МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»
Циклова комісія *комп'ютерних систем та мереж*
(повна назва циклової комісії)

Допустити до захисту
Голова випускової циклової комісії
комп'ютерних систем та мереж
(повна назва циклової комісії)
(підпис) *Ірина КРАВЧУК*
(ім'я, ПРІЗВИЩЕ)
« 10 » 06 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ
ФАХОВИЙ МОЛОДШИЙ БАКАЛАВР

Тема: *Інтерактивна двовимірна ігрова система з використанням платформи Unity*

Група: *3-011* Спеціальність: *123 «Комп'ютерна інженерія»*

Здобувач освіти (підпис) *Вікторія БЄЛЬНИЦЬКА* (ім'я, ПРІЗВИЩЕ)

Керівник роботи (підпис) *Владислав СОБЧУК* (ім'я, ПРІЗВИЩЕ)

Консультант з оформлення пояснювальної записки (підпис) *Оксана ОСАДЧА* (ім'я, ПРІЗВИЩЕ)

Кривий Ріг 2025 р.

КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

Відділення *комп'ютерної та програмної інженерії*
Циклова комісія *комп'ютерних систем та мереж*
Освітній ступінь *фаховий молодший бакалавр*
Спеціальність *123 «Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ
Голова випускової циклової комісії
комп'ютерних систем та мереж
(повна назва циклової комісії)
(підпис) *Ірина КРАВЧУК*
(ім'я, ПРІЗВИЩЕ)
« 01 » 03 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ОСВІТИ

Бєльницька Вікторія Андріївна
(прізвище, ім'я, по батькові)

1.Тема роботи *Інтерактивна двовимірна ігрова система з використанням платформи Unity*

Керівник роботи *Собчук Владислав Олегович*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по коледжу від « *04* » *04* 2025 року № *51-ст*

2. Строк подання здобувачем освіти роботи з *20.05.2025* по *16.06.2025*

3. Вихідні дані до роботи *Інтерактивна двовимірна ігрова система з використанням платформи Unity*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Історія розвитку ігор та огляд платформи Unity
Проєктування та розробка інтерактивної 2D ігрової системи
Економічне обґрунтування та перспективи

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Презентація Microsoft PowerPoint

6. Консультанти розділів роботи (проекту)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання *13.04.2025*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	*Узгодження технічного завдання з керівником кваліфікаційної роботи*	*13.04.25*	*виконано*
2	*Підбір та вивчення науково-технічної літератури за темою кваліфікаційної роботи*	*15.04.25*	*виконано*
3	*Обґрунтування вибору програмних засобів*	*19.04.25*	*виконано*
4	*Опис компонентів. Обґрунтування їх вибору.*	*23.04.25*	*виконано*
5	*Розробка програмного забезпечення*	*08.05.25*	*виконано*
6	*Дослідження ефективності реалізованих методів.*	*15.05.25*	*виконано*
7	*Написання пояснювальної записки*	*14.05.25*	*виконано*
8	*Перевірка на плагіат пояснювальної записки*	*09.06.25*	*виконано*
9	*Попередній захист кваліфікаційної роботи*	*02.06.25-06.06.25*	*виконано*
10	*Захист кваліфікаційної роботи*		

Здобувач освіти _____ *Вікторія БЄЛЬНИЦЬКА*
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник роботи _____ *Владислав СОБЧУК*
(підпис) (ім'я, ПРІЗВИЩЕ)

StrikePlagiarism.com

Дата звіту 6/9/2025
Дата редагування ---

Документ прийнятий

Звіт подібності

метадані

Назва організації
Ukrainian national aviation university
Заголовок
Бєльницька.Вікторія.Андріївна.3-011.2025.123.ДР
Автор Науковий керівник / Експерт
БєльницькаСобчук В.
підрозділ
Криворізький Фаховий коледж

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

0.40%
0.40%
КП 1

0.40%
0.40%
КЦ

25
Довжина фрази для коефіцієнта подібності 2

11613
Кількість слів

84577
Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв		0
Інтервали		0
Мікропробіли		0
Білі знаки		0
Парафрази (SmartMarks)		5

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз		Колір тексту
ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	https://ela.kpi.ua/bitstream/123456789/38152/3/Sheykina_magistr.doc	17 0.15 %
2	Звіт_з_дипломної_роботи_Ніколаєнко_Андрій_Ігорович_ПМ-61м_2024.docx [redacted] 12/12/2024 National University of Water and Environmental Engineering (National University of Water and Environmental Engineering)	14 0.12 %

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Інтерактивна двовимірна ігрова системи з використанням платформи *Unity*»: 71 сторінка основного тексту, 52 рисунка, 5 додатків, 15 використаних джерел.

АРКАДА, 2D, UNITY, ГРА, FLAPPY BIRD, ENGINE, ПРОГРАМУВАННЯ, СЦЕНА, АНІМАЦІЯ, ГЕЙМПЛЕЙ, ПЕРСОНАЖ, СПРАЙТ, ФІЗИКА, СКРИПТ, ОБ'ЄКТ, КОЛАЙДЕР, UI, C#.

У цій кваліфікаційній роботі описується процес створення *2D*-гри у жанрі аркада, подібної до *Flappy Bird*, з використанням ігрового рушія *Unity*. Основною метою проєкту є реалізація простої інтерактивної гри, де гравець керує персонажем, уникає перешкод, набирає очки та має можливість повторити гру після поразки.

Під час виконання роботи вивчено особливості аркадних ігор, розроблено логіку гри, оформлено інтерфейс і реалізовано основні елементи геймплею. Гра створена з урахуванням простоти у використанні, стабільної роботи та зручності для користувача. У роботі також розглядається структура проєкту в середовищі *Unity* та послідовність усіх етапів - від ідеї до тестування.

У результаті створено повністю функціональний прототип гри, який можна використовувати як приклад для навчання або як основу для подальшої розробки. Робота демонструє практичні навички у програмуванні, роботі з *Unity* та створенні інтерактивних застосунків.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

2D (Two-Dimensional) – двовимірний простір, який використовується для побудови графіки та анімації в іграх.

API (Application Programming Interface) – інтерфейс прикладного програмування; набір засобів для взаємодії з іншими програмами чи сервісами. *C# (C-Sharp)* – об’єктно-орієнтована мова програмування, яка використовується у Unity для розробки ігор.

Collider (Колайдер) – компонент у *Unity*, що визначає область зіткнення об'єкта з іншими об'єктами.

GameObject – базовий об'єкт у *Unity*, до якого додаються різні компоненти для реалізації функціональності.

GUI (Graphical User Interface) – графічний інтерфейс користувача, що забезпечує взаємодію між користувачем і програмою.

Inspector – вікно в *Unity*, яке дозволяє змінювати властивості об’єктів та компонентів на сцені.

Prefab – попередньо налаштований шаблон об'єкта в *Unity*, який можна багаторазово використовувати в сцені.

Rigidbody2D – компонент фізики в *Unity*, який відповідає за поведінку об’єкта під дією фізичних сил.

Scene (Сцена) – ігрове середовище в *Unity*, яке містить усі об'єкти, компоненти та налаштування.

Script (Скрипт) – програмний модуль на *C#*, який додає логіку до об’єктів у грі.

Sprite – зображення або анімація, яка представляє об’єкти у 2*D*-середовищі Unity.

UI (User Interface) – інтерфейс користувача, зокрема кнопки, текстові поля, панелі тощо, що використовуються в грі.

ВСТУП

У сучасному світі комп’ютерні ігри є однією з найпопулярніших форм розваг, що об’єднують мільйони людей різного віку по всьому світу. Ігрова індустрія стрімко розвивається, охоплюючи дедалі ширший спектр жанрів, платформ і технічних рішень. Особливо важливим є розвиток двовимірних (*2D*) ігор, які завдяки простоті, доступності та низьким вимогам до обчислювальних ресурсів залишаються актуальними навіть в умовах технологічного прогресу. Вони ідеально підходять як для початківців у розробці, так і для створення комерційно успішних мобільних застосунків.

Аркадний жанр займає значуще місце в індустрії ігор, оскільки поєднує інтуїтивно зрозумілі правила з динамічним, захоплюючим геймплеєм. Він сприяє створенню ігор, які швидко навчають користувача та водночас залишаються цікавими протягом тривалого часу. Однією з найвідоміших *2D* аркадних ігор останніх років стала *Flappy Bird*, що здобула світову популярність завдяки своїй простоті, високій динаміці та унікальному візуальному стилю. Успіх *Flappy Bird* є прикладом того, що навіть мінімалістичні проєкти можуть стати вірусними й досягти великої аудиторії, за умови вдалої реалізації основної ідеї.

Для створення подібних ігор сьогодні існує широкий спектр інструментальних засобів. Серед них особливо виділяється платформа *Unity*, яка надає розробникам потужний набір засобів для створення як *2D*, так і *3D* ігор. *Unity* підтримує багатоплатформенність, візуальне редагування сцен, фізичні рушії, систему подій, анімацію, звуковий супровід, а також інтеграцію з мовою програмування *C#*. Саме ці переваги роблять *Unity* одним із провідних рішень як для аматорів, так і для професійних студій.

Метою даної кваліфікаційної роботи є проєктування та розробка інтерактивної двовимірної аркадної гри на прикладі *Flappy Bird* із використанням платформи *Unity*. У рамках цього проєкту планується реалізувати повноцінну ігрову систему з базовим керуванням, графікою, логікою нарахування балів, *UI* компонентами та можливістю збереження найкращого результату.

Основними завданнями кваліфікаційної роботи є:

- дослідити історію розвитку аркадних ігор;

- проаналізувати можливості платформи *Unity* для створення *2D* ігор; -

спроєктувати архітектуру ігрового застосунку;

- реалізувати ігрові механіки: рух персонажа, генерацію перешкод, систему підрахунку балів та умови завершення гри;

- створити користувацький інтерфейс;

- провести тестування та базову оцінку продуктивності;

- оцінити економічну доцільність розробки подібних ігор.

Об’єктом дослідження є процес розробки інтерактивних *2D* ігор з використанням сучасних інструментів програмування.

Предметом проєктування є реалізація аркадної гри *Flappy Bird* на платформі *Unity*.

Для досягнення поставленої мети застосовуються методи системного аналізу, об’єктно-орієнтованого проєктування, програмування на мові *C#*, а також

тестування і налагодження в середовищі *Unity*.

Практичне значення роботи полягає у створенні функціонального ігрового застосунку, який може бути використаний як навчальний приклад, демонстрація можливостей *Unity* або стартова основа для майбутніх розробок у сфері інтерактивних *2D*-ігор.

РОЗДІЛ 1
ІСТОРІЯ РОЗВИТКУ ІГОР ТА ОГЛЯД ПЛАТФОРМИ *UNITY*

Індустрія комп’ютерних ігор за останні десятиліття пройшла шлях від простих піксельних аркад до складних інтерактивних світів із реалістичною графікою та продуманою механікою. [1] Розвиток технологій, апаратного забезпечення та інструментів програмування дав змогу значно розширити можливості розробників і надати гравцям новий рівень взаємодії з ігровим середовищем.

Одним із ключових етапів еволюції галузі стало створення універсальних і зручних платформ для розробки ігор, серед яких особливе місце займає *Unity*. [1] Цей розділ присвячений короткому огляду становлення індустрії комп’ютерних ігор, основним етапам її розвитку, а також аналізу платформи *Unity* як сучасного інструмента для створення двовимірних та тривимірних ігор. [4]

1.1 Еволюція аркадних ігор і поява *Flappy Bird*s

Аркадні ігри мають багаторічну історію, яка бере початок ще з 1970-х років. Саме тоді з'явилися перші прості, але захоплюючі проєкти, які дуже швидко стали популярними серед гравців. Однією з таких ігор була *Pong*, створена компанією *Atari* у 1972 році. [13] Вона імітувала настільний теніс і стала справжнім проривом у світі розваг. З того часу аркадні ігри почали стрімко розвиватися, з’являлися нові автомати з іграми на кшталт *Space Invaders, Donkey Kong і Pac-Man*, які буквально захопили геймерів у всьому світі. [4]

На початку 80-х індустрія вже приносила колосальні прибутки,

перевищуючи доходи навіть музичної та кіноіндустрії. Але швидкий ріст і перенасичення ринку низькоякісними копіями призвели до великої кризи. Цю кризу зуміла подолати компанія *Nintendo*, яка у 1985 році випустила свою консоль *NES*. [13] Саме завдяки їй та легендарним іграм, як-от *Super Mario Bros*, інтерес до аркадних ігор піднявся.

З розвитком комп'ютерів та консолей аркадні ігри поступово еволюціонували. З'явилися нові жанри, графіка ставала кращою, ігровий процес - глибшим. Проте навіть у 2000-х роках попит на прості *2D*-ігри не зникав. І яскравим доказом цього стала гра *Flappy Bird*, яка вийшла у 2013 році та буквально підірвала мобільний ринок. [4]

Flappy Bird - це класичний приклад мінімалістичної аркадної гри, яка не має складного сюжету чи графіки, але затягує з першого дотику. Її суть дуже проста: гравець натискає на екран, щоб утримати пташку в повітрі та пролетіти між трубами. Проте виконати це не так легко, як здається. [7] Висока складність, швидкий темп гри та постійне бажання побити власний рекорд зробили *Flappy Bird* справжнім хітом.

І хоча автор гри згодом прибрав її з магазинів через надмірну популярність і увагу, вона залишила значний слід в індустрії. [3] На її основі з'явилися сотні клонів, ремейків і варіацій, а сама гра стала символом того, що навіть найпростіша механіка, реалізована грамотно, може принести величезний успіх. *Flappy Bird* стала прикладом сучасної аркадної гри, яка не потребує великих ресурсів для створення, але здатна зацікавити мільйони.

Саме тому розробка подібної гри є актуальним і цікавим завданням для студентського проєкту. Вона дозволяє попрактикуватися у створенні механіки, графіки, *UI* та збереження даних і все це в межах компактного, але цілісного продукту.

1.2 Огляд платформи *Unity*: основні можливості для *2D*-ігор

Unity це потужна багатоплатформова система для створення інтерактивних застосунків і відеоігор у *2D* і *3D* форматі. [1] Вона об'єднує в собі інструменти для

проєктування, програмування, тестування та публікації проєктів на різних платформах: *Windows, macOS, Android, iOS, WebGL*, ігрові консолі та пристрої віртуальної реальності. [2] *Unity* дозволяє створювати як складні тривимірні ігри з

високим рівнем деталізації, так і прості, але захоплюючі *2D*-ігри, подібні до *Flappy Bird.* Структура початкового екрану *Unity* показана на рисунку 1.1.

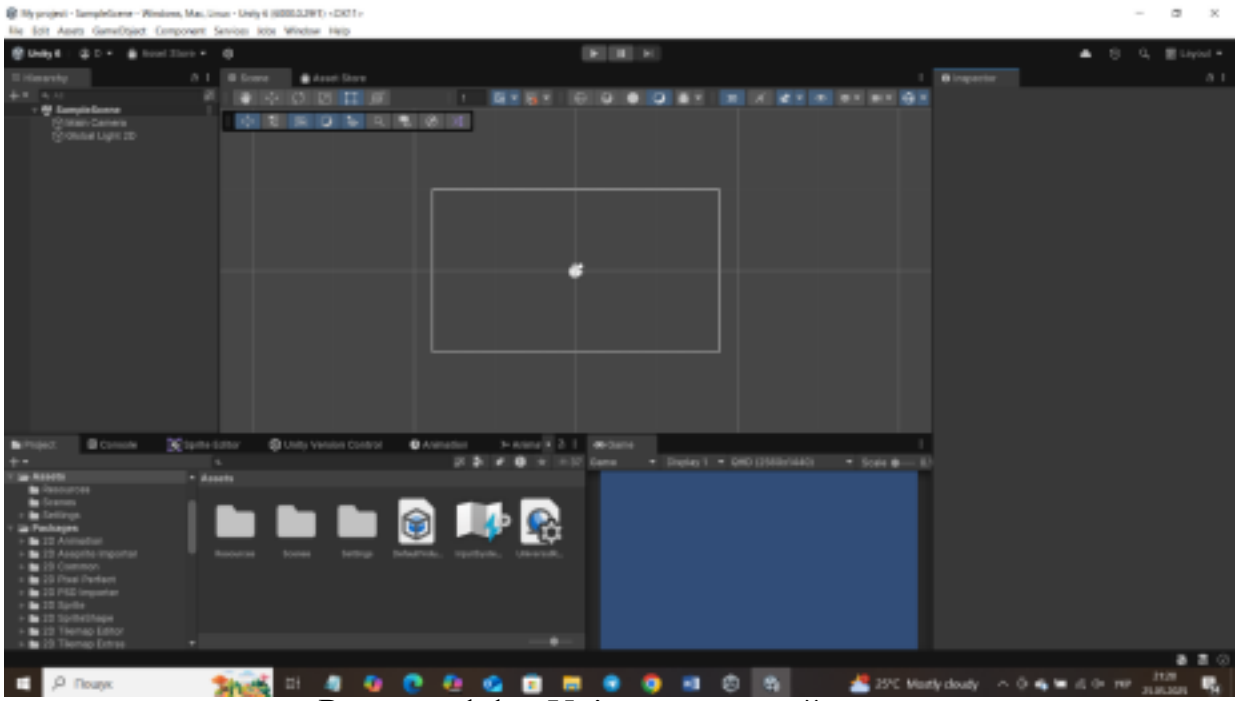

Рисунок 1.1 – *Unity* початковий екран

Однією з ключових переваг *Unity* є її зручність для розробників будь-якого рівня: від початківців до професіоналів. Ігрова логіка реалізується за допомогою мови програмування *C#*, що підтримується в редакторі *Visual Studio* або інтегрованому середовищі. Самі ігри створюються у вигляді сцен, в які розробник додає об’єкти (гравці, перешкоди, *UI*, фони тощо), прив'язує до них скрипти та налаштовує фізику.

У *2D*-проєктах *Unity* активно використовує спрайти це зображення, що представляють персонажів, елементи середовища або ефекти. Спеціальні компоненти, як *Sprite RendeRer* і *Animator*, дозволяють налаштовувати їх зовнішній вигляд і анімацію. Для колізій і фізики в *2D* використовується вбудований рушій *Box2D*, який забезпечує реалістичну поведінку об’єктів при зіткненнях, падіннях або стрибках. [4]

Unity також має гнучку систему компонентів: усі об’єкти складаються з набору властивостей (компонентів), які можна змінювати, розширювати або

відключати. [4] Наприклад, до об'єкта можна додати колайдер, скрипт управління, звук, ефекти часток тощо.

Ще однією сильною стороною *Unity* є її інтеграція з графічними, аудіо та *3D* редакторами такими як *Blender, Photoshop* або *Maya.* Завдяки цьому легко імпортувати власні ресурси без необхідності ручного налаштування.

Для тестування ігор розробник може запускати гру прямо в редакторі, перевіряючи логіку, взаємодію об’єктів, а також отримуючи зворотний зв’язок через вікно *Console*. *Unity* підтримує розширену систему профілювання, яка дозволяє відстежувати продуктивність, оптимізувати навантаження і знаходити «вузькі місця».

Для публікації готових *2D*-ігор *Unity* дозволяє експортувати проєкт у формат, відповідний для вибраної платформи мобільної, десктопної, веб або консолі і навіть налаштовувати окремо якість графіки, роздільність, частоту кадрів тощо. [8]

Таким чином, *Unity* є універсальним рішенням для створення двовимірних ігор. Вона поєднує простоту в освоєнні, гнучкість, розширюваність і потужний інструментарій, що робить її ідеальним вибором для реалізації проєкту на прикладі гри *Flappy Bird*.

Unity побудований за принципом редакторської системи, де вся робота над грою ведеться в спеціальному вікні *Unity Editor*. Це середовище надає доступ до всіх ресурсів, інструментів і налаштувань проєкту. Робота з грою відбувається у вигляді сцен (*Scenes*), кожна з яких містить ігрові об'єкти, розміщені у віртуальному просторі. Всі об’єкти мають набір компонентів (команд, властивостей), які визначають їхню поведінку, вигляд, взаємодію з іншими об’єктами.

Unity має основні вікна редактора. «*Hierarchy*» у цьому вікні показано всі об'єкти, які розміщені на поточній сцені. [1] Тут можна створювати нові об’єкти, групувати їх у батьківські та дочірні (*paRent-child*), видаляти або переміщати по структурі. Наприклад, гравець, труби, фон, камера все це видно в ієрархії. Вікно

«*Hierarchy*» наведено на рисунку 1.2.

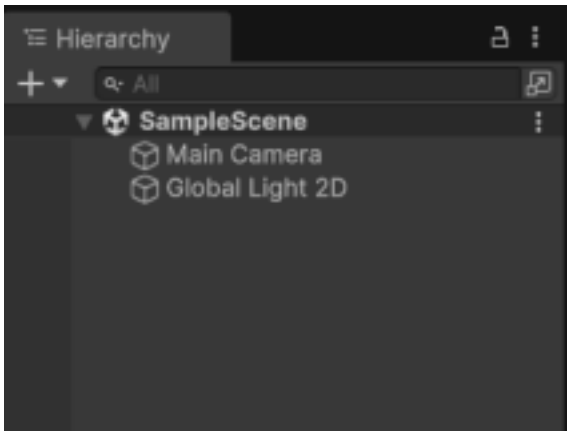

Рисунок 1.2 – Вікно «*Hierarchy*»

«*Inspector*» коли ми виділяємо об’єкт у сцені або в ієрархії, в інспекторі відображаються всі його компоненти це позиція, масштаб, візуальні властивості, фізичні налаштування, прикріплені скрипти тощо. [1] Саме тут ми можемо змінювати налаштування об’єкта, додавати компоненти або змінювати їх значення. Вікно «*Inspector*» продемонстровано на рисунку 1.3.

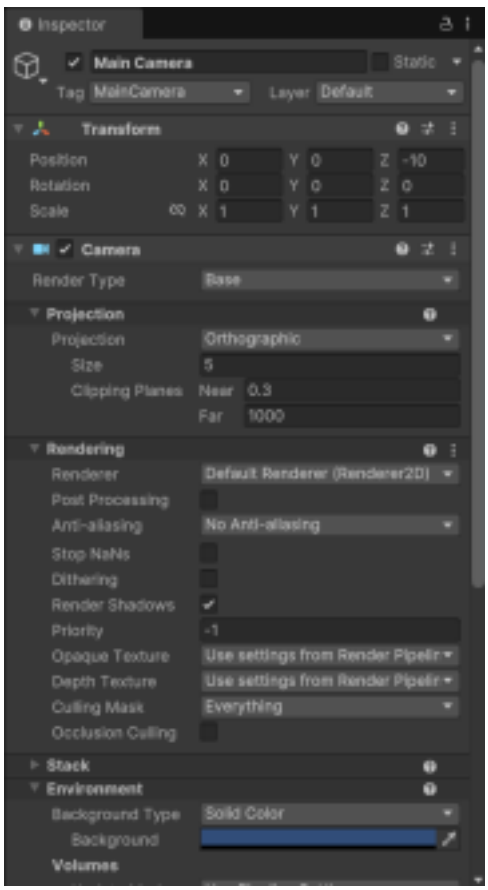

Рисунок 1.3 – Вікно «*Inspector*»

«*Scene*» це вікно дозволяє бачити і вручну редагувати об'єкти у віртуальному просторі. [1] Об’єкти можна рухати, масштабувати, обертати, розміщувати на рівні. Тут же можна додавати нові об'єкти натисканням правої кнопки або через меню *GameObject*. Вікно «*Scene*» ілюструє робочу область у *Unity* рисунок 1.4.

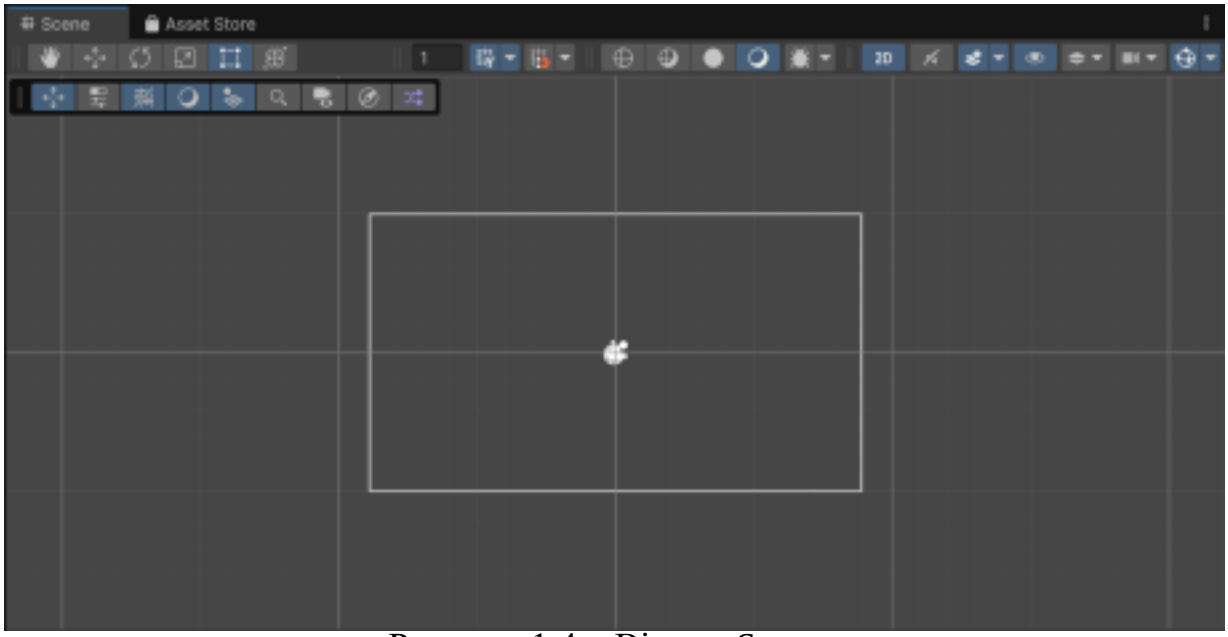

Рисунок 1.4 – Вікно «*Scene*»

«*Game*» показує, як гравець бачить сцену під час гри - з точки зору камери. [1] Саме тут можна протестувати гру, натиснувши кнопку «*Play*», і побачити, як працює логіка, *UI* та фізика. Ігровий режим наведено на рисунку 1.5.

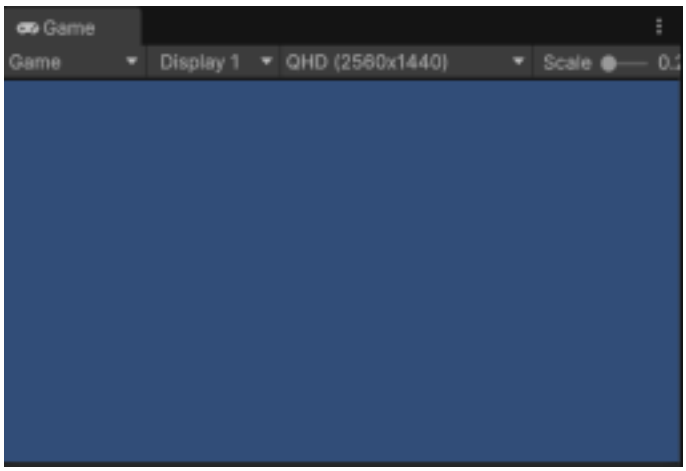

Рисунок 1.5 – Вікно «*Game*»

«Project» - У цьому вікні відображається вся структура вашого проєкту папки, спрайти, звуки, скрипти, сцени. [1] Це сховище всіх ресурсів, які ви можете використовувати в грі. Організація проєкту в *Unity* показана на рисунку 1.6.

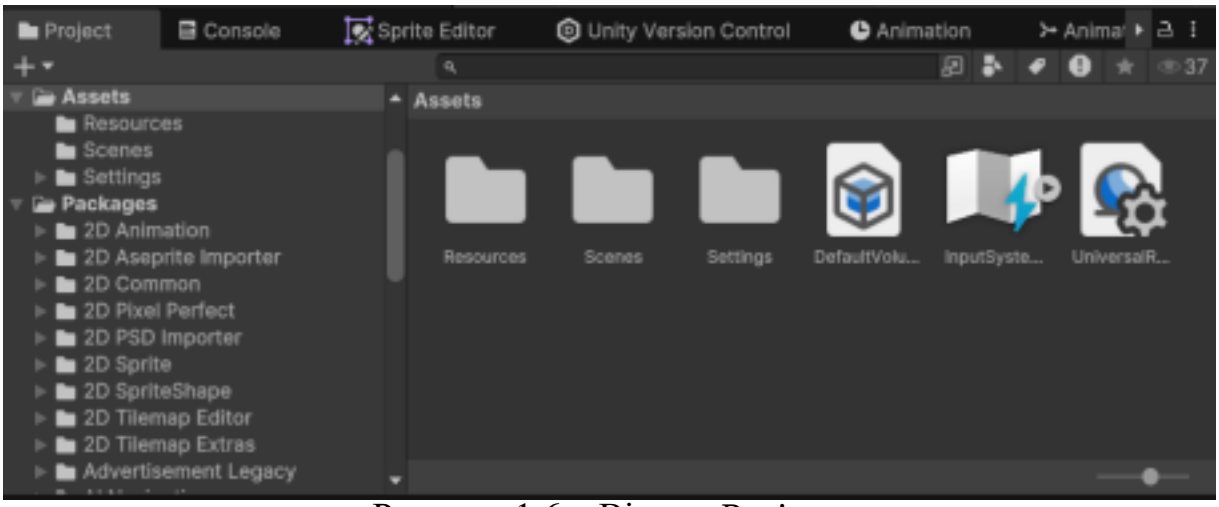

Рисунок 1.6 – Вікно «*Project*»

«Console» - Тут з’являються повідомлення про помилки, попередження та логіка, яку ви можете виводити вручну через код. [1] Це важливий інструмент для налагодження гри. Консоль *Unity* представлена на рисунку 1.7.

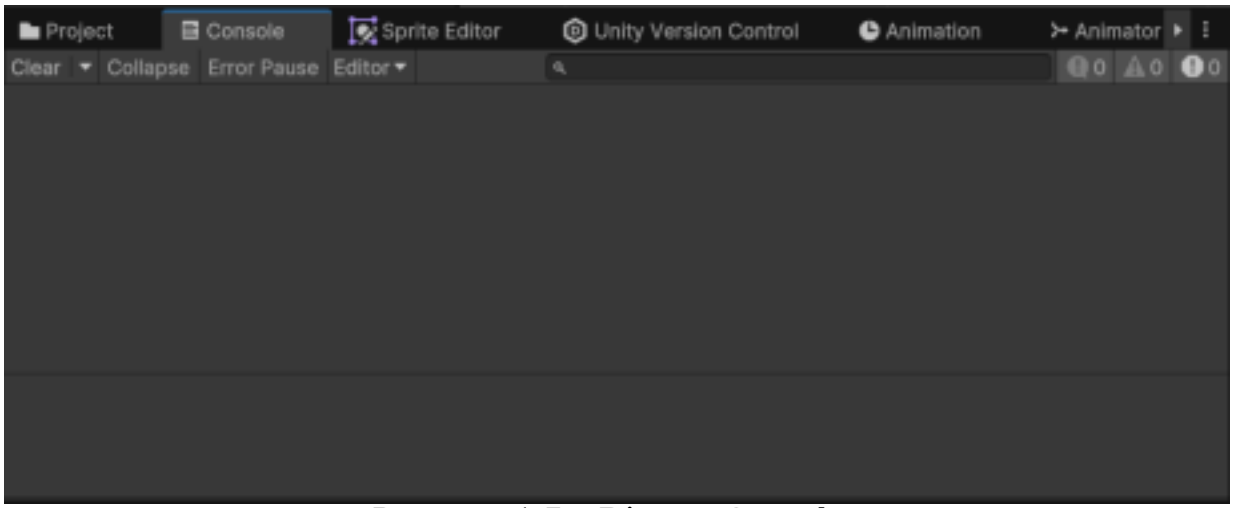

Рисунок 1.7 – Вікно «*Console*»

Кожен об’єкт у *Unity* це *GameObject*, який складається з компонентів. Мінімальний набір це *Transform* позиція, обертання і масштаб об'єкта. Додаткові

компоненти залежать від задачі, *Sprite RendeRer* (для *2D*-графіки), *Box Collider 2D*

(для фізики), *Rigidbody2D* (для руху та сили), *Script* (ваш код на *C#*). *Unity* також підтримує успадкування об'єктів, дочірні об’єкти автоматично слідують за переміщеннями батьківських, що дозволяє будувати складні структури (наприклад, персонаж, який тримає зброю, або труба, до якої прикріплений бонус). У *Unity* основна логіка гри пишеться на мові *C#*. Скрипти підключаються до об’єктів як окремі компоненти. Наприклад, скрипт керування польотом пташки в *Flappy Bird* буде прикріплено до об'єкта *Player*. У цьому скрипті можна реалізувати фізику стрибка, зіткнення з трубами, підрахунок очок тощо. [4] Скрипти мають стандартні методи, які *Unity* викликає автоматично *Start()* - викликається один раз при запуску об'єкта, *Update()* викликається кожен кадр, *OnCollisionEnter2D()* викликається при зіткненні з іншим об’єктом. *Unity* має низку переваг, які роблять її особливо зручною для створення *2D* ігор, зокрема:

- простий старт навіть для новачків, що дозволяє швидко розпочати розробку без глибоких технічних знань;

- підтримка спрайтів, анімації, інтерфейсу користувача (*UI*) та фізичного рушія прямо з коробки;

- можливість швидкого тестування гри в редакторі без необхідності компіляції;

- гнучка система компонентів, яка дозволяє легко змінювати функціональність об’єктів;

- велика спільнота розробників і доступ до *Unity* As*set* Sto*Re*, що містить тисячі безкоштовних і платних ресурсі. [1]

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА РОЗРОБКА ІНТЕРАКТИВНОЇ *2D* ІГРОВОЇ СИСТЕМИ

Процес розробки *2D*-ігрової системи охоплює декілька ключових етапів: від проєктування структури майбутньої гри до реалізації її функціоналу в програмному середовищі. У цьому розділі розглядається архітектура створеної гри *Flappy Bird*, основні компоненти, принципи взаємодії між об'єктами та застосування інструментів *Unity* для досягнення бажаного результату. Особлива увага приділяється логіці поведінки персонажа, генерації перешкод, а також підрахунку очок і умовам завершення гри.

У грі реалізовано кілька основних елементів, що взаємодіють між собою в режимі реального часу. Розглянемо два з них: персонаж (гравець) і перешкоди (труби).

Гравець керує персонажем або пташкою, яка автоматично падає вниз під дією гравітації. При натисканні (клік або дотик) викликається стрибок вгору. Це реалізується через компонент *Rigidbody2D*, до якого прикладено силу по вертикалі. Таким чином, рух гравця по вертикалі є результатом фізичної моделі.

Окрім цього, об'єкт гравця має колайдер (*BoxCollider2D*), який дозволяє визначати зіткнення з іншими об'єктами сцени.

Труби це динамічні об'єкти, що з'являються з правого боку екрана та рухаються вліво з постійною швидкістю. Вони генеруються через певні інтервали часу за допомогою *Spawner* об'єкта, який створює нову пару труб із випадковим зазором між верхньою та нижньою частиною.

Кожна труба також має компонент *BoxCollider2D*, що дозволяє відслідковувати зіткнення з гравцем.

Ключовим моментом гри є зіткнення гравця з трубами або межами сцени. При цьому спрацьовує метод *OnCollisionEnter2D*, після чого гра припиняється, і на екран виводиться вікно поразки. У той же час, проходячи між трубами без

зіткнення, гравець набирає очки це реалізовано за допомогою додаткових тригерів (*Trigger Zone*) або перевірки положення гравця відносно труб.

Таким чином, гравець і труби постійно взаємодіють між собою, створюючи нескінченний ігровий процес, де основна мета пройти якомога далі, не зіткнувшись із перешкодами.

2.1 Постановка задачі та вимоги до гри *Flappy Bird*s

Метою даної роботи є створення динамічної двовимірної аркадної гри *Flappy Bird* з використанням рушія *Unity*. Гра повинна забезпечувати інтуїтивно зрозумілий геймплей, що складається з циклу коротких, але інтенсивних сесій, де гравець прагне покращити власний результат. Основний акцент робиться на повторюваності ігрового процесу, простоті керування та викликові, що мотивує спробувати ще раз після кожної поразки.

Суть гри полягає в управлінні персонажем, яка автоматично летить уперед і повинна пролітати між численними перешкодами у вигляді труб. Гравець взаємодіє з грою через натискання, які змушують птаха злітати вгору. Будь-яке зіткнення з трубами або межами екрану веде до завершення сесії, після чого гравець бачить підсумковий рахунок та, за бажанням, починає нову спробу.

Основні задачі проєкту:

- спроєктувати архітектуру гри, що враховує її модульність і масштабованість;

- реалізувати управління персонажем через фізику *Unity* та взаємодію з користувачем (натискання/дотик);

- створити механізм генерації перешкод із випадковим розташуванням зазорів;

- налаштувати логіку зіткнень між пташкою та елементами сцени. Для досягнення поставленої мети у процесі створення інтерактивної 2D-гри було визначено кілька ключових завдань. Одним із важливих етапів стала розробка системи нарахування очок, яка працює щоразу, коли гравець успішно проходить

перешкоди. Також було реалізовано інтерфейс, що відображає поточний рахунок і найкращий результат. Додатково створено вікно завершення гри, де гравець може побачити свій результат і одразу розпочати нову спробу. Важливо було

забезпечити стабільну роботу гри на різних пристроях, незалежно від платформи.

Щоб гра була зручною для користувача, управління реалізоване максимально просто: пташка піднімається вгору за допомогою одного натискання або дотику до екрана. Перешкоди з'являються автоматично через рівні проміжки часу, кожного разу з новим положенням. У разі зіткнення з трубою або землею гра завершується. За кожну успішну спробу пролетіти між перешкодами нараховується один бал.

Найкращий результат зберігається у пам'яті пристрою і оновлюється, коли гравець перевершує свій рекорд. Візуальне оформлення гри витримано у простому, але виразному 2D-стилі, а інтерфейс містить усі необхідні елементи, зокрема кнопки для перезапуску, виходу та перегляду результатів.

Окрім цього, гра повинна працювати стабільно як на комп'ютерах, так і на мобільних пристроях. Усі елементи оптимізовані таким чином, щоб забезпечити плавне відтворення з частотою не нижче 60 кадрів на секунду. Проєкт має логічну структуру, з чітким поділом на сцени, об'єкти та скрипти, що полегшує підтримку і дає змогу легко вносити зміни в майбутньому.

2.2 Розробка ігрового сценарію та логіки

Сценарій гри у цьому проєкті побудований на простій, але ефективній механіці, яка добре підходить для аркадного жанру. Гравець управляє пташкою, що автоматично рухається вперед і повинна уникати вертикальних перешкод у вигляді труб. Основна мета пролетіти якнайдалі, не зачепивши жодної труби або меж ігрового простору.

Керування реалізоване дуже просто. Один дотик або натискання на екран змушує пташку злітати вгору. Якщо нічого не натискати, вона поступово опускається вниз. Уся суть гри зводиться до того, щоб вчасно реагувати і

натискати, коли потрібно уникнути зіткнень з трубами, які постійно генеруються на екрані.

Ігровий процес побудований так, що після кожної поразки гравець одразу

може почати нову спробу. Це створює ефект затягування, і саме завдяки цьому гра залишається цікавою навіть після багатьох повторень.

В основі гри лежить взаємодія кількох важливих елементів. Гравець керує пташкою, яка реагує на дотики, рухається під впливом гравітації та може змінювати напрямок. Перешкоди у вигляді труб з'являються з певною періодичністю, рухаються вбік і кожного разу мають інше положення, що робить гру непередбачуваною. Ігрове середовище складається з меж екрану, землі, фону та елементів інтерфейсу. Підрахунок очок відбувається автоматично кожного разу, коли пташка успішно пролітає між трубами. Поточний результат одразу відображається на екрані.

Коли пташка зіштовхується з перешкодою або виходить за межі ігрової області, сесія гри завершується. При цьому також працює система збереження рекорду. Вона запам'ятовує найкращий результат і порівнює його з новими спробами.

Інтерфейс гри містить елементи, які показують поточний рахунок, найкращий результат, а також вікно з повідомленням про поразку. Крім цього, у грі передбачено систему візуальних нагород. Наприклад, гравець може отримати медаль за досягнення певного рівня очок.

У результаті ігрова логіка Flappy Bird поєднує простоту керування з поступовим ускладненням, що забезпечує високий інтерес і повторюваність. Це робить її чудовим прикладом для вивчення основ побудови ігрової механіки в середовищі Unity.

2.3 Проєктування архітектури програмного продукту

Ефективне проєктування архітектури є основою якісної розробки будь-якого програмного продукту, включаючи ігрові застосунки. У випадку гри *Flappy Bird*,

побудова архітектури зосереджена на забезпеченні модульності, простоти обслуговування, гнучкості для подальшого розширення функціоналу, а також зручності в реалізації ігрової логіки у середовищі *Unity*.

Unity передбачає компонентно-орієнтовану модель розробки, де всі об’єкти на сцені є екземплярами класу *GameObject*, які можна доповнювати необхідною функціональністю шляхом додавання компонентів.

Цей підхід дозволяє ефективно розподіляти відповідальність між окремими елементами програми. Архітектура гри умовно поділяється на кілька логічних блоків, які взаємодіють між собою, це зображено на рисунку 2.1. Вікно ієрархія головної ігрової сцени зображено на рисунку 2.2. Продовження вікна ієрархія головної ігрової сцени зображено на рисунку 2.3.

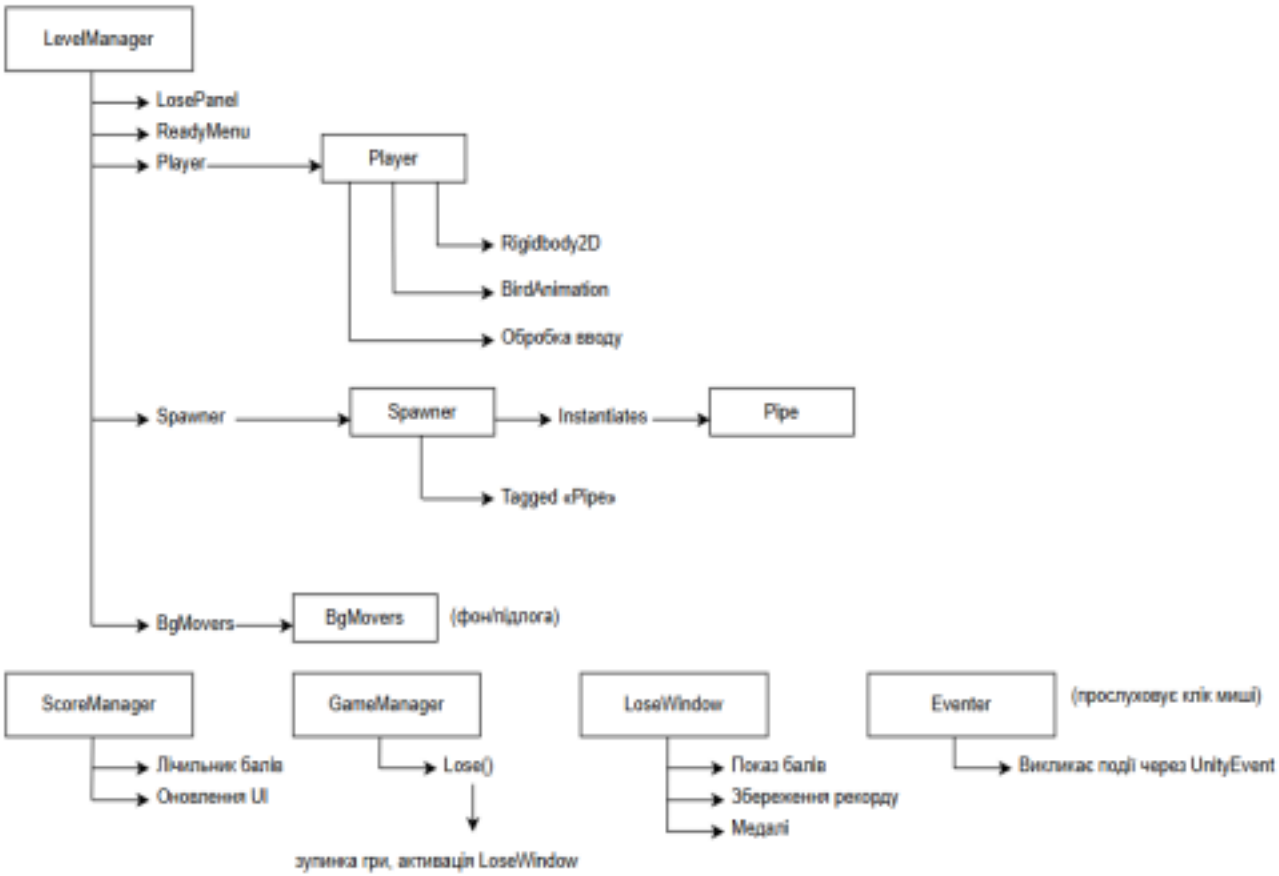

Рисунок 2.1 – Архітектура та взаємозв’язки

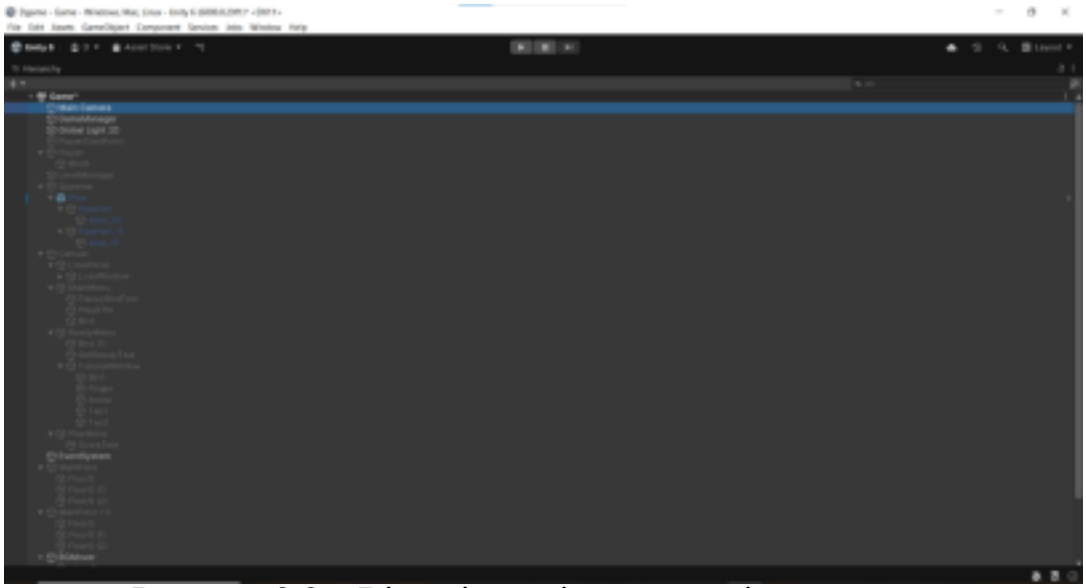

Рисунок 2.2 – Вікно ієрархія головної ігрової сцени

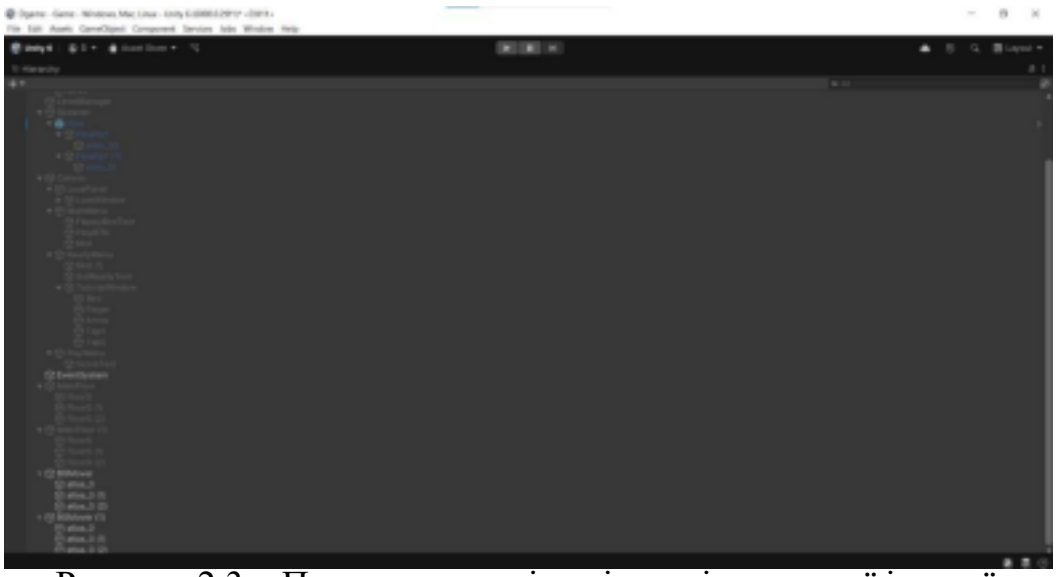

Рисунок 2.3 – Продовження вікна ієрархія головної ігрової сцени

Структура проєкту:

- *Main Camera* це фіксована камера, яка відображає сцену в ортографічному режимі.

- *Canvas* (*UI*) це усі елементи інтерфейсу, включно з текстами рахунку, вікном поразки та меню старту;

- *Background* це візуальні компоненти фону та землі, що рухаються для створення ефекту польоту;

- *GameManager* це керує станом гри: перемога, поразка, перезапуск; - *LevelManager* це відповідає за скидання рівня, запуск фону, об’єктів тощо;

- *Spawner* це створює пари труб (*PipePrefab*) на сцені під час гри; - *Player* це головний об’єкт, що представляє пташку. Має фізику, керування та анімацію;

- *Eventer* це обробляє події кліку або натискання;

- *ScoReManager* це відповідає за підрахунок очок, збереження рекорду.

Основні скрипти та їх призначення наведені в таблиці 2.1.

Таблиця 2.1 – Основні скрипти та їх призначення

№	Клас	Призначення
1	*GameManager*	Управління програшем, перезапуском сцени. *Singleton*.
2	*LoseWindow*	Показ екрана поразки та медалей.
3	*LevelManager*	Повне перезавантаження рівня, *UI*, гравець, труби, фон.
4	*Player*	Керування стрибком гравця, колізіями.
5	*Spawner*	Спавн труб у випадковій позиції.
6	*Pipe*	Взаємодія з гравцем: додавання очків.
7	*ScoReManager*	Облік очків. *Singleton*.
8	*ScoreMedal*	Структура для медалей: зображення + потрібний рахунок.
9	*BirdAnimation*	Поворот птиці в залежності від швидкості.
10	*MoveObject*	Рух об’єкта вліво.

11	*DestroyAfterSeconds*	Знищення об’єкта через заданий час.
12	*BgMover*	Зациклення фону (повернення в початкову позицію).
13	*Eventer*	Обробка кліку миші.

Об’єкт гравець (*Player*) відповідає за поведінку птаха - головного персонажа гри. До нього прикріплені компоненти фізики (гравітація, рух), а також скрипти, що реалізують реакцію на керування. Основна функція підтримувати рух угору при взаємодії з користувачем.

Перешкоди (*Pipes*) генеруються з певною періодичністю. Кожна пара труб складається з верхньої та нижньої частини, які розташовуються з випадковим зазором. Труби рухаються по екрану та знищуються після виходу за межі сцени. Створення труб виконується за допомогою окремого об’єкта генератора (*PipeSpawner*).

Система рахунку (*ScoReManager*) відповідає за нарахування очок при успішному проходженні перешкод, а також за збереження найкращого результату. Реалізує логіку порівняння поточного рахунку з попереднім рекордом.

Керування грою (*GameManager*) центральний модуль, що координує загальний стан гри: запуск, поразку, паузу, перезапуск. Через нього відбувається активація та деактивація інтерфейсних елементів.

Користувацький інтерфейс (*UI*) містить компоненти для відображення рахунку, рекорду, кнопок перезапуску та вікна поразки. Всі елементи інтерфейсу пов’язані з відповідними скриптами, які реагують на зміну стану гри.

Фонове середовище до цього блоку входять об’єкти типу фон, земля, декоративні елементи. Хоча вони не беруть участі в логіці гри, вони забезпечують візуальне ооформлення ігрового світу. Архітектура взаємозв’язків об’єктів у грі показана на рисунку 2.4. Взаємозв’язок компонентів описаний у таблиці 2.2.

Об’єкт *Player* реагує на дії користувача, рухаючись угору завдяки фізиці, і взаємодіє з перешкодами *Pipes*, які при зіткненні призводять до поразки. Об’єкт

PipeSpawner періодично створює пари труб із випадковим зазором, які рухаються по екрану і зникають після виходу за межі сцени. *ScoReManager* відстежує успішне проходження перешкод *Player*, нараховує очки та зберігає рекорд, передаючи інформацію до *UI* для відображення. *GameManager* координує весь ігровий процес, керує запуском, паузою, поразкою та перезапуском, а також управляє станом інтерфейсних елементів, забезпечуючи взаємодію всіх компонентів гри.

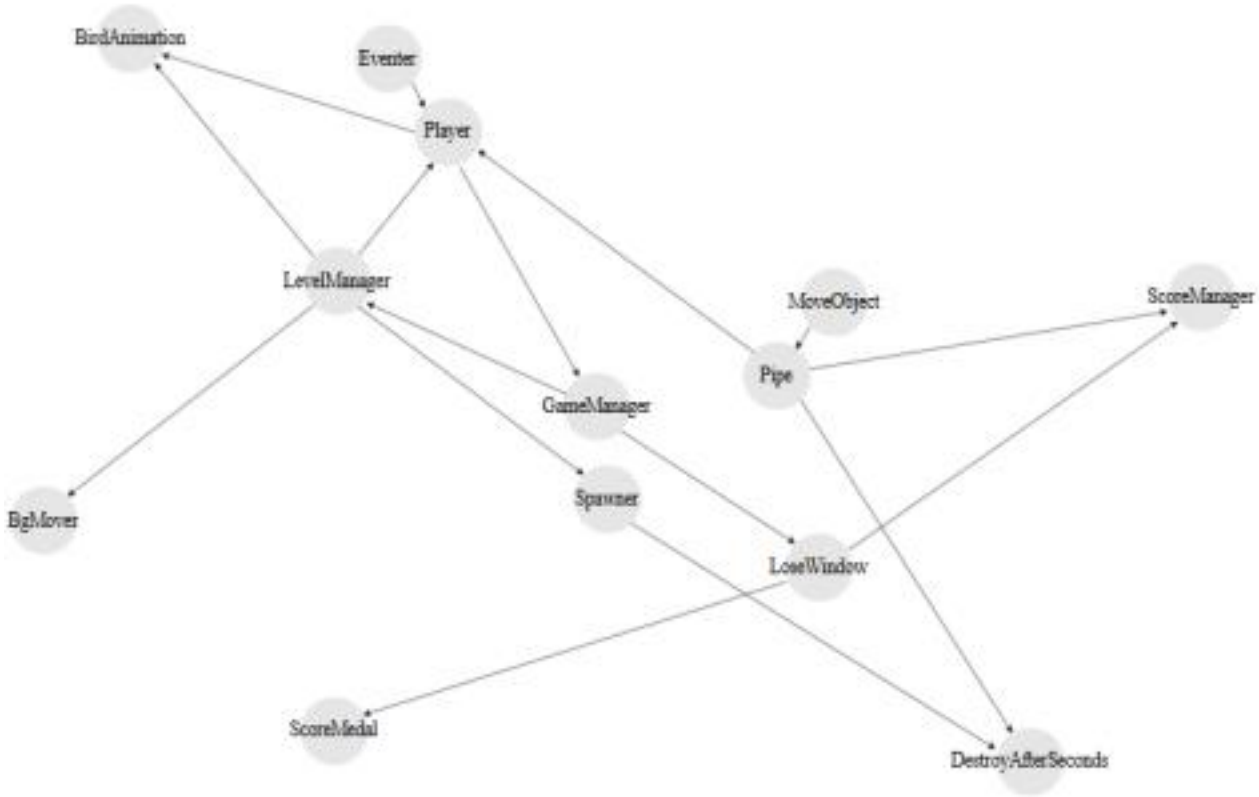

Рисунок 2.4 – Архітектура взаємозв’язків об’єктів у грі

Таблиця 2.2 – Взаємозв’язок компонентів

Джерело	Ціль	Що робить
Spawner	*LevelManager*	Викликає *SetupPipe()* для конфігурації труби.
Pipe	*ScoReManager*	Додає очко при проходженні.
Player	*GameManager*	Повідомляє про програш.

GameManager	*LoseWindow*	Активує *UI* і логіку поразки.
LoseWindow	*ScoReManager*	Читає поточний рахунок.
LevelManager	*BirdAnimation*	Запускає початкову анімацію птиці.
LevelManager	*BgMover*, *MoveObject*	Вмикає рух фону і об’єктів при рестарті.
Eventer	*Player*	Відправляє подію кліку миші.

Архітектура побудована за принципом розділення відповідальності (Separation of Concerns). Кожен скрипт або об’єкт відповідає лише за свою частину функціональності. Наприклад, *Player*Controller не взаємодіє безпосередньо з *UI* цим займається *GameManager*. Уся взаємодія між компонентами відбувається через події або доступ до публічних методів.

2.4 Створення ігрових об’єктів, сцени та анімацій

Розробка ігрового середовища у проєкті *Flappy Bird* реалізується шляхом створення та налаштування ключових об’єктів у середовищі *Unity*, а також додаванням відповідних анімаційних елементів для забезпечення візуальної динаміки гри. [12] Цей етап передбачає побудову візуального простору, визначення фізичних параметрів об’єктів та впровадження інтерактивних елементів, що взаємодіють із користувачем. Взаємозв’язок скриптів ілюструється на рисунку 2.5. Ключові функції наведені в таблиці 2.3.

Таблиця 2.3 – Ключові функції

Назва	Опис
ApplyRotation()	Застосування обертання до птиці в польоті.
StartRotation()	Встановлення кута обертання для початку польоту.

ReStartLevel()	Повний ресет гри.
SetScore(*int*)	Збільшення рахунку та оновлення *UI*.
PlayerLose()	Показ екрана поразки + підрахунок медалі.
SetupPipe()	Призначення тега та життєвого циклу труб.
Lose()	Відображення поразки, зупинка часу.
*ReStart*Scene*()*	Перезапуск сцени.
On*TriggeRe*nter*2D()*	Коли гравець проходить крізь трубу - додається очко.
OnCollisionEnter2D()	Гравець врізається в перешкоду - поразка.
Translate*()*	Рух труби або об'єкта вліво.
Destroy*()*	Видалення об'єкта через деякий час.

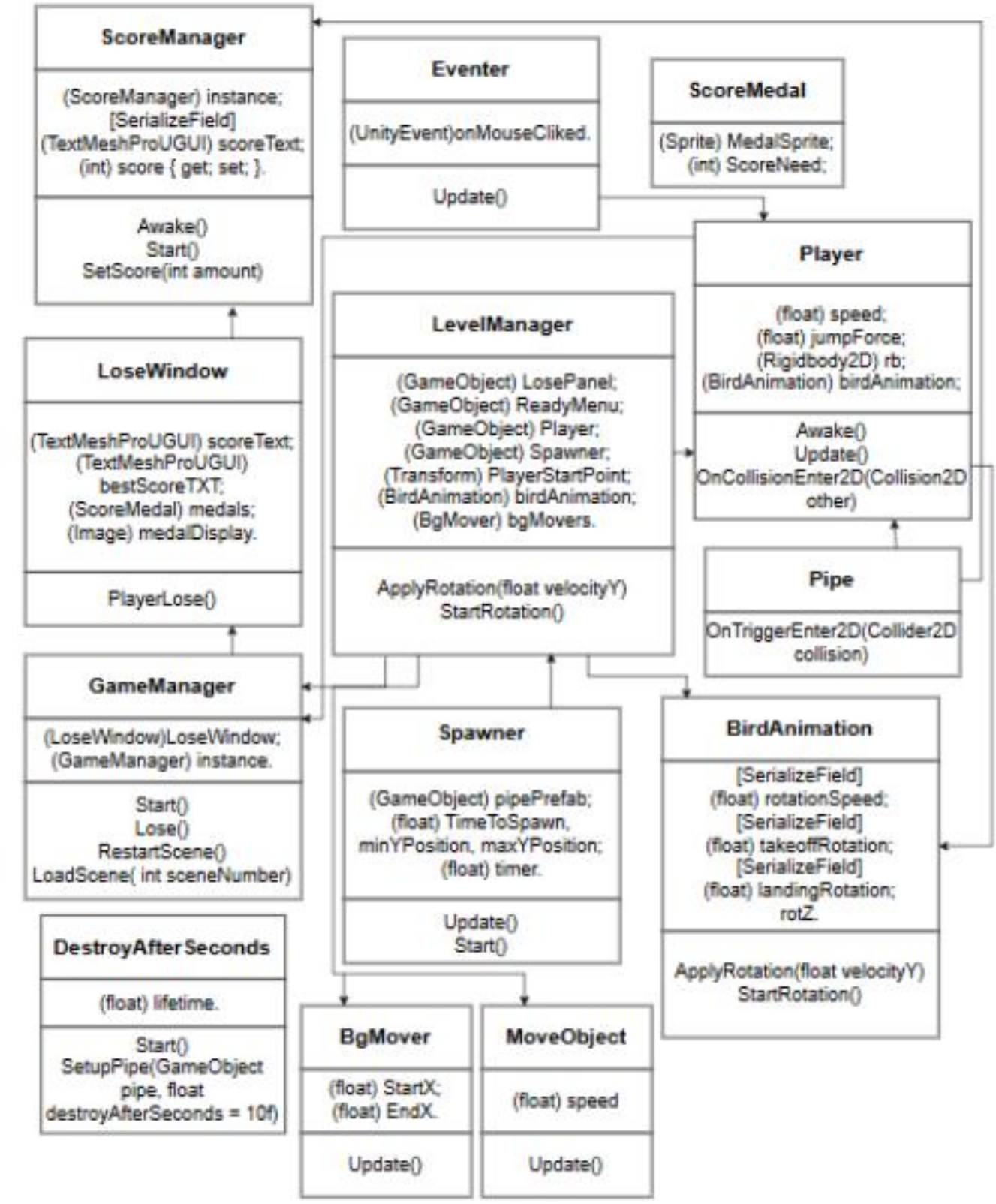

Рисунок 2.5 – Зв’язки скриптів

Player (Пташка) це головний керований об’єкт, що містить компоненти *Rigidbody2D*, *Collider2D*, а також скрипт для керування стрибками (*Player*Controller). Його фізичні властивості налаштовуються для реалістичної

поведінки (гравітація, інерція).

*Pipe*s (Труби) це перешкоди, які складаються з двох частин (верхньої та нижньої). Створюються як префаб і згодом спавняться на сцені скриптом *PipeSpawner*. Для взаємодії з гравцем додаються компоненти *Collider2D.*

Ground (Земля) це статичний об’єкт, який містить *BoxCollider2D* для визначення меж ігрової сцени.

Background (Фон) це візуальний елемент, що не впливає на геймплей, але створює атмосферу гри. Реалізований з компонентами *SpriteRendeRer* та *BgMover* для створення ефекту нескінченного руху.

ScoReZone (Точка нарахування очок) це невидимий об’єкт між трубами, який фіксує факт успішного проходження гравця. Містить *Trigger Collider* та скрипт, що викликає подію нарахування очка.

UI-елементи:

- текст поточного рахунку відображає кількість очок, набраних під час поточного проходження;

- текст найкращого рахунку показує збережене найвище досягнення гравця;

- панель поразки (*LosePanel*) з'являється після завершення гри й містить інформацію про результат;

- кнопка перезапуску (*ReStartButton*) дозволяє миттєво розпочати нову ігрову сесію.

Координація роботи всіх цих об’єктів забезпечується центральним скриптом *GameManager*, який відповідає за керування станами гри, обробку подій поразки, запуск і перезапуск ігрового процесу, а також оновлення *UI* залежно від поточного статусу. Така структура дозволяє підтримувати чітку логіку взаємодії між компонентами, забезпечуючи плавний і зручний ігровий досвід для користувача. Об’єкти та їх стани наведені у таблиці 2.4.

Таблиця 2.4 – Об’єкти та їх стани

Об'єкт	Стан / Управляється	Поведінка
Player	*Rigidbody2 D, Animator*	Реагує на Input, стрибає, обертається, зіштовхується з перешкодами.
Pipe	*Trigger*	Якщо гравець проходить через - додає очко.
MoveObject	*speed*	Рухається ліворуч постійно.
BgMover	*Start*X, *EndX*	Коли досягає краю - повертається в початок (використовується для фону/підлоги).
Spawner	*PipePrefab, timer*	Створює труби кожні N секунд у випадковій висоті.
ScoReManager	*ScoreText, scoRe*	*Singleton*: рахує очки.
LoseWindow	*ScoreText, bestScoReTXT*	Показує рахунок + медаль.
MedalDisPlay	*Sprite*	Відображає досягнуту медаль.

Клас *BgMover* наслідує *MonoBehaviour*. Поля *float Start*X координата, куди переміститься фон при досягненні межі, *float EndX* координата, за яку не можна виходити. Метод *Update()* переміщує об’єкт назад у *Start*X, якщо він пішов лівіше за *EndX*.

Під час створення механіки нескінченного руху фону я розробила власний скрипт *BgMover.cs*. Його основною задачею було забезпечити циклічний рух заднього плану, тобто фонових об’єктів, які імітують постійне переміщення ігрового середовища вперед.

Усі скрипти до гри я створювала у середовищі *Visual Studio*, яке інтегрується з *Unity*. Це забезпечувало зручну навігацію, підсвітку синтаксису та автоматичне автозаповнення. У випадку *BgMover.cs* я оголосила два публічні поля: *Start*X та *EndX*. У методі *Update()* я реалізувала логіку перевірки поточної позиції об’єкта. Фрагмент коду у методі *Update()* можна побачити на рисунку 2.6.

```
void Update()
{
    if(transform.position.x < EndX )
    {
        transform.position = new Vector2(StartX, transform.position.y);

    }
}
```

Рисунок 2.6 – Фрагмент коду методу *Update()*

Цей код я перевіряла безпосередньо в *Unity*, натискаючи кнопку *Play*, і спостерігала вікно *Game*, щоб оцінити, як фон поводиться під час гри. Після написання скрипта я повернулась до *Unity*, де прикріпила його до фонових об'єктів (наприклад, *Bg1*, *Ground1*). У вікні Inspector вручну задавала параметри *Start*X = 10, *EndX* = -10, залежно від розміру сцени. Це дозволило створити ілюзію безперервного руху - об'єкти, досягнувши краю, миттєво повертались назад. Налаштування *BgMover* та взаємодія зі скриптом *Unity* описані на рисунку 2.7.

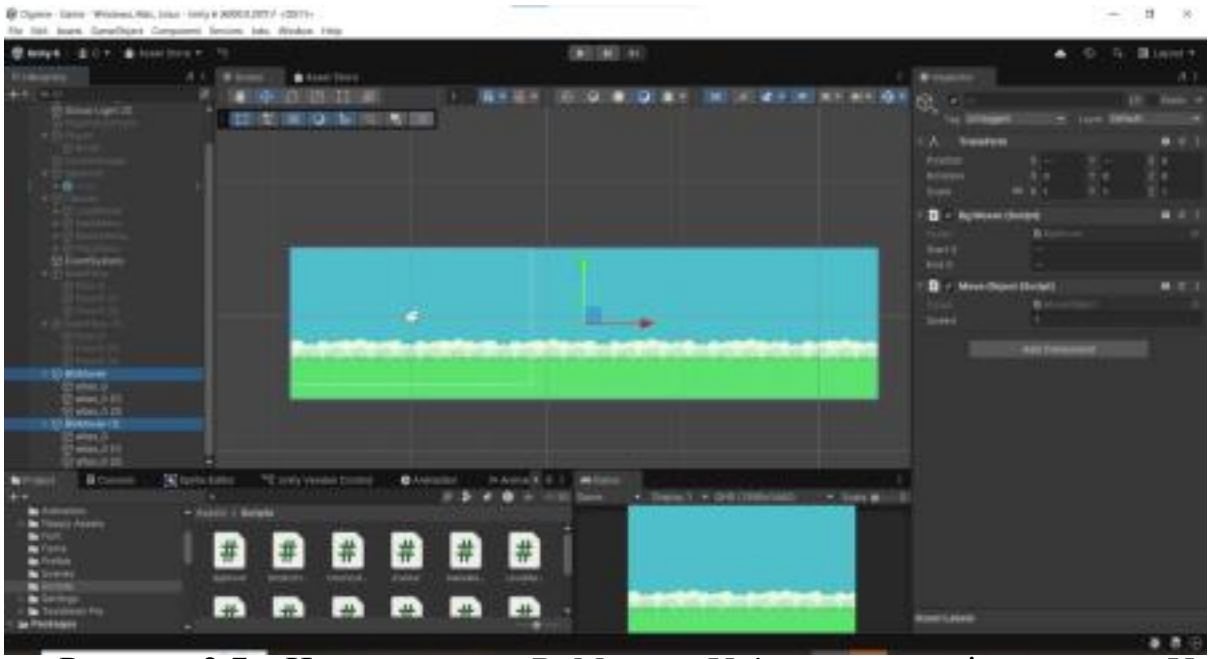

Рисунок 2.7 – Налаштування *BgMover* у *Unity* та взаємодія скрипту з *Unity*

Таке налаштування дозволяє фоновим зображенням рухатись вліво (завдяки іншому скрипту *MoveObject.cs*, і коли вони виходять за межі видимості камери, вони повертаються в початкову позицію. *Start*X, *EndX* є межі для фону. *Update* коли фон досягає *EndX*, він переноситься назад у *Start*X нескінченний скрол.

Клас *BirdAnimation* наслідує *MonoBehaviour* має поля *RotationSpeed* типу *float*, *takeoffRotation* типу *float*, *landingRotation* типу *float*, *rotZ* типу *float* та методи

ApplyRotation(*float velocityY*), *StartRotation()*.

Для покращення візуального сприйняття руху птаха під час гри я реалізувала окремий скрипт *BirdAnimation.cs*, завданням якого є зміна кута нахилу персонажа залежно від швидкості його вертикального руху. Цей ефект забезпечує реалістичну поведінку птаха під час злету та падіння.

Скрипт було написано у *Visual Studio*, де я створила клас *BirdAnimation*, що містить *RotationSpeed* швидкість, з якою змінюється кут обертання, *takeoffRotation* кут нахилу при злеті, *landingRotation* мінімальний кут обертання при падінні, *rotZ* поточне значення кута обертання по осі Z.

Метод *ApplyRotation()* викликається покадрово з параметром *velocityY*, який передається з об'єкта гравця (птаха). Якщо швидкість птаха позитивна (він злітає), якщо ж він падає обертання прискорюється до значення *landingRotation*. Значення *offset* використовується для плавного зменшення кута, враховуючи поточну швидкість. Фрагменти коду з методом *ApplyRotation* наведені на рисунку 2.8.

```
public void ApplyRotation(float velocityY)
{
    if (rotZ > landingRotation)
    {
        float offset = 1f;
        if (velocityY > 0.5f) offset = velocityY;
        offset = Mathf.Clamp01(offset);

        rotZ -= rotationSpeed * Time.deltaTime / offset;
        transform.localEulerAngles = new Vector3(0, 0, rotZ);
    }
}
```

Рисунок 2.8 – Метод *ApplyRotation*

Метод *StartRotation()* встановлює початковий кут, коли птах здійснює перший стрибок, він зображен на рисунку 2.9.

Рисунок 2.9 – Метод *StartRotation*

Після написання скрипта в *Visual Studio* я додала його як компонент до об’єкта *Player* у *Unity*. У вікні Inspector було задано значення *RotationSpeed* = 150, *takeoffRotation* = 25, *landingRotation* = -90. Взаємодія *BirdAnimation* зі скриптом представлена на рисунку 2.10.

Рисунок 2.10 – Налаштування *BirdAnimation* у *Unity* та взаємодія скрипта з *Unity*

Ці значення підібрані експериментально для досягнення природного нахилу птаха. Метод *ApplyRotation()* викликається в основному скрипті *Player.cs*, де визначається поточна швидкість руху птаха через *Rigidbody2D*.velocity.y.

Реалізація *BirdAnimation.cs* надала персонажу візуальну динаміку, що чітко відображає його стан у польоті. Завдяки плавному нахилу, гра виглядає більш "живою", а птах - переконливо реагує на дії гравця. Це також допомогло підвищити загальну якість геймплею.

ApplyRotation(*velocityY*) поступово обертає пташку вниз. *StartRotation()* встановлює початковий кут (при стрибку). *DestroyAfterSeconds.cs* Автоматичне знищення об’єкта.

Клас *DestroyAfterSeconds* наслідує *MonoBehaviour* має поля *lifetime float,* методи *Start()* та *SetupPipe*(*GameObject Pipe*, *float DestroyAfterSeconds* = 10f).

Щоб уникнути накопичення зайвих об'єктів на сцені під час гри, я реалізувала скрипт *DestroyAfterSeconds.cs*, який відповідає за автоматичне

знищення певних елементів через заданий проміжок часу. Це особливо актуально для труб, які постійно генеруються у грі та виходять за межі екрана.

Код було написано у *Visual Studio*, де я створила клас *DestroyAfterSeconds*. У скрипті використовується одне публічне поле *lifetime* - час у секундах, через який об'єкт буде знищено. Фрагмент коду з функцією *Unity Destroy* є на рисунку 2.11. Метод *Start()* викликає стандартну функцію *Unity Destroy()*, яка видаляє об'єкт після закінчення заданого часу.

Рисунок 2.11 – Фрагмент коду з функцією *Unity Destroy*

Окрім цього, я реалізувала додатковий метод *SetupPipe()*, який використовується для динамічного налаштування об’єктів типу труби. Цей метод перевіряє, чи об'єкт не є порожнім, призначає тег "*Pipe*" - це дозволяє пізніше визначати труби при зіткненні, додає до об'єкта цей скрипт, якщо його ще не було, і встановлює значення часу знищення. Метод *SetupPipe()* показан на рисунку 2.12.

Рисунок 2.12 – Фрагмент коду з методом *SetupPipe()*

Після написання скрипта я інтегрувала його в логіку гри через скрипт *LevelManager*, який викликає *SetupPipe()* під час створення кожної нової пари труб. Завдяки цьому труби автоматично знищуються через 10 секунд після появи, що дозволяє не перевантажувати сцену та зменшує витрати пам’яті під час

тривалої гри.

У самій *Unity* компонент *DestroyAfterSeconds* не потрібно додавати вручну - він додається до об'єкта програмно через *AddComponent*. Використання *DestroyAfterSeconds.cs* дозволило автоматизувати очищення сцени від непотрібних об’єктів, що є важливим для оптимізації продуктивності. Завдяки цій реалізації гра працює стабільно навіть при великій кількості згенерованих об'єктів. Поєднання роботи у *Visual Studio* та *Unity* забезпечило швидке тестування та інтеграцію скрипта у загальну архітектуру проєкту.

Клас *Eventer* наслідує *MonoBehaviour*. Поля *onMouseCliked UnityEvent*. Метод *Update()* для зручної обробки кліків миші або натискань на екран я реалізувала окремий скрипт *Eventer.cs*, який дозволяє запускати події через систему *UnityEvent*. Такий підхід спрощує логіку проєкту та дозволяє легко зв’язувати взаємодію з користувачем без жорсткого прив’язування компонентів один до одного.

Скрипт було створено у *Visual Studio*, де я імпортувала простір імен *UnityEngine.Event*s для використання подій. У класі *Eventer* я оголосила одне публічне поле *onMouseCliked* це об'єкт типу *UnityEvent*, що дозволяє зв’язати зовнішні методи з подією натискання.

У методі *Update()* я реалізувала перевірку натискання лівої кнопки миші, фрагмент коду з методом *Update()* наведено на рисунку 2.13.

Рисунок 2.13 – Фрагмент коду з методом *Update()*

При виявленні натискання викликається метод *Invoke()*, який активує всі функції, призначені на цю подію в *Unity Editor*.

У редакторі *Unity* я додала скрипт *Eventer* до окремого порожнього об’єкта. Після цього в інспекторі стало доступним поле *On Mouse Clicked*, у якому я вручну прив’язала методи, які мають запускатися при натисканні зокрема, *Player.Jump()*

або *LevelManager.ReStartLevel()*. Це дозволило організувати виклики без необхідності жорсткого зв’язування між об’єктами через код, що покращує масштабованість проєкту.

Завдяки використанню *UnityEvent* я досягла більш гнучкої структури взаємодії: при потребі я можу змінити виклики у редакторі без редагування коду. Це особливо зручно на етапі тестування або прототипування, коли потрібно швидко змінити логіку запуску певних дій.

Скрипт *Eventer.cs* став універсальним інструментом для обробки вводу. Завдяки простій реалізації й гнучкому налаштуванню через інспектор, він спростив керування взаємодією між гравцем і грою. Його створення у *Visual Studio* та подальше підключення в *Unity* значно пришвидшили розробку і дозволили централізовано обробляти події кліку без дублювання коду. Відслідковує кліки миші та викликає *onMouseClicked*.

Клас *GameManager* наслідує *MonoBehaviour*. Поля *LoseWindow*, *LoseWindow*, *instance* є *static GameManager* Методи *Start()*, *Lose()*, *ReStart*Scene*()*, *LoadScene*(*int sceneNumber*)

Щоб централізовано контролювати стан гри, запуск, поразку, перезапуск та завантаження сцен, я створила скрипт *GameManager.cs*. Цей клас є центральною точкою керування ігровою логікою та взаємодією з іншими компонентами проєкту.

Скрипт було реалізовано у *Visual Studio*, де я оголосила ключові поля. *LoseWindow* посилання на об’єкт, що відповідає за відображення вікна поразки, *instance* статична змінна, яка дозволяє отримати доступ до цього класу з будь-якої точки проєкту (реалізація патерну *Singleton*). У методі *Start()* відбувається ініціалізація *Singleton*.

Метод *Lose()* викликається при зіткненні гравця з перешкодами. Фрагмент коду з методом *Lose()* можна побачити на рисунку 2.14. Він активує вікно поразки (*LoseWindow*.*GameObject*.*SetActive*(*true*)), запускає метод *PlayerLose()*, який показує рахунок і медалі, зупиняє гру, встановлюючи *Time.timeScale* = 0.

Рисунок 2.14 – Фрагмент коду з методом *Lose()*

Метод *ReStart*Scene*()* використовується для повторного запуску поточної сцени, відновлює нормальну швидкість часу (*Time.timeScale* = 1) і перезавантажує сцену через *SceneManager*. Фрагмент коду можна побачити на рисунку 2.15.

Рисунок 2.15 – Метод *ReStart*Scene*()*

Метод *LoadScene*(*int sceneNumber*) дозволяє переходити до іншої сцени за її індексом.

У редакторі *Unity* було створено порожній об’єкт з назвою *GameManager*, до якого прикріплено відповідний скрипт. У вікні Inspector вручну було пов’язано компонент *LoseWindow*, який відповідає за графічне відображення екрану поразки.

У *Canvas* були розміщені кнопки "*ReStart*" та "*Menu*", для яких через вікно Inspector були встановлені події *OnClick*. Ці події активують публічні методи *ReStart*Scene*()* та *LoadScene()* відповідно. Взаємодія скрипту *LoadScene()* з *Unity* продемонстрована на рисунку 2.16.

Рисунок 2.16 – Взаємодія скрипту *LoadScene()* з *Unity*

У *Build Settings* було додано всі сцени, необхідні для коректної роботи методів завантаження сцен, з чітким визначенням їх порядку (індексу). Реалізація *GameManager* забезпечила чітку і централізовану структуру керування ігровим циклом. Інші компоненти, зокрема *Player*, можуть легко взаємодіяти з *GameManager* через його статичний екземпляр. Це дозволяє викликати методи, такі як *Lose()*, у момент зіткнення або завершення гри, без прямого зв'язку між об'єктами. Взаємодія скрипту *ReStart*Scene*()* з *Unity* продемонстрована на рисунку 2.17.

Рисунок 2.17 – Взаємодія скрипту *ReStart*Scene*()* з *Unity*

Використання патерну *Singleton* спрощує архітектуру та підвищує гнучкість проєкту. Уся логіка поразки, перезапуску та зміни сцен зосереджена в одному

місці, що значно полегшує супровід та розширення проєкту в майбутньому. *Lose()* зупинка гри, показ вікна поразки. *ReStart*Scene*()* та *LoadScene()* це перезапуск, перехід між сценами.

Клас *LevelManager* наслідує *MonoBehaviour*, поля *UI LosePanel, ReadyMenu,* ігрові об’єкти *Player, Spawner*, *PlayerStartPoint*. зовнішні компоненти *BirdAnimation, BgMover*s та методи *ReStartLevel(), Start(), SetupPipe*(*GameObject Pipe, float DestroyAfterSeconds* = 10f)

Під час розробки гри у середовищі *Unity* я створила керуючий компонент *LevelManager*, який відповідає за організацію ігрового процесу, зокрема, за повне перезавантаження рівня, керування станами інтерфейсу користувача, налаштування рухомих об'єктів і обнулення прогресу гравця.

Unity надала зручний інструментарій для компонування сцени: я створила ігрові об’єкти (гравця, спавнер, фон, *UI*-елементи) та через інспектор *Unity Editor* прив’язала їх до відповідних публічних полів у скрипті *LevelManager.cs*. Це дало змогу візуально контролювати взаємозв’язки між об'єктами, не змінюючи код вручну.

LosePanel і *ReadyMenu* це *UI*-елементи типу *GameObject*, які були реалізовані у вигляді панелей *Unity Canvas*. Вони керують відображенням меню поразки та початку гри.

Player це основний керований об’єкт на сцені. До нього прикріплено *Rigidbody2D, Collider2D* і кастомний скрипт гравця.

Spawner це об’єкт із логікою генерації перешкод. Його активність контролюється через *LevelManager*.

PlayerStartPoint це *Transform*, який визначає початкову позицію гравця після перезапуску.

BirdAnimation це компонент анімації гравця.

*BgMover*s це масив скриптів, які забезпечують безперервний рух фону для створення ефекту польоту.

Метод *ReStartLevel()* використовується для повного перезапуску рівня. Він викликається після програшу і виконує дії:

- вимикає *UI* панель поразки (*LosePanel*) і вмикає стартове меню (*ReadyMenu*);

- видаляє всі об'єкти з тегом "*Pipe*", які були згенеровані раніше, очищаючи сцену;

- активує гравця та переміщує його в початкову точку;

- скидає фізичні параметри об’єкта *Player*: швидкість, обертання та вмикає симуляцію;

- оновлює рахунок гравця в *ScoreManager*;

- запускає анімацію гравця знову;

- увімкнення *Spawner* відновлює генерацію перешкод;

- всі рухомі об'єкти (фон, труби) знову стають активними; - встановлюється нормальна швидкість часу *Time.timeScale* = 1. *Unity* дозволила ефективно реалізувати цю логіку за допомогою інспектора, системи тегів і компонентної архітектури. Наприклад, щоб знайти всі об'єкти з тегом "*Pipe*", використовується вбудований метод *GameObject.FindGameObjectsWithTag()*, що дозволяє динамічно керувати об’єктами без жорсткого зв’язування.

Метод *SetupPipe*(*GameObject Pipe*, *float DestroyAfterSeconds* = 10f) використовується для початкового налаштування створюваних перешкод (труб). Призначає об’єкту тег "*Pipe*", що дозволяє в подальшому знаходити і керувати ним. Додає компонент *DestroyAfterSeconds*, який відповідає за автоматичне знищення труби через заданий час, якщо такий компонент ще відсутній. Фрагмент коду в якому створюється метод *SetupPipe* продемонстровано на рисунку 2.18.

Рисунок 2.18 – Фрагмент коду в якому створюється метод *SetupPipe*

Цей підхід дозволяє уникнути накопичення зайвих об’єктів у сцені, що важливо для продуктивності мобільних ігор.

У *Unity* я активно використовувала можливості інспектора, таймлайнів, панелі ієрархії, а також сценового перегляду для налаштування поведінки об’єктів. У процесі тестування я використовувала *Debug.Log()* для відстеження викликів методів і стану об’єктів у консолі редактора.

У результаті реалізації та інтеграції скрипта *LevelManager.cs* вдалося автоматизувати процес перезапуску рівня після програшу, зменшити затримки в геймплеї, покращити зручність керування ігровими об’єктами та досягти стабільної роботи гри. Це суттєво підвищило якість користувацького досвіду та надало гнучкий механізм керування сценою без необхідності перезавантаження всієї гри. Налаштування *LevelManager* у *Unity* описані на рисунку 2.19.

Рисунок 2.19 – Налаштування *LevelManager* у *Unity* та взаємодія скрипту з *Unity*

Основний координатор скидає рівень, скидає позицію гравця, очищає старі

труби, перезапускає анімацію, фон, спавн.

Клас *LoseWindow* наслідує *MonoBehaviour*, поля *ScoreText* це *TextMeshProUGUI*, *bestScoReTXT* це *TextMeshProUGUI*, *Medals* це *ScoreMedal*, *MedalDisPlay* це *Image*

Метод *PlayerLose()* у процесі розробки гри в середовищі *Unity* я створила сценарій *LoseWindow.cs*, який відповідає за виведення екрана поразки після завершення гри. Основна мета цього компонента надати гравцеві зворотний зв’язок про його досягнення функції мають показати набрані бали, рекорд і відповідну медаль. Робота над цим скриптом охоплювала як логіку в коді, так і налаштування інтерфейсу в *Unity Editor*.

У *Unity Editor* я створила *UI*-панель (*Canvas*), на якій розмістила два текстових елементи типу *TextMeshProUGUI* для відображення очок (*ScoreText*) та найкращого результату (*bestScoReTXT*), об’єкт *Image* для виводу медалі, масив об'єктів типу *ScoreMedal*, що задаються через інспектор. Взаємодія *LoseWindow* зі скриптом показана на рисунку 2.20.

Рисунок 2.20 – Налаштування *LoseWindow* у *Unity* та взаємодія скрипту з *Unity*

Через інспектор *Unity* усі необхідні компоненти були прив’язані до відповідних полів у скрипті. Наприклад, поле для найкращого результату виглядає

так. Фрагмент коду класу *LoseWindow* наведений на рисунку 2.21.

Рисунок 2.21 – Фрагмент коду класу *LoseWindow*

А під час запуску гри, *Unity* автоматично заповнює його значенням, яке задається вручну через інтерфейс.

Масив медалей я створювала як *ScriptableObject* або просто структуру з мінімальним класом, що містить *ScoreNeed* і *MedalSprite*. Кожну медаль я додавала у список в інспекторі вручну, вказуючи поріг балів для отримання та відповідне зображення медалі (спрайт).

Робота зі скриптом *LoseWindow.cs*. Основна логіка обробки поразки реалізована в методі *PlayerLose()* зображеному на рисунку 2.22.

Рисунок 2.22 – Метод *PlayerLose()*

На цьому етапі отримується поточний рахунок з менеджера очок і виводиться на екран. Далі виконується перевірка, чи досяг гравець нового рекорду, фрагмент коду для перевірки рекорду зображено на рисунку 2.23.

Рисунок 2.23 – Фрагмент коду для перевірки рекорду

Для збереження найкращого результату використовується вбудоване сховище *PlayerPRefs* - один із найпростіших способів роботи з локальними даними в *Unity*. Завдяки цьому не потрібно створювати зовнішні файли чи базу даних, що значно пришвидшує розробку. Фрагмент коду з функцією відображення медалі на рисунку 2.24.

Рисунок 2.24 – Фрагмент коду з функцією відображення медалі

Цикл перебирає всі об’єкти медалей і, якщо гравець досяг необхідного результату, показує відповідне зображення. В *Unity* кожному елементу медалі можна було вручну призначити спрайт, поріг очок та опис.

Для перевірки роботи цього функціонала я запускала гру в *Play Mode*. Щоб спровокувати поразку, навмисно керувала персонажем до зіткнення з перешкодою, після чого викликався метод *PlayerLose()* і в інтерфейсі з’являлися оновлені очки, рекорд та медаль. Крім того, я використовувала *Debug.Log()* для виведення результатів у консоль під час тестування і налагодження логіки.

У результаті інтеграції скрипта *LoseWindow.cs* та налаштування елементів *UI* у *Unity* було реалізовано функціональне і візуально привабливе вікно поразки. Воно

не лише демонструє поточний і найкращий рахунок, але й мотивує гравця через систему винагород. Це підвищує залучення користувача та робить ігровий процес більш захопливим. Ця функція відображає поточний та кращий рахунок та показує медаль (на основі балів).

Клас *ScoreMedal* та поля *MedalSprite* типу *Sprite, ScoreNeed* типу *int*. Під час розробки гри в *Unity* для реалізації системи нагородження гравця медалями було створено окрему структуру *ScoreMedal* у скрипті *Medal.cs*. Ця структура не

наслідує *MonoBehaviour*, оскільки вона не є окремим компонентом на сцені, а виконує допоміжну роль зберігає дані про медалі, які можна призначити залежно від набраного рахунку. Фрагмент коду класу *ScoreMedal* на рисунку 2.25.

Рисунок 2.25 – Клас *ScoreMedal*

У середовищі *Unity Editor* я створила список (масив) об’єктів *ScoreMedal*, які задаються в інспекторі через інший скрипт *LoseWindow.cs*. У редакторі я вручну додала елементи до масиву *Medals*, кожен з яких містив *Sprite* це графічне зображення медалі (бронзова, срібна, золота) та *ScoreNeed* це мінімальна кількість очок, яку потрібно набрати для отримання цієї медалі.

У грі реалізована система нагород у вигляді медалей, які нараховуються залежно від результату гравця:

- бронзова медаль дається при проходженні 10 перешкод (10 очок); -

срібна медаль дається при проходженні 20 перешкод (20 очок); - золота

медаль дається при проходженні 30 перешкод (30 очок);

- платинова медаль дається при проходженні 40 перешкод (40 очок). Ці дані використовуються в методі *PlayerLose()* у *LoseWindow.cs*, щоб визначити, яку саме медаль показати після поразки гравця.

Таке відокремлення структури *ScoreMedal* дозволило спростити управління медалями їх легко додавати, змінювати або переставляти в інспекторі без

редагування логіки в коді. Усі ресурси спрайтів я завантажила до папки *Sprite*s, після чого просто перетягувала їх у відповідні поля *MedalSprite* у *Unity*. Налаштування скрипту *ScoreMedal* та його взаємодія показані на рисунку 2.26.

Рисунок 2.26 – Налаштування та взаємодія скрипту *ScoreMedal* з *Unity*

У підсумку, створення структури *ScoreMedal* забезпечило гнучкий і масштабований підхід до реалізації системи нагород у грі. Вона дозволяє легко змінювати логіку винагород без потреби переписувати скрипти, що значно покращує зручність подальшої розробки й тестування.

Клас *MoveObject* наслідує *MonoBehaviour*, поле *speed* є *float* та метод *Update()*. Під час реалізації механіки гри у *Unity* мною було створено скрипт *MoveObject.cs*, що відповідає за горизонтальний рух ігрових об’єктів (наприклад, труб або фону) вліво. Такий рух створює ефект, ніби птах летить уперед, хоча

насправді персонаж залишається майже нерухомим, а світ навколо нього рухається. Фрагмент коду з класом *MoveObject* на рисунку 2.27.

Рисунок 2.27 – Фрагмент коду з класом *MoveObject*

У середовищі *Unity Editor* я додавала цей скрипт до об’єктів, які повинні переміщуватись автоматично, наприклад, переміщення труб, хмар, землі, фону тощо. Для цього достатньо було перетягнути компонент *MoveObject* на потрібний об'єкт у сцені.

У полі *speed* через інспектор задавалася швидкість руху. Наприклад труби *speed* = 2 та фон *speed* = 1 (повільніше для створення ефекту паралакса). Метод *Update()* викликається кожен кадр. У ньому об’єкт зсувається вліво за допомогою Translate*()*.

Завдяки множенню на *Time.deltaTime* рух залишається плавним і не залежить від частоти кадрів (*FPS*), що є стандартною практикою в *Unity* для анімацій і фізики.

Для зручності я також контролювала активність цього скрипта за допомогою інших класів. Наприклад, у *LevelManager.cs* під час перезапуску рівня виконується пошук усіх об’єктів з цим скриптом:

Таким чином можна легко зупиняти або активувати рух під час паузи, поразки чи перезапуску.

У результаті, завдяки використанню скрипта *MoveObject.cs*, мені вдалося реалізувати плавний, керований рух об'єктів у грі. Це дозволило створити динамічне середовище з ефектом руху, не переміщаючи самого гравця. Крім того, універсальність скрипта дала змогу використовувати його для різних об’єктів без

дублювання коду, що сприяло чистій та масштабованій архітектурі проєкту.

Простий скролінг: рух вліво на заданій швидкості.

Клас *Pipe* наслідує *MonoBehaviour*, метод On*TriggeR*enter*2D*(*Collider2D* collision). У процесі розробки ігрової логіки в *Unity* було створено скрипт *Pipe.cs*,

що відповідає за реакцію на зіткнення гравця з трубою. Його основна задача є додавання очок до рахунку, коли гравець успішно проходить крізь трубу (тобто торкається невидимої зони-тригера всередині труби). Фрагмент коду із створенням класу *Pipe* на рисунку 2.28.

Рисунок 2.28 – Клас *Pipe*

У середовищі *Unity Editor* до об’єкта "труба" (або до окремого дочірнього об’єкта в її центрі) додавався компонент *BoxCollider2D* з увімкненим прапорцем Is *Trigger*. Це дозволяє об’єкту не блокувати гравця, а лише викликати метод On*TriggeRe*nter*2D()* під час перетину з ним.

Також на сцені є гравець із компонентом *Player*, що дозволяє за допомогою методу *TryGetComponentPlayer()* перевірити, чи саме гравець увійшов у тригер. Якщо так тоді викликається метод для зміни рахунку, у результаті, тут використовується синглтон *ScoreManager*, який централізовано керує рахунком у грі.

Під час налаштування рівня в *Unity* я переконалась, що об’єкти труб мають відповідний тег або компонент *Pipe*, усі об'єкти з цим скриптом створюються автоматично через спавнер, скрипт *Pipe.cs* прикріплений до об’єкта з тригером, а не до візуальної частини труби, що дозволяє чітко відслідковувати проходження гравця між перешкодами. Налаштування *Pipe* і взаємодія скрипту зображені на рисунку 2.29.

Рисунок 2.29 – Налаштування *Pipe* та взаємодія скрипту з *Unity*

У результаті використання скрипта *Pipe.cs* я реалізувала надійний механізм підрахунку очок. Завдяки використанню On*TriggeRe*nter*2D()* та методу *SetScore()*, гра правильно реагує на дії гравця, дозволяючи зручно змінювати логіку нарахування очок у майбутньому. Це також забезпечило легкість у масштабуванні гри, адже однакова логіка застосовується до кожної нової труби, яка з'являється в процесі гри.

Клас *Player* наслідує *MonoBehaviour*, поля *speed* типу *float, JumpForce* типу *float, rb* це *Rigidbody2D, BirdAnimation* це *BirdAnimation* та методи *Awake(), Update()* це стрибок, обертання, *OnCollisionEnter2D()* це перевірка зіткнення з трубами або поразка.

Клас *ScoReManager* наслідує *MonoBehaviour*, поля *instance* є *static ScoReManager, ScoreText* це *TextMeshProUGUI, score* типу *int* { *get*; *set*; }, методи *Awake(), Start()* та *SetScore*(*int amount*). Створення, збереження та внесення, вивід на екран.

Клас *Spawner* наслідує *MonoBehaviour*, поля *PipePrefab* це звичайний *GameObject, TimeToSpawn* типу *float, minYPosition* типу *float, maxYPosition* типу *float, timer* типу *float*, методи *Start()* та *Update()* виконують генерування труби та передає в *LevelManager.SetupPipe*

Скрипт *Spawner.cs* відповідає за автоматичну генерацію перешкод (труб) у грі. Його основна функція - створювати труби через задані інтервали часу на випадковій висоті та запускати їх у гру. Це один з ключових елементів геймплейного циклу, який безпосередньо впливає на складність і динаміку гри.

Клас реалізований як компонент *MonoBehaviour*, що дозволяє прикріпити його до об’єкта на сцені (наприклад, до невидимого об’єкта "*Spawner*", який розташований за межами правої частини екрану).

Метод *Start()* викликається один раз на початку гри. У ньому змінна *timer* ініціалізується значенням *TimeToSpawn*, що дозволяє запустити відлік до появи першої труби.

Метод *Update()* викликається покадрово. Відповідає за зменшення таймера та перевірку, чи настав час створити нову трубу, зменшує таймер відповідно до часу, що минув з попереднього кадру (*Time.deltaTime*), якщо таймер досягає нуля або нижче, виконується скидання таймера, потім визначення випадкової вертикальної координати в заданому діапазоні, створення труби на обчисленій позиції за допомогою *Instantiate()*, пошук на сцені об'єкта типу *LevelManager* і виклик методу *SetupPipe()* для подальшого налаштування створеної труби. Фрагмент коду з методом *Update() у класі Spawner* зображено на рисунку 2.30.

Рисунок 2.30 – Фрагмент коду з методом *Update() у класі Spawner* У

редакторі *Unity* для реалізації генератора було виконано різні кроки.

Створено порожній об’єкт на сцені, якому було надано ім’я *Spawner*. До об'єкта було прикріплено скрипт *Spawner.cs*. У вікні інспектора вручну налаштовано такі параметри:

- *Pipe Prefab* призначено префаб труби (заздалегідь створений об'єкт, що містить верхню та нижню трубу).

- Time To Spawn встановлено інтервал (наприклад, 1.5 секунди). -

MinYPosition і MaxYPosition задано вертикальні межі генерації труб, щоб додати випадковість. Налаштування *Spawner* і взаємодія скрипту зображено на рисунку 2.31. Взаємодія класу *Spawner* зі скриптами у таблиці 2.5.

Рисунок 2.31 – Налаштування *Spawner* та взаємодія скрипту з *Unity*

Таблиця 2.5 – Взаємодія класу *Spawner* зі скриптами

Скрипт	Взаємодія
LevelManager.cs	Метод *SetupPipe*(*GameObject Pipe*) викликається після створення труби. У ньому можуть налаштовуватися рух труби, додавання її до списку активних об'єктів, знищення після виходу за межі екрану тощо.
MoveObject.cs	Додається до *Prefab* труби для її руху ліворуч
Player.cs	Побічно взаємодіє гравець повинен ухилятись від труб

Випадковість це використання *Random.Range()* забезпечує варіативність висоти появи труб. Гнучкість це усі параметри можна змінювати в інспекторі без редагування коду.

Оптимізація це виклик *FindAnyObjectByType* у *LevelManager()* спрощує доступ до менеджера рівня, однак у великих проєктах доцільніше використовувати кешування або посилання через інспектор.

Після старту гри труби з’являються через певні інтервали. Вони з’являються на різних висотах, що підвищує складність. Труби рухаються вліво, створюючи динамічний виклик для гравця. Якщо птах зіштовхується з частиною труби - настає поразка (логіка реалізована в *Player.cs*).

Скрипт *Spawner.cs* реалізує важливу частину ігрового процесу тобто циклічне створення об’єктів, перешкод. Така реалізація дозволяє створити динамічний, адаптивний та масштабований механізм генерації, який легко налаштовується в *Unity Editor* і може бути розширений у майбутньому (наприклад, додавання нових типів перешкод, зміна частоти генерації в залежності від складності тощо). Через інтервал створює труби на випадковій висоті.

Таблиця 2.6 містить ключові назви змінних, об'єктів та методів, які використовуються у проєкті для управління логікою гри. Вона дає чітке уявлення про структуру основних компонентів, їх типи та призначення, що є фундаментальним для розуміння подальшої реалізації функціоналу.

Змінні, такі як *Start*X та *EndX*, визначають межі позиції фону, що дозволяє реалізувати механізм безперервного руху фону, що створює ілюзію польоту.
Змінні

RotationSpeed та *rotZ* відповідають за плавне обертання пташки, що додає візуальної динаміки грі.

Об'єкти *LoseWindow* та *Spawner* є важливими елементами інтерфейсу та ігрової логіки, забезпечуючи відображення стану поразки та генерацію перешкод відповідно. Масив *BgMover*s використовується для керування кількома шарами фону та підлоги, що підвищує якість візуального оформлення.

Методи *SetScore(), OnCollisionEnter2D* та On*TriggeRe*nter*2D* є критично важливими для реалізації ігрових механік: підрахунку очок, визначення моменту поразки та реагування на події зіткнення, що забезпечує інтерактивність ігрового процесу.

Таблиця 2.6 – Назви змінних, об’єктів, методів та їх призначення

Назва	Тип	Призначення
*Start*X, *EndX*	*float*	Координати для ресету позиції фону
RotationSpeed	*float*	Швидкість обертання пташки
rotZ	*float*	Поточний кут обертання пташки
LoseWindow	*GameObject*	Панель поразки
Spawner	*GameObject*	Генератор труб
*BgMover*s	*BgMover*	Массив фонів/підлог для перезапуску
scoRe	*int*	Поточний рахунок
SetScore()	Метод	Додає очки і оновлює *UI*
Medals	*ScoreMedal*	Масив медалей
PipePrefab	*GameObject*	Префаб труби
TimeToSpawn	*float*	Інтервал між спавном труб
OnCollisionEnter2D	Метод	Виклик поразки при зіткненні
On*TriggeRe*nter*2D*	Метод	Збільшує рахунок при проходженні труби

Рисунок 2.32, який ілюструє схему послідовностей, є надзвичайно важливим інструментом для розуміння внутрішньої логіки взаємодії між об'єктами та викликами методів у процесі гри. Ця схема детально відображає порядок виконання основних дій, що відбуваються під час ігрового процесу,показує, як різні

компоненти взаємодіють між собою в часі. Такий наочний опис послідовності подій дозволяє розробникам чітко простежити логіку роботи системи, виявити можливі помилки на ранніх етапах.

Рисунок 2.32 – Схема послідовностей

Розширена ієрархія, представлена на рисунку 2.33, демонструє структуру організації об'єктів та їхніх взаємозв’язків у межах проєкту. Ця ієрархія є основою побудови сцени, що відображає, як різні елементи гри від фону та пташки до генератора труб і панелі поразки організовані і взаємодіють між собою.

Під час розробки сцени для гри *Flappy Bird* в *Unity* було обрано *2D*-режим, що повністю відповідає жанру аркади та дозволяє спростити побудову логіки та візуального оформлення.

Для виводу зображення використовується ортографічна камера вона не спотворює перспективу і забезпечує чітке відображення *2D*-елементів. Її розмір підібрано так, щоб основні об’єкти пташка, труби та земля були завжди в межах видимості. Це дозволяє гравцю краще орієнтуватися в просторі.

Рисунок 2.33 – Розширена ієрархія

Оскільки гра виконана у мінімалістичному *2D*-стилі, використовується стандартне освітлення без тіней. Це дозволяє уникнути зайвого навантаження на систему. Фон та інші візуальні елементи мають власне освітлення завдяки *SpriteRendeRer* , тому додаткові джерела світла не потрібні.

У вкладці *Physics 2D Settings* встановлено стандартну гравітацію по осі Y (наприклад, -7.50), щоб пташка поступово падала вниз, якщо гравець не натискає кнопку. Це створює базову механіку геймплею гравець змушений постійно підтримувати висоту. [12]

Також налаштована матриця колізій (*Layer Collision Matrix*) об’єкти поділені на шари (*Player*, *Pipe*s, *Ground*, *UI*) та між деякими з них заборонено взаємодію, щоб уникнути непотрібних зіткнень і підвищити продуктивність.

Рух у грі реалізовано не через переміщення камери, а через переміщення труб, землі та фону у зворотному напрямку. Для цього використовується скрипт *BgMover*, який нескінченно зміщує об’єкти вліво, а коли вони виходять за межі кадру повертає їх у початкову позицію. Таким чином створюється ефект постійного руху вперед. [15]

Інтерфейс реалізовано через *Canvas*, що працює у режимі *ScReen Space Overlay*, тобто незалежно від положення камери. Для адаптації під різні розміри екранів налаштовано *Canvas Scaler* із режимом *Scale With ScReen Size.*

Елементи *UI*, як рахунок, рекорд, кнопки та панель поразки, розміщено на верхньому шарі (*Sorting Layer*: *UI*), щоб вони завжди були поверх геймплейних об’єктів.

Щоб зменшити навантаження на систему та зробити код масштабованим. Префаби для створення типових об’єктів (труб, гравця, зон рахунку). Деактивацію об’єктів неактивні елементи (наприклад, меню поразки) відключено до моменту їх виклику через *SetActive*(*false*).

Pooling (за бажанням) для обробки великої кількості труб доцільно використовувати пул об'єктів, що зменшує кількість створень/знищень у *runtime*. [11]

Для створення динаміки гри були використані прості анімації.

Анімація пташки (*Bird Animation*) реалізована через *Animator*, що дозволяє

виконувати обертання під час польоту та змінювати стан при зіткненнях. Також застосовується обертання відповідно до швидкості руху вгору/вниз (*StartRotation*, *StopRotation*). Фоновий рух (*Parallax Effect*) фони рухаються з різною швидкістю за допомогою скрипта *BgMover*, створюючи ефект глибини. Рух землі реалізується повторюваним зміщенням спрайту по осі X з циклічним поверненням до початкової позиції. Візуальні ефекти опціонально можуть бути додані частинки (*Particle System*) при зіткненні або стрибку, хоча в базовій версії не використовуються.

2.5 Розробка системи керування персонажем

Клас *Player* відповідає за керування рухом персонажа. Він обробляє введення від гравця (натискання клавіші Space або кліку миші), змінює фізичний стан тіла (*Rigidbody2D*), а також викликає анімацію обертання.

У разі зіткнення з перешкодами активується метод *Lose()* класу *GameManager*, що викликає вікно поразки, фрагмент коду на рисунку 2.34.

Рисунок 2.34 – Фрагмент коду, що викликає вікно поразки

Налаштування *Player* і взаємодія зі скриптом на рисунку 2.35. Поля (змінні) *Player* наведені у таблиці 2.7.

Рисунок 2.35 – Налаштування *Player* та взаємодія скрипту з *Unity*

Таблиця 2.7 – Поля (змінні)

Назва	Тип	Призначення
speed	*float*	(Не використовується у коді, але може бути залишено для майбутньої механіки)
JumpForce	*float*	Сила, з якою птах підкидається вгору при стрибку
rb	*Rigidbody2D*	Компонент фізики, який керує рухом птаха
BirdAnimation	*BirdAnimation*	Посилання на скрипт анімації птаха (керування обертанням крила чи всього тіла)

Об'єкт гравця (*Player*) має такі основні компоненти:

- *Rigidbody2D* забезпечує фізичну поведінку (гравітація, інерція); - *Collider2D* (наприклад, Circle*Collider2D*) дозволяє виявляти зіткнення з іншими об’єктами;

- *Player.cs* головний скрипт керування поведінкою гравця; - *BirdAnimation.cs* пов’язаний скрипт для реалізації анімації польоту. Для забезпечення реалістичного руху птаха використовуються такі параметри. *Rigidbody2D* це *Gravity Scale* > 0 — задає силу тяжіння, щоб птах падав

вниз та *Constraints → FReeze Rotation* (по осі Z) — обмежує неочікуване обертання при зіткненнях.

З метою ідентифікації частин перешкод усім елементам труб призначено тег "*PipePart*", який використовується в методі *OnCollisionEnter2D()* для обробки зіткнень.

Для анімації руху птаха до об'єкта *Player* в інспекторі *Unity* прив’язується скрипт *BirdAnimation.cs*, який зазвичай розміщується на тому ж об'єкті або його дочірньому елементі. [3] Основні параметри елементів управління персонажем у таблиці 2.8.

Клас *GameManager.cs* виконує функції керування станами гри. Його реалізація має такі особливості як метод *Lose()*, який активує екран поразки або здійснює перезапуск гри та реалізований доступ до нього здійснюється через *GameManager.instance*. Завдяки цьому стає зручніше управління, наприклад:

- інтуїтивне керування — стрибок виконується одним натисканням миші або клавіші;

- фізична модель — застосування *Rigidbody2D* забезпечує природний рух та падіння;

- реалізм — птах обертається в повітрі залежно від напрямку руху; - анімація — за допомогою *BirdAnimation* реалізовано плавну зміну кута нахилу;

- обробка колізій — поразка визначається зіткненням з об'єктами, що мають відповідні теги.

Таблиця 2.8 – Основні параметри елементів управління персонажем

Компонент	Параметр	Значення	Опис
Player	*JumpForce*	5–10	Сила, з якою гравець стрибає вгору

Rigidbody2D	*gravityScale*	1–2	Інтенсивність впливу сили тяжіння
MoveObject	*speed*	2	Швидкість руху перешкод вліво
BgMover	*StartX*, *EndX*	-10, 10	Межі циклічного руху фону
Spawner	*TimeToSpawn*	1.5	Частота появи нових перешкод

Система керування персонажем реалізована за допомогою фізичних компонентів *Unity* (*Rigidbody2D*), обробки користувацького вводу та анімацій. Додаткові скрипти забезпечують плавність гри та нескінченний рух рівня. Завдяки цьому досягнуто цільову поведінку для аркадного геймплею.

2.6 Реалізація механіки зіткнень і підрахунку балів

Одним із ключових елементів ігрового процесу є обробка зіткнень персонажа з перешкодами, а також нарахування балів за пройдений шлях. У даній грі обидві механіки реалізовані у відповідних скриптах *Unity* з використанням фізики (колізій) та логіки перевірки умов. [7]

Система зіткнень реалізована в класі *Player*. Якщо гравець стикається з об’єктом, який має тег *PipePart*, гра завершується.

Це означає, що зіткнення з частинами труби викликає метод *Lose()* у класі *GameManager*, який відкриває вікно програшу та зупиняє гру (*Time.timeScale* = 0). Підрахунок балів виконується через окремий клас *ScoReManager*. Клас реалізує патерн *Singleton*, щоб до нього можна було звертатися з будь-якого місця. Метод *SetScore*(*int amount*) додає до поточного рахунку вказану кількість очок та оновлює текстовий інтерфейс, це зображено на рисунку 2.36.

Рисунок 2.36 – Метод *SetScore*

Нарахування балів зазвичай виконується, коли персонаж успішно пройшов між трубами. У цьому випадку можна використовувати *Trigger* зони або спеціальні

об’єкти перемикачі. Компоненти зіткнення та підрахунку балів описані у таблиці 2.9.

Таблиця 2.9 – Компоненти зіткнення та підрахунку балів

Компонент	Метод / Змінна	Опис
Player	*OnCollisionEnter2D*	Перевірка зіткнення з трубою
GameManager	*Lose()*	Обробка поразки, зупинка гри
ScoReManager	*SetScore*(*int*)	Збільшення кількості очок
TextMeshProUGUI	*ScoreText*	Відображення рахунку на екрані
Pipe Collider	Is*Trigger*	Може бути використаний для активації нарахування балів без зіткнення

Таким чином, механіка зіткнень забезпечує контроль за завершенням гри при помилках гравця, а система нарахування балів підтримку змагального елементу. Вони є ключовими елементами, що визначають ігрову логіку та взаємодію користувача з грою.

2.7 Побудова користувацького інтерфейсу

Інтерфейс користувача (*UI*) є важливою складовою будь якої гри, оскільки забезпечує зручну взаємодію між гравцем та ігровим середовищем. У даному проєкті інтерфейс включає: відображення рахунку, екран завершення гри, а також кнопки для повторного запуску гри та повернення до головного меню. [8]

Для демонстрації поточного рахунку використано компонент *TextMeshProUGUI* із бібліотеки *TextMeshPro*, що забезпечує високу якість шрифтів та широкі можливості налаштування вигляду тексту. В оновленні рахунку бере участь клас *ScoReManager*. При кожному успішному проходженні перешкоди рахунок змінюється. Фрагмент коду із логікою +1 за кожну перешкоду на рисунку 2.37.

Рисунок 2.37 – Фрагмент коду із логікою інкремента за кожну перешкоду

UI-елемент *LoseWindow* представлений у вигляді панелі в межах *Canvas*. Він містить текст з повідомленням про поразку, відображає набрані очки та включає дві кнопки *ReStart* для перезапуску поточного рівня та *Main Menu* для повернення до головного меню гри. Вікно активується за допомогою методу *Lose()* у класі *GameManager*. Цей метод зупиняє гру та відображає інтерфейс завершення.

Для керування грою використовуються стандартні кнопки *Unity* (*Button*), які розміщені на *Canvas*. Кожна кнопка викликає відповідну функцію при натисканні. Фрагменти коду функцій *RestartScene()* і *LoadScene()* на рисунках 2.38 та 2.39.

Рисунок 2.38 – Фрагмент коду функції *RestartScene()*

Рисунок 2.39 – Фрагмент коду функції *LoadScene()*

Прив’язка методів до кнопок здійснюється через подію *OnClick()* у вікні інспектора *Unity*. [9] Основні елементи користувацького інтерфейсу у таблиці 2.10.

Таблиця 2.10 – Основні елементи користувацького інтерфейсу

Елемент інтерфейсу	Тип компонента	Призначення
ScoRe Text	*TextMeshProUGUI*	Відображення кількості набраних очок
Lose Window	*Panel*	Показ повідомлення про завершення гри
ReStart Button	*Button*	Перезапуск рівня
Menu Button	*Button*	Перехід до головного меню
Canvas	Контейнер *UI*	Базова площина для розміщення усіх елементів *UI*

Розроблений інтерфейс користувача є простим, інтуїтивно зрозумілим та зручним для взаємодії. Завдяки використанню *Canvas* та компонентів *TextMesh Pro* вдалося створити чітке та зрозуміле візуальне оформлення, що не відволікає від основного ігрового процесу, а навпаки допомагає орієнтуватися в ньому. [12]

2.8 Оптимізація продуктивності та збереження даних

Для забезпечення стабільної роботи гри на широкому спектрі пристроїв важливо враховувати як продуктивність, так і збереження даних користувача. У цьому підрозділі розглянуто ключові заходи з оптимізації продуктивності, а також реалізацію механізму збереження результатів гравця. [13]

Основною метою оптимізації є зменшення навантаження на процесор та відеокарту, щоб досягти плавного геймплею. У проєкті було застосовано такі

підходи.

Використання фізики лише там, де потрібно, для персонажа використовується *Rigidbody2D*, але інші об'єкти рухаються скриптово за допомогою *Transform.Translate*, що менш ресурсоємно, використання зображено на рисунку 2.40.

Рисунок 2.40 – Фрагмент коду з *Transform.Translate*

Замість постійного створення та знищення об'єктів (наприклад, труб), можна реалізувати пулінг тобто повторне використання вже існуючих об'єктів. У поточній версії це не реалізовано, але може бути враховано як рекомендація.

Оновлення логіки в *Update()* лише за потреби. У класах, де не потрібно щось виконувати покадрово, логіку розміщено в *Start()* або в окремих методах, щоб уникати зайвих обчислень. [6] Зменшення кількості активних об’єктів: Через обмеження екрана не всі об’єкти зберігаються у сцені одночасно. Фон циклічно переміщується, а перешкоди створюються лише тоді, коли потрібно, це зображено на рисунку 2.41.

Рисунок 2.41 – Циклічне переміщення фону в коді

Збереження результатів (наприклад, рекордів) дозволяє покращити користувацький досвід. *Unity* надає простий механізм через *PlayerPRefs* збереження ключ значення у локальній пам’яті. [10] Методи оптимізації та збереження наведені у таблиці 2.11.

Таблиця 2.11 – Методи оптимізації та збереження

Напрямок	Метод/Технологія	Опис
Оптимізація руху	*Transform*.Translate	Ефективне переміщення об'єктів без фізики
Зниження навантаження	Мінімізація *Update()*	Уникнення зайвих операцій у кожному кадрі
Збереження даних	*PlayerPRefs*	Збереження та завантаження найвищого рахунку
Рекомендовано	Object Pooling	Економія ресурсів шляхом повторного використання об’єктів

РОЗДІЛ 3
ЕКОНОМІЧНЕ ОБҐРУНТУВАННЯ ТА ПЕРСПЕКТИВИ

Розробка прикладного програмного забезпечення, зокрема комп’ютерних ігор, вимагає ретельного аналізу як технічних, так і економічних чинників. Важливо враховувати трудові ресурси, використані інструменти, складність реалізації та потенційні напрямки комерціалізації. У цьому розділі розглянуто структуру витрат на створення гри та її перспективи у контексті розвитку й ринкової привабливості.

3.1 Оцінка витрат на розробку та впровадження

Для реалізації проєкту було визначено ключові етапи, що охоплюють повний життєвий цикл розробки, від концептуального проєктування до тестування та документування. Кожен етап потребував використання відповідних технологій, інструментів і спеціалізованих знань. Основні етапи та ресурси на розробку гри – у таблиці 3.1.

Таблиця 3.1 – Основні етапи та ресурси на розробку гри

Етап роботи	Обсяг, год.	Інструменти технології	Опис діяльності

Аналіз і проєктування	10	*UML, Draw.io, Word*	Створення концепції гри, визначення механіки, побудова схем, написання ТЗ
Розробка ігрової логіки	25	*Unity, C#, Visual Studio*	Створення класів персонажа, руху, перешкод, реалізація фізики та логіки гри
Побудова користувацького інтерфейсу	8	*Unity UI Toolkit, TextMeshPro*	Дизайн і реалізація елементів інтерфейсу: рахунок, меню, кнопки
Тестування, відлагодження	7	*Unity Profiler, Test Runner*	Виправлення помилок, перевірка стабільності гри, балансування складності
Документування та підготовка звіту	5	*Word, Canva, PowerPoint*	Підготовка супровідної технічної документації, оформлення дипломного звіту

3.2 Аналіз економічної ефективності та потенційної монетизації

Гра, побудована за типом *Flappy Bird*, відноситься до категорії казуальних аркад, що мають низький поріг входу для гравців та добру потенційну залученість. Це відкриває можливості для її комерційного просування.

Можливі напрямки монетизації:

- використання мобільних рекламних мереж, таких як *Unity* Ads чи AdMob, дозволяє відображати банери або відео в обмін на бонуси для гравця. -

запровадження платної версії з розширеним функціоналом (наприклад, додаткові персонажі, фони, статистика тощо).

- придбання косметичних змін, нових тем, рівнів або ігрових переваг (інше управління, безсмертя тощо).

- Google *Play* (Android) та itch.io або Steam для ПК-версії

- поширення гри через цифрові магазини дозволяє як отримати відгуки, так і

протестувати інтерес користувачів.

- додавання таблиць рекордів онлайн (Leade*rb*oard) через Fi*Re*base або *Play*Fab;

- створення мультиплеєрного режиму з простим асинхронним змаганням;

- порт на мобільні пристрої, з адаптацією під сенсорне керування. **3.3**

Демонстрація гри та її функціонування

У цьому підрозділі розглянемо процес запуску гри, базові елементи ігрового процесу, а також взаємодію користувача з ігровим середовищем.

Рисунок 3.1 – Стартове вікно гри

Після запуску гри користувач бачить стартове вікно з логотипом гри, кнопкою "Почати". Це вікно реалізовано через *Unity UI*-систему, а логіка запуску контролюється через скрипт *GameManager*. Стартове вікно гри зображене на рисунку 3.1.

Після натискання кнопки "Почати" гравець переходить у режим очікування старту, або так зване *Ready*-меню. На екрані з’являється напис "*Get Ready*", а також невелика анімована інструкція, що пояснює механіку керування птахом

(наприклад: натисни на екран або клавішу, щоб птах злетів угору).

Це меню дозволяє гравцеві ознайомитися з управлінням перед початком активного геймплею. Після першого натискання кнопки стрибка починається безпосередній ігровий процес. *Ready*-меню з туторіалом наведене на рисунку 3.2.

Рисунок 3.2 – *Ready*-меню з туторіалом перед стартом гри

Після натискання кнопки "Почати" запускається основна сцена гри. Гравець керує птахом, що автоматично падає вниз під впливом гравітації. Щоб утримувати його в повітрі, необхідно натискати на екран (або клавішу миші/пробіл), чим викликається ефект "стрибка" (*Jump*). Ігровий процес показаний на рисунку 3.3.

На сцені постійно генеруються перешкоди у вигляді труб (*Pipe*), між якими потрібно пролетіти. Якщо птах стикається з трубою або землею, гра завершується.

Рисунок 3.3 – Ігровий процес: птах летить між трубами

У разі зіткнення птаха з перешкодою, гра автоматично призупиняється, і з'являється вікно поразки (*Lose* Window), в якому відображається поточний рахунок, найкращий результат та медаль, яка надається в залежності від досягнень гравця. Вікно поразки з результатом і медаллю представлено на рисунку 3.4.

Рисунок 3.4 – Вікно поразки з результатом і медаллю

ВИСНОВКИ

Підсумовуючи виконану мною кваліфікаційну роботу, можу впевнено сказати, що проєктування та реалізація інтерактивної двовимірної гри у середовищі Unity стало не лише технічним завданням, а й справжнім викликом, який потребував глибокого залучення, терпіння, послідовності й системного підходу. Ідея створити гру у стилі Flappy Bird виникла через її просту на перший погляд, але водночас цікаву структуру, яка дала мені можливість попрацювати з основними механіками ігрової логіки, візуального оформлення та користувацького інтерфейсу.

Протягом роботи мені довелося зануритися в повноцінний цикл розробки ігрового продукту. Це починалося з аналізу подібних ігор, визначення

функціоналу, планування структури гри та проєктування взаємодії об’єктів. Далі я перейшла до створення ігрової сцени, моделювання логіки поведінки гравця, додавання візуальних ефектів, реалізації реакції на дії користувача, опрацювання системи зіткнень і нарахування очок. Важливим етапом стала також реалізація механізму поразки, що активує відповідне вікно, підраховує підсумковий результат та зберігає найкраще досягнення.

У процесі розробки я переконалася, що гра це не просто набір спрайтів і сценаріїв, а складна система, яка потребує логічно продуманої архітектури. Саме правильне проєктування взаємодії між компонентами дало змогу уникнути плутанини в коді, полегшити тестування і зробити логіку гнучкою. Всі частини гри були побудовані так, щоб їх можна було легко змінювати або доповнювати без порушення загальної структури. Такий підхід виявився надзвичайно ефективним і показав, наскільки важливо ще на початку проєкту сформувати чітке уявлення про всі складові системи.

Окремо варто відзначити роль середовища *Unity*, яке дало мені змогу швидко створювати інтерактивні сцени, керувати фізикою об’єктів, працювати з анімаціями, налаштовувати камеру, інтерфейс і навіть тестувати гру в реальному часі без необхідності тривалої компіляції. Застосування мови C# допомогло

реалізувати складні елементи поведінки об’єктів, а також налагодити взаємодію між скриптами. Усі частини гри від руху персонажа до збереження рекорду були написані вручну, з урахуванням хороших практик програмування.

Готовий результат це стабільна, завершена гра з чіткою логікою, зручним керуванням, візуальною виразністю, продуманим сценарієм поразки та нагород. Вона реагує на дії гравця, зберігає результат, має високий рівень продуктивності, не містить критичних помилок і може бути використана як приклад або основа для більш складних ігрових систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Основи програмування мовою C# / Блінов І. О., Гришин С. В. – Київ: КНТЕУ, 2020. – 312 с.

2. Unity: розробка ігор на C# / Шуліка В. В. – Харків: Фоліо, 2021. – 288 с. 3. Complete C# Unity Game Developer 2D / GameDev.tv Team. – Udemy Course, 2022. – електронний курс.

4. Основи розробки мобільних ігор / Мовчан О. А. – Київ: Видавництво Ліра-К, 2020. – 224 с.

5. Game Engine Architecture / Gregory J. – Boca Raton: CRC Press, 2021. – 1240 p.

6. Introduction to Game Development / Rabin S. – Boston: Cengage Learning, 2021. – 1000 p.

7. Програмування ігор на Unity / Крейг В. – Львів: Піраміда, 2022. – 256 с. 8. Building 2D Games With Unity / Freeman J. – Sebastopol: O'Reilly Media, 2014. – 208 p.

9. Game Programming Patterns / Nystrom B. – 2014. – електронне видання. – 354 p.

10. Unity для початківців / Батуріна І. А. – Дніпро: Університет, 2021. – 198 с. 11. Pro Unity Game Development With C# / Vinter C. – New York: Apress, 2020. – 516 p.

12. Основи дизайну ігор / Мельник А. – Київ: НАУ, 2021. – 172 с. 13. Unity in Action: Multiplatform Game Development in C# / Hocking J. – Shelter Island: Manning Publications, 2018. – 400 p.

14. Алгоритми та структури даних у C# / Троян В. П. – Київ: Кондор, 2020. – 296 с.

15. Розробка ігор за допомогою Unity та C# / Колесник С. В. – Харків: Ранок, 2022. – 248 с.

ДОДАТОК А

Код класу DestroyAfterSeconds

```
using UnityEngine;

public class DestroyAfterSeconds : MonoBehaviour
```

```
{
 public float lifetime = 10f;

 void Start()
 {
 Destroy(gameObject, lifetime);
 }

 public void SetupPipe(GameObject pipe, float destroyAfterSeconds = 10f)
{
 if (pipe == null)
 return;


 pipe.tag = "Pipe";


 if (pipe.GetComponent<DestroyAfterSeconds>() == null)
 {
 var destroyScript = pipe.AddComponent<DestroyAfterSeconds>();
destroyScript.lifetime = destroyAfterSeconds;
 }
 }

 }
```


ДОДАТОК Б

Код класу GameManager

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour {

 public LoseWindow LoseWindow;
```

```
public static GameManager instance;

private void Start()
{
instance = this;
}
public void Lose()
{
LoseWindow.gameObject.SetActive(true);
LoseWindow.PlayerLose();
Time.timeScale = 0;
}
public void RestartScene()
{
Time.timeScale = 1;
SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);  }

public void LoadScene( int sceneNumber)
{
SceneManager.LoadScene(sceneNumber);
 Time.timeScale = 1;
}
}
```


ДОДАТОК В

Код класу LoseWindow

```
using UnityEngine;
using TMPro;
using UnityEngine.UI;
public class LoseWindow : MonoBehaviour
{
 public TextMeshProUGUI scoreText;
 public TextMeshProUGUI bestScoreTXT;
 public ScoreMedal[] medals;
```

```
public Image medalDisplay;

public void PlayerLose()
{
int score = ScoreManager.instance.score;
scoreText.text = score.ToString();
int bestScore = PlayerPrefs.GetInt("BestScore", 0);
if (score > bestScore)
{
bestScore = score;
PlayerPrefs.SetInt("BestScore", bestScore);
}
for (int i = 0; i < medals.Length; i++)
{
if (medals[i].ScoreNeed <= score)
{
medalDisplay.gameObject.SetActive(true);
medalDisplay.sprite = medals[i - 1].MedalSprite;
}
}
bestScoreTXT.text = bestScore.ToString();
}}
```


ДОДАТОК Г

Код класу Player

```
using UnityEngine;
using UnityEngine.SceneManagement;
public class Player : MonoBehaviour
{
 public float speed;
 public float jumpForce;
 public Rigidbody2D rb;
 public BirdAnimation birdAnimation;
 private void Awake()
```

```
{
rb = GetComponent<Rigidbody2D>();
}
private void Update()
{
if (Input.GetKeyDown(KeyCode.Space) || Input.GetMouseButtonDown(0))  {
rb.angularVelocity = 0f;
rb.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
birdAnimation.StartRotation();
}
birdAnimation.ApplyRotation(rb.linearVelocity.y);
}
private void OnCollisionEnter2D(Collision2D other)
{
if (other.gameObject.CompareTag("PipePart"))
{
GameManager.instance.Lose();
}
}
}
```


ДОДАТОК Д

Код класу Spawner

```
using UnityEngine;

public class Spawner : MonoBehaviour
{
 public GameObject pipePrefab;
 public float TimeToSpawn, minYPosition, maxYPosition;
 private float timer;
 private void Start()
 {
 timer = TimeToSpawn;
 }
```

```
 private void Update()
 {
 timer -= Time.deltaTime;

 if (timer <= 0)
 {
 timer = TimeToSpawn;
 float randY = Random.Range(minYPosition, maxYPosition);
 Vector3 spawnPosition = new Vector3(transform.position.x, randY, 0);  GameObject
newPipe = Instantiate(pipePrefab, spawnPosition, Quaternion.identity);  LevelManager
levelManager = FindAnyObjectByType<LevelManager>();  if (levelManager != null)
 {
 levelManager.SetupPipe(newPipe);
 }
 }
 }
 }
```

КРИВОРІЗЬКИЙ ФАХОВИЙ КОЛЕДЖ
ДЕРЖАВНОГО НЕКОМЕРЦІЙНОГО ПІДПРИЄМСТВА
«ДЕРЖАВНИЙ УНІВЕРСИТЕТ «КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ»

РЕЦЕНЗІЯ
на кваліфікаційну роботу

випускника спеціальності: 123 «Комп'ютерна інженерія»
відділення: комп'ютерної та програмної інженерії
циклова комісія: комп'ютерних систем та мереж
Вікторія БЄЛЬНИЦЬКА
(ім'я, прізвище)

1. Актуальність теми: Обрана тема кваліфікаційної роботи «Інтерактивна двовимірна ігрова система з використанням платформи Unity» є актуальною.
2. Кваліфікаційна робота відповідає темі, затвердженій наказом.
3. Завдання на виконання кваліфікаційної роботи виконано у повному обсязі.
4. В результаті виконання кваліфікаційної роботи було виконано проектування інтерактивної двовимірної ігрової системи з використанням платформи Unity.
5. Якість виконання пояснювальної записки та ілюстративного (графічного) матеріалу відповідає вимогам Державних стандартів.
6. Окремо слід відзначити високий рівень реалізації інтерфейсу користувача, логічну структуру програми та обґрунтованість вибору інструментів розробки. Робота засвідчує здатність автора самостійно аналізувати технічні проблеми, шукати ефективні рішення та втілювати їх у вигляд повноцінного програмного продукту..
7. Кваліфікаційна робота заслуговує оцінку «добре».

Рецензент _____
(науковий ступінь, посада)

«____»_____ 2025 р. _____ Андрій КОЖАЄВ
(підпис) (ім'я, прізвище)

З рецензією ознайомлена _____ Вікторія БЄЛЬНИЦЬКА
(підпис) (ім'я, прізвище)