Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

# КВАЛІФІКАЦІЙНА РОБОТА
## Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

**ДОСЛІДЖЕННЯ ТА ЗАСТОСУВАННЯ ZENJECT І UNIRX ДЛЯ РОЗРОБКИ МОБІЛЬНОГО ЗАСТОСУНКУ НА UNITY**
(тема)

Виконав:
студент 4 курсу, групи ІТІНФ-20-1
Махно Р.А.
(прізвище, ініціали)
Спеціальності 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Інформатика
(повна назва освітньої програми)
Керівник доц. Шафроненко.А.Ю.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ____________ Кобилін О.А.
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри ______________
(підпис)
«____»_______________ 2024 р.

# ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Махно Руслану Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження та застосування Zenject і UniRx для розробки мобільного застосунку на Unity

затверджена наказом університету від 20 травня 2024 року № 464 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 1 червня 2024 р.

3. Вихідні дані до роботи науково-методична та науково-технічна література з програмування на Unity, дані інтернет-мережі, бібліотеки з GitHub, двигун для розробки ігор Unity, середовище розробки JetBrains Rider, мова програмування C#.

4. Перелік питань, що потрібно опрацювати в роботі

1. Огляд літератури та дослідження ресурсів для розробки гри.
2. Навчання використанню Unity та експерименти з бібліотеками.
3. Розробка мобільного застосунку.
4. Тестування та оптимізація.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Актуальность проблеми розширюваної системи, постанова задачі, блок-схеми синтаксичного прикладу на мові C#, показові блок-схеми використання патернів.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

| Найменування розділу | Консультант (посада, прізвище, ім'я, по батькові) | Позначка консультанта про виконання розділу | |
|---|---|---|---|
| | | підпис | дата |
| | | | |

**КАЛЕНДАРНИЙ ПЛАН**

| № з/п | Назва етапів роботи | Терміни виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Отримання завдання на кваліфікаційну роботу | 08.04.2024 | |
| 2 | Аналіз завдання, підбір літератури | 08.04.24-17.04.24 | |
| 3 | Аналіз літератури з досліджуваної проблеми | 18.04.24-23.04.24 | |
| 4 | Аналіз технічних засобів та фреймворків | 24.04.24-30.04.24 | |
| 5 | Розробка адаптивної системи на Unity | 01.05.24-07.05.24 | |
| 6 | Програмна реалізація | 08.05.24-16.05.24 | |
| 7 | Оформлення пояснювальної записки | 17.05.24-18.05.24 | |
| 8 | Перевірка на плагіат | 24.05.24-02.06.24 | |
| 9 | Рецензування | 03.06.24-04.06.24 | |
| 10 | Підготовка презентації та доповіді | 05.06.24-09.06.24 | |
| 11 | Занесення роботи в електронний архів | 12.06.24 | |
| 12 | Попередній захист кваліфікаційної роботи | 12.06.24 | |

Дата видачі завдання 8 квітня 2024 р.

Студент ______________________________
(підпис)

Керівник роботи ______________________ доц. Шафроненко А.Ю.
(підпис) (посада, прізвище, ініціали)

# РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 55 с., 2 табл., 36 рис., 30 джерел.

АДАПТИВНА СИСТЕМА, МОДУЛЬНІСТЬ, UNITY, АРХІТЕКТУРА, ЗАСТОСУВАННЯ ФРЕЙМВОРКІВ ТА БІБЛІОТЕК, ZENJECT, UNIRX.

Об’єктом роботи є дослідження та застосування фреймворків Zenject та UniRx при створенні адаптивної системи, використовуючи ігровий двигун Unity.

Метою цієї роботи є проектування розширюваної архітектури, поділ на модулі та оптимізація процесів розробки мобільної гри на Unity, за рахунок використання патернів програмування, застосування фреймворків та бібліотек для поліпшення робочої ефективності коду.

Виконано побудову програмного середовища, завдяки двигуну Unity та використанню Unity API для проектування загальної архітектури гри, збудовано сцени та налаштовані графічні системи двигуна.

ADAPTIVE SYSTEM, MODULARITY, UNITY, ARCHITECTURE, APPLICATION OF FRAMEWORKS AND LIBRARIES, ZENJECT, UNIRX.

The object of the work is to study and apply the Zenject and UniRx frameworks in creating an adaptive system using the Unity game engine.

The aim of this work is to design an extensible architecture, divide into modules and optimize the development of a mobile game on Unity by using programming patterns, frameworks and libraries to improve the working efficiency of the code.

The software environment was built using the Unity engine and the Unity API to design the overall architecture of the game, build scenes and configure the engine’s graphics systems.

# ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів ........... 6
Вступ ........................................................................................................................ 7
1. Проєктування адаптивної системи при розробці гри .................................... 10
1.1 Поняття архітектури ...................................................................................... 10
1.2 Програмування систем на мові C# та Unity API ........................................... 12
1.3 Дослідження фреймворків та бібліотек для поліпшення системи .. 20
1.4 Постановка задачі .......................................................................................... 23
2. Моделювання системи з використанням бібліотек і фреймворків ............. 24
2.1 Вибір відповідних патернів проектування ................................................... 24
2.2 Використання проміжного програмного забезпечення ................. 32
2.3 Масштабованість і модульність ..................................................................... 41
3. Результати комп'ютерного програмування ................................................... 45
3.1 Вибір версії та середовища для програмування ............................................ 45
3.2 Створення проекту на Unity .......................................................................... 47
3.3 Програмування модульної системи ............................................................... 50
Висновки ................................................................................................................ 59
Перелік джерел посилання .................................................................................... 60

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

DI – Dependency Injection (впровадження залежностей)

OOP – Object-Oriented Programming (Об’єктно-орієнтоване програмування)

API – Application Programming Interface (Інтерфейс прикладного програмування)

MVC – Model-View-Controller (Модель-Вид-Контроллер)

MVVM – Model-View-ViewModel (Модель-Вид-Модель подання)

ECS – Entity Component System (Система сутностей і компонентів)

IDE – Integrated Development Environment (Інтегроване середовище розробки)

# ВСТУП

У такому дуже динамічному середовищі розробки мобільних ігор ефективність та модульність стають вирішальними факторами, які роблять проєкт дійсно успішним. Unity є одним з найкращих у світі рішень для розробки ігор і пропонує одну з найкращих платформ для розробки, яка допомагає створювати насичені, високоінтерактивні мобільні ігри.

Розробники по всьому світу надають перевагу Unity, оскільки вона має функції, які дозволяють прискорити розробку завдяки простому інтерфейсу та збагаченому функціоналу. Разом з тим, складне завдання управління складними залежностями та реактивними потоками даних в Unity з кінця в кінець може бути досить складним. Саме в такому середовищі працюють такі просунуті фреймворки, як Zenject та UniRx, які вдосконалюють архітектуру Unity, щоб зробити процеси розробки ефективнішими, а продуктивність ігор – кращою.

Що робить Unity особливим у розробці ігор, так це його привабливість до універсальності та гнучкості. Зазвичай Unity підтримує різні платформи, дозволяючи розробникам кодувати один раз, а потім розгортати на iOS, Android або Windows Phone. Така крос-платформна гнучкість значно скорочує вартість і час розробки, що робить його підходящим вибором для мобільних ігрових проектів. Unity також має багате сховище ресурсів, інтегроване середовище розробки, потужний 3D та 2D рендеринг та програмне забезпечення для моделювання, що допоможе створити найкращу та захоплюючу гру-блокбастер для мобільних пристроїв в рамках бюджету та за графіком. Середовище написання сценаріїв на C# є одночасно доступним для новачків і досить потужним для професіоналів.

Zenject – це фреймворк ін’єкції залежностей (DI), розроблений спеціально для Unity. Він є незамінним для розробки ігор, оскільки сприяє модульності в архітектурі програмного забезпечення, зменшуючи зв’язок між різними

частинами програмного забезпечення. Це полегшує керування, тестування і навіть масштабування системи.

Zenject, зокрема, дуже добре працює з компонентно-орієнтованою архітектурою Unity, пропонуючи шлях до більш чистого і легкого в обслуговуванні коду завдяки відмінному управлінню залежностями. Це спрощує складні ієрархії об’єктів та життєві цикли, що є надзвичайно корисним для великих мобільних ігор з багатьма системами, які взаємодіють одночасно.

UniRx (Reactive Extensions for Unity) – це бібліотека, натхненна реактивними розширеннями .NET і створена для перенесення реактивного програмування в Unity. Вона дозволяє легко обробляти асинхронні потоки даних за допомогою декларативного стилю програмування, керованого подіями. UniRx дозволяє працювати зі складними ланцюжками потоків даних і ланцюжками подій, не занурюючись у пекло зворотних викликів, тому код стає більш читабельним і зручним для супроводу. Все це виявляється ще більш корисним при розробці мобільних ігор, де продуктивність і управління ресурсами відіграють важливу роль, перебуваючи в умовах апаратних обмежень мобільних пристроїв.

По суті, інтеграція Zenject з UniRx у проекти Unity запроваджує новий спосіб розробки, який має архітектуру, що заохочує до більшої структурованості та надійності.

Механізм Zenject Di забезпечує послаблення зв’язків між компонентами і, як наслідок, їх досить легко замінити або модернізувати. З іншого боку, реактивна взаємодія з користувачем UniRx та асинхронні події гарантують досконаліть моделі в цілому. Разом вони дозволяють усунути помилки, підвищити якість коду та покращити продуктивність гри, що є критично важливими для утримання переваги на швидкоплинному ринку мобільних ігор. Іншими словами, поєднання Zenject та UniRx з Unity дозволяє подолати не лише основні проблеми, з якими стикаються розробники в процесі розробки мобільних ігор, але й дає можливість зробити гру ще більш динамічною,

масштабованою та зручною в обслуговуванні. Така інтеграція може бути репрезентативним прикладом того, як за допомогою таких просунутих фреймворків можна значно підняти рівень архітектури в розробці ігор.

У даній роботі буде запропоновано та програмно змодельовано підхід для розробки застосунку на платформі Unity з використанням фреймворків для створення ефективної та адаптивної архітектури, отримано висновки по результатам його тестування та публікації. Застосування такого підходу дозволило створити розширювану систему для подальших змін чи реструктуризації. Також цей метод може бути ефективно використаний при розробці мобільної гри у командах та у великих компаніях.

# 1 ПРОЄКТУВАННЯ АДАПТИВНОЇ СИСТЕМИ ПРИ РОЗРОБЦІ ГРИ

## 1.1 Поняття архітектури

Більшість розробників погодяться з теорією, що міцна основа і сила, яка лежить в основі ефективної розробки проекту, його ремонтопридатності та масштабованості, лежить в архітектурі проекту. Правильна архітектура дійсно буде ключем до вирішення загальних проблем, з якими більшість стикається при розробці ігор, таким чином гарантуючи, що проект буде побудований і буде успішним з часом.

У розробці програмного забезпечення та ігор архітектура просто означає високорівневу структуру системи. Вона визначає, як організовані компоненти програми, як ці компоненти взаємодіють між собою, а також настанови та принципи, що керують їхньою розробкою та розвитком. Серед найважливіших аспектів архітектури є: модульність, масштабованість, ремонтопридатність та оптимізація продуктивності [1].

Архітектура розробки ігор – це те, на чому не варто надмірно акцентувати увагу. Вона має справу з кількома аспектами процесу розробки, гарантуючи, що проект є керованим, підтримуваним, масштабованим, спільним та повторно використовуваним.

Проектування архітектури відбувається поетапно, розглянемо їх:

– Хороша архітектура декомпозує надскладні системи на невеликі, керовані модулі. Кожен модуль відповідає лише за один аспект функціональності, що робить його дуже зрозумілим і простим у реалізації. Наприклад, декомпозиція ігрової логіки на окремі модулі – такі як обробка вводу, фізика, рендеринг та ШІ – може значно спростити загальну розробку;

– Це допомагає інженеру-програмісту працювати так, щобусе, що він робить в одній частині, не впливало на інші частини системи. Таке розділення завдань є дуже фундаментальним базовим принципом модульного дизайну;

– дозволяє розробнику розділяти дані та поведінку. Таким чином, зміни можуть бути локалізовані в певній частині системи, де вони не створюють нових помилок і не ускладнюють контроль залежностей. Наприклад, компонент Health, що відповідає за здоров'я персонажа, можна розробляти і тестувати незалежно від компонента AI, що відповідає за поведінку персонажа;

– у добре структурованій архітектурі всі компоненти можуть оновлюватися незалежно. Це означає, що, наприклад, якщо потрібно оптимізувати систему рендерингу, її можна оновити, не втручаючись у фізику або системи введення, це робить кодову базу більш гнучкою для нових функцій і виправлення помилок [2];

– у такому модулі легше виявити та виправити помилку. У разі виникнення проблеми, її можна виявити в модулі, не перевіряючи всю величезну монолітну кодову базу. Оскільки програмне забезпечення має меншу модульність, налагодження та підтримка займають набагато менше часу;

– у міру розвитку гри, добре збудована структура повинна мати можливість легко підтримувати нові функції, а підтримка багатших або складніших взаємодій не повинна вимагати повного перепроектування. Це актуально для довгострокових проектів, які постійно розвиваються;

– ретельно спроектовані з самого початку гри дбають про потужність, це означає, що гра залишається ефективною при масштабуванні. Такий підхід до проектування передбачає ефективне керування пам'яттю та способи мінімізації навантаження на процесор і графічний процесор;

– коли модулі та інтерфейси чітко визначені, члени команди знають свої обов'язки, що від них очікується, що вони роблять і як їхня робота інтегрована у велику систему. Таким чином, з чіткими визначеннями в роботі зменшується двозначність, що робить співпрацю кращою;

– команди можуть працювати над різними модулями одночасно, не

заважаючи одна одній. Наприклад, одна команда може займатися розробкою системи пересування гравця, а інша зосередитися на ШІ супротивника, тим самим підвищуючи ефективність часу розробки.Уможливлення повторного використання.

## 1.2 Програмування систем на мові C# та Unity API

C# (вимовляється як «Сі-шарп») – це сучасна об’єктно-орієнтована мова програмування, розроблена компанією Microsoft. Цю мову було розроблено як просту, потужну та універсальну мову для розробки застосунків – не тільки вебзастосунків і мобільних програм, а й ігор та корпоративного програмного забезпечення.

C# було розроблено 2000 року в рамках ініціативи Microsoft .NET. Творцем мови став Андерс Гейлсберг, відомий розробкою кількох досить успішних мов програмування: Turbo Pascal і Delphi. Він спробував надати мові міць і гнучкість, та водночас зберегти простоту і легкість Visual Basic.

Мова C# може бути заснована на об’єктно-орієнтованій системі, що, як правило, допомагає полегшити роботу з великим програмним забезпеченням або його підтримку [5].

Однією з головних причин популярності мови C# є інтеграція з фреймворком .NET. Він пропонує широкий спектр готових рішень для типових програмістських завдань і середовище виконання, яке керує виконанням програм. Завдяки цій інтеграції застосунки на C# можуть легко використовувати потужні можливості .NET Framework, а саме: підтримка різних мов, надійний захист даних і великі бібліотеки (рис. 1.1) .

```
using System;

namespace HelloWorld
{
    new *
    class Program
    {
        new *
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

Рисунок 1.1 – Базовий синтакис C#

В контексті використання мови C# при програмуванні гри на Unity, не треба забувати, що основою для цього є ООР.

Об’єктно-орієнтоване програмування (ООП) – це парадигма програмування, в основі якої лежить поняття «об’єктів», які можна визначити як екземпляри класів (рис. 1.2).

Ось чотири основні принципи ООП:

– об’єднання даних і методів, що оперують цими даними, в межах одного об’єкта або класу, обмежуючи прямий доступ до деяких компонентів об’єкта називається інкапсуляція;

– спрощення складних систем шляхом моделювання класів, що відповідають проблемі, і робота на високому рівні узагальнення має назву абстрагування;

– дозволяє новим класам успадковувати властивості та методи від існуючих класів, сприяючи повторному використанню коду;

– дозволяє розглядати об’єкти як екземпляри батьківського класу, а не

як власне клас, що забезпечує гнучкість та використання загальних інтерфейсів, тому має назву поліморфізм.

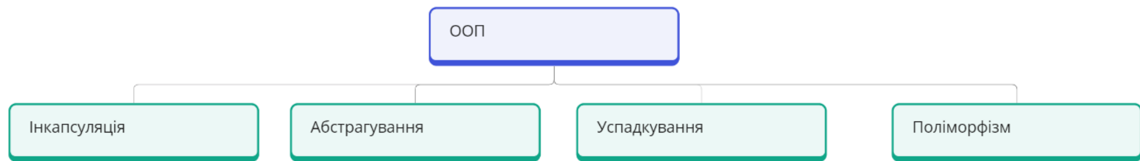


Рисунок 1.2 – Блок-схема принципів ООП

Реалізація концепцій ООП у C# гарантує, що розробники можуть створювати застосунки, які легко підтримувати та розширювати з часом. Підтримка мовою інкапсуляції, абстрагування, успадкування та поліморфізму допомагає створювати добре структуровані та надійні програми. Дотримуючись цих принципів ООП, розробники можуть ефективно вирішувати більш складні проблеми та розширювати свої можливості у проектуванні програмного забезпечення [29].

Першим приницов є інкапсуляція , одне з фундаментальних понять об’єктно-орієнтованого програмування. Вона означає об’єднання даних (змінних) і методів (функцій), що оперують цими даними, в єдиний об’єкт або клас, а також обмеження доступу до деяких компонентів об’єкта. Зазвичай це робиться для запобігання випадковій модифікації даних, чіткого розмежування між даними об’єкта та маніпуляціями з ними, а також для інкапсуляції складності.

У C# поля (змінні, оголошені на рівні класу) часто роблять закритими, щоб приховати їхні значення ззовні класу. C# використовує властивості (комбінацію методу та поля), щоб забезпечити контрольований доступ до цих полів. Властивості можуть визначати геттери та сеттери, де сеттер може включати перевірку для забезпечення цілісності даних (рис. 1.3).

```csharp
public class Person
{
    private string name;  // Private field

    public string Name    // Public property
    {
        get => name;
        set
        {
            if (!string.IsNullOrEmpty(value)) name = value;
            else throw new ArgumentException("Name cannot be null or empty");
        }
    }
}
```

Рисунок 1.3 – Приклад інкапсуляції

Абстрагування передбачає приховування складної реальності, виставляючи на показ лише необхідні частини. Це допомагає зменшити складність і трудомісткість програмування. У C# абстракція досягається завдяки використанню абстрактних класів та інтерфейсів [22].

Абстрактні класи можуть містити як абстрактні методи (без реалізації), так і конкретні методи (з реалізацією). Вони не можуть бути конкретизовані і потребують підкласів для забезпечення реалізації абстрактних методів.

Інтерфейси визначають договір для класів без надання реалізації. Класи, які реалізують інтерфейс, погоджуються реалізувати всі методи, описані інтерфейсом, що сприяє множинному успадкуванню (рис. 1.4).

```csharp
public interface IAnimal
{
    void Eat();
}

public abstract class Animal : IAnimal
{
    public abstract void Eat();

    public void Breathe()
    {
        Console.WriteLine("Breathing...");
    }
}

public class Dog : Animal
{
    public override void Eat()
    {
        Console.WriteLine("Eating dog food.");
    }
}
```

Рисунок 1.4 – Приклад абстракції

Спадкування дозволяє класу успадковувати властивості та методи іншого класу. У C# класи підтримують одиночне успадкування, але можуть реалізовувати декілька інтерфейсів. Це допомагає уникнути складності та неоднозначності, які може спричинити множинне успадкування класів (рис. 1.5).

```
public class BaseClass
{
    public void Display()
    {
        Console.WriteLine("Base display.");
    }
}

public class DerivedClass : BaseClass
{
    public void Show()
    {
        Console.WriteLine("Derived show.");
    }
}
```

Рисунок 1.5 – Приклад спадкування

Поліморфізм дозволяє методам виконувати різні дії в залежності від об’єкту, над яким вони працюють. У C# поліморфізм досягається головним чином за допомогою перевизначення методів (з використанням ключових слів virtual та override) та перевантаження методів [7].

Перевизначення методу дозволяє підкласу забезпечити специфічну реалізацію методу, який вже визначений у базовому класі. Це робиться за допомогою ключового слова virtual у методі базового класу та ключового слова override у методі похідного класу [1].

Перевантаження методів дозволяє декільком методам в одній області видимості мати однакове ім’я, але різні параметри (рис. 1.6).

```
1 usage  1 inheritor  new *
public class Animal
{
    1 override  new *
    public virtual void Speak()
    {
        Console.WriteLine("Some generic animal sound");
    }
}

new *
public class Cat : Animal
{
    new *
    public override void Speak()
    {
        Console.WriteLine("Meow");
    }
}
```

Рисунок 1.6 – Приклад поліморфізму

Переваги використання цих принципів:

– інкапсуляція та абстрагування допомагають в управлінні та підтримці коду, зменшуючи його складність;

– спадкування дозволяє розробникам повторно використовувати існуючий код, зменшуючи надмірність та зусилля;

– поліморфізм та динамічне зв’язування забезпечують гнучкість

коду, що полегшує впровадження та розширення функціональності;

– використання ООП дозволяє інтуїтивно зрозуміліше відображати реальні проблеми на програмні конструкції, покращуючи дизайн і структуру програм.

Unity – популярна платформа для розробки ігор, широко відома своїми потужними можливостями та здатністю зробити розробку ігор більш доступною як для початківців, так і для професійних розробників. Важливою складовою потужності Unity є її розширений інтерфейс прикладного програмування (API), який дозволяє розробникам програмно взаємодіяти з редактором Unity та середовищем виконання.

Unity API – це набір програмних інструментів та функцій, наданих Unity Technologies, які дозволяють розробникам ігор отримувати доступ до функцій та керувати аспектами рушія Unity Engine програмно. Це містить все, від маніпулювання ігровими об’єктами, управління фізикою та рендерингом графіки до відтворення звуку, управління користувацьким вкладом та управління мережевими операціями. Доступ до Unity API здійснюється переважно за допомогою мови C#, яка взаємодіє з рушієм, визначаючи поведінку та взаємодію гри [12].

Unity API використовується для написання сценаріїв поведінки в іграх Unity, забезпечує функціональність, необхідну для втілення ігрових концепцій в життя (рис. 1.7).

Ось ключові сфери розробки ігор, де Unity API має важливе значення:

– усе в ігровій сцені Unity є GameObject, від персонажів до світла, камер та спецефектів. Unity API дозволяє розробникам створювати, змінювати або знищувати ці об’єкти під час гри. Наприклад, ви можете використовувати API для динамічного створення ворогів, керування їхньою поведінкою або зміни оточення;

– в Unity є вбудований фізичний рушій, а API надає доступ до таких компонентів, як Rigidbodies, Colliders та Physics Materials, які дозволяють розробникам симулювати реалістичну фізичну поведінку. Це має вирішальне

значення для створення правдоподібних взаємодій між об’єктами, таких як зіткнення та гравітаційні ефекти;

– використання API надає розробникам контроль над візуальними аспектами їхньої гри, включаючи безпосереднє керування матеріалами, сітками та текстурами. Він також керує налаштуваннями освітлення та ефектами, які є життєво важливими для створення правильної атмосфери та глибини в іграх [13];

– в Unity підтримує багату систему відтворення звуку, а API дозволяє детально контролювати відтворення звуку в ігрових сценах, включаючи регулювання гучності, зациклення, просторові ефекти та інше;

– використання Unity API надає методи для обробки користувацького вводу з різних пристроїв, таких як клавіатури, миші, геймпади та сенсорні екрани. Це важливо для інтерактивних ігор, де ввід гравця диктує поведінку гри;

– для багатокористувацьких ігор Unity API містить інструменти для керування з’єднаннями, синхронізації рухів гравців та безпечної і ефективної передачі даних через мережу.

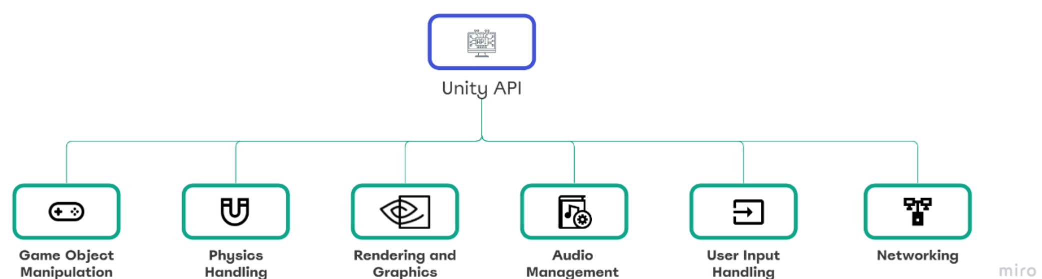


Рисунок 1.7 – блок-схема Unity API

API Unity допомагає розробникам ефективно створювати та керувати різними аспектами відеоігор.

Його комплексний характер охоплює кілька ключових сфер:

– простота та доступність API Unity, він розроблений таким чином, щоб

бути зручним для користувачів, пристосованим як для початківців, так і для досвідчених розробників. API добре задокументований, з численними навчальними посібниками та підтримкою спільноти, що полегшує його вивчення та використання [13];

– розробники можуть розширювати сам Unity Editor за допомогою API, додаючи кастомні інструменти, які покращують робочі процеси та додають нові функції, пристосовані до конкретних потреб;

– API підтримує здатність Unity розгортати ігри на більш ніж 20 платформах, включаючи Windows, macOS, різні ігрові консолі та мобільні пристрої. API обробляє багато специфічних для кожної платформи деталей, дозволяючи розробникам більше зосередитися на дизайні гри і менше – на тонкощах базової платформи;

– API надає інструменти для оптимізації продуктивності гри, включаючи інструменти профілювання, варіанти об’єднання ресурсів та ефективні методи управління сценами.

## 1.3 Дослідження фрейморків та бібліотек для поліпшення системи

Розробляючи ігри в Unity, розробники часто покладаються на бібліотеки та фреймворки, щоб спростити процес, розширити функціональність та покращити якість кінцевого продукту. Розуміння ролі та використання цих компонентів може суттєво вплинути на робочий процес розробки та продуктивність гри [19].

Бібліотека в Unity – це набір заздалегідь написаного коду, який розробники можуть включати у свої проекти для виконання певних функцій без переписування коду з нуля. Бібліотеки зазвичай надають певну функціональність, таку як обробка мережевих комунікацій, виконання складних математичних обчислень або додавання певного ефекту обробки зображень. Вони часто пакуються як DLL (бібліотеки динамічного

зв'язування) або як пакунки Unity, які можна легко імпортувати до проектів Unity.

Unity постачається з набором стандартних бібліотек, які включають базові структури даних, алгоритми та утиліти, призначені для безперешкодної роботи у редакторі Unity та під час виконання. Вони є частиною фреймворку .NET, який використовує Unity.

Існує багато сторонніх бібліотек, які можна інтегрувати у Unity для розширення його функціональності. Вони можуть виконувати такі завдання, як серіалізація JSON, зв'язок з сервером або складні фізичні розрахунки. Приклади включають Newtonsoft.Json для розбору JSON або Lidgren.Network для керування мережею [19].

Плагіни – це спеціалізовані бібліотеки, які часто включають як код, так і власні компоненти, призначені для конкретних апаратних функцій, таких як підтримка VR-гарнітури, кастомних пристроїв введення або інтеграція зі сторонніми сервісами (наприклад, Google Firebase, Facebook SDK).

Фреймворки надають ширший набір інструментів та функцій, ніж бібліотеки, пропонуючи структурований спосіб створення та керування застосунками Unity. Фреймворк зазвичай включає набір бібліотек, а також найкращі практики та парадигми програмування, які керують процесом розробки. Вони забезпечують певний спосіб роботи, який може допомогти у підтримці узгодженості та масштабованості у великих проектах.

Під час розробки ігор в Unity використання фреймворків та бібліотек дає значні переваги, які можуть зробити процес розробки більш ефективним, надійним та масштабованим. Ці інструменти є невід'ємною частиною управління складністю розробки ігор і гарантують, що розробники можуть зосередитися на унікальних і творчих аспектах своїх ігор [14].

Фреймворки та бібліотеки надають добре протестовані, багаторазово використовувані фрагменти коду, які допомагають спростити багато рутинних завдань, пов'язаних з розробкою ігор. Сюди входить робота зі складними структурами даних, виконання мережевих операцій, керування

користувацькими даними, обробка графіки тощо. Покладаючись на ці заздалегідь написані коди, розробники заощаджують значну кількість часу та зменшують ймовірність помилок, оскільки вони використовують інструменти, які були вдосконалені та перевірені в різних середовищах розробки.

Крім того, ці інструменти забезпечують дотримання певних стандартів і структур програмування, що може значно допомогти у підтримці узгодженості проекту, особливо при роботі в команді. Наприклад, фреймворк надає комплексну систему рекомендацій та інструментів для кодування, яка допомагає уніфікувати процес розробки проекту, полегшуючи розробникам розуміння коду один одного для ефективної співпраці.

Використання бібліотек і фреймворків також відкриває доступ до розширених функцій і можливостей, які можуть бути занадто складними або ресурсоємними для розробки з нуля. Наприклад, інтеграція передової фізики, штучного інтелекту або високоякісних графічних ефектів може бути значно спрощена завдяки використанню спеціалізованих бібліотек. Це дозволяє розробникам включати складні функції у свої ігри, не обов’язково маючи глибокі технічні знання, які зазвичай потрібні для створення такої функціональності з нуля [6].

Ще однією значною перевагою є підтримка та спільнота, яка постачається з популярними бібліотеками та фреймворками. Багато з цих інструментів мають великі спільноти розробників, які постійно долучаються до їхнього розвитку, надають підтримку, діляться знаннями та створюють навчальні посібники. Цей аспект спільноти може бути безцінним, особливо при усуненні несправностей або пошуку інноваційних способів вирішення проблеми.

Нарешті, використання цих інструментів у Unity часто покращує продуктивність та оптимізацію ігор. Такі фреймворки, як DOTS у Unity, спеціально розроблені для максимального використання апаратного забезпечення, що має вирішальне значення для розробки ігор, які працюють на широкому спектрі пристроїв [6].

## 1.4 Постановка задачі

Розробка адаптивної системи є актуальним та потрібним процессом під час розробки мобільного застосунку на Unity, тому важливо дослідити та впровадити фреймворк Zenject Dependency Injection та UniRx Reactive Extensions для розробки мобільного застосунку в Unity.

Метою роботи є оцінити, як ці інструменти можуть покращити процес розробки, впорядкувати управління кодовою базою та підвищити продуктивність і ремонтопридатність застосунків.

Потрібно провести ретельне дослідження особливостей, переваг та потенційних проблем, пов’язаних з Zenject та UniRx. Це включає в себе розуміння основних концепцій, що лежать в основі ін’єкції залежностей та реактивного програмування, як вони застосовуються в середовищах Unity.

Розробка невеликого мобільного застосунку в Unity з використанням Zenject для управління залежностями та UniRx для роботи з асинхронним та подієвим програмуванням. Цей додаток слугуватиме підтвердженням концепції, щоб продемонструвати практичне застосування та ефективність цих фреймворків.

Для реалізації цієї мети необхідно виконати наступні завдання:

- огляд літератури та дослідження ресурсів для розробки гри;
- навчання використанню Unity та експерименти з бібліотеками;
- розробка мобільного застосунку;
- тестування та оптимізація.

# 2 МОДЕЛЮВАННЯ СИСТЕМИ З ВИКОРИСТАННЯМ БІБЛІОТЕК І ФРЕЙМВОРКІВ

## 2.1 Вибір відповідних патернів проектування

У програмуванні Unity на C# шаблон програмування – це загальне, багаторазове рішення типової проблеми, що часто зустрічається в певному контексті. Патерни – це найкращі практики, які були протестовані та довели свою ефективність у вирішенні конкретних проблем у дизайні та архітектурі програмного забезпечення. Використання патернів може допомогти зробити ваш код більш модульним, читабельним та зручним для підтримки.

При моделюванні системи було підібрано оптимальний набір патернів, які допомогли мені при створенні модулів архітектури [1].

Патерн Спостерігач визначає залежність «один до багатьох» між об’єктами так, що коли один об’єкт змінює стан, всі його залежні об’єкти отримують сповіщення та оновлюються автоматично. В Unity цей патерн корисний для реалізації систем подій, де декілька об’єктів мають реагувати на певні події (рис. 2.1).

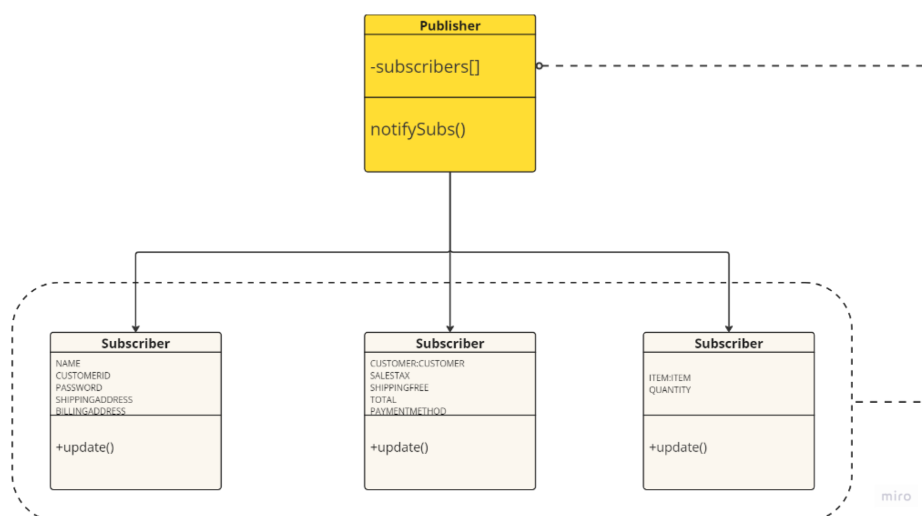


Рисунок 2.1 – Блок-схема патерн Спостерігач

Варто більш детально розповісти, що це таке, та чому треба приділяти більше уваги до цього патерна.

Патерн Фабрика – це шаблон дизайну, який використовується для інкапсуляції конкретних об’єктів. Основна ідея патерну Фабрика полягає в тому, щоб надати можливість створювати об’єкти, не вказуючи точний клас об’єкта, який буде створено. Це особливо корисно у випадках, коли точний тип об’єкта може змінюватися, але всі об’єкти мають спільний інтерфейс або базовий клас [10].

Фабричний патерн сприяє вільному зв’язуванню, зменшуючи залежність від конкретних класів. Це також робить код більш гнучким і легким для розширення або модифікації.

Існує кілька варіацій шаблону Фабрика, кожна з яких слугує певній меті і підходить для різних сценаріїв:

– проста фабрика (метод статичної фабрики);
– фабричний метод;
– абстрактна фабрика.

Проста фабрика, яку часто називають статичним фабричним методом, є базовою версією патерну Фабрики. Він містить один метод, який повертає екземпляри різних класів на основі вхідних параметрів [8].

Варіація фабричний метод визначає інтерфейс для створення об’єкта, але дозволяє підкласам змінювати тип об’єктів, які будуть створені. Ця версія особливо корисна, коли клас не може передбачити тип об’єктів, які йому потрібно створити, або коли відповідальність за створення об’єктів делегується підкласам (рис. 2.2).

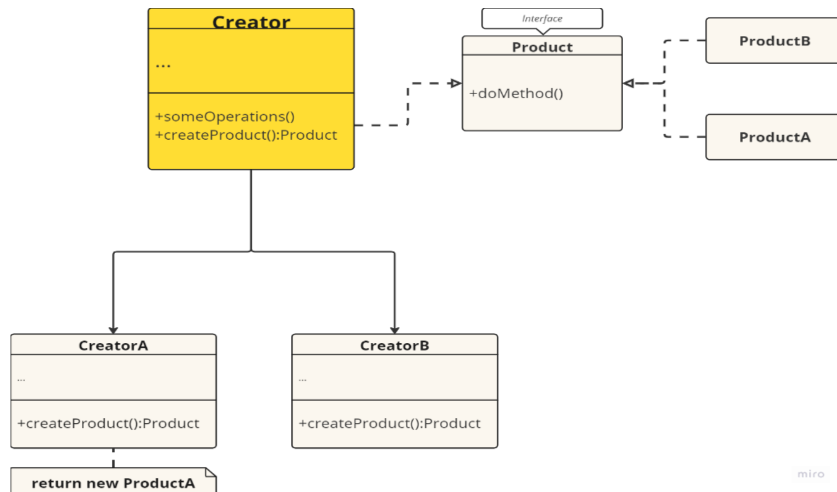


Рисунок 2.2 – Блок-схема фабричного методу

Патерн абстрактна фабрика надає інтерфейс для створення сімейств пов’язаних або залежних об’єктів без зазначення їхніх конкретних класів. Цей варіант фабрики особливо корисний, коли система повинна бути незалежною від того, як створюються, компонуються та представляються її об’єкти ( рис. 2.3).

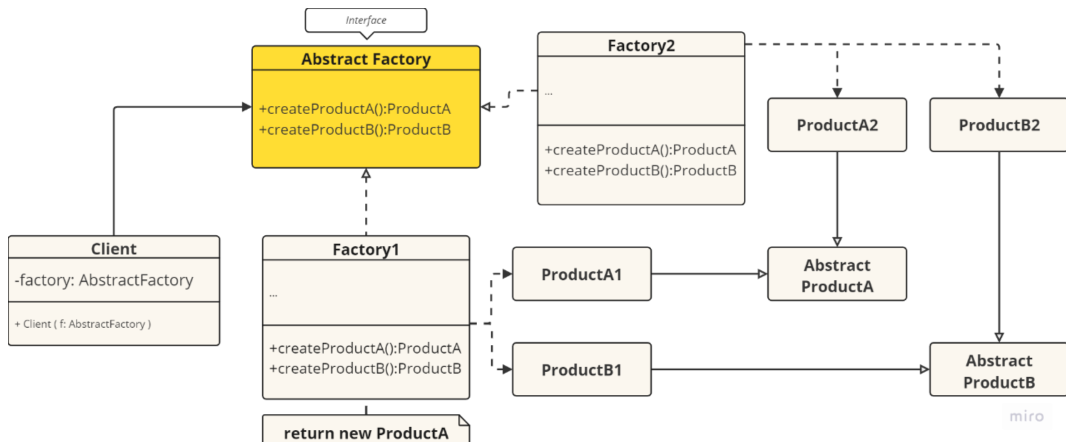


Рисунок 2.3 – Блок-схема абстрактної фабрики

Треба чітко розуміти коли можна ефективно використовувати один з цих типів фабрик.

Тож розглянемо ситуації-маркери:

– проста фабрика використовуємо, якщо маємо просту логіку створення на основі параметрів і не плануємо розширювати логіку створення;

– фабричний метод використовуємо коли клас не може передбачити тип об'єктів, які йому потрібно створити, або коли підкласи відповідають за створення об'єктів;

– абстрактна фабрика використовуємо коли потрібно створити сім'ї пов'язаних об'єктів, які повинні використовуватися разом, і треба гарантувати, що об'єкти будуть сумісними.

Використання фабричних патернів у розробці програмного забезпечення, особливо в контексті Unity та C#, має низку значних переваг, які сприяють створенню більш чистого, модульного та зручного для підтримки коду. Ці патерни допомагають керувати складністю, пов'язаною зі створенням об'єктів, сприяючи гнучкості та масштабованості дизайну [8].

Однією з основних переваг використання фабричних патернів є інкапсуляція створення об'єктів. У традиційному об'єктно-орієнтованому програмуванні процес створення об'єктів часто розкиданий по всій кодовій базі. Такий підхід може призвести до заплутаного і складного в обслуговуванні коду. Фабричні патерни вирішують цю проблему, централізуючи логіку створення об'єктів у межах фабричних класів. Така інкапсуляція гарантує, що деталі створення об'єктів приховані від клієнтського коду, що призводить до більш чистого та модульного дизайну. Наприклад, у грі, розробленій за допомогою Unity, створенням різних типів ворогів може займатися фабрика, яка абстрагується від складнощів, пов'язаних з їхньою конкретизацією [10].

Фабричні патерни значно зменшують залежність клієнтського коду від конкретних класів. Покладаючись на абстрактні класи або інтерфейси, клієнтський код взаємодіє з фабриками, а не безпосередньо з конкретними реалізаціями. Такий вільний зв'язок робить систему гнучкішою і простішою в обслуговуванні. Коли виникає потреба змінити або розширити систему, наприклад, додати нові типи ворогів або модифікувати існуючі, ці зміни можуть бути зроблені в межах фабрики, не впливаючи на клієнтський код. Таке відокремлення має вирішальне значення у великомасштабних проектах,

де зміна однієї частини системи не повинна вимагати значних модифікацій в інших частинах.

Масштабованість – ще одна важлива перевага використання фабричних патернів. Зі зростанням проектів зростає складність і кількість об’єктів, якими потрібно керувати. Фабричні патерни полегшують масштабування системи, надаючи структурований спосіб додавання нових типів об’єктів. Наприклад, додати новий тип зброї або броні в грі може бути так само просто, як створити новий клас і оновити фабричний метод. Цей підхід відповідає принципу відкритості/закритості, одному з принципів об’єктно-орієнтованого проектування SOLID, який стверджує, що програмні об’єкти повинні бути відкритими для розширення, але закритими для модифікації [17].

Супроводжуваність значно покращується завдяки використанню фабричних патернів. Оскільки логіка створення централізована, будь-які зміни, необхідні через нові вимоги, оптимізацію або виправлення помилок, можуть бути зроблені в одному місці. Це зменшує ризик внесення помилок і полегшує керування кодом. Крім того, заводські методи можуть включати ведення журналів, кешування та інші завдання управління, які ще більше підвищують надійність та ефективність системи.

Фабричні патерни також полегшують модульне тестування, спрощуючи створення об’єктів, необхідних для тестів. Тестовий код може використовувати фабрики для створення імітаційних об’єктів, забезпечуючи ізольованість та повторюваність тестів. Такий підхід зменшує складність налаштування тестів і підвищує надійність набору тестів. В Unity це може включати використання фабрик для створення ігрових об’єктів з певними конфігураціями або станами, необхідними для різних тестових сценаріїв.

Абстрагуючись і централізуючи створення об’єктів, патерни фабрик сприяють повторному використанню коду. Одну і ту ж фабрику можна повторно використовувати в різних частинах програми, забезпечуючи узгодженість у створенні та налаштуванні об’єктів. Таке багаторазове використання зменшує дублювання і допомагає підтримувати узгодженість у

всьому проекті. Наприклад, фабрику для створення елементів графічного інтерфейсу в грі можна повторно використовувати в різних сценах або модулях, забезпечуючи узгоджений вигляд і відчуття в усьому застосунку [8].

Фабричні патерни забезпечують гнучкість для динамічної зміни процесу створення об’єктів. Це означає, що на основі різних умов, таких як налаштування конфігурації, вхідні дані користувача або параметри виконання, фабрика може вирішувати, який конкретний клас інстанціювати. Ця гнучкість особливо корисна в іграх, де різні рівні або режими можуть вимагати різних конфігурацій об’єктів.

Патерн Стратегій – це поведінковий патерн проектування, який дозволяє вибрати поведінку алгоритму під час виконання. Замість того, щоб безпосередньо реалізовувати один алгоритм, код отримує інструкції під час виконання про те, який з сімейства алгоритмів використовувати. Патерн Стратегій корисний, коли до задачі можна застосувати декілька алгоритмів, а найкращий алгоритм може змінюватися залежно від контексту [10].

Контекст – це клас, який використовує стратегію. Він конфігурується за допомогою об’єкта ConcreteStrategy і зберігає посилання на об’єкт Strategy.

Інтерфейс стратегії визначає спільний інтерфейс для всіх підтримуваних алгоритмів. Контекст використовує цей інтерфейс для виклику алгоритму, визначеного об’єктом ConcreteStrategy.

Конкретна стратегія – це класи, які реалізують інтерфейс Strategy та інкапсулюють конкретні алгоритми (рис. 2.4).

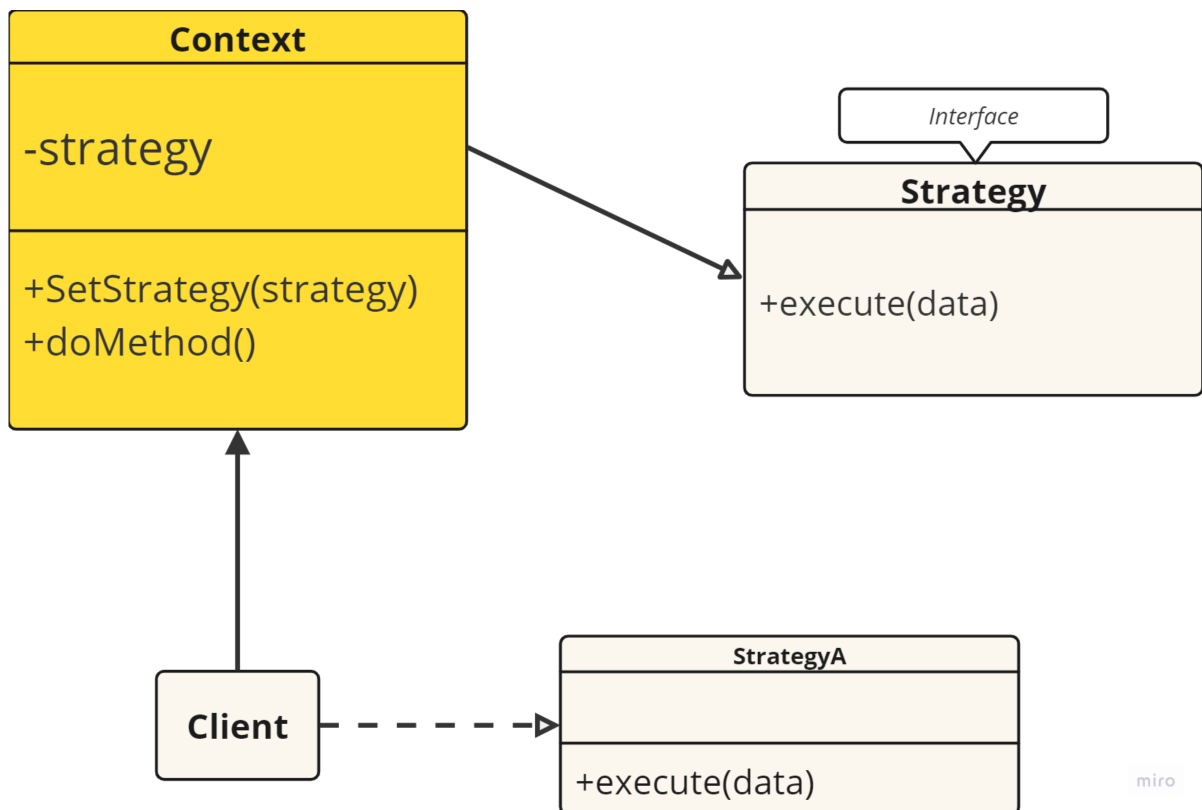


Рисунок 2.4 – Блок-схема патерну Стратегії

Патерн Стратегій пропонує структурований підхід до управління алгоритмами та поведінкою при розробці програмного забезпечення, що робить його особливо корисним у складних застосунках, таких як розробка ігор в Unity.

Однією з найважливіших переваг патерну Стратегій є інкапсуляція алгоритмів. Замість того, щоб вбудовувати декілька алгоритмів безпосередньо в клас, який їх використовує, кожен алгоритм інкапсулюється в межах власного класу. Така інкапсуляція гарантує, що кожен алгоритм можна розробляти, тестувати та підтримувати незалежно. Наприклад, у грі різні способи руху, такі як ходьба, біг та політ, можуть бути інкапсульовані в окремі класи стратегій. Така ізоляція означає, що зміни в одній поведінці не впливають на інші, що зменшує ризик появи помилок [13].

Патерн Стратегій надає гнучкий спосіб змінювати поведінку об’єкта під час виконання. Це особливо корисно у сценаріях, де поведінка об’єкта може змінюватися залежно від ситуації. Наприклад, у грі персонаж може

перемикатися між різними стратегіями пересування (ходьба, біг, політ) залежно від дій гравця або ігрових подій. Така гнучкість досягається без зміни класу, який використовує ці стратегії, що сприяє динамічному та адаптивному дизайну застосунків.

Відокремлюючи алгоритми від класів, які їх використовують, патерн Стратегій покращує зручність супроводу. Коли алгоритм потрібно оновити або замінити, це можна зробити в межах його власного класу, не впливаючи на всю систему. Такий поділ спрощує налагодження та покращує загальну читабельність коду. Наприклад, в Unity, якщо потрібно змінити поведінку персонажа під час виконання, варто змінити лише клас RunStrategy, залишивши решту кодової бази недоторканою [8].

Принцип відкритості/закритості, один з принципів об’єктно-орієнтованого проектування SOLID, стверджує, що програмні об’єкти повинні бути відкритими для розширення, але закритими для модифікації. Патерн Стратегія ідеально відповідає цьому принципу. Нові стратегії можуть бути введені шляхом створення нових класів, які реалізують інтерфейс стратегії, розширюючи функціональність системи без модифікації існуючого коду. Така розширюваність має вирішальне значення для проектів, які, як очікується, розвиватимуться з часом, дозволяючи розробникам безперешкодно додавати нові функції.

Патерн Стратегій сприяє повторному використанню коду, визначаючи загальні інтерфейси для алгоритмів. Ці інтерфейси гарантують, що будь-який клас, який їх реалізує, може використовуватися взаємозамінно. Наприклад, система штучного інтелекту в грі може мати різні стратегії для різних типів поведінки ворога (наприклад, агресивна, оборонна, ухильна). Кожна стратегія може бути повторно використана для різних типів ворогів, зменшуючи надмірність і забезпечуючи узгодженість. Таке повторне використання стратегій призводить до більш ефективного та керованого коду .

Інкапсуляція алгоритмів у власні класи спрощує тестування. Кожну стратегію можна тестувати незалежно, гарантуючи, що вона працює коректно

без втручання інших частин системи. Така ізоляція полегшує написання модульних тестів, виявлення помилок та перевірку правильності кожної поведінки. В Unity незалежне тестування різних типів поведінки рухів гарантує, що кожен тип руху (ходьба, біг, політ) працює як очікується, перш ніж інтегрувати їх у гру.

Для команд, що працюють над великими проектами, патерн стратегії може сприяти кращій співпраці. Визначивши чіткі інтерфейси та розділивши завдання, члени команди можуть працювати над різними стратегіями одночасно без конфліктів. Один розробник може зосередитися на реалізації та вдосконаленні WalkStrategy, в той час як інший працює над FlyStrategy, що призводить до більш ефективних процесів розробки та зменшує ймовірність конфліктів при злитті [10].

Патерн Стратегії дозволяє конфігурувати поведінку під час виконання. У розробці ігор це означає, що поведінка персонажа може динамічно змінюватися у відповідь на ігрові події, дії користувача або інші умови. Наприклад, персонаж може переключитися з оборонної стратегії на агресивну, залежно від стану здоров’я гравця або наявності ворогів. Така гнучкість під час виконання покращує досвід гравця, роблячи гру більш інтерактивною і такою, що реагує на мінливі умови.

## 2.2 Використання проміжного програмного забезпечення

Досліджуючи тему використання зовнішнього програмного забезпечення, було детально розглянуто та протестовано на практиці багато різніх фреймворків. Почнемо з популярного фреймворку Zenject, його часто можна побачити в проектах у компаніях, які займаються розробкою ігор в таких жанрах: гіперказуальні, казуальні, аркади. Також цей фреймворк гарно сінергує з іншими фреймворками та системами, більш складними, які використовуються при розробці мідкор проектів.

Zenject – це фреймворк ін’єкції залежностей (DI), спеціально розроблений для ігрового двигуна Unity. Ін’єкція залежностей – це патерн проектування, який допомагає керувати залежностями між об’єктами шляхом ін’єкції їх із зовнішнього джерела замість того, щоб об’єкти створювали свої власні залежності. Zenject сприяє цьому, надаючи інструменти та методи для ін’єкції залежностей, керування часом життя об’єктів та налаштування залежностей у чистий, модульний та перевірений спосіб (лістинг 2.1).

По суті, Zenject надає надійний фреймворк для ін’єкції залежностей в Unity. Ін’єкція залежностей передбачає передачу залежностей (наприклад, об’єктів або сервісів) в об’єкт, а не створення цих залежностей самим об’єктом [25]. Таке розділення завдань робить код більш модульним і легшим для тестування (рис. 2.5).

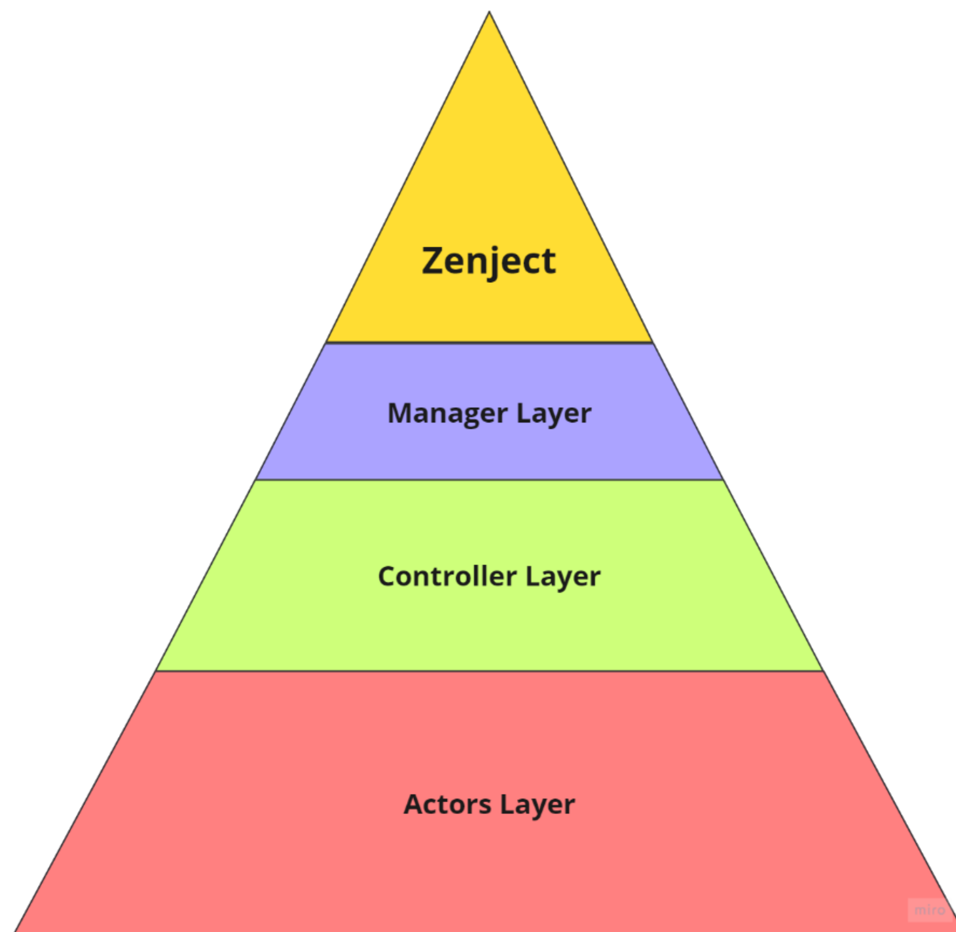


Рисунок 2.5 – Приклад архітектури використовуючи Zenject

Лістинг 2.1 Приклад з програми:

```
[Inject]
public void Construct(IWeapon weapon)
{
    _weapon = weapon;
}
```

Zenject використовує систему зв’язування для зіставлення інтерфейсів або базових класів з їх конкретними реалізаціями (лістинг 2.2).

Зв’язування визначає, як вирішуються залежності, і може бути налаштоване різними способами:

– одиночне зв’язування забезпечує використання єдиного екземпляру в усьому застосунку. (Singleton Binding);

– перехідне зв’язування створює новий екземпляр щоразу, коли запитується залежність. (Transient Binding);

– прив’язка до екземпляра використовує вже існуючий екземпляр. (Instance Binding).

Лістинг 2.2 Приклад зв’язування:

```
Container.Bind<IWeapon>().To<Sword>().AsSingle();
```

Zenject дозволяє розробникам керувати життєвим циклом об’єктів за допомогою різних сценаріїв (лістинг 2.3):

– єдиний екземпляр, який використовується у всьому застосунку;

– новий екземпляр щоразу, коли його запитують;

– схожий на синглтон, але обмежений певним контекстом, наприклад, сценою;

– для динамічного створення екземплярів.

Лістинг 2.3 Приклад зв'язування:

```
Container.Bind<IWeapon>().To<Sword>().AsTransient();
```

Zenject підтримує перевірку залежностей під час компіляції, допомагаючи виявляти помилки на ранніх стадіях. Він також надає утиліти для модульного тестування, що дозволяють вставляти імітаційні залежності та тестувати класи в ізоляції (табл. 2.1).

Таблиця 2.1 – Плюси та мінуси використання Zenject

| **Плюси** | **Мінуси** |
|---|---|
| Сприяє модульності, відокремлюючи залежності від класів, які їх використовують. | Довше навчання порівняно з простішими фреймворками DI. |
| Полегшує написання модульних тестів, дозволяючи вставляти залежності. | Додаткова складність може бути невиправданою для невеликих проектів. |
| Надає гнучкі можливості прив'язки для легкого перемикання між реалізаціями. | Деякі накладні витрати на продуктивність, особливо під час ініціалізації. |
| Покращує ремонтопридатність, локалізуючи зміни та зменшуючи вплив на інші частини коду. | Налагодження проблем з IP може бути більш складним. |
| Підтримує масштабовану архітектуру завдяки ефективному управлінню залежностями та термінами життя. | |

Zenject покращує розробку Unity, надаючи структурований та ефективний спосіб управління залежностями, покращує організацію коду, полегшує тестування та масштабування. Інтеграція зі специфічними для Unity

функціями та підтримка розширених шаблонів проектування роблять Zenject безцінним інструментом для розробників, які прагнуть створювати надійні та зручні в обслуговуванні ігри.

Zenject сприяє модульності, відокремлюючи створення та управління залежностями від об’єктів, які їх використовують. Таке розділення проблем призводить до більш організованої кодової бази, де окремі компоненти легше розуміти, розробляти та підтримувати. Наприклад, у складній грі у вас може бути кілька систем, таких як: керування гравцем, ШІ ворога та управління інтерфейсом. Використовуючи Zenject, кожна система може визначати свої залежності і вставляти їх під час виконання, замість того, щоб жорстко кодувати залежності всередині кожної системи. Така модульність не тільки робить кодову базу чистішою, але й зменшує ризик появи помилок при внесенні змін [25].

Однією з найважливіших переваг Zenject є покращена тестованість, яку він привносить у проекти Unity. Дозволяючи вводити залежності, Zenject спрощує заміну реальних залежностей на імітації або заглушки підчас тестування. Ця можливість має вирішальне значення для написання ізольованих та надійних модульних тестів. Наприклад, маємо ігрового персонажа, який залежить від системи зброї, можна вставити макет системи зброї, щоб протестувати поведінку персонажа, не використовуючи реальну систему зброї. Така ізоляція гарантує, що тести будуть сфокусовані і не зазнають невдачі через не пов’язані з ними проблеми в інших системах.

Zenject надає гнучкі можливості прив’язки, які дозволяють розробникам легко переключатися між різними реалізаціями одного і того самого інтерфейсу. Ця гнучкість особливо корисна при роботі з різними конфігураціями ігор або вимогами, що змінюються. Наприклад, можна почати з простого штучного інтелекту ворога, а згодом вирішити замінити його на більш складний. З Zenject цю зміну можна зробити просто оновивши конфігурацію прив’язки, не змінюючи основну логіку гри. Ця гнучкість поширюється і на управління часом життя об’єктів, де Zenject підтримує

одиночні, перехідні та кешовані екземпляри, гарантуючи, що об'єкти створюються і знищуються відповідно до потреб програми.

Zenject розроблений для безперешкодної інтеграції з Unity, використовуючи його можливості для покращення процесу розробки. Він надає такі інструменти, як контексти сцен та інсталятори скриптових об'єктів, які дозволяють розробникам керувати залежностями в екосистемі Unity. Контексти сцен дозволяють різним частинам гри мати власний набір прив'язок, що корисно для управління залежностями у складних сценах або багатосценарних налаштуваннях. Інсталятори ScriptableObject дозволяють визначати конфігурацію та логіку зв'язування в ресурсах ScriptableObject, що полегшує керування та налаштування залежностей без зміни коду. Ця інтеграція допомагає підтримувати чіткий поділ між конфігурацією та логікою, що ще більше підвищує ремонтопридатність проекту.

Zenject включає потужну систему сигналів, яка відокремлює видавців подій від підписників, полегшуючи управління та реагування на ігрові події. Сигнали є надійною альтернативою вбудованій системі подій Unity, забезпечуючи більш структурований і безпечний для типів спосіб обробки подій. Наприклад, можна визначити сигнал, коли гравець набирає очки, і різні системи, такі як користувацький інтерфейс або менеджер очок, можуть підписатися на цей сигнал. Таке розділення гарантує, що зміни у видавцеві подій не вплинуть на підписників, що сприяє створенню більш модульної та зручної системи обробки подій [25].

Zenject полегшує використання просунутих патернів проектування, таких як патерни Factory і Command, які ще більше підвищують гнучкість і ремонтопридатність коду. Наприклад, патерн Factory можна використовувати для динамічного створення ворогів з різною поведінкою, в той час як патерн Command може інкапсулювати дії, такі як рухи гравця або атаки, що дозволяє легко керувати ними та розширювати їх. Завдяки вбудованій підтримці цих патернів, Zenject дозволяє розробникам впроваджувати надійні та масштабовані рішення для поширених проблем у розробці ігор.

Тепер варто відмітити ще один дуже корисний фреймворк UniRx. На відміну від Zenject, цей фреймворк знаходиться в іншій парадигмі програмування, але також добре синергує з іншими фреймворками та системами. Він дуже популярний серед розробників та компаній.

Зустріти UniRx ми можемо у іграх різного жанру, найпопулярніші з них: гіперказульні, казуальні, також 3 в ряд.

UniRx, скорочено від Unity Reactive Extensions – це потужна бібліотека, яка додає реактивні розширення (Rx) до ігрового двигуна Unity. Reactive Extensions – це бібліотека для написання асинхронних програм, заснованих на подіях, з використанням спостережуваних послідовностей та операторів запитів у стилі LINQ. UniRx надає фреймворк для керування складними потоками подій та асинхронними операціями в Unity, полегшуючи написання адаптивного, підтримуваного та масштабованого коду.

Observables представляють потоки даних або події, які можна спостерігати (рис. 2.6). Вони є основними будівельними блоками в UniRx, що дозволяють розробникам працювати з асинхронними потоками даних і програмуванням, керованим подіями (рис. 2.7).

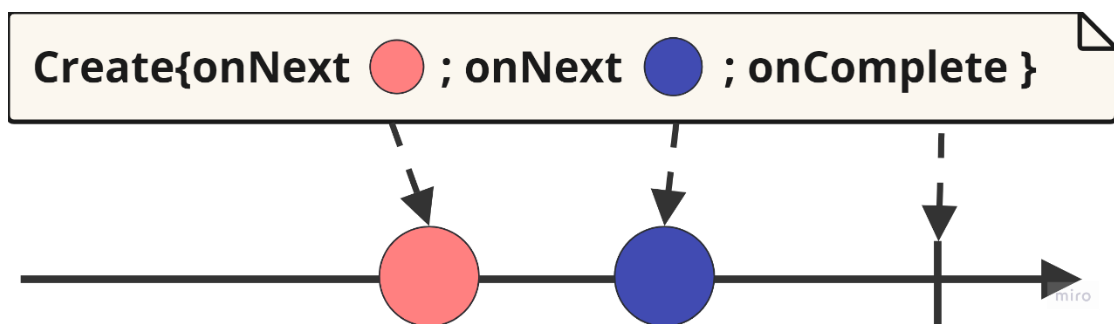


Рисунок 2.6 – Приклад роботи UniRx

Observables змінна може видавати три типи сповіщень:

– onNext висилається, коли з'являється новий фрагмент даних;
– onError висилається, коли виникає помилка;
– onCompleted згенеровано, коли потік даних завершено.

```
Observable.EveryUpdate()
    .Where(_ => Input.GetMouseButtonDown(0))
    .Subscribe(_ => Debug.Log( message: "Mouse clicked"));
```

Рисунок 2.7 – Синтаксичний приклад Observables

Observers підписуються на спостережувані об’єкти, щоб отримувати та реагувати на викиди даних (рис. 2.8). Observer обробляє сповіщення, які надсилає спостережуваний об’єкт (OnNext, OnError, OnCompleted). Метод Subscribe використовується для приєднання спостерігача до спостережуваного об’єкта.

```
IObservable<long> observable = Observable.Interval(TimeSpan.FromSeconds(1));
observable.Subscribe(
    observer: x => Debug.Log( message: "OnNext: " + x),
    ex => Debug.Log( message: "OnError: " + ex),
    () => Debug.Log( message: "OnCompleted")
);
```

Рисунок 2.8 – Синтаксичний приклад Observers

UniRx містить широкий спектр операторів, які можна використовувати для перетворення, фільтрації та комбінування спостережуваних даних. Ці оператори натхненні LINQ і надають потужні інструменти для маніпулювання потоками даних (рис. 2.9).

Загальні оператори:

– оператор Select проектує кожен існуючий елемент спостережуваної послідовності у нову форму;

– оператор Where фільтрує елементи видимої послідовності на основі предиката;

– оператор Merge об’єднує кілька видимих послідовностей в одну видиму послідовність;

– оператор Concat послідовно об’єднує декілька спостережуваних послідовностей в одну.

```
Observable.EveryUpdate()
    .Where(_ => Input.GetKeyDown(KeyCode.Space))
    .Select(_ => "Space key pressed")
    .Subscribe(message => Debug.Log(message));
```

Рисунок 2.9 – Синтаксичний приклад роботи з операторами в UniRx

Планувальники (Schedulers) контролюють контекст, у якому працюють спостережувані. Вони особливо корисні для керування потоками та забезпечення виконання певних операцій у певних потоках, таких як основний потік Unity для оновлення інтерфейсу користувача (рис. 2.10).

```
Observable.Start(() => {
        // Some background task
        return "Task completed";
    })
    .ObserveOnMainThread()
    .Subscribe(result => Debug.Log(result));
```

Рисунок 2.10 – Синтаксичний приклад використовування Планувальників (Schedulers)

Треба зазначити, що приклади наведені вище мають лише інформативний характер, коли приходить час використання даного в фреймворку на реальних проектах, то можна побачити як розгортаются цілі модулі побудовані лише завдяки UniRx.

Фреймворк UniRx має багато плюсів у використанні, але як і кожен інший інструмент, він має свої слабкі сторони.

Таблиця 2.2 – Плюси та мінуси використання UniRx

| **Плюси** | **Мінуси** |
| --- | --- |
| Забезпечує декларативний підхід до обробки асинхронних операцій, що полегшує розуміння та підтримку коду. | Вимагає вивчення нових парадигм, якщо ви не знайомі з реактивними розширеннями. |
| Дозволяє чітко та ефективно керувати складними послідовностями подій та залежностями. | Може призвести до зайвої нагрузки, особливо при неправильному використанні операторів. |
| Дозволяє створювати складну поведінку з простих, багаторазово використовуваних компонентів, підвищуючи модульність коду. | Налагодження реактивних потоків може бути складнішим порівняно з традиційною обробкою подій. |
| Робить код більш читабельним та зручним для супроводу завдяки стислому та інтуїтивно зрозумілому синтаксису. | Інтеграція UniRx з існуючими кодовими базами може вимагати значного рефакторингу. |
| Спрощує керування потоками, гарантуючи, що фонова робота може бути легко оброблена, а результати обробляються в основному потоці. | Інтенсивне використання спостережень та підписок може призвести до збільшення тиску на обробку сміття, якщо ним не керувати належним чином. |

## 2.3 Масштабованість і модульність

Розробка гри в Unity з акцентом на масштабованість та модульність має важливе значення для створення надійних, підтримуваних та розширюваних проектів. Ці принципи допомагають керувати складністю, сприяють повторному використанню коду та полегшують співпрацю між членами

команди. Ось детальний огляд того, як можна досягти масштабованості та модульності у розробці ігор за допомогою Unity.

Масштабованість – це здатність гри справлятися з ростом, незалежно від того, чи це означає додавання нових функцій, роботу з більшою кількістю гравців або оптимізацію продуктивності для ширшого спектру пристроїв.

Масштабована архітектура коду гарантує, що гра може рости і розвиватися без значного рефакторингу [30].

Ключові практики включають:

– впровадження патернів проектування, таких як MVC (Model-View-Controller), MVVM (Model-View-ViewModel) (рис. 2.12) та ECS (Entity-Component-System) допомагає організувати код та керувати складністю. Ці патерни розділяють проблеми, що полегшує розширення або модифікацію окремих частин гри;

– використання фреймворків ін’єкції залежностей, таких як Zenject, допомагає керувати залежностями та сприяє роз’єднаній архітектурі. Це полегшує заміну компонентів, додавання нових функцій або модифікацію існуючих, не впливаючи на інші частини системи;

– дуже важливо органічно використовувати пам’ять пристроя, це потрібно для високої продуктовності та плавності гемплею. Розмір застосунку, також має вплив при запуску гри, бо на початку ініціалізуються всі ресурси, які були в грі, а при великих масштабах, це може принести негативний відгук.

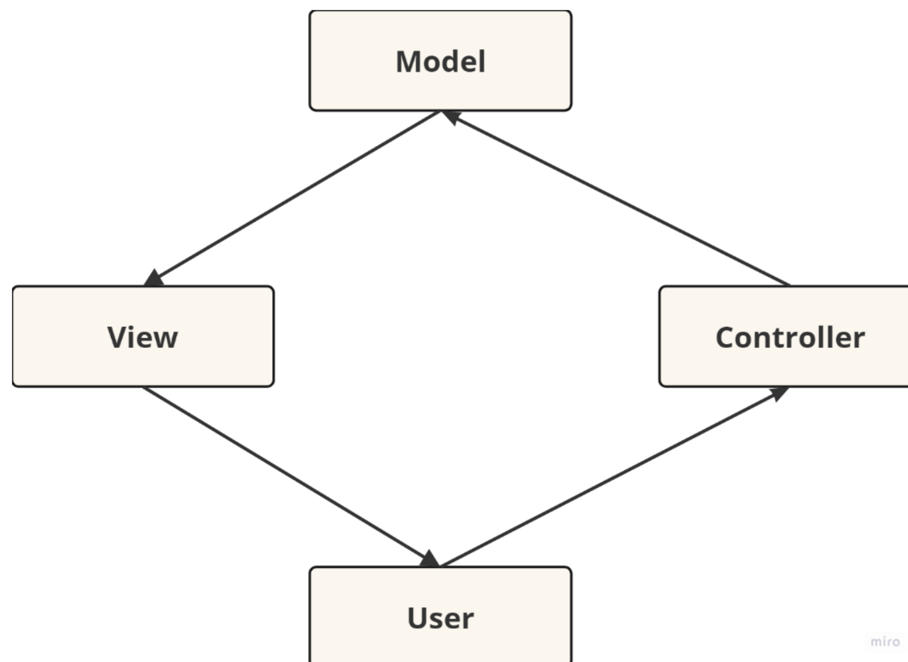


Рисунок 2.12 – Приклад MVC патерна проектування

Ефективне управління активами гарантує, що гра може масштабуватися з точки зору контенту, не стикаючись з проблемами продуктивності чи організації.

Пакети ресурсів Unity дозволяють розробникам динамічно завантажувати ресурси під час виконання, зменшуючи час початкового завантаження та використання пам’яті. Це особливо корисно для великих ігор з великою кількістю ресурсів.

Система адресних даних Unity надає гнучкий спосіб керування динамічним завантаженням контенту, полегшуючи керування та оптимізацію завантаження і вивантаження ресурсів за потреби.

Модульність – це проектування гри таким чином, щоб її компоненти або системи могли розроблятися, тестуватися та підтримуватися незалежно. Такий підхід сприяє багаторазовому використанню та спрощує співпрацю між розробниками.

Архітектура Unity природно підтримує компонентний дизайн, де поведінка інкапсульована у скриптах, прикріплених до GameObjects. Це

сприяє модульності, дозволяючи розробникам змішувати та поєднувати компоненти для створення складної поведінки.

ScriptableObjects надають можливість зберігати дані та конфігурації за межами моноповедінки, сприяючи створенню модульних та багаторазових структур даних.Використовуйте ScriptableObjects для визначення ігрових налаштувань, атрибутів персонажів або конфігурацій рівнів. Це відокремлює дані від логіки, що полегшує керування та налаштування без зміни кодової бази.

# 3 РЕЗУЛЬТАТИ КОМП’ЮТЕРНОГО ПРОГРАМУВАННЯ

## 3.1 Вибір версії та середовища для програмування

Вибір версії рушія Unity для початку створення гри залежить від низки чинників, включно з необхідними функціями, стабільністю та сумісністю з використовуваним обладнанням і платформами.

Для початку варто оцінити функціональні потреби проекту. Переконатись, що обрана версія підтримує всі цільові платформи, для яких треба розробляти гру. Важливо також враховувати питання сумісності. Варто перевірити, щоб використовувані плагіни та зовнішні бібліотеки були сумісні з версією Unity, яку ми обрали. Треба також не забути оцінити, чи відповідає обладнання вимогам обраної версії Unity. Коли справа доходить до стабільності та підтримки, для комерційних проєктів, де важливі довгострокова стабільність і підтримка, найкращим вибором можуть бути LTS-версії (Long Term Support). Ці версії забезпечуються підтримкою та оновленнями безпеки протягом трьох років після їхнього випуску [21].

Тож, версія на якій ми будемо створювати наш мобільний застосунок є 2020.3.48f1. Цю версію було обрано, бо вона є LTS-версією, а це означає що вона буде актуальною на період нашої роботи, а також вона сумісна з більш давніми версіями Android, що допоможе охопити більше пристроїв для тестування та скачування.

Для розробки програмного застосунку було використано середовище розробки JetBrains Rider 2023.2.2.

Підчас розробки гри в Unity вибір інтегрованого середовища розробки (IDE) може суттєво вплинути на продуктивність, якість коду та загальний досвід розробки. JetBrains Rider виділяється як виняткове IDE для розробки ігор в Unity, надаючи повний набір функцій, спеціально адаптованих до потреб розробників ігор. Для розробки гри на Unity було обрано Rider JetBrains.

Rider пропонує глибоку інтеграцію з редактором Unity, що спрощує робочий процес розробки та підвищує продуктивність. При відкриванні проєкту Unity у Rider, IDE автоматично розпізнає структуру та конфігурацію проекту. Ця інтеграція дозволяє запускати і налагоджувати гру безпосередньо з Rider, усуваючи необхідність перемикатися між редактором Unity і IDE. Rider також надає специфічні для Unity перевірки та аналіз коду, які допоможуть дотримуватися найкращих практик та уникати поширених помилок у розробці Unity, таких як неправильні сигнатури методів MonoBehaviour або неефективне використання API.

Однією з найважливіших переваг використання Rider є його потужні можливості редагування коду. Rider пропонує інтелектуальне завершення коду, яке розуміє Unity API та надає контекстно-залежні пропозиції, що зменшує ймовірність помилок та пришвидшує процес кодування. IDE також включає широкий спектр інструментів рефакторингу, таких як: перейменування символів, вилучення методів та зміна сигнатур методів, які допоможуть реструктурувати код без внесення помилок. Крім того, розширені функції навігації Rider, такі як «Перейти до визначення», «Знайти використання» та «Перехід до пов’язаних файлів», дозволяють легко досліджувати та розуміти великі бази коду, що має вирішальне значення у складних ігрових проектах.

Налагодження є невід’ємною частиною розробки ігор, і Rider перевершує в цій сфері завдяки своїм надійним інструментам налагодження. Можна встановлювати точки зупинки, перевіряти змінні та переглядати код безпосередньо в середовищі Unity. Цей безперебійний процес налагодження полегшує виявлення та виправлення проблем, гарантуючи безперебійну роботу вашої гри. Rider також підтримує тестування в ігровому режимі, що дозволяє запускати та налагоджувати тести в ігровому режимі редактора Unity. Ця функція особливо корисна для тестування ігрової логіки та поведінки в реальному ігровому контексті. Крім того, Rider підтримує популярні

фреймворки для тестування .NET, такі як NUnit та MSTest, що спрощує написання та запуск модульних тестів для ваших скриптів Unity.

Rider – це кросплатформенне середовище розробки, доступне для Windows, macOS та Linux. Така гнучкість дозволяє командам розробників працювати на різних операційних системах без шкоди для можливостей інструменту. Незалежно від того, чи ви розробляєте на комп’ютері з Windows, Mac або Linux, Rider надає узгоджене і потужне середовище розробки. Ця крос-платформенна підтримка особливо корисна для розподілених команд або розробників, які надають перевагу різним операційним системам.

## 3.2 Створення проекту на Unity

Тож коли усі процеси налагоджені, треба створити проект в Unity, де буде далі проводитись робота. Зробити це можна, завантаживши офіційну версію Unity Hub з сайту Unity. Далі потрібно встановити потрібну версію (2020.3.48f1) на комп’ютер, потім створити та назвати проєкт (рис 3.1 ).

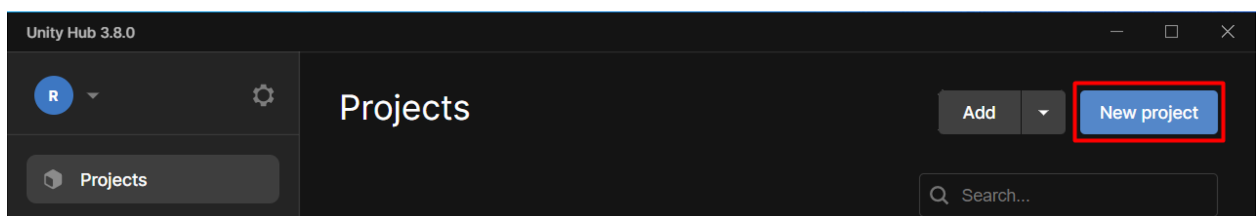


Рисунок 3.1 – Створення проекту в Unity Hub

При створенні проекту в Unity Hub, треба вибрати темплейт проекту, виходячи з того, що саме потрібно від гри.

У Unity Hub темплейт проєкту – це попередньо встановлена конфігурація для створення нового проєкту, що містить певні налаштування, бібліотеки і, можливо, активи або приклади сцен. Темплейти допомагають

швидко почати розробку, орієнтовану на певні цілі або платформи, надаючи базову структуру та налаштування, які відповідають типу проєкту.

Виходячи з того, що обрана ідея гри має 3D оточення та об’єкти, треба обрати із списку темплейтів саме 3D.

3D (Built-In Render Pipeline) – темплейт для створення 3D-ігор з використанням вбудованої системи рендерингу. Це стандартний вибір для багатьох початківців-розробників ігор, оскільки він пропонує базову і гнучку відправну точку для створення тривимірних проектів (рис. 3.2). Також, саме в цьому меню, треба вибрати версію двигуна, якій буде використовуватись. Задаємо назву нашому проєкту, шлях, де будуть знаходитись файли, а ще потрібно вказати назву організації, яка буде використовувати двигун. Організацію можна створити в обліковому записі на самій платформі Unity, це потрібно для подальшого використання сервісів платформи і двигуна, тому не вийде пропустити цей пункт.

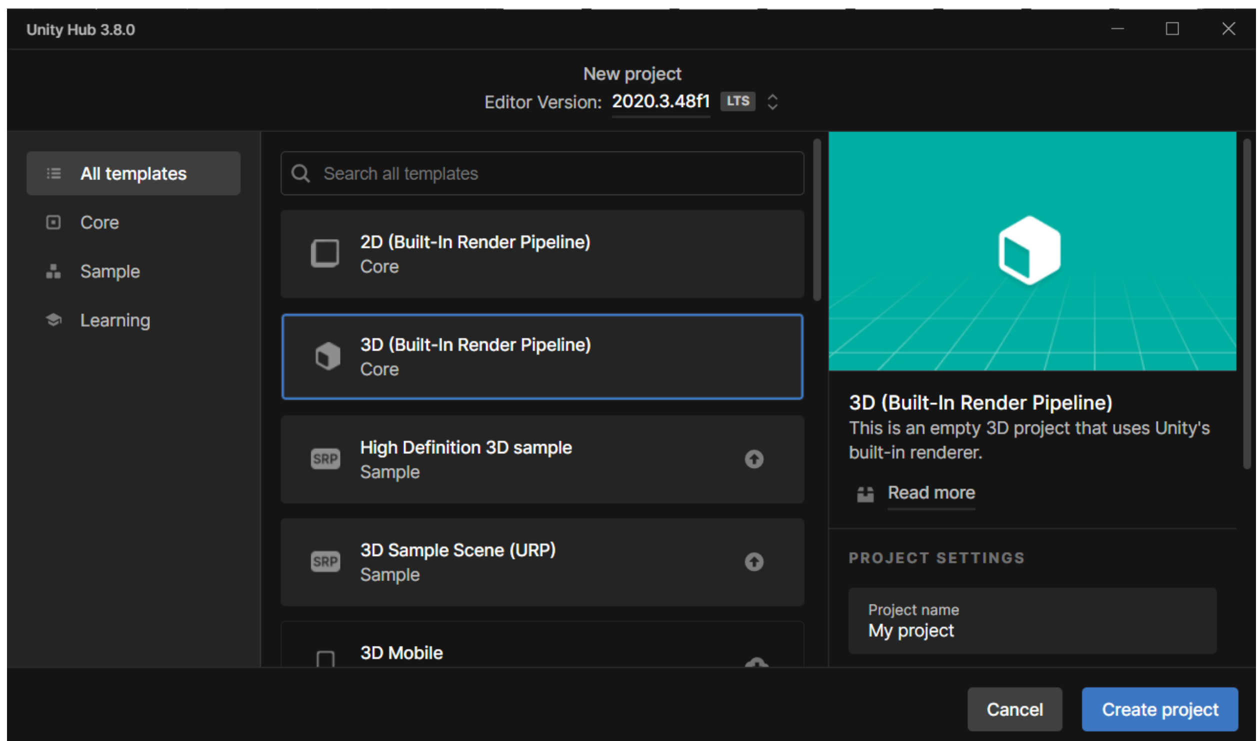


Рисунок 3.2 – Вибір темплейту та створення проекту

Після створення проекту відкриється сам двигун Unity, де ми зможемо побачити багато інструментів, допоміжних вікон та багато іншого, це все ми будемо використовувати поступово (рис 3.3).

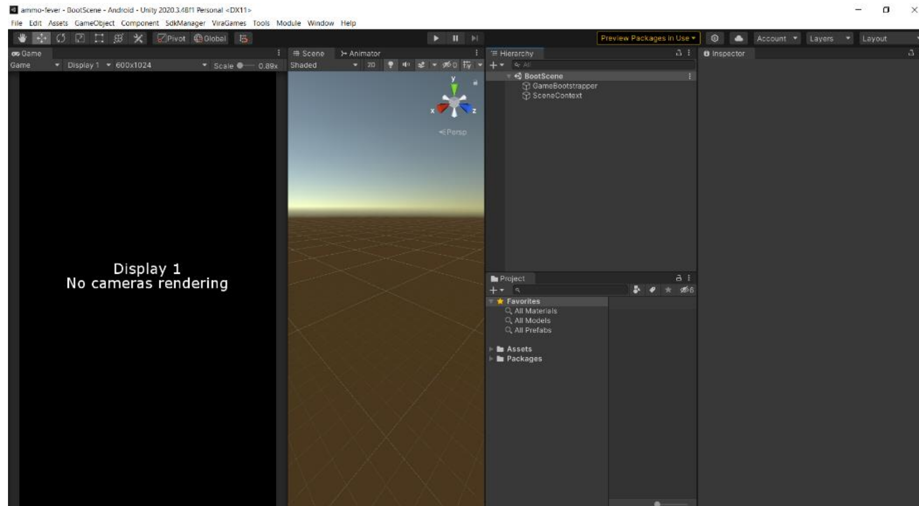


Рисунок 3.3 – Приклад візуальної частини двигуна Unity

Також треба зазначити, що проєкт буде розроблятись для платформи Android, тому потрібно змінити налаштування платформи в самому Unity.

Зробити це можна перейшовши по вкладці на верхній панелі інструментів Unity. Загальний путь File – Build Settings – Обрати у списку платформ, платформу Android – Натиснути кнопку Switch Platform (рис. 3.4).

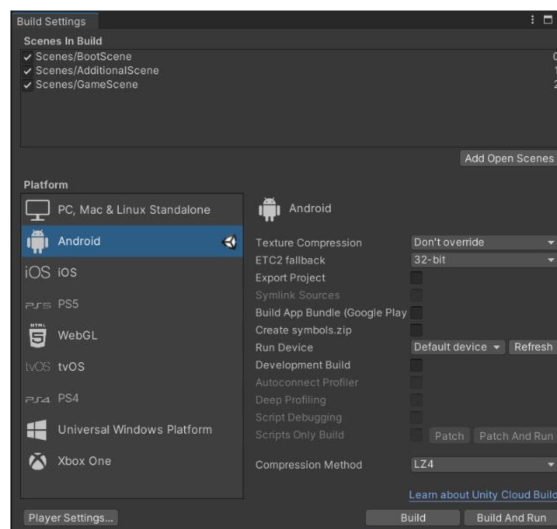


Рисунок 3.4 – Результат переходу на платформу Android

## 3.3 Програмування модульної системи

Для початку нам потрібна точка входу в нашу систему, зазвичай це називається Boostrap, тобто початкова підготовка системи до роботи.

На етапі Bootstrap зазвичай відбувається конфігурація застосунку, включно з читанням і застосуванням налаштувань із конфігураційних файлів або змінних оточення. Це дає змогу легко змінювати поведінку застосунку без необхідності зміни коду. Bootstrap допомагає коректно керувати порядком завантаження цих компонентів, гарантуючи, що всі залежності задоволені до початку роботи програми.

Говорячи про точку входу в проєкті, треба зазначити, що ініціалізація залежностей проводиться завдяки використанню Zenject для прокидування потрібних модулів до контейнера.

Підготувавши на початковії сцені 2 об’єкти, які роблять першу ініціалізацію модулів перед тим, як переходити на головну ігрову сцену(рис. 3.5). Також, треба взяти до уваги, що нам не треба використовувати на початковії сцені камеру або світло від двигуна. Це тому, що початкова сцена не буде показуватись користувачеві, тому не має потреби використовувати додаткові ресурси для цього.

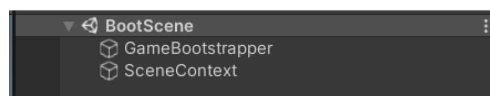


Рисунок 3.5 – Підготовка точки входу в систему гри

Сам код бустрапера складається з метода ін’єкції залежностей (рис 3.6), для подальшого їх використання, а також методу переходу на головну ігрову сцену (рис 3.7). Після переходу на сцену, відбувається івент вдалого званатажування ресурсів, тому по завершенню переходу, вже на ігровії сцені відбудеться ініціалізація потрібних нам залежностей та об’єктів.

```
[Inject]
public void Construct(ISceneLoader sceneLoader, ILoadingCurtain loadingCurtain, IStaticDataService staticDataService, IGameFactory gameFactory,
    IInputService inputService, IEventObserver eventObserver, ISaveLoadService saveLoadService, IPoolService poolService, IWindowService windowService,
    IMoneyViewer moneyViewer, IMergeService mergeService, IAmmoStack ammoStack, ISkillBoostFactory skillBoostFactory, IFXViewer fxViewer)
{
    _fxViewer = fxViewer;
    _skillBoostFactory = skillBoostFactory;
    _ammoStack = ammoStack;
    _mergeService = mergeService;
    _moneyViewer = moneyViewer;
    _windowService = windowService;
    _poolService = poolService;
    _saveLoadService = saveLoadService;
    _eventObserver = eventObserver;
    _staticDataService = staticDataService;
    _inputService = inputService;
    _gameFactory = gameFactory;
    _loadingCurtain = loadingCurtain;
    _sceneLoader = sceneLoader;
}
```

Рисунок 3.6 – Отримання залежностей використовуючи фреймворк Zenject

```
private void OnEnable()
{
    DOTween.SetTweensCapacity(tweenersCapacity: 600, sequencesCapacity: 320);
    QualitySettings.vSyncCount = 0;

    switch (gameType)
    {
        case GameType.Game:
            _sceneLoader.Load(name: ScenesKeys.AdditionalScene, OnLevelLoad);
            break;
        case GameType.Test:
            _sceneLoader.Load(name: ScenesKeys.TestScene, OnTestModeLoad);
            break;
        default:
            throw new ArgumentOutOfRangeException();
    }
}
```

Рисунок 3.7 – Метод загрузки головної ігрової сцени

Далі, коли ми перейшли на головну сцену нашої гри, яку заздалегідь підготували, а саме: зробили налаштування ігрової камери, коригування світла на сцені, виставили потрібне нам оточення, та зробили префаби об’єктів для повторного використання (рис 3.8). Далі ми побачимо вже ініціалізовані моделі та системи, які почали ігровий процеси: створення ігрових об’єктів, працююча ігрове UI вікно та автоматизовані процеси постачання патронів для наших пушок. А також, для початківців показується UI віно – ознайомлення з геймплеєм.

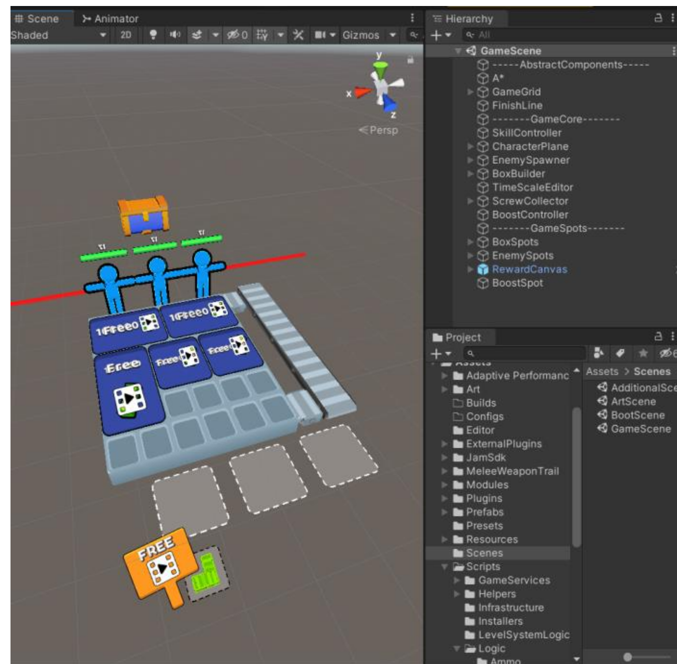


Рисунок 3.8 – Заздалегіть підготовлена головна ігрова сцена

Після переходу на головну ігрову сцену, ми вже можемо отримати залежності, які витягнули з контейнера Zenject. На самій сцені знаходиться Level Controller – клас, який відповідає за створення потрібного нам рівня та прокидування залежностей далі, тим компонентам, яким це потрібно (рис 3.9).

```
public void Construct(ISceneLoader sceneLoader, ILoadingCurtain loadingCurtain, IStaticDataService staticDataService, IGameFactory gameFactory,
    IInputService inputService, IEventObserver eventObserver, ISaveLoadService saveLoadService, IPoolService poolService, IWindowService windowService,
    IMoneyViewer moneyViewer, IMergeService mergeService, IAmmoStack ammoStack, ISkillBoostFactory skillBoostFactory, IFXViewer fxViewer)
{
    _fxViewer = fxViewer;
    _skillBoostFactory = skillBoostFactory;
    _ammoStack = ammoStack;
    _mergeService = mergeService;
    _windowService = windowService;
    _moneyViewer = moneyViewer;
    _poolService = poolService;
    _saveLoadService = saveLoadService;
    _eventObserver = eventObserver;
    _staticDataService = staticDataService;
    _inputService = inputService;
    _gameFactory = gameFactory;
    _loadingCurtain = loadingCurtain;
    _sceneLoader = sceneLoader;

    _poolContainer = GetComponentInChildren<PoolContainer>();
    SpawnStaticPools();

    _eventObserver.OnLevelStart += LoadLevel;

    var isFirstGame = _saveLoadService.LoadStructData(PlayerPrefsKeys.FirstGame, 0);
    if (isFirstGame == 0)
    {
        _eventObserver.LevelStart(0);
    }
    else
    {
        Scene scene = SceneManager.GetSceneByName(ScenesKeys.AdditionalScene);
        _windowService.Open(WindowID.MainMenuScreen , scene);
    }

    _loadingCurtain.Hide();
}
```

Рисунок 3.9 – Конструктор класу Level Controller

На цьому етапі відбувається перевірка, чи заходив користувач у цю гру раніше, якщо ні, тоді ми завантажуємо перший рівень з туторіалом, який допоможе зрозуміти гравцеві, що потрібно робити у грі. Також, ми вже можемо побачити використання певних модулів та патернів, а саме патерну Спостерігача (Observer Pattern) та модуля Save/Load (завантаження та зберігання даних користувача). Також, на момент ініціалізації, ми можемо побачити сервіс, який називається WindowService – інструмент, який має відповідну єдину задачу: завантаження UI (user interface) вікон (рис. 3.10).

```
1 usage  Ruslan
private void LoadLevel(int levelId)
{
    _selectedLevel = levelId;

    var levelNumber :int = _saveLoadService.LoadStructData(PlayerPrefsKeys.LevelNumberKey, defaultValue: 1);
    var cycleNumber :int = _saveLoadService.LoadStructData( key: PlayerPrefsKeys.CycleNumberKey + levelId, defaultValue: 0);

    ViraToolKit.Instance.analyticsManager.SendLevelStart(levelNumber ,cycleNumber , levelIndex: levelId + 1);

    levelNumber++;
    _saveLoadService.SaveStructData(PlayerPrefsKeys.LevelNumberKey, levelNumber);
    cycleNumber++;
    _saveLoadService.SaveStructData(PlayerPrefsKeys.CycleNumberKey + levelId, cycleNumber);

    _sceneLoader.Load( name: ScenesKeys.GameScene, OnLevelLoad, LoadSceneMode.Additive);
}
```

Рисунок 3.10 – Метод завантаження ігрового рівня

Після завантаження рівня, ми маємо віднайти всі потрібні нам компоненти на рівні, які потребують залежностей. На цьому моменті, користувач бачить тільки екран завантаження, але «під капотом» відбувається багато процесів, які непомітні, але дуже важливі для коректної роботи системи гри. Коли рівень, буде завантажено, ми почнемо пошук потрібних нам компонентів та підсистем для продовження гри. Після прокидування залежностей для компонентів, почнуться перші секунди гемплею (рис. 3.11).

```
private void OnLevelLoad()
{
    Scene scene = SceneManager.GetSceneByName(ScenesKeys.AdditionalScene);
    _windowService.Open(WindowID.InGameScreen, scene);

    var isFirstGame = _saveLoadService.LoadStructData(PlayerPrefsKeys.FirstGame, 0);
    if (isFirstGame == 0)
    {
        tutorScreen.gameObject.SetActive(true);
        tutorScreen.InitTutorScreen(_saveLoadService, _eventObserver, _mergeService);
    }

    var timeEditor = FindObjectOfType<TimeScaleEditor>();
    timeEditor.Construct(_eventObserver);

    _characterPlane = FindObjectOfType<CharacterPlane>();
    _characterPlane.Construct(_ammoStack, _poolService, _gameFactory, _saveLoadService, _staticDataService, _eventObserver, _windowService, _fxViewer);

    SetEnvironment();

    AstarPath astarPath = FindObjectOfType<AstarPath>();
    astarPath.Scan();

    var screwCollector = FindObjectOfType<ScrewCollector>();
    screwCollector.Construct(_eventObserver, _saveLoadService, _staticDataService, _poolService);

    var skillController = FindObjectOfType<SkillController>();
    skillController.Construct(_skillBoostFactory);

    _loadingCurtain.Hide();
}
```

Рисунок 3.11 – Метод пошуку та ініціалізації компонентів на ігровому рівні

Після усіх процесів ініціалізації та прокидування ми маємо гарний початковий функціонал для користувача. Треба зауважити, що перший запуск гри та всі інші будуть відрізнятися, бо при першому запуску ми маємо за ціль зацікавити користувача нашим застосунком. Далі буде продемонстровано варіант запуску гри, коли користувач вже неодноразово заходив у гру (рис. 3.12).

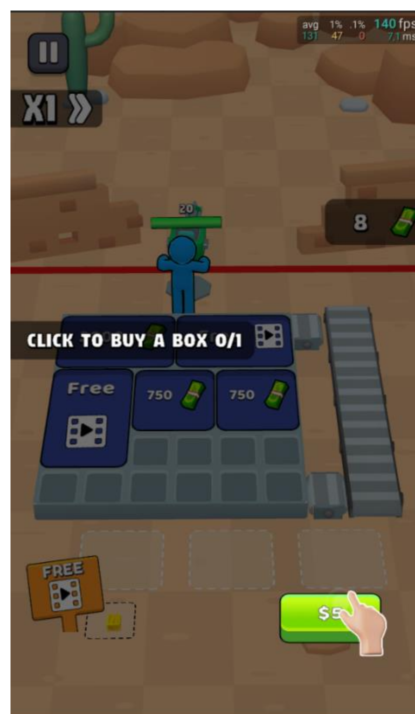


Рисунок 3.12 – Перший запуск гри на Android

Сама ідея гри полягає у тому, аби збирати і комбінувати ящики з патронами по принципу гри 2048, тобто комбінувати однакові ящики та виставляти їх на ігрову сітку для того, аби ресурси постачалися по конвеєру. Конвеєр буде перевозити ресурси та постачати на башні-турелі для захисту від ворогів, які будуть наносити урон, коли зайдуть за червону лінію. Гра виконана у жанрі гіперказульні, та має всі шанси потрапляння до чартів через свою динамічність да зацікавленість (рис. 3.13).

Рисунок 3.13 – Приклад геймплею

Також у грі було використано бібліотеку Dotween для реалізації динамічних анімації.

DOTween – це швидка, ефективна та повнофункціональна бібліотека анімації та синхронізації для Unity. Розроблена компанією Demigiant, вона використовується для створення плавної анімації, складних послідовностей та часових операцій. DOTween особливо відома своєю простотою використання та ефективністю, що робить її поширеним вибором серед розробників Unity для керування твінами (інтерполяціями) та анімацією в іграх та інтерактивних застосунках (рис 3.14).

```
Frequently called  1 usage  Ruslan
private void CompleteAction(int complete)
{
    _completeSequence = DOTween.Sequence();
    Vector3 originalPosition = tutorTaskObj.localPosition;
    Tweener jumpUp = tutorTaskObj.DOLocalMoveY( endValue: originalPosition.y + 10f,  duration: 0.2f).SetEase(Ease.OutQuad);
    Tweener jumpDown = tutorTaskObj.DOLocalMoveY( endValue: originalPosition.y,  duration: 0.2f).SetEase(Ease.InQuad);

    Tweener colorGreen = TutorText.DOColor(Color.green ,  duration: 0.1f);
    TutorText.text = tutorTaskText + $" {complete}/{completeValue}";

    _completeSequence.Append(jumpUp);
    _completeSequence.Join(colorGreen);
    _completeSequence.Append(jumpDown);

    _completeSequence.OnComplete(() =>
    {
        _completeSequence.Kill();
        _completeSequence = null;
        gameObject.SetActive(false);
    });
}
```

Рисунок 3.14 – Приклад використання бібліотеки Dotween у проекті

У цьому прикладі використовується складна функція бібліотеки Dotween, а саме Dotween.Sequence(). Sequence у DOTween – це контейнер для кількох твінів, які треба відтворити у певному порядку. Це дозволяє створювати складні анімації, де кілька анімацій можуть відтворюватися одна за одною, або навіть перекриватися з точною синхронізацією.

Ключові можливості DOTween.Sequence():

– ланцюгові анімації дозволяють з’єднати кілька анімацій у певному порядку;

– паралельні анімації дозволяють відтворювати кілька анімацій одночасно;

– вставки надають можливість вставляти анімації в певні точки послідовності;

– зворотні виклики підтримують зворотні виклики в різних точках

послідовності для більшого контролю;

– зациклення та затримки підтримують зациклення послідовності та додавання затримок між анімаціями.

Після проработки основного геймплею з використанням модулів та сервісів для ефективної роботи системи, треба зауважити, що у нашій грі, також присутній UI для більшої зацікавленості користувача. Він демонструється як у геймплею так і перед, аби дати можливість гравцю подивитися більше контенту. Зараз ми розглянемо, що відбувається перед запуском самого ігрового рівня, якщо в гру заходили не перший раз.

При типовому запуску гри (не першому), ми можемо спостерігати зібраний UI головного меню, який дає гравцю змогу обрати рівень, вибрати пушки та бонуси, які будуть на рівні (рис. 3.15). Також у грі реалізована система щоденної винагороди, яка постійно оновлює час, коли гравець зможе забрати ресурси (рис. 3.16 ) .

а) б) в)

Рисунки 3.15 – Реалізація UI головного меню у грі

а) головне меню гри; б) вікно вибору пушок; в) вікно щоденних винагород

```
public void InitDailyRewards(ISaveLoadService saveLoadService, Windows.MainMenu mainMenu)
{
    _mainMenu = mainMenu;
    _saveLoadService = saveLoadService;

    _currentStreak = _saveLoadService.LoadStructData(PlayerPrefsKeys.DailyStreakKey, defaultValue: 0);
    var timeData:string = _saveLoadService.LoadStructData(PlayerPrefsKeys.LastClaimedTimeKey, defaultValue: "");

    if (!string.IsNullOrEmpty(timeData))
    {
        _lastClaimTime = DateTime.Parse(timeData);
    }

    for (int i = 0; i < rewards.Count; i++)
    {
        var reward = rewards[i];
        reward.InitReward(disableColor, ClaimRewardAction);

        if (i < _currentStreak)
        {
            reward.DisableShopItem();
            continue;
        }

        reward.SetItemToDefault();
    }

    StartRewardCoroutine();
}
```

Рисунок 3.16 – Приклад ініціалізації щоденної винагород

# ВИСНОВКИ

У рамках кваліфікаційної роботи були проведені дослідження щодо розробки адаптивної та розширюваної системи з використанням фреймворків та бібліотек для ефективного програмування модулів та сервісів.

Практичні рекомендації із проведеної роботи полягають в оптимізації ігрових процесів, оптимальної роботи із статичною датої та зменшення розміру файлів для досягнення меншого розміру застосунку.

Новизна роботи полягає у використанні декількох фрейморків одночасно, що дало позитивний результат при розробці системи.

Практична значущість роботи полягає у розроблені застосунку на Unity, яку можно адаптувати під різні платформи (Windows, IOS , Android, PS4).

Перспективи роботи полягають у використанні нових фреймворків для поліпшення роботи із ресурсами застосунку та розширення взаємозамінних модулів при портуванні гри на різні платформи.

## ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ


1. Владстон Феррейра Фило ( 2018). Теоретичний мінімум з computer science. все що потрібно програмісту і розробнику, С. 167-200.

2. Joe Hocking (2015). Unity in Action, C. 47-60.

3. Harrison Ferrone (2022). Learning C# by Developing Games with Unity, C. 16-25.

4. Paris Buttfield-Addison, Jon Manning, and Tim Nugent (2019). Unity Game Development Cookbook, C. 69-78.

5. Joseph Albahari and Ben Albahari (2021). C# 9.0 in a Nutshell, C. 576-610.

6. Alan Thorn (2014). Pro Unity Game Development with C# by Alan Thorn, C. 87-114.

7. Феррейра Філо Владстон, Піктет Мо (2022). Теоретичний мінімум за Computer Science. Мережі, криптографія та data science, С. 140-200.

8. Unity (2021). Level up your programming with game programming patterns, C. 88-95.

9. Unity (2023). Ultimate guide to profiling Unity games, C. 60 – 75.

10. Alexander Shvets (2021). Dive Into Design Patterns, C. 140-250.

11. Andrew Troelsen and Philip Japikse (2021), C. 271-297.

12. Joe Hocking (2021). Unity in Action, Third Edition: Multiplatform Game Development in C# with Unity 2020, C. 70 – 113.

13. Nicolas Alejandro Borromeo, Juan Gabriel Gomila Salas (2024). Hands-On Unity Game Development: Unlock the power of Unity 2023 and build your dream game, C. 109-180.

14. Patrick Felicia (2015). Unity From Zero to Proficiency (Beginner): A step-by-step guide to coding your first game with Unity in C#, C. 190- 304.

15. Patrick Felicia (2016). Unity From Zero to Proficiency (Intermediate): A step-by-step guide to coding your first FPS in C# with Unity, C. 100 – 254.

16. Stephen Cleary (2019). Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming, C. 117- 200.

17. Pablo Barrón Ballesteros (2024). Advanced C# programming for video games: design patterns for Unity and Godot, C. 19 – 130.

18. Faruk Yolcu (2023). Unity DOTween Basics-Part 1. URL: https://medium.com/@farukyolcu/unity-dotween-basics-part-1-93df8504eaba (дата звернення 28.04.2024)

19. Patrick Felicia (2016). Unity From Zero to Proficiency (Advanced): Create multiplayer games and procedural levels, and boost game performances: a step-by-step guide, C. 100-244.

20. C# language documentation. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення 20.04.2024)

21. Unity Documentation. URL: https://docs.unity3d.com/Manual/index.html (дата звернення 25.04.2024).

22. RB Whitaker (2022). The C# Player's Guide(5th Edition),C.140-340.

23. Anna Skoulikari (2023). Learning Git: A Hands-On and Visual Guide to the Basics of Git , C. 90 – 140.

24. Richard Silverman (2013). Git Pocket Guide: A Working Introduction, C. 40 – 200.

25. Zenject Documentation. URL: https://github.com/modesttree/Zenject?tab=readme-ov-file (дата звернення 18.04.2024)

26. Wladston Ferreira Filho (2017). Computer Science Distilled: Learn the Art of Solving Computational Problems, C. 100-145.

27. Mark J. Price (2023). C# 12 and .NET 8 – Modern Cross-Platform Development Fundamentals: Start building websites and services with ASP.NET Core 8, Blazor, and EF Core 8, C. 230 – 290.

28. Joel Murach, Mary Delamater (2022). Murach's Asp.net Core Mvc, C. 40-120.

29. Jon Skeet (2019). C# in Depth: Fourth Edition, C. 20 – 220.

30. Robert Nystrom (2014). Game Programming Patterns, C. 267-321.