МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інженерії програмного забезпечення

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри канд. техн. наук, доцент

_____ Є. О. Давиденко

підпис

«___» _____2024 р.

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

ІГРОВИЙ ЗАСТОСУНОК В ЖАНРІ ROGUELIKE НА ОСНОВІ РУШІЯ UNITY

Спеціальність «Інженерія програмного забезпечення»

121 – КРБ.01 – 408.22010812

Здобувач _____ М.С. Кубицький

підпис

«__» _____ 2024 р.

Керівник PhD, ст. викладач кафедри ІПЗ _____ К.О. Антіпова

підпис

«__» _____ 2024 р.

Консультант канд. техн. наук., доцент _____ А.О. Алексєєва

підпис

«__» _____ 2024 р.

Миколаїв – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет імені Петра Могили
Факультет комп'ютерних наук
Кафедра інженерії програмного забезпечення

ЗАТВЕРДЖУЮ

Зав. кафедри канд. техн. наук, доцент

_____ Є. О. Давиденко

« __ » _____ 2024 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи бакалавра

Видано здобувачу групи 408 факультету комп'ютерних наук

_____Кубицькому Микиті Сергійовичу_____
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи

_____Ігровий застосунок в жанрі Roguelike на основі рушія Unity_____

Затверджена наказом по ЧНУ від «22» грудня 2023 р. № 269

2. Строк представлення кваліфікаційної роботи «____» _____ 20__ р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні

_____Очікуваним результатом є ігровий застосунок у жанрі Roguelike на основі рушія Unity2D_____

4. Перелік питань, що підлягають розробці

_____Аналіз предметної галузі, аналіз системи PathFinder для ін'єкції механік до ігрового застосунку, виконати проєктування ігрового застосунку, реалізувати ігровий застосунок, протестувати розроблений ігровий застосунок_____

5. Перелік графічних матеріалів

_____Презентація_____

6. Завдання до спеціальної частини

_____Дослідження питань охорони праці, які безпосередньо пов'язані з діяльністю розробника програмного забезпечення_____

7. Консультанти:

Консультант	Кафедра (організація)	Частина роботи
Алексєєва А.О.	Кафедра екології Медичного інституту ЧНУ ім. Петра Могили	Спеціальна частина з охорони праці

Керівник роботи ___PhD, ст. викладач Антіпова Катерина Олександрівна___
(посада, прізвище, ім'я, по батькові)

_____
(підпис)

Завдання прийнято до виконання

_____Кубицькому Микиті Сергійовичу_____
(прізвище, ім'я, по батькові здобувача)

_____
(підпис)

Дата видачі завдання «22» _грудня_ 2024 р.

КАЛЕНДАРНИЙ ПЛАН

виконання кваліфікаційної роботи

Тема: Ігровий застосунок в жанрі Roguelike на основі рушія Unity

№	Найменування роботи	Початок	Закінчення	Примітки
1.	Розробка та затвердження завдання на виконання КРБ	21.12.23	22.12.23	виконано
2.	Огляд літератури за темою роботи	24.12.23	28.12.23	виконано
3.	Складання календарного плану КРБ	09.01.24	10.01.24	виконано
4.	Аналіз предметної області	10.01.24	12.01.24	виконано
5.	Розробка проєктних рішень	13.01.24	15.01.24	виконано
6.	Моделювання та конструювання ПЗ	16.01.24	19.01.24	виконано
7.	Кодування, тестування та апробація розробленого ПЗ, аналіз результатів тестування, розробка керівництва користувача	20.01.24	20.03.24	виконано
8.	Розробка спеціальної частини з охорони праці	21.03.24	31.03.24	виконано
9.	Відгук керівника КРБ	01.04.24	03.04.24	виконано
10.	Оформлення КРБ та презентації	04.04.24	21.04.24	виконано
11.	Попередній захист	03.06.24	05.06.24	виконано
12.	Рецензування	12.06.24	13.06.24	виконано
13.	Завершення оформлення КРБ та презентації	16.06.24	17.06.24	виконано
14.	Захист кваліфікаційної роботи			

Розробив здобувач Кубицький Микита Сергійович _____
(прізвище, ім'я, по батькові) *(підпис)*

«09» січня 2024 р.

Керівник роботи PhD, ст. викладач Антіпова Катерина Олександрівна _____
(посада, прізвище, ім'я, по батькові) *(підпис)*

«__» _____ 2024 р.

АНОТАЦІЯ

до кваліфікаційної роботи бакалавра

«Ігровий застосунок в жанрі Roguelike на основі рушія Unity»

Здобувач 408 гр.: Кубицький Микита Сергійович

Керівник: PhD, ст. викладач Антіпова Катерина Олександрівна

Актуальність обраної теми полягає в тому, що на теперішній час ігрова індустрія переживає певну стагнацію через те, що ринок ігрових застосунків переповнений перевиданнями, ремастерами та продовженнями вже існуючих франшиз. І це при тому, що велика кількість цих ігрових застосунків виходять в бета-версії до завершення поліровки ігрового процесу. На ринку зараз багато однотипних ігрових застосунків із подібними жанрами та однотипними всесвітами, тому цікавість до таких ігор все падає. Такі ігри як онлайн шутери, кібер-спортивні змагальні ігри, ММО, ще викликають хоч якийсь інтерес, але й ці жанри швидко набридають гравцям, тому все більше людей приходять до висновку повернутись до витоків цих франшиз.

Об’єктом кваліфікаційної роботи є процес розробки ігрового застосунку у жанрі Roguelike на базі рушія Unity 2D.

Предметом кваліфікаційної роботи є інструментарій для розробки ігрового застосунку у жанрі Roguelike на базі рушія Unity 2D.

Метою кваліфікаційної роботи є розробка ігрового застосунку у жанрі Roguelike на базі Unity 2D для популяризації системи PathFinder та всесвіту DarkSun.

Для досягнення мети треба виконати наступні завдання:

– виконати аналіз ігрової сфери;
– виконати порівняльний аналіз аналогів;
– сформулювати вимоги до застосунку, що розробляється;
– виконати проєктування та моделювання ігрового застосунку;
– реалізувати ігровий застосунок;
– протестувати розроблений ігровий застосунок.

У першому розділі описується огляд ігрової сфери в жанрі Roguelike, всесвіту DarkSun та системи PathFinder. Виконано порівняльний аналіз існуючих аналогів та сформулювано вимоги до ігрового застосунку, що розробляється.

У другому розділі описується процес моделювання застосунку із використанням UML-діаграм, наведено опис середовища розробки та особливостей ігрового двигуну Unity 2D.

У третьому розділі описується процес розробки ігрового застосунку, а саме розробка UI елементів, дизайн рівнів, програмування функціоналу об’єктів, робота зі звуком та текстурами.

У четвертому розділі описується тестування ігрового застосунку, рефакторинг коду, застосування паттернів.

У висновках описується аналіз отриманих результатів створеного застосунку.

КРБ викладена на 65 сторінки, вона містить 4 розділи, 39 ілюстрацій, 3 таблиці, 20 джерел в переліку посилань.

ABSTRACT

of the Bachelor’s Thesis

"Roguelike game application based on the Unity engine"

Student of group 408: Kubytskyi Mykyta Serhiyovych

Supervisor: PhD, Senior Lecturer Kateryna Oleksandrivna Antipova

The relevance of the chosen topic lies in the fact that the gaming industry is currently experiencing a certain stagnation due to the fact that the gaming app market is overflowing with re-releases, remasters and sequels to existing franchises. And this is despite the fact that a large number of these gaming apps are released in beta before the gameplay is polished. There are many similar gaming apps with similar genres and similar universes on the market, so the interest in such games is decreasing. Games such as online shooters, competitive e-sports games, MMOs still generate at least some interest, but these genres quickly get boring for players, so more and more people are coming to the conclusion to return to the origins of these franchises.

The object of the qualification work is a Roguelike game application based on the Unity 2D engine.

The subject of the qualification work is a toolkit for developing a Roguelike game application based on the Unity 2D engine.

The purpose of the qualification work is to develop a Roguelike game application based on Unity 2D to promote the PathFinder system and the DarkSun universe.

To achieve the goal, the following tasks should be completed:

– analyze the gaming industry;
– perform a comparative analysis of analogues;
– formulate requirements for the application to be developed;
– design and model the game application;
– implement the gaming application;
– test the developed gaming application.

The first section describes an overview of the Roguelike gaming industry, the DarkSun universe and the PathFinder system. A comparative analysis of existing

analogues is performed and requirements for the game application under development are formulated.

The second section describes the process of modelling the application using UML diagrams, provides a description of the development environment and features of the Unity 2D game engine.

The third section describes the process of developing a game application, including the development of UI elements, level design, programming of objects functionality, work with sound and textures.

The fourth section describes testing of a game application, code refactoring, and the use of patterns.

The conclusions analyze the work and the results obtained.

The bachelor's qualification work is set out on 65 pages, it contains 4 sections, 39 illustrations, 3 tables, 20 sources in the list of references.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ММО	–	масова багатокористувацька онлайн відеогра
ПЗ	–	програмне забезпечення
ХП	–	здоров’я, яке відображається у вигляді пойнтів
ОС	–	операційна система
ООП	–	об'єктно-орієнтоване програмування
UX	–	user experience
UI	–	user interface
API	–	application programming interface
RPG	–	role-playing game
DLC	–	downloadable content
NPC	–	non-player character

ВСТУП

Актуальність обраної теми полягає в тому, що на теперішній час ігрова індустрія переживає певну стагнацію через те, що ринок ігрових застосунків переповнений перевиданнями, ремастерами та продовженнями вже існуючих франшиз. І це при тому, що велика кількість цих ігрових застосунків виходять в бета-версії до завершення поліровки ігрового процесу. На ринку зараз багато однотипних ігрових застосунків із подібними жанрами та однотипними всесвітами, тому цікавість до таких ігор все падає. Такі ігри як онлайн шутери, кібер-спортивні змагальні ігри, ММО, ще викликають хоч якийсь інтерес, але й ці жанри швидко набридають гравцям, тому все більше людей приходять до висновку повернутись до витоків цих франшиз.

Обраний жанр, а саме Roguelike, на ринку ігрових застосунків зараз не є перенасиченим, тому може стати в нагоді для того, щоб заволікти аудиторію до створюваного застосунку та популяризувати всесвіту DarkSun, але й сам по собі жанр є цікавим. Щодо всесвіту створюваного застосунку, то було обрано всесвіт DarkSun. Всесвіт DarkSun є маловідомим, але наповненим контентом і має свої цікаві сторони для реалізації жанру Roguelike, а на ринку ігрових застосунків всесвіт DarkSun не є розповсюдженим.

Щодо самого ігрового застосунку обрано ігровий двигун Unity на базі 2D. Unity має свої переваги серед аналогів, наприклад Unreal Engine, і був обраний через велику кількість документацій та посібників, які допоможуть реалізувати ігровий застосунок із обраним жанром та всесвітом.

Об’єктом кваліфікаційної роботи є процес розробки ігрового застосунку у жанрі Roguelike на базі рушія Unity 2D.

Предметом кваліфікаційної роботи є інструментарій для розробки ігрового застосунку у жанрі Roguelike на базі рушія Unity 2D.

Метою кваліфікаційної роботи є розробка ігрового застосунку у жанрі Roguelike на базі Unity 2D, для популяризації системи PathFinder та всесвіту DarkSun.

Для досягнення мети треба виконати наступні завдання:

– виконати аналіз ігрової сфери;
– виконати порівняльний аналіз аналогів;
– сформулювати вимоги до застосунку, що розробляється;
– виконати проєктування та моделювання ігрового застосунку;
– реалізувати ігровий застосунок;
– протестувати розроблений ігровий застосунок.

1 АНАЛІЗ ІГРОВОЇ СФЕРИ

1.1 Аналіз предметної області

Ігрові застосунки з самого початку їх заснування були створенні для розваг і з набуттям більшої популярності цього виду розваг, виросла кількість ігрових застосунків та їх прибутковість. Ігрова індустрія почала зростати, ігрові застосунки почали ставати краще, більше, нові жанри, механіки. Із зростанням технічного рівня також зростав апаратний рівень, бо для більш просунутих ігрових застосунків було потрібно більш просунуте залізо.

Із появою нових жанрів ігрові застосунки були задіяні не тільки для розваг, а для навчання, тестування апаратної частини, лікування, розвиток розумових здібностей. Наприклад, жанр симуляторів надавав можливість гравцю симулювати певний процес або подію, через що здобуті навички в ігровому застосунку могли бути задіяні в реальному житті. Жанр головоломок надавав можливість розвивати розумові здібності дітей, або в лікуванні розумових захворювань наприклад, Альцгеймер. Жанр RPG розвивають комунікативні здібності людини, її креативність та акторство. Жанр SandBox із поєднанням жанру симуляторів стане в нагоді для перевірки апаратної частини, бо жанр симулятору передбачає детальне моделювання об’єктів, фізики, якісне текстурування, а жанр SandBox дозволить скористатись усіма створеними ігровими компонентами для перевірки навантаження на апаратуру.

Ігрові застосунки стали невід’ємною частиною бізнесу, ціни на ігрові застосунку стали набагато більше з моменту їх заснування, але ціна обґрунтовується технологіями, що використали підчас розробки та кількості часу витраченого на розробку. В ігрових застосунках створюють підписки заради спеціального контенту, додають рекламу, як в ігровий процес так і в ігрове меню, створюють додатковий платний контент, а саме DLC. Було вигадано багато способів отримати гроші з потенційного гравця, але з часом самі гравці почали

протестувати проти таких способів, починаючи з review bombing'у на всіх торгових площадках та сайтів ігрових критиків та закінчуючи надсиланням гнівних листів до розробників ігрового застосунку.

Ігрові застосунки мають свою частку праці у розробці апаратного забезпечення, бо більшість компаній, що розробляють та продають деякі компоненти до комп'ютера орієнтуються на технологічний процес ігрових застосунків та порівнюють свої компоненти на характеристиках запущених ігрових застосунків, до таких компаній можна віднести гегемонів розробки відеокарт AMD та NVIDIA або розробки процесорів Intel. Але й навіть більше, розробники графічних компонентів створюють власні технології для оптимізації ігрових застосунків або для покращення графіки ігрового застосунку.

Розробка ігрового застосунку включає в себе багато аспектів та рішень, щодо вибору жанру, теми, технологій, всесвіту, написання сюжету, якщо такий буде, створення макетів рівнів, дизайн інтерфейсу, діаграми ігрового застосунку, створення моделей, текстур, рівнів, програмування середовища. Тому всі аспекти поділяють на декілька етапів створення ігрового застосунку:

- проєктування;
- моделювання;
- програмування;
- тестування.

Проєктування ігрового застосунку включає в себе знаходження ідеї ігрового застосунку, а точніше це обирання жанру, тематики, всесвіту, які будуть задіяні технології, який ігровий двигун буде обраний для реалізації ігрового застосунку, створення макетів рівнів, ігрових персонажів, ігрових механік. Це найбільш простий етап створення ігрового застосунку, бо для його виконання не потрібне спеціальне обладнання, потрібно лише мати креативність та мотивацію.

Моделювання ігрового застосунку включає в себе створення mockup'ів UX/UI дизайнів, моделювання проєкту щодо його класів, способів використання,

потрібного обладнання для користування, життєвого циклу ігрового застосунку та іншого, що буде пов’язано з функціоналом самого застосунку, формулювання вимог до ігрового застосунку. Для виконання цього етапу потрібно мати спеціальне програмне забезпечення для створення діаграм та mockup’ів інтерфейсу, також мати знання щодо створення діаграм та їх розуміння.

Програмування ігрового застосунку включає в себе використання функціоналу ігрового двигуна та написання скриптів у середовищі розробки за для використання їх у середині ігрового двигуну та виконання певних дій. Для програмування використовують до цього створені моделі класів та інші діаграми для полегшення роботи. Також під час цього етапу створюються текстури та моделі.

Тестування ігрового застосунку це вже останній етап на котрому проходить тестування ігрового застосунку для подальшого доведення ігрового застосунку до ідеалу. На цьому етапі створюються тест-кейси для оброблювання більшої кількості інформації та більшого обсягу ігрового застосунку, тестування на різних апаратних пристроях, тестування навантаження. І на кінці цього етапу виходить вже готовий продукт для використання.

Для створення ігрового застосунку використовують певний інструментарій, який дозволяє саме створити ігровий застосунок та його розробити. До цього інструментарію відносять ігровий двигун, що є по суті серцевиною ігрового застосунку, середовище розробки, графічний редактор, середовище моделювання об’єктів, середовище для створення діаграм, середовище контролю версій ігрового застосунку та для роботи в групі з іншими розробниками. Ігровий двигун це об’єкт на базі якого створюється ігровий застосунок, ігровий двигун має власну фізику, середовище для створення рівнів та прив’язування скриптів до об’єктів на рівні. Середовище розробки це програма, що допомагає у написанні скриптів на обраній мові ігрового двигуна. Графічний редактор дозволяє створювати текстури для ігрового застосунку. Середовище моделювання об’єктів дозволяє створювати

3D об’єкти для подальшого їх імпорту до ігрового застосунку. Середовище для створення діаграм дозволяє виконати етап моделювання ігрового застосунку. Середовище контролю версій допомагає зберігати поточну версію застосунку та створювати альтернативні їх версії, або відмінити задіяні зміни до ігрового застосунку, також допомагає у роботі в групі розробників.

1.2 Аналіз аналогів

На ринку ігрових застосунків існує досить багато застосунків із жанром roguelike, але можна відокремити певні аналоги, що будуть схожі на створюваний ігровий застосунок. Серед таких аналогів можна відокремити такі ігрові застосунки, як:

- Quest of Dungeons;
- Slay the Spire;
- Roguedice.

Ігровий застосунок Quest of Dungeons[13].

Рисунок 1.1 – Quest of Dungeons

Автор: Девід Амадор.

Архітектура: десктопний застосунок.

Мова реалізації застосунку: створено на власному ігровому двигуну.

Опис застосунку: Quest of Dungeons є покроковим dungeon crawler`ом із елементами roguelike. Суть гри пройти усі рівні обраним класом та не вмерти підчас проходження. Стилістика рівнів схожа на Dungeon and Dragons. У грі є такі класи як воїн, колдун, ловчий та шаман.

Рисунок 1.2 – Ігровий процес

Основні функції гри:

- покрокова система;
- система класів;
- вид зверху;
- рівнева система покращення персонажа;
- рівнева система рівнів.
- система інвентарю.

Переваги застосунку:

- цікава ретро стилістика;

- генерація рівнів;
- обирання класів персонажів.

Недоліки застосунку:

- скудний вибір дій;
- відсутність сюжету.

Ігровий застосунок Slay the Spire[12].

Рисунок 1.3 – Slay the Spire

Автор: Mega Crit Games.

Архітектура: десктопний застосунок.

Мова реалізації застосунку: ігровий двигун libGDX, мова програмування C, C++, Java.

Опис застосунку: Slay the Spire це roguelike, що сфокусований на взаємодію через картки. Перед гравцем відкривається мапа з безліччю шляхів за якими можна завершити гру, на кожному шляху гравця чекає неочікувана зустріч або подія. Під час зустрічі гравець обирає які картки він зіграє під час свого ходу, а

його вибір є обмежений рукою, в якій довільно обираються карти до певної кількості. Під час якоїсь події гравцю дають можливість обрати, який в результаті ефект буде використаний на гравці.

Рисунок 1.4 – Ігровий процес

Основні функції застосунку:

- раундова система;
- вид збоку;
- виборча система рівнів;
- довільна система дій описана картками;
- система покращення за допомогою придбання нових карток;
- вибір персонажів;
- система випадкових зустрічей.

Переваги застосунку:

- багата варіаційність побудування власної колоди;
- генерація шляху.

Недоліки застосунку:

- після програшу колоду гравця потрібно будувати заново.

Ігровий застосунок Roguedice[11].

Рисунок 1.5 – Roguedice

Автор: авторами цього застосунку є uritj, SeguerGameDev, larive, enricgamedev, gerarddiazvidal, Komiku.

Архітектура: десктопний застосунок.

Мова реалізації застосунку: створено на власному ігровому двигуну.

Опис застосунку: Roguedice це roguelike пов’язаний на рандомізації дій, що можуть бути виконані гравцем, а фактор рандомізації виконується за допомогою 3D графічного куба. Гра має мапу зі шляхами, які ведуть до одного кінця, тобто кінця гри, на кожному шляху гравця буде чекати певна подія чи зустріч на якій гравцю треба робити вибір. Бій проходить за допомогою рандомізації дій гравця. Для покращення персонажа купляються навики, котрі потім застосовуються до однієї зі сторін куба, також із покращень є додавання до пулу кубів додаткових кубів типу 1d2, 1d3, 1d4 або 1d6.

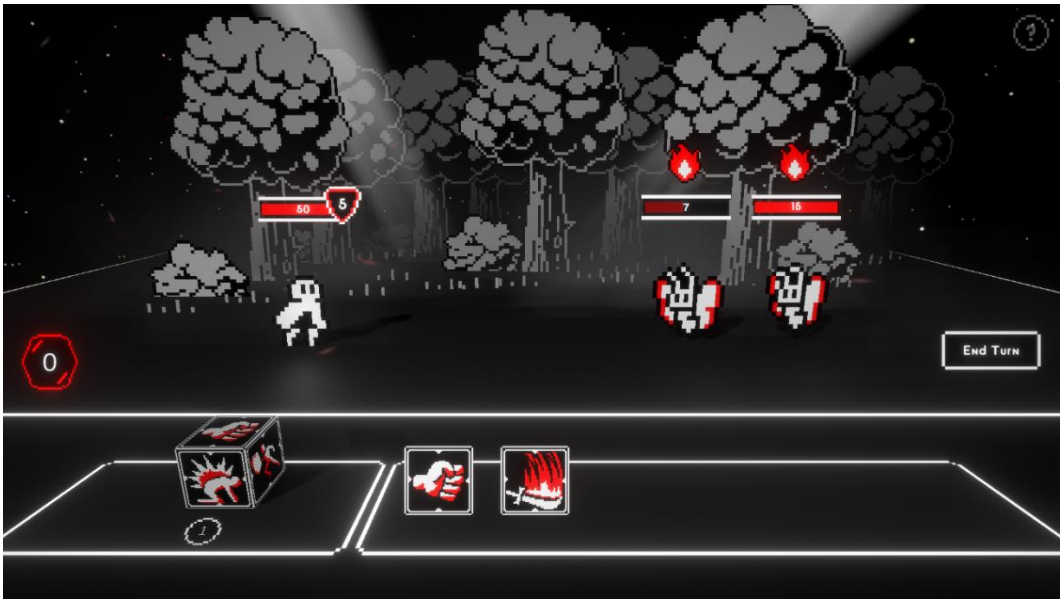

Рисунок 1.6 – Ігровий процес

Основні функції застосунку:

- раундова система;
- вид збоку;
- виборча система рівнів;
- система випадкових зустрічей;
- система покращення за допомогою придбання нових кубів та скілів;
- система бою із використанням рандомізації, а точніше графічних кубів.

Переваги застосунку:

- рандомізація дій дбає про те, щоб кожен бій був різним;
- свій похмурий стиль у темних тонах.

Недоліки застосунку:

- немає сюжету;
- початок гри один й той самий.

1.3 Специфікація вимог до застосунку

ПРИЗНАЧЕННЯ ТА МЕЖІ ПРОЄКТУ

Призначення системи (застосунку), для якої розробляється програмне забезпечення

Призначення застосунку є популяризація всесвіту DarkSun для звернення уваги до цікавого та наповненого контентом всесвіту.

Погодження, що ухвалені в програмній документації

Було погоджено, що для створення ігрового застосунку буде використано допоміжні ассети та бібліотеки Unity.

Межі проєкту ПЗ

Крайня дата завершення роботи над ПЗ – 15.06.2024 року.

ЗАГАЛЬНИЙ ОПИС

Сфера застосування

Ігровий застосунок може бути використаний задля розваг або кіберспорту, щодо проходження гри на швидкість.

Характеристики користувачів

Основна характеристика користувачів є: наявність ноутбуку або ПК.

Загальна структура і склад системи

Основні частини програмного забезпечення: застосунок.

Загальні обмеження

Вимоги відсутні.

ФУНКЦІЇ СИСТЕМИ ІГРОВОГО ЗАСТОСУНКУ В ЖАНРІ ROGULIKE НА ОСНОВІ РУШІЯ UNITY

Функція карткового бою

Опис функції

Функція карткового бою додає інтересу до складання власної колоди та використання стратегічного мислення щодо своїх ходів.

Вхідна і вихідна інформація

Відсутня.

Функціональні вимоги

Гравець повинен мати власну колоду

Функція вибору шляху при переходжені на новий рівень

Опис функції

Функція вибору шляху є основою жанру Roguelike, і додає гравцю можливість перепроходити гру різними шляхами та рівнями.

Вхідна і вихідна інформація

Відсутня.

Функціональні вимоги

Система випадкових рівнів

Функція нового початку

Опис функції

Функція нового початку не завершає гру при поразці гравця, а дає йому шанс завершити гру, але гра поскладнюється, що також додає певний інтерес та челендж.

Вхідна і вихідна інформація

Відсутня.

Функціональні вимоги

Гра поскладнюється із поразкою, гравець повертається в початок.

ВИМОГИ ДО ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ

Джерела і зміст вхідної інформації (даних)

Основним джерелом вхідної інформації ігрового застосунку є користувач. Змістом інформації є дані налаштувань, збережень.

Нормативно-довідкова інформація (класифікатори, довідники тощо)

Вимоги відсутні.

Вимоги до способів організації, збереження та ведення інформації

Збереження даних відбувається через файл, інформація збережена у вигляді JSON.

ВИМОГИ ДО ТЕХНІЧНОГО ЗАБЕЗПЕЧЕННЯ

Вимоги до технічного забезпечення не є великими, бо в ігровому застосунку не використовуються великого формату моделі або текстури, тому технічні дані повинні бути близькими до наведених аналогів.

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Архітектура програмної системи

Архітектура складається з клієнтської частини.

Системне програмне забезпечення

Гра розроблена на ігровому двигуні Unity 2D з використанням мови C#.

Мережне програмне забезпечення

Для створення ігрового застосунку було використано ОС Windows 10.

Програмне забезпечення ведення інформаційної бази

Вимоги відсутні.

Мова і технологія розробки ПЗ

Розробка гри відбувалась на ігровому двигуні Unity з використанням мови C#.

ВИМОГИ ДО ЗОВНІШНІХ ІНТЕРФЕЙСІВ

Інтерфейс користувача

Інтерфейс повинен бути приємним на око, зручним, інтуїтивно зрозумілим, не заважати ігровому процесу гравця.

Апаратний інтерфейс

Апаратним інтерфейсом є ОС Windows 10.

Програмний інтерфейс

У ході розробки було використано дві категорії Unity Scripting API: UnityEditor (Animations, Events тощо) та UnityEngine (Analytics, Audio тощо).

Комунікаційний протокол

Вимоги відсутні.

ВЛАСТИВОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Доступність

Ігровий застосунок є доступним для всіх, за умови якщо у користувача є апаратний інтерфейс.

Супроводжуваність

Не потребує.

Переносимість

Ігровий застосунок сумісний з ОС Windows 10.

Продуктивність

Продуктивність залежить від характеристик ПК чи ноутбуку.

Надійність

Не потребує.

Безпека

Не потребує.

ІНШІ ВИМОГИ

Усі вимоги сформовано.

Висновки до розділу 1

На основі проведеного аналізу ігрової сфери розробки ігрового застосунку на основі Unity 2D та за жанром Roguelike було виявлено, що для розробки ігрового застосунку потрібно мати інструментарій для розробки, розуміння жанру, тематики ігрового застосунку, мати розуміння ігрового двигуну на якому розробляється ігровий застосунок, креативність та творче бачення. Окрім цього також потрібен час та бюджет.

Сам процес створення ігрового застосунку теж є складним де потрібно спочатку спроєктувати ігровий застосунок де потрібно повністю залучити створені ідеї, далі змоделювати ігровий застосунок де потрібно залучити своє знання діаграм та креативність до їх розроблення, після цього програмування застосунку до напівготового продукту та під кінець завершити все тестування де з напівготового продукту вийде готовий продукт.

За результатом аналізу можна запевнитись у тому, що ігрова індустрія з часом стає краще, адаптується та розвивається. Появляються нові технології, які з самого початку були розроблені для використання в грі, але з розвитком ці технології перейшли до звичайного життя і до таких технологій можна віднести штучний інтелект, який спочатку був розроблений для візуалізації життя NPC. Навіть переглядаючи ігрові аналоги, що були наведені в розділі, можна спостерігати їх розвиток. Тому ігрова індустрія не зможе вимерти в житті людей або на ринку бізнесу, бо вона приносить нові технології, гроші та емоції.

2 МОДЕЛЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ІГРОВОГО ЗАСТОСУНКУ

2.1 Діаграма використання

Діаграма використання є гарним способом показати у простому вигляді, які користувачі можуть виконувати певні дії та які дії можна здійснити в певному середовищі. Основними цілями діаграми є:

- визначити загальні межі і контекст модельованого застосунку на початкових етапах розробки;
- сформулювати загальні вимоги до застосунку;
- розробити концептуальну систему для її подальшої реалізації у вигляді фізичних та логічних моделей;
- підготувати документацію для взаємодії між розробниками та замовниками.

Складається діаграма з акторів, тобто зовнішні сутності, та варіанти використання.

Актор – будь-яка роль, що є сутністю яка взаємодіє з варіантами використання. Актор на діаграмі представлений у вигляді чоловічка, що окремо стоїть від варіантів використання.

Варіант використання – дія або сукупність дій, які актор може виконати. На діаграмі варіант використання виглядає, як овал. Відношення між актором та варіантом використання зображено деякими типами ліній.

Діаграма використання в цілому повинна визначати всі можливі варіанти використання, але діаграма може бути розділена на декілька частин, такі частини називаються пакетами. Застосування таких діаграм на всіх етапах роботи над проектом дозволяє уніфікувати позначення для подання підсистемам і системам функціональної цілісності.

При модулюванні ігрового застосунку діаграму використання було розділено на дві частини, де перша частина описує функціональні можливості гравця підчас вільної взаємодії зі сценою, друга частина описує функціональні можливості гравця підчас бою. В усіх діаграмах, як актора представлено саме гравця, бо ігровий застосунок розробляється для одиночного використання без інтернету.

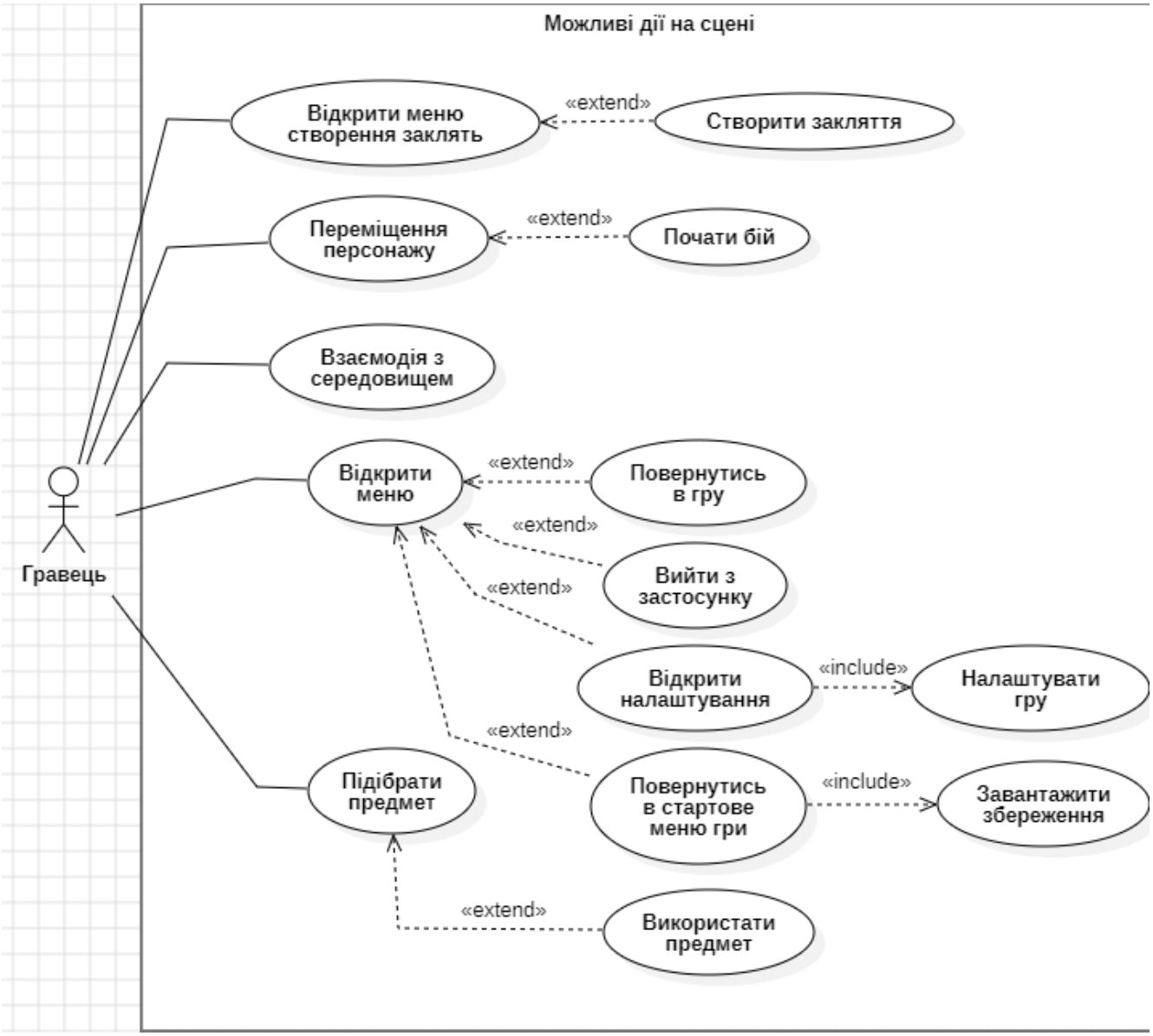

Рисунок 2.1 – Діаграма використання підчас вільної взаємодії

Ця діаграма має одного актора на ім’я Гравець та має тринадцять випадків використання. Дана діаграма описує процес ігрового застосунку, де гравець тільки

но завантажився на рівень. Діаграма має одного актора з ім’ям Гравець, який за діаграмою може виконати певні дії. Наприклад, при переміщенні персонажу, гравець може потрапити в бій, але це необов’язково, тому гравець повинен слідити куди йти. Або при відкритті меню гравець може обрати, що йому робити, а саме чи повернутись в гру або вийти з застосунку або відкрити налаштування або повернутися в стартове меню. Ще можна обрати варіант використання відкриття налаштувань, де гравець після відкриття налаштувань будь-які зміни в налаштуваннях будуть примінені, навіть якщо змін не було зроблено. Підібравши предмет гравець може вирішити його використати або якщо гравцю потрібно створити нове закляття, то він відкриває меню а вже потім його створює. Також гравець має змогу перейти до головного меню та завантажити збереження.

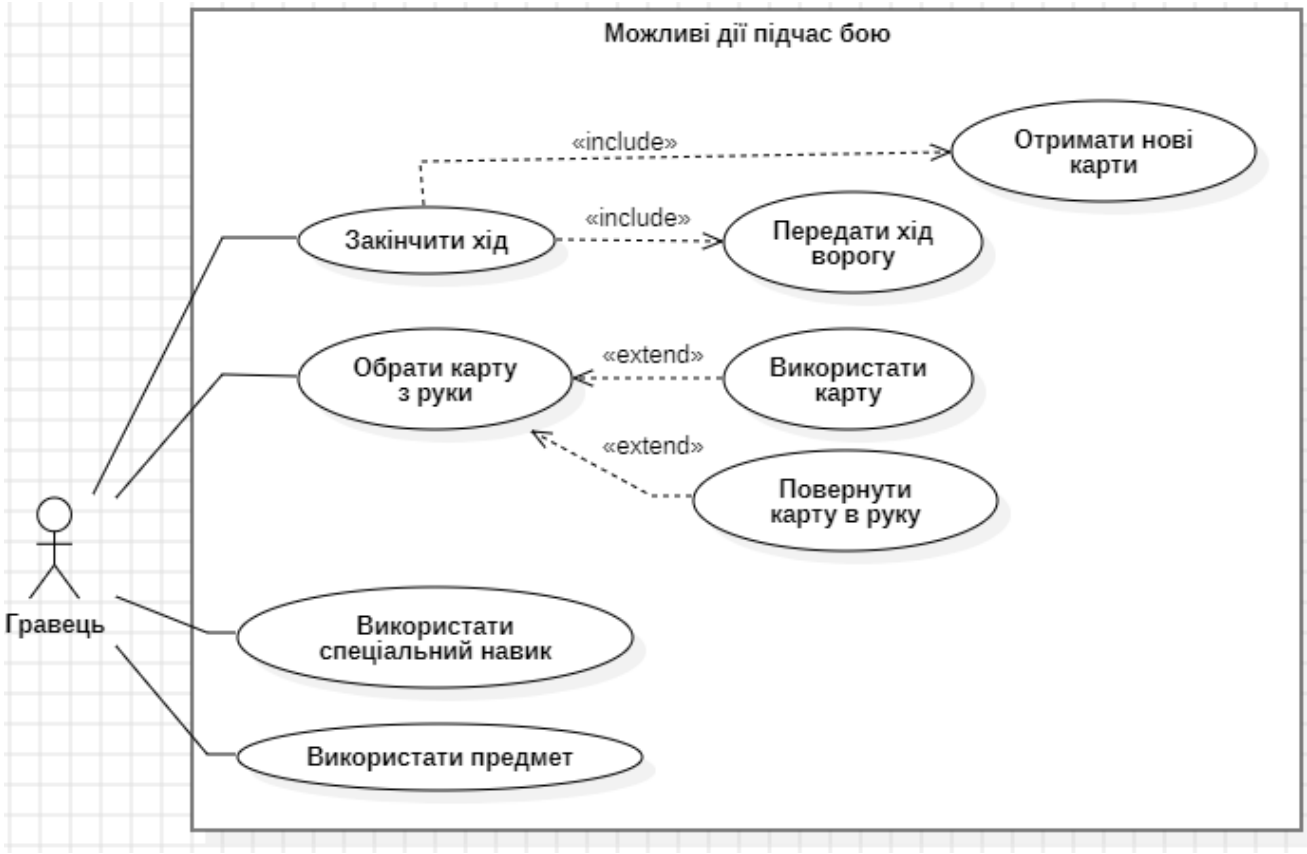

Рисунок 2.2 – Діаграма використання підчас бою

Діаграма має одного актора з ім’ям Гравець та вісім випадків використання. В цій діаграмі гравець розпочинає бій та йому пропонуються такі дії, як:

- обрати карту з руки, її використання або повернення назад до руки;
- використати спеціальний навик;
- використати предмет;
- закінчити хід, чим передасть хід ворогу та отримає нові карти до руки.

До діаграми використання також заведено демонструвати usecase сценарії, бо разом вони спроможні більше детально показати свої основні функції. Сам usecase описується, як перелік дій, тобто сценаріїв, за якими користувач може або буде взаємодіяти із застосунком підчас певної події або цілі. Usecase сценарії бувають трьох типів:

- короткий;
- поверхневий;
- повноцінний.

Короткий usecase сценарій

Гравець заходить на новий рівень. Зустрічає противника. Починається бій. Гравець обирає карту, обирає ціль. Ефект карти наносить шкоду ворогу. Ворог повержений. Бій закінчується.

Поверхневий usecase сценарій

Гравець перемагає ворога підбирає корисні предмети, що випали з ворога, взаємодіє з середовищем, отримує смертельну шкоду від середовища, гра переходить до першого рівня, інвентар гравця анулюється, вороги посилюються, певна частка карт з колоди забувається.

Таблиця 2.1 – Usecase повноцінний

Primary Actor	Гравець
Scope	Ігровий застосунок
Level	Мета користувача
Preconditions	Гравець розпочав бій
Stakeholders and	1. Гравець: виграти бій

interests	
Success guarantee	Гравець перемагає всіх ворогів підчас бою.

Продовження таблиці 2.1

Main Success Scenario	1. Настає хід гравця; 2. Гра генерує доступну кількість карт з колоди гравця до руки гравця; 3. Гравець обирає карту; 4. Гравець обирає ціль ворога; 5. Карта спрацьовує на позначеній цілі; 6. Гра зменшує кількість можливих зіграних карт в хід на число, що зазначено на карті; 7. Ворог помирає від нанесеної шкоди; 8. Бій закінчено.
Extensions	№1 ворог не помирає після ходу гравця Гравець не зміг нанести достатньо шкоди для того, щоб перемогти ворога і тому хід переходить до ворога. №2 ворог перемагає в битві Ворог встигає нанести достатню кількість шкоди підчас свого ходу, щоб перемогти гравця. №3 на руці не має карт для нанесення шкоди ворогу Гравець витратив усі карти в своїй колоді і йому потрібно перемішати колоду №4 на руці немає карт, що можуть нанести шкоду ворогу Гравець зіграв усі свої карти, що можуть нанести шкоду ворогу і в нього на руці все ще є карти.
Special Requirements	Мана, ХП, розпочатий бій

Technology and Data Variations List	Гравцю є потреба мати достатній рівень мани, ХП, карт в колоді.

Кінець таблиці 2.1

Frequency of Occurrence	Підчас переходу на новий рівень та бродіння по рівню.

Наведений вище usecase демонструє ситуацію, де гравець підчас того, як бродив по рівню або переходив на наступний рівень потрапив у бій. Він демонструє вірний шлях підчас цієї події та альтернативні шляхи, які можуть бути допущені підчас розробки.

Шляхи демонструють сценарії де покроково виконуються певні дії, що демонструють той чи інший шлях. Також наведені потрібні ресурси для завершення цього сценарію, наприклад достатній рівень мани та ХП.

2.2 Діаграма розгортання

Діаграма розгортання, ця діаграма демонструє платформу на якій існує ігровий застосунок та які потрібні додаткові пристрої чи девайси для роботи з застосунком.

Задача діаграми представити виконання програмних компонентів у реальному часі, а також процесів та об’єктів. Застосовується ця діаграма для уявлення конфігурації і топології програмної системи.

На діаграмі об’єкт позначається вузлом у вигляді куба, але можна окремий вузол зображувати у вигляді картинки цього об’єкту, наприклад якщо девайсом є комп’ютер, то вузол можна позначити картинкою у вигляді комп’ютера.

В середині вузла можуть бути специфіковані характеристики девайсу, його компоненти.

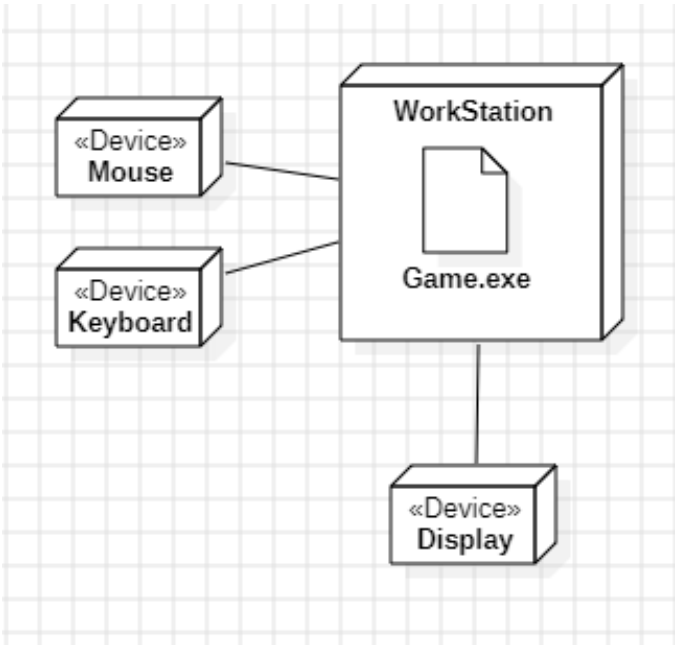

Рисунок 2.3 – Діаграма розгортання

На цій діаграмі зображено чотири вузли та один компонент. Вузли з назвами Mouse, Keyboard, Display слугують девайсами потрібними для застосування ігрового застосунку та пов’язані з вузлом WorkStation, що є сам по собі ПК або ноутбук. Вузол WorkStation має в собі компонент Game.exe, який представляє собою розробляємий ігровий застосунок.

2.3 Діаграма станів

Діаграма станів – це діаграма, що як діаграма використання демонструє певні сценарії поведінки об’єкту в формі послідовності його станів. Діаграма описує реакцію об’єкту на зовнішні події, на виконанні дії об’єктом та на зміни його властивостей. Головне призначення діаграми станів є описати можливу послідовність станів і переходів, що охарактеризують поведінку системи протягом її життєвого циклу. Також діаграма станів може бути застосована для специфікації функціональності певних класів або його альтернатив, а точніше для моделювання всіх можливостей змінення стану конкретних об'єктів.

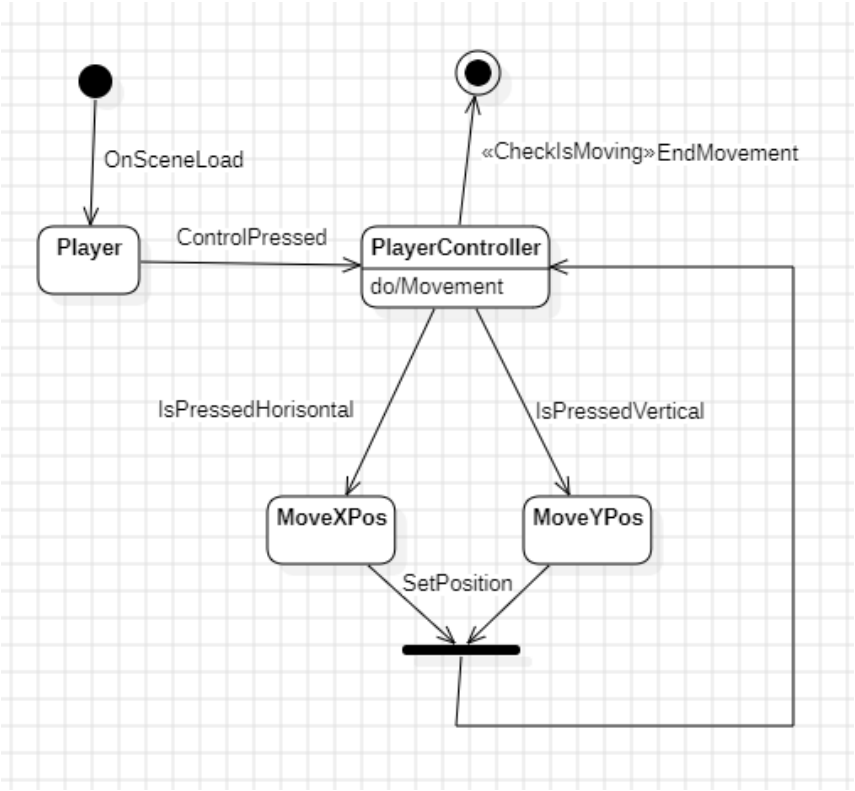

Рисунок 2.4 – Діаграма станів №1

Діаграма має 4 стани, одне поєднання, початок та кінець поведінки. Діаграма вище демонструє поведінку переміщення ігрового персонажа на сцені. На зв’язках між станами описані умови при яких стан переходить до іншого стану. Таким чином, при завантажені сцени коли гравець натискає кнопку для переміщення PlayerController виконує дію руху де визначає, яка саме кнопка була натиснута та по якій осі персонаж буде рухатись. Після отримання даної інформації гравець переходить на отриману позицію та після переходить до PlayerController де перевіряє чи перемістився персонаж, якщо так, то це кінець поведінки та руху персонажу.

Діаграма має свій початок та кінець, між початком і кінцем існує послідовність станів зі своїми зв’язками. Самі стани виглядають у якості прямокутника із власними діями або поведінкою, якщо такі є. Також є розділення та злиття стрілок.

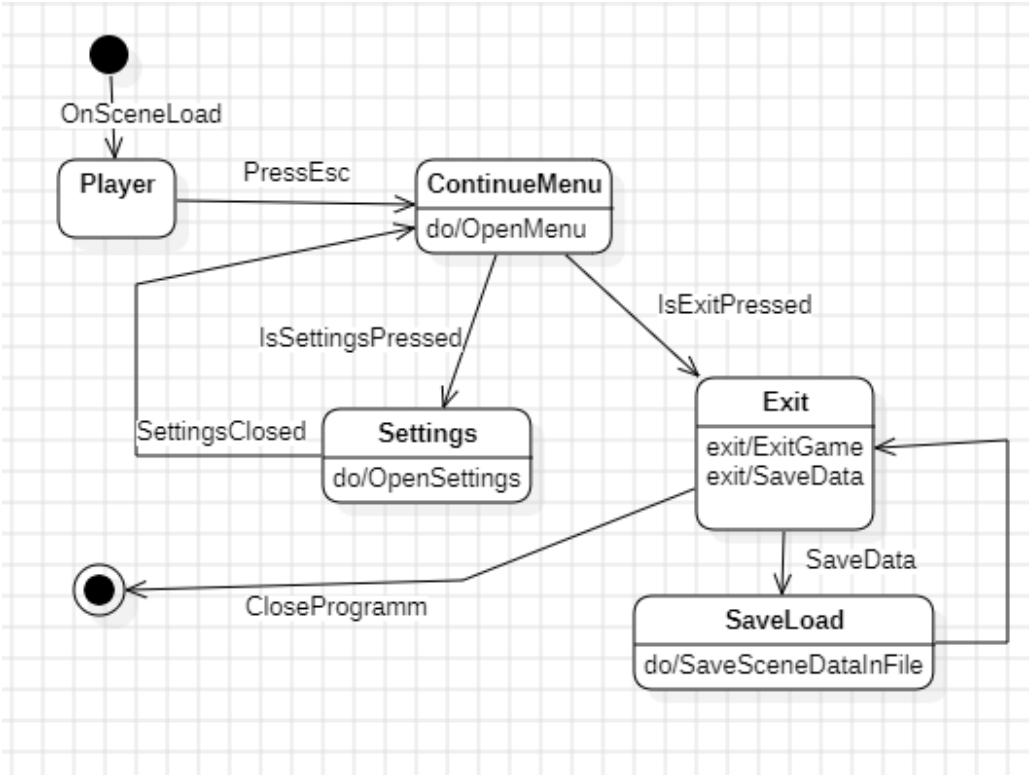

Рисунок 2.5 – Діаграма станів №2

Діаграма має п’ять станів, початок та кінець поведінки. Діаграма вище описує поведінку меню при використанні його гравцем.

Після завантаження сцени гравець натискає кнопку, що відповідає за відкриття меню, якщо гравець заходить до налаштувань, то він може здійснити певні налаштування, які будуть збережені, та повернутись до меню.

Після того як гравець вийшов з налаштувань, гравець обирає вийти з ігрового застосунку, але при виході ігровий застосунок зберігає процес пройдений гравцем і тільки потім закриває сам ігровий застосунок.

2.4 Діаграма компонентів

Діаграма компонентів описує особливості фізичного представлення системи. Діаграма дозволяє визначити архітектуру системи, встановити між компонентами залежності.

Для створення діаграми компонентів залучають програмістів, аналітиків та архітекторів. Певні компоненти існують на етапі компіляції, інші на етапі його виконання.

Компонент це фізична частина системи, що вміщує в собі реалізацію певних класів та їх відносин, але окрім цього ще й функціональну поведінку цих класів.

Компоненти на діаграмі зображені у вигляді папки, або прямокутників, також існує у вигляді документів різного типу. Також на діаграмі можуть бути зображені інтерфейси у вигляді кругів. Компоненти також мають власні зв'язки, які можуть показати їх залежність від інших компонентів.

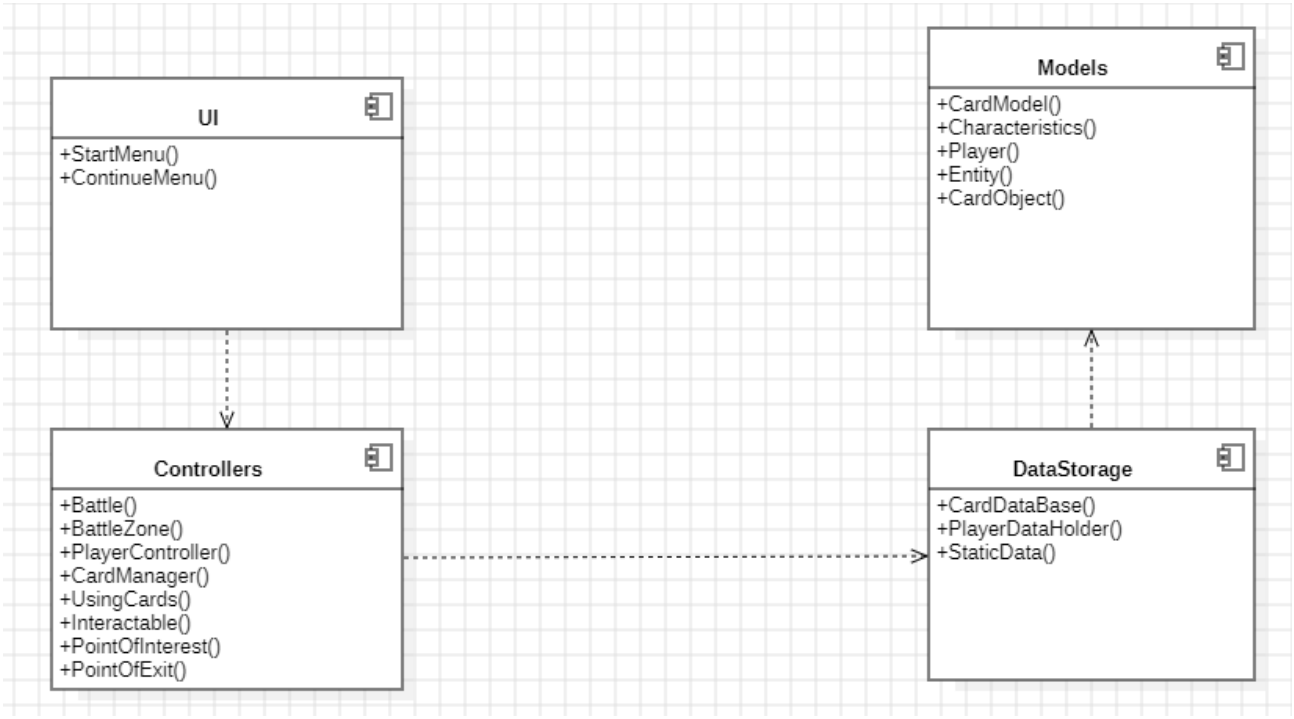

Рисунок 2.6 – Діаграма компонентів

Діаграма вище демонструє чотири компоненти, а саме:

- UI, тобто інтерфейс;
- Controllers, тобто функціонал пов'язаний із взаємодією з середовищем сцени;
- DataStorage, тобто зберігання даних;
- Models, моделі за якими будуються певні елементи.

За зв’язками на діаграмі можна зрозуміти, що незалежними є тільки моделі, але інші компоненти вже залежать один від одного. Властивості, що описані всередині компонентів є класи, які демонструють свою приналежність до певного компоненту.

2.5 Діаграма послідовності

Діаграма послідовності описує взаємодію об’єктів між собою на лінії часу цих компонентів. Діаграма послідовності надає можливість продивитись у який саме час певний об’єкт взаємодіє з іншим та за допомогою цього можна уявити відносини між цими об’єктами.

На діаграмі зображуються об’єкти, які приймають активну взаємодію між іншими об’єктами та саме ключовим аспектом діаграми є її динаміка взаємодії об’єктів на лінії часу.

На діаграмі об’єкти зображуються у вигляді прямокутника, а часовий проміжок йде вертикально вниз та помічає свій час життя у цій взаємодії. На прямокутнику в середині зображується назва об’єкту.

Крайній злів об’єкт по дефолту є ініціатором взаємодії. Саме від нього надсилаються запити на взаємодію з іншими об’єктами на часовому проміжку. На проміжку часу також можна визначити фокус керування часу та повідомлення.

Фокус управління є проміжком часу, який показує дію об’єкту саме у цьому проміжку, з початку до кінця фокусу.

Повідомлення є запитом від одного об’єкту до іншого. Він може бути зворотнім та звичайним.

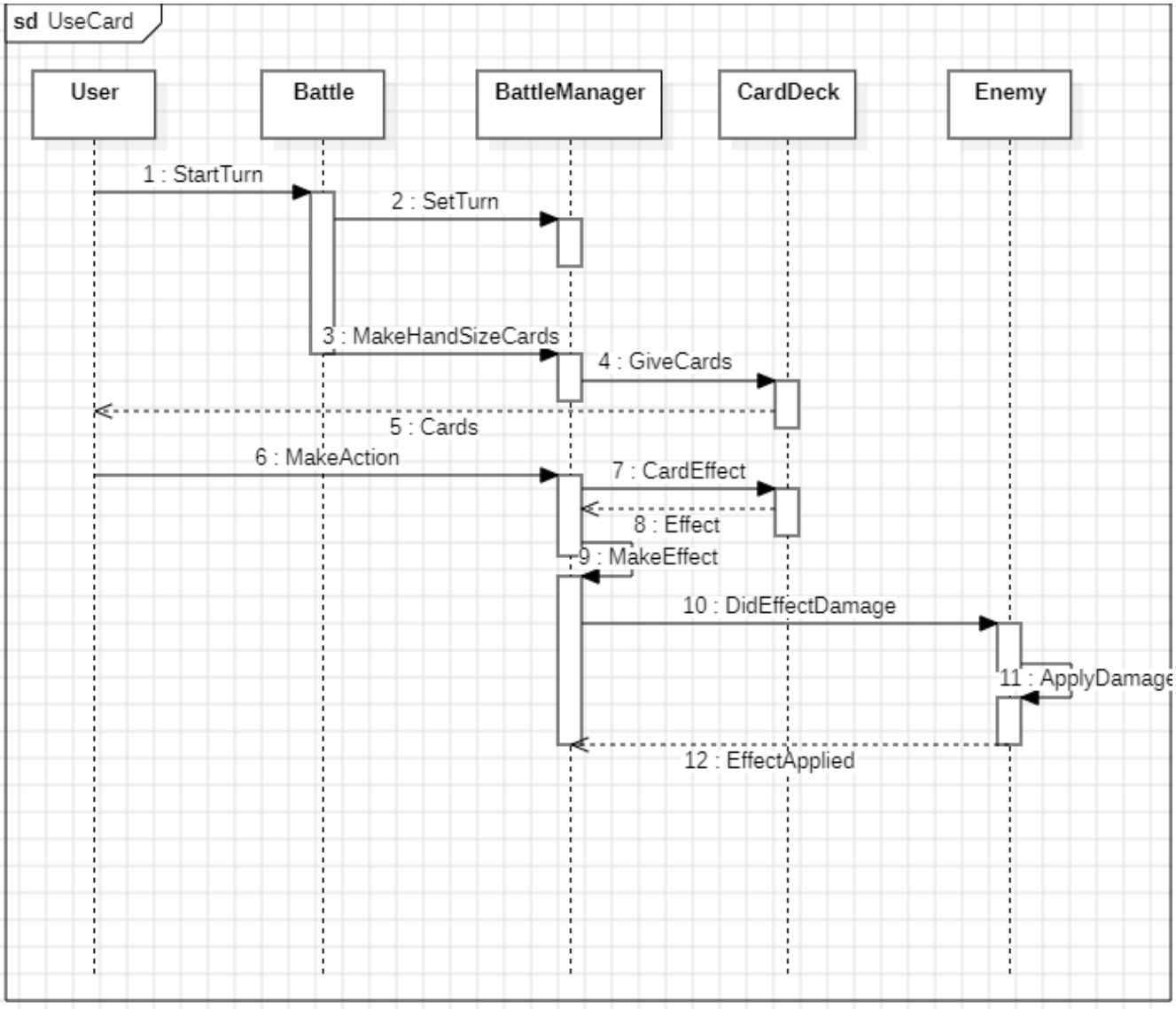

Рисунок 2.7 – Діаграма послідовності

На діаграмі зображено 5 об’єктів, а подією є використання карти. Взаємодія в цій діаграмі починається з початку ходу гравця, де визначається його хід в менеджері. Після визначення ходу гравець робить запит на карти у менеджера де після з колоди видаються карти.

Після отримання карт з колоди, гравець робить дію, а саме використовує карту. Ефект карти спрацьовує та застосовується до ворога. Ворог перевіряє чи отримав він шкоду де після виносить результат до менеджеру.

2.6 Опис ігрового циклу

Опис ігрового циклу є важливим сегментом під час моделювання ігрового застосунку, бо саме в цьому описі затверджується саме шлях ігрового процесу, а точніше його цикл. Опис допомагає розуміти в якому руслі треба розробляти ігровий застосунок та який пріоритет треба надати певній системі при розробці.

Ігровий цикл під час гри має бути приблизно таким:

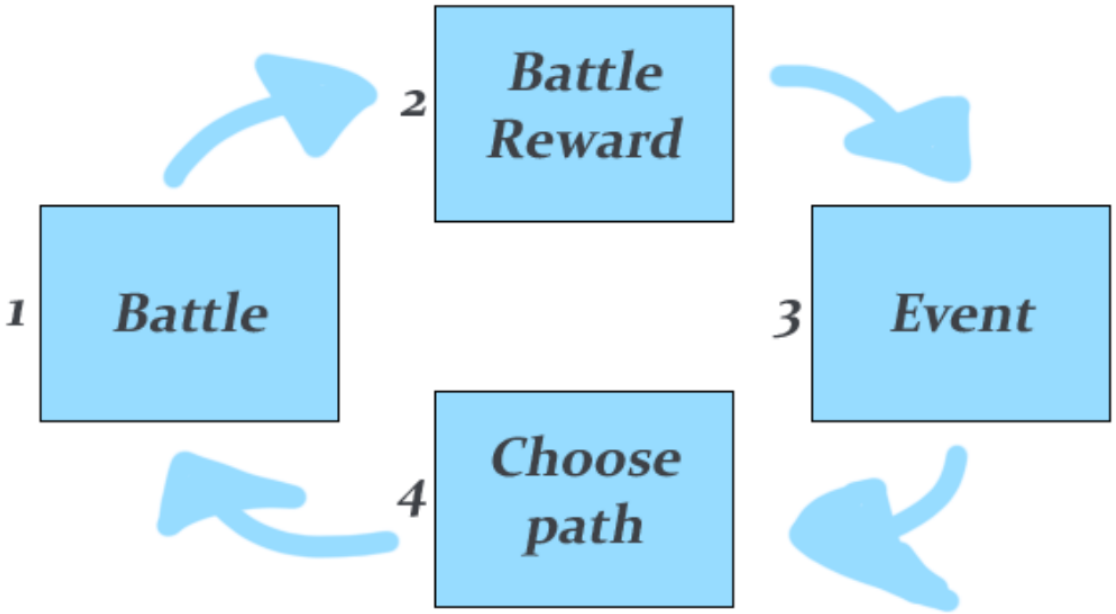

Рисунок 2.8 – Ігровий цикл

За таким циклом гравець на початку кожного рівня:

1) Проходить бій з противником. Тобто розпочинає битву проти чудовиська або групи чудовиськ де група не більше трьох;

2) Отримує нагороди за переможеного противника. Тобто в кінці битви гравець отримує в нагороду предмети чи ефекти від переможеного противника;

3) Зустрічає подію та вирішує її у будь яку з сторін. Тобто після отримання нагород гравець зустрічається з випадковою зустріччю або подією. На зустрічі або події гравець отримує декілька варіантів завершення події чи зустрічі та по завершенню отримує певний ефект;

4) Обирає наступний шлях. Тобто після завершення випадкових зустрічей гравець обирає один з трьох доступних шляхів. Шляхи представлені

собою магічні брами з телепортацією, через які можна подивитись на що чекає гравця.

Висновки до розділу 2

На основі проведеного моделювання над програмним забезпеченням ігрового застосунку було розроблено діаграми використання з наведеними до них сценаріями, usecase сценарії трьох типів на основі дій, які могли статися з актором гравця підчас використання ігрового застосунку. За допомогою цих сценаріїв можна змоделювати подалі етапи розробки та уникнути помилок.

Були розроблені діаграми станів з наведеним початком та кінцем їх поведінки та наведеними станами. Ці діаграми дуже корисні для розуміння поведінки певних частин застосунку.

Були розроблені діаграми компонентів, що зображали компоненти ігрового застосунку та їх складову, а також залежності. Було розроблено діаграми розгортання де були наведені потрібні девайси для використання ігрового застосунку.

Було зроблено опис циклу ігрового процесу де було визначено головні етапи цього процесу та їх складову.

Було зроблено діаграму послідовності, що зображала взаємодію об’єктів на часовому проміжку та їх логіку у часі. Ця діаграма допоможе розібратись з логікою об’єктів та їх взаємодією між ними.

За результатом моделювання можна побачити, як створюється ігровий застосунок та як можна полегшити розробку ігрового застосунку створюючи наперед плани та скоринки, які можна буде використати та не витрачати багато часу на рефакторинг та зміну архітектури.

3 ПРОЄКТУВАННЯ ІГРОВОГО ЗАСТОСУНКУ

3.1 Ігровий процес

Основною частиною ігрового застосунку є бойова система.

Бойова система складається з системи карткового бою яка представлена в наведеному аналогу за назвою «Slay the Spire». Основна роль цієї системи додати варіативність, тактику, різноманітність до гри. В грі до персонажа буде прив'язана колода карт, яку персонаж протягом гри доповнює та змінює. Карти будуть накладати на персонажів перманентні та тимчасові ефекти, наприклад перманентні це нанесення шкоди, а послаблення є тимчасовим. Будуть карти, що спрацьовуватимуть на персонажа гравця, на ворога або декілька ворогів. Також після використання карток або під час ефектів, що накладені на персонажа, ігровий персонаж буде проходити перевірки, які будуть визначити ефективність використаної карти. Сам ворог буде також використовувати карти з колоди, що йому визначена та буде використовувати карти проти персонажа гравця. Після бою гравець отримає нагороду за перемогу над ворогом.

Рисунок 3.1 – Приклад карткового бою з «Slay the Spire»

Дослідження рівня гри та взаємодія на рівні.

За дослідження рівня відповідає система випадкових подій, яка має за мету якось послабити, посилити персонажа гравця або розказати історію. Ці події будуть мати декілька варіантів проходження, і варіант проходження залежить від вибору гравця.

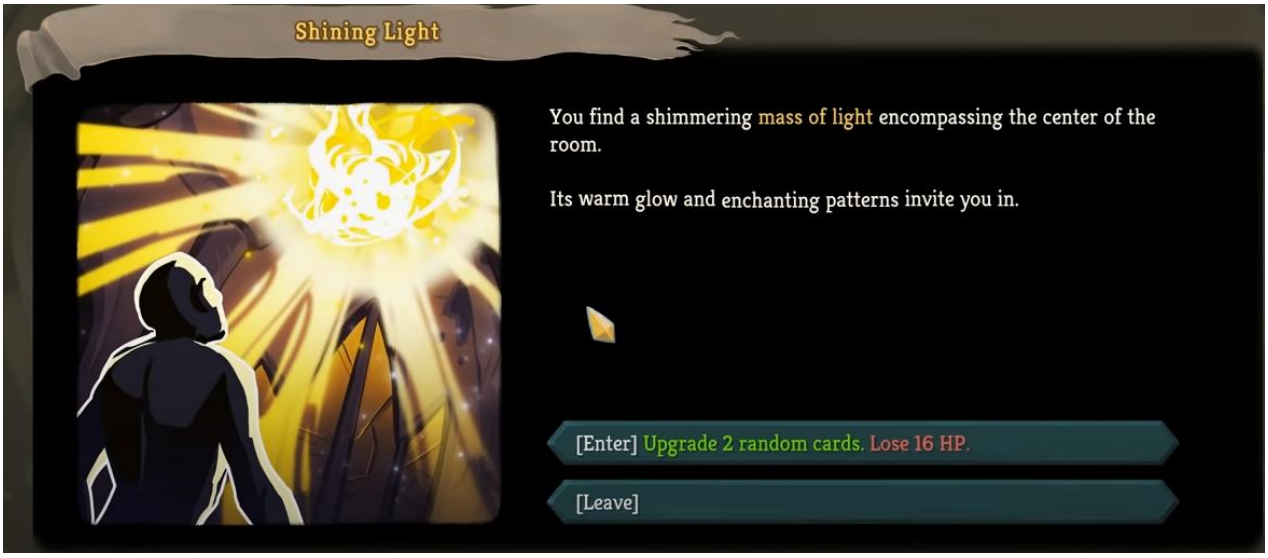

Рисунок 3.2 – Приклад випадкових подій з «Slay the Spire»

Інвентар та крафт предметів.

Застосунок буде мати систему інвентаря персонажа, що буде передбачатись, як контейнер для зберігання перманентних ефектів та предметів. До цих предметів можна віднести сигіли з походженням до сил елементалів, предмети за перемогу над ворогом підчас бою та предмети або ефекти за результат події підчас дослідження. За допомогою сигілів гравець в спеціальному меню зможе створювати нові карти для своєї колоди. До карт будуть створені рецепти, за якими можна отримати карту та складатимуться вони будуть з трьох сигілів.

Рисунок 3.3 – Приклад інвентарю з «Quest of the Dungeons»

Перехід на наступний рівень.

В ігровому застосунку буде передбачено перехід на новий рівень та супроводжуватись це буде вибором гравця. Перед гравцем буде декілька воріт з описом, що відбувається за воротами і гравець може обрати до яких воріт гравець піде. Кожні ворота будуть представляти собою окремий рівень на якому можна буде отримати певні предмети та зустріти певних ворогів. У описі до воріт будуть представлені певні ключові слова, що описуватимуть подальші події на рівні.

Смерть ігрового персонажу.

Після смерті головного персонажу гравця ігровий процес буде скинуто до самого початку та буде розпочато проходження гри з самого початку.

Система збереження.

Ігровий процес буде зберігатись одразу після виходу до головного меню або після виходу з застосунку. Збереження працюють як проходження, тобто після початку нової гри створюється збереження і якщо не буде місця для збереження, то буде видалено саме перше проходження. Завантаження збереження відбувається до моменту виходу з проходження.

Сюжет.

Для створюваної гри буде описано історію ігрового персонажу Queen зі всесвіту DarkSun, що є половинчиком котрий був проклят та став джином, що володіє стихією часу. Одного разу один пройдисвіт загадав йому бажання, щоб той виконав його. Щоб виконати бажання треба було опинитись в іншій частині світу і це зайняло певний час. Після виконання бажання Queen повернувся до пройдисвіта, але на місці вже нічого не було окрім розвалин. Так Queen почав шукати свою загублену лампу.

Минуло тисячу років, через що джин половинчик втратив свою духовну форму та отримав фізичне тіло, яке потім з часом згнило та залишились лише кістки. Queen, джин половинчик став нежиттю, що прикриває свою подобу яскравими одежами. Силами невідомих темпларів, Queen був заключений у магічному підземеллі з якого він тепер прямує вибратись.

3.2 Відповідність до жанру

Ігровий застосунок розробляється у жанрі Roguelike але цей жанр має свої риси, які треба задати для ігрового застосунку, щоб відповідати жанру.

За інтерпретацією Берлінської конференції у 2008 році, жанр roguelike має такі риси[16]:

- Випадкове розміщення середовища. Це випадкове розміщення противників, кімнат, предметів під час гри;
- Перманентна смерть. Це повна смерть із втратою прогресу та починанням гри з самого початку;
- Система ходів. Це по суті покрокові дії під час гри, ця механіка дозволяє обдумувати свої рішення під час ходу;
- Не модальний геймплей. Все відбувається на одній сцені та одразу;
- Комплексність дій, маючи лімітовані ресурси. Це коли гра передбачає декілька рішень для вирішення певної проблеми та вимагає розумного використання предметів через їх лімітованість;

- Ключова задача є перемога над чудовиськами;
- Акцент на дослідження та відкриття. Предмети, події кожен раз можуть бути різними тому треба їх досліджувати.

Якщо виділяти жанр Roguelike за наданими рисами, то розробляємий ігровий застосунок використовує повність описані риси в розробці.

- За випадкове розміщення середовища буде відповідати випадковість ворогів на рівні, подій та вибір наступних рівнів.
- Перманентна смерть буде остаточно використано реалізована в ігровому застосунку.
- Система ходів буде представляти собою ходи під час карткового бою у вигляді ходів між гравцем та противником.
- Не модальний геймплей буде представляти собою усі дії на одній сцені без переходів на інші сцени.
- Комплексність дій буде представляти собою різноманітність дій під час випадкових подій.
- Ключовою задачею буде знищити противників.
- Акцент на дослідження та відкриття буде представляти собою випадкові події та вибір наступних рівнів.

3.3 Архітектура

Розробляємий ігровий застосунок створюється на базі ігрового двигуну Unity. Ігровий двигун має два види розробки, а саме це 2D або 3D. Так, як ігровий застосунок в загальному є картковою грою, то для збереження ресурсів гравця та зручності для реалізації проєкту було обрано саме 2D[17].

Архітектура двигуна пропонує доволі прозорий спосіб створення архітектури гри. Створюваний рівень у проєкті називається сценою, а сама сцена це як контейнер котрий зберігає в собі усі об’єкти на сцені, то може бути персонаж, камера, ігровий фон, інтерфейс тощо. Архітектура двигуну надає у

сцені ієрархію об’єктів, тобто від батька до сина, для більш зручного керування об’єктами на сцені та їх доступу. До кожного об’єкту на сцені можна легко додавати зв’язки компонентів або скриптів. Також двигун надає можливість задавати параметри для змінних у скрипті за допомогою редактору та динамічно змінювати параметри під час тестування проєкту. Також двигун надає доступ до API, який дозволяє змінювати параметри самого об’єкту та його компонентів[10].

Рисунок 3.4 – Unity

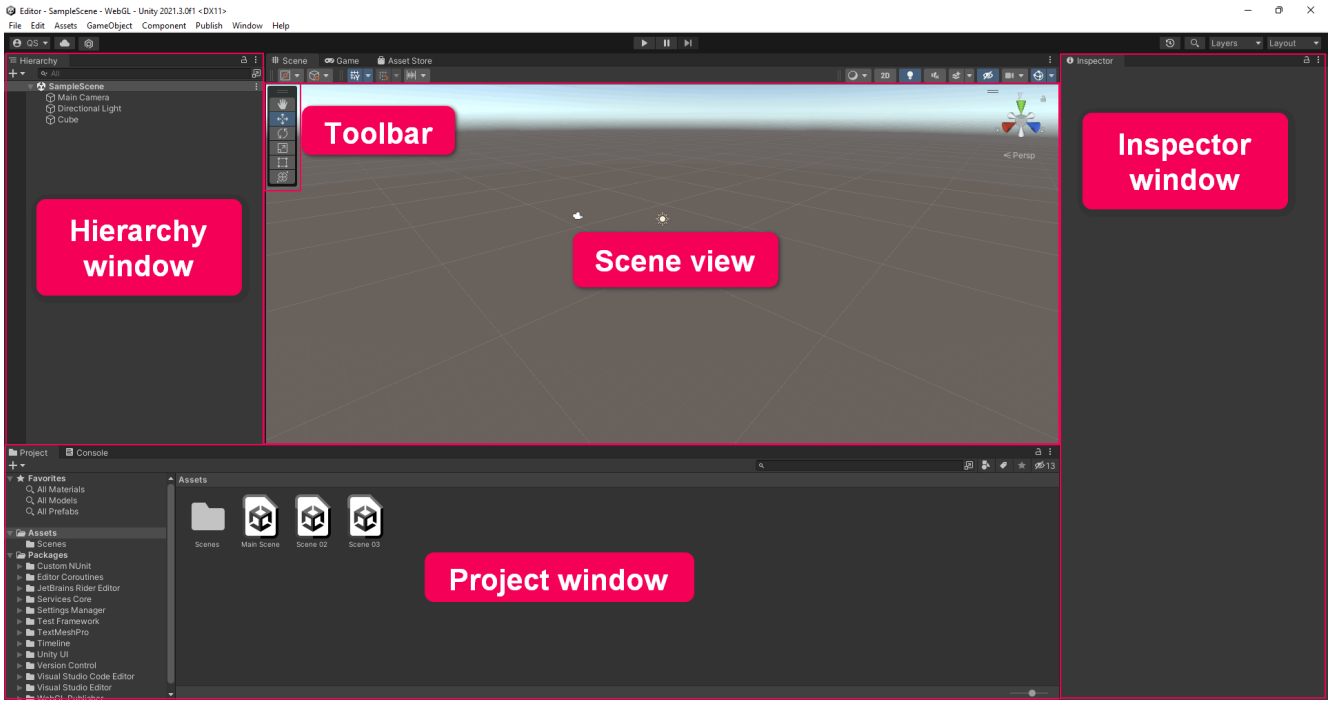

Рисунок 3.5 – Unity Editor

Базовою мовою Unity є C#, але є альтернативи у вигляді C/C++, Rust, IronPython, Lua, Java, JavaScript, SQL, HTML 5, та CSS. Для створення застосунку було обрано саме мову C#. Обрана мова дозволяє отримати більше способів та методів для використання правил ООП, паттернів розробки та методів рефакторингу. Також до такого вибору вплинула велика кількість гайдів розробки ігрового застосунку на мові C#, мультиплатформність ігрового двигуну та великий магазин вже створених ігрових компонентів.

З додаткових компонентів Unity буде використано бібліотеки ScriptableObject для формування екземплярів з вже конструйованої моделі, PixelPerfect для більш кращого відображення піксельних спрайтів.

3.4 Графіка та звук

Розробляємий ігровий застосунок створюється на базі Unity 2D, тому застосунок буде зберігати стиль 2D та використовувати піксельну графіку у стилі PixelArt. Такий стиль зберігає час для розробки ігрового застосунку, ресурси гравця, бо піксельна графіка не є дуже ресурсно-поглинаючою, надає простоти до графіки та гарно сприймається очима.

Рисунок 3.6 – Приклад PixelArt

Сам стиль PixelArt створюється за допомогою малих картинок розмірами 8x8, 16x16, 32x32, 64x64 і так далі. Числами представлено кількість пікселів на розподілену на сторону, чим більше пікселів тим краще та детальніше виходить PixelArt.

Для анімації PixelArt`у копіюється минулий образ та змінюється декілька деталей, після для анімації змінюються образи картинки та саме так анімується PixelArt. Також для більш складних анімацій, малюють декілька окремих частин PixelArt`у та потім складають разом та анімують, наприклад окремо намалювати ноги та після скласти їх у один малюнок та анімувати рух персонажа.

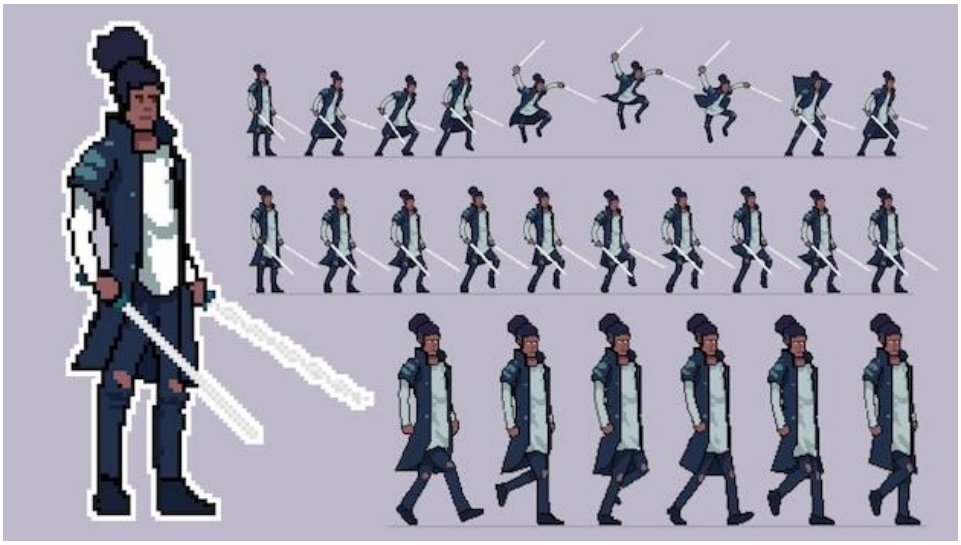

Рисунок 3.7 – Приклад анімації PixelArt

У ігровому застосунку PixelArt буде нагадувати обраний всесвіт DarkSun

Для створення PixelArt`у існує багато різного програмного забезпечення, наприклад Pixilart, Pixie, Pixlr, Canva, LunaPic, Piskel, Adobe Spark, Make Pixel Art, Adobe Photoshop та навіть звичайний Paint.

Для розробляємого ігрового застосунку було обрано саме Adobe Photoshop.

Рисунок 3.8 – Adobe PhotoShop

Він має зрозумілий інтерфейс та велику кількість інструментів, також можна додатково завантажити користувацькі інструменти. Цей графічний редактор має велику кількість справді корисних гайдів, що допоможуть створити не тільки PixelArt графіку, але й розробити інтерфейс для ігрового застосунку.

Звук до ігрового застосунку буде підібраний відповідно до всесвіту DarkSun. До звуків буде відноситись музика на задньому фоні та окремі звуки, що будуть виконуватись під час певних подій. Так як в загальному хід гри буде проходити у підземеллі, то музика повинна надихати гравця атмосферою підземель DarkSun’у.

Музика та звуки для ігрового застосунку будуть знайдені на безкоштовних ресурсах з урахуванням всіх правил щодо авторства та захисту інтелектуальної власності.

3.5 Звук та інтерфейси

Інтерфейс для розробляємого ігрового застосунку повинен бути інформативним, але й мінімальним. Інтерфейс повинен наслідувати й стиль графіки для забезпечення естетики та однорідності стилів.

Інформативність потрібна через те, що імплементація системи PathFinder додає багато параметрів для персонажів, які можуть вплинути під час бою, тому гравцю треба знати, що відбувається з персонажем.

Мінімальність допомагає гравцю концентруватись на певних аспектах гри та не звертати свою увагу на інше, також це зберігає цілісність картинки та мінімалізм.

Для проєктування інтерфейсу створюються так звані make up`и, що презентують образ по якому буде створюватись інтерфейс гри для користувача. Для ігрового застосунку треба створити інтерфейс головного меню, тимчасового меню, інтерфейс битви, інтерфейс випадкових подій, інтерфейс створення нових карток. Для створення make up`у було використано онлайн ресурс.

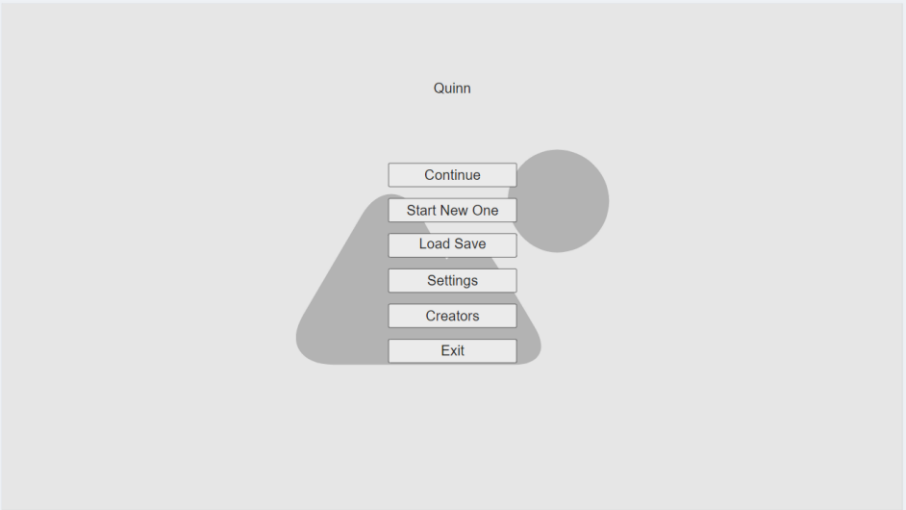

Рисунок 3.9 – Mockup інтерфейс головного меню

На цьому інтерфейсі головне меню. На задньому фоні запланована картинка та назва гри. На інтерфейсі представлені кнопки «Продовжити», «Почати нове проходження», «Завантажити збереження», «Налаштування», «Автори», «Вихід».

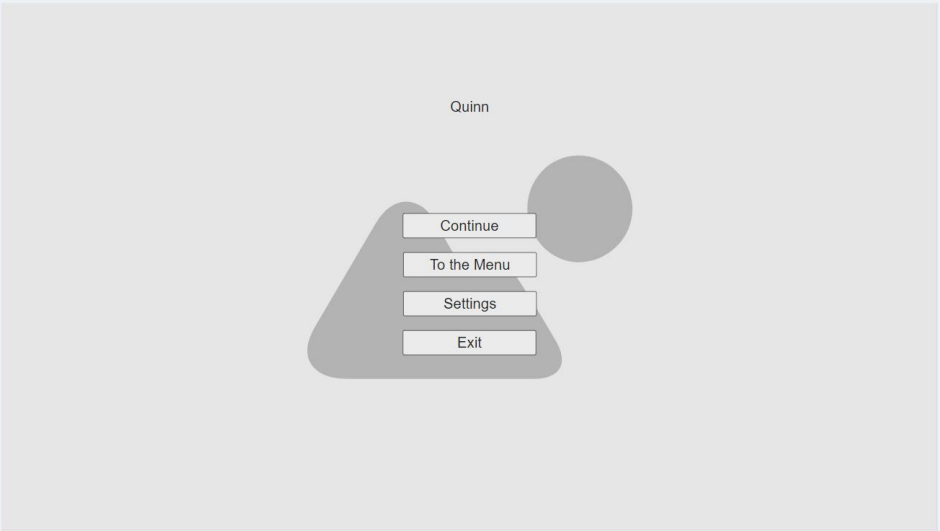

Рисунок 3.10 – Mockup інтерфейс тимчасового меню

На цьому інтерфейсі тимчасове меню. На задньому фоні запланована картинка та назва гри. На інтерфейсі представлені кнопки «Продовжити», «Повернутись до меню», «Налаштування», «Вихід».

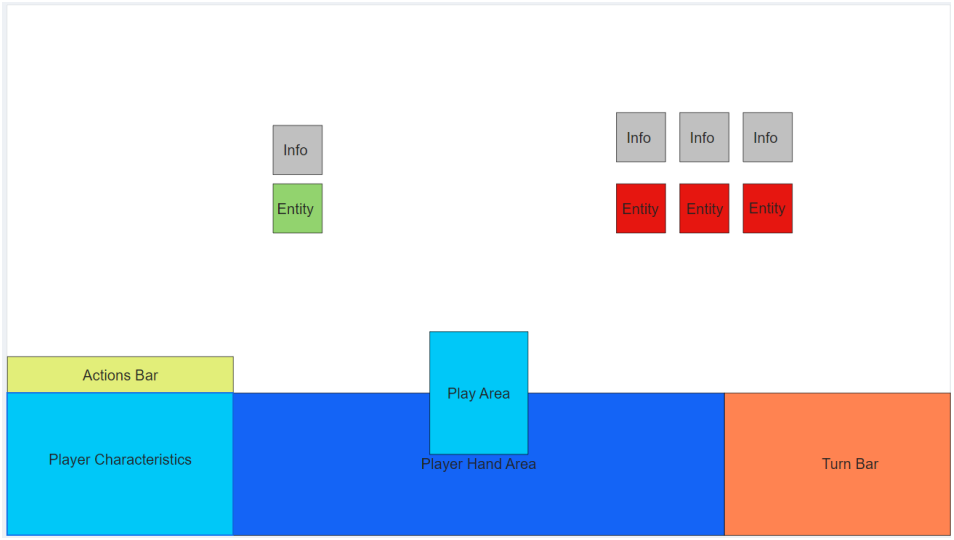

Рисунок 3.11 – Mockup інтерфейс битви

На цьому інтерфейсі битва. На інтерфейсі представлені зони «Стовпчик дій» – кількість можливих дій, «Характеристики персонажа» – характеристики головного персонажа, «Рука для карт» – місце для карт в наявності, «Стовпчик для ходів» – місце для функцій пов’язаних з чергуванням ходів, «Місце для використання карти» – місце де буде показана карта, що використовується,

«Сутність» – місце, що буде представлено головним персонажем або противником, «Інформація» – місце для відображення поточної інформації сутності.

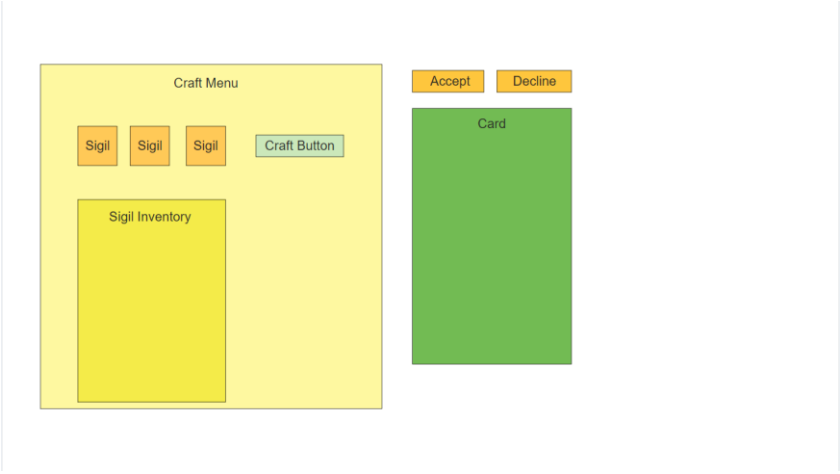

Рисунок 3.12 – Mockup інтерфейс створення нових карток

На цьому інтерфейсі створення нових карток. На інтерфейсі представлені зони «Меню крафту» – меню для крафту нової карти, «Сигіл» – обраний сигіл з інвентарю, «Кнопка для крафту» – кнопка для крафту нової карти, «Інвентар сигілів» – інвентар для зберігання сигілів, «Карта» – місце де буде показана нова карта, «Кнопка Прийняти» – для додання карти до колоди, «Кнопка Не прийняти» – для не додавання карти до колоди.

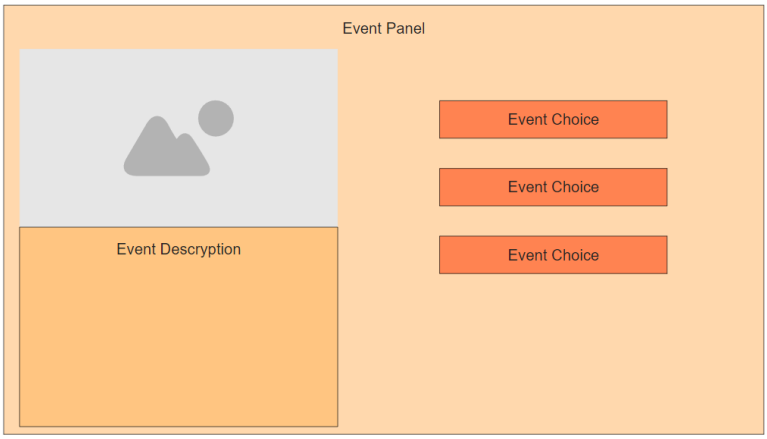

Рисунок 3.13 – Mockup інтерфейс випадкових подій

На цьому інтерфейсі випадкові події. На інтерфейсі представлені зони «Панель події» – панель на якій розташована сама подія, «Картинка» – дає візуальний опис події, «Опис події», «Вибір події».

3.6 Діаграма класів

Діаграма класів демонструє архітектуру застосунку та приблизне бачення цієї архітектури на діаграмі. Основна задача цієї діаграми це демонстрація логічної моделі системи, а точніше демонстрація статичної структури моделі системи в термінології класів ООП. Діаграма класів складається з класів та інтерфейсів, які містять в собі операції та атрибути. На діаграмі класів, класи зображені у вигляді прямокутників, але інтерфейси виглядають у вигляді кола. Параметри та операції в класах та інтерфейсах може бути поміченою символами. Вони визначають їх ступінь доступу у класі чи інтерфейсі або визначити чи є атрибут статичним чи константою. Також методу можна визначити параметри, які він прийматиме.

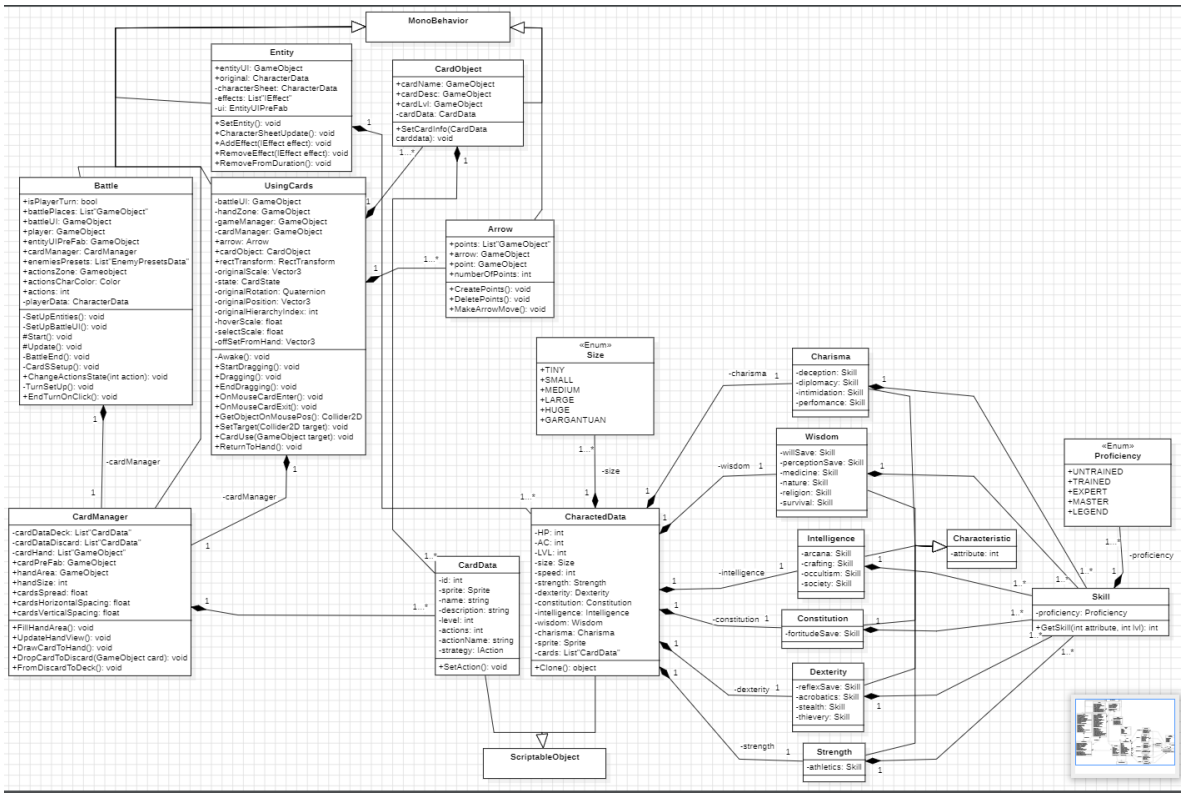

Рисунок 3.14 – Діаграма класів №1

На діаграмі продемонстровано 16 створених класів, 2 батьківські класи ігрового двигуну, 2 нумератори.

Клас CardData є моделлю для зберігання даних карти та використання даних карти під час гри.

Клас CharacterData є моделлю для зберігання даних персонажа та використання його даних під час гри.

Класи Strength, Dexterity, Constitution, Intelligence, Wisdom, Charisma є підкласами для формування класу CharacterData та кожен наслідує клас Characteristic і має свої власні атрибути. Клас Skill є підкласом для формування класів Strength, Dexterity, Constitution, Intelligence, Wisdom та Charisma[3; 4].

Клас CardManager є класом, що контролює руку гравця та її розташування.

Клас UsingCards є класом, що прив’язаний до об’єкту карти та керує подіями над картою, наприклад коли курсор мишки на карті або покинув зону карти, та керує активацією карти, знаходженням цілі, створенням стрілки.

Клас CardObject є класом, що зберігає в собі об’єкти карти для швидкої ініціалізації її параметрів.

Клас Arrow слугує класом для створення стрілки, видалення її та її руху.

Клас Entity є класом, що зберігає в собі дані про персонажа, ефекти, що задіяні на персонажі та слугує ціллю для карт.

Клас Battle є контролером для сцени та корегує всі процеси під час гри на сцені.

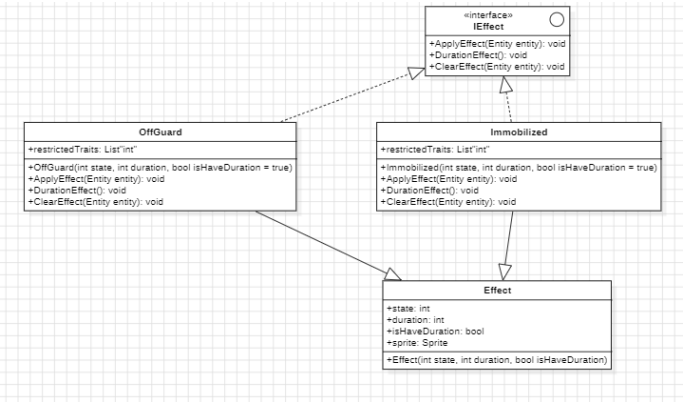

Рисунок 3.15 – Приклад реалізації ефектів

Саме таким чином створюються ефекти, бо кожен ефект є унікальним зі своєю логікою та діями.

Висновки до розділу 3

На основі проведеного проєктування ігрового застосунку було описано ігровий процес застосунку та його функції, було проведено аналіз щодо відповідності ігрового застосунку до обраного жанру, було проаналізовано обрану архітектуру для створення ігрового застосунку та обрано додаткові компоненти для розробки, було визначено як саме буде виглядати графіка та інтерфейс в ігровому застосунку та її стиль, було розроблено п’ять make up’ів для майбутнього інтерфейсу ігрового застосунку, які допоможуть в майбутньому при розробці інтерфейсу, було визначено яка саме музика та ігрові звуки повинні бути у ігровому застосунку.

За результатом проєктування можна побачити, як відбувається створення планів щодо ігрового процесу, графіки, інтерфейсу, звуку до ігрового застосунку та як можна полегшити розробку ігрового застосунку створюючи наперед плани. Також можна побачити, як аналіз архітектури може визначити майбутній ігровий двигун та його переваги в розробці.

4 РОЗРОБКА ТА ТЕСТУВАННЯ ІГРОВОГО ЗАСТОСУНКУ

4.1 Розробка скриптів до функціоналу ігрового застосунку

У ігровому двигуні Unity скриптами називають файли з розширенням .cs і ці самі файли можна використати у якості компонентів до ігрового об’єкту. До ігрового об’єкту можна прив’язати скрипт тільки, якщо ім’я класу та файлу будуть однакові та головний клас буде наслідувати батьківський клас MonoBehavior. Таким чином у проєкті можна тримати два типи скриптів, скрипти що будуть використані у якості компонентів до ігрових об’єктів та скрипти що будуть зберігати в собі класи у якості моделі для зберігання даних[2].

MonoBehavior є батьківським класом для використання функцій ігрового двигуну та всього, що є на сцені. Якщо клас наслідує MonoBehavior, то для нього відкривається багато можливостей, наприклад контролювати рух ігрового об’єкту по сцені та змінювати його розмір підчас проходження певного проміжку. Після наслідування MonoBehavior до класу додаються нові функції, а саме Awake, Start, Update, FixedUpdate і ще інші. Ці методи мають на меті діяти в певний час роботи ігрового застосунку, наприклад Awake та Start спрацьовують на запуску ігрового стосунку, тільки ось Awake спрацьовує до запуску скриптів, а Start після.

При додаванні скрипту в якості компоненту до ігрового об’єкту, в скрипту з’являються нові атрибути, а саме gameobject, transform тощо. Нові атрибути прив’язані до ігрового об’єкту, що має цей скрипт у вигляді компоненту. За допомогою цих атрибутів у середині скрипту можна впливати на ігровий об’єкт, до якого прив’язаний цей скрипт, без його ініціалізації у середині скрипту.

При створенні ігрового застосунку головною його частиною є бойова система. Для реалізації карткового бою було вирішено створити його на основі UI елементів. Для цього на сцені був створений canvas та додані елементи інтерфейсу показані на минулих mockup`ах.

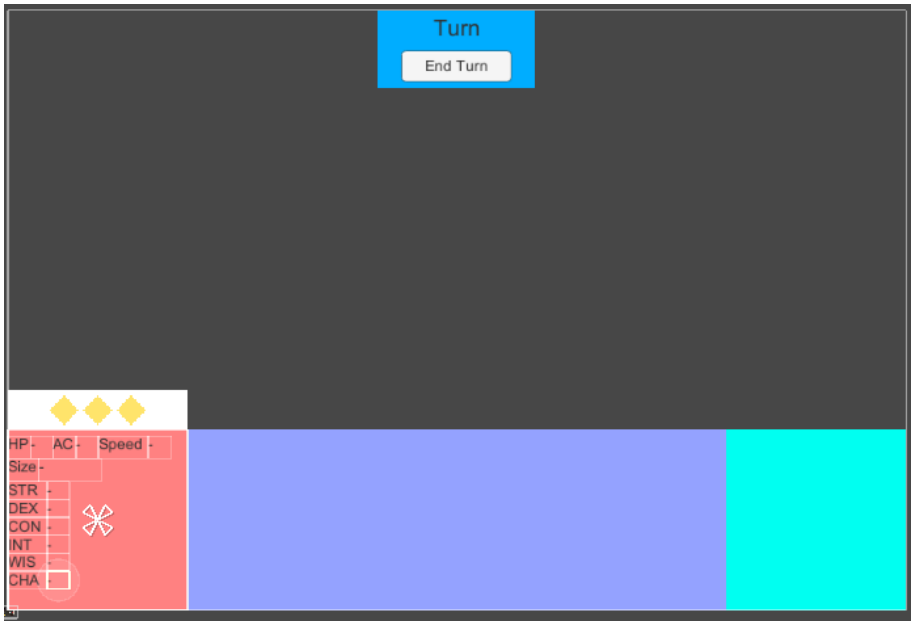

Рисунок 4.1 – BattleUI

У синій зоні будуть представлені карти, які будуть з стартом ігрового застосунку створюватись та додаватись до цієї зони. Для створення карт треба створити окремий UI карт та додати його, як Prefab об’єкт.

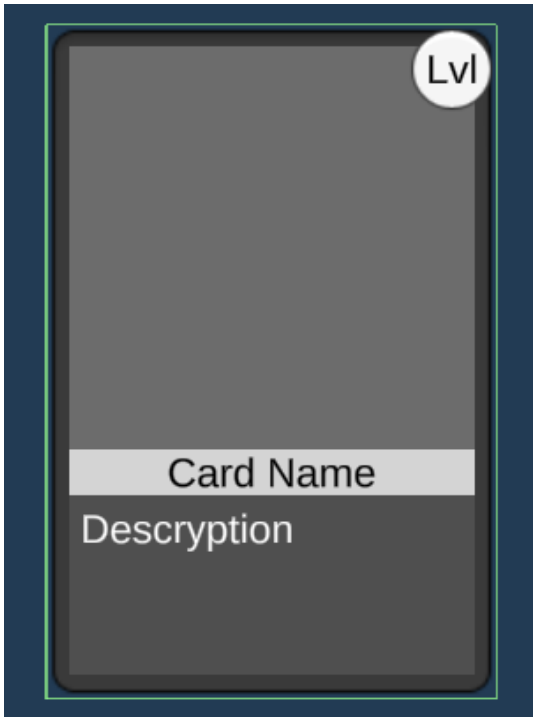

Рисунок 4.2 – Card

При створенні карт, карти повинні бути унікальними та для цього треба мати базу даних для цих карт. Для цього було вирішено використати додатковий клас типу MonoBehavior, а саме ScriptableObject.

Клас ScriptableObject дає можливість створити клас екземпляр, за яким можна створити об’єкт, який буде зберігатись в окремій папці, та визначити його параметри окремо. За допомогою цього можна зберігати унікальні об’єкти та їх параметри, а їх ініціалізація виконується навіть раніше за метод Awake. Саме таким чином буде створено нові картки, а так як у окремих персонажей буде своя колода, то зберігати всі картки в одному місці не має потреби.

Клас карти, екземпляр:

```
[CreateAssetMenu(fileName = "CardData", menuName = "Scriptable Creation/Card")]
public class CardData : ScriptableObject {
   [SerializeField] private int id;
   [SerializeField] private Sprite sprite;
   [SerializeField] private new string name;
   [SerializeField] private string description;
   [SerializeField] private int level;
   [SerializeField] private int actions;
   [SerializeField] private string actionName;
   private IAction strategy;

   public int Id {  get { return id; } set { id = value; } }
   public Sprite Sprite { get { return sprite; } set { sprite = value; } }
   public string Name { get { return name; } set { name = value; } }
   public string Description { get { return description; } set { description = value; } }
   public int Level { get { return level; } set { level = value; } }
   public int Actions { get { return actions; } set { actions = value; } }
   public IAction Strategy { get { return strategy; } set {  strategy = value; } }

   public void SetAction()
   {
      Strategy = (IAction)Activator.CreateInstance(Type.GetType(actionName));
   }
}
```

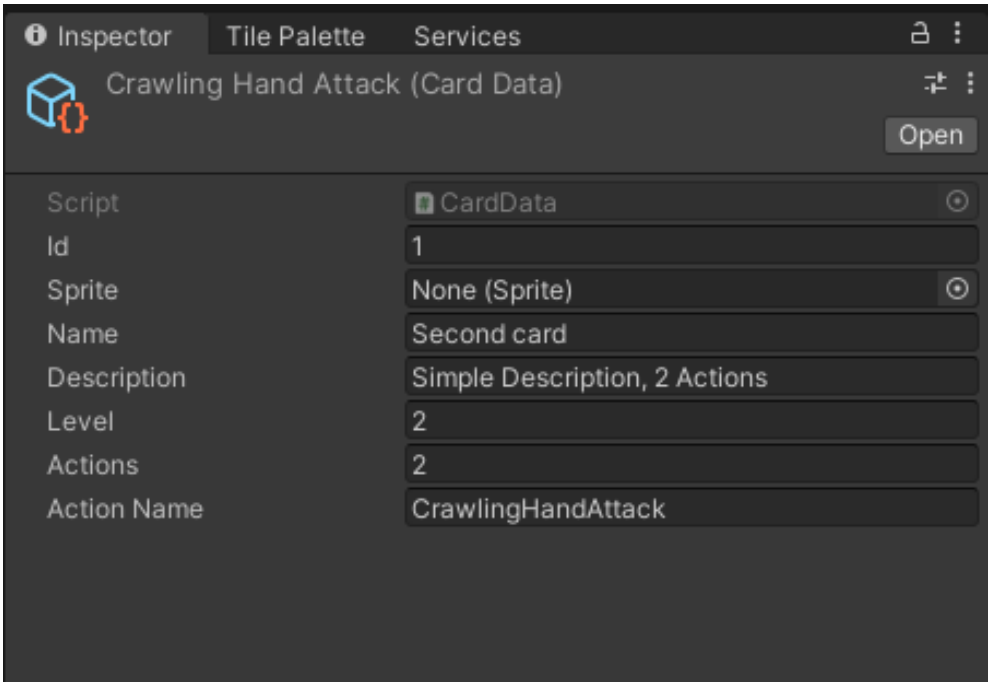

Рисунок 4.3 – Задавання параметрів карті

Саме таким чином створюється об’єкт карти, із цікавого тут використано паттерн Strategy, він використовується тут для прив’язання логіки до карти. Є інтерфейс IAction який має функцію активації карти та класи, що наслідують цей інтерфейс та реалізують цю функцію. В об’єкта карти є атрибут з типом даних IAction і саме сюди створюється об’єкт одного з класів, що реалізують інтерфейс. Також для зручності занесення даних до об’єктів UI, було створено скрипт CardPrefabUI та прив’язано об’єкти карти до цього скрипту.

Для створення карт потрібен окремий скрипт, який буде відповідати за менеджмент карт в руці, колоді та збросі. Цей клас буде CardManager, він матиме методи для створення карти, її видалення, для заповнення руки картами, для переміщення карт зі збросу до колоди. Також для огляду карт треба відображати їх в руці, для цього є два методи, це додати до зони руки компонент розташування об’єктів у горизонтальній проекції або створити власний скрипт для їх розташування. Поміж цих двох варіантів було обрано створити власний скрипт для цього.

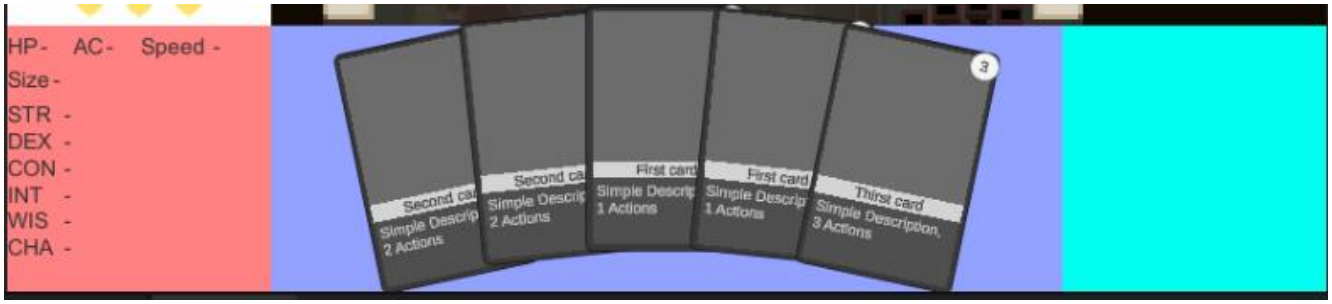

Рисунок 4.4 –Наповнення руки картами

Після реалізації цього скрипту сцена виглядає саме так. За допомогою параметрів в компоненті, розташування карт, їх нахил, відстань між ними можна змінювати.

Тепер треба додати функціонал до самих карток, тому додаємо новий скрипт UsingCards до готового об’єкту карти. Цей скрипт відповідає за взаємодію з картою, а саме при натисканні на неї, при її перенесенні та відпусканні її, коли курсор мишки на карті та покидає карту. Також повернення карти на її місце розташування серед інших карт.

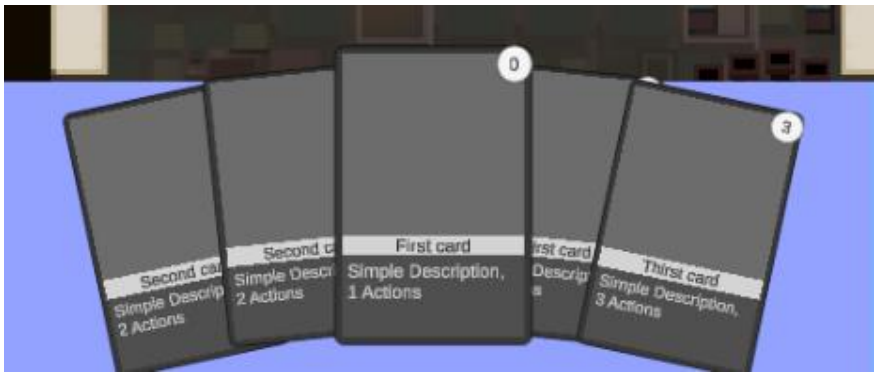

Рисунок 4.5 – Коли курсор на карті

При натисканні на карту, вона ставиться по центру руки для того, щоб можна було побачити її характеристики.

Карту треба використовувати на персонажа, тому такого персонажа треба створити. Для цього буде використано попередній метод з ScriptableObject. Персонаж буде мати характеристики, колоду карт, ім’я, спрайт. Клас персонажа, екземпляр:

```
[CreateAssetMenu(fileName = "CharacterData", menuName = "Scriptable Creation/Character")]
public class CharacterData : ScriptableObject, ICloneable {
    [Header("Sheet Settings")]
    [SerializeField] private new string name;
    [SerializeField] private int hp;
    [SerializeField] private int ac;
    [SerializeField] private int lvl;
    [SerializeField] private Size size;
    [SerializeField] private int speed;
    [SerializeField] private Strength strength;
    [SerializeField] private Dexterity dexterity;
    [SerializeField] private Constitution constitution;
    [SerializeField] private Intelligence intelligence;
    [SerializeField] private Wisdom wisdom;
    [SerializeField] private Charisma charisma;
    [Header("Character Settings")]
    [SerializeField] private Sprite sprite;
    [SerializeField] private List<CardData> cards;

    public object Clone()
    {
        var clonedScriptableObject = Instantiate(this);
        return clonedScriptableObject;
    }

    public string Name { get { return name; } set { name = value; } }
    public int Hp { get { return hp; } set { hp = value; } }
    public int Ac { get { return ac; } set { ac = value; } }
    public int Lvl { get { return lvl; } set { lvl = value; } }
    public int Speed { get { return speed; } set { speed = value; } }
    public Size Size { get { return size; } set { size = value; } }
    public Strength Strength { get { return strength; } set { strength = value; } }
    public Dexterity Dexterity { get { return dexterity; } set { dexterity = value; } }
    public Constitution Constitution { get { return constitution; } set { constitution = value; } }
    public Intelligence Intelligence { get { return intelligence; } set { intelligence = value; } }
    public Wisdom Wisdom { get { return wisdom; } set { wisdom = value; } }
    public Charisma Charisma { get { return charisma; } set { charisma = value; } }
    public Sprite Sprite { get { return sprite; } set { sprite = value; } }
    public List<CardData> Cards { get { return cards; } set { cards = value; } } }
```

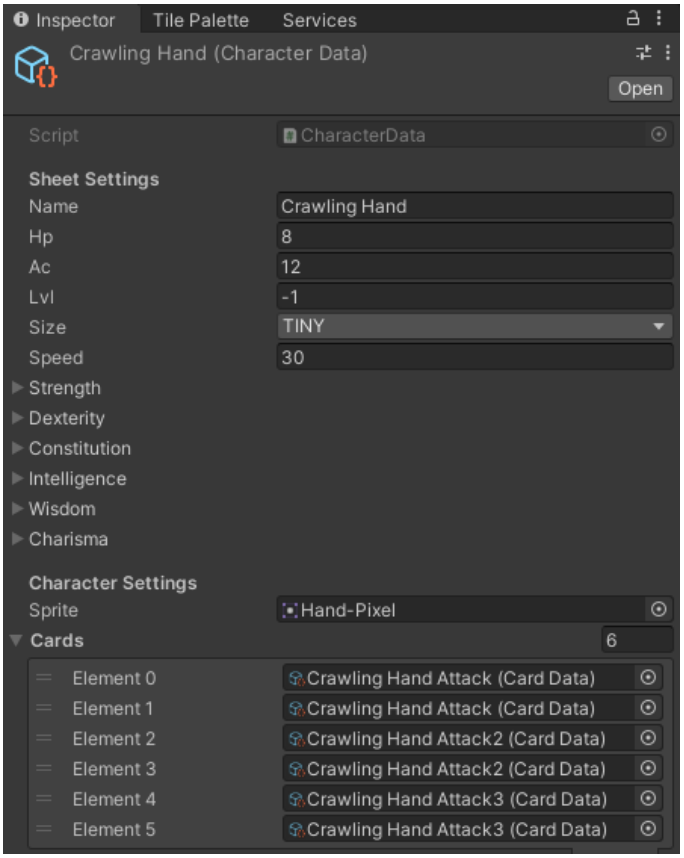

Рисунок 4.6 – Задавання параметрів персонажу

Після створення персонажів, треба створити скрипт для тримання цих даних на об’єкті, бо карти будуть направлені на ціль та отримувати дані про ціль від компонентів на цілі. Так як карти будуть наносити також ефекти, то потрібно до цього скрипту додати список для тримання ефектів та самі ефекти. Скрипт, що триматиме в собі дані персонажа та ефекти називається Entity. Для зручності треба зв’язати до скрипту інтерфейс.

Для ефектів потрібна кожному своя логіка, тому тут знову використовується паттерн Strategy.

Інтерфейс IEffect:

```
public interface IEffect {
   int State { get; set; }
   int Duration { get; set; }
   bool IsHaveDuration {  get; set; }
   public Sprite Sprite { get; set; }
   List<int> RestrictedTraits { get; }
```

```
    void ApplyEffect(Entity entity);
    void DurationEffect();
    void ClearEffect(Entity entity);
}
```

Клас ефекту Immobilized:

```
public class Immobilized : Effect, IEffect {
    private List<int> restrictedTraits = new List<int> { TraitTypeData.MOVE };

    public Immobilized(int state, int duration, bool isHaveDuration = true) : base (state,
duration, isHaveDuration)
    {
        Sprite = Resources.Load<Sprite>("Cactus_Col1");
    }

    public void ApplyEffect(Entity entity){}

    public void DurationEffect()
    {
        if (Duration > 0 && IsHaveDuration == true)
        {
            Duration -= 1;
        }
    }

    public void ClearEffect(Entity entity) {}

    public List<int> RestrictedTraits { get { return restrictedTraits; } }
}
```

Після створення ефектів та їх візуалу, йде створення UI елементів для показу параметрів на екрані. Для цього буде створено новий Prefab інтерфейсу сутності. На цьому інтерфейсі буде позначено ім’я, та після кожної зміни в характеристиках здоров’я персонажа ім’я буде змінювати свій колір. Зміна кольору відображається у процентах від характеристики здоров’я персонажа. Для ефектів буде створено окремий UI Prefab, бо він буде додаватись окремо при зміні у списку ефектів персонажа. Для відображення ефектів використовується горизонтальне розположення об’єктів[19].

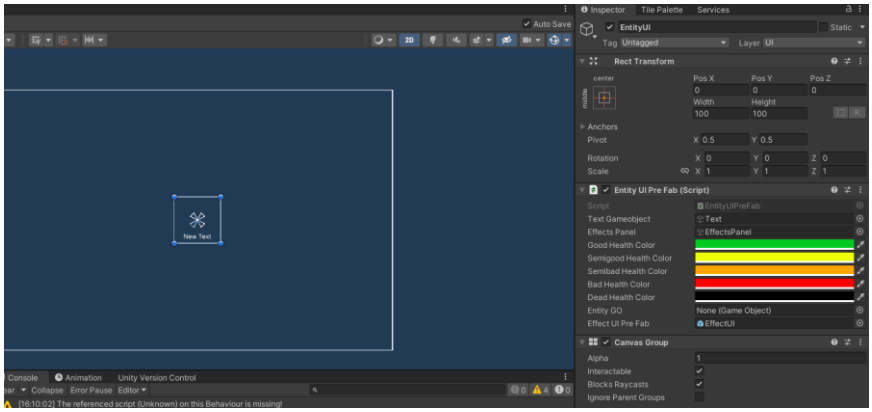

Рисунок 4.7 – Інтерфейс для сутності

Після налаштування самої цілі треба налаштувати обирання цілі та застосування ефектів карти до цілі. Та для візуального розуміння створити стрілку по якій буде показано саму ціль. Це все буде прописано у класі UsingCards.

Так як все відбувається на одній сцені та з самого початку сцени буде запускатися бій, то треба створити скрипт, який буде відповідальний за завантаження сутностей на сцену, формування руки карт гравцеві, постановлення ходу, закінчення битви. Задля цього функціоналу буде створено клас Battle. Також окремо він буде слідкувати за завантаженням інтерфейсів для сутностей. Після створення такого скрипту сцена гри буде виглядати саме так.

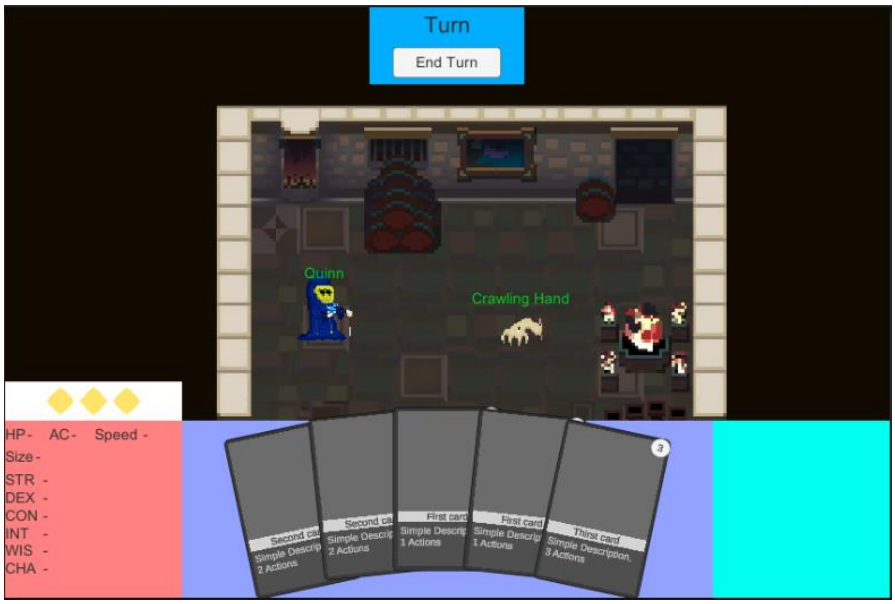

Рисунок 4.8 – Сцена після формування скрипту Battle

Також тепер після обиранні карти та її використанні з’являється ось така стрілка.

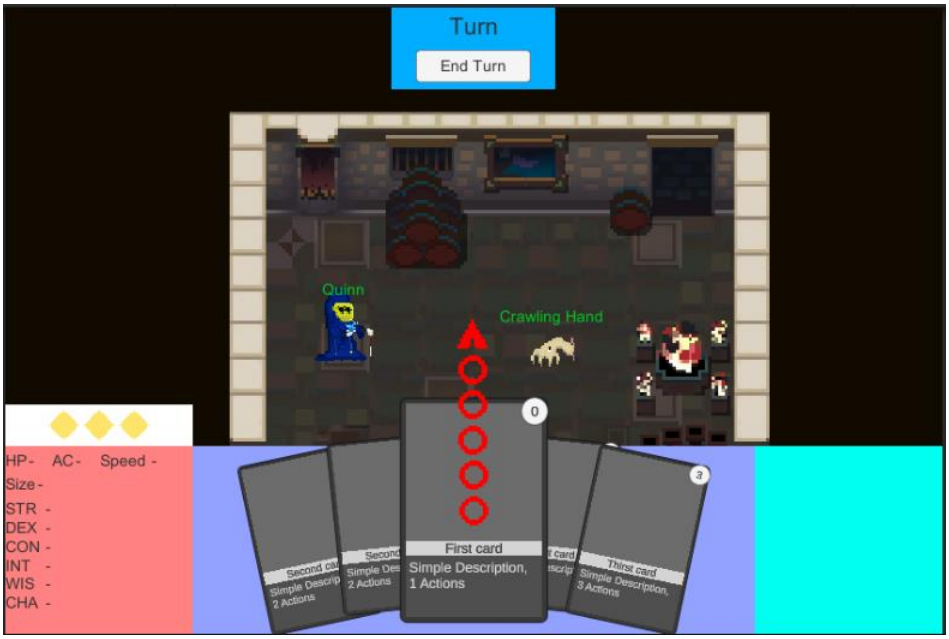

Рисунок 4.9 – Обирання цілі для карти

Після обирання цілі та використання карти на цю ціль в неї змінюється характеристика здоров’я та з’являється ефект.

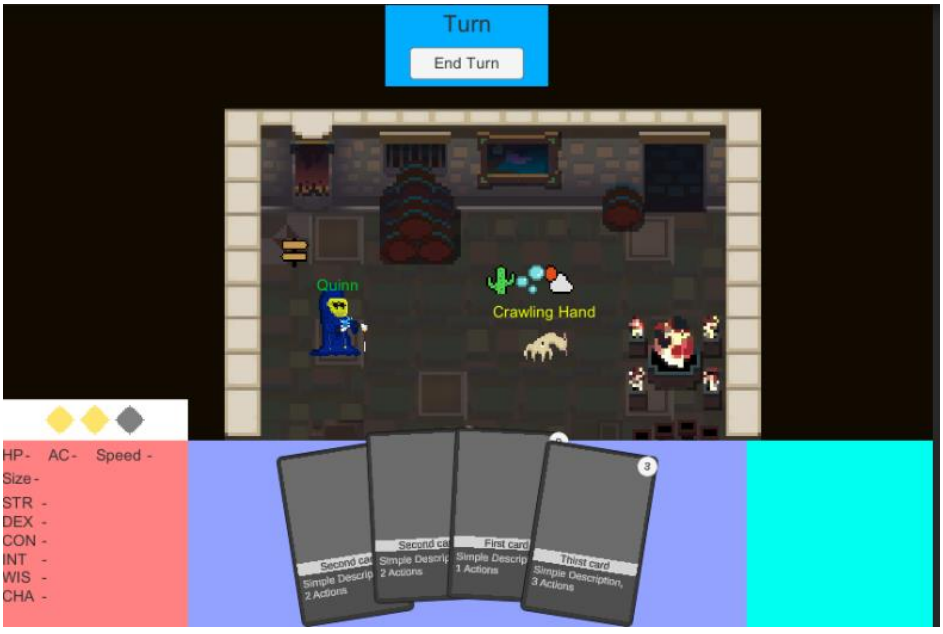

Рисунок 4.10 – Використання карти на ціль

Також карти мають такий параметр, як коштовність дії, тому при використанні карти на лівій частині екрану рахунок дій стає менше.

4.2 Тестування та рефакторинг

Тестування ігрового застосунку може бути різного виду та характеру. Для кожного тестування використовуються свої інструменти, наприклад для технічного тестування використовують інструменти ігрового двигуну. Для тестування ігрового застосунку було використано тестові сценарії. Вони є зручними для перевірки окремих об’єктів чи логіки.

Таблиця 4.1 – Тестовий сценарій №1

Особи	Гравець
Мета	Завершити хід використавши всі дії.
Передумова	У гравця не має карт із коштовністю більше 2.
Успішний сценарій: 1. Гравець починає бій. 2. Гравець отримує з колоди карти коштовністю: 1, 2, 1, 2, 1, 1. 3. Гравець обирає карту з коштовністю 2 та використовує на цілі. 4. Система наносить ефект противнику та зменшує кількість дій. 5. Гравець обирає карту з коштовністю 1, бо кількість доступних зараз дорівнює 1, та використовує карту на цілі. 6. Система наносить ефект противнику та зменшує кількість дій, Тепер гравець не має дій. 7. Гравець завершує свій хід.	
Сценарій успішний. Хід було завершено використавши усі наявні дії, система не дала змогу використати карти більше доступних дій.	

Цей тестовий сценарій перевіряє логіку системи доступних дій. У грі не було карт з коштовністю більше 2 для перевірки на менших значеннях.

Таблиця 4.2 – Тестовий сценарій №2

Особи	Гравець
Мета	Нанести ефект Grabbed на противника
Передумова	Гравцю випала карта з ефектом Grab.
Успішний сценарій: 1. Гравець починає бій. 2. Гравець обирає карту з ефектом Grab та використовує на цілі. 3. Система наносить ефект противнику та зменшує кількість дій. 4. Противник отримав ефект та тепер має менше здібностей в бою.	
Сценарій успішний. Ефект Grab було нанесено та логіка цього ефекту була спрацьована.	

Цей тестовий сценарій перевіряє логіку системи ефектів. Було перевірено саме цей ефект, бо він має найбільшу логіку серед інших ефектів та використовує разом з собою ще декілька інших ефектів.

Рефакторинг є дуже важливою частиною розробки та полірування ігрового застосунку, особливо якщо це довгостроковий проєкт або онлайн проєкт. Рефакторинг дозволяє більше легше орієнтуватись у власному коді та створювати більш гнучку структуру застосунку.

Щодо рефакторингу коду в ігровому застосунку, то під час розробки було використано паттерн Strategy у деяких скриптах, це дає певний алгоритм роботи та простоти[18].

Для зменшення коду було використано метод рефакторингу Extract Superclass, за допомогою цього методу були створенні додаткові класи для вилучення дуплікації коду у класах.

Для зменшення та розподілення функціоналу було використано метод рефакторингу Extract Method, за допомогою цього методу певний функціонал з одного методу було розподілене на декілька методів.

Для зменшення кількості полів у об’єкті було використано метод рефакторингу Replace Data Value with Object, за допомогою цього методу було створено додаткові класи та змінні замінені на нумератори.

Висновки до розділу 4

На основі розробки та тестуванні ігрового застосунку було розроблено весь функціонал, що був наведений у розділі проєктування, було продемонстровано процес розробки ігрового застосунку, процес створення ігрових об’єктів та інтерфейсу, процес використання функціоналу додаткового класу ScriptableObject.

Були змодельовані діаграми класів ігрового застосунку та розписаний функціонал наведених класів. За допомогою цих діаграм можна облегшити роботу при рефакторингу коду застосунку та для розуміння логіку проєкту.

Було проведено тестування ігрового застосунку за допомогою тест сценаріїв та було зроблено рефакторинг існуючого коду використовуючи вже існуючі методи рефакторингу.

За результатом розробки було розроблено ігровий застосунок на основі ігрового двигуну Unity 2D та за жанром roguelike. Було протестовано ігровий застосунок та проведено рефакторинг існуючого коду.

ВИСНОВКИ

Кваліфікаційна робота присвячена популяризації маловідомого всесвіту DarkSun за допомогою розробки ігрового застосунку на тему всесвіту та за допомогою розвиненого та поширеного жанру roguelike.

Використання ігрових застосунків для певної реклами чи популяризації певної теми є на зараз розповсюдженим явищем, наприклад для популяризації книжних усесвітів було створено ігри по типу Middle-Earth Shadow of War.

Результатом кваліфікаційної роботи є функціонуючий ігровий застосунок на базі Unity 2D та в жанрі roguelike.

В процесі виконання кваліфікаційної роботи було вирішено наступні завдання:

- Проаналізовано та порівняно між собою існуючі аналоги в жанрі roguelike, їх недоліки та переваги;
- Проаналізовано сам процес розробки ігрового застосунку та ігрову сферу;
- Специфіковані вимоги до ігрового застосунку;
- З модельовано ігровий застосунок, а саме його логіку та структуру за допомогою діаграм та таблиць;
- Спроєктовано ігровий застосунок, а саме визначено ігровий функціонал та відповідність до жанру. Також було продемонстровано окремі аспекти ігрового застосунку;
- Розроблено функціонал до ігрового застосунку із застосування методів рефакторингу, створено діаграму класів та протестовано застосунок.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. The C# Programming Language : підручник. A. Hejlsberg та інші. Pearson Education, 2008. 784 p. URL: http://surl.li/sdure (Last accessed: 02.04.2024)
2. Unity Documentation : вебсайт. URL: https://docs.unity.com (Дата звернення: 02.04.2024)
3. Pathfinder RPG: Pathfinder Player Core : підручник. L. Bonner та інші. Paizo Inc., 2023. 464 p.
4. Pathfinder RPG: Pathfinder GM Core : підручник. L. Bonner, M. Seifter. Paizo Inc., 2023. 336 p.
5. It’s all in here: The ultimate guide to creating UI interfaces in Unity : вебсайт. URL: https://blog.unity.com/games/ultimate-guide-to-creating-ui-interfaces (Last accessed: 02.04.2024)
6. 2D Pixel Perfect: How to set up your Unity project for retro 8-bit games : вебсайт. URL: https://blog.unity.com/games/2d-pixel-perfect-how-to-set-up-your-unity-project-for-retro-8-bits-games (Last accessed: 02.04.2024)
7. Our first-ever e-book for level designers is here : вебсайт. URL: https://blog.unity.com/games/e-book-for-level-designers (Last accessed: 02.04.2024)
8. Unity Documentation. Animation : вебсайт. URL: https://docs.unity3d.com/Manual/AnimationSection.html (Last accessed: 02.04.2024)
9. Unity Documentation. 2D game development quickstart guide : вебсайт. URL: https://docs.unity3d.com/Manual/Quickstart2D.html (Last accessed: 02.04.2024)
10. Unity Engine : вебсайт. URL: https://unity.com/products/unity-engine (Last accessed: 02.04.2024)
11. Roguedice : вебсайт. URL: https://khaliu.itch.io/roguedice (Last accessed: 02.04.2024)
12. Slay The Spire. Steam Store: вебсайт. URL: https://store.steampowered.com/app/646570/Slay_the_Spire/ (Дата звернення: 02.04.2024)
13. Quest of Dungeons. Steam Store : вебсайт. URL: https://store.steampowered.com/app/270050/Quest_of_Dungeons/ (Дата звернення: 02.04.2024)
14. 2D Pixel Perfect. Unty Manual : вебсайт. URL: https://docs.unity3d.com/Packages/com.unity.2d.pixel-perfect@5.0/manual/index.html (Last accessed: 02.04.2024)
15. Working with Unity Cameras. Unity Learn : вебсайт. URL: https://learn.unity.com/tutorial/working-with-unity-cameras-questions?signup=true# (Last accessed: 02.04.2024).
16. What Are Roguelike and Roguelite Video Games? MakeUsOf: вебсайт. URL: https://www.makeuseof.com/what-are-roguelike-and-roguelite-video-games/ (Last accessed: 03.04.2024)

17. What is Unity? – A Top Game Engine for Video Games. Zenva: вебсайт. URL: https://gamedevacademy.org/what-is-unity/ (Last accessed: 04.04.2024)
18. Refactoring. Refactoring GURU: вебсайт. URL: https://refactoring.guru (Last accessed: 20.04.2024)
19. How to Create a Card Game in Unity. Learn to Create Games: вебсайт. URL: https://learntocreategames.com/how-to-create-a-card-game-in-unity/ (Last accessed: 21.04.2024)
20. Detecting Mouse Clicks. Unity Learn: вебсайт. URL: https://learn.unity.com/tutorial/onmousedown (Last accessed: 05.05.2024)

ДОДАТОК А

Код скрипту CardManager

```
public class CardManager : MonoBehaviour
{
    #region parameters

    private List<CardData> cardDataDeck = new List<CardData>();
    private List<CardData> cardDataDiscard = new List<CardData>();

    private List<GameObject> cardHand = new List<GameObject>();

    public GameObject cardPreFab;
    public GameObject handArea;

    public int handSize;

    public float cardsSpread = -6f;
    public float cardsHorizontalSpacing = 70f;
    public float cardsVerticalSpacing = 18f;

    #endregion

    public void FillHandArea()
    {
        int tempCardHandCount = cardHand.Count;

        if (cardDataDeck.Count == 0)
        {
            FromDiscardToDeck();
        }
        if (cardHand.Count != handSize && cardDataDeck.Count != 0)
        {
            if (cardDataDeck.Count < handSize - cardHand.Count)
            {
                tempCardHandCount += cardDataDeck.Count;
            }
            for (int i = tempCardHandCount; i < handSize; i++)
            {
                DrawCardToHand();
            }
        }
```

```
    }

    public void UpdateHandView()
    {
        int cardsHandCount = cardHand.Count;

        if (cardsHandCount == 1)
        {
            cardHand[0].transform.localRotation = Quaternion.Euler(0f,0f,0f);
            cardHand[0].transform.localPosition = new Vector3(0f, 0f, 0f);
            return;
        }

        for (int i = 0; i < cardsHandCount; i++)
        {
            float rotationAngel = (cardsSpread * (i - (cardsHandCount - 1) / 2f));
            cardHand[i].transform.localRotation = Quaternion.Euler(0f, 0f, rotationAngel);

            float horizontalOffSet = (cardsHorizontalSpacing * (i - (cardsHandCount - 1) /
2f));

            float normalizedPosition = (2f * i / (cardsHandCount - 1) - 1f);

            float verticalOffSet = cardsVerticalSpacing * (1 - normalizedPosition *
normalizedPosition);

            cardHand[i].transform.localPosition = new Vector3(horizontalOffSet,
verticalOffSet, 0f);
        }
    }

    #region card transfer funcs

    public void DrawCardToHand()
    {
        GameObject tempCard = Instantiate(cardPreFab, handArea.transform.position,
Quaternion.identity, handArea.transform);

        tempCard.GetComponent<CardObject>().CardData = cardDataDeck[0];
        tempCard.GetComponent<CardObject>().SetCardInfo();

        cardHand.Add(tempCard);
```

```
        cardDataDeck.RemoveAt(0);

        UpdateHandView();
    }

    public void DropCardToDiscard(GameObject card)
    {
        CardData tempCardData = card.GetComponent<CardObject>().CardData;

        cardHand.Remove(card);
        Destroy(card);

        cardDataDiscard.Add(tempCardData);

        UpdateHandView();
    }

    public void FromDiscardToDeck()
    {
        cardDataDeck = new List<CardData>(cardDataDiscard);
        cardDataDiscard.Clear();
    }

    #endregion

    public List<CardData> CardDataDeck {  get { return cardDataDeck; } set {
cardDataDeck = value; } }
        }
```

ДОДАТОК Б

Код скрипту UsingCards

```
public class UsingCards : MonoBehaviour
{
    #region parameters

    private GameObject battleUI;
    private GameObject handZone;
    private GameObject gameManager;
    private CardManager cardManager;

    public Arrow arrow;
    public CardObject cardObject;
    public RectTransform rectTransform;

    private Vector3 originalScale;
    private CardState state = CardState.NOEVENT;
    private Quaternion originalRotation;
    private Vector3 originalPosition;
    private int originalHierarchyIndex;

    [SerializeField] private float hoverScale = 1.1f;
    [SerializeField] private float selectScale = 1.3f;
    [SerializeField] private Vector3 offSetFromHand = new Vector3(0f, 25f, 0f);

    #endregion

    #region init

    private void Awake()
    {
        handZone = GameObject.Find("Hand");
        battleUI = GameObject.Find("BattleUI");
        gameManager = GameObject.Find("GameManager");
        cardManager = gameManager.GetComponent<CardManager>();

        originalScale = rectTransform.localScale;
        originalRotation = rectTransform.localRotation;
        originalPosition = rectTransform.localPosition;
        originalHierarchyIndex = rectTransform.GetSiblingIndex();
    }
```

```
#endregion

#region event funcs

public void StartDragging()
{
    arrow.CreatePoints();

    if (state == CardState.POINTERENTER)
    {
        state = CardState.DRAGING;

        rectTransform.localRotation = Quaternion.identity;
        rectTransform.localPosition = offSetFromHand;

        rectTransform.localScale = originalScale * selectScale;
    }
}

public void Dragging()
{
    arrow.MakeArrowMove();
}

public void EndDragging()
{

    rectTransform.localPosition = originalPosition;
    rectTransform.localRotation = originalRotation;
    rectTransform.localScale = originalScale;

    arrow.DeletePoints();

    Collider2D target = GetObjectOnMousePos();
    if (target != null)
    {
        SetTarget(target);
    }

    ReturnToHand();
}
```

```
    public void OnMouseCardEnter()
    {
        if (state == CardState.NOEVENT)
        {
            originalPosition = rectTransform.localPosition;
            originalRotation = rectTransform.localRotation;
            originalScale = rectTransform.localScale;
            originalHierarchyIndex = rectTransform.GetSiblingIndex();

            state = CardState.POINTERENTER;

            rectTransform.localScale = originalScale * hoverScale;
            rectTransform.SetAsLastSibling();
        }
    }

    public void OnMouseCardExit()
    {
        if (state == CardState.POINTERENTER)
        {
            state = CardState.NOEVENT;

            rectTransform.localPosition = originalPosition;
            rectTransform.localRotation = originalRotation;
            rectTransform.localScale = originalScale;
            rectTransform.SetSiblingIndex(originalHierarchyIndex);
        }
    }

    #endregion

    #region funcs

    public Collider2D GetObjectOnMousePos()
    {
        Vector3 mousePosition =
Camera.main.ScreenToWorldPoint(Input.mousePosition);

        RaycastHit2D hitInfo = Physics2D.Raycast(mousePosition, Vector2.zero);

        return hitInfo.collider;
```

```
    }

    public void SetTarget(Collider2D target)
    {
        if
(gameManager.GetComponent<Battle>().battlePlaces.Contains(target.gameObject))
        {

            if (gameManager.GetComponent<Battle>().actions >=
cardObject.CardData.Actions)
            {
                CardUse(target.gameObject);
            }
        }
    }

    public void CardUse(GameObject target)
    {
        cardObject.CardData.Strategy.UseAction(target.GetComponent<Entity>(),
GameObject.Find("Player").GetComponent<Entity>());


gameManager.GetComponent<Battle>().ChangeActionsState(cardObject.CardData.Act
ions);

        cardManager.DropCardToDiscard(gameObject);
    }

    public void ReturnToHand()
    {
        state = CardState.NOEVENT;
        rectTransform.localScale = originalScale;
        rectTransform.localRotation = originalRotation;
        rectTransform.localPosition = originalPosition;
        rectTransform.SetSiblingIndex(originalHierarchyIndex);
    }

    #endregion
        }
```

ДОДАТОК В

Код скрипту Battle

```
public class Battle : MonoBehaviour
{
    #region parameters

    [Header("GameObjects")]
    public CardManager cardManager;

    public GameObject battleUI;
    public GameObject player;
    public GameObject rewardUI;
    public GameObject playerStatsUI;
    public GameObject eventsUI;
    public GameObject eventMenuUI;

    public List<GameObject> battlePlaces = new List<GameObject>();

    [Header("Prefabs")]
    public GameObject entityUIPreFab;

    public EnemyPresetsData enemiesPreset;

    [Header("Parameters")]
    public bool isPlayerTurn;
    public bool isBattleEnded = false;

    [Header("Actions")]
    public GameObject actionsZone;
    public Color actionsCharColor;

    public int actions = 3;
    private CharacterData playerData;

    #endregion

    private void SetUpEntities()
    {
        playerData =
GameObject.Find("DataHolder").GetComponent<PlayerDataHolder>().character;
```

```
        for (int i = 0; i < battlePlaces.Count; i++)
        {
            if (enemiesPreset.enemiesPreset.Count < i)
            {
                battlePlaces[i].gameObject.SetActive(false);
            }
            else
            {
                battlePlaces[i].GetComponent<Entity>().Original =
enemiesPreset.enemiesPreset[i];
                battlePlaces[i].GetComponent<Entity>().CharacterSheet =
enemiesPreset.enemiesPreset[i];
                battlePlaces[i].GetComponent<Entity>().SetEntity();
            }
        }

        player.GetComponent<Entity>().Original = playerData;
        player.GetComponent<Entity>().CharacterSheet = playerData;
        battlePlaces.Add(player);


playerStatsUI.GetComponent<PlayerStatsUI>().SetStats(player.GetComponent<Entity
>().Original);
    }

    private void SetUpBattleUI()
    {
        battleUI.gameObject.SetActive(true);

        foreach(GameObject entityGO in battlePlaces)
        {
            GameObject entityUI = Instantiate(entityUIPreFab, battleUI.transform, false);
            entityUI.GetComponent<EntityUIPreFab>().entityGO = entityGO;
            entityUI.GetComponent<EntityUIPreFab>().SetUIToEntity();
            entityGO.GetComponent<Entity>().UI =
entityUI.GetComponent<EntityUIPreFab>();
        }
    }

    void Start()
    {
        SetUpEntities();
```

```
        SetUpBattleUI();
        CardSSetup();
    }


    void Update()
    {
        if(!isPlayerTurn)
        {
            Debug.Log("Enemy maked his turn!");
            isPlayerTurn = true;
            TurnSetUp();
        }
        if (!isBattleEnded)
        {
            BattleEnd();
        }
    }

    private void BattleEnd()
    {
        foreach (GameObject go in battlePlaces)
        {
            if (go.activeSelf == true && go.GetComponent<Entity>().CharacterSheet.Hp >
0 && go.name != "Player" )
            {
                return;
            }
            else
            {
                isBattleEnded = true;
                battleUI.SetActive(false);
                rewardUI.SetActive(true);
            }
        }
    }

    private void CardSSetup()
    {
        cardManager.CardDataDeck =
player.GetComponent<Entity>().CharacterSheet.Cards;
        cardManager.FillHandArea();
```

```
    }

    //!!
    public void ChangeActionsState(int action)
    {
        actions -= action;

        switch (actions)
        {
            case 0:

actionsZone.transform.GetChild(0).gameObject.GetComponent<Image>().color =
Color.gray;
                goto case 1;
            case 1:

actionsZone.transform.GetChild(1).gameObject.GetComponent<Image>().color =
Color.gray;
                goto case 2;
            case 2:

actionsZone.transform.GetChild(2).gameObject.GetComponent<Image>().color =
Color.gray;
                break;
            default:
                break;
        }
    }

    private void TurnSetUp()
    {
        GameObject.Find("Turn").gameObject.GetComponentInChildren<Text>().text =
isPlayerTurn == true ? "Player turn" : "Enemy turn";
    }

    public void EndTurnOnClick()
    {
        //!!
        actions = 3;
        for(int i = 0; i < actions; i++)
        {
```

```
            actionsZone.transform.GetChild(i).gameObject.GetComponent<Image>().color
= actionsCharColor;
        }
        cardManager.FillHandArea();
        isPlayerTurn = false;
        TurnSetUp();
    }

    public void CloseRewardButton()
    {
        eventsUI.SetActive(true);
        rewardUI.SetActive(false);
    }
}
```