**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**
**МАРІУПОЛЬСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ**
**ЕКОНОМІКО-ПРАВОВИЙ ФАКУЛЬТЕТ**
**КАФЕДРА СИСТЕМНОГО АНАЛІЗУ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ**

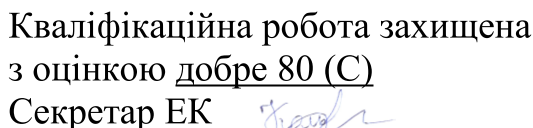
**До захисту допустити:**
**В.о. зав. кафедри**

**Ганна МАРТИНЮК**

квітня 2024 р.

# «ІГРОВИЙ ДОДАТОК У СЕРЕДОВИЩІ UNITY 3D»

Кваліфікаційна робота
здобувача вищої освіти першого (бакалаврського) рівня
освітньо-професійної програми
«Комп'ютерні науки»
Козловцева Дмитра Сергійовича
Науковий керівник:
Мнацаканян Марія Сергіївна,
кандидат технічних наук, доцент,
доцент кафедри системного аналізу та інформаційних технологій
Рецензент:
Марченко Надія Борисівна,
кандидата технічних наук, доцента,
доцента кафедри комп'ютеризованих систем управління факультету комп'ютерних наук та технологій Національного авіаційного університету

Кваліфікаційна робота захищена
з оцінкою добре 80 (С)
Секретар ЕК


» червня 2024 р.

Київ – 2024

# АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи на тему «Ігровий додаток у середовищі Unity 3D» поєднує в собі наступні розділи: аналітичний, дослідницький та проектний.

В першому розділі кваліфікаційної роботи проведено огляд сучасного стану ігрової індустрії України. В цілому, можна зазначити, що сьогодні в Україні як і у всьому світі щороку зростає попит на індустрію комп'ютерних розваг, розмір ринку яких у 2023 році досяг 200 мільярдів доларів. А по Україні можна сміливо зазначити, що індустрія комп'ютерних ігор перемогла індустрію кіно.

Також в першому розділі проведено аналіз основних жанрів комп'ютерних ігор. Тут варто зазначити, що по статистиці найбільш популярними іграми на ПК є мережеві, рольові, шутери, платформери і стратегії. А на мобільних пристроях симулятори, карткові ігри, пазли та головоломки.

У другому розділі проаналізовані основні етапи створення комп'ютерних ігор, та зазначено про важливість кожного з них. Також дана порівняльна характеристика трьох найбільш популярних ігрових рушія: Unity, Godot Engine та Unreal Ungine.

В третьому розділі кваліфікаційної роботи отримано гру з двомірною перспективою і 3D компонентами. Гра має один нескінченний рівень, під час якого ігровий персонаж – космічний корабель «Enterprise» летить у космічному просторі і на своєму шляху стикається з ворожими об'єктами – іншими космічними кораблями. Гравець методом стрілянини знищує ворожі об'єкти і отримує нарахування балів. Гра завершується, якщо гравця буде знищено ворожими об'єктами методом зіткнення з ними.

# SUMMARY

The explanatory note to the qualification work on the topic "Game application in the Unity 3D environment" combines the following sections: analytical, research and design.

In the first section of the qualification work, an overview of the current state of the gaming industry of Ukraine was carried out. In general, it can be noted that today in Ukraine, as in the whole world, the demand for the computer entertainment industry is growing every year, the market size of which in 2023 will reach 200 billion dollars. And in Ukraine, we can safely say that the computer game industry has defeated the film industry.

The second chapter analyzes the main stages of creating computer games, and indicates the importance of each of them. The comparative characteristics of the three most popular game engines are also given: Unity, Godot Engine and Unreal Engine.

In the third section of the qualification work, a game with a two-dimensional perspective and 3D components was obtained. The game has one endless level, during which the game character - the spaceship "Enterprise" flies in space and encounters enemy objects - other spaceships on its way. The player destroys enemy objects by shooting and receives points. The game ends if the player is destroyed by enemy objects by colliding with them.

# ЗМІСТ

**ВСТУП** ........ 5
**РОЗДІЛ 1** ........ 6
**АНАЛІТИЧНІ АСПЕКТИ СТВОРЕННЯ КОМП'ЮТЕРНИХ ІГОР** ........ 6
**1.1 Огляд сучасного стану ігрової індустрії України** ........ 6
**1.2 Характеристика основних жанрів комп'ютерних ігор** ........ 8
**Висновки до I розділу** ........ 11
**РОЗДІЛ 2** ........ 12
**ТЕХНОЛОГІЇ РОЗРОБКИ ІГРОВИХ ПРОДУКТІВ** ........ 12
**2.1 Основні етапи проектування і створення ігор** ........ 12
**2.2 Порівняльний аналіз інструментальних засобів розробки ігор** ........ 13
**Висновки до II розділу** ........ 17
**РОЗДІЛ 3** ........ 19
**ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ КОМП'ЮТЕРНОЇ ГРИ НА UNITY** ........ 19
**3.1 Планування розробки проекту** ........ 19
**3.2 Підготовка графічних елементів та звукових ефектів** ........ 20
**3.3 Тестування та діагностика гри** ........ 44
**Висновки до III розділу** ........ 46
**ВИСНОВКИ** ........ 48
**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ** ........ 49
**ДОДАТОК А** ........ 51

# ВСТУП

В сучасному світі достатньо доступних комп'ютерних технологій в ігри грають всі: від малечі до літніх людей. Тому можна з упевненістю сказати: ігор багато не буває. Для людей комп'ютерна гра це можливість розслабитися, вбити час і навіть «спосіб відключитися від реальності». Ринок ігрової індустрії має найбільшу цільову аудиторію, тому тема розробки ігрового додатку завжди актуальна.

Об'єктом кваліфікаційної роботи є дослідження процесу проектування та розробки комп'ютерних ігор на рушії Unity.

Предметом роботи є розробка комп'ютерної гри в жанрі Shooter на рушії Unity.

Мета роботи – проаналізувати існуючі інструментальні засоби розробки ігор (ігрові рушії) і з допомогою одного з них створити гру.

Завданням кваліфікаційної роботи є на підставі аналізу предметної області, створити гру в якій гравець методом стрілянини знищує ворожі об'єкти і отримує нарахування балів. Гра завершується, якщо гравця буде знищено ворожими об'єктами методом зіткнення з ними.

В процесі створення гри виконати налаштування ігрового поля, космічного корабля гравця і ворожих космічних кораблів, снарядів, якими гравець знищує ворожу кораблі та налаштувати фонову музику. Створити систему підрахунку балів, де кожен переможений ворог присуджує гравцеві певний номер балів, і розробити інтерфейс для відображення оцінки. Крім того провести діагностику та тестування гри.

# РОЗДІЛ 1
# АНАЛІТИЧНІ АСПЕКТИ СТВОРЕННЯ КОМП'ЮТЕРНИХ ІГОР

## 1.1 Огляд сучасного стану ігрової індустрії України

В останні десятиліття індустрія відеоігор стрімко розвивалася. Це дозволило їй стати однією з найприбутковіших і найдинамічніших серед індустрій розваг. У сучасних відеоіграх важливу роль відіграють елементи техніки, мистецтва та дизайну. Відіграє важливу роль у культурі, освіті, науці та, звісно, у сфері розваг.

Дослідницької компанії Newzoo [10] вважає що, світовий ринок відеоігор набуває подальшого зростання. За даними статистики відомо, що розмір ринку досяг 185 мільярдів доларів у 2022 році та 200 мільярдів доларів у 2023 році. Він включає ігри, сервіси, обладнання та рекламу. Основним рушієм цього зростання є поширення мобільних ігор, екосистеми кіберспорту, потокові платформи та соціальні мережі.

А що в Україні? Давайте спробуємо відповісти на питання: хто є українські геймери, що використовують ПК чи консоль, за що вони платять і чи в загалі платять і в які ігри грають?

Компанія NielsenIQ у 2020 році досліджувала ігровий ринок України. Мода на комп'ютерні ігри прийшла в Україну ще у 90-х роках. З середини 90-х з'явились ігрові консолі, найвідомішою з яких була Dendy. Потім Sega, Panasonic і нарешті Sony Playstation та Xbox. Але зараз, на відміну від європейських країн та Америки, у яких дві третини гравців використовують для ігор консолі, в Україні у геймерів переважають ПК і ноутбуки. [18]

Тепер давайте розвінчаємо міф – що переважна більшість геймерів в Україні – це підлітки! За даними того ж самого дослідження середній вік українського геймера становить 31 рік. Чоловіків - 51%, що трохи більше за жінок - 49%. Ще цікаво, 7 з 10 гравців мають вищу освіту та 65%, гравців одружені, або мають партнера(ку). [18]

Відповідь на ще одне цікаве питання навіщо українці грають у відеоігри? (рисунок 1.1) [18]

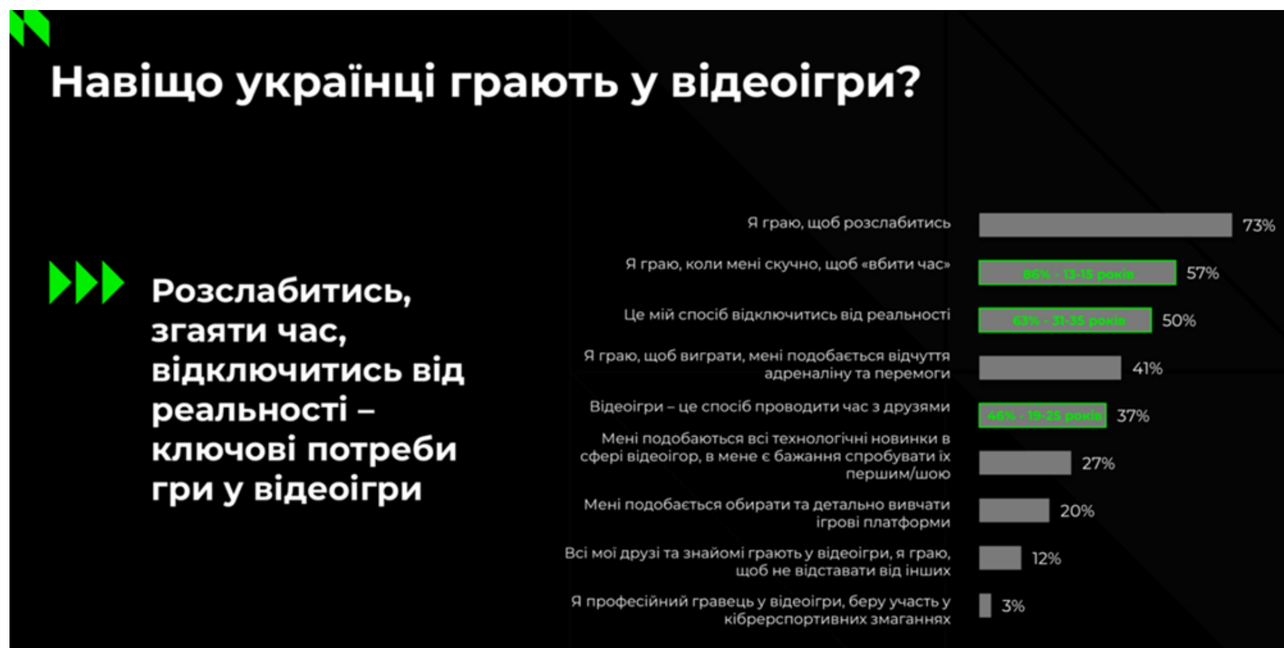



Рисунок 1.1. навіщо українці грають у відеоігри.

Переважна більшість гравців свою мотивацію грати в ігри пояснюють потребою розслабитися, вбити час. А половина визначає потребу як «спосіб відключитися від реальності». І лише 3% геймерів - професійними гравцями.

Ще одне цікаве питання: Скільки коштів витрачають українці на ігри?

За даними того ж дослідження, якщо говорити про образ середньостатистичного українського гравця, то він дотримується принципу: хочу грати, але не хочу платити. Лише троє з 10 геймерів платять за відеоігри.

(рисунок 1.2) [18]

Рисунок 1.2. Витрати українців на відеоігри

Так у середньому на відеоігри українці втрачають від 200 до 500 гривень на місяць.

І останнє питання: які ігри популярні в Україні?

За результатами дослідження NielsenIQ, найпопулярнішою грою в Україні нині є World of Tanks – танкові бої. 16% респондентів здебільшого грають в неї, а 26% опитаних мінімум один раз зіграли в цю гру за останній місяць, ідеться в дослідженні. Друге місце у футбольного симулятора FIFA20/FIFA21 (9% респондентів грають часто і 17% – мінімум раз на місяць), Minecraft (8% і 19%), Grand Theft Auto V (7% і 12%). Лише п’яте місце посідає Counter-Strike: Global Offensive (6% і 14%). Десятку найпопулярніших ігор замикає World of Warcraft. [18]

## 1.2 Характеристика основних жанрів комп’ютерних ігор

Комп’ютерні ігри – це форма інтерактивних розваг, які використовувати комп’ютерні технології. Ці програми працюють на вашому комп’ютері чи іншому пристрої та дозволяє користувачам взаємодіють із внутрішньо створеними віртуальними світами гра [13].

Існує багато типів комп’ютерних ігор і всі ці типи мають унікальні особливості. Жанри комп’ютерних ігор - категорії відрізняються своєю

основною механікою та характеристиками. Жанри допомагають групувати ігри в групи, які мають подібний стиль і геймплей. досвід гравців, що дозволяє їм краще орієнтуватися в комп’ютерному світі. Ігри та виберіть гру на основі ваших інтересів.

Комп'ютерні ігри поділяються на такі основні категорії:

- жанри;

- кількість гравців;

- візуальне виконання;

- платформа.

Тип гри залежить від цілей гри та основної механіки. давайте подумаємо про це найбільш популярні види представлені в таблиці 1.

Види, наведені в таблиці 1, не можна назвати повними, оскільки останнім часом почали з'являтися ігри свого жанру, які можуть вони можуть бути віднесені як до одного з представлених видів, так і до окремого виду.

Залежно від кількості гравців ігри поділяються на два види:

- одиночна гра

- багатокористувацька гра

Комп'ютерні ігри можна класифікувати візуально. Вони виражається наступним чином [16]:

- Гра в слова: мінімум графіки, взаємодія з гравцем робіть це словами.

- 2D-ігри: усі елементи відображаються як 2D-графіка (спрайт).

- 3D гра: всі елементи представлені в тривимірній графіці (3D моделі).

Ігри також можна класифікувати за типом платформи:

- комп’ютер

- Ігрова приставка/консоль

- Мобільний телефон

Таблиця 1 – Жанри комп'ютерних ігор

| **Жанр** | **Опис** | **Основні піджанри** | **Приклади** |
|---|---|---|---|
| Шутери (Shooter) | Гра в основному містить стрілянину Йдеться про боротьбу, бойовики і стрільбу | -від першої особи<br>-від третьої особи | Counter – strike, Doom Eternal, Neon White, Metro Exodus |
| Головоломки (Puzzle) | Ігри де на вибір гравців потрібно проходити різні завдання і головоломки | -загадки<br>- на логіку<br>- дослідження | Roll the Ball, Blendoku 2, Happy Glass, Tiny Bubbles, Aquavias |
| Стратегії (Strategy) | Ігри, в які грають гравці дозволяють їм контролювати свої армії або інші ресурси, будувати бази і розвивати економіку | -покрокові<br>-в режимі реального часу<br>-економічні | Civilization 6, Starcraft II, Warcraft III, Crusader Kings 3, Total War: Warhammer 2 |
| Симулятори (Simulation) | Симулюйте різні ігри елементи реального життя Наприклад, авіасимулятори, симулятори, які дозволяють керувати містом чи поїздом | -технічні<br>-симулятори життя | Farming Simulator 19, Euro Truck Simulator 2, War Thunder |
| Рольові ігри (RPG) | Ігри, у яких гравець керує персонажем, розвиває його навички та майстерність, виконує завдання і бориться з ворогами | -тактичні<br>-екшен (RPG)<br>-японські | The Witcher 1-3, Fallout 1-2, Red Dead Redemption 2, Diablo 2-3 |
| Пригоди (Adventure) | Ігри, у яких гравець керує персонажем, досліджує світ і приймає рішення, вирішує загадки та виконує різні місії | -квест<br>-візуалізація<br>-новели | Helldivers 2, King's Quest, Space Quest, Police Quest |

Запропонована класифікація не є повною. Наприклад, можна додати музичні ігри як окремий тип. Залежно від музики та звуку багатокористувацькі ігри можна розділити на кілька підкатегорій. Однак варто відзначити, що ця класифікація є достатньою, щоб ідентифікувати більшість існуючих ігор, оскільки в даний час немає чіткої і повної класифікації комп'ютерних ігор. Це

тому, що в багатьох іграх не має певних стандартів. Наприклад, гра може поєднувати кілька жанрів, публікуйте на різних платформах або використовуйте сервіси для одного користувача та багатокористувацький режим. Це через те, що індустрія ігор з'явилася відносно недавно і відрізняється від інших сфер розваг.

## Висновки до I розділу

В першому розділі кваліфікаційної роботи проведено огляд сучасного стану ігрової індустрії України. В цілому, можна зазначити, що сьогодні в Україні як і у всьому світі щороку зростає попит на індустрію комп'ютерних розваг, розмір ринку яких у 2023 році досяг 200 мільярдів доларів. А по Україні можна сміливо зазначити, що індустрія комп'ютерних ігор перемогла індустрію кіно.

Також в першому розділі проведено аналіз основних жанрів комп'ютерних ігор. Тут варто зазначити, що по статистиці найбільш популярними іграми на ПК є мережеві, рольові, шутери, платформери і стратегії. А на мобільних пристроях симулятори, карткові ігри, пазли та головоломки.

Таким чином, обраний для розробки гри в даній кваліфікаційній роботі жанр – шутер є одним з найбільш запитаних жанрів сучасної ігрової індустрії.

# РОЗДІЛ 2
# ТЕХНОЛОГІЇ РОЗРОБКИ ІГРОВИХ ПРОДУКТІВ

## 2.1 Основні етапи проектування і створення ігор

Процес розробки гри можна розділити на три етапи:

1. Формування поняття:

– Визначення жанру гри, цільової групи, ігрового процесу та інтерфейсу користувача;

– Розробка сюжету, візуальне та звукове оформлення;

– Вибір технічних засобів для реалізації проекту.

Створення концепції - визначає жанр, цільову групу, геймплей, інтерфейс користувача, зображення та звук, дії та технічні засоби. Важливо визначити цільову групу візуальний стиль і особливості гри. Жанр і можливості гри. Використання елементів інших жанрів визначається виходячи із загального задуму.

2. Розробка гри:

– Створення першого рівня гри та тестування її продуктивності;

– Розробка основного контенту гри, включаючи графічні ресурси, ігрові особливості та баланс;

– Створення альфа-версії, яка містить більшість із них запланований зміст.

Розробка інтерфейсу користувача – враховує вимоги геймплею і потреби користувачів. Має з'явитися інтерфейс користувача, який надає необхідну інформацію, не відволікаючи і не перевантажуючи ігровий процес.

3. Випуск і підтримка готового продукту:

– Максимальний випуск гри в популярних цифрових магазинах звітність аудиторії;

– Підтримка гри, включаючи виправлення помилок за допомогою патчів і додавання додаткового вмісту за допомогою завантажувальні дані (DLC).

Вибір технічних засобів здійснюється на основі аналізу наявних засобів програми та засоби розробки. Вибір ігрового рушія це важливо, оскільки воно впливає на весь процес розвитку. Він повинен відповідати вимогам проекту та кваліфікації розробників, з урахуванням мов програмування та фінансових аспектів.

Після створення концепції можна приступати до розробки гри. Розробка починається зі створення першого рівня, який включає основний рівень Елементи коду та графіки. У цей час можна вносити зміни у концепцію гри. З виходом першої частини гри можна починати тестування триватиме до виходу гри. Потім створюється альфа-версія, яка містить приблизно 90% запланованого вмісту. На цьому стадіоні розробник наповнює гру контентом.

Наступний етап - це бета-версія, коли гра майже закінчена і готова ефективний. На цьому етапі гра буде показана широкій публіці. Знайдіть і виправте помилки, логічні проблеми та помилки. Бета-версія містить усі важливі функції гри, достатньо контенту для гри. Довгостроковий і спеціалізований збір і аналіз статистики. Випробування проводяться згідно з планом випробувань функціональних проб. Це також включає тестування всіх функцій комп'ютерної гри. Після остаточного тестування та виправлення помилок проект готовий до випуску.

Останній етап включає випуск гри та подальшу підтримку. Підтримка ігор включає виправлення помилок на основі відгуків і додавання вмісту, щоб зацікавити громадськість. Додатково завантажуваний вміст (DLC) може бути безкоштовним або платним. Продовжити життєвий цикл гри та отримати додатковий дохід.

З цього можна зробити висновок про етапи розвитку гри майже не відрізняються від етапів розробки інших продуктів.

## 2.2 Порівняльний аналіз інструментальних засобів розробки ігор

У цьому пункті проведемо порівняльний аналіз різних засобів розробки.

Ігровий рушій (надалі – «Рушій») — це набір програм, які об'єднані різноманітні функції для всебічного ігрового досвіду. Він пропонує графічну візуалізацію, звуковий супровід, контролюйте рух персонажів у грі та їхню взаємодію відповідно із заздалегідь визначеними сценаріями. Це робить гру більшою хвилюючий і захоплюючий [1].

У розробці комп'ютерних ігор вони використовуються для спрощення речей процес створення гри.

Unity [12] є кросплатформним движком, який робить це можливим розробляйте 2D і 3D програми та ігри, логотип показано на рисунку 2.1. Unity доступний у двох версіях: безкоштовній і платній, кожна різна ряд важливих особливостей при створенні ігор: безкоштовна версія Unity підтримує Android, Web Player і персональні комп'ютери, платформи.

Рисунок 2.1 – Логотип Unity

Повна версія пропонує розробникам можливість опублікувати ваші проекти на звичайних платформах, таких як ПК, Linux, Mac, Windows Магазин, iOS, Android, Windows Phone 10 Store, Blackberry 10, Wii U, PS3, Xbox 360, PS4 і Xbox One. Крім того, Unity дозволяє розробляти програми для віртуальна реальність (VR), наприклад Hololens, Oculus Rift, StarVR та інші.

Unity можна налаштувати відповідно до потреб розробки комп'ютерних ігор. Можна змінити інтерфейс, видаливши та додавши деякі пункти меню власні налаштування, що спрощують процес розробки. єдність має інтуїтивно

зрозумілий інтерфейс перетягування та підтримка двох мов програмування: C# та JavaScript [2]

Переваги включають:

- мультиплатформенність;
- зручний інтерфейс;
- велика бібліотека ресурсів і плагінів.

До недоліків можна віднести:

- погана оптимізація великих просторів;
- високі витрати на зберігання.

Godot Engine — універсальний ігровий рушій підтримує розробку 2D і 3D ігор на різних платформах. Він надає повний набір інструментів для підтримки розробників зосередьтеся на іграх, а не на повторенні розвиток основних функцій, логотип наведено на рисунку 2.2 використовувати Godot Engine може легко імпортувати ігри на багато платформ лише одним клацанням миші. Підтримуються найважливіші настільні платформи, наприклад такі як Linux, macOS і Windows, а також мобільні платформи, включаючи Android та iOS, а також веб-платформа HTML5 [13].

Переваги включають:

- велика документація;
- невеликий розмір гри;
- підтримує кілька мов програмування.

До недоліків можна віднести:

- відсутність великої кількості функцій через новизну движка;
- мало навчального змісту.

Рисунок 2.2 – Логотип Godot engine

Unreal Engine — це ігровий движок, розроблений і підтримуваний від Epic Games. Вперше з'явилася в 1998 році разом з виходом гри. Відтоді «Unreal» став основою для розробки понад сотні ігор і не тільки для проектів логотип показаний на рисунку 3.

Рушій написаний на C++ і пропонує можливість розробки шгри для різних операційних систем і платформ.

Рисунок 2.3 – Логотип Unreal Engine

Unreal Engine використовує модульну систему залежних компонентів. Це спрощує процес портування ігор на різні платформи. Він підтримує різні системи візуалізації, такі як Direct3D, OpenGL, Pixomatic і в старих версіях також Glide, S3, PowerVR. грати звук використовує EAX, OpenAL, DirectSound3D і раніше була підтримка A3D. Також включені параметри голосового відтворення тексту та розпізнавання голосу. У движку передбачені модулі для роботи з мережею і підтримка різноманітної периферії.

Проаналізувавши перераховані вище рушії суб'єктивно критерії оцінювання за п’ятибальною шкалою, де 1 означає низький рівень оцінки та 5 -

найвищий. Суб'єктивна оцінка за результатами роботи з кожним двигуна, показано в таблиці 2.

Таблиця 2 Порівняльна характеристика ігрових рушіїв

| **Критерії** | **Ігрові рушії** | | |
|---|---|---|---|
| | **Unity** | **Godot Engine** | **Unreal Ungine** |
| Безкоштовне використання | 5 | 5 | 5 |
| Середній поріг входження | 4 | 5 | 3 |
| Кількість платних та безкоштовних додаткових ресурсів | 5 | 2 | 4 |
| Уроки, курси, документація | 5 | 2 | 4 |

Крім цим трьох найбільш розповсюджений рушіїв, існує безліч інших, які не увійшли до цієї порівняльної таблиці, наприклад: GameMaker, AppGameKit, CryEngine, Amazon Lumberyard, RPG Maker, LibGDX, Urho3D та інші.

З порівняльного аналізу суб'єктивних критеріїв оцінки гри для рушіїв, наведених у таблиці 3, оптимальним є Unity. Він отримав найвищу оцінку та найкращі збіги вибрані критерії, такі як обсяг і рівень навчання, доступність для новачків.

## Висновки до II розділу

У другому розділі проаналізовані основні етапи створення комп’ютерних ігор, та зазначено про важливість кожного з них. Так як розробка ігор це дуже тривалий процес під час якого команда розробників створює концепцію гри, визначається з її жанром та цільовою аудиторією, приділяє значну увагу інтерфейсу користувача, звуку, діям та технічним засобам. Далі йде розробка основного контенту гри, включаючи графічні ресурси, ігрові особливості та баланс. На кожному етапі не аби яка увага приділяється процесу тестування всіх функцій гри і після остаточного тестування та виправлення помилок проект готовий до випуску. Випуск і подальша підтримка гри також грає дуже важливу роль, включаючи додавання додаткового вмісту.

Також у цьому розділі кваліфікаційної роботи дана порівняльна характеристика трьох найбільш популярних ігрових рушія: Unity, Godot Engine

та Unreal Ungine. Порівняння відбувалося за наступними критеріями: безкоштовне використання, середній поріг входження, кількість платних та безкоштовних додаткових ресурсів, уроки, курси, документація. Таке порівняння надало змогу віддати перевагу соме Unity через мультиплатформенність, зручний інтерфейс та велику бібліотеку ресурсів і плагінів цього рушія.

# РОЗДІЛ 3
# ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ КОМП'ЮТЕРНОЇ ГРИ НА UNITY

## 3.1 Планування розробки проекту

Жанром гри обрано шутер. Цей жанр є одним з найстаріших жанрів сучасної ігрової індустрії, який домінував на початку 90-х років і донині залишається запитаним через свою яскравість, багатогранність і динамічність.

Шутер (від англ. «shooter» - «стрілець») – жанр, основу ігрового процесу якого складає стрілянина по цілях. Завданням гравця є знищення усіх цілей на рівні та набір найбільшої кількості очок [1].

У центрі сюжету знаходиться космічний корабель «Enterprise», яким керує гравець. Гра містить один рівень на якому «Enterprise» мчить всесвітом і стріляє у наближаючихся ворогів, якими є інші космічні об'єкти. Космічний корабель переміщається по рівню за допомогою стрілок на клавіатурі та WASD. Натискання лівої кнопки миші призведе до вистрілу патронів. Завершену сцену проекту можна побачити на рисунку 3.1.

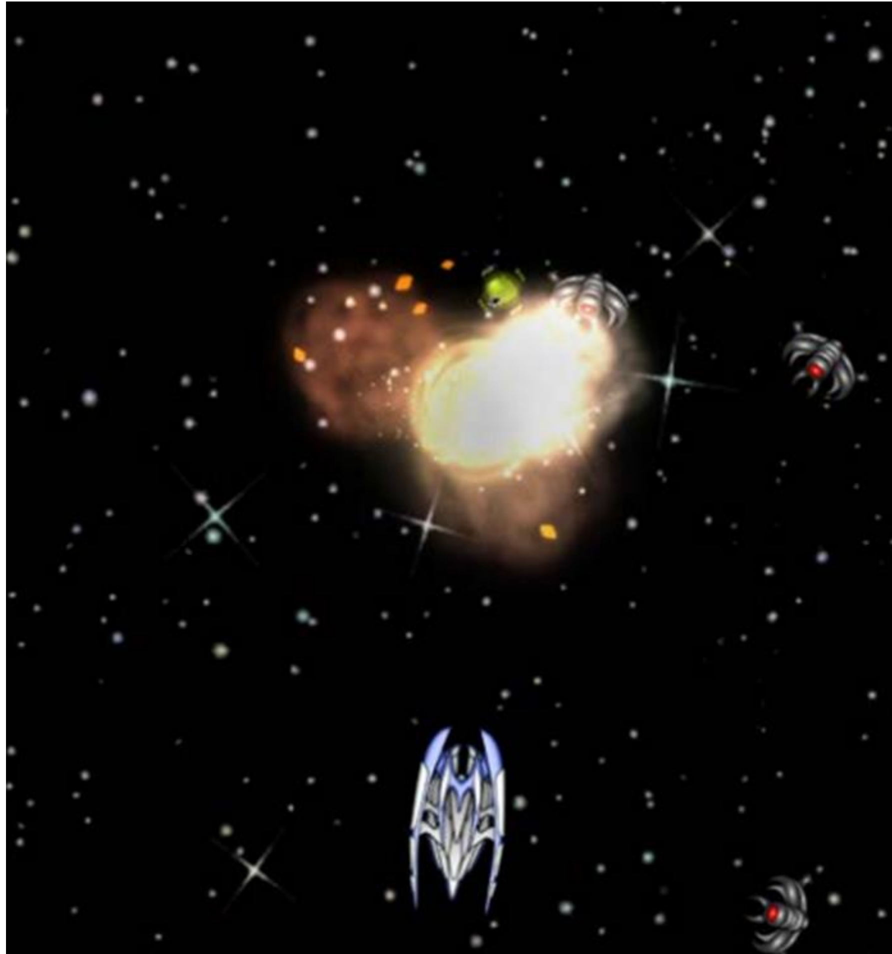


Рисунок 3.1 – Завершена сцена гри

## 3.2 Підготовка графічних елементів та звукових ефектів

Для початку роботи над проектом необхідно створити папки для структурування та організації ресурсів проекту. Створюємо папки текстури, матеріали, префаби, аудіо. Далі слід зазначити, що при розробці гри було використано деякі графічні і звукові ресурси завантажені з онлайн сервісу https://opengameart.org/.

Імпортуємо текстури для космічного корабля гравця, ворожих космічних кораблів, зоряного поля фону та боєприпасів. Перетягуємо текстури на панель проекту unity в папку «текстури». Unity імпортує та налаштовує текстури автоматично. Імпорт текстур ресурсів показано на рисунку 3.2.

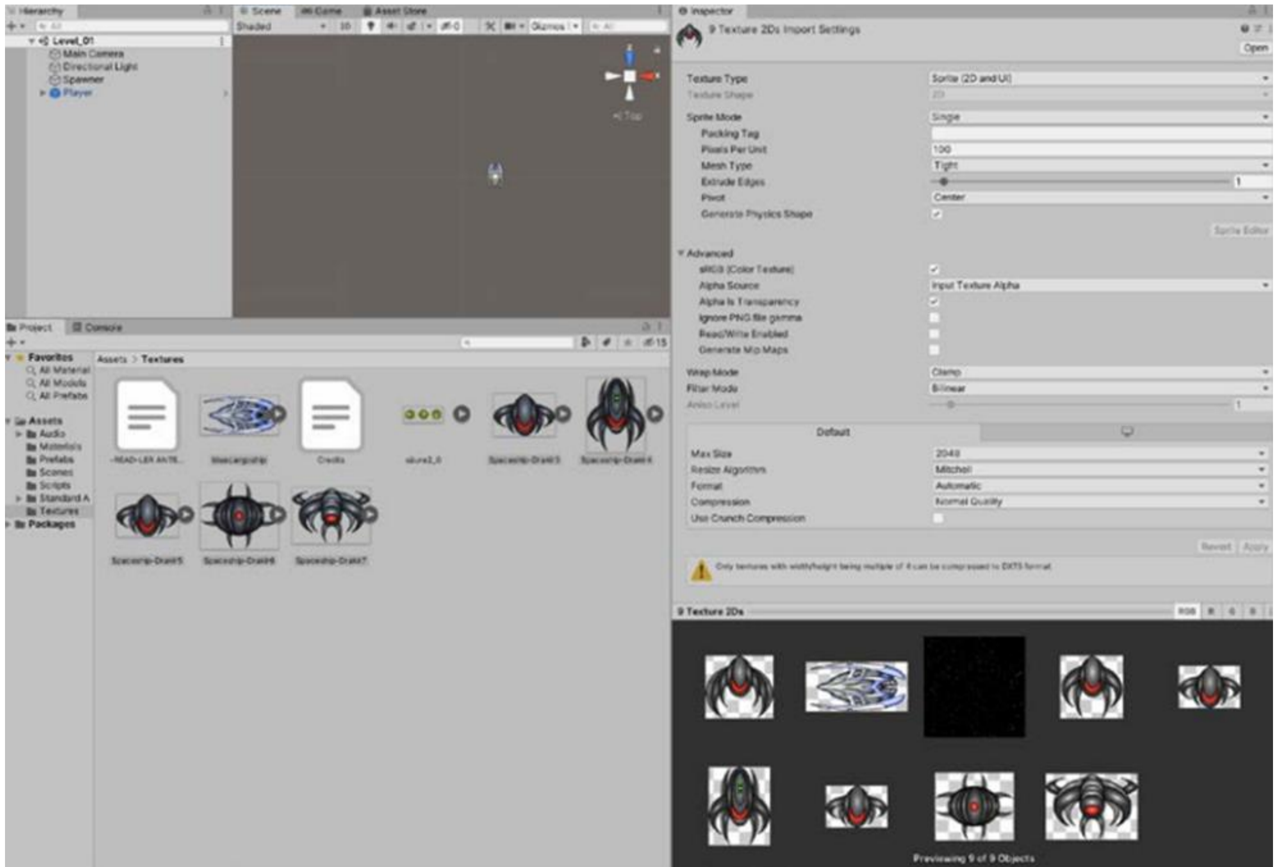


Рисунок 3.2 – Імпорт текстурних ресурсів для космічного корабля, ворогів, фону зоряного поля та боєприпасів

За замовчуванням Unity імпортує файли зображень як звичайні текстури для використання на 3D-об’єктах припускає, що їхні розміри в пікселях є розміром у ступені 2 (4, 8, 16, 32, 64, 128, 256 тощо на). Якщо розмір не є одним із цих, Unity збільшить або зменшить масштаб текстури найближчий дійсний розмір [6]. Для налаштування імпортованих 3D текстур під 2D гру виконаємо наступні дії:

1. Вибираємо усі імпортовані текстури.
2. В Inspector змінюємо Texture Type з Default на Sprite (2D та UI).
3. Натискаємо «Застосувати», щоб оновити налаштування, і текстури збережуть свої імпортні розміри.
4. Знімаємо позначку з поля Generate Mip Maps, якщо її ввімкнено.

**Імпорт аудіо.**

Музика та звукові ефекти важливі. Вони додають додатковий рівень занурення, і є основною частиною ігрового процесу. Щоб імпортувати аудіо,

перетягуємо файли з папки на панель проекту. Далі перевіряємо аудіо в редакторі Unity натиснувши «Відтворити» на панелі інструментів попереднього перегляду в інспекторі, як показано на рисунку 3.3:

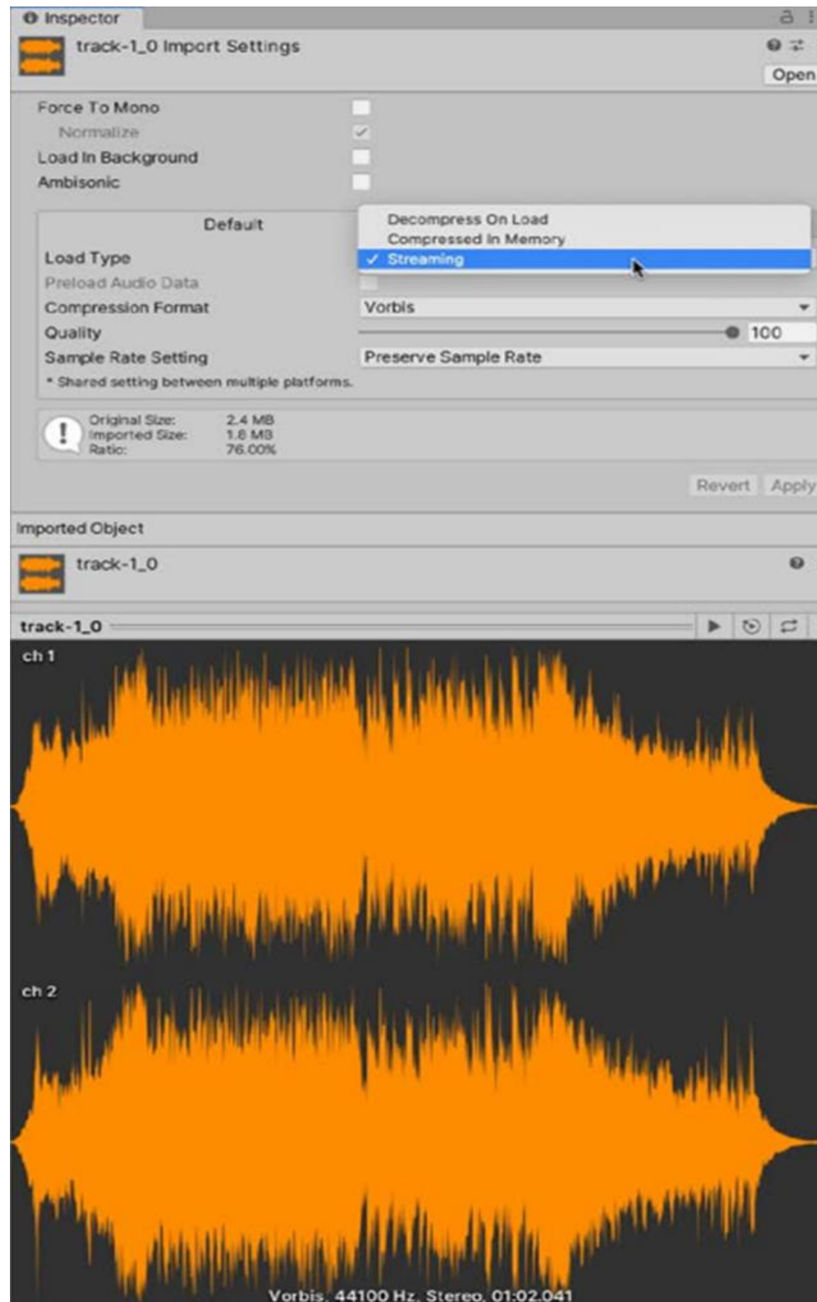



Рисунок 3.3 – Попередній перегляд звуку з інспектора об’єктів

**Створення космічного корабля гравця.**

Космічний корабель «Enterprise» -це буде об’єкт, яким керуватиме та переміщатиме гравець. Об’єкт гравця є складним об’єктом із багатьма різними поведінками. Створюємо космічний корабель за допомогою GameObject, який

міститиме спеціальні компоненти. GameObject буде зберігати всі дані та компоненти, необхідні для нашого гравця, включаючи дані про положення та зіткнення, а також спеціальні функції, які додаємо при написання сценаріїв. Щоб створити об’єкт гравця, виконуємо наступні кроки:

1. Створюємо порожній GameObject на сцені, перейшовши до GameObject/Створити Порожньо з меню програми.

2. Називаємо об'єкт Player.

3. Новостворений об’єкт може бути або не бути зосередженим у центрі (0, 0, 0), і його властивості обертання можуть бути непостійними 0 по X, Y і Z.Щоб забезпечити нульове перетворення, можна вручну встановити значення 0, ввівши їх безпосередньо в компоненті Transform для об’єкта в Inspector.

4. Перетягуємо спрайт Dropship Player (у папці Textures) з Панелі проектування до щойно створеного об’єкта Player на панелі «Ієрархія». Текстура об'єкта гравця робить його дочірнім об'єктом гравця.

5. Повертає цей об’єкт-корабель на 90 градусів по осі X і на 90 градусів по осі Z. Це обертання орієнтує спрайт у напрямку його батьківського вектора.

Тепер переконаємось, що спрайт корабля правильно вирівняно відносно його батьківського елемента. Для цього вибираємо об’єкт Player і перегляд синьої векторної стрілки вперед. Передня частина спрайта корабля та синій передній вектор мають бути спрямовані в одному напрямку. Якщо вони не спрямовані, то обертаємо спрайт на 90 градусів, поки вони не вирівняються. В подальшому це дозволить змусити корабель рухатися в потрібному напрямку. Вирівнювання спрайта корабля показано на рисунку 3.4.

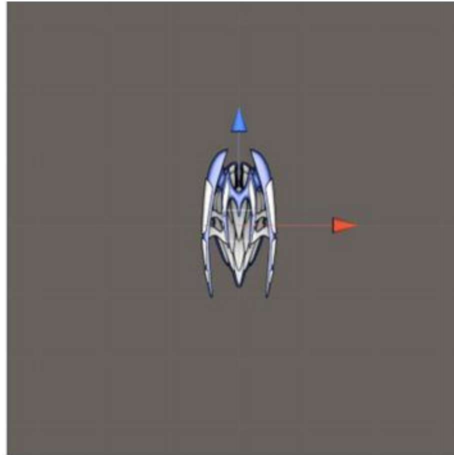


Рисунок 3.4 – Синя стрілка називається прямим вектором

**Додавання фізичних властивостей об’єкту.**

Наступним кроком є додавання об’єкту Player міцності і фізичних властивостей. За сценарієм він повинен стикатися з іншими твердими предметами та отримувати пошкодження від боєприпасів противника при попаданні. Для цього до об’єкта Player слід додати два додаткові компоненти, зокрема:

Жорстке тіло та колайдер:

1. Вибираємо об’єкт Player (а не Sprite).
2. Вибираємо Компонент / Фізика / Rigidbody з меню програми, щоб додати тверде тіло.
3. Вибираємо Компонент / Фізика / Capsule Collider з меню програми до додати колайдер.

Компонент Collider приблизно визначає об’єм об’єкта, а Rigibody компонент використовується для додавання фізичної сили. Так як налаштування за замовченням зазвичай не збігаються зі спрайтом гравця за призначенням, то потрібно відрегулювати капсульний колайдер. [7]. Налаштування значень Напрямок, Радіус і Висота показано на рисунку 3.5.

За замовчуванням компонент Rigidbody налаштовано на апроксимацію об’єктів, які є під впливом гравітації, що не підходить для космічного корабля, який літає навколо. Щоб виправити це, Жорстке тіло слід відрегулюємо наступним чином:

1. Знімаємо прапорець Використовувати силу тяжіння, щоб запобігти падінню об’єкта на землю.

2. Увімкнемо прапорці Позиція заморожування Y і Обертання заморозки Z, щоб запобігати переміщенню та обертанню космічного корабля навколо осей, небажаних у 2D іграх [8].

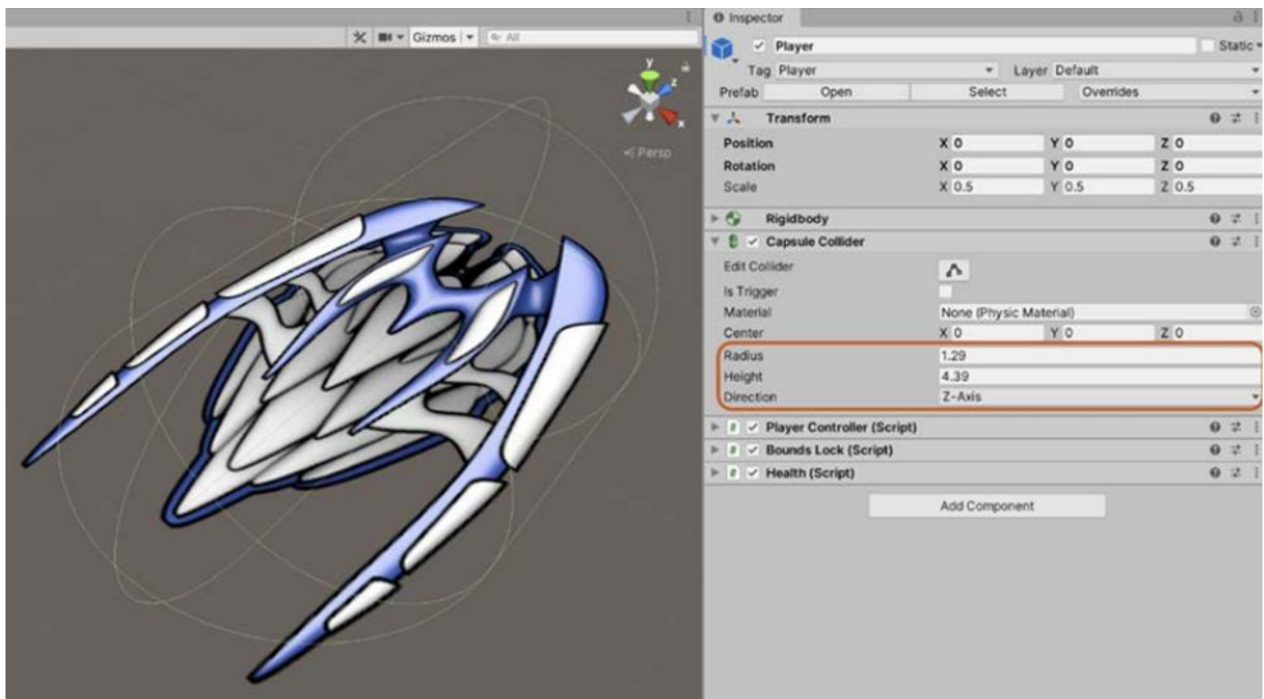



Рисунок 3.5 – Регулювання колайдера капсули космічного корабля

**Рух об’єкта.**

Тепер у сцені створено об’єкт Player, налаштований за допомогою Rigidbody і Компонентів колайдера. Однак цей об’єкт не реагує на елементи керування гравця. Гравець керує об’єктом по двох осях і зазвичай може стріляти зі зброї. Схему управління для гри можна описати так:

• Кнопки WASD на клавіатурі керують рухами гравця вгору, вниз, ліворуч і праворуч.

• Миша контролює напрямок, куди гравець дивиться і прицілюється.

• Ліва кнопка миші стріляє зі зброї.

Щоб реалізувати це, потрібно створити файл сценарію PlayerController. Вибираємо папку «Сценарії» на панелі «Проект» і створюємо новий файл сценарію C# під назвою PlayerController.cs.

За допомогою класу PlayerController об'єкт Player в сцені приймає дані від гравця та керує рухом космічного корабля. Функція Awake зазвичай викликається один раз під час створення об'єкта використовується для отримання посилань на об'єкти, необхідні компоненту [9].

Тепер, коли отримано потрібні компоненти, можемо використовувати їх у Функції FixedUpdate.

Функція FixedUpdate викликається один раз перед оновленням фізичної системи, що є фіксованою кількістю разів на секунду. FixedUpdate відрізняється від Update, яка викликається один раз на кадр і може змінюватися залежно від частоти кадрів на секунду [11].

Функція Input.GetAxis викликається на кожному FixedUpdate для читання аксіального введення даних із пристрою введення, наприклад клавіатури або геймпада. Ця функція зчитується з двох іменованих осей, горизонтальної (ліворуч-праворуч) і вертикалі (вгору-вниз). Вони працюють у нормалізованому просторі від -1 до 1. Це означає, що коли ліву клавішу натиснули й утримують, горизонтальна вісь повертає -1 і, коли праву клавішу натиснули й утримуюь, горизонтальна вісь повертає 1. Значення 0 означає що відповідна клавіша не натиснута, або натиснуті ліва і права разом, скасовуючи один одну [16].

Функція Rigidbody.AddForce застосовує фізичну силу гравця до об'єкта, переміщаючи його в певному напрямку. Вектор MoveDirection кодує напрямок руху та базується на введенні гравцем як з вертикальної, так і з горизонтальної сторони осі. Цей напрямок множиться на максимальну швидкість, щоб забезпечити це сила, прикладена до об'єкта, обмежена [13].

Функція Camera.ScreenToWorldPoint перетворює положення екрана курсор миші у вікні гри на позицію в ігровому полі. Цей код відповідає за те, щоб гравець завжди дивився на курсор миші [6].

Попередній код дозволяє керувати об’єктом Player, але є деякі проблеми. Одна з них полягає в тому, що гравець не повернутий головою до позиції миші. Причина в тому, що Камера за замовчуванням не налаштована, як це потрібно для 2D-ігри зверху вниз. Але з початку потрібно додати ще одну функцію за для запобігання виходу гравця за межі гри.

**Обмеження руху.**

У поточному стані гри гравця можна перемістити за межі екрана. Рух гравця повинен бути обмежений полем зору камери або межами, щоб він ніколи не виходив з поля зору.

Існують різні способи досягнення блокування меж, більшість із яких передбачає використання сценаріїв. Одним із способів є закріплення позиційних значень об’єкта Player між вказаним діапазоном мінімуму та максимуму. Для цього додаємо до коду C# новий клас BoundsLock.

Новою в цьому коді є тільки функція Mathf.Clamp, яка гарантує, що вказане значення обмежене між мінімальним та максимальним діапазоном [12].

Щоб візуалізувати межі рівня і у реальному часі бачити як краще зробити налаштування, які впливають на розмір і положення меж можна скористатись класом Gizmos. Для цього додаємо нову функцію до сценарію BoundsLock.

У OnDrawGizmosSelected викликаємо Gizmos.DrawWireCube, який малює каркас куба із заданим центром і розміром [7]. Розраховується центр і розмір за допомогою прямокутника levelBounds, який створили раніше. Встановлюємо cubeDepth довільно до 1, оскільки наша гра двовимірна, і нас не хвилює глибина рівня межі. Як підказує назва функції Gizmos буде намальовано, лише якщо об’єкт вибрано в ієрархії. В даному випадку потрібні лише межі рівнів, видимі під час редагування, і ця функція ідеально підходить.

Щоб перевірити, чи Gizmos працює правильно, у редакторі Unity вибираємо програвач об'єкт. Оскільки сценарій BoundsLock приєднаний до цього об’єкта, каркас білого куба має бути розміщено у вікні сцени, як показано на рисунку 3.6:

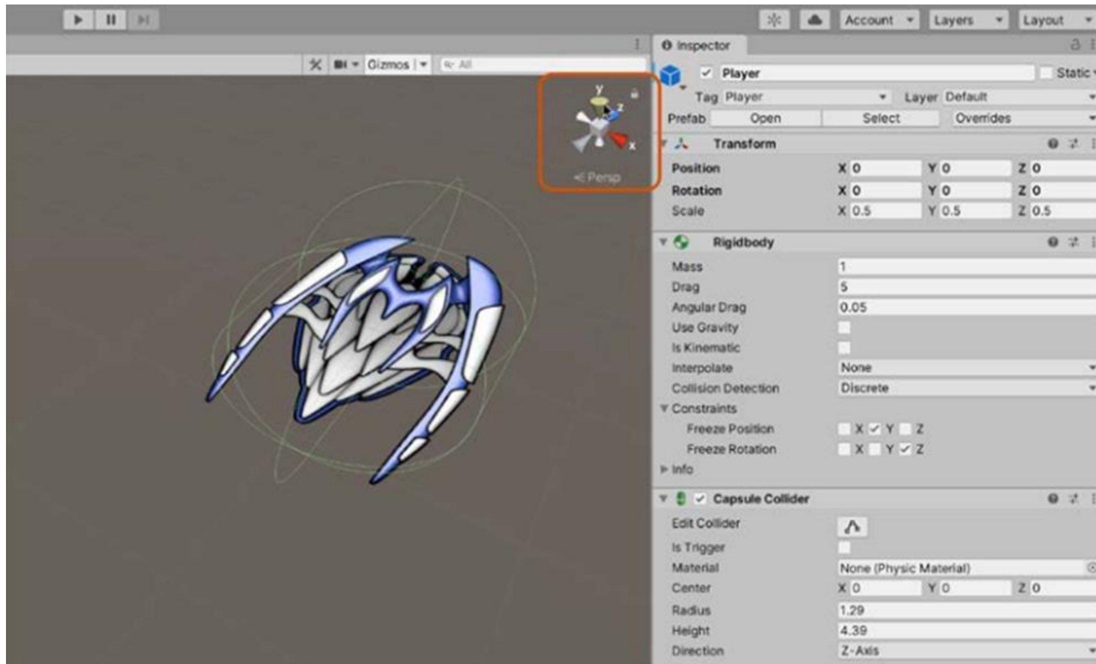



Рисунок 3.6 – Межі рівнів, що відображаються за допомогою gizmo

Якщо відредагувати прямокутник Level Bounds на об’єкті Player, то можна помітити, що Розмір gizmo автоматично регулюється відповідно до нових меж рівня.

Оскільки проект спочатку створювався як 3D-проект, його камеру не налаштовано правильно, тому потрібно перемикати камеру з 3D на 2D перспективу.

**Налаштування камер.**

Далі налаштуємо сцену та ігрову камеру. Треба зазначити, що будь-які зміни камери сцени не змінять поле гри, а змінить лише вигляд гравця.

Почнемо з камери сцени:

1. Перемикаємо вікно перегляду сцени на 2D-вид зверху вниз, клацнувши стрілку вгору у верхньому правому куті вікна перегляду сцени: (рис. 3.7). Вікно перегляду розташовано у вигляді зверху, оскільки в ньому буде показано Top як поточний вигляд.

2. Звідси можна налаштувати камеру сцени на точну відповідність камері вікна перегляду, забезпечуючи миттєвий перегляд гри зверху вниз.

Вибираємо камеру в сцені (або на панелі «Ієрархія») GameObject /Align With View з меню програми.

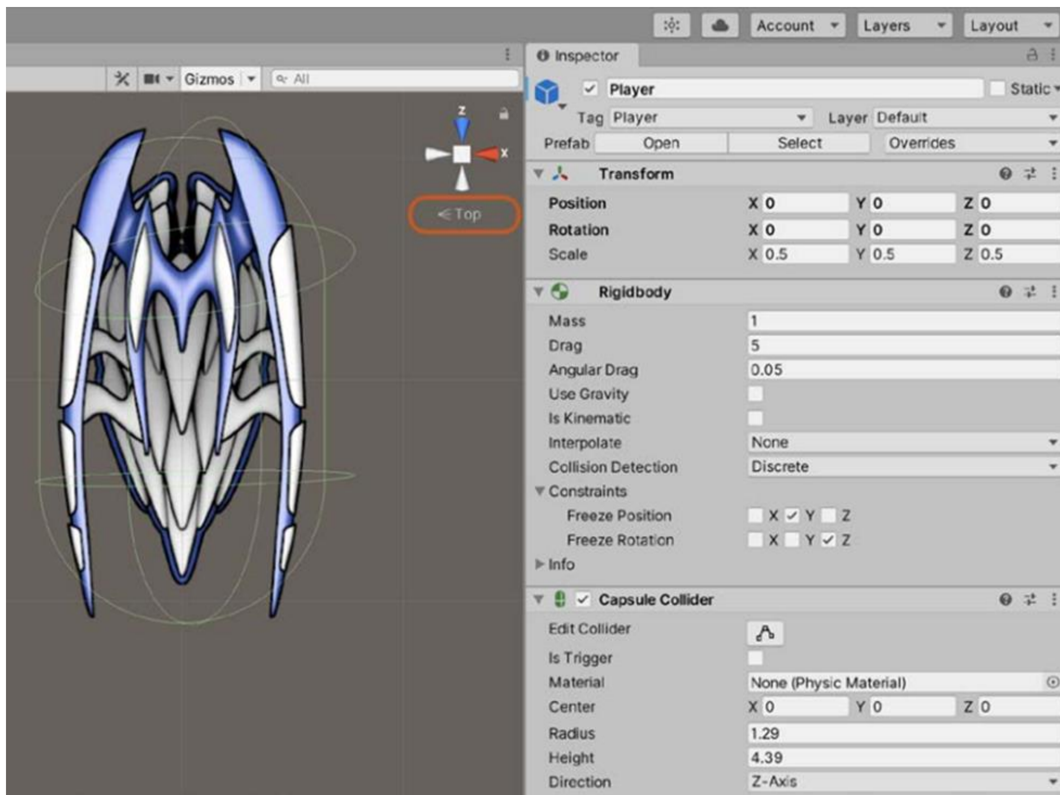



Рисунок 3.7 – Зміна перспективи вікна перегляду

Коли гра запущена, космічний корабель все ще не дивиться на курсор миші, як це було задумано. Виправити це можна змінивши камеру на ортографічну (2D) камеру:

1. Вибираємо камеру в сцені.

2. В «Інспекторі» змінюємо параметр «Проекція» з «Перспектива» на «Ортографічна».

Кожна ортографічна камера має поле розміру в інспекторі, якого немає для перспективних камер. Це поле контролює кількість одиниць у поданні світу відповідають пікселям на екрані. Встановлюємо співвідношення світових одиниць 1:1 пікселів, щоб текстури відображалися правильного розміру, а будь-який курсор руху мав очікуваний ефект. Цільовою роздільною здатністю для гри буде Full HD, який становить 1920 x 1080, і має співвідношення сторін 16:9. Для цієї резолюції встановлюємо орфографічний розмір 5,4.

**Створення компоненти Health.**

І космічному кораблю гравця, і ворогам потрібна життєздатність (health). Життєздатність - мірило характеру присутність і легітимність на сцені, зазвичай оцінюється як значення від 0 до 100. 0 означає смерть, а 100 означає повну життєздатність. Вона багато в чому специфічна для кожного випадку: гравець має свій рейтинг життєздатності, а ворог - свій. Тим не менше у них багато спільних речей з точки зору поведінки. Саме це робить сенс кодувати життєздатність як окремий компонент і клас, який можна приєднати до всіх об’єктів що її потребують. Створюємо новий клас під назвою Health, який підключаємо до гравця і всіх ворогів або об'єктів.

Клас Health підтримує життєздатність об’єкта за допомогою приватної змінної _HealthPoints. Змінну _HealthPoints оголошено як SerializedField, що дозволяє її значення бути видимим в інспекторі, зберігаючи приватну область, іншими словами, бути доступною за допомогою інших сценаріїв. З іншого боку, змінна prefab є публічною, що дозволяє побачити її значення в інспекторі та змінити в іншому місці коду при необхідності [10].

Використаємо властивості C# щоб мати можливість змінити змінну _HealthPoints з іншого сценарію та мати певну логіку для перевірки, коли вона досягає нуля.

Коли сценарій життєздатності прикріплено до космічного корабля гравця, він з’являється як компонент інспектор та містить поле для Death Particles Prefab. Це поле необов’язкове (воно може бути нульовим) і використовується для визначення системи частинок що створюються, коли життєздатність об’єкта досягає нуля [16]. Це дозволить створити ефект частинок вибуху, коли гравець помирає. Але спочатку потрібно створити вибухові частинки.

**Створення вибухових частинок.**

У грі і гравець, і вороги є космічними кораблями, тому відповідним ефектом частинок для їх знищення буде вибухова вогняна куля. Для досягнення вибухів, можна використовувати систему частинок.

Можемо створити власні системи частинок з нуля за допомогою пункту меню GameObject. Або можемо використати будь-яку готову систему частинок, що входить до Unity. У грі будемо використовувати деякі з готових систем частинок. Для цього виконаємо наступні кроки:

1. Спочатку імпортуємо пакет Unity Standard Assets.

2. Після імпорту стандартного пакету активів, імпортовані системи частинок додаємо на панель «Проект» у стандартних активах | ParticleSystems |.

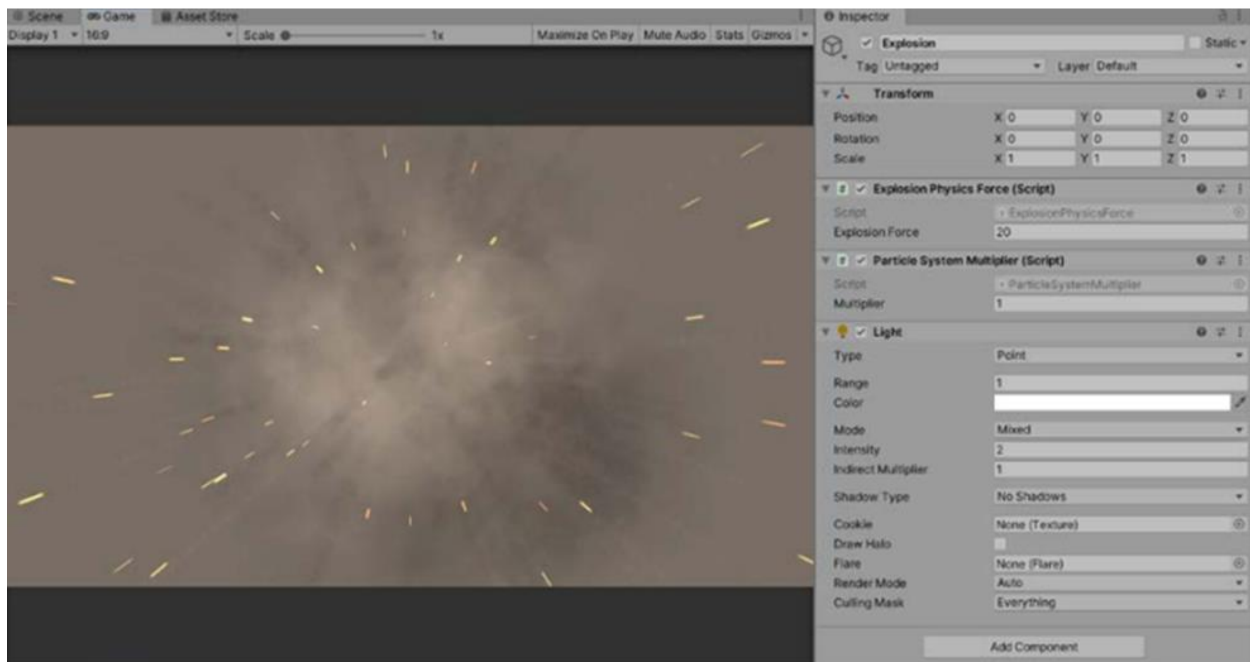



Рисунок 3.8 – Системи частинок, імпортовані на панель проекту

На рисунку 3.8 побачимо, що система вибуху включена до числа активів за замовчуванням. Щоб переконатися, що вона підходить для наших цілей, можна перевірити її, перетягнувши і скинувши вибух на сцену та натискаємо кнопку відтворення на панелі інструментів, щоб побачити вибух у дії.

У грі доступна відповідна система частинок, і можена перетягнути туди цю систему слот Death Particles Prefab у компоненті Health в Inspector. Це буде працювати технічно: коли гравець або ворог гине, запускається система вибуху.

Далі вдосконалюємо систему частинок, щоб знищити їх незабаром після створення екземпляра. до щоб досягти цього, створюємо новий сценарій C# під назвою TimedDestroy.cs.

Клас TimedDestroy знищує об’єкт, до якого він приєднаний, після вказівки інтервал (DestroyTime) минув. Скрипт простий: у функції Пуск виклик to Destroy виконується, передаючи сценарій gameObject і бажаний DestroyTime. Цей виклик не несе сюрпризів і знищить об’єкт через потрібний проміжок часу [6].

Перетягуємо сценарій TimedDestroy до системи частинок вибуху в сцені а потім натискаємо «Відтворити» на панелі інструментів, щоб перевірити, чи працює код і чи працює об’єкт, який знищено після зазначеного інтервалу.

Сценарій TimedDestroy має видалити систему частинок вибуху після затримки закінчується. Отже, створимо новий окремий префаб із цієї модифікованої версії. Зробити це, можна, виконавши наступні дії:

1. Перейменуємо систему вибуху на панелі «Ієрархія» на ExplosionDestroy.

2. Перетягнемо систему з Hierarchy на панель Project у Prefabs папку. Unity автоматично створить новий префаб, який представлятиме модифіковану система частинок.

3. Тепер перетягуємо щойно створений префаб з панелі «Проект» на «Смерть». Слот Particles Prefab на компоненті Health для гравця в Inspector. Налаштування цього поля означає, що префаб створюється, коли гравець помирає.

Далі вводимо ворогів у гру.

**Створення ворожого об'єкта.**

Вороги у грі мають форму блукаючих космічних кораблів, які з’являються на сцені через рівні проміжки часу та слідкують за гравцем, наближаючись усе ближче й ближче.

По суті, кожен ворог являє собою комбінацію кількох моделей поведінки, що працюють разом, і вони повинні бути реалізовані як окремі сценарії. Розглянемо їх по черзі:

*Життєздатність:* кожен ворог підтримує функцію життєздатності. Вороги починають сцену за визначену кількість життєздатності та знищуються, коли вона впаде нижче 0.

*Рух:* кожен ворог постійно рухатиметься по прямій лінії по передній траєкторії.

*Поворот:* кожен ворог обертатиметься та повертатиметься до гравця, навіть якщо він сам рухається. У поєднанні з функцією руху це гарантує, що ворог завжди рухається до гравця.

*Підрахунок очок:* кожен ворог нагороджує гравця оцінкою після знищення.

*Пошкодження:* вороги не можуть стріляти, але завдають шкоди гравцеві під час зіткнення.

Тепер, коли визначено діапазон поведінки, застосовної до ворога, переміщуємо ворогів на сцену.

Створимо одного конкретного ворога, а потім створимо з нього префаб і використаємо його як основу для екземпляра кілька ворогів:

1. Починаємо з вибору персонажа гравця в сцені та дублювання об’єкта Ctrl + D.

2. Перейменуємо об'єкт на Enemy і переконаємось, що він не позначений як Player, як там має бути один і тільки один об’єкт у сцені з тегом Player.

3. Тимчасово вимкнемо Player GameObject, що дозволить чіткіше зосередитися на об’єкті «Ворог» на вкладці «Сцена».

4. Вибираємо дочірній об’єкт дублюючого ворога та в інспекторі натискаємо на поле Sprite компонента Sprite Renderer, щоб вибрати новий спрайт. Вибираємо один із темніших імперських кораблів для ворожого персонажа, а спрайт оновиться для об’єкт у вікні перегляду: emy.

5. Після зміни спрайту на ворожий персонаж, його потрібно налаштувати. Значення обертання для вирівнювання спрайту щодо батьківського прямого вектора, гарантуючи, що спрайт дивитися в тому ж напрямку, що й вектор вперед.

6. Далі вибираємо батьківський об’єкт для ворога та видаляємо Rigidbody, Компоненти PlayerController і BoundsLock, але зберігаємо Health компонент, так як ворог повинен підтримувати життєздатність. Оновлену інформацію див. на

рисунку 3.9 список компонентів. Крім того, можна змінити розмір компонента Capsule Collider на кращий, тобто наблизити об'єкт противника:

Рисунок 3.9 – Налаштування обертання спрайту ворога

Тепер, коли створено основний ворожий об’єкт, можемо адаптувати його поведінку за допомогою настрою сценарію, починаючи з переміщення ворога, щоб переслідувати гравця.

**Переміщення ворога**

За сценарієм ворог повинен безперервно рухатися напрямку головного об’єкта – космічного корабля «Enterprise» з заданою швидкістю. Для цього створюємо новий сценарій під назвою Mover.cs і додаємо його до Ворожого об'єкту.

Для того щоб об’єкт переміщався із заданою швидкістю (MaxSpeed за секунду) вперед використовується компонент Transform [7].

Функція Оновлення відповідає за оновлення положення об'єкта. Це множить вектор прямого ходу на швидкість об’єкта та додає його до існуючого положення, щоб перемістити об’єкт далі вздовж лінії зору. Time.deltaTime використовується для того, щоб зробити частоту кадрів руху незалежним стилем переміщення об’єкта за секунду, а не за кадр (рисунок 3.10).

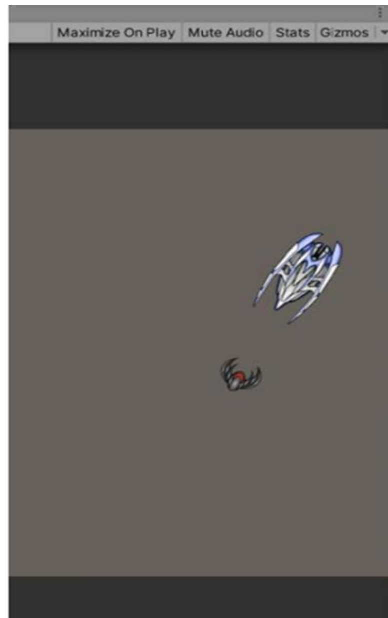



Рисунок 3.10 – Регулювання швидкості ворога

Ворог тепер рухатиметься вперед, але це не складе великого випробування для гравця, оскільки цього буде легко уникнути. Щоб збільшити складність, змусимо ворога повернутися назустріч гравцю. Поворот до гравця в поєднанні з рухом у його напрямку вперед буде створювати відповідну механіку погоні.

**Переворот ворога**

Крім руху по прямій лінії, ворог також повинен постійно повертатися передом до гравця. Щоб досягти цього, напишемо інший сценарій, який працює подібно до сценарій контролера гравця, але замість того, щоб повертатися передом до курсору, ворог повертається передом догравця. Ця функція має бути закодована в новому файлі сценарію під назвою ObjFace.cs і знову прикріпляється до ворожого об'єкта.

Сценарій ObjFace завжди обертатиме об’єкт так, щоб його вектор вказував вперед до точки призначення в сцені.

У події Awake функція FindGameObjectWithTag отримує посилання на єдиний об’єкт у сцені, позначений як Player, який має бути космічний корабель гравця. Гравець представляє місце призначення за замовчуванням для ворога об'єкт [17].

Функція Оновлення викликається автоматично один раз на кадр і генерує вектор переміщення від місця розташування об'єкта до місця призначення. Цей вектор представляє напрямок, у якому повинен дивитися об’єкт. Функція LookRotation приймає вектор напрямку та повертає об’єкт щоб вирівняти прямий вектор за заданим напрямком [8]. Це зберігає об'єкт дивлячись у бік призначення.

**Нанесення шкоди гравцеві**

За сценарієм коли ворог зіткнеться з гравцем, він повинен завдати шкоди і потенційно вбити гравця. Щоб досягти цього, необхідно зіткнення між ворогом і гравцем бути виявлені. Почнемо з конфігурації ворога. Вибираємо об’єкт «Ворог» з Інспектора, та вимкаємо прапорець Is Trigger на компоненті Capsule Collider.

Встановлення колайдера як тригера означає, що все ще можливо реагувати на події зіткнення, тому далі створюємо сценарій, який виявляє зіткнення та завдає шкоди гравцеві. Наступний код (ProxyDamage.cs) прикріплюємо до ворожого персонажа.

Сценарій ProxyDamage завдасть шкоди будь-якому об'єкту при зіткненні.

Подія OnTriggerStay викликається один раз у кожному кадрі протягом гри. Під час виконання цієї функції значення HealthPoints Компонент життєздатності зменшується на показник DamageRate, який помножується на час. deltaTime, щоб отримати шкоду за секунду (DPS) [13].

Після приєднання сценарію ProxyDamage до ворога можемо використовувати інспектор для налаштування коефіцієнта пошкодження компонента Proxy Damage. Коефіцієнт пошкодження представляє скільки життєздатності має зменшуватися у гравця за секунду під час зіткнення. Для завданням встановимо значення 100 очок життєздатності:

**Породження ворогів**

Щоб зробити рівень веселим і складним, знадобиться не просто один ворог. Насправді, для гри, яка по суті є нескінченною, потрібно буде додавати

ворогів постійно й поступово через деякий час. По суті, знадобиться або регулярне, або періодичне додавання ворогів.

Тепер створимо новий скрипт під назвою Spawner.cs, який створює нових ворогів у сцени протягом певного часу в межах заданого радіусу від космічного корабля гравця. Цей сценарій має бути приєднаний до нового порожнього GameObject на сцені.

Під час події Start функція InvokeRepeating створить екземпляри ObjToSpawn (префаб) неодноразово через указаний інтервал, виміряний у секундах. Згенеровані об’єкти будуть розміщені у випадковому радіусі від центральної точки.

Клас Spawner — це глобальна поведінка, яка застосовується до всієї сцени. Вона не залежить ні від гравця, ні будь-який конкретного ворога. З цієї причини його слід прикріпити до порожнього GameObject. Створюємо один із них, вибравши GameObject / Створити Empty з меню програми. Назизаємо новий об’єкт Spawner і приєднуємо до нього сценарій Spawner [7].

Після додавання до сцени з Інспектора перетягуємо префаб Enemy до Об’єкта Поле Spawn у компоненті Spawner. Встановлюємо інтервал на 2 секунди та збільшуємо максимальний радіус до 5 (рисунок 3.11):

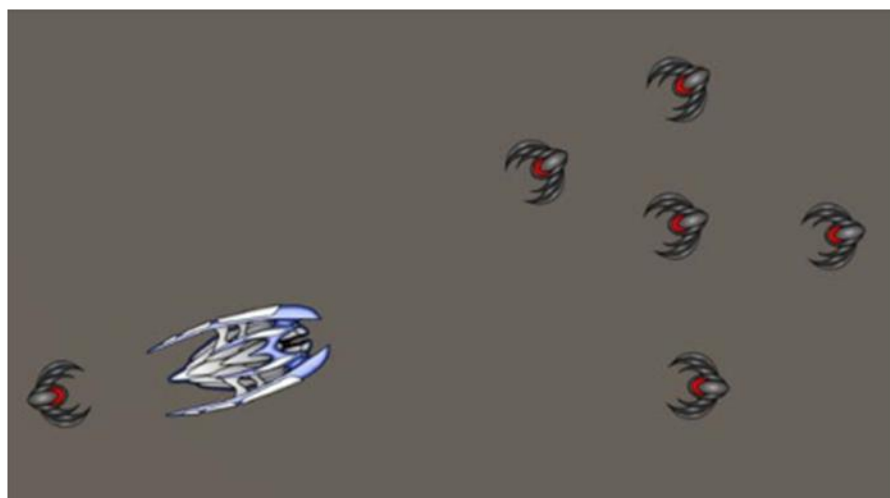


Рисунок 3.11 – Налаштування Spawner для ворожих об’єктів

Далі створимо систему підрахунку очок, де кожен переможений ворог присуджує гравцеві певний номер балів, розробимо інтерфейс для відображення очок. Та додаємо останні штрихи щоб покращити загальний вигляд гри: фонове зображення та звукові ефекти.

**Реалізація появи снарядів**

Створення префабу снаряда Коли гравець натискає кнопку вогню, космічний корабель повинен стріляти снарядами. Ці об’єкти будуть створені на базі префабу Ammo. У рамках створення префабу налаштуємо текстуру снаряда, щоб придати снаряду руху додаємо нову логіку для обробки зіткнень із ворогами. Починаємо з налаштування текстура снаряда, таким чином зможемо легше побачити, як снаряд рухається та стикається з ворогами, що допоможе усунути будь-які проблеми, коли дійдемо до налаштувань його руху.

**Налаштування текстури снаряда**

Для початку налаштуємо текстуру, яка буде використовуватися як графіка снаряда:

1. Відкриваємо папку Textures на панелі Project і вибираємо текстуру Ammo. Ця текстура містить кілька різних версій спрайту боєприпасів, вирівняних в ряд. Коли патрони стріляють показується лише одне із зображень або зображення, відтворені як анімаційна послідовність, кадр за кадром (рисунок 3.12)

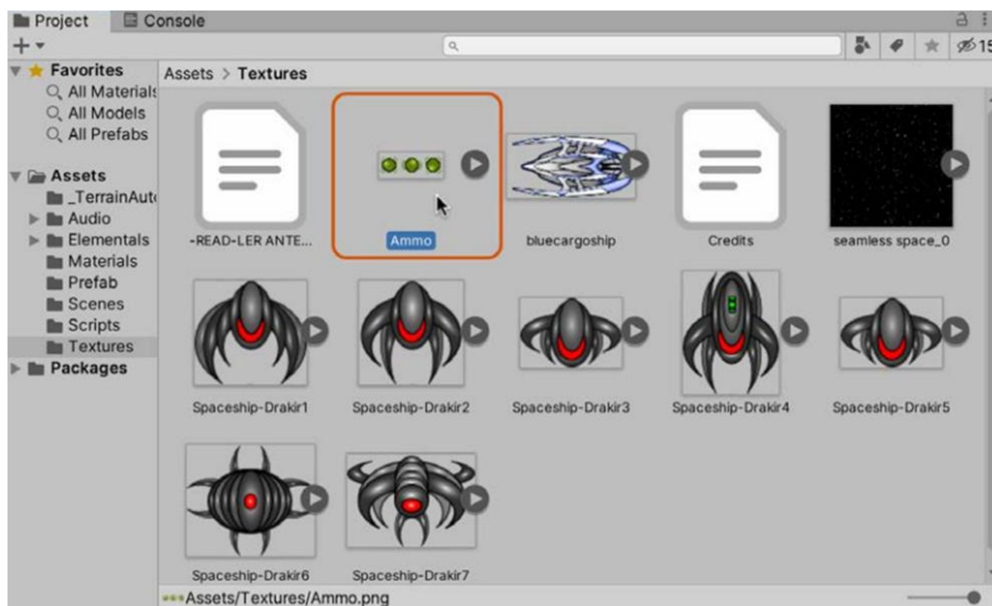


Рисунок 3.12 – Підготовка до створення префабу Ammo

2. Вибираємо текстуру на панелі «Проект».

3. У «Інспекторі» змінюємо спадне меню «Режим спрайту» з «Поодинокий» на Множинний, який інформує Unity автоматично, для того, щоб помістити більше ніж один спрайт у простір текстури.

4. Натискаємо кнопку Застосувати, щоб застосувати зміни.

5. Натискаємо кнопку «Редактор спрайтів» в інспекторі, який відкриє спрайт Вікно редактора, що дозволяє ізолювати кожен спрайт.

6. Вибираємо кожен спрайт, переконавшись, що налаштування Pivot вирівнюється по центру об’єкта.

**Додавання руху снаряда**

Оскільки раніше вже створено сценарій руху, що дозволяє снаряду бути швидким і легким. Однак потрібно приєднати сценарій руху Mover, до батьківського об'єкта для чого:

1. Створюємо новий порожній GameObject на сцені (GameObject / Create Empty from) в меню програми і перейменємо його в Ammo.

2. Робимо цей новий об’єкт батьківським для Ammo_Sprite і переконуємся, що його вектор вказує в напрямку, куди має рухатися снаряд.

3. Перетягуємо сценарій Mover.cs із панелі «Проект» до батьківського елемента Ammo за допомогою панелі «Ієрархія», щоб додати його як компонент.

4. Вибираємо об’єкт Ammo та в інспекторі змінюємо значення максимальної швидкості снаоядів у компоненті Mover до 7.

5. Далі, додаємо Box Collider до об’єкта, щоб наблизити його об’єм (Component /Фізика / Box Collider) з меню програми.

6. Натискаємо кнопку відтворення на панелі інструментів. Об’єкт Ammo вистрілює вперед.

**Налаштування фізики снаряда**

Щоб снаряди не реагували на об’єкт гравця використовуємо фізичні шари:

1. Вибираємо об’єкт Player на сцені.

2. В інспекторі натискаємо розкривне меню «Шар» і вибираємо «Додати шар».

3. Називаємо шар Player, щоб вказати, що всі об’єкти, які прикріплені до шару, пов’язані з плеєром:

4. Призначаємо як об’єкт Player у сцені, так і префаб Ammo у Project панелі до щойно створеного шару Player, вибираючи по черзі кожен і у спадному меню «Шар» і вибираємо параметр «Програвач»:

5. Виберіть також змінити дітей. Це забезпечує що всі дочірні об’єкти також пов’язані з тим самим шаром, що й батьківський.

Гравець і снаряди тепер призначені одному шару. Звідси можемо змусити всі об’єкти в одному шарі ігнорувати один одного.

6. Переходимо до Редагувати / Параметри проекту з меню програми.

7. У вікні вибираємо Фізика.

8. Глобальні налаштування фізики з’являюся у вікні Параметри проекту. На дні вікна Layer Collision Matrix відображається, як шари взаємодіють один з одним. Пересічні шари з галочкою можуть і будуть впливати один на одного. З цієї причини, знімаємо позначку для шару Player, щоб запобігти зіткненням між об’єктами на цьому шарі. Та отримуємо результат показаний на рисунку 3.13:

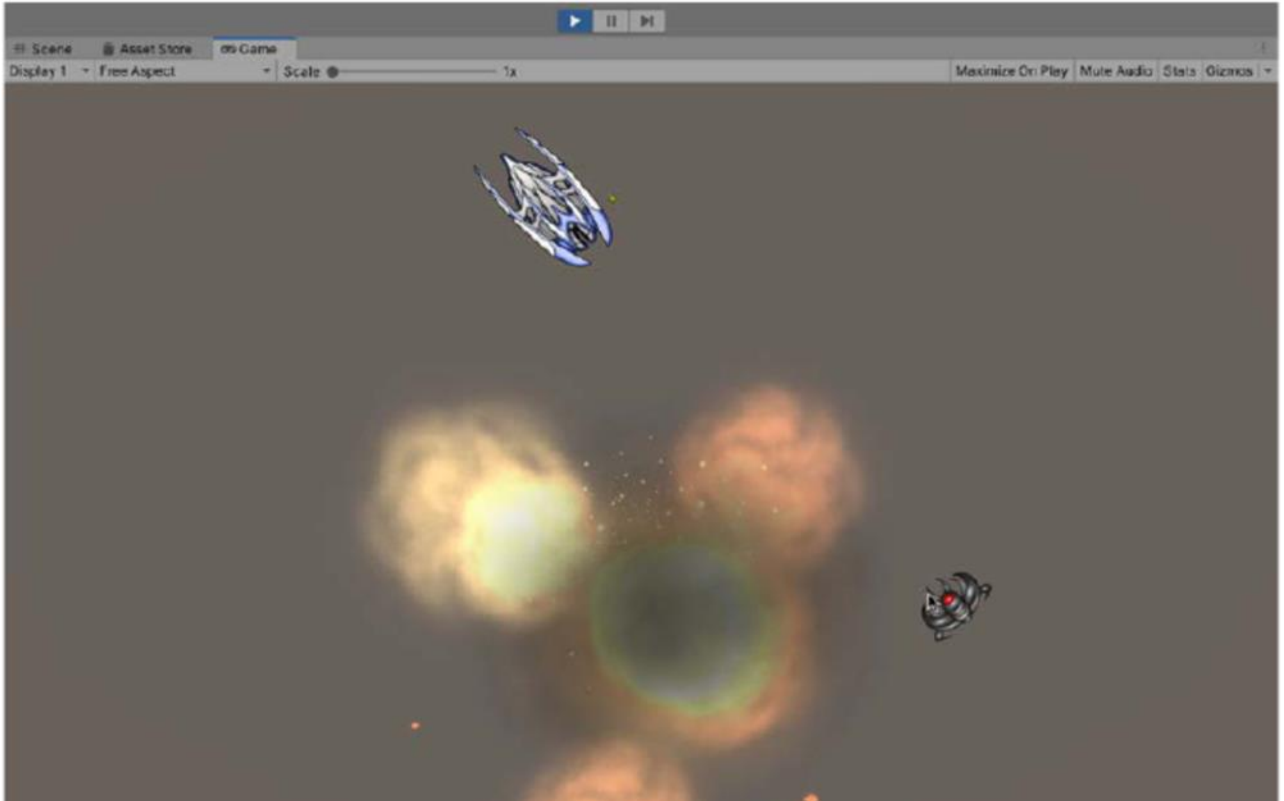



Рисунок 3.13 – Знищення ворогів за допомогою снарядів

**Створення системи нарахування балів**

Для того щоб мати можливість призначати бали кожному ворогу та щоб тали збільшувались з кожним вбивством потрібен інтерфейс користувача для відображення та потрібна логіка, щоб відстежувати поточний рахунок і вносити будь-які зміни в інтерфейс користувача. Почнемо зі створення інтерфейсу, щоб відобразити оцінку перед переходом до логіки.

Інтерфейс користувача відноситься до всіх 2D графічних елементів, які надають гравцеві інформацію:

1. Створюємо новий об’єкт UI Canvas, вибравши GameObject / UI / Полотно з в меню програми.

2. Об’єкт Canvas визначає загальну поверхню або область, на якій розміщено UI, включаючи усі кнопки, текст та інші віджети. Спочатку об’єкт Canvas може бути завеликим або занадто малим, щоб його було чітко видно у вікні перегляду, тому вибираємо об’єкт Canvas у Панель ієрархії та натисніть

клавішу F на клавіатурі, щоб сфокусувати об’єкт. Це виглядає як великий вертикально вирівняний прямокутник [11].

Сам об’єкт Canvas не відображається на вкладці Гра. Натомість він діє як контейнер. Навіть тому це сильно впливає на те, як розміщені об’єкти виглядають на екрані з точки зору розміру, положення та масштабу. Для цього налаштовуємо об’єкт Canvas:

1. Вибираємо об’єкт Canvas на сцені.

2. В інспекторі клацніть розкривне меню Режим масштабування інтерфейсу користувача під Компонент Canvas Scaler.

3. З розкривного списку виберіть опцію «Масштабувати за розміром екрана».

4. Для поля Reference Resolution вводимо 1920 для поля X і 1080 для поля Y.

**Створення логіки нарахування балів**

Щоб відобразити оцінку в щойно створеному інтерфейсі користувача, потрібно створити систему підрахунку балів код. Функціональність оцінки буде додано до загального класу GameController, відповідального за всю ігрову логіку та функції. Код для GameController включено в наступний блок коду:

Підсумовуючи, маємо наступне:

Клас GameController використовує простір імен UnityEngine. UI namespace. Це важливий клас, оскільки він надає доступ до всіх класів інтерфейсу користувача та об’єктів в Unity. Якщо не включати цей простір імен у вихідні файли, тоді не буде можливості використовувати об’єкти інтерфейсу користувача в сценарії [6].

Клас GameController містить два публічних члени Text, а саме: ScoreText і GameOverText. Вони посилаються на два текстові об’єкти, обидва з яких є необов’язковими, оскільки код GameController працюватиме нормально, навіть якщо члени нульові. ScoreText — це посилання на текстовий об’єкт інтерфейсу

користувача для відображення оцінки текст, а GameOverText використовуватиметься для відображення повідомлення після закінчення гри [6].

**Додавання фонового зображення**

Тепер приступимо до налагодження фону гри! Дотепер фон мав відображати колір фону за замовчуванням, пов’язаний з ігровою камерою. однак, оскільки дія гри розгортається в космосі, то логвчно відобразити космічний фон. Для цього потрібно:

1. Створити новий об’єкт Sprite у сцені, який відображатиме зображення простору за допомогою навігації до GameObject / 2D об'єкт / Спрайт з меню.

2. Перетягуємо космічну текстуру з панелі «Проект» у поле «Спрайт» на Компонент Sprite Renderer у сцені.

3. Повернемо об’єкт на 90 градусів навколо осі X.

4. Розташуємо об’єкт у місці початку координат (0, 0, 0).

5. Масштабуємо об’єкт, доки він не заповнить вікно перегляду, використовуючи шкалу 3 на осях X і Y.

Остаточно налаштований фон показаний на рисунку 3.14:

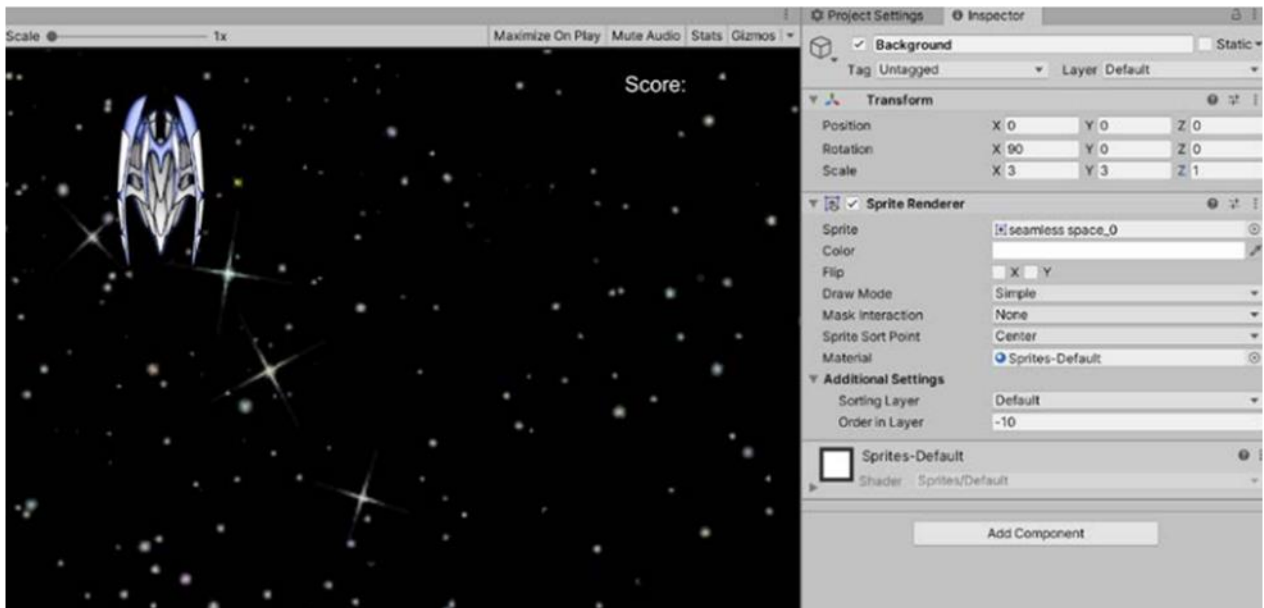



Рисунок 3.14– Налаштування фону

**Встановлення фонової музики**

Тепер, коли рівень має відповідний фон, додамо фонову музику, яка гратиме. Для цього виконаємо такі дії:

1. Вибираємо музичну доріжку на панелі «Проект» із папки «Аудіо».

2. В інспекторі встановлюємо параметр «Тип завантаження музики» на «Потокове передавання» та переконуємся, що попереднє завантаження аудіо даних вимкнено. Ці налаштування звуку покращують час завантаження оскільки Unity не потрібно буде завантажувати всі музичні дані в пам’ять, коли починається сцена.

3. Далі створюємо новий порожній GameObject у сцені та називаємо його Music.

4. Перетягуємо музичну доріжку з панелі «Проект» на об’єкт «Музика», додавши це як компонент джерела звуку. Компоненти Audio Source відтворюють звукові ефекти і музику.

5. У компоненті «Джерело аудіо» в «Інспекторі» вмикаємо «Відтворення під час пробудження», встановлюємо прапорці та цикл. Ці налаштування гарантують, що музика починається на початку рівня і безперебійно повторюється протягом усього життя сцени.

6. І нарешті, поле Spatial Blend має бути встановлено на 0, що представляє 2D.

Двовимірні звуки мають постійну гучність протягом усього рівня, незалежно від положення, оскільки 2D-звуки не розташовані в просторі. Для пострілів, кроків, вибухів та інших звуків використовуються тривимірні звуки, які існують у 3D-просторі та чия гучність має змінюватися залежно від того, наскільки близько гравець стоїть до них під час гри.

## 3.3 Тестування та діагностика гри

Заключним етапом розробки гри є її тестування, для вирішення проблем, які можуть виникнути у користувачів і для її відповідати функціональним

вимогам. Зазвичай на тестування та налагодження потрібне щоб зменшити баги та помилки витрачається дуже багато часу.

Один із способів розпочати тестування і діагностику – це скористатися панеллю статистики. Щоб відкрити цю панель, натискаємо кнопку Статистика на вкладці Гра.

Тестування відбувається з використанням Unity Test Framework, а реальне тестування на різних пристроях забезпечує оптимальне функціонування в різних умовах. Профілювання з використанням Profiler і налагодження з консоллю аналізують продуктивність, а тестування на різних ОС і роздільних здатностях, включно з бета-тестуванням, забезпечують якість і стабільність гри [17].

Профілювання - це інструмент корисний в тому випадку, якщо панель статистики вже допомогла визначити загальну проблему, таку як, наприклад, низький FPS, і потрібно копати глибше, щоб знайти місце проблеми [16].

Щоб скористатися інструментом Profiler, виконуємо такі дії:

1. Вибираємо Вікно / Аналіз / Профайлер з меню програми.

2. Відкривши вікно Profiler, натискаємо «Грати» на панелі інструментів, щоб протестувати свою гру.

Після закінчення гри, вікно Profiler заповниться кольоровими даними графіка продуктивності (рисунок 3.15) [6].

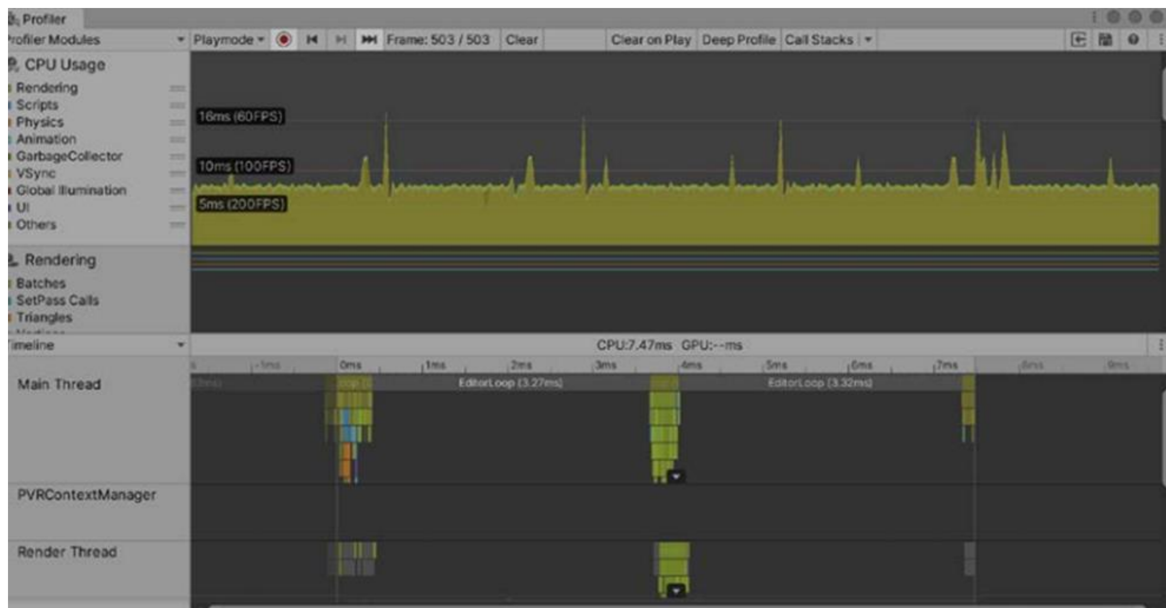



Рисунок 3.15 – Дослідження даних продуктивності за допомогою Profiler

Показники тестування за допомогою профілювання допомагають визначити такі дані, як пропускна здатність, використання пам’яті. Переривання процесора в секунду та інші.

Результати тестування гри представлені в таблиці 3.

Таблиця 3 – функціональне тестування

| № | Назва тесту | Дія тестуваньника | Очікуваний результат | Проходження тесту |
|---|---|---|---|---|
| 1 | Правильні дії при переміщенні корабля і стрілянини | При натисканні клавіш «WASD» гравець переміщується у потрібному напрямку | Гравець переміщується у правильному напрямку | Так |
| 2 | Натискання декількох клавіш при пересуванні | При одночасному натисканні клавіш «WD» гравець переміщувався або праворуч, або вгору | Гравець не повинен переміщуватись по діагоналі | Так |
| 3 | Проходження рівня | Проходження рівня завершується при ліквідації гравця, космічного корабля «Enterprise», шляхом зіткнення з ворожими об’єктами | Проходження рівня завершується при ліквідації гравця космічного корабля «Enterprise» | Так |
| 4 | Нарахування балів | Стрілянина по ворожих космічних об’єктах призводить до їх знищення і нарахування балів | Зміна кількості балів | Так |

## Висновки до III розділу

Таким чином, в третьому розділі кваліфікаційної роботи отримано гру з двомірною перспективою і 3D компонентами. Гра має один нескінченний рівень, під час якого ігровий персонаж – космічний корабель «Enterprise» летить у космічному просторі і на своєму шляху стикається з ворожими об’єктами – іншими космічними кораблями. Гравець методом стрілянини знищує ворожі об’єкти і отримує нарахування балів. Гра завершується, якщо гравця буде знищено ворожими об’єктами методом зіткнення з ними.

В процесі створення гри виконано налаштування ігрового поля, фоном якого встановлено космічний простір, космічного корабля гравця і ворожих космічних кораблів, снарядів, якими гравець знищує ворожу кораблі та налаштовано фонову музику. Створено систему підрахунку балів, де кожен переможений ворог присуджує гравцеві певний номер балів, і розроблено інтерфейс для відображення оцінки.

Крім того в цьому розділі було проведено діагностику та тестування гри. В результаті тестування за чотирма критеріями: правильні дії при переміщенні корабля і стрілянини, натискання декількох клавіш при пересуванні, проходження рівня та нарахування балів, дії гравця співпадають з очікуваним результатом.

В якості подальшої роботи над проектом планується створення головного меню гри.

# ВИСНОВКИ

В кваліфікаційній роботі був проведений огляд стану сучасної ігрової індустрії України та охарактеризовані жанри комп'ютерних ігор. Проаналізовані основні етапи розробки ігор і можливості трьох найбільш популярних ігрових рушія: Unity, Godot Engine та Unreal Ungine.

Результатом роботи стала гра з двомірною перспективою і 3D компонентами. Гра має один нескінченний рівень, під час якого ігровий персонаж – космічний корабель «Enterprise» летить у космічному просторі і на своєму шляху стикається з ворожими об'єктами – іншими космічними кораблями. Гравець методом стрілянини знищує ворожі об'єкти і отримує нарахування балів. Гра завершується, якщо гравця буде знищено ворожими об'єктами методом зіткнення з ними.

В процесі створення гри виконано налаштування ігрового поля, фоном якого встановлено космічний простір, космічного корабля гравця і ворожих космічних кораблів, снарядів, якими гравець знищує ворожу кораблі та налаштовано фонову музику. Створено систему підрахунку балів, де кожен переможений ворог присуджує гравцеві певний номер балів, і розроблено інтерфейс для відображення оцінки.

Крім того в цьому розділі було проведено діагностику та тестування гри. В результаті тестування за чотирма критеріями: правильні дії при переміщенні корабля і стрілянини, натискання декількох клавіш при пересуванні, проходження рівня та нарахування балів, дії гравця співпадають з очікуваним результатом.

В якості подальшої роботи над проектом планується створення головного меню гри.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Комп'ютерна гра [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/PC_game.

2. Класифікація ігор за жанрами [Електронний ресурс] – Режим доступу до ресурсу: https://gamesisart.ua/.

3. Найкращі ігрові рушії [Електронний ресурс] – Режим доступу до ресурсу: https://www.gamedesigning.org/career/video-game-engines.

4. Unreal Engine [Електронний ресурс] – Режим доступу до ресурсу: https://www.unrealengine.com/.

5. Cry Engine [Електронний ресурс] – Режим доступу до ресурсу: https://www.cryengine.com/.

6. Unity Game Engine [Електронний ресурс] – Режим доступу до ресурсу: https://unity.com/ua.

7. Unity Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/Manual/index.html.

8. David B. Hands-On Game Development Patterns with Unity / Baron David., 2019. – 116 c.

9. Створення 2D на Unity [Електронний ресурс] – Режим доступу до ресурсу: https://skillbox.ru/media/code/kak_sozdat_prostuyu_2d_igru_na_unity/.

10. 2D на Unity, докладний посібник [Електронний ресурс] – Режим доступу до ресурсу: http://websketches.ru/blog/2d-igra-na-unity-podrobnoyerukovodstvo-p1.

11. Пояснення про спрайти, тайли [Електронний ресурс] – Режим доступу до ресурсу: https://www.econdude.pw/2017/08/chto-takoe-graficheskij-sprajtsprite.html.

12. Hocking J. Unity in Action. Multiplatform game development in C# with Unity 5 / Joseph Hocking., 2015. – 352 c.

13. Smith G. Basic Math for Game Development with Unity 3D / G. Smith, K. Sung., 2018. – 279 c.

14. Thorn A. Mastering Unity Scripting / Alan Thorn., 2015. – 380 c.

15. Ferrone H. Mastering Unity Scripting / Harrison Ferrone., 2016. – 379 c.

16. Ferrone H. Learning C# by Developing Games with Unity 2019: Code in C# and build 3D games with Unity, 4th Edition / Harrison Ferrone., 2019. – 342 c.

17. Halpern J. Developing 2D Games with Unity: Independent Game Programming with C# / Jared Halpern., 2018. – 408 c

# ДОДАТОК А

**Лістинг програми**

**Файл PlayerController.cs**:

```
public class PlayerController : MonoBehaviour
{
public bool MouseLook = true;
public string HorzAxis = "Horizontal";
public string VertAxis = "Vertical";
public string FireAxis = "Fire1";
public float MaxSpeed = 5f;
private Rigidbody ThisBody = null;
void Awake ()
{
ThisBody = GetComponent<Rigidbody>();
}
}
```

**Функція FixedUpdate:**

```
public class PlayerController : MonoBehaviour
{
…
void FixedUpdate ()
{
float Horz = Input.GetAxis(HorzAxis);
float Vert = Input.GetAxis(VertAxis);
Vector3 MoveDirection = new Vector3(Horz, 0.0f, Vert);
ThisBody.AddForce(MoveDirection.normalized * MaxSpeed);
ThisBody.velocity = new Vector3
(Mathf.Clamp(ThisBody.velocity.x, -MaxSpeed,
MaxSpeed),
Mathf.Clamp(ThisBody.velocity.y, -MaxSpeed,
MaxSpeed),
Mathf.Clamp(ThisBody.velocity.z, -MaxSpeed,
MaxSpeed));
if(MouseLook)
{
Vector3 MousePosWorld =
Camera.main.ScreenToWorldPoint(new
Vector3(Input.mousePosition.x,
Input.mousePosition.y, 0.0f));
MousePosWorld = new Vector3(MousePosWorld.x,
0.0f, MousePosWorld.z);
Vector3 LookDirection = MousePosWorld -
transform.position;
```

```
transform.localRotation = Quaternion.LookRotation
(LookDirection.normalized,Vector3.up);
}
}
}
```

**Функція BoundsLock:**

```
public class BoundsLock : MonoBehaviour
{
public Rect levelBounds;
void LateUpdate ()
{
transform.position = new Vector3
(Mathf.Clamp(transform.position.x, levelBounds.xMin,
levelBounds.xMax), transform.position.y,
Mathf.Clamp(transform.position.z,
levelBounds.yMin, levelBounds.yMax));
}
}
```

**Функція_HealthPoints:**

```
public class Health : MonoBehaviour
{
…
public float HealthPoints
{
Get
{
 return _HealthPoints;
}
Set
{
_HealthPoints = value;
if(HealthPoints <= 0)
{
SendMessage("Die",
SendMessageOptions.DontRequireReceiver);
if(DeathParticlesPrefab != null)
{
Instantiate(DeathParticlesPrefab,
transform.position, transform.rotation);
}
if(ShouldDestroyOnDeath)
{
Destroy(gameObject);
}
```

```
}
}
}
}
```

**Функція BoundsLock:**

```
public class BoundsLock : MonoBehaviour
{
…
void OnDrawGizmosSelected()
{
const int cubeDepth = 1;
Vector3 boundsCenter = new Vector3(levelBounds.xMin +
levelBounds.width * 0.5f, 0, levelBounds.yMin +
levelBounds.height * 0.5f);
Vector3 boundsHeight = new Vector3(levelBounds.
width, cubeDepth, levelBounds.height);
Gizmos.DrawWireCube(boundsCenter, boundsHeight);
}
}
```

**Клас Health:**

```
public class Health : MonoBehaviour
{
public GameObject DeathParticlesPrefab = null;
public bool ShouldDestroyOnDeath = true;
[SerializeField] private float _HealthPoints = 100f;
}
```

**Файл TimedDestroy.cs:**

```
public class TimedDestroy : MonoBehaviour
{
public float DestroyTime = 2f;
void Start ()
{
Destroy(gameObject, DestroyTime);
}
}
```

**Файл Mover.cs:**

```
public class Mover : MonoBehaviour
{
public float MaxSpeed = 10f;
void Update ()
{
transform.position += transform.forward * MaxSpeed *
Time.deltaTime;
```

```
}
}
```

**Файл ObjFace.cs:**

```
public class ObjFace : MonoBehaviour
{
public Transform ObjToFollow = null;
public bool FollowPlayer = false;
void Awake ()
{
if(!FollowPlayer)
{
return;
}
GameObject PlayerObj =
GameObject.FindGameObjectWithTag("Player");
if(PlayerObj != null)
{
ObjToFollow = PlayerObj.transform;
}
}
void Update ()
{
if(ObjToFollow==null)
{
return;
}
//Get direction to follow object
Vector3 DirToObject = ObjToFollow.position -
transform.position;
if(DirToObject != Vector3.zero)
{
transform.localRotation = Quaternion.LookRotation
(DirToObject.normalized,Vector3.up);
}
}
}
```

**Файл ProxyDamage.cs:**

```
public class ProxyDamage : MonoBehaviour
{
//Damage per second
public float DamageRate = 10f;
void OnTriggerStay(Collider Col)
{
Health H = Col.gameObject.GetComponent<Health>();
```

```
if(H == null)
{
return;
}
H.HealthPoints -= DamageRate * Time.deltaTime;
}
}
```

**Файл Spawner.cs:**

```
public class Spawner : MonoBehaviour
{
public float MaxRadius = 1f;
public float Interval = 5f;
public GameObject ObjToSpawn = null;
private Transform Origin = null;
void Awake()
{
Origin = GameObject.FindGameObjectWithTag ("Player").transform;
}
void Start ()
{
InvokeRepeating("Spawn", 0f, Interval);
}
void Spawn ()
{
if(Origin == null)
{
return;
}
Vector3 SpawnPos = Origin.position + Random.
onUnitSphere * MaxRadius;
SpawnPos = new Vector3(SpawnPos.x, 0f, SpawnPos.z);
Instantiate(ObjToSpawn, SpawnPos, Quaternion.
identity);
}
}
```

**Клас GameController:**

```
public class GameController : MonoBehaviour
{
public static GameController ThisInstance = null;
public static int Score;
public string ScorePrefix = string.Empty;
public Text ScoreText = null;
public Text GameOverText = null;
void Awake()
{
ThisInstance = this;
}
void Update()
{
if(ScoreText!=null)
```

```
{
ScoreText.text = ScorePrefix + Score.ToString();
}
}
public static void GameOver()
{
if(ThisInstance.GameOverText!=null)
{
ThisInstance.GameOverText.gameObject.
SetActive(true);
}
}
}
```