Міністерство освіти і науки України
Університет митної справи та фінансів

Факультет інноваційних технологій
Кафедра комп'ютерних наук та інженерії програмного забезпечення

Кваліфікаційна робота магістра

на тему: «Розробка нового сценарію 2D-гри з використанням штучного інтелекту та оптимізації ігрових механік в середовищі Unity 6»

Виконав: студент групи К23-2м

Спеціальність 122 «Комп'ютерні науки»

Послєдов Данило Миколайович
(прізвище та ініціали)

Керівник к.ф.-м.н., доц. Рудянова Т.М.
(науковий ступінь, вчене звання, прізвище та ініціали)

Рецензент Університет митної справи та фінансів
(місце роботи)

доцент кафедри транспортих технологій та міжнародної логістики
(посада)

к.т.н., доц. Разгонов С.А.
(науковий ступінь, вчене звання, прізвище та ініціали)

Дніпро – 2025

АНОТАЦІЯ

Послєдов Д.М. Розробка нового сценарію 2D-гри з використанням штучного інтелекту та оптимізації ігрових механік в середовищі Unity 6.

Кваліфікаційна робота на здобуття освітнього ступеня магістр за спеціальністю 122 «Комп'ютерні науки» – Університет митної справи та фінансів, Дніпро, 2025.

Об’єктом дослідження є використання сервісів штучного інтелекту в розробках ігрових застосунків.

Предметом дослідження є нові алгоритми та методи розробки ігрового середовища.

Метою роботи є створення нової версії ігрового застосунку з використанням новітніх методів розробки, побудови дизайну рівнів та використанням сервісів штучного інтелекту для покращення або оптимізації вже наявних скриптів.

Дана кваліфікаційна робота присвячена створенню ігрових механік та дизайну рівнів в новій версії додатку на рушію Unity 6 з використанням сервісів штучного інтелекту для дороблення або оптимізаціїї вже наявних алгоритмів. Особливу увагу було виділено створенню скриптів, новому дизайну рівнів, розробленого нелінійним методом, та покращенню або оптимізації вже наявних механік. У процесі роботи було проведено дослідження сучасних тенденцій в розробці ігрових застосунків. Було проведено аналіз ролі сервісів штучного інтелекту в розробці ігрових застосунків, а також актуальності наявних технологій для розв’язання поставленої задачі. Дослідження представляє інноваційний підхід через використання сучасних методів розробки ігрових застосунків та використання інноваційних інструментів розробки для покращення та оптимізації скриптів.

Ключові слова: скрипт, штучний інтелект, клас, об’єкт.

ABSTRACT

Posliedov D.M. Development of the concept of 2D game using artificial intelligence and a new scenario for optimising game mechanics in the Unity 6 environment.

Qualification work for a master's degree in speciality 122 ‘Computer Science’ - University of Customs and Finance, Dnipro, 2025.

The object of research is the use of artificial intelligence services in the development of gaming applications.

The subject of the research is new algorithms and methods for developing a game environment.

The aim of the work is to create a new version of the game application using the latest development methods, level design, and the use of artificial intelligence services to improve or optimize existing scripts.

This qualification work is devoted to the creation of game mechanics and level design in a new version of the application on the Unity 6 engine using artificial intelligence services to improve or optimize existing algorithms. Particular attention was paid to scripting, new level design developed using a non-linear method, and improving or optimizing existing mechanics. In the course of the work, a study of current trends in the development of gaming applications was conducted. We analyzed the role of artificial intelligence services in the development of gaming applications, as well as the relevance of existing technologies for solving the task. The study presents an innovative approach through the use of modern methods of gaming application development and the use of innovative development tools to improve and optimize scripts.

Keywords: script, artificial intelligence, class, object.

ЗМІСТ

ВСТУП

30 листопада 2022 року компанія OpenAI показала світу бета версію свого віртуального помічника з генеративним штучним інтелектом ChatGPT, тим самим сколихнувши такі галузі як IT, кінематограф, література, музика, образотворче мистецтво та комікси. Більшість новинних заголовків майоріли такими заголовками як: ”Штучний інтелект замінить людей”, “Такі професії, як розробники та сценаристи, зникнуть з ринку праці”.

Цей проривний інструмент, заснований на великій мовній моделі, продемонстрував можливості штучного інтелекту в автоматизації творчих і аналітичних завдань, що раніше вважалися виключно людськими. Компанія OpenAI, заснована у 2015 році Ілоном Маском і колишнім президентом Y Combinator Семом Альтманом, стала рушійною силою цих змін, створивши платформу, яка змінила уявлення про взаємодію між людиною та штучним інтелектом.

За невеликий проміжок часу сервіси штучного інтелекту розвилися до небачених масштабів, відкриваючи нові можливості для розробки ігрових застосунків. Штучний інтелект активно використовується для створення реалістичної поведінки персонажів, адаптації ігрового процесу під дії гравців, а також для процедурної генерації контенту, такої як ландшафти, рівні чи сюжети. Це відкриває простір для нових творчих ідей та підвищення якості ігрових застосунків.

Актуальність теми полягає у швидкому розвитку ігрової індустрії та підвищені вимог до інноваційного підходу в створенні ігрових застосунків. Оскільки попередня версія ігрового додатку, розробленого автором, не відповідає за цім тенденціям, вона була оновлена за допомогою новітніх методів, алгоритмів розробки програмного забезпечення та сервісів штучного інтелекту.

Новизна дослідження полягає в впроваджені нових методів розробки ігрових застосунків та поліпшені, оптимізації вже існуючих скриптів за допомогою сервісів

штучного інтелекту, переходу з лінійного дизайну рівнів гри на нелінійний, реалізації розробки на новій версії популярного рушію Unity 6.

Метою дослідження є вдосконалення ігрового застосунку жанру 2D-платформер в середовищі Unity 6, що включає використання нових алгоритмів в розробці скриптів, створення нового дизайну рівнів а також використання сервісів штучного інтелекту для оптимізації ігрового застосунку.

Відповідно до мети реалізовані такі завдання:

1. Аналіз та дослідження явища штучного інтелекту.
2. Оновлення концепції та геймплею ігрового додатку.
3. Розробка архітектури програмного забезпечення, включаючи структуру проєкту, компоненти та системи з можливістю розширення та модифікацій для майбутніх оновлень.
4. Реалізація нових механік гри та створення рівнів що забезпечують цікаву та викликову геймплейну динаміку під ці механіки на рушію Unity 6.
5. Оновлення дізайну рівнів, розробленого нелінійним методом.
6. Тестування, виявлення та виправлення помилок та проблем, оптимізація ресурсів для забезпечення якості гри, документація цих аспектів, внесення пропозицій для подальшого вдосконалення гри.

Предмет дослідження: нові алгоритми та методи розробки ігрового середовища.

Об'єкт дослідження: процеси роботи сервісів штучного інтелекту з алгоритмами розробки скриптів.

Методи дослідження, які були використані для реалізації даного проєкту, включають:

1. Літературний аналіз. Вивчення наукової та технічної літератури, публікацій, статей та документів, що стосуються розробки ігрових застосунків. Цей метод дозволяє отримати теоретичні знання та розуміння основних концепцій, методів та підходів до розробки ігор.

2. Аналіз існуючих сервісів штучного інтелекту для подальшого вибору кращих сервісів для розробки застосунку.

3. Моніторінг методик створення архітектура та застосування проєктування ігрового застосунку.

Результати дослідження: запропонована нова концепція гри, яка містить нові дизайнерські ідеї та геймплейні внесення, що включають впровадження сервісів штучного інтелекту в розробку проєкту. Застосунок матиме потенціал вплинути на область ігрового програмування, зможе зацікавити користувача оригінальними ідеями та механіками, а в подальшому використовуватись іншими розробниками ігрових застосунків.

Структура роботи:

Розділ 1. Дослідження предметної області. В даному розділі буде досліджено вплив штучного інтелекту на ігрову індустрію .

Розділ 2. Методи та алгоритми розробки програмного забезпечення ігрового застосунку. В даному розділі буде проаналізовані методи та алгоритми в розробці нової версії ігрового застосунку.

Розділ 3. Розробка нової версії ігрового додатку з застосуванням сервісів штучного інтелекту. В даному розділі буде спроєктовано та розроблено ігровий застосунок з використанням сервісів штучного інтелекту.

Дана кваліфікаційна робота складається зі вступу, трьох розділів висновків, списку використаної літератури, 21 джерело, 1 додатку. Обсяг роботи 60 сторінок, 13 рисунків та 1 таблиця.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Аналіз сучасного стану і світові тенденції розвитку інформаційних технологій у сфері розробки ігор

Значення розробки ігрових додатків у сучасному суспільстві зростає з кожним роком (таблиця 1.1). Це пов'язано зі зростаючим попитом на розваги, стрімким розвитком технологій і появою нових пристроїв і платформ, які дають змогу людям насолоджуватися іграми на різних екранах і в різних місцях.

Ігри вже давно перестали бути просто засобом розваги і перетворилися на багатофункціональну індустрію, що охоплює такі сфери, як освіта, навчання, терапія і віртуальна реальність. Крім того, ігрові додатки стали важливими інструментами для розвитку креативності, соціальних зв'язків та інновацій. Ігровий ринок - один із найприбутковіших секторів цифрової економіки. Очікується, що він продовжуватиме зростати. Цьому сприяють розвиток мобільних ігор, зростаюча популярність хмарних ігор і використання штучного інтелекту для створення більш реалістичних ігор [1].

Таблиця 1.1

Дохід ринку розробки ігрових застосунків за 2024 рік

Категорія	Дохід (млрд $)	Річний приріст (%)	Частка на ринку (%)
Консольні ігри	51.9	-1.0	28
ПК ігри	43.2	+4.0	23
Мобільні ігри	92.6	+3.0	49
Загалом	187.7	+2.1	100

В Україні ігрова індустрія також не стоїть на місці, вона стала важливим центром розробки ігор. Тут працюють провідні студії, як-от GSC Game World (S.T.A.L.K.E.R.), 4A Games (Metro), Frogwares (Sherlock Holmes), Plarium та інші. Окрім цього, українські студії займають сильні позиції на ринку аутсорсингу,

виконуючи замовлення для міжнародних компаній. Незалежні розробники також створюють інноваційні ігри, які знаходять шанувальників у всьому світі.

Війна суттєво вплинула на ігрову індустрію. Зросла увага до українських розробників, зокрема до серії ігор S.T.A.L.K.E.R., яка стала своєрідним культурним феноменом. Багато студій зазнали змін, частина з них була змушена перемістити офіси, але більшість продовжує працювати над своїми проєктами. Світова спільнота активно підтримує українські проєкти, зокрема через спеціальні акції на платформах на кшталт Steam [2].

Високий попит на ігри змусить розробників створювати високоякісні, інтерактивні та графічно досконалі продукти. Сучасні ігрові рушії є важливими інструментами в цьому процесі, оскільки вони надають платформу для реалізації ідей і перетворення їх на функціональні продукти.

Одним із ключових чинників, що підвищують актуальність розробки ігрових додатків, є постійний розвиток нових технологій. Віртуальна реальність (VR) і доповнена реальність (AR) сьогодні дають змогу створювати інноваційні ігрові застосунки, які дають користувачам можливість по-новому взаємодіяти з віртуальним світом. Технології штучного інтелекту (ШІ), машинного навчання і великих даних допомагають зробити ігри більш персоналізованими, забезпечуючи адаптивну поведінку персонажів поза грою і створюючи унікальні сценарії для кожного гравця. Ще одним важливим фактором є зміна підходу до поширення ігор. Якщо раніше ігри часто продавалися на фізичних носіях або через цифрові платформи, то сьогодні більшість ігор поширюється через інтернет-магазини, такі як Google Play і App Store. Це дає змогу розробникам охопити глобальну аудиторію та отримувати оперативний зворотний зв'язок від користувачів, що дає їм змогу швидко виправляти помилки та покращувати свої продукти. Крім того, ігри free-to-play з мікроплатежами стали ключовою бізнес-моделлю мобільних ігор, яка приваблює велику кількість користувачів і забезпечує стабільний потік доходів за рахунок внутрішньоігрових покупок.

Ігрові додатки популярні серед різних вікових груп. Ігри, орієнтовані на дорослих, можуть бути наповнені складними сюжетами, глибокою механікою і вимогливим геймплеєм, в той час як ігри, орієнтовані на молодшу аудиторію, часто розвивають прості, барвисті ігри з цікавими персонажами та механікою, що сприяє розвитку логічного мислення та співпраці. Дитячі ігрові додатки стали важливим компонентом навчання, даючи змогу дітям не тільки розважатися, а й освоювати нові навички в ігровій формі. Водночас ігри часто використовуються в соціальних і культурних контекстах. Багато ігор мають онлайновий компонент, що дає змогу гравцям взаємодіяти один з одним і формувати великі спільноти. Це послужило основою для створення величезної популярності багатокористувацьких онлайн-ігор. Віртуальні світи, створені в іграх, можуть стати місцем соціальної взаємодії, де користувачі можуть спілкуватися один з одним, ділитися досвідом і навіть створювати власні міні-спільноти. Ця тенденція також сприяє розвитку елементів віртуальної економіки, де користувачі можуть обмінюватися віртуальними товарами, створювати контент і заробляти гроші за допомогою ігор.

Ігри також використовуються в терапевтичних цілях. Зростаюча кількість досліджень показує, що ігри можуть допомогти в лікуванні різних психічних розладів, таких як депресія і тривожність.

Ігри використовуються як частина маркетингових стратегій для залучення нових клієнтів, створення брендованих ігор і проведення промоакцій для просування товарів і послуг. Це створює нові можливості для компаній не тільки у сфері розваг, а й у сфері реклами та продажів. Ігрові додатки можуть виступати як частина рекламної кампанії, залучаючи споживачів до бренду за допомогою внутрішньоігрових взаємодій і винагород за участь.

Ігрова індустрія продовжує розвиватися завдяки інноваціям, технологічному розвитку, зміні підходів до маркетингу та реклами, а також появі нових можливостей монетизації. Розробка ігрових застосунків стає не тільки бізнесом, а й важливим елементом культурної та соціальної інтеграції, що дає змогу користувачам відчути себе частиною глобальної спільноти. Оскільки ігрові

додатки мають величезний потенціал впливу на багато аспектів життя, їхня актуальність у майбутньому тільки зростатиме.

1.2. Аналіз впливу штучного інтелекту на ігрову індустрію

Відеоігри є одним із елементів культури, з моменту своєї появи, і їхнього розвитку, вони тісно пов'язаний із технологічним прогресом. Серед безлічі інновацій, що вплинули на еволюцію ігор, важливе місце посідає інтеграція штучного інтелекту. Від простих алгоритмів на ранніх етапах до складних систем, які сьогодні створюють динамічні ігрові середовища, персоналізують досвід гравців і покращують процес розроблення, – штучний інтелект став невід'ємною частиною ігрової індустрії.

На ранніх етапах розробки відеоігор штучний інтелект обмежувався жорстко закодованими скриптами. Наприклад, у Space Invaders вороги ставали швидшими в міру їхнього знищення, створюючи ілюзію адаптації, хоча насправді це був заздалегідь заданий механізм. У Pac-Man привиди діяли за простими алгоритмами, кожен з яких мав свій власний стиль відстеження. Ці рішення були інноваційними для свого часу і заклали основу для подальшого вдосконалення штучного інтелекту у відеоіграх, а в міру подальшого розвитку технологій у 1980-х і 1990-х роках стало можливим створювати більш складні системи штучного інтелекту(ШІ).

Наприклад, у грі The Legend of Zelda персонажі реагували на дії гравця і змінювали свою поведінку залежно від ситуації. Це був перший крок до створення інтерактивних ігрових світів, у яких NPC взаємодіють із гравцем не тільки за передбачуваними сценаріями. Водночас такі ігри, як серія Civilisation, продемонстрували, що штучний інтелект може брати участь у стратегічному плануванні, ухвалювати рішення і навіть ставати противником гравця. Це був важливий крок уперед, який підняв якість ігрового досвіду на новий рівень, а у 2000-х роках індустрія зробила великий крок уперед, впровадивши просунуті алгоритми, що забезпечують глибший рівень взаємодії.

Проривом стала гра F.E.A.R., у якій вороги під час бою проявляли тактичне мислення, обходили гравця з флангів, використовували укриття і взаємодіяли один з одним для досягнення спільної мети. Ці дії ґрунтувалися на динамічних скриптах, які дозволяли NPC адаптуватися до ситуації в реальному часі. Ця епоха характеризується появою ігор, у яких NPC перестали бути передбачуваними, а гравці стали сприймати їх як частину живого світу.

Сучасні відеоігри демонструють подальший потенціал штучного інтелекту(ШІ), впроваджуючи машинне навчання та інші передові технології. Наприклад, у грі Red Dead Redemption 2 NPC запам'ятовують минулі взаємодії з гравцем і змінюють свою поведінку залежно від того, як гравець ставився до них раніше. Це створює ілюзію живого, динамічного світу, який реагує на кожну дію користувача. Інший приклад – The Last of Us Part II. У ній гравець постійно змушений адаптувати свій підхід, оскільки вороги використовують свої стратегії, а NPC проявляють емоції, які впливають на сприйняття сюжету. Такі ігри підвищують якість ігрового процесу і дають змогу гравцеві відчути себе частиною реалістичного середовища. Крім безпосередньої інтеграції в ігровий процес, Штучний інтелект також відіграє важливу роль у розробці відеоігор: процедурна генерація контенту, що набула поширення завдяки таким іграм, як Minecraft і No Man's Sky, не вимагає ручного втручання розробника і дає змогу створювати унікальні ландшафти та особливості. Даючи змогу створювати величезні світи, штучний інтелект використовується для генерації рівнів, сюжетів, діалогів та інших елементів, які роблять ігри більш різноманітними і непередбачуваними. Інструменти машинного навчання допомагають розробникам оптимізувати графіку, створювати реалістичну анімацію і моделювати поведінку персонажів.

Такі техніки, як нейронні мережі, також використовуються для поліпшення фізики ігор, моделювання поведінки натовпу і навіть для симуляції природних явищ. Наприклад, сучасні інструменти дають змогу автоматизувати тестування ігор, знаходити помилки в коді та передбачати реакцію користувачів на ті чи інші аспекти гри. Важливим результатом є впровадження адаптивних систем штучного

інтелекту, що забезпечують персоналізований ігровий досвід. Це дає змогу грі підлаштовувати складність, поведінку NPC або навіть сюжет відповідно до стилю гри користувача. Така система аналізує поведінку гравця, його помилки та успіхи і коригує ігровий процес, щоб забезпечити оптимальний баланс між складністю і задоволенням. Подібні механізми вже використовуються в багатьох популярних іграх, зокрема в серії Assassin's Creed. Штучний інтелект також допоміг створити нові жанри ігрових продуктів. Наприклад, інтерактивні історії, такі як Detroit: Become Human, зосереджені на виборі гравця, який впливає на розвиток сюжету. У таких іграх штучний інтелект аналізує рішення та пропонує різні події, створюючи унікальний досвід для кожного користувача. Ще один приклад – VR-ігри, де штучний інтелект допомагає створювати глибоко інтерактивні середовища, які в реальному часі реагують на рухи, голоси та дії гравця.

У майбутньому Штучний інтелект відіграватиме дедалі більшу роль у відеоіграх, і нещодавні роботи над генеративними моделями, такими як GPT і DALL-E, можуть зробити ігровий досвід ще більш динамічним і персоналізованим. Наприклад, такі моделі, як GPT, уже використовуються для створення діалогів для NPC, що дає змогу зробити їх природнішими та багатограннішими. Ігри стануть дуже інтерактивними, даючи змогу гравцям створювати власні світи, сценарії та персонажів у режимі реального часу за допомогою голосових і текстових команд.

Таким чином, інтеграція штучного інтелекту у відеоігри перетворила ігровий процес на одну з найбільш інноваційних форм розваг. Від найпростіших сценарних алгоритмів до складних адаптивних систем – штучного інтелетку змінив уявлення про відеоігри, їхній дизайн і сприйняття. Він дав змогу створити яскраві світи, динамічних персонажів та унікальні історії, які відповідають очікуванням і потребам сучасних гравців. У майбутньому можна буде очікувати подальших досягнень, які зроблять відеоігри невід'ємною частиною цифрового світу [3].

1.3. Генерація контенту за допомогою штучного інтелекту

За допомогою сервісів штучного інтелекту можна створювати ігрові світи, сценарії, персонажів і навіть музику, забезпечуючи високий рівень різноманітності та унікальності ігрового процесу. Використовуючи Машинне навчання та алгоритми глибокого навчання, розробники можуть автоматизувати процеси, які займають місяці, якщо не роки. Одним з важливих прикладів використання генерації контенту є процедурна генерація. Таким чином, можна створити величезний ігровий світ з мінімальними витратами часу і ресурсів. Наприклад, у грі No Man's Sky було створено понад 18 планет розміром у квінтильйон за допомогою алгоритму процедурної генерації. Кожна планета має свій власний ландшафт, флору і фауну, а також Клімат, що робить вивчення гри практично нескінченним.

Ще одним прикладом є гра Minecraft, де світ генерується в режимі реального часу відповідно до дій гравця та забезпечує унікальний досвід для кожного.

На Заході Surface and AI Microsoft показала, як інтегрувати Copilot в Minecraft (рисунок 1.1).

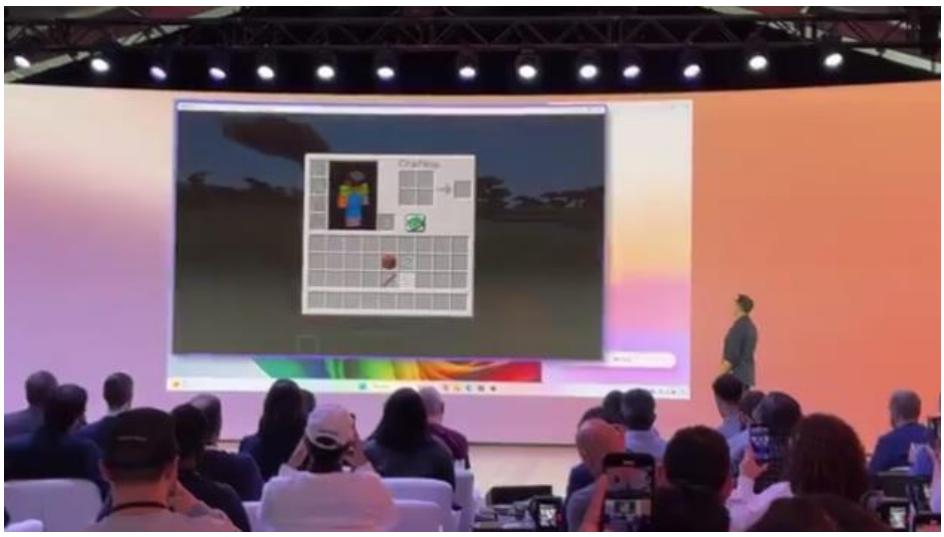

Рисунок 1.1 – Інтеграція Copilot в Minecraft

Ця функція дозволяє гравцям взаємодіяти з грою, використовуючи природну мову. Наприклад, можна запитати, як створити меч, і другий пілот підбере потрібний матеріал або пояснить процес створення. Це нововведення радикально змінює підхід до гри, дозволяючи отримувати інформацію безпосередньо під час гри [5].

Однак така інтеграція може вплинути на баланс деяких ігор. У минулому розробники витрачали багато часу на балансування API для виправлень, що автоматизують ігрові механіки, як у World of Warcraft. Якщо додати, другий пілот стане важливим помічником, оскільки це може полегшити гру або порушити баланс. Крім того, можливість отримувати відповіді безпосередньо в грі може негативно позначитися на фінансуванні веб-сайтів і баз даних, які раніше надавали таку інформацію. Це може погіршити якість контенту на платформі та вплинути на доступність точних даних в штучному інтелекті.

Генерація контенту також використовується для створення реалістичних персонажів і сюжетів. Завдяки алгоритмам штучного інтелекту персонажі можуть адаптуватися до поведінки гравця, змінювати підказки і навіть еволюціонувати по ходу гри. Наприклад, в грі The Elder Scrolls V: Skyrim неігрові персонажі мають складний розпорядок дня, реагують на дії гравця і навіть можуть починати власні пригоди. Інтерактивність таких персонажів дозволяє гравцеві глибше зануритися в гру. У іграх з відкритим світом генерація контенту допомагає створити динамічне середовище, яке реагує на дії гравця. Наприклад, у Red Dead Redemption 2 навколишнє середовище змінюється залежно від пори року, погоди та поведінки персонажа. Такі деталі додають реалістичності і роблять гру більш захоплюючою.

Штучний інтелект також використовується для створення діалогів і сюжетів. Модель генерації, така як GPT, може бути використана для створення динамічних взаємодій, які адаптуються до контексту гри. Це дозволяє розробникам створювати інтерактивні історії, в яких вибір гравця впливає на розвиток подій. Детройт: у людиноподібній грі система вибору гравців пропонує безліч закінчень та історій, що робить кожне проходження унікальним. Штучний інтелект також активно

використовується для створення музики і звукових ефектів. Завдяки алгоритмам машинного навчання розробники можуть створювати саундтреки, які змінюються залежно від ігрових подій. Це додає емоційної глибини та робить гру веселішою. Наприклад, в грі "подорож" музика динамічно змінюється в залежності від поведінки гравця і внутрішньоігрових ситуацій, створюючи незабутні емоційні враження.

Однією з ключових переваг використання сервісів штучного інтелекту для створення контенту є економія ресурсів. Розробка великого ігрового світу вручну займає досить багато часу і грошей. Завдяки штучному інтелекту цей процес буде набагато швидшим і дешевшим. Наприклад, в грі Assassin's Creed Unity розробники можуть використовувати алгоритми для автоматичної генерації архітектурних деталей, щоб створити величезний реалістичний світ з мінімальними витратами. Штучний інтелект також може допомогти у створенні текстур та моделей. Алгоритми глибокого навчання дозволяють створювати високоякісні текстури, які виглядають реалістично і відповідають стилю розробленої гри. Це дозволяє значно скоротити обсяг ручної роботи і зосередитися на інших аспектах розробки.

Використання штучного інтелекту для створення контенту відкриває нові можливості для розробників. Це дозволяє створювати ігри з високим рівнем іграбельностю, динамічним світом і глибоким сюжетом. Але є і проблеми, пов'язані з використанням цієї технології. Наприклад, створення контенту може призвести до неякісних або повторюваних елементів, які можуть негативно вплинути на сприйняття гравцем. Також важливо враховувати етичні аспекти використання штучного інтелекту, особливо при створенні персонажів і сюжетів. Незважаючи на ці проблеми, потенціал використання штучного інтелекту для створення контенту є дуже перспективним. У майбутньому технологія штучного інтелекту може повністю змінити спосіб створення ігор, пропонуючи гравцям по всьому світу новий рівень занурення та інтерактивності. Цей процес заснований на взаємодії гравця з ігровим світом, і NPC (неігровий персонаж) грає в ньому одну з центральних ролей.1 з розвитком штучного інтелекту неігрові персонажі перейшли

від простих фіксованих сценаріїв до складних систем, які можуть імітувати поведінку людини, реагувати на поведінку гравця та адаптуватися до змін в ігровому середовищі.

Перші неігрові персонажі у відеоіграх були досить примітивними. Їх поведінка базувалася на жорстко закодованих алгоритмах, які виконують фіксовані команди. Наприклад, в таких іграх, як Pac-Man, привиди переслідували героя відповідно до чітко прописаних правил. У той час це виглядало інноваційно, але рівень інтерактивності був мінімальним. Гравець не міг ні впливати на поведінку NPC, ні адаптуватися до нього.

Потім, у 1980-х та 1990-х роках, впровадження більш складних алгоритмів зробило NPC більш інтерактивними. Наприклад, в грі серії Legend of Zelda неігрові персонажі здатні реагувати на дії гравця, надаючи йому завдання і корисну інформацію. Це стало першим кроком до створення більш інтерактивного ігрового світу. Приблизно в той же час такі стратегії, як civilization, використовували штучний інтелект для створення комп'ютерних суперників, які могли приймати рішення, змагаючись з гравцями. Ці персонажі діяли за певною стратегією, демонструючи елемент адаптивності та значно підвищуючи складність гри.

На початку 2000-х розвиток NPC вийшов на новий рівень. Гра F. E. A. R. дозволяє ворожим ботам оцінювати умови бою і приймати рішення в режимі реального часу. Вони використовували прикриття, координували атаку і намагалися перейти на бік гравця. Така поведінка стала можливою завдяки складним сценаріям та правилам, що враховують багато змінних. Це створювало ілюзію справжнього тактичного мислення і значно збільшувало занурення в ігровий процес.

З розвитком обчислювальних потужностей неігрові персонажі в сучасних іграх стали ще більш реалістичними. У грі The Last Of Us Part II неігрові персонажі демонструють складну поведінку, включаючи емоції, спогади і соціальні зв'язки. Вороги можуть оплакувати полеглих товаришів, а їх дії координуються для досягнення спільної мети. У Red Dead Redemption 2 NPC має власний графік, який

дозволяє їм виконувати щоденні завдання та реагувати на дії гравця, такі як привітання та погрози. Така складність дозволяє створити яскравий ігровий світ, який здається природним і автентичним.

Системи навчання штучного інтелекту внесли особливий внесок у розвиток неігрових персонажів. Наприклад, в грі Middle-Earth: Shadow of Mordor реалізована система nemesis, яка дозволяє ворогові запам'ятовувати попередні взаємодії з гравцем. Якщо гравець не добив ворога, він може повернутися пізніше з новими навичками, зміненою зовнішністю і бажанням помститися. Це додає глибини геймплею, оскільки кожен NPC стає унікальним і важливим для сюжету.

Сучасні технології також дозволяють персонажам взаємодіяти не тільки з гравцями, але і один з одним. У грі The Sims у кожного персонажа є свої потреби, бажання і взаємини з іншими персонажами. Їх поведінка базується на складних моделях, що імітують соціальні взаємодії, такі як дружба, любов та конфлікти. Це створює динамічний ігровий процес, в якому події розгортаються органічно, без безпосереднього втручання гравця.

З розвитком віртуальної реальності (VR) роль неігрових персонажів стала ще більш важливою. У VR-іграх гравець знаходиться в тривимірному середовищі, де найдрібніші деталі можуть вплинути на відчуття занурення. Неігрові персонажі в таких іграх повинні бути максимально реалістичними, щоб уникнути "ефекту долини", який викликає дискомфорт через неприродну поведінку персонажа. Наприклад, в грі Half-Life: Alyx взаємодія з неігровими персонажами дуже переконливо, оскільки вони демонструють природні рухи, зоровий контакт і міміку.

Важливим аспектом сучасних неігрових персонажів є їх здатність адаптуватися до різних ігрових стилів. Багатокористувацькі ігри, такі як Overwatch, використовують штучний інтелект для аналізу поведінки гравця та вибору команди чи суперника, які найкраще відповідають рівню. Це дозволяє створювати збалансований і цікавий ігровий процес.

Інтерактивність та майбутнє неігрових персонажів тісно пов'язані з розробкою моделей штучного інтелекту, таких як GPT. Ці технології дозволяють створювати динамічні взаємодії, які змінюються залежно від контексту та дій гравця. Уявіть собі гру, де кожен NPC може вести невимушену розмову, розповідати унікальну історію або імпровізувати у відповідь на непередбачувану поведінку гравця. Це відкриває нові горизонти в ігровому дизайні, створюючи можливості для більш глибокого занурення і персоналізованого досвіду.

Ще одним перспективним напрямком є використання квантових обчислень для створення дуже складних моделей NPC. Це дозволяє обробляти величезні обсяги даних в режимі реального часу і створювати персонажів з рівнем складності, що перевищує сучасні стандарти. Наприклад, NPC може враховувати сотні факторів, таких як емоційний стан гравця, контекст ситуації та навіть культурні особливості.

Незважаючи на всі переваги, впровадження штучного інтелекту в NPC має певні труднощі. 1 з них-це етичне питання. Як зробити так, щоб NPC не тільки розважав, але й не мав негативного впливу на гравця. 1. Ще одна проблема полягає в тому, що вам потрібно оптимізувати свої ресурси. Навіть найкращі моделі штучного інтелекту можуть стати марними та недоступними для масового ринку, якщо вони вимагають занадто великих обчислювальних потужностей.

Вплив штучного інтелекту на інтерактивність та неігрових персонажів є революційним. Ці технології дозволяють створювати більш реалістичний, адаптивний і інтерактивний ігровий світ, який дарує кожному гравцеві унікальний досвід. У майбутньому роль неігрових персонажів буде тільки зростати, і вони стануть ще більш невід'ємною частиною відеоігор [6].

1.4. Висновок до першого розділу

У першому розділі кваліфікаційної роботи досліджується вплив штучного інтелекту на ігрову індустрію та встановлюється методологія розробки ігрових додатків у сучасних умовах. Розглядається, як штучний змінив підхід до створення ігор, поліпшивши їхню якість і розширивши можливості, а також аналізує нові можливості для розробників.

Одним із важливих аспектів впливу штучного є автоматизація таких рутинних завдань, як тестування програмного забезпечення, оптимізація коду та передбачення помилок. автоматизація. Ці технології значно скорочують час і вартість розробки, дозволяючи розробникам зосередитися на творчих і стратегічних аспектах. Крім того, алгоритми машинного навчання дають змогу обробляти великі обсяги даних, виявляти приховані закономірності та робити прогнози, забезпечуючи новий рівень аналізу в процесі розробки ігор.

У цьому розділі було розкрито те, як розробки в галузі ШІ змінили спосіб взаємодії гравців з ігровим світом. Від базових алгоритмів управління NPC у ранніх іграх до сучасних адаптивних систем, здатних створювати індивідуальні сценарії, ШІ значно підвищив рівень реалістичності та інтерактивності. Такі приклади, як NPC у The Last of Us Part II і Red Dead Redemption 2, демонструють можливості навчання моделей для врахування поведінки гравця і створення складної поведінки персонажів, яка змінюється залежно від ситуації.

Одна з центральних тем – процедурна генерація ігрового контенту, яка дає змогу гравцям створювати нескінченну кількість унікальних світів. Це не тільки розширює можливості гри, а й персоналізує ігровий досвід. аі алгоритми також використовуються для аналізу поведінки гравців, оптимізації підбору гравців і підтримки балансу в багатокористувацьких іграх, підвищуючи справедливість і задоволення від гри.

Технологічні розробки дають змогу домогтися глибшої персоналізації. У сучасних іграх ШІ підлаштовується під стиль кожного гравця, пропонуючи

індивідуальні завдання, маршрути і варіанти сюжету. Це дає унікальний досвід, коли гра ”вчиться” на поведінці гравця.

Незважаючи на численні переваги, впровадження ШІ в ігрову індустрію пов'язане і з проблемами. Зокрема, це висока вартість розробки, потреба в значних обчислювальних ресурсах і складність створення реалістичних NPC. Особлива увага також приділяється етичним аспектам, таким як використання даних гравців для навчання моделей.

У підсумку наголошено на потенціалі подальшого розвитку ШІ в ігровій індустрії. Генеративне моделювання, квантові обчислення та інші нові технології обіцяють зробити ігри інтерактивнішими, адаптивнішими та реалістичнішими, створюючи незабутні враження для гравців і розсуваючи межі можливого у світі цифрових розваг.

РОЗДІЛ 2. МЕТОДИ ТА АЛГОРИТМИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ІГРОВОГО ЗАСТОСУНКУ

2.1. Опис алгоритмів розробки ігрового застосунку

Розробка ігрового застосунку включала в себе реалізацію широкого спектра алгоритмів, які забезпечували як функціональність і оптимізацію продуктивності базових механік, так і створення глибокого інтерактивного ігрового досвіду. Алгоритми були розділені на кілька сфер, включно з керуванням персонажами, взаємодією з об'єктами, логікою поведінки ворогів, процедурною генерацією контенту, оптимізацією продуктивності та інтеграцією штучного інтелекту. Кожна з цих сфер вимагала розробки унікального рішення, що відповідає цілям проєкту.

Поведінка ворогів була заснована на машині станів, яка моделювала поведінку в різних ситуаціях. Кожен ворог мав певний стан, наприклад «у патрулі», “атакує”, “переслідує” або “чекає”. Ці стани змінювалися залежно від умов, таких як наближення гравця, нанесення шкоди або виконання завдання. Алгоритм “патрулювання” переміщав ворога за певним маршрутом. Для цього використовувалася система шляхових точок, ворог переміщався між ними, перевіряючи, чи немає зіткнень із перешкодами та межами рівня. Коли гравець потрапляв у зону видимості противника, статус змінювався на «стеження», і ворог починав рухатися до гравця, використовуючи алгоритм пошуку найкоротшого шляху. Цей алгоритм був заснований на методах графічного пошуку, таких як A* (Easter), що дають змогу визначити найкращий маршрут між двома точками з урахуванням перешкод, і використання A* давало змогу гравцеві швидко і точно знаходити маршрут навіть на складних рівнях.

Алгоритм оптимізації продуктивності включав механізм кешування об'єктів, який давав змогу уникнути повторного створення одних і тих самих елементів у грі. Це ставало особливо важливим на великих рівнях з великою кількістю об'єктів, що повторюються, як-от платформи та вороги. Кешування було реалізовано за допомогою пулів об'єктів. Коли об'єкт більше не використовувався, він не

знищувався, а повертався в пул для наступного використання. Це значно скоротило кількість операцій зі створення та видалення об'єктів, що підвищило продуктивність і знизило навантаження на систему. Крім того, було впроваджено алгоритм динамічного завантаження об'єктів, завдяки якому активними ставали тільки ті елементи рівня, які перебувають у прямій видимості гравця. Це дало змогу знизити споживання пам'яті та забезпечити стабільну частоту кадрів навіть на пристроях з обмеженими ресурсами.

Одним з інноваційних аспектів цієї роботи є використання технології штучного інтелекту для створення логіки поведінки ігрових об’єктів. Для автоматизації написання коду використовувався GitHub Copilot. Цей інструмент аналізував контекст написаних скриптів і пропонував оптимізоване рішення, що значно скорочувало час розробки. Крім того, Copilot виявляв потенційні помилки і пропонував виправлення в режимі реального часу.

Алгоритм поведінки головного героя під час зустрічі з об'єктом типу зілля ґрунтується на складних взаємодіях між фізикою, ігровою логікою та ігровими компонентами, що відповідають за оновлення стану персонажа. Коли протагоніст стикається з зіллям, спрацьовує тригер, який перевіряє, чи є об'єкт інтерактивним зіллям чи ні. Цей тригер запускається системою зіткнень і заснований на перевірці того, чи перетинаються два об'єкти в моделі зіткнень. Як тільки система вирішує, що відбулося зіткнення, викликається відповідний метод обробки взаємодії.

Об'єкти зілля зазвичай мають такі властивості, як тип зілля, ефект і тривалість дії. Коли активується тригер, гра зчитує ці дані і передає їх обробнику стану головного героя. Наприклад, якщо зілля призначене для збільшення статів, алгоритм збільшить поточний рівень статів персонажа, враховуючи максимально можливе значення. При зустрічі з зіллям, що дає тимчасову здатність, наприклад збільшення швидкості або сили, обробник активує відповідний ефект і зберігає його тривалість у таймері.

Тривалі ефекти зазвичай обробляються окремим алгоритмом, який працює покадрово або за часом. Таймер, пов'язаний з активованим ефектом, поступово

зменшується, і коли він досягає нуля, ефект закінчується, і стан персонажа повертається до початкового. Наприклад, якщо зілля збільшує швидкість, алгоритм додає модифікатор швидкості до базового значення, а коли дія зілля закінчується, цей модифікатор видаляється. Алгоритм також обробляє анімацію і звукові ефекти, пов'язані із взаємодією. Під час вживання зілля відображається коротка анімація, яка підкреслює, що зілля активне, і лунає відповідний звук. Це забезпечує естетичний і функціональний зворотний зв'язок із гравцем.

Якщо зілля є обмеженим ресурсом, то після використання його буде видалено з ігрової сцени або позначено як таке, що не підлягає повторному використанню. У таких випадках об'єкти зілля можуть бути переміщені в пул об'єктів, щоб їх можна було повторно використовувати, коли вони зустрічаються на інших рівнях або в інших умовах гри. Такий підхід дає змогу оптимізувати ресурси та знизити навантаження на систему [7].

Важливою частиною взаємодії з персонажем є обробка заскриптованих подій. Події включають активацію пасток, відкриття і закриття дверей, запуск діалогу, зміну ігрового стану тощо. Коли герой потрапляє в зону спрацьовування, алгоритм викликає сценарій, пов'язаний із відповідною подією. Наприклад, якщо герой активує важіль, запускається анімація, наприклад, піднімається міст або відкривається прохід. Ці події зазвичай містять умови спрацьовування, наявність необхідних тегів або виконання попереднього завдання. Складні події складаються з декількох етапів, наприклад, після активації важеля в сцені з'являється ворог, і гравцеві доводиться виконувати додаткові дії, щоб завершити завдання.

Маніпулювання елементами довкілля включає переміщення об'єктів, використання об'єктів і взаємодію з нестаціонарними елементами в сцені. Наприклад, персонаж може штовхнути ящик, щоб створити платформу, дістатися до високого місця або використовувати мотузку для підйому. Алгоритм визначає, чи є об'єкт таким, що взаємодіє, і перевіряє стан героя (чи перебуває він у зоні взаємодії або виконує відповідну дію). Після перевірки алгоритм прикладає до

об'єкта силу або змінює його стан. Наприклад, він може перемістити ящик у нове положення або підняти мотузку.

2.2. Опис методів розробки ігрового застосунку

1. Для розробки нової версії ігрового застосунку спершу був використаний метод прототипування – це підхід до створення спрощеної версії гри та тестування ключових механік, ідей і концепцій. Це процес спрощення ігрової системи до мінімально необхідної функціональності та оцінки того, чи працює задуманий геймплей, чи відповідає він очікуванням гравців і чи немає якихось істотних технічних або дизайнерських проблем [8]. Мета підходу – якомога швидше створити ігрову модель, яка допоможе протестувати ключові елементи гри, як-от рух персонажа, фізика, базова механіка (стрибки, взаємодія з об'єктами тощо), дизайн рівнів і баланс складності. Прототипи не фокусуються на таких деталях, як графіка, звук або анімація. Замість цього вони використовують прості форми, базові текстури та інструменти, які можна легко змінити. Таким чином, створення прототипів – це ітеративний і гнучкий процес. Можна не лише виявляти проблеми, а й експериментувати з механізмами, додавати або змінювати їх на основі отриманих результатів. Тому воно є важливим інструментом для створення ігор, що поєднують творчість і технології.

2. Після створення базового прототипу і механік гри, був використаний модульний підхід до розробки – цей підхід розділяє проєкт на незалежні компоненти (рисунок 2.1), кожен з яких відповідає за певну функцію, у контексті 2D-платформерів такий підхід дає змогу розробникам зосередитися на конкретних аспектах гри, таких як рух персонажів, фізика, поведінка ворогів, дизайн рівнів та звук. аспекти гри. Кожен модуль розробляється окремо, має чітко визначене призначення і взаємодіє з іншими компонентами через зрозумілий інтерфейс [9]. В застосунку модуль руху персонажа обробляє дані, які вводить гравець, і розраховує

зміни положення на основі швидкості та гравітації, а модуль фізики перевіряє зіткнення і взаємодію платформ.

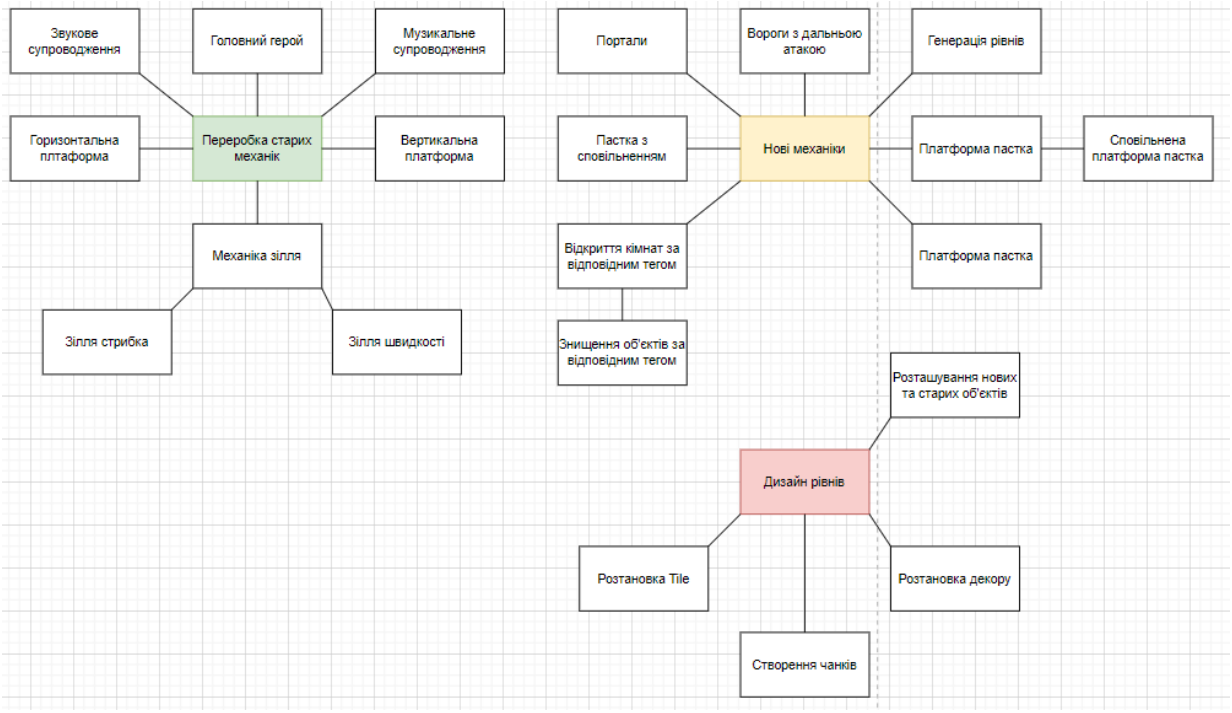

Рисунок 2.1 – Блок-схема модульного підходу до розробки

Процес починається з визначення основних функціональних частин гри, які необхідно реалізувати. Для кожного модуля створюється окремий набір правил і механік, і його реалізація проходить етап тестування і доопрацювання без впливу на іншу частину проєкту. Це дає змогу виявляти й усувати проблеми на ранній стадії, роблячи код гри більш структурованим і простим у супроводі. Інтеграція модулів має відбуватися поетапно. Усі частини системи пов'язані між собою таким чином, що кожен компонент взаємодіє з іншими в такий спосіб, що залишається незалежним і не порушує цілісність гри. Якщо одна частина гри потребує поліпшення, її можна змінити, не зачіпаючи інші частини. Це також полегшує повторне використання модулів у майбутніх проєктах, оскільки кожен компонент створюється як універсальне рішення для конкретного завдання. Таким чином,

можна оптимізувати процес розробки, знизити ризики та спростити тестування ігор.

3. Після успішного тестування основних механік ігрового застосунку було використано підхід рівневого дизайну На цьому етапі відпрацьовується баланс складності, вводяться нові виклики для гравця та додаються унікальні механіки, які роблять кожен рівень цікавим [10]. Створення рівнів передбачає врахування низки елементів, зокрема розташування платформ, розміщення ворогів, пасток і корисних предметів, таких як бонуси чи підсилення. Балансування цих елементів є ключовим для підтримки інтересу гравця: рівень не має бути надто простим або занадто складним. Зазвичай, у початкових частинах гри рівні служать для навчання основним механікам, поступово додаючи нові елементи, які ускладнюють гру.

Використання тайлсетів або інструментів, таких як Tiled, допомагає автоматизувати створення рівнів.

Естетика – ще один важливий аспект дизайну рівнів. Навіть у 2D-платформерах візуальний стиль рівня може мати значний вплив на загальне враження від гри. Тематичні елементи, як-от колірна палітра, текстури, освітлення та звук, допомагають створити атмосферу, що підсилює емоційні переживання гравця.

4. Для розробки скриптів застосунку використовувався алгоритм “ Agile ”– це забезпечило гнучкість і поетапність процесу створення гри. Розробка була розділена на кілька спринтів, кожен з яких мав чітко визначені цілі та результати. Це дозволило вчасно реагувати на зміни, вносити покращення на основі зворотного зв'язку та ефективно розподіляти ресурси між різними аспектами гри.

На ранніх етапах гнучкий підхід дозволив створити базовий прототип, який включав ключові ігрові механіки, такі як рух персонажа, стрибки та взаємодію з платформою. Прототип був протестований, щоб оцінити, наскільки комфортно гравцеві буде взаємодіяти з ігровим середовищем. Отримавши результат, було

покращено фізику руху та узгоджено швидкість і висоту стрибків із загальною динамікою гри.

На наступному етапі Agile поступово розширювався функціонал гри. В рамках одного спринту були додані вороги з різними типами поведінки, такими як патрулювання, переслідування гравців і стрілянина. Це урізноманітнило ігровий процес і додало виклику гравцеві. У наступному спринті фокус змістився на створення дизайну рівнів, який включав як лінійні, так і нелінійні елементи. Рівні ставали дедалі складнішими, з'являлися нові механіки, такі як рухомі платформи, пастки та секретні зони. Кожен спринт закінчувався етапом тестування, під час якого отримувався зворотній зв'язок. Наприклад, під час тестування було зазначено, що деякі пастки були занадто складними, тому в стратегічних місцях рівня були додані контрольні точки. Завдяки Agile ці зміни були впроваджені швидко, не впливаючи на інші аспекти гри. Agile також уможливив паралельну роботу над різними елементами гри. Фінальні етапи розробки також відбувалися за гнучкими принципами. Завдяки ітеративному процесу вдосконалення візуального стилю, виправлення дрібних помилок та покращення механіки на основі останніх тестів, гру було доведено до статусу релізу. Завдяки Agile було створено продукт, який відповідав сучасним стандартам та забезпечував захопливий ігровий досвід [11].

2.3. Концепція гри та метод дизайну рівнів

1. Концепція гри являє собою загальну ідею, що лежить в основі дизайну гри і визначає, як гравець взаємодіє з ігровим світом. Вона охоплює такі важливі аспекти, як історія, жанр, механіка, стиль, цільова аудиторія та загальна атмосфера гри. Концепція створює фундамент, на якому будується ігровий досвід, і визначає, які емоції та виклики відчуватиме гравець [12].

Наприклад, у грі “Celeste”, – це зворушлива історія про ретельний платформер і конфлікт із самим собою. Гравці долають складні рівні, з акцентом на технічну досконалість і здатність швидко адаптуватися до нових механік [13].

Ігри, подібні до гри “Hollow Knight”, можуть зосереджуватися на дослідженні величезного навколишнього світу. Головний герой відкриває в собі нові здібності, які дозволяють йому повертатися в раніше недоступні області. Ця концепція добре підходить для ігор, в яких протагоніст досліджує підводні царства, бореться з морськими істотами і вирішує головоломки, пов'язані з зануренням вглиб океану [14].

Наслідуючи гру “Limbo”, платформер можна створити в монохромному візуальному стилі, де головний герой мандрує похмурим світом, сповненим смертельних пасток. Також можна використовувати фізично реалістичну механіку, використовуючи гравітацію та інерцію для маніпулювання об'єктами, щоб рухатися вперед [15].

Гра “Super Meat Boy” задала тренд на хардкорні платформери, в яких гравцям доводилося блискавично долати небезпеки [16].

Схожою грою є гра “Robot Hero”, де гравця випробовують на групі небезпечних фабрик. Рівні сповнені лазерів, шипів і рухомих риштувань, що змушує гравців постійно вдосконалювати свої навички [17].

Для розробленого додатку було створено концепцію захопливої подорожі по світу середньовічної фентезі, де гравець втілює образ хороброго лицаря. У сетингу стародавнього королівства, наповненого загадками і небезпеками, герой вирушає у подорож, щоб виконати свою місію. Спочатку гравець опиниться в світлих локаціях де буде по троху ознайомлюватися з основними механіками гри, а потім локації будуть ставати все темніше, а разом з кольоровою палітрою локацій буде змінюватися і складність гри.

Ігровий процес побудований на платформерній механіці, що вимагає від гравця спритності, реакції та стратегічного мислення. Лицар долає перешкоди,

стрибає між платформами і уникає пасток, в порівнянні з минулою версією гри гейплей був зроблений більш динамічним та плавним.

Графічний стиль гри використовує теплі кольори для мирних приміщень і темні – для підземель і полів битв, створюючи середньовічну атмосферу. Музика та звукові ефекти підсилюють враження від гри, відображаючи моменти урочистості, небезпеки та перемоги.

Дизайн ворогів полягає у використанні доступних нам спрайтів з Unity Asset Store та гармонічному вписані їх в сетинг гри.

Частину NPC або ворогів було створено за допомогою редагуванню базової моделі головного героя в Adobe Photoshop (рисунок 2.2), таким чином можна зекономити ресурси на створені додаткових ворогів або об’єктів для фону, також за допомогою чорного кольору показаний негативний настрій і ворожість до головного герою (гравця).

Рисунок 2.2 – Спрайти для реалізації нових ворогів або NPC

NPC будуть зустрічати гравця наприкінці натякаючи гравцю, що рівень вже скоро закінчиться, також на те куди треба йти на рівні гравцю будуть натякати вказівні знаки, а щоб не починати гру з самого початку головного героя чекають спеціально розташовані об’єкти (checkpoints), які будуть зберігати прогрес гравця.

Далі головному герою потрібно було створити перешкоди на шляху, для цього було розроблено основний концепт (рисунок 2.3) для більшості перешкод і ворогів та реалізовано його, 2D-платформери славляться великим різноманіттям

своїх перешкод, від простих шипів до летючих платформ або снарядів які можуть представляти небезпеку для головного героя.

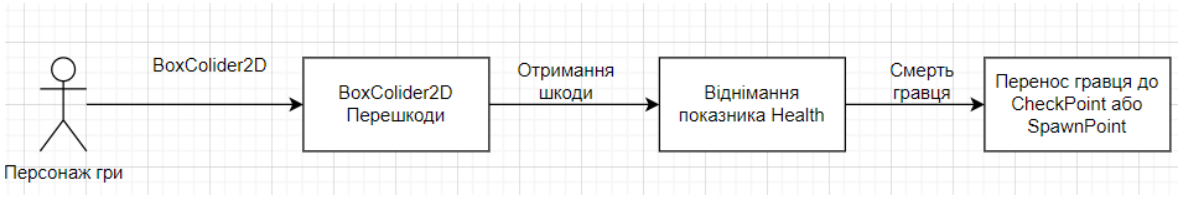

Рисунок 2.3 – Блок-схема роботи більшості перешкод

Завдяки цьому концепту було розроблено такі перешкоди: колючки, платформи з шипами, висувні шипи, снаряди, зони в які гравець не може заходити.

Одними з важливих елементів супроводу гравця під час проходження рівня – це музикальне та звукове супроводження, для 2D-платформера стандартом є піксельна музика. Для реалізації звукового супроводження гри було впроваджено відповідні піксельні музикальні композиції, а дії героя та інших елементів гри супроводжуються різними звуковими ефектами. Також елементом, який доповнює атмосферу гри, можна вважати декор-елементи візуального оформлення, гармонійний і продуманий декор допомагає передати характер локації (рисунок 2.4).

Рисунок 2.4 – Приклад декору на світлих локаціях

2. Для створення дизайну рівнів був використаний нелінійний підхід. Нелінійний підхід до дизайну рівнів у відеоіграх означає створення відкритого, гнучкого та різноманітного ігрового простору, де гравець має змогу обирати шляхи та способи досягнення мети. В контексті 2D-платформерів це означає, що рівні слідують за суворою лінійною послідовністю, а не слідують за однією, тобто пропонують кілька варіантів розвитку подій та дають гравцю більше свободи й варіативності в ухваленні рішень. Нелінійний дизайн рівнів дає змогу гравцеві мати кілька маршрутів для досягнення мети, кожен з яких має свої особливості. Це надає геймплею додаткової глибини та дає змогу гравцям обирати різні стилі гри. Іноді нелінійність включає в себе дослідження секретних зон або додаткові завдання, які спонукають гравців досліджувати ігровий світ. Одна з головних переваг нелінійного дизайну полягає в тому, що він забезпечує більшу гнучкість і спонукає гравців досліджувати ігровий світ. Нелінійність дає змогу створити відкритий, живий простір, у якому гравці можуть повернутися до попередніх частин рівня або навіть змінити порядок своїх дій. Це дає змогу створювати складніші та насиченіші ігрові світи, де кожна взаємодія з оточенням має новий результат. Такі рівні можуть містити різні шляхи досягнення однієї й тієї самої мети. Важливим елементом також стануть секретні пошуки, що відкривають додаткові бонуси, нові рівні та інші сюрпризи. Нелінійний підхід дає змогу створити складніший та інтерактивніший ігровий світ, де кожен рівень пропонує кілька варіантів розвитку подій і результатів. Це не тільки дає гравцеві свободу дій, а й сприяє високому рівню реіграбельності, оскільки гравці можуть повернутися, щоб спробувати нові маршрути та відкрити нові секрети. Ще одна важлива складова нелінійного дизайну – різноманітність підходів до виконання завдань. Гравці можуть обирати не лише шляхи, а й різні стратегії для подолання тих чи інших перешкод. Наприклад, вони можуть не лише проходити рівні, а й використовувати певні ресурси та механіки, щоб обдурити ворога або активувати певні події в ігровому світі. З погляду розробника, нелінійний дизайн потребує ретельнішого планування

та тестування. Оскільки кожен рівень має кілька варіантів розвитку подій, важливо переконатися, що всі шляхи збалансовані, цікаві та не призводять до помилок або проблем із проходженням. Також необхідно створити певні індикатори та підказки, щоб гравці могли орієнтуватися в цьому просторі, не гублячись у безлічі варіантів.

Для дизайну рівнів був використаний інструмент TileMap (рисунок 2.6), що є потужним інструментом для створення 2D-рівнів, заснованих на використанні плиток. Плитки – це невеликі графічні елементи, які багаторазово розміщуються на сітці для створення ігрового світу. Інструмент дає змогу легко збирати складні сцени з простих елементів, як-от платформи, стіни та ландшафти. TileMap працює за сіткою, кожна комірка якої відповідає одному елементу. Розмір цієї сітки можна змінювати залежно від стилю гри.

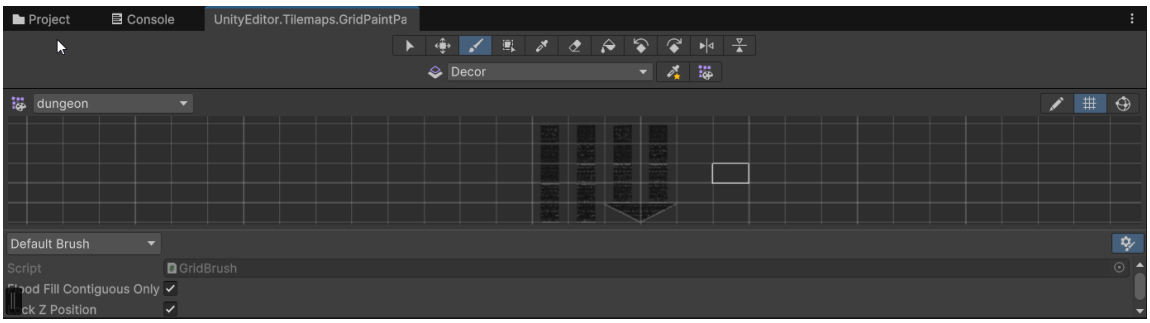

Рисунок 2.6 – Робота з інструментом TileMap

Щоб працювати з плитками, використано палітру Tile. Палітра плитки діє як віртуальна палітра, з якої плитки перетягуються і розміщуються на сцені. До самих плиток можна додавати такі властивості, як колайдери для фізичної взаємодії або анімації для створення елементів, що рухаються. За допомогою TileMap було створено окремі шари для різних типів об'єктів, як-от передній план, платформа, тло й декоративні елементи для створення багатошарових рівнів. Вона також підтримує функцію автозаповнення, яка спрощує процес малювання, автоматично вибираючи відповідні плитки залежно від навколишнього оточення. Інтегруючись з іншими компонентами Unity, TileMap забезпечує безшовну взаємодію з фізикою, системами освітлення і колайдерами для створення інтерактивних оточень.

Інструмент ідеально підходить для 2D-платформерів, головоломок, рольових ігор і будь-яких ігор, в яких потрібні повторювані елементи.

Для того, щоб персонаж міг стояти на землі, до елементу TileMap був доданий Tilemap Colider 2D – компонент у Unity, який використовується для додавання фізичних властивостей до плиток на TileMap.

Також для гравців які вже пройшли гру був створений секретний рівень з рандомною генерацією сцени, його скрипт (Додаток А) працює таким чином, що коли починається гра, починає свою роботу Start, яка викликається під час запуску сцени. У цій функції послідовно виконуються два основні методи: GenerateLevel і SpawnPlayer. Спочатку метод GenerateLevel створює рівень із заздалегідь підготовлених чанків, які зберігаються як префаби в масиві levelChunks. Позиція, в яку поміщається перший чанк, встановлюється в початкову точку Vector3.zero, а потім зміщується на фіксоване значення, що визначається параметром chunkWidth на кожній ітерації. Для кожного чанка скрипт випадковим чином обирає один із доступних префабів за допомогою методу Random.Range і поміщає його у світ за допомогою Instantiate. Після того як рівень згенеровано, викликається метод SpawnPlayer для розміщення гравця. Якщо були визначені префаби гравця і точки спавна, гравець створюється в позиції, зазначеній в Instantiate. Якщо ці параметри не вказані, скрипт виводить попередження в консоль, щоб можна було усунути проблему. Завдяки такому підходу рівні генеруються динамічно, а стартова позиція гравця завжди знаходиться в певному місці, що гарантує контрольований старт. Крім того, скрипт дає змогу легко змінювати кількість і тип чанків, а також параметри їх розміщення, що робить його гнучким для різних ігрових проєктів.

2.4. Висновок до другого розділу

У другому розділі кваліфікаційної роботи було розглянуто та реалізовано методи й алгоритми, що використовуються під час розроблення сучасних ігрових додатків. Розділ присвячено створенню ефективної та функціональної архітектури програмного забезпечення, впровадженню алгоритмів для оптимізації застосунків, розробці ігрової механіки та дизайну рівнів, а також використанню новітніх інструментів розробки. Усі ці аспекти спрямовані на досягнення головної мети – створення інноваційних ігрових продуктів із високим рівнем інтерактивності, стабільності та адаптивності. Одним із найважливіших напрямів, описаних у цьому розділі, є розробка алгоритмів керування поведінкою персонажів у грі. Було реалізовано модель машини станів, яка дає змогу ворогу адаптувати свою поведінку залежно від ситуації. У гру інтегровано алгоритм пошуку шляху (A*), який розраховує оптимальну траєкторію руху противника з урахуванням перешкод на рівні. Цей механізм робить ворога більш «розумним» і здатним адаптуватися до змін в ігровому середовищі. Оптимізація продуктивності – важливий аспект розробки ігор, особливо на обмежених апаратних платформах. У цьому розділі представлено кілька важливих методів оптимізації, включно з кешуванням об'єктів і динамічним завантаженням активних елементів. Кешування об'єктів скорочує кількість операцій зі створення та видалення об'єктів, тим самим значно знижуючи навантаження на систему. Динамічне завантаження елементів гарантує, що активними будуть тільки об'єкти, які перебувають у зоні видимості гравця. Таке рішення не тільки підвищує продуктивність гри, а й забезпечує стабільну частоту кадрів навіть на пристроях з обмеженими ресурсами. Розробка ігрової механіки ґрунтувалася на прототипуванні – ефективному способі тестування ідей та їхнього ітеративного поліпшення. Спрощені версії гри були створені для тестування основних механік, як-от рух персонажа, фізика взаємодії з об'єктами, стрибки та проходження перешкод. Прототипування дало змогу виявити і швидко усунути проблеми на ранніх етапах розробки. Водночас воно давало змогу випробувати нові

механізми, підвищуючи тим самим якість кінцевого продукту. Після успішного тестування базового функціоналу було впроваджено модульну розробку. Кожен компонент гри, такий як логіка руху персонажа, фізика об'єктів, дизайн рівнів і поведінка ворогів, було реалізовано окремо та інтегровано в загальну систему. Такий підхід полегшив тестування, виявлення та усунення помилок, а також дав змогу гнучкіше розширювати функціональність. Дизайн рівнів був ще однією важливою сферою дослідження. Рівні були спроєктовані таким чином, щоб їхня складність поступово зростала для підтримки інтересу гравця. Перші рівні мали навчальний характер і допомагали гравцям освоїти базову механіку гри. На наступних рівнях поступово з'являлися нові складнощі та елементи, як-от пастки, вороги і секретні зони. При цьому враховувалися аспекти балансу складності, щоб зробити гру захопливою, але не надто складною. Рівні створювалися за допомогою таких інструментів, як Tileset і Tiled, які забезпечують гнучкість і точність у розміщенні елементів. Графічний стиль та естетика гри також зіграли важливу роль. Розроблений застосунок створював атмосферу середньовічного фентезійного світу, в якому світлі ділянки поступово змінюються темнішими, відображаючи розвиток сюжету. З цієї причини в ранніх рівнях використовувалися теплі кольори, а в підземеллях та інших небезпечних місцях – темніші відтінки. Саундтрек був інтегрований для посилення атмосфери та емоційного сприйняття гри. Музика змінювалася від місця до місця, а звукові ефекти підкреслювали ключові моменти, такі як перемога над ворогами або виявлення секретних зон. Для забезпечення ефективного управління проєктом на всіх етапах розробки було використано підхід agile.

Процес був розділений на кілька спринтів, кожен з яких мав чіткі цілі та завдання. Це дало змогу швидко реагувати на зміни, враховувати відгуки тестувальників і впроваджувати поліпшення. Наприклад, в одному зі спринтів деякі пастки в рівні були визнані занадто складними для більшості гравців, тому були додані контрольні точки. Такий підхід забезпечив не тільки високу якість гри, а й ефективний розподіл ресурсів у процесі розробки.

Порівняно з попередньою версією, ігрова механіка стала динамічнішою і плавнішою, що робить гру більш захопливою. Вороги в грі були розроблені з урахуванням доступних ресурсів у сховищі активів Unity. Їхній дизайн гармонує із середньовічною обстановкою, а їхня поведінка покликана створювати проблеми для гравця. Деякі вороги використовують дальні атаки, а інші намагаються напасти на гравця зблизька, що вимагає від гравця адаптації до різних сценаріїв. Щоб ще більше урізноманітнити ігровий процес, у грі реалізовано різноманітні типи пасток і перешкод, що додають динаміки та складності.

РОЗДІЛ 3. РОЗРОБКА НОВОЇ ВЕРСІЇ ІГРОВОГО ДОДАТКУ З ЗАСТОСУВАННЯМ СЕРВІСІВ ШТУЧНОГО ІНТЕЛЕКТУ

3.1. Аналіз та оцінка сучасних програмних та апаратних платформ для розв’язання поставлених задач інженерії програмного забезпечення

1. Основним і центральним інструментом розробки ігрового застосунку є рушій Unity 6 – один з найпопулярніших і найпотужніших рушіїв розробки ігор для створення інтерактивних додатків та ігор для різноманітних платформ.

Ця версія є великим кроком вперед у порівнянні з попередніми версіями, з багатьма новими функціями та покращеннями, спрямованими на підвищення продуктивності, якості графіки та зручності використання, в той час як Unity 6 надає професійним розробникам інструменти для створення складних проєктів, Ключовою перевагою Unity 6 є розширені можливості візуалізації. Новий рушій підтримує найсучасніші методи рендерингу, включаючи трасування променів у реальному часі, що значно підвищує рівень реалістичності сцени. Завдяки цьому розробники можуть створювати графіку, яка виглядає майже фотореалістично.

Серед нових бібліотек є Graphics Pipeline Enhancements. Бібліотека оптимізує процес рендерингу за допомогою нових графічних API, таких як Vulkan і DirectX 12, підвищуючи швидкість і якість зображення.

AI and Machine Learning Tools. За допомогою бібліотеки машинного навчання, Інтегруйте нейронні мережі та інші методи штучного інтелекту, щоб полегшити створення складної поведінки персонажів і аналіз даних у реальному часі.

Networking and Multiplayer Solutions. Ці бібліотеки забезпечують стабільне мережеве з'єднання для багатокористувацьких ігор, зменшуючи затримки та оптимізуючи передачу даних між клієнтами.

Крім того, Unity 6 підтримує високоякісні текстури з більш детальною обробкою світла і тіні, що дозволяє створювати більш глибокі, атмосферні світи. Значну увагу також було приділено оптимізації рушія: Unity 6 ефективніше

використовує апаратне забезпечення та забезпечує плавний FPS навіть на пристроях з обмеженими ресурсами.

Окремої згадки заслуговує покращена система фізики. В Unity 6 фізична модель була вдосконалена, щоб зробити взаємодію між об'єктами в ігровому світі більш реалістичною. Тепер розробники можуть використовувати покращену динаміку рідин, твердих тіл та деформацій. Ці нововведення полегшують роботу з фізичними ефектами та зменшують потребу у складному програмуванні спеціальних симуляцій Інтерфейс редактора в Unity 6 став ще більш зручним для користувача. Команда розробників внесла низку змін, спрямованих на спрощення навігації, збільшення інформативності та персоналізацію середовища розробки. Тепер користувачі можуть створювати власні панелі інструментів і налаштовувати робочий простір відповідно до своїх потреб, що підвищує продуктивність і скорочує час, необхідний для виконання рутинних завдань. Новий редактор також підтримує ширший спектр мов програмування, таких як Python та Lua, що відкриває нові можливості для інтеграції зі сторонніми інструментами.

Unity 6 приносить значні покращення у сфері штучного інтелекту. Нова система штучного інтелекту дозволяє створювати складні поведінкові моделі NPC, не вимагаючи великих обсягів кодування. Система використовує алгоритми машинного навчання для адаптації поведінки персонажів, роблячи їхню поведінку більш природною та передбачуваною. Для розробників це означає, що вони можуть створювати більш інтерактивні та глибокі ігрові світи, де NPC навчаються та змінюють свою поведінку у відповідь на дії гравця.

Одне з найважливіших нововведень – інтеграція передових аналітичних інструментів. Unity 6 надає розробникам потужну систему збору та аналізу даних про користувачів. Це дозволяє краще зрозуміти поведінку гравців, а також знайти та виправити слабкі місця в ігровому дизайні. Завдяки аналітиці розробники можуть точніше адаптувати свій контент до потреб аудиторії, підвищуючи загальне задоволення від ігрового досвіду.Unity 6 пропонує тісну інтеграцію з популярними сторонніми інструментами, такими як Blender, Photoshop і Autodesk Maya. Unity 6

підтримує тісну інтеграцію з популярними сторонніми інструментами, такими як Blender, Photoshop і Autodesk Maya. Це дозволяє розробникам легко переносити свою роботу між різними додатками, скорочуючи час, витрачений на імпорт та експорт файлів. Крім того, новий рушій розширив підтримку форматів файлів, що полегшує роботу з великою кількістю графічних і звукових ресурсів.

Велика увага в Unity 6 також була приділена питанням безпеки. Нові функції забезпечують більш високий рівень захисту як самого коду, так і даних користувача. Розробники можуть використовувати вбудовані інструменти для шифрування даних і забезпечення їх конфіденційності. Це особливо важливо для онлайн-ігор, де безпека є критично важливою [18].

1. Для контролю версій ігрового застосунку використовувалася така платформа, як GitHub – це платформа для зберігання, обміну та спільної роботи над кодом, яка є невід'ємною частиною сучасної розробки програмного забезпечення. Заснована на розподіленій системі контролю версій Git, що дає змогу користувачам ефективно відстежувати зміни коду, GitHub надає можливість створювати сховища, даючи змогу користувачам створювати спеціальні сховища для публічних або приватних проєктів. Завдяки цьому платформа є домівкою для багатьох відкритих і комерційних проєктів. Спільнота розробників активно використовує її для обговорення змін, розв'язання технічних проблем і автоматизації таких процесів, як тестування та розгортання [19].

2. Для взаємодії з сервісом GitHub був використаний відносно новий додаток GitHub Desktop – це додаток, який спрощує навігацію по Git за допомогою зручного графічного інтерфейсу. Він ідеально підходить для розробників, які не хочуть використовувати командний рядок, але хочуть мати повний доступ до репозиторію. Додаток дає змогу легко синхронізувати локальні зміни з віддаленими сховищами, переглядати історію проєктів і керувати гілками. За допомогою GitHub Desktop розробники можуть зосередитися на своєму коді та залишатися продуктивними, виконуючи основні операції в інтуїтивно зрозумілій манері [20].

3. Для розробки написання алгоритмів застосунку використовувався популярний безкоштовний редактор з відкритим кодом, створений Microsoft Visual Studio Code з вбудованими в нього сервісом GitHub Copilot. VS Code підтримує широкий спектр мов програмування, включаючи JavaScript, Python, C++, Java, TypeScript, Go та Ruby.

Редактор побудований на платформі Electron, тому він може працювати як на Windows, так і на macOS і Linux. Однією з головних особливостей VS Code є його інтеграція з IntelliSense, інструментом завершення коду, який надає підказки на основі синтаксису мови та контексту коду. Завдяки вбудованій системі налагодження розробники можуть запускати та тестувати свій код безпосередньо з редактора.

VS Code також ідеально інтегрований з Git. Розробники можуть відстежувати зміни, створювати комітети та синхронізувати проєкти зі сховищами безпосередньо в редакторі. Для спільної роботи з кодом доступна функція спільного використання в реальному часі, що дозволяє декільком людям працювати над одним проєктом в режимі реального часу.

Редактор залишається швидким та ефективним завдяки своїй архітектурі. Завдяки безлічі розширень він здатний працювати без істотних затримок, що робить його ідеальним для роботи як на потужних машинах, так і на менш продуктивному обладнанні.

Visual Studio Code – один з найкращих варіантів для розробників завдяки своїй гнучкості, універсальності та відкритому підходу до розробки. Його постійні оновлення гарантують інтеграцію нових технологій і підтримку сучасних підходів до програмування.

GitHub Copilot – це інструмент призначений для надання допомоги розробникам, надаючи контекстно-залежні рекомендації щодо написання коду. Copilot інтегрується з популярними середовищами розробки, такими як Visual Studio Code, Neovim і JetBrains IDE, виступаючи в якості інтелектуального

помічника, аналізуючи вже написаний код, надаючи фрагменти, доповнення і навіть повну функціональність.

Основна технологія, що лежить в основі Copilot, базується на більш широкій мовній моделі, зокрема GPT-4, адаптованій для розуміння програмного коду. Copilot може генерувати код на багатьох мовах програмування, включаючи Python, JavaScript, TypeScript, Java та C++. Його функції включають завершення, пропонування альтернативних рішень, опис складного коду та створення шаблонів на основі коментарів та документації.

Розробники використовують Copilot для прискорення процесу написання коду, пошуку помилок і отримання натхнення для вирішення складних технічних завдань. Цей інструмент не тільки підвищує продуктивність, але і допомагає новачкам освоїти нові підходи до програмування, дотримуючись рекомендацій другого пілота [21].

У той же час важливо відзначити, що другий пілот не замінює людини (розробника). Інструмент може робити помилки або надавати рішення, які не відповідають найкращим практикам, тому вам потрібно переглянути згенерований код у контексті ліцензування, також обговорювалося використання відкритого коду для навчання спільних моделей пілотування, але GitHub прагне до прозорості, і вони працюють над цим.

3.2. Розробка нових інтерактивних об’єктів та алгоритмів для їх реалізації

На початку створення нових ігрових механік та їх покращення було проаналізовано стару версію додатку, проаналізовано її недоліки і розроблено стратегії дій інтерактивних об’єктів (рисунок 3.1). Інтерактивні об’єкти – це елементи оточення з якими гравець може взаємодіяти і якось вплинути на них, в іграх часто додають подібні об’єкти, серед таких можна побачити всякі важелі або кнопки для активації потаємних ходів або просто для посилення головного герою.

В розробленому додатку розроблюваного додатку – це різноманітні зілля або предмети для подальшого проходження рівня.

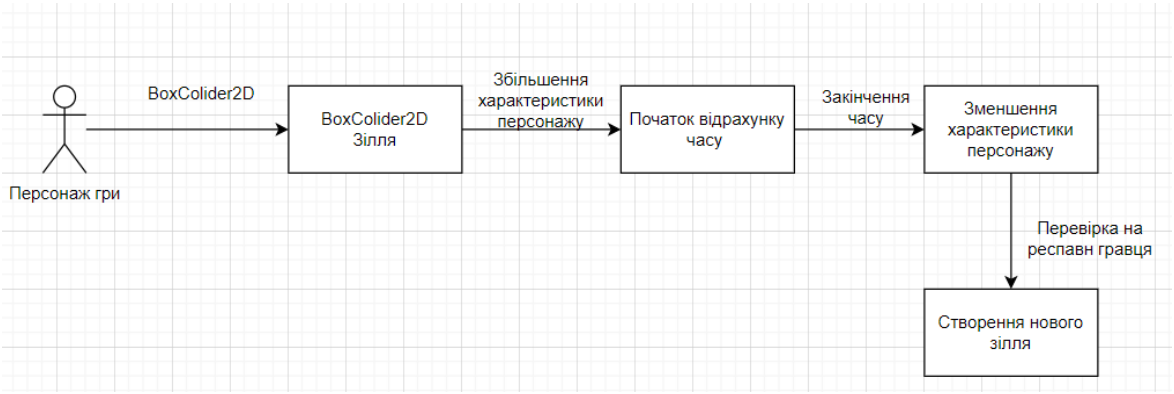

Рисунок 3.1 – Блок-схема роботи інтерактивних предметів, які впливають на характеристики персонажу

Серед інших задумок, які було реалізовано в роботі були, це потаємні кімнати, провалюючіся платформи, падаючі платформи, стріляючі вороги, телепорти, потаємні двері які відкриваються від взаємодії з іншими об`єктами, також була додана локація замку.

Логіка гри заснована на серії викликів і проходжень. Перші рівні зведені до мінімуму, щоб гравці могли ознайомитися з механікою. Згодом додаються складніші перешкоди, як-от вороги зі зростаючим рівнем сили швидкості, рухомі платформи, небезпечні пастки та головоломки. Локації розроблені таким чином, щоб мотивувати гравців досліджувати їх і знаходити приховані проходи та додаткові ресурси.

Далі було перероблено ворога (Додаток А), який патрулює певну область переміщується вперед-назад уздовж осі X. Спочатку визначається кілька змінних, як-от швидкість ворога, відстань, на яку він переміщується вліво або вправо, і компонент SpriteRenderer, який дає змогу змінювати орієнтацію ворожого спрайта. За допомогою атрибута [SerialiseField] можна задати ці параметри з інтерфейсу Unity У методі Start() обчислюються межі, до яких повернеться ворог, для цього

зберігається початкова позиція ворога по осі x і додається або віднімається відстань, на яку він може рухатися. Метод Update() який виконується, кожен кадр викликає метод Patrol(), що містить логіку руху противника. Якщо ворог рухається вправо (якщо змінна movingRight дорівнює true), його положення змінюється за допомогою transform.Translate(Vector2.right * speed * Time.deltaTime). Коли позиція ворога досягне лівої межі патруля – movingRight = true, він змінить напрямок руху, і почне рухатися вправо. Завдяки наявності у ворога скрипту Obstacle у ворога при зіткненні з головним героєм той отримає шкоду і відправиться на точку респавну.

Серед розроблених перешкод тепер є об’єкти поєднані з об’єктами з минулої версії проєкту Spikes, для того щоб снаряди не накопичувалися на карті було створено скрипт для їх знищення через пройдену відстань.

Серед інтерактивних речей було перероблено єб’єкт зілля, які створюються за допомогою об’єкту зі скриптом SpellSpawner (Додаток А), він відповідає за породження зілля в зазначених місцях на карті. Коли зілля закінчується, скрипт запускає таймер і породжує нове зілля через певний час. Коли гра починається, метод Start() негайно викликає метод SpawnPotion(), щоб породити об'єкт зілля в точці spawnPosition. Префаб зілля potionPrefab інстанціюється за допомогою Instantiate() і може бути створений на сцені. Потім зілля додається в змінну currentPotion. Умова методу Update() перевіряється в кожному кадрі. Якщо зілля не перебуває на сцені (тобто currentPotion == null), викликається корутина RespawnPotionAfterDelay(). Це дає змогу почати затримку до того, як з'явиться нове зілля. Ця корутина використовує WaitForSeconds(respawnDelay) для затримки, після чого знову викликається метод SpawnPotion() і спауниться нове зілля. Таким чином, коли попереднє зілля буде знищено або зникне, нове зілля народиться через задану кількість секунд і повторить процес.

У скриптах, пов'язаних із зіллям, основними елементами є змінні для характеристик героя (рисунок 3.2), що змінюються під дією зілля, і методи керування цими змінами. Класи героїв мають змінні, в яких зберігаються початкові значення таких характеристик, як стрибок (JumpForce) (Додаток А). Коли герой

підбирає зілля, значення цих характеристик збільшуються на величину, зазначену в самому зіллі. Наприклад, коли герой підбирає зілля, що збільшує показники стрибка – це значення тимчасово збільшується на значення, вказане в зіллі.

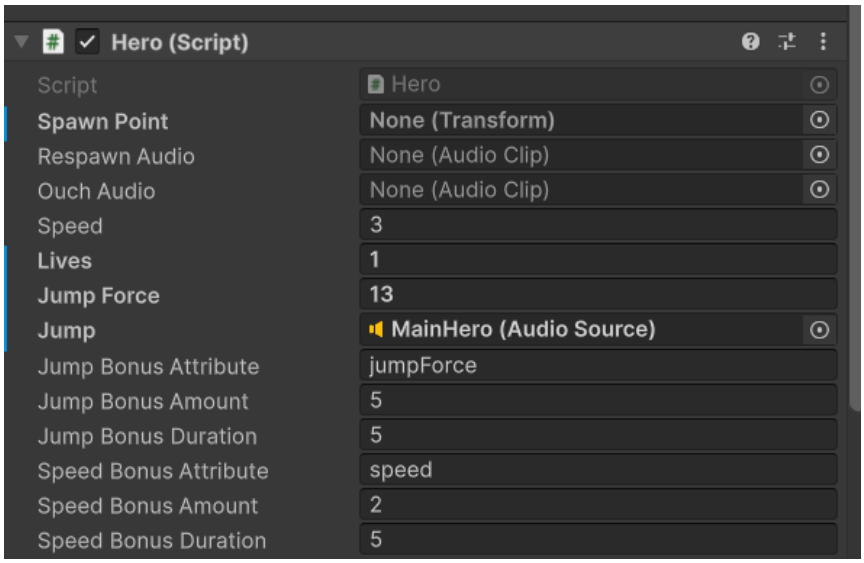

Рисунок 3.2 – Нові характеристики ігрового персонажу

Щоб забезпечити тимчасовий ефект, що змінює характеристику, у скрипті героя реалізована корутина, яка спрацьовує, коли герой отримує бонус від зілля. Тригер працює так, що після того як герой бере зілля, корутин чекає певний період часу, заданий самим зіллям, а потім відновлює характеристику до вихідного значення. Це гарантує, що зміна характеристики не є постійною і триває лише обмежений час. Крім того, у скрипті героя є метод ApplyBonus(), який приймає параметри, що вказують тип характеристики для зміни та величину зміни. Цей метод викликає корутину, яка змінює значення характеристик героя і повертає їх до вихідних значень через зазначений час. Важливо зазначити, що всі ці характеристики можна гнучко налаштовувати за допомогою скриптів зілля, дозволяючи різним типам зілля змінювати різні характеристики героя (наприклад, силу стрибка або швидкість пересування).

Було розроблено сповільнюючу зону, яка сповільнює гравця скрипт(Додаток А) працює таким чином, що в створеній зоні рух головного героя

сповільнюється, коли він опиняється всередині цієї зони. Для цього треба оголосити змінну slowDownFactor, яка визначає, наскільки знижується швидкість головного героя. Наприклад, значення 0,5 означає, що швидкість головного героя зменшується вдвічі.

Метод OnTriggerEnter2D спрацьовує, коли об'єкт із колайдером входить у зону. Цей метод перевіряє, чи є в об'єкта тег “Player”. Якщо є, він витягує компонент Hero з цього об'єкта; якщо компонент Hero присутній, він множить коефіцієнт slowDownFactor, щоб уповільнити героя.

Метод OnTriggerExit2D спрацьовує, коли об'єкт залишає зону. OnTriggerExit2D. Тут знову перевіряється, чи є в об'єкта тег 'Player'. Якщо так, то з об'єкта витягується компонент героя. Якщо компонент героя присутній, швидкість героя ділиться на коефіцієнт slowDownFactor, щоб відновити вихідну швидкість. Таким чином, якщо гравець входить у зону, його швидкість пересування зменшується і повертається до нормальної швидкості, коли він залишає зону. Цей метод ефективно контролює переміщення гравця в певних зонах гри.Було створено об’єкт Portals (Додаток А), алгоритм роботи якого базується на події, яка виникає, коли об'єкт із колайдером входить у тригер порталу. Коли це відбувається, Unity викликає метод OnTriggerEnter2D. У цьому методі виконується перевірка тега об'єкта, що перетнув тригер. Якщо тег збігається зі значенням "Player", це означає, що саме гравець увійшов у портал. У такому випадку відбувається зміна позиції об'єкта. Для цього використовується властивість transform.position, яка оновлюється новими координатами з параметра teleportPosition. Цей параметр визначає, куди саме гравець буде переміщений після входу в портал. Якщо тег об'єкта не відповідає "Player", жодні дії не виконуються, і метод завершує роботу. Таким чином, алгоритм забезпечує, що тільки гравець з певним тегом буде телепортований, а всі інші об'єкти.

Також було реалізовано потаємні кімнати (Додаток А), які відображаються поки гравець їх не знайде, алгоритм його роботи працює так, що спочатку оголошуються приватні змінні objectRenderer та objectCollider, які відповідають за

рендеринг і колізії. У методі Start() ці змінні отримують компоненти об'єкта, до якого прикріплений скрипт.

Коли інший об'єкт входить у тригерну зону, метод OnTriggerEnter2D перевіряє, чи має цей об'єкт тег "Player". Якщо так, викликається метод HideObject(), який вимикає рендеринг та колізії об'єкта. Аналогічно, коли об'єкт виходить із тригерної зони, метод OnTriggerExit2D перевіряє тег "Player" і, якщо умова виконується, викликає метод ShowObject(), який знову вмикає рендеринг та колізії.

Методи HideObject() та ShowObject() використовуються для відповідного вимкнення та увімкнення відображення та колізій об'єкта. Це корисно для створення таких елементів, як секретні двері або скарби, які з'являються та зникають у відповідь на положення гравця.

Серед інших змін – було значно перероблено скрипт головного героя (Додаток А). Перша версія класу Hero – це базова реалізація, що включає базові механіки, такі як рух, стрибки, анімацію, перевірку місцевості, обробку пошкоджень і респавн. Вона використовує шаблон синглетону через статичні властивості екземпляра. Гравець має фіксовані параметри швидкості, сили стрибка та кількості життів, а звуки стрибків реалізовано за допомогою AudioSource. Код забезпечує початкову точку перезародження, яка зберігається у currentSpawnPoint і оновлюється за потреби. Друга версія цього класу була значно розширена. Додано механізм бонусів для тимчасового збільшення певних характеристик, таких як швидкість і сила стрибка. Для цього було додано змінну для зберігання параметрів бонусу, включаючи тривалість і значення; реалізовано загальний метод TemporaryBoostCoroutine, який дозволяє застосовувати бонуси і автоматично скасовувати їх після закінчення терміну дії. У цій версії до змінних respawnAudio та touchAudio додано звуковий супровід різних подій, таких як відродження та отримання ушкоджень. Логіка стрибків, переміщення та анімації залишилася в основному тією ж самою, але код анімації став компактнішим. У цій версії немає звуків стрибків. Тому основні відмінності полягають у додаткових можливостях

другої версії: системі бонусів і додаткових звукових ефектах. Перша версія зосереджена на основних механіках, містить простішу реалізацію і є більш мінімалістичною.

Для додавання елементів головоломки до гри були додані стіни, які за допомогою скрипту WallDestroyer можна знищити, сам скрипт працює таким чином, що коли об'єкт із тегом “Box” (наприклад, коробка) потрапляє в зону спрацьовування іншого об'єкта, до якого прикріплений цей скрипт. У момент зіткнення викликається метод OnTriggerEnter2D, який перевіряє, чи має об'єкт, що стикається, тег Box. Якщо ця умова виконується, викликається метод DestroyObjectsWithTag, що передає як параметр тег, зазначений у змінній targetTag (за замовчуванням тег “Wall”) Метод DestroyObjectsWithTag, Функція GameObject.FindGameObjectsWithTag (tag) для пошуку всіх об'єктів із цим тегом. Ця функція повертає масив усіх об'єктів у сцені, що мають цей тег. Потім, використовуючи цикл foreach, скрипт переглядає кожен із цих об'єктів і викликає Destroy(obj) для знищення кожного з них. Таким чином, коли об'єкт із тегом 'Box' потрапляє в зону спрацьовування, всі об'єкти з тегом, зазначеним у targetTag (“Wall” за замовчуванням), знищуються.

Також був перероблений скрипт платформи, яка рухається по осі X, швидкість платформи задається змінною speed, а відстань, яку платформа має пройти до зміни напрямку, – змінною distance. Спочатку початкове положення платформи зберігається у змінній startPosition, але надалі ця змінна не буде використовуватися безпосередньо. Змінна movingRight визначає, куди рухається платформа – вправо або вліво. У методі Update() кожного кадру напрямок руху визначається залежно від того, вправо чи вліво рухається платформа, і відповідний рух застосовується за допомогою transform.Translate(). Швидкість руху множиться на напрямок і Time.deltaTime, щоб забезпечити плавний рух незалежно від частоти кадрів. Потім оновлюється змінна travelledDistance і записується пройдена відстань. Якщо ця відстань перевищує значення distance, напрямок руху змінюється, а пройдена відстань скидається. Платформа рухається в новому

напрямку, поки ця відстань знову не буде досягнута, і цей процес повторюється нескінченно.

Наступним об’єктом, що був створений, були маленькі платформи (рисунок 3.3), які коли гравець наступає на них падають в низ вбиваючи його. Скрипт працює таким чином (Додаток А), що платформа спочатку перебуває у кінетичному режимі завдяки елементу Rigedbody2D (не рухається), і коли гравець наступає на нього, він починає падати через фізику.

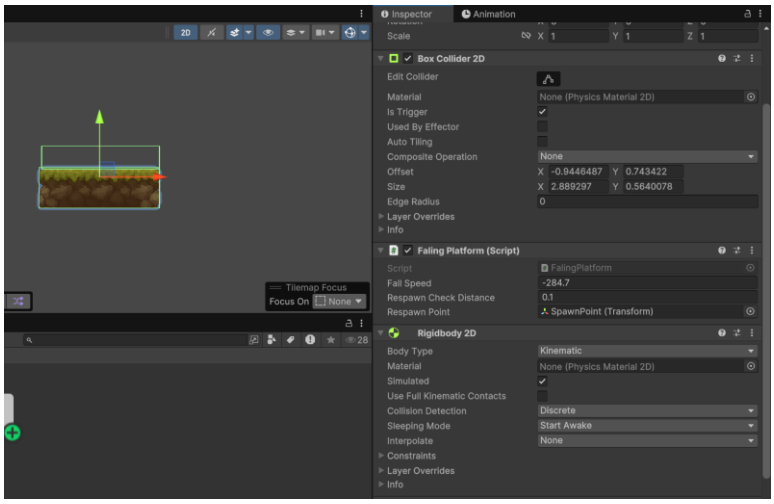

Рисунок 3.3 – Робота об’єкту fallingPlatform

Спочатку в інспекторі Unity визначається змінна, яка дає змогу задати платформу, fallSpeed визначає швидкість падіння платформи (але не використовується для керування рухом у самому коді). respawnCheckDistance задає мінімальну відстань між гравцем і точкою респауна, коли платформа повертається у вихідне положення. respawnPoint – це об'єкт у сцені, що визначає точку респауну, в яку має повернутися платформа. initialPosition зберігає початкове положення платформи, в яке вона має повернутися після падіння, a player зберігає посилання на об'єкт гравця. rb2D – компонент Rigidbody2D, який можна пов'язати з фізикою, наприклад, щоб змінити рух платформи під час падіння. У методі Start() платформа зберігає своє початкове положення та ініціалізує компонент Rigidbody2D. Спочатку платформа перебуває в кінематичному режимі

(RigidbodyType2D.Kinematic) і не рухатиметься за допомогою фізики, доки її не буде активовано (це дає змогу керувати рухом вручну) OnTriggerEnter2D() метод перевіряє, чи не зіткнеться платформа з об'єктом із тегом “Player”. Якщо так, то він зберігає посилання на об'єкт гравця і змінює bodyType платформи на RigidbodyType2D.Dynamic У методі Update() відстань між позицією гравця та точкою респауну за допомогою Vector3.Distance() щоб перевірити, чи наближається гравець до точки респауну. Якщо відстань менша або дорівнює значенню respawnCheckDistance, викликається ResetPlatform() і платформу переводять у кінематичний режим, щоб вона повернулася у вихідне положення і не зруйнувалася знову. Крім того, залишковий рух платформи припиняється шляхом скидання лінійної швидкості за допомогою rb2D.linearVelocity = Vector2.zero.

За схожою механікою був розроблений об’єкт падаючої платформи (рисунок 3.4), яка падає поступово а її скрипт працює так, що коли гравець ступає на неї вона потихеньку падає а гравець повинен зійти з неї щоб не загинути, і повертається на свою початкову позицію коли гравець досягає точки респавну (Додаток А).

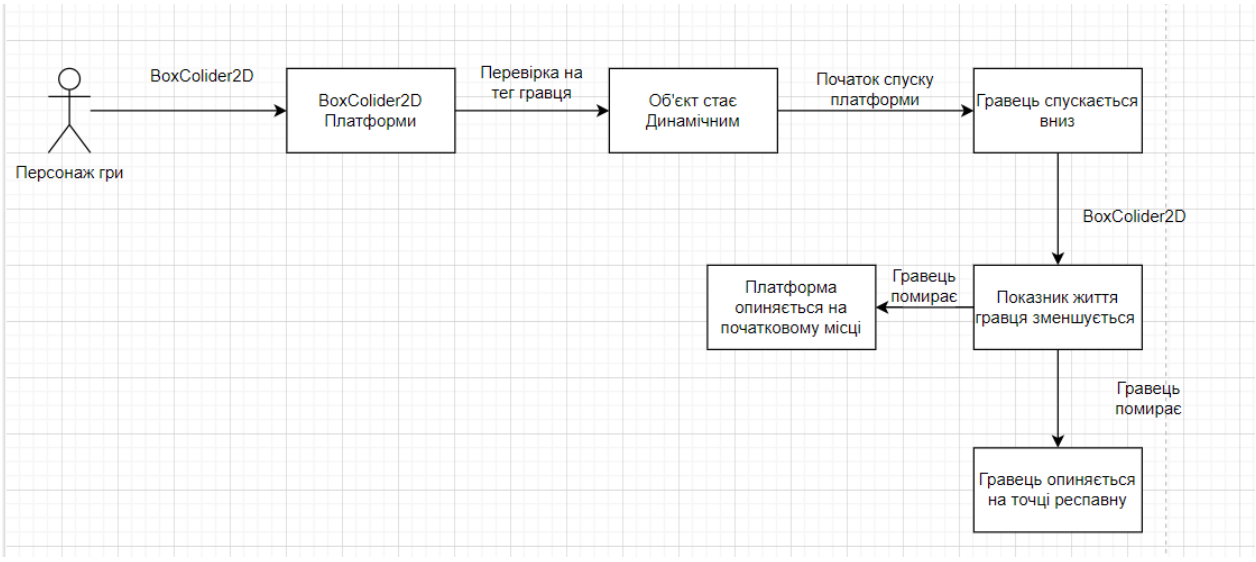

Рисунок 3.4 – Блок-схема роботи падаючої платформи

DescentSpeed визначає швидкість спуску платформи; respawnCheckDistance задає точність перевірки відстані між гравцем і точкою респауну; respawnPoint – це точка, у яку платформа має повернутися в точці респауну гравця. initialPosition – об'єкт, що вказує початкову позицію платформи, в яку гравець може повернутися після досягнення точки респауна. player – посилання на об'єкт гравця, який активує платформу. rb2D – компонент Rigidbody2D, що відповідає за фізичні властивості платформи. isDescending - булева змінна, що визначає, чи опускається платформа. Це булева змінна. Метод start() ініціалізує значення initialPosition і rb2D. Початкове положення платформи зберігається в initialPosition, а компонент Rigidbody2D витягується за допомогою GetComponent<Rigidbody2D>(). Платформа переводиться в кінематичний режим за допомогою rb2D.bodyType = RigidbodyType2D.Kinematic, тому на неї не впливає фізика та її рухом можна керувати вручну Метод OnTriggerEnter2D() використовується для запуску тригера, коли інший об'єкт (у цьому разі гравець) входить у колайдер платформи, він викликається. Якщо об'єкт, що входить у колайдер, має тег 'Player', він зберігає посилання на об'єкт гравця у змінній player і встановлює isDescending = true. Крім того, платформа вже перебуває в кінематичному режимі, але скрипт міняє його назад на RigidbodyType2D.Kinematic, щоб можна було вручну керувати її рухом без впливу фізики. Метод Update() перевіряє, чи потрібно платформі опуститися. Перевіряє, що якщо значення isDescending дорівнює true, то платформа почне спуск. Це відбувається під час зміни положення за віссю Y, а за рух платформи вниз із заданою швидкістю відповідає параметр descentSpeed * Time.deltaTime, що враховує час між кадрами (щоб забезпечити коректний рух незалежно від частоти кадрів). Метод Update() також перевіряє, чи перебуває гравець у зоні респауна. Це робиться шляхом порівняння відстані між гравцем і точкою респауна за допомогою Vector3.Distance(). Коли гравець досягає точки респауну, викликається метод ResetPlatform(), щоб повернути платформу в початкове положення і зупинити спуск. Метод ResetPlatform() повертає платформу в початкове положення, скидає значення isDescending в false Reset і зупиняє спуск.

Ще одним об’єктом взаємодії героя з оточуючим середовищем було створення пастки з таймером (рисунок 3.5), її скрипт (Додаток А) працює так, що спочатку оголошується декілька зміних та параметрів: : triggerTime - затримка перед нанесенням шкоди, damage - кількість шкоди від пастки, timer - час, який гравець проводить у пастці, playerOnTrap - визначає, чи перебуває гравець у пастці, hero зберігає посилання на об'єкт; метод Update кожного кадру перевіряє, чи перебуває гравець на пастці.

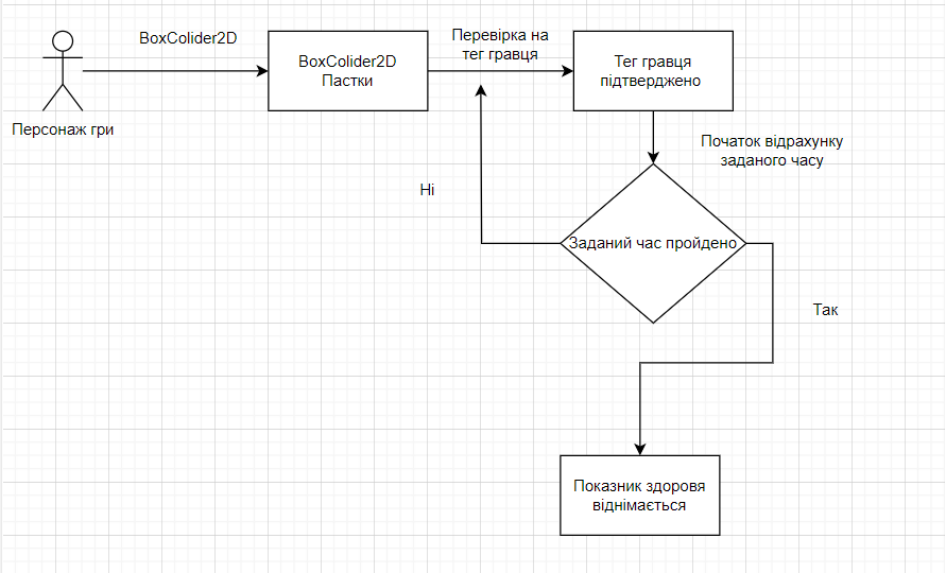

Рисунок 3.5 – Блок-схема роботи пастки

Якщо так, то таймер збільшується на час, що минув з останнього кадру. Якщо таймер перевищив triggerTime, викликається метод ApplyDamage, гравцеві завдається шкода, і таймер скидається. Якщо гравець не потрапив у пастку, таймер також скидається.Метод OnTriggerEnter2D спрацьовує, коли об'єкт із колайдером потрапляє в пастку. Якщо цей об'єкт має тег 'Player', значення playerOnTrap дорівнюватиме true, а змінна hero отримає посилання на компонент Hero об'єкта player. Це дозволяє пастці безпосередньо взаємодіяти з гравцем; метод OnTriggerExit2D викликається, коли об'єкт залишає пастку. Якщо об'єкт має тег 'Player', значення playerOnTrap дорівнюватиме false і змінна hero буде очищена.

метод ApplyDamage перевіряє, чи не дорівнює значення hero null. якщо воно не дорівнює null, герой Викликається метод GetDamage, і життя героя зменшується.

Для реалізації стріляючих ворогів був створений скрипт EnemyShooter, він створює ворога, який стріляє снарядами. Спочатку оголошуються змінні для префабу снаряду, точки вистрілу і частоти вистрілів, а також змінна для кулдауну. У методі Start компонент Animator ворога ініціалізується, і кулдаун встановлюється на нуль. У методі Update кулдаун зменшується на кожному кадрі за допомогою Time.deltaTime. Якщо кулдаун менше або дорівнює нулю, викликається метод Shoot, і кулдаун знову встановлюється відповідно до частоти вистрілів. У методі Shoot анімація ворога змінюється на анімацію вистрілу за допомогою тригера "Shoot", а в точці вистрілу створюється новий снаряд. Цей процес повторюється на кожному кадрі, забезпечуючи регулярні вистріли ворога.

3.3. Використання сервісів штучного інтелекту для оптимізації скриптів

Роль сервісу штучного інтелекту(ШІ) як інтелектуального асистента в цьому процесі полягала в допомозі в кодуванні різних механік і їх оптимізації. На всіх етапах розробки було запропоновано конкретні рішення щодо ігрової механіки, а також розв'язати більш загальні проблеми, пов'язані з дизайном, оптимізацією та поліпшенням геймплею. В роботі був використаний такий сервіс штучного інтелекту як GitHub Copilot.

Участь сервісу GitHub Copilot у проєкті почалася одразу після того, як було визначено основну концепцію та ігрову механіку гри. Одним із перших завдань сервісу було створення базової механіки для динамічної взаємодії гравця з об'єктами на сцені.

Головним завданням було розробити механізм рухомих платформ який можна було б використовувати на різних рівнях. Основним завданням було, щоб платформи рухалися за віссю X і змінювали напрямок при досягненні певної відстані. Це забезпечило б динамічність гри, що важливо для інтерактивного

геймплею. Завдання полягало в тому, щоб створити платформу, яка рухалася б в одному напрямку до певної відстані та змінювала напрямок у протилежний бік після певної відстані. Чат бот реалізував цей механізм, використовуючи принцип перевірки дальності поїздки, так що платформа змінює напрямок, коли досягає певної відстані. У коді він використовував стандартну структуру перевірки, яка змінює напрямок руху залежно від того, наскільки далеко платформа просунулася в напрямку осі X.

Наступним кроком було додавання механізму, який міг би взаємодіяти з об'єктами, що змінюють характеристики гравця. Ігровий персонаж повинен булв мати можливість взаємодіяти з певними предметами, такими як зілля, щоб тимчасово збільшити свої характеристики, наприклад, силу стрибка або швидкість. Кожне зілля справляло певний ефект на персонажа, і коли гравець брав його в руки, одна з характеристик персонажа змінювалася на деякий час. Чат бот сервісу допоміг реалізувати систему зілля і ефектів, які вони чинять на героя. Коли герой підбирає певне зілля, його характеристика, наприклад сила стрибка, збільшується на певну величину. Завдання полягало не тільки в тому, щоб забезпечити механізм збільшення характеристик, а й додати таймер, який відновлював би їхні вихідні значення через певний час. Це вимагало суворого контролю часу і логіки зміни значень. Код враховував час, що минув з моменту вибору зілля, і після закінчення цього часу знижував значення характеристики до початкового рівня. Це робило гру цікавішою і динамічнішою, оскільки зілля тимчасово збільшувало характеристику, і гравець міг використовувати цей бонус для досягнення своїх цілей, але тільки протягом обмеженого часу.

Ще одним важливим механізмом, у створенні якого було використано бота штучного інтелекту(ШІ) – це знищення предметів за допомогою інших предметів, наприклад ящиків. Завданням було розробити механізм, за якого, коли гравець перетинає колайдер з об'єктом, що має певний тег, об'єкт із цим тегом може бути знищений. У цьому разі роль героя полягала в тому, щоб нести ящик, який, перетинаючи певну зону, міг викликати знищення інших об'єктів із заданим тегом.

Для цього штучним інтелектом було створено скрипт, що використовує функцію, яка визначає наступне: коли герой перетинає певний об'єкт і колайдер, то знищується об'єкт із відповідним тегом. Цей механізм дозволив гравцеві по-різному взаємодіяти з ігровим оточенням. За допомогою таких об'єктів гравець може руйнувати перешкоди, щоб розчистити шлях, або навіть вражати ворогів, якщо у них є відповідні мітки, що було б достатньо складно реалізувати без допомоги сервісу штучного інтелекту (ШІ). Крім розробки конкретних механізмів проєкту також сприяв оптимізації ігрового процесу, надаючи поради щодо поліпшення продуктивності гри. Це охоплює виправлення можливих проблем із пам'яттю, поліпшення часу завантаження сцени та створення більш плавного й ефективного ігрового процесу. У межах цієї оптимізації з використанням сервісу штучного інтелекту (ШІ) було поліпшено управління ресурсами, щоб гра працювала стабільно навіть за великої кількості об'єктів на сцені.

3.4. Тестування ігрового застосунку

Тестування в ігровій розробці має свої особливості та нюанси. Процес тестування гри дозволяє не тільки виявити баги й помилки в механіках, а й допомогти в оптимізації продуктивності, забезпечити зручність і комфорт для гравців, а також гарантувати стабільність та коректність роботи гри на різних платформах.

Існує кілька етапів і методів тестування гри, кожен з яких має свої особливості.

По-перше, потрібно було вирішити, які механізми необхідно протестувати. У випадку створеного додатку це були скрипти, пов'язані з рухомими платформами, зіллям, що впливають на характеристики гравця, і взаємодією між об'єктами на сцені (наприклад, знищення об'єктів за допомогою інших).

Одним із перших етапів тестування була перевірка наявності рухомих платформ. Зазвичай механіка рухомих платформ потребує точної перевірки,

оскільки вони повинні змінювати напрямок тільки після досягнення певної відстані. Тестування цієї механіки полягало в перевірці правильності визначення відстані, пройденої платформою, і зміни напрямку руху після досягнення цієї відстані. Також важливо було переконатися, що платформа рухається плавно і без тряски. Початковий варіант (попередня версія розробленої гри) платформи, яка рухається по осі X включав в себе рух по 2 точках (A та B) які задавалися, але він не відповідав потрібним стандартам і в результаті скрипт був перероблений так, щоб платформа рухалася по відповідній дистанції, після цього виникла нова проблема в ключі того, що платформа після віддалення на потрібну дистанцію вона продовжувала свій шлях до нескінченності, для вирішення цієї проблеми було перероблено скрипт так що коли платформа повертається на місце старту вона по новій починає проходити відведену дистанцію.

Під час тестування ворогів виникла проблема з їх спрайтами, коли ворог мав рухатися в іншу сторону під час того як він повертався, його модель бігла задом на перед, цю помилку було виправлено перестановкою елементу SpriteRenderer і переписування компоненту Flip() в скрипті.

Далі було протестовано функціонал зілля, які впливають на характеристики гравця, працюють правильно. Було створено кілька різних зілля, кожне з яких тимчасово збільшувало певну характеристику, наприклад силу стрибка або швидкість. Обравши зілля, треба було переконатися, що характеристика персонажа справді змінилася і повернулася до початкового значення через певний час. Ще одним важливим контрольним пунктом було переконатися, що не виникнуть помилки, якщо гравець підбере кілька зілля одного типу або якщо час їхньої дії накладеться одне на одне. Під час тестування основними помилками в роботі зілля було те, що вони збільшували характеристики персонажу але не повертали їх до початкових, ще одною проблемою було те, що зілля не з’являлися по новому після їх використання. Першу проблему було вирішено тим, що характеристики які будуть збільшуватися було записано на головного героя і їх відкат також задавався в ньому а в скрипті зілля була лише реакція на взаємодію з головним героєм в

вигляді знищення об’єкту, проблему з появою інтерактивних речей було вирішено переробивши скрипт SpellSpawner.

Наступним аспектом тестування було взаємодія між об'єктами, що мали різні теги. Наприклад, якщо у персонажа є ящик і він перетинає колайдер із певним тегом, інші об'єкти з тим самим тегом мають бути знищені. Важливо було перевірити, чи правильно працює цей механізм, чи всі об'єкти знищуються, коли головний герой перетинає колайдер, і чи немає несподіваних помилок, наприклад, дерев'яної шухляди, яка некоректно взаємодіє з колайдером.

Однією із проблем під час такого тестування була некоректна робота об’єкту CheckPoint (рисунок 3.9), в той час коли головний герой вмирав він повертався не на задану точку а його спавнило на -30 по координаті Z в об’єкті Tile, для виправлення цієї помилки було створено пустий об’єкт і закинуто в нього спрайт обє’кту для зберігання позиції а до неї ще один пустий об’єкт з Box Collider 2D і помилка була виправлена.

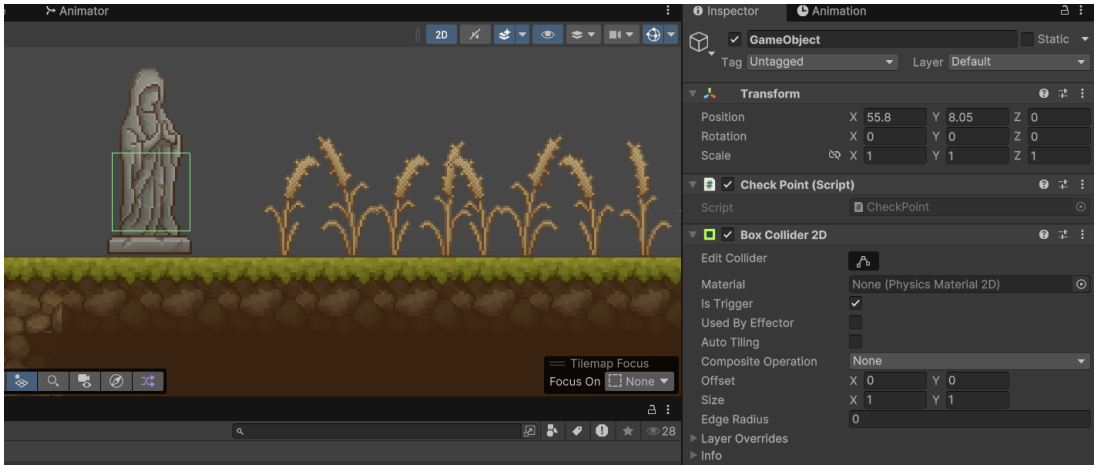

Рисунок 3.6 – Перероблений колайдер для об’єкту CheckPoint

Звук і музика також є важливою частиною ігрового процесу, і їх не можна ігнорувати. Це включає в себе перевірку правильності відтворення звукових ефектів, їхньої відповідності подіям у грі та коректної роботи системи регулювання гучності. Необхідно перевірити, щоб напрямок звуку правильно відповідав

положенню об'єктів у грі. Також слід перевірити, як звукові елементи поводяться за високих навантажень, наприклад, коли на сцені одночасно відбувається безліч подій [22].

3.5. Пропозиції щодо вдосконалення та оптимізації проєкту

Поліпшення гри – це складний процес, який залежить від безлічі факторів, включно з цільовою аудиторією, платформою, механікою, графікою, сюжетом і монетизацією. Поліпшення охоплюють як технічні аспекти, так і поліпшення загального користувацького досвіду (UX).

Покращення починається із врахування відгуків користувачів. Збір зворотного зв’язку – один із найважливіших етапів розвитку ігрового проєкту. Потрібно уважно вивчити, що тестова аудиторія скаже про гру: що їм подобається, що викликає труднощі або розчарування, і що вони хотіли б бачити в майбутньому.

На основі цих даних можна скласти план покращення. Наприклад, якщо гравці скаржаться на одноманітність рівнів, можна додати більше різноманітних механік або нових зон.

Якщо в грі є проблеми з оптимізацією, то їх потрібно вирішити, забезпечивши плавний ігровий процес навіть на пристроях із нижчими характеристиками.

Одна зі сфер для поліпшення – вдосконалення графіки. Якщо візуальні ефекти у грі виглядають застарілими або позбавлені деталізації, вам варто подумати про оновлення графічних елементів.

Наприклад, в майбутньому можна буде поліпшити текстури, додати більш деталізовані моделі або реалізувати сучасні методи освітлення, як-от динамічне освітлення або тіні в реальному часі. Крім того, сучасні ігрові рушії дають змогу реалізувати ефекти пост оброблення, як-от згладжування, розмиття, глибина різкості та інші візуальні поліпшення. Графічні поліпшення можуть значно посилити занурення гравця в гру.

Ще один важливий аспект – музика та звукові ефекти. Аудіовізуальний супровід створює атмосферу гри. Якщо музика і звуки в грі повторюються або не відповідають тому, що відбувається, якість ігрового процесу може бути зіпсована. Подумайте про те, щоб додати нові треки відповідно до ігрової ситуації, наприклад, спокійну музику для сцен дослідження і напружену музику для сцен битв. Крім того, звукові ефекти мають відповідати поведінці гравця і подіям у грі.

В подальших дослідженнях, доцільним було б додати аудіо супровід кроків гравця, звук яких змінювався залежно від дорожнього покриття, а звуки ударів і вибухів мають бути реалістичними та захопливими. Покращена ігрова механіка також може зробити гру більш цікавою та захопливою.

Для того щоб простий геймплей зробити більш динамічним і складним, доцільно додати нові механіки. Це можуть бути нові типи ворогів, унікальні здібності персонажу, інтерактивні об'єкти тощо. Наприклад, якщо гра містить елементи платформера, можна додати такі механіки, як гойдалки, телепорти, пастки та бонусні рівні.

Сюжет і персонажі також є важливими елементами, які можна розвивати. Гравці часто люблять відчувати себе частиною історії, тому інтеграція сюжету може значно покращити їхній досвід. Крім того, можна додати нових персонажів, які матимуть унікальні здібності чи особливості, або розширити можливості кастомізації для гравця, щоб зробити персонажа більш індивідуальним.

Серед майбутніх пріоритетів розробки гри важливими аспектами є оптимізація продуктивності, виправлення помилок і підвищення стабільності гри. Потрібно регулярно аналізувати роботу гри на різних пристроях і усувати проблеми, що виникають. Також розглядаються можливість підтримки модів, щоб гравці могли створювати власний контент для гри. Це приверне більше користувачів і подовжить життя проєкту.

В якості самих перших покращень для проєкту, які треба буде реалізувати самим першим, – це перероблення інтерфейсу гри: головного меню, меню вибору

рівнів, меню паузи, та меню проходження рівня. Також потрібно буде переробити скрипт головного героя і його анімації (рисунок 3.8).

Треба переробити скрипт так, щоб він успадковував від класу KinematicObject, який надає базову функціональність для оброблення фізики, руху та взаємодії з навколишнім середовищем. Це означає, що більша частина коду для роботи з властивостями фізики може бути вже реалізована в базовому класі, що скорочує дублювання коду і спрощує обслуговування.

Важливою частиною яку треба змінити – це підтримка звуку. Вбудовані посилання на аудіокліпи стрибків, пошкоджень і респаунів (наприклад, jumpAudio, respawnAudio, touchAudio) можуть додати емоційну глибину і зворотний зв'язок із гравцем, роблячи ігровий процес більш захопливим.

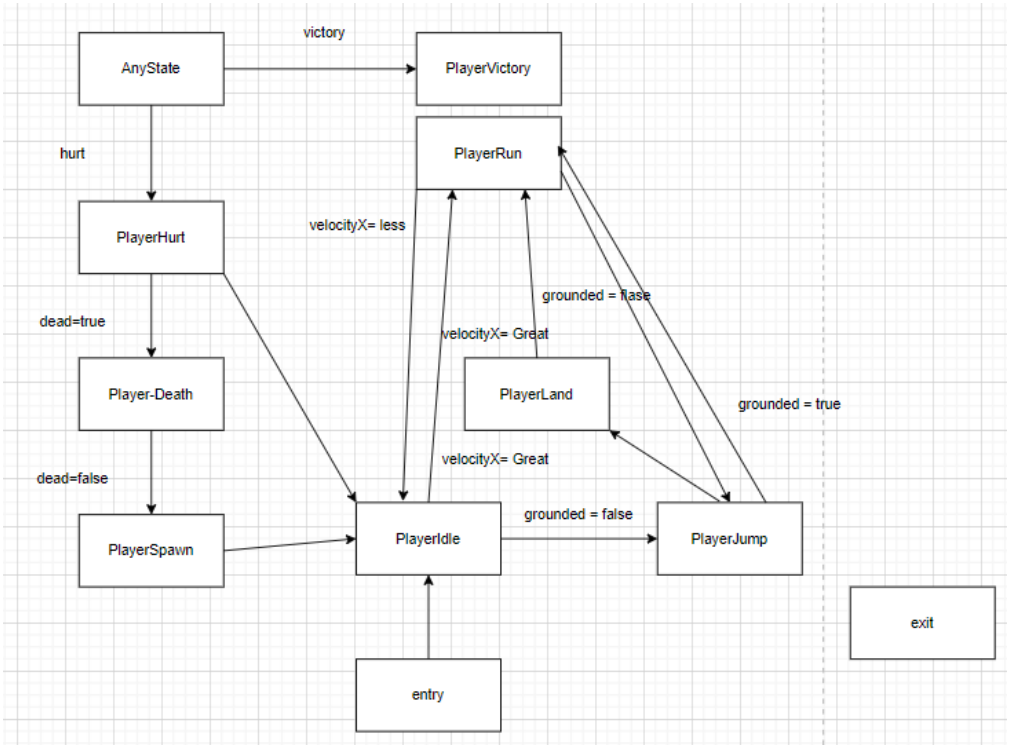

Рисунок 3.7 – Блок-схема альтернативної роботи анімацій гравця

Детальна обробка станів стрибка (JumpState) дасть змогу краще контролювати поведінку персонажа. Переходи між такими станами, як “на землі”, “готується до стрибка”, “у польоті” і “приземлення”, додають природності в

анімацію і фізику стрибка. Це дає змогу уникнути різких або неточних переходів між діями персонажа. Інтеграція з анімацією – важлива перевага.

Переробка скрипта для взаємодії з аніматором для зміни станів анімації, таких як приземлення та горизонтальна швидкість (velocityX). Це дасть плавний і контекстуально релевантний візуальний зворотний зв'язок із гравцем. Скрипт може змінювати орієнтацію персонажа за допомогою властивості spriteRenderer.flipX. Це реалізовано ефективно і без зайвих перевірок.

Механізм гальмування стрибка додає новий рівень контролю. Гравці можуть скоротити тривалість і висоту стрибка, відпустивши кнопку стрибка раніше. Це додає динаміки в ігровий процес і дає змогу гравцеві краще контролювати персонажа.

Наявність PlatformerModel, що містить такі змінні, як jumpModifier і jumpDeceleration, дає змогу легко налаштовувати поведінку персонажа в різних сценаріях. Поведінка персонажа в різних сценаріях може бути легко налаштована. Це також робить сценарій гнучкішим і масштабованішим: робота з елементами керування за допомогою Input.GetAxis і Input.GetButton забезпечує плавність ігрового процесу і підтримує стандартні інтерфейси Unity для контролерів і клавіатур. Скрипт також підтримує інтеграцію з мобільними пристроями та іншими платформами.Система подій, що використовує Schedule, дає змогу модульно обробляти такі дії, як стрибок, завершення і приземлення. Це дає змогу легко додавати нові механіки, як-от частинки, звуки та інші візуальні ефекти під час цих подій.

Зміна структури коду для легкого впроваджування нових функцій та механізмів. Наприклад, легко додати двоступеневі стрибки, спеціальні здібності та взаємодію з навколишнім середовищем. Це відбувається тому, що обробка кожного стану або механізму добре розділена і зрозуміла.

3.6. Висновок до третього розділу

У третьому розділі кваліфікаційної роботи було описано процес розробки додатку, процес тестування, взаємодії з сервісами штучного інтелекту та майбутні покращення для додатку. Ключовим елементом розробки програми було правильне планування архітектури гри. Результатом була робота над реалізацією різних механік, як-от рух персонажа, взаємодія з об'єктами, збирання предметів і створення умовної фізики для внутрішньої логіки ігрового світу. Основою для цього стало використання сучасних інструментів програмування, таких як Unity 6 і мови програмування C# для розробки скриптів. Ці інструменти дають змогу швидко створювати прототипи та реалізовувати складні сценарії взаємодії між об'єктами.

Було реалізовано багатофункціональні скрипти, що враховують різні аспекти взаємодії, від базових маніпуляцій персонажами до просунутих механік, як-от переміщення платформою і використання бонусів. Це вимагало розуміння принципів роботи ігрового рушія, тонкого налаштування фізичних параметрів і оптимізації кодової бази для забезпечення стабільної роботи гри на різних платформах.

Наступним важливим етапом стало тестування. Різні сценарії гри перевірялися і тестувалися на поведінку персонажів і взаємодію з об'єктами в різних умовах. Одним із завдань було виявлення прихованих помилок, які не завжди були очевидні на ранніх етапах розробки. Помилки обробки зіткнень або помилки в логікі руху могли призвести до непередбачуваної поведінки платформи і персонажа. Було виправлено ці помилки, запровадивши додаткові перевірки та оптимізувавши обробку даних. Крім того, особливу увагу було приділено тестуванню зілль і бонусів, оскільки вони безпосередньо впливають на характеристики персонажа. Були розглянуті різні варіанти поведінки гравця, наприклад, збір кількох бонусів у швидкій послідовності або опора на присутність об'єктів для їхнього знищення. Окремої згадки заслуговує використання послуг

штучного інтелекту під час створення гри. Інтелектуальні підходи були застосовані для оптимізації процесу розробки.

Система штучного інтелекту допомогла швидко виявити помилки в коді, пояснити складні фрагменти сценарію і навіть запропонувати можливі поліпшення наявних механік. Це дало змогу не тільки заощадити час, а й освоїти нові підходи до вирішення проблем. Впровадження цих методів у процес розробки відкрило нові горизонти, особливо в плані генерації ідей і створення прототипів нових функцій.

Розглядаючи можливості для поліпшення, було оцінено різні аспекти гри. Одним із ключових напрямків було поліпшення інтерактивності ігрового світу.

Під час майбутніх досліджень, можна було б додати складніші механізми поведінки ворогів або нові типи перешкод і об'єктів, які вимагають від гравця стратегічного підходу до їх подолання. Функціональність бонусів також можна розширити, збільшивши тривалість їхньої дії або зробивши їх залежними від обраної гравцем тактики. Наприклад, зілля можуть не тільки збільшувати швидкість і силу стрибків, а й відкривати нові маршрути і способи проходження рівня.

Ще одним важливим аспектом поліпшень стала робота над графікою і дизайном. У сучасних іграх візуальні елементи відіграють вирішальну роль у залученні гравця. В майбутніх дослідженнях можна впровадити більш деталізовані текстури, динамічне освітлення та анімації, які додають глибину ігровому світу. Таким чином, можна створити унікальний досвід, оптимізований під навички та очікування кожного користувача.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було комплексно вивченню та впроваджено процеси розробки ігрових додатків у середовищі Unity 6 з використанням новітніх технологій. Основний акцент був зроблений на створенні інноваційних ігрових механік, оптимізації алгоритмів та розширенні функціоналу ігрового додатку за рахунок інтеграції сервісів штучного інтелекту. В ході роботи було проаналізовано сучасні тенденції в розробці ігрових додатків. Особливу увагу було приділено впливу штучного інтелекту на ігрову індустрію. Аналіз підтвердив, що штучний інтелект є одним з ключових факторів, що впливають на розвиток ігрових технологій, зачіпаючи всі аспекти ігрових додатків, від оптимізації сценаріїв до процедурної генерації контенту.

Використання сервісів штучного інтелекту значно скоротило час розробки і підвищило її якість, продемонстровано, що вони надають нові можливості для взаємодії з ігровим середовищем. Одним із головних результатів цього дослідження стало створення нової покращеної версії 2D-платформера з унікальним дизайном рівнів та нелінійною ігровою механікою. В основі цієї концепції лежить ідея створення інтерактивного ігрового середовища, яке занурює гравця в динамічний і багатогранний ігровий світ. Для досягнення цієї мети були розроблені інноваційні підходи до дизайну рівнів, включаючи інтеграцію нових механізмів та оптимізацію існуючих алгоритмів. Нелінійна природа гри пропонує різноманітний геймплей та підвищує рівень залучення користувачів.

Для автоматизації процесу програмування було використано GitHub Copilot, що спростило створення коду та дозволило команді зосередитися на творчих аспектах розробки. Крім того, сервіс GitHub Desktop використовувався для контролю версій, забезпечуючи узгодженість і зручність у роботі над проєктом.

Важливим результатом стало впровадження процедурної генерації контенту. Це рішення дозволило створити динамічне ігрове середовище, яке адаптується до поведінки гравців та пропонує різноманітний геймплей. Процедурна генерація

використовувалася для створення ландшафтів, рівнів та інтерактивних об'єктів, що робить гру більш захоплюючою та унікальною для кожного користувача. Такий підхід також скоротив час і витрати на розробку, дозволивши зосередитися на реалізації своїх творчих ідей.

Розроблений ігровий додаток містить інтерактивні об'єкти, які забезпечують високу динаміку ігрового процесу. В процесі розробки було реалізовано декілька важливих функціональних рішень, які значно покращили якість гри. Зокрема, були створені алгоритми, що гарантують взаємодію між ігровими об'єктами та гравцем, оптимізовано процес тестування ігрової механіки та впроваджено системи для підтримки подальшого масштабування проєкту. Значну увагу було приділено дизайну рівнів, який поєднує в собі естетику та функціональність.

В ході роботи було проведено тестування розробленого додатку, що включало перевірку якості коду, виявлення та виправлення помилок, оптимізацію ресурсів для забезпечення високої продуктивності гри. Також була розроблена система документації, що містить опис реалізованих функцій, рекомендації щодо подальшого розвитку проєкту та пропозиції щодо покращення ігрової механіки.

Дослідження демонструє важливість інтеграції сучасних технологій у процес розробки ігор та значення штучного інтелекту для підвищення якості ігрових продуктів. Результати дослідження підтверджують, що інноваційні підходи, такі як генерація процедур, автоматизація програмування та оптимізація алгоритмів, можуть бути використані для створення конкурентоспроможних ігрових продуктів, які відповідають сучасним вимогам гравців. Розроблений ігровий додаток є прикладом вдалого поєднання технологій та креативності і може слугувати основою для подальших досліджень та розробок в ігровій індустрії.

Його структура дозволяє інтегрувати нові функції, розширювати функціональність гри та адаптувати продукт до змін на ринку. Він також може слугувати джерелом натхнення для інших розробників, які хочуть впроваджувати інновації у свої проєкти. Таким чином, кваліфікаційний тест зробив значний внесок у дослідження та комерціалізацію розробки ігор та додатків з використанням

штучного інтелекту. Він демонструє ефективність сучасних методів розробки та інтеграції інноваційних технологій. Використання штучного інтелекту дало змогу впровадити нові алгоритми, які не тільки оптимізують наявні, а й значно покращують ігровий процес. Особливу увагу було приділено поліпшенню скриптингу, реалізації нелінійного дизайну рівнів і оптимізації продуктивності, щоб створити продукт, у якому будуть зацікавлені як кінцеві користувачі, так і розробники, що використовують запропонований підхід надалі.

Отримані результати демонструють потенціал сучасних технологій для створення якісних, інтерактивних і захопливих ігор, що відповідають вимогам сучасної аудиторії. Використання Unity 6 у поєднанні з інноваційними підходами до розробки доводить, що сучасні інструменти відкривають нові горизонти для реалізації творчих ідей в ігровій індустрії.

Таким чином, ця робота не лише сприяє розширенню меж можливого у розробці ігор, але й демонструє, як ефективне поєднання сучасних технологій і творчого підходу дозволяє створювати продукти, які відповідають актуальним викликам ігрової індустрії та встановлюють нові стандарти якості.

СПИСОК ДЖЕРЕЛ

1. Ігри та ігровий ринок взагалом URL: https://mezha.media/2024/08/15/game-market-2024-expectations/ . (Дата звернення: 17.12.24)
2. В Україні на 66% більше нових інді-ігор. Головне зі звіту Unity про ігрову індустрію URL: https://gamedev.dou.ua/news/unity-gaming-report-2023-trend-and-changes/ . (Дата звернення: 17.12.24)
3. Штучний інтелект в ігровій індустрії – нові можливості для розробників та гравців URL: https://mediacom.com.ua/shtuchnij-intelekt-v-igrovij-industrii-novi-mozhlivosti-dlya-rozrobnikiv-ta-gravtsiv/ . (Дата звернення: 22.12.24)
4. Microsoft інтегрує AI у Minecraft та інші ігри для Xbox і PC URL: https://klap.com.ua/news/microsoft-intieghruie-ai-u-minecraft-ta-inshi-ighri-dlia-xbox-i-pc. (Дата звернення: 22.12.24)
5. Як штучний інтелект використовують у розробці відеоігор, і до чого це може призвести URL: https://mezha.media/articles/yak-shi-vykorystovuiut-u-rozrobtsi-videoihor/ . (Дата звернення: 22.12.24)
6. Ігри майбутнього. Як штучний інтелект трансформує геймерський досвід 2025 року URL: https://mindscope.biz.ua/igry-majbutnogo-yak-shi-transformuie-gejmerskyj-dosvid-2025-roku/ . (Дата звернення: 22.12.24)
7. A Gamified Learning Framework to Cultivate Critical Thinking Skills in Students URL: https://ieeexplore.ieee.org/abstract/document/10747276 . (Дата звернення: 22.12.24)
8. Що таке прототипування? URL: https://dizz.in.ua/uk/chto-takoe-prototipirovanie/ . (Дата звернення: 23.12.24)
9. Модульне програмування URL : https://it-rating.ua/modulne-programuvannya-v-veb-rozrobtsi-perevagi-vikoristannya-moduliv-ta-komponentiv-dlya-pidtrimki-kodu . (Дата звернення: 23.12.24)

10. Level-дизайн URL : https://avada-media.ua/services/disain-urovney-dlya-2d-igr-osnovnye-patterny/ . (Дата звернення: 23.12.24)

11. Agile як невід'ємна частина успішних проєктів URL : https://foxminded.ua/shcho-take-agile/ . (Дата звернення: 24.12.24)

12. Unity in Action: Multiplatform Game Development in C# - Джо Гельдер (400 стор., 2021 рік).

13. Celeste URL: https://store.steampowered.com/app/504230/Celeste/ . (Дата звернення: 24.12.24)

14. Hollow Knight URL: https://store.steampowered.com/app/367520/Hollow_Knight/ . (Дата звернення: 24.12.24)

15. LIMBO URL: https://store.steampowered.com/app/48000/LIMBO/ . (Дата звернення: 25.12.24)

16. Super Meat Boy URL: https://store.steampowered.com/app/40800/Super_Meat_Boy/. (Дата звернення: 25.12.24)

17. Robot Heroes: https://store.steampowered.com/app/686110/Robot_Heroes/ . (Дата звернення: 25.12.24)

18. Рендеринг, освітлення та підтримка WebGPU. Відбувся реліз Unity 6 Preview URL: https://gamedev.dou.ua/news/unity-6-preview-release/ . (Дата звернення: 25.12.24)

19. Вступ до GIT: основні поняття та налаштування URL: https://lemon.school/blog/vstup-do-git-osnovni-ponyattya-ta-nalashtuvannya . (Дата звернення: 26.12.24)

20. Документація по GitHub Desktop URL: https://docs.github.com/en/desktop/overview/about-github-desktop. (Дата звернення: 27.12.24)

21. Остання версія Visual Studio 2022 із вбудованим помічником GitHub Copilot : https://visualstudio.microsoft.com/ru/github-copilot/ . (Дата звернення: 28.12.24)

22. "Unity From Zero to Proficiency (Beginner)" - Патрік Фелісьєн (280 стор., 2017)

ДОДАТОК А

Скрипт перешкоди:

```
public class Obstacle : MonoBehaviour
{
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject == Hero.Instance.gameObject)
        {
            Hero.Instance.GetDamage();
        }
    }
}
```

Скрипт для роботи зі звуком:

```
public class AudioManager : MonoBehaviour
{
    public Slider volumeSlider;

    private void Start()
    {
        volumeSlider.value = PlayerPrefs.GetFloat("Volume", 1.0f);
    }
    public void OnVolumeSliderChanged()
    {
        float volume = volumeSlider.value;
        AudioListener.volume = volume;
        PlayerPrefs.SetFloat("Volume", volume);
    }
```

```
}
```

Скрипт для роботи з камерою:

```
public class CameraController : MonoBehaviour
{
   [SerializeField] private Transform player;

   private Vector3 pos;

   private void Awake()
   {
      if (!player)
      {
         player = Object.FindAnyObjectByType<Hero>().transform;
      }
   }
   private void Update()
   {
      pos = player.position;
      pos.z = -10f;

      transform.position = Vector3.Lerp(transform.position, pos, Time.deltaTime);
   }
}
```

Скрипт точки зберігання:

```
public class CheckPoint : MonoBehaviour
{
   private Transform respawnPoint;
```

```
    private void Start()
    {
        respawnPoint = transform;
        respawnPoint.position = new Vector3(respawnPoint.position.x,
respawnPoint.position.y, Hero.Instance.transform.position.z);
    }
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            Hero.Instance.SetSpawnPoint(respawnPoint);
        }
    }
}
```

Оновлений скрипт ворога ближнього формату:

```
public class EnemyController : MonoBehaviour
{
    [SerializeField] private float patrolDistance = 5f;
    [SerializeField] private float speed = 2f;
    private float leftBoundary;
    private float rightBoundary;
    private bool movingRight = true;
    private SpriteRenderer spriteRenderer;
    private void Start()
    {
        leftBoundary = transform.position.x - patrolDistance;
        rightBoundary = transform.position.x + patrolDistance;
        spriteRenderer = GetComponent<SpriteRenderer>();
```

```
    }
    private void Update()
    {
        Patrol();
    }
    private void Patrol()
    {
        if (movingRight)
        {
            transform.Translate(Vector2.right * speed * Time.deltaTime);
            spriteRenderer.flipX = false;
            if (transform.position.x >= rightBoundary)
            {
                movingRight = false; // Змінюємо напрямок руху на ліво
            }
        }
        else
        {
            transform.Translate(Vector2.left * speed * Time.deltaTime);
            spriteRenderer.flipX = true;
            if (transform.position.x <= leftBoundary)
            {
                movingRight = true; // Змінюємо напрямок руху на право
            }
        }
    }
}
```

Скрипт сутності:

```
public class Entity : MonoBehaviour
{
   public virtual void GetDamage()
   {
   }
   //Deth відповідає за знищення об'єкту після кількості життів ниже 0
   public virtual void Deth()
   {
      Destroy(this.gameObject);
   }
}
public class Finale : MonoBehaviour
{
   private void OnTriggerEnter2D(Collider2D collision)
   {
      if (collision.CompareTag("Player"))
      {
         SceneManager.LoadScene(5);
      }
   }
}
```

Скрипт для зілля швидкості:

```
public class HasteSpell : MonoBehaviour
{
   private void OnTriggerEnter2D(Collider2D other)
   {
      if (other.CompareTag("Player"))
```

```
        {
            Hero hero = other.GetComponent<Hero>();
            if (hero != null)
            {
                hero.ApplySpeedBonus();
                Destroy(gameObject);
            }
        }
    }
}
```

Оновлений скрипт головного героя:

```
public class Hero : Entity
{
    private Rigidbody2D rb;
    private SpriteRenderer sprite;
    private Animator anim;
    private bool isGrounded = false;
    public Transform spawnPoint;
    public AudioClip respawnAudio;
    public AudioClip ouchAudio;
    private static Transform currentSpawnPoint;
    public static Hero Instance { get; set; }
    [SerializeField] private float speed = 4f;
    [SerializeField] public int lives = 3;
    [SerializeField] public float jumpForce = 5f;
    private Vector3 respawnPosition;
    [SerializeField] private AudioSource jump;
    // Параметри бонусів для стрибка та швидкості
```

```
    [SerializeField] private string jumpBonusAttribute = "jumpForce";
    [SerializeField] private float jumpBonusAmount = 5.0f;
    [SerializeField] private float jumpBonusDuration = 5.0f;
    [SerializeField] private string speedBonusAttribute = "speed";
    [SerializeField] private float speedBonusAmount = 2.0f;
    [SerializeField] private float speedBonusDuration = 5.0f;
    private void Start()
    {
        currentSpawnPoint = spawnPoint;
    }
    private void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
        sprite = GetComponentInChildren<SpriteRenderer>();
        anim = GetComponent<Animator>();
        Instance = this;
        respawnPosition = transform.position;
    }
    private void Run()
    {
        Vector3 dir = transform.right * Input.GetAxis("Horizontal");
        transform.position = Vector3.MoveTowards(transform.position,
transform.position + dir, speed * Time.deltaTime);
        sprite.flipX = dir.x < 0.0f;
    }
    private void Jump()
    {
        rb.AddForce(transform.up * jumpForce, ForceMode2D.Impulse);
    }
```

```
private void Update()
{
    if (Input.GetButton("Horizontal"))
    {
        Run();
    }
    if (isGrounded && Input.GetButtonDown("Jump"))
    {
        Jump();
    }
    if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.D))
    {
        anim.SetBool("Runing", true);
    }
    else
    {
        anim.SetBool("Runing", false);
    }
    if (Input.GetKey(KeyCode.Space))
    {
        anim.SetTrigger("Jump");
        jump.Play();
    }

    if (isGrounded == false)
    {
        anim.SetTrigger("Fall");
    }
}
```

```
    private void CheckGround()
    {
        Collider2D[] collider = Physics2D.OverlapCircleAll(transform.position,
0.9f);
        isGrounded = collider.Length > 1;
    }
    private void FixedUpdate()
    {
        CheckGround();
    }
    public void UpdateRespawnPosition(Vector3 newRespawnPosition)
    {
        respawnPosition = newRespawnPosition;
    }
    public void SetSpawnPoint(Transform spawn)
    {
        currentSpawnPoint = spawn;
    }
    private void Respawn()
    {
        transform.position = currentSpawnPoint.position;
    }
    public override void GetDamage()
    {
        lives -= 1;
        Debug.Log(lives);
        if (lives < 1)
            Respawn();
    }
```

```
    public void ApplyJumpBonus()
    {
        StartCoroutine(TemporaryBoostCoroutine(jumpBonusAttribute,
jumpBonusAmount, jumpBonusDuration));
    }
    public void ApplySpeedBonus()
    {
        StartCoroutine(TemporaryBoostCoroutine(speedBonusAttribute,
speedBonusAmount, speedBonusDuration));
    }
    private IEnumerator TemporaryBoostCoroutine(string attribute, float
boostAmount, float duration)
    {
        switch (attribute)
        {
            case "jumpForce":
                jumpForce += boostAmount;
                break;
            case "speed":
                speed += boostAmount;
                break;
            default:
                Debug.LogWarning("Unsupported attribute: " + attribute);
                yield break;
        }
        yield return new WaitForSeconds(duration);
        switch (attribute)
        {
            case "jumpForce":
```

```
                jumpForce -= boostAmount;
                break;
            case "speed":
                speed -= boostAmount;
                break;
        }
    }
}
```

Скрипт зілля стрибку:

```
public class JumpSpell : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            Hero hero = other.GetComponent<Hero>();
            if (hero != null)
            {
                hero.ApplyJumpBonus();
                Destroy(gameObject); // Знищення зілля
            }
        }
    }
}
```

Оновлений скрипт горизонтальної платформи:

```
public class LeftRoghtPlatform : MonoBehaviour
```

```
{
    public float speed = 3f;
    public float distance = 5f;
    private Vector3 startPosition;
    private float traveledDistance = 0f;
    private bool movingRight = true;
    private void Start()
    {
        startPosition = transform.position;
    }

    private void Update()
    {
        // Визначаємо напрямок руху
        float direction = movingRight ? 1f : -1f;
        // Рухаємо платформу по осі X в залежності від напрямку
        float move = speed * direction * Time.deltaTime;
        transform.Translate(move, 0f, 0f); // Переміщаємо платформу
        traveledDistance += Mathf.Abs(move); // Збільшуємо пройдену відстань
        // Перевіряємо, чи платформа пройшла вказану відстань
        if (traveledDistance >= distance)
        {
            movingRight = !movingRight; // Міняємо напрямок
            traveledDistance = 0f; // Скидаємо пройдену відстань
        }
    }
}
```

Скрипт для снарядів:

```
public class MoveForward : MonoBehaviour
{
    public float speed = 10.0f;
    public bool moveRight = true;
    public float distance = 10.0f;
    private float traveledDistance = 0.0f;
    // Update is called once per frame
    void Update()
    {
        float direction = moveRight ? 1.0f : -1.0f;
        Vector3 move = new Vector3(speed * direction * Time.deltaTime, 0.0f, 0.0f);
        transform.position += move;
        traveledDistance += speed * Time.deltaTime;

        if (traveledDistance >= distance)
        {
            Destroy(gameObject);
        }
    }
}
```

Оновлений скрипт для вертикальної платформи:

```
public class MovingPlatform : MonoBehaviour
{
    [SerializeField] private float speed = 2f; // Швидкість руху платформи
    [SerializeField] private float upperLimit = 5f; // Верхня межа
    [SerializeField] private float lowerLimit = -5f; // Нижня межа
```

```
    private Vector3 startPosition;
    private bool movingUp = true;
    private void Start()
    {
        startPosition = transform.position;
    }
    private void Update()
    {
        if (movingUp)
        {
            transform.Translate(Vector3.up * speed * Time.deltaTime);
            if (transform.position.y >= startPosition.y + upperLimit)
            {
                movingUp = false; // Змінюємо напрямок руху
            }
        }
        else
        {
            transform.Translate(Vector3.down * speed * Time.deltaTime);
            if (transform.position.y <= startPosition.y + lowerLimit)
            {
                movingUp = true; // Змінюємо напрямок руху
            }
        }
    }
}
```

Скрипт меню паузи:

```
public class PauseMenu : MonoBehaviour
{
    public Canvas backgroundCanvas;
    public int pauseCanvasSortingOrder = 2;
    public GameObject pauseMenu;
    public GameObject[] spriteObjects;
    public Transform currentSpawnPoint;


    private int defaultSortingOrder;
    private RectTransform backgroundCanvasRectTransform;
    private RectTransform pauseMenuRectTransform;
    private SpriteRenderer[] spriteRenderers;
    private int[] defaultSpriteSortingOrders;
    private void Start()
    {
        defaultSortingOrder = backgroundCanvas.sortingOrder;
        backgroundCanvasRectTransform =
backgroundCanvas.GetComponent<RectTransform>();
        pauseMenuRectTransform = pauseMenu.GetComponent<RectTransform>();
        spriteRenderers = new SpriteRenderer[spriteObjects.Length];
        defaultSpriteSortingOrders = new int[spriteObjects.Length];
        for (int i = 0; i < spriteObjects.Length; i++)
        {
            spriteRenderers[i] = spriteObjects[i].GetComponent<SpriteRenderer>();
            defaultSpriteSortingOrders[i] = spriteRenderers[i].sortingOrder;
        }
        pauseMenu.SetActive(false);
```

```
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            TogglePauseMenu();
        }
    }
    public void TogglePauseMenu()
    {
        bool isPaused = !pauseMenu.activeSelf;
        if (isPaused)
        {
            backgroundCanvas.sortingOrder = pauseCanvasSortingOrder;
            pauseMenu.SetActive(true);
            pauseMenuRectTransform.position =
backgroundCanvasRectTransform.position;
            pauseMenuRectTransform.sizeDelta =
backgroundCanvasRectTransform.sizeDelta;
            for (int i = 0; i < spriteObjects.Length; i++)
            {
                spriteRenderers[i].sortingOrder = pauseCanvasSortingOrder + 1;
            }
        }
        else
        {
            backgroundCanvas.sortingOrder = defaultSortingOrder;
            pauseMenu.SetActive(false);
```

```
            for (int i = 0; i < spriteObjects.Length; i++)
            {
                spriteRenderers[i].sortingOrder = defaultSpriteSortingOrders[i];
            }
        }
    }
    public void Continue()
    {
        TogglePauseMenu();
    }
    public void QuitToMainMenu()
    {
        SceneManager.LoadScene(0); // Завантажуємо сцену головного меню
    }

    public void RestartGame()
    {
        Hero.Instance.UpdateRespawnPosition(currentSpawnPoint.position);          //
Оновлюємо точку респавну в головного героя
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);          //
Перезавантажуємо поточну сцену
    }
}
```

Скрипт завантажувача сцен:

```
public class SceneLoader : MonoBehaviour
{
    public void Level1()
    {
        SceneManager.LoadScene(2);
    }
    public void Level2()
    {
        SceneManager.LoadScene(3);
    }
    public void Level3()
    {
        SceneManager.LoadScene(4);
    }
    public void LevelList()
    {
        SceneManager.LoadScene(1);
    }

    public void Exit()
    {
        Debug.Log("game close");
        Application.Quit();
    }

    public void ExitInManinMenu()
    {
        SceneManager.LoadScene(0);
```

```
    }
}
```

Скрипт для налаштування звуків:

```
public class SoundSetting : MonoBehaviour
{
    private AudioSource _audiosours;
    private float musicVolue = 1f;

    private void Start()
    {
        _audiosours = GetComponent<AudioSource>();
    }
    private void Update()
    {
        _audiosours.volume = musicVolue;
    }

    public void SetVolume(float vol)
    {
        musicVolue = vol;
    }
}
```

Оновлений скрипт створення зілль:

```
public class SpellSpawner : MonoBehaviour
{
    public GameObject potionPrefab; // Префаб зілля
    public Vector3 spawnPosition; // Координати спавну
```

```
    public float respawnDelay = 5.0f;
    private GameObject currentPotion;
    private void Start()
    {
        SpawnPotion();
    }
    private void Update()
    {
        if (currentPotion == null)
        {
            StartCoroutine(RespawnPotionAfterDelay());
        }
    }
    private void SpawnPotion()
    {
        currentPotion = Instantiate(potionPrefab, spawnPosition, Quaternion.identity);
    }
    private IEnumerator RespawnPotionAfterDelay()
    {
        yield return new WaitForSeconds(respawnDelay);
        if (currentPotion == null)
        {
            SpawnPotion();
        }
    }
}
```

Скрипт генератору небезпечних снарядів:

```
public class SpikeBallSpawner : MonoBehaviour
{
    public GameObject projectilePrefab;
    public float spawnDelay = 1.0f;
    public bool spawnRight = true;
    public float spawnX = 0.0f;
    public float spawnY = 0.0f;
    public float spawnZ = 0.0f;
    public float distance = 10.0f;
    private float lastSpawnTime = 0.0f;
    // Update is called once per frame
    void Update()
    {
        if (Time.time > lastSpawnTime + spawnDelay)
        {
            lastSpawnTime = Time.time;
            SpawnProjectile();
        }
    }
    void SpawnProjectile()
    {
        Vector3 position = new Vector3(spawnX, spawnY, spawnZ);
        GameObject projectile = Instantiate(projectilePrefab, position, Quaternion.identity);
        MoveForward projectileScript = projectile.GetComponent<MoveForward>();
        projectileScript.moveRight = spawnRight;
        projectileScript.distance = distance;
```

```
    }
}
```

Скрипт для знищення об’єктів об’єктами з відповідним тегом:

```
public class WallDestroyer : MonoBehaviour
{
    public string targetTag = "Wall"; // За замовчуванням, знищуються об'єкти з тегом "Enemy"
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Box"))
        {
            DestroyObjectsWithTag(targetTag);
        }
    }
    private void DestroyObjectsWithTag(string tag)
    {
        GameObject[] objectsToDestroy = GameObject.FindGameObjectsWithTag(tag);
        foreach (GameObject obj in objectsToDestroy)
        {
            Destroy(obj);
        }
    }
}
```

Скрипт зони яка сповільнює гравця:

```
public class SlowZone : MonoBehaviour
{
```

```
    [Header("Slow Zone Settings")]
    public float slowDownFactor = 0.5f;
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            Hero hero = collision.GetComponent<Hero>();
            if (hero != null)
            {
                hero.speed *= slowDownFactor;
            }
        }
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            Hero hero = collision.GetComponent<Hero>();
            if (hero != null)
            {
                hero.speed /= slowDownFactor;
            }
        }
    }
}
```

Скрипт порталів:

```
public class Portals : MonoBehaviour
{
    public Vector2 teleportPosition; // Координати, куди телепортується гравець
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            other.transform.position = new Vector3(teleportPosition.x, teleportPosition.y, other.transform.position.z);
        }
    }
}
```

Скрипт генератору рівнів:

```
public class LvlGenerator : MonoBehaviour
{
    [Header("Chunk Settings")]
    public GameObject[] levelChunks; // Масив чанків (prefab) для генерації
    public int numberOfChunks = 5; // Кількість чанків для генерації
    [Header("Spawn Settings")]
    public GameObject playerPrefab; // Префаб гравця
    public Transform spawnPoint; // Точка спавну гравця
    [Header("Positioning")]
    public float chunkWidth = 10f;
    private Vector3 nextChunkPosition;
    void Start()
    {
        GenerateLevel();
```

```
            SpawnPlayer();
        }
        void GenerateLevel()
        {
            nextChunkPosition = Vector3.zero;

            for (int i = 0; i < numberOfChunks; i++)
            {
                GameObject selectedChunk = levelChunks[Random.Range(0, levelChunks.Length)];
                Instantiate(selectedChunk, nextChunkPosition, Quaternion.identity, transform);
                nextChunkPosition += new Vector3(chunkWidth, 0, 0); // Зміщення для наступного чанка
            }
        }
        void SpawnPlayer()
        {
            if (playerPrefab != null && spawnPoint != null)
            {
                Instantiate(playerPrefab, spawnPoint.position, Quaternion.identity);
            }
            else
            {
                Debug.LogWarning("Не задано префаб або точку спавну для гравця!");
            }
        }
    }
```

Скрипт для таємних кімнат:

```
public class InvisibleWall : MonoBehaviour
{
    private Renderer objectRenderer; // Для візуального відображення об'єкта
    private Collider2D objectCollider; // Для обробки колізій
    private void Start()
    {
        objectRenderer = GetComponent<Renderer>();
        objectCollider = GetComponent<Collider2D>();
    }
    private void OnTriggerEnter2D(Collider2D other)
    {
        // Якщо гравець входить у тригер
        if (other.CompareTag("Player"))
        {
            HideObject();
        }
    }
    private void OnTriggerExit2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            ShowObject();
        }
    }
    private void HideObject()
    {
        objectRenderer.enabled = false;
        objectCollider.enabled = false;
```

```
    }
    private void ShowObject()
    {
        objectRenderer.enabled = true;
        objectCollider.enabled = true;
    }
}
```

Скрипт для платформи пастки:

```
public class FalingPlatform : MonoBehaviour
{
    [SerializeField] private float fallSpeed = 2f; // Speed at which the platform falls
    [SerializeField] private float respawnCheckDistance = 0.1f; // Tolerance for checking the respawn coordinates
    [SerializeField] private Transform respawnPoint; // The respawn point of the player

    private Vector3 initialPosition; // Initial position of the platform
    private GameObject player; // The player that triggers the platform

    private Rigidbody2D rb2D; // Rigidbody2D of the platform to control its physics

    private void Start()
    {
        initialPosition = transform.position; // Store the initial position of the platform
        rb2D = GetComponent<Rigidbody2D>();

        // Set the Rigidbody2D to Kinematic initially so it doesn't fall until triggered
        rb2D.bodyType = RigidbodyType2D.Kinematic;
```

```
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player")) // Check if it's the player
        {
            player = other.gameObject; // Store reference to the player
            rb2D.bodyType = RigidbodyType2D.Dynamic; // Allow the platform to fall under physics
        }
    }

    private void Update()
    {
        if (player != null && Vector3.Distance(player.transform.position, respawnPoint.position) <= respawnCheckDistance)
        {
            ResetPlatform();
        }
    }
    private void ResetPlatform()
    {
        transform.position = initialPosition; // Reset the platform's position to the initial position
        rb2D.bodyType = RigidbodyType2D.Kinematic; // Set the platform to kinematic so it stops falling
        rb2D.linearVelocity = Vector2.zero; // Stop any remaining motion
    }
}
```

Скрипт повільної платформи пастки:

```
public class TrapPlatform : MonoBehaviour
{
    [SerializeField] private float descentSpeed = 2f;
    [SerializeField] private float respawnCheckDistance = 0.1f; перевірки координат респавну
    [SerializeField] private Transform respawnPoint;
    private Vector3 initialPosition;
    private GameObject player;
    private Rigidbody2D rb2D;
    private bool isDescending = false;
    private void Start()
    {
        initialPosition = transform.position; // Зберігаємо початкову позицію платформи
        rb2D = GetComponent<Rigidbody2D>();
        rb2D.bodyType = RigidbodyType2D.Kinematic;
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player")) // Перевіряємо, чи це гравець
        {
            player = other.gameObject;
            isDescending = true;
            rb2D.bodyType = RigidbodyType2D.Kinematic;
        }
    }
    private void Update()
```

```
        {
            if (isDescending)
            {
                transform.position = new Vector3(transform.position.x, transform.position.y - descentSpeed * Time.deltaTime, transform.position.z);
            }
            if (player != null && Vector3.Distance(player.transform.position, respawnPoint.position) <= respawnCheckDistance)
            {
                ResetPlatform();
            }
        }
        private void ResetPlatform()
        {
            transform.position = initialPosition; позицію
            isDescending = false;
        }
    }
```

Скрипт часової пастки

```
    public class Trap : MonoBehaviour
    {
        [Header("Trap Settings")]
        public float triggerTime = 3f;
        public int damage = 1;
        private float timer = 0f;
        private bool playerOnTrap = false;
        private Hero hero; // Посилання на героя
```

```
private void Update()
{
    if (playerOnTrap)
    {
        timer += Time.deltaTime;

        if (timer >= triggerTime)
        {
            ApplyDamage();
            timer = 0f;
        }
    }
    else
    {
        timer = 0f;
    }
}
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("Player"))
    {
        playerOnTrap = true;
        hero = collision.GetComponent<Hero>();
    }
}
private void OnTriggerExit2D(Collider2D collision)
{
    if (collision.CompareTag("Player"))
    {
```

```
            playerOnTrap = false;
            hero = null;
        }
    }

    private void ApplyDamage()
    {
        if (hero != null)
        {
            hero.GetDamage();
        }
    }
}
```

Скрипт ворога дальнього формату:

```
public class EnemyShooter : MonoBehaviour
{
    public GameObject projectilePrefab;
    public Transform firePoint;
    public float fireRate = 1f;
    private float fireCooldown;

    private Animator animator;

    void Start()
    {
        animator = GetComponent<Animator>();
        fireCooldown = 0f;
    }
```

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

        if (fireCooldown <= 0f)
        {
            Shoot();
            fireCooldown = 1f / fireRate;
        }
    }

    void Shoot()
    {
        animator.SetTrigger("Shoot");
        Instantiate(projectilePrefab, firePoint.position, firePoint.rotation);
    }
}
```