РЕФЕРАТ

Пояснювальна записка: 68 с., 8 рис., 1 табл., 5 дод., 21 джерел.

Об'єкт розробки: комп’ютерна 2D гра, розроблена в середовищі UNITY 3D.

Метою дипломного проекту є аналіз доступних технологій та методів розробки для створення ігрових програм різновиду ігор-аркад та розробка власної комп’ютерні 2D гри в середовищі UNITY 3D.

У вступі розглядається аналіз та сучасний стан проблеми, конкретизується мета кваліфікаційної роботи та галузь її застосування, наведено обґрунтування актуальності теми та уточнюється постановка завдання.

У першому розділі проведено аналіз предметної області, визначено актуальність завдання, практичне значення та призначення розробки, розроблена постановка завдання, задані вимоги до функціональних характеристик, програмної реалізації, використаних технологій та програмних засобів, забезпечення інформаційної безпеки та програмної сумісності розроблювального продукту.

У другому розділі виконано огляд та аналіз існуючих рішень, обрано вибір методів, мови та платформи для розробки, виконано проектування і розробка дизайну моделей та створено програму, наведено опис алгоритму і структури функціонування програми, визначені та обґрунтовані вхідні та вихідні дані, наведені характеристики складу параметрів технічних засобів, описаний виклик та завантаження програми, описана робота програми та розроблені правила гри.

В економічному розділі визначено трудомісткість розробленої інформаційної системи, проведений підрахунок вартості роботи по створенню програми та розраховано час на його створення.

Практичне значення розробленого проекту полягає в створенні повноцінного працеспроможного додатка, що надає можливість реалізувати гру-аркаду, в якій використовується сучасний графічний дизайн, невибагливі правила гри, зміна налаштувань під конкретного гравця, та який призначений для розваги користувачів.

Актуальність розробленого проекту полягає в тому, що в наш час є доцільним використання комп’ютерних відеоігор для розваг, а саме ця програма за допомогою сучасних методів розробки в середовищі UNITY 3D реалізує гру популярного напряму – аркаду в двовимірному просторі, а ідеї, закладені в її основу, також можуть бути в подальшому використані для створення нових ігор аналогічного спрямування.

Список ключових слів: ГРА, АРКАДА, РОЗВАГИ, ДИЗАЙН, СМАРТФОН, АЛГОРИТМ, ОПТИМІЗАЦІЯ, ПРОЕКТУВАННЯ, ІГРОВИЙ ПРОСТІР, ПРАВИЛА.

ABSTRACT

Explanatory note: 68 p., 8 fig., 1 table., 5 app., 21 sources.

Object of development: computer 2D game developed in UNITY 3D environment.

The aim of the diploma project is to analyze the available technologies and development methods for creating game programs of arcade games and to develop your own computer 2D game in the UNITY 3D environment.

The introduction considers the analysis and the current state of the problem, specifies the purpose of the qualification work and the scope of its application, provides a justification for the relevance of the topic and clarifies the problem.

The first section analyzes the subject area, determines the relevance of the task, the practical significance and purpose of development, developed the task, set requirements for functional characteristics, software implementation, technologies and software, information security and software compatibility of the developed product.

The second section reviews and analyzes existing solutions, selects methods, languages and platforms for development, designs and develops model designs and creates a program, describes the algorithm and structure of the program, identifies and justifies input and output data, provides characteristics of the parameters technical means, describes the call and download of the program, describes the operation of the program and developed the rules of the game.

The economic section determines the complexity of the developed information system, calculates the cost of work to create a program and calculates the time for its creation.

The practical significance of the developed project is to create a full-fledged workable application that provides an arcade game that uses modern graphic design, unpretentious rules of the game, changing settings for a particular player, and which is designed to entertain users.

The relevance of the developed project is that nowadays it is advisable to use computer video games for entertainment, namely this program with the help of modern development methods in the UNITY 3D environment implements a game of a popular direction - arcade in two-dimensional space, and the ideas embedded in it basis, can also be used in the future to create new games of a similar direction.

Keyword list: GAME, ARCADE, ENTERTAINMENT, DESIGN, SMARTPHONE, ALGORITHM, OPTIMIZATION, DESIGN, GAME SPACE, RULES.

ЗМІСТ

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ

ГГ – головний герой;

ІС – інформаційна система;

ЛКМ – ліва кнопка миші;

ПКМ – права кнопка миші;

ОС – операційна система;

ПЗ – програмне забезпечення;

ПК – персональний комп’ютер;

NPC – Non-Player Character, неігровий персонаж.

ВСТУП

Завдання даної кваліфікаційної роботи та об'єкт його діяльності безпосередньо пов'язані з напрямом підготовки та відповідає узагальненій тематиці кваліфікаційних робіт і переліку зазначених виробничих функцій, типових задач діяльності, умінню та компетенціям, якими повинні володіти бакалаври напряму 121 «Інженерія програмного забезпечення».

Останні 60 років комп'ютерні технології розвивалися дуже стрімко, і якщо вірити закону Мура, то вони кожні два роки стають майже в два рази потужніший. Але розвиток одного провокує розвиток всього похідного. Так і є з ігровою індустрією. Причому не тільки ігри стають краще, а весь світ, пов'язаний з ними, розвивається і щороку дивує своїм розвитком.

Комп'ютерні ігри придбали небувалу популярність за останні роки і зайняли почесне місце на ринку розваг і дозвілля. Повсякденне життя неможливо уявити без постійного припливу інформації. Для задоволення цих потреб використовуються сучасні засоби мультимедіа: телебачення, Інтернет, комп'ютерні ігри. А найпопулярнішим стає жанр аркада, який пройшов майже весь час розвитку до сих пір стоїть на першому місці серед жанрів. З одного пікселя на екрані впливу на весь світ.

Розробка міні-ігор є складним процесом, який допомагає вдосконалити навички роботи в розробці додатків. Для створення гри необхідно продумати компоненти гри і її логіку, а саме процес взаємодії гри з користувачем, умови виграшу, цілісності і т. п.

Метою дипломного проекту є аналіз доступних технологій і методів розробки для створення ігрових програм жанрового різновиду ігор-аркад і розробка власної комп'ютерні 2D гри в середовищі UNITY 3D.

Для досягнення поставленої мети, розробнику необхідно досліджувати наступні завдання:

1. Проаналізувати жанр відеоігор - аркада.
2. Детально розібратися з властивостями аркадних комп'ютерних ігор.
3. Висунути вимоги до власної гри.

4. Створити дизайн ігрового оформлення.
5. Створити програму реалізацію власної комп'ютерної гри «Orbs World».

В результаті виконання роботи повинна бути реалізована 2D гра під назвою «Orbs World» в середовищі UNITY 3D.

Для її реалізації необхідно виконати наступні етапи проектування:

- візуалізація ігрового процесу
- взаємодія з користувачем
- логіка гри

Практичне значення розробленого проекту полягає в створенні повноцінного працездатного додатки, який дозволяє реалізувати гру аркаду, в якій використовується сучасний графічний дизайн, невибагливі правила гри, достатня оптимізація, і який призначений для розваги користувачів.

Актуальність розробленого проекту полягає в тому, що в наш час доцільно використання комп'ютерних відеоігор для розваг, а саме ця програма за допомогою сучасних методів розробки в середовищі UNITY 3D реалізує гру популярного жанру - аркада в двовимірному просторі, а ідеї, закладені в її основу, також можуть бути в подальшому використані для створення нових ігор аналогічного напрямку.

Дана програма-гра може використовуватися користувачами для розважальних заходів і приємного відпочинку за комп'ютером.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Загальні відомості з предметної області

1.1.1. Визначення терміну “відеогра”

Відеогра (англ. Video game)[1] - різновид гри, яка ведеться шляхом взаємодії гравця і електронної системи, в якій пристрій візуального відображення (дисплей) є основним механізмом, що забезпечує зворотний зв'язок з гравцем. У вузькому сенсі словом «відеогра» (комп'ютерна гра) називають програму для електронної обчислювальної машини, створену спеціально для ігрових цілей.

Інші визначення терміну:

- комп'ютерна гра, відтворена на екрані телевізора за допомогою ігрової приставки.
- відеоігрової пристрій (предметом якого є відеогра) - побутової радіоелектронний апарат, призначений для відображення ігрової інформації на екрані телевізійного приймача або побутового відеомонітора.

Як правило, кажучи про відеоігри, мають на увазі цифрові відеоігри. До них також відносяться аналогові електронні ігри, в яких ігровий процес управляється не цифровими, а аналоговими схемами. Залежно від виду електронної ігрової системи (названої “платформа”), для відеоігор використовуються наступні пристрої:

- комп'ютери (комп'ютерні ігри. Іноді цим терміном розширено називають всі відеоігри),
- телеприставки (ігрові консолі),
- ігрові автомати,
- портативні ігрові пристрої,
- мобільні телефони.

Кишенькові мікропроцесорні електронні ігри, що мають сегментовані екрани, як відеоігри зазвичай не розглядаються.

Разом з тим, в даний час терміни “комп'ютерна гра” і “відеогра” можуть вживатися як синоніми і взаємно замінюватись

В українській термінології теж не існує узгодженої думки в цій сфері. Терміни «відеогра», «комп’ютерна гра» чи «гра» вживаються в одному значені, коли мова йде про їх локалізацію. В даній роботі використовується термін «відеогра», оскільки, згідно із визначенням у словнику Oxford – це гра, яка відбувається через керування візуальними образами на моніторі комп’ютера чи іншому дисплеї [2].

Найпростіша класифікація ПК-ігор ділить їх на:

1. квести – герой виконує завдання, спілкуючись з іншими персонажами, шукаючи і використовуючи різні предмети, вирішуючи завдання і головоломки;
2. рольові ігри (RPG) – гравець може покращувати здібності свого персонажа, перемагаючи ворогів та виконуючи завдання, а також продавати і купувати ресурси;
3. стратегії – в них потрібно управляти не персонажем, а внутрішньоігровим процесом, найчастіше, розвитком міста/країни/раси;
4. симулятори – імітують управління на автомобілі, літаку, кораблі чи космічному апараті, або ж спортивну гру;
5. логічні – підводять гравця до послідовних роздумів, змушують нестандартно мислити і вибудовувати логічні ланцюжки;
6. азартні – гемблінг.

Ще один важливий критерій класифікації ігор, крім жанрової приналежності – інтегровані в них системи монетизації. Те, що розробники хочуть отримувати прибуток від своєї роботи – це природно, і вибирають вони для цього різні способи. За системою монетизації ігри бувають:

- Платні (Paid). Покупець платить за копію гри один раз, отримуючи повний обсяг контенту. Можливе придбання нових доповнень.
- Умовно-безкоштовні (Freemium, Shareware). Частина ігрового контенту надається безкоштовно, якщо гравець хоче отримати більше – платить.

- З періодичної підпискою (Subscribe). Щоб мати доступ до гри, потрібно регулярно оплачувати підписку (раз на місяць/рік, ін.), При цьому, поновлення та доповнення до таких ігор, частіше за все, доступні безкоштовно.
- Безкоштовні з внутрішньоігровими покупками і/або рекламою (Free-to-Play). Такі ігри умовно безкоштовні, але, зазвичай, щоб мати можливість на рівних змагатися з іншими гравцями, доводиться платити. У деяких подібних іграх внутрішньоігрові покупки настільки необхідні, що їх часом називають Pay-To-Win (плати, щоб перемагати).

1.1.2. Історія відеоігор

Перші відеоігри використовували безліч електронних пристроїв для зв'язку з користувачем. Так, першою зареєстрованою грою, яку можна віднести до відеогри, є Ракетний симулятор, інтерфейсом якого виступало аналогове пристрій на базі електронно-променевої трубки (ЕПТ) (заявка на патент була подана 25 січня 1947 року). Створення цієї гри було обумовлено винаходом ЕПТ, що дозволяло формувати зображення на екрані. Однак, через те, що Ракетний симулятор не мав реєстрового типу зображення, яке характерно для моніторів і телевізорів, то даний пристрій частіше приймають за електронну гру, але відносять до відеоігор. Але разом з тим, з технічної точки зору, Ракетний симулятор є відеогрою.

Ранніми прикладами відеоігор є перша програмна гра OXO, перша створена для розваги Tennis for Two, Spacewar! як перша відеогра, яка була доступна поза розробила організації що її розробила. Визначення відеоігри для ігор цього часу має свої труднощі, наприклад, дані ігри не створювалися як інтерактивні програми, і не розроблялися як ігри. При цьому, вони не були популярними і масовими, тому що не тиражувалися і не виходили за межі тих організацій, де вони були створені. Це обумовлювалося рядом складнощів, коли комп'ютери 1960-х займали кімнати, використовували радіолампи, для яких потрібно охолодження і постійне обслуговування. Одна з революцій в комп'ютерних іграх

полягала в переході на транзисторную елементну базу і масове використання телетайпа. Тут одним із наслідків стало розмежування ігор по інтерфейсу взаємодії з користувачем: відеоігри та текстові ігри.

Історично відеоігри з'явилися раніше, ніж був створений персональний комп'ютер. Це обумовлено тим, що в 1960-х роках комп'ютерні технології не були масовими і не розглядалися як серйозний інструмент у цивільній сфері. Створювані відеоігри на електронній базі (електронні ігри) зіграли важливу роль в мирній соціалізації комп'ютерної техніки і її інтеграції в популярну культуру. У 1970-х відеоігри стали одним з найпопулярніших розваг, коли встановлювалися ігрові автомати в барах, кафе та інших закладах. Роль попередніх ігрових автоматів, електронних ігор і відеоігор на формування комп'ютерної ігрової індустрії колишній продюсер CNN Стівен Бакстер описує так:

«Не можна сказати, що відеоігри виросли з пинбола, але ви можете припустити, що відеоігри не могли б з'явитися без нього. Це як велосипеди і автомобілі. Одна індустрія призводить до того, що з'являється інша, і потім вони співіснують разом. Але вам доведеться спочатку винайти велосипед для того, щоб згодом з'явилися легкові автомобілі.»

Перша відеогра в сучасному розумінні була розроблена Баєром Ральфом, коли нововведення полягала в тому, що гри почали використовувати екран телевізора. Момент, коли ідея прийшла йому в голову, він згадує наступним чином:

«Під час відрядження я сидів поруч зі східним автобусним терміналом у Нью-Йорку, і думав про те, що можна зробити з телевізором крім того, як перемикати канали, які не хочеться дивитися. І тоді мені в голову прийшов концепт створення відеоігор, з продажем десь за ціною $ 19.95. Це було в 1966 році, в серпні.

З цього моменту ти, мабуть, зрозумів, що я є керівником відділу, і я розпоряджаюся фондом заробітної плати, який становить 7-8 мільйонів доларів. Я можу взяти пару хлопців і дати їм завдання щоб працювати над чимось, немає

потреби кому-небудь знати про це, тому що це практично не позначиться на моїх витратах. І саме так я почав.»

Баєр Ральф працював спільно з Біллом Харіссоном, і вони розробили першу ігрову приставку - пристрій, який передавав зображення на екран телевізора[2]. Основні проблеми були технічні, і їх розробки не представляли розважальної цінності. Пізніше до них приєднався Білл Руш, який першим усвідомив необхідність ігрового фана (англ. Fun, або задоволення від гри). Однак, в кінці 1960-х через фінансові проблем роботи були згорнуті[3].

Творцем індустрії відеоігор вважається Нолан Бушнелл, який після розробки декількох комп'ютерних ігор створив відеогру Computer Space. У 1971 році він, спільно з Nutting Associates, Випустив 1500 ігрових автоматів з даною грою. Однак, через погані продажі Computer Space, Нолан Бушнелл покинув Nutting Associates і разом з Тедом Дабні заснував фірму Atari. У цій компанії була розроблена відеогра Pong, яка є першою комерційно успішною відеогрою.

Відеоігри в 1960-х - 1970-х роках випускалися на спеціальних телеприставки і у вигляді ігрових автоматів[4]. Вважається, що тільки з початку 1980-х, коли персональні комп'ютери стали доступні досить багатьом людям, відеоігри починають розглядатися як комп'ютерні.

1.1.3. Ігровий процес

Ігровий процес — це термін, яким називають особливості взаємодії людини із грою, найчастіше відеогрою. Ці особливості створюються за допомогою правил, завдань та способів їх вирішення, які пропонує гра. Поряд з фабулою та технологією, ігровий процес є однією з трьох основних складових будь-якої відеогри[5].

Основою ігрового процесу будь-якої відеогри є повторюваний ланцюжок дія-реакція-зворотний зв'язок.

1. Дія — інформація, яку гравець надає грі з ігрових контролерів, такі як геймпад чи клавіатура;
2. Реакція — відповідь гри на дії гравця, сформована на основі наданої ним

інформації, але ще не виражена для нього в доступній формі;

3. Відгук (англ. Feedback) — інформація, отримувана з гри гравцем. Поділяється на два основних види:
   - явний (є очевидною відповіддю на дії гравця),
   - неявний (служить фоном чи інформує гравця). [6]

Зворотний зв'язок має свої різновиди в межах явного і неявного:

- візуальний (бачене гравцем на екрані),
- звуковий (чуте через пристрої відтворення звуку),
- дійовий (подія в самій грі — відповідь на дії гравця, зазвичай поєднання зображення зі звуком),
- неігрових персонажів (відповідь персонажів, що «населяють» гру, на дії гравця),
- акумулятивний (виражений певним чином прогрес гри),
- емоційний (емоції, які викликає гра),
- здійснення (вираження сенсу окремих завдань і всієї гри, їх завершення),
- інформативний (відомості, які гравець отримує в контексті грання).

Отриманий зворотний зв'язок спонукає до нових дій і так до завершення гри[7].

Загалом для ігор їхній процес найчастіше поділяється за кількістю учасників. Відповідно існують одноосібний, багатоосібний, командний.

Щодо відеоігор виділяються різновиди:

За кількістю гравців:

- одноосібний (англ. Singleplayer, однокористувацький, синглплеєр) — для кожної копії гри існує один гравець;
- кооперативний (англ. Cooperative gameplay) — двоє і більше гравців за допомогою різних ігрових контролерів, підключених до одного пристрою, або через комп'ютерну мережу, грають разом задля досягнення спільної мети;
- багатоосібний (англ. Multiplayer, багатокористувацький,

мультиплеєр) — за допомогою різних ігрових контролерів, підключених до одного пристрою, або через комп'ютерну мережу, участь у грі беруть два і більше гравців. Вони можуть як діяти задля спільної мети, поокремо чи в команді, так і бути суперниками;

- асиметричний (англ. Asymmetric gameplay) — різновид багатоосібного, за якого можливості гравців, як суперників, так і союзників, свідомо нерівні.

За лінійністю:

- лінійний (англ. Linear gameplay) — всі цілі й шляхи їх досягнення наперед визначені. Кожен гравець тільки виконує ті самі необхідні дії;
- нелінійний (англ. Nonlinear gameplay) — різні гравці можуть досягати поставлених цілей різними шляхами. Самі цілі можуть генеруватися, відповідно до дій гравця;

Окремо виділяється емерджентний ігровий процес (англ. Emergent gameplay) — він поєднує різні види дій гравця, утворюючи такий ігровий процес, що має нові якості, котрих не мають його складові окремо.

1.1.4. Ігрові жанри

Індустрія розробки та реалізації відеоігор відрізняється широкою різноманітністю жанрів і, відповідно, провідних представників для кожного виду гри. Найпопулярніші жанри відеоігор та їх представники включають в себе:

1. MOBA (Multiplayer Online Battle Arena)

Це командна гра, яка поєднує риси екшен, RPG і стратегії в реальному часі. Гравець управляє персонажем з унікальними здібностями, який бореться в одній із двох команд-суперників. Для перемоги потрібно знищити базу противника.

Найпопулярніші ігри жанру: Dota 2 і League of Legends (LoL).

2. FPS (First-Person Shooter)

У шутерах від першої особи ігровий процес ґрунтується на боях із використанням вогнепальної або будь-якої іншої зброї. Від інших шутерів вони

відрізняються тим, що гравець сприймає те, що відбувається, очима свого персонажа.

Найпопулярніші представники: Counter-Strike: Global Offensive (CS: GO), Tom Clancy's Rainbow Six Siege, Overwatch, Valorant.

3. MMORPG (Massively Multiplayer Online Role-Playing Game)

У масових багатокористувацьких рольових онлайн-іграх гравці активно взаємодіють один з одним в єдиному світі і виконують певні завдання. Виник, як піджанр RPG.

Найвідоміша гра : World of Warcraft.

4. Battle Royale

Жанр королівської битви виник нещодавно, він поєднує в собі елементи MMO і симулятора виживання з режимом last man standing. Безліч гравців з'являється на карті, яка з часом зменшується. Виграє той, чий персонаж залишається останнім вижившим.

Ігри жанру: Player Unknown's Battlegrounds (PUBG), Apex Legends, Fortnite.

5. Спортивні симулятори

Симулятори спортивних змагань бувають різними. Найбільш популярні – з командних ігор, таких як футбол і баскетбол. Також існують симулятори боксу, гонок, більярду, та ін.

Найбільш розповсюджені ігри: FIFA, NBA, F1.

6. CCG (Collectible Card Game)

Колекційні карткові ігри в останні роки частково розгубили популярність, але все ще мають велику аудиторію. У них гравці збирають власні колекції карт, що виходять у окремих наборах, і складають із них свої колоди. Цими колодами вони борються між собою.

Представники: Hearthstone, Magic: The Gathering Arena.

7. Auto battler

Автобатлери виникли, як модифікація для MOBA. Вісім гравців б'ються один з одним, формуючи свої армії і покращуючи юнітів. Чотири гравці, які найдовше протримаються, виграють.

Популярні представники: Hearthstone Battlegrounds, Teamfight Tactics, Dota Underlords.

1.1.5. Аркада як ігровий напрямок

Аркадна гра — поширений в індустрії відеоігор термін, що позначає ігри з навмисно примітивним ігровим процесом. Деякі ресурси про відеоігри виділяють їх як окремий жанр і зараховують до них платформери.

У світовій практиці, аркадами називаються ігри для аркадних ігрових автоматів. Це вважається не окремим жанром ігор, а ігровим напрямком. Відеогра називається «аркадною» в тому випадку, якщо вона прямо портована з автомата або ж схожа по концепції з іграми для автоматів.

Перші ігрові автомати працювали після вкидання монет або жетонів. Найбільш ранні аркадні автомати використовували механічний рахунок, і лише 1970 з'явився електронний. Першою аркадною грою цього типу була Galaxy Game, розроблена в 1971 році.

Види аркад (не слід їх плутати з іграми для аркадних ігрових автоматів)[8]:

1. Arcade Racing

Аркадні гонки характеризуються вкрай примітивним і до межі спрощеним геймплеєм, які не мають нічого спільного з реальністю, управлінням і поведінкою автомобілів.

Представники: Wacky Wheels, Street Racing Syndicate.

2. Classic Arcade

Суть класичних аркад пояснити досить складно. Зазвичай головною метою є проходження рівня за максимально короткий проміжок часу, збір всіх бонусів на рівні та отримання максимальної кількості очок. Сюди ж можна віднести різноманітні ігри типу Breakout (Arkanoid) і Пінбол.

Представники: Pac-Man, Digger, Battle City, Qix, Xonix,

3. Fighting (Wrestling)

У файтингах (бійках) два персонажа б'ються на арені, застосовуючи різні удари, кидки та комбінації. Характеризується великою кількістю персонажів

(бійців) і ударів (іноді понад сто для кожного персонажа). Жанр не популярній на PC через орієнтацію на спільну гру, а на клавіатурі досить проблематично одночасно грати вдвох. Проте добре розвинений на ігрових приставках. За деякими іграм цього жанру навіть проводяться світові чемпіонати. Найчастіше в деяких іграх на арені можуть зійтися і чотири противника одночасно, наприклад в Guilty Gear Isuka. Для управління в таких іграх рекомендується геймпад.

Представники: Mortal Kombat, Street Fighter, Tekken, Guilty Gear.

4. Platformer

Поняття платформерів прийшло з ігрових приставок (консолей). Саме там цей жанр найбільш популярний. Основним завданням гравця є подолання перешкод (ям, шипів, ворогів і т. д.) за допомогою стрибків. Найчастіше доводиться стрибати по абстрактно розставленим в повітрі «паличкам» (т. з. платформам), звідси і пішла назва жанру.

Представники: Mega Man, Super Mario Bros, Sonic the Hedgehog, Disney's Aladdin, Captain Claw, I wanna be the guy, Super Meat Boy, Jazz Jackrabbit, Shoot'em up.

5. Shoot 'em up

У шутемапах гравцеві пропонується знищувати ворогів та збирати бонуси, які з'являються на екрані, щодо аркадних ігор, зазвичай з постійною прокруткою рівня в одному напрямі. По напрямку руху розрізняють вертикальні (знизу вгору) і горизонтальні (зліва направо) скролери. Жанр був дуже популярний в середині 90-х років, у 2000-х - 2010-х скролери практично не випускаються. Із сучасних ігор цього жанру можна відзначити Jets'n'Guns і серію Touhou, приклади інших ігор:

Представники: AirStrike 3D, DemonStar, KaiJin, Sea Dragon.

6. Virtual Shooting

Віртуальний тир зародився на ігрових автоматах, згодом перейшов на інші ігрові платформи, включаючи ПК. Ігровий процес представляє відстріл ворогів котрі несподівано з'являються, але на відміну від екшенів ми не можемо керувати рухом гравця або камерою, всю гру ми ніби їдемо по «рейках». У зв'язку з цим

іноді роблять відеотіри, тобто всю гру знімають на відеокамеру, в певних місцях підставляючи різні варіанти відео уривків. Також «віртуальним тиром» сучасні гравці іноді жартома називають одноманітні і прості за складністю шутери (наприклад Soldier of fortune 3: Payback).

Представники: Mad Dog McGee, Серія House of the Dead.

7. Інші аркади

Ігровий процес в них підпадає під визначення «аркади», вони, як правило, дуже динамічні, а метою в них є збір всіх бонусів або набір максимально можливої кількості очок. Однак від представників інших різновидів цього жанру їх зазвичай відрізняють будь оригінальні знахідки розробників, що не дозволяють приписати ці ігри до однієї з перерахованих вище категорій.

Представники: Icy Tower, Elasto Mania.

1.1.6. Кіберспорт

Кіберспорт — це командне або індивідуальне змагання на основі відеоігор[9]. У Росії визнаний офіційним видом спорту[10].

Усі кіберспортивні дисципліни діляться на кілька основних класів, що розрізняються властивостями просторів, моделей, ігрових завдань і необхідними ігровими навичками кіберспортсменів: шутери від першої особи, стратегії в реальному часі, спортивні симулятори, автосимулятори, авіасимулятори, файтинг, командні рольові ігри з елементами тактико-стратегічної гри, тощо.

У кіберспорті загальноприйнятою практикою є наявність призових фондів, які можуть досягати декількох мільйонів доларів США. Турнір з Dota 2 “The International” кілька разів бив рекорди по виплатах: так, в 2017 було розіграно $ 25 млн, в 2018 - $ 26 млн, в 2019 - $ 34 млн.

Історія кіберспорту почалася з гри Doom 2, яка мала режим мережевої гри через локальну обчислювальну мережу. У 1996 році з'являється EVO [11] (спочатку - Battle by the Bay), який проводить турнір по грі Street Fighter II.

Пізніше, в 1997 році в США з'явилася перша ліга кіберспортсменів - Cyberathlete Professional League (CPL), яка зробила перший турнір в дисципліні

Quake. Ця було яскравою подією у сфері кіберспорту, оскільки до того як з'явилися офіційні кіберспортивні ліги, мали місце лише невеликі змагання в іграх, де проводився підрахунок балів. (Spacewar, Asteroids, Space Invaders і т. д.).

Змагання з кіберспорту проводяться по всьому світу, в тому числі і міжнародні. Найбільш значущим і аналогом Олімпійських ігор[12] був міжнародний турнір World Cyber Games (WCG), який проводився в різних країнах з 2000 по 2013 рік. Крім WCG регулярно проводяться Cyberathlete Professional League і Electronic Sports League.

На сьогоднішній день найбільшими і престижними змагання є ті, які проводять самі виробники ігор: наприклад турнір The International по Dota 2 або Чемпіонат світу з League of Legends.

Вимоги до проведення кіберспортивних змагань викладені у Правилах спортивних змагань з кіберспорту, що затверджені Міністерством молоді та спорту України №2/5, 3/21 від 26.01.2021. Крім протоколу, який підтверджує факт проведення турніру, обов'язкова наявність бригади суддів, яка складається з головного судді і суддів по іграх (якщо їх декілька), лінійних суддів, спостерігачів та технічних суддів. Створення нормативних актів, які регламентують діяльність спортсмена, тренера і судді при реалізації процесу спортивної підготовки і проведення змагань в дистанційному режимі - є актуальним нині завданням.

У киберспорті кількість гравців в команді варіюється в залежності від типу дисципліни: кількість людей у команді може бути від 1 до 15.

Зазвичай команда кіберспортсменів має наступний склад:

1. Прогеймери

Професійні гравці, які грають за гроші. Основним заробітком прогеймерів є призові фонди і зарплати за гру на змаганнях з кіберспорту. У той час як прогеймери фінансово залежні від ігор, проведення свого часу у грі часто не вважається гравцями «вільним» і може приносити менше задоволення, так як гра для нього стає роботою. У країнах Азії, зокрема в Південній Кореї і Японії, прогеймери спонсоруються великими компаніями і можуть заробляти більше $

100 000 на рік. У США Major League Gaming уклала контракт з Electronic Sports Gamers на $ 250 000 щорічно.

2. Капітан кіберспортивної команди

Капітан - в кіберспорті він є офіційним лідером серед її гравців. Зазвичай це старший або найбільш досвідчений член команди або гравець, який може сильно вплинути на результат матчу.

3. Тренер кіберспортивної команди

Тренер - фахівець в певному виді кіберспортивної дисципліни, відповідальний за тренування команди кіберспортсменів. Тренер здійснює навчально-тренувальну роботу, спрямовану на виховання, навчання і вдосконалення майстерності, розвиток функціональних можливостей своїх підопічних. Також часто виробляє аналітику і аналіз патча і намагається зрозуміти мету в новому оновленні, в деяких випадках тренер є минулим про-ігроком.

У наш час загальноприйнятими є кібеспортивні дисципліни, вказані у таблиці 1.1

Таблиця 1.1

Кібеспортивні дисципліни

Дисципліна	Описання	Приклад ігор
Бойова арена	дисципліна, в якій 2 команди гравців борються один з одним з метою знищення головної будівлі команди суперника. Кожен гравець управляє одним зі списку доступних героїв, що відрізняються характеристиками. Протягом матчу герої можуть ставати сильнішими, отримувати нові здібності і спорядження;	Dota 2 League of Legends King of Glory Brawl Stars

Змагальні головоломки	спортивна дисципліна комп'ютерного спорту, що представляє собою вирішення логічних задач учасниками змагань, результат якого залежить від швидкості логічного мислення і кмітливості, що обумовлюють вибір тієї чи іншої стратегії гри для досягнення перемоги;	Hearthstone Artifact Heroes of the Storm Dota Auto Chess Dota Underlords
Спортивний симулятор	спортивна дисципліна комп'ютерного спорту, що відтворює за допомогою відеоігри спортивну гру на арені за правилами виду спорту, визнаного в установленому порядку;	NHL 21 Rocket League
Стратегія в реальному часі	спортивна дисципліна комп'ютерного спорту, в якій протиборчі сторони учасників змагань на арені в реальному часі позиціонують і маневрують ігровими персонажами, для захисту районів карти і / або знищення активів своїх суперників. В ході гри можуть створюватися додаткові ігрові персонажі і поліпшуватися властивості вже наявних;	Warcraft 3 StarCraft 2 Clash Royale Age of Empires: Definitive Edition Total War: WARHAMMER II Clash of Clans
Технічний симулятор	спортивна дисципліна комп'ютерного спорту, в якій учасники змагань, імітуючи фізичну поведінку і управління технічними засобами,	F1 2020 Gran Tourismo Sport iRacing Project CARS 2 Trackmania2 lagoon Assetto Corsa Competizione

	досягають перемоги відповідно до Технічних правил дисципліни. В результаті змагальної діяльності учасник змагань отримує навички управління реальними технічними засобами (наприклад: танковий симулятор, авіаційний симулятор, автомобільний симулятор);	
Військові	дисципліна комп'ютерного спорту, що імітує процес єдиноборства на арені за допомогою відеогри, в якій учаснику змагань необхідно знизити до нуля параметр енергії (здоров'я) об'єкта управління суперника за відведений час.	Mortal Kombat Tekken 7 Street Fighter V Super Smash Bros. Ultimate Marvel vs. Capcom Infinite

У зв’язку з вищеописаною популярністю відеоігор, широкою їхньою розповсюдженістю та стрімко наростаючим інтересом до тематики ігор та кіберспорту можна стверджувати, що розробка та ефективне впровадження відеоігор є актуальним завданням для сучасного інженера з програмного забезпечення. Так, тематика цього дипломного проекту відповідає попиту ринку на сьогоднішній день.

1.2. Призначення розробки та область застосування

Метою дипломного проекту є аналіз доступних технологій та методів розробки для створення програм напрямку аркадних ігор та розробка власної комп’ютерні 2D гри в середовищі UNITY 3D.

Останнім часом все частіше можна почути про збільшення масштабів

розробки програмного забезпечення розважального характеру. До числа таких продуктів відносяться відеоігри. Через десятки років з моменту свого започаткування, індустрія відеоігор зайняла важливе місце на ринку, поряд з іншими розвагами сфери мультимедіа, такими як кіно, мультиплікація, музика, тощо. Це сталося завдяки створенню багатомільярдних компаній, що займаються розробкою розважальних продуктів та програмних забезпечень, що складаються з мільйонів рядків програмного коду і файлів мультимедіа.

Ігри стали охоплювати величезну аудиторію по всьому світу, з'являючись на різних ігрових пристроях. Вони стали складніше і масштабніше: покращилася обробка візуального простору, обробка фізики об'єктів і штучний інтелект. Але напрямна методологія розробки не змінилася. Сучасні комп'ютерні ігри - великі і складні програмні комплекси. Витрати тільки на програмну розробку часто вимірюються сотнями людино-місяців. За обсягом задіяних технологій залучених коштів і рівня професійної підготовки розробників ігрова індустрія давно зайняла аж ніяк не останнє місце в світі IT.

Розробка апаратної бази в секторі персональних комп'ютерів вже досить давно стимулюється саме ігровою індустрією. Сучасні новітні технології, такі як ray tracing (трасування променів) та micropolygon (мікрополігонни) були б неможливі, якщо не грошовий обсяг що вкладається щорічно у ігрову індустрію.

Але, натомість не кожна багатомільйонна гра з проривною графікою стає хітом продаж та мрією гравців. Тож зараз, коли гіганти ігрової індустрії вкладають неймовірні бюджети, нерідко набагато успішними стають проекти інді студій (маленькі колективи до 15 працівників), або і взагалі розробників одинаків. Гравці можуть полюбити такі ігри як за цікавий сюжет, або нестандартні способи виведення інформації, так і за не перевантажений, або навмисно спрощений ігровий процес.

Тож в наш час є доцільним створення комп’ютерних відеоігор напряму аркад для розваг, і саме ця програма за допомогою сучасних методів розробки в середовищі UNITY 3D реалізує не складну у розумінні гру у двовимірному просторі, а ідеї, закладені в її основу, також можуть бути в подальшому

використані для створення нових ігор аналогічного спрямування.

Дана програма - гра може використовуватись користувачами для розважальних заходів та приємного відпочинку за комп’ютером.

1.3. Підстава для розробки

Відповідно до освітньої програми, згідно навчального плану та графіків навчального процесу, в кінці навчання студент виконує кваліфікаційну роботу.

Тема роботи узгоджується з керівником проекту, випускаючою кафедрою, та затверджується наказом ректора.

Отже, підставами для розробки (виконання кваліфікаційної роботи) є:

- освітня програма 121 «Інженерія програмного забезпечення»;
- навчальний план та графік навчального процесу;
- наказ ректора Національного технічного університету «Дніпровська політехніка» № 234-с від 27.04.2021 р;
- завдання на кваліфікаційну роботу на тему «Розробка комп’ютерної аркадної двовимірної гри у середовищі UNITY 3D».

1.4. Постановка завдання

Метою проекту є розробити ігровий додаток, що змусить розвивати абстрактне мислення та розуміння позиціювання в двовимірному просторі.

Даний ігровий додаток дозволить користувачу цікаво провести час, та розвивати просторове мислення.

1.5. Вимоги до програми або програмного виробу

1.5.1. Вимоги до функціональних характеристик

Для коректного запуску додатку потрібно встановити DirectX версії 11, на більш ранніх версіях додаток не може бути запущений, через набір використаних функцій у ігровому двигуні.

DirectX - це набір API, розроблених для вирішення завдань, пов'язаних з програмуванням під Microsoft Windows.

Оскільки проект вже запакований у виконувальний файл, від користувача потрібно тільки запустити його.

Вводом даних є взаємодія з мишею та клавіатурою. Виводом даних є відтворення усіх подій вводу на ігрового персонажа.

Також при запуску можливі оповіщення про можливі проблеми:

warning — нестандартна ситуація з програмою. Не являється критичною, але користувачеві варто приділити їй увагу;

error — помилка в роботі програми. Деякі функції або файли могли бути недоступні.

1.5.2. Вимоги до інформаційної безпеки

Оскільки додаток вже запакований, якихось вимог до інформаційної безпеки не потрібно, достатньо просто дотримуватись ієрархії внутрішніх файлів, для уникнення можливих збоїв додатку

1.5.3. Вимоги до складу та параметрів технічних засобів

Для більш надійного функціонування додатку потрібен DirectX версії 11 та досить сучасне обладнання графічної карти (версія драйверу, яка рекомендована — 398.63). Додаток, спакований на запуск у середі Windows, таким чином додаток не буде запускатись на інших платформах (IOS, Linux, тощо.)

1.5.4. Вимоги до інформаційної та програмної сумісності

Програма має бути написана за допомогою мови програмування C#, у середовищі UNITY 3D.

Створений програмний додаток має відповідати таким властивостям:

1. завершеність (completeness) - властивість, що характеризує ступінь володіння ПЗ усіма необхідними частинами і рисами, що вимагаються для виконання своїх явних і неявних функцій;
2. автономність (self-containedness) - властивість, що характеризує здатність ПЗ виконувати покладені на функції без допомоги або підтримки інших компонент програмного забезпечення;
3. комунікабельність (communicativeness) - властивість, що характеризує ступінь, в якій ПЗ полегшує завдання або опис вхідних даних, і здатність видавати корисні відомості в досить простій формі і з простим для розуміння змістом;
4. структурованість (structures) - властивість, що характеризує програми ПЗ з точки зору організації взаємопов'язаних їх частин в єдине ціле певним чином (наприклад, відповідно до принципів структурного програмування);
5. читабельність (readability) - властивість, що характеризує легкість сприйняття тексту програм ПЗ (відступи, фрагментація, форматування);
6. можливість масштабування (augmentability) - властивість, що характеризує здатність ПЗ до використання більшого обсягу пам'яті для зберігання даних або розширення функціональних можливостей окремих компонент;
7. можливість модифікацій (modifiability) - міра, що характеризує ПЗ з точки зору простоти внесення необхідних змін і доробок на всіх етапах і стадіях життєвого циклу ПЗ;
8. модульність (modularity) - властивість, що характеризує ПЗ з точки зору організації його програм з таких дискретних компонент, що зміна однієї з них надає мінімальний вплив на інші компоненти.

РОЗДІЛ 2
ПРОЕКТУВАННЯ ТА РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1. Функціональне призначення системи

Функціональне призначення додатку складається з показанням можливостей ігрового двигуна UNITY 3D та покращення планувальної логіки користувача.

2.2. Опис застосованих математичних методів

В даному мобільному додатку ні під час розробки, ні під час тестування та роботи додатка, не використовуються та не розглядаються ніякі особливо складні та/або комплексні математичні методи.

2.3. Опис використаної архітектури та шаблонів проектування

Написання програмного коду, навіть у навмисно спрощеній аркадній грі, потребує від програміста навичок по організації проекту, файлів з кодом, та інших аспектів розробки щоб полегшити подальше підтримку та розвиток проекту. Може здатися що програмісту потрібно самому вигадати інноваційних підхід, але цього зовсім не слід робити без особливої потреби. Набагато краще використовувати вироблені за останні 40 років принципи, патерни (шаблони) та архітектури, які спеціально були створені як одні з найоптимальніших і ефективних способів вирішення поставленого завдання. Шаблони проектування являють собою певний прийом вирішення певної архітектурної проблеми.

Тобто до них можна й потрібно відноситись, як до багаторічного досвіду проектування та побудови складних систем, що вийшов у набір основних прийомів та методів.

Патерни розрізняють за видами:

- породжуючі патерни;
- структурні патерни;

- патерни поведінки.

Для даного додатку було обрано класичний шаблон програмування MVC. Model-View-Controller (MVC, «Модель-Представлення-Контролер», «Модель-Вид-Контролер») - схема поділу даних програми, призначеного для користувача інтерфейсу і керуючої логіки на три окремих компоненти: модель, уявлення і контролер - таким чином, що модифікація кожного компонента може здійснюватися незалежно.

Модель відповідає за внутрішню логіку роботи програми. Тут ми можемо приховати способи зберігання даних, а також правила і алгоритми обробки інформації.

Наприклад, для однієї програми ми можемо створити кілька моделей. Одна буде налагодженням, а інша робочою. Перша може зберігати свої дані в пам'яті або в файлі, а друга вже задіює базу даних. По суті це просто патерн Стратегія.

Представлення відповідає за відображення даних моделі. На цьому рівні ми лише надаємо інтерфейс для взаємодії користувача з моделлю. Сенс введення цього компонента той же, що і у випадку з наданням різних способів зберігання даних на основі декількох моделей.

Наприклад, на ранніх етапах розробки ми можемо створити просте консольне уявлення для нашого застосування, а вже потім додати красиво оформлений GUI. Причому, залишається можливість зберегти обидва типи інтерфейсів.

Крім того, слід враховувати, що в обов'язки представлення входить лише своєчасне відображення стану моделі. За обробку дій користувача відповідає контролер, про яких ми зараз і поговоримо.

Контролер забезпечує зв'язок між моделлю і діями користувача, отриманими в результаті взаємодії з представленням. Координує моменти поновлення представлення. Приймає більшість рішень про переходах додатки з одного стану в інший.

Фактично на кожну дію, яке може зробити користувач в представленні, повинен бути визначений обробник в контролер. Цей оброблювач виконає

відповідні маніпуляції над моделлю і в разі необхідності повідомить Поданню про наявність змін.

При розробці програмного продукту використовувався вбудований в UNITY 3D компілятор проекту до Visual Studio. Також оскільки MVC в першу чергу був розроблений для WEB-додатків, для ігорвого проекту більш коректним буде найменування Model-Display-Manager, MDM

Завдяки цьому проект має наступну архітектуру(рис. 2.1):

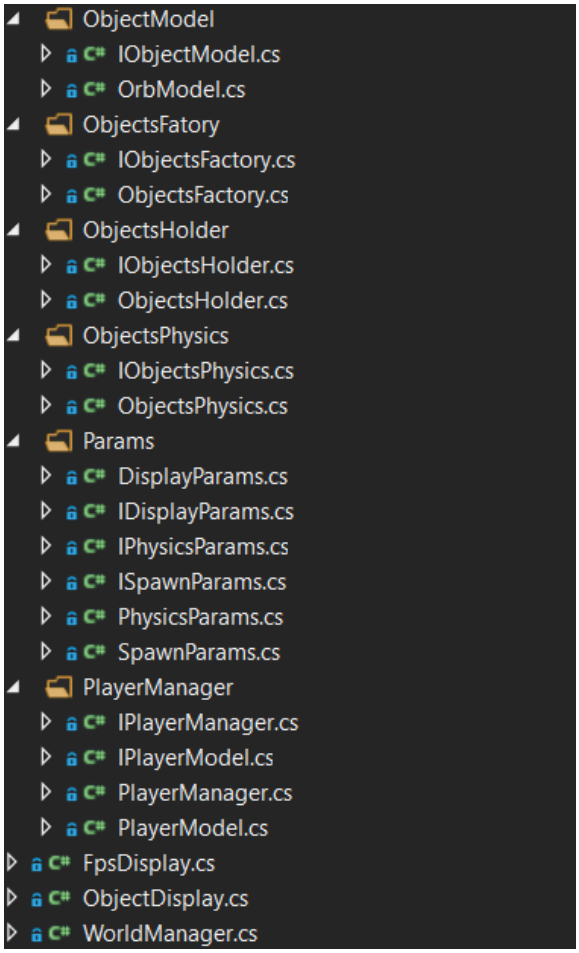

Рис. 2.1 Структура проекту

Даний шаблон найбільш підходить для розробки ігрових додатків. Через те що ідеальної оптимізації неможливо досягти у ігрових додатках дуже важливо щоб кожна анімація виконувалась у той момент в який вона потрібна та не було перебоїв цієї анімації.

Таким чином шаблон MVC найбільш відповідний для ігрових додатків.

2.4. Опис структури програми та алгоритмів її функціонування

2.4.1. Опис створення основних етапів програми

Розробка аркадних міні-ігор є складним процесом, що допомагає вдосконалити навички роботи в розробці додатків. Для створення гри необхідно продумати компоненти гри та її логіку, а саме процес взаємодії гри з користувачем, умови виграшу, фіксації результатів і т. п.

Результатом виконання дипломного проекту є повноцінний працеспроможний додаток, що реалізує двовимірну гру, в якій використовується сучасний графічний дизайн, невибагливі правила гри, який призначений для розваги користувачів.

Написання програми було розпочато з проектування архітектури системи. Щоб уникнути проблем при створені архітектури, були використані сучасні принципи проектування програмного забезпечення.

Першим і найголовнішим за таких принципів є SOLID (абревіатура з перших літер 5 вкладених принципів). SOLID[13] — це набір принципів об’єктно-орієнтованого програмування, які представив Роберт Мартін[14] (дядько Боб) у 1995 році. Їхня ідея в тому, що треба уникати залежностей між компонентами коду. Якщо є велика кількість залежностей, такий код важко підтримувати (спагетті-код). Його основні проблеми:

- жорсткість (Rigidity): кожна зміна викликає багато інших змін;
- крихкість (Fragility): зміни в одній частині ламають роботу інших частин;
- нерухомість (Immobility): код не можна повторно використати за межами його контексту.

Надалі буде наведено детальніший опис цих принципів.

Принцип єдиного обов’язку (Single Responsibility Principle).

Кожен клас повинен виконувати лише один обов’язок. Це не означає, що в нього має бути тільки один метод. Це означає, що всі методи класу мають бути сфокусовані на виконання одного спільного завдання. Якщо є методи, які не

відповідають меті існування класу, їх треба винести за його межі[15].

Наприклад, клас User. Його обов’язок надавати інформацію про користувача: ім’я, email і тип підписки, яку він використовує в сервісі. Приклад коду такого класу.

```
enum SubscriptionTypes {
  BASIC = 'BASIC',
  PREMIUM = 'PREMIUM'
}
 class User {
  constructor (
    public readonly firstName: string,
    public readonly lastName: string,
    public readonly email: string,
    public readonly subscriptionType: SubscriptionTypes,
    public readonly subscriptionExpirationDate: Date
  ) {}
   public get name(): string {
    return `${this.firstName} ${this.lastName}`;
   }
   public hasUnlimitedContentAccess() {
    const now = new Date();
     return this.subscriptionType === SubscriptionTypes.PREMIUM
      && this.subscriptionExpirationDate > now;
   }
}
```

Розглянемо метод hasUnlimitedContentAccess. На основі типу підписки він визначає, чи є у користувача необмежений доступ до контенту. Але ж стоп, хіба це відповідальність класу User робити такий висновок? Виходить, у класу User є дві мети для існування: надавати інформацію про користувача і робити висновок, який у нього доступ до контенту на основі підписки. Це порушує принцип

Single Responsibility.

Чому існування методу hasUnlimitedContentAccess у класі User має негативні наслідки? Бо контроль над типом підписки розпливається по всій програмі[16]. Крім класу User, можуть бути класи MediaLibrary та Player, які теж вирішуватимуть, що їм робити на основі цих даних. Кожен клас трактує по-своєму, що означає тип підписки. Якщо правила наявних підписок змінюються, треба оновлювати всі класи, оскільки кожен побудував свій набір правил роботи з ними.

Видалимо метод hasUnlimitedContentAccess у класі User і створимо новий клас, який буде відповідати за роботу з підписками. Приклад:

```
class AccessManager {
  public static hasUnlimitedContentAccess(user: User) {
    const now = new Date();
     return user.subscriptionType === SubscriptionTypes.PREMIUM
      && user.subscriptionExpirationDate > now;
  }
  public static getBasicContent(movies: Movie[]) {
   return movies.filter(movie => movie.subscriptionType ===
SubscriptionTypes.BASIC);
  }
  public static getPremiumContent(movies: Movie[]) {
   return movies.filter(movie => movie.subscriptionType ===
SubscriptionTypes.PREMIUM);
  }
  public static getContentForUserWithBasicAccess(movies: Movie[]) {
   return AccessManager.getBasicContent(movies);
  }
  public static getContentForUserWithUnlimitedAccess(movies: Movie[]) {
   return movies;
  }}
```

Ми інкапсулювали всі правила роботи з підписками в одному класі. Якщо будуть зміни у правилах, вони залишаться тільки у цьому класі та не зачіплять інші.

Single Responsibility Principle стосується не тільки рівня класів — модулі класів теж потрібно проєктувати таким чином, щоб вони були вузькоспеціалізовані.

Крім SOLID існує ще інший набір принципів проектування програмного забезпечення — GRASP. Деякі його принципи перетинаються з SOLID. Якщо говорити про Single Responsibility Principle, то з GRASP можно співставити:

- інформаційний експерт (Information Expert) — об’єкт, який володіє повною інформацією з предметної області;
- низька зв’язаність (Low Coupling) і високе зчеплення (High Cohesion) — компоненти різних класів або модулів повинні мати слабкі зв’язки між собою, але компоненти одного класу або модуля мають бути логічно пов’язані або тісно взаємодіяти один з одним.

Принцип відкритості/закритості (Open/Close Principle).

Класи мають бути відкриті до розширення, але закриті для змін. Якщо є клас, функціонал якого передбачає чимало розгалужень або багато послідовних кроків, і є великий потенціал, що їх кількість буде збільшуватись, то потрібно спроектувати клас таким чином, щоб нові розгалуження або кроки не призводили до його модифікації[17].

Напевно, кожен з нас бачив нескінченні ланцюжки if then else або switch. Щойно додається чергова умова, ми пишемо черговий if then else, змінюючи при цьому сам клас. Або клас виконує процес з багатьма послідовними кроками — і кожен новий крок призводить до його зміни. А це порушує Open/Close Principle.

Як можна розширювати клас і водночас не змінювати його? Розглянемо кілька способів. Приклад неправильного проектування системи розрахунку площі фігури:

```
class Rect {
  constructor(
    public readonly width: number,
    public readonly height: number
  ) { }
}
 class Square {
  constructor(
    public readonly width: number
  ) { }
}
 class Circle {
  constructor(
    public readonly r: number
  ) { }
}
 class ShapeManager {
  public static getMinArea(shapes: (Rect | Square | Circle)[]): number {
    const areas = shapes.map(shape => {
      if (shape instanceof Rect) {
        return shape.width * shape.height;
      }
       if (shape instanceof Square) {
        return Math.pow(shape.width, 2);
      }
       if (shape instanceof Circle) {
        return Math.PI * Math.pow(shape.r, 2);
      }
       throw new Error('Is not implemented');
    });
```

```
    return Math.min(...areas);
  }
}
```

Як бачимо, додавання нових фігур буде призводити до модифікації класу ShapeManager. Оскільки площа фігури тісно пов’язана із самою фігурою, можна змусити фігури самостійно рахувати свою площу, привести їх до одного інтерфейсу, а тоді передавати їх у метод getMinArea. Приклад правильної системи розрахунку площі фігури:

```
interface IShape {
  getArea(): number;
}
 class Rect implements IShape {
  constructor(
    public readonly width: number,
    public readonly height: number
  ) { }
   getArea(): number {
    return this.width * this.height;
  }
}
 class Square implements IShape {
  constructor(
    public readonly width: number
  ) { }
   getArea(): number {
    return Math.pow(this.width, 2);
  }
}
```

```
class Circle implements IShape {
  constructor(
    public readonly r: number
  ) { }
   getArea(): number {
    return Math.PI * Math.pow(this.r, 2);
  }
}
 class ShapeManager {
  public static getMinArea(shapes: IShape[]): number {
    const areas = shapes.map(shape => shape.getArea());
    return Math.min(...areas);
  }
}
```

Тепер, якщо у нас з’являться нові фігури, все, що потрібно зробити, — це імплементувати інтерфейс IShape. І клас ShapeManager відразу буде її підтримувати без жодних модифікацій.

А що робити, якщо не можемо додавати методи до фігур? Існують методи, які суперечать Single Responsibility Principle. Тоді можна скористатися шаблоном проектування «Стратегія» (Strategy): створити множину схожих алгоритмів і викликати їх за певним ключем. Приклад реалізації системи розрахунку площі фігур за допомогою патерна стратегія:

```
interface IShapeAreaStrategiesMap {
  [shapeClassName: string]: (shape: IShape) => number;
}
 class ShapeManager {
  constructor(
    private readonly strategies: IShapeAreaStrategiesMap
```

```
  ) {}

  public getMinArea(shapes: IShape[]): number {
    const areas = shapes.map(shape => {
      const className = shape.constructor.name;
     const strategy = this.strategies[className];
      if (strategy) {
       return strategy(shape);
      }
      throw new Error(`Could not find Strategy for '${className}'`);
        });
    return Math.min(...areas);
  }
}
 // Strategy Design Pattern
const strategies: IShapeAreaStrategiesMap = {
  [Rect.name]: (shape: Rect) => shape.width * shape.height,
  [Square.name]: (shape: Square) => Math.pow(shape.width, 2),
  [Circle.name]: (shape: Circle) => Math.PI * Math.pow(shape.r, 2)
};
 const shapes = [
  new Rect(1, 2),
  new Square(1),
  new Circle(1),
];
 const shapeManager = new ShapeManager(strategies);
console.log(shapeManager.getMinArea(shapes));
```

Перевага Strategy в тому, що є змога змінювати в рантаймі набір стратегій і спосіб їх вибору. Можна прочитати файл конфігурацій (.json, .xml, .yml) і на

його основі збудувати стратегії. Тоді, якщо відбувається зміна стратегій, не потрібно розробляти нову версію програми і деплоїти її на сервери, достатньо підмінити файл з конфігураціями і сказати програмі, щоб та його знову прочитала. Крім того, стратегії можна реєструвати в Inversion of Control контейнері. У такому разі клас, який їх потребує, отримає стратегії автоматично на етапі створення.

Розглянемо тепер ситуацію, коли триває послідовна багатокрокова обробка даних. Якщо кількість кроків зміниться, нам доведеться змінювати клас. Приклад неправильної реалізації послідовної обробки даних:

```
class ImageProcessor {
...
  public processImage(bitmap: ImageBitmap): ImageBitmap {
    this.fixColorBalance(bitmap);
    this.increaseContrast(bitmap);
    this.fixSkew(bitmap);
    this.highlightLetters(bitmap);

    return bitmap;
  }
}
```

Застосуємо дизайн-патерн «Конвеєр» (Pipeline):

```
type PipeMethod = (bitmap: ImageBitmap) => void;

// Pipeline Design Pattern
class Pipeline {
  constructor(
    private readonly bitmap: ImageBitmap
  ) { }

  public pipe(method: PipeMethod) {
```

```
    method(this.bitmap);
   }

   public getResult() {
    return this.bitmap;
   }
  }

  class ImageProcessor {
   public static processImage(bitmap: ImageBitmap, pipeMethods:
PipeMethod[]): ImageBitmap {
    const pipeline = new Pipeline(bitmap);
    pipeMethods.forEach(method => pipeline.pipe(method))

    return pipeline.getResult();
   }
  }

  const pipeMethods = [
   fixColorBalance,
   increaseContrast,
   fixSkew,
   highlightLetters
  ];

  const result = ImageProcessor.processImage(scannedImage, pipeMethods);
```

Тепер, якщо потрібно змінити спосіб обробки зображення, ми модифікуємо масив з методами. Сам клас ImageProcessor залишається незмінним. Тепер уявіть, що треба обробляти різні зображення по-різному. Замість того, щоб писати різні

версії ImageProcessor, по-іншому комбінуємо в масиві pipeMethods потрібні нам методи.

Ще кілька переваг. Раніше ми додавали новий метод обробки зображення прямо в ImageProcessor, і в нас виникала потреба додавати нові залежності. Наприклад, метод highlightLetters вимагає додаткову бібліотеку для пошуку символів на зображенні. Відповідно, більше методів — більше залежностей. Зараз кожен PipeMethod можна розробити в окремому модулі й підключати тільки необхідні залежності. Після такої декомпозиції все дуже легко тестувати. Ну й на останок: така структура коду мотивує розробника писати якомога коротші методи обробки з чіткими інтерфейсами. До цього можна було зробити один великий метод fixQuality, де б відбувалося і виправлення балансу кольорів, і вирівнювання зображення, і збільшення контрасту. Але в такому великому методі було б складно контролювати параметри кожного накладеного на зображення фільтру. Ймовірно, виникла б ситуація, коли fixQuality працював би добре для одного зразка зображення, але для іншого на етапі тестування він би не працював зовсім. Маючи кілька добре відладжених методів, значно простіше коригувати параметри, щоб отримати потрібний результат.

Принципами GRASP, що спільні з Open/Close Principle :

- стійкість до змін (Protected Variations): потрібно захищати компоненти від впливу змін інших компонентів. Тому для компонентів, які потенційно часто будуть зазнавати змін, ми створюємо один інтерфейс і кілька його імплементацій, використовуючи поліморфізм;
- поліморфізм (Polymorphism) — можливість мати різні варіанти поведінки на основі типу класу. Типи класу з варіативною поведінкою мають підпадати під один інтерфейс;
- перенаправлення (Indirection): слабка зв’язаність між компонентами та можливість їх перевикористання досягається завдяки створенню посередника (mediator), який бере на себе взаємодію між компонентами.

- чиста вигадка (Pure Fabrication): можна створити штучний об’єкт, якого немає в домені, але який наділений властивостями, що дають змогу зменшити залежність між об’єктами. Наприклад, в домені є товар і склад. Якщо зробимо так, що склад буде контролювати наявність товарів, буде складно створити функціонал, який, наприклад, перевірятиме наявність товару в партнерів і пропонуватиме його користувачу. Тому ми додаємо об’єкт ProductManager, який перевірятиме, чи є товар на складі. Якщо немає, перевірятиме його в партнерів. Оскільки за допомогою ProductManager ми відв’язали товар від складу, можемо повністю позбутися його та продавати товари від партнерів, якщо виникне така потреба.

Принцип підстановки Лісков (Liskov Substitution Principle).

Якщо об’єкт базового класу замінити об’єктом його похідного класу, то програма має продовжувати працювати коректно[18].

Якщо ми перевизначаємо похідні методи від батьківського класу, то нова поведінка не має суперечити поведінці базового класу. Як приклад порушення цього принципу розглянемо код:

```
class BaseClass {
  public add(a: number, b: number): number {
    return a + b;
  }
}

class DerivedClass extends BaseClass {
  public add(a: number, b: number): number {
    throw new Error('This operation is not supported');
  }
}
```

Можливий також випадок, що батьківський метод буде суперечити логіці класів-нащадків. Приклад з ієрархію транспортних засобів:

```
class Vehicle {
  accelerate() {
    // implementation
  }
   slowDown() {
    // implementation
  }
   turn(angle: number) {
    // implementation
  }
}
 class Car extends Vehicle {
}
 class Bus extends Vehicle {
}
```

Все працює до того моменту, до поки ми не додаємо новий клас Поїзд:

```
class Train extends Vehicle {
  turn(angle: number) {
    // is that possible?
  }
}
```

Оскільки поїзд не може змінювати довільно напрямок свого руху, то turn батьківського класу буде порушувати принцип підстановки Лісков.

Щоб виправити цю ситуацію, ми можемо додати два батьківські класи: FreeDirectionalVihicle — який буде дозволяти довільний напрямок руху і BidirectionalVehicle — рух тільки вперед і назад. Тепер всі класи будуть наслідувати тільки ті методи, які можуть забезпечити:

```
class FreeDirectionalVehicle extends Vehicle {
  turn(angle: number) {
    // implementation
  }
}
 class BidirectionalVehicle extends Vehicle {
}
```

Крім того, клас-нащадок не має додавати ніяких умов до виконання методу і після виконання методу. Наприклад:

```
class Logger {
  log(text: string) {
    console.log(text);
  }
}

class FileLogger extends Logger {
  constructor(private readonly path: string) {
    super();
  }

  log(text: string) {
    // append text file
  }
}

class TcpLogger extends Logger {
  constructor(private readonly ip: string, private readonly port: number) {
    super();
  }
```

```
  log(text: string) {
    // implementation
  }

  openConnection() {
    // implementation
  }

  closeConnection() {
    // implementation
  }
}
```

У цій ієрархії ми не зможемо легко замінити об’єкти батьківського класу FileLogger об’єктами TcpLogger, оскільки до та після виклику методу нам потрібно додатково викликати openConnection та closeConnection. Виходить, ми накладаємо 2 додаткові умови на виклик методу log, що також порушує принцип підстановки Лісков.

Щоб вирішити ситуацію вище, ми можемо зробити методи openConnection та closeConnection приватними. В методі log класу TcpLogger організуємо запис логів в файл. Періодично (наприклад, кожної хвилини) відкриватимемо з’єднання, відправлятимемо файл з логами і закриватемо з’єднання. Додатково потрібно переконатися, що перед тим, як програма буде закрита, ми відправили всі логи. Якщо програма була аварійно завершена, ми можемо відправити логи під час наступного її запуску.

Принцип розділення інтерфейсу (Interface Segregation Principle)

Краще, коли є багато спеціалізованих інтерфейсів, ніж один загальний. Маючи один загальний інтерфейс, є ризик потрапити в ситуацію, коли похідний клас логічно не зможе успадкувати якийсь метод[19]. Розглянемо приклад:

```
interface IDataSource {
  connect(): Promise<boolean>;
  read(): Promise<string>;
}
 class DbSource implements IDataSource {
  connect(): Promise<boolean> {
    // implementation
  }
   read(): Promise<string> {
    // implementation
  }
}
 class FileSource implements IDataSource {
  connect(): Promise<boolean> {
    // implementation
  }
   read(): Promise<string> {
    // implementation
  }
}
```

Оскільки з файлу ми читаємо локально, то метод Connect зайвий. Розділимо загальний інтерфейс IDataSource:

```
interface IDataSource {
  read(): Promise<string>;
}
 interface IRemoteDataSource extends IDataSource {
  connect(): Promise<boolean>;
}
```

```
class DbSource implements IRemoteDataSource {
}
 class FileSource implements IDataSource {
}
```

Тепер кожна імплементація використовує тільки той інтерфейс, який може забезпечити.

Принцип інверсії залежностей (Dependency Inversion Principle).

Він складається з двох тверджень:

- високорівневі модулі не повинні залежати від низькорівневих. І ті, і ті мають залежати від абстракцій[20];
- абстракції не мають залежати від деталей реалізації. Деталі реалізації повинні залежати від абстракцій.

Розберемо код, який порушує ці твердження:

```
class UserService {
  async getUser(): Promise<User> {
    const now = new Date();
     const item = localStorage.getItem('user');
    const cachedUserData = item && JSON.parse(item);
    if (cachedUserData && new Date(cachedUserData.expirationDate) > now) {
      return cachedUserData.user;
    }
     const response = await fetch('/user');
    const user = await response.json();
     const expirationDate = new Date();
    expirationDate.setHours(expirationDate.getHours() + 1);
     localStorage.setItem('user', JSON.stringify({
      user,
      expirationDate
    }));
```

```
    return user;
  }
}
```

Наш модуль верхнього рівня UserService використовує деталі реалізації трьох модулів нижнього рівня: localStorage, fetch та Date. Такий підхід поганий тим, що якщо ми, наприклад, вирішимо замість fetch користуватися бібліотекою, яка робить HTTP-запити, то доведеться переписувати UserService. Крім того, такий код важко покрити тестами.

Ще одним порушенням є те, що з методу getUser ми повертаємо реалізований клас User, а не його абстракцію — інтерфейс IUser.

Створимо абстракції, з якими було б зручно працювати всередині модуля UserService:

```
interface ICache {
  get<T>(key: string): T | null;
  set<T>(key: string, user: T): void;
}

interface IRemoteService {
  get<T>(url: string): Promise<T>;
}

class UserService {
  constructor(
    private readonly cache: ICache,
    private readonly remoteService: IRemoteService
  ) {}

  async getUser(): Promise<IUser> {
```

```
    const cachedUser = this.cache.get<IUser>('user');

    if (cachedUser) {
      return cachedUser;
    }

    const user = await this.remoteService.get<IUser>('/user');
    this.cache.set('user', user);

    return user;
  }
}
```

Як бачимо, код вийшов значно простішим і його можна легко протестувати. Тепер поглянемо на реалізацію інтерфейсів ICache та IRemoteService:

```
interface IStorage {
  getItem(key: string): any;
  setItem(key: string, value: string): void;
}

class LocalStorageCache implements ICache {
  private readonly storage: IStorage;

  constructor(
getStorage = (): IStorage => localStorage,
private readonly createDate = (dateStr?: string) => new Date(dateStr)
 ) {
this.storage = getStorage()
 }
```

```
    get<T>(key: string): T | null {
    const item = this.storage.getItem(key);
    const cachedData = item && JSON.parse(item);

    if (cachedData) {
      const now = this.createDate();

      if (this.createDate(cachedData.expirationDate) > now) {
        return cachedData.value;
      }
    }

    return null;
     }

  set<T>(key: string, value: T): void {
    const expirationDate = this.createDate();
    expirationDate.setHours(expirationDate.getHours() + 1);

    this.storage.setItem(key, JSON.stringify({
      value,
      expirationDate
    }));
  }
}

    class RemoteService implements IRemoteService {
      private readonly fetch: ((input: RequestInfo, init?: RequestInit) =>
Promise<Response>)
```

```
constructor(
  getFetch = () => fetch
   ) {
  this.fetch = getFetch()
  }

async get<T>(url: string): Promise<T> {
  const response = await this.fetch(url);
  const obj = await response.json();

  return obj;
   }
  }
```

Ми зробили врапери над localStorage та fetch. Важливим моментом у реалізації двох класів є те, що ми не використовуємо localStorage та fetch прямо. Ми весь час працюємо зі створеними для них інтерфейсами. LocalStorage та fetch будуть передаватися в конструктор, якщо там не буде вказано жодних параметрів. Для тестів можна створити mocks або stubs, які замінять localStorage або fetch, і передати їх як параметри в конструктор.

Схожий прийом використовують і для дати: якщо нічого не передати, то кожного разу LocalStorageCache буде отримувати нову дату. Якщо ж для тестів потрібно зафіксувати певну дату, треба передати її в параметрі конструктора.

2.4.2. Оптимізація проекту

Багато ігор використовують різні способи та хитрощі для того, щоб ігри задовольнили користувачів як із потужними комп’ютерами, так і зі слабкими[21]. У нашому продукті ми використували два способи, котрі полегшать навантаження на систему.

I. Використання власної системи колізій

Вбудована в UNITY 3D система колізій (взаємодії між фізичними об'єктами) є дуже універсальною, тож робить багато зайвих розрахунків фізики, які у “Orbs World” не використовуються. Тож було прийнято рішення написати власну систему взаємодії. Зараз вона підтримує виключно кола (орби), але також здатна до розширення завдяки використаним принципам SOLID та патерну стратегія.

II. Кешування вже створених об'єктів

Оскільки гра доволі динамічна, і середній час одного раунду не перевищує 2х хвилин, кожний раз видаляти та заново створювати орби доволі не ефективно, особливо враховуючи той факт що на початок раунду створюється 3000 об'єктів. Тож була реалізована система кешування (збереження) орбів. Кожного разу, коли менший орб поглинаються іншим, більшим орбом, вона не видаляє його об'єкт, а лише сховує його. І при наступних ігрових раундах, вона знов відновлює його, щоб не створювати з нуля.

2.4.3. Опис схеми роботи програми

Робота програми забезпечується виконанням наступних взаємодіючих компонентів, що відображені на рисунку 2.2.

Рис. 2.2. Умовне зображення роботи програми

2.5. Обґрунтування та організація вхідних та вихідних даних програми

Аналізуючи поставлені перед додатком завдання, доцільно виділити наступні групи потоків інформації, що забезпечують правильну роботу програми:

- вхідний потік - дані, що поступають у додаток за допомогою введення їх користувачем;
- вихідний потік - дані, що генеруються в процесі роботи додатку на підставі обробки даних користувачів і передбачених відповідей на певні його дії;

Вхідними даними для роботи додатку є інформація, яка вводиться та відноситься до групи вхідного потоку:

- вибір варіанту генерації інших орбів (можна змінити натиском

кнопки «перезавантажити»);

- координати розміщення головного гравця на карті ігрового простору;
- команди для керування ходом ігрового процесу, введені за допомогою пристроїв вводу – сенсорного екрану смартфона (обрана позиція для переміщення героя.

В результаті обробки цих даних, здійснюється виведення певної інформації або реакція додатку, яка відноситься до групи вихідного потоку, що забезпечує безпосередньо результат роботи на певному етапі функціонування додатку.

Вихідними даними є:

- зображення головного персонажу гри та його дій;
- зображення ворожих персонажів;
- зміна кольору ворогів по мірі росту орба гравця;
- «смерть» ГГ після поглинання його більшим орбом;
- вибір перезавантаження гри під час гри або після смерті.

2.6. Опис роботи розробленої системи

2.6.1. Використані технічні засоби

Для роботи даного програмного додатку необхідний смартфон під управлінням операційної системи Android з наступними характеристиками:

- операційна система Android 5.0 Lollipop, або пізніша версія;
- 512 Мб оперативної пам'яті;
- 4” сенсорний екран (можна і з більшою діагоналлю);
- 100 Мб вільного місця на смартфоні.

Швидкість роботи програмного забезпечення буде залежати від конфігурації смартфону.

Надані вище технічні характеристики є рекомендованими. При цих технічних засобах розробка ПЗ буде здійснюватись згідно вимог до надійності, швидкості обробки даних та безпеки.

2.6.2. Використані програмні засоби

В програмі використана мова програмування C#. Для створення спрайтів (об’єктів двовимірної графіки) використано інструмент розробки paint.net та інструмент для розробки двовимірних та тривимірних ігор Unity 3D.

2.6.3. Виклик та завантаження програми

Для запуску програми на смартфоні з android, необхідно завантажити файл orbsWorld.apk, та викликати його. У разі якщо смартфон буде просити надати право встановлювати додатки з невідомих ресурсів, треба надати право конкретній програмі за допомогою якого гра була завантажена. Якщо смартфон буде попереджати про потенційно небезпечний код - це сповіщення треба проігнорувати, і підтвердити встановлення додатка.

2.6.4. Опис інтерфейсу користувача

2.6.4.1. Правила та опис гри

Головна мета гри - зробити свій орб найбільшим серед усіх орбів на мапі, шляхом поглинання менших орбів.

Для поліпшення ідентифікування, які орби є більшими від гравця, а які меншими, окрім власно розмірів використовуються кольорова індикація:

- червоні орби - більші за гравця;
- сині орби - менші за гравця;
- сірий орб - ГГ.

Для управління ГГ необхідно натискати на екран смартфону у протилежному напрямку відносно ГГ, від очікуваного напряму руху. Наприклад, щоб орб гравця почав рух угору, треба натиснути знизу від ГГ. Окрім точки в якій гравець надає прискорення своєму орбу, потрібно враховувати і довжину натиску. Чим довше натискати - тим більше прискорення отримає орб. Також

важливо враховувати, що під час управління, гравець не просто рухає свій орб, він надає йому прискорення, тож після закінчення натиску, орб не зупинится одразу, а буде рухатись по інерції в своєму напрямку, поступово сповільнюючись. Щоб загальмувати, ефективніше всього буде надати прискорення протилежно від напрямку руху. Оскільки край екрану - є границею для орбів, якщо врізатись у нього, то орб відскочить. Це буде схоже на відскакування м'яча від поверхні, якщо його кинути під певним кутом до неї.

2.6.4.2. Зображення етапів гри

Після завантаження, відразу буде згенеровано поле гри, на якому у самому центрі буде розташовано орб гравця, а також буде присутня кнопка Restart (перезавантаження), яка виставляє гравця на центр поля, та перегенерує оточуючі орби. Приклад щойно завантаженої гри на рисунку 2.3.

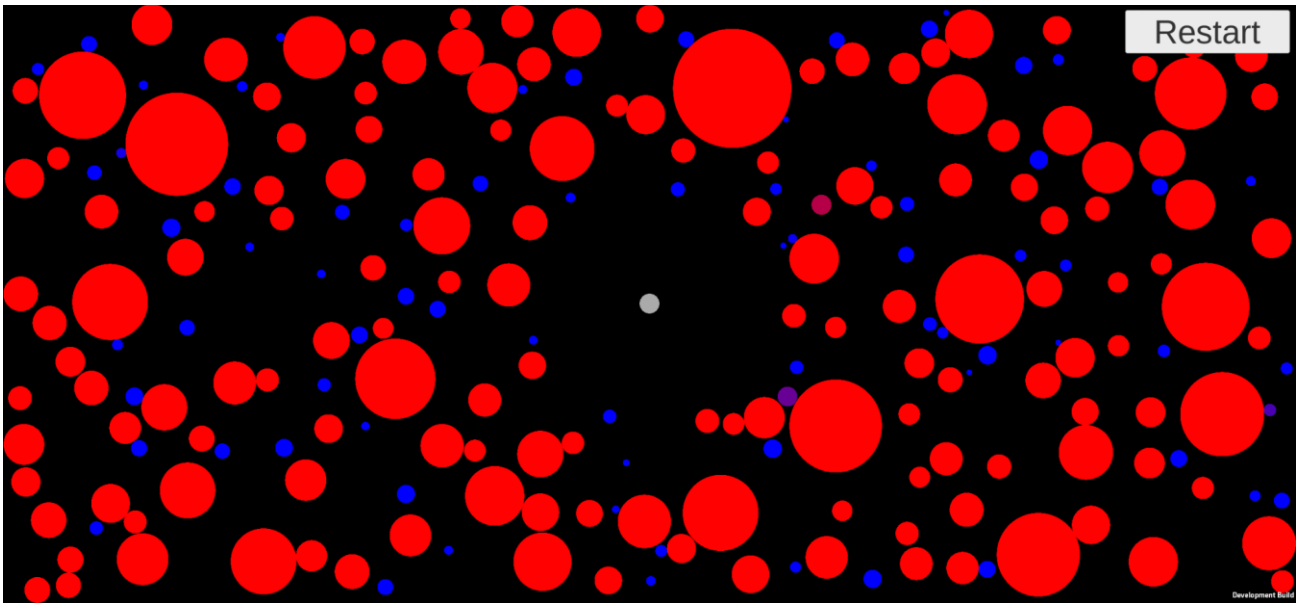

Рис. 2.3 Щойно завантаженої гра.

Відразу після завантаження, якщо перегенерувати мапу, та торихи пограти, можна побачити велику різницю (рисунок 2.4) з попередньою генерацією (рисунок 2.3).

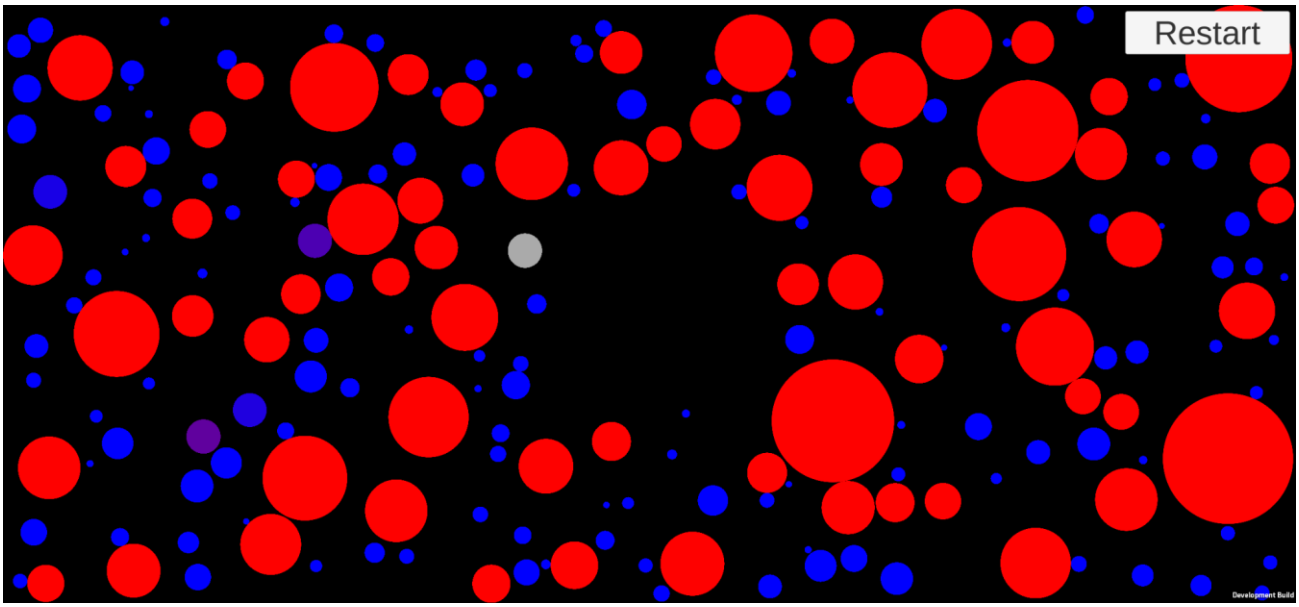

Рис. 2.4 Перегенерована мапа.

Поглинувши декілька орбів, і збільшивши свій розмір, набагато більше ворогів стануть слабшими за граця, і їх вже можна поглинути теж. Це відно на рисунку 2.5.

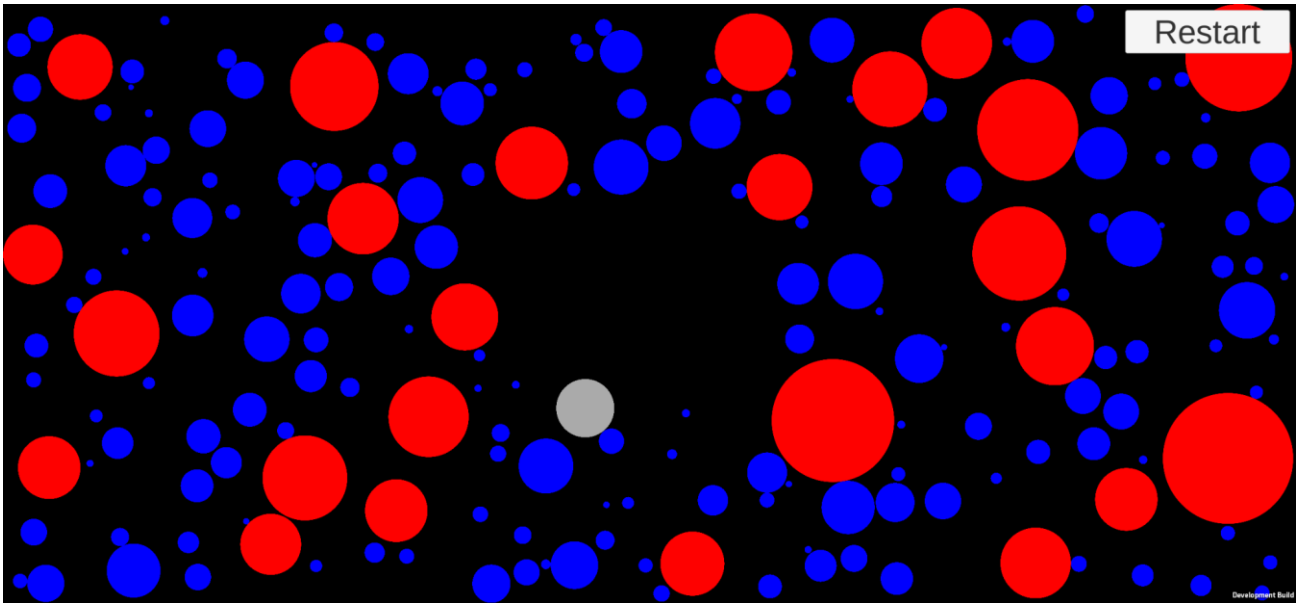

Рис. 2.5 Більшість ворожих орбів вже менші за орб гравця, тож вони змінили колір.

На рисунку 2.6 зображено стан гри, після поглинання ще певної кількості ворогів, орб гравця вже майже найбільший.

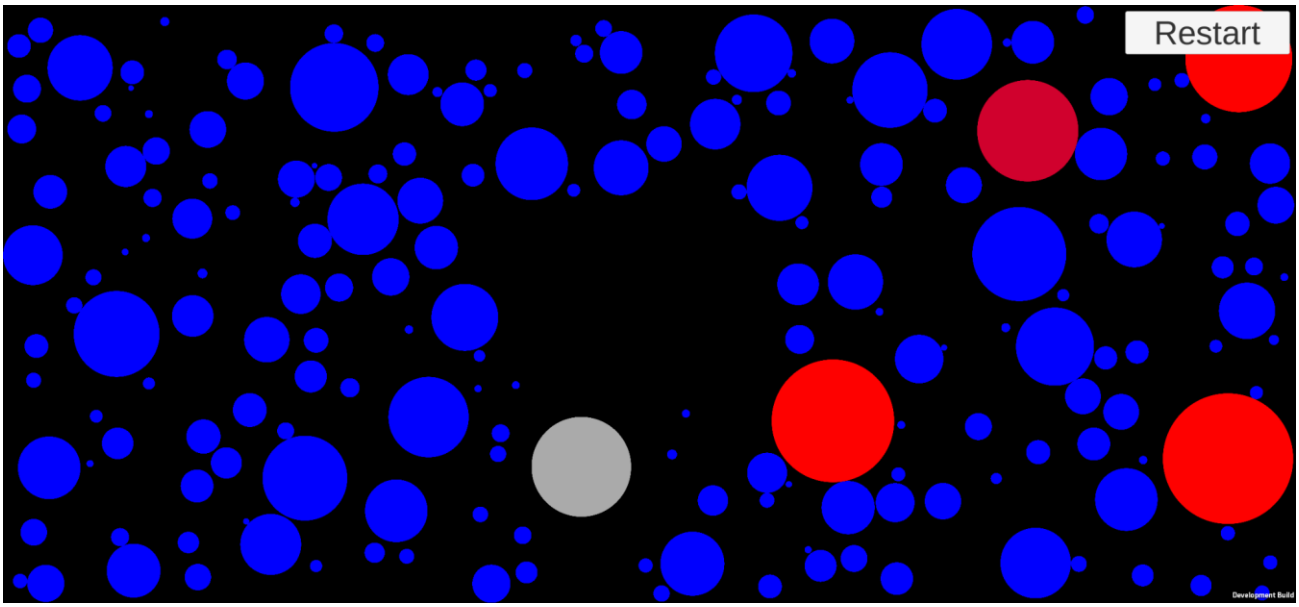

Рис. 2.6 Гравець дуже близький до виграшу.

І от, нарешті гравець став найбільшим серед усієї мапи. Екран перемоги на рисунку 2.7.

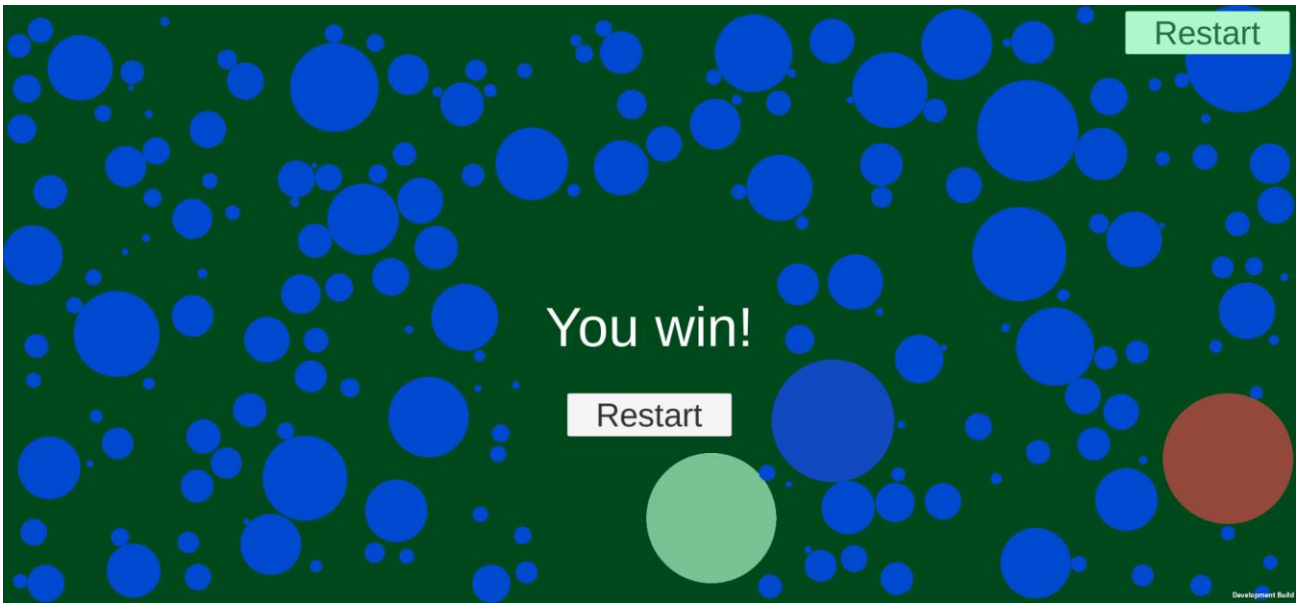

Рис. 2.7 Екран перемоги.

Також, у разі поглинання орба гравця ворогами, буде виведено екран програшу, як на рисунку 2.8.

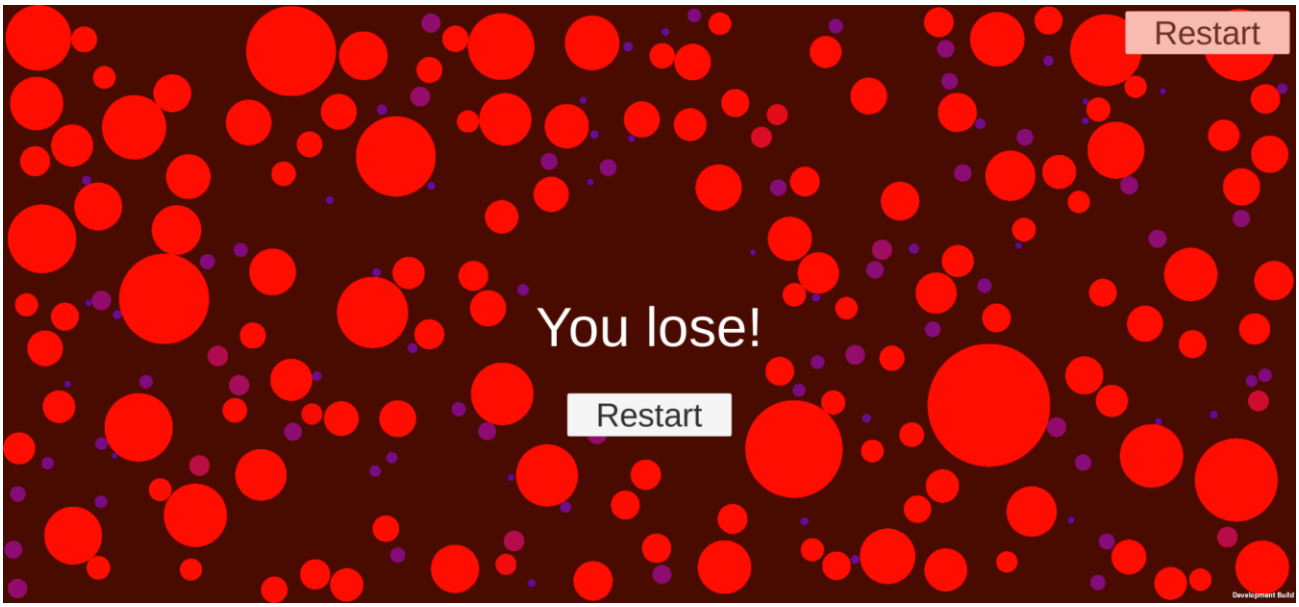

Рис. 2.8 Екран програшу.

РОЗДІЛ 3
ЕКОНОМІЧНИЙ РОЗДІЛ

3.1. Розрахунок трудомісткості та вартості розробки програмного продукту

Початкові дані:

1. передбачуване число операторів програми – 1351;
2. коефіцієнт складності програми – 1,25;
3. коефіцієнт корекції програми в ході її розробки – 0,07;
4. годинна заробітна плата програміста – 125 грн/год;
5. коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1,1;
6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 1,4;
7. вартість машино-години ЕОМ – 20 грн/год.

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки ПЗ може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

$$t = t_o + t_u + t_a + t_n + t_{отл} + t_д, \text{ людино-годин}, \qquad (3.1)$$

де t_o – витрати праці на підготовку й опис поставленої задачі (приймається 50);

t_u – витрати праці на дослідження алгоритму рішення задачі;

t_a – витрати праці на розробку блок-схеми алгоритму;

t_n – витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ – витрати праці на налагодження програми на ЕОМ;

$t_д$ – витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у ПЗ, яке розробляється.

Умовне число операторів (підпрограм):

$$Q = q \cdot C \cdot (1+q), \quad (3.2)$$

де q – передбачуване число операторів;

C – коефіцієнт складності програми;

p – коефіцієнт кореляції програми в ході її розробки.

$$Q = 1351 \cdot 1{,}25 \cdot (1 + 0{,}07) = 1806{,}96;$$

Витрати праці на вивчення опису задачі t_u визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_u = \frac{Q \cdot B}{(75...85) \cdot K}, \text{ людино-годин}, \quad (3.3)$$

де B – коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

K – коефіцієнт кваліфікації програміста, обумовлений стажем роботи з даної спеціальності;

$$t_u = \frac{1806{,}96 \cdot 1{,}1}{85 \cdot 1{,}4} = 16{,}7, \text{ людино-годин.}$$

Витрати праці на розробку алгоритму рішення задачі:

$$t_a = \frac{Q}{(20...25) \cdot K}; \quad (3.4)$$

$$t_a = \frac{1806{,}96}{20 \cdot 1{,}4} = 64{,}53, \text{ людино-годин.}$$

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20...25) \cdot K}; \quad (3.4)$$

$$t_n = \frac{1806,96}{25 \cdot 1,4} = 51,63, \text{ людино-годин.}$$

Витрати праці на налагодження програми на ЕОМ:

- за умови автономного налагодження одного завдання:

$$t_{отл} = \frac{Q}{(4...5) \cdot K}; \quad (3.6)$$

$$t_{отл} = \frac{1806,96}{5 \cdot 1,4} = 258,14, \text{ людино-годин.}$$

- за умови комплексного налагодження завдання:

$$t^{к}_{отл} = 1,2 \cdot t_{отл}; \quad (3.7)$$

$$t^{к}_{отл} = 1,2 \cdot 258,14 = 309,77, \text{ людино-годин,}$$

Витрати праці на підготовку документації:

$$t_д = t_{др} + t_{до}; \quad (3.8)$$

де $t_{др}$ – трудомісткість підготовки матеріалів і рукопису

$$t_{др} = \frac{Q}{(15...20) \cdot K}; \quad (3.9)$$

$$t_{др} = \frac{1806,96}{15 \cdot 1,4} = 86,05, \text{ людино-годин.}$$

$t_{до}$– трудомісткість редагування, печатки й оформлення документації

$$t_{до} = 0,75 \cdot t_{др}; \quad (3.10)$$

$$t_{до} = 0,75 \cdot 86,05 = 64,53, \text{ людино-годин.}$$

$$t_{д} = 86,05 + 64,53 = 150,58, \text{ людино-годин.}$$

Отримаємо трудомісткість розробки програмного забезпечення:

$$t = 50 + 16,7 + 64,53 + 51,63 + 309,77 + 150,58 = 642,21, \text{ людино-годин.}$$

У результаті ми розрахували, що в загальній складності необхідно 642,21 людино-годин для розробки даного програмного забезпечення.

3.2. Розрахунок витрат на створення програми

Витрати на створення ПЗ $К_{ПО}$ включають витрати на заробітну плату виконавця програми Зз/п і витрат машинного часу, необхідного на налагодження програми на ЕОМ.

$$К_{ПО} = З_{ЗП} + З_{МВ}, \text{ грн,} \quad (3.11)$$

де $З_{ЗП}$ – заробітна плата виконавців, яка визначається за формулою:

$$З_{ЗП} = t \cdot С_{ПР}, \text{ грн,} \quad (3.12)$$

де t – загальна трудомісткість, людино-годин;

$C_{ПР}$ – середня годинна заробітна плата програміста, грн/година

$$З_{ЗП} = 642{,}21 \cdot 125 = 80276{,}25, \text{ грн.}$$

$З_{МВ}$ – Вартість машинного часу, для налагодження програми на ЕОМ:

$$З_{МВ} = t_{отл} \cdot C_{МЧ}, \text{ грн,} \qquad (3.13)$$

де $t_{отл}$ – трудомісткість налагодження програми на ЕОМ, год.

$C_{МЧ}$ – вартість машино-години ЕОМ, грн/год.

$$З_{МВ} = 309{,}77 \cdot 20 = 6195{,}4, \text{ грн.}$$

$$К_{ПО} = 80276{,}25 + 6195{,}4 = 86471{,}65, \text{ грн.}$$

Очікуваний період створення ПЗ:

$$T = \frac{t}{B_k \cdot F_p}, \text{ мес.} \qquad (3.14)$$

де B_k- число виконавців;

$F_р$ – місячний фонд робочого часу (при 40 годинному робочому тижні $F_р$=176 годин).

$$T = \frac{642{,}21}{1 \cdot 176} = 3{,}65, \text{ міс.}$$

Висновки. На розробку даного програмного забезпечення піде 642,21 людино-годин. Тобто, очікувана тривалість розробки складатиме 3,65 місяці при стандартному 40-годинному робочому тижні і 176-годинному робочому місяці. Очікувані витрати на створення ПЗ складатимуть 86471,65 грн.

ВИСНОВКИ

В даній кваліфікаційній роботі була розроблена двовимірна аркадна гра для одного гравця з інтуітивно зрозумілими правилами і високим рівнем оптимізації.

В наш час є доцільним використання комп'ютерних відеоігор для розваг, а саме ця програма за допомогою сучасних методів розробки в середовищі UNITY 3D реалізує таку потребу. Ідеї, закладені в її основу, також можуть бути в подальшому використані для створення нових ігор аналогічного спрямування.

В результаті виконання дипломного проекту досягнуто мету даної роботи: проаналізовані доступні технології та методи розробки для створення ігрових програм та розроблена власна комп'ютерні 2D гра в середовищі UNITY 3D.

Під час виконання даного дипломного проекту були виконані наступні задачі:

- проаналізовано предметну область задачі, що розв'язується;
- проведено порівняння з можливостей існуючих подібних застосунків;
- обрано раціональну структуру і параметри програми;
- написано програмний код мобільного додатку;
- розроблено рекомендації щодо використання програми.

Результатом виконання дипломного проекту є повноцінний працеспроможний додаток, аркадан гра під назвою «Orbs World». Програма реалізована на мові програмування C# в середовищі UNITY 3D.

Також у кваліфікаційній роботі було визначено трудомісткість розробленого програмного продукту (642,21 люд-год), проведений підрахунок вартості роботи по створенню програми (86471,65 грн) та розраховано час на його створення (3,65 міс).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Видеоигра — Википедия - Свободная энциклопедия, 2021 — https://ru.wikipedia.org/wiki/Видеоигра/ . дата звернення: 24.05.2021.
2. Video Game — English Oxford Living Dictionary, 2021 — https://en.oxforddictionaries.com/definition/video_game/ . дата звернення: 25.05.2021.
3. Каманкина М. В. Видеоигры: общая проблематика, страницы истории, опыт интерпретации. — Москва: ГИИ, 2016. — 340 с. — ISBN 978-5-98287-098-8.
4. Деникин, А. А. Звуковой дизайн в видеоиграх. Технологии «игрового» аудио для непрограммистов. — Москва: ДМК Пресс, 2012. — 696 с. — ISBN 978-5-94074-234-0.
5. III, Richard Rouse. Game Design: Theory and Practice, Second Edition (en). Jones & Bartlett Learning. — Edmund, 2018. — 41 с. — ISBN 978-1-44963-345-5.
6. Lindley, Craig; Nacke, Lennart; Sennersten, Charlotte. Dissecting Play — Investigating the Cognitive and Emotional Motivations and Affects of Computer Gameplay. Proceedings of CGAMES 08 — Wolverhampton, UK: University of Wolverhampton, 2021. — 51 с. — ISBN 978-0-9549016-6-0.
7. Oxland, Kevin. Gameplay and Design (en). — Addison-Wesley, 2004. — 14–16c. — ISBN 978-0-32120-467-7.
8. Донован Т. Играй! История видеоигр. — New Jersey, 2014. — 38 с. — ISBN 978-5-9903760-4-5.
9. Juho Hamari, Max Sjöblom. What is eSports and why do people watch it?. — Internet Research ,2017. — 211-232 с. — https://doi.org/10.1108/IntR-04-2016-0085/ . дата звернення: 27.05.2021.
10. Официальный интернет-портал правовой информации. — Москва: МСРФ, 2016. — http://publication.pravo.gov.ru/Document/View/0001201606070022/ . дата звернення: 8.05.2016.
11. John Learned. The Oral History of EVO: The Story of the World's Largest

Fighting Game Tournament. — USgamer, 2017. — https://www.usgamer.net/articles/the-oral-history-of-evo/ . дата звернення: 25.10.2020.

12. WCG Concept. — World Cyber Games, 2020. — http://www.wcg.com/6th/inside/wcgc/wcgc_structure.asp/ . дата звернення: 21.11.2020. — https://www.webcitation.org/65dIUGOTu?url=http://www.wcg.com/6th/inside/wcgc/wcgc_structure.asp/ архівовано 22.02.2021.
13. Іван Бранець. Чому SOLID - важлива складова мислення програміста. — DOU.ua - Спільнота програмістів, 2021. — https://dou.ua/lenta/articles/solid-principles/ . дата звернення: 31.05.2021.
14. Robert C. Martin. "Principles Of OOD". — butUncleBob.com, 2014. — http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod/ . дата звернення: 17.07.2014.
15. Robert C. Martin. "Single Responsibility Principle". — objectmentor.com, 2013. — https://web.archive.org/web/20150206040202/http://www.objectmentor.com:80/resources/articles/srp.pdf/ . архівовано: 2.02.2015.
16. Robert C. Martin. "Getting a SOLID start". — objectmentor.com, 2013. — https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start/ . дата звернення: 19.08.2013.
17. Sandi M. "SOLID Object-Oriented Design". — GORUCO, 2009. — https://www.youtube.com/watch?v=v-2yFMzxqwU/ . дата звернення: 24.05.2015.
18. Robert C. Martin. "Liskov Substitution Principle". — objectmentor.com, 2013. — https://web.archive.org/web/20150905081111/http://www.objectmentor.com/resources/articles/lsp.pdf/ . архівовано: 2.02.2015.
19. Robert C. Martin. "Design Principles and Design Patterns". — Codeer, 2000. — https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf/ . архівовано 21.03.2020.
20. Robert C. Martin. "Dependency Inversion Principle". — objectmentor.com, 2014. — https://web.archive.org/web/20150905081103/http://www.objectmentor.com/resources/articles/dip.pdf/ . архівовано: 2.02.2015.
21. Robert C. Martin. Clean Architecture: A Craftsman's Guide to Software Structure and Design. — Los Angles, 2018. — 56 с. — ISBN 978-0- 13449416-6-6.

КОД ПРОГРАМИ

```
using System;
using UnityEngine;

namespace TEDinc.OrbsWorld
{
   // TODO : separate IObjectData
   public interface IObjectModel
   {
      Vector2 Position { get; }
      bool IsDestoyed { get; }

      float GetAvgRadius();

      float GetMass();
      void SetMass(float mass);

      void Setup(float mass, Vector2 position, Vector2 velocity, IPhysicsParams
physicsParams);

      void Accelerate(Vector2 velocity);
      void InteractWith(IObjectModel otherObject);
      void UpdatePhysics(float deltaTime);
      void Destroy();
   }
}
```

```
namespace TEDinc.OrbsWorld
{
   public interface IObjectsHolder
   {
      IObjectModel[] objects { get; set; }
      void DestoyAll();
      void TryClearDestoyedObjects();
   }
}
```

```
using System;
using UnityEngine;

namespace TEDinc.OrbsWorld
{
   public sealed class OrbModel : IObjectModel
   {
      public Vector2 Position { get; private set; }
      public bool IsDestoyed { get; private set; }
```

```
        private float radius;
        private Vector2 velocity;
        private IPhysicsParams physicsParams;

        public static float GetMass(float radius) =>
            Mathf.PI * radius * radius;
        public float GetMass() =>
            GetMass(radius);
        public void SetMass(float mass) =>
            radius = Mathf.Sqrt(mass / Mathf.PI);
        public float GetAvgRadius() =>
            radius;


        public void Accelerate(Vector2 velocity) =>
            this.velocity += velocity;

        public void InteractWith(IObjectModel otherObject)
        {
            if (otherObject is OrbModel)
                OrbInteractionStreategy(otherObject as OrbModel);
            else
                throw new NotImplementedException();



            void OrbInteractionStreategy(OrbModel other)
            {
                float colideDistance = radius + other.radius - (Position - other.Position).magnitude;

                if (colideDistance > 0)
                {
                    OrbModel smallerOrb = radius > other.radius ? other : this;
                    OrbModel biggerOrb = radius < other.radius ? other : this;

                    float radiusDelta = Mathf.Min(smallerOrb.radius, colideDistance);
                    float smallerOrbMassCashe = smallerOrb.GetMass();
                    smallerOrb.radius -= radiusDelta;
                    biggerOrb.SetMass(biggerOrb.GetMass() + smallerOrbMassCashe -
smallerOrb.GetMass());
                }
            }
        }

        public void UpdatePhysics(float deltaTime)
        {
            velocity = velocity.normalized * Mathf.Max(0f, velocity.magnitude - deltaTime *
physicsParams.FrictionDeceleration);
            Position += velocity * deltaTime;
            Vector2 newPosition = new Vector2(
                Mathf.Clamp(Position.x, physicsParams.Walls.xMin + radius,
physicsParams.Walls.xMax - radius),
```

```
                Mathf.Clamp(Position.y, physicsParams.Walls.yMin + radius,
physicsParams.Walls.yMax - radius));
            if (Mathf.Abs(newPosition.x - Position.x) > 0.01f)
                velocity = new Vector2(-velocity.x, velocity.y);
            if (Mathf.Abs(newPosition.y - Position.y) > 0.01f)
                velocity = new Vector2(velocity.x, -velocity.y);
            Position = newPosition;
        }

        public void Destroy() =>
            IsDestoyed = true;

        public void Setup(float mass, Vector2 position, Vector2 velocity, IPhysicsParams
physicsParams)
        {
            SetMass(mass);
            Position = position;
            this.velocity = velocity;
            this.physicsParams = physicsParams;
        }
    }
}
```

```
namespace TEDinc.OrbsWorld
{
    // TODO : separate IDisplayFactory
    public interface IObjectsFactory
    {
        void Setup(ISpawnParams spawnParams, IPhysicsParams physicsParams,
IDisplayParams displayParams,
            IObjectsPhysics objectsPhysics, IObjectsHolder objectsHolder, IPlayerManager
playerManager);
        void ConstructModels();
        void ConstructDisplays();
    }
}
```

```
using System.Linq;

namespace TEDinc.OrbsWorld
{
    public sealed class ObjectsHolder : IObjectsHolder
    {
        const float minDiffFactor = 1.1f;

        public IObjectModel[] objects { get; set; }
```

```
        public void DestoyAll()
        {
            if (objects != null)
                foreach (IObjectModel item in objects)
                    if (item != null)
                        item.Destroy();
        }

        public void TryClearDestoyedObjects()
        {
            int aliveObjectsCount = 0;

            for (int i = 0; i < objects.Length; i++)
                if (objects[i] == null || objects[i].IsDestoyed)
                    aliveObjectsCount++;

            if (aliveObjectsCount / (float)objects.Length > minDiffFactor)
                objects = objects.Where(o => o != null).ToArray();
        }
    }
}



using UnityEngine;

namespace TEDinc.OrbsWorld
{
    public sealed class ObjectsFactory : IObjectsFactory
    {
        private ISpawnParams spawnParams;
        private IPhysicsParams physicsParams;
        private IDisplayParams displayParams;

        private IObjectsPhysics objectsPhysics;
        private IObjectsHolder objectsHolder;
        private IPlayerManager playerManager;

        public void Setup(ISpawnParams spawnParams, IPhysicsParams physicsParams,
IDisplayParams displayParams,
            IObjectsPhysics objectsPhysics, IObjectsHolder objectsHolder, IPlayerManager
playerManager)
        {
            this.spawnParams = spawnParams;
            this.physicsParams = physicsParams;
            this.displayParams = displayParams;

            this.objectsPhysics = objectsPhysics;
            this.objectsHolder = objectsHolder;
            this.playerManager = playerManager;
        }
```

```
        public void ConstructModels()
        {
            IObjectModel[] objectModels = new OrbModel[spawnParams.TargetOpponentCount];

            for (int i = 0; i < objectModels.Length; i++)
            {
                objectModels[i] = new OrbModel();
                objectModels[i].Setup(
                    Random.Range(spawnParams.MinOpponentMass,
spawnParams.MaxOpponentMass),
                    GetRandomPos(),
                    Random.insideUnitCircle * spawnParams.MaxStartSpeed,
                    physicsParams);
            }

            Vector2 GetRandomPos() =>
                new Vector2(
                    Random.Range(physicsParams.Walls.xMin, physicsParams.Walls.xMax),
                    Random.Range(physicsParams.Walls.yMin, physicsParams.Walls.yMax));

            objectsHolder.objects = objectModels;
        }

        public void ConstructDisplays()
        {
            foreach (Transform child in spawnParams.ObjectsParent)
                GameObject.Destroy(child.gameObject);

            for (int i = 0; i < objectsHolder.objects.Length; i++)
                if (objectsHolder.objects[i] != null)
                {
                    ObjectDisplay orbDisplay =
GameObject.Instantiate(spawnParams.ObjectDisplayPrefab, spawnParams.ObjectsParent);
                    orbDisplay.Setup(objectsHolder.objects[i], objectsPhysics, displayParams,
playerManager);
                }
        }
    }
}
```

```
namespace TEDinc.OrbsWorld
{
    public delegate void Notify();

    public interface IObjectsPhysics
    {
        event Notify OnPhysicUpdateDone;
        void Setup(IObjectsHolder objectsHolder, IPhysicsParams physicsParams);
        // TODO : make async
        void UpdatePhysics(float deltaTime);
```

```
    }
}



namespace TEDinc.OrbsWorld
{
    public sealed class ObjectsPhysics : IObjectsPhysics
    {
        public event Notify OnPhysicUpdateDone;

        private IObjectsHolder objectsHolder;
        private IPhysicsParams physicsParams;

        private IObjectModel[] objects => objectsHolder.objects;


        public void Setup(IObjectsHolder objectsHolder, IPhysicsParams physicsParams)
        {
            this.objectsHolder = objectsHolder;
            this.physicsParams = physicsParams;
        }

        public void UpdatePhysics(float deltaTime)
        {
            InvokeObjectsUpdatePhysics();
            objectsHolder.TryClearDestoyedObjects();
            InteratAllObjects();

            OnPhysicUpdateDone?.Invoke();



            void InvokeObjectsUpdatePhysics()
            {
                for (int i = 0; i < objects.Length; i++)
                    if (objects[i] == null)
                        continue;
                    else if (objects[i].IsDestoyed)
                        objects[i] = null;
                    else if (objects[i].GetAvgRadius() < physicsParams.DestroyRadius)
                    {
                        objects[i].Destroy();
                        objects[i] = null;
                    }
                    else
                        objects[i].UpdatePhysics(deltaTime);
            }


            // TODO : optimize
```

```
        void InteratAllObjects()
        {
            for (int i = 0; i < objects.Length; i++)
                if (objects[i] != null)
                    for (int j = i + 1; j < objects.Length; j++)
                        if (objects[j] != null)
                            objects[i].InteractWith(objects[j]);
        }
    }
}
```

```
using System;
using UnityEngine;

namespace TEDinc.OrbsWorld
{
    [Serializable]
    public sealed class DisplayParams : IDisplayParams
    {
        public Color PlayerColor => playerColor;
        public Color OponentSmallerColor => oponentSmallerColor;
        public Color OponentBiggherColor => oponentBiggherColor;
        public float GradientMassFactor => gradientMassFactor;


        [SerializeField]
        private Color playerColor = Color.gray;
        [SerializeField]
        private Color oponentSmallerColor = Color.blue;
        [SerializeField]
        private Color oponentBiggherColor = Color.red;
        [SerializeField]
        private float gradientMassFactor = 1.05f;
    }
}
```

```
using UnityEngine;

namespace TEDinc.OrbsWorld
{
    public interface IDisplayParams
    {
        Color PlayerColor { get; }
        Color OponentSmallerColor { get; }
        Color OponentBiggherColor { get; }
        float GradientMassFactor { get; }
```

```
    }
}
```

```
using UnityEngine;

namespace TEDinc.OrbsWorld
{
    public interface IPhysicsParams
    {
        float DestroyRadius { get; }
        float FrictionDeceleration { get; }
        float PlayerAcceleration { get; }
        Rect Walls { get; }
    }
}
```

```
using UnityEngine;

namespace TEDinc.OrbsWorld
{
    public interface ISpawnParams
    {
        ObjectDisplay ObjectDisplayPrefab { get; }
        Transform ObjectsParent { get; }


        int TargetOpponentCount { get; }
        float MinOpponentMass { get; }
        float MaxOpponentMass { get; }
        float MaxStartSpeed { get; }
        float PlayerMass { get; }
        float ClearRadiusForPlayer { get; }
    }
}
```

```
using System;
using UnityEngine;

namespace TEDinc.OrbsWorld
{
    [Serializable]
    public sealed class PhysicsParams : IPhysicsParams
    {
        public float DestroyRadius => destroyRadius;
```

```
        public float FrictionDeceleration => frictionDeceleration;
        public Rect Walls => wallsRectTransform.rect;
        public float PlayerAcceleration => playerAcceleration;

        [SerializeField]
        private float destroyRadius = 5f;
        [SerializeField]
        private float frictionDeceleration = 50f;
        [SerializeField]
        private float playerAcceleration = 1000f;
        [SerializeField]
        public RectTransform wallsRectTransform;
    }
}




using System;
using UnityEngine;

namespace TEDinc.OrbsWorld
{
    [Serializable]
    public sealed class SpawnParams : ISpawnParams
    {
        public ObjectDisplay ObjectDisplayPrefab => objectDisplayPrefab;
        public Transform ObjectsParent => objectsParent;

        public int TargetOpponentCount => targetOponentCount;
        public float MinOpponentMass => minOpponentMass;
        public float MaxOpponentMass => maxOpponentMass;
        public float MaxStartSpeed => maxStartSpeed;

        public float PlayerMass => playerMass;
        public float ClearRadiusForPlayer => clearRadiusForPlayer;

        [SerializeField]
        private ObjectDisplay objectDisplayPrefab;
        [SerializeField]
        private Transform objectsParent;

        [Header("Opponents")]
        [SerializeField]
        private int targetOponentCount = 1000;
        [SerializeField]
        private float minOpponentMass = 20f;
        [SerializeField]
        private float maxOpponentMass = 100f;
        [SerializeField]
        private float maxStartSpeed = 10f;

        [Header("Player")]
```

```
        [SerializeField]
        private float playerMass = 1000;
        [SerializeField]
        private float clearRadiusForPlayer = 20f;
    }
}



namespace TEDinc.OrbsWorld
{
    public interface IPlayerManager
    {
        event Notify OnPlayerWin;
        event Notify OnPlayerDefeat;

        IPlayerModel player { get; }
        void Setup(ISpawnParams spawnParams, IPhysicsParams physicsParams, IObjectsHolder
objectsHolder);
        void CreateBeferePlayerCleaner();
        void CreatePlayerInstaedOfCleaner();
        void Update(float deltaTime);
    }
}



namespace TEDinc.OrbsWorld
{
    public interface IPlayerModel
    {
        // TODO : change to IObjectData after implement
        IObjectModel model { get; }
        void Setup(IObjectModel model);
    }
}



using UnityEngine;

namespace TEDinc.OrbsWorld
{
    public sealed class PlayerManager : IPlayerManager
    {
        public event Notify OnPlayerWin;
        public event Notify OnPlayerDefeat;
        public IPlayerModel player { get; private set; }

        private ISpawnParams spawnParams;
```

```
        private IPhysicsParams physicsParams;
        private IObjectsHolder objectsHolder;

        private const int createIndex = 0;


        public void CreateBeferePlayerCleaner() =>
            CreateMock(OrbModel.GetMass(spawnParams.ClearRadiusForPlayer));

        public void CreatePlayerInstaedOfCleaner()
        {
            CreateMock(spawnParams.PlayerMass);
            player = new PlayerModel();
            player.Setup(objectsHolder.objects[createIndex]);
        }

        private void CreateMock(float mass)
        {
            objectsHolder.objects[createIndex].Destroy();
            objectsHolder.objects[createIndex] = new OrbModel();
            objectsHolder.objects[createIndex].Setup(mass, Vector2.zero, Vector2.zero,
physicsParams);
        }

        public void Setup(ISpawnParams spawnParams, IPhysicsParams physicsParams,
IObjectsHolder objectsHolder)
        {
            this.spawnParams = spawnParams;
            this.physicsParams = physicsParams;
            this.objectsHolder = objectsHolder;
        }

        public void Update(float deltaTime)
        {
            if (ChecPlayerDefeat() || CheckPlayerWin())
                return;

            Accelerate();


            bool ChecPlayerDefeat()
            {
                if (player.model.IsDestoyed)
                    OnPlayerDefeat?.Invoke();

                return player.model.IsDestoyed;
            }

            bool CheckPlayerWin()
            {
                float maxMass = 0f;
                float plyerMass = player.model.GetMass();
```

```
                foreach (IObjectModel model in objectsHolder.objects)
                    if (model != null)
                    {
                        float mass = model.GetMass();
                        if (mass > maxMass)
                        {
                            maxMass = mass;
                            if (maxMass > plyerMass)
                                return false;
                        }
                    }

                    OnPlayerWin?.Invoke();
                return true;
            }

            void Accelerate()
            {
                if (!Input.GetKey(KeyCode.Mouse0))
                    return;
                // TODO : remove dependency from Camera.main
                Vector3 worldPos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
                Vector2 localPos = spawnParams.ObjectsParent.InverseTransformPoint(worldPos);
                Vector2 moveDirection = player.model.Position - localPos;
                moveDirection.Normalize();
                player.model.Accelerate(moveDirection * deltaTime *
physicsParams.PlayerAcceleration);
            }
        }
    }
}



using UnityEngine;

namespace TEDinc.OrbsWorld
{
    public sealed class PlayerModel : IPlayerModel
    {
        public IObjectModel model { get; private set; }

        public void Setup(IObjectModel model) =>
            this.model = model;
    }
}



using System.Linq;
using UnityEngine;
```

```
using UnityEngine.UI;

public sealed class FpsDisplay : MonoBehaviour
{
   public Text label;
   private int[] queue = new int[50];
   private int queueIndex;

#if !DEVELOPMENT_BUILD && !UNITY_EDITOR
   private void Awake() =>
      Destroy(gameObject);
#endif

   public void Update()
   {
      int currnet = (int)(1f / Time.unscaledDeltaTime);
      queue[queueIndex++] = currnet;
      queueIndex %= queue.Length;
      label.text = $"CUR {currnet}\nAVG {queue.Sum() / queue.Length}";
   }
}
```

```
using UnityEngine;
using UnityEngine.UI;

namespace TEDinc.OrbsWorld
{
   // TODO : inherit from interface and remove all external dependencies to class
   public sealed class ObjectDisplay : MonoBehaviour
   {
      [SerializeField]
      private Image image;
      [SerializeField]
      private float scaleFactor = 100f;
      [SerializeField, Range(0.001f, 1f)]
      private float scaleLerpfactor = 0.05f;
      [SerializeField, Range(0.001f, 1f)]
      private float colorLerpfactor = 0.05f;

      // TODO : change to IObjectData after implementation
      private IObjectModel model;
      private IObjectsPhysics objectsPhysics;
      private IDisplayParams displayParams;
      private IPlayerManager playerManager;

      public void Setup(IObjectModel model, IObjectsPhysics objectsPhysics, IDisplayParams
displayParams, IPlayerManager playerManager)
      {
         this.model = model;
         this.objectsPhysics = objectsPhysics;
```

```
        this.displayParams = displayParams;
        this.playerManager = playerManager;

        objectsPhysics.OnPhysicUpdateDone += AfterPhysicsDone;
        transform.localScale = Vector3.one / scaleFactor;
    }

    // TODO : change to support of async physics update
    private void AfterPhysicsDone()
    {
        if (model.IsDestoyed)
        {
            model = null;
            objectsPhysics.OnPhysicUpdateDone -= AfterPhysicsDone;
            Destroy(gameObject);
            return;
        }

        TweenPosition();
        TweenColor();



        void TweenPosition()
        {
            transform.localPosition = model.Position;
            transform.localScale = Vector3.Lerp(
                transform.localScale,
                Vector3.one * model.GetAvgRadius() / scaleFactor,
                scaleLerpfactor);
        }

        void TweenColor()
        {
            if (model == playerManager.player.model)
                image.color = displayParams.PlayerColor;
            else
            {
                float massRatio = model.GetMass() / playerManager.player.model.GetMass();
                float colorFactor = Mathf.InverseLerp(
                    1f / displayParams.GradientMassFactor,
                    displayParams.GradientMassFactor,
                    massRatio);
                Color targetColor = Color.Lerp(
                    displayParams.OponentSmallerColor,
                    displayParams.OponentBiggherColor,
                    colorFactor);

                image.color = Color.Lerp(image.color, targetColor, colorLerpfactor);
            }
        }
    }
}
```

```
}



using UnityEngine;

namespace TEDinc.OrbsWorld
{
   // TODO : separate UI manager
   public sealed class WorldManager : MonoBehaviour
   {
      [SerializeField]
      private SpawnParams spawnParams;
      [SerializeField]
      private PhysicsParams physicsParams;
      [SerializeField]
      private DisplayParams displayParams;
      [SerializeField]
      private GameObject winPopup;
      [SerializeField]
      private GameObject defeatPopup;

      private IObjectsFactory objectsFactory;
      private IObjectsPhysics objectsPhysics;
      private IObjectsHolder objectsHolder;
      private IPlayerManager playerManager;

      private bool isPaused;

      private void Start()
      {
         objectsFactory = new ObjectsFactory();
         objectsPhysics = new ObjectsPhysics();
         objectsHolder = new ObjectsHolder();
         playerManager = new PlayerManager();

         objectsFactory.Setup(spawnParams, physicsParams, displayParams, objectsPhysics,
objectsHolder, playerManager);
         objectsPhysics.Setup(objectsHolder, physicsParams);
         playerManager.Setup(spawnParams, physicsParams, objectsHolder);

         playerManager.OnPlayerWin += OnPlayerWin;
         playerManager.OnPlayerDefeat += OnPlayerDefeat;

         RestartLevel();
      }

      private void OnDestroy()
      {
         playerManager.OnPlayerWin -= OnPlayerWin;
         playerManager.OnPlayerDefeat -= OnPlayerDefeat;
      }
```

```
        public void RestartLevel()
        {
            winPopup.SetActive(false);
            defeatPopup.SetActive(false);

            objectsHolder.DestoyAll();
            objectsFactory.ConstructModels();
            playerManager.CreateBeferePlayerCleaner();
            objectsPhysics.UpdatePhysics(0f);
            playerManager.CreatePlayerInstaedOfCleaner();
            objectsFactory.ConstructDisplays();

            isPaused = false;
        }

        private void Update()
        {
            if (isPaused)
                return;

            objectsPhysics.UpdatePhysics(Time.deltaTime);
            playerManager.Update(Time.deltaTime);
        }

        private void OnPlayerWin()
        {
            isPaused = true;
            winPopup.SetActive(true);
        }
        private void OnPlayerDefeat()
        {
            isPaused = true;
            defeatPopup.SetActive(true);
        }
    }
}
```

ДОДАТОК Б

ВІДГУК КЕРІВНИКА ЕКОНОМІЧНОГО РОЗДІЛУ

ДОДАТОК В

ПЕРЕЛІК ФАЙЛІВ НА ДИСКУ

ПЕРЕЛІК ДОКУМЕНТІВ НА МАГНІТНОМУ НОСІЇ

Ім’я файлу	Опис
Пояснювальні документи	
Кваліфікаційна робота Бобко.docx	Пояснювальна записка до кваліфікаційної роботи в форматі Word
Кваліфікаційна робота Бобко.pdf	Пояснювальна записка до кваліфікаційної роботи в форматі PDF
Програма	
Бобко.zip	Архів. Містить коди програми і скомпільовану програму
Презентація	
Бобко.ppt	Презентація кваліфікаційної роботи

ВІДГУК
на кваліфікаційну роботу бакалавра
на тему:
"Розробка комп'ютерної аркадної двовимірної гри у середовищі UNITY 3D"
студента групи 121-17з-1 Бобка Михайла Олексійовича

Розроблена в кваліфікаційній роботі програма призначена для розваги користувача за допомогою цікавого і невибагливого ігрового додатку на смартфоні.

Як інструмент для проектування і реалізації був використане середовище розробки UNITY 3D за допомогою мови програмування C#.

Практична значимість створення даного програмного продукту полягає в можливості використання даного додатку для проведення кібер-спортивних турнірів серед студентів та викладачів кафедри.

Працездатність представленої програми підтверджена налагоджувальними випробуваннями та тестуванням програми.

В економічному розділі визначено трудомісткість розробленої ІС, проведений підрахунок вартості роботи по створенню програми та розраховано час на його створення.

Тема кваліфікаційної роботи безпосередньо пов'язана з об'єктом діяльності бакалавра за напрямом підготовки 121 Інженерія програмного забезпечення.

Оформлення пояснювальної записки до роботи виконано відповідно до стандартів на програмну документацію.

Кваліфікаційну роботу виконано самостійно та заслуговує оцінки 82 бала «добре», а студент Бобко Михайло Олексійович заслуговує присвоєння йому кваліфікації бакалавра за спеціальністю «фахівець в галузі інформаційних технологій».

Керівник кваліфікаційної роботи
доцент каф. ПЗКС, к.т.н. **С.Д. Приходченко**

РЕЦЕНЗІЯ
на кваліфікаційну роботу бакалавра
на тему:
"Розробка комп’ютерної аркадної двовимірної гри у середовищі UNITY 3D"
студента групи 121-17з-1 Бобка Михайла Олексійовича

Кваліфікаційну роботу на тему «Розробка комп’ютерної аркадної двовимірної гри у середовищі UNITY 3D» виконаний в повному обсязі, відповідно до технічного завдання.

Мета кваліфікаційної роботи: Метою дипломного проекту є аналіз доступних технологій і методів розробки для створення ігрових програм жанрового різновиду ігор-аркад і розробка власної комп'ютерні 2D гри в середовищі UNITY 3D.

У пояснювальній записці розглянуто необхідність створення і сфера застосування розробленої, виконано постановку завдання, опис вхідних і вихідних даних, розроблено інформаційне забезпечення системи, наведені загальні відомості про додаток, визначені джерела, використані при розробці.

Вважаю завдання і зміст кваліфікаційної роботи відповідним для перевірки ступеня підготовленості Бобка М.О. за напрямом 121 Інженерія програмного забезпечення.

Список літератури, наведений в роботі, налічує більше 21 джерело, що свідчить про вміння автора працювати з літературою та іншими джерелами інформації. Якість оформлення кваліфікаційної роботи можна визнати добре, з незначними недоліками.

Кваліфікаційну роботу виконано самостійно та заслуговує оцінки «добре», а студент Бобко Михайло Олексійович заслуговує присвоєння йому кваліфікації бакалавра за спеціальністю «фахівець в галузі інформаційних технологій».

Рецензент дипломного проекту
доцент каф. БІТ, к.т.н. **О.В. Герасіна**