

Міністерство освіти і науки України  
Державний заклад  
«Луганський національний університет імені Тараса Шевченка»

Навчально-науковий інститут математики та інформаційних технологій

Кафедра інформаційних технологій та систем

**Леонова Катерина Геннадіївна**

**АНАЛІЗ ТА РОЗРОБКА КОМП'ЮТЕРНОЇ ГРИ В ЖАНРІ 2D  
ПЛАТФОРМЕРА НА БАЗІ ІГРОВОГО РУШІЯ UNITY**

**кваліфікаційна робота**

**здобувача вищої освіти другого (магістерського) рівня**

**освітньої програми «Мультимедійні системи»**

**за спеціальністю 121 Інженерія програмного забезпечення**

Особистий підпис \_\_\_\_\_ Катерина ЛЕОНОВА

Науковий керівник \_\_\_\_\_ Володимир ДОНЧЕНКО,  
старший викладач кафедри  
інформаційних технологій та систем

Завідувач кафедри \_\_\_\_\_ Микола СЕМЕНОВ,  
кандидат педагогічних наук, доцент  
кафедри інформаційних технологій  
та систем

Полтава – 2025

## АНОТАЦІЯ

**Леонова К. Г.**

**Тема:** Аналіз та розробка комп'ютерної гри в жанрі 2D платформера на базі ігрового рушія Unity.

**Спеціальність:** 121 «Інженерія програмного забезпечення».

**Установа:** ЛНУ імені Тараса Шевченка, 2025р.

**Магістерська робота містить:** 81 с., 36 рис., 7 табл., 2 додатка, 31 джерело.

**Об'єкт дослідження** – гра у жанрі 2D платформер.

**Предмет дослідження** - технології розробки гри у жанрі 2D платформер.

**Мета роботи** - аналіз та розробка комп'ютерної гри в жанрі 2D платформера на базі ігрового рушія Unity.

**Результати роботи** – для досягнення мети було проаналізовано жанрову класифікацію ігор, проаналізовані сучасні представники цього жанру, а також були досліджені різні середовища розробки ігор та ігрові рушії. На основі цього було реалізовано прототип двовимірного платформера. У ході розробки були написані скрипти, необхідні для роботи гри, та створенні або запозиченні графічні та звукові матеріали. Для розробки відповідної програми використано ігровий рушій Unity 2019, що використовує мову C# та засобом програмування алгоритму є середовище розробки Visual Studio 2019.

**Ключові слова:** ГРА, UNITY3D, UI, ГРАФІЧНИЙ ІНТЕРФЕЙС, ДВИЖОК, 2D ПЛАТФОРМЕР, СЕРЕДОВИЩЕ РОЗРОБКИ, C#

## ANNOTATION

**Leonova Kateryna**

**Theme:** Analysis and development of a computer game in the 2D platformer genre based on the Unity game engine.

**Speciality:** 121 "Software Engineering".

**Institution:** Luhansk Taras Shevchenko National University (LTSNU), 2025 year.

**Bachelor work of:** 81 p., 36 im, 7 table, 2 ap., 31 sources.

**A research object of:** - 2D platformer game.

**The article of research-** 2D platformer game development technologies.

**An aim of research is** - analysis and development of a computer game in the 2D platformer genre based on the Unity game engine.

**Job performanes.-** To achieve the goal, the genre classification of games was analyzed, modern representatives of this genre were analyzed, and various game development environments and game engines were also studied. Based on this, a prototype of a two-dimensional platformer was implemented. During the development, scripts necessary for the game to work were written, and graphic and sound materials were created or borrowed. To develop the corresponding program, the Unity 2019 game engine was used, which uses the C# language and the Visual Studio 2019 development environment is the algorithm programming tool.

**Keywords:** GAME, UNITY3D, UI, GRAPHICAL INTERFACE, ENGINE, 2D PLATFORMER, DEVELOPMENT ENVIRONMENT, C#.

## ЗМІСТ

<b>ВСТУП .....</b>	<b>6</b>
<b>РОЗДІЛ 1. ДОСЛІДЖЕННЯ КЛАСИФІКАЦІЙ ТА АНАЛІЗ ІСНУЮЧИХ РОЗРОБОК КОМП'ЮТЕРНИХ ІГОР .....</b>	<b>10</b>
1.1. Аналіз та класифікація комп'ютерних ігор .....	10
1.2. Аналіз жанру платформерів .....	14
1.2.1. Dead Cells .....	17
1.2.2. Cuphead .....	18
1.2.3. Mark of the Ninja .....	20
1.2.4. Starbound .....	21
Висновки до розділу .....	23
<b>РОЗДІЛ 2. ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ КОМП'ЮТЕРНОЇ ГРИ В ЖАНРІ 2D ПЛАТФОРМЕРА .....</b>	<b>25</b>
2.1. Порівняльна характеристика ігрових движків .....	25
2.1.1. Unity .....	26
2.1.2. Torque 2D/3D .....	28
2.1.3. CryEngine 3 .....	30
2.1.4. UDK .....	32
2.1.5. Frostbite Engine .....	34
2.2. Архітектура, моделювання та проєктування гри .....	36
2.2.1. Створення Usecase .....	36
2.2.2. Діаграми варіантів використання .....	38
2.2.3. Діаграма класів .....	40
2.2.4. Діаграми станів та переходів .....	42
Висновки до розділу .....	44
<b>РОЗДІЛ 3. ОСНОВНІ ЕТАПИ РЕАЛІЗАЦІЇ ІГРОВОГО ПРОЄКТУ .....</b>	<b>45</b>
3.1. Візуальне оформлення: вибір та використання графічних активів .....	45
3.1.1. Створення анімованого персонажа: астронавт .....	46
3.1.2. Спрайти інтерфейсу .....	46
3.1.3. Візуальне представлення противників: спрайт літаючої тарілки ..	48

3.2. Проєктування гри .....	49
3.2.1. Імплементация фізичних властивостей ігрових об'єктів .....	49
3.2.2. Рух об'єктів .....	50
3.2.3. Створення анімацій для персонажа .....	53
3.2.4. Реалізація вогнепальної зброї .....	54
3.2.5. Гибель та відродження героя .....	57
3.2.6. Штучний інтелект ворогів: навігація та поведінка .....	59
3.2.7. Графічний інтерфейс для героя та ворога.....	61
3.2.8. Взаємодія між об'єктами .....	62
3.2.9. Генерація хвиль ворогів.....	64
3.2.10. Графічний інтерфейс до генератору хвиль.....	66
3.2.11. Умови для завершення гри .....	68
3.2.12. Меню початку та кінця гри .....	70
3.3. Інструкція користувача.....	72
3.3.1. Стартове меню .....	72
3.3.2. Інтерфейс та ігровий процес .....	72
3.3.3. Умови та меню завершення гри.....	75
Висновки до розділу .....	75
<b>ВИСНОВКИ.....</b>	<b>77</b>
<b>СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....</b>	<b>79</b>
<b>ДОДАТОК А .....</b>	<b>82</b>
<b>ДОДАТОК Б .....</b>	<b>85</b>

## ВСТУП

На сьогоднішній день мобільні та ігрові технології досягли того рівня, що при використанні смартфонів, планшетів, всіляких гаджетів, стали невід'ємною частиною сучасного суспільства. Не можна залишити без уваги ігрову індустрію. Для створення ігор використовується безліч різних технологій, мов, додатків і платформ. Кожна з них має свої плюси і мінуси. Без стільникового зв'язку сучасне людство не може уявити собі й дня. І буквально зовсім недавно головними функціями були тільки дзвінки та відправка повідомлень. Однак сьогодні пріоритети дещо змінюються. Телефон став далеко не просто засобом для комунікації людей, але і багатофункціональним пристроєм, що дозволяє пізнавати світ, проводити дозвілля і навіть заробляти гроші. І, звичайно, все це стало можливо тільки завдяки розвитку Інтернету для мобільних пристроїв і створених з цієї нагоди додатків. У XXI столітті почався бурхливий розвиток ринку мобільного контенту, зокрема, мобільних додатків. Здешевити мобільний інтернет трафік дозволило поява нових технологій - GPRS і EDGE. Користувачі починають викачувати із Всесвітньої мережі величезну кількість картинок, ігор, рингтонів і пр.[25]

Мобільні ігри набули небувалої популярності за останні роки та посіли почесне місце на ринку розваг та дозвілля. Для задоволення потреб розробників було створено величезну кількість інструментальних засобів та продовжується розробка нових. Загальна ціль всіх інструментальних засобів - полегшення та покращення розробки, використання передових технологій обробки графіки, фізики, забезпечення кросплатформенності розроблених проєктів.

Додаток Unity являє собою професійний ігровий двигун, який використовується в створенні відеоігор для різних платформ. Це інструмент, яким щодня користуються досвідчені розробники, а також один із доступніших інструментів для новачків. До недавнього часу людина, яка вирішила навчитися програмуванню ігор (особливо тривимірних), відразу ж стикався з безліччю серйозних перешкод, в той час як інструмент Unity дозволив значно полегшити життя новачкам [1].

Створення відеоігор в своїй основі не відрізняється від написання будь-якого іншого ПО. Здебільшого відмінності виявляються в кількісній площині. Гра більш інтерактивна, ніж більшість веб-сайтів, а отже, потрібно зовсім інший тип коду, але при цьому в обох випадках будуть задіяні подібні навички і процеси [1].

Будь-який ігровий двигун надає безліч функціональних можливостей, які застосовуються в різних іграх. Реалізована на цьому двигуну гра отримує всі ці функціональні можливості, крім того, додаються її власні ігрові ресурси і код ігрового сценарію. Unity пропонує моделювання фізичних середовищ, карти нормалей, загородження навколишнього світу в екранному просторі (Screen Space Ambient Occlusion, SSAO), динамічні тіні, тощо [1].

Подібний набір функціональних можливостей є в багатьох ігрових двигунів, але у Unity є дві основні переваги перед іншими передовими інструментами розробки ігор: продуктивний візуальний робочий процес і потужна міжплатформенна підтримка [1].

Візуальний робочий процес виділяє даний інструмент з-посеред інших середовищ розробки ігор. Частина інструментів розробки ігор часто являють собою сукупність розрізнених частин, які потрібно контролювати, чи бібліотеку, для роботи з якою потрібно налаштовувати власне інтегроване середовище розробки, процес збирання проєкту, тощо. Робочий процес в Unity прив'язаний до ретельно продуманого візуального редактору. Редактор надає можливість компоувати сцени майбутньої гри, пов'язуючи ігрові ресурси і код в інтерактивні об'єкти. Він дозволяє швидко і раціонально створювати професійні гри, забезпечуючи високу продуктивність праці розробників і надаючи в їх розпорядження вичерпний перелік сучасних технологій в області відеоігор. Редактор дозволяє редагувати об'єкти в редакторі і рухати елементи в сцені при запусненій грі. Unity надає можливість налаштовувати редактор за допомогою сценаріїв, що додають нові функціональні можливості і елементи меню до інтерфейсу [1].

Unity підтримує наступні платформи: Windows, OSX, Linux, Android,

WebGL, iOS, BlackBerry, Tizen, Xbox One, Xbox 360, PS3, PS4, PS Vista, Windows Store, Samsung TV [1].

**Об'єкт дослідження** – гра у жанрі 2D платформер.

**Предмет дослідження** - технології розробки гри у жанрі 2D платформер.

**Мета роботи** - аналіз та розробка комп'ютерної гри в жанрі 2D платформера на базі ігрового рушія Unity.

Відповідно до мети роботи необхідно вирішити наступні **завдання**:

1. Проаналізувати актуальні підходи до створення відеогри;
2. Виконати аналіз сучасних засобів розробки комп'ютерних ігор та досвіду їх застосування.
3. Зробити порівняльний аналіз актуальних ігрових рушіїв та їх функціоналу.
4. Розробити комп'ютерну гру в жанрі 2D платформера на базі ігрового рушія Unity.

В першому розділі містяться загальні відомості про комп'ютерні ігри та їх жанрову класифікацію. Крім цього, було проаналізовано основні переваги та підсумовано недоліки існуючих сучасних прототипів.

Другий розділ містить у собі опис вибраного жанру проєкту, а також аналіз існуючих методів і функціоналу розробки ігор, та обґрунтування вибраного ігрового рушія для розробки. Крім цього було описано повний алгоритм створення гри, проаналізовано потенційну аудиторію гравців та загальну актуальність проєкту, а також сформульовано мету розробки та функціонал гри, якому вона повинна дотримуватися. Проведено моделювання гри.

У третьому розділі описана повна поетапна реалізація проєкту. Спершу було визначено основні поняття стосовно візуальної складової проєкту, а потім описано основні графічні компоненти та елементи які використовувались при розробці гри. Крім цього було детально описано кожен етап розробки з докладним описом кожного важливого методу та класу, який використовувався при написанні коду. Також було описано створення меню початку та завершення гри. Описано інструкцію користувача, або потенційного гравця. Докладно



описано кожен крок, який повинен виконати гравець, а також кожен елемент інтерфейсу. Описана бойова система, мета, умови для перемоги чи програшу, та робота з меню початком і кінцем гри.

# **РОЗДІЛ 1. ДОСЛІДЖЕННЯ КЛАСИФІКАЦІЙ ТА АНАЛІЗ ІСНУЮЧИХ РОЗРОБОК КОМП'ЮТЕРНИХ ІГОР**

## **1.1. Аналіз та класифікація комп'ютерних ігор**

Комп'ютерна гра – це програмний продукт, що імітує взаємодію користувача з віртуальним середовищем за певними правилами. Взаємодія здійснюється за допомогою введення даних через периферійні пристрої (клавіатура, миша, геймпад тощо), а візуалізація ігрового процесу відбувається на екрані дисплея. Комп'ютерні ігри можуть бути як індивідуальними, так і багатокористувацькими, передбачаючи взаємодію гравців в мережі.

Існують різноманітні підходи до класифікації комп'ютерних ігор. За жанром ігри поділяють на ролеві, стратегічні, шутери, симулятори тощо. За платформою – на ПК-ігри, консольні ігри, мобільні ігри. За віковою категорією – на ігри для дітей, підлітків та дорослих. Важливим критерієм класифікації є також механіка гри, тобто набір правил та дій, доступних гравцеві.

Комп'ютерні ігри часто черпають натхнення з інших видів мистецтва, таких як література, кіно, музика. Ігрові сюжети можуть бути засновані на літературних творах або фільмах, а ігрові світи можуть бути детально пропрацьовані на основі існуючих міфологій та культур. Водночас, успішні комп'ютерні ігри можуть стати основою для створення нових творів мистецтва, таких як книги, фільми, мультфільми, музичні композиції. Цей взаємозв'язок свідчить про те, що комп'ютерні ігри є невід'ємною частиною сучасної культури.

Комп'ютерні ігри відіграють все більш значущу роль у сучасному суспільстві, виходячи за рамки розважальної функції. Вони активно використовуються в освітньому процесі та наукових дослідженнях. Спеціально розроблені навчальні ігри дозволяють ефективно засвоювати нові знання та навички, а ігрові платформи можуть слугувати інструментом для проведення експериментів та збору даних.

Популярність комп'ютерних ігор призвела до виникнення кіберспорту – змагань з відеоігор, які за масштабом та організацією не поступаються традиційним спортивним змаганням. Визнання кіберспорту як офіційного виду

спорту в деяких країнах свідчить про зростання його престижу та соціальної значущості.

Тенденція до гейміфікації – впровадження ігрових елементів у неігрові програми – є ще одним свідченням впливу комп'ютерних ігор на різні сфери життя. Гейміфікація дозволяє підвищити залученість користувачів, мотивацію до навчання та продуктивність праці.

Таким чином, комп'ютерні ігри трансформуються з розваги в потужний інструмент, який використовується в освіті, науці, бізнесі та інших сферах. Їхній вплив на суспільство є багатограним і продовжує зростати. Офіційне визнання комп'ютерних ігор як виду мистецтва та спорту підтверджує їхню культурну та соціальну значимість.

Незважаючи на значний прогрес у галузі розробки комп'ютерних ігор, проблема їхньої класифікації залишається актуальною. Відсутність чітко визначених і універсально прийнятих критеріїв призводить до неоднозначного віднесення ігор до певних жанрів та розбіжностей у класифікаціях, представлених у різних джерелах.

Одним з найпоширеніших підходів до класифікації комп'ютерних ігор є аналіз домінуючих дій, які виконує гравець. За цим критерієм ігри можна розділити на три основні категорії:

*Ігри контролю:* характеризуються необхідністю управління віртуальним світом, будівництва, стратегічного планування та прийняття рішень.

*Ігри дії:* орієнтовані на активні дії гравця, такі як біг, стрибки, стрільба, рукопашні бої.

*Ігри інформації:* передбачають збір, обробку та використання інформації для досягнення поставленої мети.

Для більш детальної класифікації комп'ютерних ігор доцільно використовувати аналіз геймплейних елементів. Було виділено 15 основних елементів, які можуть бути присутні в різних комбінаціях в різних іграх: навчання, загадки, спілкування, роль, вивчення, збирання, ухилення, знищення,

змагання, техніка, турбота, розвиток, контроль, тактика, планування. Кожен жанр характеризується переважанням певних елементів (таблиця 1.1).

Таблиця 1.1

### Класифікація комп'ютерних ігор

Категорія	Основні геймплейні елементи	Типові механіки	Приклади жанрів	Приклади ігор	Рівень складності	Цільова аудиторія
Ігри інформації	Аналіз, стратегія, розв'язання задач	Пазли, головоломки, квести	Логічні ігри, головоломки, текстові квести	Portal, The Witness, Myst	Високий	Широка аудиторія
Ігри дії	Реакція, швидкість, координація	Стрілянина, біг, стрибки, рукопашний бій	Шутери, платформери, файтинги	Doom, Mario, Street Fighter	Середній-високий	Молодь
Ігри контролю	Планування, будівництво, управління ресурсами	Стратегія в реальному часі, стратегія по черзі, симулятори	RTS, 4X, симулятори будівництва	StarCraft, Civilization, Cities: Skylines	Високий	Досвідчені гравці

**Ігри інформації** характеризуються пріоритетом когнітивних компонентів над фізичними діями гравця. Отримання, обробка та використання інформації є ключовими аспектами геймплею в таких іграх. Хоча в деяких випадках можуть бути присутні елементи планування та динаміки, саме інформація виступає основним рушієм ігрового процесу. Типовим представником цього жанру є рольові ігри (RPG), які занурюють гравця в детально пропрацьований ігровий світ, дозволяючи йому приймати рішення, розвивати персонажа та взаємодіяти з оточенням.

**Ігри дії** фокусуються на фізичній активності гравця та його реакції на зовнішні подразники. Головною метою є виконання певних дій у динамічному середовищі. Типовим представником цього жанру є екшени (action games), які вимагають від гравця швидких рефлексів та координації рухів. Саме в іграх цього жанру часто проводяться змагання, оскільки вони дозволяють порівняти швидкість реакції та вміння гравців.

**Ігри контролю** орієнтовані на стратегічне планування та управління ресурсами. Гравці в таких іграх приймають рішення, які впливають на розвиток ситуації в ігровому світі. До цього жанру належать стратегії в реальному часі

(RTS), глобальні стратегії (4X), економічні симулятори та інші. Головною метою в таких іграх є досягнення переваги над суперниками або досягнення певних цілей шляхом ефективного використання доступних ресурсів.

Комп'ютерні ігри можна класифікувати за різними критеріями, одним з яких є кількість учасників. За цією ознакою виділяють два основних типи ігор: одиночні та мультиплеєрні.

Одиночні ігри передбачають участь лише одного гравця. В таких іграх роль опонентів зазвичай виконує штучний інтелект. Мета гравця може полягати в проходженні сюжетної лінії, накопиченні ресурсів, розвитку персонажа або комбінації цих цілей.

Мультиплеєрні ігри дозволяють одночасно грати кільком гравцям. Залежно від способу організації мультиплеєру, ігри можна поділити на такі категорії:

- Мультиплеєр на одному комп'ютері: декілька гравців грають на одному пристрої, використовуючи різні контролери.
- Мультиплеєрні оффлайн-ігри: гравці з'єднуються між собою за допомогою локальної мережі.
- Масові онлайн-ігри (ММО): велика кількість гравців одночасно взаємодіє в загальному віртуальному світі.

Останнім часом спостерігається тенденція до комбінування одиночного та мультиплеєрного режимів в одній грі. Це дозволяє задовольнити потреби різних категорій гравців та продовжити інтерес до гри після проходження основної сюжетної лінії.

Одним з ключових критеріїв класифікації комп'ютерних ігор є візуальне представлення ігрового світу. За цією ознакою ігри можна поділити на такі категорії:

**Текстові ігри.** Характеризуються мінімальним використанням графічних елементів. Взаємодія з гравцем відбувається переважно за допомогою тексту. Такі ігри були популярними на ранніх етапах розвитку комп'ютерних ігор і досі мають своїх прихильників.

**Двовимірні ігри (2D).** Усі елементи ігрового світу представлені у вигляді плоских зображень. Цей формат дозволяє створювати стилізовану графіку та є відносно невибагливим з точки зору обчислювальних ресурсів.

**Тривимірні ігри (3D).** Забезпечують більш реалістичне та деталізоване зображення ігрового світу. Використання тривимірної графіки дозволяє створювати складні інтерактивні середовища та забезпечувати високий рівень занурення гравця.

Після аналізу існуючих класифікацій комп'ютерних ігор було прийнято рішення розробити двовимірну гру в жанрі платформер. Цей вибір обумовлений кількома факторами. Ігри жанру платформер мають відносно просту і зрозумілу механіку, що дозволяє сфокусуватися на розробці цікавого ігрового процесу, а не на складних алгоритмах. Розробка двовимірної гри зазвичай вимагає менших ресурсів, ніж розробка тривимірної гри. Це дозволяє залучити меншу команду розробників та скоротити час розробки. Платформери є популярним жанром серед гравців різного віку, що забезпечує потенційну аудиторію для розробленої гри.

## **1.2. Аналіз жанру платформерів**

Платформери є одним з найдавніших і найпопулярніших жанрів комп'ютерних ігор, що характеризуються акцентом на фізичній взаємодії гравця з ігровим середовищем. Основним елементом геймплею платформерів є виконання акробатичних трюків, таких як стрибки, перестрибування перешкод та маніпуляції з елементами ландшафту. Цей жанр визначається необхідністю точного контролю над рухами персонажа в динамічному дво- або тривимірному просторі.

Типовим для платформерів є присутність колекційних предметів, які можуть надавати персонажу додаткові здібності або відкривати нові рівні. Механіка збору предметів нерідко пов'язана з дослідженням ігрового світу. В класичних платформерах збір предметів зазвичай здійснювався простим дотиком до них, проте сучасні ігри пропонують більш складні механіки збору, що вимагають від гравця виконання певних дій або розгадування головоломок.

Противники в платформерах, як правило, мають обмежений набір дій і часто не відрізняються високим рівнем штучного інтелекту. Їхня основна функція – створювати перешкоди для гравця та урізноманітнювати геймплей. Однак, в сучасних платформерах можна зустріти більш складних противників, які здатні адаптуватися до дій гравця та використовувати різноманітні тактики.

Одним з ключових елементів геймплею в платформерах є взаємодія гравця з ворожими сутностями. Конфлікт з противниками зазвичай призводить до втрати очок здоров'я або смерті персонажа. Поведінка ворогів може варіюватися від пасивного переслідування до активної атаки. Деякі противники здатні змінювати свою поведінку залежно від дій гравця або стану навколишнього середовища.

Для подолання противників гравці використовують різноманітні зброю та прийоми. В класичних платформерах основним способом боротьби було фізичне знищення ворогів шляхом стрибків на них. Сучасні платформери пропонують більш широкий арсенал засобів для боротьби з противниками, включаючи вогнепальну зброю, холодну зброю, магію та інші спеціальні здібності.

Рівневий дизайн платформів передбачає створення складних і цікавих маршрутів для проходження. Для урізноманітнення геймплею розробники використовують різноманітні елементи рівневого дизайну, такі як:

- 1) Приховані проходи: сховані шляхи, що дозволяють гравцеві знайти секретні кімнати або обійти складні ділянки рівня.
- 2) Колекційні предмети: предмети, розкидані по рівню, які необхідно зібрати для досягнення певних цілей або отримання додаткових можливостей.
- 3) Пастки та небезпеки: елементи, що створюють додаткові труднощі для гравця і вимагають від нього обережності та швидкості реакції.

Платформери часто відрізняються яскравим і стилізованим візуальним оформленням. Використання мальованої графіки, анімації та звукових ефектів дозволяє створити унікальну атмосферу і занурити гравця в ігровий світ.

Персонажі в платформерах можуть бути як типовими представниками різних жанрів фентезі (ельфи, орки, дракони), так і оригінальними авторськими персонажами. Вибір персонажа залежить від тематики гри та її сюжету.

Жанр платформерів зародився на початку 1980-х років і з того часу зазнав значних змін. Серед піонерських робіт у цій галузі можна виділити ігри "Pitfall" та "Super Mario Bros". Саме ці проєкти заклали фундамент для подальшого розвитку жанру, визначивши його основні механіки та естетику. "Pitfall" ввів горизонтальний скролінг, який став стандартом для більшості платформерів, тоді як "Super Mario Bros" продемонстрував потенціал жанру для створення складних і різноманітних ігрових світів.

З розвитком технологій платформи стали більш складними та деталізованими. Сучасні платформери використовують потужні графічні движки, що дозволяють створювати реалістичні візуальні ефекти та динамічні ігрові світи. Для подальшого аналізу розглянемо кілька прикладів сучасних платформерів, які демонструють різноманітність підходів до дизайну та геймплею:

*Dead Cells.* Ця гра поєднує в собі елементи ролевої гри та roguelike, пропонуючи процедурно генеровані рівні та постійно змінювані умови проходження.

*Cuphead.* Вирізняється унікальним ручним малюнком і складними бойовими механіками, що нагадують класичні мультфільми.

*Mark of the Ninja.* Це двовимірний платформер, який фокусується на стелсі та тактичному проходженні рівнів.

*Starbound.* Це двовимірна пісочниця з елементами платформера, яка дозволяє гравцеві досліджувати величезний космос і створювати свій власний світ.

Аналіз цих ігор дозволяє виявити сучасні тенденції в розвитку жанру платформерів, такі як використання процедурної генерації, акцент на сюжеті та персонажах, а також експерименти з різними жанровими гібридами.



### 1.2.1. Dead Cells

Dead Cells – це інноваційна комп'ютерна гра, яка поєднує в собі елементи жанрів roguelike та платформер [1]. Розроблена французькою студією Motion Twin, гра була випущена для широкого спектру платформ, включаючи персональні комп'ютери та ігрові консолі.

Ключовою особливістю Dead Cells є процедурна генерація рівнів, що забезпечує високий рівень реграбельності. Кожне проходження гри є унікальним завдяки випадковому розташуванню ворогів, предметів та елементів ландшафту. Це дозволяє гравцеві досліджувати нові шляхи та стикатися з несподіваними викликами на кожному етапі проходження.

Механіка смерті та відродження, характерна для roguelike-ігор, також є важливим елементом геймплею Dead Cells. Після смерті персонажа гравець починає нове проходження, зберігаючи лише частину своїх досягнень. Такий підхід стимулює гравця до постійного вдосконалення своїх навичок та експериментування з різними комбінаціями зброї та екіпіровки.

Dead Cells пропонує гравцеві нелінійний геймплей, заснований на дослідженні процедурно згенерованих підземель. Процес проходження гри полягає в постійному русі вперед, подоланні численних перешкод, битвах з різноманітними ворогами та зборі цінних ресурсів. Система процедурної генерації забезпечує високий рівень реграбельності, оскільки кожне проходження гри є унікальним.

Ключову роль в геймплеї відіграє система зброї та екіпіровки. Гравець має можливість одночасно використовувати кілька видів зброї, кожна з яких має свої унікальні характеристики та особливості застосування. Випадкова генерація характеристик зброї та предметів додає елемент непередбачуваності та спонукає гравця до експериментів з різними комбінаціями обладнання.

Однією з відмінних рис Dead Cells є високий рівень складності. Гра передбачає часті смерті персонажа, що змушує гравця постійно вчитися на своїх помилках та адаптуватися до мінливих умов ігрового світу. Механіка смерті та

відродження, характерна для roguelike-ігор, додає грі додаткову глибину та стимулює до повторних проходжень.

Dead Cells отримала високі оцінки від критиків та гравців завдяки вдалому поєднанню елементів різних жанрів, інтуїтивно зрозумілому керуванню та високій якості графіки та звуку. Гра стала одним з найяскравіших представників жанру roguelike-платформерів і справила значний вплив на розвиток ігрової індустрії.

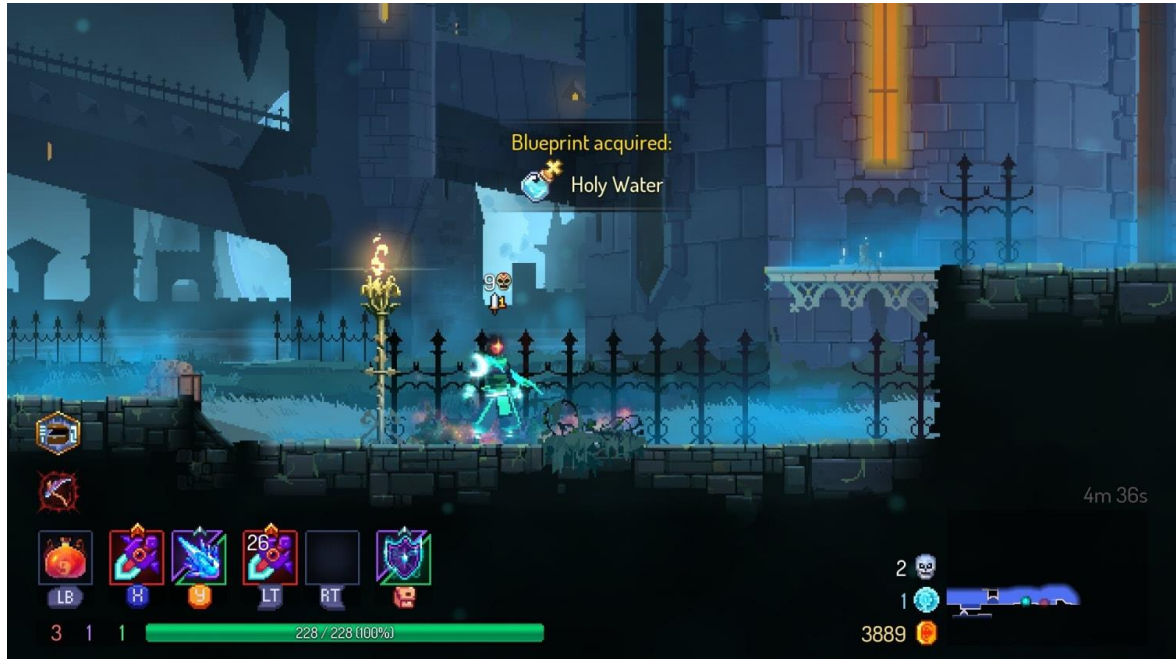


Рис. 1.1. Знімок екрана з гри Dead Cells

### 1.2.2. Cuphead

Cuphead – це інтерактивне мистецьке творіння, що поєднує в собі елементи жанрів платформер та shoot 'em up. Розроблена канадською студією StudioMDHR Entertainment, гра відзначається унікальним візуальним стилем, який є даниною класичним мультфільмам 1930-х років. За допомогою сучасного ігрового рушія Unity творці досягли вражаючої візуальної схожості з роботами таких аніматорів як Уолт Дісней, відтворюючи характерні риси ручної анімації, акварельних фонів та джазового саундтреку.

Сюжет гри обертається навколо пригоди головного героя, Cuphead – це хлопчик з чашкою замість голови, що за допомогою пострілів із пальця бореться з ворогами, який у результаті невдалого парі з дияволом опиняється у боргах. Для повернення своєї душі Cuphead має пройти через серію складних випробувань,

перемагаючи різноманітних босів, кожен з яких має унікальний зовнішній вигляд та стиль атаки. Геймплей Cuphead зосереджений на динамічних боях з босами, що вимагають від гравця високої реакції та знання патернів атак противників.

Cuphead пропонує гравцеві нелінійну структуру ігрового світу, представлену у вигляді інтерактивної мапи, стилізованої під жанр action-RPG. Ця мапа є вузловою точкою, з якої гравець може переміщатися між різноманітними рівнями. Наявність секретних проходів та магазину, в якому можна придбати додаткові життя та вдосконалення, стимулює гравця до повторного дослідження ігрового світу.

Бойова система Cuphead базується на поєднанні елементів жанрів платформер та shoot 'em up. Крім основної атаки, персонаж може використовувати унікальну механіку парирування, яка дозволяє йому відбивати снаряди та атаки ворогів. Успішне парирування заповнює спеціальну шкалу, яка дозволяє виконувати потужні комбінації атак або активувати спеціальні здібності.

Особливу увагу в грі приділено бойовим сутичкам з босами. Кожен бос має унікальний набір атак та вимагає від гравця розробки індивідуальної тактики. Після перемоги над босом, гравець отримує детальну статистику свого проходження, що дозволяє йому аналізувати свої дії та прагнути до кращих результатів.

Cuphead підтримує локальний кооперативний режим, що дозволяє двом гравцям одночасно проходити гру. Другий гравець керує персонажем Mugman і допомагає головному герою у боротьбі з ворогами. Кооперативний режим додає в гру новий рівень складності та дозволяє гравцям спільно подолати найскладніші випробування.



Рис. 1.2. Знімок екрану з гри Cuphead

### 1.2.3. Mark of the Ninja

Mark of the Ninja – це двовимірний платформер з елементами стелс-екшну, розроблений канадською студією Klei Entertainment. Гра відзначається вишуканим візуальним стилем, який поєднує в собі традиційні японські мотиви та сучасні графічні технології.

Сюжет гри обертається навколо досвідченого ніндзя, який прагне помститися за загибель свого клану. Головний герой відрізняється високою рухливістю та різноманітними інструментами для виконання стелс-місій. Спеціальне татуювання, яке він носить, надає йому надзвичайних здібностей, але водночас поступово руйнує його розум.

Mark of the Ninja пропонує глибокий і занурюючий досвід стелс-екшну, фокусуючись на безшумному проходженні рівнів та уникненні зіткнень з ворогами. Геймплей гри побудований навколо механік стелсу, які включають у себе можливість ховатися в тіні, використовувати відволікання та знешкоджувати пастки. Головний герой, ніндзя, озброєний мечем ніндзято та бамбуковими дротиками, що дозволяє йому ефективно ліквідувати ворогів або відволікати їхню увагу.

Однією з відмінних рис гри є наявність розвиненої системи прокачування персонажа. За виконання різних дій, таких як усунення ворогів без виявлення або знешкодження пасток, гравець отримує бали, які можна витратити на відкриття

нових прийомів, вдосконалення зброї та придбання нового спорядження. Ця система дозволяє гравцеві адаптувати стиль проходження до власних уподобань, обираючи між агресивним та більш обережним підходом.

Mark of the Ninja відзначається вишуканим візуальним стилем, який створює атмосферу таємничості та небезпеки. Двовимірний графічний дизайн, деталізовані фони та ефективне використання світла і тіні дозволяють гравцеві повністю зануритися в ігровий світ (рис. 1.3).

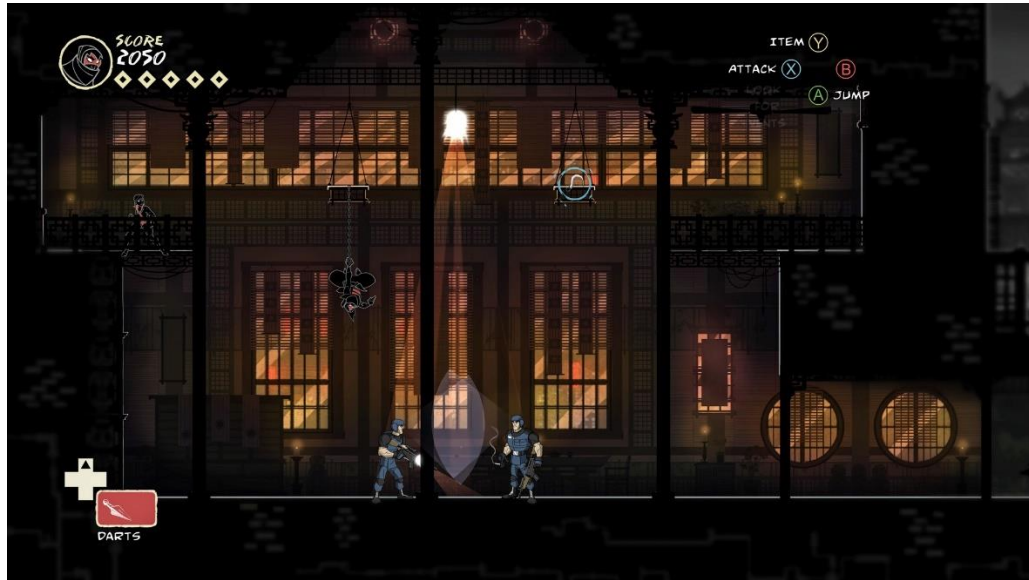


Рис. 1.3. Знімок екрана з гри Mark of the Ninja

#### 1.2.4. Starbound

Starbound – це інноваційна комп'ютерна гра, яка поєднує в собі елементи жанрів платформер та пісочниці. Розроблена студією Chucklefish Games, гра пропонує гравцям необмежені можливості для дослідження процедурно згенерованого космосу. В основі гри лежить власний ігровий рушій, написаний на мові програмування C++, що дозволило розробникам реалізувати глибоку систему кастомізації та широкий спектр ігрових можливостей.

Гравці починають гру на своєму космічному кораблі, який служить мобільним базовим табором для дослідження космосу. Кожна нова планета, яку відвідує гравець, є унікальним світом зі своїми особливостями клімату, флори, фауни та ресурсами. Процедурна генерація забезпечує велику різноманітність планет, що дозволяє гравцям досліджувати нові біоми та зустрічати незвичайних істот.

Starbound – це масштабна пісочниця, яка пропонує гравцям безмежні можливості для творчості та дослідження. Однією з ключових особливостей гри є глибока система кастомізації, що дозволяє гравцям створювати унікальних персонажів та будувати власні світи. Вибір раси на початку гри визначає не лише зовнішній вигляд персонажа, але й його здібності, культуру та взаємовідносини з іншими расами. Крім того, гравець має можливість будувати різноманітні споруди, створювати інструменти та зброю, а також модифікувати свій космічний корабель.

Процедурна генерація є ще однією важливою складовою геймплею Starbound. Кожна планета в грі є унікальною, оскільки її характеристики генеруються випадковим чином. Це означає, що кожна нова планета, яку відвідує гравець, пропонує нові виклики та можливості для дослідження. Різноманітність біомів, від пустель і джунглів до засніжених планет і водних світів, забезпечує високий рівень реграбельності. Крім того, процедурна генерація впливає на такі аспекти ігрового світу, як:

- 1) Флора і фауна: різноманітність рослин і тварин, які адаптовані до умов конкретної планети.
- 2) Ресурси: різні типи матеріалів, необхідних для будівництва та крафту.
- 3) Погодні умови: динамічні зміни погоди, які можуть впливати на геймплей.
- 4) Гравітація: рівень гравітації може відрізнятися на різних планетах, що вимагає від гравця адаптації до нових умов.

Поєднання глибокої кастомізації та процедурної генерації дозволяє гравцям створювати власні історії в безмежному космічному просторі. Starbound пропонує гравцям не лише досліджувати готові світи, але й активно впливати на їх формування.





дозволяє створити більш глибокий і захоплюючий ігровий досвід, що є важливим фактором для залучення та утримання гравців.



## **РОЗДІЛ 2. ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ КОМП'ЮТЕРНОЇ ГРИ В ЖАНРІ 2D ПЛАТФОРМЕРА**

### **2.1. Порівняльна характеристика ігрових движків**

Більшість інструментальних засобів для розробки ігор можна поділити на 3 групи:

1. Фреймворк - програмна платформа, яка визначає структуру програмної системи; програмне забезпечення, що полегшує розробку і об'єднання різних компонентів великого програмного проєкту. Фреймворк відрізняється від поняття бібліотеки тим, що бібліотека може бути використана в програмному продукті просто як набір підпрограм близькою функціональності, не впливаючи на архітектуру програмного продукту і не накладаючи на неї ніяких обмежень. У той час як фреймворк диктує правила побудови архітектури додатку, задаючи на початковому етапі розробки поведінка за умовчанням, каркас, який потрібно буде розширювати і змінювати відповідно до зазначених вимог.
2. Ігровий рушій - центральний програмний компонент комп'ютерних та відеоігор або інших інтерактивних додатків з графікою, оброблюваної в реальному часі. Він забезпечує основні технології, спрощує розробку і часто дає грі можливість запускатися на декількох платформах, таких як ігрові консолі та настільні операційні системи, наприклад, GNU / Linux, Mac OS X і Microsoft Windows. Основну функціональність зазвичай забезпечує ігровий движок, що включає движок рендерінга («візуалізатор»), фізичний движок, звук, систему скриптів, анімацію, штучний інтелект, мережевий код, управління пам'яттю і багатопоточність. Часто на процесі розробки можна заощадити за рахунок повторного використання одного ігрового движка для створення безлічі різних ігор.
3. Конструктор ігор - програма, яка об'єднує в собі ігровий рушій і інтегроване середовище розробки, і, як правило, включає в себе

редактор рівнів, що працює за принципом WYSIWYG. Такі програми значно спрощує процес розробки ігор, роблячи його доступним любителям-непрограмістів, і можуть бути використані в початковому навчанні програмуванню [1-3].

### 2.1.1. Unity

Кросплатформерний рушій для розробки ігор, розроблений компанією Unity Technologies. Unity підтримує майже усі платформи – Windows, Linux, MacOS, Android, iOS, FireOS, PlayStation Vita, PlayStation 4, Xbox One, Nintendo Switch, Nintendo 3DS, Oculus Rift, Steam VR, Gear VR, PlayStation VR, Android TV, Smart TV, TvOS.

Для роботи з Unity потрібен комп'ютер з мінімальними вимогами:

- Персональний комп'ютер з операційною системою Windows 7/8/10 x64 або MacOS X 10.9+;
- Процесор з підтримкою SSE2;
- Відеокарта з підтримкою DirectX 10.

Як і більшість рушіїв, Unity має простий та зрозумілий для розробників інтерфейс (рис 2.1).

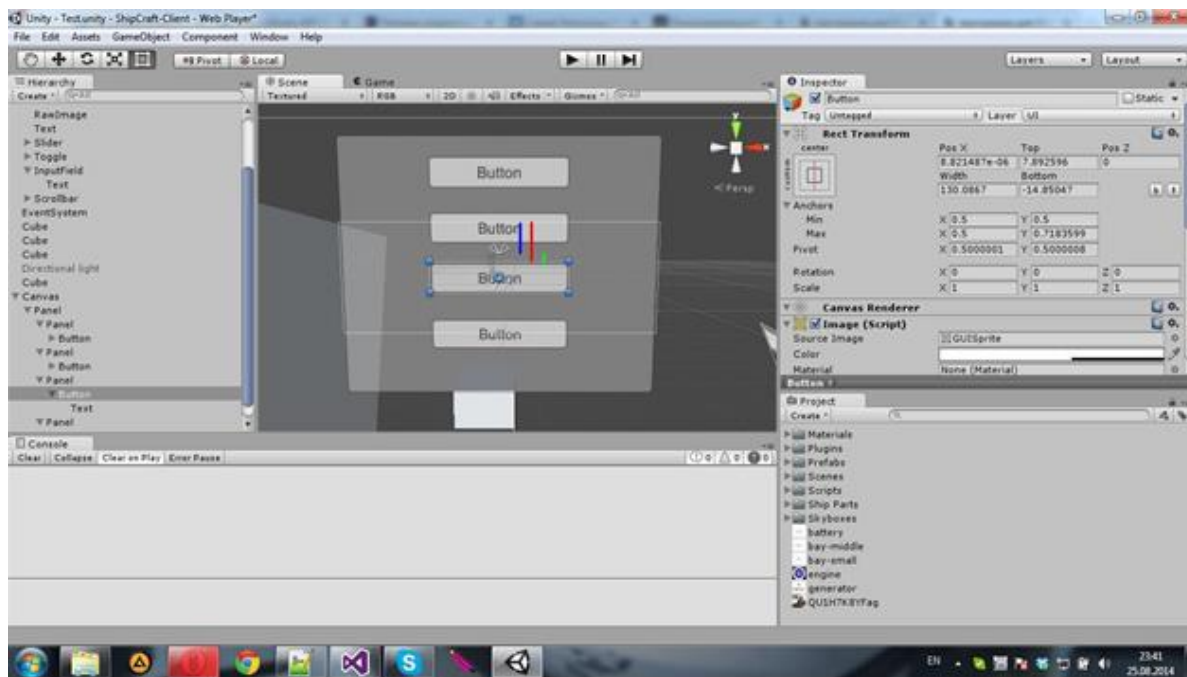


Рис. 2.1. Інтерфейс Unity

Unity є різносторонньою програмою розробки як для програмістів так і для художників завдяки різним інструментам, таким як: Timeline – для розробки анімаційних сцен, Cinemachine – набір динамічних камер, Progressive Lightmapper – для роботи з освітленням, Autodesk Maya – робота з моделюванням та 3D анімацією. Рушій дає можливість розроблювати проєкти як у 2D, так і у 3D стилях. Також є підтримка рушіїв NVIDIA PhysX для роботи з фізикою об'єктів та Box2D для роботи з двовимірними об'єктами.

Unity відомий відмінною оптимізацією, що помітно позначається на якості та швидкості праці проєкту. Рушій підтримує мови програмування такі як C# та UnityScript.

Як і попередники, що були проаналізовані вище, сайт Unity має дуже велику кількість документації різного напрямлення, у тому числі і навчаючої. Саме навчаюча документація представлена у вигляді окремих уроків на розборів різних тем.

Unity має свій відомий магазин Asset Store, у якому представлена велика кількість матеріалів для розробки: спрайти, моделі, скрипти, фонові рисунки, різні доповнення та розширення до клієнту Unity, тощо.

Даний рушій має 3 версії придбання: Personal (безкоштовна), Plus (35\$/місяць), Pro (125\$/місяць). Версії відрізняються лише пропонованим функціоналом та максимально допустимим річним прибутком (100 тисяч доларів, 200 тисяч доларів та без обмежень відповідно). Рушій є дуже різностороннім та унікальним, що дозволяє використовувати його для розробки різноманітних продуктів.

Так як для реалізації нашого проєкту головними критеріями вибору середі розробки є вільна ліцензія, навчальна документація, зручний та зрозумілий інтерфейс, низькі системні вимоги та інструменти для розробки 2D проєктів, нашим вибором стане ігровий рушій Unity.

Реалізація програмного коду у Unity відбувається за допомогою Microsoft Visual Studio, яка має можливість інтеграції додаткового функціоналу для Unity (рис 2.2).

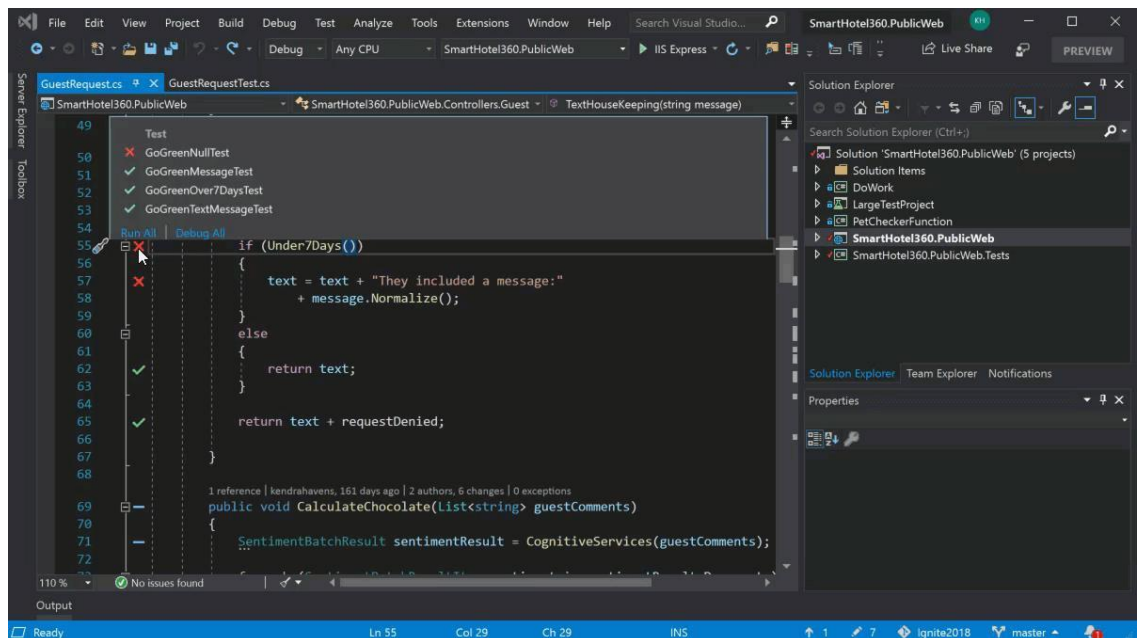


Рис. 2.2. Інтерфейс Visual Studio

Visual Studio Community 2019 – інтегрована середовище розробки програмного забезпечення розроблена та видана компанією Microsoft. Visual Studio включає у себе редактор вихідного коду та має можливість рефакторингу. Також існує редактор форм для створення графічного інтерфейсу додатків. Дана середовище розробки є зручним інструментом програмування та відмінно підходить для написання коду.

### 2.1.2. Torque 2D/3D

Torque 2D / 3D, був свого часу лідером, але під натиском Unity втратив свої позиції. Проте до цих пір на ньому розробляється безліч успішних проєктів, оскільки він активно розвивається спільнотою. Відмінності між двовимірної і тривимірної версіями досить значні, але є і загальні елементи, наприклад розвинена мережева підсистема. Після виходу в світ open source T3D зберіг і навіть збільшив свої можливості, а T2D, навпаки, багато втратив. Наприклад, він втратив абсолютно всі вбудовані редактори, які, очевидно, були вилучені за певних юридичних угод. Проте на ньому можна розробляти ігри для трьох платформ: Windows, OS X і, що найцікавіше, iOS (і продавати ігри в App Store, не відраховуючи ні копійки авторам движка). Даний движок - це одна кодова база на C++ без додаткових експортерів. Як видно, фундаментальні відмінності 2D і 3D - версій полягають в графічній підсистемі: T2D для візуалізації

використовує OpenGL, а T3D - Direct.

В якості скриптової мови в T2D, як і в T3D, використовується Torque Script. Разом з тим в T2D для опису ігрових елементів служить XML-подібна мова TAML. Вона дозволяє визначити властивості об'єктів на стадії ініціалізації рівня гри. Для відтворення звуків T2D використовує бібліотеку OpenAL. Симуляція фізики здійснюється за допомогою движка Box2D, що став стандартом в двовимірних фізичних обчисленнях. Незважаючи на те що в двовимірному ТОРК ще поки немає конструктора GUI, за допомогою засобів движка (в скриптовому коді) можна створювати призначений для користувача інтерфейс звичними компонентами, а не простими спрайтами.

Однак, якщо необхідний компонент відсутній, його можна створити на основі спрайтів. Маючи аналогічну з 3D-версією мережеву систему, на T2D можна розробляти мультиплеєрні ігри, які набирають популярність, - наприклад P2P з планшетів.

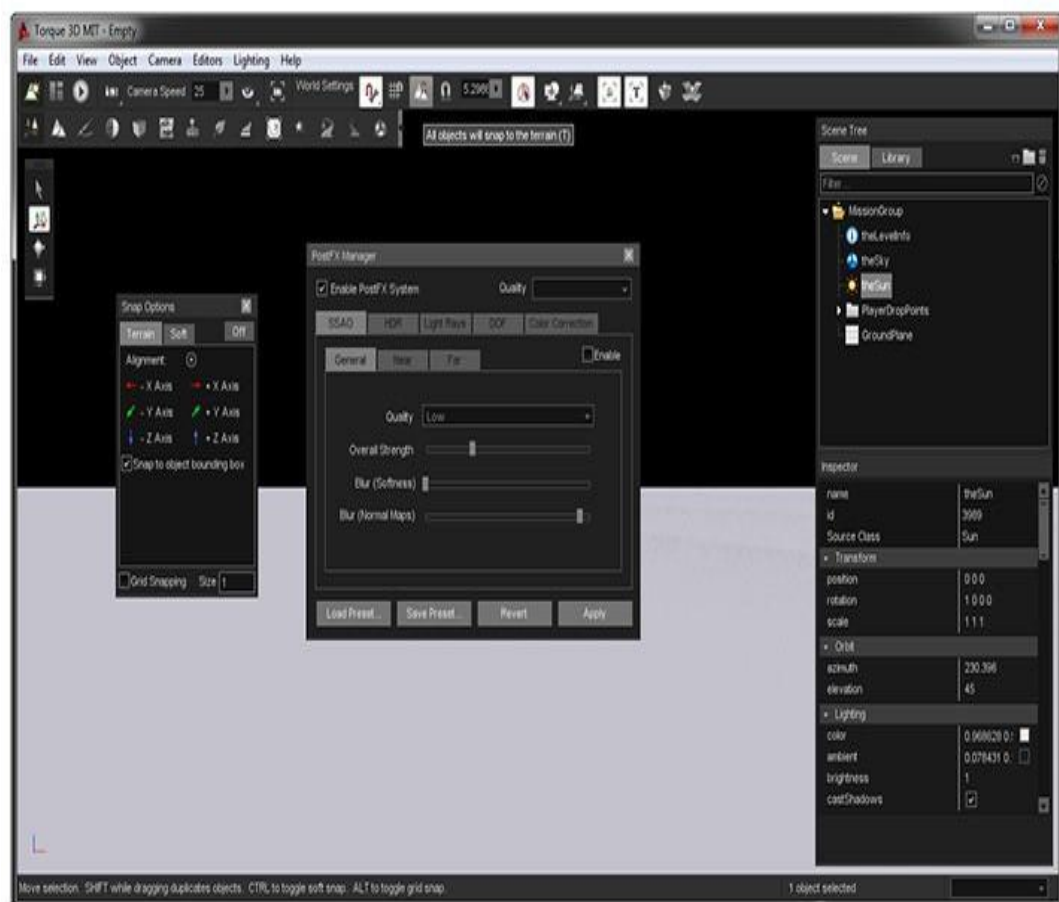


Рис. 2.3. Основний інтерфейс Torque 3D

**Переваги і недоліки Torque 3D**

<b>Недоліки</b>	<b>Переваги</b>
Немає якісного редактора рівнів	Якісний
Повільна компіляція	Багатоплатформовий
Високі вимоги до заліза	Легкий в освоєнні
Немає вихідного коду	Безкоштовний

**2.1.3. CryEngine 3**

CryEngine 3 бере початок своєї історії в 2001 році, коли була анонсована перша розроблена на ньому гра Far Cry. З тих пір минуло багато часу, і поточна на даний момент п'ята - остання версія була випущена в жовтні 2016-го. Розробники цього движка з самого початку мали на меті не самим створювати на ньому ігри, а продавати його як технологію. Отже, всі розроблені Crytek'ом ігрові програми - це з метою зробити додаткову рекламу своєму головному продукту. Хоча для вивчення він доступний безкоштовно, щоб розробляти на ньому комерційні проєкти, необхідно заплатити, при чому ціна публічно не оголошується. В результаті ліцензіат отримує движок, документацію (навчальні матеріали), вихідний код, а також оперативну підтримку. Крім того, процес ліцензування движка таїть в собі безліч підводних каменів - хоча б те, що ліцензувати його може тільки юридична особа, яка має надати дані про розроблені продукти і в окремих випадках про всіх своїх співробітників.

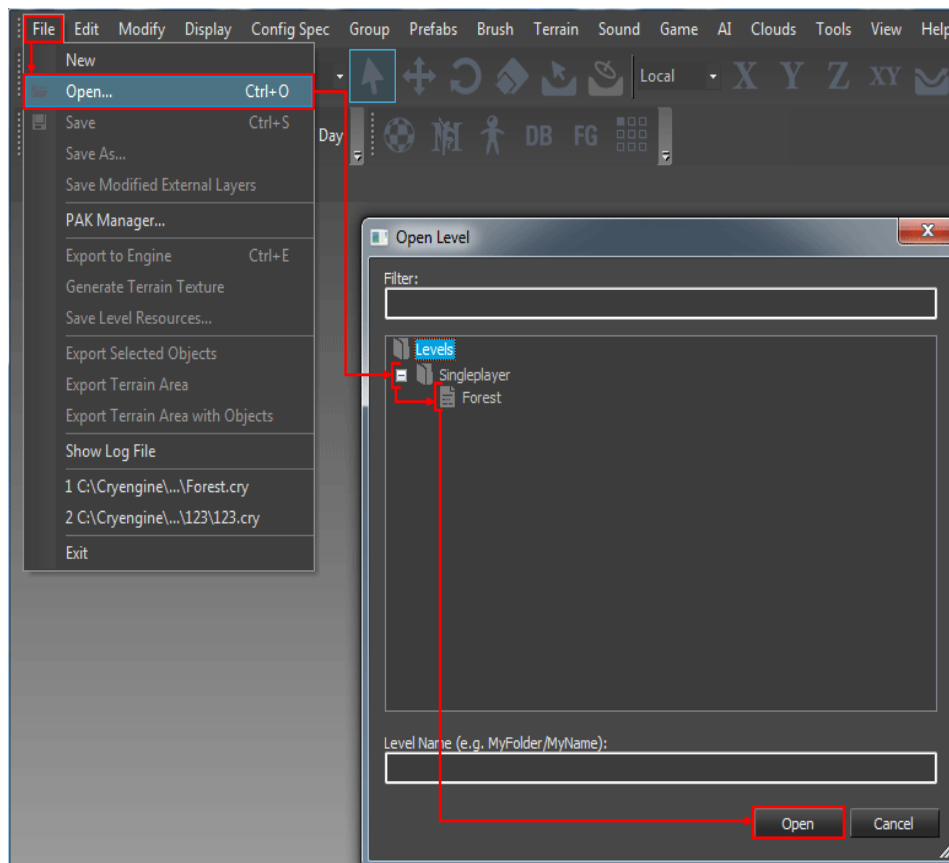


Рис. 2.4. Основний інтерфейс CryEngine 3

Таблиця 2.2.

### Переваги і недоліки CryEngine 3

Недоліки	Переваги
Не безкоштовний	Якісний
Повільна компіляція	Багато документації та уроків
Високі вимоги до заліза	Легкий в освоєнні
Не багатоплатформовий	
Неможливо продавати свої ассети та скрипти	

На відміну від попередніх движків лінійки (які були виключно PC-орієнтованими), CryEngine 3 орієнтований на створення крос-платформних ігор, призначених для PC і консолей. В даний час підтримуються платформи Xbox 360, Xbox One, PlayStation 3-4, WiiU, а також технології візуалізації настільної Windows - DirectX 9-11. Як можна помітити, підтримки мобільних платформ немає. У ньому спочатку присутня підтримка глобальних мультиплеєрних (MMO) ігор. CryEngine 3 володіє приголомшливим списком технологій

візуалізації, ось деякі з них: динамічне освітлення і затінення в реальному часі, затуманення, карти нормалей і паралакс-маппінг, підповерхнєве розсіювання, світлові промені і хвилі, управління рівнем деталізації ландшафту, а також багато іншого. Фізичний компонент движка CryPhysics також працює незалежно від фізичних API, таких як PhysX. Вбудована система анімації пропонує кілька відмінних підсистем: індивідуалізація персонажів, параметрична скелетна анімація, процедурне деформування руху. Також заслуговує на окрему увагу вбудована система AA, яка дозволяє обробляти поведінку не тільки персонажів, але і транспортних засобів. Вона складається з трьох модулів: розумні об'єкти, алгоритми динамічного виявлення шляху, а також система, керована сценаріями.

#### **2.1.4. UDK**

UDK - це безкоштовна версія движка Unreal Engine 3, що володіє всім успадкованим інструментарієм останнього для створення ігрових світів. Список підтримуваних платформ не такий широкий, як у Unity, але цього цілком вистачає, щоб окупити розробку: Windows PC, Windows Store, OS X, iOS, Android і консолі передостаннього покоління. Для скриптинга в движку використовується власна мова - UnrealScript. На сайті розробників представлено багато навчальних матеріалів, як текстових, так і відео, як по редактору, так і по скриптингу. UE3 отримав безліч нагород на індустріальних заходах, а також в кінематографії і не раз ставав кращим ігровим / графічним движком року. Можна сказати, що UDK відрізняється від UE3 тільки відсутністю вихідного коду.



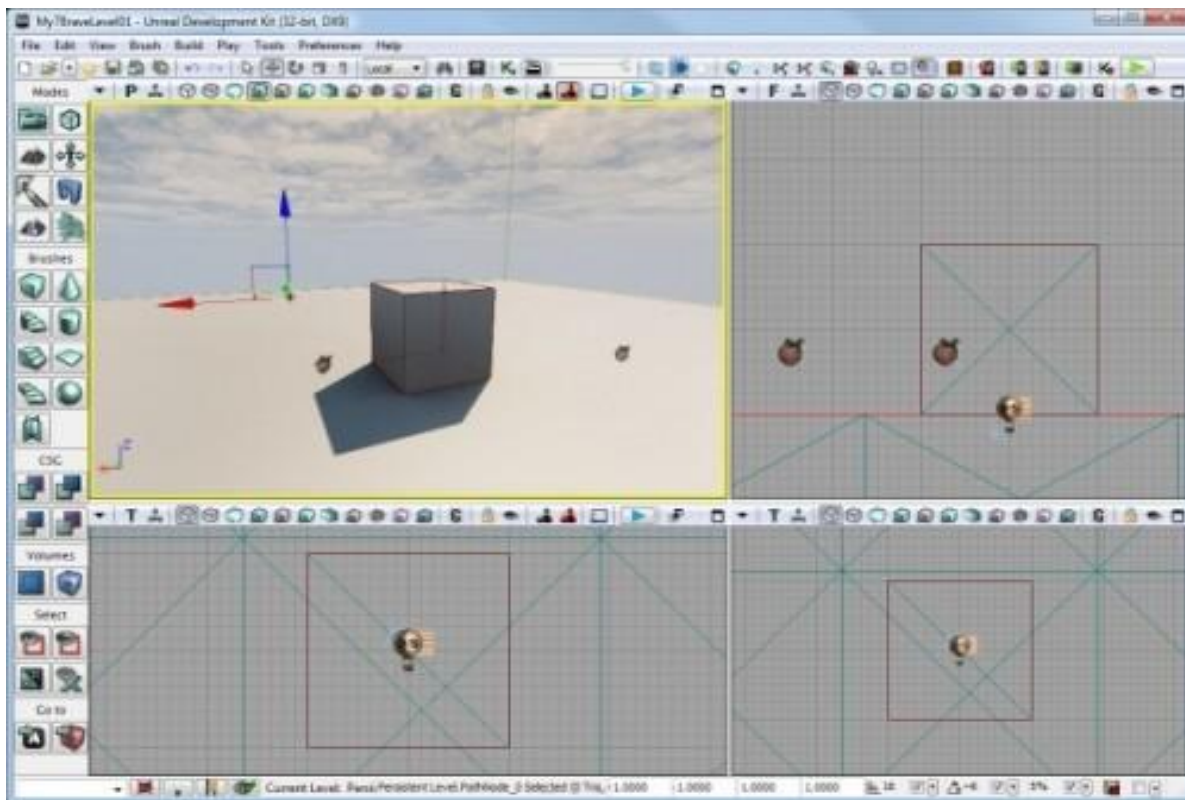


Рис. 2.5. Основний інтерфейс UDK

Таблиця 2.3.

### Переваги і недоліки UDK

Недоліки	Переваги
Тільки для професіоналів	Безкоштовне розповсюдження
Повільна компіляція	Простий, зручний інтерфейс;
Не самі передові технології	Великий набір інструментів для створення ігор;
Не багатоплатформовий	Багатоплатформовий

Що до функціоналу, гнучка система анімації дозволяє контролювати кожну деталь анімованого об'єкта. Анімаційна модель контролюється системою AnimTree, яка включає наступні механізми: контролер змішання (Blend), контролер керований даними, фізичні, процедурно-скелетні контролери. Для імпортування об'єктів використовується формат FBX, що став стандартом для експорту моделей між редакторами. Для візуалізації UE3 використовує 64-бітний кольоровий HDR графічний конвеєр, який здійснює гамма-корекцію,

розмиття рухомих об'єктів, зовнішню окклюзію і інші ефекти постобробки. Движком підтримуються всі сучасні ефекти освітлення і технології візуалізації: нормалізовані карти, параметризоване освітлення по Фонгу, різні анізотропні ефекти та інше. UE3 відомий своєю високо оптимізованою мережевою архітектурою, що включає підтримку онлайн-ових баталій для ігор різних жанрів.

### 2.1.5. Frostbite Engine

Frostbite Engine - ігровий движок, розроблений компанією EA Digital Illusions CE; застосовується як у власних розробках, так і проєктах інших філіалів Electronic Arts (EA). Першою грою, створеною з використанням цього движка стала Battlefield: Bad Company 2008 року. Движок був розроблений для заміни технічно застарілої технології Refractor Engine, яка використовувалася в попередніх іграх фірми.

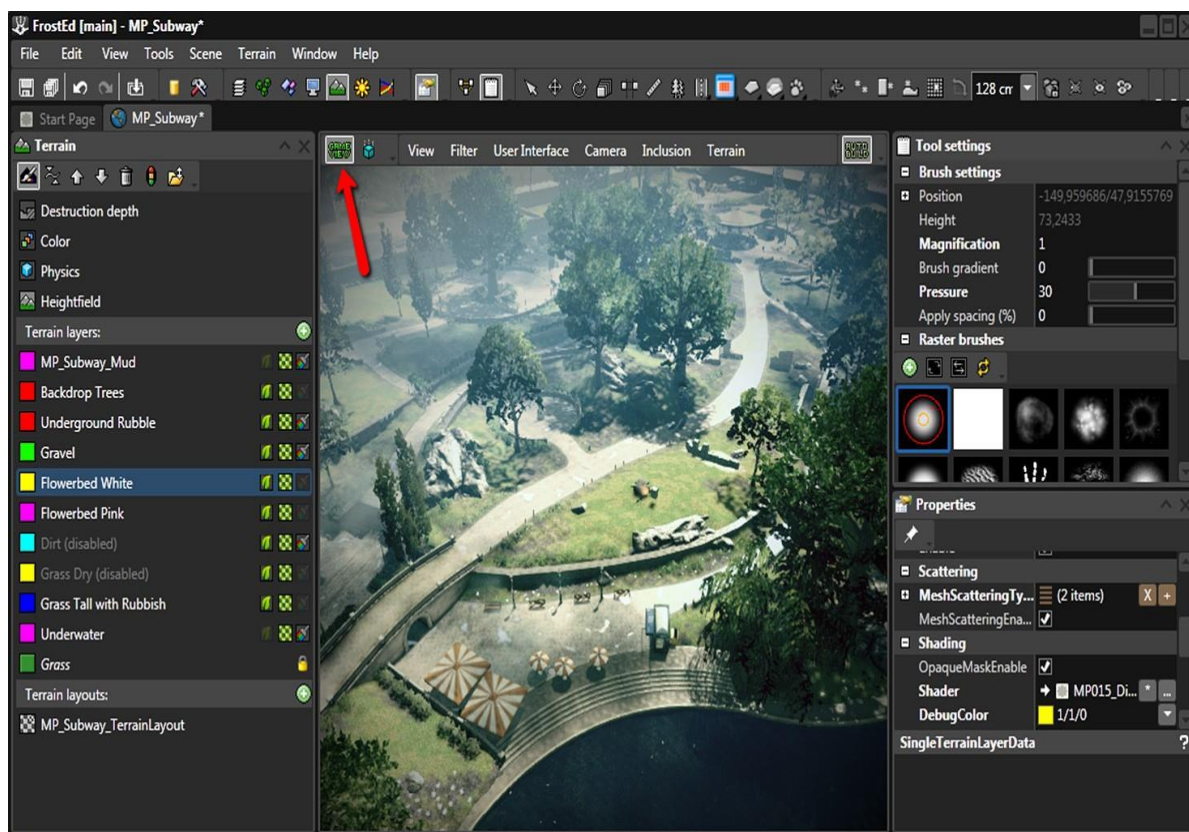


Рис. 2.6. Основний інтерфейс Frostbite Engine

Таблиця 2.4.

### Переваги і недоліки Frostbite Engine

Недоліки	Переваги
Не безкоштовний	Ідеальне руйнування ландшафту

Недоліки	Переваги
Не багатоплатформовий	Динамічне освітлення
Високі вимоги до заліза	Ігровий редактор FrostED
Неможливо продавати свої ассети та скрипти	Власний звуковий движок

Движок відноситься до типу підпрограмного забезпечення (англ. Middleware) і являє собою зв'язку декількох компонентів, таких як графічний движок, звуковий движок і т.д. В операційній системі Microsoft Windows ігровий движок підтримує відображення графіки за допомогою Mantle починаючи з версії 3, DirectX 9, DirectX 10, DirectX 10.1, а починаючи з версії 1.5 - і DirectX 11. Однією із заявлених особливостей є оптимізація для роботи на багатоядерних процесорах.

Технологія здатна обробляти знищення ландшафту і оточення (наприклад, будівель, дерев, автомобілів). Підтримується динамічне освітлення і затінення з функцією НВАО, процедурний шейдинг, різні пост- ефекти (наприклад, HDR і depth of field), система частинок і техніки текстуривання, такі, як бамп-маппінг. Максимальний розмір локації становить обмеження в  $32 \times 32$  кілометри відображаємої площі і  $4 \times 4$  кілометри ігрового простору. Крім цього, за твердженням творців, максимальна дистанція промальовування дозволяє побачити рівень аж до горизонту. Також вбудований власний звуковий движок, що не вимагає використання спеціалізованих засобів, подібних EAX.

Згадана лише мізерна частина доступних для використання ігрових движків.

Таблиця 2.5.

#### Допоміжна порівняльна характеристика усіх движків

Характеристика	Вимоги	Unity 3D	Torque 2D/3D	CryEngi ne 3	UDK	Frostbite Engine
Режим рендеринга	3D	+	-	+	+	+
Звуковий движок		-	-	-	+	+
Фізичний движок		+	+	+	+	+
Ігровий П		+	-	-	+	+
Цільовий жанр	RPG	+	+	+	-	+

Характеристика	Вимоги	Unity 3D	Torque 2D/3D	CryEngi ne 3	UDK	Frostbite Engine
Мова розробки	C# или JavaScript	+	-	+	-	+
Інтеграція з IDE	Visual Studio	+	+	+	+	-
ОС для розробки	Windows	+	+	+	+	+
Цільові платформи	Windows, OSX	+	+	+	+	+
Ціна	безкоштовно	+	+	-	+	-
Цільова аудиторія	Для професіоналів	Так	Так	Так	Ні	Так
Якість документації		3	2	2	1	3
Інтерфейс користувача		3	2	3	3	2
Результуючий рейтинг		6	4	5	4	5

## 2.2. Архітектура, моделювання та проєктування гри

### 2.2.1. Створення Usecase

Usecase – це текстовий опис сукупності сценаріїв взаємодії користувача із системою, які можуть завершитися успішно або ні, залежно від досягнення певних цілей. Сценарій являє собою послідовність дій, що виконуються користувачем для реалізації операцій у системі. Опис варіантів використання допомагає чітко визначити користувачів системи, їхні завдання, а також цілі використання системи. Usecase відображають функціональні та поведінкові вимоги до системи, демонструючи, які операції вона повинна виконувати [12].

Існує три форми опису листа:

1. **Коротка форма** – стисле викладення одного зі сценаріїв (зазвичай успішного) взаємодії із системою. Застосовується на етапі початкового аналізу вимог.
2. **Поверхнева форма** – загальний опис усіх сценаріїв у вільному стилі, включно з основним та альтернативними сценаріями одного варіанта використання. Використовується під час первинного аналізу системних вимог.
3. **Повна форма** – детальний опис усіх кроків і дій, включно з передумовами та постумовами виконання юзкейсу. Використовується на етапі

відбору найбільш важливих варіантів використання з коротких і поверхневих форм. Застосовується для критичних для роботи системи процесів, таких як використання касових апаратів [7].

### **Коротка форма «Генерація рівня»**

Користувач запускає гру та відкриває головне меню. Натискає кнопку «Play», після чого переходить на головну сцену гри. Успішно проходить рівень і опиняється на екрані вибору: завершити гру або перейти до наступного рівня.

### **Поверхнева форма «Генерація рівня»**

Користувач запускає гру, відкриває головне меню та натискає кнопку «Play».

**Повна форма «Генерація рівня»**

<b>Usecase section</b>	<b>Comment</b>
Use Case Name	Генерація рівня
Scope	Система генерації рівня
Level	Генерація рівня
Primary Actor	Користувач
Stakeholders and interests	Користувач – користування застосунком
Preconditions	Користувач повинен мати телефон із встановленою OS Android
Success guarantee	1. у користувача встановлена остання версія; 2. у користувача є мінімальні вимоги, які потрібен додаток;
Main Success Scenario	1.користувач входить у гру; 2.натискає на кнопку запуску гри; 3.потрапляє в ігровий рівень;
Special Requirements	1. OS Android має бути версії 6.0; 2. апаратне забезпечення повинне мати Оперативна пам'ять не менше 1 ГБ; 3. апаратне забезпечення повинне мати 50 МБ вільної пам'яті чи більше.
Frequency of Occurrence	50%

**2.2.2. Діаграми варіантів використання**

Діаграма варіантів використання є концептуальною моделлю, яка відображає систему під час її проєктування та розробки. Основною метою створення таких діаграм є візуалізація сценаріїв використання системи. У рамках проєкту були розроблені діаграми варіантів використання. На (рис. 2.7)

представлено приклад діаграми варіантів використання для мобільного застосунку.

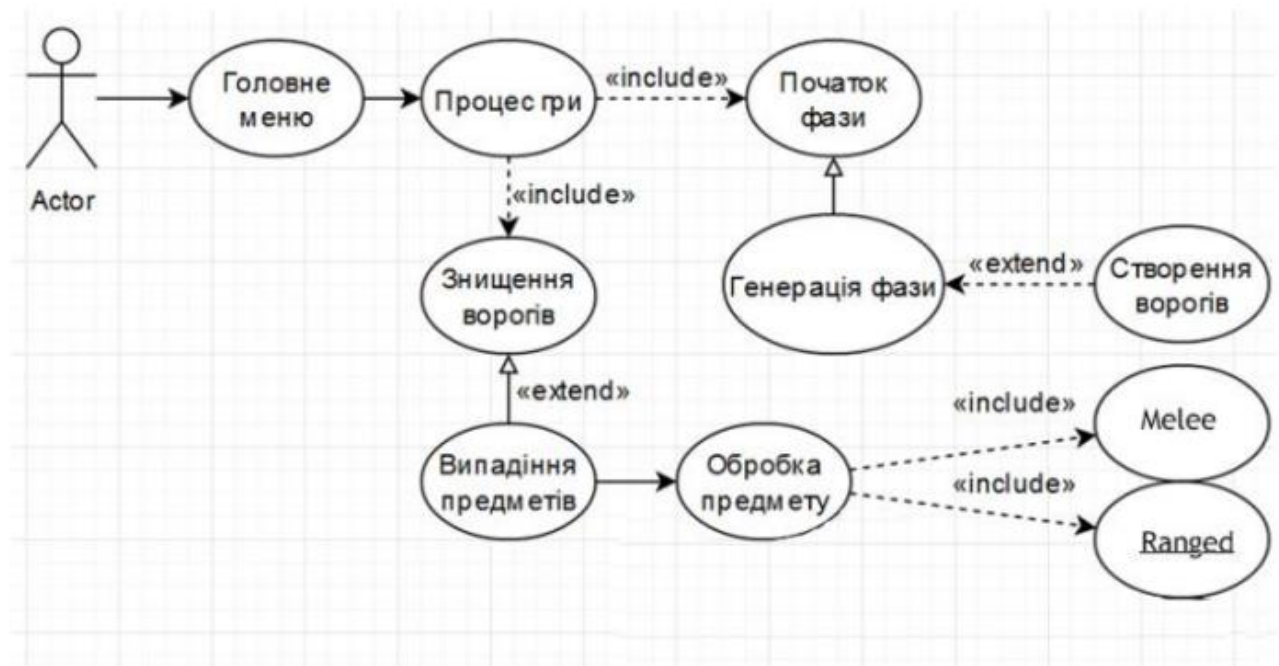


Рис. 2.7. Діаграма варіантів використання

Гравець запускає гру, потрапляє у головне меню та розпочинає гру, переходячи на перший рівень.

**Структура специфікації варіанта використання** включає:

- Найменування;
- Короткий опис;
- Ідентифікатор;
- Актори (основні та другорядні);
- Передумови;
- Основний потік;
- Альтернативні потоки;
- Післяумови;
- Спеціальні вимоги.

**А) Специфікація варіанта використання «Скорочення часу»**

- **Ім'я:** скорочення часу
- **ID:** 0
- **Короткий опис:** гравець грає у гру під час перерви між роботою чи



домашніми справами.

- **Основна дійова особа:** гравець
- **Другорядні дійові особи:** немає
- **Передумови:** немає
- **Основний потік:**
  - ВВ починається, коли гравець запускає систему.
- **Післяумови:**
  - Гравець програв та вимкнув систему.

- **Альтернативні потоки:** немає

#### **Б) Специфікація варіанта використання «Грати в гру»**

- **Ім'я:** грати в гру
- **ID:** 1
- **Короткий опис:** знайомство з додатком та отримання ігрового досвіду.

- **Основна дійова особа:** користувач
- **Другорядні дійові особи:** вороги, ігрові об'єкти, ігровий персонаж
- **Основний потік:**
  - ВВ починається, коли гравець натискає кнопку «Play».
  - Система виконує свої функції.
- **Післяумови:**
  - Гравець керує персонажем, знищує ворогів, підіймає предмети та вдосконалює персонажа.
- **Альтернативні потоки:**
  - Гравець повертається у головне меню.

#### **2.2.3. Діаграма класів**

Наступна діаграма представляє діаграму класів, яка є одним із ключових типів діаграм у UML. Діаграми класів відображають логічну структуру програмної системи, що має важливе значення для процесу генерації коду. Основними елементами цієї діаграми є класи та зв'язки між ними.



**Клас** — це сукупність логічних об'єктів, які характеризуються певними атрибутами (характеристики) та операціями (поведінка). В ООП класи мають 5 основних типів зв'язків:

- **Асоціація (association)** — абстрактний зв'язок між класами.
- **Агрегація (aggregation)** — зв'язок "частина — ціле", де час життя частини не збігається з часом життя цілого.
- **Композиція (composition)** — зв'язок "частина — ціле", де час життя частини залежить від часу життя цілого.
- **Наслідування (generalization)** — зв'язок "загальне — часткове".
- **Інстанціювання (instantiation)** — створення нового класу шляхом підстановки фактичних параметрів у параметризований клас.

**Архітектура системи** є незвичною для проєктів Unity. Замість багатьох компонентів використовується один головний клас **Manapp**, через який реалізуються основні функції двигуна.

Опис ключових компонентів:

- **GameClient** — підключається до класів через інтерфейс **IService**.
- **UIManager** — відповідає за управління інтерфейсом користувача через інтерфейси **IUIPopup**. Забезпечує взаємодію зі сторінками.
- **DataManager** — зберігає інформацію та стан гри.
- **LoadObjectManager** — відповідає за завантаження ресурсів (звуків, зображень, колекцій).
- **GameManager** — головний керуючий геймплеєм, працює через класи, які реалізують інтерфейс **IController**.
- **InputManager** — обробляє введення користувача (клавіші, натискання) через події.
- **SoundManager** — керує звуковими ефектами.

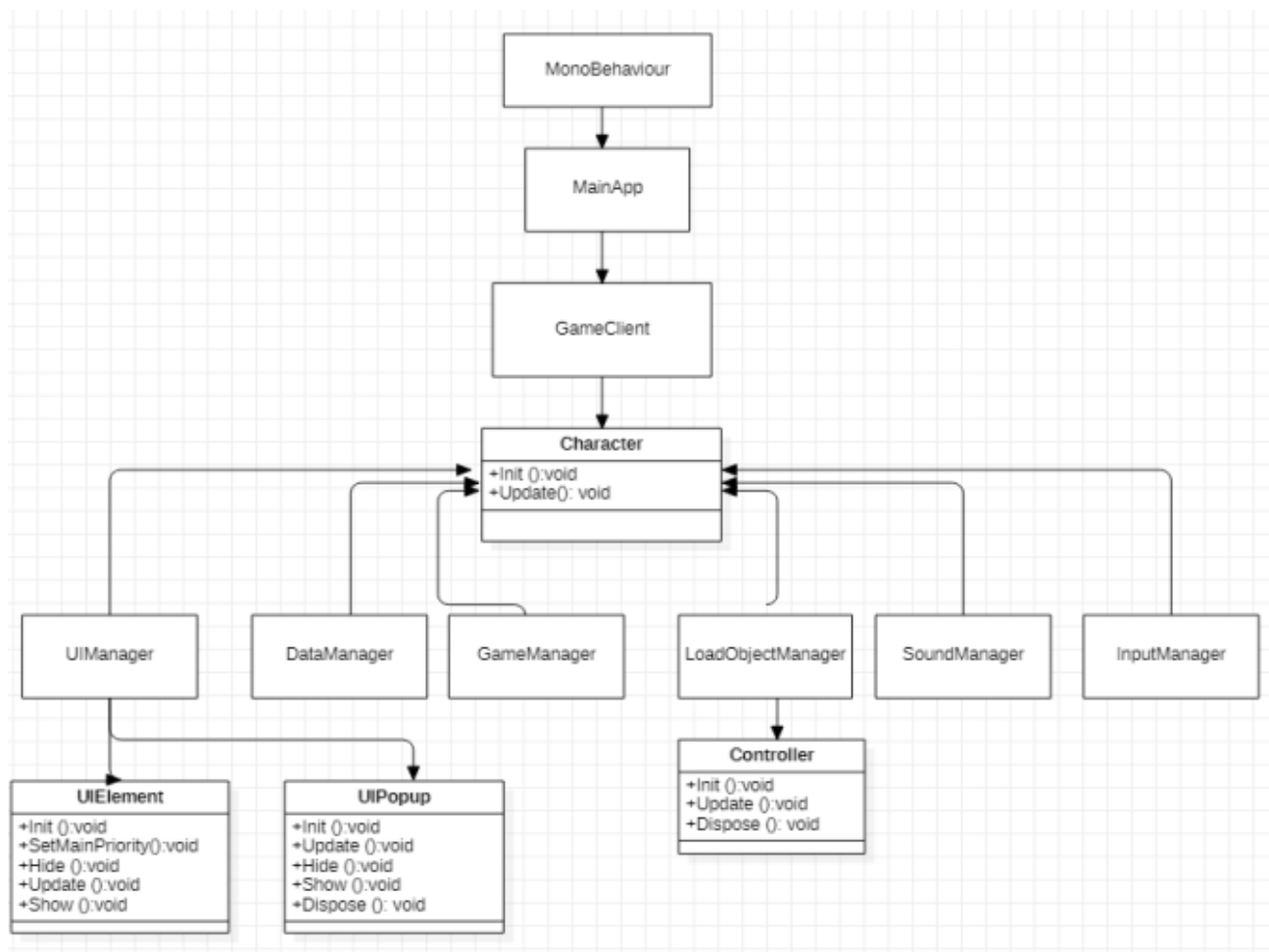


Рис. 2.8. Діаграма класів

#### 2.2.4. Діаграми станів та переходів

Статистичні діаграми разом із іншими діаграмами ілюструють окремі або всі сценарії, які виконуються в системі загалом. Діаграма станів представляє готову машину станів у вигляді графа, де вершини відповідають станам об'єкта, чия поведінка моделюється, а переходи — це події, які змінюють стан об'єкта [11].

**Стан (state)** — це логічний опис певної ситуації, дії чи процесу. Кожен стан має назву та перелік внутрішніх дій, що виконуються, коли об'єкт або система знаходяться в цьому стані. Дії вказуються у форматі: "**Період виконання**" – "**Назва дії**", де період може бути наступним:

- **OnEntry** — дія виконується під час входу в цей стан.
- **OnExit** — дія виконується при виході з цього стану.
- **Do** — дія виконується, поки система перебуває в цьому стані.

- **OnEvent** — дія виконується у відповідь на певну подію [8].

На (рис. 2.9) наведена схема станів і переходів системи ураження. Значення за замовчуванням автоматично вводяться в поля панелі генерації при її відкритті. Далі, коли користувач обирає місцезнаходження, запис оновлюється новими значеннями. Зміна розміру також перезаписує попередні дані. Аналогічний процес відбувається з іншими полями. Нарешті, після натискання кнопки "Генерація" система ініціює подію, яка завершується створенням ігрового регіону.

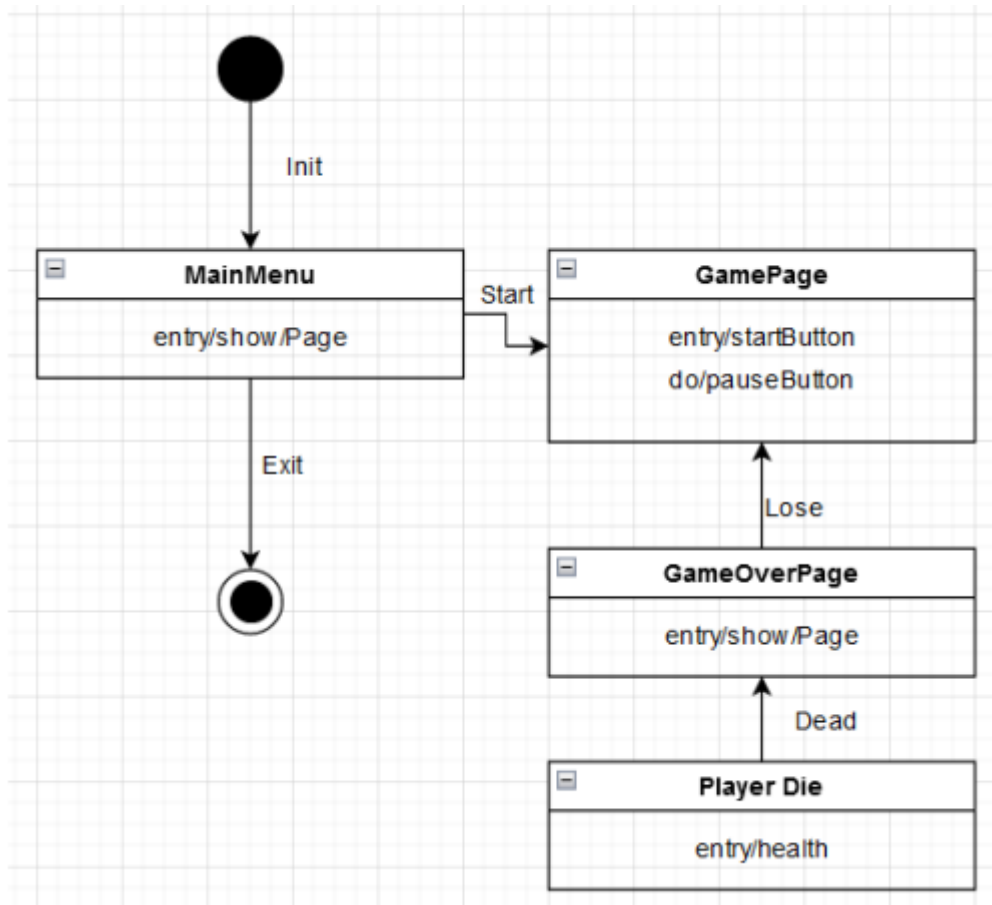


Рис. 2.9. Діаграма станів та переходів поразки

Діаграма станів та переходів процесу гри зображені на (рис. 2.10).

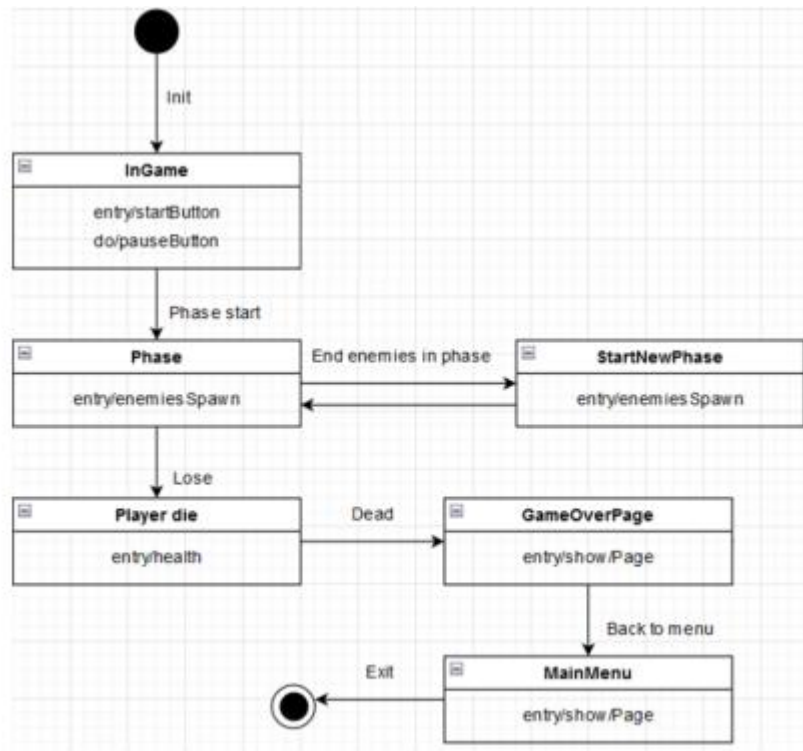


Рис. 2.10. Діаграма станів та переходів процесу гри

Відтворення ігрового процесу демонструє геймплей. На етапі генерації вороги з'являються на сцені, а після їх появи фаза переходить у стан оновлення. Крім того, показано, що у разі поразки гра завершується, однак користувач має можливість перезавантажити рівень і продовжити гру.

### Висновки до розділу

У даному розділі було детально досліджені середовища розробки ігор такі як Unreal Engine 4, Cry Engine 5, Unity. Був проведений аналіз технічних характеристик для роботи з даними ігровими рушіями, а також було проаналізовано їх інтерфейс, наявність навчальної документації та можливості доступу до роботи з середовищами. Оскільки для розробки нашого прототипу було обрано ігровий рушій Unity, був проведений аналіз середовища Microsoft Visual Studio, яке інтегрується з клієнтом Unity та у якому проводиться уся робота з програмним кодом проєкту. Також проведено моделювання гри.

## РОЗДІЛ 3. ОСНОВНІ ЕТАПИ РЕАЛІЗАЦІЇ ІГРОВОГО ПРОЄКТУ

### 3.1. Візуальне оформлення: вибір та використання графічних активів

Візуальна складова є невід'ємною частиною будь-якої комп'ютерної гри, оскільки саме вона формує перше враження у гравця та забезпечує естетичне сприйняття ігрового світу. Для створення привабливої графіки в двовимірних іграх часто використовують такі поняття як спрайт та тайл.

Спрайт - це двовимірне зображення, яке використовується для представлення об'єктів у грі. Спрайти можуть бути статичними або анімованими, що дозволяє створювати динамічні ефекти та персонажів. Тайл - це невеликий фрагмент зображення, який повторюється для створення більших текстур, таких як підлоги, стіни та інші елементи ландшафту. Використання тайлів дозволяє ефективно створювати деталізовані ігрові світи.

Для розробки графічної складової нашого проєкту було прийнято рішення використовувати готові двовимірні спрайти з магазину Unity Asset Store. Такий підхід дозволив зекономити час на створенні власних графічних ресурсів та забезпечив високу якість візуальних ефектів. Були обрані спрайти, які відповідають стилістиці нашого проєкту та гармонійно поєднуються між собою.

Для роботи зі спрайтами в Unity використовується вбудований інструмент Sprite Editor (рис. 3.1). Цей редактор дозволяє імпортувати, редагувати та оптимізувати спрайти, а також створювати атласи спрайтів для більш ефективного використання відеопам'яті.

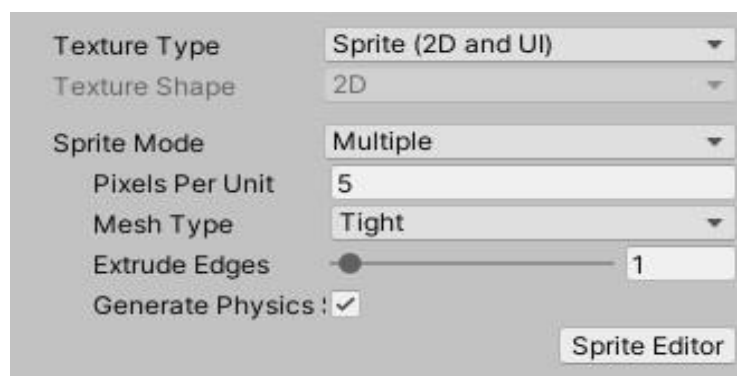


Рис. 3.1. Вікно функції Sprite Editor

Sprite Editor надає можливість імпортувати великі зображення, що містять кілька окремих елементів (спрайтів), та розділити їх на складові частини для

подальшого використання в анімації.

### 3.1.1. Створення анімованого персонажа: астронавт

Для створення динамічного та привабливого ігрового персонажа було обрано зображення астронавта. Цей вибір обумовлений бажанням створити атмосферу космічних пригод та додати оригінальності проєкту.

Анімація персонажа здійснюється шляхом створення набору окремих спрайтів, що відображають різні стани персонажа: спокій, біг, стрибок тощо. Кожен такий спрайт є окремим кадром анімації. Для створення анімації в Unity Sprite Editor дозволяє розділити великий спрайт-атлас на окремі кадри, які потім можуть бути об'єднані в анімацію.

Зброя астронавта також представлена окремим спрайтом. Це дозволяє реалізувати різні види атак та взаємодії з оточенням. Спрайт зброї буде динамічно відображатися на екрані залежно від дій персонажа (рис 3.2).



Рис. 3.2. Спрайти головного героя

### 3.1.2. Спрайти інтерфейсу

Після деталізації зображення головного героя, наступним етапом розробки візуальної складової гри стало формування загального фону та інших графічних елементів інтерфейсу.

Фонове зображення відіграє ключову роль у створенні атмосфери гри. Для нашого проєкту було обрано однотонне синє забарвлення, яке асоціюється з безмежним космічним простором. Такий вибір обумовлений бажанням створити

нейтральний фон, що не відволікатиме увагу від основних елементів гри, а водночас підкреслить тематику космічних подорожей. Синій колір також буде використовуватися для інших елементів інтерфейсу, створюючи тим самим єдину стилістику гри (рис 3.3).

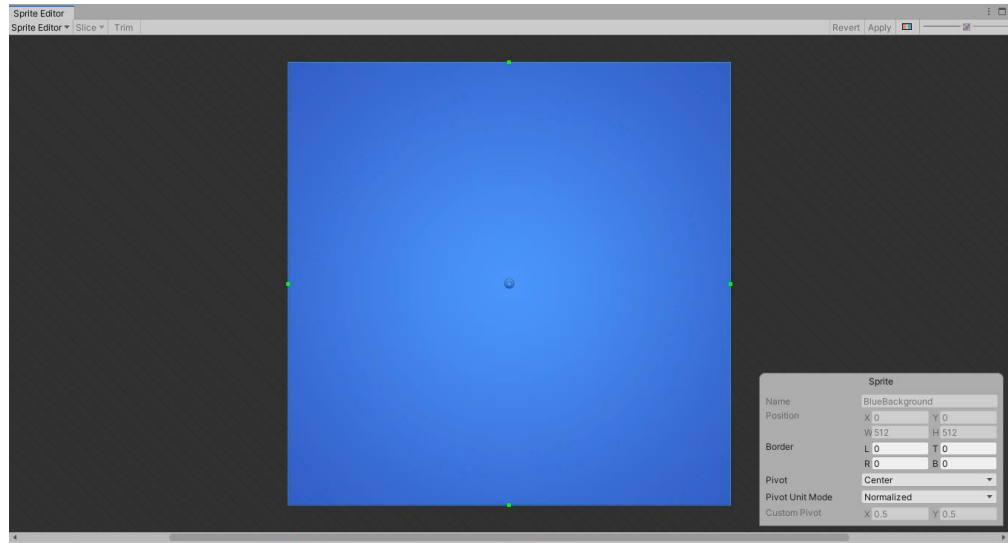


Рис. 3.3. Фонове зображення

Для створення ілюзії прибуття на невідому планету було використано різнобарвні спрайти гір. За допомогою функції Sprite Editor ці зображення були розділені на окремі елементи, що дозволило створити більш деталізований та реалістичний ландшафт (рис 3.4). Це рішення додає грі атмосферності та підкреслює відчуття ізоляції та небезпеки, з якими стикається астронавт.

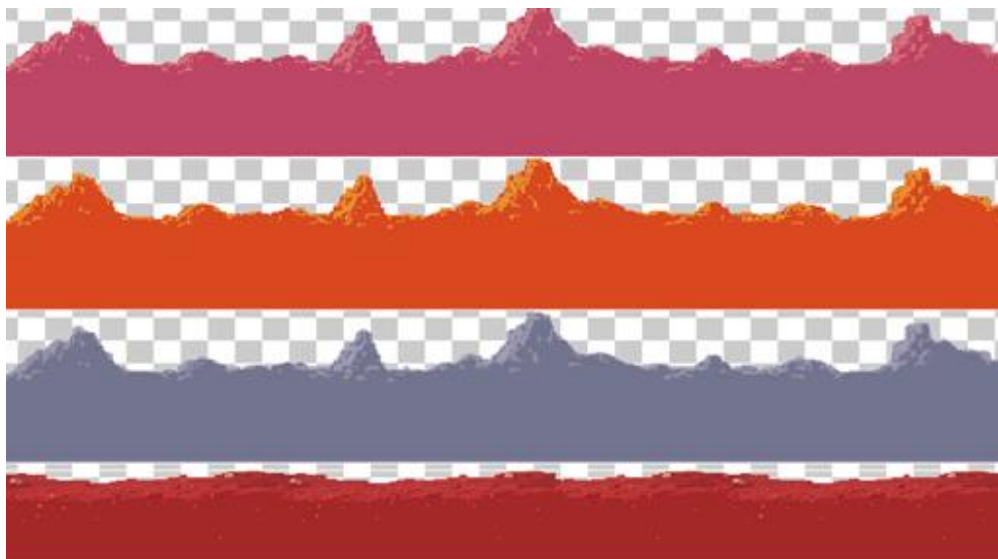


Рис. 3.4. Спрайти гір

З метою оптимізації ресурсів та забезпечення плавного геймплею, було розроблено набір текстур платформ, об'єднаних в атлас спрайтів за допомогою

Sprite Editor. Такий підхід дозволив зменшити кількість окремих файлів та підвищити швидкість завантаження рівнів. Одночасно, єдиний стилістичний напрямок платформ сприяє створенню цілісного ігрового світу, що позитивно впливає на загальне враження від гри (рис 3.5).



Рис. 3.5. Спрайти платформ

### 3.1.3. Візуальне представлення противників: спрайт літаючої тарілки

Для створення антагоністичних сил у грі було обрано стилізоване зображення класичного елемента наукової фантастики – літаючої тарілки з інопланетним пасажиром. Це рішення було прийнято з метою підкреслити тематику космічних подорожей та створити впізнаваний образ ворога.

Використання спрощеного, піксельного стилю для спрайта ворога дозволяє досягти кількох цілей. Спрощений стиль спрайта органічно вписується в загальну візуальну концепцію гри, підкреслюючи її ретрофутуристичний характер. Прості форми та невелика кількість кольорів дозволяють зменшити розмір файлу спрайта, що є важливим для оптимізації гри, особливо для мобільних платформ. Завдяки своїй простоті та характерним рисам, спрайт ворога легко впізнається гравцем навіть на великій відстані. (рис 3.6).



Рис. 3.6. Спрайт ворога



### **3.2. Проектування гри**

Після завершення дизайну візуальних елементів гри, наступним етапом розробки стало вирішення завдання їх ефективного розташування та взаємодії на ігровому полі. Для забезпечення коректної відображення об'єктів та уникнення візуальних артефактів, таких як перекриття елементів, було прийнято рішення використовувати систему шарів (Layers).

Кожному візуальному елементу гри (спрайту) було призначено свій шар. Це дозволяє контролювати порядок відображення об'єктів на екрані. Наприклад, спрайти фону розміщуються на нижніх шарах, а інтерфейсні елементи – на верхніх. Такий підхід забезпечує чітку візуальну ієрархію та дозволяє створювати складні ігрові сцени без втрати якості зображення.

#### **3.2.1. Імплементация фізичних властивостей ігрових об'єктів**

На етапі розробки прототипу ігрового світу ключовим завданням є визначення фізичних властивостей його складових. Цей процес є необхідним для забезпечення реалістичної взаємодії ігрових об'єктів між собою та з оточенням.

Для того, щоб ігровий персонаж міг взаємодіяти з платформами (стрибати, бігати), необхідно надати їм фізичні властивості. Для цього до кожного спрайту платформи додається компонент Box Collider 2D. Цей компонент створює навколо спрайту невидимий об'єм у формі прямокутного паралелепіпеда, який використовується для визначення зіткнень з іншими об'єктами. Таким чином, ігровий двигун розуміє, що платформа є твердим тілом, з яким персонаж може взаємодіяти.

Ігровий персонаж, на відміну від статичних платформ, має більш складний набір фізичних властивостей. Для забезпечення плавного руху та реалістичної взаємодії з оточенням, персонажу надається компонент Capsule Collider 2D. Цей компонент створює навколо спрайта персонажа невидимий об'єм у формі капсули, що дозволяє більш точно моделювати форму людського тіла.

Крім того, для того, щоб ігровий двигун міг обчислювати рух персонажа під дією фізичних сил (гравітація, сили тертя), до нього додається компонент Rigidbody 2D. Цей компонент надає об'єкту масу, інерцію та дозволяє

застосовувати до нього різноманітні сили. Завдяки Rigidbody 2D персонаж може реагувати на зіткнення з іншими об'єктами, стрибати, падати та виконувати інші дії.

### 3.2.2. Рух об'єктів

Після визначення фізичних властивостей ігрового персонажа, наступним кроком є програмування його руху та взаємодії з оточенням. Для реалізації цієї функціональності необхідно створити скрипт, який буде керувати поведінкою персонажа.

На початку скрипту оголошуються наступні змінні:

`Rigidbody2D rb`: змінна типу `Rigidbody2D`, яка буде використовуватися для доступу до фізичних властивостей персонажа та застосування до нього сил.

`public float speed`: публічна змінна типу `float`, яка визначає швидкість горизонтального переміщення персонажа.

`public float jumpHeight`: публічна змінна типу `float`, яка визначає висоту стрибка персонажа.

`public Transform groundCheck`: публічна змінна типу `Transform`, яка посилається на порожній об'єкт, розміщений трохи нижче ніг персонажа. Цей об'єкт використовується для визначення того, чи стоїть персонаж на поверхні.

`bool isGrounded`: булева змінна, яка зберігає інформацію про те, чи контактує персонаж з поверхнею.

Для реалізації механіки стрибків необхідно виконати наступні дії:

1. Визначення моменту стрибка: за допомогою тригера (наприклад, натискання клавіші) визначаємо момент, коли гравець хоче здійснити стрибок.
2. Перевірка контакту з поверхнею: перед виконанням стрибка перевіряємо значення змінної `isGrounded`. Якщо персонаж контактує з поверхнею (значення `isGrounded` дорівнює `true`), то дозволяємо виконати стрибок.
3. Застосування сили стрибка: якщо умова стрибка виконана, то до компонента `Rigidbody2D` персонажа застосовується сила,

спрямована вертикально вгору. Величина цієї сили визначається значенням змінної `jumpHeight`.

Для визначення того, чи контактує персонаж з поверхнею, використовується змінна `groundCheck`. Ця змінна посилається на порожній об'єкт, який розміщується трохи нижче ніг персонажа. Регулярно перевіряється, чи відбулося зіткнення цього об'єкта з яким-небудь коллайдером. Якщо зіткнення відбулося, то значення змінної `isGrounded` встановлюється в `true`, в іншому випадку – в `false`.

Для реалізації механіки стрибків та обмеження кількості стрибків у повітрі використовується функція `CheckGround()`. Ця функція перевіряє, чи знаходиться персонаж на поверхні, аналізуючи наявність коллайдерів у заданому радіусі навколо точки `groundCheck`.

```
void CheckGround()
{
    // Виконуємо перевірку на перетин з іншими коллайдерами в
    заданому радіусі
    Collider2D[] colliders =
    Physics2D.OverlapCircleAll(groundCheck.position, 0.2f);
    // Якщо знайдено хоча б один коллайдер (крім власного), то
    вважаємо, що персонаж на землі
    isGrounded = colliders.Length > 1;
}
```

Для реалізації горизонтального руху персонажа використовується метод `GetAxis("Horizontal")` класу `Input`. Цей метод повертає значення від -1 до 1, яке відображає напрямок руху по горизонтальній осі. Отримане значення множиться на швидкість руху `speed` і присвоюється горизонтальній складовій вектора швидкості `rb.velocity`. Вертикальна складова швидкості залишається незмінною, що дозволяє персонажу зберігати поточну вертикальну швидкість (наприклад, при стрибку).

```
rb.velocity = new Vector2(Input.GetAxis("Horizontal") * speed,
rb.velocity.y);
```

Для зміни напрямку орієнтації персонажа використовується метод `Flip()`. Цей метод перевіряє напрямок руху персонажа за допомогою `Input.GetAxis("Horizontal")` і обертає спрайт персонажа на 180 градусів навколо осі Y, якщо напрямок руху змінився.

```
void Flip()
{
    if (Input.GetAxis("Horizontal") > 0)
        transform.localRotation = Quaternion.Euler(0, 0, 0);
    else if (Input.GetAxis("Horizontal") < 0)
        transform.localRotation = Quaternion.Euler(0, 180, 0);
}
```

Для оптимізації можна використовувати кешовані значення, наприклад, зберегти попереднє значення `Input.GetAxis("Horizontal")`, щоб уникнути повторних обчислень.

Для створення більш плавної анімації руху персонажа можна використовувати аніматор і змінювати параметри анімації залежно від напрямку руху та стану персонажа (біг, стрибок, стояння).

Для стрибків на платформах з різною висотою можна використовувати різні значення сили стрибка або динамічно змінювати висоту стрибка залежно від властивостей платформи.

Для реалізації механіки стрибків використовується наступний фрагмент коду:

```
if (Input.GetKeyDown(KeyCode.Space) && isGrounded)
{
    rb.AddForce(transform.up * jumpHeight, ForceMode2D.Impulse);
}
```

`Input.GetKeyDown(KeyCode.Space)`: перевіряє, чи була натиснута клавіша Space. `isGrounded`: перевіряє, чи знаходиться персонаж на землі (результат функції `CheckGround()`). `rb.AddForce(transform.up * jumpHeight, ForceMode2D.Impulse)`: додає імпульсну силу до `Rigidbody2D` персонажа. `transform.up`: вектор, що вказує вгору відносно об'єкта. `jumpHeight`: висота

стрибка, задана у змінній. `ForceMode2D.Impulse`: режим додавання сили, який застосовує силу миттєво.

Таким чином, при натисканні клавіші `Space` і за умови, що персонаж знаходиться на землі, до нього застосовується сила, що штовхає його вгору.

### 3.2.3. Створення анімацій для персонажа

Для створення анімацій персонажа необхідно виконати наступні кроки:

- 1) У редакторі спрайтів розділити великий спрайт персонажа на окремі спрайти, що відповідають різним станам (ходьба, біг, стрибок, і так далі).
- 2) Додати компоненту `Animator` до об'єкта персонажа.
- 3) Для кожного набору спрайтів створити новий кліп анімації в редакторі анімацій. Встановити тривалість кліпу та налаштувати переходи між кліпами.
- 4) Прив'язати кліпи анімації до параметрів контролера анімації. Наприклад, параметр `"Speed"` може керувати переходом між анімаціями ходьби і бігу. (рис 3.7).

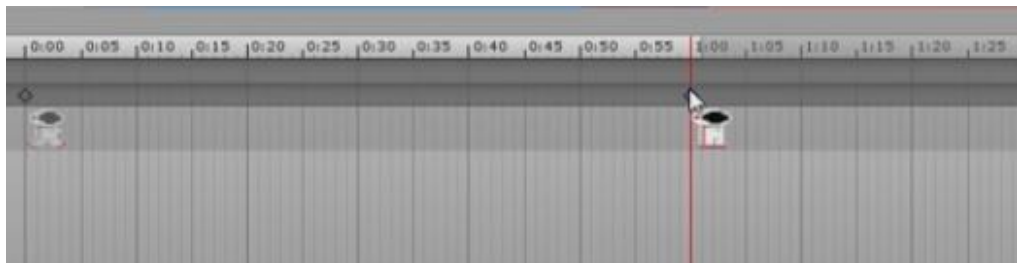


Рис. 3.7. Вікно Animation

Наступним етапом є конфігурація переходів між анімаціями, що дозволить плавно переходити від однієї дії до іншої (наприклад, від ходьби до стрибка) відповідно до ігрових подій. Це досягається шляхом встановлення зв'язків між різними анімаційними кліпами в контролері анімацій (рис. 3.8).

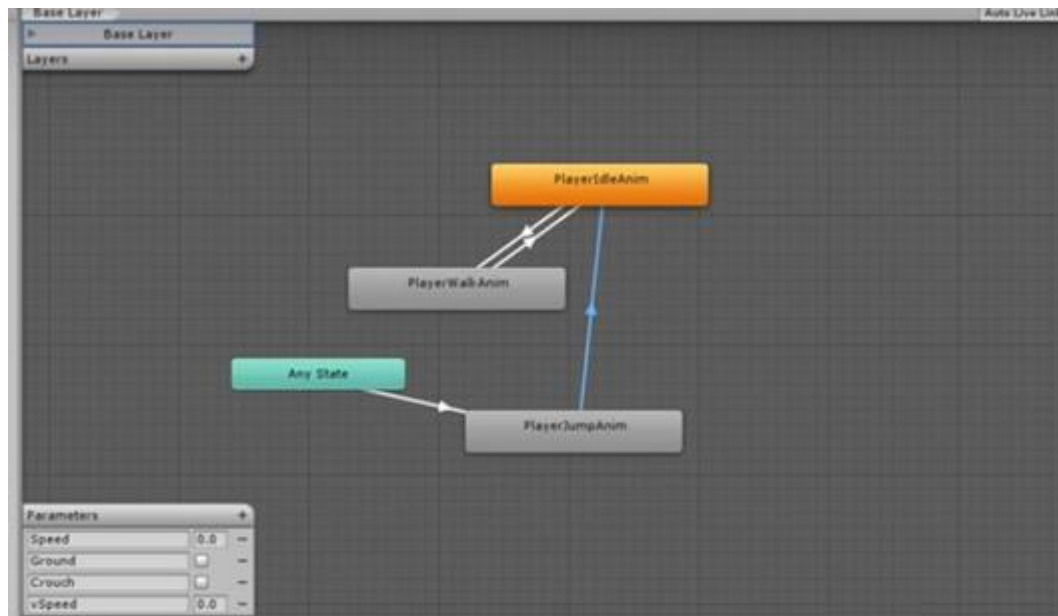


Рис. 3.8. Вікно зв'язків анімацій

### 3.2.4. Реалізація вогнепальної зброї

Після створення анімацій та рухів персонажа, логічним кроком є додавання функціоналу стрільби. Для цього ми інтегруємо в ігровий об'єкт зброю, налаштовуємо її характеристики та реалізуємо механіку пострілу.

Для керування зброєю вводимо наступні змінні:

fireRate: частота стрільби, визначає кількість пострілів за секунду.

Damage: величина шкоди, що наноситься при пострілі.

whatToHit: маска шарів, що визначає, з якими об'єктами може взаємодіяти куля.

```
public float fireRate = 0;
```

```
public float Damage = 10;
```

```
public LayerMask whatToHit;
```

Для реалізації механіки стрільби створюємо функцію Shoot(), яка виконується при натисканні на кнопку стрільби. Функція перевіряє, чи минув достатній час з моменту попереднього пострілу, і якщо так, то створює кулю та додає їй необхідні сили.

```
void Shoot()
{
    // Створення кулі (prefab)
```

```

        Instantiate(bulletPrefab, firePoint.position,
firePoint.rotation);
    }

```

Для обмеження швидкості стрільби використовується змінна `timeToFire`. При кожному пострілі ця змінна оновлюється, і наступний постріл може бути здійснений тільки після закінчення заданого інтервалу часу.

```

if (fireRate == 0)
{
    if (Input.GetButtonDown("Fire1"))
    {
        Shoot();
    }
}
else
{
    if (Input.GetButton("Fire1") && Time.time > timeToFire)
    {
        timeToFire = Time.time + 1 / fireRate;
        Shoot();
    }
}

```

Для визначення об'єктів, які можуть бути уражені кулею, використовується маска шарів `whatToHit`. Ця маска дозволяє фільтрувати зіткнення кулі з іншими об'єктами в сцені.

// Приклад використання маски шарів при зіткненні кулі з іншим об'єктом:

```

void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.layer == whatToHit)
    {
        // Застосування шкоди до об'єкту
        Health enemyHealth =
collision.gameObject.GetComponent<Health>();
        if (enemyHealth != null)

```

```

        {
            enemyHealth.TakeDamage(Damage);
        }
    }
}

```

Для створення більш інтерактивного та візуально привабливого геймплею, необхідно синхронізувати напрямок пострілу з положенням курсору миші та візуалізувати траєкторію кулі.

Для визначення точки, в яку спрямований постріл, використовується метод `Camera.main.ScreenToWorldPoint`. Цей метод перетворює координати курсору миші в екранових координатах у світові координати гри.

```

Vector2 mousePosition = new
Vector2(Camera.main.ScreenToWorldPoint(Input.mousePosition).x,
Camera.main.ScreenToWorldPoint(Input.mousePosition).y);

```

Для візуалізації траєкторії кулі використовується метод `Physics2D.Raycast`. Цей метод створює промінь (`ray`) з точки стрільби в напрямку курсору миші і перевіряє, чи перетинається цей промінь з якимись об'єктами в сцені.

```

Vector2 firePointPosition = new Vector2(firePoint.position.x,
firePoint.position.y);
RaycastHit2D hit = Physics2D.Raycast(firePointPosition,
mousePosition - firePointPosition, 100, whatToHit);

```

`firePointPosition`: початок променя (точка стрільби).

`mousePosition - firePointPosition`: напрямок променя (вектор від точки стрільби до точки прицілювання).

`100`: максимальна відстань, на яку поширюється промінь.

`whatToHit`: маска шарів, що визначає, з якими об'єктами може взаємодіяти промінь.

Для візуалізації променя можна використовувати різні підходи:

`Debug.DrawLine`: цей метод дозволяє малювати лінії в `Scene View` для відлагодження.

`Line Renderer`: цей компонент дозволяє створювати лінії в ігровому світі.



Prefab кулі: можна інстанціювати префаб кулі і задати його положення і масштаб відповідно до результатів Raycast.

```
// Приклад використання Debug.DrawLine
```

```
Debug.DrawLine(firePointPosition, hit.point, Color.red);
```

Якщо промінь перетнув якийсь об'єкт, то в змінній hit буде міститися інформація про точку перетину та інший корисний дані. Цю інформацію можна використовувати для застосування шкоди, відтворення звукових ефектів або інших дій.

### 3.2.5. Гибель та відродження героя

Для створення більш динамічної та цікавої гри необхідно реалізувати механіку смерті та відродження персонажа. Це дозволить гравцеві продовжувати гру після загибелі, але при цьому створить додаткові виклики.

Для відстеження стану здоров'я персонажа створюємо клас PlayerStats:

```
public class PlayerStats {  
    public int Health = 100;  
}
```

Функція DamagePlayer зменшує кількість очок здоров'я та перевіряє, чи не загинув персонаж:

```
void DamagePlayer(int damage) {  
    playerStats.Health -= damage;  
    if (playerStats.Health <= 0) {  
        GameManager.KillPlayer(this);  
    }  
}
```

Для симуляції смерті від падіння вводимо змінну fallBoundary, яка визначає нижню межу ігрового поля. Якщо координат Y персонажа стає меншою за це значення, то персонаж вважається загиблим:

```
public int fallBoundary = -20;  
if (transform.position.y <= fallBoundary) {  
    DamagePlayer(9999999);  
}
```

Для реалізації механіки відродження необхідно:

Prefab персонажа: змінна `playerPrefab` містить посилання на префаб персонажа, який буде використовуватися для створення нового екземпляра при відродженні.

Точка відродження: змінна `spawnPoint` визначає позицію, в якій буде з'являтися персонаж після відродження.

Затримка відродження: змінна `spawnDelay` визначає час, через який персонаж буде відроджений після смерті.

```
public Transform playerPrefab;
```

```
public Transform spawnPoint;
```

```
public int spawnDelay = 2;
```

Функція відродження (приклад реалізації в класі `GameMaster`)

```
public void KillPlayer(Player player)
{
    Destroy(player.gameObject);
    Invoke("RespawnPlayer", spawnDelay);
}
void RespawnPlayer()
{
    Instantiate(playerPrefab, spawnPoint.position,
spawnPoint.rotation);
}
```

Для створення нового екземпляру персонажа в заданій точці використовуємо компонент `Transform`. Змінна `playerPrefab` містить посилання на префаб персонажа, який буде клонований, а `spawnPoint` визначає позицію і обертання, в які буде поміщений новий екземпляр.

```
public Transform playerPrefab; // Префаб персонажа
```

```
public Transform spawnPoint; // Точка відродження
```

Для затримки відродження персонажа використовується корутина. Корутина дозволяє тимчасово зупинити виконання функції і продовжити його пізніше.

```
public static void KillPlayer(Player player) {
    Destroy(player.gameObject);
}
```

```

        gm.StartCoroutine(gm.RespawnPlayer());
    }

```

Destroy(player.gameObject): видаляє ігровий об'єкт персонажа.

gm.StartCoroutine(gm.RespawnPlayer()): запускає корутину RespawnPlayer в класі GameMaster.

```

IEnumerator RespawnPlayer() {
    yield return new WaitForSeconds(spawnDelay);
    Instantiate(playerPrefab, spawnPoint.position, spawnPoint.rotation);
}

```

yield return new WaitForSeconds(spawnDelay): зупиняє виконання корутини на spawnDelay секунд.

Instantiate(playerPrefab, spawnPoint.position, spawnPoint.rotation): створює новий екземпляр префаба персонажа в заданій позиції і з заданим обертанням.

### 3.2.6. Штучний інтелект ворогів: навігація та поведінка

Для створення більш складних і цікавих ігрових ситуацій необхідно реалізувати штучний інтелект ворогів. Це дозволить ворогам переслідувати гравця, уникати перешкод і приймати інші розумні рішення.

Для реалізації навігації ворогів використовується зовнішня бібліотека A\* Pathfinding Project. Ця бібліотека надає ефективний алгоритм пошуку найкоротшого шляху, який дозволяє ворогам обходити перешкоди і досягати заданої точки.

```

public Transform target; // Ціль для переслідування (гравець)
public float updateRate = 2f; // Частота оновлення шляху
private Seeker seeker; // Компонент для пошуку шляху
private Rigidbody2D rb; // Компонент для фізичних властивостей
public float speed = 300f; // Швидкість руху ворога
public ForceMode2D fMode; // Режим застосування сили

```

Для відстеження стану здоров'я ворога створюється клас EnemyStats:

```

public class EnemyStats {
    public int Health = 100;
}

```

Функція DamageEnemy зменшує кількість очок здоров'я ворога і перевіряє, чи не загинув він:

```
public void DamageEnemy(int damage) {  
    stats.Health -= damage;  
    if (stats.Health <= 0) {  
        GameManager.KillEnemy(this);  
    }  
}
```

Рух ворога по шляху

У методі Start отримуємо компоненти Seeker і Rigidbody2D, а також перевіряємо, чи задана ціль для переслідування.

За допомогою методу seeker.StartPath запускаємо пошук шляху до цілі. Цей метод асинхронний, тому результат пошуку буде доступний пізніше.

У методі FixedUpdate перевіряємо, чи знайдено шлях. Якщо шлях знайдено, то рухаємо ворога до наступної точки шляху.

```
void FixedUpdate() {  
    if (path == null)  
        return;  
    if (currentWaypoint >= path.vectorPath.Count)  
        return;  
    Vector3 dir = (path.vectorPath[currentWaypoint] -  
transform.position).normalized;  
    rb.AddForce(dir * speed * Time.fixedDeltaTime, fMode);  
    float dist = Vector3.Distance(transform.position,  
path.vectorPath[currentWaypoint]);  
    if (dist < nextWaypointDistance) {  
        currentWaypoint++;  
        return;  
    }  
}
```

Додаткові можливості:

- Можна створити різні типи ворогів з різною поведінкою (наприклад, дальній бій, ближній бій, літаючі вороги).

- Вороги можуть змінювати свою поведінку залежно від ситуації (наприклад, переходити від патрулювання до переслідування гравця).
- Вороги можуть мати обмежену видимість або чути звуки, що впливає на їхню поведінку.
- Вороги можуть діяти групами, координуючи свої дії.

### 3.2.7. Графічний інтерфейс для героя та ворога

Наступним етапом розробки є впровадження графічного інтерфейсу користувача (GUI). Першим кроком буде створення смужок здоров'я для ворогів та гравця (див. рис. 3.9). Для цього ми створимо в Unity порожній об'єкт, який буде містити скрипт, що відповідає за оновлення та відображення стану здоров'я.

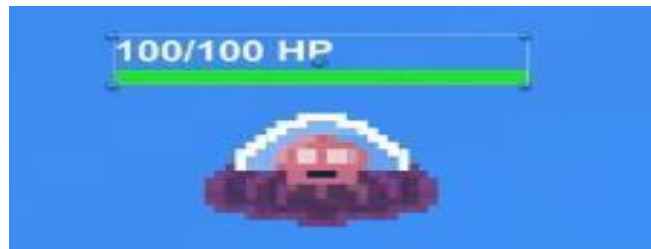


Рис. 3.9. Смуга інтерфейсу життя ворога

Функція `SetHealth` відповідає за візуальне представлення стану здоров'я ігрового об'єкта. Вона приймає поточне та максимальне значення здоров'я, обчислює відношення цих значень і масштабує прямокутник, що представляє смугу здоров'я, відповідно до отриманого результату. Текстове поле над смугою оновлюється, відображаючи поточне і максимальне значення здоров'я.

`public void SetHealth(int _cur, int _max):` оголошує функцію `SetHealth`, яка приймає два цілих числа: `_cur` (поточне здоров'я) і `_max` (максимальне здоров'я).

`float _value = (float)_cur / _max;:` обчислюємо відношення поточного здоров'я до максимального і перетворюємо результат в число з плаваючою точкою. Це значення буде використовуватися для масштабування смужки здоров'я.

`healthBarRect.localScale = new Vector3(_value, healthBarRect.localScale.y, healthBarRect.localScale.z);:` змінюємо масштаб прямокутника, який представляє смугу здоров'я. Значення по осі `X` встановлюється рівним обчисленій величині

\_value, тобто смужка буде заповнена пропорційно відношенню поточного здоров'я до максимального. Висота і глибина смужки залишаються незмінними.

healthText.text = \_cur + "/" + \_max + " HP";: оновлюємо текст, який відображає поточне і максимальне значення здоров'я.

Копіюємо у вікні сцени створений скрипт до об'єкту гравця, та маємо такий саме результат графічному інтерфейсом на ігровому персонажі (рис 3.10).



Рис. 3.10. Смуга інтерфейсу життя героя

### 3.2.8. Взаємодія між об'єктами

Реалізуємо механіку завдання шкоди. Введемо змінну Damage, що визначає силу атаки гравця. При зіткненні снаряду з ворожим об'єктом отримуємо доступ до скрипту ворога за допомогою hit.collider.GetComponent<Enemy>() і викликаємо метод DamageEnemy, передаючи йому значення завданої шкоди. Для відлагодження додаємо повідомлення в консоль.

```
public void DamageEnemy(int damage)
{
    health -= damage;
    healthBar.SetHealth(health, maxHealth); // Оновлюємо смугу
здоров'я

    if (health <= 0)
    {
        // Ворог загинув, виконуємо необхідні дії, наприклад,
відтворення анімації смерті, видалення об'єкта
        Destroy(gameObject);
    }
}
```

Тепер ми зробимо так, щоб ворог міг завдавати шкоди нашому герою. Коли герой отримує удар, ми віднімаємо кількість завданих очок здоров'я від його поточного здоров'я. Якщо здоров'я героя стане нулем або менше, то він програє. Також ми оновлюємо смугу здоров'я на екрані, щоб гравець бачив, скільки здоров'я у нього залишилося. Ми зробимо те саме і для ворога, щоб він теж міг отримувати ушкодження і гинути.

`public void DamagePlayer(int damage):` функція викликається, коли гравець отримує ушкодження.

`stats.curHealth -= damage;` зменшуємо поточне здоров'я гравця на значення `damage`.

`if (stats.curHealth <= 0):` перевіряємо, чи не стало здоров'я гравця менше або рівним нулю. Якщо так, то гравець гине.

`GameMaster.KillPlayer(this);` викликаємо функцію `KillPlayer` в класі `GameMaster`, яка відповідає за обробку смерті гравця (наприклад, перезапуск рівня, відображення екрану смерті).

`statusIndicator.SetHealth(stats.curHealth, stats.maxHealth);` оновлюємо індикатор здоров'я на екрані, передаючи йому поточне і максимальне значення здоров'я.

`DamageEnemy:` функція працює аналогічно, але застосовується до ворогів.

Для реалізації механіки миттєвої загибелі ворога при зіткненні з гравцем використовується функція `OnCollisionEnter2D`. При зіткненні з гравцем ворог завдає йому шкоди, а потім сам отримує смертельну дозу ушкоджень, що призводить до його знищення.

Покрокове пояснення коду:

`void OnCollisionEnter2D(Collision2D _colInfo):` функція викликається, коли відбувається зіткнення між двома об'єктами, що мають коллайдери.

`Player _player = _colInfo.collider.GetComponent<Player>();` частина коду отримує компонент `Player` з об'єкта, з яким відбулося зіткнення. Якщо зіткнення відбулося з гравцем, то змінна `_player` буде містити посилання на скрипт гравця.

`_player.DamagePlayer(stats.damage);`:: викликаємо функцію `DamagePlayer` у скрипті гравця, передаючи їй значення завданої шкоди.

`DamageEnemy(9999999);`:: викликаємо функцію `DamageEnemy` для самого себе (ворога), передаючи їй дуже велике значення шкоди, що гарантує його знищення.

### 3.2.9. Генерація хвиль ворогів

Реалізуємо механіку спавну хвиль ворогів. Створимо систему, яка дозволяє задавати кількість ворогів у хвилі, інтервал між хвилями та точки спавну. Для управління процесом спавну використовуємо скінченний автомат з трьома станами: SPAWNING, WAITING, COUNTING.

```
public class WaveSpawner : MonoBehaviour
{
    // ... змінні ...
    void Update()
    {
        switch (state)
        {
            case State.SPAWNING:
                // Спавним ворогів
                break;
            case State.WAITING:
                // Чекаємо
                break;
            case State.COUNTING:
                // Відраховуємо час
                break;
        }
    }
    void SpawnEnemy(Transform _enemy)
    {
        // ... код функції ...
    }
}
```



Перевірка наявності живих ворогів

```
bool EnemyIsAlive()
{
    // Кожні секунду перевіряємо, чи є живі вороги
    searchCountdown -= Time.deltaTime;
    if (searchCountdown <= 0f)
    {
        searchCountdown = 1f;
        // Шукаємо об'єкти з тегом "Enemy"
        if (GameObject.FindGameObjectWithTag("Enemy") == null)
        {
            return false; // Якщо не знайшли жодного, повертаємо
false
        }
    }
    return true; // Інакше повертаємо true (є живі вороги)
}
```

Ця функція використовується для періодичної перевірки, чи залишилися живі вороги на сцені. Вона шукає об'єкти з тегом "Enemy" і повертає true, якщо такі об'єкти знайшлися.

Перехід до наступної хвилі

```
if (waveCountdown <= 0)
{
    if (state != SpawnState.SPAWNING)
    {
        StartCoroutine(SpawnWave(waves[nextWave]));
    }
}
```

Якщо таймер для наступної хвилі закінчився, і ми не знаходимося в процесі спавну, то запускаємо корутину SpawnWave, яка відповідає за появу нової хвилі ворогів.

Спавн хвилі

```
IEnumerator SpawnWave(Wave _wave)
{
```

```

        // ... код функції ...
    }

```

Ця функція ітерує по всіх ворогах у хвилі, спавнюючи їх з певним інтервалом. Після завершення спавну всіх ворогів, переводить стан в WAITING.

Перевірка на завершення хвилі та початок нової

```

if (state == SpawnState.WAITING)
{
    if (!EnemyIsAlive())
    {
        WaveCompleted();
    }
    else
    {
        return;
    }
}

```

Якщо ми перебуваємо в стані очікування і всі вороги знищені, то викликаємо функцію WaveCompleted().

Завершення хвилі та перехід до наступної

```

void WaveCompleted()
{
    // ... код функції ...
}

```

Ця функція обробляє завершення хвилі: скидає таймер для наступної хвилі, перемикає стан і збільшує номер наступної хвилі. Якщо це була остання хвиля, то цикл хвиль починається спочатку.

### 3.2.10. Графічний інтерфейс до генератору хвиль

Для створення більш інтуїтивного користувацького інтерфейсу, ми додамо до гри елемент, який буде відображати номер поточної хвилі ворогів. Це буде реалізовано за допомогою трьох текстових об'єктів, які будуть динамічно оновлюватися відповідно до поточного стану гри. Таким чином, гравець завжди

буде знати, на якому етапі він знаходиться і може планувати свої дії відповідно (рис 3.11).



Рис. 3.11. Відображення поточної хвилі

Оновлення інтерфейсу залежно від стану спавнера. Цей фрагмент коду забезпечує синхронізацію між станом спавнера хвиль і графічним інтерфейсом гри. Щоб зробити його більш зрозумілим, розглянемо кожен частину окремо.

Основна функція Update:

```
void Update () {  
    // Перевіряємо стан спавнера і викликаємо відповідну функцію  
    оновлення інтерфейсу  
    switch (spawner.State)  
    {  
        case WaveSpawner.SpawnState.COUNTING: UpdateCountingUI();  
break;  
        case WaveSpawner.SpawnState.SPAWNING: UpdateSpawningUI();  
break;  
    }  
    // Запам'ятовуємо попередній стан для наступної ітерації  
    previousState = spawner.State;  
}
```

Ця функція виконується кожен кадр і перевіряє поточний стан спавнера хвиль. Залежно від стану, викликається відповідна функція для оновлення графічного інтерфейсу.

Оновлення інтерфейсу під час відліку до хвилі:

```
void UpdateCountingUI ()  
{  
    // ... код ...
```

```
}
```

Ця функція відповідає за відображення інформації про відлік часу до початку нової хвили. Вона включає анімацію і оновлення тексту таймера.

Оновлення інтерфейсу під час спавну хвили:

```
void UpdateSpawningUI()
{
    // ... код ...
}
```

Ця функція відповідає за відображення повідомлення про початок нової хвили. Вона також включає анімацію і оновлення тексту з номером хвили.

Функції UpdateCountingUI і UpdateSpawningUI взаємодіють з елементами графічного інтерфейсу, такими як анімації і текстові поля. Це дозволяє динамічно змінювати зовнішній вигляд гри залежно від стану спавнера хвиль:

UpdateCountingUI: включає анімацію відліку. Оновлює текст таймера, відображаючи залишився час до початку хвили.

UpdateSpawningUI: включає анімацію початку хвили. Відображає номер наступної хвили. Зв'язок з графічним інтерфейсом

### 3.2.11. Умови для завершення гри

Для реалізації механіки смерті гравця введемо змінну `_remainingLives`, яка буде зберігати кількість залишившихся життів. Також створимо відповідний елемент інтерфейсу для відображення цієї інформації користувачеві.

```
private static int _remainingLives = 3;;
```

Оголошується приватна статична змінна `_remainingLives` і ініціалізується значенням 3. Це означає, що спочатку гравець має 3 життя.

`private`: змінна доступна тільки всередині класу, де вона оголошена.

`static`: змінна є спільною для всіх екземплярів класу. Це означає, що всі об'єкти цього класу будуть мати доступ до однієї і тієї ж змінної.

`public static int RemainingLives`: властивість, яка дозволяє отримати доступ до значення змінної `_remainingLives` з інших частин коду.

`get`: аксесор дозволяє отримати поточне значення змінної.

Далі розробляємо:

Графічний інтерфейс: створення візуального елемента (наприклад, серцечок або цифри), який буде відображати кількість залишившихся життів.

Механіка втрати життя: додавання логіки, яка буде зменшувати кількість життів при певних умовах (наприклад, зіткнення з ворогом).

Обробка кінця гри: реалізація дій, які будуть виконуватися при втраті всіх життів (наприклад, відображення екрану кінця гри, перезапуск рівня).

```
// Зменшення кількості життів
void LoseLife()
{
    _remainingLives--;
    if (_remainingLives <= 0)
    {
        // Викликаємо функцію, яка обробляє кінець гри
        GameOver();
    }
}
```



Рис 3.12. Інтерфейс поточної кількості життів героя.

Коли гравець втрачає всі свої життя, гра закінчується. Тому ми додамо до функції `KillPlayer` умову: якщо кількість життів стала рівною нулю, то викликається функція `EndGame()`, яка відповідає за завершення гри. Якщо ж життів залишилося більше нуля, то гравець відроджується у визначеній точці.

```
private Text livesText;
```

Оголошується приватна змінна `livesText` типу `Text`, яка буде використовуватися для посилання на текстовий об'єкт на сцені, де буде відображатися кількість життів.

```
livesText = GetComponent<Text>();
```

В методі `Awake` (викликається перед першим кадром) присвоюється посилання на компонент `Text`, який знаходиться на тому ж об'єкті, де

прикріплений цей скрипт. Тобто, ми знаходимо текстове поле на сцені і зберігаємо посилання на нього в змінній `livesText`.

```
livesText.text = "LIVES: " + GameManager.RemainingLives.ToString();
```

В методі `Update` (викликається кожен кадр) оновлюється текст текстового поля. Ми беремо значення змінної `RemainingLives` з класу `GameManager`, перетворюємо його в рядок і додаємо перед ним текст `"LIVES: "`.

```
if (_remainingLives <= 0)
{
    gm.EndGame();
} else
{
    gm.StartCoroutine(gm._RespawnPlayer());
}
```

Якщо кількість життів стала менше або рівною нулю, викликається метод `EndGame()` в об'єкті `gm` (ймовірно, це `GameManager`), який відповідає за завершення гри.

Якщо ж життів залишилося більше нуля, то викликається корутина `_RespawnPlayer()` в об'єкті `gm`, яка відповідає за відродження гравця.

### 3.2.12. Меню початку та кінця гри

Розробимо два окремі екрани: один для початку гри, а другий для її завершення. На екрані початку буде кнопка `"Почати гру"`, яка ініціює початок ігрового процесу. Екран завершення гри міститиме повідомлення про програш, кнопку `"Повторити"` для перезапуску гри та кнопку `"Вийти"` для закриття гри. Ці екрани забезпечать користувача зрозумілим та інтуїтивним інтерфейсом для керування грою (рис 3.13).



Рис. 3.13. Інтерфейс завершення гри

Ми створимо дві прості функції, які дозволяють гравцеві керувати грою:

- `Quit()`: функція повністю закриває гру.
- `Retry()`: функція перезапускає поточну сцену, тобто перезапускає рівень.

Створимо окрему сцену, яка буде служити головним меню гри. На цьому екрані будуть дві кнопки (рис 3.14):

- `Play`: при натисканні цієї кнопки гра почнеться, і гравець перейде до першого рівня.
- `Quit`: кнопка, як і раніше, повністю закриє гру."



Рис. 3.14. Кнопки головного меню

Для того, щоб наше головне меню працювало, ми напишемо невеликий програмний код. У ньому будуть дві функції:

`StartGame()`:функція використовує вбудовану функцію `Unity SceneManager.LoadScene` для завантаження наступної сцени. Тобто, коли гравець натискає на кнопку "Грати", він переходить на сцену, де починається сама гра.

`QuitGame()`:використовує функцію `Application.Quit()` для того, щоб повністю закрити гру.

Кожна кнопка в головному меню буде пов'язана з однією з цих функцій. Коли гравець натискає на кнопку, викликається відповідна функція. Функція `StartGame` відповідає за перехід між сценами в `Unity`. Це дозволяє нам створювати різні екрани для меню, гри, налаштувань тощо. Функція `QuitGame` використовується для завершення роботи гри.

Таким чином, коли гравець натисне на кнопку "Play", скрипт виконає функцію `StartGame` і переведе його на сцену з грою. А якщо гравець натисне на кнопку "Quit", гра просто закриється.

### **3.3. Інструкція користувача**

#### **3.3.1. Стартове меню**

Стартове меню гри представлено у вигляді простого та лаконічного інтерфейсу, що складається з двох основних елементів керування: кнопки "Грати" та "Вийти". Вибір користувача визначає подальші дії програми: запуск ігрового процесу або завершення роботи. Для створення приємного візуального враження було використано мінімалістичний дизайн з переважанням синього кольору. Графічні елементи меню, такі як зображення інопланетної істоти та платформи, відповідають загальній тематиці гри та створюють відповідну атмосферу (рис 3.15).



Рис. 3.15. Стартове меню гри

#### **3.3.2. Інтерфейс та ігровий процес**

Після ініціації гри користувач керує персонажем за допомогою клавіатури (A – рух вліво, D – рух вправо, Space – стрибок) та миші (ЛКМ – стрільба). Ігровий світ представлений двовимірною площиною, на якій розташовані платформи для пересування. Інформація про стан гри (кількість життів, час до наступної хвилі ворогів) візуалізується на екрані у вигляді текстових елементів та індикаторів (рис 3.16).





Рис. 3.16. Інтерфейс гри

Після кінця відліку хвилі ворогів, з'являється надпис про наступ першої хвилі (рис 3.17).



Рис. 3.17. Анонсування першої хвилі ворогів

Активація ворожої хвилі ініціює бойовий сценарій, в якому противники здійснюють пряму атаку на позиції гравця. Для успішного проходження рівня гравець повинен використовувати вогнепальну зброю для знищення всіх ворогів до того, як вони завдадуть смертельного удару (рис 3.18).

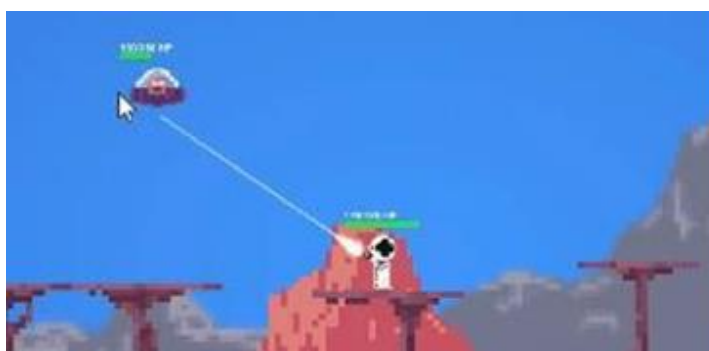


Рис. 3.18. Постріли по ворогу

При отриманні смертельного пошкодження від противника кількість очок здоров'я гравця дорівнюється нулю, що ініціює процес відродження. Про початок відродження свідчить таймер, що відображається на екрані (рис 3.19).

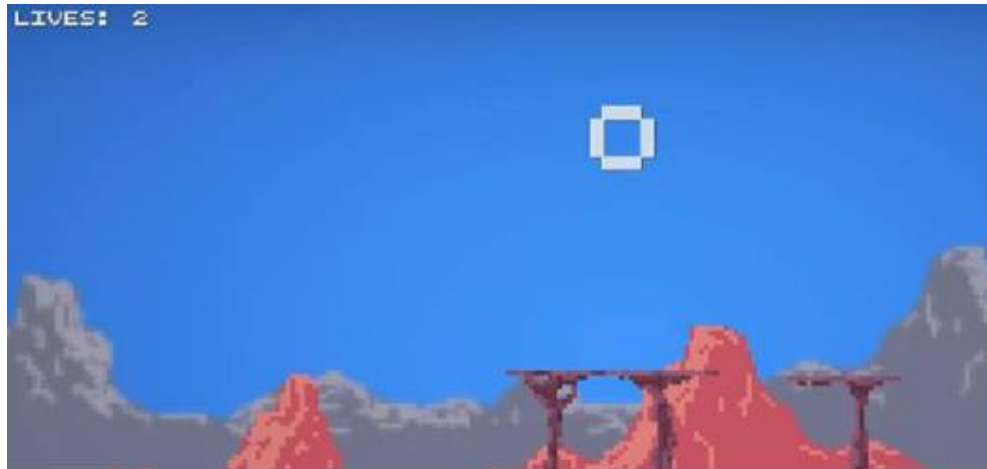


Рис. 3.19. Гибель героя та відлік до відродження

У разі ліквідації всіх ворогів поточної хвилі, ігровий двигун ініціює запуск наступного етапу, про що інформує гравця відповідне текстове повідомлення (рис 3.20).



Рис. 3.20. Інтерфейс з анонсом другої хвилі ворогів

Оскільки гра належить до жанру платформерів, то крім бойових дій, гравець повинен демонструвати високий рівень координації рухів, здійснюючи стрибки між розташованими на різній висоті платформами. Нездатність здійснити стрибок призводить до падіння персонажа та втрати одного життя (рис 3.21).



Рис. 3.21. Знімок екрана з платформами

### 3.3.3. Умови та меню завершення гри

Умовами перемоги є успішне відбиття всіх хвиль ворожих атак. У разі загибелі персонажа певне число разів гра вважається програною. Завершення гри супроводжується відображенням на екрані меню, яке надає гравцеві можливість розпочати гру спочатку або вийти в головне меню (рис 3.22).



Рис. 3.22. Меню завершення гри

### Висновки до розділу

Розділ присвячений практичній реалізації ігрового проєкту. Було детально описано процес створення візуальної частини гри, включаючи вибір графічних елементів та їхню інтеграцію в ігровий рушій. Особливу увагу приділено розробці анімаційних систем для персонажів та ефектів, що забезпечують плавність та реалістичність ігрового процесу.

Реалізовано складну ігрову механіку, що включає в себе систему бойових дій, штучний інтелект ворогів та систему прогресування. Було розроблено

інтуїтивний інтерфейс користувача, який дозволяє гравцеві легко орієнтуватися в ігровому світі та відстежувати свої досягнення.

Завершальним етапом стала розробка системи меню, яка забезпечує зручний доступ до різних функцій гри, таких як початок нової гри, продовження збереженої гри та вихід. Розглянуто процес створення інструкції з користування розробленим прототипом гри. Зокрема, було надано вичерпний опис інтерфейсу головного меню, а також основних дій, доступних гравцеві, таких як біг, стрибки та стрільба.

Особливу увагу було приділено опису стартового екрана та його графічних елементів, включаючи індикатор кількості життів, таймер до початку першої хвили, анімацію оголошення про початок хвили та появу нових ворогів.

## ВИСНОВКИ

У магістерській роботі було проведено аналіз та розробку комп'ютерної гри в жанрі 2D платформера на базі ігрового рушія Unity.

В результаті виконаної роботи було створено функціональний продукт, який відповідає поставленим завданням та демонструє потенціал подальшого розвитку.

Основні досягнення проєкту:

1. Створено мінімальну життєздатну модель гри, яка включає в себе основні ігрові механіки та елементи інтерфейсу.
2. Було проведено детальний аналіз існуючих ігор жанру платформер, що дозволило визначити сильні та слабкі сторони попередніх розробок та сформулювати унікальні особливості розроблюваного проєкту.
3. Для реалізації проєкту було обрано відповідні програмні засоби та інструменти розробки, що забезпечили ефективність та якість кінцевого продукту.
4. Була створена докладна технічна документація, яка описує всі етапи розробки, архітектуру проєкту та алгоритми реалізації окремих функціональних блоків.
5. Інтерфейс користувача розроблений таким чином, щоб мінімізувати час, необхідний для освоєння гри, та забезпечити максимальне занурення в ігровий процес.

Розроблений прототип демонструє високий рівень якості та відповідає сучасним стандартам розробки ігор. Вдало реалізовані основні ігрові механіки, такі як рух персонажа, стрибки, взаємодія з елементами оточення та система бою. Інтерфейс користувача інтуїтивно зрозумілий і не вимагає додаткового пояснення. Проте, для подальшого розвитку проєкту необхідно врахувати наступні аспекти:

- Розширення геймплею. Додавання нових типів ворогів, босів, зброї та предметів дозволить урізноманітнити ігровий процес та збільшити час проходження гри.
- Оптимізація продуктивності. Здійснення оптимізації коду та графіки дозволить підвищити стабільність роботи гри та розширити сумісність з різними пристроями.
- Тестування та налагодження. Проведення більш детального тестування допоможе виявити та усунути помилки в роботі гри.

Проєкт розробки прототипу гри в жанрі платформер є успішним. Отримані результати підтверджують доцільність обраного напрямку дослідження та розробки. Розроблений прототип може слугувати основою для створення повноцінної комерційної гри.

Для подальшого розвитку проєкту рекомендується провести маркетингове дослідження з метою визначення цільової аудиторії та розробки маркетингової стратегії.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гра Cuphead [Електронний ресурс] – Режим доступу до ресурсу: <https://www.xbox.com/ru-RU/games/cuphead>.
2. Гра Dead Cells [Електронний ресурс] – Режим доступу до ресурсу: [https://deadcells.gamepedia.com/Dead\\_Cells\\_Wiki](https://deadcells.gamepedia.com/Dead_Cells_Wiki).
3. Гра Mark of the Ninja [Електронний ресурс] – Режим доступу до ресурсу: <https://www.klei.com/games/mark-ninja>.
4. Гра Starbound [Електронний ресурс] – Режим доступу до ресурсу: <https://playstarbound.com/>.
5. Жанр Платформер [Електронний ресурс] – Режим доступу до ресурсу: [en.wikipedia.org/wiki/Platform\\_game](https://en.wikipedia.org/wiki/Platform_game).
6. Класифікація ігор за жанрами [Електронний ресурс] – Режим доступу до ресурсу: <https://gamesisart.ru/>.
7. Комп'ютерна гра [Електронний ресурс] – Режим доступу до ресурсу: [https://en.wikipedia.org/wiki/PC\\_game](https://en.wikipedia.org/wiki/PC_game).
8. Найкращі ігрові рушії [Електронний ресурс] – Режим доступу до ресурсу: <https://www.gamedesigning.org/career/video-game-engines>.
9. Пояснення про спрайти, тайли [Електронний ресурс] – Режим доступу до ресурсу: <https://www.econdude.pw/2017/08/что-takoe-graficheskij-sprajt- sprite.html>.
10. Створення 2D на Unity [Електронний ресурс] – Режим доступу до ресурсу: [https://skillbox.ru/media/code/kak-sozdat-prostuyu\\_2d\\_igru\\_na\\_unity/](https://skillbox.ru/media/code/kak-sozdat-prostuyu_2d_igru_na_unity/).
11. 2D на Unity, докладний посібник [Електронний ресурс] – Режим доступу до ресурсу: <http://websketches.ru/blog/2d-igra-na-unity-podrobnoye-rukovodstvo-p1>.
12. Abel Gomes. Geometric Computing. Lecture 4 – Interpolation methods [Електронний ресурс] / Abel Gomes. – 2010. – Режим доступу: <http://www.di.ubi.pt/~agomes/tcg/lectures/04-lecture.pdf>.
13. About CMake [Електронний ресурс]. Режим доступу:

<https://cmake.org/overview.>

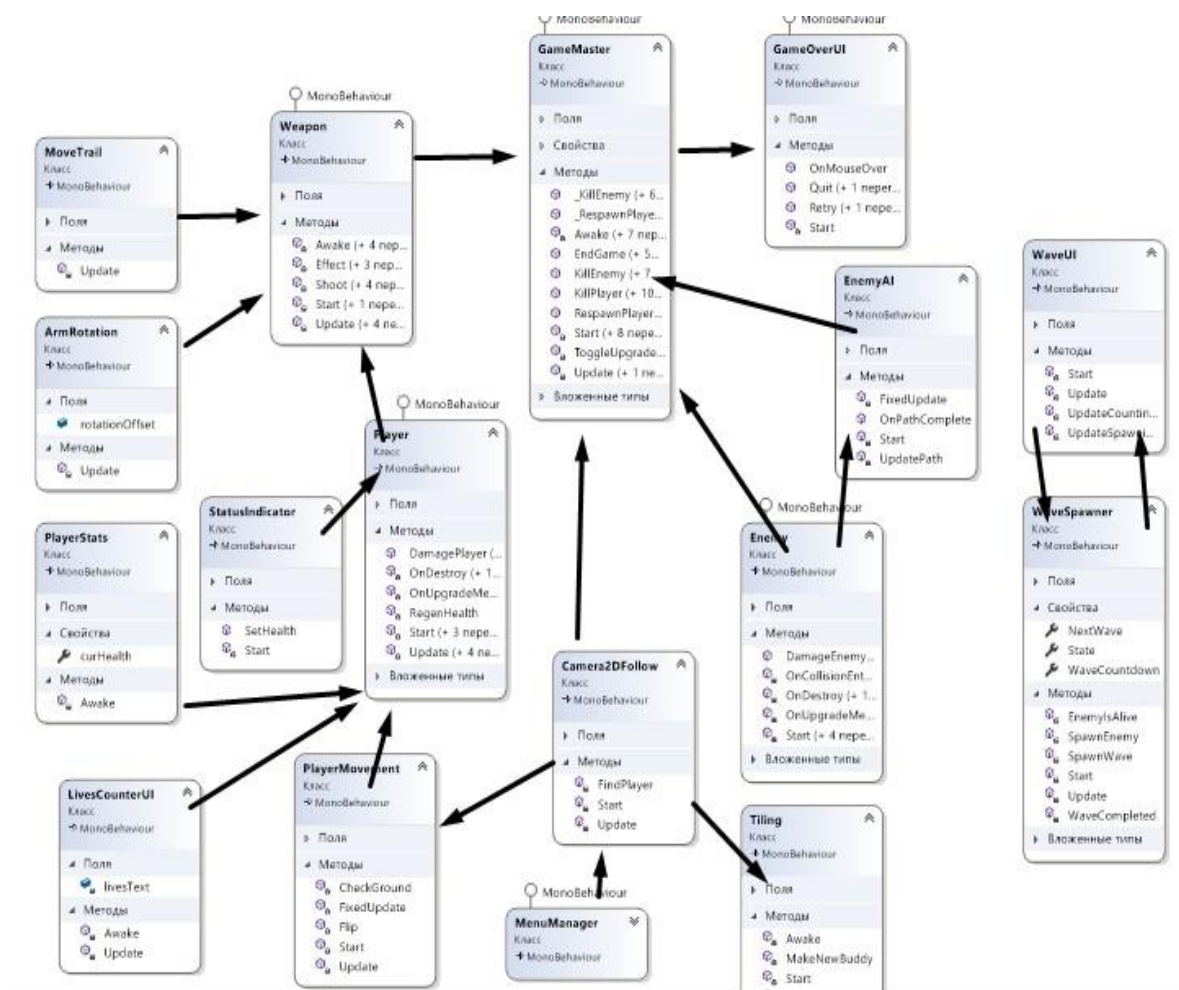
14. Andrew M. Understanding Open Source and Free Software Licensing / Andrew M. St. Laurent. – O'Reilly Media, 2004. – 207 с.
15. Aung Sithu Kyaw. Unity 4.x Game AI Programming / Aung Sithu Kyaw, Clifford Peters, Thet Naing Swe. – Packt Publishing, 2013. – 232 с.
16. Bevilacqua Fernando. Understanding Steering Behaviors [Электронный ресурс] / Fernando Bevilacqua. – 2013. – Режим доступа: <https://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-movement-manager--gamedev-4278>.
17. Bourg David M. Physics for Game Developers, 2nd Edition / David M Bourg, Bryan Bywalec. – O'Reilly Media, 2013. – 578 с.
18. Corless Robert. A Graduate Introduction to Numerical Methods – From the Viewpoint of Backward Error Analysis / Robert Corless, Nicolas Fillion. – Springer Science & Business Media, 2013. – 869 с.
19. Cry Engine [Электронный ресурс] – Режим доступа до ресурсу: <https://www.cryengine.com/>.
20. David B. Hands-On Game Development Patterns with Unity / Baron David., 2019. – 116 с.
21. Despain. 100 Principles of Game Design / Despain, Wendy. – New Riders, 2012. – 240 с.
22. DevGam сравнение игровых движков URL: <http://devgam.com/sravnenie-igrovyyx-dvizhkov-kakoj-vybrat>.
23. 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 с.
24. Ferrone H. Mastering Unity Scripting / Harrison Ferrone., 2016. – 379 с.
25. Halpern J. Developing 2D Games with Unity: Independent Game Program- ming with C# / Jared Halpern., 2018. – 408 с.
26. Hocking J. Unity in Action. Multiplatform game development in C# with Unity 5 / Joseph Hocking., 2015. – 352 с.

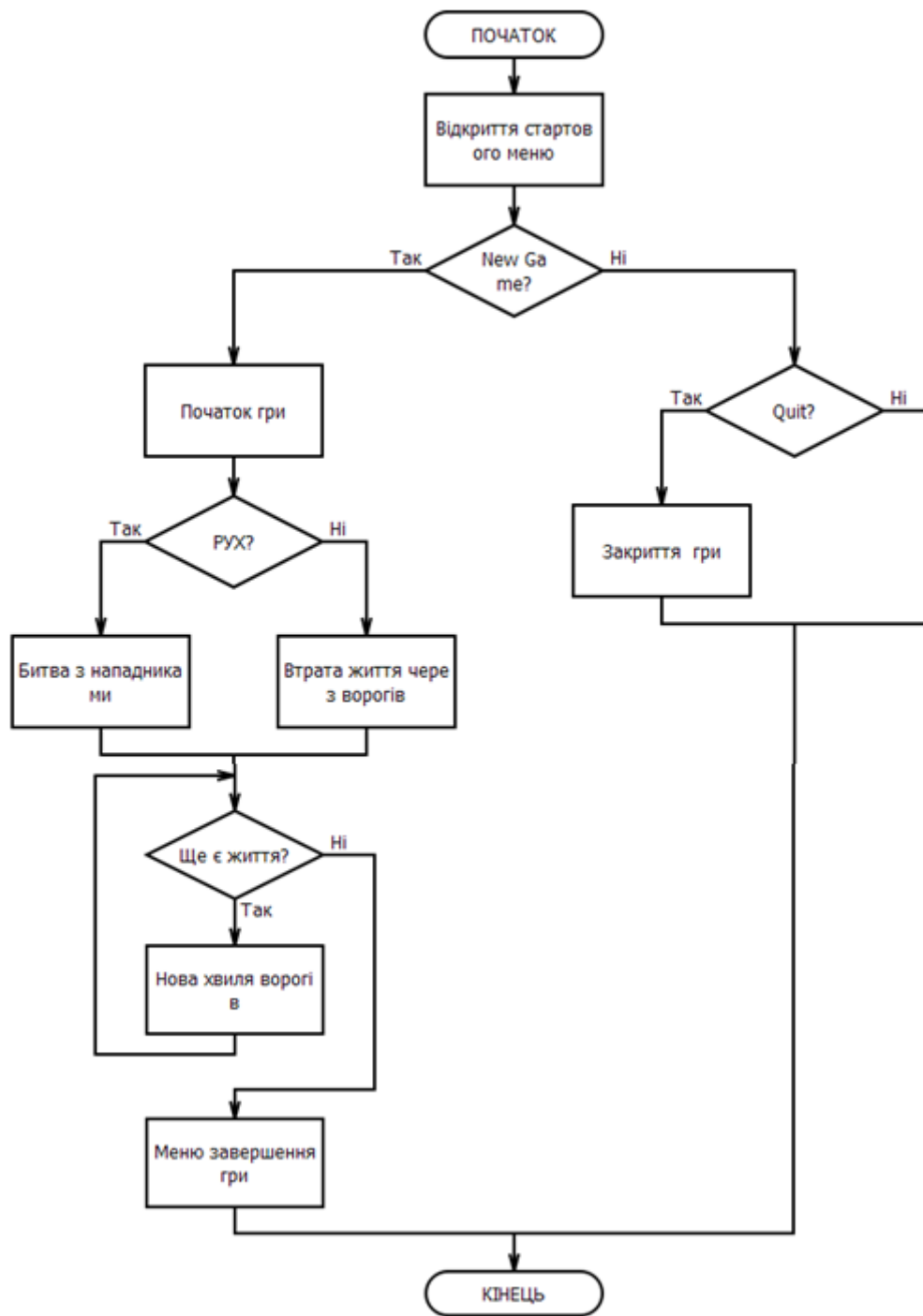


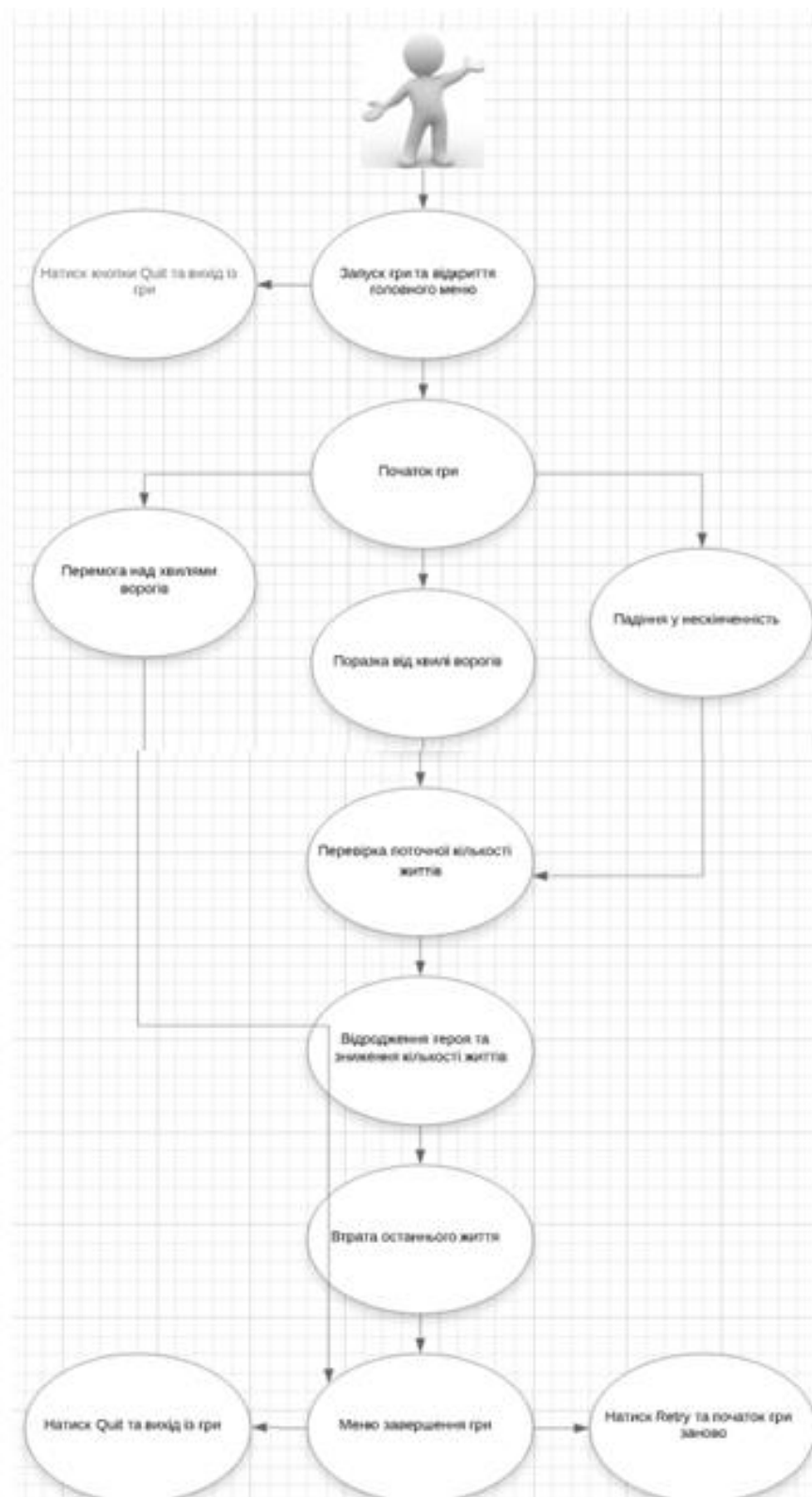
27. Smith G. Basic Math for Game Development with Unity 3D / G. Smith, K. Sung., 2018. – 279 с.
28. Thorn A. Mastering Unity Scripting / Alan Thorn., 2015. – 380 с.
29. Unity Documentation [Электронный ресурс] – Режим доступа до ресурсу: <https://docs.unity3d.com/Manual/index.html>.
30. Unity Game Engine [Электронный ресурс] – Режим доступа до ресурсу: <https://unity.com/ru>.
31. Unreal Engine [Электронный ресурс] – Режим доступа до ресурсу: <https://www.unrealengine.com/>.

# ДОДАТОК А

## Графічні матеріали







## ДОДАТОК Б

### Лістинг коду

#### Файл **PlayerMovement.cs**

```
using System.Collections.Generic; using UnityEditor.Build;
using UnityEngine;
public class PlayerMovement : MonoBehaviour
{
    Rigidbody2D rb; public float speed;
    public float jumpHeight;
    public Transform groundCheck; bool isGrounded;
    // Start is called before the first frame update void Start()
    {
        rb = GetComponent<Rigidbody2D>();
    }
    // Update is called once per frame void Update()
    {
        Flip(); CheckGround();
    }
    void FixedUpdate()
    {
        rb.velocity = new Vector2(Input.GetAxis("Horizontal") * speed, rb.velocity.y); if
        (Input.GetKeyDown(KeyCode.Space) && isGrounded)
        rb.AddForce(transform.up * jumpHeight, ForceMode2D.Impulse);
    }
    void Flip()
    {
        if (Input.GetAxis("Horizontal") > 0) transform.localRotation = Quaternion.Euler(0, 0, 0);
        if (Input.GetAxis("Horizontal") < 0)
        transform.localRotation = Quaternion.Euler(0, 180, 0);
    }
    void CheckGround()
    {
        Collider2D[] colliders = Physics2D.OverlapCircleAll(groundCheck.position, 0.2f); isGrounded =
        colliders.Length > 1;
    }
}
using System; using UnityEngine;
```

#### Файл **Camera2DFollow.cs**

```
namespace UnityStandardAssets._2D
{
    public class Camera2DFollow : MonoBehaviour
    {
        public Transform target; public float damping = 1;
        public float lookAheadFactor = 3;
        public float lookAheadReturnSpeed = 0.5f; public float lookAheadMoveThreshold = 0.1f;
        private float m_OffsetZ;
        private Vector3 m_LastTargetPosition; private Vector3 m_CurrentVelocity; private Vector3
        m_LookAheadPos;
        // Use this for initialization private void Start()
        {
            m_LastTargetPosition = target.position;
            m_OffsetZ = (transform.position - target.position).z; transform.parent = null;
        }
    }
}
```

```

}
// Update is called once per frame private void Update()
{
// only update lookahead pos if accelerating or changed direction float xMoveDelta =
(target.position - m_LastTargetPosition).x;
bool updateLookAheadTarget = Mathf.Abs(xMoveDelta) > lookAhead- MoveThreshold;

if (updateLookAheadTarget)
{
m_LookAheadPos = lookAheadFactor*Vec- tor3.right*Mathf.Sign(xMoveDelta);
}
else
{
m_LookAheadPos = Vector3.MoveTowards(m_LookAheadPos, Vector3.zero,
Time.deltaTime*lookAheadReturnSpeed);
}
Vector3 aheadTargetPos = target.position + m_LookAheadPos + Vector3.for- ward*m_OffsetZ;
Vector3 newPos = Vector3.SmoothDamp(transform.position, aheadTargetPos, ref
m_CurrentVelocity, damping);
transform.position = newPos;
m_LastTargetPosition = target.position;
}
}
}

```

### Файл Parallaxing.cs

```

using UnityEngine;
using System.Collections;
public class Parallaxing : MonoBehaviour {
public Transform[] backgrounds; // Array (list) of all the back- and foregrounds to be parallaxed
private float[] parallaxScales;// The proportion of the camera's movement to move the backgrounds
by
public float smoothing = 1f; // How smooth the parallax is going to be. Make sure to set this above
0
private Transform cam; // reference to the main cameras transform
private Vector3 previousCamPos; // the position of the camera in the previous frame
// Is called before Start(). Great for references. void Awake () {
// set up camera the reference cam = Camera.main.transform;
}
// Use this for initialization void Start () {
// The previous frame had the current frame's camera position previousCamPos = cam.position;
// asigning coresponding parallaxScales parallaxScales = new float[backgrounds.Length];
for (int i = 0; i < backgrounds.Length; i++) { parallaxScales[i] = backgrounds[i].position.z*-1;
}
}
// Update is called once per frame void Update () {
// for each background
for (int i = 0; i < backgrounds.Length; i++) {
// the parallax is the opposite of the camera movement because the previous frame multiplied by the
scale
float parallax = (previousCamPos.x - cam.position.x) *
parallaxScales[i];
parallax

```

```

// set a target x position which is the current position plus the float backgroundTargetPosX =
backgrounds[i].position.x + parallax;
// create a target position which is the background's current position with it's target x position
Vector3 backgroundTargetPos = new Vector3 (backgroundTargetPosX, backgrounds[i].position.y,
backgrounds[i].position.z);

// fade between current position and the target position using lerp backgrounds[i].position =
Vector3.Lerp (backgrounds[i].position,
backgroundTargetPos, smoothing * Time.deltaTime);
}
frame

}
}

// set the previousCamPos to the camera's position at the end of the previousCamPos = cam.position;

```

### Файл Tiling.cs

```

using UnityEngine;
using System.Collections; [RequireComponent (typeof(SpriteRenderer))] public class Tiling :
MonoBehaviour {
public int offsetX = 2;          // the offset so that we don't get any weird errors

// these are used for checking if we need to instantiate stuff public bool hasARightBuddy = false;
public bool hasALeftBuddy = false;
public bool reverseScale = false;    // used if the object is not tilable

private float spriteWidth = 0f;      // the width of our element private Camera cam;
private Transform myTransform;

void Awake () {
cam = Camera.main; myTransform = transform;
}

// Use this for initialization void Start () {
SpriteRenderer sRenderer = GetComponent<SpriteRenderer>(); spriteWidth =
sRenderer.sprite.bounds.size.x;
}

// Update is called once per frame void Update () {
// does it still need buddies? If not do nothing
if (hasALeftBuddy == false || hasARightBuddy == false) {
// calculate the cameras extend (half the width) of what the camera can see in world coordinates
float camHorizontalExtend = cam.orthographicSize * Screen.width/Screen.height;
sprite (element)
// calculate the x position where the camera can see the edge of the
float edgeVisiblePositionRight = (myTransform.position.x +
spriteWidth/2) - camHorizontalExtend;
float edgeVisiblePositionLeft = (myTransform.position.x - spriteWidth/2) + camHorizontalExtend;
// checking if we can see the edge of the element and then calling MakeNewBuddy if we can
if (cam.transform.position.x >= edgeVisiblePositionRight - offsetX
&& hasARightBuddy == false)

```

```

{
MakeNewBuddy (1); hasARightBuddy = true;
}
else if (cam.transform.position.x <= edgeVisiblePositionLeft + off- setX && hasALeftBuddy ==
false)
{
MakeNewBuddy (-1); hasALeftBuddy = true;
}
}
}
}
// a function that creates a buddy on the side required void MakeNewBuddy (int rightOrLeft) {
// calculating the new position for our new buddy
Vector3 newPosition = new Vector3(myTransform.position.x + myTrans- form.localScale.x *
spriteWidth * rightOrLeft, myTransform.position.y, myTrans- form.position.z);
// instantating our new body and storing him in a variable
Transform newBuddy = Instantiate (myTransform, newPosition, myTrans- form.rotation) as
Transform;
// if not tilable let's reverse the x size og our object to get rid of ugly seams if (reverseScale == true)
{
newBuddy.localScale = new Vector3 (newBuddy.localScale.x*-1, newBuddy.localScale.y,
newBuddy.localScale.z);
}

newBuddy.parent = myTransform; if (rightOrLeft > 0) {
newBuddy.GetComponent<Tiling>().hasALeftBuddy = true;

}
else {

}
}
}

newBuddy.GetComponent<Tiling>().hasARightBuddy = true;

```

### **Файл Movetrail.cs**

```

using UnityEngine;
using System.Collections;
public class MoveTrail : MonoBehaviour { public int moveSpeed = 230;
// Update is called once per frame void Update () {
transform.Translate (Vector3.right * Time.deltaTime * moveSpeed); Destroy (gameObject, 1);
}
}
using UnityEngine;
using System.Collections;

```

### **Файл ArmRotation.cs**

```

public class ArmRotation : MonoBehaviour { public int rotationOffset = 90;
// Update is called once per frame void Update () {
// subtracting the position of the player from the mouse position
Vector3 difference = Camera.main.ScreenToWorldPoint (Input.mousePo- sition) -
transform.position;

```



```
difference.Normalize ();      // normalizing the vector. Meaning that all the sum of the vector will
be equal to 1
```

```
float rotZ = Mathf.Atan2 (difference.y, difference.x) * Mathf.Rad2Deg; // find the angle in degrees
transform.rotation = Quaternion.Euler (0f, 0f, rotZ + rotationOffset);
}
}
using UnityEngine;
using System.Collections;
```

### **Файл GameMaster.cs**

```
using UnityStandardAssets.ImageEffects;
public class GameMaster : MonoBehaviour
{
    public static GameMaster gm;
    [SerializeField]
    private int maxLives = 3;
    private static int _remainingLives; public static int RemainingLives
    {
        get { return _remainingLives; }
    }
    [SerializeField]
    private int startingMoney; public static int Money;
    void Awake()
    {
        if (gm == null)
        {
            gm = GameObject.FindGameObjectWithTag("GM").GetComponent<GameMaster>();
        }
    }
    public Transform playerPrefab; public Transform spawnPoint; public float spawnDelay = 2; public
    Transform spawnPrefab;
    public string respawnCountdownSoundName = "RespawnCountdown"; public string
    spawnSoundName = "Spawn";
    public string gameOverSoundName = "GameOver"; public CameraShake cameraShake;
    [SerializeField]
    private GameObject gameOverUI; [SerializeField]
    private GameObject upgradeMenu;
    [SerializeField]
    private WaveSpawner waveSpawner;
    public delegate void UpgradeMenuCallback(bool active); public UpgradeMenuCallback
    onToggleUpgradeMenu;
    //cache
    private AudioManager audioManager;
    void Start()
    {
        if (cameraShake == null)
        {
            Debug.LogError("No camera shake referenced in GameMaster");
        }
        _remainingLives = maxLives; Money = startingMoney;
    }
    //caching
```

```

audioManager = AudioManager.instance; if (audioManager == null)
{
scene.");
}
}
Debug.LogError("FREAK OUT! No AudioManager found in the
void Update()
{
if (Input.GetKeyDown(KeyCode.U))
{
ToggleUpgradeMenu();
}
}
private void ToggleUpgradeMenu()
{
upgradeMenu.SetActive(!upgradeMenu.activeSelf); waveSpawner.enabled =
!upgradeMenu.activeSelf;
onToggleUpgradeMenu.Invoke(upgradeMenu.activeSelf);
}

public void EndGame()
{
audioManager.PlaySound(gameOverSoundName);

Debug.Log("GAME OVER");
gameOverUI.SetActive(true);
}

public IEnumerator _RespawnPlayer()
{
audioManager.PlaySound(respawnCountdownSoundName); yield return new
WaitForSeconds(spawnDelay);

audioManager.PlaySound(spawnSoundName); Instantiate(playerPrefab, spawnPoint.position,
spawnPoint.rotation); GameObject clone = Instantiate(spawnPrefab, spawnPoint.position,
spawnPoint.rotation) as GameObject;
Destroy(clone, 3f);
}

public static void KillPlayer(Player player)
{
Destroy(player.gameObject);
_remainingLives -= 1;
if (_remainingLives <= 0)
{

}
else
{

}

}
}
gm.EndGame();

```

```

gm.StartCoroutine(gm._RespawnPlayer());
public static void KillEnemy(Enemy enemy)
{
gm._KillEnemy(enemy);
}
public void _KillEnemy(Enemy _enemy)
{
// Let's play some sound audioManager.PlaySound(_enemy.deathSoundName);

// Gain some money
Money += _enemy.moneyDrop; audioManager.PlaySound("Money");

// Add particles
GameObject _clone = Instantiate(_enemy.deathParticles, _enemy.trans- form.position,
Quaternion.identity) as GameObject;
Destroy(_clone, 5f);

// Go camerashake
cameraShake.Shake(_enemy.shakeAmt, _enemy.shakeLength); Destroy(_enemy.gameObject);
}

}

using UnityEngine;
using System.Collections;

```

### Файл Enemy.cs

```

[RequireComponent(typeof(EnemyAI))] public class Enemy : MonoBehaviour {

[System.Serializable] public class EnemyStats {
public int maxHealth = 100;

private int _curHealth; public int curHealth
{
get { return _curHealth; }
set { _curHealth = Mathf.Clamp (value, 0, maxHealth); }
}

public int damage = 40;

public void Init()
{
curHealth = maxHealth;
}
}

public EnemyStats stats = new EnemyStats(); public Transform deathParticles;
public float shakeAmt = 0.1f; public float shakeLength = 0.1f;

public string deathSoundName = "Explosion"; public int moneyDrop = 10;
[Header("Optional: ")] [SerializeField]
private StatusIndicator statusIndicator;

```

```

void Start()
{
    stats.Init();

    if (statusIndicator != null)
    {
        statusIndicator.SetHealth(stats.curHealth, stats.maxHealth);
    }

    GameManager.gm.onToggleUpgradeMenu += OnUpgradeMenuToggle;
    if (deathParticles == null)
    {
        Debug.LogError("No death particles referenced on Enemy");
    }
}

void OnUpgradeMenuToggle(bool active)
{
    GetComponent<EnemyAI>().enabled = !active;
}

public void DamageEnemy (int damage) { stats.curHealth -= damage;
if (stats.curHealth <= 0)
{
    GameManager.KillEnemy (this);
}

if (statusIndicator != null)
{
    statusIndicator.SetHealth(stats.curHealth, stats.maxHealth);
}
}

void OnCollisionEnter2D(Collision2D _colInfo)
{
    Player _player = _colInfo.collider.GetComponent<Player>(); if (_player != null)
    {
        _player.DamagePlayer(stats.damage); DamageEnemy(9999999);
    }
}

void OnDestroy ()
{
    GameManager.gm.onToggleUpgradeMenu -= OnUpgradeMenuToggle;
}
}

```

### **Файл Player.cs**

```

using UnityEngine;
using System.Collections;

```

```

[RequireComponent(typeof(Platformer2DUserControl))] public class Player : MonoBehaviour

```

```

{
public int fallBoundary = -20;

public string deathSoundName = "DeathVoice"; public string damageSoundName = "Grunt";

private AudioManager audioManager;
[SerializeField]
private StatusIndicator statusIndicator; private PlayerStats stats;
void Start()
{
stats = PlayerStats.instance; stats.curHealth = stats.maxHealth;
if (statusIndicator == null)
{
}
else
{
}

}

Debug.LogError("No status indicator referenced on Player");
statusIndicator.SetHealth(stats.curHealth, stats.maxHealth);
GameManager.gm.onToggleUpgradeMenu += OnUpgradeMenuToggle;

audioManager = AudioManager.instance; if (audioManager == null)
{
Debug.LogError("PANIC! No audiomanager in scene.");
}

InvokeRepeating("RegenHealth", 1f/stats.healthRegenRate, 1f/stats.healthRe- genRate);
}

void RegenHealth ()
{
stats.curHealth += 1; statusIndicator.SetHealth(stats.curHealth, stats.maxHealth);
}

void Update()
{
if (transform.position.y <= fallBoundary) DamagePlayer(9999999);
}

void OnUpgradeMenuToggle(bool active)
{
GetComponent<Platformer2DUserControl>().enabled = !active; Weapon _weapon =
GetComponentInChildren<Weapon>();
if (_weapon != null)
_weapon.enabled = !active;
}
void OnDestroy()
{
GameManager.gm.onToggleUpgradeMenu -= OnUpgradeMenuToggle;
}

```

```

public void DamagePlayer(int damage)
{
    stats.curHealth -= damage;

    if (stats.curHealth <= 0)
    {
        //play death sound audioManager.PlaySound(deathSoundName);

    }
    else
    {
    }

    //kill player GameManager.KillPlayer(this);

    //play damage sound audioManager.PlaySound(damageSoundName);

    statusIndicator.SetHealth(stats.curHealth, stats.maxHealth);
}

using UnityEngine;

```

#### **Файл PlayeStats.cs**

```

public class PlayerStats : MonoBehaviour
{
    public static PlayerStats instance; public int maxHealth = 100;
    private int _curHealth; public int curHealth
    {
        get { return _curHealth; }
        set { _curHealth = Mathf.Clamp(value, 0, maxHealth); }
    }

    public float healthRegenRate = 2f; public float movementSpeed = 10f;
    void Awake()
    {
        if (instance == null)
        {
            instance = this;
        }
    }
}

```

#### **Файл UpgradeMenu.cs**

```

using UnityEngine;
using UnityEngine.UI;
public class UpgradeMenu : MonoBehaviour { [SerializeField]
    private Text healthText;

    [SerializeField]
    private Text speedText;

```

```

[SerializeField]
private float healthMultiplier = 1.3f;

[SerializeField]
private float movementSpeedMultiplier = 1.3f;

[SerializeField]
private int upgradeCost = 50; private PlayerStats stats;
void OnEnable ()
{
stats = PlayerStats.instance; UpdateValues();
}

void UpdateValues ()
{
healthText.text = "HEALTH: " + stats.maxHealth.ToString(); speedText.text = "SPEED: " +
stats.movementSpeed.ToString();
}

public void UpgradeHealth ()
{
if (GameMaster.Money < upgradeCost)
{
AudioManager.instance.PlaySound("NoMoney"); return;
}

stats.maxHealth = (int)(stats.maxHealth * healthMultiplier);

GameMaster.Money -= upgradeCost; AudioManager.instance.PlaySound("Money");
UpdateValues();
}

public void UpgradeSpeed()
{
if (GameMaster.Money < upgradeCost)
{
AudioManager.instance.PlaySound("NoMoney"); return;
}

stats.movementSpeed = Mathf.Round (stats.movementSpeed * move- mentSpeedMultiplier);

GameMaster.Money -= upgradeCost; AudioManager.instance.PlaySound("Money");

UpdateValues();
}
}

using UnityEngine; using UnityEngine.UI;

```

### **Файл MoneyCounterUI.cs**

```

[RequireComponent(typeof(Text))]

```

```

public class MoneyCounterUI : MonoBehaviour
{
private Text moneyText; void Awake()
{
moneyText = GetComponent<Text>();
}

// Update is called once per frame void Update()
{
moneyText.text = "MONEY: " + GameMaster.Money.ToString();
}
}

```

### Файл AudioManager.cs

```

using UnityEngine;

[System.Serializable] public class Sound
{
public string name; public AudioClip clip;
[Range(0f, 1f)]
public float volume = 0.7f; [Range(0.5f, 1.5f)]
public float pitch = 1f;

[Range(0f, 0.5f)]
public float randomVolume = 0.1f; [Range(0f, 0.5f)]
public float randomPitch = 0.1f; public bool loop = false;
private AudioSource source;

public void SetSource(AudioSource _source)
{
source = _source; source.clip = clip; source.loop = loop;
}

public void Play()
{
source.volume = volume * (1 + Random.Range(-randomVolume / 2f, ran- domVolume / 2f));
source.pitch = pitch * (1 + Random.Range(-randomPitch / 2f, randomPitch

/ 2f));
}

source.Play();

public void Stop()
{
source.Stop();
}
}

```



```

}

public class AudioManager : MonoBehaviour
{
    public static AudioManager instance; [SerializeField]
    Sound[] sounds;

    void Awake()
    {
        if (instance != null)
        {
            if (instance != this)
            {
                Destroy(this.gameObject);
            }
        }
        else
        {
            instance = this; DontDestroyOnLoad(this);
        }
    }

    void Start()
    {
        for (int i = 0; i < sounds.Length; i++)
        {
            sounds[i].name);

            GameObject _go = new GameObject("Sound_" + i + "_" +
            _go.transform.SetParent(this.transform);
            sounds[i].SetSource(_go.AddComponent<AudioSource>());

            PlaySound("Music");
        }

        public void PlaySound(string _name)
        {
            for (int i = 0; i < sounds.Length; i++)
            {
                if (sounds[i].name == _name)
                {
                    sounds[i].Play(); return;
                }
            }
        }
    }
}

```

```

// no sound with _name
Debug.LogWarning("AudioManager: Sound not found in list, " + _name);
}

public void StopSound(string _name)
{
for (int i = 0; i < sounds.Length; i++)
{
if (sounds[i].name == _name)
{
sounds[i].Stop(); return;
}
}

// no sound with _name
Debug.LogWarning("AudioManager: Sound not found in list, " + _name);
}
}

```

### **Файл MenuManager.cs**

```

using UnityEngine;
using UnityEngine.SceneManagement;
public class MenuManager : MonoBehaviour { [SerializeField]
string hoverOverSound = "ButtonHover";

[SerializeField]
string pressButtonSound = "ButtonPress"; AudioManager audioManager;
void Start ()
{
audioManager = AudioManager.instance; if (audioManager == null)
{
Debug.LogError("No audiomanager found!");
}
}

public void StartGame ()
{
audioManager.PlaySound(pressButtonSound);

1);
}

SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex +

public void QuitGame()
{
audioManager.PlaySound(pressButtonSound);

```

```

Debug.Log("WE QUIT THE GAME!");
Application.Quit();
}

public void OnMouseOver ()
{
    audioManager.PlaySound(hoverOverSound);
}

}

```

### Файл Weapon.cs

```

using UnityEngine;
using System.Collections;

public class Weapon : MonoBehaviour {
    public float fireRate = 0; public int Damage = 10; public LayerMask whatToHit;

    public Transform BulletTrailPrefab; public Transform HitPrefab;
    public Transform MuzzleFlashPrefab; float timeToSpawnEffect = 0;
    public float effectSpawnRate = 10;

    // Handle camera shaking
    public float camShakeAmt = 0.05f; public float camShakeLength = 0.1f; CameraShake camShake;
    public string weaponShootSound = "DefaultShot"; float timeToFire = 0;
    Transform firePoint;

    // Caching
    AudioManager audioManager;

    // Use this for initialization void Awake () {
    firePoint = transform.FindChild ("FirePoint"); if (firePoint == null) {
    Debug.LogError ("No firePoint? WHAT?!");
    }
    }

    void Start()
    {
        camShake = GameManager.gm.GetComponent<CameraShake>(); if (camShake == null)
        Debug.LogError("No CameraShake script found on GM object.");

        audioManager = AudioManager.instance; if (audioManager == null)
        {

            scene.");
        }
    }

    Debug.LogError("FREAK OUT! No audiomanager found in

```

```

// Update is called once per frame void Update () {
if (fireRate == 0) {
if (Input.GetButtonDown ("Fire1")) { Shoot();
}
}
else {

if (Input.GetButton ("Fire1") && Time.time > timeToFire) { timeToFire = Time.time + 1/fireRate;
Shoot();

}
}
}

void Shoot () {
Vector2 mousePosition = new Vector2 (Camera.main.Screen- ToWorldPoint
(Input.mousePosition).x, Camera.main.ScreenToWorldPoint(In- put.mousePosition).y);
Vector2 firePointPosition = new Vector2 (firePoint.position.x, fire- Point.position.y);
RaycastHit2D hit = Physics2D.Raycast (firePointPosition, mousePosition- firePointPosition, 100,
whatToHit);

Debug.DrawLine (firePointPosition, (mousePosition-firePointPosi- tion)*100, Color.cyan);
if (hit.collider != null) {
Debug.DrawLine (firePointPosition, hit.point, Color.red); Enemy enemy =
hit.collider.GetComponent<Enemy>(); if (enemy != null) {
enemy.DamageEnemy (Damage);
//Debug.Log ("We hit " + hit.collider.name + " and did " +
Damage + " damage.");
}
}

if (Time.time >= timeToSpawnEffect)
{
Vector3 hitPos; Vector3 hitNormal;

if (hit.collider == null) {
hitPos = (mousePosition - firePointPosition) * 30; hitNormal = new Vector3(9999, 9999, 9999);
}
else
{

}

hitPos = hit.point; hitNormal = hit.normal;

Effect(hitPos, hitNormal);
timeToSpawnEffect = Time.time + 1 / effectSpawnRate;
}
}

```

```

void Effect(Vector3 hitPos, Vector3 hitNormal)
{
    Transform trail = Instantiate (BulletTrailPrefab, firePoint.position, firePoint.ro-
    tation) as
    Transform;
    LineRenderer lr = trail.GetComponent<LineRenderer>();

    if (lr != null)
    {
        lr.SetPosition(0, firePoint.position); lr.SetPosition(1, hitPos);
    }

    Destroy(trail.gameObject, 0.04f);

    if (hitNormal != new Vector3(9999, 9999, 9999))
    {
        Transform hitParticle = Instantiate(HitPrefab, hitPos, Quater-
        nion.FromToRotation (Vector3.right,
        hitNormal)) as Transform;
        Destroy(hitParticle.gameObject, 1f);
    }

    Transform clone = Instantiate (MuzzleFlashPrefab, firePoint.position, firePoint.rota-
    tion) as
    Transform;
    clone.parent = firePoint;
    float size = Random.Range (0.6f, 0.9f); clone.localScale = new Vector3 (size, size, size); Destroy
    (clone.gameObject, 0.02f);

    //Shake the camera camShake.Shake(camShakeAmt, camShakeLength);

    //Play shoot sound audioManager.PlaySound(weaponShootSound);
}
}

```

### **Файл GameoverUI.cs**

```

using UnityEngine;
using UnityEngine.SceneManagement;
public class GameOverUI : MonoBehaviour { [SerializeField]
    string mouseHoverSound = "ButtonHover";

    [SerializeField]
    string buttonPressSound = "ButtonPress"; AudioManager audioManager;
    void Start ()
    {
        audioManager = AudioManager.instance; if (audioManager == null)
        {

            scene.");
        }
    }
}

```

```
Debug.LogError("FREAK OUT! No AudioManager found in the
```

```
public void Quit ()
{
    audioManager.PlaySound(buttonPressSound);

    Debug.Log("APPLICATION QUIT!");
    Application.Quit();
}

public void Retry ()
{
    audioManager.PlaySound(buttonPressSound);

    SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
}

public void OnMouseOver ()
{
    audioManager.PlaySound(mouseHoverSound);
}
```

#### **Файл LiveCounterUI.cs**

```
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Text))]
public class LivesCounterUI : MonoBehaviour { private Text livesText;
void Awake () {
    livesText = GetComponent<Text>();
}

// Update is called once per frame void Update () {
    livesText.text = "LIVES: " + GameManager.RemainingLives.ToString();

}
}
```

#### **Файл WaveUI.cs**

```
using UnityEngine;
using UnityEngine.UI;

public class WaveUI : MonoBehaviour {

    [SerializeField] WaveSpawner spawner;
    [SerializeField]
    Animator waveAnimator;
```

```

[SerializeField]
Text waveCountdownText;

[SerializeField]
Text waveCountText;

private WaveSpawner.SpawnState previousState;

// Use this for initialization void Start () {
if (spawner == null)
{
    Debug.LogError("No spawner referenced!"); this.enabled = false;
}
if (waveAnimator == null)
{
    Debug.LogError("No waveAnimator referenced!"); this.enabled = false;
}
if (waveCountdownText == null)
{
    Debug.LogError("No waveCountdownText referenced!"); this.enabled = false;
}
if (waveCountText == null)
{
    Debug.LogError("No waveCountText referenced!"); this.enabled = false;
}
}

// Update is called once per frame void Update () {
switch (spawner.State)
{
    case WaveSpawner.SpawnState.COUNTING: UpdateCountingUI();
    break;
    case WaveSpawner.SpawnState.SPAWNING: UpdateSpawningUI();
    break;
}

previousState = spawner.State;
}

void UpdateCountingUI ()
{
    if (previousState != WaveSpawner.SpawnState.COUNTING)
    {
        waveAnimator.SetBool("WaveIncoming", false); waveAnimator.SetBool("WaveCountdown",
        true);
        //Debug.Log("COUNTING");
    }
    waveCountdownText.text = ((int)spawner.WaveCountdown).ToString();
}

void UpdateSpawningUI()
{
    if (previousState != WaveSpawner.SpawnState.SPAWNING)

```

```

{
waveAnimator.SetBool("WaveCountdown", false); waveAnimator.SetBool("WaveIncoming",
true);

waveCountText.text = spawner.NextWave.ToString();

//Debug.Log("SPAWNING");
}
}
}

```

### Файл StatusIndicator.cs

```

using UnityEngine;
using UnityEngine.UI;
public class StatusIndicator : MonoBehaviour { [SerializeField]
private RectTransform healthBarRect; [SerializeField]
private Text healthText;

void Start()
{
if (healthBarRect == null)
{

referenced!");
}

Debug.LogError("STATUS INDICATOR: No health bar object

if (healthText == null)
{

referenced!");
}
}

Debug.LogError("STATUS INDICATOR: No health text object

public void SetHealth(int _cur, int _max)
{
float _value = (float)_cur / _max;
healthBarRect.localScale = new Vector3(_value, healthBarRect.lo- calScale.y,
healthBarRect.localScale.z);
healthText.text = _cur + "/" + _max + " HP";
}

}

```

### Файл EnemyAI.cs



```

using UnityEngine;
using System.Collections;
using Pathfinding;

[RequireComponent (typeof (Rigidbody2D))] [RequireComponent (typeof (Seeker))] public class
EnemyAI : MonoBehaviour {

// What to chase?
public Transform target;

// How many times each second we will update our path public float updateRate = 2f;

// Caching
private Seeker seeker; private Rigidbody2D rb;

//The calculated path public Path path;

//The AI's speed per second public float speed = 300f; public ForceMode2D fMode;

[HideInInspector]
public bool pathIsEnded = false;

point

// The max distance from the AI to a waypoint for it to continue to the next way- public float
nextWaypointDistance = 3;
// The waypoint we are currently moving towards private int currentWaypoint = 0;

void Start () {
seeker = GetComponent<Seeker>(); rb = GetComponent<Rigidbody2D>();

if (target == null) {
Debug.LogError ("No Player found? PANIC!"); return;
}
// Start a new path to the target position, return the result to the OnPath-
Complete method
seeker.StartPath (transform.position, target.position, OnPathComplete);

StartCoroutine (UpdatePath ());
}

IEnumerator UpdatePath () { if (target == null) {
//TODO: Insert a player search here. return false;
}

// Start a new path to the target position, return the result to the OnPath-
Complete method
seeker.StartPath (transform.position, target.position, OnPathComplete);

yield return new WaitForSeconds ( 1f/updateRate ); StartCoroutine (UpdatePath());
}

```

```

public void OnPathComplete (Path p) {
    Debug.Log ("We got a path. Did it have an error? " + p.error); if (!p.error) {
        path = p; currentWaypoint = 0;
    }
}

void FixedUpdate () {
    if (target == null) {
        //TODO: Insert a player search here. return;
    }
    //TODO: Always look at player? if (path == null)
    return;

    if (currentWaypoint >= path.vectorPath.Count) { if (pathIsEnded)
    return;

    Debug.Log ("End of path reached."); pathIsEnded = true;
    return;
    }
    pathIsEnded = false;

    //Direction to the next waypoint
    Vector3 dir = ( path.vectorPath[currentWaypoint] - transform.position dir *= speed *
    Time.fixedDeltaTime;
    //Move the AI rb.AddForce (dir, fMode);
    float dist = Vector3.Distance (transform.position, path.vectorPath[current- if (dist <
    nextWaypointDistance) {
    currentWaypoint++; return;

    }
    }
    }

```

### Файл Fading.cs

```

using UnityEngine;
using System.Collections;
public class Fading : MonoBehaviour {

    public Texture2D fadeOutTexture; // the texture that will overlay the screen.
    This can be a black image or a loading graphic
    public float fadeSpeed = 0.8f; // the fading speed

    private int drawDepth = -1000; // the texture's order in the draw hier- archy: a low number
    means it renders on top
    private float alpha = 1.0f; // the texture's alpha value between 0

    and 1

    = 1

```

```

private int fadeDir = -1;    // the direction to fade: in = -1 or out

void OnGUI()
{
    // fade out/in the alpha value using a direction, a speed and Time.deltaTime to convert the
    // operation to seconds
    alpha += fadeDir * fadeSpeed * Time.deltaTime;
    // force (clamp) the number to be between 0 and 1 because GUI.color uses Alpha values between 0
    // and 1
    alpha = Mathf.Clamp01(alpha);

    // set color of our GUI (in this case our texture). All color values remain the same & the Alpha is set
    // to the alpha variable
    GUI.color = new Color (GUI.color.r, GUI.color.g, GUI.color.b, alpha); GUI.depth = drawDepth;

    // render on top (drawn last)

    // make the black tex-

    GUI.DrawTexture(new Rect(0, 0, Screen.width, Screen.height), fadeOutTexture);    // draw the
    // texture to fit the entire screen area
}
// sets fadeDir to the direction parameter making the scene fade in if -1 and out if 1
public float BeginFade (int direction)
{
    fadeDir = direction; return (fadeSpeed);
}

// OnLevelWasLoaded is called when a level is loaded. It takes loaded level index (int) as a
// parameter so you can limit the fade in to certain scenes.
void OnLevelWasLoaded()
{
    // alpha = 1; // use this if the alpha is not set to 1 by default BeginFade(-1);    // call the
    // fade in function
}
}

```

### **Файл Wavespawner.cs**

```

using UnityEngine;
using System.Collections;

public class WaveSpawner : MonoBehaviour {
    public enum SpawnState { SPAWNING, WAITING, COUNTING }; [System.Serializable]
    public class Wave
    {
        public string name; public Transform enemy; public int count;
        public float rate;
    }

    public Wave[] waves; private int nextWave = 0; public int NextWave

```

```

{
get { return nextWave + 1; }
}

public Transform[] spawnPoints;

public float timeBetweenWaves = 5f; private float waveCountdown;
public float WaveCountdown
{
get { return waveCountdown; }
}

private float searchCountdown = 1f;

private SpawnState state = SpawnState.COUNTING;
public SpawnState State
{
get { return state; }
}

void Start()
{
if (spawnPoints.Length == 0)
{
Debug.LogError("No spawn points referenced.");
}

waveCountdown = timeBetweenWaves;
}

void Update()
{
if (state == SpawnState.WAITING)
{
if (!EnemyIsAlive())
{

}
else
{

}
}
}

WaveCompleted();

return;

if (waveCountdown <= 0)
{
if (state != SpawnState.SPAWNING)

```

```

{

}
}
else
{

StartCoroutine( SpawnWave ( waves[nextWave] ) );

waveCountdown -= Time.deltaTime;
}
}

void WaveCompleted()
{
Debug.Log("Wave Completed!");

state = SpawnState.COUNTING; waveCountdown = timeBetweenWaves;

if (nextWave + 1 > waves.Length - 1)
{
nextWave = 0;

}
else
{

}
}

Debug.Log("ALL WAVES COMPLETE! Looping...");

nextWave++;

bool EnemyIsAlive()
{
searchCountdown -= Time.deltaTime; if (searchCountdown <= 0f)
{
searchCountdown = 1f;
if (GameObject.FindGameObjectWithTag("Enemy") == null)
{
return false;
}
}
return true;
}

IEnumerator SpawnWave(Wave _wave)
{
Debug.Log("Spawning Wave: " + _wave.name); state = SpawnState.SPAWNING;

```

```

for (int i = 0; i < _wave.count; i++)
{
    SpawnEnemy(_wave.enemy);
    yield return new WaitForSeconds( 1f/_wave.rate );
}

state = SpawnState.WAITING;

yield break;
}

void SpawnEnemy(Transform _enemy)
{
    Debug.Log("Spawning Enemy: " + _enemy.name);

    Transform _sp = spawnPoints[ Random.Range (0, spawnPoints.Length) ]; Instantiate(_enemy,
    _sp.position, _sp.rotation);
}

}

```