Вінницький національний технічний університет
(повне найменування вищого навчального закладу)

Факультет інформаційних технологій та комп'ютерної інженерії
(повне найменування інституту)

Кафедра обчислювальної техніки
(повна назва кафедри)

Пояснювальна записка

до магістерської кваліфікаційної роботи
магістр
(освітньо-кваліфікаційний рівень)

на тему: Мультимедійне ігрове застосування на основі об'єктно-орієнтованого підходу на базі платфори unity3d

Виконав: студент 2 курсу, групи КІ —19м
спеціальності:
123 «Комп'ютерна інженерія»
(шифр і назва напряму підготовки)

Омельченко Сергій Сергійович
(прізвище та ініціали)

Керівник: к.т.н., доц. Савицька Л.А.
(прізвище та ініціали)

м. Вінниця — 2020 рік

Вінницький національний технічний університет
(повне найменування вищого навчального закладу)

Факультет Інформаційних технологій та комп'ютерної інженерії
Кафедра Обчислювальної техніки
Освітньо-кваліфікаційний рівень магістр
Спеціальність 123 «Комп'ютерна інженерія»
(шифр і назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри_____

д.т.н., професор Азаров О. Д.___

"____" _____20___року

З А В Д А Н Н Я

на магістерську кваліфікаційну роботу

Омельченку Сергію Сергійовичу

(прізвище, ім'я, по батькові)

1 Тема магістерської кваліфікаційної роботи: Мультимедійне ігрове застосування на основі об'єктно- орієнтованого підходу на базі платфори unity3d
керівник магістерської кваліфікаційної роботи: Савицька Л.А., к. т. н., доцент кафедри ОТ.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від "___"_____20__року №___
2 Строк подання студентом роботи_____
3 Вихідні дані до роботи технічний опис програмного застосунку, мова програмування C#, середовище розробки Unity ;
4 Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Вступ. Аналіз методів і технологій розробки мультимедійних додатків. Аналіз існуючих аналогів. Варіантний вибір засобів для розробки мультимедійного додатку на базі Unity. Програмна реалізація мультимедійного програмного засобу. Тестування розробленого програмного забезпечення.
5 Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Вікна інтерфейсу користувача. Ігрові спрайти.

6 Консультанти розділів проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-3	Савицька Людмила Анатолівна, к. т. н., доцент	_____ дата	_____ дата
		_____ підпис	_____ підпис
4	Руда Лілія Петрівна к. е. н., доцент	_____ дата	_____ дата
		_____ підпис	_____ підпис

7 Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Пошук та огляд інформаційних джерел	07.09.20р.	виконано
2	Огляд і аналіз методів розробки мультимедійного додатку	20.09.20р.	виконано
3	Аналіз існуючих аналогів	01.10.20р.	виконано
4	Розробка та відлагодження програмного забезпечення	09.10.20р.	виконано
5	Економічна частина	18.11.20р.	виконано
6	Оформлення пояснювальної записки і презентації	20.11.20р.	виконано
7	Попередній захист	23.11.20р.	виконано

Студент _____ Омельченко С.С.

(підпис) (прізвище та ініціали)

Керівник роботи _____ Савицька Л.А.

(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дана магістерська кваліфікаційна робота присвячений розробці програмного забезпечення в сучасній ігровій індустрії.

У дипломній роботі виконаний аналіз найоптимальніших і найбільш зручних програмних засобів, інструментів для створення гри. Під час розробки програмного забезпечення був використаний об’єктно-орієнтований підхід і описана логіка гри.

Перевірка працездатності гри була протестована шляхом практичних тестувань кожного окремого компонента.

ANNOTATION

This master's thesis is devoted to software development in the modern gaming industry.

In the thesis the analysis of the most optimum and most convenient software, tools for game creation is executed. An object-oriented approach was used in software development and the logic of the game was described.

The game's performance was tested by practical testing of each individual component.

ЗМІСТ

ВСТУП

Мультимедійні ігри — це ігри, у яких гравець взаємодіє з віртуальністю побудованою комп’ютером і навчається біля нього. Комп’ютерні ігри є невід’ємними в сучасності. Вчені з Сінгапурського технологічного університету провели експеримент, який показав, що комп'ютерні відеоігри можуть розвинути здібності мозку. Адам Чі-Мінг Ой (Adam Chie-Ming Oei) і Майкл Дональд Паттерсон (Michael Donald Patterson) поставили експеримент на п'яти людях, які раніше не захоплюються відеоіграми [1]. У процесі дослідження впливу ігор на головний мозок, вчені запропонували учасникам експерименту кожен день запускати на своїх комп'ютерах, наприклад, «The Sims», «Battlefield», «Hidden Expedition», «Portal» і інші ігри.

Через місяць вчені провели повторний тест, який і показав, що у випробовуваних розвинулася уважність до деталей, вони навчилися стратегічно мислити і швидко приймати рішення, грунтуючись на великому масиві інформації. Також було відзначено, що розвиток тих чи інших здібностей безпосередньо залежить від жанру відеоігри. Тобто, ті, хто протягом місяця грав в action-ігри, стали краще й ефективніше контролювати власні цілі і застосовувати стратегічні ходи, щоб швидше домогтися їх, прийти до перемоги.

У зв’язку з такими відкриття стала набувати популярності розробка ігор в якості незалежних студій, навіть, як людина “оркестр”, яка відповідає за всі напрямки у розробці ігор. Нинішня тенденція, звісно, не дозволяє, за розумний проміжок часу, досягти належної якості кожного елементу ігор, як це відбувається у великих студіях при розробці крупних ігор.

В наслідок таких дій, зменшується якість та глибина нових ігор, що має негативний вплив на ігрову індустрію, в цілому. Задля того, щоб уникнути необхідності розбиратися у кожній ділянці розробки ігор, створюються спеціальні інструменти, які забирають специфічну роботу на себе роботу на себе.

Як один із прикладів - візуальне програмування, яке дозволяє створювати логічні елементи майже будь-якої складності у вигляді кінцевого автомату,

зібраного, з найменших, елементарних дії [2]. Такі інструменти існують для багатьох сфер розробки ігор, починаючи від графіки та закінчуючи музикою. Отже, розробка даного додатку є **актуальною**.

Метою даної магістерської роботи є зменшення операційних витрат на процес програмування логіки мультимеійної гри

Об'єктом — процес написання логіки мультимедійної гри об'єктно-орієнтованим методом.

Предметом — процес програмування логіки мультимедійної гри на базі об'єктно-орієнтованого програмування.

Для виконання поставленої у магістерської дипломній роботі мети необхідно виконати таке завдання:

1) провести аналіз сучасних методів та засобів розробки логіки програмування мультимедійних ігор, що дасть можливість обрати найбільш оптимальний інструмент для виконання робот;

2) визначити архітектуру програмного засобу, що дає можливість економії ресурсів та витрат на процеси програмування та перепрограмування логіки додатку;

3) реалізувати логіку і компоненти гри, що дає можливість використовувати гру, як готовий продукт;

4) розробити мультимедійну гру та виконати її тестування, що дає можливість локалізація можливих помилок під час програмування логіки гри.

1 СУЧАСНІ МЕТОДИ ТА ЗАСОБИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ЛОГІКИ ПРОГРАМУВАННЯ ТА ІНТЕРФЕЙСУ МУЛЬТИМЕДІЙНИХ ІГОР

1.1 Роль та місце мультимедійних ігор в сучасному сегменті мультимедійних розваг

Мультимедійні ігри — це комп’ютерна програма, яка в більшості випадків створена для розваг. Є безліч видів ігор, які в свою чергу діляться ще на декілька підтипів. Відеоігри пройшли довгий шлях від використання простого джойстика для керівництва Pac-Man у його місії знаходити їжу та уникати привидів. Це ілюструє рекламний ролик Southwest Airlines 2007 року, в якому двоє друзів грають у бейсбольну відеоігру на консолі, схожій на Nintendo Wii. Драматий друг каже другому кинути висоту - і він робить це, схвильовано спрацьовуючи своїм контролером у середину плазмового телевізора, який потім падає зі стіни. Обидва друзі вражено дивляться на розбитий плоский екран, коли оповідач запитує: "Хочете піти геть?"

Така сцена навряд чи мала місце в перші дні відеоігор, коли Атарі панував, а дії гри були зосереджені виключно на координації рук і очей. Крива навчання порівняно не існувала; гравці маневрували джойстиком, щоб стріляти в небо лініями прибульців, або вони повертали колесо на веслу, щоб грати у віртуальну гру в настільний теніс.

Але оскільки відеоігри ставали дедалі популярнішими, вони також ставали дедалі складнішими. Консолі регулярно модернізувались і розвивались, а ігри не відставали. Гравці зателефонували один одному з лазівками та порадами, як перевести Маріо та Луїджі на наступний рівень, і тепер вони обмінюються своїми трюками в ігрових блогах. Такі ігри, як The Legend of Zelda та Final Fantasy, створили альтернативні світи та хитромудрі сюжетні лінії, забезпечуючи багатогодинні епічні пригоди.

Багато відеоігор, яких давно критикували за те, що вони виводили дітей із заднього двору та в сидяче місце біля телевізора, повернулися до їхнього більш

простого походження і, роблячи це, зробили гравців більш активними. Повсякденні геймери, які могли швидко зрозуміти, як скласти шматочки головоломки в Тетрісі, тепер можуть так само легко зрозуміти, як "розмахувати" тенісною ракеткою за допомогою Wiimote. Відеоігри вже не є зручним козлом відпущення для проблем ожиріння в Америці; Wii Fit пропонує все: від йоги до боксу, а Dance Dance Revolution оцінює спалені калорії під час танцю гравців.

Відеоігри та мультимедійні ступені дозволяють критично вивчати ігри. Ця дисципліна зосереджена на теоріях та концепціях дизайну ігор та створення мультимедійного контенту на різних типах платформ. Як частина розважального мультимедійного поля, створення відеовмісту є найважливішим процесом у розробці відеоігор.

Відеоігри та мультимедіа об'єднують елементи графічного дизайну, інформатики та IT.Мультимедіа стосується використання різних форм носіїв, таких як графіка, звук, анімація та текст, для передачі інформації користувачам. Поняття, що означає сполучення звукових, відео, графічних, текстових і цифрових сигналів, а також нерухомих і рухомих образів і конструкцій. Мультимедіа пов’язані з галузями комунікації та журналістики і, здебільшого, засновані на впровадженні комп’ютерних технологій. Курси у відеоіграх та мультимедіа пропонують знання та відповідну інформацію на такі теми, як: тривимірна графіка, дизайн аналізу ігор, розважальні технології, ігрова анімація, програми віртуальної реальності.

Студенти розвивають навички програмування та зможуть розробляти концепції та рішення за допомогою спеціалізованого комп’ютерного програмного забезпечення. Майбутні дизайнери відеоігор використовують уяву та візуальну чутливість для визначення відповідних та застосовних принципів у розробці та реалізації ігор.

Випускники можуть знайти роботу в галузі дизайну відеоігор як розробники програмного забезпечення, дизайнери графіки та анімації, творці мультимедіа, спеціалісти з оптимізації мережі, спеціалісти з моделей процесів тощо.

1.2 Історія створення жанру Tower Defence в мультимедійних іграх

Tower Defence скорочено (TD) – один з найпопулярніших жанрів стратегічних відеоігор. Завдання гравця в іграх цього жанру обороняти свою головну будівлю, за допомогою захисних веж або персонажів. Протягом цілої гри на будівлю будуть насуватись хвилі ворогів, в яких головна мета її зруйнування:

Рисунок 1.1 — Шлях просування ворогів до головної будівлі

Зазвичай противники і захисні вежі різняться за характеристиками, тому ворогів значно більше чим веж і з кожним наступним разом їх стає все більше і більше. Після закінчення хвиль, гравець отримує гроші чи очки, завдяки яким він зможе використати для будівництва нових споруд, або для модернізації старих.

Стратегічна гра цілком присвячена розміщенню захисних веж, при цьому важливо враховувати їхні типи. Зазвичай монстри проходять через подібність лабіринту, що дає гравцеві можливість стратегічного розміщення веж (наприклад, гра TDMM), хоча також існують популярні ігри (лінійні TD), де використовуються прямі шляхи замість лабіринтів (наприклад, гра The Last Shelter).

Один з найбільш ранніх представників TD – аркадна гра Rampart, перенесена згодом на безліч платформ. Гра стала причиною створення ряду карт гри «StarCraft», які отримали назву Turret Defense, які в свою чергу надихнули на створення карт Sunken і Hero Defense (Warcraft III). Такі модифікації стали популярними серед гравців Warcraft III і Age of Empires II, потім зміцнившись у вигляді окремого жанру. Першим представником окремої гри в жанрі Tower Defense стала комп'ютерна гра «Master of Defense», що вийшла 7 листопада 2005 року. Вона стала дуже популярна, отримала нагороду «Стратегія року» від GameTunnel в 2006-му.

Розробник гри «WarCraft», Blizzard, створив додатковий рівень «Tower Defense» в «Warcraft III: The Frozen Throne»:

Рисунок 1.2 — Tower Defence у грі «Warcraft III: The Frozen Throne»

1.3 Сучасні IT програмування логіки мультимедійних ігор

Для створення мультимедійної гри під ОС Windows потрібно обрати інтегроване середовище розробки, яке буде підтримувати мови C#, C++ (мови які використовуються для створення програм під ОС Windows). Таких середовищ є досить багато, одними з найоптимальнішими є Unity та Unreal Engine [2].

1.3.1 Аналіз платформи розробки ігор Unity

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

Є можливість створювати інтернет–додатки за допомогою спеціального під'єднуваного модуля для браузера Unity, а також за допомогою експериментальної реалізації в межах модуля Adobe Flash Player. Застосунки, створені за допомогою Unity, підтримують DirectX та OpenGL.

Як правило, ігровий движок надає безліч функціональних можливостей, що дозволяють їх задіяти в різних іграх, в які входять моделювання фізичних середовищ, карти нормалей, динамічні тіні і багато іншого. На відміну від багатьох ігрових движків, у Unity є дві основні переваги: наявність візуальної середовища розробки та межплатформенная підтримка . Перший фактор включає не тільки інструментарій візуального моделювання, а й інтегровану середу, лінію складання, що направлено на підвищення продуктивності розробників, зокрема, етапів створення прототипів і тестування. Під міжплатформеній підтримкою надається не тільки місця розгортання (установка на персональному комп'ютері, на мобільному пристрої, консолі і т. д.), Але і наявність інструментарію розробки (інтегроване середовище може використовуватися під Windows і Mac OS).

Третьою перевагою називається модульна система компонентів Unity, за допомогою якої відбувається конструювання ігрових об'єктів, коли останні є комбіновані пакети функціональних елементів. На відміну від механізмів успадкування, об'єкти в Unity створюються за допомогою об'єднання функціональних блоків, а не приміщення в вузли дерева успадкування. Такий підхід полегшує створення прототипів, що актуально при розробці ігор.

Як недоліки наводяться обмеження візуального редактора при роботі з багатокомпонентними схемами, коли в складних сценах візуальна робота ускладнюється. Другим недоліком називається відсутність підтримки Unity

посилань на зовнішні бібліотеки, роботу з якими програмістам доводиться налаштовувати самостійно, і це також ускладнює командну роботу. Ще один недолік пов'язаний з використанням шаблонів примірників (англ. prefabs). З одного боку, ця концепція Unity пропонує гнучкий підхід візуального редагування об'єктів, але з іншого боку, редагування таких шаблонів є складним. Також, WebGL–версія движка, в силу специфіки своєї архітектури (трансляція коду з C # в C ++ і далі в JavaScript), має ряд невирішених проблем з продуктивністю, споживанням пам'яті і працездатністю на мобільних пристроях.

1.3.2 Огляд мови програмування під Unity C#

C # відноситься до сім'ї мов з C — подібним синтаксисом, з них його синтаксис найбільш близький до C ++ і Java. Мова має статичну типізацію, підтримує поліморфізм, перевантаження операторів (в тому числі операторів явного і неявного приведення типу), делегати, атрибути, події, властивості, узагальнені типи і методи, ітератори, анонімні функції з підтримкою замикань, LINQ, виключення, коментарі у форматі XML.

Перейнявши багато від своїх попередників — мов C ++, Pascal, Модула, Smalltalk і, особливо, Java C #, спираючись на практику їх використання, виключає деякі моделі, що зарекомендували себе як проблематичні при розробці програмних систем, наприклад, C # на відміну від C ++ і деяких інших мов, не підтримує множинне успадкування класів (між тим допускається множинне спадкування інтерфейсів)

C # розроблявся як мова програмування прикладного рівня для CLR і, як такий, залежить, перш за все, від можливостей самої CLR. Це стосується, перш за все, системи типів C #, яка відображає BCL. Присутність або відсутність тих чи інших виразних особливостей мови диктується тим, чи може конкретна мовна особливість бути трансльований в відповідні конструкції CLR. Так, з розвитком CLR від версії 1.1 до 2.0 значно збагатився і сам C #; подібної взаємодії слід очікувати і в подальшому (проте, ця закономірність була порушена з виходом C # 3.0, що представляє собою розширення мови, що не спираються на розширення

платформи .NET). CLR надає C #, як і всім іншим .NET–орієнтованим мовам, багато можливостей, яких позбавлені «класичні» мови програмування. Наприклад, прибирання сміття не реалізована в самому C #, а проводиться CLR для програм, написаних на C # точно так же, як це робиться для програм на VB.NET, J # і ін.

1.3.3 Аналіз платформи розробки ігор Unreal Engine

Unreal Engine – ігрова платформа, що розробляється і підтримується компанією E

Перша гра, створена на цьому движку Unreal з'явилася в 1998 році. З тих пір різні версії движка були використані в більш ніж сотні ігор і інших проектів . Написаний на мові C ++, движок дозволяє створювати ігри для більшості операційних систем і платформ: Microsoft Windows, Linux, Mac OS і Mac OS X; консолей Xbox, Xbox 360, Xbox One, PlayStation 2, PlayStation 3, PlayStation 4, PSP, PS Vita, Wii, Dreamcast, GameCube і ін., а також на різних портативних пристроях, наприклад, пристроях Apple (iPad, iPhone), керованих системою iOS і інших. (Вперше робота з iOS була представлена в 2009 році , в 2010 році продемонстровано роботу движка на пристрої з системою webOS).

Для спрощення портування движок використовує модульну систему залежних компонентів; підтримує різні системи рендеринга (Direct3D, OpenGL, Pixomatic; в ранніх версіях: Glide, S3, PowerVR), відтворення звуку (EAX, OpenAL, DirectSound3D; раніше: A3D), засоби голосового відтворення тексту, розпізнавання мови, модулі для роботи з мережею та підтримуваних пристроїв введення.

Для гри по мережі підтримуються технології Windows Live, Xbox Live, GameSpy і інші, включаючи до 64 гравців (клієнтів) одночасно. Таким чином, движок адаптували і для застосування в іграх жанру MMORPG (один із прикладів: Lineage II).

В 2014 році вийшов Unreal Engine 4 – виглядає багатообіцяюче. Фактично за 19 $ ми отримуємо движок з великим функціоналом ніж Unity і з відкритими

початковими кодами. Крім того для нього була створена нова екосистема з великим числом навчального матеріалу, магазином контенту, форумом і співтовариством розробників. Можливість писати як на чистому коді, так і на візуальному редакторі.

Тепер про мінуси і недоліки.

Движок все таки більш професійне від цього більш складний в освоєнні, ніж Unity. Магазин контенту на жаль поки дуже примітивний. Також немає редактора під Linux і платформ для складання поки всього чотири. Scaleform не входить в Unreal Engine 4.

1.3.3 Огляд мови програмування під Unreal Engine C ++

C ++ – компільована, статично типізована мова програмування загального призначення.

Підтримує такі парадигми програмування, як процедурне програмування, об'єктно–орієнтоване програмування, узагальнене програмування. Мова має багату стандартну бібліотеку, яка включає в себе поширені контейнери і алгоритми, введення–виведення, регулярні вирази, підтримку багатопоточності і інші можливості. C ++ поєднує властивості як високорівневих, так і низькорівневих мов. У порівнянні з його попередньою мовою C, найбільшу увагу приділено підтримці об'єктно–орієнтованого і узагальненого програмування.[3]

C ++ широко використовується для розробки програмного забезпечення, будучи одною з найпопулярнішою мов програмування . Область його застосування включає створення операційних систем, різноманітних прикладних програм, драйверів пристроїв, додатків для вбудованих систем, високопродуктивних серверів, а також ігор. Існує безліч реалізацій мови C ++, як безкоштовних, так і комерційних і для різних платформ. Наприклад, на платформі x86 це GCC, Visual C ++, Intel C ++ Compiler, Embarcadero (Borland) C ++ Builder і інші. C ++ зробила величезний вплив на інші мови програмування, в першу чергу на Java і C #.

Синтаксис C ++ успадкований від мови C. Одним з принципів розробки було збереження сумісності з C. Проте, C ++ не є в строгому сенсі надбезліччю C; безліч програм, які можуть однаково успішно транслюватися як компіляторами C, так і компіляторами C ++, досить велике, але не включає всі можливі програми на C.C ++ містить засоби розробки програм контрольованої ефективності для широкого спектра задач, від низькорівневих утиліт і драйверів до вельми складних програмних комплексів.

Зокрема, аналіз виявив таке:

— висока сумісність з мовою C: код на C може бути з мінімальними переробками скомпільовано компілятором C ++, зовнішньо–мовний інтерфейс є прозорим, так що бібліотеки на C можуть викликатися з C ++ без додаткових витрат, і більш того —при певних обмеженнях код на C ++ може експортуватися зовні не відрізнити від коду на C (конструкція extern "C");

— підтримка різних стилів програмування: традиційне імперативне програмування (структурний, об'єктно–орієнтоване), узагальнене програмування, функціональне програмування, що породжує метапрограмування;

— автоматичний виклик деструкторів об'єктів в адекватному порядку (зворотному виклику конструкторів) спрощує і підвищує надійність управління пам'яттю і іншими ресурсами (відкритими файлами, мережевими з'єднаннями, сполуками з базами даних і т.п.);

— перевантаження операторів дозволяє коротко і легко записувати вирази над користувацькими типами у природному алгебраїчній формі;

— є можливість управління константністю об'єктів (модифікатори const, mutable, volatile), використання константних об'єктів підвищує надійність і служить підказкою для оптимізації, перевантаження функцій–членів за ознакою константності дозволяє визначати вибір методу в залежності мету виклику (легкий для читання,важкий для зміни), оголошення mutable дозволяє зберігати логічну константность побачивши ззовні коду, що використовує кеші і ледачі обчислення;

— шаблони C ++ дають можливість побудови узагальнених контейнерів і алгоритмів для різних типів даних, попутно шаблони дають можливість виробляти обчислення на етапі компіляції;

— можливість вбудовування предметно–орієнтованих мов програмування в основний код, саме такий підхід використовує, наприклад бібліотека Boost.Spirit, що дозволяє задавати EBNF–граматику парсеров прямо в коді C ++;

— основною перевагою є доступність, для C ++ існує величезна кількість навчальної літератури, перекладеної на всілякі мови,

Аналіз виявив такі недоліки мови C++:

— відсутність системи модулів, це змушує дублювати опису об'єктів, породжує неочевидні вимоги до коду (див. правило одного визначення) і збільшує обсяг компилируемого тексту, а значить і час компіляції;

— наявність більш ніж одного механізму для виконання одних і тих же завдань, що ускладнює мову і призводить до неоптимальному і небезпечному кодування;

— успадковані від C небезпечні і провокують помилки можливості, такі як макроозначення #define, адресна арифметика і неявне приведення типів;

— можливість прямого управління розподілом пам'яті провокує помилки, що призводять до раптового краху програм через руйнування стека або звернення до невиділеної пам'яті;

— шаблони породжують об'ємний і не завжди оптимальний код, часткове визначення шаблонів ускладнює як сама мова, так і програми, де воно використовується;

— об'єктно-орієнтована підсистема побудована так, що виявляється важко застосовувати багато прийомів, звичайні для інших об'єктних мов;

— множинне (в тому числі віртуальне) успадкування ускладнює транслятор і призводить до створення громіздких ієрархій класів, які при будь–якій зміні вимог до програми можуть зажадати серйозного перегляду;

— складний синтаксис і об'ємна специфікація мови ускладнюють її вивчення;

— мова не заохочує створення надійного, легко читається і зручного в супроводі коду, замість цього часто пропонуючи вибір між короткими і простими, але небезпечними засобами, успадкованими від C, і новими, об'ємними і складними, але більш безпечними механізмами.

1.4 Роль проектування архітектури ПЗ

Архітектура програмного забезпечення — спосіб структурування програмної абстракції, і мати багато фаз роботи, кожна з яких може мати окрему архітектуру.

Дослідження архітектури програмного забезпечення намагається визначити як найкраще розбити систему на частини, як ці частини визначають та взаємодіють одна з одною, як між ними передається інформація, як ці частини розвиваються поодинці і як все вищеописане найкраще записати використовуючи формальну чи неформальну нотацію.[4]

Архітектура повинна будуватись щоб найкраще відповідати вимогам до системи що створюється, згідно принципу "форма відповідає функції".

Згідно Perry та Wolf архітектурою є набір елементів що мають певну форму (властивості і обмеження що накладаються на елементи), і їх обґрунтування. Обґрунтування фіксує мотиви вибору певного архітектурного стилю, елементів і обмежень. Рой Філдінг вважає що обґрунтування необхідне на етапі створення архітектури, і корисне надалі але не є невід'ємним її елементом. Тому що архітектура має набір властивостей що дозволяють їй задовольняти вимоги, і незнання цих вимог може призвести до змін що порушують архітектуру, але до архітектури входять властивості а не вимоги. Наприклад можна не знати що в "архітектуру" стола закладену вимогу стійкості, і тому він повинен мати більше двох ніжок. Не знаючи про цю вимогу, ми можемо відпиляти забагато ніжок і стіл впаде. Але це тому що ми порушили архітектурне обмеження "мати три чи більше ніжок".

Терміном "Архітектура" також називають документування архітектури програмного забезпечення. Документування архітектури ПЗ спрощує процес комунікації між зацікавленими особами, дозволяє зафіксувати прийняті на ранніх

етапах проектування рішення про високорівневий дизайн системи і дозволяє використовувати компоненти цього дизайну і шаблони проектування повторно в інших проектах.

1.5 Огляд UI користувача та його видів

Інтерфе́йс користувача́ (ІК), (англ. user interface, UI) — засіб зручної взаємодії користувача з інформаційною системою. Сукупність засобів для обробки та відбиття інформації, якнайбільше пристосованих для зручності користувача; у графічних системах інтерфейс користувача, втілюється багатовіконним режимом, змінами кольору, розміру, видимості вікон, їхнім розташуванням, сортуванням елементів вікон, гнучкими налаштуваннями як самих вікон, так і окремих їх елементів доступністю багатокористувацьких налаштувань. Як правило, мета створення інтерфейсу користувача, полягає у тому, щоби зробити інтерфейс користувача, який спрощує, ефективний і приємний для керування машиною таким чином, щоби забезпечити бажаний результат. Це, зазвичай, означає, що оператор повинен застосовувати щонайменші зусилля для досягнення очікуваного підсумку, а також, щоби машина зменшувала небажані результати для людини.

Види інтерфейсів :

— інтерфейс прямої обробки, це назва загального класу інтерфейсів користувача, який дозволяє користувачам маніпулювати наданими їм об'єктами, з використанням дій, котрі принаймні, відповідають фізичному світу;

— графічні інтерфейси користувача (GUI) приймають вхідні дані за допомогою таких пристроїв, як комп'ютерна клавіатура та миша, й забезпечують графічний висновок на моніторі комп'ютера, у GUI-дизайні, широко використовуються як мінімум, два різні принципи, а саме об'єктно-орієнтовані інтерфейси користувача(OOUI) й інтерфейси, орієнтовані на додатки;

— веб-інтерфейси користувача або веб-інтерфейси користувача (WUI), які приймають вхідні дані та забезпечують виведення, створенням веб-сторінок, які

передаються інтернетом і проглядаються користувачем за допомогою програми веб-браузера;

— сенсорні екрани – дисплеї, які приймають введення, дотиком пальців або стилусом, використовуються у все більшій кількості мобільних пристроїв і багатьох засобах точок продажу, промислових процесах і машинах, пристроях самообслуговування тощо;

— апаратні інтерфейси – фізичні, просторові інтерфейси, присутні на виробах у повсякденному житті від тостерів, до приладових панелей автомобілів чи кабін літаків, вони, як правило, є поєднанням ручок, кнопок, слайдерів, перемикачів і сенсорних екранів;

— перехресні інтерфейси — графічні інтерфейси користувача, в яких основне завдання полягає у перетині меж, а не зосередженні на вказівках;

— інтерфейси жестів, є графічні інтерфейси користувача, які приймають вхідні дані у вигляді;

— масштабований інтерфейс користувача — це графічні інтерфейси користувача, в яких інформаційні об'єкти з'являються на різних рівнях та деталях, і користувач може змінювати масштаби області, яка відображається, щоб побачити більше деталей.

Отже Перш за все розберемся з платформою. Чому було обрано саме Unity а не Unreal Engine 4 . У той час, як багато студій–розробники ігор використовують власні ігрові движки, є ще величезний ринок інді–розробників і навіть великих студій, які потребують ігровому движку, який допоможе створити свою гру за короткі терміни. Unity3d і Unreal Engine 4 – це одні з найпопулярніших ігрових движків, доступних на сьогоднішній день. Уявімо собі що я і є тою самою невеличкою інді–компнією яка планує створити даний проект завдяки одному із цих двох платформ. У кожного з двигунів є свої сильні сторони для різних завдань. Unity підійде новачкам і любителям, в той час як Unreal — строго для професійних розробників. Також в Unreal Engine 4 використовується мова програмування C ++. У Unity3d в основному C # або JavaScript. Одразу скажу,

мене більше приваблює C # . Тому враховуючи всі переваги і недоліки які описанні в попередніх розділах було обрано Unity3d для даної роботи.

По–друге я обрав саме Об’єктно–орієнотоване програмування в створенні мультимедійного додатку оскільки, я створюю перш за все гру . В ігрі присутні безліч ігрових об’єктів, які можуть повторюватись і повинні регувати на кожен інший об’єкт і при стандартному підході спадкуванні виникає проблема тендітних базових класів – коли змінити реалізацію типу–предка неможливо, не порушивши коректність функціонування типів–нащадків. А при ООП ми уникаєм цієї проблеми оскільки в ньому всі компоненти чи об’єкти стають автономними і нагадують інтерфейси проте інтерфейси дозволяють тільки виділити у класів загальну сигнатуру функцій і властивостей, а компоненти дозволяють винести загальну реалізацію класів окремо. Також при використанні стандартному підході прийдеться витратити більше часу на кодування і переробку коду для всього проекту одразу, коли в ООП для кожного об’єкта потрібен свій окремий і якщо потрібно його переробити це не займе багато часу. Тому об’єктно-орієнтоване програмування є більш вигоднішим і доцільнішим для створення ігри.

2 ПРОГРАМНА РЕАЛІЗАЦІЯ ЛОГІКИ ГРИ МОВОЮ C# ТА СТВОРЕННЯ ІНТЕРФЕЙСУ

2.1 Архітектура програмного засобу

Розробку гри Tower-Defence було розділено на 4 кроки, щоб показати створення її архітектури та компонентів.

Перший крок: постановка задачі – скільки б багато сторінкової документації не було створено по грі, потрібно завжди вміти виділити основне. Зразкове опис ігрового процесу, звичайно, краще розділити на use case:

Гравець повинен не дати ворогам зруйнувати Будинок. Для це він повинен розставити вежі, які будуть знищувати ворогів, які потрапили в радіус дії. Одночасно одна вежа може атакувати тільки одного ворога. Коли він гине або виходить з радіусу дії вежі, вибирається наступний з доступних. Вороги з'являються хвилями, з кожною хвилею їх кількість збільшується. Вороги рухаються з точки появи по дорозі до Будинку. Підійшовши до будинку починають його руйнувати. Коли будинок зруйнований – гра закінчена. Отже потрібно розробити логіку гри.

Другий крок: аналіз завдання – почнемо з опрацювання головного: веж, ворогів і будинку. Так, створивши сцену з уже розставленими ігровими об'єктами (вежами і будинком), ми повинні отримати результат: вороги з'являються, рухаються до будинку, гинучи по шляху від пошкоджень веж, а дійшовши до будинку, наносять йому пошкодження. При смерті ворог зникає зі сцени. При руйнуванні будинку або знищенні всіх ворогів -гра закінчена.

Виділимо основні ігрові сутності, вкажемо їх властивості та поведінку.
Вежа її властивості це шкода, кулдаун між пострілами, радіус вибору мети, логіка вибору мети.Поведінка – вибір мети і атака мети.

Також для різноманітності геймплея будуть різні види веж: по пошкодженням, радіусу ураження і логіці вибору мети.

Ворог його властивості це HP, пошкодження, кулдаун між атаками.

Поведінка – переміщення по маршруту до Будинку. Атака будинку при підході до нього впритул. Коли HP = 0, вважається вбитим.

Різні види ворогів – наприклад, по HP

Будинок має такі властивості як HP. Його поведінка – коли HP = 0, гравець програв.

Третій крок: декомпозиція – вважається, що правильна декомпозиція задач дуже важлива. Велику частину часу варто витратити на опрацювання абстракцій і логічного зв'язку між ними.

Почнемо з узагальнення поведінки, щоб виявити які повинні бути компоненти.

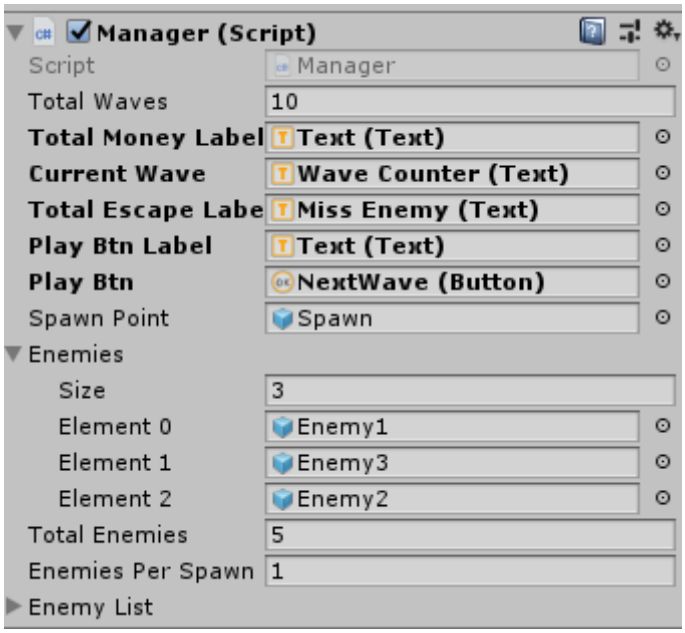


Рисунок 2.1 — Менеджер логіки гри

У логіці реалізації будемо при Start викликати метод Manager який розпочне гру. Він в Update буде перевіряти скільки ворогів мертві з тих що були створенні, і, з урахуванням кількості хвиль і поточної хвилі або говорити, що гравець переміг. Також перевіряти не зруйнований чи будинок: якщо так, то гравець програв. За допомогою цього скрипту ми дізнаємся скільки ворогів видів ворогів взагалі є в грі, з яким інтервалом вони з’являються, і загальна кількість їх на екрані.

Тепер можна зробити реалізацію першого варіанту гри. У Unity створимо префаб, налаштовані як візуально (вежі різних типів, Крип, будинок), так і логічно – тобто з доданими компонентами. Також властивості компонентів повинні бути налаштовані, тому немає необхідності створювати окремі класи. Але нам знадобиться розуміти, що є кріп, а що будинок. Для цієї мети в Unity є теги.

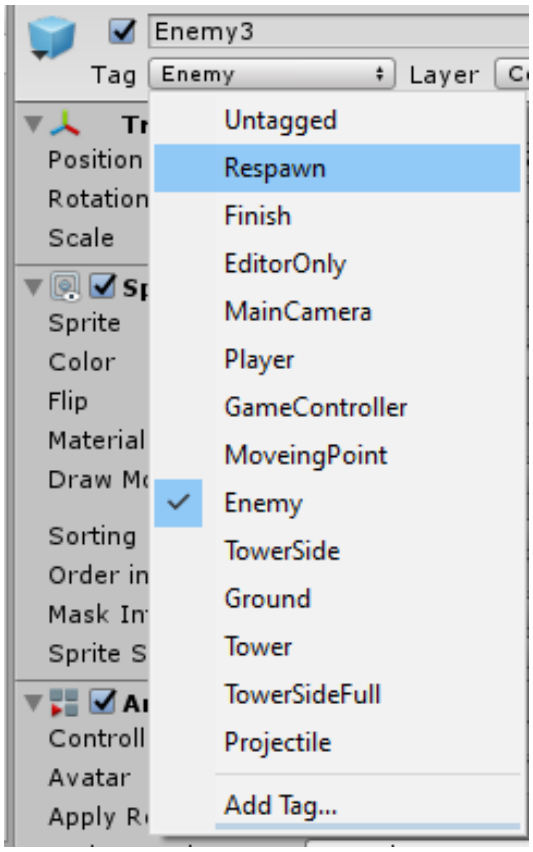


Рисунок 2.2 —Теги кожного префабу гри

Почнемо реалізацію з генерації коду UML діаграм, отримавши скелет–пустушку. Після чого створимо реалізацію поведінки компонентів. Не забувайте про те, що краще використовувати абстракції, а не конкретні реалізації, хоча це і збільшить зв'язність.

А як тепер розширити геймплей – припустимо, додавши економіку. За вбивство крипта тепер треба отримувати гроші, а їх витрачати на будівництво веж. І змінювати вартість веж, адже кожна вежа має свої значення урону по ворогам. Окрім того кожна з них має своє унікальну анімацію, атаки яка відображається на екрані.

Рисунок 2.3 — Вартість будівництва кожної вежі

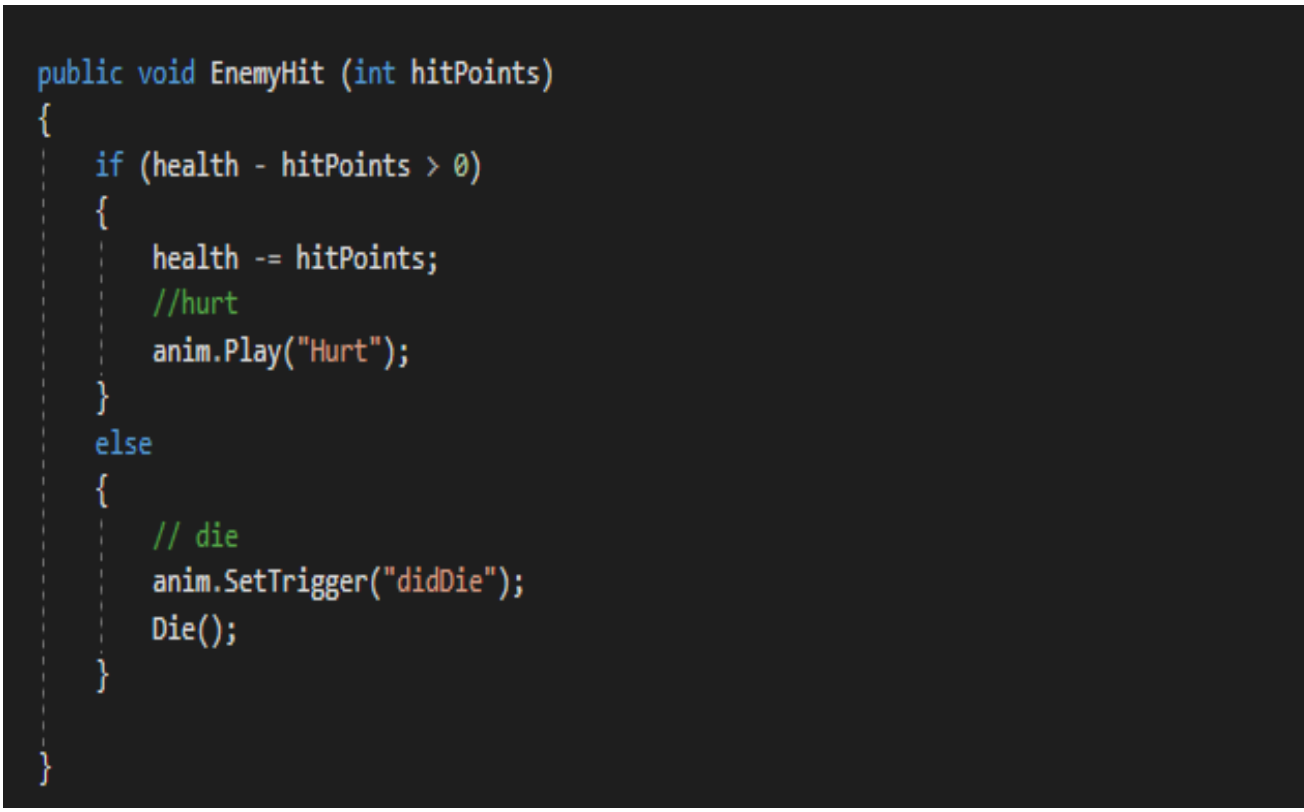
Головне питання "А як же інкапсуляція? Виходить, що хто завгодно може змінити кулдаун і пошкодження? "Ні, тільки геймдизайнер, налаштовуючи інтуїтивно–зрозумілим чином властивості компонента. Тому у всіх властивостей буде тільки get, щоб відображати інформацію в UI, а set не знадобиться. Крім того, Unity не зможе відобразити в інспектора властивості з вказаною конструкцією get / set, і знадобиться створювати поля, відмічені [SerializeField]. І це правильно, інші класи з коду не зможуть змінити значення властивості.

Повернемося до компоненту. Хтось буде викликати запуск цієї поведінки і зупиняти його, вказувати мету, змінювати мету.

Так як логіка вибору ворога буде різна у різних типів веж, а також не тільки вежа, але і кріп використовуватимуть цю поведінку, то треба виділити два логічних компонента.

Перший буде працювати з фізикою, як тільки в коллайдер–тригер увійшов ворог, він його додає в чергу цілей, створюючи подію, що мовляв цілей додалося. Як тільки кріп покине зону ураження – прибрати цю ціль з черги, створити подію видалення ворога з черги.

Другий логічний компонент буде реагувати на ці події і використовувати вже сам EnemyHit для нанесення шкоди обраної мети та припинення нанесення шкоди.



```
public void EnemyHit (int hitPoints)
{
    if (health - hitPoints > 0)
    {
        health -= hitPoints;
        //hurt
        anim.Play("Hurt");
    }
    else
    {
        // die
        anim.SetTrigger("didDie");
        Die();
    }

}
```

Рисунок 2.4 — Функція логіки нанесення пошкоджень

Лістинг коду 2.1 — Облислення пошкоджень

```
public void EnemyHit (int hitPoints)
  {
     if (health - hitPoints > 0)
     {
        health -= hitPoints;
        //hurt
        anim.Play("Hurt");
     }
     else
     {
        // die
        anim.SetTrigger("didDie");
        Die();
     }

  }
```

Одразу після створення функції EnemyHit яка відповідає за нанесення шкоди обраної мети та припинення нанесення шкоди. Нам потрібно реалізувати логіку вибору найближчого ворога башнею.

```
private List<Enemy> GetEnemiesInRange()
{
    List<Enemy> enemiesInRange = new List<Enemy>();

    foreach (Enemy enemy in Manager.Instance.EnemyList)
    {
        if(Vector2.Distance(transform.localPosition, enemy.transform.localPosition) <= attackRadius )
        {
            enemiesInRange.Add(enemy);

        }
    }
    return enemiesInRange;
}

private Enemy GetNearestEnemy()
{
    Enemy nearestEnemy = null;
    float smallestDistance = float.PositiveInfinity;

    foreach (Enemy enemy in GetEnemiesInRange() )
        {

        if (Vector2.Distance(transform.localPosition, enemy.transform.localPosition) < smallestDistance)
        {
            smallestDistance = Vector2.Distance(transform.localPosition, enemy.transform.localPosition);
            nearestEnemy = enemy;
        }

    }
    return nearestEnemy;
}
}
```

Рисунок 2.6 — Функції логіки логічного вибору ворогів

GetEnemiesInRange: вибирає ворога який увійшов в радус для пострілу вежі

GetNearestEnemy: вибирає ворога який найближчий до вежі

Таким чином легко розширити core–gameplay механіку вибору цілі (механіка основна, так як вежі тільки їй розрізняються). Однак, можна і інший варіант зробити, з успадкуванням. Буде базовий компонент GetEnemiesInRange з дефолтной логікою для вежі. А клас GetNearestEnemy і будуть перевизначати метод і змінювати ціль на найближчу.

2.2 Створення ігрового майданчика гри

Як тільки ми запускаєм наш обрану платформу Unity і створюєм проект, окрім головної камери і інструментів для обробки та програмування нічого більше немає.

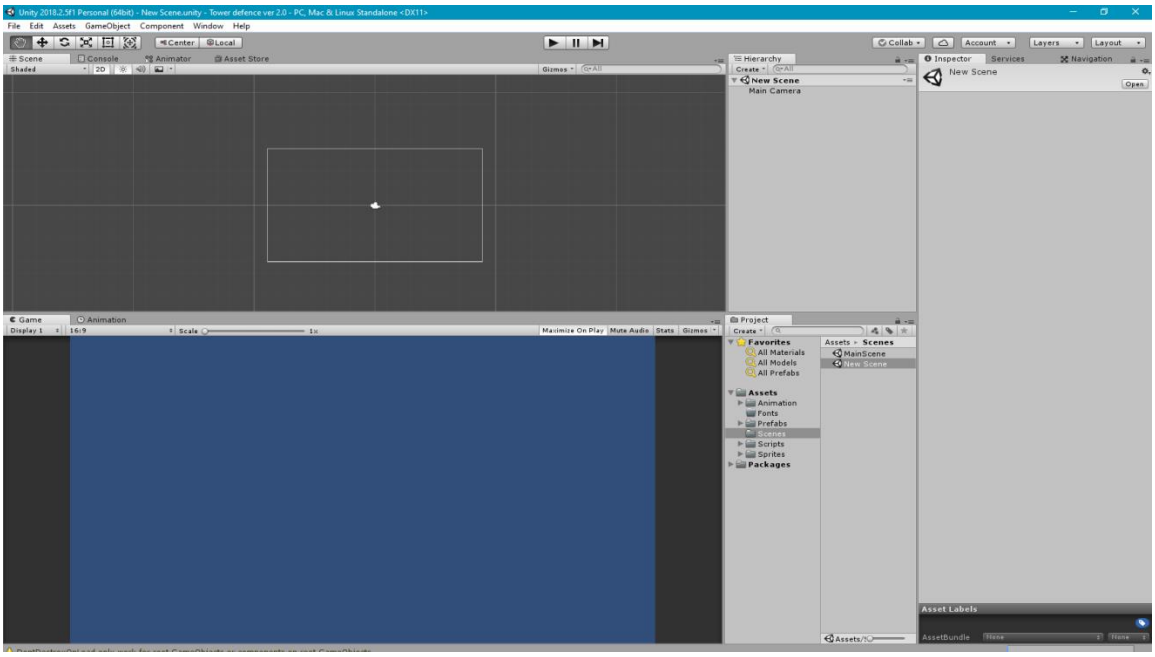


Рисунок 2.7 — Базове вікно в Unity

І потрібно розпочинати все з нуля. І перед тим як розпочати роботу з платформою Unity. Треба намалювати Sprite тобто зовнішнє зображення ігрових об’єктів, а саме: веж, дорожку по якій будуть крокувати наші вороги, самих ворогів і інші об’єкти, які будуть доповнювати нашу гру.

Рисунок 2.8 — Sprite ворогів та веж

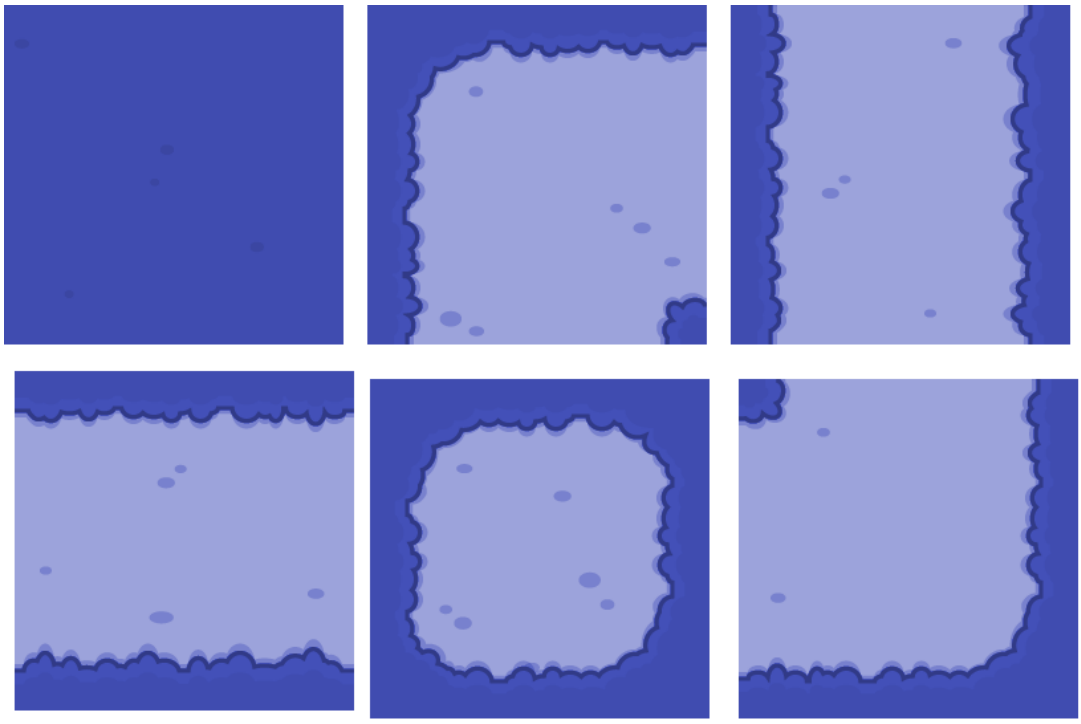


Рисунок 2.9 — Sprite землі та доріг

Рисунок 2.10 — Sprite інших ігрових об’єктів

Після створення всіх спрайтів ми загрузим їх в наш проект в Юніті, де вже розпочнем заповнювати базове вікно. Оскільки нам потрібно заповнити всю Main Camera тобто головну камеру, яку буде бачити гравець весь час під час гри, то нам потрібно брати по 1 спрайту і розміщувати їх рівномірно по горизонталі вертикалі, щоб не залишати пустих проміжків. І щоб не допустити промаху, можна використати внутрішню функції платформи Юніті Snap Setting. За її

допомогою ми можем рівно розподілити всі ігрові об’єкти не витрачаючи на це безліч час.

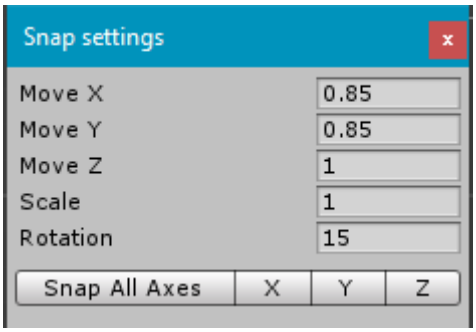


Рисунок 2.11 — Функція Snap Setting

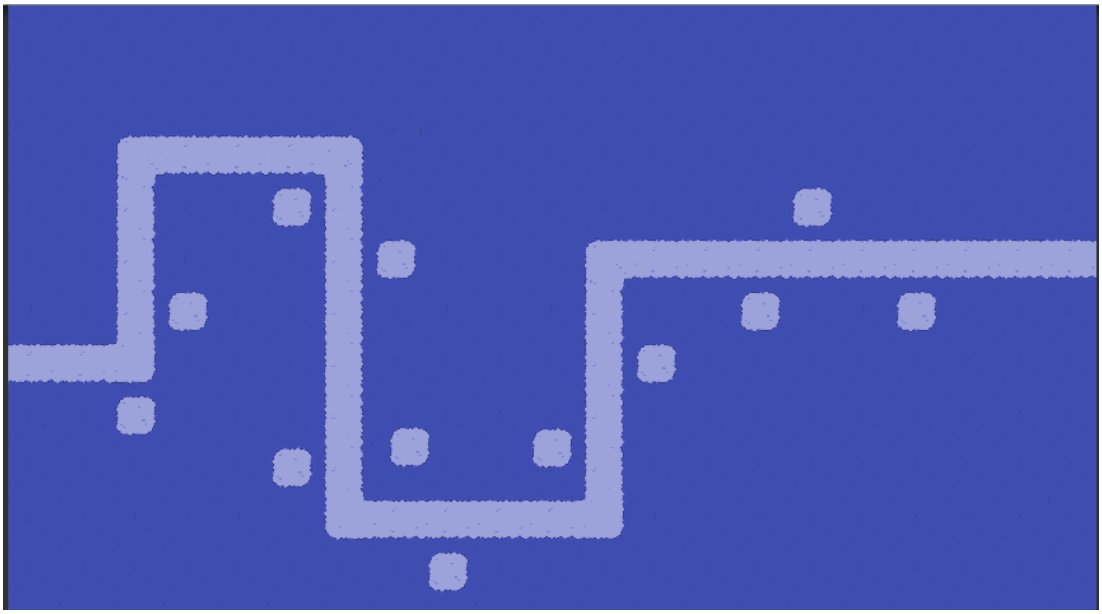


Рисунок 2.12 — Ігрове поле вже з готовою дорогою

Після заповнення Main Camera нашими спрайтами землі та доріг, щоб було більш приємніше для зору гравця, нам потрібно заповнити ігрове поле іншими ігровими об’єктами, які ми також завчасно підготували для проекта.

Тут вже функція Snap Setting нам не потрібна, адже ми розміщуєм як побажаєм. І оскільки в нас ігрових об’єктів не багато ми можем їх відзеркалювати, що допоможе розширити унікальність ігрового майданчика і надасть йому певної краси. Завдяки такій невеличкії дії ми можем заохочувати грати гравця більше. Адже всі люди потребують різновидності і хочу чогось нового.

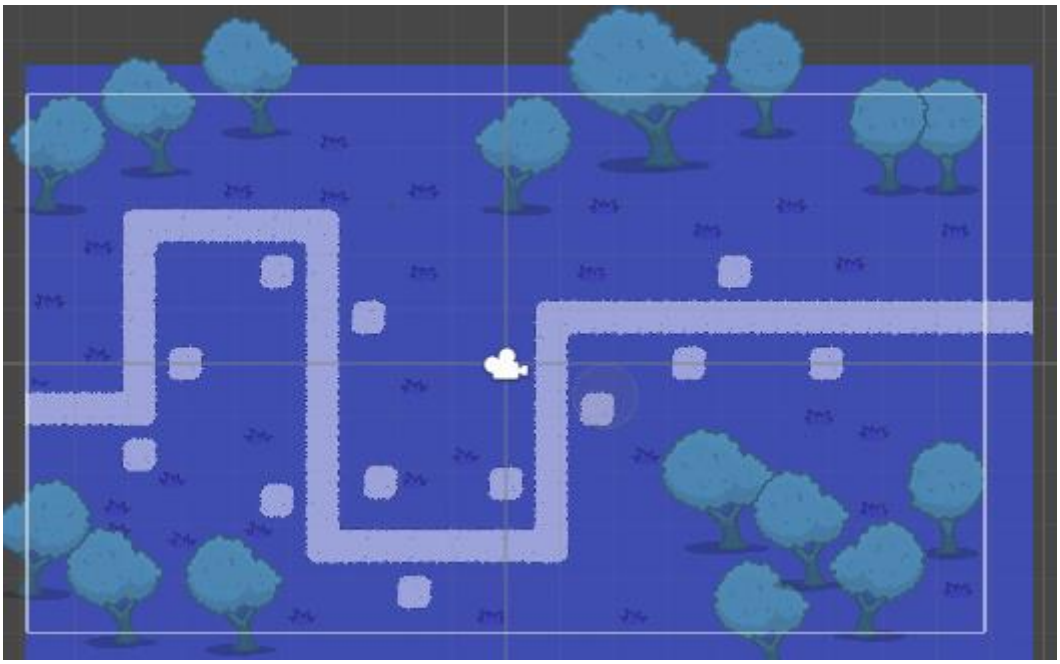


Рисунок 2.13 — Ігрове поле в Unity

2.3 Розробка та реалізація UI користувача

Інтерфейс користувача представляє собою панель, яка знаходиться в нижній частині екрану. Він представляє собою інформаційну та інструментальну панель, на якій будуть показуватись:

1) кількість доступних грошей;
2) види доступних захисних споруд та їх ціна;
3) табличка яка вказує на кількість хвиль і кількість ворогів які пройшли до кінця дороги.

Канвас інтерфейсу користувача буде містити в собі три текстові блоки, що показують нам інформацію про наявні кошти , теперішня хвиля та скільки ворогів Escaped відповідно. Беручи до уваги той факт значна частина робочої зони вкрита синім кольором, можемо зробити висновок що цей колір найбільше кидається в очі, тому ці текстові блоки мають бути іншого кольору (жовтим, або відтінком жовтого). По середині ігрового інтерфейсу будуть знаходитись види захисних споруд та ціни на них. Вони створюються за допомогою компонента Panel (з англ. Панель). Вони будуть відігрувати роль своєрідних кнопок об′єднаними із текстовими блоками, завдяки яким користувач буде мати

можливість купувати доступні гармати. Аналогічно до бокових текстових блоків, ціна покупки зображується відповідним кольором. Для різноманітності, шрифт яким зображується ціна на захисні споруди змінено з Arial на Roboto-Light. В кінцевому варіанті на панелі зображено три кнопки із зображенням захисної вежі на кожній, які розташовані по порядку зростання ціни їх покупки.

Окрім вище сказаних компонентів, структура інтерфейсу (рисунок 2.14) користувача також включає в себе і елементи які містять графічну частину карти, тобто зону вільного користування, дорогу, точки старту та кінця, тощо.

Рисунок 2.14 — Загальна структура UI

Верхня частина користувацького інтерфейсу, використана для текстового табла, яке відображає кількість життів, які має гравець. Канвас цієї частини інтерфейсу дублюється із попереднього, та містить у собі лише одну текстову зону, оформлену жовтому кольорі та шрифтом Roboto-Thin 204-го розміру, для того щоб гравецю завжди кидалась в очі інформація про кількість його життів і кількість хвиль, адже це допоможе йому побудавати певний алгоритм протистоянна проти ворогів. Завдяки цьому користувач неначе спілкується з грою і задає їй певні команди, які вона безпідказно виконується, що є дуже комфортно в ігровій індустрії

Кінцевий варіант користувацького інтерфейсу (UI) показано на рисунку 2.14.

Рисунок 2.14 — Кінцевий вигляд UI користувача

2.4 Створення захисних споруд

Всього в грі є 3 різновиди захисних споруд:

—turrets tower (з англ. Захисні турелі);

—stone tower (з англ. Кам’яні башні);

—fire tower (з англ. Огняні башні).

Кожна з них має свої різні характеристики такі як:

— розмір;

— damage(з англ. Завдана шкода);

— тип снаряду;

— швидкість польоту снаряду;

— анімація пострілу/попадання;

— ціна покупки.

Графічні моделі захисних веж створюються в іншому редакторі, в нашому випадку Adobe Photoshop, так як Unity має свій обмежений спектр можливостей. Вигідніше використовувати інші графічні редактори для створення саме моделей об′єктів. В цьому редакторі є свої стандартні моделі таких споруд, їх ми і використаємо.

Рисунок 2.15 —Захисні вежі

Файли які містять графічні моделі веж переносяться у папку імпортованих файлів Unity, для того що щоб двигун опрацював їх, що дасть нам змогу їх редагувати та обробляти [5]. Вежі будуть розташовуватись на, призначених для них, клітинках зони вільного користування Tower Side (рисунок 2.16).

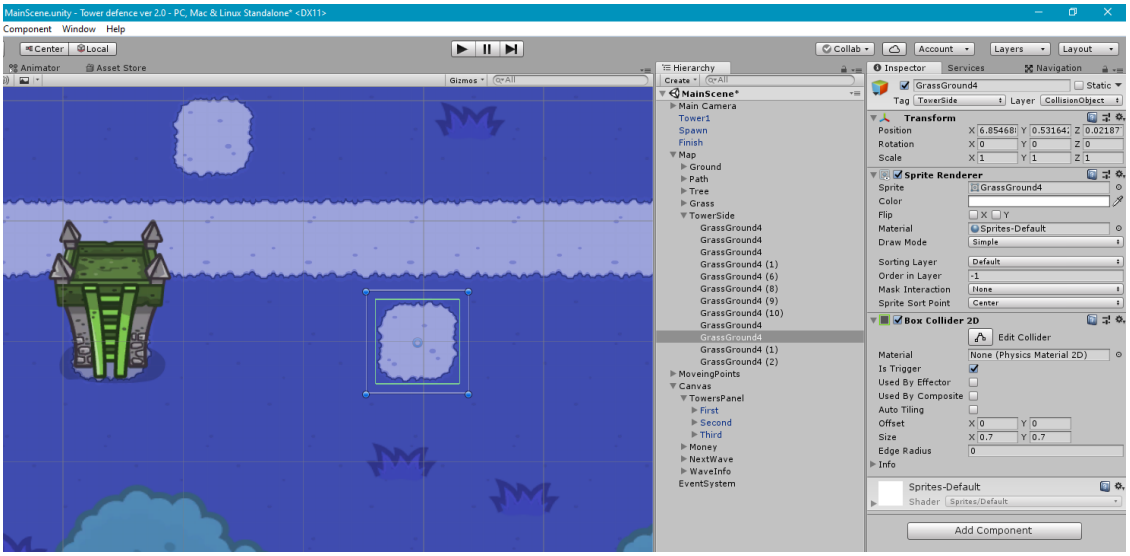


Рисунок 2.16 —Розташування захисної вежі

Обравши сегмент PartToRotate, створюється віртуальна точка FirePoitn(з англ. точка вогню). Вона розміщується на рівні сегмента Tower(вежі). Ця точка буде слугувати місцем де буде відбуватись вистріл з вежі, відповідно для цього необхідно створити новий об′єкт який буде Projectile снарядом, що вилітає з даної точки.

Для того щоб користувач бачив що постріл його гармати приніс якийсь результат, додаються анімації удару (рисунок 2.17). Анімація має свою певну кількість кадрів, та повторюється лише один раз на один постріл. Вона представляє собою попадання снарядом у персонажа. При попаданні у ворога, показується анімація нанесення пошкоджень персонажу і якщо пошкодження перевищили позначку здоров'я персонажа то виникає анімація смерті (рисунок 2.18).

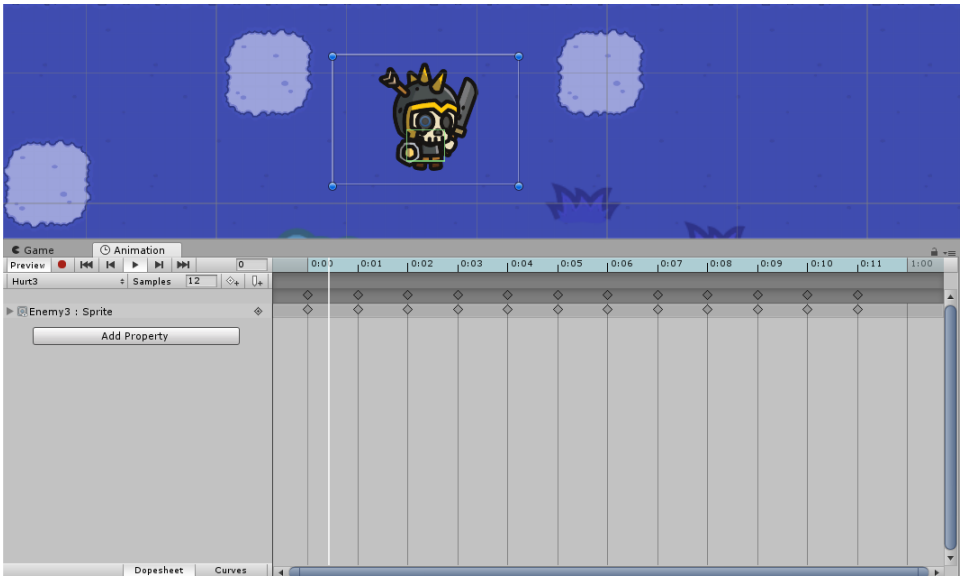


Рисунок 2.17 — Анімація удару по ворогу

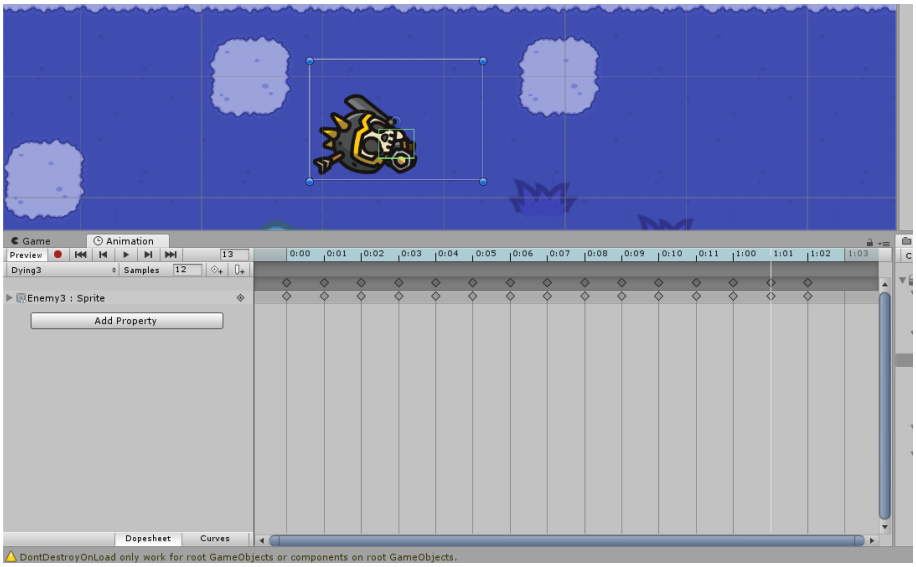


Рисунок 2.18 — Анімація смерті ворога

Кожна вежа має свій тип снаряду (рисунок 2.19) які створюється абсолютно аналогічним чином, тобто створюються нові матеріали, фарбуються, розбиваються на сегменти, тощо. Відмінністю є вирахування пошкоджень, які ми задаєм в інспекторі.

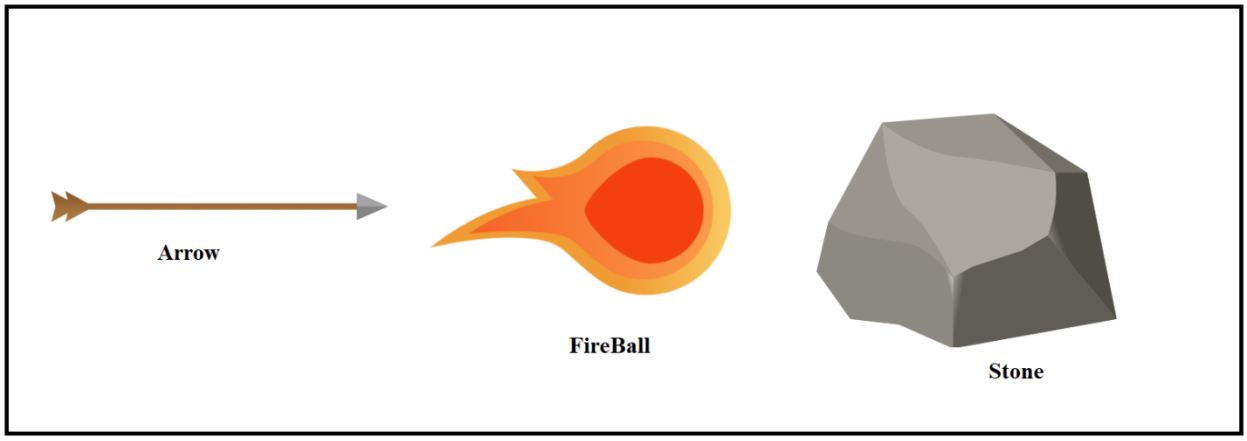


Рисунок 2.19 — усі види снарядів веж

Кожному снаряду задається своя унікальна швидкість польоту ніж у попередньої споруди, а також затримка між пострілами (рисунок 2.20).

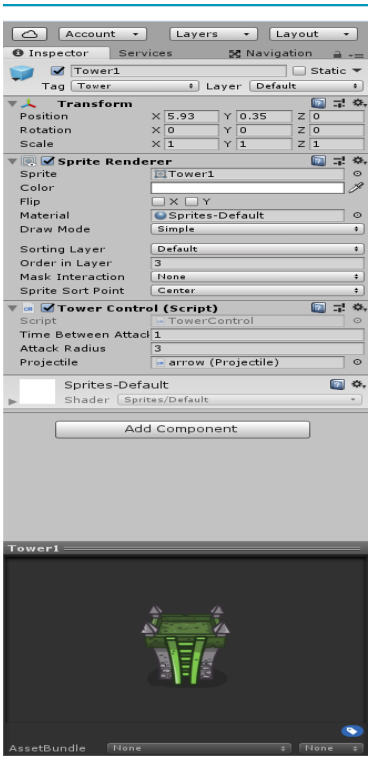


Рисунок 2.20 — інспектор кожної вежі в Unity

2.5 Реалізація компоненту Manager

Компонент Manager – логіка гри. Які умови гри. Скільки хвиль. Які вороги. Можливо гра йде на виживання між багатьма гравцями. Іншими словами компонент Manager є основною логікою гри від якого залежать всі рухи, всі дії гравця. Також на його базі створенно багато функції без яких запуск гри не є можливим.

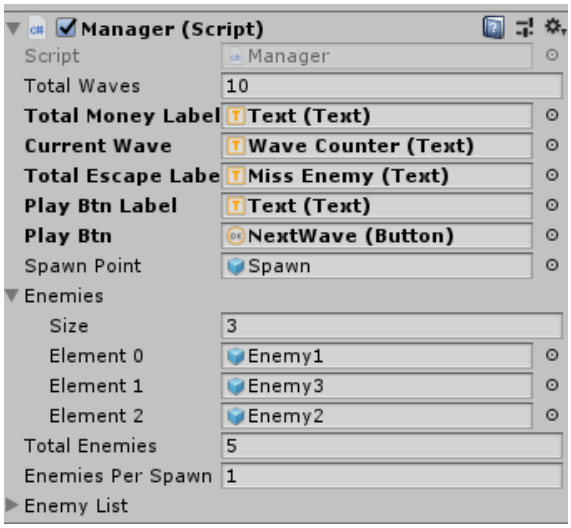


Рисунок 2.21 — компонент Менеджер логіки гри

В кожній грі є умови перемоги і поряд з ним умови поразки. Якщо забрати умови поразки, тобто гравець буде перемагати без тяжких зусиль, то через деякий проміжок часу йому набридне і гра буде не цікава, тай за гру її буде важко вважати.[6] Саме тому сучасні ігри роблять велику ставку на азарт гравців. Ця гра не є виключенням.

Гра завершується коли кількість ворогів, які пройшли весь шлях переткне межу в 10 , а як тоді перемогти в ній виникає запитання. В даній грі існує 10 кількість рівнів, які стають доступні після проходження попереднього рівня. Кожен рівень ускладнюється будь-якими різними способами. Починаючи від різних видів ворогів, закінчуючи їх швидкість і кількістю HP. Це декілька ускладнює ігровий процес, що змушує гравця приділяти більше уваги і розумових вмінь щоб перемогти в ній.

Рисунок 2.22 — компонент Менеджер в дії

Щоб реалізувати логіку початку гри ми викликаєм функцію ShowMenu() в Start, щоб гравець міг натиснути на кнопку коли розмістив уже всі вежі так як йому зручно (рисунок 2.23).. Одразу після натискання кнопки викликаєкаться функція IEnumerator Spawn(), яка розпочинає створення ворогів на даному етапі гри ,щоб перевіряти скільки ворогів мертві з тих що були на початку хвилі, якщо нікого з ворогів немає, гравець переміг і наступає наступна хвиля. Також в Update() перевіряється чи не зруйнований будинок бо в іншому разі гравець програє.

Рисунок 2.23 — виклик функції ShowMenu()

Лістинг коду 2.2—функція ShowMenu()

```
public void ShowMenu()
   {
      switch(currentState)
      {
         case gameStatus.gameover:
            playBtnLabel.text = "Play Again";

            break;

         case gameStatus.next:
            playBtnLabel.text = "Next wave";

            break;

         case gameStatus.play:
            playBtnLabel.text = "Play game";

            break;

         case gameStatus.win:
            playBtnLabel.text = "Play game again";

            break;


      }

      playBtn.gameObject.SetActive(true);
          }
```

Функція Меню було створена, щоб гравець розумів, що зараз відбувається в грі. Чи йому продовжувати гру, чи він програв це все показує функція. Вона створена за допомогою ігрового статусу і змінюється, при виконанні одно з них. Це все реалізовується за допомогою функції SetCurrentGameState(). В ній перевіряються всі умови гри. І якщо б хоча одна з умов виконалась, то ігровий статус змінюється відповідно до умови та викликається меню.

Лістинг коду 2.3 — функція SetCurrentGameState ()

```
if(totalEscaped >= 10)
    {
       currentState = gameStatus.gameover;
    }
    else  if(waveNumber == 0 && (RoundEscaped + TotalKilled) == 0 )
    {
       currentState = gameStatus.play;
    }
    else if (waveNumber >= totalWaves)
    {
       currentState = gameStatus.win;
    }
    else
    {
       currentState = gameStatus.next;
    }
        }
```

Отже в розділі було розглянуті основні ігрові компоненти а також розробка ігрового майданчика, які забезпечують якісну і функціональну роботу програмного забезпечення. Також ці функції надають можливість гравцю приймати важливі рішення під час самої гри, що є найважливішим при розробці комп’ютерної гри.

3 ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Роль тестування під час циклу розробки програмних засобів

Тестування програмного забезпечення —це перевірка кінцевого продукту, щоб на система програмного забезпечення на виході не була з дефектами. Вона передбачає виконання компонента програмного забезпечення або компонента системи для оцінки одного або декількох властивостей, що представляють інтерес.

Тестування програмного забезпечення також допомагає виявити помилки, прогалини або відсутні вимоги всупереч реальним вимогам. Це можна зробити або вручну, або за допомогою автоматизованих інструментів. Деякі вважають за краще говорити про тестування програмного забезпечення як білу коробку і тестування на чорному ящику.

Рисунок 3.1 — Місце тестування у створенні ПЗ

Тестування важливо [7], оскільки помилки програмного забезпечення можуть бути дорогими або навіть небезпечними. Помилки програмного забезпечення можуть потенційно викликати монетарні та людські втрати, а історія наповнена такими прикладами:

— у квітні 2015 р. Термінал Bloomberg в Лондоні зазнав аварії через програмний збій, який торкнувся більш ніж 300 000 трейдерів на фінансових ринках, це змусило уряд відкласти продаж боргу на 3 млрд фунтів;

— автомобілі Nissan повинні відкликати понад 1 мільйон автомобілів з ринку через відмову програмного забезпечення в сенсорних детекторах подушки безпеки, було повідомлено про дві аварії внаслідок цього збою програмного забезпечення;

— компанія Starbucks була змушена закрити близько 60% магазинів у США та Канаді через відмову програмного забезпечення в системі POS, в одній точці зберігають каву безкоштовно, оскільки вони не в змозі обробити операцію;

— деякі з третіх сторін роздрібної торгівлі Amazon побачили, що їхня ціна продукту знижена до 1р через програмний збій, вони залишилися з великими втратами;

Після популярності ігрової індустрії утворилось збільшення кількості студій, які розробляють ігри і оскільки більшість з них випускали ігри на “швидку руку” то вони зіткнулись з такими проблемами як баги (Баг — це жаргонізм, що означає помилку, ваду або дефект в програмі). У зв’язку з тим, щоб не втрачати прихильність своїх гравців, з’явився новий вид інженерії — тестор комп’ютерних ігор.

Тестер комп'ютерних ігор — це людина, яка в основному працює для компаній-виробників відеоігор, щоб ретельно перевіряти відеоігри до остаточної версії, яка надається громадськості. Також називаються бета-тестерами, тестери гри отримують звіт про гру, яка близька до її завершального етапу. Вони повинні грати в гру кілька разів, від початку до кінця, щоб розкрити помилки або глюки, які є в грі. Завдяки їхній роботі, ми маємо можливість грати в ігри, немаючи прикрих моментів.

Без тестера комп'ютерних ігор, помилок і збоїв насправді збільшиться в іграх, можливо, будуючи їх на неграмотному етапі і порушуючи статус компанії-виробника відеоігор.

Для кожної платформи та жанру потрібні ігрові тестери. Перед розташуванням і компанією тестери будуть грати в ігри на платформах XBox, Playstation, Nintendo Wii і PC. Рольові ігри, зокрема багатокористувацькі онлайн-ігри, екшн-ігри та ігри знань - лише деякі з жанрів ігор, які гравець повинен грати і ретельно оцінювати перед виходом.

Тестер комп'ютерних ігор отримує розширені копії ігор для того, щоб обережно протестувати гру до того, як вона вийде на публічний реліз. Вони повинні переміщатися по всіх меню, щоб переконатися, що все працює належним чином, і відводить гравця до точного підменю або етапу гри.

Якщо гра пропонує більше ніж один засіб гри, наприклад, учень, між собою або просунутий, тестер повинен грати через кожне з цих засобів від початку до кінця, розкривши всі глюки, якщо вони є. Якщо гра дозволяє грати як більш ніж один тип персонажа, тестер також повинен грати в кінці гри з кожною особистістю або аватаром.

Під час ігрового процесу, тестер намагається виявити приховані помилки, намагаючись здійснити всі можливі кроки, і проаналізувати вибір, який гравець може зробити під час звичайного ігрового процесу. Тестер може навіть вибирати певні речі в грі, які середній гравець може і не піти, наприклад, намагаючись застрягти в стіні чи перешкоді в грі. Цей тип помилки називається відсіканням. Після того, як персонаж застряг, гравець неодноразово не в змозі звільнити персонажа, викликаючи подальший геймплей неможливо. Тестери відзначають виникнення цих і того ж виду ігрових несправностей.

Після того, як помилка була виявлена, тестери гри повинні написати запис про те, що сталося разом з інструкціями, які точно пояснюють, як знайти помилку в грі. Після подання цієї інформації програмістам, тестерам може бути запропоновано протестувати переглянуті версії гри, щоб зробити постійну версією.

3.2 Різновиди циклів розробки програмних засобів

Створення програмного засобу прирівнюють до життєвого циклу. З цього порівння і утворилось визначення в програмуванні – модель життєвого циклу програмного забезпечення. Модель життєвого циклу є однією з ключових концепцій інженерії систем (SE). Життєвий цикл системи, як правило, складається з серії етапів, які регулюються набором управлінських рішень, які підтверджують, що система є достатньо зрілою, щоб залишити один етап і ввести іншу.

3.2.1 Waterfall – каскадна молель життєвого циклу

Вперше введений доктором Уінстоном У. Ройсом у статті, опублікованій в 1970 році, модель водоспаду є процесом розробки програмного забезпечення. Модель водоспаду підкреслює, що логічний прогрес кроків повинен бути здійснений протягом всього життєвого циклу розробки програмного забезпечення (SDLC), подібно до того, як каскадні кроки йдуть по поступовому водоспаду. Хоча популярність моделі водоспаду зменшилася протягом останніх років на користь більш гнучких методологій, логічний характер послідовного процесу, який використовується в методі водоспаду, не може бути заперечений, і він залишається загальним процесом проектування в галузі.

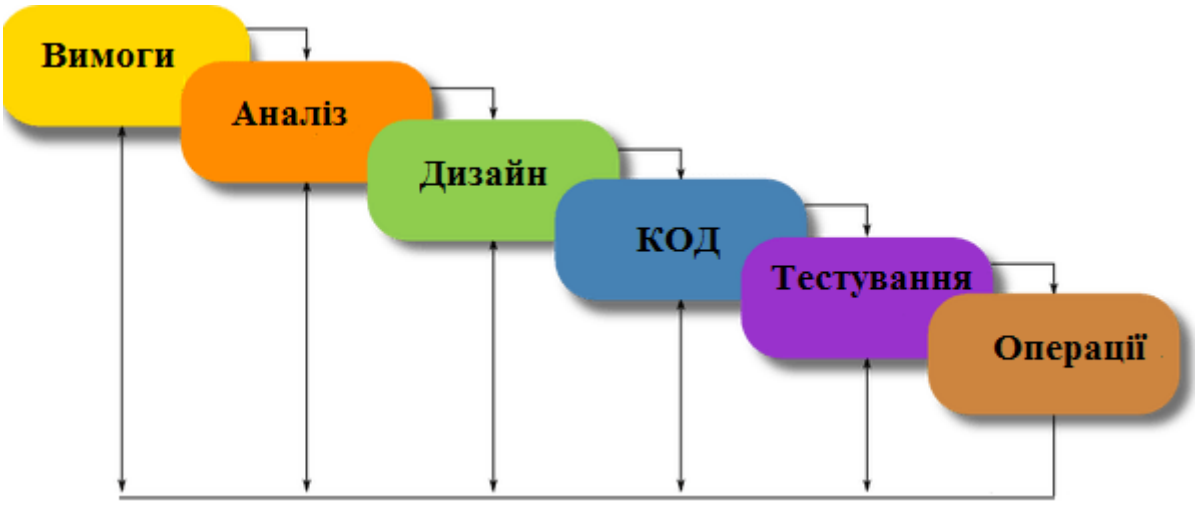


Рисунок 3.2 — Каскадна молель життєвого циклу

Насправді реалізація моделі водоспаду в рамках нового програмного проекту є досить простим процесом, багато в чому завдяки покроковому характеру самого методу, існують незначні відмінності в числах і описах кроків, задіяних у методі водоспаду, залежно від розробника, якого ви запитуєте (і навіть року, коли ви його запитуєте), незважаючи на це, концепція однакова і охоплює широкий обсяг того, що потрібно, щоб почати з ідеї та розробити повномасштабне, живу програму.

— на цьому початковому етапі потенційні вимоги до застосування методологічно аналізуються і записуються в документі специфікації, яка служить основою для всього майбутнього розвитку, результатом є, як правило, документ з вимогами, який визначає, що програма повинна робити, але не те, як вона повинна це робити;

— на цьому другому етапі система аналізується для того, щоб правильно генерувати моделі та бізнес-логіку, які будуть використовуватися в додатку;

— цей етап значною мірою охоплює технічні вимоги до проектування, такі як мова програмування, шари даних, послуги тощо, зазвичай буде створена специфікація проекту, яка окреслює, як саме технічна логіка, що розглядається, аналізується технічно;

— кодування це фактичний вихідний код, нарешті, написаний на цьому четвертому етапі, реалізуючи всі моделі, бізнес-логіку та інтеграції послуг, які були визначені на попередніх етапах;

— на цьому етапі тестувальники, бета-тестери та всі інші тестери систематично виявляють і повідомляють про проблеми, що потребують вирішення, це не рідкість для цієї фази, щоб викликати "необхідне повторення" попередньої фази кодування, для того, щоб виявлені помилки були належним чином роздавлені;

— Нарешті, програма готова до розгортання до живого середовища, етап операцій включає в себе не тільки розгортання програми, але й подальшу підтримку та підтримку, які можуть знадобитися для того, щоб підтримувати його функціональність та оновлення.

Хоча модель водоспаду спостерігається повільний поступовий припинення в останні роки на користь більш гнучких методів, він все ще може забезпечити ряд переваг, особливо для великих проектів і організацій, які вимагають жорстких етапів і термінів, доступних в цих прохолодних, каскадних водах.

Пристосовується до команд, що переходять, хоча це не обов'язково є специфічним для моделі водоспаду, використання методу водоспаду дозволяє проекту в цілому підтримувати більш детальну, надійну сферу та структуру проекту завдяки всім етапам планування та документації.[8] Це особливо добре підходить для великих команд, які можуть бачити, що члени приходять і йдуть протягом всього життєвого циклу проекту, дозволяючи тягар дизайну розміщуватися на основній документації і менше на будь-якого окремого члена команди.

Структуровані силою організації, хоча деякі можуть сперечатися, що це тягар, а не користь, факт залишається фактом, щ о модель водоспаду змушує проект, і навіть організацію будівництва сказав проект, бути надзвичайно дисциплінованим у його дизайні і структурі. Більшість великих проектів, за необхідністю, включатимуть детальну процедуру управління кожним аспектом проекту, від проектування та розробки до тестування та впровадження.

Дозволяє до ранніх змін у проектуванні. Хоча це може бути важко зробити зміни дизайну пізніше в процесі, підхід водоспаду добре піддається змінам на початку життєвого циклу. Це надзвичайно добре, коли доопрацьовують документи з специфікаціями на перших етапах роботи з командою розробників і клієнтами, оскільки зміни можуть бути зроблені негайно і з мінімальними зусиллями, оскільки до цього моменту не відбулося жодного кодування або реалізації.

Придатний для розвитку, орієнтованого на віх: Завдяки притаманній лінійній структурі проекту водоспаду, такі додатки завжди добре підходять для організацій або команд, які добре працюють за парадигмою, орієнтованою на віху і дату. З чіткими, конкретними і добре зрозумілими етапами, які всі в команді можуть зрозуміти і підготуватися, відносно просто розробити часову лінію для

всього процесу і призначити окремі маркери і віхи для кожного етапу і навіть завершення. Це не означає, що розробка програмного забезпечення часто не поширюється на затримки (оскільки вона є), але водоспад підходить до типу проекту, який потребує термінів.

Хоча деякі речі в розробці програмного забезпечення ніколи не змінюються, багато інших часто падають на другий план. Хоча початкова пропозиція д-ра Ройса про те, що зараз відома як модель водоспаду, була новаторською, коли вона була опублікована в 1970 році, за чотири десятиліття пізніше, ряд тріщин показують у броні цієї колись оголошеної моделі.Тому каскадна модель також має ряд недоліків.

Неадаптивні перешкоди при проектуванні — хоча на цю тему можна було б написати цілу книгу, найбільш вагомим аспектом моделі водоспаду є її невідповідність до всіх стадій життєвого циклу розробки. Коли тест на п'ятій стадії виявляє фундаментальний недолік у розробці системи, він вимагає не лише драматичного стрибка назад на етапах процесу, але в деяких випадках часто може призвести до руйнівної реалізації щодо легітимності всієї системи. системи. Хоча більшість досвідчених команд і розробників (справедливо) стверджують, що такі розкриття не повинні відбуватися, якщо система була належним чином спроектована в першу чергу, не всі можливості можуть бути враховані, особливо коли стадії так часто затримуються до кінця процесу.

Ігнорує зворотній зв'язок між користувачем і клієнтом середнього процесу: у зв'язку з суворим покроковим процесом, який застосовує модель водоспаду, іншим особливо важким питанням є те, що відгуки користувачів або клієнтів, які надаються пізно у циклі розробки, часто можуть бути занадто мало, занадто пізно. Хоча менеджери проектів можуть, очевидно, запровадити процес, щоб повернутися на попередню стадію через непередбачені вимоги або зміни, що надходять від клієнта, це буде як дорогою, так і багато часу, як для команди розробників, так і для клієнта.

Період відстроченого тестування, хоча більшість сучасних моделей SDLC(Software Development Lifecycle — життєвий цикл розробки програмного

забезпечення) намагаються інтегрувати тестування як фундаментальний і постійно діючий процес у процесі розвитку, модель водоспаду значною мірою ухиляється від тестування до досить пізнього періоду життєвого циклу. Це не тільки означає, що більшість помилок або навіть дизайнерські питання не будуть відкриті до самого пізнього процесу, але також заохочує практику кодового кодування, оскільки тестування є лише запізненням.

3.2.2 Spiral SDLC Model — спіральна молель життєвого циклу

Модель спіралі — це модель SDLC, яка поєднує в собі архітектуру і прототипи по етапах. Це поєднання ітеративних і водоспадних моделей SDLC зі значним акцентом на аналізі ризиків. Головне питання спіральної моделі - це визначення правильного моменту, щоб зробити крок у наступний етап. Попередні встановлені часові рамки рекомендуються як рішення цієї проблеми. Перехід на наступний етап здійснюється відповідно до плану, навіть якщо робота на попередньому етапі ще не виконана. План впроваджується на основі статистичних даних, отриманих під час попередніх проектів, навіть з досвіду особистого розробника.

Рисунок 3.3 —Спіральна молель життєвого циклу

Використовуйте модель спіралі у таких випадках:

— клієнт не впевнений у вимогах;

— основні зміни очікуються протягом циклу розробки;

— проекти з середнім або високим рівнем ризику, де важливо запобігти цим ризикам;

— новий продукт, який слід випустити на кілька етапів, щоб мати достатню кількість відгуків клієнтів.

Існує ряд переваг і недоліків, що стосуються спіральної моделі. Модель спіралі є традиційною гнучкою моделлю, яка містить 4 різних етапи: планування, аналіз ризиків, інженерія та оцінка. Програмні проекти постійно проходять через фази ітерацій, тобто спіралі. Базові спіралі, починаючи з фази планування, збирають вимоги та оцінюють ризик. Кожна наступна спіраль будується на базовій спіралі.Основними перевагами цієї моделі є:

— таким чином, підвищується рівень аналізу ризиків, уникнення ризику;

— добре для великих і критично важливих проектів;

— сильне схвалення та контроль документації;

— додаткову функціональність можна додати пізніше;

— програмне забезпечення виробляється на початку життєвого циклу програмного забезпечення;

— оцінки проекту з точки зору графіка, вартості тощо стають все більш і більш реалістичними, коли проект рухається вперед, і петлі в спіралі завершуються;

— він підходить для проектів з високим рівнем ризику, де потреби бізнесу можуть бути нестабільними;

— використовуючи це, може бути розроблений продукт, що налаштовується;

Недоліків у спіральної моделі менше, але вони також суттєві, а саме:

— можна використовувати дорогу модель;

— аналіз ризиків вимагає високоспецифічної експертизи;

— успіх проекту значною мірою залежить від фази аналізу ризику;

— Не працює добре для невеликих проектів.
— Він не підходить для проектів з низьким рівнем ризику.
— Може бути важко визначити об'єктивні, перевірені віхи.
— Спіраль може тривати нескінченно.

3.3 Тестування логіки додатку

Сьогодні на ринку є багато інструментів тестування додатків. Вони включають як платні, так і відкриті інструменти. Більш того, деякі інструменти є цільовими.

Наприклад тестування користувальницького інтерфейсу, функціональне тестування, тестування БД, тестування навантаження, продуктивність, тестування безпеки і тестування на перевірку зв'язку.[9]

Найважливішою концепцією «Тестування прикладних програм» є функціональне тестування. Таким чином, наша увага буде зосереджена на інструментах функціонального тестування.

Ось список найважливіших і найважливіших функцій, які надаються практично всіма інструментами «Функціонального тестування», а саме

— запис і відтворення;
— параметрити значення;
— редактор сценаріїв;
— запуск (тест або сценарій, з режимами налагодження та оновлення);
— звіт сеансу виконання.

Різні постачальники надають певні особливості, які роблять свій продукт унікальним для інших конкурентних продуктів. Але п'ять перерахованих вище особливостей є найбільш поширеними і можуть бути знайдені практично у всіх інструментах функціонального тестування.

Но данна робота було створена на базі платформи Unity, яка позволяє проводити тестування основних принципів чи логіки гри в собі. Тому було виконано

тестування таких компонентів як:

1) кнопка старту гри;

2) ігрової карти;

3) завершення гри;

4) перехід між рівня;

5) знищення ворога .

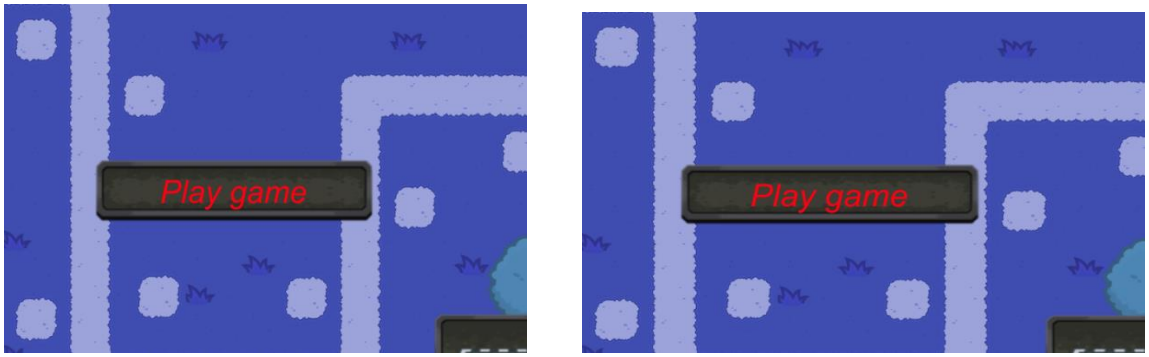


Рисунок 3.4 —Кнопка старту гри (як має бути і як є)

Рисунок 3.5 —Ігрова карта (як має бути і як є)

Рисунок 3.6 — Завершення гри (як має бути і як є)

Рисунок 3.9 — Перехід між рівнями (як має бути і як є)

Рисунок 3.10 — Знищення ворога (як має бути і як є)

3.4 Тестування компонентів

Загальні правила розробки компонента, який можна буде легко протестувати, сформовано так:

1) те, що може бачити / чути тільки людина, має бути в окремому компоненті;

2) у компонента має бути «дефолтний» поведінку. Краще вважати його еталонним;

3) один компонент – одне поведінку.

Таким чином, було внесенно коригування в реалізацію core gameplay гри жанру Tower Defense:

Деякі компоненти звертаються до Renderer через GetComponent () – це непотрібна залежність від обов'язкової присутності компонента Renderer. Вся

візуалізація повинна бути в окремих компонентах. Вказав дефолтні значення всім полям: це спростить і саму настройку компонента. Також розширив функціонал гри, ввівши Гравця:

Гравець — людина, чия поведінка полягає в передачі управління в гру, за допомогою пристроїв введення.

Тепер, для початку гри, треба натиснути кнопку «Почати гру» в UI, після чого вибрати в UI вежі для побудови із зазначенням куди їх побудувати. І коли все вежі будуть побудовані, почнеться гра.

Варто пам'ятати, що загальна поведінка ігрового об'єкта складається з поводжень компонентів.

Необхідно оцінити складність реалізації логіки всередині компонента. Якщо в ній легко припуститися помилки, варто підстрахуватися і зробити окремий юніт-тест для компонента. В інших випадках можна зробити загальну специфікацію, яка буде перевіряти поведінку ігрового об'єкта в цілому.

Підготовча робота заключаєится з доповнень для Юнити, нам знадобиться наступне:

— unityTestTools — runner NUnit-тестів від розробників Unity в середовищі редактора;

— visual Studio Tools (колишній UnityVS) —плагін для налагодження.

Для Visual Studio знадобиться SpecFlow

І почнемо ми з Unit-тестів, які є надзвичайно важливими в нашому мультимедійному додатку.

Багато додають юніт-тести безпосередньо в сам проект, але не варто обтяжувати збірку зайвим. Не так багато часу займе виділення набору тестів в окрему збірку, яка не буде включена в фінальну версію гри (build).

Після додавання UnityTestTools в Unity, потрібно відкрити рішення (сгереровані у UnityVs) і додати в нього новий проект типу «бібліотека класів», в ній і будуть наші юніт-тести компонентів. Для прикладу, назвемо збірку TowerDefenseCore.UnitTests. Важливим моментом є налагодження збірки:

— «nunit.framework.dll» і «nunit.core.dll» з Assets \ UnityTestTools \ UnitTesting \ Editor \ NUnit \ Libs;

—«UnityEngine.dll» з Library \ UnityAssemblies;

—збірку «Assembly-CSharp.dll» з вирішення;

Важливо – не треба їх копіювати локально.

Необхідно налаштувати властивості збірки, вказавши шлях виведення.

Збірка повинна бути в директорії Assets – наприклад, Assets \ Tests.

Написання юніт-тестів, виконуваних в Unity Test Runner, нічим не відрізняється від звичайних юніт-тестів. Однак, є кілька важливих моментів.

Перше: робити створення ігрового об'єкта (метод GameObject.Instantiate) не обов'язково. Ось простий тест компонента DamageApplicator:

```
[Test]
public void DamageApplicator_DirectDamage()
{
        var damageApplicator = new DamageApplicator();
        var hp = damageApplicator.DirectDamage(100, 10);
        Assert.AreEqual(90f, hp);
}
```

Рисунок 3.11 — Тестування логіки нанесення пошкоджень

Лістинг коду 3.1 — Тестування обчислення нанесення пошкоджень

```
public void DamageApplicator_DirectDamage()
{
        var damageApplicator = new DamageApplicator();
        var hp = damageApplicator.DirectDamage(100, 10);
        Assert.AreEqual(90f, hp);
}
```

DamageApplicator не має ніяких залежностей від інших компонентів, які вирішувалися б в методі Awake / Start / OnEnable. Але якщо компонент використовує Awake з цією метою, то його потрібно викликати примусово:

```
[Test]
public void Hittable_DeadOnDirectDamage()
{
        var componentsHolder = new GameObject();
            //Добавляем компонент, от которого зависит тестируемый компонент
        componentsHolder.AddComponent<DamageAplicator>();
        var hittable = componentsHolder.AddComponent<Hittable>();
        hittable.Awake();//Важно: мы не сделали инстанс объекта, потому вручную внедрим
 зависимость от DamageApplicator

        hittable.DirectDamage(50);
        Assert.AreEqual(50, hittable.HP);
        hittable.DirectDamage(50);
        Assert.AreEqual(true, hittable.IsDead);
}
```

Рисунок 3.12 — Тестування логіки нанесення пошкоджень

Лістинг коду 3.2 — Тестування обчислення нанесення пошкоджень

```
public void Hittable_DeadOnDirectDamage()
{
        var componentsHolder = new GameObject();
        componentsHolder.AddComponent<DamageAplicator>();
        var hittable = componentsHolder.AddComponent<Hittable>();
        hittable.Awake();
        hittable.DirectDamage(50);
        Assert.AreEqual(50, hittable.HP);
        hittable.DirectDamage(50);
        Assert.AreEqual(true, hittable.IsDead);
}
```

Друге: перевірка роботи компонентів, що використовують Coroutine. Тут важливо вказувати MaxTime, розраховане еталонне час виконання (якщо воно є) або в циклі while додаткову перевірку, чи пройдено тест.

```
[Test]
[MaxTime(10000)]//BeginDPS наносит 5 единиц урона каждый 0.5 сек, у цели 100HP
public void DamageInflictor_BeginDPS()
{
        var target = new GameObject();
        target.AddComponent<DamageApplicator>();
        var hittable = target.AddComponent<Hittable>();
        hittable.Awake();
        var tower = new GameObject();
        var dmger = tower.AddComponent<DamageInflictor>();
        dmger.BeginDPS(hittable);//Запускаем Coroutine
        while (dmger.inflictDamage().MoveNext())//Симулируем работу Coroutine
                Thread.Sleep(100);
        Assert.True(hittable.IsDead);
}
```

Рисунок 3.13 — Тестування логіки нанесення пошкоджень

Лістинг коду 3.3 — Тестування обчислення нанесення пошкоджень

```
 [MaxTime(10000)]
Public void DamageInflictor_BeginDPS()
{
        var target = new GameObject();
        target.AddComponent<DamageApplicator>();
        var hittable = target.AddComponent<Hittable>();
        hittable.Awake() ;
        var tower = new GameObject();
        var dmger = tower.AddComponent<DamageInflictor>();
        dmger.BeginDPS(hittable);
        while (dmger.inflictDamage().MoveNext())
                Thread.Sleep(100);
        Assert.True(hittable.IsDead);
```

}

Третій момент краще тестувати поведінку кожного компонента, а не сцени в цілому. В налаштуваннях Test Runner в опціях вкажіть Run test on a new scene.

Зберемо збірку, перемкнемося на Unity і в меню Unity Test Tools виберемо Test Runner. Як побачите, тести з збірки з'являться для запуску.

Створимо бібліотеку класів TowerDefenseCore.Specs в рішенні. Налаштуємо її:

— додамо в неї пакети: Install-Package SpecFlow.Nunit;

— додамо залежності збірки;

— «TechTalk.SpecFlow.dll» з \ packages \ SpecFlow.1.9.0 \ lib \ net35;

— «UnityEngine.dll» з Library \ UnityAssemblies;

— збірку «Assembly-CSharp.dll» з вирішення.

— необхідно налаштувати властивості збірки, вказавши шлях виведення, збірка повинна бути в директорії Assets —наприклад Assets \ Tests;

— важливо: скопіюємо «TechTalk.SpecFlow.dll» в Assets \ UnityTestTools \ UnitTesting \ Editor \ NUnit \ Libs.

Тепер в Test Runner крім тестів з TowerDefenseCore.UnitTests будуть і Steps з TowerDefenseCore.Specs.

Простий приклад Feature, перевіряючий поведінку кріпа при отриманні шкоди:

Feature: CreepLogic

Creep alive, take damage and dead.

@ creep

Creep alive, take damage and dead.

Scenario: Check creep is dead

Given Creep is alive

When Creep take damage 100

Then Creep is dead

Given Creep is alive

Given Creep is alive

Given Creep is alive

І згенеровані кроки:

```
[Binding]
public class CreepLogicSteps
{
     private Hittable _creep;
     [Given(@"Creep is alive")]
     public void GivenCreepIsAlive()
     {
         var componentsHolder = new GameObject();
         componentsHolder.AddComponent<DamageApplicator>();
         _creep = componentsHolder.AddComponent<Hittable>();
         _creep.Awake();
     }
     [When(@"Creep take damage (.*)")]
     public void WhenCreepTakeDamage(int dmg)
     {
         _creep.DirectDamage(dmg);
     }
     [Then(@"Creep is dead")]
     public void ThenCreepIsDead()
     {
         NUnitFramework.Assert.AreEqual(true, _creep.IsDead);
     }
}
```

Рисунок 3.14 — Тестування логіки нанесення пошкоджень

Лістинг коду 3.4 — Тестування обчислення нанесення пошкоджень

```
[Binding]

public class CreepLogicSteps
{
        private Hittable _creep;
        [Given(@”Creep is alive”)]
        public void GivenCreepIsAlive()
        {
        var componentsHolder = new GameObject();
        componentsHolder.AddComponent<DamageApplicator>();
```

```
        _creep = componentsHolder.AddComponent<Hittable>();
        _creep.Awake();
        }
        [When((@”Creep take damage (.*)”]
        public void WhenCreepTakeDamage (int dmg)
        {
          _creep.DirectDamage(dmg);
        }
        [Then(@”Creep is dead”)]
        public void ThenCreepIsDead()
        {
           NUnitFramework.Assert.AreEqual(true, _creep.IsDead);
        }
}
```

У Test Runner тест буде відображений з назвою CheckCreepIsDead.

3.5 Інтегральне тестування

Тестування інтеграції підтримує збірку цілісної програмної системи. Мета інтегрального тестування: узяти модулі, протестовані як елементи, і побудувати програмну структуру згідно з вимогами проекту.[10]

Тести проводяться для виявлення помилок інтерфейсу. Перерахуємо деякі категорії помилок інтерфейсу:

1) втрата даних при проходженні через інтерфейс; відсутність в модулі необхідного посилання; несприятливий вплив одного модуля на інший;

2) підфункції при об'єднанні не утворюють необхідну головну функцію; окремі (допустимі) неточності при інтеграції виходять за допустимий рівень; проблеми при роботі з глобальними структурами даних;

3) як і для відлагодження, для тестування існує два варіанти, що підтримують процес інтеграції: низхідне тестування і висхідне тестування.

Отож ми перевіряємо всі компоненти разом і одразу, щоб виявити чи з'явились якісь дефекти при компілюванні їх в одне ціле. Як бачим результат є позитивним. Гра працює і ніяких проблем не виявленно.

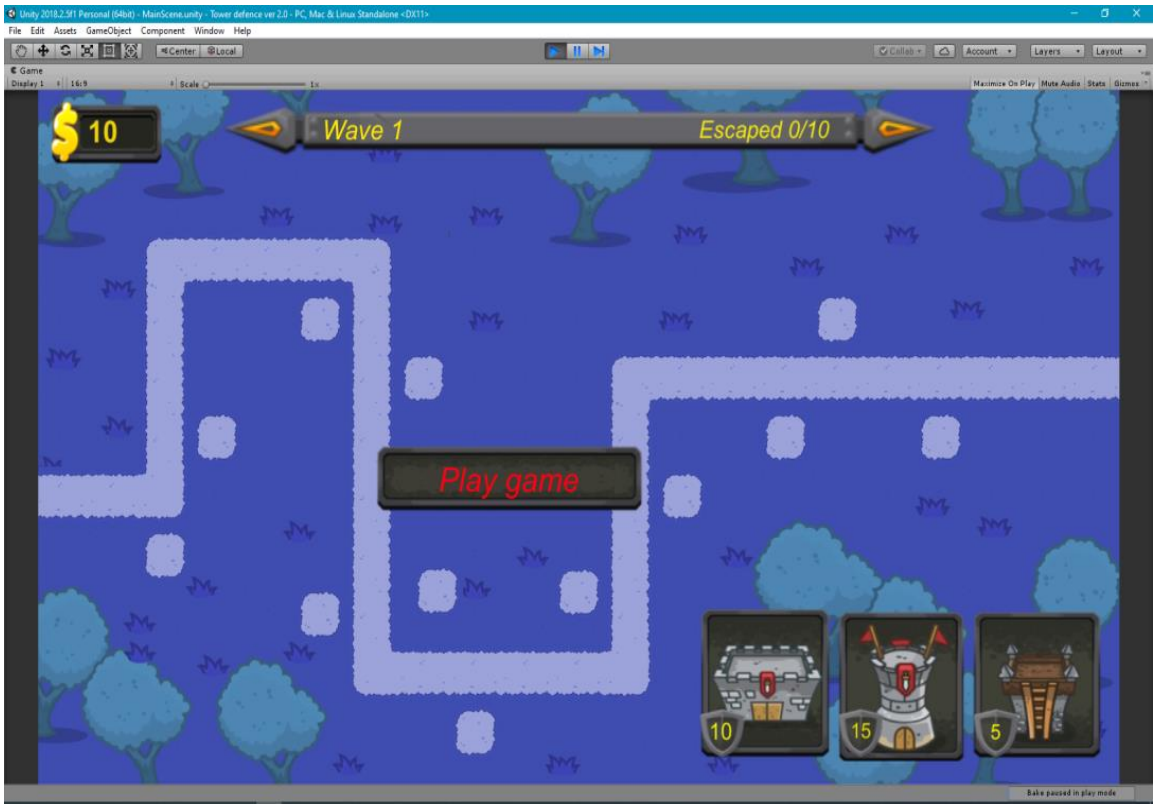


Рисунок 3.15 —Результат інтегрально тестування

3.6 Тестування дизайну графічних моделей елементів гри

Перевірка графічних моделей гри, здійснюється за допомогою методу виміру продуктивності. В процесі даного методу, графічні елементи гри проходять перевірку в програмі налагодження [11]. Вона тестує імпортовані файли графічних 3d моделей на швидкість їх оброблення двигуном Unity, тобто каже нам, чи коректно відображаються та працюють графічні моделі, чи немає втрат кадрів під час запуску та процесу гри. В результаті тестування імпортованих графічних 3d моделей захисних споруд, програма видала повідомлення про коректність їх роботи (рисунок 3.16).

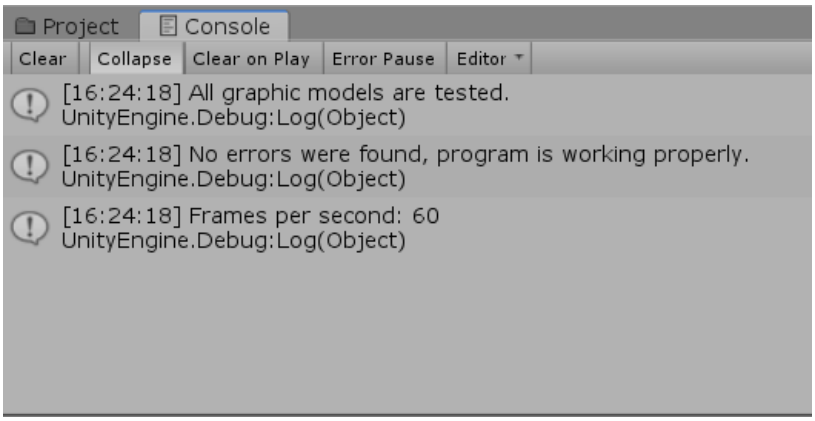


Рисунок 3.16 — Результат тестування графічних 3d моделей

Отже, графічні моделі обробляються без проблем, а це означає що під час гри, у користувача не буде виникати проблем із втратою кадрів на екрані.

3.7 Загальне тестування UI/UX дизайну розробленої гри

Для загального тестування гри, було обрано метод анкетування. Для цього було задіяно сервіс Ploys.me (сайт що дозволяє створювати опитування, та приймати в них участь анонімно). Було створено голосування із чотирма варіантами відповіді (рисунок 3.17).

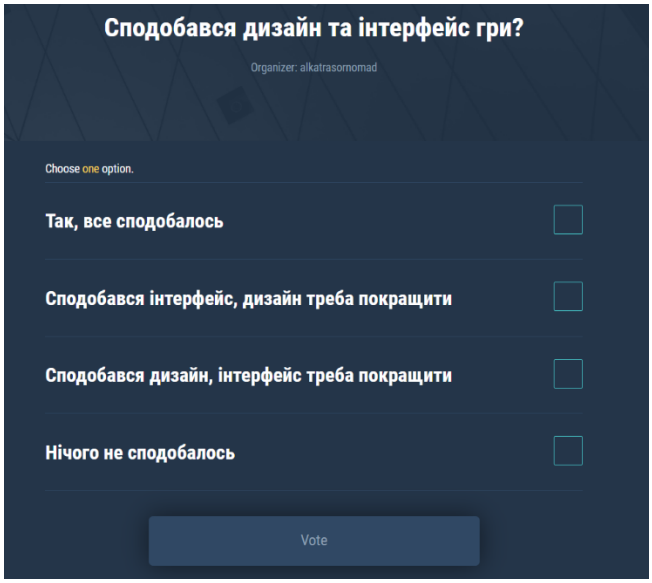


Рисунок 3.17 — Опитування для анкетного тестування usability гри

Результат даного опитування показано на рисунку 3.18. В своїй більшості має позитивні результати, проте це не означає що UI/UX дизайн гри не має не доліків. Потрібно взяти до уваги те, що аж 30% учасників опитування не обрали перший варіант відповіді, а це означає, що деякі моменти потребують допрацювання та вдосконалення.

Отже тестування є дуже важливим у створенні програмного забезпечення і мультимедійної гри. Розглянувши даний розділ, можна, дійти висновку, що жоден успішний проект, додаток, гра пройшла безліч тестів і перевірок, перш ніж випуститись в загальний доступ. І дана гра не є виключенням. Кожен компонент був перевірений окремо і дав позитивний результат.

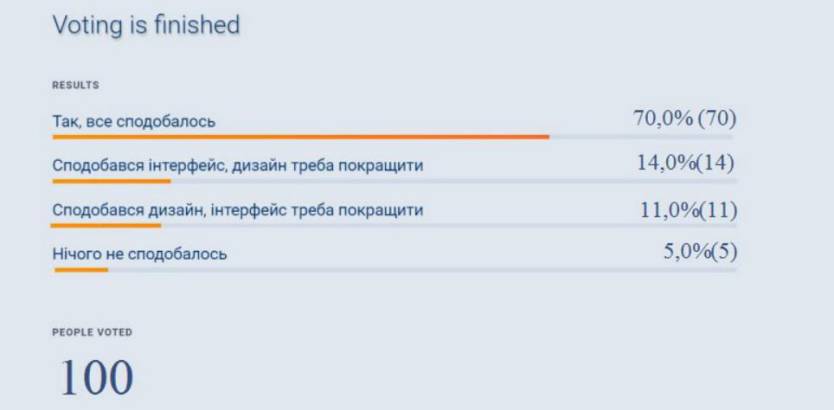


Рисунок 3.18 — Результат опитування про UI/UX дизайн гри

4 ЕКОНОМІЧНА ЧАСТИНА

Темою магістерської кваліфікаційної роботи є «Мультимедійне ігрове застосування на основі об’єктно- орієнтованого підходу на базі платфори unity3d». За цією темою в економічній частині проводяться розрахунки економічних показників на розробку програмного продукту та впровадження його на ринок аналогічних товарів.

4.1 Оцінювання комерційного потенціалу розробки

Метою проведення технологічного аудиту є оцінювання комерційного потенціалу розробки. Для проведення технологічного аудиту було залучено 2-х незалежних експертів. Такими експертами будуть СавицькаЛ.А. та Кадук О.В.

Здійснюємо оцінювання комерційного потенціалу розробки за 12-ма критеріями, наведеними у таблиці 4.1, за 5-ти бальною шкалою.

Таблиця 4.1 — Критерії оцінювання комерційного потенціалу розробки та їх бальна оцінка

Бали (за 5-ти бальною шкалою)					
Кри-те-Рій	0	1	2	3	4
Технічна здійсненність концепції:					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено роботоздат-ність продук-кту в реаль-них умовах
Ринкові переваги (недоліки):					
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому рин-ку	Продукт не має аналогів на великому ринку
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно до-рівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів

Продовження таблиці 4.1

4	Технічні та споживчі властивості продукту значно гірші,ніж в аналогів	Технічні та споживчі властивості продукту трохи гірші,ніж в аналогів	Технічні та споживчі властивості продукту на рівні аналогів	Технічні та споживчі властивості продукту трохи кращі,ніж в аналогів	Технічні та споживчі властивості продукту значно кращі,ніж в аналогів
5	Експлуатаційні витрати значно вищі,ніж в аналогів	Експлуатаційні витрати дещо вищі,ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційних витрат аналогів	Експлуатаційні витрати трохи нижчі,ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів
Ринкові перспективи					
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на ринку	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає
Практична здійсненність					
8	Відсутні фахівці як з технічної,так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання наявних фахівців	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної,так і з комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси,які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси.Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали,що використовуються у військово-промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуються у виробництві
11	Термін реалізації ідеї більший за 10років	Термін реалізації ідеї більший за 5років. Термін окупності інвестицій більше 10-ти років	Термін реалізації ідеї від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій від 3-х до 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій менше 3-х Років

Кінець таблиці 4.1

12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту,що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки повідомлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь-які регламентні обмеження на виробництво та реалізацію продукту

Результати оцінювання комерційного потенціалу розробки наведено в таблиці 4.2.

Таблиця 4.2 — Результати оцінювання комерційного потенціалу розробки

Критерії	Прізвище, ініціали, посада експерта	
	1. Савицька Л.А., к.т.н доц. кафедри ОТ	2. Кадук О.В., к.т.н, доц. кафедри ОТ
	Бали, виставлені експертами:	
1	4	4
2	0	1
3	4	4
4	2	3
5	4	3
6	2	3
7	2	3
8	3	3
9	4	4
10	4	4
11	4	3
12	4	4
Сума балів	$СБ_1 = 37$	$СБ_2 = 39$
Середньоарифметична сума балів $\overline{СБ}$	$\overline{СБ} = \frac{\sum_{1}^{3} СБ_i}{2} = 38$	

Отже, з отриманих даних таблиці 4.1 видно, що рівень потенціалу нової розробки – вище середнього. Дана розробка є конкурентоспроможною з аналогами, так як для її розробки було проаналізовано недоліки та переваги аналогових продуктів, та на основі цього впроваджену у розробку. Вона має соціологічний вплив, так як покращує ефективність контролю бюджету користувачів, що є важливим у житті людей з обмеженими фінансами.

4.2 Прогнозування витрат на виконання науково-дослідної роботи та конструкторсько-технологічної роботи

Для розробки нового програмного продукту необхідні такі витрати. Основна заробітна плата для розробників визначається за формулою (4.1):

$$З_о = \frac{М}{Т_р} \cdot t, \quad (4.1)$$

де М – місячний посадовий оклад конкретного розробника;

$Т_р$ – кількість робочих днів у місяці, $Т_р$ = 20 днів;

t – число днів роботи розробника, t = 37 днів.

Розрахунки заробітних плат для керівника і програміста в таблиці 4.3.

Таблиця 4.3 — Розрахунки основної заробітної плати

Працівник	Оклад М, грн.	Оплата за робочий день, грн.	Число днів роботи, t	Витрати на оплату праці, грн.
Науковий керівник	7500	375	4	1500
Інженер-програміст	10200	510	33	16830
Всього:				18330

Розрахуємо додаткову заробітну плату, вона розраховується як 10-12% від суми основної заробітної плати всіх розробників:

$$З_{дод} = 0{,}1 \cdot 18330 = 1833 \text{ (грн.)}$$

Нарахування на заробітну плату $Н_{зп}$ для працівників бюджетної сфери становить 22% від суми основної та додаткової заробітної плати:

$$Н_{зп} = (З_о + З_р) \cdot \frac{\beta}{100},$$

$$Н_{зп} = (18330+1833) \cdot \frac{22}{100} = 4435{,}86 \text{ (грн.)}.$$

Розрахунок амортизаційних витрат для програмного забезпечення виконується за такою формулою:

$$А=\frac{Ц \cdot Н_а}{100} \cdot \frac{Т}{12},$$

де Ц – балансова вартість обладнання, грн;

$Н_а$ – річна норма амортизаційних відрахувань % (для програмного забезпечення 25%);

Т – Термін використання (Т=2 міс.).

Таблиця 4.4 — Розрахунок амортизаційних відрахувань

Найменування програмного забезпечення	Балансова вартість, грн.	Норма амортизації, %	Термін використання, міс.	Величина амортизаційних відрахувань, грн
Персональний комп'ютер	13700	25	2	568,5
Мобільний пристрій	4300	15	2	103,2
Всього:				671.7

Розрахуємо витрати на комплектуючі. Витрати на комплектуючі розрахуємо за формулою:

$$K = \sum_1^n Н_і \cdot Ц_і \cdot К_і,$$

де n – кількість комплектуючих;

Ні – кількість комплектуючих і-го виду;

Ці – покупна ціна комплектуючих і-го виду, грн;

Кі – коефіцієнт транспортних витрат (приймемо Кі = 1,1).

Таблиця 4.5 — Витрати на комплектуючі, що були використані для розробки ПЗ.

Найменування матеріалу	Одиниці виміру	Ціна, грн.	Витрачено	Вартість витрачених матеріалів, грн.
USB — microUSB кабель	шт.	75	1	75
Пачка паперу	уп.	100	1	100
Ручка	шт.	4	1	4
Всього з урахуванням транспортних витрат				196,9

Витрати на силову електроенергію розраховуються за формулою:

$$В_е = В \cdot П \cdot Ф \cdot К_п ;$$

де В – вартість 1кВт-години електроенергії (В=2.35 грн/кВт);

П – установлена потужність комп'ютера (П=0,6кВт);

Ф – фактична кількість годин роботи комп'ютера (Ф=296 год.);

$К_п$ – коефіцієнт використання потужності ($К_п < 1$, $К_п = 0,7$).

$$В_е = 2,15 \cdot 0,6 \cdot 256 \cdot 0,7 = 231,16 \text{ (грн.)}$$

Розрахуємо інші витрати $В_{ін}$. Інші витрати $І_в$ можна прийняти як (100…300)% від суми основної заробітної плати розробників та робітників, які були виконували дану роботу, тобто:

$$В_{ін} = (\mathbf{1}..\mathbf{3}) \cdot (З_о + З_р).$$

Отже, розрахуємо інші витрати:

$$В_{ін} = 1 * (18330 + 1833) = 20163 \text{ (грн).}$$

Сума всіх попередніх статей витрат дає витрати на виконання даної частини роботи:

$$В = З_о + З_д + Н_{зп} + А + К + В_е + І_в$$

$$В = 18330+1833+4435,86+671.7+196,9+231,16+20163 = 45\ 861,62 \text{ (грн.)}$$

Розрахуємо загальну вартість наукової роботи $B_{заг}$ за формулою:

$$В_{заг} = \frac{В_{ін}}{\alpha}$$

де α – частка витрат, які безпосередньо здійснює виконавець даного етапу роботи, у відн. одиницях = 1.

$$В_{заг} = \frac{45861,62}{1} = 45861,62 \text{ грн.}$$

Прогнозування загальних витрат ЗВ на виконання та впровадження результатів виконаної наукової роботи здійснюється за формулою:

$$ЗВ = \frac{В_{заг}}{\beta}$$

де β – коефіцієнт, який характеризує етап (стадію) виконання даної роботи.

Отже, розрахуємо загальні витрати:

$$ЗВ = \frac{45861,62}{0,9} = 50957,34 \text{ грн.}$$

4.3 Прогнозування комерційних ефектів від реалізації результатів розробки

Спрогнозуємо кількісно, яку вигоду можна отримати у майбутньому від впровадження результатів виконаної наукової роботи.

Виконання наукової роботи та впровадження її результатів буде здійснюватися протягом одного року. Основні позитивні результати від впровадження розробки очікуються протягом 3-х років після її впровадження. Одним із основних позитивних результатів є зростання величини прибутку.

При реалізації результатів наукової розробки покращується якість програмного продукту, що дозволяє підвищити ціну його реалізації на 150 грн. Кількість одиниць реалізації програмного засобу також збільшиться: протягом першого року – на 700 шт., протягом другого року – ще на 450 шт., протягом третього року – ще на 300 шт.

Реалізація продукції до впровадження результатів наукової розробки складала 50 шт., а ціна – 250 грн.

Спрогнозуємо збільшення чистого прибутку підприємства від впровадження результатів наукової розробки у кожному році відносно базового за такою формулою 4.2:

$$\Delta П_i = \sum_1^n (\Delta П_я \cdot N + П_я \Delta N)_i \qquad (4.2)$$

де $\Delta П_о$ – покращення основного оціночного показника від впровадження результатів розробки у даному році. Зазвичай таким показником може бути ціна одиниці нової розробки;

N – основний кількісний показник, який визначає діяльність підприємства у даному році до впровадження результатів наукової розробки;

ΔN – покращення основного кількісного показника діяльності підприємства від впровадження результатів розробки;

$Ц_о$ – основний оціночний показник, який визначає діяльність підприємства у даному році після впровадження результатів наукової розробки;

n – кількість років, протягом яких очікується отримання позитивних результатів від впровадження розробки.

λ – коефіцієнт, який враховує сплату податку на додану вартість. Ставка податку на додану вартість дорівнює 20%, а коефіцієнт 0,8333.

ρ – коефіцієнт, який враховує рентабельність продукту. Рекомендується приймати = 0,2...0,3;

υ – ставка податку на прибуток. υ = 18%.

Тоді, збільшення чистого прибутку підприємства протягом першого року складе:

$$\Delta П_1 = [150 \cdot 50 + (250 + 150) \cdot 700] \cdot 0{,}8333 \cdot 0{,}2 \cdot \left(1 - \frac{18}{100}\right) = 39290{,}09 \text{ грн}$$

Протягом другого року:

$$\Delta П_2 = [150 \cdot 50 + (250 + 150) \cdot (700 + 450)] \cdot 0{,}8333 \cdot 0{,}2 \cdot \left(1 - \frac{18}{100}\right) = 61\,710 \text{ грн}$$

Протягом третього року:

$$\Delta П_3 = [150 \cdot 50 + (250 + 150) \cdot (700 + 450 + 300)] \cdot 0{,}8333 \cdot 0{,}2 \cdot \left(1 - \frac{18}{100}\right) = 79\,900 \text{ грн}$$

Отже, протягом трьох років підприємство може розраховувати на збільшення чистого прибутку від реалізації наукової розробки.

4.4 Розрахунок ефективності вкладених інвестицій та період їх окупності

Визначимо абсолютну і відносну ефективність вкладених інвестором інвестицій та розрахуємо термін окупності.

Абсолютна ефективність $E_{абс}$вкладених інвестицій розраховується за формулою:

$$E_{абс} = (ПП - PV),$$

де ПП – приведена вартість всіх чистих прибутків, які отримає підприємство (організація) від реалізації результатів наукової розробки, грн.;

PV – теперішня вартість інвестицій PV = ЗВ = 50957,34грн.

Рисунок, що характеризує рух платежів (інвестицій та додаткових прибутків) буде мати вигляд, рисунок 4.1.

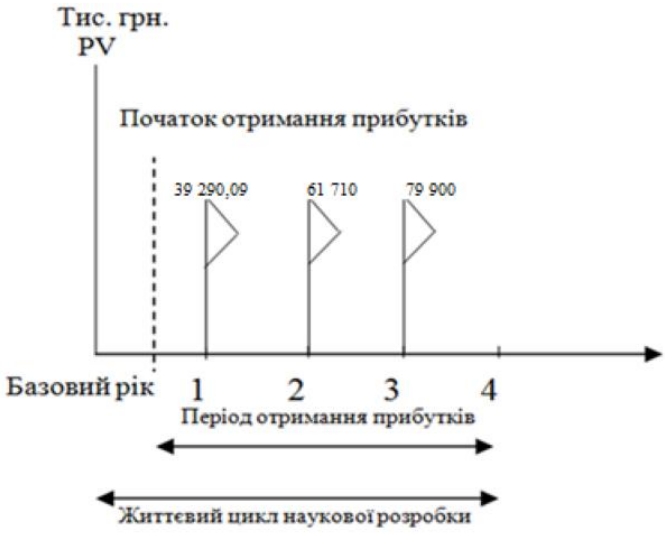


Рисунок 4.1 — Вісь часу з фіксацією платежів, що мають місце під час розробки та впровадження результатів наукової роботи

У свою чергу, приведена вартість всіх чистих прибутків ПП розраховується за формулою:

$$ПП = \sum_1^T \frac{\Delta П_i}{(1+\tau)^t},$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої НДДКР, грн;

т – період часу, протягом якого виявляються результати впровадженої НДДКР, 3 роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні; для України цей показник знаходиться на рівні 0,1;

t – період часу (в роках) від моменту отримання чистого прибутку до точки “0”.

Отже, розрахуємо вартість чистого прибутку:

$$ПП = \frac{50957{,}34}{(1+0{,}1)^0} + \frac{39290{,}09}{(1+0{,}1)^2} + \frac{61710}{(1+0{,}1)^3} + \frac{79900}{(1+0{,}1)^4} = 184\,542{,}28 \text{ грн.}$$

Тоді розрахуємо $E_{абс}$:

$$Е_{абс} = 184\,542{,}28 - 50\,957{,}34 = 133\,584{,}94 \text{ грн.}$$

Оскільки $E_{абс} > 0$, то вкладання коштів на виконання та впровадження результатів НДДКР буде доцільним.

Розрахуємо відносну (щорічну) ефективність вкладених в наукову розробку інвестицій $Е_в$ за формулою:

$$Е_в = \sqrt[T]{1 + \frac{Е_{абс}}{PV}} - 1,$$

де $Е_{абс}$ – абсолютна ефективність вкладених інвестицій, грн;

PV – теперішня вартість інвестицій PV = ЗВ, грн;

Т – життєвий цикл наукової розробки, роки.

Тоді будемо мати:

$$E_в = \sqrt[з]{1 + \frac{133584,94}{50957,34}} - 1 = 0,53 \text{ або } 53\ \%$$

Далі, розраховану величина $E_в$ порівнюємо з мінімальною (бар'єрною) ставкою дисконтування $\tau_{мін}$, яка визначає ту мінімальну дохідність, нижче за яку інвестиції вкладатися не будуть. У загальному вигляді мінімальна (бар'єрна) ставка дисконтування $\tau_{мін}$ визначається за формулою:

$$\tau = d + f,$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2020 році в Україні d = 0,2;

f – показник, що характеризує ризикованість вкладень, величина f = 0,1.

$$\tau = 0,2 + 0,1 = 0,3$$

Оскільки $E_в$ = 53% > $\tau_{мін}$ = 0,3 = 30%, то інвестор буде зацікавлений вкладати гроші в дану наукову розробку.

Термін окупності вкладених у реалізацію наукового проекту інвестицій $T_{ок}$ розраховується за формулою:

$$T_{ок} = \frac{1}{E_в},$$

Якщо $T_{ок}$< 3…5-ти років, то фінансування наукової розробки є доцільним.

$$T_{ок} = \frac{1}{0,53} = 1,8 \text{ років}$$

Обрахувавши термін окупності даної наукової розробки, можна зробити висновок, що фінансування даної наукової розробки буде доцільним, оскільки Ток<3 років.

Отже, результати проведених розрахунків дають можливість зробити висновок про доцільність розробки та впровадження нашої наукової роботи. Це підтверджують такі показники як:

— абсолютна ефективність вкладених інвестицій, яка дорівнює 133 584,94 грн., що є більшим 0 і вказує на те, що інвестор може бути зацікавленим у нашій розробці;

— відносна ефективність наукової розробки становить 53%, що є вищим за мінімальну ставку дисконтування (30%), тому вкласти кошти у нашу розробку є вигідніше, ніж покласти кошти на депозит;

— термін окупності вкладених у реалізацію наукового проекту інвестицій складе 1,8 років, що є менше 3-ох і вказує швидку окупність інвестицій.

Крім того, розраховано, що наукова розробка приноситиме підприємству додатковий прибуток протягом 3-х років за рахунок покращення її якості порівняно з існуючими аналогами. Усе це, узяте разом, забезпечує прийняття рішення про доцільність виготовлення нового продукту.

ВИСНОВКИ

Під час виконання даної магістерської кваліфікаційної роботи було детально проаналізовано, узгоджено план робіт, були обрані найліпші інструменти для створення мультимедійної гри, виконано порівняння з аналогами, проаналізовано методи розв’язання задачі.

У ході проектування програмного модуля було створено основну логіку гри на базі об’єктно-орієнтованого методу. Були проаналізовані основні методи програмування.

Проаналізовані мови програмування для розробки та обрано мову C# для розробки мультимедійного додатку. Проведено порівняльний аналіз інтегрованих середовищ розробки та визначено, що середовище Unity є найоптимальнішим для розробки додатку.

Проаналізовані основні методики тестування, різновиди циклів розробки програмних засобів, було проводено тести кожного компонента і логіки гри,а також інтегральне тестування, що підтвердило правильну роботу основних функціональних одиниць програмного модуля та готовність його до експлуатації гравцем.

Отже, в ході написання магістерської кваліфікаційної дипломній роботи було отримано практичні навички у плануванні процесу розробки, програмування на базі мови C#, отримані навички розробки фреймворку, розробки програмної логіки гри.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Синтаксис регулярных выражений [Електронний ресурс]. — Режим доступу: http://regexpstudio.com/ru/regexp_syntax.html.
2. Multiple-Document Interface (MDI) [Електронний ресурс]. — Режим доступу: https://msdn.microsoft.com/en-us/library/xyhh2e7e(v=vs.110).aspx.
3. Страуструп Б. Язык программирования C++. Специальное издание = The C++ programming language. Special edition./Б. Страуструп —М.: Бином-Пресс, 2007. —1104 с. —ISBN 5-7989-0223-4
4. Стефенс Д. Р. C++. Сборник рецептов./ Д. Р. Стефенс —КУДИЦ-ПРЕСС, 2007. —624 с. —ISBN 5-91136-030-6
5. Sells, Chris. Windows Forms Programming in C# (1st ed.), Addison-Wesley Professional, ст. 39.
6. Трояновська Т. І. Інформаційна технологія доставки контенту у системах комп’ютеризованої підготовки спеціалістів. // Гороховський О. І., Трояновська Т. І., Азаров О. Д. Монографія. Вінниця : ВНТУ, 2016.–160 с.
7. Рейсиг Д. Инструменты отладки и тестирования/Д.Рейсиг. — СПб.: Питер, 2008. — С. 76. — (Библиотека программиста). — 2500 экз. — ISBN 978-5-91180-904-1.
8. Бейзер Б. Тестирование чёрного ящика. Технологии функционального тестирования программного обеспечения и систем./ Б. Бейзер. — СПб.: Питер, 2004. — 320 с. — ISBN 5-94723-698-2
9. Трояновська Т. І. Метод покращення візуальним керуванням галереями графічних файлів / Т. І. Трояновська, Л. А. Савицька, І. А. Жарий // Інформаційні технології та комп’ютерна інженерія. —Вінниця, 2016. —№3, с. 33–37.
10. Трояновська Т. І. Методи та засоби популяризації комерційних веб–ресурсів / Т. І. Трояновська, Л. А. Савицька, В. Ю. Тарануха // Інформаційні технології та комп’ютерна інженерія. —Вінниця, 2017. —№2, С. 23–30.

11. Трояновська Т. І. Розробка комп’ютерної підсистеми аналізу та формування предметно-орієнтованої домінанти студента системи дистанційного навчання / Т. І. Трояновська // Вісник Черкаського державного технологічного університету. —2007. —Вип. 3–4. —С. 41–46. —ISSN 2306–4412.
12. Трояновська Т. І. Інформаційна технологія побудови системи комп’ютеризованої підготовки спеціалістів / Т. І. Трояновська // «Новини на научния прогрес» : IX Міжнародна науково–практична конференція, 17–25 серпня 2013 р. : тези доповідей. —София : «Бял ГРАД–БГ», 2013. —С. 36–42. —ISBN 978–966–8736–05–6.

ДОДАТОК А

Технічне завдання

Міністерство освіти та науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

ЗАТВЕРДЖЕНО
Завідувач кафедри ОТ
_____проф., д.т.н., О.Д. Азаров
«___»_____2020 р.

Технічне завдання
до магістерської кваліфікаційної роботи на тему:
«Мультимедійне ігрове застосування на основі об'єктно-орієнтованого підходу на базі платфори unity3d»
08-23.МКР.009.00.000 ТЗ

Керівник роботи:
_____ доц., к.т.н. Савицька Л.А.
«___»_____2020 р.
Виконав: студент гр. 1КІ-19м
Омельченко Сергій Сергійович
«___»_____2020 р.

Вінниця 2020

1 Підстава для виконання магістерської кваліфікаційної роботи (МКР)

а) актуальність досліджень;

б) наказ про затвердження теми дипломної роботи.

2 Мета і призначення МКР

а) мета — є зменшення операційних витрат на процес програмування логіки мультимеійної гри.;

б) призначення розробки — процес написання логіки мультимедійної гри об’єктно-орієнтованим методом.

3 Вихідні дані для виконання МКР

— технічний опис програмного застосунку;

— мова програмування C#;

— середовище розробки Unity;

4 Вимоги до виконання МКР

— огляд і аналіз методів розробки;

— аналіз та порівняння існуючих IT платформ;

— розробка мультимедійного додатку на базі Unity.

5 Етапи МКР та очікувані результати наведено у таблиці А.1

Таблиця А.1— Етапи розробки МКР

№ Етапу	Назва етапу	Термін виконання		Очікувані результати
		початок	кінець	
1	Пошук та огляд інформаційних джерел	15.09.20р.	01.10.20р.	Розділ 1
2	Програмна реалізація додатку	02.10.20р.	25.10.20р.	Розділ 2
3	Тестування розробленого програмного засобу	26.10.20р.	27.10.20р.	Розділ 3
4	Економічна частина	15.11.20р.	18.11.20р.	Розділ 4

6 Матеріали, що подаються до захисту МКР

Пояснювальна записка МКР, графічні і ілюстративні матеріали, протокол попереднього захисту МКР на кафедрі, відгук наукового керівника, відгук опонента, протоколи складання державних екзаменів, анотації до МКР українською та іноземною мовами, нормоконтроль про відповідність оформлення МКР діючим вимогам.

7 Порядок контролю виконання та захисту МКР

Виконання етапів графічної та розрахункової документації МКР контролюється науковим керівником згідно зі встановленими термінами. Захист МКР відбувається на засіданні Державної екзаменаційної комісії, затвердженою наказом ректора.

8 Вимоги до оформлення МКР

Вимоги викладені в МЕТОДИЧНИХ ВКАЗІВКАХ до дипломного проектування, ДСТУ 3008-95, ДСТУ 3974-2000 «Правила виконання дослідно-конструкторських робіт. Загальні положення» та діючого ГОСТ 2.114-95 ЕСКД.

9 Вимоги щодо технічного захисту інформації в МКР з обмеженим доступом відсутні.

ДОДАТОК Б

Лістинг програми

```
using UnityEngine;
public class Enemy : MonoBehaviour {
    [SerializeField]
    Transform exit;
    [SerializeField]
    Transform[] wayPoints;
    [SerializeField]
    float navigation;
    [SerializeField]
    int health;
    [SerializeField]
    int rewardAmount;

    int target = 0;
    Transform enemy;
    Collider2D enemyCollider;
    Animator anim;
    float navigationTime = 0;
    bool isDead = false;
    public bool IsDead

    {
        get
        {
            return isDead;
        }
    }
        // Use this for initialization
        void Start () {
        enemy = GetComponent<Transform>();
        enemyCollider = GetComponent<Collider2D>();
        anim = GetComponent<Animator>();
        Manager.Instance.RegisterEnemy(this);
        }
        // Update is called once per frame
        void Update () {
        if (wayPoints != null && isDead == false)
        {
            navigationTime += Time.deltaTime;
            if (navigationTime > navigation)
            {
                if (target < wayPoints.Length)
```

```
            {
                enemy.position= Vector2.MoveTowards(enemy.position,
wayPoints[target].position, navigationTime);
            }
            else
            {
                enemy.position = Vector2.MoveTowards(enemy.position, exit.position,
navigationTime);
            }
            navigationTime = 0;
        }
    }
     }
  void OnTriggerEnter2D(Collider2D collision)
   {
      if (collision.tag == "MoveingPoint")
      {
         target += 1;
      }
      else if (collision.tag == "Finish")
      {
         Manager.Instance.RoundEscaped += 1;
         Manager.Instance.TotalEscaped += 1;
         Manager.Instance.UnregisterEnemy(this);
         Manager.Instance.IsWaveOver();
      }
      else if (collision.tag == "Projectile")
      {
         Projectile newP = collision.gameObject.GetComponent<Projectile>();
         EnemyHit(newP.AttackDamage);
         Destroy(collision.gameObject);
      }
   }

   public void EnemyHit (int hitPoints)
   {
      if (health - hitPoints > 0)
      {
         health -= hitPoints;
         //hurt
         anim.Play("Hurt");
      }
      else
      {
         // die
```

```
            anim.SetTrigger("didDie");
            Die();
        }
    }
    public void Die()
    {
        isDead = true;
        enemyCollider.enabled = false;
        Manager.Instance.TotalKilled += 1;
        Manager.Instance.addMoney(rewardAmount);
        Manager.Instance.IsWaveOver();
    }
}

public class TowerManager : Loader<TowerManager>
{
    public TowerBtn towerBtnPressed { get; set; }
    SpriteRenderer spriteRenderer;
    private List<TowerControl> TowerList = new List<TowerControl>();
    private List<Collider2D> BuildList = new List<Collider2D>();
    private Collider2D buildTile;
    // Use this for initialization
    void Start()
    {
        spriteRenderer = GetComponent<SpriteRenderer>();
        buildTile = GetComponent<Collider2D>();
        spriteRenderer.enabled = false;
    }
    // Update is called once per frame
    void Update()
    {
        if(Input.GetMouseButtonDown(0))
        {
            Vector2 mousePoint =
Camera.main.ScreenToWorldPoint(Input.mousePosition);
            RaycastHit2D hit = Physics2D.Raycast(mousePoint, Vector2.zero);
            if(hit.collider.tag == "TowerSide" )
            {
                buildTile = hit.collider;
                buildTile.tag = "TowerSideFull" ;
                RegisterBuildSite(buildTile);
                 PlaceTower(hit);
            }
        }
```

```
    if (spriteRenderer.enabled)
    {
        FollowMouse();
    }
}
public void RegisterBuildSite (Collider2D buildTag)
{
    BuildList.Add(buildTag);
}
public void RegisterTower(TowerControl tower)
{
    TowerList.Add(tower);
}
public void RenameTagBuildSite()
{
    foreach (Collider2D buildTag in BuildList)
        {
        buildTag.tag = "TowerSide";
    }
    BuildList.Clear();
}
public void DestroyAllTowers()
{
    foreach (TowerControl tower in TowerList)
        {
        Destroy(tower.gameObject);
    }
    TowerList.Clear();
}
public void PlaceTower(RaycastHit2D hit)
{
    if(!EventSystem.current.IsPointerOverGameObject() && towerBtnPressed != null)
    {
        TowerControl newTower = Instantiate(towerBtnPressed.TowerObject);
          newTower.transform.position = hit.transform.position;
        BuyTower(towerBtnPressed.TowerPrice);
        RegisterTower(newTower);
        DisableDrag();
    }
}
public void BuyTower(int price)
{
    Manager.Instance.subtractMoney(price);
}
```

```
    public void SelectedTower(TowerBtn towerSelected)
    {
        if(towerSelected.TowerPrice <= Manager.Instance.TotalMoney)
        {
            towerBtnPressed = towerSelected;
            EnableDrag(towerBtnPressed.DragSprite);
        }
    }
    public void FollowMouse()
    {
        transform.position = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        transform.position = new Vector2(transform.position.x, transform.position.y);
    }
    public void EnableDrag(Sprite sprite)
    {
        spriteRenderer.enabled = true;
        spriteRenderer.sprite = sprite;
    }
    public void DisableDrag()
    {
        spriteRenderer.enabled = false;
    }
}

public class Loader<T> : MonoBehaviour where T : MonoBehaviour
{
    private static T instance;
    public static T Instance {
        get {
            if (instance == null)
            {
                instance = FindObjectOfType<T>();
            }
            else if (instance != FindObjectOfType<T>())
            {
                Destroy(FindObjectOfType<T>());
            }
            DontDestroyOnLoad(FindObjectOfType<T>());
            return instance;
        }
    }
}

public enum gameStatus
```

```
{
    next, play, gameover, win
}
public class Manager : Loader<Manager>
{
    [SerializeField]
    int totalWaves = 10;
    [SerializeField]
    Text totalMoneyLabel;
    [SerializeField]
    Text currentWave;
    [SerializeField]
    Text totalEscapeLabel;
    [SerializeField]
    Text playBtnLabel;
    [SerializeField]
    Button playBtn;
    [SerializeField]
    GameObject spawnPoint;
    [SerializeField]
    GameObject[] enemies;
    [SerializeField]
    int totalEnemies = 5;
    [SerializeField]
    int enemiesPerSpawn;
    int waveNumber = 0;
    int totalMoney = 10;
    int totalEscaped = 0;
    int roundEscaped = 0;
    int totalKilled = 0;
    int whichEnemyToSpawn = 0;
    gameStatus currentState = gameStatus.play;
    public List<Enemy> EnemyList = new List<Enemy>();
    const float spawnDelay = 0.5f;
    public int TotalEscaped
    {
        get
        {
            return totalEscaped;
        }
        set
        {
            totalEscaped = value;
        }
```

```
}
public int RoundEscaped
{
    get
    {
        return roundEscaped;
    }
    set
    {
        roundEscaped = value;
    }
}
public int TotalKilled
{
    get
    {
        return totalKilled;
    }
    set
    {
        totalKilled = value;
    }
}


public int TotalMoney
{
    get
    {
        return totalMoney;
    }
    set
    {
        totalMoney = value;
        totalMoneyLabel.text = TotalMoney.ToString();
    }
}
// Use this for initialization
void Start()
{

    playBtn.gameObject.SetActive(false);
    ShowMenu();
}
```

```
private void Update()
{
   HandleEscape();
}


IEnumerator Spawn()
{
   if (enemiesPerSpawn > 0 && EnemyList.Count < totalEnemies)
   {
      for (int i = 0; i < enemiesPerSpawn; i++)
      {
         if (EnemyList.Count < totalEnemies)
         {
            GameObject newEnemy = Instantiate(enemies[1]) as GameObject;
            newEnemy.transform.position = spawnPoint.transform.position;
      }
      yield return new WaitForSeconds(spawnDelay);
      StartCoroutine(Spawn());
   }
}
public void RegisterEnemy(Enemy enemy)
{
   EnemyList.Add(enemy);
}
public void UnregisterEnemy(Enemy enemy)
{
   EnemyList.Remove(enemy);
   Destroy(enemy.gameObject);
}
public void DestroyEnemies()
{
   foreach (Enemy enemy in EnemyList)
   {
      Destroy(enemy.gameObject);
   }
   EnemyList.Clear();
}
public void addMoney(int amount)
{
   TotalMoney += amount;
}
```

```
public void subtractMoney(int amount)
{
   TotalMoney -= amount;
}
public void IsWaveOver()
{
   totalEscapeLabel.text = "Escaped" + TotalEscaped + "/ 10";
   if ((RoundEscaped + TotalKilled) == totalEnemies)
   {
      SetCurrentGameState();
      ShowMenu();
   }
}
public void SetCurrentGameState()
{
   if(totalEscaped >= 10)
   {
      currentState = gameStatus.gameover;
   }
   else  if(waveNumber == 0 && (RoundEscaped + TotalKilled) == 0 )
   {
      currentState = gameStatus.play;
   }
   else if (waveNumber >= totalWaves)
   {
      currentState = gameStatus.win;
   }
   else
   {
      currentState = gameStatus.next;
   }
}
public void PlayButtonPressed()
{
   switch(currentState)
   {
      case gameStatus.next:
         waveNumber += 1;
         totalEnemies += waveNumber + 1;
         break;
      default:
         totalEnemies = 5;
         TotalEscaped = 0;
         TotalMoney = 10;
```

```
            TowerManager.Instance.DestroyAllTowers();
            TowerManager.Instance.RenameTagBuildSite();
            totalMoneyLabel.text = TotalMoney.ToString();
            totalEscapeLabel.text = "Escaped" + TotalEscaped + "/ 10 ";
            break;
        }
        DestroyEnemies();
        TotalKilled = 0;
        RoundEscaped = 0;
        currentWave.text = "Wave" + (waveNumber + 1);
        StartCoroutine(Spawn());
        playBtn.gameObject.SetActive(false);
    }
    public void ShowMenu()
    {
        switch(currentState)
        {
            case gameStatus.gameover:
                playBtnLabel.text = "Play Again";
                break;
            case gameStatus.next:
                playBtnLabel.text = "Next wave";
                break;
            case gameStatus.play:
                playBtnLabel.text = "Play game";
                break;
            case gameStatus.win:
                playBtnLabel.text = "Play game again";
                break;
        }

        playBtn.gameObject.SetActive(true);
    }
    private void HandleEscape()
    {
        if(Input.GetKeyDown(KeyCode.Escape))
        {
            TowerManager.Instance.DisableDrag();
            TowerManager.Instance.towerBtnPressed = null;
        }
    }
}

public class TowerBtn : MonoBehaviour {
```

```
    [SerializeField]
    TowerControl towerObject;
    [SerializeField]
    Sprite dragSprite;
    [SerializeField]
    int towerPrice;
    public TowerControl TowerObject
    {
        get
        {
            return towerObject;
        }
    }
    public Sprite DragSprite
    {
        get
        {
            return dragSprite;
        }
    }
    public int TowerPrice
    {
        get
        {
            return towerPrice;
        }
    }
}

public class TowerControl : MonoBehaviour {
    [SerializeField]
    float timeBetweenAttacks;
    [SerializeField]
    float attackRadius;
    [SerializeField]
    Projectile projectile;
    Enemy targetEnemy = null;
    float attackCounter;
    bool isAttacking = false;
    // Use this for initialization
    void Start () {
        }
        // Update is          called once per frame
        void Update () {
```

```
        attackCounter -= Time.deltaTime;

        if(targetEnemy  == null || targetEnemy.IsDead)
        {
            Enemy nearestEnemy = GetNearestEnemy();
            if (nearestEnemy != null && Vector2.Distance(transform.localPosition,
nearestEnemy.transform.localPosition)  <= attackRadius)
            {
                targetEnemy = nearestEnemy;
            }
        }
        else
        {
            if(attackCounter <= 0)
            {
                isAttacking = true;

                attackCounter = timeBetweenAttacks;
            }
            else
            {
                isAttacking = false;
            }
            if (Vector2.Distance(transform.localPosition,
targetEnemy.transform.localPosition) > attackRadius)
            {
                targetEnemy = null;
            }
        }
          }
    public void FixedUpdate()
    {
        if (isAttacking == true)
        {
            Attack();
        }
    }
    public void Attack()
    {
        isAttacking = false;
        Projectile newProjectile = Instantiate(projectile) as Projectile;
        newProjectile.transform.localPosition = transform.localPosition;
        if (targetEnemy == null)
        {
```

```
            Destroy(newProjectile);
        }
        else
        {
            //move projectile to enemy
            StartCoroutine(MoveProjectile(newProjectile));
        }
    }
    IEnumerator MoveProjectile (Projectile projectile)
    {
        while(GetTargetDistance(targetEnemy) > 0.20f && projectile !=  null &&
targetEnemy != null)
        {
            var dir = targetEnemy.transform.localPosition - transform.localPosition;
            var angleDirection = Mathf.Atan2(dir.y, dir.x) * Mathf.Rad2Deg;
            projectile.transform.rotation = Quaternion.AngleAxis(angleDirection,
Vector3.forward);
            projectile.transform.localPosition =
Vector2.MoveTowards(projectile.transform.localPosition,
targetEnemy.transform.localPosition, 5f * Time.deltaTime);
            yield return null;
        }
        if (projectile != null || targetEnemy == null)
        {
            Destroy(projectile);
        }
    }
    private float GetTargetDistance(Enemy thisEnemy)
    {
        if(thisEnemy == null)
        {
            thisEnemy = GetNearestEnemy();
            if (thisEnemy == null)
            {
                return 0f;
            }
        }
        return Mathf.Abs(Vector2.Distance(transform.localPosition,
thisEnemy.transform.localPosition));
    }
    private List<Enemy> GetEnemiesInRange()
    {
        List<Enemy> enemiesInRange = new List<Enemy>();
        foreach (Enemy enemy in Manager.Instance.EnemyList)
```

```
        {
            if(Vector2.Distance(transform.localPosition, enemy.transform.localPosition) <=
attackRadius )
            {
                enemiesInRange.Add(enemy);
            }
        }
        return enemiesInRange;
    }
   private Enemy GetNearestEnemy()
   {
        Enemy nearestEnemy = null;
        float smallestDistance = float.PositiveInfinity;
        foreach (Enemy enemy in GetEnemiesInRange() )
            {
            if (Vector2.Distance(transform.localPosition, enemy.transform.localPosition) <
smallestDistance)
            {
                smallestDistance = Vector2.Distance(transform.localPosition,
enemy.transform.localPosition);
                nearestEnemy = enemy;
            }
        }
        return nearestEnemy;
   }
}

public enum projectileType
{
    rock, arrow, fireball
};
public class Projectile : MonoBehaviour {
    [SerializeField]
    int attackDamage;
    [SerializeField]
    projectileType pType;
    public int AttackDamage
    {
        get
        {
            return attackDamage;
        }
    }
    public projectileType PType
        {
```

```
public class Projectile : MonoBehaviour {
    [SerializeField]
    int attackDamage;
    [SerializeField]
    projectileType pType;
    public int AttackDamage
    {
        get
        {
            return attackDamage;
        }
    }
    public projectileType PType
        {
          get
            {
            return pType;
             }
        }
}
```

ДОДАТОК В

Інтерфейс розробленого мультимедійного додатку

Рисунок В.1 — Інтерфейс додатку

ДОДАТОК Г

Sprite гри

Рисунок Г.1 — Sprite ворогів

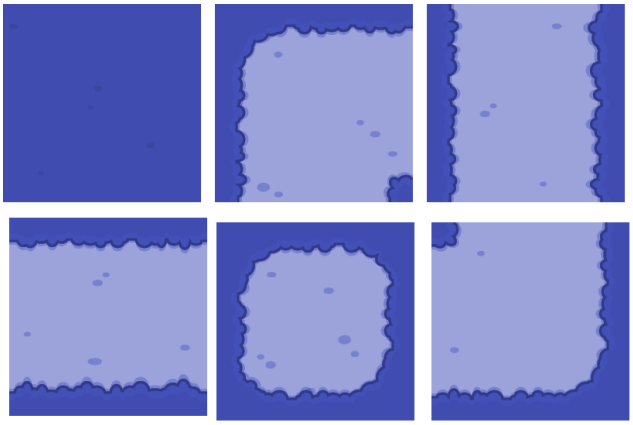


Рисунок Г.2 — Sprite доріг

Рисунок Г.3 — Sprite веж