ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУШКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ

ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

Пояснювальна записка

до бакалаврської роботи

на ступінь вищої освіти бакалавр

на тему: **«Розробка гри у жанрі Tower Defense на Unity за допомогою мови програмування C#»**

Виконав: студент 4 курсу, групи ПД-42

спеціальності

121 Інженерія програмного забезпечення_

(шифр і назва спеціальності)

___Александренко М.А.____________

(прізвище та ініціали)

Керівник Дібрівний О.А.______________

(прізвище та ініціали)

Рецензент___________________________

(прізвище та ініціали)

Київ – 2021

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

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

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп’ютерної інженерії
Негоденко О. В.
«____» ___________2021 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

____
(прізвище, ім‘я, по батькові)

1.Тема роботи: «Тема роботи за наказом»______________________________
____

Керівник роботи____,
(прізвище, ім‘я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої освіти від «___»________ року №__.

2. Строк подання студентом роботи______________________________________

3. Вихідні дані до роботи:

3.1 Вимоги до кваліфікаційної роботи магістра з актуальних завдань спеціальності;
3.2 Нормативні матеріали (стандарти, Гости);
3.3 Технічні вимоги;
3.4 Науково-технічна література з питань, пов‘язаних з темою работи.

____

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):

4.1 Порівняльний аналіз результатів, отриманих іншими авторами
4.2 Методика дослідження;
4.3 Результати дослідження
4.4 Висновки.

5. Перелік графічного матеріалу.

6. Дата видачі завдання____

РЕФЕРАТ

Текстова частина бакалаврської роботи : 77 с., 19 рис., 9 джерел.

ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ, ІГРОВИЙ ПРОЕКТ, КОМП'ЮТЕРНА ГРАФІКА, UNITY, ЯЗЫК C#, ADOBE ILLUSTRATOR, ІГРИ TOWER DEFENSE.

Мета роботи – створити ігровий проект на основі Unity. Для досягнення поставленої мети були визначені наступні завдання:

- провести аналіз предметної області;
- освоїти інструментальні засоби;
- створити концептуальну модель;
- здійснити відмальовку дизайну;
- здійснити програмну реалізацію ігрового проекту.

У результаті виконання випускної кваліфікаційної роботи була створено гру у жанрі Tower Defense на основі Unity за допомогою мови програмування C#.

ЗМІСТ

Стор.

1. Загальні відомості………………………………………………………...…..7
1.1. Опис гри жанру Tower Defense……………………………………….7
1.2. Особливості балансу і механіки мобільного додатку жанру Tower Defense………………………………………………………………………..8
1.3. Огляд різновидів ігор у жанрі Tower Defense…………………….....15
2.2 Інструментальні засоби………………………………………………24
2.2.1 Unity………………………………………………………………24
2.2.2 MonoDevelop……………………………………………..……27
2.2.3 Мова програмування C#..28
2.2.3.1 Відомості про мову…………………………………….…29
2.2.3.2 Структура C# скрипта в Unity……………………...……30
2.2.4 Adobe Illustrator………………………………………………..32
3 Програмна реалізація…………………………………………..…34
3.1 Концептуальна модель……………………………………………..34
3.2 Відмалювання дизайну…………………………………………….…35
3.3. Створення 2D макета у середовищі Unity……………………………..41
3.3.1 Основи Unity………………………………………………………...41
3.3.2 Створення користувацького інтерфейсу………………..…………46
3.4 Створення сінглплеєра………………………………………….…50
3.5. Скриптинг…………………………………………………………50
ВИСНОВКИ……………………………………………………………74
ПЕРЕЛІК ПОСИЛАНЬ...76

ВСТУП

XXI століття – століття стрімкого розвитку інформаційних технологій. Вони дозволяють швидко розв’язувати поставлені завдання, економити час і досягати максимально комфортного рівня життя. Сучасні смартфони, планшети та інші гаджети люди носять з собою всюди, тому мобільні технології стали невід’ємною частиною нашого життя. Разом зі збільшенням ринку персональної електроніки так само стрімко швидко зростає ринок розваг – тому здійснення проектів по створенню ігор в цей час дуже актуально і можна сміливо стверджувати, що затребуваність таких не зменшиться у найближчі десятиліття.

Глобальний ігровий ринок у 2020 році виріс майже на 20 %. Його вартість становить 174,9 мільярди доларів. При цьому 49% припадає на мобільні ігри, на другому місці – PC (21,4%). Всі консольні проекти посідають 29% ринку ігрової індустрії [6].

Комп’ютерні ігри користуються незмінною увагою. Вони приносять чимало користі. Багато навчальних програм в своїй основі містять ігрову складову. Ігри вчать людини швидко реагувати, приймати рішення, при цьому відчуваючи свою відповідальність за це. Вони сприяють розвиткові образного мислення, критичного, стратегічного мислення. Розвивають дрібну моторику, вчать планувати свої дії. Можуть допомогти з вибором професії. Крім розвивальної та навчальної функції, ігри дозволяють людині провести дозвілля, відволіктися від справ і просто відпочити. Теоретична значущість даної роботи полягає у тому, що набутий досвід на всіх етапах розробки мобільного застосування дозволить у майбутньому набагато впевненіше ставити перед собою цілі з проектування більш складних ігор і досягати їх. До того ж дана робота показує, що процес створення гри захоплюючий і пізнавальний, дозволяє втілити в життя всі свої ідеї та задумки. І в кінцевому підсумку скористатися всіма корисними властивостями кінцевого продукту – гри. Пристосованість гри до різних мобільних пристроїв, і, відповідно до

найпоширеніших платформ є відмінною перспективою для її успішного просування, залучення великої кількості аудиторії при реалізації кінцевому користувачеві. На сьогоднішній день мобільні додатки у декілька разів перевершують за популярністю додатки для комп'ютера. Впродовж останніх 10 років набув чималої популярності жанр мобільних стратегічних ігор Tower Defense («Захисні вежі»). Оригінальність цього жанру полягає у тому, що вежі прив'язані до точки, де була проведена споруда, а супротивники рухаються по певній траєкторії, яку гравець може контролювати.

Слід наголосити, що для успішного проекту, який зможе довго утримувати гравців, а також приносити прибуток упродовж декількох років необхідно створити захопливий світ гравців, наповнений безліччю активностей, а також здійснювати впровадження нового контенту та довготривалу підтримку продукту. Для всього цього потрібне використання найбільш придатних засобів розробки.

Таким чином, було визначено мету бакалаврської роботи, а саме: створити ігровий проект у жанрі Tower Defense на Unity за допомогою мови програмування C#.

Для досягнення цієї мети потрібно:

- провести аналіз предметної області;
- освоїти інструментальні засоби;
- створити концептуальну модель;
- здійснити отрисовку дизайну;
- здійснити програмну реалізацію ігрового проекту.

1. Загальні відомості

1.1. Опис гри жанру Tower Defense

Ігри жанру Tower Defense (у перекл. з англ. – «Вежовий захист») належать до стратегічних ігрових додатків. Завдання гравця в іграх цього жанру перемогт наступаючих врагів, яких у деяких ігрових додатках цього типу називають «крипами» (у перекл. з англ.мови creep – «Повзуча тварюка»), про тих, хто перетнувши карту, шляхом будівництва веж,атакуючих, коли поблизу проходить супротивник. Суротивники та вежі розрізняються зазвичай за їхньою ціною та характеристиками. Перімігши ворога, гравець заробляє очки, або гроші, котрі можна застосовувати для модернізації, або покупки веж.

Підбір веж різного типу, їх розташування є невід'ємною стратегією гри. «Повзучі тварюки» зазвичай пробігають ходами, схожими на лабіринти, що дає гравцеві можливість стратегічно розмістити вежі. Існують також досить відомі версії гри, що мають назву лінійні TD, у яких використовуються прямі шляхи замість лабіринтів. При цьому, у деяких версіях ігрового додатку жанру Tower Defense гравець має можливість самостійно вибудовувати з блоків і веж лабіринти.

Тобто, сутність гри полягає у тому, що потрібно знищити тих супротивників, котрі здійснюють і не допустити проходження супротивника до краю ігрового поля. Противника можна знищувати тільки будуючи вежі, які гравець має можливість розташовувати вздовж ігрового поля, яким рухаються супротивники.

Гра жанру Tower Defense відбувається на карті, або полі, – як правило, прямокутної форми, проте поле / карта може мати й іншу форму.

Рух ігрових одиниць здійснюється картою / полем, які мають назву юніти. При цьому, юніти можуть інтерпретуватися по-різному. Зокрема, це може бути бойова техніка, монстри, військові. Серед усіх наявних юнітів виділяється «бос», тобто максимально сильна одиниця. Завдання гравця полягає в

наступному: необхідно відповідно до певних характеристик впливу на юнітів підібрати вежі й розмістити їх так, щоб ні один із юнітів свій шлях не пройшов до кінця. Деякі з ігор цього жанру можуть мати головні замки, або форти, котрі слід захищати від юнітів.

Одним з найскладніших аспектів у розробці ігор цього жанру є те, що кожен з ігрових додатків повинен мати так званий баланс гри. В сучасному середовищі розробників ігор Tower Defense загальноприйнятою є думка про те, що баланс гри можна визначити експериментальним шляхом, у процесі бета-тестування гри певної версії, способом підбору певних параметрів керування, – асаме характеристик веж і юнітів.

Проте баланс гри остаточно відточується впродовж певного часу вже після виходу самого ігрового додатку. Так, після того, як була випущена нова версія балансу розробниками пропонується встановлення патчу, за допомогою якого можливим стає виправлення всіх наявних недоліків у грі, а також її збалансування.

1.2. Особливості балансу і механіки мобільного додатку жанру Tower Defense

Гра жанру Tower Defense будується від процесу зведення захисних веж до грамотного управління грошовими ресурсами – на балансі механіки та економіки. Більшість розробників-початківців припускаються помилок на ранніх етапах створення гри. Найголовніші помилки – це недотримання балансу ігрової механіки, затягнутість ігрового процесу, неправильні розрахунки атаки веж і швидкості руху супротивників, неправильні підрахунки інтервалів руху «хвиль». Також розробники часто стикаються з проблемою «виродження» – це ситуації, коли гравець потрапляє у глухий кут і закінчує гру через її непрохідність, яка виникає у зв'язку логічними, або технічними помилками.

Важлива складова ігор у цьому жанрі – це баланс та ігрова механіка, адже від цього залежить весь ігровий gameplay.

Використовувана термінологія [1]:

1. Вежа – тактичний ігровий елемент, установлюваний гравцем. Вона атакує ворожих юнітів. Від розстановки веж залежить проходження рівня, а також гри у цілому.

2. База – найголовніший об'єкт, що захищається у грі, при руйнуванні його відбувається завершення рівня (ураження).

3. Клітка – один осередок поля.

4. Ворожий юніт – супротивник, що рухається з певної точки до бази гравця. При досягненні бази гравця, він починає її атакувати, завдавши значних втрат, руйнує її. Не в силах атакувати вежі.

5. Хвиля – певна кількість супротивників, що рухаються з певним інтервалом часу. Після знищення ворожих хвиль гравцеві дається невелика кількість часу на стратегічну оцінку ситуації та прийняття рішень щодо поліпшення або спорудження веж.

6. HP (ХП) – очки здоров'я супротивників, що витрачаються при атаці вежами.

Оригінальність жанру. Оригінальність цього жанру полягає у тому, що вежі прив'язані до точки, де була проведена споруда, а супротивники рухаються по певній траєкторії, яку гравець може контролювати. Завдяки цим факторам, процес створення гри спрощується. Граничні умови гри.

1. Гру можна успішно завершити, якщо виконати хоча б одну послідовність дій.

2. Гравець повинен вчиняти певні дії для успішного завершення рівня. Складність гри залежить від стратегічних дій гравця – це прийняття правильних рішень у певні моменти. Бездіяльність у певних ігрових ситуаціях також може виявитися правильним рішенням.

Тактика гравця має залежати від карти рівня, а також типу супротивників на ньому, виходячи з яких гравець може побудувати стратегію проходження рівня.

У роботі розглянута механіка гри на поле розміром 18 * 9 клітин без скролінгу. Завдяки такому рішенню gameplay гри з «побудуй якомога більше веж на карті» переходить в грамотне будівництво N-го кількості веж.

Базова вежа і базовий супротивник. Виходячи з розмірів обраної ігрової карти (18 * 9), на початковій стадії гри довжина шляху супротивника при малозаповненому полі від точки його появи до просування до бази має бути в середньому 18-20 клітин. Гравець у цей час встигає відреагувати, а також виконати тактичні дії в грі [1].

Приймемо швидкість супротивника за 0,7 клітин у секунду. При такій швидкості юніт динамічно виглядає, і гравець встигає реагувати на gameplay гри. Швидкість зі схожим інтервалом часу використовується у грі «Plant's vs Zombie».

Для зручності розрахунків візьмемо втрату вежі на першому рівні за 5 одиниць і швидкість атаки – 2 постріли в секунду. Для того, щоб визначити HP супротивника, необхідно виконати наступну умову: 4 клітини / 0,7 клітини = 5,7 секунди. За цей проміжок часу вежа зробить 10 пострілів. Виходить, що HP юніта не має перевищувати 5X5X2 = 50 HP.

Для визначення швидкості і області (радіусу) атаки стандартної вежі необхідно враховувати, що стандартна вежа має знищувати супротивника, який самотньо рухається в порожньому полі. Щоб супротивник при русі до бази при появі з будь-якої точки ігрової карти потрапив під обстріл вежі, необхідно вказати радіус обстрілу вежі, що дорівнює чотирьом клітинам. При цьому радіусі вежа, встановлена по центру карти, буде діставати до супротивника в порожньому полі, незалежно по якій траєкторії він рухається.

Варіанти «виродження» ситуації. При розробці гри необхідно враховувати ситуації, які призводять гру у глухий кут, або роблять її надміру простою. Перелічимо найбільш важливі з них.

1. Не збалансований заробіток грошових коштів. Можлива ситуація, коли при досягненні певного числа веж, гравець знищує будь-яке число супротивників і з однієї хвилі супротивників заробляє необхідну кількість грошей для проходження всієї гри.

2. Немає вільних місць для побудови вежі. Ця ситуація можлива в разі, якщо вартість веж дуже низька, або ігрове поле занадто мале.

3. Чергова дія гравця стає передбачуваною за N кроків до її здійснення. Це відбувається в іграх, де є вільна забудова у відкритому полі, і всі вороги пересуваються з 1 точки.

Балансу економіки і «хвиль» у грі можна досягти, дотримуючись наступних рекомендацій:

1. Гравцеві необхідно будувати вежі через кожні 1-2 хвилі. Якщо гравець не потребує будівництва веж упродовж 3-4 хвиль – баланс поганий, динаміка гри йде на спад, гравець швидко втрачає інтерес.

2. Ігрових грошових коштів, отриманих від знищення хвилі, має вистачати на будівництво нових веж. Можна підвищити запас коштів для того, щоб відбити наступну хвилю.

3. Рівень складності гри має рівномірно зростати від початку до кінця рівня. Кількість ресурсів у гравця не має бути у надлишку.

4. До кінця рівня виникає ситуація, коли будувати нові або поліпшувати старі вежі вже нікуди, це максимальна межа рівня гри. У такому випадку рівень має бути закінчений до цього моменту. Рівень має завершитися (бути пройденим), коли забудовано не більше 70% можливих місць для забудови веж та ігрових елементів.

5. Перед гравцями стоїть вибір – побудувати нову вежу або поліпшити стару, де розмістити нову вежу. Складність гри має зростати до кінця рівня.

Вираховування кількості супротивників у хвилі. Візьмемо ідеальну ситуацію, при якій хвиля супротивників рухається вздовж ряду веж по прямій. Розрахуємо скільки супротивників може бути знищено при певній швидкості руху і довжині доріжки.

Розглянемо базового супротивника і базову вежу. При фіксованому радіусі атаки базової вежі, одного базового противника можуть атакувати одночасно 4 вежі. Обгрунтування прописано в [9].

Виходячи з обраних розмірів карти і швидкості руху противників, візьмемо такі параметри веж. Швидкість атаки вежі – 2 постріли в секунду, шкоди – 5 HP за постріл, HP противника – 50 одиниць. Виходить, що базовий супротивник витримує 10 влучень.

Одночасно хвилю супротивників може атакувати N кількість веж. Від кількості супротивників залежить кількість веж. Якщо довжина хвилі супротивників збільшується на одну клітку, то кількість одночасно атакуючих веж збільшується на одну. Довжина хвилі супротивників розраховується за такою формулою: N X dt X v, де N - число, dt - інтервал часу між появами супротивників, V - швидкість супротивника.

Приймемо інтервал часу перед появою наступного супротивника за 0,5 секунд. Це найменший інтервал часу. Якщо збільшити інтервал часу, то складно буде розрізнити противників між собою, вони будуть зливатися в одну фігуру), V = 0,7 / t.

Зробимо розрахунок проміжку часу, впродовж якого вежі будуть атакувати супротивників. Цей розрахунок можна виконати за умови, що відома кількість веж, які атакують одночасно. Розділимо довжину шляху супротивників під атакою на швидкість їхнього руху. Якщо виконати добуток цих величин, то отримаємо кількість шкоди, яку завдадуть вежі цій хвилі.

Отримуємо нерівність:

$$(4 + N \times dt \times v) \times K \times \frac{L}{v} \geq N \times h$$

, де: h - стійкість супротивника (кількість ударів, які він може витримати); L - довжина шляху супротивників під атакою; К - кількість пострілів вежі на секунду; v - швидкість супротивників; N - кількість супротивників.

$$K = 2; v = 0,7; dt = 0,5; h = 10.$$

У підсумку виходить проміжний результат:

$$L \times (4 + 2 \times N * dt) > 10 \times N;$$
$$(12 + N) \times L \geq 10 \times N.$$

Таким чином, якщо довжина шляху, який прострілюється вежами більше 10 клітин, вежі знищать будь-яку кількість супротивників. у кінцевому підсумку отримали формулу, яка дозволяє обчислити, при якій довжині шляху і яку кількість супротивників треба запускати. Таким чином, щоб нерівність виконувалася, L і N визначаються довільним чином. Необхідно, щоб кількість супротивників не порушува баланс економіки. Гравцеві з кожної хвилі має вистачати на будівництво хоча б однієї вежі, або на її поліпшення. У той же час не можна допускати, щоб гравець після першої хвилі супротивників забудував всю карту вежами.

Розрахунок балансу перших хвиль супротивників. Розглянемо наступну ігрову ситуацію: на початку гри гравцеві вистачає коштів на будівництво двох веж. Довжина шляху під обстрілом супротивників одночасно двома вежами буде 4 клітки (радіус атаки вежі 4 клітини), а загальна довжина шляху під обстрілом буде 4 + 4 = 8 клітин. Обчислимо кількість супротивників у першій хвилі за такою формулою:

$$(12 + N) \times 4 \geq 10 \times N, \text{ при } dt = 0,5c$$
$$48 > 4N, \qquad N < 10$$

Щоб гравцеві було достатньо грошей для розвитку і проходження другої хвилі, він має отримувати з кожного юніта в першій хвилі у середньому по 5 монет.

Після підрахунків кількості грошових коштів гравця після першої хвилі, можна розрахувати потужність ігрових веж і супротивників у другій хвилі.

Поведінка гравця, при межах забудови. Під час ігрової паузи (інтервал часу між хвилями супротивників) гравцеві дана можливість покращувати

існуючі або будувати нові вежі. При проходженні 10 хвиль гравець відповідно справить 10 будівель або поліпшень.

Гравцеві теоретично виділено 162 місця для забудови баштами (18 X 9 = 162).

Точка виходу супротивника знаходиться навпроти точки бази гравця. Всі противники виходять з однієї точки.

При реалізації «доріжки» (гравець вибудовує траєкторію руху противників шляхом вибудовування веж, в зв'язку з чим противники обходять перешкоду перед ними, щоб продовжувати рухатися до бази) противники перебувають тривалий час під обстрілом. В даному випадку довжина «доріжки» буде близько 50 клітин. Щоб змусити супротивників рухатися по «доріжці», гравцеві необхідно побудувати мінімум 32 вежі. Якщо довжина шляху противника буде 45 клітин, то його шлях до бази складе 45 - 0,7 = 64 секунди. Шлях всієї хвилі займе близько 1 хвилини і 15 секунд. В середньому хвиля супротивників буде займати близько 45 секунд, значить час проходження одного рівня при 10 хвилях складе близько 5 хвилин на високому рівні складності. Для різноманітного геймплея буде досить і 5-7 хвиль.

Розрахуємо, за яких параметрах противники зможуть дійти до бази гравця.

$$(12 + N) \times 45 \geq h \times N.$$

Кількість HP супротивників, щоб вони дійшли до бази гравця, має бути не менше 45 x 5 = 225, а це в 4,5 рази більше, ніж HP стандартного юніта. Він має витримати 45 снарядів стандартної вежі.

$$(12 + N) \times 45 \geq h \times N$$
$$540 + 45N \geq h \times N$$

При h=55 отримуємо:

$$540 \geq 10 \times N, N \leq 54$$

При 54 супротивниках у 1 хвилі вежі не зможуть їх всіх знищити перш, ніж вони зруйнують базу гравця.

Для зручності розрахунку економічної складової гри візьмемо вартість базової вежі 80 монет. Можна обчислити кількість грошей, необхідних для будівництва «доріжки» (необхідно 32 вежі).

80 x 32 = 2560 монет

Таким чином, удалося визначити граничні умови гри. Розрахований баланс ігрових веж, розглянуті баланс хвиль і економіки гри. Визначено варіанти «виродження» (безвихідній) ситуації для гравця і знайдено їх вирішення. Виконано розрахунок кількості ворожих юнітів в 1 «хвилі». Виконано розрахунок характеристик ворожих юнітів.

1.3. Огляд різновидів ігор у жанрі Tower Defense

Один з найбільш ранніх представників TD – аркадна гра Rampart, перенесена згодом на безліч платформ. Ця гра стала поштовхом для створення низки карт для гри «StarCraft», що мають назву Turret Defense, які, своєю чергою, надихнули на створення карт Hero Defense (Warcraft III) і Sunken. Такі модифікації стали популярними серед гравців у Warcraft III і Age of Empires II, згодом зміцнившись у вигляді окремого жанру. Першим представником окремої гри у жанрі Tower Defense стала комп'ютерна гра «Master of Defense», яка вийшла 7 листопада 2005 року і набула великої популярності і в 2006 здобула нагороду «Стратегія року» від GameTunnel. Розробник гри «WarCraft» компанія Blizzard створила додатковий рівень «Tower Defense» в «Warcraft III: The Frozen Throne». Популярним зразком жанру є браузерна флеш-гра GemCraft.

Варто відзначити, що цей жанр існує і на інших платформах, таких як мобільні телефони (Town Defense, Tower Defense, Plants vs Zombies), PSP (VectorTD, Castle Rustle, Field Runner) тощо.

Існують певні несуттєві відмінності між різними іграми цього типу. До прикладу, в більшості версій, коли здійснюється модернізація вежі, її потужність, рівень і радіус дії зростає одночасно. Проте у такій версії гри, що

має назву Onslaught Defence, кожен і параметрів можна окремо поліпшувати. У деяких із існуючих ігор жанру Tower Defense супротивник має змогу чинити оборону. Маршрут просування ворога в окремих з версій гри не обмежується будь-якими межами (стінками). Хвилі настання розпочинаються за командою гравця (що надає можливість спокійно здійснити підготувку), або ж із певним часовим проміжком. Присутніми можуть бути елементи економічної стратегії (до прикладу, можна звести банк, який грошові кошти буде збільшувати в геометричній прогресії).

Супротивники у різних версіях ігри цього жанру можуть відрізнятися за низкою властивостей – до прикладу, деякі з них мають можливість здійснювати польоти, і для їхньої поразки мають бути побудовані спеціальні «протиповітряні» вежі. Зокрема, в грі GemCraft вежі (й пастки) розділені від їхніх активних здібностей за атакою (їх забезпечую дорогоцінним камінням). При цьому, дорогоцінні камені можна виймати з веж і об'єднувати, що сприяє посиленню. З'явилися також характеристики гравця, котрі можна й необхідно розвивати проходячи від карти до карти. З'явилися також й інші ігри, котрі застосовують цю знахідку.

Існує також такий тип ігор Tower Invasion, де гравець має можливість управляти «Кріпом», що має оминути вежі супротивника.

Нині вже розроблено чимало різновидів ігор жанру Tower Defense. До прикладу, в інді-грі Tower Chip Defence вежами є впаяні у друковану плату електронні елементи.

Також Tower Defense може бути «змішаним» з певним іншим жанром. До прикладу, Sanctum є міксом класичного TD, а також шутера від першої особи, де гравець ніби допомагає відбивати захисним спорудам власною зброєю хвилі супротивників. А в такій серії, як Toy Solders, можна допомогти своїм вежам не лише героєм, а й ручним управлінням башт та іграшкової технікою.

Окрім того, існують ігри жанру Tower Defense, у яких замість відбивання облоги гравець самостійно намагається прорватися крізь захист.

Серед найкращих ігор жанру Tower Defense одними з найбільш популярних є: Plants vs. Zombie, Orcs Must Die, Plants vs. Zombies 2: Garden Warfare, Orcs Must Die! 2, Anomaly: Warzone Earth, Plants vs. Zombies 2: It's About Time, Deathtrap, Rock of Ages, Sanctum 2 та ін.

Plants vs. Zombie (для платформ PC / Xbox 360 / Android / iOS) – це нетиповий представник жанру "tower defence". Суть гри зрозуміла вже з назви і полягає вона в порятунок заміського будинку від навали зомбі. Численні мерці човгає, пливуть і біжать по галявині, яку садівник-любитель засаджує бойовими рослинам – картопляними мінами, горохострелами, і іншими настільки ж божевільними фруктами і овочами. Мета гравця – відбити всі хвилі наступаючих живих мерців, який з кожним разом стають все сильніше і настирного, знаходячи нові можливості та екіпіровку.

Plants vs. Zombies 2: Garden Warfare (PC / PlayStation 4 / Xbox One / PlayStation 3 / Xbox 360) – розрахований на багато користувачів мультиплеєрний шутер у всесвіті Plants vs. Zombies, в якому гравці можуть зіграти як за рослини, так і за зомбі.

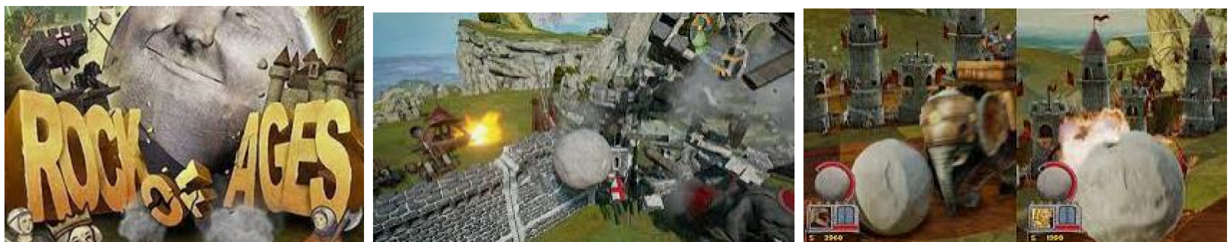
Orcs Must Die (PC / PlayStation 3 / Xbox 360) – це фантастична екшен-стратегія, мета якої – змусити гравця захищати фортецю.

Orcs Must Die! 2 (PC) – це друга частина оригінальної стратегії, де гравцеві належить прорубуватися через незліченні полчища орків на пару з другом.

Anomaly: Warzone Earth (PC / Xbox 360 / iOS / Mac) – це стратегія в реальному часі з елементами аркадной гри і Tower-defence-вкраплюваннями, в якій гравцеві доведеться боротися з інопланетянами на поверхні Землі.

Deathtrap (PC / PlayStation 4 / Xbox One / Mac) – це tower defense на чотирьох гравців, де гравцям належить використовувати різні хитрощі, щоб домогтися перемоги.

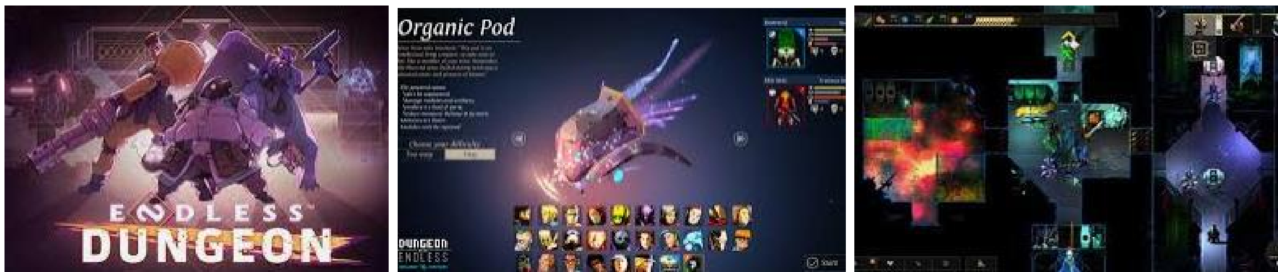
Rock of Ages (PC / PlayStation 3 / Xbox 360) – поєднує в собі жанри аркади і покрокової стратегії. Ми захищаємо свій замок від нападок ворогів.



CastleStorm (PC / PlayStation 3 / Xbox 360 / PlayStation Vita / Nintendo 3DS) – ця гра виконана у змішуванні жанрів стратегії в реальному часі і tower defence у середньовічному сетингу. За сюжетом гравець зводить власний замок

і обороняє його від наступаючих супротивників. Доступний редактор рівнів, який дозволяє створювати власні вежі, які можна використовувати в грі.

Dungeon of the Endless (PC / PlayStation 4 / Xbox One / Nintendo Switch / Android / iOS) – це пригодницький «рогалик» у всесвіті Endless Space і Endless Legends, дія якого розгортається після краху космічного корабля на планету Ауриго. Гравцям належить провести загін з чотирьох героїв до самого серця зловісного лабіринту, заповненого монстрами.



Toys 'n' Traps (PC / PlayStation 4 / Xbox One / Nintendo Switch) – це екшен-стратегія у жанрі tower defense, у якій гравці мають захищати іграшкове місто від атаки єнотів-розкрадачів. За допомогою різноманітних пасток і унікальних умінь кожної конкретної іграшки гравці мають відбивати атаки супротивників, не допускаючи їх до своєї бази. Гравців чекає чотири сценарії на різних локаціях, можливість покращувати і розблокувати нові вежі, зброя та пастки, а також багато іншого.

Martians Vs Robots (PC) – це аналог Plants vs Zombies, тільки в цій грі не зомбі атакують ділянку садівника-любителя, а марсіани намагаються повернути контроль над своєю планетою, атакуючи роботів і турелі гравців.

Decision: Red Daze (PC) – це рольовий екшен про виживання, змішаний з tower defense-грою. Сюжет розповідає про те, як континент охопила жахлива напасть під назвою «червона імла», що перетворює живих істот в мутантів. Ті, що вижили намагаються об'єднатися в команду, щоб дати відсіч чудовиськам і спробувати розібратися з джерелом проблеми. Гравці мають набрати загін різноманітних героїв, створювати і зміцнювати існуючі поселення, а також досліджувати світ, повний небезпечних чудовиськ, пасток і цінних предметів.

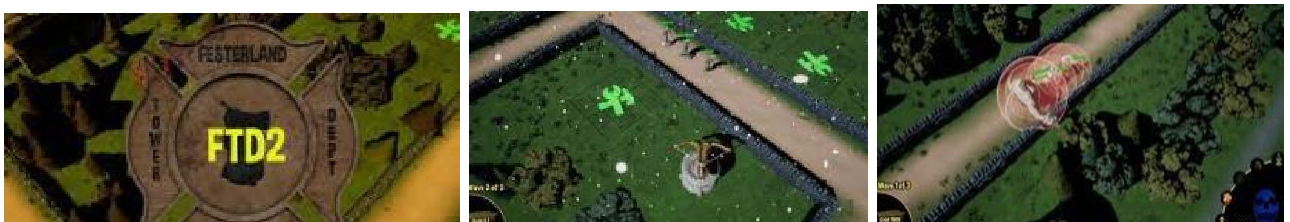
Boom Boom Tower (PC) – це казуальна гра, в якій гравці повинні захистити свою башту від різних ворогів, оснащених різною зброєю.

Braverz (PC) – це кооперативна MOBA / tower defense-гра, в якій гравці мають захищати свою базу від полчищ NPC, які прагнуть її знищити. Гравцям доступні декілька унікальних героїв, кожен зі своїм стилем гри і набором навичок, можливість зводити захисні вежі, а також вид від третьої особи, який дозволяє брати особисту участь у всіх боях.

Arknights (Android / iOS) – це мобільна гра в жанрі tower defense з елементами «гачі», в якій гравці беруть під свій контроль організацію під

назвою Rhodes Island – фармацевтичну корпорацію, яка намагається відшукати порятунку від смертельної хвороби на ім'я оріпатія, яка захлеснула світ Терри. У ролі Доктора, номінального глави «Родосу», гравці кидають виклик терористичній організації Reunion, зайнятої роздуванням пожежі війни між найбільшими державами континенту. Геймплей розділений на декілька сегментів. У бою гравці повинні виставляти на поле бою «вежі» – оперативників, кожен з яких є унікальним персонажем зі своїм класом, здібностями й особливостями. Сюжет подається у вигляді візуальної новели, а в перервах між боями і вивченням сюжету гравці можуть зайнятися відбудовою і доведенням до розуму власної пересувної бази, на якій між завданнями відпочивають і тренуються оперативники. Персонажів в Arknights можна отримати за допомогою "гачі" – тобто внутрішньо рулетки, яку можна крутити як за внутрішньоігрову, так і за реальну валюту.

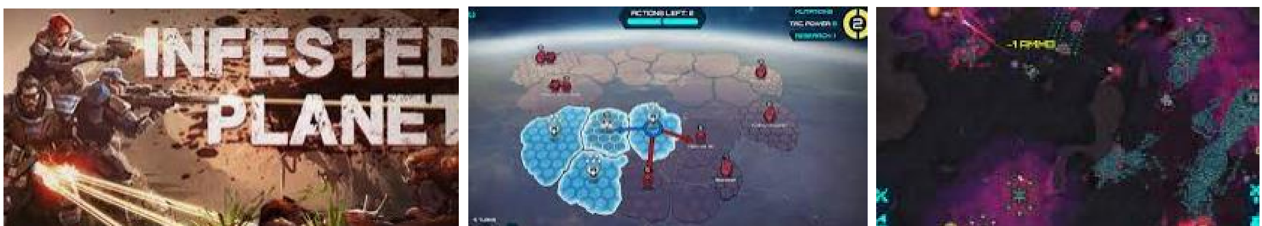
FTD2 (PC) – це фентезійна гра в жанрі «захист вежі», в якій гравців чекають вісім типів веж з різними поліпшеннями, вісімнадцять видів супротивників, а також чотирнадцять різних ігрових режимів. Мета гри – врятувати місто від навали монстрів, а також дізнатися, хто їх відправляє і з якою метою цього невідомого потрібні були душі померлих містян.



Goblin Quest: Escape! (PC) – це суміш гри про зачистку підземель і «реверсного» tower defense, де гравці мають не зводити вежі для оборони, а навпаки – нападати на них і руйнувати.

SolSeraph (PC / PlayStation 4 / Xbox One / Nintendo Switch) – це ретро екшен-платформер з захистом веж, в якій гравців чекає необхідність врятувати світ від нападу жахливих монстрів. Гравцям потрібно будувати захисні споруди, атакувати лігва монстрів силами своїх персонажів, а також застосовувати різні заклинання, щоб знищувати їх максимально ефективно.

Infested Planet (PC / Mac) – це tower defense-стратеія і продовження Attack of the Paper Zombies. У порівнянні з оригіналом у грі стало більше класів (шість замість трьох), більше будівель, а геймплей був істотно перероблений.



Project RTD: Random Tower Defense VR (PC) – це VR-гра в жанрі «баштового захисту», в якій гравці мають відображати хвилі супротивників за допомогою різноманітних веж. Особливість гри полягає в тому, що карта в Project RTD повністю тривимірна, і вороги біжать не по одній площині – вони біжать від низу до верху, до самого серця фортеці. Щоб перемогти, гравці мають грамотно розташовувати свої вежі, яких у грі понад тридцять і вивчати всі карти для створення оптимальної стратегії.

Winged Sakura: Mindy's Arc 2 (PC / Mac / Linux) – це суміш tower defense і візуальної новели, що є сиквелом Winged Sakura: Mindy's Arc. Втім, сюжет першої частини для розуміння сиквела знати зовсім не обов'язково.

2.2 Інструментальні засоби

2.2.1 Unity

Ігрова платформа (движок) – це базове програмне забезпечення, на якому розробляється і виконується гра, являє собою сукупність декількох підсистем, в які входять звукова, графічна та фізична. У сучасних ігрових платформах такі системи є модульними, а їх опрацювання і функціонал залежать від фонової програми.

Якщо піонерам розробки ігор доводилося розробляти свої платформи, то в наш час, незалежно від того, чи розрахована гра на одного користувача, або вона розрахована на багато користувачів, розробники в більшості випадків обирають вже готові інструменти розробки.

Визначитися з ігровим движком означає побудувати правильну стратегію розробки та підтримки програми [8].

Unity є мультиплатформовим інструментом для розроблення двох- і тривимірних додатків та ігор, що працює під операційними системами Windows і OS X. Ігри, створені на цьому інструменті, можна перенести на: Windows, OS X, Android, Apple iOS, Linux, а також на ігрові приставки Wii, PlayStation 3 і XBox 360. Так само можна створювати ігри, що працюють у браузері, що, своєю чергою, вимагає встановлення спеціального модуля Unity Web Player. Також додатки створені, за допомогою Unity3D підтримують обидві специфікації 3D графіки DirectX і OpcnGL.

Unity – багатоплатформовий ігровий движок. Це повноцінна сфера розробки комп'ютерних ігор, що включає в себе різні програмні засоби для створення ПЗ – компілятор, текстовий редактор, відладчик і т.ін. Цим движком користуються як великі розробники, так і незалежні студії, для багатьох з яких саме він став дверима в світ розробки ігор.

Вагомим аргументів на користь вибору платформи Unity є її безкоштовна модель поширення для проектів з оборотом не більше 100 тис. доларів США за останні 12 місяців. Завдяки компонентно-орієнтованому підходу Unity дає

можливість створювати ігри розробникам з мінімальним досвідом у цій сфері. Движок має зручний інтерфейс Drag & Drop з можливістю розставляти об'єкти і у реальному часі тестувати результат.

Також Unity володіє великим співтовариством розробників, величезною бібліотекою Ассет і плагінів, що прискорює процес розроблення. За допомогою движка можна розробляти гри на всі актуальні ігрові платформа і перенести їх без особливої складності. Цей факт означає доступність GaaS проекту на більшій кількості платформ.

Підтримка DirectX і OpenGL означає не тільки гнучкість налаштування графіки додатки, але і доступ до новітніх технологій, таким як трасування променів в реальному часі.

За допомогою движка можна налаштовувати фізику об'єктів, створювати складні анімації.

На рис. 2.1 представлена графічна оболонка Unity3D.

Рисунок 2.1 – Інтерфейс Unity

Project Window (Оглядач проекту)

Ця частина інтерфейсу, щоб виконувати завдання ресурсами, які знаходяться в ігровому проекті.

У лівій частині оглядача міститься ієрархічний список, який відображає структуру папок проекту. Окремі ресурси представлені у вигляді іконок, що вказують їх тип (спрайт, скрипт і т.д.).

Hierarchy Window (Ієрархія)

Вікно містить всі об'єкти, що знаходяться в сцені. Це можуть бути будь-які UI елементи (кнопки, картинки і т.д.), 3D моделі, екземпляри об'єктів та прочісування. Кожен об'єкт (GameObject) може містити дочірні об'єкти, або входити в об'єкти більш високого рангу. Можна вибрати об'єкт в ієрархії і перемістити його в інший об'єкт, таким чином сформується батьківський зв'язок (parenting).

Дочірній об'єкт успадковує зміни свого батька, пов'язані з переміщенням, обертанням і масштабуванням.

Toolbar (Панель інструментів)

У цій частині розташовуються елементи, потрібні для трансформації, кнопки запуску і зупинки гри, меню, що випадає, шари.

Scene View (Сцена)

У цьому вікні встановлюється позиціонування елементів гри.

Game View (Гра)

У цьому вікні відображається все, що побачить користувач.

Inspector Window (Інспектор)

Інспектор відображає інформацію про конкретний обраний об'єкт.

Однак Unity має й свої мінуси. Для створення складних ігор потрібна наявність у команді розробки хорошого програміста C # для написання скриптів і компонентів. Так само движок погано себе показує падінням FPS в масштабних сценах з безліччю компонентів і одночасно присутніх гравців. Звідси випливає необхідність в ретельному процесі оптимізації ігрових ресурсів. Також проекти, створені в Unity, займають багато місця на жорсткому диску користувача, що може бути вирішальним фактором у виборі MMO проекту споживачем, який хоче грати саме в даний момент.

2.2.2 MonoDevelop

MonoDevelop – це інтегроване середовище розробки (IDE), що поставляється разом з Unity. IDE поєднує в собі функції текстового редактора з додатковими можливостями для налагодження і виконання інших завдань з управління проектами.

MonoDevelop володіє всіма основними можливостями, необхідними для сучасного інтегрованого середовища розробки:

- налаштовується підсвічування синтаксису;
- автоматичне доповнення коду;
- виділення блоків коду з можливістю їх згортання / розгортання;
- інтелектуальна робота з відступами в коді;
- можливості рефакторинга (перейменування класів і методів, автоматична реалізація інтерфейсів в похідних класах);
- зручна навігація по коду (навігація по класах, методам, властивостям);
- візуальний редактор форм для проектів на Gtk #;
- створення кількох розкладок інтерфейсу і перемикання між ними;
- безліч стандартних шаблонів проектів;
- можливість автоматичного створення бінарних пакетів і архівів після компіляції вихідного коду;
- робота з базами даних;
- створення додатків з GUI, що підтримує декілька мов;
- інтеграція з Subversion для управління вихідним кодом;
- підтримка NUnit для створення Unit-тестів;
- автоматичне створення документації;
- розширення можливостей за рахунок доповнень і зовнішніх інструментів;
- можливість інтеграції з Microsoft Visual Studio і .NET Framework (в середовищі Microsoft Windows).

На рисунку 2.2 представлено інтерфейс роботи програми.

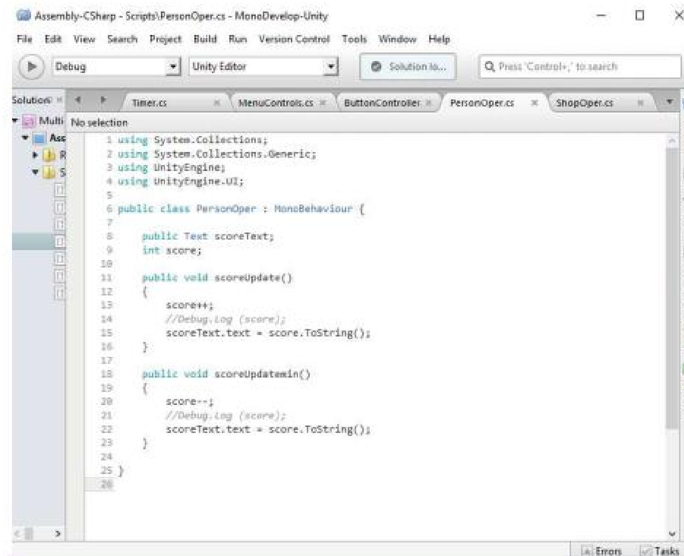


Рисунок 2.2 – Інтерфейс MonoDevelop

2.2.3 Мова програмування C#

2.2.3.1 Відомості про мову

Мова програмування – це особлива мова програмістів для розробки програмного забезпечення, або це інші набори інструкцій і алгоритмів. У нинішньому столітті вже існує дуже багато мов програмування, але по-справжньому популярні і широко затребувані тільки деякі з них. Мови програмування бувають двох видів: низького і високого рівня.

Першим комп'ютерам доводилося програмувати двійковими машинними кодами. Однак програмувати таким чином досить трудомістко і складно, для спрощення цього завдання з'явилися мови програмування низького рівня, які дозволяли задавати машинний двійкового коду у більш зрозумілому для людини вигляді, – наприклад «Асемблер».

Мова програмування високого рівня – це мова програмування, розроблена для швидкості і зручності використання програмістом. Основна риса високорівневих мов – це введення смислових конструкцій, що коротко описують такі структури даних і операції над ними, опису яких на машинному коді, або іншій низкорівневій мові програмування дуже довгі і складні для розуміння.

Портал DOU опублікував черговий рейтинг мов програмування, популярних серед українських розробників у 2021 році [7]. Рейтинг составлен на основе опроса 7211 респондентов, 92% которых живут в Украине (див. рис. 2.1.).

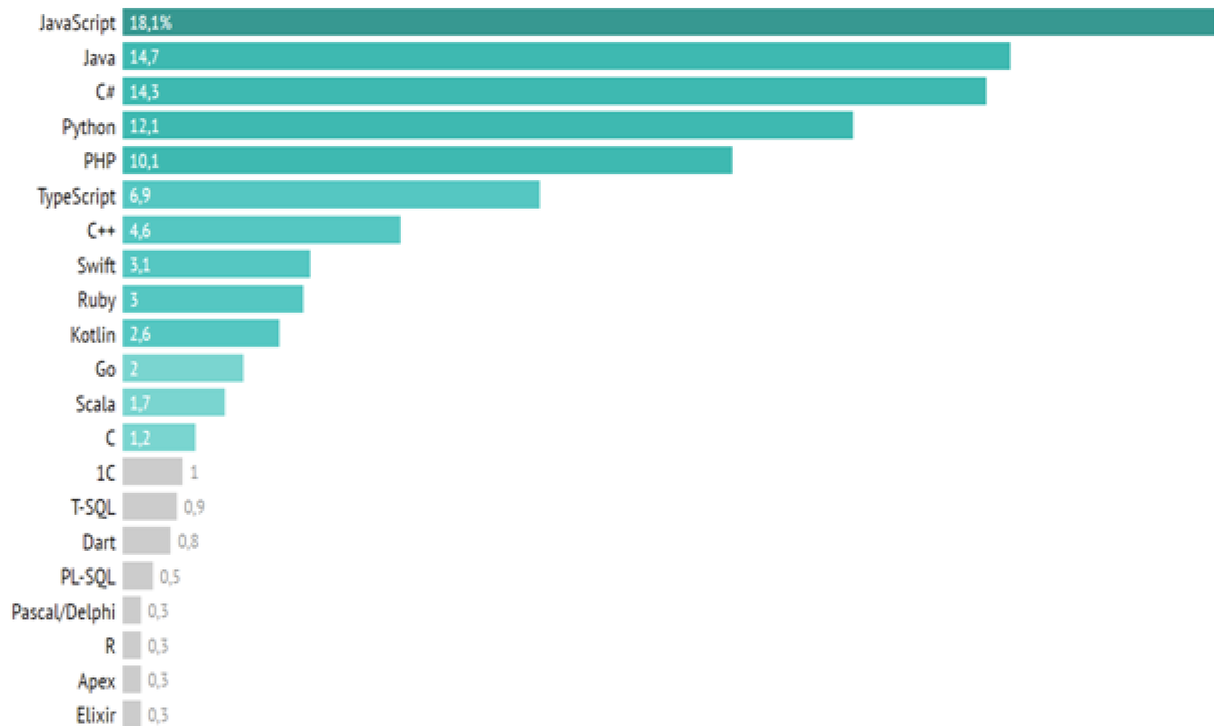


Рисунок 2.3 – Рейтинг мов програмування в Україні в 2021 році

С # (вимовляється Сі Шарп) – об'єктно-орієнтована мова програмування, що використовується для написання користувацьких програм.

Мова програмування C# розроблена в 1998-2001 роках групою інженерів під керівництвом Андерса Хсйлсбсрга в компанії Microsoft як мова розробки додатків для платформи Microsoft.NET Framework і згодом була стандартизована як ECMA-334 і ISO / IEC 23270.

С # належить до сім'ї мов с подібним синтаксисом, з них його синтаксис найбільш близький до C ++ і Java. Також вводить деякі, по-своєму унікальні і досить потужні функції, такі як, наприклад, делегати. Як і Java має велику бібліотеку класів.

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, вимкнення, коментарі у форматі XML.

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

С # не підтримує множинне успадкування, натомість забезпечує рішення інтерфейсу. Інтерфейс допомагає уникати множинного спадкоємства, зберігаючи здатність давати декілька класів для реалізації [5]. Оскільки Microsoft є досить популярним серед користувачів, то допомагає розробляти програмне забезпечення, що може виявитися цілком корисним і прибутковим.

Дана мова була обрана в якості основної, оскільки має потрібні якості для реалізації, має вбудовану підтримку узагальнень, делегатів і подій, що полегшить реалізацію.

2.2.3.2 Структура C# скрипта в Unity

Unity використовує компонентний підхід. Компонент – це клас, успадкований від MonoBehaviour. Один компонент має відповідати за одну поведінку.

Скрипти можуть бути створені в панелі проектора. Для цього достатньо натиснути кнопку Create і вибрати мову, якою буде створюватися скрипт. Прикріпити скрипт до об'єкта можна шляхом перетягування, або натискання на кнопку Add Component.

Після того, як буде створений скрипт в Unity і відкриється за замовчуванням в MonoDevelop, можна буде побачити наступне:

```
using UnityEngine;
using System.Collections;
public class MainPlayer : MonoBehaviour {
// Use this for initialization
void Start () {
}
// Update is called once per frame
void Update () {
  }
}
```

Назва створеного скрипта має повністю відповідати назві створеного автоматично класу, який успадковується від вбудованого класу MonoBehaviour.

Функція Update – це функція, що найчастіше використовується в Unity. Вона викликається один раз за кадр в кожному скрипті, що використовує її. Майже все, що вимагає змін, або регуляції на постійній основі, прописується в цій функції. Переміщення нефізичних об'єктів, прості таймери і виявлення введення зазвичай реалізуються в цій функції. Варто зазначити, що ця функція не викликається на основному таймлайнс. Якщо один кадр займає більше часу для обробки, ніж наступний, тоді час між викликами функції буде різним.

Функція Start – це функція, що викликається автоматично перед функцією Update, коли скрипт вже завантажений. Функцію можна використовувати для всього, що потрібно, тільки коли компонент скрипта увімкнений. Це дозволяє відстрочити будь-яку частину коду ініціалізації, поки вона не знадобиться.

2.2.4 Adobe Illustrator

Adobe Illustrator є професійним графічним редактором, призначений для створення та редагування ілюстрацій у векторі від компанії Adobe.

Програма володіє широким набором інструментів для малювання та макетування з можливостями управління кольором і текстом.

Зображення у векторній графіці складається з опорних точок і кривих, тому, на відміну від растрової графіки, легко піддається масштабуванню без збільшення пікселів.

При масштабуванні малюнок не втрачає якості і це дасть можливість друку на високій роздільній здатності.

Застосування векторної графіки буває різноманітним: розробка web-сторінок і створення для них логотипів та іконок, поліграфія, а також оформлення різних робіт, які не потребують фотореалістичності.

Розглянемо головні переваги Adobe Illustrator від інших векторних редакторів.

Навігація

Зменшення / збільшення ctrl + / ctrl-. Переміщення полотна за допомогою утримування клавіші «пробіл».

Робота з текстом

Налаштування тексту в Adobe Illustrator реалізовані зручно. Панель Символ налаштовує конкретно текст і його накреслення. Абзац налаштовує групу тексту. У редакторі є художній текст.

Робота з зображеннями

Усі зображення вставляються зв'язками, тому навіть складні за конструкцією файли можуть важити не багато, оскільки в них немає зображень, є тільки зв'язку, а зображення вже впроваджується в той файл, який готується до друку.

Сторінки

Сторінки можуть розміщуватися на одному і тому ж полотні, розміщуватися у просторі в різних площинах, мати різні розміри.

Ефекти

Всі ефекти, створені через програму Illustrator (тіні, прозорості та ін.) Добре виглядають при друку.

Малювання на планшеті

Працюючи в програмі через планшет можна створювати фігури будь-якої складності і легко їх редагувати. Кисть типу «Клякса» реагує на натиск і її насиченість залежить від того, як сильно натиснули на планшет.

Зовнішній вигляд програми і її робота представлені на рис. 2.4.

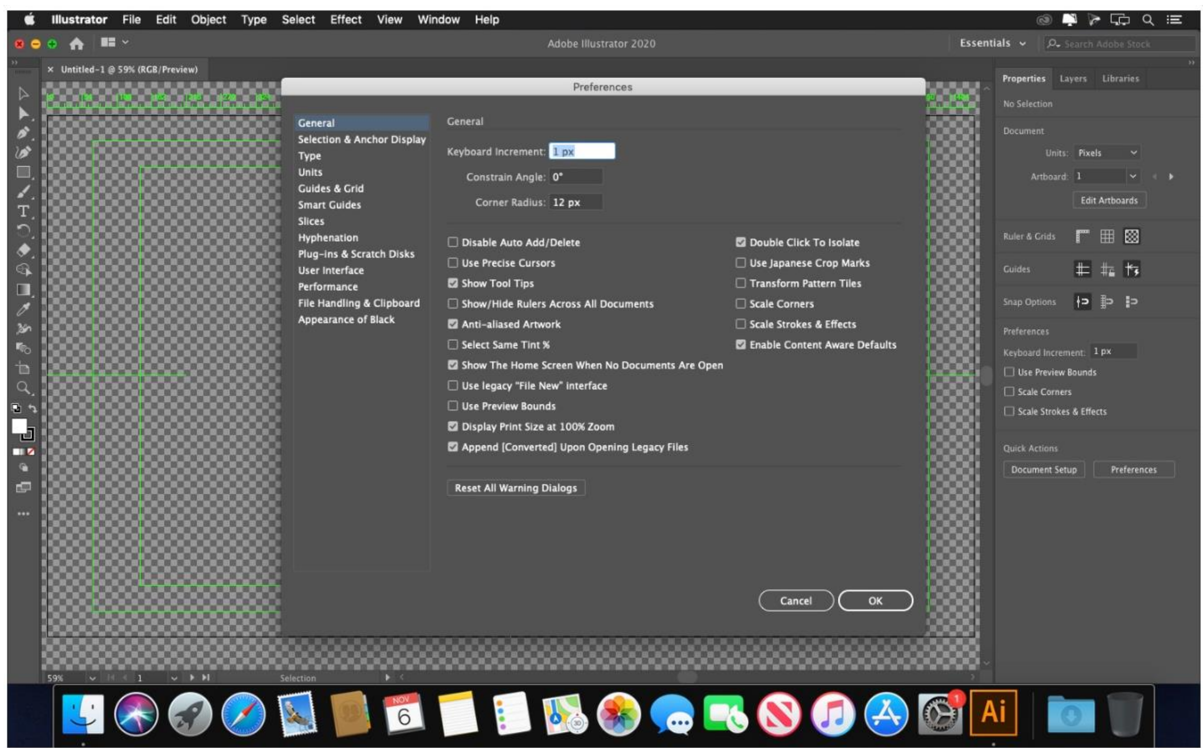


Рисунок 2.4 – Інтерфейс Adobe Illustrator 2021

У жовтні 2020 Adobe Inc. представила версію Adobe Illustrator для операційної системи iPadOS. Додаток має схожий набір функцій з Adobe Illustrator для Mac і Windows, проте призначений для користувача інтерфейс, оптимізований для роботи з жестами і Apple Pencil.

3 Програмна реалізація

3.1 Концептуальна модель

Концептуальна модель потрібна для того, щоб визначити структурні елементи предметної області і позначити зв'язки між ними.

У результаті концептуальна модель являє собою логічну структуру розглянутої області.

Концептуальна модель – перший і найважливіший крок для створення ігрового проекту. На цьому етапі геймдизайнер створює й описує свої ідеї в спеціальному документі. На виході має вийти документ, що описує гру як кінцевий продукт, а також початкове опрацювання всіх елементів гри. Далі документ використовується тестувальниками, продюсерами, дизайнерами, програмістами й інвесторами.

Оскільки даний ігровий проект створюється одним розробником-початківцем, то має сенс описати концепт лише у вигляді тез, і далі змінювати їх у процесі розв’язання практичних завдань.

При описі концептуальної моделі ігрового проекту були сформульовані наступні тези:

Жанр: Економічна стратегія у жанрі Tower Defense.

Режим: однокористувацька гра (Single-player).

Графіка: Векторна.

Простір: 2D.

Режим камери: ортогональний (Orthographic).

Кольорові рішення: теплокоричневі тони з контрастним кольоровим ритмом.

Мета: знищити найбільшу кількість супротивників, не дати супротивникам дійти до края ігрового поля.

Світ: Складається з області для одного гравця.

Гроші: Лічильник грошей має динамічну структуру:

- збільшується за кожного знищеного супротивника;

- операції над ресурсами мають прив'язку до лічильника.

Ігрові об'єкти:

- вежі, які будує гравець;
- лабіринт, по якому хвиля за хвилею наступає супротивник;

Баланс: Кількість гравців ділиться на 1 контейнер:

Розгортання: локально з мобільного пристрою.

Засіб: Unity.

Мова програмування: C#.

Таким чином, в даних тезах було визначено всі деталі гри, описані ігрові об'єкти, а також позначені засоби проектування, необхідні для реалізації проекту.

3.2 Відмалювання дизайну

На цьому етапі слід реалізувати зовнішній вигляд гри, зобразити у векторі елементи інтерфейсу і спрайт, а також визначитися з видом камери.

Потрібно налаштувати камеру так, щоб вона стала «придатною» для 2D. У Unity під камерою розуміється клас Camera. Для того, щоб її використовувати, потрібно:

1. Створити порожній об'єкт (GameObject -> Create Empty).
2. Обрати його і додати йому компонент Camera (Component -> Rendering -> Camera).

Спочатку камера перпендикулярна площині XOY і спрямована уздовж осі Z у позитивному напрямку. Надалі для простоти спрайти будуть лежати в площині XOY і в напрямку до камери, так що камеру необхідно помістити в центр координат і віддалити її по осі Z на необхідну відстань в негативний напівпростір (скажімо, в точку [0, 0, -100]).

Для 2D-графіки положення спрайтів у просторі не важливе. На багато важливіше, як спрайти перекривають один одного. Камера підтримує два режими (виду проекції): перспективний (Perspective) і ортогональний

(Orthographic). Перший використовується у всіх 3d-іграх: об'єкти, розташовані далі від камери, виглядають меншими. Це майже те, як ми бачимо наш світ.

Для розробки гри у жанрі Tower Defense на Unity за допомогою мови програмування C# викорустовувався інший режим – orthographic – об'єкти завжди малюються реального розміру і перекривають один одного в залежності від відстані до камери. Ідеальний режим камери для 2d й ізометрії. У вікні Inspector у компоненті Camera новоствореного об'єкта в полі Projection обираємо Orthographic. При цьому деякі параметри (відповідні Perspective-режиму) пропадуть, але з'явиться параметр Size – розмір ортогональної камери.

Тепер налаштуємо камеру так, щоб кожен піксель на екрані відповідав одній одиниці (unit) простору в Unity. Надалі це буде зручно при переміщенні спрайтів і в представленні їх розмірів у пікселях.

Для цього розмір ортогональної камери (параметр Size) має дорівнювати половині висоти екрану в пікселях. Наприклад, якщо це екран iPhone 3G у портретному режимі, розширення екрана якого 320x480, то Size = h / 2 = 480/2 = 240.

Для того щоб кожен раз не робити всього цього вручну, напишемо скрипт:

```
using UnityEngine;
[ExecuteInEditMode]
[RequireComponent(typeof(Camera))]
internal class Ortho2dCamera : MonoBehaviour
{
  [SerializeField] private bool uniform = true;
```

```
[SerializeField] private bool autoSetUniform = false;
private void Awake()
{
  camera.orthographic = true;
  if (uniform)
    SetUniform();
}
private void LateUpdate()
{
  if (autoSetUniform && uniform)
    SetUniform();
}
private void SetUniform()
{
  float orthographicSize = camera.pixelHeight/2;
  if (orthographicSize != camera.orthographicSize)
```

```
camera.orthographicSize = orthographicSize;
}
}
```

Якщо додати цей скрипт на будь-який ігровий об'єкт (GameObject), то: Автоматично цьому об'єкту додасться компонент Camera. За це відповідає атрибут RequireComponent.

1. Виконуючи функції Awake. За це відповідає атрибут ExecuteInEditMode, який змушує виконуватися скрипти прямо в редакторі.

2. У результаті виклику цієї функції камера стане ортогональною.

3. Її розмір буде встановлений таким чином, щоб один піксель на екрані відповідав одній одиниці Unity (виклик функції SetUniform). Це буде виконуватися автоматично для будь-якого екрану.

Покращення. Якщо розмір екрану може змінюватися під час виконання (поворот екрану смартфона, зміна дозволу користувачем), непогано б автоматично змінювати розмір камери. Це можна робити у функції LateUpdate.

Якщо освітлення використовуватися не буде (як буває у більшості 2d-ігор), рекомендується в налаштуваннях проекту (File-> Build Settings-> Player Settings-> Other Settings) встановити параметр Rendering Path у значення Vertex Lit. Це найпростіший спосіб відтворення об'єктів (кожен об'єкт за один крок для всіх джерел світла), підтримуваний більшістю пристроїв.

У випадку для iOS-пристроїв це дає стрибок у продуктивності. Те ж саме можна зробити для конкретної камери. За замовчуванням камера використовують значення з Player Settings.

Спрайт – прямокутник з накладеною на нього текстурою. Домовимося, що він за замовчуванням буде розташований в площині XOY. Тоді за взаємне розташування спрайтів (шари) буде відповідати координата Z.

Для того, щоб легше було побачити спрайт, повернути координатну вісь можна клацаючи на осях сцени, справа вгорі, поки вони не приймуть належний вигляд.

За допомогою програми Adobe Illustrator 2021, використовуючи примітиви, були створені векторні зображення. Як було сказано раніше, векторне зображення легко піддається масштабуванню без втрати якості, тому можна спочатку створити зображення, а далі підігнати їх під потрібне розширення.

Розшрення кожного зображення у пікселях може бути будь-яким, але в сфері розробок ігор прийнято брати розміри, керуючись формулою 2ЛП (64x64, 256x256 і т.д.). Всі зображення будуть збережені в растровому форматі PNG, оскільки платформа Unity не сприймає векторні формати. У зв'язку з цим, рекомендується відразу визначити розміри зображень, щоб уникнути втрат якості при перенесенні зображень в середу Unity.

Графічні елементи в комп'ютерній графіці називаються спрайтами. На рис. 3.1 представлено ігровий спрайт з підписаними розширеннями. Найдрібніші спрайти мають розмір 16x32, 32x32. Спрайти з розширенням 64x64 – це ігрові вежі. Розмір 256x256 мають персонажі супротивників.

Рисунок 3.1 – Ігрові спрайти

Далі були створені елементи інтерфейсу. Вони потрібні для взаємодії користувачів з апаратними засобами комп'ютера. Така взаємодія відбувається завдяки кнопкам, полях для введення тексту і іншим графічним елементам. На рис. 3.2. зображено створені елементи інтерфейсу – це різні панелі для виведення інформації на екран і кнопки.

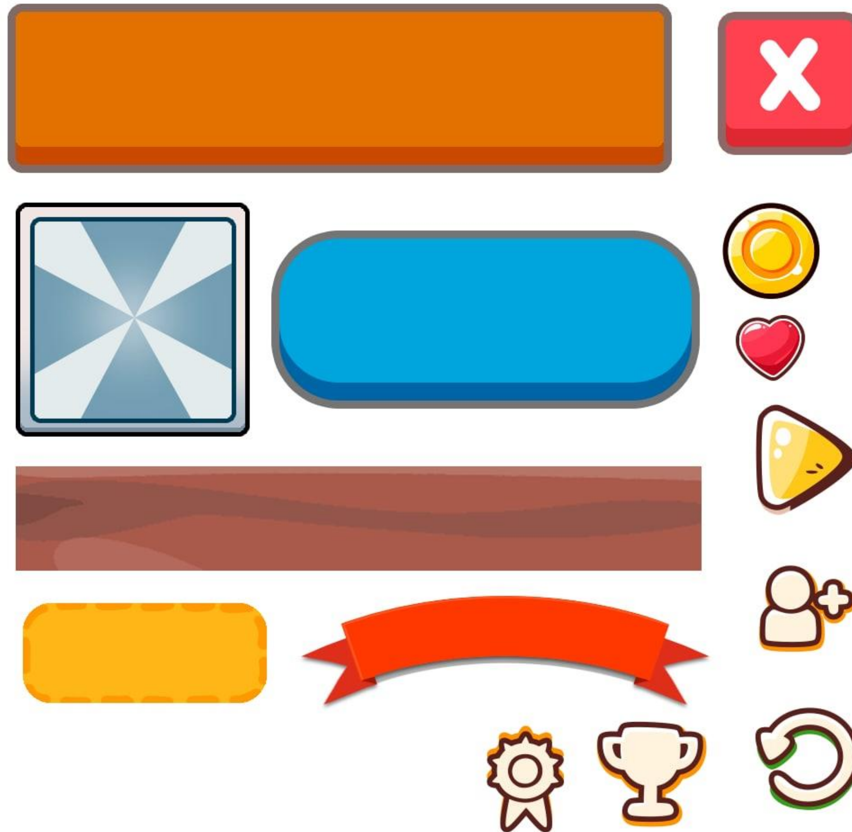


Рисунок 3.2 – Елементи інтерфейса

3.3. Створення 2D макета у середовищі Unity

3.3.1 Основи Unity

Інтерфейс у середовищі комп'ютерних розробок має назву UI. UI – це все інформаційне середовище, що відображається на екрані, і з якою взаємодіє гравець. Раніше для створення UI була потреба у написанні чималої кількості скриптів. На даний момент розроблена об'єктна модель створення інтерфейсів за допомогою візуалізації.

Для створення будь-якого UI елемента у середовищі Unity необхідно натиснути правою кнопкою миші у вікні ієрархії, обрати UI, і, відповідно, вибрати потрібний графічний елемент.

На рис. 3.3 представлено види UI в Unity.

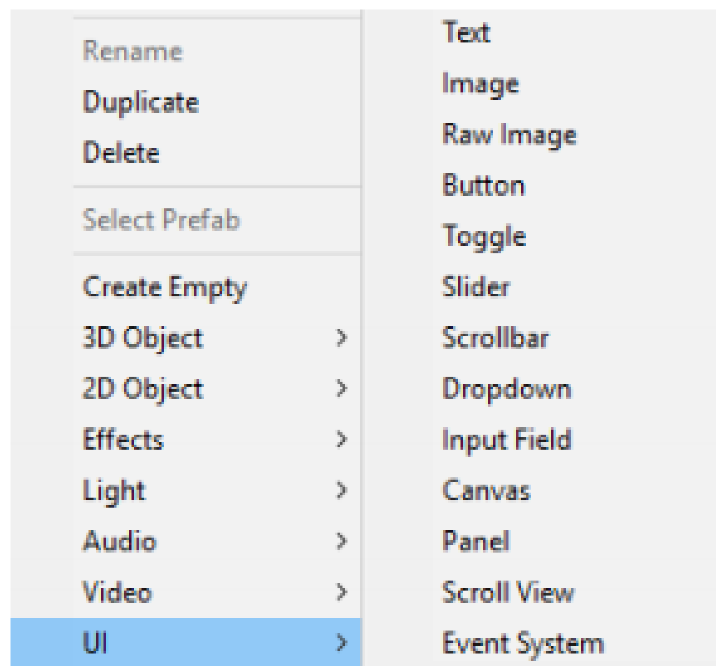


Рисунок 3.3 – UI в Unity

Будь-графічний елемент створюється дочірнім до об'єкта Canvas. Об'єктна модель створення інтерфейсу в Unity передбачає, що всі елементи мають бути дочірніми до Canvas.

Canvas (Полотно)

Canvas – це компонент, який відображає графічні елементи (картинки, текст, кнопки і т.д.).

Canvas зображений на рис. 3.4.

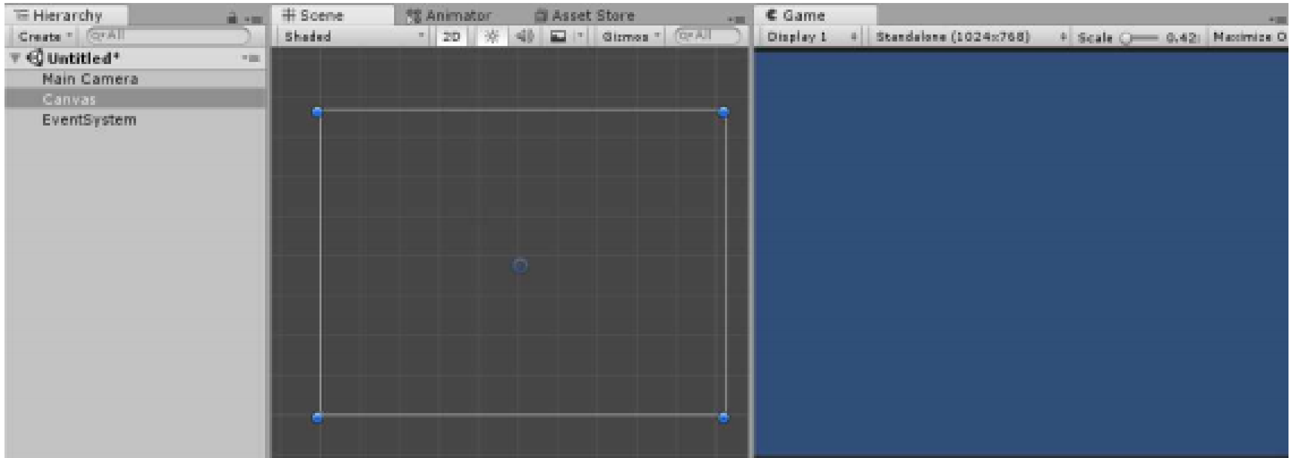


Рисунок 3.4 – Canvas

Canvas автоматично змінюється у залежності від розширення екрану. Властивості Canvas зображені на рис. 3.5.

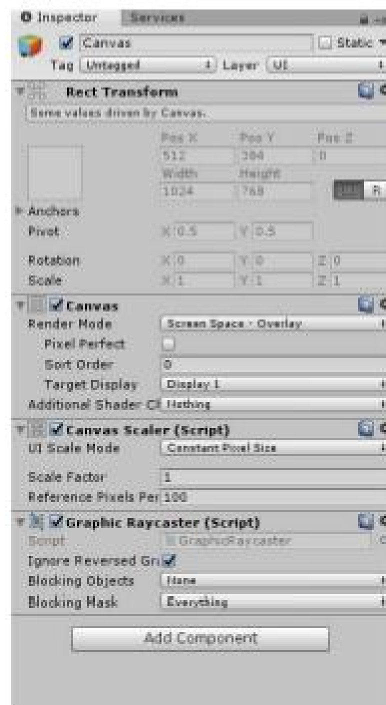


Рисунок 3.5 – Властивості Canvas

Render Mode (рис. 3.6) – властивість, яка надає можливість налаштувати графічні елементи, що знаходяться всередині Canvas та повинні відображатися на екрані.

- Screen Space – Overlay – всі графічні елементи відображаються поверх ігрової сцени. У цьому режимі розміри полотна автоматично підлаштовуються під розміри екрану.

- Screen Space – Camera – усі графічні елементи відображаються за допомогою камери. Для цього потрібно помістити елемент Camera у властивість Render Camera. Змінюючи налаштування камери, можна впливати на зовнішній вигляд Canvas.

- World Space – усі UI знаходяться в 3D просторі і вважаються звичайними 3D об'єктами.

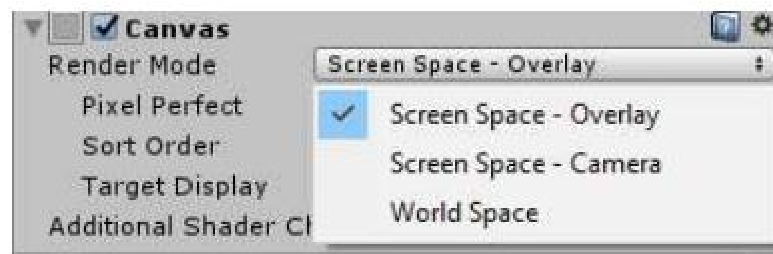


Рисунок 3.6 – Render Mode

Будь-який створений на Canvas графічний елемент можна прив'язати до однієї з дев'яти точок (до кутів, середин сторін, або до центру екрану). Для цього в панелі Rect Transform (рис. 3.7) використовується Anchor Presets (рис. 3.8).

Рисунок 3.7 – Rect Transform

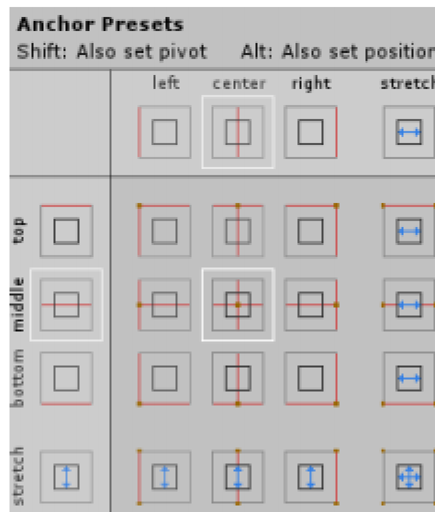


Рисунок 3.8 – Anchor Presets

Координати розташування елемента будуть відраховуватися від точки, що була обрана в якості «якоря».

Також у будь-якого елементу в панелі React Transform є властивість Pivot – це та точка на самому елементі, від якої будуть відраховуватися її координати (рис. 3.10).

Text. Являє собою елемент, що відображає текст, який можна задати в спеціальному полі, або в скриптах.

Image. За допомогою цього елемента можна розташувати на екрані будь-яке зображення. За замовчуванням об'єкт картинки створюється завжди одного і того ж розміру, але за допомогою кнопки Set Native Size в інспекторі Image можна підігнати розміри зображення під розміри спрайту.

Button. Кнопка є складовим елементом, тому текст всередині кнопки редагується в дочірньому до кнопки елементі Text. На кнопку можна натискати і обробляти це натискання. На компоненті Button є властивість, що дозволяє налаштовувати ті функції, які будуть оброблятися після натискання на кнопку. Щоб кнопка розуміла, яку функцію викликати, ця функція описується в окремому скрипті, при цьому функція має бути public. Події можна налаштовувати як на натискання кнопки (On Click ()), так і на відпускання кнопки (додатковий компонент Event Trigger).

Toggle. Це складний елемент, до якого входять декілька картинок і тексту. Текст вказується в дочірньому елементі Text, а зображення вказується в елементі Image. Даний елемент має тільки одна подія, що спрацьовує при зміні стану даного об'єкта.

Sider. За допомогою даного елемента управління можна робити будь-які налаштування, до прикладу, гучності звуків або яскравість екрану. Елемент також є складеним: складається з простіших. Завдяки вбудованій властивості, яка дозволяє слайдеру зафарбовувати ту область, яка знаходиться лівіше повзунка, його дуже часто використовують в іграх для організації в інтерфейсі таких об'єктів як смужка стану здоров'я або мани.

Scrollbar. Майже те ж саме, що Slider, тільки використовується у складі більш складних компонентів, щоб взаємодія з Scrollbar давала певний ефект.

Dropdown. Даний елемент є складовим і являє собою список, що розкривається. Елементи списку задаються в налаштуваннях самого компонента.

Input Held. Являє собою текстове поле, в яке користувач може вносити будь-які дані. У цього елемента досить багато налаштувань, основні з них:

- зміна тексту про прохання ввести будь-яку інформацію (дочірній елемент Placeholder);

- content type – властивість, що являє собою список, який розкривається та містить декілька елементів (рис. 3.9). Обравши один з них можна задати символи, які можна написати всередину Input Field (цілі числа, букви тощо);

- можливість коригування миготливого курсору. Налаштувати можна частоту миготіння, товщину, колір;

- елемент може бути однорядковим, або багаторядковим (надається можливість при натисканні клавіші Enter перейти на інший рядок).

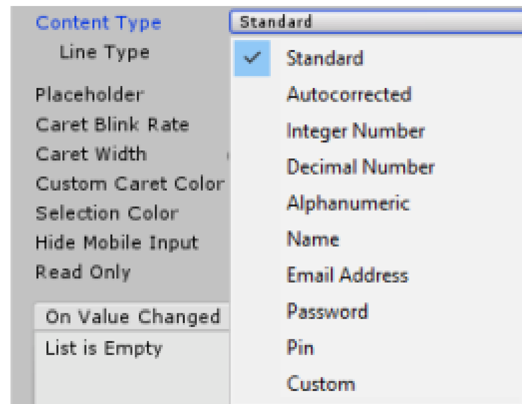


Рисунок 3.9 – Content type

На відміну від усіх попередніх елементів інтерфейсу у даного елемента присутні одразу 2 події, що спрацьовують при роботі з ним. Перша – це подія, що спрацьовує при будь-якій зміні. Друга – спрацьовує після того, як користувач закінчив друкувати.

Panel. Являє собою підкладку, на яку можна помістити будь-який інший UI.

Scroll View. Комбінований елемент, який ґрунтується на компоненті Scroll Reet. Для нормального функціонування потрібні декілька властивостей:

- площа покриття, всередині якої елемент буде прокручувати весь контент;
- контент;
- scrollbar, який буде прокручувати елементи.

3.3.2 Створення користувацького інтерфейсу

Створимо сцени: Menu і Main. Menu буде містити 2 кнопки: створення сервера і підключення. Main – основна сцена, в якій будуть відбуватися всі ігрові дії.

Перемістимо створені сцени в Scenes in build, як на рис. 3.10.

Рисунок 3.10 – Параметри складання

Тепер перейдемо до формування самих сцен. Відкриємо сцену Menu і створимо Canvas (полотно), на яке будемо розміщувати елементи інтерфейсу (рисунок 3.12).

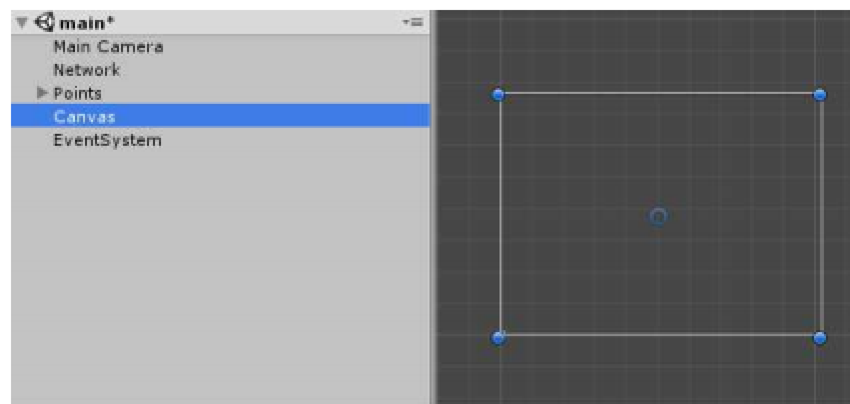


Рисунок 3.11 – Створення Canvas в сцені Menu

На рис 3.13 зображений інспектор canvas.

Рисунок 3.12 – Інспектор Canvas

У цьому завданні ми будемо використовувати властивість Render mode - Screen Space - Overlay.

Налаштуємо колір панелі. Можна також додати картинку на фон. Для цього потрібно звернути увагу на компонент Image в Інспекторі панелі.

Тепер створимо папку Sprites і помістимо туди раніше створені ігрові спрайти й елементи інтерфейсу (рисунок 3.13).

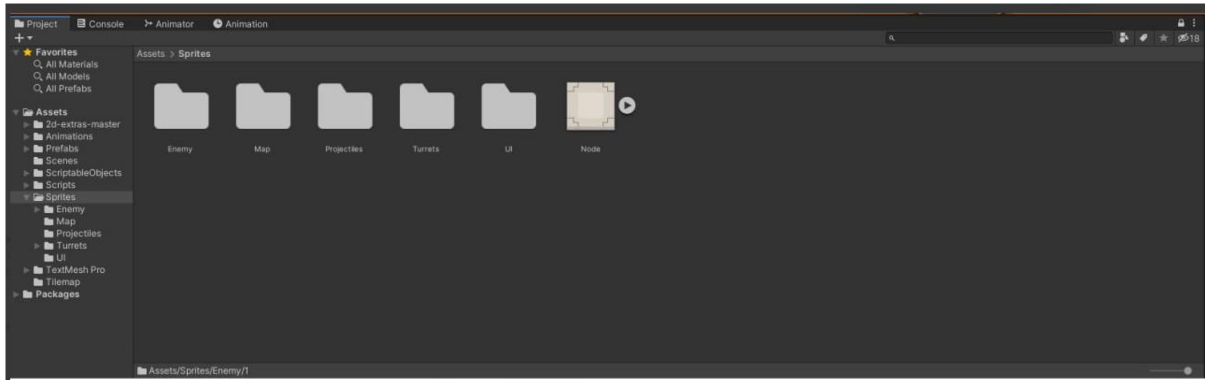


Рисунок 3.13 – Вміст папки Sprites

У папках Sprites зберігаються всі спрайти, потрібні для гри, а саме:

- enemy - спрайти з ворогами;
- map - спрайти з елементами карти;
- projectiles - спрайти зі снарядами, якими стріляють башти;
- turrets - спрайти з елементами башт;
- UI - спрайти з інтерфесом гри

Створимо дочірні UI об'єкти для Canvas: GameOverPanel, TurretShopPanel, AchievementPanel, NodeUIPanel, CoinPanel, WavePanel, GameSpeedPanel (рис. 3.14).

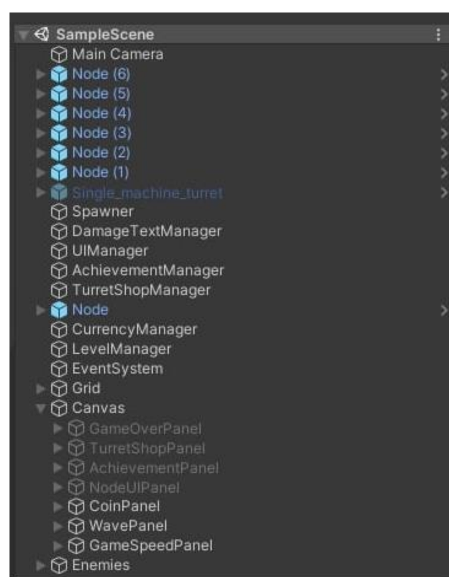


Рисунок 3.14 – Дочірні об’єкти canvas

GameOverPanel – відображається коли, гра закінчена і дозволяє поновити гру спочатку;

TurretShopPanel – відображається магазин при натисканні спеціальної комірки, коли гравцеві потрібно поставити башту;

AchievementPanel – панель з досягненями, дозволяє забрати винагороду, коли досягнуто одну з цілей (наприклад, знищити 30 ворогів);

NodeUIPanel – з’являється при натисканні на башту та дозволяє продати або зробити апгрейд башти;

CoinPanel – відображає доступну кількість золота яку можна витратити на покупку башт;

WavePanel – відображає яка саме хвиля ворогів на данний момент;

GameSpeedPanel – кнопки які дозволяють прискорити гру, або вповільнити, або зробити нормальну швидкість.

На рис. 3.15 зображено підсумковий вид сцени Menu

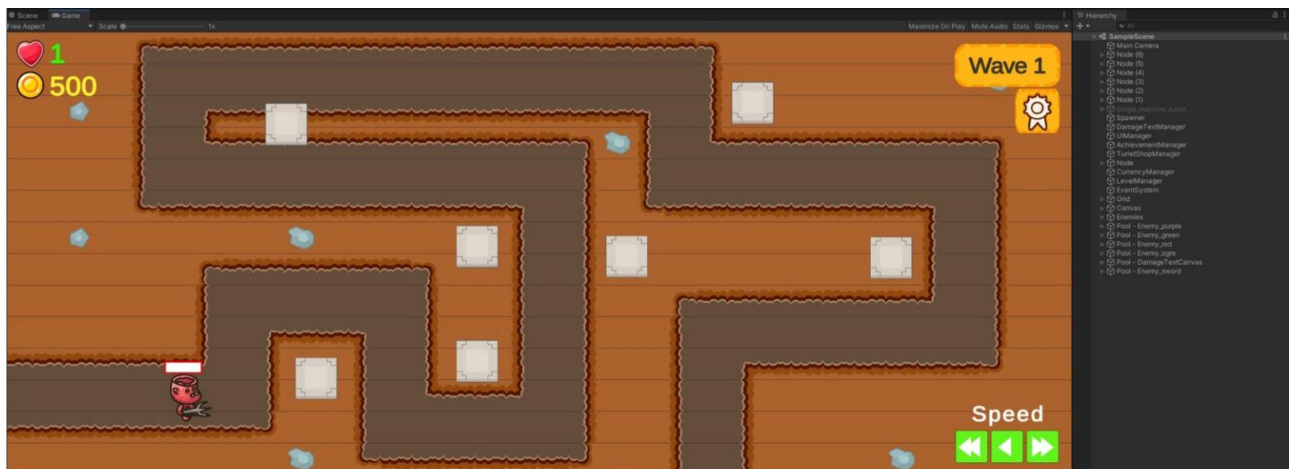


Рисунок 3.15 Підсумковий вид сцени Menu

Така інтерактивність реалізується за допомогою скриптів і буде розглядатися далі.

3.4 Створення сінглплеєра

Створена гра розрахована, що в неї гратиме тільки один користувач.

Відеогра розрахована на одного гравця – це відео гра, в якій очікується участь тільки одного гравця впродовж ігрового сеансу. Зазвичай, однокористувальницька гра є грою, в яку може грати лише одна особа. «режим одного користувача» – призначений, як правило, для одиночного гравця ігровий режим, хоча гра також містить багатокористувацькі режими.

Більша частина сучасних аркад і консольних ігор розроблені таким чином, щоб у них могла грати лише одна людина-гравець; У багатьох з цих ігор є також режими, що дають можливість грати двом і більше гравцям (не обов'язково водночас). При цьому, мало хто з них вимагає насправді для гри більше, ніж одного гравця.

Найважливішим елементом в іграх є мультиплеер. Був створений спеціальний режим комп'ютерних ігор – singleplayer, щоб грати могла тільки одна людина. Щоб реалізувати в середовищі Unity сінглплеер, все вже передбачено.

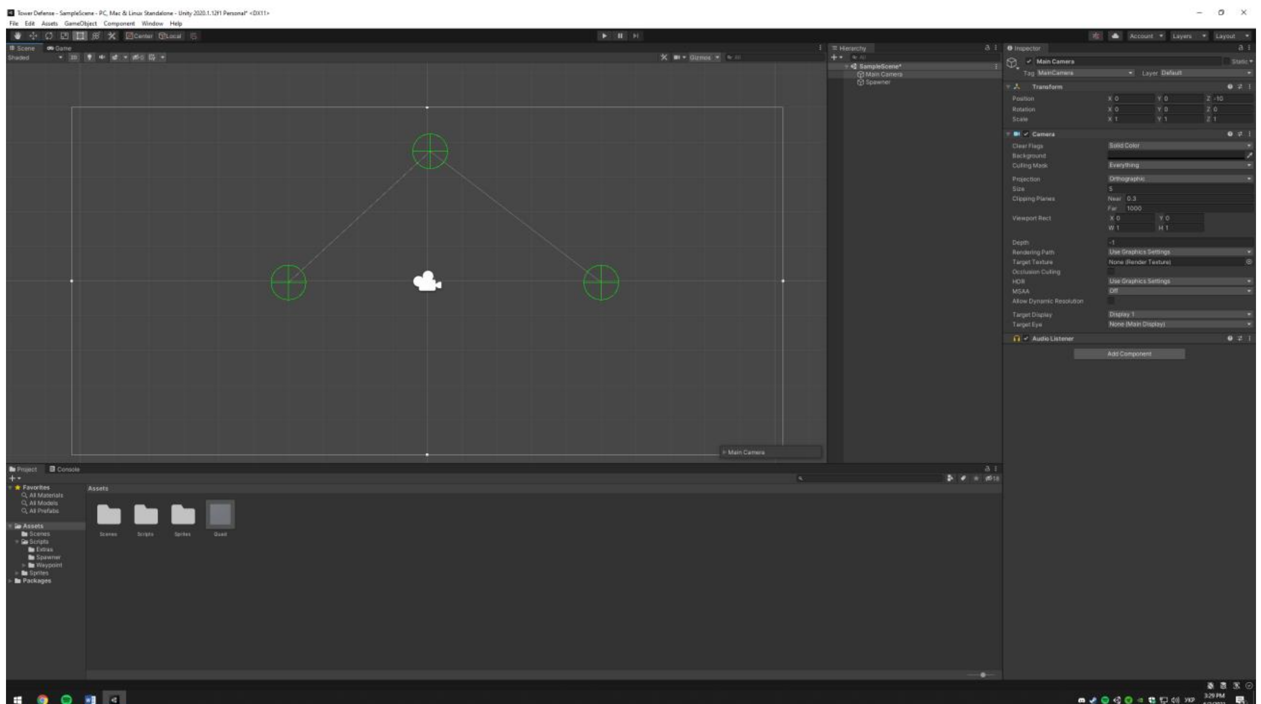
3.5. Скриптинг

WayPoint Class – це одна із основних функцій в грі Tower Defence, що надає можливість прокладати маршрут або шлях для ворог. Це реалізовано за допомогою WayPoint class. Як це працює: ми можемо сказати що наш шлях – це група позиції у просторі і означити на мові коду, що наш шлях це группа vector (ця структура використовується в Unity для передачі 3D-позицій та напрямків навколо. Він також містить функції для виконання загальних векторних операцій), позицій які будуть зберігатися в масиві.

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

```
using UnityEngine;

public class Waypoint : MonoBehaviour
{
    [SerializeField] private Vector3[] points;

    public Vector3[] Points => points;
    public Vector3 CurrentPosition => _currentPosition;

    private Vector3 _currentPosition;
    private bool _gameStarted;

    // Start is called before the first frame update
    private void Start()
    {
        _gameStarted = true;
        _currentPosition = transform.position;
    }

    public Vector3 GetWaypointPosition(int index)
    {
        return CurrentPosition + Points[index];
    }

    private void OnDrawGizmos()
    {
        if (!_gameStarted && transform.hasChanged)
        {
            _currentPosition = transform.position;
```

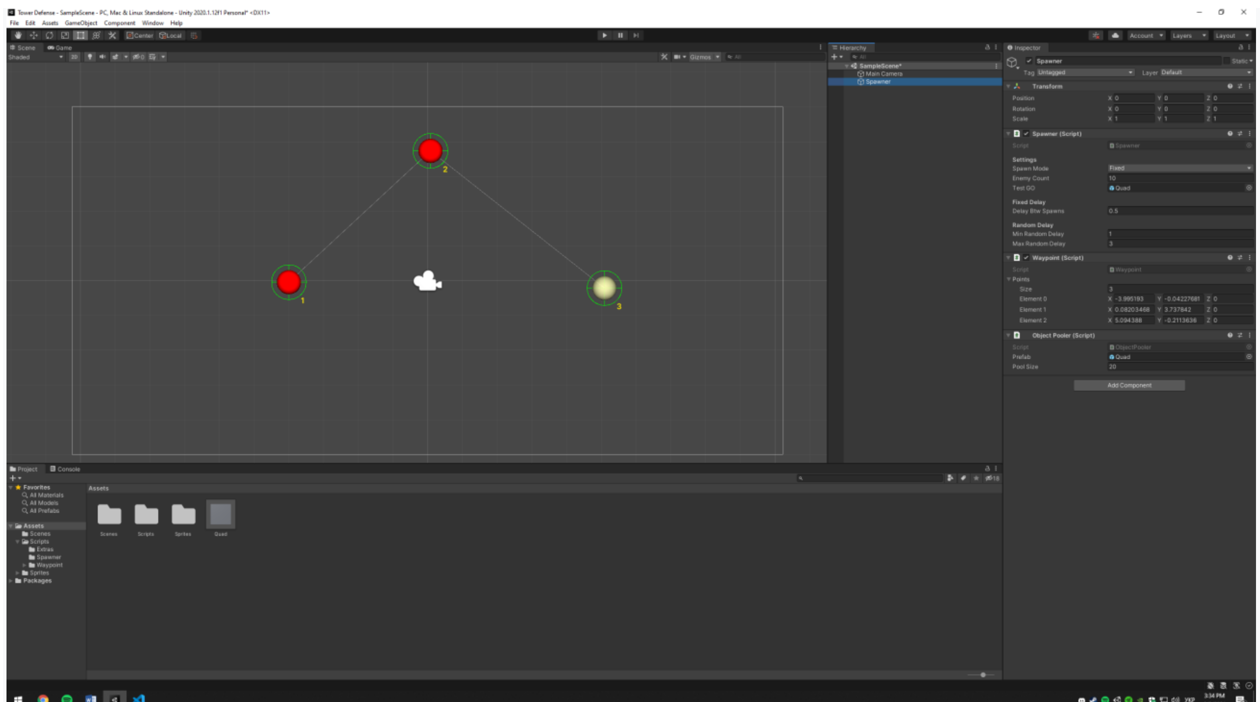
```
            }

            for (int i = 0; i < points.Length; i++)
            {
                Gizmos.color = Color.green;
                Gizmos.DrawWireSphere(points[i] + _currentPosit
ion, 0.5f);

                if (i < points.Length - 1)
                {
                    Gizmos.color = Color.gray;
                    Gizmos.DrawLine(points[i] + _currentPositio
n, points[i + 1] + _currentPosition);
                }
            }
        }
    }
```



WayPoint Editor – для того, щоб було зручно гравцеві виставляти наші waypoint'и і потрібно зрибити редактор для них. Ідея полягає в тому, щоб ми могли рухати наші waypoint'и за допомогою миші.

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

[CustomEditor(typeof(Waypoint))]
public class WaypointEditor : Editor
{
   Waypoint Waypoint => target as Waypoint;
   private void OnSceneGUI()
   {
      Handles.color = Color.red;
      for (int i = 0; i < Waypoint.Points.Length; i++)
      {
         EditorGUI.BeginChangeCheck();

         // Create Handles
         Vector3 currentWaypointPoint = Waypoint.CurrentPosition + Waypoint.
Points[i];
         Vector3 newWaypointPoint = Handles.FreeMoveHandle(currentWaypoi
ntPoint,
            Quaternion.identity, 0.7f,
            new Vector3(0.3f, 0.3f, 0.3f), Handles.SphereHandleCap);

         // Create text
         GUIStyle textStyle = new GUIStyle();
```

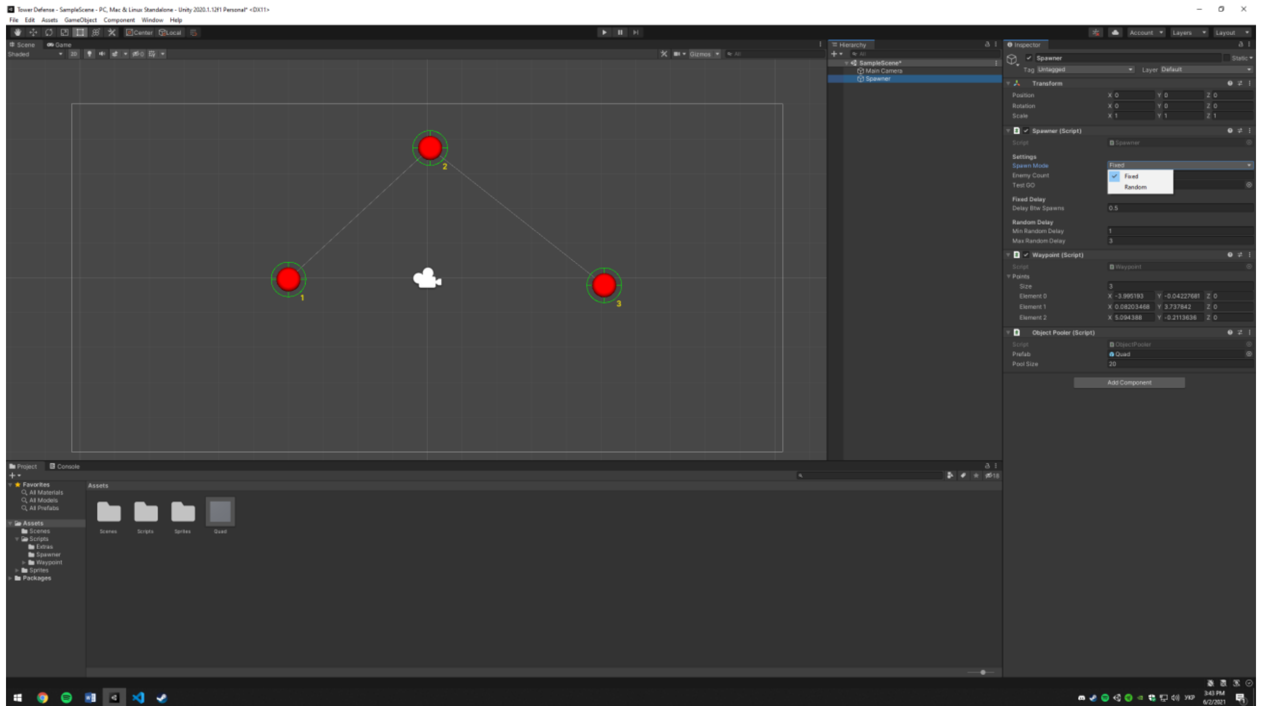
```
            textStyle.fontStyle = FontStyle.Bold;
            textStyle.fontSize = 16;
            textStyle.normal.textColor = Color.yellow;
            Vector3 textAlligment = Vector3.down * 0.35f + Vector3.right * 0.35f;
            Handles.Label(Waypoint.CurrentPosition + Waypoint.Points[i] + textAlli
gment,
              $"{i + 1}", textStyle);
            EditorGUI.EndChangeCheck();

            if (EditorGUI.EndChangeCheck())
            {
              Undo.RecordObject(target, "Free Move Handle");
              Waypoint.Points[i] = newWaypointPoint - Waypoint.CurrentPosition;
            }
          }
        }
    }
```



Spawner Class – цей скріпт нам потрібен для того, щоб спавнити ворогів у нащій грі. Є два режими, в яких він може працювати, перший це fixed час між спавном ворогів виставляється вручну і не змінюється впордовж гри, random виставляється певний проміжок часу, між яким буде проходити спавн ворогів.

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

public enum SpawnModes
{
    Fixed,
    Random
}

public class Spawner : MonoBehaviour
{
    [Header("Settings")]
    [SerializeField] private SpawnModes spawnMode = SpawnModes
.Fixed;
    [SerializeField] private int enemyCount = 10;
    [SerializeField] private GameObject testGO;

    [Header("Fixed Delay")]
    [SerializeField] private float delayBtwSpawns;

    [Header("Random Delay")]
    [SerializeField] private float minRandomDelay;
    [SerializeField] private float maxRandomDelay;

    private float _spawnTimer;
    private int _enemiesSpawned;

    private ObjectPooler _pooler;

    private void Start()
    {
        _pooler = GetComponent<ObjectPooler>();
    }

    private void Update()
```

```
    {
        _spawnTimer -= Time.deltaTime;
        if (_spawnTimer < 0)
        {
            _spawnTimer = GetSpawnDelay();
            if (_enemiesSpawned < enemyCount)
            {
                _enemiesSpawned++;
                SpawnEnemy();
            }
        }
    }

    private void SpawnEnemy()
    {
        GameObject newInstance = _pooler.GetInstanceFromPool()
;
        newInstance.SetActive(true);
    }

    private float GetSpawnDelay()
    {
        float delay = 0f;
        if (spawnMode == SpawnModes.Fixed)
        {
            delay = delayBtwSpawns;
        }
        else
        {
            delay = GetRandomDelay();
        }

        return delay;
    }

    private float GetRandomDelay()
    {
        float randomTimer = Random.Range(minRandomDelay, maxRa
ndomDelay);
        return randomTimer;
    }
}
```



Object pooler. Призначений для того, щоб покращити продуктивність гри. Як працює: при старті гри створює одразу декілька об’єктів (ворогів) і після того, як вони були знищені в самій грі баштами не видаляє їх за допомогою GarbegeCollector, а повертає в пул створених об’єктів для подальшого використання і тим самим зменшує навантаження на залізо пристрою.

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

public class ObjectPooler : MonoBehaviour
{
    [SerializeField] private GameObject prefab;
    [SerializeField] private int poolSize = 10;

    private List<GameObject> _pool;
    private GameObject _poolContainer;
```

```
    private void Awake()
    {
        _pool = new List<GameObject>();
        _poolContainer = new GameObject($"Pool -
  {prefab.name}");

        CreatePooler();
    }

    private void CreatePooler()
    {
        for (int i = 0; i < poolSize; i++)
        {
            _pool.Add(CreateInstance());
        }
    }

    private GameObject CreateInstance()
    {
        GameObject newInstance = Instantiate(prefab);
        newInstance.transform.SetParent(_poolContainer.transfo
rm);
        newInstance.SetActive(false);
        return newInstance;
    }

    public GameObject GetInstanceFromPool()
    {
        for (int i = 0; i < _pool.Count; i++)
```

```
        {
            if (!_pool[i].activeInHierarchy)
            {
                return _pool[i];
            }
        }

        return CreateInstance();
    }
}
```

Enemy. Призначений для визначення поведінки супротивників: рух по waypoint'ам, швидкість руху ворогів. По досогненню останньої точки нашого маршруту, повертає перезавантаженого ворога назад в ObjectPooler, а також віднімає одне життя у гравця.

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

public class Enemy : MonoBehaviour
{
    public static Action<Enemy> OnEndReached;

    [SerializeField] private float moveSpeed = 3f;

    /// <summary>
    /// Move speed of our enemy
    /// </summary>
    public float MoveSpeed { get; set; }
```

```
    /// <summary>
    /// The waypoint reference
    /// </summary>
    public Waypoint Waypoint { get; set; }

    /// <summary>
    /// Returns the current Point Position where this enemy ne
eds to go
    /// </summary>
    public Vector3 CurrentPointPosition => Waypoint.GetWaypoin
tPosition(_currentWaypointIndex);

    private int _currentWaypointIndex;
    private Vector3 _lastPointPosition;

    private EnemyHealth _enemyHealth;
    private SpriteRenderer _spriteRenderer;

    private void Start()
    {
        _enemyHealth = GetComponent<EnemyHealth>();
        _spriteRenderer = GetComponent<SpriteRenderer>();

        _currentWaypointIndex = 0;
        MoveSpeed = moveSpeed;
        _lastPointPosition = transform.position;
    }
```

```
    private void Update()
    {
        Move();
        Rotate();

        if (CurrentPointPositionReached())
        {
            UpdateCurrentPointIndex();
        }
    }

    private void Move()
    {
        transform.position = Vector3.MoveTowards(transform.pos
ition,
            CurrentPointPosition, MoveSpeed * Time.deltaTime);
    }

    public void StopMovement()
    {
        MoveSpeed = 0f;
    }

    public void ResumeMovement()
    {
        MoveSpeed = moveSpeed;
    }

    private void Rotate()
```

```
    {
        if (CurrentPointPosition.x > _lastPointPosition.x)
        {
            _spriteRenderer.flipX = false;
        }
        else
        {
            _spriteRenderer.flipX = true;
        }
    }

    private bool CurrentPointPositionReached()
    {
        float distanceToNextPointPosition = (transform.positio
n - CurrentPointPosition).magnitude;
        if (distanceToNextPointPosition < 0.1f)
        {
            _lastPointPosition = transform.position;
            return true;
        }

        return false;
    }

    private void UpdateCurrentPointIndex()
    {
        int lastWaypointIndex = Waypoint.Points.Length - 1;
        if (_currentWaypointIndex < lastWaypointIndex)
        {
```

```
                _currentWaypointIndex++;
            }
            else
            {
                EndPointReached();
            }
        }

        private void EndPointReached()
        {
            OnEndReached?.Invoke(this);
            _enemyHealth.ResetHealth();
            ObjectPooler.ReturnToPool(gameObject);
        }

        public void ResetEnemy()
        {
            _currentWaypointIndex = 0;
        }
    }
```

LevelMeneger. Призначений для відслідковування кількості наших життів у грі, а також тримає в собі умови завершення гри, тобто коли кількість життів становить «0», то гра завершується.

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

public class LevelManager : MonoBehaviour
{
```

```
    [SerializeField] private int lives = 10;

    public int TotalLives { get; set; }

    private void Start()
    {
        TotalLives = lives;
    }

    private void ReduceLives(Enemy enemy)
    {
        TotalLives--;
        if (TotalLives <= 0)
        {
            TotalLives = 0;
            // Game Over
        }
    }

    private void OnEnable()
    {
        Enemy.OnEndReached += ReduceLives;
    }

    private void OnDisable()
    {
        Enemy.OnEndReached -= ReduceLives;
    }
}
```

Turret. Задає модель поведінки башти, а саме, дає можливість створити башту, а також відслідковує цілі, якщо вони увійшли в область ураження баштою.

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

public class Turret : MonoBehaviour
{
    [SerializeField] private float attackRange = 3f;

    public Enemy CurrentEnemyTarget { get; set; }

    private bool _gameStarted;
    private List<Enemy> _enemies;

    private void Start()
    {
        _gameStarted = true;
        _enemies = new List<Enemy>();
    }

    private void Update()
    {
        GetCurrentEnemyTarget();
        RotateTowardsTarget();
    }

    private void GetCurrentEnemyTarget()
```

```
    {
        if (_enemies.Count <= 0)
        {
            CurrentEnemyTarget = null;
            return;
        }

        CurrentEnemyTarget = _enemies[0];
    }

    private void RotateTowardsTarget()
    {
        if (CurrentEnemyTarget == null)
        {
            return;
        }

        Vector3 targetPos = CurrentEnemyTarget.transform.posit
ion - transform.position;
        float angle = Vector3.SignedAngle(transform.up, target
Pos, transform.forward);
        transform.Rotate(0f, 0f, angle);
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Enemy"))
        {
            Enemy newEnemy = other.GetComponent<Enemy>();
```

```
                _enemies.Add(newEnemy);
            }
        }

        private void OnTriggerExit2D(Collider2D other)
        {
            if (other.CompareTag("Enemy"))
            {
                Enemy enemy = other.GetComponent<Enemy>();
                if (_enemies.Contains(enemy))
                {
                    _enemies.Remove(enemy);
                }
            }
        }

        private void OnDrawGizmos()
        {
            if (!_gameStarted)
            {
                GetComponent<CircleCollider2D>().radius = attackRa
    nge;
            }

            Gizmos.DrawWireSphere(transform.position, attackRange)
    ;
        }
    }
```

TurretProjectile

Задає логіку для поведінки снаряду, який випущений з башти, вказує кількість урону який буде нанесений ворогу, а також швидкість стрільби з башти.

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

public class TurretProjectile : MonoBehaviour
{
    [SerializeField] protected Transform projectileSpawnP
osition;
    [SerializeField] protected float delayBtwAttacks = 2f
;

    protected float _nextAttackTime;
    protected ObjectPooler _pooler;
    protected Turret _turret;
    protected Projectile _currentProjectileLoaded;

    private void Start()
    {
        _turret = GetComponent<Turret>();
        _pooler = GetComponent<ObjectPooler>();

        LoadProjectile();
    }

    protected virtual void Update()
```

```
    {
        if (IsTurretEmpty())
        {
            LoadProjectile();
        }

        if (Time.time > _nextAttackTime)
        {
            if (_turret.CurrentEnemyTarget != null && _cu
rrentProjectileLoaded != null &&
                _turret.CurrentEnemyTarget.EnemyHealth.Cu
rrentHealth > 0f)
            {
                _currentProjectileLoaded.transform.parent
 = null;
                _currentProjectileLoaded.SetEnemy(_turret
.CurrentEnemyTarget);
            }

            _nextAttackTime = Time.time + delayBtwAttacks
;
        }
    }

    protected virtual  void LoadProjectile()
    {
        GameObject newInstance = _pooler.GetInstanceFromP
ool();
```

```
        newInstance.transform.localPosition = projectileS
pawnPosition.position;
        newInstance.transform.SetParent(projectileSpawnPo
sition);

        _currentProjectileLoaded = newInstance.GetCompone
nt<Projectile>();
        _currentProjectileLoaded.TurretOwner = this;
        _currentProjectileLoaded.ResetProjectile();
        newInstance.SetActive(true);
    }

    private bool IsTurretEmpty()
    {
        return _currentProjectileLoaded == null;
    }

    public void ResetTurretProjectile()
    {
        _currentProjectileLoaded = null;
    }
}
```

Node

Визначає місце на карті при натисканні, на якому у нас відкривається магазин для покупки башт, або ж якщо на цьому місці вже стоїть башта, то дозволяє проапгрейдити її або ж продати.

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

```
public class Node : MonoBehaviour
{
    public static Action<Node> OnNodeSelected;
    public static Action OnTurretSold;

    [SerializeField] private GameObject attackRangeSprite
;

    public Turret Turret { get; set; }

    private float _rangeSize;
    private Vector3 _rangeOriginalSize;

    private void Start()
    {
        _rangeSize = attackRangeSprite.GetComponent<Sprit
eRenderer>().bounds.size.y;
        _rangeOriginalSize = attackRangeSprite.transform.
localScale;
    }

    public void SetTurret(Turret turret)
    {
        Turret = turret;
    }

    public bool IsEmpty()
    {
        return Turret == null;
```

```
    }

    public void CloseAttackRangeSprite()
    {
        attackRangeSprite.SetActive(false);
    }

    public void SelectTurret()
    {
        OnNodeSelected?.Invoke(this);
        if (!IsEmpty())
        {
            ShowTurretInfo();
        }
    }

    public void SellTurret()
    {
        if (!IsEmpty())
        {
            CurrencySystem.Instance.AddCoins(Turret.Turre
tUpgrade.GetSellValue());
            Destroy(Turret.gameObject);
            Turret = null;
            attackRangeSprite.SetActive(false);
            OnTurretSold?.Invoke();
        }
    }
```

```
    private void ShowTurretInfo()
    {
        attackRangeSprite.SetActive(true);
        attackRangeSprite.transform.localScale = _rangeOr
iginalSize * Turret.AttackRange / (_rangeSize / 2);
    }
}
```

ВИСНОВКИ

Отже, ігри жанру Tower Defense нині користуються великою популярністю, сутність яких полягає знищенні супротивників, не допустити їх проходження до кінця ігрового поля. При цьому супротивника можна знищувати тільки будуючи вежі, які гравець має можливість розташовувати вздовж ігрового поля, яким рухаються супротивники.

Розглянуто особливості балансу мобільного додатку жанру Tower Defense, який є одним з найскладніших аспектів ігробудування та багато в чому визначає плавність, інтерес і складність самого ігрового процесу. Також розглянуто особливості механіки гри, тобто набір способів і правил, за допомогою якого формується конкретна реалізацію ігрового процесу. Представлено визначення граничних умов гри, розраховано баланс ігрових веж та хвиль і економіки гри.

Існує чимала кількість різновидів ігр у жанрі Tower Defense для різних платформ – PC, Xbox 360, Android, iOS, PlayStation 4, Xbox One, PlayStation Vita, Nintendo 3DS, Nintendo Switch, Mac, Linux, – які мають певні несуттєві відмінності. Серед найкращих ігор жанру Tower Defense одними з найбільш популярних є: Plants vs. Zombie, Orcs Must Die, Plants vs. Zombies 2: Garden Warfare, Orcs Must Die! 2, Anomaly: Warzone Earth, Plants vs. Zombies 2: It's About Time, Deathtrap, Rock of Ages, Sanctum 2 та ін.

Також було визначено і освоєні інструментальні засоби, за допомогою яких буде відбувається реалізація ігрового проекту. Основний і найбільш важливою програмою для розробки ігор є движок, в нашому випадку є Unity. Unity використовує мову C# і має закритий вихідний код.

При складанні концептуальної моделі було прийнято рішення створювати гру в жанрі C#. Виходячи з цього, були підібрані і створені ігрові спрайт і елементи інтерфейсу в програмі Adobe Illustrator 2021, з подальшим впровадженням їх у середу Unity.

При програмуванні в середовищі MonoDevclop на движку Unity, були створені скрипти для ігрових кнопок, лічильника і різних операцій над ресурсами. Також був реалізований синглплеер – режим гри, при якому одночасно грає одна людина.

Таким чином, в процесі написання бакалаврської роботи було здійснено наступне:

- проведено аналіз предметної області;
- освоєні інструментальні засоби;
- створена концептуальна модель;
- здійснено відмалювання дизайну;
- здійснена програмна реалізація ігрового проекту.

ПЕРЕЛІК ПОСИЛАНЬ

1. Горовых И.И. Описание баланса и механики мобильного приложения жанра Tower Defense / И.И. Горовых // Молодой следователь Дона. – 2019. – № 5(20). – С. 12-17.

2. Движок Unity – особенности, преимущества и недостатки [Электронный ресурс]// Cubiq.ru – игровой портал URL: https://cubiq.ru/dvizhok-unity/ (дата звернення: 29.05.2020)

3. Зайнутдинов А.Д. Разработка игрового приложения жанра платформер на движке unity с использованием языка программирования C#: «LEGENDS OF PLANET Z-ZERRA: BLACK AND WHITE WORLD» / А.Д. Зайнутдинов // Старт в науке. – 2018. - № 5. – С. 372-377.

4. Использование AI и утилит при разработке игр жанра Tower Defense. URL: http://habrahabr.ru/post/189198/ (дата звернення: 29.05.2020)

5. Наумов Р.В. Актуальные языки программирования / Р.В. Наумов // Academy. – 2016. – № 7. – С. 15-18.

6. Нелипа С. Рынок игровой индустрии в 2020 году вырос почти на 20%. URL: https://www.igromania.ru/news/101001/Rynok_igrovoy_industrii_v_2020_godu_vyros_pochti_na_20perc.html (дата звернення: 29.05.2020)

7. Рейтинг языков программирования в Украине в 2021 году: доля Python уменьшилась впервые с 2014 года [Электронный ресурс]. – Режим доступа: https://ain.ua/2021/02/15/top-yazykov-programmirovaniya-v-ukraine-2021/ (дата звернення: 29.05.2020)

8. Усков М.А. Обзор преимуществ и недостатков игровых движков. обоснование выбора инструментов и технологий разработки клиентской части игровых приложений https://cyberleninka.ru/article/n/obzor-preimuschestv-i-nedostatkov-igrovyh-dvizhkov-obosnovanie-vybora-instrumentov-i-tehnologiy-razrabotki-klientskoy-chasti (дата звернення: 29.05.2020)

9. Understanding Tower Defense games [Електронний ресурс] – Режим доступу: http: //www.loopinsight.cont 2010 03 30 understanding -tower-defense-games (дата звернення: 29.05.2020)