ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра комп‘ютерної інженерії

Пояснювальна записка

до бакалаврської роботи
на ступінь вищої освіти бакалавр

на тему: «Розробка гри – стратегії з використанням технології Unity»

Виконав: студент 5 курсу, групи ППЗ-51
спеціальності
121 ___Інженерія програмного забезпечення
(шифр і назва спеціальності)
_____Галеєв В.В._____
(прізвище та ініціали)
Керівник_____Гаманюк І.М._____
(прізвище та ініціали)
Рецензент_____
(прізвище та ініціали)
Нормконтроль_____
(прізвище та ініціали)

Київ – 2022

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Навчально-науковий інститут Телекомунікацій

Кафедра Інженерії програмного забезпечення_____
Ступінь вищої освіти - «Бакалавр»____
Напрям підготовки -121 – Інженерія програмного забезпечення"

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____О.В. Негоденко
“____”_____2022 року

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

_____Галеєву Владиславу Володимировичу_____
(прізвище, ім‘я, по батькові)

1.Тема роботи: «Розробка гри – стратегії з використанням технології Unity»
Керівник роботи старший викладач кафедри ІПЗ Гаманюк І.М. ,
(прізвище, ім‘я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від “18” лютого 2022 р. №__.
2. Строк подання студентом роботи 03.06.2022.
3. Вхідні дані до роботи:
- 3.1. Положення побудови ігр-стратегій
- 3.2. Методи побудови ігр-стратегій
- 3.3. Існуючі ігри-стратегії
- 3.4. Науково-технічна література

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити).
- 4.1. Аналіз предметної галузі та постановка задачі з розробки розважального програмного забезпечення гри-стратегії
- 4.2. Розробка проекту програмного забезпечення гри-стратегії
- 4.3. Якість та тестування програмного забезпечення
- 4.4. Висновки

5. Перелік графічного матеріалу
1. Обкладинки ігор
2. Діаграми прецедентів, схеми
3. Ієрархія класів
4. Скріншоти ігр аналогів, а також скріншоти розробленної програми

6. Дата видачі завдання___11.04.2022_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	11.04-14.04	
2	Вивчення та аналіз задачі	15.04-17.04	
3	Розробка структури додатку	18.04-21.04	
4	Розробка дизайну та графічних елементів	22.04-25.04	
5	Програмна реалізація системи	26.04-05.05	
6	Налагодження програми	05.05-07.05	
7	Вступ, висновки, реферат	07.05-10.05	
8	Розробка обов’язкових демонстраційних матеріалів	11.05-15.05	
9	Попередній захист роботи	16.05-01.06	
10	Подання роботи в деканат	03.06	

Студент _____ Галеєв В.В
(підпис) (прізвище та ініціали)

Керівник роботи _____ Гаманюк І.М.
(підпис) (прізвище та ініціал)

РЕФЕРАТ

Дипломна робота виконана на 73 сторінках тексту, містить 24 рисунки, 19 таблиць, 1 додаток та список використаних джерел з 11 найменувань.

Ключеві слова: Гра-стратегія, комп’ютерна стратегія, стратегія в реальному часі, Unity, C#

Об’єкт дослідження - підвищення закцікавленості у цільової аудиторії до жанру, та розвиток жанру комп’ютерних стратегій за допомогою розробки власного розважального програмного забезпечення - гри-стратегії з унікальними особливостями

Предмет дослідження - розважальне програмне забезпечення у вигляді ігор-стратегій.

Мета роботи – розробка розважального програмного забезпечення у вигляді гри-стратегії з використанням технології Unity та створенного на мові C#.

Аналіз особливостей процесу роботи розважальних ігор та їх існуючих екземлярів дозволяє сформулювати вимоги до майбутнього програмного продукту. Розважальна комп'ютерна стратегія повинна бути реалізована для десктопної платформи, бути реализованною за допомогою ігрового рушія Unity та мови програмування C#. У відповідності з поставленою метою для вирішення технічної задачі в роботі вирішено такі завдання:

- Аналіз доступних існуючих розважальних ігор стратегій та історії їх виникнення, а також аналіз їх недоліків та переваг.
- Аналіз технічних засобів для розробки розжаважального програмного забезпечення та вибір оптимально зручного рішення для виконання поставленного завдання.
- Аналіз сучасного ринку ігор та потреби в створенні нових стратегічних ігор, аналіз зацікавленності аудиторії.
- На основі результатів аналізу досліджен розроблений розважальний додаток для персональних комп’ютерів у вигляді гри-стратегії в реальному часі.

Результатом роботи є розробка розважального програмного забезпечення у вигляді гри-стратегії з орієнтацією на аудиторію поціновувачів жанру компьютерних стратегій в реальному часі.

Розроблене програмне забезпечення може функціонувати під управлінням операційних системи Windows 7, чи більш сучасних ОС.

ЗМІСТ

ВСТУП

На 2021 рік згідно статистичних данних компанії Newzoo що займається професійним збором та аналізом ігрової статистики, ігрова індустрія за 2021 рік поставила новий рекорд у виручці, яка досягла 180,3 мілліарда долларів, що означає ріст у 1.4% у порівнянні з минулим роком, а загальна кількість гравців сягнула 3х мілліардів. Індустрія давно вже давно стоїть поряд з такими гігантами як кіноідустрія та індустрія спортивних розваг, та навіть випереджає їх у деяких аспектах, та щороку привертає увагу у все більшої аудиторії.

Постійний ріст ігрової індустрії спричиненний постійним розвитком технологій робить її однією з дуже перспективних галузей роботи, завдяки цьому сьогодні серед ігор існує чимало різноманітних жанрів, тут знайдеться майже все навіть для самого вибагливого гравця, не обійшов і стороною цей ріст і комп'ютерні стратегії що вже досить давно мають свою сформовану аудиторію, тим не менш незважаючи на досить великий розвиток усієї індустрії в цілому, жанр комп’ютерних стратегій не має такого розквіту на відміну від деяких інших, це відбувається через часткову стагнацію ідей, тим не меньш аудиторії досі приділяє цьому жанру увагу через його в своєму роді унікальність та цікасть ігрового процессу, саме тому гравці з нетерпінням чекають на новий виток розвитку стратегій та з пильністю придивляється до всього нового що з’является на ринку.

Результатом роботи є розробка розважального програмного забезпечення у вигляді гри-стратегії з орієнтацією на аудиторію поціновувачів жанру компьютерних стратегій в реальному часі, завдяки чому до жанру комп’ютерних стратегій може бути знову привернута увага її цільової аудиторії, що також може спричинити новий розквіт жанру комп’ютерних стратегій. Розроблене гра буде функціонувати під управлінням комп’ютера на базі операційної системи Windows 7, чи більш сучасних ОС.

Для реалізації було використано середовище розробки Microsoft Visual Studio 2019, мова програмування C# , а також ігровий рушій Unity 2021.3.1f1.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ З РОЗРОБКИ ГРИ-СТРАТЕГІЇ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ UNITY

1.1 Аналіз структури ігр-стратегій

Ігрова індустрія демонструє ріст, і з кожним роком стає дедалі більшою, особливий період її розквіту прийшовся на глобальну пандемію COVID19, в період якої в зв’язку з тим що більшість людей перейшли на дистанційну роботу та навчання, а також в зв’язку з закриттям більшості розважальних центрів все більше уваги стали привертати саме ігри через їх високу доступність і досить демократичні ціни. Комп’ютерних стратегій займають далеко не останне місце в ігровій індустрії в цілому, хоч і не мають такого швидкого розвитку на відміну від деяких інших жанрів. Загалом це відбувається через часткову стагнацію ідей, тим не меньш аудиторії досі приділяє цьому жанру увагу через його в своєму роді унікальність, і саме тому гравці з нетерпінням чекає на новий виток розвитку стратегій та з пильністю придивляється до всього нового що з’является на ринку.

Відеогра — це електронна гра, в ігровому процесі якої гравець використовує інтерфейс користувача, для отримування зворотної інформації з відеопристрою. Електронні пристрої, що зазвичай використовуються для ігор, називаються ігровими платформами, наприклад, до таких платформ належить персональний комп'ютер та ігрова консоль. Зазвичай пристоєм для введення данних в іграх є ігрові контролери, до яких належать наприклад, джойстик, клавіатура та мишка, геймпад або сенсорний екран.

Стратегічна гра — це жанр відеоігор, в якому основним елементом перемоги є стратегічне мислення і планування своїх подальших дій. Існують чимало стратегій на різну тематику, наприклад є військові(Total War, Command & Conquer: Generals), економічні(Factorio, Caesar, Planet Coaster) суспільнознавчі(серія ігор Civilization) тощо.

Загалом стратегії поділяють на такі два основні види:

Покрокові стратегії (англ. Turn-Based Strategy, або скорочено TBS) —

ігровий процес у них поділено на кроки, впродовж яких кожен з гравців має можливість виконати обмежену кількість дій(Рис.1).

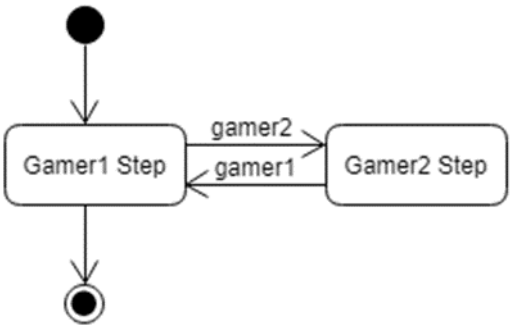


Рисунок 1.1 - Діаграма станів покрокової стратегії.

Кроки в такій грі відбуваються почергово: спершу ходить один з гравеців, потім за ним наступний, в деяких випадках крок завершується тільки тоді, коли всі гравці виконали свої дії. Покрокові стратегії беруть свій початок ще від настільних ігор, де гравці також ходять по черзі.

Стратегії в реальному часі (англ. Real Time Strategy, RTS) — ігровий процес триває безперервно і гравці можуть виконувати одночасні дії(Рис.2)

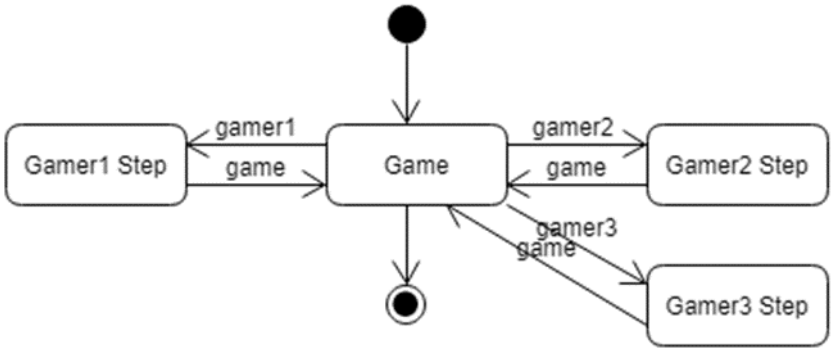


Рисунок 1.2 - Діаграма станів стратегії в реальному часі

Цей жанр відрізняється більшим динамізмом і ухилом до тактики. Час у самій грі тут може не відповідати реальному, а масштабуватися — дії, що в реальності зайняли б години чи дні (наприклад будівництво чи переміщення військ), у грі можуть тривати лічені хвилини.

Також інколи зустрічається їх комбінування — покрокове вирішення певних задач на стратегічній мапі з боями у реальному часі на тактичній. Окрім ділення на два основні види за час свого розвитку стратегії сформували декілька окремих піджанрів:

Глобальна стратегія — піджанр стратегій, в якому гравець бере на себе роль правителя у великому масштабі: імперії, країни, планети чи навіть галактики. Англомовна назва цього жанру — 4X, що відображає чотири основні складові ігрового процесу: explore, expand, exploit, exterminate. (досліджувати, розширювати, розробляти, винищувати), таким чином гравець повинен досліджувати світ, розвивати технології, добувати ресурси та воювати арміями.

Артилерія — тут зазвичай відтворюються невеликі битви, наприклад танкові, де гравцю слід розрахувати траєкторію польоту снаряду, враховуючи силу вітру та інші фактори. Переважно ігри цього піджанру покрокові.

Варгейм — ігровий процес зосереджено на боях, він може відбуватися як покроково, так і в реальному часі.

Баштовий захист — основним завданням гравця є протистояння наступаючим впродовж декількох хвиль ворогам за допомогою будівництва різноманітних оборонних споруд, що атакують повз проходящі юніти противнки в свому ефективному радіусі. Ігровий процес зазвичай відбувається в реальному часі, а економічна складова мінімальна — оборонні споруди створюються за ресурси, що надходять автоматично або нараховуються за знищення ворогів.

МОВА — гравець контролює тільки одного власного персонажа в складі однієї з двох протиборчих команд. Основним завданням ігр такого виду є знищення ключової інфраструктури на базі противника. Будівництво при цьому відсутнє, або обмежене невеликими тимчасовими спорудами.

Автобій або автошахи — зосереджені на боях, що відбуваються на полі поділеному на клітинки, сам бій відбувається автоматично без контролю з боку гравців, а його результати залежать від того, які бойові одиниці будуть обрані та як розташовані.

1.2 Дослідження та аналіз існуючих ігор-стратегій

Ігрова індустрія з кожним роком розвивається швидше та стає все більше, сьогодні вже важко уявити сучасний світ без мобільних ґаджетів, комп'ютерів,

ігрових приставок та іншої розважальної техніки, яка формує наше дозвілля у вільний час. Після технічної революції та приходу комп'ютерів у кожен дім, почався процес активного розвитку ігрової індустрії, розробка комп'ютерних ігор перетворилась на цілу індустрію з безліччю напрямків, одним з таких напрямків стали стратегічні ігри, або ігри-стратегії, які дозволяють гравцям розвивати в собі стратегічні навички та дають змогу розвивати критичне мислення. Загалом історії виникнення стратегічних ігр налічує вже не один десяток століть, людству завжди було цікаво робити своє дозвілля різноманітним, особливої популярності набували різноманітні інтелектуальні ігри по типу Шахів, Го, Нард та інших їх аналогів, саме подібні ігри можно заслужено вважати першими представниками подібного жанру.

Перші саме комп’ютерні відеоігри-стратегії почали з’являтися у 80х з розвитком персональних комп’ютерів та портативних приставок. На сьогоднішній час жанр комп'ютерної стратегій розподіляється на Покрокові стратегії(TBS) та на Стратегії в реальному часі (RTS). У зв’язку з тим що проект дипломної роботи був виконаний у якості RTS стратегії розглянемо їх більш детально:

Одним з перших родоначальників жанру RTS можна заслужено вважати гру Herzog Zwei (Герцог Цвей), що вийшла в 1989 році за видавництва Sega Enterprises Ltd.

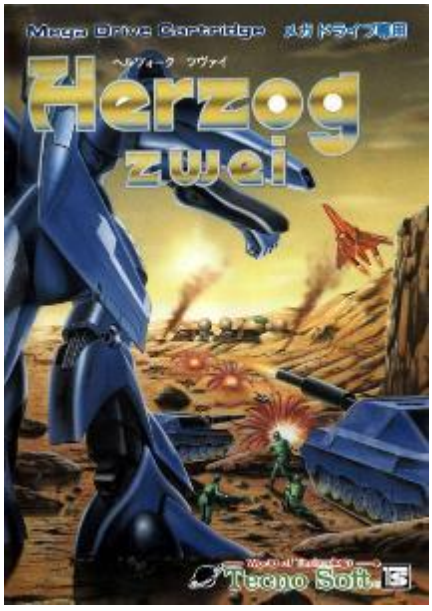


Рисунок 2.1 Обкладинка гри Herzog Zwei

Вона була розрахована на двох гравців, основною задачею гравця було знищення бази ворога. Кожному гравцю надавалась головна база та декілька підконтрольних баз, а також на мапі були присутні нейтральні бази

Рисунок 2.2 Нейтральна база

Єдиний підрозділ що був під командуванням гравця був його боєць, котрий може змінювати форму (міг ставати повітряним підрозділом чи наземним). Основним фокусом цієї гри був саме боєць під керуванням гравця, а всі інші виступали у ролі підтримки.

Рисунок 2.3 Два гравця б'ються один з одним

Гра являлась гібридом action/стратегії, але ідея командування індивідуальним підрозділом в реальному часі з віддачею наказів і слідкуванням за ходом їх виконання досі не зустрічалась в такому масштабі. Значною частиною гри є ваш боєць, в своїй бойовій формі він потребує більше ресурсів, але сражається сильніше, пехота та техніка коштують грошей, і на задньому плані доводиться керувати як розходом ресурсів так і грошовими потоками. Herzog

Zwei виявив велику кількість рис, які стали характерними для сучасних стратегій в реальному часі, наприклад битва за бази.

Але гра мало також і деякі відмінності, наприклад на відміну від сучасників місце розташування баз було фіксоване, і вони не можуть бути відбудовані заново чи відремонтовані.

Справжнім же піонером жанру комп'ютерних RTS стратегій стала гра студії Westwood Dune II, що вийшла за видавництва Virgin Interactive у 1992 році.

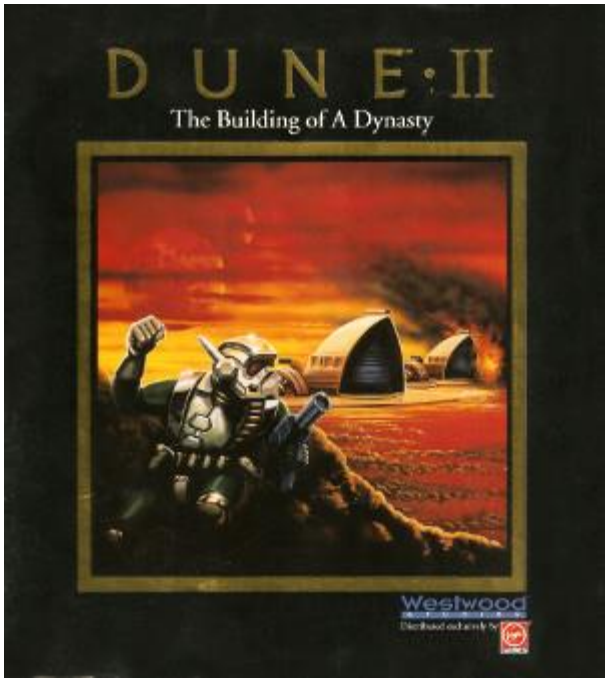


Рисунок 2.4 Обкладинка гри Dune II

Dune II основана на знайомій по роману Френка Херберта місцевості і має в собі дуже багато деталей з книги. Гра перевершує Herzog Zwei в деяких аспектах, котрі можуть показатись тривіальними зараз, але саме ці аспекти і легли в основу жанру RTS. По-перше ви можете будувати базу де завгодно, та будувати у будь-якому порядку, по-друге хід гри залежить від побудованих структур, так для побудови харвестеров і танків потрібно побудувати Важку Фабирку(Heavy Factory), але щоб збудувати її потрібно мати Легку Фабрику (Light Factory), по-третє в Dune II протиборчі сторони мають різноманітні підрозділи так зброю.

Рисунок 2.5 База гравця з побудованними спорудами

Але не обійшлось і без деяких особливостей що були виправлені в більш сучасних стратегіях, так наприклад в Dune II є ліміт на кількість підрозділів в 25 одиниць, що не дивно дивлячись на «залізо» того часу. Тим не менш в цій грі вже були помітні всі характеристики жанру RTS стратегій таких які ми їх знаємо зараз.

Наприкінці 1994 року розробники з Blizzard випустили Warcraft: Orcs & Humans, що переніс стратегію реального часу з галузі наукової фантастики в казковий світ фентезі.

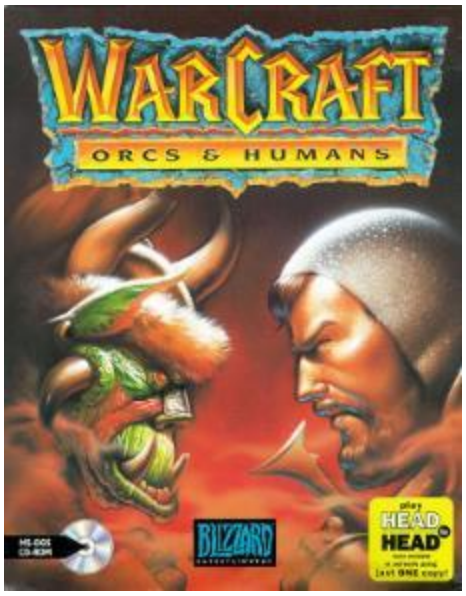


Рисунок 2.6 Обкладинка гри

Уникаючи футуристичної техніки Dune, Warcraft використовує битви, орієнтовані на рукопашні поєдинки. Таким чином, перебіг гри кардинально

змінюється, такого ще до цього не було у стратегіях реального часу. Як і раніше, найсильнішими підрозділами залишаються ті, що б'ють з відстані. Ефективною тактикою в одиночній грі стає створення достатньої кількості рукопашних підрозділів для захисту тих підрозділів, які можуть здалеку завдавати сильних ушкоджень. У грі присутні два ресурси: дерево і золото, а більшість будівель потребує обидва ресурси. Тут все ще чудово працює модель "збирай, стрій, завойовуй", жанр повільно, але вірно еволюціонує.

Війська обох сторін дуже схожі одна на одну. Наприклад, їх можна порівняти в однакові категорії (робочий, воїн, наїзник/лицар, маг і т.д.), а структури, хоч і називаються по-різному, мають однакову функціональність.

Рисунок 2.7 Селище Орків

Рисунок 2.8 Селище Людей

Але не дивлячись на всю іноваційність та переваги Warcraft все ж не був позбавлений недоліків, так наприклад штучний інтелект (AI) у Warcraft був далекий від досконалості, вороги наосліп атакують при будь-якому зіткненні, тому їх легко було виманити і знищити. Понад те, пошук вірного шляху до обраної точки був зроблений дуже посередньо. Через такі недоліки як тільки гравець розумів роботу AI, проходження гри в режимі одного користувача сильно

полегшувалося. Але незважаючи на всі ці недліки Warcraft підтримував багатокористувацьку гру по серійному порту або модемному з'єднанню, що зробило його першою багатокористувацькою RTS стратегією на комп'ютері. Неможливо також не відзначити й генератор випадкових карт, що вперше тут з'явився.

Осінью 1995 року вийшла гра Command & Conquer (Командуй та завойовуй) за видавництва Virgin Interactive, що стала кульмінацією роботи Westwood, яка тривала кілька років.

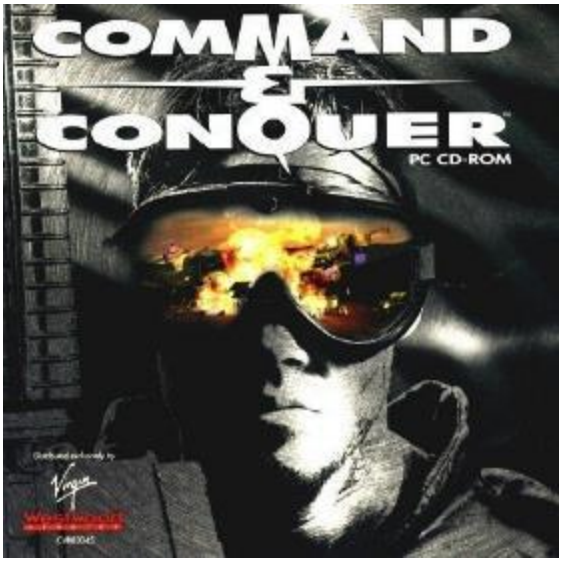


Рисунок 2.9 Обкладинка гри Command & Conquer

Звернемося до слів Сперрі, «Завершуючи розробку Dune II, ми вже обмірковували деякі ідеї Command & Conquer. Створюючи Dune II, ми дуже багато чого навчилися, як, втім, це зазвичай відбувається і з будь-якою іншою грою. Наближаючись до кінця проекту, завжди думаєш Ось наступного разу ми обов'язково зробимо це, це і це. Command & Conquer стала результатом втілення наших побажань». Зараз, коли C&C вже стала синонімом ігор RTS, буде цікаво дізнатися, що вона могла називатися і якось інакше. Як говорить Сперрі, "З комерційним успіхом Dune II, настав час створити просунутий варіант RTS, яка не буде скута ліцензією як Dune II, так і з'явилася Command & Conquer. Я фанатично відстоював назву C&C, тому що мені здавалося, що такий варіант найкраще описує дії гравця Хоча під час розробки відділ маркетингу та інші особи боялися, що така назва нагадуватиме сексуальні садо-мазохістські

захоплення. Зараз складно описувати ту лихоманку, яку викликала C&C у 1995 році. Деякі гравці пам'ятають SVGA версію гри під Windows 95, але вона була "золотим" варіантом, що вийшов на рік пізніше. Оригінальна C&C була випущена під DOS. Хоча в неї не було такого SVGA блиску, як у Warcraft, вона все ж таки мала дуже пристойну графіку і захоплюючий ігровий процес. З цих причин гру можна вважати справжнім сином Dune II, не забуваючи і футуристичну зброю в арсеналі кожного гравця.

У C&C відбувається битва між Глобальною Оборонною Ініціативою (GDI) та Братством Нодів (Brotherhood of Nod). Початковий ролик просто фантастичний, а відеозаставки між битвами нагадують класичну битву між добром та злом. На відміну від Warcraft, кожна сторона має підрозділи з різними можливостями. GDI більше розраховує на вогневу міць, тоді як Братство Нодов ставить швидкість. До недоліків можна віднести те що у грі немає морських одиниць, але це компенсується різноманіттям підрозділів яких виявляється достатньою для реалізації різних стратегій.

Рисунок 2.10 База Нодів

Рисунок 2.11 База GDI

Успішний продаж C&C перевершив усі очікування, настільки, що Westwood навіть розробила онлайн додаток, Sole Survivor, в якій видається один підрозділ, і ви використовуєте його для битви з іншими гравцями. Ви можете знайти модернізації, типу здоров'я, невидимості, броню тощо. Також пропонувалися і додаткові режими гри (типу захоплення прапора чи футболу), які можуть комусь здатися цікавішими. Але основна ідея – гра одним підрозділом, так і не увійшла до моди. Врешті-решт Westwood випустила кілька успішних розширень C&C до яких входить і легендарне продовження: Red Alert.

Отже аналіз існуючих стратегій показує, що існуючі продукти не позбавлені недоліків, бо немає нічого ідеально, а також що вони не повністю задовільняют потреби сучасних гравців, тому розробка власної гри-стратегії з урахуванням всіх недоліків допущенних у минулих іграх, а також усіх їх переваг, є повністю доцільним та необхідним для розвитку жанру комп’ютерних стратегій у реальному часі. Створена гра буде надавати нові можливість гравцю, завдяки новому усучасненому геймплею гравець зможе отримати новий унікальний досвід, та провести свій вільний час за цікавим заняттям.

1.3 Використані програмні засоби

Для написання гри-стратегії було використано середовище програмування Microsoft Visual Studio 2019, мова програмування C#, а також ігровий рушій Unity версії 2021.3.1f1.

Microsoft Visual Studio — продукт фірми Майкрософт, який включає інтегроване середовище розробки програмного забезпечення та ряд інших інструментальних засобів. Цей продукт дозволяє розробляти як консольні програми, так і програми з графічним інтерфейсом, в тому числі з підтримкою технології Windows Forms, а також веб-сайти, веб-застосунки, веб-служби як в рідному, так і в керованому кодах для всіх платформ, що підтримуються Microsoft Windows, Windows Mobile, Windows Phone, Windows CE, .NET Framework, .NET Compact Framework та Microsoft Silverlight, яке працює під Microsoft Windows і підтримує розробку застосунків для операційних систем Microsoft Windows x86 та x64, Mac OS x86, Apple iOS та Android. Середовище Visual Studio дозволяє розробляти додатки, використовуючи різні мови програмування: Visual C#, Visual Basic, Visual F#, Visual C++, Python і т.д.

C# (вимовляється Сі-шарп) — об'єктно-орієнтована мова програмування з безпечною системою типізації для платформи .NET. Розроблена Андерсом Гейлсбергом, Скотом Вілтамутом та Пітером Гольде під егідою Microsoft Research (при фірмі Microsoft).

Синтаксис C# близький до C++ і Java. Мова має строгу статичну типізацію, підтримує поліморфізм, перевантаження операторів, вказівники на функції-члени класів, атрибути, події, властивості, винятки, коментарі у форматі XML. Перейнявши багато що від своїх попередників — мов C++, Delphi, Модула і Smalltalk — C#, спираючись на практику їхнього використання, виключає деякі моделі, що зарекомендували себе як проблематичні при розробці програмних систем, наприклад множинне спадкування класів (на відміну від C++).

C# є дуже близьким родичем мови програмування Java. Мова Java була створена компанією Sun Microsystems, коли глобальний розвиток інтернету

поставив задачу розосереджених обчислень. Взявши за основу популярну мову C++, Java виключила з неї потенційно небезпечні речі (типу вказівників без контролю виходу за межі). Для розосереджених обчислень була створена концепція віртуальної машини та машинно-незалежного байт-коду, свого роду посередника між вихідним текстом програм і апаратними інструкціями комп'ютера чи іншого інтелектуального пристрою. Java набула чималої популярності, і була ліцензована також і компанією Microsoft. Але з плином часу Sun почала винуватити Microsoft, що та при створенні свого клону Java робить її сумісною виключно з платформою Windows, чим суперечить самій концепції машинно-незалежного середовища виконання і порушує ліцензійну угоду. Microsoft відмовилася піти назустріч вимогам Sun, і тому з'ясування стосунків набуло статусу судового процесу. Суд визнав позицію Sun справедливою, і зобов'язав Microsoft відмовитися від позаліцензійного використання Java.

У цій ситуації в Microsoft вирішили, користуючись своєю вагою на ринку, створити свій власний аналог Java — мову, в якій корпорація стане повновладним господарем. Ця новостворена мова отримала назву C#. Вона успадкувала від Java концепції віртуальної машини (середовище .NET), байт-коду (MSIL) і більшої безпеки вихідного коду програм, плюс врахувала досвід використання програм на Java.

Нововведенням C# стала можливість легшої взаємодії, порівняно з мовами-попередниками, з кодом програм, написаних на інших мовах, що є важливим при створенні великих проектів. Якщо програми на різних мовах виконуються на платформі .NET, .NET бере на себе клопіт щодо сумісності програм (тобто типів даних, за кінцевим рахунком).

Unity — багатоплатформовий інструмент для розробки дво- та тривимірних додатків та ігор, що працює на операційних системах Windows і OS X. Створені за допомогою Unity застосування працюють під системами Windows, OS X, Android, Apple iOS, Linux, а також на гральних консолях Wii, PlayStation 3 і XBox 360.

Редактор Unity має простий Drag & Drop інтерфейс, який легко налаштовувати, що складається з різних вікон, завдяки чому можна проводити

налагодження гри прямо в редакторі. Рушій підтримує три сценарних мови: C #, JavaScript (модифікація). Проект в Unity ділиться на сцени (рівні) — окремі файли, що містять свої ігрові світи зі своїм набором об'єктів, сценаріїв, і налаштувань. Сцени можуть містити в собі як, об'єкти (моделі), так і порожні ігрові об'єкти — тобто ті, які не мають моделі. Об'єкти, в свою чергу містять набори компонентів, з якими і взаємодіють скрипти. Також у них є назва (в Unity допускається наявність двох і більше об'єктів з однаковими назвами), може бути тег (мітка) і шар, на якому він повинен відображатися. Так, у будь-якого предмета на сцені обов'язково присутній компонент Transform — він зберігає в собі координати місця розташування, повороту і розмірів по всіх трьох осях. У об'єктів з видимою геометрією також за замовчуванням присутній компонент Mesh Renderer, що робить модель видимою.

Також Unity підтримує фізику твердих тіл і тканини, фізику типу Ragdoll (ганчіркова лялька). У редакторі є система успадкування об'єктів; дочірні об'єкти будуть повторювати всі зміни позиції, повороту і масштабу батьківського об'єкта. Скрипти в редакторі прикріплюються до об'єктів у вигляді окремих компонентів.

При імпорті текстури в рушій можна згенерувати alpha-канал, mip-рівні, normal-map, light-map, карту відображень, проте безпосередньо на модель текстуру прикріпити не можна — буде створено матеріал, з яким буде призначений шейдер, і потім матеріал прикріпиться до моделі. Редактор Unity підтримує написання і редагування шейдерів. Крім того він містить компонент для створення анімації, яку також можна створити попередньо в 3D-редакторі та імпортувати разом з моделлю, а потім розбити на файли.

Графічний рушій використовує DirectX (Windows), OpenGL (Mac, Windows, Linux), OpenGL ES (Android, iOS), та спеціальне власне API для Wii. Також підтримуються bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), динамічні тіні з використанням shadow maps, render-to-texture та повноекранні ефекти post-processing.

Unity підтримує файли 3ds Max, Maya, Softimage, Blender, modo, ZBrush, Cinema 4D, Cheetah3D, Adobe Photoshop, Adobe Fireworks та Allegorithmic Sub-

stance. В ігровий проект Unity можна імпортувати об'єкти цих програм та робити налаштовування за допомогою графічного інтерфейсу.

Для написання шейдерів використовується ShaderLab, що підтримує шейдерні програми написані на GLSL або Cg. Шейдер може включати декілька варіантів реалізації, що дозволяє Unity визначати найкращий варіант для конкретної відеокарти.Unity також має вбудовану підтримку фізичного рушія Nvidia PhysX (колишнього Ageia), підтримку симуляції одягу в системі реального часу на довільній та прив'язаній полігональній сітці (починаючи з Unity 3.0), підтримку системи ray casts та шарів зіткнення.

Скриптова система ігрового рушія зроблена на Mono — вільний відкритий проект з реалізації .NET Framework. Програмісти можуть використовувати UnityScript (власна скриптова мова, подібна до JavaScript та ECMAScript), C# або Boo (мова програмування, подібна до Python).Починаючи з версії 3.0, до Unity входить перероблена версія MonoDevelop для зневадження скриптів. Із виходом версії 5.2 передбачається вбудована можливість редагувати скрипти у середовищі Visual Studio.

В Unity включено систему контролю версій Unity Asset Server для ігрових об'єктів та скриптів. Система використовує PostgreSQL, роботу зі звуком, побудовану на основі бібліотеки FMOD (з можливістю програвати Ogg Vorbis аудіофайли), відеопрогравач із кодеком Theora, рушій для побудови ландшафтів рослинності, вбудовану систему карт освітлення (Beast), мережу для мультиплеєру (RakNet) та вбудовані навігаційні меші для пошуку шляху.

1.4 Постановка задачі з розробки розважального програмного забезпечення гри-стратегії

Аналіз вже існуючих стратегій показує, що існуючі програмні продукти не повністю задовільняють потреби сучасних гравців, тому розробка власної гри-стратегії з урахуванням всіх недоліків допущенних у минулих іграх, а також усіх їх переваг, є повністю доцільним та необхідним для розвитку жанру

комп’ютерних стратегій у реальному часі. Створена гра буде надавати нові можливість гравцю, завдяки новому усучасненому геймплею гравець зможе отримати новий унікальний досвід, та провести свій вільний час за цікавим заняттям.

Взаємодія користувача(гравця) з грою буде відбуватись через візуальний зручний інтерфейс, що буде відібражати для нього всі його ресурси, надавати можливість заходити в розділ Магазину з якого він буде у разі наявності потрібних ресурсів зможе побудувати потрібну йому споруду у вибранному місці. Розроблена гра-стратегія повинна містити наступні функції та геймплейні особливості:

1) Будування основної бази у будь-якому містці

2) Можливість накопичувати та добувати різноманітні ресурси

3) Спроможність розвивати вже побудовані споруди

4) Демонтування вже побудованних раніше споруд

5) Здатність гравця будувати захисні споруди

Для успішної і коректної роботи програмного забезпечення необхідний персональний комп'ютер з наступною сукупністю апаратних і програмних засобів:

Мінімальні системні вимоги

– процесор Intel Pentium G4400;
– 4 GB оперативної пам'яті;
– Відеокарта рівня Nvidia Gtx 650 або краще;
– 120 GB HDD;
– ОС Microsoft Windows 7 або вище

Рекомендовані системні вимоги

– процесор Intel Core I5 6500, AMD Ryzen 1600;
– 8 GB оперативної пам'яті;

– Відеокарта рівня Nvidia Gtx 960 або краще;
– 240 GB SSD;
– ОС Microsoft Windows 7 або вище

2 РОЗРОБКА РОЗВАЖАЛЬНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ У ВИГЛЯДІ ГРИ-СТРАТЕГІЇ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ UNITY

2.1 Опис програми та її алгоритмів

Будь-який проєкт в ігровому рушії Unity складається зі сцен, на яких розташовані ігрові об'єкти(GameObject) з прикріпленими до них певними компонентами, кожен об’єкт має обов’язковий компонент Transform, який відповідає за розташування об'єкта на сцені, його координати, розмір об’єкту у вигляді його скейлу, а також поворот. До будь якого ігрового об'єкту може бути додані додаткові компоненти, це можут бути як заздалегідь вбудовані компоненти що надаються рушієм Unity для більш зручної розробки, чи якісь сторонні або навіть самописні компоненти.

На високому рівні компоненти в Unity — це просто набір скриптів C#, які можна додати до GameObject, вони можуть бути розроблені та реалізовані багатьма способами. Традиційно, успадкування використовується в об'єктно-орієнтованому програмуванні для повторного використання коду без його переписування, однак в Unity краще використовувати підхід на основі композиції, створення ігрових функцій за допомогою підкомпонентів (а не успадкування від об’єкта) зазвичай є ефективнішим і легшим у створенні та подальшому обслуговуванні, саме через це в Unity використовується компонентний дизайн. Всі призначені для користувача класи повинні успадковуватися від класу MonoBehaviour, або його похідних. Також завдяки наявності такого дизайну можливе більш точне та зручне налаштуванні певних об’єктів під потреби відповідно до геймдизайну, а також такий дизайн надає можливості створювати для геймдизайнерів інструменти що надають їм можливості змінювати та

налаштовувати об’єкти як їм потрібно без залучення до цього процессу безпосередньо програмістів.

Основним завданням моєї роботи було створення гри-стратегія з урахуванням недоліків минулих проектів, а також створення максимально комфортних умов для подальшого розширення проекту. Тож розглянемо більш детально функції системи, що представлені у нотації мови UML у вигляді діаграм прецедентів на рисунках 2.1 та 2.2.

Перша діаграма містить набір функцій актора Користувач. Користувач – це людина, яка користується програмою за допомогою інтерфейсу користувача.

1. Заходити в гру за допомогою натискання мишею або вибору за допомогою клавіатури кнопки головного меню Play;
2. Ставити гру на паузу, знаходячись у самій грі за допомогою натискання кнопки Escape на клавіатурі, або за допомогою натискання на пункт меню що символізує паузу на ігровому інтерфейсі;
3. Знімати паузу, знаходячись у меню паузи за допомогою натискання мишею або вибору за допомогою клавіатури кнопки меню паузи Resume або за допомогою натискання кнопки Escape на клавіатурі;
4. Виходити в головне меню, знаходячись у меню паузи за допомогою натискання мишею або вибору з допомогою клавіатури кнопки меню паузи Exit to main menu;
5. Виходити з програми, знаходячись у головному меню за допомогою натискання мишею або вибору за допомогою клавіатури кнопки меню Quit або за допомогою натискання кнопки Escape на клавіатурі.
6. Будувати основну базу гравця вибранному місці
7. Будувати наступні типи споруд у вибранному гравцем місці:
    - Захисні споруди, як атакуючого (турелі, ловушки), так і захистного типу (стіни)
    - Споруди для добування корисних ресурсів, наприклад Шахти, чи Зернові поля

- Декоративні спорди, що служать для декорації(наприклад вуличні ліхтарі, декоративні дерева)

8. Руйнувати вже побудовані споруди, а також заважаючі елементи декоарції

9. Проводити модернізацію вже побудованних споруд за певну кількість ресурсів(кількість залежить вид типу будівля, а також її рівня)

10. Ремонтувати пошкоджені споруди

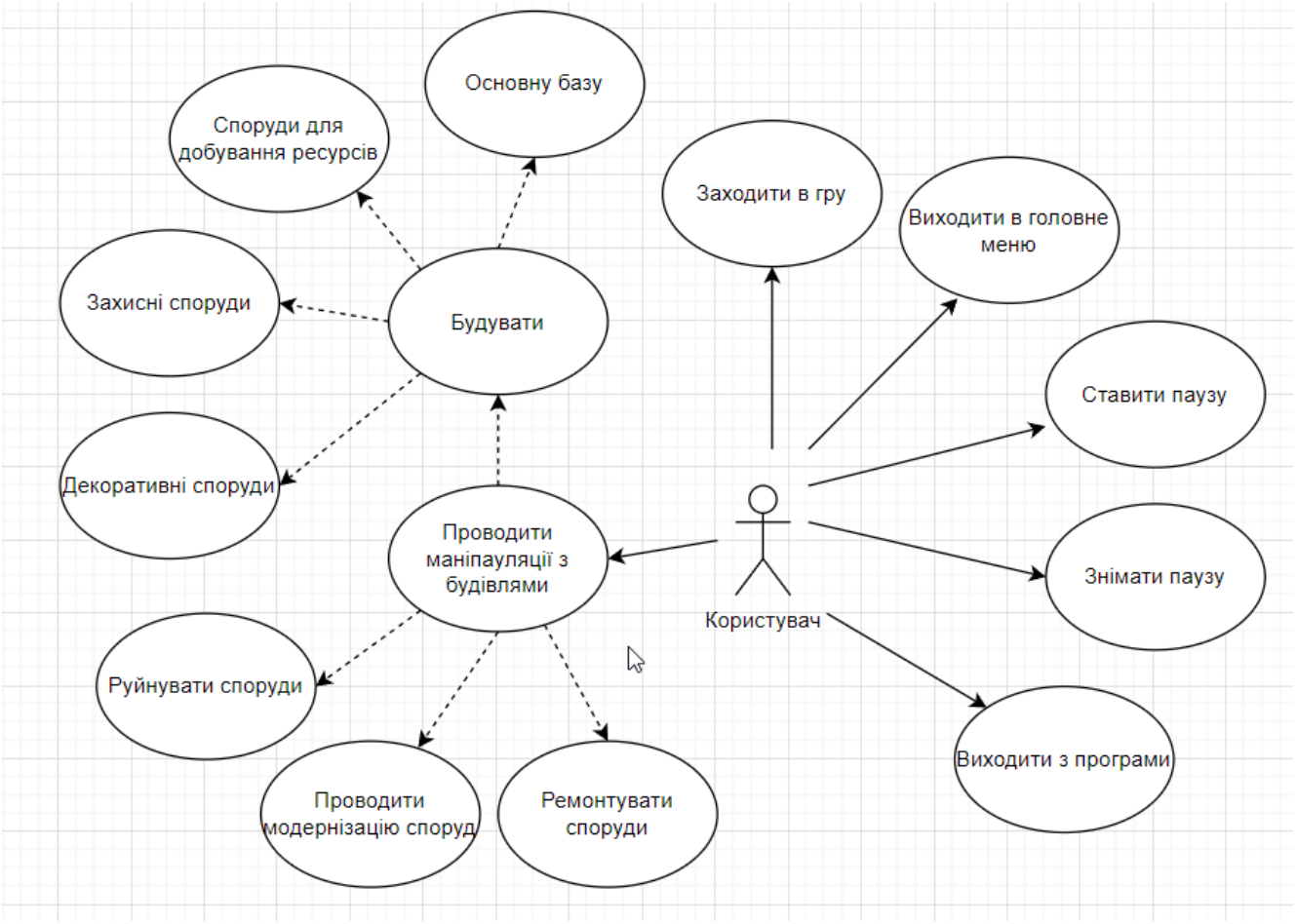


Рисунок 2.1 - UML діаграма прецедентів актора Користувач

Друга діаграма зображує набір функцій, які виконує ядро Unity. Ядро Unity – це внутрішній процес, який відповідає за малювання кадрів та оброблює дані, що надходять від Користувача.

Ядро Unity виконує наступні функції:

- Перемальовування кадрів сцени відповідно до змін що відбулись;

- Керування ігровими об`єктами;
- Створення нових об’єктів;
- Завантаження ігрової сцени;

Описану вище діаграму представлено на рисунку 2.7.

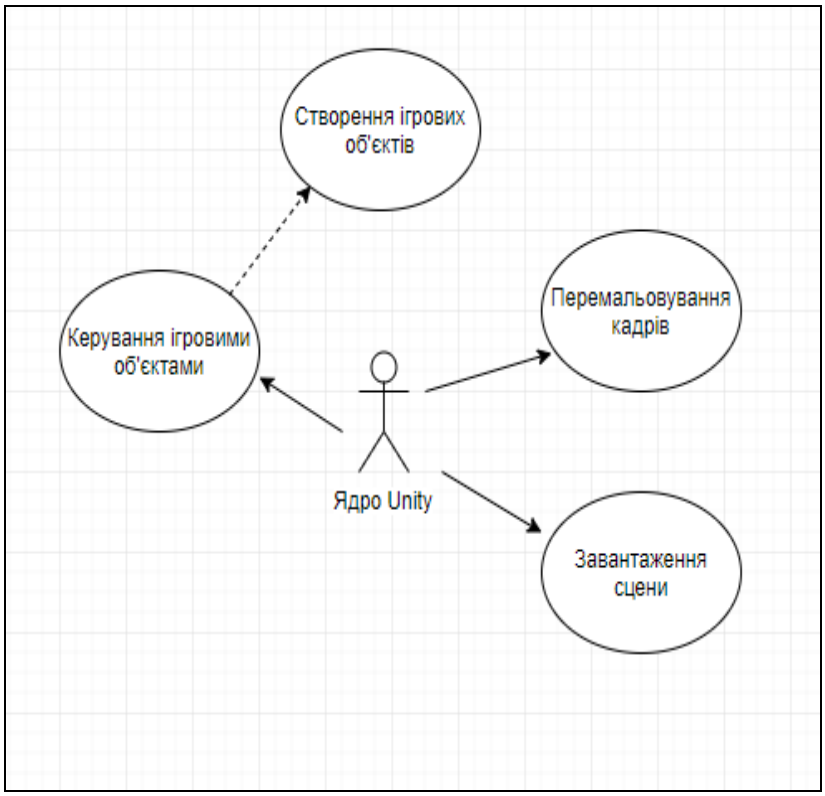


Рисунок 2.2 – Діаграма прецедентів актора Ядро Unity

2.2 Опис ієрархії класів

Діаграми класів використовуються для представлення програмних моделей що можуть відображати різні відносини між сутностями у предметній області, а також їх внутрішню структуру, зв’язки та їх типи. Діаграма зазвичай не містить інформації про виконання програмних операцій, вона складається з багатьох елементів, які разом відображають декларативні знання предметної області.

Для реалізації логіки програми було розроблено набір класів, діаграму представлено на рисунку 2.3.

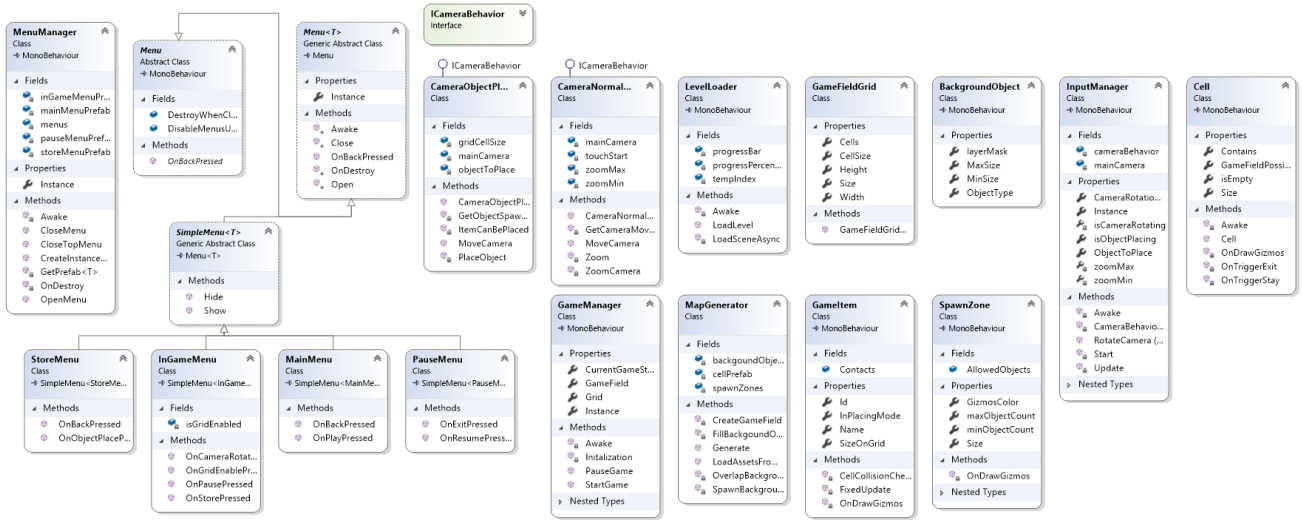


Рисунок 2.3 – Діаграма класів

Для більш детального сприйняття ієрархії класів потрібно звернутися до специфікації класів наведенної нижче. Загалом представлені класи можна поділити на декілька груп, однією з них є класи що відподівають за користувацький інтерфейс, до них належить:

1) Клас Menu, представляє собою абстрактний класс меню, що наслідується від MonoBehaviour. Методи та поля класу представлені у таблиці 2.1

Таблиця 2.1 – Описи методів та полів класу Menu

DestroyWhenClosed	Публічне булеве поле, що відповідає за встановлення флагу чи знищувати об’єкт меню коли він закритий чи просто вимикити його
DisableMenusUnderneath	Публічне булеве поле, що відповідає за встановлення флагу чи вимикати всі меню що знаходятся під цим меню по ієрархії
OnBackPressed()	Публічний абстрактний метод що викликається коли нажимається кнопка назад

2) Клас Menu<T> що являє собою абстрактний дженерік клас, від якого наслідуються типи меню. Методи та поля класу представлені у таблиці 2.2

Таблиця 2.2 – Описи методів та полів класу Menu<T>

Instance	Статичне поле дженерік типу що відражає поточний екземпляр класу Menu<T>
Awake()	Захищенний віртуальний метод що відповідає за створення поточного екземпляру класу Menu<T> якщо він відсутній, та за знищення у разі того якщо їх забагато(Реалізація паттерну Singleton для Unity)
OnDestroy()	Захищенний віртуальний метод що виникає при знищенні об’єкта
Open()	Захищенний статичний метод що відповідає за відкриття цього меню
Close()	Захищенний статичний метод що відповідає за закриття цього меню
OnBackPressed()	Публічний метод що викликає закриття меню у разі натискання назад

3) Клас SimpleMenu<T> що являє собою абстрактний дженерік клас, унаслідуванний від класу Menu<T>, від якого наслідуються основні меню. Методи та поля класу представлені у таблиці 2.3

Таблиця 2.3 – Описи методів та полів класу SimpleMenu<T>

Show()	Публічний статичний метод що відповідає за показ меню
Hide()	Публічний статичний метод що відповідає за скривання меню

4) Клас StoreMenu що являє собою клас, унаслідуванний від класу SimpleMenu<T>. Методи та поля класу представлені у таблиці 2.4

Таблиця 2.4 – Описи методів та полів класу StoreMenu

OnObjectPlacePressed (GameObject spawnObject)	Публічний статичний метод що відповідає за показ меню
OnBackPressed()	Публічний метод що викликає закриття меню у разі натискання назад

5) Клас PauseMenu що являє собою клас, унаслідуванний від класу SimpleMenu<T>. Методи та поля класу представлені у таблиці 2.5

Таблиця 2.5 – Описи методів та полів класу PauseMenu

OnResumePressed()	Публічний метод що виникає якщо користувач натиснув кнопку «Продовжити»
OnExitPressed()	Публічний метод що викликає якщо користувач натиснув кнопку «Вийти в головне меню»

6) Клас InGameMenu що являє собою клас, унаслідуванний від класу SimpleMenu<T>. Методи та поля класу представлені у таблиці 2.6

Таблиця 2.6 – Описи методів та полів класу InGameMenu

isGridEnabled	Приватне булеве поле, що відповідає за встановлення флагу чи вімкнута сітка в грі чи ні
OnPausePressed()	Публічний метод що викликає якщо користувач натиснув кнопку паузи
OnGridEnablePressed()	Публічний метод що викликає якщо користувач натиснув кнопку вмикання\вимикання сітки
OnStorePressed()	Публічний метод що викликає якщо користувач натиснув кнопку магазину
OnCameraRotatePressed()	Публічний метод що викликає якщо

	користувач натиснув кнопку зміни ракурсу камери

7) Клас MainMenu що являє собою клас, унаслідуванний від класу SimpleMenu<T>. Методи та поля класу представлені у таблиці 2.7

Таблиця 2.7 – Описи методів та полів класу MainMenu

OnPlayPressed()	Публічний метод що викликає старт гри у разі натискання на кнопку Play
OnBackPressed()	Публічний метод що викликає закриття гри у разі натискання назад

8) Клас MenuManager що являє собою клас менеджеру меню, що займається керуванням усієї системи меню, унаслідуванний від класу MonoBehaviour. Методи та поля класу представлені у таблиці 2.8

Таблиця 2.8 – Описи методів та полів класу

Instance	Статичне поле типу MenuManager що відражає поточний екземпляр класу MenuManager
menus	Приватний стек меню в якому містяться об’єкти типу Menu
mainMenuPrefab	Приватне поле що представляє собою ссилку на префаб типу MainMenu
inGameMenuPrefab	Приватне поле що представляє собою ссилку на префаб типу InGameMenu
pauseMenuPrefab	Приватне поле що представляє собою ссилку на префаб типу PauseMenu
storeMenuPrefab	Приватне поле що представляє собою

	ссилку на префаб типу StoreMenu
Awake()	Приватний метод що відповідає за створення поточного екземпляру класу MenuManager якщо він відсутній, та за знищення у разі того якщо їх забагато(Реалізація паттерну Singleton для Unity)
OnDestroy()	Захищенний віртуальний метод що виникає при знищенні об’єкта
CreateInstance<T>()	Публічний метод що створює екземпляр об’єкту заданого типу меню
OpenMenu(Menu instance)	Публічний метод що відкриває переданий екземпляр меню
GetPrefab<T>()	Приватний метод що повертає префаб об’єкту заданного типу меню
CloseMenu(Menu menu)	Публічний метод що закриває задане меню
CloseTopMenu()	Публічний метод що закриває верхне меню

До наступної групи відносятся класи керування камерою:

1) Клас CameraNormalMode що являє собою клас що описує поведінку камери в стандартному моді, клас реалізує інтерфейс ICameraBehavior. Методи та поля класу представлені у таблиці 2.9

Таблиця 2.9 – Описи методів та полів класу CameraNormalMode

mainCamera	Приватне поле, що містить ссилку на об’єкт типу Camera
touchStart	Приватне поле типу Vector3, що містить в собі точку початкового натискання при переміщенні камери

zoomMin	Приватне поле типу float, що зберігає мінімальне значенная Зуму
zoomMax	Приватне поле типу float, що зберігає максимальне значенная Зуму
CameraNormalMode (Camera camera, float zoomMin, float zoomMax)	Публічний метод конструктор класу, в якому відбувается ініціалізація класу CameraNormalMode, приймає три змінні
MoveCamera()	Публічний метод, що відповідає за зміну позиції камери
ZoomCamera()	Приватний метод, що відповідає за зум камери
Zoom(float increment)	Приватний метод, що відповідає за зум камери, приймає змінну типу float, що характеризує наскільки потрібно приблизити чи віддалити камеру.
GetCameraMovePosition()	Приватний метод, що відповідає за повернення ветору руху камери

2) Клас CameraObjectPlacingMode що являє собою клас що описує поведінку камери при розташуванні об’єкта на полі, клас реалізує інтерфейс ICameraBehavior. Методи та поля класу представлені у таблиці 2.10

Таблиця 2.10 – Описи методів та полів класу CameraObjectPlacingMode

mainCamera	Приватне поле, що містить ссилку на об’єкт типу Camera
touchStart	Приватне поле типу Vector3, що містить в собі точку початкового натискання при переміщенні камери
zoomMin	Приватне поле типу float, що зберігає мінімальне значенная Зуму

zoomMax	Приватне поле типу float, що зберігає максимальне значенная Зуму
CameraNormalMode (Camera camera, float zoomMin, float zoomMax)	Публічний метод конструктор класу, в якому відбувается ініціалізація класу CameraNormalMode, приймає три змінні
MoveCamera()	Публічний метод, що відповідає за зміну позиції камери
ZoomCamera()	Приватний метод, що відповідає за зум камери
Zoom(float increment)	Приватний метод, що відповідає за зум камери, приймає змінну типу float, що характеризує наскільки потрібно приблизити чи віддалити камеру.
GetCameraMovePosition()	Приватний метод, що відповідає за повернення ветору руху камери

До наступної категорії відносятся класи що відповідають безпосередньо за геймпелей, та керують ігровим процесом:

1) Клас GameManager що являє собою клас що відповідає за основний поток гри, містить те керує основними об’єктами, унаслідуванний від класу MonoBehaviour. Методи та поля класу представлені у таблиці 2.11

Таблиця 2.11 – Описи методів та полів класу GameManager

CurrentGameState	Публічна властивість типу GameStates, що зберігає поточний стан гри
Grid	Публічна властивість типу GameObject, що містить ссилку на сітку
GameField	Публічна властивість типу GameFieldGrid, що містить посилання на ігрове поле

Instance	Публічне статичне поле типу GameManager що відражає поточний екземпляр класу GameManager
Awake()	Приватний метод що відповідає за створення поточного екземпляру класу GameManager якщо він відсутній, та за знищення у разі того якщо їх забагато(Реалізація паттерну Singleton для Unity)
Initalization()	Приватний метод що відповідає за створення ініціалізацію класу GameManager.
PauseGame()	Публічний метод що відповідає за постановку гри на паузу.
StartGame()	Публічний метод що відповідає за запуск гри.

2) Клас InputManager що являє собою клас що відповідає за отримання та обробку дій користувача в грі, унаслідуванний від класу MonoBehaviour.. Методи та поля класу представлені у таблиці 2.12

Таблиця 2.12 – Описи методів та полів класу InputManager

mainCamera	Приватне поле типу Camera, що містить ссилку на основну камеру
ObjectToPlace	Публічна властивість типу GameObject, що містить ссилку на об’єкт для розташування
zoomMin	Приватне поле типу float, що зберігає мінімальне значенная Зуму
zoomMax	Приватне поле типу float, що зберігає максимальне значенная Зуму
CameraRotationSpeed	Публічна властивість типу float, що

	зберігає швідкість поворота камери
isObjectPlacing	Публічна властивість типу bool, що містить флаг чи існує об’єкт для розташування(ObjectToPlace)
isCameraRotating	Приватна властивість типу bool, що містить флаг повороту камери
Instance	Публічне статичне поле типу InputManager що відражає поточний екземпляр класу InputManager
Awake()	Приватний метод що відповідає за створення поточного екземпляру класу InputManager якщо він відсутній, та за знищення у разі того якщо їх забагато(Реалізація паттерну Singleton для Unity)
Start()	Приватний метод що відповідає за ініціалізацію
Update()	Приватний метод що відповідає за перевірку на наявність інпуту, відбувається кожний кадр
CameraBehaviorChanger()	Приватний метод що відповідає за зміну поведінки камери
RotateCamera (RotationDirection direction)	Публічний метод що відповідає за поворот камери, приймає в себе змінну типу RotationDirection в якій міститься напрямок в який потрібно повертати камеру
RotateCamera (RotationDirection direction, float inTime)	Приватний метод що відповідає за поворот камери, приймає в себе змінну типу RotationDirection в якій міститься напрямок в який потрібно повертати камеру, а також змінну типу float що відповідає за те скільки часу треба на цю дію

3) Клас MapGenerator що являє собою клас що відповідає за генерацію карти в грі. Методи та поля класу представлені у таблиці 2.13

Таблиця 2.13 – Описи методів та полів класу MapGenerator

spawnZones	Приватний масив полів типу SpawnZone, що зберігає спавн зони
cellPrefab	Приватне поле що представляє собою ссилку на префаб типу GameObject
backgoundObjectsByType	Приватний словник що поле що містить тип об’єкту як ключ, та список типу BackgroundObject як значення.
Generate()	Публічний метод що відповідає за генерацію об’єктів
LoadAssetsFromResources()	Публічний метод що відповідає за завантаження необхідних ресурсів
FillBackgoundObjectByType	Приватний метод що відповідає за заповнення об’єктами
CreateGameField()	Приватний метод що відповідає за генерацію ігрового поля
SpawnBackgroundObjects()	Приватний метод що відповідає за спавн бекграунд об’єктів
OverlapBackgroundSpawnCheck	Приватний булевий метод що відповідає за перевірку існування об’єкта у заданій точці, повертає false якщо об’єкт є у певному радіусі, та true якщо об’єктів немає

4) Клас GameItem що являє собою клас що відповідає за ігрові об’єкти на мапі, унаслідуванний від класу MonoBehaviour. Методи та поля класу представлені у таблиці 2.14

Таблиця 2.14 – Описи методів та полів класу GameItem

Id	Публічна властивість типу int, що зберігає унікальний ідентифікатор об’єкта
Name	Публічна властивість типу string, що зберігає назву об’єкта
Contacts	Публічний список об’єктів типу Cell, що зберігає список ячейок до яких доторкається поточний об’єкт
SizeOnGrid	Публічна властивість типу Vector2Int, що зберігає розмрір об’єкту на ігровій мапі
OnDrawGizmos()	Приватний метод що виникає при перемалюванні індикаторів, потрібен для дебагу
FixedUpdate()	Приватний метод що виникає при кожному тіку фізики в Unity, викликається 60 разів на секунду
CellCollisionCheck()	Приватний метод що відповідає за перевірку колізії об’єкту

5) Клас LevelLoader що являє собою клас що відповідає за завантаження рівнів, унаслідуванний від класу MonoBehaviour. Методи та поля класу представлені у таблиці 2.15

Таблиця 2.15 – Описи методів та полів класу LevelLoader

tempIndex	Приватне поле типу int що представляє собою індекс сцени для завантаження
progressBar	Приватне поле що представляє собою ссилку на префаб типу Slider
progressPercentageText	Приватне поле типу TMP_Text що

	представляє собою ссилку на текстову змінну що відповідає за відсоток завантаження
Awake()	Приватний метод що відповідає за ініціалізацію
LoadLevel(int sceneIndex)	Публічний метод що відповідає за завантаження відповідної сцени по переданому індексу
LoadSceneAsync(int sceneIndex)	Публічний метод що відповідає за асинхронне завантаження відповідної сцени по переданому індексу

6) Клас SpawnZone що являє собою клас що відповідає за налаштування зони спавну об’єктів, унаслідуванний від класу MonoBehaviour. Методи та поля класу представлені у таблиці 2.16

Таблиця 2.16 – Описи методів та полів класу SpawnZone

GizmosColor	Публічна властивість типу Color, що відповідає за колір відображення зони
AllowedObjects	Публічний масив типу ObjectType, що містить дозволені типи об’єктів для спавну
minObjectCount	Публічна властивість типу int, що відповідає за мінімальну кількість об’єктів
maxObjectCount	Публічна властивість типу int, що відповідає за максимальну кількість об’єктів
Size	Публічна властивість типу Vector3, що відповідає за розмір зони
OnDrawGizmos()	Приватний метод що виникає при перемалюванні індикаторів зони

7) Клас BackgroundObject що являє собою клас що відповідає за об’єкт заднього плану, унаслідуванний від класу MonoBehaviour. Методи та поля класу представлені у таблиці 2.17

Таблиця 2.17 – Описи методів та полів класу BackgroundObject

layerMask	Публічна властивість типу LayerMask, що відповідає за маску
ObjectType	Публічна властивість типу ObjectType, що відповідає за тип об’єкту
MinSize	Публічна властивість типу Vector3, що відповідає за мінімальний розмір об’єкту
MaxSize	Публічна властивість типу Vector3, що відповідає за максимальний розмір об’єкту

8) Клас GameFieldGrid що являє собою клас що відповідає за сітку ігрового поля. Методи та поля класу представлені у таблиці 2.18

Таблиця 2.18 – Описи методів та полів класу GameFieldGrid

GameFieldGrid(int size)	Публічний метод конструктору призначений для створення класу, що приймає змінну типу int, що відповідає за розмір сітки що буде створена
GameFieldGrid(int width, int height)	Публічний метод конструктору призначений для створення класу, що приймає дві змінні типу int, що відповідають за розмір сітки що буде створена
Height	Публічна властивість типу int, що відповідає за висоту об’єкту

Width	Публічна властивість типу int, що відповідає за ширину об’єкту
Cells	Публічний двомірний масив типу Cell, що містить сітку
CellSize	Публічна властивість типу float, що повертає розмір ігрової клітинки

9) Клас Cell що являє собою клас що відповідає за клітинку ігрового поля. Методи та поля класу представлені у таблиці 2.19

Таблиця 2.19 – Описи методів та полів класу Cell

Cell(int x, int y)	Публічний метод конструктору призначений для створення класу, що приймає дві змінні типу int, що відповідають за місцерозташування клітинки
Awake()	Приватний метод що відповідає за ініціалізацію
isEmpty	Публічна властивість типу bool, що повертає true у разі якщо в цій клітинці нічого нема, та false якщо тут розташований якийсь об’єкт
GameFieldPossition	Публічна властивість типу Vector2Int, що відповідає за місцерозташування клітинки
Contains	Публічна властивість типу GameItem, що відповідає за збереження об’єкту що знаходится в клітинці
Size	Публічна властивість типу float, що відповідає за фізичний розмір клітинки
OnTriggerStay(Collider	Приватний метод що виникає коли об’єкт

collider)	знаходится всередині коллайдеру
OnTriggerExit(Collider collider)	Приватний метод що виникає коли об’єкт виходить з коллайдеру
OnDrawGizmos()	Приватний метод що виникає при перемалюванні індикаторів

3 ЯКІСТЬ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Тестування класів гри-стратегії

Під час розробки гра проходить декілька стадій, всі вони відображають життєвий цикл розробки гри, що зображений на рисунку 3.1.

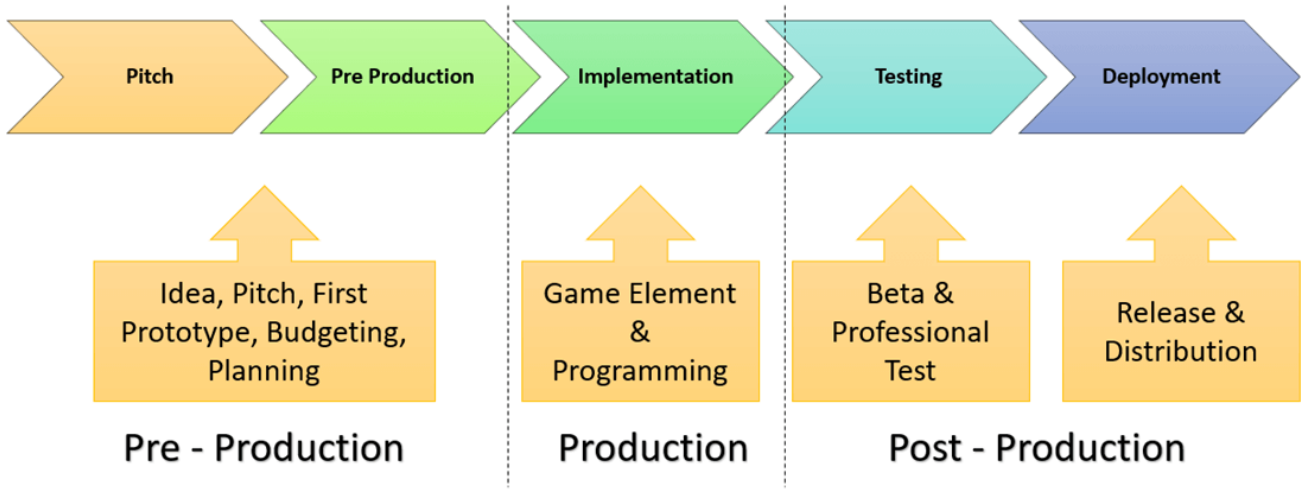


Рисунок 3.1 Життєвий цикл розробки гри

Підготовка: на цьому етапі виконується ідея гри, розкадровка, функції, аналіз вимог та документація. Цей етап включає технічний проект і специфікації функцій, архітектуру гри, накладення кадрів, анімацію, розглядаються наступні пункти:

- Робота над музикою, камерою (налаштування зуму, кінематографічної камери і т.д.), гравцем та його діями.
- Логіка ходу гри, правила та умови для досягнення наступного рівня

- Тригери об’єктів і подій, результати, переміщення та позиціонування гравця, статистика гравця.

- Неінтерактивна послідовність, спеціальні ефекти, титульні екрани, багатокнопкові дії

- Способи керування, фунціонал кнопок, робота над керуванням геймпаду, ефекти удару/вібрації, оформлення юридичної сторони питання, користувацької угоди

Виробництво: на цьому етапі виконується фактичне кодування. Цей етап включає кодування та інтеграцію різних модулів.

Тестування та розгортання: на цьому етапі виконуються функціональні тести, регресійне тестування, альфа, бета тестування. Тестування покриття і потоків, цілісності даних, тестування на основі алгоритму, тестування шляху, а також інкрементне тестування, що проводиться за допомогою інструментів тестування мобільних ігор.

Тестування ігор — це процес тестування відеоігр для контролю якості. Основна мета тестування ігор полягає в тому, щоб виявити та виявити дефекти та помилки у відеогрі та покращити стабільність та продуктивність. Тестування ігор є компонентом розробки ігор, який допомагає переконатися, що відеоігри, які будуть опубліковані, не мають критичних помилок та недоліків. Також тестування э повторюваним процесом, через те що кожна нова збірка може містити нові помилки, і саме тому кожного разу її потрібно ретельно перевіряти

Усе тестування гри має базову структуру, незалежно від розміру гри та часу, необхідного для створення гри. Фахівець із забезпечення якості повинен вивчити правила та вимоги гри, розумійти загальну архітектуру компонентів гри та архітектуру файлів, потіків, а також файлові структури та залежності, пов’язані з грою, з кожним новим прототипом гри потрібно часто переглядати тестові документи, щоб оновити будь-які зміни в специфікаціях, нові тестові приклади

гри та підтримку нової конфігурації. Тестер відеоігор повинен переконатися, щоб жодних нових проблем не було, а ті які були виявлені повинні бути ретельно задукоментовані, щоб надати можливість розробнику швидко виявити та виправити проблемне місце.

На сьогоднішній день існує досить багато різноманітних видів тестувань, нижче наведено популярні методи тестування ігор:

1) **Функціональне тестування -** Тестери функціональності шукають загальні проблеми в грі або в інтерфейсі користувача та графіці, наприклад, проблеми з ігровою механікою, проблеми зі стабільністю та цілісністю ігрових активів. Тестування інтерфейсу користувача забезпечує зручність гри.

2) **Тестування на сумісність -** Перевірка сумісності гри на різних пристроях і на різних конфігураціях обладнання та програмного забезпечення. Приклад: встановлення та гри на всіх підтримуваних консолях/настільних комп'ютерах/мобільних телефонах та перевірка її працездатності

3) **Тестування продуктивності -** Перевіряється загальна продуктивність гри, налаштування продуктивності виконується для оптимізації гри та покращення її швидкодії. Важливі параметри перевіряються під час тестування продуктивності:

- Час відповіді на клієнті та серверах, час завершення транзакції, максимальна продуктивність, довговічність, покриття мережі, витік пам'яті, низький рівень пам'яті, низький рівень заряду акумулятора, час завантаження програм, одночасний доступ до сервера програми, швидкість, пропускна здатність, надійність, масштабованість тощо.

- Споживання батареї та продуктивність графіки - вимірюється споживання батареї мобільної гри, споживання батареї має бути оптимальним протягом довгих годин, а реакція гри має бути задовільною при різних значних навантаженнях на різних пристроях

- Обмеження процесора та пам’яті - лічильники продуктивності використовуються для вимірювання навантаження процессора та споживання пам’яті програми.

- Підключення до мережі - вимірюється час відгуку мобільних ігор у різних типах мереж (Wi-Fi, 2G, 3G, 4G), це дає загальне уявлення про те, наскільки добре гра буде працювати в ненадійних мережах, також перевірка зв’язку між мобільними пристроями, центрами обробки даних, тут більше приділяюить увагу часам пік, нестабільним з’єднанням, дублюванню даних, втраті пакетів та фрагментація даних.

4) Тестування на відповідність - відповідність вимогам ринку (наприклад, правилам Apple App Store, або Google Play Market), відповідність корпоративній політиці (наприклад перевірка на заборонений вміст). Відповідність також може стосуватися регуляторних органів, таких як PEGI та ESRB тому що кожна гра націлена на певний рейтинг, якщо є несприятливий вміст, який невідповідні для бажаного рейтингу, тоді всі ці місця повинні ідентифікуютися та про них необхідно повідомити для виправлення, бо навіть одне порушення під час подання на схвалення ліцензії може призвести до відхилення гри, що спричинить додаткові витрати на подальше тестування та повторну подачу.

5) Тестування локалізації - Тестування локалізації стає надзвичайно важливим, коли гра орієнтована на глобальні ринки, назви ігор, вміст і тексти потрібно перекладати та тестувати за допомогою

пристроїв кількома мовами, такі типи тестів зазвичай виконуються швидко за допомогою хмарного доступу до пристроїв і автоматизації тестування.

6) **Тестування на замочування -** Це тестування автоматизації гри передбачає відтворення гри протягом тривалого часу в самих різноманітних режимах роботи, наприклад, робота коли персонаж просто стоїть якийсь проміжок часу, перевірка гри на паузі або на титульному екрані. Завдяки цьому тестуванню можна виявити досить критичні недоліки, такі як витік пам’яті або помилки округлення.

7) **Тестування на відновлення -** У програмному забезпеченні тестування відновлення перевіряє наскільки добре програму можна відновити після збоїв обладнання та інших збоїв. Програму спеціально виводять з ладу, а потім спострегіють, як вона відновлюється після умов збою. Цей вид тестування допомагає протестувати програму на рахунок втрати цілісності данних при різноманітних збоях.

8) **Перевірка безпеки –** робляться такі перевірки для того, щоб перевірити наскільки програмне забезпечення захищене від різних зовнішніх загроз. Сюди входять перевірки на захист даних від зовнішніх загроз, можливість неконтрольованих обмежень доступу до системи, порушення даних, помилок операційної системи, систем зв'язку та перевірка алгоритмів шифрування.

Під час розробки гри-стратегії для відлагодження роботи та пошуку багів були виконані функціональні тестування, невелике тестування на сумісність(за рахунок тестування на ноутбуці, а також емуляторі), а також тестування продуктивності для пошуку та виправлення критичних недоліків. Тестування накшталт тестування на відповідність, безпекове тестування, тестування локалізації виконані не були через те що представлена і розробела програма є технічним прототипом для демонстрації основних можливостей розробленої системи. У подальшому розвитку під час фінальних стадій розробки зазначених

вище ці тестування будуть виконані у разі необхідності. Код розробленої програми міститься у Додатку А

3.2 Демонстрація програми та перспективи розвитку проєкту

На попередньому етапі розробки на підставі технічного завдання, було створено повністю робочу програму гру-стратегію для десктопних комп’ютерів. Програмне забезпечення було розроблене з урахуванням новітніх технологій, які значно зменшують навантаження на сервер, та покращують враження користувача від користування програмою. Програма повністю відповідає вимогам до функціональних характеристик, вимогам до організації вхідних та вихідних даних, вимогам до надійності ПЗ, та інших вимог, зазначених в технічному проекті.

Гра-сратегія в режимі реального часу може бути завантажена з сайту дистриб’ютора, магазину ігор(такому як Steam, Epic Games Store чи подібних) або куплена безпосередньо у якості фізичної копії у магазина на диску, чи іншому носії. Розмір програми не первищує 200 мегабайт . Для початку роботи з програмою її необхідно встановити за допомогою інсталера. Для коректної роботи програми комп’ютер користувача повинен відповідати мінімальним рекомендованим вимогам, а також на комп’ютері повинні бути встановлені всі потрібні драйвера для роботи з апаратною частиною, та набір загальних бібліотек.

Нижче на рисунках 3.2-3.8 представлений основний інтерфейс та можливості гри.

Рисунок 3.2 Головне меню гри

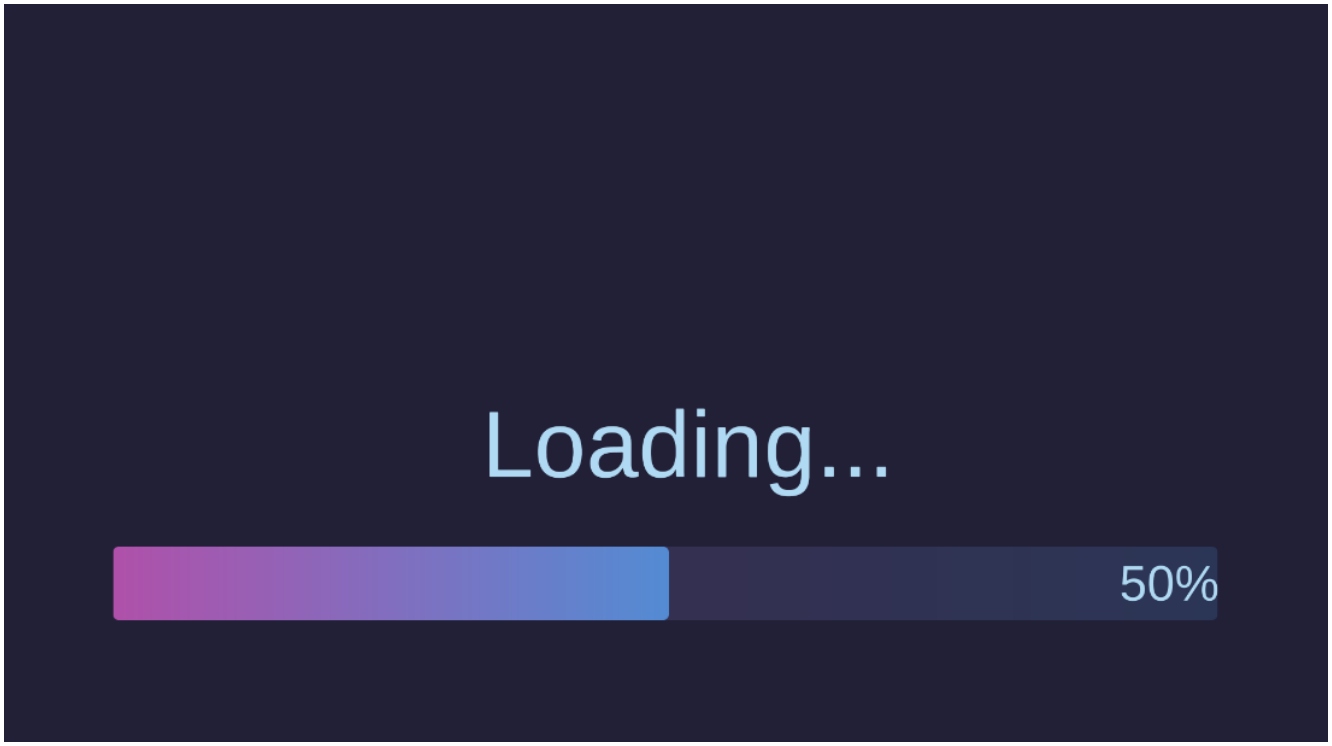


Рисунок 3.3 Єкран завантаженя гри

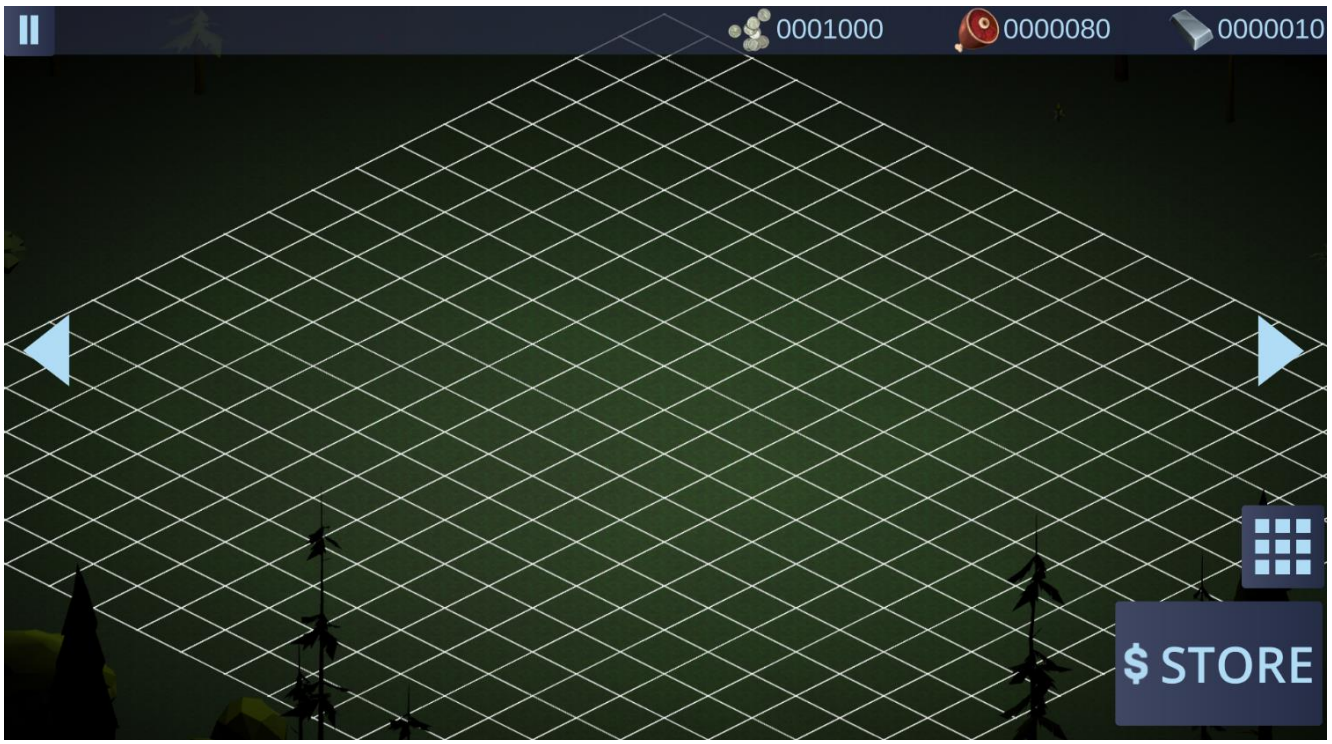


Рисунок 3.4 Головний єкран Гри

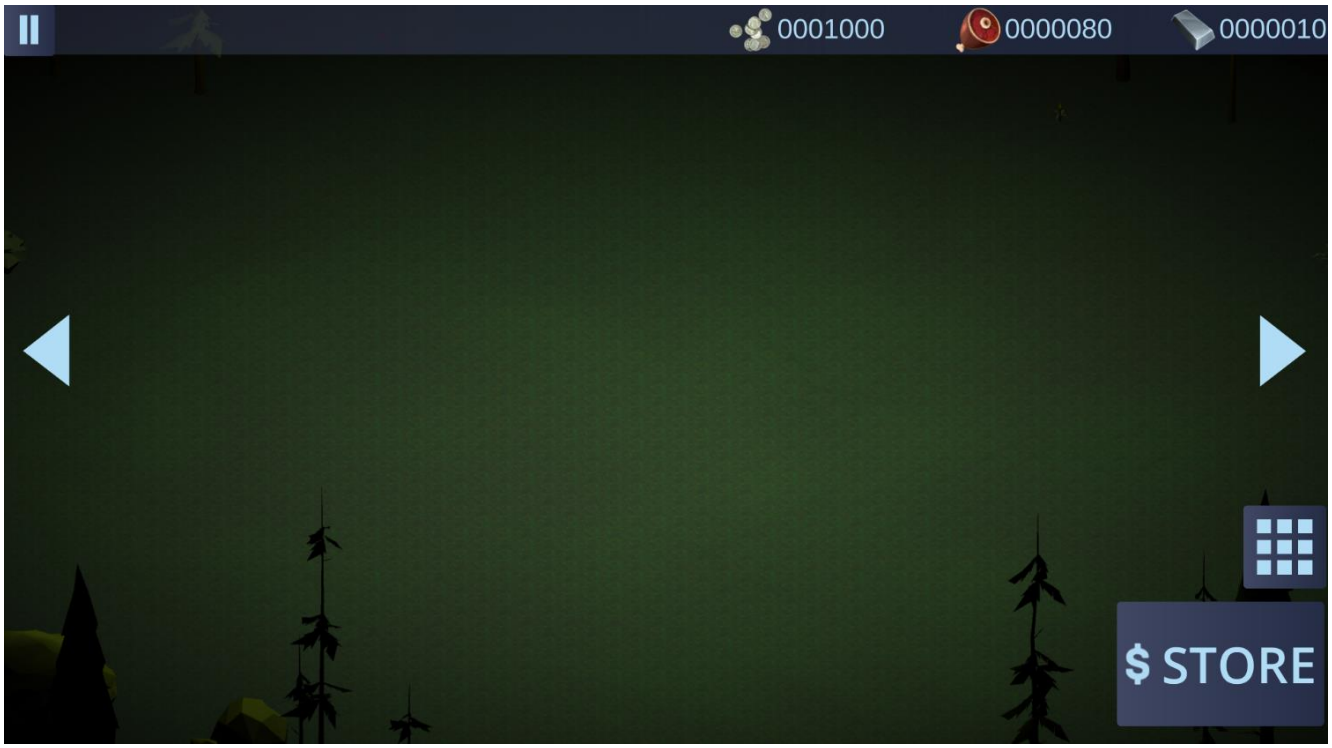


Рисунок 3.5 Вимкненна сітка

Рисунок 3.6 Меню магазину

Рисунок 3.7 Побудована ферма

Рисунок 3.8 Змінений ракурс камери

ВИСНОВКИ

В ході виконання дипломної роботи проведено аналіз предметної галузі розробки ігор-стратегій. Ключові проблеми, які можна виділити, пов’язані з відстуністю ідей у жанрі та його частковому застою. Це призводить до втрачення цільової аудиторії. Також було проведено аналіз засобів розробки розважального програмного забезпечення, в результаті якого у якості мови програмування було обрано C#, що забезпечує реалізовано повну підтримку об’єктно-орієнтованого програмування, має потужні вбудовані функції для обробки текстової інформації, а також широку підтримку та розповсюдження, а також у якості ігрового рушія було обрано Unity, що забезпечує прискорення розробки завдяки зручному інтерфейсу, вбудованним функціям та налаштуванням, а також можливістю збирати готовий проект під різні платформи.

В результаті виконання дипломної роботи було досягнуто поставлених цілей і виконано розробку розважального програмного забезпечення у вигляді гри-стратегі, яке може бути використано для розважальних цілей під час супроводу дозвілля. Програмне забезпечення реалізоване у вигляді програми та задовольняє вимогам, поставленим в технічному завданні і виконує всі основні функції.

Програмне забезпечення може працювати під управлінням операційної системи Windows 7 та вище, та потребує наявність комп’ютера що відповідає як мінімум мінімальним системним вимогам. Тестування та випробування даного програмного продукту показало, що програмне забезпечення успішно справляється з поставленими перед ним задачами, та не має критичних недоліків та багів. Використання програмного забезпечення через використання типового для граця інтерфейса не потребує додаткового навчання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Джозеф А. C# 6.0 in a Nutshell: The Definitive Reference / А. Джозеф, А. Бен., 2018., – ст. 354.
2. Герберт Ш. C# 4.0 The Complete Reference / Шилдт Герберт., 2010., – ст. 114.
3. Tristem B. Unity Game Development in 24 Hours / B. Tristem, M. Geig., 2013.
4. Lukosek G. Learning C# by Developing Games with Unity 5.x / Greg Lukosek., 2016, – ст. 46.
5. McGuire M. Creating Games: Mechanics, Content and Technology / M. McGuire, O. Chadwicke Jenkins., 2008, – ст. 58.
6. Методичні вказівки до виконання кваліфікаційної роботи за напрямом підготовки 6.050103 – «Програмна інженерія» / С.Б. Приходько, С.В. Суслов, Л.О. Латанська, І.В. Устенко, Т.В. Пономаренко. – Миколаїв: НУК, 2012, – ст. 31.
7. How Unity3D Became a Game-Development Beast [Електронний ресурс]: [Вебсайт]. – Режим доступу: https://insights.dice.com/2013/06/03/how-unity3d-become-agame-development-beast/ (дата звернення 15.04.2022)
8. The History and Evolution of the Video Games Market [Електронний ресурс]: [Вебсайт]. – Режим доступу: https://www.businessinsider.com/the-history-and-evolutionof-the-video-games-market-2017-1 (дата звернення 15.04.2022)
9. Armstrong, Deborah J. The Quarks of Object-Oriented Development. February 2006. Communications of the ACM 49, – ст. 128.
10. John Skit. C# in Depth, Manning Publications; 4 edition March 23, 2019, – ст. 604.
11. Кристиан Нейгел , Professional C# 5.0 and .NET 4.5, 2013, – ст.1440

ДОДАТКИ

Додаток А

Код програми

Menu.cs

```
using UnityEngine;

namespace Menu
{
    public abstract class Menu<T> : Menu where T : Menu<T>
    {
        public static T Instance { get; private set; } = null;

        protected virtual void Awake()
        {
            if (Instance == null)
                Instance = (T)this;
            else
                Destroy(this.gameObject);
        }

        protected virtual void OnDestroy()
        {
            Instance = null;
        }

        protected static void Open()
        {
            if (Instance == null)
                MenuManager.Instance.CreateInstance<T>();
            else
                Instance.gameObject.SetActive(true);

            MenuManager.Instance.OpenMenu(Instance);
        }

        protected static void Close()
        {
            if (Instance == null)
            {
                Debug.Log("Instance is null");
                return;
            }

            MenuManager.Instance.CloseMenu(Instance);
        }

        public override void OnBackPressed()
        {
            Close();
        }
    }

    public abstract class Menu : MonoBehaviour
    {
        [Tooltip("Destroy the Game Object when menu is closed (reduces memory usage)")]
        public bool DestroyWhenClosed = true;

        [Tooltip("Disable menus that are under this one in the stack")]
        public bool DisableMenusUnderneath = true;

        public abstract void OnBackPressed();
    }
}
```

MenuManager.cs

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

namespace Menu
{
    public class MenuManager : MonoBehaviour
    {
        public static MenuManager Instance { get; private set; } = null;

        private Stack<Menu> menus = new Stack<Menu>();

        [SerializeField]
        private MainMenu mainMenuPrefab;
        [SerializeField]
        private InGameMenu inGameMenuPrefab;
        [SerializeField]
        private PauseMenu pauseMenuPrefab;
        [SerializeField]
        private StoreMenu storeMenuPrefab;

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

            MainMenu.Show();
            DontDestroyOnLoad(this);
        }

        private void OnDestroy()
        {
            Instance = null;
        }

        public void CreateInstance<T>() where T : Menu
        {
            var prefab = GetPrefab<T>();
            Instantiate(prefab, transform);
        }

        public void OpenMenu(Menu instance)
        {
            // De-activate top menu
            if (menus.Count > 0)
            {
                if (instance.DisableMenusUnderneath)
                {
                    foreach (var menu in menus)
                    {
                        menu.gameObject.SetActive(false);

                        if (menu.DisableMenusUnderneath)
                            break;
                    }
                }

                var topCanvas = instance.GetComponent<Canvas>();
                var previousCanvas = menus.Peek().GetComponent<Canvas>();
                topCanvas.sortingOrder = previousCanvas.sortingOrder + 1;
            }
```

```
            menus.Push(instance);
        }
        /// <summary>
        /// Get prefab dynamically, based on private fields set from Unity
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <returns></returns>
        private T GetPrefab<T>() where T : Menu
        {
            // You can use public fields with too, but need to change BindingFlags
            var fields = this.GetType().GetFields(BindingFlags.NonPublic | Bind-
ingFlags.Instance | BindingFlags.DeclaredOnly);
            foreach (var field in fields)
            {
                var prefab = field.GetValue(this) as T;
                if (prefab != null)
                {
                    return prefab;
                }
            }

            throw new MissingReferenceException("Prefab not found for type " + typeof(T));
        }

        public void CloseMenu(Menu menu)
        {
            if (menus.Count == 0)
            {
                Debug.LogErrorFormat(menu, "{0} cannot be closed because menu stack is empty",
menu.GetType());
                return;
            }

            if (menus.Peek() != menu)
            {
                Debug.LogErrorFormat(menu, "{0} cannot be closed because it is not on top of
stack", menu.GetType());
                return;
            }

            CloseTopMenu();
        }

        public void CloseTopMenu()
        {
            var instance = menus.Pop();

            if (instance.DestroyWhenClosed)
                Destroy(instance.gameObject);
            else
                instance.gameObject.SetActive(false);

            // Re-activate top menu
            // If a re-activated menu is an overlay we need to activate the menu under it
            foreach (var menu in menus)
            {
                menu.gameObject.SetActive(true);

                if (menu.DisableMenusUnderneath)
                    break;
            }
        }
```

```
    }

}
```

SimpleMenu.cs

```
namespace Menu
{
    /// <summary>
    /// A base menu class that implements parameterless Show and Hide methods
    /// </summary>
    public abstract class SimpleMenu<T> : Menu<T> where T : SimpleMenu<T>
    {
        public static void Show()
        {
            Open();
        }

        public static void Hide()
        {
            Close();
        }
    }

}
```

StoreMenu.cs

```
using UnityEngine;

namespace Menu
{
    public class StoreMenu : SimpleMenu<StoreMenu>
    {
        public void OnObjectPlacePressed(GameObject spawnObject)
        {
            InputManager.Instance.ObjectToPlace = GameObject.Instantiate(spawnObject);
            GameManager.Instance.StartGame();

            InGameMenu.Show();
        }

        public override void OnBackPressed()
        {
            GameManager.Instance.StartGame();
            base.OnBackPressed();
        }
    }

}
```

InGameMenu.cs

```
using UnityEngine;

namespace Menu
{
    public class InGameMenu : SimpleMenu<InGameMenu>
    {
        private bool isGridEnabled = true;

        public void OnPausePressed()
        {
            PauseMenu.Show();
```

```
            GameManager.Instance.PauseGame();
        }

        public void OnGridEnablePressed()
        {
            isGridEnabled = !isGridEnabled;
            GameManager.Instance.Grid.SetActive(isGridEnabled);
        }

        public void OnStorePressed()
        {
            if (InputManager.Instance.ObjectToPlace != null)
                Destroy(InputManager.Instance.ObjectToPlace);
            GameManager.Instance.PauseGame();
            StoreMenu.Show();
        }

        public void OnCameraRotatePressed(int direction)
        {
            switch (direction)
            {
                case 0:
                    InputManager.Instance.RotateCamera(InputManager.RotationDirection.Left);
                    break;
                case 1:
                    InputManager.Instance.RotateCamera(InputManager.RotationDirection.Right);
                    break;
            }

        }

    }
}
```

MainMenu.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

namespace Menu
{
    public class MainMenu : SimpleMenu<MainMenu>
    {
        public void OnPlayPressed()
        {
            MainMenu.Hide();
            InGameMenu.Show();
            SceneManager.LoadSceneAsync(1);
        }

        public override void OnBackPressed()
        {
            Application.Quit();
        }

    }
}
```

PauseMenu.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

namespace Menu
```

```
{
    public class PauseMenu : SimpleMenu<PauseMenu>
    {
        public void OnResumePressed()
        {
            GameManager.Instance.StartGame();
            PauseMenu.Hide();
        }

        public void OnExitPressed()
        {
            GameObject.Destroy(MenuManager.Instance.gameObject);
            SceneManager.LoadSceneAsync(0);
        }
    }
}
```

ICameraBehavior.cs

```
public interface ICameraBehavior
{
    void MoveCamera();
}
```

CameraNormalMode.cs

```
using UnityEngine;

public class CameraNormalMode : ICameraBehavior
{
    private Camera mainCamera;
    private Vector3 touchStart;
    private float zoomMin;
    private float zoomMax;

    public CameraNormalMode(Camera camera, float zoomMin, float zoomMax)
    {
        this.mainCamera = camera;
        this.zoomMin = zoomMin;
        this.zoomMax = zoomMax;
    }

    public void MoveCamera()
    {
        if (Input.GetMouseButtonDown(0))
        {
            touchStart = mainCamera.ScreenToWorldPoint(Input.mousePosition);
        }
        if (Input.touchCount == 2)
        {
            ZoomCamera();
        }
        else if (Input.GetMouseButton(0))
        {
            mainCamera.transform.parent.position = GetCameraMovePosition();
        }
    }

    private void ZoomCamera()
    {
        Touch touchZero = Input.GetTouch(0);
        Touch touchOne = Input.GetTouch(1);

        Vector2 touchZeroPrevPos = touchZero.position - touchZero.deltaPosition;
        Vector2 touchOnePrevPos = touchOne.position - touchOne.deltaPosition;
```

```
        float prevMagnitude = (touchZeroPrevPos - touchOnePrevPos).magnitude;
        float currentMagnitude = (touchZero.position - touchOne.position).magnitude;

        float difference = currentMagnitude - prevMagnitude;

        Zoom(difference * 0.01f);
    }
    private void Zoom(float increment)
    {
        mainCamera.orthographicSize = Mathf.Clamp(mainCamera.orthographicSize - increment,
zoomMin, zoomMax);
    }

    private Vector3 GetCameraMovePosition()
    {
        Vector3 direction = touchStart - mainCamera.ScreenToWorldPoint(Input.mousePosition);
        Vector3 cameraParentPos = mainCamera.transform.parent.position;
        Vector3 value = new Vector3(cameraParentPos.x + direction.x, cameraParentPos.y + di-
rection.y, cameraParentPos.z + direction.z);

        float coefxz = 15f / mainCamera.orthographicSize;
        float coefy = 7f / mainCamera.orthographicSize;

        return new Vector3(Mathf.Clamp(value.x, -coefxz, coefxz), Mathf.Clamp(value.y, -coefy,
coefy), Mathf.Clamp(value.z, -coefxz, coefxz));
    }

}
```

CameraObjectPlacingMode.cs

```
using UnityEngine;
using System;

public class CameraObjectPlacingMode : ICameraBehavior
{
    private GameObject objectToPlace;
    private Camera mainCamera;

    private float gridCellSize;
    public CameraObjectPlacingMode(Camera mainCamera, GameObject objectToPlace)
    {
        this.objectToPlace = objectToPlace;
        this.mainCamera = mainCamera;

        gridCellSize = GameManager.Instance.GameField.CellSize;
        objectToPlace.GetComponent<GameItem>().InPlacingMode = true;
    }

    public void MoveCamera()
    {
        Vector3 position = GetObjectSpawnpoint();
        objectToPlace.GetComponent<Rigidbody>().position = position;

        if (Input.GetMouseButtonUp(0) && ItemCanBePlaced(position))
        {
            PlaceObject(position);
        }
    }

    private Vector3 GetObjectSpawnpoint()
    {
        var groundPlane = new Plane(Vector3.up, Vector3.zero);
```

```
        Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);

        if (groundPlane.Raycast(ray, out float position))
        {
            Vector3 worldPosition = ray.GetPoint(position);
            Vector3 currentPos;

            float x = Mathf.Round(worldPosition.x / gridCellSize) * gridCellSize;
            float z = Mathf.Round(worldPosition.z / gridCellSize) * gridCellSize;

            currentPos = new Vector3(x, 0, z);

            return currentPos;
        }

        throw new ArgumentOutOfRangeException();
    }

    private void PlaceObject(Vector3 position)
    {
        GameItem gameItem = objectToPlace.GetComponent<GameItem>();
        gameItem.InPlacingMode = false;
        gameItem.transform.position = position;
        InputManager.Instance.ObjectToPlace = null;
    }

    private bool ItemCanBePlaced(Vector3 position)
    {
        GameItem gameItem = objectToPlace.GetComponent<GameItem>();

        if (gameItem.Contacts.Count < (gameItem.SizeOnGrid.x * gameItem.SizeOnGrid.y))
        {
            return false;
        }
        else
        {
            foreach (Cell cell in gameItem.Contacts)
            {
                if (!cell.isEmpty)
                {
                    return false;
                }
            }
        }
        return true;
    }
}
```

GameManager.cs

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

public class GameManager : MonoBehaviour
{
    public enum GameStates
    {
        InGame, Pause,
    }
    public GameStates CurrentGameState { get; set; }

    [field: SerializeField]
    public GameObject Grid { get; set; }
```

```
    public GameFieldGrid GameField { get; set; }

    public static GameManager Instance { get; private set; }
    private void Awake()
    {
        if (Instance == null)
            Instance = this;
        else
            Destroy(this.gameObject);

        Initalization();
    }


    private void Initalization()
    {
        MapGenerator mapGen = new MapGenerator();
        mapGen.LoadAssetsFromResources();
        mapGen.Generate();

        Menu.MenuManager[] menuManag = Resources.FindObjectsOfTypeAll<Menu.MenuManager>();
        if (menuManag.Length > 0)
        {
            menuManag[0].gameObject.SetActive(true);
        }

        StartGame();
    }

    public void PauseGame()
    {
        CurrentGameState = GameStates.Pause;
        Time.timeScale = 0;
    }
    public void StartGame()
    {
        CurrentGameState = GameStates.InGame;
        Time.timeScale = 1;
    }
}
```

InputManager.cs

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

public class InputManager : MonoBehaviour
{
    public enum RotationDirection
    {
        Left, Right
    }

    [SerializeField]
    private Camera mainCamera;

    [field: SerializeField]
    public GameObject ObjectToPlace { get; set; }

    [field: SerializeField]
    private float zoomMin { get; set; }
    [field: SerializeField]
```

```
    private float zoomMax { get; set; }
    [field: SerializeField]
    public float CameraRotationSpeed { get; set; } = 1f;


    public bool isObjectPlacing
    {
        get {
            if (ObjectToPlace == null)
                return false;
            else
                return true;
        }
    }

    private bool isCameraRotating { get; set; } = false;

    private ICameraBehavior cameraBehavior;

    public static InputManager Instance { get; private set; }
    private void Awake()
    {
        if (Instance == null)
            Instance = this;
        else
            Destroy(this.gameObject);
    }

    private void Start()
    {
        StartCoroutine(CameraBehaviorChanger());
    }
    private void Update()
    {

        if (!isCameraRotating && GameManager.Instance.CurrentGameState == GameManag-
er.GameStates.InGame)
            cameraBehavior.MoveCamera();


        if (Input.GetKeyDown(KeyCode.F))
        {
            Debug.Log("F pressed");
            Debug.Log("Camera rotation: " + isCameraRotating);
            RotateCamera(RotationDirection.Left);
        }
    }

    private IEnumerator CameraBehaviorChanger()
    {
        while (true) {
            cameraBehavior = new CameraNormalMode(mainCamera, zoomMin, zoomMax);
            yield return new WaitUntil(() => isObjectPlacing);
            cameraBehavior = new CameraObjectPlacingMode(mainCamera, ObjectToPlace);
            yield return new WaitUntil(() => !isObjectPlacing);
        }
    }

    public void RotateCamera(RotationDirection direction)
    {
        if (!isCameraRotating)
        {
            isCameraRotating = true;
            StartCoroutine(RotateCamera(direction, CameraRotationSpeed) );
        }
```

```
    }

    private IEnumerator RotateCamera(RotationDirection direction, float inTime)
    {
        Quaternion fromAngle = mainCamera.transform.parent.transform.rotation;
        int angles = 90;
        switch (direction)
        {
            case RotationDirection.Left:
                angles = 90;
                break;
            case RotationDirection.Right:
                angles = -90;
                break;
        }
        Quaternion toAngle = Quaterni-
on.Euler(mainCamera.transform.parent.transform.eulerAngles + new Vector3(0, angles, 0));
        for (var t = 0f; t < 1f; t += Time.deltaTime / inTime)
        {
            mainCamera.transform.parent.transform.rotation = Quaternion.Slerp(fromAngle, toAn-
gle, t);
            yield return null;
        }
        isCameraRotating = false;
    }
}
```

MapGenerator.cs

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

public class MapGenerator
{
    private SpawnZone[] spawnZones;
    private GameObject cellPrefab;
    private Dictionary<SpawnZone.ObjectType, List<BackgroundObject>> backgoundObjectsByType;

    public void Generate()
    {
        SpawnBackgroundObjects();
        CreateGameField();
    }

    public void LoadAssetsFromResources()
    {
        spawnZones = Resources.LoadAll("SpawnRestrictions",
typeof(SpawnZone)).Cast<SpawnZone>().ToArray();
        cellPrefab = Resources.Load("Cell", typeof(GameObject)) as GameObject;
        GameObject[] backgroundObjects = Resources.LoadAll("BackgroundObjects",
typeof(GameObject)).Cast<GameObject>().ToArray();
        FillBackgoundObjectByType(backgroundObjects);
    }

    private void FillBackgoundObjectByType(GameObject[] gameObjects)
    {
        backgoundObjectsByType = new Dictionary<SpawnZone.ObjectType,
List<BackgroundObject>>();

        foreach (SpawnZone.ObjectType objectType in
Enum.GetValues(typeof(SpawnZone.ObjectType)) )
        {
```

```
            backgoundObjectsByType.Add(objectType, new List<BackgroundObject>());
        }

        foreach (GameObject gameObject in gameObjects)
        {
            BackgroundObject bckgrObject;
            gameObject.TryGetComponent<BackgroundObject>(out bckgrObject);
            SpawnZone.ObjectType objectType = bckgrObject.ObjectType;

            backgoundObjectsByType[objectType].Add(bckgrObject);
        }

    }

    private void CreateGameField()
    {
        int fieldSize = 20;
        GameManager.Instance.GameField = new GameFieldGrid(fieldSize);

        Vector3 center = Vector3.zero;
        Vector3 startPos = new Vector3((center.x + GameManager.Instance.GameField.Width/2f) -
0.5f, 0.5f, (center.z + GameManager.Instance.GameField.Height / 2f) - 0.5f);
        Vector3 pos = startPos;

        GameObject grid = new GameObject("Grid");
        grid.transform.position = Vector3.zero;

        for(int i = 0; i < GameManager.Instance.GameField.Cells.GetLength(0); i++)
        {
            for (int j = 0; j < GameManager.Instance.GameField.Cells.GetLength(1); j++)
            {
                GameManager.Instance.GameField.Cells[i, j] = GameOb-
ject.Instantiate(cellPrefab, pos, Quaternion.identity, grid.transform).GetComponent<Cell>(); ;
                GameManager.Instance.GameField.Cells[i, j].GameFieldPossition = new Vec-
tor2Int(i, j);

                pos = new Vector3(pos.x - 1f, pos.y, pos.z);
            }
            pos = new Vector3(startPos.x, pos.y, pos.z - 1f);
        }
    }

    private void SpawnBackgroundObjects()
    {
        GameObject backgroudParent = new GameObject("BackgoundObjects");
        backgroudParent.transform.position = Vector3.zero;

        foreach (SpawnZone spawnZone in spawnZones)
        {
            int objectCount = UnityEngine.Random.Range(spawnZone.minObjectCount,
spawnZone.maxObjectCount);

            Vector3 origin = spawnZone.transform.position;
            Vector3 range = spawnZone.transform.localScale / 2.0f;

            for (int i = 0; i < objectCount; i++)
            {
                //Getting the random object type to spawn form the allowed object types
                SpawnZone.ObjectType objectTypeToSpawn =
spawnZone.AllowedObjects[UnityEngine.Random.Range(0, spawnZone.AllowedObjects.Length)];

                //Getting the random object to spawn specified by its object type from the
dictionary
```

```
                var spawnObject = backgoundOb-
jectsByType[objectTypeToSpawn][UnityEngine.Random.Range(0, backgoundOb-
jectsByType[objectTypeToSpawn].Count)];

                int attempts = 0;
                bool canSpawn = false;

                Vector3 randomRange = new Vector3(UnityEngine.Random.Range(-range.x, range.x),
0,
                                                  UnityEngine.Random.Range(-range.z, range.z));
                Vector3 randomCoordinate = origin + randomRange;
                Vector3 randomRotation = new Vector3(0, UnityEngine.Random.Range(0f, 360f),
0);
                Vector3 randomScale = new Vec-
tor3(UnityEngine.Random.Range(spawnObject.MinSize.x, spawnObject.MaxSize.x),
                    UnityEngine.Random.Range(spawnObject.MinSize.y, spawnObject.MaxSize.y),
                    UnityEngine.Random.Range(spawnObject.MinSize.z, spawnObject.MaxSize.z));

                canSpawn = OverlapBackgroundSpawnCheck(randomCoordinate, 1);

                while (!canSpawn)
                {
                    randomRange = new Vector3(UnityEngine.Random.Range(-range.x, range.x), 0,
                                              UnityEngine.Random.Range(-range.z, range.z));
                    randomCoordinate = origin + randomRange;
                    attempts++;

                    if (attempts > 60)
                    {
                        randomCoordinate = Vector3.zero;
                        attempts = 0;
                        break;
                    }
                }

                if (canSpawn)
                {
                    GameObject gm = spawnObject.gameObject;
                    gm.transform.localScale = randomScale;
                    GameObject.Instantiate(gm, randomCoordinate, Quaterni-
on.Euler(randomRotation), backgroudParent.transform);
                }
            }
        }
    }

    private bool OverlapBackgroundSpawnCheck(Vector3 center, float radius)
    {
        int layerMask = 1 << 6;
        Collider[] hitColliders = Physics.OverlapSphere(center, radius, layerMask);
        if (hitColliders.Length > 0)
            return false;
        return true;
    }
}
```

LevelLoader.cs

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

public class LevelLoader : MonoBehaviour
```

```
{
    [SerializeField]
    private int tempIndex;
    [SerializeField]
    private Slider progressBar;
    [SerializeField]
    private TMP_Text progressPercentageText;

    private void Awake()
    {
        GameObject.FindObjectOfType<Menu.MenuManager>().gameObject.SetActive(false);
        LoadLevel(tempIndex);
    }

    public void LoadLevel(int sceneIndex)
    {
        StartCoroutine(LoadSceneAsync(sceneIndex));
    }

    private IEnumerator LoadSceneAsync(int sceneIndex)
    {
        AsyncOperation operation = SceneManager.LoadSceneAsync(sceneIndex);

        while (!operation.isDone)
        {
            float progress = Mathf.Clamp01(operation.progress/0.9f);
            progressBar.value = progress;
            progressPercentageText.text = progress * 100f + "%";

            yield return null;
        }
    }
}
```

BackgroundObject.cs

```
using UnityEngine;

public class BackgroundObject : MonoBehaviour
{
    [field: SerializeField]
    public LayerMask layerMask { get; set; }

    [field: SerializeField]
    public SpawnZone.ObjectType ObjectType { get; set; }

    [field: SerializeField]
    public Vector3 MinSize { get; set; }
    [field: SerializeField]
    public Vector3 MaxSize { get; set; }
}
```

Cell.cs

```
using UnityEngine;

public class Cell : MonoBehaviour
{
    public Cell(int x, int y)
    {
        GameFieldPossition = new Vector2Int(x, y);
    }

    private void Awake()
```

```
    {
        Size = 1f;
    }

    public bool isEmpty {
        get {
            if (Contains == null)
                return true;
            else
                return false;
        }
    }

    [field: SerializeField]
    public Vector2Int GameFieldPossition { get; set; }

    public GameItem Contains { get; set; }

    /// <summary>
    /// The cell size in Unity units
    /// </summary>
    public float Size { get; set; }

    private void OnTriggerStay(Collider collider)
    {
        if (collider.TryGetComponent(out GameItem gameItem))
        {
            if(!gameItem.InPlacingMode)
                Contains = gameItem;
        }
    }
    private void OnTriggerExit(Collider collider)
    {
        if(Contains != null)
        {
            Contains = null;
        }
    }

    private void OnDrawGizmos()
    {
        if (isEmpty)
            Gizmos.color = new Color(0, 1f, 0f, 0.3f); //Green
        else
            Gizmos.color = new Color(0f, 0f, 1f, 0.3f); //Blue

        Gizmos.DrawCube(new Vector3(this.transform.position.x, this.transform.position.y-0.5f,
this.transform.position.z) , new Vector3(1f, 0.1f, 1f));
    }
}
```

GameFieldGrid.cs

```
using UnityEngine;

public class GameFieldGrid
{
    public GameFieldGrid(int size)
    {
        Height = size;
        Width = size;

        Cells = new Cell[Height, Width];
    }
    public GameFieldGrid(int width, int height)
```

```
    {
        Height = height;
        Width = width;
        Cells = new Cell[Height, Width];
    }

    public int Height { get; private set; }
    public int Width { get; private set; }

    public Vector2Int Size
    {
        get => new Vector2Int(Height, Width);
    }

    public Cell[,] Cells { get; private set; }

    public float CellSize
    {
        get => Cells[0, 0].Size;
    }

}
```

SpawnZone.cs

```
using UnityEngine;

public class SpawnZone : MonoBehaviour
{
    /// <summary>
    /// The type of the object by size
    /// </summary>
    public enum ObjectType
    {
        Small, Medium, Large
    }

    [field: SerializeField]
    public Color GizmosColor { get; set; }  = new Color(0.5f, 0.5f, 0.5f, 0.2f);
    [SerializeField]
    public ObjectType[] AllowedObjects;
    [field: SerializeField]
    public int minObjectCount { get; set; }
    [field: SerializeField]
    public int maxObjectCount { get; set; }

    public Vector3 Size
    {
        get => this.transform.localScale;
        set => this.transform.localScale = value;
    }

    private void OnDrawGizmos()
    {
        Gizmos.color = GizmosColor;
        Gizmos.DrawCube(transform.position, transform.localScale);
    }
}
```

GameItem.cs

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

```
public class GameItem : MonoBehaviour
{
    /// <summary>
    /// Returns the uniq game item id based on Unity.GetInstanceID()
    /// </summary>
    public int Id { get => this.gameObject.GetInstanceID(); }

    [field: SerializeField]
    public string Name { get; set; }

    [SerializeField]
    public List<Cell> Contacts = new List<Cell>();

    /// <summary>
    /// Object size on grid, by default 1x1
    /// </summary>
    [field: SerializeField]
    public Vector2Int SizeOnGrid { get; set; } = Vector2Int.one;

    public bool InPlacingMode { get; set; }

    private void OnDrawGizmos()
    {
        Gizmos.color = Color.cyan;
        BoxCollider coll = this.gameObject.GetComponent<BoxCollider>();

        Vector3 collCent = transform.TransformPoint(coll.center);
        Vector3 center = new Vector3(collCent.x, 0, collCent.z);
        Vector3 halfExtends = new Vector3((coll.bounds.size.x / 2) - 0.025f, 0.25f,
(coll.bounds.size.z / 2) - 0.025f);
        Gizmos.DrawWireCube(center, halfExtends * 2);
    }

    private void FixedUpdate()
    {
        CellCollisionCheck();
    }

    private void CellCollisionCheck()
    {
        float cellSize = GameManager.Instance.GameField.Cells[0, 0].Size;

        BoxCollider coll = this.gameObject.GetComponent<BoxCollider>();
        Vector3 collCent = transform.TransformPoint(coll.center);
        Vector3 center = new Vector3(collCent.x, 0, collCent.z);
        Vector3 halfExtends = new Vector3((coll.bounds.size.x / 2) - 0.025f, 0.25f,
(coll.bounds.size.z / 2) - 0.025f);
        Collider[] hitColliders = Physics.OverlapBox(center, halfExtends,
this.gameObject.transform.rotation);

        Contacts.Clear();
        if (hitColliders.Length > 0)
        {
            foreach(Collider collider in hitColliders)
            {
                Cell cell;
                if(collider.TryGetComponent<Cell>(out cell))
                    Contacts.Add(cell);
            }

        }
    }
}
```

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

РОЗРОБКА ГРИ – СТРАТЕГІЇ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ UNITY

Виконав студент 5 курсу

групи ППЗ-51
Галеєв В.В.

Керівник роботи

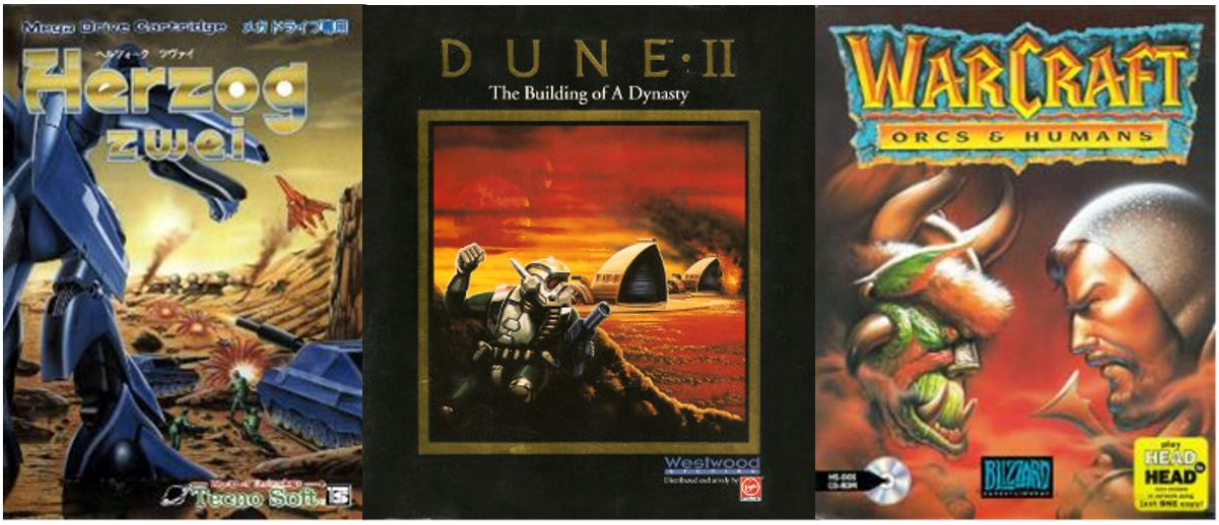
Гаманюк І.М.

Київ – 2022

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи:** розробка розважального програмного забезпечення у вигляді гри-стратегії з використанням технології Unity та створенного на мові C#.
- **Об’єкт дослідження:** підвищення закцікавленості у цільової аудиторії до жанру, та розвиток жанру комп’ютерних стратегій за допомогою розробки власного розважального програмного забезпечення - гри-стратегії з унікальними особливостями
- **Предмет дослідження:** розважальне програмне забезпечення у вигляді ігор-стратегій.

АНАЛОГИ



3

Порівняння аналогів

Ігри-стратегії	Наявність мікроменеджменту ресурсів	Чіткий поділ фракцій та їх унікальність	Максимальна кількість юнітів	Можливість відновлення пошкоджених будівель	Можливість будувати де завгодо	Залежність ходу гри від побудованних структур
Herzog Zwei (Герцог Цвей)	Номінальная та мінімальна	Немає, гравці по суті однакові	50	Ні, ремонт та відновлення неможливий	Ні, бази розташовані заздалегідь	Не залежить
Dune II	Виражена	Є чіткий поділ на дві фракції з особливостями	50	Так	Так, де завгодно, ліміт 99 будівель на всій карті	Сильно залежить
Warcraft: Orcs & Humans	Виражена	Є чіткий поділ на дві фракції з їх особливостями	50	Так	Так, де завгодно	Залежить
Command & Conquer	Виражена	Є чіткий поділ на фракції з їх особливостями	50	Так	Так, де завгодно	Сильно залежить

4

ТЕХНІЧНІ ЗАВДАННЯ

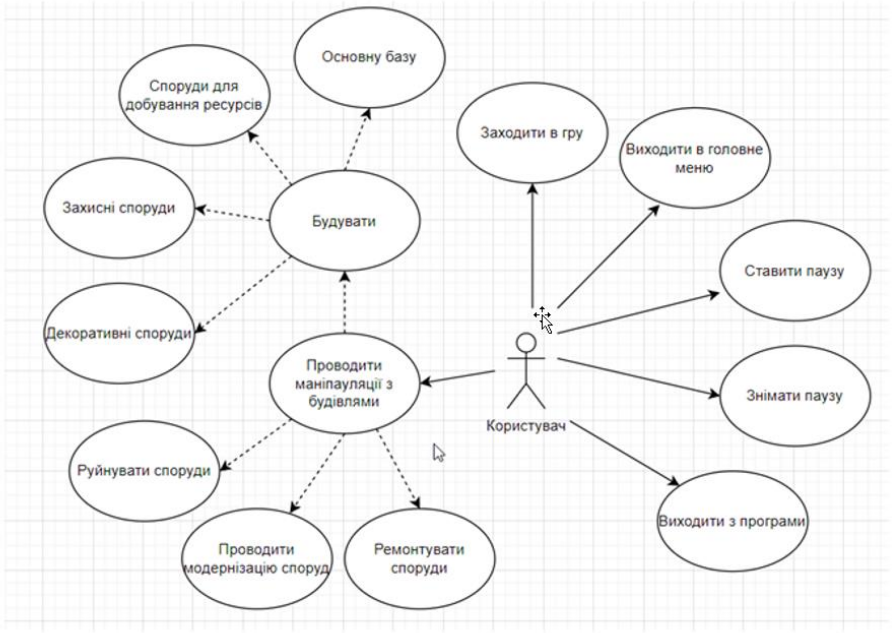
Відповідно до завдання потрібно розробити комп’ютерну стратегічну гру в реальному часі яка буде мати наступні головні функції та особливості:

- Здатність будувати будь-які споруди та саму базу у будь-якому місці
- Можливість накопичувати та добувати різноманітні ресурси
- Можливість апгрейду вже існуючих побудованних структур
- Можливість будувати додаткові захисні споруди
- Без внесення якихось кардинальних змін можливість перенести гру на інші платформи(досягається шляхом використання рушія Unity)

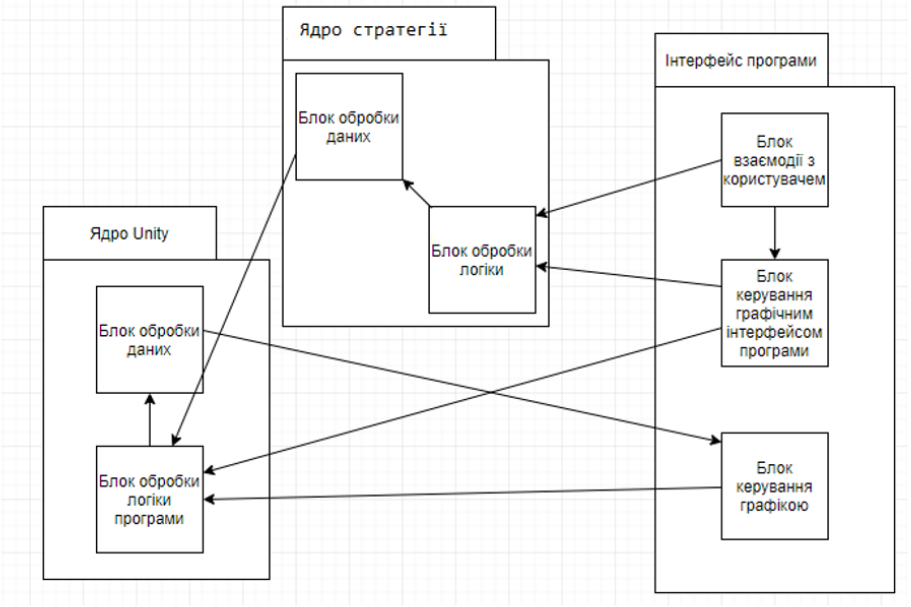
ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Для розробки гри-стратегії у якості мови програмування було обрано C#, що забезпечує реалізовано повну підтримку об’єктно-орієнтованого програмування, має потужні вбудовані функції для обробки текстової інформації, а також широку підтримку та розповсюдження, а також у якості ігрового рушія було обрано Unity, що забезпечує прискорення розробки завдяки зручному інтерфейсу, вбудованним функціям та налаштуванням, а також можливістю збирати готовий проект під різні платформи.

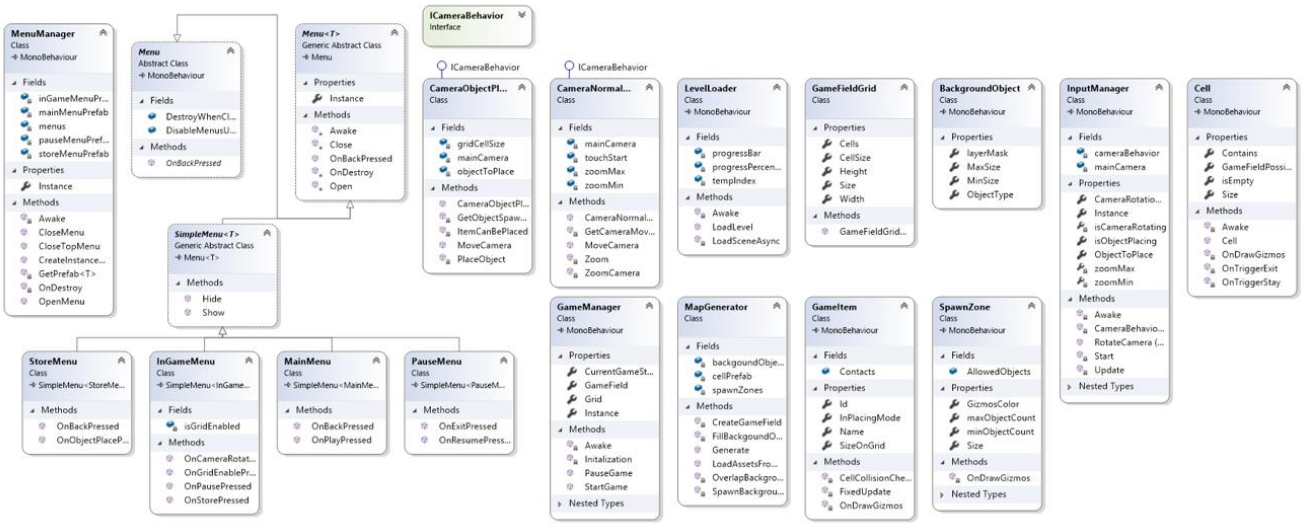
Діаграма прецедентів Користувача



Фунціональна схема роботи гри



Діаграма ієрархії класів



ВИСНОВКИ

В результаті виконання дипломної роботи було досягнуто поставлених цілей і виконано розробку розважального програмного забезпечення у вигляді гри-стратегі, яке може бути використано для розважальних цілей під час супроводу дозвілля. Програмне забезпечення реалізоване у вигляді програми та задовольняє вимогам, поставленим в технічному завданні і виконує всі основні функції. Програмне забезпечення може працювати під управлінням операційної системи Windows 7 та вище, та потребує наявність комп'ютера що відповідає як мінімум мінімальним системним вимогам. Тестування та випробування даного програмного продукту показало, що програмне забезпечення успішно справляється з поставленими перед ним задачами, та не має критичних недоліків та багів. Використання програмного забезпечення через використання типового для граця інтерфейса не потребує додаткового навчання.

Робота пройшла апробацію на Науково-технічна конференція «Застосування програмного забезпечення в ІКТ», м. Київ, ДУТ, 20 квітня 2022 року. За результатами апробації опубліковано тези доповідей: розробка гри-стратегії з використання технології Unity.

ДЯКУЮ ЗА УВАГУ!