БАКАЛАВРСЬКА РОБОТА

БР.ІП – 44.00.00.000 ПЗ

Група ІП-22-2

Остапчук Максим

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Остапчук Максим Вікторович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Проектування та розробка ігрового застосунку в жанрі 2D-платформер з використанням об'єктно-орієнтованих патернів
(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____Остапчук М.В._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник ____Григорчук Любомир Іванович, доцент, к.т.н_____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Остапчук Максиму Вікторовичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Проектування та розробка ігрового застосунку в жанрі 2D-платформер з використанням об'єктно-орієнтованих патернів "

керівник проекту (роботи) доц., к.т.н., Григорчук Л.І.

затверджені наказом вищого навчального закладу від “ 21 ” квітня 2026 р. № 179/7

2. Строк подання студентом проекту (роботи) 10 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

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

1. Аналіз вимог до програмного забезпечення
2. Аналіз вимог і постановка задачі
3. Проектування системи
4. Реалізація проекту
5. Тестування та впровадження

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

1 Вік та стать гравців. (рис. 1.1, ст. 11)
2 Загальна архітектура гри. (рис. 1.2, ст. 13)
3 Стратегічний шаблон - приклад шахів. (рис. 2.1, ст. 24)
4 Шаблон спостерігача - приклад футбольного менеджера (рис. 2.2, ст. 26)
5 Головний інтерфейс Unity Hub (рис. 3.1, ст. 44)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	17.03.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	21.04.2026	виконано
3	Побудова моделі або алгоритму власного рішення	01.05.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	21.05.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	10.06.2026	виконано

Студент _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 79 сторінок, 29 рисунків, 1 таблицю, список використаних джерел із 40 найменувань.

Метою роботи є дослідження принципів проєктування та розробки ігрових застосунків у жанрі 2D-платформер із використанням об'єктно-орієнтованих патернів.

Об'єкт дослідження: процес розробки ігрових застосунків.

Предмет дослідження: методи, технології та об'єктно-орієнтовані патерни проєктування, що застосовуються для створення ігрових застосунків.

Результати дослідження: проведено аналіз особливостей розробки ігор жанру 2D-платформер, досліджено принципи об'єктно-орієнтованого програмування та патерни проєктування, розглянуто сучасні технології створення ігор, а також проаналізовано підходи до реалізації і тестування.

У першому розділі розглянуто теоретичні основи розробки ігрових застосунків жанру 2D-платформер, принципи об'єктно-орієнтованого програмування та класифікацію патернів проєктування.

У другому розділі досліджено методи та інструменти розробки, виконано аналіз архітектурних рішень, патернів проєктування та сучасних технологій, що використовуються під час створення ігрових застосунків.

У третьому розділі розглянуто процес проєктування архітектури гри, реалізацію основних функціональних модулів із використанням об'єктно-орієнтованих патернів, а також результати тестування застосунку.

Висновок: використання об'єктно-орієнтованих патернів проєктування під час створення 2D-платформерів забезпечує підвищення модульності, повторного використання програмного коду, спрощує супровід ПЗ.

КЛЮЧОВІ СЛОВА: 2D-ПЛАТФОРМЕР, ІГРОВИЙ ЗАСТОСУНОК, ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ, ПАТЕРНИ ПРОЄКТУВАННЯ, GAME DEVELOPMENT, UNITY, C#, ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

ANNOTATION

The bachelor's thesis contains of 79 pages, 29 figures, 1 table, and a reference list containing 40 sources.

The aim of the thesis is to study the principles of designing and developing 2D platformer game applications using object-oriented design patterns and to analyze approaches for building flexible, scalable, and maintainable software architecture.

Research object: the process of developing 2D platformer game applications.

Research subject: methods, technologies, and object-oriented design patterns used in game application development.

Research results: the study analyzes the characteristics of 2D platformer game development, investigates the principles of object-oriented programming and software design patterns, reviews modern game development technologies, and examines approaches to software implementation and testing.

The first chapter presents the theoretical foundations of 2D platformer game development, object-oriented programming principles, and the classification of software design patterns.

The second chapter analyzes development methods and tools, architectural approaches, design patterns, and modern technologies used for creating game applications.

The third chapter describes the design of the game architecture, implementation of the main functional modules using object-oriented design patterns, as well as testing procedures and evaluation of the developed application.

Conclusion: the application of object-oriented design patterns in 2D platformer game development improves software modularity, code reusability, maintainability, and scalability, contributing to the creation of high-quality and efficient game applications.

KEY WORDS: 2D PLATFORMER, GAME APPLICATION, OBJECT-ORIENTED PROGRAMMING, DESIGN PATTERNS, GAME DEVELOPMENT, UNITY, C#, SOFTWARE DESIGN.

ЗМІСТ

ВСТУП 7

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ПРОЄКТУВАННЯ 2D-ПЛАТФОРМЕРІВ ТА ОБ'ЄКТНО-ОРІЄНТОВАНИХ ПАТЕРНІВ 9

1.1. Основи розробки ігрових застосунків жанру 2D-платформер 9

1.2. Принципи об'єктно-орієнтованого програмування та патерни проєктування 13

1.3. Класифікація та застосування поведінкових, структурних і породжувальних патернів у розробці ігор 16

1.4. Висновок по розділу 19

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ РОЗРОБКИ ІГРОВОГО ЗАСТОСУНКУ 20

2.1. Аналіз патернів проєктування для реалізації ігрової логіки 20

2.2. Архітектурні рішення та моделювання компонентів гри 27

2.3. Програмні засоби та технології розробки 2D-платформера 35

2.4. Висновок по розділу 38

РОЗДІЛ 3. ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ІГРОВОГО ЗАСТОСУНКУ 39

3.1. Проєктування архітектури та функціональних модулів гри 39

3.2. Реалізація ігрового застосунку з використанням об'єктно-орієнтованих патернів 41

3.3. Тестування, оцінювання результатів та аналіз ефективності реалізованого рішення 60

3.4. Висновок ро розділу 65

ВИСНОВКИ 66

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА 68

ВСТУП

Індустрія комп'ютерних ігор є однією з найдинамічніших галузей сучасної інформаційної індустрії, що постійно розвивається завдяки появі нових технологій, програмних платформ і підходів до розробки програмного забезпечення. Значна увага приділяється створенню якісних ігрових застосунків, які поєднують привабливу графіку, інтуїтивний користувацький інтерфейс, високу продуктивність та можливість подальшого розширення функціональних можливостей. Одним із популярних жанрів залишаються 2D-платформери, які завдяки простоті реалізації, широким можливостям творчого дизайну та невисоким системним вимогам активно використовуються як у комерційній розробці, так і в навчальних проєктах.

Актуальність теми зумовлена необхідністю створення ефективних методів проєктування ігрових застосунків, які дозволяють мінімізувати складність програмного коду та підвищити якість програмного продукту. Використання об'єктно-орієнтованих патернів під час розробки 2D-платформера дає можливість структурувати архітектуру застосунку, оптимізувати взаємодію між його компонентами та спростити подальшу модернізацію гри. Крім того, сучасні ігрові рушії підтримують реалізацію шаблонів проєктування, що робить їх невід'ємною складовою професійної розробки програмного забезпечення.

Метою роботи є проєктування та розробка ігрового застосунку в жанрі 2D-платформер із використанням об'єктно-орієнтованих патернів для забезпечення гнучкої архітектури, високої продуктивності та можливості подальшого розширення функціональних можливостей гри.

Для досягнення поставленої мети необхідно вирішити такі завдання: проаналізувати теоретичні основи об'єктно-орієнтованого програмування та шаблонів проєктування; дослідити класифікацію породжувальних, структурних і поведінкових патернів та особливості їх використання у розробці програмного забезпечення; виконати аналіз сучасних методів і технологій створення 2D-ігор; обґрунтувати вибір програмних засобів і технологій розробки; спроєктувати

архітектуру ігрового застосунку із використанням об'єктно-орієнтованих патернів; реалізувати основні програмні модулі гри;

Об'єктом дослідження є процес проєктування та розробки ігрових застосунків жанру 2D-платформер із використанням сучасних принципів об'єктно-орієнтованого програмування.

Предметом дослідження є методи, програмні засоби та об'єктно-орієнтовані патерни проєктування, що застосовуються під час створення архітектури ігрових застосунків, а також способи реалізації основних ігрових механік, взаємодії між програмними компонентами та забезпечення ефективності роботи програмного забезпечення.

Методи дослідження включають аналіз сучасної наукової літератури та технічної документації, порівняльний аналіз існуючих підходів до розробки 2D-ігор, застосування принципів об'єктно-орієнтованого проєктування, моделювання архітектури програмної системи, програмну реалізацію окремих компонентів гри, експериментальне тестування та аналіз отриманих результатів.

У результаті виконання роботи буде розроблено ігровий застосунок у жанрі 2D-платформер, архітектура якого побудована із застосуванням об'єктно-орієнтованих патернів проєктування. Реалізоване програмне рішення забезпечуватиме ефективне керування ігровими об'єктами, обробку подій, взаємодію персонажа з ігровим середовищем, підтримку масштабування функціональних можливостей та спрощення подальшого супроводу програмного коду. Очікується, що використання шаблонів проєктування дозволить підвищити якість програмного забезпечення, зменшити складність архітектури та забезпечити можливість швидкого розширення ігрового функціоналу.

Бакалаврська робота містить 79 сторінок, 29 рисунків, 1 таблицю, три розділи, висновки до кожного розділу, загальні висновки та список використаних джерел із 40 найменувань

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ПРОЄКТУВАННЯ 2D-ПЛАТФОРМЕРІВ ТА ОБ'ЄКТНО-ОРІЄНТОВАНИХ ПАТЕРНІВ

1.1. Основи розробки ігрових застосунків жанру 2D-платформер

Сьогодні індустрія відеоігор є однією з найбільших галузей, і вона продовжує дуже швидко розвиватися. Вона забезпечує багато робочих місць для людей з різними талантами та щороку виробляє велику кількість відеоігор, що призводить до значних доходів для компаній , які їх створюють. У ці ігри грають мільйони людей по всьому світу на різних платформах, включаючи Microsoft Xbox One, Sony Playstation 4, Nintendo Switch, різні мобільні платформи та, звичайно ж, найбільші та найкращі ПК.

З 2012 року дохід світового ринку ігор зріс більш ніж на 30% завдяки різним мобільним платформам, зайнявши 34% його частки ринку у 2017 році. Завдяки постійному зростанню, компанії-розробники програмного забезпечення почали створювати більше додатків для розробки ігор, і наразі існує багато різних інструментів, які дозволяють одній людині або дуже невеликій команді людей успішно розробляти та продавати відеоігри, що ще більше прискорює зростання ринку цієї галузі.

В ігри грають люди різного віку та статі, як показано на діаграмах, наведених на рисунку 1.1. Цей факт забезпечує простий спосіб зв’язку між батьками та дітьми або різними людьми по всьому світу. Беручи до уваги цю інформацію , легко зрозуміти, чому розробка ігор є гарною галуззю для роботи.

Сьогодні об'єктно-орієнтована парадигма стала нормою. Однак, з постійним збільшенням розміру та складності програмних систем, з'явилася концепція шаблонів проектування для покращення архітектур програмного забезпечення [1]. І сьогодні шаблони проектування пропонують загальні рішення для поширених проблем у розробці програмного забезпечення [2]. Шаблон проектування описує проблему проектування, рішення та наслідки [3].

Останнім часом індустрія мобільних ігор переживає величезне зростання

Як наслідок, розробка нових ігор може призвести до повторюваних подібних проблем дизайну. Таким чином, шаблони проектування можна використовувати для оптимізації дизайну та реалізації ігор для мобільних пристроїв [4]. Наприклад, шаблони проектування можуть надавати можливості для створення нової дошки в настільній грі, для зміни розмірів дошки, для підтримки та збереження стану гри, для обчислення наступного ходу, для візуалізації екрана дисплея та для управління сховищами записів. Вирішення цих проблем за допомогою першопринципів, шляхом дублювання кодів та рішень може бути недоцільним, що призводить до розробки монолітного, негнучкого, складного в обслуговуванні, неефективного програмного забезпечення з небажаними функціями [5].

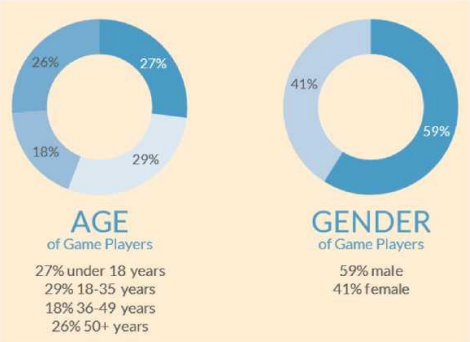


Рисунок 1.1 - Вік та стать гравців

Хоча мобільні ігри можна розробляти з використанням різних технологій, таких як мобільна платформа Windows, Flash, Python та інших, було розглянуто платформу Java 2 Micro Edition (J2ME). J2ME є домінуючою платформою в розробці ігор для мобільних пристроїв. У цій роботі шаблони «Банда чотирьох» використовуються в контексті програмування ігор J2ME для створення масштабованого, зручного в обслуговуванні та надійного програмного забезпечення. Головоломка судоку та N-головоломка, які є настільними іграми для одного гравця, обрані для ілюстрації застосування шаблонів проектування для розробки ігор. Зазначимо, що ці ігри можна модифікувати в багатокористувацькі ігри, дозволяючи різним гравцям незалежно розв'язувати

одну й ту саму головоломку, наприклад, через бездротову мережу. Ми обрали однокористувацьку гру, щоб зосередитися на шаблонах у дизайні гри, а не на розширених функціональних можливостях гри.

Останнім часом ігри стали одним із найприбутковіших факторів у програмній індустрії. Зокрема, протягом останніх кількох років вважається, що ігрова індустрія приносить більший дохід, ніж кіноіндустрія, а темпи її розвитку були одними з найшвидше зростаючих в економіці Сполучених Штатів [6]. Крім того, дизайн ігор та методи, що використовуються для легшої та ефективнішої розробки, є дуже цікавою відкритою галуззю досліджень. Само собою зрозуміло, що комп'ютерні ігри відіграють дуже важливу роль у сучасному способі життя. Тому більше немає потреби пояснювати, що таке комп'ютерна гра. З іншого боку, не так очевидно, чому дослідження ігор є надзвичайно цікавою галуззю і водночас чому розробка ігор є дуже складним завданням.

Відповідь на перше питання має багато рівнів. Як згадувалося вище, хоча розробка ігор є дуже сильною галуззю, дослідження в цій галузі перебувають на початковій стадії. Цей факт спонукає фахівців з програмування ігор вимагати кращих методологій розробки та методів програмної інженерії [7]. Крім того, ігри є першим, а іноді й єдиним ринком для передових графічних технологій, що демонструє якість графіки, яку вони створюють. Було визнано, що ігрова індустрія спирається на дослідження академічних кіл, корпоративних дослідницьких лабораторій та власну роботу розробників ігор. На конференціях SIGGRAPH з'являються кілька статей, що вказують на необхідність перенесення знань з досліджень до промисловості.

Щоб довести, наскільки складним завданням є розробка ігор, ми представимо мінімальний персонал, який потрібен типовій такій компанії. Дискретні ролі персоналу не доводять складність самого завдання, оскільки це поширена тактика для команд розробників програмного забезпечення. Різниця між іграми та іншими видами програмного забезпечення полягає в тому, що в іграх групи розробників складаються з людей з різними галузями знань рисунок 1.2. Перш за все, потрібен сценарист; ця людина напише сценарій гри та

заповнить документ, який зазвичай називається «концептуальний документ» [8]. Провідний дизайнер ігор перетворить інформацію з концептуального документа в інший документ, який називається «проектний документ», який буде дороговказом протягом усього процесу розробки. Крім того, компанія наймає групу програмістів з різними навичками та досвідом, таких як програмісти движків та графіки, програмісти штучного інтелекту, програмісти звуку та програмісти інструментів. У співпраці зі звуковими програмістами компанія з розробки ігор найме музиканта та групу звукових ефектів. Крім того, мистецтво гри буде створено графічними дизайнерами, такими як художники персонажів, 3D-моделери та художники текстур. Зрештою, компанія повинна найняти тестувальників, які б грали в гру, щоб знаходити помилки та вносити пропозиції щодо змін у ігровий процес, графіку та сюжет [9].

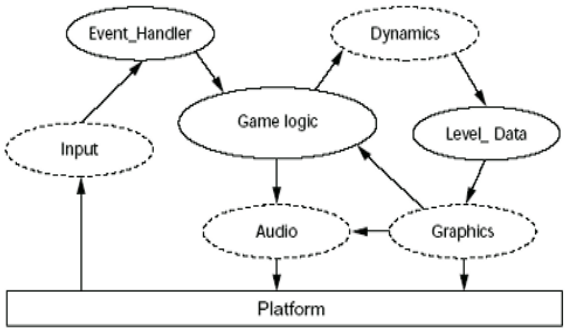


Рисунок 1.2 - Загальна архітектура гри

Програмування ігор є основним курсом у багатьох університетах, який триває один або два семестри, і існує чимало магістерських програм, пов'язаних з цією галуззю, але не так багато докторських дисертацій на цю тему. Цей факт свідчить про потребу галузі в методологіях розробки ігор, а також про брак відповідних наукових досліджень. Припущення щодо відсутності наукових досліджень полягає в тому, що ігри широко вважаються темою "м'яких навичок". Протягом останніх кількох років ця ситуація дещо змінилася завдяки кільком публікаціям про шаблони ігрового дизайну та використання ігрових двигунів для створення віртуальних світів і ГІС-додатків. Способи та обсяг викладання

програмування ігор в магістратурі та аспірантурі також розглядалися в кількох статтях [10].

1.2. Принципи об'єктно-орієнтованого програмування та патерни проєктування

Шаблони створення пов'язані з різними способами створення об'єктів. Слід зазначити, що рішення не створювати об'єкт і використовувати натомість існуючий екземпляр також може бути частиною шаблону, як у випадку шаблону "Сінглтон". Шаблони "Сінглтон" та "Пул об'єктів" будуть розглянуті детальніше на прикладах.

Найпростіший шаблон проектування, Singleton, гарантує, що існує лише один екземпляр класу з глобальною точкою доступу, що дозволяє використовувати його з будь-якого місця в програмному забезпеченні. Це означає, що кожен фрагмент коду, який посилається на відповідний клас, звертається до того самого екземпляра, незалежно від того, де або коли в програмі відбувається посилання.

Хоча це може бути результатом вибору програміста, коли розробник просто намагається не створювати екземпляр класу в кількох частинах коду, цей шаблон також може бути підкріплений програмно. Наприклад, виклик відповідного класу може включати перевірку, чи існує вже екземпляр. Якщо це так, новий екземпляр не створюється, а посилання спрямовується на існуючий екземпляр класу.

Оскільки шаблон Singleton означає, що клас створюється лише один раз і доступ до нього можливий з будь-якої точки проєкту, він є гарним вибором для класів, де потрібен глобальний, повторний доступ, але кілька копій можуть призвести до помилок, надмірності або проблем з продуктивністю. Потенційним прикладом шаблону в розробці ігор може бути контролер гравця. У цьому випадку доступ до контролера потрібен у всіх ігрових розділах гри, але кілька екземплярів можуть призвести до таких проблем, як дублювання або неточні

введення. Це робить singleton гарним вибором.

Новіше доповнення до словника шаблонів проектування, шаблон «Пул об'єктів», має на меті оптимізувати використання пам'яті шляхом переробки екземплярів часто потрібних класів, а не постійного знищення та повторного створення нових екземплярів. Це мінімізує навантаження на продуктивність від ініціалізації нових екземплярів та робить використання пам'яті більш передбачуваним, що може допомогти з оптимізацією. Пул об'єктів також керує загальною кількістю екземплярів, динамічно створюючи нові екземпляри за потреби, одночасно знищуючи будь-яке переповнення.

Оскільки це стосується переважно повторного використання кількох екземплярів одного й того ж класу, пул об'єктів найбільш корисний під час роботи з класами об'єктів, екземпляри яких запитуються, а потім неодноразово відкидаються. Якщо об'єкти ніколи не використовуються повторно після відкидання, їх зберігання в пам'яті є непотрібним витратою ресурсів. Аналогічно, якщо лише невелика кількість екземплярів використовується повторно рідко, вплив на продуктивність від підтримки пулу об'єктів може переважити переваги [11].

Оскільки відеоігри за своєю природою є динамічними програмними засобами, вони часто можуть отримати користь від використання шаблону «Пул об'єктів». Прикладом цього може бути гра, де гравець намагається стріляти по ворогах, що постійно з'являються, намагаючись уникнути їхніх атак. Така гра може використовувати пули об'єктів для керування як екземплярами ворогів, так і снарядами, що використовуються гравцем і ворогами. Ці типи об'єктів, як правило, є ідентичними екземплярами, створеними у великій кількості та з коротким терміном служби, що робить їх чудовим варіантом використання для шаблону «Пул об'єктів». Крім того, оскільки вони є такою великою частиною ігрового контенту, оптимізація використання пам'яті в цих випадках може суттєво вплинути на загальну продуктивність.

Інші шаблони створення, представлені в оригінальній книзі GoF, включають Абстрактну фабрику, Конструктор, Фабричний метод та Прототип.

Шаблони Абстрактної фабрики, Конструктора та Фабричного методу надають інтерфейси для створення об'єктів з різними підходами до того, наскільки суворо процес побудови обмежений певними класами або реалізаціями. Шаблон Прототип надає визначений шаблон для створення нових об'єктів.

Структурні шаблони пов'язані з більшими структурами, утвореними класами та об'єктами, та тим, як вони взаємодіють один з одним. У цій роботі детальніше розглянуто шаблони Декоратор та Композит [13].

Шаблон «Декоратор» можна використовувати як альтернативу підкласам, динамічно додаючи більше компонентів до об'єкта (Project Management Institute 2020, The Decorator). Оскільки декоратори можна використовувати під час виконання та комбінувати для нашарування їхніх ефектів, вони є гнучкішими, ніж попередньо визначені підкласи, і можуть краще реагувати на потреби програмного забезпечення.

Недоліком такої гнучкості є те, що використання кількох декораторів на одному об'єкті може призвести до конфліктів, оскільки їхні ефекти взаємодіють непередбачуваним або небажаним чином. Це можна пом'якшити, ввівши правила або обмеження на тип декораторів, які можна використовувати одночасно, або в якому порядку вони застосовуються до об'єкта, але за рахунок певної гнучкості.

Одним із прикладів використання шаблону декоратора в розробці ігор є ідея ефектів статусу. Ефекти статусу зазвичай є тимчасовими ефектами, які можна застосовувати до персонажів під час гри. Вони можуть змінювати параметри персонажів, на які вони впливають, такі як здоров'я або завдана шкода, додавати або видаляти корисні навички, або повністю вимикати персонажів до їх видалення. Оскільки ці ефекти зазвичай можна застосовувати до багатьох різних персонажів, але впливають на всіх них однаково, використання шаблонів декоратора для зберігання ефектів статусу, а не їх кодування окремо в кожен клас персонажів, не тільки економить час і зусилля, але й мінімізує ймовірність помилок або невідповідностей.

Шаблон «Композит» дозволяє розглядати кілька пов’язаних елементів, або листових об’єктів, як частини ієрархічної системи. Цю систему можна

розглядати як єдине ціле, звертаючись до найвищого елемента, який також називають вузлом. Елементи розташовані в деревоподібній структурі, яка може мати кілька шарів.

Основними цілями використання шаблону «Композит» є організація, навігація та мінімізація конфліктів між об’єктами. У розробці ігор шаблон Composite може використовуватися для керування підрозділами у військовій стратегії. У деяких іграх цього жанру гравець може давати завдання окремим бійцям або вибирати кілька таких підрозділів для формування більших груп. Ці групи можуть складатися з кількох різних типів об'єктів і бути різних розмірів, і групам можуть бути призначені ті ж завдання, що й окремим персонажам. Шаблон Composite, який дозволяє програмі обробляти листові та вузлові елементи однаково, добре працює для цього типу застосувань [14].

Інші структурні шаблони, включені до шаблонів проектування, включають Adapter, Bridge, Fagade, Flyweight та Proxy. Fagade концептуально схожий на Composite тим, що надає інтерфейс вищого рівня для використання кількох інтерфейсів підсистем уніфікованим чином. Ці інтерфейси можуть бути несумісними; метою Fagade є спрощення використання підсистем. Шаблон Adapter, з іншого боку, змінює інтерфейс одного класу, щоб дозволити йому взаємодіяти з іншим класом з несумісним інтерфейсом. Шаблон Flyweight підтримує обробку великої кількості подібних об'єктів. Bridge використовується для відокремлення абстракції від однієї або кількох реалізацій, що дозволяє їм змінюватися, не впливаючи одна на одну. Нарешті, Proxy використовується для забезпечення контрольованого доступу до об'єкта, де було б важко або небажано дозволити іншим об'єктам безпосередньо взаємодіяти.

1.3. Класифікація та застосування поведінкових, структурних і породжувальних патернів у розробці ігор

Поведінкові патерни описують способи, якими об'єкти спілкуються та поводяться один з одним. У той час як структурні патерни визначають спосіб,

яким об'єкти з'єднуються один з одним в ієрархії, поведінкові патерни стосуються інформації та запитів, а також того, як вони спрямовуються, передаються та на які реагують. Патерни, описані тут детальніше, включають Команду, Пам'ятку, Спостерігача та Стан.

Шаблон «Команда» інкапсулює запит як об’єкт, який потім можна передавати між об’єктами. Це дозволяє записувати запити для подальшого доступу. Запис запитів не лише означає, що їх можна відкласти та виконати пізніше, але й до них можна звернутися після виконання. Це означає, що дії можна скасувати або виконати кілька разів, що було б неможливо без зберігання інформації про запити.

Оскільки шаблон Command дозволяє зберігати запити та отримувати до них доступ пізніше, він має кілька можливих застосувань у розробці ігор. Наприклад, деякі головоломки дозволяють гравцеві повернутися на певну кількість ходів назад. Стратегічні ігри, де кілька гравців одночасно вирішують свої дії, які потім виконуються після вибору всіх дій, також вимагають запису різних запитів [15].

Там, де шаблон Command зберігає запит, шаблон Memento використовується для збереження внутрішнього стану об'єкта у зовнішньому об'єкті, щоб дозволити подальше відновлення внутрішнього стану. Memento зберігає лише стан без додаткової інформації, і доступ до нього має лише об'єкт, з якого він походить. Це дозволяє записувати та відновлювати стан без порушення інкапсуляції ініціатора.

Як подібний шаблон до Команди, шаблон Пам'яті також може бути використаний для реалізації функції скасування. Це особливо корисно у випадках, коли простого скасування використаних команд недостатньо для повернення до попереднього стану. Наприклад, будь-які дії з елементом випадковості неможливо точно скасувати без знання попереднього стану.

Шаблон «Спостерігач» пов’язує об’єкти між собою у відношенні «один до багатьох», таким чином, коли стан одного об’єкта змінюється, ця зміна передається всім його залежним об’єктам, і вони оновлюються відповідно. Сам

спостерігач не реалізує жодної функціональності; його роль полягає виключно в передачі інформації.

Передача даних залежним об'єктам та їхнє оновлення у відповідь на це відбувається автоматично, коли змінюється стан вихідного об'єкта [16]. Оскільки шаблон «Спостерігач» передає інформацію в порядку «один до багатьох», його можна використовувати для різних типів менеджерів подій у розробці ігор. Наприклад, персонаж гравця може мати внутрішню змінну, яка відстежує його поточний стан здоров'я. Це відображається у візуальному трекері в інтерфейсі користувача, а також змінює зовнішній вигляд персонажа, коли він падає нижче певного рівня. Замість того, щоб внутрішня функція здоров'я гравця відстежувала ці зміни та запитувала оновлення, можна використовувати шаблон «Спостерігач». У цьому випадку «Спостерігач» відстежуватиме стан здоров'я персонажа гравця та передаватиме будь-які оновлення як об'єкту трекера, так і системі рендерингу персонажа. Потім ці об'єкти впроваджують необхідні зміни в рамках свого регулярного циклу оновлення, без необхідності запиту від вихідного скрипта.

Шаблон «Стан» дозволяє об’єкту мати кілька різних моделей поведінки, які змінюються залежно від його внутрішнього стану. Шаблони поведінки є частиною функціональності об’єкта, але різні моделі поведінки активуються або вмикаються залежно від активного стану. Це усуває необхідність модифікувати об’єкт під час виконання, оскільки вся необхідна функціональність інкапсульована в ньому.

Прикладом використання шаблону «Стан» у розробці ігор є використання станів для керування інструментом, що використовується у фермерській грі. У багатьох іграх жанру фермерських симуляторів існує лише обмежена кількість інструментів, кожен з яких має свою власну функціональність. Це можна реалізувати за допомогою шаблону «Стан», використовуючи внутрішній стан для запису того, який інструмент, якщо такий є у гравця. Цей стан потім використовується для визначення того, яка дія виконується, коли гравець вводить дані інструмента, а також може бути використаний для вибору

відповідної анімації.

Категорія поведінкових шаблонів у шаблонах проектування також включає шаблони Ланцюг відповідальності, Інтерпретатор, Ітератор, Медіатор, Стратегія, Метод шаблону та Відвідувач.

Шаблон відповідальності дозволяє передавати запити кільком одержувачам, будь-який з яких потенційно може обробити запит. Шаблон інтерпретатора використовується для визначення граматики простих мов та використання отриманих правил для інтерпретації речень цією мовою. Шаблон ітератора може отримувати доступ до окремих елементів агрегованого об'єкта по одному без необхідності доступу до внутрішньої структури. Медіатора можна розглядати як версію шаблону спостерігача «багато до багатьох», оскільки він інкапсулює взаємодію набору об'єктів, спрямовуючи запити та мінімізуючи прямі зв'язки між класами. Шаблон відвідувача використовується для визначення та виконання операцій над набором класів без необхідності модифікації цільових класів [17].

Нарешті, шаблони «Стратегія» та «Метод шаблону» стосуються алгоритмів. Шаблон «Стратегія» дозволяє взаємозамінно використовувати набір пов'язаних алгоритмів, тоді як шаблон «Метод шаблону» визначає базовий скелет алгоритму, а потім дозволяє підкласам змінювати окремі кроки алгоритму, не впливаючи на його фактичну структуру.

1.4. Висновок по розділу

У першому розділі розглянуто теоретичні основи проєктування ігрових застосунків жанру 2D-платформер та принципи використання об'єктно-орієнтованих патернів. Проаналізовано особливості побудови ігрової логіки, принципи об'єктно-орієнтованого програмування та класифікацію патернів проєктування. Проведене дослідження показало, що застосування поведінкових, структурних і породжувальних патернів сприяє створенню гнучкої, масштабованої та легко підтримуваної архітектури ігрового застосунку.

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ РОЗРОБКИ ІГРОВОГО ЗАСТОСУНКУ

2.1. Аналіз патернів проєктування для реалізації ігрової логіки

Термін «шаблони проектування» стосується визначених рішень поширених проблем проектування. Поняття шаблонів вперше ввів Крістофер Александер, який визначив та запропонував рішення поширених архітектурних проблем. У своїй роботі він розглядав питання, чи може якість в архітектурі бути об'єктивною. Досліджуючи кілька архітектурних артефактів, він виявив, що «хороші» якісні проекти мають деякі спільні характеристики або спільні «спільні рішення поширених проблем» [19]. Шаблони також можна використовувати в архітектурі програмного забезпечення, і, якщо їх правильно застосовувати, вони підвищують гнучкість та можливість повторного використання базової системи. Об'єктно-орієнтовані шаблони проектування визначають зв'язки між класами-учасниками та їхню співпрацю. Такі рішення особливо спрямовані на покращення адаптивності шляхом модифікації початкового проекту, щоб полегшити майбутні зміни. Кожен шаблон дозволяє змінювати деякий аспект структури системи незалежно від інших аспектів. Досліджували вплив шаблонів проектування на зрозумілість та зручність обслуговування. Їхній експеримент проаналізував витрачений час та правильність результату для реалізації заданого набору вимог з використанням систем зі шаблонами проектування та без них. Результати показали, що деякі шаблони набагато легше зрозуміти та використовувати, ніж інші, і що шаблони проектування не є універсально хорошими чи поганими. Однак, мається на увазі, що якщо шаблони використовуються правильно та у відповідних випадках, вони виявляються надзвичайно корисними щодо зручності підтримки та зрозумілості.

На відміну від попередньої думки, що класи, що беруть участь у шаблонах проектування, більш схильні до змін, ніж інші класи системи. Їхня оцінка базувалася на аналізі існуючих систем, і один з її результатів показав, що «класи,

які відіграють певну роль у шаблонах проектування, змінюються частіше, ніж інші класи. Дані тематичного дослідження не показують, що шаблони проектування підтримують адаптивність» [20]. На думку авторів, класи-учасники шаблонів забезпечують ключову функціональність системи, що може пояснити, чому ці класи, як правило, змінюються відносно часто.

Використання шаблонів проектування в розробці ігор є відкритою галуззю досліджень. У літературі нелегко знайти каталоги зі шаблонами, які можна було б використовувати як «загальні рішення поширених проблем» в іграх. Такі каталоги спростили б спілкування між розробниками та зробили б документацію такого роду програм більш зрозумілою. Шаблони проектування для ігор можна розглядати з двох різних точок зору: по-перше, як шаблони, що використовуються для опису ігрової механіки (геймплею та правил гри), а по-друге, як використання об'єктно-орієнтованих шаблонів проектування в програмуванні ігор.

Щодо ігрової механіки, представили набір шаблонів проектування, які по суті є описами (використовуючи єдиний словник) повторюваних схем взаємодії, що стосуються сюжету та ігрового процесу гри. Таким чином, ці шаблони не пов'язані з архітектурою чи кодом програмного забезпечення. Запропоновані шаблони зібрані в результаті інтерв'ю з професійними програмістами ігор, аналізу існуючих ігор та трансформації ігрової механіки. Як згадувалося, «Спосіб розпізнати шаблони — це грати в ігри, думати про ігри, мріяти про ігри, розробляти ігри та читати про ігри» [21]. Прикладом такого шаблону є шаблон «Папір — Камінь — Ножиці» , добре відомий у спільноті геймдизайнерів (також як трикутність*)*. Цей шаблон використовується, коли є три дискретні стани (або варіанти для гравця), і варіант A перемагає варіант B, варіант B перемагає варіант C, а варіант C перемагає варіант A.

Щодо програмування ігор, у наступних розділах будуть представлені приклади об'єктно-орієнтованих шаблонів проектування у двох ігрових модулях. Будуть розглянуті шаблони проектування - "Стратегія" та "Спостерігач", що стосуються модуля ігрової логіки (рисунок 2.1), а також шаблони "Стан" та

"Міст", що стосуються 3D-аспектів у графічному модулі (рисунок 2.1). Само собою зрозуміло, що об'єктно-орієнтовані шаблони проектування можна застосовувати при проектуванні та кодуванні будь-якого ігрового модуля. Вибір модуля ігрової логіки та графіки не означає, що шаблони більш застосовні до цих галузей.

Об'єктно-орієнтовані шаблони проектування в ігровій логіці

Хоча досі не знайдено багато робіт щодо використання об'єктно-орієнтованих шаблонів проектування в іграх, ми вважаємо, що таке використання може бути дуже корисним у цій галузі. Цей факт можна перевірити, дослідивши вихідний код ігор на наявність шаблонів проектування. Як перший підхід, ми наведемо приклади того, як об'єктно-орієнтовані шаблони проектування можна використовувати в простих іграх. Крім того, ми представимо реальні ігри, які можуть використовувати ці об'єктно-орієнтовані шаблони проектування та покращити їх дизайн стратегії визначає сімейство алгоритмів, інкапсулює кожен з них та робить їх взаємозамінними [22]. Стратегія здається дуже ефективним способом розробки способу, яким шахова гра може імітувати поведінку різних гравців зі штучним інтелектом. Припустимо, що гра представляє шаховий турнір. Кожен гравець використовує різний евристичний алгоритм для обчислення мінімального та максимального значення, щоб визначити наступний хід комп'ютера.

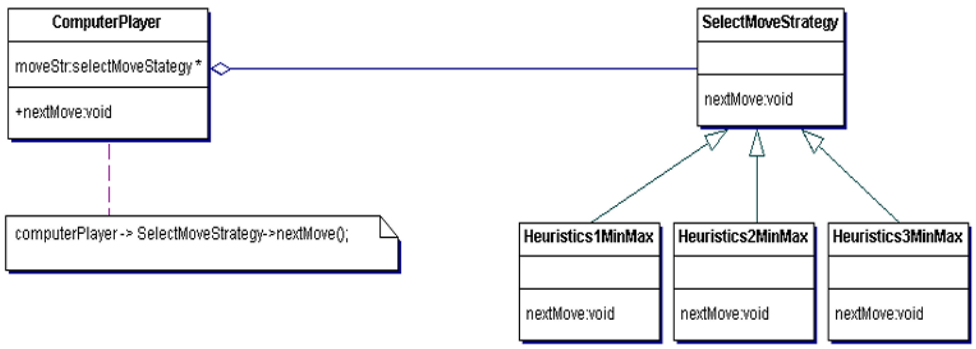


Рисунок 2.1 - Стратегічний шаблон - приклад шахів

На наведеній вище UML-діаграмі клас computerPlayer представляє контекст. Він підтримує посилання на об'єкт стратегії; він налаштований з конкретним об'єктом стратегії та може визначати інтерфейс, який дозволяє стратегії отримувати доступ до даних. Клас seiectMovestrategy оголошує інтерфейс, спільний для всіх підтримуваних алгоритмів. Він використовується контекстом (computerPlayer) для виклику алгоритму, визначеного конкретною стратегією. Нарешті, конкретна стратегія (Класи Heuristics1MinMax, Heuristics2MinMax, Heuristics3MinMax) реалізує алгоритми відповідно до інтерфейсу стратегії (selectMovestrategy) [23].

Використовуючи цей шаблон, пропонується альтернатива умовним операторам для вибору поведінки. Крім того, цей дизайн є більш зрозумілим, легшим у підтримці та більш розширюваним завдяки дискретним підкласам. [24].

Цей шаблон також можна було б використовувати в таких іграх, як нарди, карткові ігри та будь-де, де можна застосовувати алгоритми штучного інтелекту (наприклад, алгоритми A* для гравців, керованих комп'ютером, у рольових іграх RPG, пригодницьких та стратегічних іграх). Однак, шаблон «Стратегія» є одним з найпоширеніших шаблонів і може виявитися корисним у багатьох випадках.

Крім того, шаблон спостерігача визначає залежність «один до багатьох» між об'єктами, яка полягає в тому, що коли один об'єкт змінює стан, усі його залежні об'єкти автоматично повідомляються та оновлюються [25]. Приклад цього шаблону трохи складніший і використовується в більш вимогливих іграх. Припустимо, що у футбольному менеджері команда наймає тренерів, які покращують атрибути гравців. Це відбувається не лише один раз і не саме тоді, коли наймають тренера. Оновлення застосовується, коли спеціальна змінна досягає певного значення, і її значення збільшується відповідно до навичок тренера та графіка тренувань команди. Атрибути гравців команди повинні автоматично оновлюватися, коли змінна рівня тренування досягає вищезгаданого значення.

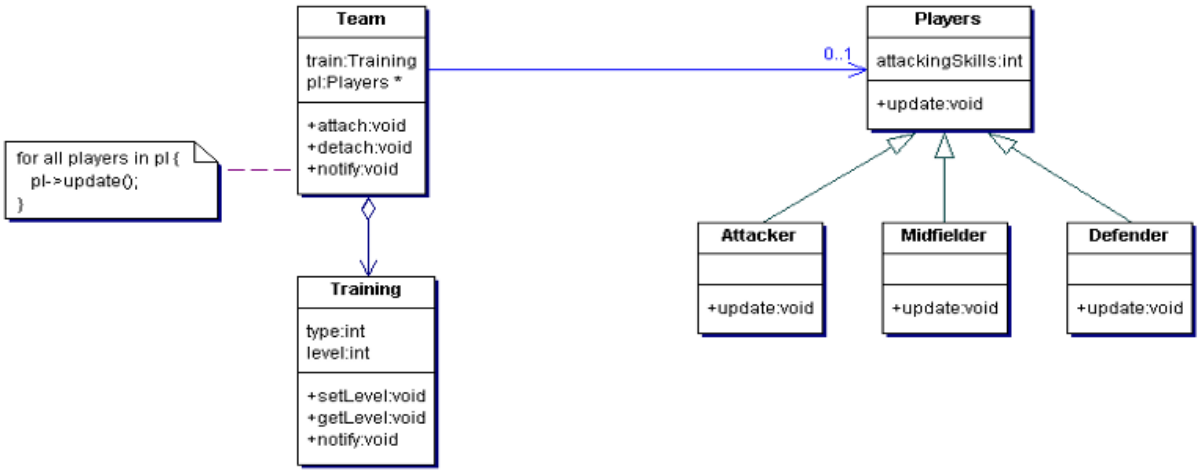


Рисунок 2.2 - Шаблон спостерігача - приклад футбольного менеджера

На наведеному вище рисунку клас Team є суб'єктом, який знає своїх спостерігачів та надає інтерфейс для їх підключення та від'єднання. Спостерігач (клас Players) визначає інтерфейс оновлення для об'єктів, які повинні бути повідомлені про зміни в суб'єкті. Клас Training зберігає інформацію про поточний стан суб'єктів. Конкретні спостерігачі (класи Attacker, Midfielder та Defender) реалізують інтерфейс оновлення спостерігача, щоб підтримувати свій стан узгодженим зі станом суб'єкта [26].

Цей шаблон дозволяє розробникам незалежно змінювати суб'єктів та спостерігачів. Таким чином, вони можуть повторно використовувати суб'єктів, не використовуючи повторно своїх спостерігачів, і навпаки. Крім того, існує абстрактний зв'язок між суб'єктом та спостерігачами. Більше того, сповіщення про оновлення транслюється, і в результаті суб'єкт не цікавиться, яких спостерігачів цікавить зміна, оскільки це його відповідальність реагувати на неї.

Об'єктно-орієнтовані шаблони проектування в ігровій графіці

Шаблон стану дозволяє об'єкту змінювати свою поведінку, коли змінюється його внутрішній стан [27]. Цей шаблон можна використовувати в іграх, де об'єкт змінює рівень деталізації (LOD) свого зовнішнього вигляду залежно від відстані від камери. Наприклад, у будь-якій грі-шутері від першої особи всі об'єкти сцени повинні виглядати елегантніше, коли головний герой до них наближається. Отже, зображення, що представляє текстуру об'єкта, матиме

більшу роздільну здатність, коли об'єкт наближається до положення головного героя.

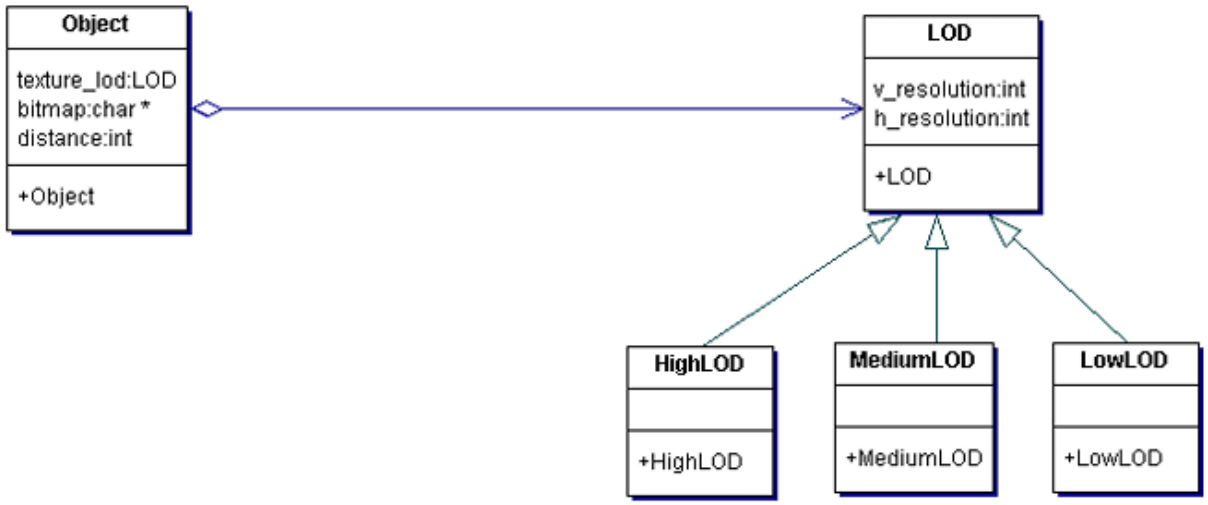


Рисунок 2.3 - Шаблон станів – приклад шутера від першої особи

На діаграмі вище контекст (клас об'єкта) підтримує конкретний стан у будь-який заданий момент часу. Стан (клас LOD) визначає інтерфейс для інкапсуляції поведінки, пов'язаної з певним станом контексту. Підкласи конкретних станів (класи HighLOD, MediumLOD та LOWLOD) реалізують поведінку, пов'язану зі станами контексту [28].

Завдяки цьому шаблону кожна поведінка, пов'язана з певним станом, прив'язується до одного об'єкта. Отже, нові стани та переходи можна легко додавати, визначаючи нові підкласи. Крім того, об'єкти станів можна спільно використовувати, якщо вони не мають змінних екземплярів. Нарешті, об'єкти станів захищають контекст від несумісних внутрішніх станів, оскільки переходи станів є атомарними (перехід між станами відбувається шляхом зміни значення лише однієї змінної, а не кількох) [29]. Цей шаблон можна застосувати в грі Pac-Man, де стани представляють поведінку та швидкість ворогів, в грі-арканоїд, де розмір та атрибути смуги прив'язані до конкретних підкласів станів, а також у простих іграх типу "стріляй по них", таких як Space Invaders та Brain Wave. У них гравець стріляє по кількох цілях та заробляє очки. Під час гри він також може

змінювати зброю, одягати броню, щити тощо. Всі ці зміни можна представити як різні стани мосту відокремлює абстракцію від її реалізації, завдяки чому вони можуть змінюватися незалежно [30]. Цей шаблон можна використовувати в будь-якій грі з 3D-об'єктами, які можна малювати різними способами (текстура, матеріал тощо). Водночас 3D-об'єкт також може змінюватися, оскільки він може бути примітивом (сфера, куб тощо) або експортом з 3D-пакета (файл .3ds, модель .m2 тощо).

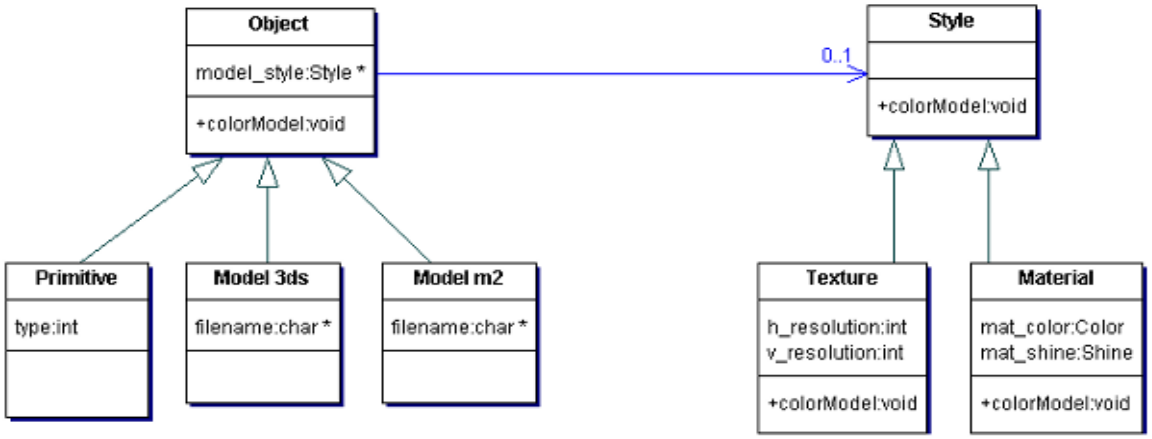


Рисунок 2.4 - Шаблон мосту - приклад 3D-гри

Абстракція (клас об'єкта) визначає інтерфейс абстракції, який підтримує посилання на об'єкт типу реалізатор (клас стилю). Реалізатор визначає інтерфейс для класів реалізації. Цей інтерфейс не обов'язково має точно відповідати інтерфейсу абстракції. Зазвичай інтерфейс реалізатора надає лише примітивні операції та високорівневі операції абстракції, засновані на цих примітивах. Удосконалені абстракції (клас modei_3ds , клас примітивів та клас modei_m2) розширюють інтерфейс, визначений абстракцією. Нарешті, конкретний реалізатор (клас Texture та клас Material) реалізує інтерфейс реалізатора та визначає його конкретну реалізацію [31].

Застосовуючи цей шаблон, реалізація не прив'язана постійно до інтерфейсу. Реалізацію абстракції можна налаштувати під час виконання, і навіть об'єкт може змінювати свою реалізацію під час виконання. Таким чином, шаблон

розділяє інтерфейс та реалізацію та усуває залежності від реалізації під час компіляції. Крім того, система зі шаблоном мосту є більш розширюваною. У будь-який час ви можете незалежно розширити абстракцію та ієрархії реалізаторів. У прикладі, наведеному на рисунку 2.4, абстракції Об'єкта та Стиль можуть змінюватися незалежно. Отже, якщо пізніше з'явиться новий 3D-пакет і буде доступний завантажувач, який обробляє його моделі, його можна буде додати до програми без особливих труднощів. Аналогічно, якщо виникає нова вимога до матеріалів кольорового типу, відмінного від RGB (червоний-зелений-синій), наприклад, індексовані кольори (використовуються в OpenGL), клієнт може використовувати вищезгадану структуру з незначними змінами.

2.2. Архітектурні рішення та моделювання компонентів гри

Щоб оцінити переваги об'єктно-орієнтованих шаблонів проектування в розробці ігор, ми дослідили дві існуючі ігри (Cannon Smash та Ice Hockey Manager), випущені з відкритим вихідним кодом в Інтернеті [32]. Ці ігри були обрані, оскільки їхній код є загальнодоступним, і випущено кілька їхніх версій. Цей факт у поєднанні з існуванням принаймні одного шаблону проектування принаймні в одній версії зробив оцінку можливою. Оцінювання було проведено для двох версій програм, перша з реалізованим досліджуваним шаблоном проектування, а інша - без нього. Дві оцінювані програми були розроблені з використанням різних мов програмування (C++ та Java). Шаблони проектування в програмі на C++ були виявлені шляхом ручної перевірки та використання інструментів зворотного проектування. У програмі на Java також використовувався інструмент виявлення шаблонів проектування. Критерії оцінювання будуть не лише якісними, але й кількісними, з використанням добре відомих метрик. У першому розділі цього розділу буде представлено огляд того, як розраховуються ці метрики та чому ці метрики були обрані.

Метрики якості програмного забезпечення

Розраховані метрики програмного забезпечення можна розділити на

чотири основні категорії: метрики розміру, складності, зв'язку та когезії. Ці категорії були обрані відповідно, використовуючи метрики з тих самих категорій, провели дослідницький аналіз емпіричних даних щодо продуктивності. Вплив шаблонів проектування на показники метрик зв'язку також досліджувався. Метрики розміру, когезії та когезії використовувалися для ідентифікації екземплярів шаблонів, що означає, що наявність шаблону впливає на їхні значення. Міри зв'язку є найбільш очевидним вибором, оскільки, використання абстракцій у шаблонах проектування в першу чергу спрямоване на зменшення залежностей між класами. Шаблони також відповідають принципу єдиної відповідальності і в результаті цілком обґрунтовано очікувати, що когезія зростає при правильному використанні шаблонів. Крім того, застосування шаблонів, безумовно, передбачає введення нових класів/інтерфейсів, переміщення методів і в більшості випадків вимагає додаткового коду. Отже, очікується, що метрики розміру вказуватимуть на збільшення розміру коду. Зрештою, очікується, що поліморфізм, що використовується в багатьох шаблонах, усуне складні фрагменти коду (такі як каскадні оператори if або switch) і таким чином зменшить складність відповідних класів.

Розраховуються метрики розміру: рядки коду (LOC) та кількість класів (NOC). Метрика LOC – це традиційна міра розміру. Вона підраховує кількість рядків коду. Метрика NOC підраховує кількість класів у системі [33].

Розраховані три метрики складності: складність атрибутів (AC), зважені методи для класу 1 (WMPC1) та зважена

Методи класу 2 (WMPC2). Метрика AC визначається як сума значень кожного атрибута в класі (кожен тип і тип масиву має заздалегідь визначене значення складності), таким чином, складність зростає зі збільшенням значення. Метрика WMPC1 – це сума складності всіх методів класу, де кожен метод зважується за своєю *цикломатичною складністю* (CC). CC представляє кількість можливих шляхів через алгоритм шляхом підрахунку кількості різних областей на графі потоку, тобто кількість випадків, якщо, ***для*** та оператори

while в тілі операції. Отже, складність зростає, якщо значення метрики збільшується. Метрика WMPC2 призначена для вимірювання складності класу, припускаючи, що клас з більшою кількістю методів, ніж інший, є складнішим, і що метод з більшою кількістю параметрів, ніж інший, також, ймовірно, буде складнішим [34].

Зв'язок оцінюється за допомогою однієї метрики – *Коефіцієнта Зв'язку* (CF). Більш конкретно, метрика CF обчислюється як дріб, де чисельник представляє кількість зв'язків без успадкування, а знаменник – максимально можливу кількість зв'язків у системі.

Зрештою, когезія оцінюється за допомогою метрики *відсутності когезії методів* (LCOM). Цей показник перевіряє для кожної пари методів у класі, чи мають вони доступ до спільного атрибута. У цьому випадку два методи вважаються когерентними.

Вимірювання було проведено для класів, що беруть участь у шаблонах, для всієї програми, а також для програмного забезпечення на основі Java для пакета, до якого належать класи, що беруть участь у шаблонах. Сукупні метрики обчислюються як середнє значення метрик їхніх членів.

Приклад оцінювання 1 - Cannon Smash

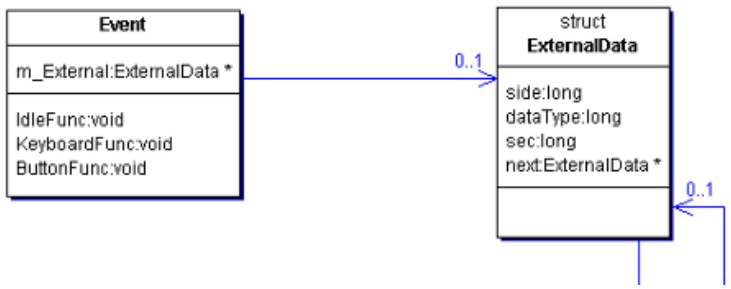


Рисунок 2.5 - Підхід до гри cannon smash версії 0.4.5 для багатокористувацької гри

Гра наразі перебуває у версії 0.6.6 та підтримує багатокористувацький режим. Цю функцію було додано у версії 0.4.5, де програмісти використовували клас ExternaiData для обробки повідомлень, що надходять з мережі, таких як

інструкції від віддалених гравців (ця операція належить модулю введення гри, показаному на рисунку 2.5). Клас Event , який використовується як обробник подій, використовуючи функції введення OpenGL, мав асоціацію через посилання на структуру ExternaiData .

Недоліком представленого підходу є те, що вхідні дані можуть бути отримані у трьох різних типах, і залежно від поточного типу може бути використаний різний слід використовувати стратегію читання. У цій версії програмісти використали оператор switch для реалізації цього механізму, як показано нижче.

```
switch (theEvent.m_External->dataType) {

    case DATA_PV:
        targetPlayer->Warp(theEvent.m_External->data);

        if (targetPlayer == thePlayer) {
            fThePlayer = true;
        } else if (targetPlayer == comPlayer) {
            fComPlayer = true;
        }
        break;

    case DATA_PS:
        targetPlayer->ExternalSwing(theEvent.m_External->data);

        if (targetPlayer == thePlayer) {
            fThePlayer = true;
        } else if (targetPlayer == comPlayer) {
            fComPlayer = true;
        }
        break;

    case DATA_BV:
        theBall.Warp(theEvent.m_External->data);
        fTheBall = true;
        break;
}

// Очищення пам'яті та перехід до наступного елемента
externalOld = theEvent.m_External;
theEvent.m_External = theEvent.m_External->next;
delete externalOld;
```

Рисунок 2.6 - Підхід «Cannon Smash 0.4.5» для багатокористувацького режиму — вихідний код

З рисунка 2.6 очевидно, що код зразка страждає від «непотрібного повторення» [35]. Тому, якщо пізніше з'явиться вимога щодо додаткових «стратегій читання», оператор switch слід змінити. Це доводить, що код зразка не є повторно використовуваним, оскільки програма недостатньо гнучка, щоб легко адаптувати додаткові стратегії. Такі вимоги з'явилися в наступних

поколіннях програми. Отже, в поточній версії є п'ять стратегій читання на відміну від трьох стратегій, що з'явилися у версії 0.4.5.

Наведений вище опис стосується поліморфізму та, зокрема, використання шаблону стратегії, як описано на рисунку 2.7. Таким чином, складність моделі зменшиться, а гнучкість збільшиться. Розробники застосували цей шаблон до наступної версії програми (версія 0.4.6), що дозволило їм легко додавати більше стратегій для зчитування даних з мережі. Діаграма UML, що представляє дизайн класів, що обробляють дані багатокористувацького режиму у версії 0.4.6, показана нижче, на рисунку 2.7.

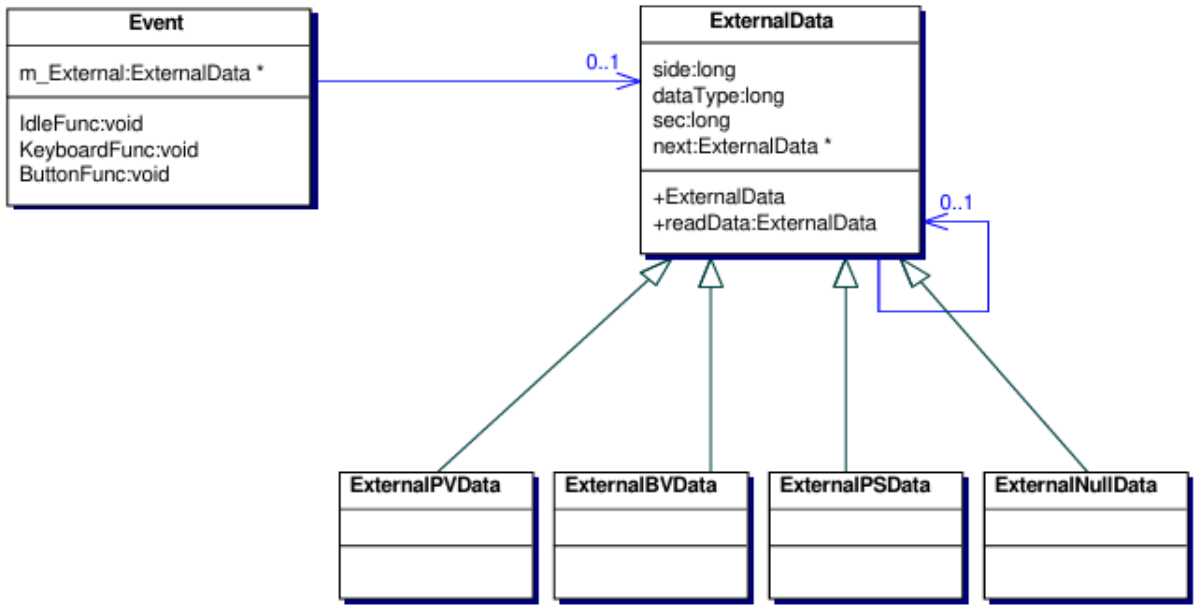


Рисунок 2.7 - багатокористувацький підхід до гри cannon smash 0.4.6

Два підходи до проектування були проаналізовані за допомогою інструменту CASE з можливостями зворотного проектування для розрахунку показників.

Найбільш значущим значенням, яке підтверджує цей факт, є метрика WMPC1, яка зменшується на 23,6% з використанням шаблону проектування. Зменшення складності також відображається на метриках на рівні проекту, проте в меншій мірі, через залучення інших класів. Єдина метрика складності, яка збільшується, це Складність атрибутів. Однак це абсолютно не має

значення для використання шаблону, оскільки це збільшення спричинене додаванням змінної, яка не бере участі в шаблоні.

Зменшення значення метрики LCOM для класу Event (навіть незначне, 0,9%) свідчить про те, що зв'язність між методами збільшилася, і шаблон використовується правильно. Більше того, зв'язок для всієї програми зменшився, як підказує значення метрики CF. Відповідне покращення розраховано на 16,6%.

Хоча розмір (рядків коду) класу Event зменшується, використання шаблону проектування збільшило LOC та NOC всієї програми. Шаблон відповідає за 4 з 9 (тобто 44,4%) класів, доданих у покращеній версії, та близько 3% доданих рядків. Навіть якщо цим збільшенням не можна нехтувати, використаний шаблон сприяє повторному використанню, гнучкості та зменшенню складності коду.

Результати узгоджуються з попередніми дослідженнями, які стверджували, що загалом застосування шаблонів проектування призводить до більш зрозумілого та зручного в обслуговуванні коду. Результати таблиці 1 означають, що гра, яка використовує шаблони, має зменшену складність, зменшену зв'язність та дещо підвищену зв'язність, і в цьому сенсі її легше зрозуміти, протестувати та підтримувати. Більше того, було помічено, що розмір коду збільшується при використанні шаблонів, що також підтверджується результатами [37].

Приклад оцінювання 2 - Менеджер з хокею на льоду

Ice Hockey Manager (IHM) – це симуляція тренування хокейної команди. Поточна версія гри – 0.2, і ще дві версії випущені з відкритим кодом. У версії 0.1.1 розробники використовували вісім (8) екземплярів шаблонів проектування. Зокрема, один (1) шаблон фабрики, три (3) шаблони спостерігачів, два (2) шаблони стратегії та два (2) шаблони. Розпізнавання шаблонів стало можливим завдяки використанню відповідного інструменту [38]. Незважаючи на те, що дизайн цієї версії 0.1.1 був надзвичайно добре структурованим, деякі пакети та деякі класи страждали від підвищеного

WMPC1.

Цю проблему було вирішено завдяки використанню більшої кількості шаблонів проєктування, незважаючи на те, що було застосовано додаткову функціональність. У версії 0.1.2 було визначено двадцять шість (26) екземплярів шаблонів проєктування. Зокрема, два (2) шаблони фабрики, один (1) шаблон прототипу, три (3) шаблони адаптера, один (1) композитний шаблон, два (2) шаблони декоратора, шість (6) шаблонів спостерігача, вісім (8) шаблонів станів та три (3) шаблони шаблонів. Хоча опис використання всіх 26 шаблонів виходить за рамки цієї роботи, будуть розглянуті два шаблони (міст та стан) через їх попередній опис у розділі 3.2. Незважаючи на це, наведено кумулятивні результати, які показують сукупний вплив вісімнадцяти (18) додаткових шаблонів.

Шаблон станів відображає той факт, що хокеїст може бути як воротарем, так і польовим гравцем. Це означає, що всі гравці, незалежно від їхньої позиції, мають деякі спільні характеристики, а також відрізняються за кількома іншими. Тому створюється суперклас Player, який інкапсулює всі спільні характеристики та спільну поведінку хокеїстів. Цей клас успадковується двома підкласами, Goalie та FieidPlayer, які представляють варіанти поведінки та характеристик кожної категорії. Таким чином, будь-який клієнт програми може створювати або керувати гравцем, не знаючи заздалегідь, чи є він воротарем, чи польовим гравцем. Наприклад, клас Team , який містить посилання на масив гравців, не повинен обробляти їх окремо, а через спільний інтерфейс.

Шаблон мосту використовується для забезпечення зв'язку між інтерфейсами Player та player_Attributes, щоб реалізацію класу player можна було налаштувати під час виконання. Отже, підклас Field_player можна пов'язати з підкласом Field_player_Attributes через спільний інтерфейс як для воротарів, так і для польових гравців. Цей факт може бути надзвичайно корисним для будь-яких процедур, які потребують доступу до атрибутів гравця, не знаючи його позиції в складі.

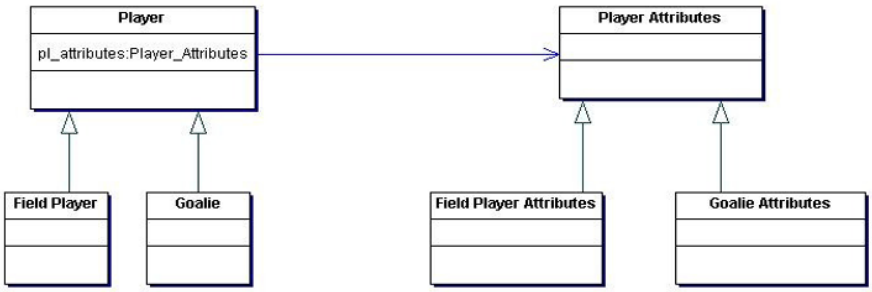


Рисунок 2.8 - Менеджер хокею на льоду – версія 0.1.2 шаблон Bridge-State

Поєднання цих шаблонів впливає на багато класів проекту. Одним з найбільш характерних є клас PlayerAttributes. У версії 0.1.2 шаблони проектування виключили два умовні оператори, що зменшило WMPC1 пакета (один з цих умовних операторів показано нижче, рисунок 2.9).

```
public int getSpecificAttributesAverage() {
    int specificAttributes = 0;

    // Specific Attributes average
    if (getPosition() == POSITION_GOALTENDING) {
        int total = 0;
        total += goalie_gloveLeft;

        // . . . (додаткові атрибути воротаря)

        specificAttributes = total / 7;
    } else {
        int total = 0;
        total += field_shotQuickness;

        // . . . (додаткові атрибути польового гравця)

        total += field_scoring;
        specificAttributes = total / 10;
    }

    return specificAttributes;
}
```

Рисунок 2.9 - Менеджер хокею на льоду версії 0.1.1 - вихідний код

Результати кількісної оцінки проекту наведено нижче. Використані метрики ідентичні тим, що описані в розділі 4.1. Проект було оцінено у три рівні. Спочатку було розраховано метрики для класу PlayerAttributes , потім метрики для пакета гравця та, нарешті, метрики для всього проекту IHM.

З наведеної вище таблиці стає очевидним, що шаблони проєктування зменшили складність класу, що бере участь у шаблонах. Це підтверджують

метрики WMPC1 та WMPC2, значення яких для класу PlayerAttributes зменшилися на 47,4% та 68,2% відповідно . Пакет ihm.game.player.* є найчастіше використовуваним пакетом у доданих шаблонах проєктування (у 6 з 26 шаблонів, 23,1%), тому він демонструє найзначніше зниження складності (43,6% та 57,3%). З іншого боку, складність системи залишається стабільною або дещо зростає, можливо, через додану функціональність.

Подібно до прикладу в 4.2, когезія покращилася згідно з метрикою LCOM, яка зменшилася на 44%, 77,4% та 90,2% для проекту, пакета та класу відповідно. Ці результати здаються досить екстремальними, але вони відображають наслідки вісімнадцяти (18) додаткових шаблонів, а не лише мосту, про який йшлося вище. Крім того, зв'язок у системі зменшився на 44,4%, як це випливає зі значення метрики CF.

На противагу цьому, використання шаблонів проектування збільшило розмір проекту щодо кількості класів та рядків коду. На рівні пакета ці показники зросли на 116,6% та 5,9% відповідно, а на рівні проекту – на 25,6% та 21,2%.

2.3. Програмні засоби та технології розробки 2D-платформера

У цьому підрозділі представлено всі технології, що використовуються в кінцевому продукті, загальний опис технологій, їхню мету в цій роботі та причину, чому вони були обрані для кінцевого продукту.

Ігровий рушій Unity

Ігровий движок — це програмний інструмент, розроблений спеціально для розробки ігор, оптимізований та налаштований таким чином, щоб спростити та оптимізувати процес розробки ігор. Зазвичай він підтримує різні мови програмування, 2D- та 3D-рендерери графіки, фізичний движок для імітації фізики реального світу в грі та звуковий движок для обробки всіх аспектів гри, пов'язаних зі звуком, а також має багато функцій, таких як анімація, підтримка багатокористувацької гри та багато іншого.

Для цього проєкту було обрано ігровий рушій Unity Engine, розроблений

Unity Technologies. Unity — це ігровий рушій, створений у 2005 році, який підтримує розробку як 2D, так і 3D ігор. Він був створений, щоб надати - розробникам ігор інструмент для легкого доступу до нього на шляху розробки ігор. Протягом багатьох років рушій зазнав значних удосконалень, щоб йти в ногу з найновішими практиками та технологіями.

Ігровий рушій Unity було обрано з кількох основних причин. Оскільки це проста 2D-гра, було б розумно обрати ігровий рушій, який простий у використанні, зручний для початківців та підходить для розробки простої гри. Unity відповідає всім цим критеріям, і з огляду на величезну популярність Unity, підтримка спільноти є величезним фактором, що забезпечує миттєву підтримку та зворотний зв'язок у проблемних ситуаціях.

Unity переважно використовує C# як основну мову сценаріїв, але також дозволяє користувачам використовувати JavaScript, що робить його дуже зручним майже для кожного програміста, оскільки JavaScript є однією з найпопулярніших мов.

Доступ до магазину ресурсів Unity є однією з головних причин вибору ігрового рушія для цієї роботи. Магазин ресурсів Unity надає різноманітні ігрові ресурси і спрайти, платні чи безкоштовні, які можна імпортувати та впроваджувати безпосередньо в гру.

Усі вищезазначені причини, разом із попереднім досвідом автора з Unity, роблять його дуже підходящим, розумним та економним за часом ігровим рушієм для цього дипломного продукту.

Магазин активів Unity

Unity Asset Store – це місце, де творці та розробники ігор можуть легко надавати та отримувати доступ до ігрових ресурсів через цифровий магазин. Творці ігор можуть створювати ігри швидше завдяки готовим ресурсам, що, в свою чергу, робить гру більш складною. Готові ресурси також можна використовувати як тимчасові ресурси під час створення гри, залишаючи тягар створення ресурсів позаду, поки ви працюєте над логікою та зосереджуєтеся на самому ігровому процесі, щоб якомога швидше реалізувати прототип.

У цій роботі було використано безкоштовний пакет Unity Asset Store під назвою "Pixel Adventure 1" для надання ресурсів та спрайтів для рівнів та персонажів, які створюються в кінцевому продукті.

Контроль версій - Git - GitHub.

Контроль версій, у програмній інженерії, також відомий як управління вихідним кодом , — це інструмент для контролю змін, внесених до програми. Зміни зазвичай зберігаються у вигляді числового або літерного коду, відомого як «редакція», і пов’язані з особою, яка внесла зміни, разом із позначкою часу, коли зміни були внесені. Редакцію можна порівнювати, перемотувати та поєднувати з іншими редакціями.

Git — це розподілена система контролю версій, яка широко використовується серед програмістів для відстеження змін у системі під час розробки програмною командою. Її головна особливість — це висока швидкість та підтримка паралельної роботи на кількох машинах.

GitHub — це хостинг-сервіс для розробки програмного забезпечення та систем контролю версій за допомогою Git. Він дозволяє працювати з Git, а також відстежувати помилки, надсилати запити на нові функції, керувати завданнями та створювати вікі для окремих проектів. Він також дозволяє розміщувати проекти з відкритим кодом і наразі є найпопулярнішим хостингом вихідного коду станом на листопад 2021 року, маючи понад 100 мільйонів розробників та 372 мільйони репозиторіїв, з яких 28 мільйонів є публічними).

Причина такого поєднання контролю версій полягає в тому, що з ними легко почати працювати, і вони настільки популярні та відомі, а також мають величезну базу користувачів, що для кожної проблеми, з якою виникне проблема, вже є рішення.

JetBrains Rider

JetBrains Rider — це .NET IDE з потужною підтримкою мов, що використовуються в .NET-розробці , включаючи C#, VB.NET, F#, синтаксис ASP.NET, XAML, XML, JavaScript, TypeScript, JSON, HTML, CSS та SQL, а також багатьма потужними інструментами, які допомагають вам швидко та

краще розробляти код.

Причиною, через яку JetBrains Rider було обрано як IDE для цієї роботи, була його здатність інтегруватися з Unity. Він надає багато додаткових функцій при використанні в поєднанні з Unity, включаючи швидкі виправлення, контекстні дії та перевірки, автодоповнення коду , підсвічування синтаксису, налагодження за допомогою Unity, запуск тестів Unity та багато іншого.

Ще однією причиною є його інтеграція із системами контролю версій. JetBrains Rider включає повноцінний клієнт VCS, який підтримує всі основні системи контролю версій . Це дозволяє працювати з Git безпосередньо в середовищі розробки Rider IDE без потреби в іншому програмному забезпеченні. У цьому випадку Rider IDE використовувалося для обробки контролю версій Git, який був обраний для цієї роботи.

C# — це мова програмування високого рівня, яка підтримує кілька парадигм, статичну типізацію, сильну типізацію, лексичну область видимості, імперативну, декларативну, функціональну, узагальнені, об'єктно-орієнтовані (на основі класів) та компонентно-орієнтовані дисципліни програмування.

Мову C# було обрано, оскільки вона є офіційною мовою, яку підтримує Unity. C# працює в Unity як мова сценаріїв. Скрипти вказують ігровим об'єктам, як поводитися, саме сценарії та компоненти, приєднані до ігрових об'єктів, та те, як вони взаємодіють один з одним, створюють ігровий процес.

2.4. Висновок по розділу

У другому розділі досліджено методи та інструменти розробки ігрового застосунку. Проаналізовано доцільність використання різних патернів проєктування для реалізації ігрової логіки, розглянуто архітектурні рішення та моделювання компонентів гри, а також сучасні програмні засоби розробки 2D-платформерів. Результати аналізу підтвердили, що правильний вибір архітектури та технологій забезпечує модульність, розширюваність і високу якість програмної реалізації.

РОЗДІЛ 3. ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ІГРОВОГО ЗАСТОСУНКУ

3.1. Проєктування архітектури та функціональних модулів гри

Метою дипломної роботи було створення 2D платформної гри-ранера з нескінченним циклом дій, використовуючи вищезгадані технології. У цьому розділі буде обговорено бачення та план розробки кінцевого продукту.

Дизайн цієї гри був простим, поєднуючи два жанри: «2D платформер» та «нескінченний раннер».

Платформер — це тип екшн-відеогри, головною метою якої є переміщення персонажа гравця з однієї точки в іншу в заданому середовищі. Ці ігри мають рівні з різними типами місцевості та платформами на різній висоті. Для просування гравцеві потрібно стрибати та лазити по цих платформах. Завдання полягає в маневруванні персонажа по цих платформах, щоб досягти мети. Ключовий висновок тут — «досягти мети».

Однак, нескінченний раннер не має мети. Нескінченний раннер – це тип гри, де персонаж гравця бігає без зупинки, а метою є виживання та досягнення високого балу. Гра генерує рівень або середовище безперервно та випадковим чином. Нескінченні раннери можна переглядати збоку, зверху або у 3D, але гравець завжди знаходиться на нескінченному рівні, де персонаж автоматично рухається вперед. Єдине, що гравець може контролювати, це змушувати персонажа уникати перешкод, рухаючись або натискаючи кнопки [35]. Жанр вибухнув на мобільних платформах після успіху Temple Run, серед інших популярних прикладів – Jetpack Joyride, Subway Surfer та Canabalt.

Мета цієї роботи — поєднати два жанри, натхненні грою Canabalt, щоб створити середовище, подібне до Mario, для головного персонажа, в якому він зможе переміщатися (платформерний аспект цієї гри), але без необхідності зупинятися або мати кінцівку, як у різних платформерних іграх (нескінченний раннерний аспект цієї гри), а натомість, за допомогою генерації рівнів, з

можливістю продовження, грайте та виживайте, не відстаючи від карти та не випадаючи за її межі, якомога довше.

Ігрові ресурси та спрайти

Ігровий ресурс — це цифровий компонент, який використовується у відеоіграх, такий як моделі, текстури, анімація, звуки, музика тощо. Вони створюються художниками та розробниками, щоб покращити зовнішній вигляд і звучання гри та забезпечити необхідні елементи для ігрового процесу. Ігрові ресурси покращують загальний ігровий досвід.

Спрайти — це двовимірні зображення, що використовуються як ігрові об'єкти, тоді як у 3D вони використовуються як текстури. У цій роботі, коли згадуються спрайти, мається на увазі, що вони є двовимірними спрайтами.

Для цієї роботи для створення теми всієї гри було використано безкоштовний пакет ресурсів з магазину ресурсів Unity, згаданий у розділі 2.1.1, під назвою "Pixel Adventure 1". Цей пакет складався з багатьох різних спрайтів, включаючи спрайт із 4 персонажами разом із відповідною анімацією; 12 різних комбінацій наборів плиток місцевості ; та багатьох дрібних елементів, таких як пастки, бонуси та коробки [36]. Основні елементи цього пакету ресурсів, використані в роботі, включають спрайт головного персонажа та його анімацію для створення головного персонажа гри та всіх його рухів; 12 різних наборів плиток місцевості для створення перешкод, які використовуються для методу генерації основного рівня; та набір кольорів фону для створення рухомого фону позаду гри як основного фону.

Генерація рівнів

Оскільки це гра з нескінченним раннером, генерація світів перед грою не є ідеальним вибором, оскільки технічно кажучи, немає способу згенерувати "нескінченний" або "безкінечний". Оскільки попередня генерація не є можливим вибором, іншим, більш розумним варіантом, який є найбільш підходящим методом серед жанру нескінченних раннерів, є процедурна генерація, яка також є методом вибору для генерації рівнів у цій грі [38].

3.2. Реалізація ігрового застосунку з використанням об'єктно-орієнтованих патернів

У цьому підрозділі обговорюється розробка гри після вибору всіх технологій та варіантів ігрового дизайну, а також методів її реалізації. У цьому розділі передбачається, що всі необхідні технології вже встановлені та налаштовані.

Щоб створити гру за допомогою Unity Engine, спочатку потрібно створити проєкт у Unity Hub, який є основним застосунком для доступу до екосистеми Unity, зокрема для надання доступу до проєктів користувача та створення нових.

У Unity Hub просто натисніть «Новий проект», щоб створити новий проект. З’явиться нова панель шаблонів, де користувач зможе вибрати шаблон, з яким хоче працювати, і оскільки метою є створення 2D-гри, обраним шаблоном буде «2D core». Цей шаблон надасть нам усе необхідне для початку розробки 2D-гри [39].

На цьому кроці також потрібно ввести назву проекту рисунок 3.1.

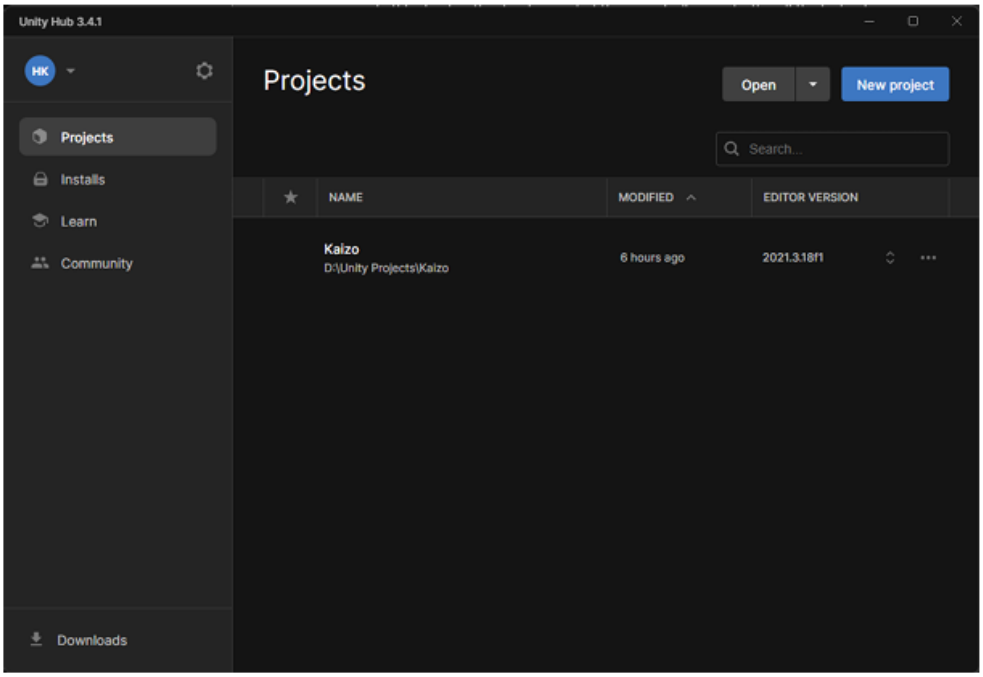


Рисунок 3.1 - Головний інтерфейс Unity Hub

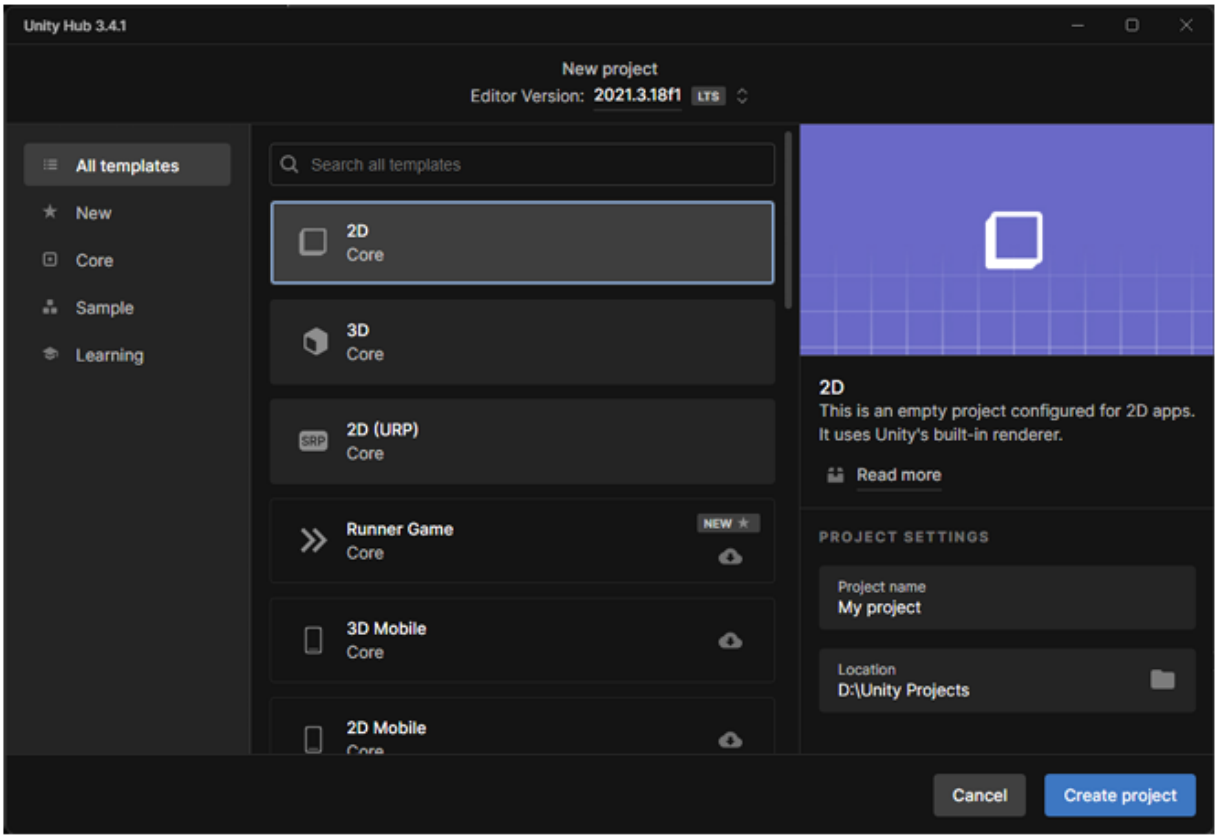


Рисунок 3.2 - Інтерфейс вибору шаблону нового проєкту Unity Hub.

Структура папок проекту

Стандартна структура папок проєкту Unity складається з кількох різних папок, кожна з яких має своє призначення, але в самому Unity фактично використовуються та відображаються лише дві з цих папок, а саме Asset та Package.

Папка Asset (Активи) – це основна папка, в якій кожен розробник ігор проводитиме найбільше часу. Тут зберігаються всі ресурси, які розробник ігор використовуватиме для своєї гри, від імпортованих або створених моделей, спрайтів персонажів, аудіофайлів, фонових зображень до скриптів та префабів, які розробники писатимуть та створюватимуть [38]. Після створення нового проекту ця папка порожня.

Папка пакета — це місце, де зберігаються всі пакети, встановлені разом із проектом, і ними керує менеджер пакетів Unity. Розробник може ігнорувати

цю папку, оскільки всі необхідні зміни для пакетів будуть виконані за допомогою менеджера пакетів Unity.

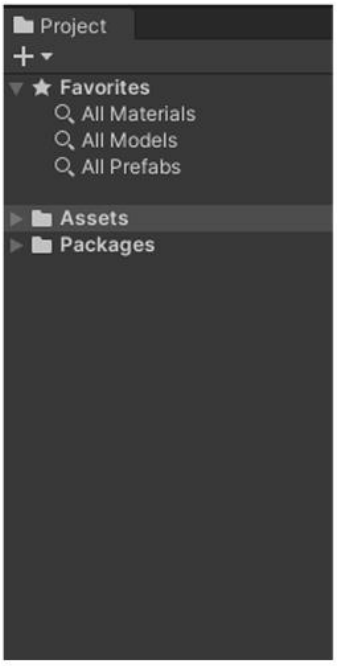


Рисунок 3.3 - Початкова структура папок проекту

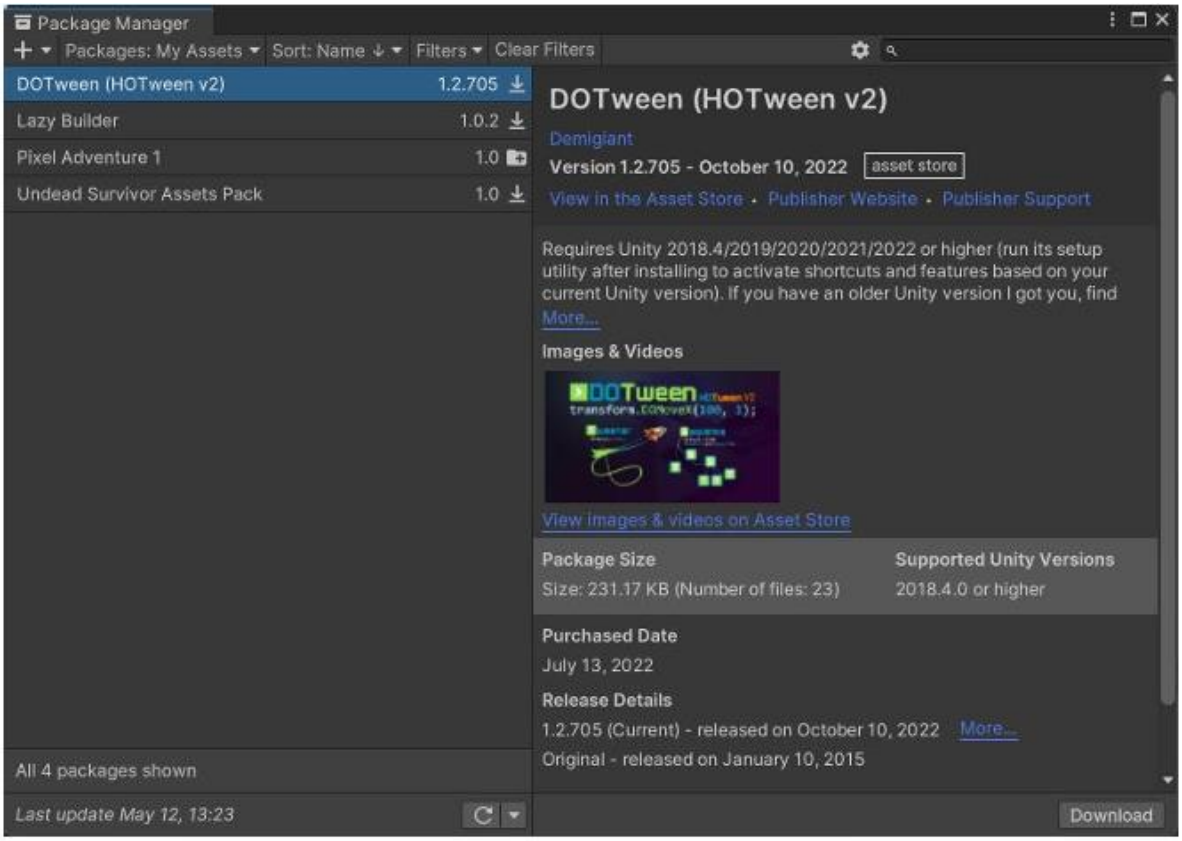


Рисунок 3.4 - Приклад інтерфейсу менеджера пакетів Unity

Імпорт активів

Як обговорювалося вище, пакет ресурсів, який використовується в цьому проєкті, – це пакет «Pixel Adventure 1» від Pixel Frog. Це безкоштовний пакет ресурсів, який можна легко імпортувати з менеджера пакетів Unity, або додавши пакет у сховище ресурсів Unity до свого облікового запису Unity та імпортувавши їх з розділу «Мої ресурси», або ж здійснивши пошук за назвою безпосередньо в менеджері пакетів Unity [39].

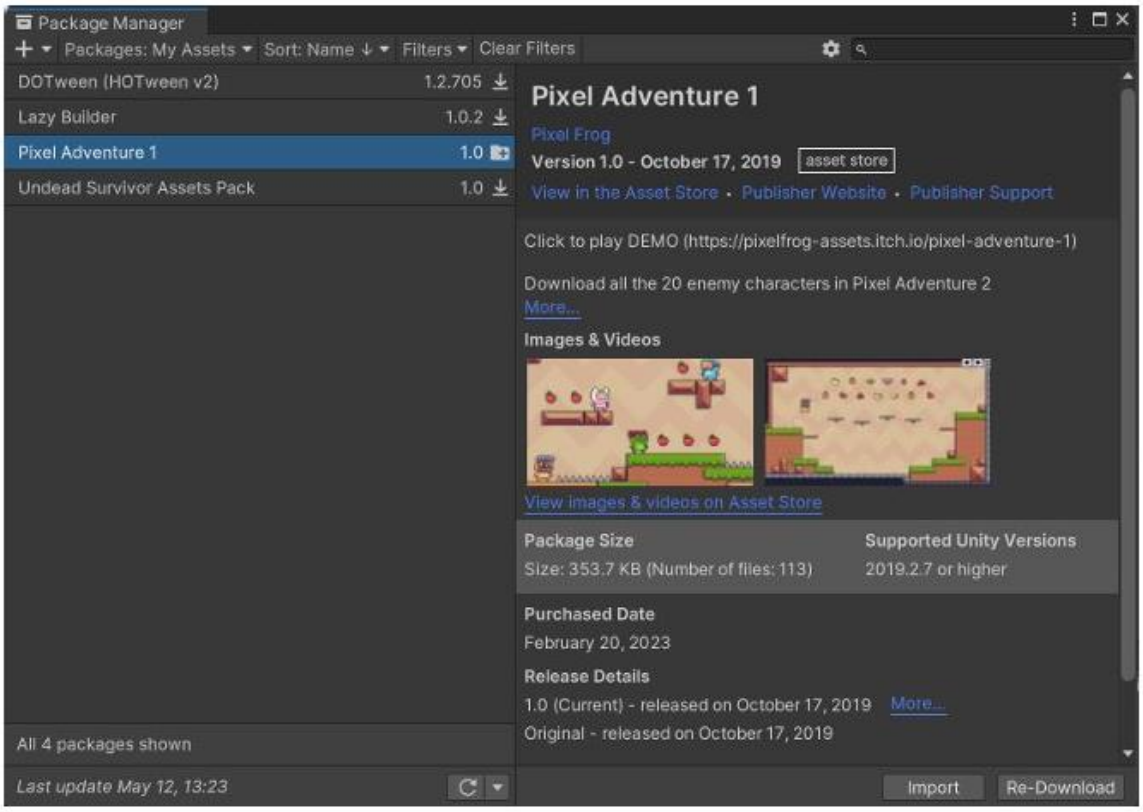


Рисунок 3.5 - Імпорт гри «Pixel Adventure 1»

Після імпорту пакету ресурсів має бути створена нова папка під назвою "Pixel Adventure 1", яка містить усі дані з самого пакету ресурсів, готові до використання.

Розробка гри

Початком кожної розробки нової гри є створення нової сцени. У цьому випадку спочатку створюється основна ігрова сцена, де відбуваються всі дії. Спочатку створюється нова папка під назвою «Сцени» для структурування, а

потім нова Далі створюється сцена всередині неї під назвою "MainScene". Тепер у нас є новий шаблон сцени для початку створення гри.

Головний персонаж спочатку реалізується шляхом створення нового ігрового об'єкта під назвою Гравець, клацнувши правою кнопкою миші вікно Ієрархії та вибравши «Створити новий».

Для відображення спрайту персонажа у реальній грі потрібен компонент під назвою Sprite Renderer. Просто виберіть гравця, натисніть «Додати компонент» у вікні інспектора та виберіть «Sprite Renderer». Потрібен спрайт персонажа з Pixel Adventure 1 , просто перетягніть потрібного персонажа в розділ «Спрайти» на Sprite Renderer, і тепер у нас є персонаж, готовий до роботи [40].

Повторіть крок з іншим компонентом під назвою Rigidbody 2D, який надає персонажу можливість взаємодіяти з гравітацією в движку Unity, дозволяючи йому рухатися, стрибати та падати. Для виявлення зіткнень також потрібен 2D-коробчастий колайдер.

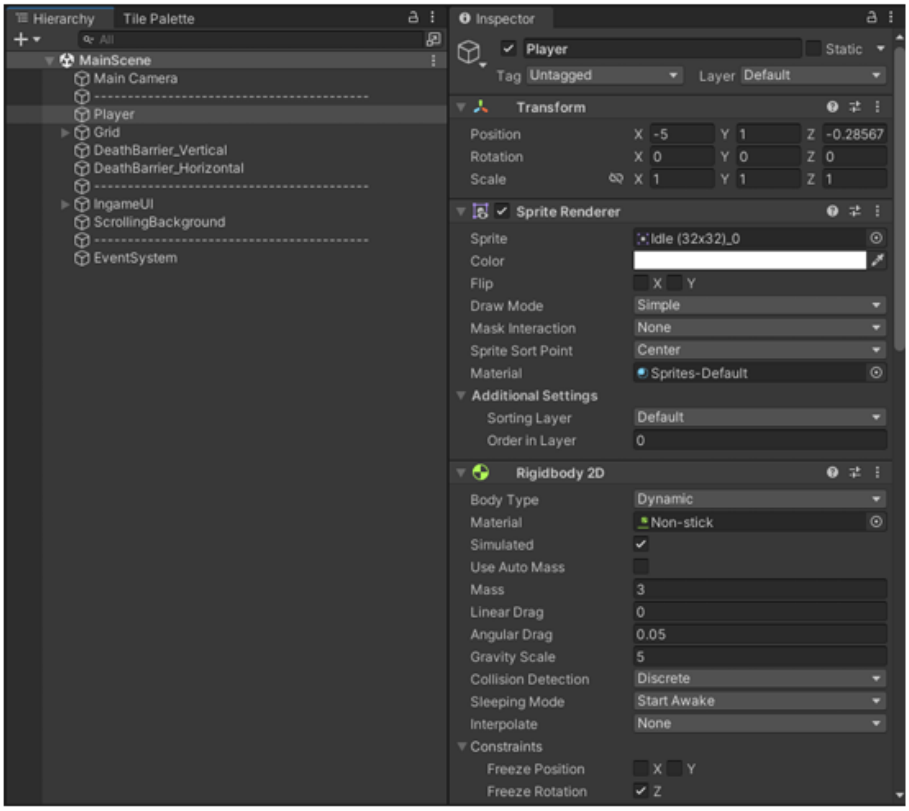


Рисунок 3.6 - Ігровий об'єкт гравця та його компоненти

Персонажа створено, але він не має можливості рухатися та стрибати. Рух персонажа додається до нього за допомогою скрипта, який приєднується до персонажа як компонент. Це дозволяє персонажу рухатися, коли користувач натискає клавіші зі стрілками, та стрибати, коли користувач натискає пробіл. Скрипт також дозволяє змінювати анімацію гравця на певну, що відповідає тому, що робить персонаж у цей момент.

```
using UnityEngine;
using UnityEngine.UI;

public class PlayerMovement : MonoBehaviour {

    public float speed = 5f;
    public float jumpForce = 10f;
    [SerializeField] private Button replayButton;
    [SerializeField] private Button backButton;
    private bool isGrounded;
    private Rigidbody2D rb;
    private BoxCollider2D col;
    private Animator anim;

    void Start() {
        rb = GetComponent<Rigidbody2D>();
        col = GetComponent<BoxCollider2D>();
        anim = GetComponent<Animator>();
    }

    void Update() {
        float moveInput = Input.GetAxis("Horizontal");

        rb.velocity = new Vector2(moveInput * speed, rb.veloc-
ity.y);

        anim.SetFloat("Speed", Mathf.Abs(moveInput));

        isGrounded = IsGrounded();
        anim.SetBool("IsJumping", !isGrounded);

        if (isGrounded && Input.GetKeyDown(KeyCode.Space)) {
            rb.velocity = new Vector2(rb.velocity.x, jumpForce);
        }
    }
```

Рисунок 3.7 - Сценарій руху гравця

Проблема під час розробки полягає в тому, що якщо гравець постійно

натискатиме клавішу пробілу , персонаж постійно стрибатиме в повітрі та може злетіти за межі карти. Щоб запобігти цьому, було створено функцію під назвою isGrounded() за допомогою raycast, яка відстежує, чи торкається персонаж чогось під собою, у цьому випадку землі. Якщо raycast нічого не виявляє під персонажем, що означає, що персонаж тепер знаходиться в повітрі, ми вимикаємо здатність персонажа стрибати, і навпаки.

Перешкоди створюються за допомогою системи Grid та Tilemap від Unity. Компонент Grid – це інструмент, який допомагає вирівнювати ігрові об'єкти, такі як плитки, у вибраному макеті. Він приймає позиції комірок сітки та перетворює їх у відповідні координати для ігрового об'єкта. Потім компонент Transform перетворює ці локальні координати на глобальні або світові просторові координати.. Система Tilemap від Unity – це функція, яка керує та зберігає ресурси плиток для створення 2D-рівнів. Вона спрощує процес створення та ітерації дизайну рівнів в Unity, полегшуючи проектування та зміну ігрових рівнів.

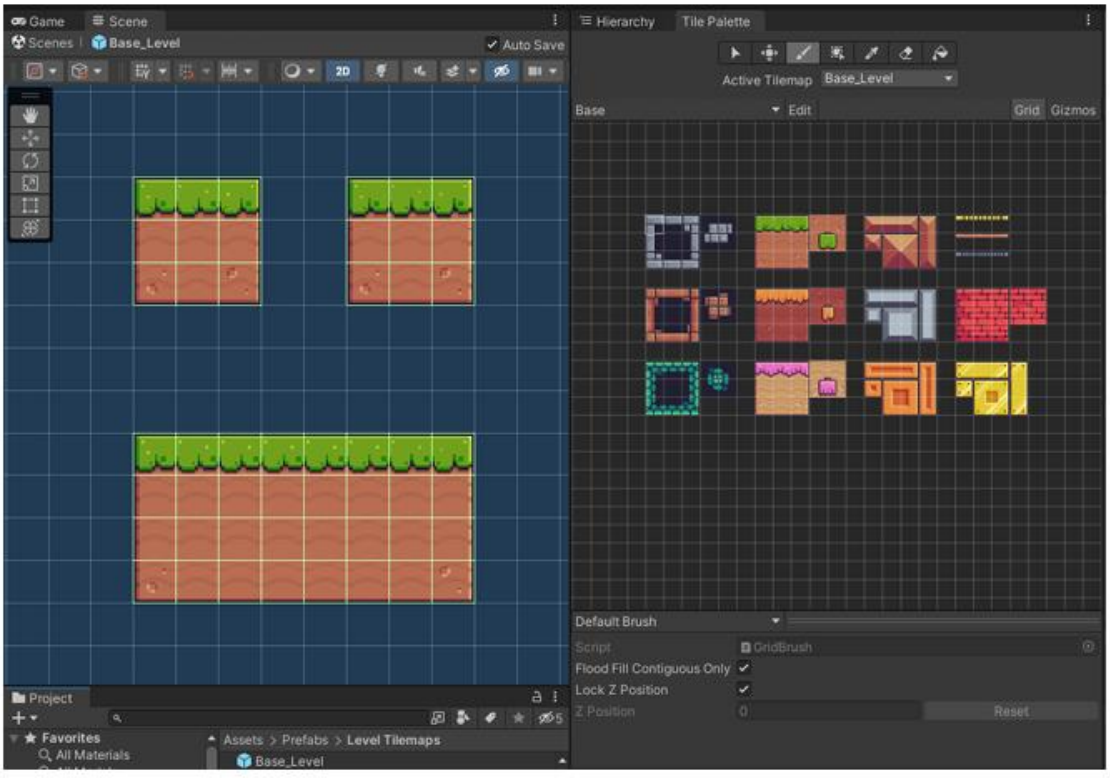


Рисунок 3.8 - Базова перешкода, розфарбована плитковою палітрою.

Плитки, що використовуються в цій грі, включені до набору Pixel Adventure 1 з 12 різними текстурами, які пізніше використовуються для розрізнення трьох типів перешкод, обговорених вище. Вони використовуються для створення нової палітри плиток для створення перешкод.

Для створення перешкод, які будуть постійно генеруватися для цієї гри, використовується метод використання різних попередньо підготовлених частин ігрового світу та їх зшивання разом, умовно, для створення ілюзії нескінченно довгого світу.

Кожен префаб перешкоди являє собою область розміром 8 x 10, де 8 – це ширина, а 10 – висота перешкоди. Перешкода також матиме кілька властивостей, таких як тип перешкоди (Кімната, Лабіринт або Небо), початкове положення та кінцеве положення.

В грі доступні 3 типи перешкод :

• Кімната: Кімнатні перешкоди нагадують невелику кімнату, дуже спрощену смугу перешкод, яку можна побачити в багатьох подібних 2D-платформерах, таких як Mario. Уявіть собі кімнатні перешкоди як невелику частину якогось випадкового світу Mario. Рівні кімнат дуже прості для навігації та не вимагають від гравця особливих навичок.

• Лабіринт: Перешкоди в лабіринті нагадують лабіринт, що характеризується кількома можливими шляхами, вузькими просторами для навігації та деякими заблокованими стежками. Рівні лабіринтів, мабуть , є - найскладнішим типом кімнат у Kaizo, вимагаючи від гравця розв'язати лабіринт на місці, щоб знайти правильний шлях, а через велику щільність розташування перешкод від гравця також вимагає багато маневрування.

• Небо: Небесна перешкода дуже схожа на кімнатну перешкоду, але замість неї немає допоміжної платформи, і це створює ще одну програшну умову для гри, а саме падіння з карти. Небесні рівні – це перешкода середньої складності в цій грі, яка вимагає від гравця трохи більше навичок пересування, щоб не впасти та не завершити гру.

Серед трьох типів перешкод, кожен з них має свій унікальний набір початкових та кінцевих позицій: Нижня - Середня - Верхня.

- Нижня позиція: стосується нижньої позиції (початкової або кінцевої). Нижня позиція розташована на 3-й та 4-й позиціях знизу вгору на межі перешкоди висотою 10 дюймів.
- Середина: стосується середньої позиції (початкової або кінцевої). Середня позиція розташована на 5-й та 6-й позиціях знизу вгору від перешкоди висотою 10.
- Верх: стосується нижньої позиції (початкової або кінцевої). Верхня позиція розташована на 7-й та 8-й позиціях знизу вгору на межі перешкоди висотою 10 дюймів.

Поєднавши всі ці властивості, було створено 27 різних та унікальних перешкод, щоб зробити гру максимально випадковою та унікальною, а також щоб кожна нова гра, в яку гравець грає, ніколи не відчувалася такою ж, як коли він грав у цьому ігровому світі раніше. Щоб розрізнити їх, для кожної перешкоди було дано різні назви, і вони досить прості: тип перешкоди (Кімната, Лабіринт або Небо) йде далі за її початковою позицією (Нижня, Середня або Верхня) і закінчується кінцевою позицією. Було створено ще одну папку під назвою «Префаби», а всередині неї — «Лівневі карти» для зберігання всіх префабів перешкод для зручності доступу пізніше.

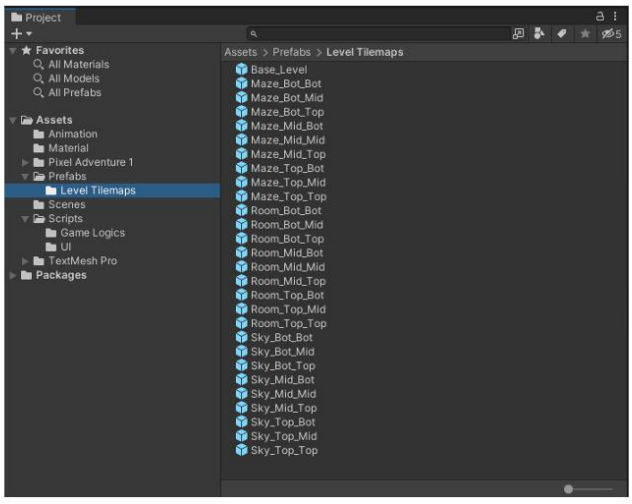


Рисунок 3.9 - Збірні світові перешкоди

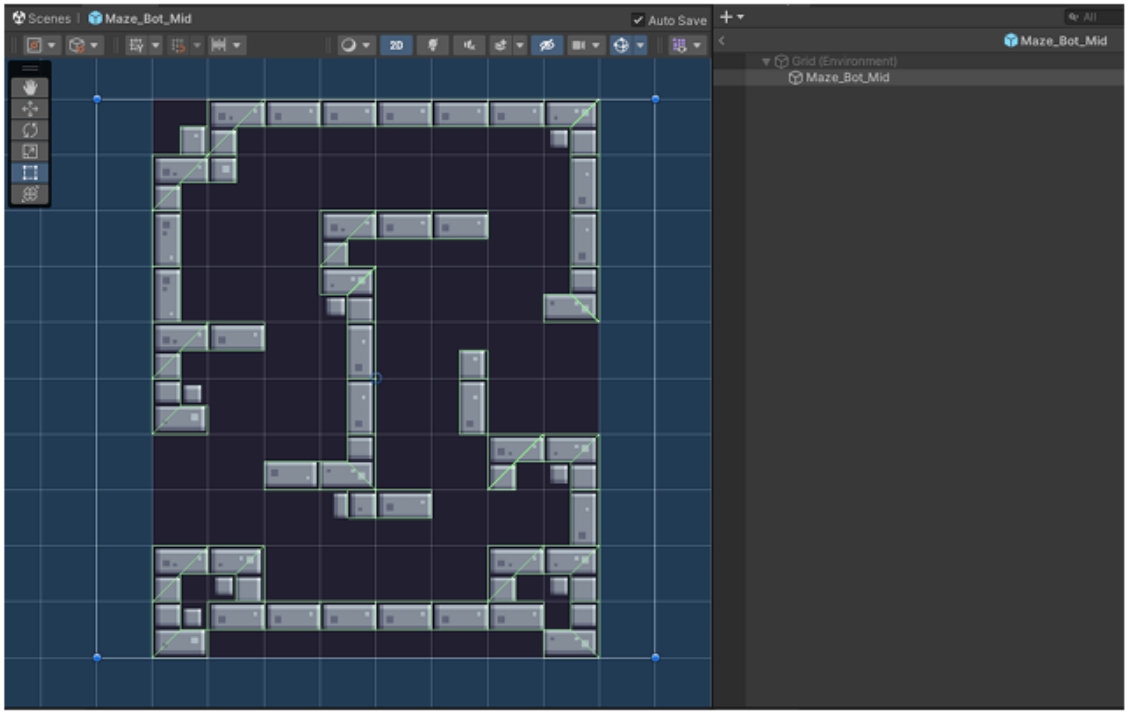


Рисунок 3.10 - Приклад перешкоди, лабіринт з нижнім отвором та середнім кінцем

Кожній із перешкод додатково додано компонент під назвою Tilemap Collider 2D, який допомагає їй взаємодіяти з фізичною системою, а також взаємодіяти та стикатися з різними ігровими об'єктами, у цьому випадку з гравцем. Без цього компонента всі створені вище перешкоди були б не чим іншим, як картинкою.

Кожна плитка місцевості має свою різну текстуру, тому для них потрібна власна фізична форма. Без неї всі плитки матимуть власний колайдер, який займатиме всю плитку, і жодна частина перешкод не буде рухомою та не дозволить переміщатися через неї. Щоб досягти цього, перейдіть до оригінального ресурсу, де зберігаються плитки місцевості, і відкрийте його редактор спрайтів. Відкриється редактор спрайтів цієї текстури місцевості, і у верхньому лівому куті виберіть "Власна фізична форма". Тепер кожна окрема плитка готова до нанесення її фізичної форми.

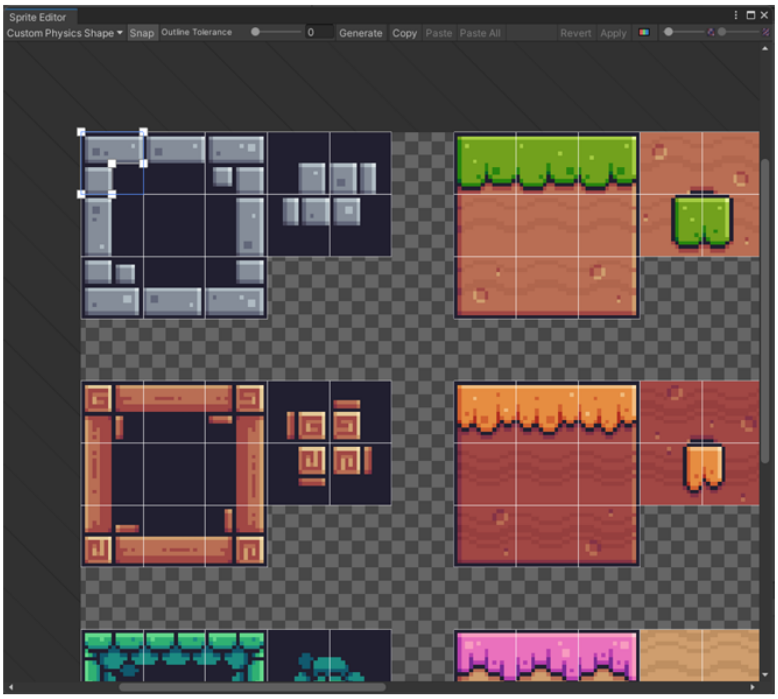


Рисунок 3.11 - Приклад намальованої за допомогою редактора спрайтів фізичної форми

Варто зазначити, що темна середня частина ландшафту просто являє собою порожній простір, але якщо її залишити недоторканою, Unity автоматично заповнить фізичну форму як цілу плитку. Щоб запобігти цьому, потрібно згорнути її фізичну форму в одну точку.

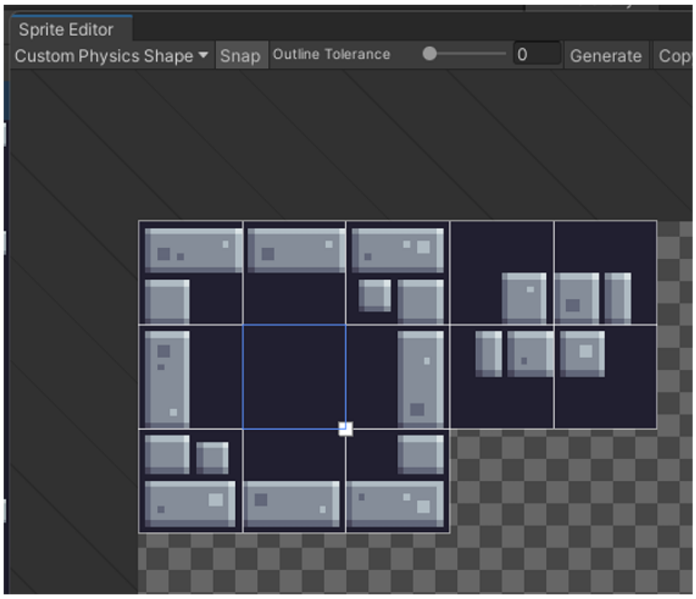


Рисунок 3.12 - Згортання фізичної форми порожньої плитки в точку

У цій грі використовується той самий метод руху світом, що й Flappy Bird, де замість того, щоб птах рухався вперед по трубах, він фіксований у центрі екрана, а труба рухається назад, створюючи ілюзію руху птаха вперед. У грі використовуються ті ж методи руху перешкод назад, але гравець не фіксований, а може вільно рухатися, поки всі рухаються назад, що дозволяє гравцеві долати деякі перешкоди, що вимагає руху назад або будь-якої зміни положення.

Скрипт під назвою "Moveable" створюється та додається як компонент майже до кожного ігрового об'єкта (включно з гравцем, про який буде сказано пізніше). Скрипт Moveable був написаний для переміщення ігрового об'єкта, до якого він прикріплений, ліворуч з певною швидкістю, а також для надання йому можливості самознищення, коли він вже проходить екран ліворуч, оскільки ця перешкода вже готова і більше не використовується. Причина, чому сам гравець, хоча й має власний рух, також отримує компонент Moveable, полягає в тому, що якщо у нього немає цього компонента, він стоятиме нерухомо на одному місці, поки все рухатиметься ліворуч під його ногами, доки попереду не з'явиться колайдер, а потім його відштовхне назад, але нам це не потрібно, оскільки якщо гравець стоїть нерухомо, ми хотіли б, щоб він також рухався разом із землею під ним.

```
using UnityEngine;

public class Moveable : MonoBehaviour
{
    public float speed;

    private void Update()
    {
        // Move the pipe to the left
        transform.position += Vector3.left * speed * Time.del-
taTime;

        // Check if pipe is offscreen
        if (transform.position.x < -20f)
        {
            // Destroy the pipe
            Destroy(gameObject);
        }
    }
}
```

Рисунок 3.13 - Переміщуваний скрипт

Тепер, коли все готове, останнє, що потрібно зробити, це фактичне створення перешкод.

Створюється новий порожній GameObject та скрипт під назвою WorldSpawner для обробки всієї логіки створення перешкод. Скрипт включає:

- Усі префаби Перешкод поділено на 3 масиви: botOpeningPrefabs, midOpeningPrefabs та topOpeningPrefabs. Як випливає з їхніх назв, вони містять префаби, початкова позиція яких відповідає назві масиву. Оскільки існує 27 унікальних Перешкод, ми знаємо, що в кожному масиві є 9 Перешкод.
- Число з плаваючою комою під назвою spawnIntervals, яке визначає частоту появи перешкод.
- Функція оновлення, яка викликається кожного кадру для перевірки інтервалів таймера появи та викликає функції, що створюють екземпляри Перешкод, у цей інтервал появи.
- Змінна GameObject під назвою currentObstacles, яка містить інформацію про останню створювану перешкоду, та змінна під назвою currentObstacleEndPosition, яка містить інформацію про кінець цієї перешкоди.

Три функції під назвою InstantiateBotOpening, InstantiateMidOpening та InstantiateTopOpening створюють екземпляри випадкових перешкод з масивів перешкод, які мають певну позицію відкриття.

Логіка появи перешкод така:

На початку гри завжди створюється випадкова перешкода, початкова позиція якої знаходиться посередині, потім інформація про неї, включаючи кінцеву позицію, зберігається. Під час наступного інтервалу створення, залежно від кінцевої позиції останньої перешкоди, буде викликана відповідна функція для створення нової перешкоди з початковою позицією така ж, як кінцева позиція останньої перешкоди, щоб переконатися, що вони з'єднані. Потім її інформація зберігається, і цикл продовжується до кінця гри.

Тепер, коли гра працює, останнім кроком є перевірка на наявність програшних умов для гравця. У цій грі гравець програє, якщо не встигає за

колією та занадто сильно відстає, або якщо він впаде за межі світу через одну з численних небесних перешкод.

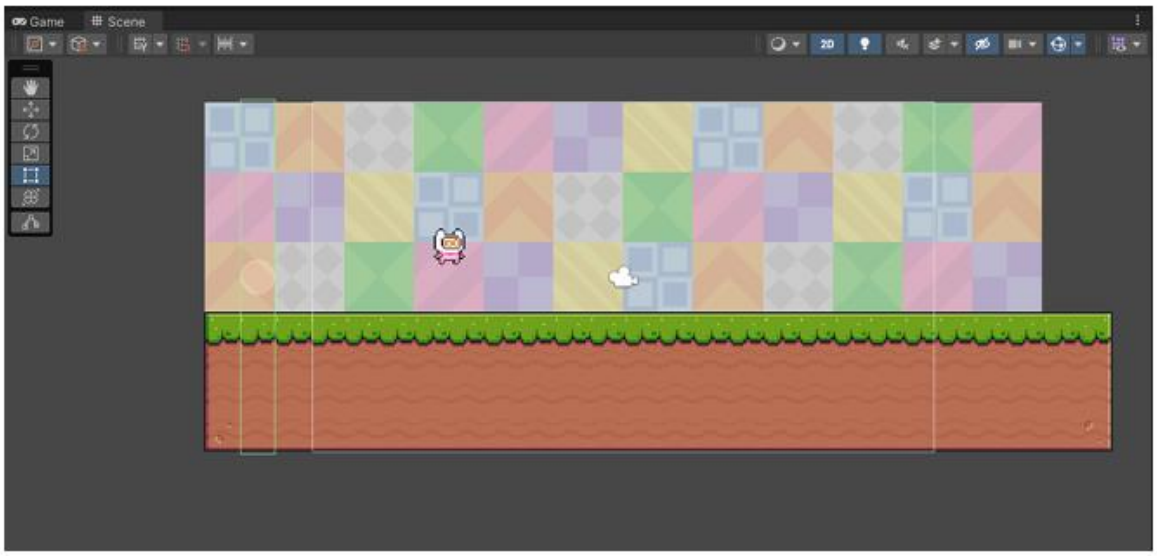


Рисунок 3.14 - Лівий бар'єр смерті (зеленим кольором)

Ми можемо досягти цього, створивши два окремих колайдери з власним тегом під назвою "Бар'єр", у крайній лівій та нижній частинах екрана, але з зовнішньої сторони. Потім колайдер встановлюється на "Існує тригером", що означає, що при дотику він надсилатиме тригерний сигнал до ігрового об'єкта, який його запустив, у цьому випадку, гравця.

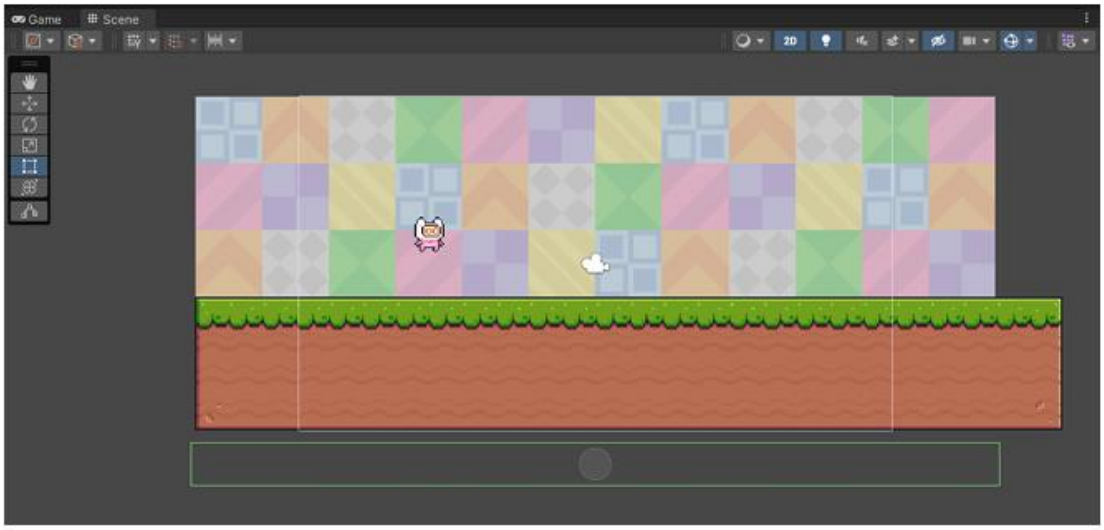


Рисунок 3.15 - Нижній бар'єр смерті (зеленим кольором)

До скрипта гравця додано нову функцію OnTriggerEnter2D, яка визначає, чи спрацьовує тригер під назвою "Бар'єр", і якщо так, то гра зупиняється, якщо гравець занадто сильно відстає або опускається нижче межі світу.

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.tag == "Barrier")
    {
        GameManager.Instance.EndGame();
    }
}
```

Рисунок 3.16 - OnTriggerEnter2D гравця

Скрипт посилається на інший скрипт під назвою GameManager, який створений для обробки загальних ігрових подій, таких як завершення гри та пізніше створення системи інтерфейсу користувача, для обробки натискань кнопок для запуску гри та повернення до головного меню.

Наразі єдине завдання скрипта GameManager — заморозити гру після смерті гравця, що сигналізує про її завершення. Змінивши часову шкалу на нуль, ми можемо досягти цього ефекту.

```
public void EndGame()
{
    Time.timeScale = 0;
}
```

Рисунок 3.17 - Заморожування гри за допомогою шкали часу

Після цього основна гра завершена та готова до гри.

Хоча основний ігровий процес завершено, гра все ще не є повністю функціональною, оскільки щойно програма запускається, гра починається негайно.

Спочатку ми можемо створити ще одну сцену для головного меню під назвою MainMenu. Використовуючи цю нову сцену, ми можемо створити полотно (Clan) та додати до нього різні компоненти інтерфейсу користувача, такі як тексти, кнопки та зображення, щоб створити простий інтерфейс головного меню.

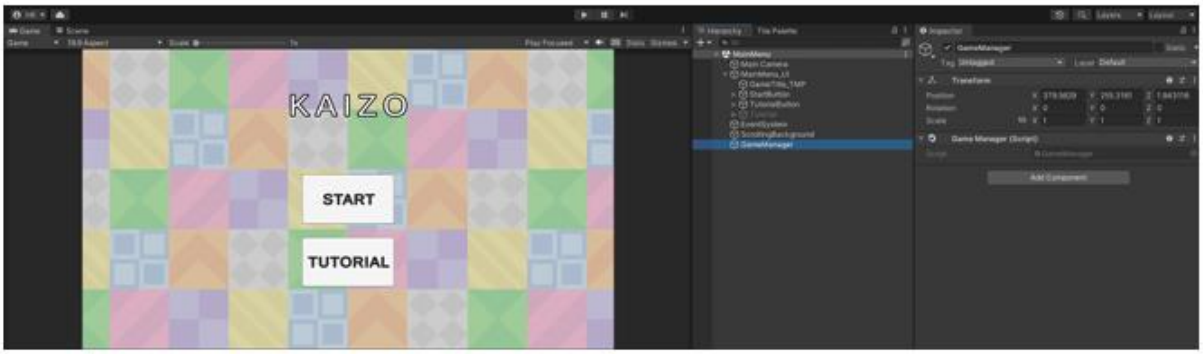


Рисунок 3.18 - Сцена головного меню

Для обробки кнопок та подій створюється скрипт GameManager, як згадувалося вище, і оскільки GameManager контролює важливі функції для основних подій, таких як зміна сцен, вхід у гру та завершення гри.

Скрипти GameManager дотримуються шаблону Singleton. Шаблон Singleton — це шаблон проектування програмного забезпечення, який обмежує створення екземплярів класу одним екземпляром . Один із добре відомих шаблонів проектування «Банда чотирьох», який описує Як вирішувати повторювані проблеми в об'єктно-орієнтованому програмному забезпеченні, шаблон корисний, коли для координації дій у всій системі потрібен лише один об'єкт. Більш конкретно, шаблон Singleton дозволяє об'єктам гарантувати наявність лише одного екземпляра , забезпечувати легкий доступ до цього екземпляра та керувати його створенням .

Його часто використовують у розробці ігор для керування основними системами, такими як введення, аудіо або мережа, які потрібно використовувати спільно для кількох сцен і скриптів.

Скрипт GameManager під час першого запуску викликає функцію

DontDestroyOnLoad, яка, як і обіцяє, не знищує себе під час завантаження іншої сцени, що допомагає об'єкту GameManager зберігатися навіть після зміни сцени на головну GameScene. Цей крок необхідний , оскільки GameManager також містить функції для обробки завершення гри та завантаження назад до MainMenu, що може відбутися лише в GameScene. Оскільки GameManager запускається в MainMenu, нам потрібен спосіб для GameManager зберігатися в іншій сцені після завантаження.

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour
{
    public static GameManager Instance { get; private set; }

    //private bool gameEnded = false;

    void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        else
        {
            Destroy(gameObject);
        }
        DontDestroyOnLoad(gameObject);
    }

    public void EndGame()
    {
        Time.timeScale = 0;
    }

    public void BackToMainMenu()
    {
        Time.timeScale = 1;
        SceneManager.LoadScene(0);
    }

    public void StartGame()
    {
        Time.timeScale = 1;
        SceneManager.LoadScene(1);
    }

}
```

Рисунок 3.19 - Скрипт GameManager

Також у ігровій сцені додано полотно для відображення часу, протягом якого гравець вижив, а також для відображення кнопок для гравця, щоб він міг

переграти гру або повернутися до головного меню після смерті. До полотна прикріплено скрипт під назвою GameUI для обробки всієї логіки, пов'язаної з інтерфейсом користувача, включаючи підрахунок часу, проведеного гравцем у житті, та навігацію за допомогою кнопок.

```
public class GameUI : MonoBehaviour
{
    private float point;
    [SerializeField] private TextMeshProUGUI pointText;
    [SerializeField] private Button replayButton;
    [SerializeField] private Button backButton;
    void Start()
    {
        point = 0;
        replayButton.gameObject.SetActive(false);
        backButton.gameObject.SetActive(false);
        replayButton.onClick.AddListener(() =>
        {
            GameManager.Instance.StartGame();
        });
        backButton.onClick.AddListener(() =>
        {
            GameManager.Instance.BackToMainMenu();
        });
    }

    // Update is called once per frame
    void Update()
    {
        point += Time.deltaTime;
```

Рисунок 3.20 - Скрипт GameUI

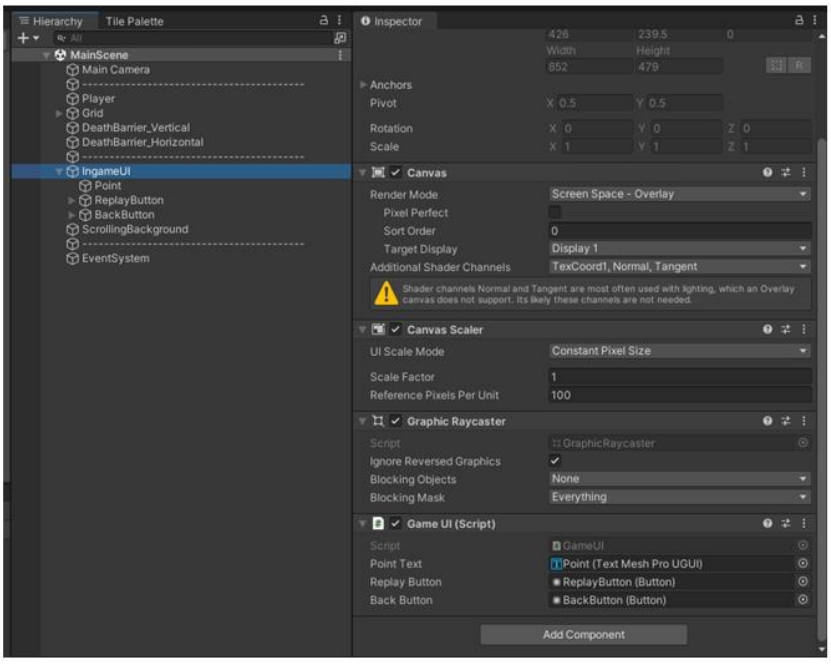


Рисунок 3.21 - Ієрархія ігрового інтерфейсу

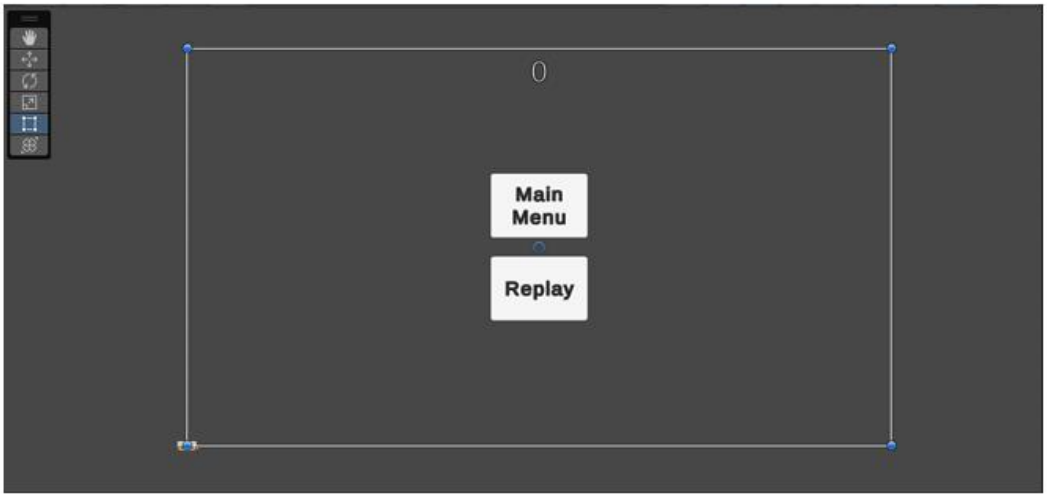


Рисунок 3.22 - Інтерфейс користувача в грі

Кнопки MainMenu та Replay пов'язані з функціями в GameManager, і оскільки GameManager є Singleton, як обговорювалося раніше, вони можуть отримати доступ до відповідної функції GameManager безпосередньо через її екземпляр. На початку роботи кнопки MainMenu та Replay будуть вимкнені, не відображатимуться, оскільки ми хочемо, щоб вони відображалися лише тоді, коли гравець фактично помирає. Для цього нам потрібно посилатися на ці дві кнопки всередині скриптів Player, і коли вони помирають, що також відбувається під час виклику OnTriggerEnter2D, ми можемо знову ввімкнути кнопку m.

```
[SerializeField] private Button replayButton;
[SerializeField] private Button backButton;


private void OnTriggerEnter2D(Collider2D other)
{
    if (other.gameObject.tag == "Barrier")
    {
        GameManager.Instance.EndGame();
        replayButton.gameObject.SetActive(true);
        backButton.gameObject.SetActive(true);
    }
}
```

Рисунок 3.23 - Оновлений скрипт програвача для обробки інтерфейсу користувача

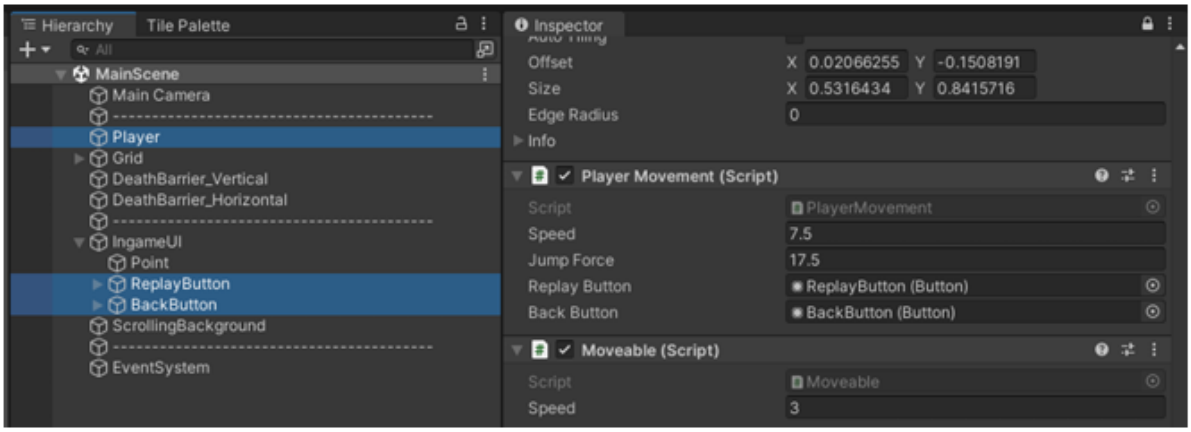


Рисунок 3.24 - Посилання на кнопки інтерфейсу користувача у скрипті програвача

3.3. Результати і обговорення

У цьому підрозділі представлені кінцеві результати різних завдань, виконаних протягом усього проєкту. Результати включають тести, проведені під час графічного дизайну, тестування кольорів на наявність дальтонізму, тестування продуктивності гри та різних схем керування.

Під час розробки графічних аспектів проєкту однією з найважливіших частин було коректне експортування SVG-зображень у формат PNG, а потім їх імпорт у Unity без втрати якості.

В Unity, якщо імпортувати зображення дуже малого розміру, наприклад, 64 x 64 пікселів , а потім відобразити його у відповідному розмірі, воно виглядатиме пікселізованим. З іншого боку, якщо використовувати зображення дуже великого розміру, наприклад, 4096 x 4096 пікселів, Unity створюватиме артефакти, такі як аліасинг, під час зменшення його розміру для відображення, оскільки фільтри, що використовуються в цьому процесі, створені для швидкого, але не точного виконання. З цієї причини я провів багато тестів з різними зображеннями різних розмірів і виявив, що найкраще виглядають зображення приблизно 512 x 512 пікселів.

Після кількох тестів з використанням різних кольорів та врахуванням

трьох типів фільтрів для дальтоніків, я вибрав найкращі кольори, які дозволяють користувачеві розрізняти Гравця та його снаряди, а також різні об'єкти, які він має знищити або уникнути, щоб перемогти. Ці об'єкти включають Ворогів та їхні снаряди, Лазерні бар'єри, Боса та його зброю, снаряди та лазери.

Остаточну конфігурацію кольорів, отриману в результаті цього етапу тестування, можна побачити на рисунку 3.25, де показано відмінності між програвачем та Вороги в грі. Слід підкреслити, що найважливішим аспектом тут є те, що користувач повинен вміти розрізняти союзних та ворожих юнітів: союзні юніти мають синій колір, тоді як ворожі юніти мають інші кольори. Зауважте, що кольорова палітра, яка нарешті використовується у відеогрі.

Оскільки це проста 2D-гра, дуже важливим аспектом є те, що її можна запускати на широкому спектрі обладнання без суттєвої втрати продуктивності. Загалом кажучи, продуктивність в іграх вимірюється в кадрах за секунду (fps), що є частотою, з якою ігрові зображення відображаються на екрані. Цей показник називається частотою кадрів, і його ідеальне значення в іграх становить 60 fps. При такій частоті кадрів ігровий процес відчувається плавним та реагує швидко.

Іншою важливою концепцією з точки зору продуктивності в грі є Vsync. Вона синхронізує кадри , що надсилаються графічним процесором, з частотою оновлення екрана та обмежує її на максимальному значенні, яке на більшості екранів становить 60 Гц. Реальна продуктивність у грі досягається з вимкненою цією опцією, і чим вища кількість кадрів в секунду, тим краще. Однак користувачі зазвичай віддають перевагу її ввімкненню, оскільки інакше на екрані можуть з'являтися деякі артефакти, такі як розриви екрана.

Усі тестування проводилися за допомогою MSI Afterburner, програмного забезпечення, яке надає детальну інформацію та статистику продуктивності гри. Тестування проводилося на різних операційних системах, включаючи різні версії Windows (7, 8 та 10), а також Linux (Ubuntu та Debian). У цьому сенсі слід зазначити, що продуктивність на всіх розглянутих операційних системах була майже однаковою. Що стосується протестованих комп'ютерів, то всі вони мали

процесори Intel і видавали в середньому 60 кадрів/с з увімкненою опцією Vsync, що є найкращим можливим результатом для екранів з частотою оновлення 60 Гц. У таблиці 6.1 наведено додаткові дані про тестування продуктивності гри та використання ресурсів на різних апаратних конфігураціях. Судячи з частоти кадрів з вимкненою опцією Vsync, отриманої продуктивності більш ніж достатньо, щоб насолоджуватися грою.

Таблиця 3.1

Продуктивність гри на різних конфігураціях обладнання

Процесор (CPU)	ОЗП (RAM)	Відеокарта (GPU)	Використання пам'яті: ОЗП (RAM)	Використання пам'яті: Відеопам'ять (VRAM)	Роздільна здатність	Кадрова частота (Vsync off)
Core i7-5930K	16 ГБ	NVIDIA GTX 980 TI 6GB	90 МБ	444 МБ	2560x1440	1300 кадр/с
Core i7-2670QM	8 ГБ	NVIDIA GT 540M 2GB	100 МБ	167 МБ	1366x768	500 кадр/с
Core i5-M460	4 ГБ	ATI Mob. Radeon HD5470 1GB	108 МБ	256 МБ	1600x900	425 кадр/с
Q6600	4 ГБ	ATI Radeon HD7000 1GB	105 МБ	188 МБ	1400x900	750 кадр/с
Core i5-4570	3.5 ГБ	Intel HD Graphics 4600 512 MB	255 МБ	240 МБ	2048x1152	150 кадр/с

Остаточні елементи керування для Xbox 360, Xbox One або будь-якого сумісного ігрового контролера XInput були налаштовані наступним чином: лівий стік використовується для руху гравця та навігації по меню; кнопки A та B встановлені як основні та додаткові дії стрільби; пункти меню можна виконувати за допомогою кнопки A; кнопка «Старт» використовується для паузи та відновлення гри. Усі елементи керування геймпада проілюстровано на рисунку 6.2.

Що стосується керування на клавіатурі, то воно детально описане наступним чином: клавіші зі стрілками призначені для переміщення гравця та навігації по меню; клавіші Ctrl та Alt призначені для основних або додаткових дій стрільби; а клавіша Esc використовується для паузи та відновлення гри. Усі

стандартні елементи керування клавіатурою можна побачити на рисунку 6.3. Крім того, користувач може використовувати клавіші (W, A, S, D) для переміщення гравця та меню.



Рисунок 3.25 - Розташування кнопок на ігровому контролері Xbox 360 за замовчуванням

Якщо користувачам не подобаються елементи керування за замовчуванням, вони завжди можуть змінити їх перед початком гри в опціях, які Unity надає за замовчуванням під час запуску гри. Це можливо завдяки спеціальному набору попередньо визначених дій у Unity, таких як, наприклад: Горизонтальна та Вертикальна вісь надають доступ до клавіш навігації; тоді як дії «Вогонь1» та «Вогонь2» надають доступ до інших функцій, які в цьому проєкті використовуються для основної та додаткової стрільби.

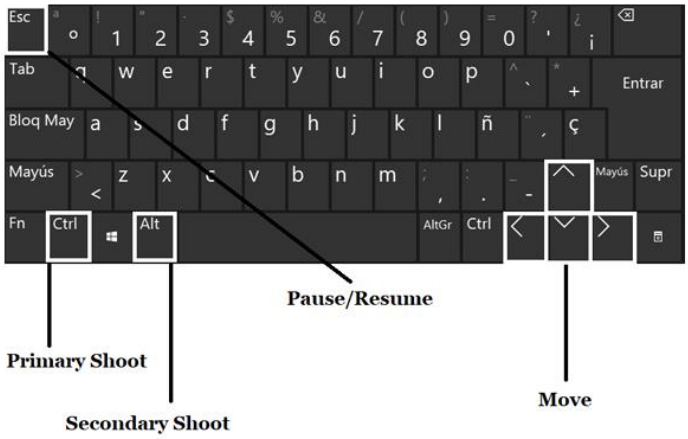


Рисунок 3.26 - Клавіші управління за замовчуванням

Головною метою цього проєкту було розробити двовимірну відеогру-стрілялку та вивчити якомога більше речей під час її виконання.

Для цієї мети були розроблені різні графічні елементи відеогри , що дозволило мені навчитися ефективно використовувати програмне забезпечення Inkscape для досягнення бажаних результатів. Таким чином, кожен дизайн був виконаний швидше та краще, ніж попередній. Так само я навчився розробляти прості, хоча й необхідні звукові ефекти для гри, поєднуючи аудіопрограми, зокрема Audacity та Bfxr.

Що стосується самої відеогри, то вся ігрова логіка та об'єкти були розроблені та згодом реалізовані за допомогою Unity та його 2D-інструментів. В результаті відеогра складається з одного ідеально ігрового рівня, який включає багато різних об'єктів та перешкод, які потрібно подолати. Це та частина, яка мені найбільше сподобалася, тому що мені дуже подобається програмувати та розробляти ігрову логіку. Крім того, придумування різних ворогів та логіки босів стало для мене цікавим викликом та дало мені можливість вивчити багато різних питань щодо дизайну та розробки ігор.

Крім того, я дізнався багато цікавого про розробку гри для людей з дальтонізмом та знайшов спосіб, щоб вони могли повною мірою насолоджуватися грою.

З іншого боку, отриману відеогру можна грати на різних операційних - системах, і вона має дуже хорошу продуктивність навіть на застарілих комп'ютерах. Більше того, її можна легко портувати на будь-яку ігрову консоль сучасного покоління.

Ігровий продукт було завершено за планом, і ця робота надає гарну основу та інструкції щодо створення 2D-платформерної гри-ранера з нескінченним циклом дій за допомогою ігрового движка Unity.

Хоча проєкт було завершено за планом, його ще не можна вважати повноцінно розробленою грою. Можна було б реалізувати більше ігрового балансу, щоб зробити відчуття руху гравця якомога плавнішим. Можна було б

додати більше перешкод, а майбутні знаряддя могли б містити пастки та навіть рухомих ворогів. Звукова система є обов'язковою функцією надалі, а спосіб зберігати та показувати таблиці лідерів гравців, які вижили найдовше, є приємною функцією.

3.4. Висновок по розділу

У третьому розділі виконано проєктування архітектури, реалізацію та оцінювання ефективності ігрового застосунку. Реалізовано основні функціональні модулі із застосуванням об'єктно-орієнтованих патернів, що дозволило підвищити структурованість коду та спростити його супровід. За результатами тестування встановлено, що розроблене програмне рішення забезпечує коректне функціонування ігрової логіки, достатню продуктивність і підтверджує ефективність використання патернів проєктування під час створення 2D-платформера.

ВИСНОВКИ

У ході виконання дипломної роботи було досліджено принципи проєктування та розробки ігрових застосунків у жанрі 2D-платформер із використанням об'єктно-орієнтованих патернів проєктування. Робота охопила аналіз сучасних підходів до створення комп'ютерних ігор, вивчення принципів об'єктно-орієнтованого програмування, дослідження шаблонів проєктування та їх практичного застосування під час побудови архітектури програмного забезпечення. У результаті було сформовано програмне рішення, яке характеризується модульністю, гнучкістю, простотою супроводу та можливістю подальшого розширення функціональних можливостей.

На початковому етапі було виконано аналіз теоретичних основ об'єктно-орієнтованого програмування, розглянуто класифікацію породжувальних, структурних і поведінкових патернів, а також досліджено особливості їх використання під час створення сучасного програмного забезпечення. Проведений аналіз показав, що використання шаблонів проєктування дозволяє значно покращити структуру програмного коду, забезпечити слабке зв'язування між компонентами системи та підвищити можливість повторного використання програмних модулів.

Під час дослідження методів та інструментів розробки було проаналізовано сучасні технології створення 2D-ігор, принципи побудови архітектури ігрових застосунків та особливості використання шаблонів проєктування для реалізації окремих компонентів системи. Було обґрунтовано вибір відповідних програмних засобів, визначено структуру програмного забезпечення та спроєктовано взаємодію між основними класами й модулями. Особливу увагу приділено використанню патернів Factory, Singleton, State, Observer та Strategy, які дозволили організувати ефективне керування ігровими об'єктами, обробкою подій, станами персонажів та взаємодією компонентів системи.

Практична частина роботи була присвячена реалізації ігрового застосунку в жанрі 2D-платформер. Було реалізовано основні ігрові механіки, систему керування персонажем, взаємодію з ігровим середовищем, обробку зіткнень, логіку проходження рівнів, систему керування ігровими ресурсами та механізми переходу між станами гри.

У процесі реалізації було проведено комплексне тестування програмного забезпечення, яке включало перевірку працездатності основних функціональних модулів, правильності взаємодії між компонентами системи, стабільності роботи ігрової логіки та оцінювання продуктивності застосунку.

Загалом поставлена мета дипломної роботи була досягнута. Розроблений ігровий застосунок демонструє ефективність використання об'єктно-орієнтованих патернів під час створення програмного забезпечення для комп'ютерних ігор. Використання шаблонів проєктування забезпечило високу модульність, зручність супроводу, можливість масштабування та повторного використання програмного коду, що значно спрощує подальший розвиток програмного продукту.

Практична цінність роботи полягає у можливості використання запропонованих архітектурних рішень під час створення інших ігрових застосунків різних жанрів, а також у навчальному процесі при вивченні дисциплін, пов'язаних із програмною інженерією, об'єктно-орієнтованим програмуванням та розробкою комп'ютерних ігор.

Перспективами подальшого розвитку роботи є реалізація багаторівневої системи гри, удосконалення поведінки неігрових персонажів із використанням алгоритмів штучного інтелекту, додавання мережевого режиму гри, інтеграція фізичних рушіїв, підтримка мобільних платформ, розширення системи анімації та звукового супроводу, а також оптимізація продуктивності для сучасних апаратних платформ.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Unity Technologies. (2025). *Unity Manual*. https://docs.unity3d.com/Manual/
2. Unity Technologies. (2025). *Unity Scripting API*. https://docs.unity3d.com/ScriptReference/
3. Microsoft. (2025). *.NET Documentation*. https://learn.microsoft.com/dotnet/
4. Microsoft. (2025). *C# Documentation*. https://learn.microsoft.com/dotnet/csharp/
5. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. https://www.pearson.com/store/p/design-patterns-elements-of-reusable-object-oriented/P100000252058
6. Freeman, E., & Robson, E. (2021). *Head First Design Patterns* (2nd ed.). O'Reilly Media. https://www.oreilly.com/library/view/head-first-design/9781492077992/
7. Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson. https://www.pearson.com/store/p/clean-architecture-a-craftmans-guide-to-software-structure-and-design/P100001465201
8. Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson. https://www.pearson.com/en-us/subject-catalog/p/clean-code/P200000009529
9. Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. https://martinfowler.com/books/eaa.html
10. Nystrom, R. (2014). *Game Programming Patterns*. https://gameprogrammingpatterns.com/
11. Gregory, J. (2018). *Game Engine Architecture* (3rd ed.). CRC Press.
12. Novak, J. (2019). *Game Development Essentials: An Introduction* (4th ed.). Cengage Learning.

13. Rabin, S. (Ed.). (2024). *Game AI Pro Online Edition*. CRC Press. https://www.gameaipro.com/

14. Millington, I. (2019). *Artificial Intelligence for Games* (3rd ed.). CRC Press.

15. Rogers, S. (2014). *Level Up! The Guide to Great Video Game Design* (2nd ed.). Wiley.

16. Schell, J. (2020). *The Art of Game Design: A Book of Lenses* (3rd ed.). CRC Press.

17. Brackeys. (2021). *Unity Game Development Tutorials*. https://brackeys.com/

18. Unity Learn. (2025). *Learning Pathways*. https://learn.unity.com/

19. Microsoft. (2025). *.NET Best Practices*. https://learn.microsoft.com/dotnet/standard/

20. Microsoft. (2025). *Object-Oriented Programming Guide (C#)*. https://learn.microsoft.com/dotnet/csharp/fundamentals/object-oriented/

21. Refactoring.Guru. (2025). *Design Patterns*. https://refactoring.guru/design-patterns

22. Refactoring.Guru. (2025). *SOLID Principles*. https://refactoring.guru/design-patterns/solid

23. OWASP Foundation. (2024). *OWASP Top Ten*. https://owasp.org/www-project-top-ten/

24. NUnit. (2025). *NUnit Documentation*. https://docs.nunit.org/

25. Microsoft. (2025). *MSTest Documentation*. https://learn.microsoft.com/dotnet/core/testing/unit-testing-with-mstest

26. JetBrains. (2025). *Rider Documentation*. https://www.jetbrains.com/rider/documentation/

27. Git. (2025). *Git Documentation*. https://git-scm.com/doc

28. GitHub Docs. (2025). *GitHub Actions Documentation*. https://docs.github.com/actions

29. Microsoft. (2025). *Visual Studio Documentation*. https://learn.microsoft.com/visualstudio/

30. Unity Technologies. (2025). *Unity Physics*. https://docs.unity3d.com/Manual/PhysicsSection.html

31. Unity Technologies. (2025). *2D Game Development*. https://docs.unity3d.com/Manual/2DFeature.html

32. Unity Technologies. (2025). *Tilemap System*. https://docs.unity3d.com/Manual/class-Tilemap.html

33. Unity Technologies. (2025). *Animator Controller*. https://docs.unity3d.com/Manual/class-AnimatorController.html

34. Unity Technologies. (2025). *ScriptableObject Manual*. https://docs.unity3d.com/Manual/class-ScriptableObject.html

35. Unity Technologies. (2025). *Prefab Workflow*. https://docs.unity3d.com/Manual/Prefabs.html

36. Unity Technologies. (2025). *Scene Management*. https://docs.unity3d.com/Manual/MultiSceneEditing.html

37. Unity Technologies. (2025). *Input System Documentation*. https://docs.unity3d.com/Packages/com.unity.inputsystem/

38. Unity Technologies. (2025). *UI Toolkit Documentation*. https://docs.unity3d.com/Manual/UIElements.html

39. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley. https://algs4.cs.princeton.edu/home/

40. Sommerville, I. (2020). *Software Engineering* (10th ed.). Pearson.

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: " Проектування та розробка ігрового застосунку в жанрі 2D-платформер з використанням об'єктно-орієнтованих патернів "

Обсяг пояснювальної записки: 70 аркушів

Дата закінчення дипломної роботи 10 червня 2026р.

Підпис студента _____