Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти бакалавра

зі спеціальності 121 – Інженерія програмного забезпечення

На тему: Розробка гри Pixel Reign 2D

Засвідчую, що в цій кваліфікаційній роботі немає запозичень із праць інших авторів без відповідних посилань.
Студент гр. ІПЗ-20-1
_____ / А. С. Антоновський /

Керівник
кваліфікаційної роботи _____ / І. О. Доценко /

Завідувач кафедри _____ / А. М. Стрюк /

Кривий Ріг
2024

Криворізький національний університет

Факультет: Інформаційних технологій

Кафедра: Моделювання та програмного забезпечення

Ступінь вищої освіти: бакалавр

Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедри
_____ А. М. Стрюк
«___» _____ 2024 р.

ЗАВДАННЯ
на кваліфікаційну роботу

студенту групи ІПЗ-20-1 Антоновському Андрію Сергійовичу

1. Тема: Розробка гри Pixel Reign 2D затверджено наказом по КНУ № 275с від «15» квітня 2024 р
2. Термін подання студентом закінченої роботи: «01» червня 2024р.
3. Вихідні дані по роботі: розроблювана гра повинна бути 2D піксельною аркадою, з елементами файтингу, де гравець знищує ворогів за заданий час.
4. Зміст пояснювальної записки (перелік питань, що їх треба розробити): провести аналіз літературних джерел щодо поняття та особливостей гри 2D піксель файтінг; розробити алгоритми для програмного забезпечення; спроектувати систему та розробити інтерфейс для гри, а також провести тестування.
5. Перелік ілюстративного матеріалу: скріншоти інтерфейсу користувача та ігрових сцен, графічні концепти персонажів та локацій, декомпозиція діаграми ігрової системи, діаграма послідовності, діаграма станів ігрової системи, IDEF0 ігрової системи.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Планування робіт	27.03.2024 – 02.04.2024
2	Розробка технічного завдання	05.04.2024 – 13.04.2024
3	Проведення аналізу предметної області	15.04.2024 – 17.04.2024
4	Проведення структурно-функціонального моделювання	20.04.2024 – 30.04.2024
5	Розробка додатку	01.05.2024 – 27.05.2024
6	Тестування додатку	27.05.2024 – 28.05.2024
7	Оформлення та здача пояснювальної записки та файлів розробленого проекту	29.05.2024 – 07.06.2024

Дата видачі завдання: «15» квітня 2024 р.

Студент _____ / А. С. Антоновський /

Керівник роботи _____ / І. О. Доценко /

РЕФЕРАТ

РОЗРОБКА ІГОР, 2D ПІКСЕЛЬ ФАЙТІНГ, ПРОГРАМУВАННЯ, ДИЗАЙН, ТЕХНОЛОГІЧНІ ПОКАЗНИКИ, ГРАФІКА.

Пояснювальна записка: 71 с., 31 рис., 1 додаток, 11 джерел.

Об’єктом розробки є ігровий продукт – гра жанру 2D піксель файтінг, для вдосконалення навичок програмування та дизайну в галузі інформаційних технологій.

Метою кваліфікаційної роботи є розробка та аналіз особливостей гри 2D піксель файтінг як інструменту для покращення навичок програмування та дизайну, а також оцінка впливу розробки гри на процес навчання студентів у галузі інформаційних технологій.

Для досягнення поставленої мети було виконано наступні кроки:

- проведено аналіз літературних джерел щодо поняття та особливостей гри 2D піксель файтінг;
- розроблено прототип гри з використанням інструментів програмування та дизайну;
- проведено тестування прототипу гри для оцінки його функціональності та користувацької зручності;
- проведено аналіз впливу розробки гри на засвоєння здобувачами концепцій програмування та дизайну.

Розроблена гра може бути використана як навчальний інструмент для здобувачів, що вивчають програмування та дизайн ігор, а також як розважальна продукція для широкого кола гравців.

На основі проведеного дослідження можна зробити висновок про ефективність використання гри 2D піксель файтінг як інструменту для покращення навичок програмування та дизайну в галузі інформаційних технологій.

ABSTRACT

GAME DEVELOPMENT, 2D PIXEL FIGHTING, PROGRAMMING, DESIGN, TECHNOLOGICAL INDICATORS, GRAPHICS.

Explanatory Note: Volume – 71 pages, Number of Images – 31, Appendices – 0, Used Sources – 11.

The object of development is a gaming product – a 2D pixel fighting game, developed to improve programming and design skills in the field of information technology.

The purpose of this work is the development and analysis of the features of a 2D pixel fighting game as a tool for improving programming and design skills, as well as evaluating its impact on the learning process of students in the field of information technology.

To achieve the set goal, the following steps were taken:

- Conducted an analysis of literary sources regarding the concept and features of 2D pixel fighting games;
- Developed a prototype of the game using programming and design tools;
- Conducted testing of the game prototype to evaluate its functionality and user-friendliness;
- Conducted an analysis of the impact of game development on students' understanding of programming and design concepts.

The developed game can be used as an educational tool for students studying game programming and design, as well as entertainment for a wide range of players.

Based on the conducted research, it can be concluded that the use of a 2D pixel fighting game as a tool to improve programming and design skills in the field of information technology is effective.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AR – розширена реальність.

VR – віртуальна реальність.

RPG – (англ. role-playing game): Комп'ютерна рольова гра – жанр комп'ютерних ігор.

IDE – інтегроване середовище розробки.

UE5 – Unreal Engine 5.

UI – («User Interface») – інтерфейс користувача.

UML – (Unified Modeling Languagehttps) уніфікована мова моделювання, використовується у парадигмі об'єктно-орієнтованого програмування

ВСТУП

У сучасному світі ігрова індустрія знаходиться на піку свого розвитку, пропонуючи гравцям безліч незабутніх вражень та неповторних пригод. Завдяки постійному росту технологій і широкому впровадженню цифрових розваг, ігрові проекти стають все більш складними, деталізованими та захоплюючими. Одним із найцікавіших та популярних жанрів є 2D піксель файтінг, який привертає до себе увагу як шанувальників ретро-ігор, так і нового покоління гравців.

Розробка ігор цього жанру є складним та цікавим завданням, яке поєднує в собі різноманітні аспекти програмування та дизайну. Вона дозволяє розробникам не лише проявити свою креативність та технічні знання, але й зануритися у захоплюючий світ ігор, вивчаючи при цьому найновітніші технології та методи розробки.

Мета та цілі кваліфікаційної роботи: розробка гри 2D піксель файтінг як інструменту для покращення навичок програмування та дизайну в галузі інформаційних технологій. Це включає розробку якісної та привабливої гри, вдосконалення навичок програмування та дизайну, а також дослідження впливу гри на процес навчання.

Об’єкт розробки: ігровий продукт – гра жанру 2D піксель файтінг.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд ринку мобільних ігор

Мобільні ігри залишаються одним із найшвидше зростаючих сегментів глобальної ігрової індустрії. З кожним роком вони здобувають все більше популярності серед різних категорій користувачів, включаючи як досвідчених гравців, так і новачків. Станом на 2024 рік мобільні ігри займають вражаючі позиції на ринку, пропонуючи широкий спектр ігор на будь-який смак і бюджет.

Однією з основних тенденцій розвитку мобільних ігор є зростання якості графіки та ігрового досвіду. Завдяки постійному розвитку технологій і використанню новітніх графічних движків, мобільні ігри стають все більш реалістичними та захоплюючими. Крім того, з'являється все більше ігор, які використовують розширену реальність (AR) та віртуальну реальність (VR), що дозволяє користувачам зануритися у неймовірні віртуальні світи прямо зі свого смартфона.

Ще однією важливою тенденцією є зростання популярності ігор зі спільним онлайн-ігровим досвідом. Мобільні ігри, які пропонують можливість грати разом з друзями або в інтернеті з випадковими гравцями, стають все більш популярними серед широкого кола користувачів. Це стимулює розвиток соціальних аспектів гри та сприяє формуванню активної спільноти гравців.

Крім того, зростає інтерес до мобільних ігор серед користувачів усіх вікових категорій, включаючи дорослих. Ігрові розробники активно працюють над створенням ігор з цікавим сюжетом та складними механіками, що привертають увагу навіть найвибагливіших гравців.

Загалом у 2023 році індустрія заробила 184 мільярди доларів, причому майже половина цієї суми припала на мобільні ігри. Іншу половину поділили консольні ігри (53 мільярди), браузерні ігри (1,9 мільярда) і традиційні ігри на ПК (лише 38,4 мільярда). Зростання обсягу мобільного ігрового ринку є

наслідком постійної популярності цього сегменту серед користувачів мобільних пристроїв.

Варто зазначити, що ігрова індустрія зберігає своє лідерство серед мобільних додатків на планеті, але пандемія коронавірусу значно збільшила популярність мобільних ігор. У порівнянні з 2020 роком кількість установок додатків у 2023 році зросла на 23,3%. У 2020 році було встановлено 29,6 мільярдів додатків, а до 2023 року ця кількість збільшилася завдяки високому попиту на мобільні додатки, що сприяло росту ринку управління додатками [1].

Загалом, у 2023 році було завантажено 257 мільярдів додатків, що представляє лише невелике збільшення на менш ніж один відсоток порівняно з 2022 роком, але значне зростання порівняно з 2020 роком [2].

Бізнес-модель у мобільній галузі також зазнала значних змін. Розробники ігор відмовляються від старої моделі одноразової покупки на користь нової моделі мікротранзакцій. Серед популярних методів монетизації можна відзначити продаж персонажів, зброї, пакетів розширення контенту та інші мікротранзакції, що створюють додаткові можливості для комфортної гри.

Мобільні ігри продовжують бути одним із найбільш швидко зростаючих сегментів ігрової індустрії. Завдяки можливості грати у них будь-де і будь-коли, вони привертають увагу широкого кола користувачів. Від простих ігор на кілька хвилин до складних ігрових проектів, мобільні ігри стають популярними серед різних груп людей. Станом на 2023 рік, середній вік гравців мобільних ігор становить 36 років. Це відображає загальну тенденцію до зростання середнього віку гравців, що підкреслює, що мобільні ігри стали популярними серед усіх вікових груп, а не лише серед молоді [3].

Ця статистика показує, що мобільні ігри залучають різноманітну аудиторію, зокрема дорослих старше 35 років, які складають значну частку гравців, що становить майже половину кількості гравців і це, в свою чергу

свідчить про те, що ця форма розваг стає дедалі більш популярною серед старших вікових груп.

Зокрема, у 2021 році мобільні ігри були визнані окремим видом мистецтва в США. Також у цей період став помітний активний розвиток хмарних ігор, що сприяло широкому впровадженню технології 5G. Компанії Google і Microsoft у цьому році представили нові хмарні ігрові рішення, що розширило можливості гравців на мобільних платформах. Крім цього, у 2020 році компанії Apple і Google запустили нові сервіси підписок на ігрові сервіси, що змінило модель доходів від ігрових покупок та реклами у додатках на щомісячні платежі. Однак, на думку експертів, казуальні гравці, можливо, не будуть так активно користуватися такими підписками, як хардкорні геймери. Усього, ринок мобільних ігор станом на 2024 рік продовжує показувати стійкий ріст і популярність, що свідчить про його значний потенціал і можливості для майбутнього розвитку.

1.2 Огляд ігор та ігрових платформ

У сучасному світі кожен гравець може знайти ігрову платформу, що відповідає його фінансовим можливостям і вподобанням. У цьому розділі надано широкий огляд різних класифікацій ігор та платформ для кращого розуміння їх характеристик і параметрів.

1.2.1 Категорії

Різні ігрові платформи мають різноманітні категорії ігор, які виконують різні функції пошуку. Незважаючи на це, загальна назва та призначення категорій ігор залишаються однаковими для гравців на всіх платформах. Більшість сучасних ігор включають кілька категорій, але важливо зазначити, що наявність лише однієї категорії не впливає на їх популярність [4].

Казуальні. Ігри казуального жанру характеризуються простотою правил і доступністю для гравців будь-якого рівня. Вони мають простий

геймплей і яскраву графіку, призначені для коротких сесій, забезпечуючи відпочинок і розвагу у вільний час.

Стратегії. В стратегічних іграх гравці приймають рішення і керують ресурсами для досягнення цілей. Ці ігри можуть включати будівництво міст, управління армією, розвиток торгівлі тощо. Жанр поділяється на піджанри, такі як військові, економічні та глобальні стратегії.

Симулятори. Створюють віртуальні оточення або ситуації, що імітують реальний світ. Гравці можуть керувати автомобілем, літаком, вести господарство або будувати міста. Симулятори можуть бути як реалістичними, так і стилізованими.

Action. Жанр характеризується динамікою, швидкістю та інтенсивним геймплеєм. Включає бойові сцени, перегони, стрільбу та швидкі реакційні завдання, забезпечуючи адреналін від неперервних битв.

Рольові. RPG дозволяють гравцям приймати на себе роль персонажа у віртуальному світі. Гравці розвивають персонажа, виконують завдання, б'ються з монстрами і вирішують загадки. RPG можуть бути одиночними або мультиплеєрними, з великим акцентом на сюжет і взаємодію з оточенням.

Настільні. Цей жанр мобільних ігор включає класичні настільні ігри, такі як шахи, шашки, го, нарди та карти, адаптовані для смартфонів і планшетів. Вони пропонують режими гри з комп'ютером, онлайн або офлайн з іншими гравцями, а також можливість грати з друзями через інтернет або місцеву мережу.

Карткові. Мобільні карткові ігри використовують колоди карт для створення різних ігрових досвідів, включаючи бої, стратегії зі збором колод та головоломки. Гравці можуть будувати власні колоди, вибираючи карти з різними властивостями. Ігри доступні для гри в будь-який час і місці, як самостійно, так і з іншими гравцями онлайн або офлайн.

Музика. В музичних іграх основою геймплею є музика. Ігровий світ реагує на музику, що надихає гравців до дій. Прикладом є гра "Beat Blade".

Спорт. Спортивні мобільні ігри відтворюють різні спортивні дисципліни, такі як футбол, баскетбол, гольф та бокс. Вони пропонують реалістичну графіку та фізичні моделі, різні режими гри, включаючи кар'єру, одиночні матчі та онлайн змагання.

1.2.2 Операційні системи

Ігри також можна класифікувати за типами та поколіннями підтримуваних операційних систем. З погляду корпорацій, які прагнуть збільшити свій постійний дохід та клієнтську базу гравців, вигідно підтримувати різні типи та версії операційних систем. Однак для розробників завжди постає питання, чи це вигідно робити? Чи призведе постійна або тимчасова підтримка, наприклад, декількох менш популярних систем для ігор, до зростання числа нових гравців чи ні? Тому багато компаній-розробників обмежуються підтримкою обмеженої кількості ігрових операційних систем, що дозволяє зменшити основний перелік ОС, які є цільовими на ринку або навіть обов’язковими платформами для максимального поширення гри серед фанатів та гравців [5].

Підтримувані операційні системи існують у трьох формфакторах: комп’ютерні, консольні та мобільні системи.

Мобільні ОС: iOS, Android, Windows Phone, менш популярна Blackberry. Всі придатні для використання користувачами та ігор. Комп’ютерні ОС: Windows, MacOS, Linux. Консольні ОС: Xbox OS, Orbis OS (Playstation OS), NintendoOS (Nintendo Switch OS).

iOS:

- інтуїтивний інтерфейс: легкість використання, простота управління;
- стабільність та безпека: регулярні оновлення для виправлення помилок і захисту від загроз;
- широкий вибір додатків: великий асортимент в App Store;
- інтеграція з іншими пристроями Apple: плавна робота з Mac, Apple Watch, Apple TV;

- оновлення ПЗ: регулярні оновлення додають нові функції та покращують продуктивність.

iOS є потужною та надійною ОС, що забезпечує високу продуктивність і зручність для користувачів Apple.

Android. Операційна система для мобільних пристроїв, розроблена Google на базі ядра Linux. Відомі своєю відкритістю та широким функціоналом.

- відкритість: можливість модифікації та адаптації системи;
- широкий вибір пристроїв: підтримка різноманітних мобільних пристроїв;
- Google Play: магазин додатків з широким асортиментом;
- персоналізація: налаштування пристрою за вподобаннями;
- функціонал: підтримка багатозадачності, сповіщень, сервісів Google;
- безпека: регулярні оновлення для захисту від загроз.

Android є потужною та гнучкою ОС, що підходить для різноманітних потреб користувачів мобільних пристроїв.

Microsoft Windows. Операційна система для персональних комп'ютерів, розроблена корпорацією Microsoft.

- інтуїтивний інтерфейс: легкий у використанні;
- підтримка програм: великий асортимент програмного забезпечення та ігор;
- оновлення: регулярні оновлення для забезпечення безпеки та продуктивності;
- апаратна підтримка: працює на багатьох пристроях, від ПК до ноутбуків і планшетів.

Windows є потужною та універсальною ОС, яка задовольняє потреби різних користувачів і залишається однією з провідних платформ у сфері комп'ютерних технологій.

Linux. Відкрите програмне забезпечення на базі ядра Linux, відоме гнучкістю, надійністю та безпекою.

- відкритість: вільне програмне забезпечення дозволяє користувачам змінювати та адаптувати систему;
- гнучкість та налаштовуваність: різноманітні дистрибутиви та конфігурації під потреби користувачів;
- безпека: вбудовані механізми захисту та шифрування даних;
- надійність: висока стабільність, популярний на серверах та вбудованих системах;
- підтримка спільноти: активна спільнота, що постійно вдосконалює систему.

Ці характеристики роблять Linux популярним для робочих станцій, серверів та вбудованих систем.

MacOS. Операційна система для комп'ютерів Mac, розроблена Apple, відома стабільністю, продуктивністю та інтеграцією з іншими пристроями Apple.

- інтерфейс: елегантний дизайн з корисними функціями, такими як Siri та Metal;
- інтеграція: плавна синхронізація з iCloud та іншими Apple пристроями;
- мультимедіа: оптимізація для роботи з фото-, відео- та музичними редакторами;
- безпека: стійкість до вірусів та шкідливих програм;
- простота: легка установка, оновлення та високий рівень безпеки.

MacOS є привабливим вибором для професіоналів у галузі дизайну, відео- та музичної редакції, які цінують якість та надійність.

1.2.3 Режими

Часом розробники ігор класифікують свої продукти за наявними режимами гри або за можливістю підключення до Інтернету. Понад 9 років тому більшість нових ігрових проектів мали офлайн режими, тобто не потребували підключення до Інтернету. Проте, з часом ринок ігор значно розширився, і тепер все частіше можна зустріти серйозні та дорогі ігрові

продукти, які вимагають постійного підключення до Інтернету або є повністю онлайн проєктами. Щодо режимів гри, основними можуть бути виділені три основні: одиночна гра, кооперативний режим з декількома гравцями та мультиплеєрний режим, який дозволяє грати на різних онлайн серверах. У нових іграх, однак, кооперативний та мультиплеєрний режими часто доступні лише у вигляді онлайн ігрових сесій. Навіть у випадках, коли є комбіновані рішення для ігор, іноді гравці можуть стикатися з одиночними іграми, які все ж вимагають підключення до Інтернету.

1.2.4 Ігрові платформи

Щодня ігрові платформи, що дозволяють гравцеві насолоджуватися відеоіграми, розвиваються. Вони охоплюють весь спектр від звичайних ігрових автоматів у торгових центрах до вдосконалених аркадних машин та великих комп'ютерних клубів. У зв'язку з таким розмаїттям ігрових платформ можна виділити три основних лідера. Перша серед них – персональні комп'ютери, друга – мобільні телефони, а третя – портативні ігрові консолі.

Персональні комп’ютери (ПК) є одними з найпопулярніших ігрових систем завдяки великому вибору ігор, високій якості графіки та звуку, можливості налаштування параметрів та використання різних контролерів. Ігри на ПК часто підтримують модифікації, що розширює їхній геймплей. ПК забезпечують захоплюючий ігровий досвід та взаємодію з великою онлайн-спільнотою гравців.

Консолі – спеціальні пристрої для відеоігор з вбудованим жорстким диском та контролерами. Їх перевагою є ексклюзивні ігри, оптимізоване апаратне забезпечення та операційні системи. Популярні консолі включають PlayStation від Sony, Xbox від Microsoft та Nintendo Switch, кожна з яких має унікальні функції та ексклюзивні ігри.

Мобільні ігрові платформи дозволяють грати в ігри на смартфонах та планшетах з iOS або Android. Їх основні переваги – доступність, зручність і можливість грати будь-де та будь-коли. Мобільні платформи пропонують

широкий вибір ігор різних жанрів та можливість швидкого випуску нових ігор та оновлень. Багато мобільних ігор мають соціальні функції, що дозволяють грати з друзями або змагатися з ними, роблячи геймплей більш захоплюючим.

1.3 Актуальність теми кваліфікаційної роботи

Актуальність теми кваліфікаційної роботи полягає в її зв'язку з поточними тенденціями та викликами у відповідній галузі. У даному випадку, розробка гри жанру 2D піксель файтінг є актуальною темою з огляду на наступні аспекти:

- популярність ігрової індустрії: Галузь відеоігор продовжує зростати та розвиватися, що створює попит на нові ігри та інноваційні підходи до розробки;
- зростання інтересу до ретро-стилю: Ігри в стилі піксель-арту та 2D графіки знову набирають популярності серед гравців, що робить розробку гри даного жанру актуальною та популярною;
- навчання та дослідження: Розробка гри може бути не лише розважальним процесом, але й може слугувати засобом навчання для здобувачів у галузі програмування, дизайну ігор та інформаційних технологій;
- культурний контекст: Ігрова культура та спільнота гравців постійно шукають нові та захоплюючі ігри, що робить розробку ігор актуальною з точки зору задоволення попиту на нові продукти.

Отже, обрана тема має актуальність з погляду популярності ігор, тенденцій в галузі ретро-стилю, можливостей для навчання та розвитку, а також культурного контексту.

1.4 Цілі та завдання кваліфікаційної роботи

Мета кваліфікаційної роботи: розробка гри 2D піксель файтінг, як інструменту для покращення навичок програмування та дизайну в галузі інформаційних технологій.

Об’єктом розробки є ігровий продукт – Pixel Reign 2D.

Розглянувши ринкові аспекти галузі ігор, проаналізувавши найпопулярніші типи ігрових платформ сформулювати задачі, які необхідно вирішити при виконанні роботи, провести детальний аналіз особливостей та можливостей 2D піксель файтінг ігор з метою визначення їхнього впливу на галузь ігрової індустрії та процес навчання студентів у галузі інформаційних технологій.

Задачі кваліфікаційної роботи:

- провести детальний аналіз особливостей та можливостей 2D піксель файтінг ігор з метою визначення їхнього впливу на галузь ігрової індустрії та процес навчання студентів у галузі інформаційних технологій;
- планування роботи;
- розробка геймплею;
- графічний дизайн та анімація;
- програмування та реалізація функціоналу;
- тестування та вдосконалення гри

2 ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Моделювання ігрової системи

Для наочного демонстрування можливих способів використання ігрової системи було створено діаграми сценаріїв використання. Вони наведені нижче на рисунку (див. рис. 2.1), де зображена проектована система у вигляді множини сутностей або акторів, які взаємодіють із системою через так звані варіанти використання.

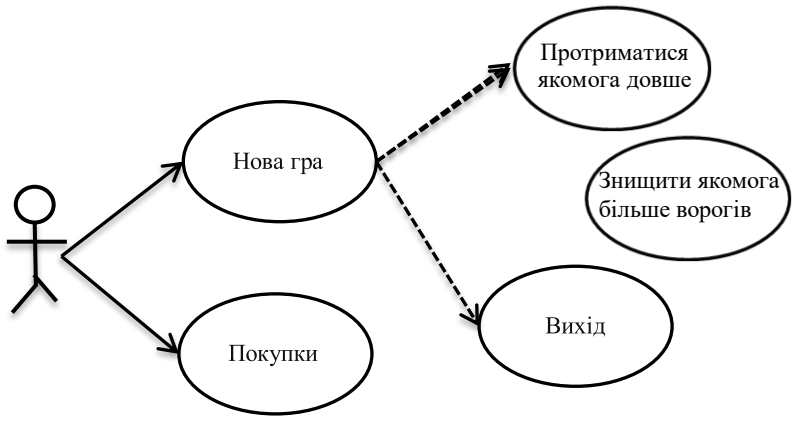

Рисунок 2.1 – Діаграми сценаріїв використання

Для наочної демонстрації функціонального моделювання ігрової системи була створена IDEF0 діаграма, яка показує побудову ієрархічної системи діаграм.

Спочатку описується система в цілому та її взаємодія з навколишнім світом (контекстна діаграма). Потім здійснюється функціональна декомпозиція: система розбивається на підсистеми, кожна з яких описується окремо (діаграми декомпозиції). Підсистеми розбиваються на дрібніші частини, і цей процес повторюється, доки не буде досягнуто необхідної деталізації.

Відповідні діаграми представлені нижче на рисунках (див. рис. 2.2 – 2.3).

Функціональне моделювання ігрової системи (в IDEF0).

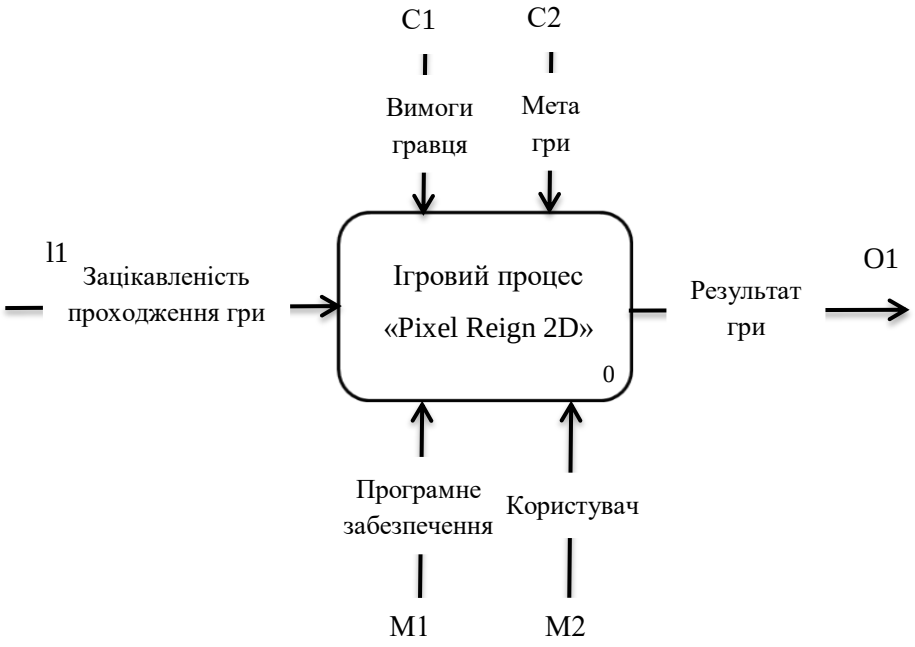

Рисунок 2.2 – IDEF0 ігрової системи

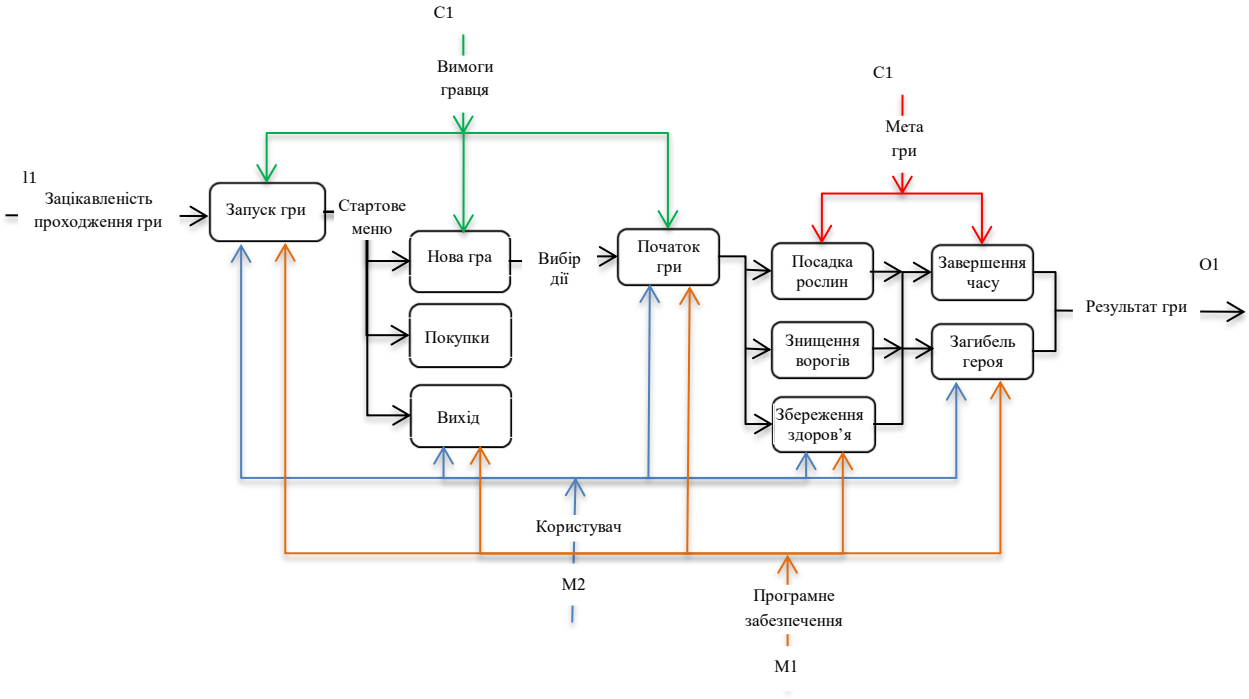

Рисунок 2.3 – Декомпозиція діаграми ігрової системи

Діаграма послідовності відображає взаємодії об'єктів впорядкованих за часом. Зокрема, такі діаграми відображають задіяні об'єкти та послідовність

відправлених повідомлень. На діаграмі послідовностей показано у вигляді вертикальних ліній різні процеси або об'єкти, що існують водночас.

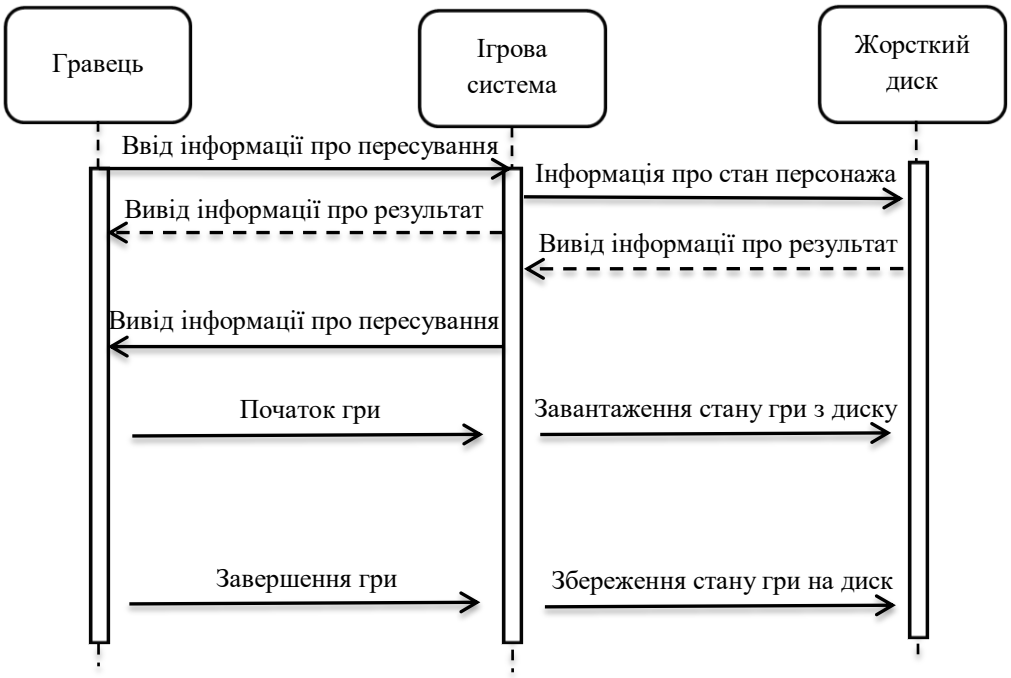

Рисунок 2.4 – Діаграма послідовності

Діаграма станів – діаграма, що визначає зміну станів об'єкта у часі, одна з діаграм моделювання поведінки в UML [6].

Елементи діаграми:

- коло, що позначає початковий стан;
- коло з невеличким точкою у середині, що позначає кінцевий стан;
- стрілка, що позначає перехід. Назва події, що викликає перехід, відзначається над/під стрілкою.

Представлення даної діаграми наведено нижче на рисунку (див. рис. 2.5).

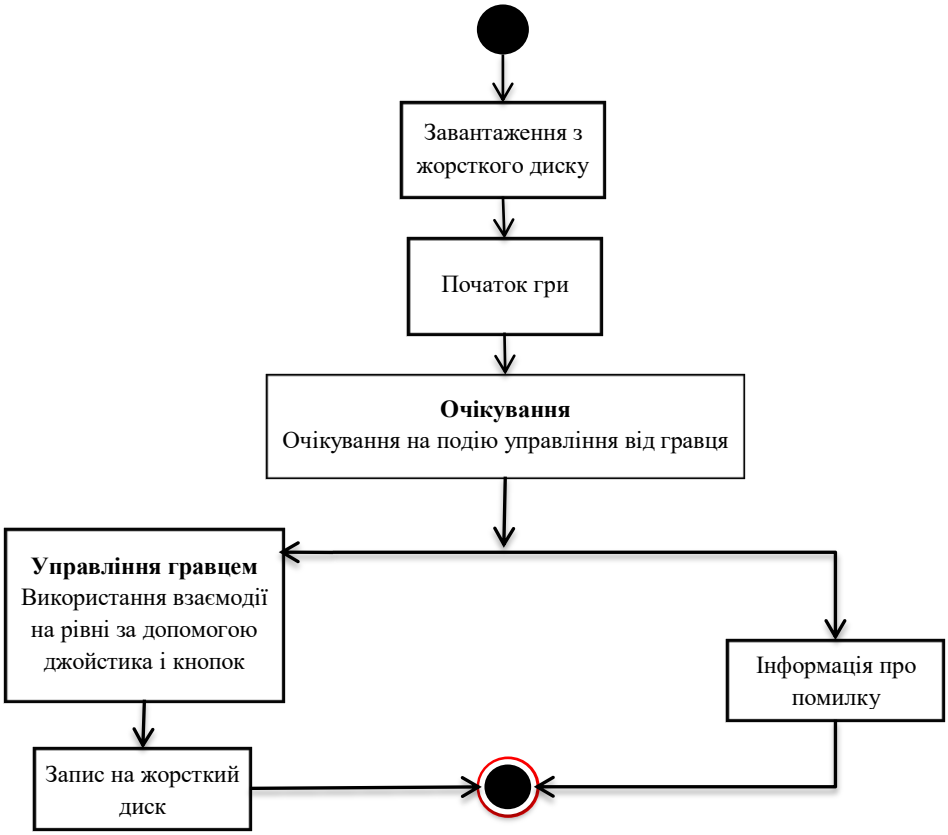

Рисунок 2.5 Діаграма станів ігрової системи.

3 РОЗРОБКА МОБІЛЬНОЇ ГРИ

3.1 Аналіз засобів для розробки мобільної гри

3.1.1 Огляд загальних аспектів ігрових рушіїв

Рушій у світі відеоігор – це як мотор у автомобілі. Він є основою, яка визначає, як гра працює, виглядає та взаємодіє з гравцями. Від моменту, коли гравець натискає кнопку «Старт», до закінчення гри, рушій відповідає за кожен аспект гри.

Уявіть, що рушій – це складний механізм, який складається з різних частин, які працюють разом, щоб забезпечити вам експертний геймплей. Він містить компоненти для рендерингу графіки, обробки фізики, управління штучним інтелектом персонажів, а також звукових ефектів.

Основними складовими рушія є графічний двигун, який відповідає за відображення візуальної частини гри; фізичний двигун, який моделює реалістичну поведінку об'єктів у грі, а також двигун штучного інтелекту, який керує поведінкою ворогів та неігрових персонажів.

Рушій відеогри складається з різних компонентів, які спільно працюють для створення геймплею:

- графічний двигун (Graphics Engine) – відповідає за відображення графіки у грі. Це включає в себе рендеринг 2D та 3D об'єктів, управління освітленням, тінь та ефекти. Графічний двигун оптимізує роботу з графічними ресурсами для плавного та ефективного відображення на екрані;

- фізичний двигун (Physics Engine) – відповідає за моделювання фізики об'єктів у грі. Це включає в себе розрахунок руху, колізії, взаємодії між об'єктами та симуляцію реалістичних фізичних ефектів, таких як гравітація, тертя та інші;

- штучний інтелект (AI Engine) – відповідає за поведінку неігрових персонажів та ворогів у грі. Штучний інтелект може бути програмованим для виконання різних завдань та прийняття рішень на основі встановлених правил та алгоритмів;

- звуковий двигун (Audio Engine) – відповідає за відтворення звуків та музики у грі. Це включає в себе обробку аудіофайлів, створення звукових ефектів, просторову аудіо-модель та інші аспекти аудіо-відтворення;

- мережевий двигун (Network Engine). Якщо гра має мультиплеєрний режим, то мережевий двигун відповідає за забезпечення підключення гравців до мережі, передачу даних між гравцями та синхронізацію гри в реальному часі;

- інструменти розробки (Development Tools) – це набір програмних інструментів для створення, налагодження та тестування відеогри. Вони допомагають розробникам створювати гру швидше та ефективніше.

Ці компоненти працюють разом, щоб створити комплексний геймплей для гравців. Кожен рушій може мати свої унікальні особливості та можливості, які розробники використовують для створення різноманітних ігор. Отже, рушій – це серце кожної відеогри, яке робить гру живою, захоплюючою та неповторною для гравців.

3.1.2 Сучасні графічні двигуни: їх переваги та недоліки.

На даний час існує велика кількість різних двигунів гри. У даному розділі розглянуто найвідоміші глобальні рушії.

3.1.2.1 Unity

Unity – це кросплатформне інтегроване середовище розробки (IDE) для створення ігор і інших інтерактивних додатків. Воно було створене компанією Unity Technologies і було вперше випущене у 2005 році.

Unity надає розробникам зручний інтерфейс та широкий спектр інструментів для розробки ігор для різних платформ, включаючи ПК, консолі, мобільні пристрої, веб- та мобільні пристрої віртуальної реальності [7].

Починаючи зі створення простих ігор для мобільних пристроїв, Unity швидко став одним з найпопулярніших інструментів розробки в індустрії

ігор та інтерактивних додатків. Він використовується як незалежними розробниками, так і великими компаніями, відомий своїм дружнім до користувача середовищем розробки та потужним функціоналом.

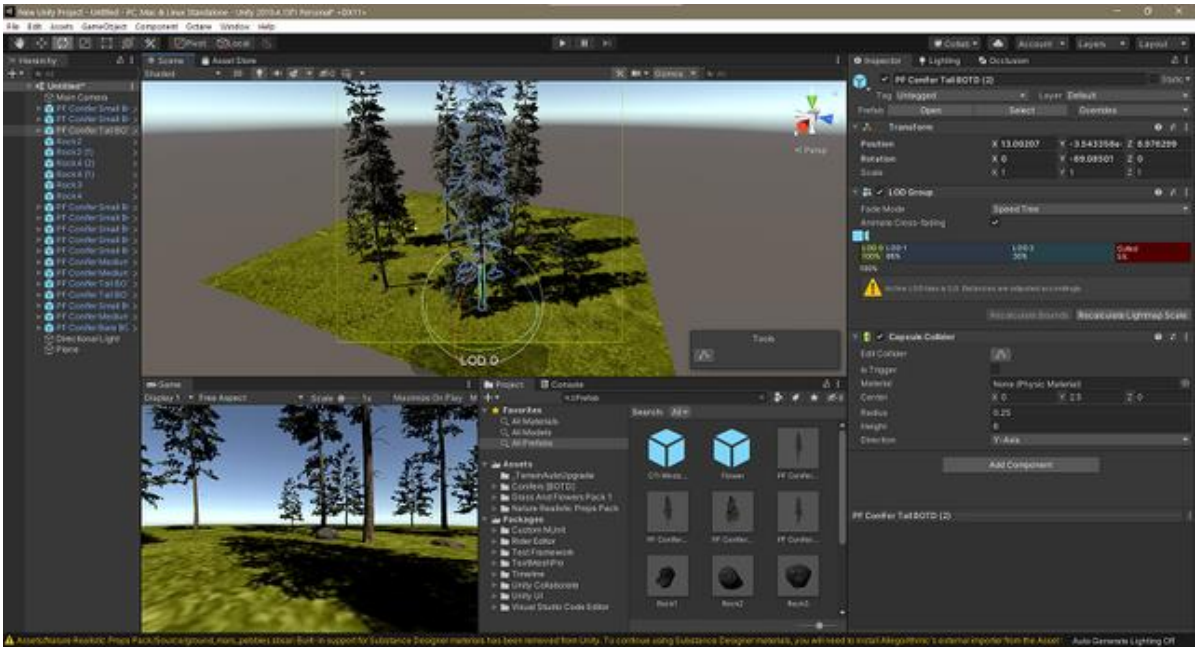

Рисунок 3.1 – Інтерфейс Unity

Ось деякі з його переваг і недоліків:

Переваги Unity:

- кросплатформенність: Unity підтримує розробку для багатьох платформ, включаючи Windows, macOS, Android, iOS, Xbox, PlayStation та VR-пристрої;

- широкий спектр функцій: Unity пропонує інструменти для графіки, фізики, аудіо, анімації, штучного інтелекту та мережевої роботи;

- спільнота та ресурси: Велика спільнота розробників надає підтримку та ресурси, включаючи документацію та уроки;

- швидкість розробки: Використання готових рішень, компонентів і плагінів прискорює процес розробки;

- інтегровані інструменти: Вбудовані редактори для анімації, аудіо, фізики та інших функцій.

Недоліки Unity:

- вартість: Платні ліцензії необхідні для великих комерційних проектів;

- оптимізація для мобільних пристроїв: Можливі проблеми з продуктивністю на мобільних платформах;

- графічна якість за замовчуванням: Може не забезпечити найвищу якість графіки порівняно з іншими двигунами;

- кількість плагінів: Використання сторонніх плагінів може призвести до конфліктів;

- залежність від платформи: Потрібні додаткові зусилля для оптимізації під різні платформи.

Unity залишається популярним інструментом для розробки інтерактивних додатків. Використовується мова програмування C#, тоді як підтримка UnityScript і Boo припинилася. Unity пропонує безкоштовну версію з обмеженнями, включаючи показ логотипу компанії та ліміт доходу у 100 тис. доларів на рік для безкоштовної версії.

Нижче представлені приклади відомих ігор, що були розроблені за допомогою двигуна Unity [8].

Рисунок 3.2 –Приклад гри на Unity. Hearthstone: Heroes of Warcraft

Рисунок 3.3 –Приклад гри на Unity. Cities Skylines

Рисунок 3.4 – Приклад гри на Unity. Subnautica

3.1.2.2 Unreal Engine 5

Unreal Engine 5 – остання версія потужного ігрового двигуна, розробленого Epic Games. Випущений у 2021 році, UE5 вражає передовою технологією й потужністю, роблячи його одним з найбільш передових інструментів для розробки ігор та інтерактивних візуальних додатків. Наніт – нова технологія, що дозволяє відтворювати деталізовані світи без втрат продуктивності. Lumen – глобальна система освітлення, що працює в реальному часі, надаючи реалістичні зображення. UE5 має інтеграцію з

Quixel Megascans для імпорту текстур та матеріалів. Масштабність UE5 дозволяє розробляти великі світи з високою деталізацією. Інструменти розробки включають редактори для логіки гри, анімації, аудіо та штучного інтелекту. У майбутньому UE5 обіцяє надавати розробникам безмежні можливості для створення вражаючих віртуальних світів [9].

Рисунок 3.5 – Інтерфейс Unreal Engine 5

Unreal Engine 5 має вражаючу графіку, інноваційні технології Nanite і Lumen для створення реалістичних світів, швидкість розробки завдяки різноманітним інструментам і редакторам, і можливість створювати величезні світи з великою кількістю деталей. Однак, є деякі недоліки: вимогливість до обладнання, великий розмір файлів, складність вивчення і ліцензійні обмеження. Незважаючи на це, Unreal Engine 5 залишається потужним інструментом з великим потенціалом для розробки ігор і додатків, проте розробники повинні уважно розглянути ці фактори при виборі платформи для розробки. Нижче представлені приклади ігор на Unreal Engine 5 [10]:

Рисунок 3.6 – Приклад гри на Unreal Engine 5. ILL

Рисунок 3.7 – Приклад гри на Unreal Engine 5. Ashes of Creation

Рисунок 3.8 – Приклад гри на Unreal Engine 5. Black Myth: Wukong

3.1.2.3 Godot

Godot – безкоштовний та відкритий ігровий двигун і середовище розробки для створення ігор та інтерактивних додатків. Він має безкоштовний код і підтримує мультиплатформенність на Windows, macOS, Linux, Android, iOS і веб-браузерах. У нього є вбудовані інструменти розробки для сцен, анімації, фізики, штучного інтелекту, а також можливість програмування на GDScript, C# і C++. Завдяки активній спільноті розробників і різноманітній документації, Godot дозволяє створювати ігри і додатки навіть при невеликих вимогах до обладнання [11].

Рисунок 3.9 – Приклад гри на Godot. Lumencraft

Рисунок 3.10 – Приклад гри на Godot. TailQuest Defense

3.1.3 Шаблони та демо-проекти

Після запуску Unity розробнику буде запропоновано кілька варіантів. Перше – створити новий проект; другий – вибрати готові проекти або скористатися різноманітними курсами навчання в розділі «Посібники». Для розробника доступні різноманітні шаблони, такі як 2D, 3D, 3D з додатковими можливостями тощо. Крім того, розробник може випробувати кілька демо-проектів, включаючи:

- "Roll a Ball";
- "Create Kit: FPS";
- "Platformer Microgame";
- "Create Kit: Puzzle";
- "Beginner Scripting" та інші.

3.2 Вибір технології

3.2.1 Вибір ігрового рушія

З огляду на мій досвід роботи з Unity і всі переваги та недоліки інших ігрових рушіїв, розглянутих у цій кваліфікаційній роботі, було вирішено розробляти гру на Unity. Unity має зручний інтерфейс Drag&Drop та можливість легко налаштовувати плагіни, такі як KALI, що складається з різних вікон, дозволяючи відлагоджувати гру прямо в редакторі. Рушій використовує C# для написання скриптів. Раніше підтримувалися Boo (діалект Python, підтримка якого була припинена у версії 5) і модифікація JavaScript, відома як UnityScript (підтримка припинена у версії 2017.1). Фізичні розрахунки виконує PhysX від NVIDIA. Графічний API включає DirectX (зараз підтримується DX 11 та DX 12). Unity також підтримує фізику твердих тіл, тканини та Ragdoll (тряпична лялька). В редакторі є система успадкування об'єктів, де дочірні об'єкти повторюють усі зміни позиції, повороту та масштабу батьківського об'єкта. Скрипти прикріплюються до об'єктів як окремі компоненти. Обравши цей рушій, я отримаю багато

переваг та скорочу час створення гри, оскільки всі необхідні компоненти вже доступні в ньому.

3.2.2 Вибір платформи

Було вирішено обрати ПК як ігрову платформу, оскільки це дозволяє негайно тестувати готовий продукт. Також, враховуючи статистику найпопулярнішого онлайн-сервісу для цифрового розповсюдження мобільних ігор та програм, розробленого та підтримуваного компанією Google, вибір ПК надає ширші можливості для досягнення аудиторії.

3.2.3 Вибір мови програмування

Оскільки основним засобом для розробки була обрана платформа Unity, то і для реалізації представленої системи буде використовуватися мова програмування C#. У цьому рушії відсутній вбудований текстовий редактор, що унеможливлює написання коду на C# безпосередньо в Unity. Тому необхідно вибрати середовище розробки для написання та компіляції коду. Unity за замовчуванням підтримує розробку з використанням Microsoft Visual Studio, але я буду працювати з Sublime Text 4.

Sublime Text 4 – це потужний, багатофункціональний текстовий редактор, призначений для програмістів і веб-розробників. Він відомий своєю високою швидкістю, мінімалістичним інтерфейсом та розширюваністю. Має функції, які полегшують навігацію та редагування коду, підтримує багато мов програмування та має велику кількість плагінів. Забезпечує інтеграцію з Git і підтримку різних систем збірки. Доступний на різних платформах з невисокими системними вимогами. Sublime Text 4, у поєднанні з Unity, забезпечує розробникам потужні функції, які покращують процес розробки гри.

3.3 Графічне оформлення

Основною частиною будь-якої комп'ютерної гри є її візуальна складова. Перед початком опису графічного оформлення, необхідно розглянути основні поняття, що використовуються у геймдизайні, такі як спрайт і тайл. Тайл (англ. Tile) – це повторюваний фрагмент невеликих розмірів, який використовується для створення великих зображень; цей процес називається тайловою графікою. Методи тайлової графіки застосовуються для створення рівнів у двовимірних та тривимірних іграх. Спрайт (англ. Sprite) – у двовимірних іграх є графічним зображенням об'єкта, що може містити кілька зображень для створення анімацій. Для реалізації графічної складової даного прототипу був використаний готовий набір двовимірних спрайтів з Unity Asset Store, який містить спрайти різних жанрів та типів ігор. Було обрано спеціальні спрайти та текстури, що підходять для даного проекту. Для роботи зі спрайтами Unity має вбудовану функцію Sprite Editor (див. рис. 3.11).

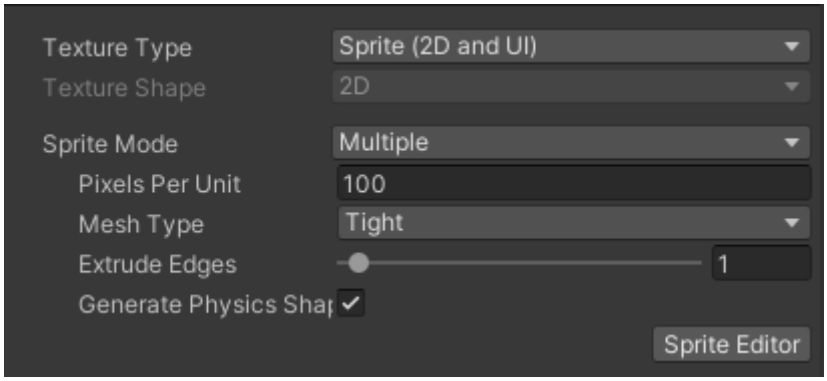

Рисунок 3.11 – Вікно функції Sprite Editor.

Sprite Editor дозволяє відкривати великий спрайт-об'єкт і розділяти його на складові частини для створення анімацій на наступних етапах.

3.3.1 Спрайти головного героя

У якості персонажа в грі будуть використані спрайти рицаря. Він має окремі спрайти для створення анімації простою, бігу, стрибків, атаки та інші.

Нижче буде приведено декілька прикладів створених спрайтів для анімації персонажа.

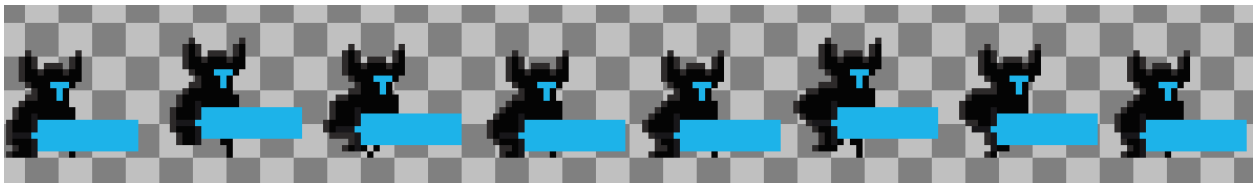

Рисунок 3.12 – Спрайти для анімації бігу

Рисунок 3.13 – Спрайти для анімації атаки

3.3.2 Спрайти інтерфейсу

Після створення спрайтів для ігрового персонажа наступним кроком є підбір спрайтів для фонового зображення та інших візуальних елементів гри. Як фонове зображення було обрано гори – як задній фон, руїни – слідуючий фон (див.рис. 3.14). Руїни були накладені на фон гір, що б під час бігу створювався ефект переміщення персонажа в просторі. Елементи UI для також були створені та підібрані під стиль гри (див. рис. 3.15).

Рисунок 3.14 – Спрайти фонового зображення

Рисунок 3.15 – Спрайти інтерфейсу користувача

Для реалізації ігрового процесу було підібрано спрайти навколишнього середовища, по яких буде стрибати та бігати ігровий персонаж. Було обрано спрайти, які імітували скелястий ґрунт з елементами рослинності та каміння, які дуже добре пасують до атмосфери створюваного світу (див. рис. 3.16).

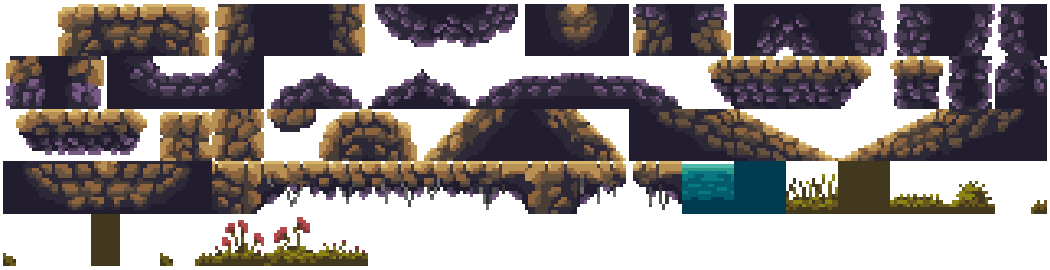

Рисунок 3.16 – Спрайти навколишнього середовища

3.3.3 Спрайт ворогів

Важливим елементом візуальної складової гри є спрайти ворогів, які атакуватимуть ігрового персонажа. Було обрано декілька ворогів, що б було цікавіше грати, нижче приведені їх спрайти (див. рис. 3.17).

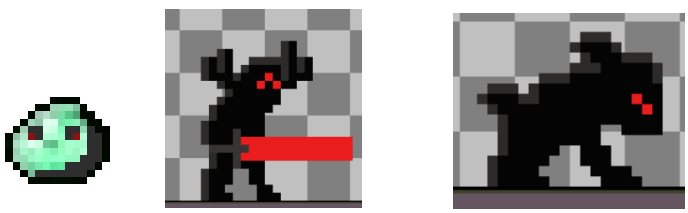

Рисунок 3.17 – Спрайти ворогів

3.4 Гемплей гри

Візуальну складову було завершено, тому можна переходити до поетапного проектування елементів гри. Перед початком програмування та написання скриптів необхідно вирішити проблему з накладанням візуальних елементів гри один на одного. Для цього використовується функція Tags & Layers, яка дозволяє призначити кожному елементу свій шар і регулювати, який шар відображатиметься поверх іншого, щоб уникнути візуальних проблем.

3.4.1 Фізичні характеристики об'єктів

Першим кроком у розробці прототипу гри є надання ігровим об'єктам фізичних властивостей. Спочатку потрібно надати фізичні властивості платформі, яка буде шляхом руху для персонажа. Без цього програма не визначатиме платформу як фізичний об'єкт, і гравець безперервно падатиме. Фізичні властивості об'єктів реалізуються за допомогою колайдерів – спеціальних компонентів, які визначають форму об'єкта для зіткнень з іншими. Таким чином, за допомогою функціоналу Unity, спочатку необхідно додати платформі компонент Polygon Collider 2D, який визначає фізичну форму багатограної фігури.

Після встановлення фізичних властивостей платформ потрібно зробити те ж саме для ігрового персонажа, але з деякими різницями. Оскільки платформа залишається нерухомою, а головний герой виконує дії, спочатку додаємо йому компонент Capsule Collider 2D, який працює так само, як Polygon Collider 2D для платформи, але має форму капсули. Основним компонентом для героя є Rigidbody 2D, який надає можливість руху об'єкту як фізичній одиниці, і дозволяє за допомогою коду задавати різні властивості герою, що є передумовою для програмної частини проекту.

3.4.2 Рух об’єктів

Наступним кроком розробки ігрового персонажа є написання скрипту для руху та стрибків. Спершу задамо змінні:

```
public float speed;
public float jumpForce;
public Joystick joystick;
private bool isGrounded;
public Transform feetPos;
public float checkRadius;
public LayerMask whatisGround;
private Rigidbody2D rb;
```

Змінна rb приймає значення класу Rigidbody2D, який використовується як база для виконання фізичних дій об'єктом. Змінні speed і jumpForce використовуються розробником для налаштування швидкості та сили стрибка відповідно. Особлива увага приділяється змінним whatisGround і isGrounded. Спочатку ігровий об'єкт може стрибати нескінченну кількість разів угору, тому розробник повинен встановити деякі обмеження. Шляхом створення пустого об'єкту з назвою feetPos і його прикріплення як дочірнього до об'єкту Player, можна звертатися до об'єкту feetPos у скрипті.

```
isGrounded = Physics2D.OverlapCircle(feetPos.position,
checkRadius, whatisGround);
if (isGrounded == true && verticalMove>=.5f)
{
      rb.velocity = Vector2.up * jumpForce;
      anim.SetTrigger("takeOff");
}
```

За допомогою цієї частини коду, яка розміщена в функції FixedUpdate відбувається перевірка: у певному заданому радіусі від об’єкту feetPos буде знаходитись інший колайдер, (наприклад, Polygon Collider 2D, який використовується для платформи) тоді виконується перевірка, чи стоїть ігровий персонаж на поверхні, яка у свою чергу обмежує стрибки персонажа.

Далі необхідно зробити так, щоб персонаж рухався. Звертаємось до класу Joystick, який відповідає за рух персонажа по горизонталі. Метод Horizontal приймає значення -1, якщо персонаж рухається вліво, та 1, якщо

рухається вправо. Помноживши на змінну speed, регулюємо, з якою швидкістю буде рухатись наш об’єкт.

```
moveInput = joystick.Horizontal;
rb.velocity = new Vector2(moveInput*speed, rb.velocity.y);
```

Однак цього недостатньо, оскільки об'єкт рухається, але дивиться лише в праву сторону. Тому створюємо наступну функцію:

```
void Flip()
    {
        facingRight =!facingRight;
        Vector3 Scaler = transform.localScale;
        Scaler.x *= -1;
        transform.localScale=Scaler;
    }
```

Спочатку змінемо булеву змінну facingRight на протилежне значення. Якщо вона була true, стає false, і навпаки. Інший блок коду створює нову змінну Scaler і копіює в неї поточний локальний масштаб об'єкта. Потім множить компонент x на -1, тим самим віддзеркалюючи об'єкт по горизонталі. Після цього змінений масштаб повертається назад об'єкту через transform.localScale.

Після того, як було навчено ігрового персонажа рухатися, потрібно додати йому можливість стрибати. Для цього:

```
float verticalMove = joystick.Vertical;
isGrounded = Physics2D.OverlapCircle(feetPos.position,
checkRadius, whatisGround);
   if (isGrounded == true && verticalMove>=.5f)
   {
        rb.velocity = Vector2.up * jumpForce;}
```

Знову звертаємось до класу Joystick, але тепер до методу Vertical. Також слід привернути увагу до змінної isGrounded, яка перевіряє, чи знаходиться об’єкт на поверхні та обмежує стрибки

3.4.3 Анімації для героя

Перший крок у створенні анімації передбачає повернення до функції Sprite Editor, за допомогою якої розділяємо великий спрайт астронавта та

виділяємо з нього спрайти, що показують рух та стрибки астронавта. Для створення анімацій використовуються вбудовані функції Animation та Animator (див. рис 3.18.).

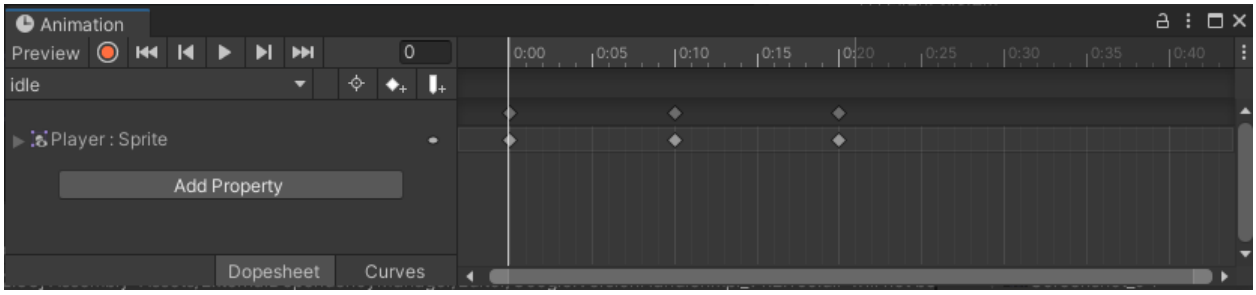

Рисунок 3.18 Вікно Animation

Наступний крок передбачає встановлення зв'язків між кожною створеною анімацією для забезпечення плавних переходів від однієї дії до іншої. (див. рис 3.19).

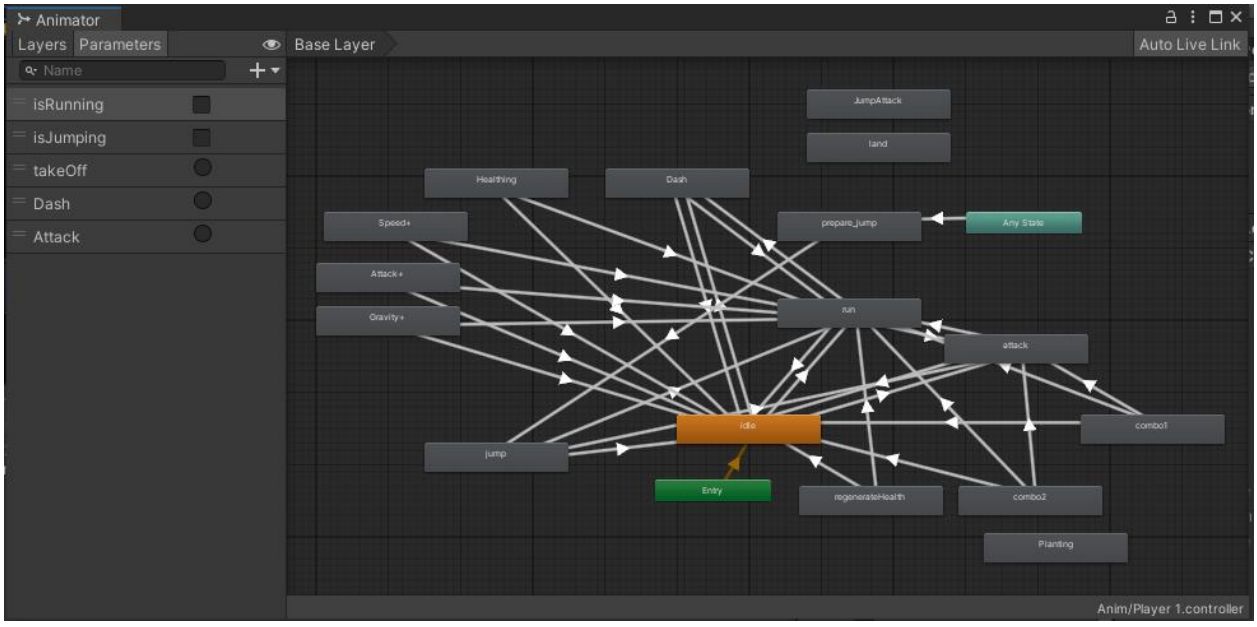

Рисунок 3.19 – Вікно зв’язків анімацій.

3.4.4 Загибель героя та регенерація

Перед тим як приступити до створення ворогів, необхідно налаштувати умови та наслідки смерті головного героя, включаючи шкалу життя.

```
public class Health : MonoBehaviour
```

```
{
    public int health;
}
```

Спершу створюємо клас Health та вводимо зміну health, яка буде дорівнювати значенню, яке буде вказано в інспекторі в Unity та буде базовим значенням очок життя персонажа.

```
public void TakeHit(int damage)
    {
     health -= damage;
     if (health <= 0)
        {
        crystal =
(float)Math.Round(score.playertotalScore/100f);
        Purchase.purchaseCoin += crystal;
        PanelGameOver.SetActive(true);
        Destroy(gameObject, 0.3f);
        }
}
```

Створимо функцію TakeHit і додамо умову, за якою, якщо змінна health зменшується до 0 або нижче, програма визначає, що персонаж загинув. Якщо персонаж помирає, то зароблені очки «score» за допомогою формули перетворюються в кристали «crystal», за які потім можна буде купувати зілля та бомби перед початком гри (див. рис. 3.20) і з'являється панель програшу.

Рисунок 3.20 – Вікно з покупками перед початком гри

Також якщо протягом 10-ти секунд головний герой не отримує пошкодження, відбувається повільна регенерація здоров'я, тобто, кожні 4 секунди відновлюється 1 одиниця здоров'я.

```
IEnumerator RegenerateHealth()
{
    while (health < maxHealth)
    {
        health += 1;
        anim.Play("regenerateHealth");
        if (health > maxHealth)
        {
            health = maxHealth;
        }

        // Обновляємо інтерфейс здоров'я
        healthbar.UpdateHealthbar(maxHealth, health);
        yield return new WaitForSeconds(4f);
    }
    isRegenerating = false;

}
```

3.4.5 Штучний інтелект ворогів

Важливим кроком є створення ворогів. Код, який заставлятиме ворога йти до персонажа з заданою швидкістю:

```
    transform.position = Vector2.MoveTowards(transform.position,
target.position, speed * Time.deltaTime);
```

Якщо ворог знайшов персонажа, тобто він знаходиться в заданому радіусі («radius») то він починає наносити пошкодження з заданим значенням «damage» і перезарядкою «rechange»:

```
public void Attack()
    {
         if(rechange <=0)
         {
         Collider2D[] players =
Physics2D.OverlapCircleAll(attackPos.position, radius,
playerMask);
         for (int i=0; i < players.Length; i++)
              {
              players[i].GetComponent<Health>().TakeHit(damage);

    }
              rechange = startRechange;
```

```
            }
            else
                {
                    rechange -= Time.deltaTime;
                }
    }
```

3.4.6 Генерація хвиль ворогів

Наступний етап розробки стане створення хвиль ворогів, які з’являтимуться з певною періодичністю, для надання мети та динаміки ігровому процесу. Спочатку розміщуємо на сцені кілька порожніх ігрових об'єктів, до яких буде застосовано написаний скрипт. Для початку вводимо змінні, що позначають кількість хвиль та періодичність їх появи:

```
private int rand;
private int randPosition;
public float StartTimeBtwSpawns;
private float timeBtwSpawns;
private float timeSinceLastDecrease;
```

Створюємо масив, в який буде добавлено через інспектор префаби ворогів та точки для їх спавна:

```
public GameObject[] enemy;
public GameObject[] spawnpos;
```

Створюємо функцію, яка відповідатиме за спавн випадкового префаба ворога з масиву enemy[] у випадковій точці для спавну із масиву spawnpos[], зменшуючи час між спавнами за заданою формулою, представленою нижче:

```
void Update()
    {
    if (timeBtwSpawns <= 0)
        {
        rand = Random.Range(0, enemy.Length);
        randPosition = Random.Range(0, spawnpos.Length);
        Instantiate(enemy[rand],
spawnpos[randPosition].transform.position, Quaternion.identity);
        // Зменшити час між спавнами після кожних 2 хвилин
         if (Time.time - timeSinceLastDecrease >= 120f) //
Перевіряємо чи пройшло 2 хвилини
            {
            StartTimeBtwSpawns *= 0.8f; // Зменшуємо час між
спавнами
```

```
            timeSinceLastDecrease = Time.time; // Оновлюємо час
останнього зменшення
            }

           timeBtwSpawns = StartTimeBtwSpawns;
        }
        else
        {
            timeBtwSpawns -= Time.deltaTime;
        }
    }
```

3.4.7 Висадка рослини, зілля та бомби

В грі добавлено таку функцію, як висадка рослини. Це можливо зробити тоді, знаходячись біля спеціальної зони та маючи насіння рослини.

Частина коду яка реалізовує висадку рослини:

```
public void PlantSeed()
{
    if(playerController.playerSeed > 0)
    {
        if (plantedPlant == null)
        {
        // Створюєм новий екземпляр рослини
plantedPlant = Instantiate(plantPrefab,     transform.position +
new Vector3(0.1f, 0.27f, 0f), Quaternion.identity);
                playerController.playerSeed -= 1;
         }
     }
}
```

Після такого як було посаджено рослину і вона виросла (див. рис. 3.21), можна зібрати урожай натиснувши на неї.

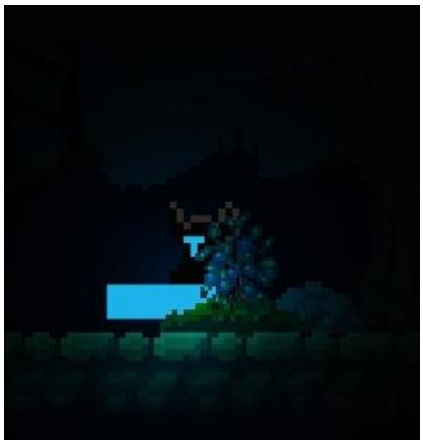

Рисунок 3.21 – Головний персонаж знаходиться біля рослини

Також ресурси випадають з ворогів після їх знищення. В грі існує 4 види ресурсів: золото, м’ясо, ягоди, насіння (див. рис. 3.22). Вони потрібні для прокачки персонажа під час гри.

Рисунок 3.22 – Види ресурсів у грі

Також в грі передбачено ще один вид шкоди для ворогів – бомби. Для запуску потрібно натиснути на персонажа і, не відпускаючи, натягнути в потрібну сторону. Нижче представлена частина коду для запуску бомби:

```
if(numBomb>0)
{
   if (Input.touchCount > 0)
   {
   Touch touch = Input.GetTouch(0);
   Vector3 touchPosition =
mainCamera.ScreenToWorldPoint(touch.position);
   touchPosition.z = 0f;
   switch (touch.phase)
       {
       case TouchPhase.Began:
       if (IsTouchOnPlayer(touchPosition))
           {
           isDragging = true;
           dragStartPosition = touchPosition;
           }
           break;
       case TouchPhase.Ended:
       if (isDragging)
          {
          isDragging = false;
          LaunchGrenade(dragStartPosition, touchPosition);
          numBomb -= 1;
          SaveBomb();
          }
          break;
        }
    }
}
```

Бомби купляються в лобі за кристали і під час гри за золото.

Також в грі добавлено 4 види зілля: атаки (кількість шкоди нанесеного персонажем подвоюється), швидкості (швидкість персонажа зростає вдвічі), регенерації (відновлює 10 одиниць здоров'я), невагомості (персонаж на деякий час стає невагомий) (див. рис. 3.23). У грі добавлено рівні для персонажа, в залежності від рівня відкриваються нові зілля.

Рисунок 3.23 – Зілля та бомба в грі

Приклад коду одного з даних видів зілля, а саме «атаки»:

```
public void PointDamage()
     {
          if(potions.pointDamage > 0)
          {
          potions.pointDamage -=1;
               if (rechangeDamage <= 0)
               {
               GameObject soundAttack =
Instantiate(soundAttackPrefab, transform.position,
Quaternion.identity);
                Destroy(soundAttack, 1f);
                anim.Play("Attack+");
                StartCoroutine(ApplyBonusDamage());
                rechangeDamage = startRechangeDamage;
               }
          potions.SavePoints();
          }

     }
public IEnumerator ApplyBonusDamage()
     {
     playerController.damage *= 2;
     yield return new WaitForSeconds(10f);
     playerController.damage /= 2;
     }
```

3.5 Інструкція користувача

Зустріч починається з головного екрану меню, яке знаходиться у правій частині екрану. З меню можна потрапити в саму гру, міні-магазин, гайд по грі або у налаштування (див. рис. 3.24).

Рисунок 3.24 – Головне меню

Після натискання на кнопку «Start», а потім на «Offline» Відбувається перехід безпосередньо у гру (див. рис. 3.25).

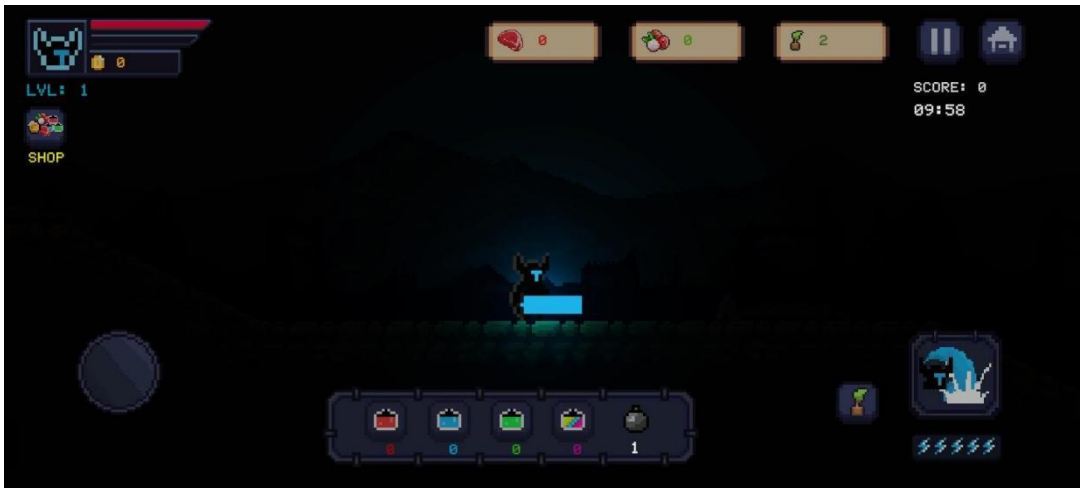

Рисунок 3.25 – Скриншто на початку гри

Попереду головний персонаж на якого прикріплене освітлення. Вам потрібно буде якомога довше протриматися впродовж заданого часу. Після

чого буде видана нагорода в вигляді кристалів, які можна використати в міні-магазині в лобі гри для покупки зілля або бомб.

Зліва знизу розташований джойстик, за допомогою нього можна керувати головним героєм (бігати вліво, вправо та пригати). Код, який відповідає за біг:

```
void FixedUpdate()
      {
      moveInput = joystick.Horizontal;
      rb.velocity = new Vector2(moveInput*speed, rb.velocity.y);
      }
```

Праворуч знизу – кнопка атака, посадки рослини та телепорту.

Частина коду для кнопки атаки:

```
Collider2D[] enemies =
Physics2D.OverlapCircleAll(attackPos.position, radius, enemy);
   for (int i = 0; i < enemies.Length; i++)
   {
enemies[i].GetComponent<EnemyTakeDamage>().TakeDamage(damage);
   }
```

Після гибелі з’являється панель програшу:

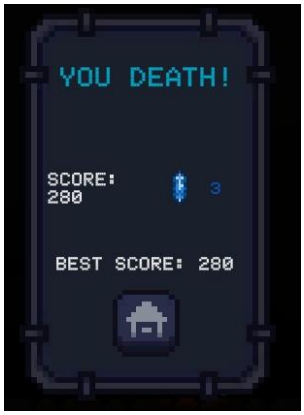

Рисунок 3.26 – Панель програшу

3.6 Тестування гри

Тестування гри Pixel Reign 2D полягала у виявленні помилок, оцінці продуктивності та перевірці відповідності технічним і функціональним вимогам. Тестування здійснювалося вручну на різних пристроях, наданих близькими, включаючи смартфони з операційною системою Android від версії 6.0 до 12.0.

У ході тестування були випробувані різні сценарії, зокрема, перевірка всіх можливих дій користувача, таких як робота з меню, ігровими режимами. Було протестовано продуктивність гри при різних навантаженнях, таких як велика кількість об'єктів на екрані. Особлива увага приділялася зручності інтерфейсу, швидкості реакції на команди та інтуїтивності управління.

Результати тестування показали, що на більшості пристроїв гра працює стабільно, хоча на деяких старіших версіях Android спостерігалися випадкові збої. Було виявлено кілька незначних графічних помилок при високому навантаженні на процесор. Загалом гра витримала високі навантаження без значної втрати продуктивності.

Висновки показують, що гра Pixel Reign 2D в цілому готова до випуску, але рекомендується виправити виявлені графічні помилки на старих версіях Android, покращити інтуїтивність користувацького інтерфейсу в меню та продовжувати моніторинг продуктивності і стабільності гри на різних пристроях після випуску.

ВИСНОВКИ

Проведене дослідження та реалізація проекту гри жанру 2D піксель файтінг дозволили досягти кількох важливих результатів. Детальний аналіз особливостей цього жанру показав його популярність і значний вплив на гравців. Виявлено, що цей жанр приваблює як шанувальників ретро-ігор, так і нове покоління геймерів, завдяки своїй ностальгійній естетиці та простоті управління, у поєднанні з глибоким геймплеєм.

У процесі розробки було успішно створено ключові компоненти гри, включаючи геймплей, графічний дизайн, анімації та реалізацію основного функціоналу. Велика увага була приділена деталізації персонажів і навколишнього середовища, що забезпечило високий рівень візуальної привабливості та занурення у гру. Крім того, інтеграція плавних і динамічних анімацій дозволила підвищити якість ігрового досвіду.

Дослідження показало, що розробка ігор такого жанру значно сприяє покращенню навичок програмування та дизайну у студентів. Працюючи над проектом, зміг покращити свої знання в таких областях, як об'єктно-орієнтоване програмування, створення алгоритмів та структур даних, графічний дизайн та анімація, управління проектами. Це підкреслює важливість включення подібних проектів у навчальні програми з інформаційних технологій. Розробка гри допомогла зрозуміти реальні виклики, з якими стикаються професійні розробники, і підготуватися до роботи в індустрії.

Розробка гри жанру 2D піксель файтінг не лише створює захоплюючий ігровий досвід, але й служить ефективним інструментом для розвитку професійних навичок у галузі програмування та дизайну. Цей проект дозволив проявити творчий підхід, розвинути технічні навички та отримати цінний досвід роботи над комплексними завданнями.

Включення проектів з розробки ігор у навчальний процес сприяє більш глибокому розумінню теоретичних концепцій і їх практичному

застосуванню. Гра в 2D піксель файтінг ігри може сприяти розвитку когнітивних навичок та підвищити концентрацію. Важливо зберігати баланс між грою та навчанням, щоб не втрачати продуктивність у навчальному процесі.

Таким чином, проект створення гри жанру 2D піксель файтінг не лише підтвердив свою актуальність і важливість, але й продемонстрував значний потенціал для подальшого розвитку як в освітньому, так і в комерційному контексті.

ПЕРЕЛІК ПОСИЛАНЬ

1. Applic ation Management Services Global Market Report 2023 [Електроннийресурс] – Режим доступу: https://www.globenewswire.com/en/news-release/2023/4/19/2650064/0/en/Application-Management-Services-Global-Market-Report-2023.html.

2. Прогнози ігрової індустрії на 2024 рік | DeanBeat [Електронний ресурс]. – Режим доступу: https://www.oksim.ua/2024/01/02/prognozi-igrovo%D1%97-industri%D1%97-na-2024-rik-deanbeat/.

3. Експерти прогнозують зростання ринку мобільних ігор у 2024 році [Електронний ресурс]. – Режим доступу: https://chasdiy.org/society/eksperty-prohnozuiut-zrostannia-rynku-mobilnykh-ihor-u-2024-rotsi.html.

4. Ігрові жанри/ QA Teat Lab [Електронний ресурс]. – Режим доступу: https://training.qatestlab.com/blog/technical-articles/games-genres/.

5. FoxmindEd Розуміння різноманітності всіх ОС [Електронний ресурс]. – Режим доступу: https://foxminded.ua/usi-operatsiini-systemy/.

6. Unified Modeling Languagehttps [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/Unified_Modeling_Language.

7. LemonSchool Що таке Unity? [Електронний ресурс]. – Режим доступу: http://surl.li/tyxtn.

8. 7 найкращих ігор, написаних на Unity [Електронний ресурс]. – Режим доступу: https://itvdn.com/ua/blog/article/7best-unity-games.

9. Огляд технологій Unreal Engine 5 з розробниками: застосування, переваги та перспективи [Електронний ресурс]. – Режим доступу: https://gamedev.dou.ua/articles/unreal-engine-technologies-review/.

10. 20 кращих ігор на Unreal Engine 5 [Електронний ресурс]. – Режим доступу: https://cq.ru/articles/gaming/vse-izvestnye-na-dannyi-moment-igry-razrabatyvaemye-na-unreal-engine-5.

11. Документація Godot Engine 4.2 [Електронний ресурс]. – Режим доступу: https://docs.godotengine.org/ru/4.x/about/introduction.html.

Додаток А – Код програми

Файл PlayerController.cs

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

public class PlayerController : MonoBehaviour
{
    public float speed;
    public float jumpForce;
    private float moveInput;
    public Joystick joystick;
    private Rigidbody2D rb;
    private bool facingRight = true;
    private bool facingRight1 = true;
    private bool isGrounded;
    public Transform feetPos;
    public float checkRadius;
    public LayerMask whatisGround;
    private Animator anim;
    private string currentAnimation;
    public float rechange;
    public float startRechange;
    public float rechangeDash;
    public float startRechangeDash;

    [Header("Зброя")]
    //змінні
    public Transform attackPos;
    public LayerMask enemy;
    public float radius;
    public int damage;
    public int combo1;
    public GameObject dashEffect;
    private Animation animDash;
    public Button ButtonAttack;
    public Button ButtonDash;

    private const string CoinKey = "PlayerCoins";
    public int playerCoins;
    public TMP_Text coinText;

    private const string MeatKey = "PlayerMeats";
    public int playerMeats;
    public TMP_Text meatText;

    private const string BerryKey = "PlayerBerry";
```

```
    public int playerBerry;
    public TMP_Text berryText;

    private const string SeedKey = "PlayerSeed";
    public int playerSeed =2;
    public TMP_Text seedText;

    private const string ScoreKey = "PlayerScore";
    public float playerScore;
    public TMP_Text scoreText;

    private const string totalScoreKey = "totalPlayerScore";
    public TMP_Text totalscoreText;
    public float playertotalScore;

    private const string bestScoreKey = "bestPlayerScore";
    public TMP_Text bestscoreText;
    public float playerbestScore;

    public GameObject soundAttackPrefab;
    public GameObject soundComboPrefab;
    public GameObject soundAttackforEnemyPrefab;
    private AudioSource audioSource;
    public CameraPlayer cameraPlayer;
    private bool isRunning = false;
    private Animator CameraAnim;

    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        playerSeed = 2;
        SaveResourse();
        LoadResourse();
        rb = GetComponent<Rigidbody2D>();
        anim = GetComponent<Animator>();
        CameraAnim =
GameObject.FindGameObjectWithTag("MainCamera").GetComponent<Anim
ator>();
        animDash = dashEffect.GetComponent<Animation>();
        combo1 = 0;
        CameraPlayer cameraPlayer =
FindObjectOfType<CameraPlayer>();
    }

    public void UpdateResourseText()
    {
    meatText.SetText(" " + playerMeats.ToString());
    coinText.SetText(" " + playerCoins.ToString());
    berryText.SetText(" " + playerBerry.ToString());
    seedText.SetText(" " + playerSeed.ToString());
    scoreText.SetText("Score: " + playerScore.ToString());
```

```
    totalscoreText.SetText("Score: " +
playertotalScore.ToString());
    bestscoreText.SetText("Best Score: " +
playerbestScore.ToString());
    }

    void Update()
    {
        playertotalScore = playerScore;
        if (rechange > 0)
        {
            rechange -= Time.deltaTime;
            if (rechange < 0)
            {
                rechange = 0;
            }
        }
        if (rechangeDash > 0)
        {
            rechangeDash -= Time.deltaTime;
            if (rechangeDash < 0)
            {
                rechangeDash = 0;
            }
        }
        UpdateResourseText();
    }

    void FixedUpdate()
    {
        moveInput = joystick.Horizontal;
        rb.velocity = new Vector2(moveInput*speed,
rb.velocity.y);

        if(facingRight == false && moveInput>0)
        {
            Flip();
        }
        else if(facingRight == true && moveInput<0)
        {
            Flip();
        }

        if(moveInput==0)
        {
            anim.SetBool("isRunning",false);
            isRunning = false;

        }
        else
        {
            anim.SetBool("isRunning",true);
```

```
            isRunning = true;

        }
        float verticalMove = joystick.Vertical;
        isGrounded = Physics2D.OverlapCircle(feetPos.position,
checkRadius, whatisGround);
        if (isGrounded == true && verticalMove>=.5f)
        {
            rb.velocity = Vector2.up * jumpForce;
            anim.SetTrigger("takeOff");
        }
        if(isGrounded == true)
        {
            anim.SetBool("isJumping", false);
        }
        else
        {
            anim.SetBool("isJumping",true);
        }

        if(rechange <=0)
        {
            ButtonAttack.interactable = true;
        }
        else
        {
            ButtonAttack.interactable = false;
        }
        if(rechangeDash <=0)
        {
            ButtonDash.interactable = true;
        }
        else
        {
            ButtonDash.interactable = false;
        }

         if (isRunning && isGrounded)
        {
            if (!audioSource.isPlaying)
            {
                audioSource.Play();
            }
        }
        else
        {
            if (audioSource.isPlaying)
            {
                audioSource.Stop();
            }
        }
```

```
    }



    void Flip()
    {
        facingRight =!facingRight;
        Vector3 Scaler = transform.localScale;
        Scaler.x *= -1;
        transform.localScale=Scaler;
    }
    void FlipDash()
    {
        facingRight1 =!facingRight1;
        Vector3 Scaler1 = dashEffect.transform.localScale;
        Scaler1.x *= -1;
        dashEffect.transform.localScale=Scaler1;
    }


public void Attack()
{

    if (rechange <= 0.0001f)
    {


        Debug.Log("Attack");
        combo1 += 1;
        if (combo1 == 3)
        {
            if (anim.gameObject.activeSelf) // Перевірка
активності об'єкта з аніматором
            {
                anim.Play("combo1");
                GameObject soundCombo =
Instantiate(soundComboPrefab, transform.position,
Quaternion.identity);
                Destroy(soundCombo, 1f);
                Invoke("ShakeCombo1", 0.5f);
            }
        }
        else if (combo1 == 6)
        {
            if (anim.gameObject.activeSelf) // Перевірка
активності об'єкта з аніматором
            {
                anim.Play("combo2");
                combo1 = 0;
```

```
                GameObject soundCombo =
Instantiate(soundComboPrefab, transform.position,
Quaternion.identity);
                Destroy(soundCombo, 1f);
                Invoke("ShakeCombo1", 0.6f);
            }
        }
        else
        {
            if (anim.gameObject.activeSelf) // Перевірка
активності об'єкта з аніматором
            {
                anim.Play("attack");
                GameObject soundAttack =
Instantiate(soundAttackPrefab, transform.position,
Quaternion.identity);
                Destroy(soundAttack, 1f);
            }
        }
        Collider2D[] enemies =
Physics2D.OverlapCircleAll(attackPos.position, radius, enemy);
        for (int i = 0; i < enemies.Length; i++)
        {

enemies[i].GetComponent<EnemyTakeDamage>().TakeDamage(damage);
            GameObject soundAttackforEnemy =
Instantiate(soundAttackforEnemyPrefab, transform.position,
Quaternion.identity);
            Destroy(soundAttackforEnemy, 1f);
        }
        rechange = startRechange;
    }
}

    void ShakeCombo1()
    {
        cameraPlayer.StartShakeCombo1();
    }
    public void Dash()
    {
        if(rechangeDash < 1f)
        {
        if(facingRight == true)
        {
            anim.SetTrigger("Dash");
            GameObject dashEffectClone = Instantiate(dashEffect,
new Vector2(transform.position.x + 1.8f, transform.position.y),
Quaternion.identity);
            anim.SetTrigger("dashEffect");
            Destroy(dashEffectClone, 0.28f);
            transform.position =
Vector2.MoveTowards(transform.position, new
```

```
Vector2(transform.position.x + 4f, transform.position.y),
speed);
        }
        else if(facingRight == false)
        {
            FlipDash();
            anim.SetTrigger("Dash");
            GameObject dashEffectClone = Instantiate(dashEffect,
new Vector2(transform.position.x - 1.8f, transform.position.y),
Quaternion.identity);
            anim.SetTrigger("dashEffect");
            Destroy(dashEffectClone, 0.28f);
            transform.position =
Vector2.MoveTowards(transform.position, new
Vector2(transform.position.x - 4f, transform.position.y),
speed);
        }
        rechangeDash = startRechangeDash;
    }
    }

    private void OnDrawGizmosSelected()
    {
        Gizmos.color = Color.black;
        Gizmos.DrawWireSphere (attackPos.position, radius);
    }

    void ChangeAnimation(string animation)
    {
        if(currentAnimation == animation) return;

        anim.Play(animation);
        currentAnimation = animation;
    }

    void LoadResourse()
    {
        if (PlayerPrefs.HasKey(CoinKey))
        {
            playerCoins = PlayerPrefs.GetInt(CoinKey);
        }
        else
        {
            playerCoins = 0;
        }
        if (PlayerPrefs.HasKey(MeatKey))
        {
            playerMeats = PlayerPrefs.GetInt(MeatKey);
        }
        else
        {
            playerMeats = 0;
```

```
        }
        if (PlayerPrefs.HasKey(BerryKey))
        {
            playerBerry = PlayerPrefs.GetInt(BerryKey);
        }
        else
        {
            playerBerry = 0;
        }
        if (PlayerPrefs.HasKey(SeedKey))
        {
            playerSeed = PlayerPrefs.GetInt(SeedKey);
        }
        else
        {
            playerSeed = 2;
        }
        if (PlayerPrefs.HasKey(ScoreKey))
        {
            playerScore = PlayerPrefs.GetFloat(ScoreKey);
        }
        else
        {
            playerScore = 0f;
        }
        if (PlayerPrefs.HasKey(totalScoreKey))
        {
            playertotalScore =
PlayerPrefs.GetFloat(totalScoreKey);
        }
        else
        {
            playertotalScore = 0f;
        }
        if (PlayerPrefs.HasKey(bestScoreKey))
        {
            playerbestScore =
PlayerPrefs.GetFloat(bestScoreKey);
        }
        else
        {
            playerbestScore = 0f;
        }
    }
    public void bestScore()
    {
        if(playertotalScore > playerbestScore)
        {
            playerbestScore = playertotalScore;
            SaveResourse();
            UpdateResourseText();
        }
```

```
    }

    public void SaveResourse()
    {
        PlayerPrefs.SetInt(CoinKey, playerCoins);
        PlayerPrefs.SetInt(MeatKey, playerMeats);
        PlayerPrefs.SetInt(BerryKey, playerBerry);
        PlayerPrefs.SetInt(SeedKey, playerSeed);
        PlayerPrefs.SetFloat(ScoreKey, playerScore);
        PlayerPrefs.SetFloat(totalScoreKey, playertotalScore);
        PlayerPrefs.SetFloat(bestScoreKey, playerbestScore);
        PlayerPrefs.Save();
    }

    void OnApplicationQuit()
    {
        SaveResourse();
    }
}
```

Файл Health.cs

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

public class Health : MonoBehaviour
{
    public int health;
    public int bonusHealth = 10;
    [SerializeField] public int maxHealth;
    [SerializeField] private HealthBar healthbar;
    private Animator anim;
    private float rechange;
    public float startRechange;
    private Material matBlink;
    private Material matDefault;
    private SpriteRenderer spriteRend;
    private UnityEngine.Object explosion;
    public Button ButtonSetHealth;
    public BuyPotions potions;
    public PlayerController score;
    public CameraPlayer cameraPlayer;
    private float timeSinceLastDamage;
    public bool isRegenerating = false;
    public GameObject PanelGameOver;
    public GameObject ScoreGameOver;
    public bool isDamaging = false;
    private Coroutine regenerationCoroutine;
```

```
    public GameObject soundHealthPrefab;
    public TMP_Text crystalText;
    private float crystal;
    //Reclams
    public InterAd interAd;
    private int tryCount;

    private void Awake()
    {
        health = maxHealth;
    }

    void Start () {
        spriteRend = GetComponent<SpriteRenderer>();
        matBlink = Resources.Load("EnemyBlink",
typeof(Material)) as Material;
        matDefault = spriteRend.material;
        explosion = Resources.Load("explosion");
        GameObject playerObject =
GameObject.FindGameObjectWithTag("Player");
        potions = playerObject.GetComponent<BuyPotions>();
        CameraPlayer cameraPlayer =
FindObjectOfType<CameraPlayer>();
        anim = GetComponent<Animator>();
        score = playerObject.GetComponent<PlayerController>();
        tryCount = PlayerPrefs.GetInt("tryCount");
    }

    void Update()
    {
        crystalText.SetText(" " + crystal.ToString());
    }

    void FixedUpdate()
    {
        if(rechange <=0)
        {
            ButtonSetHealth.interactable = true;
        }
        else
        {
            ButtonSetHealth.interactable = false;
            rechange -= Time.deltaTime;
        }

        // Обновляємо час без шкоди
        if (timeSinceLastDamage < 10f && health < maxHealth &&
isDamaging)
        {
            StopRegeneration();

           timeSinceLastDamage += Time.deltaTime;
```

```
            if (timeSinceLastDamage>=10f)
            {
             isRegenerating = true;
             isDamaging = false;
            }

        }
        else if (timeSinceLastDamage >= 10f && health <
maxHealth && !isDamaging)
        {
            if (isRegenerating)
            {
                StartRegeneration();
                 // Встановлюємо тут, щоб запобігти запуску
декількох корутин
            }
        }
    }
    void StartRegeneration()
    {
        // Перевіряємо, чи регенерація вже не відбувається
        if (regenerationCoroutine == null)
        {
            // Починаємо регенерацію і зберігаємо посилання на
корутину
            regenerationCoroutine =
StartCoroutine(RegenerateHealth());
        }
    }

    void StopRegeneration()
    {
        // Перевіряємо, чи регенерація відбувається і чи існує
корутин
        if (regenerationCoroutine != null)
        {
            StopCoroutine(regenerationCoroutine); // Зупиняємо
корутину
            regenerationCoroutine = null; // Обнуляємо змінну,
щоб позначити, що регенерація була зупинена
        }
    }

    IEnumerator RegenerateHealth()
{
    while (health < maxHealth)
    {
        health += 1;
        anim.Play("regenerateHealth");
        if (health > maxHealth)
        {
            health = maxHealth;
```

```
        }
        healthbar.UpdateHealthbar(maxHealth, health);
        yield return new WaitForSeconds(4f);
    }
    isRegenerating = false;
}

    public void TakeHit(int damage)
    {
        health -= damage;
        isDamaging = true;
        isRegenerating = false;
        timeSinceLastDamage = 0f;
        Invoke("CallStartShake", 0.3f);
        if (health <= 0)
        {
            tryCount ++;
            PlayerPrefs.SetInt("tryCount", tryCount);
            if(tryCount % 3 == 0)
            {
                interAd.ShowAd();
            }
            crystal =
(float)Math.Round(score.playertotalScore/100f);
            Purchase.purchaseCoin += crystal;
            Purchase.SavePurchase();
            GameObject explosionRef =
(GameObject)Instantiate(explosion);
            explosionRef.transform.position = new
Vector3(transform.position.x, transform.position.y - 1,
transform.position.z);
            health = 0;
            Destroy(gameObject, 0.3f);
            Destroy(explosionRef, 0.5f);
            score.bestScore();
            PanelGameOver.SetActive(true);
            ScoreGameOver.SetActive(true);
        }
        else
        {
            StartCoroutine(BlinkCoroutine());
        }
        healthbar.UpdateHealthbar(maxHealth, health);

    }

    void CallStartShake()
    {
        if (cameraPlayer != null)
        {
            cameraPlayer.StartShake();
        }
```

```
        else
        {
            Debug.LogError("CameraPlayer not found!");
        }
    }

    public void SetHealth(int bonusHealth)
    {
        if(potions.pointHealth > 0)
        {
            potions.pointHealth -=1;
        if(rechange <=0)
        {
            GameObject soundHealth =
Instantiate(soundHealthPrefab, transform.position,
Quaternion.identity);
            Destroy(soundHealth, 1f);
            anim.Play("Healthing");
            health += bonusHealth;
            if (health > maxHealth)
            {
                health = maxHealth;
            }
            healthbar.UpdateHealthbar(maxHealth, health);
            rechange = startRechange;
        }
        potions.SavePoints();
        }
    }

   IEnumerator BlinkCoroutine()
    {
        yield return new WaitForSeconds(0.1f);
        spriteRend.material = matBlink;
        yield return new WaitForSeconds(0.12f);
        spriteRend.material = matDefault;
    }
}
```

Файл Enemy.cs

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

public class Enemy : MonoBehaviour
{
    [SerializeField] public int maxHealth;
    private Animator anim;
    private Rigidbody2D rb;
    public float speed;
    private Transform target;
```

```
    public float stop;
    public GameObject player;
    public bool flip;
    private float rechange;
    public float startRechange;

    [Header("Урон")]
    public Transform attackPos;
    public LayerMask playerMask;
    public float radius;
    public int damage;
    private bool facingRight = true;
    public GameObject soundAttackPrefab;
    private AudioSource audioSource;
    public float jumpForce = 10f;
    public LayerMask obstacleLayer;

    void Start () {
        audioSource = GetComponent<AudioSource>();
        anim = GetComponent<Animator>();
        rb = GetComponent<Rigidbody2D>();
        target =
GameObject.FindGameObjectWithTag("Player").GetComponent<Transfor
m>();

    }

    void FixedUpdate()
    {
        Attack();
        if(facingRight == false && target.transform.position.x >
transform.position.x)
        {
            Flip();
        }
        else if(facingRight == true &&
target.transform.position.x < transform.position.x)
        {
            Flip();
        }
        if(Vector2.Distance(transform.position,
target.position)>stop)
        {
            anim.SetBool("isRunning",true);
            if (!audioSource.isPlaying)
            {
                audioSource.Play();
            }
            transform.position =
Vector2.MoveTowards(transform.position, target.position, speed *
Time.deltaTime);
        }
```

```
        else
        {
            if (audioSource.isPlaying)
            {
                audioSource.Stop();
            }
            anim.SetBool("isRunning",false);
        }
    }

    public void Flip()
    {

        facingRight =!facingRight;
        Vector3 Scaler = transform.localScale;
        Scaler.x *= -1;
        transform.localScale=Scaler;
    }

    public void Attack()
    {
        anim.SetBool("isRunning",false);

         if(rechange <=0)
        {

            Collider2D[] players =
Physics2D.OverlapCircleAll(attackPos.position, radius,
playerMask);
            for (int i=0; i < players.Length; i++)
            {
                anim.SetTrigger("Attack");
                Invoke("SoundAttack", 0.2f);

players[i].GetComponent<Health>().TakeHit(damage);
            }
            rechange = startRechange;
        }
        else
        {
            rechange -= Time.deltaTime;
        }
    }

    void SoundAttack()
    {
        GameObject soundAttack = Instantiate(soundAttackPrefab,
transform.position, Quaternion.identity);
        Destroy(soundAttack, 1f);
    }

    private void OnDrawGizmosSelected()
```

```
    {
        Gizmos.color = Color.black;
        Gizmos.DrawWireSphere (attackPos.position, radius);
    }

}
```

Файл Plant.cs

```
using System.Collections;
using UnityEngine;

public class PlantController : MonoBehaviour
{
    public Sprite[] growthSprites;
    private float totalGrowthTime = 120f;
    private float growthRate;
    private float currentGrowth = 0f;
    private int currentSpriteIndex = 0;
    public bool isMature = false;

    void Start()
    {
        growthRate = growthSprites.Length / totalGrowthTime;
    }

    void Update()
    {
        if (!isMature && currentSpriteIndex <
growthSprites.Length)
        {
            currentGrowth += growthRate * Time.deltaTime;
            // Перевіряємо, чи не досягла рослина максимальної
стадії зростання
            if (currentGrowth >= growthSprites.Length)
                {
                // Якщо досягло, встановлюємо поточний індекс
спрайту у максимальне значення
                currentSpriteIndex = growthSprites.Length - 1;
                UpdatePlantVisuals();
                isMature = true;  // Рослина дозріла
                }

            if (currentGrowth >= (currentSpriteIndex + 1) &&
currentSpriteIndex < growthSprites.Length - 1)
            {
                // Якщо досягло наступної стадії зростання,
перемикаємось на наступний спрайт і оновлюємо візуальне подання
                currentSpriteIndex++;
                UpdatePlantVisuals();

            }
```

```
        }
    }

    void UpdatePlantVisuals()
    {
        if (currentSpriteIndex < growthSprites.Length)
        {
            GetComponent<SpriteRenderer>().sprite =
growthSprites[currentSpriteIndex];
        }
    }
}
```

Файл GranadeLauncher.cs

```
using UnityEngine;
using TMPro;

public class GranadeLauncher : MonoBehaviour
{
    private const string bombKey = "bomb";
    public TMP_Text bombText;
    public int numBomb = 3;
    public GameObject grenadePrefab;
    public float maxDragDistance = 5f;
    public float launchForce = 10f;
    private Camera mainCamera;
    private bool isDragging = false;
    private Vector3 dragStartPosition;
    public PlayerController playerController;

    void Start()
    {
        numBomb = Purchase.purchaseBomb + 1;
        SaveBomb();
        LoadBomb();
        mainCamera = Camera.main;
        GameObject playerObject =
GameObject.FindGameObjectWithTag("Player");
        playerController =
playerObject.GetComponent<PlayerController>();
    }

    void Update()
    {
        UpdateBombText();
        if(numBomb>0)
        {
        if (Input.touchCount > 0)
        {
            Touch touch = Input.GetTouch(0);
            Vector3 touchPosition =
mainCamera.ScreenToWorldPoint(touch.position);
```

```
            touchPosition.z = 0f;

            switch (touch.phase)
            {
                case TouchPhase.Began:
                    if (IsTouchOnPlayer(touchPosition))
                    {
                        isDragging = true;
                        dragStartPosition = touchPosition;
                    }
                    break;
                case TouchPhase.Ended:
                    if (isDragging)
                    {
                        isDragging = false;
                        LaunchGrenade(dragStartPosition,
touchPosition);
                        numBomb -= 1;
                        SaveBomb();
                    }
                    break;
            }
        }
    }
    }

    bool IsTouchOnPlayer(Vector3 touchPosition)
    {
        Collider2D playerCollider = GetComponent<Collider2D>();
        return playerCollider.bounds.Contains(touchPosition);
    }

    void LaunchGrenade(Vector3 startPosition, Vector3
endPosition)
    {
        float dragDistance =
Mathf.Min(Vector3.Distance(startPosition, endPosition),
maxDragDistance);
        Vector3 launchDirection = (startPosition -
endPosition).normalized;
        float launchForceMultiplier = Mathf.Clamp01(dragDistance
/ maxDragDistance);

        GameObject grenade = Instantiate(grenadePrefab,
transform.position, Quaternion.identity);
        Rigidbody2D rb = grenade.GetComponent<Rigidbody2D>();
        if (rb != null)
        {
            rb.AddForce(launchDirection * launchForce *
launchForceMultiplier, ForceMode2D.Impulse);
        }
    }
```

```
    public void UpdateBombText(){
        bombText.SetText(" " + numBomb.ToString());
    }

    public void BuyBomb()
    {
        if(playerController.playerCoins >=50)
        {
            numBomb += 1;
            playerController.playerCoins -=50;
            playerController.SaveResourse();
            SaveBomb();
            playerController.UpdateResourseText();
            UpdateBombText();
        }
    }

    void LoadBomb()
    {
        if (PlayerPrefs.HasKey(bombKey))
        {
            numBomb = PlayerPrefs.GetInt(bombKey);
        }
        else
        {
            numBomb = 2;
        }
    }

    public void SaveBomb()
    {
        PlayerPrefs.SetInt(bombKey, numBomb);
        PlayerPrefs.Save();
    }
    void OnApplicationQuit()
    {
        SaveBomb();
    }
}
```