Національний лісотехнічний університет України

(повне найменування вищого навчального закладу)

Інститут деревообробних та комп'ютерних технологій і дизайну

(повне найменування інституту, назва факультету(відділення))

Кафедра інформаційних технологій

(повна назва кафедри (предметної циклової комісії))

Пояснювальна записка

до дипломної роботи

перший (бакалаврський)

(освітньо-кваліфікаційний рівень)

на тему: **Розробка інтерактивної гри "Побудова лабіринтів" для дошкільнят**

Виконав студент групи ІСТС-21

126 „Інформаційні системи та технології"

(шифр і назва напрямку підготовки спеціальності)

Шишковський І.О.

(прізвище, ініціали)

Керівник: Карашецький В.П.

(прізвище, ініціали)

Рецензент: _____

(прізвище, ініціали)

Львів – 2022

Національний лісотехнічний університет України

(повне найменування вищого навчального закладу)

ННІ деревооброблювальних та комп'ютерних технологій і дизайну

Кафедра Інформаційних технологій

Рівень вищої освіти перший (бакаравський)

Спеціальність 126 „Інформаційні системи та технології"

ЗАТВЕРДЖУЮ:

Завідувач кафедри ІТ

_____*Крошний І.М.*

„___" _____ *2022*

ЗАВДАННЯ

НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТУ

Шишковський Ілля Олегович

(прізвище, ім'я, по батькові)

1.Тема бакалаврска роботи: «Розробка інтерактивної гри «Побудова лабіринтів» для дошкільнят»

керівник роботи : доц. Карашецький В.П., к.т.н ,

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від "11" 10 2021 року, №С-528

2.Термін подання студентом роботи 14 червня 2022р.

3. Вихідні дані до роботи Розробити програмну реалізацію та візуалізацію роботи ігрового застосунку "Лабіринт". Створити застосунок засобами MonoGame для ОС Android, також реалізувати алгоритм генерації лабіринту з скінченним заданим розміром. Розробити алгоритм пошуку шляху в лабіринті та реалізувати відмальовування 2Д графіки в застосунку.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

Стан проблемної області

Інформаційне забезпечення

Програмне забезпечення

Висновки

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

Підготовка матеріалу до доповіді.

6. Консультанти розділів проекту (роботи)

7. Дата видачі завдання 15 грудня 2021р.

КАЛЕНДАРНИЙ ПЛАН

№, з/п	Етапи бакалаврської роботи	Термін виконання етапів роботи	Примітка
1.	Огляд літератури згідно досліджуваної теми. Збір необхідних матеріалів.	15.12-25.01	Виконано
2.	Постановка задачі.	25.01-10.02	Виконано
3.	Розроблення дизайну та ідеї гри та Реалізація алгоритмів генерації лабіринту	10.02-20.02	Виконано
4.	Накладання алгоритмів для генерування рівнів у грі.	20.02-12.03	Виконано
5.	Реалізація пошуку шляху у лабіринті алгоритмом A*	12.03.-01.04.	Виконано
6.	Додавання алгоритмів в ігровий рушій	01.04.-05.04.	Виконано
7.	Здача пояснювальної записки на рецензування. Підготовка доповіді.	05.05.-14.06.	Виконано

Студент _____ Шишковський І.О.

(підпис) (прізвище та ініціали)

Керівник роботи _____ Карашецький В.П.

(підпис) (прізвище та ініціали)

РЕФЕРАТ

Дипломна робота містить 49 сторінок пояснювальної записки, 34 рисунків, 1 таблицю, 3 додатки, 10 джерел.

У даній дипломній роботі досліджується технологія MonoGame та її використання в поєднанні з XamarinForms. Вебзастосунок оптимізовано під пристрої на ОС Android. Інтерфейс зручний для користування, керування грою відлагоджено. Прописано всі можливості побудови лабіринтів та пошуку шляху у них. Дана гра матиме яскраву та динамічну 2Д графіку з картами колізій.

Ключові слова: MonoGame, алгоритми, Android, графіка, колізія, мапінг.

ABSTRACT

Graduate work contains 49 pages of explanatory note, 34 images, 1 table, 3attachments, 10 sources.

This thesis examines MonoGame technology and its use in combination with XamarinForms. The web application is optimized for different types of devices on Android. The interface is easy to use, game control is debugged. All possibilities of construction of birints and search of a way in them are registered. This game will have bright and dynamic 2D graphics with collision maps.

Keywords: MonoGame, algorithms, Android, graphics, collision, mapping.

ТЕХНІЧНЕ ЗАВДАННЯ

Розробити програмну реалізацію та візуалізацію роботи ігрового застосунку “Лабіринт”, яка передбачає виконання наступних завдань:

- Створити застосунок засобами MonoGame для ОС Android;
- Реалізувати алгоритм генерації лабіринту з скінченним заданим розміром;
- Реалізувати алгоритм пошуку шляху в лабіринті;
- Реалізувати відмальовування 2Д графіки в застосунку;
- Оптимізувати роботу застосунку для ОС Android;
- Наповнити гру наступними функціональними можливостями:
  - генерування лабіринту заданого користувачем розміру
  - реалізований зручний user-friendly інтерфейс;
  - реалізований алгоритм пошуку шляху в генерованому лабіринті;
  - реалізований вибір складності гри;

ЗМІСТ

ВСТУП

З приходом сучасної комп'ютерної ери цифрові ігри стали дуже популярними завдяки їх візуальній інтерактивності та здатності демонструвати історії, знання, віртуальні світи та інтерактивні об'єкти в привабливий і ефектний спосіб. Більшість цифрових ігор розроблено в 2D або 3D форматі. Сьогодні велика кількість ігор містять в собі різного виду лабіринти. Лабіринти можуть використовуватись в різних видах ігор, до прикладу: навчальні ігри, екшн-ігри, квести, тайм кіллери та звичайні бродилки. Основна проблема, яка постає перед розробниками цих ігор це створення лабіринтів. Враховуючи те, що даний вид ігор володіє великим контентним розмаїттям, то лабіринтів доведеться створювати досить багато, відповідно витрачати час на малювання даних лабіринтів дуже накладно. В вирішенні даної проблеми нам допоможе створення алгоритмів генерації лабіринтів, які не просто здатні намалювати лабіринт самі, а й роблять його унікальним, ніколи не повторюючи вже існуючі конфігурації. Створення даного продукту можливе завдяки такій технології, як MonoGame.

Об'єктом дослідження є розробка ігрового застосунку, методами Xamarin Forms та MonoGame.

Метою роботи є створення гри використовуючи середовище VisualStudio при цьому реалізація 2Д графіки в даному середовищі.

Предметом дослідження є алгоритми генерації лабіринтів, їхня неповторювана унікальна конфігурація.

Практичне значення застосунку та алгоритму генерації дає можливість швидшого створення ігор, що містять систему лабіринтів.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1. Огляд проблемної області

Розробка додатків – це процес проектування, створення та впровадження програмних додатків. Даними проектами можуть займатися як великі організації з великими командами, так і один позаштатний розробник. Розробка додатків визначає процес створення програми і, як правило, слідує певній стандартній методології. Існує багато факторів, які впливають на те, як здійснюється розробка додатків. Слід враховувати розмір проекту, його гнучкість, наскільки велика команда розробників, наскільки досвідчена команда розробників та кінцевий термін виконання проекту. Адже сьогодні створення застосунків вимагає великих часових затрат та супроводжується певними зусиллями, розробник, який береться за реалізацію проекту повинен бути вправним програмістом, який володіє необхідними методами та програмами.

Зрештою, самого лимше створення застосунку не буде достатньо, оскільки користувачі можуть використовувати різні мобільні телефони з різними операційними системами (чи то на базі Android, чи то iOS і т.д.) Дана проблема стосується і користувачів комп’ютерів, які відповідно можуть користуватися різними операційними системами Windows чи Linux. Звісно, що ми не можемо відкидати шанувальників ігрових платформ, таких, як Xbox, Nintendo та PlayStation. Відповідно питання крос-платформності додатку відіграє дуже важливу роль в його популярності та просуванні.

Сьогодні користується попитом таке середовище Unity 3D. Unity – міжплатформне середовище розробки комп'ютерних ігор. Завдяки неперевершеним міжплатформним можливостям Unity користується популярністю як у розробників хобі, так і у студій AAA. Його використовували для створення таких ігор, як Pokemon Go, Heathstone, Rimworld, Cuphead та багато інших. Хоча 3D в назві, Unity 3D також містить інструменти для розробки 2D ігор.

Програмістів Unity 3D приваблює API сценаріями C# і вбудованою інтеграцією Visual Studio. Unity також пропонує JavaScript як мову сценаріїв і MonoDevelop як IDE для тих, хто хоче альтернативу Visual Studio. Але художники також люблять його, оскільки він поставляється з потужними інструментами анімації, які дозволяють легко створювати власні тривимірні ролики або створювати 2D-анімацію з нуля. У Unity можна анімувати майже все.

Також Unity 3D пропонує безкоштовну версію, щоб розробники могли випускати ігри, створені за допомогою Unity Personal, не сплачуючи за програмне забезпечення, якщо вони заробляють менше 100 000 доларів США на іграх, створених за допомогою Unity. Для тих, хто готовий платити, Unity пропонує деякі додаткові функції та гнучкий план ліцензування за багаторівневою моделлю підписки. Користувачі преміум-класу отримають доступ до вихідного коду Unity і підтримки розробників.

Оскільки Unity існує з 2005 року, вона зібрала величезну кількість користувачів і дивовижну бібліотеку ресурсів. Unity не тільки має фантастичну документацію, але й величезну кількість відео та навчальних посібників.

Мова, яка використовується в розробці ігор Unity, — це C#. Раніше ми також могли використовувати Javascript для створення ігор на Unity.

Але ми отримуємо більше можливостей для програмування з C# (цикли, умовні оператори, використання змінних і функцій). Будь-який творець ігор має на меті – створити власну поведінку гри за допомогою сценаріїв/програмування. Наприклад, зробити так, щоб звук продовжував відтворюватися безперервно, поки два об’єкти торкаються один одного. Коли вони втрачають контакт один з одним, відтворення звуку припиняється. Все це робиться за допомогою програмування, з мовою програмування, пов’язаною з ігровим двигуном, якою в даному випадку є C#. Кожен гейм-девелопер використовує бібліотеку Unity C# або API. Вся інформація та навчальні посібники є на веб-сайті Unity. Це додаткові функції, класи та змінні, спеціально створені для створення відеоігор на C# та Unity. Як ігровий движок, Unity може надати багато найважливіших

вбудованих функцій, які забезпечують роботу гри. Це означає такі речі, як фізика, 3D-рендерінг та виявлення зіткнень. З точки зору розробника, це означає, що не потрібно винаходити велосипед заново. Замість того, щоб починати новий проект, створюючи новий фізичний двигун з нуля – обчислюючи кожен останній рух кожного матеріалу або спосіб, яким світло має відбиватися від різних поверхонь.

Що робить Unity ще потужнішим, так це те, що він також включає процвітаючий і постійно діючий «Asset Store». По суті, це місце, куди розробники можуть завантажувати свої творіння та робити їх доступними для спільноти. Хочете мати гарний ефект вогню, але не маєте часу створити його з нуля? Перевірте магазин активів, і ви, ймовірно, щось знайдете. Усе це означає, що розробник гри може зосередитися на тому, що важливо: створити власну гру, при цьому кодуючи лише функції, унікальні для його бачення.

Крім ігрового двигуна, Unity є IDE. IDE означає «інтегроване середовище розробки», що описує інтерфейс, який надає вам доступ до всіх інструментів, необхідних для розробки, в одному місці. Програмне забезпечення Unity має візуальний редактор, який дозволяє творцям просто перетягувати елементи в сцени, а потім маніпулювати їх властивостями.

Програмне забезпечення Unity також надає безліч інших корисних функцій та інструментів: наприклад, можливість навігації по папках у проекті або створення анімації за допомогою інструмента шкали часу. Коли справа доходить до кодування, Unity перейде на альтернативний редактор на ваш вибір. Найпоширенішим варіантом є Visual Studio від Microsoft, яка здебільшого інтегрується без проблем.

Звичайно, для розробки доступні й інші великі ігрові двигуни. Ігровий двигун Unity стикається з жорсткою конкуренцією. Тому варта зазначити переваги Unity: Unity є кросплатформенна, а це означає, що створювати ігри для iOS, ПК чи навіть ігрових консолей так само легко. Unity пропонує чудову підтримку віртуальної реальності для тих розробників, які зацікавлені в розробці

для Oculus Rift або HTC Vive. Для розробника, орієнтованого на мобільні пристрої, Unity є найкращим варіантом.

Також широкою популярністю користується, таке середовище створення ігор, як Unreal Engine. В даний час Unreal Engine привернув увагу багатьох розробників та художників ігор. Широкий спектр інструментів, доступних у його комплекті, дає користувачам ліцензію на створення власних ігор. Він найбільш відомий своєю великою та легкою настроюваністю, а також володіє різними навичками платформи та інструментами для легкого створення HD AAA ігор. Більшість початківців і тих, хто починає працювати вперше, вибирають саме Unreal Engine. Дане середовище складається з частин, які включають графічну платформу, онлайн-модуль, фізичну платформу та звукову платформу, введення та ігровий механізм. У ньому є кілька редакторів, які допомагають у розробці гри. Під час запуску він має редактор Unreal Editor за замовчуванням. Цей редактор є основним редактором, який допомагає користувачеві переглядати та працює з іншими підплатформами та редакторами.

Досвідчені програмісти використовують мову C++ для створення власних скриптів для запуску в ігровій платформі. Розробники-початківці можуть використовувати готові блоки коду.

Серед усіх вище перелічених переваг дане середовище має одну досить важливу, а саме - доступність. Завдяки цьому, а також завдяки високій оптимізації творці ігор мають можливість швидко моделювати та створювати оптимізовані додатки. Одним з недоліків даного середовища є той факт, що Unreal Engine підтримує не так багато платформ.

1.2. Актуальність розроблення продукту

Створення ігор на даний час є одною з найприбутковіших галузей в сфері інформаційних технологій і статки, пов’язані з ігровим бізнесом, продовжують приваблювати до нього розробників. Однак на практиці розробка ігор є однією з найскладніших областей розробки програмного забезпечення. Варто пригадати

уроки тригонометрії, геометрії та фізики, які, як ми думали, ніколи не доведеться використати. А тут, ваша гра повинна поєднувати звук, відео та історію таким чином, щоб користувач захотів грати в неї все більше і більше. Щоб полегшити роботу, доступні фреймворки для розробки ігор не лише на C і C++, але навіть на C# або JavaScript.

Найбільшою проблемою перед лицем якої стоять усі розробники є можливість програмування відразу на всі платформи, а також простота реалізації певних операцій. Саме тому, серед усіх можливих середовищ для реалізації даного проекту обрано MonoGame.

Дане середовище дає можливість створювати застосунки майже на всі доступні платформи, а також використовує зрозумілу для усіх платформ мову програмування C#. Попередні версії MonoGame дозволяли розробникам ігор створювати 2D-ігри на основі спрайтів, але команда MonoGame припинила це в середині червня 2012 року. Починаючи з 2012 року, MonoGame почала дуже добре працювати з 3D (рис.1.1).

Отже, які складові MonoGame:

- ✓ Використовуючи як SharpDX, так і DirectX, MonoGame ідеально підходить для платформ Microsoft Windows.
- ✓ MonoGame використовує Microsoft XNA 4 API. Фреймворк XNA – це в основному набір інструментів розробки програмного забезпечення, розроблених Microsoft для створення комплексних ігор. Таким чином, XNA має величезну кількість інформації, інструментів і ресурсів для будь-якого типу розробки.

✓ MonoGame отримує свої графічні можливості від OpenGL. Точніше, OpenGL допоміг MonoGame з ефектами затінення та візуалізації.

Рис. 1.1. Складові MonoGame

Monogame підтримує наступні платформи:

- iOS
- macOS
- Android
- Linux
- Windows Phone 8, Windows 10
- PlayStation 4, PlayStation Vita
- Xbox One
- Nintendo Switch
- tvOS

Як ми вже згадували, фреймворк абсолютно безкоштовний. Він використовує публічну ліцензію Microsoft (Ms-PL) і взаємну ліцензію Microsoft (Ms-RL).

Unity 3D	**Project** Anarchy	**GameSalad**
Переваги: 1. вигідна ліцензійна політика; 2. легкість у використанні; 3. сумісність з будь-якою платформою 4. відмінне ком'юніті; 5. популярний серед розробників	*Переваги:* 1. якщо ви плануєте розробляти гри на платформах iOS, Android і Tizen, то ліцензія - безкоштовна; 2. потужні інструменти для пошуку і усунення багів; 3. сильне ком'юніті; 4. видавець надає чітку, зрозумілу документацію і зразки; 5. Fmod для аудіо-супроводу; 6. потужний Havok AI.	*Переваги:* 1. безкоштовна ліцензія (гроші з вас попросять тільки за PRO-версію); 2. активне ком'юніті; 3. відмінний рушій для швидкого створення прототипу; 4. сумісність з популярними мобільними платформами такими, як Cocona і Moai.
Недоліки: 1. обмежений набір інструментів (вам, швидше за все, доведеться розробити деякі з них самим); 2. процес виготовлення гри забирає багато часу.	*Недоліки:* 1. відсутня можливість розробляти гру на Mac і Linux; 2. немає вступного керівництва для початківців розробників;	*Недоліки:* 1. обмежений набір інструментів розробки; 2. немає доступу до більшості можливостей платформи iOS.

Основними конкурентами MonoGame є: Unity 3D, Project Anarchy, GameSalad

1.3. Переваги використання обраного фреймворку

Існує кілька чудових рис, завдяки яким Monogame вирізняється серед інших ігрових платформ, навіть таких, як Unity. Ось лише кілька сильних сторін:

1. Налаштування
2. Створено для програмістів
3. Відкрите джерело
4. Використовує Microsoft XNA
5. Ідеально підходить для кросплатформних проектів
6. Хороші графічні можливості

MonoGame — це реалізація Microsoft XNA 4 Framework з відкритим вихідним кодом. Метою є дозволити розробникам XNA на Xbox 360, Windows і Windows Phone переносити свої ігри на iOS, Android, Mac OS X, Linux і Windows 8 Metro. Платформи PlayStation Mobile, Raspberry PI і PlayStation 4 наразі працюють. DigitalRune Engine підтримує XNA і MonoGame кількома способами:

- ✓ DigitalRune Mathematics містить допоміжні методи для перетворення типів з/у типи XNA (наприклад, DigitalRune Vector3F в XNA Vector3).
- ✓ Кілька бібліотек DigitalRune надають спеціальні DLL для конвеєра вмісту XNA.
- ✓ Деякі бібліотеки DigitalRune вимагають XNA або MonoGame:
- ✓ DigitalRune Graphics вимагає XNA/MonoGame для відтворення 2D та 3D-графіки.
- ✓ Інтерфейс користувача DigitalRune Game вимагає XNA/MonoGame для зчитування пристроїв введення та візуалізації 2D-графіки.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Mono в Android

2.1.1 Архітектура

Програми Xamarin. Android запускаються в середовищі виконання Mono. Це середовище виконання працює поруч із віртуальною машиною Android Runtime (ART). Обидва середовища виконання працюють над ядром Linux і відкривають різні API для коду користувача, що дозволяє розробникам отримувати доступ до базової системи. Моно-runtime написана мовою C.

Ви можете використовувати системи System, System.IO, System.Net та інші бібліотеки класів .NET для доступу до базових засобів операційної системи Linux.

На Android більшість системних засобів, таких як Аудіо, Графіка, OpenGL та Телефонія, недоступні безпосередньо для власних програм, вони відкриваються лише через API Java Runtime Java, що знаходяться в одному з Java. * Просторів імен або Android. * Просторах імен. Архітектура приблизно така:

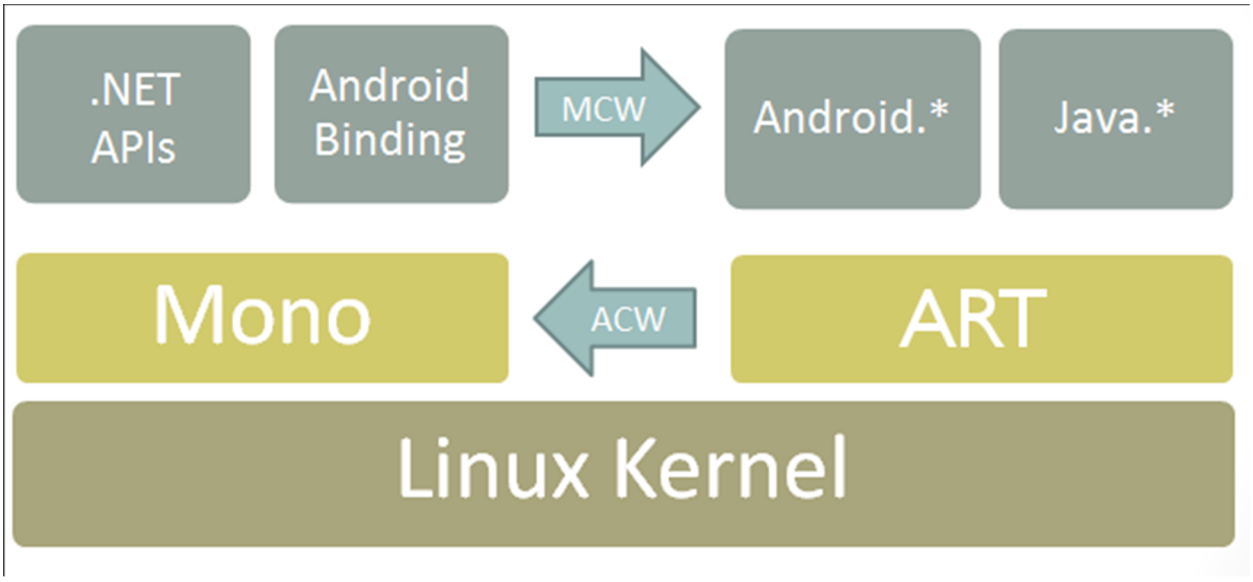


Рис. 2.1. Діаграма Mono та ART. NET / Java + прив'язки

Розробники Xamarin в Android володіють доступом до розмаїття функцій в операційній системі. Вхід відбувається за допомогою API .NET, або застосовуючи класи, відкриті у просторі імен Android, що створює своєрідний міст до API API Java.

2.1.2 Пакети програм

Пакети програм для Android - це контейнери ZIP з розширенням apk. Пакети програм Xamarin.Android володіють тою ж структурою та макетом, що й типові пакети Android, з наступними доповненнями:

- Збірки додатку (що містить IL) будуть збережені без стиснення в збірках папки. Під час запуску у релізі будується.apk і збірки вивантажуються з пам'яті. А це пришвидшує роботу додатка, оскільки не затрачається час на вивільнення збірки перед виконанням.
- На інформацію про розташування збірки, таку як Assembly.Location та Assembly.CodeBase, не можна покладатися у версії об’єктів. Адже вони не мають статичного розташування.
- Рідні бібліотеки. Додаток Xamarin.Android мусить включати вбудовані бібліотеки для цільової архітектури Android. Програми Xamarin.Android не працюватимуть на платформі без доступу до відповідних бібліотек виконання.

2.2. MonoGame

MonoGame – це реалізація Microsoft XNA 4 Framework з відкритим кодом. Це дозволяє розробникам XNA на Xbox 360, Windows і Windows Phone розгортати свої ігри на iOS, Android та багатьох інших платформах. Він також дуже добре настроюється як система з відкритим вихідним кодом, що дозволяє розробникам пограти з його інструментами відповідно до потреб (рис.2.2).

Рис. 2.2. MonoGame

Розробник: MonoGames

Платформи: iOS, Android, Windows, Linux, Windows Phone, PlayStation, Xbox тощо.

Клієнти: Tribute Games, Kongregate, Supergiant Games, 17-BIT тощо

Ігри: Bastion, Fez, Wizorb, Infinite Flight тощо

MonoGame є підтримуваний практично усіма консольними платформами. Як мова середовища MonoGame використовується C#. MonoGame використовує проміжний код для компіляції, а для користування CIL-кодом передбачено віртуальне середовище, більше відоме як .NET в Windows, та MonoGame в Linux, Android, iOS та ігрових консолях.

Здійснення проекту цілком і повністю може бути і в Windows, а процес компіляції та старт вихідного файлу може бути майже на всіх платформах(рис.2.3).

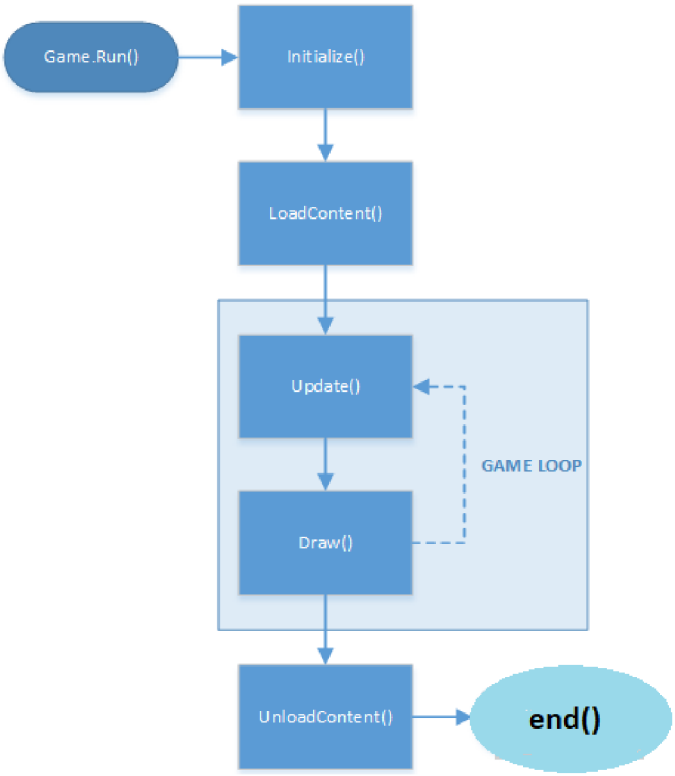


Рис. 2.3. Реалізація вихідного коду

Однією з суттєвих переваг даного середовища є його доступність та той факт, що код пишеться на C# та .NET. Актуальність даних технологій дозволяє рухатися в в такому ж швидкому темпі, що й більшість нових засобів та дає можливість використовувати їх у різноманітних проектах. До переваг, також, належиться віднести той факт, що дана мова є швидкою в написанні та обробці. Відповідно затрати часу в написанні коду будуть мінімальними.

Але ж, звичайно, крос-платформність є основною перевагою MonoGame. Для зміни платформи, достатньо змінити вид бібліотеки, що під'єднана до актуального проекту, на інші бібліотеки інших платформ. MonoGame здатне цілком повторювати архітектуру XNA. Для розробника важливо підключити відповідні директиви до самого проекту, в випадку, якщо переносити його з XNA на MonoGame. Директиви різних проектів, мають різні способи підключення, що залежить від платформи. Для контролю вибору платформи на якій відбуватиметься компіляція та які директиви необхідно підключити, використовуються правила BUILD.

Нижче долучено частину коду класу GraphicsDevice, в якому за допомогою препроцесорних директив доєднуються графічні бібліотеки для кожної платформи, що підтримується :

```
#if OPENGL
        #if MONOMAC
                using MonoMac.OpenGL;
        #elif WINDOWS || LINUX
                using OpenTK.Graphics.OpenGL;
        #elif GLES
                using OpenTK.Graphics.ES20;
#endif

#elif DIRECTX
        using SharpDX;
        using SharpDX.Direct3D;
#if WINDOWS_PHONE
        using SharpDX.Direct3D11;
```

MonoGame застосовує бібліотеку SharpDX, з відкритим кодом, а це дає доступ до DirectX. В такий спосіб ми маємо повну підтримку на Windows 8. Також застосовуються бібліотеки OpenTK та LidGren. OpenTK – це розширена низькорівнева бібліотека C#, яка спрощує роботу з OpenGL, OpenCL та OpenAL. OpenTK можна використовувати для ігор, наукових програм або інших проектів, що потребують тривимірної графіки, аудіо або обчислювальної функціональності. Lidgren Network — це мережева бібліотека для .NET Framework, яка використовує один UDP-сокет для доставки простого API для підключення клієнта до сервера, читання та надсилання повідомлень. Система керування контентом ідентична до системи XNA 4.

Суттєвою перевагою є існування спільнот, які в режимі нон-стоп створюють нові можливості для MonoGame.

Процес створення гри розпочинається з WPF проекту, тоді з легкістю переноситься в MonoGame.

- WPF проект компілюється швидко та є можливість його запуску без емулятора Android4;
- легке завантаження зображень;
- корекція помилок мінімізує затрати часу.

Для імпорту WPF проекту в MonoGame достатньо скопіювати C# файл та додати зображення в проект.

Команда MonoGame створила власну мову для створення шейдерних ефектів — MGFX.

Властивості MGFX:

- ✓ Володіє компільованим та оптимізованим бінарним форматом для застосування під час роботи MonoGame
- ✓ Підтримує різноманітні шейдерні мови, тобто є кросплатформна
- ✓ Їй властиве подальше розширення та вдосконалення
- ✓ Програми шейдерів володіють такою ж структурою, як Microsoft FX файли
- ✓ Для простоти редагування містять текстовий формат

Для роботи з MGFX компанія MonoGame створила утиліту 2MGFX, що дає можливість перетворення програми MGFX шейдерів. Так само як в XNA, MonoGame містить вбудовані шейдерні ефекти, що є підтримувані на таких платформах, як: SkinnedEffect, AlphaTestEffect, DualTextureEffect, BasicEffect, EnvironmentMapEffect.

2.3. Алгоритм Еллера в системі генерації лабіринту

Алгоритм Еллера є досить популярним алгоритмом для генерування «ідеальних» (між двома точками тільки один шлях) лабіринтів, оскільки він є одним з найшвидших і генерує «цікаві» лабіринти. Вони «цікаві» тим, що цей алгоритм не базується на пошуку остовного дерева в графі, в результаті чого генерує менше фрактальних структур. У тому числі однією з переваг алгоритму

є те, що він дозволяє генерувати лабіринти нескінченного розміру за лінійний час.

Алгоритм заснований на тому, що ми об’єднуємо клітинки лабіринту в різні набори, а на останньому кроці алгоритму об’єднуємо їх в один загальний набір. Комірки знаходяться в одному наборі - якщо одна клітинка може перейти до іншої. Спочатку всі стінки лабіринту піднімаються, так що всі клітини знаходяться в різних наборах.

Зберігати карту лабіринту можна, наприклад, у двох двовимірних масивах: для вертикальних і горизонтальних стінок, відповідно.

В порівнянні з іншими генераторами, даний алгоритм є одним з найшвидших. Особливо варте уваги те, що він не потребує багато. Нам потрібно зберегти у пам’яті лише останній рядок, щоб генерувати продовження лабіринту необмежену кількість разів.

Покроковий приклад генерації лабіринту розглянемо нижче:

1. Створимо перший рядок.

```
 __ ___ __ ___ __ ___
|                    |
```

2. Присвоїмо всім клітинкам унікальну множину

```
 __ ___ __ ___ __ ___
| 1   2   3   4   5   6 |
```

3. Рухаючись зліва направо, випадковим чином вирішимо, чи додавати стінки між клітинками(Оскільки на даному кроці всі клітинки належать до різних множин, наш вибір ніщо не обмежує)

```
 __ ___ __ ___ __ ___
|(1 | 2)  3   4   5   6 | - додаємо праву стінку після першої клітинки
 __ ___ __ ___ __ ___
| 1 |(2   3)  4   5   6 | - не додаємо стінку після другої
 __ ___ __ ___ __ ___
| 1 | 2   2   4   5   6 | - об'єднуємо множини, до яких належать 2 та 3 клітинки
 __ ___ __ ___ __ ___
| 1 | 2  (2   4)  5   6 |
 __ ___ __ ___ __ ___
| 1 | 2   2   2   5   6 |
 __ ___ __ ___ __ ___
| 1 | 2   2  (2 | 5)  6 |
 __ ___ __ ___ __ ___
| 1 | 2   2   2 |(5 | 6)|
 __ ___ __ ___ __ ___
| 1 | 2   2   2 | 5 | 6 |
```

4. Додамо нижні стінки

```
 __ ___ __ ___ __ ___
|(1)| 2   2   2 | 5 | 6 | - не додаємо стінку, це - остання клітинка у множині з
індексом 1, що не має нижньої стінки.
 __ ___ __ ___ __ ___
| 1 |_2_  2   2 | 5 | 6 |
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
```

5. Вітаю! Ми отримали перший завершений рядок нашого лабіринту. Додамо наступний. Для цього виведемо поточний іще раз:

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1 |_2_  2  _2_| 5 | 6 |
```

5. 1. 1. Видалимо всі праві стінки

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1  _2_  2  _2_  5   6 |
```

5. 1. 2. Усі клітинки, що містять нижні стінки, видалимо з їх множин

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1  _ _  2  _ _  5   6 |
```

5. 1. 3. Видалимо нижні стінки

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1       2       5   6 |
```

Повернемося до пункту 2. Присвоїмо клітинкам, що не належать до множин, унікальні множини

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1   3   2   4   5   6 |
```

3. Створимо праві стінки

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1   1 | 2   2 | 5   5 |
```

5. 1. 3. Видалимо клітинки з нижніми стінками

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1  _1_|_2_  2 | 5   5 |
| 1  _ _ _ _  2  5   5 |
```

5. 1. 4. Видалимо нижні стінки

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1  _1_|_2_  2 | 5   5 |
| 1           2  5   5 |
```

Повернемося до пункту 2. Присвоїмо порожні клітинки множинам

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1  _1_|_2_  2 | 5   5 |
| 1   3   4   2  5   5 |
```

3. Створимо праві стінки

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1  _1_|_2_  2 | 5   5 |
| 1   1 | 4   4  5   5 |
```

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1  _1_|_2_  2 | 5   5 |
| 1   1 | 4  (4   5)  5 |
```

Зверніть увагу: ми об'єднуємо множини клітинок, тобто остання клітинка також стане належати до множини 4 після даного кроку.

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1  _1_|_2_  2 | 5   5 |
| 1   1 | 4   4   4   4 |
```

Тут обов'язково поставити стінку після 5 клітинки — наступна з тієї самої множини

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1  _1_|_2_  2 | 5   5 |
| 1   1 | 4   4  (4 | 4)|
```

4. Додамо нижні стінки

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1  _1_|_2_  2 | 5   5 |
| 1  _1_| 4  _4_ _4_| 4 |
```

5. Більше не додаватимемо рядків

5. 2. 1.

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1  _1_|_2_  2 | 5   5 |
|_1_ _1_|_4_ _4_ _4_|_4_|
```

5. 2. 2. Праву стінку після 2 клітинки потрібно видалити та об'єднати множини 2 та 3 клітинок(після завершення проходження по рядку усі клітинки мають належати одній множині).

```
 __ ___ __ ___ __ ___
| 1 |_2_  2  _2_| 5 | 6 |
| 1  _1_|_2_  2 | 5   5 |
|_1_ _1_ _1_ _1_ _1_|_1_|
```

І от, наш лабіринт нарешті завершено. Вхід та вихід можна зробити з будь-якої зовнішньої стінки лабіринту.

```
 __ ___ __ ___ __ ___
   |_ _     _ _|   |   |
|    _ _|_ _    |       |
|_ _ _ _ _ _ _ _ _ _|_ _
```

Зрештою ми отримали ідеальний лабіринт (рис. 2.4), в якому немає циклів (між двома осередками є лише один шлях) та ізольованих частин (комірки або груп осередків, які не пов'язані з іншими частинами лабіринту). Тепер ми можемо призначити будь-які два осередки відповідно «входом» та «виходом».

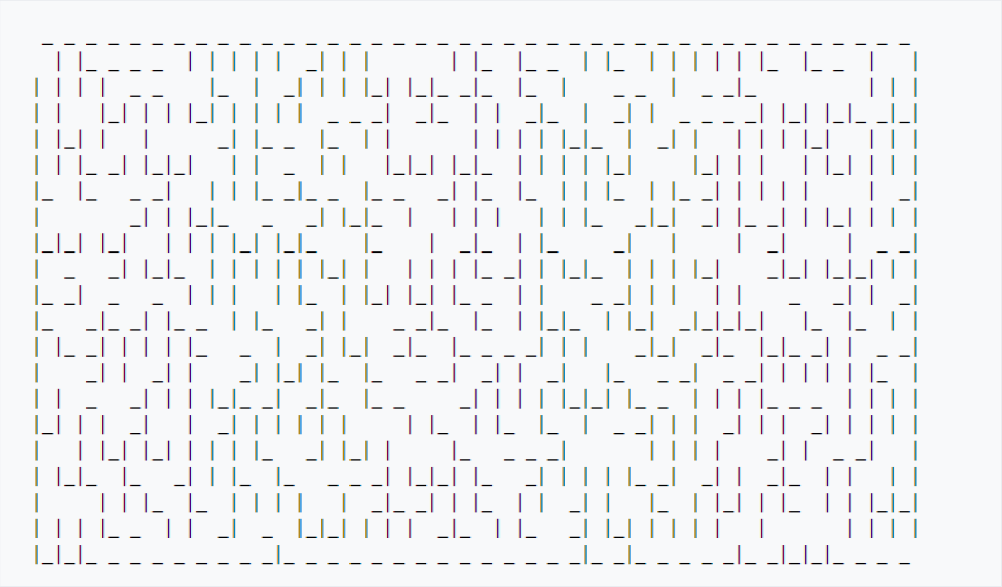


Рис. 2.4. Лабіринт

2.4. Collision

Намагаючись визначити, чи відбувається зіткнення між двома об’єктами, ми зазвичай не використовуємо дані про вершини самих об’єктів, оскільки ці об’єкти часто мають складні форми; це, у свою чергу, ускладнює виявлення зіткнення. З цієї причини поширеною практикою є використання більш простих фігур (які зазвичай мають гарне математичне визначення) для виявлення зіткнень, які ми накладаємо на вихідний об’єкт. Потім ми перевіряємо наявність зіткнень на основі цих простих фігур; це полегшує код і значно економить продуктивність. Декількома прикладами таких фігур зіткнення є кола, сфери, прямокутники та прямокутники; з ними набагато простіше працювати в порівнянні з довільними сітками з сотнями трикутників.

Хоча прості фігури дають нам простіші та ефективніші алгоритми виявлення зіткнень, вони мають загальний недолік, оскільки ці форми зазвичай не повністю оточують об’єкт. Ефект полягає в тому, що може бути виявлено зіткнення, яке насправді не зіткнулося з фактичним об'єктом; завжди слід пам’ятати, що ці форми є лише наближеннями до реальних форм.

2.4.1 Вирівнювання по осі обмеженого хітбокса

Однією з найпростіших форм виявлення зіткнення є між двома прямокутниками, які вирівняні по осі (рис. 2.5), тобто без обертання. Алгоритм працює, забезпечуючи відсутність проміжку між жодною з 4 сторін прямокутників. Щонайменший розрив означатиме, що зіткнення не існує.

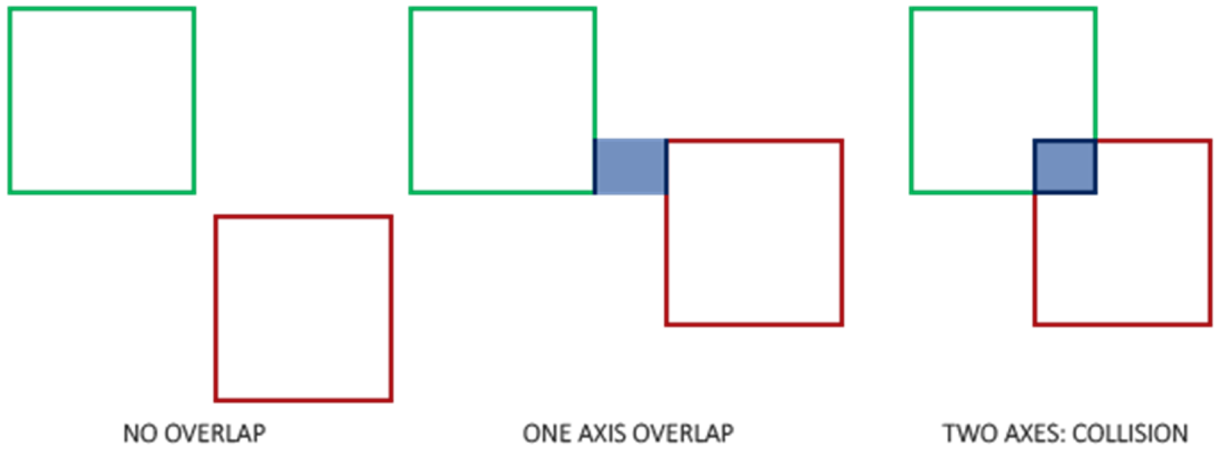


Рис. 2.5. Осі хітбокса

2.4.2 Зіткнення кругів

Інша проста форма для виявлення зіткнення - це зіткнення між двома колами. Цей алгоритм працює, беручи центральні точки двох кіл і забезпечуючи відстань між центральними точками меншою, ніж два радіуси разом.

2.5. Алгоритм пошуку шляхів

Алгоритм пошуку є одним з найкращих і популярних методів, що використовуються для пошуку шляхів. Неформально кажучи, алгоритми пошуку A*, на відміну від інших методів обходу, мають «мізки». Це означає, що це дійсно розумний алгоритм, який відрізняє його від інших звичайних алгоритмів.

Також варто згадати, що багато ігор і веб-карт використовують цей алгоритм, щоб дуже ефективно знаходити найкоротший шлях (апроксимація).

Якщо розглянути квадратну сітку з безліччю перешкод, де нам дано початкову та цільову клітинку. І ми хочемо якомога швидше досягти цільової

комірки (якщо можливо) із початкової комірки. Тут на допомогу приходить алгоритм пошуку A*. Алгоритм пошуку A* полягає в тому, що на кожному кроці він вибирає вузол відповідно до значення «f», яке є параметром, рівним сумі двох інших параметрів — «g» і «h». На кожному кроці він вибирає вузол/комірку з найнижчим «f», і обробляє цей вузол/комірку.

Евристичний метод не шукає шлях на всі боки відразу і не витрачає час на очевидно безперспективні шляхи, які віддаляють від мети, а не наближають до неї. Він допомагає це виправити. Він працює так само, як і алгоритм Дійкстри, але з однією зміною. При виборі наступної точки на розгляд першої вибирається та точка, яка ближче до початку шляху, а та, що ближче до кінця. Відстань до кінця розраховується приблизно, наприклад, просто як відстань між двома точками на карті. Мінус алгоритму в тому, що знайдений шлях не обов'язково буде найкоротшим. Але його знаходження потребує менше часу.

Алгоритм пошуку A* можна подати у вигляді псевдокоду(рис.2.6):

```
program a-star
    // Ініціалізація списку відомих вершин, список досліджених порожній
    // (f-значення початкової вершини відсутнє)
    openlist.enqueue(startknote, 0)
    // цей шлях буде пройдений доки:
    // - буде знайдено оптимальний розв'язок або
    // - встановлено, що розв'язків не існує
    repeat
        // Вилучити вершину з найменшим f-значенням
        currentNode := openlist.removeMin()
        // Досягнута кінцева вершина?
        if currentNode == endknote then
            return PathFound
        // Якщо кінцева вершина не досягнута: додати суміжні
        // до поточної вершини в список відомих
        expandNode(currentNode)
        // поточна вершина вже повністю досліджена
        closedlist.add(currentNode)
    until openlist.isEmpty()
    // список відомих порожній, розв'язків не існує
    return NoPathFound
end
```

```
// перевіряє суміжні вершини та додає до списку відомих якщо:
// - суміжні вершини зустрічаються вперше або
// - знайдений кращий шлях до цієї вершини
function expandNode(currentNode)
    foreach successor of currentNode
        // пропустити, якщо вершина знаходиться в списку досліджених
        if closedlist.contains(successor) then
            continue
        // обчислити значення g нового шляху:
        // значення g попередньої вершини + вартість щойно пройденого ребра
        tentative_g = g(currentNode) + c(currentNode, successor)
        // якщо суміжна вершина вже в списку відомих,
        // але знайдений шлях не кращий за вже відомий - пропустити
        if openlist.contains(successor) and tentative_g >= g[successor] then
            continue
        // встановити вказівник на попердню вершину та зберегти g
        successor.predecessor := currentNode
        g[successor] = tentative_g
        // оновити значення f вершини у списку відомих
        // або додати вершину до списку відомих
        f := tentative_g + h(successor)
        if openlist.contains(successor) then
            openlist.decreaseKey(successor, f)
        else
            openlist.enqueue(successor, f)
    end
end
```

Рис. 2.6. Код побудови алгоритму A*

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1. Розробка застосунку

Загальну структуру даного застосунку можна переглянути у діаграмі, зображеній на наступному рисунку (рис. 3.1).

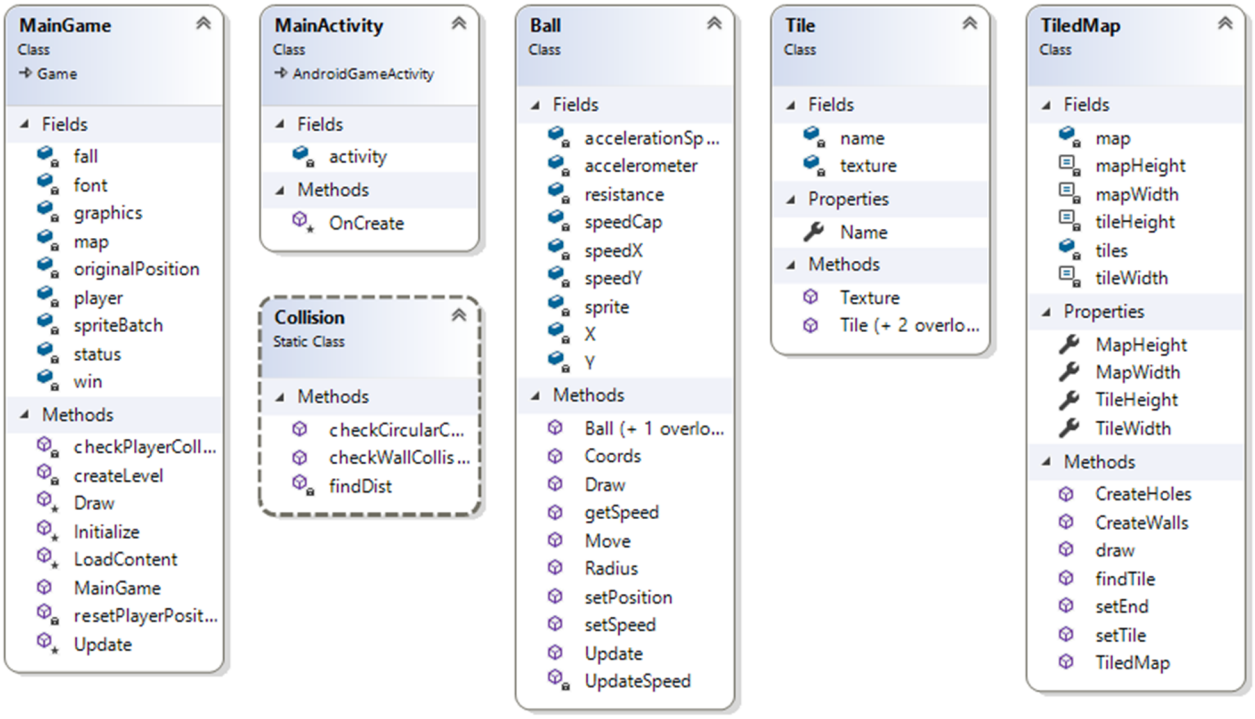


Рис. 3.1. Діаграма класів

3.1.1. Ініціалізація MonoGame завдяки XamarinForms

Для старту роботи з MonoGame необхідна перш за все ініціалізація. Реалізація активації (рис. 3.2) та Ініціалізація рушія (рис 3.3) відбувається в наступний спосіб:

```
[Activity(Label = "MonoGame"
    , MainLauncher = true
    , Icon = "@drawable/icon"
    , AlwaysRetainTaskState = true
    , LaunchMode = LaunchMode.SingleInstance
    , ScreenOrientation = ScreenOrientation.Landscape
    , ConfigurationChanges = ConfigChanges.Orientation |
    ConfigChanges.Keyboard |
    ConfigChanges.KeyboardHidden)]
1 reference
public class MainActivity : Microsoft.Xna.Framework.AndroidGameActivity
{
    MainActivity activity;
    0 references
    protected override void OnCreate(Bundle bundle)
    {
        activity = this;
        base.OnCreate (bundle);

        var g = new MainGame ();
        SetContentView((View)g.Services.GetService(typeof(View)));
        g.Run ();
    }
}
```

Рис. 3.2. Фрагмент коду

```
public MainGame()
{
    graphics = new GraphicsDeviceManager(this);
    graphics.IsFullScreen = true;
    graphics.SupportedOrientations = DisplayOrientation.LandscapeLeft;
    Content.RootDirectory = "Content";
}

0 references
protected override void Initialize()
{
    base.Initialize();
}
```

Рис. 3.3. Фрагмент коду

Для того, збереження часу та ресурсів пристрою ресурси необхідно завантажити перед початком роботи з ними. Для цього існує метод LoadContent(), який представлено на рисунку 3.4.

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    font = Content.Load<SpriteFont>("Font");
    win = Content.Load<SoundEffect>("victoire");
    fall = Content.Load<SoundEffect>("trou");

    map = new TiledMap(this.GraphicsDevice);
    player = new Ball(this.GraphicsDevice, 50, 50);

    status = 0;
    createLevel();
}
```

Рис. 3.4. Фрагмент коду

За оновлення графіки в кожному відтвореному кадрі відповідає метод Update, відповідно до інтерфейсу Game (рис 3.5).

```
protected override void Update(GameTime gameTime)
{
    base.Update(gameTime);
    if (status == 0)
    {

        TouchCollection touchCollection = TouchPanel.GetState();

        if (touchCollection.Count > 0)
        {
            status++;
            createLevel();
        }
    }
    player.Update();
    checkPlayerCollision();
    player.Move();
}
```

Рис. 3.5. Фрагмент коду

На завершення представлено метод відмалювання кожного кадру (рис. 3.6).

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();

    map.draw(spriteBatch);
    player.Draw(spriteBatch);

    if(status == 0)
    {
        spriteBatch.DrawString(font, "Press anywhere to start", new Vector2(800, 600), Color.Black);
    }
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Рис. 3.6. Фрагмент коду

3.1.2. Реалізація та попередньо створення юніта гравця

За допомогою акселерометра, що міститься в в Андроїд гравець керуватиме кулькою. Зображення опису об’єкта кульки подано на рисунку 3.7.

```
public Ball(GraphicsDevice graphicsDevice)
{
    X = 0;
    Y = 0;

    speedX = 0.5f;
    speedY = 0.5f;

    accelerationSpeed = 1.0f;

    speedCap = 5f;
    resistance = 0.3f;


    if (accelerometer == null)
    {
        accelerometer = new Accelerometer();

        accelerometer.Start();

    }
    using (var stream = TitleContainer.OpenStream("Content/balle.png"))
    {
        sprite = Texture2D.FromStream(graphicsDevice, stream);
    }

}
```

Рис. 3.7. Фрагмент коду опису об’єкта кульки

З вище наведеного коду можемо побачити, що рисунок кульки перетворюється в спрайт. Дана дія для того аби швидше та коректніше відображати певний об’єкт в кожному кадрі. Для вираження швидкості перекочування кульки, ми використовуємо параметри speedX та speedY. За прискорення швидкості кульки відповідає параметр accelerationSpeed.

Метод UpdateSpeed було реалізовано для зміни швидкості руху в залежності від нахилу акселерометра, про це описано на рисунку 3.8.

```
private void UpdateSpeed()
{

    float orientationX = accelerometer.CurrentValue.Acceleration.Y;
    float orientationY = accelerometer.CurrentValue.Acceleration.X;

    orientationX = (float)(Math.Round((double)orientationX, 1));
    orientationY = (float)(Math.Round((double)orientationY, 1));


    if (orientationX == 0)
    {
        if (speedX < -resistance || speedX > resistance)
        {
            speedX += speedX > 0 ? -resistance : resistance;
        }
        else {
            speedX = 0;
        }
    }
    else
    {
        speedX += orientationX * accelerationSpeed;
    }
```

```
    if (orientationY == 0)
    {
        if (speedY < -resistance || speedY > resistance)
        {
            speedY += speedY > 0 ? -resistance : resistance;
        }
        else {
            speedY = 0;
        }
    }
    else
    {
        speedY += orientationY * accelerationSpeed;
    }

    speedX = speedX > speedCap ? speedCap : speedX < -speedCap ? -speedCap : speedX;
    speedY = speedY > speedCap ? sp                              peedCap : speedY;
                                 (field) float Ball.accelerationSpeed

    X += X % 1920;
    Y += Y % 1080;
}
```

Рис. 3.8. Фрагмент коду

Для розширення можливостей гри у майбутньому, було створено мінорні методи, такі як: Move(), SetSpeed(), GetSpeed(), Cords(), Show() та інші.

3.1.3. Побудова фону карти та об’єкти лабіринту

Побудова основного ігрового поля карти здійснюється через створення класу, який наповнює поле об’єктами фону. На рисунку 3.9 описано конструктори об’єкта фону.

```
public Tile(GraphicsDevice graphicsDevice, string name)
{
    using (var stream = TitleContainer.OpenStream("Content/sol.jpg"))
    {
        texture = Texture2D.FromStream(graphicsDevice, stream);
    }
}

0 references
public Tile(GraphicsDevice graphicsDevice, Texture2D sprite)
{
    texture = sprite;
}

4 references
public Tile(GraphicsDevice graphicsDevice, string sprite, string tileName)
{
    this.name = tileName;
    using (var stream = TitleContainer.OpenStream("Content/" + sprite + ".jpg"))
    {
        texture = Texture2D.FromStream(graphicsDevice, stream);
    }
}
```

Рис. 3.9. Фрагмент коду конструктора об’єкта фону

Фон мапи з об’єктів типу Sprite використано для побудови об’єкту поля.

Конструктор об’єкта поля реалізуємо за допомогою коду, представленого на рисунку 3.10.

```
public TiledMap(GraphicsDevice graphicsDevice)
{
    map = new Tile[mapWidth, mapHeight];
    tiles = new Dictionary<string, Tile>();
    /* Creating the tiles */
    tiles.Add("mur", new Tile(graphicsDevice, "mur", "mur"));
    tiles.Add("sol", new Tile(graphicsDevice, "sol", "sol"));
    tiles.Add("sortie", new Tile(graphicsDevice, "sortie", "sortie"));
    tiles.Add("trou", new Tile(graphicsDevice, "trou", "trou"));

    /* Surrounding walls */
    for (int x = 0; x < MapWidth; x++)
    {
        map[x, 0] = tiles["mur"];
        map[x, MapHeight - 1] = tiles["mur"];
    }
    for (int y = 0; y < MapHeight; y++)
    {
        map[0, y] = tiles["mur"];
        map[MapWidth - 1, y] = tiles["mur"];
    }

    /* Floor */
    for (int x = 1; x < MapWidth - 1; x++)
    {
        for (int y = 1; y < MapHeight - 1; y++)
        {
            map[x, y] = tiles["sol"];
        }
    }
```

Рис. 3.10. Фрагмент коду

Методи відмалювання поля та присвоєння коміркам об’єктів типу Tile та вивантаження цих об’єктів на поле реалізуємо також у цьому класі. Все це відбувається в такий спосіб:

```
public void draw(SpriteBatch spriteBatch)
{
    for (int x = 0; x < MapWidth; x++)
    {
        for (int y = 0; y < MapHeight; y++)
        {
            spriteBatch.Draw(map[x, y].Texture(), new Vector2((float)(x * TileWidth), (float)(y * TileHeight)), Co
        }
    }
}

3 references
public Tile findTile(int x, int y)
{
    return map[x, y];
}

0 references
public bool setTile(Vector2 position, string name)
{
    if (position.X >= 0 && position.X < MapWidth && position.Y >= 0 && position.Y < MapHeight && tiles.ContainsKey
    {
        map[(int)position.X, (int)position.Y] = tiles[name];
        return true;
    }
    else
    {
        return false;
    }
}
```

Звичайно, у даному класі відтворено методи установки початку лабіринту, його завершення та створення стін (рис. 3.11).

```
public void CreateWalls(int[,] position)
{
    for (int x = 0; x < position.GetLength(0); x++)
    {
        map[position[x, 0], position[x, 1]] = tiles["mur"];
    }
}
3 references
public void setEnd(int[] position)
{
    map[position[0], position[1]] = tiles["sortie"];
}
```

Рис. 3.11. Фрагмент коду

Об’єкт Tile може приймати на себе відповідні значення спрайтів sol та mur. Це ми можемо спостерігати з опису коду. Ці об’єкти визначені та містять в собі зображення спрайтів. А саме, фон поля (рис. 3.12), та стіни лабіринту (рис. 3.13).

Рис. 3.12. Фон поля

Рис. 3.13. Стіна лабіринту

Все, що ми маємо можливість бачити на полі є об’єктами фону. Загалом, те, що ми можемо спостерігати за своєю суттю, виглядає лише зображенням без жодного функціоналу.

3.1.4. Побудова мапи колізій

Для того, щоб побудувати колізії стін, використано метод вирівнювання по осі обмеженої коробки (рис. 3.14).

```
static public bool checkWallCollision(Vector2 wallCoords, int spriteSize, Vector2 ballCoords, Vector2 ballPosition, int
{
    //TODO: Fix the collision detection. Currently not working
    /* Checking left collision */
    if (wallCoords.X + 1 == ballCoords.X)
    {
        if (ballPosition.X - ballRadius <= 0)
        {
            return true;
        }
    }
    /* Checking right collision */
    else if (wallCoords.X - 1 == ballCoords.X)
    {
        if (ballPosition.X + ballRadius >= spriteSize)
        {
            return true;
        }
    }
    /* Checking top collision */
    else if (wallCoords.Y + 1 == ballCoords.Y)
    {
        if(ballPosition.Y - ballRadius <= 0)
        {
            return true;
        }
    }
    /* Checking bot collision */
    else if (wallCoords.Y - 1 == ballCoords.Y)
    {
        if (ballPosition.Y + ballRadius >= spriteSize)
        {
```

Рис. 3.14. Метод вирівнювання по осі обмеженої коробки

При кожному оновленні кадру буде використовуватись цей метод. Тобто відбуватиметься порівняння координат стін та координат кулі, враховуючи радіус першої та розмір другої.

Подібний метод реалізовано для кулі, за допомогою методу визначення зіткнення кругів (рис. 3.15).

```
static public bool checkCircularCollision(Vector2 circleCenter, int circleRadius,Vector2 ballCoords, Vector2 ballPosition)
{
    int ballX = (int)(ballCoords.X + ballPosition.X);
    int ballY = (int)(ballCoords.Y + ballPosition.Y);
    int dist = (int)findDist(circleCenter, new Vector2(ballX, ballY));
    return dist <= circleRadius;
}
```

Рис. 3.15. Метод визначення зіткнення кругів

Обидва методи використовують функцію визначення відстані між об’єктами.

У вхідних параметрах розглядаються об’єкти «від» та «до», тоді відбувається процес порівняння з урахуванням розміру стін та радіусу кулі (рис. 3.16).

```
static private double findDist(Vector2 from, Vector2 to)
{
    double a2 = Math.Pow(from.X - to.X, 2);
    double b2 = Math.Pow(from.Y - to.Y,2);
    double dist = Math.Sqrt(a2 + b2);
    return dist;
}
```

Рис. 3.16. Порівняння з урахуванням розміру стін та радіусу кулі

3.2. Реалізація алгоритму побудови лабіринту

Реалізацію генерації лабіринту описано у коді на рисунку 3.17.

```
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 };
        }
    }

    for (int x = 0; x < cells.GetLength(0); x++)
    {
        cells[x, Height - 1].WallLeft = false;
    }

    for (int y = 0; y < cells.GetLength(1); y++)
    {
        cells[Width - 1, y].WallBottom = false;
    }

    RemoveWallsWithBacktracker(cells);

    Maze maze = new Maze();

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

    return maze;
}
```

Рис. 3.17. Генерації лабіринту

Використовуючи наступні функції, RemoveWallsWithBacktracker(), RemoveWall(), PlaceMazeExit(), відтворено згенерований лабіринт.

Ці функції зображено на рисунках 3.18 – 3.20.

```
private void RemoveWallsWithBacktracker(MazeGeneratorCell[,] maze)
{
    MazeGeneratorCell current = maze[0, 0];
    current.Visited = true;
    current.DistanceFromStart = 0;

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

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

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

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

            chosen.Visited = true;
            stack.Push(chosen);
            chosen.DistanceFromStart = current.DistanceFromStart + 1;
            current = chosen;
        }
```

Рис. 3.18. RemoveWallsWithBacktracker

```
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;
    }
}
```

Рис. 3.19. RemoveWall

```
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;

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

Рис. 3.20. PlaceMazeExit

Надалі даний метод (GenerateMaze) використовується у відтворені лабіринту безпосередньо в методі createLevel основного екрану гри. Продовжимо розгляд написання лабіринту за допомогою алгоритму та вручну (рис.3.21).

```
case 1:
    {
        map = new TiledMap(this.GraphicsDevice);
        int[,] walls = { { 1, 2 }, { 1, 4 }, { 1, 5 }, { 1, 6 }, { 1, 7 }, { 1, 8 }, { 1, 11 },
            { 2, 2 }, { 2, 9 }, { 2, 11 }, { 2, 13 }, { 2, 14 }, { 2, 15 }, { 2, 16 }, { 2, 17 },
            { 2, 18 }, { 2, 19 }, { 3, 2 }, { 3, 4 }, { 3, 5 }, { 3, 6 }, { 3, 7 }, { 3, 8 }, { 3, 9 },
            { 3, 11 }, { 3, 13 }, { 3, 19 }, { 4, 2 }, { 4, 11 }, { 4, 13 }, { 4, 19 }, { 4, 15 },
            { 4, 16 }, { 4, 17 }, { 5, 2 }, { 5, 3 }, { 5, 4 }, { 5, 5 }, { 5, 6 }, { 5, 7 }, { 5, 8 },
            { 5, 9 }, { 5, 11 }, { 5, 13 }, { 5, 15 }, { 5, 17 }, { 5, 19 }, { 6, 19 }, { 7, 1 },
            { 7, 2 }, { 7, 3 }, { 7, 4 }, { 7, 5 }, { 7, 6 }, { 7, 7 }, { 7, 8 }, { 7, 10 }, { 7, 11 },
            { 7, 12 }, { 7, 13 }, { 7, 14 }, { 7, 15 }, { 7, 17 }, { 7, 18 }, { 7, 19 }, { 8, 8 },
            { 8, 11 }, { 8, 15 }, { 8, 17 }, { 9, 2 }, { 9, 3 }, { 9, 4 }, { 9, 5 }, { 9, 6 }, { 9, 8 },
            { 9, 10 }, { 9, 11 }, { 9, 12 }, { 9, 13 }, { 9, 15 }, { 9, 17 }, { 9, 19 }, { 10, 6 },
            { 10, 8 }, { 10, 15 }, { 10, 17 }, { 10, 19 }, { 11, 2 }, { 11, 3 }, { 11, 4 }, { 11, 5 },
            { 11, 6 }, { 11, 8 }, { 11, 10 }, { 11, 11 }, { 11, 12 }, { 11, 13 }, { 11, 15 }, { 11, 17 },
            { 11, 19 }, { 12, 6 }, { 12, 8 }, { 12, 10 }, { 12, 13 }, { 12, 15 }, { 12, 17 }, { 12, 19 },
            { 13, 1 }, { 13, 3 }, { 13, 4 }, { 13, 5 }, { 13, 6 }, { 13, 10 }, { 13, 11 }, { 13, 13 },
            { 13, 14 }, { 13, 15 }, { 13, 17 }, { 13, 19 }, { 14, 1 }, { 14, 8 }, { 14, 17 }, { 14, 19 },
            { 15, 1 }, { 15, 3 }, { 15, 4 }, { 15, 5 }, { 15, 6 }, { 15, 7 }, { 15, 8 }, { 15, 9 },
            { 15, 10 }, { 15, 11 }, { 15, 12 }, { 15, 13 }, { 15, 14 }, { 15, 15 }, { 15, 16 }, { 15, 17 }
        int[,] holes = { };
        int[] end = { 20, 20 };
        int[] spawn = { 1,1 };
        map.CreateWalls(walls);
        map.CreateHoles(holes);
        map.setEnd(end);
        originalPosition = new Vector2((spawn[0] * map.TileWidth) + (map.TileWidth / 2) - player.Radius(),
        player.setPosition(originalPosition);
        break;
```

Рис. 3.21. Створення лабіринту вручну

```
case 2:
    {
        int[,] holes = { };
        int[,] walls = new GenerateMaze();
        map = new TiledMap(this.GraphicsDevice);
        int[] spawn = { 1, 1 };
        int[] end = { 23, 12 };
        map.CreateWalls(walls);
        map.CreateHoles(holes);
        map.setEnd(end);
        originalPosition = new Vector2(spawn[0] * map.TileWidth, spawn[1] * map.TileHeight);
        player.setPosition(originalPosition);
        break;
```

Рис 3.21. Створення лабіринту за допомогою алгоритму

Перше, що помічаємо – це простота створення нового лабіринту за допомогою алгоритму, який економить затрачений час на написання координат вручну.

3.3 Реалізація алгоритму A*

Для нашого ігрового продукту було використано типову бібліотеку A* pathfinder, яку було підлаштовано безпосередньо під даний проект.

Оголошення комірок лабіринту зображено на рисунку 3.22.

```
public class PathNode
{
    0 references
    public Point Position { get; set; }
    1 reference
    public int PathLengthFromStart { get; set; }
    0 references
    public PathNode CameFrom { get; set; }
    1 reference
    public int HeuristicEstimatePathLength { get; set; }
    0 references
    public int EstimateFullPathLength
    {
        get
        {
            return this.PathLengthFromStart + this.HeuristicEstimatePathLength;
        }
    }
}
```

Рис. 3.22. Оголошення комірок лабіринту

На рисунку 3.23 описано основний метод вирахування.

```
public static List<Point> FindPath(int[,] field, Point start, Point goal)
{
    var closedSet = new Collection<PathNode>();
    var openSet = new Collection<PathNode>();
    PathNode startNode = new PathNode()
    {
        Position = start,
        CameFrom = null,
        PathLengthFromStart = 0,
        HeuristicEstimatePathLength = GetHeuristicPathLength(start, goal)
    };
    openSet.Add(startNode);
    while (openSet.Count > 0)
    {
        var currentNode = openSet.OrderBy(node =>
          node.EstimateFullPathLength).First();
        if (currentNode.Position == goal)
            return GetPathForNode(currentNode);
        openSet.Remove(currentNode);
        closedSet.Add(currentNode);
        foreach (var neighbourNode in GetNeighbours(currentNode, goal, field))
        {
            if (closedSet.Count(node => node.Position == neighbourNode.Position) > 0)
                continue;
            var openNode = openSet.FirstOrDefault(node =>
              node.Position == neighbourNode.Position);
            if (openNode == null)
                openSet.Add(neighbourNode);
            else
              if (openNode.PathLengthFromStart > neighbourNode.PathLengthFromStart)
            {
                openNode.CameFrom = currentNode;
                openNode.PathLengthFromStart = neighbourNode.PathLengthFromStart;
            }
        }
    }
    return null;
```

Рис. 3.23. Метод вирахування

Код, що використовується для отримання списку сусідніх точок, описаний на рисунку 3.24.

```
private static Collection<PathNode> GetNeighbours(PathNode pathNode,
goal, int[,] field)
{
    var result = new Collection<PathNode>();

    Point[] neighbourPoints = new Point[4];
    neighbourPoints[0] = new Point(pathNode.Position.X + 1, pathNode.Position.Y);
    neighbourPoints[1] = new Point(pathNode.Position.X - 1, pathNode.Position.Y);
    neighbourPoints[2] = new Point(pathNode.Position.X, pathNode.Position.Y + 1);
    neighbourPoints[3] = new Point(pathNode.Position.X, pathNode.Position.Y - 1);

    foreach (var point in neighbourPoints)
    {
        if (point.X < 0 || point.X >= field.GetLength(0))
            continue;
        if (point.Y < 0 || point.Y >= field.GetLength(1))
            continue;
        if ((field[point.X, point.Y] != 0) && (field[point.X, point.Y] != 1))
            continue;
        var neighbourNode = new PathNode()
        {
            Position = point,
            CameFrom = pathNode,
            PathLengthFromStart = pathNode.PathLengthFromStart +
            GetDistanceBetweenNeighbours(),
            HeuristicEstimatePathLength = GetHeuristicPathLength(point, goal)
        };
        result.Add(neighbourNode);
    }
    return result;
}
```

Рис. 3.24. Отримання списку сусідніх точок

На завершення описано метод повернення маршруту (рис. 3.25.).

```
private static List<Point> GetPathForNode(PathNode pathNode)
{
    var result = new List<Point>();
    var currentNode = pathNode;
    while (currentNode != null)
    {
        result.Add(currentNode.Position);
        currentNode = currentNode.CameFrom;
    }
    result.Reverse();
    return result;
}
```

Рис. 3.25. Метод повернення маршруту

3.4. Тестування проекту

Вихідний варіант дизайну застосунку:

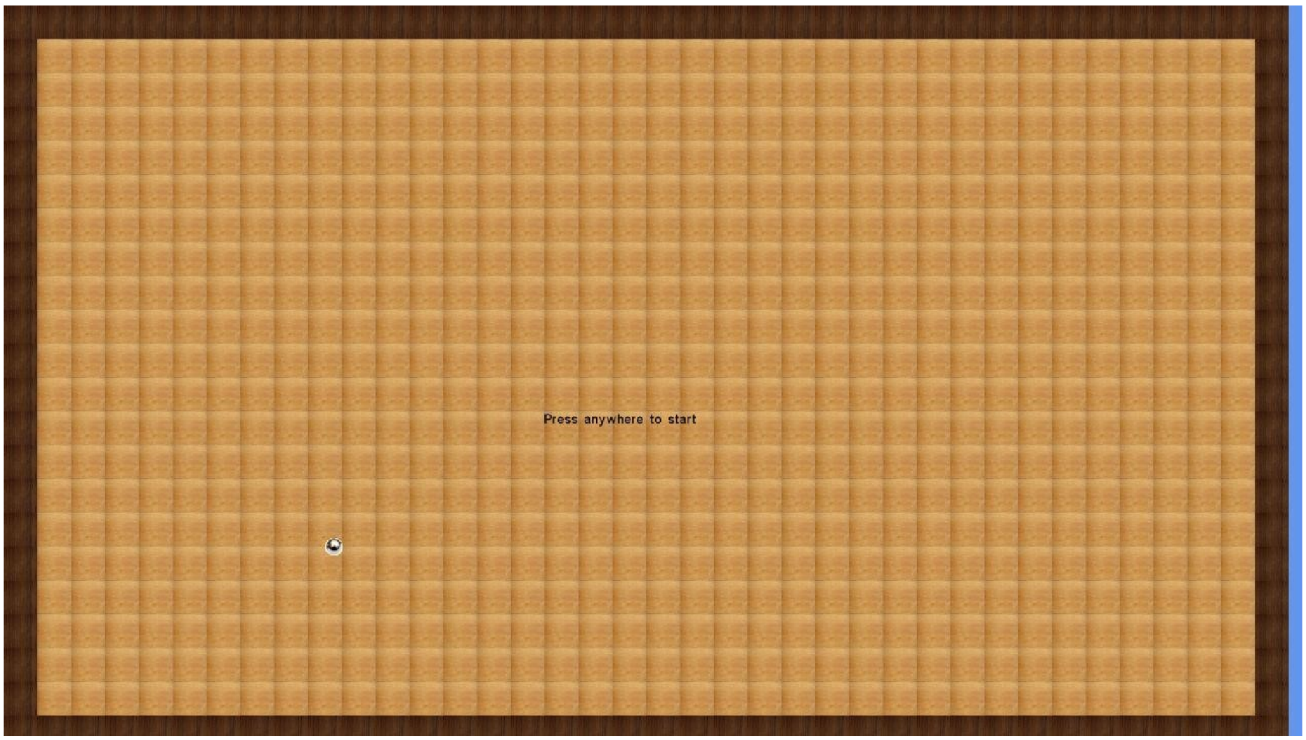


Рис 3.26. Головна сторінка застосунку

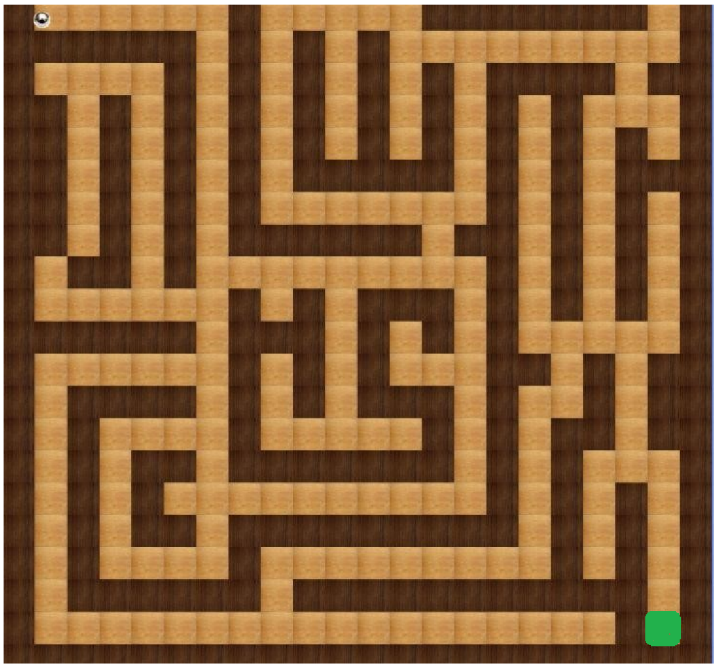


Рис. 3.27. Згенерований лабіринт 20x20

ВИСНОВКИ

Під час виконання дипломної роботи було досліджено технологію MonoGame та її використання в поєднанні з XamarinForms. Було розроблено ігровий застосунок “Лабіринт”. Даний застосунок створено за допомогою засобів MonoGame для ОС Android. Вебзастосунок оптимізовано для пристроїв на ОС Android. Інтерфейс зручний для користування, керування грою відлагоджено. Прописано всі можливості побудови лабіринтів та пошуку шляху у них. Дана гра має яскраву та динамічну 2Д графіку з картами колізій.

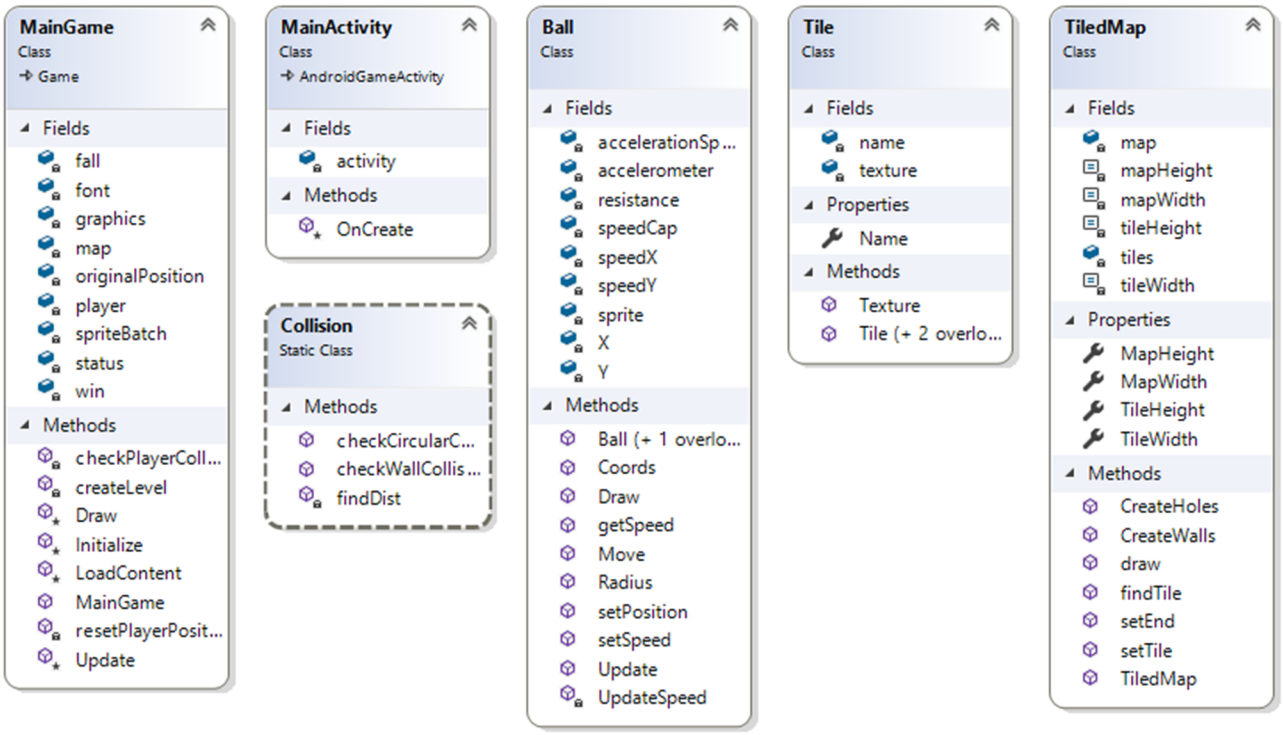
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. github.com
2. https://xamarin.com
3. https://metanit.com
4. https://stackoverflow.com
5. https://www.habr.com
6. https://www.wikipedia.org
7. http://www.gamedev.ru/
8. https://www.codeguru.com/csharp
9. https://lsreg.ru/
10. https://app2top.ru/game_development

ДОДАТКИ

Додаток 1

Class Diagram



Додаток 2

Основний код застосунку – Головна сторінка

```
public MainGame()
    {
        graphics = new GraphicsDeviceManager(this);
        graphics.IsFullScreen = true;
        graphics.SupportedOrientations = DisplayOrientation.LandscapeLeft;
        Content.RootDirectory = "Content";
    }

    protected override void Initialize()
    {
        base.Initialize();
    }
    /// <summary>
    /// Load the game content
```

```
/// </summary>
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    font = Content.Load<SpriteFont>("Font");
    win = Content.Load<SoundEffect>("victoire");
    fall = Content.Load<SoundEffect>("trou");

    map = new TiledMap(this.GraphicsDevice);
    player = new Ball(this.GraphicsDevice, 50, 50);

    status = 0;
    createLevel();
}
/// <summary>
/// Update the game to its next state.
/// </summary>
/// <param name="gameTime">current GameTime</param>
protected override void Update(GameTime gameTime)
{
    base.Update(gameTime);
    if (status == 0)
    {

        TouchCollection touchCollection = TouchPanel.GetState();

        if (touchCollection.Count > 0)
        {
            status++;
            createLevel();
        }
    }
    player.Update();
    checkPlayerCollision();
    player.Move();
}
/// <summary>
/// Draw the current game status
/// </summary>
```

```
/// <param name="gameTime">current GameTime</param>
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();

    map.draw(spriteBatch);
    player.Draw(spriteBatch);

    if(status == 0)
    {
        spriteBatch.DrawString(font, "Press anywhere to start", new Vector2(800, 600), Color.Black);
    }
    spriteBatch.End();

    base.Draw(gameTime);
}
/// <summary>
/// Checks the collision of the player with holes and walls.
/// Updates the player speed accordingly
/// </summary>
private void checkPlayerCollision()
{
    Vector2 playerPos = player.Coords();
    int playerRadius = player.Radius();
    int playerCoordX = (int)(playerPos.X + playerRadius) / map.TileWidth;
    int playerCoordY = (int)(playerPos.Y + playerRadius) / map.TileHeight;
    int playerPosX = (int)(playerPos.X + playerRadius) % map.TileWidth;
    int playerPosY = (int)(playerPos.Y + playerRadius) % map.TileHeight;
    Log.Debug("Main", "Coords: " + playerPos.ToString());
    Vector2 newSpeed = player.getSpeed();
    /* Checking for wall collision */
    for (int x = -1; x < 2; x++)
    {
        for (int y = -1; y < 2; y++)
        {
            Vector2 coords = new Vector2(playerCoordX + x, playerCoordY + y);
            /* Checking if we're within range */
            if (coords.X >= 0 && coords.X < map.MapWidth && coords.Y >= 0 && coords.Y < map.MapHeight)
            {
```

```
                /* Checking if it is a wall */
                if (map.findTile((int)coords.X, (int)coords.Y).Name == "mur")
                {
                  /* Checking if there is a collision */
                  if (Collision.checkWallCollision(coords, map.TileWidth, new Vector2(playerCoordX, playerCoordY),
new Vector2(playerPosX, playerPosY), playerRadius))
                  {
                    /* If the wall was left */
                    if (x == -1 && y == 0)
                    {
                      Log.Debug("Collision", "Left");
                      newSpeed.X = newSpeed.X < 0 ? 0 : newSpeed.X;
                    }
                    /* If the wall was right */
                    else if (x == 1 && y == 0)
                    {
                      Log.Debug("Collision", "Right");
                      newSpeed.X = newSpeed.X > 0 ? 0 : newSpeed.X;
                    }

                    /* If the wall was top */
                    else if (y == -1 && x == 0)
                    {
                      Log.Debug("Collision", "Top");
                      newSpeed.Y = newSpeed.Y < 0 ? 0 : newSpeed.Y;
                    }
                    /* If the wall was bot */
                    else if (y == 1 && x == 0)
                    {
                      Log.Debug("Collision", "Bot");
                      newSpeed.Y = newSpeed.Y > 0 ? 0 : newSpeed.Y;
                    }

                  } /* End of collision check */
                } /* End of wall check */
              }/* End of range check */
            } /* end of for y */
          }/* end of for x */

          /* Checking for the exit */
          if (map.findTile(playerCoordX, playerCoordY).Name == "sortie")
```

```
        {
            win.Play();
            status++;
            createLevel();
            Log.Debug("Main", "Sortie!");
        }
        player.setSpeed(newSpeed);

        /* Checking for a hole */
        if (map.findTile(playerCoordX, playerCoordY).Name == "trou") {
            if (Collision.checkCircularCollision(new Vector2(playerCoordX + map.TileWidth / 2, playerCoordY + map.TileHeight / 2), 20, new Vector2(playerCoordX, playerCoordY), new Vector2(playerPosX, playerPosY))){
                fall.Play();
                resetPlayerPosition();
                Log.Debug("Main", "Trou!");
            }
        }


    } /* End of checkPlayerCollision */

    /// <summary>
    /// Replace the current map with the next level.
    /// Level is found using the "status" variable.
    /// </summary>
    private void createLevel()
```

Додаток 3

Клас обробник кулі

```
public Ball(GraphicsDevice graphicsDevice)
    {
        X = 0;
        Y = 0;

        speedX = 0.5f;
        speedY = 0.5f;

        accelerationSpeed = 1.0f;

        speedCap = 5f;
```

```
    resistance = 0.3f;


    if (accelerometer == null)
    {
        accelerometer = new Accelerometer();

        accelerometer.Start();

    }
    using (var stream = TitleContainer.OpenStream("Content/balle.png"))
    {
        sprite = Texture2D.FromStream(graphicsDevice, stream);
    }

}
public Ball(GraphicsDevice graphicsDevice, int x, int y)
{
    X = x;
    Y = y;
    accelerationSpeed = 1.0f;
    speedCap = 10f;
    resistance = 0.1f;
    if (accelerometer == null)
    {
        accelerometer = new Accelerometer();

        accelerometer.Start();

    }
    using (var stream = TitleContainer.OpenStream("Content/balle.png"))
    {
        sprite = Texture2D.FromStream(graphicsDevice, stream);
    }
}
public void Update()
{
    UpdateSpeed();
    Log.Debug("Ball", "Position: " + X + " " + Y);
}
```

```
private void UpdateSpeed()
{

    float orientationX = accelerometer.CurrentValue.Acceleration.Y;
    float orientationY = accelerometer.CurrentValue.Acceleration.X;

    orientationX = (float)(Math.Round((double)orientationX, 1));
    orientationY = (float)(Math.Round((double)orientationY, 1));
    if (orientationX == 0)
    {
        if (speedX < -resistance || speedX > resistance)
        {
            speedX += speedX > 0 ? -resistance : resistance;
        }
        else {
            speedX = 0;
        }
    }
    else
    {
        speedX += orientationX * accelerationSpeed;
    }
    if (orientationY == 0)
    {
        if (speedY < -resistance || speedY > resistance)
        {
            speedY += speedY > 0 ? -resistance : resistance;
        }
        else {
            speedY = 0;
        }
    }
    else
    {
        speedY += orientationY * accelerationSpeed;
    }
    speedX = speedX > speedCap ? speedCap : speedX < -speedCap ? -speedCap : speedX;
    speedY = speedY > speedCap ? speedCap : speedY < -speedCap ? -speedCap : speedY;
    X = X % 1920;
    Y = Y % 1080;
}
```

```
public void Move()
{
   X += speedX;
   Y += speedY;
}
public void setSpeed(Vector2 speed)
{
   speedX = speed.X;
   speedY = speed.Y;
}
public Vector2 getSpeed()
{
   return new Vector2(speedX, speedY);
}

public void setPosition(Vector2 position)
{
   X = position.X;
   Y = position.Y;
   speedX = 0;
   speedY = 0;
}
public Vector2 Coords()
{
   return new Vector2(X, Y);
}
public int Radius()
{
   return sprite.Width / 2;       }
               public void Draw (SpriteBatch spriteBatch)
               {          spriteBatch.Draw(sprite, new Vector2(X, Y), Color.White);              }   }
```