

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалавр

(ступінь вищої освіти)

на тему РОЗРОБКА ІГРОВОГО ЗАСТОСУНКУ У ЖАНРІ ПРИГОДИ З
ЕЛЕМЕНТАМИ RPG НА ПЛАТФОРМІ UNITY
DEVELOPMENT OF AN ADVENTURE GAME WITH RPG ELEMENTS ON
THE UNITY PLATFORM

Виконала: студентка 4 курсу, групи КНТ-111сп
Спеціальності 121 Інженерія програмного
забезпечення

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

ГРИШКО Н.С.

(ПРІЗВИЩЕ та ініціали)

Керівник ДУБРОВІН В.І.

(ПРІЗВИЩЕ та ініціали)

Рецензент КАЗУРОВА А.Е.

(ПРІЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти бакалавр
Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
“ ” 2024 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТКИ

ГРИШКО Ніки Сергіївни

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка ігрового застосунку у жанрі пригоди з елементами RPG на платформі Unity. Development of an Adventure Game with RPG Elements on the Unity Platform

керівник проєкту (роботи) к.т.н., професор, ДУБРОВІН Валерій Іванович,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “24” квітня 2024 року № 168

2. Строк подання студентом проєкту (роботи) 10 червня 2024 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Порівняльний аналіз технологій розробки комп'ютерних ігор. 3. Розробка комп'ютерних ігор. 4. Проєктувати комп'ютерного додатку. 5. Розробка комп'ютерного додатку “Маленькі пригоди в Балабурсі”

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)

Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ДУБРОВІН В.І., професор		
Нормоконтроль	БСЛОВА А.В., асистент		

7. Дата видачі завдання « 15 » квітня 2023 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	2 тиждень	Розділ 2
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програмного забезпечення.	5 тиждень	Розділ 4, 5
7	Оформлення пояснювальної записки та документів до неї.	6 тиждень	Додатки
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студентка

(підпис) Ніка ГРИШКО,
(Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Валерій ДУБРОВІН
(Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 116 с., 43 рис., 3 дод., 6 джерел.

UNITY, C#, RPG, ІГРОВІ КОМП'ЮТЕРНІ ДОДАТКИ, PIXELART, MODELIO.

Об'єктом дослідження є процес розробки ігрових комп'ютерних додатків, що представляють собою програми, створені для розваг та відпочинку гравців у віртуальних світах.

Для створення комп'ютерних ігор існують різноманітні інструменти та технології.

Предметом дослідження є технології проєктування та розробки ігрових комп'ютерних додатків, які є ключовим елементом у сучасній індустрії розваг. Ця область охоплює всі аспекти: від створення ідеї та концепції гри до програмування, візуалізації, звукового оформлення та тестування. Ознайомлення з цією темою допомагає розуміти та застосовувати найкращі практики у сфері розробки ігор, що сприяє подальшому розвитку цього захоплюючого та перспективного сегменту індустрії програмного забезпечення.

Мета цього дослідження полягає в тому, щоб оволодіти навичками розробки ігрових додатків, зосереджуючись на створенні комп'ютерної 2D пригодницької гри з елементами RPG.

Під час розробки гри на Unity успішно створено систему діалогів та набір завдань для гравця. За допомогою програми "PixelArt" створено спрайти персонажів та за допомогою "draw.io" змодельовано їх діалоги, що впливають на сюжет. Крім того, використовуючи "modelio", розроблено гнучку архітектуру гри.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 116 pages, 43 figures, 3 appendices, 6 sources.

UNITY, C#, RPG, GAMING APPLICATIONS, PIXELART, MODELIO.

The object of the study is the process of developing gaming computer applications, which are programs created for the entertainment and recreation of players in virtual worlds.

There are various tools and technologies for creating computer games.

The subject of research is the technology of designing and developing gaming computer applications, which are a key element in the modern entertainment industry. This area covers all aspects, from ideation and game concept to programming, rendering, sound design and testing. Familiarity with this topic helps you understand and apply best practices in game development, helping to further develop this exciting and promising segment of the software industry.

The purpose of this study is to master the skills of game application development, focusing on the creation of a computer 2D adventure game with RPG elements.

During the development of the game on Unity, a dialogue system and a set of tasks for the player were successfully created. With the help of the "PixelArt" program, sprites of the characters were created and with the help of "draw.io" their dialogues, which affect the plot, were modeled. In addition, using "modelio", a flexible game architecture has been developed.

ЗМІСТ

	С.
Вступ	8
1 Аналіз предметної області	10
1.1 Роль ігор у сучасному світі	10
1.2 Аналіз популярних 2D ігор в жанрі RPG	11
1.3 Висновки за розділом 1	12
2 Порівняльний аналіз технологій розробки комп'ютерних ігор	14
3 Розробка комп'ютерних ігор	22
3.1 Етапи розробки гри	22
3.2 Пре-продакшн	24
3.3 Виробництво	27
3.4 Забезпечення якості	32
3.5 Випуск	34
3.6 Пост-продакшн	34
3.7 Висновки за розділом 3	35
4 Проєктування комп'ютерного додатку	37
5 Розробка комп'ютерного додатку “Маленькі пригоди в Балабурсі”	52
5.1 Розробка сюжету гри	52
5.2 Розробка дизайну ігрових локацій	56
5.3 Розробка ігрових персонажів та їх спрайтів	57
5.4 Програмна реалізація комп'ютерного додатку	60
5.5 Висновки за розділом 5	68
Висновки	69
Перелік джерел посилання	71
ДОДАТОК А Текст програми	72
ДОДАТОК Б Слайди презентації	106
ДОДАТОК В Відео-трейлер гри	116

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

- RPG – Role-Playing game;
2D – 2-dimensional;
NPC – non-playable character;
GDD – game design document.

ВСТУП

Ігри в сучасному світі стали не лише засобом розваги, а й важливим елементом культури та соціальної взаємодії. Завдяки швидкому розвитку технологій, ігрова індустрія набула неймовірних розмірів та впливу на суспільство. Від мобільних ігор на смартфонах, до складних віртуальних світів на персональних комп'ютерах та консолях. В наш час, ігри стають все більш доступними та різноманітними. Деякі ігри також використовуються для розвитку креативності або стратегічного мислення. У сучасному світі ігри вже давно перетнули межі простої розваги, ставши важливим аспектом культурного та соціального життя.

Для розробки гри “Маленькі пригоди в Балабурсі” використаємо ігровий рушій Unity. Він був обраний, бо його основні переваги включають потужний редактор, підтримку різних мов програмування, в випадку даної гри – C#, високу продуктивність та можливості розробки багатоплатформних проєктів.

Графічну частину гри створимо за допомогою безкоштовної програми PixelArt, яка дозволяє малювати спрайти для 2D ігор.

В процесі розробки гри, реалізуємо систему діалогів та множину завдань, які повинен виконати гравець щоб досягти ігрових цілей. В сюжеті оповідатиметься історія ігрового персонажа на ім'я "Ронн", якого викрали та дезорієнтували. Завдання гравця допомогти герою знайти спосіб зцілитися. Він може знайти союзників, врятувати поселення та їх цілительку від монстрів. За допомогою програми "PixelArt" було реалізовано спрайти ігрових персонажів, за допомогою "draw.io" – змодельовано діалоги ігрових героїв, в деяких з яких гравцю дається вибір, який впливає на подальший розвиток сюжету у грі, а за допомогою "modelio" розроблено гнучку архітектуру програмного продукту.

Під час розробки гри на Unity успішно створено систему діалогів та набір завдань для гравця. За допомогою програми "PixelArt" створено спрайти персонажів та за допомогою "draw.io" змодельовано їх діалоги, що впливають на сюжет. Крім того, використовуючи "modelio", розроблено гнучку архітектуру гри.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Роль ігор у сучасному світі

Роль ігор у сучасному світі є досить значною і різноманітною. Ігри стали не лише розважальним видом діяльності, але й мають великий вплив на різні аспекти життя суспільства. Ось кілька ключових аспектів, що варто врахувати:

- розваги. Ігри є важливим джерелом розваг для мільйонів людей у всьому світі. Вони надають можливість відпочити, розважитися і знайти спільну мову з іншими гравцями;

- навчання та освіта. Ігри можуть бути потужним інструментом навчання. Вони допомагають розвивати різноманітні навички, такі як проблемне мислення, стратегічне планування, співпраця та креативність;

- соціальна взаємодія. Онлайн-ігри дозволяють людям з усього світу спілкуватися та взаємодіяти один з одним. Це може сприяти розвитку соціальних навичок та формуванню дружби;

- професійний геймінг. Професійні геймери, а також ігрові турніри та чемпіонати, стали значною частиною геймінг-індустрії. Вони надають можливість кар'єрного зростання та заробітку грошей;

- культурний вплив. Ігри часто мають великий культурний вплив, включаючи фільми, музику, книги та інші форми мистецтва. Вони можуть впливати на спосіб мислення та уявлення про світ;

- здоров'я і фітнес. Деякі ігри, такі як віртуальна реальність (VR) ігри та ігри з руховим керуванням, можуть сприяти фізичному здоров'ю та фітнесу;

- технологічний прогрес. Ігрова індустрія є драйвером технологічного прогресу, сприяючи розвитку комп'ютерної графіки, штучного інтелекту, віртуальної реальності та інших інновацій.

У цілому, ігри відіграють важливу роль у сучасному світі, впливаючи на розвиток технологій, культури, освіти та соціальних відносин.

1.2 Аналіз популярних 2D ігор в жанрі RPG

Аналіз популярних 2D ігор в жанрі RPG дає можливість краще зрозуміти тенденції та особливості ігор цього жанру. Розглянемо на прикладі трьох популярних 2D RPG ігор та основні аспекти їхнього успіху.

Undertale від розробників Toby Fox.

У свій час ця гра стала найпопулярнішою грою у жанрі RPG головоломки.

Геймплей. Undertale відома за свою унікальну бойову систему, яка комбінує елементи ходових боїв з елементами реального часу. Гравці мають можливість вибирати, чи хочуть вони атакувати, використовувати діалог або спробувати мирно вирішити конфлікт. Гравці можуть взаємодіяти з монстрами у бойових сценах, спробуючи змусити їх змінити свої шляхи без насильства. Вирішення конфліктів мирним чином або виконання злочинів впливає на сюжет та кінцівку гри.

Історія та персонажі. Гра має запам'ятовувану історію, де головний герой – дитина, яка потрапляє у підземне царство, населене монстрами, після того, як впала у глибокий тунель. Головна мета гравця – вибрати, чи буде вони зберігати мир з монстрами чи боротися з ними,

Атмосфера. Undertale має унікальний, добре прорисований світ, який є сумішшю фентезійних та сучасних елементів. Підземелля, де відбувається більшість дії, наповнене загадками, секретами та різноманітними монстрами.

Stardew Valley від розробників ConcernedApe.

Геймплей. Stardew Valley комбінує елементи фармінгу, симуляції та RPG. Гравці можуть насолоджуватися широким спектром активностей, від фермерства до соціальної взаємодії та пригодницьких випробувань, що робить гру дуже захоплюючою та різноманітною.

Історія та персонажі. це один із ключових аспектів, які роблять гру захоплюючою та цікавою для гравців. Головний герой успадковує невелику ферму від свого дідуся в Стардью Веллі, містечку, що знаходиться у рівнинній місцевості. Він втомився від міського життя та вирішує залишити свою роботу в корпорації, щоб розпочати нове життя на фермі. Головний герой має можливість відновити ферму, розвивати її, взаємодіяти з місцевими жителями та вирішувати різні завдання та проблеми.

Атмосфера. Stardew Valley занурює вас у спокійне та мальовниче сільське життя. Стиль мистецтва гри, який нагадує піксельне мистецтво, створює чарівну та затишну атмосферу. Кожен персонаж, місце та об'єкт має свій власний унікальний вигляд, що додає грі особливого шарму. У грі можлива взаємодія з місцевими мешканцями, кожен з яких має свою власну історію та особистість, додає грі глибину. Ви можете вступити в дружбу, побудувати романтичні стосунки або просто поспілкуватися з ними, відчувачи себе частиною спільноти.

CrossCode від розробників Radical Fish Games.

Геймплей. Гра CrossCode поєднує в собі різноманітні елементи RPG, екшну та головоломок, створюючи захоплюючий інтерактивний досвід для гравців.

Історія та персонажі. Гра розповідає історію Леи, головної героїні, яка грає в MMORPG-подібну віртуальну реальність, відому як CrossWorlds. Лея втратила пам'ять і намагається відновити свою минулу життєву історію шляхом відкриття різних таємниць у світі CrossWorlds.

Атмосфера. У цілому, атмосфера в CrossCode характеризується своєрідним стилем мистецтва, захоплюючим світом та емоційно насиченою історією, що робить гру цікавою та запам'ятовуваною для гравців.

1.3 Висновки за розділом 1

Ігри відіграють важливу роль у сучасному світі, як розважальна діяльність, що має значний вплив на різні аспекти життя суспільства. Вони є джерелом розваг для мільйонів людей, а також потужним інструментом навчання та освіти, сприяючи розвитку різноманітних навичок. Ігри сприяють соціальній взаємодії, культурному впливу, розвитку професійного геймінгу та технологічному прогресу.

Було проаналізовано 3 доволі популярні 2D RPG гри: Undertale, Stardew Valley та CrossCode, які відзначаються своїм унікальним геймплеєм, історією та атмосферою. Зрозуміли, що кожна з цих ігор має свої унікальні особливості, що роблять їх привабливими для гравців. Undertale славиться своєю унікальною бойовою системою та глибокою історією, Stardew Valley пропонує затишне сільське життя та багатофункціональність, а CrossCode вражає своєрідним стилем мистецтва та захоплюючою головоломками.

У цілому, ігри мають значний вплив на сучасне суспільство як засіб розваг, навчання, соціальної взаємодії та культурного впливу. Популярні 2D RPG ігри відображають цей вплив через свою унікальну геймплей, історію та атмосферу, які приваблюють гравців своєю різноманітністю та захоплюючістю.

2 ПОРІВНЯЛЬНИЙ АНАЛІЗ ТЕХНОЛОГІЙ РОЗРОБКИ КОМП'ЮТЕРНИХ ІГОР

2.1 Класифікація та аналіз комп'ютерних ігор

У епоху стрімкого науково-технічного прогресу спостерігається явище, що відоме як розподіл у сприйнятті світу молоді. Це викликає потребу у виробництві нових методів та технологій для розвитку якостей, які сприяють активізації розумових здібностей під час навчання через комп'ютерні ігри. Галузь геймінгу зосереджена не лише на підвищенні інтелектуального потенціалу, але й на створенні розважальних ігрових програм як засобів для відпочинку.

Отже, ігрові програми сьогодні набули значної популярності в різних сферах і мають великий попит. Тому дослідження та розвиток ігрових технологій є важливим завданням сучасності.

Комп'ютерна гра – це програмне забезпечення, призначене для забезпечення ігрового процесу. Вона може взаємодіяти з гравцями і навіть виступати як партнер. Для керування ігровим процесом використовуються різні пристрої введення, такі як комп'ютерна миша, клавіатура, камера, джойстик тощо. [1].

Ігра може бути комбінацією наявних жанрів або не відноситися до жодного з них. Незважаючи на це, існує класифікація комп'ютерних ігор (рис. 2.1) , яка розвивалася в процесі їхнього розвитку. До неї входять такі категорії:

- 3D-шутери;
- Аркада;
- Бійки / Файтинги;
- Платформер;
- Скролер;
- Симулятор;
- Стратегія;

- Спортивні ігри;
- Головоломки, ігри на логіку;
- Традиційні, або ж настільні ігри [2].

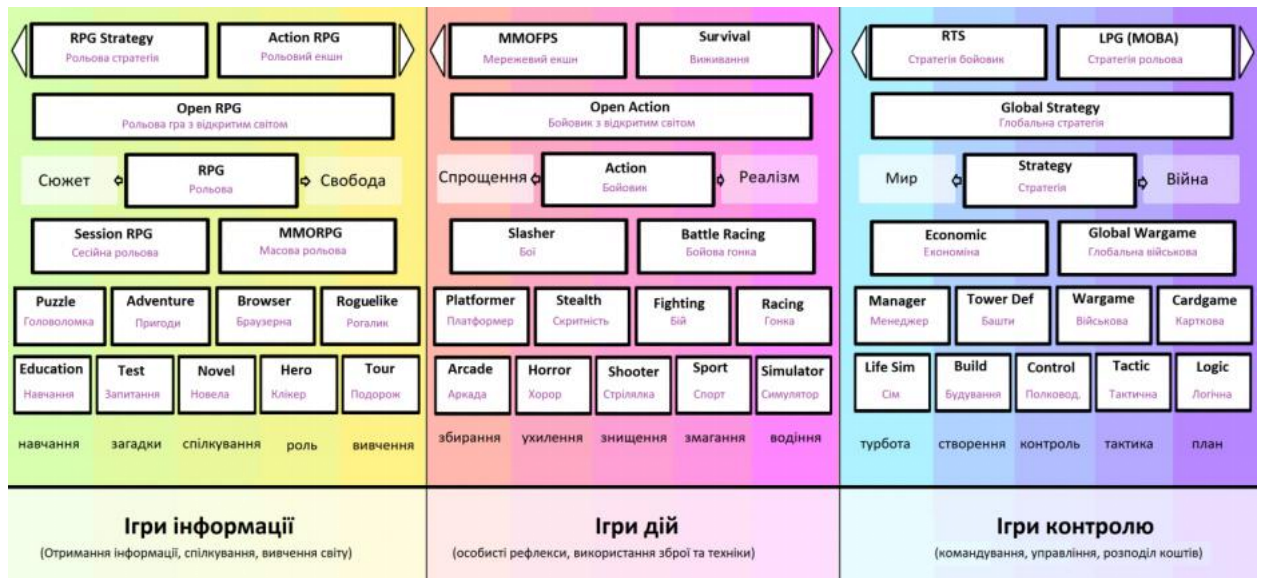


Рисунок 2.1 – Класифікація ігор по жанрам

Класифікація комп'ютерних ігор також може здійснюватися за іншими параметрами:

- за ігровими платформами;
- за кількістю гравців (одно- та багатокористувацькі ігри);
- за умовами розповсюдження (shareware, freeware);
- за видом поставки (на оптичному носії, коробочна версія, доповнення, антологія тощо).

Класифікація ігор за жанрами ґрунтується на принципі розробки ігрових стратегій. Таким чином, можна виділити три основні групи ігрових програм: дії, інформаційні ігри та ігри контролю, які сприяють різним аспектам розвитку галузі.

Інформаційні ігри мають на меті навчати гравців новим знанням у форматі гри, активізуючи навчальний процес.

Ігри дій можна розглядати як ефективні тренажери, спрямовані на покращення реакції та точності рухів в екстремальних умовах.

Ігри контролю включають розробку та реалізацію власних стратегій для досягнення переваг у майбутньому.

Стадія прихильності характеризується зниженням ігрової активності та зміщенням психологічного балансу особистості до нормального стану. У цей період взаємодія з комп'ютером нагадує натягнутий, але міцний гудзик: людина тримає дистанцію від комп'ютера, але не може повністю позбутися психологічної прихильності до комп'ютерних ігор. Ця стадія може тривати все життя, залежно від того, як швидко зникає захопленість [3].

Розвиток комп'ютерних ігор зростає щодня, разом з ним зростає і кількість людей, які в них грають. Цих людей часто називають "геймерами" або "комп'ютерними фанатами". Їхня основна зайнятість - гра за комп'ютером. Це може призвести до психологічних проблем, особливо у випадку захоплення рольовими іграми, які відіграють велику роль у формуванні психологічної залежності та впливають на особистість гравця.

Динаміка розвитку комп'ютерної залежності може бути описана таким чином:

- початкова стадія легкої захопленості. Після декількох разів гри в комп'ютерні ігри людина починає відчувати неусвідомлену потребу у грі. Задоволення від гри супроводжується позитивними емоціями, і людина починає шукати можливості для гри, хоча це ще не стає домінуючим фактором у її житті.

- стадія заглибленої захопленості. Людина починає систематично проводити час за грою, і гра стає важливим елементом її щоденного життя. Нова потреба в грі виявляється в ієрархії її потреб.

- стадія залежності. На цій стадії гра стає основним фактором, що визначає поведінку людини і змінює її цінності. Залежність може виражатися в індивідуальній формі або у формі соціалізованої залежності, коли гра стає центром соціальних взаємодій і стосунків [4].

Індивідуалізована форма ігрової залежності – це екстремальна стадія, коли порушуються не лише нормальні особистісні орієнтації, але й

спілкування з оточуючим світом. Функціонування психіки починає відображати вплив віртуальної реальності, що викликає значні перешкоди для сприйняття об'єктивного світу.

Соціалізована форма ігрової залежності відрізняється тим, що вона включає в себе підтримку соціальних контактів з іншими залежними особами. Головна мотивація для гри в цьому випадку має змагальний характер. Вплив цієї форми залежності на психіку людини менш шкідливий порівняно з індивідуалізованою формою.

Стадія прихильності характеризується згасанням ігрової активності і зміщенням психологічного балансу особистості в бік нормальності. В цей період взаємодія з комп'ютером може бути порівняна з натягнутим, але міцним гудзиком: людина тримає дистанцію від комп'ютера, але не може повністю відірватися від психологічної симпатії до комп'ютерних ігор. Ця стадія може тривати протягом усього життя, залежно від того, наскільки швидко згасає захопленість [5].

Таким чином, ігрова промисловість використовується як потужний інструмент для створення навчальних симуляцій, емуляцій, і тренувальних програм, спрямованих на покращення реакції гравців та впровадження власних стратегій розвитку. Окрім розважальних можливостей, сучасні ігрові програми мають на меті виконувати розвивальні та тренувальні функції через інтерактивну взаємодію людини з комп'ютером.

2.2 Класифікація та аналіз програмного забезпечення для створення комп'ютерних ігор

Обираючи програмне забезпечення для розробки ігор, важливо врахувати конкретні потреби вашого проєкту, ресурси, якими ви володієте, і ваш рівень досвіду у розробці ігор. Однак, ось кілька програмних засобів, які часто рекомендується розглянути.

Unity. Як вже згадувалося, Unity є одним з найпопулярніших ігрових двигунів з великою кількістю ресурсів, активною спільнотою та кросплатформеним підходом. Це хороший вибір для широкого спектра проєктів, від маленьких інді-ігор до великих комерційних проєктів.

Unreal Engine. Unreal Engine від Epic Games також є потужним інструментом для розробки ігор з вражаючою графікою та функціональністю. Він часто використовується для створення великих AAA-ігор та ігор віртуальної реальності.

Godot Engine. Godot – це безкоштовний та відкритий ігровий двигун з власним скриптовим мовою GDScript. Він відмінно підходить для невеликих інді-проєктів та має добре організований редактор та документацію.

GameMaker Studio. Це інструмент, що спрямований на початківців у галузі розробки ігор. GameMaker має простий у використанні інтерфейс та вбудовану мову програмування, що робить його ідеальним для невеликих 2D-ігор.

Construct. Ще один інструмент для розробки 2D-ігор, який спрощує процес створення ігор без програмування. Він має візуальний інтерфейс та легко вивчається для новачків.

2.3 Обґрунтування вибору програмного забезпечення

Для роботи над дипломною роботою було обране програмне забезпечення Unity.

Unity – один з найпопулярніших та потужних ігрових двигунів у світі, і вибір його може бути обґрунтованим з кількох причин:

- кросплатформеність. Unity дозволяє розробляти ігри для різних платформ, включаючи ПК, мобільні пристрої, консолі, віртуальну реальність та діджитал-аудіовізуальні пристрої. Це робить його чудовим вибором для розробників, які хочуть долучити свою гру до максимально можливого аудиторії;

- легкий у використанні. Unity має інтуїтивний інтерфейс та велику кількість документації, яка спрощує процес вивчення та використання. Це робить його відмінним вибором для початківців у галузі розробки ігор;
- велика спільнота користувачів. Unity має велику та активну спільноту користувачів, що означає, що ви завжди зможете знайти підтримку, поради та відповіді на свої запитання в інтернеті;
- розширюваність. Unity підтримує розширення та плагіни, що дозволяє розробникам розширювати функціональність движуна та використовувати сторонні інструменти для поліпшення розробки;
- велика кількість ресурсів. Unity має велику кількість безкоштовних та платних ресурсів, таких як готові моделі, ассети, tutorіали, що дозволяє прискорити розробку гри та знизити витрати на розробку;
- потужність. Unity має потужний графічний движун, який дозволяє створювати 2D та 3D ігри з високою якістю графіки та анімацією.

Отже, для розробки гри я обрала движун Unity, бо використовуючи його я отримую доступ до потужного та високопродуктивного інструменту, який пропонує велику кількість можливостей, ресурсів та підтримки спільноти.

Для роботи над дипломною роботою було обрано мову програмування C++, через такі причини:

—) висока продуктивність:

- 1) C++ дозволяє створювати високопродуктивні ігри завдяки прямому управлінню пам'яттю і оптимізованому коду;
- 2) використовується для розробки багатьох AAA-ігор, де критичними є продуктивність та низькорівневий контроль;

—) гнучкість та контроль:

- 1) більший контроль над апаратним забезпеченням та можливість оптимізувати код під специфічні вимоги гри;
- 2) можливість використання власних алгоритмів та структур даних для максимального оптимізації;

三) широке використання в індустрії:

1) багато провідних ігрових движків, таких як Unreal Engine, використовують C++;

2) великий досвід використання в ігровій індустрії забезпечує наявність великої кількості ресурсів, бібліотек та фреймворків;

四) можливість інтеграції з іншими мовами та технологіями. Легко інтегрується з іншими мовами програмування та технологіями, що дозволяє використовувати змішану архітектуру для проєктів.

Для роботи над дипломною роботою було обрано редактор коду Visual Studio Code, через такі причини:

- легка вага та швидкість. Visual Studio Code є легким редактором коду, який швидко завантажується та працює без затримок;

- розширюваність. Підтримка численних плагінів та розширень дозволяє налаштувати VS Code під конкретні потреби розробника, включаючи підтримку мов програмування, інтеграцію з системами контролю версій та багато іншого;

- безкоштовність та доступність. Visual Studio Code є безкоштовним і доступним для різних операційних систем, що робить його популярним вибором серед розробників;

- підтримка C# та C++. Завдяки плагінам, Visual Studio Code підтримує роботу з C# та C++, що дозволяє використовувати його як основний редактор для розробки в Unity та інших середовищах.

2.4 Висновки за розділом 2

У сучасному світі ігрові програми набули значної популярності і використовуються у різних галузях діяльності. Це пояснюється потребою у виробництві нових методів та технологій для розвитку розумових здібностей, а також як засіб відпочинку. Галузь геймінгу активно працює над підвищенням інтелектуального потенціалу і створенням розважальних ігор.

Ігрові програми можуть бути різноманітними за жанром та функціональністю, і вони використовують різні пристрої введення для здійснення ігрового процесу. Таким чином, дослідження та розвиток ігрових технологій залишається актуальним завданням сучасності.

3 РОЗРОБКА КОМП'ЮТЕРНИХ ІГОР

3.1 Етапи розробки гри

Процес розробки ігор, який є наріжним каменем у створенні захоплюючого ігрового досвіду, ретельно структурований на окремі етапи. Кожен етап служить незамінною контрольною точкою, яка гарантує, що гра просувається до передбачуваного успіху. Як правило, процес розробки гри складається з п'яти основних етапів розробки ігор: пре-продакшн, виробництво, забезпечення якості, запуск і пост-продакшн (рис. 3.1).

ЕТАПИ РОЗРОБКИ ІГОР

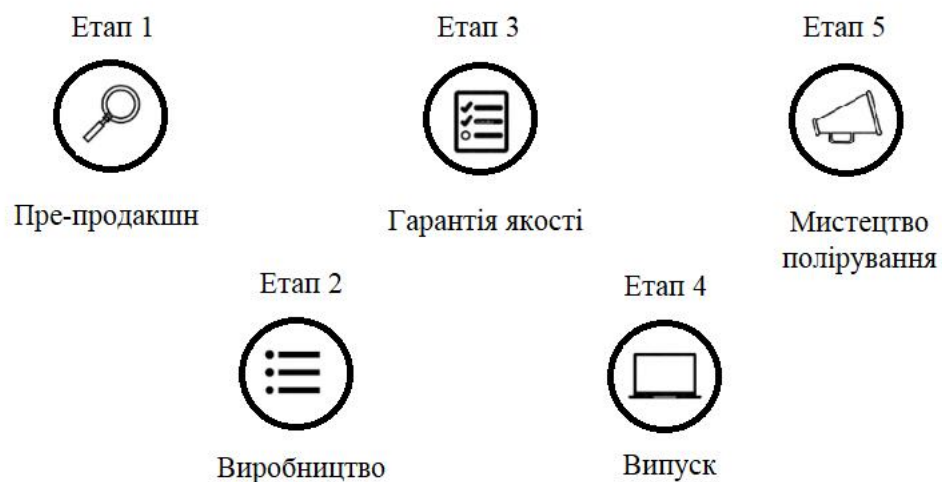


Рисунок 3.1 – Етапи розробки ігор

Першим етапом у процесі розробки гри є препродакшн. Цей етап закладає основу для всього циклу розробки гри. Він включає мозковий штурм ідей, концептуалізацію ігрової механіки та складання попереднього дизайну. Цей етап є ключовим, оскільки він задає тон і напрямок до кінця циклу розробки гри, гарантуючи, що всі наступні етапи відповідають основним цілям гри.

Дотримуючись плану, встановленого на етапі препродакшену, на перший план виходить виробничий етап розробки гри. На цьому етапі відбувається основна частина розробки ігор. Художники, програмісти та геймдизайнери співпрацюють, щоб втілити концептуалізовані ідеї в життя, перетворюючи їх на відчутні ігрові елементи.

Наступним етапом розробки гри, який має важливе значення в процесі розробки гри, є забезпечення якості. Кожна гра, незалежно від її складності, повинна пройти ретельне тестування, щоб виявити будь-які збої, помилки або проблеми з продуктивністю. Цей етап полягає не лише у виявленні проблем, а й у вдосконаленні ігрової механіки, покращенні графіки та забезпеченні того, щоб загальний користувацький досвід відповідав встановленим стандартам.

Після того, як гра була оптимізована на етапі забезпечення якості розробки гри, вона переходить до довгоочікуваного етапу запуску. Це момент, коли гра випускається для публіки, дозволяючи гравцям по всьому світу зануритися в нещодавно створене віртуальне середовище.

Однак процес розробки гри не зупиняється після запуску гри. Останній етап, постпродакшн-технічне обслуговування, гарантує, що гра продовжуватиме забезпечувати безперебійний досвід своїм гравцям. Регулярні оновлення, додатковий контент і оперативне виправлення помилок – ось деякі із завдань, які виконуються на цьому етапі процесу розробки гри.

Зрештою, процес розробки ігор з його багатогранними етапами розробки ігор – це заплутана подорож, яка перетворює просту ідею на гру, в яку можна грати. Важливість кожного етапу в розробці гри не можна недооцінювати, оскільки вони в сукупності сприяють успіху гри.

Від зародження ідеї на етапі препродакшну, через ретельні випробування на забезпечення якості до постійного вдосконалення післявиробничого обслуговування, кожен етап процесу розробки гри відіграє вирішальну роль. Цей систематичний процес гарантує, що ігри будуть не

тільки приємними, але й найвищої якості, що робить весь шлях розробки ігор справді корисним [6].

3.2 Пре-продакшн

Розробка відеоігор – це складний процес, що включає серію ретельно спланованих етапів. Вона розпочинається з ключового етапу планування, який суттєво впливає на всі подальші цикли розробки.

Початок процесу розробки комп'ютерної гри вимагає детального збору даних про майбутній продукт. Це передбачає розуміння технічних аспектів, які визначатимуть розробку гри:

- менеджери проєктів і власники продуктів беруть на себе ключову роль на цих ранніх етапах розробки гри. Вони визначають цілі проєкту та формують основну концепцію гри;

- аналіз ринку та конкурентів. У процесі розробки гри необхідно враховувати сучасні тенденції в ігровій індустрії. Важливо визначити, які ігри та жанри найбільш популярні та приносять значний дохід. Цільова аудиторія. Ключовим етапом у процесі розробки гри є визначення цільової аудиторії. Знання уподобань гравців дає змогу розробникам удосконалювати гру, забезпечуючи її глибокий зв'язок з аудиторією;

- вибір платформи. Це рішення в процесі розробки гри є подвійним. На це вплинула популярність платформи серед користувачів, а також досвід і можливості команди розробників у кодуванні;

- розподіл ресурсів. Один із інструментальних етапів розробки гри передбачає стратегію розміщення людських і технічних ресурсів. Це включає визначення необхідних спеціалістів і обладнання та складання графіка процесу розробки відеоігор, що забезпечує безперебійне виконання проєкту;

- концепт-арт. Розробка відеоігор значною мірою завдячує своєю візуальною пишністю концепт-художникам. Ці майстри створюють

персонажів, пейзажі, реквізит і безліч ігрових ресурсів, що надає грі її виразної візуальної ідентичності;

- тип монетизації. Модель доходу від гри є життєво важливим рішенням у процесі розробки гри. Чи буде гра отримувати дохід від реклами, покладатися на внутрішньоігрові покупки чи її можна буде придбати за встановленою ціною;

- аналіз ризиків. Часто недооцінений, але критичний етап у процесі розробки гри включає оцінку потенційних проблем. Розробники ідентифікують потенційні ризики та розробляють стратегії їхнього усунення, забезпечуючи безперешкодний процес розробки гри;

- маркетингова стратегія. Оскільки розробка гри наближається до завершення, складання маркетингового плану стає найважливішим. Мета полягає в тому, щоб розробити стратегії, які підвищать привабливість гри, гарантуючи, що вона захопить широку аудиторію.

Після завершення цих етапів ретельного збору даних і розробки стратегії розробки гри відкривається шлях до документу про дизайн гри (GDD). GDD є ключовим елементом у процесі розробки гри. Цей документ детально описує складні деталі гри, охоплюючи її концепцію, жанр, сюжетну лінію, тип ігрового процесу, ігрову механіку, конфігурацію рівнів і загальне середовище.

Процес розробки гри об'єднує різноманітні етапи, кожен з яких відіграє критично важливу роль у формуванні кінцевого продукту. Кожен етап відіграє найважливішу роль у формуванні кінцевого продукту гри. Від початкових етапів планування до ретельного створення GDD, кожен етап у процесі розробки гри гарантує, що остаточна гра є взірцем досконалості в динамічному світі розробки відеоігор.

Розробка відеоігор – це не лише технічне завдання, а й творчий процес, що поєднує мистецтво зі складною механікою. Кожен етап цього процесу є ключовим, адже розробка гри поєднує мистецтво, технології та бізнес-стратегії. Глибоке занурення в цей процес дозволяє нам розкрити

історії, стратегії та основні тенденції індустрії, яка постійно формує глобальний ландшафт розваг:

- натхнення за грою. За кожною грою криється іскра натхнення. Незалежно від того, чи це сюжетна лінія, яка викликає враження, чи захоплююча історична подія, чи проста, але переконлива механіка, ця початкова ідея діє як Полярна зірка, яка спрямовує весь процес розробки гри. Цей початковий задум розвивається під час генерування ідей, розкадровок і грубих ескізів, поступово перетворюючись на повноцінну концепцію гри;

- петлі зворотного зв'язку. Невід'ємною частиною етапів розробки гри є зворотній зв'язок. Розробники часто випускають бета-версії для обраної групи, збираючи статистику для покращення ігрового процесу, графіки та інтерфейсу користувача. Цей ітеративний процес гарантує, що коли гра виходить на ринок, вона не тільки відповідає, але й перевершує очікування гравців;

- співпраця та динаміка команди. Розробка відеоігор — це не одиночна справа. Він процвітає завдяки співпраці. Від дизайнерів ігор, програмістів і звукорежисерів до сценаристів і професіоналів з маркетингу, кожен учасник має унікальний набір навичок. Поєднання цих талантів протягом усього процесу розробки гри призводить до продукту, який є технічно надійним і художньо глибоким;

- культурна та соціальна значимість. Ігри часто відображають суспільство, відображаючи культурні нюанси та історичні події, а іноді навіть кидаючи виклик суспільним нормам. У процесі розробки гри розробники зобов'язані забезпечити резонанс своєї гри серед різноманітної аудиторії. Це передбачає широкі дослідження, культурні консультації, а іноді навіть екскурсії для перевірки автентичності;

- емоційна залученість. В основі будь-якої гри, що запам'ятовується, лежить її здатність викликати емоції. Незалежно від того, чи це викид адреналіну від високооктанового екшену, чи зворушливі моменти, процес

розробки гри передбачає створення досвіду. Йдеться про створення моментів, які залишаються з гравцями ще довго після завершення гри;

- інновації та технологічний прогрес. Світ розробки відеоігор постійно розвивається. З появою віртуальної реальності, доповненої реальності та наративів, керованих штучним інтелектом, розробники постійно напружені. Сьогоднішній процес розробки ігор пов'язаний не тільки з використанням цих нових технологій, але і з розповіддю;

- економічний ефект. Незаперечно, що ігри мають величезний економічний вплив. Крім прямих доходів, вони впливають на продаж товарів, екранізацію фільмів і навіть туризм для ігор, які відбуваються в знакових місцях. На етапах розробки гри розробка стратегії для таких побічних ефектів може посилити довгостроковий успіх гри та культурний слід [6].

3.3 Виробництво

В процесі розробки відеоігор стадія виробництва є центральною та фундаментальною частиною всього циклу. Як найбільш тривала та трудомістка фаза, вона включає численні види діяльності та інновації, які розподіляються на менші, але важливі етапи розробки:

Етап 1. Прототипування – етап реалізації концепцій.

На початку стадії виробництва ми познайомимося з важливістю прототипування гри. Це можна порівняти з генеральною репетицією перед великою виставою, яка надає відчутне уявлення про основну ігрову механіку та визначає напрямок подальшого розвитку гри задаючи тон наступним етапам розробки. Для підкреслення значення цього етапу варто згадати приклад шведського військового корабля "Vasa" XVII століття.

Незважаючи на значні інвестиції та майстерність, корабель затонув під час свого першого плавання, що свідчить про помилку в проєктуванні. Ця історична помилка підкреслює важливість етапу створення прототипу в

процесі розробки гри. Без перевірки навіть найгеніальніші ідеї ризикують спіткати подібну долю.

Етап 2. Створення візуального вмісту – малювання всесвіту гри.

Кожна гра є складною системою, що складається з багатьох елементів: персонажів, які викликають у гравців різні емоції, реквізиту, що додає автентичності, захоплюючого середовища, яке занурює гравців, та інших важливих ігрових активів. Складність і складність цього етапу в процесі розробки гри значно відрізняються в залежності від обраного формату гри 2D або 3D.

2D ігри. Створення 2D гри, хоч і менш складне, але не менш мистецьке. Процес розробки цих візуальних ігор нагадує 3D-ігри, хоча й спрощений. Їх створення передбачає створення ескізів, оживлення за допомогою кольорів і, нарешті, анімацію. Внутрішня простота 2D означає, що процес часто є швидким і залежить від глибини деталізації вмісту. Однак це не знижує вимог до мистецького підходу та відданості проєкту.

3D ігри. Початок розробки 3D-відеоігор – це складний процес, що охоплює багатоступеневі етапи розробки ігор. Процес починається з багатокутного моделювання, закладаючи основну структуру скелета за допомогою геометричних фігур. У міру розвитку гри скульптура додає глибини та характеру. Текстурування підкреслює шкіру, надаючи грі вигляд і відчуття. З іншого боку, оснащення та анімація вдихають життя в ці моделі, готуючи основу для захоплюючого досвіду. Процес розробки 3D-ігор є вимогливим, що потребує використання спеціалізованого програмного забезпечення та інструментів. Крім того, це вимагає глибшого набору навичок, гарантуючи, що кінцевий продукт є свідченням професіоналізму та майстерності.

По суті, на стадії виробництва в циклі розробки гри бачення втілюються в реальність. Це місце, де перші іскри з етапів розробки гри отримують пальне відданості та досвіду, готуючи основу для шедеву в

майбутньому. Цей етап являє собою гармонійне поєднання творчості та технології, де кожен крок і кожна дія мають важливе значення.

Етап 3. Розробка дизайну рівня гри.

Глибоко занурюючись у процес розробки дизайну на рівні гри, можна порівняти це з написанням розділів епічного роману. Подібно до того, як автор ретельно обирає кожне слово, фразу та поворот сюжету, дизайнер рівнів ретельно створює кожен компонент, гарантуючи, що кожен дюйм ігрового ландшафту є цілеспрямованим і привабливим. Якщо поглянути на незліченну кількість етапів розробки гри, то стає очевидним, що це щось більше, ніж просто кроки; це ключові розділи, які визначають суть процесу розробки відеоігор.

Розглянемо це через призму процесу розробки гри. Цей процес, складна та динамічна подорож, покликаний вдихнути життя у всесвіт гри. З кожним прогресом у циклі розробки гри історія розгортається, знайомлячи гравців з новими викликами, ландшафтами та таємницями. Оповідь гри розвивається, черпаючи натхнення з кожного етапу, таким чином створюючи гобелен досвіду, який одночасно зачаровує та захоплює гравців.

Етап 4. Створення логіки гри – більше, ніж просто прогрес.

У центрі циклу розробки гри лежить етап, присвячений дизайну рівнів. Тут досвідчені спеціалісти ретельно формують логіку гри, адаптуючи її до унікального колориту жанру. Незалежно від того, чи це поступове ускладнення викликів після завершення захоплюючого лінійного квесту, чи привабливість навігації великим відкритим світом, етапи розробки гри забезпечують динамічний досвід для гравця. Кожен момент і кожне випробування, з якими стикається гравець, демонструють майстерність у дизайні рівнів.

Дизайн рівнів, незважаючи на те, що він невід’ємний, є лише одним із багатьох аспектів у процесі розробки ігор. Глибше занурюючись у процес розробки гри, взаємозалежність кожного етапу стає відчутно очевидною. Від

концептуалізації основної ідеї гри до завершальних етапів забезпечення якості, кожен етап є важливою складовою великої системи.

Завдяки різним етапам процесу розробки гравець стає не просто спостерігачем, а активним учасником, повністю залученим і зануреним у гру. Важливі не лише механіка чи графіка; йдеться про створення незабутньої подорожі, яка залишиться з гравцем ще довго після завершення гри.

Етап 5. За межами рівнів. Формування концепції гри.

Розробка гри не обмежується лише просуванням рівнів. Сюжет стає багатшим завдяки вплетенню різноманітних елементів, життєво важливих для захоплюючого ігрового процесу. Центральне місце займає дизайн ігрової економіки, яка керує діями та рішеннями гравця. Цей аспект розробки гри є критично важливим, оскільки визначає винагороди, наслідки та загальний баланс, за яким орієнтуються гравці.

Не менш важливими є тонкі нюанси сюжету, перипетії та повороти, що викликають емоційне залучення гравців. З кожним етапом розробки ці сюжетні елементи додають напруги та очікування. Підказки та натяки, розміщені по всій грі, слугують дороговказом, вводять нові механіки та м'яко спрямовуючи гравця у потрібному напрямку, щоб він не загубився в просторі гри. Ці підказки, хоч і непомітні, є ключовими елементами розробки, які забезпечують складну та інтуїтивно зрозумілу подорож гравця.

І оскільки один рівень плавно переходить у наступний, переходи стають ключовими. Вони стосуються не просто переходу від одного етапу до іншого, а й забезпечення безперервності в подорожі гравця. Ці переходи, ретельно розроблені в процесі розробки гри, задають тон, темп і настрій, забезпечуючи згуртованість і переконливість ігрового досвіду.

Таким чином, розробка дизайну на рівні гри, хоча й є лише одним із етапів у ширшому циклі розробки гри, є окремим світом. Саме на цьому етапі гра набуває своєї остаточної форми завдяки досвіду і креативності дизайнерів рівнів. Завдяки їхньому баченню, відданості та розумінню процесу розробки,

створюються захоплюючі, викликаючі виклик і захоплюючі пригоди для гравців у всьому світі.

Етап 6. Імерсивний дизайн гри. Аудіо-синергія.

Занурення в захоплюючий ігровий дизайн підкреслює важливість аудіодизайну та озвучення. Це не просто звукові ефекти; це оркестрування симфонії, яка спрямовує, тривожить і захоплює гравців на кожному кроці. Під час розробки гри стає очевидним, що аудіо є потужним інструментом у процесі створення гри;

Етап 7. Тонкощі кодування.

Перейдемо до основного етапу розробки гри – кодування. З просуванням циклу розробки відбувається перехід від абстрактних концепцій до конкретних реалізацій. Сценарій підготовлено, і настав час вдихнути життя в гру.

Створення цифрового шедевра. На етапі кодування розробники отримують всі ігрові елементи. Вони починають інтегрувати ці компоненти, перетворюючи їх на складний цифровий шедевр. Хоча написання тисяч рядків коду може здатися виснажливим, цей процес нагадує створення цифрового полотна, яке повністю занурює гравців у світ гри.

Використання ігрових движків. Ігрові движки полегшують це завдання, але вимагають точності для забезпечення безперебійного функціонування гри. Процес можна порівняти зі складанням головоломки, де кожен елемент ідеально підходить до решти. Це сутність циклу розробки гри: кожен етап, кожне рішення та кожен рядок коду повинні гармонійно поєднуватися. Мета полягає в тому, щоб створити безпомилкову гру, яка не лише працює, але й захоплює.

Балансування. Протягом усіх етапів розробки розробники стикаються з численними викликами. Включення захоплюючих механік без суперечностей, забезпечення безперебійної роботи гри та оптимізація для досягнення найкращого користувацького досвіду стають пріоритетами. Це балансування є наріжним каменем процесу розробки гри [6].

3.4 Забезпечення якості

Забезпечення плавного та захоплюючого досвіду гравця має першочергове значення. В основі процесу розробки гри лежить ключовий крок: ретельне тестування. Це не просто про виявлення недоліків; це комплексна процедура, яка гарантує, що подорож гравця буде пальною, захоплюючою та безперервною. Навіть один збій або помилка можуть призвести до втрати інтересу гравця та зіпсувати враження від гри, заподіюючи шкоду витраченим годинам на розробку. Враховуючи серйозність цього етапу, не дивно, що цикл розробки гри інвестує так багато в вдосконалення цього процесу.

Етапи розробки гри схожі на розділи захоплюючого роману, кожна з яких має свою суть, кожна потребує свого унікального штриху. І, як і остаточний чернетковий огляд перед публікацією книги, ігри також повинні пройти сувору перевірку. Це забезпечує плавність оповіді, виклики захоплюють, а пригоди залишаються незабутніми.

Процес розробки відеоігор класифікує ці тести на функціональні та нефункціональні типи.

Типи функціонального тестування гри:

- тестування сумісності гри. Наскільки добре гра працює на різних платформах і системах;
- тестування регресійної гри. Чи порушують існуючі функції якісь нові функції чи налаштування;
- випробування на дим. Попередній тест, щоб виявити кричущі проблеми, перш ніж заглиблюватися глибше;
- тестування локалізації. Переконайтеся, що гра культурно та мовно резонує з різними демографічними групами;
- тестування гри контролю доступу безпеки. Утримання неавторизованих користувачів і потенційних хакерів;

– прийнятне тестування користувача. Лакмусовий папірець, щоб оцінити, чи відповідає гра очікуванням і вимогам цільової аудиторії.

Типи тестування нефункціональних ігор:

– тестування продуктивності. Гра працює гладко, забезпечуючи оптимальний ігровий процес;

– завантажити тестування гри. Як працює гра під навантаженням кількох користувачів або операцій;

– стрес-тестування. Доведіть гру до своїх можливостей, щоб побачити, в який момент вона ламається або виходить з ладу;

– тестування стабільності. Перевірка стійкості гри протягом тривалих періодів використання;

– об'ємне ігрове тестування. Як це працює, коли його бомбардують великою кількістю даних;

– зручність використання тестування. Гра інтуїтивно зрозуміла та зручна;

– тестування гри на відповідність. Переконайтеся, що гра відповідає галузевим стандартам і правилам;

– тестування установки. Чи встановлюється гра без проблем на різних платформах;

– тестування гри конфігурації. Перевірка продуктивності гри в різних конфігураціях;

– тестування аварійного відновлення. Перевірка здатності гри відновлюватися після збоїв або збоїв.

У циклі розробки гри кожен із цих етапів тестування гарантує, що коли гравець занурюється в гру, його зустрічає досвід, який не лише позбавлений помилок, але й глибоко захоплює. Етапи розробки ігор, особливо етап тестування, є свідченням прагнення індустрії надавати першокласні розваги.

Занурившись у величезний і заплутаний світ розробки ігор, стає очевидним, що тестування є невід'ємною складовою процесу розробки ігор.

Для збагачення розуміння суті створення гри, яка зацікавить гравців, необхідно досліджувати різноманітні типи ігрового тестування, кожний з яких має свої особливості та відмінності. Це дозволяє значно покращити цикл розробки гри, оптимізуючи геймплей від початку до кінця [6].

3.5 Випуск

Кожен етап у процесі розробки гри відіграє вирішальну роль, але деякі з них є такими захоплюючими, як етап запуску. Коли піднімається завіса над цим передостаннім етапом розробки гри, з'являється атмосфера очікування та хвилювання. Однак ті, хто добре розбирається в процесі розробки відеоігор, знають, що це не грандіозний фінал. Коли гра вперше зустрічається зі світом, у ній все ще можуть бути гострі краї.

Навіть найретельніше створені ігри можуть містити непередбачені проблеми. Поки гравці занурюються в цей новий всесвіт, команда розробників ігор невтомно працює у фоновому режимі, вдосконалюючи та покращуючи ігровий процес. Водночас починається хвиля зворотного зв'язку від справжніх гравців, надаючи розробникам скарбницю ідей для точного налаштування своїх шедеврів [6].

3.6 Пост-продакшн

Перехід до етапу пост-продакшну циклу розробки гри схожий на корабель, який вирушає на незвідані води. Тепер, коли гру запущено і гравці з усіх куточків пірнають, починається справжня робота. Етапи розробки гри не просто завершуються після випуску; вони розвиваються.

На етапі пост-продакшн розробки гри акцент зосереджується на стабільності та продуктивності. З дичиною в дикій природі, збільшувальне скло розміщене на кожному її куточку та щілині. Оскільки платформи постійно розвиваються та оновлюються, ігрові студії повинні бути

спритними, щоб їхні ігри залишалися сумісними та оптимізованими. Регулярні оновлення стають основою успішної гри, вирішуючи технічні проблеми та надаючи нові можливості, які збагачують ігровий досвід.

Крім того, у рамках постійного процесу розробки ігор розробники враховують постійно змінювані бажання своїх гравців. Вони не просто реактивні, а активні. Представляючи новий захоплюючий ігровий контент, механіку та функції, вони залучають ігрову спільноту, інвестують і завжди бадьорі, нетерпляче до того, що буде далі. Цей етап також підкреслює критичну роль забезпечення якості. З кожним оновленням і модифікацією ретельне тестування гарантує, що нові версії представляють захоплюючий вміст і зберігають цілісність і якість гри.

Отже, процес створення гри – це подорож, наповнена численними етапами, кожен з яких важливіший за попередній. Цикл розробки гри – це заплутаний танець створення, зворотного зв'язку, удосконалення та еволюції. Від початкової ідеї до безперервного розвитку після запуску, кожен етап розробки гри є свідченням відданості, пристрасності та досвіду команд, які створюють наші улюблені цифрові пригоди [6].

3.7 Висновки за розділом 3

У підсумку, процес розробки гри представляє собою поєднання артистизму, технологій і стратегічного мислення. Як споживачі, ми спостерігаємо кінцевий продукт, проте за цим досконалим результатом стоїть напружена подорож, в яку вкладені пристрасність, відданість та безліч годин важкої праці. Це свідчення сильного духу індустрії, яка продовжує розширювати горизонти та переосмислювати розваги.

Розробка ігор – це витончений баланс між аудіовізуальним елементом та програмним кодом. Процес розробки гри – це етап за етапом створення майстерної творчої роботи, що обіцяє неповторний і захоплюючий геймплей.

Кожен крок, звук і рядок коду – це складові частини магії розробки гри, неперервного циклу, який постійно вдосконалюється і розширюється.

Етапи розробки гри передбачають низку ретельних процесів, кожен з яких надбудовується на інший. На передньому краї цих етапів розробки гри знаходиться етап тестування, який ретельно вивчає кожен аспект гри, гарантуючи, що вона працює без збоїв. Для тих, хто прагне зрозуміти внутрішню роботу процесу розробки ігор, є численні ресурси, щоб дізнатися більше про основні типи тестування ігор, їх конкретні випадки використання та цінні поради щодо тестування ігор, розроблені для вдосконалення процесу.

Кожен тип тестування гри зосереджується на певному аспекті гри. Наприклад, у той час як одні зосереджені на оцінці функціональності гри, забезпеченні належної роботи всіх функцій і механізмів, інші можуть зосереджуватися на вимірюванні продуктивності гри на різних пристроях, забезпеченні сумісності та плавного ігрового процесу. Цей суворий процес допомагає виявити будь-які недоліки в процесі розробки гри [6].

4 ПРОЄКТУВАННЯ КОМП'ЮТЕРНОГО ДОДАТКУ

4.1 Визначення структури та функціоналу

Діаграма прецедентів (або діаграма варіантів використання, англ. Use Case diagram) описує функціональні потреби системи з точки зору прецедентів (рис. 4.1). Вона демонструє різні способи використання та типи користувачів системи. Способи використання зображуються колами або еліпсами, а актори (користувачі) часто представлені у вигляді паличок.

Дана діаграма описує дії, які може здійснювати гравець та ігрова система. Гравець може подавати різні запити до ігрової системи, а вона в свою чергу оброблює та виконує ці запити.

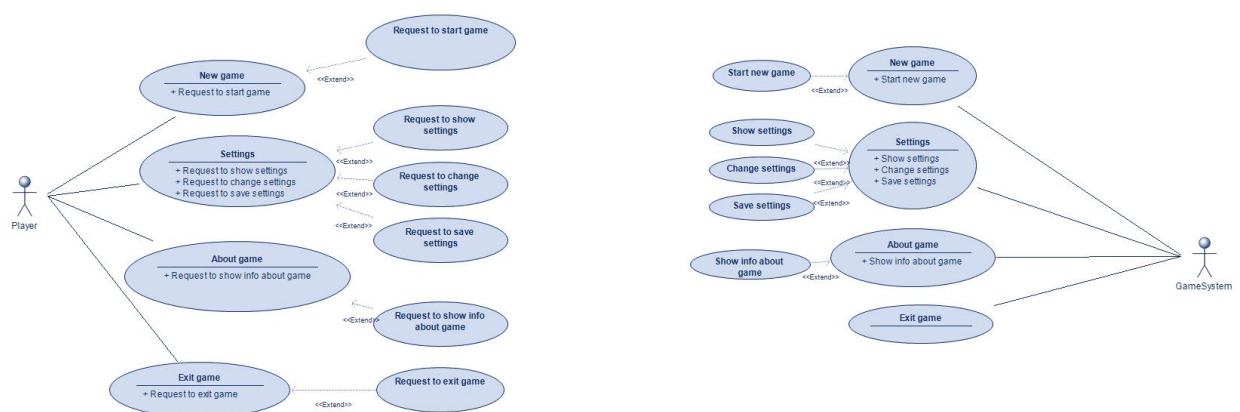


Рисунок 4.1 – Діаграма прецедентів (Use Case)

Діаграма діяльності (англ. Activity diagram) – це графічне зображення графу діяльностей (рис. 4.2). Граф діяльностей є типом графу станів скінченного автомату, де вершини представляють конкретні дії, а переходи відбуваються після завершення цих дій. На даному графі наочно видно структуру гри. Від запуску в головне меню і до виходу із самої гри.

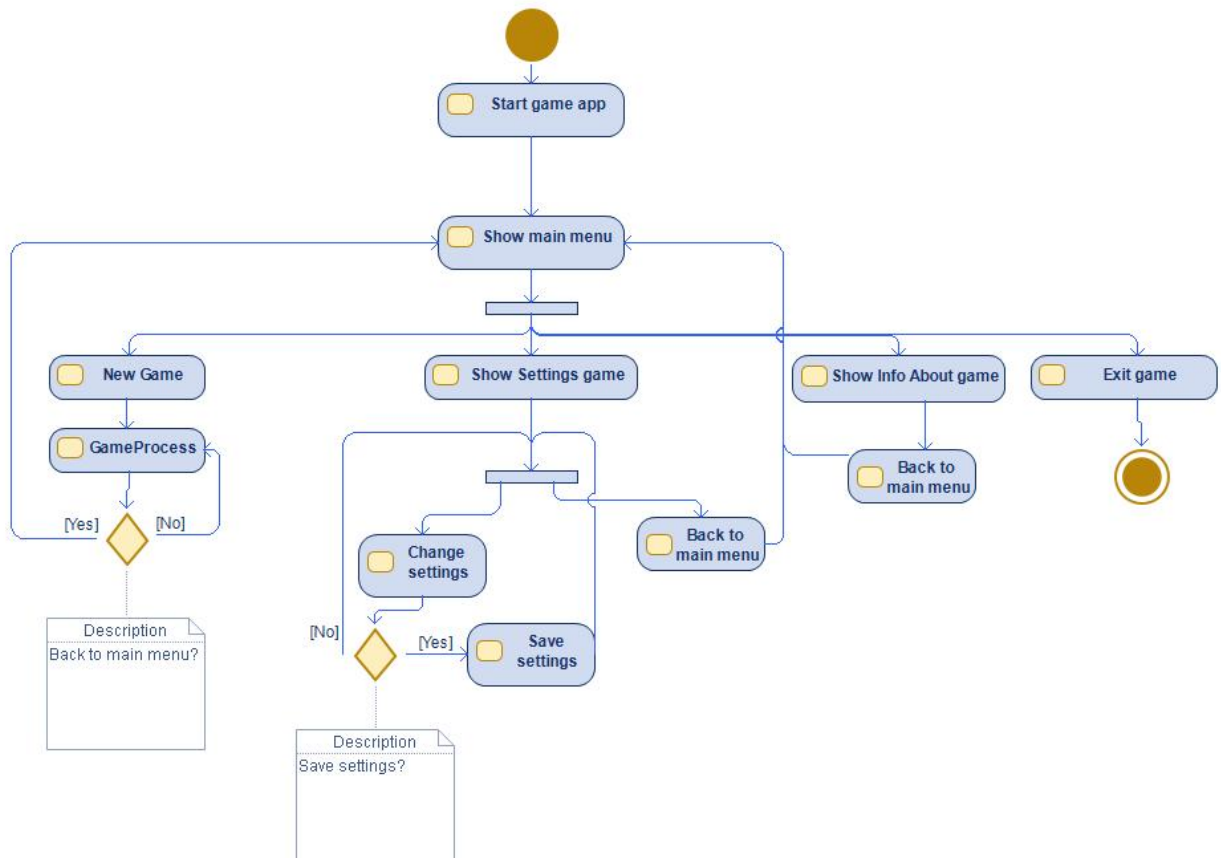


Рисунок 4.2 – Діаграма діяльності (Activity diagram)

Діаграма класів (англ. Class diagram) відображає статичні елементи, такі як класи, типи даних, їх зміст і взаємозв'язки. Вона використовується для представлення статичної структури системи з точки зору об'єктно-орієнтованого програмування. Нижче наведено кілька прикладів діаграм класів:

- діаграма класів для Dialogue Manager;
- діаграма класів для Game Manager;
- діаграма класів для Input Manager;
- діаграма класів для UI Manager.

Кожен з цих менеджерів має свій функціонал.

Dialogue Manager – керує ігровими діалогами . Запускає або зупиняє. Але також може запускати різні дії перед запуском або після зупинки діалогу (рис. 4.3).

Game Manager – керує процесом гри. Це головний менеджер, свого роду точка входу в гру (рис. 4.4).

Input Manager – керує вводом гравця. Тут ми налаштовуємо дії для клавіш, які потрібні нам у грі (рис. 4.5).

UI Manager – керує ігровими вікнами. Запуск, закривання або повернення на попереднє вікно яке було відкрите (рис. 4.6).

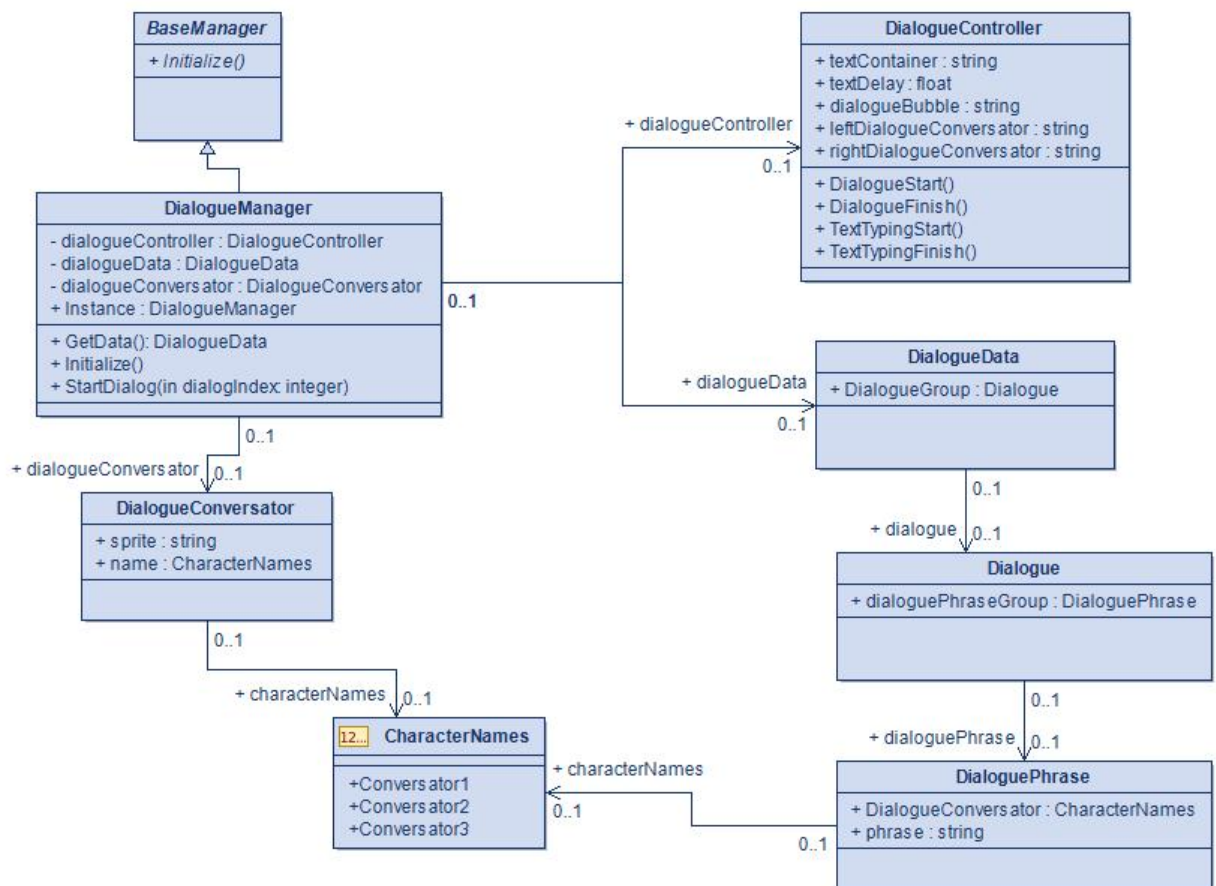


Рисунок 4.3 – Діаграма класів (Dialogue Manager)

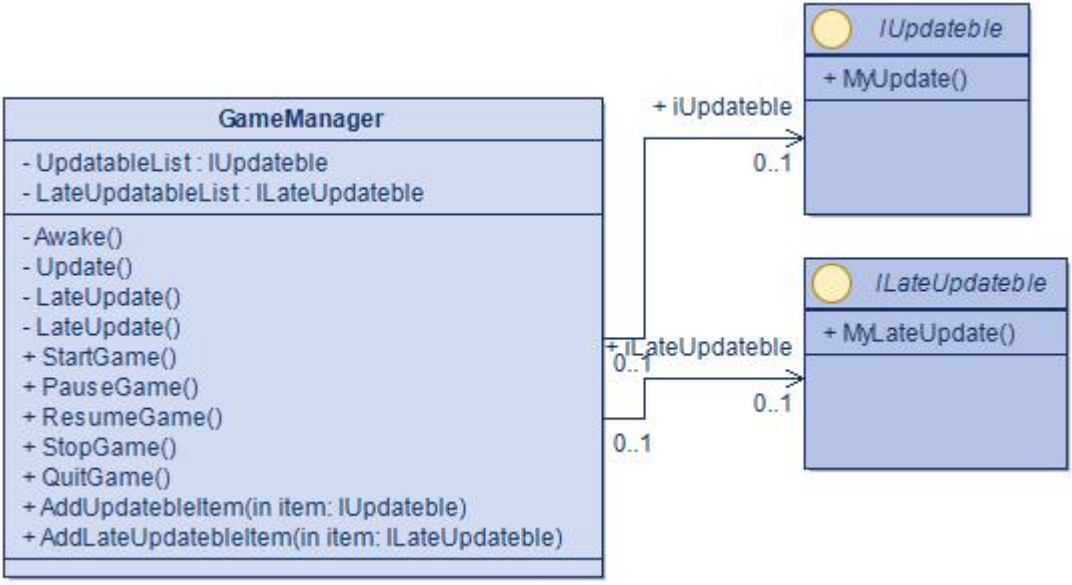


Рисунок 4.4 – Діаграма класів (Game Manager)

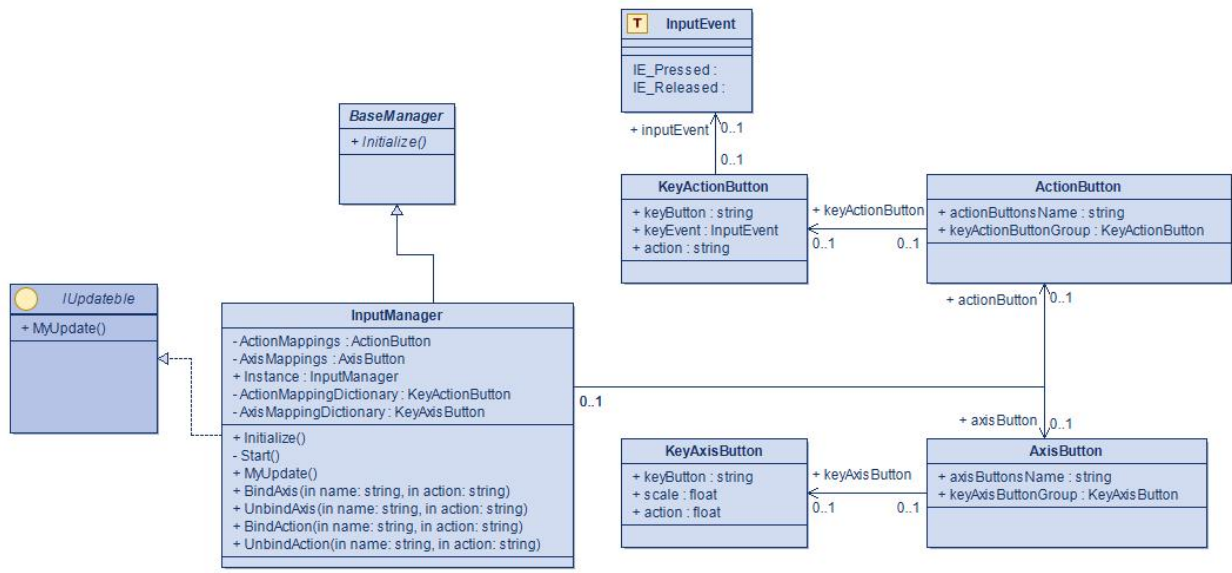


Рисунок 4.5 – Діаграма класів (Input Manager)

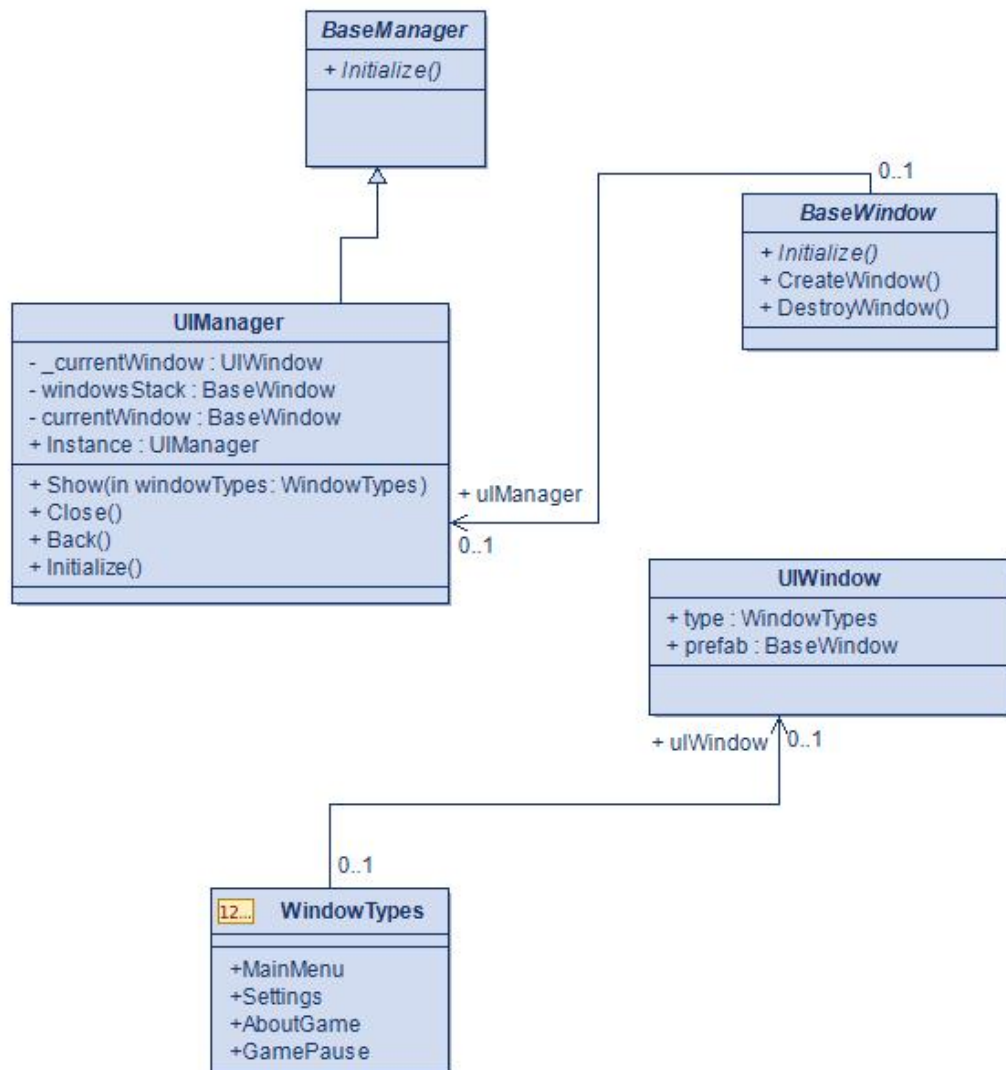


Рисунок 4.6 – Діаграма класів (UI Manager)

Діаграма послідовності (англ. sequence diagram) – відображає взаємодії об'єктів впорядкованих за часом (рис. 4.7).

Ця діаграма описує життєвий цикл гри. Коли користувач запускає гру, то перед собою він бачить головне меню, в якому він може змінити налаштування гри, або переглянути відомості про гру. Також з головного меню можна розпочати сам процес гри.

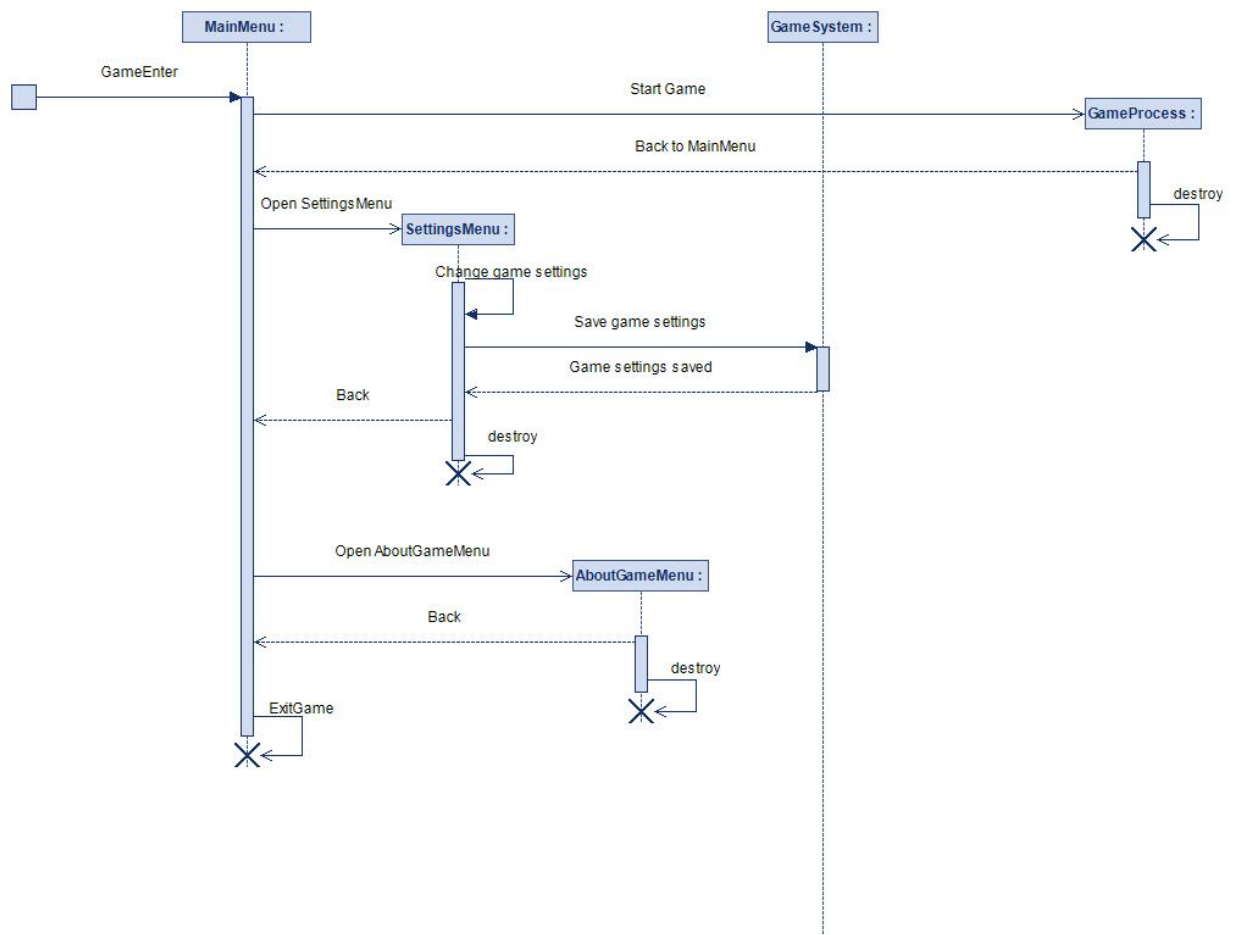


Рисунок 4.7 – Діаграма послідовності (Sequence diagram)

4.2 Визначення концепції

“Маленькі пригоди в Балабурсі” – це пригодницька гра з елементами рольової (RPG).

Основні концептуальні особливості гри:

а) контекст і сюжетна лінія:

1) світ гри. Дія відбувається у фантазійному світі, на просторах лісу, місцевості під назвою “Смарагдовий ліс”;

2) атмосфера. “Маленькі пригоди в Балабурсі” створює затишну, спокійну атмосферу, досліджує теми відповідальності та наслідків вибору. Гра ставить гравця перед моральними дилемами, що змушує задуматися про наслідки своїх дій;

3) сюжет. Гравець починає свою пригоду після викрадення несутвітними істотами і зараженням паразитом, що загрожує перетворенням на одного з цих монстрів. Гравець повинен знайти спосіб позбутися паразита, подорожуючи Смарагдовому гаю він стикається з численними небезпеками;

б) ігровий процес:

1) покрокові бої. Гра “Маленькі пригоди в Балабурсі” використовує покрокову бойову систему, що базується на правилах D&D 5-ї редакції. Це додає тактичної глибини боєзйткненням;

2) діалоги і рішення. Гра пропонує розгалужену систему діалогів, де вибір гравця впливає на розвиток сюжету, взаємовідносини з іншими персонажами та навіть на світ гри (рис. 4.8);



Рисунок 4.8 – Система діалогів з ігровими персонажами

в) візуальна і технологічна складова:

1) графіка. Гра виконана стилі з використанням піксельної графіки, що надає їй унікального шарму і викликає ностальгію за класичними іграми;

2) інтерактивність світу. Гра пропонує високий рівень інтерактивності, дозволяючи гравцям взаємодіяти з оточенням різними способами, що впливає на розвиток подій.

Вплив Dungeons & Dragons.

Атмосфера та сюжетна база гри. “Маленькі пригоди в Балабурсі” зберігає дух і атмосферу класичних D&D, надаючи гравцям можливість зануритися в епічну фентезійну пригоду.

4.3 Проєктування гри “Маленькі пригоди в Балабурсі”

Проєктування гри – це складний і багатогранний процес, який включає концептуальну розробку, дизайн, програмування, тестування та багато інших аспектів. Мета цього підрозділу – надати огляд основних етапів і методів проєктування гри, а також обговорити ключові елементи.

Концептуальний етап.

Ідея гри. Основна концепція гри, яка визначає її жанр, механіки та загальний напрямок.

Цільова аудиторія. Визначення аудиторії, на яку спрямована гра, її вікових груп, інтересів та очікувань.

Сеттинг і сюжет.

Сеттинг. Визначення світу гри, його атмосфери, естетики та основних елементів.

Сюжет. Розробка основної історії, її ключових подій, персонажів та конфліктів.

Кожна гра починається з головного меню (рис. 4.9), яке одразу дає зрозуміти нам концепт гри, і цей проєкт не став виключенням.



Рисунок 4.9 – Ескіз головного меню гри

На початку процесу створення сюжету було звернуто увагу на популярну фантазійну літературу, ігри та фільми. Були знайдені цікаві ідеї та концепції які могли б стати основою захоплюючого сюжету для гри. На основі цього було створено уявний світ, в якому відбуватимуться події гри. Світ “Маленькі пригоди в Балабурсі” складається з міста Балабурс і його навколишніх територій, таких як ліси, печери, та інші. Цей світ живий та різноманітний, наповнений різноманітними персонажами та квестами. Була розроблена карта ігрового світу (рис. 4.10), визначила основні фракції, місця, історія та культуру.

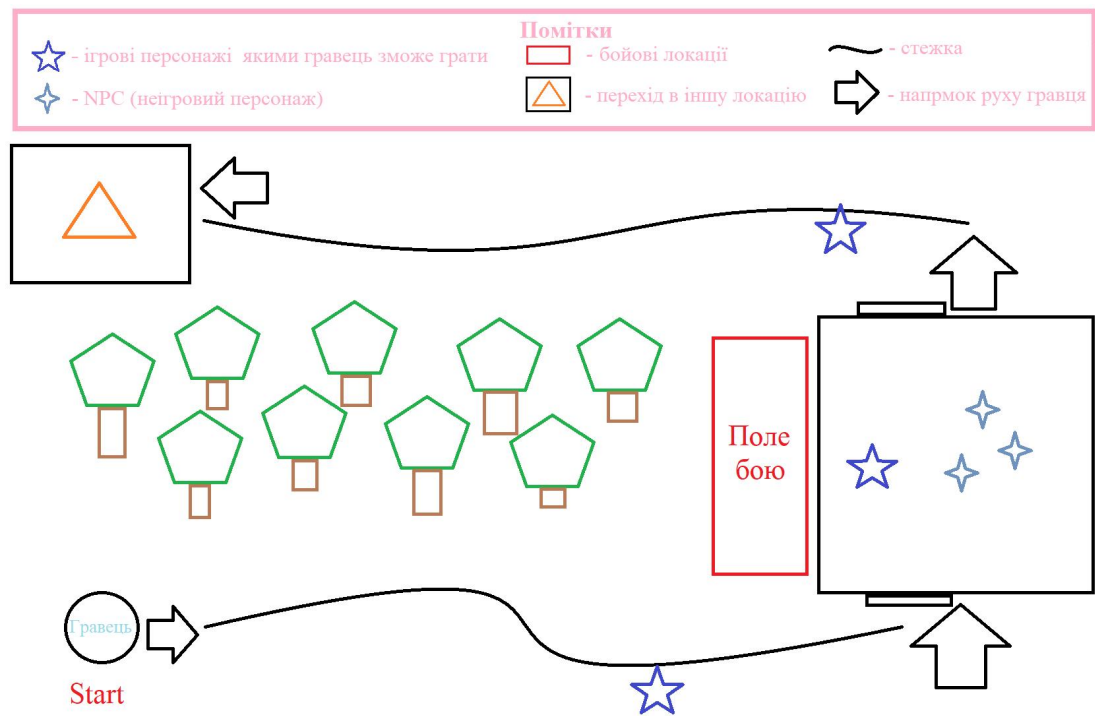


Рисунок 4.10 – Ескіз ігрової карти першого рівня

Створення дизайну персонажа – це індивідуальний процес, тому важливо дозволити своєму творчому потенціалу проявитися та експериментувати, щоб створити персонажа. Перш ніж щось створити, ми розробляємо концепцію: визначаємо, що це буде, яким воно буде і для чого це потрібно. Те саме стосується і персонажа: важливо зрозуміти, ким він є, які в нього особистісні риси та що його мотивує.

При створенні дизайнів персонажів (рис. 4.11) було ураховано: вік, стать, раса (так як це фентезійний світ), стиль одягу, характер персонажа, концепція гри, його роль, характеристики, його особливості, такі як історія та мотивація. Дуже важливо показати, що зовнішній вигляд персонажа відображає його особистість. Також, зброя кожного персонажу відповідає його ігровому класу.



Рисунок 4.11 – Ескізи персонажів

Також дуже важливим був етап створення сюжету. Ідея була створена на основі ігор у жанрі D&D (Dungeons and Dragons), де зазвичай у персонажів можуть бути різні раси і класи.

З самого початку було введено конфлікт і напруження в сюжет, щоб зацікавити гравців і підтримати їх інтерес. Тому сюжет починається з передісторії. Далі було розроблено приблизний сценарій (рис. 4.12) з прописаними діалогами і різними замітками, як у текстовому вигляді, так і в графічному.

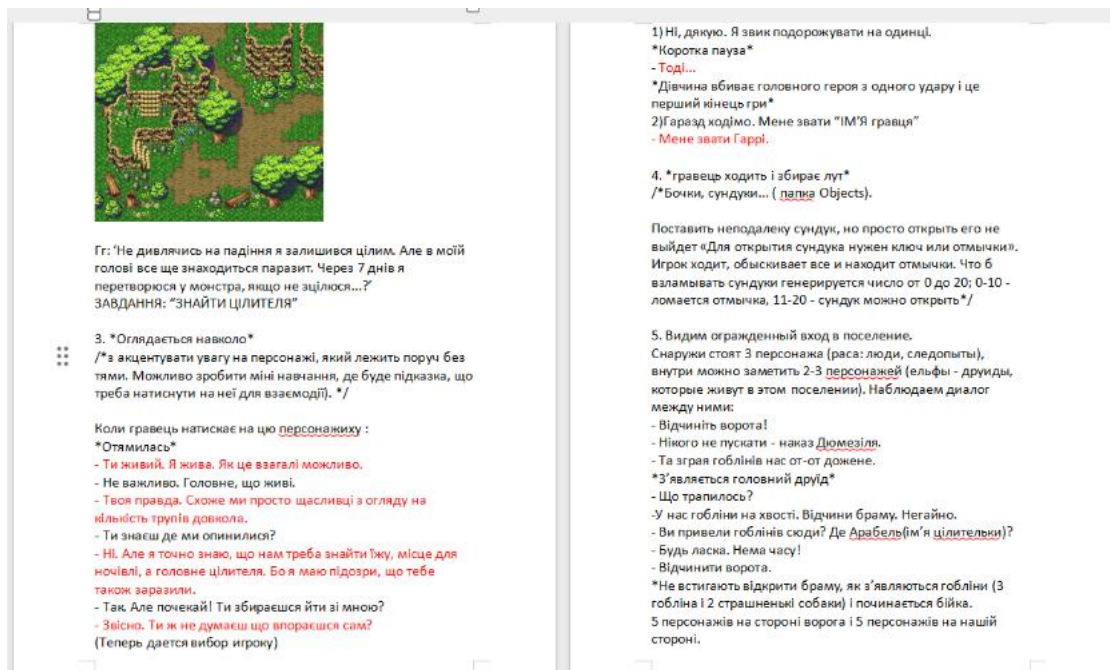


Рисунок 4.12 – Сценарна частина

4.4 Проєктування першого ігрового рівня

Проєктування рівня для гри “Маленькі пригоди в Балабурсі” було складним процесом, який включає в себе розробку структури, дизайну, механік та інтерактивності, що забезпечує захоплюючий та незабутній ігровий досвід. Ось основні етапи та компоненти проєктування першого ігрового рівня:

а) концепція рівня:

1) загальний задум:

- обстановка. Перше, що гравець побачить у грі, це місце, де він опиниться після викрадення монстрами. Місце дії – ліс в якому знаходиться поселення;
- атмосфера. Атмосфера повинна бути напруженою і загадковою, передаючи відчуття небезпеки та терміновості;

2) сюжетна лінія:

- основна сюжетна арка. На першому рівні гравець виявляє більше про походження паразита, що впливає на

головного героя, і зустрічає нових ключових персонажів, які можуть стати союзниками;

– завдання. Основне завдання – знайти цілителя, який може допомогти знищити паразита. Гравець дізнається про його місцезнаходження через нову інформацію, отриману від NPC;

3) цілі:

– основна ціль. Врятуватися від впливу паразиту, що б не стати монстром;

– побічні цілі. Ознайомлення з базовими механіками гри, знайомство з ключовими персонажами;

б) ігрові механіки:

1) навчання. Контрольовані зустрічі простих ворогів для демонстрації бойової системи;

2) взаємодія. Діалоги. Перші діалоги з іншими персонажами, що знайомлять гравця з системою діалогів;

в) елементи дизайну:

1) візуальний стиль. Ліс повній зелені, навіть персонажі, що живуть там, уособлюють дух природи;

2) зони та шляхи. Рівень складається з кількох зон, кожна з яких має свої унікальні особливості та виклики;

3) звуковий дизайн. Музика властива іграм у жанрі D&D в тематиці лісу або бою відповідно до ситуації в грі;

г) сюжетні елементи:

1) текстовий вступ, в якому гравець дізнається передісторію про викрадення і зараження персонажа паразитом, що дає контекст і мотивує гравця до дій;

2) зустріч з персонажами:

– ключові NPC. Зустріч з персонажами, які допоможуть гравцеві;

– діалоги. В гравця є можливість взяти персонажа в команду, або ж ні;

г) бойові зіткнення. Вороги – низькорівневі вороги, які служать для ознайомлення з бойовою системою;

д) підсумок рівня. Завершення:

1) кульмінація. Рятування цілительки і поселення, зцілення головного персонажу і персонажів команди;

2) завершальна сцена. Цілителька в знак подяки, за врятування її і її поселення зцілює ігрових персонажів;

е) технічний етап. Програмування:

1) двигун гри. Вибір і налаштування ігрового двигуна, такого як Unity;

2) розробка. Написання коду, що реалізує ігрові механіки, фізику та інші системи гри;

є) проведення тестування.

Існує два типи тестування:

– альфа-тестування. Перевірка основних механік і взаємодій на рівні, виявлення багів і технічних проблем;

– бета-тестування. Ширше тестування з залученням більшої кількості тестувальників для отримання зворотного зв'язку і виявлення менш очевидних проблем.

Для даного проєкту було проведено альфа-тестування, де виявлялися помилки, було здійснено балансування геймплею та виправлено будь-які недоліки.

4.5 Висновки за розділом 4

Проектування гри – це складний процес, що вимагає творчого підходу, технічних знань і ретельного планування. Від початкової ідеї до кінцевого релізу гра проходить через численні етапи розробки, кожен з яких є критично

важливим для досягнення кінцевого успіху. Розуміння цього процесу та ефективна його реалізація дозволяють створювати ігри, які захоплюють і приносять задоволення гравцям.

Проектування першого рівня гри "Baldur's Gate 3" є критично важливим етапом, який задає тон всій грі, знайомить гравця з основними механіками та залучає його в сюжет. Успішне поєднання навчальних елементів, інтригуючого сюжету та захоплюючого геймплею допоможе створити незабутній досвід для гравців з самого початку.

Використання діаграм у проєкті Unity дозволяє наочно представити різні аспекти системи, що значно полегшує розробку, управління та документування. Таким чином, вони є важливим інструментом для моделювання, аналізу та комунікації в процесі розробки гри, забезпечуючи ясність та структурованість у представленні системи та її компонентів.

Тестування. Після створення базового сюжету та геймплею, гра проходила етап тестування, де виявлялися помилки, було здійснено балансування геймплею та виправлено будь-які недоліки.

5 РОЗРОБКА КОМП'ЮТЕРНОГО ДОДАТКУ “МАЛЕНЬКІ ПРИГОДИ В БАЛАБУРСІ”

5.1 Розробка сюжету гри

На початку процесу створення сюжету було звернуто увагу до фантазійної літератури, ігор та фільмів. Були в пошуку ідеї та концепції, які були б цікавими і могли б стати основою захоплюючого сюжету для гри.

На наступному етапі було створено уявний світ, в якому відбуватимуться події гри. Світ “Маленькі пригоди в Балабурсі” складається з поселення Балабурс і його навколишніх територій, таких як ліси, печери, та інше. Цей світ живий та різноманітний, наповнений різноманітними персонажами та квестами. Була розроблена карта світу, визначила основні фракції, місця, історія та культуру.

При запуску гри на фоні ми бачимо 2D ліс, у центрі вгорі назва гри “Маленькі пригоди в Балабурсі” і зліва знизу 3 кнопки (рис. 5.1).



Рисунок 5.1 – Головний екран гри

На початку гри нас зустрічає передмова (рис. 5.2) “В народі говорять, що бачили летючий інопланетний корабель, яким керують потвори з великими головами й довгими мацаками, що ростуть з їх обличчя. Раз у 2 роки вони прилітають на землю і викрадають людей, для того що б перетворити їх у таких же почвар і забирають до себе. Я читав багато книжок про це. І ось цього дня мені не пощастило потрапити до їх мерзенних лап. Я отямився у темному приміщені, яке здавалося живим, ніби зроблено із плоті. Переді мною стояла та сама потвора. Вона взяла у свої кігті створіння схоже на черв’яка і притулила його до мого ока. Ще секунда, і воно залізло до мене у мозок. Я відчув це. Різкий ріжучий біль промайнула у самісіньку глиб моєї свідомості. Не пам’ятаю точно скільки хвилин, годин або навіть днів провів там. Почувши шум я відкрив очі та відчув ніби все навколо тремтить. Крики й звуки боротьби лунали в дивного вигляду двері. Після цього я пам’ятаю, як відчувся поштовх, здається корабель у щось вривався. Хвилина у невагомості і темрява перед очима...” яка вводить гравця у суть історії і дає зрозуміти йому ціль і завдання.



Рисунок 5.2 – Екран з передмовою

Після передмови гравець має можливість написати ім'я для персонажа за якого він грає, або обрати варіант за замовченням (рис. 5.3). Це може бути справжнє ім'я, або вигадане.



Рисунок 5.3 – Екран вибору ім'я ігровому персонажу

Наступний найважливіший аспект будь-якої гри - це персонажі. Було створено широкий спектр унікальних персонажів, які зустрічаються гравцю протягом гри. Ці персонажі мали власні історії, мотивації та характери, які розкривалися протягом гри і кожен з них мав свою манеру спілкування. Всього в грі 3 персонажа, яких можна запросити у свою команду, тобто гравець може керувати ними.

Перший персонаж: “Гаррі”. Він є першим з потенційних союзників гравця в грі. Гаррі є вікінгом, який є членом вікінгського клану. Його як і нашого головного персонажу викрали монстри, вживили в його мізки паразита і йому вдалось вижити після корабельної аварії. Цей персонаж відважний і рішучий, а також вправно користується своєю сокирою.

Другий персонаж: “Артуро”. Житель Балабурсу. Один з охоронців, що стоять на вході у поселення. Дізнавшись, що гравець йде врятувати їх цілительку і відповідно захистити селище від ворогів, пропонує приєднатися. У демо версії гри характер персонажу розкрито не достатньо добре, але точно можна сказати, що він добрий, доброзичливий і хвилюється за людей зі свого поселення.

Третій персонаж: “Гусаріон”. Малий гусак, чий характер може бути описаний як хитрий, зухвалий. Його особистість може містити елементи таких рис, як самовпевненість, складність та відчуття власної важливості. Його моральність може бути неоднозначною, і він може виступати як антигерой або персонаж змінливої вірності.

Ці персонажі стали ключовими фігурами в розвитку сюжету та взаємодії з гравцем. Вони мають ту ж саму проблему, що і головний герой і допомагають гравцю з її вирішенням.

Збираючи всі ідеї та концепції, був розроблений основний сюжет гри: епічна пригода, яка складалася зі сюжетних ліній, побічних квестів (рис. 5.4) та важливих рішень (рис 5.5), які впливали на хід історії. Була задача створити сюжет, який би був цікавим, захоплювальним та здатним зацікавити гравців.

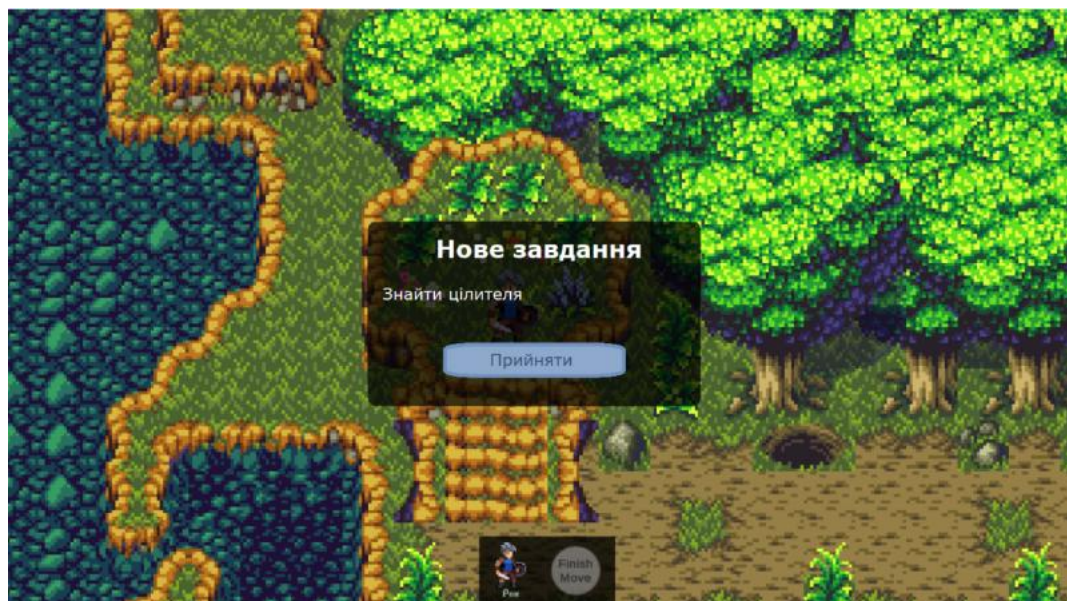


Рисунок 5.4 – Приклад ігрового квесту



Рисунок 5.5 – Приклад рішення в грі, яке впливає на розвиток сюжету

Взявши завдання і союзників гравець повинен вирішити, чи буде він рятувати тільки себе чи інших також. Адже, попереду пригоди і небезпеки.

5.2 Розробка дизайну ігрових локацій

У цьому розділі розглядається процес розробки дизайну ігрової локації, що є важливою складовою успішної гри.

Дизайн ігрової локації (рис. 5.6) включає в себе створення середовища, у якому гравець буде взаємодіяти з ігровим світом, та визначення різноманітних елементів, таких як об'єкти, території, ландшафт і архітектура.

Етапи:

- визначення концепції ігрових локацій. Як властиво для більшості подібних ігор було обрано тему “ліс”, так як кожен звір, чи чужинець може бути потенційним ворогом для гравця;
- проектування локацій. Для цього етапу було використано страйти тематики “ліс” і “селище”;

- створення 2D моделей і об'єктів. Для розробки спрайтів локації використовувалася програма PoxelArt;
- розміщення об'єктів у локації було зроблено, безпосередньо, завдяки кросплатформовому середовищу розробки комп'ютерних ігор Unity.



Рисунок 5.6 – Дизайн ігрової локації

5.3 Розробка ігрових персонажів та їх спрайтів

У цьому розділі розглянуто процес розробки ігрових спрайтів, що є важливою складовою будь-якої гри. Ігрові спрайти відображають об'єкти та персонажів гри, що робить їх ключовим елементом візуального досвіду користувача.

Спрайти повинні відповідати стилістиці та атмосфері гри. У даній грі кожен персонаж має бути не тільки сам по собі особливим, а ще й мати свою індивідуальну зброю.

Спрайти були розроблені у програмі PixelArt (рис. 5.7).



Рисунок 5.7 – Робота в програмі PixelArt

Перший спрайт який бачить гравець – це головний герой, за якого він буде грати. Тому головний персонаж зображений лицарем, що б у гравця були асоціації з кимось сміливим, рішучим і завжди готовим прийти на допомогу. Як і кожного лицаря у головного героя є великий меч и броня (рис. 5.8).



Рисунок 5.8 – Спрайт головного персонажу яким керує гравець

В грі також є персонажі, які будуть допомагати гравцю. Кожен з цих персонажів має свій індивідуальний характер і зовнішній вигляд.

Персонаж “Артуро” – корінний житель поселення Балабурсу, тому йому притаманний невеликий зріст і зброя у вигляді луку (рис. 5.9).



Рисунок 5.9 – Спрайт одного з ігрових персонажів

“Гусаріон” являється протирічним персонажем. Тому його милий зовнішній вигляд був спеціально зроблений повною протилежністю його негідному характеру (рис. 5.10).

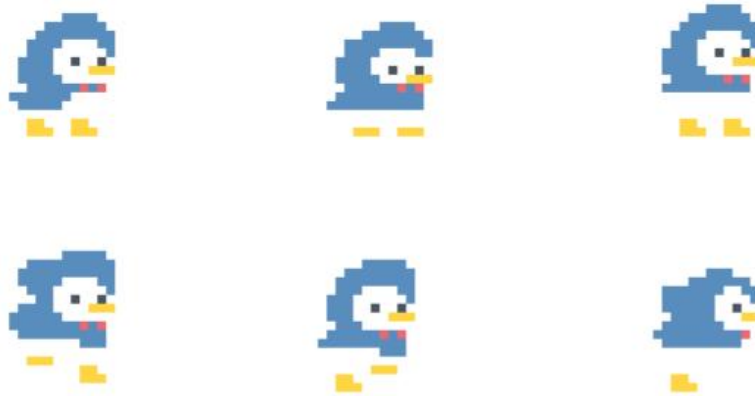


Рисунок 5.10 – Спрайт одного з ігрових персонажів

Для створення злих персонажів був обран образ жаби і гобліна. В грі вони називаються “жабліни” (рис 5.11).



Рисунок 5.11 – Спрайт одного з злодійських персонажів

5.4 Програмна реалізація комп’ютерного додатку

В даному підрозділі описуються код всіх менеджерів присутніх у проєкті Unity, від даних та окремих рядків до функцій та методів.

AnimatorManager – менеджер анімацій, який дозволяє перемикати аніматори за іменем:

а) функціональність:

1) список аніматорів – доступ до аніматорів здійснюється за іменем, яке береться з переліку enum;

2) створення структури об'єктів для катсцени – забезпечення організації об'єктів, необхідних для катсцени;

3) створення анімації – реалізація нової анімації, включаючи вимкнення аніматора, додавання об'єкта до списку та призначення йому імені;

б) основні методи:

1) Play (enumAnimatorName) – запуск анімації за вказаним ім'ям з переліку enum;

2) Pause (enumAnimatorName) – пауза анімації за вказаним ім'ям з переліку enum;

3) Stop(enumAnimatorName) – зупинка анімації за вказаним ім'ям з переліку enum;

в) події. Підписка на ці події здійснюється перед запуском анімації, а відписка – після її завершення:

1) OnAnimationStarted – викликається при початку анімації;

2) OnAnimationFinished – викликається при завершенні анімації;

г) скрипт для анімацій. AnimationBehaviour – скрипт, який слід прикріпити до анімації для забезпечення її функціональності. У ньому визначено методи, які спрацьовують, коли анімація почалася, закінчилася або працює.

AudioManager – менеджер, що відповідає за створення, видалення, програвання, зупинення та відновлення аудіо:

а) процес створення аудіо:

1) створити префаб аудіо джерела (AudioSource);

- 2) записати цей аудіо джерело у словник аудіо джерел;
- 3) призначити до нього аудіо кліп (AudioClip);

б) основні методи:

1) Create (AudioIndex, isLoop, Vector3Pos): створює інстанцію префабу звуку або музики за індексом у списку, розташовує її у заданій точці на сцені та додає в словник 'Dictionary<IDSound, Sound>'. Це дозволяє в подальшому програвати, ставити на паузу, зупиняти або видаляти аудіо за 'IDSound';

2) CreateAndPlay (AudioIndex, isLoop, Vector3Pos): створює інстанцію префабу звуку або музики за індексом у списку, розташовує її у заданій точці на сцені, додає в словник 'Dictionary<IDSound, Sound>' і одразу починає програвання;

3) PlayOneShot (AudioIndex, Vector3Pos): створює інстанцію префабу звуку або музики за індексом у списку, розташовує її у заданій точці на сцені та одразу видаляє після програвання;

4) Play (IDAudio, IDAudioSource, isLoop): призначає 'AudioClip' зі словника за 'IDAudio' певному аудіо джерелу, знайденому за 'IDAudioSource' у словнику аудіо джерел, і починає його програвання;

5) Play (IDAudio, AudioSource, isLoop): призначає 'AudioClip' зі словника за 'IDAudio' певному аудіо джерелу на сцені та починає його програвання;

6) Pause (IDAudio): ставить на паузу 'AudioClip' через аудіо джерело, знайдене за 'IDAudio' у словнику аудіо джерел;

7) Pause (AudioSource): ставить на паузу 'AudioClip' через певний аудіо джерело на сцені;

8) Resume (IDAudio): продовжує програвання 'AudioClip' через аудіо джерело, знайдене за 'IDAudio' у словнику аудіо джерел;

9) Resume (AudioSource): продовжує програвання 'AudioClip' через певний аудіо джерело на сцені;

10) Stop (IDAudio): зупиняє 'AudioClip' за 'IDAudio' зі словника;

11) Stop (AudioSource): зупиняє 'AudioClip' через певний аудіо джерело на сцені;

12) Delete (IDAudio): видаляє префаб аудіо джерела за 'IDAudio' зі словника та зі сцени;

13) Delete (AudioSource): видаляє об'єкт з компонентом 'AudioSource' зі сцени.

DataManager – менеджер керування даними (збереження, завантаження, видалення):

- public static void SaveData<T>(T dataObject, DataType dataType) – збереження даних типу T у реєстр на ім'я dataType;

- public static T LoadData<T>(DataType dataType) - завантаження збережених даних типу T з реєстру на ім'я dataType;

- public static void ResetData(DataType dataType) – видалення збережених даних типу dataType;

- public static void ResetAllData() – видалення всіх збережених даних.

DataType (знаходиться у файлі DataCore) – перелік типів даних, які зберігаються в реєстр. Наприклад, створено скрипт Settings, який зберігає дані налаштувань. Для збереження або завантаження даних з реєстру використовується ім'я, що відповідає певному типу даних. Це ім'я зберігається у перерахунку DataType.

DialogueManager – менеджер діалогів. Запускає певний діалог, який зберігається у DialogueData:

а) DialogueData – клас, який зберігає дані діалогу:

- 1) репліки;

- 2) загальна кількість персонажів діалогу (імена та спрайти);

б) CharacterNames – перерахування, яке зберігає імена персонажів діалогів;

в) DialogueController – контролер, який управляє діалогом;

- г) `FTextContainer` – контейнер, де друкується текст діалогу;
- г) `FTextDelay` – затримка друку літер діалогу;
- д) `FDialogueTapPlace` – область натискання щоб пропустити/прискорити друкування тексту діалогу та перейти до наступної репліки;
- е) `FLeftDialogueConversator` – лівий персонаж діалога;
- е) `FRightDialogueConversator` – правий персонаж діалогу;
- ж) `FDialogueBubble` – область діалогу за якою визначається чи змінюється розмір шрифту надрукованого тексту. Якщо зменшується, значить текст не поміщається, а значить, потрібно припинити друкувати, а при натисканні на `FDialogueTapPlace` продовжуємо друкувати з того місця, з якого зупинилися;
- з) `OnWriteTextStarted` – подія, яка викликається коли текст діалогу почав друкуватись;
- и) `OnWriteTextFinished` – подія, яка викликається коли текст діалогу перестав друкуватися;
- і) `OnDialogueStarted` – подія, яка викликається коли діалог розпочався;
- ї) `OnDialogueFinished` – подія, що викликається коли діалог закінчився.

`GameManager` – точка входу в програму. `DontDestroyOnLoad` – не видаляється в процесі всієї гри.

Цей об'єкт зберігає список всіх елементів, які мають виконуватися кожен кадр. Отже, для того, щоб скрипт виконувався кожен кадр, він повинен реалізовувати інтерфейси `IUpdatable` або `ILateUpdatable`. У методі `Awake` цього скрипта необхідно додати його до `GameManager` за допомогою методів `GameManager.AddUpdatableItem(IUpdatable item)` або `GameManager.AddLateUpdatableItem(ILateUpdatable item)`.

Події і методи:

- `OnGamePaused` – подія, яка виконується якщо гра на паузу встала;
- `OnGameResumed` – подія, яка виконується якщо гра знята з паузи;
- `StartGame()` – статичний метод, який викликається при старті гри;

- `StopGame()` – статичний метод, який викликається при зупинці гри;
- `QuitGame()` – статичний метод, який викликається при виході з гри.

`InputManager` – спрощений `InputManager` Unity.

`ActionMappings` – список кнопок, які виконують підписану на їх `action` якусь дію при натисканні/відпусканні цієї кнопки:

а) `oKeyActionButton`:

- 1) `keyEvent` – подія кнопки (натискання/відпускання);
- 2) `keyButton` – кнопка на яку поширюється подія;
- 3) `action` – дія (потрібно підписувати методи), яка виконується

після взаємодії з кнопкою;

б) `oActionButton`:

- 1) `actionButtonsName` – (перерахування (enum)) назва списку кнопок-дій (наприклад “Interact”);
- 2) `keyActionButtonGroup` – список кнопок (`KeyActionButton`), які потрібно підписувати на події.

`AxisMappings` – список кнопок, які виконують підписану на їх `action` якусь дію при натисканні та утриманні цієї кнопки.

Змінні тут виконують ту саму функцію, що й кнопки у `ActionMappings`, але додатково додана змінна `Scale`. Ця змінна приймає значення, яке буде передано параметру методу, який підписується на `Action<float>`. Наприклад, якщо ми створимо список кнопок з назвою `MoveForward` і включимо до нього кнопки, які треба утримувати для руху вперед та назад (W, S), і при цьому привласнимо значення 1 і -1 змінній `Scale` відповідно, то метод, на який буде підписано `action` для цих кнопок, буде мати назву `MoveForward(float scaleAmount)`, де параметр `scaleAmount` буде приймати значення 1, якщо утримується кнопка W, або -1, якщо утримується кнопка S.

Методи, класи і подія:

- BindAction(ActionButtonName name, Action action) – метод, який підписує дію action, на кнопки під назвою name;
 - UnbindAction(ActionButtonName name, Action action) – метод, який відписує дію action, від кнопок під назвою name;
 - TaskManager – головний клас, який зберігає список тасок, спавн поінтів для цих тасок, TaskPopup (впливаюче вікно Прийняти-Отклонить таску);
 - TaskBase – клас-батько для створення нової таски. Щоб створити тягу, потрібно успадковуватись від цього скрипту;
 - OnDialogueFinished – подія, що викликається коли діалог закінчився.
- GameManager – точка входу в програму. DontDestroyOnLoad – не видаляється в процесі всієї гри.

Управління Кадрами. Цей механізм зберігає список усіх об'єктів, які потрібно виконувати кожен кадр. Тому будь-який скрипт, що має бути виконаний кожен кадр, повинен реалізувати інтерфейс IUpdatable або ILateUpdatable. В методі Awake цього скрипта слід додати його до GameManager за допомогою методу GameManager.AddUpdatableItem(IUpdatable item) або GameManager.AddLateUpdatableItem(ILateUpdatable item).

Події і методи:

- OnGamePaused – подія, яка виконується якщо гра на паузу встала;
- OnGameResumed – подія, яка виконується якщо гра знята з паузи;
- StartGame() – статичний метод, який викликається при старті гри;
- StopGame() – статичний метод, який викликається при зупинці гри;
- QuitGame() – статичний метод, який викликається при виході з гри.

InputManager – спрощений InputManager Unreal Engine.

ActionMappings – список кнопок, які виконують підписану на їх action якусь дію при натисканні/відпусканні цієї кнопки:

a) oKeyActionButton:

- 1) keyEvent – подія кнопки (натискання/відпускання);

- 2) `keyButton` – кнопка на яку поширюється подія;
- 3) `action` – дія (потрібно підписувати методи), яка виконується після взаємодії з кнопкою;

б) `oActionButton`:

- 1) `actionButtonsName` – (перерахування (`enum`)) назва списку кнопок-дій (наприклад “Interact”);
- 2) `keyActionButtonGroup` – список кнопок (`KeyActionButton`), які потрібно підписувати на події.

`AxisMappings` – список кнопок, які виконують підписану на їх `action` якусь дію при натисканні та утриманні цієї кнопки.

Для змінних застосовується аналогічний підхід, як і для кнопок у `ActionMappings`, з додаванням змінної `Scale`. Ця змінна приймає значення, яке буде передано як параметр методу, підписаного на `Action<float>`.

Наприклад, створено список кнопок під назвою `MoveForward`. У цьому списку визначені кнопки, які потрібно утримувати для руху вперед і назад: `W` і `S`. Значенню `scale` присвоєно 1 і -1 відповідно. Метод, який буде підписано на `action` цих кнопок, називається `MoveForward(float scaleAmount)`. Параметр `scaleAmount` буде приймати значення 1 при утримуванні кнопки `W` або -1 при утримуванні кнопки `S`.

Методи:

- `BindAction(ActionButtonName name, Action action)` – метод, який підписує дію `action`, на кнопки під назвою `name`;
- `UnbindAction(ActionButtonName name, Action action)` – метод, який відписує дію `action`, від кнопок під назвою `name`.

`TaskManager` – Основний Клас.

`TaskManager` є головним класом, який зберігає список завдань (тасок), спавн-поінти для цих завдань, а також відповідає за відображення `TaskPopup` (випливаюче вікно для прийняття або відхилення завдання).

`TaskBase` – Базовий Клас для Завдань.

TaskBase є базовим класом для створення нових завдань. Для створення нового завдання необхідно успадковуватися від цього класу.

5.5 Висновки за розділом 5

Перш ніж розпочати активну розробку гри, було визначено загальну концепцію та структуру історії гри. Це включало в себе створення фантазійного світу, де гравець міг би відчувати себе частиною епічного пригодницького шляху. Було спрацьовано над створенням головної сюжетної лінії, персонажів, квестів та взаємодії з персонажами. Сюжет гри включав елементи пригод та боротьби зі злом.

Цей розділ розкриває ключові аспекти розробки дизайну ігрових локацій, персонажів, що дозволить створити захоплююче інтерактивне середовище для гравців.

Розглянули менеджери і їх взаємодію між собою, що забезпечує комплексну і злагоджену роботу проєкту, яка полегшує розробку, управління і розширення функціональності гри.

ВИСНОВКИ

В ході виконання дипломної роботи бакалавра було проаналізовано та досліджено всі професії, які існують при розробці гри, процес розробки комп'ютерних ігор та створено власну комп'ютерну гру в жанрі пригоди з елементами RPG.

Аналіз професій у розробці ігор. Визначено, що у наш час існують 4 основні категорії розробників:

- програмісти. Відповідальні за написання коду, реалізацію ігрової логіки та механік;
- художники. Створюють візуальні елементи гри, такі як персонажі, середовища, анімації;
- геймдизайнери. Розробляють концепції та механіки гри, визначають правила та ігровий процес;
- звукорежисери. Створюють звукові ефекти та музику для гри.

Етапи розробки гри. Процес розробки гри включає 7 основних етапів:

- концепція. Формулювання ідеї гри, визначення основних механік та стилю;
- прототипування. Створення базової версії гри для перевірки основних ідей та механік;
- вертикальний зріз. Розробка невеликого, але повністю функціонального фрагменту гри;
- виробництво. Основний етап розробки, під час якого створюються всі елементи гри;
- тестування. Перевірка гри на наявність багів, тестування ігрового процесу;
- підготовка до релізу. Завершення всіх робіт, підготовка гри до випуску.

Вимоги до гри. Сформульовано вимоги до комп'ютерної гри:

- використання платформи Unity, яка забезпечує весь необхідний функціонал для розробки гри та є легшою у навчанні для новачків;
- використання середовища розробки Visual Studio та мови програмування C# для реалізації проекту.

Розробка гри. Розроблено схематичну структуру гри та створено гру у жанрі Roguelike. Гра має основний функціонал, властивий цьому жанру:

- 2D графіка. Створення двовимірних зображень та анімацій;
- динамічні бойові механіки, що включають бій у ближньому бою.

Тестування. Виконано тестування комп'ютерної гри. За результатами тестування гри на платформі Unity виявлено, що програма відповідає технічному завданню, функціонує коректно та дозволяє завершити гру.

Таким чином, в ході розробки гри у форматі 2D у жанрі пригоди з елементами RPG було успішно застосовано принципи і методи, описані в дипломній роботі, що підтверджує доцільність та ефективність обраних підходів у створенні якісної комп'ютерної гри.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Комп'ютерна гра [Електронний ресурс]. – Режим доступу : <https://uk.wikipedia.org/wiki/%D0%92%D1%96%D0%B4%D0%B5%D0%BE%D0%B3%D1%80%D0%B>
2. Грамолин, В.В. Обучающие компьютерные игры / В.В. Грамолин. – М. : ИНФО. 1994. – № 4. – 60 с.
3. Литвин, І.І. Інформатика / І.І. Литвин, О.М. Конончук, Ю.Л. Дещинський. – Львів : Новий світ, 2004. – 304 с.
4. Дуброва, Т. Вплив комп'ютерних ігор на особистість дитини / Т. Дуброва // Психолог. Шкільний світ. – 2010. – № 8. – С. 26-30.
5. Бурлаков, И. Психология компьютерных игр / И. Бурлаков // Наука и жизнь. — 1999. — № 5 — С. 88-92.
6. Five Key Game Development Stages: A Look Behind The Scenes [Electronic resource] – Access mode : <https://game-ace.com/blog/game-development-stages/>

ДОДАТОК А
Текст програми

Скрипт “MainPlayerActor.cs”

```

using System;
using System.Collections.Generic;
using UnityEngine;
using GameLogic.Actors;
using Core.GameManagerLogic;

namespace GameLogic.Actors.ActorTypes
{
    public class MainPlayerActor:PlayerActor
    {
        public Dictionary<ActorType, PlayerActor> myTeam = new
Dictionary<ActorType, PlayerActor>();
        private Vector3 offset;
        public bool IsMyTeamFollowsMe;
        public static MainPlayerActor Instance;

        public void Awake()
        {
            if (Instance == null) Instance = this;
            IsMyTeamFollowsMe = true;
            maxHealth = 100;
            //moveSpeed = 1.5f;
            Debug.Log("AWA");
            maxDistancePerMove = 10;
            SetAction(new Behaviours.SwordAction());
        }

        public override void MyUpdate()

```

```

    {
        if (isHandleMove)
        {
            if (rb.velocity.x != 0 || rb.velocity.y != 0)
            {
                anim.SetBool("Run", true);
            }
            else
            {
                anim.SetBool("Run", false);
            }
            rb.velocity = newPos;
        }
        if (myTeam.Count > 0 && IsMyTeamFollowsMe)
MyTeamFollowMe();
    }

    private void MyTeamFollowMe()
    {
        foreach (var myFriend in myTeam)
        {
            myFriend.Value.FollowTo(transform.position);
        }
    }

    public void AddToMyTeam(ActorType characterType,
PlayerActor character)
    {
        myTeam.Add(characterType, character);
    }

```

```

        HUDController.Instance.AddTeamUserToHUD(character.info.actorIcon,
character.info.actorName);
    }
    public void AskForHelp(ActorType characterHelper,
PlayerActor characterNeedHelp)
    {
        if (!myTeam.ContainsKey(characterHelper))
        {
            Debug.LogWarning("You dont have this guy in
your team");

            return;
        }

        myTeam[characterHelper].PerformAction(characterNeedHelp);
    }

    public override void MyLateUpdate()
    {
        Debug.LogWarning("MainPL");
    }
}
}

```

Скрипт “Enemy Actor”

```

using System;
using UnityEngine;

namespace GameLogic.Actors
{

```

```

public class EnemyActor:CharacterActor
{
    private void Awake()
    {
        health = 5;
        SetAction(new
GameLogic.Actors.Behaviours.SwordAction());
    }
    public override void ReachedDestinationPoint()
    {
        base.ReachedDestinationPoint();
        Debug.Log("Kick U!");
    }
    private void OnMouseDown()
    {
        if (Level1EntryPoint.Instance.movesCount >= 1
|| !Level1EntryPoint.Instance.CanAttackEnemy) return;
        Level1EntryPoint.Instance.SelectedEnemyActor = this;
        Level1EntryPoint.Instance.movesCount++;
        Debug.Log("OK");
    }
}
}

```

“ArrowActor.cs”

```

using System;
using System.Collections;
using UnityEngine;

namespace GameLogic.Actors

```

```

{
    public class ArrowActor:MonoBehaviour,
Core.GameManagerLogic.IUpdateble
    {
        private Rigidbody2D rb;
        public float damage = 1;
        public int dir;

        private void Start()
        {

Core.GameManagerLogic.GameManager.FInstance.AddUpdatableItem(this);
            rb = GetComponent<Rigidbody2D>();
            Destroy(gameObject, 2);
            StartCoroutine(EnableCollider());
        }

        private IEnumerator EnableCollider()
        {
            yield return new WaitForSeconds(.1f);
            GetComponent<BoxCollider2D>().enabled = true;
        }

        private void OnCollisionEnter2D(Collision2D collision)
        {
            var enemyCollides =
collision.collider.GetComponent<EnemyActor>();
            if (enemyCollides)
            {
                enemyCollides.GetDamage(damage);
            }
        }
    }
}

```

```

        Destroy(gameObject);
    }
}

private void OnDestroy()
{
    Core.GameManagerLogic.GameManager.FInstance.RemoveUpdatableItem(this);
}

public void MyUpdate()
{
    Debug.Log("AAA");
    rb.velocity = dir * Vector2.right * 500 * Time.deltaTime;
}
}
}

```

“CharacterActor.cs”

```

using System;
using System.Collections;
using UnityEngine;
using Core.GameManagerLogic;

namespace GameLogic.Actors
{
    public abstract class CharacterActor : MonoBehaviour, IUpdateble,
    ILateUpdateble
    {
        public SpriteRenderer characterSprite;
    }
}

```

```

public Animator anim;
public float maxHealth;
[HideInInspector] public float health;
public float moveSpeed;
[HideInInspector] public float maxDistancePerMove;
protected Rigidbody2D rb;
protected Behaviours.ActionBehaviour ActionBehaviour;
private bool IsFollow;
[HideInInspector] public bool IsInvertedSpriteFlip;

public Action OnPathFinished;
public Action OnDied;
public Action OnReachedDestinationPoint;
public bool IsHealthDisplay;
public TMPro.TextMeshProUGUI healthText;

public virtual void ReachedDestinationPoint() { }

private Collider2D myCollider2D;
public void Start()
{
    health = maxHealth;
    myCollider2D = GetComponent<Collider2D>();
    rb = GetComponent<Rigidbody2D>();
    GameManager.FInstance.AddUpdatableItem(this);
    GameManager.FInstance.AddLateUpdatableItem(this);
}
public void OnDestroy()
{
    GameManager.FInstance.RemoveUpdatableItem(this);
}

```

```

GameManager.FInstance.RemoveLateUpdatableItem(this);
    }

    public void FreezePosition() => myCollider2D.enabled =
false;//rb.constraints = RigidbodyConstraints2D.FreezeAll;
    public void UnfreezePosition() {
        myCollider2D.enabled = true;
        //rb.constraints =
RigidbodyConstraints2D.FreezeRotation;
    }

    public IEnumerator MovePath(Vector3[] path)
    {
        var currentPathPointID = 0;
        int lastPathPointID() => path.Length - 1;
        float distanceToPoint() =>
Vector3.Distance(transform.position, path[currentPathPointID]);
        bool isReachedThePoint() => distanceToPoint() < 0.5f;
        bool isPathFinished() => currentPathPointID >
lastPathPointID();

        Vector3 moveDirection() => path[currentPathPointID] -
transform.position;

        anim.SetBool("Run", true);
        while (!isPathFinished())
        {
            while (!isReachedThePoint())
            {

```

```

        Debug.Log($"ReachedPoint:
{isReachedThePoint()}" +
        $"DistanceToPoint:
{distanceToPoint()}");

        if (moveDirection().x > 0)
characterSprite.flipX = IsInvertedSpriteFlip;
        if (moveDirection().x < 0)
characterSprite.flipX = !IsInvertedSpriteFlip;

        transform.position =
Vector3.MoveTowards(transform.position, path[currentPathPointID], moveSpeed*
Time.deltaTime);

        yield return null;
    }
    Debug.Log("POINT REACHED");
    currentPathPointID++;
    yield return null;
}
anim.SetBool("Run", false);
Debug.Log("ON PLACE");
OnPathFinished?.Invoke();
}

public void FollowTo(Vector3 targetPoint)
{
    var distance = Vector2.Distance(transform.position,
targetPoint);

    var moveDirection = targetPoint - transform.position;

    if (distance>1.5f)
    {

```

```

        if (!IsFollow)
        {
            IsFollow = true;
            UnfreezePosition();
        }
        anim.SetBool("Run", true);
        if (moveDirection.x > 0) characterSprite.flipX =
IsInvertedSpriteFlip;

        if (moveDirection.x < 0) characterSprite.flipX
= !IsInvertedSpriteFlip;

        transform.position =
Vector3.MoveTowards(transform.position, targetPoint, moveSpeed *
Time.deltaTime);
    }
    else
    {
        if (IsFollow)
        {
            anim.SetBool("Run", false);
            ReachedDestinationPoint();
            OnReachedDestinationPoint?.Invoke();
            IsFollow = false;
            FreezePosition();
        }
    }
}

public void SetAction(Behaviours.ActionBehaviour
actionBehaviour)
{

```

```

        Debug.Log("SETTED");
        ActionBehaviour = actionBehaviour;
    }
    public void PerformAction(CharacterActor character)
    {
        anim.SetTrigger("Action");
        ActionBehaviour.Action(character);
    }
    public void GetDamage(float damage)
    {
        Debug.Log($"MY HP {health}");
        health -= damage;
        if (IsHealthDisplay) healthText.text = $"{health}";
        if (health <= 0)
        {
            health = 0;
            anim.SetTrigger("Death");
            OnDied?.Invoke();
            GetComponent<Collider2D>().enabled = false;
            anim.SetTrigger("Death");

            //transform.GetChild(0).gameObject.SetActive(false);
            if (IsHealthDisplay)

            healthText.transform.parent.gameObject.SetActive(false);
            //Destroy(gameObject);
            Debug.Log($" {gameObject.name} is DEAD");
        }
    }
}

```

```

        public virtual void MyLateUpdate() { }

        public virtual void MyUpdate() { }
    }
}

```

“PlayerActor.cs”

```

using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using Core.GameManagerLogic;
using Core.InputManagerLogic;

namespace GameLogic.Actors
{
    public class PlayerActor : CharacterActor
    {
        public ActorsInfo.ActorInfo info;
        [HideInInspector] public bool isHandleMove;
        protected Vector2 newPos = Vector2.zero;
        private int positiveMove = 1;
        private int negativeMove = -1;

        public void LockMove()
        {
            newPos.y = newPos.x = 0;
            rb.velocity = Vector2.zero;
            isHandleMove = false;
            anim.SetBool("Run", false);
        }
    }
}

```

```

        InputManager.FInstance.UnbindAction(ActionButtonName.StartMoveForward, StartMoveForward);

```

```

        InputManager.FInstance.UnbindAction(ActionButtonName.StopMoveForward, StopMoveForward);

```

```

        InputManager.FInstance.UnbindAction(ActionButtonName.StartMoveBackward, StartMoveBackward);

```

```

        InputManager.FInstance.UnbindAction(ActionButtonName.StopMoveBackward, StopMoveBackward);

```

```

        InputManager.FInstance.UnbindAction(ActionButtonName.StartMoveLeft, StartMoveLeft);

```

```

        InputManager.FInstance.BindAction(ActionButtonName.StopMoveLeft, StopMoveLeft);

```

```

        InputManager.FInstance.UnbindAction(ActionButtonName.StartMoveRight, StartMoveRight);

```

```

        InputManager.FInstance.UnbindAction(ActionButtonName.StopMoveRight, StopMoveRight);

```

```

    }
    public void UnlockMove()
    {

```

```
isHandleMove = true;
```

```
//InputManager.FInstance.BindAxis(AxisButtonName.MoveForward,  
MoveForward);
```

```
//InputManager.FInstance.BindAxis(AxisButtonName.MoveBackward,  
MoveBackward);
```

```
//InputManager.FInstance.BindAxis(AxisButtonName.MoveRight,  
MoveRight);
```

```
//InputManager.FInstance.BindAxis(AxisButtonName.MoveLeft, MoveLeft);
```

```
InputManager.FInstance.BindAction(ActionButtonName.StartMoveForward,  
StartMoveForward);
```

```
InputManager.FInstance.BindAction(ActionButtonName.StopMoveForward,  
StopMoveForward);
```

```
InputManager.FInstance.BindAction(ActionButtonName.StartMoveBackward,  
StartMoveBackward);
```

```
InputManager.FInstance.BindAction(ActionButtonName.StopMoveBackward,  
StopMoveBackward);
```

```
InputManager.FInstance.BindAction(ActionButtonName.StartMoveLeft,  
StartMoveLeft);
```

```
InputManager.FInstance.BindAction(ActionButtonName.StopMoveLeft,
StopMoveLeft);
```

```
InputManager.FInstance.BindAction(ActionButtonName.StartMoveRight,
StartMoveRight);
```

```
InputManager.FInstance.BindAction(ActionButtonName.StopMoveRight,
StopMoveRight);
```

```
    }

    protected virtual void StartMoveForward()
    {
        newPos.y = moveSpeed * positiveMove;
    }
    protected virtual void StopMoveForward()
    {
        newPos.y = 0;
    }

    protected virtual void StartMoveBackward()
    {
        newPos.y = moveSpeed * negativeMove;
    }
    protected virtual void StopMoveBackward()
    {
        newPos.y = 0;
    }
}
```

```

protected virtual void StartMoveRight()
{
    newPos.x = moveSpeed * positiveMove;
    characterSprite.flipX = false;
}

protected virtual void StopMoveRight()
{
    newPos.x = 0;
}

protected virtual void StartMoveLeft()
{
    newPos.x = moveSpeed * negativeMove;
    characterSprite.flipX = true;
}

protected virtual void StopMoveLeft()
{
    newPos.x = 0;
}

//protected virtual void MoveForward(float scale)
//{
//    newPos.y += scale * moveSpeed * Time.deltaTime;
//}

//protected virtual void MoveBackward(float scale)
//{
//    newPos.y += scale * moveSpeed * Time.deltaTime;
//}

protected virtual void MoveLeft(float scale)
//{

```

```

        //      newPos.x += scale * moveSpeed * Time.deltaTime;
    //}
    //protected virtual void MoveRight(float scale)
    //{
    //      newPos.x += scale * moveSpeed * Time.deltaTime;
    //}
    }
}

```

“SpriteSkin.cs”

#pragma warning disable 0168 // variable declared but not used.

```

#if      ENABLE_ANIMATION_COLLECTION      &&
ENABLE_ANIMATION_BURST
#define ENABLE_SPRITESKIN_COMPOSITE
#endif

```

```

using System;
using System.Collections.Generic;
using UnityEngine.Scripting;
using UnityEngine.U2D.Common;
using Unity.Collections;
using UnityEngine.Rendering;
using UnityEngine.Scripting.APIUpdating;

```

```

namespace UnityEngine.U2D.Animation
{
    public struct PositionVertex
    {
        public Vector3 position;
    }
}

```

```

public struct PositionTangentVertex
{
    public Vector3 position;
    public Vector4 tangent;
}

struct DeformVerticesBuffer
{
    public const int k_DefaultBufferSize = 2;
    int m_BufferCount;
    int m_CurrentBuffer;
    NativeArray<byte>[] m_DeformedVertices;

    public DeformVerticesBuffer(int bufferCount)
    {
        m_BufferCount = bufferCount;
        m_DeformedVertices = new NativeArray<byte>[m_BufferCount];
        for (int i = 0; i < m_BufferCount; ++i)
        {
            m_DeformedVertices[i] = new NativeArray<byte>(1,
Allocator.Persistent);
        }
        m_CurrentBuffer = 0;
    }

    public void Dispose()
    {
        for (int i = 0; i < m_BufferCount; ++i)
        {

```

```

        if (m_DeformedVertices[i].IsCreated)
            m_DeformedVertices[i].Dispose();
    }
}

public ref NativeArray<byte> GetBuffer(int expectedSize)
{
    m_CurrentBuffer = (m_CurrentBuffer + 1) % m_BufferCount;
    if (m_DeformedVertices[m_CurrentBuffer].IsCreated &&
        m_DeformedVertices[m_CurrentBuffer].Length != expectedSize)
    {
        m_DeformedVertices[m_CurrentBuffer].Dispose();
        m_DeformedVertices[m_CurrentBuffer] = new
NativeArray<byte>(expectedSize, Allocator.Persistent);
    }
    return ref m_DeformedVertices[m_CurrentBuffer];
}

internal ref NativeArray<byte> GetCurrentBuffer()
{
    return ref m_DeformedVertices[m_CurrentBuffer];
}
}

/// <summary>
/// Deforms the Sprite that is currently assigned to the SpriteRenderer in the
same GameObject
/// </summary>
[Preserve]
[ExecuteInEditMode]

```

```
[DefaultExecutionOrder(-1)]
[DisallowMultipleComponent]
[RequireComponent(typeof(SpriteRenderer))]
[AddComponentMenu("2D Animation/Sprite Skin")]
[MovedFrom("UnityEngine.U2D.Experimental.Animation")]
```

```
[HelpURL("https://docs.unity3d.com/Packages/com.unity.2d.animation@4.1/manual/index.html")]
```

```
public sealed partial class SpriteSkin : MonoBehaviour,
ISerializationCallbackReceiver
```

```
{
    [SerializeField]
    private Transform m_RootBone;
    [SerializeField]
    private Transform[] m_BoneTransforms = new Transform[0];
    [SerializeField]
    private Bounds m_Bounds;
    [SerializeField]
    private bool m_UseBatching = true;
    [SerializeField]
    private bool m_AlwaysUpdate = true;
```

```
// The deformed m_SpriteVertices stores all 'HOT' channels only in single-
stream and essentially depends on Sprite Asset data.
```

```
// The order of storage if present is POSITION, NORMALS, TANGENTS.
```

```
private DeformVerticesBuffer m_DeformedVertices;
private int m_CurrentDeformVerticesLength = 0;
private SpriteRenderer m_SpriteRenderer;
private int m_CurrentDeformSprite = 0;
private bool m_ForceSkinning;
```

```
private bool m_BatchSkinning = false;
bool m_IsValid = false;
int m_TransformsHash = 0;
```

```
internal bool batchSkinning
{
    get { return m_BatchSkinning; }
    set { m_BatchSkinning = value; }
}
```

```
#if UNITY_EDITOR
```

```
    internal static Events.UnityEvent onDrawGizmos = new Events.UnityEvent();
    private void OnDrawGizmos() { onDrawGizmos.Invoke(); }
```

```
private bool m_IgnoreNextSpriteChange = true;
internal bool ignoreNextSpriteChange
{
    get { return m_IgnoreNextSpriteChange; }
    set { m_IgnoreNextSpriteChange = value; }
}
```

```
#endif
```

```
private int GetSpriteInstanceID()
{
    return sprite != null ? sprite.GetInstanceID() : 0;
}
```

```
void OnEnable()
{
    m_TransformsHash = 0;
```

```

        CacheCurrentSprite();
        OnEnableBatch();
        m_DeformedVertices = new
DeformVerticesBuffer(DeformVerticesBuffer.k_DefaultBufferSize);
    }

    void CacheValidFlag()
    {
        m_IsValid = isValid;
        if(!m_IsValid)
            DeactivateSkinning();
    }

    void Reset()
    {
        CacheValidFlag();
        OnResetBatch();
    }

    internal void UseBatching(bool value)
    {
        if (m_UseBatching != value)
        {
            m_UseBatching = value;
            UseBatchingBatch();
        }
    }

    internal ref NativeArray<byte> GetDeformedVertices(int spriteVertexCount)
    {

```

```

    if (sprite != null)
    {
        if (m_CurrentDeformVerticesLength != spriteVertexCount)
        {
            m_TransformsHash = 0;
            m_CurrentDeformVerticesLength = spriteVertexCount;
        }
    }
    else
    {
        m_CurrentDeformVerticesLength = 0;
    }
    return ref

m_DeformedVertices.GetBuffer(m_CurrentDeformVerticesLength);
}

/// <summary>
/// Returns whether this SpriteSkin has currently deformed vertices.
/// </summary>
/// <returns>Returns true if this SpriteSkin has currently deformed vertices.
Returns false otherwise.</returns>
public bool HasCurrentDeformedVertices()
{
    if (!m_IsValid)
        return false;

#if ENABLE_SPRITESKIN_COMPOSITE
    return m_DataIndex >= 0;
#else

```

```

        return m_CurrentDeformVerticesLength > 0 &&
m_DeformedVertices.GetCurrentBuffer().IsCreated;
#endif
    }

    /// <summary>
    /// Gets a byte array to the currently deformed vertices for this SpriteSkin.
    /// </summary>
    /// <returns>Returns a reference to the currently deformed vertices. This is
valid only for this calling frame.</returns>
    /// <exception cref="InvalidOperationException">Thrown when there are no
currently deformed vertices</exception>
    internal NativeArray<byte> GetCurrentDeformedVertices()
    {
        if (!m_IsValid)
            throw new InvalidOperationException("The SpriteSkin deformation is
not valid.");

#if ENABLE_SPRITESKIN_COMPOSITE
        if (m_DataIndex < 0)
        {
            throw new InvalidOperationException("There are no currently deformed
vertices.");
        }
        return
SpriteSkinComposite.instance.GetDeformableBufferForSprite(m_DataIndex);
#else
        if (m_CurrentDeformVerticesLength <= 0)
            throw new InvalidOperationException("There are no currently deformed
vertices.");
#endif
    }

```

```

var buffer = m_DeformedVertices.GetCurrentBuffer();
if (!buffer.IsCreated)
    throw new InvalidOperationException("There are no currently deformed
vertices.");
return buffer;
#endif
}

/// <summary>
/// Gets an array of currently deformed position vertices for this SpriteSkin.
/// </summary>
/// <returns>Returns a reference to the currently deformed vertices. This is
valid only for this calling frame.</returns>
/// <exception cref="InvalidOperationException">
/// Thrown when there are no currently deformed vertices or if the deformed
vertices does not contain only
/// position data.
/// </exception>
internal NativeSlice<PositionVertex> GetCurrentDeformedVertexPositions()
{
    if (sprite.HasVertexAttribute(VertexAttribute.Tangent))
        throw new InvalidOperationException("This SpriteSkin has deformed
tangents");
    if (!sprite.HasVertexAttribute(VertexAttribute.Position))
        throw new InvalidOperationException("This SpriteSkin does not have
deformed positions.");

    var deformedBuffer = GetCurrentDeformedVertices();
    return deformedBuffer.Slice().SliceConvert<PositionVertex>();
}

```

```

    /// <summary>
    /// Gets an array of currently deformed position and tangent vertices for this
    SpriteSkin.
    /// </summary>
    /// <returns>
    /// Returns a reference to the currently deformed position and tangent vertices.
    This is valid only for this calling frame.
    /// </returns>
    /// <exception cref="InvalidOperationException">
    /// Thrown when there are no currently deformed vertices or if the deformed
    vertices does not contain only
    /// position and tangent data.
    /// </exception>
    internal NativeSlice<PositionTangentVertex>
    GetCurrentDeformedVertexPositionsAndTangents()
    {
        if (!sprite.HasVertexAttribute(VertexAttribute.Tangent))
            throw new InvalidOperationException("This SpriteSkin does not have
            deformed tangents");
        if (!sprite.HasVertexAttribute(VertexAttribute.Position))
            throw new InvalidOperationException("This SpriteSkin does not have
            deformed positions.");

        var deformedBuffer = GetCurrentDeformedVertices();
        return deformedBuffer.Slice().SliceConvert<PositionTangentVertex>();
    }

    /// <summary>

```

/// Gets an enumerable to iterate through all deformed vertex positions of this SpriteSkin.

/// </summary>

/// <returns>Returns an IEnumerable to deformed vertex positions.</returns>

/// <exception cref="InvalidOperationException">Thrown when there is no vertex positions or deformed vertices.</exception>

public IEnumerable<Vector3> GetDeformedVertexPositionData()

{

bool hasPosition =

sprite.HasVertexAttribute(Rendering.VertexAttribute.Position);

if (!hasPosition)

throw new InvalidOperationException("Sprite does not have vertex position data.");

var rawBuffer = GetCurrentDeformedVertices();

var rawSlice =

rawBuffer.Slice(sprite.GetVertexStreamOffset(VertexAttribute.Position));

return new NativeCustomSliceEnumerator<Vector3>(rawSlice, sprite.GetVertexCount(), sprite.GetVertexStreamSize());

}

/// <summary>

/// Gets an enumerable to iterate through all deformed vertex tangents of this SpriteSkin.

/// </summary>

/// <returns>Returns an IEnumerable to deformed vertex tangents.</returns>

/// <exception cref="InvalidOperationException">Thrown when there is no vertex tangents or deformed vertices.</exception>

public IEnumerable<Vector4> GetDeformedVertexTangentData()

{

```

        bool hasTangent =
sprite.HasVertexAttribute(Rendering.VertexAttribute.Tangent);
        if (!hasTangent)
            throw new InvalidOperationException("Sprite does not have vertex
tangent data.");

        var rawBuffer = GetCurrentDeformedVertices();
        var rawSlice =
rawBuffer.Slice(sprite.GetVertexStreamOffset(VertexAttribute.Tangent));
        return new NativeCustomSliceEnumerator<Vector4>(rawSlice,
sprite.GetVertexCount(), sprite.GetVertexStreamSize());
    }

    void OnDisable()
    {
        DeactivateSkinning();
        m_DeformedVertices.Dispose();
        OnDisableBatch();
    }

#if ENABLE_SPRITESKIN_COMPOSITE
        internal void OnLateUpdate()
#else
        void LateUpdate()
#endif
    {
        CacheCurrentSprite();
        if (IsValid && !batchSkinning && this.enabled && (this.alwaysUpdate ||
this.spriteRenderer.isVisible))
        {

```

```

var transformHash = SpriteSkinUtility.CalculateTransformHash(this);
var    spriteVertexCount    =    sprite.GetVertexStreamSize()    *
sprite.GetVertexCount();
    if (spriteVertexCount > 0 && m_TransformsHash != transformHash)
    {
        var inputVertices = GetDeformedVertices(spriteVertexCount);
        SpriteSkinUtility.Deform(sprite,
gameObject.transform.worldToLocalMatrix, boneTransforms, ref inputVertices);
        SpriteSkinUtility.UpdateBounds(this, inputVertices);
        InternalEngineBridge.SetDeformableBuffer(spriteRenderer,
inputVertices);
        m_TransformsHash = transformHash;
        m_CurrentDeformSprite = GetSpriteInstanceID();
    }
}
}
}

```

```

void CacheCurrentSprite()
{
    if (m_CurrentDeformSprite != GetSpriteInstanceID())
    {
        DeactivateSkinning();
        m_CurrentDeformSprite = GetSpriteInstanceID();
        UpdateSpriteDeform();
        CacheValidFlag();
        m_TransformsHash = 0;
    }
}
}

```

```

internal Sprite sprite

```

```

{
    get
    {
        if (spriteRenderer == null)
            return null;
        return spriteRenderer.sprite;
    }
}

```

internal SpriteRenderer spriteRenderer

```

{
    get
    {
        if (m_SpriteRenderer == null)
            m_SpriteRenderer = GetComponent<SpriteRenderer>();
        return m_SpriteRenderer;
    }
}

```

/// <summary>

/// Returns the Transform Components that is used for deformation

/// </summary>

/// <returns>An array of Transform Components</returns>

public Transform[] boneTransforms

```

{
    get { return m_BoneTransforms; }
    internal set
    {
        m_BoneTransforms = value;
        CacheValidFlag();
    }
}

```

```

        OnBoneTransformChanged();
    }
}

```

```

/// <summary>
/// Returns the Transform Component that represents the root bone for
deformation

```

```

/// </summary>
/// <returns>A Transform Component</returns>
public Transform rootBone
{
    get { return m_RootBone; }
    internal set
    {
        m_RootBone = value;
        CacheValidFlag();
        OnRootBoneTransformChanged();
    }
}

```

```

internal Bounds bounds
{
    get { return m_Bounds; }
    set { m_Bounds = value; }
}

```

```

/// <summary>
/// Determines if the SpriteSkin executes even if the associated
/// SpriteRenderer has been culled from view.
/// </summary>

```

```

public bool alwaysUpdate
{
    get => m_AlwaysUpdate;
    set => m_AlwaysUpdate = value;
}

```

```

internal bool isValid
{
    get { return this.Validate() == SpriteSkinValidationResult.Ready; }
}

```

```

void OnDestroy()
{
    DeactivateSkinning();
}

```

```

internal void DeactivateSkinning()
{
    var sprite = spriteRenderer.sprite;

    if (sprite != null)
        InternalEngineBridge.SetLocalAABB(spriteRenderer, sprite.bounds);
}

```

```

SpriteRendererDataAccessExtensions.DeactivateDeformableBuffer(spriteRenderer)
;
}

```

```

internal void ResetSprite()
{

```

```
m_CurrentDeformSprite = 0;
CacheValidFlag();
}

public void OnBeforeSerialize()
{
    OnBeforeSerializeBatch();
}

public void OnAfterDeserialize()
{
    OnAfterSerializeBatch();
}
}
```

ДОДАТОК Б
Слайди презентації



Рисунок Б.1 – Слайд 1

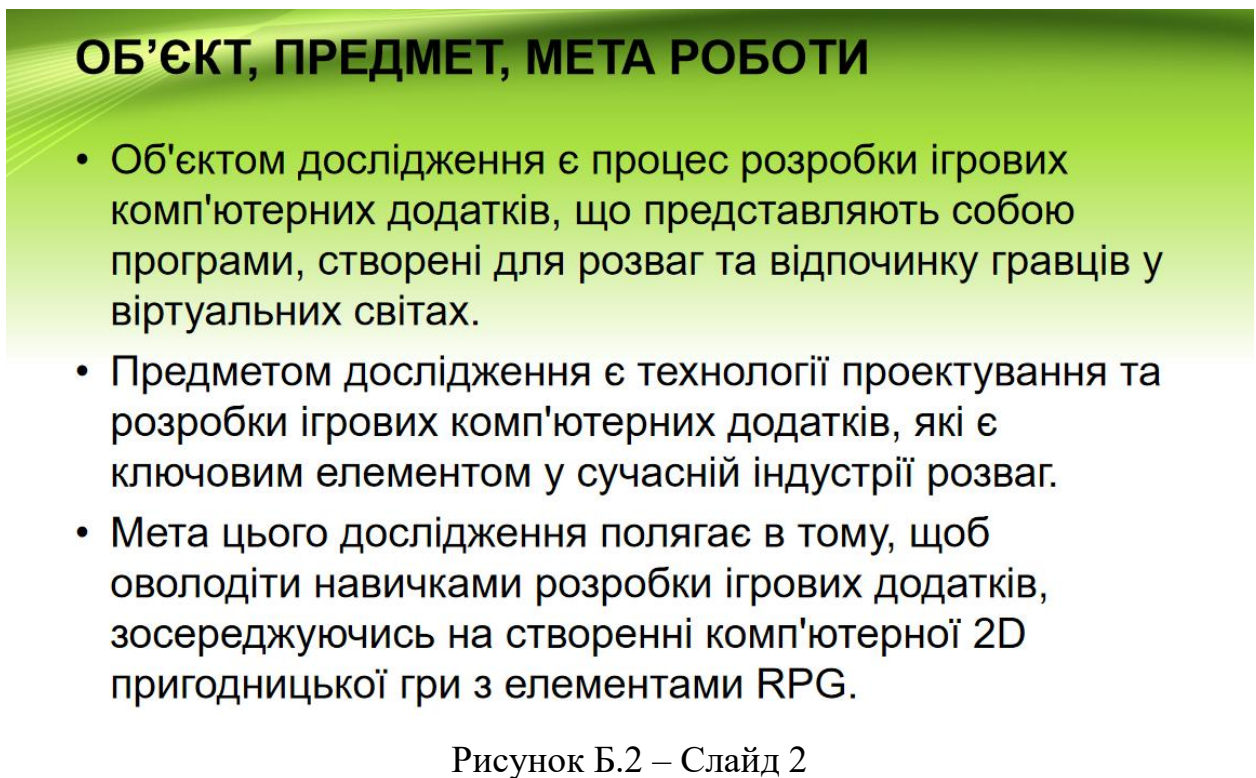


Рисунок Б.2 – Слайд 2

Класифікація комп'ютерних ігор

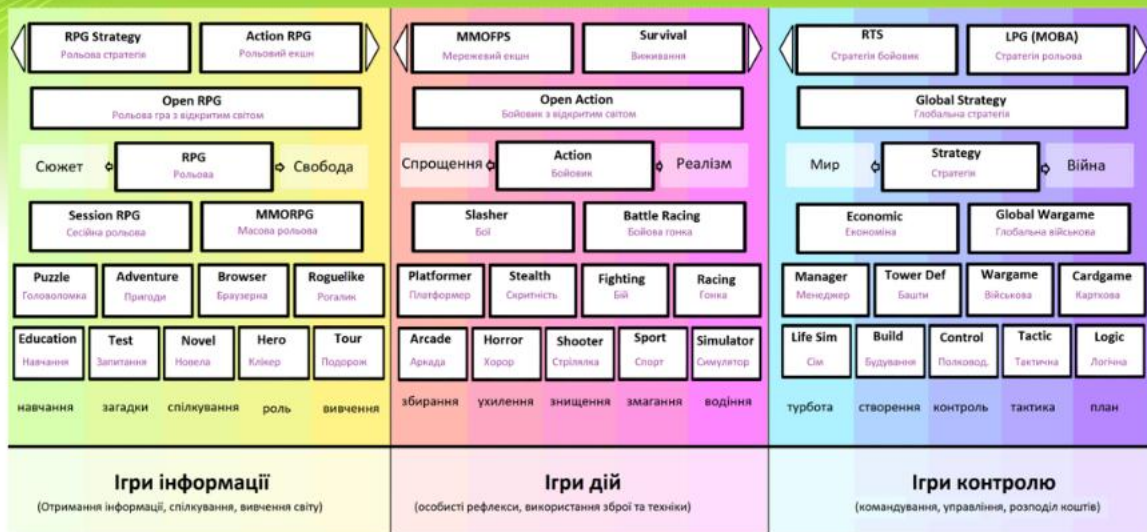


Рисунок Б.3 – Слайд 3

Вибір програмного забезпечення і мови програмування

Провевши аналіз і порівняв різні програмні забезпечення, було обрано один з найпопулярніших та потужних ігрових двигунів у світі - Unity.

Обґрунтування вибору:

- Мультиплатформенність
- Зручність та простота використання
- Сильна спільнота та підтримка
- Багатофункціональність та потужні інструменти
- Розширюваність та інтеграції
- Потужність



Рисунок Б.4 – Слайд 4

Обґрунтування вибору мови програмування C++



- **Висока продуктивність:**

C++ часто вважається мовою з вищою продуктивністю порівняно з C#, оскільки він дає більше контролю над апаратними ресурсами та оптимізацією.

- **Офіційна підтримка**

C++ є однією з основних мов програмування, яка підтримується Unity, і всі основні функції та API Unity розроблені з урахуванням цієї мови.

- **Простота та зручність**

C++ повністю підтримує ООП, що дозволяє організовувати код у вигляді класів та об'єктів, полегшуючи розробку та підтримку великих проектів.

- **Інструменти та підтримка**

C++ добре інтегрується з Visual Studio, одним з найпотужніших і зручних IDE (інтегрованих середовищ розробки). Це забезпечує доступ до розширених інструментів для налагодження, рефакторингу та аналізу коду.

Рисунок Б.5 – Слайд 5

РОЗРОБКА КОМП'ЮТЕРНОГО ДОДАТКУ

Рисунок Б.6 – Слайд 6

Визначення концепції

“Маленькі пригоди в Балабурсі” — це пригодницька гра з елементами рольової (RPG).

Світ гри: Дія відбувається у фантазійному світі, на просторах лісу, місцевості під назвою “Смарагдовий ліс”.

Атмосфера: “Маленькі пригоди в Балабурсі” створює затишну, спокійну атмосферу, досліджує теми відповідальності та наслідків вибору. Гра ставить гравця перед моральними дилемами, що змушує задуматися про наслідки своїх дій.

Сюжет: Гравець починає свою пригоду після викрадення несусвітними істотами і зараження паразитом, що загрожує перетворенням на одного з цих монстрів. Гравець повинен знайти спосіб позбутися паразита, подорожуючи Смарагдовими лісом він стикається з численними небезпеками.

Рисунок Б.7 – Слайд 7

Діалоги і рішення: Гра пропонує розгалужену систему діалогів, де вибір гравця впливає на розвиток сюжету, взаємовідносини з іншими персонажами та навіть на світ гри.



Рисунок Б.8 – Слайд 8

Фрагмент діалогу зроблений завдяки сайту model.io



Рисунок Б.9 – Слайд 9

Розробка дизайну ігрових локацій

При розробці ігрової локації було 4 етапи:

- Визначення концепції ігрових локацій. Як властиво для більшості подібних ігор було обрано тему "ліс", так як кожен звір, чи чужинець може бути потенційним ворогом для гравця.
- Проєктування локацій. Для цього етапу було використано спрайти тематики "ліс" і "селище".
- Створення 2D моделей і об'єктів. Для розробки спрайтів локації використовувалася програма PixelArt.
- Розміщення об'єктів у локації було зроблено, безпосередньо, завдяки кроссплатформовому середовищу розробки комп'ютерних ігор Unity.



Рисунок Б.10 – Слайд 10

Розробка ігрових персонажів та їх спрайтів

Спрайти повинні відповідати стилістиці та атмосфері гри. У даній грі кожен персонаж має бути не тільки сам по собі особливим, а ще й мати свою індивідуальну зброю.

Всі спрайти були розроблені у програмі PixelArt



Рисунок Б.11 – Слайд 11

Програмна реалізація комп'ютерного додатку

Приклади менеджерів:

- AnimatorManager – менеджер анімацій, який дозволяє перемикаати аніматори за іменем.
- AudioManager – менеджер, що відповідає за створення, видалення, програвання, зупинення та відновлення аудіо.
- DataManager – менеджер керування даними (збереження, завантаження, видалення).
- DialogueManager – менеджер діалогів. Запускає певний діалог, який зберігається у DialogueData.
- GameManager - точка входу в програму. DontDestroyOnLoad - не видаляється в процесі всієї гри.

Рисунок Б.12 – Слайд 12

Приклади діаграмів класів

Діаграма класів призначена для представлення внутрішньої структури програми у вигляді класів та зв'язків між ними. Ось приклад декількох із них:

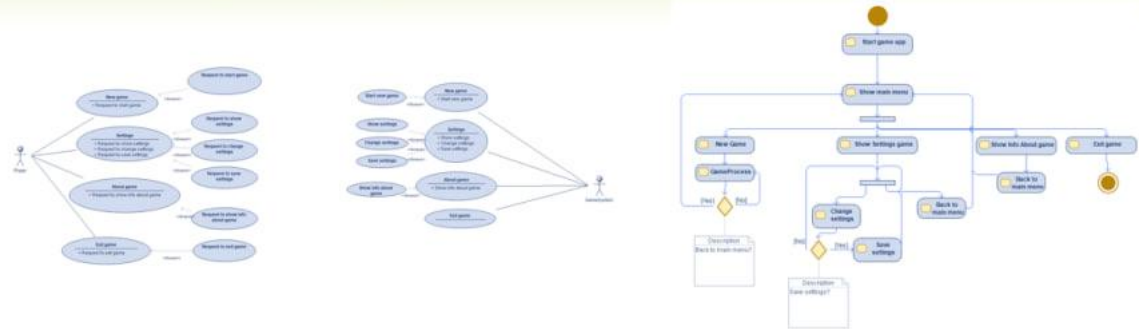


Рисунок Б.13 – Слайд 13

Діаграма прецедентів

Дана діаграма описує дії, які може здійснювати гравець та ігрова система. Гравець може подавати різні запити до ігрової системи, а вона в свою чергу оброблює та виконує ці запити.



Рисунок Б.14 – Слайд 14

Діаграма діяльності

Граф діяльностей є різновидом графу станів скінченного автомату, вершинами якого є певні дії, а переходи відбуваються по завершенню дій. На даному графі наочно видно структуру гри. Від запуску в головне меню і до виходу із самої гри.

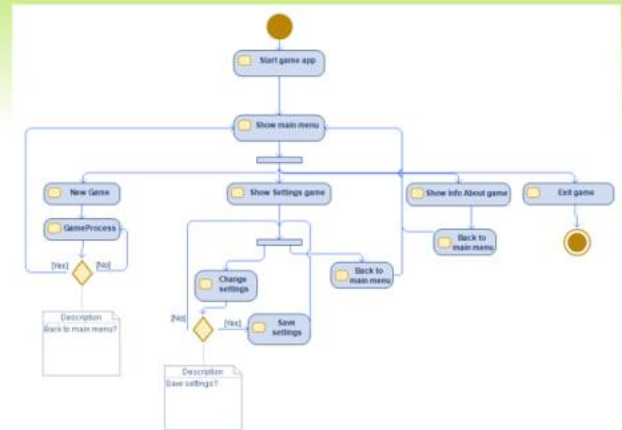


Рисунок Б.15 – Слайд 15

Діаграма послідовності

Ця діаграма описує життєвий цикл гри. Коли користувач запускає гру, то перед собою він бачить головне меню, в якому він може змінити налаштування гри, або переглянути відомості про гру. Також з головного меню можна розпочати сам процес гри.

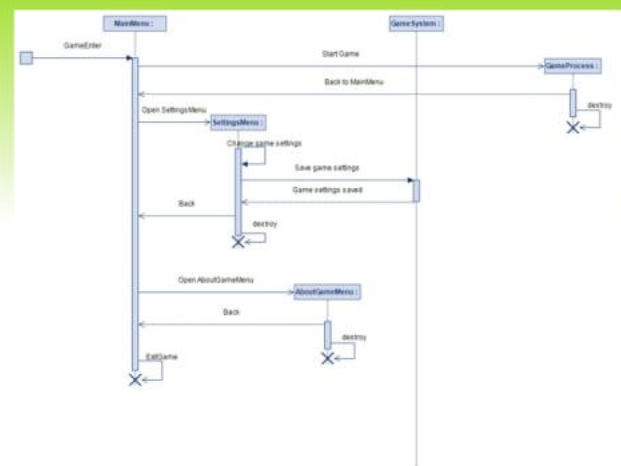


Рисунок Б.16 – Слайд 16

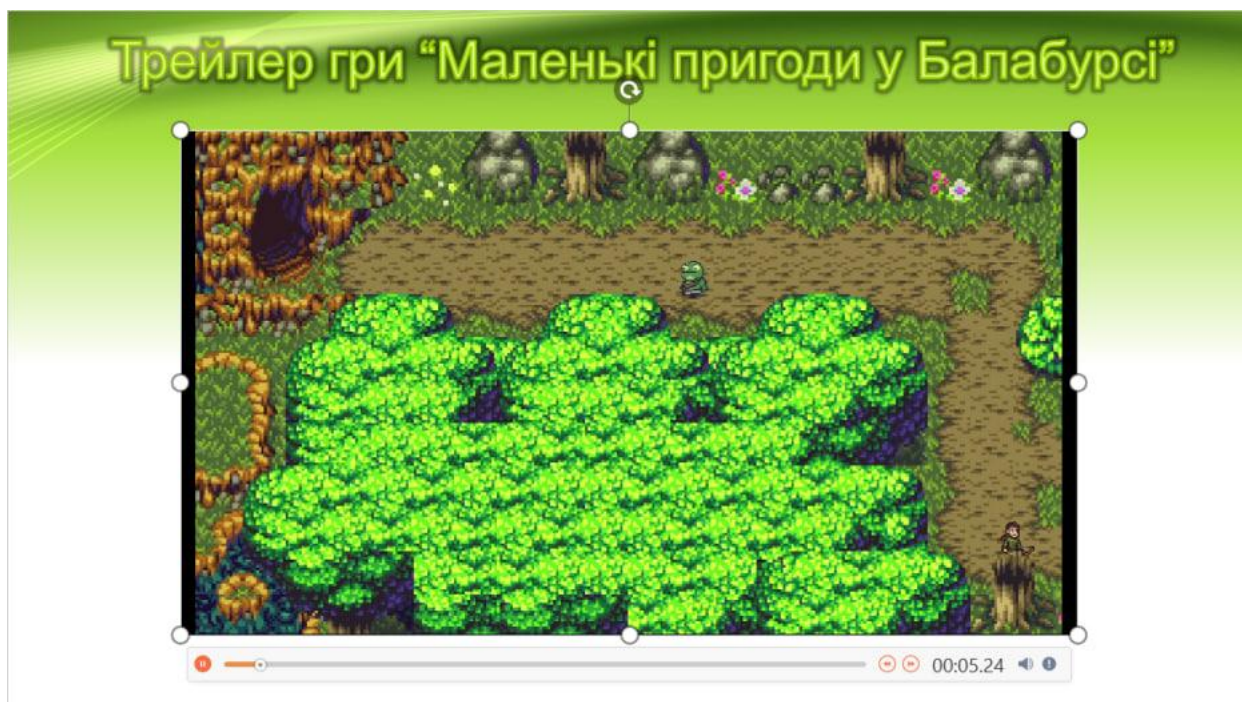


Рисунок Б.17 – Слайд 17

Висновки

В ході виконання дипломної кваліфікованої роботи було проаналізовано та досліджено процес розробки гри у жанрі пригоді з елементами RPG.

Гра має наступні функції:

- анімований персонаж із можливістю руху, атаки, можливістю взаємодіяти з іншими персонажами;
- діалоги з персонажами, в ході яких є можливість зробити вибір впливаючий на розвиток сюжету;
- отримання і виконання ігрових завдань;
- ведення пошагового бою.

Рисунок Б.18 – Слайд 18

ДОДАТОК В**Відео-трейлер гри**

<https://youtu.be/uDRlif8iwDU>