МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«МОБІЛЬНИЙ ІГРОВИЙ ЗАСТОСУНОК ЗАСОБАМИ UNITY»

Рівень «бакалавр»
Галузь знань 12 «Інформаційні технології»,
спеціальність 121 «Інженерія програмного забезпечення»
Виконав здобувач 4 курсу
Чепендюк Олександр Олександрович

Керівник к.т.н., доцент
Манаков Сергій Юрійович

Рецензент к.т.н., доцент
Задерейко Олександр Владиславович

Одеса – 2024

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
Факультет кібербезпеки та інформаційних технологій
Кафедра інформаційних технологій
Галузь знань: 12 Інформаційні технології
Спеціальність: 121 Інженерія програмного забезпечення
Назва освітньої програми: Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри інформаційних технологій
доцент Н.І. Логінова _____
«____ »_____ 20__ року

З А В Д А Н Н Я
на кваліфікаційну (бакалаврську) роботу
здобувачу Чепендюку Олександру Олександровичу

1. Тема роботи: «Мобільний ігровий застосунок засобами Unity»
Керівник роботи: к.т.н, доцент Манаков Сергій Юрійович
затверджені наказом НУ «ОЮА» № 809-06 від 02 квітня 2024 року
2. Дата видачі завдання 15 вересня 2024 року
3. Строк подання здобувачем роботи 20 травня 2024 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної області та вибір засобів розробки	28.10.2024	
2	Моделювання та проєктуванняя гри	03.04.2024	
3	Розробка гри «Rabbit Run»	27.04.2024	
4	Тестування та оцінка якості ігрового застосунку	05.05.2024	
5	Оформлення звітної частини	19.05.2024	

Здобувач _____
(підпис) (прізвище та ініціали)
Керівник роботи _____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Чепендюк О. О. Мобільний ігровий застосунок засобами Unity. – Кваліфікаційна робота бакалавра за спеціальністю 121 «Інженерія програмного забезпечення», освітньою програмою «Інженерія програмного забезпечення».

Кваліфікаційна робота викладена на 48 сторінці основного тексту і 64 сторінках загального тексту, містить 4 розділи, 25 рисунків, 3 таблиці, 11 використаних джерел.

Метою роботи є розробка мобільного ігрового застосунку у жанрі раннер з використанням технологій Unity.

Об’єктом дослідження є специфіка розробки ігрового застосунку та організація його інтерфейсу та функціоналу.

Предмет дослідження – розробка та вдосконалення механіки геймплею мобільної гри через інтеграцію різноманітних функцій та ефектів у середовищі Unity.

У кваліфікаційній роботі розроблено мобільний ігровий застосунок у жанрі раннер з використанням технологій Unity, який включає в себе меню налаштувань для користувача та можливість збереження рекордного результату гравця. Досліджено специфіку розробки застосунків, в даному випадку ігор. за допомогою Unity.

Розроблена гра може використовуватись на будь яких мобільних пристроях із операційною системою Android.

У першому розділі проводиться аналіз предметної області та проводиться вибір засобів розробки. У другому розділі здійснено проєктування гри. У третьому розділі пояснюється розробка гри. У четвертому – тестування гри.

За результатами роботи розроблено висновки щодо виконаних та поставлених задач.

Ключові слова: Unity, префаб, MagicaVoxel, скрипт, post-processing.

ABSTRACT

Chependiuk O. O. Mobile Game Application Using Unity. – Bachelor's Qualification Work in the specialty 121 "Software Engineering", educational program "Software Engineering".

The qualification work is presented on 48 pages of main text and 64 pages of total text, contains 4 sections, 25 illustrations, 3 tables, and 11 used sources.

The aim of the work is to develop a mobile game application in the runner genre using Unity technologies.

The object of the research is the specifics of developing a game application and the organization of its interface and functionality.

The subject of the research is the development and improvement of mobile game gameplay mechanics through the integration of various features and effects in the Unity environment.

In the qualification work, a mobile game application in the runner genre is developed using Unity technologies, which includes a user settings menu and the ability to save the player's high score. The specifics of application development, particularly games, using Unity have been studied.

The developed game can be used on any mobile devices with the Android operating system.

In the first section, an analysis of the subject area is conducted, and the development tools are selected. In the second section, the game is designed. The third section explains the development of the game. The fourth section covers game testing.

Based on the results of the work, conclusions are drawn regarding the completed and set tasks.

Keywords: Unity, prefab, MagicaVoxel, script, post-processing.

ЗМІСТ

ВСТУП

Мобільні ігри відрізняються особливою привабливістю через свою доступність: мобільний пристрій завжди під рукою, що дає можливість грати в ігри, навіть під час очікування у черзі, перерви на роботі або під час подорожі.

У цьому контексті розробка мобільного ігрового застосунку в жанрі раннер може не лише зацікавити розробників, а і стати практичною необхідністю для задоволення потреб користувачів.

Мета роботи – розробка мобільного ігрового застосунку у жанрі раннер з використанням технологій Unity.

Об'єкт дослідження – специфіка розробки ігрового застосунку та організація його інтерфейсу та функціоналу.

Предмет дослідження – розробка і вдосконалення механіки геймплею мобільної гри через інтеграцію різних функцій та ефектів у середовищі Unity.

Відповідно до поставленої мети, у рамках роботи необхідно зробити аналіз предметної області, моделювання та проєктування, зробити та описати різні аспекти розробки ігрового продукту, а також, в кінці треба провести тестування, яке покаже чи гра працює правильно.

Під час роботи над проєктом вивчалися можливості розробки ігор для мобільних платформ. Був проведений детальний аналіз та вивчення різних інструментів і технологій, що застосовуються в цій галузі. Це дозволило визначити оптимальний підхід до створення ігрового застосунку у жанрі раннер, який би відповідав сучасним вимогам мобільного геймінгу.

Кваліфікаційна робота спрямована на використання можливостей Unity для створення мобільного ігрового застосунку у жанрі раннер. В ній розглянуто широкий спектр елементів розробки ігрового продукту, включаючи створення анімацій, додавання та налаштування звуків, роботу з графікою та інших технічних аспектів.

Публікації автора роботи:

Трофименко О.Г., Чепендюк О.О. Чому C++? *Кібербезпека в сучасному світі: актуальні виклики* : матер. III всеукр. наук.-практ. конф. (19 листопада 2021 р.). Одеса, 2021. С. 105–110. DOI 10.32837/11300.15973

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИБІР ЗАСОБІВ РОЗРОБКИ

1.1 Специфіка розробки ігрових застосунків

Можна сказати, що в сучасному світі розвиток мобільних ігор здійснюється зі швидкістю світла, завдяки постійному удосконаленню технологій розробки та розширенню можливостей інструментів для створення ігрового контенту. Це зумовлено доступністю і швидким розвитком мобільних пристроїв. За даними Statista, до 2027 року кількість мобільних геймерів зросте з 2,22 мільярдів до 2,32 мільярдів. До речі, ця цифра перевищує населення Китаю, яке становить 1,43 мільярди осіб.

У 2022 році користувачі завантажили близько 89,74 мільярдів мобільних ігор і витратили на мобільні додатки в середньому 5,3 години. Однак, згідно з дослідженням AppLovin, 85% гравців мобільних ігор все ще не вважають себе мобільними геймерами [1].

Розробка ігрових застосунків – це комплексний процес, який включає ряд кроків та аспектів, починаючи від визначення концепції і закінчуючи тестуванням і випуском готового продукту.

Ця гра була виконана в жанрі раннер. Раннери – популярний жанр відеоігор, в якому гравець керує персонажем або об'єктом, щоб продовжувати рухатися вперед через середовище, наповнене перешкодами. Мета – подолати перешкоди, набрати високий бал або виконати певне завдання.

У іграх-раннерах персонажі та об'єкти рухаються вперед автоматично, а гравець повинен стрибати, ковзати та ухилятися, щоб подолати перешкоди та небезпеки. Гра стає дедалі складнішою зі збільшенням швидкості та появою нових перешкод. Метою часто є вижити якомога довше, набрати найбільшу кількість балів або досягти певної віхи. Ігри-раннери часто мають нескінченний цикл, де персонаж продовжує бігти, поки не натрапить на перешкоду або не впаде з екрана [2].

Для розробки мобільного ігрового застосунку у жанрі раннер, необхідно ретельно розглянути кожен етап процесу та врахувати специфіку цього жанру.

Першим і одним з найважливіших кроків є визначення концепції та механіки гри. Так як у жанрі раннер характерними елементами є постійний рух головного персонажу вперед, швидкий темп гри та уникання перешкод. Ці аспекти визначають основні принципи геймплею та механіку нашого ігрового застосунку.

На наступному етапі слід звернути спрочаткуувагу на технічні аспекти розробки. Вибір правильних інструментів для розробки мобільних ігор є ключовим кроком. Уданому випадку було вибрано Unity, завдяки своїй потужності та дружньому інтерфейсу, є одним з найкращих варіантів для створення ігор для мобільних платформ.

Окрім технічних аспектів, слід приділити увагу естетичним та аудіовізуальним аспектам ігрового застосунку. Висока якість графіки, анімації та звукового супроводу стануть важливими елементами, які допоможуть створити неповторний ігровий досвід та забезпечити гравцям насолоду від гри.

1.2 Аналіз наявних аналогів

Перед початком розробки мобільного раннера важливо проаналізувати існуючі на ринку аналоги. Це допоможе краще зрозуміти тенденції у цьому жанрі та виявити сильні та слабкі сторони конкурентів, що може стати джерелом цінних відкриттів які допоможуть в процесі розробки власної гри.

Слід зазначити що ігор в жанрі раннер є дуже багато. Ось декілька із них:

1. «Temple Run» – це захоплююча гра, в якій гравець управляє персонажем, який біжить по стародавньому храмі, уникаючи перешкод і збираючи скарби. Випущена 4 серпня 2011 року. Доступна на платформах iOS, Android і Windows Phone;

2. «Subway Surfers» – це ще одна популярна гра, в якій гравці біжать від поліції по вулицях міста, уникаючи перешкод і збираючи монети. Доступна на платформах iOS, Android і Windows Phone. Існує версія для комп'ютерів. Розроблена данськими компаніями Kiloo і SYBO Games, одна з найпопулярніших мобільних ігор у світі. Випущена 23 травня 2012 року;

3. «Minion Rush» – це гра, створена на основі популярного анімаційного бренду "Міньйони", де гравці мають можливість бігати та збирати банани. Гра приваблює гравців своїми різноманітними рівнями та приємним геймплеєм. доступна на Microsoft Windows, iOS та Android. Випущена 10 червня 2013 року;

4. «Sonic Dash» – це гра 2013 року, розроблена Hardlight і видана японською ігровою студією Sega. участю відомого персонажа Sonic the Hedgehog. Гравці керують Соніком, який безперервно біжить через різноманітні рівні, уникаючи перешкод і збираючи кільця;

5. «Talking Tom Gold Run» – гра розроблена студією Outfit7. Вона була випущена у 2016 році і стала швидко популярною серед користувачів мобільних пристроїв. Під час гри гравці керують котом Томом, який біжить за злодіями, збираючи золоті монети на своєму шляху та оминаючи перешкоди.

Таблиця 1.1 – SWOT-аналіз конкурентів у формі таблиці

Програмний продукт	**Сильні сторони**	**Слабкі сторони**	**Можливості**	**Загрози**
Temple Run	- Захоплюючий геймплей; - Популярність серед гравців.	- Монотонність у геймплеї; - Застарілий дизайн і графіка.	- Вдосконалення механіки гри.	- Поява конкурентів з новими ідеями.
Subway Surfers	- Висока популярність; - Колоритна графіка; - Постійні оновлення.	- Потреба у постійному оновленні контенту	- Впровадження нових героїв та механік гри.	- Збільшення конкуренції на ринку.
Minion Rush	- Популярний бренд «Міньйони»; - Різноманітність рівнів.	- Наявність реклами.	- Розширення магазину.	- Поява конкурентів.
Talking Tom Gold Run	- Вже відомий в індустрії головний герой; - Ігровий процес, що розвивається.	- Велика Кількість реклами.	- Впровадження нових локацій та завдань; - Розширення асортименту головоломок.	- Зміна інтересів гравців.

Опираючись на наведену вище інформацію, можна зробити висновок, що всі вказані конкуренти у жанрі раннер дуже схожі між собою. Наприклад, багато з них використовують захоплюючий геймплей, високоякісну графіку та можливості

для монетизації гри. Також, багато з них ставлять на популярні бренди або персонажів, що допомагає привернути увагу гравців.

Однак, не зважаючи на схожість, кожен з них має свої унікальні особливості, які можуть привернути свою аудиторію. Наприклад, «Temple Run» відомий своїм захоплюючим геймплеєм, «Subway Surfers» – колоритною графікою та широкою географією, «Minion Rush» – популярним брендом «Міньйони», а «Talking Tom Gold Run» – своїми веселими героями та інноваційним підходом до геймплею.

Також ураховуючи попит на оновлені та захоплюючі ігри можна зробити висновок з того що мобільні ігри все ще користуються великою популярністю і хоча ігор у жанрі раннер дуже багато, все одно розробка такого продукту є доцільною, оскільки вони легко можуть знайти свою аудиторію серед широкого кола гравців.

1.3 Розробка функціональних вимог та виявлення варіантів використання створюваного програмного продукту

Одним з найважливіших етапів у розробці будь-якого програмного продукту є сформулювання вимог до його функціоналу. Це необхідно для того, щоб чітко визначити, які можливості повинен мати продукт, щоб відповідати потребам майбутніх користувачів і цілям проєкту. На цьому етапі вирішується, якими будуть функції продукту, які дії може здійснювати користувач і як буде взаємодіяти власник сайту або його працівники з програмним забезпеченням.

Перш ніж розпочати аналіз, важливо визначити мету проєкту. У даному випадку, це розробка мобільного ігрового застосунку у жанрі раннер з використанням технологій Unity.

Також для розробки програмного продукту, необхідно визначити функціональні вимоги та можливі варіанти використання створюваного програмного продукту. Це дозволить чітко визначити цілі проєкту та спрямувати розробку у правильному напрямку.

Функціональні вимоги (англ. Functional Requirements), представляють собою основні функції або можливості, які повинна мати програма, у нашому випадку гра, щоб відповідати своєму призначенню. Простіше кажучи, ці вимоги визначають, що повинна робити програма. Вони описують взаємодію між програмним забезпеченням і користувачем та поведінку програмного забезпечення за різних умов.

Функціональні вимоги зазвичай мають такі характеристики:

- Специфіка: вони є детальними та конкретними, з невеликою кількістю двозначностей або взагалі без них. Вони окреслюють точні функції, входи та виходи системи;
- Можливість перевірки: функціональні вимоги можна протестувати і перевірити, щоб переконатися, що програмне забезпечення працює за призначенням;
- Орієнтованість на користувача: тісно пов'язана з потребами та очікуваннями користувачів і гарантує, що програмне забезпечення виконує своє призначення;
- Змінність: функціональні вимоги можуть бути змінені впродовж проєкту у відповідь на відгуки користувачів та мінливі бізнес-потреби [4].

Також вимоги повинні враховувати технічні аспекти розробки, такі як масштабованість, безпека та продуктивність, а також під час опису виявлених вимог, важливо врахувати потреби користувачів та інших зацікавлених сторін. Це допоможе створити продукт, який буде оптимально відповідати потребам всіх його користувачів.

Отже, вимоги до програмного продукту визначаються його цілями, потребами користувачів та технічними можливостями.

Таким чином, розробка функціональних вимог дозволить нам чітко визначити, які функції повинен мати наш програмний продукт і які можливості ми повинні забезпечити для користувачів.

В даній роботі, є такі функціональні вимоги:

Перелік функціональних вимог для мобільного раннера в жанрі раннер може включати:

1) Керування головним персонажем:
   а) Можливість переміщення персонажа вправо та вліво за допомогою нахилу пристрою;
   б) Можливість стрибка за допомогою натискання на екран.
2) Геймплей:
   а) Генерація нескінченного рівня із різними перешкодами які спавняться у випадковому порядку і у випадкових місцях;
   б) Генерація мирних персонажів які спавняться у випадковому порядку і також бігтимуть разом із головним персонажем оминаючи перешкоди. У разі зіткнення їх із головним персонажем вони повинні відстрибнути і видати спеціальних звук;
   в) Завершення гри через зіткнення із перешкодою;
   г) У випадку якщо відбудеться якась помилка і головний персонаж провалиться під карту повинно відкритись спеціальне меню, яке сповістить про це і надасть можливість почати гру спочатку;
3) Головне меню:
   а) Наявність головного меню яке надає доступ до меню налаштувань, надає можливість вийми із гри та почати її;
4) Система досягнень:
   а) Введення системи підрахунку і збереження рекорду максимальної відстані яку пробіг головний персонаж. Рекорд повинен зберігатися навіть після закриття гри;
   б) Виведення рекорду максимальної відстані яку пробіг головний персонаж в головне меню;
5) Меню налаштувань:
   а) Наявність меню налаштувань в якому можна налаштувати гучність та обнулити значення рекорду максимальної відстані яку пробіг головний персонаж.

1.4 Вибір засобів розробки

Для створення мобільних ігор розробники широко використовують різноманітні інструменти, одним з яких є Unity – одна з найпопулярніших та потужних платформ для розробки ігор та інтерактивних 2D і 3D додатків. Він надає все, що потрібно розробникам для створення ігор [2]. Саме тому для виконання даної роботи було обрано Unity.

Для створення 3D графіки в грі було використано програму MagicaVoxel, адже вона є простою, зручною і безкоштовною.

1.4.1 Unity для розробки гри

У цьому підрозділі буде обґрунтовано доцільність використання програмних засобів розробки які було використано під час виконання цієї роботи, зокрема Unity та Magica Voxel.

Для розробки гри було обрано Unity. Unity – це потужний і популярний кросплатформний рушій для розробки ігор та додатків. Він пропонує програмістам і дизайнерам можливість створювати інтерактивні проєкти для комп'ютерів, мобільних пристроїв, ігрових консолей та інших платформ. Відомий своєю гнучкістю, широким набором інструментів і підтримкою різноманітних технологій, Unity є ідеальним вибором для початківців і досвідчених розробників. Це ідеальний вибір як для початківців, так і для досвідчених розробників.

Unity має чіткі переваги, які роблять його вибором багатьох розробників. По-перше, він має інтуїтивно зрозумілий інтерфейс, що робить процес розробки більш доступним для програмістів-початківців. По-друге, Unity має потужний графічний рушій, який дозволяє створювати приголомшливі візуальні ефекти та анімацію. Крім того, Unity має широку підтримку плагінів та активну спільноту, яка готова ділитися досвідом та ресурсами.

Однак Unity має деякі недоліки. По-перше, деякі проєкти можуть вимагати більш просунутого програмування, особливо при створенні складних ігрових

механік. Крім того, Unity може бути вимогливим до ресурсів комп'ютера, особливо під час роботи над великими проєктами.

Unity має інтуїтивно зрозумілий інтерфейс, що складається з різних вікон та панелей. Основними компонентами інтерфейсу є редактор сцен, де можна створювати та редагувати оточення гри або програми, та інспектор, та можна встановлювати властивості об'єктів та компонентів. Існує також бібліотека ресурсів, де можна зберігати текстури, моделі та інші ресурси проєкту і керувати ними. Ігрові скрипти написані мовою програмування C# і використовуються для додавання логіки та функціональності до проєкту.

Для роботи з Unity потрібні базові знання програмування, зокрема мови C#. Рекомендується ознайомитися з основами об'єктно-орієнтованого програмування (ООП) і патернів розробки ігор. Також важливо вивчити основні поняття розробки ігор, такі як анімація, фізика і механіка гри. Unity має велику документацію, відеоуроки та керівництва, які допоможуть оволодіти цими знаннями.

Крім того, треба завантажити та встановити Unity з офіційного вебсайту, а також мати доступ до комп'ютера з достатніми ресурсами для роботи з графічними програмами [5].

Unity є потужним інструментом для створення ігор і додатків, який надає розробникам широкий спектр можливостей. Він дозволяє створювати як 2D, так і 3D проєкти, додаючи анімацію, фізику, звукові ефекти та інші елементи. На відміну від інших рушіїв, Unity також підтримує віртуальну та доповнену реальність, що дозволяє створювати захопливі проєкти для VR– і AR-пристроїв. Unity підтримує різні платформи, такі як Windows, macOS, Android і iOS, що робить його універсальним інструментом для розробки.

Unity також широко використовується в інших галузях, таких як освіта, архітектура та медицина. Він дозволяє створювати візуалізації, тренувальні симулятори, віртуальні тури та інші застосунки. Зручні інструменти Unity спрощують роботу з асетами, дозволяючи легко імпортувати та експортувати

ресурси без додаткових зусиль. Оптимізація для різних платформ є ключовою функцією, що дозволяє максимально ефективно використовувати ресурси.

Хоча Unity є одним з лідерів у своєму сегменті, існують інші альтернативи, такі як Unreal Engine, Cocos2d, Godot і Amazon Lumberyard. Вибір залежить від потреб і уподобань розробника.

Узагальнюючи, Unity – це інструмент з великим потенціалом та можливостями для творчості і інновацій. Вивчаючи його можливості та експериментуючи, можна створювати дивовижні проєкти у світі розробки ігор і додатків. Саме тому його було вибрано для виконання цієї роботи.

1.4.2 MagicaVoxel для створення воксельних моделей

Всю графіку в грі було вирішено зробити воксельною, так як її легко робити і вона не застаріває.

Воксель – це тривимірний аналог пікселя. Це кубики із заданими властивостями, такими як колір, прозорість і текстура. Вокселі використовуються для створення зображень і моделей.

Структура вокселя визначається його розміром і властивостями. Вокселі мають три виміри: висоту, ширину і глибину. Ці розміри також визначають роздільну здатність, тобто рівень деталізації зображення або моделі.

Воксельні зображення створюються шляхом рендерингу вокселів відповідно до їхніх властивостей. Для цього використовується алгоритм рендерингу, який визначає, як колір і прозорість впливають на кінцеве зображення.

Воксельні моделі створюються шляхом об'єднання вокселів у тривимірну решітку. Розмір решітки визначає рівень деталізації об'єкта.

Воксельні моделі можуть бути статичними або динамічними. Статичні моделі не змінюються з часом, тоді як динамічні моделі можуть змінювати форму, розмір і положення.

Перші воксельні зображення базувалися на технології 2.5D-графіки і використовувалися для створення псевдо-тривимірних ефектів. Однак, на відміну

від воксельної графіки, 2.5D-графіка не створювала реального тривимірного об'єму, а лише ілюзію глибини. Спочатку така графіка використовувалася переважно в науковій сфері, наприклад, для візуалізації медичних даних.

У 1980-х роках воксельну графіку почали використовувати у відеоіграх, починаючи з Castle Wolfenstein у 1981 році. Незважаючи на ранній успіх, воксельна графіка все ще була менш популярною, ніж полігональна графіка в 1990-х роках.

В останні роки вона знову привернула увагу розробників і дизайнерів завдяки розвитку технологій, які дозволяють створювати більш складні та деталізовані моделі [6].

Для створення воксельних моделей було обрано Magica Voxel.

MagicaVoxel – це програма для створення 3D воксельної графіки, вона дозволяє розробникам створювати складні об'єкти та сцени шляхом додавання, видалення та модифікації вокселів. Програма має інтуїтивний і легкий у використанні інтерфейс, що робить її доступною для користувачів з будь-яким рівнем навичок, до того ж вона є безкоштовною.

Основні функції MagicaVoxel включають:

- Створення і редагування воксельних моделей;
- Зміна розміру, форми та текстур векселів;
- Малювання текстур та накладання їх на об'єкти;
- Експорт моделей у різноманітні формати, такі як OBJ, STL, PNG тощо.

MagicaVoxel широко використовується для розробки ігор, анімацій, візуалізацій та інших проєктів, де потрібна воксельна графіка. Її простота та потужність роблять її популярним вибором серед розробників, що працюють у цьому стилі.

1.5 Маркетингові інструменти в створенні та просуванні програмного продукту

Перед початком розробки гри слід розробити карту клієнта та провести аналіз ринку

Карта клієнта – це інструмент, який дозволяє розробникам отримати глибоке розуміння своєї цільової аудиторії. Вона включає в себе детальний опис потенційних користувачів, їхніх потреб, проблем та переваг. За допомогою карти клієнта розробники можуть краще налаштувати продукт на потреби своїх користувачів та ефективно спрямовувати свої маркетингові зусилля.

Таблиця 1.2 – Карта клієнта для даного мобільного раннера

Категорія	**Опис**
Цільова аудиторія	- Вікова категорія: 13-35 років; - Мобільні геймери, активні користувачі мобільних пристроїв, любителі захоплюючих ігор;
Потреби та проблеми	- Потреба в захоплюючій грі для розваги та релаксації; - Проблема з відсутністю цікавих ігор з високою геймплейною якістю на мобільних платформах; - Потреба в грі з простим, але цікавим геймплеєм, який відповідає активному ритму життя;
Уподобання	- Уподобання до швидких та динамічних ігор. Популярність в середовищі друзів та онлайн-спільнот; - Уподобання до гіперкінетичних образів та кольорових ефектів;

Закінчення табл. 1.2

Очікування від гри	- Безкоштовний доступ з можливістю внутрішніх покупок; - Швидкий та динамічний геймплей; - Яскрава графіка та ефекти;Можливість конкурувати з іншими гравцями; - Високоякісний розважальний досвід на мобільному пристрої; - Різноманітні перешкоди та бонуси для підвищення цікавості гри; - Адреналін та емоційне занурення в гру.

Також слід зробити аналіз ринку. Цей аналіз включає в себе оцінку ринкового потенціалу, ідентифікацію конкурентів, визначення трендів та прогнозування майбутніх змін на ринку.

За останні кілька років спостерігається зростання числа мобільних гравців, що сприяє популярності ігор на цій платформі. Крім того, технологічний прогрес дозволяє створювати все більше захоплюючих інтерактивних ігор з високоякісною графікою та різноманітними ігровими механіками.

Серед конкурентів у жанрі раннер можна виділити описані вище популярні ігри, такі як "Temple Run", "Subway Surfers", "Minion Rush", "Sonic Dash", "Talking Tom Gold Run", які мають велику кількість завантажень та активних користувачів.

Ці ігри відомі своєю захоплюючою геймплейною механікою, яскравою графікою та постійними оновленнями, що дозволяють їм зберігати популярність серед гравців.

Аналізуючи ринок, стає зрозумілим, що для успішного виходу на нього необхідно розробити ігру з унікальними особливостями, які зможуть виділити її серед конкурентів. Також важливо враховувати вимоги та потреби цільової аудиторії, щоб забезпечити популярність та успіх гри на ринку мобільних ігор. Уданому випадку цими унікальними особливостями будуть:

- Пересування в сторони за допомогою нахилу телефона а не свайпами;
- Наявність інших позитивних персонажів, які ігтимуть разом із головним героєм;
- Наявність невеликого світу, складеного з двох маленьких островів, по якому можна вільно пересуватися.

1.6 Висновки до розділу

У данму розділі розглянуто ключові аспекти створення мобільного ігрового застосунку у жанрі раннер з використанням технології Unity. Були проаналізовані технічні можливості Unity, що дозволяють створювати якісні ігрові продукти для мобільних пристроїв. Крім того, розглянуто конкурентне середовище на ринку мобільних ігор у жанрі раннер, виявлено основних гравців та їхні особливості.

На основі проведеного аналізу можна зробити висновок, що розробка мобільного раннера є перспективною і має великий потенціал на ринку мобільних ігор. Важливою умовою успіху такого продукту буде унікальність та привабливість геймплею, а також врахування потреб цільової аудиторії.

2 МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ ГРИ

2.1 Створення діаграми прецедентів

У процесі проєктування програмного продукту для створення моделей, які допомагають візуалізувати його архітектуру, структуру та поведінку використовується UML (Unified Modeling Language).

Універсальна мова моделювання (Unified Modelling Language або UML) – це мова позначень або побудови діаграм, призначена для визначення, візуалізації і документування моделей зорієнтованих на об’єкти систем програмного забезпечення. UML не є методом розробки, іншими словами, у конструкціях цієї мови не повідомляється про те, що робити першим, а що останнім, і не надається інструкцій щодо побудови вашої системи, але ця мова допомагає вам наочно переглядати компонування системи і полегшує співпрацю з іншими її розробниками. Розробкою UML керує Object Management Group (OMG). Ця мова є загальноприйнятим стандартом графічного опису програмного забезпечення [7].

Процес проєктування програмного продукту з використанням UML включає створення різних типів діаграм, таких, наприклад, діаграми Use Cases для визначення функціональних вимог.

Use Cases (діаграми прецедентів, сценарій використання, діаграми варіантів використання) – дозволяють користувачеві уявити типи ролей та їхню взаємодію з системою. Однак вони не показують порядок виконання кроків [8].

Use Case діаграма для даної гри у жанрі раннер наведена на рис. 2.1. На діаграмі зображені різні взаємодії користувачів або системи з програмним продуктом (у цьому випадку – грою). Основні складові цієї діаграми включають акторів та юзкейси.

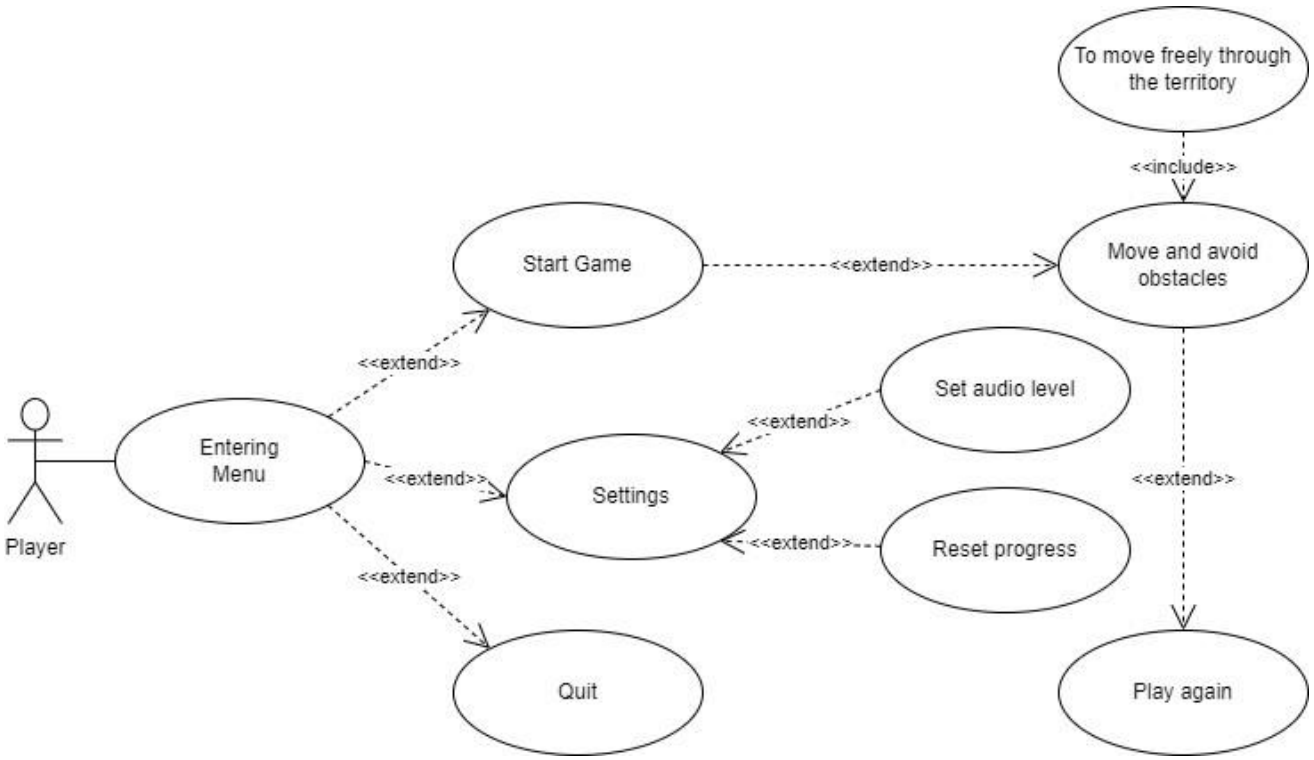

Рисунок 2.1 – Use Case діаграма

2.2 Створення діаграми класів

Для моделювання класів та їх взаємозв'язків використовують діаграми класів.

Діаграма класів – це один з основних видів діаграм у мові моделювання UML (Unified Modeling Language), яка використовується для візуалізації структури програмного продукту. Основна мета діаграми класів – це показати класи, їх атрибути, методи та зв'язки між ними.

Основні елементи діаграми класів:

- Класи: Представлені у вигляді прямокутників з трьома секціями: назва класу, атрибути та методи;
- Атрибути: Властивості класу, які визначають його стан або характеристики. Вони зазвичай вказуються під назвою класу у другій секції прямокутника;
- Методи: Операції або функції, які може виконувати клас. Вони також вказуються під назвою класу у третій секції прямокутника;

- Зв'язки між класами: Відображають взаємозв'язки між класами, такі як асоціації, агрегації, композиції, успадкування тощо. Зв'язки зображуються за допомогою стрілок між класами;
- Контекстні меню та інші елементи: Додаткові елементи, такі як контекстні меню, можуть бути використані для деталізації діаграми та показу різних аспектів класів.

Діаграма класів допомагає розуміти структуру програмного продукту та взаємозв'язки між його складовими частинами, що сприяє кращому розумінню коду та його подальшому розвитку.

Діаграма класів для даної гри у жанрі раннер виглядає так:

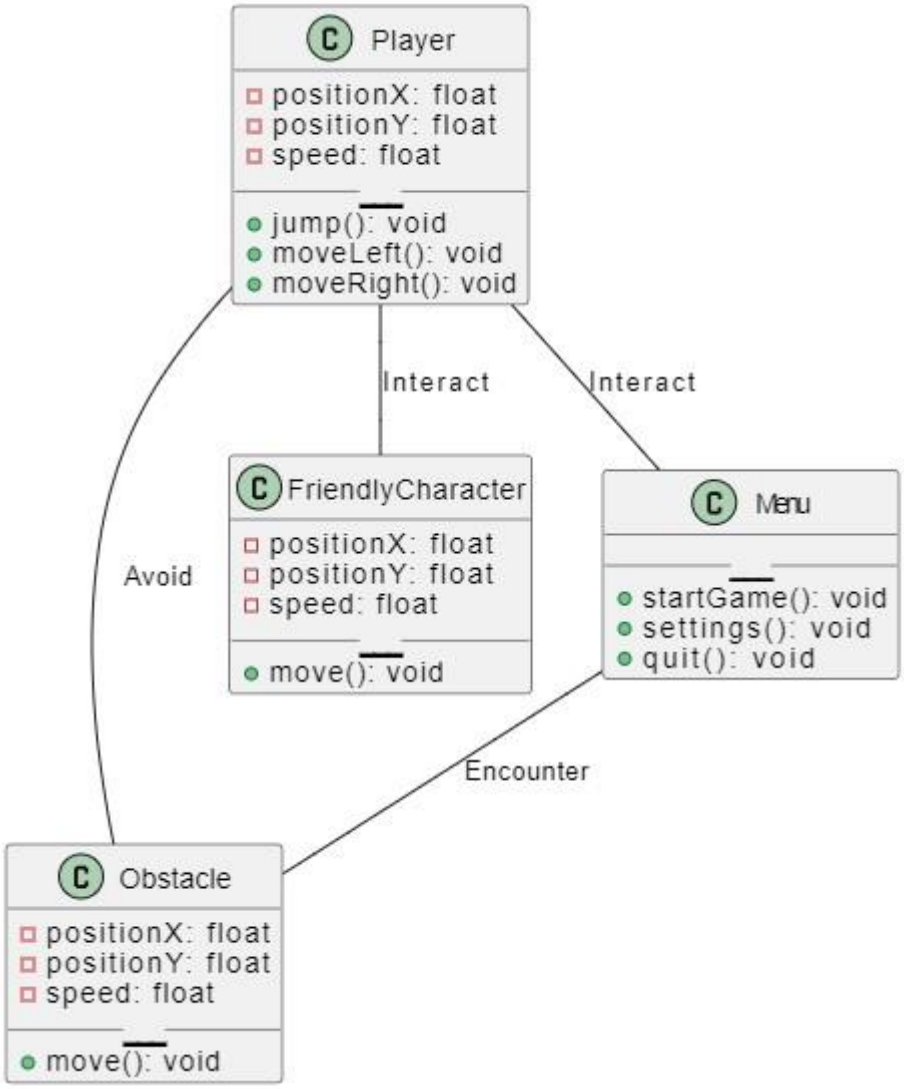

Рисунок 2.2 – Діаграма класів

Результати цих діаграм допомагають уточнити вимоги до програмного продукту і забезпечити однозначне розуміння функціональних та структурних аспектів системи. Вони стануть основою для подальшої розробки програмного забезпечення та його успішної імплементації.

2.3 Алгоритм роботи програмного забезпечення

Алгоритм роботи програмного забезпечення – це набір інструкцій або процедур, які визначають послідовність операцій, необхідних для виконання конкретного завдання чи досягнення певної мети в програмі. Це базовий компонент будь-якої програми, який визначає, як вона працює та як взаємодіє з користувачем і оточуючою системою.

Основна мета алгоритму роботи програмного забезпечення – це забезпечити ефективне виконання програми, забезпечити правильне взаємодію з користувачем і іншими компонентами системи, а також досягнення поставлених цілей та вимог до програми. Алгоритми визначають, як програма буде обробляти дані, приймати рішення і взаємодіяти з користувачем або іншими програмами.

Алгоритм роботи програмного забезпечення даної гри наведено на рис.2.3.

Дана схема представляє алгоритм роботи гри. Вона описує такий алгоритм роботи:

1. Запуск гри та відображення головного меню: Гравець запускає гру, після чого на екрані відображається головне меню з доступними опціями;

2. Початок гри та геймплей: Після вибору опції "Грати" гра розпочинається. Гравець взаємодіє з грою, керуючи персонажем та виконуючи різноманітні дії;

3. Перевірка на зіткнення з перешкодою: В процесі гри перевіряється, чи сталося зіткнення головного героя з перешкодою. Якщо так, гра завершується, і відображається екран із висновками та можливістю перегляду рекордів;

4. Продовження гри: Якщо зіткнення не сталося, гра продовжується, і гравець може продовжити грати;

5. Вихід з гри: Після завершення гри або при бажанні гравця вийти, програма закривається.

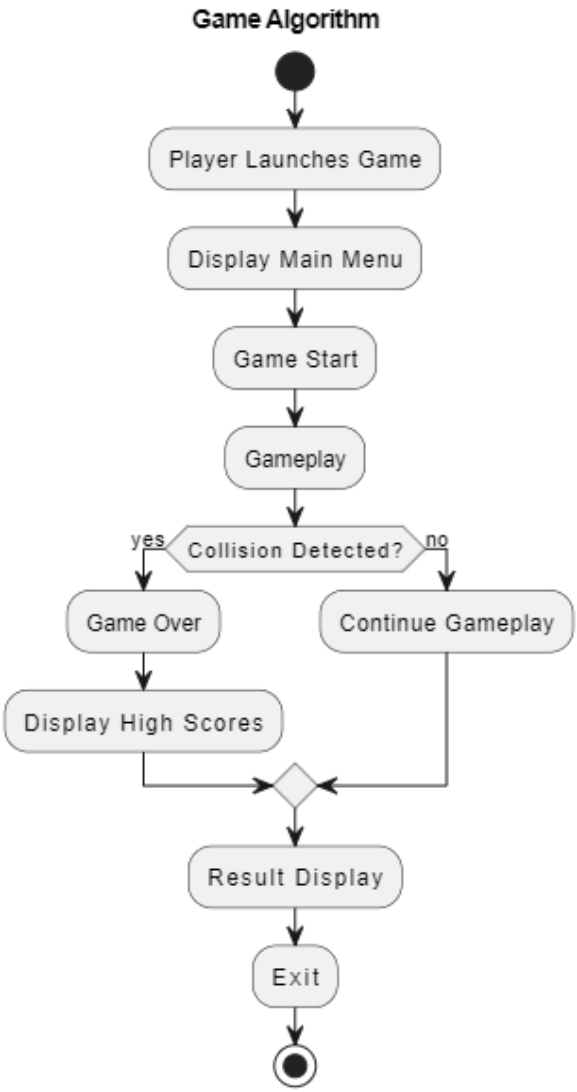

Рисунок 2.3 – Блок-схема роботи програмного забезпечення гри

Даний алгоритм допомагає управляти логікою гри та забезпечити коректний хід подій під час гри. Це дозволяє зробити процес гри логічним та зрозумілим для користувача, що підвищує задоволення від гри.

2.4 Висновки до розділу 2

Розділ присвячений моделюванню програмного продукту за допомогою UML-діаграм. Було створено діаграму прецедентів та діаграму класів для гри у жанрі "раннер". Аналіз цих діаграм надав чітке уявлення про основні взаємодії користувача з програмою та її структуру.

Крім того, був розроблений алгоритм роботи програмного забезпечення, який визначає основні кроки, що відбуваються під час використання гри. Цей алгоритм описує процес запуску гри, відображення головного меню, сам процес гри, обробку колізій та відображення результатів.

Аналіз цих моделей та розроблений алгоритм дав чітке уявлення про те, як буде працювати дане програмне забезпечення та які функціональні можливості воно матиме. У подальших розділах буде описано реалізацію програмного продукту на основі цих моделей та алгоритму.

3 РОЗРОБКА ГРИ «RABBIT RUN»

3.1 Створення 3D моделей

Створення якісних 3D моделей є важливим етапом у розробці гри, оскільки вони визначають вигляд та взаємодію об'єктів у грі. Всі моделі в грі було створено за допомогою програми MagicaVoxel. Спочатку було створено декорації і елементи дороги які будуть рухатися одна за одною, імітуючи рух гравця, який на справді стоїть на місці.

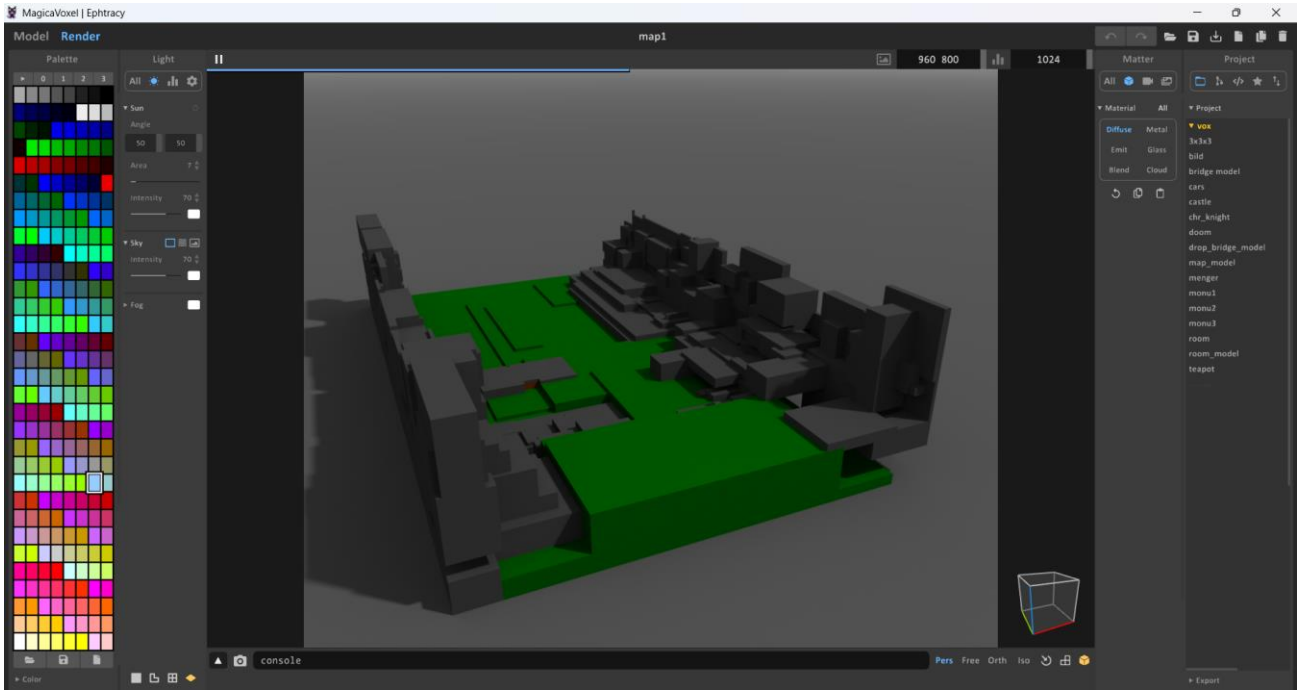

Рисунок 3.1 – Декорації які рухатимуться разом із частинами дороги

Після чого було створено всі перешкоди та персонажів, а сами різних кроликів та лисицю. Головним героєм яким керує гравець було обрано кролика.

Всі кінцівки для всіх персонажів, включаючи головного героя, збережені окремими файлами, для того щоб в подальшому зробити анімації цих персонажів в Unity.

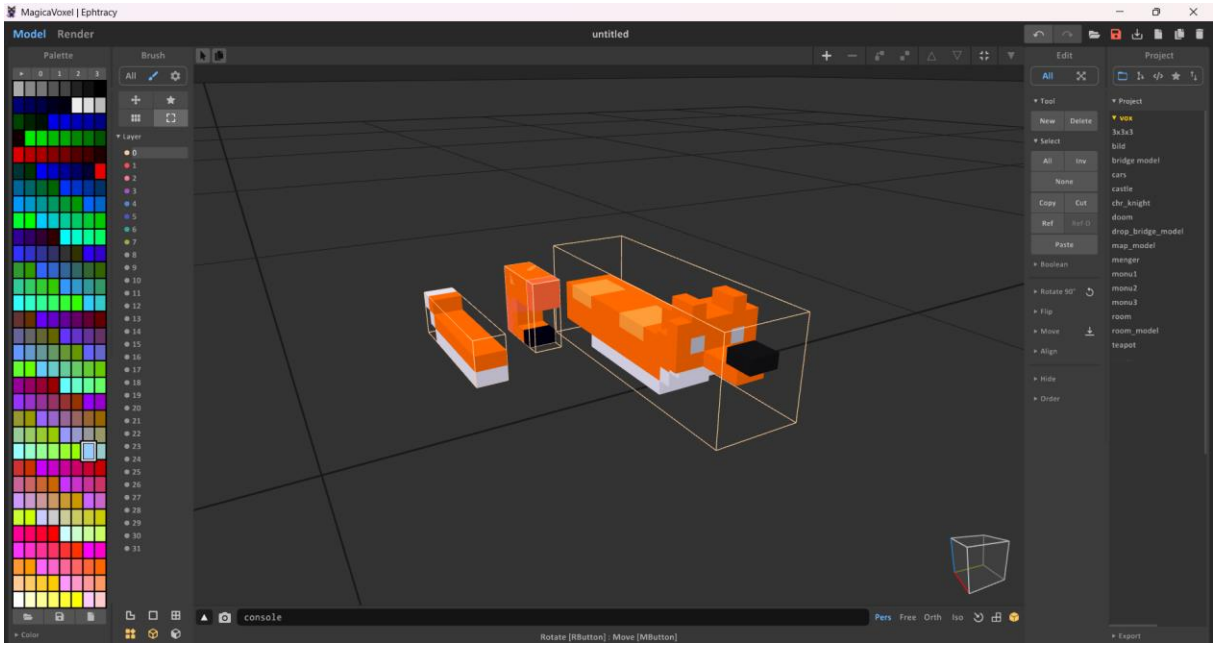

Рисунок 3.2 – Вигляд моделі лисиці в редакторі MagicaVoxel, де всі кінцівки зроблені окремо

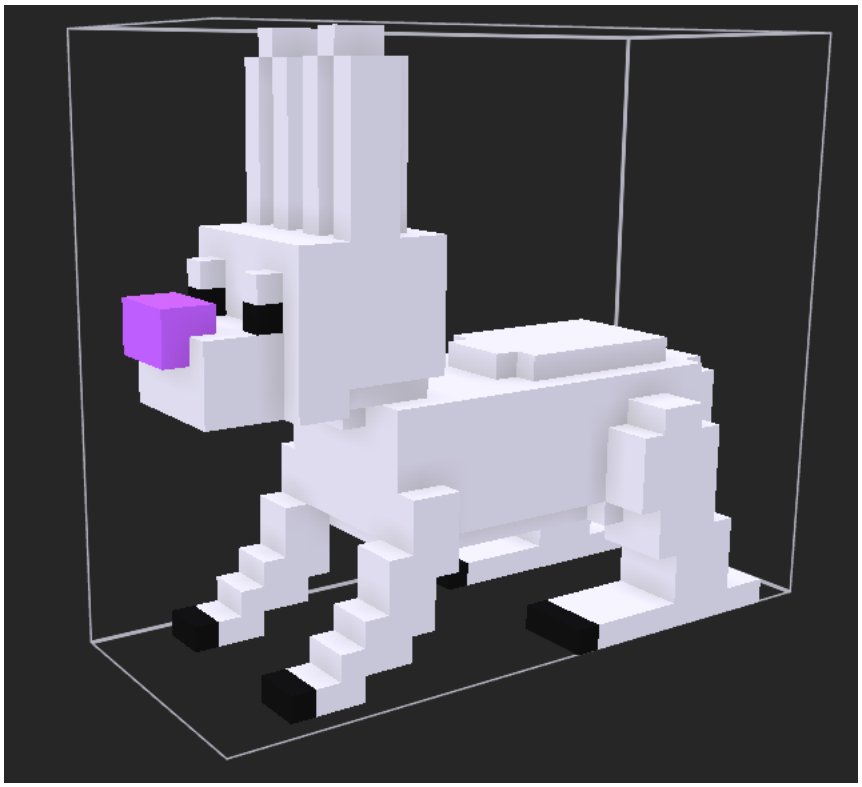

Рисунок 3.3 – Вигляд моделі головного героя

Після цього всі ці моделі було перенесено до Unity, та всі частини персонажів було зібрано до купи та збережено у префаби.

Перфаби – це готові шаблони об'єктів в Unity, які можуть містити в собі різні компоненти, такі як моделі, скрипти, колайдери тощо. Перфаби можна легко перетягувати та розміщувати у сцені, що робить їх ідеальними для створення складних об'єктів без необхідності постійного перетворення одних та тих самих елементів. Вони зберігаються в окремому файлі та можуть бути легко модифіковані та оновлені. Це дозволяє зберігати консистентність об'єктів у проєкті та швидко вносити зміни. Також використання перфабів дозволяє ефективно використовувати ресурси, оскільки одні й ті ж об'єкти можуть бути використані в різних частинах гри без необхідності створення копій [9].

3.2 Створення анімацій в Unity

Аніматори та анімації в Unity – це потужні інструменти для створення рухомих об'єктів та живого геймплею в іграх. Вони дозволяють програмувати різноманітні анімаційні переходи та взаємодіяти з ними через скрипти, створювати вражаючі спецефекти та візуальні ефекти.

Ось основні поняття, пов'язані з аніматорами та анімаціями в Unity:

- Аніматор в Unity – це засіб керування анімаціями, що дозволяє програмувати різноманітні стани та переходи між ними. Він визначає, які анімації будуть відтворюватися в певний момент часу, залежно від різних умов та подій у грі;
- Анімаційні кліпи – це набори ключових кадрів, які описують рух або дії об'єкта. Вони можуть включати в себе рухи персонажів, зміни форми, спецефекти тощо. Анімаційні кліпи можна створювати у спеціалізованих програмах або в самому Unity;
- Параметри аніматора – це змінні, які впливають на роботу аніматора та його переходи між станами. Наприклад, параметр «Speed» може визначати швидкість руху персонажа, і на його основі можуть відбуватися переходи між анімаціями ходьби, бігу тощо;

- Переходи – визначають, як аніматор переходить від одного стану до іншого. Переходи можуть залежати від умов, таких як значення параметрів аніматора, час або зовнішні події;
- Параметризація – це можливість динамічно змінювати параметри аніматора через скрипти. Це дає можливість гнучко керувати анімаціями в залежності від різних факторів у грі, таких як дії гравця чи зміни у грі[10].

Так, до всіх персонажів, чи будь-яких інших об’єктів які мають мати анімації, було додано аніматори, за допомогою яких налаштовуються і відображаються всі анімації, а також записано анімаційні кліпи.

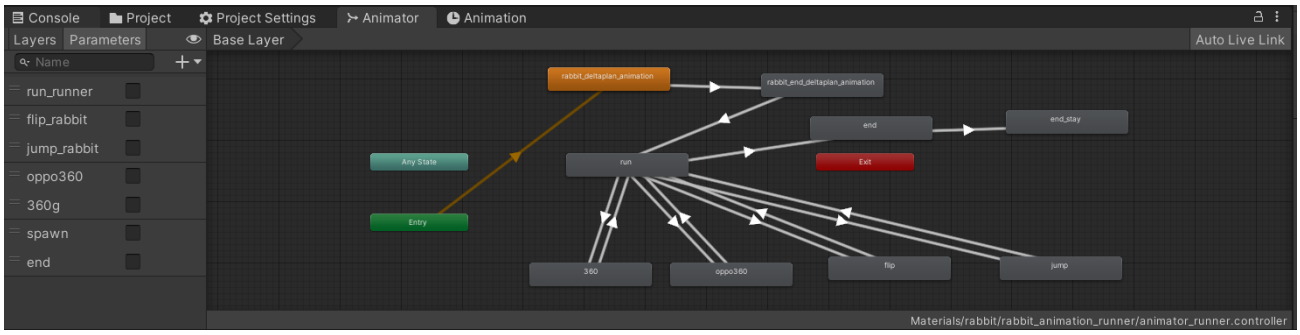

Рисунок 3.4 – Аніматор головного героя

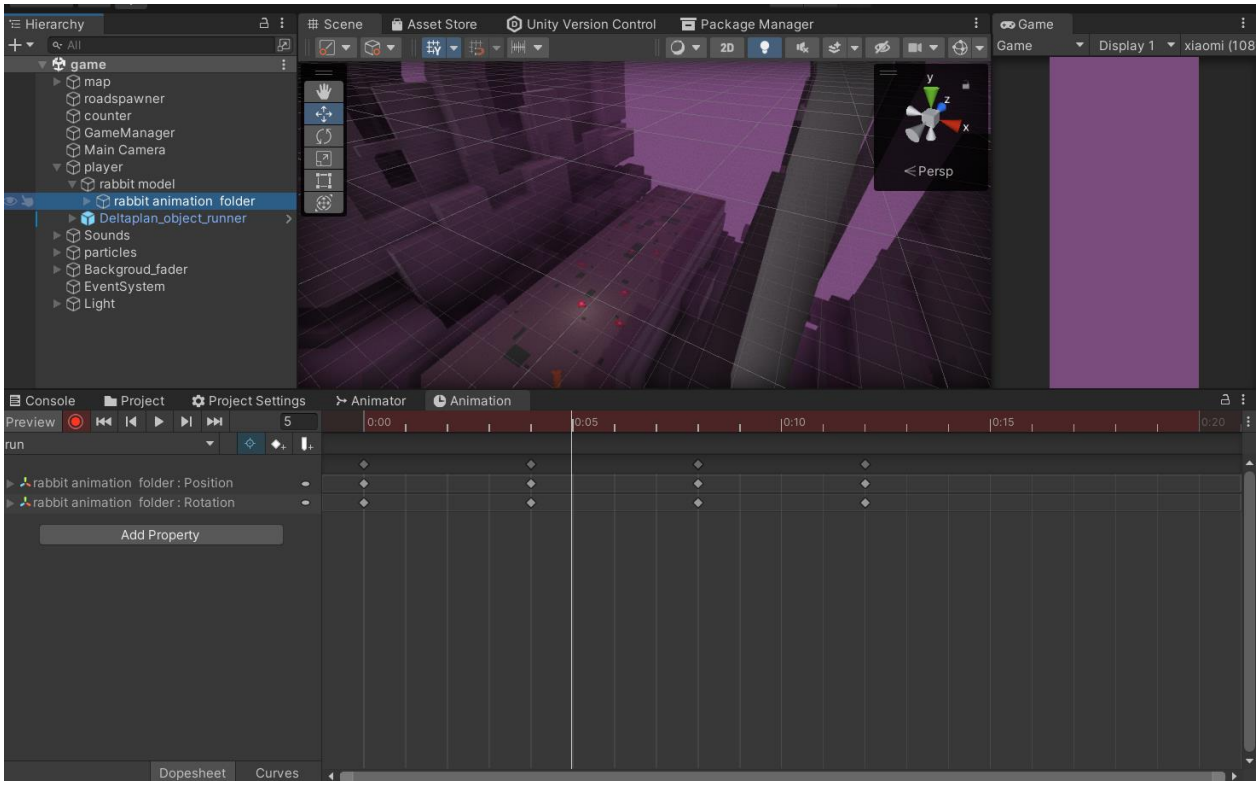

Рисунок 3.5 – Процес запису анімаційних кліпів для головного героя

3.3 Організація скриптів в Unity

Скрипти в Unity – це шматочки програмного коду, які використовуються для керування поведінкою об'єктів у грі. Вони можуть містити інструкції для руху об'єктів, обробки введення користувача, взаємодії з іншими об'єктами в грі та багато іншого. Основна мова програмування для написання скриптів у Unity – це C#. Скрипти можуть бути прикріплені до будь-якого об'єкта в сцені Unity і виконуватися відповідно до подій, які відбуваються в грі, або з викликом певних методів.

Використання скриптів дозволяє розробникам створювати складну поведінку для об'єктів у грі, таку як рух персонажа, поведінку штучного інтелекту ворогів, реакцію на взаємодію гравця з об'єктами тощо. Вони також використовуються для реалізації різних ігрових механік, таких як системи управління, обробки фізики, анімації, обробки подій тощо.

Важливим аспектом використання скриптів є їх організація, яка включає в себе структурування коду, розділення функціональності на логічні блоки, оптимізацію та документування коду. Це допомагає підтримувати читабельність та розширюваність проєкту, спрощує виявлення та усунення помилок, а також полегшує співпрацю між розробниками.

3.4 Реалізація генерації локації

Однією з основних ознак ігор у жанрі раннер є генерація нескінченного рівня з перешкодами. Скрипт який реалізує цю генерацію називається «road_spawner.cs» і має такі об’єкти та методи:

1) Списки об'єктів:
   а) roads: Список доступних дорожніх шляхів (доріг);
   б) preps: Список доступних перешкод на дорозі;
2) Змінні:
   а) road: Поточний об'єкт дороги;
   б) prep: Поточний об'єкт перешкоди;

в) RoadLong: Довжина дороги;

3) Методи:

а) Start(): При запуску гри ініціалізується початковий стан генерації локації. За допомогою функції «Instantiate()» створюються початкові об'єкти дороги і перешкоди;

б) Spawnprep(): Викликається для створення нової перешкоди на дорозі. Вона розміщується на випадковій позиції в межах заданих координат і випадково обирається зі списку доступних перешкод;

в) Spawn(): Викликається для створення нової дороги. Нова дорога створюється на певній відстані вперед від поточної дороги з урахуванням заданої довжини дороги.

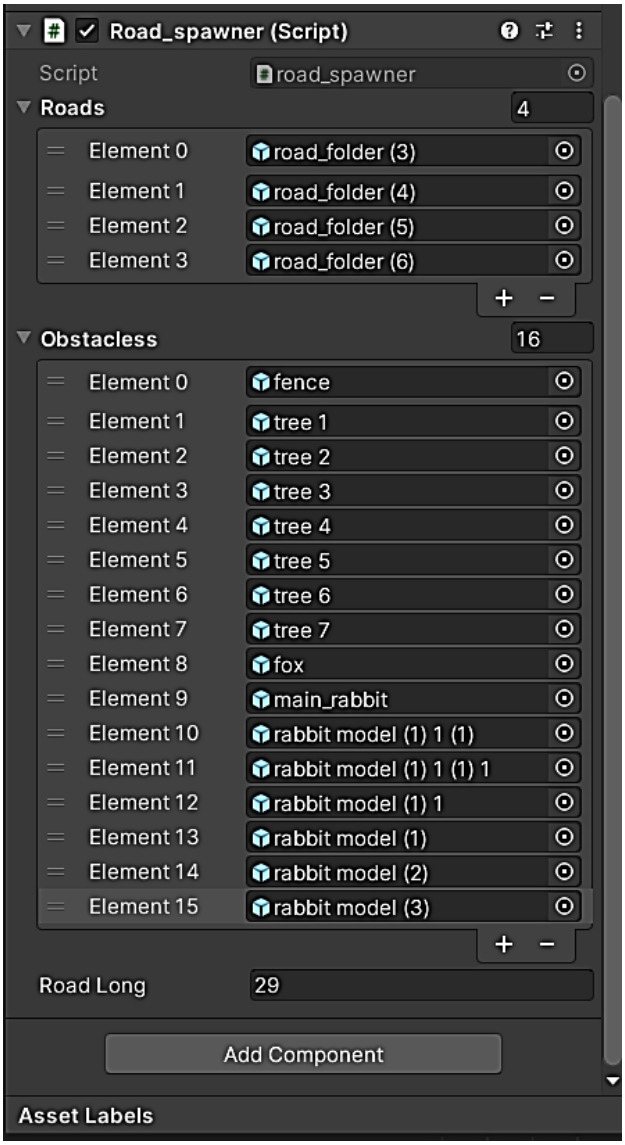

Рисунок 3.6 – Дороги та перешкоди додані в списки

В списки «_roads» та «_obstacles» слід додати елементи які повинні спавнитися у якості дороги та перешкод. До речі, інші позитивні персонажі які будуть бігти разом із головним героєм (у даному випадку це зайці та лисиці) також вважаються перешкодами і додаються в список «_obstacles».

3.5 Налаштування руху перешкод

Рух доріг та перешкод відбувається за допомогою скрипта «road_script.cs».

Цей скрипт відповідає за рух та взаємодію об'єктів на дорозі у грі. Основні функції цього скрипта включають:

- Налаштування параметрів руху та фізики об'єктів;
- Керування швидкістю руху об'єктів;
- Обробка зіткнень з іншими об'єктами та виконання відповідних дій в залежності від типу зіткнення;
- Видалення дорожнього елементу, якщо він виходить за межі видимої області;
- Запуск анімації персонажів та інші дії, пов'язані з їх рухом.

Цей скрипт слід прикріпити до кожного елементу який додано в список «_obstacles», тобто до всіх перешкод, доріг та інших персонажів, адже саме він ними всіма керує, визначає їх поведінку, та відображає анімації.

До речі, завдяки «road_script.cs» всі персонажі, не включаючи головного героя, оминають перешкоди. У випадку коли перешкоду не можна перестрибнути вони її обходять, а коли можна перестрибнути – перестрибують. Під час стрибка відображається одна із чотирьох анімацій. Як це реалізовано в коді, наведено на рис. 3.7.

```
if (-_speed != 2 && collision.collider.tag == "jumptag")
{
    rb.AddForce(Vector3.back * 400, ForceMode.Impulse);
    rb.AddForce(Vector3.up * 600, ForceMode.Impulse);
    jump_value = Random.Range(0, 4);

    if (jump_value == 0)
        bot_animator.SetBool("flip_rabbit", true);
    if (jump_value == 1)
        bot_animator.SetBool("jump_rabbit", true);
    if (jump_value == 2)
        bot_animator.SetBool("360g", true);
    if (jump_value == 3)
        bot_animator.SetBool("oppo360", true);

}
```

Рисунок 3.7 – Фрагмент коду який реалізує стрибок не ігрових персонажів

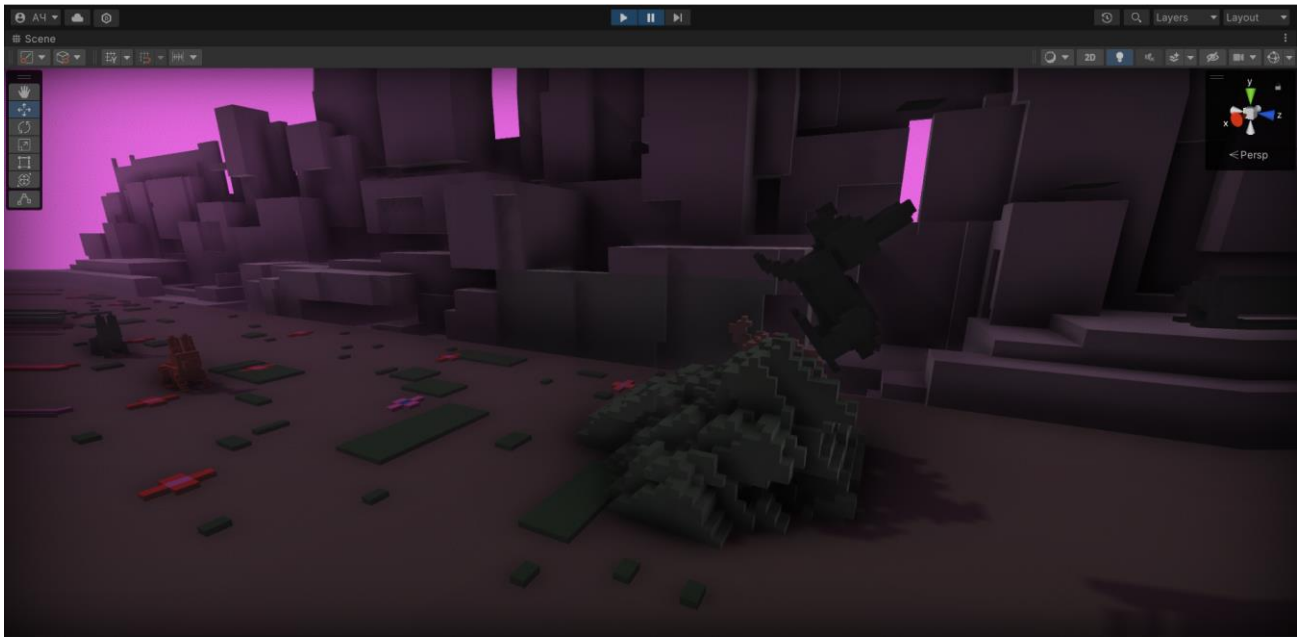

Рисунок 3.8 – Перестрибування не ігровим персонажем перешкоди

3.6 Реалізація керування головним героєм

Головний герой може пересуватися вліво та вправо, а також стрибати, за допомогою скрипта «Player.cs». В ньому реалізовано пересування персонажем вліво і вправо за допомогою нахилу телефона (чи будь-якого іншого пристрою з операційною системою Android). А стрибок – натисканням на будь-яку частину екрана. Також в цьому скрипті реалізовано керування персонажем за допомогою

клавіатури, для того щоб було зручніше тестувати гру в самому редакторі Unity.

Скрипт «Player.cs» містить такі методи:

- Start() – викликається при запуску гри і відповідає за початкові налаштування гравця та анімацій;
- OnCollisionEnter(Collision collision) – викликається при зіткненні гравця з іншим об'єктом та виконує різні дії в залежності від типу зіткнення;
- OnTriggerEnter(Collider other) – викликається, коли гравець зіштовхується з тригером, і відповідає за спавн нових доріжок та перешкод;
- Update() – викликається кожен кадр і відповідає за оновлення стану гравця та обробку введення;
- FixedUpdate() – викликається з фіксованою частотою і відповідає за фізичне моделювання руху гравця.

Для головного героя та інших зайців розроблено чотири різні анімації стрибка. Тож в даному скрипті відбувається вибір (випадковим чином) однієї із цих анімацій і відображення її.

Рисунок 3.9 – Відображення анімації стрибка головного героя під час гри

3.7 Створення інших сцен

3.7.1 Сцени в Unity

Unity сцена – це область, де розробляється та відтворюється гра чи додаток. Вона визначає простір, де відбувається вся дія гри та взаємодія гравців з контентом. Сцени допомагають організувати проєкт, встановлюючи різні «сцени» або етапи, які гравець може пройти. Наприклад, може бути окрема сцена для головного меню, інша для рівнів гри, і ще одна для екрану налаштувань.

Сцени також дозволяють ефективно керувати ресурсами проєкту, такими як моделі, текстури, звуки тощо. Можна завантажувати та вивантажувати ресурси для кожної сцени окремо, що допомагає оптимізувати використання пам'яті та прискорює завантаження гри.

Крім того, сцени допомагають у керуванні потоком гри: вони визначають порядок, в якому гравці переходять від одного етапу до іншого. Наприклад, коли гравець завершує один рівень гри, він може автоматично перейти до наступного, який представлений як окрема сцена.

Загалом, сцени в Unity допомагають організувати, керувати та розробляти проєкт, що робить процес створення гри більш зручним та ефективним [11].

3.7.2 Створення початкової сцени

Для того, щоб при запуску гри одразу не запускався основний геймплей, слід зробити початкову сцену. Тому було створену нову сцену «start_scene». В неї додано два острови із різноманітними декораціями, та ігрового персонажа, який може рухатися за допомогою скрипта «Player_open_world.cs».

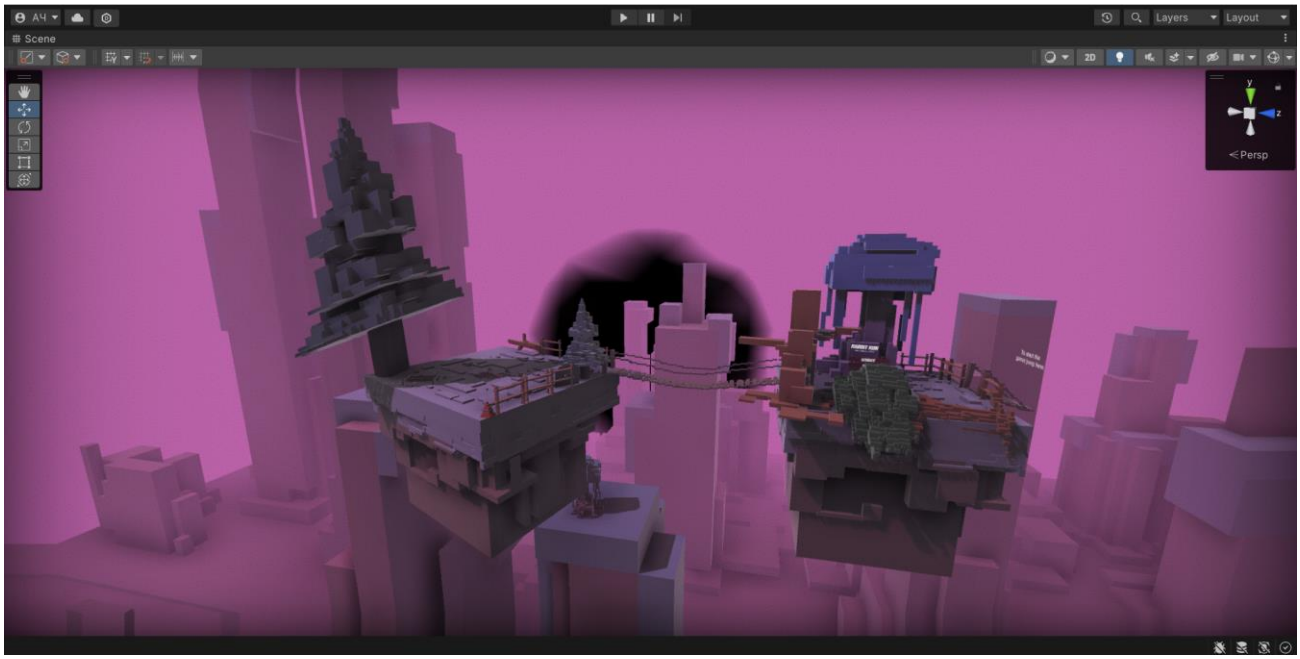

Рисунок 3.10 – Вигляд початкової сцени

Якщо користувач спробує зістрибнути з острова, то запуститься так звана «анімація порятунку», тобто анімація того як персонаж повисне на краю острова і застрибне назад на нього.

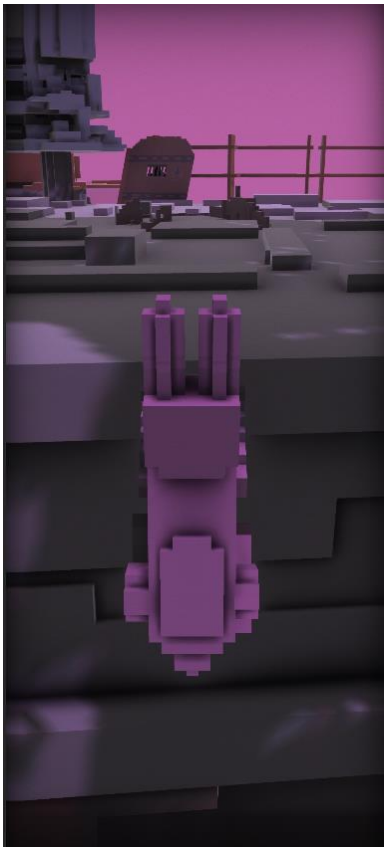

Рисунок 3.11 – Вигляд «анімації порятунку»

Для того щоб почати гру і запустився основний геймплей, треба зістрибнути зі спеціального мосту, зображення якого наведено на рис. 3.10. Після чого з’явиться дельтаплан і відобразиться анімація польоту на ньому і через кілька секунд почне плавно зявлятися прозорий фон, імітуючи туман.

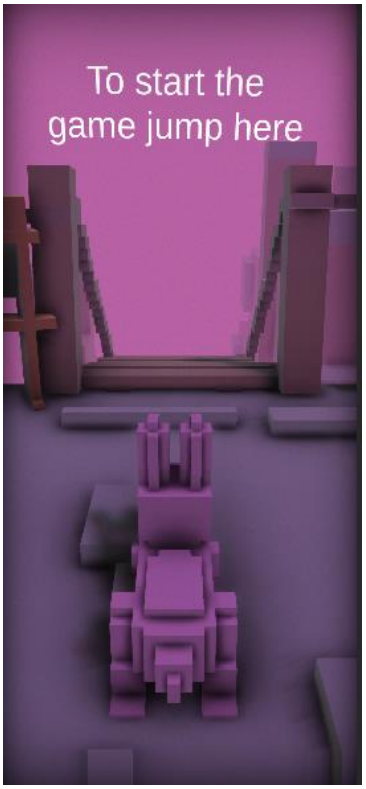

Рисунок 3.12 – Вигляд мосту, із якого треба зістрибнути для запуску основного геймплею

Рисунок 3.13 – Анімація польоту на дельтаплані

Після цього запускається сцена основного геймплею раннера, фон поступово зникає і заєць на дельтаплані приземляється на землю і починається гра.

3.8 Збереження прогресу

Підрахунок і збереження значення рекорду максимальної відстані, яку пробіг персонаж відбувається за допомогою скрипта «kol_script.cs». Значення рекорду можна побачити в меню. А моніторити пройдену відстань можна під час самої гри.

Рисунок 3.14 – Лічильник відстані

Значення рекорду зберігається у PlayerPrefs в форматі .json. PlayerPrefs – це простий механізм для збереження та завантаження даних в Unity. Він дозволяє зберігати дані між різними сеансами гри, що дозволяє, наприклад, зберегти рівень гравця, налаштування гри або прогрес в досягненнях. Отже навіть після закриття гри чи перезавантаження пристрою дані рекорду не втратяться.

3.9 Звуки в грі

У Unity звуки використовуються для створення атмосфери, надання гри більшої імерсії та підсилення враження від гри. Основні компоненти звуків у Unity включають аудіо джерела (Audio Sources), які дозволяють об'єктам відтворювати звуки; аудіо файли, які можна імпортувати в проєкт і

використовувати в грі; мікшери звуку (Audio Mixers), які дозволяють керувати рівнем гучності та ефектами звуку; 3D звук, який змінюється в залежності від положення та орієнтації слухача; і звукові події (Sound Events), які можна програмувати для відтворення звуків у відповідь на певні події в грі. Усі ці компоненти допомагають створити реалістичне звукове середовище, яке збагачує геймплей та збільшує його захоплюючість.

Для того, щоб додати звук треба створити об’єкт на сцені, додати до нього Audio Source і в полі AudioClip вибрати потрібний звук. Після чого цей звук можна відтворювати за допомогою коду, наприклад:

```
run_sound.Play();
```

3.10 Меню та налаштування

В початковій сцені зроблено меню, яке містить три кнопки, а саме: Start; Settings; Quit the game. А також назву гри та значення рекорду максимальної відстані, яку пробіг персонаж.

Рисунок 3.15 – Вигляд меню

В налаштуваннях можна змінити гучність всіх звуків у грі (рис. 3.17) та обнулити значення рекорду максимальної відстані, яку пробіг персонаж (рис. 3.18).

Рисунок 3.16 – Вигляд меню налаштувань

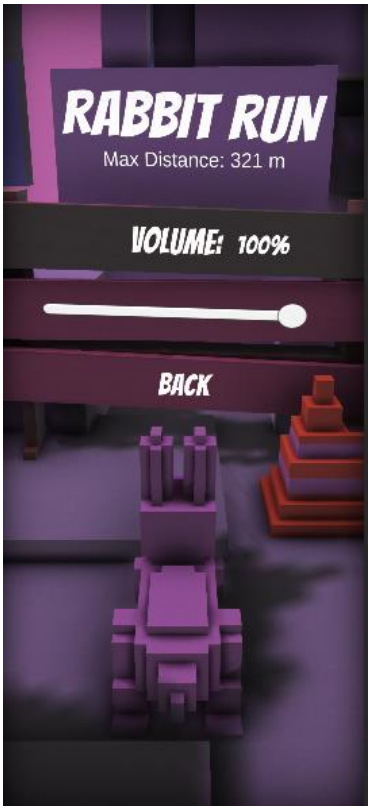

Рисунок 3.17 – Вигляд меню налаштувань гучності

Рисунок 3.18 – Вигляд меню скидання рекорду

3.11 Post-processing

Одним із останніх етапів розробки гри і найголовніших для її зовнішнього вигляду є Post-Processing.

Вбудований постпроцесинг в Unity – це техніка обробки зображення, яка використовується для покращення візуальної якості гри після того, як зображення було згенероване рушієм гри. Цей ефект застосовується після відображення всіх об'єктів на сцені та може включати такі ефекти, як кольорові фільтри, блюр, тінь та інші.

У Unity вбудований постпроцесинг можна реалізувати за допомогою компонентів Post-Processing Stack, які дозволяють додавати різноманітні ефекти до зображення без потреби писати власний код. Ці компоненти включають в себе широкий спектр ефектів, таких як масштабування розмите, додавання кольорових фільтрів, корекція освітлення та інші.

Вбудований постпроцесинг дозволяє досягти вражаючої візуальної якості та покращити атмосферу гри, не тільки за рахунок візуальних ефектів, але і за рахунок оптимізації шляхом обробки зображення на етапі його відображення, що зменшує навантаження на процесор та графічний прискорювач.

Рисунок 3.19 – Вигляд гри до постпроцесингу

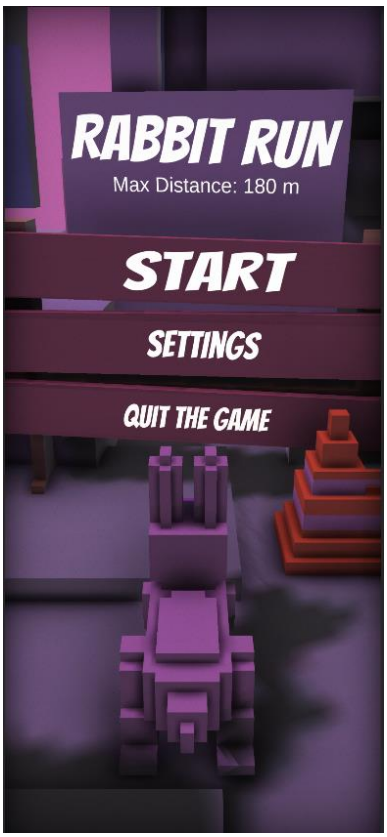

Рисунок 3.20 – Вигляд гри після постпроцесингу

3.12 Налаштування та збірка проєкту у виконуваний файл

Проєктні налаштування (Project Settings) в Unity – це набір параметрів, які визначають поведінку та властивості проєкту в цілому. Ці налаштування включають в себе різноманітні параметри, такі як налаштування фізики, аудіо, графіки, введення та інші. Вони дозволяють змінювати поведінку і вигляд гри, не змінюючи коду.

Компіляція в файл (Build) в Unity – це процес створення виконувального файлу гри для певної платформи. Після компіляції отримується виконуваний файл (наприклад, .apk для Android, .exe для Windows), який можна розповсюджувати серед користувачів. Під час компіляції Unity збирає всі ресурси гри, оптимізує їх та створює виконуваний файл, який готовий до запуску на певній платформі.

Крім того, в Project Settings в Unity можна налаштовувати параметри компіляції для різних платформ. Наприклад, можна вибрати платформу, для треба

скомпілювати гру (наприклад, Android, iOS, PC), налаштувати параметри збірки та вибрати необхідні настройки оптимізації та ресурсів.

У вкладці Project Settings було вибрано назву гри, у даному випадку «Rabbit Run», та іконку гри, яку було заздалегідь зроблено у MagicaVoxel.

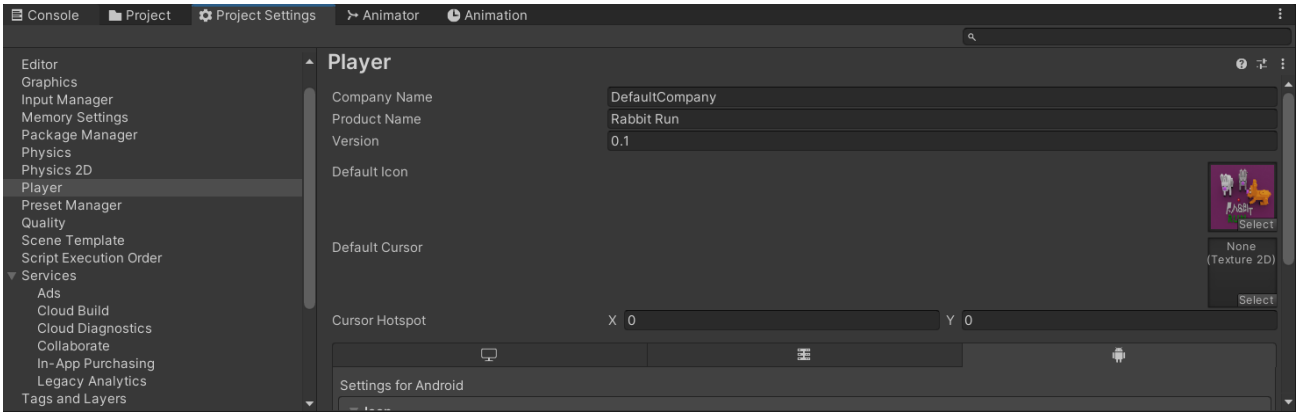

Рисунок 3.21 – Вигляд вкладки Project Settings

Коли все готово, можна запустити процес компіляції, і Unity збере проєкт в виконуваний файл для обраної платформи.

Після чого цей файл можна встановлювати і грати в гру.

Рисунок 3.22 – Вигляд іконки встановленої гри

4 ТЕСТУВАННЯ ТА ОЦІНКА ЯКОСТІ ІГРОВОГО ЗАСТОСУНКУ

Тестування програмного забезпечення – це важливий процес, спрямований на перевірку відповідності програмного продукту вимогам і очікуванням. Його мета – забезпечити якість програми шляхом виявлення та виправлення помилок, недоліків або неполадок перед її випуском.

Тестування дозволяє:

- Виявляти помилки та дефекти у програмному забезпеченні для їх подальшого виправлення;
- Забезпечувати відповідність програмного продукту вимогам і очікуванням користувачів;
- Підвищувати якість і стабільність програми;
- Зменшувати ймовірність аварійного завершення програми та непередбачуваних проблем.

Є багато різних видів тестування, але для тестування даної гри було обрано функціональне тестування.

Функціональне тестування було вибране, оскільки це є одним з найважливіших видів тестування, який дозволяє перевірити функціональні можливості програмного забезпечення. Для забезпечення зручності та задоволення користувачів, надзвичайно важливо щоб функціональність гри працювала належним чином.

Для цього виду тестування було розроблено тест-кейси. Тест-кейси – це документовані сценарії або інструкції, які описують кроки для проведення певного тесту. Кожен тест-кейс включає в себе опис кроків, вхідні дані, очікувані результати та умови, при яких тест може бути визнаний успішним або невдалим. Вони дозволяють систематизувати тестування, забезпечуючи повноту та точність випробувань, а також спрощують процес виявлення, відстеження та виправлення помилок.

Таблиця 4.1 – Тест-кейси для функціонального тестування

Назва тест-кейсу	**Опис тест-кейсу**	**Очікуваний результат**
1. Перевірка всіх кнопок в меню	1. Відкрити гру; 2. Перевірити, що всі кнопки працюють коректно; 3. Перевірити, перевірити меню налаштувань гучності та скидання рекорду.	Всі кнопки працюють коректно і виконують свої функції
2. Перевірка пересування по стартовій сцені, та запуску процесу початку геймплея раннера	1. В меню натиснути кнопку «Start»; 2. Перевірити, чи коректно працює керування персонажем; 3. Перевірити, чи при спробі гравцем випасти із острова, запускається «анімація порятунку».	Пересування коректно працює. «Анімація порятунку» запускається.
	1. Зістрибнути зі спеціального мосту який запускає геймплей раннера.	Має з’явитися дельтаплан, та запуститися основна сцена.
3. Перевірка основного геймплею	1. Перевірити, чи коректно працює керування персонажем; 2. Перевірити спавн і поведінку всіх об’єктів в основній сцені; 3. Перевірити чи при контакті із перешкодами головний герой переміщається в меню;	Керування героєм маж коректно працювати. Поведінка всіх об’єктів у сцені має бути правильною. У випадку контакту з перешкодою головний герой переміщається в меню.
4. Перевірка збереження рекорду	1. Перевірити чи правильно працює лічильник значення рекорду максимальної відстані, яку подолав головний герой в основній сцені; 2. Перевірити чи зберігається значення рекорду максимальної відстані, яку подолав головний герой, і відображається в меню.	Лічильник працює правильно.Значення рекорду зберігається і відображається правильно.

Згідно із тест-кейсами, які наведено в таблиці 4.1, було проведено функціональне тестування. Всі тест-кейси пройдено успішно із цього можна зробити висновок що всі функції в грі працюють правильно і нею можна користуватися та поширювати її.

ВИСНОВКИ

У кваліфікаційній роботі розглянуто процес розробки мобільного ігрового застосунку у жанрі раннер з використанням інструментів Unity. Проведений аналіз показав, що Unity надає широкі можливості для створення захоплюючих ігор, що привертають увагу гравців та забезпечують їм неповторний ігровий досвід.

Відповідно до завдання у рамках роботи зроблено аналіз предметної області, а саме розглянуто специфіку розробки ігрових за стосунків, проведено аналіз наявних аналогів, розроблено функціональні вимоги, виявлено варіанти використання створюваного програмного продукту, вибрано і обґрунтовано вибір Unity та MagicaVoxel для розробки гри та створення 3D моделей, а також розглянуто маркетингові інструменти в створенні та просуванні програмного продукту.

Під час моделювання та проєктування гри створено діаграми класів та прецендентів за допомогою уніфікованої мови моделювання (UML), а також розроблено алгоритм роботи програмного забезпечення.

Були розглянуті різні аспекти розробки ігрового продукту, такі як створення анімацій, додавання та налаштування звуків, робота з графікою та інші технічні аспекти.

Також, в кінці було проведено тестування, яке показало, що гра працює правильно та всі присутні функції правильно працюють.

Метою роботи було створення ігрового застосунку, який не лише привертає увагу гравців, але й задовольняє їхні потреби та відповідає вимогам жанру раннер. В результаті розробки гри «Rabbit Run» був досягнутий успіх, ігровий застосунок готовий задовольнити сучасних користувачів та надати їм незабутні враження від геймплею.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. У 2027 році 2,32 млрд людей будуть грати в мобільні ігри. Як це вплине на бізнес та відносини з клієнтами? URL: https://cases.media/article/u-2027-roci-2-32-mlrd-lyudei-budut-grati-v-mobilni-igri-yak-ce-vpline-na-biznes-ta-vidnosini-z-kliyentami
2. Основи раннер-ігор. URL: https://uk.sharpcoderblog.com/blog/runner-game-fundamentals
3. Ігровий движок Unity: що це таке та на що він здатний? URL: https://itproger.com/ua/news/igrovoy-dvizhok-unity-razbiraemsya-chto-k-chem
4. Функціональні та нефункціональні вимоги. URL: https://visuresolutions.com/uk/requirements-management-traceability-guide/functional-vs-non-functional-requirements/
5. Що таке Unity? URL: https://lemon.school/blog/shho-take-unity
6. Про воксельну графіку. URL: https://foxminded.ua/vokselna-hrafika/
7. UML для бізнес-моделювання: для чого потрібні діаграми процесів. URL: https://evergreens.com.ua/ua/articles/uml-diagrams.html
8. Як будувати UML-діаграми. Розбираємо три найпопулярніші варіанти. URL: https://dou.ua/forums/topic/40575/
9. Prefabs. URL: https://docs.unity3d.com/Manual/Prefabs.html
10. Animation. URL: https://docs.unity3d.com/Manual/AnimationSection.html
11. Scenes. URL: https://docs.unity3d.com/Manual/CreatingScenes.html

ДОДАТКИ

Додаток А. Програмний код

Player.cs

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

public class Player : MonoBehaviour
{
    public kol_script kol_Script; // Посилання на скрипт підрахунку пройдених
метрів
    [SerializeField] private road_spawner roadspawner; // Посилання на спавнер
доріжок

    public Rigidbody rb; // Фізичне тіло гравця
    public GameManager gm;
    private int jump_value;
    public float jumpforce = 11f;   // Сила стрибка
    public float moveSpeed = 100f; // Швидкість руху гравця
    public float maxSpeed = 150f; // Максимальна швидкість гравця

    private int kolls; // Лічильник для підготовки нових доріжок

    private bool jump = false; // Змінна для стрибка
    public bool end = false; // Змінна, що позначає кінець гри
    public bool is_ground; // Змінна, що вказує, чи перебуває гравець на землі

    public Animator animator; // Аніматор гравця
    public bool kol_restart = true; // Змінна для перезапуску гри
    public GameObject backgroundObject; // Об'єкт фону
    public GameObject Deltaplan; // Об'єкт дельтаплану

    [SerializeField] private AudioSource jumpsound; // Звук стрибка
    [SerializeField] private AudioSource crashsound; // Звук зіткнення
    [SerializeField] private AudioSource botsound; // Звук зіткнення з ботами

    void Start()
    {
        // Запускаємо ефект зникаючого фону
        StartBackgroundFade();
        // Встановлюємо анімацію бігу
        animator.SetBool("run_runner", true);
    }
    private void OnCollisionEnter(Collision collision)
    {
        // Перевіряємо, чи гравець на землі
        if (collision.collider.tag == "ground")
        {
            is_ground = true;
        }
        else
        {
            is_ground = false;
        }
```

```
        if (collision.collider.tag == "spawn_barier")
        {
            // Відтворюємо анімацію спавну
            animator.SetBool("spawn", true);
            // Знищуємо дельтаплан
            Destroy(Deltaplan);
        }
        // Перевіряємо зіткнення з ботами
        if (collision.collider.tag == "bot")
        {
            botsound.Play(); // Відтворюємо звук
        }
        // Перевіряємо зіткнення з перешкодами
        if (collision.collider.tag == "lr" || collision.collider.tag == "jumptag"
|| collision.collider.tag == "zabor_tag")
        {
            crashsound.Play(); // Відтворюємо звук зіткнення
            kol_Script.endtextkol = true; // Позначаємо кінець гри в скрипті
підрахунку пройдених метрів
            end = true;
            animator.SetBool("end", true); // Відтворюємо анімацію закінчення гри
            gm.endGame(); // Завершуємо гру
        }
        else
        {
          animator.SetBool("end", false); // Відтворюємо анімацію закінчення гри
        }
    }
    // Метод для запуску ефекту зникаючого фону
    public void StartBackgroundFade()
    {
        // Отримуємо компонент фонового контролера
        BackgroundControllerBasicWorld BackgroundControllerBasicWorld =
backgroundObject.GetComponent<BackgroundControllerBasicWorld>();
        // Якщо компонент існує
        if (BackgroundControllerBasicWorld != null)
        {
            // Запускаємо ефект
            BackgroundControllerBasicWorld.FadeInBackground();
        }
    }
    private void OnTriggerEnter(Collider other)
    {
        // Спавнимо нові доріжки
        roadspawner.Spawn();
        roadspawner.Spawnprep();
    }
    private void Update()
    {

        rb.velocity = Vector3.ClampMagnitude(rb.velocity, maxSpeed);
        // Лічильник для підготовки нових доріжок
        kolls += 1;
        if (kolls >= 70)
        {
            kolls = 0;
            roadspawner.Spawnprep();
        }
```

```
        // Перевіряємо гравець не провалився за карту
        if (transform.position.y < -0.2)
        {
            gm.Error();
            Debug.Log("end");
        }
        // Перевіряємо стрибок
        if ((Input.touchCount > 0 || Input.GetKey("space")) &&
transform.position.y < 0.0135f)
        {
            jump = true;
        }
        else
        {
            jump = false;
        }

        if (jump == true)
        {
            jumpsound.Play();
            animator.SetBool("run_runner", false);
        }
        else
        {
            jump_value = Random.Range(0, 4);
            animator.SetBool("run_runner", true);
        }
        // Встановлюємо випадкову анімацію стрибка
        if (jump_value == 0)
            animator.SetBool("flip_rabbit", jump);
        if (jump_value == 1)
            animator.SetBool("jump_rabbit", jump);
        if (jump_value == 2)
            animator.SetBool("360g", jump);
        if (jump_value == 3)
            animator.SetBool("oppo360", jump);
    }
    private void FixedUpdate()
    {
        // Отримуємо значення нахилу
        float tilt = Input.acceleration.x;
        // Отримуємо значення горизонтального вводу
        float horizontalInput = Input.GetAxis("Horizontal");
        // Застосовуємо силу до гравця в залежності від значень вводу
        if (horizontalInput == 0)
        {
            rb.AddForce(Vector3.right * tilt * moveSpeed * 7, ForceMode.Force);
        }
        else
        {
            rb.AddForce(Vector3.right * horizontalInput * moveSpeed * 6,
ForceMode.Force);
        }
        // Коригуємо орієнтацію гравця
        if (transform.rotation.x != 0 || transform.rotation.y != 0 ||
transform.rotation.z != 0)
            transform.rotation = Quaternion.Euler(new Vector3(0, 0, 0));
```

```
        // Рухаємо гравця вперед під час кінця гри
        if (end)
        {
            transform.Translate(-transform.forward * 11 * Time.fixedDeltaTime);
            rb.AddForce(Vector3.up * 3, ForceMode.Impulse);
        }
        if (jump)
        {
            rb.AddForce(Vector3.up * jumpforce * 10, ForceMode.Impulse);
            jump = false;
        }
    }
}
```

BackgroundController.cs

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

public class BackgroundController : MonoBehaviour
{
    public float fadeSpeed = 1f;
    private Image fadeImage;

    public void FadeInBackground()
    {
        StartCoroutine(FadeIn());
    }

    private IEnumerator FadeIn()
    {
        fadeImage = GetComponent<Image>();
        Color color = fadeImage.color;
        while (color.a < 1f)
        {
            color.a += fadeSpeed * Time.deltaTime;
            fadeImage.color = color;
            yield return null;
        }
    }
}
```

BackgroundControllerBasicWorld.cs

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

public class BackgroundControllerBasicWorld : MonoBehaviour
{
    public float fadeSpeed = 0.1f;
    private Image fadeImage;

    public void FadeInBackground()
    {
        StartCoroutine(FadeIn());
    }
    private IEnumerator FadeIn()
    {
```

```
        fadeImage = GetComponent<Image>();
        Color color = fadeImage.color;
        while (color.a > 0f)
        {
            color.a -= fadeSpeed * Time.deltaTime;
            fadeImage.color = color;
            yield return null;
        }
    }
}
```

camera.cs

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

public class camera : MonoBehaviour
{
    public Transform player;
    public Vector3 ofset;

    void Update()
    {
            transform.position = player.position + ofset;
    }
}
```

DisplayMaxDistance.cs

```
using UnityEngine.UI;
using TMPro;
using UnityEngine;

public class DisplayMaxDistance : MonoBehaviour
{
    public TMP_Text maxDistanceText;
    void Start()
    {
        float maxDistance = LoadMaxDistance();
        maxDistanceText.text = "Max Distance: " + maxDistance.ToString() + " m";
    }
    private float LoadMaxDistance()
    {
        kol_script.GameData data = kol_script.SaveSystem.LoadGameData();
        return data.maxDistance;
    }
    public void ResetHighScore()
    {
        kol_script.SaveSystem.ResetHighScore();
        maxDistanceText.text = "Max Distance: 0 m";
    }
}
```

fox_orientation_fix.cs

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

public class fox_orientation_fix : MonoBehaviour
```

```
{
    public Rigidbody fox_rb;

void Update()
 {
  if (fox_rb.rotation.x != 0 || fox_rb.rotation.y != 0 || fox_rb.rotation.z != 0)
            fox_rb.rotation = Quaternion.Euler(new Vector3(0, 0, 0));
 }
}
```

GameManager.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;
public class GameManager : MonoBehaviour
{
    public float levelRestartDelay = 0.1f;

    public void Error()
    {
        SceneManager.LoadScene("error_menu");
    }
    public void endGame()
    {
        Invoke("restartlevel", levelRestartDelay);
    }
    void restartlevel()
    {
        SceneManager.LoadScene("start world");
    }
}
```

kol_script.cs

```
using UnityEngine.UI;
using TMPro;
using UnityEngine;

public class kol_script : MonoBehaviour
{
    public bool endtextkol;
    private int kol_plus_speed;
    public TMP_Text counter_text;
    private int col_value;
    [System.Serializable]
    public class GameData
    {
        public float maxDistance;
    }
    public class SaveSystem
    {
        public static void SaveGameData(GameData data)
        {
            string json = JsonUtility.ToJson(data);
            PlayerPrefs.SetString("gameData", json);
        }
        public static GameData LoadGameData()
        {
            string json = PlayerPrefs.GetString("gameData");
            return JsonUtility.FromJson<GameData>(json);
```

```
        }
        public static void ResetHighScore()
        {
            PlayerPrefs.DeleteKey("gameData");
        }
    }
    private float maxDistance;
    void Start()
    {

        LoadMaxDistance();
    }

    void Update()
    {
        if (endtextkol == false)
            kol_plus_speed++;
        col_value = kol_plus_speed / 15;
        counter_text.text = col_value + " m";


        if (col_value > maxDistance)
        {

            maxDistance = col_value;

            SaveMaxDistance();
        }
    }


    private void SaveMaxDistance()
    {
        GameData data = new GameData();
        data.maxDistance = maxDistance;
        SaveSystem.SaveGameData(data);
    }


    private void LoadMaxDistance()
    {
        GameData data = SaveSystem.LoadGameData();
        maxDistance = data.maxDistance;
    }


    public void ResetHighScore()
    {
        SaveSystem.ResetHighScore();
        maxDistance = 0;
    }
}
```

menu_script.cs

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

```
public class menu_script : MonoBehaviour
{
    public void playmenu()
    {
        SceneManager.LoadScene("menu");

    }

    public void playGame()
    {
        SceneManager.LoadScene("open world");

    }
    public void exitgame()
    {

        Application.Quit();
        Debug.Log("exit");
    }

}
```

particle_script.cs

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

public class particle_script : MonoBehaviour
{

    private ParticleSystem partSys;
    public Player_open_world player;

    void Start()
    {
        partSys = GetComponent<ParticleSystem>();
        partSys.Stop();
        var particleDuration = partSys.main;
        particleDuration.duration = 5f;


    }
    void Update()
    {

        if (player.Run && player.jump==false)
        {

            partSys.Play();
        }
        else
        {
            partSys.Pause();
            partSys.Clear();
        }
    }
}
```

Player_open_world.cs

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

public class Player_open_world : MonoBehaviour
{
    [SerializeField] private Rigidbody rb_open;
    public GameManager gm;
    public GameObject deltaplan_object;
    public GameObject Island;
    private int rand;
    public Animator animators;

    public bool Run;
    public bool jump;
    public bool jump_button;

    private bool drop;
    private bool isdeltaplan;
    private bool ismenu;

    public float fadespeed = 1f;
    public GameObject backgroundObject;
    public GameObject backgroundObjectFolder;
    public GameObject start_button_destroy;
    private GameObject parentObject;

    [SerializeField] private AudioSource deltaplan_sound;
    [SerializeField] private AudioSource run_sound;
    [SerializeField] private AudioSource wind_sound;

    void Start()    // Метод, що викликається при початку гри
    {
        parentObject = GameObject.Find("Player_open");
        ismenu = true;


    }
    private void OnCollisionEnter(Collision col) // Метод, що викликається при
зіткненні з об'єктами
    {
        if (col.collider.tag == "ground")
        {
            drop = false;
        }

        if (col.collider.tag == "deltaplan_plane")
        {

            backgroundObjectFolder.SetActive(!backgroundObjectFolder.activeSelf);
            rb_open.AddForce(Vector3.forward * 30, ForceMode.Impulse);

            if (isdeltaplan == false)
            {
```

```
                isdeltaplan = true;
                SpawnDeltaplan();
            }
        }

        if (col.collider.tag == "drop")
        {
            Run = false;
            drop = true;
            jump = true;
            animators.SetBool("left_head", false);
            animators.SetBool("right_head", false);
            animators.SetBool("run", false);
            animators.SetBool("drop_save", true);

            Vector3 direction = Island.transform.position - transform.position;

            Quaternion lookRotation = Quaternion.LookRotation(direction,
Vector3.up);
            transform.rotation = lookRotation;

            Invoke("Jump_invoke", 2f);

        }

    }
    public void Jump_invoke()
    {

        rb_open.AddForce(Vector3.up * 70, ForceMode.Impulse);
        transform.Translate(Vector3.forward * 2);
        rb_open.AddForce(Vector3.up * 10, ForceMode.Impulse);
        transform.Translate(Vector3.forward * 1);
        jump = false;
        animators.SetBool("drop_save", false);
    }
    public void StartBackgroundFade() // Метод, який розпочинає зникаючий ефект
фону
    {
        BackgroundController backgroundController =
backgroundObject.GetComponent<BackgroundController>();


        if (backgroundController != null)
        {

            backgroundController.FadeInBackground();
        }
    }

    public void SpawnDeltaplan() // Метод, який створює дельтаплан
    {
        wind_sound.Play();
        Invoke("spawnDeltaplanInvokwElements", 1f);
        animators.SetBool("rabbit_fly_in_deltaplan", true);
        Invoke("StartBackgroundFade", 2f);
        Invoke("playGame", 3f);
```

```
    }
    public void spawnDeltaplanInvokwElements()
    {
        deltaplan_sound.Play();
        // Створюємо дельтаплан з батьківським об'єктом parentObject
        Quaternion heroRotation = transform.rotation;

        Quaternion deltaRotation = Quaternion.Euler(0, 180, 0) * heroRotation;

        // Створюємо дельтаплан з батьківським об'єктом parentObject та
орієнтацією героя + 180 градусів
        Instantiate(deltaplan_object, parentObject.transform.position + new
Vector3(-0.1f, 0.4f, 0), deltaRotation, parentObject.transform);
        animators.SetBool("deltaplan", true);
    }

    public void Start_button() // Метод, що видаляє кнопку "Start"
    {
        Destroy(start_button_destroy);
        transform.Rotate(Vector3.up * 90);
        ismenu = false;
    }
    public void playGame()  // Метод, що викликається для початку гри
    {
        SceneManager.LoadScene("game");
    }



    void Update()   // Метод, що викликається при оновленні кадрів гри
    {
        transform.rotation = Quaternion.Euler(new Vector3(0,
transform.rotation.eulerAngles.y, 0));

        if (Input.touchCount > 0 || Input.GetKey("w"))
        {

            if (ismenu == false && drop == false && isdeltaplan == false)
            {
                Run = true;

                animators.SetBool("run", true);
                animators.SetBool("left_head", false);
                animators.SetBool("right_head", false);
            }
        }
        else
        {
            run_sound.Play();

            Run = false;
            animators.SetBool("run", false);
        }
    }


    private void FixedUpdate()  // Метод, що викликається при оновленні кадрів
фізики
```

```
    {
        Vector3 acceleration = Input.acceleration;

        if (acceleration.x < -0.25 || acceleration.x > 0.25)
        {
            if (jump == false && ismenu == false && drop == false)
            {
                transform.Rotate(Vector3.up * acceleration.x * 2);
                if (Run == false && acceleration.x < -0.25 && isdeltaplan ==
false)
                {
                    animators.SetBool("left_head", true);
                }
                else if (Run == false && acceleration.x > 0.25 && isdeltaplan ==
false)
                {
                    animators.SetBool("right_head", true);

                }
            }
            else
            {
                animators.SetBool("left_head", false);
                animators.SetBool("right_head", false);
            }
        }
        else
        {
            animators.SetBool("left_head", false);
            animators.SetBool("right_head", false);
        }

        if (Run == true)
        {
            transform.Translate(Vector3.forward / 20);
        }

        if (isdeltaplan == true)
            transform.rotation = Quaternion.Euler(new Vector3(0, 0, 0));

        if (transform.position.y < -110)
        {
            gm.Error();

            Debug.Log("end");
        }
    }
}
```

road_script.cs

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

public class road_script : MonoBehaviour
{
    public Player end;
    private bool jump_anim_starter;
```

```
    public int jump_value;
    public Animator bot_animator;
    public Rigidbody rb;
    [SerializeField] private float _speed;
    private float spedcontroller;

    private void Start()
    {
        spedcontroller = _speed;
    }

    void Update()
    {


        if (transform.position.y > 1.2f && _speed != 10)
        rb.AddForce(Vector3.forward * 7, ForceMode.Impulse);

        if (transform.rotation.x != 0 || transform.rotation.y != 0 ||
transform.rotation.z != 0)
            transform.rotation = Quaternion.Euler(new Vector3(0, 0, 0));

        destroy_road();

    }

    private void FixedUpdate()
    {
        Move();
    }

    private void OnCollisionEnter(Collision collision)
    {
        if (_speed != 10)
        {
            if (collision.collider.tag == "Player")
            {
                rb.AddForce(Vector3.up * 300, ForceMode.Impulse);
                rb.AddForce(Vector3.forward * 300, ForceMode.Impulse);
            }
            if (-_speed != 2 && collision.collider.tag == "jumptag")
            {
                rb.AddForce(Vector3.back * 400, ForceMode.Impulse);
                rb.AddForce(Vector3.up * 600, ForceMode.Impulse);
                jump_value = Random.Range(0, 4);

                if (jump_value == 0)
                    bot_animator.SetBool("flip_rabbit", true);
                if (jump_value == 1)
                    bot_animator.SetBool("jump_rabbit", true);
                if (jump_value == 2)
                    bot_animator.SetBool("360g", true);
                if (jump_value == 3)
                    bot_animator.SetBool("oppo360", true);

            }
            else if (-_speed != 2 && collision.collider.tag == "zabor_tag")
            {
```

```
                Destroy(gameObject);
            }
            else
            {


                if (jump_value == 0)
                    bot_animator.SetBool("flip_rabbit", false);
                if (jump_value == 1)
                    bot_animator.SetBool("jump_rabbit", false);
                if (jump_value == 2)
                    bot_animator.SetBool("360g", false);
                if (jump_value == 3)
                    bot_animator.SetBool("oppo360", false);

            }
        }
        if (collision.collider.tag == "lr")
        {
            _speed = 1;
            if (rb.position.x > 0)
            {
                transform.position = transform.position + new Vector3(-40 *
Time.deltaTime, 0, 0);
            }
            else
            {
                transform.position = transform.position + new Vector3(40 *
Time.deltaTime, 0, 0);
            }



        }
        else
        {

            _speed = spedcontroller;
        }
    }

    private void Move()
    {
        transform.Translate(-transform.forward * _speed * Time.fixedDeltaTime);
    }

    private void destroy_road()
    {
        if (transform.position.z < -50f)
            Destroy(gameObject);
    }
}
```

road_spawner.cs

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

```
public class road_spawner : MonoBehaviour
{
    [SerializeField] private List<GameObject> _roads;
    private GameObject _road;
    [SerializeField] private List<GameObject> _obstacless;
    private GameObject _obstacles;
    [SerializeField] private float RoadLong=25;
    // Start is called before the first frame update
    void Start()
    {
        _road = Instantiate(_roads[Random.Range(0, _roads.Count - 1)],
transform.position, Quaternion.identity);
        _obstacles = Instantiate(_obstacless[Random.Range(0, _obstacless.Count -
1)], transform.position, Quaternion.identity);
    }

    public void Spawnprep()
    {

        Vector3 positionpreps = new Vector3((Random.Range(-1, 3)), 0,
Random.Range(70, 80));

        _obstacles = Instantiate(_obstacless[Random.Range(0, _obstacless.Count)],
positionpreps, Quaternion.identity);
    }
    public void Spawn()
    {
        Vector3 positionroads = new Vector3(0, 0, _road.transform.position.z +
RoadLong);

        _road = Instantiate(_roads[Random.Range(0, _roads.Count - 1)],
positionroads, Quaternion.identity);

    }
}
```

VolumeControl.cs

```
using UnityEngine;
using UnityEngine.UI;
using TMPro;

public class VolumeControl : MonoBehaviour
{
    public AudioSource[] audioSources;
    public Slider volumeSlider;
    public TMP_Text volumeText;
    private const string VolumePrefKey = "Volume";

    void Start()
    {
        float savedVolume = PlayerPrefs.GetFloat(VolumePrefKey, 1f); //
Завантаження збереженого значення гучності
        volumeSlider.value = savedVolume;
        UpdateVolumeText(savedVolume);
SetVolume(savedVolume); // Застосування збереженого значення гучності при запуску
    }

    public void SetVolume(float volume)
```

```
    {
        foreach (AudioSource source in audioSources)
        {
            source.volume = volume;
        }

        UpdateVolumeText(volume);
        PlayerPrefs.SetFloat(VolumePrefKey, volume); // Збереження гучності
    }

    void UpdateVolumeText(float volume)
    {
        volumeText.text = Mathf.RoundToInt(volume * 100) + "%";
    }
}
```

VolumeControlGame.cs

```
using UnityEngine;

public class VolumeControlGame : MonoBehaviour
{
    public AudioSource[] audioSources;
    private const string VolumePrefKey = "Volume";

    void Start()
    {
        // Отримуємо збережене значення гучності з PlayerPrefs
        float savedVolume = PlayerPrefs.GetFloat(VolumePrefKey, 1f);

        // Встановлюємо гучність для всіх аудіоджерел на сцені
        SetVolume(savedVolume);

    }

    // Метод для встановлення гучності
    public void SetVolume(float volume)
    {
        foreach (AudioSource source in audioSources)
        {
            source.volume = volume;
        }
    }

    // Метод, який буде викликаний, коли змінюється гучність в PlayerPrefs
    public void UpdateVolumeFromPlayerPrefs()
    {
        float savedVolume = PlayerPrefs.GetFloat(VolumePrefKey, 1f);
        SetVolume(savedVolume);
    }
}
```