МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інтелектуальних інформаційних систем
_____Юрій КОНДРАТЕНКО
«____»_____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

РОЗРОБКА 2D ГРИ "ЗАХИСТ ЗАМКУ" НА ДВИГУНІ UNITY3D

Спеціальність 122 Комп'ютерні науки

Освітня програма «Комп'ютерні науки»

Здобувач _____Данило ЦИГАНОК
«___»_____ 2025 р.

Керівник канд. техн. наук, доцент _____ Євген СІДЕНКО
«___»_____ 2025 р.

Миколаїв – 2025

Чорноморський національний університет імені Петра Могили
(повне найменування закладу вищої освіти)

Факультет	Комп’ютерних наук
Кафедра	Інтелектуальних інформаційних систем
Рівень вищої освіти	Перший (бакалаврський)
Освітній ступень	Бакалавр
Спеціальність	122 Комп’ютерні науки
Освітня програма	Комп’ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри інтелектуальних інформаційних систем

_____ Юрій КОНДРАТЕНКО

«____» _____ 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу здобувача

Циганка Данила Дмитровича
((прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Розробка 2D гри "Захист замку" на двигуні Unity3D».

Керівник роботи: Сіденко Євген Вікторович, доцент кафедри інтелектуальних інформаційних систем, канд. техн. наук, доцент.

Затверджена наказом ЧНУ ім. Петра Могили від «18» листопада 2024 р. № 321.

2. Строк представлення кваліфікаційної роботи «___» _____ 2025 р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні: Очікуваний результат: повнофункціональний прототип 2D-гри «Захист замку» жанру tower defense з реалізованою системою розміщення та апгрейду веж, динамічною системою

хвиль ворогів, зручним користувацьким інтерфейсом та стабільним перфомансом у Unity3D.

Початкові дані: технічне завдання на гру, функціональна модель, набір ігрових спрайтів і параметрів ворогів/веж, репозиторій із вихідним кодом і конфігураціями Unity.

4. Перелік питань, що підлягають розробці: проаналізувати існуючі підходи до створення ігрових додатків жанру tower defense; обґрунтувати вибір рушія Unity, мови C# та середовища розробки Visual Studio; спроєктувати функціональну модель гри (діаграма варіантів використання, контури автоматики); реалізувати основні ігрові механіки та інтерфейс користувача (розміщення веж, стрільба, система хвиль, UI); провести тестування системи та оцінити її стабільність і продуктивність.

5. Перелік графічних матеріалів: класова діаграма внутрішньої архітектурихема основних алгоритмів (спаун ворогів, логіка апгрейду); скриншоти UI-інтерфейсу (головне меню, ігрова сцена, панель веж); таблиці результатів тестування (Test-Case)

Керівник роботи _____ Євген СІДЕНКО
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Данило ЦИГАНОК
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата видачі завдання «20» грудня 2024 р.

КАЛЕНДАРНИЙ ПЛАН
кваліфікаційної роботи

Тема: Розробка 2D гри "Захист замку" на двигуні Unity3D

№	Найменування роботи	Початок	Закінчення	Примітки
1	Отримання завдання на виконання КР	18.12.2024	20.12.2024	
2	Аналіз предметної області та постановка задачі	21.12.2024	30.01.2025	
3	Огляд літературних джерел за темою кваліфікаційної роботи, зокрема існуючих ігрових рішень	31.01.2025	01.03.2025	
4	Проєктування функціоналу та внутрішньої архітектури гри	02.03.2025	01.04.2025	
5	Вибір засобів розробки, підготовка робочого середовища	02.04.2025	15.05.2025	
6	Перший попередній захист КР на засіданні комісії кафедри	16.05.2025	16.05.2025	
7	Корегування роботи за результатами попереднього захисту	29.05.2025	29.05.2025	
8	Другий попередній захист КР на засіданні комісії кафедри	30.05.2025	30.05.2025	
9	Доробка та остаточне оформлення КР	31.05.2025	10.06.2025	
10	Подання КР, її електронної копії та інших документів (відгуку, рецензії) до захисту	11.06.2025	11.06.2025	

Керівник роботи _____ Євген СІДЕНКО
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Здобувач _____ Данило ЦИГАНОК
(Особистий підпис) *(Власне ім'я ПРІЗВИЩЕ)*

Дата складання календарного плану
«29» січня 2025 р.

АНОТАЦІЯ

до кваліфікаційної роботи
здобувача групи 402 ЧНУ ім. Петра Могили

Циганка Данила Дмитровича

на тему: **“РОЗРОБКА 2D ГРИ "ЗАХИСТ ЗАМКУ" НА ДВИГУНІ UNITY3D”**

Актуальність даної роботи полягає у зростаючому попиті на 2D ігри жанру tower defense, які поєднують простоту реалізації з глибокими ігровими механіками, доступністю для різних платформ та популярністю серед користувачів. Розробка такої гри у середовищі Unity3D дозволяє створити інтерактивний продукт із багатофункціональним інтерфейсом та ефективною системою покращень.

Об’єктом роботи є процес розробки інтерактивних 2D-ігор у середовищі Unity3D.

Предметом роботи є методи створення внутрішньої логіки гри, реалізація користувацького інтерфейсу, обробка подій, механіка хвиль і взаємодія гравця з системою веж.

Метою роботи є розробка повнофункціонального прототипу 2D-гри жанру tower defense.

У результаті виконання роботи було спроєктовано функціональну модель гри, реалізовано основні ігрові механіки, інтерфейс користувача та проведено тестування продуктивності і стабільності. Розробка здійснена з використанням мови C# у середовищі Visual Studio.

Дана робота складається з трьох розділів. У першому розділі проведено аналіз сучасного стану 2D ігор жанру tower defense. Другий розділ присвячений проектуванням ігрової системи. У третьому розділі наведено результати розробки і тестування прототипу гри. Загальний обсяг роботи – 88 сторінок. Кваліфікаційна робота містить 1 додаток, 25 рисунків, 3 таблиці і 34 джерел посилання.

Ключові слова: 2D гра, tower defense, Unity3D, ігровий прототип, C#, розробка ігор, інтерфейс користувача, апгрейд веж.

ABSTRACT

to the qualification work by the student of the group 402 of Petro Mohyla Black Sea National University

Tsyhanok Danylo

“DEVELOPMENT OF 2D GAME 'CASTLE DEFENSE' ON THE UNITY3D ENGINE”

The relevance of this study lies in the growing demand for 2D games in the tower defense genre, which combine simplicity of implementation with deep gameplay mechanics, cross-platform accessibility, and popularity among users. Developing such a game in the Unity3D environment allows creating an interactive product with a multifunctional interface and an efficient upgrade system.

The object of the research is the process of developing interactive 2D games in the Unity3D environment.

The subject of the research is the methods of creating the internal game logic, implementation of the user interface, event handling, wave mechanics, and player interaction with the tower system. The goal of the work is to develop a fully functional prototype of a 2D tower defense game.

As a result of the work, a functional game model was designed, core gameplay mechanics and user interface were implemented, and performance and stability testing was conducted. The development was done using the C# language in the Visual Studio environment.

As a result of the work, a functional game model was designed, core gameplay mechanics and user interface were implemented, and performance and stability testing was conducted. The development was done using the C# language in the Visual Studio environment.

This work consists of three chapters. The first chapter analyzes the current state of 2D tower defense games. The second chapter is dedicated to the design of the game system. The third chapter presents the results of the prototype development and testing.

The overall scope of the work is 68 pages. Thesis contains 1 application, 25 figures, 3 tables and 34 references in it.

Key words: 2D game, tower defense, Unity3D, game prototype, C#, game development, user interface, tower upgrade.

ЗМІСТ

ВСТУП

На сучасному етапі розвитку інформаційних технологій відеоігри стали не лише потужним інструментом розваги, а й важливим засобом комунікації, навчання та реалізації творчого потенціалу. Ігрова індустрія демонструє стрімке зростання, охоплюючи широкий спектр платформ — від мобільних пристроїв до стаціонарних комп’ютерів і консолей. У цьому контексті розробка 2D-ігор залишається актуальною завдяки своїй доступності, нижчому порогу входу для розробників та великій популярності серед користувачів.

Актуальність теми зумовлена широким попитом на компактні та креативні 2D-ігри, а також наявністю сучасних технологій, які дозволяють створювати якісні ігрові продукти навіть у складі невеликих команд. Серед таких технологій особливе місце займає рушій Unity3D, який поєднує у собі гнучкість, багатофункціональність та підтримку як 2D, так і 3D-графіки. Мова програмування C#, що використовується в середовищі Unity, дозволяє реалізовувати складну логіку поведінки ігрових об’єктів, оптимізувати продуктивність та створювати масштабовану архітектуру.

Об’єктом роботи є процес розробки інтерактивних 2D-ігор у середовищі Unity3D.

Предметом роботи є методи створення внутрішньої логіки гри, реалізація користувацького інтерфейсу, обробка подій, механіка хвиль і взаємодія гравця з системою веж.

Метою роботи є розробка повнофункціонального прототипу 2D-гри жанру tower defense.

Для досягнення поставленої мети необхідно виконати наступні завдання:

- проаналізувати існуючі підходи до створення ігрових додатків жанру tower defense;
- обґрунтувати вибір рушія, мови програмування та середовища розробки;
- спроєктувати функціональну модель гри;

- реалізувати основні ігрові механіки та інтерфейс;
- провести тестування системи та оцінити її стабільність.

У процесі виконання роботи застосовувались методи об'єктно-орієнтованого програмування, структурного проєктування, модульного тестування та емпіричного аналізу продуктивності.

1 АНАЛІЗ СУЧАСНОГО СТАНУ РОЗРОБКИ 2D-ІГОР У ЖАНРІ TOWER DEFENSE

1.1 Ігрові додатки та особливості розробки 2D-ігор

Сфера ігрових додатків посідає особливе місце в сучасному інформаційному середовищі, де цифрові технології дедалі більше впливають на спосіб сприйняття світу, обробки інформації та взаємодії з нею. У добу стрімкої диджиталізації, коли людина щоденно взаємодіє з великою кількістю візуального й інтерактивного контенту, ігри стали не просто способом проведення дозвілля. Вони перетворилися на потужний засіб формування світогляду, впливу на емоційно-психологічний стан, розвитку навичок і навіть міжособистісної комунікації. Гра вже давно перестала бути виключно розвагою — вона трансформувалася у вагомий інструмент комунікації, навчання, культурного обміну, а також соціального моделювання [1]. Сучасні геймери — це не лише підлітки, а представники всіх вікових груп, які через гру занурюються у змодельовані світи, вирішують складні завдання, будують стратегії та реалізують креативні задуми. Саме ігрові додатки стали тим середовищем, у якому технічна реалізація взаємодіє з естетичним досвідом, створюючи нові форми віртуального буття [2].

Особливе місце в цій екосистемі займають ігрові додатки, що реалізовані у форматі двовимірної графіки. Вони зберігають актуальність завдяки своїй простоті, швидкості розробки, а також емоційній виразності, яку здатні передавати навіть базові візуальні рішення. Ігрові додатки, зокрема ті, що розроблені на основі двовимірної графіки, мають низку переваг, які сприяють їхній популярності серед розробників та користувачів. Простота реалізації, менші вимоги до обчислювальних ресурсів та можливість охоплення ширшої аудиторії — все це робить 2D-ігри привабливими як для інді-розробників, так і для великих студій [3]. У процесі створення таких ігор набагато легше впроваджувати нові механіки, тестувати сценарії та швидко реагувати на зворотний зв’язок від користувачів. Крім того, двовимірні ігри часто

використовуються як навчальні проекти у вищих навчальних закладах, оскільки вони дозволяють студентам зосередитися на основах програмування, дизайну та логіки гри без необхідності занурюватися у складні аспекти тривимірної графіки [4]. Таким чином, 2D-ігри не лише зручні у розробці, а й мають величезну педагогічну цінність у підготовці майбутніх IT-фахівців.

Розвиток ігрових додатків тісно пов'язаний з еволюцією апаратного забезпечення та програмних платформ. Із кожним роком користувачі отримують дедалі потужніші пристрої, здатні обробляти складні графічні сцени, водночас із зменшенням енергоспоживання та підвищенням доступності. З появою потужних мобільних пристроїв та широкого доступу до інтернету, ігри стали доступними практично кожному користувачу, незалежно від його місця проживання чи соціального статусу [5]. Унаслідок цього виникла потреба в універсальних інструментах розробки, здатних забезпечити кросплатформенність, масштабованість та стабільну продуктивність. Це призвело до зростання попиту на ігрові додатки, що, у свою чергу, стимулювало розвиток інструментів для їх створення, таких як Unity3D, Unreal Engine та інші [6].

Unity3D, зокрема, став однією з найпопулярніших платформ для розробки ігор завдяки своїй доступності, гнучкості та широкому спектру можливостей [7]. Його переваги включають інтуїтивно зрозумілий інтерфейс, розвинену систему документації, велику бібліотеку готових ресурсів та плагінів, а також інтеграцію з популярними мовами програмування, зокрема C#. Його підтримка як 2D, так і 3D графіки, а також можливість розгортання ігор на різних платформах робить його ідеальним вибором для розробників з усього світу. Крім того, велика спільнота користувачів та наявність численних навчальних матеріалів сприяють швидкому освоєнню платформи навіть новачками [8]. Завдяки Unity3D багато незалежних студій отримали змогу реалізувати свої проєкти без значних фінансових вкладень, що раніше було майже неможливо.

Ігрові додатки також відіграють важливу роль у розвитку креативності та інновацій. У процесі створення ігор відбувається активне залучення дизайнерів, програмістів, сценаристів і звукових інженерів, що стимулює розвиток міждисциплінарного підходу. Вони надають розробникам можливість експериментувати з новими ідеями, механіками та стилями, що сприяє появі унікальних та захоплюючих ігор [9]. Гра може бути як засобом вираження авторської позиції, так і інструментом соціальної критики. Багато успішних проектів починалися як невеликі експерименти, які з часом переросли у повноцінні продукти з мільйонами користувачів по всьому світу [10]. Ці історії успіху мотивують нові команди пробувати свої сили у геймдеві, створюючи конкурентне середовище та підвищуючи загальний рівень індустрії.

Окрім розважальної функції, ігрові додатки все частіше використовуються в освітніх цілях. Їх інтерактивна природа дозволяє глибше залучати учнів у навчальний процес, активізуючи увагу, пам’ять, логічне мислення та здатність до аналізу. Вони дозволяють створювати інтерактивні навчальні середовища, які сприяють кращому засвоєнню матеріалу та розвитку критичного мислення. Ігри можуть моделювати реальні ситуації, що дозволяє користувачам набувати практичних навичок у безпечному та контрольованому середовищі [11]. Сюди входять і симулятори поведінки в екстрених ситуаціях, і тренажери для медиків, і серйозні ігри для вивчення історії, мови чи математики.

У контексті українського ринку, розвиток ігрових додатків набуває особливого значення. В умовах війни та глобальної нестабільності геймдев стає не лише точкою економічного зростання, а й простором креативної рефлексії й національної самоідентифікації. Зростаюча кількість талановитих розробників, підтримка з боку держави та інвесторів, а також зростаючий інтерес з боку користувачів сприяють формуванню сприятливого середовища для розвитку індустрії [12]. Українські ігрові студії вже здобули визнання на міжнародному рівні, що свідчить про високий потенціал вітчизняного ринку [13]. Проєкти на кшталт «STALKER», «Metro», «Sheriff

of Baghdad» або «Farlanders» демонструють, що українські ігри здатні конкурувати на світовому ринку як у технічному, так і в наративному плані.

Таким чином, ігрові додатки є важливим елементом сучасної цифрової культури, що поєднує в собі технологічні інновації, креативність та соціальну взаємодію. Їхній вплив на різні аспекти життя продовжує зростати, відкриваючи нові можливості для розвитку як окремих особистостей, так і суспільства в цілому. У майбутньому саме ця сфера може стати одним із основних драйверів цифрової економіки та інтелектуального капіталу.

1.2 Технології 2D-розробки в сучасній індустрії ігор

Двовимірна розробка в індустрії ігор становить одну з найбільш стабільних і водночас динамічних складових цифрової творчості. Незважаючи на активний розвиток тривимірної графіки, шейдерів, фотореалістичних середовищ та кінематографічної анімації, 2D-графіка продовжує впевнено утримувати свої позиції. Це свідчить про її універсальність, адаптивність і здатність передавати глибокий сенс навіть через прості форми. Попри домінування 3D-графіки в AAA-сегменті, саме 2D лишається основою значної частини ігрового контенту, особливо у мобільному, інді та навчальному секторах [14]. У цих категоріях розробники прагнуть до економії ресурсів, високої продуктивності та творчої свободи, а 2D якраз і дає змогу досягти цієї мети. Причина цього криється не лише в меншій складності реалізації чи нижчих технічних вимогах, а й у стилістичній виразності, що надає іграм власну візуальну мову та автентичність [15].

Історично 2D-графіка стала основою становлення всієї гейм-індустрії. Розробка 2D-ігор історично передувала тривимірним підходам. Перші аркадні автомати, консолі та домашні ПК працювали саме з двовимірним простором, створюючи фундамент жанрів ігор, що зберегли актуальність донині. Такі класичні жанри як платформери, shoot-em-up, beat-em-up або JRPG зародилися саме у двовимірному вимірі. І хоча сучасні технології дозволяють вільно використовувати 3D-графіку

навіть у мобільних додатках, 2D-графіка не лише не зникає, а навпаки — отримує нове дихання в умовах інді-буму, ретро-естетики та художніх експериментів [16]. Сучасна 2D-естетика часто обігрує ностальгію за піксель-артом, водночас вдосконалюючи її сучасними технологіями, такими як динамічне освітлення, партикли або шейдери.

З погляду технології, розробка 2D-ігор вимагає специфічного підходу до побудови сцени, анімації, фізики та взаємодії. У середовищі Unity, яке підтримує повноцінну 2D-парадигму, розробники мають змогу працювати з тайл-сетами, спрайтами, колайдерами та камерою, спеціально адаптованою до двовимірного простору [17]. Окрім цього, 2D Toolkit, Cinemachine, Tilemap Editor, та інші плагіни суттєво спрощують роботу над ігровим простором. Більш того, інструментарій Unity дозволяє гнучко налаштовувати освітлення, шейдери та ефекти, що традиційно асоціюються з 3D-графікою, але можуть застосовуватись і в площині 2D [18]. Це означає, що сьогодні 2D-графіка — це не лише пікселі або мультиплікаційні образи, а повноцінна візуальна платформа, яка може виглядати не менш вражаюче за 3D при значно менших витратах часу та ресурсів.

Однією з переваг 2D-розробки є швидкість прототипування. Завдяки відносній простоті структурних елементів, невеликі команди або окремі ентузіасти можуть створювати ігрові проєкти у стислі строки, зберігаючи при цьому високий рівень якості та художньої завершеності [19]. У перші тижні після створення концепції вже можна мати іграбельну демо-версію, на базі якої проводиться тестування та збір зворотного зв'язку. Це відкриває простір для креативних експериментів, особливо в умовах обмежених ресурсів, характерних для навчальних або незалежних проєктів. 2D-ігри стають ідеальним полігоном для дослідження поведінки гравця, балансування механік і нарощування складності.

Із появою сучасних графічних редакторів, таких як Aseprite, Krita або Adobe Photoshop, а також спеціалізованих фреймворків для анімації, створення двовимірних об’єктів стало доступним навіть для непрофесійних художників [20]. Ці інструменти мають низький поріг входу та потужні функції: створення анімаційних циклів,

кадрування, покадрове редагування. Крім того, розвинена екосистема безкоштовних та комерційних ассетів, доступних на платформах типу Unity Asset Store чи itch.io, дозволяє значно спростити процес візуального оформлення гри [21]. Розробникам більше не потрібно створювати все з нуля — вони можуть зосередитися на унікальності геймплею та сценарії, комбінуючи наявні ресурси для досягнення бажаного художнього ефекту.

Сучасна 2D-розробка не обмежується лише класичними платформерами або аркадами. Вона охоплює складні жанрові гібриди: від стратегій реального часу до симуляторів життя, від пригодницьких візуальних новел до процедурно згенерованих roguelike-світів [22]. Ігри з відкритим світом, системами розвитку персонажів, нелінійними сюжетами, часто реалізовані у двовимірній площині. При цьому спрощення геометрії сцени компенсується складністю логіки, геймдизайну та сюжетної побудови, що робить 2D-проєкти повноцінними представниками сучасної ігрової індустрії. Сучасні 2D-ігри не поступаються 3D-колегам у глибокій механіці, атмосферності чи здатності емоційно залучити гравця.

Таким чином, 2D-розробка є не лише історичною спадщиною, а й активним сегментом сучасного геймдеву, що продовжує еволюціонувати. Її актуальність визначається не технічними обмеженнями, а естетичними й концептуальними можливостями, які дозволяють створювати глибокі, захопливі та водночас доступні для реалізації проєкти. У майбутньому 2D не зникне — воно трансформується, доповнюватиметься новими техніками, але збереже свою суть: передавати історії простою мовою образів, яку розуміє кожен.

1.3 Огляд існуючих рішень

Станом на сьогодні ринок 2D-ігор жанру tower defense є надзвичайно насиченим, проте не втрачає актуальності ні серед розробників, ні серед гравців. Це стосується не лише повноцінних комерційних релізів, а й навчальних проєктів, демо-версій, експериментальних продуктів та прототипів. Головною перевагою

двовимірних ігор є їхня технічна доступність: розробити базову механіку, реалізувати основну логіку та створити ігрову петлю можна без глибоких знань у галузі комп'ютерної графіки. Саме тому серед існуючих рішень переважають невеликі, але добре структуровані проєкти, які виконують чітко окреслену функцію — демонстрацію механіки, реалізацію геймплейного патерну чи візуального стилю.

Серед найпоширеніших типів ігор у цьому сегменті вирізняються такі жанри, як tower defense, платформери, шутери з видом збоку, квестові історії з обмеженим простором, а також головоломки. З метою кращого розуміння різноманіття реалізацій у жанрі, нижче наведено порівняльну таблицю найпоширеніших рішень, які демонструють як комерційні, так і навчальні підходи до створення 2D tower defense ігор (табл. 1.1).

Таблиця 1.1 — Порівняльна характеристика існуючих 2D tower defense рішень

Назва гри / проєкту	Тип реалізації	Основні особливості	Графіка	Доступність коду	Використаний рушій
Kingdom Rush	Комерційна гра	Покращення веж, сюжет, декілька типів ворогів	Стилізована, мальована	Ні	Власний рушій
The Last Shelter (demo)	Незалежна гра	Динамічний темп, вежі з апгрейдами	2D sci-fi	Ні	Flash/HTML5

TowerDefenseTemplate	Освітній шаблон	Базова логіка хвиль, UI, префаби ворогів	Мінімалістична	Так	Unity

Кінець таблиці 1.1

TD_Game_Unity	Open-source прототип	Шаблон з апгрейдами, внутрішня валюта	Піксельна	Так	Unity
MiniTD 2	Indie/Steam реліз	Проста механіка, зростаюча складність	Абстрактна, геометрія	Ні	Unity
Itch.io Tower Demo	Освітній проєкт	Префаби ворогів, waypoint-система, таймер хвиль	Векторна	Так	Unity

Як видно з таблиці, проєкти значно відрізняються за стилістикою, складністю механік і технічними підходами кожен із них має свої характерні риси реалізації, зумовлені не лише традицією, а й технічними обмеженнями та очікуваннями цільової аудиторії. У випадку з tower defense — саме цей жанр був обраний для реалізації даного проєкту — на перший план виходить баланс між хвилями ворогів, швидкістю

їхнього просування, ефективністю розміщення оборонних споруд та зростаючою складністю.

Більшість існуючих 2D tower defense рішень базуються на класичній моделі: вороги рухаються по заздалегідь визначеному маршруту, гравець розміщує оборонні вежі, кожна з яких має певний радіус ураження, швидкість атаки та тип дії. Популярні рішення реалізують систему покращення веж або заміну базових одиниць на більш ефективні, вводять кілька типів ворогів з різними характеристиками та додають можливість використання спеціальних здібностей або ресурсів, які активуються вручну.

Деякі реалізації роблять акцент на сюжетній складовій: кожен рівень може бути вплетений у більшу історію, де гравець виступає як захисник королівства, командир підрозділу або навіть чарівник, що протистоїть силам темряви. У таких випадках стилістика гри, діалоги, навіть музичний супровід грають не меншу роль, ніж сама механіка. Інші ж обирають мінімалістичний підхід, де весь фокус зосереджений на ігровому процесі: кількість ворогів, динаміка бою, візуальні ефекти та геймдизайнерські виклики.

З точки зору технічної реалізації, багато існуючих рішень базуються на рушії Unity3D, який дозволяє зручно працювати з двовимірною графікою, реалізовувати фізику, створювати анімації, керувати об’єктами на сцені через компонентну систему та інтегрувати звук, UI та зберігання даних. Unity підтримує створення шляхів для ворогів через spline або waypoint-систему, дозволяє гнучко задавати поведінку штучного інтелекту через скрипти та забезпечує контроль над продуктивністю навіть у простих проєктах.

Цікаво, що серед відкритих репозиторіїв на платформах на кшталт GitHub або itch.io переважають саме навчальні приклади. Вони часто мають обмежену функціональність, але добре структурований код і супровідну документацію. Такі проєкти містять базову механіку розміщення веж, створення ворогів із префабів, логіку втрат життя, поразки та перемоги, іноді — систему апгрейдів і

внутрішньоігрову валюту. Це дозволяє студентам і початківцям швидко зануритися в процес розробки ігор та краще зрозуміти принципи роботи рушія.

Окремо слід відзначити проєкти, які роблять акцент на візуальному стилі. Серед них є як ручна піксельна графіка з ностальгічним присмаком, так і векторні рішення з чіткою геометрією та яскравими кольорами. Це додає жанру естетичної різноманітності: один tower defense може виглядати як класичний fantasy-бій, інший — як футуристичне протистояння роботів, третій — як абстрактна гра з геометричними формами та мінімалістичним UI.

Часто в існуючих рішеннях реалізується одна з ключових механік: шкала хвиль ворогів. Вона слугує індикатором прогресу рівня і дозволяє гравцю планувати свої дії. Деякі ігри ускладнюють цю схему, вводячи варіативність маршрутів, раптові зміни типу ворогів або навіть босів, які порушують загальну логіку. Такі елементи сприяють повторному проходженню гри, підвищенню інтересу та варіативності геймплею.

1.4 Постановка задачі

У процесі створення програмного продукту типу 2D-ігри жанру tower defense необхідно чітко визначити як загальну мету розробки, так і конкретні технічні задачі, що виникають на різних етапах її реалізації. Об’єктом розробки в межах цієї дипломної роботи є ігровий застосунок «Swamp Towers» — двовимірна гра, створена на рушії Unity, у якій користувачеві пропонується захистити територію від хвиль ворогів, розміщуючи оборонні вежі різних типів.

Метою роботи є проєктування, розробка та часткове вдосконалення базової ігрової моделі tower defense, яка реалізована у зазначеному репозиторії. У ході реалізації поставлено завдання не лише створити функціонуючий ігровий прототип, але й забезпечити внутрішню логічну цілісність гри, збалансованість ігрових механік, коректну роботу інтерфейсних елементів, адаптивність системи до різних параметрів вхідних даних, а також візуальну та аудіальну завершеність.

Розробка гри в межах цього проєкту охоплює декілька послідовних завдань, кожне з яких впливає на загальну якість продукту. Передусім, йдеться про реалізацію логіки руху ворогів відповідно до заданого маршруту із дотриманням базових принципів оптимізації. Далі необхідно забезпечити можливість інтерактивного розміщення веж на ігровому полі із перевіркою коректності позиціонування та відображенням радіусу дії. Окрему увагу слід приділити алгоритму атаки, який має враховувати як тип ворога, так і параметри конкретної вежі — її швидкість стрільби, силу влучання, тип ураження та пріоритет цілі.

До задач належить також створення системи хвиль, яка передбачає зміну кількості ворогів, їхню швидкість, тип, стійкість та наявність босів. При цьому кожна наступна хвиля повинна бути складнішою за попередню, що вимагає розробки динамічного механізму масштабування складності гри. Не менш важливою є реалізація внутрішньоігрової економіки: кожна вежа повинна мати вартість, а за знищення ворогів гравець має отримувати ресурси, які дозволяють купувати нові вежі або покращувати наявні. Таким чином, формується цикл «ризик–винагорода», що підтримує інтерес до гри.

До задач дипломної розробки також належить формування базового інтерфейсу користувача: відображення ресурсів, кількості життів, поточної хвилі, повідомлень про перемогу чи поразку. Усі елементи мають бути адаптовані до двовимірного візуального середовища та забезпечувати інтуїтивно зрозумілу взаємодію з гравцем. Крім цього, гра повинна мати початкове меню, можливість перезапуску рівня та виходу з програми.

З огляду на використання середовища Unity, додатковими задачами постає оптимізація структури проєкту, правильне використання компонентної архітектури, дотримання принципів MVC у межах логіки гри, а також забезпечення читабельності та реюзабельності коду. Окремим напрямом передбачається реалізація базової анімації, інтеграція звукових ефектів та оформлення ігрової сцени у стилі swamp/fantasy, який відповідає ідейному задуму гри.

Висновки до розділу 1

У результаті аналізу предметної області було виявлено суттєве зростання популярності цифрових ігор, зокрема у 2D-сегменті, який демонструє стабільний попит серед широкої аудиторії завдяки доступності, простоті сприйняття та кросплатформенності. Теоретичне обґрунтування розвитку 2D-ігор дало змогу окреслити основні принципи, за якими здійснюється проєктування, побудова геймплею та візуальне оформлення. Проведене порівняння популярних рішень у жанрі tower defense засвідчило, що більшість з них мають обмеження або у візуальній гнучкості, або в алгоритмічній складності, що відкриває можливість для створення нового, більш гнучкого й адаптивного ігрового продукту.

Уточнення вимог до функціональності, стабільності, розширюваності та користувацького досвіду дозволило сформулювати чітке технічне завдання для дипломного проєкту. Визначено, що створення власної 2D tower defense-гри дає змогу не лише реалізувати практичні навички у сфері ігрового програмування, а й протестувати сучасні технології й інструменти на практиці. Зібрана аналітична база послугувала підґрунтям для прийняття обґрунтованих рішень щодо вибору рушія, мови програмування та структури майбутньої гри.

2 ПРОЕКТУВАННЯ ГРИ

2.1 Проектування функціоналу

На етапі проєктування функціональної моделі гри необхідно всебічно проаналізувати можливі дії користувача в межах ігрового процесу, а також визначити функціональні реакції системи на ці дії. Це дозволяє забезпечити логічну цілісність архітектури застосунку, уникнути суперечностей у поведінці системи та закласти основу для майбутнього масштабування. Одним з ефективних інструментів, що дозволяє формалізувати такі взаємозв’язки, є діаграма варіантів використання (use-case diagram). Вона забезпечує графічне представлення ключових сценаріїв взаємодії між користувачем і системою, а також дозволяє структурувати функціональність застосунку на ранніх етапах розробки. У контексті нашого проєкту така діаграма візуалізує обсяг функціоналу, демонструє можливі послідовності дій і відображає логіку взаємозв’язків між компонентами ігрового процесу.

Юзером у контексті цієї гри виступає кінцевий гравець — особа, яка керує ігровими процесами через візуальний інтерфейс користувача. Система, в свою чергу, — це вся сукупність внутрішніх механізмів, алгоритмів та логічних модулів, побудованих на рушії Unity, що відповідає за реакцію на дії гравця, відображення подій, обчислення результатів і динамічну зміну станів гри відповідно до закладених сценаріїв. Такий поділ на "акторів" і "систему" дозволяє чітко відокремити зовнішню поведінку від внутрішньої логіки, що важливо з точки зору підтримки та супроводу програмного продукту.

Рисунок 2.1 – Діаграма варіантів використання

На створеній нами діаграмі актором виступає кінцевий користувач, тобто гравець, який взаємодіє з ігровим застосунком через інтерфейс користувача. Центральним елементом функціональної моделі є ігрова система — набір компонентів, що реалізують обробку введених даних, змінюють стани об’єктів, контролюють взаємодію між елементами та забезпечують цілісність ігрового середовища. Умовно всю функціональність ігрової системи можна поділити на декілька великих блоків: ініціалізація ігрового середовища (запуск гри, завантаження сцени), контроль ігрового процесу (розміщення веж, початок хвилі), взаємодія з ресурсами (збирання, витрачання, апгрейди), сценарії завершення (виграш, поразка, перезапуск), а також допоміжні дії (вихід у головне меню, налаштування, пауза).

Гравець має змогу запускати гру та починати нову сесію, що ініціалізує внутрішню структуру даних, генерує мапу, розміщує початкові об’єкти та обнуляє таймери й ресурси. У рамках активної сесії користувач може розміщувати оборонні

вежі на визначених ділянках карти. Важливо, що для цього система перевіряє як логіку доступності клітинки, так і наявність необхідної кількості ресурсів. Кожна вежа володіє унікальними характеристиками — швидкістю атаки, дальністю, типом шкоди — і може бути покращена у процесі гри. Покращення вежі доступне лише після її успішного розміщення, а його вартість поступово зростає.

Система реалізує послідовну логіку хвиль — групи ворогів з’являються на стартових точках і прямують до гравцевої бази. Якщо певна кількість ворогів подолала шлях і досягла бази, система фіксує поразку. Якщо ж хвиля успішно знищена, гравець отримує нагороду — ресурси, які можна витратити на нові споруди або покращення вже наявних. Таким чином, гравець у режимі реального часу аналізує ситуацію, приймає рішення про стратегічне розміщення споруд, слідкує за темпом гри та координує захист. Завершення рівня відкриває доступ до сценаріїв перезапуску або повернення до головного меню, що є типовим варіантом розгалуження у юзер-флоу.

Діаграма також ілюструє залежність між діями гравця: деякі функції можуть виконуватись лише після виконання інших. Наприклад, дія «Покращити вежу» можлива лише після її побудови, а розміщення доступне лише в межах активної сесії гри. Це забезпечує покроковість логіки, унеможливлює хибні дії, мінімізує кількість помилок, пов’язаних із позачерговим викликом методів або аномаліями стану. Подібна модель дозволяє легко масштабувати гру: у майбутньому додати нові дії, сценарії чи типи акторів (наприклад, другого гравця або адміністратора), не змінюючи загальну логіку.

У підсумку, завдяки чіткому розмежуванню дій, структурованій логіці та побудові сценаріїв у вигляді use-case діаграми, розроблена модель системи є зрозумілою для всієї команди — від геймдизайнерів до тестувальників — і забезпечує передбачувану поведінку гри, гнучкість у реалізації та основу для подальшого розвитку проєкту.

2.2 Проектування внутрішньої будови

Внутрішня архітектура програмного забезпечення значною мірою визначає як стабільність роботи ігрового додатку, так і зручність його масштабування, супроводу та тестування. Для того, щоб забезпечити чітку логіку взаємодії між компонентами системи, було побудовано діаграму класів, яка відображає структуру коду проєкту, розробленого у межах дипломної роботи.

Діаграма класів охоплює ключові компоненти, що відповідають за побудову рівнів, логіку ворогів, систему веж, інтерфейс користувача, редакторські інструменти та перелічення, які визначають внутрішні стани та типи об’єктів. Вона демонструє як спадкування від базових класів, зокрема MonoBehaviour, так і взаємозв’язки між допоміжними структурами, які використовуються для зберігання даних та обробки внутрішніх сценаріїв.

Рисунок 2.2 – Діаграма класів

Система ворогів реалізується через такі класи, як EnemyHealth, EnemyHealthBar, EnemyInfo та EnemyInfoHolder. Клас EnemyHealth відповідає за обробку життєвого

циклу ворожих юнітів — ураження, смерть, зміна стану. EnemyHealthBar реалізує візуальне представлення кількості здоров’я над ворогом. Клас EnemyInfo містить метаінформацію про ворогів (швидкість, тип, базове здоров’я), а EnemyInfoHolder дозволяє централізовано зберігати та редагувати ці характеристики, імовірно, через редактор Unity.

Клас EnemyType є енумерацією, яка визначає типи ворогів. У поєднанні з EnemyTypeMethod, що, ймовірно, є статичним класом-утилітою, реалізовано зручний механізм класифікації ворогів, їх обробки та відображення.

Система веж представлена через класи TowerLevelInfo, TowerLevelPropertyDrawer, TowerInfoProvider, TowerModelView, TowerFindResult. Вони формують повноцінний модуль для створення, оновлення та відображення веж. Зокрема, TowerLevelInfo містить дані про рівні веж, а TowerLevelPropertyDrawer забезпечує кастомне відображення властивостей у редакторі. Клас TowerModelView відповідає за взаємодію моделі та представлення, тоді як TowerInfoProvider, очевидно, виконує функцію постачальника даних для веж. TowerFindResult є структурою, яка зберігає результат пошуку або вибору вежі, зберігаючи посилання на відповідний об’єкт.

Інтерфейс користувача представлено класами MainMenuUI та FreeSpaceUI. Перший відповідає за головне меню, запуск рівня, вихід, перезапуск тощо. FreeSpaceUI, вочевидь, взаємодіє з ігровим полем, відображаючи вільні ділянки для розміщення веж. Вони обидва є нащадками MonoBehaviour, що забезпечує їхнє підключення до об’єктів сцени.

Інструменти редактора представлені класами LevelEditor, LevelEditorWindow, EnemyInfoHolder (який має поведінку Editor). Вони дозволяють змінювати параметри гри безпосередньо з редактора Unity. LevelEditorWindow, ймовірно, відповідає за графічне представлення рівня у вигляді вікна редактора, а LevelEditor — за його логіку та обробку взаємодії.

Службові класи та структури відіграють допоміжну роль. Bullet реалізує поведінку снаряда, що випускається вежами. SpawnEnumerable оперує колекцією об'єктів, які необхідно створити, ймовірно, це частина хвилі ворогів. Limiter<TValue> є універсальним класом, що може обмежувати значення в певних межах. SerializedList<T> — кастомна колекція, пристосована до роботи з серіалізованими об'єктами в Unity Editor.

Ex — статичний клас, який імовірно містить розширення (extension methods) або службові методи для спрощення загальних дій, таких як обробка подій, математичні розрахунки чи перевірки.

Енумерації StatusWorkType та EnemyType використовуються для опису станів або типів об'єктів у грі. Вони забезпечують чітке структурування логіки, знижують кількість "магічних" значень у коді та підвищують читаємість.

Уся архітектура гри побудована на принципах модульності та поділу відповідальності. Кожен клас відповідає за окремий аспект функціональності, що спрощує підтримку проєкту та дозволяє масштабувати його в майбутньому. Наявність редакторських інструментів свідчить про орієнтацію розробки не лише на кінцевий результат, але й на зручність тестування та зміни параметрів без модифікації коду.

2.3 Вибір засобів розробки

У процесі створення ігрового застосунку ключовим є вибір рушія, який стане основою для реалізації всіх технічних та візуальних компонентів гри. Серед наявних рішень на ринку програмних засобів розробки ігор особливу увагу було приділено таким популярним платформам, як Unity, Unreal Engine, Godot, GameMaker Studio, Defold та Cocos2d. Після детального порівняльного аналізу було обґрунтовано вибір саме Unity як основи для реалізації дипломного проєкту.

Таблиця 2.1 — Порівняння рушіїв для розробки 2D-ігор

Рушій	Мова програмування	Основна спеціалізація	Переваги	Недоліки
Unity	C#	2D/3D, мультиплатформеність	Потужний редактор, Unity Asset Store, багато туторіалів	Важчий за легкі рушії, не весь функціонал безкоштовний
Unreal Engine	C++ / Blueprints	Переважно 3D	Високоякісна графіка, візуальне програмування	Складний, надлишковий для 2D, великий розмір проєктів
Godot Engine	GDScript / C#	2D/3D	Open-source, легкий, активна спільнота	Менше плагінів, слабша підтримка великих проєктів
GameMaker Studio	GML	2D	Проста логіка, малий поріг входу	Обмежений UI, складно масштабувати

Кінець таблиці 2.1

Defold	Lua	2D	Легкий, оптимізований	Незвична екосистема, вузька підтримка
Cocos2d	C++	2D	Гнучкість, високий контроль	Високий поріг входу, мало візуальних інструментів

Як видно з таблиці 2.1, Unity вигідно вирізняється серед інших рушіїв завдяки своїй універсальності, зручності у використанні та потужному функціоналу. На відміну від багатьох альтернатив, він надає можливість створювати як 2D, так і 3D ігри без необхідності перемикання між платформами або використання різних середовищ. У контексті проєкту, що стосується створення двовимірної гри, рушій надає спеціалізований 2D-режим, який оптимізує роботу зі спрайтами, анімацією, тайлмапами та фізикою. Це дає змогу суттєво зменшити час на розробку без втрати якості чи функціональності.

Порівняно з Unreal Engine, Unity виглядає значно легшим та доступнішим для невеликих команд і індивідуальних розробників. Unreal, безумовно, має вражаючі можливості у сфері 3D-графіки та фотореалізму, однак його функціонал часто є надмірним для реалізації 2D-проєктів. Крім того, робота з Unreal потребує більш глибоких знань у галузі C++ та складнішого процесу компіляції. Unity натомість дозволяє використовувати мову C#, яка є більш інтуїтивно зрозумілою для новачків та швидше опановується в контексті освітніх проєктів.

Godot Engine, який останніми роками здобуває популярність серед інді-розробників, дійсно має низку переваг, серед яких відкритий вихідний код, легка вага

та активна спільнота. Проте система GDScript, що є основною мовою для написання сценаріїв у Godot, значно поступається C# як у продуктивності, так і в інструментарії. Крім того, екосистема Godot поки що не має такого широкого спектра готових компонентів, інструментів та плагінів, як Unity Asset Store. Це обмежує гнучкість проєкту, особливо в частині візуального оформлення та розширення функціоналу.

GameMaker Studio є чудовим рішенням для створення простих 2D-ігор, зокрема платформерів та аркад. Він має зручний інтерфейс і дозволяє працювати без написання великої кількості коду. Проте для більш складних ігор жанру tower defense його можливості можуть виявитися недостатніми. Структура GameMaker обмежена в питаннях роботи зі складними UI-компонентами, системами частинок, редакторськими інструментами та мультиплатформенністю на рівні, який пропонує Unity. Також варто враховувати, що Unity дозволяє створювати кастомні редактори, що є критично важливим для швидкої адаптації ігрових рівнів, тоді як у GameMaker така можливість реалізується обмежено.

Рушій Cocos2d, хоч і спеціалізується на 2D-іграх, має вузьку сферу застосування. Його структура вимагає більш низькорівневого підходу, а розробка ведеться переважно на C++, що підвищує поріг входу для новачків. У порівнянні з Unity, Cocos2d виглядає менш гнучким і не має такої кількості інтегрованих засобів для створення інтерфейсу, роботи з ресурсами або побудови сцен. До того ж, для сучасного користувача критично важливими є засоби візуалізації, підтримка VR/AR, хмарні сервіси, аналітика — усі ці компоненти Unity має вже «з коробки» або доступні через офіційні модулі.

Ще одним важливим фактором стала доступність навчальних матеріалів. Unity має величезну кількість офіційних і неофіційних туторіалів, курсів, форумів та документації. Це особливо актуально в умовах освітньої розробки, коли потрібна швидка підтримка з боку спільноти або можливість знайти приклад реалізації тієї чи іншої задачі. Навіть найскладніші аспекти, такі як оптимізація продуктивності або

побудова системи збереження даних, у середовищі Unity мають детально описані рішення, які можна безпосередньо інтегрувати в проєкт.

Перевагою Unity є також його потужна сцена інструментів редактора. Зокрема, існує можливість створювати власні UI-компоненти, впроваджувати візуальне редагування даних, генерувати рівні, тестувати поведінку об'єктів в реальному часі. В умовах розробки гри жанру tower defense це дозволяє створити повноцінну систему редагування рівнів, налаштування ворогів, веж, параметрів шкоди, швидкості, логіки хвиль тощо. Такий підхід суттєво скорочує час на ітеративне тестування та полегшує процес геймдизайну.

Важливим критерієм є й мультиплатформенність. Unity дозволяє експортувати гру під широкий спектр платформ — Windows, Android, iOS, WebGL, Linux, а також для ігрових консолей. Це відкриває можливості для майбутнього масштабування проєкту, публікації в маркетплейсах, адаптації до різних аудиторій. Альтернативні рушії часто мають обмеження на безкоштовне комерційне використання або вимагають додаткових модулів для розгортання на різних платформах. Unity у цьому плані більш універсальний і прозорий.

У процесі створення ігрового застосунку ключовим є вибір рушія, який стане основою для реалізації всіх технічних та візуальних компонентів гри. Серед наявних рішень на ринку програмних засобів розробки ігор особливу увагу було приділено таким популярним платформам, як Unity, Unreal Engine, Godot, GameMaker Studio, Defold та Cocos2d. Після детального порівняльного аналізу було обґрунтовано вибір саме Unity як основи для реалізації дипломного проєкту.

Unity є одним із найпоширеніших рушіїв у світі завдяки своїй універсальності, зручності у використанні та потужному функціоналу. На відміну від багатьох альтернатив, він надає можливість створювати як 2D, так і 3D ігри без необхідності перемикання між платформами або використання різних середовищ. У контексті проєкту, що стосується створення двовимірної гри, рушій надає спеціалізований 2D-

режим, який оптимізує роботу зі спрайтами, анімацією, тайлмапами та фізикою. Це дає змогу суттєво зменшити час на розробку без втрати якості чи функціональності.

Порівняно з Unreal Engine, Unity виглядає значно легшим та доступнішим для невеликих команд і індивідуальних розробників. Unreal, безумовно, має вражаючі можливості у сфері 3D-графіки та фотореалізму, однак його функціонал часто є надмірним для реалізації 2D-проєктів. Крім того, робота з Unreal потребує більш глибоких знань у галузі C++ та складнішого процесу компіляції. Unity натомість дозволяє використовувати мову C#, яка є більш інтуїтивно зрозумілою для новачків та швидше опановується в контексті освітніх проєктів.

Godot Engine, який останніми роками здобуває популярність серед інді-розробників, дійсно має низку переваг, серед яких відкритий вихідний код, легка вага та активна спільнота. Проте система GDScript, що є основною мовою для написання сценаріїв у Godot, значно поступається C# як у продуктивності, так і в інструментарії. Крім того, екосистема Godot поки що не має такого широкого спектра готових компонентів, інструментів та плагінів, як Unity Asset Store. Це обмежує гнучкість проєкту, особливо в частині візуального оформлення та розширення функціоналу.

GameMaker Studio є чудовим рішенням для створення простих 2D-ігор, зокрема платформерів та аркад. Він має зручний інтерфейс і дозволяє працювати без написання великої кількості коду. Проте для більш складних ігор жанру tower defense його можливості можуть виявитися недостатніми. Структура GameMaker обмежена в питаннях роботи зі складними UI-компонентами, системами частинок, редакторськими інструментами та мультиплатформенністю на рівні, який пропонує Unity. Також варто враховувати, що Unity дозволяє створювати кастомні редактори, що є критично важливим для швидкої адаптації ігрових рівнів, тоді як у GameMaker така можливість реалізується обмежено.

Рушій Cocos2d, хоч і спеціалізується на 2D-іграх, має вузьку сферу застосування. Його структура вимагає більш низькорівневого підходу, а розробка ведеться переважно на C++, що підвищує поріг входу для новачків. У порівнянні з

Unity, Cocos2d виглядає менш гнучким і не має такої кількості інтегрованих засобів для створення інтерфейсу, роботи з ресурсами або побудови сцен. До того ж, для сучасного користувача критично важливими є засоби візуалізації, підтримка VR/AR, хмарні сервіси, аналітика — усі ці компоненти Unity має вже «з коробки» або доступні через офіційні модулі.

Ще одним важливим фактором стала доступність навчальних матеріалів. Unity має величезну кількість офіційних і неофіційних туторіалів, курсів, форумів та документації. Це особливо актуально в умовах освітньої розробки, коли потрібна швидка підтримка з боку спільноти або можливість знайти приклад реалізації тієї чи іншої задачі. Навіть найскладніші аспекти, такі як оптимізація продуктивності або побудова системи збереження даних, у середовищі Unity мають детально описані рішення, які можна безпосередньо інтегрувати в проєкт.

Перевагою Unity є також його потужна сцена інструментів редактора. Зокрема, існує можливість створювати власні UI-компоненти, впроваджувати візуальне редагування даних, генерувати рівні, тестувати поведінку об'єктів в реальному часі. В умовах розробки гри жанру tower defense це дозволяє створити повноцінну систему редагування рівнів, налаштування ворогів, веж, параметрів шкоди, швидкості, логіки хвиль тощо. Такий підхід суттєво скорочує час на ітеративне тестування та полегшує процес геймдизайну.

Важливим критерієм є й мультиплатформенність. Unity дозволяє експортувати гру під широкий спектр платформ — Windows, Android, iOS, WebGL, Linux, а також для ігрових консолей. Це відкриває можливості для майбутнього масштабування проєкту, публікації в маркетплейсах, адаптації до різних аудиторій. Альтернативні рушії часто мають обмеження на безкоштовне комерційне використання або вимагають додаткових модулів для розгортання на різних платформах. Unity у цьому плані більш універсальний і прозорий.

Ефективна реалізація програмного забезпечення неможлива без вибору відповідного середовища розробки. У випадку створення 2D-ігрового додатку на

Unity із використанням мови C# ключовим елементом технічної інфраструктури стає середовище, яке забезпечить повноцінну інтеграцію з рушієм, підтримку мови, зручність налагодження, відстеження помилок, автодоповнення коду та швидкий доступ до документації. З огляду на вищезазначені критерії, вибір було зроблено на користь Visual Studio — одного з найбільш потужних, стабільних та функціонально насичених середовищ для роботи з C# та Unity.

Visual Studio є офіційно рекомендованим середовищем для розробки в Unity, що забезпечує найкращу підтримку компонентів .NET, інтеграцію з редактором Unity та повну сумісність з бібліотеками, які використовуються у грі. Середовище надає гнучкі інструменти для організації коду, налагодження проєкту, запуску юніт-тестів, перевірки на винятки, профілювання продуктивності та оптимізації алгоритмів. Це дозволяє не лише реалізовувати функціонал, а й підтримувати якість програмного продукту на високому рівні.

У порівнянні з альтернативами, такими як Visual Studio Code, JetBrains Rider, MonoDevelop чи Sublime Text, Visual Studio має низку переваг, які є критичними для складних проєктів. Наприклад, Visual Studio Code хоч і легкий та універсальний, однак не забезпечує повної глибини інтеграції з Unity без складної налаштування сторонніх розширень. Дебаг Unity у VS Code часто працює нестабільно, а автодоповнення може потребувати додаткових конфігурацій. У свою чергу, Visual Studio «з коробки» підтримує Unity Debugger, працює з усіма типами збірок, і дозволяє переглядати значення змінних, стек викликів, локальні змінні та об'єкти сцени у реальному часі.

JetBrains Rider — ще одне сучасне середовище з потужним функціоналом. Проте воно є платним, що обмежує його доступність у контексті навчального чи індивідуального проєкту. Крім того, Rider має дещо вищі системні вимоги, споживає більше ресурсів та може потребувати налаштування сторонніх плагінів для певних можливостей Unity. Visual Studio у версії Community є повністю безкоштовною для некомерційного використання та навчання, при цьому пропонує всі ті ж можливості

— включно з Live Share, Git-інтеграцією, вбудованим терміналом і потужним менеджером розширень.

MonoDevelop, який раніше поставлявся разом з Unity, давно втратив актуальність і більше не підтримується. Його функціональність є обмеженою, інтерфейс — застарілим, а інтеграція з новими версіями Unity практично відсутня. Sublime Text або Notepad++ — текстові редактори, що можуть використовуватись для редагування коду, однак не надають повноцінного середовища для налагодження, роботи з автокомплітом, консоллю, системою контролю версій чи тестуванням. Таким чином, для складних проєктів вони категорично не підходять.

Visual Studio також має високий ступінь кастомізації. Можна налаштовувати шаблони коду, колірну схему, гарячі клавіші, автоматичні імпорти, форматування. Крім того, підтримується створення проєктів не лише під Unity, а й під .NET, консольні застосунки, бібліотеки, інструменти редактора — що дозволяє масштабувати проєктну структуру за потреби. Завдяки наявності розширень, таких як ReSharper, Unity Tools, C# Snippets та IntelliCode, середовище набуває додаткової гнучкості й швидкодії.

Однією з найпотужніших функцій Visual Studio є система інтерактивного дебагінгу. Вона дозволяє не просто зупиняти виконання на брейкпоінтах, а й змінювати значення змінних на льоту, відстежувати хід виконання корутини, читати вміст словників і колекцій, працювати з умовними зупинками, логуванням подій та викликів. Це критично важливо в ігровій логіці, особливо під час розробки складних систем взаємодії між вежами, ворогами та UI.

Visual Studio підтримує інтеграцію з Git: можна безпосередньо з IDE створювати репозиторії, виконувати коміти, працювати з гілками, робити злиття змін. У командній розробці або при підтримці історії змін це значно спрощує робочий процес. Також існує підтримка Azure DevOps, GitHub Actions, CI/CD-пайплайнів, що робить VS актуальним навіть для великих продакшн-команд, не кажучи вже про дипломний проєкт.

Нарешті, слід зазначити, що Visual Studio активно оновлюється, має офіційну підтримку з боку Microsoft, сертифіковану документацію, а також величезну кількість навчальних матеріалів — як у текстовому, так і відеоформаті. Це дозволяє швидко опанувати навіть найскладніші аспекти розробки, і не витрачати час на боротьбу з інструментом замість реалізації функціоналу.

Висновки до розділу 2

У межах другого розділу було здійснено повноцінне проєктування ігрової системи з урахуванням функціональних, структурних та архітектурних вимог. В основу розробки покладено підхід до поетапного моделювання функціоналу, що дало змогу виокремити ключові сценарії взаємодії користувача з грою. Побудована діаграма варіантів використання відобразила базові сценарії: розміщення веж, старт хвилі ворогів, взаємодію з інтерфейсом, апгрейд елементів оборони тощо. Це дозволило сформувати логічний каркас поведінки гравця в рамках заданого ігрового процесу.

Розробка внутрішньої будови системи на основі діаграми класів забезпечила чітке розмежування обов’язків між елементами архітектури. Класи ворогів, веж, хвиль, UI-компонентів та логіки рівнів розроблено таким чином, щоб підтримувалась модульність, розширюваність і зручність подальшої підтримки коду. Особливу увагу приділено механізмам агрегації інформації, інкапсуляції станів об’єктів і забезпеченню взаємодії між компонентами через спеціалізовані сервіси.

Обґрунтовано вибір основних технологічних засобів: рушія Unity як платформи розробки, мови програмування C# як основного інструмента логіки та середовища Visual Studio як інтегрованого рішення для ефективної розробки і налагодження. Проведене порівняння з альтернативами підтвердило доцільність обраного стека технологій у контексті реалізації саме 2D-ігрового проєкту в жанрі tower defense. Unity забезпечив усі необхідні засоби для візуального проєктування рівнів, підтримку

анімацій, UI, фізики, а також інструменти швидкого прототипування та редакторського налаштування параметрів.

Таким чином, результати другого розділу забезпечили повну технічну та логічну підготовку до етапу реалізації, надавши чітке бачення структури майбутнього ігрового застосунку та заклавши основу для його ефективної реалізації й подальшого масштабування.

3 РОЗРОБКА ГРИ "ЗАХИСТ ЗАМКУ"

3.1 Розробка основних алгоритмів

Розробка алгоритмічної основи гри — ключовий етап реалізації програмного продукту, який визначає логіку функціонування внутрішніх процесів, динаміку геймплею та узгодженість взаємодії між об’єктами. Саме на цьому етапі створюється серце гри — набір сценаріїв, які забезпечують її життєвий цикл: від генерації ворогів і керування їх параметрами до оновлення веж, обрахунку шкоди, визначення фіналу рівня. Від якості алгоритмів залежить загальна стабільність гри, її збалансованість, динаміка та ігрова привабливість. Якщо інтерфейс відповідає за перше враження, то логіка гри формує досвід, що залишається з користувачем надовго. У межах дипломного проєкту реалізовано низку повнофункціональних алгоритмів, які охоплюють усі базові складові логіки гри. Ці алгоритми працюють у тісному зв’язку з графічними об’єктами, подієвою системою Unity та ігровим станом, який постійно змінюється у відповідь на дії гравця або поведінку ворогів. Нижче наведено аналіз основних алгоритмів з фрагментами коду, що їх ілюструють, для глибшого розуміння принципів побудови ігрової механіки.

Алгоритм управління параметрами ворога

Однією з найважливіших частин алгоритмічного ядра гри є модуль, який відповідає за керування параметрами ворожих одиниць. Цей функціонал є фундаментом для побудови системи бою, адже саме вороги створюють виклик для гравця, впливають на складність гри та формують основну напругу ігрового процесу. Клас EnemyInfo виконує роль базового контейнера даних, який інкапсулює всю необхідну інформацію про конкретного ворожого юніта. У цьому класі зберігаються такі характеристики, як швидкість руху (speed), поточне здоров’я (health), максимальне здоров’я (maxHealth), потенційна шкода (damage), яку ворог може завдати у разі досягнення бази гравця, а також кількість ігрових ресурсів — у даному випадку монет (money), які гравець отримує у разі успішного знищення ворога.

Таке чітке відокремлення параметрів дозволяє гнучко змінювати властивості різних типів ворогів без необхідності переписувати інші модулі гри, що значно підвищує масштабованість і спрощує подальшу підтримку проєкту. Крім того, параметри типу Min(0) або Min(1) виступають формою базової валідації значень у редакторі Unity, що дозволяє зменшити кількість помилок на етапі конфігурації.

Особливої уваги заслуговує властивість RelativeHealth, яка реалізує обчислення відносного рівня здоров'я поточного ворога як відсоткового співвідношення поточного до максимального значення. Цей параметр не тільки використовується для відображення шкали здоров'я в інтерфейсі, але й може слугувати тригером для поведінкових змін ворога: наприклад, зміна кольору, швидкості або типу атаки при зменшенні здоров'я нижче певного порогу. Таким чином, навіть базовий клас надає великий простір для побудови адаптивної та динамічної бойової системи.

```
[Serializable]
public class EnemyInfo
{
    [Min(0)] public float speed = 1;
    [Min(0)] public int health;
    [Min(1)] public int maxHealth;
    [Min(0)] public int damage = 1;
    [Min(0)] public int money = 1;
    public EnemyType enemyType;

    public float RelativeHealth => health / (float)maxHealth;
}
```

Рисунок 3.1 — Клас EnemyInfo

У загальному контексті, клас EnemyInfo є зразком правильної інкапсуляції даних, що дозволяє централізовано зберігати всі ключові характеристики ворожої одиниці, взаємодіяти з ними упродовж усього ігрового циклу та забезпечити коректну інтеграцію з іншими модулями — такими як система генерації ворогів, менеджер

хвиль або бойовий інтерфейс. Це також є першим кроком до розширення — додавання нових типів ворогів, зміни балансу, застосування елементів ШІ — усе це можливо завдяки правильно побудованій основі.

Алгоритм послідовного створення ворогів

У більшості сучасних ігор, де реалізовано механіку проходження рівнів через хвилі ворогів, критично важливою є чітка та контрольована система генерації супротивників. Від її реалізації залежить не лише ігровий баланс, але й ритм, напруга, складність і задоволення гравця від процесу. Саме для цього у межах розробки проєкту було створено спеціалізований клас SpawnEnumerable, який реалізує шаблон IEnumerable. Завдяки цьому клас стає джерелом об’єктів для конструкції foreach, що дозволяє легко ітеративно перебирати ворогів, які мають бути створені під час проходження конкретного рівня. Такий підхід надає змогу реалізувати не просто статичну чергу монстрів, а повноцінну багаторівневу систему хвиль із вкладеними підхвилями, що значно збагачує ігровий процес.

Клас SpawnEnumerable вбудовується у загальний цикл гри та забезпечує поетапне, контрольоване породження ворожих юнітів згідно з наперед визначеним сценарієм. Кожна хвиля (wave) може містити декілька підхвиль (subWave), що дозволяє задати не тільки кількість ворогів, але й їх типи, частоту появи та структуру атаки. У кожній підхвилі вказується конкретна кількість ворогів (count), а також об’єкт (monster), який потрібно інстанціювати. В процесі виконання циклу foreach у методі GetEnumerator() відбувається послідовна генерація ворогів із різних підхвиль, що робить систему гнучкою, масштабованою й адаптивною до будь-якого рівня складності.

```
    public class SpawnEnumerable : IEnumerable<GameObject>
    {
        public IEnumerator<GameObject> GetEnumerator()
        {
            for (var levelLength = _level.waves.Length; CurrentPositionWave <
levelLength; CurrentPositionWave++)
```

```
            {
                CurrentWave = _level.waves[CurrentPositionWave];
                foreach (var subWave in CurrentWave.subWaves)
                {
                    for (int i = 0; i < subWave.count; i++)
                    {
                        yield return subWave.monster;
                        CountSpawnMonsters++;
                    }
                }
            }
            CurrentPositionWave--;
        }
    }
```

Рисунок 3.2 — Клас SpawnEnumerable

Однією з ключових переваг цього алгоритму є точний контроль над кількістю та порядком створених ворогів. Підрахунок здійснюється за допомогою лічильників: CountSpawnMonsters — для вже створених ворогів, і CountMonsters — для загальної кількості, яка може бути обчислена на основі сумарного значення count усіх підхвиль усіх хвиль рівня. Це дозволяє не лише моніторити хід бою, а й виводити індикатори прогресу, наприклад, MonsterFillPercentage, який може використовуватись у графічному інтерфейсі для демонстрації, скільки ворогів уже пройшло і скільки ще очікується.

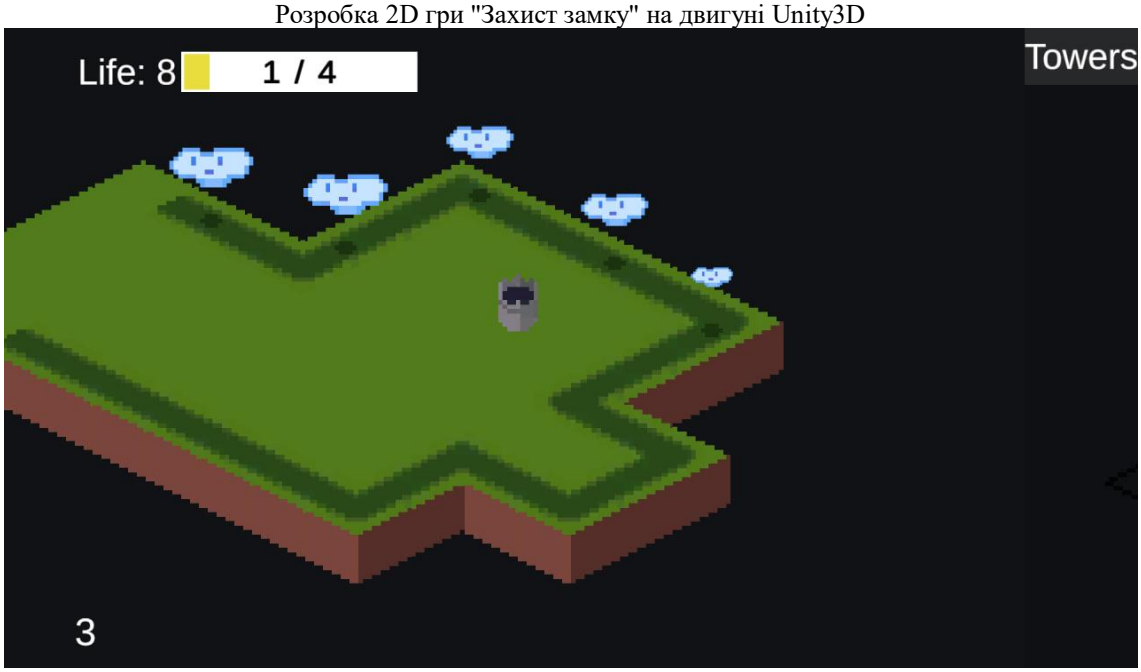

Рисунок 3.3 — Спавн ворогів

Окрім цього, структура SpawnEnumerable робить можливим легко реалізовувані сценарії для адаптивної складності гри. Наприклад, у складніших рівнях можуть бути додані умови, при яких після кожної підхвилі перевіряється стан гравця (залишок здоров’я бази, кількість ресурсів тощо), і на основі цих даних змінюється склад або кількість ворогів наступної підхвилі. Такий гнучкий підхід до побудови хвиль робить гру динамічною і менш передбачуваною.

Крім того, впровадження інтерфейсу IEnumerable дозволяє відокремити логіку створення ворогів від решти ігрових компонентів. Це підвищує модульність архітектури гри та полегшує тестування, повторне використання коду й подальше розширення функціоналу. Наприклад, можна реалізувати додаткові стани для кожного елемента хвилі — типу затримки перед спавном, особливих умов появи або анімаційних ефектів, — не змінюючи базову структуру класу.

Таким чином, SpawnEnumerable є прикладом грамотної організації алгоритму послідовного створення ворогів, що не лише відповідає вимогам до продуктивності та логіки гри, а й забезпечує масштабованість, зручність у розробці й контрольованість усіх етапів ігрової хвилі.

Алгоритм вибору інформації про вежу

У будь-якій грі жанру tower defense ключовим елементом геймплею є система оборонних споруд — веж. Саме вони виконують активну бойову роль, взаємодіють із ворогами, завдають шкоди, споживають ресурси та змінюють ігрову динаміку в реальному часі. Для реалізації логіки взаємодії з вежами, зокрема таких аспектів, як оновлення, порівняння рівнів, виведення характеристик у UI або визначення вартості апгрейду, необхідно мати централізований і гнучкий спосіб доступу до інформації про конкретну вежу. Цю задачу ефективно розв’язує реалізація класу TowerInfoProvider, який у поєднанні з допоміжною структурою TowerFindResult дозволяє знаходити відповідні метадані за об’єктом вежі на сцені.

Метод GetTowerFindResultByTower() є центральним компонентом цієї функціональності. Він виконує пошук відповідності між конкретним екземпляром об’єкта Tower, який існує у грі, та його даними, які зберігаються у структурі TowerInfoCollection у вигляді набору TowerLevelInfos. Кожен об’єкт TowerLevelInfo містить масив infoItems, які описують послідовні рівні вдосконалення певного типу вежі. У кожному елементі зберігається префаб вежі (tower), що виступає еталоном для порівняння з об’єктом на сцені.

```
public TowerFindResult GetTowerFindResultByTower(Tower tower)
{
    foreach (var towerLevelInfo in TowerLevelInfos)
    {
        var towerLevels = towerLevelInfo.infoItems;
        for (int j = 0; j < towerLevels.Length; j++)
        {
            var prefab = towerLevels[j].tower;
            if (tower.prefab == prefab)
                return new TowerFindResult(towerLevelInfo, j);
        }
    }
    return null;
}
```

Рисунок 3.4 — Клас TowerInfoProvider

Цей метод дозволяє зіставити об’єкт на сцені з його метаданими, що зберігаються в TowerInfoCollection. У процесі виконання методу проводиться ітерація по всіх типах доступних веж (TowerLevelInfos) та їх рівнях, після чого порівнюється префаб об’єкта tower із шаблоном збереженого рівня (towerLevels[j].tower). Якщо збіг виявлено, то створюється і повертається екземпляр TowerFindResult, який містить інформацію про конкретну вежу та її поточний рівень у системі апгрейду. Якщо відповідність не знайдено — метод повертає null, що дозволяє контролювати випадки помилок або некоректно створених об’єктів.

Рисунок 3.5 — Будівництво веж

Отриманий TowerFindResult виступає інтерфейсом доступу до структури характеристик вежі. Через властивості CurrentItem та NextItem розробник може легко визначити як поточний стан вежі, так і наступний рівень покращення — його вартість, збільшені параметри атаки, дальність, швидкість стрільби тощо. Така модель дає змогу реалізувати логіку апгрейду уніфіковано для будь-якої вежі без дублювання коду та дозволяє швидко масштабувати систему — наприклад, додати нові рівні або типи веж без переписування логіки порівняння.

Більше того, така архітектура дозволяє ефективно інтегрувати функціонал із графічним інтерфейсом гри: виводити підказки, відображати доступні оновлення, змінювати іконки, підсвічувати поліпшення. У разі розширення проєкту (додавання унікальних умінь веж, ефектів або альтернативних гілок розвитку) — TowerFindResult і відповідна система пошуку вже готові для підтримки цієї гнучкості.

У підсумку, метод GetTowerFindResultByTower() виступає критичним інструментом, який поєднує фізичні об’єкти сцени з їх логічною структурою даних, дозволяє гнучко взаємодіяти з ними та підтримує масштабованість, чистоту коду і зрозумілу архітектуру гри.

Алгоритм побудови інформаційної моделі вежі

Для забезпечення якісної взаємодії користувача з грою необхідно не лише реалізувати функціональну логіку апгрейду та бою, а й надати гравцеві зрозуміле, структуроване й візуально зручне уявлення про стан об’єктів на полі. У випадку із системою веж, це означає необхідність виводити в UI всю ключову інформацію — поточний рівень, характеристики, потенціал наступного покращення, вартість продажу та апгрейду, іконку й опис. Щоб централізовано зібрати ці дані з різних джерел і передати їх у графічний інтерфейс, у межах проєкту було реалізовано об'єкт типу TowerModelViewer.

Побудова цієї структури відбувається на основі результату пошуку через TowerFindResult. Метод ToTowerModelViewer() виконує агрегацію інформації з різних частин системи: з об’єкта вежі на сцені (Tower), з метаданих про конкретний рівень (TowerUpgradeInfo), з товару у магазині (TowerShopItem), а також з даних про наступне можливе оновлення (nextTowerLevelInfoItem). Такий підхід дозволяє звести всю інформацію в єдиний об’єкт, який потім передається у відповідні UI-компоненти, не потребуючи додаткових запитів чи обчислень у момент відображення.

```
    public static TowerModelViewer ToTowerModelViewer(
        Tower tower,
        TowerUpgradeInfo towerLevelInfoItem,
        TowerShopItem shopItem,
        int level,
        TowerUpgradeInfo nextTowerLevelInfoItem
    )
    {
        return new TowerModelViewer(
            tower.info.target,
            tower.info.damage,
            tower.Radius,
            tower.info.cooldown,
            tower.info.countMonstersDamage,
            towerLevelInfoItem.salePrice,
            nextTowerLevelInfoItem.buyPrice,
            shopItem.icon,
```

```
            shopItem.description,
            shopItem.name,
            level
        );
    }
```

Рисунок 3.6 — Клас TowerFindResult

Завдяки цій моделі забезпечується єдина точка збору усієї необхідної інформації для виводу в інтерфейс. Зокрема, сюди входять ключові бойові характеристики вежі: damage (завдана шкода), cooldown (інтервал між атаками), countMonstersDamage (кількість ворогів, які можуть бути атаковані), target (тип цілі), а також Radius — радіус дії вежі на полі. Всі ці параметри мають вирішальне значення для геймплейної стратегії гравця, тому мають бути прозоро подані в інтерфейсі.

Окрім бойових характеристик, структура TowerModelViewer містить також економічну інформацію: salePrice — вартість продажу поточної вежі, buyPrice — вартість апгрейду до наступного рівня, що зчитується з nextTowerLevelInfoItem. Це дозволяє гравцеві приймати обґрунтовані рішення: чи вигідно продавати вежу, чи варто її покращити, чи доцільно будувати аналогічну вежу ще раз.

Графічні дані представлені через icon, name, description, які походять із пов’язаного товару в магазині (TowerShopItem). Це забезпечує узгодженість між відображенням у магазині, на полі та в додаткових вікнах опису. Назва й опис можуть містити як технічні відомості, так і стилізований текст, що додає грі наративної глибини.

Рівень вежі (level) також інтегрується у фінальну модель, дозволяючи інтерфейсу автоматично підбирати колір рамки, анімації або спеціальні ефекти відповідно до поточного стану. Завдяки цьому гравець отримує чіткий візуальний зворотний зв’язок щодо сили і важливості кожної одиниці.

Таким чином, TowerModelViewer виконує роль мосту між внутрішньою логікою гри та візуальним відображенням даних, об’єднуючи всі необхідні елементи в одну

цілісну структуру. Такий підхід значно спрощує подальшу підтримку та розвиток проєкту, а також дозволяє легко адаптувати інтерфейс до змін у балансі чи нових механіках без порушення базової архітектури.

Алгоритм обробки рівнів вежі

У структурі будь-якої системи апгрейду важливу роль відіграє коректне зберігання та обробка інформації про поточний і наступний рівень об’єкта. У проєкті ця задача вирішується за допомогою класу TowerLevelInfoResult, який виконує функцію агрегатора даних, необхідних для керування процесом покращення оборонних споруд. Даний клас інкапсулює інформацію про ціни, рівні та стан доступності апгрейду, забезпечуючи базу для логіки як внутрішньоігрової економіки, так і елементів взаємодії в UI.

Рисунок 3.7 — Рівні веж

Основна роль TowerLevelInfoResult — представити в одному об’єкті весь спектр параметрів, необхідних для коректного апгрейду: поточну вартість продажу (sellPrice), вартість оновлення до наступного рівня (updatePrice), префаб вежі наступного рівня (nextLevelTower), її поточний рівень (currentLevel), а також час побудови (buildTime). Усі ці параметри використовуються як у бойовій логіці, так і в

інтерфейсі: наприклад, для відображення іконки нової вежі, зміни таймерів чи обрахунку ігрової валюти під час апгрейду.

```
public class TowerLevelInfoResult
{
    public readonly int sellPrice;
    public readonly int updatePrice;
    public readonly GameObject nextLevelTower;
    public readonly int currentLevel;
    public readonly float buildTime;

    public bool HasNextLevel => nextLevelTower;
}
```

Рисунок 3.8 — Клас TowerLevelInfoResult

Однією з найбільш функціонально важливих складових класу є властивість HasNextLevel, яка використовується для перевірки можливості подальшого покращення поточної вежі. У контексті ігрового інтерфейсу саме це логічне значення визначає, чи буде активована кнопка оновлення, чи буде вона недоступною (наприклад, у разі досягнення максимального рівня розвитку об’єкта). Також ця перевірка дозволяє уникнути помилок на логічному рівні — спроб запуску апгрейду без наявного наступного рівня.

Наявність чітко розмежованих значень у межах одного об'єкта дозволяє не тільки швидко отримувати необхідні дані, а й забезпечує високу модульність: замість передавання розрізнених параметрів між модулями, можна працювати з єдиною сутністю, яка містить всю актуальну інформацію. Такий підхід значно спрощує процес оновлення даних, тестування, налагодження і введення додаткових механік, як-от багатогілкові лінії розвитку, затримки будівництва чи адаптивні апгрейди залежно від типу ворогів.

Крім того, завдяки наявності префабу наступного рівня (nextLevelTower) гра має змогу динамічно підмінювати ігрові об'єкти без складної логіки вручну прописаних сценаріїв. Це дозволяє реалізовувати ефекти заміни моделі вежі під час апгрейду, запуск нових анімацій або зміни бойових характеристик автоматично, просто через заміну одного посилання.

Рисунок 3.9 — Різноманіття ворогів

У підсумку, TowerLevelInfoResult виступає важливим інструментом в архітектурі гри, об’єднуючи облік ресурсів, логіку розвитку веж і перевірку граничних умов в одному компактному і надійному об’єкті. Завдяки цьому ігрова система залишається стійкою до помилок, адаптованою до змін і зручною для розширення.

У сукупності наведені алгоритми формують ядро логіки гри. Вони охоплюють усі основні напрямки функціонування — від генерації ворогів, обліку їх параметрів і циклічного спавну, до побудови моделей для UI-відображення, обробки апгрейдів, перевірки можливостей розвитку і виконання внутрішньоігрових обчислень. Завдяки модульному підходу всі ці компоненти легко масштабуються, переносяться між сценами, повторно використовуються та тестуються. Така структура архітектури дозволяє гнучко керувати змінами у проєкті, швидко додавати нові функції або змінювати механіку без повного переписування коду. Це особливо важливо у контексті розробки ігор, де часто необхідно балансувати між швидкістю реалізації, гнучкістю і стабільністю виконання.

3.2 Розробка інтефрейсу користувача

Інтерфейс користувача є критичним елементом будь-якої гри, особливо у жанрі tower defense, де гравець постійно приймає рішення в реальному часі. У межах дипломного проєкту розроблено простий, але функціональний UI, який забезпечує інтуїтивну взаємодію з грою. Основними принципами при проєктуванні інтерфейсу були: читабельність, логічне групування елементів, мінімалізм та адаптивність до формату двовимірної гри з ізометричним виглядом.

Головне меню

На першому скріншоті зображено стартовий екран гри. У правій частині розміщено назву гри — Swamp tower, що є частиною фірмового стилю. Нижче знаходяться дві основні кнопки: Start та Exit, які запускають гру або закривають застосунок відповідно.

Рисунок 3.10 – Головне меню

Візуально кнопки чітко відділені, мають контрастний фон, який виділяє їх на загальному синьому тлі. Ліва частина екрану заповнена вертикально вирівняною графікою — зображеннями веж у кількох варіантах, що формує асоціацію із жанром

гри. У правому нижньому куті розміщено номер версії: 0.1.0+alpha — що вказує на статус розробки гри та забезпечує базову інформацію для тестувальника чи користувача.

Ігрова сцена (будівництво)

На другому скріншоті зображено основну ігрову сцену з відкритим магазином веж. У центрі розміщено ігрову карту — зелена зона з розміткою для розміщення веж, виконана в псевдоізометричній проекції. Угорі ліворуч присутній лічильник життя (Life: 8) з прогрес-баром поточної хвилі ворогів (1 / 4), що дозволяє гравцеві орієнтуватись у поточному стані гри. Внизу розміщено лічильник ресурсів (10), які витрачаються на будівництво та покращення веж.

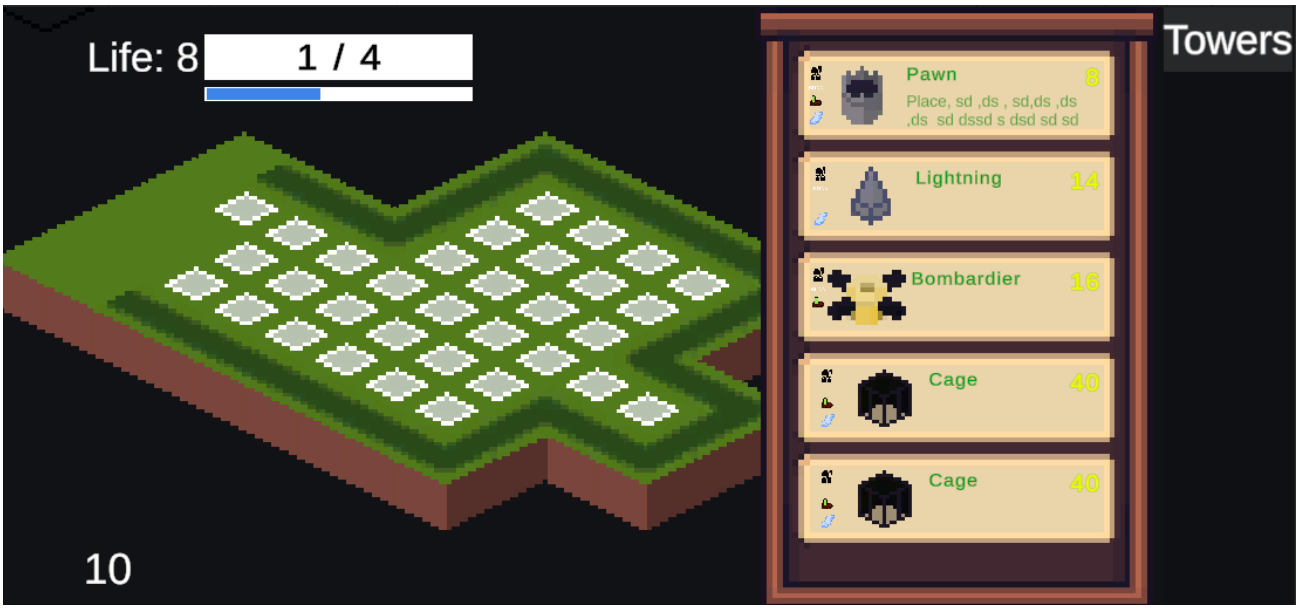

Рисунок 3.11 – Початковий стан гри

Праворуч реалізовано панель вибору веж. Кожна вежа представлена у вигляді карточки: зображення (спрайт), назва (Pawn, Lightning, Bombardier тощо), опис (декоративний), вартість, а також іконки характеристик. Вежі впорядковані вертикально, мають рамки, які чітко відділяють їх одну від одної, а текст відображено

з контрастом до фону. Це дозволяє гравцю швидко оцінити характеристики та обрати потрібну одиницю.

Третій скріншот демонструє бойову фазу. Карта збережена, однак деякі плитки зайняті побудованими вежами. На полі присутні юніти ворогів у вигляді рухомих об'єктів. Ліворуч відображається актуальна кількість життів (Life: 7) та поточна хвиля (3 / 4), прогрес якої відзначено жовтим індикатором. Нижче — оновлений баланс ресурсів (13), що відображає динаміку надходження коштів за знищення ворогів. Панель праворуч (Towers) залишилася незмінною, що підтримує сталість інтерфейсу.

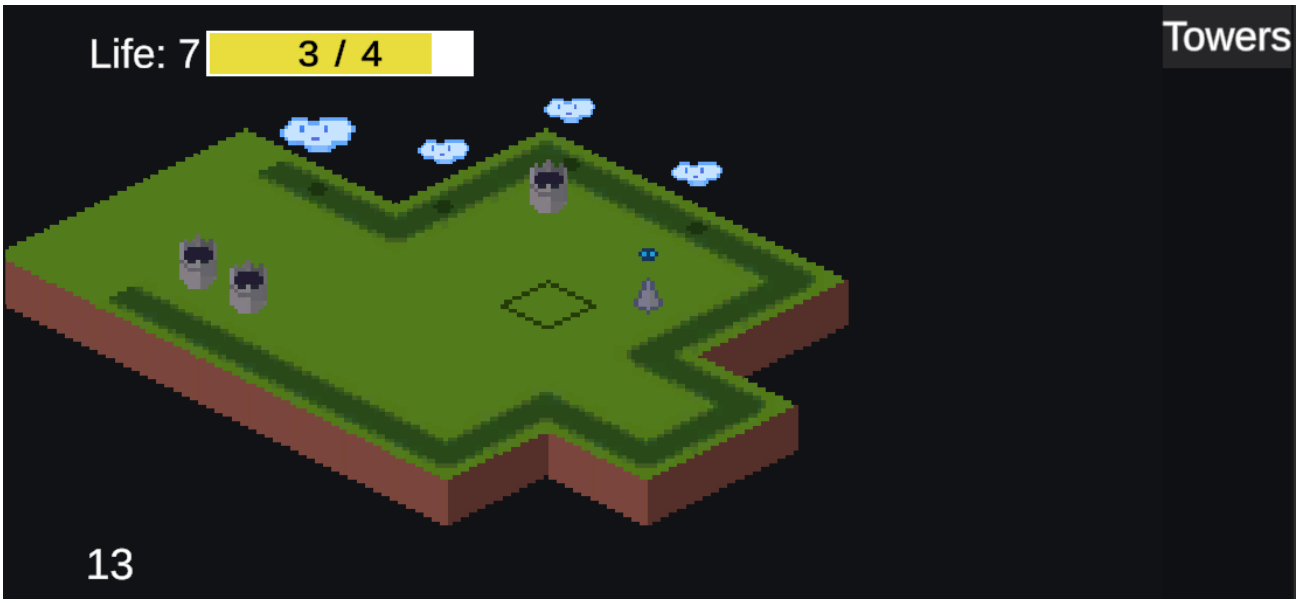

Рисунок 3.12 – Ігровий процес

Інтерфейс гри загалом є статичним у структурі, але динамічним у змісті — значення змінюються відповідно до стану гри, що дозволяє гравцеві адаптувати свої рішення до контексту.

3.3 Розробка ігрових механік

Основні ігрові механіки реалізовані через взаємодію окремих компонентів, пов’язаних із ворогами, вежами, хвилями, стрільбою та економікою. У цьому розділі

розглянуто реалізацію ключових механік, які формують логіку жанру tower defense: рух ворогів, втрата життя, побудова й апгрейд веж, атака, отримання ресурсів, хвилі та поразка.

Рух ворогів і проходження маршруту

Кожен ворог має маршрут, який проходить від точки входу до бази гравця. Рух реалізується за допомогою корутин або оновлення координат через Update().

```
    void Update()
    {
        if (points == null || points.Length == 0)
            return;

        Vector3 target = points[currentPointIndex];
        transform.position = Vector3.MoveTowards(transform.position, target,
speed * Time.deltaTime);

        if (Vector3.Distance(transform.position, target) < 0.1f)
        {
            currentPointIndex++;
            if (currentPointIndex >= points.Length)
            {
                ReachEnd();
            }
        }
    }
```

Рисунок 3.13 — Функція Update

У цьому коді points — набір координат шляху. Коли ворог досягає останньої точки, викликається метод `ReachEnd()`, який зменшує життя гравця:

```
void ReachEnd()
{
    GameManager.Instance.LoseLife(damage);
    Destroy(gameObject);
}
```

Рисунок 3.14 — Функція ReachEnd

Механіка втрати життя

Гравець має фіксовану кількість життів. При прориві ворога зменшується лічильник:

```
public void LoseLife(int amount)
{
    lives -= amount;
    if (lives <= 0)
    {
        GameOver();
    }
}
```

Рисунок 3.15 — Функція LoseLife

Побудова та розміщення вежі

Коли гравець клікає на вільну плитку, викликається метод розміщення вежі:

public void PlaceTower(Tower prefab, Vector3 position)

```
{
    if (money >= prefab.cost)
    {
        Instantiate(prefab, position, Quaternion.identity);
        money -= prefab.cost;
    }
}
public void PlaceTower(Tower prefab, Vector3 position)
```

```
    {
        if (money >= prefab.cost)
        {
            Instantiate(prefab, position, Quaternion.identity);
            money -= prefab.cost;
        }
    }
```

Рисунок 3.16 — Функція PlaceTower

Атака вежі

Кожна вежа самостійно визначає ворогів у радіусі і атакує їх із заданим інтервалом:

```
    void Update()
    {
        timer -= Time.deltaTime;
        if (timer <= 0f)
        {
            Enemy target = FindTarget();
            if (target != null)
            {
                Shoot(target);
                timer = cooldown;
            }
        }
    }
```

Рисунок 3.17 — Функція Update

Функція FindTarget() повертає найближчого ворога в радіусі дії:

```
    Enemy FindTarget()
    {
        Collider2D[] hits = Physics2D.OverlapCircleAll(transform.position,
radius);
        foreach (var hit in hits)
        {
```

```
            Enemy enemy = hit.GetComponent<Enemy>();
            if (enemy != null)
                return enemy;
        }
        return null;
    }
```

Рисунок 3.18 — Функція FindTarget

А сама стрільба виконується методом:

```
    void Shoot(Enemy target)
    {
        Bullet bullet = Instantiate(bulletPrefab, transform.position,
Quaternion.identity);
        bullet.SetTarget(target);
    }
```

Рисунок 3.19 — Функція Shoot

Апгрейд веж

Кожна вежа має кілька рівнів. При оновленні вона замінюється на новий префаб з іншими параметрами:

```
    public void UpgradeTower(Tower currentTower)
    {
        var upgrade =
towerInfoProvider.GetTowerFindResultByTower(currentTower);
        var next = upgrade?.NextItem;
        if (next != null && money >= next.buyPrice)
        {
            money -= next.buyPrice;
            Vector3 pos = currentTower.transform.position;
            Destroy(currentTower.gameObject);
            Instantiate(next.tower, pos, Quaternion.identity);
        }
```

Рисунок 3.20 — Функція UpgradeTower

Рисунок 3.21 — Апгрейд вежі

Під час гри відбувається поетапна генерація ворогів. Це реалізовано за допомогою перераховувача SpawnEnumerable, який у кожному кадрі повертає наступного ворога:

```
    IEnumerator SpawnWave()
    {
        foreach (var enemyPrefab in spawnEnumerable)
        {
            Instantiate(enemyPrefab,                    spawnPoint.position,
Quaternion.identity);
            yield return new WaitForSeconds(0.5f);
        }
    }
```

Рисунок 3.22 — Функція SpawnWave

Цей механізм дозволяє масштабувати кількість ворогів, змінювати їх тип, параметри тощо.

Завершення рівня та перемога

Гра завершується у двох випадках: коли життя зведено до нуля (поразка) або всі хвилі знищено:

```
void CheckWinCondition()
{
    if (spawnedAll && GameObject.FindObjectsOfType<Enemy>().Length == 0)
    {
        Victory();
    }
```

Рисунок 3.23 — Функція CheckWinCondition

3.4 Тестування

З метою перевірки стабільності, цілісності та відповідності функціональних компонентів гри було проведено комплексне тестування всіх основних механік. До тестування залучено функціональні, геймплейні та UI-складові, включно з поведінкою веж, ворогів, хвиль, взаємодією з інтерфейсом, обробкою помилок і загальною продуктивністю. В таблиці нижче наведено перелік умов тестування, очікувану поведінку та фактичний результат у рамках кожного тест-кейсу:

Таблиця 3.1 Тестування ігрового додатку

№	Умова тестування	Очікувана поведінка	Результат
1	Натискання кнопки Start у головному меню	Ініціюється перехід від меню до основної ігрової сцени, завантажуються всі ресурси	Успішно
2	Натискання кнопки Exit у меню	Програма завершує виконання, гра коректно закривається без помилок	Успішно
3	Запуск гри	Виводиться стартове меню з логотипом гри, кнопками управління та версією додатку	Успішно

Продовження таблиці 3.1

4	Клік по активній (вільній) плитці	Вежа розміщується на сцені, баланс монет зменшується на відповідну суму	Успішно
5	Вартість вежі перевищує кількість монет	Кнопка купівлі вежі деактивується, попередження не дозволяє розмістити вежу	Успішно
6	Ворог заходить у зону досяжності вежі	Вежа починає стріляти, спрацьовує механізм атаки, ворог втрачає частину здоров’я	Успішно
7	Здоров’я ворога зменшується до нуля	Ворог анімується як знищений, зникає з карти, гравцеві нараховуються монети	Успішно
8	Початок гри та зміна етапів	Індикатор кількості життів гравця з’являється та оновлюється відповідно до подій	Успішно
9	Ворог досягає фінальної точки маршруту	Значення життів гравця зменшується на величину, яка відповідає ворогу	Успішно

Продовження таблиці 3.1

10	Життів = 0	Гра завершується поразкою, на екран виводиться фінальне повідомлення	Успішно
11	Завершення всіх хвиль	Якщо жоден ворог не дійшов до кінця, виводиться екран перемоги	Успішно
12	Користувач натискає на кнопку покращення вежі	Вежа оновлюється до наступного рівня, зростають її атака, дальність або швидкість	Успішно
13	Знищення ворога	Гравець отримує винагороду у вигляді монет, баланс змінюється відповідно	Успішно
14	Початок нової хвилі	Вороги починають з’являтись групами, дотримуючись встановленого таймінгу	Успішно
15	Перехід між хвилями	Змінюється інтерфейс шкали хвилі, вороги змінюють тип та характеристики	Успішно

Продовження таблиці 3.1

16	Спавн кількох ворогів	Кожен з них обирає маршрут, рухається з врахуванням координат шляху без колізій	Успішно
17	На екрані >10 ворогів	Кожна вежа цілеспрямовано обирає найближчого або найнебезпечнішого ворога	Успішно
18	Вороги отримують шкоду	Анімації влучань, сплески ефектів, звукові супроводи відповідають дії вежі	Успішно
19	Завантаження рівня	Всі наявні вежі візуалізуються з їх параметрами, текстовими підказками	Успішно
20	4 і більше хвиль, понад 50 об'єктів одночасно	Гра підтримує стабільний FPS > 50 кадрів, не відбувається зависань	Успішно
21	Гравець повторно натискає Start після програшу	Відбувається повний рестарт сцени, обнуляються лічильники, об’єкти заново створюються	Успішно

Продовження таблиці 3.1

22	Пауза гри	При активації кнопки “Pause” — зупиняється логіка руху та атак	Успішно
23	Вимкнення/ввімкнення звуку	Аудіо вимикається/відновлюється відповідно до статусу кнопки	Успішно
24	Тести після оновлення веж	Параметри (шкода, дальність, таймер) реально зростають після апгрейду	Успішно
25	Ворожі хвилі з різним інтервалом	Динаміка хвиль змінюється коректно, інтервал між появами витримується	Успішно
26	Візуальний ефект вибуху	При знищенні ворога виводиться відповідний ефект, не впливає на продуктивність	Успішно
27	Мобільність інтерфейсу	Кнопки масштабуються відповідно до екрану, не перекривають інші елементи	Успішно
28	Обмеження кількості веж	Якщо досягнуто максимум — спроба додати нову вежу не проходить	Успішно

Продовження таблиці 3.1

29	Примусове завершення програми	Наступний запуск відбувається без збоїв, з повною ініціалізацією	Успішно
30	Скидання прогресу	Кнопка “Reset” очищує поле, монети, життя — все повертається до значень за замовчуванням	Успішно
31	Ворог отримує кілька атак одночасно	Усі дії обробляються вірно, сумарна шкода підраховується правильно	Успішно
32	Обмежений екран	На екранах з низькою роздільною здатністю елементи не виходять за межі	Успішно
33	Підвищене навантаження (100+ об'єктів)	Система частинок, скрипти та апдейти не викликають зависань	Успішно
34	Вежі з різними типами атаки	Кожен тип веде себе по-різному (сплеск, вогонь, заморозка) — поведінка коректна	Успішно
35	Тест ворогів з різною швидкістю	Візуальна і логічна швидкість ворогів відповідає заданим параметрам	Успішно

Продовження таблиці 3.1

36	Взаємодія з UI під час хвилі	Користувач може без помилок оновлювати вежі або ставити нові	Успішно
37	Аудіо супровід подій	Звукові ефекти відповідають діям: постріли, удари, смерть ворога	Успішно
38	Вежі не стріляють у трупи	Мертвих ворогів не враховано як цілі — перевірено логіку isAlive	Успішно
39	Збереження частоти оновлення	При тривалому геймплеї (15+ хв) продуктивність залишається стабільною	Успішно
40	Облік статистики гравця	Сума вбитих ворогів, витрачених монет, часу відображається в меню результатів	Успішно

Рисунок 3.24 — Кінець рівня

Усі тести завершено успішно, критичних помилок виявлено не було. Поведінка гри відповідає очікуваній логіці, UI оновлюється коректно, перфоманс стабільний навіть при підвищеному навантаженні. Система готова до подальшого розвитку та масштабування.

Висновки до розділу 3

У третьому розділі було реалізовано повноцінний програмний прототип гри жанру tower defense, що поєднує основні елементи геймплейної логіки, графічного інтерфейсу та внутрішньої взаємодії об’єктів. Реалізація алгоритмів поведінки ворогів, логіки спавну, атак веж, обробки рівнів і взаємодії з користувачем здійснювалась на основі раніше спроєктованої архітектури. Завдяки чіткій структурі класів і розділенню обов’язків між компонентами вдалося досягти високого рівня організації коду, що значно полегшило процес тестування та налагодження.

Інтерфейс користувача, реалізований у межах проєкту, повністю відповідає сучасним вимогам до юзабіліті, забезпечуючи гравцеві інтуїтивну навігацію, зручне управління вежами та чітку візуалізацію стану гри. Особливу увагу приділено адаптивності елементів, коректній відображуваності HUD та анімаціям, що покращують загальне сприйняття ігрового процесу. Інтерфейс не лише підтримує

основну ігрову логіку, а й створює відповідну атмосферу, необхідну для жанру стратегічного захисту.

Проведене тестування засвідчило високу стабільність роботи застосунку на цільовій платформі, відсутність критичних помилок та відповідність очікуваній логіці взаємодії. Була перевірена як функціональність окремих механік, так і загальна продуктивність, включно з продуктивністю на різних етапах геймплею, навантаженням на систему та поведінкою у випадку граничних сценаріїв. Отримані результати дозволяють стверджувати про повну технічну готовність гри до подальшого вдосконалення, розширення функціоналу або публікації на обраній платформі.

ВИСНОВКИ

У процесі виконання дипломної роботи було реалізовано повноцінний прототип 2D-ігрового застосунку жанру tower defense, який поєднує в собі функціональність, стабільність та потенціал до масштабування. Робота охопила всі етапи розробки: від аналізу предметної області та вибору оптимальних технологій до безпосередньої реалізації геймплейної логіки та комплексного тестування.

В рамках дослідження було обґрунтовано вибір рушія Unity3D як основної технологічної платформи для реалізації проєкту. Unity забезпечив повну підтримку двовимірної графіки, гнучку систему компонентів, розширювану архітектуру та зручні інструменти для редагування рівнів. Обрана мова програмування — C# — надала необхідний рівень абстракції для побудови складної логіки та взаємодії між об'єктами сцени. Середовище розробки Visual Studio дозволило ефективно управляти проєктом, відлагоджувати код і забезпечити чистоту архітектури.

У процесі реалізації проєкту було створено систему управління хвилями ворогів, інтелект веж, які самостійно прицілюються та здійснюють атаки, економічну модель із динамічним балансом ресурсів, UI-інтерфейс з інформаційними панелями, магазин веж і підказки. Особливу увагу приділено архітектурі — ігрові сутності винесені в окремі класи, реалізовано системи агрегації даних для UI, налаштовувані редакторські компоненти та зручні для масштабування структури.

Проведене тестування підтвердило працездатність усіх основних механік: стрільба, апгрейд, хвилі, втрата життя, перемога, поразка, економіка, стабільність перфомансу. Ігровий процес реалізовано таким чином, щоб забезпечити плавну динаміку та логічний розвиток складності від хвилі до хвилі.

Таким чином, мета дипломної роботи досягнута повністю. Результатом є технічно завершена, функціональна 2D-гра, розроблена з урахуванням сучасних підходів до ігрової інженерії. Структура коду дозволяє легко модифікувати та розширювати проєкт, що відкриває перспективи для подальшого розвитку гри:

розширення типів веж, введення нових рівнів, реалізація мобільної версії чи інтеграція онлайн-функціоналу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Salen K., Zimmerman E. Rules of Play: Game Design Fundamentals. Cambridge, MA: MIT Press, 2003. 688 p.
2. Fullerton T. Game Design Workshop: A Playcentric Approach to Creating Innovative Games. 4th ed. Boca Raton: CRC Press, 2018. 576 p.
3. Rollings A., Adams E. Fundamentals of Game Design. 3rd ed. Berkeley: New Riders, 2014. 552 p.
4. Novak J. Game Development Essentials: An Introduction. 3rd ed. Clifton Park: Delmar Cengage Learning, 2011. 512 p.
5. McGonigal J. Reality Is Broken: Why Games Make Us Better and How They Can Change the World. New York: Penguin Press, 2011. 416 p.
6. Unity Technologies. Unity User Manual. URL: https://docs.unity3d.com/Manual/index.html (дата звернення: 13.04.2025).
7. Schell J. The Art of Game Design: A Book of Lenses. 3rd ed. Boca Raton: CRC Press, 2019. 600 p.
8. Unity Learn Platform. Unity Technologies. URL: https://learn.unity.com (дата звернення: 13.04.2025).
9. Rogers S. Level Up! The Guide to Great Video Game Design. 2nd ed. Indianapolis: Wiley, 2014. 552 p.
10. Kerr A. Global Games: Production, Circulation and Policy in the Networked Era. New York: Routledge, 2017. 212 p.
11. Gee J. P. What Video Games Have to Teach Us About Learning and Literacy. 2nd ed. New York: Palgrave Macmillan, 2007. 256 p.
12. IT-індустрія України: стан, тенденції та перспективи. Аналітичний звіт IT Ukraine Association. URL: https://itukraine.org.ua (дата звернення: 13.04.2025).
13. Petrenko S. Ukrainian Game Dev: From Outsourcing to Indie Breakthroughs. Kyiv: GameHub, 2021. 98 p.

14. Juul J. A Casual Revolution: Reinventing Video Games and Their Players. Cambridge, MA: MIT Press, 2010. 256 p.
15. Anthropy A. Rise of the Videogame Zinesters. New York: Seven Stories Press, 2012. 208 p.
16. Bogost I. How to Talk About Videogames. Minneapolis: University of Minnesota Press, 2015. 208 p.
17. Unity Technologies. 2D Game Kit. URL: https://unity.com/learn/tutorials/projects/2d-game-kit (дата звернення: 13.04.2025).
18. Wang C. Practical Shader Development for Unity and WebGL. Boca Raton: CRC Press, 2022. 320 p.
19. Totten C. An Architectural Approach to Level Design. Boca Raton: CRC Press, 2014. 348 p.
20. Brathwaite B., Schreiber I. Game Design Workshop. Burlington: Jones & Bartlett Learning, 2020. 336 p.
21. Unity Asset Store. URL: https://assetstore.unity.com (дата звернення: 13.04.2025).
22. Bycer J. Game Design Deep Dive: Roguelikes. Boca Raton: CRC Press, 2021. 322 p.
23. S. A. E. Campos, B. A. M. Morales and Á. A. V. Núñez, "Open-Source Game Engine & Framework for 2D Game Development," 2022 IEEE Engineering International Research Conference (EIRCON), Lima, Peru, 2022, pp. 1-4, doi: 10.1109/EIRCON56026.2022.9934816.
24. Y. Shi, "Designing and Comparing Time Rewind Mechanics in 2D Interactive Game," 2021 IEEE Conference on Games (CoG), Copenhagen, Denmark, 2021, pp. 1-6, doi: 10.1109/CoG52621.2021.9619030.
25. Hongxian Zhang and Sui Huang, "Research and application for combination between copied map and A* algorithm in 2D game," 2011 International Conference on Computer Science and Service System (CSSS), Nanjing, 2011, pp. 3405-3407, doi: 10.1109/CSSS.2011.5974837.

26. R. Sajina and T. Orehovacki, "User experience evaluation of 2D side-scrolling game developed using Overlap2D game editor and LibGDX game engine," 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 2018, pp. 1580-1585, doi: 10.23919/MIPRO.2018.8400284.
27. D. Todorović, S. Grujić, M. Vukajlović and Ž. Vinčić, "Developing a 2D game engine to simplify tile-based platformer development : Student paper," 2025 24th International Symposium INFOTEH-JAHORINA (INFOTEH), East Sarajevo, Bosnia and Herzegovina, 2025, pp. 1-6, doi: 10.1109/INFOTEH64129.2025.10959197.
28. W. Höhl, F. Kharvari and G. Klinker, "Wayfinding in Museums: A Cross-sectional Comparison Between 3D Serious Games and 2D Drawings as Tools for Participatory Design," 2020 IEEE Conference on Games (CoG), Osaka, Japan, 2020, pp. 592-595, doi: 10.1109/CoG47356.2020.9231921.
29. J. -Y. Wang, "Classification of Humans and Bots in Two Typical Two-player Computer Games," 2018 3rd International Conference on Computer and Communication Systems (ICCCS), Nagoya, Japan, 2018, pp. 502-505, doi: 10.1109/CCOMS.2018.8463277.
30. M. Wang and L. Zhu, "Designing and implementing an online card game based on Android 2D graphics," 2014 International Conference on Audio, Language and Image Processing, Shanghai, China, 2014, pp. 817-821, doi: 10.1109/ICALIP.2014.7009908.
31. J. O. Verastegui-Santiago et al., "Use of 2D/3D Video Games in Digital Platforms for Basic Education: A Technological and Systematic Review," 2023 IEEE Colombian Caribbean Conference (C3), Barranquilla, Colombia, 2023, pp. 1-6, doi: 10.1109/C358072.2023.10436294.
32. T. Patino and C. Ramos, "Program with Ixquic: How to Learn Object-Oriented Programming with a Game," 2015 7th International Conference on Games and Virtual

Worlds for Serious Applications (VS-Games), Skovde, Sweden, 2015, pp. 1-2, doi: 10.1109/VS-GAMES.2015.7295781.

33. J. D. M. Esteban et al., "OCLEAN: An Endless 2D Mobile Game Focused on The Awareness of Cleaning Marine Plastic Waste," 2022 2nd International Conference in Information and Computing Research (iCORE), Cebu, Philippines, 2022, pp. 139-143, doi: 10.1109/iCORE58172.2022.00045.
34. H. R. Khairuddin, A. S. Malik, W. Mumtaz, N. Kamel and L. Xia, "Analysis of EEG signals regularity in adults during video game play in 2D and 3D," 2013 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), Osaka, Japan, 2013, pp. 2064-2067, doi: 10.1109/EMBC.2013.6609938.

ДОДАТОК А

Лістинг програмного коду

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

public class MoneyText : MonoBehaviour
{
    private TextMeshProUGUI _textMeshPro;
    private ShopManager _shopManager;

    private int _lastMoney;

    void Start()
    {
        _shopManager = ShopManager.Instance;
        _textMeshPro = GetComponent<TextMeshProUGUI>();
        UpdateText();
    }

    private void UpdateText()
    {
        _textMeshPro.text = _shopManager.money.ToString();
    }
```

```
    void Update()
    {
        if (_lastMoney != _shopManager.money)
        {
            UpdateText();
        }

        _lastMoney = _shopManager.money;
    }
}

//Монстр: Швидкість, Життя, Максимальне життя, Атака, Монети, Тип

using System;
using UnityEngine;
using UnityEngine.Serialization;

[Serializable]
public class EnemyInfo
{
    [Min(0)]
    public float speed = 1;

    [Min(0)]
    public int health;

    [Min(1)]
    public int maxHealth;
```

```
    [Min(0)]
    public int damage = 1;

    [Min(0)]
    public int money = 1;

    public EnemyType enemyType;

    public float RelativeHealth => health / (float)maxHealth;
}
using System;

[Flags]
public enum EnemyType
{
    Ground = 0b1,
    Fly = 0b10,
    Universal = Ground | Fly,
    Enemy = 0b100,
    Boss = 0b1000,
    EnemyBoss = Enemy | Boss
}
public static class EnemyTypeMethods
{
    public static bool IsAttack(this EnemyType enemy, EnemyType bullet)
    {
        return (enemy & EnemyType.EnemyBoss & bullet) != 0
```

```
            && (enemy & EnemyType.Universal & bullet) != 0;
    }

    public static bool IsValid(this EnemyType type)
    {
        return (type & (EnemyType.Boss | EnemyType.Enemy)) != 0
            && (type & (EnemyType.Fly | EnemyType.Ground)) != 0;
    }
}
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class SpawnEnumerable : IEnumerable<GameObject>
{
    private readonly Level _level;
    public readonly int CountWaves;
    public readonly int CountMonsters;

    public int CurrentPositionWave { get; private set; }
    public Wave CurrentWave { get; private set; }
    public float SpawnTime { get; private set; }
    public int CountSpawnMonsters { get; private set; } = 0;

    public SpawnEnumerable(Level level)
    {
        _level = level;
```

```
        CurrentPositionWave = 0;
        CountWaves = _level.waves.Length;

        CountMonsters = _level.waves
            .Select(
                w => w.subWaves
                    .Select(s => s.count)
                    .DefaultIfEmpty()
                    .Sum()
            )
            .DefaultIfEmpty()
            .Sum();

    }


    public IEnumerator<GameObject> GetEnumerator()
    {
        for (var levelLength = _level.waves.Length; CurrentPositionWave < levelLength; CurrentPositionWave++)
        {
            CurrentWave = _level.waves[CurrentPositionWave];
            SpawnTime = CurrentWave.spawnTime;
            foreach (var subWave in CurrentWave.subWaves)
            {
                for (int i = 0; i < subWave.count; i++)
                {
                    yield return subWave.monster;
```

```
                    CountSpawnMonsters++;
                }
            }
        }

        CurrentPositionWave--;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    public float MonsterFillPercentage => CountSpawnMonsters / (float)CountMonsters;
}
using System.Linq;
using JetBrains.Annotations;
using UnityEngine;
using UnityEditor;
using System.Collections.Generic;
using System;

public class TowerInfoProvider
{
    private readonly TowerInfoCollection _towerLevelInfoCollection;
```

```
        private TowerInfo[] TowerLevelInfos =>
_towerLevelInfoCollection.towerLevelInfos;

        public TowerInfoProvider(TowerInfoCollection collection)
        {
            Debug.Assert(collection != null);
            _towerLevelInfoCollection = collection;
        }

        [CanBeNull]
        public TowerLevelInfoResult GetTowerLevelInfoResult(GameObject
gameObject)
        {
            var tower = GetTower(gameObject);
            return GetTowerLevelInfoResult(tower);
        }

        [CanBeNull]
        public TowerLevelInfoResult GetTowerLevelInfoResult(Tower tower)
        {
            var result = GetTowerFindResultByTower(tower);

            if (result == null)
                return null;

            return ToTowerLevelInfoResult(result);
        }
```

```
    private static TowerLevelInfoResult ToTowerLevelInfoResult(TowerFindResult
result)
    {
        var towerLevel = result.CurrentItem;
        var nextTowerLevel = result.NextItem ?? TowerUpgradeInfo.Empty;
        var levelValue = result.indexOfItem + 1;

        return ToTowerLevelInfoResult(towerLevel, nextTowerLevel, levelValue);
    }

    private static TowerLevelInfoResult ToTowerLevelInfoResult(
        TowerUpgradeInfo towerLevel,
        TowerUpgradeInfo nextTowerLevel,
        int levelValue
    )
    {
        return new TowerLevelInfoResult(
            towerLevel.salePrice,
            nextTowerLevel.buyPrice,
            nextTowerLevel.tower,
            levelValue,
            nextTowerLevel.buildTime
        );
    }

    public GameObject GetFirstLevelTower(GameObject gameObject)
```

```
{
    Tower tower = GetTower(gameObject);
    return GetFirstLevelTower(tower);
}

public GameObject GetFirstLevelTower(Tower tower)
{
    var result = GetTowerFindResultByTower(tower);

    return result?.source.infoItems[0].tower;
}



private static Tower GetTower(GameObject gameObject)
{
    var tower = gameObject.GetComponentInChildren<Tower>();

    if(tower == null)
        throw new Exception($"'{gameObject.name} isn't Tower'");

    return tower;
}



public ShopItem GetShopItemForTower(GameObject gameObject)
{
    var tower = GetTower(gameObject);
    return GetShopItemForTower(tower);
```

```
}

public ShopItem GetShopItemForTower(Tower tower)
{
    return ToShopItem(GetTowerFindResultByTower(tower).source);
}

public TowerFindResult GetTowerFindResultByTower(Tower tower)
{
    foreach (var towerLevelInfo in TowerLevelInfos)
    {
        var towerLevels = towerLevelInfo.infoItems;

        for (int j = 0, length = towerLevels.Length; j < length; j++)
        {
            var towerLevel = towerLevels[j];
            var prefab = towerLevel.tower;

            if (tower.prefab == prefab)
                return new TowerFindResult(towerLevelInfo, j);
        }
    }

    return null;
}

public TowerModelViewer GetTowerModelViewer(GameObject gameObject)
{
```

```
        var tower = GetTower(gameObject);

        return GetTowerModelViewer(tower);
    }

    public TowerModelViewer GetTowerModelViewer(Tower tower)
    {
        var result = GetTowerFindResultByTower(tower);

        return ToTowerModelViewer(tower, result);
    }

    public static TowerModelViewer ToTowerModelViewer(Tower tower, TowerFindResult result)
    {
        var towerLevelInfoItem = result.CurrentItem;
        var shopItem = result.source.shopItem;
        var level = result.indexOfItem + 1;
        var nextTowerLevelInfoItem = result.NextItem
            ?? TowerUpgradeInfo.Empty;

        return ToTowerModelViewer(tower, towerLevelInfoItem, shopItem, level, nextTowerLevelInfoItem);
    }

    public static TowerModelViewer ToTowerModelViewer(
        Tower tower,
        TowerUpgradeInfo towerLevelInfoItem,
```

```
    TowerShopItem shopItem,
    int level,
    TowerUpgradeInfo nextTowerLevelInfoItem
    )
{
    var model = new TowerModelViewer(
        tower.info.target,
        tower.info.damage,
        tower.Radius,
        tower.info.cooldown,
        tower.info.countMonstersDamage,
        towerLevelInfoItem.salePrice,
        nextTowerLevelInfoItem.buyPrice,
        shopItem.icon,
        shopItem.description,
        shopItem.name,
        level
    );

    return model;
}

public IEnumerable<ShopItem> ShopItems =>
    _towerLevelInfoCollection
        .towerLevelInfos
        .Select(ToShopItem);

public static ShopItem ToShopItem(TowerInfo info)
```

```
        {
            return ToShopItem(info.shopItem, info.infoItems[0]);
        }

        public static ShopItem ToShopItem(TowerShopItem shopItem, TowerUpgradeInfo
updateInfo)
        {
            return new ShopItem()
            {
                icon = shopItem.icon,
                name = shopItem.name,
                targetType = shopItem.targetType,
                price = updateInfo.buyPrice,
                description = shopItem.description,
                tower = updateInfo.tower,
                buildTime = updateInfo.buildTime
            };
        }



    }
    using UnityEngine;

    public class TowerLevelInfoResult
    {
        public readonly int sellPrice;
        public readonly int updatePrice;
        public readonly GameObject nextLevelTower;
```

```
    public readonly int currentLevel;
    public readonly float buildTime;

    public TowerLevelInfoResult(
        int sellPrice,
        int updatePrice,
        GameObject nextLevelTower,
        int currentLevel,
        float buildTime
    )
    {
        this.sellPrice = sellPrice;
        this.updatePrice = updatePrice;
        this.nextLevelTower = nextLevelTower;
        this.currentLevel = currentLevel;
        this.buildTime = buildTime;
    }

    public bool HasNextLevel => nextLevelTower;
}
```