення, що слідує зовнішнім краєм, коли вся внутрішня частина лабіринту з'єднана з кордоном одним проходом.
- 2) алгоритм Краскала: це алгоритм, який створює мінімальне сполучне дерево. Це цікаво, тому що він не «виросує» лабіринт подібно дереву, а скоріше вирізає сегменти проходів по всьому лабіринту випадковим чином, і тим не менш в результаті створює в кінці ідеальний лабіринт. Для його роботи необхідний обсяг пам'яті, пропорційний розміру

лабіринту, а також можливість перерахування кожного ребра або стіни між осередками лабіринту у випадковому порядку (зазвичай для цього створюється список усіх ребер і перемішується випадковим чином). Помічаємо кожну клітинку унікальним ідентифікатором, а потім обходимо всі ребра у випадковому порядку. Якщо осередки по обидва боки від кожного ребра мають різні ідентифікатори, то видаляємо стіну і задаємо всім осередкам з одного боку той самий ідентифікатор, що і осередкам з іншого. Якщо осередки на обох сторонах стіни вже мають однаковий ідентифікатор, то між ними вже існує якийсь шлях, тому стіну можна залишити, щоб не створювати петель. Цей алгоритм створює лабіринти з низьким показником плинності, але не таким низьким, як у алгоритму Прима. Об'єднання двох множин по обидва боки стіни буде повільною операцією, якщо у кожного осередку є тільки номер і вони об'єднуються в цикл. Об'єднання, а також пошук можна виконувати майже за постійний час завдяки використанню алгоритму об'єднання-пошуку (union-find algorithm): поміщаємо кожну клітинку в деревоподібну структуру, кореневих елементом є ідентифікатор. Об'єднання виконується швидко завдяки зрощенню двох дерев. При правильній реалізації цей алгоритм працює досить швидко, але повільніше більшості через створення списку ребер і управління множинами.

- 3) Алгоритм Олдоса-Бродера: цікаво в цьому алгоритмі те, що він однорідний, тобто він з однаковою ймовірністю створює всі можливі лабіринти заданого розміру. Крім того, йому не потребує додаткової пам'яті або стека. Вибираємо точку і випадковим чином переміщуємось в сусідню клітинку. Якщо ми потрапили в невирізану комірку, то врізаємо в неї прохід з попередньої комірки. Продовжуємо рухатися в сусідні осередки, поки не виріжемо проходи в усі осередки. Цей алгоритм створює лабіринти з низьким показником плинності, всього трохи вище, ніж у алгоритму Краскала. (Це означає, що для заданого розміру існує більше лабіринтів з низьким показником плинності, ніж з високим, тому що лабіринт із середньою однаковою ймовірністю має низький показник.) Погано в цьому алгоритмі те, що він дуже повільний, тому що не виконує інтелектуального пошуку останніх осередків, тобто по суті не

має гарантій завершення. Однак через свою простоту він може швидко проходити безлічню осередків, тому завершується швидше, ніж можна було б подумати. В середньому його виконання займає в сім разів більше часу, ніж у стандартних алгоритмів, хоча в поганих випадках воно може бути набагато більше, якщо генератор випадкових чисел постійно уникає останніх кількох осередків. Він може бути реалізований як той що додає стіни, якщо стіну кордону вважати єдиної вершиною, тобто, наприклад, якщо хід переміщує нас до стіни кордону, ми телепортуємося до випадкової точки вздовж кордону, а вже потім продовжуємо рухатися. У разі додавання стін він працює майже в два рази швидше, тому що телепортація уздовж стіни кордону дозволяє швидше отримувати доступ до далеких частин лабіринту.

У курсовому проєкті було вирішено використовувати алгоритм Recursive backtracker, так як він вважається швидким для невеликих двомірних лабіринтів з ортогональною теселяцією.

1.2 Огляд існуючих програмних засобів, що вирішують аналогічні завдання

1.2.1 Mazes & More: класична мобільна головоломка з використанням лабіринтів.

Основними особливостями даної гри можна виділити: цікавий ігровий процес, досить важкі завдання, та дуже цікаві механіки. Та можна також виділити мінус для аналізу — це, то що усі лабіринти в цій грі були збудовані розробниками власноруч.

Інші особливості гри Mazes & More:

- 1) легке управління персонажем, яке виконується за допомогою стрілочок біля гравця, а не за допомогою маркерів або акселерометра.
- 2) усі лабіринти зроблені вручну, без рандомізованих рівнів.
- 3) 6 категорій рівнів: «Класика», «Вороги», «Крижаний підлогу», «Темрява», «Пастки» та «Проба часу». Різні біоми та механіки рівнів не дають нудьгувати гравцям.
- 4) головоломки варіюються від простих до складних лабіринтів
- 5) мінімальна та ретро 2D графіка.

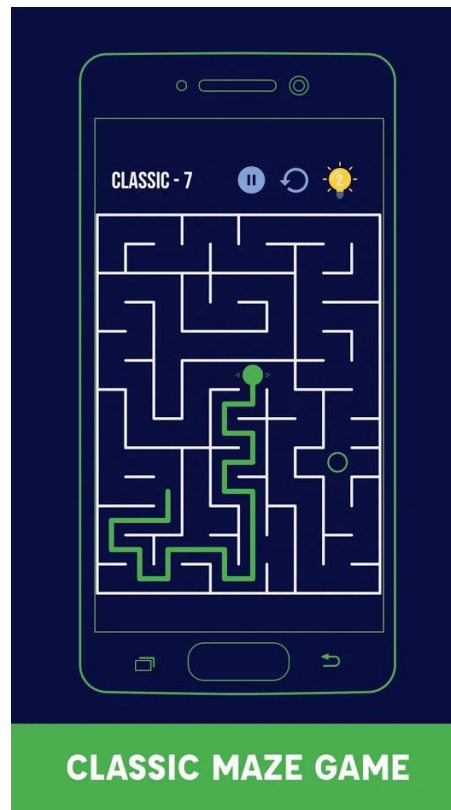


Рисунок 1.7 – Приклад ігрового процесу гри Mazes & More

1.2.2 Flow Free: мобільна гра-головоломка з механікою, яка схожа на проходження лабіринту.

Основну ідею яку транслює дана гра — це схожість механікою на гру-лабіринт, але вам потрібно скоріш його саморуч побудувати.

У грі присутні різні режими з таймером або без. Також у грі є режим мультиплеєра та кооперативний, які допоможуть з'ясувати хто найкмітливіший з ваших друзів.

Ключові особливості даного проекту:

- 1) більше 2500 головоломок;
- 2) режими вільної гри і гонки з часом;
- 3) чітка і яскрава графіка;
- 4) приємні звукові ефекти.

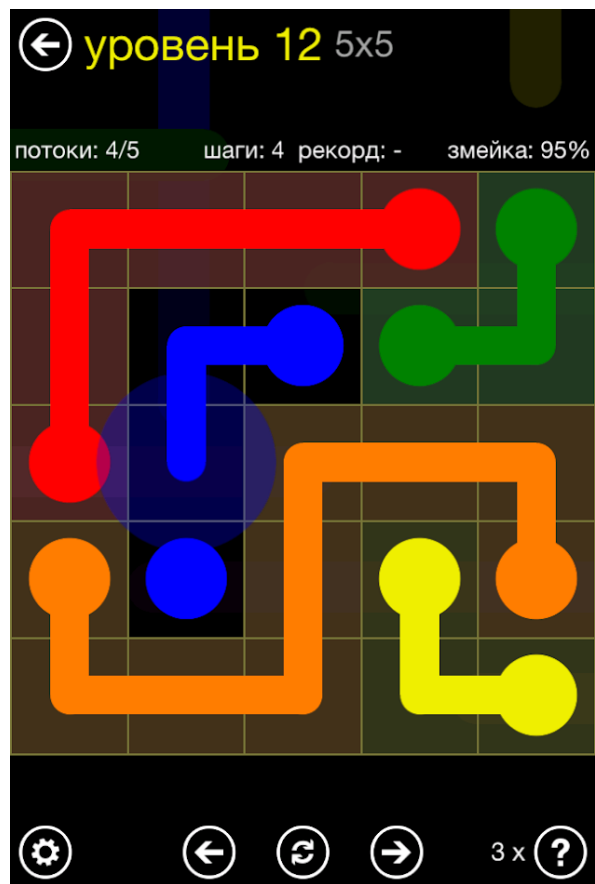


Рисунок 1.8 – Приклад ігрового процесу гри Flow Free

1.3 Постановка завдання до роботи

Створити ігровий продукт, основною задачею якого повинна бути генерація рівнів-лабіринтів.

У грі повинні бути присутні такі елементи:

- 1) головне меню;
- 2) меню кінця гри;
- 3) процедурна генерація лабіринту;
- 4) підказка гравцю;
- 5) мінімальний ігровий процес.

2 АНАЛІЗ ПРОГРАМНИХ ЗАСОБІВ

2.1 Огляд особливостей рушія гри.

Unity — це більше, ніж рушій, це середовище для розробки комп'ютерних ігор, в якій об'єднані різні програмні засоби, що використовуються при створенні ПЗ — текстовий редактор, компілятор, відладчик і так далі. При цьому, завдяки зручності використання, Unity робить створення ігор максимально простим і комфортним, а мультиплатформеність рушія дозволяє розробникам охопити якомога більшу кількість ігрових платформ і операційних систем.

Чому Unity вдала платформа для розробки ігор? В першу чергу, як ми вже згадували, рушій Unity3D дає можливість розробляти гри, не вимагаючи для цього якихось особливих знань. Тут використовується компонентно-орієнтований підхід, в рамках якого розробник створює об'єкти (наприклад, головного героя) і до них додає різні компоненти (наприклад, візуальне відображення персонажа і способи управління ним). Завдяки зручному Drag & Drop інтерфейсу і функціональним графічному редактору рушій дозволяє малювати карти і розставляти об'єкти в реальному часі і відразу ж тестувати результат.

Друга перевага рушія — наявність величезної бібліотеки Ассет і плагінів, за допомогою яких можна значно прискорити процес розробки гри. Їх можна імпортувати й експортувати, додавати в гру цілі заготовки: рівні, ворогів, патерни поведінки AI і так далі. Ніякої метушні з програмуванням. Багато Ассет надаються безкоштовно, інші пропонуються за невелику суму, і при бажанні можна створювати власний контент, публікувати його в Unity Asset Store і отримувати від цього прибуток.

Третя сильна сторона Unity 3D — підтримка величезної кількості платформ, технологій, API. Створені на рушії ігри можна легко перенести між ОС Windows, Linux, OS X, Android, iOS, на консолі сімейств PlayStation, Xbox, Nintendo, на VR- і AR-пристрої. Unity підтримує DirectX і OpenGL, працює з усіма сучасними ефектами рендерингу, включаючи новітню технологію трасування променів в реальному часі.

Фізика твердих тіл, ragdoll і тканин, система Level of Detail, колізії між об'єктами, складні анімації — все це можна реалізувати силами рушія. Стереотипна думка про те, що рушій придатний тільки для невеликих інді-ігор і нездатний видавати красиву картинку, давно вже не актуально: досить подивитися технодемо ADAM, The Blacksmith і Book of the Dead від творців середовища Unity, щоб переконатися в її видатних здібностях.

Нарешті, Unity доступний безкоштовно, що відкриває перед незалежними розробниками двері в ігрову індустрію. Звичайно, існують обмеження: безкоштовна версія рушія демонструє лого Unity перед запуском гри, а проект, створений з її допомогою, не повинен приносити розробнику більше \$100 тисяч в рік. Втім, тарифи на підписку НЕ спустошать гаманці навіть починаючої команди: Про-версія коштує \$125 на місяць, що не так вже й багато в порівнянні з іншими рушіями, причому базова версія містить рівно той самий функціонал, що і професійна.

Які недоліки приховує Unity? При всіх своїх перевагах, рушій має і свої недоліки. Так, якщо команда захоче розробити що-небудь складніше простого клікера або платформера, то їй доведеться шукати хорошого програміста на C#, який напише скрипти і компоненти, запровадить їх у гру і змусить працювати.

З цього випливає інша проблема движка Unity — повільність. Створення масштабних, складних сцен з великою кількістю компонентів може негативно вплинути на продуктивність гри, в результаті чого розробникам доведеться витратити додатковий час і ресурси на оптимізацію, а, можливо, і видалення деяких елементів з проекту.

Крім того, додатки, створені на Unity, досить «важкі»: навіть найпростіша піксельна гра може займати кілька сотень мегабайт на ПК. Так, для жорсткого диска комп'ютерів це невеликий обсяг, але, якщо проект розробляється і для мобільних платформ, слід задуматися про оптимізацію його розміру.

2.2 Топ-5 кращих ігор створених на Unity:

1. Rust — популярний симулятор виживання з крафтингом, будівництвом, прокачуванням, битвами та іншими атрибутами жанру.



Рисунок 2.1 – Приклад ігрового процесу гри Rust.

2. Pillars of Eternity — ізометрична партійна CRPG «старої школи» від студії Obsidian, присвячена пригодам героїв в фентезійному світі, поглиненому надприродною напастю.



Рисунок 2.2 – Приклад ігрового процесу гри Pillars of Eternity .

3. Ori and the Blind Forest — казково красивий і зубодробильний складний платформер, в якому гравцеві належить врятувати чарівний ліс від жорстокого лиходія.

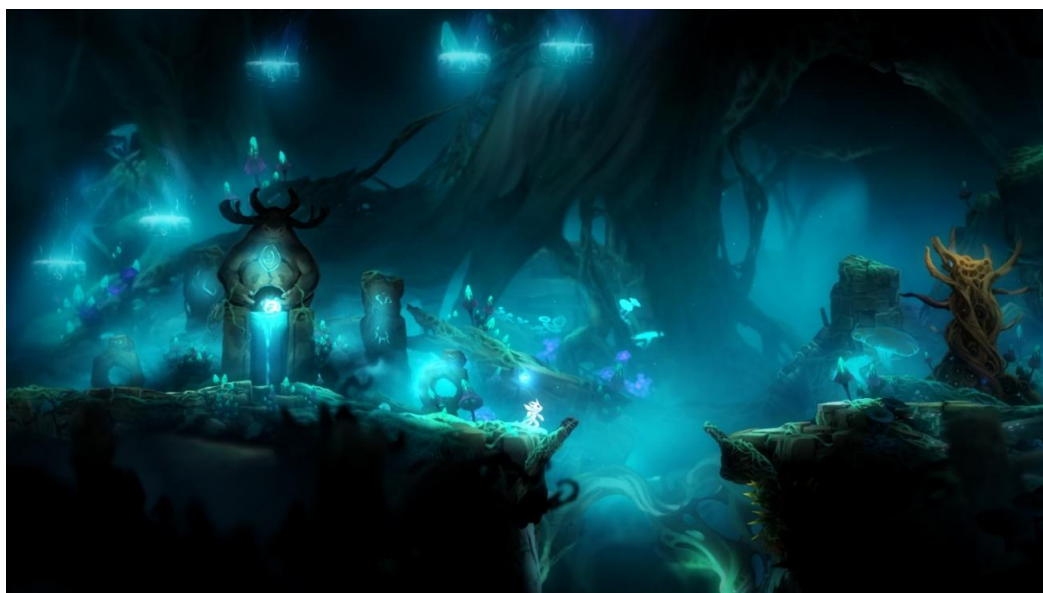


Рисунок 2.3 – Приклад ігрового процесу гри Ori and the Blind Forest .

4. 7 Days to Die — симулятор виживання в сетінгу зомбі-апокаліпсису з процедурною генерацією світу, руйнуванням локацій і величезними натовпами ходячих мерців.



Рисунок 2.4 – Приклад ігрового процесу гри Ori and the 7 Days to Die .

5. Endless Legend — фентезійна 4X-стратегія, яка пропонує гравцеві завоювати далеку планету, використовуючи всі можливі способи - від дипломатії до військової могутності.

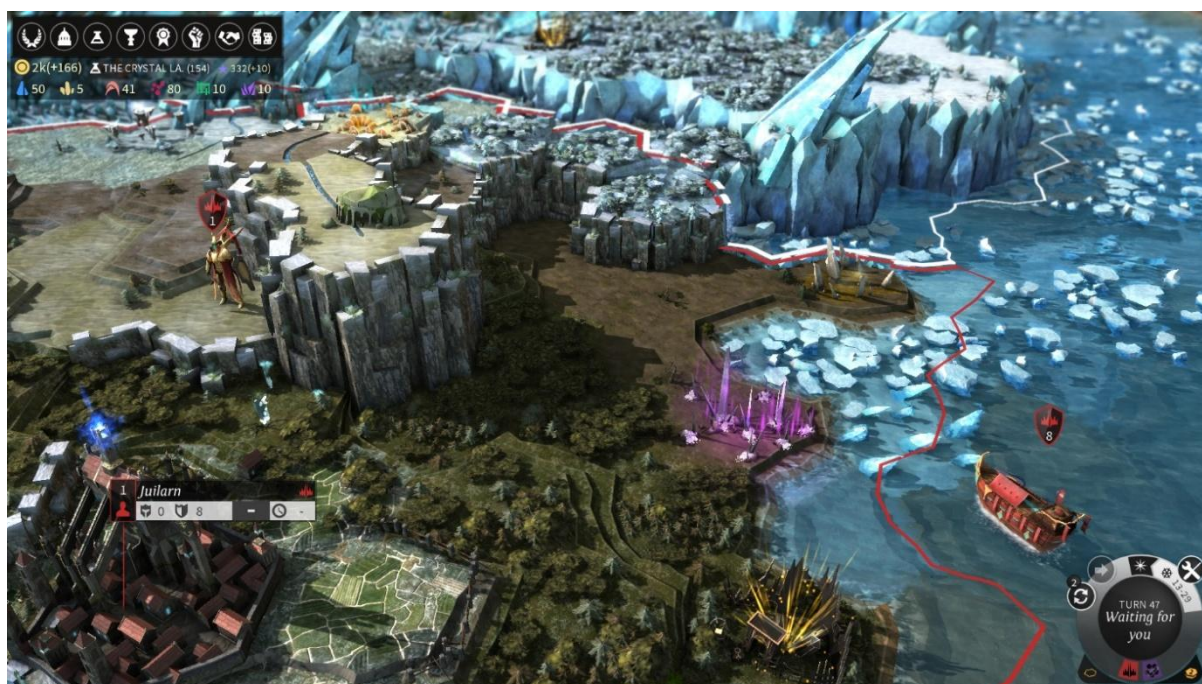


Рисунок 2.5– Приклад ігрового процесу гри Endless Legend .

2.2 Огляд особливостей мови програмування

Мова C#, розроблена компанією Майкрософт, одна з найпопулярніших сучасних мов програмування. Вона затребувана на ринку розробки в різних країнах, C# застосовують при роботі з програмами для ПК, створення складних веб-сервісів або мобільних додатків. З'явилася як мова для власних потреб платформи Microsoft.NET, поступово ця мова стала дуже популярною. А тому було вирішено зробити невеликий огляд для тих, хто вибирає, які інструменти розробки варто освоїти найближчим часом.

Отже, розробка мови почалася в 1998 році, а перша версія побачила світ в 2001. Групою розробників керував відомий в професійних колах фахівець Андерс Хейлсберг. Нові версії C# виходять порівняно часто, а поточні доопрацювання, виправлення багів і розширення бібліотек ведеться практично на постійній основі.

В результаті мова вийшла вкрай гнучкою, потужною та універсальною. На ній пишуть практично все, від невеликих веб-додатків до потужних програмних систем, які об'єднують в собі веб-структури, додатки для десктопів і мобільних пристроїв. Все це стало можливим завдяки зручному Сі-подібного синтаксису, суворому структуруванню, величезній кількості фреймворків і бібліотек (їх число досягає декількох сотень).

Довгий час платформа .NET поставлялася з закритим ядром, що створювало певні труднощі в розробці і знижувало популярність C# в професійному середовищі. Але в листопаді 2014 Майкрософт радикально змінила підхід і стала видавати безкоштовні ліцензії для Visual Studio вже з відкритим вихідним кодом для всіх наборів інструментів.

Чим цікавий C#? C# — дійсно цікавий інструмент, гідний уваги. Він впевнено займає високі позиції в рейтингах затребуваних мов програмування на ринку праці. Тому має сенс вивчити його можливості докладніше і зрозуміти, для чого і де варто застосовувати C#.

Компанія Microsoft приділяє значну увагу підтримці мови розробки, а тому регулярно з'являються оновлення та доповнення, виправляються виявлені баги в компіляторі, розширюються бібліотеки. Розробники зацікавлені в популяризації інструменту і докладають до цього масу зусиль.

Розробники надають докладну і розгорнуту документацію на своїх офіційних ресурсах. Крім того, відповіді практично на будь-які питання, пов'язані з роботою в C#, можна знайти в мережі. Популярність мови привела до появи безлічі професійних

співтовариств, присвячених C#. Існує безліч підручників, курсів для новачків і мідл розробників, відео добірок та інших навчальних матеріалів.

Інструментарій C# дозволяє вирішувати широке коло завдань, мова дійсно дуже потужна та універсальна. На ній розробляють:

- 1) додатки для WEB;
- 2) різні ігрові програми.
- 3) додатки платформ Андроїд або iOS;
- 4) програми для Windows.

Перелік можливостей розробки практично не має обмежень завдяки найширшому набору інструментів і засобів. Звичайно, все це можна реалізувати за допомогою інших мов, але деякі з них вузькоспеціалізовані, в інших доведеться використовувати додаткові інструменти сторонніх розробників. У C# рішення широкого кола завдань можливе швидше, простіше і з меншими витратами часу і ресурсів.

Інструмент «Прибирання сміття» дозволяє в автоматичному режимі очистити пам'ять від об'єктів, які не використовуються, або знищених додатків.

«Обробка винятків». За допомогою цього інструменту можна легко виявляти і обробляти помилки в коді. Спосіб є структурованим з широким набором функцій. При цьому важливо не зловживати можливостями роботи з винятками, так як при неправильному використанні з'являється ризик появи «багів».

Єдина система типів. У мові прийнята загальна система роботи з типами, починаючи від примітивів і закінчуючи складними, в тому числі, призначеними для користувача наборами. Застосовується єдиний набір операцій для обробки і зберігання значень типізації. Також можна використовувати посилальні типи користувача, що дозволяє динамічно виділити пам'ять під об'єкт або зберігати спрощену структуру в мережі.

Мова програмування забороняє звернення до змінних, що не були ініційовані, що виключає можливість виконання безконтрольного приведення типів або виходу за межі певного масиву даних.

Управління версіями. Дуже цікава особливість мови програмування. Суть в тому, що багато мов не приділяють належної уваги цьому питанню, і програми нерідко перестають коректно працювати при переході на нову версію продукту. У C# це було виправлено.

Для використання цієї функції на C# необхідно встановити і налаштувати платформу .NET Framework. Вона поставляється повністю безкоштовно, застосовується вкрай широко, а тому проблем з користувацькими пристроями зазвичай не виникає. Платформа вбудована в інсталяційний пакет Windows, при необхідності її також можна скачати і «поставити» окремо. Існують версії для Лінукс і MAC.

В рамках платформи до обробки виконуваного коду підключають середу CLR — єдиний об'єднаний набір бібліотек і класів, який був розроблений Майкрософт і є реалізацією світового стандарту Common Language Infrastructure (CLI).

Після роботи компілятора текст програми перекладається в проміжний мову IL, яку «розуміє» CLI. Працює це так: IL і всі необхідні ресурси, включаючи рядки і малюнки формату BMP, зберігаються на жорсткий диск у вигляді виконуваного файлу dll або exe. З таких файлів з проміжним кодом формується збірка додатка, яка включає в себе опис з повною інформацією про всі важливі параметри роботи.

Безпосередньо при виконанні програми CLR звертається до збірки і виробляє дії в залежності від отриманих відомостей. Якщо код написаний правильно і проходить перевірку безпеки системи, проводиться компіляція з IL в інструкції в машинні команди. Серед CLR попутно виконує ще багато побічних функцій:

- 1) видалення «програмного» сміття;
- 2) робота з винятками;
- 3) розподіл ресурсів;
- 4) контроль типізації;
- 5) управління версіями.
- 6) типізація.
- 7) управління версіями.

В результаті код C# вважається керованим, тобто він компілюється в двійковий вид на призначеному для користувача пристрої з урахуванням особливостей встановленої системи.

2.3 Огляд можливостей бібліотеки класів .NET

Реалізації .NET містять класи, інтерфейси, делегати та типи значень, які полегшують і оптимізують процес розробки, а також забезпечують доступ до функцій системи. Для спрощення взаємодії між мовами більшість типів платформи .NET є CLS-сумісними, і тому їх можна використовувати в будь-якій мові програмування, компілятор якого відповідає специфікації CLS.

Типи .NET являють собою основу для створення елементів управління, компонентів і додатків .NET. Реалізації .NET містять типи, призначені для виконання таких завдань:

- 1) уявлення базових типів даних і винятків;
- 2) інкапсуляція структур даних;
- 3) операції введення-виведення;
- 4) доступ до даних про завантажені типи;
- 5) виклик перевірок безпеки .NET Framework;
- 6) доступ до даних, надання графічного інтерфейсу користувача на стороні клієнта і керованого сервером графічного призначеного для користувача інтерфейсу на стороні клієнта.

.NET пропонує розширений набір інтерфейсів, а також абстрактних і конкретних (не абстрактних) класів. Можна використовувати існуючі конкретні класи, крім того, у багатьох випадках на їх основі можна створювати власні похідні класи. Щоб скористатися наявними можливостями інтерфейсу, можна або створити клас, який реалізує інтерфейс, або створити похідний клас на основі одного з класів .NET, що реалізує інтерфейс.

2.4 Огляд шаблонних класів бібліотеки класів .NET, що використовується в роботі

При розробці гри використовувалися такі бібліотеки класів .NET, як: System.Collections.Generic та System.

З бібліотеки System використовувалась TimeSpan структура. Об'єкт TimeSpan представляє інтервал часу (тривалість часу або час, що минув), вимірюваний як позитивне або негативне число днів, годин, хвилин, секунд і часток секунди. Структуру TimeSpan можна також використовувати для представлення часу доби, але тільки в тому випадку, якщо час не пов'язаний з певною датою. В іншому випадку замість неї слід використовувати структуру DateTime або DateTimeOffset.

Приклад використання можна побачити у скрипті Timer.cs який розроблений у якості таймера для створеної гри:

```
using System;
using UnityEngine;
```

```

using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class Timer : MonoBehaviour
{
    private TimeSpan timer = new TimeSpan();
    public Text seconds;

    void Update()
    {
        timer = timer.Add(TimeSpan.FromSeconds(Time.deltaTime));
        seconds.text = timer.Seconds.ToString("00");
        if (seconds.text == "60")
        {
            SceneManager.LoadScene("End");
        }
    }
}

```

System.Collections.Generic ми використовували для використання у нашому коді таких структур даних як List, Stack та інші.

У C# колекція представляє собою сукупність об'єктів. У середовищі .NET Framework є чимало інтерфейсів і класів, в яких визначаються і реалізуються різні типи колекцій. Колекції спрощують вирішення багатьох завдань програмування завдяки тому, що пропонують готові рішення для створення цілого ряду типових, але часом трудомістких для розробки структур даних. Наприклад, в середу .NET Framework вбудовані колекції, призначені для підтримки динамічних масивів, зв'язкових списків, стеків, черг і хеш-таблиць. Колекції є сучасним технологічним засобом, що заслуговує пильної уваги всіх, хто програмує на C#.

Спочатку існували тільки класи неузагальнених колекцій. Але з впровадженням узагальнень у версії C# 2.0 середу .NET Framework було доповнено багатьма новими узагальненими класами та інтерфейсами. Завдяки введенню узагальнених колекцій загальна кількість класів і інтерфейсів подвоїлася. Разом з бібліотекою розпаралелювання завдань (TPL) у версії 4.0 середовища .NET Framework з'явився ряд

нових класів колекцій, призначених для застосування в тих випадках, коли доступ до колекції здійснюється з кількох потоків. Незавжди здогадатися, що прикладний інтерфейс Collections API становить значну частину середовища .NET Framework.

Короткий огляд колекцій. Головна перевага колекцій полягає в тому, що вони стандартизують обробку груп об'єктів в програмі. Всі колекції розроблені на основі набору чітко визначених інтерфейсів. Деякі вбудовані реалізації таких інтерфейсів, в тому числі ArrayList, Hashtable, Stack і Queue, можуть застосовуватися в початковому вигляді і без будь-яких змін. Є також можливість реалізувати власну колекцію, хоча потреба в цьому виникає вкрай рідко.

У середовищі .NET Framework підтримуються п'ять типів колекцій: неузагальнені, спеціальні, з поразрядною організацією, узагальнені та паралельні.

Неузагальнені колекції. Реалізують ряд основних структур даних, включаючи динамічний масив, стек, чергу, а також словники, в яких можна зберігати пари "ключ-значення". Відносно неузагальнених колекцій важливо мати на увазі наступне: вони оперують даними типу object. Таким чином, неузагальнених колекції можуть служити для зберігання даних будь-якого типу, причому в одній колекції допускається наявність різнотипних даних. Очевидно, що такі колекції не типізовані, оскільки в них зберігаються посилання на дані типу object. Класи і інтерфейси неузагальнених колекцій знаходяться в просторі імен System.Collections.

Спеціальні колекції. Оперують даними конкретного типу або ж роблять це якимось особливим чином. Наприклад, є спеціальні колекції для символьних рядків, а також спеціальні колекції, в яких використовується односпрямований список. Спеціальні колекції оголошуються в просторі імен System.Collections.Specialized.

Порозрядна колекція. У прикладному інтерфейсі Collections API визначена одна колекція з поразрядною організацією — це BitArray. Колекція типу BitArray підтримує порозрядні операції, тобто операції над окремими двійковими розрядами, наприклад І, АБО, що виключає АБО, а отже, вона істотно відрізняється своїми можливостями від інших типів колекцій. Колекція типу BitArray оголошується в просторі імен System.Collections.

Узагальнені колекції. Забезпечують узагальнену реалізацію декількох стандартних структур даних, включаючи зв'язкові списки, стеки, черги і словники. Такі колекції є типізованими в силу їх узагальненого характеру. Це означає, що в узагальненій колекції можуть зберігатися тільки такі елементи даних, які сумісні за

типом з даною колекцією. Завдяки цьому виключається випадкова розбіжність типів. Узагальнені колекції оголошуються в просторі імен System.Collections.Generic.

Паралельні колекції. Це узагальнені колекції, певні в просторі імен System.Collections.Concurrent.

У просторі імен System.Collections.ObjectModel знаходиться також ряд класів, що підтримують створення користувачами власних узагальнених колекцій.

Основоположним для всіх колекцій є поняття перераховувача, який підтримується в не узагальнених інтерфейсах IEnumerator і IEnumerable, а також в узагальнених інтерфейсах IEnumerator <T> і IEnumerable <T>. Нумератор забезпечує стандартний спосіб почергового доступу до елементів колекції. Отже, він перераховує вміст колекції. У кожній колекції повинна бути реалізована узагальнена або не узагальнена форма інтерфейсу IEnumerable, тому елементи будь-якого класу колекції повинні бути доступні за допомогою методів, визначених в інтерфейсі IEnumerator або IEnumerator <T>. Це означає, що, внісши мінімальні зміни в код циклічного звернення до колекції одного типу, його можна використовувати для аналогічного звернення до колекції іншого типу. Цікаво, що для почергового звернення до вмісту колекції в циклі foreach використовується нумератор.

Приклад використання бібліотеки System.Collections.Generic:

```
private void WallsBackTracker(MazeGeneratorCell[,] maze)
{
    MazeGeneratorCell current = maze[0, 0];

    current.isVisited = true;
    current.DistanceFromStart = 0;

    Stack<MazeGeneratorCell> stack = new
Stack<MazeGeneratorCell>();
    do
    {
        List<MazeGeneratorCell> unvisitedCells = new
List<MazeGeneratorCell>();

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

```

        //Check neighboring cells
        if (x > 0 && !maze[x - 1, y].isVisited)
unvisitedCells.Add(maze[x - 1, y]);
        if (y > 0 && !maze[x, y - 1].isVisited)
unvisitedCells.Add(maze[x, y - 1]);
        if (x < Width - 2 && !maze[x + 1, y].isVisited)
unvisitedCells.Add(maze[x + 1, y]);
        if (y < Height - 2 && !maze[x, y + 1].isVisited)
unvisitedCells.Add(maze[x, y + 1]);

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

            choosen.isVisited = true;
            stack.Push(choosen);
            choosen.DistanceFromStart = current.DistanceFromStart +
1;

            current = choosen;
        }
        else
            current = stack.Pop();

    } while (stack.Count > 0);
}

```

У цьому прикладі було використано Stack для запису клітинок майбутнього лабіринту.

Для рендеру підказки було використано метод DrawPath() у якому ми використовували List, а потім додавали в нього елементи за допомогою методу Add().

Нижче можна побачити приклад використання:

```

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

```

```

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());
    }

```

2.5 Огляд можливостей Scripting API у Unity Engine.

Рушій гри Unity має власний Scripting API у якому описано безліч класів та методів для створення ігор на будь-який смак та жанр. У цьому розділі ми розглянемо їх особливості та приклади використання.

MonoBehaviour — базовий клас, з якого походить кожен сценарій Unity. Він має безліч методів серед них ми розглянемо такі як Start() та Update().

Start() викликається на кадрі, коли сценарій увімкнено безпосередньо перед тим, як будь-який із методів оновлення буде викликаний вперше.

Приклад використання методу Start() у класі HintRenderer даного курсового проекту:

```

private LineRenderer componentLineRenderer;
private void Start()
{
    componentLineRenderer = GetComponent<LineRenderer>();
}

```

У цьому випадку ми декларуємо елемент LineRenderer зі змінною componentLineRenderer, а потім присвоюємо до неї сам компонент.

Update() визивається кожним кадром, якщо MonoBehaviour увімкнено. Update() — це функція яка найчастіше використовується для реалізації будь-якого ігрового сценарію. Не кожен сценарій клас MonoBehaviour потребує оновлення.

Приклад використання методу Update() у класі HintRenderer даного курсового проекту:

```

void Update()
{
    if (Input.GetKey(KeyCode.R))
        componentLineRenderer.sortingOrder = 1;
    if (Input.GetKey(KeyCode.F))
        componentLineRenderer.sortingOrder = -1;
}

```


В даному випадку виконується перевірка натискання клавіш на клавіатурі кожен фрейм, якщо умова збігається — ми виконуємо нашу дію. Метод Update() дуже зручно використовувати для створення кастомних обробників подій.

Також у класі PlayerMovement використовувався стандартний метод OnTriggerEnter2D. Даний метод використовується як обробник події, якщо наш елемент проходить в зону триггера, то виконується якась дія.

Нижче наведено приклад використання у PlayerMovement.cs:

```
void OnTriggerEnter2D(Collider2D col)
{
    if (col.gameObject.name == "finish")
    {
        SceneManager.LoadScene("End");
    }
}
```

У цьому прикладі йде перевірка на те, що якщо наш гравець заходить у триггер елемента «finish», то виконується умова, а саме завершення гри.

2.6 Висновки з розділу

У даному розділі було описано особливості мови програмування C# та рушія гри Unity. Окрім цього у ньому описані класи бібліотеки .NET, що були використані у ході розробки гри та основні можливості Scripting API у Unity Engine.

3 ОСНОВНІ РІШЕННЯ З РЕАЛІЗАЦІЇ КОМПОНЕНТІВ СИСТЕМИ

3.1 Основні рішення щодо уявлення даних системи

При розробці гри було розроблено такі класи та описані їх методи:

PlayerMovement, MenuControl, CameraController, Cell, HintRenderer, Maze, MazeGeneratorCell, MazeGenerator, MazeGeneratorCell, MazeSpawner, Timer.

Клас PlayerMovement містить у собі загальну інформацію про пересування нашого гравця.

Таблиця 3.1 – Дані та методи класу PlayerMovement

Поля та методи класу	Опис
1	2
public:	
float Speed = 2f;	Швидкість переміщення гравця.
private:	
Rigidbody2D rb;	Змінна для компоненту Rigidbody2D для нашого гравця.
void Start()	Стандартний метод рушія. У ньому ми присвоюємо змінній rb компонент Rigidbody2D за допомогою метода GetComponent<Rigidbody2D>()
void Update()	Стандартний метод рушія. У ньому ми викликаємо метод HandleMovement();
void HandleMovement()	Перевірка заданих клавіш клавіатури, якщо виконується умова - гравець рухається.
void OnTriggerEnter2D(Collider2D col)	Йде перевірка на те, що якщо наш гравець заходить у триггер елементу "finish" - то виконується умова, а саме завершення гри.

Клас MenuControl містить методи для переходу між сценами гри.

Таблиця 3.2 – Дані та методи класу MenuControl

Поля та методи класу	Опис
1	2
public:	
public void PlayPressed()	У цьому методі йде перехід до основної сцени гри “Game” за допомогою: SceneManager.LoadScene("Game");
public void ExitPressed()	Вихід з гри.
public void MenuPressed()	Перехід до головного меню.

Клас CameraController описує переміщення камери та контролює прив'язку камери до координат гравця.

Таблиця 3.3 – Дані та методи класу CameraController

Поля та методи класу	Опис
1	2
public:	
Vector3 offset	Офсет камери відносно гравця.
Transform player	Доступ до компоненту гравця та його пересуванню..
float smooth = 5.0f;	Змінна, за допомогою якої хід нашої камери буде більш плавним.

Продовження таблиці 3.3

private:	
void Update()	Стандартний метод рушія. У ньому ми transform.position нашої камери відносно гравця.

Клас Cell описує префаб-стіни для подальшого контролю над ними.

Таблиця 3.4 – Дані та методи класу Cell

Поля та методи класу	Опис
1	2
public:	
GameObject WallLeft;	Доступ до лівої стіни префабу Cell.
GameObject WallBottom;	Доступ до правої стіни префабу Cell.

Клас Maze містить двомірний масив з нашим лабіринтом та його фінішну точку.

Таблиця 3.5 – Дані та методи класу Maze

Поля та методи класу	Опис
1	2
public:	
MazeGeneratorCell[,] cells;	Двомірний масив з точками лабіринта.
Vector2Int finishPos;	Фінішна точка лабіринту.

Клас `MazeGeneratorCell` зберігає дані для класу `MazeGenerator`. Такі як координати, дані про присутність, дистанцію від старту.

Таблиця 3.6 – Дані та методи класу `Maze`

Поля та методи класу	Опис
1	2
public:	
<code>int X;</code>	Координата X для нашої клітинки
<code>int Y</code>	Координата Y для нашої клітинки
<code>bool WallLeft = true;</code>	Булева змінна яка свідчить о присутності лівої стіни.
<code>bool WallBottom = true;</code>	Булева змінна яка свідчить о присутності правої стіни.
<code>int DistanceFromStart;</code>	Змінна яка свідчить відстань від старту.
<code>bool isVisited = false;</code>	Булева змінна яка свідчить о відвідуванні нашої клітинки алгоритмом.

Клас `MazeGenerator` описує генерацію нашого лабіринту за допомогою алгоритма, видаляє зламані стіни, ставить вихід з лабіринту.

Таблиця 3.7 – Дані та методи класу `MazeGenerator`

Поля та методи класу	Опис
1	2
public:	
<code>int Width = 23;</code>	Змінна яка контролює ширину нашого лабіринту
<code>int Height = 15;</code>	Змінна яка контролює довжину нашого лабіринту
<code>GameObject FinishFlag</code>	Змінна об'єкту флагоу фінішу.

Продовження таблиці 3.7

Maze GenerateMaze()	Метод який генерує остаточну версію лабіринту
void RemoveBrokenWall(MazeGeneratorCell[,] maze)	При створенні лабіринту з боків створюються зламані стіни, даний метод приймає в себе масив лабіринту та видаляє зламані стіни.
private:	
void WallsBackTracker(MazeGeneratorCell[,] maze)	Даний метод є реалізацією алгоритма Recursive backtracker. Який створює стек з нашими клітинками, далі у циклі перевіряє усі сусідні клітинки та видаляє зайві стіни для створення проходу.
Vector2Int PlaceMazeExit(MazeGeneratorCell[,] maze)	Цей метод приймає в якості аргументу масив клітинок лабіринту. Визначає найвіддаленішу точку від входу та генерує вихід через видалення стіни та розміщення на цьому місці флагу фініша.
void RemoveWall(MazeGeneratorCell a, MazeGeneratorCell b)	Метод який приймає в себе 2 стіни, та вирішує яку стіну видалити.

Клас MazeSpawner зав'язан на пустий GameObject на сцені та виконує генерацію лабіринту та підказки з класу HintRenderer.

Таблиця 3.8 – Дані та методи класу MazeSpawner

Поля та методи класу	Опис
1	2
public:	
GameObject CellPrefab	Змінна префабу клітинки
HintRenderer HintRenderer;	Доступ до класу HintRenderer
Maze maze	Доступ до класу Maze та лабіринту

Продовження таблиці 3.8

Vector3 CellSize = new Vector3(1, 1, 0);	Розмір клітини. Був зроблений для подальшого масштабування процесу розробки.
private:	
void Start()	Стандартний метод рушія. Викликає метод генератора, а далі через 2 цикли генерує лабіринт та робить активними визначені стіни.
void Update()	Стандартний метод рушія. Викликає метод генерації підказки з класу HintRenderer.

Клас HintRenderer підраховує шлях підказки починаючи з кінця та до початку гри.

Таблиця 3.9 – Дані та методи класу HintRenderer .

Поля та методи класу	Опис
1	2
public:	
MazeSpawner MazeSpawner;	Доступ до класу MazeSpawner. Потрібен для отримання лабіринту та розміру клітинок.
HintRenderer HintRenderer;	Доступ до класу HintRenderer
void DrawPath()	Метод який виконує генерацію шляху-підказки для гравця. У ньому отримується фінішна позиція, та List з позиціями шляху, а далі у циклі генерує його для подальшого виклику у класі MazeSpawner.

Продовження таблиці 3.9

private:	
LineRenderer componentLineRenderer	Компонент LineRenderer за допомогою якого буде рендеритися шлях-підказка.
void Update()	Стандартний метод рушія. Містить у собі перевірки натискання кнопок клавіатури і якщо умова збігається — переводить наш шлях на передній план.
void Start()	Стандартний метод рушія. Отримуємо у змінну componentLineRenderer сам компонент.

Клас Timer виконує функцію таймера для гри.

Таблиця 3.10 – Дані та методи класу Timer.

Поля та методи класу	Опис
1	2
public:	
Text seconds;	Змінна для доступу елементу UI Text.
private:	
TimeSpan timer = new TimeSpan();	Декларація таймеру.
void Update()	Стандартний метод рушія. У тілі метода додається новий часовий проміжок для таймера. Отримуємо контент елементу Text. І робимо перевірку, на те що, якщо у змінній seconds значення дорівнює 60 - переводимо до сцени кінця гри.

3.2 Основні розроблені алгоритми

Під час розробки було реалізовано наступні алгоритми:

- 1) Контроль стану сцен.
- 2) Контроль камери та створення трекінг камери для гравця.
- 3) Переміщення персонажу.
- 4) Обробка триггерів.
- 5) Алгоритм таймеру та приєднання його до GUI.
- 6) Генерація сітки для подальшої обробки алгоритмом.
- 7) Алгоритм видалення стінок для генерації лабіринту
- 8) Видалення зламаних стінок.
- 9) Розміщення виходу.
- 10) Поміщення лабіринту на сцену.
- 11) Створення шляху-підказки та його рендер.

Контроль стану сцен виконується за допомогою скрипта *MenuControl.cs*.

Даний скрипт ми додаємо до об'єкта UI на сцені. Далі у меню інспектора кнопок на сценах End та Menu ми вибираємо функцію, яка буде працювати по кліку для даної кнопки. У самому скрипті ми виконуємо переходи в методах за допомогою метода *SceneManager.LoadScene()* або *Application.Quit()*; як у методі з виходом. У сцені End виконується все так само. Нижче показана реалізація цього алгоритму.

```
public class MenuControl : MonoBehaviour
{
    public void PlayPressed()
    {
        SceneManager.LoadScene("Game");
    }

    public void ExitPressed()
    {
        Debug.Log("EXIT PRESSED");
        Application.Quit();
    }

    public void MenuPressed()
    {
        SceneManager.LoadScene("Menu");
    }
}
```

```

    }
}

```

Алгоритм контролю камери виконується за допомогою скрипта *CameraController.cs*. Даний скрипт ми додали як новий компонент для елемента *Main Camera* та вказали дані для публічних змінних. У змінну *Player* ми додали елемент нашого гравця для того, щоб камера слідувала за ним. Також ми вказали офсет та плавність ходу камери. Офсет визначає на скільки камера буде віддален від гравця.

У самому скрипті в методі *Update()* ми прив'язуємо координати камери до координат гравця за допомогою *Vector3.Lerp()* яка виконує лінійну інтерполяцію між двома точками. Реалізація даного алгоритму:

```

public class CameraController : MonoBehaviour
{
    public Vector3 offset;
    public Transform player;
    public float smooth = 5.0f;
    //Create smooth tracking camera
    void Update()
    {
        transform.position = Vector3.Lerp(transform.position,
player.position + offset, Time.deltaTime * smooth);
    }
}

```

Переміщення персонажа описано в скрипті *PlayerMovement.cs*. Для початку визначаємо швидкість з якою буде рухатися персонаж та отримуємо посилання на його компонент *Rigidbody2D*. Далі присвоюємо змінній *rb* цей компонент у методі *Start()*.

Потім декларується метод *HandleMovement()* у якому описано переміщення персонажа. Для початку ми присвоюємо *rb.velocity = Vector2.zero* для того, щоб наш персонаж не отримував занадто сильного прискорення. Потім виконуємо перевірку, якщо ми натискаємо на клавіатурі зазначені кнопки (W, A, S, D), то гравець змінює свої координати враховуючи швидкість, яка задана раніше. І потім ми виконуємо цей метод пофреймово в методі *Update()*. Реалізація алгоритму зазначена нижче:

```

public float Speed = 2f;
private Rigidbody2D rb;

```

```

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

void Update()
{
    HandleMovement();
}

void HandleMovement()
{
    rb.velocity = Vector2.zero;
    if (Input.GetKey(KeyCode.A)) { rb.velocity += Vector2.left *
Speed; }
    if (Input.GetKey(KeyCode.D)) rb.velocity += Vector2.right *
Speed;
    if (Input.GetKey(KeyCode.W)) rb.velocity += Vector2.up * Speed;
    if (Input.GetKey(KeyCode.S)) rb.velocity += Vector2.down *
Speed;
}

```

Алгоритм контролю тригерів також імплементовано у скрипт *PlayerMovement.cs*. Для початку був створений елемент на який було додано компонент Box Collider 2D та встановлена властивість Is Trigger. У самому тілі скрипта ми викликаємо стандартний метод *OnTriggerEnter2D(Collider2D col)* у якому робиться перевірка за іменем об'єкта. І якщо наш гравець заходить в тригер, у нашому випадку це флаг фінішу, то завантажуються сцена End. Реалізація алгоритму в коді гри:

```

void OnTriggerEnter2D(Collider2D col)
{
    if (col.gameObject.name == "finish")
    {
        SceneManager.LoadScene("End");
    }
}

```

Імпліментация таймеру у даний проект виконана в скрипті *Timer.cs*. Спочатку був на сцені Game був створений елемент Canvas, у ньому був доданий прямокутник, який відіграє роль HUD'у, на було розташовано 2 елементи Text для позначення таймеру. Також на сцені був створений пустий GameObject для якого був доданий компонент скрипта *Timer.cs*. У тілі скрипта задеклароване приватне поле `timer` за допомогою *TimeSpan()* та поле `Text seconds` за допомогою якого буде отриманий доступ до текстового елемента. У методі `Update()` додається проміжок часу за допомогою метода `timer.Add(TimeSpan.FromSeconds(Time.deltaTime))` та додаємо секунди с таймеру до текстового елемента переводячи його у строковий елемент за допомогою метода `ToString()`. А далі виконується перевірка на проміжок часу і якщо умова виконується, то гра закінчується і завантажуються сцена `End`. Реалізація алгоритма таймеру:

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

public class Timer : MonoBehaviour
{
    private TimeSpan timer = new TimeSpan();
    public Text seconds;

    private void Update()
    {
        timer = timer.Add(TimeSpan.FromSeconds(Time.deltaTime));
        seconds.text = timer.Seconds.ToString("00");
        if (seconds.text == "60")
        {
            SceneManager.LoadScene("End");
        }
    }
}
```

Генерація сітки виконана у скрипті *MazeGenerator.cs*, а саме у методі *GenerateMaze()*. Цей алгоритм є проміжним і використовується у подальшій генерації і виділенні клітинок, але заслуговує уваги. Для початку був створений пустий *GameObject* з назвою *Cell*, який буде представляти одну клітинку лабіринту. І до нього додали ще два об'єкта з компонентом *LineRenderer* для відрисовки стін, та були названі як *WallLeft* та *WallBottom*. До цих двох елементів був доданий компонент *EdgeCollider*, для створення колізії у стін. Об'єкт *Cell* був збережений як префаб та був видалений зі сцени. Для створення сітки та подальшої генерації ми створили клас *MazeGenerator* у якому були задекларовані публічні поля ширини та висоти лабіринту та публічний метод *GenerateMaze()*, який і створює сітку лабіринту. Також був створений клас *MazeGeneratorCell*, який буде зберезувати наш двомірний масив з лабіринтом, та має поля присутності стін. У класі *MazeGenerator* в методі *GenerateMaze()* утворюється двомірний масив *MazeGeneratorCell[,] cells = new MazeGeneratorCell[Width, Height];* у який передаються його розміри, а саме ширина та висота. І потім ми проходимо двома циклами по елементам масива у яких ми будемо створювати *cells[x, y]*. Та у кінці методу ми повернемо наш лабіринт за допомогою *return*. Даний метод буде доповнено у наступних алгоритмах. Приклад реалізації наведено нижче:

```
public int Width = 23;
public int Height = 15;

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 };
        }
    }
    return cells;
}
```

Алгоритм видалення стінок для генерації лабіринту — це основна реалізація алгоритму Recursive backtracker. Для імплементації цього алгоритму був створений приватний метод *WallsBackTracker()* який приймає в якості аргумента двовірний масив *MazeGeneratorCell*. Для початку ми вибрали стартову клітинку та *MazeGeneratorCell current = maze[0, 0]*; Для виконання дій алгоритма нам потрібен *Stack*. Тому що кожен крок алгоритма ми рухаємось у випадковому напрямку та зносимо стіну в цю сторону, коли ми доходимо до тупика ми прямуємо назад до моменту де ми можемо кудись рушитися, так повторюється поки лабіринт не буде побудовано і ми не прийдемо у стартову точку. Для того, щоб мати можливість прямувати назад було використано *Stack*. Далі створювався цикл *do...while* який виконує дію доки у нашому стеці є клітинки. У тілі циклу створюємо *List* з клітинами які ми ще не відвідали, для перевірки відвідана клітинка чи ні, було створена булева змінна у класі *MazeGeneratorCell* — *isVisited* зі стандартним значенням *false*, а у методі *WallsBackTracker()* вже робимо її *true* для *current* клітинки. Далі йде перевірка усіх чотирьох сусідів *current* клітинки. Якщо умова перевірки дійсна, ми додаємо клітинку до *List unvisitedCells* за допомогою методу *unvisitedCells.Add()*. У наступному кроці створюється перевірка на те, що якщо у *unvisitedCells* є елементи, ми обираємо випадкового та зайти у нього цю клітинку ми присвоюємо змінній *choosen*. Потім за допомогою методу *RemoveWall()* який приймає у якості аргументів клітинку у якій ми знаходимося та наступну обрану, ми видаляємо непотрібну стіну. Метод *RemoveWall()* буде описаний нижче. Потім ми помічаємо наступну обрану клітинку *choosen* як *isVisited*, для того щоб не зайти до неї знову та переходимо до неї. Також потрібно додати обрану клітинку у наш стек за допомогою методу *stack.Push(choosen)*. І якщо умова перевірки не виконується і у нашому списку немає елементів, ми робимо *current = stack.Pop()*; для того щоб перейти назад. Алгоритм методу *RemoveWall()* виконаний наступним чином: Так як у метод було передано дві сусідні стіни, вони сусідні або по горизонталі, або по вертикалі і ми перевіряємо це, якщо умова виконується, то поля *WallLeft* та *WallBottom* які описані у класі *MazeGeneratorCell* стануть *false*. І останнє що ми робимо — це визиваємо метод *WallsBackTracker()* у методі *GenerateMaze()*. Реалізація методів *WallsBackTracker()* та *RemoveWall()* показано нижче:

```
//This method removes walls from MazeGeneratorCell array via
"Backtracker algorithm"

private void WallsBackTracker(MazeGeneratorCell[,] maze)
```

```

{
    MazeGeneratorCell current = maze[0, 0];

    current.isVisited = true;
    current.DistanceFromStart = 0;

    Stack<MazeGeneratorCell> stack = new
Stack<MazeGeneratorCell>();
    do
    {
        List<MazeGeneratorCell> unvisitedCells = new
List<MazeGeneratorCell>();

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

        //Check neighboring cells
        if (x > 0 && !maze[x - 1, y].isVisited)
unvisitedCells.Add(maze[x - 1, y]);
        if (y > 0 && !maze[x, y - 1].isVisited)
unvisitedCells.Add(maze[x, y - 1]);
        if (x < Width - 2 && !maze[x + 1, y].isVisited)
unvisitedCells.Add(maze[x + 1, y]);
        if (y < Height - 2 && !maze[x, y + 1].isVisited)
unvisitedCells.Add(maze[x, y + 1]);

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

            choosen.isVisited = true;
            stack.Push(choosen);
            choosen.DistanceFromStart = current.DistanceFromStart +
1;

            current = choosen;
        }
    }
    else

```

```

        current = stack.Pop();

    } while (stack.Count > 0);
}

//Remove wall method definition
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;
    }
}
}

```

У ході виконання генерації ми отримали вертикальні та горизонтальні зайві стіни зверху та праворуч тому що використовувалися лише ліва та нижня стіна. Для цього було створено метод *RemoveBrokenWall()* який приймає масив *MazeGeneratorCell[,] maze* у якості аргумента. Виконується це дуже просто, за допомогою двох циклів ми проходимо по координатах x та y лабіринту, та видаляємо крайні стіни. Даний метод визивається у методі *GenerateMaze()*. Реалізація цього алгоритму наведена нижче:

```

public void RemoveBrokenWall(MazeGeneratorCell[,] maze)
{
    for (int x = 0; x < maze.GetLength(0); x++)
    {
        maze[x, Height - 1].WallLeft = false;
    }
    for (int y = 0; y < maze.GetLength(1); y++)
    {
        maze[Width - 1, y].WallBottom = false;
    }
}
}

```


Алгоритм розміщення виходу був розроблений наступним чином. Для розрахунку ідеальної позиції для виходу ми створимо публічну змінну у класі *MazeGeneratorCell*, а саме змінну *DistanceFromStart*. Ця змінна описує від старту до цієї клітинки та присоюємо їй значення 0 у методі *WallsBackTracker()* для поточної клітинки і потім прибавляємо одиницю у тілі циклу *do...while*. У якості фініша ми розглядаємо тільки клітинки які знаходяться з країв лабіринту. Для старту ми створюємо змінну майбутньої клітинки *furthest* та проходимо двома циклами *for* по координатам x та y з країв лабіринту. Далі, коли ми знайшли найвіддаленішу клітинку ми робимо перевірку задля того щоб видалити стіну виходу. Так як ми отримали позицію фінішної точки, ми можемо поставити фінішний флаг, який був створений раніше у ігровій сцені та позначений як тригер. Реалізація цього алгоритму у коді гри:

```
//Placing exit from maze
private Vector2Int PlaceMazeExit(MazeGeneratorCell[,] maze)
{
    MazeGeneratorCell furthest = maze[0, 0];

    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];
    }

    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;
        FinishFlag = GameObject.Find("finish");

        FinishFlag.transform.position = new Vector2(furthest.X + 0.5f,
furthest.Y+ 0.5f);
        return new Vector2Int(furthest.X, furthest.Y);
    }

```

Поміщення лабіринта на сцену виконуються у скрипті *MazeSpawner.cs*. Для початку ми створили пустий *GameObject* з іменем *MazeSpawner* до якого був доданий новий компонент скрипт *MazeSpawner.cs*. У тілі скрипта ми декларуємо публічне поле *GameObject CellPrefab*, яке надає доступ до префабу клітинок, поле *Maze maze* — доступ до лабіринту, *HintRenderer HintRenderer* яке дає доступ до скрипта з підказкою (буде описаний пізніше), та поле *Vector3 CellSize* з розмірами нашої клітинки. Далі у стандартному методі *Start()* ми створюємо новий генератор, потім викликаємо у нього метод *GenerateMaze()* та збережемо це у масив *maze*. Далі проходимо по масиву з клітинками за допомогою двох циклів та клонуємо клітинки через стандартний метод *Instantiate()*, а потім робимо стіни активними за допомогою методу *SetActive()*. Приклад поміщення лабіринту на сцену у коді гри:

```

public GameObject CellPrefab;
public HintRenderer HintRenderer;
public Maze maze;
public Vector3 CellSize = new Vector3(1, 1, 0);

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 Vector2(x, y),
Quaternion.identity).GetComponent<Cell>();

            c.WallLeft.SetActive(maze.cells[x, y].WallLeft);
            c.WallBottom.SetActive(maze.cells[x, y].WallBottom);

```

```

    }

    }

}

```

Для створення шляху-підказки на ігровій сцені було створено пустий *GameObject* з іменем *Hint* та скопійовано на нього компонент *LineRenderer* з однієї з стін та змінено колір та розмір. Далі у тілі скрипта робиться публічне посилання на клас *MazeSpawner*, для доступу до лабіринту, також був декларован елемент *LineRenderer* для подальшого надання йому точок позицій. Алгоритм підказки розроблений наступним чином: початок шляху підказки буде стартувати з фінішу та йти до старту, а так як у кожної клітинки є властивість *DistanceFromStart*, то алгоритм буде при рендері шляху, спиратися на змінну *DistanceFromStart - 1* у сусідніх клітинках. Реалізація даного алгоритму:

```

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

    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());
}

```

3.3 Основні рішення щодо модульного уявлення гри

Архітектура гри створена таким чином, щоб розділити усі асети між різними директоріями для зручного сприйняття та подальшої розробки. Кореневою папкою проекту є папка Assets, яка в свою чергу розбита на чотири піддиректорії, а власне:

- 1) Scripts — папка зі скриптами гри, вона в свою чергу розділена на 3 піддиректорії:
 - a) Controllers — контролери гри.
 - i) CameraController.cs — контролер камери, створює трекінг камери, яка слідує за гравцем.
 - ii) HintRenderer.cs — скрипт генерації підказки-шляху для гравця.
 - iii) PlayerMovement.cs — логіка пересування гравця та взаємодія з тригерами.
 - iv) Timer.cs — скрипт таймеру.
 - b) Maze — логіка лабіринту.

- i) Cell.cs — допоміжний клас за допомогою якого виконується простий доступ до стін клітинок.
 - ii) Maze.cs — клас який зберігає масив лабіринту та фінішну позицію.
 - iii) MazeGenerator.cs — клас у якому описуються методи створення лабіринту та генерація виходу.
 - iv) MazeSpawner.cs — клас рендеру лабіринту та шляху-підказки.
- c) Screens — логіка управління сцен.
 - i) MenuControl.cs — клас управління методами переходів між сценами та виходу з гри.
- 2) Sprites — папка зі спрайтами гри:
 - a) player.png — основний спрайт гравця.
 - b) finish.png — спрайт прапорцю фінішу.
- 3) Scenes — папка зі сценами ігрового проекту, яка містить:
 - a) End.unity — кінцева сцена гри. Завантажується після поразки гравця або фінішу.
 - b) Game.unity — основна сцена гри, на якій проходить ігровий процес.
 - c) Menu.unity — сцена головного меню.

3.4 Особливості реалізації системи

В ході розробки форм були використані різні елементи інтерфейсу, які вбудовані в Unity.

Розглянемо елементи головного меню (рис. 3.1):

- 1) елемент Canvas.
- 2) елемент Panel.
- 3) елемент Text.
- 4) елементи Button.



Рисунок 3.1 – Головне меню.

Далі розглянемо елементи ігрового HUD. (рис. 3.2):

- 1) елемент Canvas.
- 2) елемент Sprite.
- 3) елементи Text

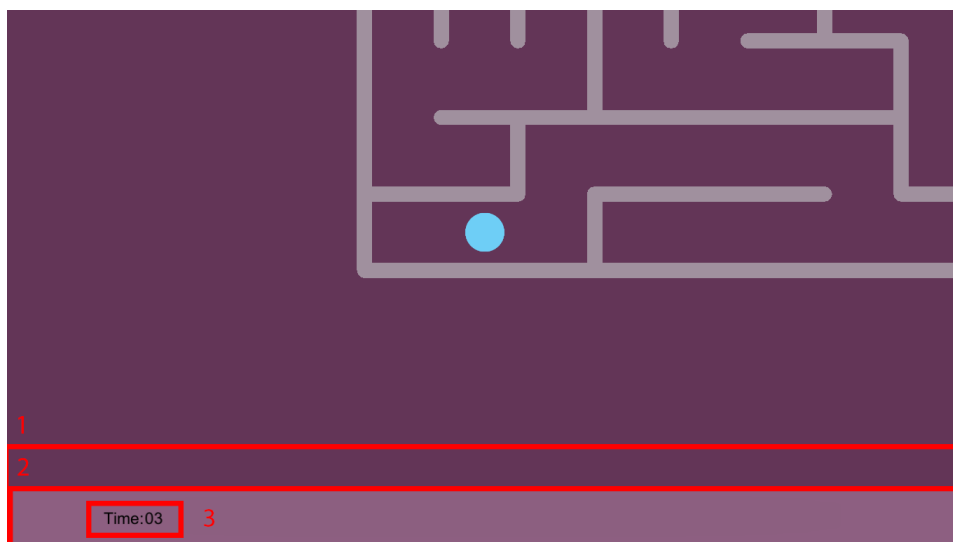


Рисунок 3.2 – Ігровий HUD.

Розглянемо елементи головного меню (рис. 3.1):

- 1) елемент Canvas.
- 2) елемент Panel.
- 3) елемент Text.
- 4) елементи Button.

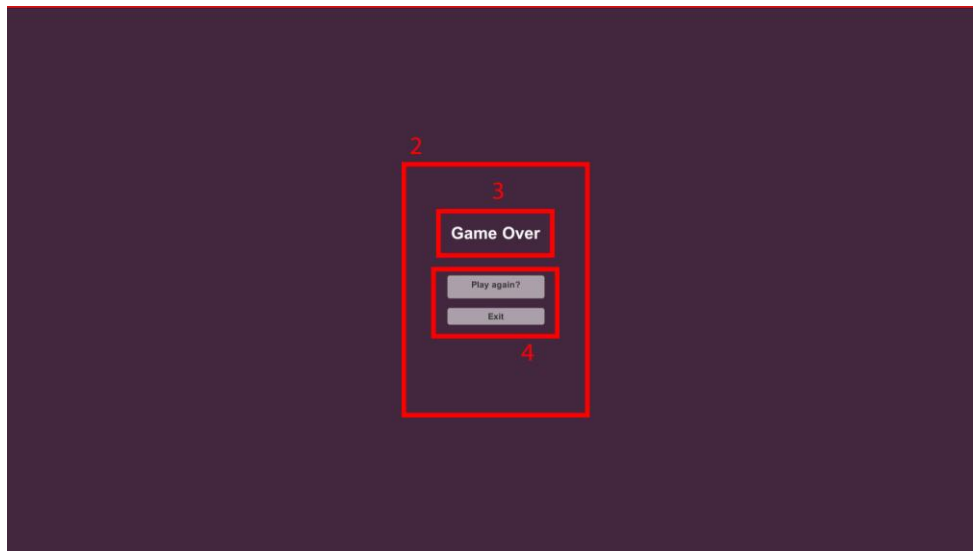


Рисунок 3.3 – Кінцеве меню.

3.5 Висновки з розділу

В ході роботи над грою курсового проекту було розроблено такі класи та описані їх методи: `PlayerMovement`, `MenuControl`, `CameraController`, `Cell`, `HintRenderer`, `Maze`, `MazeGeneratorCell`, `MazeGenerator`, `MazeGeneratorCell`, `MazeSpawner`, `Timer`.

При виконанні курсового проекту було використано алгоритм створення алгоритмів `Recursive backtracking`. Таким чином був згенерований лабіринт у грі. Використовувались стандартні бібліотеки `.NET` такі як: `System` та `System.Collections.Generic` для створення колекцій елементів. Також, були використані і стандартні бібліотеки `Unity`: `UnityEngine`, `UnityEngine.SceneManagement`, `UnityEngine.UI`.

У ході побудови користувацького інтерфейсу були використані такі як: `Canvas`, `Panel`, `Text`, `Button`, `Sprite`.

4 ПОСІБНИК ПРОГРАМІСТА

4.1 Призначення та умови застосування програми

Гра «Лабіринт» — це весела гра головоломка для веселого проведення часу та тренування своєї кмітливості.

Умови застосування для коректної роботи програми:

- 1) використання останніх версій ПЗ;
- 2) підключення до мережі Інтернет;

4.2 Характеристики гри

Гра виконана за допомогою мови програмування високого рівня C# за допомогою рушія гри Unity 2019.3.7f1. Проект (рис.4.1) містить класи, їх реалізацію, файли сцен, спрайтів, префабів.

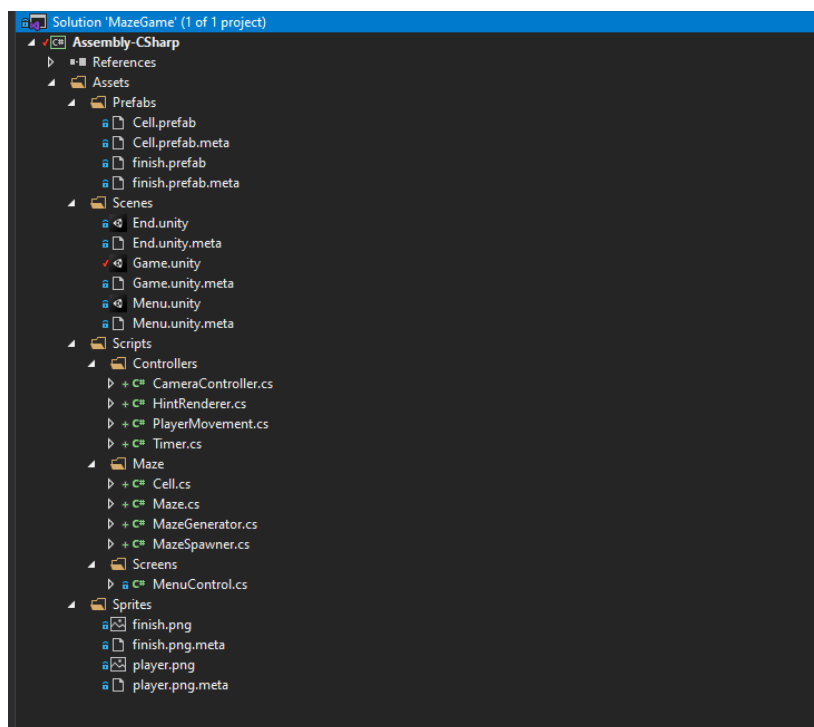
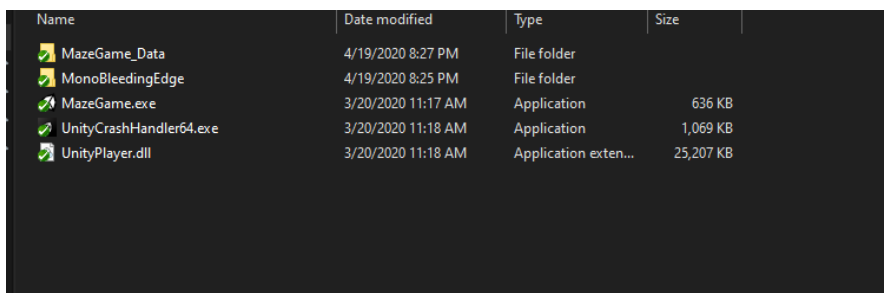


Рисунок 4.1 – Структура проекту

4.3 Звертання до гри

Для звернення до гри потрібно запустити Unity 2019.3.7f1, та додати проект MazeGame до хабу. Потім в редакторі Unity натиснути Ctrl + B та вибрати папку для компіляції проекту. У скомпільованому вигляді гра має таку структуру (рис.4.2)



Name	Date modified	Type	Size
MazeGame_Data	4/19/2020 8:27 PM	File folder	
MonoBleedingEdge	4/19/2020 8:25 PM	File folder	
MazeGame.exe	3/20/2020 11:17 AM	Application	636 KB
UnityCrashHandler64.exe	3/20/2020 11:18 AM	Application	1,069 KB
UnityPlayer.dll	3/20/2020 11:18 AM	Application exten...	25,207 KB

Рисунок 4.2 – Структура скомпільованого проекту

5 ІНСТРУКЦІЯ КОРИСТУВАЧА

5.1 Призначення програми

Гра «Лабіринт» — це весела гра головоломка для веселого проведення часу та тренування своєї кмітливості.

5.2 Умови виконання гри

Умови застосування для коректної роботи програми:

- а) використання останніх версій ПЗ;
- б) підключення до мережі Інтернет;

5.3 Як запустити програму

Щоб розпочати ігровий процес гравець потрібен запустити виконуючий файл гри — MazeGame.exe. Гра буде готова до роботи.

5.4 Виконання програми

Після того, як було запущено гру, перед користувачем головне меню гри, де у нього є вибір: зіграти у гру або вийти. (рис. 5.1).



Рисунок 5.1 – Головне меню

Якщо користувач натискає Play, то його перенаправляє на сцену ігрового процесу (рис. 5.2), або користувач може натиснути Exit та вийти з гри.

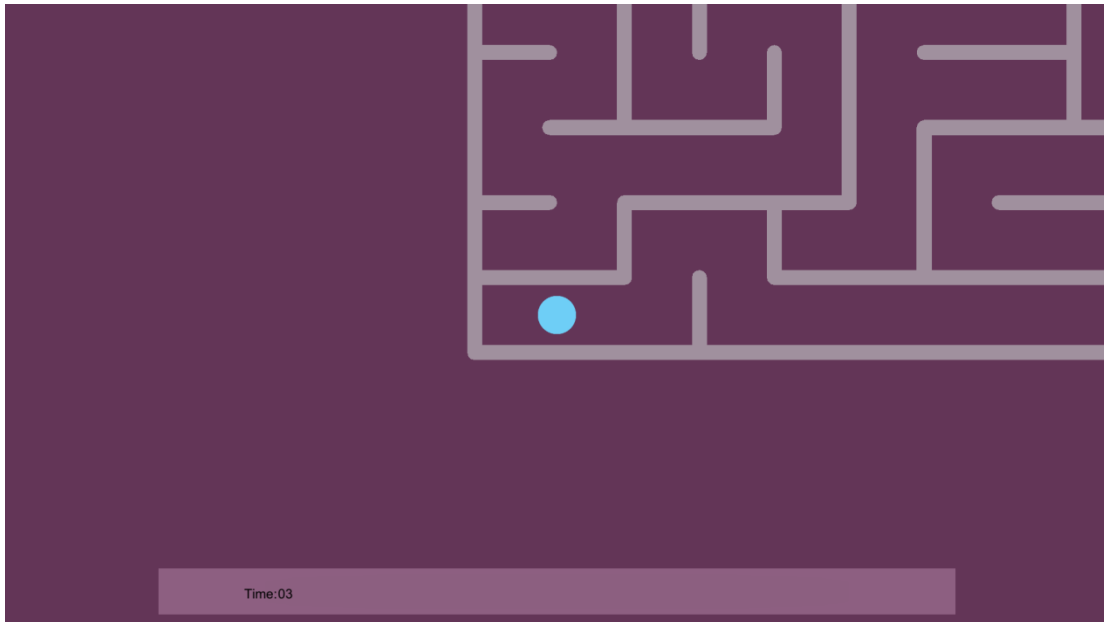


Рисунок 5.2 – Сцена ігрового процесу.

Після переходу до сцени гри перед користувачем з'являється згенерований лабіринт, елемент гравця, та таймер. Ціль гри — вийти з лабіринту за 60 секунд.

Основні елементи керування персонажем гравця:

- 1) W - рух вгору.
- 2) S - рух вниз.
- 3) A - рух ліворуч.
- 4) D - рух праворуч.
- 5) R - показати шлях-підказку.
- 6) F - приховати шлях-підказку

Якщо гравець заплутався та не знає як знайти вихід, можна активувати підказку, яка покаже шлях до виходу (рис. 5.3).

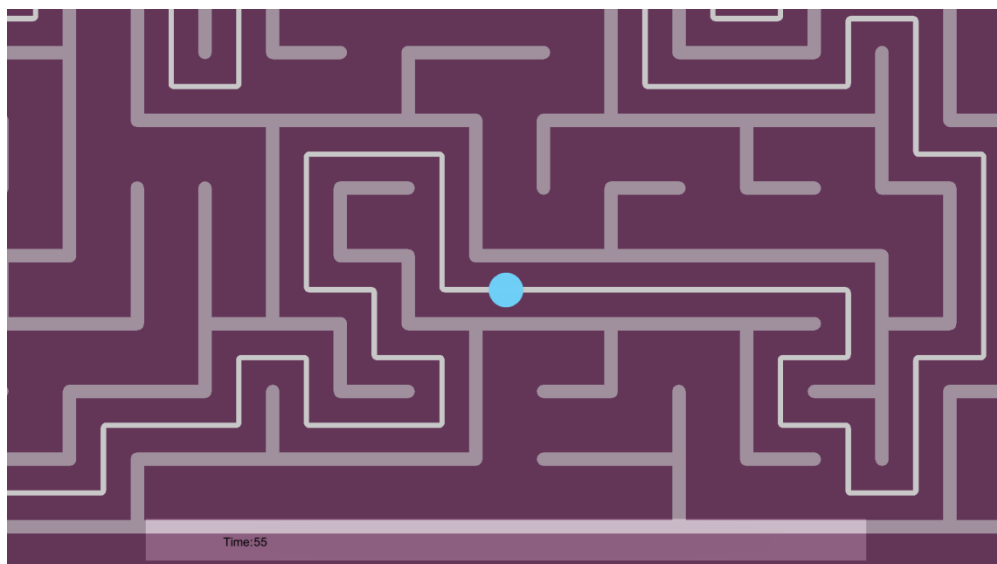


Рисунок 5.3 – Шлях-підказка для гравця.

Якщо гравець дістався до виходу, або не впорався вчасно, його перенаправляє у меню закінчення гри (рис. 5.4) де у нього є вибір: зіграти ще раз або вийти у головне меню.

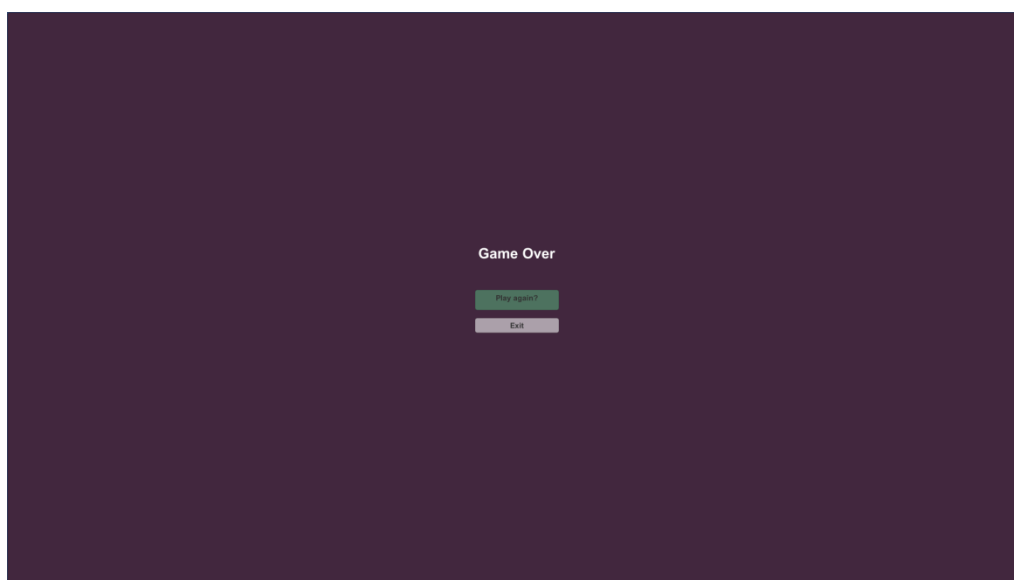


Рисунок 5.4 – Меню закінчення гри .

ВИСНОВКИ

У процесі виконання даного курсового проекту була розроблена гра «Лабіринт» з використанням алгоритмів процедурної генерації клітинок та стін лабіринту. Також були розроблені підказки для гравців, основні меню та мінімальний ігровий процес.

В процесі виконання роботи було виконано аналіз та огляд існуючих рішень, таких як мобільна гра Mazes & More та Flow Free. Були розглянуті класифікації лабіринтів та алгоритми їх генерації. Також було розглянуто основні можливості мови C# та класів стандартної бібліотеки .NET які були використані у проекті. Далі були розглянуті основні можливості рушія гри Unity та найпопулярніші ігри на цій платформі.

У ході розробки курсового проекту були вирішені наступні задачі:

- 1) виконано огляд сучасних програмних засобів.
- 2) проаналізовано існуючі рішення та алгоритми побудови лабіринтів
- 3) розглянуті особливості мови C# та можливості рушія гри Unity.
- 4) була розроблена гра «Лабіринт» з процедурною генерацією лабіринтів, основними меню та мінімальним ігровим процесом.

ПЕРЕЛІК ПОСИЛАНЬ

1. C Sharp – Вікіпедія: [Електрон. ресурс]. – Режим доступу:
https://uk.wikipedia.org/wiki/C_Sharp
2. Движок Unity – особенности, преимущества и недостатки – cubiq: [Електрон. ресурс]. – Режим доступу: <https://cubiq.ru/dvizhok-unity/>
3. Язык программирования C#: краткий обзор – [Електрон. ресурс]. – Режим доступу: <https://techrocks.ru/2019/02/16/c-sharp-programming-language-overview/>
4. Обзор коллекций –: [Електрон. ресурс]. – Режим доступу:
https://professorweb.ru/my/csharp/charp_theory/level12/12_1.php
5. Unity (рушій гри) – Вікіпедія: [Електрон. ресурс]. – Режим доступу:
[https://uk.wikipedia.org/wiki/Unity_\(рушій_гри\)](https://uk.wikipedia.org/wiki/Unity_(рушій_гри))
6. Unity Scripting API : [Електрон. ресурс]. – Режим доступу:
<https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>
7. .NET Framework – Вікіпедія: [Електрон. ресурс]. – Режим доступу:
https://uk.wikipedia.org/wiki/.NET_Framework

ДОДАТОК А Текст програми

Скрипт CameraController.cs

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

public class CameraController : MonoBehaviour
{
    public Vector3 offset;
    public Transform player;
    public float smooth = 5.0f;

    //Create smooth tracking camera
    void Update()
    {
        transform.position = Vector3.Lerp(transform.position,
player.position + offset, Time.deltaTime * smooth);
    }
}
```

Скрипт HintRenderer.cs

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

public class HintRenderer : MonoBehaviour
{
    public MazeSpawner MazeSpawner;

    private LineRenderer componentLineRenderer;

    private void Start()
    {
        componentLineRenderer = GetComponent<LineRenderer>();
    }

    void Update()
    {
        if (Input.GetKey(KeyCode.R)) componentLineRenderer.sortingOrder
= 1;
    }
}
```

```

        if (Input.GetKey(KeyCode.F)) componentLineRenderer.sortingOrder
= -1;
    }
    public void DrawPath()
    {
        Maze maze = MazeSpawner.maze;
        int x = maze.finishPos.x;
        int y = maze.finishPos.y;
        List<Vector3> positions = new List<Vector3>();

        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());
}
}

```

Скрипт PlayerMovement.cs

```

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

public class PlayerMovement : MonoBehaviour
{
    public float Speed = 2f;
    private Rigidbody2D rb;

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

    void Update()
    {
        HandleMovement();
    }

    void HandleMovement()
    {
        rb.velocity = Vector2.zero;
        if (Input.GetKey(KeyCode.A)) { rb.velocity += Vector2.left *
Speed; }
        if (Input.GetKey(KeyCode.D)) rb.velocity += Vector2.right *
Speed;
        if (Input.GetKey(KeyCode.W)) rb.velocity += Vector2.up * Speed;
        if (Input.GetKey(KeyCode.S)) rb.velocity += Vector2.down *
Speed;
    }
}

```

```

void OnTriggerEnter2D(Collider2D col)
{
    if (col.gameObject.name == "finish")
    {
        SceneManager.LoadScene("End");
    }
}
}

```

Скрипт Timer.cs

```

using System;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class Timer : MonoBehaviour
{
    private TimeSpan timer = new TimeSpan();
    public Text seconds;

    private void Update()
    {
        timer = timer.Add(TimeSpan.FromSeconds(Time.deltaTime));
        seconds.text = timer.Seconds.ToString("00");
        if (seconds.text == "60")
        {
            SceneManager.LoadScene("End");
        }
    }
}

```

Скрипт Cell.cs

```

using UnityEngine;

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

```

Скрипт Maze.cs

```

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

public class Maze
{
    public MazeGeneratorCell[,] cells;
    public Vector2Int finishPos;
}

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

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

    public bool isVisited = false;

    public int DistanceFromStart;
}

```

Скрипт MazeGenerator.cs

```

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

public class MazeGenerator
{
    public int Width = 23;
    public int Height = 15;
    public GameObject FinishFlag;

    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 };
        }
    }
    RemoveBrokenWall(cells);
    WallsBackTracker(cells);

    Maze maze = new Maze();

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

    return maze;
}

//This method removes walls from MazeGeneratorCell array via
"Backtracker algorithm"
private void WallsBackTracker(MazeGeneratorCell[,] maze)
{
    MazeGeneratorCell current = maze[0, 0];

    current.isVisited = true;
    current.DistanceFromStart = 0;

    Stack<MazeGeneratorCell> stack = new
Stack<MazeGeneratorCell>();
    do
    {
        List<MazeGeneratorCell> unvisitedCells = new
List<MazeGeneratorCell>();

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

        //Check neighboring cells
        if (x > 0 && !maze[x - 1, y].isVisited)
unvisitedCells.Add(maze[x - 1, y]);

```

```

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

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

            choosen.isVisited = true;
            stack.Push(choosen);
            choosen.DistanceFromStart = current.DistanceFromStart +
1;

            current = choosen;
        }
        else
            current = stack.Pop();

    } while (stack.Count > 0);
}

//Placing exit from maze
private Vector2Int PlaceMazeExit(MazeGeneratorCell[,] maze)
{
    MazeGeneratorCell furthest = maze[0, 0];

    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];
    }

    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;
    FinishFlag = GameObject.Find("finish");

    FinishFlag.transform.position = new Vector2(furthest.X + 0.5f,
furthest.Y+ 0.5f);
    return new Vector2Int(furthest.X, furthest.Y);
}

//Method consider removing side-walls
public void RemoveBrokenWall(MazeGeneratorCell[,] maze)
{
    for (int x = 0; x < maze.GetLength(0); x++)
    {
        maze[x, Height - 1].WallLeft = false;
    }
    for (int y = 0; y < maze.GetLength(1); y++)
    {
        maze[Width - 1, y].WallBottom = false;
    }
}

//Remove wall method definition
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;
    }
}

```

```

    }
}

```

Скрипт MazeSpawner.cs

```

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

public class MazeSpawner : MonoBehaviour
{
    public GameObject CellPrefab;
    public HintRenderer HintRenderer;
    public Maze maze;
    public Vector3 CellSize = new Vector3(1, 1, 0);

    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 Vector2(x, y),
Quaternion.identity).GetComponent<Cell>();

                c.WallLeft.SetActive(maze.cells[x, y].WallLeft);
                c.WallBottom.SetActive(maze.cells[x, y].WallBottom);
            }
        }
    }

    void Update()
    {
        HintRenderer.DrawPath();
    }
}

```

Скрипт MenuControl.cs

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

public class MenuControl : MonoBehaviour
{
    public void PlayPressed()
    {
        SceneManager.LoadScene("Game");
    }

    public void ExitPressed()
    {
        Debug.Log("EXIT PRESSED");
        Application.Quit();
    }

    public void MenuPressed()
    {
        SceneManager.LoadScene("Menu");
    }
}
```


ДОДАТОК Б Інтерфейс гри



Рисунок Б.1 – Головне меню

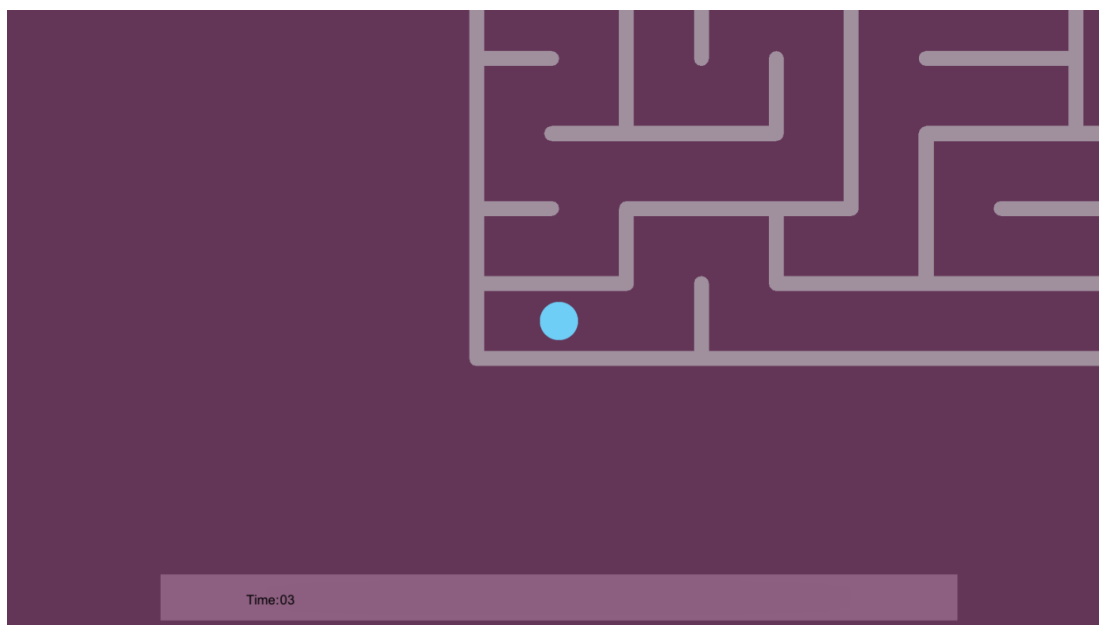


Рисунок Б.2 – Сцена ігрового процесу.

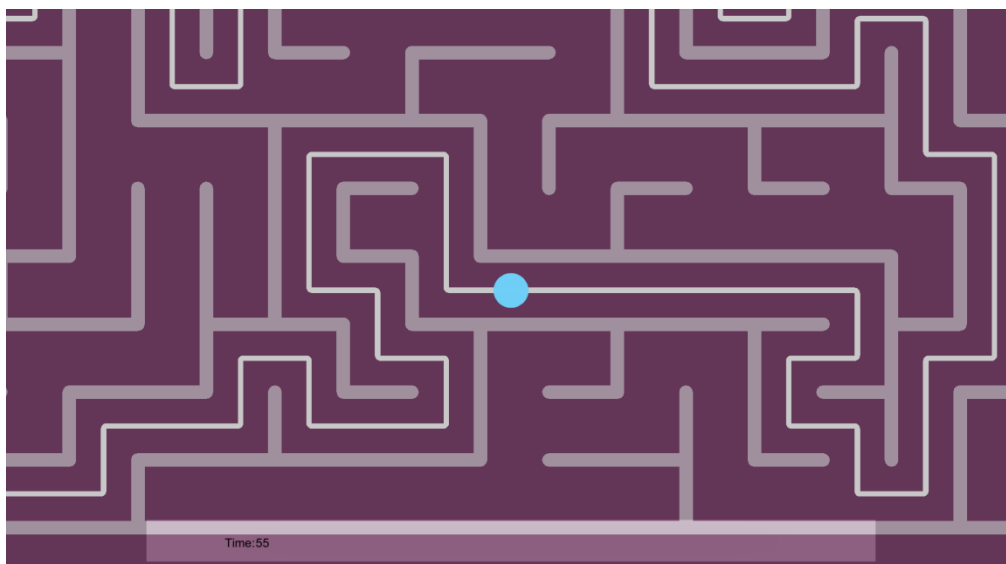


Рисунок Б.3 – Шлях-підказка для гравця.

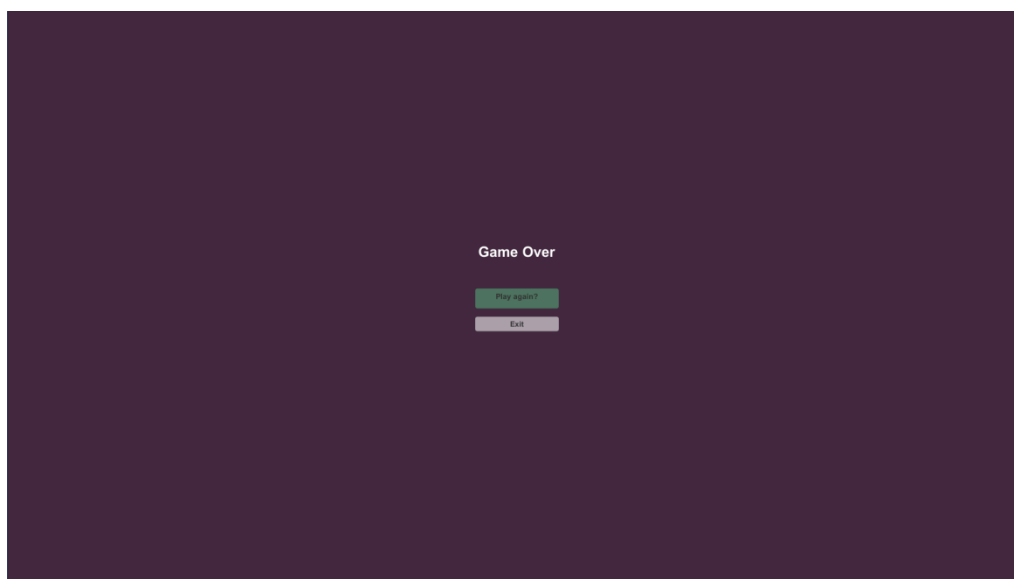


Рисунок Б.4 – Меню закінчення гри .