

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Гра-симулятор знешкодження рухомих об'єктів зі
складними траєкторіями руху у 3D просторі»**

Виконав студент групи ЗППЗ-19-1
ВОРОНЮК Іван Олександрович

Керівник роботи:
МАРЧУК Галина Вікторівна

Рецензент:
ЄФРЕМОВ Юрій Миколайович

Житомир – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

В.о. зав. кафедри інженерії
програмного забезпечення
Андрій МОРОЗОВ
«14» лютого 2023 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ВОРОНІЮК Іван Олександрович**

Керівник роботи: **МАРЧУК Галина Вікторівна**

Тема роботи: **«Гра-симулятор знешкодження рухомих об'єктів зі складними траєкторіями руху у 3D просторі»,**

затверджена Наказом закладу вищої освіти від «14» лютого 2023 р., №70с

Вихідні дані для роботи: **розробка та дослідження ігрової системи, яка забезпечує можливість за допомогою системи ППО збивати літаки зі складними траєкторіями руху у тривимірному просторі**

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Марчук Г.В.	20.02.2023р.	20.02.2023р.
2	Марчук Г.В.	20.04.2023р.	20.04.2023р.
3	Марчук Г.В.	16.05.2023р.	16.05.2023р.

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра складається з програмного комплексу «Гри-симулятора знешкодження рухомих об'єктів зі складними траєкторіями руху у 3D просторі» та пояснювальної записки. Пояснювальна записка до випускної роботи містить 64 сторінки, 36 ілюстрацій та 5 таблиць.

Метою роботи є розробка за допомогою рушія Unity гри-симулятора роботи ППО, що матиме можливість керувати пусковою установкою та наводитись на ціль за допомогою курсору .

В роботі визначено основні завдання на розробку системи, проаналізовано схожі додатки. Наведено сценарії роботи програмного комплексу, загальна структура програмного комплексу, опис алгоритмів взаємодії окремих модулів системи, опис об'єктної структури системи (діаграма класів). Також наводиться результати тестування системи.

КЛЮЧОВІ СЛОВА: ГРА, СИМУЛЯТОР, ППО, UNITY3D, BLENDER, UI, ГРАФІЧНИЙ ІНТЕРФЕЙС, РУШІЙ, СЕРЕДОВИЩЕ РОЗРОБКИ, C#.

					ІПЗ.КР.Б-121-23-ПЗ			
Зм.	Арк.	№ докум.	Підпис	Дата				
Розроб.		Воронюк І.О.			Гра-симулятор знешкодження рухомих об'єктів зі складними траєкторіями руху у 3D просторі.	Літ.	Арк.	Аркушів
Керівник		Марчук. Г. В.					3	
н. контр.		Тичина О.П.				Житомирська політехніка, група ЗІПЗ-19-1		
Зав. каф.		Морозов А.В.						

ABSTRACT

Bachelor's graduate work consists of software package of the air defense simulation system and a documentation. Documentation for bachelor's graduate work contain 64 pages, 36 illustrations, 5 tables.

The goal of the work is to develop, using the Unity engine, an air defense simulator game that will be able to control the launcher and aim at the target using the cursor.

Graduation work contains main tasks for system development, similar applications were analyzed. There are scenarios of the software package, the overall structure of the software system, description of module's algorithms interaction, description of the system objects structure (class diagram). The results of the system testing are also present in the graduate work.

KEYWORDS: GAME, SIMULATOR, ANTI-AIRCRAFT DEFENSE, UNITY3D, BLENDER, UI, GUI, ENGINE, DEVELOPMENT ENVIRONMENT, C#.

					ИП3.КР.Б-121-23-ИЗ	
						3

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ	5
РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ДЛЯ СИМУЛЯЦІЇ РОБОТИ ППО	8
1.1 ПОСТАНОВКА ЗАДАЧІ	8
1.2 АНАЛІЗ АНАЛОГІВ ПРОГРАМНОГО ПРОДУКТУ	9
1.3 ОБґРУНТУВАННЯ ВИБОРУ ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ ТА ВИМОГИ ДО АПАРАТНОГО ЗАБЕЗПЕЧЕННЯ.....	12
Висновки до першого розділу	17
РОЗДІЛ 2. ПРОЕКТУВАННЯ ГРИ-СИМУЛЯТОРА ЗНЕШКОДЖЕННЯ РУХОМИХ ОБ'ЄКТІВ ЗІ СКЛАДНИМИ ТРАЄКТОРІЯМИ РУХУ У 3D ПРОСТОРІ.....	18
2.1 Визначення варіантів використання та об'єктно-орієнтованої структури системи.	18
2.2 ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ АЛГОРИТМІВ РОБОТИ СИСТЕМИ.....	28
2.3 РЕАЛІЗАЦІЯ ФУНКЦІОНАЛУ СИСТЕМИ ГРИ.....	34
Висновки до другого розділу	42
РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З СИСТЕМОЮ СИМУЛЯЦІЇ	43
3.1 Порядок встановлення та налаштування системи	43
3.2 Структура інтерфейсу ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З СИСТЕМОЮ СИМУЛЯЦІЇ.....	43
3.3 ТЕСТУВАННЯ РОБОТИ ПРОГРАМНОГО КОМПЛЕКСУ	46
Висновки до третього розділу	47
ВИСНОВКИ	48
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	50
ДОДАТКИ.....	52

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ППО – Протиповітряна оборона.

GUI – Graphic User Interface – графічний інтерфейс користувача.

ОС – операційна система.

					ІПЗ.КР.Б-121-23-ПЗ	
						5

ВСТУП

Актуальність теми. На сьогоднішній день доступні різноманітні ігри-симулятори на ринку, проте для гравців важливим є не тільки загальний контекст, але й реалістичність відчуття предметів, процесів та подій, які відтворюються у доступному форматі. Для досягнення цього, рухомі об'єкти зі складними траєкторіями руху у 3D просторі відіграють важливу роль.

Вони додають до гри реалістичність та динамічність, сприяючи зануренню гравця в ігрове середовище. Вони створюють відчуття активності та реального впливу на події, що робить гру більш захоплюючою та викликає емоційну реакцію.

Крім того, треба враховувати вплив сучасного світу, де споживачі прагнуть бути частиною важливих подій. У грі, де можна знешкоджувати ворожі рухомі об'єкти, гравець отримує можливість відчути себе активним учасником подій. Це стає особливо актуальним, оскільки вона задовольняє потребу споживача бути важливим гравцем у ситуаціях, де вирішуються важливі завдання.

Таким чином, гра-симулятор з рухомими об'єктами складними траєкторіями руху у 3D просторі має потужний потенціал забезпечити гравцям реалістичний досвід та задовольнити їхню потребу бути частиною важливих подій. Вона створює можливості для активної взаємодії та динамічного розвитку ситуацій, що робить її актуальною та привабливою для широкої аудиторії.

Метою випускної кваліфікаційної роботи бакалавра є проектування та розробка гри-симулятора на рушії Unity3D, яка вимагає весь комплекс знань для її реалізації.

Встановлена мета обумовлює наступні завдання:

- проведення аналізу задач симуляції: дослідження та розуміння основних аспектів руху об'єктів у тривимірному просторі;
- визначення архітектури та узагальненої структури системи;
- обґрунтування та вибір засобів реалізації системи;

					ІПЗ.КР.Б-121-23-ПЗ	
						6

- розроблення моделей та алгоритмів, що описують рух об'єктів, їх поведінку та взаємодію з оточуючим середовищем;
- визначення структури програмного комплексу, розробка алгоритмів управління рухом об'єктів та інших функціональних елементів гри;
- фактична розробка та реалізація гри-симулятора з використанням побудованих моделей, алгоритмів та компонентів.

Об'єктом дослідження є розробка та дослідження ігрової системи, яка забезпечує можливість за допомогою системи ППО збивати літаки зі складними траєкторіями руху у тривимірному просторі.

Предметом дослідження є використання рушія Unity3D для розробки гри-симулятора.

В роботі було використано методи об'єктно-орієнтованого проектування, об'єктно-орієнтованого програмування та дизайну інтерфейсів.

Розроблена система може бути використана для розваги.

РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ДЛЯ СИМУЛЯЦІЇ РОБОТИ ППО

1.1 Постановка задачі

Основною метою є проектування та розробка гри-симулятора на рушії Unity3D в розважальних цілях. Такий симулятор ознайомлює з принципами та методикою роботи систем ППО та набуття навичок управління знешкодженням рухомих об'єктів у 3D просторі. Симулятор може допомогти вдосконалити навички управління та прийняття рішень у складних ситуаціях.

В випускній кваліфікаційній роботі необхідно розробити додаток у вигляді гри-симулятора керування ППО. Для реалізації поставленого завдання, основними етапами є:

1. Розробити дизайн користувацького інтерфейсу:
 - 1.1. створити загальну концепцію дизайну користувацького інтерфейсу;
 - 1.2. Зверстати та застосувати елементи інтерфейсу за допомогою UnityEditor:
 - 1.2.1. головне меню;
 - 1.2.2. панель ігрового користувацького інтерфейсу.
2. Проектування та розробка основних ігрових модулів:
 - 2.1. модуль керування системою ППО;
 - 2.2. модуль перезарядки;
 - 2.3. модуль пострілу і польоту до цілі;
 - 2.4. модуль руху літака;
 - 2.5. модуль анімації системи ППО.
3. Моделювання 3D об'єктів за допомогою Blender3D:
 - 3.1. система ППО;
 - 3.2. ракети;
 - 3.3. літак;
 - 3.4. навколишнє середовище.

4. Накладання текстур на 3D об'єкти:

4.1. створення матеріалів у Blender;

4.2. текстурування, використовуючи Blender.

Результатом реалізації поставленого завдання є ігрова система симуляції роботи ППО.

1.2 Аналіз аналогів програмного продукту

Симулятори - це програмні засоби, що дозволяють імітувати реальні процеси або ситуації у контрольованому середовищі [1]. Вони можуть мати різні форми і використовуються в різних сферах, таких як авіація, автомобільна промисловість, медицина, військовий тренінг, навчання та багато інших. Основна потреба використання симуляторів полягає в наступних аспектах:

- Віртуальні тренажери створюють безпечне та контрольоване середовище для навчання й тренування різних спеціалістів, таких як пілоти, водії, медичний персонал тощо. Вони дозволяють відточувати точність й реагування на різні ситуації.

- Реалістичні віртуальні середовища дозволяють проводити випробування та дослідження без потреби будувати фізичні прототипи або витрачати великі кошти на експерименти. Вони дозволяють вивчати вплив різних факторів, моделювати складні процеси й прогнозувати результати.

- Віртуальні моделі сприяють вдосконаленню й оптимізації різних систем та процесів. Вони дозволяють аналізувати й розв'язувати проблеми, покращувати ефективність, зменшувати ризики й впроваджувати нові рішення.

- Емулятори також використовуються як форма розваг та віртуального досвіду. Вони дозволяють користувачам побачити та відчувати реалістичні ситуації, такі як польоти, гонки, робота з важкими машинами та інше.

Загалом використання симуляторів дозволяє економити час, кошти та ресурси, забезпечувати безпеку та ефективність, а також дозволяє людям отримати практичний досвід та навички в умовах, близьких до реальних.

Узагальнюючи вище сказане можна поділити симулятори на два види:

- навчальні або тренувальні симулятори;
- розважальні (ігрові) симулятори.

У сучасних ігрових симуляторах, подібних до обраної теми, використовуються системи ППО, що надають можливість збивати ракети та інші повітряні цілі. Ці симулятори дозволяють гравцям уявити себе в ролі оператора ППО, який приймає рішення про захист від ворожих ракет або повітряних цілей.

Використання різних контролерів, таких як мишка, клавіатура, джойстики або віртуальна реальність, додає більшу імерсію та реалістичність грі. Такі симулятори дозволяють гравцям відчувати напруження та відповідальність, які супроводжують роботу з системою ППО. Вони можуть покращити навички в прийнятті рішень, реагуванні на ворожі атаки та координації дій в умовах високого тиску. Крім того, ці симулятори можуть бути корисними для навчання військових, які прагнуть покращити свої навички в області протиповітряної оборони.

Прикладом такого симулятора є Steel Beasts(рис.1.1), розроблена компанією eSim Games яка прагне забезпечити якісний геймплей і реалістичне відтворення бойових ситуацій за рахунок вдосконалення фізики поведінки техніки, системи зброї та електроніки, а також розширенням можливостей гри. Важливим аспектом гри є планування та координація дій в рамках протиповітряної оборони. Гравець повинен розставляти свої протиповітряні засоби в оптимальних позиціях, забезпечуючи максимальну покриття території та захист від ворожих нападів. Він також повинен взаємодіяти з іншими членами команди та іншими бойовими одиницями на полі бою для ефективної спільної протидії ворожим атакам.



Рис. 1.1. Steel Beasts

Steel Beasts надає гравцеві можливість відчувати реалістичність протиповітряної оборони, включаючи фізику руху літаків, визначення цілей, розрахунок траєкторій снарядів та наведення вогню. Це дозволяє гравцям отримати практичний досвід управління протиповітряними системами та розвивати свої тактичні навички у сфері оборони від повітряних загроз.

C-RAM Simulator: Air Defense(рис.1.2) є симулятором протиповітряної оборони, який дозволяє гравцям відчувати себе в ролі оператора системи C-RAM (Counter Rocket, Artillery, and Mortar). Ця гра спрямована на моделювання ситуацій, коли необхідно захищати територію від нападу ворожих снарядів, артилерії та ракет.

Гравцям доводиться керувати різними системами протиповітряної оборони, такими як гармати, ракетні комплекси. Вони мають виконувати завдання з виявлення, відстеження та збивання ворожих снарядів, забезпечуючи захист та безпеку важливих об'єктів.

Гра пропонує різноманітні сценарії та рівні складності, що дозволяє гравцям випробувати свої навички в реалістичних умовах.



Рис. 1.2. C-RAM Simulator: Air Defense

Переглянувши та проаналізувавши аналоги, можна із впевненістю сказати, що кожен програмний продукт суттєво відрізняється за призначенням від своїх конкурентів, що створює певну різноманітність та свободу вибору для користувача.

Враховуючи дані по кожній з них, виникає потреба спроектувати та розробити ігрові модулі гри-симулятора ППО створивши відповідний програмний продукт.

Таким чином, основними рисами даного продукту мають бути: сучасний фізичний рушій, наявність зручного GUI, цікавий ігровий світ з різноманітністю ігрових об'єктів.

1.3 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення

Рушій. Unity3D - це потужне інтегроване середовище розробки ігор, яке надає широкий спектр функціональності та інструментів для створення графіки, фізики, штучного інтелекту, анімації та багато іншого (рис.1.3) [2]. Деякі переваги вибору Unity3D для розробки гри-симулятора включають:

- підтримку розробки ігор для різних платформ, таких як ПК, мобільні пристрої, консолі, віртуальна реальність (VR) та інші. Це дозволяє створювати гру, яка буде доступна для широкого кола гравців;

Autodesk Maya(рис.1.4) є однією з найпоширеніших комерційних програм для 3D моделювання та анімації. Вона має велику кількість функцій і підтримує різноманітні формати файлів. Maya широко використовується в індустрії кіно і відеоігор.

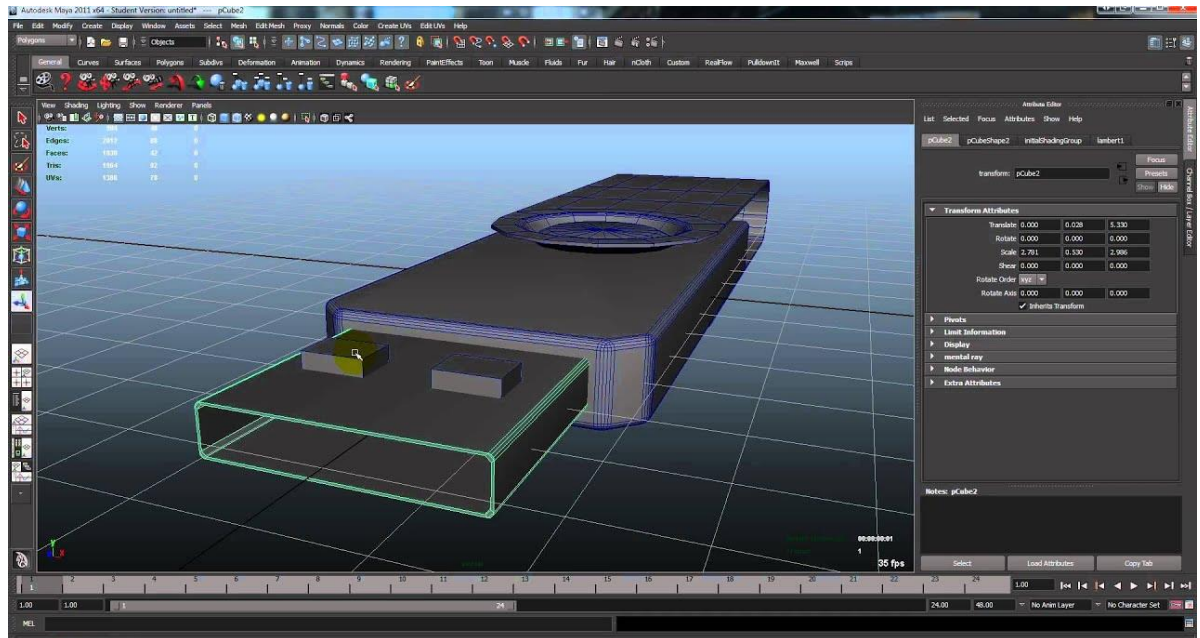


Рис. 1.4. Maya 3D

3ds Max(рис.1.5) є ще одним популярним комерційним програмним забезпеченням для 3D моделювання, анімації та візуалізації. Воно має розширені можливості для моделювання, анімації персонажів, створення віртуальних світів та багато іншого.

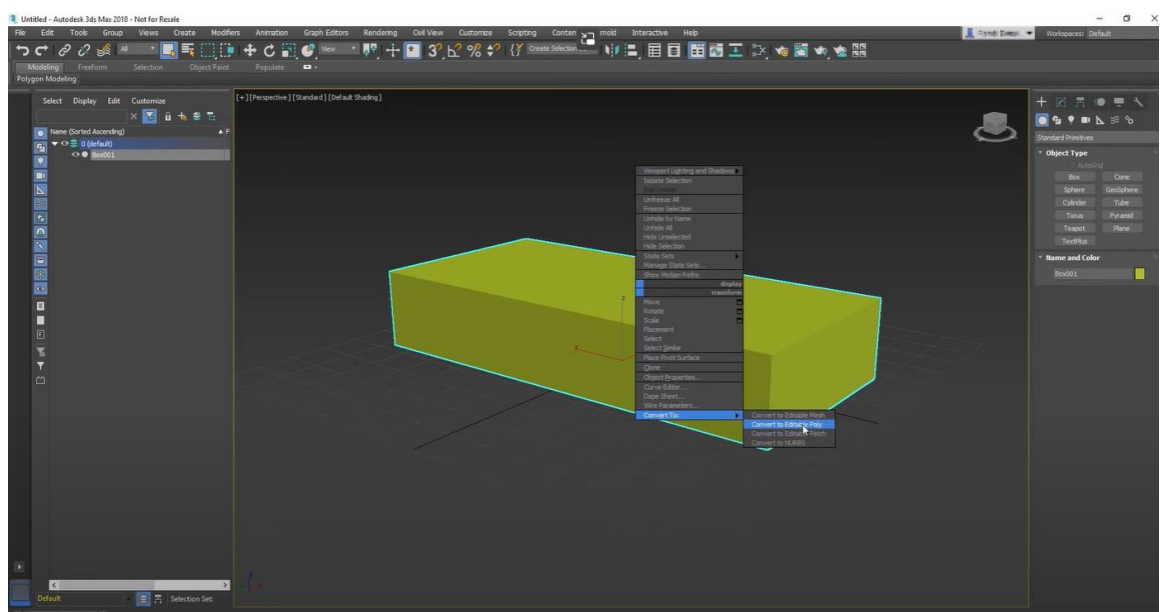


Рис. 1.5. 3ds Max

Але було обране Blender з-за декількох причин:

- Він безкоштовний та відкритий, його можна завантажити та використовувати безкоштовно, що робить його доступним для широкого кола користувачів.
- Редактор має велику та активну спільноту користувачів, що означає наявність багато документації, підручників та онлайн-ресурсів для навчання та підтримки.
- Blender працює на багатьох операційних системах, таких як Windows, macOS і Linux, що дозволяє використовувати його на різних пристроях.
- Blender має набір інструментів для створення 3D об'єктів, формування їх геометрії, роботи з вершинами, ребрами та поверхнями моделей. Це включає інструменти для полігонального моделювання, скульптури, моделювання поверхонь та багато іншого.

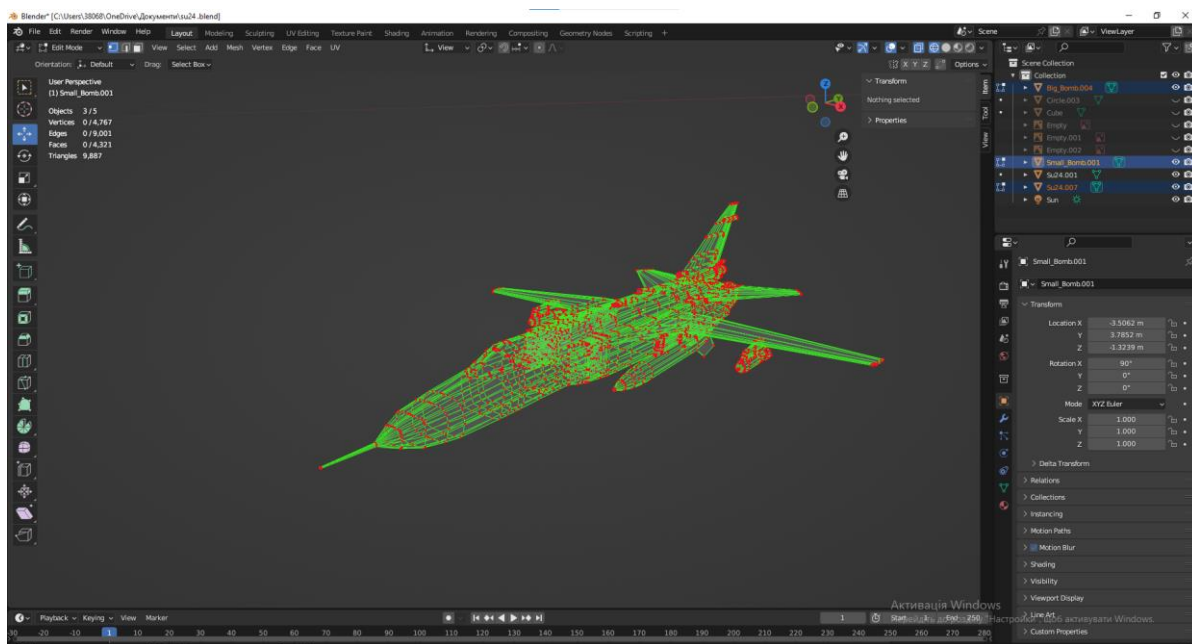


Рис. 1.5. Blender

Редактор дозволяє застосовувати текстури до 3D моделей, створювати матеріали, налаштовувати освітлення та тіні, а також працювати з UV-розгортками для точного нанесення текстур на поверхні моделей. Також надає можливості для створення складних анімаційних сцен, ключових кадрів, розміщення об'єктів у просторі, анімації кісткової системи для персонажів та

використання різних методів інтерполяції для плавного переміщення та трансформації об'єктів.

Вимоги до апаратного забезпечення для Blender можуть варіюватися залежно від розміру та складності проекту, але загалом потрібно мати комп'ютер з достатньою потужністю процесора, відеокарти та оперативної пам'яті для плавної роботи зі складними 3D моделями та анімацією.

Їх порівняльна характеристика програм для 3D моделювання наведена у таблиці 1.1.

Таблиця 1.1

Порівняльна характеристика програм для 3D моделювання

Назва	Вартість	Функціональність	Розширення та плагіни	Використання в індустрії
Blender	Безкоштовна	Має потужний набір інструментів для 3D моделювання, анімації, рендерингу, композитингу та VFX. Може задовольнити потреби багатьох користувачів.	Має велику кількість безкоштовних розширень та плагінів, які розробляються спільнотою.	В різних галузях, включаючи анімацію, відеоігри
Maya 3D	Платна	Має широкий набір функцій для 3D моделювання, анімації, рендерингу, симуляцій, VFX та розробки ігор. Часто використовується в індустрії кіно, відеоігор та архітектурного дизайну.	Має підтримку розширень та плагінів від сторонніх розробників, які часто є платними.	В індустрії кіно, відеоігор, архітектурного дизайну та VFX.
3ds Max	Платна	Має сильний набір інструментів для 3D моделювання, анімації, рендерингу та VFX. Застосовується в галузях, таких як архітектурний дизайн, відеоігри та візуалізація.	Має велику кількість розширень та плагінів від сторонніх розробників	В галузі відеоігор, архітектурного дизайну та візуалізації.

Отже, використано такі програмні продукти та середовища розробки:

- рушій Unity 2021.3.2f1;
- середовище розробки Visual Studio 22;
- 3D-редактор Blender3D 3.41.

Висновки до першого розділу

Проведений аналіз предметної області дозволив визначити основні аспекти симуляції роботи ППО в грі.

Огляд наявних показав, що є головними функціями майбутньої системи має бути: зручність GUI, цікавий ігровий світ і різноманітністю ігрових об'єктів.

Визначившись з оптимальною архітектурою, обрано необхідний та достатній набір програмних продуктів для реалізації поставлених задач: рушій Unity 2021.3.2f1, 3D-редактор Blender3D 3.41.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ГРИ-СИМУЛЯТОРА ЗНЕШКОДЖЕННЯ РУХОМИХ ОБ'ЄКТІВ ЗІ СКЛАДНИМИ ТРАЄКТОРІЯМИ РУХУ У 3D ПРОСТОРІ

2.1 Визначення варіантів використання та об'єктно-орієнтованої структури системи.

Система симуляції роботи ППО створюється з розважальною метою.

Вимоги користувачів.

Користувачі мають доступ до наступних функцій:

1. керування кутом повороту пускової установки;
2. перезарядка установки;
3. постріл і відстеження цілі;
4. пауза, рестарт гри.



Рис. 2.1. Варіанти використання

Функціональні вимоги:

1. Геймплей: користувач матиме можливість керувати пусковою установкою і коригувати траєкторію руху ракети.
2. Користувач може грати в будь-який момент часу.

Нефункціональні вимоги:

1. Сприйняття:
 - Час, потрібний для освоєння симулятора – 2-3 хвилини.
 - Час відповіді системи для звичайних запитів не повин

перевищувати 0.02 секунди.

- Інтерфейс симулятора повинен бути інтуїтивно зручним для користувача.
- Надійність.
- Середній час безперервної роботи — 2 годин.
- Максимальна норма помилок та дефектів в роботі програми - 1 помилка на 1000 запитів користувача.

2. Продуктивність. Гра повинна підтримувати безперебійну роботу користувача.

3. Можливість експлуатації:

- Масштабування - після розширення гри або внесення оновлень, система повинна продовжувати працювати безперебійно, без виникнення помилок.
- Оновлення версій - оновлення гри до нової версії може бути здійснено за допомогою патчу, що дозволяє внести необхідні зміни без необхідності повного переустановлення.

Аналіз функціональних вимог дозволив виділити наступні сутності, що забезпечать реалізацію програмного комплексу системи. На рисунку 2.2 наведено діаграму класів, які використовуються для керування ігровою сценою (світом).

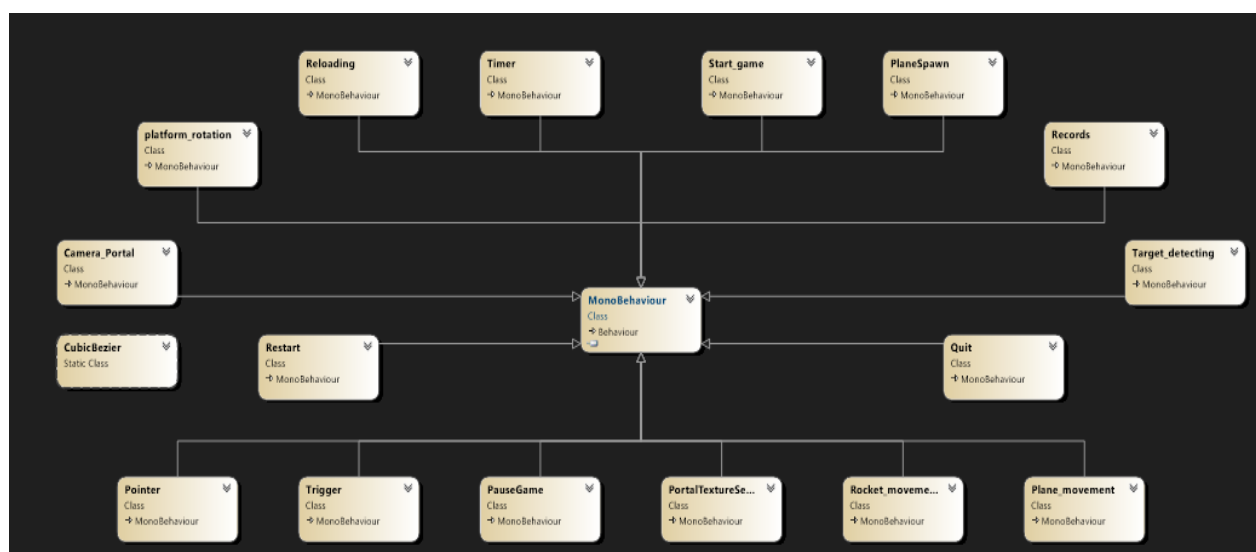


Рис. 2.2. Діаграма класів

Класи в Unity є основними засобами організації коду. Вони дозволяють створювати власні типи даних, методи, властивості та інші функції, які можуть бути використані в програмі. Класи дозволяють створювати об'єкти з визначеними характеристиками і функціональністю. MonoBehaviour є базовим класом, який додає поведінку до об'єктів у сцені Unity. Він містить низку методів, які можуть бути перевизначені у похідних класах, щоб відреагувати на певні події та стан об'єкта[9]. Наприклад, методи Start() та Update() є частими методами, які можна перевизначити для виконання певних дій під час запуску сцени та на кожному кадрі відповідно.

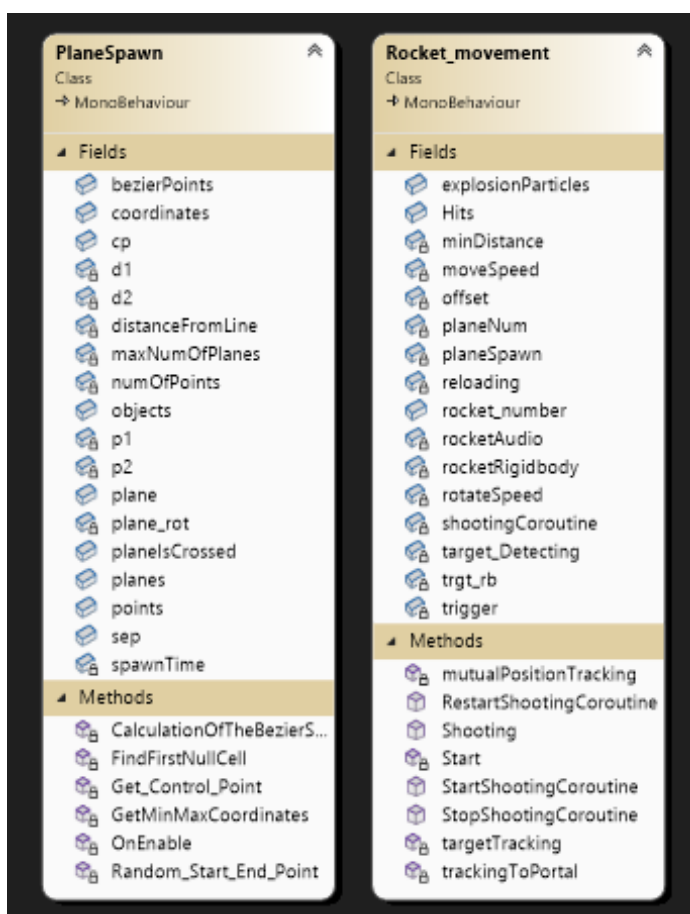


Рис. 2.3. Клас PlaneSpawn та Rocket_movement

Розглянемо клас PlaneSpawn, який спавнить літаки на рис. 2.3[10]:

- bezierPoints – 2D масив типу Vector3. Містить координати які потрібні для візуалізації кривої Безьє.
- Coordinates – 2D масив типу float. Містить максимальні і мінімальні координати по всіх осях об'єктів, всередині яких будуть знаходитись контрольні точки.

- `Cr` – `GameObject`, префаб для візуалізації кривої Безьє.
- `d1`, `d2`, `distanceFromLine` – змінні типу `float`, які виступають параметрами для завдання контрольних точок.
- `maxNumOfPlanes` – змінна типу `int`, що відповідає за максимальну одночасну кількість літаків у сцені.
- `numOfPoints` – змінна типу `int`, що задає максимальну кількість контрольних точок на основі кількості об'єктів з яких будуть обраховуватись початкові та кінцеві точки кривої Безьє.
- `Objects` – масив типу `GameObject`, який містить об'єкти з яких будуть вираховуватись точки для сплайну.
- `p1`, `p2` – `Vector3`, необхідні для обрахунків контрольних точок.
- `plane` – об'єкт типу `GameObject`, що містить префаб літака.
- `Plane_rot` – `Quaternion`, що задає поворот літака.
- `planeIsCrossed` – масив типу `bool`, який потрібний для перевірки того чи літак пройшов крізь портал.
- `planes` – масив типу `GameObject`, що містить в собі всі наявні в сцені літаки.
- `points` – 2D масив типу `Vector3`, що містить координати всі контрольних точок всіх літаків.
- `sep(start/end point)` – об'єкт типу `GameObject`, є префабом для візуалізації кривої Безьє.
- `spawnTime` – змінна типу `float` для задачі часу затримки після спавну літака.
- `OnEnable` – первинна ініціалізація, запускається метод `GetMinMaxCoordinates` та корутина `CalculationOfTheBezierSpline`.
- `GetMinMaxCoordinates` – визначення контрольний точок шляхом отримання інформації про вершини мешу, знаходження і обрахунок їх мінімальних і максимальних координат.
- `Random_Start_End_Point` – рандомний вибір початкових і кінцевих кривої точок з масиву `coordinates`.

– Get_Control_Point – отримання контрольних точок, які задають кривизну кривої.

– FindFirstNullCell – функція знаходження першого null об'єкта в масиві planes.

– CalculationOfTheBezierSpline – корутина, яка виконує Random_Start_End_Point, Get_Control_Point та спавн літаків.

Клас-скрипт Rocket_movement на рис. 2.3 відповідає за рух ракети:

– explosionParticles – тип GameObject, префаб для вибуху.

– Hits – змінна типу int, потрібна для записування кількості попадань.

– minDistance – змінна типу float, мінімальна дистанція, що потрібна для пострілу.

– moveSpeed – змінна типу float, швидкість польоту ракети.

– Offset – змінна типу Vector3, зміщення координат ракети або літака після переходу порталу.

– planeNum – змінна типу int, вказує індекс цілі (літака).

– planeSpawn – змінна типу PlaneSpawn, доступ до відповідного класу.

– reloading – змінна типу Reloading, доступ до відповідного класу.

– rocket_number – змінна типу int, вказує на поточну ракету.

– rocketAudio – змінна типу AudioSource, звук польоту літака.

– rocketRigidbody – змінна типу Rigidbody, rigidbody ракети.

– rotateSpeed – змінна типу float, швидкість повороту ракети.

– shootingCoroutine – змінна типу Coroutine, потрібна для рестарту корутини.

– target_Detecting – змінна типу Target_detecting, доступ до відповідного класу.

– trgt_rb – змінна типу Rigidbody, rigidbody літака.

– Trigger – змінна типу Trigger, доступ до відповідного класу.

- `mutualPositionTracking()` – корутина, що потрібна для відстеження взаємного положення ракети і літака.
- `RestartShootingCoroutine()` – метод, що виконує перезапуск корутини `Shooting()`.
- `Shooting()` – корутина, що відслідковує потребу у перезарядці, і сам процес пострілу.
- `Start()` – первинна ініціалізація.
- `targetTracking()` – корутина що коригує політ до цілі.
- `trackingToPortal()` – корутина що коригує політ ракети до порталу.

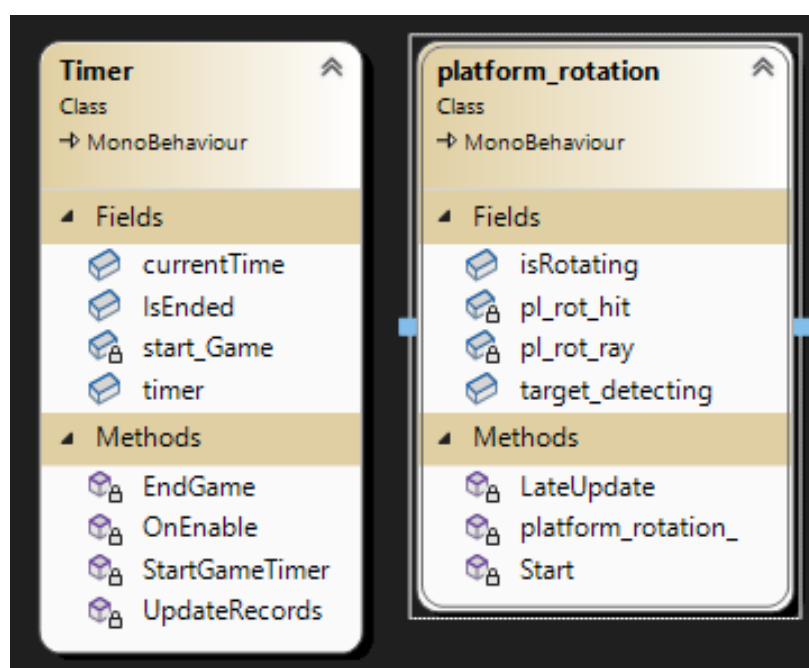


Рис. 2.4. Клас Timer та platform_rotation

Клас `Timer` (див. рис. 2.4) відповідає за відлік часу та запису результатів гри в файл:

- `currentTime` – змінна типу `float`, вказує на поточний час
- `IsEnded` – змінна типу `bool`, повідомляє чи завершилась гра.
- `start_Game` – змінна типу `Start_game`, потрібно для доступу до змінної в яку записується поточний час.
- `EndGame()` – метод, що відповідає за перевірку рекордів і завершення гри.
- `OnEnable()` – первинна ініціалізація.
- `StartGameTimer()` – корутина, що змінює час.

- UpdateRecords() – метод, що записує значення рекордів в текстовий файл.

Клас platform_rotation (див. рис. 2.4) відповідає за поворот установки:

- isRotating – змінна типу bool, вказує чи платформа може обертатись.
- pl_rot_hit – змінна типу RaycastHit, промінь випущений з платформи.
- pl_rot_ray – змінна типу Ray, промінь.
- target_detecting – змінна типу Target_detecting.
- LateUpdate() – метод, що відповідає за перевірку того чи гра на паузі.
- platform_rotation_() – метод, що відповідає за поворот платформи
- Start() – первинна ініціалізація.

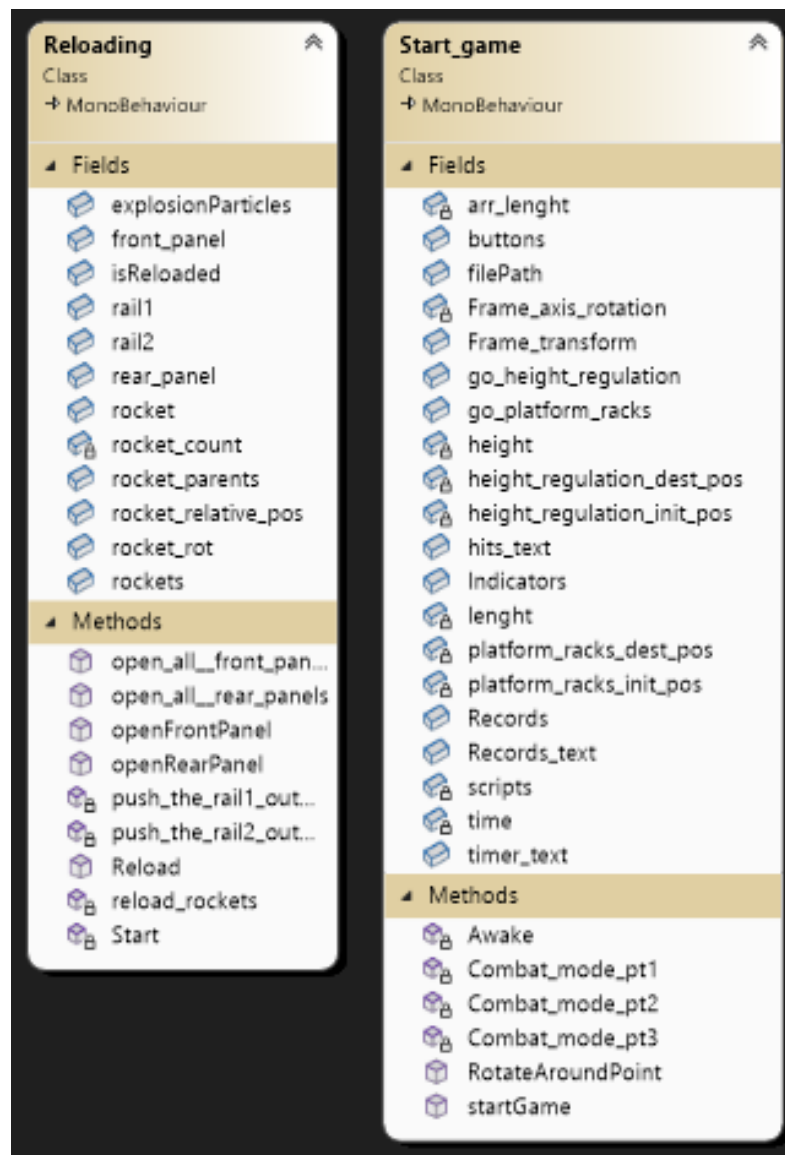


Рис. 2.5. Клас Reloading та Start_game

Клас Reloading (див. рис. 2.5) відповідає за перезарядку пускової установки:

- Rocket – змінна типу GameObject, префаб ракети.
- Rockets – масив типу GameObject, містить ракети.
- isReloaded – змінна типу bool, показує необхідність перезарядки.
- rocket_count – змінна типу int, вказує максимально допустиму кількість ракет.
- rocket_rot – змінна типу Quaternion, поворот ракети під час її створення.
- rocket_relative_pos – змінна типу Vector3, положення ракети.
- rocket_parents – масив типу GameObject, батьківські об'єкти для ракети.
- front_panel – масив типу GameObject, передня панель тубусу.
- rear_panel – масив типу GameObject, задня панель тубусу.
- rail1 – масив типу GameObject, головна рейка.
- rail2 – масив типу GameObject, другорядна рейка.
- explosionParticles – змінна типу GameObject, об'єкт який має Particle System.
- open_all__front_panels(), open_all__rear_panels – корутини які керують всіма панелями.
- openFrontPanel(), openRearPanel() – корутини які керують конкретною панеллю.
- push_the_rail1_outward(), push_the_rail2_outward() – корутини які керують рухом рейок.
- Reload() – головна корутина, запускає по черзі виконання вище перелічених корутин.
- reload_rockets() – корутина яка створює ракети.
- Start() – первинна ініціалізація.

Клас-скрипт Start_game (див. рис. 2.5) відповідає в основному за анімацію переходу установки в бойовий режим:

- arr_lenght – змінна типу int, зберігає довжину масиву об’єктів-опор установки.
- Buttons – об’єкт типу GameObject, зберігає canvas в якому знаходяться кнопки меню.
- filePath – змінна типу string, зберігає шлях до файлу з рекордами.
- Frame_axis_rotation – Vector3, вісі обертання.
- Frame_transform – об’єкт типу Transform, зберігає об’єкт на якому знаходяться тубуси.
- go_height_regulation, go_platform_racks – масив типу GameObject, зберігає об’єкти, що складають опору установки.
- Lenght, height – змінні типу float, вказують наскільки мають висунутись і піднятись опори.
- height_regulation_dest_pos, height_regulation_init_pos, platform_racks_dest_pos, platform_racks_init_pos – масив типу Vector3, зберігає інформацію про початкову і кінцеву позицію опор установки.
- hits_text – змінна типу TMP_Text, зберігає інформацію про кількість влучень.
- Indicators – об’єкт типу GameObject, зберігає canvas в якому знаходяться ігрові індикатори.
- Records – масив типу int, зберігає інформацію про рекорди.
- Records_text – масив типу TMP_Text, зберігає інформацію про рекорди.
- Scripts – масив типу MonoBehaviour, зберігає всі масиви.
- Time – змінна типу float, що визначає довжину анімації.
- timer_text – змінна типу TMP_Text, зберігає показника таймера.
- Програмний код класу-скрипта Reloading та Start_game наведено у додатку А та Б.

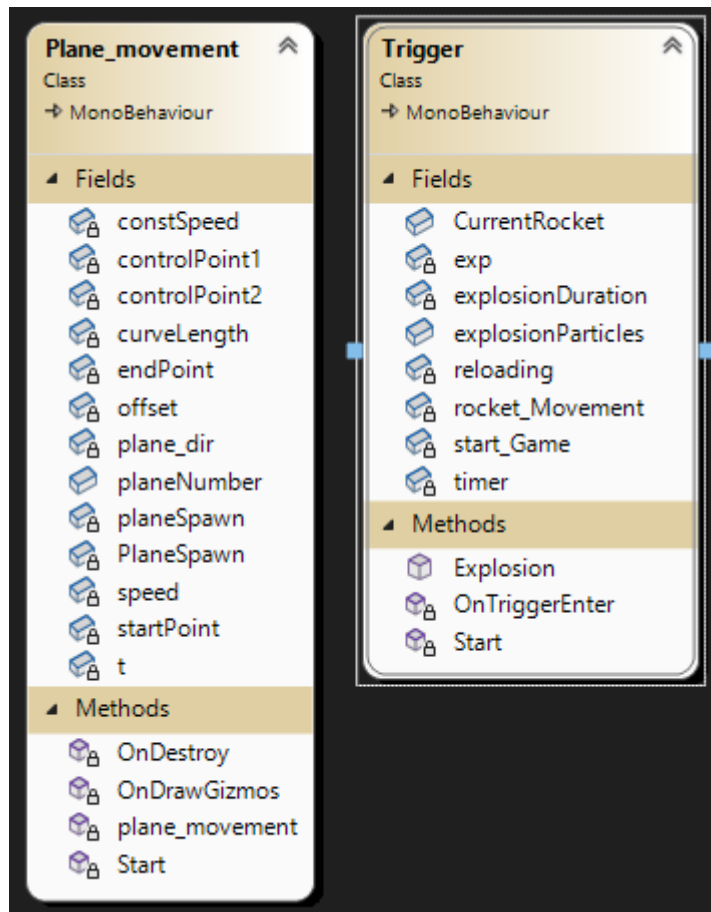


Рис. 2.6. Клас Plane_movement та клас Trigger

Клас Plane_movement (див. рис. 2.6) відповідає за рух літака:

- constSpeed – змінна типу float, визначає швидкість літака.
- controlPoint1 – змінна типу Vector3, перша контрольна точка кривої.
- controlPoint2 – змінна типу Vector3, друга контрольна точка кривої.
- curveLength – змінна типу float, визначає довжину між кінцевою і початковою точкою.
- endPoint – змінна типу Vector3, кінцева точка кривої.
- startPoint – змінна типу Vector3, початкова точка кривої.
- offset – змінна типу Vector3, зміщення по осі Z.
- plane_dir – змінна типу Vector3, записується дотична до кривої.
- planeNumber – змінна типу int, номер літака
- planeSpawn – змінна типу PlaneSpawn, доступ до відповідного класу.
- Speed – змінна типу float, визначає величину на яку буде збільшуватись змінна t після кожної ітерації.

- T – змінна типу float, параметр для знаходження точок на кривій.
- OnDestroy() – метод, що під час знищення літака знищує контрольні точки, які були створені для візуалізації кривої.
- OnDrawGizmos() – метод, що відповідає за візуалізацію сплайну.
- plane_movement() – корутина, що відповідає за рух літака.
- Start() – первинна ініціалізація.

Клас Trigger (див. рис. 2.6) відповідає за :

- CurrentRocket – змінна типу int, номер поточної ракети.
- Exp – об'єкт типу GameObject, створений на основі explosionParticles.
- explosionDuration – змінна типу float, визначає час тривалості вибуху.
- explosionParticles – об'єкт типу GameObject, містить об'єкт з Particle System.
- reloading – змінна типу Reloading, для отримання положення поточної ракети, щоб у цій точці створити об'єкти який має Particle System вибуху.
- rocket_Movement – змінна типу rocket_movement, необхідна для доступу до методу RestartShootingCoroutine().
- start_Game – змінна типу start_game, необхідна для доступу до тексту, що показує кількість попадань.
- timer – змінна типу Timer, необхідна для того щоб додати 5 секунд до таймеру після влучання цілі.
- Explosion() – корутина, що відповідає за ефект вибуху.
- OnTriggerEnter() – метод, що відслідковує, зіткнення об'єктів.
- Start() – первинна ініціалізація.

2.2 Проектування та реалізація алгоритмів роботи системи

Алгоритми які покладені в основу гри є:

- постріл і відстеження ракети;
- рух літака.

Реалізація першого алгоритму включає в себе скрипти, які:

					ІПЗ.КР.Б-121-23-ПЗ	28

- відстежують чи потрібна перезарядка;
- відстежують клік по цілі і коригують рух до неї;
- відстежують зіткнення ракети з літаком.

Принцип дії алгоритму проілюстровано на рисунку 2.7. Запускається корутина Shooting в безкінечному циклі [11,12]. Перш за все перевіряється чи гра на паузі, якщо так то корутина призупиняється, далі перевіряється умова завершення гри, тобто якщо умова виконується то видаляються всі ракети. Наступний код перевіряє чи потрібно проводити перезарядку. У випадку якщо ракет не залишилось виконується корутина Reload. Якщо ж перезарядка вже виконана або є доступні ракети виконується блок коду, який відповідає за постріл ракетою. У цьому блоці визначається де розташована ракета та літак відносно порталу. Якщо під час кліку на літак він знаходиться по одній стороні порталу, тобто вже пройшов крізь нього і відстань між ними більша за мінімальну то запускається корутина targetTracking(). В іншому випадку, тобто коли ракета та літак знаходяться по різні сторони порталу то запускається корутина mutualPositionTracking(). Після виконання відповідних блоків коду корутина Shooting призупиняється на наступну ітерацію що дозволяє іншим процесам виконуватись.

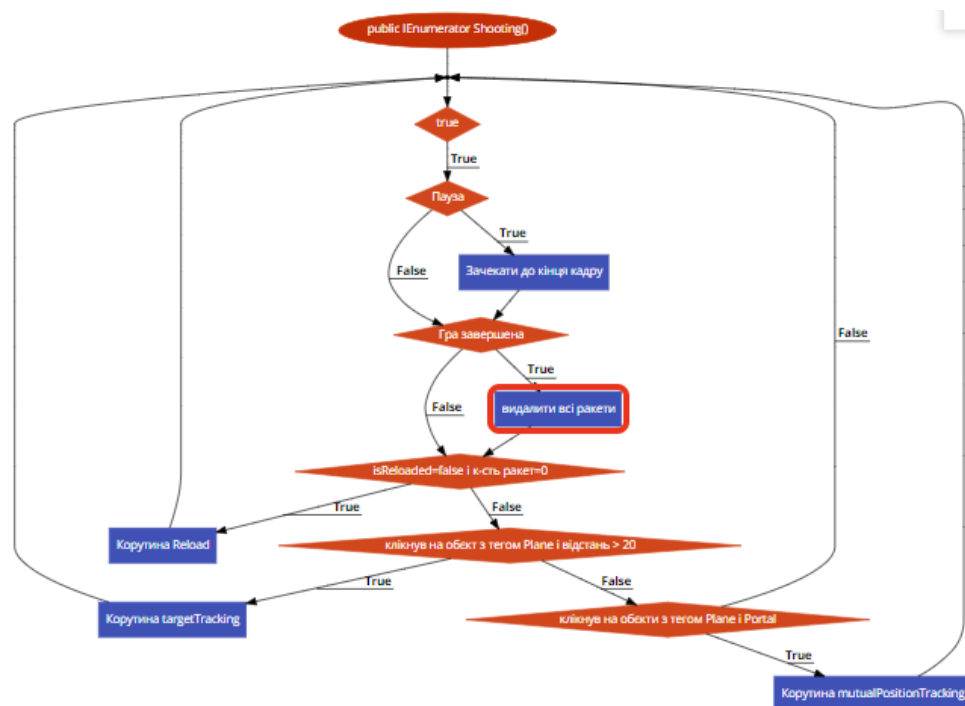


Рис. 2.7. Схема виконання корутини Shooting.

Для розуміння принципу дії потрібно розглянути принцип роботи корутин `targetTracking()`, `trackingToPortal()`, `mutualPositionTracking()` та методу `onTriggerEnter`.

Корутина `mutualPositionTracking()` потрібна для визначення взаємного положення літака та ракети, принцип її роботи наведений на рисунку 2.8.

Спершу запускається корутина `trackingToPortal()` яка задає напрямок ракети до порталу, якщо під час виконання корутини літак пройшов через портал дія даної корутини зупиняється та починає діяти корутина `targetTracking()` яка задає напрям руху до цілі, завершення даної корутини зупиняє дії зовнішньої корутини `mutualPositionTracking()`. Якщо `trackingToPortal()` виконалась, тобто ракета пройшла крізь портал, починає виконуватись корутина `targetTracking()` яка вже коригує політ ракети до цілі в локації їх спавну. Може статись так що ракета і літак розминулись, о попередня корутина зупиняється і ракета починає рух назад до порталу за допомогою корутини `trackingToPortal()` і після її виконання ракета продовжує рух до цілі за допомогою `targetTracking()`.

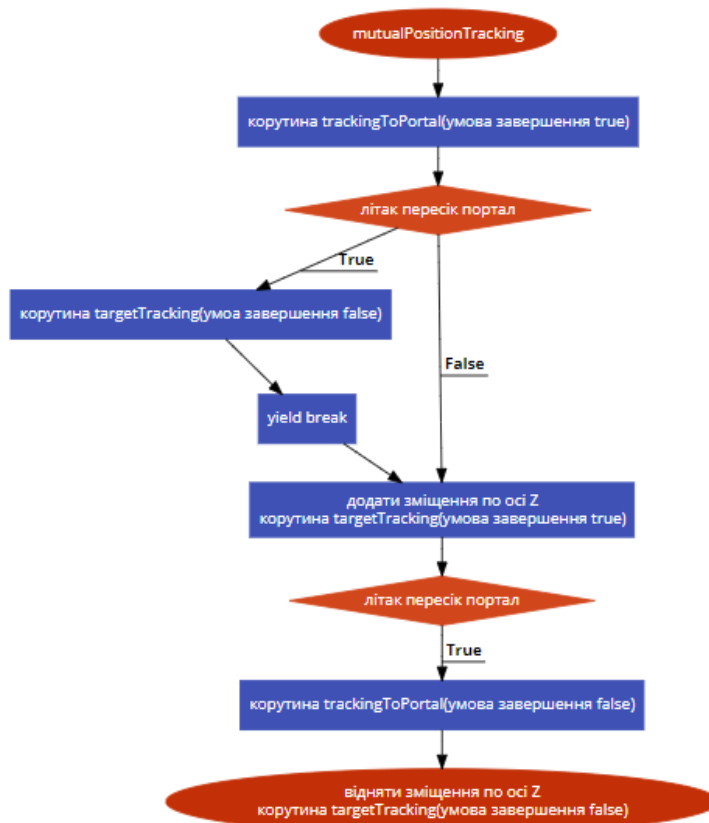


Рис. 2.8. Схема виконання корутини `mutualPositionTracking`.

Корутина `targetTracking()` (рис 2.9) потрібна для коригування траєкторії ракети до цілі. Вона являє собою цикл `while`, який діє то тих пір поки ракета і ціль існує. Тіло циклу спочатку перевіряє чи промінь випущений з точки де курсор потрапляє у ціль, у випадку виконання умови ракета рухається до цілі. В той момент коли курсор на ціль не наведений, з ракети по локальній осі Z випускається промінь, точка попадання якого є точкою до якої буде рухатись ракета, це потрібно для більш правдоподібної та логічної поведінки ракети. Наприкінці виконання тіла циклу перевіряється чи реальний час польоту до цілі більший за прогнозований. У випадку якщо так, то ракета вибухає і перезапускається корутина `Shooting()`.

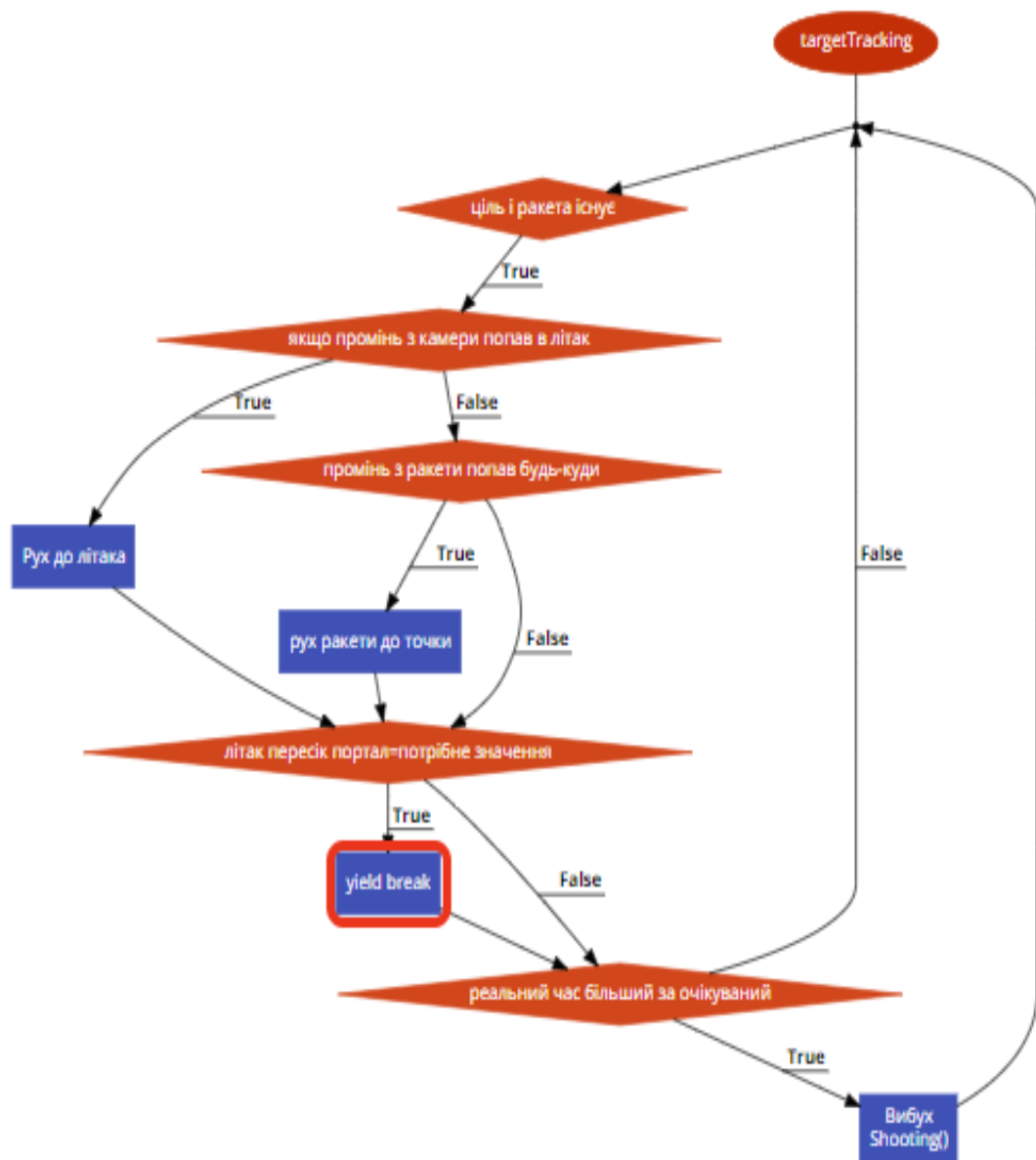


Рис. 2.9. Схема виконання корутини `targetTracking`.

Принцип дії корутини `trackingToPortal()` (рис. 2.10) схожий до `targetTracking()`. Внутрішнє тіло циклу виконується до тих пір поки відстань ракети до порталу більше нуля. Теж відбувається перевірка того, чи курсор наведений на портал. Якщо наведений то відбувається рух до порталу, попередньо перевіривши чи ракета ще не знищена задля уникнення помилок.

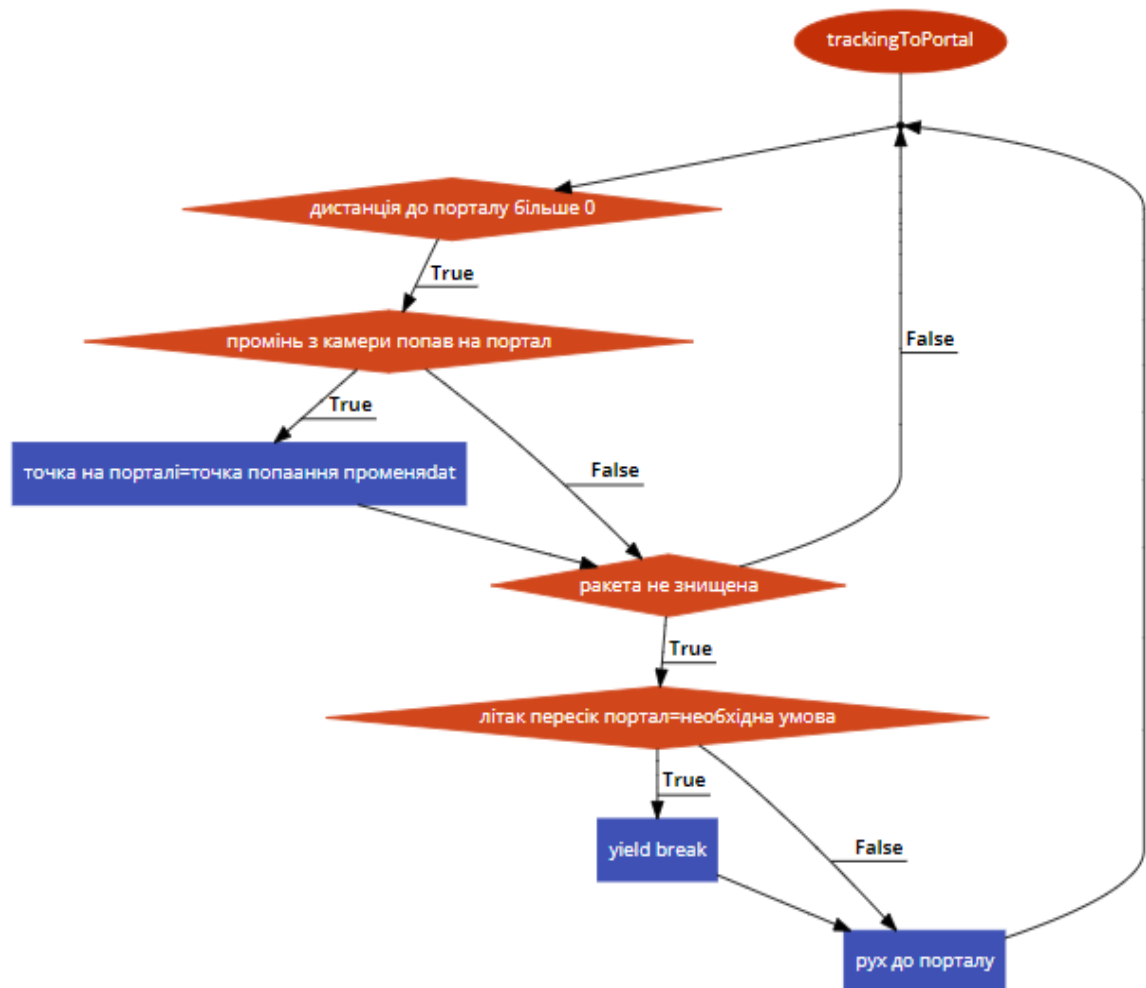


Рис. 2.10. Схема виконання корутини `trackingToPortal`.

До кожної ракети присвоєно скрипт, який відслідковує попадання ракети в об'єкт за допомогою методу `OnTriggerEnter` (рис 2.11)[13]. В даному методі визначається у що влучила ракета. Якщо в об'єкт з тегом "Exp"(об'єкт який не літаком, тобто ракета схибила), то виконується корутина вибуху та перезапускається корутина `Shooting()`. У випадку коли ракета влучила в літак, оновлюється лічильник попадань та додаються додаткові п'ять секунд і відповідно виконуються вище наведені дії.

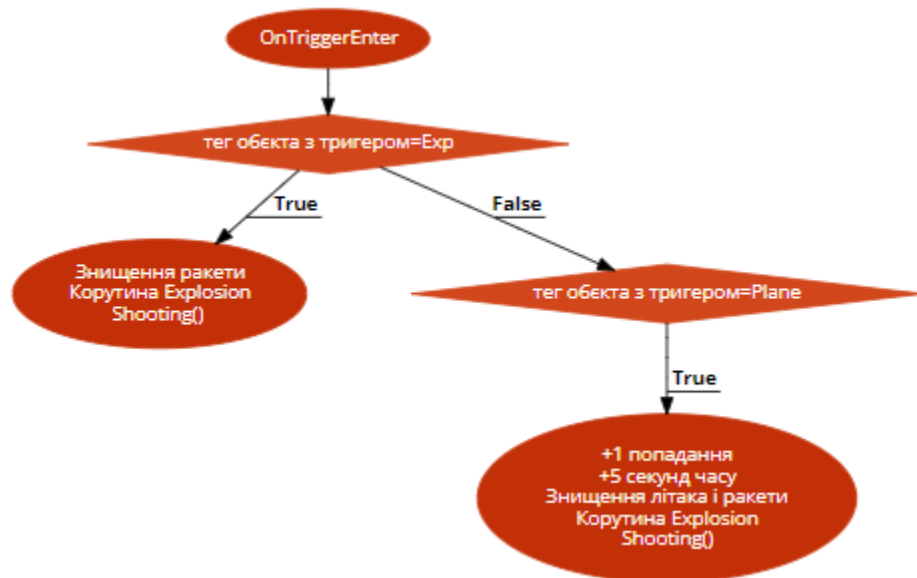


Рис. 2.11. Схема виконання методу OnTriggerEnter

Спавн літаків виконує корутину CalculationOfTheBezierSpline() (рис 2.12).

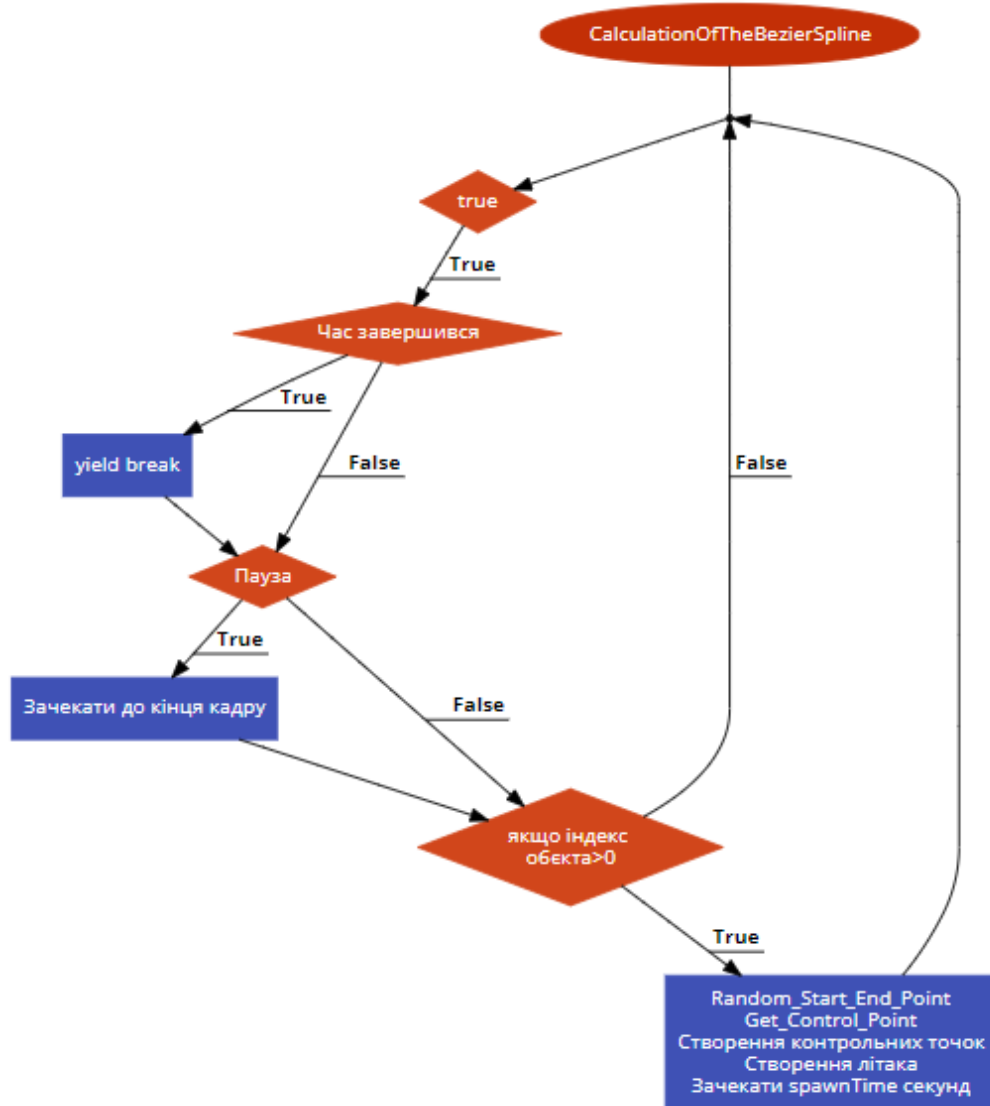


Рис. 2.12. Схема виконання корутини CalculationOfTheBezierSpline

Корутина має нескінченний цикл `while(true)`, який виконується кожні `spawnTime` секунд. Принцип дії такий:

- є масив типу `GameObject`, розміром `maxNumOfPlanes` в якому знаходяться створені літаки;
- масив перевіряється на знаходження першого `null` значення за допомогою ф-ї `FindFirstNullCell()` яка якщо не знайшла `null` значення повертає `-1`, а якщо знайшла то повертає індекс;
- якщо функція повертає не відємне значення то розпочинається виконання методів `Random_Start_End_Point()` та `Get_Control_Point()` які обраховують контрольні точки сплайну;
- після виконання створюється літак до якого приєднується скрипт для руху;

2.3 Реалізація функціоналу системи гри

Система симуляції повинна мати модель поведінки, що балансує між реалістичними аспектами поведінки ігрових об'єктів та свободою дій користувача включно з емоційною складовою.

За керування пусковою установкою відповідають скрипти:

1. `platform_rotation` і `Target_detecting` (керує кутом повороту пускової установки)

Список основних методів скрипту наведено у таблиці 2.1

Таблиця 2.1

Список основних методів

Назва	Опис та призначення
<code>Start()</code>	Первинна ініціалізація. Містить присвоєння значень за замовчуванням
<code>LateUpdate()</code>	Викликається після виклику всіх функцій оновлення.
<code>platform_rotation_()</code>	Викликається в <code>LateUpdate</code> . Містить дії, які потрібні для слідування точки, на яку наведений курсор

Cursor_hit()	Викликається в LateUpdate. Випускає з двох камер промені для подальшого оперування даними, отриманими після попадання в певні об'єкти
--------------	---

2. Start_game(перехід пускової установки в бойове положення)

Використання корутин для простої, плавної анімації є дуже зручним. Корутини є частиною системи керування потоками в Unity і дозволяють виконувати певні частини коду асинхронно. Корутини які використані для анімації мають шаблон зображений на рисунку 2.13.

```

Enumerator Animate()

    startPosition = targetObject.position;
    float timeElapsed = 0.0f;

    while (timeElapsed < duration)
    {
        timeElapsed += Time.deltaTime;
        float t = Mathf.Clamp01(timeElapsed / duration);

        // Застосовуємо ефект анімації
        targetObject.position = Vector3.Lerp(startPosition, targetPosition, t);

        yield return null;
    }

    // Код, який виконується після завершення анімації
    Debug.Log("Анімація завершена!");

```

Рис. 2.13. Шаблон анімації за допомогою корутини

Список методів та корутин скрипту наведено у таблиці 2.2

Таблиця 2.2

Список методів та корутин скрипту

Назва	Опис та призначення
Awake()	Передпервинна ініціалізація значень.
startGame()	Створює масиви з опорними елементами установки і запускає корутини - анімації

platform_rotation_()	Викликається в LateUpdate. Містить дії, які потрібні для слідування точки, на яку наведений курсор
Combat_mode_pt1() Combat_mode_pt2() Combat_mode_pt3()	Висувають та піднімають опорні елементи конструкції та встановлюють кут 30 градусів конструкції що тримає тубуси

3. Reloading(Перезарядка)

Цей скрипт з основною корутиною Reload() являє собою систему послідовних анімацій необхідних для процесу перезарядки.

Список методів та корутин скрипту наведено у таблиці 2.3.

Таблиця 2.3

Список методів та корутин скрипту

Назва	Опис та призначення
Start()	Первинна ініціалізація. Містить присвоєння значень за замовчуванням
Reload()	Корутина яка по черзі запускає корутини для перезарядки
open_all__rear_panels()	Відкриває/закриває на певний кут всі задні панелі тубуса
push_the_rail1_outward()	Висуває/засовує головну рейку
push_the_rail2_outward()	Висуває/засовує другорядну рейку
reload_rockets()	Створення ракет і додавання до них певних компонентів

За відображення ігрових показників(таймера, рекордів та кількості влучань) відповідають скрипти Timer і Trigger(описаний вище).

Список основних методів та корутин скрипту наведено у таблиці 2.4.

Таблиця 2.4

Список методів та корутин скрипту

Назва	Опис та призначення
OnEnable()	Первинна ініціалізація. Старт корутини StartGameTimer()
StartGameTimer()	Корутина яка змінює час що залишився і по завершенню якого запускає метод EndGame();

EndGame()	Метод в якому оновлюється таблиця рекордів
UpdateRecords()	Метод в якому дані з оновленої таблиці рекордів записуються в файл Records.txt

Основою для реалізації руху літаків було обрано кубічну криву Безьє (рис. 2.13)[14]. Крива Безьє - це гладка математична крива, що використовується для опису форм та анімації. Крива Безьє визначається контрольними точками (P1, P2, P3, P4), які визначають її форму та t - параметра кривої, що варіюється від 0 до 1 [15].

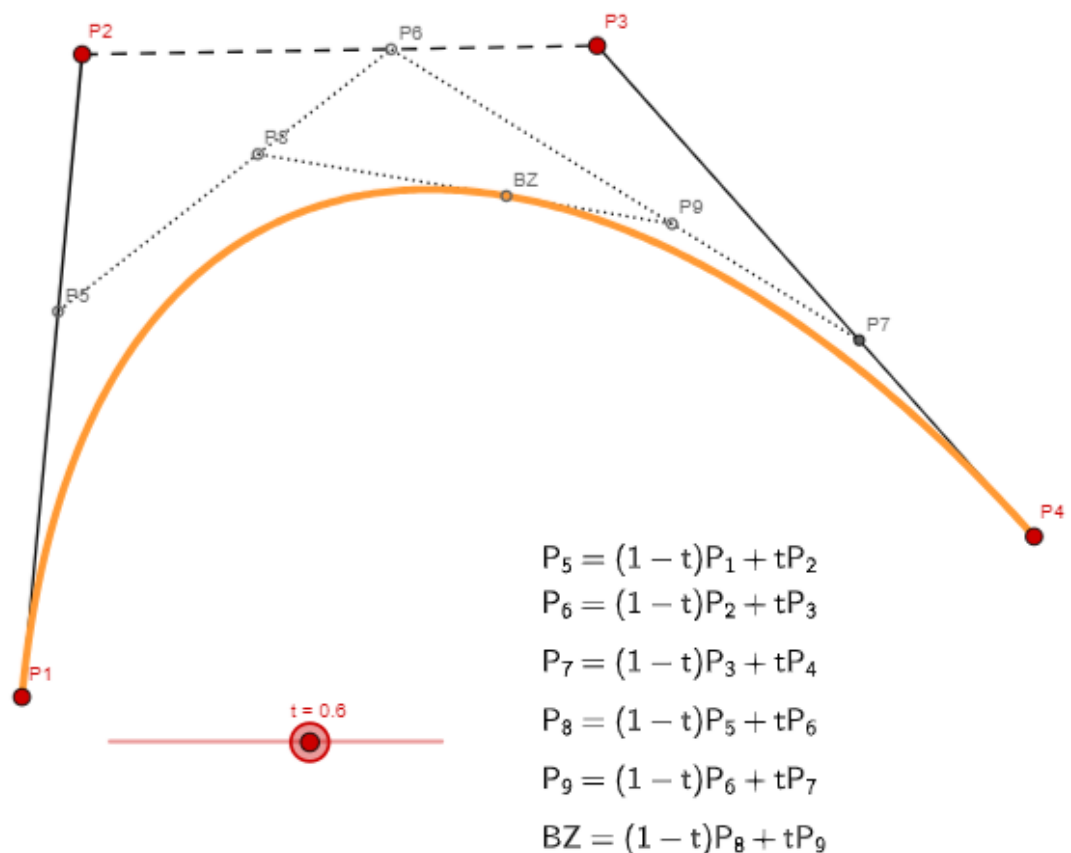


Рис. 2.14. Кубічна крива Безьє

Основний принцип побудови кубічної кривої Безьє полягає у визначенні проміжних точок на кривій шляхом лінійної комбінації контрольних точок з вагами, що змінюються від 0 до 1. Це дозволяє контролювати форму та поведінку кривої. Для знаходження кривої можна скористатись методом Lerp() (рис. 2.15) Функція GetPoint буде знаходитись у статичному класі CubicBezier і

буде повертати Vector3, це дуже зручно адже не потрібно додавати скрипт до об'єкта[16].

```
public static Vector3 GetPoint(Vector3 p0, Vector3 p1, Vector3 p2, Vector3 p3, float t) {
    Vector3 p01 = Vector3.Lerp(p0, p1, t);
    Vector3 p12 = Vector3.Lerp(p1, p2, t);
    Vector3 p23 = Vector3.Lerp(p2, p3, t);

    Vector3 p012 = Vector3.Lerp(p01, p12, t);
    Vector3 p123 = Vector3.Lerp(p12, p23, t);

    Vector3 p0123 = Vector3.Lerp(p012, p123, t);

    return p0123;
}
```

Рис. 2.15. Знаходження точок кривої через Lerp()

Для того щоб код був більш оптимізований, краще скористатись формулою (рис. 2.16).

```
public static Vector3 GetPoint(Vector3 p0, Vector3 p1, Vector3 p2, Vector3 p3, float t)
{
    t = Mathf.Clamp01(t);
    float OneMinusT = 1f - t;
    return
        OneMinusT * OneMinusT * OneMinusT * p0 +
        3f * OneMinusT * OneMinusT * t * p1 +
        3f * OneMinusT * t * t * p2 +
        t * t * t * p3;
}
```

Рис. 2.16. Знаходження точок кривої через формулу

Але цього не достатньо, об'єкт не буде підлаштовуватись до вигинів кривої. Для реалізації цього потрібно знайти дотичну до кривої в точці t. Як відомо дотичною до кривої є перша похідна. Функція GetFirstDerivative буде мати вигляд представлений на рисунку 2.17.

```
2 references
public static Vector3 GetFirstDerivative(Vector3 p0, Vector3 p1, Vector3 p2, Vector3 p3, float t)
{
    t = Mathf.Clamp01(t);
    float OneMinusT = 1f - t;
    return
        3f * OneMinusT * OneMinusT * (p1 - p0) +
        6f * OneMinusT * t * (p2 - p1) +
        3f * t * t * (p3 - p2);
}
```

Рис. 2.17. Знаходження дотичної до кривої

Наступним кроком є урізноманітнення кривої. Це можна зробити такими способами:

1. Криві вищих ступенів.

2. Сплайни.

Використання кривої Безьє вищого ступеня (рис. 2.18) є недоцільним адже для її обрахування потрібно більше потужностей процесора на відміну від сплайну.

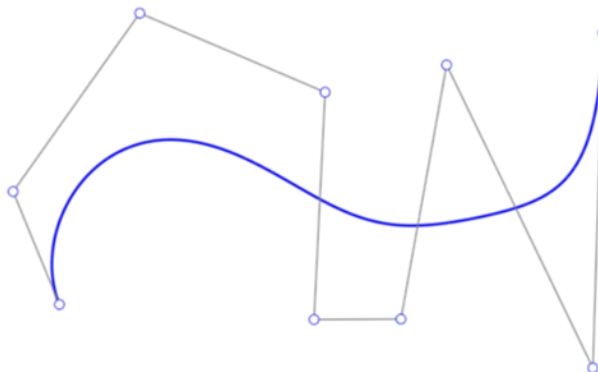


Рис. 2.18. Крива Безьє вищого ступеня

Сплайн - це метод апроксимації кривої, який в даному випадку використовує послідовні сегменти кубічних кривих Безьє для створення плавного переходу між контрольними точками рис 2.19[17]. При використанні сплайну кубічної кривої Безьє, кожна крива між двома сусідніми контрольними точками залежить від своїх власних контрольних точок, але також враховує положення та напрямки кривих з сусідніх сегментів. Це дозволяє створити плавну та неперервну криву, яка проходить через всі контрольні точки.

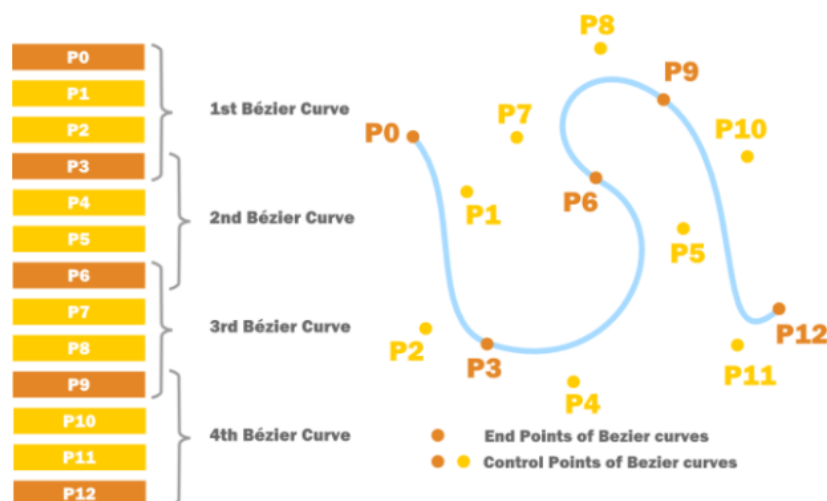


Рис. 2.19. Сплайн

Точки початку/кінця кривої будуть враховуватись у методі GetMinMaxCoordinates() і Random_Start_End_Point() шляхом рандомно обраної точки яка лежить всередині деяких об'єктів (рис 2.20).

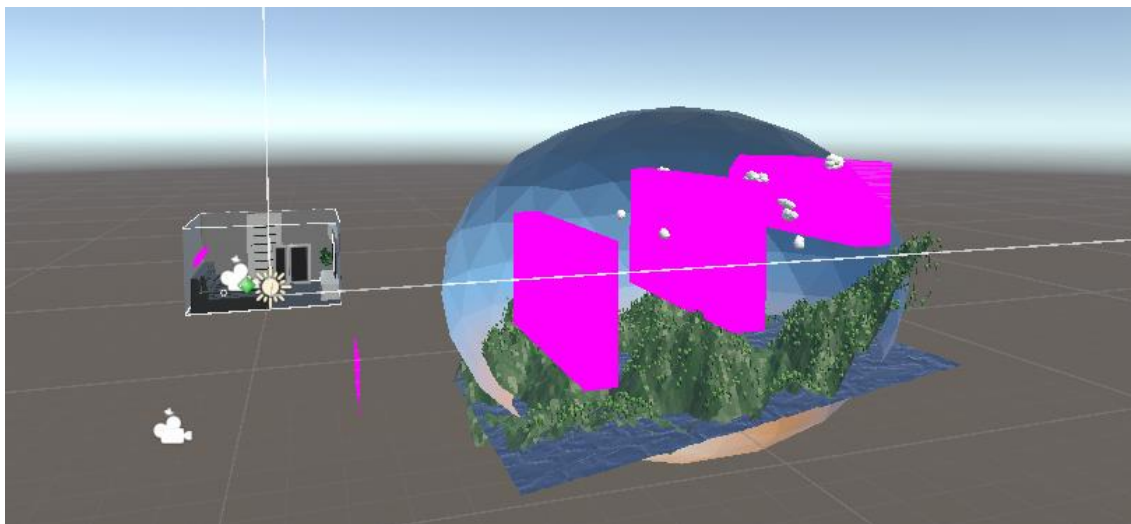


Рис. 2.20. Об'єкти(фіолетовий колір)

Контрольні точки задаються методом `Get_Control_Point()` який враховує те, що контрольні точки мають бути опозитними відносно початкової чи кінцевої точки кривої задля її плавності (рис. 2.21).

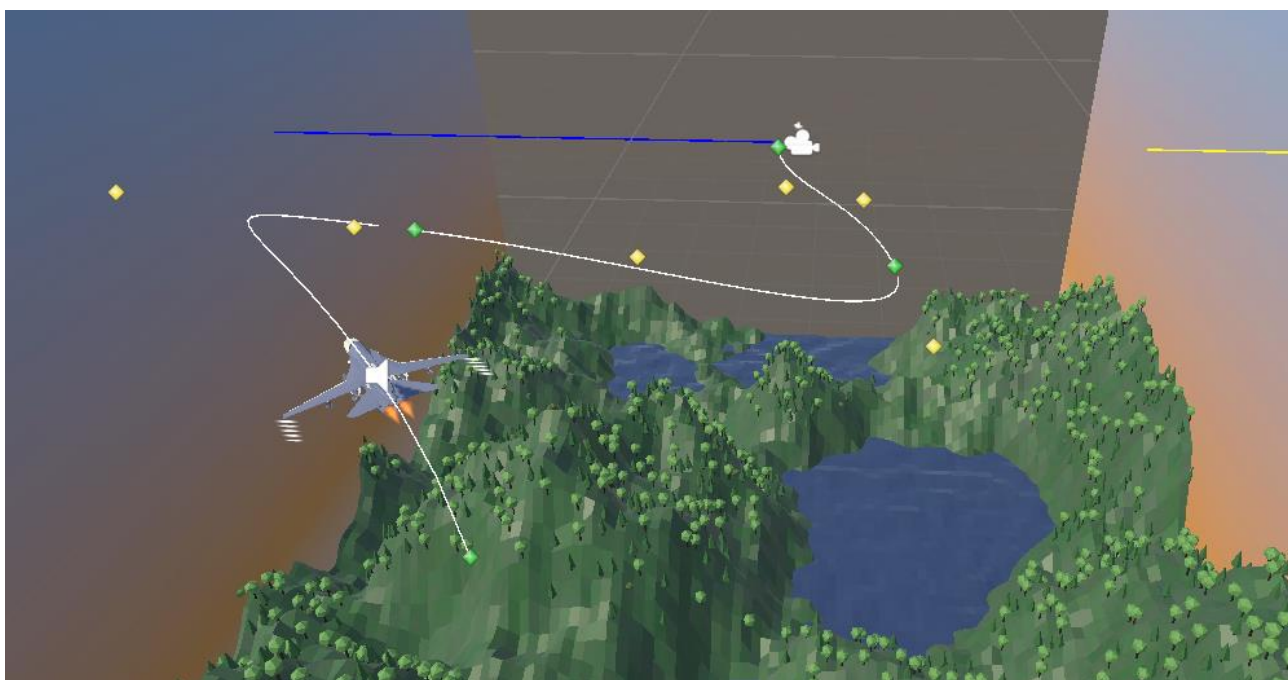


Рис. 2.21. Сплайн

За рух літака відповідає корутина `plane_movement()`[18]. Для того щоб швидкість літака залишалась сталою не залежно від довжини кривої, тобто щоб параметр t масштабувався потрібно : задану швидкість поділити на відстань між початковою і кінцевою точкою кривої (рис. 2.22).

Програмний код, що написано на основі даного алгоритму наведено у додатку В та Г.

```

curveLength = Vector3.Distance(PlaneSpawn.points[pl_num,i], PlaneSpawn.points[pl_num,i + 3]);
speed = constSpeed / curveLength;
t = 0;
while (t < 1)
{
    Vector3 plane_position = CubicBezier.GetPoint(PlaneSpawn.points[pl_num,i], PlaneSpawn.points[pl_num, i + 1], PlaneSpawn.points[pl_num, i + 2], PlaneSpawn.points[pl_num, i + 3], t);
    plane_dir = CubicBezier.GetFirstDerivative(PlaneSpawn.points[pl_num, i], PlaneSpawn.points[pl_num, i + 1], PlaneSpawn.points[pl_num, i + 2], PlaneSpawn.points[pl_num, i + 3], t);
    PlaneSpawn.planes[pl_num].transform.position=plane_position;
    PlaneSpawn.planes[pl_num].transform.rotation=Quaternion.LookRotation(plane_dir);
    t += speed * Time.deltaTime;
    yield return null;
}

```

Рис. 2.22. Код руху літака

Ефект порталу реалізований за допомогою двох камер які мають ідентичний кут повороту і положення з врахуванням зміщення по осі Z, за це відповідає клас `Camera_Portal` та `PortalTextureSetup` (рис. 2.23-2.24)[19]. Останній створює `renderTexture` і накладає її на екран (текстура має те саме розширення що й екран гри), при цьому `ScreenCutoutShader` вирізає лише необхідну частину зображення що дає друга камера (рис 2.25).

```

public class Camera_Portal : MonoBehaviour
{
    public Transform playerCamera;

    // Update is called once per frame
    [Unity Message | 0 references]
    void FixedUpdate()
    {
        Vector3 camera1StartPosition = playerCamera.transform.position;
        Vector3 camera2StartPosition= transform.position;
        Vector3 displacement = playerCamera.transform.position - camera1StartPosition;
        transform.position = camera2StartPosition + displacement;
    }
}

```

Рис. 2.23. Скрипт `Camera_Portal`

```

public class PortalTextureSetup : MonoBehaviour {

    public Camera cameraB;

    public Material cameraMatB;

    // Use this for initialization
    [Unity Message | 0 references]
    void Start () {

        if (cameraB.targetTexture != null)
        {
            cameraB.targetTexture.Release();
        }
        cameraB.targetTexture = new RenderTexture(Screen.width, Screen.height, 24);
        cameraMatB.mainTexture = cameraB.targetTexture;
    }
}

```

Рис. 2.24. Скрипт `PortalTextureSetup`



Рис. 2.25. Портал

Висновки до другого розділу

Система симуляції реалізує функціонал управління системою ППО. Симуляція забезпечується наборами скриптів.

Було розглянуто алгоритми основних процесів системи та проаналізовано взаємодію класів системи під час симуляції.

В останньому пункті наводиться детальний розгляд деяких аспектів реалізації: детально описано принцип спавну та польоту літаків, а також ефекту порталу.

РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З СИСТЕМОЮ СИМУЛЯЦІЇ

3.1 Порядок встановлення та налаштування системи

Система симуляції роботи ППО це додаток який було створено на рушії Unity 3D. Рушій Unity підтримує багато платформ, тому ви зможете запустити симулятор на будь-якому пристрої, який має сумісну конфігурацію з мінімальними вимогами до системи. Наприклад, якщо ви зібрали збірку для Windows, ви зможете комфортно запустити симулятор на будь-якій системі, яка відповідає наступним мінімальним вимогам:

- ОС: Windows 7/8/10
- Процесор: Intel Core i3 або еквівалентний
- Оперативна пам'ять: 4 ГБ
- Графічна карта: сумісна з DirectX 11
- Місце на жорсткому диску: залежить від розміру проекту і його ресурсів

Таким чином, розроблена гра матиме достатню частоту кадрів (до 60 fps) при розширенні екрану FullHD на системах з вище зазначеною конфігурацією.

3.2 Структура інтерфейсу Інтерфейс та порядок роботи з системою симуляції.

Після запуску симулятора, користувач знаходиться в головному меню, де може здійснювати подальші дії (рис 3.1).

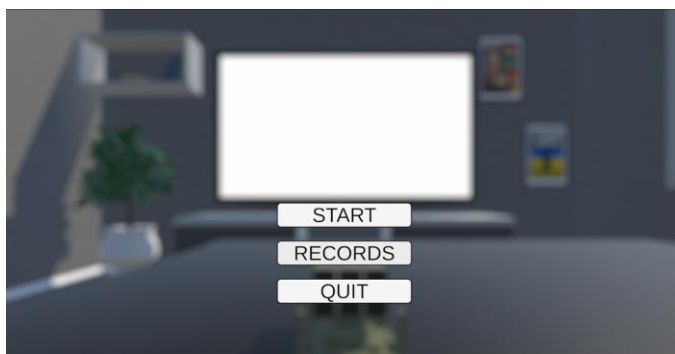


Рис. 3.1. Головне меню

					ІПЗ.КР.Б-121-23-ПЗ	43

Меню складається з трьох елементів керування(рис. 3.2):

1. START – розпочинає гру.
2. RECORDS – перегляд рекордів, повторний клік закриває перегляд (рис 3.2).
3. QUIT – вихід з гри.



Рис. 3.2. Рекорди

Загальне керування. Після того як розпочалась гра користувач отримує доступ до керування пусковою установкою, для керування поворотом досить навести курсор мишки на ціль чи будь-який інший об'єкт (рис 3.3).



Рис. 3.3. Демонстрація повороту установки

Навівши на ціль відкривається тубус з ракетою і ракета летить до цілі вказаної курсором (рис 3.4), курсор має бути постійно наведеним на ціль

інакше ракети продовжить шлях у попередньому напрямку до тих пір поки не вийшов час прогнозованого польоту до цілі. У випадку ураження цілі лічильник попадань збільшується на 1 і додається 5 секунд до часу які відображені у верхній частині екрану (рис 3.5).



Рис. 3.4. Постріл (вид з огляду сцени)



Рис. 3.5. Вибух після попадання

Перезарядка відбувається автоматично, тобто тоді коли закінчились ракети. В будь-який момент гри користувач може поставити гру на паузу або розпочати заново, натиснувши на літеру “P” і “R” відповідно. Після рестарту користувач опиняється в головному меню (рис3.6).

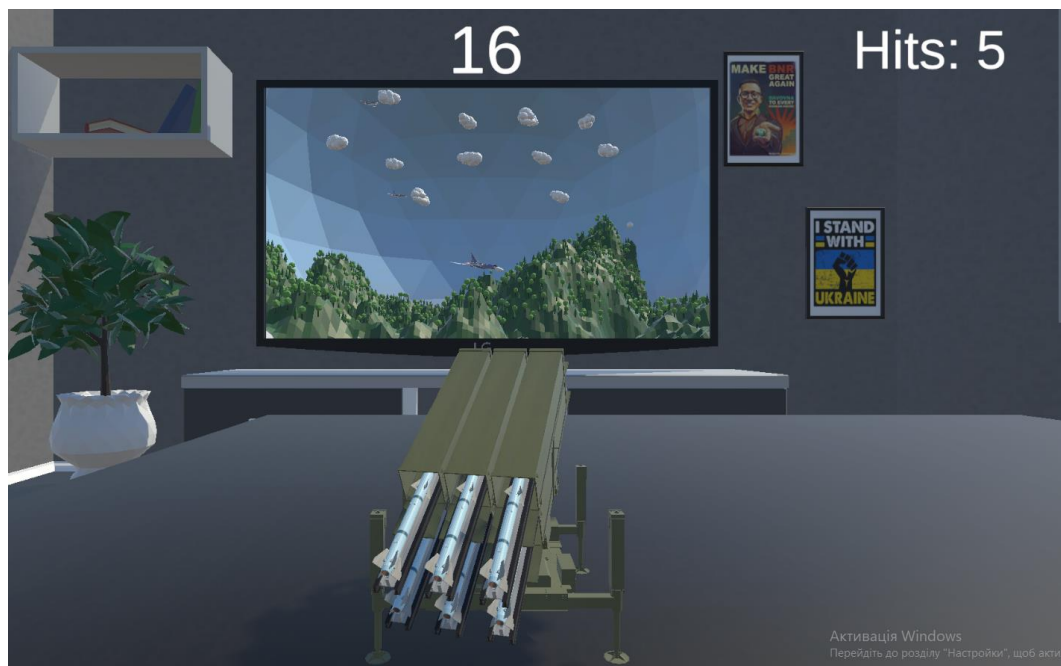


Рис. 3.6. Процес перезарядки (скріншот зроблений під час паузи)

3.3 Тестування роботи програмного комплексу

Тестування є важливою складовою розробки гри або симулятора. Необхідність тестування полягає в тому, щоб перевірити, чи працює система згідно з очікуваннями, чи відповідає вона поставленим вимогам та функціональним вимогам проекту. Тестування також дозволяє ідентифікувати та усунути проблеми, що можуть виникнути в процесі використання системи, щоб забезпечити якісний продукт.

Тестування ігрових систем містить такі основні види:

- користувацьке тестування.
- тестування функціональності;
- тестування геймплею;
- тестування контенту;

Тестування функціональності. Нижче наведено тест-кейси роботи головного меню.

Кнопка «START»:

передумови: гравець повинен знаходитись у головному меню;

кроки: натискання кнопки «START»;

очікуваний результат: розпочинається гра, у випадку якщо гра запускала не один раз не мають існувати в сцені об'єкти які були створенні в попередній грі також повинні відображатись ігрові індикатори;

фактичний результат: відповідає очікуванню.

Кнопка «RECORDS»:

передумови: гравець повинен знаходитись у головному меню;

кроки: натискання кнопки «RECORDS»;

очікуваний результат: у правому верхньому куті мають відобразитись рекорди, після другого натискання на кнопку таблиця з рекордами має зникнути;

фактичний результат: відповідає очікуванню.

Тестування геймплею. Було проведено тестування геймплейних особливостей, включаючи основну - керування пусковою установкою, процесу перезарядки, процес пострілу включно з логікою поведінки ракети при попаданні в ціль та в будь-який інший предмет та її переходу через портал. Процес спавну та руху ракет включно з таймером гри теж перевірено. Під час тестування доступного функціоналу у нього не було виявлено недоліків чи помилок у роботі.

Висновки до третього розділу

В даному розділі було наведено порядок встановлення та налаштування гри-симулятора та визначено мінімальну конфігурацію системи для комфортного геймплею.

Гра має різні інтерфейси, такі як головне меню та основний ігровий інтерфейс, і були наведені розгорнуті деталі щодо роботи та функціональності цих елементів. Було проведено функціональне та геймплейне тестування програми, помилки були виправлені, і система зараз працює належним чином, виконуючи всі поставлені завдання.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи бакалавра було спроектовано та реалізовано Гру-симулятор знешкодження рухомих об'єктів зі складними траєкторіями руху у 3D просторі, для цього було вирішено наступні завдання:

1. Проведений аналіз предметної області дозволив визначити основні аспекти симуляції роботи ППО. Огляд наявних аналогічних додатків та виявлення відмінностей між наявними можливостями та потребами показав, що головними функціями майбутньої системи має бути: зручність GUI, цікавий ігровий світ з різноманітністю ігрових об'єктів. Обрано необхідний та достатній набір програмних продуктів для реалізації поставлених задач: рушій Unity3D, 3D-редактор Blender3d.

2. Виконано аналіз функціональних та не функціональних вимог до гри-симулятора. Для цього була побудована діаграма варіантів використання, яка допомагає визначити основні функції, які повинна виконувати система. На основі цієї діаграми було розроблено діаграму класів, яка допомагає проаналізувати структуру системи та взаємозв'язки між її компонентами.

3. Проведено аналіз алгоритмів, що використовуються в основних процесах системи, а також проаналізовано взаємодію класів системи під час стимуляційних процесів.

4. Було наведено детальний розгляд деяких аспектів реалізації: описано методи головного скрипта-контролера, методи скрипта для контролю струмоприймачів, детально описано процедуру розрахунку точки прив'язки для штанг.

5. Реалізована система симуляції містить інтерфейси головного меню та основний ігровий інтерфейс, робота та функції елементів якого були наведені.

6. Додаток було піддано функціональному та геймплейному тестуванню. Також виявлено помилки, які були виправлені, і зараз система працює бездоганно, виконуючи всі поставлені завдання.

Реалізована гра-симулятор готова до використання. Система може бути використана в розважальних цілях.

В подальшому гра може бути розширена шляхом впровадження нових ігрових механік та додаванням до неї нових ігрових об'єктів таких як різні моделі літаків, систем ППО та локації.

					ІПЗ.КР.Б-121-23-ПЗ	
						49

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Difference Between Games and Simulations [Електронний ресурс] // Difference Between. – 2017. – Режим доступу до ресурсу: <http://www.differencebetween.net/technology/difference-between-games-and-simulations/>.
2. Unity Game Engine [Електронний ресурс] // Unity. – 2023. – Режим доступу до ресурсу: <https://unity.com/>.
3. Thorn A. Mastering Unity Scripting / Alan Thorn. – Birmingham, 2015. – 380 с. – (Packt Publishing).
4. Ferrone H. Mastering Unity Scripting / Harrison Ferrone., 2016. – 379 с.
5. Unity User Manual [Електронний ресурс] // Unity. – 2023. – Режим доступу до ресурсу: <https://docs.unity3d.com/Manual/index.html>.
6. Ferrone H. Learning C# by Developing Games with Unity / Harrison Ferrone., 2019. – 342 с.
7. C# programming guide [Електронний ресурс] // Microsoft. – 2023. – Режим доступу до ресурсу: <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/>.
8. Top 10: The Best 3D Modeling Software [Електронний ресурс] // All3DP. – 2023. – Режим доступу до ресурсу: <https://all3dp.com/1/best-free-3d-modeling-software-3d-cad-3d-design-software/>.
9. MonoBehaviour [Електронний ресурс] // Unity. – 2023. – Режим доступу до ресурсу: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>.
10. Object.Instantiate [Електронний ресурс] // Unity. – 2023. – Режим доступу до ресурсу: <https://docs.unity3d.com/ScriptReference/Object.Instantiate.html>.
11. Carrillo U. Using coroutines in Unity [Електронний ресурс] / Uriel Carrillo // LogRocket. – 2022. – Режим доступу до ресурсу: <https://blog.logrocket.com/using-coroutines-unity/>.

					ІПЗ.КР.Б-121-23-ІЗ	
						50

12. John F. Coroutines in Unity (how and when to use them) [Электронный ресурс] / French John // gamedevbeginner. – 2020. – Режим доступа до ресурсу: <https://gamedevbeginner.com/coroutines-in-unity-when-and-how-to-use-them/>.
13. OnTriggerEnter [Электронный ресурс] // Unity. – 2021. – Режим доступа до ресурсу: <https://docs.unity3d.com/2021.1/Documentation/ScriptReference/Collider.OnTriggerEnter.html>.
14. Tank V. How to work with Bezier Curve in Games with Unity [Электронный ресурс] / Vivek Tank // Game Developer. – 2018. – Режим доступа до ресурсу: <https://www.gamedeveloper.com/business/how-to-work-with-bezier-curve-in-games-with-unity#close-modal>.
15. Kamerman M. A Primer on Bézier Curves [Электронный ресурс] / Mike Kamerman // pomax.github.io. – 2016. – Режим доступа до ресурсу: <https://pomax.github.io/bezierinfo/#molding>.
16. Rizov D. Bezier Curves [Электронный ресурс] / Denis Rizov // Programming Tutorials. – 2016. – Режим доступа до ресурсу: <https://denisrizov.com/2016/06/02/bezier-curves-unity-package-included/>.
17. Cope G. DRAWING A CONTINUOUS BEZIER CURVE [Электронный ресурс] / Greg Cope // Algosome. – 2016. – Режим доступа до ресурсу: <https://www.algosome.com/articles/continuous-bezier-curve-line.html>.
18. Pav. Gentle introduction to shaders and custom materials in Unity [Электронный ресурс] / Pav // Pav Creations. – 2022. – Режим доступа до ресурсу: <https://pavcreations.com/gentle-introduction-to-shaders-and-custom-materials-in-unity/>
19. Export A Game Asset From Blender Into Unity [Электронный ресурс] // Blenderbasecamp. – 2023. – Режим доступа до ресурсу: <https://www.blenderbasecamp.com/home/how-to-export-a-game-asset-from-blender-into-unity/>.
20. Create user interfaces [Электронный ресурс] // Unity. – 2021. – Режим доступа до ресурсу: <https://docs.unity3d.com/Manual/UIToolkits.html>.

ДОДАТКИ

					ІІЗ.КР.Б-121-23-ІІЗ	
						52

```

using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;

public class Reloading : MonoBehaviour
{
    public GameObject rocket;
    public GameObject[] rockets;
    static public bool isReloaded;
    private int rocket_count = 6;
    public Quaternion rocket_rot;
    public Vector3 rocket_relative_pos;
    public Transform[] rocket_parents;
    public Transform[] front_panel;
    public Transform[] rear_panel;
    public Transform[] rail1;
    public Transform[] rail2;
    public GameObject explosionParticles;

    // Start is called before the first frame update
    void Start()
    {
        rockets = new GameObject[rocket_count];
        rocket_relative_pos = new Vector3(-0.00099f, 0f, 0.000459184579f);
        isReloaded = false;
        rocket_rot = Quaternion.Euler(0f, 90f, 90f);
    }

    public IEnumerator Reload()
    {
        {
            if (Timer.IsEnded == true)
            {
                yield break;
            }
            yield return StartCoroutine(open_all__rear_panels(0.5f, -95f));
            yield return StartCoroutine(push_the_rail1_outward(0.7f, -0.004f));
            yield return StartCoroutine(push_the_rail2_outward(0.7f, -0.00354f));
            yield return StartCoroutine(reload_rockets(1f));
            yield return StartCoroutine(push_the_rail2_outward(0.7f, -0.0002934977f));
            yield return StartCoroutine(push_the_rail1_outward(0.7f, 0.0003955555f));
            yield return StartCoroutine(open_all__rear_panels(0.5f, 0));
            isReloaded=true;
            yield return null;
        }
        public IEnumerator open_all__rear_panels(float time, float rot_y)
        {
            float elapsed_time = 0;
            while (elapsed_time < time)
            {
                if (Timer.IsEnded == true)
                {
                    yield break;
                }
                for (int k = 0; k < rocket_count; k++)
                {
                    rear_panel[k].transform.localRotation =
Quaternion.Slerp(rear_panel[k].localRotation, Quaternion.Euler(0f, rot_y, 0f),
(elapsed_time / time));
                }
                elapsed_time += Time.deltaTime;
                yield return null;
            }
            yield break;
        }
    }
}

```

```

}
public IEnumerator open_all__front_panels(float time, float rot_y)
{
    float elapsed_time = 0;
    while (elapsed_time < time)
    {
        if (Timer.IsEnded == true)
        {
            yield break;
        }
        for (int k = 0; k < rocket_count; k++)
        {
            front_panel[k].transform.localRotation =
Quaternion.Slerp(front_panel[k].localRotation, Quaternion.Euler(0f, rot_y, 0f),
(elapsed_time / time));
        }
        elapsed_time += Time.deltaTime;
        yield return null;
    }
    yield break;
}
IEnumerator push_the_rail1_outward(float time, float init_x)
{
    float elapsed_time = 0;
    while (elapsed_time < time)
    {
        if (Timer.IsEnded == true)
        {
            yield break;
        }
        for (int k = 0; k < rocket_count; k++)
        {
            rail1[k].transform.localPosition =
Vector3.Lerp(rail1[k].localPosition, new Vector3(init_x, rail1[k].localPosition.y,
rail1[k].localPosition.z), (elapsed_time / time));
        }
        elapsed_time += Time.deltaTime;
        yield return null;
    }
    yield break;
}
IEnumerator push_the_rail2_outward(float time, float dest_x)
{
    float elapsed_time = 0;
    while (elapsed_time < time)
    {
        if (Timer.IsEnded == true)
        {
            yield break;
        }
        for (int k = 0; k < rocket_count; k++)
        {
            rail2[k].transform.localPosition =
Vector3.Lerp(rail2[k].localPosition, new Vector3(dest_x, rail2[k].localPosition.y,
rail2[k].localPosition.z), (elapsed_time / time));
        }
        elapsed_time += Time.deltaTime;
        yield return null;
    }
    yield break;
}
IEnumerator reload_rockets(float time)
{
    int count = 0;
    while (count < rocket_count)
    {

```

```

        if (Timer.IsEnded == true)
        {
            yield break;
        }
        rockets[count] = Instantiate(rocket, rocket_relative_pos, rocket_rot,
rocket_parents[count]);
        rockets[count].name = "Rocket_" + count + 1.ToString();
        rockets[count].transform.localPosition = rocket_relative_pos;
        rockets[count].transform.localRotation = rocket_rot;
        Trigger trigger= rockets[count].AddComponent<Trigger>();
        trigger.explosionParticles = explosionParticles;
        trigger.CurrentRocket = count;
        count++;
    }
    yield break;
}
public IEnumerator openFrontPanel(float time, float rot_y, int rocket_num)
{
    if (Timer.IsEnded == true)
    {
        yield break;
    }
    float elapsed_time = 0;
    while (elapsed_time < time)
    {
        front_panel[rocket_num].transform.localRotation =
Quaternion.Slerp(front_panel[rocket_num].localRotation, Quaternion.Euler(0f, rot_y,
0f), (elapsed_time / time));
        elapsed_time += Time.deltaTime;
        yield return null;
    }
    yield break;
}
public IEnumerator openRearPanel(float time, float rot_y, int rocket_num)
{
    if (Timer.IsEnded == true)
    {
        yield break;
    }
    float elapsed_time = 0;
    while (elapsed_time < time)
    {
        rear_panel[rocket_num].transform.localRotation =
Quaternion.Slerp(rear_panel[rocket_num].localRotation, Quaternion.Euler(0f, rot_y,
0f), (elapsed_time / time));
        elapsed_time += Time.deltaTime;
        yield return null;
    }
    yield break;
}
}

```

```

using System.Collections;
using System.Collections.Generic;
using TMPPro;
using Unity.VisualScripting;
using UnityEngine;
using System.IO;
using static TMPPro.SpriteAssetUtilities.TexturePacker_JsonArray;

public class Start_game : MonoBehaviour
{
    public GameObject buttons;
    public GameObject Indicators;
    public GameObject[] go_height_regulation;
    public GameObject[] go_platform_racks;
    public Transform Frame_transform;
    private Vector3[] platform_racks_init_pos;
    private Vector3[] platform_racks_dest_pos;
    private Vector3[] height_regulation_init_pos;
    private Vector3[] height_regulation_dest_pos;
    private Vector3 Frame_axis_rotation;
    private float time = 1f;
    private float lenght;
    private float height;
    private int arr_lenght;
    public TMP_Text hits_text;
    public TMP_Text timer_text;
    public TMP_Text[] Records_text;
    public int[] Records;
    public string filePath = "Assets/Records.txt";
    [System.Obsolete]
    MonoBehaviour[] scripts;

    [System.Obsolete]

    private void Awake()
    {
        Records = new int[6];
        if (File.Exists(filePath))
        {
            string[] lines = File.ReadAllLines(filePath);

            for (int i=0;i<lines.Length;i++)
            {
                // Перетворення рядка на int за допомогою int.Parse()
                Records[i] = int.Parse(lines[i]);
            }
        }
        else
        {
            Debug.LogError("Файл не знайдено: " + filePath);
        }
        for (int i = 0; i < Records_text.Length; i++)
        {
            Records_text[i].text = (i + 1).ToString() + ". " + Records[i].ToString();
        }
    }
    public void startGame()
    {
        // Змінюємо активність всіх скриптів
        scripts = FindObjectsOfType<MonoBehaviour>();
        foreach (MonoBehaviour script in scripts)
        {
            script.enabled = false;
        }
    }
}

```

```

// Змінюємо активність всіх скриптів
//rot_pl_pivot_point = new Vector3(-6.7525f, 6.7791f, -1.086f);
buttons.active = false;
Indicators.active = true;
arr_lenght = go_platform_racks.Length;
platform_racks_init_pos = new Vector3[arr_lenght];
platform_racks_dest_pos = new Vector3[arr_lenght];
height_regulation_init_pos = new Vector3[arr_lenght-1];
height_regulation_dest_pos = new Vector3[arr_lenght-1];
lenght = 0.1f;
height = 0.1f;
Frame_axis_rotation = new Vector3(0, 0, 1);
/*-----*/
for (int i = 0; i < arr_lenght-1; i++)
{
    height_regulation_init_pos[i] = new
Vector3(go_height_regulation[i].transform.position.x,
go_height_regulation[i].transform.position.y,
go_height_regulation[i].transform.position.z);
}
for (int j = 0; j < arr_lenght-1; j++)
{
    if (j < 2)
    {
        height_regulation_dest_pos[j] = new
Vector3(go_height_regulation[j].transform.position.x,
go_height_regulation[j].transform.position.y,
go_height_regulation[j].transform.position.z - lenght);

    }
    else
    {
        height_regulation_dest_pos[j] = new
Vector3(go_height_regulation[j].transform.position.x,
go_height_regulation[j].transform.position.y,
go_height_regulation[j].transform.position.z + lenght);

    }
}
/*-----*/
StartCoroutine(Combat_mode_pt1(time));
StopCoroutine(Combat_mode_pt1(time));
}
IEnumerator Combat_mode_pt1(float time)
{
    float elapsed_time = 0;
    while (elapsed_time < time)
    {
        for (int k = 0; k < arr_lenght-1; k++)
        {
            go_height_regulation[k].transform.position =
Vector3.Lerp(height_regulation_init_pos[k], height_regulation_dest_pos[k],
(elapsed_time / time));
        }
        elapsed_time += Time.deltaTime;
        yield return null;
    }
    StartCoroutine(Combat_mode_pt2(time));
}
IEnumerator Combat_mode_pt2(float time)
{
    float elapsed_time = 0;
    for (int i = 0; i < arr_lenght; i++)
    {

```

```

        platform_racks_init_pos[i] = new
Vector3(go_platform_racks[i].transform.position.x,
go_platform_racks[i].transform.position.y, go_platform_racks[i].transform.position.z);
    }
    for (int j = 0; j < arr_lenght; j++)
    {
        platform_racks_dest_pos[j] = new
Vector3(go_platform_racks[j].transform.position.x,
go_platform_racks[j].transform.position.y + height,
go_platform_racks[j].transform.position.z);
    }
    while (elapsed_time < time)
    {
        for (int k = 0; k < arr_lenght; k++)
        {
            go_platform_racks[k].transform.position =
Vector3.Lerp(platform_racks_init_pos[k], platform_racks_dest_pos[k], (elapsed_time /
time));
        }
        elapsed_time += Time.deltaTime;
        yield return null;
    }
    StartCoroutine(Combat_mode_pt3());
}
IEnumerator Combat_mode_pt3()
{
    RotateAroundPoint(Frame_transform, transform.position, Frame_axis_rotation,
30f);
    foreach (MonoBehaviour script in scripts)
    {
        script.enabled = true;
    }
    yield return null;
}
public void RotateAroundPoint(Transform transform, Vector3 pivotpoint, Vector3
axis, float angle)
{
    Quaternion rot=Quaternion.AngleAxis(angle, axis);
    transform.position = rot * (transform.position - pivotpoint) + pivotpoint;
    transform.rotation = rot * transform.rotation;
}
}

```

```

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

public class PlaneSpawn : MonoBehaviour
{
    private int maxNumOfPlanes = 6;
    private float spawnTime = 5f;
    private int numOfPoints;
    public GameObject[] objects; //масив об'єктів в з яких будуть обраховуватись точки
    для сплайну
    public GameObject cp;
    public GameObject sep;
    public float[,] coordinates;
    public Vector3[,] points;
    public GameObject[,] bezierPoints;
    public GameObject[] planes;
    public GameObject plane;
    public bool[] planeIsCrossed;
    Quaternion plane_rot;

    float d1 = 1f / 3f;
    float d2 = 2f / 3f;
    float distanceFromLine = 5f;

    Vector3 p1 = new Vector3();
    Vector3 p2 = new Vector3();

    void OnEnable()
    {
        numOfPoints = (objects.Length - 2) * 3 + 1;
        planes = new GameObject[maxNumOfPlanes];
        planeIsCrossed = new bool[maxNumOfPlanes];
        points = new Vector3[maxNumOfPlanes, numOfPoints];
        bezierPoints = new GameObject[maxNumOfPlanes, numOfPoints];
        coordinates = new float[objects.Length, 6];
        for (int i = 0; i < objects.Length; i++)
        {
            GetMinMaxCoordinates(objects[i], i);
        }
        Debug.Log(Timer.IsEnded);
        StartCoroutine(CalculationOfTheBezierSpline());
    }

    void GetMinMaxCoordinates(GameObject obj, int j)
    {
        // Отримуємо компонент меша з об'єкта
        MeshFilter meshFilter = obj.GetComponent<MeshFilter>();
        if (meshFilter == null)
        {
            Debug.LogError("MeshFilter component not found on object: " + obj.name);
            return;
        }

        // Отримуємо меш
        Mesh mesh = meshFilter.sharedMesh;

        // Отримуємо вершини меша
        Vector3[] vertices = mesh.vertices;

        // Отримуємо трансформ об'єкта
        Transform objTransform = obj.transform;
    }
}

```

```

// Ініціалізуємо змінні для збереження мінімальних та максимальних координат
Vector3 minCoords = objTransform.TransformPoint(vertices[0]);
Vector3 maxCoords = objTransform.TransformPoint(vertices[0]);

// Ітеруємося по кожній вершині меша
for (int i = 1; i < vertices.Length; i++)
{
    Vector3 vertex = objTransform.TransformPoint(vertices[i]);

    //мінімальні та максимальні координати
    minCoords.x = Mathf.Min(minCoords.x, vertex.x);
    minCoords.y = Mathf.Min(minCoords.y, vertex.y);
    minCoords.z = Mathf.Min(minCoords.z, vertex.z);

    maxCoords.x = Mathf.Max(maxCoords.x, vertex.x);
    maxCoords.y = Mathf.Max(maxCoords.y, vertex.y);
    maxCoords.z = Mathf.Max(maxCoords.z, vertex.z);
}
coordinates[j, 0] = minCoords.x;
coordinates[j, 1] = minCoords.y;
coordinates[j, 2] = minCoords.z;
coordinates[j, 3] = maxCoords.x;
coordinates[j, 4] = maxCoords.y;
coordinates[j, 5] = maxCoords.z;
// Виводимо результат у консоль
Debug.Log("Object: " + obj.name);
Debug.Log("Min Coordinates: " + minCoords.x + ", " + minCoords.y + ", " +
minCoords.z);
Debug.Log("Max Coordinates: " + maxCoords.x + ", " + maxCoords.y + ", " +
maxCoords.z);
}
//Знаходження початкових і кінцевих точок
void Random_Start_End_Point(int objectsLenght, int planeNum)
{
    int step = 0;
    for (int i = 0; i < objectsLenght-1; i++)
    {
        points[planeNum,step] = new
Vector3(UnityEngine.Random.Range(coordinates[i, 0], coordinates[i, 3]),
UnityEngine.Random.Range(coordinates[i, 1], coordinates[i, 4]),
UnityEngine.Random.Range(coordinates[i, 2], coordinates[i, 5]));
        step += 3;
    }
}
//Знаходження контрольних точок
void Get_Control_Point(int pointsLenght, int planeNum)
{
    for (int i = 1; i < pointsLenght; i += 3)
    {
        float randomX1 = UnityEngine.Random.Range(0f, 360f);
        float randomY1 = UnityEngine.Random.Range(0f, 360f);
        float randomZ1 = UnityEngine.Random.Range(0f, 360f);
        Vector3 randomRotation1 = new Vector3(randomX1, randomY1,
randomZ1).normalized;
        p1 = points[planeNum, i - 1] + (points[planeNum, i + 2] - points[planeNum,
i - 1]) * d1;
        points[planeNum, i] = p1 + randomRotation1 * distanceFromLine;
        Debug.Log(i);
        Debug.Log(points[planeNum, i]);
    }
    for (int i = 2; i <= 5;)
    {
        points[planeNum, i] = points[planeNum, i + 1] - (points[planeNum, i + 2] -
points[planeNum, i + 1]);
        i = i + 3;
    }
}

```

```

float randomX = UnityEngine.Random.Range(0f, 360f);
float randomY = UnityEngine.Random.Range(0f, 360f);
float randomZ = UnityEngine.Random.Range(0f, 360f);
Vector3 randomRotation = new Vector3(randomX, randomY, randomZ).normalized;
p2 = points[planeNum,6] + (points[planeNum, 9] - points[planeNum, 6]) * d2;
points[planeNum, 8] = p2 - randomRotation * distanceFromLine;
}
int FindFirstNullCell(GameObject[] array)
{
    for (int i = 0; i < array.Length; i++)
    {
        if (array[i] == null)
        {
            return i;
        }
    }
    return -1;
}
IEnumerator CalculationOfTheBezierSpline()
{
    while (true)
    {
        if (Timer.IsEnded == true)
        {
            yield break;
        }
        if (PauseGame.isPaused == true)
        {
            yield return new WaitForEndOfFrame();
        }
        int firstNullCellIndex = FindFirstNullCell(planes);
        if (firstNullCellIndex >= 0)
        {
            Random_Start_End_Point(objects.Length, firstNullCellIndex);
            Get_Control_Point(points.GetLength(1), firstNullCellIndex);
            for(int i = 0; i < numOfPoints;)
            {
                bezierPoints[firstNullCellIndex, i] = Instantiate(sep);
                bezierPoints[firstNullCellIndex, i].transform.position =
points[firstNullCellIndex, i];
                i += 3;
            }
            for (int i = 1; i < numOfPoints;)
            {
                bezierPoints[firstNullCellIndex, i] = Instantiate(cp);
                bezierPoints[firstNullCellIndex, i+1] = Instantiate(cp);
                bezierPoints[firstNullCellIndex, i].transform.position =
points[firstNullCellIndex, i];
                bezierPoints[firstNullCellIndex, i+1].transform.position =
points[firstNullCellIndex, i+1];
                i += 3;
            }
            planes[firstNullCellIndex] = Instantiate(plane,
points[firstNullCellIndex,0], plane_rot);
            planes[firstNullCellIndex].name = "Plane_" +
firstNullCellIndex.ToString();
            planeIsCrossed[firstNullCellIndex] = false;
            Plane_movement plane_Movement =
planes[firstNullCellIndex].AddComponent<Plane_movement>();
            plane_Movement.planeNumber= firstNullCellIndex;
            yield return new WaitForSeconds(spawnTime);
        }
        yield return null;
    }
}
}
}

```

```

using System.Collections;
using System.Collections.Generic;
using System.Net.Sockets;
using UnityEngine;

public class Plane_movement : MonoBehaviour
{
    public int planeNumber;
    private PlaneSpawn PlaneSpawn;
    private PlaneSpawn planeSpawn;
    private float curveLength;
    private float speed;
    private float constSpeed;
    private float t = 0;
    private Vector3 controlPoint1 = new Vector3();
    private Vector3 controlPoint2 = new Vector3();
    private Vector3 endPoint=new Vector3();
    private Vector3 startPoint = new Vector3();
    private Vector3 offset = new Vector3(0, 0, 162.3f);
    private Vector3 plane_dir=new Vector3();
    private void Start()
    {
        PlaneSpawn=FindObjectOfType<PlaneSpawn>();
        planeSpawn = FindObjectOfType<PlaneSpawn>();
        StartCoroutine(plane_movement(planeNumber));
    }
    IEnumerator plane_movement(int pl_num)
    {
        constSpeed = Random.Range(8, 12);
        for (int i = 0; i < PlaneSpawn.points.GetLength(1)-3;)
        {
            if (Timer.IsEnded == true)
            {
                Destroy(PlaneSpawn.planes[pl_num]);
                yield break;
            }
            if (PauseGame.isPaused == true)
            {
                yield return new WaitForEndOfFrame();
            }
            curveLength = Vector3.Distance(PlaneSpawn.points[pl_num,i],
PlaneSpawn.points[pl_num,i + 3]);
            speed = constSpeed / curveLength;
            t = 0;
            while (t < 1)
            {
                Vector3 plane_position =
CubicBezier.GetPoint(PlaneSpawn.points[pl_num,i], PlaneSpawn.points[pl_num, i + 1],
PlaneSpawn.points[pl_num, i + 2], PlaneSpawn.points[pl_num, i + 3], t);
                plane_dir = CubicBezier.GetFirstDerivative(PlaneSpawn.points[pl_num,
i], PlaneSpawn.points[pl_num, i + 1], PlaneSpawn.points[pl_num, i + 2],
PlaneSpawn.points[pl_num, i + 3], t);
                PlaneSpawn.planes[pl_num].transform.position=plane_position;

PlaneSpawn.planes[pl_num].transform.rotation=Quaternion.LookRotation(plane_dir);
                t += speed * Time.deltaTime;
                yield return null;
            }
            i += 3;
        }
        //

        PlaneSpawn.planes[planeNumber].transform.position += offset;
        planeSpawn.planeIsCrossed[planeNumber] = true;
    }
}

```

```

//Крива Безье після порталу
t = 0;
float randomX = UnityEngine.Random.Range(0f, 360f);
float randomY = UnityEngine.Random.Range(0f, 360f);
float randomZ = UnityEngine.Random.Range(0f, 360f);
startPoint = transform.position;
Vector3 randomRotation = new Vector3(randomX, randomY, randomZ).normalized;
endPoint = new Vector3(UnityEngine.Random.Range(PlaneSpawn.coordinates[4, 0],
PlaneSpawn.coordinates[4, 3]), UnityEngine.Random.Range(PlaneSpawn.coordinates[4, 1],
PlaneSpawn.coordinates[4, 4]), UnityEngine.Random.Range(PlaneSpawn.coordinates[4, 2],
PlaneSpawn.coordinates[4, 5]));
curveLength = Vector3.Distance(transform.position, endPoint);
Vector3 c = transform.position - (PlaneSpawn.points[pl_num, 8] + offset);
controlPoint1 = transform.position + c;
Vector3 p2 = PlaneSpawn.planes[pl_num].transform.position + (endPoint -
PlaneSpawn.planes[pl_num].transform.position) * 2f/3f;
controlPoint2 = p2 - randomRotation * 3f;
Debug.Log(transform.position);
Debug.Log(controlPoint1);
Debug.Log(controlPoint2);
Debug.Log(endPoint);

speed = (constSpeed-1) / curveLength;
while (t < 1)
{
    Vector3 plane_position = CubicBezier.GetPoint(startPoint, controlPoint1,
controlPoint2, endPoint, t);
    plane_dir = CubicBezier.GetFirstDerivative(startPoint, controlPoint1,
controlPoint2, endPoint, t);
    PlaneSpawn.planes[pl_num].transform.position = plane_position;
    PlaneSpawn.planes[pl_num].transform.rotation =
Quaternion.LookRotation(plane_dir);
    t += speed * Time.deltaTime;
    yield return null;
}
Destroy(PlaneSpawn.planes[planeNumber]);
yield break;
}
private void OnDestroy()
{
    for (int i = 0; i < planeSpawn.bezierPoints.GetLength(0); i++)
    {
        Destroy(planeSpawn.bezierPoints[planeNumber, i]);
    }
}
void OnDrawGizmos()
{
    for (int j = 0; j < 7; j+=3)
    {
        int segments_number = 20;
        Vector3 previouspoint = planeSpawn.points[planeNumber, j];
        for (int i = 0; i < segments_number; i++)
        {
            float parameter = (float)i / segments_number;
            Vector3 point = CubicBezier.GetPoint(planeSpawn.points[planeNumber,
j], planeSpawn.points[planeNumber, j+1], planeSpawn.points[planeNumber, j+2],
planeSpawn.points[planeNumber, j+3], parameter);
            Gizmos.DrawLine(previouspoint, point);
            previouspoint = point;
        }
    }
}
}

```