МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ, КОМП'ЮТЕРНОЇ
ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ
КАФЕДРА ПРИКЛАДНОЇ ІНФОРМАТИКИ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
_____В.П. Гамаюн

«_____»_____2021 р.

ДИПЛОМНИЙ ПРОЕКТ
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСНИКА ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА
ЗА СПЕЦІАЛЬНІСТЮ
122 «КОМП'ЮТЕРНІ НАУКИ ТА ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ»

Тема: «Кросплатформний 2D-додаток для ігрового застосування на Unity»

Виконавець: Федорищев Микита Юрійович

Керівник: Гамаюн Володимир Петрович

Нормоконтролер: Боровик Володимир Миколайович

КИЇВ 2021

ВСТУП

У Unity скрипти можна використовувати для розробки практично будь-якого елементу гри або інтерактивного контенту з графікою реального часу. Unity підтримує скрипти на C #, створені відповідно до одного з двох основних підходів: традиційним і об'єктно-орієнтованим підходом, що широко використовується і інформаційно-орієнтованим підходом, який тепер теж підтримується в Unity, в окремих випадках завдяки високопродуктивному багатопоточному стеку інформаційно-орієнтованих технологій (DOTS).

Unity підтримує C #, стандартну в галузі програмування мову, в деякій мірі C# схожий на Java або C ++. У порівнянні з C ++, C # легше в вивченні. Крім того, він відноситься до категорії мов «з керуванням пам'яттю», тобто він автоматично розподіляє пам'ять, усуває витоку і так далі. Як правило C # зручніше C ++, якщо вам потрібно в першу чергу розробити гру, а потім вже працювати над різними складними аспектами програмування.

Традиційна модель «ігровий об'єкт - компонент» добре працює і сьогодні, оскільки вона проста як для програмістів, так і для інших користувачів, а також зручна для створення інтуїтивних інтерфейсів. Додайте компонент Rigidbody до об'єкта GameObject - він почне падати, додайте компонент Light - GameObject почне випромінювати світло. Все інше також підпорядковується цій простій логіці.

При цьому система компонентів створена на основі об'єктно-орієнтованої платформи, що створює складності для розробників при роботі з кешем і пам'яттю обладнання що розвивається.

Компоненти і ігрові об'єкти відносяться до «важких об'єктів C ++». Всі об'єкти GameObject мають ім'я. Їх компоненти являють собою оболонки для C # поверх компонентів на C ++. Це спрощує роботу з ними, але може впливати на продуктивність, якщо вони будуть зберігатися в пам'яті без явної структури. Об'єкт C # може перебувати на будь-якій ділянці пам'яті. Об'єкт C ++ також може знаходитися в будь-якій ділянці пам'яті. Угруповання і послідовне розміщення об'єктів в пам'яті

відсутні. При кожному завантаженні в центральний процесор для обробки об'єкт доводиться збирати по частинах з різних ділянок пам'яті. Це може сильно уповільнити завантаження, а оптимізація потребує багато зусиль.

.NET - раніше в Unity використовувалась реалізація стандартного середовища виконання Mono з нативної підтримкою C #. Тепер Unity для MacOS поставляється з Visual Studio for Mac замість MonoDevelop-Unity. Unity для Windows поставляється з Visual Studio.

Середовище програмування .NET 4.6 в Unity підтримує більшість існуючих функцій C # і дозволяє здійснювати налагодження для мови C # версії 6.0 і вище. Таке рішення також забезпечує зручність роботи в IDE зі скриптами на C # з підтримкою нових можливостей C #.

IL2CPP: система програмування від Unity, яку можна використовувати в якості альтернативи Mono при роботі над проектами для деяких платформ. Якщо ви створюєте проект з використанням IL2CPP, Unity перетворює код IL з скриптів і виконує складання в код C ++, а потім створює нативний двійковий файл (наприклад, .exe, apk, .xap) для обраної платформи. IL2CPP - це єдиний варіант системи програмування проектів, призначених для iOS і WebGL.

Для вирішення цих проблем почали переробляти базові системи Unity на основі високопродуктивного, багатопоточного стеку інформаційно-орієнтованих технологій або DOTS.

DOTS дозволяє вашій грі ефективно використовувати всі можливості новітніх багатоядерних процесорів. Стек складається з наступних компонентів:

- система завдань C # для ефективного виконання коду на багатопотокових системах;
- Entity Component System (ECS) для розробки високопродуктивного коду за замовчуванням;
- компілятор Burst для компіляції скриптів в оптимізований нативний код.

ECS - це нова система компонентів в складі DOTS. Всі традиційні об'єктно-орієнтовані маніпуляції над GameObject відображаються на примірнику в новій системі. Назва «Компонент» ніяк не змінилась. Найважливіша відмінність - в структурі даних.

Редактор Unity має інтерфейс, що складається з різних вікон, які можна розташувати на свій розсуд. Завдяки цьому можна проводити налагодження гри чи застосунка прямо в редакторі. Головні вікна — це оглядач ресурсів проєкту, інспектор поточного об'єкта, вікно попереднього перегляду, оглядач сцени та оглядач ієрархії ресурсів.

Проект в Unity поділяється на сцени (рівні) — окремі файли, що містять свої ігрові світи зі своїм набором об'єктів, сценаріїв, і налаштувань. Сцени можуть містити в собі як об'єкти-моделі (ландшафт, персонажі, предмети довкілля тощо), так і порожні ігрові об'єкти — ті, що не мають моделі, проте задають поведінку інших об'єктів (тригери подій, точки збереження прогресу тощо). Їх дозволяється розташовувати, обертати, масштабувати, застосовувати до них скрипти. В них є назва (в Unity допускається наявність двох і більше об'єктів з однаковими назвами), може бути тег (мітка) і шар, на якому він повинен відображатися. Так, у будь-якого предмета на сцені обов'язково наявний компонент Transform — він зберігає в собі координати місця розташування, повороту і розмірів по всіх трьох осях. У об'єктів з видимою геометрією також за умовчанням присутній компонент Mesh Renderer, що робить модель видимою. Різні моделі можуть об'єднуватися в набори (ассети) для швидкого доступу до них. Наприклад, моделі споруд на спільну тему.

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

У 2D іграх Unity переважно використовує спрайти. В 3D іграх Unity здебільшого використовує тривимірні моделі (меші), на які накладаються текстури (зумовлюють

вигляд поверхні об'єктів), матеріали (зумовлюють як поверхня реагуватиме на різні фактори) та шейдери (невеликі скрипти, за яким вираховується зміна кольору кожного пікселя згідно заданих параметрів, як-от розсіяння відбитого світла). В обох видах застосовуються системи часток для відображення субстанцій, таких як рідини чи дим.

Unity підтримує стиснення текстур, міпмапінг і різні налаштування роздільності екрана для кожної платформи; забезпечує бамп-мапінг, мапінг відображень, паралакс-мапінг, затінення навколишнього світла у екранному просторі, динамічні тіні за картами тіней, рендер у текстуру та повноекранні ефекти обробки зображення, такі як зернистість, глибина чіткості, розмиття в русі, відблиски віртуальних лінз або ореол навколо джерел світла.

Рендеринг зображення відбувається через віртуальну камеру огляду. В робочій області редактора ігрова сцена може розміщуватися як завгодно, а при рендерингу — так, як її видно з камери. В сцені може бути декілька камер, які рухаються за персонажем чи за вказаною траєкторією. Вигляд з камери подається в двовимірно чи тривимірно (в перспективі або ортографічно). Фон сцени, видимий через камеру, типово зображає небо, утворене скайбоксом, але може презентувати й інше довкілля.

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

Unity підтримує файли 3ds Max, Maya, Softimage, Blender, modo, ZBrush, Cinema 4D, Cheetah3D, Adobe Photoshop, Adobe Fireworks та Allegorithmic Substance. В ігровий проєкт Unity можна імпортувати об'єкти цих програм та виконувати налаштовування за допомогою графічного інтерфейсу.

Для написання шейдерів використовується ShaderLab, що підтримує шейдерні програми написані на GLSL або Cg. Шейдер може включати декілька варіантів

реалізації, що дозволяє Unity визначати найкращий варіант для конкретної відеокарти. Unity також має вбудовану підтримку фізичного рушія Nvidia PhysX (колишнього Ageia), підтримку симуляції одягу в системі реального часу на довільній та прив'язаній полігональній сітці (починаючи з Unity 3.0), підтримку системи ray casts та шарів зіткнення.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАННЯ ДЛЯ СТВОРЕННЯ КРОСПЛАТФОРМНОГО 2D-ДАДАТКУ ДЛЯ ІГРОВОГО ЗАСТОСУВАННЯ НА UNITY.

1.1. Актуальніть досліждення.

Unity – найпопулярніший у світі ігровий движок. Він поєднує у собі величезну кількість різноманітних функцій і є досить гнучким, щоб зробити майже будь-яку гру, яку можна тільки собі уявити.

Завдяки неперевершеним крос-платформним функціям Unity користується популярністю як серед розробників, для яких розробка є хобі, так і серед AAA студій. Він використовувався для створення таких ігор як Pokemon Go, Heartstone, Rimworld, Cuphead, Genshin Impact.

Unity містить інструменти як для 3D, так і для 2D – розробки ігор. З ним дуже зручно працювати завдяки API, та скриптам на C#, також вбудовані інтеграції Visual Studio. Unity також пропонує JavaScript, як мову сценаріїв та MonoDevelop, як IDE для тих, хто хоче альтернативу Visual Studio. Ігровий рушій постачається потужними анімаційними інструментами, які спрощують створення власних 3D – роликів або створення 2D – анімацій з нуля. У Unity можна анімувати майже все. Розробниками пропонується безкоштовна версія, щоб можна було випускати ігри створені за допомогою Unity Personal, не платячи за програмне забезпечення, якщо вони заробляють менше 100 000$ на іграх створених за допомогою цієї платформи.

Оскільки Unity існує с 2005 року, він має величезну кількість користувачів та величезну бібліотеку ресурсів. Unity надає робочий простір, що поєднує зручні для

художника інструменти та дизайн, керований компонентами, що робить розробку ігри досить зрозумілою інтуїтивно.

Фізика 2D обробляється популярним двигуном Box2D. Unity використовує компонентний підхід до розробки ігор, що обертається навколо префабів (prefabs). За допомогою цих префабів ігрові дизайнери можуть будувати об’єкти та середовища ефективніше та масштабуватись швидше. Завдяки потужним шейдерам, фізико-заснованим матеріалам, системами освітлення з високою роздільною здатністю, Unity може надати вражаючу графіку.

Усі, від Ubisoft до NASA використовують технологію Unity VR. Сама платформа була побудована на C++ і оптимізується протягом багатьох років для підвищення продуктивності. Користувачі преміум-класу матимуть доступ до вихідного коду (source code) Unity для ще більших можливостей.

Міжплатформне разгортання – це одна з найпривабливіших особливостей для сучасних розробників. Завдяки підтримці всіх основних консолей та операційних систем, ігри, розроблені на Unity, - можна розгорнути на абсолютно будь-якій платформі. За допомогою інструментів Unity, можна одночасно обробляти дані для мишей, клавіатур та ігрових контролерів.

Існує також досить потужна підтримка хмарних рішень для багатокористувацьких ігор із хостингом на сервері та масштабованим встановленням зв’язків, що робить його універсальним рішенням для мультиплікаційного досвіду.

Співпраця команд значно покращилась у останніх версіях Unity. Вбудований контроль версій та хмарна інтеграція полегшують роботу з іншими, як ніколи раніше.

Unity має настроюваний редактор з повною підтримкою API для створення власних інструментів (можливість зробити будь-який інструмент за допомогою Unity, який ви хотіли б мати в Unity) та скриптів редактора (напр. Visual Studio). Варто згадати про магазин активів (Assets):

- *Unity Asset Store* – це зростаюча бібліотека активів. Unity Technologies і члени спільноти створюють ці активи та публікують їх у магазині. У

магазині є майже все: починаючи від різних типів об’єктів, видів, текстур, анімацій та моделей до цілих прикладів проектів, навчальних посібників та розширень редактора. Існує поєднання безкоштовних комерційних активів, які ви можете завантажити безпосередньо у свій проект Unity. Також можна створити свої власні активи та продати їх у цьому магазині Unity.

Для деяких людей ігри – це сенс життя, для деяких спосіб убити час, а для деяких спосіб заробити. Unity – це по перше ігровий движок, тому розглянемо найпопулярніші види 2D ігор:

- *Single Screen* – цей тип 2D-ігор був одним із оригінальніших типів додатків для ігрового застосування. Дійсно легко дізнатись, чи граєте ви у Single Screen гру, тому що кожен рівень, а іноді і вся гра проходить на одному екрані. Інша особливість, яка зазвичай загальна для всіх ігор даного типу, хоч з прогресом проходження рівнів гра і становиться важче рівні зазвичай такі ж як і раніше. Це означає, що можуть з’явитися нові вороги або скарби, однак по більшій частині, рівні майже не змінюються. Деякі популярні Single Screen ігри включають Dig Dug, Frogger and Pac-Man. Ці екрани зовсім не рухаються під час гри, і тільки персонаж, що взаємодіє з грою буде рухатися.
- *Side Scroller* – Ці ігри трохи вдосконаленіші ніж single-screen ігри, тому що гра рухається із сторони в сторону: справа – наліво, та навпаки. Як правило персонаж стоїть на місці поки фон рухається навколо персонажа. Як правило гру розглядають збоку (це також часто використовується у іграх жанру «платформер»). У цьому жанрі герой як правило бігає, лазить та стрибає через різні рівні. Є кілька відеоігор з бічним скролером, які дозволяють гравцям відступити і відвідати частину рівня, який вони вже оббігали. Інші продовжують рухатися, а це означає, що, коли гравець пройшов повне місце розташування, - цю область на рівні не можна відвідувати знову. Є ще інші відеоигри з бічним скролером, де фон

рухається і гравець повинен просто намагатися не відставати, стрибаючи, щоб подолати перешкоди та зібрати предмети, перш ніж вони пройдуть повз екран.

- *Scrollers* – Перші відеоігри покладалися на технологію «скролінгу» для того, щоб виглядало, наче гра «рухається». Такі ігри мали персонажа або об'єкт, який був встановлений на місці, але який, здавалося, рухався завдяки руху фонової прокрутки. Один із найбільш ранніх прикладів цього типу гри – гоночна гра. Транспортний засіб стояв нерухомо на дорозі, але дорога рухалася. Зазвичай, такі ігри скролились вертикально, ніж горизонтально. Гравець повинен був швидко реагувати на різні змінення, що відбувалися на фоні, щоб вони не врізались у що-небудь, та могли отримувати різні бонуси, що з'являлися на фоні. Після випуску Scroller and Side-Scroller ігор розробники відеоігор швидко почали створювати ігри які могли прокручуватись як вертикально, так і горизонтально.
- *Adventure* – ігри, які дуже швидко набрали популярність, тому що дозволяли гравцю досліджувати своє оточення та віртуальний ландшафт, що було нечуваним раніше. Ці ігри, як правило мали один екран, і персонаж гравця тоді міг вибрати рух в будь-якому з чотирьох напрямків. Гравці не просто досліджували і боролися з ворогами; їм часто доводилося брати участь у вирішенні проблем, щоб прокласти собі шлях через різні рівні або мати можливість знайти обладнання та предмети, необхідні їм для прогресу та виживання. Ці ігри в основному керуються історією і часто ведуть багато діалогу, який бере участь у грі. Вони були однією з перших ігор, що використовували діалог і представляли його як дерево розмов, де гравці можуть взаємодіяти з неігровими персонажами, вибираючи із заздалегідь записаного діалогу, а потім чекаючи відповіді від неігрового персонажа. Оскільки гравці можуть використовувати необмежену кількість варіантів діалогу, це створює дуже відкритий світ з великою кількістю можливостей.

Кожна 2D – пригодницька гра має свій власний квест, який гравці повинні спробувати виконати, але вони можуть робити це лише по одному екрану під час руху вгору, вниз, вліво або вправо.

- *Second-Generation Side Scrollers* – після створення 16-бітних домашніх консолей, - ігри були дуже популярні у більшості домів та ігрові розробники хотіли створити ігри, які не тільки легко впізнати, але і продемонструвати вдосконалені можливості та геймплей цих 2D-ігор. Їжачок Сонік була однією с перших випущених ігор с бічним скролом другого покоління. Він не тільки подорожував традиційно: із сторони в сторону, з чим було знайомо багато гравців, але він зміг прокручуватись у всіх напрямках, по кривих пагорбах і пересіченій місцевості, і навіть пересуватися за допомогою роликів і стрибків. Цей прогрес призвів до напливу інших ігор і відкрив шлях до ігор з плавною графікою, яка не здригалася під час гри.
- *Platform Games* – нарешті, 2D – ігри перейшли від того, що гравці просто повинні пересувати своїх персонажів вліво, або вправо, і дозволяли їм стрибати вгору та вниз, на платформах. Це особливий тип ігор з бічним скролером або одиночним екраном, але вони мають набагато більше рухів і забезпечують гравцям значно більший контроль над своїми персонажами. У цих іграх є платформи, до яких гравці можуть досягти, стрибаючи вгору, або падаючи вниз, але вони повинні це робити, намагаючись уникати перешкод та ворогів, що ускладнює таких тип гри. Платформи, як правило не мають однакової висоти і роблять дуже нерівну місцевість, яку деяким гравцям може бути важко пройти. Гравці мають контроль не тільки над напрямком, в якому їх персонажі рухатимуться та стрибатимуть, а також відстанню та висотою стрибків, що допомогає їм спробувати сісти, на платформу, замість того щоб впасти, можливо, до смерті. На додаток до того, що вони вимагають від гравців переходу с однієї платформи на іншу, ці ігри часто містять інші елементи, які роблять їх набагато складнішими, ніж інші типи

2D-ігор. Ці ігри можуть іноді вимагати від гравців контролю персонажів, щоб змусити їх відхилятися від таких предметів, як мотузки або лози, і навіть відбиватися від батутів чи трамплінів. Ігри, де гравці не мають контролю над своїми персонажами і де стрибки автоматизовані грою, не потрапляють до цієї категорії 2D-ігор платформерів. Дуже поєднуються з іншими жанрами та типами ігор, оскільки це додає елементу веселощів і більше захоплення в цілому. Це означає, що ці ігри часто поєднуються з іграми шутерів або пригод. Раніше ці ігри були найпопулярнішим видом відеоігор.

1.2. Постановка завдань.

Щоб процес розробки над проектом став простішим, потрібне розуміння декількох важливих кроків.

Першим кроком, потрібно обрати жанр гри, яку я буду створювати. Вибір стояв перед наступними жанрами 2D-ігор:

- *Арканоід* – суть такої гри полягає у тому, що потрібно відбивати кульку, як у пінг-понгу керуючи «ракеткою» від якої він відбивається. Мета гри – розбити всі блоки, які будуть на рівні.
- *TDS(Top Down Shooter)* – як зрозуміло із назви, - це шутер з видом зверху. Зазвичай – стрілянина за допомогою миші. Іноді його малюють в ізометрії для більшої краси.
- *Скроллер* – суть гри полягає у тому, що екран постійно у русі, вимушуючи нашого персонажа рухатися за ним, однак у більшості випадків, - гравець сам рухається за екраном. Зазвичай такий жанр також називають скролл-шутерами, оскільки майже завжди ціль гри стріляти у ворогів.

- *Платформер* – зазвичай, вид збоку. Гравцю доведеться стрибати на платформи, знищувати супротивників. Жанр дуже різноманітний.

Через певний ряд причин, мною був обраний жанр «Платформер». Я багато часу витратив на ігри схожого жанру, тож знаю, с чим мені доведеться мати справу, що робить розробку більш швидкою та продуктивною. Така гра набагато цікавіша, ніж інші 2D-ігри. Цей жанр розкриває всю красу 2D – ігор, завдяки тому, що у гравця є конкретні завдання, він знає, що йому потрібно робити, але у той же час у нього доволі широка свобода дій. Щоб пройти рівень йому потрібно подолати певні труднощі. Завдяки тому, що Unity надає можливість зробити гру кросплатформенною, в неї можна буде грати як с ПК, так і з мобільного девайсу, що дозволяє весело проводити час майже у будь-якому місці.

Наступним кроком, потрібно визначити ключові характеристики гри:

- *Персонаж.* Головний герой гри, який у ході дослідження світу стикатиметься з різними труднощами та ворогами. Треба визначитись з його характеристиками, наскільки він буде сильним, порівняно с ворогами. Визначитись с стилем бою гравця.
- *Карта.* Це буде гра, у декілька рівнів. Ціль кожного рівня - дійти до фінішу та натиснути кнопку, щоб завершити рівень. Треба грамотно розташувати платформи, ворогів та інші завади гравцеві.
- *Монстри.* Уявити платформер без ворогів доволі таки важко. У моєму випадку, це будуть монстри, які контролюють локацію. Вони будуть «патрулювати» місцевість, якщо персонаж попаде у їх зону зору, то монстр повинен почати переслідувати гравця. За певних умов, він повертається до патрулю на свою початкову точку.
- *Ігрова система.* Наостанок потрібно визначитися з ігровою системою – набором правил, за яким у грі будуть виникати певні події, а саме:
  - ключові ігрові алгоритми;

- управління;
- ігрове меню;
- ігрова пауза;
- рух камери;
- фонова музика;
- анімації;
- планування рівня ігри;
- зміна ключових сцен ігри (перехід між рівнями);
- кількість здоров’я у гравця та його ворогів;
- кількість шкоди яку наноситимуть монстри при атаці по персонажу та навпаки;
- поведінка монстрів та її зміна за певних умов;
- умови завершення гри;
- фінал;
- мобільне управління.

1.3. Аналіз методів розв’язання поставленої задачі.

Так як розробка гри буде проходити на Unity, то доволі логічно буде використовувати мову програмування C#, - для описання скриптів, для руху персонажу.

Найзручнішим та звичним буде управління персонажем за домогою клавіш WASD. Додатково було вирішено додати клавішу «Space», яка буде виконувати ту же функцію, що і клавіша «W», - а саме стрибок угору. Атака буде виконуватись за допомогою лівої кнопки мишки.

Також буде добавлено управління для мобільних девайсів за допомогою віртуальною клавіатури на екрані смартфону.

Для конструювання візуальної частини карти використані спрайти із Assets Store. За допомогою інструментів Unity, створені колайдери, аналогічно візуальній частині,

по яких зможе рухатися гравець. Використовуючи інструмент Tilemap у Tile Palette намальована інша частина карти, це можна назвати «малювання за допомогою спрайтів».

Для візуальної складової добавлені різноманітні елементи, такі як дома, гори, небо тощо. У ролі монстрів використані інші моделі юнітів із Assets Store. Керувати ними буде штучний інтелект, а точніше певний алгоритм, за яким вони себе поводять.

У грі декілька типів монстрів:

- *рухомі*, що «патрулюють» місцевість, які починають переслідувати гравця, який потрапив у певну невидиму область навколо них.
- *нерухомі*, що стоятимуть, доки до них не підійде гравець.

Монстри наносять шкоду персонажу, у той час, коли знаходяться у безпосередній близькості до нього.

Ціль гри: знайти важіль, активувати його, дійти до фінішу та завершити рівень. Через виклик ігрового меню відбудеться перехід на наступний рівень. Взаємодія з такими об’єктами як фініш та важіль, - відбуватиметься за допомогою клавіши «F». На кожному рівні знаходиться декілька ворогів, які намагатимуться ліквідувати персонажа гравця.

Камера повинна постійно та плавно рухатися за персонажем. У гру додана фонова музика, яка відрізняється у залежності від того, що відбувається на екрані ігрока.

1.4. Порівняльна характеристика інших ігрових рушіїв із Unity.

Чому саме Unity, а не будь-які інші ігрові движки, такі як Unreal Engine, GameMaker або Godot?

Для початку, Unity – це хороший, універсальний двигун, що може задовільнити майже будь-які потреби. Він чудово підходить для ігор-прототипів.

Система префабів полегшує повторне використання коду та ресурсів інших проектів, та їх редагування для нових цілей.

Побудова складних світів в Unity стає питанням складання багатьох компонентів, що складаються з їх власних компонентів. Головною причиною Unity – є величезна бібліотека ресурсів доступних кожному. Навіть досвідченні розробники можуть заощадити час і багато чому навчитись у спільноти.

Також Unity пропонує надійний набір хмарних інструментів, які дозволяють легко монетизувати свій додаток та додавати багатокористувацькі можливості. Завдяки Unity Analytics, Unity Ads, Unity Collaborate та Unity Multiplayer користувачі мають доступ до неймовірно сумісного набору інструментів для створення динамічних ігор – все в одному місці. Дуже мало інших ігрових двигунів пропонують таку централізацію.

З мінусів: багатоцільовий підхід Unity робить його незграбнішим, ніж двигуни з більшим фокусом. Створення 2D – ігор в Unity є більш болісним, ніж у Godot чи GameMaker, тоді як системи освітлення та рендерінгу Unreal Engine є більш здібними, ніж в Unity.

Unity блокує функції тих, хто не хоче (або не може) придбати преміум версію. Більшість не матиме доступу до вихідного коду (source code), що може зробити його схожим на чорну скриньку.

Система ліцензування Unity може заплутати. Наприклад: програма безкоштовна, якщо ви не заробляєте більше 100 000$ зі своїх продажів, і в цьому випадку вам потрібно буде придбати план підписки, який також може заплутати, щоб зрозуміти багаторівневі ціни, які вони вимагають.

Інші двигуни вимагають більш прямі угоди і можуть бути простішими, якщо гроші викликають серйозне занепокоєння.

Висновки до розділу

У результаті виконання цього розділу було проаналізовано актуальність даної теми, виявлено, що ігрова індустрія активно розвивається та стає більш глобальною. Проаналізована актуальність Unity для розробки 2D-ігор, встановлено, що є досі актуальними. Враховуючи безпосередньо кросплатформні додатки, з впевненістю можна заявити, що є одною з найперспективніших гілок розробки на даний момент і найближче майбутнє.

Найбільші переваги розробки на Unity:

- Єдина система активів(assets);
- Вбудований редактор рівнів;
- Відмінна підтримка налаштування та налагодження проекту;
- Обширна бібліотека готових ресурсів;
- Mono у якості хоста скриптів. Mono надає велику кількість функцій:
    - колекції;
    - input\output(можна одразу бачити дані вхідні та вихідні дані);
    - багатопоточність;
    - LINQ – використання якого значно прискорює розробку.

РОЗДІЛ 2. ПРОЕКТУВАННЯ 2D-ДОДАТКУ НА UNITY.

2.1. Архітектура додатку

Добра архітектура це насамперед вигідна архітектура, що робить процес розробки та супроводу програми більш простим та ефективнішим. Програму з доброю архітектурою легше розширити та редагувати, а також тестувати, налагоджувати та розуміти. Це означає, що можна сформувати список доволі доречних, розумних та універсальних критеріїв.

2.1.1. Ефективність системи

В першу чергу програма, звичайно ж, повинна вирішувати поставлені задачі та добре виконувати свої функції, причому у різних умовах. Сюди можна віднести такі характеристики, як надійність, безпечність, продуктивність, здатність справлятися с підвищенням навантаження (масштабування).

У випадку в представленим додатком, а саме кросплатформним 2D-платфомером на Unity, повинні дотримуватися такі умови:

- гра повинна бути оптимізована та не повинна сильно навантажувати систему;
- програма повинна добре працювати на всіх девайсах;
- додаток повинен бути однаково зручним в управлінні як на PC, так і на мобільному пристрої;
- вона повинна бути безпечна для користувача;
- гра повинна бути не дуже складка, але й не дуже легка, щоб користувачу було цікаво в неї грати;

- повинна мати гарний вигляд, щоб у неї було приємно грати;
- не повинна мати помилок у програмі або дефектів у відлові помилок;
- інтерфейс повинен бути інтуїтивно зрозумілим.

2.1.2. Гнучкість системи

Будь-який додаток доводиться змінювати з часом – змінюються вимоги, додаються нові. Чим швидше та зручніше можна внести зміни в існуючий функціонал, чим менше проблем і помилок це викликає – тим гнучкіше і конкурентоздатна система. Тому у процесі розробки потрібно намагатися оцінювати те, що виходить, на предмет того, як вам це у майбутньому, можливо, доведеться змінювати. Змінення одного елемента системи не повинно впливати на її інші фрагменти. По можливості, архітектурні рішення не повинні «вирубуватися на каменю», і наслідки архітектурних помилок повинні бути у розумному ступені обмежені. «Добра архітектура дозволяє ВІДКЛАДАТИ прийняття ключових рішень» (Боб Мартін) ті мінімізує «ціну» помилок.

Також систему бажано проектувати так, щоб її фрагменти можна було повторно використовувати в інших системах. Важливою частиною є здатність скоротити час за рахунок додавання до проекту нових людей. Архітектура повинна дозволяти розділити процес розробки, так щоб багато людей могли працювати над програмою одночасно.

2.1.3. Розширюваність системи

Можливість додавати у систему нові сутності і функції, не порушуючи її основної структури. На начальному етапі у систему має сенс закладати лише основний та найнеобхідніший функціонал. Але при цьому архітектура повинна дозволяти легко нарощувати додатковий функціонал за мірою необхідності. Причому так, щоб внесення найбільш можливих змінень потребувало найменших зусиль.

Потреба, щоб архітектура системи володіла гнучкістю та розширюваністю (тобто була здатна до змін та еволюції) є настільки важливим, що це навіть сформовано у виді принципу – «Принцип відкритості\закритості» (Open-Closed Principle – другий з п’яти принципів SOLID): програмні сутності (класи, модулі, функції тощо) повинні бути відкритими для розширення, але закритими для модифікації.
Іншими словами: повинна бути можливість розширити або змінити поведінку системи без змінення\переписування вже існуючих частин системи.
Це означає, що додаток слід проектувати так, щоб змінення його поведінки і додавання нового функціоналу досягалося б за рахунок написання нового коду (розширення), і при цьому не доводилося б змінювати вже існуючий код. У такому випадку поява нових вимог не спричинить модифікації вже існуючої логіки, а може бути реалізовано насамперед за рахунок її розширення. Саме цей принцип є основою «плагінної архітектури» (Plugin Architecture).

2.1.4. Можливість тестування

Код, який простіше тестувати, буде мати менше помилок та надійніше працювати. Але тести не тільки поліпшують якість коду. Багато розробників приходять до висновку, що потреба «хорошої тестованості» є також направляючою силою, автоматично веде до доброго дизайну, та водночас є одним із найголовніших критеріїв, які дозволяють оцінити його якість.
Існує ціла методологія розробки програм на основі тестів, яка так і називається – «Розробка через тестування (Test-Driven Development, TDD)».

2.1.5. Добре структурований, зрозумілий та читаємий код. Супроводження.

Над програмою, як правило, працює багато людей – одні уходять, інші приходять. Після написання супроводжувати програму, як правило, доводиться іншим

людям, що не приймали участь у її розробці. Тому добра архітектура повинна надавати можливість відносно легко та швидко розбиратись у системі новим людям. Проект повинен бути добре структурованим, не мати дублювання, мати добре оформлений код та бажано документацію. По можливості у системі краще використовувати стандартні, загальноприйняті рішення, звичні для програмістів. Чим більш екзотична система, тим важче її зрозуміти іншим людям (Принцип найменшого здивування – Principle of least astonishment. Зазвичай, він використовується у відношенні користувацького інтерфейсу, але також застосовується до написання коду).

2.2. Діаграма станів

У результаті проектування була розроблена UML-діаграма станів ігрового додатку (рис. 2.1).

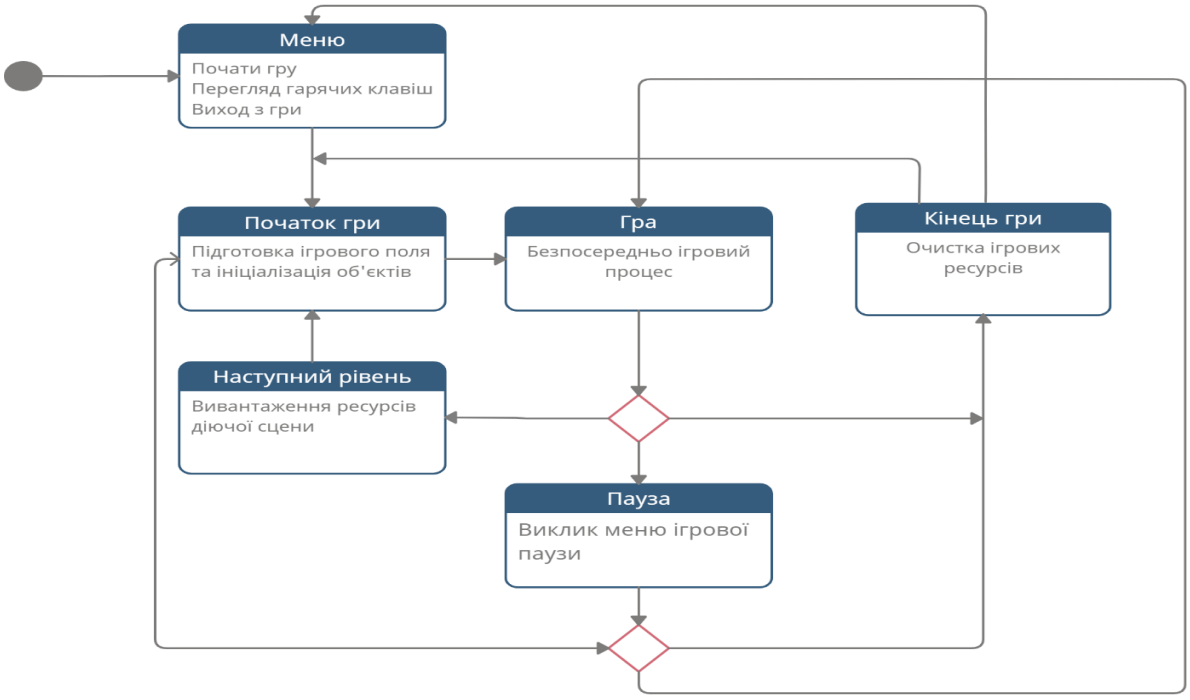

Рис. 2.1. Діаграма станів ігрового додатку

Були визначені наступні стани гри:

- Стан «*Меню*» - перший стан ігрового додатку. При переході у нього відкривається головне меню. У якому можна подивитися гарячі клавіши

(*Hotkeys*), можна вийти із ігри та почати гру. Після команди користувача відбувається перехід у стан «*Початок гри*».

- Стан «*Початок гри*» - при переході у цей стан відбувається ініціалізація необхідних для гри об’єктів. Перехід у наступний стан відбувається автоматично після ініціалізації усіх потрібних ресурсів.
- Стан «*Гра*» - безпосередньо сама гра. Коли ігровий додаток знаходиться у цьому стані, гравцеві доступно управління персонажем на ігровому полі та виклик додаткового меню «*Пауза*». Якщо гравець програв викликається меню «*Кінець* гри», якщо гравець завершив рівень, викликається меню «Перехід на наступний рівень».
- Стан «*Наступний рівень*» - при переході у цей стан ігрові ресурси діючої сцени вивантажуються, після цього відбувається перехід у стан «Початок гри» у рамках іншої сцени.
- Стан «*Пауза*» - ігрова пауза, яка може викликатись за допомогою кнопки «*Escape*» (недоступно для мобільних девайсів), або натисканням на відведений для цього елемент ігрового інтерфейсу. Знаходячись у цьому стані ігровий процес призупиняється, а гравцеві доступно почати рівень заново, продовжити гру або вийти з неї.
- Стан «*Кінець гри*» - при переході у цей стан гравець може почати гру заново або вийти з неї. Якщо він починає гру заново - відбувається перехід у стан «*Початок гри*», якщо він завершує гру, то гра переходить у стан «*Меню*».

2.3. Прототипи ігрових інтерфейсів

Етапу створення прототипів не марно приділяється стільки часу під час створення додатків. Прототипи здатні допомогти у різних ситуаціях і на різних етапах створення продукту.

Прототип – це модель, прообраз кінцевого продукту. Прототипи відрізняються за ступенем точності та приближення до реального додатку. Крім цього, різні види прототипів слугують різним цілям та здатні вирішувати різні задачі. За стадією готовності їх можна поділити на 3 етапи:

- Концептуальні
- Інтерактивні
- Анімовані

Прототипи надають відмінну можливість не тільки для залучення користувачів у процес дизайну, але і найбільш швидкого створення продукту, відповідного очікуванням клієнта. Прототипи допомагають при спілкуванні з замовником, можуть бути наглядною картиною для розробників та здатні презентувати ідею компанії.

2.3.1. Концептуальні прототипи

Концептуальний прототип представляє собою схематичне зображення майбутніх екранів та створюється на ранніх етапах розробки продукту.

Концептуальний прототип потрібно робити завжди, при створенні інтерфейсу нового додатку. Такий спосіб допоможе на ранніх етапах вирішити більшість питань використання.

Концептуальне проектування відмінно підходить для найбільш швидкого тестування ідей, тому що дозволяє створити основні елементи екранів за лічені хвилини. Крім того, для створення такого прототипу не потрібно володіти навичками праці с спеціальними інструментами, достатньо скористуватись підручними засобами – блокнотом, дошкою, або навіть стікерами.

Концептуальні прототипи невід’ємні, коли потрібно перенести користувацькі сценарії на екрани майбутнього додатку. Таким чином, ваш додаток у перший раз становиться прообразом кінцевого результату. Перевагами концептуальних прототипів

є можливість командної роботи. Дуже часто відбувається, що при візуалізації того чи іншого функціоналу потрібна підтримка окремих спеціалістів. Концептуальний прототип може представляти собою результати міркування групи людей, що є дуже ефективним способом рішення проблем використання та пошуку рівноваги між цілями бізнесу і цілями користувачів. Прототип допомагає говорити з будь-яким спеціалістом «одною мовою».

2.3.2. Інтерактивні прототипи

Інтерактивний прототип, як правило складається з екранів, що пройшли стадію концептуальних прототипів. Прототип становиться доволі реалістичним, щоб тестувати його на кінцевих користувачах.

Коли потрібно робити інтерактивні прототипи:

- Коли потрібно змоделювати який-небудь користувацький сценарій (наприклад реєстрацію у додатку).
- Коли необхідно протестувати частину сценарію на користувачах, а готового додатку ще немає. У цьому випадку інтерактивний прототип є чудовим способом вирішити задачу у короткий проміжок часу.
- Щоб показати команді, на якому етапі ви знаходитесь. Пояснити розробникам логіку роботи інтерфейсу.
- Коли потрібно коротко та наглядно показати керівництву над чим ви працюєте. Такий спосіб може добре здивувати менеджерів, що буде плюсом вам та команді.
- Якщо потрібно справити враження на потенційного інвестора. Або проштовхнути якусь ідею додатку у своїй компанії. Тут працює правило «Краще один раз побачити, ніж сто раз почути». Ні один розказ і ні одна презентація не замінять прототип готового додатку.

2.3.3. Анімовані прототипи

Рухання – це спосіб надихнути життя у дизайн. Анімований прототип є одним с найбільш високорівневих прототипів. Деякі з них здатні практично повністю моделювати роботу справжнього додатку, людині, яка у цьому не розбирається майже неможливо їх відрізнити. Головна перевага створення анімованого прототипу це, як не дивно, його анімованність. На цьому етапі дизайнер продумує дуже важний аспект в UX – взаємодія додатку с користувачем, візуалізація якого створюється за допомогою анімації.

Анімація є способом комунікації додатку з користувачем. Вона дозволяє користувачу залишатися у курсі усіх подій, що відбуваються у додатку, та в рази полегшує використання інтерфейсу. Коли рух елементів у додатку моделює природні фізичні процеси, вони зчитуються мозком на підсвідомому рівні, і користувач не вдумуючись, розуміє що відбувається. Таким чином, рух робить дизайн більш орієнтованим на користувача.

Ми очікуємо, що елементи інтерфейсу відгукнуться на наші дії, на засмучуємося, коли цього не відбувається. Нам необхідно знати, що наші дії не є марними. Уявіть, що ви кажете щось людині, а вона аж ніяк не реагує на ваші слова. Це може вивести із себе. Одного разу тестувався додаток-помічник продавця-консультанта для магазинів крупної торговельної мережі. Додаток був у достатньо сирому стані, але готовий до тестування певного функціоналу. Анімація елементів на цьому етапі була не продумана. У продавців викликало певно роздратування те, що кнопка «Покласти у корзину» візуально не відкликалася на натиск. Не дивлячись на те, що вона виконувала свій основний функціонал – товар потрапляє у корзину – користувач не розуміє, виконалась цільова дія, чи ні.

Для створення анімованого прототипу необхідне знання деяких спеціальних інструментів, працюючих з анімацією. Більшість таких програм підтримує пошаровий імпорт із Sketch, що дозволяє працювати не з екраном цілком, а з окремими елементами. Прототип можна відкрити на мобільному додатку. Анімований прототип

займає набагато більше часу – на продумування деталей і на виконання, однак результат виправдовує себе.

Коли потрібно робити анімований прототип:

- Якщо ви хочете продумати анімацію керування додатком, крім тої, яку він вже має.
- Пояснити розробникам не тільки взаємозв’язки екранів, але й логіку роботи окремих елементів та відгук цих елементів на дії користувача. Це дуже важливо, тому що саме їм реалізовувати ці ідеї.
- Коли необхідне високорівневе тестування, цілі якого пов’язані з ступенем чуйності додатку.
- Анімований прототип підвисить ефект при подачі ідеї.

2.3.4. Концептуальний прототип для цільового додатку проекту

У результаті аналізу предметної області був розроблений інтерфейс користувача ігрового додатку. Він складається з декількох основних екранів – головного меню (рис. 2.2).

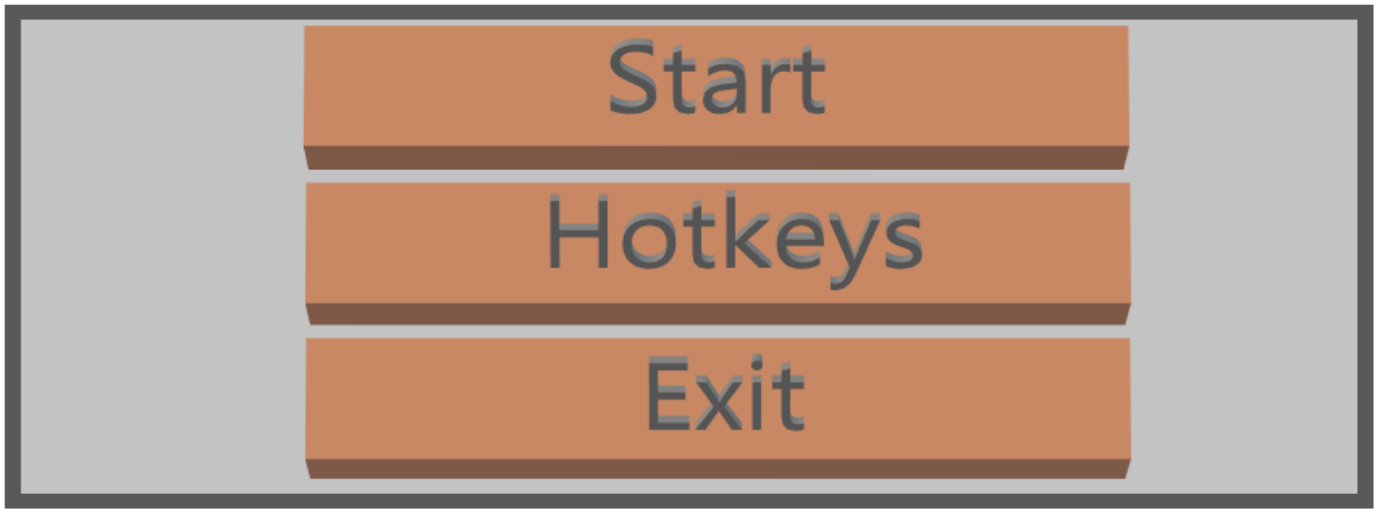

Рис. 2.2. Концептуальний прототип головного меню

В головному меню знаходяться клавіши початку гри, виходу із неї, та кнопка виклику додаткового інтерфейсу (Hotkeys), що дозволяє подивитись список гарячих клавіш (рис. 2.3).

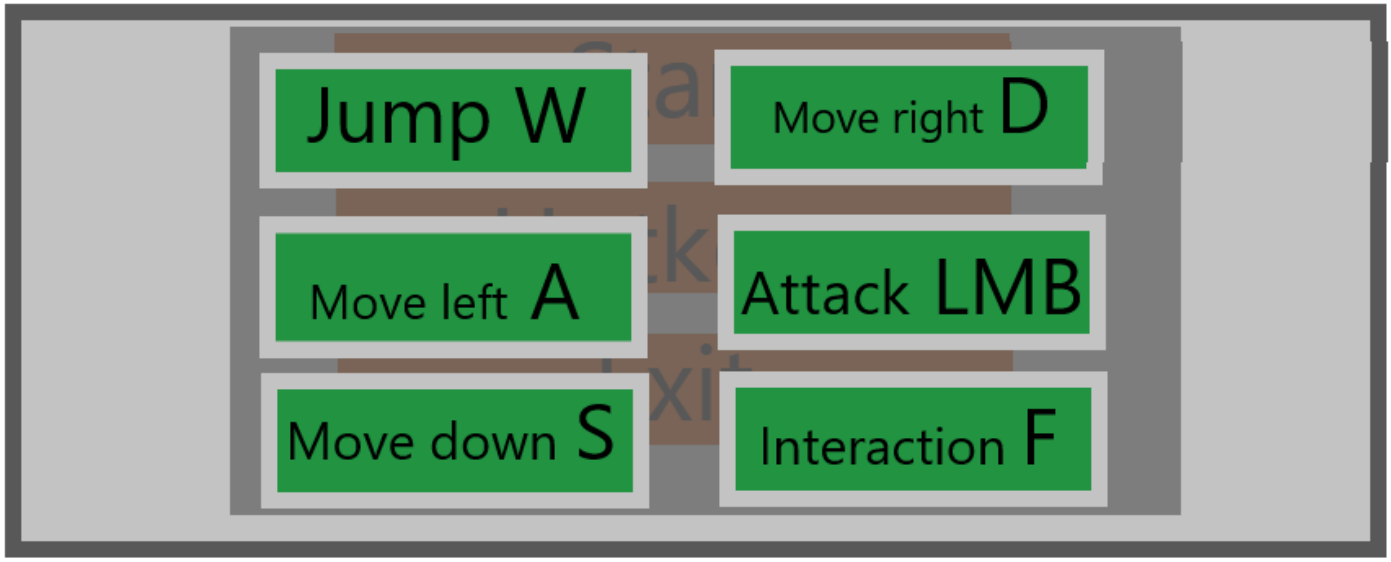

Рис. 2.3. Концептуальна модель інтерфейсу «Hotkeys» у головному меню

На ігровому екрані (рис. 2.4) розташована кнопка паузи, смуги здоров’я над ворогами (красного кольору) та над персонажем гравця (зеленим кольором) та всі головні об’єкти гри.

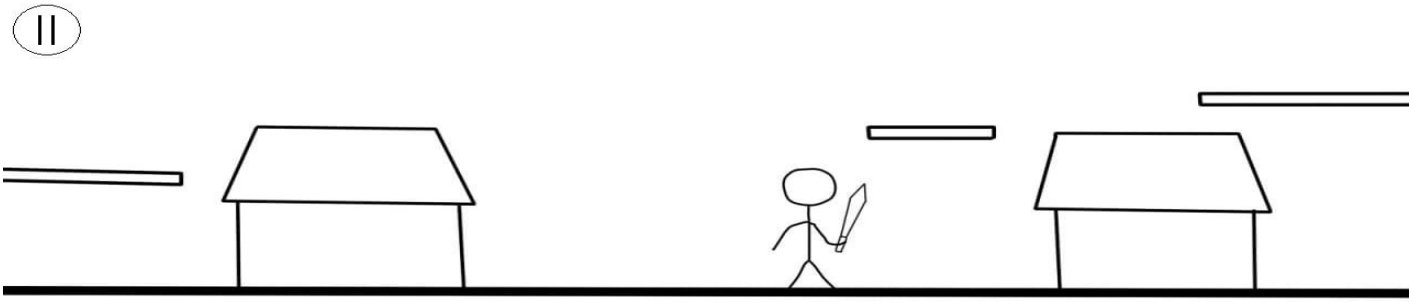

Рис. 2.4. Концептуальний прототип ігрового поля

Кнопка паузи викликає додатковий інтерфейс (рис. 2.5), який призупиняє гру, та надає гравцю можливості продовжити гру, почати заново, або вийти у головне меню.

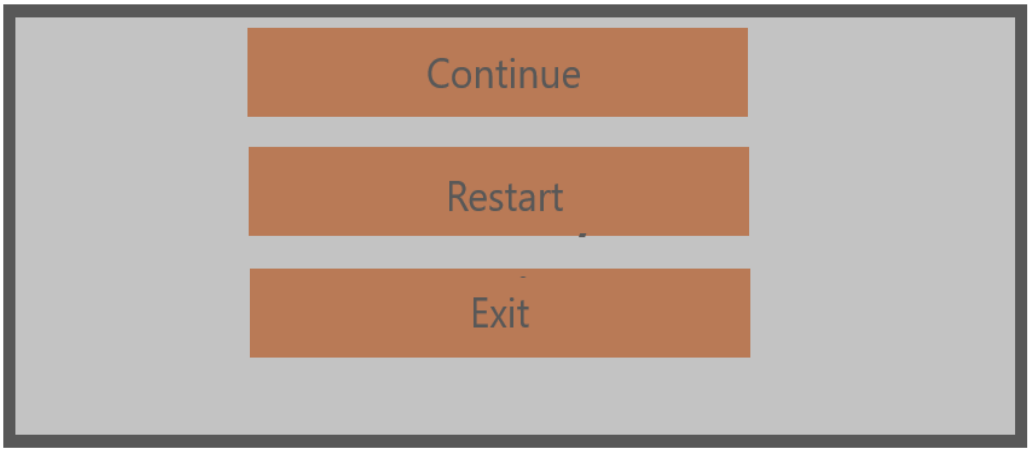

Рис. 2.5. Концептуальна модель меню паузи

Якщо гравця подолали вороги, та його персонаж загинув – викликається меню, що оповіщає про завершення гри (рис. 2.6).

Рис. 2.6. Концептуальний прототип екрану завершення гри

Наразі завершення рівня є відповідне меню, яке надає можливість перейти на наступний рівень (рис. 2.7).

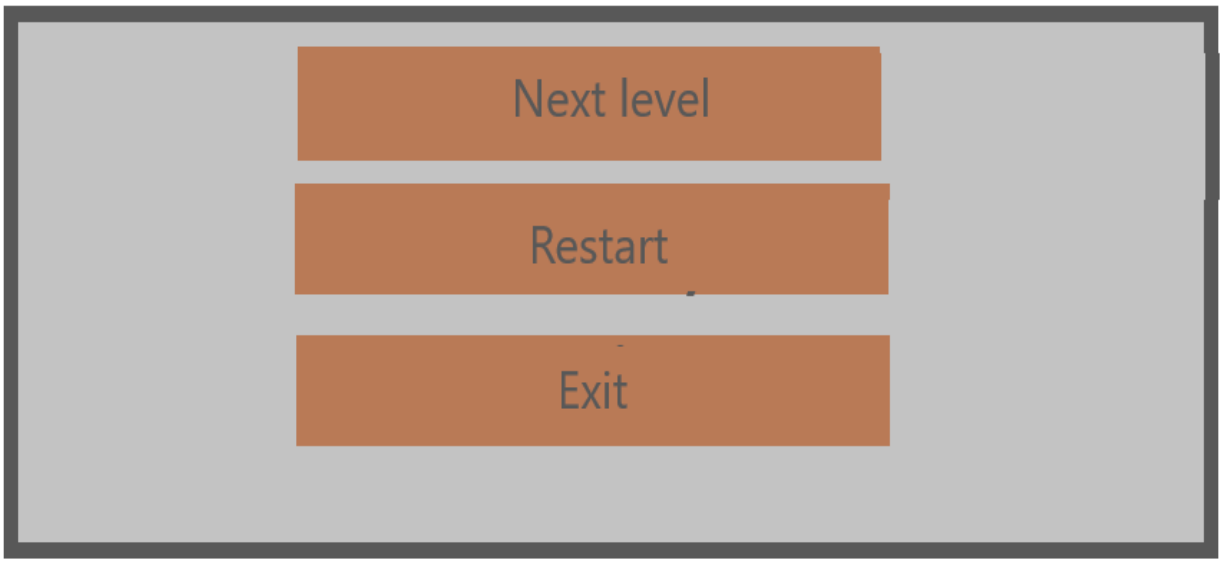

Рис. 2.7. Концептуальна модель переходу на наступний рівень

Додатково був створений концептуальний прототип ігрового поля для мобільних пристроїв. Були додані наступні елементи інтерфейсу користувача: кнопки стрибку, атаки та взаємодії із ігровими об’єктами, також був доданий зручний віртуальний джойстик для руху гравця (рис. 2.8).

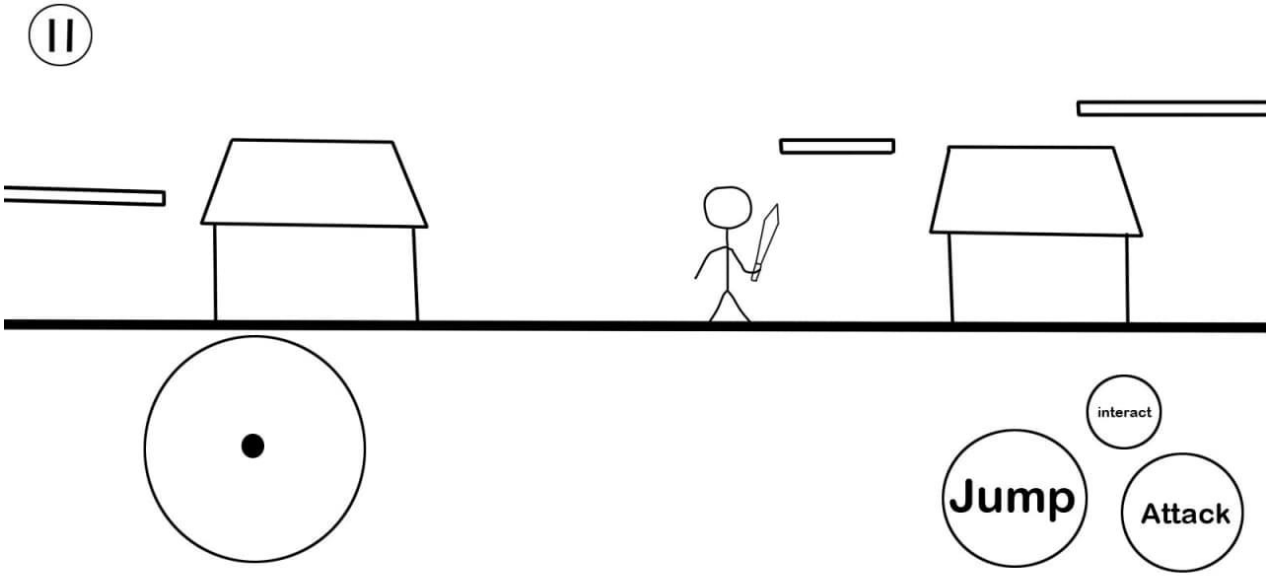

Рис. 2.8. Концептуальний прототип ігрового екрану для мобільних пристроїв

2.4. Діаграма класів

У будь-якому об’єктно-орієнтованому процесі проектування діаграма класів є результатом, тому що є моделлю, найбільш близькою до реалізації (коду). Існують інструменти, здатні конвертувати діаграму класів у код – такий процес називається «кодогенерація» та підтримується більшістю IDE та засобів проектування. Наприклад кодогенерацію виконує Visual Paradigm (доступно у види плагінів для більшості IDE), нові версії Microsoft Visual Studio, такі засоби проектування як StarUML, ArgoUML тощо. Щоб побудувати за діаграмою добрий код, вона повинна бути доволі докладною.

У результаті проектування була розроблена UML-діаграма класів ігрового додатку(рис. 2.9).

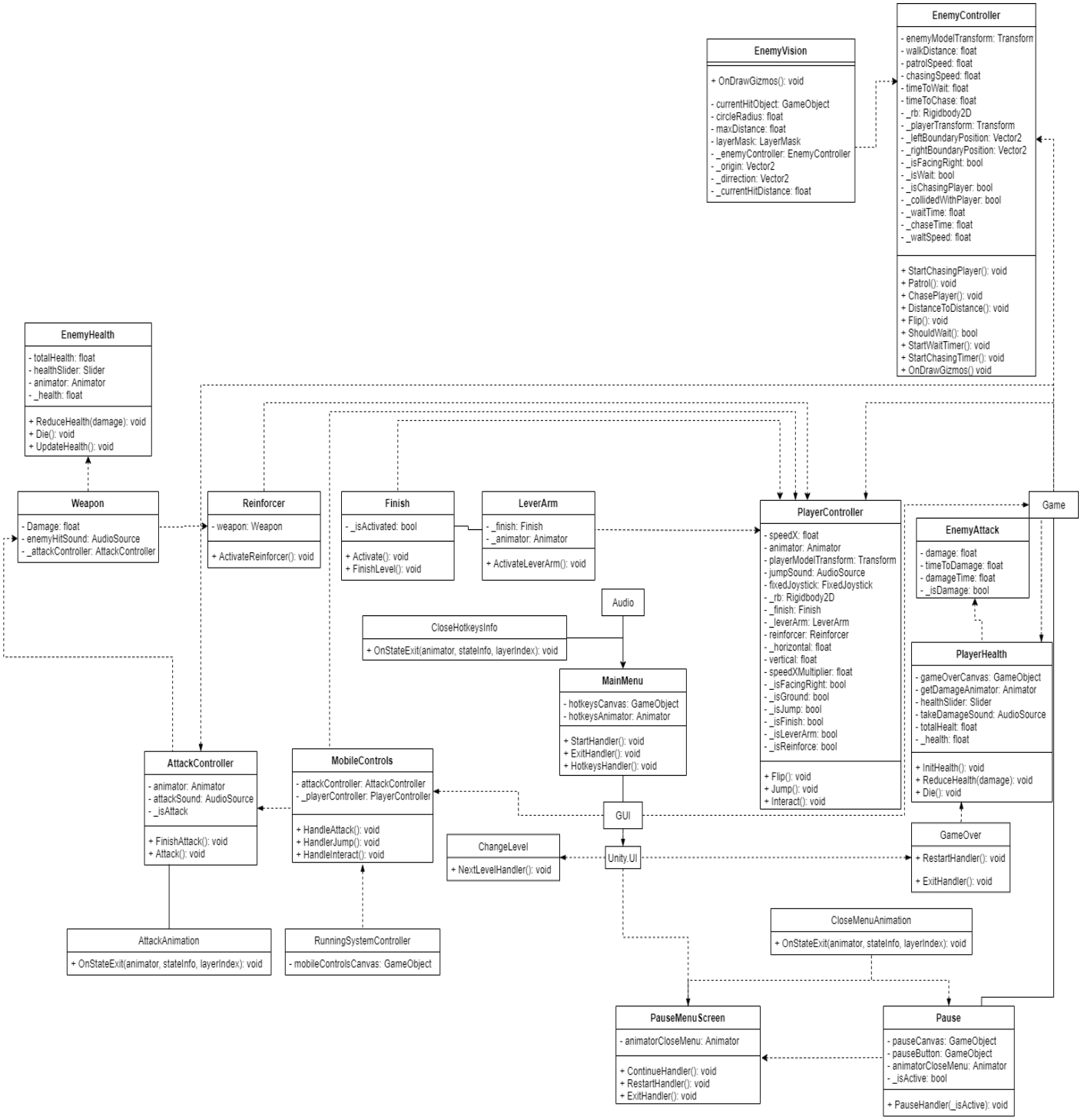

Рис. 2.9. Діаграма класів

Unity – це інструмент для розробки двовимірних і тривимірних додатків та ігор, працюючий під операційними системами Windows, Linux, OS X. Створені за

допомогою Unity додатки працюють під операційними системами Windows, Linux, OS X, Windows Phone, Universal Windows Platform, Apple IOS, Android, а також на ігрових приставках Wii, PlayStation 3, PlayStation 4, Xbox 360, Xbox One та MotionParallax3D.

Архітектура проекту Unity основана на шаблоні Entity-Component-System. Згідно цього шаблону, додаток складається з базових сутностей, функціональність яких поширюється за допомогою спеціалізованих компонентів.

Проект Unity складається з декількох сцен (Scene), на котрих розташовані ігрові об’єкти (GameObject) з прикріпленими до них компонентами (Component). У кожного ігрового об’єкта є обов’язковий компонент Transform, який відповідає за розташування об’єкту на сцені. Окрім цього можуть бути підключені як готові компоненти (наприклад Rigidbody2D, що відповідає за фізичну симуляцію), так і користувацькі компоненти.

Безпосередньо програмування в Unity полягає у насамперед розробці користувацьких класів, які підключаються до ігрових об’єктів, як компоненти. Всі такі класи повинні унаслідуватися від класу MonoBehaviour (включаючи унаслідуванні методи, такі як Start(), Update, FixedUpdate тощо). Вказання цього відношення значно перевантажило б діаграму, тому для позначення класів-компонентів до їх імен було додано суфікс «Script».

Були спроектовані наступні класи ігрового додатку.

- Класи «*Audio»* та «*AudioSource»* відповідають за відтворення музики та звукових ефектів ігри.
- Клас «*CloseHotkeyInfo*» відповідає за анімацію та закриття інтерфейсу користувача «*Hotkeys*», на якому він може подивитися гарячі клавіши.
- Клас «*MainMenu*» відповідає за розташування елементів у головному меню, їх працездатність та функціонал.
- Клас «*ChangeLevel*» відповідає за перехід на наступний рівень, після завершення поточного.

- Клас «*CloseMenuAnimation*» відповідає за анімацію та закриття головного меню.
- Клас «*PauseMenuScreen*» відповідає за набір функцій, які викликаються за допомогою елементів користувача (Button – елементи).
- Клас «*Pause*» реалізує логіку ігрової паузи, та відповідає за перехват клавіш (або кнопки інтерфейсу користувача), які визивають ігрову паузу.
- Клас «*GameOver*» відповідає за реалізацію інтерфейсу, який викликається якщо гравець загинув.
- Клас «*RunningSystemController*» відповідає за відображення та наявність елементів ігрового інтерфейсу, логіку, функціонал та об’єкти, які повинні бути присутні тільки за умови, що гра буде компілюватись під мобільний девайс.
- Клас «*MobileControls*» відповідає за управління та дії, які може виконувати персонаж гравця, якщо він грає на мобільному пристрої.
- Клас «*AttackAnimation*» відповідає за анімацію атаки гравця, та виклик функції атаки із класу «AttackContoller».
- Клас «*AttackController*» відповідає за атаку гравця та її звук (клас AudioSource).
- Клас «*Weapon*» відповідає за шкоду, яку наносить гравець своїм ворогам. Також він відповідає за логіку яка перевіряє чи дійсно гравець влучив у ворога.
- Клас «*Reinforcer*» відповідає за підвищення шкоди, яку гравець може наносити своїм ворогам, за перевірку чи активував гравець це посилення.
- Клас «*LeverArm*» відповідає за активацію фінішу, якщо гравець на натиснув на важіль, він не може завершити рівень.
- Клас «*Finish*» відповідає за перевірку, чи був активований важіль та за виклик меню завершення рівня.

- Клас «*PlayerController*» відповідає за:
    - здатність гравця рухатися;
    - швидкість руху;
    - анімації руху гравця, атаку, бігу, стану спокою;
    - поворот модельки персонажу;
    - стрибки гравця;
    - емуляцію фізики;
    - перевірку чи знаходиться гравець у колайдері фінішу (*Finish*), важелю (*LeverArm*) та зони посилення (*Reinforcer*);
    - перевірку у яку сторону дивиться персонаж гравця;
    - перевірку стоїть він на землі чи ні (щоб надати можливість стрибку);
    - відтворення звуків анімацій.
- Клас «*PlayerHealth*» відповідає за виклик меню завершення гри (*GameOver*) та смерть гравця, за звук, анімацію та отримання шкоди від ворогів, за рівень здоров’я, максимальну кількість та його відображення.
- Клас «*EnemyAttack*» відповідає за шкоду, яку будуть наносити вороги гравцю та можливість цього, за час, до повторної атаки.
- Клас «*EnemyHealth*» відповідає за відображення, максимальну кількість, зменшення, оновлення та поточний рівень здоров’я. Також відповідає за смерть ворога.
- Клас «*EnemyVision*» відповідає за відстань на якому ворог може «побачити» гравця, і почати переслідування. Також відповідає за перевірку поточної цілі для переслідування.
- Клас «*EnemyController*» відповідає за:
    - здатність ворога рухатися;
    - дистанцію патрулювання;
    - швидкість руху;
    - швидкість с якою ворог переслідує гравця;

- час який ворог «спостерігає» доходячи до кінця зони патрулювання;
- час на протязі якого ворог переслідує гравця;
- фізику ворогів;
- поворот модельки ворога;
- перевірку, чи достатньо близько знаходиться гравець, щоб нанести йому шкоди.

Висновки до розділу

У результаті проектування 2D-платформера була оцінена архітектура додатку. Що дозволило скласти критерії доброго додатку, яким повинена відповідати програма. Система повинна задовольняти наступним критеріям: ефективність, гнучкість, розширюваність, можливість тестування, код повинен бути добре структурований, зрозумілий та добре читаємий.

Були створені концептуальні прототипи головних екранів додатку. Таким чином, прототип дає розуміння, що до розташування елементів інтерфейсу. Насамперед дозволяє вирішувати питання юзабіліті-інтерфейсу, ще до його створення у реальному додатку.

Була створена діаграма станів, яка дає розуміння всіх можливих станів, у яких може знаходитися програма, а також процес зміни станів у результаті зовнішнього впливу.

Була складена діаграма класів, яка дає повне концептуальне розуміння системи – які класи потрібні, яка функціональність та інформація буде знаходитися у них, я вони будуть взаємодіяти один с одним тощо. Діаграма класів – це відмінний спосіб візуалізувати класи системи додатку, перш ніж почати писати код. Вони представляють собою статичне представлення структури системи програми. Саме діаграма класів дає найбільш повне та розгорнуте розуміння представлення о структурі та зв’язках у програмному коді. Найбільш значущими перевагами цієї діаграми є:

- економія часу при написанні коду;
- більш точне та наглядне представлення основних елементів системи;
- аналіз коду, що до початку його написання;
- представлення основних гілок взаємодії елементів між собою.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ 2D-ПЛАТФОРМЕРА ЗА ДОПОМОГОЮ РЕСУРСІВ UNITY ТА МОВИ ПРОГРАММУВАННЯ C#.

3.1. Структура проекту

Безпосередньо сама розробка ігри проводилась на мові програмування C# у середовищі розробки Visual Studio.

Microsoft Visual Studio – це лінійка продуктів компанії Microsoft, що включає в себе інтегроване середовище розробки програмного забезпечення та ряд інших інструментальних засобів. Данні продукти дозволяють розроблювати як консольні додатки, так і додатки з графічним інтерфейсом, у тому числі з підтримкою технології Windows Forms, а також веб-сайти, веб-додатки, веб-служби як у рідному, так і в керованому коді для усіх платформ, які підтримуються Windows, Windows Mobile, Windows CE, .NET Framework, Xbox, Windows Phone .NET Compact Framework и Silverlight.

Ігрові додатки складаються з трьох сцен: Меню та ігрового поля, яке у цей час поділяється ще на 2 рівні. Ієрархія об’єктів ігрового поля представлена на малюнках нижче (рис. 3.1 та рис. 3.2). Ігрові об’єкти умовно можна поділити на декілька категорій: гравець (Player з усіма вкладеними об’єктами та класами), вороги (Enemy та усі вкладені об’єкти та класи), об’єкти ландшафту (Platforms та Grid, що був створений за допомогою інструменту «Tilemap»), «розумна» камера, що плавно слідує за гравцем (CinemachineVirtualCamera), об’єкти взаємодії з гравцем (LeverArm, Finish та Reinforcer), об’єкти звуку (AudioSource, Music), об’єкти інтерфейсу (ButtonMeshPro, Canvas, Panel, ImageMeshPro, TextMeshPro), об’єкти декорацій (Sky, Clouds, Mountains, Buildings) та об’єкт, що відповідає за відображення мобільного інтерфейсу («RunningSystemController»).

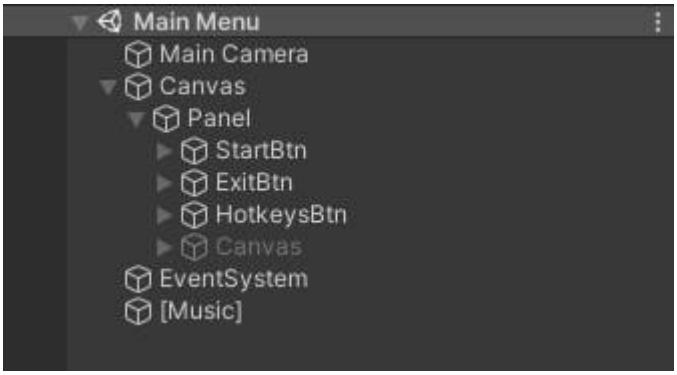

Рис. 3.1. Ієрархія об’єктів головного меню

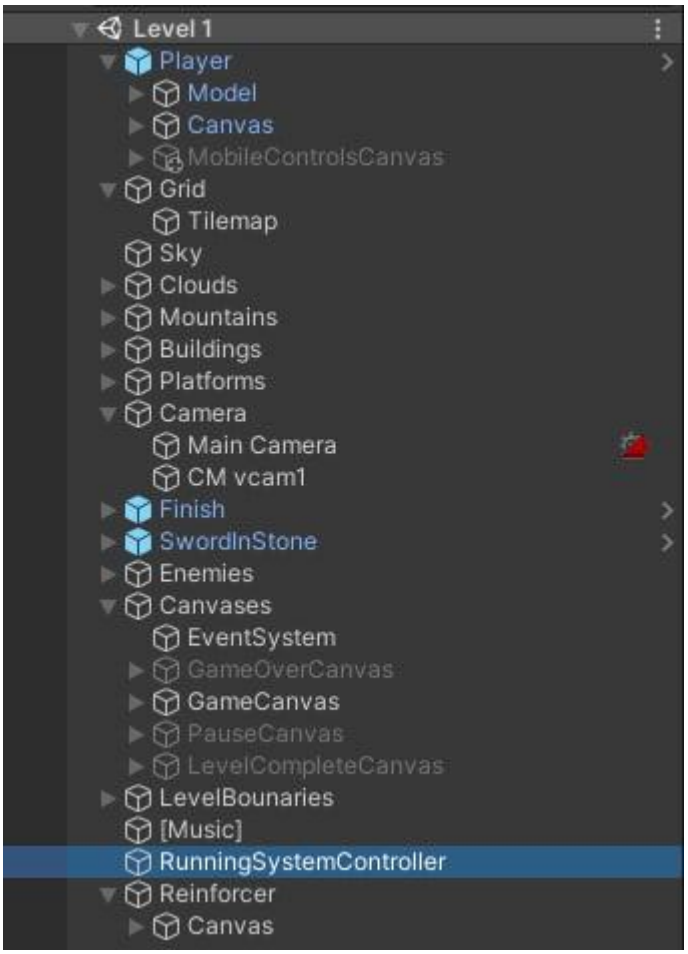

Рис. 3.2. Ієрархія об’єктів ігрового поля

Вкладеність ігрових об’єктів визначає точку відліку їх координат: об’єкт верхнього рівня за точку приймає координати на сцені, а вкладений об’єкт – координати об’єкту більш вищого рівня.

Canvas – це головний елемент UI інтерфейсу, усі елементи, що відображають який-небудь інтерфейс повинні знаходитися у канвасі. Канвас представляє собою абстрактний простір, в якому відбувається настройка та відмальовка UI. Всі UI елементи повинні бути потомками ігрових об’єктів до яких був приєднаний Canvas. Коли создається будь-який UI-елемент із пункту меню (GameObject -> Create UI), Canvas буде доданий автоматично, якщо його немає на сцені.

Традиційно, користувацькі елементи інтерфейсу відображаються прямо на екрані як прості елементи. Це означає, що вони не мають поняття 3D простору, що відображається камерою. Unity підтримує цей спосіб відмальовки у екранному просторі, але також дозволяє інтерфейсам малюватися, як об’єктам на сцені, у залежності від режиму відмальовки (Render Mode). Доступні режими: Screen Space – Overlay, Screen Space – Camera та World Space.

Screen Space – Overlay (простір екрану - поверх). У цьому режимі полотно масштабується для заповнення всього екрану, а потім малюється на пряму, не посилаючись на сцену або камеру (інтерфейс пишеться навіть, якщо на сцені зовсім немає камери). Якщо розмір або роздільна здатність екрану змінюються, інтерфейс автоматично масштабується. Інтерфейс малюється поверх будь-якої іншої графіки. Screen Space - Overlay canvas - потрібно зберігати на верхньому рівні ієрархії. Якщо цього не використовувати, інтерфейс може зникнути з поля зору. Це вбудоване обмеження (рис. 3.3).

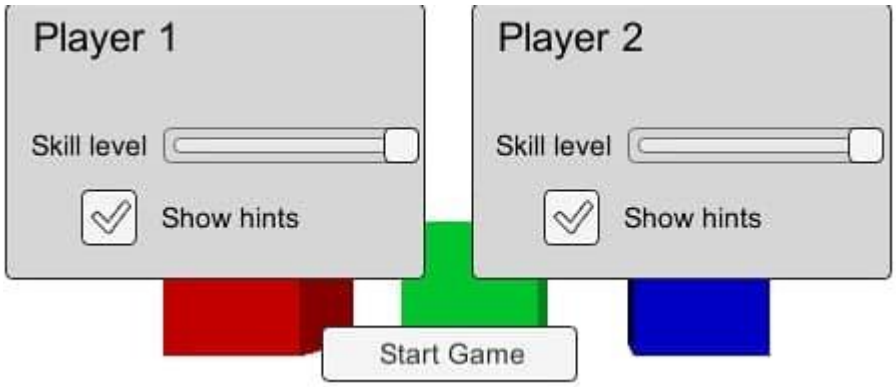

Рис. 3.3. Інтерфейс, що малюється поверх об’єктів сцени

Screen Space – Camera (простір екрану - камера). В цьому режимі, полотно відображається так, як буд-то воно було намальовано на плоскому об’єкті, на деякій відстані заданої камери. Екранний розмір інтерфейсу не змінюється з відстанню, тому що він завжди масштабується щоб у точності заповнювати піраміду видимості камери (Camera frustum). Якщо розмір або роздільна здатність екрану, або піраміда видимості, змінюються – інтерфейс автоматично масштабується, щоб вміщуватися. Будь-які 3D об’єкти сцени, розташовані ближче до камери, ніж плоскість інтерфейсу, будуть відмальовані «над» інтерфейсом, у той час як інші об’єкти, що знаходяться за плоскістю, будуть загороджені (рис. 3.4).

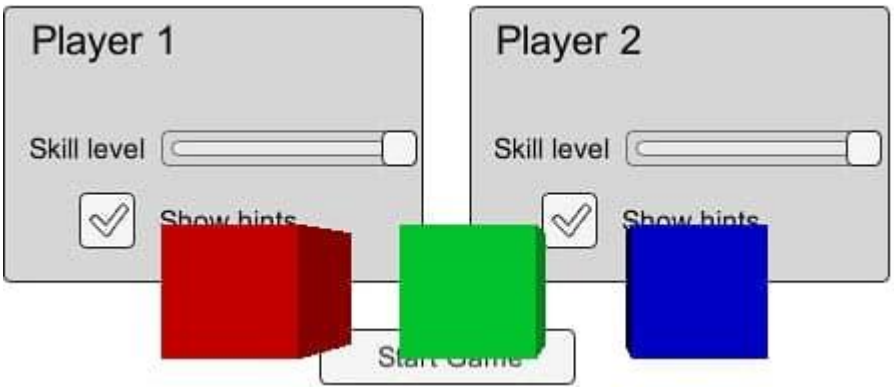

Рис. 3.4. Інтерфейс у режимі «Camera mode» з об’єктами сцени спереду

World Space (пространство миру). Цей режим малює інтерфейс, так, як буд-то він є плоским об’єктом сцени. На відміну від режиму Screen Space – Camera, плоскість не зобов’язана бути перпендикулярною до направлення камери, і може бути орієнтована як завгодно. Розмір полотна може бути встановлений через його Rect Transform компонент, але його екранний розмір буде залежати від куту зору до відстані від камери. Інші об’єкти сцени можуть проходити позаду, крізь або спереду полотна (рис. 3.5).

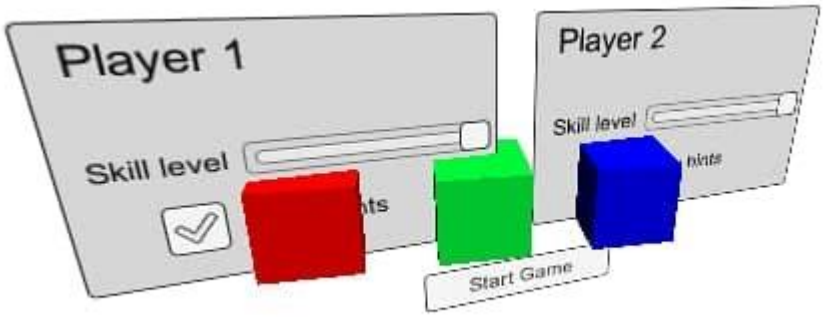

Рис. 3.5. Інтерфейс у просторі миру, що перетинається з об’єктами сцени

3.2. Реалізація ігрового об’єкту Platforms та Grid

Спрайти – це основні 2D-об’єкти, на яких є графічне зображення (так названі текстури). Кожний раз, як Unity створює новий спрайт, він використовує текстуру. Потім ця текстура застосовується до свіжого об’єкту (GameObject), і до нього прикріпляється компонент «Sprite Renderer».

Об’єкти «Platforms» та «Grid» відповідають за так названу «землю» на якій можуть знаходитися персонаж гравця, вороги та інші ігрові об’єкти. Текстури для цих об’єктів були взяті у Asset Store.

Було взято один великий спрайт та він був обрізаний на велику кількість більш менших спрайтів, які мають розмір 190x190 пікселів. Після цього у меню головного спрайту було вказано, что 190x190 пікселів повинно рівнятися одному юниту (Unit – одиниця виміру відстані у Unity) на сцені. Після цього був доданий елемент «Tilemap» з палітрою, куди були перетягнуті всі «нарізані» спрайти. Виконані дії дозволяють «малювати» оточення за допомогою Tilemap (рис. 3.6). Для того, щоб об’єкти мали фізичну поверхню, на якій можуть розташовуватись об’єкти потрібно додати «Tilemap Collider 2D», щоб не навантажувати систему та згрупувати усі вкладені колайдери був доданий додатковий елемент «Composite Collider 2D», - який об’єднує усі колайдери що перетинаються. До ігрового об’єкту «Tilemap» та «Platforms» був доданий тег

«Ground», для чого це було зроблено, буде описано у розділі реалізації об’єкту «Player».

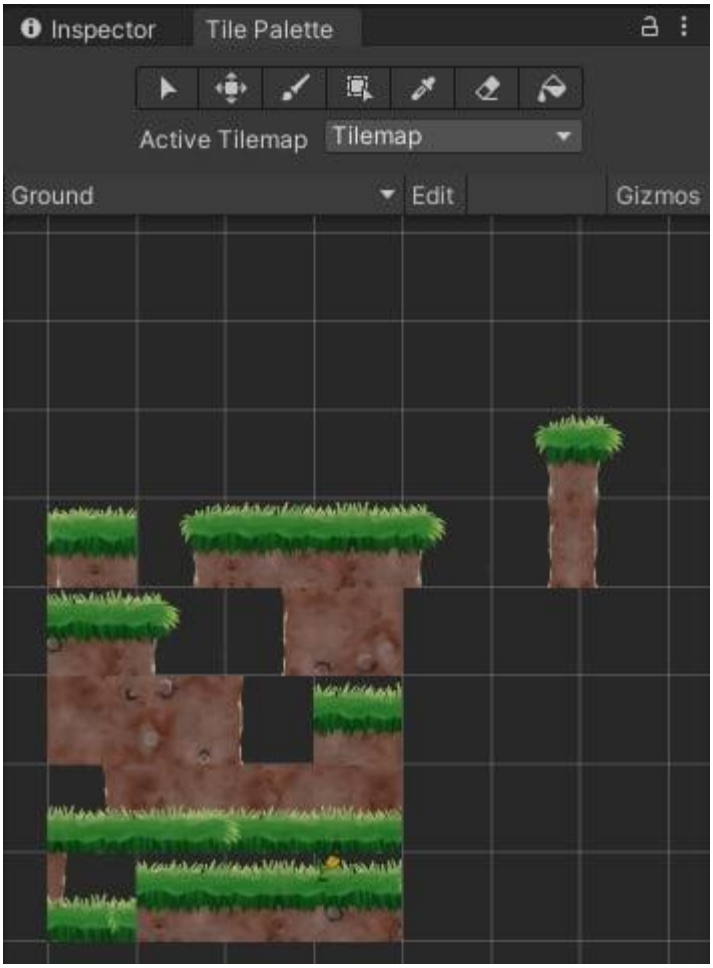

Рис. 3.6. Палітра Tilemap

Безпосередньо об’єкти «Platform» були вирізані за допомогою «Sprite Editor» інструменту (рис. 3.7) та збережені, як нові спрайти ігри. На ігровій сцені є 3 типу платформ: GrassPlatform_short (3 об’єкти цього типу на кожній сцені), GrassPlatform (7 об’єктів цього типу на кожній сцені) та GrassPlatform_large (4 об’єкти цього типу на кожній сцені), - коротка, середня та велика відповідно. Перш за все, усі платформи є префабами проекту. Це було зроблено з такою ціллю: якщо відбувається зміна об’єкту префабу, змінення будуть застосовані до кожного об’єкту цього типу. Якщо потрібно додати який-небудь елемент до об’єкту GrassPlarform, - це не буде потрібно робити для кожного об’єкту безпосередньо. Достатньо додати зміну до об’єкту префабу та

застосувати її до всіх інших об’єктів цього префабу, - це значно заощаджує час, та допомагає уникнути можливих помилок.

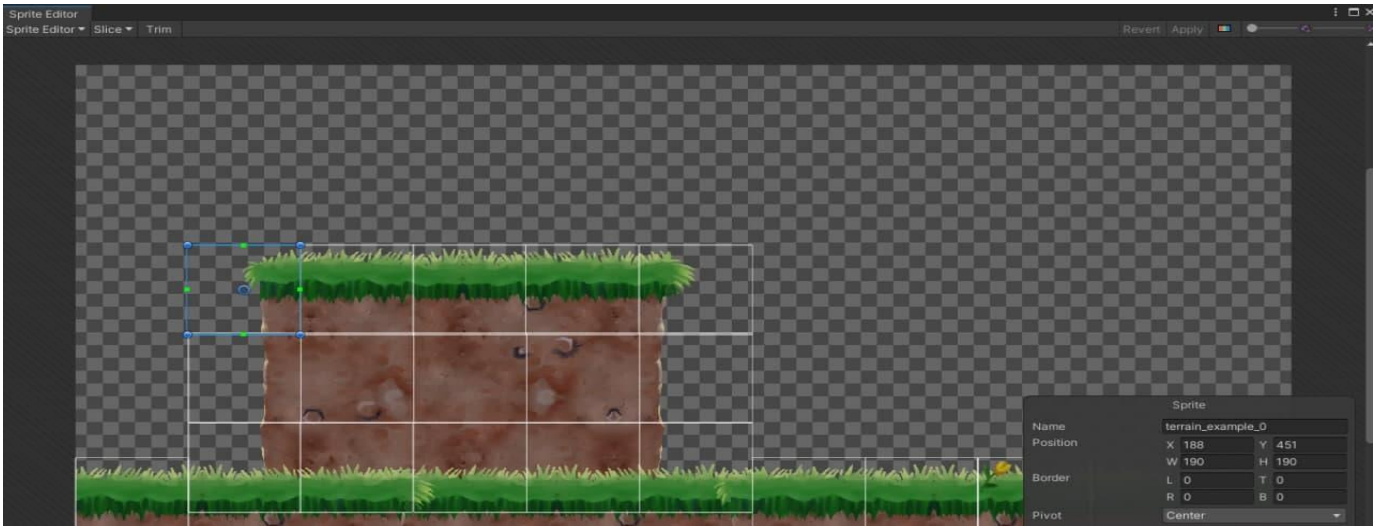

Рис. 3.7. Sprite Editor інструмент

Об’єкти «Platform» так само як і об’єкт «Grid» - має тег «Ground» та містять у собі такі компоненти (об’єкт Transform упускається, бо його містить кожен об’єкт на сцені. Він відповідає за розташування, розмір та масштаб елементів): Sprite Renderer, Box Collider 2D, Platform Effector 2D.

Безпосередньо об’єкт Sprite Renderer відповідає за відображення, форму та інші характеристики що пов’язані с спрайтом об’єкту.

Box Collider 2D – має форму прямокутника, відповідно до форми спрайту землі. Ціль цього компоненту, додати можливість інших колайдерів взаємодіяти з об’єктами, що також мають колайдери (наприклад можливість ворогів або гравця рухатись по землі) (рис. 3.8).

Також платформи містять додаткові 2 об'єкти з межами за допомогою компоненту «Box Collider 2D», за які об’єкт «Enemy» не може виходити під час переслідування гравця (рис. 3.8). У настройках проекту була вимкнута можливість перетину об’єкта «Player» з цими колайдерами.

Platform Effector 2D – модифікує Box Collider 2D, а саме: додає можливість гравцю проходити скрізь колайдер у проміжку 185°. Це зроблено с ціллю, щоб гравець мав змогу стрибнути на платформу знизу, та пройти скрізь неї, але коли він почне падати вниз (завдяки фізиці), він не зможе пройти крізь колайдер зверху вниз - він залишиться на ньому (рис. 3.8).

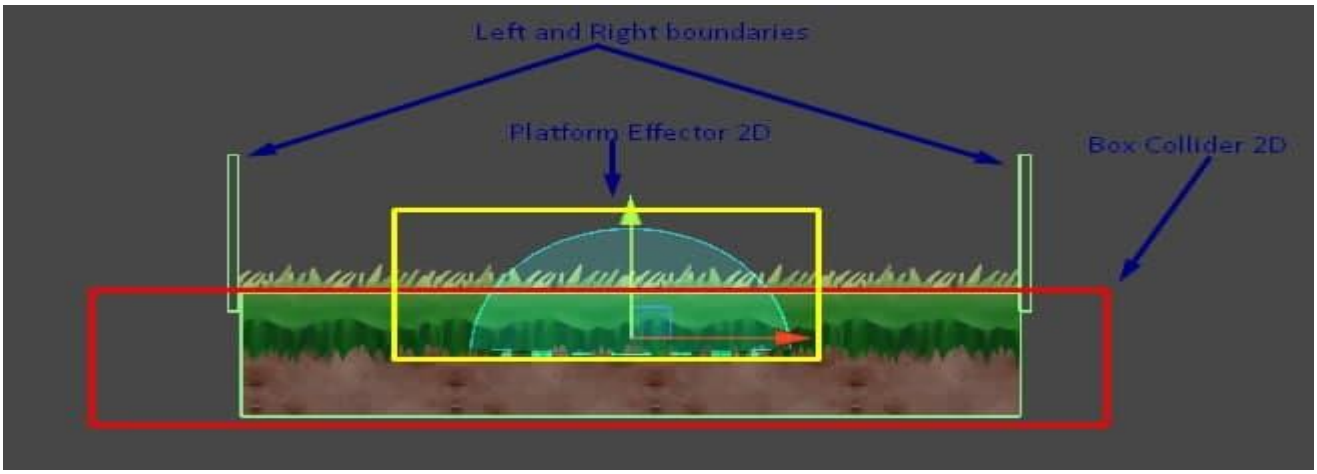

Рис. 3.8. Структура ігрового об’єкту «Platform»

3.3. Реалізація ігрового об’єкту Player

Спрайт гравця було взято із Asset Store у Unity. Для того, щоб емулювати фізику у персонажа гравця та надати йому властивості фізичного об’єкту потрібно додати до ігрового об’єкту Player компонент «Rigidbody 2D».

Компонент «Capsule Collider 2D» надає гравцю можливості бути чутливим до дотику с іншими колайдерами, що дозволяє взаємодіяти з ними.

До гравця був доданий аніматор (Animator – елемент Unity, який відповідає за взаємодію станів ігрового об’єкту). Були створені наступні анімації: анімація отримання шкоди (Hit), анімація спокою (Idle), анімація бігу (Run), анімація атаки (Attack).

«Player» містить у собі 3 компоненти «AudioSource», які відповідають за відтворення звуку наступник анімацій: отримання шкоди, атаки та стрибку. Логіка

подій, які вказують у який саме час потрібно відтворити той чи інший звук описана у скриптах.

Ігровий об’єкт гравця містить у собі 3 скрипта написаних на мові програмування C#: «PlayerHealth», «PlayerController» та «AttackController».

Скрипт AttackController – як, зрозуміло з назви, відповідає за атаку гравця. Він містить у собі 3 змінні, які мають типи: «Animator», «AudioSource» та «Boolean» «_isAttack». «Animator» викликає трігер «attack», що викликає анімацію атаки, та у той же час програється звук атаки за допомогою змінної типу «AudioSource». Безпосередньо сам аніматор об’єкту «Player» містить у собі додатковий скрипт «AttackAnimation», який вимикає можливість гравця наносити шкоду, після завершення анімації.

Скрипт «PlayerHealth» – відповідає за відтворення звуку та анімації коли гравець отримує шкоду, за відображення шкали здоров’я персонажу, за смерть, зменшення та стан здоров’я гравця. Також цей клас викликає меню завершення гри (GameOver), якщо здоров’я гравця впало до нуля.

Скрипт «PlayerController» – один із найбільших та найголовніших класів гри. Він відповідає за анімацію та звук стрибку гравця, можливість руху, фізику об’єкта, за поворот моделі персонажу, за можливість стрибків (якщо колайдер гравця стикнувся з колайдером, що має тег «Ground»), перевірки можливості взаємодії з об’єктами «LeverArm», «Finish» та «Reinforcer».

Безпосередньо сам об’єкт «Player» є префабом ігрового рушія Unity. Він має такі характеристики за замовчуванням:

- SpeedX = 12F. Це означає, що гравець може рухатись з максимальною швидкістю у 3 юніти за секунду. Юніт – як було описано вище, - це одиниця виміру відстані у Unity.
- TotalHealth = 100F. Це означає, що гравець має 100 одиниць здоров’я на старті гри. Ця кількість може змінюватися у залежності від рівня. У сцені «Level 1» - ця кількість дорівнює 120F, у «Level 2» - 100F. Це значення

присвоюється лише один раз – під час ініціалізації об’єкту «Player» на сцені.

- Health – на старті гри дорівнює «TotalHealth». Під час гри може змінюватись, коли гравець отримує шкоди від ворога. Зміна цієї змінної відображається у компоненті «Slider» який знаходиться над моделлю гравця.
- Damage – знаходиться у дочірньому об’єкті «R_Weapon» у скрипті «Weapon». Відповідає за кількість шкоди, яку буде наносити гравець своїм ворогам. За замовчуванням це значення дорівнює 20F. Може змінюватись, якщо гравець активував об’єкт посилення (під час взаємодії із об’єктом «Reinforcer», буде описано нижче).

Він містить у собі такі об’єкти: «Model», «Canvas», «MobileControlsCanvas».

«Model» – вкладає у себе усі спрайти частин тіла персонажа, що утворюють одне ціле. Це зроблено для того, щоб можна було якісно анімувати персонажа гравця. Також туди входить об’єкт «R_Weapon», який містить у собі такі компоненти: «Box Collider 2D», «Weapon» (Script), «AudioSource». За допомогою колайдера відбувається взаємодія з ворогами, коли колайдер торкається об’єкту «Enemy», та булева змінна «_isAttack» (що знаходиться у скрипті «AttackContoller») дорівнює true гравець наносить шкоди ворогу. Також скрипт «Weapon» містить у собі показники шкоди (змінна Damage), яку буде наносити персонаж.

«Canvas» – відповідає за відображення полоси здоров’я (Slider, - показники отримуються із класу «PlayerHealth»). Цей канвас має тип режиму відмальовки як «World Space» (описано у розділі *3.1*). Він є вкладеним у об’єкт Player, тому завжди рухається за ним.

«MobileControlsCanvas» – відповідає за розташування елементів мобільного інтерфейсу, та їх взаємодію с користувачем. Він містить у собі наступні елементи: префаб ігрового об’єкту «FixedJoystick» (дозволяє рухати персонажа у різні сторони), кнопки «AttackBtn», «JumpBtn», «InteractBtn», скрипт «MobileControls». Цей канвас має

режим відмальовки (Render Mode) – Screen Space – Overlay, для того, щоб він був завжди поверх об’єктів сцени, що дозволяє завжди взаємодіяти з ним (усі подробиці цього виду канвасу були описані у розділі 3.1). За замовчуванням цей ігровий об’єкт не є активним. За його активацію відповідає клас «RunningSystemController», який знаходиться в ігровому об’єкті «SystemController».

3.4. Реалізація ігрового об’єкту Enemy

Спрайт ворога було взято із Asset Store. Для того щоб надати властивості фізичного об’єкту до об’єкту «Enemy» було додано «Rigidbody 2D» компонент.

Компонент «Capsule Collider 2D» використовується для того щоб об’єкт «Enemy» міг взаємодіяти с іншими об’єктами, що мають колайдери. До ворога був доданий аніматор та були створені наступні анімації: анімація спокою (Idle), та анімація атаки отримання шкоди (Hit).

Він містить такі характеристики за замовчуванням:

- WalkDistance = 20F. Відстань, яку буде патрулювати ворог. Ця змінна може мати різні значення, у залежності від того, на якій платформі знаходиться ворог та яку відстань йому потрібно проходити.
- PatrolSpeed = 3F. Значення відстані (у одиницях виміру Unit), яку ворог подолає за одну секунду під час патрулю, ця змінна для всіх ворогів є однаковою.
- ChasingSpeed = 6F. Значення відстані, яку ворог подолає за одну секунду під час переслідування об’єкту «Player», змінна для всіх об’єктів має однакове значення.
- TimeToWait = 2F. Коли ворог підходить до кінця зони патрулю, він зупиняється на дві секунди, та дивиться у ту сторону, начебто перевіряючи, чи немає гравця поблизу.

- TimeToChase = 3F. Якщо гравець попав у зону «зору» ворога, то він починає переслідування. Коли персонаж виходить з цієї зони, та ворог «не бачить» гравця запускається таймер на вказаний час, тобто 3 секунди. Якщо за цей час ворог на знайшов гравця, він повертається до зони патрулю.
- MaxDistance = 10F. Ця змінна однакова для всіх ворогів. Містить у собі відстань на яку ворог може «бачити» гравця.
- CircleRadius = 4F. Утворює коло радіусом чотири юніти (знаходиться у кожній точці на протязі відстані MaxDistance), у якому ворог може бачити гравця (рис. 14).
- TotalHealth = 100F. Для кожного об’єкту «Enemy» може бути індивідуальне значення. Встановлюється один раз, при ініціалізації об’єкту, під час завантаження сцени.
- Health = TotalHealth. Змінюється коли ворог отримує шкоду від гравця. Це значення зчитується об’єктом «Slider» для коректного відображення рівня здоров’я.
- Damage = 20F. Показник шкоди, яку ворог може наносити гравцю. Значення є однаковим для усіх ворогів.
- TimeToDamage = 1F. Затримку в одну секунду, після нанесення шкоди гравцю. Після закінчення цього часу, ворог знову може вдарити гравця.

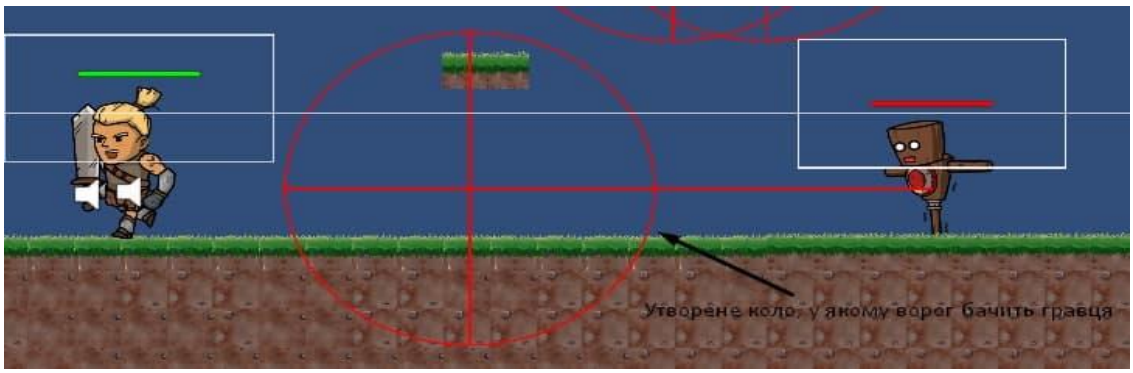

Рис. 3.9. Гізмос у якому ворог бачить гравця

Ігровий об’єкт «Enemy» містить у собі 4 скрипти, що були написані на мові програмування C#: «EnemyController», «EnemyVision», «EnemyHealth» та «EnemyAttack».

Скрипт «EnemyVision» відповідає за поточну ціль атаки, це потрібно, щоб ворог мав змогу переслідувати гравця, якщо той попав у його зону «зору». Суть цього класу полягає у відображенні сфери на певній відстані, цю сферу можна побачити у режимі розробки ігрового додатку. У кінцевому продукті користувач її не буде бачити. Коли об’єкт з тегом «Player» потрапляє у межі цієї сфери, його ігровий об’єкт типу «GameObject» потрапляє у змінну «currentHitObject», ворог починає зчитувати координати гравця та бігти за ним, якщо гравець під час переслідування вийшов за межі сфери – запускається таймер, після завершення якого – ворог перестає переслідування.

Скрипт «EnemyController» - найголовніший клас об’єкту «Enemy». Він відповідає за поворот модельки ворога за допомогою методу Flip(), за відстань яку він буде патрулювати, за швидкість патрулювання, за швидкість переслідування, перевірку чи може ворог нанести атаку гравцю. Також він малює гізмоз (Gizmos) який можна побачити тільки у режимі розробки. Він показує зону, яку патрулює ворог. Якщо колайдери гравця та ворога зіткнулися, то гравцю наноситься шкода.

Скрипт «EnemyAttack» - у цьому скрипті описана кількість шкоди яку він наносить гравцю, час, через який ворог може зробити наступну атаку, запуск таймеру, який контролює цей процес.

Скрипт «EnemyHealth» - містить у собі кількість здоров’я ворога, слайдер (Slider), у який передається інформація про поточний рівень здоров’я, аніматор. Цей клас містить у собі логіку отримання шкоди від гравця, оновлення здоров’я та «смерть» об’єкту «Enemy».

Ігровий об’єкт «Enemy» містить у собі наступні ігрові об’єкти: «Model», «Canvas».

У об’єкті «Model» містяться об’єкти с усіма більш меншими спрайтами, що у сукупності утворюють єдиний цільний спрайт ворога. Це було зроблена для більш детального та точного моделювання анімацій ворога. Усі вкладені об’єкти мають точку відліку початку координат залежно від розташування родительського елементу. Тобто рухаючи по сцені елемент, що знаходиться вище по ієрархії, разом з ним рухаються і усі залежні об’єкти.

Об’єкт «Canvas» містить у собі слайдер (Slider), значення якого встановлюється у класі «EnemyHealth». Цей канвас має режим відмальовки «World Space». Він є вкладеним у об’єкт «Enemy», тому рухається завжди разом з цим об’єктом.

На ігровій сцені на обох рівнях існує по 8 об’єктів типу «Enemy», це означає, що на кожному рівні гравець може зустріти 8 ворогів.

Кількість здоров’я ворогів для кожного індивідуального об’єкту може бути різною, це залежить від розташування, кількості ворогів на платформі та інших ігрових обставин.

3.5. Реалізація ігрових об’єктів взаємодії

Об’єкти взаємодії – це елементи, які знаходяться на ігровій сцені, з якими гравець може безпосередньо взаємодіяти натиснувши клавішу «F», або відповідну кнопку на ігровому інтерфейсі мобільного додатку. Під час взаємодії з цими об’єктами відбуваються різні маніпуляції з ігровою сценою. Ціль цих об’єктів додати деякої ізюминки та азартного інтересу для користувача. На даному етапі розробки міститься 3 ігрові об’єкти взаємодії з гравцем: LeverArm, Finish та Reinforcer. Усі канваси, що вкладені у ці ігрові об’єкти мають тип відмальовки «World Space», - тому що, вони статично стоять на сцені, та моделі гравця, та ворогів повинні бути поверх цих канвасів. Подробиці та можливості цих об’єктів будуть описані нижче.

3.5.1. Ігровий об’єкт LeverArm

Насамперед потрібно сказати, що об’єкт є префабом. Це зроблено для того, щоб усі зміни об’єкту розповсюджувалися на усі сцени. Коли буде додано багато рівнів у гру, зміна префабу торкнеться усіх ігрових об’єктів LeverArm на кожній ігровій сцені.

Об’єкт містить у собі «Box Collider 2D», скрипт «LeverArm», компонент «Sprite Renderer», аніматор, та дві анімації: Activate та Levitate. Коли гравець натискає кнопку взаємодії поряд з об’єктом, меч дістається з каменю та перегортається. Після цього активується наступна анімація, яка примушує його парити у повітрі впродовж усієї сцени, доки гравець не завершить рівень.

Компонент Box Collider 2D не є чутливим до сутичок. У настройках колайдера була виставлена галочка «Is Trigger» (рис. 3.10), що викликає певну подію, коли інший ігровий об’єкт входить, або виходить із зони колайдера. Це було зроблено з однією ціллю: коли гравець знаходиться у межах цього об’єкту та натисне клавішу взаємодії, спрацює подія, яка призведе до активації важелю. Після активації, буде розблокований доступ до взаємодії с об’єктом Finish (рис. 3.11, рис. 3.12).

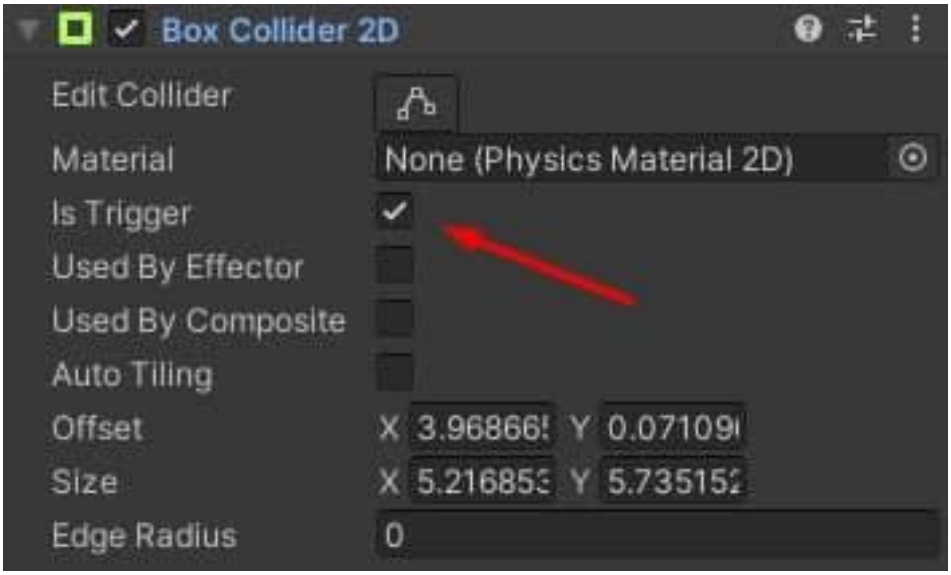

Рис. 3.10. Настройки об’єкту Box Collider 2D

Рис. 3.11. LeverArm до активації

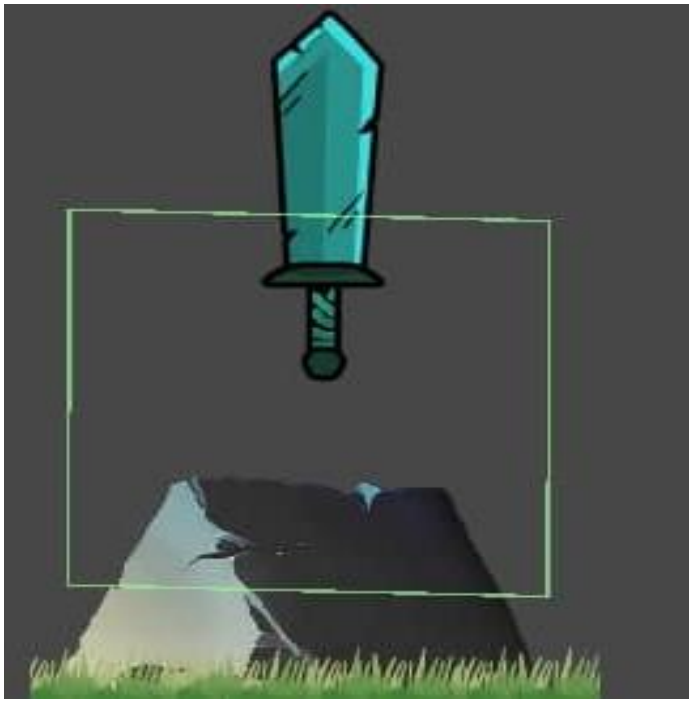

Рис. 3.12. LeverArm після активації

3.5.2. Ігровий об’єкт Finish

Перш за все ігровий об’єкт «Finish» є префабом, згідно з очевидних розумінь. Коли гравець взаємодіє з цим об’єктом – відбувається перехід на наступний рівень. Він містить у собі 2 об’єкти інтерфейсу Canvas, та 3 компоненти: Sprite Renderer, Box Collider 2D та скрипт «Finish».

Sprite Renderer містить у собі спрайт, який відповідає за модель зображення фінішу.

Box Collider 2D так само як і LeverArm має властивість «Is Trigger», що була активована. Це зроблено для того, щоб коли гравець опинявся у зоні колайдеру фінішу, він міг взаємодіяти з ним не впираючись у нього.

Скрипт «Finish» містить у собі булеву змінну «_isActivate» значення якої встановлюється із скрипту «LeverArm» завдяки виклику публічного методу «Activate», який знаходиться у класі «Finish». Якщо ця змінна має значення True, то під час взаємодії з цим ігровим об’єктом – гравець може завершити рівень, після чого активується один із канвасів ігрового інтерфейсу (буде описано у розділі *3.6*), що дозволить перейти на наступний етап гри.

Перший канвас фінішу містить у собі об’єкт «Image» що містить у собі зображення з вкладеним у нього спрайтом та надписом «Finish», з розумінь декоративності гри, та щоб гравець розумів, що саме цей об’єкт є фінішом.

Другий канвас має зображення підказки для гравця за надписом «Activate lever arm», що дає розуміння: для завершення рівню, потрібно активувати важіль. Цей надпис вимикається, із скрипту «Finish», коли ігровий об’єкт LeverArm був активований гравцем (рис. 3.13, рис. 3.14).

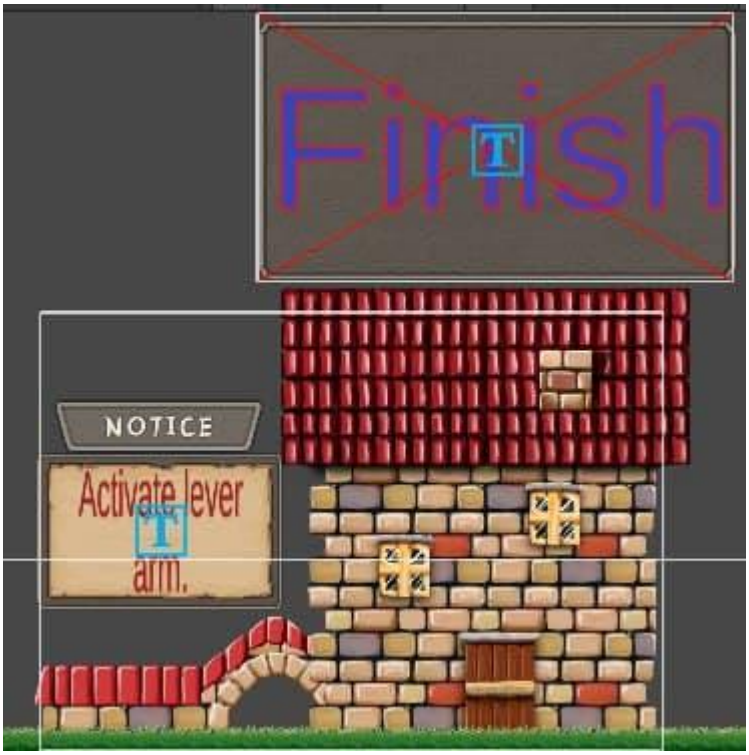

Рис. 3.13. Фініш до активації важелю

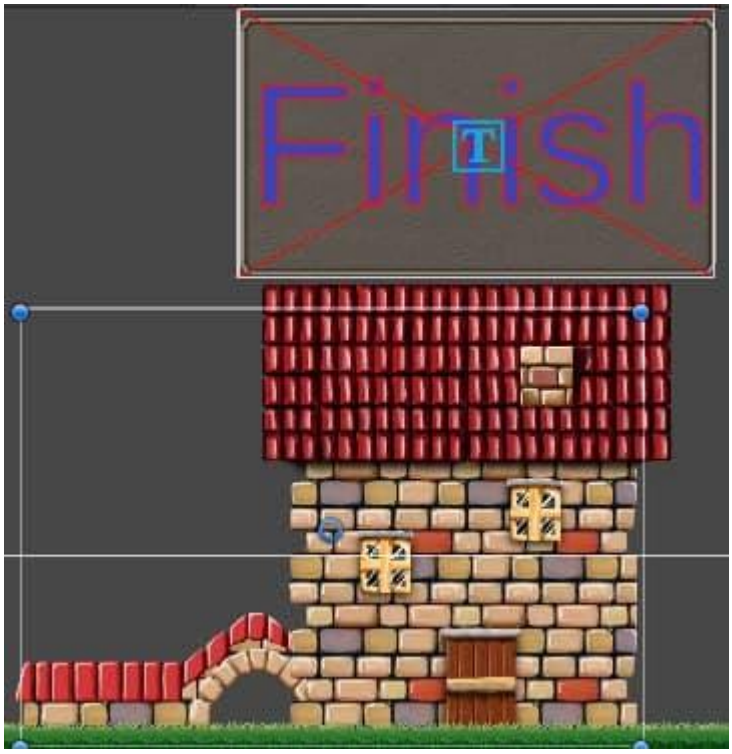

Рис. 3.14. Фініш після активації важелю

3.5.3. Ігровий об’єкт Reinforcer

Reinforcer – є також префабом додатку. Якщо персонаж зміг знайти цей об’єкт та при взаємодії з ним гравець отримує посилення (Reinforce), до кінця поточного рівня. Це посилення підвищує шкоду, яку наносить гравець ворогам. Ця подія значно полегшує проходження рівня, тому що користувач починає бити у 2 рази сильніше.

Reinforcer містить у собі один вкладений об’єкт інтерфейсу – canvas, компоненти «Sprite Renderer», «Box Collider 2D» та скрипт «Reinforcer».

Sprite Renderer – містить у собі спрайт цього ігрового об’єкту.

Box Collider 2D – аналогічно до об’єктів LeverArm та Finish, містить активовану властивість «Is Trigger», щоб Unity «розумів», коли гравець знаходиться поряд з цим об’єктом.

Скрипт «Reinforcer» містить у собі змінну «Weapon». Після взаємодії гравця з ігровим об’єктом «Reinforcer» через змінну «weapon» цей скрипт звертається до змінної «Damage», яка знаходиться у класі «Weapon», та збільшує значення цієї змінної на 20F. Після цього колір зброї гравця буде змінений на темно-червоний. Це дає розуміння, що посилення відбулося. Після того, як об’єкт виконав свої безпосередні задачі, він пропадає зі сцени (рис. 3.15, рис. 3.16).

Елемент ігрового інтерфейсу Canvas, являє собою дочірній елемент об’єкту «Reinforcer», також він вкладає у себе підказку для гравця з текстом «Pick it up to double damage» (рис. 3.15).

Рис. 3.15. Reinforcer та Player до активації посилення

Рис. 3.16. Reinforcer та Player після активації посилення

3.6. Реалізація графічного інтерфейсу

Як вже було описано – об’єкт Canvas у Unity відповідає за відображення графічного інтерфейсу. По-перш потрібно сказати, що розроблена гра має доволі багато графічних інтерфейсів. А точніше їх п’ять: головне меню (MainMenuCanvas), додатковий інтерфейс, що дозволяє подивитись список гарячих клавіш, викликається із основного меню (HotkeysCanvas), меню паузи (PauseCanvas), екран завершення гри (GameOverCanvas) та екран переходу на наступний рівень (LevelCompleteCanvas).

Канвас MainMenu представляє собою ігровий об’єкт, що містить у собі скрипт «MainMenu». Також у нього вкладені наступні ігрові об’єкти: панель з спрайтом (полотно на якому розташовані елементи інтерфейсу), безпосередньо сама панель містить у собі 3 кнопки: StartBtn – відповідає за початок гри, ExitBtn – кнопка завершення гри, HotkeysBtn – кнопка виклику додаткового канвасу с набором гарячих клавіш.

У скрипті «MainMenu» - описані методи-хендлери, що визиваються за допомогою кнопок на графічному інтерфейсі (рис. 3.17).

Рис. 3.17. Графічний інтерфейс головного меню

Канвас Hotkeys – являє собою дочірній елемент об’єкту «MainMenu», та викликається за допомогою кнопки «Hotkeys». Метод HotkeysHandler() що знаходиться у скрипті «MainMenu» активує цей канвас, під час активації викликається анімація появи інтерфейсу.

Меню Hotkeys (рис. 3.18) – має кнопку закриття меню та опис шести кнопок:

- jump – W;
- move right – D;
- move left – A;
- move down – S;
- attack – LMB (Left mouse button – ліва кнопка миші);
- interaction – F.

Коли це меню закривається, визивається анімація закриття.

Рис. 3.18. Меню Hotkeys поверх головним меню

Канвас Pause – викликається безпосередньо під час самої ігри за допомогою кнопки «Escape» або відповідної кнопки (вимикається під час знаходження на екрані паузи) у верхньому лівому куті ігрового екрану. Це меню містить у собі логіку призупинення гри під час знаходження у меню паузи (рис. 3.19).

Містить у собі кнопки Restart – починає рівень заново, Continue – виходить з меню паузи та продовжує гру, Exit – виходить у головне меню.

Рис. 3.19. Меню паузи

GameOverCanvas – викликається, коли гравець загинув, окно меню сповіщує, що гра закінчена. Користувач може за допомогою кнопки Restart почати рівень спочатку, або вийти у головне меню за допомогою кнопки Exit (рис. 3.20).

Рис. 3.20. Меню закінчення гри (GameOver)

LevelCompleteCanvas – викликається із класу «Finish», коли гравець зміг активувати фініш. Активує паузу, щоб вороги, які можливо не були вбиті, не подолали гравця. Містить у собі набір із трьох кнопок: Next level – що переводить гравця на наступний рівень, Restart – що дозволяє почати рівень спочатку, Exit – що виводить у головне меню.

Рис. 3.21. Меню переходу на наступний рівень

Графічний інтерфейс гри на мобільному додатку містить у собі елемент «FixedJoystick», - що відповідає за рух об’єкту «Player». Він містить у собі 3 кнопки: Jump, Attack та Interact:

- Кнопка Jump – викликає метод, який відповідає за стрибок.
- Кнопка Attack – викликає метод, який відповідає за атаку гравця.
- Кнопка Interact – викликає метод, який відповідає за взаємодію гравця із іншими ігровими об’єктами.

Рис. 3.22. Готова ігрова модель

Висновки до розділу

Гра була розроблена у повному обсязі із поставленими задачами, а саме:

- Був реалізований об’єкт «Player», який може добре рухатися у різні сторони, атакувати, стрибати та взаємодіяти з іншими об’єктами. Атакувати своїх ворогів, та бути атакованим. Якщо рівень здоров’я гравця падає до нуля, - гра автоматично завершується.
- Був реалізований об’єкт «Enemy», який патрулює територію у пошуках гравця, якщо він заходить його, він збільшує свою швидкість, та починає переслідування гравця. Ворог може атакувати гравця, та навпаки. Якщо рівень здоров’я ворога зменшується до нуля, він гине.
- Була намальована карта за допомогою Tilemap. Також використовуючи Sprite Editor було зроблено декілька видів платформ, по яких можуть рухатися вороги та гравець.
- Були реалізовані об’єкти взаємодії: LeverArm, Finish та Reinforcer (у цьому випадку план розробки був навіть перевиконаний, на стадії аналізу задачі та її постановки – цей елемент не планувався.)
- Були реалізовані умови завершення гри – якщо гравець не натиснув важіль, він не може перейти на наступний рівень.
- Був реалізований графічний інтерфейс, - що дозволяє користувача взаємодіяти з сценаріями гри.
- Як і планувалось, гра є кросплатформною та може працювати на різних пристроях.
- Був доданий зручний інтерфейс для мобільного управління: джойстик який відповідає за рух гравця, кнопки стрибку, атаки та взаємодії.

ВИСНОВКИ

Була розроблена кросплатформна 2D-гра, що базується на ігровому рушії Unity. Unity - використовує компонентний підхід до розробки ігор, що обертається навколо префабів. Безпосередньо використання префабів дуже облегшує розробку ігри. Не доводиться декілька раз виконувати одні і ті же дії над однаковими об’єктами. Це дозволяє швидше масштабуватись та ефективніше витрачати час. Розробка ігор на Unity – базується на мові програмування C# - що є доволі серйозним аргументом під час вибору ігрового рушія для створення своєї власної гри.

Найбільші переваги розробки на Unity:

- Єдина система активів(assets);
- Вбудований редактор рівнів;
- Відмінна підтримка налаштування та налагодження проекту;
- Обширна бібліотека готових ресурсів;
    - Mono у якості хоста скриптів. Mono надає велику кількість функцій:
    - колекції;
    - input\output(можна одразу бачити дані вхідні та вихідні дані);
    - багатопоточність;
    - LINQ – використання якого значно прискорює розробку.

Задача гравця у створеній грі елементарно проста: знайти та активувати важіль, паралельно долаючи своїх ворогів та збираючись по платформам угору.

Вороги створені доволі сильними, тому для полегшення ігрового процесу на кожному ріні ігрової сцени був схований підсилювач гравця. Якщо він активує його, він підвищить свій рівень шкоди, яку він може наносити ворогу – удвічі.

Також був розроблений зручний інтерфейс для мобільного керування: джойстик, який відповідає за рух гравця, кнопки стрибку, атаки та взаємодії.

На перший погляд гра здається простою, але вона дуже швидко «затягує» користувача своїми труднощами.

Тож, як висновок, гра буде дуже добре монетизуватися за рахунок великої кількості гравців, яка набувається завдяки її кросплатформності та низькому порогу входження нових гравців через простоту керування. Завдяки збільшеній складності проходження кожного наступного рівня вона доволі довго утримує інтерес до цієї гри.

Через зручність використання ігрового рушія Unity, що дозволяє швидко реагувати на потреби гравців, та реалізовувати різноманітні інновації, гра буде дуже конкурентоспроможна у нинішній ігровій індустрії.