МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ»

Факультет комп'ютерних наук і технологій
Кафедра прикладної математики і інформатики

Допущений до захисту
Завідувач каф. ПМІ _____ О.А. Дмитрієва
(підпис, дата) (ініціали, прізвище)

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи бакалавра
за напрямом підготовки 050103 «Програмна інженерія»

на тему
«Інструментальна підтримка розробки ігрових додатків»
зі спецчастиною
«Розробка розважального ігрового додатку на базі Unity»

Виконав: студент групи ІПЗ–13 Кривенко О.М.
(прізвище та ініціали)

Керівник проф. каф. ПМіІ, д. т. н. Зорі С. А.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент доц. каф. КІ, к.т.н. Цололо С.О.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Нормоконтролер доц. каф. ПМіІ, к. т. н. Назарова І. А.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Покровськ - 2017

ДВНЗ «ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ»
Кафедра прикладної математики і інформатики

Освітньо-кваліфікаційний рівень – спеціаліст (бакалавр)
Напрям підготовки 050103 «Програмна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри прикладної
математики і інформатики
_____ О.А. Дмитрієва
«___» _____ 2017 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТА

Кривенко Олександра Миколайовича
(прізвище, ім'я, по батькові)

1. Тема роботи «Інструментальна підтримка розробки ігрових додатків» спецчастина «Розробка розважального ігрового додатку на базі Unity»

керівник роботи проф. каф. ПМіІ, д. т. н. Зорі С. А.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу № 136-08 від 05.05 2017 р.

2. Строк подання студентом роботи _____

3. Вихідні дані до роботи _____

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

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

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка

Студент _____ Кривенко О.М.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Зорі С.А.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Кваліфікаційна робота бакалавра за темою «Інструментальна підтримка розробки ігрових додатків» зі спецчастиною «Розробка розважального ігрового додатку на базі движка «Unity».

Пояснювальна записка кваліфікаційної роботи містить: 75 сторінок, 34 рисунки, 48 таблиць, 3 додатки, 21 використаних джерел.

Об’єктом дослідження роботи є процес розробки розважального ігрового додатку на базі движка «Unity». Предметом дослідження виступають методи, алгоритми та технології, застосовувані при розробці відеоігор на базі рушія, які забезпечать підвищення ефективності розробки відеоігор.

Мета кваліфікаційної роботи – дослідження сучасного інструментарію для розробки відеоігор, а також розробка концепції відеогри та її реалізація.

В роботі розглядається реалізація розважального ігрового додатку «Seeker» на базі проведеного аналізу сучасних засобів створення ігрових додатків. Для розробки використовувалися ігровий рушій Unity, засіб створення та редагування зображень Gimp, а також засіб для створення 3D моделей Autodesk 3ds Max.

Матеріали даної роботи можуть бути запозичені до робіт відповідних за жанровою характеристикою та ігровою механікою. А сама робота може бути використана як приклад для подальшого застосування відповідного інструментарію.

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, РОЗВАЖАЛЬНІ ІГРОВІ ДОДАТКИ, РУШІЙ UNITY, ІГРОВА СЦЕНА, СКРИПТИ, ЛОКАЦІЯ, ТЕКСТУРА, СПРАЙТ, 3DS MAX, GIMP, МОДЕЛЬ, АНІМАЦІЯ, ГРАВЕЦЬ, ТЕХНОЛОГІЯ, АЛГОРИТМИ НАВІГАЦІЇ І ПОШУКУ ШЛЯХІВ

ЗМІСТ

ВСТУП

Індустрія відеоігор бере свій початок у 1970х роках зі створення першої масової гри. До неї відносять все, що пов’язано з маркетингом, монетизацією і розробкою відеоігор. З моменту свого народження вона пройшла великий шлях. Згідно до маркетингових досліджень, сумарний світовий дохід індустрії відеоігор у 2016 році склав 100 мільярдів доларів США, а сумарна кількість геймерів – близько 2 мільйонів осіб [1]. В Україні було згенеровано 0.14 % від загального обсягу доходів, а саме 142 мільйони доларів США [1].

По мірі розвитку індустрії відеоігор зростають системні вимоги до обладнання, а також висуваються все нові і нові вимоги до графічних та сюжетних характеристик відеоігор. Завдяки цьому виробники комп’ютерного обладнання були вимушені випускати більш потужні компоненти і створювати технології, що поліпшують якість чи полегшують розробку. Не відстаючи від темпів вдосконалення апаратної складової, удосконалювався і розширявся інструментарій розробників. Завдяки цьому до інструментарію розробника входить величезне різноманіття доступних засобів та методів створення відеоігор. Одними з таких засобів є ігрові рушії (англ. Game engine).

Завдяки широкій доступності ігрових рушіїв та їхній відносній легкості їх розробки з’явилося таке поняття, як «інді ігри» (англ. Indie game). Воно позначає гру, що створена без фінансування проекту від видавців відеоігор, а також має невеликий розмір бюджету, чи не мають його зовсім [2]. Оскільки розробники інді ігор не потребують схвалення, що є обов’язковим для розробників масових ігор, вони можуть створювати інноваційні та креативні ігри [2]. Прикладом такої гри є «Minecraft», що була розроблена у 2009 році та продана у 2012 році компанії «Microsoft» за 2,5 мільярди доларів США [3].

У роботі було проведено розгляд сучасного інструментарію для розробки відеоігор, оцінка сучасних тенденцій створення відеоігор, досліджено етапи і процеси, пов’язані з розробкою відеоігор, організовано

ефективний і стабільний процес розробки гри, розроблено концепції відеогри.

Під час виконання поставлених завдань, буде проведена велика кількість роботи по дослідженню створення відеоігор, адже розробка відеоігор – це динамічний процес. У випадку розробки інді ігор слід приділити увагу креативній ідеї та якості кінцевого продукту. Розробка інді ігри не має на увазі постійну поточну розробку додаткового контенту, як цим займаються такі відомі студії як «Paradox Interactive» [4], що після випуску відеогри довгий час випускають пакет доповнень (англ. Downloadable content) для гри, продаж котрих і формує основну частину їхнього прибутку.

Розробка інді гри це тривала, важка праця. Але користь для індустрії відеоігор величезна. Розробка є генерацією нового, актуального, незалежного, що через відсутність фінансування являє собою щось виключне. Це формує найближче майбутнє індустрії відеоігор, адже світова практика свідчить про переймання дослідів і методів розробки професійними ігровими студіями.

Об’єктом дослідження роботи є процес розробки розважального ігрового додатку на базі движка «Unity». Предметом дослідження виступають методи, алгоритми та технології, застосовувані при розробці відеоігор на базі рушія, які забезпечать підвищення ефективності розробки відеоігор.

Мета кваліфікаційної роботи – дослідження сучасного інструментарію для розробки відеоігор, а також розробка концепції відеогри та її реалізація.

В роботі розглядається реалізація розважального ігрового додатку «Seeker» на базі проведеного аналізу сучасних засобів створення ігрових додатків. Для розробки використовувалися ігровий рушій Unity, засіб створення та редагування зображень Gimp, а також засіб для створення 3D моделей Autodesk 3ds Max.

Матеріали, розроблені в рамках, роботи можуть бути використані при розробці відеоігор відповідного жанру.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ Й ПОСТАНОВКА ЗАДАЧІ

1.1 Загальні тенденції у розробці відеоігор

Сучасні засоби розробки можуть бути вчасності представлені в вигляді середовища розробки або набору бібліотек, що дозволяє спростити та пришвидшити розробку відеогри. В той час методи розробки мають на увазі базові концепції, які використовуються для побудови додатку. Такий інструментарій прийнято називати ігровим рушієм.

Ігровий рушій може створюватись, як до окремої гри, так і для багатократного використання. Сучасні ігрові рушії створюються саме для багатократного використання, оскільки створювати для кожної нової гри новий ігровий рушій, який буде дублювати функціонал попередніх, не є оптимально [5]. Вони мають певний набір компонентів та готових модулів, що можуть бути використанні при створенні гри за певною тематикою.

Наслідуючи сучасні тенденції розробки програм, ігрові рушії мають у своїй основі багатошарову та модульну архітектору. Вона дозволяє полегшити процес створення і підтримки додатку, а також зменшити складність розробки окремого проекту завдяки повторному використанню модулів. [6]

Згідно з наявних модулів у рушії та їх можливостей, ігрові рушії розділяють на спеціалізовані та загальні. Різниця в тому, що спеціалізовані мають набір компонентів, заточений саме під створення ігор однакового плану, в той час як модулі загальних рушіїв пропонують більший набір функцій. Також спеціалізовані рушії майже завжди мають у собі декілька засобів для побудови штучного інтелекту, чи мають декілька варіантів реалізованого. Навіть при наявності таких наче очевидних плюсів спеціалізованих рушіїв перед загальними, другим віддається перевага, оскільки не потрібно освоювати новий рушій для наступних проектів.

При розробці певного ігрового додатку постає багато вимог. Деякі з них можуть бути задоволені за допомогою стандартних засобів та функцій рушію

або дописано розробниками гри самостійно. Більш оптимальним вибором є рушій, який задовольняє більшу частину необхідних для реалізації вимог. В залежності від розроблюваної гри ці вимоги мають певний характер.

До початку розробки необхідне чітке уявлення про розроблювану відеогру, яке повинно включати до себе класифікаційні відомості, ігрові ескізи, а також опис або сценарій гри. Це дозволяє уявити загальну картину розроблюваного продукту і визначити підходящий рушій для розробки.

Одним із важливих атрибутів ігрового рушія є мови програмування, які він підтримує. Часто сучасні ігрові рушії підтримують декілька мов розробки, що мають однакові або різні призначення. Рушії мають свої загальні тенденції та свої структурні особливості. Значна частина характеристик та можливостей ігрового рушію визначається компонентами, з яких він складається. Такими компонентами можуть бути програмні модулі або проміжне програмне забезпечення (англ. Middleware).

Важливими параметрами при виборі рушія є його документація та спільнота. Окрім цих параметрів важливим є тип розповсюдження ігрового додатку. До класифікаційних відомостей гри, що важливі при виборі ігрового рушію, можна віднести підтримувані платформи, графічну складову, кількість гравців, жанр гри, мету гри.

1.2 Визначення параметрів класифікації ігрових рушіїв

1.2.1 Мови програмування ігрових рушіїв

Сучасні ігрові рушії підтримують декілька мов програмування, які служать для різних цілей. Є основна мова програмування (англ. Primary language) та мова написання скриптів (англ. Scripting language), окрім цього деякі рушії мають мову для збереження даних (англ. Data storage language). Основна мова програмування представляє собою базову мову програмування, на якій передбачається розробка загальної гри. Мова написання скриптів призначена для швидкого створення ігрових скриптових сценаріїв, що описують послідовність у діяльності ігрових об’єктів. На відмінну від перших

двох мова збереження даних описує ігрові дані, таким чином, що їх можна прочитати. Розробник найчастіше працює з первиною та скриптовою мовами програмування.

1.2.2 Структурні особливості ігрових рушіїв

Сучасні ігрові рушії зазвичай мають приблизно однаковий набір компонентів [6], що включає до себе наступні частини.

Графічний рушій або рушій візуалізації (англ. Graphics engine) – це програмний компонент, основним завданням якого є візуалізація двомірної або тривимірної ігрової графіки.

Фізичний рушій (англ. Physics engine) – це програмний компонент, що виконує моделювання фізичних законів віртуального світу. Він моделює не самі закони фізики, а лише деякі фізичні системи, такі як динаміка твердого тіла (включно з визначенням зіткнень), динаміка м’якого тіла, динаміка рідини та тощо. Моделювання лише деяких систем дозволяє спростити моделювання для того, щоб воно виконувалось в режимі реального часу.

Звуковий рушій (англ. Sound/Audio engine) – програмний компонент, який відповідає за відтворення звуку (шумове та музичне оформлення, голосів персонажів) в комп'ютерній грі або іншому додатку. Крім цього звуковий рушій може здійснювати імітацію акустичних умов, відтворення звуку відповідно до місця розташування для створення ефекту глибини.

Систему скриптів (англ. Scripting language) – високорівнева мова сценаріїв (англ. Script) – коротких описів дій, виконуваних системою. Сценарій може описувати послідовність операцій, яку виконує певний ігровий об’єкт. Різниця між програмами і сценаріями розмита. Але вважаться, що сценарій це програма, що має справу з готовими програмними компонентами.

Анімаційний рушій (англ. Animation engine) – програмний компонент, який відповідає за відображення ігрової анімації. Під ігровою анімацією здебільшого розуміють анімацію, яка не оброблюються фізичним рушієм гри. Анімаційний рушій оброблює анімацію, яка створена за допомогою редактора

ігрових моделей.

Штучний інтелект (англ. Artificial intelligence) – програмний компонент, що представлений набором програмних методів, які використовуються в комп’ютерних іграх для створення ілюзії інтелекту в поведінці віртуальних персонажів. На відміно від традиційного в ігровому штучному інтелекті широко використовуються різного роду спрощення, обмани та емуляції.

Мережевий код (англ. Network code) – програмний компонент, що призначений для реалізації роботи з мережею. Може використовуватися, як для обміну даними з окремим сервером для збирання ігрової статистики, так і для налагодження мережевої гри.

Засоби ігрової оптимізації – програмні компоненти, які дозволяють програмісту оптимізувати продуктивність ігрового додатку. Засоби ігрової оптимізації здебільшого представлені засобами управління пам'яттю (англ. Memory management) і багатонитевісттю (англ. Multi-threading).

На додаток до програмних компонентів, ігрові рушії мають додаткові візуальні інструменти, які полегшують розробку. Ці інструменти зазвичай представлені в вигляді компонентів, інтегрованих до середи розробки. Завдяки ним можлива спрощена, швидка розробка ігор на манер поточного виробництва. В купі ці засоби створюють гнучку і багаторазово використовувану програмну платформу з усією необхідною функціональністю для розробки ігрової програми, скорочуючи витрати, складність і час розробки – всі критичні фактори в сильно конкурентній індустрії відеоігор.

Більшість ігрових рушіїв однакового класу мають однакові функціональні можливості, хоча трохи відрізняються за своєю структурою. Тому для аналізу повністю досліджувати структуру кожного ігрового рушію не є доцільним.

1.2.3 Тип розповсюдження ігрових рушіїв

Принципи розповсюдження ігрових рушіїв нічим не відрізаються від

принципів розповсюдження іншого програмного забезпечення. Основним документом, який визначає права і обов'язки користувача, є ліцензійна угода (англ. License agreement), що додається до програмного продукту або у вигляді паперового документа, або в електронному вигляді. Саме ця угода визначає правила використання даного екземпляра продукту. По суті, ліцензія виступає гарантією того, що видавець програмного забезпечення, якому належать виключні права на програму, не звернеться до суду на того, хто нею користується. Іншими словами, видавець програмного забезпечення ставить певні захисні рамки по використанню його програмного забезпечення.

В основному програми діляться на дві великі групи – вільного використання (безкоштовна і відкрита ліцензія) і невільного (комерційна ліцензія), а також між ними існують умовно-безкоштовні програми, які можна віднести до двох груп навпіл, такі програми можна завантажити і використовувати, але поки її не оплатити у вас можуть виникнути деякі проблеми або обмеження. Загальні типи ліцензій наведено на рис. 1.1.

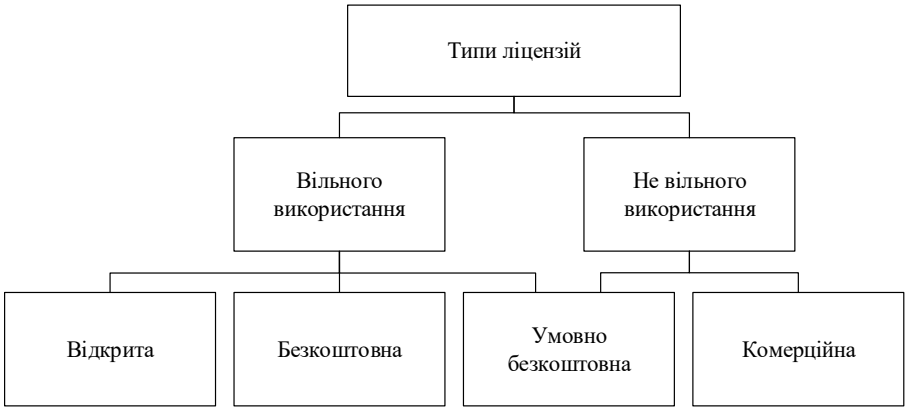


Рисунок 1.1 – Тип ліцензування програмного забезпечення

1.2.4 Наявність документації та спільноти

Наявність якісної документації і інструкцій, а також розвиненої спільноти розробників (користувачів ігрового движка) є ще одним із важливих факторів при виборі ігрового рушію.

Наявність гарної документації та великої кількості прикладів дозволяє

швидко освоїти та почати використовувати для себе новий інструмент, а також підвищувати свої професійні навики.

Одним з показників актуальності та необхідності інструменту є наявність розвиненої спільноти. Вона є допоміжним інструментом, який здатен допомогти розробнику в вивченні нового інструменту та у вирішенні проблем, які пов'язані з нюансами його використання.

1.2.5 Платформи відеоігор

Основним з основних видів поділу відеоігор на категорії є поділ за платформами (рис. 1.2). Вони вказують, на якому конкретному пристрої можна запустити гру. Це є найбільш важливим тому, що якщо у гравця немає відповідної платформи, то він не зможе пограти в гру, розраховану саме для цієї платформи.

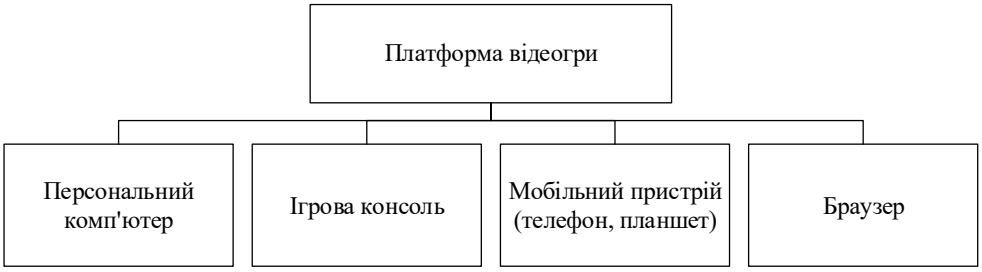


Рисунок 1.2 – Відеоігри за підтримуваними платформами

Існують ексклюзивні ігри та багато платформні ігри. Ексклюзивні ігри – це ігри, створені для якоїсь певної платформи, а багато платформні – розроблені відразу для декількох платформ.

Розробники намагаються охопити найбільшу аудиторію, тому випускають відразу гру на декілька платформ. Такий випуск є можливим, якщо ігровий рушій підтримує цю платформу. Але потрібно зазначити, що доступні методи керування ігровим світом у різних платформах відрізняються, тому у іграх, які розраховуються на багато платформну підтримку, потрібно передбачити різні методи керування ігровим оточенням. У випадку якщо ігровий рушій не підтримує визначену платформу, то для випуску ігри

потрібно створювати гру заново на ігровому рушії, що підтримує платформу.

Першою з платформ є персональні комп’ютери. Вони можуть бути складені з найрізноманітніших комплектуючих. Це дозволяє плавно оновлювати свій комп'ютер, замінюючи частину за частиною, не відстаючи від технічного прогресу. Персональні комп'ютери в свою чергу поділяються на кілька платформ за ОС. У кожній ОС розроблені свої інструменти для обробки відеоігор, тому не всі ігри можуть підійти відразу до всіх ОС. Найбільш популярними серіями ОС є MS Windows, Apple Mac OS і Linux.

Іншою ігровою платформою є вузькоспеціалізована ігрова модифікація персональних комп’ютерів, тобто консолі. Вузько спрямованість цих пристроїв робить їх простіше у використанні і ефективніше (внутрішня архітектура консолей дозволяє видавати набагато кращий результат, при однакових з комп'ютером технічних характеристиках).

На відміну від комп'ютерів, консолі є готовими нерозбірними пристроями (замінювати і оновлювати можна лише деякі зовнішні деталі). У зв'язку з цим, розвиток консолей є процес, розділений на чіткі етапи - покоління консолей. Це робить консольні ігри більш стандартизованими, ніж комп'ютерні, але, через це ж, вони постійно відстають від технічного прогресу. Також розробка під консулі вимагає більшої кількості затрат, а також сертифікацію від виробників консолей. Найбільш популярнішими ігровими консолями є Sony PlayStation, Microsoft Xbox, Nintendo 3DS, Wii.

Іншою популярною платформою є мобільні пристрої. Вони за технічними характеристиками набагато слабкіше стаціонарних комп'ютерів, тому мобільні ігри виглядають простіше і бідніше від звичайних ігор, але прогрес йде, і ситуація поступово поліпшується. Дуже часто на мобільні телефони переносяться старі ігри з персональних комп’ютерів.

Браузери – це різновид платформи для відеоігор, основною характеристикою якого є використання інтерфейсу веб-браузера. Такі ігри представлені різноманітними жанрами і, як правило, не вимагають встановлення іншого програмного забезпечення, окрім самого браузера та за

потреби відповідних плагінів для нього. Хоча відкривати інтернет сайти можна на багатьох платформах, але переважна більшість браузерних ігор підтримують лише декілька, або на деяких платформах не можливо грати, оскільки інтерфейс гри не розрахований на керування на цих платформах.

Хоча усі платформи одного плану достатньо різні з програмної точки зору, але вони мають багато суспільних засобів керування, що мало відрізняються друг від друга. А різність програмної складової абстрагується ігровим рушієм, але розробник повинен ураховувати різницю в доступних засобах керування.

1.2.6 Формат графіки відеогри

Головною прикрасою відеоігор є їх зовнішній вигляд. Існує поділ ігор за типом графічного зображення (рис. 1.3). Формат графіки описується типом графічного світу, а також ракурсом погляду на нього.

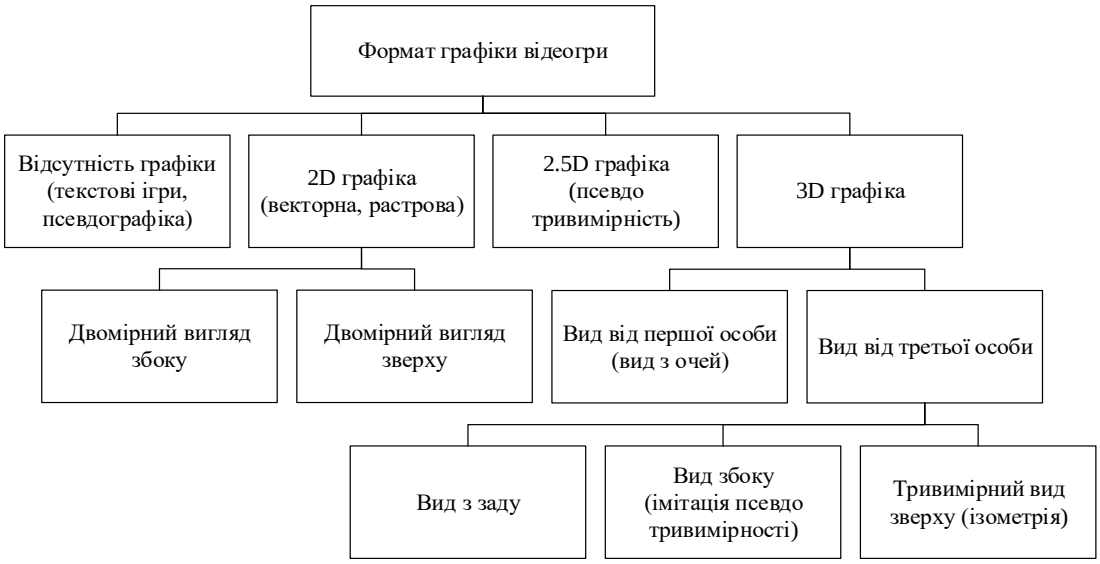


Рисунок 1.3 – Відеоігри за форматом графіки

Відсутність графіки характерна для перших поколінь комп'ютерів, оскільки мала потужність ставила перед розробниками масу обмежень. Через це у багатьох старих іграх застосовувалася псевдо графіка. Ігри такого виду більше схожі на інтерактивну книгу, а не на відеогру. Але і в наш час

зустрічаються подібні ігри.

2D графіка найбільш природний вид графіки для двомірних екранів моніторів. Існує декілька варіантів представлення об’єктів. Перший, векторний, де об'єкти складаються з геометричних координат, з'єднаних лініями. Другий, растровий, де об’єкти гри складаються з окремих пікселів - кольорових квадратів.

Один із варіантів ракурсу в 2D іграх – це вигляд збоку. Вид збоку дозволяє бачити всі перепади висот на рівні: ями, прірви, всі поверхи, платформи. Відсутність третього виміру значно спрощує сприйняття ігрового світу, в ньому легко орієнтуватися. Інший - двомірний вигляд зверху (англ. 2D Top Dawn). Вид згори відмінно підходить для того, щоб бачити розташування відразу багатьох ігрових об'єктів: персонажів, військ, техніки, наземних будівель. Ідеально підходить для ігор, в яких потрібно контролювати велику кількість об'єктів. Використовується в жанрах: стратегія, РПГ, тактика, головоломка, логічні ігри.

Завдяки застосуванню тригонометричних формул у розробників ігор з'явилася можливість створювати ілюзію тривимірного світу, що відображається на двомірної площини екрану. У комп'ютері обчислюються справжні 3D моделі, а на екран виводиться математично обчислювані 2D проекції цих тривимірних об'єктів.

Одним з варіантів відображення є вид від першої особи, при якому ми бачимо віртуальний світ очима головного героя. Такий вид найбільш зручний, щоб вживатися в роль віртуального героя. Іншим видом є вид від третьої особи. Вид від третьої особи може бути з заду, при якому ми бачимо віртуальний світ так, що головний герой виявляється перед нами в центрі екрану. Дозволяє краще оцінювати ситуацію, зручніше розглядати навколишнє оточення. Головний герой завжди на виду, тому його зовнішній вигляд і анімації повинні бути на вищому рівні. Також він може бути збоку (імітація псевдо тривимірності) та з гори (3D Top Dawn, ізометрія), що імітує відповідні вигляди 2D ігор.

1.2.7 Кількість гравців відеогри

За кількістю гравців (рис. 1.4) всі ігри можна умовно поділити на ігри без гравців, з одним гравцем та ігри з декількома гравцями. Багато ігор мають декілька ігрових режимів з різною кількістю гравців.

Рисунок 1.4 – Класифікація відеоігор за кількістю гравців

Гра без участі гравців представляє собою гру, де за всіх ігрових персонажів грає комп’ютер, а людина не бере участі в грі. Цей клас ігор деколи називають заставка на ігровому движку.

В одиночній грі ігровий процес розрахований на одного гравця. Всіма противниками та союзниками в такій грі керує комп'ютер.

Ігри з декількома гравцями можуть здійснюватися на одному пристрої, у локальній мережі, або онлайн. На одному пристрої можуть бути реалізовані за допомогою покрокової гри (англ. Hot seat) чи розділення екрану на декілька частин (англ. Split screen).

У випадку гри у локальній мережі чи групової онлайн гри об'єднуючою ланкою в мережі пристроїв стає один з гравців (англ. Hester) або спеціально призначений для цього комп'ютер (сервер). Кожен гравець входить в гру через свій пристрій.

В масових онлайн іграх може брати участь величезна кількість людей

(десятки і сотні тисяч гравців). Такі онлайн ігри побудовані за принципом «клієнт-сервер». В такому випадку гравці повинні мати аккаунт на центральному ігровому сервері і проходять авторизацію через нього. За обробку ігрового процесу можуть відповідати, як спеціальний потужний сервер, чи роль такого серверу може виконувати комп’ютер одного з гравців.

1.2.8 Жанри відеоігор

Жанри відеоігор на відміну від класичних жанрів фільмів та книг описують інші особливості. Класичні жанри описують сюжет. Ігрові жанри визначаються за ігровою механікою. В той час ігровий сюжет прийнято назвати сетингом (англ. Setting).

Ігрова механіка – це набір ігрових дій, які робить гравець під час проходження гри. На зорі створення ігрової індустрії розробники ігор проводили сміливі експерименти, створюючи ігрові механіки інтуїтивно. Невдалі експерименти були забуті, а вдалі ігри стали прикладом для інших. Інші розробники копіювали популярну ігрову механіку, додаючи трохи ідей від себе, таким чином навколо найбільш популярних ігор утворювалися цілі класи схожих між собою ігор, ці класи стали називати ігровими жанрами.

Сьогодні більшість знає популярні жанри відеоігор і їх характерні риси. Але повної і чітко структурованої класифікації жанрів комп'ютерних ігор немає до сих пір. Класифікація за жанром є самою спірною та складною, про що свідчать спори стосовно жанрової приналежності деяких ігор.

На перший погляд, з ігровими жанрами все дуже заплутано і неоднозначно. Багато хто вважає, що чітка класифікація ігор в принципі неможлива. Оскільки, комп'ютерні ігри - твори мистецтва, вони не знають кордонів (насправді, жанри виступають саме в ролі кордонів), дуже часто буває, що гра містить ознаки майже всіх жанрів, і в таких пропорціях, що дійсно незрозуміло, який же жанр для неї є основним. Але самі по собі жанри - структури прості і їх можна виділити.

Для формування класифікації було виділено 15 елементарних жанрів

відеоігор. Які були умовно розділено на наступні категорії: ігри спілкування або ігри інформації, ігри дій, ігри контролю. Після чого було відділено більшість складових жанрів та деякі гібридні жанри.

Ігри спілкування мають на увазі під собою, що гравець під час гри отримує інформацію, здійснює спілкування та вивчення світу. При тому для даної категорії ігор досить характерним є те, що отримані раніше відомості знадобляться для проходження сюжетної лінії далі.

Ігри дії характеризуються переміщенням у просторі, використанням техніки, зброї та інших внутрішньо ігрових засобів для досягнення кінцевої мети. Вони не вимагають від гравця специфічних навичок для проходження.

Ігри контролю вимагають від гравця навички командуванням, управлінням і розподілом матеріальних благ. Класичним для даного жанру є управління не одним ігровим персонажем, а декількома.

Після того, як було створено розгорнуту класифікацію відеоігор (рис. 1.5), з неї було виділено короткий набір жанрів відеоігор (рис. 1.6), який достатньо точно підходить для класифікації ігрових рушіїв.

1.2.9 Мета відеоігор

Класифікація відеоігор за метою (рис. 1.7) найбільш проста з наведених. Ця класифікація виникла внаслідок обширного значення терміну гра. Вона призначена для виділення основної мети проходження гри.

В грі на проходження ігрові задачі взаємозв’язані друг з другом і слідують одна за одною, що утворює лінію сюжету. Ігри даного типу характерні тим, що після проходження сюжетної лінії до них зникає інтерес.

Навчальні ігри зазвичай представлені іграми для дітей або вузькоспеціалізованими симуляторами для дорослих. В процесі гри в ігровій формі подається інформація для вивчення.

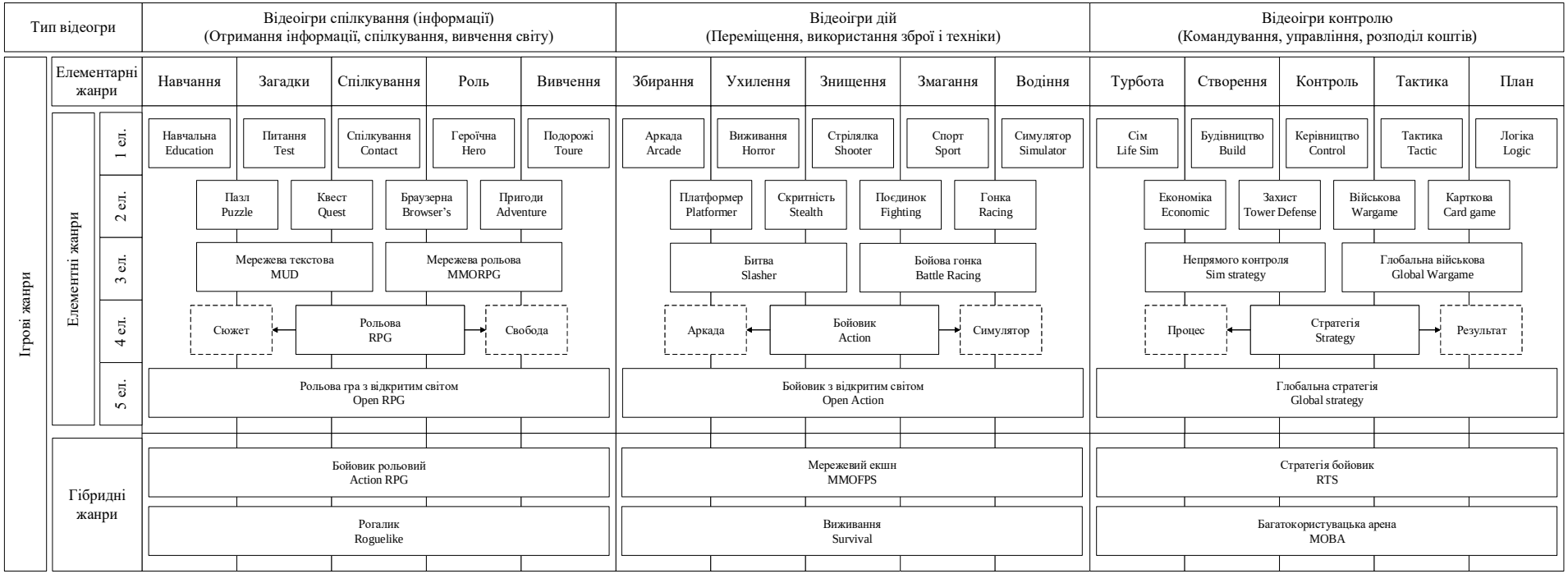


Рисунок 1.5 – Розгорнута класифікація відеоігор за жанром

Рисунок 1.6 – Коротка класифікація відеоігор за жанром

Рисунок 1.7 – Класифікація відеоігор за метою

Казуальні ігри вимагають мінімум часу на освоєння ігрової механіки і зазвичай просто та інтуїтивно зрозумілі. Гра побудована так, що її можна тимчасово перервати в будь-який момент, а потім продовжити. Часто процес гри розділений на невеликі рівні.

Гра-пісочниця – це гра без сюжету і цілей. Основою гри-пісочниці є різноманітні ігрові можливості, які гравець може застосовувати на власний розсуд. Досить часто пісочниці, це не окремі ігри, а спеціальні режими в сюжетних іграх.

У грі-змаганні гравці змагаються між собою за статус переможця. У багатьох іграх є можливість змагатися як з гравцями, так і з комп'ютерними противниками. Якісні гри-змагання дуже довговічні – в них можуть грати через десятиліття після дати виходу гри, що не природно для інших відеоігор.

Хардкорні ігри створені спеціально для досвідчених гравців, для випробування їх ігрових навичок. Відмітна особливість: гра розділена на невеликі рівні, в кожному рівні підраховується витрачений час або кількість зароблених очок. Гравці раз по раз проходять одні й ті ж відрізки гри, щоб отримати найкращий результат - рекорд. В таких іграх на першому місці стоїть ігровий процес і хитромудрий дизайн рівнів.

1.3 Аналіз і вибір ігрового рушію

На сьогодні створено більш ніж 150 повно функціональних ігрових рушіїв, які доступні для розробників [7]. Що викликає труднощі при виборі ігрового рушія для реалізації певного проекту. Оцінити ігрові рушії можливо

за їхніми характеристиками, а також за деякими суб’єктивними параметрами, які мають певну вагу для розробки [8].

Із великої кількості рушіїв для первинного аналізу було обрано рушії керуючись наступними критеріями:

– рушій повинен бути безкоштовним або умовно безкоштовним;

– рушій повинен підтримувати багато платформну розробку;

– рушій повинен активно розвиватися та на ньому повинні створюватися нові ігрові додатки.

Таким чином, було встановлено, що з проаналізованих ігрових рушіїв ігрові рушії Unity, Unreal і CryEngine надають найбільшу функціональність (таблиця 1.1). Розглянув рушії більш детально, було визначено, що найбільш оптимальним вибором для створення інді-ігор на сьогоднішній день є Unity [9] (таблиця 1.2).

Таблиця 1.1 – Аналіз загальних характеристик ігрових рушіїв

Назва рушію	Мови програмування	Характеристика за можливостями розробки	Характеристика за додатковими параметрами ігрового рушію
Unity	Первині: C++, C# Скриптові: Mono, Unity Script (версія JavaScript), C#, Boo, Cg, HLSL	Є змога перенесення більш ніж на 21 платформу. Підтримує 2D та 3D графіку різних типів. Має засоби для створення багатокористувацьких ігор. Має засоби реалізації ігрового інтелекту для багатьох жанрів.	Ліцензія: безкоштовна Unity Personal при річному заробітку меншому за $100,000. Документація: вирішує більшість виниклих питань. Спільнота: більшість учасників не професійні розробники Unity, але активно вирішуються проблеми
Unreal	Первині: C++ Скриптові: GLSL, Cg, HLSL, UnrealScript (версія JavaScript), C++, Blueprints	Є змога перенесення більш ніж на 11 платформ. Підтримує 2D та 3D графіку різних типів. Має засоби для створення багатокористувацьких ігор. Має велику кількість засобів реалізації ігрового інтелекту для багатьох жанрів. Також має в собі інтегровані засоби для створення ігрових моделей.	Ліцензія: безкоштовна публічна версія для некомерційних проектів, якщо проект має прибуток, то виплачується 5% роялті. Документація: достатньо повна, але не вирішує деяких питань. Спільнота: більшість учасників мають гарний опит у розробці ігор, але не дуже активно вирішуються проблеми
CryEngine	Первині: C++, C# Скриптові: Lua, C#	Є змога перенесення на 7 платформ. Підтримує 2D та 3D графіку різних типів. Присутній розширений набір засобів для створення багатокористувацьких ігор. Має засоби реалізації ігрового інтелекту для деяких жанрів.	Ліцензія: повністю безкоштовна без виплат роялті, відкритий вихідний код. Документація: є не розкриті питання. Спільнота: не дуже активна, на деякі питання можна не отримати відповідь.

Продовження таблиці 1.1

Panda3D	Первині: C++, Python Скриптові: Python	Є змога перенесення на 3 платформи. Підтримує 2D та 3D графіку. Не можливе створення багатокористувацьких ігор за допомогою бібліотек ігрового рушія, потребує сторонні бібліотеки. Відсутні засоби реалізації ігрового інтелекту.	Ліцензія: безкоштовна. Документація: детально не описані важливі нюанси. Спільнота: активна, деякі питання залишаються без відповіді.
Urho3D	Первині: C++ Скриптові: –	Є змога перенесення на 4 платформи. Підтримує 2D та 3D графіку. Не можливе створення багатокористувацьких ігор за допомогою бібліотек ігрового рушія, потребує стороні бібліотеки. Відсутні засоби реалізації ігрового інтелекту.	Ліцензія: безкоштовна. Документація: описує тільки загальні положення. Спільнота: офіційної спільноти не має
Wave	Первині: C#; VB; F# Скриптові: C#; VB; F#	Є змога перенесення на 3 платформи. Підтримує 2D та 3D графіку. Не можливе створення багатокористувацьких ігор за допомогою бібліотек ігрового рушія, потребує стороні бібліотеки. Відсутні засоби реалізації ігрового інтелекту.	Ліцензія: безкоштовна, але для повноцінної роботи потребує придбання інших продуктів. Документація: описує тільки загальні положення і деякі заміжні аспекти. Спільнота: офіційної спільноти не має

Таблиця 1.2 – Аналіз характеристик ігрових рушіїв Unity, Unreal і CryEngine

Назва рушію	Позитивні сторони	Негативні сторони
Unity	Дуже популярний та має гарну ліцензійну політику. Потужний та інтуїтивно зрозумілий редактор сцен. Багато навчальних ресурсів і має найкращу спільноту серед розглянутих. Дозволяє швидке прототипування. Має продуману система анімації. Дуже оптимізований для мобільних пристроїв. Найменший розмір файлів компіляції. Гнучка система створення компонентів	Закритий вихідний код. Специфічний опит розробки порівняно з іншими двигунами, а також присутні погані практики розробки. Слабке керування пам'яттю і відсутні налаштування збірника сміття. Обмежені можливості створення 3D моделей; Документація має суперечливу інформацію з приводу скриптових мов. Відносно погана графіка порівняно з іншими.
Unreal	Відкритий вихідний код. Найбільш якісне середовище розробки зі зручними та необхідними візуальними інструментами. Потужна система редагування матеріалів. Інноваційна система освітлення. Присутня технологія швидкої компіляції з використанням раніше компільованих фрагментів. Велика кількість навчальних ресурсів. Досвідчені учасники спільноти.	Великий розмір вихідних файлів та повна компіляція займає багато часу. Ліцензійна політика заснована на роялті. Повільне середовище розробки порівняно з іншими двигунами. Проблеми зі швидкістю на слабких пристроях пов'язані з недосконалими засобами пакетування графічних викликів і обмеженою апаратною підтримкою. Більшість документації автоматично генерується і має не працездатні приклади.
CryEngine	Повністю безкоштовний та має відкритий код. Найбільш інноваційний рушій з точки зору графічної складової. Краща імітація погодних умов. Можливо представляти сценарії у вигляді графіків потоків. Забороняє погані практики у створенні моделей.	Високий поріг входження, оскільки для програмування потрібні специфічні навички. Важко розроблювати ігри з видом не від першої персони. Малий набір можливостей 3D моделювання.

1.4 Аналіз підходів для створення програмної архітектури ігор

1.4.1 Компонентна архітектура і спадкування

Великі ігри мають складну архітектуру, сутності та взаємодію між класами. При стандартному ООП підході при розробці ігор необхідні постійні переробки купи коду і сильне збільшення тривалості розробки, тому що використовується спадкування [10]. Тобто змінити реалізацію типу-предку неможливо, не порушивши коректність функціонування типів-нащадків [10]. Для вирішення цієї проблеми був сформований компонентне-орієнтований підхід (КОП): «Є певний клас-контейнер, а також клас-компонент, який можна додати в клас-контейнер. Об'єкт складається з контейнера і компонентів в цьому контейнері.» В ООП підході об'єкт визначається описуваним його класом, а в КОП підході – компонентами, з яких він складається.

КОП спрощує повторне використання написаного коду: використання одного компонента в різних об'єктах; новий тип об'єкта отримують з різних комбінацій вже існуючих компонентів [11]. Треба зазначити, що КОП підхід для розробки ігрових додатків є стандартом для більшості ігрових рушіїв.

Використання даного підходу робить можливим створення складних ієрархії класів ігрових персонажів, предметів та іншого [12]. Але, багато розробників не використовують КОП при проектуванні складних систем класів для ігрових персонажів, предметів, або правильно виділяють компоненти, але роблять спадкування компонентів штучного інтелекту. Проблемні місцем при не використанні даного методу виражається у тому, що спадковий тип повинен володіти властивостями, поведінкою різних типів, не вирішена проблема постійних змін усіх класів в даній ієрархії. При використані КОП об'єкти збираються з компонентів і вже немає проблеми, що при зміні одного об'єкта потрібно змінювати інші.

1.4.2 Робота з об’єктами зі складною поведінкою

Для спрощення роботи з об'єктами зі складною поведінкою можуть бути використанні бути використані машини станів та дерева поводжень [13].

У машинах станів логіка об'єкту розбивається на стани, події, переходи, а також може розбиватися ще й на дії. Варіації реалізації цих елементів можуть помітно відрізнятися. Стан об'єкта може бути як класом без ігрової логіки, просто зберігає якісь дані, наприклад, назву стану об'єкта: атака, переміщення. Або клас «стан» може описувати поведінку об'єкта в конкретному стані.

Машини станів тісно зв’язані з трьома ключовими поняттями. Перший, дія – це функція, яка може здійснитися в даному стані. Другий, перехід – зв'язок між 2-ма станами. Вказує, з якого стану в який можливий перехід. Третій, подія – якесь повідомлення/команда, що передається в машину станів або викликається всередині неї. Служить для вказівки, що треба виконати перехід в інший стан, якщо це можливо з поточного стану.

Робота з графом машини станів ускладняється, коли станів багато, збільшується кількість зав’язків між ними. Для спрощення роботи з ним можна використовувати ієрархічну машину станів, в якої в якості стану можна використовувати вкладену машину станів. Таким чином виходить деревоподібна ієрархія станів.

Іншим розповсюдженим методом є дерева поведінки. Вони представляють собою деревовидну структуру, в якості вузлів якої виступають невеликі блоки ігрової логіки. З різних блоків логіки розробник конструює в візуальному редакторі деревоподібну структуру, налаштовує вузли дерева. Ця структура буде відповідати за прийняття рішень персонажем та його взаємодію з ігровим світом. Кожен вузол повертає результат, від якого залежить, як будуть оброблятися інші вузли дерева [14]. Варіанти результату, що повертаються зазвичай наступні: успіх, невдача, виконується.

Виділяють декілька основних типів вузлів у деревах поведінки. Вузол дії (англ. Action Node) – просто деяка функція, яка повинна здійснитися при відвідуванні даного вузла. Вузол умови (англ. Condition Node) – зазвичай служить для того, щоб визначити, виконувати чи ні наступні за ним вузли. При true поверне Success, а при false повертає Fail. Вузол послідовності (англ. Sequencer Node) – виконує всі вкладені вузли по порядку, поки який-небудь з

них не завершиться невдачею (в такому випадку повертає Fail), або поки всі вони успішно завершаються (тоді повертає Success). Вузол селектору (англ. Selector) – на відміну від Sequencer, припиняє обробку, як тільки будь вкладений вузол поверне Success. Вузол інтегратору (англ. Iterator Node) – виконує роль циклу for), використовується для виконання в циклі серії дій деяке число раз. Паралельні вузли (англ Parallel Node) відрізняються тим, що виконують всі свої дочірні вузли «одночасно». Тут не мається на увазі, що вузли виконуються декількома потоками. Просто створюється ілюзія паралельного виконання, аналогічно корутинам в Unity3d.

Дерева поведінки складніше реалізувати, якщо їм потрібно реагувати на події ззовні. Для вирішення проблеми можна ввести вузли-завдання (наприклад: патрулювання, переслідування супротивника, атака супротивника) в дерево поведінки. Тобто передбачається, що контролер дерева поведінки буде переходити на інший вузол-завдання, щоб змінити поведінку. Іншим варіантом вирішення проблеми є об'єднання дерева поведінки з машиною станів.

Таким чином, якщо штучний інтелект сам отримує дані зі світу і не отримує жодних команд, то дерево поведінки підходить для нього. Якщо ж штучний інтелект чимось управляється людиною або іншим штучним інтелектом (наприклад, ігровий персонаж в стратегіях управляється гравцем-комп'ютером або загоном), то краще використовувати машину станів.

1.4.3 Абстракції ігрових об'єктів

Гравець (може бути як комп'ютер, так і людина; в одному класі змішувати їх не варто) – це окремий об'єкт. Ігровий персонаж– також окремий об'єкт, яким може керувати будь-який гравець. В стратегічних іграх до того ж можна окремо винести об'єкт «загін» [15]. Розділити ігрову логіку на подібні об'єкти не складно. До того ж вона буде застосована до безлічі різних ігор.

1.4.4 Доступ до компонентів об'єктів і сцен

При великій кількості компонентів в об'єкті, з'являється незручність при потребі звернення до них [16]. Постійно доводиться заводити в кожному компоненті поля для зберігання посилань на інші компоненти або звертатися до них через GetComponent(), що є представленням патерну медіатору.

Патерн медіатор наштовхнув ввести якийсь компонент-посередник, через який компоненти могли б звертатися один до одного. До того ж це дозволить винести перевірку існування інших компонентів в даний компонент та код знадобитися писати тільки 1 раз. Такий компонент у різних типів об'єктів теж варто робити різним, оскільки використовуються різні набори компонентів. В даному випадку – це не реалізація патерну медіатор, а просто збереження посилань в одному класі для зручності доступу до інших компонентів об'єкта.

У сценах аналогічна ситуація [17]. Можна в об'єкті зробити посилання на часто використовувані компоненти, щоб в інспекторі конкретних компонентів не доводилося постійно вказувати посилання на інші об'єкти. Так як компоненти потрібно вказати тільки в одному об'єкті, а не в декілька, трохи зменшується обсяг роботи у редакторі, так і коду в цілому буде менше.

1.4.5 Складні складові ігрових об'єктів

Персонажі, елементи інтерфейсу і деякі інші об'єкти можуть складатися з більшого числа компонентів-скриптів і безлічі вкладених об'єктів. Якщо погано продумана ієрархія подібних об'єктів, то це може сильно ускладнити розробку [18]. Важливо продумувати ієрархію об'єкта коли окремі частини об'єкта повинні замінюватися іншими в процесі гри або частина скриптів об'єкта повинна працювати тільки в одній сцені, а інша частина – в інший.

У першому випадку можна повністю заміняти об'єкт, але якщо він після заміни повинен знаходитися в тому ж стані і мати ті ж дані, що і до заміни, то завдання ускладняються. Тобто для спрощення заміни будь-якої частини об'єкта, його розбивають на складові. Подібне розділення дозволить змінити

зовнішній вигляд об'єкта або контролер анімації, не сильно впливаючи на інші компоненти [19].

У другому випадку, наприклад, є якийсь об'єкт зі спрайтом і даними. При клікі на нього в сцені поліпшень, потрібно покращити даний об'єкт. А при натисканні на нього в сцені гри повинно виконатися якась ігрова дія. Можна зробити 2 префаба, але тоді, якщо об'єктів багато, доведеться налаштовувати в 2 рази більше префабів.

1.4.6 Модифікатори (баффи/дебаффи)

Характеристики персонажа/предмета/здібності можуть змінюватися з-за впливу яких-небудь накладених ефектів і ефектів одягнених предметів [20]. Сутність, яка змінює ці характеристики, у даному контексті називають модифікатором. Модифікатор – певний компонент, в якому перераховані характеристики, на які він впливає, і величини впливу. Можливо буде навіть краще, якщо модифікатор буде впливати тільки на одну зазначену характеристику. Коли на персонажа накладається ефект, до нього додається компонент-модифікатор. Далі модифікатор викликає функцію «застосувати себе до такого об'єкту», і виконується перерахунок характеристик об'єкта. Причому лише тих характеристик, які він зачіпає. При видаленні модифікатора – аналогічно виконується перерахунок.

Перерахунок потрібен, щоб не доводилося при кожному зверненні до даних персонажа постійно обчислювати актуальні значення. Тобто звичайне керування для збільшення продуктивності.

1.4.7 Серіалізація даних

При розробці ігор все ще дуже часто використовують XML, хоча є альтернативи, найчастіше більш зручні – JSON, SQLite, зберігання даних в префабах. Звісно, вибір залежить від завдань.

При використанні XML або JSON, багато використовують досить неоптимальні способи роботи з точки зору великою кількістю коду для роботи

з структурами даних форматах, та ще й з необхідністю вказувати у строковому вигляді назви імен елементів, до яких потрібно звертатися.

Замість усього цього можна використовувати серіалізацію. Структура XML, JSON в цьому випадку буде генеруватися з коду (Code First підхід). Для серіалізації XML Unity3D можна використовувати вбудовані .NET кошти, а для JSON можна використовувати плагін JsonFx.

1.5 Формальна постановка задачі розробки

Завдання роботи полягає в дослідженні інструментарію для розробки відео ігор. Тобто завдання роботи полягає в розробці розважальної відеогри, яка реалізовувала базові ігрові механіки та за допомогою якої користувач зміг би відпочити або скоротати свій час. Програма повинна дозволяти користувачеві використовувати всі необхідні операції. Інтерфейс повинен бути простим, зрозумілим і зручним для будь-якого користувача. Гра не повинена мати зайві елементи, які нагромаджують ігровий інтерфейс.

Основна гра («Seeker») повинна складатися з основної ігрової сцени та двох міні ігор.

Основна сцена гри має відповідати наступним критеріям:

а) представлена у вигляді 3D будівлі з декількома кількістю кімнатами;

б) у кімнатах будівлі повинні знаходитися два комп’ютери, при взаємодії гравця з котрими мають починатись міні ігри «Flappy Bat» та «Tower Defense»;

в) гравець повинен мати можливість вільно пересуватись по будівлі з урахуванням реальних законів фізики.

У основній грі має відбуватися взаємодія гравця з двома незалежними один від одного об’єктами (ПК) та випробування двох ігор. Обидва комп’ютера мають знаходитись на одній локації.

Геймплей на цій локації має виглядати наступим чином:

а) гравець має мати вид від третьої особи;

б) маніпулювання персонажем має відбуватися за допомогою кліку мишки (гравець кликнув мишкою – персонаж пересунувся у координати на

локації, на котрі було зроблено клік);

в) якщо персонаж підійшов до одного з двох об’єктів, то має з’явиться можливість взаємодіяти з ними за допомогою клавіши дії;

г) при взаємодії персонажа з об’єктом має початись одна з двох ігор;

д) після взаємодії з об’єктом та гри персонаж має мати можливість взаємодіяти з іншим об’єктом та грати у іншу гру, або продовжити взаємодію з поточним об’єктом та грати у обрану раніше гру знов;

е) дизайн локації має бути лінійним у тому сенсі, що кордони локації (будівлі) мають бути зроблені у вигляді стін.

Міні-гра «Flappy Bat» має являти собою типову 2D відеогру, шо є складовим елементом основної гри, у котрій гравець має мати можливість обрати комп’ютер з однією з ігор («Flappy Bat»), мета якої полягає у «вбиванні» часу через управління польотом кажана у печері, в якої не має освітлення.

Механіка гри «Flappy Bat» має полягати у тому, що кажан починає політ у темряві печери, під час польоту він випромінює хвилі, завдяки яким орієнтується в просторі, при зустрічі з перешкодами кажан впаде та гра закінчиться.

Отже, ігровий процес має мати наступний вигляд:

а) кажан планує у печері під час старту гри;

б) клацанням мишки гравець змушує кажана набирати висоту, щоб кажан не зіткнувся з перепоною у виді сталактиту, сталагміту чи землею;

в) у разі зіткнення кажана з перепоною гра закінчується та гравцю повідомляють кількість набраних ним балів під час польоту кажана;

г) бали нараховуються поточно, з рівними проміжками часу, а максимальна кількість балів майже не обмежена.

Друга гра «Tower defense» – це гра, основна мета якої – не дати рухомим об’єктам дійти до кінця мапи за допомогою башт, які мають наносити шкоду цим рухомим об’єктам.

Гра має мати мапу (поле для гри), башти, точки, на яких мають бути

побудовані башти, рухомі об’єкти, заздалегідь прокладений маршрут руху об’єктів, кінцеву точку руху об’єктів, при досягненні котрої, гравець програє.

Вежі мають будуватись гравцем у конкретних точках, але не більше однієї на точці, за умовою, щоб гравець розосередив башти так, щоб вони не стояли на шляху у рухомих об'єктів, тобто не заважали їм рухатись своїм шляхом до кінця мапи. Об’єкти мають рухатись з фіксованою швидкістю, мати фіксовану кількість очок міцності, фіксований розмір та колір, щоб не вводити гравця в оману. Башти мають мати фіксовані розміри, фіксовані графічні данні, фіксовані показники атаки по рухомим об’єктам та відстань нанесення шкоди по об’єкту. Якщо один конкретний рухомий об’єкт під час свого пересування під вогнем башти отримує фіксовану кількість шкоди, то він має бути знищеним вогнем башти. Якщо бодай один рухомий об’єкт пересунеться через увесь свій шлях, не отримавши потрібної кількості шкоди від діяльності башт, то він перетинає мапу та гра закінчується. У такому сценарії гравець програє, тобто єдина мета гри – не дати рухомим об’єктам дійти до кінця, знищівши їх усіх за допомогою башт.

Рухомі об’єкти мають рухатись по заздалегідь фіксованому маршруту, який нагадуватиме простий лабіринт, біля котрого мають бути побудовані вежі, котрі будуть атакувати рухомі об’єкти.

Таким чином, гра має виконувати такі дії:

а) початок гри (старт);

б) будівництво башт у конкретних точках на мапі;

в) рух об’єктів, які мають бути знищені баштами гравця;

г) якщо об’єкти не були знищені, то гравець програє;

д) якщо об’єкти знищені, то гравець перемагає;

е) повторити пункт 1-5, за умов, що пункт 4 не виконується.

2 ПРОЕКТУВАННЯ РОЗВАЖАЛЬНОГО ІГРОВОГО ДОДАТКУ «SEEKER»

2.1 Функціонування програмної системи

Розроблена програма призначена для використання на персональному комп’ютері для гравців різного віку та соціального стану. Вона повинна надавати свій функціонал у повному обсязі. Список виконуваних програмою функцій можна визначити за допомогою діаграми прецедентів. Загальна діаграма прецедентів (рис. 2.1) включає до себе трьох акторів.

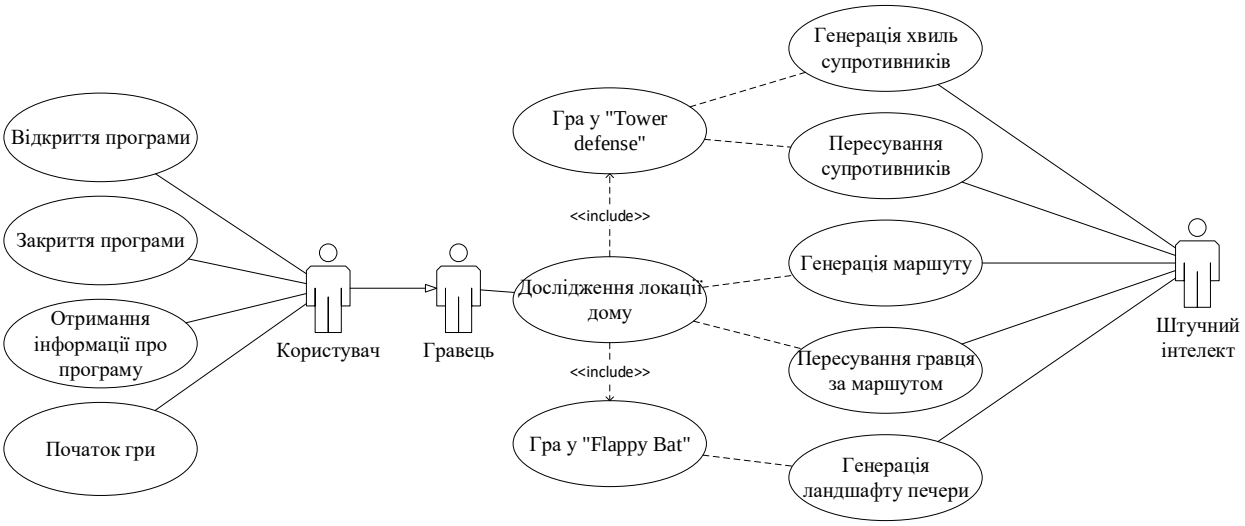


Рисунок 2.1 – Діаграма прецедентів системи

Загальна діаграма станів ігрового процесу з початку гри наведена на рис. 2.2. При старті програми відкривається головне меню, у якому можна почати гру, відобразити інформацію про гру та завершити роботу з грою. Якщо користувач обирає відображення інформації про гру, то перед користувачем відобразиться інформація про автора та короткий опис гри, який зникне через певний час. Якщо користувач обрав почати гру, відкриється основна гра, де користувач зможе вільно переміщатися та досліджувати ігровий світ. Під час дослідження ігрового світу користувач може обрати дію, яка приведе до відкриття під ігор «Flappy Bat» та «Tower Defense».

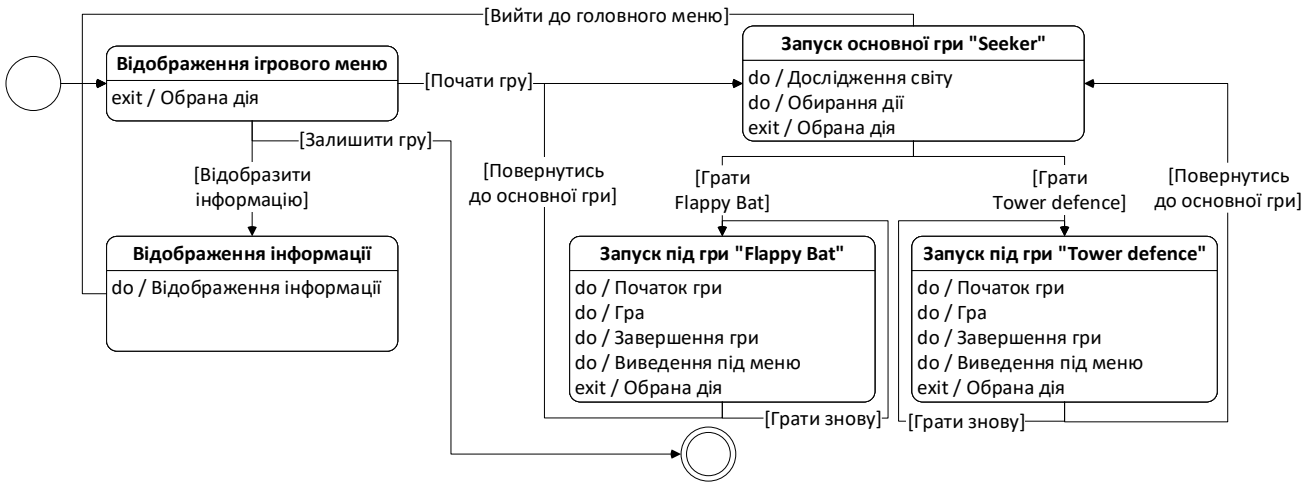


Рисунок 2.2 – Діаграма станів системи

2.2 Визначення структури системи

Програмна система окрім меню і базової ігрової локації повинна мати ще дві міні гри. Тому, реалізація ігрового додатку доцільна за використанням декількох моделей, які прийнято називати сценами.

Таблиця 2.1 – Відповідність частин гри до сцен

Назва сцени	Що знаходиться на сцені
Menu	Головного меню гри, що запускається зразу після запуску гри
Info	Інформація з відомостями про програму та автора
Home	Основна сцена гри, з якої запускаються міні ігри
Cave	Сцена міні гри «Flappy Bat»
Tower	Сцена міні гри «Tower Defense»

Сцени мають зв'язок між собою. Тому структуру програмної системи доцільно визначити за допомогою графу переходів (рис. 2.3) між ігровими сценами. Також для зручності звернення пронумеруємо вершини графу та переходи та зазначимо коли вони відбуваються.

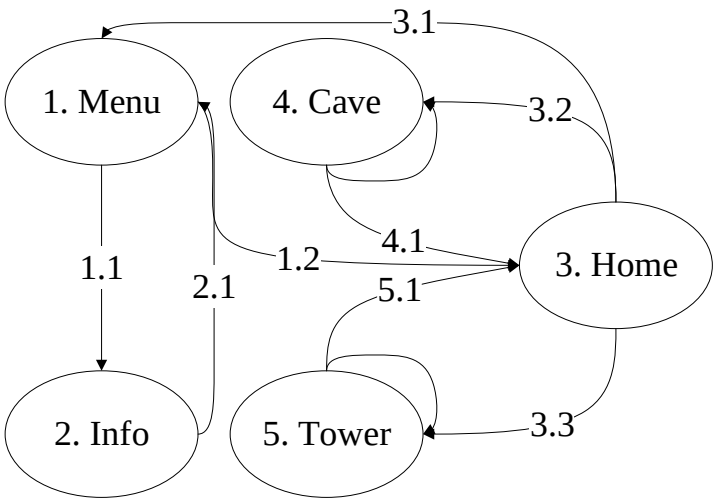


Рисунок 2.3 – Граф переходів між ігровими сценами

З сцени меню можливі переходи за двома напрямками в середині гри, а також можливо завершити роботу з програмою. Напрямок 1.1 активується відповідною кнопкою головного меню. Після переходу за цим напрямком через певний час відбувається автоматичний перехід до головного меню за напрямком 2.1.

Якщо користувач обрав грати, то перейде за напрямком 1.2. Звідки він зможе повернутися до головного меню за напрямком 3.1. Перехід за яким активується за допомогою натискання відповідної кнопки. Також користувач зможе продовжити досліджувати ігрову сцену, а також може перейти за допомогою кнопки активності до сцен з міні іграми за напрямками 3.2 та 3.3. Після закінчення гри можливо перезапустити відповідну міні гру або повернутися до головної сцени гри за напрямками 4.1 та 5.1 відповідно.

Така структурна організація гри на сцені зобов’язує до певної файлової структури проекту. В даному рішенні використовувалась зображена на рис. 2.4 структура активів проекту (англ. Project Assets).

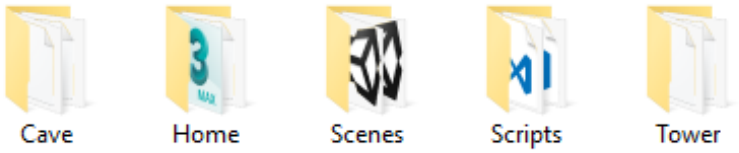


Рисунок 2.4 – Структура активів проекту

В теках Cave, Home та Tower знаходяться ігрові матеріали відповідних частин гри. Ці теки мають у собі під теки, що відповідають певним типам активів проекту. У теці «Scripts» знаходяться скрипти, які не відносяться до ігрових матеріалів вказаних раніше сцен. У теці «Scenes» зберігаються усі ігрові сцени.

2.3 Структура програмного коду системи

Із міркувань не розумного ускладнення структури проекту та різноплановості розроблюваних ігрових механік наслідування не використовувалось. Також варто зазначити, що для реалізації ігрових сценаріїв у невеликих проектах та у складних ігрових механіках не бажано використання наслідування. В таких випадках краще організовувати системи згідно компонентного підходу. Така організація системи наддасть більш гнучкий підхід до подальшої підтримки та доробки системи.

Доцільним є визначення взаємодії між компонентами системи за допомогою діаграм послідовностей для сцен, що реалізують ігри. Але слід зазначити, що дана діаграма не демонструє реальну специфіку роботи системи, вона демонструє лише загальні тенденції поведінки під модулів системи. Це зв’язано з тим, що дана діаграма не призвана враховувати усі вчасні випадки поведінки системи, а демонструє лише її загальні тенденції.

Діаграма активності головної сцени (рис. 2.5) і діаграми активності міні ігор «Flappy Bat» (рис. 2.6) та «Tower Defense» (рис. 2.7) мають свої особливості. Ці особливості виражені в специфіки кожної окремої гри. Для першої гри ця специфіка виражена тим, що дана гра має ігрову механіку, характерну для відкритого світу (можливо переміщатися за мапою та обирати будь-яку активність). А міні ігри мають певну обумовлену послідовність дій для проходження гри.

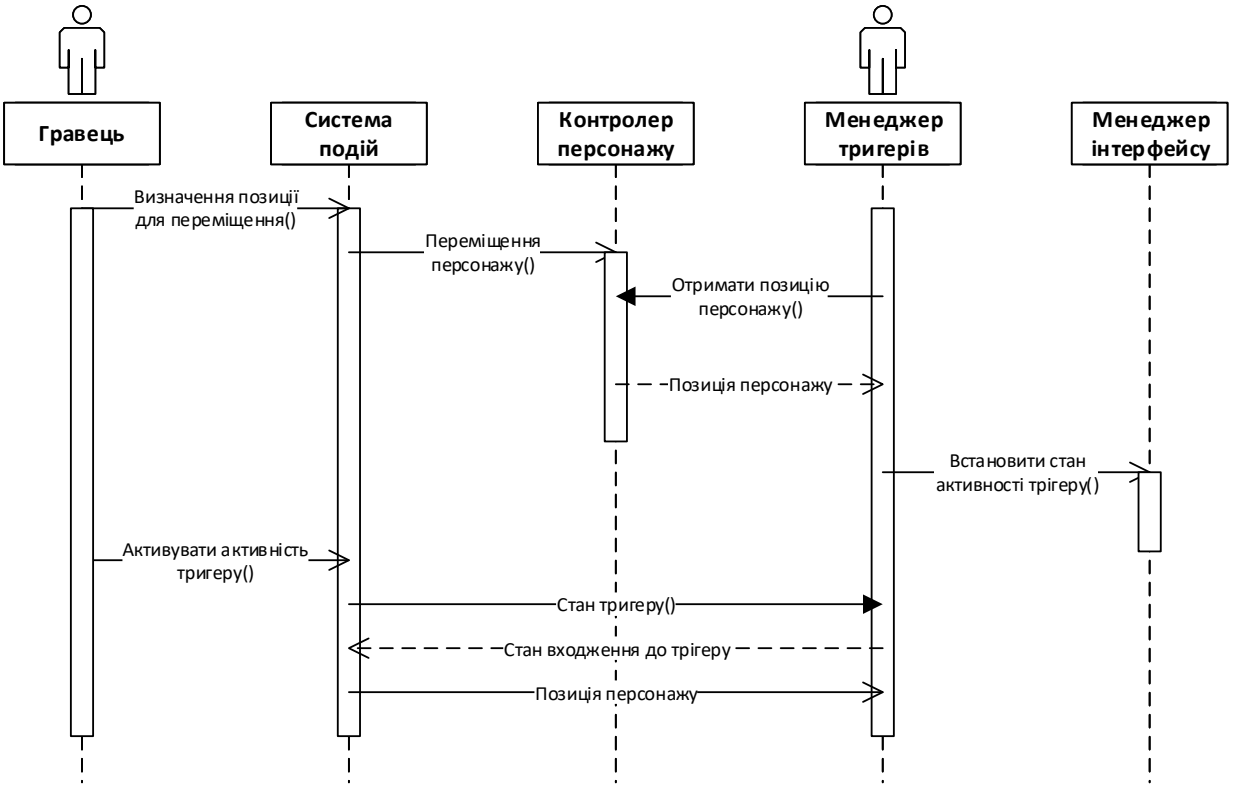


Рисунок 2.5 – Діаграма активності для головної сцени

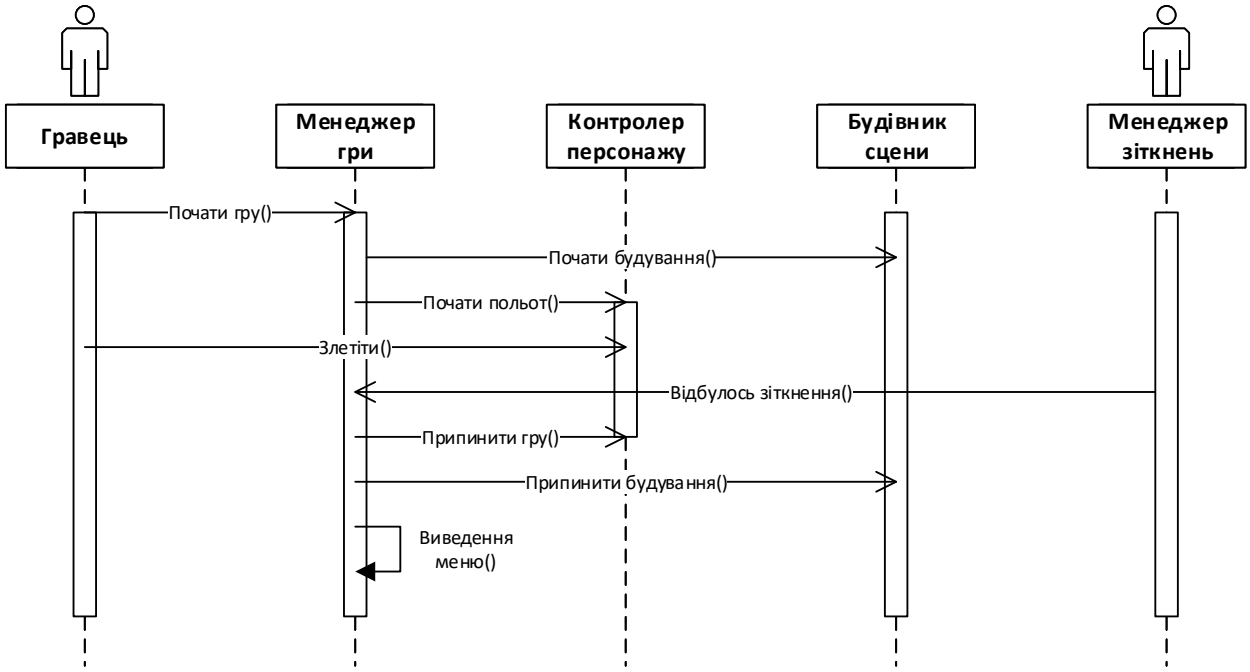


Рисунок 2.6 – Діаграма активності для сцени з грою «Flappy Bat»

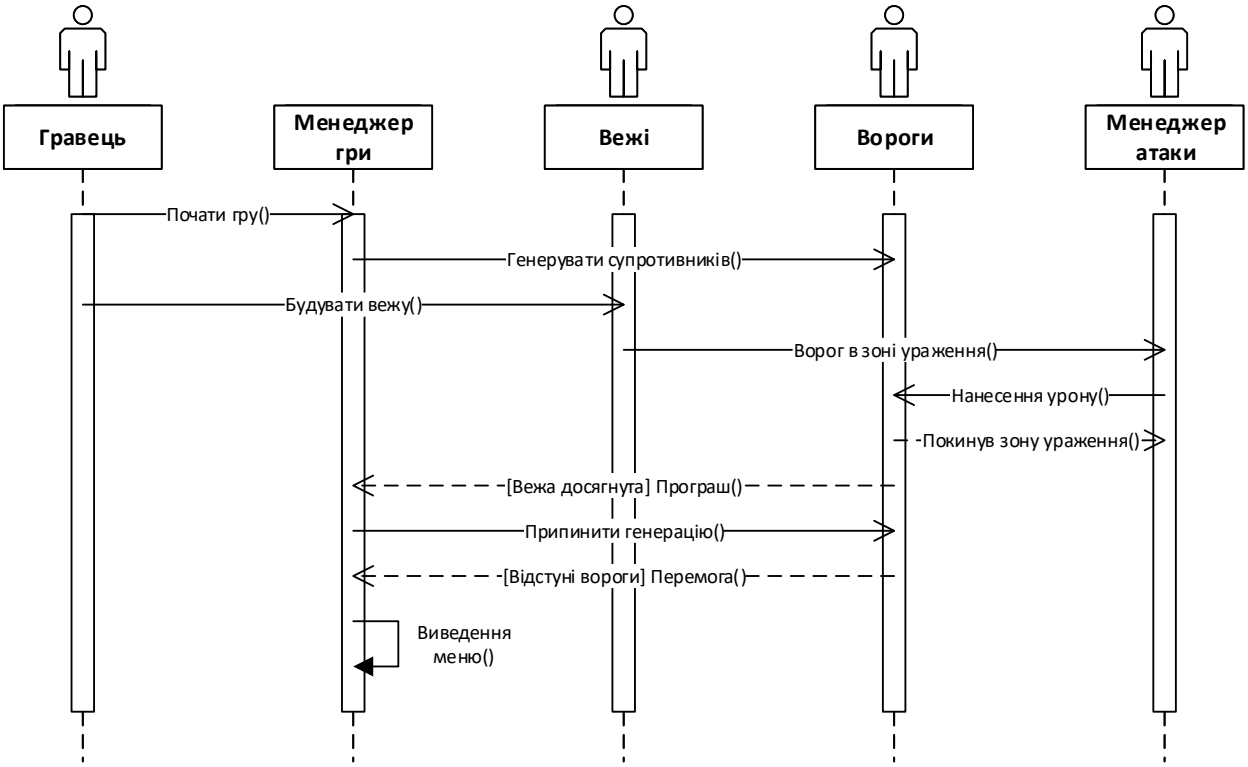


Рисунок 2.7 – Діаграма активності для сцени з грою «Tower Defense»

2.4 Розміщення компонентів програми

Гра реалізується як звичайний програмний додаток, тому компоненти відеогри повинні розміщуватися на комп’ютері користувача. Уся специфіка розміщення компонентів задана середою розробки та наведена на рисунку 2.8.

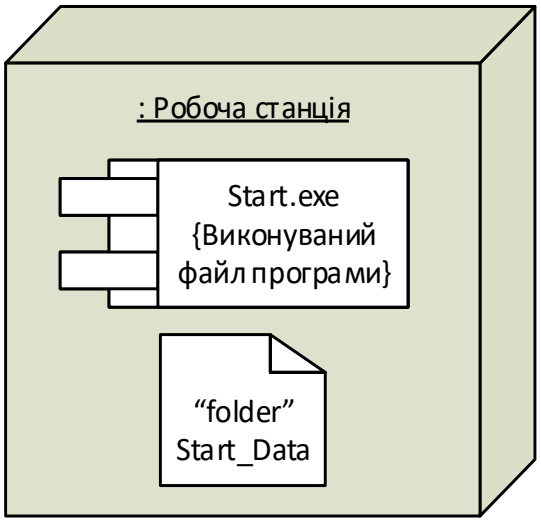


Рисунок 2.8 – Діаграма розміщення

Файл Start.exe відповідає за запуск гри. При запуску файлу відкривається вікно, у якому можна налаштувати параметри гри та безпосередньо запустити саму гру. Папка «Start_Data» зберігає файли гри та необхідні бібліотеки. Зміст цієї папки автоматично генерується середою розробки «Unity». Але потрібно зазначити, що такий формат вихідних файлів характерний тільки для операційної системи «MS Windows». При компіляції для платформ «MacOS», «iOS», «Linux», «Android» створюються лише файл програмного пакету, який включає до себе усі необхідні матеріали для роботи програми.

3 РОЗРОБКА РОЗВАЖАЛЬНОГО ІГРОВОГО ДОДАТКУ «SEEKER»

3.1 Структура ігрових сцен відеогри

Структура ігрових сцен на першому рівні вкладеності є достатньо простою і включає до себе деякі базові елементи, які є у всіх сценах. До таких елементів можна віднести Menu Transition, який відповідає за графічну складову переходу між сценами. Елемент Main Camera, який представляє головну камеру відповідної сцени, елемент Canvas, який слугує для виведення елементів інтерфейсу на екрані користувача. Також базовим є Audio Source, що відповідає за програвання музики під час знаходження на сцені.

Базова структура ігрової сцени «Menu» (рис. 3.1) окрім стандартних компонентів включає також компонент бекграунду, на який направлена камера, і він створює задній фон «Menu» та елемент «Menu Controller», який відповідає за підключення скрипту контролера ігрового меню.

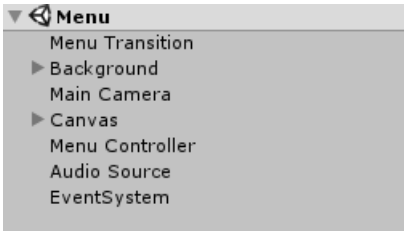


Рисунок 3.1 – Структура сцени «Menu»

У корені структури ігрової сцени «Info» (рис. 3.2), окрім стандартних компонентів, включає до себе «Info Manager», який відповідає за виведення інформації та повернення до ігрового меню через заданий час. Окрім цього елементу на сцені присутня система часток під назвою «Rain». Вона для виведення дощу на сцені.

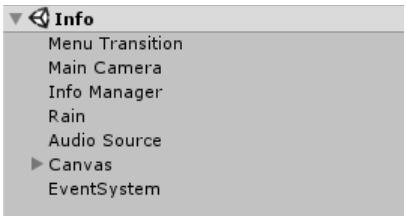


Рисунок 3.2 – Структура сцени «Info»

Базовий рівень структури ігрової сцени «Home» (рис. 3.3), окрім стандартних компонентів, включає до себе «Home», який включає до себе елементи сцени, а також елементи «Player» та «Target Picker», які зберігають модель гравця і точку, в яку повинен переміститися гравець.

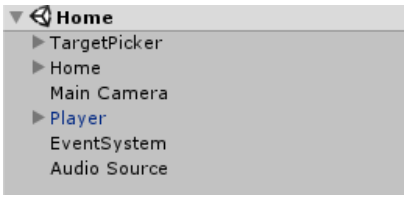


Рисунок 3.3 – Структура сцени «Home»

Ігрова сцена «Cave» у корені (рис. 3.4), окрім стандартних елементів, включає до себе 6 інших елементів. Елемент «Quad» відповідає за виведення заднього фону сцени, елемент «Bat» є безпосередньо нашим кажаном, яким відбувається гра. Елемент «Cave Limits» визначає фактичні границі верху та низу печери, в той час як елемент «Rocks» відповідає за графічну складову гори та низу печери. Елементи «Shadows» та «White Black» відповідають за затемнення ігрової сцени.

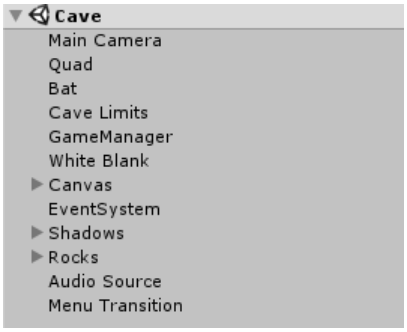


Рисунок 3.4 – Структура сцени «Cave»

Ігрова сцена «Tower» (рис. 3.5) має на першому рівні лише один нестандартний елемент, який включає в себе усі елементи ігрової сцени.

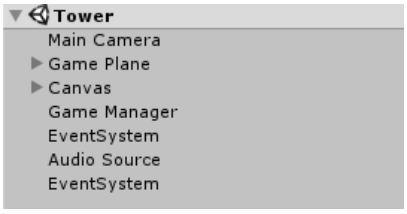


Рисунок 3.5 – Структура сцени «Tower»

3.2 Реалізація навігації і пошук шляху

Існує велике різноманіття алгоритмів пошуку шляху. Але найбільш популярними в розробці ігрових додатків стали алгоритми «Waypoints» та «NavMesh». Ці методи можуть існувати як окремі методи і можуть об’єднуватися для створення штучного інтелекту персонажів.

3.2.1 Реалізація «Waypoints» навігації

Навігація за допомогою «Waypoints» представляє собою звичайне переміщення за прямою між певними заздалегідь зазначеними точками. Ігрову механіку на базі «Waypoints» було реалізовано в ігровій сцені «Tower» (рис. 3.6). Вона необхідна для переміщення цілей від точки створення до точки виходу. Точкою створення є місце створення (англ. «Respawn»), а точкою виходу є наша башта, у яку не повинні потрапити цілі. Тобто, «Waypoints» є не регулярними та вручну встановленими вузлами, які слугують маяками для переміщення.

«Waypoints» навігація має ряд недоліків, які не суттєві для ігор цього типу. До таких недоліків можна віднести те, що будь-яке змінення на маршруті між точками вимагає його доробки. Також до недоліків можна віднести необхідність проставлення великої кількості точок переміщення для ігор з складною структурою світу.

Логіка переміщення «Waypoints» в «Unity» реалізується за допомогою

оператору трансформації і циклічного проходження за точками.

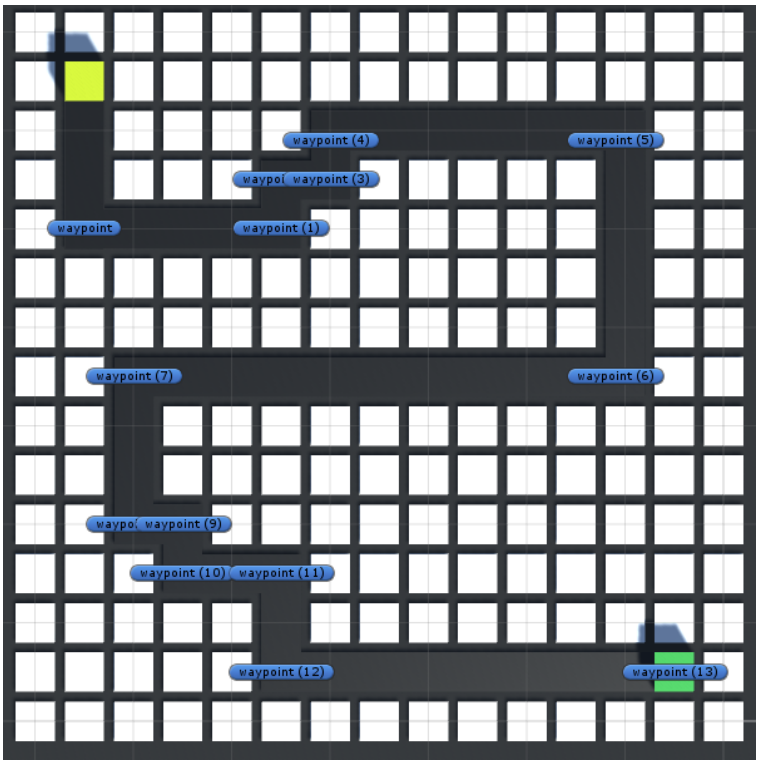


Рисунок 3.6 – Приклад розміщення точок «Waypoints» на маршруті

3.2.2 Алгоритм пошуку шляху «NavMesh»

«NavMesh» є сучасною технологією пошуку шляху, а точніше, спосіб представлення даних про шляхи на ігровій карті. «NavMesh» є полігональною поверхнею, яка покриває ті частини ігрового ландшафту, де можуть пройти наші персонажі. Полігони «NavMesh» будуються таким чином, щоб вийшли опуклі примітиви. Опуклість примітиву дає нам гарантію того, що всі об'єкти, що знаходяться всередині цього багатокутника доступні нам без жодних перешкод. Це одне з основних властивостей «NavMesh», і всі алгоритми пошуку шляху через «NavMesh» активно використовують цей постулат.

Завдяки своїй природі, алгоритми, побудовані на базі «NavMesh», є дуже швидкими. Це обумовлено відсутністю необхідності застосовувати трасування променю (англ. RayCast) для пошуку перешкод, а також мінімальна кількість вузлів в графі шляхів в порівнянні з іншими методами подання карти (наприклад, «PathNodes»), особливо на великій карті.

До переваг даного алгоритму відноситься зниження витрати пам'яті на зберігання графа шляхів, більш природне переміщення по карті, на відміну від незграбних переходів інших алгоритмів.

Ігрову механіку на базі технології «NavMesh» реалізовано на сцені «Home». Для цього було згенеровано навігаційну сітку будівлі (рис. 3.7) за допомогою вкладки навігація, прикріплено до персонажа вбудований до «Unity» скрипт «NavMeshAgent», а також створено скрипт отримання кінцевої точки переміщення від користувача.

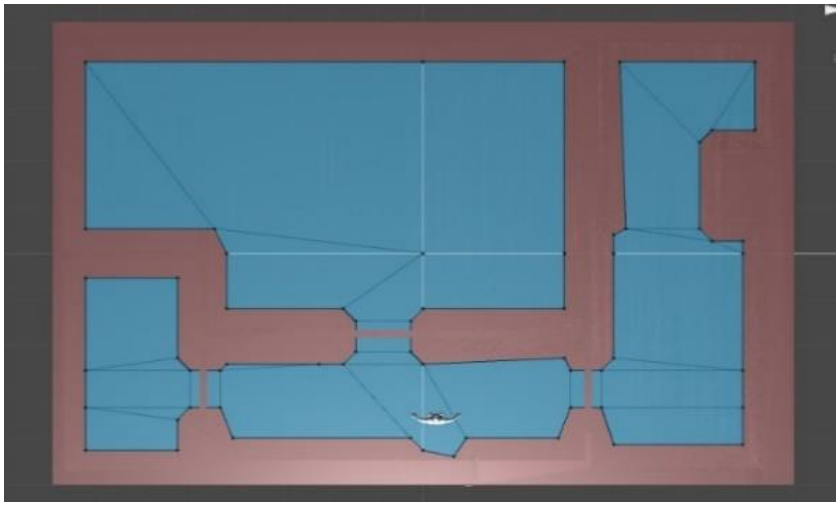
Б.1. 

Рисунок 3.7 – Приклад генерованої сітки «NavMesh»

3.3 Опис структури розроблених скриптів

3.3.1 Загальні скрипти

В рамках даної роботи було розроблено один загальний скрипт «Transition» для здійснення плавного переходу між сценами. Скрипт включає до себе 11 властивостей, представлених як полями, так і параметрами, які наведені в таблиці 3.1, а його методи наведено в таблиці 3.2.

Таблиця 3.1 – Властивості скрипту «Transition»

Тип	Назва	Призначення
Bool	StartDark	Обмін місцями кольорів переходу
Float	TimerMultiplyer	Множник таймеру переходу
Bool	TransitionFinished	Флаг завершення переходу
Bool	TransitionStarte	Флаг початку переходу
Bool	StartTransitioningImmediatel	Флаг негайного переходу
Float	MinAlpha	Мінімальна прозорість переходу
Float	MaxAlpha	Максимальна прозорість переходу
SpriteRenderer	SpriteRenderer	Визначення відобразника сцени
Float	Timer	Таймер здійснення переходу
Color	StartColor	Початковий колір переходу
Color	EndColor	Кінцевий колір переходу

Таблиця 3.2 – Методи скрипту «Transition»

Тип	Назва	Призначення
Void	Start	Підготовка до здійснення переходу
Void	Update	Здійснення переходу
Void	Reset	Скидання переходу з його повторенням
Void	StartTransition	Почати здійснення переходу

3.3.2 Скрипти сцени «Menu»

Для реалізації функціонування меню було розроблено один скрипт без параметрів «MenuController», що включає до себе активності, доступні в меню. Скрипт включає до себе 3 властивості, які наведені в таблиці 3.3.

Таблиця 3.3 – Методи скрипту «MenuController»

Тип	Назва	Призначення
Void	OnButtonPlayGameDown	Почати гру
Void	OnButtonShowInfoDown	Демонстрація інформації
Void	OnButtonExitGameDown	Покинути гру

3.3.3 Скрипти сцени «Info»

Для реалізації виведення інформаційної заставки було розроблено скрипт «InfoSplash», який відповідає за відображення сцени «Info». Скрипт включає до себе 5 властивостей, які наведені в таблиці 3.4, а його методи наведено у таблиці 3.5.

Таблиця 3.4 – Властивості скрипту «InfoSplash»

Тип	Назва	Призначення
Float	Timer	Таймер спалаху
Float	TimeOut	Тривалість спалаху
Transition	transition	Контролер переміщення
Bool	transitionBegun	Переміщення почалось
Float	transitionTimeOut	Час переміщення

Таблиця 3.5 – Методи скрипту «InfoSplash»

Тип	Назва	Призначення
Void	Start	Підготовка сцени
Void	Update	Програвання сцени

3.3.4 Скрипти сцени «Home»

Для реалізації ігрової механіки сцени «Home» було реалізовано скрипти «PlaceTargetWithMouse» і «ActionTrigger». Скрипт «PlaceTargetWithMouse» для вказання позиції, яку повинен набути персонаж. Скрипт включає до себе 2 властивості, які наведені в таблиці 3.6, а його методи наведено в таблиці 3.6. Скрипт «ActionTrigger» – для визначення взаємодії гравця з ігровим світом. Скрипт включає до себе 3 властивості, які наведені в таблиці 3.8, а його методи наведено в таблиці 3.9.

Таблиця 3.6 – Властивості скрипту «PlaceTargetWithMouse»

Тип	Назва	Призначення
Float	surfaceOffset	Зсув площини
GameObject	setTargetOn	Вказівник переміщення

Таблиця 3.7 – Методи скрипту «PlaceTargetWithMouse»

Тип	Назва	Призначення
Void	Update	Встановлення нової позиції вказівника переміщення

Таблиця 3.8 – Властивості тригеру «ActionTrigger»

Тип	Назва	Призначення
GameObject	TriggerMesh	Область дії тригеру
GameObject	ActionHelper	Допоміжник інтерфейсу
String	SceneName	Назва сцени для переходу

Таблиця 3.9 – Методи тригеру «ActionTrigger»

Тип	Назва	Призначення
Void	OnTriggerEnter	Користувач зайшов до тригеру
Void	OnTriggerExit	Користувач вийшов з тригеру
Void	OnTriggerStay	Взаємодія користувача з тригером

3.3.5 Скрипти сцени «Cave»

Скрипти «BatController» та «BatWave» реалізують логіку роботи з кажаном. Скрипт «BatController» – для керування кажаном. Скрипт включає до себе 9 властивостей, які наведені в таблиці 3.10, а його методи наведено у таблиці 3.11. Скрипт «BatWave» це контролер хвилі. Скрипт включає до себе 5 властивостей, які наведені в таблиці 3.12, а його методи наведено у таблиці 3.13.

Таблиця 3.10 – Властивості скрипту «BatController»

Тип	Назва	Призначення
CaveGameManager	gameManager	Менеджер ігрової сцени
AudioSource	flapSound	Звук змаху крил кажана
GameObject	wavePrefab	Каркас звукової хвилі кажана
Rigidbody2D	body2D	Фізичне тіло кажана
Vector2	waveOffset	Зсув початкової точки хвилі
Float	screamingPeriod	Період випуску хвилі
Float	screamingTimer	Таймер випуску хвилі
GameObject	shadow	Об’єкт тіні печери
SpriteRenderer	ShadowRender	Відображувач тіней

Таблиця 3.11 – Методи скрипту «BatController»

Тип	Назва	Призначення
Void	Start	Отримання значень параметрів
Void	Update	Здійснювати політ
Void	OnCollisionEnter2D	Відстежувати зіткнення
Void	Rise	Підняти кажана
Void	Scream	Випустити звукову хвилю
Void	GenerateWave	Генерувати звукову хвилю
Void	FixedUpdate	Анімація кажана не в грі

Таблиця 3.12 – Властивості скрипту «BatWave»

Тип	Назва	Призначення
SpriteRenderer	WaveRenderer	Відображувач хвиль
Rigidbody2D	body2D	Фізичне тіло хвилі
Float	Speed	Швидкість руху хвилі
Float	Scale	Швидкість масштабування хвилі
Float	Alpha	Прозорість хвилі

Таблиця 3.13 – Методи скрипту «BatWave»

Тип	Назва	Призначення
Void	Start	Отримання значень параметрів
Void	Update	Поширення хвилі

Скрипти для роботи з текстурами є найбільшим класом скриптів у даній сцені. Серед цих скриптів можна виділити один скрипт для обробки фону сцени та чотири скрипти, які відповідають за перепони у сцені.

Скрипт «CaveBackground» призначений для керування фоном сцени. Скрипт включає до себе 3 властивості, які наведені в таблиці 3.14, а його методи наведено в таблиці 3.15. Інші скрипти текстур відповідають за роботу з камінням у печері.

Таблиця 3.14 – Властивості скрипту «CaveBackground»

Тип	Назва	Призначення
Renderer	rend	Відображувач фону
Float	offset	Зсув текстури фону
Float	speed	Швидкість зсуву

Таблиця 3.15 – Методи скрипту «CaveBackground»

Тип	Назва	Призначення
Void	Start	Отримання значень параметрів
Void	Update	Зсування текстури фону

Скрипт «CaveRock» – для описання каменів у печері. Скрипт включає до себе 3 властивості, які наведені в таблиці 3.16, а його методи наведено у таблиці 3.17. Скрипт «CaveRockOutline» – для описання границь каменів у печері. Скрипт включає до себе 3 властивості, які наведені в таблиці 3.18, а його методи наведено в таблиці 3.19.

Скрипт «CaveRocksGen» – для генерації каменів у печері. Скрипт включає до себе 5 властивостей, які наведені в таблиці 3.20, а його методи

наведено в таблиці 3.21. Скрипт «CaveShadow» призначений для відображення тіні каменю. Скрипт має 2 властивості, які наведені в таблиці 3.22, а його методи наведено в таблиці 3.23.

Таблиця 3.16 – Властивості скрипту «CaveRock»

Тип	Назва	Призначення
Float	speed	Швидкість створення об’єктів
Rigidbody2D	body2D	Фізичне тіло каменю
CaveRockOutline	outline	Зовнішня границя каменю

Таблиця 3.17 – Методи скрипту «CaveRock»

Тип	Назва	Призначення
Void	Start	Ініціалізація створення об’єктів
Void	OnTriggerEnter2D	Підсвічення кордонів каменю

Таблиця 3.18 – Властивості скрипту «CaveRockOutline»

Тип	Назва	Призначення
SpriteRenderer	rend	Відображувач кордонів каменю
Color	defaultColor	Базовий колір каменів
CaveRockOutline	alpha	Прозорість каменю

Таблиця 3.19 – Методи скрипту «CaveRockOutline»

Тип	Назва	Призначення
Void	Start	Формування кордонів
Void	Update	Формування відтінку кольору
Void	Glow	Підсвітити камінь

Таблиця 3.20 – Властивості скрипту «CaveRocksGen»

Тип	Назва	Призначення
1	2	3
GameObject[]	prefabsList	Добірка каменів

Продовження таблиці 3.20

1	2	3
CaveGameManager	gameManager	Менеджер ігрової сцени
Float	generationPeriod	Період генерації каменів
Float	generationTimer	Таймер генерації каменів
Bool	canGenerate	Камінь згенеровано

Таблиця 3.21 – Методи скрипту «CaveRocksGen»

Тип	Назва	Призначення
Void	Start	Підготовка до генерації каменів
Void	Update	Контроль процесу генерації
Void	CreatePrefab	Генерація каменів
Void	StopGenerating	Припинення генерації каменів

Таблиця 3.22 – Властивості скрипту «CaveShadow»

Тип	Назва	Призначення
SpriteRenderer	srender	Відображувач тіні каменю
Float	alpha	Прозорість тіні каменю

Таблиця 3.23 – Методи скрипту «CaveShadow»

Тип	Назва	Призначення
Void	Start	Підготовка до відображення
Void	Update	Відображення тіні

Скрипт «CaveGameManager» призначений для управління ігровим процесом міні гри «Tower Defense». Скрипт включає до себе 14 властивостей, які наведені в таблиці 3.24, а його методи наведено у таблиці 3.25.

Таблиця 3.24 – Властивості скрипту «CaveGameManager»

Тип	Назва	Призначення
GameObject	blank	Фон ігрового меню
GameObject	gameoverPanel	Панель ігрового меню
SpriteRenderer	blankRenderer	Відображувач фону ігрового меню
Float	alpha	Прозорість фону ігрового меню
CaveRocksGen	rocksGen	Генератор скал печери
Bool	gameIsOver	Гра закінчена
Bool	gameStarted	Гра почата
Text	infoText	Інформаційний текст у початку гри
Int	score	Результат гри у числовому форматі
Int	bestScore	Кращій результат гри у числовому форматі
Float	scoreTimer	Проміжний результат гри у числовому форматі
Text	scoreText;	Проміжний результат гри у текстовому форматі
Text	finalScoreText	Результат гри у текстовому форматі
Text	bestText;	Кращій результат гри у текстовому форматі

Таблиця 3.25 – Методи скрипту «CaveGameManager»

Тип	Назва	Призначення
Void	Start	Підготовка ігрової сцени
Void	Update	Оновлення ігрової сцени
Void	OnButtonRetryDown	Повторний запуск міні гри
Void	OnButtonHomeDow	Перехід до сцени Home
Void	ShowGameOver	Показати результат гри
Void	GameOver	Сформувати результат гри
Void	StartGame	Почати гру

3.3.6 Скрипти сцени «Tower»

Скрипт «Enemy» – для опису мішені. Скрипт включає до себе 7 властивостей, які наведені в таблиці 3.26, а його методи наведено в таблиці 3.27. Скрипт «EnemySpawner» – для генерації мішеней. Скрипт включає до себе 5 властивостей, які наведені в таблиці 3.28, а його методи наведено в таблиці 3.29.

Скрипт «EnemyWave» – для опису хвиль породження мішеней. Скрипт включає до себе 3 властивості, які наведені в таблиці 3.30. Скрипт «EnemyWaypoints» – для зберігання інформації про маршрут переміщення цілей. Скрипт включає до себе 1 властивість, яка наведена в таблиці 3.31, а його методи наведено в таблиці 3.32

Таблиця 3.26 – Властивості скрипту «Enemy"

Тип	Назва	Призначення
Float	totalHp	Сумарна кількість одиниць життя
Float	hp	Поточна кількість одиниць життя
Slider	hpSlider	Рядок життя
Float	speed	Швидкість пересування
GameObject	explosionEffectPrefab	Анімація вибуху
Transform[]	positions	Маршрут пересування
Int	index	Поточна точка на маршруті пересування

Таблиця 3.27 – Методи скрипту «Enemy»

Тип	Назва	Призначення
Void	Start	Підготовка до пересування
Void	Update	Здійснення активності
Void	Move	Пересування
Void	ReachDestination	Маршрут пройдено
Void	OnDestroy	Якщо знищили
Void	TakeDamage	Отримати урон
Void	Die	Об’єкт помирає

Таблиця 3.28 – Властивості скрипту «EnemySpawner»

Тип	Назва	Призначення
1	2	3
EnemyWave[]	waves	Хвилі нападу
Transform	START	Початкова позиція генерації

Продовження таблиці 3.28

1	2	3
Float	waveRate	Затримка між хвилями
Int	CountEnemyAlive	Кількість що вижили для генерації наступної хвилі
Coroutin	coroutine	Корутин генерації

Таблиця 3.29 – Методи скрипту «EnemySpawner»

Тип	Назва	Призначення
Void	Start	Початок генерації
IEnumerator	SpawnEnemy	Створення генерації
Void	Stop	Закінчити генерацію

Таблиця 3.30 – Властивості скрипту «EnemyWave»

Тип	Назва	Призначення
GameObject	enemyPrefab	Модель цілі
Int	count	Кількість цілей
Float	rate	Інтервал генерації

Таблиця 3.31 – Властивості скрипту «EnemyWaypoints»

Тип	Назва	Призначення
Transform[]	positions	Позиції маршруту переміщень

Таблиця 3.32 – Методи скрипту «EnemyWaypoints»

Тип	Назва	Призначення
Void	Awake	Отримання маршруту переміщень

Розглянемо скрипти, як реалізують логіку роботи веж та їх будівництво. Скрипт «Turret» описує вежу. Скрипт включає до себе 10 властивостей, які наведені в таблиці 3.33, а його методи наведено в таблиці 3.34. Скрипт «TurretBuildManager» відповідає за будування веж. Скрипт включає до себе 11 властивостей, які наведені в таблиці 3.35, а його методи наведено в таблиці 3.36. Скрипт «TurretBullet» описує снаряд, що випускає вежа. Скрипт включає

до себе 4 властивості, які наведені в таблиці3.37, а його методи наведено в таблиці 3.38. Скрипт «TurretData» описує тип вежі. Скрипт включає до себе 4 властивості, які наведені в таблиці 3.39. Скрипт «TurretMapCube» відповідає за параметри розміщення веж на ігровій локації. Скрипт включає до себе 5 властивостей, які наведені в таблиці 3.40, а його методи наведено в таблиці 3.41.

Таблиця 3.33 – Властивості скрипту «Turret»

Тип	Назва	Призначення
List<GameObject>	enemies	Цілі
Float	attackRateTime	Довжина інтервалу атаки
Float	timer	Таймер інтервалу атаки
GameObject	bulletPrefab	Модель снаряду
Transform	firePosition	Огньова позиція
Transform	head	Тип башти
Bool	isUseLaser	Флаг використання лазеру
Float	damageRate	Кількість дамагу
LineRenderer	laserRenderer	Лінія лазеру
GameObject	laserEffect	Ефект атаки лазером

Таблиця 3.34 – Методи скрипту «Turret»

Тип	Назва	Призначення
Void	Start	Підготовка до атаки
Void	Update	Робота башти
Void	OnTriggerEnter	Вхід цілі в зону ураження
Void	OnTriggerExit	Вихід цілі з зони ураження
Void	Attack	Атака цілі
Void	UpdateEnemys	Оновити список цілей

Таблиця 3.35 – Властивості скрипту «TurretBuildManager»

Тип	Назва	Призначення
TurretData	laserTurretData	Дані лазерної вежі
TurretData	missileTurretData	Дані ракетної вежі
TurretData	standardTurretData	Дані стандартної вежі
TurretData	selectedTurretData	Дані обраної вежі
TurretMapCube	selectedMapCube	Обраний куб
Text	moneyText	Кількість грошей текст
Animator	moneyAnimator	Кількість грошей анімація
Int	money	Кількість грошей чисельне значення
GameObject	upgradeCanvas	Канвас Оновлення
Button	upgradeButton	Кнопка Оновлення
Animator	upgradeCanvasAnimator	Анімація зміни канвасу оновлення

Таблиця 3.36 – Методи скрипту «TurretBuildManager»

Тип	Назва	Призначення
Void	ChangeMoney	Змінити суму, що залишилася
Void	Update	Будування вежі
Void	OnLaserSelected	Обрано лазерну вежу
Void	OnMissileSelected	Обрану ракетну вежу
Void	OnStandardSelected	Обрано стандартну вежу
Void	ShowUpgradeUI	Показати інтерфейс покращення вежі
IEnumerator	HideUpgradeUI	Сховати інтерфейс покращення вежі
Void	OnUpgradeButtonDown	Покращити вежу
Void	OnDestroyButtonDown	Видалити вежу

Таблиця 3.37 – Властивості скрипту «TurretBullet»

Тип	Назва	Призначення
Int	damage	Кількість завдається дамагу
Float	speed	Швидкість польоту
GameObject	explosionEffectPrefab	Модель вибуху снаряду
Transform	target	Ціль снаряду

Таблиця 3.38 – Методи скрипту «TurretBullet»

Тип	Назва	Призначення
Void	SetTarget	Встановлення цілі
Void	Update	Польоту
Void	OnTriggerEnter	Снаряд потрапив в ціль
Void	Die	Вибух снаряду

Таблиця 3.39 – Властивості скрипту «TurretData»

Тип	Назва	Призначення
GameObject	turretPrefab	Модель вежі
Int	cost	Вартість вежі
GameObject	turretUpgradedPrefab	Модель покращеної вежі
Int	costUpgraded	Ціна покращення

Таблиця 3.40 – Властивості скрипту «TurretMapCube»

Тип	Назва	Призначення
GameObject	turretGo	Вежа під поточним кубом
Bool	isUpgraded	Вежа поліпшена
GameObject	buildEffect	Модель будування вежі
Renderer	cubeRenderer	Поточний куб
TurretData	turretData	Інформація про вежу

Таблиця 3.41 – Методи скрипту «TurretMapCube»

Тип	Назва	Призначення
Void	Start	Підготовка платформи
Void	BuildTurret	Будівництво вежі
Void	OnMouseEnter	Активність при наведенні курсору
Void	OnMouseExit	Активність при виході курсору
Void	UpgradeTurret	Активність з поліпшення вежі
Void	DestroyTurret	Активність з знищення вежі

Скрипт «TowerCameraController» описує рух камери. Скрипт включає до себе 2 властивості, які наведені в таблиці 3.42, а його методи наведено в

таблиці 3.43. Скрипт «TowerGameManager» відповідає за менеджмент сцени. Скрипт включає до себе 3 властивості, які наведені в таблиці 3.44, а його методи наведено в таблиці 3.45.

Таблиця 3.42 – Властивості скрипту «TowerCameraController»

Тип	Назва	Призначення
float	translateSpeed	Кут зору
float	scaleSpeed	Швидкість збільшення кута

Таблиця 3.43 – Методи скрипту «TowerCameraController»

Тип	Назва	Призначення
void	Update	Здійснення активності

Таблиця 3.44 – Властивості скрипту «TowerGameManager»

Тип	Назва	Призначення
GameObject	endUI	Меню кінця гри
Text	endMessage	Повідомлення
EnemySpawner	enemySpawner	Генератор цілей

Таблиця 3.45 – Методи скрипту «TowerGameManager»

Тип	Назва	Призначення
void	Awake	Підготовка до початку гри
void	Win	Виведення повідомлення про перемогу
void	Failed	Виведення повідомлення про програш
void	OnButtonRetryDown	Повторити гру
void	OnButtonHomeDown	Повернутися на сцену Home

4 МЕТОДИКА РОБОТИ З РОЗВАЖАЛЬНИМ ІГРОВИМ ДОДАТКОМ «SEEKER»

4.1 Меню запуску гри «Unity»

Для завантаження додатку необхідно запустити бінарний файл програми. В результаті з'явіться вікно конфігурації програми «Unity», в якому присутні вкладки «Graphics» і «Input».

На вкладці «Graphics» (рис. 4.1) присутній флаг «Windowed» для запуску гри в віконному форматі, кнопки «Play!» та «Quit» для початку гри та виходу з неї, відповідно, а також доступні параметри налаштування розширення, якості графіки та дисплею для запуску.

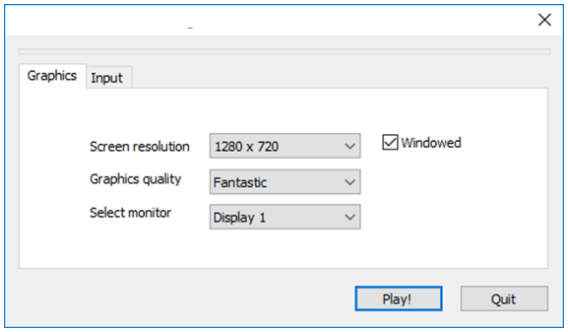


Рисунок 4.1 – Вкладка «Graphics» вікна конфігурації програми «Unity»

.

Вкладка «Input» (рис. 4.2) дозволяє переглянути доступні для керування грою кнопки. А також перекинути використанні в грі кнопки, здійснивши подвійний клик по відповідному рядку.

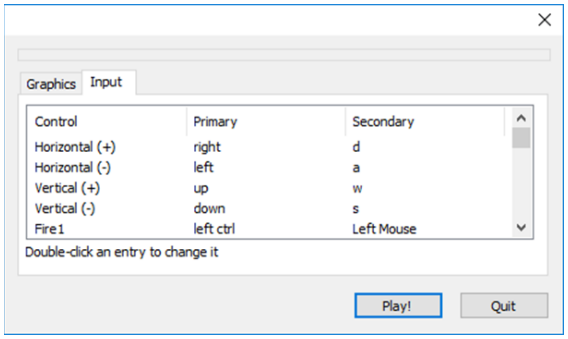


Рисунок 4.2 – Вкладка «Input» вікна конфігурації програми «Unity»

4.2 Головне ігрове меню

Головне ігрове меню (рис. 4.3) дозволяє переглянути інформацію про гру, почати гру, а також вийти з гри. Для перегляду інформації про гру потрібно клацнути на клавішу з буквою “*i*” у верхньому правому куті програми. Після чого відкриється екран інформації. Для початку гри потрібно клацнути на кнопку грати, а для завершення гри потрібно кликнути на кнопку з хрестиком, після чого гра закриється.

Рисунок 4.3 – Головне ігрове меню

4.3 Екран інформації

Екран інформації «Info» програми запускається з головного меню та призначений для отримання інформації про гру та автора (рис. 4.4). Перехід зі сцени відбувається автоматично через певний час.

Рисунок 4.4 – Інформаційний екран гри

4.4 Головна ігрова локація

Після натиснення кнопки гри в головному меню програми запускається базова ігрова локація (рис. 4.5). Вона представлена 3D сценою у вигляді будівлі, котра має декілька кімнат. Дизайн локації проектувався на базі моделей, зроблених за принципом «Low Poly».

Ця локація на відмінно від всіх інших характерна тим, що маніпулювання персонажем відбувається за допомогою кликів миші. Після того, як гравець зробить клік мишею до відповідної точки ігрової локації персонаж намагається пройти до неї. Персонаж пересувається по будівлі з урахуванням реальних законів фізики.

Гравець дивиться на кімнату з точки зору третьої особи, яка знаходиться над будівлею та не переміщується.

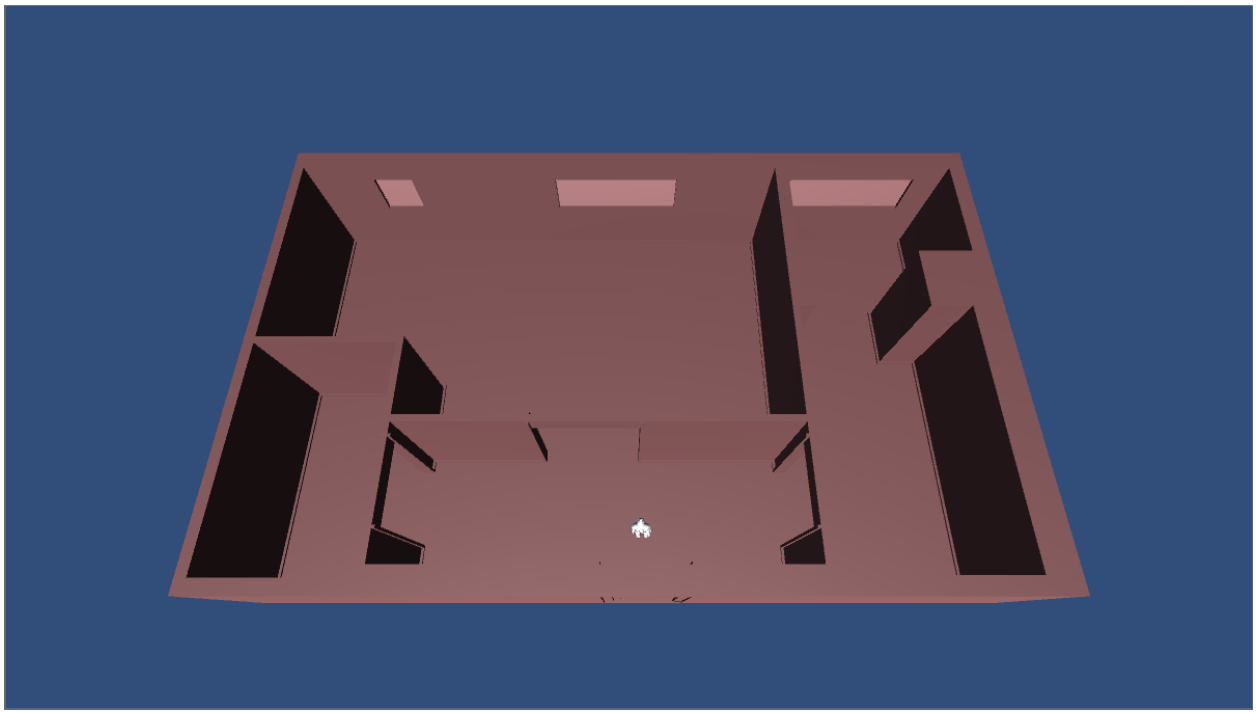


Рисунок 4.5 – Основна ігрова локація

Механіка гри заснована на принципах відкритого світу. На сцені є два активних компоненти - це комп’ютери. Про можливість взаємодіяти з ними повідомляється користувачу у правому верхньому куті програми. При взаємодії з ними відкривається одна з міні ігор. Тобто, для того, щоб грати у «Flappy Bat» або «Tower Defense», гравець має можливість обрати комп’ютер з відповідною міні грою.

4.5 Міні гра «Flappy Bat»

Міні-гра «Flappy Bat» являє собою типову 2D відео-гру та є складовим елементом головної локації. Основною ідеєю гри є управління польотом кажана у печері без освітлення.

Під час запуску гри перед нами з’являється вітальне меню гри (рис. 4.6), яке має підказку як почати гру та здійснювати гру. Після початку гри починається збільшення ігрового рахунку, чим далі відлітає кажан від точки старту, тим більше його ігровий рахунок. Бали нараховуються поточно, з рівними проміжками часу, а максимальна кількість балів майже не обмежена. Кінцевою метою гравця в грі є пролетіти якнайбільшу відстань без зіткнень з

перешкодами. Під час усього ігрового процесу ігровий геймплей не змінюється.

Рисунок 4.6 – Вітальне меню гри «Flappy Bat»

Механіка міні гри представлена управлінням польотом кажана у печері, в якій немає освітлення. Через рівні проміжки часу кажан випромінює хвилі, котрі, у свою чергу, підсвічують частину перешкоди. Що дозволятиме з часом увійти у ритм та заробити якнайбільше балів, тобто гра має під собою повторення ігрового процесу з метою збільшення здобутих балів.

На старті гри кажан знаходиться у польоті, але після старту гри гравець бере на себе функцію керування кажаном. В процесі гри (рис. 4.7) гравець повинен боротися з гравітацією і уникати перешкод, які йому зустрічаються.

Після чого виводиться меню результату гри (рис. 4.8), у якому користувач може обрати повернутися до основної ігрової локації, чи продовжити грати.

Рисунок 4.7 – Процес гри «Flappy Bat»

Рисунок 4.8 – Меню результату гри «Flappy Bat»

4.6 Міні гра «Tower Defense»

Міні гра «Tower Defense» – це гра, основна мета якої полягає в тому, щоби не дати рухомим об’єктам дійти до кінця мапи за допомогою башт, котрі мають наносити шкоду цим рухомим об’єктам.

Рухомі об’єкти рухаються по заздалегідь фіксованому маршруту, який

нагадуватиме простий лабіринт, біля котрого мають бути побудовані вежі, котрі будуть атакувати рухомі об’єкти.

Ігровий геймплей (рис. 4.9) представлений двома компонентами: сценою, де відбуваються усі дії, та меню вибору.

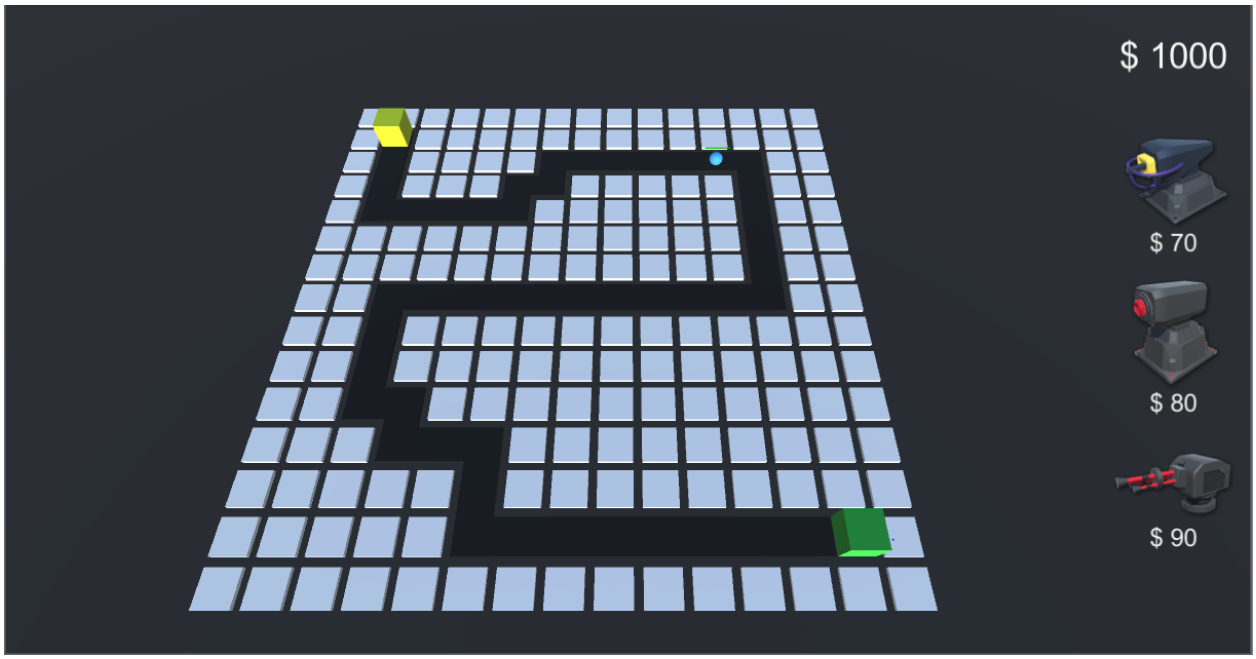


Рисунок 4.9 – Ігровий геймплей «Tower Defense»

Закінчення гри можливо за двома сценаріями (рис. 4.10). Перший, виграш, який відбувається після закінчення усіх хвиль супротивників. Інший, програш, який відбувається якщо бодай один рухомий об’єкт пересунеться через увесь свій шлях, не отримавши потрібної кількості шкоди від діяльності башт, то він перетинає мапу та гра закінчується. У такому сценарії гравець програє, тобто єдина мета гри – не дати рухомим об’єктам дійти до кінця, знищівши їх усіх за допомогою башт.

Для встановлення башти на ігрову мапу потрібно спочатку обрати модель башти та натиснути по місцю необхідної дислокації, яке підсвічується червоним кольором.

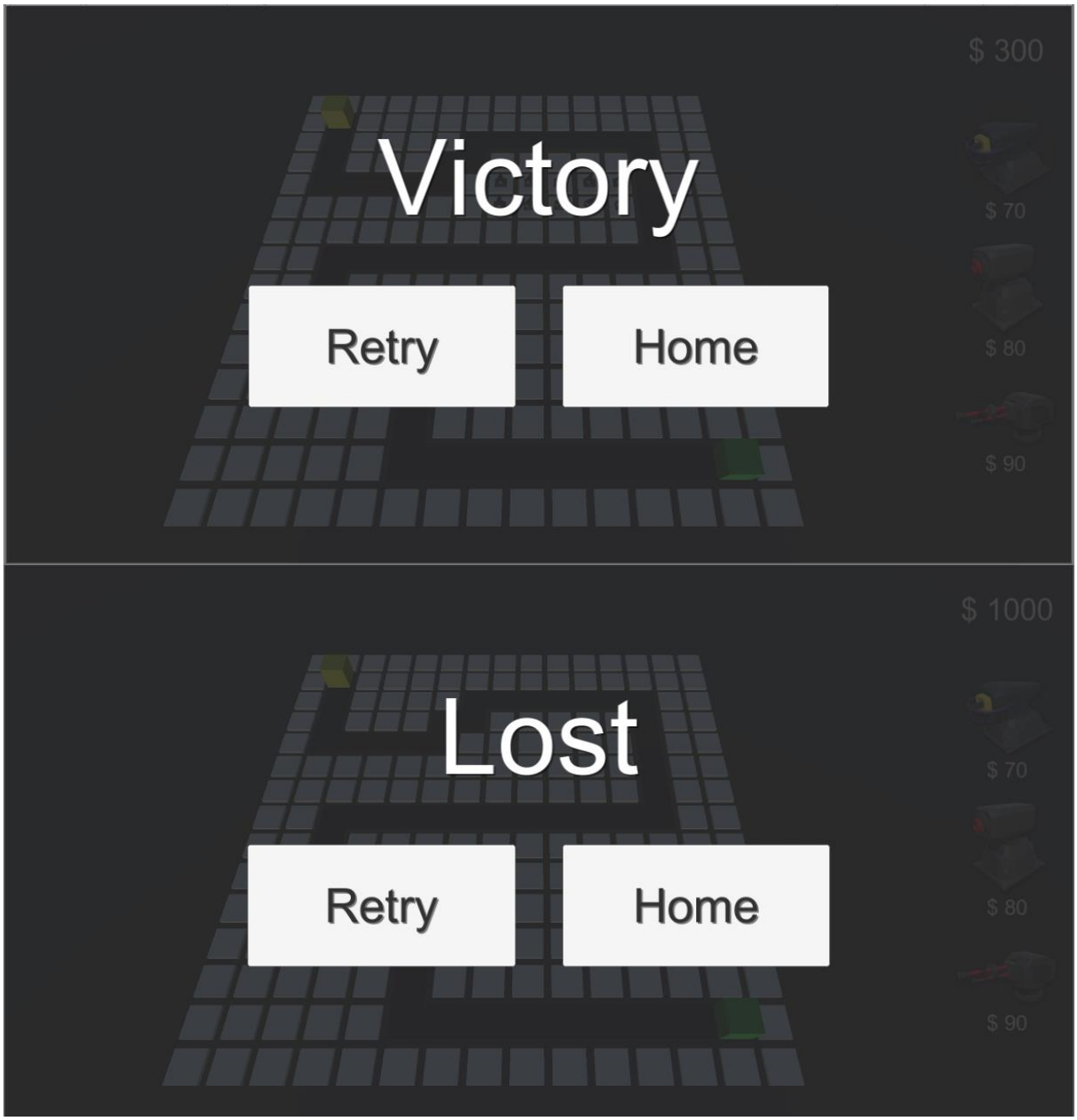


Рисунок 4.10 – Результати гри «Tower Defense»

5 ТЕСТУВАННЯ РОЗВАЖАЛЬНОГО ІГРОВОГО ДОДАТКУ «SEEKER»

5.1 Загальні положення тестування

Для тестування системи використовувався динамічний підхід до тестування, що має на увазі виконання вихідного коду.

Тестування системи здійснювалося з точки зору функціональної складової. Яка має на увазі під собою здійснення тестування на функціональну придатність системи, яка виражається в регламентованій поведінці системи.

Тестування системи здійснювалося методом інтеграції. Цей метод практично ідеально підходить для тестування ігрових додатків і дозволяє виявити проблеми на ранніх етапах створення гри. Тестування методом інтеграції має на увазі під собою три рівні тестування.

Перший рівень – тестування компонентів, проходи за принципом тестування через розробку. Цей принцип має чотири етапи: постановка завдання, визначення очікуваного результату, розробка модулю, перевірка його відповідним вимогам.

Другий рівень – інтеграційне тестування. Після написання модулю перевірялась сумісна робота моделей. Ця перевірка спрямована на перевірку взаємодії інтегрованого модуля з іншими компонентами системи.

Третій рівень – системне тестування. Тестування готової системи на працездатність. Підсумкове тестування програми, яка покликана виявити недоліки, які були виявлені в інтеграційному тестуванні. Часто на цьому етапі тестування переробляється призначений для користувача інтерфейс додатків.

Для скорочення кількості тестів та поліпшення якості тестування використовувалися формальний та інтуїтивний підходи до тестування.

5.2 Порядок здійснення тестування

Порядок здійснення тестування залежить від послідовності розробки програмного продукту. Для скорочення витрат і кількості циклів проведення

тестування були спочатку розроблені дві міні ігри («Flappy Bat» і «Tower Defense»), а так само був створений каркас основної ігрової локації системи.

Коли дані компоненти були розроблені і було проведено їх компонентне тестування, почався процес інтеграції. Спочатку в основну ігрову локацію було інтегровано міні гру «Flappy Bat» і було проведено тестування коректної спільної роботи модулів. Після чого аналогічнім чином біло інтегровано міні гру «Tower Defense».

Після завершення процесу інтеграційного тестування почалася розробка основного меню гри і додатково сцени про автора. Оскільки ці елементи досить маленькі, то вони створювалися як один компонент. По закінченню розробки даного компонента було проведено його компонентне тестування і він був інтегрований в середу. Після інтегрування до середи було проведено інтеграційне тестування.

Процес системного тестування включив до себе тестування системи за деякими підходами сірого ящику.

5.3 Результати проведення тестування

В результаті проведення тестування усувалися, як серйозні так і не серйозні помилки. Після чого вони були усунуті.

Однією з виявлених проблем (рис. 5.1) при проведенні тестування був маленький розмір сцени гри «Flappy Bat», що призводило в разі широких моніторів до синіх смуг по боках. Це відбувалося оскільки орографічна камера збільшує кут свого огляду [11] в будь-якому обраному користувачем дозволу. Цей дефект вийшло ліквідувати без перероблювання сцени, просто за допомогою заміни кольору заднього фону камери на чорний колір.

Іншою виявленою проблемою (рис. 5.2) при створенні програми сталі розгорнуті полігони на 3D моделях. Вона була виявлена, коли створенні моделі з «Autodesk 3ds Max» переносились до сцени у «Unity». Моделі у «Unity» відображуватись поніввеченими чи з прозорими гранями під певним кутом. Причину виникнення проблеми так і не вдалося встановити. Також не

вдалось розвернути полігони до нормального стану із-за чого 47 розроблених моделей не є пригідними для використання. Це привело до створення нових моделей.

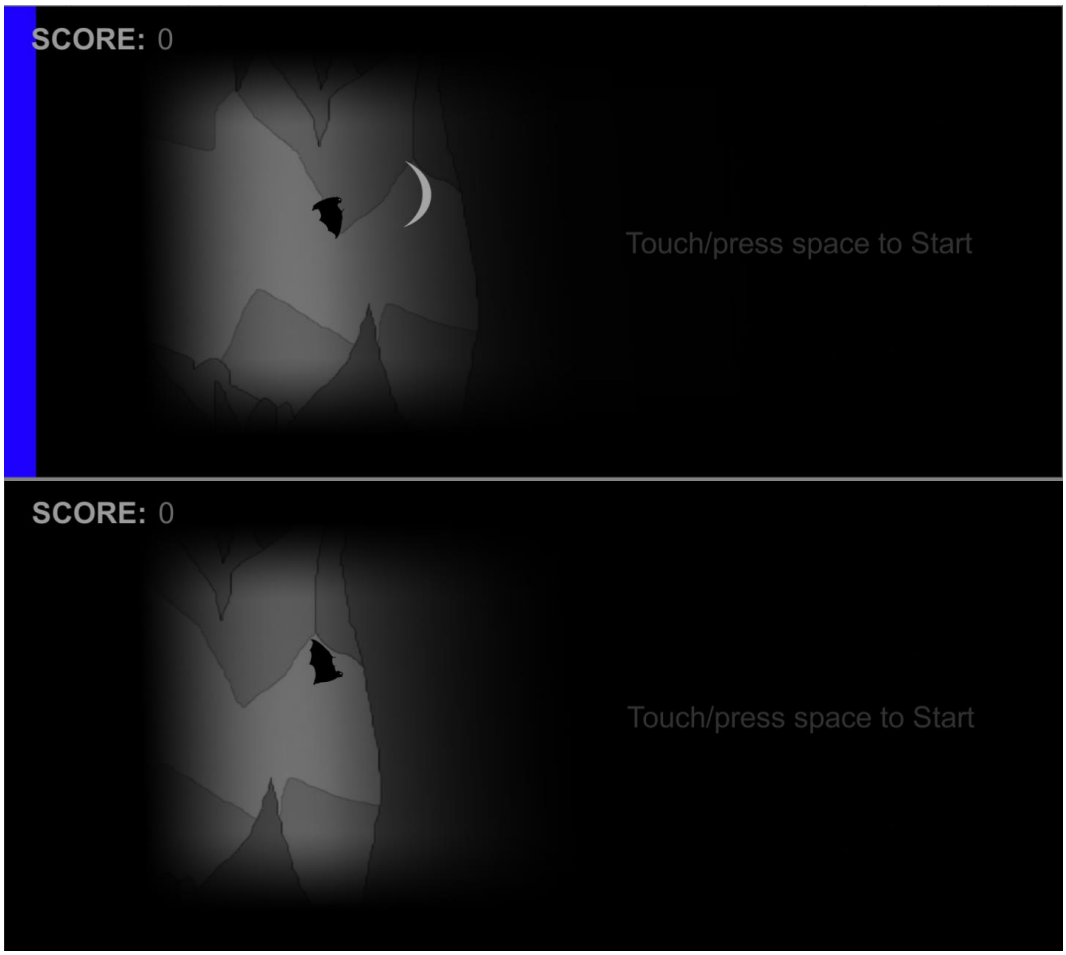


Рисунок 5.1 – Проблема розміру камери

Рисунок 5.2 – Приклад об’єкта з розгорнутими полігонами

ВИСНОВКИ

В процесі виконання роботи були розглянуті технології, застосовувані при розробці відеогри на базі рушія, існуючі аналоги та розробки, поняття, стандарти й дослідження в цій галузі.

На основі зібраного матеріалу були спроектовано та розроблено розважальну відеогру на базі Unity. Розроблена відеогра складається з 3 ігрових сцен та 2 додаткових сцен, які мають у собі головне меню гри та інформацію про гру. Розробка сцен включала знаходження відповідних ігрових матеріалів або їх самостійне виготовлення. Розроблені ігрові матеріали представлені текстурами та відповідними матеріалами, а також спрайтими та моделями, до яких вони використовувались. Розроблена програма включає більш ніж 1300 строк вихідного коду.

Результати тестування підтвердили працездатність розробленого ігрового додатку. Розроблена гра підтверджує можливість використання ігрового рушію для розробки великої кількості різнопланових за жанром та ігровою механікою відеоігор.

У даній роботі було розглянуто і використано три варіанти переміщення. Перше, переміщення за вектором - це гра «Flappy Bat», вектор в одну сторону. Другий варіант переміщення - це переміщення за точками «Waypoints». Третій варіант переміщення «NavMesh». Дані технології є базовими для переміщення в комп'ютерних іграх і в основному використовуються тільки вони. Більшість технології переміщення та пошуку маршруту будується на цих базових технологіях. Існує ще комбінація технологій «Waypoints» і «NavMesh», яка дозволяє будувати досить складні ігрові логіки і інтелект.

В результаті дослідження отримано, що «Unity» – це найкраще з існуючих середовище розробки простих комп'ютерних ігор, яке дозволяє розробляти тривимірні та двовимірні ігри практично для будь-якої платформи. До переваг «Unity» можна віднести легкість у використанні; сумісність з будь-

якою платформою. Однак, як і у будь-якого програмного продукту, у нього також є недоліки – це обмежений набір інструментів для розробників і процес виготовлення гри забирає багато часу. Найголовніший недолік «Unity» – це 2D графіка. Мається на увазі створення 2D ігор. Створити таку гру можна, але доведеться затратити немало часу. Недостатньо префабів, якихось готових речей, об'єктів, наприклад, будинків, предметів елементарного інтер'єру - столів, стільців. Необхідно, щоби робота з оптимізацією зображення під різні дозволи екрану була простіше. Адже достатньо трохи попрацювати з перетворенням координат і тоді не треба буде замислюватися про те, що треба обчислювати розміри, координати через розмір екрана.

Матеріали даної роботи можуть бути запозичені до робіт відповідних за жанровою характеристикою та ігровою механікою. А сама робота може бути використана як приклад для подальшого застосування відповідного інструментарію.

Надалі планується покращення функціональних характеристик додатку та розширення кількості ігрових локацій. Також планується вдосконалення логіки, додавання підтримки для інших платформ, зокрема мобільних.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Newzoo Free 2016 Global Games Market // an overview of trends & insights - June 2016 [Електронний ресурс] — Режим доступу. — URL: http://resources.newzoo.com/hubfs/Reports/Newzoo_Free_2016_Global_Games_Market_Report.pdf (Дата звернення: 23.04.2017)

2. Thomsen, Michael. The 'Indie' Delusion: The Gaming Category that Doesn't Exist. IGN (January 25, 2011). [Електронний ресурс] – Режим доступу. – URL: http://www.ign.com/articles/2011/01/26/the-indie-delusion-the-gaming-category-that-doesnt-exist?page=1 (Дата звернення: 23.04.2017)

3. Indie game - Wikipedia, the free encyclopedia [Електронний ресурс] – Режим доступу. – URL: https://en.wikipedia.org/wiki/Indie_game

4. Paradox Interactive – Wikipedia, the free encyclopedia. [Електронний ресурс] – Режим доступу. – URL: https://en.wikipedia.org/wiki/Paradox_Interactive (Дата звернення: 23.04.2017)

5. Кривенко О.М. Огляд і аналіз технологій і засобів розробки комп'ютерних ігор / О.М. Кривенко // XI Регіональна студентська науково-технічна конференція «Наука – перші кроки»: тези доповідей: Т. 4. – Маріуполь : ПДТУ, 2017. – 68 с.

6. Кривенко О.М. Аналіз сучасних технологічних тенденцій у розробці комп'ютерних ігор / О.М. Кривенко, С.А. Зорі // XXXVI Всеукраїнська науково-практична Інтернет-конференція «Перспективи розвитку наукових досліджень в XXI столітті» – Україна, м. Дніпропетровськ, 2017. – 68 с.

7. List of game engines [Електронний ресурс] – Режим доступу. – URL: https://en.wikipedia.org/wiki/List_of_game_engines (Дата звернення: 23.04.2017)

8.][-обзор: сравниваем топ-6 лучших игровых движков для программиста – Компьютерный журнал хакер – Режим доступу. – URL: https://xakep.ru/2016/10/17/top6-game-engines/

9. Сделай сам: Кроссплатформенные игровые движки 2016 – Apptractor

Електронний ресурс] – Режим доступу. – URL: http://apptractor.ru/info/articles/igrovyie-dvizhki.html

10. Alex Okita. Learning C# Programming with Unity 3D / Alex Okita. – CRC Press, 2014. – 671 Pages.

11. Patrick Felicia. A Beginner's Guide to 2D Platform Games with Unity: Create a Simple 2D Platform Game and Learn to Code in C# in the Process (Kindle Edition) / Patrick Felicia. – Patrick Felicia, 2016. – 144 Pages.

12. Кенни Ламмерс. Шейдеры и эффекты в Unity. Книга рецептов / Кенни Ламмерс. – ДМК Пресс, 2014. – 274 Pages.

13. Brian Moakley. Unity Games by Tutorials: Make 4 Complete Unity Games from Scratch Using C# / Brian Moakley, Mike Berg and other. – Apress, 2016. – 352 Pages.

14. Jere Miles. Unity 3D and Playmaker Essentials: Game Development from Concept to Publishing / – CRC Press, 2016. – 482 Pages.

15. Joe Hocking. Unity in Action: Multiplatform Game Development in C# with Unity 5 / Joe Hocking. – Manning, 2015. – 327 Pages.

16. Jere Miles. Unity 3D and PlayMaker Essentials: Game Development from Concept to Publishing / Jere Miles. – CRC Press, 2016. – 482 Pages.

17. Alan Thorn. Mastering Unity Scripting / Alan Thorn. – CRC Press, 2015. – 482 Pages.

18. Sue Blackman, Adam Tuliper. Learn Unity for Windows 10 Game Development / Sue Blackman, Adam Tuliper. – Apress, 2016. – 572 Pages.

19. John P. Doran. Unity Game Development Blueprints / John P. Doran.– Paperbac, 2014. – 297 Pages.

20. Michael Kelley. No-Code Video Game Development Using Unity and Playmaker/ Michael Kelley. – CRC Press, 2016. – 292 Pages.

21. How to make a Tower Defense Game (E01) - Unity Tutorial [Електронний ресурс] – Режим доступу. – URL: https://youtu.be/beuoNuK2tbk?list=PLPV2KyIb3jR4u5jX8za5iU1cqnQPmbzG0

Додаток А

Зауваження нормоконтролера

Кваліфікаційна робота бакалавра за темою «Інструментальна підтримка розробки ігрових додатків» зі спецчастиною «Розробка розважального ігрового додатку на базі движка «Unity».

Таблиця А.1 – Зауваження нормоконтролера

Сторінка	Позначення	Зміст зауваження

Додаток Б

Лістинг програми розважального ігрового додатку «Seeker»

Б.2. Лістинги базових скриптів

Б.2.1. Лістинг скрипту «Transition.cs»

```
using UnityEngine;
using System.Collections;

public class Transition : MonoBehaviour
{

 public bool startDark = false;
 public float timerMultiplyer = 2f;
 public bool TransitionFinished{get;
private set;}
 public bool TransitionStarted { get;
private set; }
 public bool
startTransitioningImmediately = true;
 public float minAlpha = 0;
 public float maxAlpha = 1;
 SpriteRenderer spriteRenderer;
 float timer = 0f;
 public Color startColor;
 public Color endColor;

 // Use this for initialization
 void Start () {
  spriteRenderer =
GetComponent<SpriteRenderer>();
  startColor = new Color(startColor.r,
startColor.g, startColor.b, startDark ?
maxAlpha : minAlpha);
  endColor = new Color(endColor.r,
endColor.g, endColor.b, startDark ?
minAlpha : maxAlpha);
  //HACK .. I Dunno Why This Is
Happening -_-
  if (startDark)
   spriteRenderer.color = startColor;
  else spriteRenderer.material.color =
startColor;

  if (startTransitioningImmediately)
   TransitionStarted = true;
 }

 // Update is called once per frame
 void Update () {
  if (!TransitionStarted ||
TransitionFinished) return;
  timer += Time.deltaTime*
timerMultiplyer;
  spriteRenderer.material.color =
Color.Lerp(startColor, endColor,
timer);
  if ((startDark &&
spriteRenderer.material.color.a <=
minAlpha) || (!startDark &&
spriteRenderer.material.color.a >=
maxAlpha))
   TransitionFinished = true;
 }

 public void Reset(bool startDark,Color
startColor,Color endColor)
 {
  TransitionFinished = false;
  TransitionStarted = false;
  this.startDark = startDark;
  this.startColor = new
Color(startColor.r, startColor.g,
startColor.b, startDark ? maxAlpha :
minAlpha);
  this.endColor = new Color(endColor.r,
endColor.g, endColor.b, startDark ?
minAlpha : maxAlpha);
  spriteRenderer.material.color =
startColor;
  timer = 0;
 }

 public void StartTransition()
 {
  TransitionStarted = true;
 }
}
```

Б.2.2. Лістинг скрипту «MenuController.cs»

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

public class MenuController :
MonoBehaviour {

 public AudioSource BackgroundMusic;
```

```
 // Use this for initialization
 void Start () {
  DontDestroyOnLoad (BackgroundMusic);
 }

 // Update is called once per frame
 void Update () {

 }

 public void OnButtonPlayGameDown(){
  SceneManager.LoadScene("Home",
LoadSceneMode.Single);
 }

 public void OnButtonShowInfoDown(){
  SceneManager.LoadScene("Info",
LoadSceneMode.Single);
 }

 public void OnButtonExitGameDown(){
  Application.Quit ();
 }
}
```

Б.2.3. Лістинг скрипту «InfoSplash.cs»

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

public class InfoSplash : MonoBehaviour
{

 public float Timer = 0;
 public float TimeOut = 5;
 public Transition transition;
 bool transitionBegun = false;
 float transitionTimeOut;
 // Use this for initialization
 void Start () {
  transitionTimeOut = TimeOut - 2;
 }

 // Update is called once per frame
 void Update () {
  Timer += Time.deltaTime;
  if (!transitionBegun && Timer >=
transitionTimeOut)
  {
   transition.Reset(false, Color.black,
Color.black);
   transition.StartTransition();
   transitionBegun = true;
  }

  if (Timer < TimeOut) return;

  SceneManager.LoadScene("Menu",
LoadSceneMode.Single);
 }
}
```

Б.3. Лістинг скриптів сцени «Home»

Б.3.1. Лістинг скрипту «PlaceTargetWithMouse.cs»

```
using System;
using UnityEngine;


namespace
UnityStandardAssets.SceneUtils
{
 public class PlaceTargetWithMouse :
MonoBehaviour
 {
  public float surfaceOffset = 1.5f;
  public GameObject setTargetOn;

  // Update is called once per frame
  private oid Update()
  {
   if (!Input.GetMouseButtonDown(0))
   {
    return;
   }
   Ray ray =
Camera.main.ScreenPointToRay(Input.mous
ePosition);
   RaycastHit hit;
   if (!Physics.Raycast(ray, out hit))
   {
    return;
   }
   transform.position = hit.point +
hit.normal*surfaceOffset;
   if (setTargetOn != null)
   {
```

```
setTargetOn.SendMessage("SetTarget",
transform);
   }
  }
 }
}
```

Б.3.2. Лістинг скрипту «ActionTrigger.cs»

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

public class ActionTrigger :
MonoBehaviour {

 public GameObject TriggerMesh;
 public GameObject ActionHelper;
 public string SceneName;

 bool inTrigger = false;

 void OnTriggerEnter(Collider other)
 {
  if (!inTrigger && other.tag ==
"Player") {
   ActionHelper.SetActive(true);
   Debug.Log("Enter trigger");
   inTrigger = true;
  }
 }

 void OnTriggerStay(Collider other)
 {
  if (other.tag == "Player" &&
Input.GetKeyDown (KeyCode.Space)) {
   SceneManager.LoadScene(SceneName,
LoadSceneMode.Single);
  }
 }

 void OnTriggerExit(Collider other)
 {
  if (inTrigger && other.tag ==
"Player") {
   ActionHelper.SetActive(false);
   Debug.Log("Exit trigger");
  }
 }

}
```

Б.4. Лістинг скриптів сцени «Cave»

Б.4.1. Лістинг скрипту «BatController.cs»

```
using UnityEngine;
using System.Collections;

public class BatController :
MonoBehaviour {

 CaveGameManager gameManager;
 public AudioSource flapSound;
 public GameObject wavePrefab;
 private Rigidbody2D body2D;
 public Vector2 waveOffset;
 public float screamingPeriod;
 float screamingTimer;
 public GameObject shadow;
 SpriteRenderer shadowRender;
 // Use this for initialization
 void Start () {
  body2D = GetComponent<Rigidbody2D>
();
  gameManager =
GameObject.FindObjectOfType<CaveGameMan
ager> ();
  shadowRender =
shadow.GetComponent<SpriteRenderer> ();
 }

 void OnCollisionEnter2D(Collision2D
target)
 {
  if (target.gameObject.tag == "Rock")
   gameManager.GameOver ();
 }

 // Update is called once per frame
 void Update () {
  if (screamingTimer > 0) {
   screamingTimer -= Time.deltaTime;
  } else {
   Scream ();
  }

  if (Input.GetKeyDown (KeyCode.Space)
|| Input.GetMouseButtonDown(0) ||
Input.touches.Length>0 &&
```

```
Input.touches[0].phase ==
TouchPhase.Began) {
   if (gameManager.gameStarted)
    Rise ();
   else {
    body2D.isKinematic = false;
    gameManager.StartGame ();
   }
  }
 }

 void Rise()
 {
  if (!gameManager.gameStarted)
   return;
  if (!gameManager.gameIsOver) {
   body2D.velocity = new Vector2
(body2D.velocity.x, 4.2f);
   flapSound.PlayOneShot
(flapSound.clip);
  }
 }

 void Scream(){
  if (gameManager.gameIsOver)
   return;
  shadowRender.color *= 0.95f;
  GenerateWave ();
  screamingTimer = screamingPeriod;
 }

 void GenerateWave()
 {
  Instantiate (wavePrefab, new
Vector2(transform.position.x+waveOffset
.x,transform.position.y+waveOffset.y),
Quaternion.identity);
 }

 void FixedUpdate(){
  var angle = 0f;
  if (body2D.velocity.y < 0) {
   angle = Mathf.Lerp (0, -90, -
body2D.velocity.y/10);
  }
  transform.rotation = Quaternion.Euler
(0, 0, angle);
 }
}
```

Б.4.2. Лістинг скрипту «BatWave.cs»

```
using UnityEngine;
using System.Collections;

public class BatWave : MonoBehaviour {

 public float speed;
 public float scaleSpeed;
 public float alphaSpeed;
 private SpriteRenderer WaveRenderer;
 private Rigidbody2D body2D;
 // Use this for initialization
 void Start () {
  WaveRenderer =
GetComponent<SpriteRenderer> ();
  body2D = GetComponent<Rigidbody2D>
();
  body2D.velocity = new Vector2 (speed,
0);
 }

 // Update is called once per frame
 void Update () {
  transform.localScale = new Vector2
(transform.localScale.x,
transform.localScale.y + scaleSpeed);
  WaveRenderer.color = new Color (0.7f,
0.7f, 0.7f, WaveRenderer.color.a -
alphaSpeed);
  if (transform.position.x>=10)
   Destroy (this.gameObject);
 }
}
```

Б.4.3. Лістинг скрипту «CaveBackground.cs»

```
using UnityEngine;
using System.Collections;

public class CaveBackground :
MonoBehaviour {
 Renderer rend;
 float offset;
 public float speed;
 // Use this for initialization
 void Start () {
  rend = GetComponent <Renderer> ();
 }

 // Update is called once per frame
 void Update () {
  offset += speed;
  rend.material.mainTextureOffset = new
Vector2 (offset, 0);
 }
}
```

Б.4.4. Лістинг скрипту «CaveGameManager.cs»

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

public class CaveGameManager :
MonoBehaviour {

 public GameObject blank;
 public GameObject gameoverPanel;
 SpriteRenderer blankRenderer;
 public float alphaSpeed = 0.01f;
 private CaveRocksGen rocksGen;
 public bool gameIsOver;
 public bool gameStarted = false;
 public Text infoText;
 int score;
 int bestScore;
 float scoreTimer;
 public Text scoreText;
 public Text finalScoreText;
 public Text bestText;
 // Use this for initialization
 void Start () {
  rocksGen =
GameObject.FindObjectOfType<CaveRocksGe
n> ();
  blankRenderer =
blank.GetComponent<SpriteRenderer> ();

  if (!PlayerPrefs.HasKey
("SavedHiScore"))
   PlayerPrefs.SetInt ("SavedHiScore",
bestScore);
  else
   bestScore = PlayerPrefs.GetInt
("SavedHiScore");
 }

 // Update is called once per frame
 void Update () {
  if (gameIsOver &&
blankRenderer.color.a < 1) {
   blankRenderer.color = new Color (1,
1, 1, blankRenderer.color.a +
alphaSpeed);
  }
  if (gameIsOver || !gameStarted)
   return;
  scoreTimer += Time.deltaTime;
  score = (int)(scoreTimer * 2);
  scoreText.text = score.ToString ();
 }

 public void OnButtonRetryDown()
 {

SceneManager.LoadScene(SceneManager.Get
ActiveScene().buildIndex,
LoadSceneMode.Single);
 }

 public void OnButtonHomeDow()
 {
  SceneManager.LoadScene("Home",
LoadSceneMode.Single);
 }

 public void ShowGameOver()
 {
  gameoverPanel.SetActive (true);
 }

 public void GameOver()
 {
  if (gameIsOver)
   return;
  gameIsOver = true;
  rocksGen.StopGenerating ();
  if (score > bestScore) {
   bestScore = score;
   bestText.text = bestScore.ToString
();
   PlayerPrefs.SetInt ("SavedHiScore",
bestScore);
  } else
   bestText.text = bestScore.ToString
();
  finalScoreText.text = scoreText.text;
  ShowGameOver ();
 }

 public void StartGame(){
  Destroy (infoText);
  gameStarted = true;
 }
}
```

Б.4.5. Лістинг скрипту «CaveRock.cs»

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

```
public class CaveRock : MonoBehaviour {
 public float speed;
 Rigidbody2D body2D;
 public CaveRockOutline outline;
 // Use this for initialization
 void Start () {
  body2D = GetComponent<Rigidbody2D>
();
  body2D.velocity = new Vector2 (speed,
0);
 }

 // Update is called once per frame

 void OnTriggerEnter2D(Collider2D
target)
 {
  if (target.tag == "wave") {
   outline.Glow ();
  }
 }
}
```

Б.4.6. Лістинг скрипту «CaveRockOutline.cs»

```
using UnityEngine;
using System.Collections;

public class CaveRockOutline :
MonoBehaviour {

 SpriteRenderer rend;
 private Color defaultColor;
 public float alphaSpeed;
 // Use this for initialization
 void Start () {
  defaultColor = new Color (0.5f, 0.5f,
0.5f);
  rend = GetComponent<SpriteRenderer>
();
  rend.color = defaultColor*0;
 }

 // Update is called once per frame
 void Update () {
  if (rend.color.a > 0)
   rend.color = new Color
(defaultColor.r, defaultColor.g,
defaultColor.b, rend.color.a -
alphaSpeed);
 }

 public void Glow(){
  rend.color = defaultColor;
 }
}
```

Б.4.7. Лістинг скрипту «CaveRocksGen.cs»

```
using UnityEngine;
using System.Collections;

public class CaveRocksGen :
MonoBehaviour {

 public GameObject[] prefabsList;
 CaveGameManager gameManager;
 public float generationPeriod;
 float generationTimer;
 bool canGenerate = true;
 void Start () {
  gameManager =
GameObject.FindObjectOfType<CaveGameMan
ager> ();
 }

 void CreatePrefab(){
  if (!canGenerate)
   return;
  var prefabIndex = Random.Range (0,
prefabsList.Length);
  GameObject clone =
Instantiate(prefabsList[prefabIndex])
as GameObject;
  UnityEngine.Behaviour.Destroy (clone,
15);
  generationTimer = generationPeriod;
 }

 // Update is called once per frame
 void Update () {
  if (!gameManager.gameStarted)
   return;
  if (generationTimer > 0)
   generationTimer -= Time.deltaTime;
  else
   CreatePrefab ();
 }

 public void StopGenerating(){
  canGenerate = false;
 }
}
```

Б.4.8. Лістинг скрипту «CaveShadow.cs»

```
using UnityEngine;
using System.Collections;

public class CaveShadow : MonoBehaviour
{

 SpriteRenderer srender;
 public float alphaSpeed;
 // Use this for initialization
 void Start () {
  srender =
GetComponent<SpriteRenderer> ();
 }

 // Update is called once per frame
 void Update () {
  if (srender.color.a < 1)
   srender.color = new Color (1, 1, 1,
srender.color.a + alphaSpeed);
 }
}
```

Б.5. Лістинг скриптів сцени «Tower»

Б.5.1. Лістинг скрипту «Enemy.cs»

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

// 一个Enemy就是一个敌人
public class Enemy : MonoBehaviour {

 private float totalHp; // Save the
total amount of blood to calculate the
blood stripe progress. Because hp will
be diminished
 public float hp = 150; // enemy blood
 public Slider hpSlider; // blood
strips
 public float speed = 10; // Moving
speed
 public GameObject
explosionEffectPrefab; // enemy death
explosion effects
 private Transform[] positions; // all
path points
 private int index = 0; // Which point
is moving now?

 void Start()
 {
  // Save the blood
  totalHp = hp;
  // Get an array of path points
  positions = EnemyWaypoints.positions;
 }

 void Update()
 {
  Move();
 }

 // Monster mobile processing
 void Move()
 {
  // To prevent cross-border
  if (index > positions.Length - 1)
return;
  // Target point - own point =
direction of its own point to the
target point

transform.Translate((positions[index].p
osition -
transform.position).normalized *
Time.deltaTime * speed);
  // If the position of the target
point and its own point is less than
0.2 m, it starts to move like the next
target point
  if
(Vector3.Distance(positions[index].posi
tion, transform.position) < 0.2)
  {
   index++;
  }
  // The last execution, index will be
greater than the array maximum angle
  if (index > positions.Length - 1)
  {
   // The current enemy has arrived at
the destination
   ReachDestination();
  }
 }
```

```
 // The enemy arrives at the
destination
 void ReachDestination()
 {
  // After the enemy arrives at the
destination, destroy the current game
object
  GameObject.Destroy(gameObject);
  // The game failed
  TowerGameManager.instance.Failed();
 }

 // The enemy is destroyed
 void OnDestroy()
 {
  // When the enemy is destroyed, it
will survive the counter -1
  EnemySpawner.CountEnemyAlive--;
 }

 // Hurt
 public void TakeDamage(float damage)
 {
  if (hp <= 0)
  {
   return;
  }
  // Reduce blood volume and update UI
  hp -= damage;
  hpSlider.value = hp / totalHp;
  if (hp <= 0)
  {
   Die();
  }
 }

 // The enemy is dead
 void Die()
 {
  // enemy death explosion effects
  GameObject effect =
GameObject.Instantiate(explosionEffectP
refab, transform.position,
transform.rotation);
  // delay 1.5 seconds to destroy the
explosion effects
  Destroy(effect, 1.5f);
  // destroy the enemy
  Destroy(gameObject);
 }

}
```

Б.5.2. Лістинг скрипту «EnemySpawner.cs»

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

// Enemy incubator
public class EnemySpawner :
MonoBehaviour {

 public EnemyWave[] waves; // Each
element holds the desired attributes
for each wave of enemies, and how many
elements are generated
 public Transform START; // Generate
the enemy's position
 public float waveRate = 0.2f; // the
last wave of enemies dead and then
rebuild the waves of the enemy's time
interval
 public static int CountEnemyAlive = 0;
// the number of surviving enemies
 private Coroutine coroutine; //
Associations

 void Start()
 {
  coroutine =
StartCoroutine(SpawnEnemy());
 }

 // generate enemies
 IEnumerator SpawnEnemy()
 {
  foreach (EnemyWave wave in waves)
  {
   for (int i = 0; i < wave.count; i++)
   {
    // generate enemies

GameObject.Instantiate(wave.enemyPrefab
, START.position, Quaternion.identity);
    // Every time enemies make enemies
survive the counter +1, and when the
enemy is dead -1
    CountEnemyAlive++;
    // Every wave after the last enemy
does not need to be paused
    if (i != wave.count - 1)
    {
     // Each enemy generates an
interval
     yield return new
WaitForSeconds(wave.rate);
    }
   }
   while (CountEnemyAlive > 0)
   {
```

```
    // If there is no enemy dead, it
has been waiting
    yield return 0;
   }
   // the last wave of enemies dead and
then rebuild the waves of the enemy's
time interval
   yield return new
WaitForSeconds(waveRate);
  }

  while (CountEnemyAlive > 0)
  {
   yield return 0;
  }
  // no enemies survive, game victory
  TowerGameManager.instance.Win();
 }

 // stop generating enemies
 public void Stop()
 {
  StopCoroutine(coroutine);
 }

}
```

Б.5.3. Лістинг скрипту «EnemyWave.cs

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

// Save each wave of enemies to
generate the desired attributes
[System.Serializable] // can be
serialized, that is, it can be
displayed on the Inspector panel
public class EnemyWave {
 public GameObject enemyPrefab; //
enemy preforms
 public int count; // the number of
enemies
 public float rate; // Every enemy
generated interval
}
```

Б.5.4. Лістинг скрипту «EnemyWaypoints.cs»

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

// Enemy path point
public class EnemyWaypoints :
MonoBehaviour {

 public static Transform[] positions;
// all path points
 void Awake()
 {
  positions = new
Transform[transform.childCount];
  for (int i = 0; i < positions.Length;
i++)
  {
   positions[i] =
transform.GetChild(i);
  }
 }
}
```

Б.5.5. Лістинг скрипту «TowerCameraController.cs»

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

// 控制摄像机
public class TowerCameraController :
MonoBehaviour {

 public float translateSpeed = 25; //
angle of view movement speed
 public float scaleSpeed = 500; //
angle zoom speed

 void Update ()
 {
  // Direction buttons to control the
viewing angle before and after moving
  float h = Input.GetAxis("Horizontal")
* translateSpeed;
  float v = Input.GetAxis("Vertical") *
translateSpeed;
```

```
  // mouse pulley controls the
perspective of the distance
  float mouse = Input.GetAxis("Mouse
ScrollWheel") * scaleSpeed;

  // perspective in accordance with the
world coordinate system, so that it is
not affected by its own rotation
  transform.Translate(new Vector3(h,
mouse, v) * Time.deltaTime,
Space.World);
 }
}
```

Б.5.6. Лістинг скрипту «TowerGameManager.cs»

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

public class TowerGameManager :
MonoBehaviour {

 public GameObject endUI; // End UI
 public Text endMessage; // End of Game
Message
 public static TowerGameManager
instance;
 private EnemySpawner enemySpawner; //
enemy incubator

 void Awake()
 {
  instance = this;
  enemySpawner =
GetComponent<EnemySpawner>();
 }

 public void Win()
 {
  endUI.SetActive(true);
  endMessage.text = "Victory";
 }

 public void Failed()
 {
  endUI.SetActive(true);
  endMessage.text = "Lost";
  // stop generating enemies
  enemySpawner.Stop();
 }

 // Replay
 public void OnButtonRetryDown()
 {
  // reload the game scene

SceneManager.LoadScene(SceneManager.Get
ActiveScene().buildIndex,
LoadSceneMode.Single);
 }

 // menu
 public void OnButtonHomeDown()
 {
  // Load the menu scene
  SceneManager.LoadScene("Home",
LoadSceneMode.Single);
 }

}
```

Б.5.7. Лістинг скрипту «Turret.cs»

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

// turret script, control turret attack
public class Turret : MonoBehaviour {

 // the enemy in the attack range
 private List<GameObject> enemies = new
List<GameObject>();
 public float attackRateTime = 1.0f; //
attack interval
 private float timer = 0; // attack
interval
 public GameObject bulletPrefab; //
bullet preform
 public Transform firePosition; //
turret launch port position
 public Transform head; // turret head
 public bool isUseLaser = false; //
whether to use a laser turret
 public float damageRate = 70; // Laser
turret attack damage 1 sec 70 damager
 public LineRenderer laserRenderer; //
laser rendere
```

```
 public GameObject laserEffect; //
laser attack effects

 void Start()
 {
  // Turret just instantiation will be
able to start attack, so timer> =
attackRateTime set up
  timer = attackRateTime;
 }

 void Update()
 {
  // The turret is aimed at the enemy
  if (enemies.Count > 0 && enemies[0]
!= null)
  {
   Vector3 targetPosition =
enemies[0].transform.position;
   // Let the turret height remain the
same
   targetPosition.y = head.position.y;
   head.LookAt(targetPosition);
  }

  // Whether it is a laser turret
  if (isUseLaser)
  {
   if (enemies.Count > 0)
   {
    // If the target has been killed or
has reached the end, the collection is
removed
    if (enemies[0] == null)
    {
     UpdateEnemys();
    }
    // Clean up the empty element, once
again to determine whether there are
enemies can be attacked
    if (enemies.Count > 0)
    {
     if (laserRenderer.enabled ==
false)
     {
      laserRenderer.enabled = true;
      laserEffect.SetActive(true);
     }
     // Laser attack target
     laserRenderer.SetPositions(new
Vector3[]{firePosition.position,
enemies[0].transform.position});
     laserEffect.transform.position =
enemies[0].transform.position;
     laserEffect.transform.LookAt(new
Vector3(transform.position.x,
enemies[0].transform.position.y,
transform.position.z));
     // Causing sustained damage

enemies[0].GetComponent<Enemy>().TakeDa
mage(damageRate * Time.deltaTime);
    }
   }
   else
   {
    // Can attack state
    laserRenderer.enabled = false;
    laserEffect.SetActive(false);
   }

  }
  else
  {
   // Ordinary turret, timer increments
   timer += Time.deltaTime;
   // There is a place, and the timer
is reset over the attack interval, and
the attack method is called
   if (enemies.Count > 0 && timer >=
attackRateTime)
   {
    // The timer is cleared
    timer = 0;
    Attack();
   }
  }

 }

 // Into the attack range
 void OnTriggerEnter(Collider other)
 {
  // The enemy enters the attack range
and joins the collection
  if (other.tag == "Enemy")
  {
   enemies.Add(other.gameObject);
  }
 }

 // Leave the range of attacks - if the
turret range envelops the end, will not
remove the enemy
 void OnTriggerExit(Collider other)
 {
  // The enemy leaves the attack range
and removes the collection
  if (other.tag == "Enemy")
  {
   enemies.Remove(other.gameObject);
  }
 }

 // Attack the enemy
 void Attack()
 {
```

```
  // If the target has been killed or
has reached the end, the collection is
removed
  if (enemies[0] == null)
  {
   UpdateEnemys();
  }
  // Clean up the empty element, once
again to determine whether there are
enemies can be attacked
  if (enemies.Count > 0)
  {
   // The instantaneous bullets, bullet
positions and directions coincide at
the gun muzzle
   GameObject bullet =
GameObject.Instantiate(bulletPrefab,
firePosition.position,
firePosition.rotation);
   // Set the target to the bullet

bullet.GetComponent<TurretBullet>().Set
Target(enemies[0].transform);
  }
  else
  {
   // Can attack state
   timer = attackRateTime;
  }

 }
```

```
 // Update the enemy collection -
remove enemies that have been killed or
reached the finish line
 void UpdateEnemys()
 {
  // Store all empty elements
  List<int> emptyIndexList = new
List<int>();
  for (int i = 0; i < enemies.Count;
i++)
  {
   if (enemies[i] == null)
   {
    emptyIndexList.Add(i);
   }
  }
  // Remove empty elements
  for (int i = 0; i <
emptyIndexList.Count; i++)
  {
   enemies.RemoveAt(emptyIndexList[i] -
i);
  }
 }

}
```

Б.5.8. Лістинг скрипту «TurretBuildManager.cs»

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

// Construction turret management class
public class TurretBuildManager :
MonoBehaviour {

 public TurretData laserTurretData; //
Laser turret data
 public TurretData missileTurretData;
// Turret turret data
 public TurretData standardTurretData;
// Standard turret data
 private TurretData selectedTurretData;
// The currently selected turret data
will be built
 private TurretMapCube selectedMapCube;
// The currently selected turret is
located in the cube, and the turret
will show or hide the upgrade UI
```

```
 public Text moneyText; // Show the
text of money本
 public Animator moneyAnimator; //
Money animated state机
 private int money = 1000; // money
 public GameObject upgradeCanvas; //
Upgrade the canvas of the turret
 public Button upgradeButton; //
Upgrade by press
 public Animator upgradeCanvasAnimator;
// Turret upgrade canvas state machine

 // Money has changed
 void ChangeMoney(int change)
 {
  money += change;
  // Modify the money UI
  moneyText.text = "$ " + money;
 }

 void Update()
 {
  // Press the left mouse button
```

```
  if (Input.GetMouseButtonDown(0))
  {
   // Whether the mouse clicks on the
UI - if it is on the phone, you need to
determine the touch
   if
(EventSystem.current.IsPointerOverGameO
bject() == false)
   {
    // Emitting rays
    Ray ray =
Camera.main.ScreenPointToRay(Input.mous
ePosition);
    RaycastHit hit;
    // Ray detection, parameters: ray,
collision information, maximum
distance, detection layer. Return
whether to crash to
    bool isCollider =
Physics.Raycast(ray, out hit, 1000,
LayerMask.GetMask("MapCube"));
    if (isCollider)
    {
     // Get the click to the Cube
     TurretMapCube mapCube =
hit.collider.gameObject.GetComponent<Tu
rretMapCube>();
     // Has chosen a default turret
type, and the click of the location of
the turret has not yet been created
     if (selectedTurretData != null &&
mapCube.turretGo == null)
     {
      // If you click on the cube
without a turret, you can create it
      if (money >=
selectedTurretData.cost)
      {
       // Change in the number of money
       ChangeMoney(-
selectedTurretData.cost);
       // Create turret

mapCube.BuildTurret(selectedTurretData)
;
      }
      else
      {
       // TODO money is not enough,
give a hint

moneyAnimator.SetTrigger("Flicker");
      }
     }
     else if (mapCube.turretGo != null)
     {
      if (mapCube == selectedMapCube &&
upgradeCanvas.activeInHierarchy)
      {
       // Choose the same turret, and
the turret has been shown on the
upgrade UI
       StartCoroutine(HideUpgradeUI());
      }
      else
      {
       // There is already a turret,
passing turret location and whether it
has been upgraded

ShowUpgradeUI(mapCube.transform.positio
n, mapCube.isUpgraded);
      }
      // Record the currently selected
turret
      selectedMapCube = mapCube;
     }
    }
   }
  }
 }

 // Chose the laser turret
 public void OnLaserSelected(bool isOn)
 {
  if (isOn)
  {
   selectedTurretData =
laserTurretData;
  }
 }

 // Chose the turret turret
 public void OnMissileSelected(bool
isOn)
 {
  if (isOn)
  {
   selectedTurretData =
missileTurretData;
  }
 }

 // Chose the standard turret
 public void OnStandardSelected(bool
isOn)
 {
  if (isOn)
  {
   selectedTurretData =
standardTurretData;
  }
 }

 // Show turret upgrade UI
 void ShowUpgradeUI(Vector3 position,
bool isDisableUpgrade)
 {
```

```
  // Stop the last hidden animation - do not work用
  StopCoroutine(HideUpgradeUI());
  // Each activation is disabled first, so that the animation can be displayed properly
  upgradeCanvas.SetActive(false);

  // Activate turret upgrade UI
  upgradeCanvas.SetActive(true);
  // UpgradeCanvas audience only one object, each time to show him to set the location.
  upgradeCanvas.transform.position = position;
  // Whether the upgrade button is disabled or disabled if it is already upgraded or not enough money
  upgradeButton.interactable = !isDisableUpgrade;
 }

 // 隐藏炮塔升级UI
 IEnumerator HideUpgradeUI()
 {

upgradeCanvasAnimator.SetTrigger("Hide");
  // Disable the UI after 0.5 seconds
  yield return new WaitForSeconds(0.5f);
  // Hide the turret upgrade UI
  upgradeCanvas.SetActive(false);
 }
```

```
 // Click the upgrade button
 public void OnUpgradeButtonDown()
 {
  // If you click on the cube without a turret, you can create it
  if (money >= selectedMapCube.turretData.costUpgraded)
  {
   // Change in the number of money
   ChangeMoney(-selectedTurretData.costUpgraded);
   // Upgrade the turret on the cube
   selectedMapCube.UpgradeTurret();
   // Hide UI
   StartCoroutine(HideUpgradeUI());
  }
  else
  {
   // TODO money is not enough, give a hint
   moneyAnimator.SetTrigger("Flicker");
  }
 }

 // Click the Remove button
 public void OnDestroyButtonDown()
 {
  // Remove the turret on the cube
  selectedMapCube.DestroyTurret();
  // Hide UI
  StartCoroutine(HideUpgradeUI());
 }

}
```

Б.5.9. Лістинг скрипту «TurretBullet.cs»

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

// Bullet AI - instantiate out to attack the enemy
public class TurretBullet : MonoBehaviour {

 public int damage = 50; // Bullet damage
 public float speed = 40; // Bullet firing speed
 public GameObject explosionEffectPrefab; // The bullet hit the enemy's explosive effect of the preform
 private Transform target; // Attack target
```

```

 // Instructing bullets requires a given target
 public void SetTarget(Transform target)
 {
  this.target = target;
 }

 void Update()
 {
  // When the target reaches the finish line, or the target is killed. The bullet is destroyed
  if (target == null)
  {
   // Destroy the bullet
   Die();
   return;
```

```
  }
  // The bullet points to the target
  transform.LookAt(target.position);
  // Launch to attack target
  transform.Translate(Vector3.forward *
Time.deltaTime * speed);
 }

 // Bullet collision detection
 void OnTriggerEnter(Collider other)
 {
  // If the attack is the enemy
  if (other.tag == "Enemy")
  {
   // Let the enemy Diaoxie

other.GetComponent<Enemy>().TakeDamage(
damage);
   // Destroy the bullet
   Die();
  }
 }

 void Die()
 {
  // Explosive effect
  GameObject effect =
GameObject.Instantiate(explosionEffectP
refab, transform.position,
transform.rotation);
  // Destroy effects
  Destroy(effect, 1);
  // Destroy the bullet
  Destroy(gameObject);
 }

}
```

Б.5.10. Лістинг скрипту «TurretData.cs»

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

// Fort data category
[System.Serializable] // Serializable
public class TurretData {
 public GameObject turretPrefab; //
Basic version of the preform
 public int cost; // Basic Edition
price
 public GameObject
turretUpgradedPrefab; // Reinforced
preforms
 public int costUpgraded; // Upgrade
the price
}

// Turret type enumeration
public enum TurretType {
 LaserTurret, // Laser turret
 MissileTurret, // Turret turret
 StandardTurret // Turret turret
}
```

Б.5.11. Лістинг скрипту «TurretMapCube.cs»

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

// map cube, can be placed turret
public class TurretMapCube :
MonoBehaviour {

 [HideInInspector] // [HideInInspector]
can hide the public property displayed
in the inspector panel
 public GameObject turretGo; // The
turret under the current cube, if
empty, indicates that there is no
turret at the current location
 [HideInInspector] // [HideInInspector]
can hide the public property displayed
in the inspector panel
 public bool isUpgraded = false; //
Whether the turret has been upgraded
 public GameObject buildEffect; //
Build the turret's special effects
preform
 private Renderer cubeRenderer; //
Renderer
 public TurretData turretData; // the
turret data under the current cube

 void Start()
 {
  cubeRenderer =
GetComponent<Renderer>();
 }
```

```
 // Build turret
 public void BuildTurret(TurretData
turretData)
 {
  // let the current cube hold the
turret data to facilitate the upgrade
on the turret on the cube
  this.turretData = turretData;
  // Each build turret resets the
upgrade logo
  isUpgraded = false;
  // instantiate the turret
  turretGo =
GameObject.Instantiate(turretData.turre
tPrefab, transform.position,
Quaternion.identity);
  // Construction of turret dust
effects
  GameObject effect =
GameObject.Instantiate(buildEffect,
transform.position,
Quaternion.identity);
  Destroy(effect, 1.5f);
 }

 void OnMouseEnter()
 {
  // There is no turret in the current
position, the mouse is not on the UI.
Change the renderer color
  if (turretGo == null &&
!EventSystem.current.IsPointerOverGameO
bject())
  {
   cubeRenderer.material.color =
Color.red;
  }
 }

 void OnMouseExit()
 {
  // Restore the color after the mouse
is removed
  cubeRenderer.material.color =
Color.white;
 }

 // Upgrade the turret under the
current cube
 public void UpgradeTurret()
 {
  // Has been upgraded
  if (isUpgraded)
  {
   return;
  }

  Destroy(turretGo);
  // Modify the logo after upgrading
the turret
  isUpgraded = true;
  // Instantiate the enhanced version
of the turret
  turretGo =
GameObject.Instantiate(turretData.turre
tUpgradedPrefab, transform.position,
Quaternion.identity);
  // Upgrade the dust of the turret
  GameObject effect =
GameObject.Instantiate(buildEffect,
transform.position,
Quaternion.identity);
  Destroy(effect, 1.5f);
 }

 // Remove the turret
 public void DestroyTurret()
 {
  Destroy(turretGo);
  isUpgraded = false;
  turretGo = null;
  turretData = null;
 }
}
```

ДОДАТОК В
РОЗДАТКОВИЙ МАТЕРІАЛ

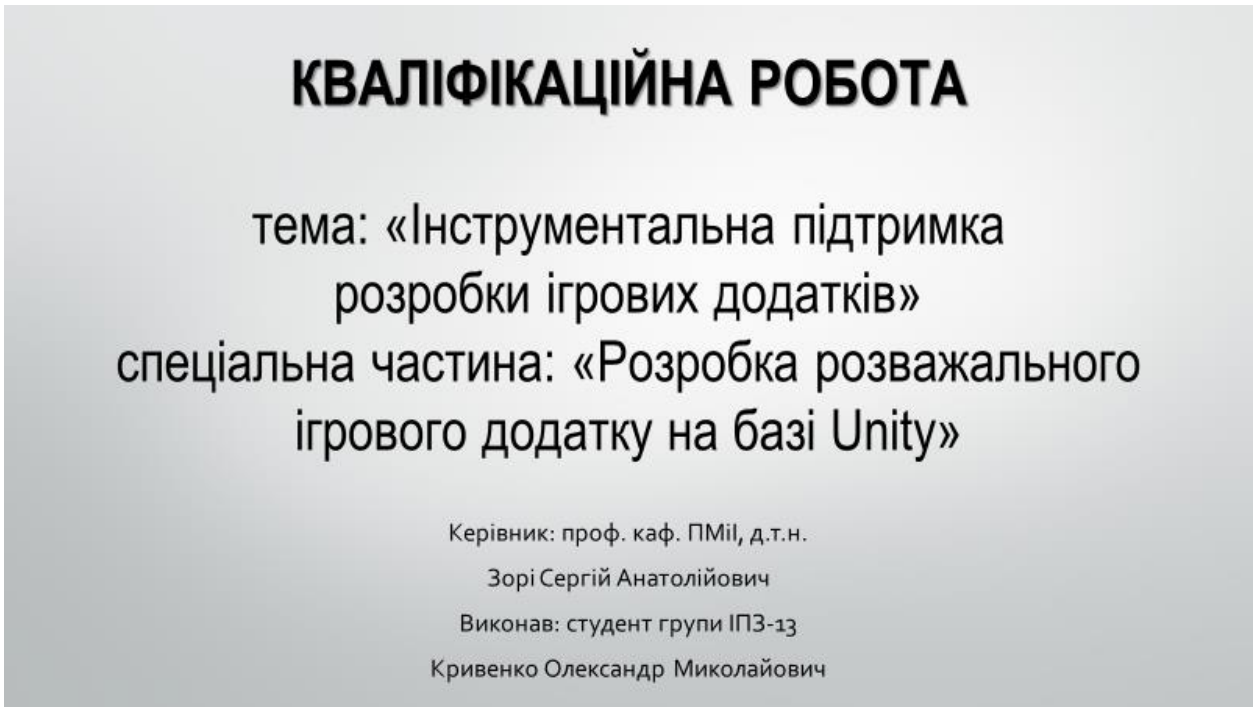


Рисунок В.1 – Титульний лист презентації

Рисунок В.2 – Загальні відомості про роботу

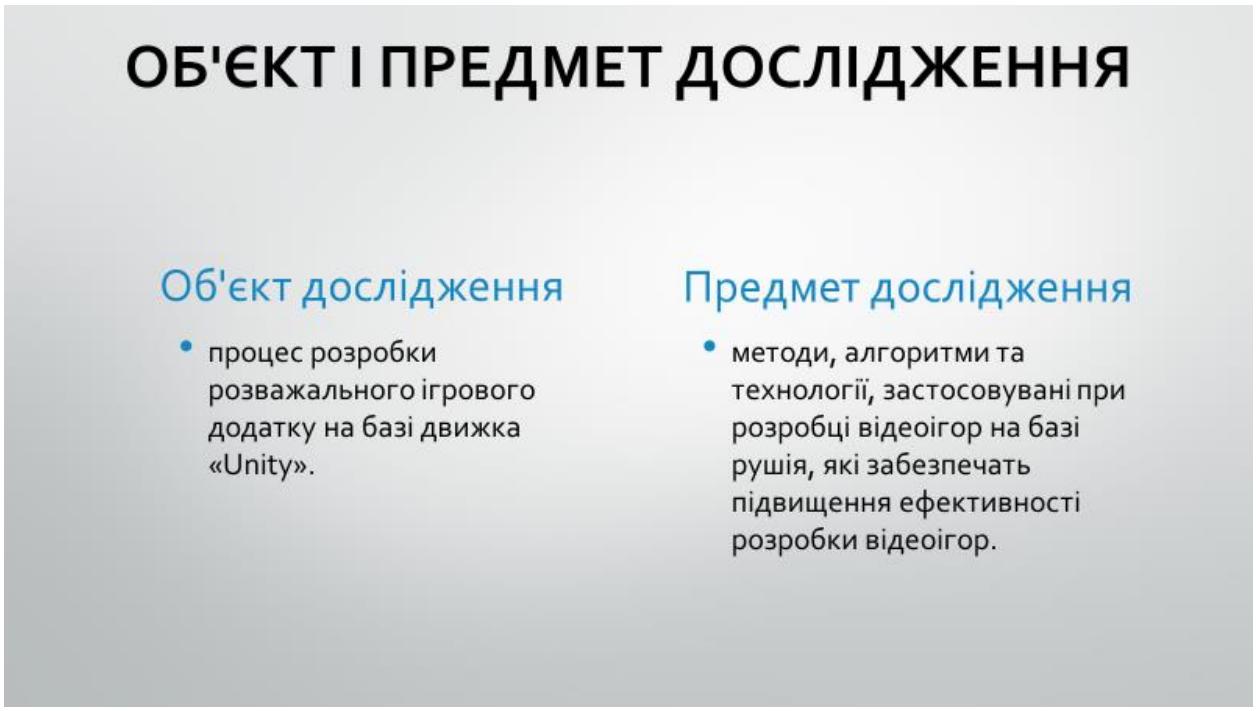


Рисунок В.3 – Об'єкт і предмет дослідження

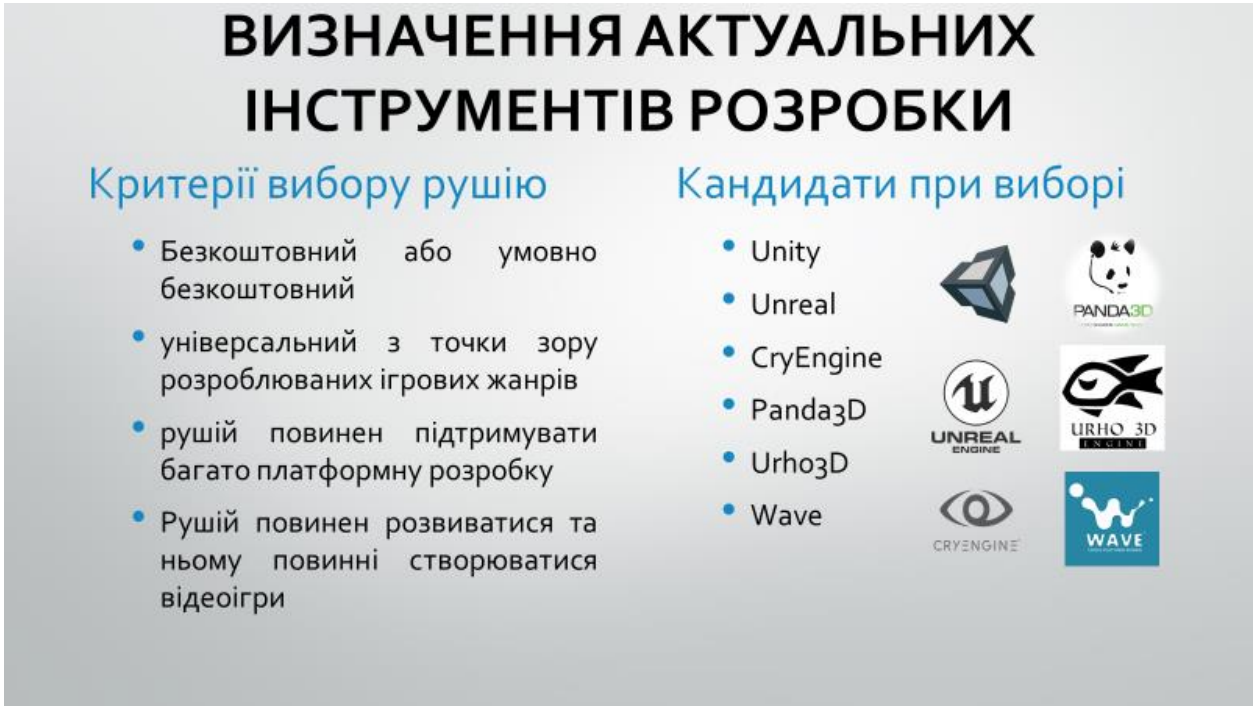


Рисунок В.4 – Визначення актуальних інструментів розробки

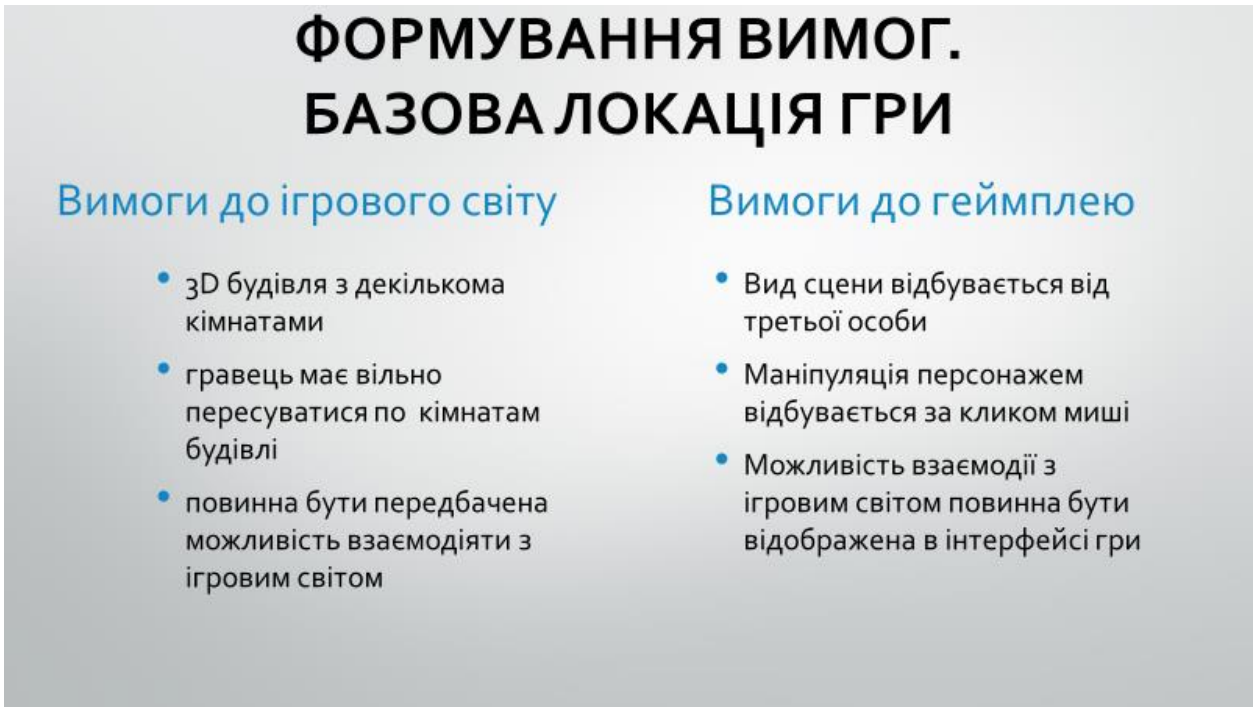


Рисунок В.5 – Формування вимог. Базова локація гри

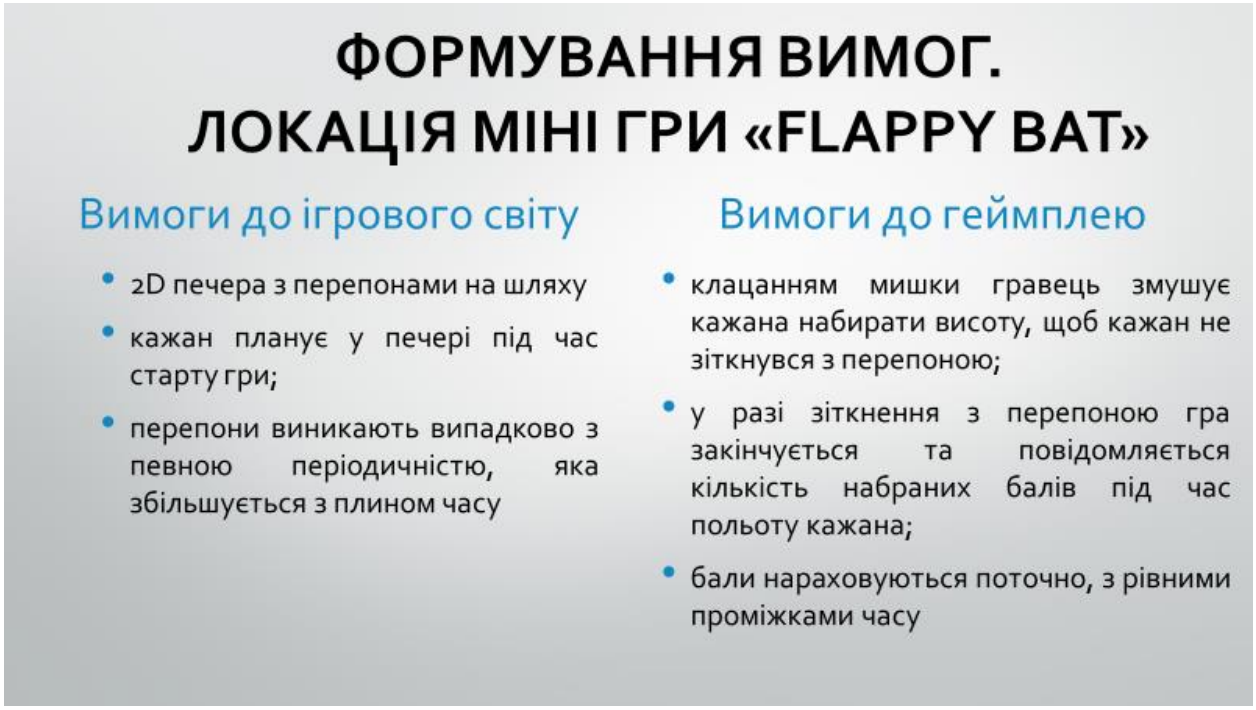


Рисунок В.6 – Формування вимог. Локація міні гри «FLAPPY BAT»

Рисунок В.7 – Формування вимог. Локація міні гри «TOWER DEFENSE»

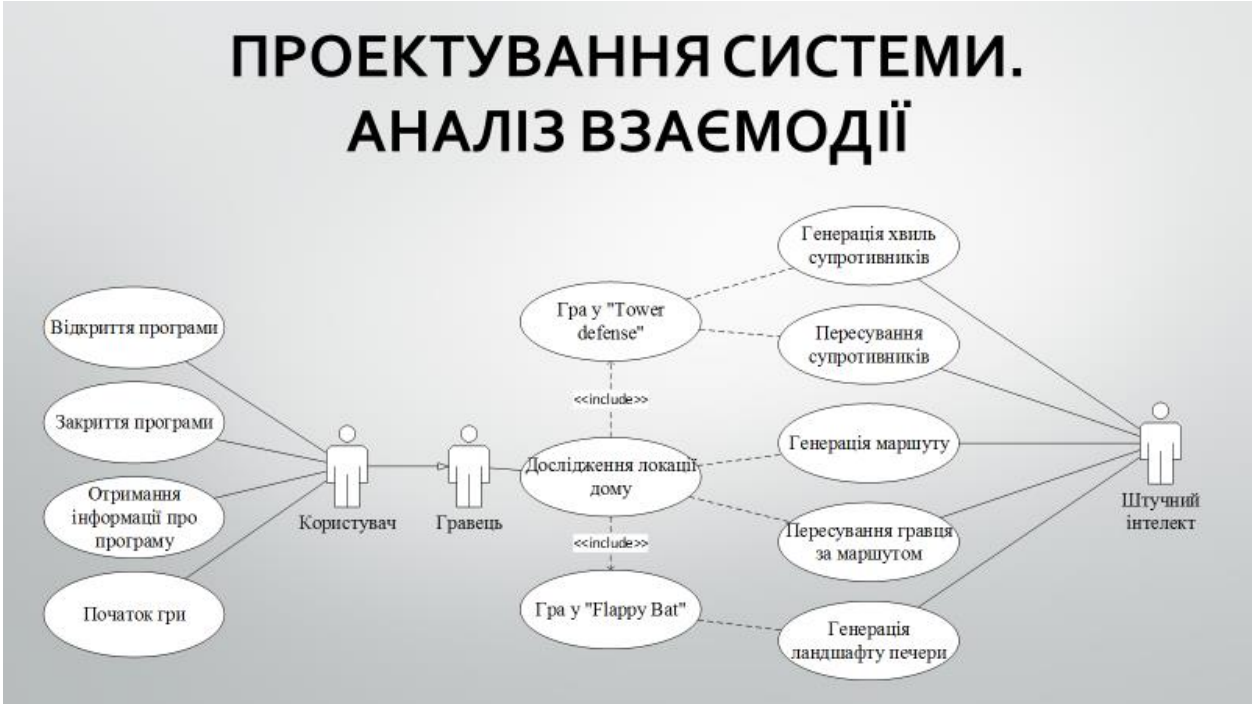


Рисунок В.8 – Проектування системи. Аналіз взаємодії

Рисунок В.9 – Структура програмної системи

Рисунок В.10 – Розробка системи. Структура проекту та ігрових сцен

Рисунок В.11 – Розробка системи. Навігація і пошук шляху

Рисунок В.12 – Тестування системи

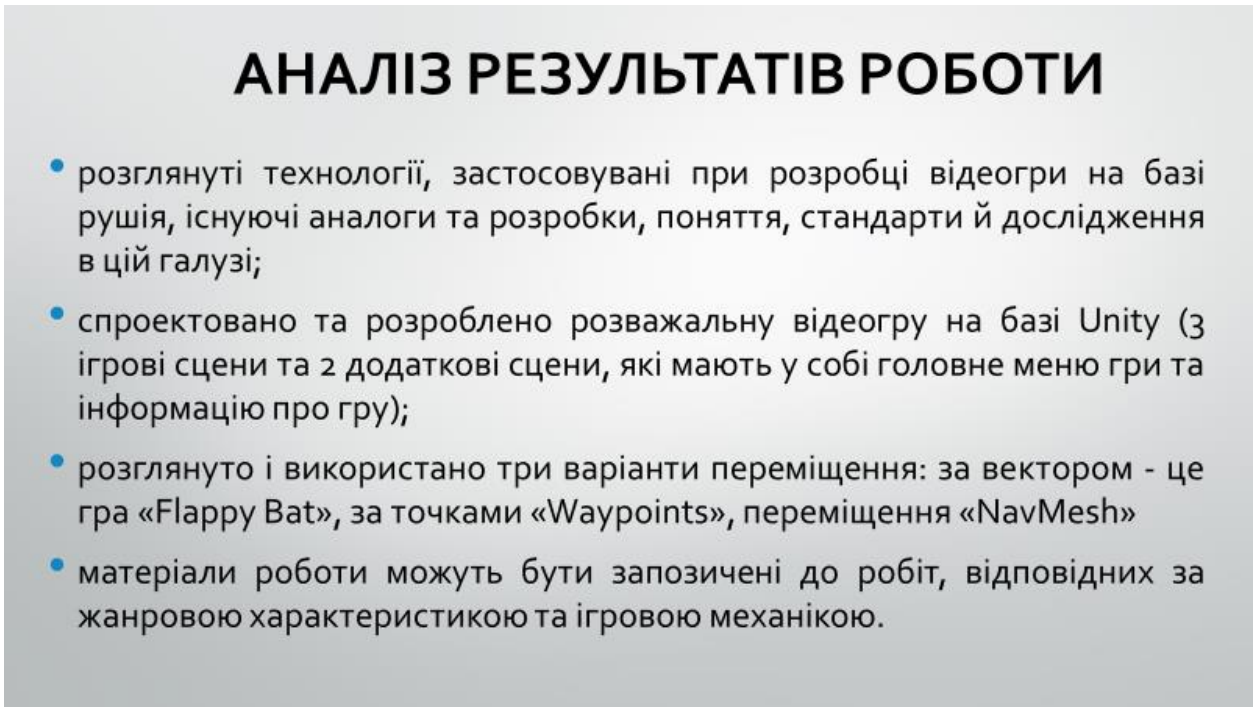


Рисунок В.13 – Аналіз результатів роботи

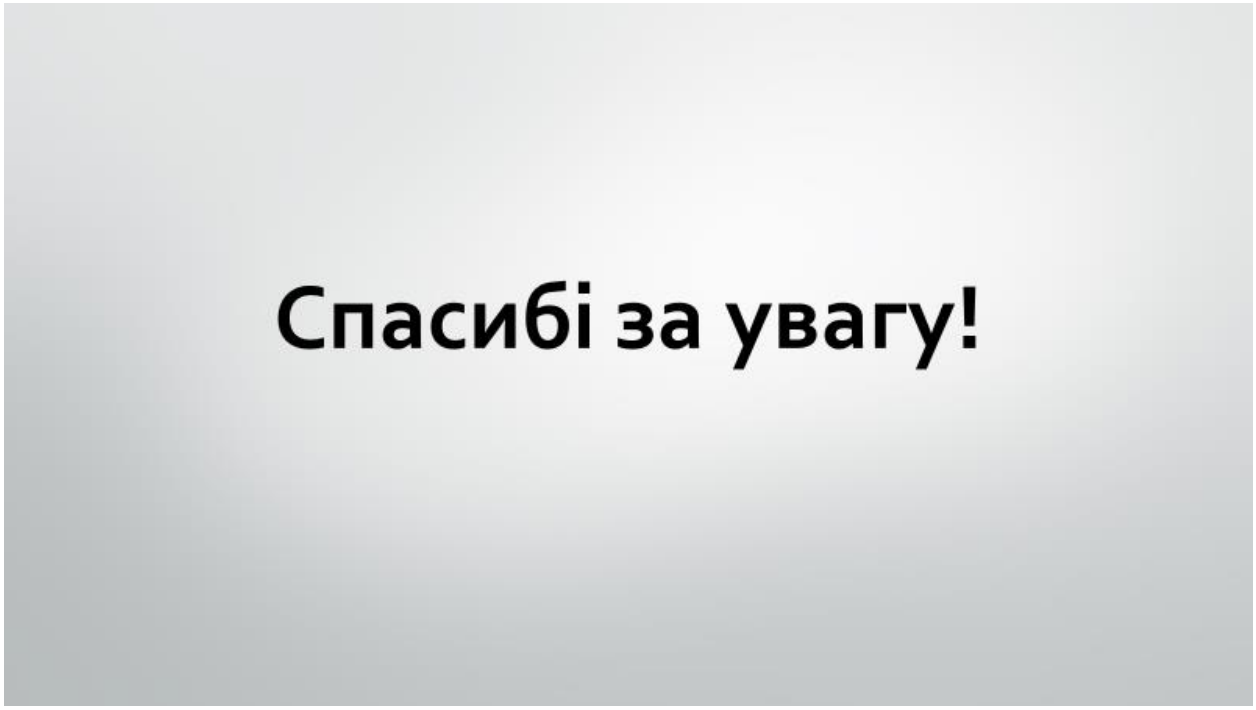


Рисунок В.14 – Завершення презентації