ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ

ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

ПОЯСНЮВАЛЬНА ЗАПИСКА

до бакалаврської роботи

на ступінь вищої освіти бакалавр

на тему: «**РОЗРОБКА ГРИ У ЖАНРІ TOWER DEFENSE З ВИКОРИСТАННЯМ ДВИГУНА UNITY НА МОБІЛЬНУ ПЛАТФОРМУ ANDROID**»

Виконав: студент 4 курсу, групи ПД – 42

спеціальності:

121 Інженерія програмного забезпечення

(шифр і назва спеціальності)

Мидинський О.І.

(прізвище та ініціали)

Керівник Дібрівний О.А.

(прізвище та ініціали)

Рецензент_____

(прізвище та ініціали)

Нормконтроль_____

(прізвище та ініціали)

Київ – 2022

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти - «Бакалавр»
Спеціальність - 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного
забезпечення
О.В. Негоденко
«____»_____ 2022 року

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

Мидинському Олегу Ігоровичу

(прізвище, ім'я, по батькові)

1. Тема роботи: «Розробка гри у жанрі Tower Defense з використанням двигуна Unity на мобільну платформу Android»
   Керівник роботи доктор філософії Дібрівний О.А.
   (прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
   затверджені наказом вищого навчального закладу від «18» лютого 2022 року №65.
2. Строк подання студентом роботи 03.06.2022
3. Вихідні данні до роботи:
   3.1. Положення побудови гри;
   3.2. Методи побудови гри;
   3.3. Розробка моделі гри;
   3.4. Науково-технічна література.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно зробити):
   4.1. Загальні положення побудови гри;
   4.2. Аналіз технології і методів побудови гри;
   4.3. Висновки
5. Перелік графічного матеріалу:
   5.1. Основні характеристики роботи;
   5.2. Актуальність задачі;
   5.3. Реалізація алгоритмів гри

5.4. Висновки

6. Дата видачі завдання 11.04.2022

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір джерел інформації	11.04.2022	Виконано
2	Вимоги до встановленого додатку	23.04.2022	Виконано
3	Оцінка якості тестування до системи	29.04.2022	Виконано
4	Концепція та архітектура додатку	02.05.2022	Виконано
5	Вступ, висновки, реферат	05.05.2022	Виконано
6	Розробка презентації	06.05.2022	Виконано
7	Попередній захист	16.05.2022	
8	Здача роботи	03.06.2022	

Студент _____ Мидинський О.І.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Дібрівний О.А.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина бакалаврської роботи: 66 стр., 4 рис., 1 дод., 18 джерел.

UNITY, C#, TOWER DEFENSE, ГРА, ВІДЕОІГРИ, ANDROID.

Об’єкт дослідження – створення тривимірного ігрового додатку у жанрі «tower defense» для Android.

Предмет дослідження – методи, патерни та технології розробки ігрового програмного забезпечення для Android платформи.

Мета роботи – спроектувати та розробити мобільний ігровий додаток на базі Android платформи.

Методи дослідження – методи створення ігрового мобільного додатку та опис розроблених ігрових механік для більш ефективної взаємодії їх один з одним.

Для досягнення поставленої мети було вирішено наступні завдання:

1. Провести аналіз предметної області.
2. Дослідити рушії розробки ігор, мови програмування, необхідні для них, та середовища розробки програмного коду.
3. Розробити архітектуру ігрового мобільного додатку.
4. Створити програмний продукт.
5. Провести тестування.

Предметом дослідження є гра на платформу Android.

Практичне значення отриманих результатів полягає у розробленому програмному ігровому продукті у жанрі «tower defense» на ігровому рушії Unity та на мові програмування C# на мобільну платформу. Таку гру можна поширити, виклавши її в магазині мобільних ігор.

Сфера застосування – будь який користувач Android-смартфону може встановити додаток та пограти в нього.

ЗМІСТ

ВСТУП **9**
1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ **11**
1.1 Відеоігри 11
1.1.1 Загальні відомості 11
1.1.2 Історія розвитку 11
1.1.3 Розробка відеоігор 13
1.2 Tower defense 15
1.2.1 Загальні відомості 15
1.2.2 Виникнення та розвиток жанру tower defense 16
1.2.3 Ігровий процес 19
1.3 Мобільні ігри 21
2 ВИБІР ЗАСОБІВ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ **23**
2.1 Ігрові рушії 23
2.1.1 Unreal Engine 23
2.1.2 Unity 24
2.1.3 Порівняльна характеристика 27
2.2 Мова програмування C# 28
2.3 Вибір середовища програмування 31
2.3.1 IntelliJ IDEA 31
2.3.2 Visual Studio Code 32
2.3.3 Visual Studio 34
2.4 Особливості архітектури додатку на Unity 35
3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ **37**
3.1 Файлова структура логіки програми 37
3.2 Логіка рівня 37
3.3 Object Pooling 38
3.4 Скриптинг ворогів 40
3.5 Логіка оборонних веж та головної будівлі 40
3.6 Нанесення пошкоджень ворогам 41

3.7 Скриптування інтерфейсу користувача.....42
3.8 Тестування розробленого програмного продукту.....44
ВИСНОВКИ 45
ПЕРЕЛІК ПОСИЛАНЬ.....47
ДОДАТОК А.....49
ДОДАТОК Б.....62

ВСТУП

Актуальність теми. В даний час ринок мобільних ігор не стоїть на місці та активно розвивається. За оцінками Newzoo, до кінця 2020 року дохід від мобільних ігор досяг 76,1 мільярда доларів [1], що на 12% більше, ніж у 2019 році. Для порівняння: глобальний дохід від касових зборів становив 42,5 мільярда доларів. У 2019 році світова музична індустрія принесла оптовий виторг у розмірі 20,2 мільярда доларів. Іншими словами, до кінця 2020 року тільки мобільні ігри будуть принесли більше річного доходу, ніж музична і кіноіндустрія разом узяті.

Однак якість більшості ігор, що випускаються, залишає бажати кращого, і користувачі вибирають саме якісні ігри. Але не лише якість приваблює користувачів, а також новизна, нові ігрові механіки чи нові поєднання ігрових механік. Під новизною розуміється новий ігровий контент, якщо користувач проходить одну з ігор і йому подобаються ігри даного жанру, він з великою ймовірністю буде шукати подібну гру, але більш якісну і популярну, з хорошими відгуками інших користувачів.

Для програміста у команді розробників постає питання: як прискорити процес розробки з його боку? Для цього необхідно розробити архітектуру програми, яка буде здебільшого незмінна і лише покращуватиметься з кожним новим проєктом. При розширенні команди, для нових програмістів не повинно виникати труднощів з архітектурою проєкту, тому код має бути підтримуваним і розширюваним. Архітектура проєкту має бути правильно організована та структурована, що дозволить спростити роботу з великим обсягом коду, якщо за проєкт наважиться взятися інший програміст. Для вирішення цього завдання необхідно вибрати шаблони проектування, причому без урахування поточного проєкту, зробивши архітектуру якомога абстрактнішою, щоб отримати модель, що підходить для всіх ігрових проєктів.

Об’єктом дослідження є створення тривимірного ігрового додатку у жанрі «tower defense» для Android.

Предметом дослідження є методи, патерни та технології розробки ігрового програмного забезпечення для Android платформи.

Метою є виявлення недоліків та проблем при проектуванні та розробці мобільного ігрового додатку на базі Android платформи.

Завдання дослідження, які необхідно виконати для досягнення поставленої мети:

1. Провести аналіз предметної області.
2. Дослідити рушії розробки ігор, мови програмування, необхідні для них, та середовища розробки програмного коду.
3. Розробити архітектура ігрового мобільного додатку.
4. Створити програмний продукт.
5. Провести тестування.

Методика дослідження. Під час розробки дипломної роботи використовувались інструменти (Unity3d, Visual Studio, C#) та патерни (SOLID, Singleton, Object Pooling, DRY та ін.) створення ігрового мобільного додатку.

Наукова новизна полягає в наступному: створення великого, комплексного, оптимізованого програмного продукту з можливістю подальшого його розширення.

Практична значущість результатів полягає у розробленому програмному ігровому продукті у на ігровому рушії Unity та на мові програмування C# на мобільну платформу. Теоретичний матеріал, отриманий в результаті роботи, може бути використаний в наступних проєктах.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Відеоігри

1.1.1 Загальні відомості

Відео або комп’ютерна гра — це електронна гра, яка передбачає взаємодію з інтерфейсом користувача або пристроєм введення (джойстиком, контролером, клавіатурою, пристроєм для визначення руху тощо) і створює візуальний зворотний зв’язок. Цей відгук відображається на пристроях відображення відео, таких як телевізори, монітори, сенсорні екрани та гарнітури віртуальної реальності. Відеоігри часто підкріплюються аудіо зворотнім зв’язком, що передається через динаміки та навушники, і може бути доповнений іншими типами зворотного зв’язку, включаючи тактильні прийоми. Комп’ютерні ігри – це не всі відеоігри. Наприклад, текстові пригодницькі ігри та комп’ютерні шахи не обов’язково покладаються на графічні дисплеї.

Відеоігри визначаються на основі платформ, які включають аркадні відеоігри, консольні ігри та ігри для персональних комп’ютерів (ПК). Останнім часом галузь поширилася на смартфони та планшетні комп’ютери, системи віртуальної та доповненої реальності та мобільні ігри через віддалені хмарні ігри. Відеоігри поділяються на широкий діапазон жанрів залежно від типу та мети ігрового процесу.

1.1.2 Історія розвитку

Ранні відеоігри використовували інтерактивні електронні пристрої в різних форматах відображення. Найдавніший приклад датується 1947 роком. «Пристрій для розваг із електронно-променевою трубкою» був поданий на патент Томасом Т. Голдсмітом-молодшим і Естрелейманом 25 січня 1947 року і був виданий як патент США 2455992 14 грудня 1948 року [2]. Натхненний технологією радіолокаційного

відображення, аналоговий пристрій, який дозволяє користувачеві керувати параболічною дугою точок на екрані, щоб імітувати ракету, запущену по цілі, яка являє собою паперовий малюнок, закріплений на екрані. Інші ранні приклади включають проєкт гри Крістофера Стрейчі, Nimrod Computer на Британському фестивалі 1951 року. OXO, хрестики-нолики, комп'ютерна гра Олександра С. Дугласа для EDSAC 1952 року. Теніс для двох, електронна інтерактивна гра, розроблена Вільямом Хігінботамом у 1958 році. Spacewar!, створена студентами Массачусетського технологічного інституту Мартіном Грецем, Стівом Расселом і Уейном Вітаненом на комп'ютері DEC PDP-1 в 1961 році. Для кожної гри спосіб відображення відрізняється. NIMROD має панель підсвічування для ігор Nim, OXO має графічний дисплей для гри в хрестики-нулики. Tennis for Two має осцилограф для перегляду тенісного корту збоку. Spacewar! має векторне представлення DEC PDP-1 для двох космічних кораблів для боротьби один з одним.

Ці попередні винаходи проклали шлях до витоків сучасних відеоігор. Ральф Г. Баєр розробив систему керування для гри в настільний теніс на телевізійному екрані, працюючи в Sanders Associates у 1966 році. Зі схвалення компанії Баєр створив прототип «Brown box». Sanders запатентував винахід Баєра і ліцензував його в Magnavox, який комерціалізували як Magnavox Odyssey - першу домашню ігрову консоль, випущену в 1972 році. Крім того, Нолан Бушнелл і Тед Дабні були натхненні, побачивши Spacewar!, що працює в Стенфордському університеті, розробили подібну версію, яка працює в меншому монетному аркадному автоматі за допомогою менш дорогого комп'ютера. Вона була випущена в 1971 році як Computer Space, перша аркадна відеогра. Бушнелл і Дебні заснували Atari, Inc. і працювали з Алланом Алкорном над створенням другої аркадної гри в 1972 році. Це гра у стилі Pong, натхнена безпосередньо грою в настільний теніс Odyssey. Сандерс і Magnavox подали до суду на Atari за порушення патенту Бера, але Atari оплатила постійні права на патент і вирішила справу у позасудовому порядку. Після їхньої домовленості Atari створила домашню версію Pong, яка була випущена до Різдва 1975 року. Успіх

Odyssey і Pong як аркадних ігор і домашніх автоматів започаткував індустрію відеоігор. І Бера, і Бушнелла називають «батьками відеоігор» через їхній внесок.

У 2000-х роках основна індустрія оберталася навколо «ААА» ігор, залишаючи мало місця для експериментальних ігор із високим ризиком. У поєднанні з доступністю Інтернету та цифрового розповсюдження це забезпечило простір для незалежної розробки відеоігор (або інді-ігор) і привернуло увагу в 2010-х роках. Відтоді комерційне значення індустрії відеоігор зросло. Азіатські ринки, що розвиваються, і мобільні ігри на смартфонах, зокрема, стимулюють монетизацію, перетворюючи демографічні дані гравців на звичайні ігри та включаючи ігри як послуги. За оцінками, станом на 2020 рік світовий ринок відеоігор генерує 159 мільярдів доларів США щорічного доходу від обладнання, програмного забезпечення та послуг в цілому. Це втричі більше світової музичної індустрії в 2019 році і в чотири рази більше, ніж кіноіндустрія в 2019 році [3].

1.1.3 Розробка відеоігор

Розробка та авторство відеоігор, як і інших видів розваг, часто є міждисциплінарною сферою. Команда розробників відеоігор в основному складається з програмістів і графічних дизайнерів. З роками цей перелік розширився, охоплюючи майже всі типи навичок, які зазвичай можна знайти у виробництві фільмів і телевізійних шоу, включаючи звукорежисерів, музикантів та інших технічних спеціалістів. Існують також навички, характерні лише для відеоігор, таких як ігровий дизайнер. Усім цим керує продюсер.

На перших порах індустрії було звично, коли одна людина керувала всіма ролями, необхідними для створення відеоігри. Оскільки типи матеріалів, які може представляти платформа, ставали комплекснішими та потужнішими, потрібні були більші команди для створення всього, від графіки до програмування та сюжету. Це не означає, що ера маленьких компаній минула, оскільки все ще іноді зустрічаються такі представники у виробництві казуальних ігор та на ринку портативних пристроїв [4],

де невеликі ігри стають все більш популярними, через технічні обмеження, такі як оперативна пам’яті та функції візуалізації 3D-графіки [5].

Відеоігри програмуються, як і будь-яке інше комп’ютерне програмне забезпечення. До середини 1970-х років аркади та домашні ігрові консолі програмувалися шляхом збирання окремих електромеханічних компонентів на друкованій платі. Це обмежувало гру відносно простою логікою. До 1975 року велика кількість недорогих мікропроцесорів можна було використовувати для обладнання відеоігор, що дозволило розробникам ігор програмувати більш детальні ігри, розширюючи діапазон можливостей. Постійні вдосконалення технології комп’ютерного обладнання розширюють можливості створення відеоігор, а інтеграція загального обладнання між консолями, комп’ютерами та аркадними платформами стимулює процес розробки [6]. Сьогодні розробники ігор мають ряд комерційних і відкритих інструментів, які вони можуть використовувати для створення ігор. Ці інструменти часто охоплюють кілька платформ для підтримки портативності. Сьогодні багато ігор побудовано на ігровому движку, який обробляє більшу частину логіки гри, ігрового процесу та графіки. Ці механізми можна розширити за допомогою механізмів, які є специфічними для певної функції, наприклад, фізичним двигуном, який імітує фізику об’єкта в реальному часі. Існує цілий ряд проміжного програмного забезпечення, яке допомагає розробникам отримати доступ до інших функцій, таких як відтворення відео в грі, ігровий мережевий код, який спілкується через онлайн-сервіси, підбір партнерів в онлайн-іграх та подібні функції. Ці функції доступні з мови програмування розробника на ваш вибір. Крім того, ви можете використовувати набір для розробки ігор, який обмежує кількість налаштувань, які ви можете додати до своєї гри, мінімізуючи кількість прямого програмування. Як і будь-яке програмне забезпечення, відеоігри зазвичай перевіряються на якість перед випуском, щоб переконатися, що продукт не містить помилок і дефектів, але розробники часто випускають виправлення та оновлення.

ЗІ збільшенням команди розробників, зростають і кошти на виробництво. Студії розробки потребують найкращих талантів, але видавці скорочують витрати, щоб зберегти свої інвестиції прибутковими. Групи розробників відеоігор на консолі зазвичай налічують від 5 до 50 осіб, інколи і більше 100 чоловік. Повідомлялося, що в травні 2009 року в Assassin's Creed II було 450 розробників [7]. Зростаючий розмір команди в поєднанні з великим тиском на те, щоб вивести завершені проєкти на ринок і почати відшкодовувати витрати на виробництво, призводить до переносу дедлайнів, спішки та випуску незавершених продуктів.

Програмування ігор аматорами і ентузіастами існує з кінця 1970-х років з появою домашніх комп'ютерів, в 2000-х роках з'явився тренд розробки інді-ігор. Інді-ігри створюють невеликі команди, які видавці не можуть безпосередньо контролювати. Обсяг гри менший, ніж у великих ігрових «ААА» студіях, і часто експериментують з ігровим процесом та художнім стилем. Розробка інді-ігор підтримується більшою доступністю цифрового розповсюдження, включаючи нові маркери мобільних ігор, а також готові недорогі інструменти розробки для цих платформ.

1.2 Tower defense

1.2.1 Загальні відомості

Tower Defense (TD) — це піджанр стратегічних ігор, спрямований на захист території та володінь гравця, блокуючи ворожих нападників і не даючи їм дістатися до виходу. Зазвичай оборонна споруда розміщується на або вздовж шляху атаки. В основному це різноманітні структури, які допомагають автоматично блокувати, саботувати, атакувати або знищувати ворогів. Tower defense вважається піджанром стратегічних відеоігор у реальному часі через його походження, але багато сучасних

ігор Tower Defense включають аспект покрокової стратегії. Суть гри в цьому жанрі є стратегічний підбір і розміщення захисних елементів.

1.2.2 Виникнення та розвиток жанру tower defense

Зародження жанру tower defense можна простежити в золоту добу аркадних відеоігор у 1980-х роках. Метою Space Invader, аркадної гри, випущеної в 1978 році, був захист території гравця (розміщеної в нижній частині екрана) від ворожих хвиль. У грі був щит, який можна було використовувати, щоб стратегічно зупинити атаки ворога на гравця та допомогти йому захистити територію, але явно територію не захищав. Команда Game Missile 1980 року змінила це, надавши щиту більш стратегічну роль. Гравці могли блокувати вхідні ракети, і кожна хвиля атаки мала кілька напрямків атаки. Missile Command була також першою у своєму роді. Гра використовувала трекбол, вказівний пристрій, який дозволяє гравцям використовувати перехрестя. Інновації випереджала свій час і очікувала подальшого буму в цьому жанрі, пов’язаного з широким поширенням комп’ютерних мишок. Крім того, в Missile Command єдиною метою нападника є база, а не конкретний персонаж гравця. З цих причин деякі вважають її першою справжньою грою в цьому жанрі [8].

Вважається, що випущений у 1990 році Rampart створив типову оборонну башту. Rampart представив захист, який розміщується гравцем та який автоматично атакує наступаючих ворогів. Крім того, є чіткі етапи будівництва, захисту та ремонту. Наразі це основні елементи ігрового процесу багатьох ігор цього жанру. Також це була одна з перших багатокористувацьких відеоігор такого роду.

Геймплей tower defense також доступний на консолях у кількох міні-іграх серії Final Fantasy, включаючи міні-ігри tower defense у Final Fantasy VI (1994) та міні-ігри Fort Condor у Final Fantasy VII (1997), яка була одиною з перших, хто має 3D-графіку. Dungeon Keeper (1997) пропонує гравцям захищати серце підземелля, гігантську коштовність у центрі підземелля. У разі знищення гравець програє гру. Центральною

темою 3D-шутера від першої особи Turok2: Seeds of Evil (1998) був захист енергетичного тотема від орд зловмисників. Fortress була випущена для Game Boy Advance в 2001 році.

Оскільки стратегічні ігри в режимі реального часу стали більш популярними в іграх для ПК, багато хто впровадив режим tower defense в ігровий процес, особливо в багатокористувацьких іграх. Вперше створені в лютому 2006 року за допомогою World Editor Warcraft III, спеціальні карти Warcraft III (2002), Element TD і Gem Tower Defense відновили цей жанр практично руками однієї людини. Ці представники також вперше вносять в цей жанр рольовий елемент.

З 2007 по 2008 рік цей жанр став феноменом, частково через популярність режиму tower defense в стратегічних іграх в реальному часі, але в основному з появою незалежних розробників Adobe Flash, Apple і Google. Перша самостійна браузерна гра з'явилася в 2007 році. Серед них дуже популярний Flash Element Tower Defense, випущений у січні, Desktop Tower Defense, випущений у березні, і Antbuster, випущений у травні. Desktop Tower Defense отримав нагороду Independent Games Festival Award [9], і її успіх призвів до версії, створеної для мобільних телефонів іншим розробником. Також важливим проєктом Flash, випущеним в 2008 році, був GemCraft. Портативні ігрові консолі не були проігноровані під час буму. У вересні та жовтні були випущені Lock's Quest і Ninjatown відповідно. Bloons Tower Defense вперше було опубліковано в 2007 році. Це одна з багатьох випущених мультиплатформних ігор на тему повітряних куль.

Успіх цього жанру також призвів до нових випусків на ПК та ігрових консолях. Популярні ігри 2008 року включають PixelJunk Monsters, випущену в січні, Defense Grid: The Awakening і Savage Moon в грудні. У 2008 році також були випущені geoDefense Swarm, geoDefense, GemCraft, Savage Moon, Fieldrunners, Harvest: Massive Encounter і Crystal Defenders. GauntNet вийшов у квітні 2009 року. Випущена в травні 2009 року, Plants vs. Zombies була ще однією дуже популярною грою, яка стала успішною серією на мобільних пристроях. Також того року були випущені Sentinel,

TowerMadness, Babel Rising, Creeper World, Sol Survivor, Comet Crash, Final Fantasy Crystal Chronicles: My Life as Dark Lord, South Park Let's Go Tower Defense Play!, Starship Patrol та Trenches.

Наприкінці буму більшість ігор Tower Defense все ще застрягли в графічному середовищі із боковою прокруткою, ізометрією або перспективою зверху вниз. Iron Grip: Warlord, випущений у листопаді 2008 року, невдало став першопрохідцем, створивши шутер від першої особи в цьому жанрі. Неприємне поєднання експериментальної механіки захисту вежі та 3D-графіки не було дуже добре прийнято, але пізніші представники вдосконалили її виконання та проклали шлях для нового покоління популярних ігор. Dungeon Defenders, випущена в жовтні 2010 року, була однією з перших ігор tower defense, яка перенесла цей жанр в перспективу третьої особи. За перші два тижні після випуску було продано понад 250 000 копій [10], а до кінця 2011 року було продано понад 600 000 копій [11]. У 2010 році були випущені SteamWorld Tower Defense, Protect Me Knight, The Tales of Bearsworth Manor, Revenge of the Titans, Arrow of Laputa, Toy Soldiers, Robocalypse: Beaver Defense.

Гра Sanctum 2011 року та її продовження 2013 року популяризували гібриди шутерів від першої особи, започатковані цими ранніми іграми. В Orcs Must Die! також інтегрований жанр FPS у повне 3D-середовище та створено кілька сиквелів. Anomaly: Warzone Earth, випущена в 2011 році, представила варіації гри під назвою «Reverse Tower Defense» «Tower Attack» та «Tower Offense». У грі необхідно атакувати базу противника, захищену численними оборонними спорудами. Продовження та інші ігри з тих пір експериментували в обох стилях tower defense. Tiny Heroes, Army of Darkness: Defense, Iron Brigade, Rock of Ages, Trends 2 також були випущені в 2011 році.

Defender's Quest, Bad Hotel, Toy Defense, Strikefleet Omega, Unstoppable Gorg, Defenders of Ardania, Orcs Must Die! 2, Fieldrunners 2, Dillon's Rolling Western, Oil Rush і Elf Defense Eng з’явилися в 2012 році. Приблизно в цей час жанр дозрів і був визнаний чітким піджанром стратегічних ігор і повернувся з численними оновленими версіями. Chain Chronicle і Castle Storm були випущені в 2013 році. У 2014 році було

представлено ряд нових ігор, зокрема Space Run, Dungeon of the Endless, Island Days, Final Horizon, The Battle Cats, Age of Empires: Castle Siege, Defense Grid 2, TowerMadness 2. Deathtrap, Krinkle Krusher.

Сучасніші представники в цьому жанрі включають Rock of Ages 2: Bigger & Boulder (2017) і Orcs Must Die! Unchained (2017), Dillon's Dead-Heat Breakers (2018), Eden Rising: Supremacy (2018), Aegis Defenders (2018), Arknights (2019), Taur (2020), Element TD 2 (2020), Scuffed Tower Defense (2021).

З появою програм соціальних мереж, таких як платформа Facebook, tower defense став популярним жанром, а такі ігри, як Bloons TD і Plants vs. Zombies Adventures, перейшли на покрокову систему. Останні релізи включають Star Fox Guard і McDroid, які вийшти в 2016 році.

1.2.3 Ігровий процес

Основними елементами ігрового процесу tower defense є:

- території або володіння (разом «бази»), які повинен захищати гравець або гравці;
- кілька наступаючих «ворожих» хвиль для захисту;
- розташування елементів «веж» та перешкоди на шляху атакуючого противника.

Tower defense відрізняється від інших ігор із захистом бази здатність гравця стратегічно розташовувати, будувати або викликати перешкоди на шляху атаки ворога. Головний герой часто безсмертний, оскільки головна задача — зберегти базі, а не вижити гравцеві.

Деякі особливості сучасних tower defense:

- перешкоди, встановлені гравцем, які можуть пошкодити або вбити ворожих нападників, перш ніж вони зруйнують базу;
- здатність усувати перешкоди;

- можливість покращувати перешкоди;
- можливість ремонту покращень до перешкод;
- певні валюти для придбання покращень і ремонтів, якими може бути час, ігрова валюта або бали досвіду, наприклад, отримані за поразку атакуючого підрозділу;
- вороги можуть пройти кількома напрямками одночасно;
- кожна хвиля зазвичай має певну кількість і тип ворогів;
- розблокування нових карти та рівні;
- вежа здатна рухатися.

Багато сучасних ігор tower defense еволюціонують від реального часу до покрокових, циклічно проходячи через окремі фази гри, такі як будівництво, захист і ремонт. У багатьох іграх, таких як Flash Element Tower Defense, вороги пробігають через «лабіринт», що дозволяє гравцям стратегічно розташовувати «вежі» для досягнення оптимального ефекту. Однак деякі версії цього жанру вимагають від користувачів створювати «лабіринти» зі своїх «веж», наприклад, Desktop Tower Defense. Деякі версії цього жанру є гібридами цих двох типів із попередньо встановленими шляхами, які можуть бути змінені до певної міри за допомогою розміщення башти, або вежами, які можуть бути змінені за допомогою розміщення шляхів. Часто ключовою стратегією є «лабіринт». Це тактика, яка створює довгий звивистий шлях, щоб збільшити відстань, яку ворог повинен пройти крізь оборону. Іноді «жонглювання» може бути можливим, загороджуючи один вихід, а потім наступний, щоб ворог рухався вперед-назад, поки не буде переможений.

Ступінь контролю гравця (або її відсутність) в таких іграх також варіюється від ігор, в яких можливе керування одиницею в ігровому світі, до ігор, в яких не можливо безпосередньо контролювати одиницю, або ігор, в яких гравець не можу керувати нічим.

Поширеною темою в іграх tower defense є «повітряні» одиниці, які ігнорують схему дошки (тобто шляхи лабіринту та перешкоди), кінцеві пункти призначення або цілі, які відрізняються від основного призначення, рухаючись безпосередньо до бажаного ворога.

У деяких іграх tower defense і на користувацьких картах гравці не тільки захищають свою дошку, а й посилають ворогів атакувати ігрове поле свого супротивника (або області, які контролюються їхніми супротивниками на спільній ігровій дошці).

1.3 Мобільні ігри

Мобільні ігри – це відеоігри, у які зазвичай грають на мобільних телефонах. Термін відноситься до всіх ігор, у які можна грати на портативних пристроях від мобільних телефонів (функціональних телефонів або смартфонів), планшетів, КПК до портативних ігрових консолей, портативних медіа-плеєрів, графічних калькуляторів, з доступом до мережі або без неї. Найпершою відомою грою для мобільних телефонів був варіант тетрісу на пристрій Hagenuk MT-2000 1994 року.

У 1997 році Nokia випустила дуже успішну модель Snake. Попередньо встановлена на більшості мобільних пристроїв виробництва Nokia, Snake (і її варіанти) з тих пір стала однією з найпопулярніших ігор з понад 350 мільйонами пристроїв у всьому світі. Варіант гри Snake для Nokia 6110 з використанням інфрачервоного порту також був першою грою для двох гравців для мобільних телефонів.

Наразі мобільні ігри зазвичай завантажуються з магазинів додатків і порталів мобільних операторів, але в деяких випадках виробники або оператори мобільного зв’язку можуть попередньо завантажувати їх на портативні пристрої через інфрачервоне з’єднання, Bluetooth або карту пам’яті під час покупки. Мобільні

ігри, які можна скачувати, були вперше комерціалізовані в Японії приблизно в той час, коли в 1999 році була запущена платформа I-mode NTT DoCoMo, а на початку 2000-х років у азіатських, європейських, північноамериканських. Зрештою вона стала доступна у більшості регіонах, де існують мережі та мобільні телефони. Платформа проіснувала до середини 2000-х років. Однак перша монетизація мобільних ігор, що розповсюджуються мобільними операторами та сторонніми порталами (завантажувані мелодії дзвінка, шпалери та інший невеликий вміст із використанням преміальних SMS або прямих тарифів оператора як механізму виставлення рахунків. Канал розвивався), залишалася маргінальною формою розповсюдження ігор до Apple iOS App Store, який був запущений у 2008 році. Як перший ринок мобільного контенту, яким безпосередньо керував власник платформи, App Store змінив поведінку споживачів, швидко збільшив ринок мобільних ігор, оскільки майже всі власники смартфонів починають завантажувати мобільні програми [12].

Мобільні ігри розроблені для роботи на різних платформах і технологіях. До них належать Palm OS, Symbian, Adobe Flash Lite, DoJa NTT DoCoMo, Java Sun, BREW, WIPI, BlackBerry, Nook та ранні варіанти Windows Mobile. На даний момент найбільш широко підтримуваною платформою є iOS від Apple і Android від Google. Мобільні версії Windows 10 від Microsoft (раніше Windows Phone) також активно підтримуються, але залишаються незначними з точки зору частки ринку порівняно з iOS та Android.

Java колись була найпопулярнішою платформою для мобільних ігор, але через її обмеження продуктивності більш складні ігри прийняли різноманітні рідні двійкові формати.

Unity є одним із найбільш широко використовуваних движків у сучасних мобільних іграх через його легкість портування на різні мобільні операційні системи та широкою спільнотою розробників. Apple пропонує ряд запатентованих технологій (наприклад, Metal), спрямованих на те, щоб допомогти розробникам більш ефективно використовувати своє обладнання в іграх iOS.

2 ВИБІР ЗАСОБІВ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

2.1 Ігрові рушії

2.1.1 Unreal Engine

Unreal Engine — ігровий движок 3D комп'ютерної графіки, розроблений Epic Games і вперше представлений у шутері від першої особи Unreal 1998 року. Спочатку він був розроблений для шутерів від першої особи для ПК, але з тих пір використовувався в різних жанрах тривимірних ігор і був прийнятий в інших галузях, особливо в кіно та телевізійній індустрії. Написаний на C++, Unreal Engine дуже портативний і підтримує широкий спектр настільних, мобільних, консольних і віртуальних платформ.

Останнє покоління Unreal Engine 5 було запущено в квітні 2022 року. Як попередник, випущений у березні 2014 року, вихідний код буде доступний на GitHub після реєстрації облікового запису, а комерційне використання буде дозволено на основі моделі лояльності. Epic звільняє маржу лояльності гри, поки розробник не заробить 1 мільйон доларів США, і звільняє від плати, якщо розробник публікує її в Epic Games Store. Движок Epic включає функції придбаних компаній, таких як Quixel. Вважається, що цьому допомогла ситуація з доходами Fortnite.

UnrealScript (часто скорочено як UScript) була рідною мовою скриптів Unreal Engine, яка використовувалася для створення ігрового коду та ігрових подій до випуску Unreal Engine 4. Ця мова була розроблена для простого програмування ігор високого рівня. Інтерпретатор UnrealScript був запрограмований Суїні, який також створив попередню мову скриптів гри, ZZT-оор.

Як і Java, UnrealScript є об'єктно-орієнтованим без множинного успадкування (всі класи успадковуються від загального класу Object), а класи визначаються в окремих файлах, названих на честь класу, який вони визначають. На відміну від Java,

UnrealScript не мав обгортки примітивного типу. Інтерфейс підтримувався лише в Unreal Engine 3-го покоління та деяких іграх Unreal Engine 2. UnrealScript підтримує перевантаження операторів, але не перевантаження методів, за винятком необов'язкових параметрів.

На конференції розробників ігор 2012 року Epic оголосила, що UnrealScript буде видалено з Unreal Engine 4 на користь C++. Візуальний сценарій підтримується системою Visual Scripting Blueprints, яка замінює попередню систему візуальних сценаріїв Kismet.

Рушій підтримує більшість операційних систем та платформ: Microsoft Windows, Linux, Mac OS та Mac OS X; консолей Xbox, Xbox 360, Xbox One, PlayStation 2, PlayStation 3, PlayStation 4, PlayStation 5, PSP, PS Vita, Wii, Dreamcast, GameCube тощо, а також на різноманітних портативних пристроях, наприклад, на пристроях Apple (iPad, iPhone), керованих системою iOS та ін.

Для спрощення портування рушій використовуємодульну систему залежних компонентів. Також є підтримка різних систем рендерингу (Direct3D, OpenGL, Pixomatic), відтворення звуку (EAX, OpenAL, DirectSound3D), засоби голосового відтворення тексту, розпізнання мови, модулі для роботи з мережею і підтримка різних пристроїв вводу.

Для мережевих ігор є підтримка технологій Windows Live, Xbox Live, GameSpy тощо, включаючи до 64 гравців (клієнтів) одночасно. Таким чином, рушій адаптувати і для використання в іграх жанру MMORPG.

2.1.2 Unity

Unity — це кросплатформний ігровий движок, розроблений Unity Technologies і вперше був оголошений і випущений у червні 2005 року на Всесвітній конференції розробників Apple Inc як ігровий движок лише для Mac OS X. З тих пір двигун поступово розширювався, щоб підтримувати різноманітні платформи для комп'ютерів, мобільних пристроїв, консолей і віртуальної реальності. Особливо

популярний при розробці мобільних ігор для iOS і Android, він використовується в таких іграх, як Pokémon Go, Monument Valley, Call of Duty: Mobile, Beat Saber і Cuphead. Вважається простим у використанні для початківців розробників і популярним для розробки інді-ігор.

Цей рушій можете використовуватись для створення 3D і 2D ігор, інтерактивних симуляцій та інших можливостей. Також використовується в інших галузях, окрім відеоігор, таких як кіно, автомобілі, архітектура, інженерія, будівництво та навіть армія США.

Unity пропонує основний API для скриптингу на C# за допомогою Mono, як для редактора Unity у формі плагінів, так і для самих ігор. До того, як C# став основною мовою програмування для движка, використовувалася Boo та була видалена у випуску Unity 5.

У 2D-іграх Unity дозволяє імпортувати спрайти та розширювати засоби візуалізації 2D світу. Для 3D-ігор Unity дозволяє специфікувати стиснення текстур, mipmaps, налаштування роздільної здатності для кожної платформи, яку підтримує ігровий движок, відображення рельєфу, відображення віддзеркалень, відображення паралакса, екранного простору навколишнього середовища (SSAO) і динамічні тіні за допомогою карт тіней, текстур і повноекранними ефектами постобробки.

На додаток до традиційного вбудованого пайплайна візуалізації доступні два окремих: High Definition Render Pipeline (HDRP) і Universal Render Pipeline (URP).

Усі три пайплайни візуалізації несумісні один з одним. Unity надає інструменти для оновлення шейдерів до URP або HDRP за допомогою застарілих засобів візуалізації.

Творці можуть розробляти створені користувачами асети та продавати їх іншим розробникам ігор через Unity Asset Store. Сюди входять 3D- та 2D-асети та середовища, які розробники можуть купувати та продавати. Unity Asset Store був запущений у 2010 році. До 2018 року було близько 40 мільйонів завантажень через цифровий магазин.

Unity — це кросплатформний движок. Хоча редактор Unity підтримується на платформах Windows, macOS та Linux, сам движок тепер підтримує створення ігор для більш ніж 19 різних платформах, таких як мобільні пристрої, настільні комп'ютери, консолі та віртуальна реальність. Офіційно підтримувані платформи з Unity 2020 LTS [13]:

- Мобільна платформа iOS, Android (Android TV), tvOS;
- Настільна платформа Windows (Універсальна платформа Windows), Mac, Linux;
- Веб-платформа WebGL;
- Консольна платформа PlayStation (PS4, PS5), Xbox (Xbox One, Xbox Series X / S), Nintendo Switch, Stadia;
- Платформа віртуальної / доповненої реальності Oculus, PlayStation VR, Google ARCore, Apple ARKit, Windows Mixed Reality (HoloLens), Magic Leap, і Via Unity XR SDK Steam VR,Google Cardboard.

Раніше підтримувалися платформи Wii, Wii U, PlayStation 3, Xbox 360, Tizen, PlayStation Vita, 3DS, BlackBerry 10 , Windows Phone 8, Samsung Smart TV, Gear VR, Daydream, Vuforia, Facebook Gameroom.

Станом на 2018 рік Unity використовувався для створення приблизно половини мобільних ігор на ринку та 60% контенту доповненої та віртуальної реальності. Це включає близько 90 відсотків нових платформ доповненої реальності, таких як Microsoft HoloLens, і 90 відсотків контенту Samsung GearVR. Технологія Unity є основою більшості віртуальної та доповненої реальності, і Fortune стверджує, що Unity «домінує у бізнесі віртуальної реальності» [14]. Unity Machine Learning Agents — це програмне забезпечення з відкритим кодом, яке з'єднує платформу Unity з програмами машинного навчання, такими як TensorFlow від Google. Методом проб і помилок із агентами машинного навчання Unity віртуальні персонажі використовують навчання з підкріпленням для створення творчих стратегій у реалістичних

віртуальних ландшафтах. Це програмне забезпечення використовується для розробки роботів і самокерованих автомобілів.

Раніше Unity підтримувала інші платформи, такі як власний веб-програвач Unity, плагіни для веб-браузера. Однак WebGL було визначено пріоритетнішим. Починаючи з версії 5, Unity надає пакети WebGL, скомпільовані в JavaScript, використовуючи двоетапний мовний перекладач (з C# на C ++ і, нарешті, JavaScript).

2.1.3 Порівняльна характеристика

Переваги Unreal Engine:

- Пробна версія абсолютно безкоштовна.
- Рушій має відкритий код. Тому в ньому можна розібратись і зрозуміти, як працюють окремі деталі. Крім того, є можливість навіть виправляти баги в рушії або самостійно доповнювати його функціонал.
- Чудова візуалізація в редакторі. Є велика кількість сучасних шейдерів, які поставляються разом з рушієм. Unreal пропонує найкращий механізм рендерингу на ринку.
- Для швидкої та простої реалізації базової ігрової логіки є зручний інструмент креслень (blueprints). Вони чудово інтегруються з C++. Це відкриває широкі можливості не тільки для початківців, а й для досвідчених розробників.

Серед недоліків виділяється нехватка документації по C++. Крім того, через постійні оновлення більшість можливостей швидко застарівають. Потрібно бути дуже уважним при перегляді матеріалів, оскільки там може описуватись неактуальна версія рушія і функції, які більше не використовуються.

Також, варто зазначити, що мобільна розробка дуже повільна. Програма розгортається на пристрої дуже довго. На Android інколи виникають візуальні проблеми – наприклад, розпливчаті контури та некоректне освітлення. В iOS рушій

підтримує тільки зборку програм, які побудовані лише із креслень. Unreal Engine – це не найкращий варіант для розробки мобільних додатків.

Переваги Unity:

- Редактор можна розширювати власними скриптами.
- Unity Asset Store має безліч ресурсів (моделі, матеріали, скрипти тощо), які можна інтегрувати у власний проєкт.
- Unity має широку спільноту спеціалістів, тому легко знайти відповідь на будь яку питання.
- Документація по рушію дуже суттєва.

До недоліків можна віднести закритий код. Тому, у випадку появи проблеми у редакторі, через яку ускладняється робота, потрібно чекати оновлення, яке повинно виправити цю проблему.

Інша проблема зв’язана з користувацьким інтерфейсом і з масштабуванням його під різноманітні дисплеї. У редактора Unity є така опція, але організувати попередній перегляд дуже важко, тому приходиться кілька разів розгортати додаток на пристрої, поки не підбереться правильна конфігурація.

В кінцевому результаті я зробив свій вибір на користь Unity. Краща пристосованість до розробки мобільних додатків стала для мене ключовою у виборі рушія. Як я писав вище, Unreal Engine справляється з цією задачею гірше. Для скриптингу в Unity використовується C#, тому саме цю мову програмування я обрав для розробки додатку.

2.2 Мова програмування C#

Мова програмування — це набір правил, який перекладає рядок або, у випадку мови візуального програмування, графічний елемент програмування в різні типи

виводу машинного коду. Мова програмування — це тип комп’ютерної мови, що використовується в комп’ютерному програмуванні для реалізації алгоритмів.

Більшість мов програмування складаються з інструкцій для комп’ютерів. Існують програмовані машини, які використовують специфічний набір інструкцій, а не загальну мову програмування. З початку 1800-х років програми використовувалися для керування роботою таких машин, як жакардові ткацькі верстати, музичні шкатулки та фортепіано. Програми на цих машинах (наприклад, сувої фортепіано) не виробляли різної поведінки у відповідь на різні введення та умови.

C # — це універсальна багатопарадигмальна мова програмування. C # включає в себе статичну типізацію, жорстку типізацію, лексичну область, обов'язкову, декларативну, функціональну, загальну, об'єктно-орієнтовану (на основі класів) і компонентно-орієнтовану області програмування.

C# був розроблений Microsoft Андерсом Хейлсбергом у 2000 році та затверджений як міжнародний стандарт Ecma (ECMA-334) у 2002 році та ISO (ISO / IEC 23270) у 2003 році. Microsoft представила C# з .NET Framework і Visual Studio. Обидва були із закритим програмним кодом. У той час у Microsoft не було продукту з відкритим кодом. Через чотири роки (у 2004 р.), був запущений безкоштовний проєкт з відкритим вихідним кодом під назвою Mono, який забезпечував міжплатформний компілятор і середовище виконання для мови програмування C #. Через десять років Microsoft випустила Visual Studio Code (редактор коду), Roslyn (компілятор) і інтегровану платформу .NET (фреймворк програмного забезпечення). Усі вони підтримують C# і є безкоштовними кросплатформними з відкритим вихідним кодом. Mono також приєднався до Microsoft, але він не був інтегрований в .NET. Станом на 2021 рік останньою версією цієї мови є C # 10.0 і була випущена в 2021 році разом із .NET 6.0.

Стандарт Ecma перелічує такі цілі проектування для C #: [15]

- Ця мова належить до простих, сучасних об’єктно-орієнтованих мов програмування загального призначення.

- Мова та її реалізація повинні підтримувати принципи програмної інженерії, такі як сильна типізація, перевірка меж масиву, виявлення спроб використання неініціалізованих змінних і автоматичне збирання сміття. Надійність програмного забезпечення, довговічність і продуктивність програміста важливі.
- Ця мова призначена для розробки програмних компонентів, придатних для розгортання в розподілених середовищах.
- Портативність дуже важлива для вихідного коду і програмістів, особливо для програмістів, які вже знайомі з C і C ++.
- Підтримка інтернаціоналізації надзвичайно важлива.
- C # підходить для створення додатків як для розміщених, так і для вбудованих систем, від дуже великих із розширеними операційними системами до дуже маленьких із виділеними функціями.
- Програми C# економні з точки зору вимог до пам’яті та потужності обробки, але ця мова не може конкурувати з C чи мовою асемблера з точки зору продуктивності та розміру.

  Основний синтаксис мови C# схожий на синтаксис інших мов у стилі C, таких як C, C++ і Java:
- Крапка з комою використовується для позначення кінця твердження.
- Для групування операторів використовуються дужки. Оператори зазвичай групуються в методи (функції), методи — у класи, а класи — у простори імен.
- Змінні призначаються за допомогою знака рівності, але порівнюються за допомогою двох послідовних знаків рівності.
- Квадратні дужки використовуються для оголошення масиву та отримання значення певного індексу на одному з них.

2.3 Вибір середовища програмування

2.3.1 IntelliJ IDEA

IntelliJ IDEA — це інтегроване середовище розробки програмного забезпечення, розроблене JetBrains для багатьох мов програмування, особливо Java, JavaScript та Python.

Перша версія з’явилася в січні 2001 року і швидко стала популярною як перше середовище для Java з широким набором інтегрованих інструментів рефакторингу, що дозволило програмістам швидко реорганізувати вихідний код своїх програм. Дизайн середовища фокусується на продуктивності програміста, що дозволяє зосередитися на функціональних задачах, а IntelliJ IDEA відповідає за повсякденні завдання.

Починаючи з шостої версії продуктів IntelliJ IDEA, надає інтегрований набір інструментів для розробки графічних інтерфейсів користувача. Серед інших функцій це середовище дуже сумісне з багатьма популярними безкоштовними інструментами розробника, такими як CVS, Subversion, Apache Ant, Maven, JUnit тощо. У лютому 2007 року розробники IntelliJ оголосили про ранню версію плагіна, який підтримує програмування на Ruby.

Починаючи з версії 9.0, середовище доступне в двох версіях: Community Edition і Ultimate Edition. Community Edition — це повністю безкоштовна версія, доступна під ліцензією Apache 2.0, з повною підтримкою Java SE, Kotlin, Groovy, Scala та інтеграцією з найпопулярнішими системами контролю версій. Ultimate Edition, доступний за комерційною ліцензією, включає підтримку Java EE, діаграм UML, обчислень покриття коду та інших систем контролю версій, мов і фреймворків.

IntelliJ IDEA підтримує безліч мов програмування, але C# серед них немає. Тому цей IDE не підходить для виконання, поставлених мною, завдань.

2.3.2 Visual Studio Code

Visual Studio Code, широко відомий як VS Code, — це редактор вихідного коду, створений Microsoft для Windows, Linux і macOS. Функції включають дебагінг, виділення синтаксису, інтелектуальне завершення коду, рефакторинг коду та підтримку вбудованого Git. Користувачі можуть змінювати теми, комбінації клавіш, налаштування та встановлювати розширення, щоб додати функціональність.

Опитування розробників Stack Overflow 2021 оцінює Visual Studio Code як найпопулярніший інструмент середовища розробника, про що повідомили 70% з 82 000 респондентів [16].

Visual Studio Code підтримує різні мовах програмування, такі як Java, JavaScript, Go, Node.js, Python, C#, C++ і Fortran. Він заснований на фреймворку Electron, який використовується для розробки веб-додатків Node.js, що працюють на движку макета Blink. Visual Studio Code використовує той самий компонент редактора (під кодовою назвою «Monaco»), що використовується в Azure DevOps (раніше відомий як Visual Studio Online і Visual Studio Team Services).

Із коробки Visual Studio Code включає базову підтримку більшості популярних мов програмування. Також є підсвічування синтаксису, узгодження дужок та згортання коду. Visual Studio Code поставляється з IntelliSense для JavaScript, TypeScript, JSON, CSS і HTML, а також підтримує налагодження Node.js. Підтримка додаткових мов може бути забезпечена розширеннями, доступними безкоштовно на VS Code Marketplace.

Замість системи проєкту користувачі можуть відкрити один або кілька каталогів і зберегти їх у робочій області для подальшого повторного використання. Це дозволяє йому діяти як мовно-незалежний редактор коду для будь-якої мови. Він підтримує багато мов програмування та набір функцій, які відрізняються від мови до мови. Непотрібні файли та папки можна виключити з дерева проєкту за допомогою налаштувань. Багато функцій Visual Studio Code не доступні з меню чи інтерфейсу користувача, але доступ до них можна отримати з палітри команд.

На Visual Studio Code можна встановлювати розширення, які доступні з центрального репозитарію. Вони включають додатки до редактора та підтримку мов. Корисною функцією є можливість створювати розширення, які додають підтримку нових мов, тем і дебагерів, інструменти для виконання статичного аналізу коду.

Visual Studio Code має вбудовану систему контролю версій. У рядку меню є окрема вкладка, яка дозволяє отримати доступ до налаштувань контролю версій, щоб переглянути зміни, внесені в поточний проєкт. Щоб скористатися цією функцією, потрібно підключити Visual Studio Code до підтримуваної системи контролю версій (Git, Apache Subversion, Perforce тощо). Це дозволяє створювати репозиторії та робити запити push та pull безпосередньо з програми Visual Studio Code.

Visual Studio Code включає кілька розширень FTP, тому ви можете використовувати програмне забезпечення як безкоштовну альтернативу веб-розробці. Є можливість синхронізувати код між редактором і сервером без необхідності завантажувати будь-яке додаткове програмне забезпечення.

Visual Studio Code дозволяє встановити кодову сторінку, на якій зберігається активний документ, символ нового рядка та мову програмування активного документа. Його можна використовувати на будь-якій платформі та будь-якій мові програмування.

Visual Studio Code збирає дані про використання та надсилає їх до Microsoft. Цю функцію можна вимкнути. Через природу програми з відкритим вихідним кодом, телеметричний код, як правило, доступний і дозволяє побачити, що саме було зібрано.

Ця програма мені не підходить, тому що це не повноцінне середовище розробки, а редактор коду. До того ж Visual Studio Code переважно використовується для веб-розробки.

2.3.3 Visual Studio

Microsoft Visual Studio — це інтегроване середовище розробки (IDE) від Microsoft. Він використовується для розробки комп'ютерних програм і веб-сайтів, веб-додатків, веб-сервісів та мобільних додатків. Visual Studio використовує платформи розробки програмного забезпечення Microsoft, такі як Windows API, Windows Forms, Windows Presentation Foundation, Windows Store і Microsoft Silverlight. Ви можете створювати як рідний, так і керований код.

Visual Studio містить редактор коду, який підтримує IntelliSense (компонент завершення коду) і рефакторинг коду. Інтегрований дебагер діє як на рівні джерела, так і як на рівні машини. Інші вбудовані інструменти включають профайлери коду, конструктори для створення програм із графічним інтерфейсом користувача, веб-дизайнери, дизайнери класів і схем баз даних. Майже на кожному є підтримка систем контролю джерел (таких як Subversion і Git), нових наборів інструментів, таких як редактори та візуальні дизайнери для мов, що належать до домену, або наборів інструментів для інших аспектів розробки програмного забезпечення. Також є плагіни для інших аспектів життєвого циклу програми (клієнт Azure DevOps: Team Explorer тощо).

Visual Studio підтримує 36 різних мов програмування, дозволяючи редакторам коду та дебагерам підтримувати майже будь-яку мову програмування (різною мірою) за наявності служб, що відповідають певній мові. Вбудовані мови включають C, C ++, C ++ / CLI, Visual Basic .NET, C #, F #, JavaScript, TypeScript, XML, XSLT, HTML і CSS. Підтримка інших мов, таких як Python, Ruby, Node.js і M, доступна через плагіни. Java (і J #) підтримувалася раніше. Найпростіша версія Visual Studio, Community Edition, доступна безкоштовно. Гасло видання Visual Studio Community — «безкоштовна повнофункціональна IDE для студентів, відкритй код та індивідуальних розробників».

Станом на 8 листопада 2021 року поточна версія Visual Studio, готова до виробництва в 2022 році, старіші версії, такі як 2013 і 2015, для розширеної підтримки, а 2017 та 2019 для основної підтримки.

Visual Studio (як і будь-яка інша IDE) містить редактор коду, який використовує IntelliSense для змінних, функцій, методів, циклів і запитів LINQ для підтримки підсвічування синтаксису та завершення коду. IntelliSense підтримується не тільки на включених мовах, а й у XML, CSS і JavaScript під час розробки веб-сайтів і веб-додатків. Пропозиції автозавершення відображаються в безмодовому списку біля курсору редагування у вікні редактора коду. У Visual Studio 2008 і новіших версіях ви можете тимчасово зробити його напівпрозорим, щоб побачити, чи не зламано ваш код.

Саме цей IDE я обрав для скриптингу, тому що він найбільше пристосований до написання коду мовою C#.

2.4 Особливості архітектури додатку на Unity

Програмна частина додатку на Unity складається із скриптів. Вони можуть бути представлені класом, інтерфейсом тощо. Такі скрипти додаються до ігрових об’єктів та взаємодіють між собою під час роботи програми. Про таку взаємодію і піде мова далі.

Найпростіший спосіб взаємодії скриптів – це передача посилань на клас напряму. Тобто в одному класі вказати поле типу клас і серіалізувати його, а в інспекторі передати екземпляр цього класу в поле. Проте в цього методу є один суттєвий недолік – при великій кількості скриптів, така архітектура викликає труднощі в її розумінні. Система починає бути схожою на павутину.

Також можна використовувати ScriptableObject. Це клас, який не прив’язаний до об’єктам сцени, а існує у вигляді окремих асетів і може зберігати і переносити данні між ігровими сесіями. Можна створити кілька скриптів типу ScriptableObject і зберігати посилання на інші об’єкти. Достатньо буде один раз налаштувати їх в

інспекторі, а потім, за необхідності, звертатися до них з інших класів. Така система також має недолік. ScriptableObject не прив’язаний до конкретної ігрової сцени, тому якщо в певній такій сцені звернутися до об’єкту, який знаходиться в іншій сцені, то це викличе ряд проблем, вирішення яких може ускладнити систему, що нівелює переваги ScriptableObject. Тому цей метод не підходить для вирішення всіх завдань.

Наступний спосіб називається Singleton. Це об’єкт, який існує тільки в одному екземплярі на ігровій сцені. Таким чином, він може зберігати в собі всю інформацію про гру, а також переміщатися між сценами, переносячи данні, та видаляти своїх двійників. Такий тип зв’язку не потребує налаштовувати посилання на інші класи вручну. Складається ситуація, коли будь який об’єкт має доступ до будь якої інформації. Це може зламати систему. Грамотно налаштована інкапсуляція знизить ризики, але не виправить ситуацію повністю. Тому не варто зловживати синглтоном.

Підводячи підсумок, варто зазначити, що кожен із методів має свої переваги і недоліки, тому потрібно використовувати їх відносно того, яке завдання потрібно вирішити.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Файлова структура логіки програми

Логіка програми реалізована за допомогою скриптів, які визначають як повинен діяти той ти інший об’єкт на ігровій сцені. В основному скрипти представлений за допомогою класів, більшість з яких наслідуються від юнітівського класу MonoBehaviour.

Програма налічує 23 файли скриптів. Їх, для зручності, можна поділити на 6 частин: логіка самого рівня, логіка ворогів, реалізація патерну Object Pooling, логіка оборонних веж, нанесення шкоди ворогам, інтерфейс користувача. Весь код представлений в додатку А.

3.2 Логіка рівня

Клас LVLManager реалізує патерн Singleton, тому він існує лише в одному екземплярі на ігровій сцені. Він виконує 3 задачі: стартує появу ворогів на сцені, рахує кількість ігрової валюти, завершає рівень. Після загрузки сцени, запускається таймер, по закінченні якого вороги починають штурмувати головну будівлю, яку, за правилами гри, потрібно захищати. Затримка перед штурмом потрібна для того, щоб гравець зміг оцінити рівень і як він повинен діяти. Ігрова валюта потрібна, щоб будувати вежі, які будуть захищати головну будівлю від знищення. Саме в класі LVLManager відбувається збереження та підрахунок цієї валюти та виведення її на екран. Також клас перевіряє наявність ворогів на сцені і у випадку, якщо вони скінчились, завершається рівень і з’являється UI елемент, який пропонує перейти на наступний рівень, пройти поточний ще раз, або повернутись в головне меню.

Клас WavesManager контролює появу хвиль ворогів на сцені. Після команди від класу LVLManager, скрипт починає спавнити ворогів у точці, де знаходиться ігровий об’єкт, який має в собі цей скрипт. Безпосередньо сама поява відбувається за рахунок пула об’єктів, який буде описуватись пізніше. Основне поле скрипта – це масив екземплярів класу Wave.

Клас Wave потрібен лише для зручного користування попереднім класом. Він відповідає за налаштування кожної хвилі окремо та зберігає інформацію про конкретного ворога, якого потрібно поставити на сцені, кількість таких ворогів та інтервал між їх появою.

3.3 Object Pooling

Object pooling, або пул об’єктів — це чудовий спосіб оптимізувати ваші проєкти та зменшити навантаження на центральний процесор, коли доводиться швидко створювати та знищувати ігрові об’єкти. Це хороша практика та патерн дизайну, який слід пам’ятати, щоб допомогти розвантажити процесорну потужність CPU для обробки більш важливих завдань і не засипати повторюваними викликами створення та знищення [17]. Це особливо корисно, коли ви маєте справу з кулями в шутері.

Якщо вам доводиться працювати з великою кількістю об’єктів, екземпляр яких особливо «дорогий», і кожен об’єкт потрібен лише протягом короткого періоду часу, це може негативно вплинути на загальну продуктивність вашої програми. У таких випадках можна вважати бажаним патерн дизайну Object pooling.

Патерн створює набір повторно використовуваних об’єктів. Коли потрібен новий об’єкт, він буде викликаний пулом. Якщо раніше підготовлений об’єкт доступний, він буде негайно повернений, уникаючи витрат на створення екземпляра. Якщо об’єкта не існує в пулі, буде створено та повернуто новий елемент. Коли об’єкт використаний і більше не потрібен, він повертається в пул і може бути використаний

знову в майбутньому без повторення, дорогого для обчислень процесу створення, екземпляра [18].

Деякі пули об’єктів мають обмежені ресурси, тому вказується максимальна кількість. Якщо це число буде досягнуто, і запитуватиметься новий елемент, викличеться помилка або потік буде заблоковано, поки об’єкт не з’явиться в пулі.

Пул нічого не знає про реалізацію збереженого об'єкта. Тому повернений об’єкт вважається таким, що перебуває у невизначеному стані. Для подальшого використання його необхідно перевести (скинути) у вихідний стан. Повторне використання може призвести до помилок. Тому при скиданні необхідно очистити поля (повернути до початкового стану).

Можливе сумісне використання Object pooling з іншими патернами. Наприклад, ви можете використовувати Prototype для створення об’єкта в певному стані. А за допомогою Singleton створити єдиний екземпляр пулу у вашій системі.

Приховування пулу за іншими патернами – погана практика. Розробники, які використовують такий «гібрид», не очікують обов’язкового повернення об’єкта із, наприклад, фабричного методу. І якщо об’єкт не повернутий, сам пул стає марним. У цьому випадку потрібне відокремлення реалізації класів, які створюють об’єкти.

Конкретно в моєму проєкті патерн представлений трьома файлами: двома класам та інтерфейсом. Останній називається IPulable і потрібен для взаємодії з ігровими об’єктами, які зберігаються в пулі.

Основним класом є ObjectPool. У грі є кілька видів об’єктів, які потрібно зберігати в пулі, тому клас має словник для таких цілей. Тобто кожен вид ігрового об’єкту зберігається окремо. Це потрібно для зручного користування пулом. Він самостійно додає пункти словника за потреби. Клас також реалізує патерн Singleton.

Останнім класом пулу об’єктів є PoolTask. Екземплярів цього класу може бути кілька, залежно від кількості видів ігрових об’єктів, які з’явилися на сцені. Клас відповідає за додавання об’єктів в пул (або їх створення за потреби) збереження їх у вигляді списку та вилучення об’єктів з пулу.

3.4 Скриптинг ворогів

Клас EnemyMover відповідає за рух ворогів до головної будівлі та атаку її, а також керування анімацій, які супроводжують ці дії. Сам рух реалізований за допомогою Nav Mesh Agent. Це компонент, який спрощує роботу з навігацією ігрових об’єктів у просторі. Для роботи потрібно додати його до потрібного об’єкта, налаштувати навігаційну сітку в панелі Navigation, а в скрипті вказати точку, в яку потрібно рухатись. Програма сама прокладе найоптимальніший маршрут до потрібної координати. При підході до потрібної дистанції, ворог переходить у бойовий режим і починає завдавати шкоди головній будівлі.

Клас EnemyManager відповідає за отримання ворогами шкоди від оборонних веж. Ключовим полем є currentHP, яке зберігає бали шкоди, які ворог може отримати до свого знищення. На ігровій сцені вони відображаються у відсотковому еквіваленті у вигляді слайдера над ворогом. При нанесенні шкоди ворогу, бали шкоди зменшуються і їх кількість актуалізується на слайдері. У випадку, коли кількість балів перестане бути додатною ворог знищується, тобто переходить у пул об’єктів, а за нього начисляється нагорода у вигляді ігрової валюти.

3.5 Логіка оборонних веж та головної будівлі

Клас PlatformManager відповідає за розміщення веж на платформі та їх знищення. При натисненні на платформу, а екрані з’являється UI елемент, де гравець повинен вибрати потрібну йому вежу. Також там є кнопка очистки платформи від вежі, яка там уже стоїть. Це потрібно для того, щоб поміняти одну вежу на іншу, адже її можна розмістити лише одну на платформі. В цьому ж класі відбувається списання ігрової валюти з рахунку гравця за розташування башти та повернення половини її ціни за знищення.

Скрипт TowerButton знаходиться безпосередньо на кнопці кожної вежі на UI елементі із їх вибором. Він потрібен для того, щоб повідомляти класу PlatformManager яку саме башту потрібно поставити на платформу.

Клас TowerChecker – це система наведення вежі. Він перевіряє наявність ворогів у певному радіусі і передає інформацію про них в інший клас, якщо такі знайдуться. Скрипт розпізнає ворог це, чи ні за шаром (Layer) Enemy. Коли ворог покидає область видимості веж, та перестає слідкувати і стріляти по ньому.

Клас TowerManager відповідає за стрільбу башти по ворогам. Саме йому TowerChecker вказує по кому потрібно стріляти. По суті, скрипт просто ініціює появу куті та вказує їй куди потрібно летіти. Безпосередньо за саму появу кулі відповідає пул об’єктів.

Клас TargetManager відповідає пошкодження головної будівлі. Логіка схожа із отриманням шкоди ворогами. Після зниження, головна будівля зникає, а на екрані з’являється UI елемент із повідомленням про поразку гравця.

3.6 Нанесення пошкоджень ворогам

Клас DamageDealer лежить в основі нанесення шкоди ворогам. Він реалізує інтерфейс IPoolable для перенесення кулі в пул об’єктів. Скрипт заставляє кулю летіти точно в ціль, а у випадку, якщо ціль було знищено до того, як куля в неї потрапила, то об’єкт кулі зникає, тобто потрапляє в пул. Також клас відслідковує зіткнення снаряду від вежі із порогом та ініціює нанесення шкоди.

Класи SingleDD і RangeDD наслідуються від класу DamageDealer і відповідають безпосередньо за нанесення шкоди ворогам. Різниці між ними полягає в тому, що перший завдає пошкоджень лише цілі, в яку потрапила куля, а другий – всім цілям у певному радіусі потрапляння снаряду від вежі.

Клас SetFire наслідується від класу SingleDD і потрібен для накладання на ворога ефекту «горіння». Суть його полягає в тому, що ціль отримує певну кількість шкоди типу «вогонь» протягом 5 секунд раз в секунду. Варто зазначити, що гра має 2 типу шкоди: фізичний і вогняний. Суть їх в тому, що різні вороги по-різному сприймають ці типи: якийсь б'є по ним сильніше а якийсь слабше. В коді це реалізовано у вигляді коефіцієнтів, тобто будь яка шкода спочатку множиться на коефіцієнт а потім віднімається від балів шкоди.

Скрипт DeathBox використовується лише один раз на сцені головного меню. Він потрібен для того щоб вороги не накопичувались і не впливали сильно на продуктивність роботи програми. Клас знищує ворогів, які зайшли за межі камери.

3.7 Скриптування інтерфейсу користувача

Клас ChangeMenuStatus потрібен, щоб на сцені з'являлися, чи зникали певні UI елементи при натисненні на потрібну кнопку. Скрипт використовується в двох містях панель вибору рівня в головному меню (Рисунок 3.1) та меню паузи (Рисунок 3.2).

Рисунок 3.1 – Панель вибору рівня

Рисунок 3.2 – Меню паузи

Клас ChangeLVL переключає сцени, тобто коли потрібно обрати певний рівень, перезапустити поточний, переключити на наступний, або вийти в головне меню.

Клас Pause керує часом на рівні. При натисненні на потрібно кнопку час на рівні зупиняється, або продовжує далі йти.

Клас TowerUIController по-суті є посередником між PlatformManager і TowerButton. Перший вказує куди саме потрібно поставити вежу, тобто куди передати інформацію про башту, яку обрав гравець, а другий вказує, яку саме вежу потрібно поставити. Приклад UI елементу із вибором башти зображений на рисунку 3.3.

Рисунок 3.3 – Панель вибору башти

Клас ExitButton відповідає за вихід із гри при натисненні на кнопку. Сама кнопка знаходиться, лише у головному меню (Рисунок 3.4).

Рисунок 3.4 – Панель головного меню

3.8 Тестування розробленого програмного продукту

Реалізована гра відповідає таким вимогам:

— керування інтуїтивно зрозуміле для користувача;
— механіки гри інтуїтивно зрозумілі та освоюються емпіричним шляхом;
— вороги не відкидають один одного;
— всі рівні можливо пройти;
— кожен об’єкт має чіткі границі, які відповідають візуальним розмірам модельки предмета;
— навігаційна сітка відповідає стежкам на об’єкті поверхні землі;
— при недостатній кількості ігрової валюти, вежа не ставиться і рахунок не йде в мінус;
— кулі не залишаються на сцені, якщо зник ворог до того, як вони в нього влучил

ВИСНОВКИ

Проаналізувавши предметну область, у ході дипломної роботи було створено ігровий мобільний додаток у жанрі «tower defense». В процесі виконання даного проєкту було досліджені методи, патерни та технології розробки ігрового програмного забезпечення для Android платформи.

1. Було проведено аналіз предметної області. Було досліджено явище відеоігор, їх історію та особливості розробки. Виявлено суть жанру «tower defense» та його ігрового процесу. Задля розуміння концепції майбутньої системи було проведено аналіз існуючих представників цього жанру. Також було проведено дослідження особливостей відеоігор на мобільні платформи.
2. Було проведено аналіз існуючих програмних засобів та інструментів для реалізації проєкту. Серед ігрових рушіїв був вибір між Unreal Engine та Unity. В кінцевому результаті було обрано останній. Краща пристосованість до розробки мобільних додатків стала для мене ключовою у виборі рушія. Unreal Engine справляється з цією задачею гірше. Для скриптингу в Unity використовується C#, тому саме цю мову програмування я обрав для розробки додатку. При виборі середовища розробки, було розглянуто Intellij IDEA, Visual Studio Code та Visual Studio. Intellij IDEA не підтримує мову програмування C#, Visual Studio Code – це не повноцінне середовище розробки, а просто редактор коду, тому ці програми не можуть виконати поставлені задачі. Тому в процесі аналізу було обрано Visual Studio, який найбільше пристосований для написання коду мовою C#.
3. Була спроектована архітектура ігрового додатку. Програмна частина проєкту складається із 23 файлів: 22 класів і одного інтерфейсу. Скрипти взаємодіють між собою напряму, передачею через інспектор або методом, який відповідає за опрацювання входження одного колайдера в інший. Також використовується патерн Singleton.

4. Був створений програмний продукт. Скрипти можна поділити на 6 частин: логіка самого рівня, логіка ворогів, реалізація патерну Object Pooling, логіка оборонних веж, нанесення шкоди ворогам, інтерфейс користувача.

Логіка рівня відповідає за його початок і завершення, а також за появу на ігровій сцені ворогів. Логіка ворогів – це скрипти, які відповідають за їх рух, атаку та отримання шкоди. Реалізація патерну Object Pooling потрібна для ефективного використання ресурсів мобільного пристрою під час роботи додатку. Логіка оборонних веж представлена скриптами, які потрібні для встановлення цих веж на відповідні платформи, пошуку ворогів на ігровій сцені та ініціювання появи куль, які мають скрипти для нанесення шкоди ворогам. Класи для інтерфейсу потрібні, для його функціонування.

5. Було проведене тестування, в результаті якого було визначено, що програма відповідає наступним вимогам:

— керування інтуїтивно зрозуміле для користувача;
— механіки гри інтуїтивно зрозумілі та освоюються емпіричним шляхом;
— вороги не відкидають один одного;
— всі рівні можливо пройти;
— кожен об’єкт має чіткі границі, які відповідають візуальним розмірам модельки предмета;
— навігаційна сітка відповідає стежкам на об’єкті поверхні землі;
— при недостатній кількості ігрової валюти, вежа не ставиться і рахунок не йде в мінус;
— кулі не залишаються на сцені, якщо зник ворог до того, як вони в нього влучили.

ПЕРЕЛІК ПОСИЛАНЬ

1. Newzoo’s Mobile Trends to Watch in 2020. [Електронний ресурс] – Ресурс доступу: https://newzoo.com/cn/articles/newzoos-mobile-trends-to-watch-in-2020/.
2. U.S. Patent 2,455,992, 1948 [Електронний ресурс] – Ресурс доступу: https://patents.google.com/patent/US2455992
3. Stefan H., How COVID-19 is taking gaming and esports to the next level, World Economic Forum, 2020.
4. The Edge of Reason?, 2009. [Електронний ресурс] – Ресурс доступу: http://www.eurogamer.net/articles/the-edge-of-reason
5. Reimer J., Cross-platform game development and the next generation of consoles, Ars Technica, 2005.
6. Edwards B., Son of PC: The History of x86 Game Consoles, PC Magazine, 2016.
7. Reynolds C., Assassin's Creed II dev team triples in size, NOW Gamer, 2009.
8. Rubens A., The Creation of Missile Command and the haunting of its creator Dave Theurer, Polygon, 2013.
9. Valadares G., Antbuster, jogo brasileiro é premiado em disputa internacional - Papo de Homem, Papo de Homem, 2007.
10. Andrews J., Dungeon Defenders Exceeds More than a Quarter of a Million in Sales, Internal News, 2012.
11. Sliwinski A., Dungeon Defenders picks up gold from 600K sales, Joystiq, 2013.
12. State of the Art of the European Mobile Games Industry, 2013. [Електронний ресурс] – Ресурс доступу: https://web.archive.org/web/20170505184626/http://www.mobilegamearch.eu/wp-content/uploads/2012/12/Mobile-Game-Arch_D3.1_06122012_PU.pdf
13. Unity - Manual: System requirements for Unity 2020 LTS. [Електронний ресурс] – Ресурс доступу: https://docs.unity3d.com/Manual/system-requirements.html

14. Gaudiosi, J., This company dominates the virtual reality business, and it's not named Oculus, Fortune, USA, 2016.
15. C# Language Specification (4th ed.). [Електронний ресурс] – Ресурс доступу: https://www.ecma-international.org/publications-and-standards/standards/ecma-334/
16. Stack Overflow Developer Survey 2021 - Integrated Development Environment, 2021. [Електронний ресурс] – Ресурс доступу: https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-integrated-development-environment
17. Unity – Scripting API: ObjectPool<T0>. [Електронний ресурс] – Ресурс доступу: https://docs.unity3d.com/2022.1/Documentation/ScriptReference/Pool.ObjectPool_1.html
18. Kircher M., Prashant J. Pooling, *EuroPLoP 2002*, Germany, 2002, pp. 8-9.

ДОДАТОК А

Код програми

LVLManager.cs:

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

public class LVLManager : MonoBehaviour
{
    public static LVLManager Instance;

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

    [SerializeField] private float startTime;
    [SerializeField] private LayerMask enemies;
    [SerializeField] private float radiusOfVisibility;
    [SerializeField] private WavesManager[]
wavesManager;
    private bool lvlEnded = false;
    [SerializeField] public int coins;
    [SerializeField] private GameObject coinsText;
    [SerializeField] private GameObject completeUI;

    private void Start()
    {
        StartCoroutine("StartWavesManager");
        coinsText.GetComponent<TMP_Text>().text =
coins.ToString();
    }

    private void FixedUpdate()
    {
        if (lvlEnded)
            return;

        if (!EnemyCheck())
        {
            for (int i = 0; i < wavesManager.Length; i++)
            {
                if (!wavesManager[i].endWork)
                    return;
            }

            EndLvl();
        }
    }

    IEnumerator StartWavesManager()
    {
        yield return new WaitForSeconds(startTime);

        for (int i = 0; i < wavesManager.Length; i++)
        {
            wavesManager[i].started = true;
        }
```

```
    }

    private void EndLvl()
    {
        Debug.Log("End");
        completeUI.SetActive(true);
        lvlEnded = true;
    }

    private bool EnemyCheck()
    {
        return Physics.CheckSphere(transform.position,
radiusOfVisibility, enemies);
    }

    public void ChangeCoins(int value)
    {
        coins += value;
        coinsText.GetComponent<TMP_Text>().text =
coins.ToString();
    }

    private void OnDrawGizmos()
    {

Gizmos.DrawWireSphere(gameObject.transform.position,
radiusOfVisibility);
    }
}
```

WavesManager.cs:

```
using System.Collections;
using UnityEngine;
using Object_Pooling;
public class WavesManager : MonoBehaviour
{
    [SerializeField] private Wave[] waves;
    private bool working = false;
    private int numberOfWave = 0;
    [HideInInspector] public bool started;
    [HideInInspector] public bool endWork = false;
    private ObjectPool objectPool;

    private void Start()
    {
        objectPool = ObjectPool.Instance;
    }

    private void FixedUpdate()
    {

        if (numberOfWave >= waves.Length)
        {
            endWork = true;
            return;
        }
        else if (!working &&
waves[numberOfWave].countOfEnemies > 0)
            StartCoroutine("Spawn");
        else if (waves[numberOfWave].countOfEnemies <=
0)
            numberOfWave++;
    }

    IEnumerator Spawn()
    {
        working = true;

objectPool.GetObject(waves[numberOfWave].enemyPrefa
b).transform.position = transform.position;
        waves[numberOfWave].countOfEnemies--;
```

```
        yield return new
WaitForSeconds(waves[numberOfWave].interval);
        working = false;
    }
}
```

Wave.cs:

```
[System.Serializable]
public class Wave
{
    public EnemyManager enemyPrefab;
    public float interval;
    public int countOfEnemies;
}
```

ObjectPool.cs:

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



namespace Object_Pooling
{
    class ObjectPool : MonoBehaviour
    {
        private Dictionary<IPoolable, PoolTask>
activePoolTask;
        [HideInInspector] public Transform
objectPoolTransform;
        public static ObjectPool Instance;


        private void Awake()
        {
            if (Instance == null)
                Instance = this;
            else
                Destroy(gameObject);
            objectPoolTransform = gameObject.transform;
            activePoolTask = new Dictionary<IPoolable,
PoolTask>();
        }


        public T GetObject<T>(T prefab) where T :
MonoBehaviour, IPoolable
        {
            if (!activePoolTask.TryGetValue(prefab, out var
poolTask))
                AddTaskToPool(prefab, out poolTask);


            return poolTask.GetFreeObject<T>(prefab);
        }


        private void AddTaskToPool<T>(T prefab, out
PoolTask poolTask) where T : MonoBehaviour, IPoolable
        {
            GameObject container = new GameObject
            {
                name = $"{prefab.name}s_pool"
            };

container.transform.SetParent(objectPoolTransform);
            poolTask = new PoolTask(container.transform);
            activePoolTask.Add(prefab, poolTask);
        }
    }
}
```

PoolTask.cs:

```
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using Object = UnityEngine.Object;
```

```
namespace Object_Pooling
{
    class PoolTask
    {
        private readonly List<IPoolable> freeObjects;
        private readonly Transform container;

        public PoolTask (Transform container)
        {
            freeObjects = new List<IPoolable>();
            this.container = container;
        }

        public T GetFreeObject<T>(T prefab) where T :
MonoBehaviour, IPoolable
        {
            T poolObject = null;
            if (freeObjects.Count > 0)
            {
                poolObject = freeObjects.Last() as T;
                poolObject.GameObject.SetActive(true);
                freeObjects.Remove(poolObject);
            }

            if (poolObject == null)
                poolObject = Object.Instantiate(prefab);
            poolObject.OnReturnToPool += ReturnToPool;
            return poolObject;
        }

        private void ReturnToPool (IPoolable poolObject)
        {
            freeObjects.Add(poolObject);
            poolObject.GameObject.SetActive(false);
            poolObject.GameObject.transform.position = new
Vector3(0, 0, 0);
            poolObject.Transform.SetParent(container);
            poolObject.OnReturnToPool -= ReturnToPool;
        }
    }
}
```

IPoolable.cs:

```
using System;
using UnityEngine;

namespace Object_Pooling
{
    public interface IPoolable
    {
        Transform Transform { get; }
        GameObject GameObject { get; }
        event Action<IPoolable> OnReturnToPool;
        void ReturnToPool();
    }
}
```

EnemyManager:cs:

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

public class EnemyManager : MonoBehaviour, IPoolable
{
    [SerializeField] private float maxHP;
    private float currentHP;
```

```
    [SerializeField] private float
physicalDamageMultiplayer = 1;
    [SerializeField] private float fireDamageMultiplayer = 1;
    [SerializeField] private Canvas canvas;
    [SerializeField] private Slider hpSlider;
    [SerializeField] private int revard;
    private LVLManager lvlManager;

    public Transform Transform => transform;
    public GameObject GameObject => gameObject;
    public event Action<IPoolable> OnReturnToPool;
    public void ReturnToPool()
    {
        OnReturnToPool?.Invoke(this);
    }



    void Start()
    {
        currentHP = maxHP;
        hpSlider.maxValue = maxHP;
        hpSlider.value = maxHP;
        lvlManager = LVLManager.Instance;
    }


    void FixedUpdate()
    {
        canvas.transform.rotation = new Quaternion(0,
transform.rotation.y * -1, 0, 0);
    }


    public void GetDamage (float damage, DamageType
damageType)
    {
        switch (damageType)
        {
            case DamageType.Physical:
                currentHP -= damage *
physicalDamageMultiplayer;
                break;
            case DamageType.Fire:
                currentHP -= damage * fireDamageMultiplayer;
                break;
        }


        hpSlider.value = currentHP;


        if (currentHP <= 0)
            OnDeath();
    }


    public void OnDeath ()
    {
        StopCoroutine("Burning");
        lvlManager.ChangeCoins(revard);
        ReturnToPool();
        currentHP = maxHP;
        hpSlider.value = maxHP;
    }
    IEnumerator Burning(float damage)
    {
        for (int i = 0; i < 5; i++)
        {
            GetDamage(damage, DamageType.Fire);
            yield return new WaitForSeconds(1f);
        }
    }


}


public enum DamageType
```

```
{
    Physical = 1,
    Fire = 2,
}
```

EnemyMover.cs:

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

public class EnemyMover : MonoBehaviour
{
    private NavMeshAgent navMeshAgent;
    [SerializeField] private float speed;
    private Transform target;
    [SerializeField] private LayerMask targetMask;
    [SerializeField] private float damage;
    [SerializeField] private float delay;
    private bool isAttack = false;
    private Animator animator;

    void Start()
    {
        navMeshAgent = GetComponent<NavMeshAgent>();
        navMeshAgent.speed = speed;
        animator = GetComponent<Animator>();
    }

    private void FixedUpdate()
    {
        if (Physics.CheckSphere(transform.position, 100f, targetMask) && target == null)
            target = Physics.OverlapSphere(transform.position, 100f, targetMask)[0].transform;
        else if (!Physics.CheckSphere(transform.position, 100f, targetMask))
            return;

        if (Vector3.Distance(gameObject.transform.position, target.position) > 2f)
        {
            animator.SetBool("Move", true);
            navMeshAgent.speed = speed;
            navMeshAgent.destination = target.position;
        }

        if (Vector3.Distance(gameObject.transform.position, target.position) < 2f)
        {
            navMeshAgent.speed = 0;
            gameObject.transform.LookAt(target);
            animator.SetBool("Move", false);
            if (!isAttack)
            {
                isAttack = true;
                animator.SetBool("Attack", true);
            }
        }
    }

    private void Attack()
    {
        if (target != null)
        {
            target.GetComponent<TargetManager>().GetDamage(damage);
        }

    }

    IEnumerator AttackDelay()
    {
```

```
        animator.SetBool("Attack", false);
        yield return new WaitForSeconds(delay);
        isAttack = false;
    }
}
```

PlatformManager.cs:

```
using UnityEngine;

public class PlatformManager : MonoBehaviour
{
    [SerializeField] private GameObject[] towers;
    [SerializeField] private GameObject towerUI;
    [SerializeField] private Vector3 d;
    private LVLManager lvlManager;
    private GameObject currentTower;
    private bool isPlaced = false;

    private void Start()
    {
        lvlManager = LVLManager.Instance;
    }
    private void OnMouseDown()
    {
        towerUI.SetActive(true);

towerUI.GetComponent<TowerUIControllser>().platform
= this;
    }

    public void ChooseTower (Towers tower)
    {
        if
(towers[(int)tower].GetComponent<TowerManager>().cost
> lvlManager.coins || isPlaced)
            return;
        currentTower =
GameObject.Instantiate(towers[(int)tower],
gameObject.transform.position + d, Quaternion.identity);
        lvlManager.ChangeCoins(-
towers[(int)tower].GetComponent<TowerManager>().cost)
;
        towerUI.SetActive(false);
        isPlaced = true;
    }

    public void RemoveTower()
    {
        if (!isPlaced)
            return;

lvlManager.ChangeCoins(currentTower.GetComponent<T
owerManager>().cost / 2);
        Destroy(currentTower);
        isPlaced = false;
    }
}

public enum Towers
{
    SimpleTower,
    RocketLauncher,
    Flamer,
    DethLaser,
}
```

TargetManager.cs:

```
using UnityEngine;
using UnityEngine.UI;

public class TargetManager : MonoBehaviour
```

```
{
    [SerializeField] private float maxHP;
    [SerializeField] private Slider slider;
    [SerializeField] private GameObject loseUI;
    private float currentHP;

    void Start()
    {
        currentHP = maxHP;
        slider.maxValue = maxHP;
        slider.value = maxHP;
    }

    public void GetDamage(float damage)
    {
        currentHP -= damage;
        slider.value = currentHP;
        if (currentHP <= 0)
        {
            loseUI.SetActive(true);
            Destroy(gameObject);
        }
    }
}
```

TowerButton.cs:

```
using UnityEngine;
using UnityEngine.UI;

public class TowerButton : MonoBehaviour
{
    [SerializeField] private Towers tower;
    [SerializeField] private GameObject towerUI;
    private Button button;

    void Start()
    {
        button = GetComponent<Button>();
        button.onClick.AddListener(ChooseTower);
    }

    public void ChooseTower()
    {
towerUI.GetComponent<TowerUIControllser>().ChooseTower(tower);
    }

    private void OnDestroy()
    {
        button.onClick.RemoveAllListeners();
    }
}
```

TowerChecker.cs:

```
using UnityEngine;

public class TowerChecker : MonoBehaviour
{
    private TowerManager towerManager;
    [SerializeField] private float radiusOfVisibility;
    [SerializeField] private LayerMask enemies;
    private GameObject currentTarget;

    private void Start()
    {
        towerManager = GetComponent<TowerManager>();
    }
```

```
    private void FixedUpdate()
    {
        if(Physics.CheckSphere(transform.position, radiusOfVisibility, enemies) && currentTarget == null)
        {
            currentTarget = Physics.OverlapSphere(transform.position, radiusOfVisibility, enemies)[0].gameObject;
            towerManager.ChangeTarget(currentTarget);
        }
        else if (currentTarget != null)
        {
            if(Vector3.Distance(transform.position, currentTarget.transform.position) > radiusOfVisibility || currentTarget.activeInHierarchy == false)
            {
                currentTarget = null;
                towerManager.ChangeTarget(currentTarget);
            }
        }
    }

    private void OnDrawGizmos()
    {
        Gizmos.DrawWireSphere(transform.position, radiusOfVisibility);
    }
}
```

TowerManager.cs:

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

public class TowerManager : MonoBehaviour
{
    [SerializeField] private GameObject head;
    private GameObject currentTarget = null;

    [SerializeField] private Transform shootPoint;
    [SerializeField] private float delay;
    [SerializeField] private DamageDealer bullet;
    private bool isShooting = false;
    [SerializeField] public int cost;
    private ObjectPool objectPool;

    void Start()
    {
        objectPool = ObjectPool.Instance;
    }

    private void FixedUpdate()
    {
        if (currentTarget != null && currentTarget.activeInHierarchy == true)
        {
            head.transform.LookAt(currentTarget.transform);
            if (!isShooting)
                StartCoroutine("Shoot");
        }
    }

    public void ChangeTarget(GameObject target)
    {
        currentTarget = target;
    }

    IEnumerator Shoot ()
    {
        isShooting = true;

objectPool.GetObject(bullet.GetComponent<DamageDealer>()).OnSpawn(shootPoint, currentTarget);
```

```
        yield return new WaitForSeconds(delay);
        isShooting = false;
    }
}
```

DamageDealer.cs:

```
using UnityEngine;
using Object_Pooling;
using System;

public abstract class DamageDealer : MonoBehaviour, IPoolable
{
    private GameObject target;
    [SerializeField] private float speed;
    [SerializeField] protected float damage;
    [SerializeField] protected DamageType damageType;
    [SerializeField] protected LayerMask enemyLayer;
    private Vector3 lastTargetPoint;

    public event Action<IPoolable> OnReturnToPool;
    public Transform Transform => transform;
    public GameObject GameObject => gameObject;
    public void ReturnToPool()
    {
        OnReturnToPool?.Invoke(this);
    }


    void Start()
    {
        lastTargetPoint = gameObject.transform.position;
    }


    void FixedUpdate()
    {
        if (target != null && target.activeInHierarchy == true)
        {
            transform.position = Vector3.MoveTowards(transform.position, target.transform.position, Time.deltaTime * speed);
            lastTargetPoint = target.transform.position;
            transform.LookAt(target.transform);
        }
        else
        {
            transform.position = Vector3.MoveTowards(transform.position, lastTargetPoint, Time.deltaTime * speed);
            transform.LookAt(lastTargetPoint);
            if (Vector3.Distance(transform.position, lastTargetPoint) < 0.5f)
                ReturnToPool();
        }


    }


    public void OnSpawn(Transform spawnPoin, GameObject target)
    {
        gameObject.SetActive(false);
        transform.position = spawnPoin.position;
        gameObject.SetActive(true);
        this.target = target;
    }


    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Enemy"))
        {
            TakeDamage(other);
            ReturnToPool();
```

```
        }
    }
    protected virtual void TakeDamage(Collider other) {}
}
```

DeathBox.cs:

```
using UnityEngine;

public class DeathBox : MonoBehaviour
{
    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Enemy"))
        {

other.gameObject.GetComponent<EnemyManager>().On
Death();
        }
    }
}
```

RangeDD.cs:

```
using UnityEngine;

public class RangeDD : DamageDealer
{
    [SerializeField] private float radiusOfDamage;
    private Collider[] enemies;

    protected override void TakeDamage(Collider other)
    {
        base.TakeDamage(other);
        enemies =
Physics.OverlapSphere(other.transform.position,
radiusOfDamage, enemyLayer);
        for (int i = 0; i < enemies.Length; i++)

enemies[i].GetComponent<EnemyManager>().GetDamage
(damage, damageType);
    }
}
```

SetFire.cs:

```
using UnityEngine;

public class SetFire : SingleDD
{
    [SerializeField] private float fireDamage;

    protected override void TakeDamage(Collider other)
    {
        base.TakeDamage(other);

other.GetComponent<EnemyManager>().StartCoroutine("
Burning", fireDamage);
    }
}
```

SingleDD.cs:

```
using UnityEngine;
public class SingleDD : DamageDealer
{
    protected override void TakeDamage(Collider other)
    {
        base.TakeDamage(other);

other.GetComponent<EnemyManager>().GetDamage(dam
age, damageType);
    }
}
```

ChangeLVL.cs:

```
using UnityEngine;
```

```
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class ChangeLVL : MonoBehaviour
{
    private Button button;
    [SerializeField] private bool thisLvl;
    [SerializeField] private bool nextLvl;
    [SerializeField] private Scene scene;

    private void Start()
    {
        button = GetComponent<Button>();
        button.onClick.AddListener(ChangeLvl);

        if (thisLvl) scene =
(Scene)SceneManager.GetActiveScene().buildIndex;
        else if (nextLvl)
        {
            scene =
(Scene)SceneManager.GetActiveScene().buildIndex + 1;

            if ((int)scene >=
SceneManager.sceneCountInBuildSettings)
                scene = Scene.MainMenu;
        }
    }

    private void OnDestroy()
    {
        button.onClick.RemoveAllListeners();
    }

    public void ChangeLvl()
    {
        SceneManager.LoadScene((int)scene);
    }
}

public enum Scene
{
    MainMenu,
    First,
    Second,
    Third,
    Fourth,
}
```

ChangeMenuStatus.cs:

```
using UnityEngine;
using UnityEngine.UI;

public class ChangeMenuStatus : MonoBehaviour
{
    private Button button;
    [SerializeField] private GameObject menu;
    void Start()
    {
        button = GetComponent<Button>();
        button.onClick.AddListener(UseLvlMenu);
    }

    private void UseLvlMenu()
    {
        menu.SetActive(!menu.activeInHierarchy);
    }

    private void OnDestroy()
    {
        button.onClick.RemoveAllListeners();
    }
```

```
}
```

ExitButton.cs:

```
using UnityEngine;
using UnityEngine.UI;

public class ExitButton : MonoBehaviour
{
    private Button button;
    void Start()
    {
        button = GetComponent<Button>();
        button.onClick.AddListener(Quit);
    }

    public void Quit()
    {
        Application.Quit();
        Debug.Log("Quit");
    }
    private void OnDestroy()
    {
        button.onClick.RemoveAllListeners();
    }
}
```

Pause.cs:

```
using UnityEngine;
using UnityEngine.UI;

public class Pause : MonoBehaviour
{
    private Button button;
    private static bool isPause = false;
    [SerializeField] private GameObject menu;
    void Start()
    {
        button = GetComponent<Button>();
        button.onClick.AddListener(ChangeTimeScale);
    }
    private void ChangeTimeScale()
    {
        Time.timeScale = isPause ? 1 : 0;
        isPause = !isPause;
    }

    private void OnDestroy()
    {
        button.onClick.RemoveAllListeners();
    }
}
```

TowerUIControllser.cs:

```
using UnityEngine;

public class TowerUIControllser : MonoBehaviour
{
    [HideInInspector] public PlatformManager platform;

    public void ChooseTower(Towers tower)
    {
        platform.ChooseTower(tower);
    }

    public void RemoveTower()
    {
        platform.RemoveTower();
    }
```

ДОДАТОК Б

Графічна частина

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

РОЗРОБКА ГРИ У ЖАНРІ TOWER DEFENSE З ВИКОРИСТАННЯМ ДВИГУНА UNITY НА МОБІЛЬНУ ПЛАТФОРМУ ANDROID

Виконав студент 4 курсу
групи ПД-42
Мидниський О.І.
Керівник роботи
Дібрівний О.А.

Київ – 2022

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

В даний час ринок мобільних ігор не стоїть на місці та активно розвивається. Однак якість більшості ігор, що випускаються, залишає бажати кращого, і користувачі вибирають саме якісні ігри. Програмісту необхідно правильно розробити архітектуру програми, яка буде зрозуміла іншим програмістам та зможе легко розширюватись.

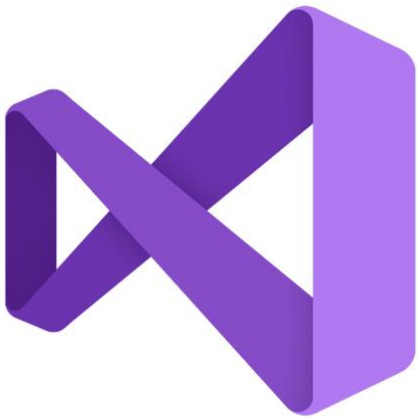
- **Мета роботи**: виявлення недоліків та проблем при проектуванні та розробці мобільного ігрового додатку на базі Android платформи.
- **Об'єкт дослідження:** створення тривимірного ігрового додатку у жанрі «tower defense» для Android.
- **Предмет дослідження:** методи, патерни та технології розробки ігрового програмного забезпечення для Android платформи.

АНАЛОГИ

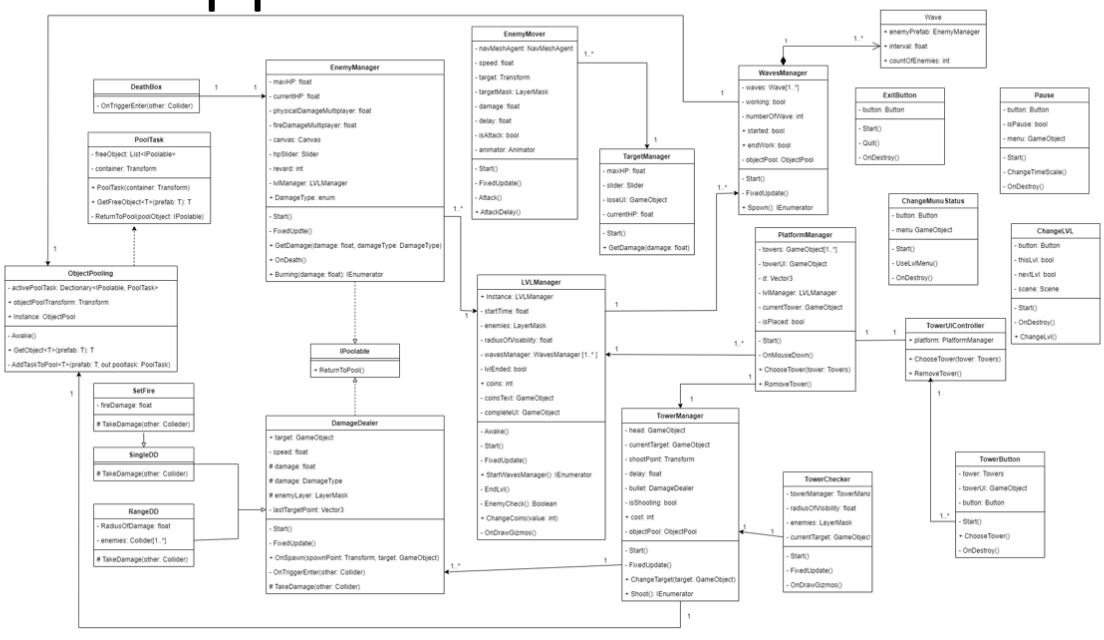
ТЕХНІЧНІ ЗАВДАННЯ

- 1. Провести аналіз предметної області.
- 2. Дослідити рушії розробки ігор, мови програмування, необхідні для них, та середовища розробки програмного коду.
- 3. Розробити архітектуру ігрового мобільного додатку.
- 4. Створити програмний продукт.
- 5. Провести тестування.

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ



МЕТОДИ ТА КЛАСИ ПРОГРАМИ



АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

- Newzoo's Mobile Trends to Watch in 2020. [Електронний ресурс] – Ресурс доступу: https://newzoo.com/cn/articles/newzoos-mobile-trends-to-watch-in-2020/.
- Unity - Manual: System requirements for Unity 2020 LTS. [Електронний ресурс] – Ресурс доступу: https://docs.unity3d.com/Manual/system-requirements.html
- C# Language Specification (4th ed.). [Електронний ресурс] – Ресурс доступу: https://www.ecma-international.org/publications-and-standards/standards/ecma-334/
- Stack Overflow Developer Survey 2021 - Integrated Development Environment, 2021. [Електронний ресурс] – Ресурс доступу: https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-integrated-development-environment
- Unity – Scripting API: ObjectPool<T0>. [Електронний ресурс] – Ресурс доступу: https://docs.unity3d.com/2022.1/Documentation/ScriptReference/Pool.ObjectPool_1.html

ВИСНОВКИ

1. Було досліджено явище відеоігор, їх історію та особливості розробки, а також виявлено суть жанру «tower defense» та його ігрового процесу.
2. Було проведено аналіз існуючих програмних засобів та інструментів для реалізації проєкту. Серед ігрових рушіїв був обраний Unity. Для скриптингу використовується C#. При виборі середовища розробки я зупинився на Visual Studio.
3. Була спроектована архітектура ігрового додатку. Програмна частина проєкту складається із 23 файлів: 22 класів і одного інтерфейсу.
4. Був створений програмний продукт. Скрипти можна поділити на 6 частин: логіка самого рівня, логіка ворогів, реалізація патерну Object Pooling, логіка оборонних веж, нанесення шкоди ворогам, інтерфейс користувача.
5. Було проведене тестування, в результаті якого було визначено, що програма відповідає поставленим вимогам.

ДЯКУЮ ЗА УВАГУ!