БАКАЛАВРСЬКА РОБОТА

БР. ІП - 06.00.00.000 ПЗ

Група ІП-21-4

Гірняк Андрій

2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Гірняк Андрій Віталійович

(прізвище, ім'я, по батькові)

УДК 004

(індекс)

БАКАЛАВРСЬКА РОБОТА

Розробка мобільного варіанту крос-платформенної гри

(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 рівня Гірняк А.В.

(підпис, ініціали та прізвище здобувача)

Науковий керівник Корнута Володимир Андрійович, к.т.н., доцент

(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 – 2025

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**
доц. В.В. Бандура
“ ____ ” _____2025 р.

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Гірняку Андрію Віталійовичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) **“ Розробка мобільного варіанту крос-платформенної гри”**

керівник проекту (роботи) Корнута В.А., к.т.н., доцент

затверджені наказом закладу вищої освіти від **“** 28 ” квітня 2025 р. № 264/7

2. Строк подання студентом проекту (роботи) 10 червня 2025 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 предметної області розробки крос-платформенних ігор
2. Розробка архітектури та дизайну мобільного варіанту крос-платформенної гри
3. Проектування UML-діаграм класів програмного продукту
4. Програмна реалізація інтерфейсу мобільного варіанту крос-платформенної гри
5. Реалізація ігрової механіки

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

1. Порівняння сучасної версії Unity (ліворуч) та оригінальної Unity 1 (праворуч) (рис. 1.1)
2. Solar2D (рис. 1.2)
3. Платформа Cocos2D JS (рис. 1.3)
4. Платформа Appcelerator Titanium (рис. 1.4)
5. Adobe Animate (рис. 1.5)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання **22 травня 2025 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Аналіз предметної області розробки крос-платформенних ігор	04.05.2025	виконано
2	Розробка архітектури та дизайну мобільного варіанту крос-платформенної гри	14.05.2025	виконано
3	Проектування UML-діаграм класів програмного продукту	25.05.2025	виконано
4	Програмна реалізація інтерфейсу мобільного варіанту крос-платформенної гри	01.06.2025	виконано
5	Реалізація ігрової механіки	03.06.2025	виконано
6	Оформлення пояснювальної записки дипломної роботи завідувачем кафедри	10.06.2025	виконано

Студент – дипломник _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 77 сторінок, 44 рисунки, список використаних джерел із 35 найменуваннями.

Мета роботи - розробити мобільний варіант крос-платформенної гри з використанням сучасних технологій та інструментів, забезпечивши її функціональність, візуальну привабливість і продуктивність на різних платформах.

Об'єкт дослідження - процес розробки крос-платформених ігор.

Предмет дослідження - методи, інструменти та технології розробки крос-платформенної мобільної гри.

В першому розділі у результаті аналізу предметної області визначено ключові особливості крос-платформенного підходу та обґрунтовано вибір ігрового рушія для реалізації проєкту.

В другому розділі було спроєктовано архітектуру гри, визначено структуру класів і реалізовано основні ігрові компоненти, що забезпечують взаємодію елементів гри.

В третьому розділі реалізовано повноцінний мобільний інтерфейс гри, інтегровано ігрову логіку та додаткові об'єкти, що забезпечують гнучкість і розширюваність застосунку.

Висновок: розроблено архітектуру гри, що включає проектування та реалізацію основних класів, UML-діаграм і компонентів ігрової механіки. Особливу увагу приділено реалізації інтерфейсу користувача, ігрових елементів та інтерактивних об'єктів.

КЛЮЧОВІ СЛОВА: КРОС-ПЛАТФОРМЕННА РОЗРОБКА, МОБІЛЬНА ГРА, ІГРОВИЙ РУШІЙ, ІНТЕРФЕЙС КОРИСТУВАЧА, ІГРОВА МЕХАНІКА, UML-ДІАГРАМА, МУЛЬТИПЛАТФОРМЕННЕ СЕРЕДОВИЩЕ.

ANNOTATION

The bachelor's thesis contains 77 pages, 44 figures, a list of used sources with 35 names.

The purpose of the work is to develop a mobile version of a cross-platform game using modern technologies and tools, ensuring its functionality, visual appeal and performance on different platforms.

The object of the study is the process of developing cross-platform games.

The subject of the study is methods, tools and technologies for developing a cross-platform mobile game.

In the first section, as a result of the analysis of the subject area, the key features of the cross-platform approach were identified and the choice of a game engine for the implementation of the project was justified.

In the second section, the game architecture was designed, the class structure was determined and the main game components were implemented that ensure the interaction of game elements.

In the third section, a full-fledged mobile game interface was implemented, game logic and additional objects were integrated that provide flexibility and extensibility of the application.

Conclusion: The game architecture has been developed, including the design and implementation of core classes, UML diagrams, and game mechanics components. Particular attention has been paid to the implementation of the user interface, game elements, and interactive objects.

KEYWORDS: CROSS-PLATFORM DEVELOPMENT, MOBILE GAME, GAME ENGINE, USER INTERFACE, GAME MECHANICS, UML DIAGRAM, MULTI-PLATFORM ENVIRONMENT.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

AI - Articial Intelligence

Apk - Android file format

GPU - Graphics Processing Unit

GUI - Graphical User Interface

HP - Health Points

HUD - Heads Up Display

ipa - iOS file format

SDK - Software Development Kit

XML - Extensible Markup Language

MIT - Massachusetts Institute of Technology

NDA - Non-disclosure agreement

RAM - Random-access memory

ВСТУП

Упродовж останніх десятиліть ігрова індустрія перетворилася на одну з найдинамічніших галузей сучасного цифрового ринку. Розвиток мобільних технологій, зростання популярності портативних пристроїв, а також зростаюча вимога користувачів до зручного доступу до ігрового контенту з різних платформ призвели до широкого впровадження крос-платформенних підходів у розробці ігор.

Крос-платформенна розробка передбачає створення програмного продукту, який може функціонувати на декількох операційних системах (Android, iOS, Windows, macOS тощо) без суттєвих змін у кодовій базі. Такий підхід дозволяє не лише зменшити витрати на розробку та підтримку, але й значно пришвидшити виведення продукту на ринок, розширити цільову аудиторію та підвищити конкурентоспроможність проєкту.

У межах цієї роботи розглядається повний цикл розробки мобільного варіанту крос-платформенної гри, який включає аналіз предметної області, вибір технологій та інструментів, проектування архітектури та інтерфейсу, реалізацію функціональних компонентів гри, а також інтеграцію ігрової логіки.

Особлива увага приділяється порівняльному аналізу популярних ігрових рушіїв, таких як Unity, Solar2D (Corona), Cocos2D JS та Appcelerator Titanium. Обґрунтовується вибір рушія для реалізації поставлених завдань, досліджуються особливості їх застосування у контексті мобільної крос-платформенної розробки.

Результатом дослідження є створення функціонального мобільного ігрового застосунку з підтримкою основних ігрових механік, графічного інтерфейсу та інтерактивної взаємодії з користувачем. Представлена робота демонструє практичне застосування теоретичних знань з програмної

інженерії, об'єктно-орієнтованого програмування, проектування архітектури ПЗ, що є важливим етапом підготовки фахівця в галузі ІТ.

Актуальність роботи

Актуальність даної роботи зумовлена потребою у розробці ігор, які можуть функціонувати на різних платформах із мінімальними витратами на адаптацію. Завдяки крос-платформенним технологіям розробники отримують змогу швидко запускати продукт на мобільних, настільних та веб-платформах, що значно скорочує терміни розробки та підвищує рентабельність проєкту. Розробка мобільної гри в такому підході вимагає детального аналізу інструментів, проектування архітектури, розробки взаємодії користувача з інтерфейсом та ефективної реалізації ігрової логіки. Робота поєднує теоретичні основи, практичні реалізації та рекомендації щодо подальшого розвитку ігор у мультиплатформенному середовищі.

Мета роботи - розробити мобільний варіант крос-платформенної гри з використанням сучасних технологій та інструментів, забезпечивши її функціональність, візуальну привабливість і продуктивність на різних платформах.

Завдання дослідження:

- Провести аналіз особливостей крос-платформенної розробки ігор та порівняння ігрових рушіїв.
- Обґрунтувати вибір інструментів для реалізації гри.
- Спроєктувати архітектуру та дизайн мобільної гри.
- Реалізувати класи, що забезпечують ігрову логіку та інтерфейс.
- Створити повноцінну програмну реалізацію інтерфейсу та механіки гри.

Об’єкт дослідження - процес розробки крос-платформених ігор.

Предмет дослідження - методи, інструменти та технології розробки крос-платформенної мобільної гри.

Методи дослідження:

- Порівняльний аналіз (ігрових рушіїв та інструментів).
- Моделювання (архітектури та класів гри).
- Проєктування UML-діаграм.
- Експериментальна розробка програмного забезпечення.
- Тестування реалізованого ігрового функціоналу.

Наукова новизна

Запропоновано структурований підхід до створення крос-платформенної гри з урахуванням специфіки мобільного застосування, а також створено узагальнений механізм реалізації ігрової логіки з використанням об’єктно-орієнтованого підходу.

Практичне застосування

Результати роботи можуть бути використані для створення ігор, що підтримують декілька платформ, а також як навчальний приклад для студентів спеціальностей, пов’язаних із розробкою програмного забезпечення.

Бакалаврська робота містить 77 сторінок, 44 рисунки, 3 розділи список використаних джерел із 35 найменуваннями.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ РОЗРОБКИ КРОС-ПЛАТФОРМЕННИХ ІГОР

1.1. Особливості крос-платформенної розробки

Крос-платформенна розробка визначається як процес створення програмного забезпечення, призначеного для функціонування на множинних обчислювальних платформах з єдиної кодової бази. Цей підхід спрямований на мінімізацію дублювання зусиль при адаптації програмного продукту до специфічних вимог кожної цільової платформи.

Існують дві основні парадигми крос-платформенної розробки:

- Компільована крос-платформенна розробка (Compiled Cross-Platform Development) - це підхід, що передбачає створення програмного коду з використанням уніфікованого набору мов програмування та фреймворків. Для розгортання на кожній цільовій платформі вимагається окремий етап компіляції та пакування, що генерує нативний або майже нативний виконуваний файл для конкретної операційної системи (наприклад, Windows, macOS, Android, iOS). Прикладом такого підходу є розробка ігрових додатків за допомогою ігрових рушіїв, таких як Unity або Unreal Engine. У цьому випадку розробники працюють над єдиним проєктом, а рушій забезпечує функціональність для створення спеціалізованих білдів для різних платформ, як правило, без необхідності суттєвого переписування основного коду.

- Інтерпретована / Веб-орієнтована крос-платформенна розробка (Interpreted/Web-Based Cross-Platform Development) - цей підхід використовує універсальні технології та середовища виконання, які доступні на багатьох платформах. Програмний код виконується в межах цього середовища, яке вже адаптоване до специфіки конкретної ОС. Класичним прикладом є веб-додатки, що функціонують у веб-браузерах. Веб-браузери розроблені для роботи на широкому спектрі пристроїв (персональні комп'ютери, мобільні

пристрої, ігрові консолі, телевізори). Вони інтерпретують код, написаний з використанням веб-стандартів (HTML, CSS, JavaScript), забезпечуючи, як правило, візуально та функціонально послідовне відображення контенту на різних платформах, хоча адаптивні веб-дизайни можуть динамічно змінювати макет залежно від розміру екрана.

1.1.1. Крос-платформенна гра як застосування крос-платформенної розробки

Однією з ключових функціональних можливостей, що стала доступною завдяки розвитку крос-платформенних технологій, є крос-платформенна гра (cross-platform play або cross-play). Це означає, що гравці, які використовують різні апаратні платформи (наприклад, ПК та ігрову консоль, або різні моделі консолей, або консоль та мобільний пристрій), можуть одночасно взаємодіяти в межах спільної ігрової сесії в одному й тому ж програмному продукті.

Технічна можливість реалізації крос-платформенних ігрових сесій значною мірою забезпечується прогресом у сфері комп'ютерних апаратних засобів та розвитку мережевої інфраструктури, що дозволяє підтримувати синхронізований ігровий стан для великої кількості користувачів на різних пристроях. Однак, широке впровадження крос-платформенної гри історично та на поточному етапі обмежується двома основними факторами:

- Диференціація схем управління, оскільк існують значні відмінності у методах введення даних між різними ігровими платформами. Зокрема, комбінація клавіатури та миші на персональних комп'ютерах часто надає гравцям потенційну перевагу у точності та швидкості реакції порівняно з використанням контролерів на ігрових консолях. Ця асиметрія може порушувати змагальний баланс, особливо в певних жанрах ігор (наприклад, шутери від першої особи), що вимагає від розробників спеціальних заходів

(таких як розділення пулів гравців за типом пристрою введення або адаптація ігрової механіки) для забезпечення справедливих умов змагання.

- Політика та екосистеми виробників консолей, оскільки функціонування онлайн-сервісів на ігрових консолях контролюється відповідними виробниками (Sony PlayStation, Microsoft Xbox, Nintendo Switch). Ці платформи створюють власні, часто закриті екосистеми для забезпечення безпечного та послідовного онлайн-досвіду. Дозвіл на крос-платформенну взаємодію залежить від політики та технічних можливостей, що надаються власником платформи. Історично, міжплатформна гра була обмежена, часто дозволяючи взаємодію лише між консоллю та ПК одного виробника (наприклад, Microsoft Xbox One та Windows/ПК). Проте, останнім часом спостерігається тенденція до більшої відкритості, коли деякі видавці та виробники платформ (зокрема, Microsoft) активно підтримують крос-платформенну гру між конкуруючими консолями (PlayStation, Xbox, Switch) та мобільними пристроями. Успішна реалізація вимагає узгодження технічних стандартів, процесів оновлення та управління користувацькими обліковими записами між різними платформами.

1.2. Розробка для кількох платформ

При плануванні початку розробки для кількох платформ рекомендується вибрати ігровий рушій, який сумісний з обраними платформами та може компілюватися для них. Це слід вирішити на ранніх етапах розробки, оскільки портування ігор може виявитися складним і викликати несподівані вимоги, що можуть повністю зупинити розробку або значно сповільнити її. Деякі недавні приклади портування гри, у цьому випадку для операційної системи Linux, — це Gauntlet і Divinity: Original Sin. Вони зіткнулися з численними проблемами через різні причини, навіть якщо обидві студії, що стоять за цими іграми, дуже досвідчені у роботі з

операційними системами Windows, але крос-платформенна розробка була для них новою.

Крім цього, є кілька ключових моментів, які слід враховувати при розробці для кількох платформ одночасно:

- Співвідношення сторін — це рідко стандартне і може варіюватися від 4:3 до 21:9

- Розмір екрана — смартфони, планшети, ПК та телевізори, розмір може варіюватися від 4 дюймів до 50+ дюймів.

- Зазвичай ігрові рушії пропонують спосіб емулювати різні розміри екранів, щоб ці проблеми можна було вирішити досить легко.

- Аудіо- та відеокодеки — підтримувані кодеки різняться залежно від платформи

- Специфічні для платформи соціальні функції — ці інтерфейси та функціональність зазвичай дуже різні залежно від платформи.

Ці проблеми впливають на всі платформи в цілому, але є також специфічні для платформи проблеми, які вимагають особливої уваги при розробці для них.

1.2.1. Настільні системи

Завдяки Valve's SteamPlay стало звичайним випускати ігри для трьох найбільш використовуваних настільних платформ, а не лише для Windows. Схеми введення практично ідентичні на всіх платформах, але розробникам слід звернути увагу на сумісність сторонніх плагінів та фреймворків. Крім цього, ключові відмінності включають:

- Файлові системи різняться за платформою, також синтаксис шляху унікальний для Windows.

- Windows — нечутлива до регістру, домашня директорія: C:\Users\example}

- OS X — нечутлива до регістру, домашня директорія: /Users/example/

- Ubuntu/SteamOS — чутлива до регістру, домашня директорія: /home/example/

- GPU драйвери — драйвери Intel, AMD та nVidia різняться за платформою. Сумісність та функціональність цих драйверів слід перевіряти на ранніх етапах розробки, оскільки якщо їх перевіряти занадто пізно, це може призвести до проблем, таких як гра може працювати лише з відеокартами nVidia на Linux.

- Правильна інтеграція гри в менеджери вікон платформи (мінімізація, повноекранний режим).

1.2.2. Веб-ігри

На сьогоднішній день веб-ігри стають все популярнішими та складнішими. Вони високоякісні та візуально вражаючі, як ігри на інших платформах. Веб також є чудовою платформою для розповсюдження ігор. У минулому більшість веб-ігор створювалися за допомогою Flash, але завдяки зростанню продуктивності JavaScript та новим API ігри можна робити більш вимогливими та запускати з HTML5-браузерів. Однак у веб є обмеження, наприклад, єдина мова програмування — це JavaScript, тому код гри зазвичай потрібно писати на JS або конвертувати за допомогою компілятора, такого як Emscripten, який використовується принаймні в Unity при збірці для WebGL. Emscripten перетворює рідну мову C# Unity на C++, а потім на asm.js, який є JavaScript. Крім того, потокова обробка не працює при збірці для Web, оскільки JavaScript не підтримує потоки.

1.2.3. Мобільні ігри

Для розробки мобільних ігор існують різні методи введення, які потрібно узгодити з тими, що на настільних пристроях. Також існує велика

кількість різних типів екранів, які потрібно враховувати. Деякі ключові моменти, які слід враховувати:

- Файлові системи Android та iOS чутливі до регістру.

- Права доступу до файлових систем дуже різні залежно від платформи.

Більшість інших основних проблем у розробці пов'язані з Windows Phone та її системою, але Microsoft заявила, що вони припиняють підтримку цієї платформи.

1.2.4. Консолі

Більшість, якщо не все, з консольних наборів для розробки (SDK) є конфіденційними та захищеними угодою про нерозголошення (NDA), тому не так багато інформації відомо про конкретні консолі. Консолі нового покоління мають дуже схоже апаратне забезпечення порівняно з настільними системами, а екосистема додатків схожа на мобільні пристрої. Консолі як платформа мають такі відмінності:

- Вони підтримують різні шейдерні мови:
  - Xbox One: HLSL (Direct X 11)
  - PlayStation 4: Playstation Shader Language (GNM, GNMX)

Якщо використовується рушій Unity, шейдери GLSL не можуть бути крос-компіляцією. Ігри зазвичай працюють у пісочниці, що обмежує права доступу до файлової системи. Деякі платформи мають дуже унікальні периферійні пристрої, наприклад, пульт дистанційного керування Wii.

1.3 Критерії вибору та порівняльний аналіз крос-платформенних ігрових рушіїв

Вибір відповідного інструментального засобу розробки є критично важливим етапом у реалізації крос-платформенних ігрових проєктів, визначаючи ефективність процесу створення та подальшого розповсюдження

програмного продукту. В умовах сучасного ринку цифрових розваг, випуск ігор одночасно на множинних платформах або з мінімальним часовим лагом між релізами став стандартною практикою. У цьому контексті використання крос-платформенного ігрового рушія надає значні переваги, включаючи скорочення термінів розробки та уніфікацію кодової бази. У даному розділі представлено порівняльний аналіз декількох популярних крос-платформенних ігрових рушіїв, ґрунтуючись на огляді літературних джерел. Розглядаються такі рушії, як Unity, Corona, Cocos2D JS та Appcelerator Titanium. Рушій Unreal Engine, незважаючи на його широкі можливості та статус одного з лідерів галузі, має значні архітектурні та функціональні спільні риси з Unity, що дозволяє виключити його з детального порівняння в рамках цієї роботи, хоча він залишається рекомендованим для розробників, які мають досвід роботи з мовою C++.

1.3.1. Ігровий рушій Unity

Unity є пропрієтарним крос-платформенним ігровим рушієм, що підтримує розробку як 2D, так і 3D ігор. Робочий процес в рушії відбувається у 3D-середовищі, де ігрові об'єкти конфігуруються шляхом приєднання до них різних компонентів та сценаріїв. Основною мовою для написання сценаріїв у Unity є C#, хоча також підтримуються Boo та UnityScript (останній менш поширений у сучасній розробці).

Входження у робоче середовище Unity може становити певний виклик для нових користувачів через багатофункціональність інтерфейсу та широкий спектр концепцій, які необхідно опанувати. За оцінками, час, необхідний для базового ознайомлення з інтерфейсом користувача та основними принципами роботи, становить приблизно 8-12 годин, що вказує на помірну криву навчання.

Інтерфейс Unity, який був випущений у 2005 році, зберігає візуальну та структурну схожість з ранніми версіями (рис. 1.1). Деякі рутинні операції,

пов'язані з розробкою, такі як додавання аудіоджерел, оновлення префабів та імпорт ресурсів, можуть здаватися застарілими та менш ефективними (Weiss, 2014).

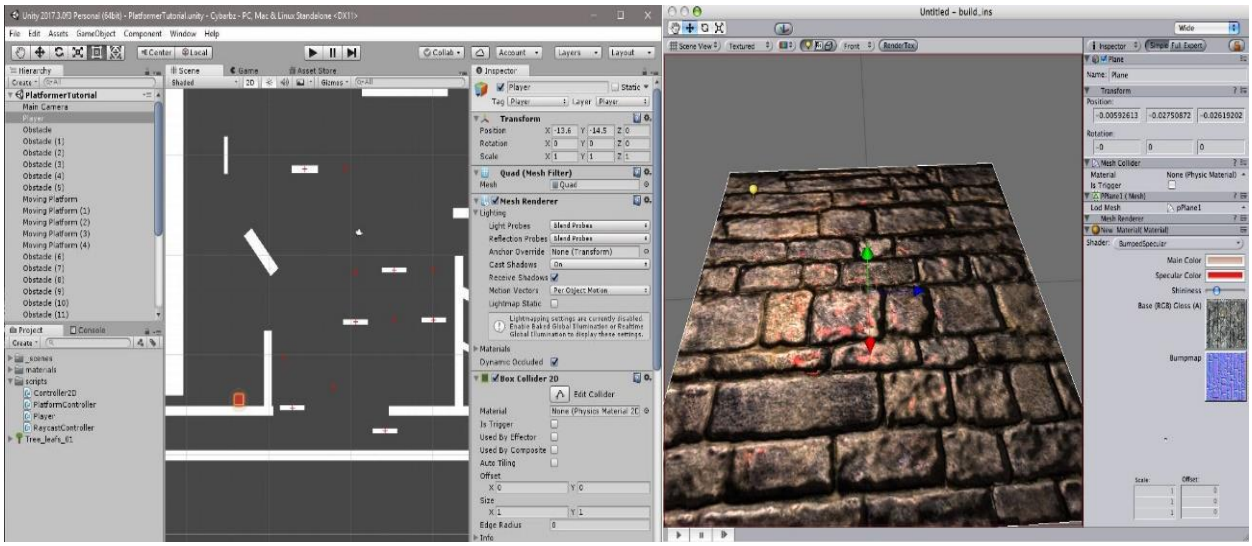

Рисунок 1.1 - Порівняння сучасної версії Unity (ліворуч) та оригінальної Unity 1 (праворуч)

Однією з ключових переваг Unity є простота процесу розповсюдження. Створену гру можна експортувати на численні платформи (мобільні, настільні, веб) за допомогою мінімальної кількості дій. Веб-розповсюдження, яке раніше вимагало використання окремого плагіна Unity Player, тепер реалізовано через технологію WebGL, що усуває необхідність встановлення додаткового програмного забезпечення користувачем. За наявності відповідних ліцензій можливе також розповсюдження ігор на ігрові консолі. Unity пропонує багаторівневу модель ліцензування, включаючи безкоштовну версію, яка, хоч і має певні функціональні обмеження (наприклад, відсутність підтримки відтворення відео високої якості), дозволяє розповсюджувати ігри на більшості підтримуваних платформ з відображенням логотипу Unity під час запуску. Сучасні ціни та умови ліцензування можуть відрізнятися.

1.3.2. Corona (Solar2D)

Corona (згодом перейменована на Solar2D після зміни розробника) є пропрієтарним (на момент огляду Weiss, 2014) двовимірним ігровим рушієм, що включає симулятор та хмарний сервіс збірки. Розробка здійснюється шляхом написання сценаріїв мовою Lua, виконання яких моделюється в інтегрованому симуляторі Corona. Симулятор надає широкий спектр опцій для тестування на різних віртуальних пристроях з різними роздільними здатностями та співвідношеннями сторін екрана. Процес розповсюдження вимагає використання хмарного сервісу для компіляції та пакування гри у формат, сумісний з iOS або Android, після чого збірка доставляється розробнику.

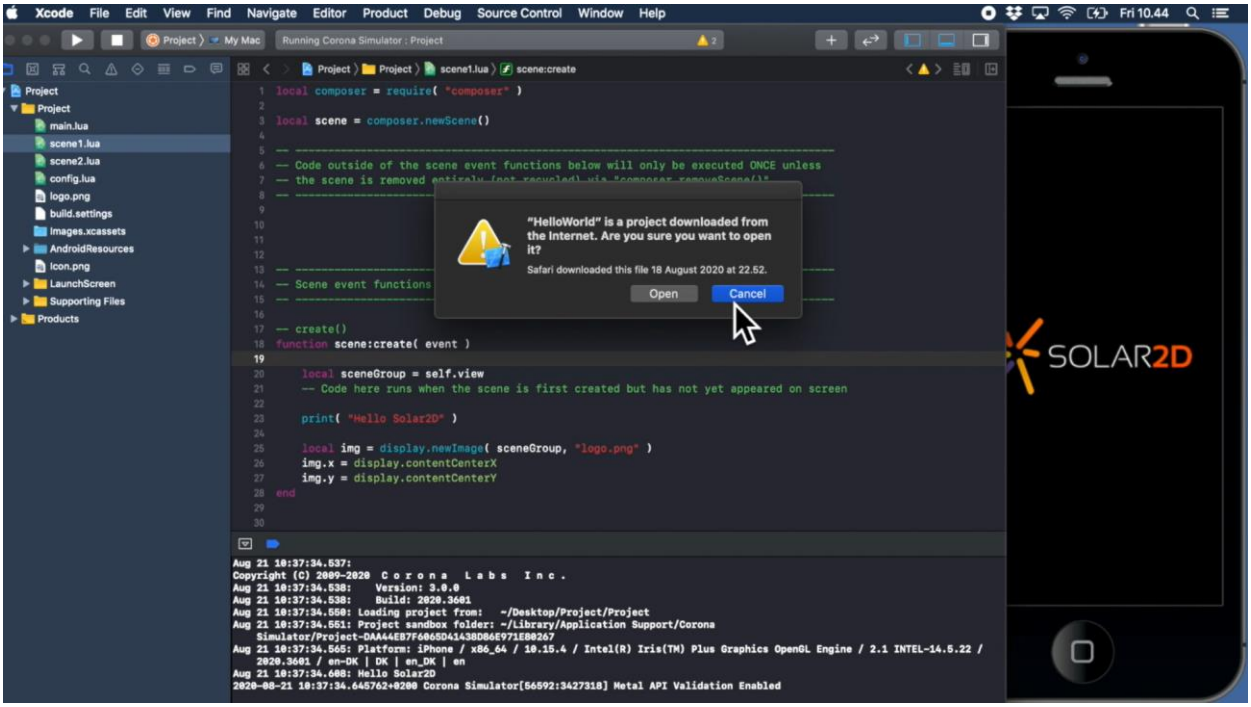

Рисунок 1.2 - Solar2D

Мова програмування Lua відрізняється простотою синтаксису, що сприяє швидкому написанню коду. Наприклад, ініціалізація фізичного тіла може бути виконана одним рядком коду. Крива навчання для освоєння базового функціоналу платформи оцінювалася у 2-4 години. Симулятор

відрізняється високою швидкістю реагування на внесені зміни в коді та економним споживанням ресурсів розробницької машини. Можливість миттєвого оновлення симулятора після збереження змін у редакторі коду значно прискорює цикл ітеративної розробки.

Одним з ключових обмежень Corona була обмежена кількість цільових платформ для розповсюдження, що включала лише мобільні пристрої (iOS, Android, Kindle, Nook). Процес хмарної збірки, хоч і зручний у деяких аспектах, міг становити виклик, вимагаючи кілька хвилин на отримання нової тестової збірки на пристрої, що могло бути незручним, особливо на пізніх стадіях розробки, коли вимагається часте тестування.

Історично Corona мала модель ліцензування з обмеженням доходу для безкоштовного використання, проте згодом ядро рушія стало повністю безкоштовним для всіх користувачів, з моделлю монетизації через ринок ресурсів.

1.3.3. Платформа розробки ігор Cocos2D JS

Cocos2D JS є відкритим, безкоштовним комплектом розробки ігор (SDK), що підтримує крос-платформенність. Цей проєкт є поєднанням функціональності двох популярних відкритих проєктів: Cocos2D X (для настільних та мобільних систем) та Cocos2D HTML5 (для веб-платформ). Рушій призначений насамперед для розробки 2D та 2.5D ігор, хоча існують плани щодо додавання підтримки 3D.

Розробка для нативних платформ (мобільних та настільних) здійснюється повністю на мові JavaScript. JavaScript-код взаємодіє з нативними об'єктами, написаними на C++, що забезпечує високу швидкість виконання без необхідності написання нативного коду безпосередньо розробником. Веб-версії ігор функціонують на чистому JavaScript і візуалізуються за допомогою HTML5 Canvas або WebGL, не вимагаючи встановлення сторонніх плагінів або гравців.

Найпростіший спосіб розпочати розробку з Cocos2D JS — це використання веб-орієнтованого підходу (HTML5). Це дозволяє швидко ітерувати, оновлюючи гру в браузері шляхом простого збереження змін у JavaScript-файлах. Для розповсюдження на нативних платформах необхідне використання відповідних зовнішніх середовищ розробки, таких як Xcode, Visual Studio або Eclipse, для процесу збірки.

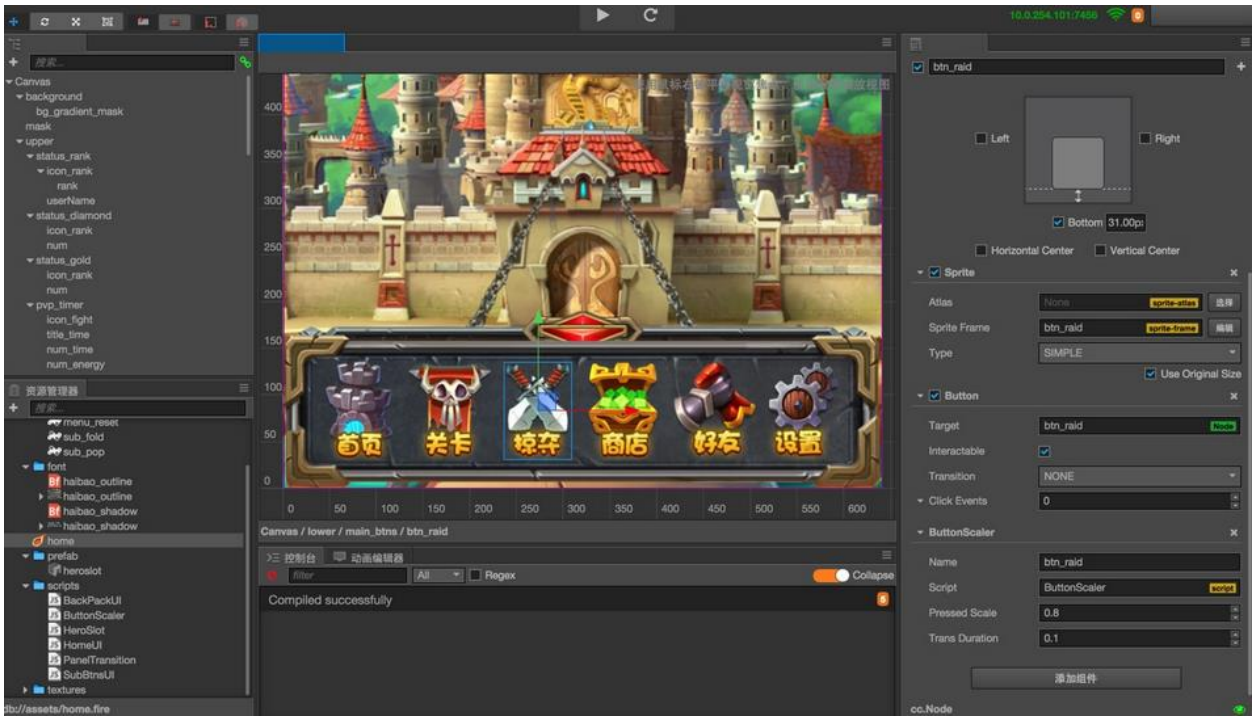

Рисунок 1.3 – Платформа Cocos2D JS

Cocos2D JS поширюється під ліберальною ліцензією MIT, що дозволяє вільне використання та модифікацію вихідного коду, навіть у комерційних пропрієтарних проєктах, за умови збереження тексту ліцензії. Відкритий вихідний код надає розробникам можливість детально вивчати внутрішню роботу рушія та адаптувати його до специфічних потреб.

1.3.4. Платформа Appcelerator Titanium

Appcelerator Titanium є крос-платформенним комплектом розробки додатків (SDK) та інтегрованим середовищем розробки (IDE), заснованим на

платформі Eclipse. Код додатків пишеться переважно на мові JavaScript, який потім виконується нативно через спеціальний міст (bridge), на відміну від виконання виключно у веб-перегляді (WebView). IDE Titanium Studio надає інструменти для розробки, тестування та розповсюдження додатків на мобільні та веб-платформи. Важливо зазначити, що Titanium насамперед орієнтований на розробку бізнес-додатків, а не складних ігор.

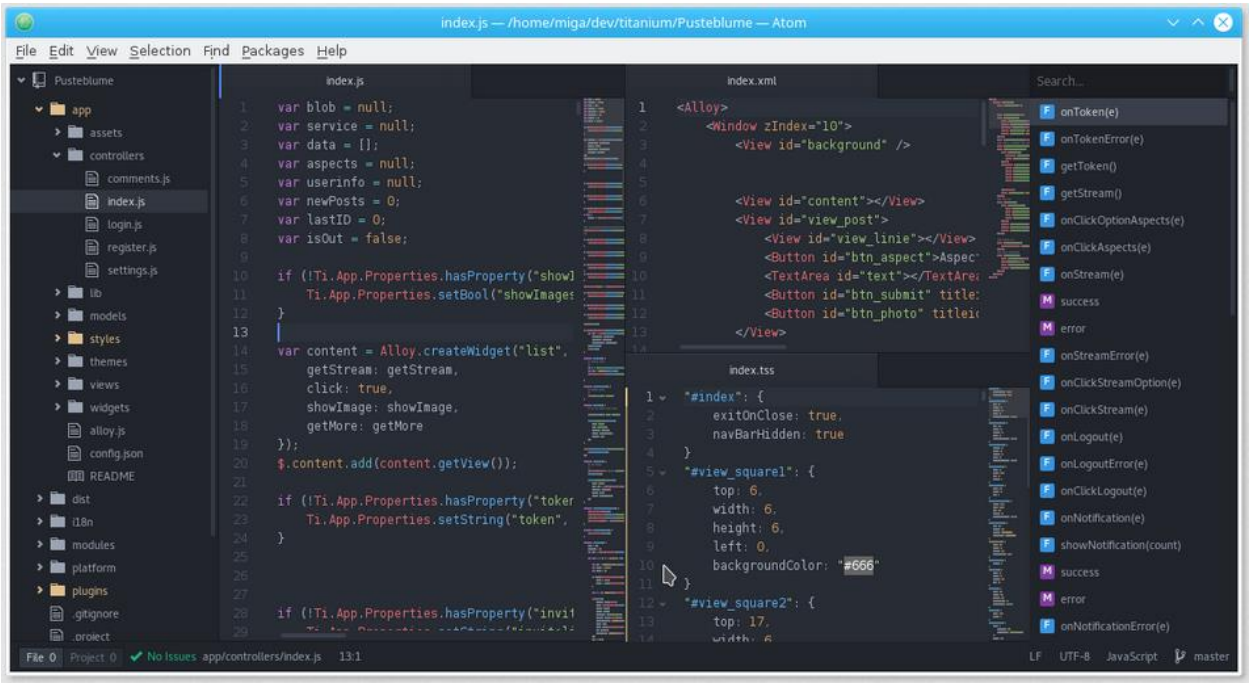

Рисунок 1.4 - Платформа Appcelerator Titanium

Для розробки 2D ігор у стеку Titanium існує спеціалізований рушій Platino Game Engine. Це рішення з відкритим вихідним кодом, але його використання вимагає придбання ліцензії (не безкоштовне SDK). Документація для Platino оцінювалася як обмежена та неповною, що може ускладнювати його вивчення та ефективне застосування. Фізичний рушій, інтегрований у Platino, на момент огляду вважався застарілим, вимагаючи ручної синхронізації фізичних тіл та візуальних спрайтів через C-подібний API, що є незручним підходом. Незважаючи на ці недоліки, Titanium відомий високою швидкістю збірки, оскільки використовує попередньо компільований SDK, що дозволяє швидко запускати розроблені додатки у

симуляторі або на реальному пристрої. Розповсюдження додатків, створених за допомогою базового SDK Titanium, можливе на iOS, Android, Blackberry та HTML5 без додаткової плати. Проте, використання рушія Platino та інших допоміжних інструментів може вимагати щорічної передплати (Weiss, 2014).

1.3.5. Порівняльний аналіз та рекомендації щодо вибору ігрового рушія

На основі проведеного порівняльного аналізу розглянутих крос-платформенних ігрових рушіїв, можна сформулювати наступні рекомендації щодо вибору, виходячи з характеристик проєкту:

- для 3D-проєктів або складних 2D-ігор з високими вимогами до продуктивності та візуальної якості рекомендується обирати Unity. Він надає повноцінне 3D-середовище, потужний інструментарій та широкі можливості оптимізації для різноманітних платформ.

- для простих 2D-ігор з основним фокусом на мобільні платформи та швидке прототипування - Corona (Solar2D) може бути ефективним вибором завдяки простоті мови Lua, швидкому симулятору та легкості початку розробки. Слід враховувати обмеження щодо цільових платформ та особливості процесу збірки.

- для 2D-ігор, призначених для широкого спектра платформ (мобільні, настільні, веб) або для розробників, які віддають перевагу відкритому вихідному коду та мові JavaScript - Cocos2D JS (включаючи сімейство Cocos2D-X) є сильним кандидатом. Його відкрита ліцензія та архітектура, що поєднує продуктивність C++ та гнучкість JavaScript, надають значну свободу та можливості для адаптації.

- для проєктів, що мають більше характеристик бізнес-додатків з елементами 2D-ігрової механіки, де пріоритетом є швидка збірка та використання JavaScript - Titanium може бути розглянутий. Проте, для складних ігрових сценаріїв з інтенсивною фізикою або графікою, його

обмеження та особливості (зокрема, застарілий фізичний рушій та платний ігровий додаток) можуть становити суттєві перешкоди.

Важливо підкреслити, що вибір рушія також має ґрунтуватися на досвіді та вподобаннях команди розробників, бюджеті проєкту, специфічних вимогах до функціоналу та довгострокових планах підтримки та масштабування. Наведене порівняння базується на інформації, яка може частково відображати стан рушіїв на момент публікації відповідних джерел і поточний стан, функціонал та ліцензійні моделі можуть відрізнятися.

1.4. Мета та об’єм програмної розробки гри

Недавні технології зробили створення та розгортання ігрових додатків для мобільних пристроїв все простіше для незалежних розробників. Ця робота присвячена дослідженню одного з таких наборів технологій, включаючи Adobe AIR, Starling і Feathers, які дозволяють створювати ігри, які працюють на пристроях Android і iOS. Для цієї мети було вирішено використати ці технології для проектування та реалізації гри. Її можна розгортати на різних операційних системах, включаючи Android, iOS, OSX і Windows. У цій роботі представляється дизайн гри та те, як було вирішено реалізувати ці вимоги. Це включає архітектуру системи та основні функції, такі як інтерфейс користувача, ігрові екрани, управління ресурсами, ШІ, системи частинок та інші ігрові механіки. Також обговорюються інструменти розробки, які використовувалися для створення програмних компонентів гри.

Основна мета — зрозуміти, як розробляти ігри для мобільних пристроїв за допомогою останніх інструментів і технологій, а також покращити мої навички як програміста ігрового процесу. Гра також розроблена таким чином, щоб її можна було розширювати, щоб інші подібні ігри можна було створювати за допомогою тієї ж архітектури.

Розробка гри, навіть простої, — це тривалий і всеохоплюючий процес, який включає навички з різних галузей. Ця робота обговорює лише розробку програмного забезпечення для гри. Також буде обговорено дизайн гри, але лише для того, щоб допомогти читачеві зрозуміти, які вимоги до програмного забезпечення потрібно було реалізувати. Інші аспекти розробки ігор, такі як мистецтво, музика, управління та маркетинг, не будуть розглянуті. Крім того, протягом розробки пройшла кілька рефакторингів і змін дизайну гри. Ця робота охоплює лише концепції дизайну гри та архітектуру програмного забезпечення поточної версії ANTics.

1.5. Інструменти розробки кросс-платформенної гри

Цей проєкт був створений за допомогою різних інструментів розробки ігор. Інструменти програмування, які використовувалися, включають: крос-платформенну систему Adobe AIR, середовище розробки FlashDevelop та бібліотеки ActionScript Starling і Feathers. Вихідний код був повністю написаний на ActionScript 3.0. Також використовувалися інші інструменти для створення художніх та дизайнерських ресурсів для проєкту.

Adobe AIR (Adobe Integrated Runtime) — це крос-платформенне середовище виконання (runtime environment), розроблене спочатку компанією Adobe Systems, а з 2019 року підтримуване та розвивається компанією HARMAN. Воно дозволяє розробникам створювати та розгортати настільні (desktop), мобільні (mobile) та телевізійні (TV) додатки, використовуючи єдину кодову базу, написану з використанням веб-технологій (HTML, CSS, JavaScript) або технологій на основі Flash/ActionScript (ActionScript 3.0, SWF).

Основна сутність AIR як крос-платформенної системи полягає в наступному:

- Розробники пишуть код додатку один раз, використовуючи знайомі їм технології (будь то веб-стандарти чи ActionScript). Цей початковий код є спільним для всіх цільових платформ.

- На кожній підтримуваній операційній системі (Windows, macOS, Android, iOS, а історично також BlackBerry Tablet OS та деякі платформи для TV) інсталюється або вбудовується спеціальне середовище виконання AIR. Саме в цьому середовищі виконується код додатку. Це дозволяє додатку працювати незалежно від особливостей конкретної ОС.

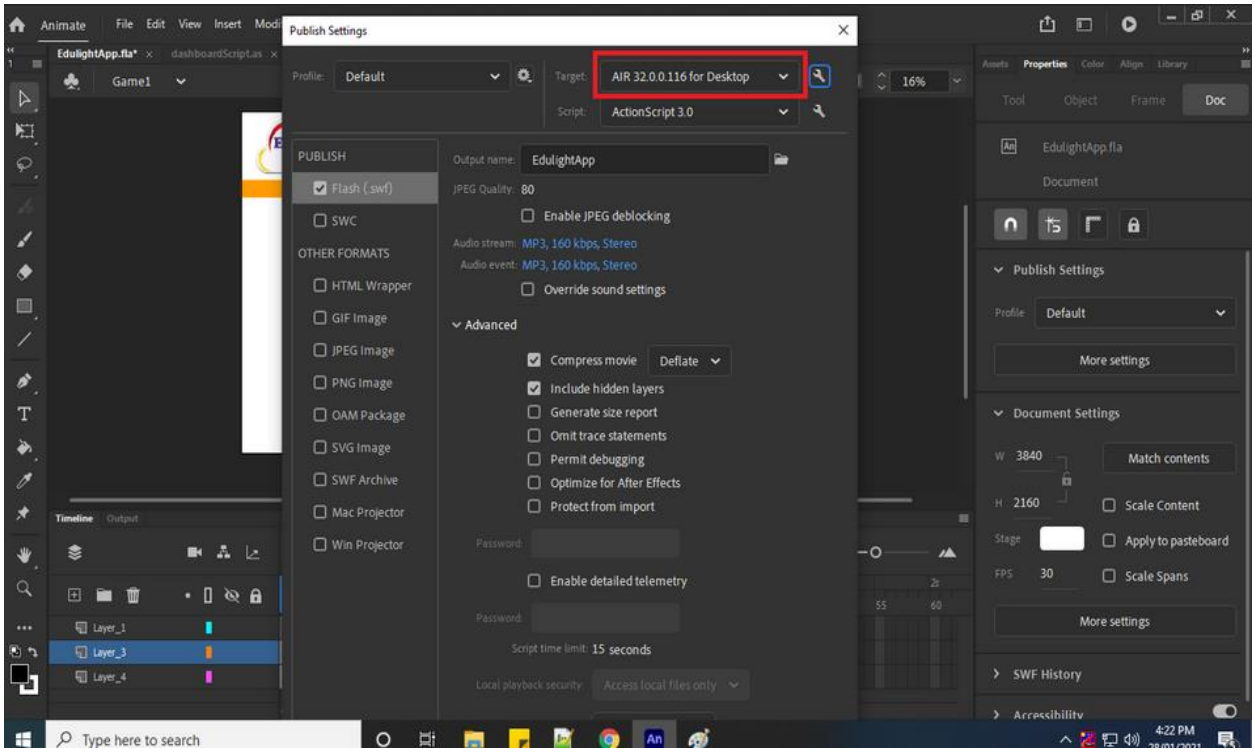

Рисунок 1.5 - Adobe Animate

- Незважаючи на роботу в ізольованому середовищі, AIR надає розробникам доступ до багатьох нативних функцій пристрою або операційної системи через спеціалізовані API. Це включає роботу з файловою системою, мережею, локальними базами даних (наприклад, SQLite), а на мобільних пристроях – доступ до камери, мікрофона, акселерометра тощо.

- AIR дозволяє упакувати додаток разом із середовищем виконання (або з вимогою його попередньої інсталяції) у стандартні для кожної платформи дистрибутиви (наприклад, .exe для Windows, .dmg для macOS, .apk для Android, .ipa для iOS). Це робить AIR-додатки зручними для розповсюдження та інсталяції як звичайні програми для конкретної платформи.

- Система орієнтована на розробників, які вже мають досвід роботи з Flash/ActionScript або веб-технологіями, дозволяючи їм створювати повноцінні крос-платформенні додатки без необхідності вивчати нативні мови програмування для кожної ОС окремо.

Незважаючи на те, що пік популярності Adobe AIR вже минув і в сучасній крос-платформенній розробці домінують інші технології, він залишається прикладом однієї з перших широкодоступних платформ, що дозволила створювати додатки для різних операційних систем з єдиного початкового коду. Його спадщина та існуючі додатки все ще підтримуються.

FlashDevelop — це безкоштовне інтегроване середовище розробки (IDE) з відкритим вихідним кодом, яке було особливо популярним серед розробників, що працюють з ActionScript 3.0 (AS3) та Haxe. Воно призначалося для створення застосунків та ігор, націлених на Adobe Flash Player та Adobe AIR, а також підтримувало розробку на Haxe для різних платформ.

Основні характеристики та можливості FlashDevelop включають:

1. Безкоштовність та відкритий вихідний код - це одна з головних переваг, яка робила його доступним для широкого кола розробників, включно з незалежними та студентами.

2. Легкість та швидкодія, оскільки FlashDevelop відомий своєю високою швидкістю запуску та відносно низьким споживанням ресурсів порівняно з деякими іншими IDE, особливо тими, що базуються на Eclipse (як Flash Builder).

3. Потужний редактор коду - надає якісне підсвічування синтаксису, розумне автодоповнення коду (IntelliSense), навігацію по коду, рефакторинг та підтримку сніпетів, що значно прискорювало процес написання коду на AS3 та Haxe.

4. Забезпечує зручні інструменти для організації файлів проєкту, налаштування параметрів компіляції та управління збірками.

5. Має тісну інтеграцію з Adobe Flex SDK (який містить компілятор AS3) та компілятором Haxe, дозволяючи легко компілювати проєкти.

6. Включає різноманітні шаблони для швидкого створення нових типів проєктів

7. Начвність системи плагінів, що дозволяє розширювати функціональність IDE за допомогою сторонніх плагінів.

FlashDevelop часто обирають як альтернативу офіційним IDE від Adobe, зосереджуючись саме на оптимізації процесу написання та налагодження коду, а не на візуальному дизайні, який був сильнішою стороною інших інструментів.

Хоча з падінням популярності Flash Player та зменшенням використання Adobe AIR актуальність FlashDevelop як основної IDE для AS3 також значно знизилася, він все ще використовується розробниками, які підтримують такі проєкти на цих технологіях, або тими, хто продовжує розробляти на Haxe.

Висновки до розділу

В даному розділі було розглянуто основні концепції та підходи до крос-платформенної розробки, а також особливості створення ігор, орієнтованих на роботу на різних типах пристроїв, включаючи настільні системи, мобільні платформи, веб-браузери та ігрові консолі. Показано, що

крос-платформенність є ключовим чинником підвищення охоплення аудиторії, ефективності розробки та зменшення витрат часу й ресурсів.

Окрему увагу приділено вивченню актуальних інструментів та ігрових рушіїв, таких як Unity, Corona (Solar2D), Cocos2D JS та Appcelerator Titanium. Було здійснено порівняльний аналіз їхніх можливостей, функціональності та доцільності застосування у різних контекстах розробки ігор. На основі цього аналізу зроблено висновки щодо оптимального вибору рушія залежно від вимог до гри, її цільових платформ та технічних ресурсів команди розробників.

У підсумку визначено цілі та обсяг проєктної реалізації, а також сформовано набір інструментів для ефективної реалізації мобільної версії крос-платформенної гри. Отримані результати слугуватимуть базою для практичної частини проєкту, пов’язаної безпосередньо з реалізацією та впровадженням програмного продукту.

РОЗДІЛ 2. РОЗРОБКА АРХІТЕКТУРИ ТА ДИЗАЙНУ МОБІЛЬНОГО ВАРІАНТУ КРОС-ПЛАТФОРМЕННОЇ ГРИ

2.1. ActionScript та ієрархія списку відображення

ActionScript — це мова програмування, яка використовується для розробки Flash-додатків. ActionScript є об'єктно-орієнтованою мовою та синтаксично схожою на Java. Він також є подієво-орієнтованим і працює з обробкою подій і зворотними викликами так само, як і в JavaScript. Кожен додаток ActionScript також має ієрархію об'єктів відображення під назвою список відображення. Це показано на рисунку 2.1.

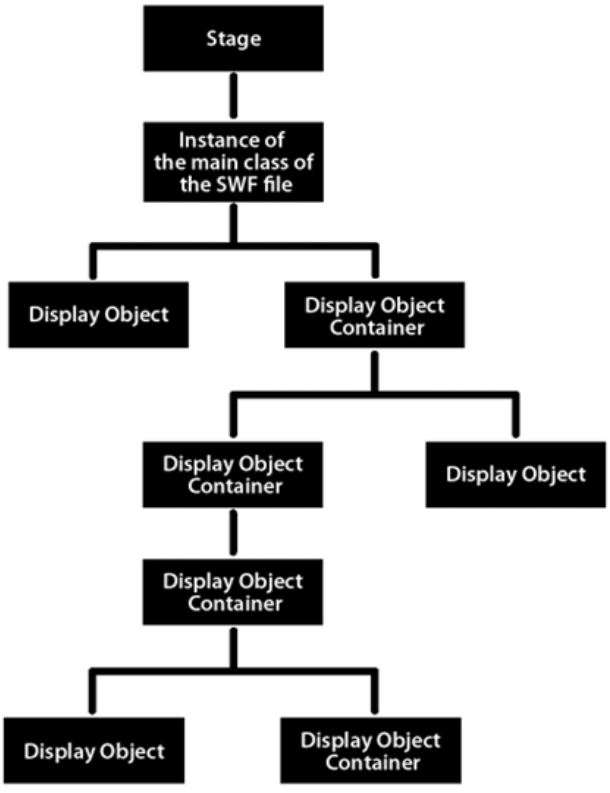

Рисунок 2.1 - Ієрархія списку відображення

Список відображення Flash містить усі візуальні елементи в додатку. Кожен додаток має єдину сцену, яка діє як базовий контейнер для об'єктів відображення. Кожен файл має головний клас, який AIR інстанціює та додає

до сцени одразу після запуску. Об'єкт відображення — це будь-який елемент, який відображається на екрані та слугує базовим класом, який розширюється багатьма іншими класами, такими як спрайти, кліпи фільмів, векторні форми, текстові поля, кнопки, щоб назвати кілька. Об'єкт контейнера відображення також є підкласом об'єкта відображення, який має додаткову функціональність для зберігання дочірніх об'єктів відображення. Перехід та управління списком відображення інтуїтивно зрозумілі та прості.

Starling — це бібліотека ActionScript з відкритим вихідним кодом, яка відтворює традиційну архітектуру списку відображення Flash. Однак об'єкти відображення в Starling відображаються безпосередньо GPU через Stage3D (конвеєр рендерингу GPU від Adobe). Це апаратне прискорення забезпечує швидше рендеринг [3]. Starling також надає багато інших функцій, необхідних для розробки 2D-ігор, таких як текстурні атласи для анімації спрайт-листів, бібліотека твінінгу та система частинок, щоб назвати кілька. Starling також є базовою платформою для бібліотеки інтерфейсу користувача під назвою Feathers. Згідно з [5], "Feathers об'єднує все в одному пакеті: надзвичайно швидке рендеринг GPU, вражаюча кількість варіантів скінінгу та розширювана архітектура компонентів... щоб створити плавний і реактивний досвід". Кожен компонент GUI, такий як кнопки, мітки та піктограми в ANTics, походить з бібліотеки Feathers.

2.2. Реалізація класів Game та AssetManager

Клас Game — це перший клас, який інстанціюється, і він є першим нащадком, доданим до сцени. Усі наступні об'єкти відображення в додатку будуть додані під об'єкт відображення Game. Основні завдання класу Game — створити початкові змінні додатку, читати та записувати в файлову систему, призупиняти та відновити стан гри, завантажувати та відображати стартовий екран, інстанціювати клас AssetManager, створити навігатор

екранів та перейти до першого екрана додатку. Його властивості та методи в основному статичні та доступні для всіх класів у додатку. Більшість цих статичних властивостей складаються з об'єктів Texture, TextureAtlas, ParticleEmitter та Sound, які містять усі художні та звукові ресурси. Інші статичні властивості зберігають дані, які представляють прогрес гравця, які пізніше будуть збережені в файловій системі, такі як найвищий результат гравця та найкращий час. Статичні методи складаються з різних функцій, таких як управління музикою та звуковими ефектами, збереження та завантаження даних про прогрес гравця та метод скидання, який скидає всі дані про прогрес гравця до їхнього початкового стану. Рисунки 2.2 – 2.4 показують UML-діаграми класів Game та AssetManager.

Рисунок 2.2 - UML-діаграма класу Game

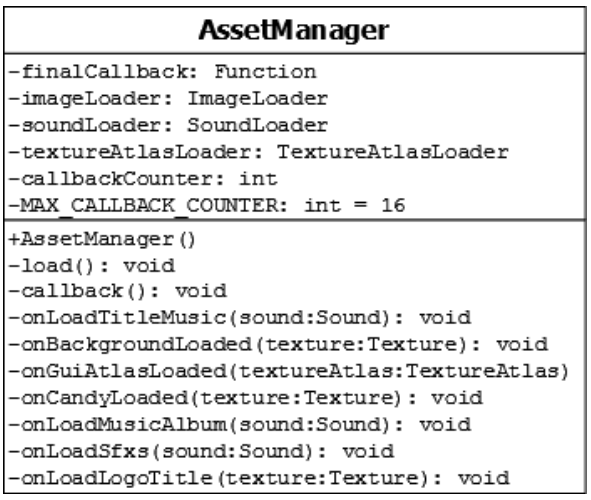

Рисунок 2.3 - UML-діаграма класу AssetManager

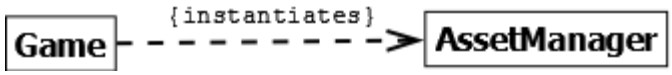

Рисунок 2.4 - UML-діаграма зв'язку між Game і AssetManager

Після інстанціювання класу Game перше, що він робить, — це слухає, коли він був доданий до сцени, і потім викликає свій метод init. Метод init ініціалізує його властивості, додає слухачів подій на випадок втрати та відновлення фокусу, завантажує дані про прогрес гравця та завантажує стартовий екран. Стартовий екран потім відображається, який показує гравцеві логотип гри. Поки відображається стартовий екран, клас Game інстанціює клас AssetManager, який завантажує решту початкових ігрових ресурсів. Таким чином, стартовий екран слугує екраном завантаження. Ресурси складаються з музики, текстур для титульного екрану, даних системи частинок та текстурних атласів для жуків та елементів GUI. Усі ці ресурси завантажуються паралельно та відстежуються за допомогою лічильника зворотних викликів. Цей лічильник зворотних викликів збільшується за допомогою методу зворотного виклику, який викликає кожен завантажувач після завершення завантаження свого вмісту. Коли лічильник зворотних викликів дорівнює кількості ресурсів, призначених для

завантаження, метод зворотного виклику викличе метод finalCallback, який є методом зворотного виклику, переданим у його конструктор класом Game. Рисунок 2.5 показує код класу AssetManager, який ілюструє цей процес.

```
public function AssetManager(callback:Function)
{
    finalCallback = callback;
    callbackCounter = 0;
}

// Завантаження початкових ресурсів. Ресурси завантажуються паралельно.
public function load() :void
{
    textureAtlasLoader = new TextureAtlasLoader("atlas/gui", onGuiAtlasLoaded);
    imageLoader = new ImageLoader("loadFeatures/logoTitle", onLoadLogoTitle);
    //...14 інших завантажувачів ресурсів створюються тут
}

// Кожного разу, коли ресурс завершує завантаження, викликається ця функція.
private function callback() :void
{
    if (callbackCounter == MAX_CALLBACK_COUNTER) // всі ресурси завантажені
    {
        finalCallback();
    }
    else
    {
        callbackCounter++;
    }
}

private function onGuiAtlasLoaded(textureAtlas:TextureAtlas):void
{
    Game.guiAtlas = textureAtlas;
    callback();
}
private function onLoadLogoTitle(texture:Texture):void
{
    Game.logoTitleTexture = texture;
    callback();
}
//...14 інших методів зворотного виклику визначено тут
```

Рисунок 2.5 - Код AssetManager

Метод finalCallback викликає метод fadeOutSplashScreen класу Game. Цей метод викликає метод createScreen після завершення операції твінінгу для зникнення стартового екрану. Метод createScreen встановлює тему

Feathers для всіх елементів GUI, інстанціює навігатор екранів та переходить до титульного екрану. Рисунок 2.6 показує код методів fadeOutSplashScreen і createScreen класу Game.

```
{
    var tween:Tween = new Tween(splash, 1);
    tween.fadeTo(0);
    Starling.juggler.add(tween);
    tween.onComplete = createScreens;
    progressBar.removeFromParent(true);
}

// Створює всі екрани для гри та навігатор екранів.
private function createScreens():void
{
    chmenMusic = MusicConstants.TITLE_MUSIC;
    // Встановлює скіни для всіх елементів GUI на основі теми Feathers
    var theme:MetalWorkMobileThemeExtension = new MetalWorkMobileThemeExtension(stage,
    removeChild(splash);
    splash.dispose();
    splash = null;
    screens = new Screens();
    screenNavigator = new ScreenNavigator();
    addChild(screenNavigator);
    // Встановлює ефект переходу для екранів.
    transition = new ScreenSlidingStackTransitionManager(screenNavigator);
    transition.duration = DURATION;
    screenNavigator.addScreen(screens.TITLE_SCREEN, new ScreenNavigatorItem(TitleScree
    screenNavigator.addScreen(screens.GAMEPLAY_SCREEN, new ScreenNavigatorItem(Gamepla
    screenNavigator.addScreen(screens.CREDITS, new ScreenNavigatorItem(CreditsScreen))
    isOnSplash = false;
    // Показує перший екран для нашої гри
    screenNavigator.showScreen(screens.TITLE_SCREEN);
}
```

Рисунок 2.6 - Код методів fadeOutSplashScreen і createScreen

Тема — це компонент у Feathers, який використовується для скінінгу кожного елемента GUI в додатку. "Тема Feathers — це клас, який пакує код скінінгу для кількох компонентів інтерфейсу користувача в одному місці. Скіни реєструються глобально, і коли будь-який компонент Feathers інстанціюється, він автоматично отримує скін" [5]. Я розширив стандартну тему, яку надає Feathers, підкласом, який я називаю MetalWorkMobileThemeExtension. Це дозволяє додати додаткову функціональність до теми для подальшої кастомізації компонентів GUI.

ScreenNavigator — це клас, який надає бібліотека Feathers. "Контейнер, схожий на стопку відображення, який підтримує навігацію між екранами за допомогою подій" [8]. Кожен клас ігрового екрану в цьому додатку розширює клас Feathers Screen і додається до навігатора екранів. Коли потрібно перейти до певного екрана, я просто викликаю метод showScreen на screenNavigator і передаю константну строку, яка асоціюється з цільовим екраном. Метод showScreen скидає поточний екран, якщо він існує, інстанціює цільовий екран, додає його до списку відображення та переходить до нього.

2.3. Проектування UML-діаграми класу TitleScreen

Перший екран, який навігатор екранів додає до списку відображення та переходить до нього, — це титульний екран. Рисунок 2.7 показує UML-діаграму класів TitleScreen і Option.

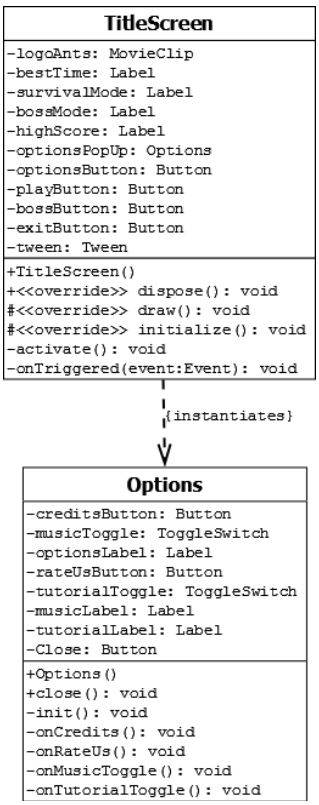

Рисунок 2.7 - UML-діаграма класів TitleScreen і Option

Після того, як титульний екран додається до списку відображення, навігатор екранів викликає його метод initialize, потім викликає його метод draw, і, нарешті, відображає його об'єкти відображення. Я перевизначаю метод draw, щоб встановити положення та масштаб фонових зображень і меню кнопок та додати їх до списку відображення. Спочатку кнопки меню розташовуються за межами екрана, а потім твінінгом переміщуються в поле зору. Після завершення операції твінінгу викликається метод activate. Метод activate додає всі слухачі подій для кожної кнопки меню. Обробники подій для кнопок режимів гри та босів переходять до екрана Gameplay Screen і встановлюють прапорець у класі Game, який повідомляє екрану Gameplay Screen, чи потрібно працювати в режимі виживання чи в режимі босів. Кнопка налаштувань відображає поп-ап екран налаштувань. Цей поп-ап екран налаштувань — це об'єкт Options, який не є класом екрана Feathers і не додається до навігатора екранів. Замість цього Options інстанціюється класом TitleScreen і додається як дитина до об'єкта відображення TitleScreen. Клас Options надає прості налаштування, які користувач може змінити, такі як перемикання музики та інструкцій, а також кнопку, яка переводить користувача з додатку в магазин для оцінки додатку. Нарешті, титульний екран має кнопку виходу, яка просто закриває додаток.

2.4. Реалізація діаграм класів забезпечення ігрової механіки

Екран гри GameplayScreen забезпечує основну ігрову механіку. Термін "ігрова механіка" означає набір правил, процесів та систем, які визначають, як гравець може взаємодіяти з ігровим світом та як ігровий світ реагує на ці дії. Це фундаментальні будівельні блоки, що формують ігровий процес. Він ініціалізує та оновлює жуків, цукерки та HUD. Рисунки 2.8 - 2.11– це UML-діаграми класів GameplayScreen, HUD і PlayAreaSprite.

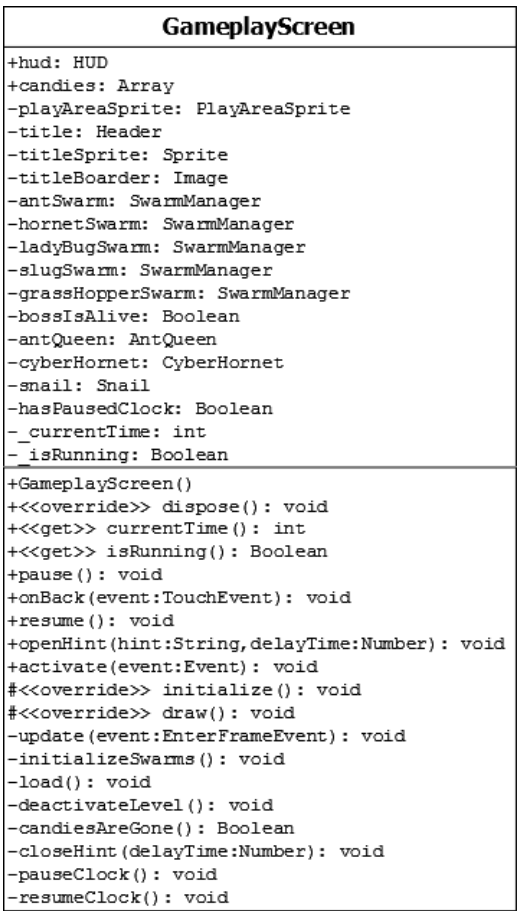

Рисунок 2.8 - UML-діаграма класу GameplayScreen

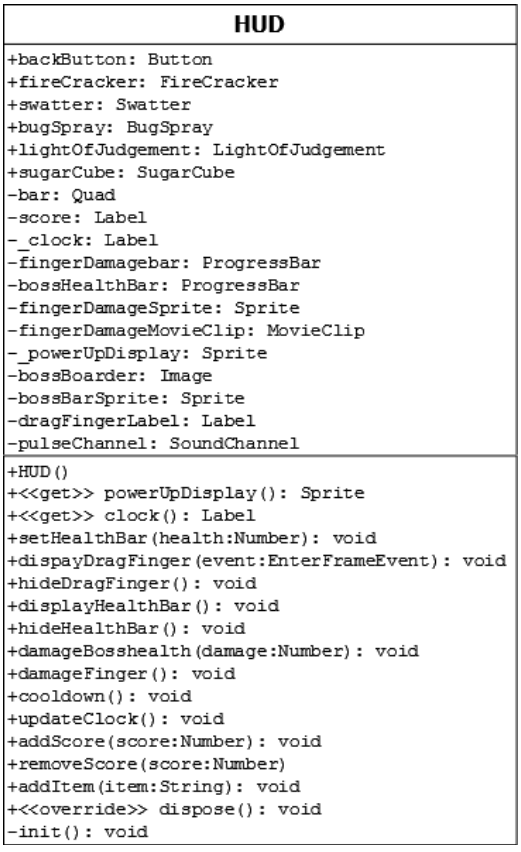

Рисунок 2.9 - UML-діаграма класу HUD

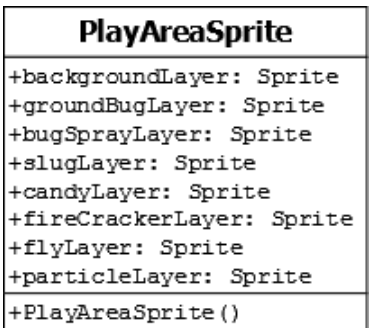

Рисунок 2.10 - UML-діаграма класу PlayAreaSprite

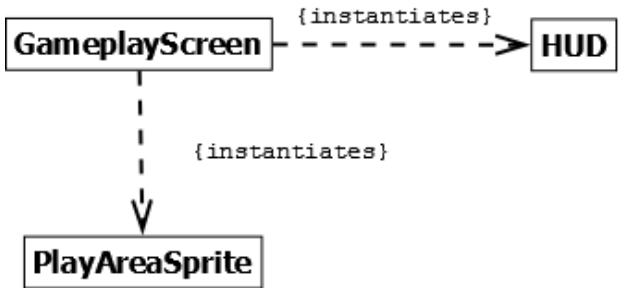

Рисунок 2.11 - UML-діаграма зв'язку між Gameplay, HUD і PlayAreaSprite

Є два режими, в яких працює екран гри Gameplay Screen: режим виживання та режим босів. Це визначається прапорець isBossMode у класі Game.

Перше, що робить екран гри, — це перевіряє, чи був встановлений цей прапорець. Це визначає, яку музику потрібно програти, яких жуків потрібно створити спочатку та яку ігрову логіку потрібно запускати. Перший об'єкт відображення, який екран гри додає як дитину до свого об'єкта відображення, — це екземпляр PlayAreaSprite.

PlayAreaSprite розширює Sprite і діє як контейнер для всіх інших об'єктів Sprite, які стосуються ігрового процесу та відображаються на екрані. Ці об'єкти Sprite ігрового процесу включають фон, цукерки, жуків та підсилювачі. Ігрове поле спрайта складається з кількох екземплярів Sprite, які діють як шари. Ці шари забезпечують правильний порядок відображення об'єктів Sprite ігрового процесу. Наприклад, шар спрайта літаючих жуків відображається поверх шару спрайта землі. Тому жуків, які літають, додають

до шару літаючих жуків, а жуків, які повзають, — до шару землі. Це гарантує, що літаючі жуки завжди відображаються поверх повзаючих жуків. Останній об'єкт відображення, який екран гри додає до списку відображення, — це HUD.

2.5. Проектування та опис компонентів класу HUD

HUD складається з чотирьох основних компонентів:

- titleBar,
- powerUpDisplay,
- fingerDamageSprite,
- bossBarSprite.

TitleBar — це екземпляр класу Sprite, який містить об'єкт Image та об'єкт заголовка Feathers. Об'єкт Image діє як шкіра для покращення зовнішнього вигляду titleBar. Об'єкт заголовка відображає мітку поверх кольорового фону. Мета titleBar — відображати повідомлення користувачеві під час гри. Одним із прикладів є повідомлення "Get Ready!", яке слугує екраном завантаження, коли користувач входить на екран гри Gameplay Screen. Воно залишається відображеним, поки не завершаться завантаження всіх необхідних ресурсів. Інші приклади включають: серію інструктивних повідомлень, які пояснюють, як грати в гру, повідомлення "Game Over!" коли гравець програє в режимі виживання чи босів, і, нарешті, повідомлення "You Win", коли гравець виграє в режимі босів.

Power-up display — це екземпляр Sprite, який містить п'ять екземплярів класу PowerUp як свої діти. Інші діти, які він містить, — це мітки, які відображають інформацію про годинник та результат гравця, а також екземпляр класу Feathers Button, який дозволяє користувачеві вийти з екрана гри. Power-up display має об'єкт Quad та об'єкт Image, які слугують простими

шкірами. Основне призначення power-up display — слугувати інтерфейсом користувача для гравця для активації функціональності підсилювачів.

Finger damage sprite — це екземпляр Sprite, який містить об'єкт MovieClip та об'єкт Feathers ProgressBar як свої діти. Об'єкт finger damage sprite відображається лише тоді, коли користувач доторкається до об'єкта Hornet. Коли він відображається, об'єкт MovieClip запускає анімацію пульсуючого пальця, а смуга прогресу відображається зі значенням, яке поступово зменшується. Це повідомляє гравцеві, що він не зможе розчавити іншого жука, поки не завершиться час охолодження.

Boss bar sprite — це екземпляр Sprite, який містить об'єкт Image та об'єкт Feathers ProgressBar як свої діти. Об'єкт Image відображає текстуру шкали боса, а смуга прогресу представляє HP боса. Коли бос отримує пошкодження, значення смуги прогресу зменшується. Об'єкт boss bar sprite відображається лише в режимі босів і коли об'єкт Boss з'являється на сцені.

2.6. Розробка діаграм класів SwarmManager і SwarmPool

Екран гри створює, ініціалізує та оновлює п'ять екземплярів класу SwarmManager, по одному для кожного типу жука (за винятком босів). Клас SwarmManager відповідає за управління пулом жуків одного типу. Цей пул об'єктів — це екземпляр класу SwarmPool. Рисунок 2.12 — це UML-діаграма класів SwarmManager і SwarmPool.

Клас SwarmPool використовує оптимізаційну техніку під назвою шаблон пулу об'єктів. Шаблон пулу об'єктів — це програмний креаційний шаблон проектування, який використовує набір ініціалізованих об'єктів, які готові до використання з "пулу", а не виділяються та знищуються за потребою. Клієнт пулу запитує об'єкт з пулу та виконує операції з повернутого об'єкта [7].

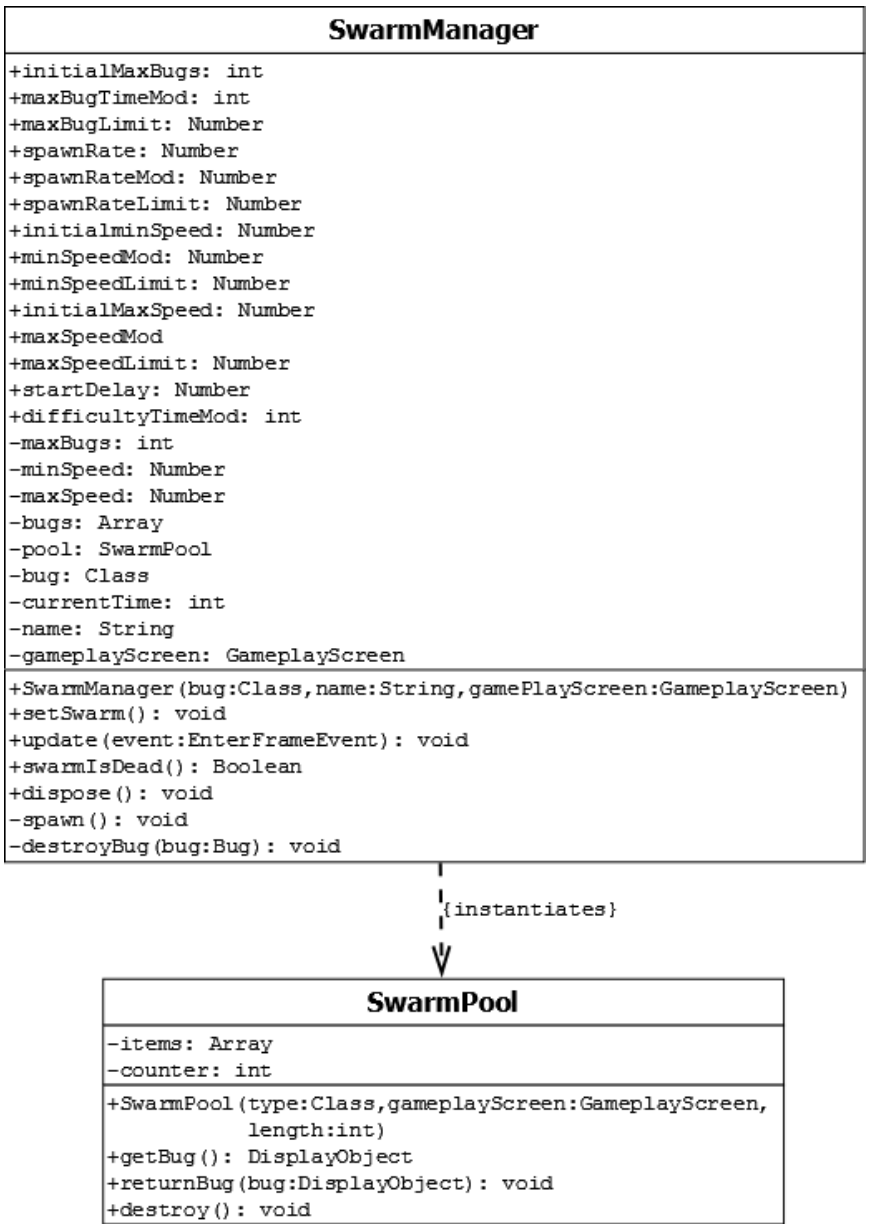

Рисунок 2.12 - UML-діаграма класів SwarmManager і SwarmPool

Клас SwarmPool створює статичний масив заданої довжини, який містить екземпляри жуків одного типу. Клас SwarmManager викликає метод getBug свого об'єкта пулу кожного разу, коли йому потрібно дістати жука з пулу. Це видаляє жука з масиву в об'єкті пулу та додає його до масиву, яким керує SwarmManager. Це виконується в методі spawn класу SwarmManager, як показано на рисунку 2.13.

```
private function spawn():void
{
    var bug:Bug = pool.getBug() as Bug;
    bugs.push(bug);
    bug.initialize(minSpeed, maxSpeed);
}
```

Рисунок 2.13 - Код методу spawn

Аналогічно, кожного разу, коли класу SwarmManager потрібно знищити жука, він вирізає його з масиву та повертає в пул. Це виконується в методі destroyBug класу SwarmManager, як показано на рисунку 2.14.

```
private function destroyBug(bug:Bug):void
{
    for (var i:int = 0; i < bugs.length; i++)
    {
        if (bug == bugs[i])
        {
            bugs.splice(i, 1);
            pool.returnBug(bug);
        }
    }
}
```

Рисунок 2.14 - Код методу destroyBug

Коли класу SwarmManager більше не потрібен, викликається його метод dispose. Це знищить масив жуків та об'єкт пулу. Код цього методу показано на рисунку 2.15.

```
public function dispose():void
{
    if (pool != null)
    {
        pool.destroy();
        pool = null;
    }
    for (var i:int = 0; i < bugs.length; i++)
    {
        bugs[i].dispose();
    }
    bugs = [];
}
```

Рисунок 2.15 - Код методу dispose

Метод initializeSwarms екрана гри ініціалізує всі властивості класу SwarmManager, такі як maxBugLimit, minSpeed, maxSpeed, spawnRate та багато інших. Ці властивості балансують складність кожного жука, що, у свою чергу, визначає складність гри. Кожен тип жука має клас балансу, який зберігає ці дані балансування, з яких метод initializeSwarms бере. Наприклад, рисунок 2.16 показує частковий код методу initializeSwarms, який ініціалізує екземпляр SwarmManager для типу мурашок. Рисунок 2.17 показує код класу AntBalance.

```
antSwarm.startDelay = AntBalance.startDelay;
antSwarm.initialMaxBugs = AntBalance.initialMaxBugs;
antSwarm.initialMinSpeed = AntBalance.initialMinSpeed;
antSwarm.initialMaxSpeed = AntBalance.initialMaxSpeed;
antSwarm.spawnRate = AntBalance.initialSpawnRate;
antSwarm.maxBugTimeMod = AntBalance.maxBugTimeMod;
antSwarm.maxBugLimit = AntBalance.maxBugLimit;
antSwarm.minSpeedMod = AntBalance.minSpeedMod;
antSwarm.minSpeedLimit = AntBalance.minSpeedLimit;
antSwarm.maxSpeedMod = AntBalance.maxSpeedMod;
antSwarm.maxSpeedLimit = AntBalance.maxSpeedLimit;
antSwarm.spawnRateMod = AntBalance.spawnRateMod;
antSwarm.spawnRateLimit = AntBalance.spawnRateLimt;
antSwarm.difficultyTimeMod = AntBalance.difficultyTimeMod;
antSwarm.setSwarm();
```

Рисунок 2.16 - Код методу initializeSwarms

```
public class AntBalance
{
    public static const startDelay:Number = 0; // час у секундах, який потрібен цьому жуку, щоб
    // початкові властивості складності
    public static const initialMaxBugs:int = 3; // максимальна кількість жуків, які з'являються
    public static const initialSpawnRate:Number = 0.1; // ймовірність того, що цей жук з'явитьс
    public static const initialMinSpeed:Number = 1; // найменша швидкість, з якою рухається цей
    public static const initialMaxSpeed:Number = 2; // найбільша швидкість, з якою рухається це
    public static const maxBugTimeMod:int = 20; // час у секундах, який потрібен для збільшення
    public static const difficultyTimeMod:int = 10; // час у секундах, який потрібен для застос
    // Модифікації додаються до властивостей складності за їхньою кількістю кожні x секунд. Де
    public static const spawnRateMod:Number = 0.001;
    public static const minSpeedMod:Number = 0.1;
    public static const maxSpeedMod:Number = 0.1;
    // Обмеження. Модифікації не збільшують властивості складності вище цієї межі.
    public static const maxBugLimit:Number = 15; // ймовірно, не варто встановлювати це більше
    public static const spawnRateLimt:Number = 1; // це повинно бути значення 0 > n <= 1
    public static const minSpeedLimit:Number = 5;
    public static const maxSpeedLimit:Number = 6;
}
```

Рисунок 2.17 - Вихідний код класу AntBalance

Це робить балансування зручним, оскільки можна просто змінити значення, збережені в класі AntBalance, щоб зробити мурашок у грі простішими або складнішими.

Після ініціалізації екземплярів класу SwarmManager метод update екрана гри викликає метод update на кожному об'єкті SwarmManager. Метод update класу SwarmManager відстежує поточний час у секундах, щоб збільшити складність наступного жука, який з'явиться після певної кількості секунд. Ця кількість секунд визначається модифікаційним значенням під назвою difficultyTimeMod.

Складність кожного жука збільшується шляхом додавання модифікаційного значення до властивостей minSpeed, maxSpeed та spawnRate жука. Ці модифікаційні значення встановлюються в методі initializeSwarms екрана гри, як згадувалося вище. Також існує ще одне модифікаційне значення часу під назвою maxBugTime, яке визначає, коли збільшувати максимальну кількість жуків, яких можна створити. Кожна з цих властивостей балансування також має обмежувальне значення, яке запобігає додаванню модифікацій після певної точки.

2.7. Реалізація класів Bug

Є дев'ять різних типів жуків у грі, і всі вони успадковуються від загального класу Bug. Рисунок 4.17 показує UML-діаграму базового класу Bug, а рисунок 4.18 ілюструє спадкоємність.

Клас Bug містить властивості, спільні для всіх типів жуків. До цих властивостей належать, але не обмежуються ними, масиви кліпів фільмів для анімацій, зображення, які відображаються, коли жука вбивають, властивості, які відстежують його поточний напрямок та стан ШІ, а також властивості балансування, такі як minSpeed, maxSpeed, hp, maxHp та crunchPower.

Властивість crunchPower визначає, скільки HP жук віднімає від цукерки, яку він їсть.

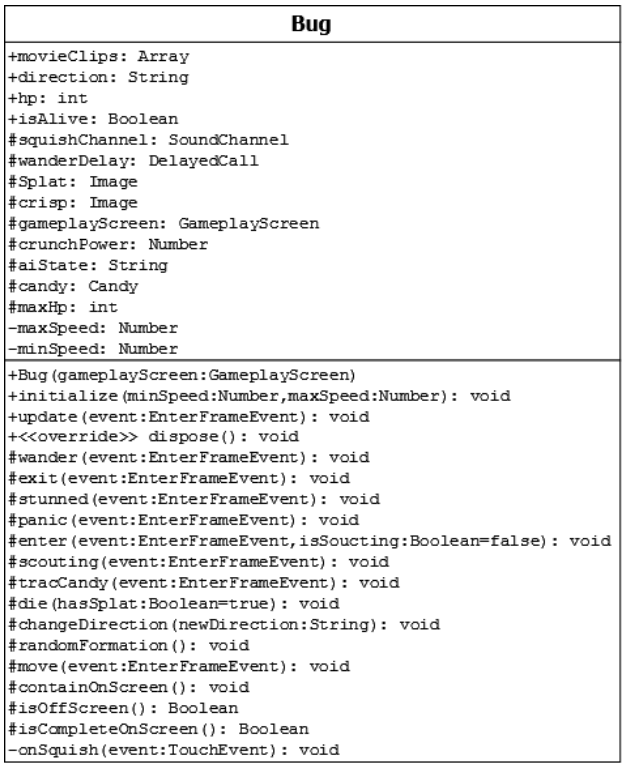

Рисунок 2.18 - UML-діаграма класу Bug

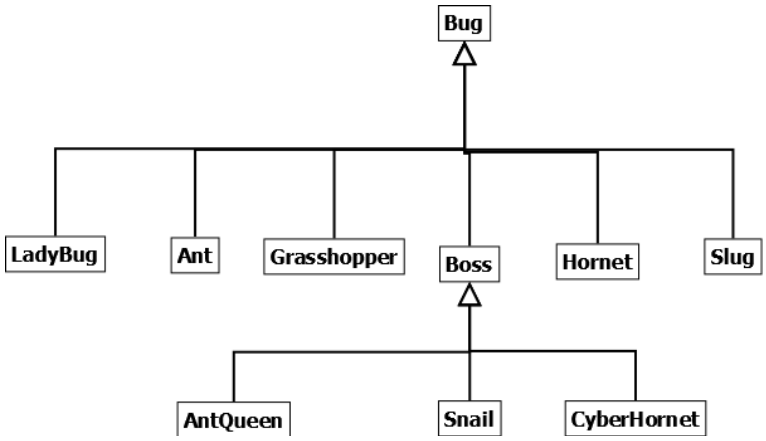

Рисунок 2.19 - UML-діаграма спадкування класів Bug

Конструктор класу Bug, який перевизначається підкласами, ініціалізує властивості, які не змінюються протягом життєвого циклу жука. До цих властивостей належать, але не обмежуються ними, кліпи фільмів,

зображення, max HP, початковий стан ШІ та crunchPower. Навпаки, метод initialize ініціалізує змінні, які змінюються протягом життєвого циклу жука. До цих змінних належать, але не обмежуються ними, minSpeed, maxSpeed, стан ШІ, напрямок та прапорець isAlive. Оскільки я використовую дизайн пулування, мені потрібно використовувати цей метод initialize, щоб повторно ініціалізувати той самий об'єкт жука в пам'яті кожного разу, коли його дістають з пулу. Метод initialize викликається в методі spawn класу SwarmManager (рисунок 2.16).

Метод onSquish класу Bug — це обробник події, який викликається, коли жука доторкається пальцем гравця. Метод onSquish зменшує HP жука, якщо гравець не був пошкоджений шершнем. Потім він викликає метод die, якщо HP дорівнює нулю. Метод die встановлює прапорець isAlive на true, відображає відповідне зображення смерті та видаляє жука зі списку відображення.

Метод update класу Bug оновлює стан жука на основі його поточного стану ШІ. Стан ШІ отримує значення константи з класу AIState. Поведінка жука змінюється залежно від його поточного стану ШІ. Рисунок 2.20 показує частковий код методу update, який ілюструє цей процес.

```
if (aiState != AIStates.DEAD)
{
    switch(aiState)
    {
        case AIStates.TRACK_CANDY:
        {
            trackCandy(event);
        }
        case AIStates.WANDER:
        {
            wander(event);
            break;
        }
        case AIStates.ENTER:
        {
            enter(event);
            break;
        }
        case AIStates.EXIT:
        {
            exit(event);
            break;
        }
        //...Ще чотири стани ШІ йдуть сюди.
        default:
        {
            break;
        }
    }
}
```

Рисунок 2.20 – Код методу update класу Bug

Просто кажучи, на основі його поточного стану ШІ викликається відповідний метод. Ці методи можуть бути перевизначені підкласами класу Bug для надання унікальної поведінки. Крім того, кожен підклас класу Bug переходить між підмножиною цих станів ШІ інакше, ніж інші. Щоб проілюструвати цей момент, нижче наведено опис процесу переходу ШІ двох підкласів класу Bug: класу Ant і класу Hornet.

Клас Ant ініціалізує свій стан ШІ до "Enter", який викликає метод enter. Клас Ant перевизначає стандартну поведінку методу enter і встановлює свій прапорець isScouting від false до true. Це повідомляє методу enter, що після того, як він приведе жука на екран, він повинен перейти свій стан ШІ до "Scouting" замість переходу до "Wandering", який є стандартною поведінкою. Метод scouting потім викликається в наступному оновленні. Цей метод змусить мурашку рухатися випадково по екрану, поки її прямокутник зіткнення не перетинається з прямокутником зіткнення об'єкта Candy. Потім стан ШІ мурашки перейде до "Track Candy", і метод trackCandy буде викликаний в наступному оновленні. Метод trackCandy змусить мурашку рухатися ближче до цукерки, з якою вона зіткнулася, і потім додасть її crunchPower до властивості crunchPower об'єкта Candy. Мурашка залишатиметься в цьому стані, поки цукерка не зникне, і потім перейде назад до стану ШІ "Scouting". Мурашка продовжуватиме переходити таким чином, поки її не вб'ють.

Поведінка ШІ шершня відрізняється. Подібно до класу Ant, він ініціалізує свій стан ШІ до "Enter", але залишає стандартну поведінку методу enter, переходячи до "Wander". Він перевизначає метод wander, змінюючи його стандартну функціональність з випадкового руху невизначено до руху випадково протягом певної кількості часу. Коли цей час минає, метод wander перейде стан ШІ до "Exit". Метод exit просто виводить шершня за межі екрана, а потім викликає метод die.

Подібно до оновлення стану жука на основі його станів ШІ, метод update також змінює поведінку жука залежно від того, який підсилювач зараз активований. Спочатку метод update перевіряє на наявність імунітету. Імунітет означає, що жука не реагує на підсилювач і, отже, не змінює свою поведінку. Для підсилювача цукровий кубик, коли цукровий кубик додається до ігрового поля спрайта, метод update призначає цукровий кубик як об'єкт Candy для відстеження. Для підсилювача феєрверк метод update викликає метод kill жука тільки тоді, коли об'єкт FireCrackerSprite, з яким він зіткнувся, має встановлений прапорець isExploding на true.

Жуки типу босів працюють майже так само, як і будь-які інші жуки. Вони мають унікальну поведінку ШІ та перевизначають певні методи для додавання додаткової функціональності. Однак є кілька відмінностей. По-перше, їх HP відображається як шкала здоров'я на HUD. По-друге, вони не керуються менеджером рою, оскільки існує лише три екземпляри босів, які коли-небудь створюються, один для кожного типу. Нарешті, вони інстанціюються лише тоді, коли гравець грає в режимі босів, і лише один екземпляр боса відображається на полі в один час. Метод die боса повідомляє екран гри, коли він мертвий, а потім екран гри виведе наступного боса.

2.8. Представлення класу Candy

Екран гри інстанціює масив об'єктів Candy і додає їх до шару цукерок PlayAreaSprite. Метод update екрана гри оновлює кожен об'єкт Candy і також перевіряє, чи всі цукерки зникли. Якщо так, екран гри відобразить "Game Over" на HUD і потім перейде до титульного екрану. Рисунок 2.21 показує UML-діаграму класу Candy.

Основні властивості класу Candy в основному складаються з candyImage, healthBar, hp, crunchPower та crunchChannel. Об'єкт candyImage використовується для відображення однієї з п'яти текстур цукерок. Під час

оновлення для candyImage призначається нова текстура залежно від поточного стану HP цукерки. Шкала здоров'я — це об'єкт ProgressBar, який використовується для відображення стану HP цукерки користувачеві. Стан HP зменшується на crunchPower кожне оновлення. Канал crunchChannel — це об'єкт класу Flash SoundChannel. Клас SoundChannel є частиною бібліотеки Flash і використовується для керування звуком у додатку. Канал crunchChannel використовується для програвання звукового ефекту хрускіння, коли цукерку їсть жук.

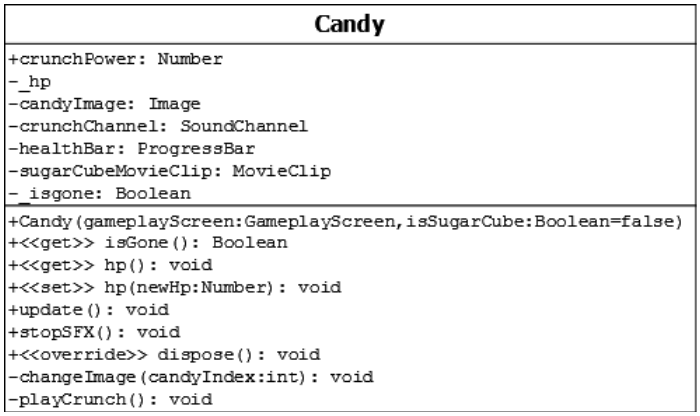

Рисунок 2.21 - UML-діаграма класу Candy

Клас Candy також діє як підсилювач цукровий кубик. Конструктор класу Candy приймає булеве значення, щоб визначити, чи потрібно відображати цукерку як цукерку чи як цукровий кубик. Різниця полягає в тому, що якщо відображати як цукровий кубик, клас Candy використовує кліп фільму замість зображення для відображення своїх текстур, оскільки мистецтво для цукрового кубика має анімацію, а мистецтво для цукерки — ні.

2.9. Класи Power-up

Є п'ять типів класів підсилювачів у грі: SugarCube, BugSpray, Swatter, MagnifyingGlass і FireCracker. Ці класи всі успадковуються від класу

PowerUp. Рисунок 2.22 показує UML-діаграму базового класу PowerUp, а рисунок 2.23 ілюструє спадкоємність.

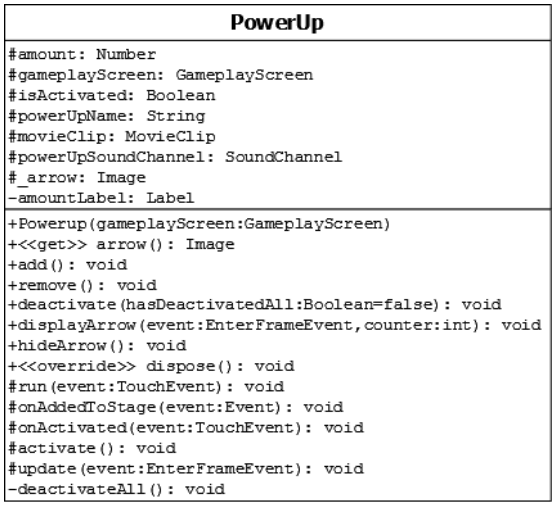

Рисунок 2.22 - UML-діаграма класів PowerUp

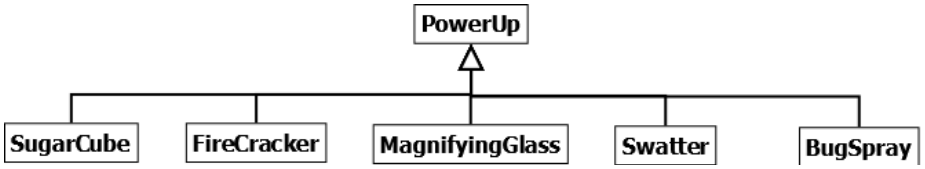

Рисунок 2.23 - UML-діаграма спадкування класів PowerUp

Основні властивості базового класу PowerUp складаються з кліпу фільму, змінної amount, звукового каналу powerUpSoundChannel та прапора isActivated. Кліп фільму використовується для відображення анімованої піктограми підсилювача користувачеві, яка відображається на HUD. Змінна amount відстежує, скільки підсилювачів зараз доступно у гравця. Звуковий канал підсилювача програє звуковий ефект, пов'язаний з типом підсилювача. Прапорець isActivated відстежує, коли поточний підсилювач активований. Підсилювач працюватиме лише тоді, коли цей прапорець встановлений на true.

Мова програмування ActionScript не підтримує абстрактні функції чи класи. Однак я розглядаю дві функції в базовому класі PowerUp так, ніби

вони були абстрактними. Ці функції — run та update. Реалізація обох цих функцій здійснюється лише в підкласах. Функція run — це обробник події, який викликається, коли гравець доторкається до ігрового поля спрайта. Кожна реалізація методу run підкласами спочатку перевіряє, чи встановлений прапорець isActivated на true, перш ніж виконувати свою логіку run. Це гарантує, що працює лише поточний активований підсилювач. Метод update — це обробник події входу в кадр, який викликається кожний кадр. Кожен підклас додає цей слухач події до свого об'єкта відображення. Кожен підклас реалізує методи run і update по-різному. Рисунок 4.23 показує код методу run класу MagnifyingGlass.

```
{
    if (isActivated)
    {
        if (event.getTouch(stage) != null)
        {
            var touch:Touch = event.getTouch(stage);
            globalX = touch.globalX;
            globalY = touch.globalY;
            if(touch.phase == TouchPhase.BEGAN)// on finger down
            {
                PowerUpMode.mode = powerUpName;
                particleEmiter.particles.start();
                addEventListener(starling.events.Event.ENTER_FRAME, update);
                powerUpSoundChannel = Game.playSfx(SfxConstants.fire);
                onComplete) {
                isDown = true;
            }
            if(touch.phase == TouchPhase.ENDED) // on finger up
            {
                particleEmiter.particles.stop();
                removeEventListener(starling.events.Event.ENTER_FRAME, update);
                powerUpSoundChannel.stop();
                powerUpSoundChannel.removeEventListener(flash.events.Event.SOUND_COMPLETE,
                onComplete);
                deactivate();
                isDown = false;
            }
        }
    }
}
```

Рисунок 2.24 - Код методу run класу MagnifyingGlass

Метод run відстежує положення та натискання пальця гравця. Коли палець опускається, статична властивість mode в класі PowerUpMode встановлюється на назву підсилювача, який зараз працює. Це забезпечує те,

щоб жуки знали, як реагувати, коли активований підсилювач. Метод також запускає емітер частинок і зациклює звуковий канал підсилювача. Додається слухач події входу в кадр, який викликає метод update кожного кадру. Коли палець піднімається, емітер частинок і звуковий канал підсилювача зупиняються, слухачі подій видаляються, а підсилювач деактивовано. Рисунок 2.25 показує код методу update класу MagnifyingGlass.

```
protected override function update(event:EnterFrameEvent):void
{
    if (gamePlayScreen.isRunning)
    {
        if (energy > 0)
        {
            energy=event.passedTime;
            energyBar.value = energy / MAX_ENERGY * 100;
            particleEmitter.update(globalX, globalY);
        }
        else
        {
            remove();
            if (amount > 0)
            {
                energy = MAX_ENERGY;
            }
            else
            {
                powerUpSoundChannel.stop();
                powerUpSoundChannel.removeEventListener(flash.events.Event.SOUND_COMPLETE,
                onComplete);
                particleEmitter.particles.stop();
                deactivate();
                removeEventListener(starling.events.Event.ENTER_FRAME, update);
            }
        }
    }
    else
    {
        powerUpSoundChannel.stop();
        powerUpSoundChannel.removeEventListener(flash.events.Event.SOUND_COMPLETE,
        onComplete);
        particleEmitter.particles.stop();
        deactivate();
        removeEventListener(starling.events.Event.ENTER_FRAME, update);
    }
}
```

Рисунок 2.25 - Код методу update класу MagnifyingGlass

Метод update оновлює стан змінної energy, яка відстежує, скільки часу гравець може утримувати лупу активною. Коли енергія досягає нуля, змінна amount зменшується. Якщо amount залишається більше нуля, енергія ініціалізується знову. Якщо amount дорівнює нулю, лупа деактивовано. Коли

енергія більше нуля, метод update викликає метод update емітера частинок і передає йому поточні позиції дотику. Це дозволяє ефекту частинок слідувати за пальцем гравця. Метод update також деактивує лупу, якщо екран гри перейшов у стан "Game Over".

Метод run класу Swatter встановлює положення його квадратного об'єкта та спрайта swatter на основі позиції дотику і робить їх видимими, коли палець гравця опускається. Він також додає затримку виклику, яка чекає частки секунди, перш ніж деактивувати підсилювач. Клас Swatter не реалізує метод update.

Метод run класу SugarCube створює новий об'єкт Candy, встановлює його позицію, додає його до шару цукерок PlayAreaSprite та додає його до масиву, коли палець гравця опускається. Метод update викликає метод update на першому об'єкті Candy в масиві. Коли метод update бачить, що прапорець isGone об'єкта Candy встановлений на true, він видаляє його з масиву, видаляє його з ігрового поля спрайта та, нарешті, скидає його.

Метод run класу FireCracker слухає події дотику гравця, додає новий об'єкт FireCrackerSprite до масиву, встановлює його позицію, додає його до шару firecracker PlayAreaSprite та викликає його метод explode. Метод update класу FireCracker скидає об'єкти FireCrackerSprite, якщо їхній прапорець isExploded встановлений.

Клас FireCrackerSprite в основному складається з кліпу фільму для анімації, квадрата для прямокутника зіткнення та емітера частинок для ефектів частинок. Рисунок 2.26 показує код методів explode і update класу FireCrackerSprite.

Метод explode має обробник затримки виклику, який чекає одну секунду, перш ніж виконувати свою функцію зворотного виклику. Функція зворотного виклику програє звуковий ефект вибуху, оновлює емітер вибуху та додає слухача події входу в кадр, який викликає його метод update кожного кадру. Метод update перевіряє, чи немає більше частинок в емітері, і

потім встановлює прапорець isExploded на true. Це вказує на те, що об'єкт FireCrackerSprite більше не використовується, і клас FireCracker може його скинути.

```
public function explode():void
{
    const explosionTimer:Number = .5;
    movieClip.currentFrame = 0;
    const timer:Number = 1;
    soundChannel = Game.playSfx(SfxConstants.lit);
    saveSfx = SfxConstants.lit;
    var delay:DelayedCall = new DelayedCall(function() :void {
        Starling.juggler.remove (delay);
        soundChannel.stop();
        soundChannel = Game.playSfx(SfxConstants.explosion);
        saveSfx = SfxConstants.explosion;
        Game.explodeEmitter.update(x + (width / 2), y + (height / 2)) ;
        Game.explodeEmitter.particles.start(explosionTimer);
        Game.explodeEmitter.particles.advanceTime(.1);
        movieClip.alpha = 0;
        isExploding = true;
        addEventListener(Event.ENTER_FRAME, update);
        }, timer);
    Starling.juggler.add (delay);
}

private function update(event:EnterFrameEvent):void
{
    if (Game.explodeEmitter.particles.numParticles == 0)
    {
        isExploded = true;
        isExploding = false;
    }
}
```

Рисунок 2.26 - Код методів explode і update класу FireCrackerSprite

Клас BugSpray — це єдиний підсилювач, який не реалізує методи run і update. Вся функціональність спрею від жуків реалізована через його метод activate. Коли гравець доторкається до кліпу фільму спрею від жуків на HUD, підсилювач активовано та відразу використовується. Рисунок 2.27 показує вихідний код методу activate класу BugSpray.

Метод activate запускає емітер частинок і програє звуковий ефект, а також змінює колір кліпу фільму на червоний, щоб повідомити гравцеві, що спрей від жуків зараз використовується і його не можна активовувати знову,

поки він не закінчить. Є дві затримки виклику, перша з яких зупиняє емітер частинок, а друга — деактивує підсилювач.

```
override protected function activate():void
{
    if (movieClip.color == Color.WHITE) // активовувати тільки тоді, коли спрей від жуків ще не
    {
        super.activate();
        isSpraying = true;
        PowerUpMode.mode = powerUpName;
        powerUpSoundChannel = Game.playSfx(SfxConstants.spray);
        movieClip.color = Color.RED;
        particleEmitter.particles.start();
        remove();
        // Зупинити ефект частинок. Дві затримки потрібні, оскільки ефект частинок продовжує ві
        delay = new DelayedCall(function stop() :void {
            powerUpSoundChannel.stop();
            Starling.juggler.remove(delay);
            particleEmitter.particles.stop();
            }, 5);
        // Деактивувати підсилювач
        delay2 = new DelayedCall(function stop() :void {
            Starling.juggler.remove(delay2);
            movieClip.color = Color.WHITE;
            deactivate();
            }, 5);
        Starling.juggler.add(delay);
        Starling.juggler.add(delay2);
    }
}
```

Рисунок 2.27 - Код методу activate класу BugSpray

2.10. Реалізація класі DropItem, ParticleEmitter, PDparticleSystem і PexLoader

Клас DropItem використовується для додавання більше підсилювачів до HUD або для поповнення HP цукерки. Рисунок 2.28 — це UML-діаграма класу DropItem.

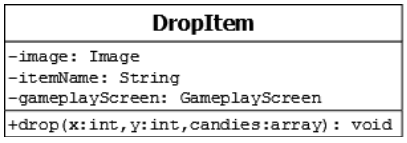

Рисунок 2.28 - UML-діаграма класу DropItem

Гравець накопичує підсилювачі та поповнює цукерки, вбиваючи божих жуків. Клас LadyBug інстанціює новий об'єкт DropItem кожного разу, коли

він ініціалізується. Клас DropItem призначає рядок з випадково вибраною назвою підсилювача або цукерки. Цей рядок визначає дві дії: якого типу підсилювач додати до HUD або чи потрібно поповнити HP цукерки. Ці дії відбуваються, коли викликається його метод drop. Метод drop викликається всередині методу die класу LadyBug.

Клас ParticleEmitter надає інтерфейс для системи частинок у грі. Система частинок — це екземпляр класу Starling PDparticleSystem. Дані для системи частинок завантажує клас PexLoader. Рисунок 2.29 — це UML-діаграма цих трьох класів.

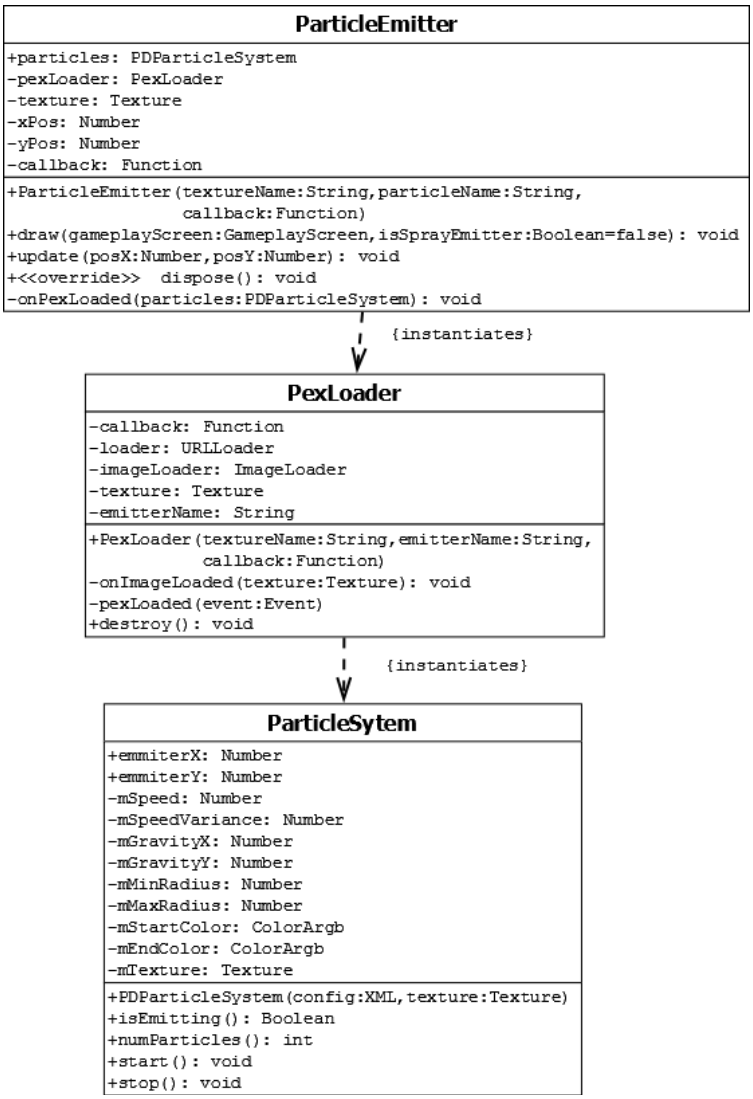

Рисунок 2.29 - UML-діаграма класів ParticleEmitter, PDparticleSystem і PexLoader

Клас ParticleEmitter складається з об'єкта PexLoader, об'єкта Texture, об'єкта PDparticleSystem класу Starling під назвою particles, та двох змінних для відстеження позицій x та y емітера. Клас ParticleEmitter в своєму конструкторі завантажує ресурси, створюючи екземпляр PexLoader. Рисунок 2.30 показує конструктори класів ParticleEmitter і PexLoader.

```
public function ParticleEmitter(textureName:String, particleName:String, callback:Function)
{
    pexLoader = new PexLoader(textureName, particleName, onPexLoaded);
    this.callback = callback;
}



public function PexLoader(textureName:String, emitterName:String, callback:Function)
{
    this.callback = callback;
    this.emitterName = emitterName;
    imageLoader = new ImageLoader("particles/"+textureName, onImageLoaded);
}
```

Рисунок 2.30 – Конструктори класів ParticleEmitter і PexLoader

Клас PexLoader завантажує текстуру та файл XML, який визначає поведінку частинок. Дані завантажуються послідовно. Рисунок 2.31 показує код методів зворотного виклику для кожного з цих завантажених ресурсів.

```
private function onImageLoaded(texture:Texture):void
{
    this.texture = texture;
    loader = new URLLoader();
    loader.load(new URLRequest("assets/images/particles/"+emitterName+".pex"));
    loader.addEventListener(Event.COMPLETE, pexLoaded);
}

private function pexLoaded(event:Event): void
{
    loader.removeEventListener(Event.COMPLETE, pexLoaded);
    var xml:XML = XML(Loader.data);
    callback(new PDParticleSystem(xml, texture));
}
```

Рисунок 2.31 - Методи зворотного виклику класу PexLoader

Після завершення остаточного завантаження метод pexLoaded створює об'єкт PDParticleSystem, який приймає дані XML та текстуру. Потім

викликається метод зворотного виклику, який був переданий у його конструктор класом ParticleEmitter. Цей метод зворотного виклику — це метод onPexLoaded класу ParticleEmitter, який показано на рисунку 2.32.

```
private function onPexLoaded(particles:PDParticleSystem):void
{
    this.particles = particles;
    callback();
}
```

Рисунок 2.32 - Метод onPexLoaded класу PexLoader

Метод onPexLoaded отримує об'єкт PDParticleSystem і призначає його властивості particles. Потім він викликає метод зворотного виклику, який був переданий у конструктор класу ParticleEmitter. Метод draw класу ParticleEmitter додає його властивість particles до шару частинок на GameplayScreen PlayAreaSprite. Цей метод показано на рисунку 2.33.

```
public function draw(gameplayScreen:GameplayScreen, isSprayEmitter:Boolean = false):void
{
    Starling.juggler.add(particles);
    if (!isSprayEmitter)
    {
        gameplayScreen.playAreaSprite.particleLayer.addChild(particles);
    }
    else
    {
        gameplayScreen.playAreaSprite.bugSprayLayer.addChild(particles);
    }
}
```

Рисунок 2.33 - Метод draw класу ParticleEmitter

Цей метод відображає частинки на екрані в правильному порядку відображення. Метод update просто оновлює положення частинок. Рисунок 2.34 показує код цього методу.

```
public function update(posX:Number, posY:Number):void
{
    particles.emitterX = posX;
    particles.emitterY = posY;
}
```

Рисунок 2.34 - Метод update класу ParticleEmitter

Властивість particles — це екземпляр класу Starling PDParticleSystem. Властивості emitterX та emitterY відстежують положення частинок. Він також має методи start та stop, які використовуються для відображення частинок. Ці два методи викликаються безпосередньо на властивості particles іншими класами в інших місцях додатку.

Висновки до розділу

У другому розділі було здійснено поетапну розробку архітектури мобільного варіанту крос-платформенної гри з використанням об’єктно-орієнтованого підходу. Розглядалася специфіка використання мови програмування ActionScript, зокрема концепція списку відображення (display list), що відіграє важливу роль у графічному поданні об’єктів.

Були реалізовані ключові класи, які формують ядро ігрової логіки, серед яких Game, AssetManager, SwarmManager, SwarmPool, Bug, Candy, DropItem, а також елементи системи візуалізації — ParticleEmitter, PDParticleSystem та PexLoader. Вони забезпечують керування ресурсами, відтворення анімації, генерацію об’єктів гри та візуальні ефекти.

Окремо було приділено увагу проектуванню інтерфейсу користувача через реалізацію класу HUD (Head-Up Display) і проектування UML-діаграм, зокрема TitleScreen, що забезпечує зручну навігацію та інтерактивну взаємодію з гравцем.

Реалізовані компоненти демонструють добре структуровану архітектуру, яка сприяє масштабованості, повторному використанню коду та підтримці гнучкої ігрової логіки. Розроблені діаграми класів та обґрунтовані проектні рішення створюють надійне підґрунтя для подальшої реалізації функціональності та тестування ігрового застосунку.

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ІНТЕРФЕЙСУ МОБІЛЬНОГО ВАРІАНТУ КРОС-ПЛАТФОРМЕННОЇ ГРИ

3.1. Опис розроблюваної гри

Розроблюваний проєкт являє собою приклад казуальної відеогри, що функціонально відповідає жанру Tower Defense (захист вежі), де основним завданням гравця є оборона трьох стратегічних об'єктів — накопичень ресурсів (цукеркових "куп") — від агресивного впливу різноманітних ентомологічних агентів (різних видів жуків).

Розглянемо основні механіки ігрового процесу.

1. Генерація та рух ворогів. Ентомологічні агенти з'являються випадковим чином із трьох дискретних точок входу по периметру ігрового поля. Їхня траєкторія руху спрямована на досягнення та поглинання одного з трьох цільових об'єктів на полі. Різні види агентів можуть відрізнятися за швидкістю пересування, стійкістю до пошкоджень або іншими параметрами.

2. Взаємодія гравця. Оборона об'єктів здійснюється двома основними методами:

- Пряма Елімінація – це коли користувач може безпосередньо взаємодіяти з ентомологічними агентами через сенсорний інтерфейс (торкання) для їх усунення.

- Застосування Тактичних Модулів (Підсилювачів) - коли гравець має доступ до п'яти унікальних допоміжних інструментів (підсилювачів), кожен з яких надає специфічні тактичні переваги:

- Мухобійка (Fly Swatter) - забезпечує одноразове, точкове усунення цілі або значне пошкодження.

- Лупа (Magnifying Glass) - може функціонувати як засіб нанесення безперервного пошкодження по площі протягом певного часу.

- Спрей від жуків (Bug Spray) - застосовується для уповільнення руху групи агентів або нанесення періодичного пошкодження по площі.

- Цукрові кубики (Sugar Cubes) - призначені для відновлення або поповнення цільових накопичень ресурсів.

- Феєрверки (Fireworks) - спричиняють значне пошкодження по площі в момент активації.

Гра завершується поразкою гравця у випадку повного вичерпання всіх трьох накопичень ресурсів (цукерок).

Проєкт передбачає два основні режими функціонування, що визначають мету та критерії оцінки:

- Режим виживання (Survival Mode) - основна мета полягає у максимальному продовженні часу оборони цільових об'єктів. Критерії оцінки результативності включають кількість успішно усунутих ентомологічних агентів та тривалість ігрової сесії до моменту поразки.

- Режим босів (Boss Mode) - метою є послідовна елімінація трьох унікальних, значно більш потужних ентомологічних агентів (босів). Перемога досягається після успішного усунення останнього боса. Оцінка ефективності у цьому режимі базується на кількості усунутих рядових агентів та швидкості проходження всіх етапів (часі, витраченому на перемогу над усіма босами).

Таким чином, дана гра пропонує стандартизовані для жанру Tower Defense механіки захисту цільових об'єктів від хвиль ворогів, доповнені системою тактичних підсилювачів та двома режимами гри з різними цілями та системами оцінки результатів.

Tower Defense (TD), або "Захист вежі" — це популярний піджанр стратегічних відеоігор. Основна ідея жанру полягає в тому, що гравець повинен захистити певну територію, об'єкт або "базу" від наступаючих хвиль ворожих підрозділів, будуючи та покращуючи оборонні споруди, які зазвичай називаються "вежами"

3.2. Реалізація ігрової механіки

3.2.1. Розробка діаграми потоку екранів

Гра має простий сенсорний інтерфейс з кнопками GUI, які перемикають між кількома ігровими екранами. Це показано на діаграмі потоку екранів на рисунку 3.1.

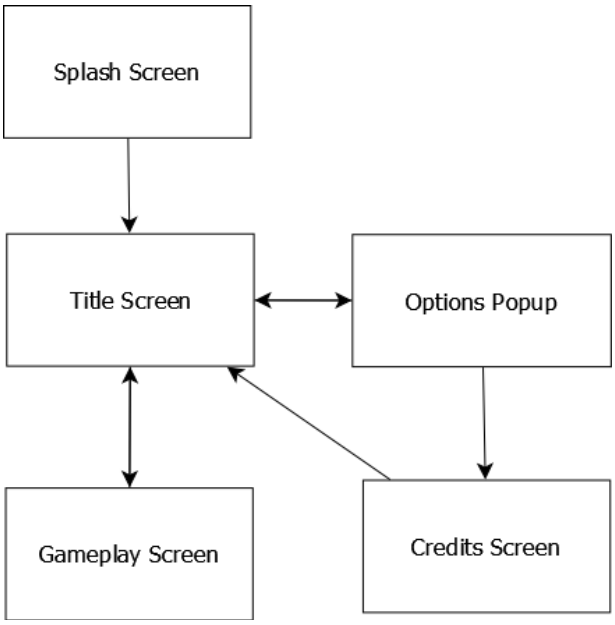

Рисунок 3.1 - Діаграма потоку екранів

Першим екраном, який з'являється при запуску додатку, є стартовий екран. Цей екран відображає логотип гри протягом короткого часу, поки завантажуються деякі ресурси. Потім він переходить до титульного екрану.

3.2.2. Розробка початкового екрану та екрану налаштувань

Титульний екран діє як інтерфейс для доступу до різних компонентів гри. Він має кнопку, яка відкриває екран налаштувань, дві кнопки, які перемикають на екран Gameplay Screen, і кнопку, яка закриває додаток. Це показано на рисунку 3.2.

Проведемо опис елементів показаних на рисунку 3.2

Верхня частина екрана займає яскравий, стилізований фон, що нагадує небо з хмарами у верхній частині та темним, абстрактним ландшафтом або землею нижче.

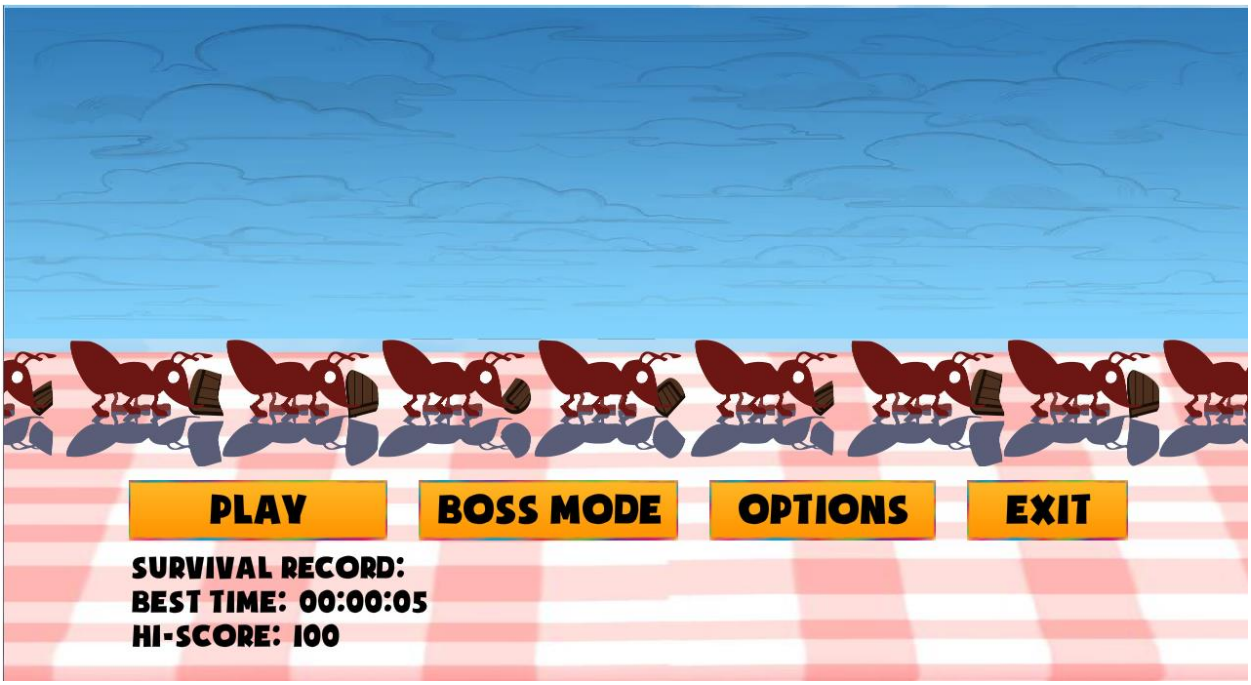

Рисунок 3.2 - Знімок екрану Title Screen

Під фоном розташований ряд ідентичних коричневих стилізованих мурах. Кожен з них тримає в роті чорний квадратний об'єкт. Вони повторюються по горизонталі екрана, створюючи враження "навали".

Нижче ряду мурах знаходяться чотири великі прямокутні кнопки помаранчевого кольору з чорним текстом, розташовані горизонтально:

- PLAY (Грати);
- BOSS MODE (Режим босів);
- OPTIONS (Опції);
- EXIT (Вихід).

Нижче наведена інформація про рекорди. Під кнопками відображається текстова інформація, що стосується ігрових досягнень:

- SURVIVAL RECORD: (Рекорд виживання:)
- BEST TIME: 00:00:05 (Найкращий час: 00:00:05)

- HI-SCORE: 100 (Найкращий результат: 100)

Візуальний стиль гри є простим, мультяшним, з чіткими контурами та яскравими кольорами. Це типово для казуальних або мобільних ігор.

Скріншот відображає початковий інтерфейс гри, який дозволяє гравцеві вибрати режим гри, налаштування або вийти, а також показує його попередні найкращі результати в режимі виживання.

Екран налаштувань діє як наклад поверх титульного екрану. Це показано на рисунку 3.3.

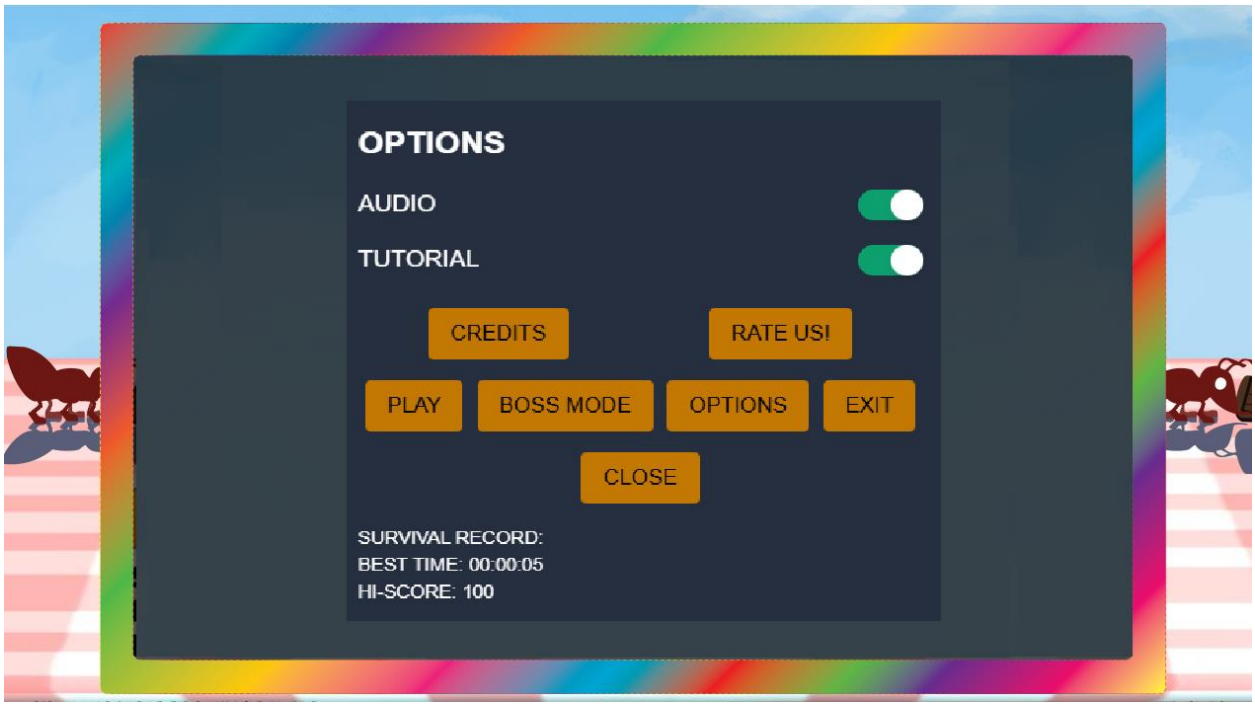

Рисунок 3.3 - Знімок екрану Options

Екран налаштувань дозволяє користувачеві змінювати налаштування гри, такі як увімкнення або вимкнення звуку та інструкцій. Він також має кнопки, які переводять користувача на екран підсумків та кнопку, яка переводить користувача в Google Play Store для оцінки гри.

Екран гри Gameplay Screen запускає різну ігрову логіку та завантажує різні ресурси в залежності від режиму, в якому він працює. Режим визначається кнопкою на титульному екрані, яку користувач активував для

переходу на екран Gameplay Screen. Екран Gameplay Screen переходить назад на титульний екран, коли гравець виграє в режимі босів і програє в обох режимах. Рисунок 3.4 показує знімок екрану Gameplay Screen.

Рисунок 3.4 - Знімок екрану Gameplay Screen

Екран гри Gameplay Screen складається з ігрового поля та HUD (інтерфейсу, який постійно відображається). Ігрове поле складається з фонового зображення столу для пікніка, куп цукерок та жуків. Гравець взаємодіє з ігровим полем, або натискаючи на жуків пальцем, або використовуючи можливості підсилювачів.

HUD — це те, що відображається поверх ігрового поля. У верхній частині екрана HUD використовується для відображення корисних повідомлень користувачеві. У нижній частині екрана HUD використовується для відображення поточного статусу підсилювачів. Користувач може взаємодіяти з цією частиною HUD, натискаючи на піктограму підсилювача, щоб активувати його.

3.3. Реалізація додаткових предметів у грі

3.2.2. Опис предметів-підсилювачів

Підсилювачі — це предмети, які додають гравцеві додаткові можливості для перемоги над жуками. Є п'ять різних підсилювачів, кожен з яких представляє реальний інструмент для знищення жуків. Нижче наведено список підсилювачів та їхні можливості.

Цукровий кубик: цукрові кубики перетягуються та скидаються на ігрове поле. Жуки тимчасово будуть притягуватися до цукрового кубика замість цукерок.

Мухобійка: після активації користувач доторкається до екрана, і зображення мухобійки з'являється в точці дотику. Вона розчавлює жуків всередині її зони зіткнення.

Феєрверк: феєрверки перетягуються та скидаються на ігрове поле. Вони мають затримку на два секунди, а потім вибухають з ефектом частинок, вбиваючи жуків у межах їхньої зони зіткнення.

Лупа: після активації користувач може перетягувати пальцем по ігровому полю, залишаючи слід частинок, схожих на вогонь. Жуки, які потрапляють у цей слід, миттєво гинуть.

Спрей від жуків: після активації все ігрове поле покривається ефектом частинок, схожим на токсичний газ. Усі жуки на екрані сповільнюються на дві секунди, а потім гинуть.

3.3.2. Реалізація об’єктів candy

Цукерки (candy) — це об'єкти, які гравець повинен захищати, щоб залишитися в живих. У гравця є максимум три цукерки. Жуки будуть притягуватися до цих цукерок і намагатимуться їх з'їсти. Рисунок 3.5 показує серію знімків екрану, які показують кожен стан цукерки.

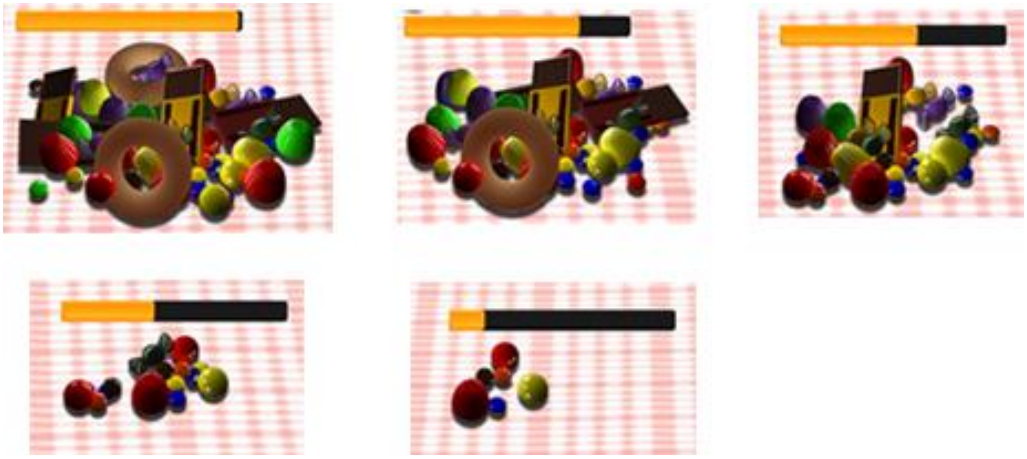

Рисунок 3.5 - Знімки екранів стану об’єктів Candy

У кожної цукерки є смуга здоров'я, яка відображається над нею, щоб показати користувачеві стан її поточного HP (Health Points - очки здоров'я). Залежно від стану HP змінюється зображення цукерки на зображення купи з меншою кількістю цукерок. Коли HP досягає нуля, цукерка вважається з'їденою, і її зображення більше не відображається. Гравець програє гру, коли всі три стопки цукерок з'їдені.

3.3.3. Реалізація об’єктів Bugs

Жуки (Bugs) вважаються ворогами в грі та перешкодами, які гравець повинен подолати. Більшість жуків можна знищити, або натискаючи на них пальцем, або використовуючи один з підсилювачів. Є різні види жуків у грі, кожен з яких має свою поведінку. Нижче наведено список усіх жуків та їхні поведінки.

1. Мурашки. Мурашки одразу йдуть до найближчої купи цукерок зі средньою швидкістю. Вони будуть продовжувати їсти цукерку, поки їх не вб'ють.

2. Коники. Коники блукають по ігровому полю протягом трьох-п'яти секунд, перш ніж йти до купи цукерок. Після того, як вони знайдуть купу цукерок, вони завдадуть значної шкоди та вийдуть з ігрового поля. Вони також потрібні два натискання, щоб розчавити.

3. Шершні. Шершні блукають по ігровому полю протягом п'яти-восьми секунд, а потім виходять. Їх не можна вбити, і якщо користувач натискає на них, він не зможе натискати на іншого жука протягом трьох секунд.

4. Слимаки. Слимаки одразу йдуть до найближчої купи цукерок з повільною швидкістю. Їх потрібно розчавити три рази. Кожен натискання зменшує їхній розмір і збільшує їхню швидкість, що ускладнює натискання на них.

5. Божі жуки. Божі жуки — найшвидші з усіх жуків. Вони блукають по ігровому полю протягом п'яти-восьми секунд, а потім виходять. Якщо їх вбити, вони нагородять гравця підсилювачем або поповнять HP купи цукерок.

6. Королева мурах. Королева мурах блукає по полю та викликає спеціальний вид мурашок, які несуть шматки цукерки до неї, що збільшує її здоров'я. Чим менше її здоров'я, тим більше мурашок вона викличе.

7. Слимак. Слимак блукає по ігровому полю з твердою шкаралупою, яку потрібно спочатку розбити, перш ніж можна буде завдати їй шкоди. Після того, як шкаралупа тріснe, слимак збільшує свою швидкість, що ускладнює натискання на нього. Коли гравець натискає на нього без шкаралупи, він викликає слимаків з його тіла, які йдуть до цукерок.

8. Кібершершень. Кібершершень блукає по полю протягом короткого часу, перш ніж стріляти в цукерку з гармат, прикріплених до його крил. Після стрільби в купу цукерок він знову блукає протягом короткого часу, перш ніж знайти іншу купу цукерок для пошкодження. Він також піднімає щит кожні три-п'ять секунд, що робить його невразливим до всіх атак.

Отже, в цьому розділі було досліджено процеси розробки програмного забезпечення для мобільних пристроїв із застосуванням актуальних інструментальних засобів та технологій. У межах проєкту було розроблено прототип мобільної гри, орієнтований на функціонування на операційних системах Android та iOS. Реалізація проєкту здійснювалася з використанням

крос-платформового середовища виконання у поєднанні з фреймворками Starling (для апаратно прискореної 2D-графіки) та Feathers (для компонентів інтерфейсу користувача). Архітектура гри була спроєктована з урахуванням її потенційної розширюваності для створення інших ігрових продуктів аналогічного типу на базі тієї ж структурної моделі.

Перспективи подальшої роботи включають продовження діяльності у сфері розробки програмного забезпечення для мобільних ігрових платформ з метою поглиблення професійних навичок у галузі ігрового програмування.

Висновки до розділу

У третьому розділі було здійснено практичну реалізацію ключових компонентів інтерфейсу мобільного варіанту крос-платформенної гри. Проведено загальний опис гри, визначено її основну концепцію, а також детально реалізовано основні елементи ігрової механіки.

Розробка діаграми потоку екранів дозволила формалізувати логіку переходів між станами гри, такими як початковий екран, меню налаштувань та ігрове середовище. Реалізовані екрани забезпечують інтуїтивну взаємодію користувача з грою, включаючи можливість змінювати параметри відповідно до особистих уподобань.

Особлива увага приділялася реалізації об’єктів ігрового світу — зокрема, предметів-підсилювачів (power-ups), об’єктів Candy та Bugs, що формують базову динаміку гри. Кожен з цих елементів було спроектовано з урахуванням вимог до крос-платформенності, гнучкості та масштабованості.

Запропоновані рішення забезпечують візуальну привабливість, функціональну стабільність та відповідність очікуванням цільової аудиторії. Таким чином, реалізовані інтерфейсні компоненти закладають основу для цілісного та якісного ігрового досвіду на різних мобільних платформах.

ВИСНОВКИ

У процесі виконання дипломної роботи було здійснено повний цикл розробки мобільної крос-платформенної гри — від теоретичного аналізу предметної області до реалізації архітектури, дизайну та інтерфейсу програмного продукту.

У першому розділі було розглянуто особливості крос-платформенної розробки, класифікацію цільових платформ, а також проаналізовано актуальні інструменти створення ігор. У результаті проведеного аналізу предметної області крос-платформенної розробки ігор було встановлено, що створення ігор для кількох платформ водночас є ефективним підходом до охоплення ширшої аудиторії та оптимізації витрат на розробку. Особливості такої розробки включають необхідність уніфікації логіки програми, адаптивного дизайну інтерфейсів та ефективного управління ресурсами під час підтримки сумісності з різними операційними системами та типами пристроїв. Проведено огляд ігрових рушіїв, таких як Unity, Solar2D, Cocos2D JS та Appcelerator Titanium, з позицій їх функціональних можливостей, підтримуваних платформ і продуктивності. Порівняльний аналіз дозволив обґрунтувати доцільний вибір рушія для реалізації гри.

У другому розділі було розроблено архітектуру мобільного варіанта гри, спроєктовано основні програмні компоненти, зокрема класи Game, AssetManager, TitleScreen, HUD, Bug, Candy, SwarmManager, SwarmPool, тощо. Для кращого розуміння структурних взаємозв’язків між об'єктами були створені UML-діаграми. Реалізація класів забезпечила логічну цілісність гри та її ключових механік.

У третьому розділі було здійснено практичну реалізацію інтерфейсу гри, зокрема розроблено екран вітання, меню налаштувань, діаграму потоку екранів, а також описано і впроваджено ігрову механіку. Особливу увагу приділено реалізації інтерактивних елементів, таких як об’єкти типу Candy,

Bugs, Power-up, що забезпечують динаміку ігрового процесу та підвищують його залученість.

Загалом, результати виконаної роботи демонструють ефективність використання сучасних підходів до крос-платформенної розробки в контексті створення мобільних ігор. Розроблена архітектура та програмна реалізація можуть слугувати основою для подальшого розширення функціональності гри, а також адаптації її під інші цільові платформи.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Ciman, M., Gaggi, O., & Gonzo, N. (2014). Cross-Platform Mobile Development: A Study on Apps with Animations. Proceedings of the 29th Annual ACM Symposium on Applied Computing, 757–759.
2. Martinez, M. (2017). Two Datasets of Questions and Answers for Studying the Development of Cross-platform Mobile Applications using Xamarin Framework. arXiv preprint arXiv:1712.09569.
3. Martinez, M., & Lecomte, S. (2017). Towards the Quality Improvement of Cross-Platform Mobile Applications. arXiv preprint arXiv:1701.06767.
4. Almeida, L. G. G., de Vasconcelos, N. V., Winkler, I., & Catapan, M. F. (2023). Innovating Industrial Training with Immersive Metaverses: A Method for Developing Cross-Platform Virtual Reality Environments. Applied Sciences, 13(1), 123.
5. S. Padmini, M. D. Shafeulwara, P. Sivasankari, & V. Mridula. (2019). Development of Virtual Simulator for Paddle Powered Vehicle. International Journal of Engineering and Advanced Technology, 8(5), 1234–1238.
6. Lucas, G. G. A., de Vasconcelos, N. V., Winkler, I., & Catapan, M. F. (2023). Innovating Industrial Training with Immersive Metaverses: A Method for Developing Cross-Platform Virtual Reality Environments. Applied Sciences, 13(1), 123.
7. Huang, Y., & Zhang, Y. (2015). Cross-Platform Mobile Application Development: A Case Study. International Journal of Software Engineering and Its Applications, 9(12), 161–172.
8. Palmieri, M., Singh, I., & Cicchetti, A. (2012). Comparison of Cross-Platform Mobile Development Tools. Proceedings of the 16th International Conference on Intelligence in Next Generation Networks, 179–186.

9. Malavolta, I., Ruberto, S., Soru, T., & Terragni, V. (2015). End Users' Perception of Hybrid Mobile Apps in the Google Play Store. Proceedings of the 2015 IEEE International Conference on Mobile Services, 25–32.
10. Charland, A., & Leroux, B. (2011). Mobile Application Development: Web vs. Native. Communications of the ACM, 54(5), 49–53.
11. Heitkötter, H., Hanschke, S., & Majchrzak, T. A. (2013). Evaluating Cross-Platform Development Approaches for Mobile Applications. Proceedings of the 8th International Conference on Web Information Systems and Technologies, 120–130.
12. Lehmann, S., & Hummel, O. (2013). Native, Web, or Cross-Platform? A Framework for Mobile App Development. Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference, 593–602.
13. Biørn-Hansen, A., Grønli, T.-M., & Ghinea, G. (2017). A Comparison of Cross-Platform Mobile Development Tools. Proceedings of the 2017 IEEE 24th International Conference on Web Services, 799–802.
14. Rieger, C., & Majchrzak, T. A. (2016). Evaluating the Performance of Cross-Platform Mobile Applications. Proceedings of the 2016 IEEE 23rd International Conference on Web Services, 583–586.
15. Aloraini, S., & Alghamdi, R. (2018). A Comparative Study of Cross-Platform Mobile Application Development Frameworks. International Journal of Computer Applications, 180(47), 1–7. Starling, "Starling Framework," Доступно: http://gamua.com/starling/.
16. Feathers, "Feathers UI Components," Доступно: http://feathersui.com/.
17. Adobe, "Feathers MetalWorksMobileTheme Доступно: https://github.com/feathersui/feathers-themes.
18. Brown, C., & Davis, E. (2021). Challenges and Opportunities in Porting PC Games to Mobile Platforms. Proceedings of the International Conference on Game Development (ICGD '21), 112-120.

19. Garcia, F., & Martinez, R. (2020). Architectural Patterns for Scalable Cross-Platform Mobile Games. International Journal of Interactive Multimedia and Artificial Intelligence, 6(4), 45-58.
20. Lee, S., & Kim, H. (2019). Performance Analysis of Cross-Platform Game Engines on Mobile Devices. IEEE Transactions on Games, 11(3), 250-261.
21. Nguyen, T., & Le, M. (2023). User Interface and Control Adaptation in Mobile Game Ports. Journal of Mobile Computing Research, 8(2), 95-110.
22. Wang, H., & Li, P. (2022). Evaluating the Development Efficiency of Cross-Platform Mobile Game Frameworks. International Journal of Computer Games Technology, 2022, Article ID 8451073.
23. Gonzalez, M., & Rodriguez, L. (2021). Asset Optimization Strategies for Cross-Platform Mobile Game Development. Proceedings of the Conference on Mobile and Ubiquitous Systems (MobiQuitous '21), 345-352.
24. Miller, T., & Wilson, S. (2020). Mobile Game Design and Development Best Practices. CRC Press.
25. Ali, S., & Khan, B. (2019). A Comparative Study of Native vs. Cross-Platform Mobile Game Performance. International Journal of Software Engineering and Its Applications, 13(5), 65-78.
26. Peterson, L., & White, D. (2023). Addressing Hardware Fragmentation in Cross-Platform Mobile Game Development. ACM Transactions on Applied Perception, 20(1), Article 5.
27. Zhao, Y., & Guo, X. (2022). Network Architecture for Real-Time Multiplayer Cross-Platform Mobile Games. International Journal of Networked and Distributed Computing, 10(3), 210-225.
28. Fernandez, A., & Perez, C. (2021). Cross-Platform Development with Unity: A Practical Guide. Packt Publishing.
29. Kim, Y., & Park, J. (2020). A Study on User Experience in Cross-Platform Mobile Game Adaptation. Journal of Human-Computer Interaction, 36(5), 401-418.

30. Harris, B., & Clark, R. (2019). Game Engine Selection Criteria for Cross-Platform Mobile Development. Proceedings of the International Conference on Software Engineering and Applications (SEA '19), 233-240.
31. "Object Pool Design Pattern," Доступно: http://gameprogrammingpatterns.com/object-pool.html.
32. Feathers, "ScreenNavigator,": https://github.com/feathersui/feathers/wiki/ScreenNavigator.
33. Uddin, M., & Islam, M. (2019). A Comparative Analysis of Cross-Platform Mobile Game Development Frameworks: React Native vs. Flutter. International Journal of Computer Applications, 179(46), 1-6.
34. White, J., & Black, P. (2023). Input Method Adaptation in Cross-Platform Mobile Game Ports. Journal of Interactive Systems, 1(1), Article 3.
35. Xu, L., & Zhou, Q. (2022). Cloud Gaming Integration in Cross-Platform Mobile Games. IEEE Access, 10, 78901-78915.

БІБЛІОГРАФІЧНА ДОВІДКА

Тема дипломної роботи: “ Розробка мобільного варіанту крос-платформенної гри ”

Обсяг пояснювальної записки: __77__ аркушів.

Дата закінчення роботи: 10 червня 2025 р.

Підпис студента _____