Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *Корнієнко Данила Андрійовича*
(ПІБ)

академічної групи *121-18-2*
(шифр)

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

освітньої програми *Інженерія програмного забезпечення*
(назва освітньої програми)

на тему: *Розробка програмного забезпечення для комп'ютерної гри жанру roque-like з використанням фреймворка Unity та C#*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікаційної роботи	*проф. Бердник М.Г.*			
розділів:				
спеціальний	*проф. Бердник М.Г.*			
економічний	*доц. Касьяненко Л.В.*			
Рецензент	*доц.Шедловський І.А..*			
Нормоконтролер	*доц. Гуліна І.Г.*			

Дніпро
2022

Міністерство освіти і науки України
НТУ «Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
завідувач кафедри
програмного забезпечення комп'ютерних систем
(повна назва)

_____________ І.М. Удовик
(підпис) (прізвище, ініціали)

« » ____________ 2022 року

ЗАВДАННЯ
на кваліфікаційну роботу
бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *121-18-2* *Корнієко Данила Андрійовича*
(група) (прізвище та ініціали)

тема кваліфікаційної роботи *Розробка програмного забезпечення для комп'ютерної гри жанру roque-like з використанням фреймворка Unity та C#*

затверджена наказом ректора НТУ «ДП» від 18.05.2022р. № 268-с

Розділ	Зміст виконання	Термін виконання
Спеціальний	***На основі матеріалів проектно-технологічної та інших науково-технічних джерел провести аналіз стану рішення проблеми та постановку задачі. Обґрунтувати вибір та здійснити реалізацію методів вирішення проблеми***	***13.05.2022 р.***
Економічний	***Провести розрахунок трудомісткості розробки програмного забезпечення, витрат на створення ПЗ й тривалості його розробки***	***27.05.2022 р.***

Завдання видав _____________ *проф. Бердник М.Г.*
(підпис) (посада, прізвище, ініціали)

Завдання прийняв до виконання _____________ *Корнієко Д. А.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *14.01.2022 р.*

Термін подання кваліфікаційної роботи до ЕК: *13.06.2022 р.*

РЕФЕРАТ

Пояснювальна записка: 81с., 21 рис., 3 дод., 25 джерел.

Об'єкт розробки: мобільний додаток комп'ютерної гри у жанрі Rogue-Like.

Мета кваліфікаційної роботи: Розробити програмне забезпечення з інструментом для розробки ігор Unity, яке дозволить пограти у гру жанру Rogue-Like де завгодно, з мобільного пристрою.

У вступі розглядається сучасний ігровий світ, знаходяться переваги ігрового додатку по відношенню до іншого програмного забезпечення. Роз'яснюються основні особливості ігрових додатків. Наведено обґрунтування актуальності та уточнюється постанова задачі.

У першому розділі проведено аналіз предметною області, визначено актуальність завдання та призначення розробки, розроблена постаново завдання, задані вимоги до програмної реалізації, технологій та програмних засобів.

У другому розділі виконано обґрунтування використання інструменту для розробки ігор Unity, виконано проектування архітектури програмного забезпечення та механіки ігрового процесу. Розроблена програма. Наведені характеристики складу параметрів технічних засобів, описаний виклик та завантаження роботи, описана робота програми.

В економічному розділі розраховано трудомісткість розробленої програми розробленого програмного забезпечення, та підрахована вартість розробки створення застосунку, та розраховано час розробки додатку.

Практичне значення полягає у створенні мобільного додатку комп'ютерної гри у жанрі Rogue-Like, для того щоб підвищити настрій людини та створити унікальний опит.

Актуальність даної системи визначається збільшенням ринку мобільних додатків та великим попитом на ігри даного жанру.

Список ключових слів: Unity, C#, SRP, RogueLike, гра.

ABSTRACT

Explanatory note: 81p., 21 fig., 3 appendices, 25 sources.

Object of development: a mobile application of a computer game in the Rogue-Like genre.

The purpose of the qualification work: To develop software with a tool for developing games Unity, which will allow you to play the game genre Rogue-Like anywhere, from a mobile device.

The modern game world is considered in the spotlight, there are advantages of the game application in relation to other software. The main features of game applications are explained. The substantiation of urgency is given and the solution of the problem is specified.

In the first section the analysis of the subject area is carried out, the urgency of the task and the purpose of development are determined, the resolution of the task is developed, the requirements to the software implementation, technologies and software are set.

In the second section the substantiation of use of the tool for development of Unity games is executed, design of architecture of the software and mechanics of game process is executed. Developed program. The characteristics of the parameters of technical means are given, the call and loading of work are described, the work of the program is described.

In the economic section, the complexity of the developed software program is calculated, and the cost of application development is calculated, and the application development time is calculated.

The practical value is to create a mobile application of the computer game in the genre of Rogue-Like, in order to lift a person's mood and create a unique experience.

The relevance of this system is determined by the growing market for mobile applications and high demand for games of this genre.

Keyword list: Unity, C #, SRP, RogueLike, game.

ЗМІСТ

РЕФЕРАТ 3

ABSTRACT 4

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ........ 7

ВСТУП........ 8

1.1. Загальні відомості з предметної галузі 9

1.2. Призначення розробки та область застосування 12

1.3. Підстава для розробки 12

1.4. Постановка задачі........ 13

1.5. Вимоги до програми або програмного виробу........ 14

Вимоги містять ряд правил та рекомендацій, які необхідно задовільнити під час виконання даної кваліфікаційної роботи. 14

1.5.1. Вимоги до функціональних характеристик........ 14

1.5.2. Вимоги до інформаційної безпеки 14

1.5.3. Вимоги до складу та параметрів технічних засобів 14

1.5.4. Вимоги до інформаційної та програмної сумісності........ 14

РОЗДІЛ 2 16

ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ 16

2.1. Функціональне призначення програми........ 16

2.2. Опис застосованих математичних методів........ 17

2.3. Опис використовуваної архітектури та шаблонів проектування........ 17

2.4. Опис використаних технологій та мов програмування........ 23

2.5. Опис структури програми та алгоритми її функціонування 27

2.7. Опис розробленого програмного забезпечення 40

2.7.1. Використані технічні засоби 40

2.7.3. Виклик та завантаження програми 40

2.7.4. Опис інтерфейсу користувача 41

РОЗДІЛ 3 44

ЕКОНОМІЧНИЙ РОЗДІЛ 44

3.1 Визначення трудомісткості та вартості розробки програмного продукту 44

3.2. Витрати на створення програмного забезпечення 46

ВИСНОВКИ 48

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 49

ДОДАТОК А. Код програми 51

ДОДАТОК Б.Відгук керівника економічного розділу 83

ДОДАТОК В 84

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ – Програмне забезпечення

ОС – Операційна система

UI – User interface

SDK– Software Development Kit

JDK– Java Development Kit

NDK– Native Development Kit

SRP – Scriptable Render Pipeline

RP – Render Pipeline

ВСТУП

Комп’ютерні та мобільні ігри зараз знаходяться на передовій сучасного світу. Усі нові технології завжди знаходять своє використання індустрією розробки ігор: зменшення портативних пристроїв, збільшення продуктивності персональних комп’ютерів, окуляри доповненої реальності тощо. Але, нові технології - це не єдине, що робить комп’ютерні ігри, такими унікальними. Велика перевага перед іншим програмним забезпеченням - це азарт, бажання перемогти та спектр емоцій, що отримує гравець[12].

Характерними особливостями для комп’ютерних та мобільних ігор є:

– створення неповторного досвіду;
– наявність проблеми, яку треба вирішити;
– завчасно запрограмовані процеси та розгортання подій за задумкою автора;
– відчуття змагання між гравцями;
– наявність основної механіки;
– наявність історії, на основі якої зав’язується сюжет;
– естетичне відчуття, графічний вид тощо.

Ціллю дипломного проекту є створення гри жанру rogue - like, де у гравця є лише одне життя, щоб здобути перемогу. Це створює перед ним декілька серйозних викликів, які він буде намагатися подолати:

– пройти гру за одне життя;
– дізнатися, що буде на наступних рівнях;
– здобути найсильнішу комбінацію здібностей;
– здобути перемогу за найкоротший час.

Результатом виконання даної кваліфікаційної роботи є програмне забезпечення, яке створює усі вищезазначені виклики перед користувачем.

Задача даного дипломного проекту та об'єкт його діяльності безпосередньо пов'язані з напрямом підготовки «Інженерія програмного забезпечення» та відповідає узагальненій тематиці кваліфікаційних робіт і переліку зазначених виробничих функцій, типових задач діяльності, умінню та компетенціям.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Загальні відомості з предметної галузі

Гра – це процес вирішення проблеми, який приносить задоволення. Гра може бути не лише настільною, фізичною (футбол, баскетбол та інші), комп’ютерною, а також будь-яка дія, що створює проблематику і правила. Наприклад, збирати деталь на заводі це праця, а збирати її при цьому на час, це вже гра, у якій можна змагатися самим з собою намагаючись побити попередній рекорд.

Гра – це прекрасне проведення часу за яким людина переходить до стану потоку, у якому людина забуває про час та відчуває радість[3].

Але, гра – це дуже широке розуміння, що не відображає в цілому кінцевий результат, тому, треба звузити визначення.

Тож, електронна гра – це різновид звичайної гри, який передбачає взаємодію з інтерактивною системою за допомогою електронних пристроїв (програмованих чи не програмованих).

Це вже більш конкретніше визначення, ніж попереднє, і тепер ми розуміємо, що для того, щоб пограти в нашу гру, треба взаємодіяти з інтерактивною системою за допомогою електронних пристроїв. Але, що саме це за система – поки не зовсім зрозуміло.

Яким чином гравець буде з нею взаємодіяти та яким саме буде електронний пристрій – саме це і треба визначити. Звузимо визначення ще більше.

Відеогра – це різновид електронної гри, який передбачає вивід на відео пристрій зображення для керування візуальними образами.

Тепер вже зрозуміло системи для якої треба розробити гру:

Гравець буде бачити результат своїх дій на відео-пристрої, що допоможе створити незабутній досвід для людини, занурюючи його у неймовірний світ зображення та образів. Персональний комп’ютер – це пристрій, що допоможе нам взаємодіяти з електронною системою. Тож, визначення тепер набуває

такого вигляду: комп’ютерна гра – це відеогра, що запускається на персональному комп’ютері.

Тепер зрозуміло, гру якого типу мені необхідно створити. Комп’ютерні ігри захопили індустрію розваг, диктуючи свої умови. Але, чим комп’ютерна гра краще за інші?

Існують такі переваги над іншими видами ігор:

– можливість отримати новий досвід з неймовірними можливостями дій. Відчути себе героєм, чи навпаки – злочинцем. Відчути себе в будь-який незвичайній ролі, в якої людина не здатна опинитися в реальному житті. Що робить комп’ютерні ігри не лише унікальними у технологічному, а й у творчому плані;

– комп’ютерні ігри дають більш гнучкий вибір між жанрами ігор, у які людина бажає пограти: хтось вибере спортивний симулятор, а інший, навпаки, захоче відчути себе стратегом. Найрозумніші можуть випробувати свої сили у шахах чи інших інтелектуальних змаганнях;

– можливість пограти у себе вдома в будь-який час. Комп’ютерні ігри, на відміну від консольних, в які грають більш у колі сім'ї, припускає більш затишний відпочинок на одинці з собою, чи у колу Інтернет-друзів, не потребуючи від гравця якихось активних дій;

– соціальний досвід. Не всі комп’ютерні ігри мають мультіплеер, проте, це дозволяє відчути себе у багатомільйонній спільноті прямо зі свого дому;

– дешевизна та легкість здобуття задоволення.

Вірогідніше, що у всіх людей зараз є персональний комп’ютер. Зіграти в комп’ютерні ігри можливо, навіть не завантажуючи їх, – потрібно лише запустити браузер та зіграти з друзями у форматі онлайн. А тим, хто потребує більш унікальних вражень, ринок надає можливість пограти в ігри з більш якіснішим контентом за додаткову плату. Наразі ринок комп’ютерних ігор є чи не найбільшим в індустрії надання розваг, що приносить щороку близько 40 мільярдів доларів. Ці фактори приваблюють нових гравців та розробників.

Комп'ютерні ігри поділяються на жанри. Жанри потрібні, щоб класифікувати ігри по набору дій гравця, та відрізняються використовуваними механіками. Жанр не залежить від графічної обгортки, чи написаного сюжету[13].

Rogue-like – це один з жанрів, названий в честь гри прабатька: Rogue. Механіками які класифікують цей жанр :

– якщо гравець програє (смерть у видуманому світі), то гра перезапускається заново;

– гра повинна бути наповнена випадковістю, тобто кожен сеанс повинен відрізнятися від попереднього;

– чим далі йде гра, тим складнішою вона стає;

– додавання предметів, які вводять нові правила у гру, модифікуючи її.

Щоб створити гру самотужки, потрібно багато знань у програмуванні, гейм-дизайні, моделюванні, малюванні, а також в анімації та комп'ютерній графіці.

Розберемо кожну сферу більш докладно:

– програмування являє собою проектування архітектури проекту, написання коду на мові програмування (в моєму випадку це C#), та тестуванні програмного забезпечення;

– гейм-дизайн – процес створення ідеї, яка буде приносити незабутній, приємний та новий досвід гравцю;

– моделювання – створення 3Д моделей, які інтегруються у гру;

– малювання – створення текстур, інтерфейсу та багатьох інших зображень, що можуть знадобитися при розробці гри;

– анімування – процес створення анімації для 3Д моделей чи зображення;

– комп'ютерна графіка – процес обробки та перенесення на екрани монітору зображення, а також робота з відео-картою.

Інструмент для розробки комп'ютерних ігор набагато спрощує роботу розробника, даючи можливість не писати з нуля оболонку. А також, бере на

себе обробку зображення, дозволяючи розробнику створювати нові шейдери для покращення графіки. В моєму випадку доцільним буде використати платформу Unity.

Unity – це найпопулярніший інструмент для розробки комп’ютерних ігор, що підтримує DirectX та OpenGL.

Завдяки Unity розробники ігор можуть заощадити місяці чи навіть роки розробки. Це готове рішення, що не вимагає створювати зручний редактор рівнів, розробляти з нуля відмолювання комп’ютерної графіки та має ще багато інших переваг для розробника.

1.2. Призначення розробки та область застосування

Цільова аудиторія моєї комп’ютерної гри – чоловіки 15 – 40 років.

В комп’ютерній грі користувачі можуть випробувати свої сили через завищення складності з кожним рівнем та увійти у почуття потоку – максимального зосередження.

Область застосування широка, бо кожна людина, маючи при собі лише смартфон з операційною системою Android, зможе пограти в цю гру.

Місце гри може бути будь-яким:

- дома, сидячи на дивані;
- у потягу чи літаку;
- сидячи на лавці в парку;
- рухаючись по вулиці тощо.

1.3. Підстава для розробки

Підставами для розробки (виконання кваліфікаційної роботи) є:

- освітня програма спеціальності 121 Інженерія програмного забезпечення ;
- навчальний план та графік навчального процесу;
- наказ ректора Національного технічного університету «Дніпровська політехніка» №268-с від 18.05.2022р.

Завдання на кваліфікаційну роботу на тему: «Розробка програмного забезпечення для комп’ютерної гри жанру roque-like з використанням фреймворка Unity та C#".

1.4. Постановка задачі

Задача кваліфікаційної роботи – розробити комп'ютерну гру жанру roque-like. Розробка повинна здійснюватися мовою програмування C#. Програмне забезпечення повинно бути сучасним та дозволяти розробникам розширювати новий функціонал та механіки. Код гри повинен відповідати стандартам чистого коду, мати корисні назви та легко сприйматися.

Основні механіки гри:

1. Рух головного персонажа.
2. Рух інших персонажів.
3. Система зчитування дій персонаж (джойстик).
4. Синхронізація руху з діями користувача.
5. Стрільба.
6. Снаряд:
   6.1 Застосування бонусів до снарядів.
   6.2 Зміна снаряду від поточних предметів.
7. Система здоров'я.
8. Нанесення шкоди.
9. Контроль здоров'я.
10. Ворожий інтелект
    10.1 Пошук гравця
    10.2 Паттерни руху
    10.3 Атака
11. Пошук Ворога
12. Створення випадкових рівнів.
13. Перемога.
14. Програш.
15. Інтерфейс.
    16.1 Інтерфейс перемоги.
    16.2 Інтерфейс програшу.
    16.3 Інтерфейс стартового екрану.

Програмне забезпечення має бути реалізоване як мобільний додаток APK для платформи Android з використанням мови програмування C# та інструменту для розробки комп’ютерних ігор Unity.

1.5. Вимоги до програми або програмного виробу

Вимоги містять ряд правил та рекомендацій, які необхідно задовільнити під час виконання даної кваліфікаційної роботи.

1.5.1. Вимоги до функціональних характеристик

Для забезпечення максимального обсягу допустимих пристроїв, програмне забезпечення повинне підтримувати:

– платформу операційної системи Android;
– мінімальна версія Android повинна бути 4.4.

1.5.2. Вимоги до інформаційної безпеки

Щоб уникнути небажаних ситуацій та некоректної роботи програмного забезпечення, потрібно максимально розділити відповідальність класів. Система повинна вести себе однаково стабільно, інтерфейс користувача повинен бути адаптивним під усі розміри мобільних пристроїв чи планшетів.

1.5.3. Вимоги до складу та параметрів технічних засобів

Для забезпечення максимальної стабільності мобільні пристрої повинні мати такі характеристики:

– кількість ядер: 4 та більше;
– частота процесору: 1.3 ГЦ;
– Оперативна пам’ять: 2 ГБ та більше;
– Оперативна система: Android 4.4 (kitkat) та вище.

1.5.4. Вимоги до інформаційної та програмної сумісності

У якості мови програмування було вибрано мову програмування C# – це основна мова програмування інструменту для розробки ігор на платформі Unity.

Unity дозволяє розробляти програмне забезпечення для більшості популярних операційних систем на різних платформах.

Середовище розробки обрано Rider, який надає найбагатший функціонал при розробці ігор та спрощує життя розробника у багатьох аспектах.

РОЗДІЛ 2
ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

2.1. Функціональне призначення програми

Результатом цієї кваліфікаційних роботи є комп'ютерна гра яку можливо бути запустити на сучасних на смартфонах з ОС Android, що дозволить пограти в неї де завгодно, маючи при собі лише смартфон. Дана гра повинна бути у жанрі Rogue – like, що принесе найяскравіші емоції кінцевому користувачу.

Основний функціонал додатка полягає в кор геймплеї (коло гри, яке гравець повторює раз за разом, щоб дійти до кінця) і повинен бути таким: гравець вбиває монстрів проходячи рівні, по завершенню одного чи декількох рівнів він здобуває посилення, яке трохи змінює тактику гри гравця, та посилює шкоду яку гравець наносить монстрам.

З кожним рівнем складність гри росте разом з силою та здоров'ям монстрів (рис. 2.1).

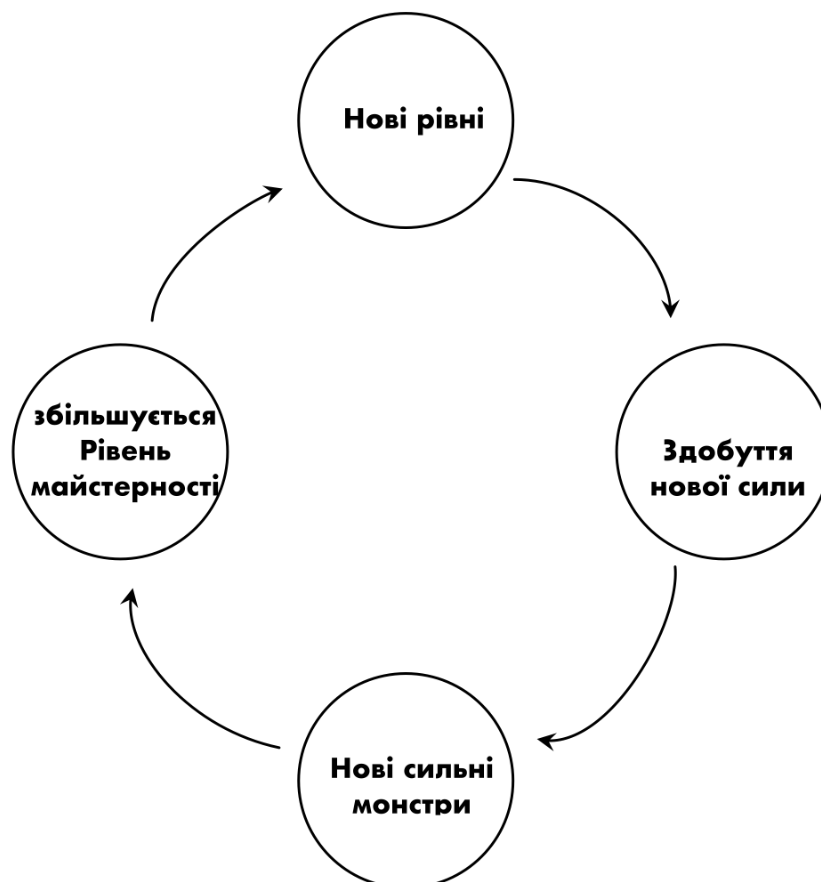

Рис. 2.1. Кор-геймплей

2.2. Опис застосованих математичних методів

Під час проектування та розробки даного програмного забезпечення, використовувалися лише прости арифметичні дії. Математичні методи не використовувалися.

2.3. Опис використовуваної архітектури та шаблонів проектування

Під час проектування ПЗ було вирішено використовувати інструмент для розробки ігор Unity.

Unity був створений у 2005 році «від розробників для розробників», щоб полегшити розробку комп'ютерних ігор та дати безкоштовний інструмент, який облегшить розробку ігор в декілька разів. Цей інструмент облегшує розробку у багатьох областях одразу[6].

Основні переваги інструменту Unity:

– допомагає розробляти фізичні ігри, даючи можливість використовувати вже запроектовану фізичну модель в реальному часу завдяки простому функціоналу класів

– Unity повністю бере на себе рендерінг зображення в реальному часі, використовуючи найсучасніші засоби оптимізації, та дозволяє не витрачаи багато часу на роботу з відео картою[5];

– зберігає гнучкість та весь функціонал для розробника, надаючи доступ до Scriptable Rendering Pipeline, дозволяючи писати свої Render Path, додаючи новий функціонал до обробки зображення, та надає свою мову програмування шейдерів Shader Lab, яка дозволяє розробляти шейдери як на мові шейдерів CG, так і на HLSL6;

– спрощує роботу з UI system, дозволяючи легко адаптувати один UI під усі платформи та розміри.

Unity одразу ж реалізує багато паттернів проектування, що використовуються в ігровій розробці:

– Пристосуванець (Flyweight) – розділення даних об'єкта на два типи. Перший тип - внутрішні (intrinics) дані, які не є унікальними для кожного екземпляру та зовнішні (extrinsic) дані, які є унікальними для кожного

екземпляру. Шаблон виносить усі внутрішні дані виносяться в один екземпляр для всіх об’єктів, що зменшує час рендирінгу однотипних об’єктів (приклад дерева) [15]

– Ігровий цикл (Game Loop). На кожному циклі обробляє клієнтське введення, оновлює стан гри, виповнює всі скріпти та рендерить зображення гри. [15]

– Об’єкт тип (Type object). Кожен екземпляр класу може представляти інший тип.

– Сутність-Компонент (Entity Component). Одна сутність може охоплювати багато областей, не зв’язуючи їх. Для ізоляції областей, код поміщають до класу компонента. Сутність спрощується до контейнера компонентів. [13]

Найважливішою сутністю із себе представляє GameObject, яка може обробляти компоненти та є основною точкою входу для розробника. Кожен GameObject завжди має компонент Transform. Цей компонент відповідає за положення об'єкта на сцені. (рис 2.2)

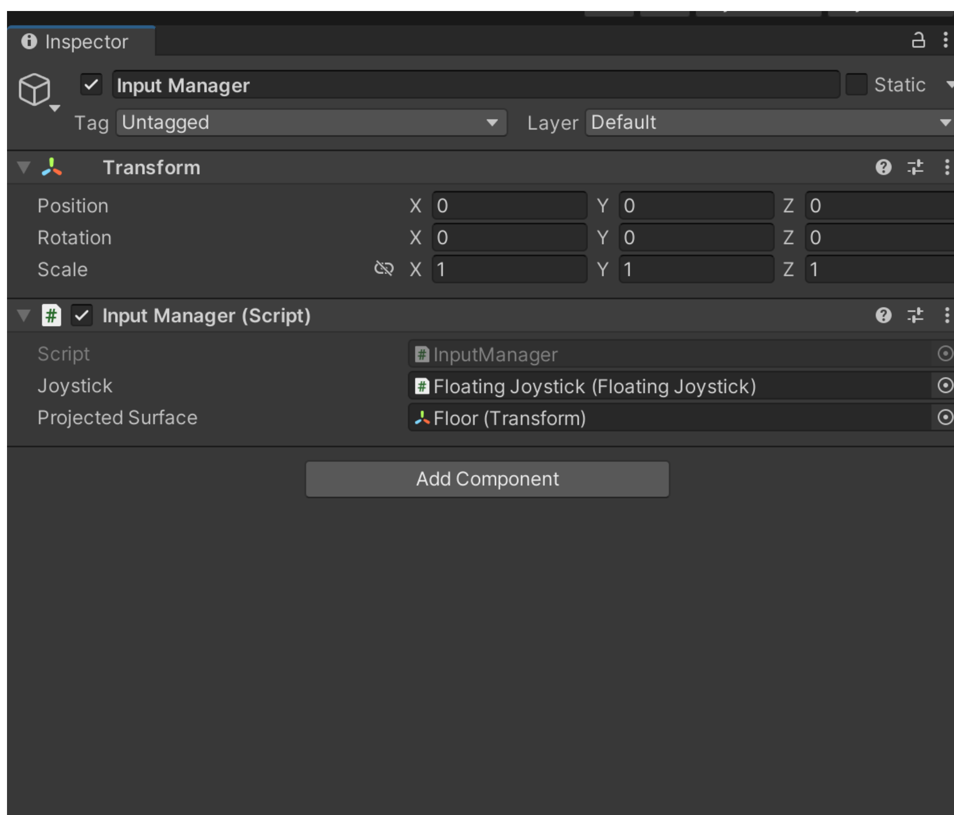

Рис. 2.2. Положення об’єктів на сцені

Інші компоненти, які можливо додати, повинні унаслідуватися від MonoBehavior класу бібліотеки Unity. (рис 2.3)

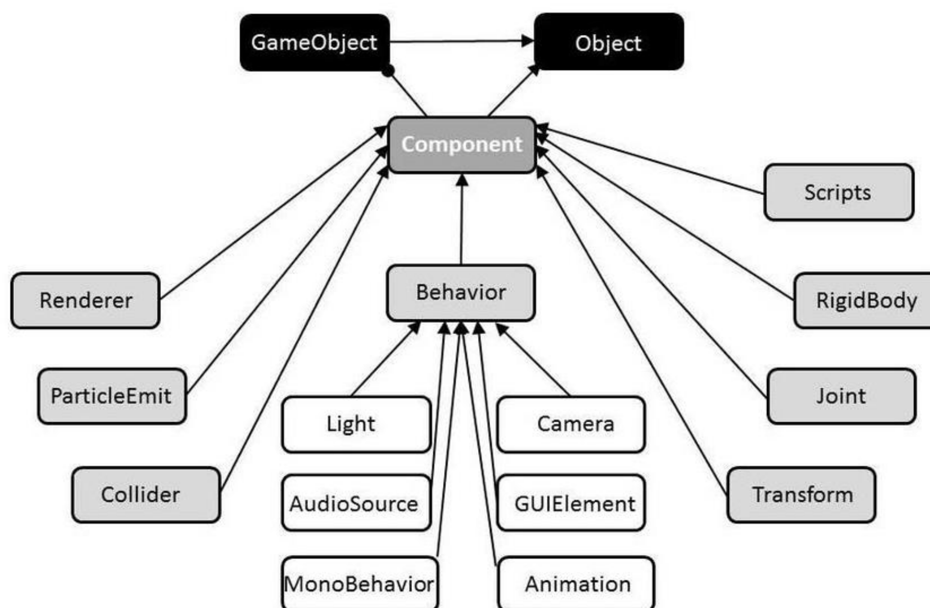

Рис 2.3. Приклад архітектури Unity

MonoBehavior реалізує багато функцій, які розробник може перевизначити, але найважливіші є такими (рис 2.6):

Awake – це перша функція, яка визивається при завантажені сцени, як тільки GameObject визивається перед Start (не викликається, якщо об’єкт не активний, доки його не активують).

Не активний об’єкт – це об’єкт, у якому параметру Active in Hierarchy присвоєно значення False. (рис 2.4) Можливо вмикати та вимикати завдяки редактору та через код. (рис 2.5)

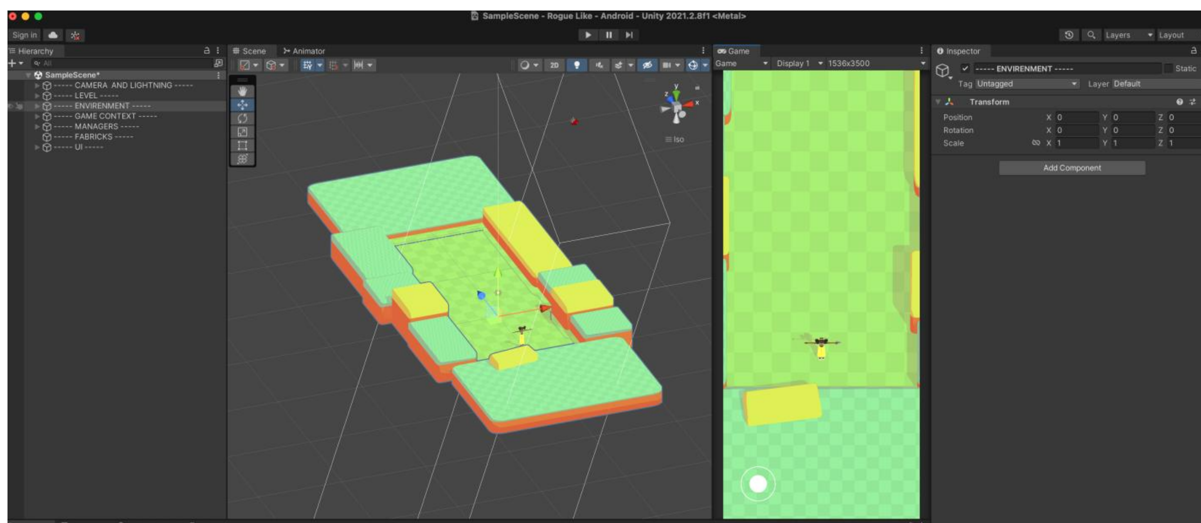

Рис 2.4 .Увімкнений об’єкт

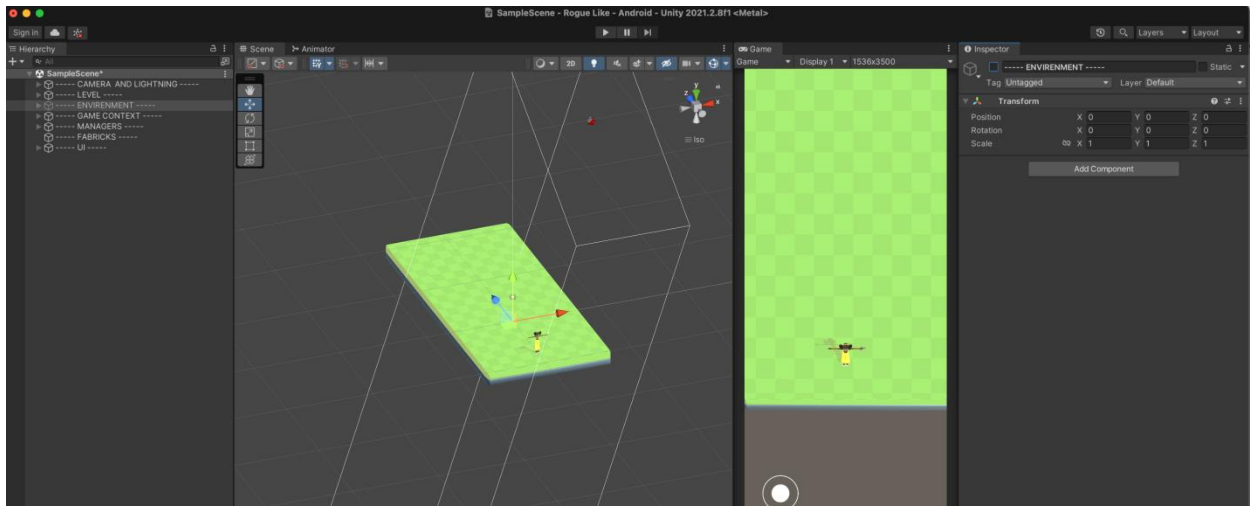

Рис 2.5. Вимкнений об’єкт

Як бачимо, коли об’єкт вимкнений, то його компонент, що відповідає за вимальовування, також перестає працювати.

OnEnable – ця функція визивається кожного разу коли об’єкт вмикається на сцені, а також коли MonoBehavior створює об’єкт свого класу.

Start – викликаєтьмя перед Update.

Update –викликається кожний кадр. В цій функції зазвичай пишеться функціонал, який повинен постійно оновлюватися. Може мати різну частоту визову, в залежності від часу обробки кадру.

FixedUpdate являє собою функцію такого ж типу, як і Update, але викликається не кожного кадру, а маж фіксований проміжок часу між викликом. За замовченням це 0.02 с, що дозволяє викликати функцію 50 разів на секунду. Це може бути більше або менше, ніж викликів функції Update. Зазвичай, використовується для обробки фізики.

LateUpdate виконується так же, як і Update кожного кадру, але особливість в тому, що функція реалізується саме перед створенням кадру.

OnDisable – функція визивається кожного разу, коли об’єкт вимикається на сцені.

OnDestroy – функція викликається для останнього кадру, коли GameObject існує, саме перед тим, як його видалять. Також викликається при зміні сцени. Цю функцію зазвичай використовують задля відписки від івентів.

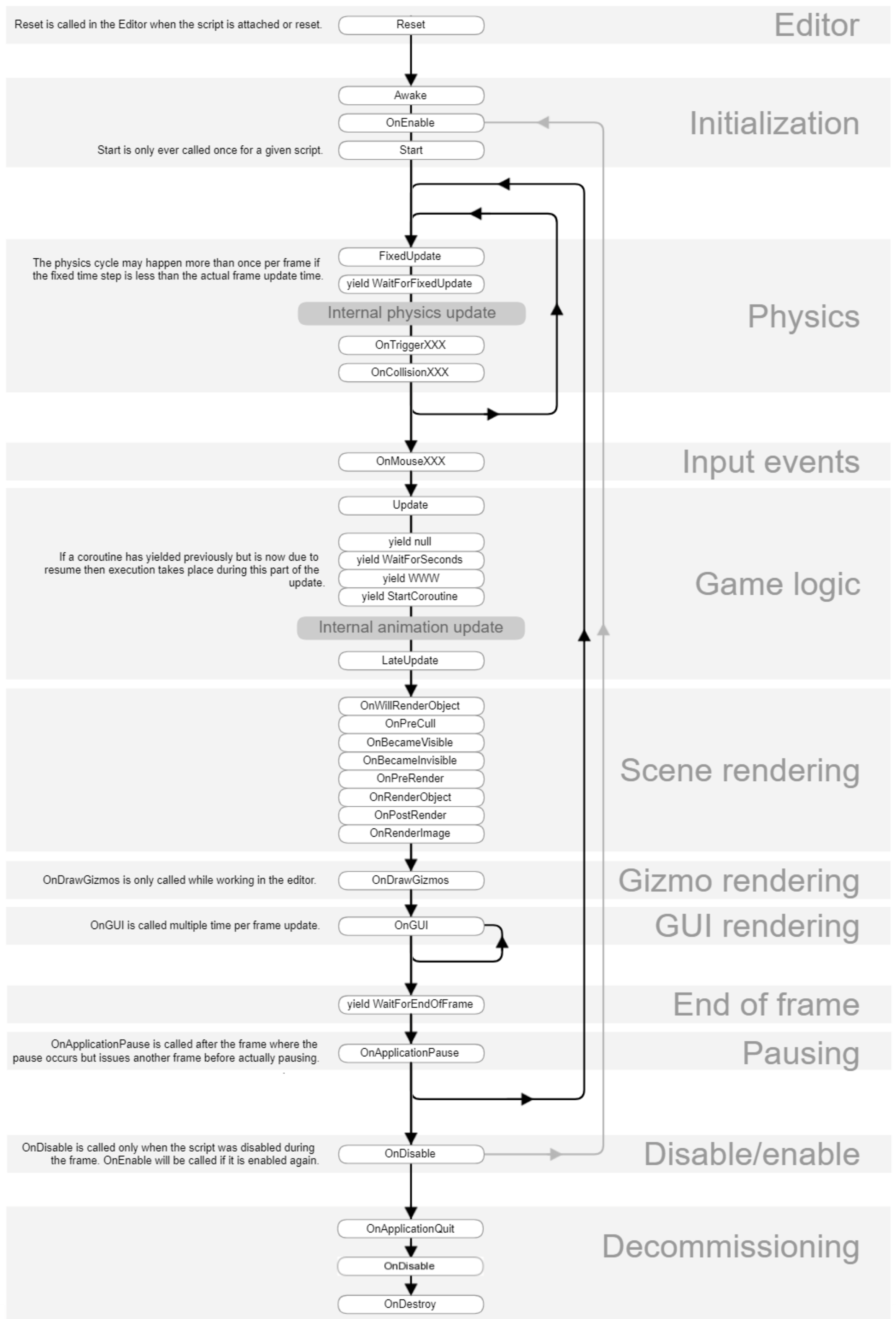

Рис 2.6. Діаграма визву функцій у об’єкта MonoBehaviour

Архітектура коду, вистроєна на Actor та Behaviour, це також Entity – Component паттерн, задля того щоб не створювати один великий, чи багато

маленьких MonoBehabiour класів. В такому випадку MonoBehaviour виступає в якості вхідний точки (рис 2.7).

Кожний актор має набор поведінки та даних. Додання поведінки зроблено за патерном Стратегія.

Стратегія (Strategy) – один з основних поведінкових паттернів проектування, який дозволяє створювати схожі алгоритми у сімейство класів та динамічно підміняти їх під час виконання роботи ПЗ[14].

В даному випадку. поведінка змінюється в залежності від типу поведінки

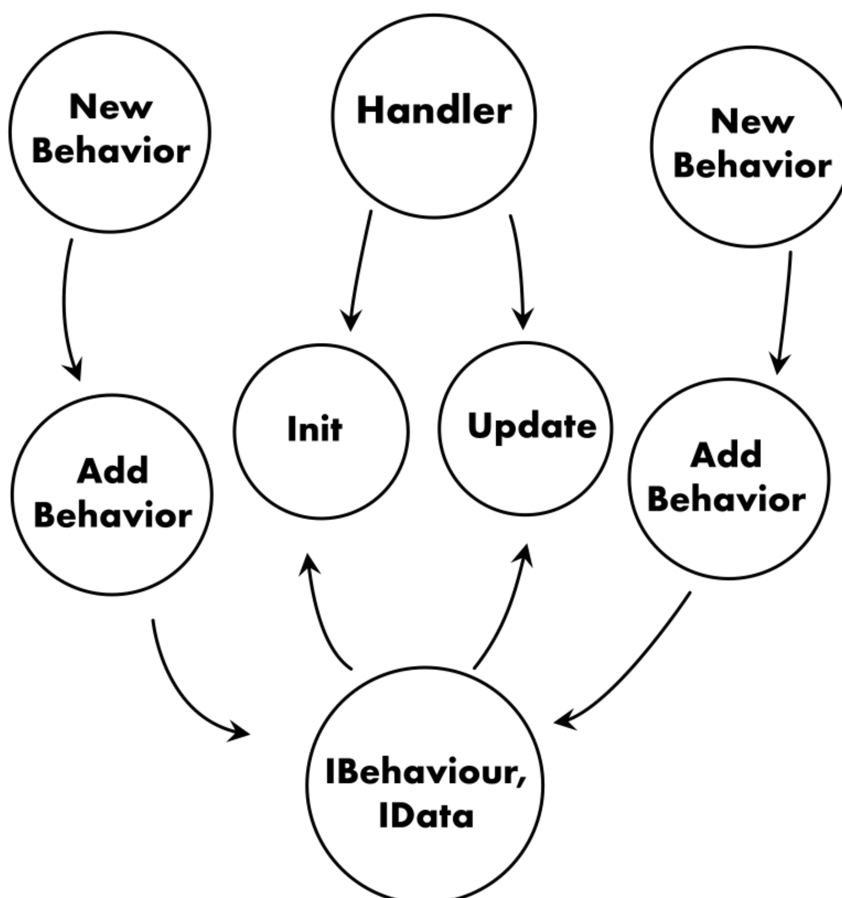

Рис 2.7. Архітектура обробника

Обробника ініціює та оновлює усі свої класи поведінки, а також пропрасує залежності через DIContainer.

DIContainer це частина патерну Dependency Injection.

Впровадження залежностей (Dependency Injection) – шаблон розроблений для проектування ПЗ, що передбачає надання зовнішньої залежності об’єкту, передаючи через інверсію залежностей розв’язані залежності.

DIContainer – це контейнер який зберігає в собі усі залежності. Контейнерів може бути декілька на сцені, глобальний який працює на весь проект, контейнер для сцени, чи для об’єкта, нижчі об’єкту можуть перевизначати залежності.

2.4. Опис використаних технологій та мов програмування

Необхідно більш детально розглянути використані технології та їх властивості. Тож, при виконані даної кваліфікаційної роботи були задіяні такі програмні продукти:

- C#;
- Unity;
- AI;
- Physics;
- Unity UI;
- Cinemachine;
- TextMeshPro;
- Timeline;
- Core RP Library;
- Shader Graph;
- Universal RP;
- DoTween;
- Android Build Support.

C# – багатофункціональна мова програмування високого рівня, розроблена компанією Microsoft. Являє собою C-подібну мову, орієнтовану на об’єктно орієнтовану розробку для платформи .Net. Використовується як основна мова програмування для інструменту розробки ігор Unity.

Unity – Інструмент для розробки ігор.

AI – Пакет (Бібліотека) Unity яка дозволяє працювати з NavMesh системою, яка являє собою структуру даних для пошуку правильного та найкоротшого шляху до обраної точки (рис 2.4).

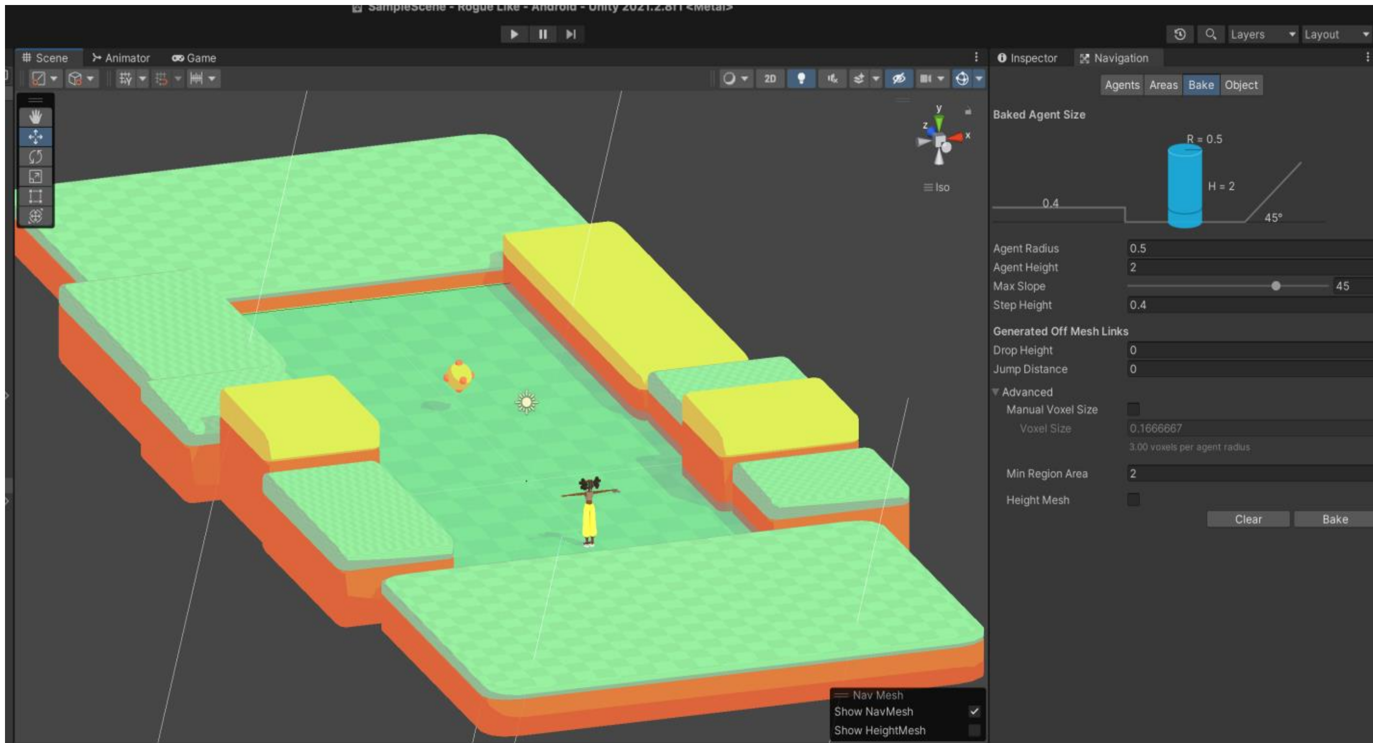

Рис 2.8.Сітка NavMesh

Physics – стандартний пакет Unity, що надає доступ до фізичних компонентів.

Unity UI – пакет Unity, який надає скрипти та компоненти для роботи з UI інтерфейсом. Основним компонентом є Canvas, що оновлюється кожного кадру, коли якийсь з його дітей по ієрархії змінює свій стан, та повністю перерисовує полотно. Завдяки набору для верстки легко створювати UI для екранів різного розміру.

Cinemachine – покращення звичайної вистроєної камери, що додає більший контроль за діями гравця, додаючи нову віртуальну камеру, яка бере повний доступ над стандартною камерою Unity.

TextMeshPro – покращений пакет для праці з текстами в оболонці Unity UI, який не затискає та не блюрить текст по краям. Реалізує текст плоскості з

накинутим шейдером, що покращує якість відображення тексту та надає властивості тексту, що схожі з HTML.

TimeLine – часова шкала для праці з анімаціями, яка дозволяє робити чи редагувати анімації завдяки часовій шкалі. Розробник може задати будь-яке значення будь-якого параметру як точку на часовій шкалі. Декілька точок створюють анімацію.

Core RP Library – багатофункціональна бібліотека для створення Scriptable Render Pipiline (SRP). Основною бібліотекою для взаємодії є Core Rp, яка надає великий функціонал. SRP – функція більшого контролю для розробників, що надає можливість змінювати конфігурації завдяки скриптам, а саме:

1. Оптимізація продуктивності на конкретному пристрої.
2. Можливість налаштовувати RP завдяки тонкому налаштування процесу рендерінгу.
3. Керувння споживанням обчислюваних ресурсів.

ShaderGraph – пакет для візуального програмування шейдерів, що дозволяє створювати складні шейдери, не знаючи мови програмування hlsl. ShaderGraph надає блоки для взаємодії з піксельним та геометричним шейдером, а також математичні блоки для полегшення роботи. Надходить одразу з пакетом Univesal RenderPipeline.

Universal Render Pipeline (URP) являє собою один з готових варіантів RP. Це оптимальний варіант для охоплення максимальної кількості платформ. Це збалансоване рішення, яке надає якісний високопродуктивний та оптимізований графік без необхідності змінювати проект. Окрім цього, він адаптовано масштабується під кожну платформу.

DoTween – одна з найважливіших бібліотек. DoTween надає доступ до легкого створення анімацій з періодом послаблення та діями після завершення анімацій. Можливо інтерполювати у часі будь-що, у тому числі поля та змінні класу, що надає велику кількість можливостей.

Android Build Support. Unity дозволяє збирати на різні платформи, надаючи можливість збирати запускний файл, дозволяючи розробнику писати файли лише на мові програмування C#.

Android Build Support – це бібліотеки Unity, що дозволяє створювати apk чи aab файл (рис 2.10) . Усе що потрібно – це налаштувати android sdk, ndk, jdk and Gradle (рис 2.9). Більш того, Unity дозволяє експортувати проект у Android Studio, де розробник може зробити це сам, вручну.

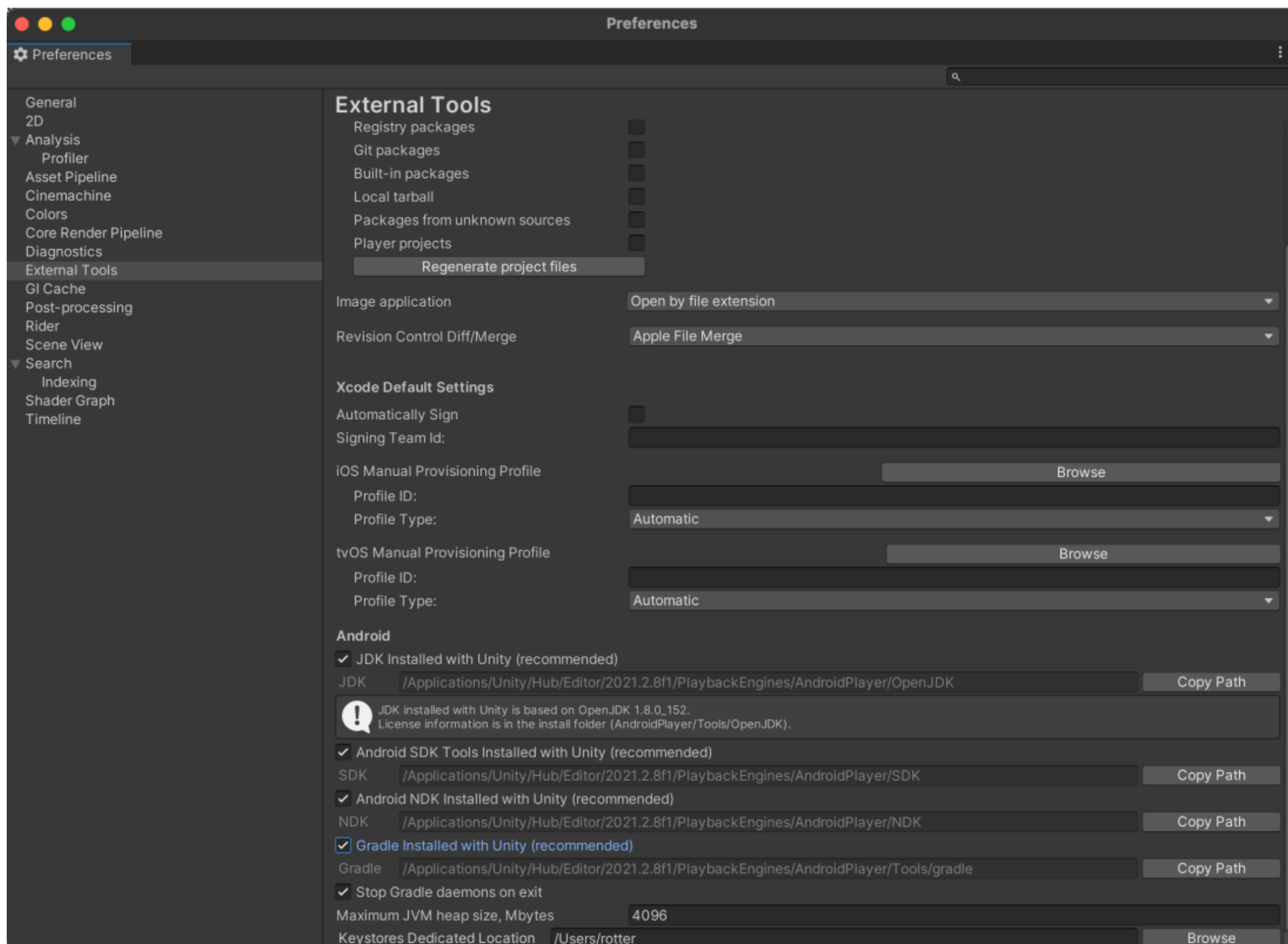

Рис 2.9 – Налаштування sdk для збирання apk

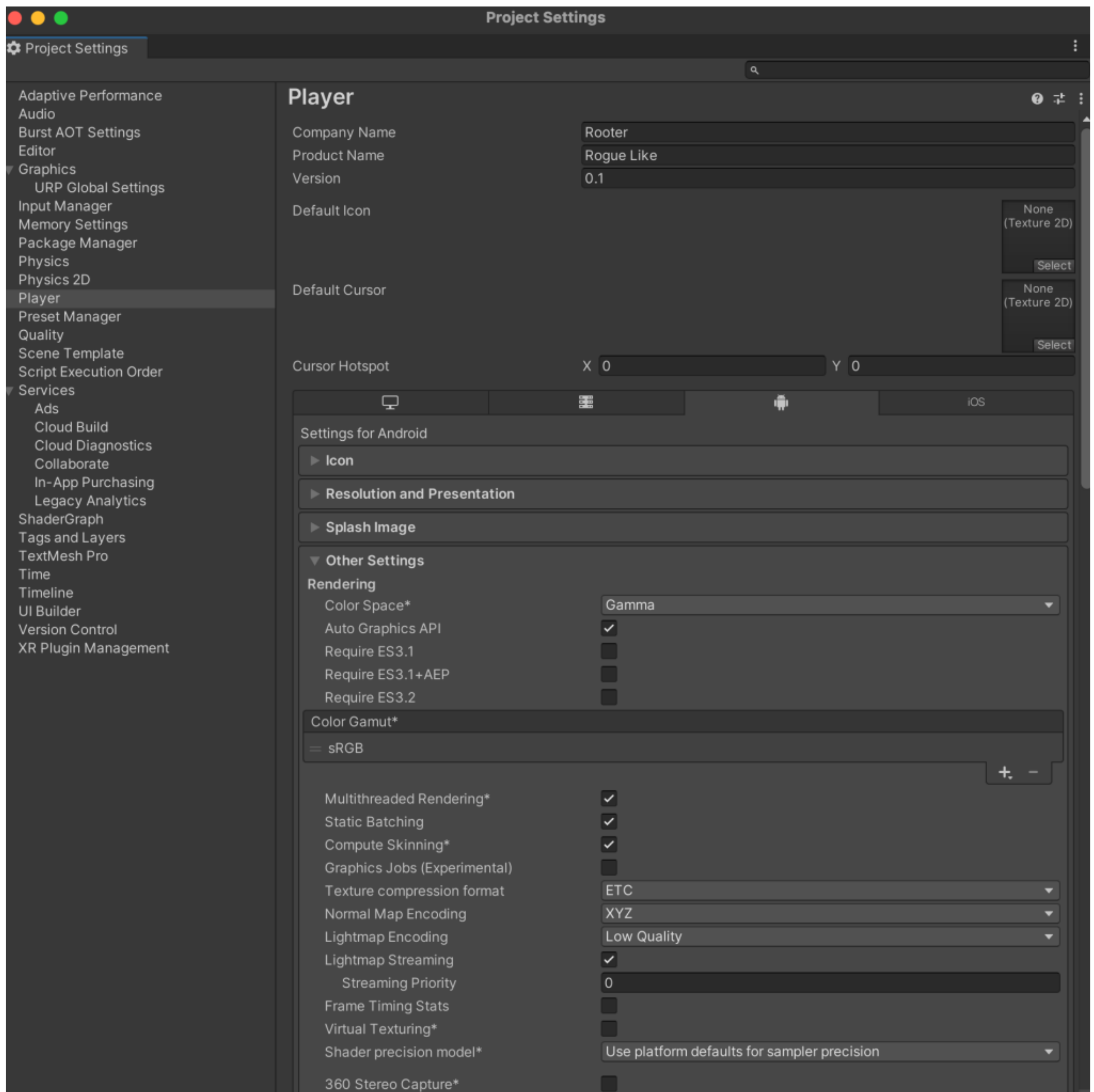

Рис 2.10. Інші налаштування для створення apk файлу

2.5. Опис структури програми та алгоритми її функціонування

Основна ціль гри – вбити усіх ворогів та перейти на наступний рівень з більш сильними монстрами.

Основна сцена гри утворює з себе невелику арену, де є простір щоб порухатися. На сцені є два види сутностей – гравець та ворог.

Розберемо поведінку гравця більш детальніше:

З початку дивимося, чи робить якісь дії користувач, якщо зафіксований якесь введення даних через цифровий джойстик, то вираховуємо вектор руху та додаємо до фізичного тіла прискорення в потрібному векторі. По висоті (вектор Y) залишаємо незмінним. (рис 2.11)

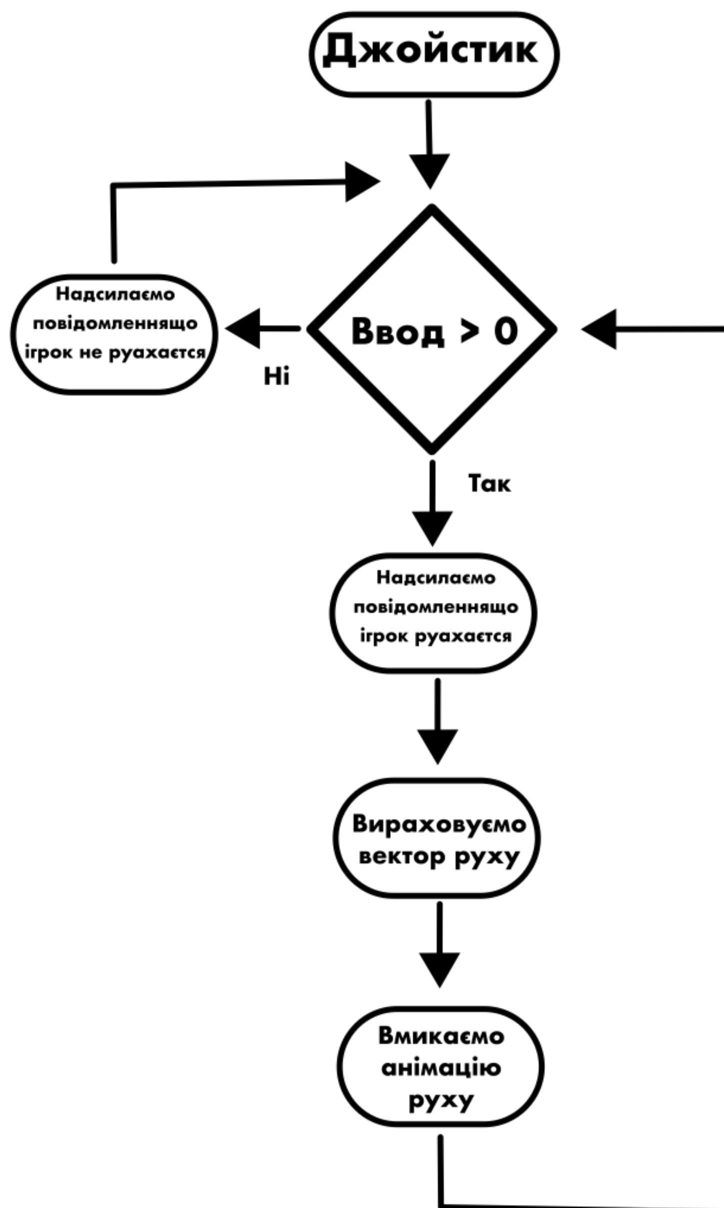

Рис 2.11. Блок-схема руху

Вступні дані надаються вже зміненими на основі положення камери та площини, по котрій рухається персонаж. Це дозволяє гарантувати коректну поведінку головного персонажа у різних положеннях камери, та створює більшу відмово стійкість.

На основі надісланою інформації про рух, різні класи поведінки обробника виконують нові дії.

Стрільба об’єднує дві системи.

Перша система – це пошук цілі. Клас пошуку інжектить (впроваджує залежності) до себе на дані рівня. Дані рівня зберігають таку інформацію :

- стіни;
- перешкоди;
- ворогів;
- гравця.

З цих даних класу пошуку потрібен масив даних юнітів. В залежності від того, ким себе являє обробник, гравець отримує масив даних ворогів, де кожна сутність є юнітом. Сортування масиву даних здійснюється по параметру transform.position, де перший йде найближчий до гравця та бере перший елемент. Далі створюється нове повідомлення з параметром Transform (компонент будь-якого GameObject, який зберегає позицію у координатах) та передається до свого обробника. (рис 2.12)

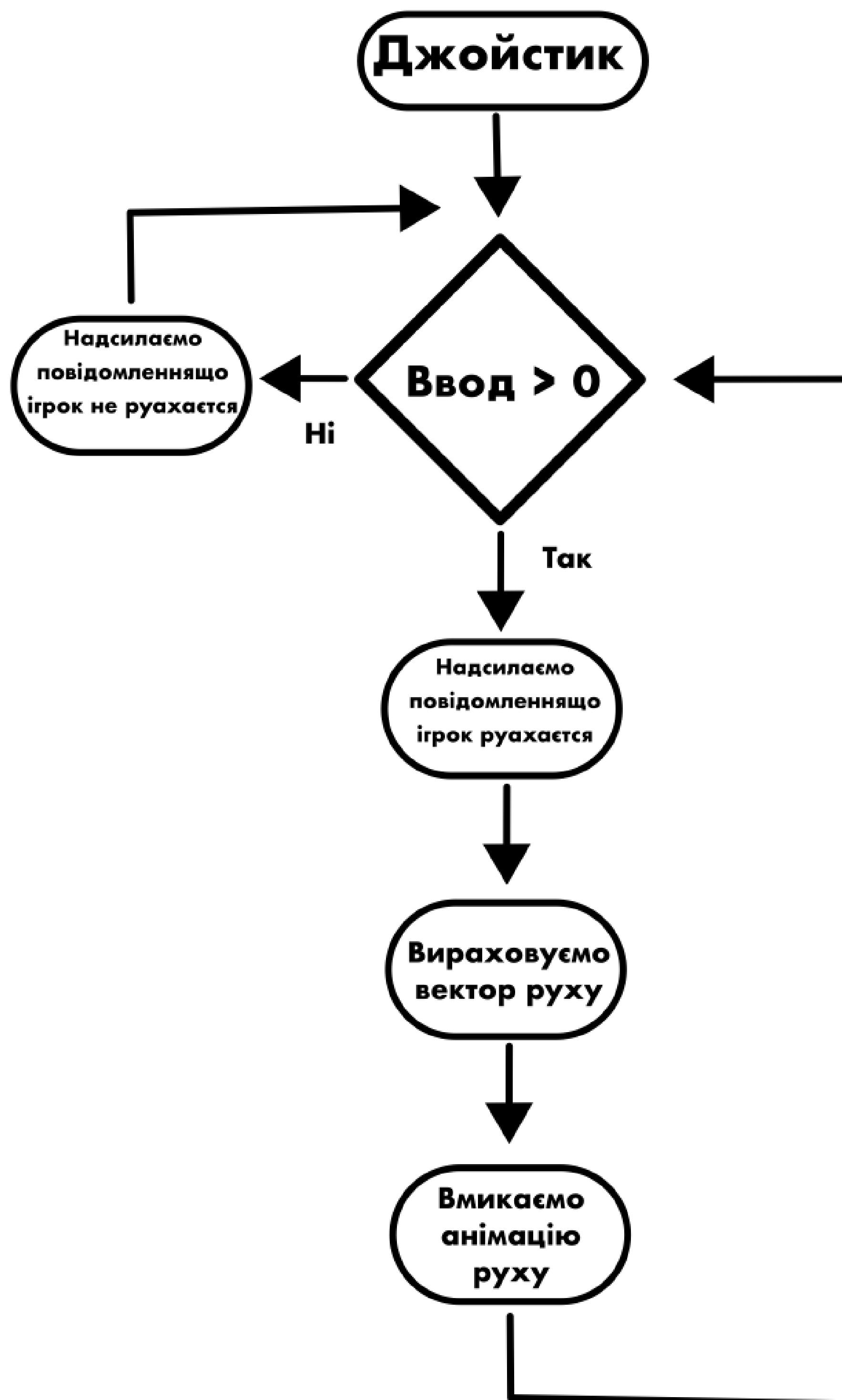

Рис. 2.12. Блох-схема роботи джойстика

Клас, який слідкує за повідомленнями класу пошуку – це клас стрільби.

Клас стрільби має таку поведінку:

Коли клас стрельби отримує повідомлення про те, що ціль найдена, він оновлює свої стартові параметри, повертає GameObject обробника до цілі. Коли GameObject повернувся з невеликою погрішністю, то запускається анімація пострілу, перевіряються модифікатори, створюється пуля як обробник та GameObject на сцені. Вона є сутністю, яку потім буде вимальовувати Unity, та в неї заповнюється потрібна поведінка (рис 2.13):

– рух по прямій свого локального положення (туди, куди дивиться вектор Z);

– реагування на колайдери;

– смерть обробника після торкання ворога, чи після торкання колайдерів визначеної кількості раз;

– відбиття від стін, якщо такий модифікатор є в гравця.

Після створення пулі, надається стартова позиція, а також число шкоди, яку вона наносить, швидкість польоту, та поворот у сторону ворога.

По закінченню створення пулі кодом, також запускаються MonoBehavior скрипт. (рис 2.14)

Рис 2.13. Фрагмент з гри

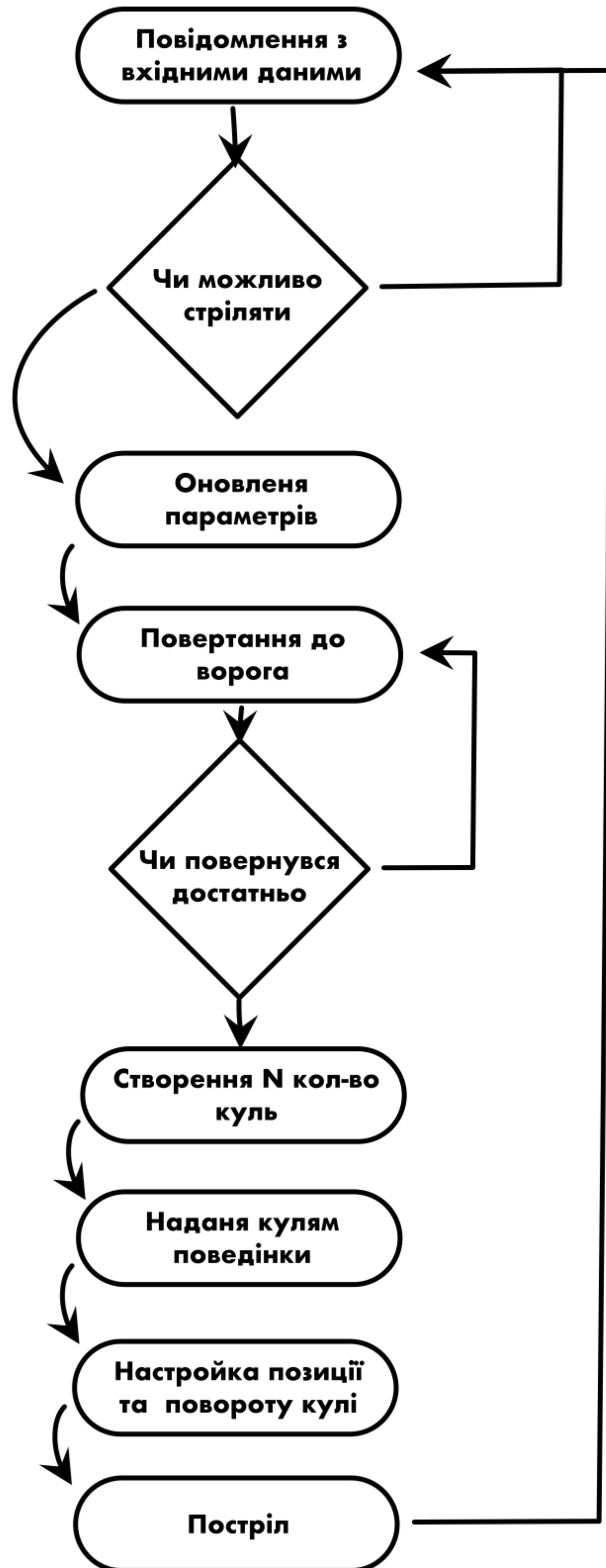

Рис 2.14. Блок схема стрільби

Для коректної роботи потрібно ще два класи, клас обробник доторкання фізичних об'єктів та клас обробник здоров’я.

Клас обробника доторкання існує як компонент GameObject’а. Для його роботи потрібні ще два компоненти: перший – це Rigidbody (тверде тіло). Компонент відповідає за взаємодію фізики. Якщо ввімкнена взаємодія з гравітацією, то на тіло впливає сила, яка тягне його донизу. Rigidbody прораховує взаємодію фізики та фізичних колайдерів (Collider).

Колайдер – це компонент, описуючий форму об'єктів для прорахунку фізичних зіткнень. Це дозволяє додати кожному об’єкту (таким як: куля, персонаж, стіни тощо) фізичну інтерпретацію. Колайдери невидимі, тому їх неможливо побачити. Колайдер може бути будь-якої форми, але найоптимальніше використовувати найпростіші форми, такі як:

– куб;
– сфера;
– капсула;
– циліндр.

Для взаємодії\ та прорахунку зіткнень з ними використовують простіші математичні формули.

Колайдери можуть бути двох типів:

– звичайними – коли відбувається зіткнення, тверде тіло прораховує силу, яка дія на нього та додає до свого прискорення, також створюючи крутний момент, дозволяючи розробникам працювати з фізичними симуляціями;

– тригерними – тип не впливає на тверде тіло та його прорахування фізики, але також регіструє зіткнення.

Rigidbody прораховує, чи зіткнеться його колайдер на наступному кадрі, та кожного кадру. Коли відбувається зіткнення двох колайдерів, MonoBehavior викликає функції в середині себе, які приймають в якості параметра колайдер, що дозволяє через код оброблювати зіткнення, які потрібні розробнику.

Методи, які викликаються при зіткненні колайдерів:

– OnCollisionEnter – викликається, коли колайдер Rigidbody взаємодіє з нормальним колайдером.

– OnTrigerEnter – працює по принципу OnCollisionEnter, але для тригерного колайдера.

– OnCollisionStay – викликається, коли в колайдері Rigidbody знаходитися інший колайдер.

– OnTrigerStay – працює по принципу OnCollisionStay, але для тригерного колайдера.

– OnCollisionExit – викликається, коли колайдер покидає рамки форми колайдеру Rigidbody.

– OnTrigerExit – працює по принципу OnCollisionExit, але для тригерного колайдера.

Клас для оброки зіткнень являється гібридним, тому унаслідується від MonoBehavior. На OnColisionEnter створюється повідомлення, яке відсилається до обробника. (рис 2.15)

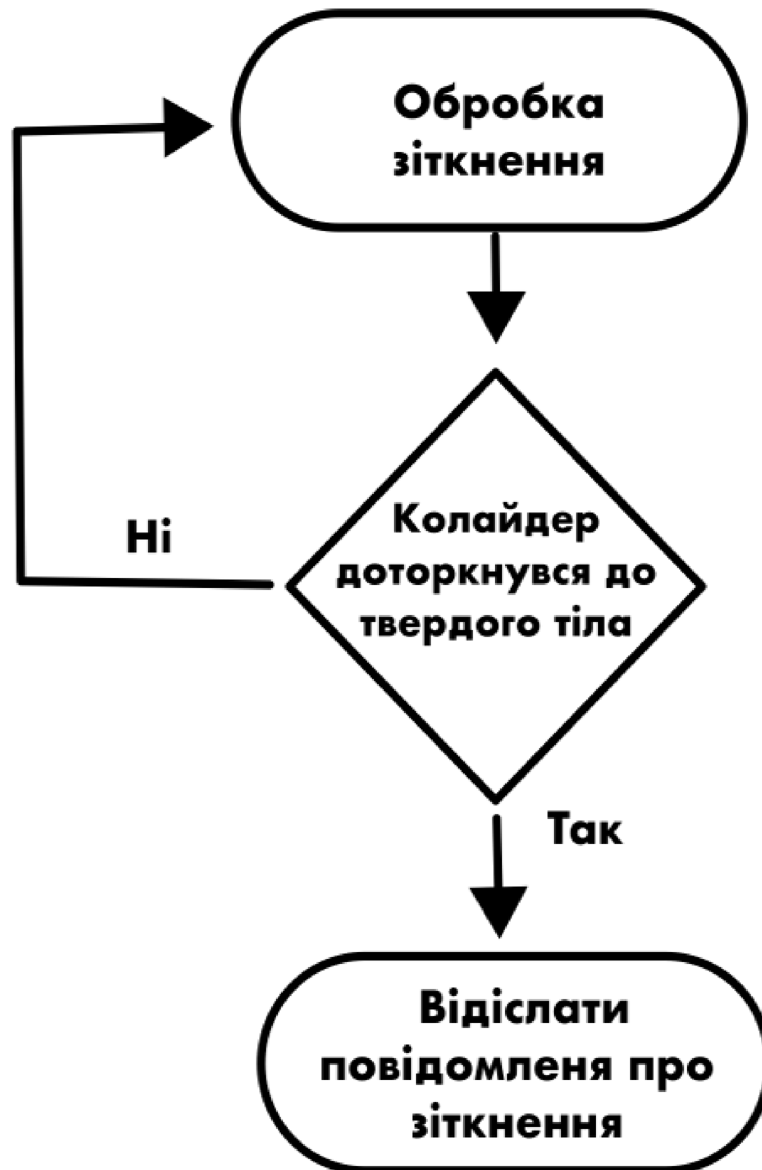

Рис 2.15. Блок - схема класу оброки зіткнення

Клас обробки здоров'я зав'язаний на повідомлені про зіткнення. Коли якийсь колайдер торкається обробника, то отримаємо доступ до GameObject, що зітнувся з колайдером, та перевіряємо на тег. Якщо тег дорівнює String значенням «Damage», то намагаємося взяти його компонент обробника, а звідти витягнути дані шкоди.

Дані шкоди зберігають лише значення шкоди. Як тільки отримаємо доступ до даних про шкоду, то застосовуємо до дати HealthData. Дата зберігає

кількість здоров'я обробника та надає доступ через інтерфейси до зміни цього значення. Коли значення стає менше, ніж «нуль», то об'єкт вмирає та викликається функція Destroy. (рис 2.16)

Рис 2.16. Блок схема поведінки обробки здоров'я

Дії Ворогів складаються з тих самих поведінок, окрім однієї відміни: це – штучний інтелект. Так як зараз усе управління здійснюється завдяки джойстику, треба створити нові можливості руху штучного інтелекту.

В цьому допоможіть пакет Unity – NavMesh.

NavMesh – пакет, що надає здобний функціонал розробнику для створення навігації та пошуку шляху. Ця навігаційна система дозволяє створювати штучний інтелект, який розумно рухається навколо ігрового світу, використовуючи навігаційну сітку, яка автоматично створюється з геометрії на сцені. (рис 2.17)

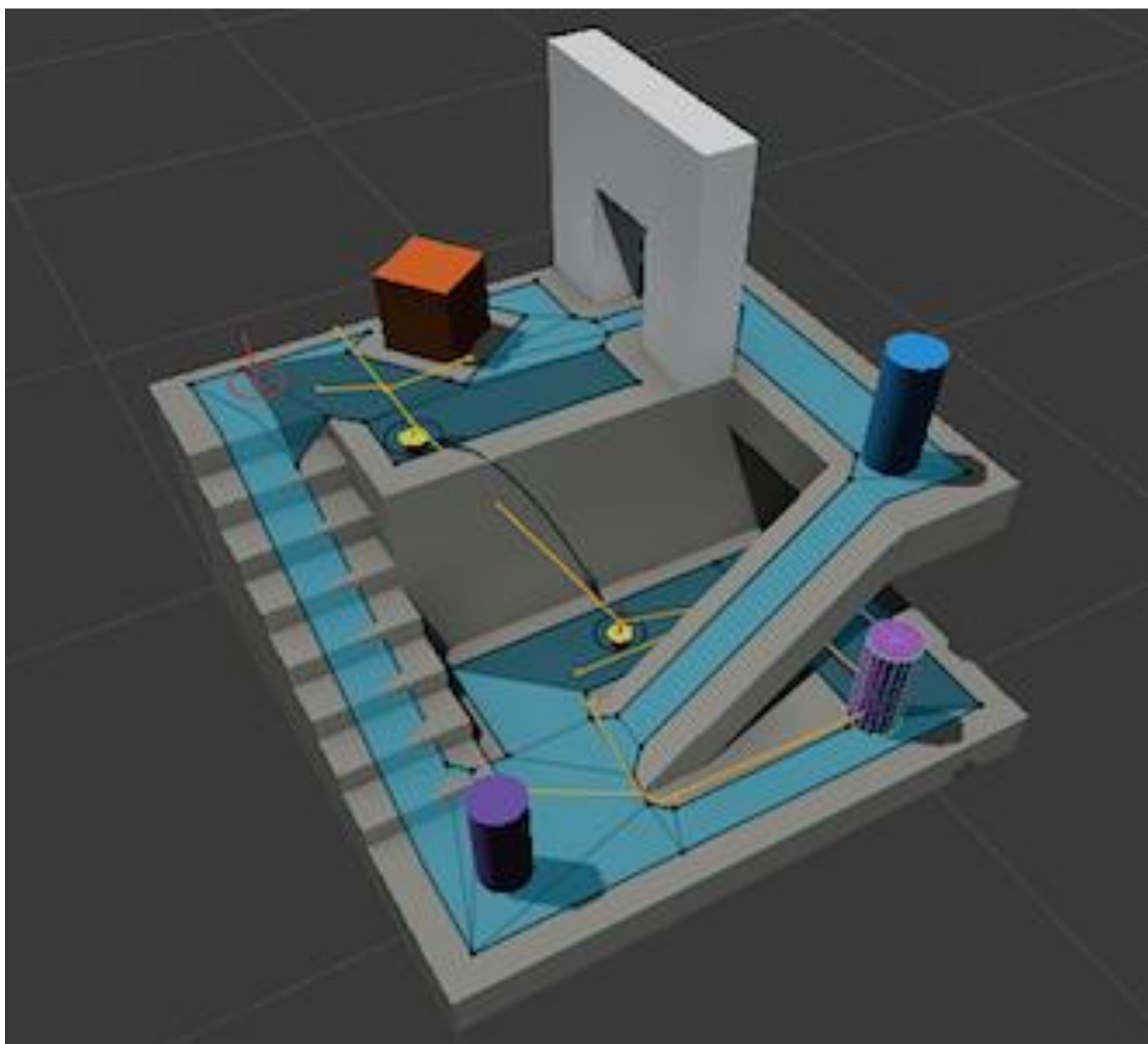

Рис 2.17. Зображення навігаційної сітки та пошуку шляху

Компонент, який розраховує шлях, називається NavMesh Agent. Він сам розраховує та обходить всі перешкоди на його шляху, використовуючи навігаційну сітку.

В моєму випадку навігаційна сітка використовується для розробки простішого ворога.

Ворог буде мати два стани: очікування та рух. Стани змінюються кожні дві секунди, знаходячи випадкову точку на площині, та по аналгії з класом руху гравця, відправляють повідомлення зміни свого стану. (рис 2.18)

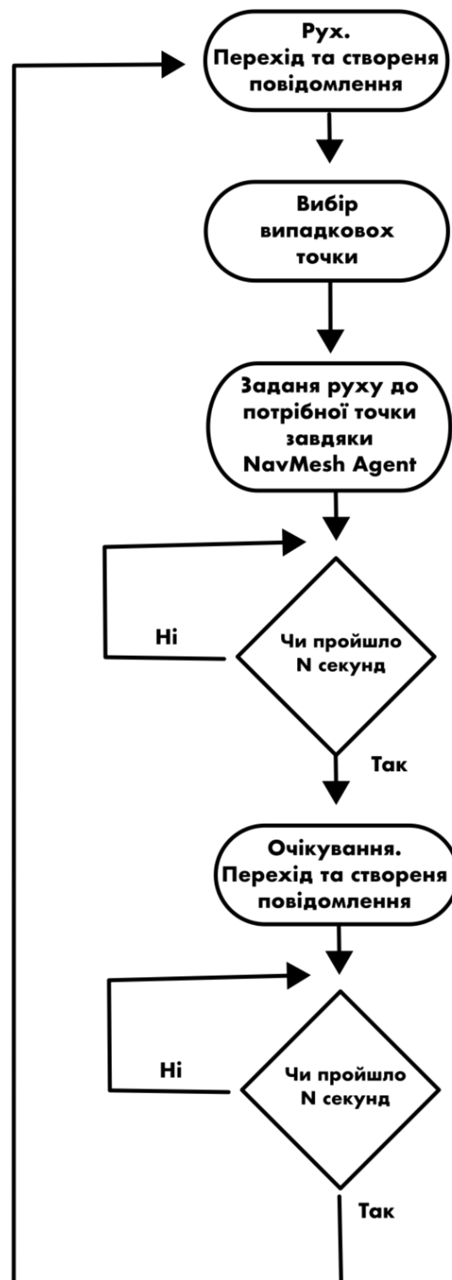

Рис 2.18. Блок схема руху ворога

2.7. Опис розробленого програмного забезпечення

Після зборки в один в пакет, отримаємо виконавчий файл з розширенням .apk, який можливо відкрити на мобільному пристрої на базі ОС Android та у емуляторі для запуску виконавчих файлів типу .apk.

2.7.1. Використані технічні засоби

Під час розробки ПЗ була використана персональна ЕОМ з наступними характеристиками:

- операційна система macOS Monterey;
- чип Apple M1. 8-ядерний процесор: 4 ядра продуктивності;
- 8-ядерний графічний процесор, 16-ядерна система Neural Engine;
- пам'ять 8 ГБ Об'єднана пам'ять;
- накопичувач 256 ГБ;
- підтримка відео;
- можливість підключення до двох моніторів;
- можливості підключення Ethernet;
- зв'язок Wi-Fi.

2.7.2. Використані програмні засоби

Перелік програмних засобів:

– ПЗ було створена завдяки інструменту для розробки ігор Unity 2021.2.8f1 SILICON;

– IDE було обрано середовище Rider від компанії Jetbrains;

– контроль версій – git, з використанням консолі та візуальної оболонки Fork;

– для створення зображень та блок-схем використовувався Vectornator;

– для пошуку безкоштовних 3Д моделей використовувався браузер Google Chrome.

2.7.3. Виклик та завантаження програми

Для запуску програми, потрібно встановити .apk файл на мобільний пристрій чи емулятор з ОС Android.

2.7.4. Опис інтерфейсу користувача

Інтерфейс додатку є інтуїтивно-зрозумілим та стандартним для цього типу ігор, та не має перевантаженого функціоналу, виконуючи лише найважливіші функції.

При вході гравець бачить напис великими буквами написано «TAP TO PLAY», як показано на рисунку 2.19.

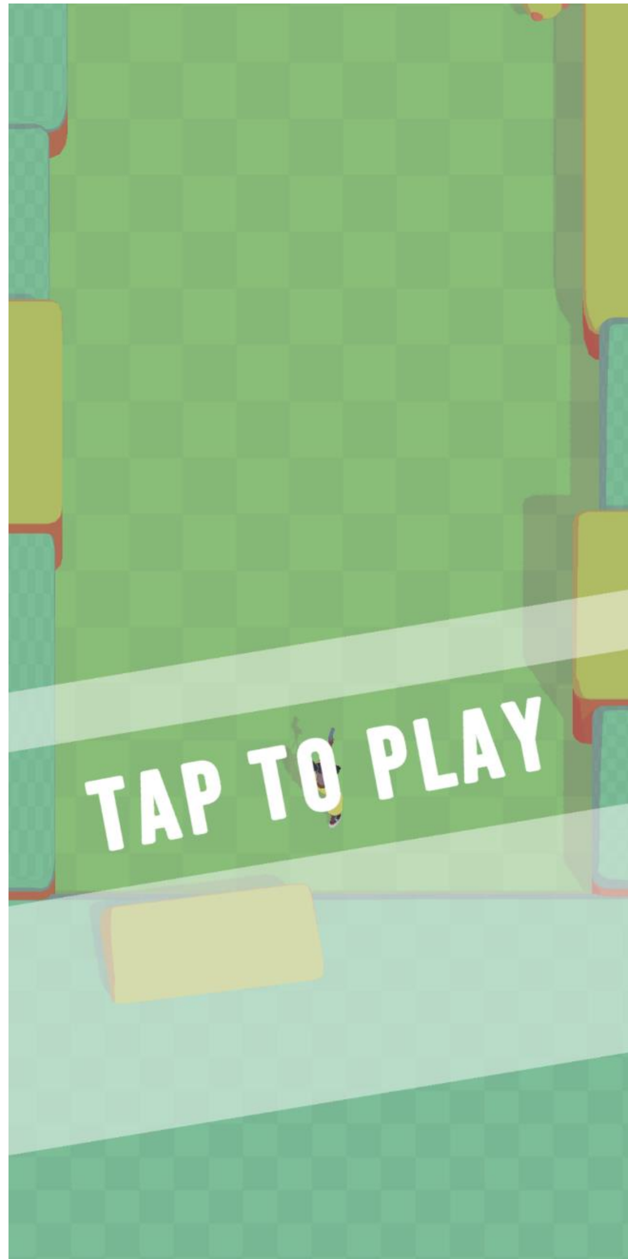

Рис 2.19. Інтерфейс старту гри

Також, є фінальний екран виграшу, з великим надписом «YOU WIN» і кнопкою «START NEW GAME». (рис 2.20)

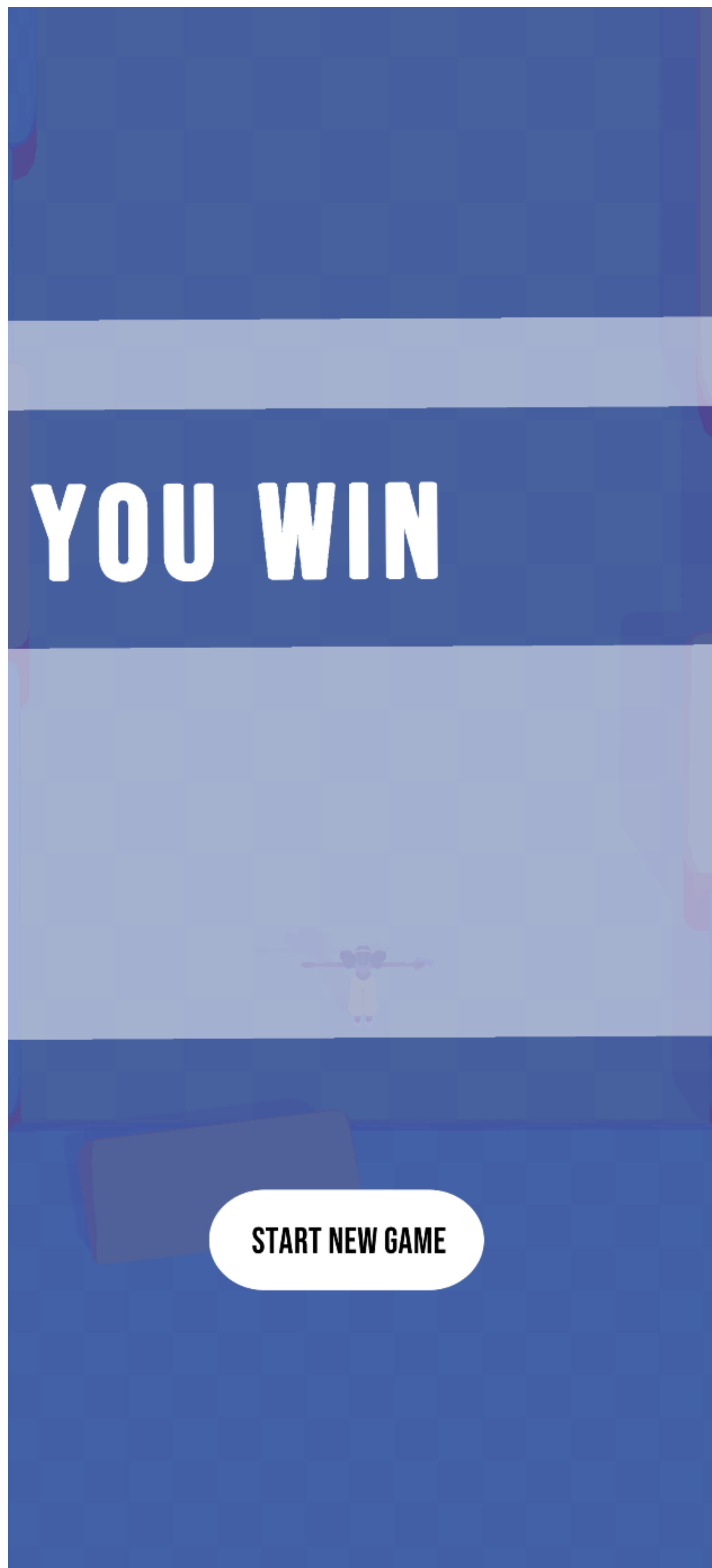

Рис 2.20. Інтерфейс виграшу

На фінальному екрані, при програші, з великим надписом «YOU LOSE» і кнопка «RESTART» (рис 2.21)

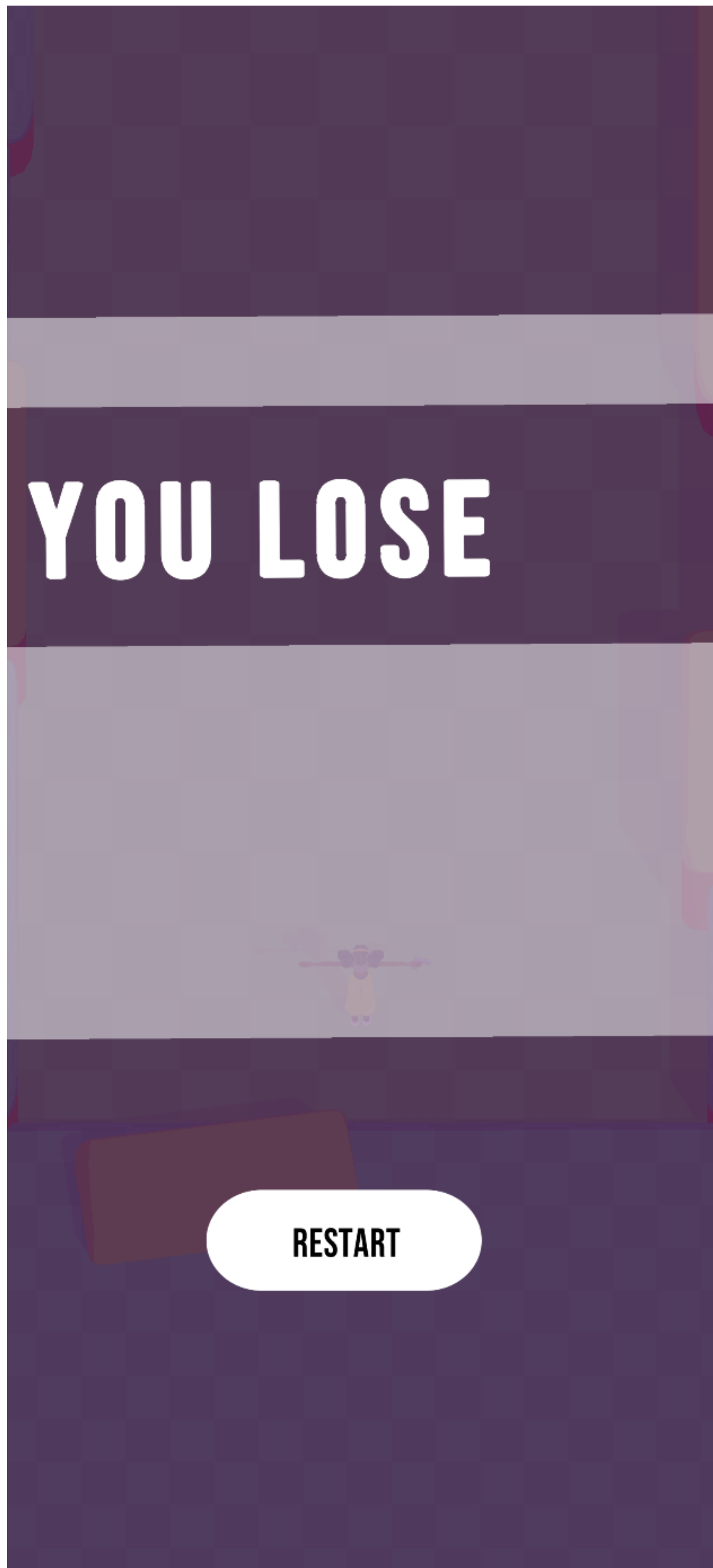

Рис 2.21. Інтерфейс програшу гри

РОЗДІЛ 3
ЕКОНОМІЧНИЙ РОЗДІЛ

3.1 Визначення трудомісткості та вартості розробки програмного продукту

Початкові дані:

1. передбачуване число операторів програми – 2042;
2. коефіцієнт складності програми – 1,4;
3. коефіцієнт корекції програми в ході її розробки – 0,2;
4. годинна заробітна плата програміста – 150 грн/год;
5. коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1,3;
6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 0.95;
7. вартість машино-години ЕОМ – 50 грн/год.

Через творчий характер роботи програміста, нормування праці в процесі створення програмного забезпечення є дуже складним. З цієї причини трудомісткість розробки ПЗ може бути визначена на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

$$t = t_о + t_и + t_а + t_п + t_{отл} + t_д ,$$

де $t_о$ – витрати праці на підготовку й опис поставленої задачі (приймається 50);

$t_и$ – витрати праці на дослідження алгоритму рішення задачі;

$t_а$ – витрати праці на розробку блок-схеми алгоритму;

$t_п$ – витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ – витрати праці на налагодження програми на ЕОМ;

$t_д$ – витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у ПЗ, яке розробляється.

Умовне число операторів (підпрограм):

$$Q = q \cdot C \cdot (1 + p)$$

де q – передбачуване число операторів (2000);

C – коефіцієнт складності програми (1,7);

p – коефіцієнт кореляції програми в ході її розробки (0,2).

$$Q = 2042 \cdot 1{,}4 \cdot (1 + 0{,}2) = 3430{,}56;$$

Витрати праці на вивчення опису задачі визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_u = \frac{Q \cdot B}{(75 \ldots 85) \cdot K}$$

де B – коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі (1,3)

K – коефіцієнт кваліфікації програміста, обумовлений стажем роботи з даної спеціальності (0.95);

$$t_u = \frac{3430{,}56 \cdot 1{,}3}{85 \cdot 0.95} = 55{,}23, \text{ людино-годин.}$$

Витрати праці на розробку алгоритму рішення задачі:

$$t_a = \frac{Q}{(20 \ldots 25) \cdot K}$$

$$t_a = \frac{3430{,}56}{20 \cdot 0.95} = 180{,}55 \text{ людино-годин.}$$

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20 \ldots 25) \cdot K}$$

$$t_n = \frac{3430{,}56}{25 \cdot 0{,}95} = 144{,}44 \text{ людино-годин.}$$

Витрати праці на налагодження програми на ЕОМ:

– за умови автономного налагодження одного завдання:

$$t_{\text{отл}} = \frac{Q}{(4 \ldots 5) \cdot K};$$

$$t_{\text{отл}} = \frac{3430{,}56}{5 \cdot 0{,}95} = 722{,}22, \text{ людино-годин,}$$

– за умови комплексного налагодження завдання:

$$t_{\text{отл}}^{\text{к}} = 1{,}2 \cdot t_{\text{отл}}$$

$$t_{\text{отл}}^{\text{к}} = 1{,}2 \cdot 722{,}22 = 866{,}66, \text{ людино-годин}$$

Витрати праці на підготовку документації:

$$t_{\partial} = t_{\partial p} + t_{\partial o}$$

де $t_{\partial p}$ – трудомісткість підготовки матеріалів і рукопису

$$t_{\partial \text{р}} = \frac{Q}{(15 \ldots 20) \cdot K}$$

$$t_{\partial \text{р}} = \frac{3430{,}56}{20 \cdot 0.95} = 180{,}55, \text{ людино-годин,}$$

де $t_{\partial o}$ – трудомісткість редагування, печатки й оформлення документації

$$t_{\partial o} = 0{,}75 \cdot t_{\partial \text{р}}$$

$$t_{\partial o} = 0{,}75 \cdot 180{,}55 = 135{,}41, \text{ людино-годин}$$

$$t_{\partial} = 180{,}55 + 135{,}41 = 315{,}96, \text{ людино-годин}$$

Отримаємо трудомісткість розробки програмного забезпечення:

t = 50 + 55,23 + 180,55 + 144,44 + 722,22 + 315,96 = 1468,4 людино-годин

У результаті ми розрахували, що в загальній складності необхідно 1468,4 людино-годин для розробки даного програмного забезпечення.

3.2. Витрати на створення програмного забезпечення

Витрати на створення ПЗ *Кпо* включають витрати на заробітну плату виконавця програми $З_{ЗП}$ і витрат машинного часу, необхідного на налагодження програми на ЕОМ.

$$К_{ПО} = З_{ЗП} + З_{МВ} \text{ грн,}$$

де *Зп* - заробітна плата виконавців, яка визначається за формулою:

$$З_{ЗП} = t \cdot C_{ПР}, \text{ грн}$$

де t - загальна трудомісткість, людино-годин;

$C_{ПР}$ – середня годинна заробітна плата програміста, грн/година.

З урахуванням того, що середня годинна зарплата програміста становить 120 грн/год, то отримаємо:

$$З_{\text{ЗП}} = 1468{,}4 \cdot 150 = 220260, \text{ грн.}$$

Вартість машинного часу $З_{МВ}$, необхідного для налагодження програми на ЕОМ, визначається за формулою:

$$З_{МВ} = t_{отл} \cdot C_{М} \text{, грн}$$

де $t_{отл}$ – трудомісткість налагодження програми на ЕОМ, год;

$C_{МЧ}$ – вартість машино-години ЕОМ, грн/год.

$$З_{МВ} = 722{,}22 \cdot 50 = 36111 \text{ грн.}$$

Звідси витрати на створення програмного продукту:

$$К_{ПО} = 220260 + 36111 = 256371 \text{ грн.}$$

Очікуваний період створення ПЗ:

$$T = \frac{t}{Bk * Fp}$$

де B_k – число виконавців;

F_p – місячний фонд робочого часу (при 40 годинному робочому тижні Fp=160 годин).

Витрати на створення програмного продукту:

$$T = \frac{1468{,}4}{1 \cdot 160} = 9{,}17 \text{ міс.}$$

ВИСНОВКИ

Метою кваліфікаційної работи є розробка програмного забезпечення для комп'ютерної гри жанру Rogue-Like.

В процесі виконнаня була сформована цільова мета, та розроблене програмне забезпечення, яке можливо виконати як виконавчий файл на мобільних пристроях з операційною системи Android.

Була спроектована система з використанням паттернів проектування, та розроблена кор механіка гравця з введеням даних через джойстик, штучний інтелект ворога та інтерфейс користувача.

В якості інструмента для розробеи ігор використовувався Unity, з редактором коду Rider від Jetbtains.

Під час розробки були використані стандартні пакети (бібліотеки) Unity для работи з фізикою, штучним інтелектов, адаптивною версткою UI, та Scriptable Render Pipeline для створення візуального стилю гри. Також використовувались стороні білбліотеки для Unity розроблених посторонімі розробникам. Вони використовувалися для впровадження залежностей в класси, та для створення простих анімацій кодом.

Увесь код створенний на високорівневій мові програмування C#

В кваліфікаційній работі було прорахвано трудомісткість розробки програмного забезпечення (1468,4 год), вартість розробки гри (256371 грн), та розрахований період розробки додатку (9,17 місяці)

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Unity Doc : https://docs.unity3d.com/ScriptReference/index.html
2. Unity in Action: Multiplatform Game Development
3. Wikipedia про значення гри: https://uk.wikipedia.org/wiki/%D0%92%D1%96%D0%B4%D0%B5%D0%BE%D0%B3%D1%80%D0%B0
4. Unity и C#. Геймдев от идеи до реализации. 2-е изд Джереми Гибсон Бонд. 2019
5. HLSL/CG Library: https://developer.download.nvidia.com/cg/index_stdlib.html
6. Unity 5.x Shaders and Effects Cookbook Кеннет Ламмерс 2016
7. Unity Game Development Cookbook: Essentials for Every Game 1st Edition. Jon Manning. 2016
8. C# для профессіоналів: тонкості программування, 3-е изд. Скіт, Джон. 2021
9. C# 9.0. Справочник. Повний опис мови. Джозеф Альбахарі. 2021
10. Zeenject: https://github.com/modesttree/Zenject
11. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C# 2013
12. Гейм дизайн. Яз зробити гру в яку будуть грати усі. Джесі Шел. 2021.
13. “Game Design Workshop: A Playcentric Approach to Creating Innovative Games” Tracy Fullerton. 2004
14. Паттерни проектування: https://refactoring.guru/uk/design-patterns
15. Design Patterns. Elements of Reusable. Object-Oriented Software. Erich Gamma. 2001
16. Бойко В.В. Экономика предприятий Украины. Основной курс: Учебник для вузов. - Д.: Пороги, 1997. - 312 с.
17. Андрусенко Г.О. Основы маркетинга. – Е.: НМК ВО, 1992. – 143 с.

18. Баркан Д.И. Маркетинг для всех. – Л.: «Культинформпрес», 1991.– 256 с.

19. БУСЫГИН А.В. Предпринимательство. Основной курс: Учебник для вузов. – М.: ИНФРА-М, 1997. – 608 с.

20. Завялов П.С., Демидов В.Е. Формула успеха – маркетинг. – М.: Междунар. Отношения, 1991.

21. Котлер Ф. Основы маркетинга: Пер. с англ. / Общ. Ред. и вступ. ст. Е.М. Пеньковой – М.: Прогресс, 1990. – 636с.

22. Скворцов Н.Н. Бизнес-план предприятия. – К.: Вища школа, 1995. – 187 с. 8. Мескон М.Х., Альберт М., Хедоури Ф. Основы менеджмента. – М.: Дело, 1992. - 702 с.

23. Мете А.Ф., Штец К.А., Бельгольский Б.П. и др. Организация и планирование предприятий. - М.: Металлургия, 1986. - 560 с.

24. Основы инновационного менеджмента: Теория и практика: Учеб. пособие / Под ред. П.Н. Завлина и др. – М: ОАО Издательство «Экономика». 2000. – 475 с.

25. Савчук В.П., Прилипко С.И., Величко Е.Г. Анализ и разработка инвестиционных проектов. - Учебное пособие. - Киев: Абсолют-В. Эльга. 1999. - 304 с.

ДОДАТОК А

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

public interface IHandler : IPausable {
  public DiContainer GameObjectContainer { get; }
  public SignalBus SignalContainer { get; }
  public void SetData(IData data);
  public void SetBehavior(IBehavior behavior);

  public bool GetData<T>(DataType data, out T outData) where T : IData;
  public bool GetBehavior(StrategyAct _strategy, out IBehavior outBehavior);

  public void DestroyMe();
  public void NotifyAll(INotification notification);
  GameObject GameObjectOwner { get; }
  public Action<Collider> WhenTriggerEnter { get; set; }
}

public abstract class Handler : MonoBehaviour, IHandler {
  private readonly Dictionary<StrategyAct, IBehavior> _behaviors = new();
  private readonly Dictionary<DataType, IData> _data = new();
  private Context _gameObjectContainer;
  private SignalBus _signalBus;

  public void DestroyMe() {
    Destroy(gameObject);
  }

  public bool GetBehavior(StrategyAct _strategy, out IBehavior outBehavior) {
    if (_behaviors.ContainsKey(_strategy)) {
      outBehavior = _behaviors[_strategy];
      return true;
    }

    outBehavior = null;
    return false;
  }

  public bool GetData<T>(DataType data, out T outData) where T : IData {
    if (_data.ContainsKey(data)) {
      if (_data[data] is T tData) {
        outData = tData;
      } else {
        outData = default;
        return false;
      }

      return true;
    }

    outData = default;
    return false;
  }

  public void NotifyAll(INotification notification) {
    foreach (var behavior in _behaviors.Values) {
      behavior.Notify(notification);
```

```
    }
  }

  public void SetBehavior(IBehavior behavior) {
    if (_behaviors.ContainsKey(behavior.Act)) {
      RemoveBehavior(behavior.Act);
    }

    behavior.CurrentHandler = this;
    _behaviors.Add(behavior.Act, behavior);
    GameObjectContainer.Inject(behavior);
    behavior.Init();
  }

  public void SetData(IData data) {
    if (_data.ContainsKey(data.CurrentType)) {
      RemoveData(data.CurrentType);
    }

    data.CurrentHandler = this;

    GameObjectContainer.Inject(data);
    _data.Add(data.CurrentType, data);
  }

  private void Awake() {
    TryGetComponent(out _gameObjectContainer);

    Pause += OnPause;
    UnPause += OnUnPause;
  }

  private void Update() {
    foreach (var behavior in _behaviors.Values.Where(behavior => behavior.UpdateType ==
UpdateType.NormalUpdate)) {
      behavior.Update();
    }
  }

  private void FixedUpdate() {
    foreach (var behavior in _behaviors.Values.Where(behavior => behavior.UpdateType ==
UpdateType.PhysicsUpdate)) {
      behavior.Update();
    }
  }

  private void OnDestroy() {
    foreach (var behavior in _behaviors) {
      behavior.Value.Dispose();
    }

    _behaviors.Clear();

    Pause -= OnPause;
    UnPause -= OnUnPause;
  }

  private void OnCollisionEnter(Collision collision) {
    WhenTriggerEnter?.Invoke(collision.collider);
  }

  private void OnTriggerEnter(Collider other) {
    WhenTriggerEnter?.Invoke(other);
```

```
  }

  private void OnPause() {
    foreach (var behavior in _behaviors) {
      behavior.Value.Pause?.Invoke();
    }
  }

  private void OnUnPause() {
    foreach (var behavior in _behaviors) {
      behavior.Value.UnPause?.Invoke();
    }
  }

  private void RemoveBehavior(StrategyAct act) {
    GetBehavior(act, out var lastBehavior);
    lastBehavior.Dispose();
    _behaviors.Remove(act);
  }

  private void RemoveData(DataType type) {
    GetData(type, out IData data);
    data.Dispose();
    _data.Remove(type);
  }

  public DiContainer GameObjectContainer {
    get {
      if (!_gameObjectContainer) {
        TryGetComponent(out _gameObjectContainer);
      }

      return _gameObjectContainer.Container;
    }
  }

  public GameObject GameObjectOwner => gameObject;

  public Action Pause { get; set; }

  public SignalBus SignalContainer {
    get {
      if (_signalBus == null) {
        SignalBusInstaller.Install(GameObjectContainer);
      }

      return _signalBus;
    }
  }

  public Action UnPause { get; set; }
  public Action<Collider> WhenTriggerEnter { get; set; }
}

using System;


public enum StrategyAct {
  Move,
  Animation,
  Jump,
```

```
  Attack,
  FindTarget,
  Collision,
  Health,
  TimesToDie,
  Die,
  RotateOnTouch,
  Speak
}

public interface ICanPause {
  public bool InPause { get; }

  public Action OnPauseChanged { get; set; }

  public void Pause();
  public void UnPause();
}

public interface IPausable {
  public Action Pause { get; set; }
  public Action UnPause { get; set; }
}

public interface IBehavior : IDisposable, IVisitable, IPausable {
  public StrategyAct Act { get; }
  public IHandler CurrentHandler { get; set; }

  public void Notify(INotification notification);

  public void Init();
  public void Update();

  public UpdateType UpdateType { get; }
}

public enum UpdateType {
  NormalUpdate,
  PhysicsUpdate
}
```

```
using System;

public abstract class BaseBehavior : IBehavior {
  public virtual void Dispose() {
  }

  public virtual void Accept(IVisitor visitor) {
  }

  public Action Pause { get; set; }
  public Action UnPause { get; set; }
  public abstract StrategyAct Act { get; }
  public IHandler CurrentHandler { get; set; }

  public void DestroyMe() {
  }

  public virtual void Notify(INotification notification) {
  }

  public virtual void Init() {
  }

  public virtual void Update() {
  }

  public virtual UpdateType UpdateType => UpdateType.NormalUpdate;
}

using System;
using UnityEngine;

public enum DataType {
  BaseData,
  HealthData,
  MoveData,
  AttackData,
  FindTargetData
}

public interface IData : IVisitable, IDisposable {
  public IHandler CurrentHandler { get; set; }
```

```
  public DataType CurrentType { get; }
}

public interface IMoveData : IData {
  int Speed { get; }
  Vector3 MoveDirection { get; set; }
}

public abstract class Data : IData {
  public virtual void Accept(IVisitor visitor) {
    visitor.Visit(this);
  }

  public virtual void Dispose() {
  }

  public IHandler CurrentHandler { get; set; }
  public abstract DataType CurrentType { get; }
}

public class MoveData : Data, IMoveData {
  public int Speed { get; set; }
  public Vector3 MoveDirection { get; set; }
  public override DataType CurrentType => DataType.MoveData;
}


using System;
using UnityEngine;

public abstract class BaseHybridBehavior : MonoBehaviour, IBehavior {
  public virtual void Accept(IVisitor visitor) {
  }

  public virtual void Dispose() {
  }

  public virtual void Init() {
  }

  public virtual void Notify(INotification notification) {
  }
```

```
  public virtual void Update() {
  }

  public abstract StrategyAct Act { get; }
  public IHandler CurrentHandler { get; set; }

  public Action Pause { get; set; }
  public Action UnPause { get; set; }

  public virtual UpdateType UpdateType => UpdateType.NormalUpdate;
}

using System;
using UnityEngine;

public class BulletMoveBehavior : BaseBehavior {
  private IMoveData _data;
  private Rigidbody _rigidbody;

  public override void Init() {
    base.Init();
    CurrentHandler.GameObjectOwner.TryGetComponent(out _rigidbody);
    CurrentHandler.GetData(DataType.MoveData, out _data);
  }

  public override void Update() {
    base.Update();

    _rigidbody.velocity = CurrentHandler.GameObjectOwner.transform.forward * _data.Speed;
  }



  public override StrategyAct Act => StrategyAct.Move;
}



public class CollideNotification : ICollideNotification {
  public Collider _collider { get; set; }
}

public interface ICollideNotification : INotification {
```

```
  public Collider _collider { get; set; }
}


public class HealthBehavior : BaseBehavior {
  private IHealthData _healthData;

  public override void Init() {
    base.Init();
    CurrentHandler.GetData(DataType.HealthData, out _healthData);
  }

  public override void Notify(INotification notification) {
    base.Notify(notification);

    if (notification is ICollideNotification collideData) {
      if (collideData._collider.CompareTag("Damage")) {
        var projectile = collideData._collider.GetComponent<Projectile>();
        IDamageData damageData;
        projectile.GetData(DataType.AttackData, out damageData);
        _healthData.ChangeHealthByFloat(-damageData.Damage);
      }
    }
  }

  public override StrategyAct Act => StrategyAct.Health;
}

public class HealthData : Data, IHealthData {
  public HealthData(float startHealth) {
    CurrentHealth = startHealth;
  }

  public override DataType CurrentType => DataType.HealthData;

  public float CurrentHealth { get; private set; }

  public void ChangeHealthByFloat(float value) {
    CurrentHealth += value;
    CheckDie();
  }
```

```
  public void SetHealthToFloat(float value) {
    CurrentHealth = value;
    CheckDie();
  }

  private void CheckDie() {
    if (CurrentHealth < 0 && IAlive) {
      CurrentHandler.DestroyMe();
      IAlive = false;
      OnDie?.Invoke();
    }
  }

  public bool IAlive { get; private set; }
  public Action OnDie { get; set; }
}

public interface IHealthData : IData {
  float CurrentHealth { get; }

  void ChangeHealthByFloat(float value);
  void SetHealthToFloat(float value);

  bool IAlive { get; }
  Action OnDie { get; set; }
}

using UnityEngine;

public class CollisionBehavior : BaseHybridBehavior {
  private void OnCollisionEnter(Collision collision) {
    CurrentHandler.NotifyAll(new CollideNotification { _collider = collision.collider });
  }

  private void OnTriggerEnter(Collider other) {
    CurrentHandler.NotifyAll(new CollideNotification { _collider = other });
  }

  public override StrategyAct Act => StrategyAct.Collision;
}
```

```
public class FindTargetData : Data, IFindTargetData {
  public void Accept(IVisitor visitor) {
  }

  public void Dispose() {
  }

  public override DataType CurrentType => DataType.FindTargetData;
}

public interface IFindTargetData : IData {
}

using System.Linq;
using Script.Core.StateMachine;
using UnityEngine;
using Zenject;

public class FindTargetForPlayerBehavior : BaseBehavior {
  private IStateMachine<FindTarget> _stateMachine;
  private ILevelData _data;

  [Inject]
  private void Construct(ILevelData data) {
    _data = data;
  }

  public override void Init() {
    base.Init();

    _stateMachine = new StateMachine<FindTarget>();
    _stateMachine.States[FindTarget.looking].Update += SearchForTarget;
  }

  public override void Update() {
    base.Update();

    _stateMachine.CurrentState.Update?.Invoke();
  }

  public override void Notify(INotification notification) {
    base.Notify(notification);
```

```
    if (notification is IChangeMoveState state) {
      _stateMachine.ChangeState(state.MoveState == MoveEnum.wait ? FindTarget.looking : FindTarget.wait);
    }
  }


  private void SearchForTarget() {
    var enemy = _data.Enemies
      .OrderBy(unit => Vector3.Distance(unit.transform.position, _data.Player[0].transform.position)).First();

    CurrentHandler.NotifyAll(new FindTargetNotification { Target = enemy.transform });
  }

  public override StrategyAct Act => StrategyAct.FindTarget;
  public override UpdateType UpdateType => UpdateType.PhysicsUpdate;
}

public class FindTargetForEnemyBehavior : BaseBehavior {
  private IStateMachine<FindTarget> _stateMachine;
  private ILevelData _data;

  [Inject]
  private void Construct(ILevelData data) {
    _data = data;
  }

  public override void Init() {
    base.Init();

    _stateMachine = new StateMachine<FindTarget>();
    _stateMachine.States[FindTarget.looking].Update += SearchForTarget;
  }

  public override void Update() {
    base.Update();

    _stateMachine.CurrentState.Update?.Invoke();
  }

  public override void Notify(INotification notification) {
    base.Notify(notification);
```

```
    if (notification is IChangeMoveState state) {
      _stateMachine.ChangeState(state.MoveState == MoveEnum.wait ? FindTarget.looking : FindTarget.wait);
    }
  }


  private void SearchForTarget() {
    var enemy = _data.Player.OrderBy(player =>
      Vector3.Distance(player.transform.position, CurrentHandler.GameObjectOwner.transform.position)).First();

    CurrentHandler.NotifyAll(new FindTargetNotification { Target = enemy.transform });
  }

  public override StrategyAct Act => StrategyAct.FindTarget;
  public override UpdateType UpdateType => UpdateType.PhysicsUpdate;
}

using ModestTree;
using UnityEngine;

public class InputManager : MonoBehaviour, IInputManager {
  [SerializeField] private Joystick _joystick;
  [SerializeField] private Transform _projectedSurface;

  private void Start() {
    if (_joystick == null) {
      Debug.LogError("Pls set joystick to input manager");
      return;
    }

    if (_projectedSurface == null) {
      _projectedSurface = new GameObject().transform;

      Assert.IsNotNull(Camera.main);
      _projectedSurface.parent = Camera.main.transform;
      _projectedSurface.localPosition = Vector3.zero;
      _projectedSurface.localRotation = Quaternion.Euler(Vector3.right * -60f);
    }
  }

  public Vector3 InputDirection =>
```

```
    _joystick.Horizontal * _projectedSurface.right + _joystick.Vertical * _projectedSurface.forward;
}

public interface IInputManager {
  Vector3 InputDirection { get; }
}

public interface IVisitable {
  public void Accept(IVisitor visitor);
}

public interface IVisitor {
  public void Visit(IVisitable visitable);
}

using System.Collections.Generic;
using UnityEngine;

public class LevelData : MonoBehaviour, ILevelData {
  [SerializeField] public GameObject _floor;
  [SerializeField] public List<Collider> _walls;
  [SerializeField] public List<Collider> _obstacles;
  [SerializeField] public List<Unit> _player;
  [SerializeField] public List<Unit> _enemies;
  public List<Unit> Enemies => _enemies;


  public GameObject Floor => _floor;

  public List<Collider> Obstacles {
    get => _obstacles;
    set => _obstacles = value;
  }

  public List<Unit> Player => _player;
  public List<Collider> Walls => _walls;
}

public interface ILevelData {
  public GameObject Floor { get; }

  public List<Collider> Walls { get; }
```

```
  public List<Collider> Obstacles { get; set; }

  public List<Unit> Player { get; }

  public List<Unit> Enemies { get; }
}

using UnityEngine;
using Zenject;

public class MonoSetBullet : MonoInstaller<MonoSetBullet> {
  [SerializeField] private Bullet _bullet;

  public override void InstallBindings() {
    Container.Bind<IBullet>().FromInstance(_bullet).AsCached();
  }
}

using DG.Tweening;
using Script.Core.StateMachine;
using UnityEngine;
using UnityEngine.AI;
using Zenject;

public class MoveBehavior : BaseBehavior {
  private IMoveData _data;
  private IInputManager _inputManager;
  private IStateMachine<MoveEnum> _stateMachine;
  private Rigidbody _rigidbody;
  private SignalBus _bus;

  [Inject]
  private void Construct(IInputManager inputManager, SignalBus bus) {
    _inputManager = inputManager;

    var rigidbody = CurrentHandler.GameObjectContainer.TryResolve<Rigidbody>();

    if (!rigidbody) {
      if (CurrentHandler.GameObjectOwner.TryGetComponent(out _rigidbody)) {
        CurrentHandler.GameObjectContainer.Bind<Rigidbody>().FromInstance(_rigidbody);
      }
```

```
    } else {
      _rigidbody = rigidbody;
    }

    _bus = bus;
  }

  public override void Init() {
    base.Init();

    if (CurrentHandler.GetData(DataType.MoveData, out IMoveData moveData)) {
      _data = moveData;
    }

    _stateMachine = new StateMachine<MoveEnum>();

    _stateMachine.States[MoveEnum.wait].Init += StartWait;
    _stateMachine.States[MoveEnum.wait].Update += Slowing;

    _stateMachine.States[MoveEnum.move].Init += StartMove;
    _stateMachine.States[MoveEnum.move].Update += CalculateMove;
  }

  private void StartWait() {
    CurrentHandler.NotifyAll(new ChangeMoveState(MoveEnum.wait));
  }

  private void StartMove() {
    CurrentHandler.NotifyAll(new ChangeMoveState(MoveEnum.move));
  }



  public override void Update() {
    base.Update();
    CalculateDirection();
    StateMachineControlMethod();
    _stateMachine.CurrentState.Update?.Invoke();
  }

  private void StateMachineControlMethod() {
    if (Vector3.Magnitude(_data.MoveDirection) < 0.01f && _stateMachine.CurrentState.StateType ==
MoveEnum.move) {
```

```
      _stateMachine.ChangeState(MoveEnum.wait);
    } else if (Vector3.Magnitude(_data.MoveDirection) > 0.1f && _stateMachine.CurrentState.StateType !=
MoveEnum.move) {
      _stateMachine.ChangeState(MoveEnum.move);
    }
  }

  private void Slowing() {
    _rigidbody.velocity = Vector3.Lerp(_rigidbody.velocity, Vector3.zero, Time.fixedDeltaTime * 2f);
  }

  private void CalculateMove() {
    Move();
    Rotate();
  }

  private void CalculateDirection() {
    _data.MoveDirection = _inputManager.InputDirection;
  }

  private void Move() {
    var moveDelta = _data.MoveDirection * _data.Speed * Time.fixedDeltaTime;
    _rigidbody.velocity = moveDelta;
  }

  private void Rotate() {
    if (_data.MoveDirection.magnitude == 0) {
      return;
    }

    _rigidbody.MoveRotation(Quaternion.LookRotation(_data.MoveDirection,
CurrentHandler.GameObjectOwner.transform.up));
  }

  public override StrategyAct Act => StrategyAct.Move;
  public override UpdateType UpdateType => UpdateType.PhysicsUpdate;
}

public class ChangeMoveState : IChangeMoveState {
  public ChangeMoveState(MoveEnum moveState) {
    MoveState = moveState;
  }
```

```
  public MoveEnum MoveState { get; }
}

public interface IChangeMoveState : INotification {
  MoveEnum MoveState { get; }
}

public enum MoveEnum {
  wait,
  move
}

public enum FindTarget {
  looking,
  wait
}

public class FindTargetNotification : IFindTargetNotification {
  public Transform Target { get; set; }
}

public interface IFindTargetNotification : INotification {
  Transform Target { get; set; }
}

public enum ShootEnum {
  wait,
  shoot
}

public class ShootBehavior : BaseBehavior {
  private IStateMachine<ShootEnum> _stateMachine;
  protected IBullet _bullet;
  protected Gun _gun;
  protected Animator _animator;

  private readonly float _startDelay = 0.15f;
  private float _timerToDelay;
  protected bool _isRotatedToTarget;
  protected bool _isCanAttack = true;
```

```
protected Transform _target;

[Inject]
private void Construct(IBullet bullet) {
  _bullet = bullet;
}

public override void Init() {
  base.Init();

  _stateMachine = new StateMachine<ShootEnum>();



  _gun = CurrentHandler.GameObjectOwner.GetComponentInChildren<Gun>();

  _stateMachine.States[ShootEnum.shoot].Init += OnShootInit;
  _stateMachine.States[ShootEnum.shoot].Update += RotateToTarget;
  _stateMachine.States[ShootEnum.shoot].Update += Shoot;

  _animator = CurrentHandler.GameObjectOwner.GetComponentInChildren<Animator>();
}

private void OnShootInit() {
  _timerToDelay = _startDelay;
  _isCanAttack = true;
  _isRotatedToTarget = false;
}

public override void Update() {
  base.Update();

  _stateMachine.CurrentState.Update?.Invoke();
}

private void RotateToTarget() {
  if (_timerToDelay > 0) {
    _timerToDelay -= Time.deltaTime;
    return;
  }
```

```
    CurrentHandler.GameObjectOwner.transform.rotation = Quaternion.Slerp(
      CurrentHandler.GameObjectOwner.transform.rotation,
      Quaternion.LookRotation(_target.position - CurrentHandler.GameObjectOwner.transform.position, Vector3.up),
      Time.deltaTime * 5f);

    if (Quaternion.Angle(CurrentHandler.GameObjectOwner.transform.rotation,
          Quaternion.LookRotation(_target.position - CurrentHandler.GameObjectOwner.transform.position, Vector3.up))
<
        15f) {
      _isRotatedToTarget = true;
    }
  }

  protected virtual void Shoot() {
    if (!_isCanAttack || !_isRotatedToTarget) {
      return;
    }

    _isCanAttack = false;
    if (_animator) {
      _animator.Play("Shoot");
    }


    var bullet =
      CurrentHandler.GameObjectContainer.InstantiatePrefabForComponent<IBullet>(_bullet.GameObjectOwner);

    bullet.GameObjectOwner.transform.parent = null;

    _gun.Shoot(bullet.GameObjectOwner);

    var rotation = Quaternion.LookRotation(_target.position - CurrentHandler.GameObjectOwner.transform.position,
      Vector3.up);
    rotation.x = 0;


    bullet.GameObjectOwner.transform.rotation = rotation;



    bullet.SetData(new MoveData { Speed = 24 });
    bullet.SetData(new DamageData { Damage = 1 });
    bullet.SetBehavior(new DieOnTouchBehavior(1));
```

```
    bullet.SetBehavior(bullet.GameObjectOwner.AddComponent<CollisionBehavior>());
    bullet.SetBehavior(new BulletMoveBehavior());
  }

  public void SetUnitInfo(Unit bullet) {
  }


  public override void Notify(INotification notification) {
    base.Notify(notification);


    if (notification is IChangeMoveState changeMoveState) {
      if (changeMoveState.MoveState == MoveEnum.move) {
        _stateMachine.ChangeState(ShootEnum.wait);
      }
    }

    if (notification is IFindTargetNotification findTargetNotification) {
      _target = findTargetNotification.Target;
      _stateMachine.ChangeState(ShootEnum.shoot);
    }
  }

  public override StrategyAct Act => StrategyAct.Attack;
}

public class PlayerShootBehavior : ShootBehavior {
  protected override void Shoot() {
    if (!_isCanAttack || !_isRotatedToTarget) {
      return;
    }

    _isCanAttack = false;
    if (_animator) {
      _animator.Play("Shoot");
    }


    for (var i = -1; i < 2; i++) {
      var bullet =
        CurrentHandler.GameObjectContainer.InstantiatePrefabForComponent<IBullet>(_bullet.GameObjectOwner);
```

```
      bullet.GameObjectOwner.transform.parent = null;

      _gun.Shoot(bullet.GameObjectOwner);
      var vector = _target.position - CurrentHandler.GameObjectOwner.transform.position;

      var rotation = Quaternion.LookRotation(vector,
        Vector3.up);
      rotation.x = 0;
      rotation = Quaternion.Euler(rotation.eulerAngles + Vector3.up * i * 30);

      bullet.GameObjectOwner.transform.rotation = rotation;



      bullet.SetData(new MoveData { Speed = 24 });
      bullet.SetData(new DamageData { Damage = 1 });
      bullet.SetBehavior(new DieOnTouchBehavior(1));
      bullet.SetBehavior(new RotateOnTouchBehavior());
      bullet.SetBehavior(bullet.GameObjectOwner.AddComponent<CollisionBehavior>());
      bullet.SetBehavior(new BulletMoveBehavior());
    }
  }
}



public class DieOnTouchBehavior : BaseBehavior, IDieOnTouchBehavior {
  public DieOnTouchBehavior(int timesToDestroy) {
    TimesToDestroy = timesToDestroy;
  }

  public override void Notify(INotification notification) {
    base.Notify(notification);

    if (notification is ICollideNotification collideData) {
      if (collideData._collider.gameObject.layer == LayerMask.NameToLayer("Player") ||
          collideData._collider.gameObject.layer == LayerMask.NameToLayer("Enemy")) {
        CurrentHandler.DestroyMe();
        return;
      }

      Debug.Log("");
      TimesToDestroy--;
```

```
      if (TimesToDestroy <= 0) {
        CurrentHandler.DestroyMe();
      }
    }
  }

  public int TimesToDestroy;



  public override StrategyAct Act => StrategyAct.TimesToDie;
}

public interface IDieOnTouchBehavior : IBehavior {
}

public class RotateOnTouchBehavior : BaseBehavior, IBehavior {
  private BulletLayerData _data;

  public override void Init() {
    base.Init();
    CurrentHandler.GameObjectOwner.GetComponent<BulletLayerData>();
  }

  public override void Notify(INotification notification) {
    base.Notify(notification);

    if (notification is ICollideNotification collideData) {
      Physics.Raycast(
        CurrentHandler.GameObjectOwner.transform.forward * -1 +
CurrentHandler.GameObjectOwner.transform.position,
        CurrentHandler.GameObjectOwner.transform.forward, out var hit, 5, _data.Mask);
    }
  }

  public override StrategyAct Act => StrategyAct.RotateOnTouch;
}



public interface IDamageData : IData {
  public float Damage { get; set; }
}
```

```
public class DamageData : Data, IDamageData {
  public override DataType CurrentType => DataType.AttackData;
  public float Damage { get; set; }
}



public class MoveEnemyBehavior : BaseBehavior {
  private IMoveData _data;
  private NavMeshAgent _agent;
  private readonly Bounds _bounds;
  private IStateMachine<MoveEnum> _stateMachine;
  private Rigidbody _rigidbody;
  private SignalBus _bus;

  [Inject]
  private void Construct(SignalBus bus) {
    _bus = bus;
  }

  public MoveEnemyBehavior(Bounds bounds) {
    _bounds = bounds;
  }

  public override void Init() {
    base.Init();

    _agent = CurrentHandler.GameObjectOwner.GetComponent<NavMeshAgent>();



    if (CurrentHandler.GetData(DataType.MoveData, out IMoveData moveData)) {
      _data = moveData;
    }

    _stateMachine = new StateMachine<MoveEnum>();

    _stateMachine.States[MoveEnum.wait].Init += StartWait;

    _stateMachine.States[MoveEnum.move].Init += StartMove;
    _stateMachine.ChangeState(MoveEnum.move);
  }

  private void StartWait() {
```

```
    _agent.ResetPath();
    CurrentHandler.NotifyAll(new ChangeMoveState(MoveEnum.wait));
    DOTween.To(() => 0, x => Mathf.Cos(x), 1, 2f).OnComplete(() => {
_stateMachine.ChangeState(MoveEnum.move); });
  }

  private void StartMove() {
    CurrentHandler.NotifyAll(new ChangeMoveState(MoveEnum.move));
    _agent.SetDestination(GetRandomPointInCube(_bounds));
    DOTween.To(() => 0, x => Mathf.Cos(x), 1, 2f).OnComplete(() => { _stateMachine.ChangeState(MoveEnum.wait);
});
  }

  public Vector3 GetRandomPointInCube(Bounds bounds) {
    return bounds.center + new Vector3((Random.value - 0.5f) * bounds.size.x, (Random.value - 0.5f) * bounds.size.y,
      (Random.value - 0.5f) * bounds.size.z);
  }



  public override StrategyAct Act => StrategyAct.Move;
  public override UpdateType UpdateType => UpdateType.PhysicsUpdate;
}

public interface INotification {
}

public class Notification : INotification {
}

using UnityEngine;
using Zenject;

namespace DefaultNamespace {
  public class PlayerFabric : MonoBehaviour {
    [SerializeField] private Handler _handler;
    [SerializeField] private CollisionBehavior _collisionBehavior;

    [SerializeField] private Context _container;

    private void Awake() {
      _handler.SetData(new MoveData {
        Speed = 300
```

```
      });

      _handler.SetData(new HealthData(5));
      _handler.SetBehavior(new MoveBehavior());
      _handler.SetBehavior(new SkinController());
      _handler.SetBehavior(new FindTargetForPlayerBehavior());
      _handler.SetBehavior(new ShootBehavior());
      _handler.SetBehavior(new HealthBehavior());
      _handler.SetBehavior(_collisionBehavior);
    }
  }
}

using UnityEngine;
using Zenject;

public class SceneMainMonoInstaller : MonoInstaller<SceneMainMonoInstaller> {
  [SerializeField] private InputManager _inputManager;
  [SerializeField] private LevelData _data;

  public override void InstallBindings() {
    SignalBusInstaller.Install(Container);
    Container.Bind<IInputManager>().FromInstance(_inputManager).AsCached();
    Container.Bind<ILevelData>().FromInstance(_data).AsCached();
  }
}

using Script.Core.StateMachine;
using UnityEngine;
using Zenject;

public enum AnimEnum {
  wait,
  move,
  attack
}

public class SkinController : BaseBehavior {
  private SignalBus _bus;
  private Rigidbody _rigidbody;
  private Animator _animator;
```

```
private IStateMachine<AnimEnum> _stateMachine;

private static readonly int Velocity = Animator.StringToHash("Velocity");

[Inject]
private void Construct(SignalBus bus) {
  _bus = bus;
}

public override void Init() {
  base.Init();



  _stateMachine = new StateMachine<AnimEnum>();

  _stateMachine.States[AnimEnum.wait].Update += MoveAnim;
  _stateMachine.States[AnimEnum.move].Update += MoveAnim;

  var rigidbody = CurrentHandler.GameObjectContainer.TryResolve<Rigidbody>();

  if (!rigidbody) {
    if (CurrentHandler.GameObjectOwner.TryGetComponent(out _rigidbody)) {
      CurrentHandler.GameObjectContainer.Bind<Rigidbody>().FromInstance(_rigidbody);
    }
  } else {
    _rigidbody = rigidbody;
  }

  _animator = CurrentHandler.GameObjectOwner.GetComponentInChildren<Animator>();
}

public override void Notify(INotification notification) {
  base.Notify(notification);

  if (notification is IChangeMoveState currentState) {
    _animator.Play("Move Tree");
  }
}

public override void Update() {
  base.Update();
```

```
    _stateMachine.CurrentState.Update?.Invoke();
  }

  private void MoveAnim() {
    _animator.SetFloat(Velocity, _rigidbody.velocity.magnitude / 6f);
  }

  public override StrategyAct Act => StrategyAct.Animation;
  public override UpdateType UpdateType => UpdateType.NormalUpdate;
}

using System;

namespace Script.Core.StateMachine {
  public class State<T> : IState<T> where T : Enum {
    public State(T type) {
      StateType = type;
    }

    public Action Update { get; set; }

    public T StateType { get; }
    public Action Init { get; set; }
  }

  public interface IState<T> where T : Enum {
    Action Update { get; set; }

    T StateType { get; }
    Action Init { get; set; }
  }
}

using System;
using System.Collections.Generic;

namespace Script.Core.StateMachine {
  public class StateMachine<T> : IStateMachine<T> where T : Enum {
    public StateMachine() {
      States = new Dictionary<T, IState<T>>();

      foreach (T value in Enum.GetValues(typeof(T))) {
```

```
      States.Add(value, new State<T>(value));
    }

    ChangeState((T)Enum.GetValues(typeof(T)).GetValue(0));
  }

  public void ChangeState(T newState) {
    if (CurrentState == States[newState]) {
      return;
    }

    CurrentState = States[newState];
    CurrentState.Init?.Invoke();
  }

  public Dictionary<T, IState<T>> States { get; }
  public IState<T> CurrentState { get; private set; }

  public Action<T> OnChangeState { get; set; }
 }

 public interface IStateMachine<T> where T : Enum {
  void ChangeState(T newState);
  public Dictionary<T, IState<T>> States { get; }
  IState<T> CurrentState { get; }
  Action<T> OnChangeState { get; set; }
 }
}

using System;
using System.Collections.Generic;

namespace Script.Core.StateMachine {
 public class StateMachine<T> : IStateMachine<T> where T : Enum {
  public StateMachine() {
    States = new Dictionary<T, IState<T>>();

    foreach (T value in Enum.GetValues(typeof(T))) {
      States.Add(value, new State<T>(value));
    }

    ChangeState((T)Enum.GetValues(typeof(T)).GetValue(0));
```

```
    }

    public void ChangeState(T newState) {
      if (CurrentState == States[newState]) {
        return;
      }

      CurrentState = States[newState];
      CurrentState.Init?.Invoke();
    }

    public Dictionary<T, IState<T>> States { get; }
    public IState<T> CurrentState { get; private set; }

    public Action<T> OnChangeState { get; set; }
  }

  public interface IStateMachine<T> where T : Enum {
    void ChangeState(T newState);
    public Dictionary<T, IState<T>> States { get; }
    IState<T> CurrentState { get; }
    Action<T> OnChangeState { get; set; }
  }
}

using UnityEngine;
using Zenject;

namespace DefaultNamespace {
  public class TargetStandFabric : MonoBehaviour {
    [SerializeField] private Handler _handler;
    [SerializeField] private CollisionBehavior _collisionBehavior;

    [SerializeField] private Context _container;

    private void Awake() {
      _handler.SetData(new HealthData(3));
      _handler.SetBehavior(_collisionBehavior);
      _handler.SetBehavior(new HealthBehavior());
    }
  }
}
```

```
public class Unit : Handler {
}

public class Projectile : Handler {
}

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;
using UnityEngine.UIElements;
using Zenject;
using Random = UnityEngine.Random;

public class EnemyFabric : MonoBehaviour {
   [SerializeField] private BoxCollider _collider;
   [SerializeField] private Unit _enemyPrefab;
   [SerializeField] private LevelData _data;
   [SerializeField] private SceneContext _context;

  private static int countOfEnemy = 3;

   private IEnumerator Start() {
      for (int i = 0; i < countOfEnemy; i++) {
         var enemy = _context.Container.InstantiatePrefabForComponent<Unit>(_enemyPrefab, transform);
         enemy.transform.position = GetRandomPointInCube(_collider.bounds);

        _data._enemies.Add(enemy);

         enemy.SetData(new MoveData {
            Speed = 300
          });

          yield return new WaitForSeconds(0.2f);

         enemy.SetData(new HealthData(1));
         enemy.SetBehavior(new MoveEnemyBehavior(_collider.bounds));
         enemy.SetBehavior(new FindTargetForEnemyBehavior());
         enemy.SetBehavior(new ShootBehavior());
         enemy.SetBehavior(new HealthBehavior());
```

```
            enemy.SetBehavior(new OnDieEnemyBehavior());
            enemy.SetBehavior( enemy.gameObject.AddComponent<CollisionBehavior>());
        }
    }

    public Vector3 GetRandomPointInCube(Bounds bounds) {
        return bounds.center + new Vector3((Random.value - 0.5f) * bounds.size.x, (Random.value - 0.5f) * bounds.size.y,
(Random.value - 0.5f) * bounds.size.z);
    }
}

public class OnDieEnemyBehavior : BaseBehavior, IBehavior {
    private ILevelData _levelData;

   [Inject]
    private void Construct(ILevelData levelData) {
        _levelData = levelData;
    }
    public override void Init() {
        base.Init();

        CurrentHandler.GetData(DataType.HealthData, out IHealthData _data);
        _data.OnDie += OnDie;
    }

    private void OnDie() {
        _levelData.Enemies.Remove(CurrentHandler as Unit);
    }

    public override StrategyAct Act => StrategyAct.Die;
}



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

public class Gun : MonoBehaviour {
  [SerializeField]
  private Transform ShootPoint;

  public void Shoot(GameObject bullet) {
```

```
    bullet.transform.position = ShootPoint.transform.position;
  }
}
```

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

public class Bullet : Handler, IBullet
{

}

public interface IBullet : IHandler {

}
```

ДОДАТОК Б

ВІДГУК
керівника економічного розділу
на кваліфікаційну роботу бакалавра
на тему:
"Розробка програмного забезпечення для комп'ютерної гри жанру roque-like з використанням фреймворка Unity та C#. "
студента групи 121-18-2 Корнієнко Данила Андрійовича

Керівник економічного розділу
доцент каф. ПЕП та ПУ, к.е.н **Л. В. Касьяненко**

ДОДАТОК В

Ім’я файла	**Опис**
Пояснювальні документи	
Корнієнко_121-18-2.doc	Пояснювальна записка до кваліфікаційної роботи. Документ Word.
Корнієнко_121-18-2.pdf	Пояснювальна записка до кваліфікаційної роботи. в форматі PDF
Програма	
Корнієнко_121-18-2.zip	Архів. Містить коди програми і откомпільовану програму
Презентація	
Корнієнко_121-18-2.ppt	Презентація кваліфікаційної роботи.