Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

Бакалаврська дипломна робота

на тему: Розробка мультиплеєрної 2D-гри з механікою ідентифікації малюнків

Виконав: студент IV курсу
групи ЗПІ-20Б
спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Рибачок Я.А.
(прізвище та ініціали)

Керівник: к.т.н., доц. каф. ПЗ Хошаба О.М.
(прізвище та ініціали)

Рецензент: к.т.н., доц. каф ОТ Черняк О.І.
(прізвище та ініціали)

Допущено до захисту

Зав. кафедри ПЗ Романюк О.Н.
« 10 » червня 2024 р.

ВНТУ – 2024

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший (бакалаврський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
О. Н. Романюк
« 12 » березня 2024 р.

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Рибачку Ярославу Аркадійовичу

1. Тема роботи – розробка мультиплеєрної 2D-гри з механікою ідентифікації малюнків.

Керівник роботи: Хошаба Олександр Мирославович, к.т.н., доц. кафедри ПЗ, затверджені наказом вищого навчального закладу від 11 березня 2024 р. № 80.

2. Строк подання студентом роботи 3 червня 2024.

3. Вихідні дані до роботи: генерація теплокарти, генерація індикаторів кольорів, оптимізація процесів малювання, синхронізація текстур через мережу, визначення загальної подібності між зображеннями.

4. Зміст текстової частини: вступ; аналіз предметної області; аналіз аналогів розроблюваного програмного застосунку; аналіз методів розв'язання завдання; розробка моделі системи; розробка системи підказок для порівняння зображень; розробка архітектури для мультиплеєрної гри; розробка модуля для малювання; розробка машини станів; розробка інтерфейсу; розробка серіалізації малюнків через мережу; аналіз методів тестування; тестування програмного застосунку.

5. Перелік ілюстративного матеріалу: титульний слайд; мета, об'єкт та предмет дослідження; актуальність розробки; UML діаграма варіантів використання; UML діаграма послідовності; UML діаграма класів машини станів; UML діаграма ключових модуль архітектури; генерація індикаторів; використання теплокарти; евклідова відстань; генерація теплокарти; алгоритм Mean Squared Error; результати досліджень; новизна отриманих результатів; апробація результатів дослідження.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
1-4	Хошаба О.М., к.т.н., доц. кафедри ПЗ	13.03.24	03.06.24

7. Дата видачі завдання 13 березня 2024р

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз стану мультиплеєрних ігор	14.03.24 – 20.03.24	виконано
2	Розробка моделі системи та архітектури	22.03.24 – 29.03.24	виконано
3	Розробка системи підказок	02.04.24 – 15.04.24	виконано
4	Розробка модулів застосунку	15.04.24 – 23.05.24	виконано
5	Тестування застосунку	24.05.24 – 26.05.24	виконано
6	Оформлення матеріалів до захисту БДР	27.05.24 – 30.05.24	виконано

Студент _____ **Я. А. Рибачок**
(підпис) (ініціали та прізвище)

Керівник бакалаврської кваліфікаційної роботи _____ **О. М. Хошаба**
(підпис) (ініціали та прізвище)

АНОТАЦІЯ

УДК 004.912.032.26

Рибачок Я. А. Розробка мультиплеєрної 2D-гри з механікою ідентифікації малюнків : бакалаврська кваліфікаційна робота зі спеціальності 121 – інженерія програмного забезпечення, освітня програма – інженерія програмного забезпечення. Вінниця: ВНТУ, 2024. 104 с.

На укр. мові. Бібліогр. : 23 назв; рис. : 33; табл. : 9.

Проведено аналіз мультиплеєрних ігор та аналогів застосунку, аналіз методів створення 2D-ігор та методів порівняння зображень. За результатами аналізів вибрано засоби вирішення задач.

Розроблено модель системи та визначено архітектурні особливості інфраструктури застосунку. Створено модулі генерації підказок для механіки ідентифікації малюнків.

Реалізовано та оптимізовано модуль для малювання на полотні. Створено графічний інтерфейс користувача та розроблено тип даних для синхронізації малюнків через мережу.

Проведено аналіз методів тестування, на основі цього аналізу складено сценарії використання та написано тест-кейси, за якими було проведено тестування та перевірено функціонування системи.

Отримані в бакалаврській дипломній роботі результати можна використовувати для покращення дедуктивного мислення.

Ключові слова: ігри, 2D, мультиплеєр, ідентифікація, малюнки, підказки, дедукція.

ABSTRACT

UDC 004.912.032.26

Rybachok Y. A. Development of a multiplayer 2D game with drawing identification mechanics: bachelor's thesis in specialty 121 – Software Engineering, educational program – Software Engineering. Vinnytsia: VNTU, 2024. 104 pages.

In Ukrainian. Bibliography: 23 references; figures: 33; tables: 9.

An analysis of multiplayer games and application analogs, methods for creating 2D games, and methods for image comparison has been conducted. Based on the analysis, tools for solving the tasks were selected.

A system model was developed, and the architectural features of the application infrastructure were determined. Modules for generating hints for the drawing identification mechanics were created.

A module for drawing on a canvas was implemented and optimized. A graphical user interface was created, and a data type for synchronizing drawings over the network was developed.

An analysis of testing methods was conducted, based on which usage scenarios were compiled and test cases were written. Testing was carried out, and the system's functionality was verified.

The results obtained in the bachelor's thesis can be used to improve deductive thinking.

Keywords: games, 2D, multiplayer, identification, drawings, hints, deduction.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження.

У сучасному суспільстві майже кожний принаймні раз мав досвід гри у відеоігри. Відеоігри охоплюють широкий віковий діапазон, оскільки їхній спектр варіюється від навчально-розвивальних проєктів до тих, що зосереджуються на сюжетних лініях або багатокористувацькій взаємодії. Проте, незважаючи на значний розвиток ігрової індустрії, існує проблема недостатньої кількості інноваційних підходів до створення нових ігор, які б об’єднували елементи навчання та розваги. Особливо це стосується соціальних ігор, які спрямовані на створення та підтримку віртуальних спільнот.

Мультиплеєрні ігри відповідають потребі людей у спілкуванні та взаємодії. Вони створюють платформи для об’єднання гравців з усього світу, дозволяючи їм співпрацювати, змагатися і будувати спільноти. Це особливо важливо в умовах глобалізації та зростання використання цифрових технологій, коли фізичні відстані стають менш значущими.

Розвиток інтернет-технологій, збільшення швидкості передачі даних і поширення доступу до високошвидкісного інтернету сприяють зростанню популярності мультиплеєрних ігор. Хмарні технології та вдосконалення апаратного забезпечення дозволяють створювати масштабні, реалістичні та інтерактивні ігрові середовища, які приваблюють все більше гравців.

Багатокористувацькі ігри використовуються також у освітніх цілях. Вони можуть сприяти розвитку різних навичок, таких як стратегічне мислення, співпраця, комунікація та лідерські якості. Використання ігрових методів у навчанні допомагає залучити студентів і зробити процес навчання більш інтерактивним та цікавим.

Мультиплеєрні ігри демонструють високу адаптивність до змін в культурі та технологіях. Вони постійно оновлюються, пропонуючи новий контент, події та функції, що дозволяє підтримувати інтерес гравців і залучати нову аудиторію.

Популярність мультиплеєрних ігор обумовлена кількома ключовими факторами. По-перше, мультиплеєрні ігри надають гравцям можливість взаємодії з іншими гравцями в реальному часі, що створює динамічний і соціально насичений ігровий досвід. По-друге, конкурентний аспект таких ігор стимулює гравців до постійного вдосконалення своїх навичок та досягнення високих результатів. По-третє, регулярні оновлення контенту та події, організовані розробниками, підтримують інтерес до гри на довготривалій основі.

Згідно з даними Steam Charts, мультиплеєрні ігри займають 90% серед найпопулярніших ігор на платформі. Більшість ігор у топ-10 за кількістю гравців на Steam є мультиплеєрними, що підкреслює домінування цього жанру на ринку.

Тому актуальним є розробка мультиплеєрної 2D-гри з механікою ідентифікації малюнків.

Зв'язок роботи з науковими програмами, планами, темами.

Робота виконувалася згідно плану виконання наукових досліджень на кафедрі програмного забезпечення.

Мета та завдання дослідження.

Метою роботи є підвищення соціальної взаємодії користувачів та розвиток дедуктивного мислення за рахунок створення доступної та оригінальної мультиплеєрної гри з механікою ідентифікації малюнків.

Відповідно до поставленої мети потрібно вирішити такі завдання:

– проаналізувати аналоги розроблюваного застосунку, дослідити інструменти для створення мультиплеєрних ігор та методи порівняння малюнків;

– розробити модель системи;

– дослідити алгоритми порівняння картинок та розробити модуль для генерації статистичних даних, які відображають різницю між оригіналом та зміненим зображенням;

– спроектувати та розробити архітектуру гри, що описує ключові елементи інфраструктури та їх роль;

– розробити модуль для малювання на полотні;

– реалізувати графічний інтерфейс гри з елементами управління, відображенням інформації про стан гри;

– реалізувати метод серіалізації та передачі картинок через мережу з урахуванням оптимізації розміру даних і швидкості передачі для ефективного обміну об'єктами;

– провести тестування застосунку.

Об’єктом дослідження є особливості та принципи роботи мультиплеєрних ігор, та методи ідентифікації малюнків.

Предметом дослідження є методи та засоби створення мультиплеєрних ігор, алгоритми для порівняння зображень, принципи роботи ігрового рушія Unity, використання фреймворку Netcode for GameObjects, ігрові сервіси Unity Gaming Services та принципи мови програмування C#.

Методи дослідження.

У процесі дослідження використовувались такі методи дослідження: методи алгоритмів та структур даних для написання коду програми, методи об’єктно-орієнтованого програмування для розробки додатку, методи комп’ютерного моделювання для тестування роботи додатку.

Новизна отриманих результатів.

Здобув подальшого розвитку метод “візуалізація різниці за допомогою теплової карти”, особливість якого полягає у використання евклідової відстані для знаходження дистанції між кольорами, який замінив використання різниці інтенсивності, що дозволило використовувати теплокарту для ідентифікації правильного кольору.

Запропоновано підхід ідентифікації малюнків у соціальних іграх, основною відмінністю якого є корегування власного малюнку, відповідно до отриманих підказок, що дає можливість покращувати дедуктивне мислення.

Удосконалено процес малювання на полотні, який полягає у використанні шейдерів, що дає можливість знизити навантаження на центральний процесор через перенесення частини обчислень на графічний процесор, і дозволяє

зменшити час рендирингу кадру з 250 мілісекунд до 4 мілісекунд, що підвищує продуктивність на 6250%.

Практична цінність отриманих результатів. Практична цінність роботи полягає в тому, що на основі проведених у БДР досліджень розроблено алгоритми та програмні модулі мультиплеєрної 2D-гри з механікою ідентифікації малюнків.

Особистий внесок здобувача. Наукові результати отримано здобувачем самостійно. У праці, опублікованій у співавторстві, студенту належить виконання аналізу методів для порівняння зображень [1].

Апробація результатів роботи. Результати роботи були представленні на конференції «Молодь в науці: дослідження, проблеми, перспективи (МН-2024)».

Публікації. Результати роботи були опубліковані в матеріалах конференції «Молодь в науці: дослідження, проблеми, перспективи (МН-2024)».

Пояснювальна записка містить 104 сторінок тексту, 33 рисунка, 9 таблиці, 4 додатки, 23 найменування в списку використаних джерел.

1 АНАЛІЗ СТАНУ ПИТАННЯ ТА ПОСТАНОВКА ЗАДАЧ

1.1 Аналіз предметної області

Однією з перших мультиплеєрних ігор була "Spacewar!", створена у 1962 році у Массачусетському технологічному інституті (MIT)[2]. Ця гра дозволяла двом гравцям керувати космічними кораблями та боротися один з одним в космічному просторі.

Першою мережевою грою була "MUD1" (Multi-User Dungeon), створена у 1978 році. Ця гра представляла собою текстовий мультиплеєрний рольовий квест, де гравці могли взаємодіяти один з одним у віртуальному світі.

Згодом, з появою Інтернету та поширенням комп'ютерів, мережеві ігри почали набирати популярності. 1990-2000-і роки були періодом активного розвитку мережевих ігор, з'явилися такі жанри, як онлайн-шутери, MMORPG та онлайн-стратегії.

У 2000-х роках з появою швидкого Інтернету та покращенням технічних можливостей геймерів, мережеві ігри стали ще більш популярними. Гравці могли змагатися або співпрацювати з іншими гравцями з усього світу.

Від моменту створення першої мультиплеєрної гри минуло багато часу, за який візуальна складова покращилась, як для два-вимірних так і для три-вимірних ігор. У порівнянні з 2D графікою, 3D вимагає більше обчислювальних потужностей, а з розвитком технологій що дають змогу передати віртуальний світ ще більш якісно, необхідність мати потужніший пристрій зростає. На противагу, використання двомірної графіки дозволяє створювати привабливі та доступні для широкої аудиторії проекти, які не вимагатимуть потужних пристроїв, і що дозволить портувати велику частину таких проектів на різні платформи.

Останні роки характеризуються зростанням кросс-платформених ігор, які дозволяють гравцям з різних платформ грати разом.

Кросс-платформені мультиплеєрні соціальні ігри є популярними серед більшості гравців, оскільки вони дозволяють спілкуватися, взаємодіяти та співпрацювати з іншими гравцями, підсилюючи аспекти соціальної взаємодії.

Кооперативні ігри відіграють важливу роль у сучасному ігровому світі, спонукуючи гравців до співпраці та колективного досягнення цілей в межах віртуального середовища. Одним із яскравих прикладів таких ігор є "Overcooked" – хаотична кулінарна гра, де команді гравців-кухарів доручено підготовку, приготування та подачу різноманітних замовлень, перед тим як клієнти покинуть заклад.

У "Overcooked" (див. рисунок 1.1) співпраця між гравцями не лише важлива, але і необхідна для успішного завершення рівнів. Гравці повинні взаємодіяти, обмінюватись ресурсами та ідеями, розподіляти обов'язки та координувати свої дії, щоб ефективно керувати рестораном у грі. Це вимагає від них не лише індивідуальних навичок, але і здатності працювати в команді, враховуючи потреби і можливості кожного гравця.

Рисунок 1.1 – Гра "Overcooked"

Масові онлайн-ігри (Mass Multiplayer Online) зазвичай мають велику кількість гравців, які взаємодіють у великих віртуальних світах. Гравці можуть

спілкуватися, обмінюватися ресурсами, створювати групи або клани та взаємодіяти у великих масштабах.

Прикладом ММО гри слугуватиме "Fortnite" (див. рисунок 1.2) – проект в жанрі королівської битви.

Рисунок. 1.2 – Гра "Fortnite"

Групові ігри зі спільними завданнями відкривають перед гравцями можливість об'єднатися у групи або команди для виконання спільних завдань. Це вимагає взаємодії, планування стратегій та координації дій для досягнення успіху в межах гри.

"Among Us" – один з найвідоміших прикладів групової гри зі спільними завданнями. Ця багатокористувацька відеогра на соціальну дедукцію в космічному антуражі пропонує гравцям дві ролі: цивільні члени екіпажу та самозванці. Метою екіпажу є знайти самозванців і нейтралізувати їх, в той час як самозванці мають за завдання знищити всіх членів екіпажу. Ця гра вимагає від гравців глибокого аналізу, стратегічного мислення та взаємодії, оскільки кожен крок може вплинути на результат ігри.

Що стосується взаємодії в групових іграх, "Among Us" (див. рисунок 1.3) демонструє, як правильне планування, співпраця та взаєморозуміння між гравцями можуть бути вирішальними для досягнення цілей гри. Гравці повинні ретельно аналізувати поведінку інших учасників, спілкуватися та обмінюватися інформацією, щоб приймати обдумані рішення, які сприятимуть команді в цілому.

Рисунок 1.3 – Гра "Among Us"

У соціальних іграх, що базуються на механіках малювання та вгадування малюнків, гравці мають можливість виразити своє мистецтво та креативність через малювання об'єктів або сцен. Ці механіки нерідко використовуються для створення комунікації між гравцями, розвиваючи спільноту та сприяючи соціальній взаємодії.

Гравці, які малюють, мають можливість обирати інструменти для малювання, такі як пензлі, кольори та товщина ліній, щоб створити точне або художнє відображення об'єкта або ідеї. Їхні малюнки потім передаються іншим гравцям для вгадування.

Гравці, які вгадують малюнки, повинні використовувати свою увагу та спостережливість, щоб розпізнати предмети або сюжети, намальовані іншими гравцями. Це може вимагати креативного мислення та асоціаційних здібностей для правильного вгадування об'єктів, які можуть бути відображені різними способами.

1.2 Аналіз аналогів розроблюваного програмного застосунку

На сьогоднішній день існує чимало соціальних ігор, особливо в яких ігровий процес будується навколо малювання і вгадування. Але аналіз і розробка власного рішення дає можливість запозичити кращі аспекти аналогів і побудувати власний унікальний ігровий процес.

Першим аналогом є Scribble It!! – популярна гра, в якої ігровий процес суміжний з грою “Крокодил”, кожний гравець по черзі має вибрати одне слово з трьох на вибір і зобразити об’єкт або передати значення малюнком, а інші гравці під час раунду мають вгадати і написати слово, теми можуть бути різними, від предметів, явищ, і до абстрактних понять.

На рисунку 1.4 зображено інтерфейс гравця, який обирає слово.

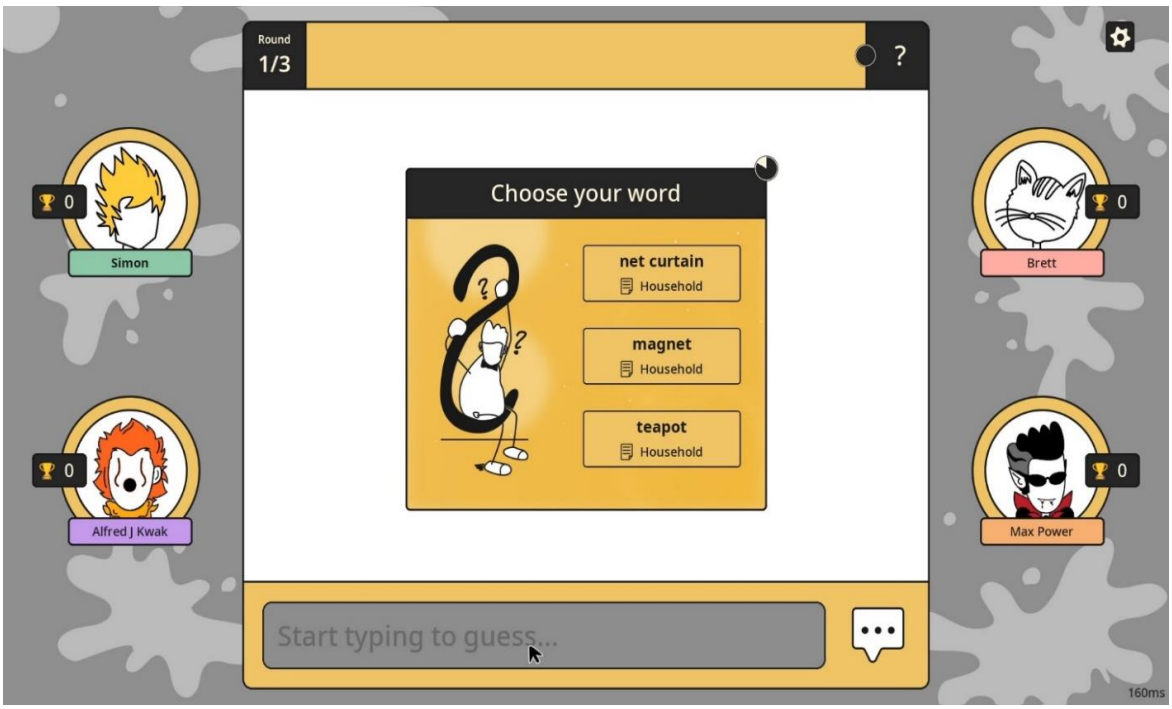


Рисунок 1.4 – Інтерфейс з опціями слів

Основні функції аналогу Scribble It!!:

– Функція малювання слів, де гравці по черзі намагаються намалювати слово, яке їм випало, а інші гравці спостерігають за процесом малювання та намагаються відгадати слово.

– Гра пропонує кілька режимів для різноманітності ігрового процесу. Серед них є режим з фіксованим часом, де гравці повинні швидко малювати і відгадувати слова в межах встановленого часу.

– Гравцям надається можливість створювати та використовувати власні набори слів та фраз для гри. Це дозволяє персоналізувати ігровий процес та адаптувати його під власні уподобання та інтереси.

Додаткові функції:

– Гра надає гравцям доступ до різноманітних професійних інструментів для малювання. Це включає різні кисті, олівці, інструменти для заповнення та інші корисні інструменти.

– Гравці можуть відстежувати свої ігрові результати та переглядати детальну статистику своїх ігор.

– Гра підтримує різні типи пристроїв, включаючи телефони, планшети та комп'ютери.

Drawful 2 – це інша соціальна гра, де гравці отримують незвичайні теми для малюнків і мають зобразити їх так, щоб інші гравці змогли вгадати, що це за тема або поняття. Один гравець обирає тему та малює, а інші намагаються вгадати, що це за малюнок. Після цього гравці голосують, яка відповідь найбільш смішна або точна.

На рисунку 1.5 зображено екран з малюнком та написами, з яких треба обрати відповідь.

Основні функції аналогу Drawful 2:

– Малювання слів подібне до Scribble it!, але з додатковими функціями та можливостями.

– Гра пропонує кілька різних режимів для різноманітності та розширення ігрового процесу. Класичний режим передбачає стандартне малювання та

відгадування слів, в той час як режим з підказками надає додаткові підказки для полегшення процесу відгадування.

– Гравцям надається можливість створювати та використовувати власні набори слів та фраз для гри. Це дозволяє персоналізувати ігровий процес та адаптувати його під власні уподобання та інтереси.

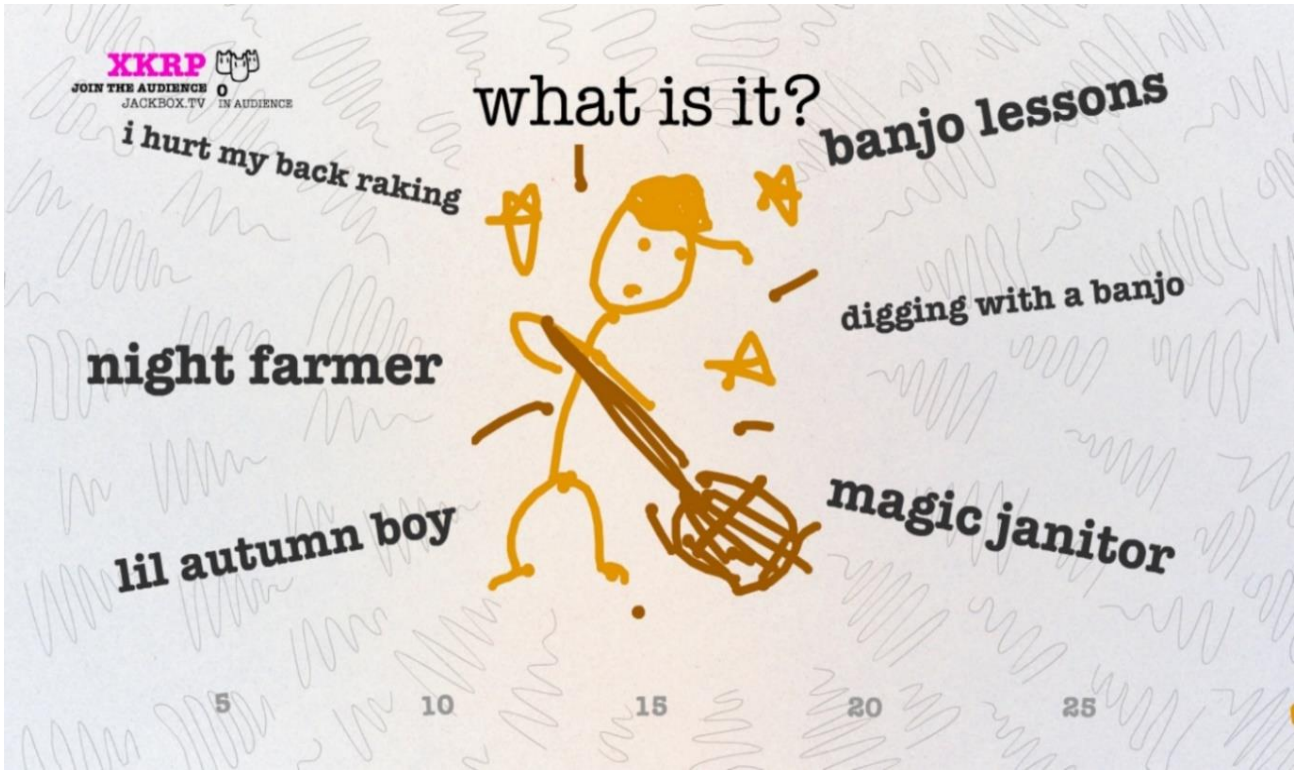


Рисунок 1.5 – Екран з вибором відповіді у “Drawful 2”

Додаткові функції:

– Професійні інструменти для малювання такі як: різні кисті, олівці, інструменти для заповнення та інші.

– Можливість відстежувати та переглядати статистику своїх ігор та покращувати власні навички малювання.

– Інтерактивні елементи: Деякі слова містять інтерактивні елементи, які можуть оживити малюнки.

Gartic Phone – це ще один варіант гри, де кожен гравець починає з фрази або малюнка, який потім інші гравці спробують відтворити або перекласти в

малюнок своєю чергою. Це продовжується, доки останній гравець намагається вгадати, що сталося зі змістом вихідного завдання.

На рисунку 1.6 зображено полотно на якому гравець малює картинку по загаданому описі.

Основні функції аналогу Gartic Phone:

– Гравці по черзі отримують слово, яке вони мають намалювати, тоді як інші гравці намагаються відгадати його.

– Gartic Phone пропонує кілька різних режимів гри для різноманітності ігрового процесу. Звичайний режим включає стандартне малювання та відгадування слів. В режим анімації гравці мають використовувати попередні малюнки, щоб намалювати наступний кадр. Є режими де використовується одне речення на всю гру, або раунд починається реченням і закінчується поясненням.

– Gartic Phone надає можливість гравцям переглядати весь ланцюжок малюнків і описів, які були створені під час гри. Крім того, гравці можуть зберегти ці результати у вигляді зображень або відео.

Рисунок 1.6 – Гравець візуалізує речення “Крокодил у відпустці”

Додаткові функції:

– Gartic Phone забезпечує користувачам простий та інтуїтивний інтерфейс, що робить гру доступною навіть для новачків.

– Гравці можуть налаштовувати рівень приватності гри, вибираючи, чи буде гра публічною або приватною.

– Можливість приєднатися до гри у режимі спостереження.

– Гра підтримує багато мов, зокрема українську.

– Доступна підтримка різних пристроїв, що дозволяє грати на телефоні, планшеті або комп'ютері.

Таблиця 1.1 – Порівняльний аналіз аналогів

Критерій оцінювання	Scribble it!	Drawful 2	Gartic Phone	Blind Crocodile
Безкоштовність	0	0	1	1
Кросс-платформеність	0	0	1	1
Оптимізація та доступність	0	0	1	1
Наявність методів порівняння малюнків	0	0	0	1
Ігровий процес та механіки	1	1	1	1
Загальна оцінка	1	1	4	5

Виходячи з результатів аналізу аналогів, застосунок Blind Crocodile має вищий сумарний коефіцієнт для Scribble it!, Drawful 2 на 80% ((100% - 1/5*100)% = 80 %)), для Gartic Phone на 20% ((100% - 4/5*100)% = 20 %)). З чого можна

зробити висновок, що застосунок Blind Crocodile може бути конкурентоспроможним та інноваційним для жанру соціальних ігор з ідентифікацією малюнків.

1.3 Аналіз методів розв’язання завдання

Для визначення найбільш оптимальних методів вирішення проблем потрібно детально розглянути існуючі рішення. Розробка мультиплеєрної 2D гри з механікою вгадування малюнків вимагає вирішення таких проблем:

- Визначити інструменти та засоби створення 2D застосунків.
- Визначити фреймворк для мережевого програмування.
- Визначити методи порівняння зображень.

При розробці 2D застосунків в ігровій сфері, одним із широко використовуваних підходів є використання графічних бібліотек, таких як SFML (Simple and Fast Multimedia Library) та SDL (Simple DirectMedia Layer) [3]. SFML надає зручний інтерфейс та підтримує кілька мов програмування, включаючи C++, C#, Python тощо. Вона дозволяє ефективно працювати з графікою, звуком, мережею та введенням користувача. З свого боку, SDL надає доступ до низькорівневих операцій з графікою та іншими мультимедійними аспектами, підтримуючи також різні мови програмування.

Unity є інтегрованим середовищем розробки (IDE), яке забезпечує можливість створення як 2D, так і 3D ігор [4]. Для розробки 2D застосунків у Unity, розробники можуть використовувати інструменти, такі як Sprite Editor для обробки та анімації спрайтів, а також Unity UI для розробки інтерфейсу користувача. Unity забезпечує зручний графічний інтерфейс, що полегшує роботу зі спрайтами, анімацією та іншими аспектами графіки.

Unreal Engine є ще однією потужною платформою для розробки ігор, яка підтримує як 2D, так і 3D ігрові середовища [5]. Для роботи з 2D графікою в Unreal Engine можна використовувати інструменти, такі як Paper2D, який дозволяє створювати та анімувати спрайти, реалізовувати фізику для 2D об'єктів

тощо. Unreal Engine надає потужні інструменти для створення графіки, фізики та інших аспектів гри.

При розробці мережевих застосунків у галузі ігор, важливим елементом є вибір підходящого фреймворку для забезпечення ефективного та надійного мережевого взаємодії. Розглянемо деякі фреймворки для мережевого програмування в контексті платформ Unity та Unreal Engine, а також незалежні фреймворки.

"Netcode for GameObjects" є новим фреймворком у Unity, призначеним для створення мережевого функціоналу [6]. Він спрощує інтеграцію мережевого функціоналу у проекти та забезпечує швидку та надійну синхронізацію об'єктів між клієнтами та сервером. Цей фреймворк дозволяє ефективно вирішувати завдання, пов'язані з мережевою синхронізацією об'єктів та дій гравців у реальному часі. Що стосується сервера, "Netcode for GameObjects" має можливість налаштування власного сервера для обробки мережевої взаємодії, але також може працювати з хмарними сервісами для спрощення розгортання.

Photon є популярним фреймворком для мережевого програмування у Unity [7]. Він надає широкі можливості для розробки розподілених мережевих додатків з підтримкою TCP/IP та UDP, аутентифікацією користувачів, синхронізацією стану гри тощо. Photon дозволяє розв'язувати багато складних завдань, пов'язаних з мережевими іграми, таких як мультиплеєрна гра з реальним часом, масштабна мережева взаємодія та інші.

Fishnet – це фреймворк для мережевого програмування у Unity, спрямований на реалізацію розподіленої архітектури мережі для ігрових додатків [8]. Він дозволяє ефективно вирішувати завдання, пов'язані з синхронізацією стану гри між клієнтами та сервером, оптимізацією мережевого трафіку та забезпеченням стабільної мережевої взаємодії.

Щодо Unreal Engine, фреймворки Unreal Networking та Unreal Engine Replication System надають потужні інструменти для реалізації мережевого функціоналу [9], зокрема синхронізацію стану гри між клієнтами та сервером, оптимізацію мережевого трафіку та вирішення багатьох складних завдань,

пов'язаних з мережевими іграми. Налаштування власних серверів для цих фреймворків може бути необхідним залежно від потреб конкретного проекту та обсягу мережевої взаємодії.

Поза платформами Unity та Unreal, існують різноманітні незалежні фреймворки для мережевого програмування, які можуть бути використані в різних проектах. Декілька прикладів таких фреймворків включають Lidgren.Network для розробки на C#, ENet для C/C++ та Netty для Java. Ці фреймворки надають різні можливості для мережевого програмування, включаючи підтримку різних протоколів, управління пакетами даних, маршрутизацію та інші функції, необхідні для створення ефективних та надійних мережевих додатків.

Методи та алгоритми порівняння зображень включають різноманітні підходи, що дозволяють визначати схожість або відмінність між двома або більше зображеннями.

Structural Similarity Index (SSI) є популярним методом для визначення схожості між зображеннями [10]. Його основними параметрами є контраст, яскравість та структурні особливості зображення. Під час порівняння двох зображень, SSI аналізує структуру об'єктів та їх розміщення на зображенні, враховуючи зміни у контрасті та яскравості. Цей метод дозволяє виявляти навіть дрібні зміни у зображеннях, що є важливим у таких галузях, як медична діагностика та комп'ютерний зір.

Feature Matching (FM) базується на виявленні особливостей зображення, таких як кути, краї та текстурні деталі. Під час порівняння, FM аналізує області зображення, де знаходяться ці особливості, та визначає їхнє подібність чи відмінність. Цей метод допомагає виявляти об'єкти на зображенні та визначати їхні характеристики, що корисно у різних додатках від відновлення зображень до розпізнавання об'єктів на зображеннях.

Histogram Comparison (HC) використовує гістограми кольорів для визначення подібності між зображеннями. Цей метод порівнює розподіл кольорів на двох зображеннях та визначає, наскільки вони схожі між собою. HC допомагає

виявляти зміни у колірній палітрі та встановлювати ступінь схожості між різними версіями зображень.

Mean Squared Error (MSE) є базовим методом порівняння зображень [11], який вимірює середньоквадратичну відстань між пікселями на двох зображеннях. Цей підхід дозволяє оцінити ступінь різниці між зображеннями: чим більше значення MSE, тим більша різниця між ними. MSE часто використовується для оцінки якості відтворення зображень або для виявлення змін у них.

Узагальнюючи, для кожної проблеми свій набір методів має переваги та недоліки, тому вибір методів залежить від конкретних потреб проекту. Так як ігрові рушії мають багато інтегрованих інструментів що спрощують роботу з 2D графікою, було вирішено використовувати ігровий рушій, від вибору якого залежить мережевий фреймворк. Для порівняння зображень підходить найпростіший із методів, а саме Mean Squared Error.

1.4 Висновки

У першому розділі нашого дослідження було проведено детальний аналіз предметної області, спрямований на визначення актуальності розробки власного застосунку у контексті соціальних ігор з механіками малювання та ідентифікації малюнків. Цей аналіз включав огляд різноманітних аналогічних застосунків, зокрема Scribble it!, Gartic Phone та Drawful 2.

В рамках порівняльного аналізу було враховано різноманітність функцій та особливостей цих ігор, їхню популярність у гравців, а також виявлено переваги та недоліки кожного з аналогів.

На основі результатів порівняльного аналізу були визначені головні аспекти, які необхідно реалізувати у власному застосунку для того, щоб зробити його кращим, доцільним та більш новаторським у своєму жанрі.

2 РОЗРОБКА МОДЕЛІ СИСТЕМИ ТА ПРИНЦИПУ РОБОТИ ПІДКАЗОК

2.1 Розробка моделі системи

Розробка моделі основної системи дозволяє зрозуміти структуру застосунку та охоплюваний ним функціонал. Це дає можливість абстрагуватися від деталей та створити абстрактне представлення системи, що включає її основні складові, функціональність, взаємозв'язки та поведінку, що важливо для правильного взаємодії користувачів з застосунком.

На рисунку 2.1 зображено UML діаграму варіантів використання застосунку Blind Crocodile [12]. Ця діаграма відображає важливі аспекти системи, ролі користувачів та їх взаємодії з системою. Вона створена для аналізу вимог і допомагає зрозуміти, як користувачі будуть взаємодіяти з застосунком і які функції вони зможуть використовувати. Такий підхід до моделювання дозволяє краще уявити структуру системи та потреби користувачів.

Під час порівняння аналогів та проектування взаємодії користувача з системою було виявлено, що великою перевагою є швидкий процес входу в гру, який вимагає малої кількості дій щоб запустити гру і приєднатися до ігрової сесії. Так як гра Blind Crocodile на зберігає статистичних даних гравця, та не має даних прогресу, тому немає змісту вимагати користувача зареєструватися чи увійти з використанням зовнішніх сервісів по типу Google чи Discord щоб зв’язати аккаунт гри для збереження даних. Але аутентифікація потрібна для використання інших мережевих сервісів, тому під час запуску гри виконується анонімна аутентифікація і гравець отримує свій унікальний ідентифікатор поки не закінчить ігрову сесію.

Після аутентифікації та отримання унікального ідентифікатору, гравець може створити власне лобі і отримати роль хоста, або під єднатися до існуючого лобі за приватним кодом, який клієнти отримують від хоста до якого хочуть підключитися. Коли лобі створюється воно за замовчуванням стає приватним,

щоб незнайомі користувачі не могли підключитися. Також за замовчуванням доступно п’ять вільних місць у лобі і тільки хост може змінити їхню кількість.

Наступним є етап підготовки до гри, в якому всі гравці знаходяться в лобі та очікують коли хост розпочне ігрову сесію. Хост у свою чергу може запустити гру коли всі підключені клієнти змінять свій статус на “Готовий”.

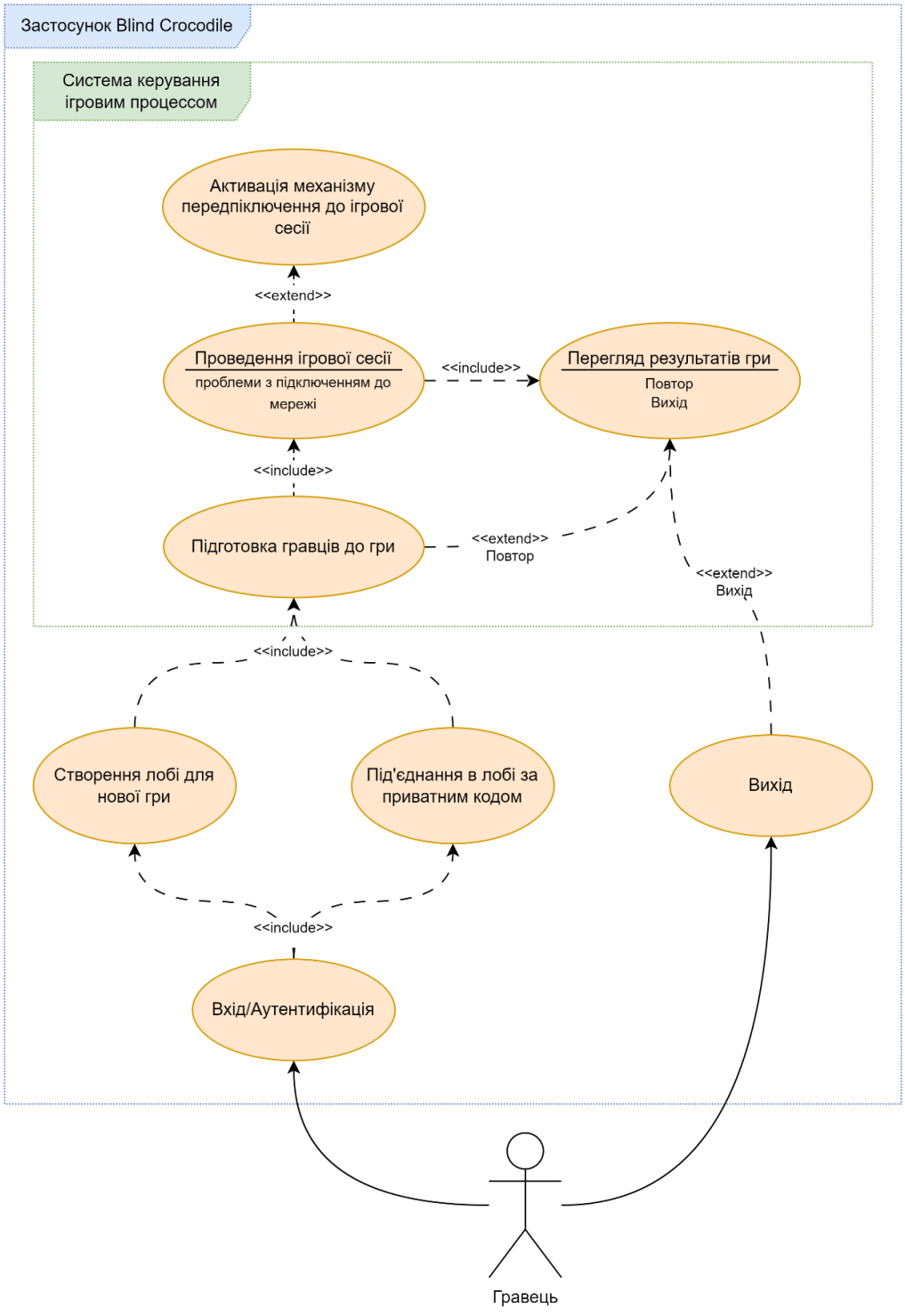


Рисунок 2.1 – UML діаграма варіантів використання

Коли хост лобі розпочинає ігрову сесію, проводиться ініціалізація гри та розподіляються ролі для гравців – “Artist” та “Guesser”, з яких “Artist” буде малювати картинку щоб використати її як оригінал, а інші з роллю “Guesser” будуть цю картинку вгадувати. Раунд розпочинається з очікування поки “Artist” намалює картинку і збереже її, після чого в інших гравців розблокується потрібний функціонал для малювання і вони зможуть почати вгадувати, і весь процес має обмежений час. Гра може закінчитися і раніше, якщо всі гравці закінчать вгадувати поки час ще не сплив.

Під час гри можуть виникати проблеми з мережею чи з’єднанням до неї, тому важливо мати механізм, який буде відслідковувати причину відключення від гри і контролювати обіг даних гравців та ігровий процес. Дана система допоможе зрозуміти лобі, як гравець чи гравці залишили гру, призупинити гру і не видаляти дані користувачів що втратили зв'язок.

Під кінець раунду гравців переміщує на екран подіуму, де відображаються результати порівняння і відповідно переможця з найбільшим результатом. Далі користувачі мають змогу продовжити гру далі, або відключитися та вийти з гри.

Діаграма варіантів використання є корисним інструментом для розуміння обсягу роботи системи зі сторони користувача. Однак важливо зазначити, що ця діаграма не розкриває детальних кроків у системі або те, які конкретні модулі використовуються в процесі взаємодії.

Для розкриття деталей роботи системи було створено діаграму послідовності, яка зображена на рисунку 2.2. Ця діаграма демонструє послідовність подій та взаємодій між різними об'єктами системи у сценарії підключення клієнта до серверу. На діаграмі послідовності показано, як користувач підключається до лобі гри та які саме сервіси використовуються під час цього процесу.

Коли запускається застосунок відбувається налагодження основних модулів та сервісів через точку входження. Одним із процесів є ініціалізація Unity сервісів, яка відбувається для відповідних встановлених пакетів у проекті, і при успішному виконанні операції дає змогу використовувати їх.

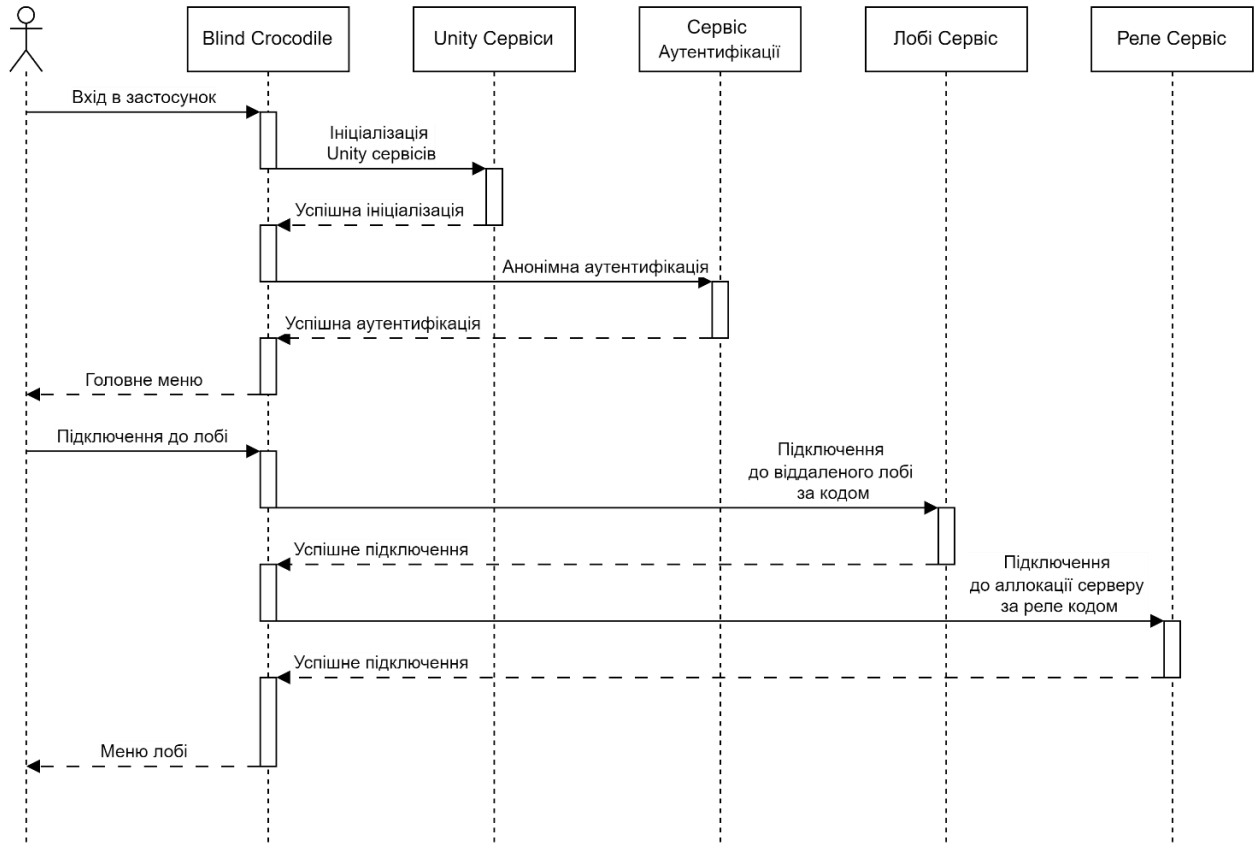


Рисунок 2.2 – UML діаграма послідовності

Після ініціалізації виконується анонімна аутентифікація, це дає змогу створити нового гравця для ігрового сеансу без будь-яких додаткових даних з його боку. Це швидкий спосіб для користувача розпочати гру. Якщо токен сеансу кешований на пристрої, тоді метод аутентифікації відновлює наявні облікові дані, незалежно від того, чи він увійшов анонімно чи через обліковий запис платформи. Якщо немає інформації про вхід гравця, цей метод створює нового анонімного. Після чого завершується запуск гри і завантажується сцена головного меню.

У сценарії коли користувач підключається до існуючого лобі, йому потрібно отримати приватний код-запрошення і за допомогою нього ініціювати процес під’єднання. Користувач вводить код у відповідне поле і натискає кнопку підключення, після чого відбувається налагодження зв’язку з віддаленим лобі використовуючи Lobby сервіс.

Коли результат підключення до віддаленого лобі успішний, у застосунок приходить об’єкт який представляє віддалене лобі і використовується для

синхронізації певних даних. Ці данні використовуються для підключення до виділених ігрових серверах через Relay сервіс, що інкапсулює логіку P2P з’єднання. Сервіс реле слугує проксі між клієнтом та сервером, забезпечує стабільну ігрову сесію, зменшуючи навантаження на основний ігровий сервер і покращуючи якість з'єднання для гравців. Результатом успішного підключення до серверу є завантаження сцени лобі.

Отже для кращого розуміння роботи програмних модулів мультиплеєрної гри було використано UML діаграми для моделювання системи. UML діаграми варіантів використання, послідовності надають візуальну репрезентацію та допомагають краще зрозуміти систему.

2.2 Розробка системи підказок для порівняння зображень

Механіка ідентифікації вимагає наявність інструментів за допомогою яких гравець зможе відслідковувати свій прогрес і покращувати результат порівняння. Так як користувач не має жодної ідеї про малюнок який він вгадує, тому стояла ціль розробити систему яка зможе візуалізувати набір даних, які представляють похибку, або різницю між двома малюнками, і допоможе швидко зорієнтуватися у тому, що потрібно виправити. Ключовими модулями які допоможуть вирішити цю проблему мають бути такі, за якими користувач зможе швидко відфільтрувати непотрібні кольори, визначити області в яких використаний не правильний колір та метод, який покаже на скільки змінився результат порівняння.

Першою частиною цієї системи є модуль для генерації індикаторів, що візуалізують співвідношення кількісті замальованих пікселів на картинці гравця в порівнянні з оригінальним зображень. Цей підхід дає змогу оцінити об’єм використання конкретного кольору, щоб на основі цих даних збільшити, зменшити або зовсім прибрати певний колір з картинки, тому в даній системі це можна вважати своєрідним фільтром.

На рисунку 2.3 зображено дві картинки: зліва показано малюнок “Сонце та хмари”, який намалював гравець, а справа оригінальний малюнок “Апельсин” який треба вгадати.

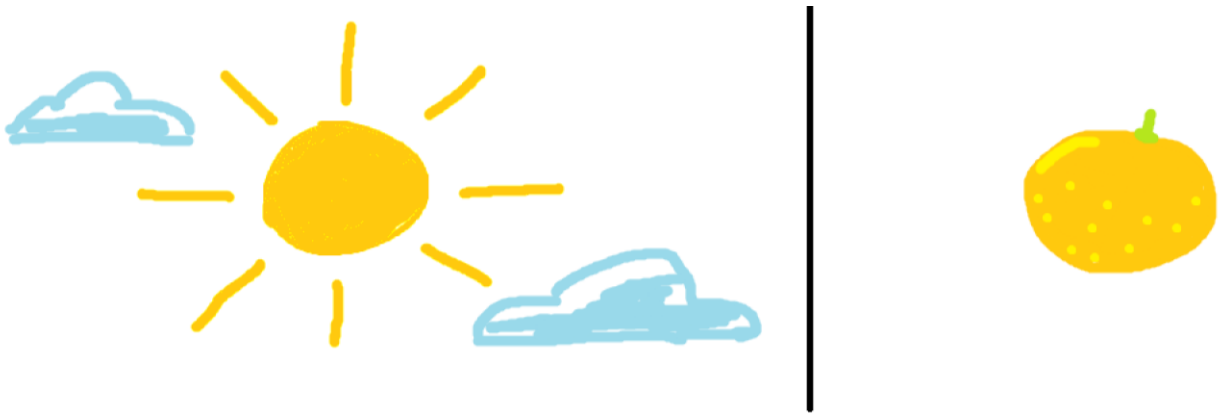


Рисунок 2.3 – Зліва – спроба користувача вгадати правий малюнок

На рисунку 2.4 зображено як буде відображатися згенеровані індикатори. Можна побачити, що оранжевого кольору використано 101%, це означає що потрібно зменшити об’єм цього кольору на малюнку, в той час як блакитного на оригіналі нема, тому індикатор має значення 0% і не замальований. Це означає що блакитний не потрібно використовувати.

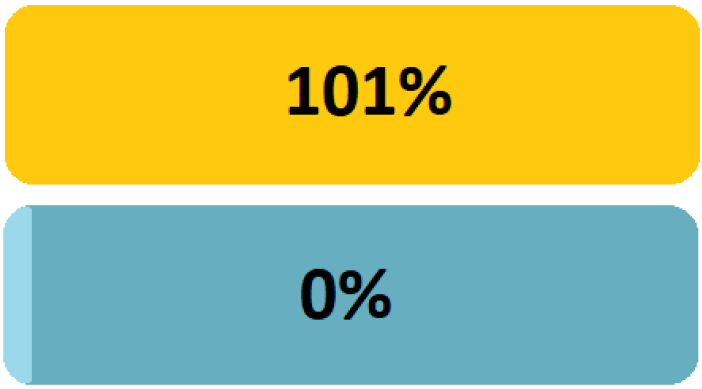


Рисунок 2.4 – Згенеровані індикатори

Наступний модуль має вирішувати проблему підказки, яка має допомогти побачити чи в правильному місці користувач намалював певний колір. Складність вирішення цієї проблеми полягає у знаходженні простого графічного методу, засобами якого можна продемонструвати різницю між двома малюнками, але й при цьому не показувати різницю всього малюнку, лише певну частину, щоб зробити процесу аналізу для гравця трохи важчим.

Під час дослідження різних існуючих методів було вирішено використовувати теплокарту. Теплова текстура (див. рисунок 2.5) [13], або heatmap texture, це графічний елемент, який використовується для візуалізації теплового поля на поверхні об'єкта в ігрових і програмних середовищах. Вона вказує на зони великого тепловиділення або інтенсивності дій об'єктів. Такі текстури використовуються для аналізу даних, візуалізації інформації або показу теплової активності, як, наприклад, відстеження кліків на веб-сайті використовує колір для показу інтенсивності кліків у певних зонах, а не для відображення точної кількості кліків. Такий підхід дозволяє швидко оцінити, де саме на сторінці було найбільше активності користувачів без детального аналізу кожного окремого кліку.

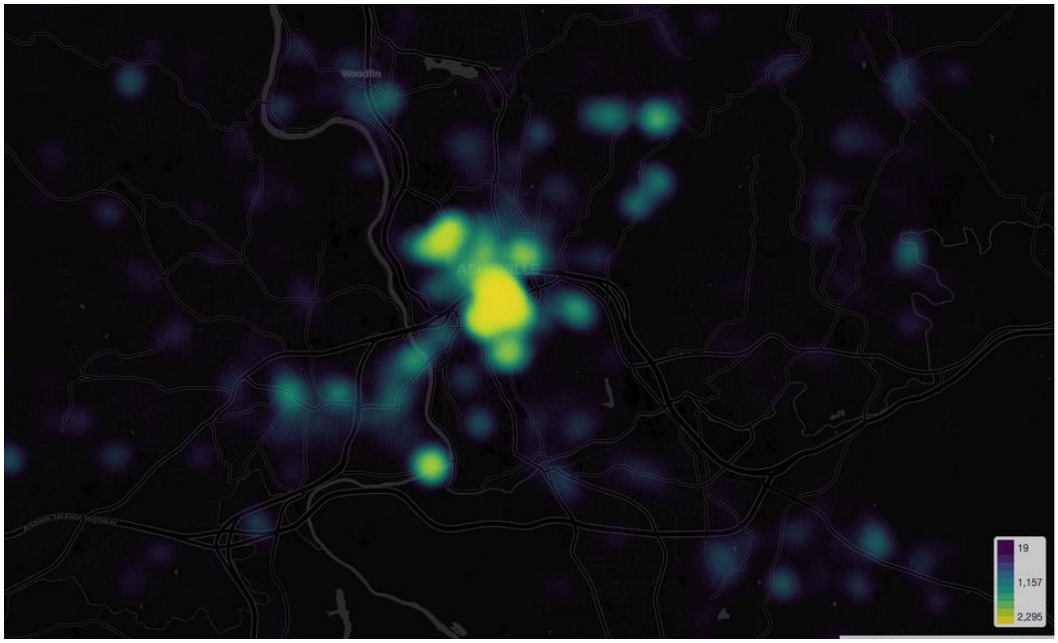


Рисунок 2.5 – Приклад теплокарти

У контексті порівняння двох малюнків даний підхід можна використати взявши за факт, що коли колір пікселя та положення співпадає з оригінальним малюнком, то використовувати зелений колір, коли піксель не співпадає – то контрастний червоний. Однак, таке порівняння пікселів за кольором є бінарним і обмежує можливість використання більш широкого спектру кольорів для відображення відмінностей між зображеннями.

Щоб порівняння дало більший діапазон даних для аналізу, можна використовувати різницю між кольорами. Найбільше підходить евклідова дистанція [14], або в комп'ютерних науках її ще називають кольорова відстань чи кольорова різниця. У просторі кольорів (наприклад, RGB), кожен колір можна представити як точку у тривимірному просторі. Таким чином, відстань між двома кольорами обчислюється як евклідова відстань між відповідними точками у цьому просторі.

Найбільша різниця між двома кольорами буде досягнута, коли вони будуть протилежними один одному на колі кольорів. Наприклад, у випадку кола кольорів у просторі RGB, колір RGB (1, 0, 0) (чистий червоний) буде протилежний кольору RGB (0, 1, 1) (синьо-зелений або ціан). Максимальну різницю між кольорами на колі зображено на рисунку 2.6, такі кольори ще називають комплементарні.

Обчислити кольорову дистанцію можна за формулою (2.1):

$$d = \sqrt{(R_2 - R_1)^2 + (G_2 - G_1)^2 + (B_2 - B_1)^2} \quad (2.1)$$

де R_1, G_1, B_1 – компоненти першого кольору в діапазоні від 0 до 1, R_2, G_2, B_2 – компоненти другого кольору в діапазоні від 0 до 1.

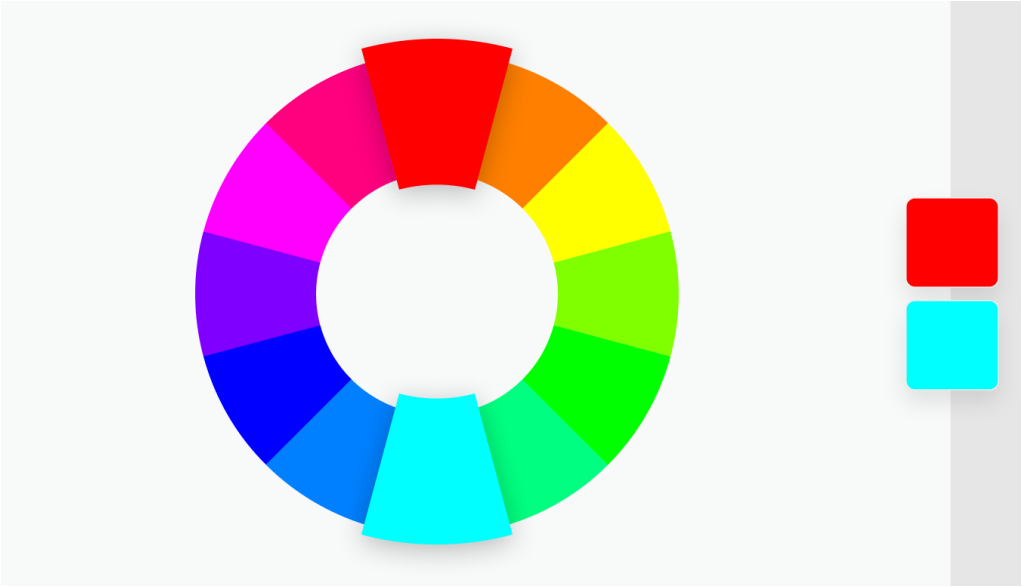


Рисунок 2.6 – Приклад комплементарних кольорів

Для того щоб можна було відобразити більше кольорів можна використати метод лінійної інтерполяції між червоним та зеленим[15]. На рисунку 2.7 показано градієнт між цими кольорами. Дуже добре видно що середина градієнту набагато темніша ніж ключові точки по бокам, що може бути проблемою для розуміння що цей колір означає на тепловій текстурі.

Рисунок 2.7 – Лінійна інтерполяція між двома кольорами

Цю проблему можна просто вирішити додавши ще один колір по центру. Жовтий дуже добре підтримує гаму і виділяє середину. Принцип лінійної інтерполяції залишається таким самим, тільки тепер буде відбуватися дві інтерполяції залежно від різниці кольорів.

Щоб використовувати кольорову відстань у формулі інтерполяції потрібно її нормалізувати – значення мають бути в діапазоні від 0 до 1. Це можна зробити взявши дистанцію яку знайшли і поділити на максимальну дистанцію. У свою чергу максимальну дистанцію можна знайти, і взяти за константу, з формули 2.1, як відстань між комплементарними кольорами, і вона буде дорівнювати 1,732. Якщо нормалізоване значення менше 0,5, то інтерполяція буде між зеленим та жовтим, якщо більше чи дорівнює 0,5 – тоді між жовтим та червоним.

Якщо нормалізоване значення буде 0,5, тоді за вищеописаним методом потрібно робити інтерполяцію між жовтим та червоним, але використання параметру 0,5 дасть нам не той колір. Для того щоб отримати жовтий колір треба нормалізувати параметр 0,5 ще раз. Так як функція інтерполяції поверне жовтий,

коли аргумент буде 0, то універсальною формулою для параметрів більше чи рівне 0,5 буде: (параметр - 0,5) * 2, а для значень менше за 0,5: параметр * 2. Дані формули перетворюють початкові параметри в діапазон від 0 до 1, при цьому зберігають їхнє відношення в межах глобального діапазону, і тоді інтерполяція буде відбуватися правильно і градієнт буде плавним (див. рисунок 2.8).

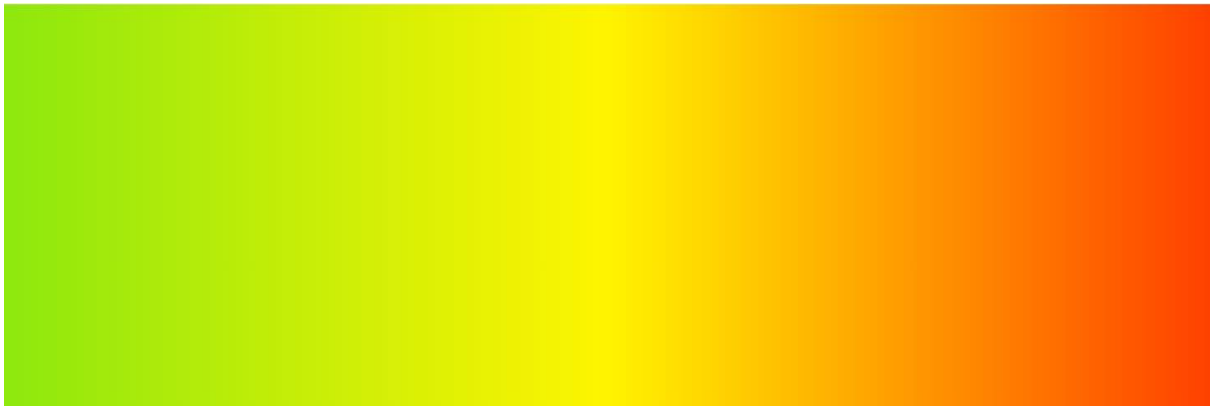


Рисунок 2.8 – Лінійна інтерполяція між трьома кольорами

На основі цих досліджень можна побудувати алгоритм, який буде через евклідову дистанцію генерувати теплову текстуру використовуючи три кольори: зелений, коли кольори співпадають на малюнку гравця з оригіналом, червоний, коли кольори максимально віддалені та жовтий, щоб додати контрасту значенням по середині.

Наступна частина системи відповідає за визначення ступеня подібності між малюнками користувача та оригінальним зображенням. Для цього використовується метод середньоквадратичної похибки (Mean Square Error, MSE), який був вибраний у попередньому розділі як відповідний для цієї задачі.

Формула 2.2 використовується для обчислення MSE. Ця формула порівнює значення пікселів у двох зображеннях і обчислює середньоквадратичну помилку між ними. Проте отримане значення MSE вказує на рівень відмінності між зображеннями, тобто наскільки вони різняться одне від одного.

Щоб отримати інформацію про ступінь подібності зображень, зазвичай застосовують обернену до MSE формулу. Таким чином, вираз 1 - MSE дозволяє

з'ясувати, наскільки зображення подібні. Чим більше значення 1 - MSE, тим більше схожість між зображеннями, оскільки це виражає відстань між ними у просторі подібності, де більше значення вказує на більшу схожість.

$$MSE = \frac{1}{h \times w} \cdot \sum_{i=1}^{h} \sum_{j=1}^{w} (I_j - T_j)^2 \quad (2.2)$$

де h та w відповідно висота і ширина зображення, I_j – значення пікселя на зображенні користувача, T_j – значення пікселя на оригінальному зображенні.

2.3 Розробка архітектури для мультиплеєрної гри

Розробка архітектури для мультиплеєрної гри є важливим етапом у процесі створення ігрового середовища, яке забезпечує не лише гладку роботу гри, але й ефективне керування та оптимізацію ресурсів. Правильно підібрана архітектура дозволяє отримати більше контролю над системою, що в свою чергу сприяє забезпеченню високої якості геймплею, зменшенню затримок та вирішенню проблем, пов'язаних зі спільною грою кількох користувачів одночасно.

Архітектура застосунку Blind Crocodile розроблена з урахуванням принципів мікросервісної архітектури та використанням машин станів для управління різними аспектами гри. Це дозволяє забезпечити високу гнучкість та модульність системи, що є критичним для розширення функціоналу.

Машина станів (state machine) – це математична модель [16], яка використовується для опису поведінки системи залежно від її поточного стану і вхідних подій. Вона складається зі скінченного набору станів, переходів між цими станами та дій, які відбуваються при переходах.

Машина станів може бути використана як основний елемент в архітектурі проекту, особливо у великих системах чи програмах, де потрібно ефективно керувати станом системи та переходами між різними її частинами. Можна виділити ключові переваги використання машини станів в архітектурі проекту:

1. Простота реалізації та розуміння: впровадження машини станів у програмний код досить просте завдяки використанню структур даних, таких як таблиці переходів або графи станів. Це сприяє швидкому розумінню логіки системи розробниками.

2. Масштабованість: машина станів легко піддається розширенню за рахунок можливості додавати нові стани та переходи. Це робить систему більш масштабованою та гнучкою для змін вимог.

3. Тестування та налагодження: чітка структура та умови переходів у машині станів сприяють ефективному тестуванню та відлагодженню. Це допомагає підвищити якість програмного продукту та забезпечити його стабільну роботу.

Один з прикладів використання машини станів у галузі геймдеву – це гра "Half-Life". У цій грі машина станів застосовується для контролю поведінки ворожих персонажів, які патрулюють, ведуть вогонь та реагують на гравця. Крім того, вона використовується для управління анімацією персонажів та їх взаємодією з оточуючим середовищем.

Інший приклад – гра "Grand Theft Auto V". Тут машина станів контролює поведінку неігрових персонажів (NPC), таких як ходьба, водіння та реагування на події. Крім того, вона управляє системами погоди, трафіку та злочинності.

У розробці застосунку "Blind Crocodile" машини станів використовуються для ефективного керування поведінкою системи та контролю переходів між різними станами. Особливо важливою є машина станів, що лежить в основі архітектури проекту. Ця машина станів визначає головні стани системи і відповідає за ініціалізацію різноманітних сервісів, сцен та обробку специфічної логіки. До цих станів відносяться:

– Початкове завантаження (Bootstrap).

– Завантаження сцени.

– Головне меню.

– Створення лобі та ініціалізація серверу.

– Підключення до існуючої гри.

– Ігровий цикл.

Друга машина станів у проекті використовується для роботи з мережею. Оскільки стан у гравця може бути тільки один, для ефективного керування мережевою логікою відповідає мережева машина станів. Цей модуль дозволяє системі ефективно взаємодіяти з мережею, керувати підключеннями, передачею даних та обробкою мережевих подій. До мережевих станів відносяться:

– Офлайн стан.

– Стан хостингу гри.

– Стан підключення до гри.

– Стан повторного підключення до гри.

Клас Game є основним класом, який слугує точкою входу для гри. Він відповідає за ініціалізацію гри та координацію роботи інших компонентів. При запуску гри, клас Game створює екземпляри GameStateMachine та NetworkStateMachine. На рисунку 2.9 зображено UML діаграму класів.

ServiceLocator – це клас, який надає механізм для реєстрації та пошуку сервісів[17]. Він виступає централізованим точкою доступу до всіх сервісів, необхідних для роботи застосунку. За допомогою ServiceLocator інші компоненти можуть легко отримати доступ до різноманітних сервісів, таких як сервіси мережевого зв'язку, управління станами, фабрики та інші. Це значно спрощує управління сервісами та дозволяє легко додавати нові або змінювати існуючі сервіси без значних змін в архітектурі.

При запуску гри, клас Game ініціалізує екземпляри GameStateMachine та NetworkStateMachine. Спочатку створюється GameStateMachine, тому що цей клас відповідає за ініціалізацію стану BootstrapState, у якому відбувається реєстрація всіх необхідних сервісів у ServiceLocator на тривалість роботи всього застосунку. У BootstrapState реєструються такі критично важливі сервіси, як менеджери ресурсів, фабрики для створення об’єктів , системи збереження даних та інші. Після завершення ініціалізації BootstrapState, GameStateMachine створює інші стани, використовуючи ServiceLocator для отримання необхідних сервісів. Кожен новий стан гри отримує доступ до зареєстрованих сервісів, що дозволяє

їм взаємодіяти з різними аспектами гри без прямої залежності один від одного. Після цього відбувається ініціалізація NetworkStateMachine. Цей клас також використовує ServiceLocator для доступу до мережевого менеджера, сервісу для керування з’єднанням, сервісу лобі та компоненту переходів між сценами.

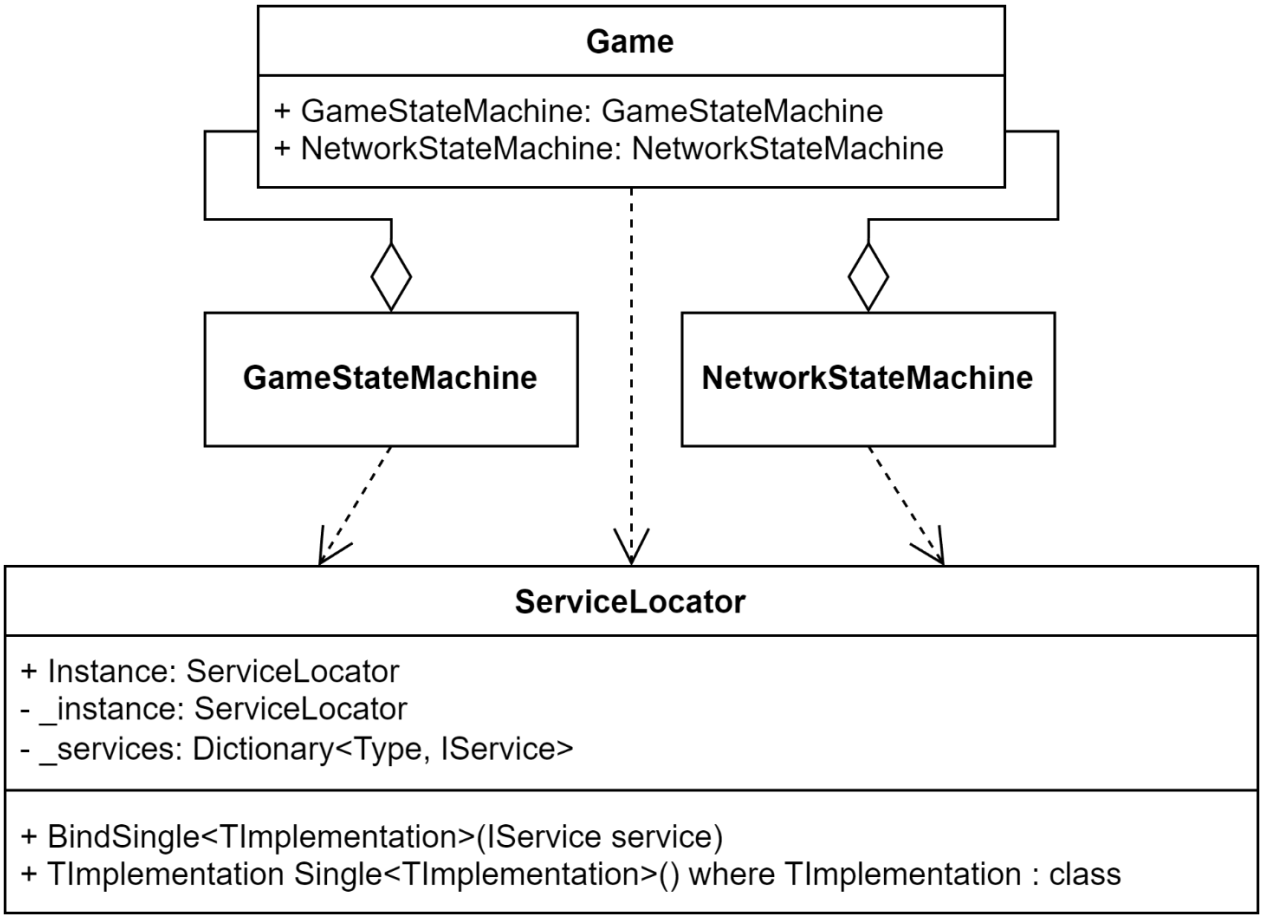


Рисунок 2.9 – UML діаграма основних класів застосунку

Архітектура застосунку Blind Crocodile базується на використанні машин станів та мікросервісній архітектурі з використанням ServiceLocator для реєстрації сервісів. Це забезпечує високу гнучкість, модульність та легкість в управлінні сервісами, що є критично важливим для сучасних додатків. Використання машин станів дозволяє чітко розділити логіку гри на окремі стани, що спрощує тестування та налагодження, тоді як локатор сервісів забезпечує централізоване управління сервісами та знижує зв'язність між компонентами.

2.4 Висновок

У другому розділі було розроблено модель системи, яка дає змогу зрозуміти важливі аспекти системи, та взаємодію користувачів з системою. Це допомагає зрозуміти, як користувачі будуть взаємодіяти з застосунком і які функції вони зможуть використовувати.

Також було продемонстровано роботу важливих сервіс застосунка, які дають змогу зі сторони розробника легко реалізувати мережеву архітектуру для налаштування з’єднання та обміном даних між сервером та клієнтами, і з сторони гравця легко використовувати відповідну систему.

Було розроблені принципи та алгоритми роботи модулів для системи підказок, яка дає змогу користувачу фільтрувати кольори та змінювати об’єм їх використання, інший модуль генерує теплову карту щоб показати правильність розміщених кольорів, і додатковий метод для визначення подібності малюка гравця з оригінальним зображенням.

3. РОЗРОБКА ПРОГРАМНИХ МОДУЛІВ

3.1 Розробка модуля для малювання

Механіка малювання є ключової в даному розроблюваному застосунку. За допомогою неї користувачі створюють вхідні дані для системи порівняння зображень і, фактично, взаємодіють з усією системою. Тому дуже важливо розробити модуль, який буде відповідати за контролювання системи введення та проектування рухів вказівника миші на полотно, що дасть змогу покращити досвід гравця користуючись інструментами для малювання.

Розробка цієї системи повинна включати низку важливих елементів. По-перше, система введення повинна бути точною та чутливою до різних швидкостей і напрямків руху миші. Це забезпечить плавність і природність малювання, що важливо для точного відтворення ліній та форм, які задумав намалювати користувач.

По-друге, алгоритми, які обробляють і перетворюють дані на зображення, повинні бути оптимізовані. Щоб відображати кожен рух вказівника на екрані миттєво, це включає використання ефективних методів для зменшення затримок і забезпечення високої частоти оновлення полотна.

По-третє, важливо дати можливість змінювати параметри малювання, такі як колір, товщина ліній. Це дозволить користувачам налаштовувати інструменти відповідно до своїх потреб, що підвищує задоволеність користувачів від використання застосунку.

Ігрові об’єкти Unity містить різні методи, які викликаються в певному порядку та з певною частотою під час обробки кадру [18]. FixedUpdate викликається з фіксованим інтервалом часу і використовується для роботи з фізикою. Цей метод гарантує стабільність фізичних обчислень, незалежно від частоти кадрів. Частота виклику FixedUpdate визначається параметром Fixed Timestep у налаштуваннях проекту Unity. Наприклад, якщо Fixed Timestep встановлено на 0.02 (що є типовим значенням), FixedUpdate викликається 50 разів на секунду. Цей метод зазвичай використовується для оновлення фізичних

об'єктів (наприклад, руху або обертання об'єктів, які мають компоненти Rigidbody), а також для будь-якої логіки, яка повинна бути синхронізована з фізичним оновленням.

Update викликається один раз за кадр і використовується для більшості логіки гри. Частота виклику цього методу залежить від продуктивності гри та апаратного забезпечення. Висока частота кадрів, наприклад, 60 FPS, означає, що Update викликається 60 разів на секунду. Цей метод зазвичай використовується для оновлення вводу користувача, наприклад обробки натискань клавіш або руху миші, та будь-якої логіки, яка не залежить від фізики, але повинна оновлюватися кожен кадр, а також анімацій та інших аспектів, що залежать від часу.

LateUpdate викликається один раз за кадр, але після того, як усі Update методи були викликані. Використовується для логіки, яка повинна виконуватися після оновлення всіх інших об'єктів у кадрі. Частота виклику LateUpdate, як і Update, залежить від частоти кадрів. Цей метод зазвичай використовується для коригування камери (наприклад, слідування за персонажем після того, як усі переміщення об'єктів були оброблені), а також для будь-якої логіки, яка повинна бути виконана після оновлення інших компонентів гри.

З описаних методів найбільше підходить Update. За замовчуванням у нього частота оновлення залежить від платформи на якій запускається застосунок, в свою чергу оновлення платформи залежить від частоти оновлення дисплея. Для більшості ігор, щоб показати плавну картинку, підходить 60 кадрів але у випадку обробки вводу користувача цього буте замало для точного відтворення рухів мишки. Для того щоб у всіх користувачів точність була однаковою, незалежно від пристроїв та їх характеристик, потрібно відключити вертикальну синхронізацію та змінити частоту оновлення гри до 144 кадрів. Відключення вертикальної синхронізації дасть змогу не синхронізувати оновлення дисплея з грою, тим самим дозволяє змінювати частоту оновлення на бажану, і Unity буде намагатися відмальовувати гру при заданій частоті оновлення. З власних досліджень було виявлено, що частота 144 кадри на секунду – це мінімальна частота, яка задовольняє точність оновлення.

Обрані методи та налаштування для зчитування вводу гравця є ефективними, але і мають певний недолік – вони не можуть обробляти рух мишки так часто, щоб отримати всі позиції на траєкторії руху вказівника. Для того щоб отримати інші позиції вказівника, і використати їх для малювання, буде достатньо апроксимації, так як відображення точної траєкторії вимагатиме більше ресурсів і буде повільніше. Для апроксимації використовується метод лінійної інтерполяції, який застосовується на поточне положення вказівника та на попереднє у минулому кадрі. Крок для інтерполяції вибраний такий, що дає змогу замалювати повністю всю лінію в залежності від товщини пензля. Розроблений метод апроксимації дає змогу отримати потрібні позиці вказівника, але, в залежності від товщини пензля, може вимагати більше 1000 ітерацій на кадр щоб намалювати лінію. Враховуючи що потрібно ще модифікувати полотно по якому малюють, і якщо це все обчислювати на стороні процесора, то буде помітно як понижується продуктивність застосунку.

Щоб підвищити ефективність малювання на полотні та покращити продуктивність застосунку, слід використовувати шейдери. Шейдери – це невеликі програми, які виконуються на графічному процесорі, що дозволяє здійснювати багато складних обчислень паралельно [19]. Це особливо корисно для графічних задач, де важлива швидкість і плавність виконання. Для оптимізації процесу малювання необхідно залишити обчислення інтерполяції позицій вказівника на центральному процесорі, а логіку замалювання перенести на графічний процесор.

У шейдерну програму необхідно передати позицію вказівника, колір та товщину пензля. Це можна зробити за допомогою графічного інтерфейсу Unity, відомого як Graphics [20]. Наприклад, метод SetFloat дозволяє змінювати товщину пензля, метод SetColor змінює його колір, а SetVector оновлює позицію вказівника. Після ініціалізації цих параметрів викликається метод Blit, який використовує шейдер для копіювання даних пікселів з однієї текстури в іншу. Спочатку полотно обробляється шейдером і копіюється в тимчасову текстуру,

після чого через метод CopyTexture дані з тимчасової текстури переносяться в основне полотно.

Таким чином, використання шейдерів дозволяє значно знизити навантаження на центральний процесор та підвищити швидкість малювання. Це особливо важливо для цього застосунку, де необхідно забезпечити миттєвий відгук на дії користувача. Завдяки паралельному виконанню обчислень на графічному процесорі, користувачі можуть малювати плавно та швидко, що покращує загальне враження від роботи.

3.2 Розробка машини станів

Машина станів є гнучким інструментом для моделювання поведінки систем, що може перебувати у скінченній кількості станів і змінювати свій стан у відповідь на деякі події. Так як концепція машини станів буде використовуватися в декількох місцях, є сенс зробити один модуль з універсальною логікою та структурою, що дасть змогу себе перевикористовувати. Даний підхід має декілька значних переваг. Універсальний модуль можна використовувати в різних частинах проекту без потреби написання унікальної логіки для кожного випадку. Використання одного модуля знижує ймовірність виникнення помилок, оскільки перевірена та протестована логіка використовується багаторазово. Також зміни в логіці чи структурі машини станів потрібно вносити тільки в одному місці, що значно спрощує підтримку та оновлення коду. Однорідний підхід до реалізації машини станів робить код зрозумілішим та легшим для читання.

Абстрактна машина станів AbstractStateMachine<TBaseState> використовує дженерики для забезпечення гнучкості та перевикористовуваності [21]. Вона оперує колекцією станів, зберігаючи активний стан та забезпечуючи механізми для входу та виходу зі станів. На рисунку 3.1 зображено UML діаграму класів абстрактної машини станів та залежних модулів.

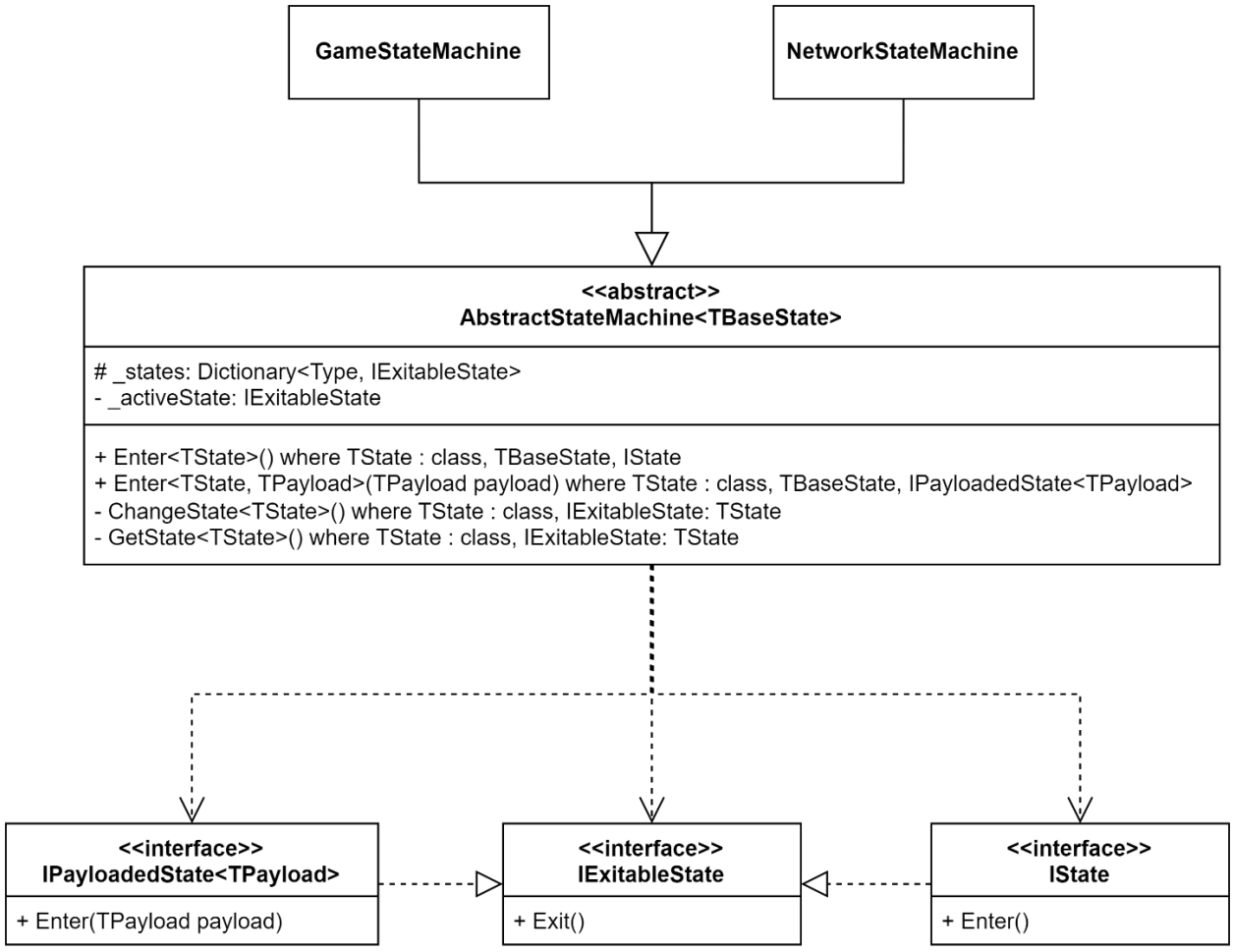


Рисунок 3.1 – UML діаграма машини станів

Основними інтерфейсами є IExitableState, що визначає метод для виходу зі стану, IState, що визначає метод для входу в стан, та IPayloadedState<TPayload>, який додає метод для входу в стан з передачею додаткових даних.

Клас AbstractStateMachine<TBaseState> зберігає всі можливі стани у словнику _states, відстежує активний стан через _activeState та має методи для входу в новий стан без додаткових даних і з передачею додаткових параметрів. Метод для входу в новий стан без параметрів використовує метод ChangeState<TState>, що змінює активний стан на новий та викликає метод входу у цей новий стан. Метод для входу в новий стан з передачею параметрів також використовує ChangeState<TState>, але викликає метод входу у стан з передачею аргументів. Метод ChangeState<TState> виходить з активного стану, якщо він

існує, змінює активний стан на новий та повертає новий стан. Метод GetState<TState> повертає стан з колекції _states на основі типу TState.

Розроблений модуль забезпечує усім необхідним функціоналом, що потребують більшість машин станів. Для того щоб створити машину станів для використання в проекті достатньо створити новий клас, що наслідує AbstractStateMachine, замість TBaseState задати інтерфейс, що буде слугувати обмеженням для типів станів. До прикладу в GameStateMachine обмеження задано на IGameState, а клас NetworkStateMachine має обмеження на INetworkState, що слугує своєрідним механізмом валідації типів станів, котрі можуть бути використані в даній машині станів.

3.3 Розробка інтерфейсу

Розробка інтерфейсу користувача є ключовим етапом створення програмного застосунку, оскільки саме інтерфейс визначає взаємодію користувача з програмою. Добре спроектований інтерфейс сприяє інтуїтивному використанню, зменшує кількість помилок та підвищує загальну задоволеність користувача.

Екран стартового меню складається з панелі на якій розміщено поле введення для ім’я гравця, а нижче знаходиться кнопка для створення лобі і праворуч від неї поле з лобі кодом і кнопкою для підключення до віддаленого лобі. На рисунку 3.2 показано графічну схему, а на 3.3 розроблений інтерфейс за цією схемою.

Елементи головного меню:

1. Робоча область.
2. Кнопка виходу з програми.
3. Панель з основними функціями меню.
4. Поле для введення ім’я.
5. Кнопка для створення лобі.
6. Поле для введення коду лобі.
7. Кнопка для підключення до лобі.

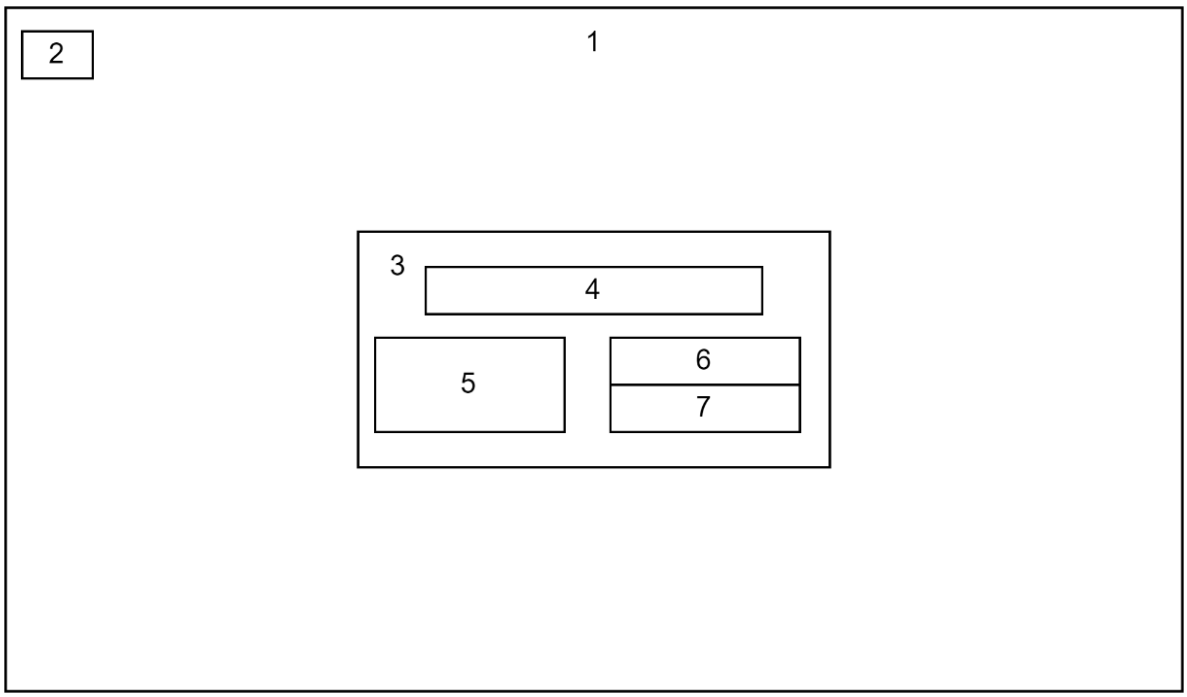


Рисунок 3.2 – Графічна схема інтерфейсу головного меню

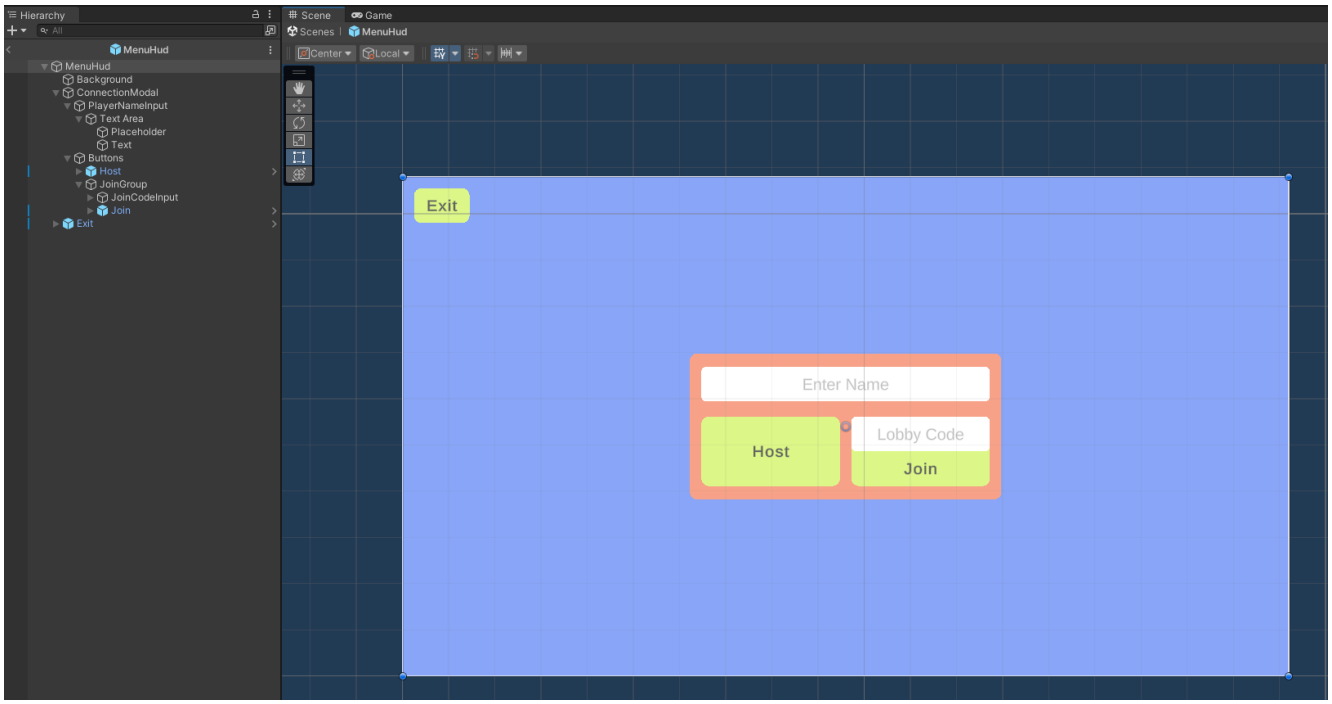


Рисунок 3.3 – Головне меню

Екран гри містить в собі декілька модальних вікон. Лобі є одним з таких вікон. На ньому міститься інформація про назву лобі, код лобі яким можна поділитися з іншими гравцями, кнопка для зміни статусу готовності, зона з

інформацією про гру та зона, де показано всіх гравців і максимальну кількість гравців у лобі разом із зайнятими місцями. На рисунку 3.4 показано графічну схему лобі, а на 3.5 розроблений інтерфейс за цієї схемою.

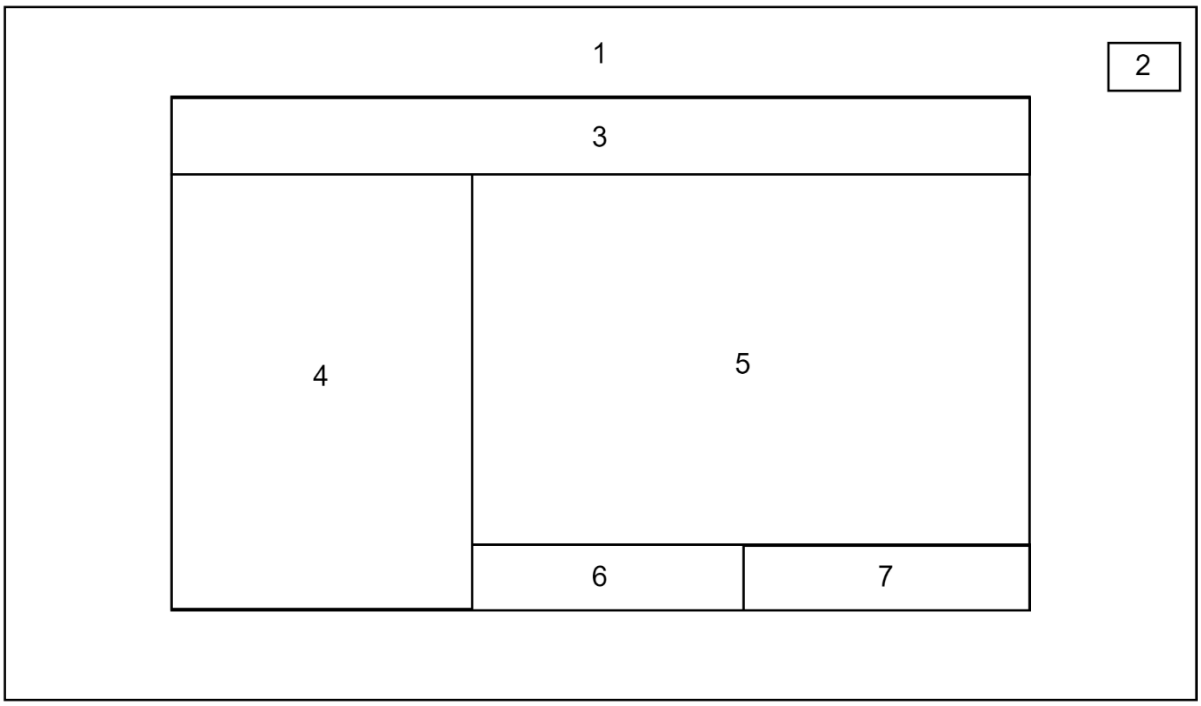


Рисунок 3.4 – Схема екрану гри з модальним вікном лобі

Елементи меню лобі:

1. Робоча область.
2. Кнопка виходу з програми.
3. Хедер модального вікна.
4. Зона зі списком усіх користувачів, та інформацією про кількість місць.
5. Область з інформацією про гру.
6. Лобі код, який можна скопіювати.
7. Кнопка для зміни статусу готовності.

Наступне модальне вікно на екрані гри містить полотно, призначене для малювання та порівняння створеної картинки з оригіналом. У цьому вікні гравцям надається доступ до 12 різних кольорів, що дозволяє вибирати відтінки для створення більш детальних і точних малюнків. Крім того є інструмент для

зміни розміру пензля, що забезпечує додаткову гнучкість у процесі малювання. Праворуч знаходиться кнопка, яка використовується для генерування підказок та інших даних для вгадування малюнків. Коли гравець входить в режим вгадування, у нього блокується можливість малювати на полотні поки він не вийде з цього режиму.

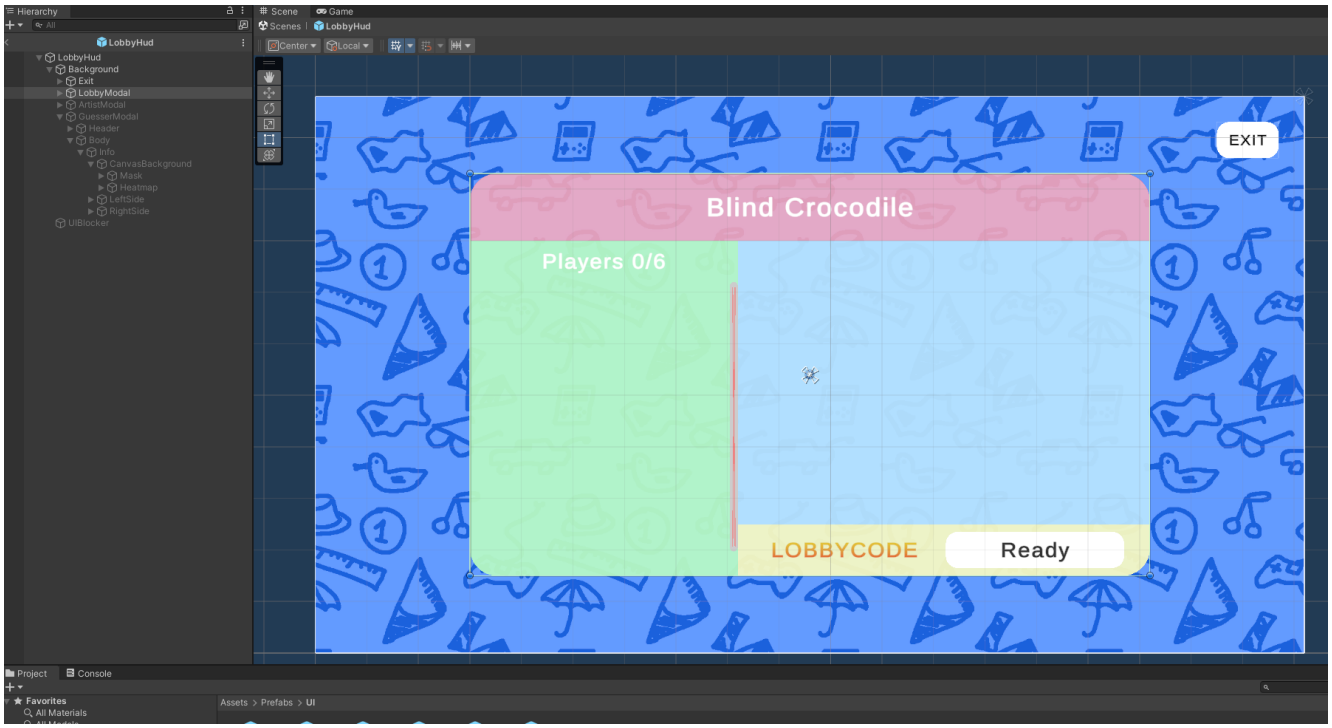


Рисунок 3.5 – Екран гри з модальним вікном лобі

На рисунку 3.6 зображено графічну схему модального вікна з полотном для малювання, а на рисунку 3.7 показано готовий інтерфейс.

Елементи вікна для малювання:

1. Робоча область.
2. Кнопка виходу з програми.
3. Хедер модального вікна.
4. Полотно для малювання.
5. Палітра з кольорами.
6. Набір розмірів пензля.
7. Кнопка для порівняння з оригінальним зображенням.

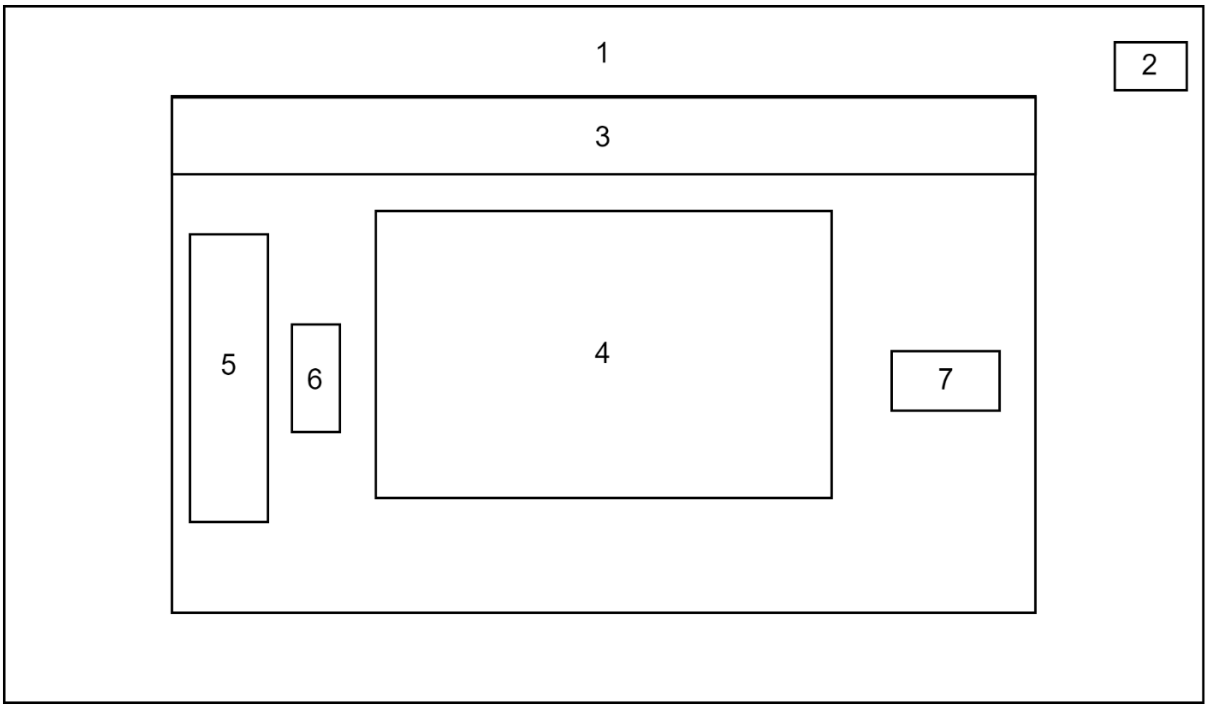


Рисунок 3.6 – Схеми модального вікна для малювання

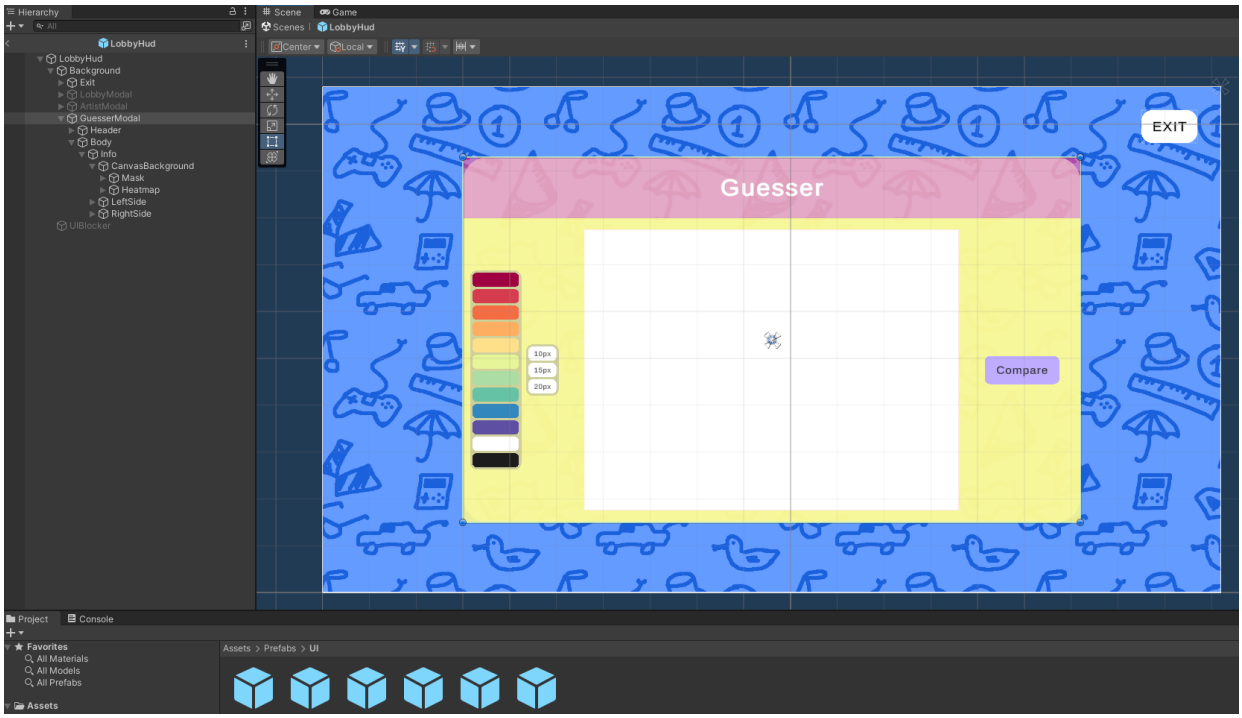


Рисунок 3.7 – Екран гри з полотном для малювання

3.4 Розробка серіалізації малюнків через мережу

У рамках розробки оптимізованого методу серіалізації малюнків через мережу в проекті, була зосереджена увага на ефективній передачі даних між

клієнтами. Хоча фреймворк Unity "Netcode for Gameobjects" вже має вбудовані засоби для передачі даних через мережу, було необхідно розробити власний метод серіалізації для оптимізації передачі графічних даних.

Так як фреймворк дає змогу синхронізувати лише типи значень, це накладає обмеження на типи посилань. Враховуючи що картинка у форматі PNG це масив байтів а у C# це тип посилань, обмін даними у такому виді стає неможливим [22]. У цьому контексті, було створено структуру даних BytesContainer, яка відповідає за серіалізацію масиву байтів, що підходить для стиснених малюнків. Ця структура дозволяє ефективно управляти та передавати масиви типів значень через мережу, зберігаючи цілісність даних та забезпечуючи стабільну передачу у мережевому середовищі. Діаграму класів представлено на зображені 3.8.

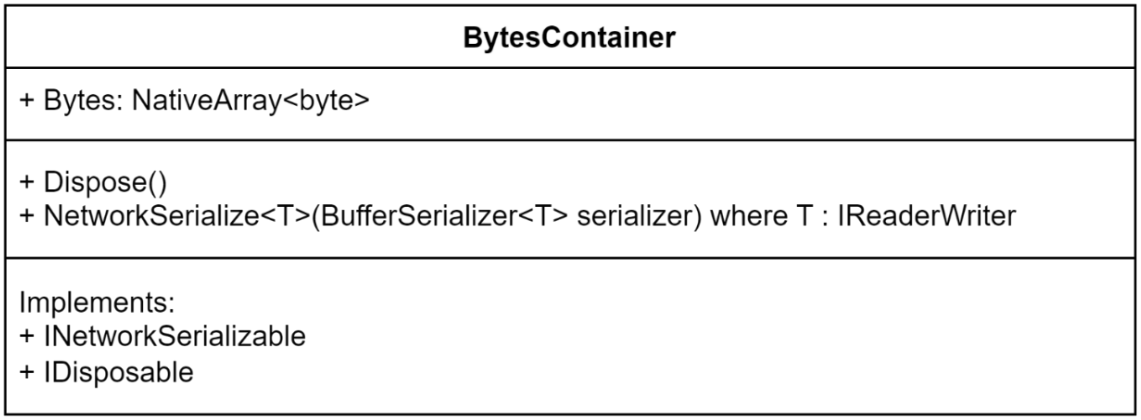


Рисунок 3.8 – UML діаграма класів структури “BytesContainer”

Особливу увагу при розробці цього типу даних було приділено оптимізації роботи з пам'яттю та швидкість обробки даних під час серіалізації та десеріалізації. Для цього було використано NativeArray – спеціалізований тип, важливий для задач із інтенсивною продуктивністю в Job System і Burst Compiler від Unity, що пропонує ефективне зберігання масивів даних у некерованій пам’яті, на відміну від сховища керованої пам’яті [23]. Забезпечує прямий доступ до пам’яті та спрощену багатопотоковість, що робить його ідеальним для

сценаріїв, які вимагають високої продуктивності та паралельної обробки. Виділення та звільнення ресурсів з пам’яті відбувається вручну.

3.5 Висновки

У третьому розділі було розроблено модуль, що відповідає за контролювання механіки малювання, в процесі створення було експериментальним методом оптимізовано даний модуль, а саме вдалось знизити навантаження обробки даних на центральному процесорі шляхом перенесення частини обчислень на графічний процесор. У рамках аналізу була ретельно розглянута машина станів та інші ключові складові системи, які використовуються для контролю та управління станами системи. Також розроблено метод передачі масиву даних через мережу, що є важливим елементом проекту з урахуванням його специфіки та вимог до ефективності передачі даних.

Зокрема було спроектовано графічні схеми екранів інтерфейсу та модальних вікон. На основі цих схем було побудовано екран головного меню, екран гри з модальним вікном лобі, та екран гри з модальним вікном що містить полотно та інші інструменти для малювання.

4 ТЕСТУВАННЯ ПРОГРАМИ

4.1 Аналіз методів тестування

У розробці програмного забезпечення, тестування відіграє ключову роль у забезпеченні якості та надійності продукту. Цей процес дозволяє виявляти помилки та дефекти, забезпечуючи виправлення їх перед випуском продукту на ринок. Важливість тестування полягає в забезпеченні відповідності програмного забезпечення вимогам замовника, уникненні негативних впливів помилок на користувачів, а також покращенні загальної якості продукту.

Юніт тестування — це перевірка окремих частин програми на коректність їх роботи. Цей метод дозволяє виявляти проблеми на ранніх етапах розробки, коли компоненти ще не інтегровані в систему. Важливою перевагою юніт тестування є можливість швидкої локалізації та виправлення помилок, що сприяє підвищенню якості коду та зниженню загальних витрат на виправлення дефектів пізніше в розробці.

Метод чорного ящика орієнтований на перевірку зовнішньої поведінки програми без знання внутрішньої структури чи алгоритмів. Він дозволяє виявляти непередбачувані результати або неочевидні помилки, що можуть виникнути у реальному використанні програми.

Метод білого ящика зосереджується на перевірці внутрішніх механізмів програми, включаючи структуру коду та логіку його роботи. Цей метод дозволяє виявляти недоліки у реалізації алгоритмів та оптимізувати роботу програми, забезпечуючи її ефективність та надійність. Цей підхід до тестування є важливим для перевірки правильності роботи складних функцій, управління пам'яттю та оптимізації продукту.

Метод сірого ящика комбінує підходи чорного та білого ящиків для комплексного аналізу програми. Цей метод дозволяє виявляти проблеми як на рівні внутрішніх механізмів, так і на рівні зовнішнього інтерфейсу користувача. Використання сірого ящика дозволяє забезпечити комплексний огляд продукту та забезпечити його оптимальну роботу в умовах реального використання.

Функціональне тестування є одним з ключових методів перевірки програмного продукту. Воно спрямоване на те, щоб переконатися, що програма виконує всі очікувані функції і завдання з точки зору користувача. Під час цього тестування перевіряються не лише основні функції, а й взаємодія з користувачем, інтерфейс, обробка даних та виведення результатів. Такий підхід дозволяє виявити помилки в роботі програми та вдосконалити її з точки зору користувача.

Інтеграційне тестування спрямоване на перевірку взаємодії між різними компонентами або модулями програми. Цей метод тестування особливо важливий у великих проектах, де різні частини системи розробляються окремо. Під час інтеграційного тестування перевіряється, чи працюють вони коректно разом, чи немає конфліктів та проблем з взаємодією, що дозволяє уникнути помилок при об'єднанні окремих компонентів в єдину систему.

Системне тестування є комплексним підходом до перевірки програмного продукту як цілісної системи. Під час цього тестування оцінюється робота програми в цілому, включаючи функціональність, продуктивність, сумісність, безпеку та інші аспекти. Важливою частиною системного тестування є перевірка відповідності програмного продукту вимогам та специфікаціям, а також виявлення і усунення будь-яких недоліків чи проблем в роботі системи.

Автоматизоване тестування є важливим елементом сучасного розробки програмного забезпечення. Воно використовує автоматизовані інструменти та скрипти для виконання тестів, що дозволяє швидко та ефективно перевіряти функціональність та якість програми. Автоматизовані тести особливо корисні для повторюваних тестів, тестування регресії, швидкого виявлення помилок та забезпечення високої якості програмного продукту.

4.2 Тестування програмного застосунку

Для тестування відмінним вибором буде функціональне тестування, оскільки цей метод дозволяє перевіряти програмне забезпечення з точки зору користувача, виявляючи потенційні проблеми та непередбачувані ситуації, які можуть виникнути в реальному використанні продукту.

Для проведення тестування було складено сценарії використання застосунку, які відповідають за ключові взаємодії між користувачем. Список тестових сценаріїв наведено в таблиці 4.1.

Таблиця 4.1 – Сценарії використання

Ідентифікатор	Назва сценарію
ВС-001	Створення та хостинг лобі
ВС-002	Під’єднання до лобі за кодом
ВС-003	Хост ініціює запуск гри коли усе лобі готове
ВС-004	Користувач може змінити колір пензля
ВС-005	Користувач може змінити розмір пензля
ВС-006	Розблокування інтерфейсу гравців “Guesser”
ВС-007	В режимі порівняння генеруються індикатори та теплокарта

Після розробки тестових сценаріїв для кожного з них було створено відповідні тест-кейси. Кожен тест-кейс детально описує кроки, необхідні для перевірки певної функціональності системи, та очікуваний результат. Це дозволяє забезпечити високий рівень покриття тестування та виявити можливі дефекти в роботі програмного забезпечення.

Перший тест-кейс перевіряє процес створення лобі в системі. Метою є перевірка можливості користувача створити лобі, отримати унікальний код лобі та забезпечити його відображення на екрані. Інші деталі наведені в таблиці 4.2.

Таблиця 4.2 – Тест-кейс ВС-001

Код тест-кейсу	ВС-001
Назва	Створення та хостинг лобі
Попередні умови	– Користувач має підключення до інтернету. – Застосунок запущений та користувач успішно аутентифікований.

Продовження таблиці 4.2

Кроки виконання	– Дочекатися завантаження головного меню. – Увести ім'я хоста у відповідне поле введення. – Натиснути кнопку "Host".
Очікуваний результат	– Система повинна створити нове лобі. – Має завантажитися нова сцена та змінитися інтерфейс. – Має з'явитися унікальний код лобі, який можна використовувати для приєднання інших гравців. – Елемент гравця, який створив лобі, має містити в собі оранжевий маркер хоста. – Елемент гравця-хоста має містити в собі зелену галочку готовності.

Фактичним результатом роботи (див. рисунок 4.1) є завантажена сцена гри з модальним вікном лобі, де на футері знаходиться приватний код, а на панелі зліва розташований список присутніх гравців у лобі і загальну кількість місць. Оранжевий маркер в кутку картки гравця означає що цей користувач є хостом лобі і автоматично отримує галочку статусу готовності.

Рисунок 4.1 – Результат створення лобі

Другий тест-кейс призначений для перевірки процесу під’єднання користувача до існуючого лобі за кодом. Метою є впевнитися, що користувач може успішно під’єднатися до лобі, введучи правильний код, та побачити інших гравців у лобі. Детальний опис наведено в таблиці 4.3.

Таблиця 4.3 – Тест-кейс ВС-002

Код тест-кейсу	ВС-002
Назва	Під’єднання до лобі за кодом
Попередні умови	– Користувач має підключення до інтернету. – Застосунок запущений та користувач успішно аутентифікований. – Лобі існує і доступне для підключення. – Користувач має код для підключення.
Кроки виконання	– Дочекатися завантаження головного меню. – Увести ім'я користувача у відповідне поле введення. – Увести код лобі у відповідне поле введення. – Натиснути кнопку "Join".
Очікуваний результат	– Система повинна під’єднати користувача до лобі. – Користувач має потрапити на екран лобі, де буде видно інших підключених гравців. – За замовчуванням елемент гравця клієнту не має зеленої галочки готовності. – Серед елементів гравців є хост з оранжевим маркером.

Фактичний результат підключення до лобі показаний на рисунку 4.2. Клієнта завантажує на нову сцену з новим інтерфейсом та модальним вікном лобі, елементи попередньо-присутніх гравців правильно відображаються зліва,

серед них є хост з оранжевим маркером також елемент підключеного граця не має зеленої галочки готовності.

Рисунок 4.2 – Результат підключення до лобі

Третій тест-кейс перевіряє можливість хоста ініціювати запуск гри після того, як всі учасники лобі підтвердять свою готовність. Метою є перевірити правильність зміни стану системи від підготовки до фактичного початку гри. Інші деталі наведені в таблиці 4.4.

Таблиця 4.4 – Тест-кейс BC-003

Код тест-кейсу	BC-003
Назва	Хост ініціює запуск гри коли усе лобі готове
Попередні умови	– Користувачі повинні бути підключені до лобі. – Лобі повинно мати мінімум двох гравців. – Всі гравці повинні бути готові до початку гри (мати зелену галочку праворуч від імені).

Продовження таблиці 4.4

Кроки виконання	– Поділитися кодом лобі з гравцями. – Дочекатися підключення гравців. – Натиснути кнопку "Ready".
Очікуваний результат	– Гра повинна початися для всіх гравців у лобі. – Одному гравцю випаде роль “Artist”, іншим випаде “Guesser”. – У користувача з роллю “Artist” має бути розблокований інтерфейс з полотном. – У користувача з роллю “Guesser” має бути заблокований інтерфейс з полотном

Фактичний результат запуску гри зображено на рисунках 4.3 та 4.4. На рисунку 4.3 зображено вікно користувача з роллю “Guesser”, весь функціонал заблоковано та інтерфейс має ефект розмитості. На рисунку 4.4 зображено вікно гравця “Artist”, інтерфейс схожий до інтерфейсу “Guesser”, тільки розблокований.

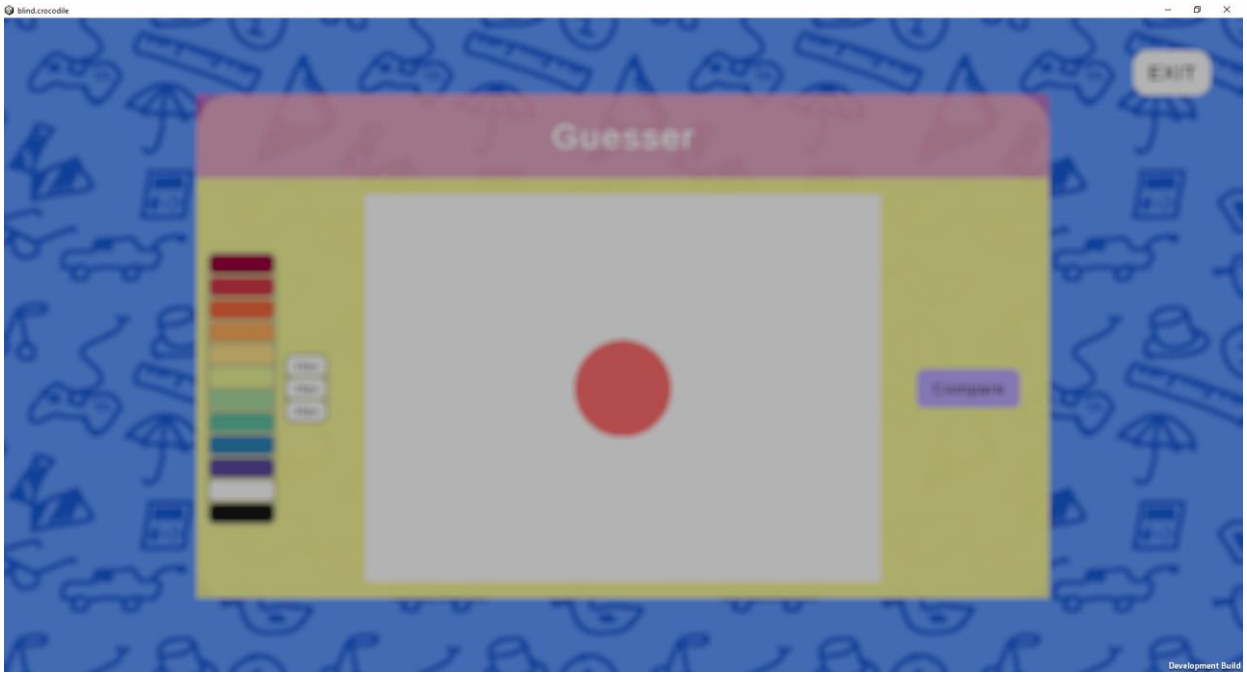


Рисунок 4.3 – Інтерфейс гравця “Guesser”

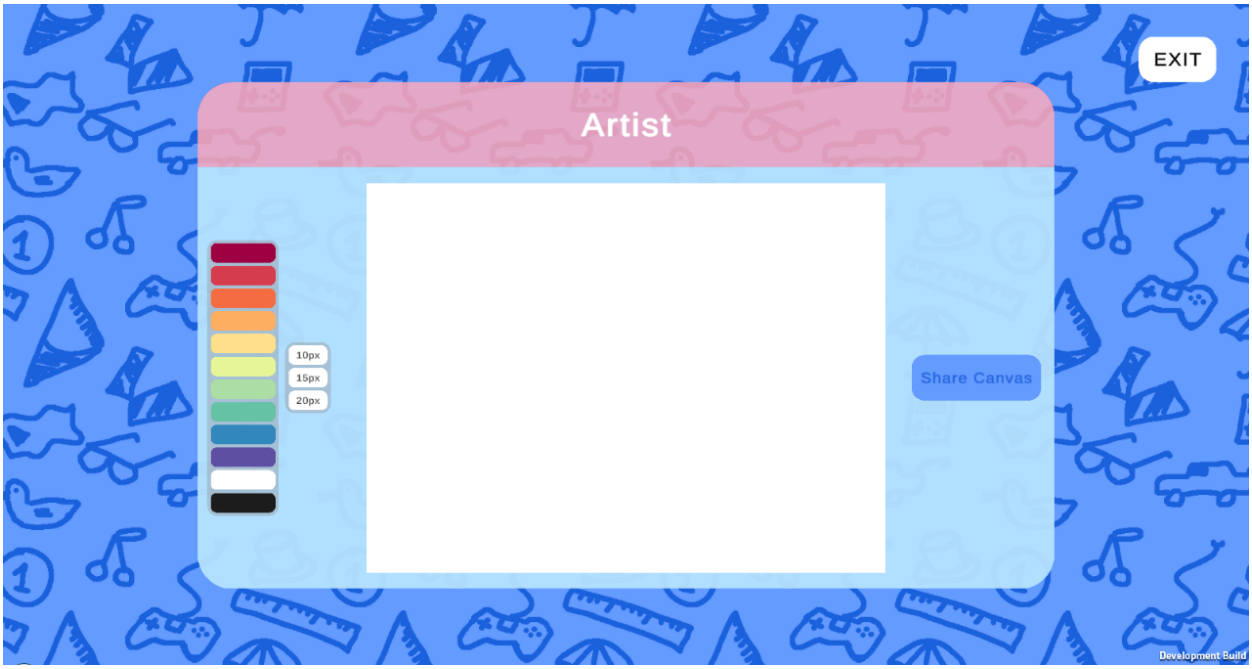


Рисунок 4.4 – Інтерфейс гравця “Artist”

Наступний, четвертий, тест-кейс перевіряє функціональність зміни кольору пензля користувачем. Метою є впевнитися, що користувач може вибрати новий колір пензля та використовувати його для малювання. Інші деталі наведені в таблиці 4.5.

Таблиця 4.5 – Тест-кейс ВС-004

Код тест-кейсу	ВС-004
Назва	Користувач може змінити колір пензля
Попередні умови	– Користувач повинен знаходитися на екрані малювання, де доступна функція зміни кольору пензля.
Кроки виконання	– Знайти ліворуч панель з кольорами. – Вибрати новий колір пензля з доступної палітри. – Переконатися, що новий колір активовано, і почати малювати на полотні.

Продовження таблиці 4.5

Очікуваний результат	– Пензель змінює свій колір на вибраний користувачем. – Користувач може малювати новим кольором на полотні. – Новий колір відображається правильно без затримок або помилок.

Фактичний результат зміни кольори пензля зображено на рисунку 4.5. Протестовано зміну кольорів пензля на панелі зліва, та як кожний з цих кольорів виглядає на полотні.

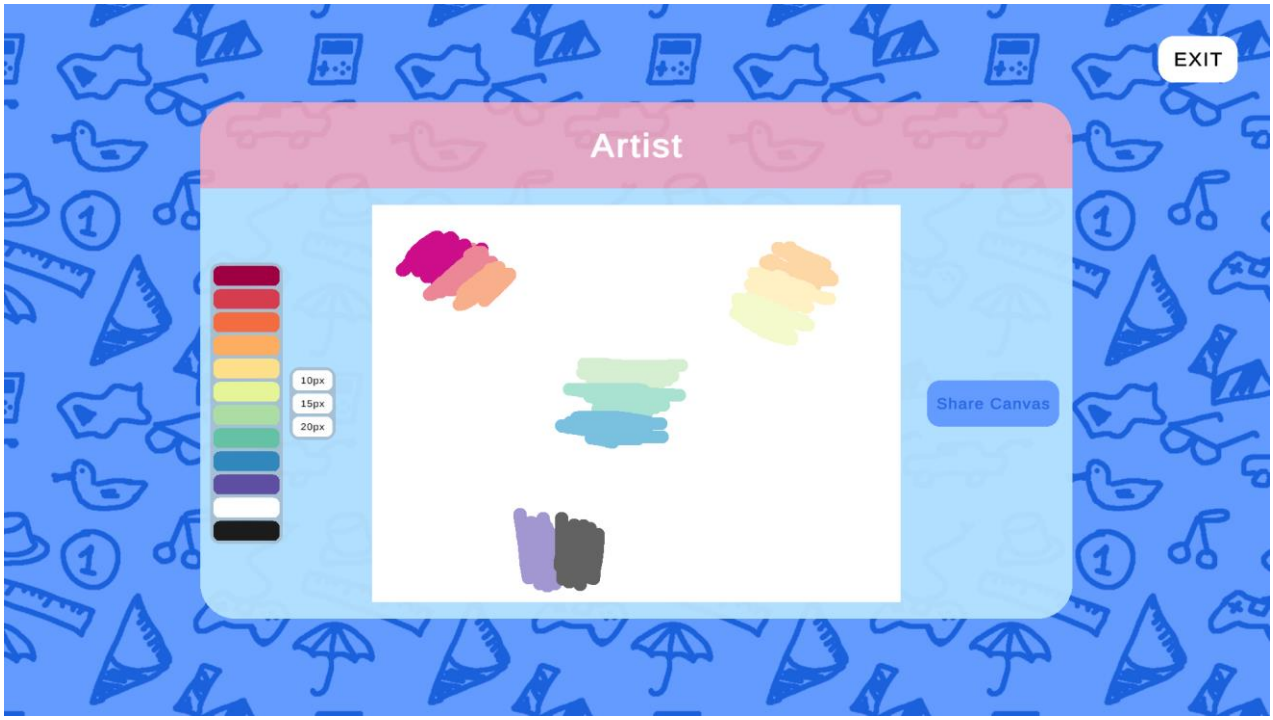


Рисунок 4.5 – Результат зміни кольору пензля

П’ятий тест-кейс перевіряє функціональність зміни розміру пензля. Метою тестування є впевнитися, що користувач може вибрати новий розмір пензля та використовувати його для малювання. Детальний опис наведено в таблиці 4.6.

Таблиця 4.6 – Тест-кейс ВС-005

Код тест-кейсу	ВС-005
Назва	Користувач може змінити розмір пензля
Попередні умови	– Користувач повинен знаходитися на екрані малювання, де доступна функція зміни розміру пензля.
Кроки виконання	– Знайти ліворуч панель з розмірами. – Вибрати новий розмір пензля. – Переконатися, що новий розмір активовано, і почати малювати на полотні.
Очікуваний результат	– Пензель змінює свій розмір на вибраний користувачем. – Розмір нових намальованих ліній змінився.

Фактичний результат зміни розміру пензля зображено на рисунку 4.6. Розміри змінюються правильно, швидкодія малювання не залежить від радіусу пензля, все працює швидко і без затримок.

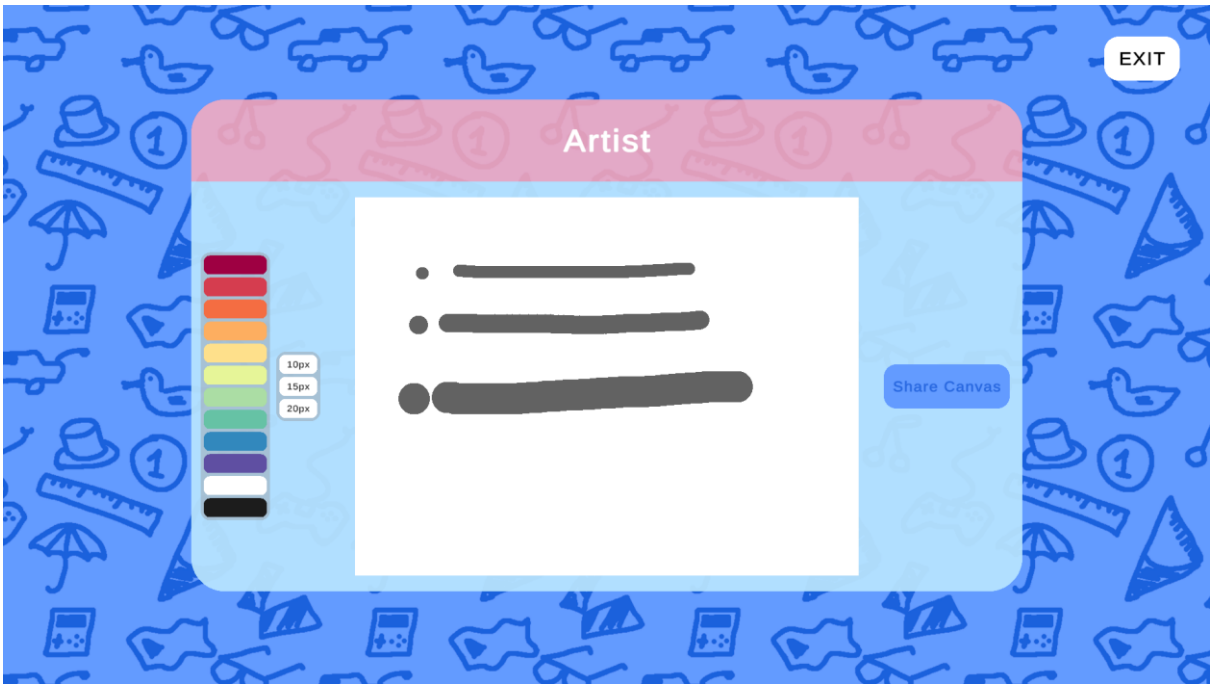


Рисунок 4.6 – Результат зміни розміру пензля

Шостий тест-кейс перевіряє функціонал розблокування інтерфейсу користувача “Guesser”, коли гравець “Artist” закінчить загадувати малюнок. Метою є перевірити, що користувач “Artist” не може ще раз поділитися загаданим малюнком з іншими користувачами, а гравці з роллю “Guesser” отримують доступ до функціоналу малювання та порівняння малюнків, коли одержать загаданий малюнок. Детальний опис наведено в таблиці 4.7.

Таблиця 4.7 – Тест-кейс ВС-006

Код тест-кейсу	ВС-006
Назва	Розблокування інтерфейсу гравців “Guesser”, коли користувач “Artist” ділиться загаданим малюнком
Попередні умови	– Гра розпочалась. – Гравець з роллю “Artis” закінчив малювати.
Кроки виконання	– Гравець з роллю “Artis” загадує малюнок. – Натиснути кнопку "Share Canvas".
Очікуваний результат	– У гравця “Artist” блокується кнопка "Share Canvas". – У гравців “Guesser” розблокується інтерфейс для малювання та порівняння малюнків.

Фактичний результат підтверджує, що після натискання кнопки "Share Canvas" у гравця "Artist" кнопка "Share Canvas" блокується (див. рисунок 4.7). Гравець далі може продовжувати малювати, але під час раунду надіслати малюнок ще раз не вийде. Інтерфейс для малювання та порівняння малюнків у гравців "Guesser" розблоковується (див. рисунок 4.8), що відповідає очікуваному результату. Це означає, що система функціонує правильно і надає користувачам відповідні можливості у встановлений стан гри.

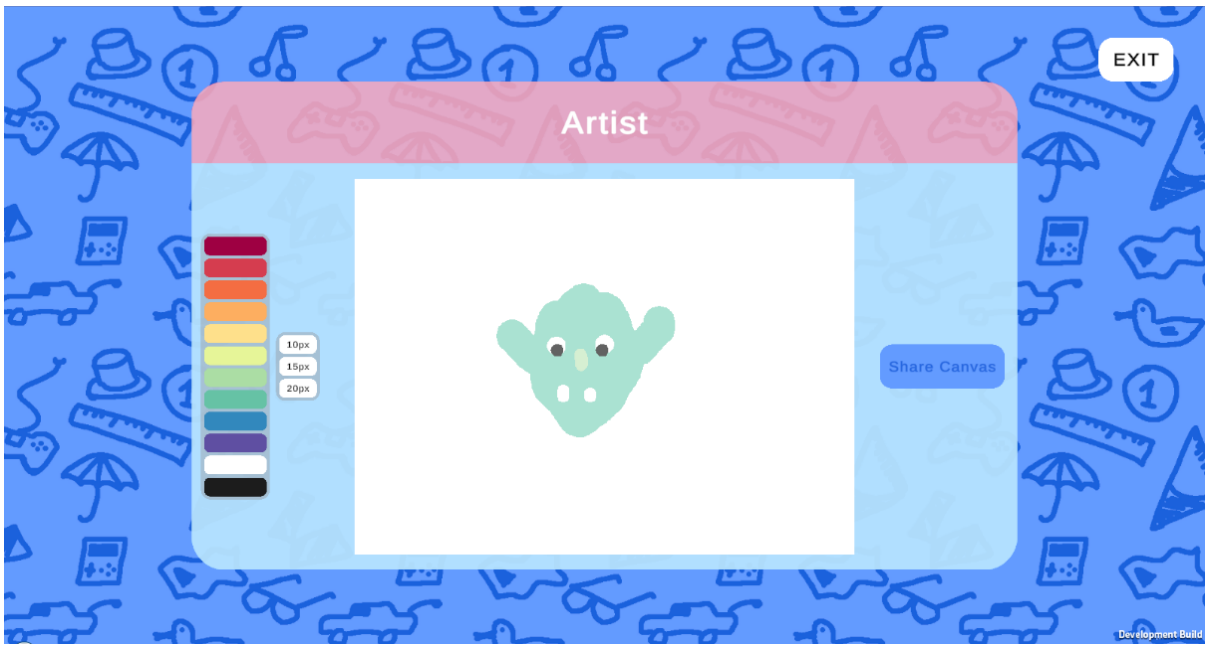


Рисунок 4.7 – Результат блокування кнопки "Share Canvas"

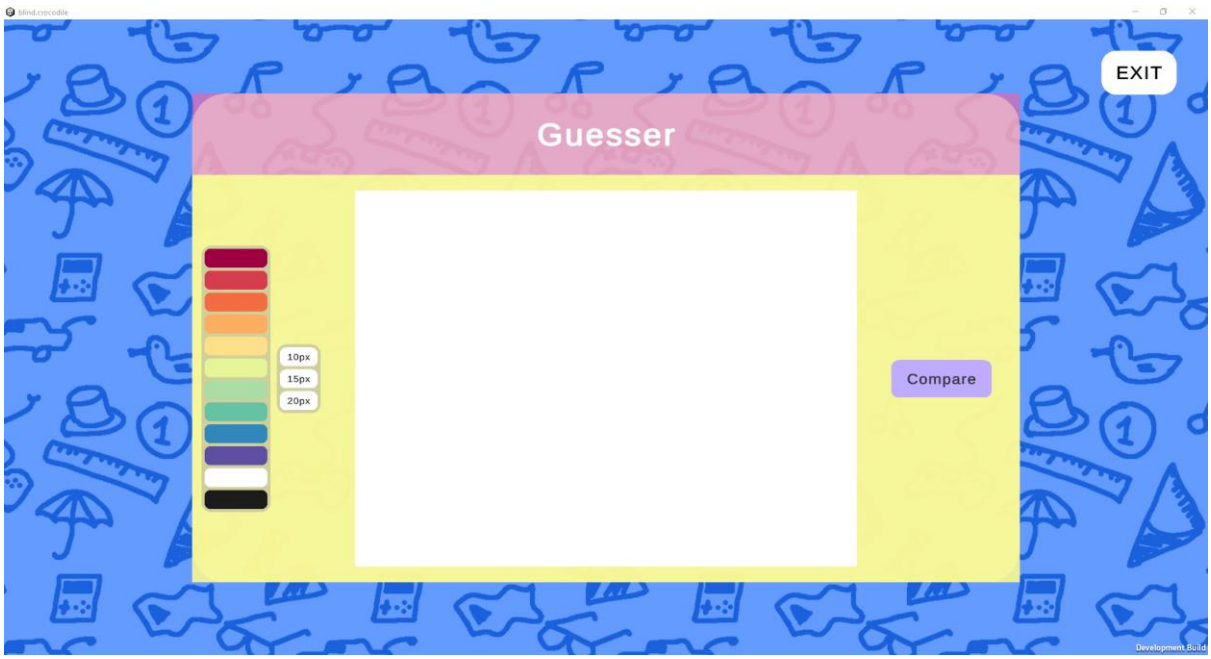


Рисунок 4.8 – Результат розблокування інтерфейсу гравця “Guesser”

Сьомий тест-кейс перевіряє функціональність генерації індикаторів та теплокарти у режимі порівняння. Метою є впевнитися, що після завершення малювання користувачем і натискання на кнопку “Compare”, інтерфейс переходить у режим порівняння, де користувач може переглядати теплокарту для аналізу малюнка. Детальний опис наведено в таблиці 4.8.

Таблиця 4.8 – Тест-кейс ВС-007

Код тест-кейсу	ВС-007
Назва	В режимі порівняння генеруються індикатори та теплокарта
Попередні умови	– Користувач отримав роль “Guesser”. – Користувач намалював будь-який малюнок.
Кроки виконання	– Натиснути кнопку "Compare". – Затиснути вказівник миші на полотні для проявлення теплокарти.
Очікуваний результат	– Зникне інтерфейс для малювання. – Зліва з’явиться список з кольорами-індикаторами. – У список індикаторів мають входити тільки ті кольори, що є на малюнку. – Справа кнопка буде мати назву “Back to Canvas”. – У місці де користувач затиснув вказівник з’явиться круглий отвір з теплокартою.

Після натискання кнопки "Compare" інтерфейс для малювання зникає, зліва з’являється список з кольорами-індикаторами, а справа кнопка набуває назви “Back to Canvas”. Також видно що кількість індикаторів відповідає кількості використаних кольорів на малюнку, що зображено на рисунку 4.9, де. показано фрукт, який складається з чотирьох кольорів, та панель з індикаторами, яка містить ті самі кольори. При затисканні вказівника миші на полотні з'являється круглий отвір з теплокартою у місці натискання (див. рисунок 4.10), що повністю відповідає очікуваному результату. Це свідчить про коректне функціонування режиму порівняння та візуалізації теплокарти, що надає користувачам інструменти для детального аналізу малюнків.

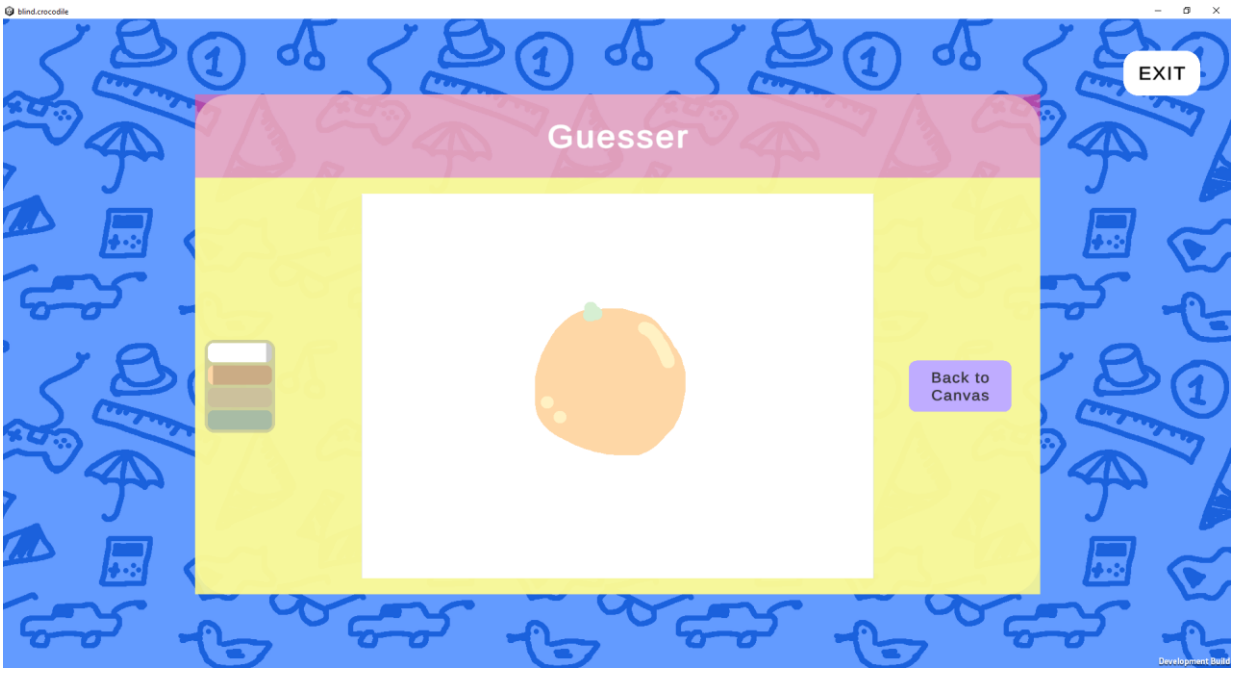


Рисунок 4.9 – Гравець ввійшов у режим порівняння

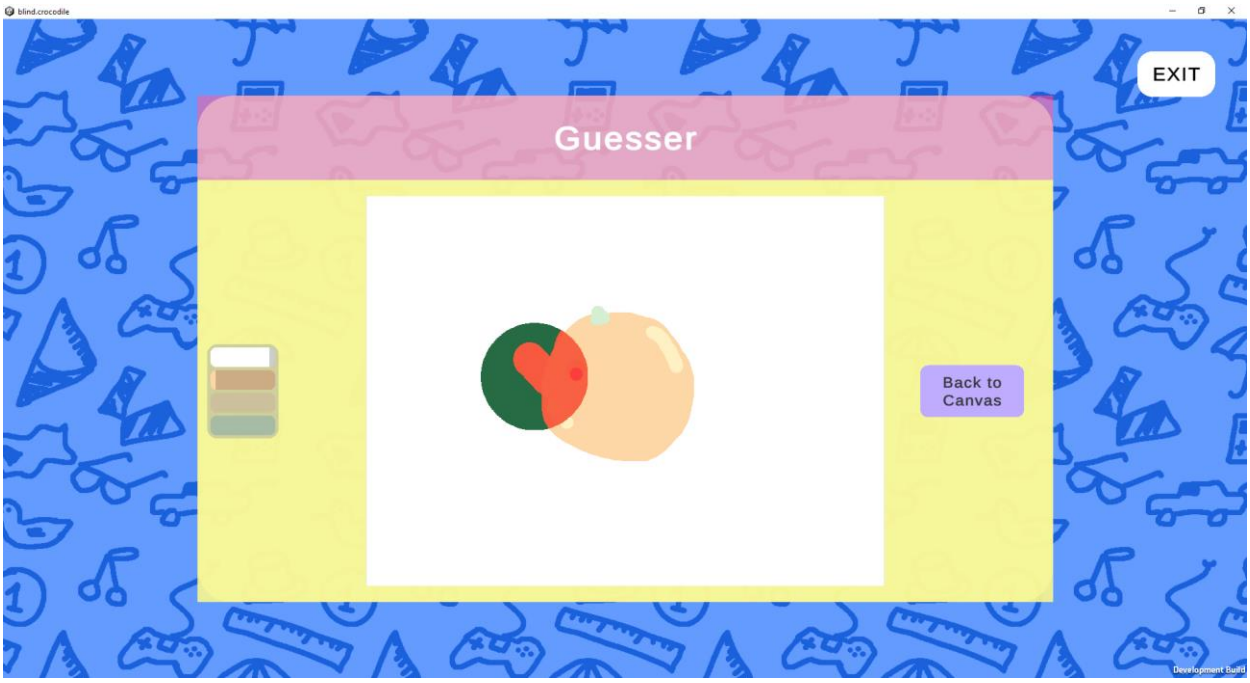


Рисунок 4.10 – Гравець перевіряє подібність по теплокарті

Таким чином, усі результати тестування збігаються з очікуваними, що свідчить про правильну роботу системи і всіх підмодулів. Основний функціонал відповідає очікуванням користувачів і працює належним чином, забезпечуючи стабільну та надійну роботу програми.

4.3 Висновки

У даному розділі було проведено аналіз методів тестування програмного застосунку Blind Crocodile. Було обрано функціональне тестування як найбільш підходящий метод для перевірки програмного забезпечення. Функціональне тестування дозволяє перевірити, чи відповідає система вимогам і очікуванням користувачів, зосереджуючись на перевірці всіх функціональних аспектів без необхідності знання внутрішньої реалізації.

Основною метою вибору функціонального тестування було забезпечення перевірки головних функціональних можливостей застосунку для впевненості у їх правильній реалізації та відповідності специфікаціям. Для цього було розроблено сценарії використання та тест-кейси. Кожен сценарій використання описував конкретну взаємодію користувача з системою, а тест-кейси детально перевіряли ці взаємодії.

Тестування охоплювало такі аспекти, як створення і підключення до лобі, ініціація запуску гри, розблокування інтерфейсу користувачів з різними ролями, та інші критичні функції. Усі тестові сценарії були успішно виконані, а результати тестування збігалися з очікуваними результатами. Це свідчить про правильну роботу системи і стабільну та надійну роботу програми.

Таким чином, результати функціонального тестування підтвердили, що Blind Crocodile готовий до експлуатації в реальних умовах, а користувачі можуть бути впевнені у його якості та ефективності.

ВИСНОВКИ

За результатами аналізу розкрито стан розвитку мультиплеєрних ігор та продемонстровано різних представників жанру соціальних ігор. Розглянуто аналоги розроблюваного застосунку та проаналізовано ключові переваги та критичні недоліки, які було враховано в процесі власної розробки. Було проаналізовано інструменти та методи вирішення задач, на основі чого прийнято рішення використовувати Unity як основний інструмент розробки, та алгоритм Mean Squared Error для визначення подібності зображень, оскільки його результати часто співпадають з оцінками, які дає людина.

Засобами моделювання UML було розроблено діаграми для візуальної репрезентації програмних модулів, що дозволяє зрозуміти систему. Діаграма варіантів використання відображає важливі аспекти системи, ролі користувачів та їх взаємодії з системою. Вона створена для аналізу вимог і допомагає зрозуміти, як користувачі будуть взаємодіяти з застосунком і які функції вони зможуть використовувати. Для розкриття певних деталей системи під час підключення клієнта до серверу було створено діаграму послідовності. Вона використовується для демонстрації зовнішніх сервісів, які відіграють важливу роль в процесі підключення, та на яких етапах вони застосовуються.

Для механіки ідентифікації малюнків було визначено та розроблено основні модулі, які використовуються для створення підказок. Перший модуль використовуються для генерації індикаторів кольорів на основі співвідношення кількісті замальованих пікселів на картинці гравця в порівнянні з оригінальним зображень, що допомагає фільтрувати та змінювати використання певних кольорів. Другий модуль відповідає за створення теплокарти, яка генерується на основі відстані між кольорами на малюнках. Для визначення відстані використовується формула евклідової дистанції. Знайдена дистанція застосовується в обчислені кольору для теплокарти у формулі інтерполяції між зеленим та червоним кольорами. Третій модуль розроблено для визначення

загального значення подібності між малюнками користувача та оригінальним зображенням, застосовуючи в обчисленнях формулу Mean Square Error.

Розроблено низькорівневу інфраструктуру застосунку, яка описує ключові системи, які контролюють життєвий цикл програми та керують внутрішніми та зовнішніми ресурсами. Архітектура застосунку базується на машинах станів та мікросервісній архітектурі. Використання машин станів дозволяє чітко розділити логіку гри на окремі стани, що спрощує тестування та налагодження, тоді як мікросервісний підхід з використанням локатору сервісів забезпечує централізоване управління сервісами та знижує зв'язність між компонентами.

Реалізовано модуль, що контролює процес малювання на полотні. Враховуючи важливість оптимізації даної механіки, так як вона є однією із головних в застосунку, було також проаналізовано способи для покращення продуктивності. В основу покращень ліг метод Update, що дає змогу відслідковувати позицію вказівника до 144 разів за секунду, метод лінійної інтерполяції для знаходження проміжних значень які не вийшло відслідкувати в методі Update, та перенесено алгоритм замальовування пікселів на графічний процесор, використовуючи шейдери, що дозволило значно знизити навантаження на центральний процесор та підвищити швидкість малювання.

Було спроектовано та розроблено інтерфейс користувача програмного застосунку. Наведено схеми таких вікон як: головне меню, меню лобі, модальне вікно з полотном для малювання, та відповідно створені вікна.

На основі функціоналу фреймворку "Netcode for Gameobjects" було створено власну структуру даних, яка використовується для синхронізації масиву байт між клієнтом та сервером. Враховуючи специфіку та обмеження даного фреймворку було використано NativeArray, що може зберігати масив даних у некерованій пам’яті, і є типом значенням, що грає важливу роль для механізмів синхронізації.

Проведено тестування програмного застосунку, що включало у себе аналіз та написання сценарії використання, написання тест-кейсів під відповідні сценарії, та власне саме тестування. Всього було протестовано сім сценаріїв:

створення та хостинг лобі, під’єднання до лобі за кодом, ініціація запуску гри, перевірка зміни кольору пензля, перевірка зміни розміру пензля, механізм розблокування інтерфейсу гравців з роллю “Guesser” та перевірка генерації індикаторів та теплокарти в режимі порівняння. Усі результати тестів збігаються з очікуваними і функціонал працює належним чином.

Результатом проведених досліджень у бакалаврській дипломній роботі є мультиплеєрна 2D-гра з механікою ідентифікації малюнків. Розроблений застосунок призначений для підвищення соціальної взаємодії за рахунок доступності під різні операційні системи та значної оптимізації, що знижує поріг системних вимог пристроїв, та покращення дедуктивного мислення за рахунок процесу ідентифікації малюків, використовуючи згенеровані підказки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Рибачок Я.А., Хошаба О.М. Аналіз методів для порівняння зображень. Молодь в науці: дослідження, проблеми, перспективи (МН-2024) [Електронний ресурс] – Режим доступу до ресурсу: https://conferences.vntu.edu.ua/index.php/mn/mn2024/paper/view/21194.

2. Multiplayer video game [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Multiplayer_video_game (дата звернення: 14.03.2024).

3. Simple and Fast Multimedia Library [Електронний ресурс] – Режим доступу до ресурсу: https://www.sfml-dev.org/ (дата звернення: 15.03.2024).

4. Unity Engine [Електронний ресурс] – Режим доступу до ресурсу: https://unity.com/products/unity-engine (дата звернення: 20.03.2024).

5. Unreal Engine [Електронний ресурс] – Режим доступу до ресурсу: https://www.unrealengine.com/en-US?sessionInvalidated=true (дата звернення: 20.03.2024).

6. Netcode for GameObjects networking framework [Електронний ресурс] – Режим доступу до ресурсу: https://docs-multiplayer.unity3d.com/netcode/current/about/index.html (дата звернення: 20.03.2024).

7. Photon networking framework [Електронний ресурс] – Режим доступу до ресурсу: https://www.photonengine.com/ (дата звернення: 21.03.2024).

8. Fishnet networking framework [Електронний ресурс] – Режим доступу до ресурсу: https://www.firstgeargames.com/ (дата звернення: 21.03.2024).

9. Unreal networking [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Networking/ (дата звернення: 21.03.2024).

10. Structural Similarity Index Measure [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/srm-mic/all-about-structural-similarity-index-ssim-theory-code-in-pytorch-6551b455541e (дата звернення: 01.04.2024).

11. Mean Squared Error [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Mean_squared_error (дата звернення: 01.04.2024).

12. Unified Modeling Language [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Unified_Modeling_Language (дата звернення: 10.04.2024).

13. Heatmap [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Heat_map (дата звернення: 20.04.2024).

14. Евклідова дистанція [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Euclidean_distance (дата звернення: 20.04.2024).

15. Лінійна інтерполяція [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Linear_interpolation (дата звернення: 20.04.2024).

16. Hassan Gomaa. Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures. Cambridge University Press. 2011 p. – 151 c.

17. Robert Nystrom. Game Programming Patterns. Genever Benning. 2014 p. – 354 c.

18. Порядок виклику функцій оновлення об’єкту [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/Manual/ExecutionOrder.html (дата звернення: 25.04.2024).

19. GLSL shader program [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/ru/2019.4/Manual/SL-GLSLShaderPrograms.html/ (дата звернення: 05.05.2024).

20. API до графічних функцій Unity [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/ScriptReference/Graphics.html (дата звернення: 05.05.2024).

21. Generic type parameters [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/uk-ua/dotnet/csharp/programming-guide/generics/generic-type-parameters (дата звернення: 08.05.2024).

22. Arrays and native containers [Електронний ресурс] – Режим доступу до ресурсу: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/serialization/arrays/ (дата звернення: 10.05.2024).

23. Job system overview [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/Manual/JobSystemOverview.html (дата звернення: 10.05.2024).

ДОДАТКИ

Додаток А – Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
Романюк О.Н.
«12» березня 2024 р.

Технічне завдання
на бакалаврську дипломну роботу «Розробка мультиплеєрної 2D-гри з механікою ідентифікації малюнків» за спеціальністю
121 – Інженерія програмного забезпечення

Керівник бакалаврської дипломної роботи:
к.т.н., доц. Хошаба О.М.
«12» березня 2024 року.
Виконав:
студент гр. ЗПІ-20Б Рибачок Я.А.
«12» березня 2024 року.

Вінниця – 2024 року

1. Найменування та галузь застосування

Бакалаврська дипломна робота: «Розробка мультиплеєрної 2D-гри з механікою ідентифікації малюнків».

Галузь застосування – розваги та медіа.

2. Підстава для розробки

Підставою для виконання бакалаврської дипломної роботи (БДР) є індивідуальне завдання на БДР та наказ №80 від «11» березня 2024 р. ректора ВНТУ про закріплення тем БДР.

3. Мета та призначення розробки

Метою роботи є підвищення соціальної взаємодії користувачів та розвиток дедуктивного мислення за рахунок створення доступної та оригінальної мультиплеєрної гри з механікою ідентифікації малюнків.

4. Вихідні дані для проведення НДР

1. Рибачок Я.А., Хошаба О.М. Аналіз методів для порівняння зображень. Молодь в науці: дослідження, проблеми, перспективи (МН-2024) [Електронний ресурс] – Режим доступу до ресурсу: https://conferences.vntu.edu.ua/index.php/mn/mn2024/paper/view/21194.
2. Hassan Gomaa. Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures. Cambridge University Press. 2011 p. – 151c.
3. Robert Nystrom. Game Programming Patterns. Genever Benning. 2014 p. – 354c.

5. Технічні вимоги

Вихідні дані до роботи: генерація теплокарти, генерація індикаторів кольорів, оптимізація процесів малювання, синхронізація текстур через мережу, визначення загальної подібності між зображеннями.

6. Конструктивні вимоги

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред'являється по закінченню робіт:

– пояснювальна записка до бакалаврської дипломної роботи;

– технічне завдання;

– лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

9. Стадії та етапи розробки:

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи
1	Аналіз стану мультиплеєрних ігор	14.03.24 – 20.03.24
2	Розробка моделі системи та архітектури	22.03.24 – 29.03.24
3	Розробка системи підказок	02.04.24 – 15.04.24
4	Розробка модулів застосунку	15.04.24 – 23.05.24
5	Тестування застосунку	24.05.24 – 26.05.24
6	Оформлення матеріалів до захисту БДР	27.05.24 – 30.05.24

10. Порядок контролю та прийняття

Виконання етапів бакалаврської кваліфікаційної роботи контролюється керівником згідно з графіком виконання роботи. Захист бакалаврської кваліфікаційної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком.

Додаток Б – Протокол перевірки на плагіат

ПРОТОКОЛ
ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ
НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: Розробка мультиплеєрної 2D-гри з механікою ідентифікації малюнків.

Тип роботи: БДР

Підрозділ : кафедра програмного забезпечення, ФІТКІ

Науковий керівник: к.т.н., доц. кафедри ПЗ Хошаба О.М.

Оригінальність	91,4%
Схожість	8,6%

Аналіз звіту подібності

■ **Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.**

□ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.

□ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку _____ Черноволик Г. О.

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck

Автор роботи _____ Рибачок Я.А.

Керівник роботи _____ Хошаба О.М.

Додаток В – Лістинг програми

```
using BlindCrocodile.Core.Services;
using Unity.Services.Authentication;
using Unity.Services.Core;
using UnityEngine;
using System.Threading.Tasks;
using BlindCrocodile.GameStates;
using Unity.Netcode;
namespace BlindCrocodile.Core
{
   public class Bootstrapper : MonoBehaviour, ICoroutineRunner
   {
      [SerializeField] private LoaderWidget _loaderWidget;
      private static Game _game;
      private async void Awake()
      {
         await InitializeUnityServices();
         SceneLoader sceneLoader = new(coroutineRunner: this);
         _game = new Game(sceneLoader, _loaderWidget, ServiceLocator.Instance,
NetworkManager.Singleton, coroutineRunner: this);
         _game.GameStateMachine.Enter<BootstrapState>();
         DontDestroyOnLoad(this);
      }
      private async Task InitializeUnityServices()
      {
         await UnityServices.InitializeAsync();
         await AuthenticationService.Instance.SignInAnonymouslyAsync();
      }
   }
}
using BlindCrocodile.Core.Services;
using BlindCrocodile.GameStates;
using BlindCrocodile.NetworkStates;
using BlindCrocodile.GameplayStates;
using Unity.Netcode;
namespace BlindCrocodile.Core
```

```
{
    public class Game
    {
        public GameStateMachine GameStateMachine { get; private set; }
        public GameplayStateMachine GameplayStateMachine { get; private set; }
        public NetworkStateMachine NetworkStateMachine { get; private set; }
        public NetworkManager NetworkManager { get; private set; }
        public Game(SceneLoader sceneLoader, LoaderWidget loaderWidget, ServiceLocator services,
NetworkManager networkManager, ICoroutineRunner coroutineRunner)
        {
            NetworkManager = networkManager;
            NetworkStateMachine = new NetworkStateMachine();
            GameStateMachine = new GameStateMachine(sceneLoader, loaderWidget, services,
NetworkStateMachine, coroutineRunner);
            NetworkStateMachine.Construct(services, NetworkManager, loaderWidget);
        }
    }
}
namespace BlindCrocodile.Core.Services
{
    public interface IService { }
}
using System.Collections.Generic;
using System;
namespace BlindCrocodile.Core.Services
{
    public class ServiceLocator
    {
        public static ServiceLocator Instance => _instance ??= new ServiceLocator();
        private static ServiceLocator _instance;
        private static readonly Dictionary<Type, IService> _services = new();
        private ServiceLocator() { }
        public void BindSingle<TImplementation>(IService service) =>
            _services.Add(typeof(TImplementation), service);
        public TImplementation Single<TImplementation>() where TImplementation : class =>
            _services[typeof(TImplementation)] as TImplementation;
    }
}
```

```
using System;
using System.Collections.Generic;
namespace BlindCrocodile.Core.StateMachine
{
    public abstract class AbstractStateMachine<TBaseState>
    {
        protected Dictionary<Type, IExitableState> _states;
        protected IExitableState _activeState;
        public void Enter<TState>() where TState : class, TBaseState, IState
        {
            TState state = ChangeState<TState>();
            state.Enter();
        }
        public void Enter<TState, TPayload>(TPayload payload) where TState : class, TBaseState,
IPayloadedState<TPayload>
        {
            TState state = ChangeState<TState>();
            state.Enter(payload);
        }
        private TState ChangeState<TState>() where TState : class, IExitableState
        {
            _activeState?.Exit();
            TState state = GetState<TState>();
            _activeState = state;
            return state;
        }
        private TState GetState<TState>() where TState : class, IExitableState =>
            _states[typeof(TState)] as TState;
    }
}
namespace BlindCrocodile.Core.StateMachine
{
    public interface IExitableState
    {
        void Exit();
    }
}
namespace BlindCrocodile.Core.StateMachine
```

```
{
    public interface IState : IExitableState
    {
        void Enter();
    }
}
namespace BlindCrocodile.Core.StateMachine
{
    public interface IPayloadedState<TPayload> : IExitableState
    {
        void Enter(TPayload payload);
    }
}
using System.Collections.Generic;
using System;
using BlindCrocodile.Services.LobbyFactory;
using BlindCrocodile.Services.MenuFactory;
using BlindCrocodile.Core.Services;
using BlindCrocodile.Core;
using BlindCrocodile.Core.StateMachine;
using BlindCrocodile.Services.Factories;
using BlindCrocodile.NetworkStates;
namespace BlindCrocodile.GameStates
{
    public class GameStateMachine : AbstractStateMachine<IGameState>
    {
        // BootstrapState - initialize game
        // LoadMenuState
        // MenuState - join or host the game
        // LoadLobbyState - separate for host and clients (HostGameState/JoinGameState)
        // WaitingLobbyState - wait all the players to be ready. Start countdown and begin the game
        // GameLoopState - handle game logic
        // RoundEndState - showroom for winners
        public GameStateMachine(SceneLoader sceneLoader, LoaderWidget loaderWidget,
ServiceLocator services, NetworkStateMachine networkStateMachine, ICoroutineRunner coroutineRunner)
        {
            _states = new Dictionary<Type, IExitableState>()
            {
```

```
                [typeof(BootstrapState)] = new BootstrapState(this, services, coroutineRunner, networkStateMachine),
                [typeof(LoadMenuState)] = new LoadMenuState(this, services.Single<IMenuFactory>(), sceneLoader, loaderWidget),
                [typeof(MenuState)] = new MenuState(this),
                [typeof(HostGameState)] = new HostGameState(this, services.Single<ILobbyFactory>(), services.Single<INetworkFactory>(), sceneLoader, loaderWidget, networkStateMachine),
                [typeof(JoinGameState)] = new JoinGameState(this, services.Single<ILobbyFactory>(), services.Single<INetworkFactory>(), sceneLoader, loaderWidget, networkStateMachine),
                [typeof(WaitingLobbyState)] = new WaitingLobbyState(this, sceneLoader),
                [typeof(GameLoopState)] = new GameLoopState(this),
            };
        }
    }
}
using BlindCrocodile.Core.StateMachine;
using System.Collections.Generic;
using System;
using Unity.Netcode;
using BlindCrocodile.Core.Services;
using BlindCrocodile.Services.Network;
using System.Text;
using BlindCrocodile.Lobbies;
using System.Threading.Tasks;
using UnityEngine;
using BlindCrocodile.Core;
using NetworkPlayer = BlindCrocodile.Services.Network.NetworkPlayer;
namespace BlindCrocodile.NetworkStates
{
    public class NetworkStateMachine : AbstractStateMachine<INetworkState>
    {
        public void Construct(ServiceLocator services, NetworkManager networkManager, LoaderWidget loaderWidget)
        {
            _states = new Dictionary<Type, IExitableState>()
            {
                // Authenticate state
                [typeof(OfflineState)] = new OfflineState(),
```

```
                [typeof(HostState)] = new HostState(networkManager, this,
services.Single<ILobbyService>(),
                    services.Single<INetworkService>(), loaderWidget),
                [typeof(JoinState)] = new JoinState(networkManager, this,
services.Single<ILobbyService>(),
                    services.Single<INetworkService>(), loaderWidget),
            };
        }
    }
    public class HostState : INetworkState, IPayloadedState<NetworkPlayersLobby> //
StartHostState | HostingState ?
    {
        private const string LOBBY_NAME = "GameLobby";
        private const int MAX_CONNECTIONS = 5;
        private readonly NetworkStateMachine _networkStateMachine;
        private readonly NetworkManager _networkManager;
        private readonly INetworkService _networkService;
        private readonly ILobbyService _lobbyService;
        private readonly LoaderWidget _loaderWidget;
        private NetworkPlayersLobby _networkPlayerView;
        public HostState(NetworkManager networkManager, NetworkStateMachine
networkStateMachine,
            ILobbyService lobbyService, INetworkService networkService, LoaderWidget
loaderWidget)
        {
            _networkManager = networkManager;
            _networkStateMachine = networkStateMachine;
            _lobbyService = lobbyService;
            _networkService = networkService;
            _loaderWidget = loaderWidget;
        }
        public async void Enter(NetworkPlayersLobby payload)
        {
            Debug.Log("HostState");
            _networkPlayerView = payload;
            // sub on server callbacks
            _networkManager.ConnectionApprovalCallback += OnApprovalCheck;
            _networkManager.OnClientConnectedCallback += OnClientConnected;
```

```
    _networkManager.OnServerStopped += OnServerStopped;
    // do some connection stuff
    //_loaderWidget.Show();
    await StartHostAsync();
    //_loaderWidget.Hide();
}
public void Exit()
{
    // unsub on server callbacks
    _networkManager.ConnectionApprovalCallback -= OnApprovalCheck;
    _networkManager.OnClientConnectedCallback -= OnClientConnected;
    _networkManager.OnServerStopped -= OnServerStopped;
}
private async Task StartHostAsync()
{
    try
    {
        await _lobbyService.CreateLobbyAsync(LOBBY_NAME, MAX_CONNECTIONS);
        await _networkService.StartHostAsync(MAX_CONNECTIONS);
    }
    catch
    {
        SessionData.ClearPlayers();
        _networkStateMachine.Enter<OfflineState>();
    }
}
private void OnServerStopped(bool _)
{
    SessionData.ClearPlayers();
    _networkStateMachine.Enter<OfflineState>();
}
private void OnApprovalCheck(NetworkManager.ConnectionApprovalRequest request,
    NetworkManager.ConnectionApprovalResponse response)
{
    string lobbyId = Encoding.UTF8.GetString(request.Payload);
    SessionData.AddPlayer(request.ClientNetworkId, lobbyId);
    Debug.Log("connection approved");
    response.Approved = true;
```

```
            response.Pending = false;
        }
        private void OnClientConnected(ulong clientId)
        {
            string lobbyId = SessionData.ConnectedPlayerIds[clientId];
            Debug.Log("client lobby id: " + lobbyId);
            _networkPlayerView.Players.Add(
                new NetworkPlayer(
                    clientId,
                    _lobbyService.LocalLobby.Players[lobbyId].Name,
                    lobbyId,
                    _lobbyService.LocalLobby.Players[lobbyId].IsHost,
                    PlayerRole.Guesser,
                    Array.Empty<byte>().ToBytesContainer()));
        }
    }

    public class JoinState : INetworkState, IPayloadedState<string>
    {
        private readonly NetworkManager _networkManager;
        private readonly NetworkStateMachine _networkStateMachine;
        private readonly INetworkService _networkService;
        private readonly ILobbyService _lobbyService;
        private readonly LoaderWidget _loaderWidget;
        public JoinState(NetworkManager networkManager, NetworkStateMachine
networkStateMachine,
            ILobbyService lobbyService, INetworkService networkService, LoaderWidget
loaderWidget)
        {
            _networkManager = networkManager;
            _networkStateMachine = networkStateMachine;
            _lobbyService = lobbyService;
            _networkService = networkService;
            _loaderWidget = loaderWidget;
        }
        public void Enter(string lobbyCode)
        {
            Debug.Log("JoinState");
```

```
            _networkManager.OnServerStopped += OnServerStopped;
            //_loaderWidget.Show();
            Task.Run(async () => await JoinServerAsync(lobbyCode));
            //await JoinServerAsync(lobbyCode);
            //_loaderWidget.Hide();
        }
        private async Task JoinServerAsync(string lobbyCode)
        {
            await _lobbyService.JoinLobbyByCodeAsync(lobbyCode);
            await _networkService.JoinServerAsync();
        }
        public void Exit()
        {
            _networkManager.OnServerStopped -= OnServerStopped;
        }
        private void OnServerStopped(bool _) =>
            _networkStateMachine.Enter<OfflineState>();
    }
    public abstract class AbstractOnlineState<TPayload> : INetworkState, IPayloadedState<TPayload>
    {
        public abstract void Enter(TPayload payload);
        public abstract void Exit();
    }
}
using System.Collections;
using UnityEngine;
namespace BlindCrocodile.Core
{
    public interface ICoroutineRunner
    {
        Coroutine StartCoroutine(IEnumerator routine);
        void StopCoroutine(Coroutine routine);
    }
}
using Unity.Netcode.Transports.UTP;
using Unity.Netcode;
using BlindCrocodile.Services.LobbyFactory;
```

```
using BlindCrocodile.Services.Network;
using BlindCrocodile.Services.StaticData;
using BlindCrocodile.Services.Relay;
using BlindCrocodile.Services.MenuFactory;
using BlindCrocodile.Core.Services;
using BlindCrocodile.Core;
using BlindCrocodile.Core.StateMachine;
using BlindCrocodile.Lobbies;
using UnityEngine;
using BlindCrocodile.Services.Factories;
using BlindCrocodile.NetworkStates;
namespace BlindCrocodile.GameStates
{
   public class BootstrapState : IGameState, IState
   {
      private readonly ServiceLocator _services;
      private readonly GameStateMachine _gameStateMachine;
      private readonly NetworkStateMachine _networkStateMachine;
      private readonly ICoroutineRunner _coroutineRunner;
      public BootstrapState(GameStateMachine gameStateMachine, ServiceLocator services, ICoroutineRunner coroutineRunner, NetworkStateMachine networkStateMachine)
      {
         _gameStateMachine = gameStateMachine;
         _networkStateMachine = networkStateMachine;
         _services = services;
         _coroutineRunner = coroutineRunner;
         BindServices();
      }
      public void Enter()
      {
         QualitySettings.vSyncCount = 0;
         Application.targetFrameRate = 240;
         _gameStateMachine.Enter<LoadMenuState>();
      }
      public void Exit() { }
      private void BindServices()
      {
         BindRelayService();
```

```
            BindLobbyService();
            BindNetworkService();
            BindStaticDataService();
            BindNetworkFactory();
            BindLobbyFactory(); // make one ui factory
            BindMenuFactory();
        }
        private void BindNetworkFactory() =>
            _services.BindSingle<INetworkFactory>(new
NetworkFactory(_services.Single<IStaticDataService>(), _services.Single<INetworkService>()));
        private void BindMenuFactory() =>
            _services.BindSingle<IMenuFactory>(new
MenuFactory(_services.Single<IStaticDataService>(), _gameStateMachine,
_services.Single<ILobbyService>(), _networkStateMachine));
        private void BindLobbyFactory() =>
            _services.BindSingle<ILobbyFactory>(new
LobbyFactory(_services.Single<INetworkService>(), _services.Single<IStaticDataService>(),
_gameStateMachine, _services.Single<ILobbyService>(), _services.Single<INetworkFactory>()));
        private void BindStaticDataService()
        {
            StaticDataService staticDataService = new();
            staticDataService.Load();
            _services.BindSingle<IStaticDataService>(staticDataService);
        }
        private void BindRelayService() =>
            _services.BindSingle<IRelayService>(new
RelayService(Unity.Services.Relay.RelayService.Instance));
        private void BindLobbyService() =>
            _services.BindSingle<ILobbyService>(new
LobbyService(Unity.Services.Lobbies.Lobbies.Instance, _coroutineRunner, _gameStateMachine));
        private void BindNetworkService()
        {
            UnityTransport unityTransport = Object.FindObjectOfType<UnityTransport>();
            NetworkService networkService = new(unityTransport, NetworkManager.Singleton,
_services.Single<IRelayService>(), _services.Single<ILobbyService>()); // network manager
            _services.BindSingle<INetworkService>(networkService);
        }
    }
```

```
}
using BlindCrocodile.Core;
using BlindCrocodile.Core.StateMachine;
using BlindCrocodile.Services.LobbyFactory;
using BlindCrocodile.Services.Factories;
using BlindCrocodile.NetworkStates;
using BlindCrocodile.Services.Network;
namespace BlindCrocodile.GameStates
{
   public class HostGameState : IGameState, IState
   {
      private const string GAME_SCENE = "GameScene";
      private const string LOBBY_NAME = "GameLobby";
      private const int MAX_CONNECTIONS = 5;
      private readonly AbstractStateMachine<IGameState> _gameStateMachine;
      private readonly NetworkStateMachine _networkStateMachine;
      private readonly ILobbyFactory _lobbyFactory;
      private readonly INetworkFactory _networkFactory;
      private readonly SceneLoader _sceneLoader;
      private readonly LoaderWidget _loaderWidget;
      public HostGameState(AbstractStateMachine<IGameState> stateMachine, ILobbyFactory lobbyFactory, INetworkFactory networkFactory, SceneLoader sceneLoader, LoaderWidget loaderWidget, NetworkStateMachine networkStateMachine)
      {
         _gameStateMachine = stateMachine;
         _lobbyFactory = lobbyFactory;
         _sceneLoader = sceneLoader;
         _loaderWidget = loaderWidget;
         _networkFactory = networkFactory;
         _networkStateMachine = networkStateMachine;
      }
      public void Enter()
      {
         _loaderWidget.Show();
         _sceneLoader.Load(GAME_SCENE, OnLoaded);
      }
      public void Exit() =>
         _loaderWidget.Hide();
```

```
            private void OnLoaded()
            {
                NetworkPlayersLobby networkList = _networkFactory.CreateNetworkPlayer();
                _lobbyFactory.CreateHud();
                _gameStateMachine.Enter<GameLoopState>();
                _networkStateMachine.Enter<HostState, NetworkPlayersLobby>(networkList);
            }
        }
    }
    using BlindCrocodile.Core;
    using BlindCrocodile.Core.StateMachine;
    using BlindCrocodile.Services.LobbyFactory;
    using BlindCrocodile.Services.Factories;
    using BlindCrocodile.NetworkStates;
    namespace BlindCrocodile.GameStates
    {
        public class JoinGameState : IGameState, IPayloadedState<string>
        {
            private const string GAME_SCENE = "GameScene";
            private readonly AbstractStateMachine<IGameState> _gameStateMachine;
            private readonly NetworkStateMachine _networkStateMachine;
            private readonly INetworkFactory _networkFactory;
            private readonly ILobbyFactory _lobbyFactory;
            private readonly SceneLoader _sceneLoader;
            private readonly LoaderWidget _loaderWidget;
            public JoinGameState(AbstractStateMachine<IGameState> stateMachine, ILobbyFactory
lobbyFactory, INetworkFactory networkFactory, SceneLoader sceneLoader, LoaderWidget loaderWidget,
NetworkStateMachine networkStateMachine)
            {
                _gameStateMachine = stateMachine;
                _lobbyFactory = lobbyFactory;
                _sceneLoader = sceneLoader;
                _loaderWidget = loaderWidget;
                _networkFactory = networkFactory;
                _networkStateMachine = networkStateMachine;
            }
            public void Enter(string lobbyCode)
            {
```

```
            _loaderWidget.Show();
            _sceneLoader.Load(GAME_SCENE, delegate { OnLoaded(lobbyCode); });
        }
        public void Exit() =>
            _loaderWidget.Hide();
        private void OnLoaded(string lobbyCode)
        {
            _networkFactory.CreateNetworkPlayer();
            InitLobbyHudAsync(lobbyCode);
            _gameStateMachine.Enter<GameLoopState>();
            _networkStateMachine.Enter<JoinState, string>(lobbyCode);
        }
        private void InitLobbyHudAsync(string lobbyCode)
        {
            _lobbyFactory.CreateHud();
        }
    }
}
    public struct BytesContainer : INetworkSerializable, IDisposable
    {
        public NativeArray<byte> Bytes;
        public BytesContainer(byte[] bytes)
        {
            Bytes = new(bytes, Allocator.Persistent);
        }
        public void Dispose() =>
            Bytes.Dispose();
        public void NetworkSerialize<T>(BufferSerializer<T> serializer) where T : IReaderWriter
        {
            int length = 0;
            if (!serializer.IsReader)
                length = Bytes.Length;
            serializer.SerializeValue(ref length);
            if (serializer.IsReader)
            {
                if (Bytes.IsCreated)
                    Bytes.Dispose();
                Bytes = new NativeArray<byte>(length, Allocator.Persistent);
```

```
            }
            for (int n = 0; n < length; ++n)
            {
                byte val = Bytes[n];
                serializer.SerializeValue(ref val);
                Bytes[n] = val;
            }
        }
    }
    public enum PlayerRole : byte
    {
        Artist,
        Guesser
    }
using BlindCrocodile.Core.Services;
using System.Threading.Tasks;
using UnityEngine;
namespace BlindCrocodile.Services.Network
{
    public interface INetworkService : IService
    {
        public bool IsServer { get; }
        public bool IsClient { get; }
        public ulong ClientId { get; }
        Task StartHostAsync(int maxConnections);
        Task JoinServerAsync();
        void Disconnect();
        void AddNetworkPrefab(GameObject gameObject);
    }
}
using BlindCrocodile.Services.Relay;
using BlindCrocodile.Lobbies;
using Unity.Services.Relay.Models;
using Unity.Netcode.Transports.UTP;
using Unity.Netcode;
using Unity.Networking.Transport.Relay;
using System.Threading.Tasks;
using UnityEngine;
```

```
using System.Text;
namespace BlindCrocodile.Services.Network
{
   public class NetworkService : INetworkService
   {
      private const string DTLS_CONNECTION = "dtls";
      public bool IsServer => _networkManager.IsServer;
      public bool IsClient => _networkManager.IsClient;
      public ulong ClientId => _networkManager.LocalClientId;
      private readonly UnityTransport _unityTransport;
      private readonly NetworkManager _networkManager;
      private readonly IRelayService _relayService;
      private readonly ILobbyService _lobbyService;
      public NetworkService(UnityTransport unityTransport, NetworkManager networkManager,
IRelayService relayService, ILobbyService lobbyService)
      {
         _unityTransport = unityTransport;
         _networkManager = networkManager;
         _relayService = relayService;
         _lobbyService = lobbyService;
         _lobbyService.LocalPlayer.NetworkId = ClientId;
         _networkManager.NetworkConfig.ConnectionApproval = true;
      }
      public async Task StartHostAsync(int maxConnections)
      {
         Allocation allocation = await _relayService.CreateAllocationAsync(maxConnections);
         string relayCode = await _relayService.GetJoinCodeAsync(allocation.AllocationId);
         _lobbyService.LocalLobby.RelayCode = relayCode;
         await _lobbyService.UpdateRemoteLobbyDataAsync();
         await     _lobbyService.UpdateRemotePlayerDataAsync(allocation.AllocationId.ToString(),
relayCode);
         _unityTransport.SetRelayServerData(new                    RelayServerData(allocation,
DTLS_CONNECTION));
         _networkManager.NetworkConfig.ConnectionData                                        =
Encoding.UTF8.GetBytes(_lobbyService.LocalPlayer.Id);
         _networkManager.StartHost();
      }
      public async Task JoinServerAsync()
```

```
            {
                string relayCode = _lobbyService.LocalLobby.RelayCode;
                if (string.IsNullOrEmpty(relayCode))
                    return;
                JoinAllocation joinAllocation = await _relayService.JoinAllocationAsync(relayCode);
                await
_lobbyService.UpdateRemotePlayerDataAsync(joinAllocation.AllocationId.ToString(), relayCode);
                _unityTransport.SetRelayServerData(new RelayServerData(joinAllocation,
DTLS_CONNECTION));
                _networkManager.NetworkConfig.ConnectionData =
Encoding.UTF8.GetBytes(_lobbyService.LocalPlayer.Id);
                _networkManager.StartClient();
            }
            public void Disconnect()
            {
                _lobbyService.DisconnectFromLobby();
                _networkManager.Shutdown();
            }
            public void AddNetworkPrefab(GameObject gameObject)
            {
                _networkManager.AddNetworkPrefab(gameObject);
            }
        }
        public record PayloadData // data for connection to server
        {
        }
    }
    using BlindCrocodile.Gameplay.Drawing;
    using BlindCrocodile.Services.LobbyFactory;
    using System.Collections.Generic;
    using UnityEngine;
    using UnityEngine.UI;
    namespace BlindCrocodile.UI
    {
        public class GuesserHudController : MonoBehaviour
        {
            [SerializeField] private MaskController _maskController;
            [SerializeField] private Drawer _canvasDrawer;
```

```
[SerializeField] private Button _compareButton;
[SerializeField] private Button _backGuessingButton;
[SerializeField] private GameObject _guessingPanel;
[SerializeField] private GameObject _drawingPanel;
[SerializeField] private Transform _colorStatItemsParent;
[SerializeField] private ComparisonController _comparisonController;
private ILobbyFactory _lobbyFactory;
private List<ColorStatItem> _colorStatItems;
private void Awake()
{
   _colorStatItems = new List<ColorStatItem>();
   _compareButton.onClick.AddListener(OnCompare);
   _backGuessingButton.onClick.AddListener(OnBackGuessing);
   _canvasDrawer.SetBlocked(false);
   _maskController.SetBlocked(true);
}
public void Construct(ILobbyFactory lobbyFactory, byte[] artistCanvasBytes)
{
   _lobbyFactory = lobbyFactory;
   _comparisonController.Construct(artistCanvasBytes, _canvasDrawer.CanvasSize);
}
private void OnCompare()
{
   _drawingPanel.SetActive(false);
   _compareButton.gameObject.SetActive(false);
   _guessingPanel.SetActive(true);
   _backGuessingButton.gameObject.SetActive(true);
   _canvasDrawer.SetBlocked(true);
   _maskController.SetBlocked(false);
   _comparisonController.CompareTo(_canvasDrawer.CanvasTexture);
   Color[] canvasColors = _canvasDrawer.GetCanvasColors();
   Dictionary<Color, int> uniqueColors = new();
   foreach (Color color in canvasColors)
   {
      if (!uniqueColors.TryAdd(color, 1))
         uniqueColors[color]++;
   }
   foreach (ColorStatItem item in _colorStatItems)
```

```
                Destroy(item.gameObject);
            _colorStatItems.Clear();
            foreach (KeyValuePair<Color, int> color in uniqueColors)
            {
                float colorAmount = (float)color.Value / canvasColors.Length;
                ColorStatItem colorStatItem = _lobbyFactory.CreateColorStatItem(color.Key,
colorAmount, _colorStatItemsParent);
                _colorStatItems.Add(colorStatItem);
            }
        }
        private void OnBackGuessing()
        {
            _drawingPanel.SetActive(true);
            _compareButton.gameObject.SetActive(true);
            _guessingPanel.SetActive(false);
            _backGuessingButton.gameObject.SetActive(false);
            _canvasDrawer.SetBlocked(false);
            _maskController.SetBlocked(true);
        }
    }
}
using BlindCrocodile.Gameplay.Drawing;
using System;
using UnityEngine;
using UnityEngine.UI;
namespace BlindCrocodile.UI
{
    public class ArtistHudController : MonoBehaviour
    {
        public event Action<byte[]> OnCanvasShared;
        [SerializeField] private Drawer _canvasDrawer;
        [SerializeField] private Button _shareButton;
        private void Awake()
        {
            _shareButton.onClick.AddListener(OnShare);
            _shareButton.interactable = true;
        }
        private void OnShare()
```

```
        {
            _shareButton.interactable = false;
            byte[] canvas = _canvasDrawer.GetCanvasBytes();
            OnCanvasShared?.Invoke(canvas);
        }
    }
}
using BlindCrocodile.UI;
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.UI;
namespace BlindCrocodile.Gameplay.Drawing
{
    public class Drawer : MonoBehaviour
    {
        private static readonly int PaintColorID = Shader.PropertyToID("_PaintColor");
        private static readonly int PaintPosID = Shader.PropertyToID("_PaintPos");
        private static readonly int BrushSizeID = Shader.PropertyToID("_BrushSize");
        public Vector2 CanvasSize { get; private set; }
        public RenderTexture CanvasTexture { get; private set; }
        [SerializeField] private RectTransform _rectTransform;
        [SerializeField] private Material _paintMaterial;
        [SerializeField] private RawImage _canvas;
        [SerializeField] private BrushColorButton[] _brushColorButtons;
        [SerializeField] private BrushSizeButton[] _brushSizeButtons;
        private Color _color;
        private float _brushSize = 15f;
        private Vector2 _previousMousePos;
        private RenderTexture _tmpRT;
        private bool _isBlocked;
        private CommandBuffer _canvasCommandBuffer;
        private void Start()
        {
            for (int i = 0; i < _brushColorButtons.Length; i++)
                _brushColorButtons[i].OnColorChanged += SetColor;
            for (int i = 0; i < _brushSizeButtons.Length; i++)
                _brushSizeButtons[i].OnSizeChanged += SetSize;
            _canvasCommandBuffer = new CommandBuffer();
```

```
    CanvasSize = _rectTransform.sizeDelta;
    CanvasTexture = RenderTexture.GetTemporary((int)CanvasSize.x, (int)CanvasSize.y);
    CanvasTexture.filterMode = FilterMode.Point;
    _canvasCommandBuffer.SetRenderTarget(CanvasTexture);
    ClearCanvas(Color.white);
    _tmpRT = RenderTexture.GetTemporary(CanvasTexture.descriptor);
    _canvas.texture = CanvasTexture;
}
private void OnDestroy()
{
    _canvasCommandBuffer.Dispose(); // null ref
    _tmpRT.Release();
    CanvasTexture.Release();
    for (int i = 0; i < _brushColorButtons.Length; i++)
        _brushColorButtons[i].OnColorChanged -= SetColor;
    for (int i = 0; i < _brushSizeButtons.Length; i++)
        _brushSizeButtons[i].OnSizeChanged -= SetSize;
}
private void Update()
{
    if (_isBlocked)
        return;
    RectTransformUtility.ScreenPointToLocalPointInRectangle(
        _rectTransform,
        Input.mousePosition,
        null,
        out Vector2 currentMouse);
    currentMouse += CanvasSize / 2f;
    if (Input.GetMouseButton(0))
    {
        _paintMaterial.SetColor(PaintColorID, _color);
        _paintMaterial.SetFloat(BrushSizeID, _brushSize);
        PaintCanvas(currentMouse);
    }
    if (Input.GetMouseButton(1))
        ClearCanvas(Color.white);
    _previousMousePos = currentMouse;
}
```

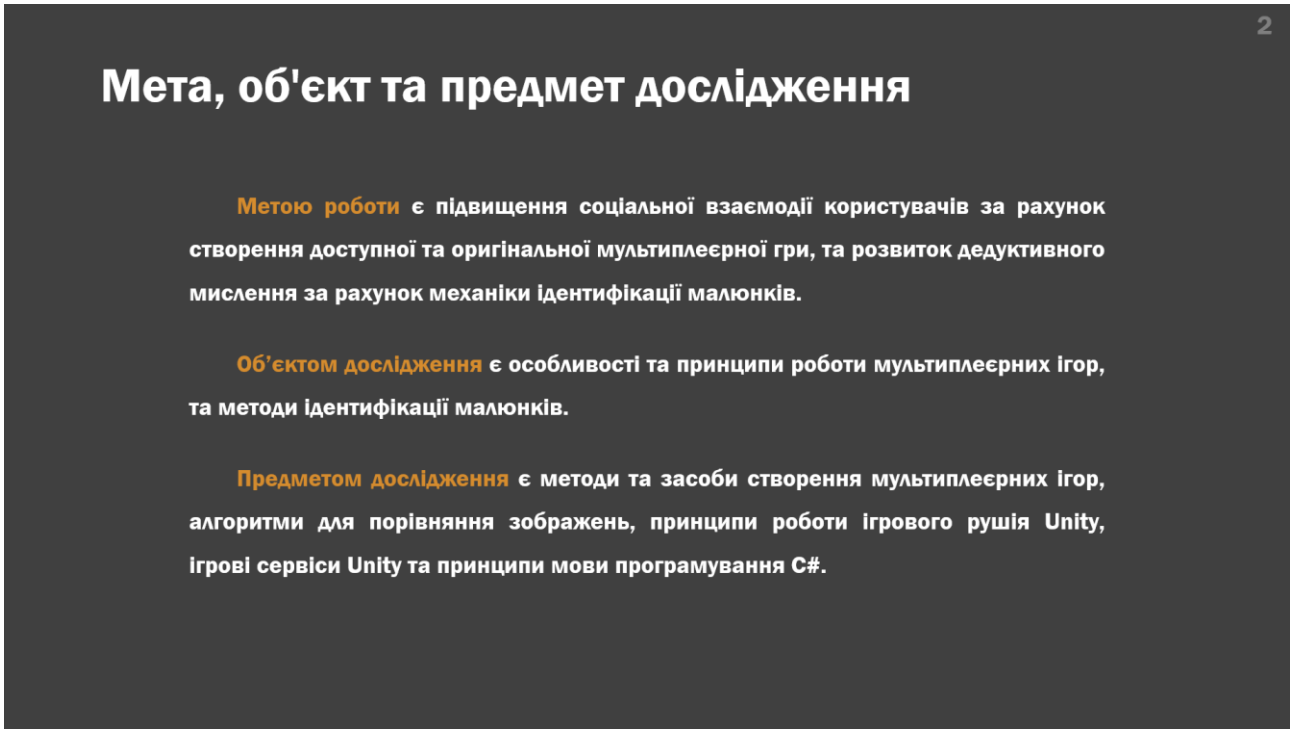
```
public void SetColor(Color color) =>
    _color = color;
public void SetSize(float size) =>
    _brushSize = size;

public void SetBlocked(bool blocked) =>
    _isBlocked = blocked;
public Color[] GetCanvasColors()
{
    Texture2D texture2D = new((int)CanvasSize.x, (int)CanvasSize.y);
    RenderTexture.active = CanvasTexture;
    texture2D.ReadPixels(new Rect(Vector2.zero, CanvasSize), 0, 0);
    texture2D.Apply();
    Color[] colors = texture2D.GetPixels();
    Destroy(texture2D);
    return colors;
}
public byte[] GetCanvasBytes()
{
    Texture2D texture2D = new((int)CanvasSize.x, (int)CanvasSize.y);
    RenderTexture.active = CanvasTexture;
    texture2D.ReadPixels(new Rect(Vector2.zero, CanvasSize), 0, 0);
    texture2D.Apply();
    byte[] png = texture2D.EncodeToPNG();
    Destroy(texture2D);
    return png;
}
public void CreateFromBytes(byte[] textureBytes)
{
    Texture2D texture = new((int)CanvasSize.x, (int)CanvasSize.y);
    texture.LoadImage(textureBytes);
    texture.Apply();
    Graphics.Blit(texture, CanvasTexture);
    Destroy(texture);
}
private void PaintCanvas(Vector2 mousePos)
{
    float distance = Vector2.Distance(mousePos, _previousMousePos);
```

```
            float lerpStep = _brushSize * 0.5f / distance;
            for (float lerp = 0; lerp <= 1f; lerp += lerpStep)
            {
                Vector2 pos = Vector2.Lerp(_previousMousePos, mousePos, lerp);
                _paintMaterial.SetVector(PaintPosID, pos);
                Graphics.Blit(CanvasTexture, _tmpRT, _paintMaterial);
                Graphics.CopyTexture(_tmpRT, CanvasTexture);
            }
        }
        private void ClearCanvas(Color color)
        {
            _canvasCommandBuffer.ClearRenderTarget(true, true, color);
            Graphics.ExecuteCommandBuffer(_canvasCommandBuffer);
            _canvasCommandBuffer.Clear();
        }
    }
}
```

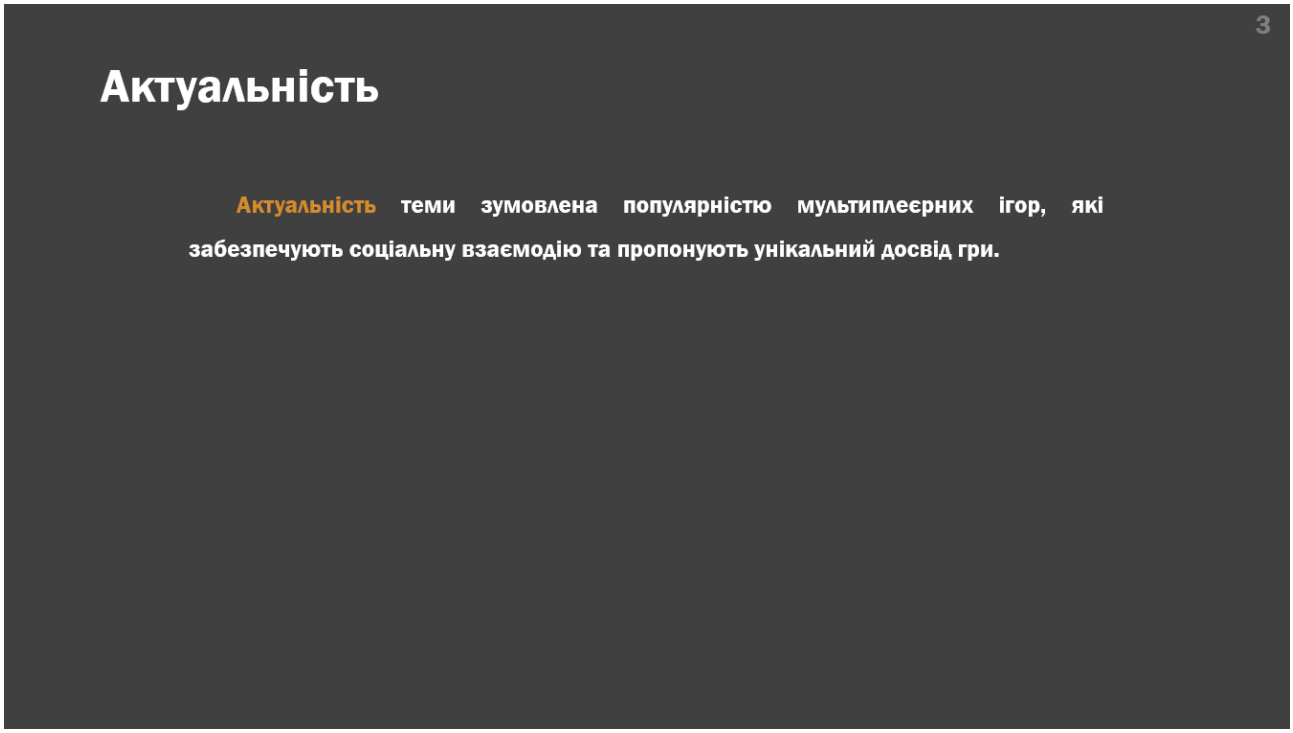
Додаток Г – Графічна частина

РОЗРОБКА МУЛЬТИПЛЕЄРНОЇ 2D-ГРИ З МЕХАНІКОЮ ІДЕНТИФІКАЦІЇ МАЛЮНКІВ

Г.1 – Титульний слайд



Г.2 – Мета, об’єкт та предмет дослідження

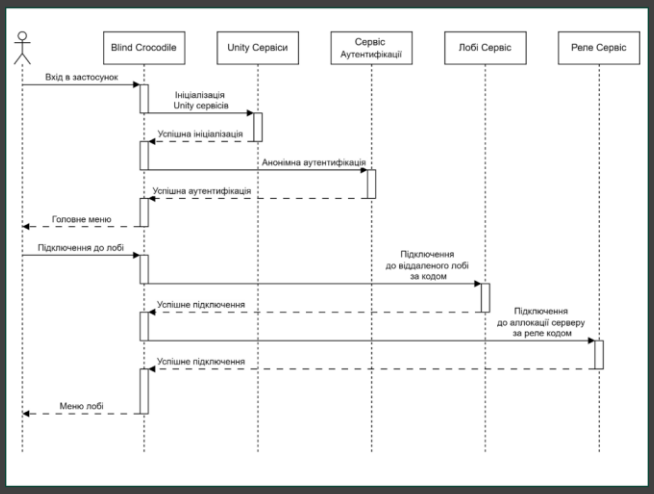


Г.3 – Актуальність розробки

Г.4 – UML діаграма варіантів використання

5

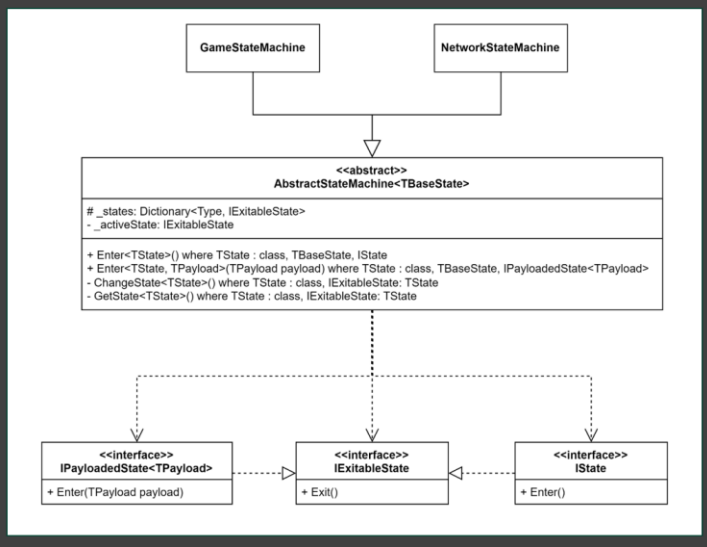
Діаграма послідовності під'єднання до лобі



Г.5 – UML діаграма послідовності

6

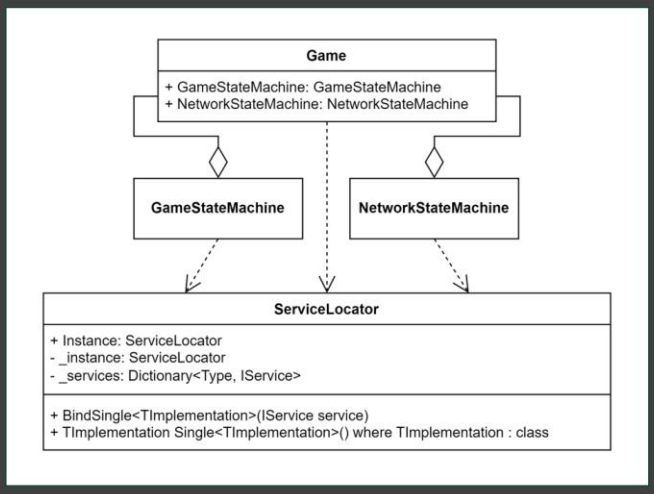
Діаграма класів машини станів



Г.6 – UML діаграма класів машини станів

7

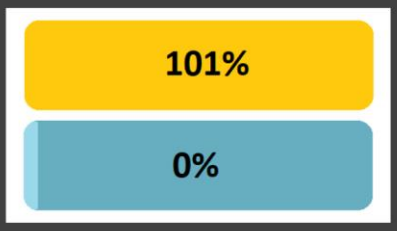
Діаграма класів ключових модулів системи



Г.7 – UML діаграма ключових модуль архітектури

8

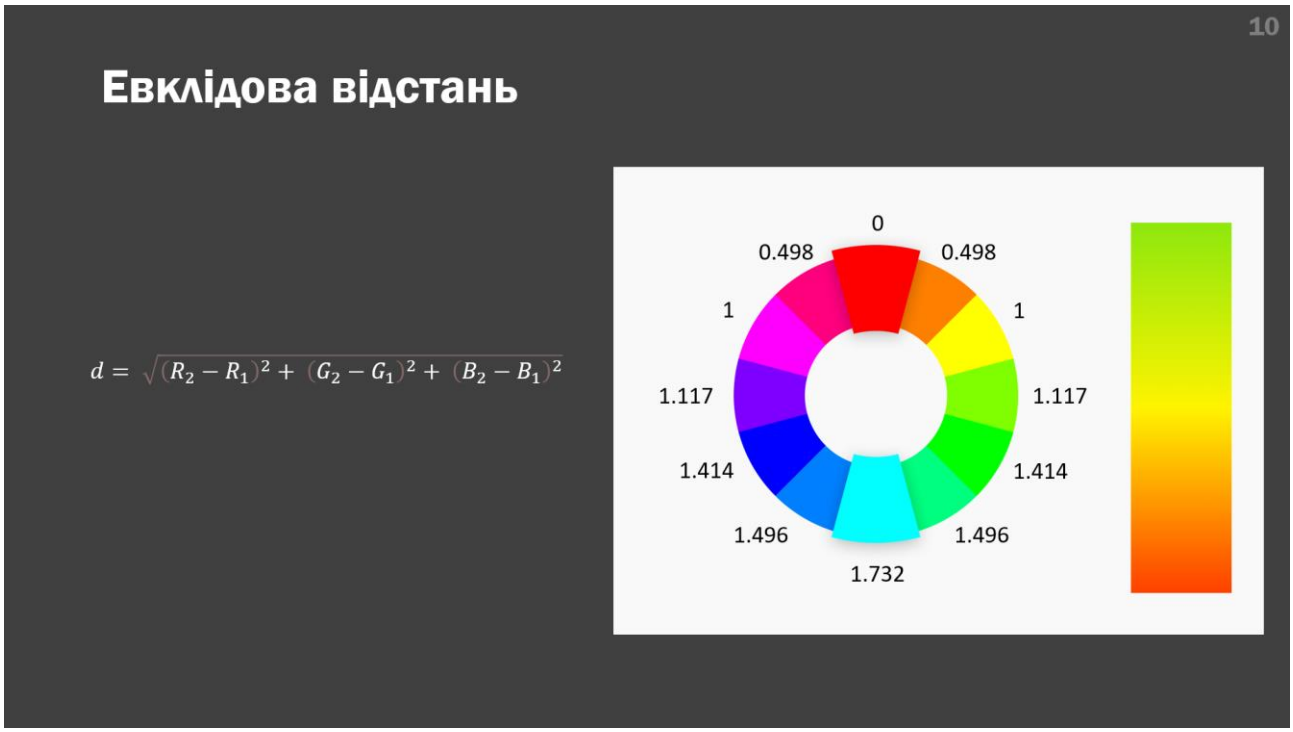
Індикатори для фільтрування кольорів



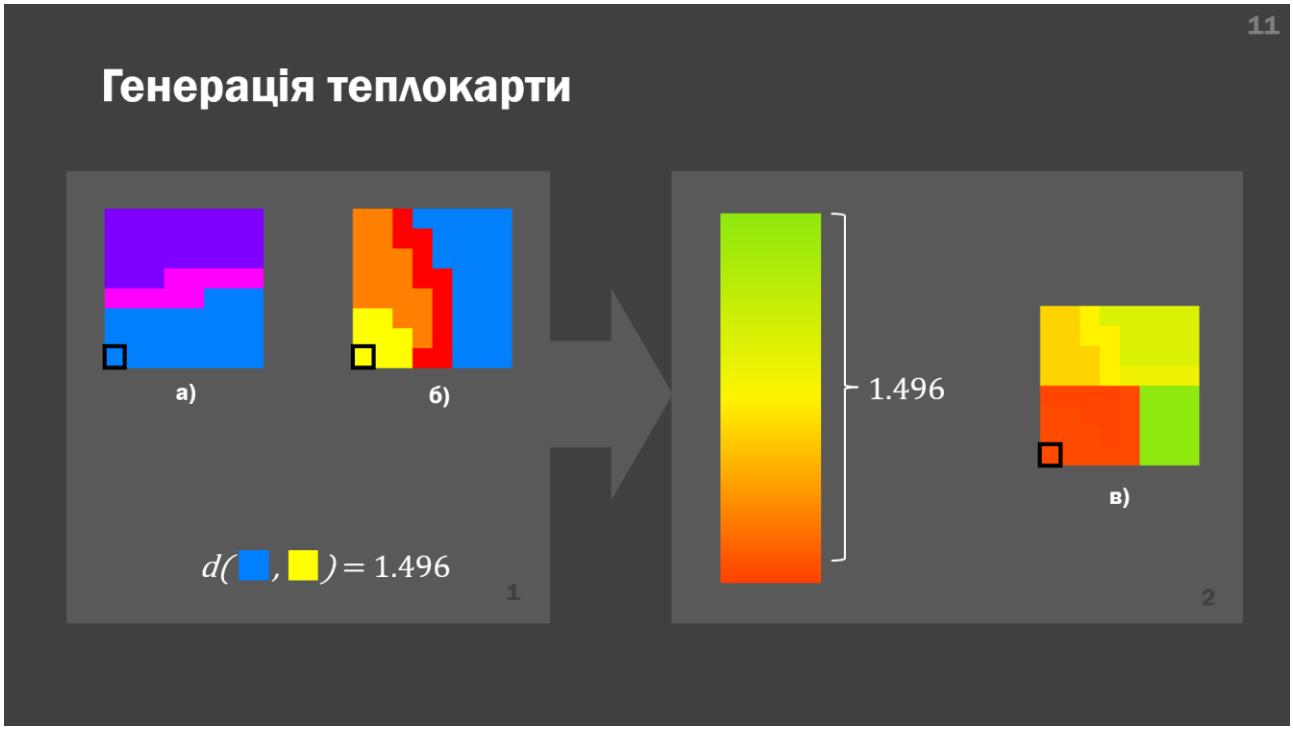
Використовуються для зображення співвідношення кількості замальованих пікселів на картинці гравця в порівнянні з оригінальним зображень

Г.8 – Генерація індикаторів

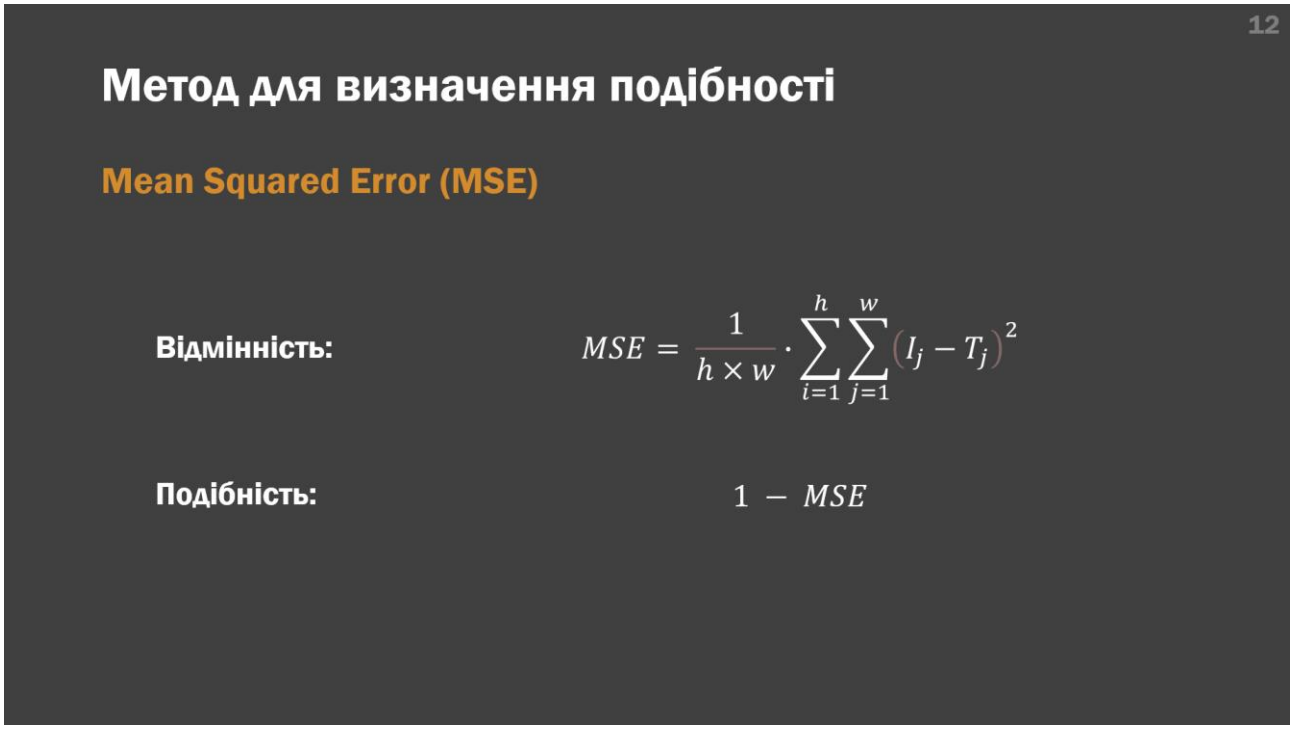
Г.9 – Використання теплокарти



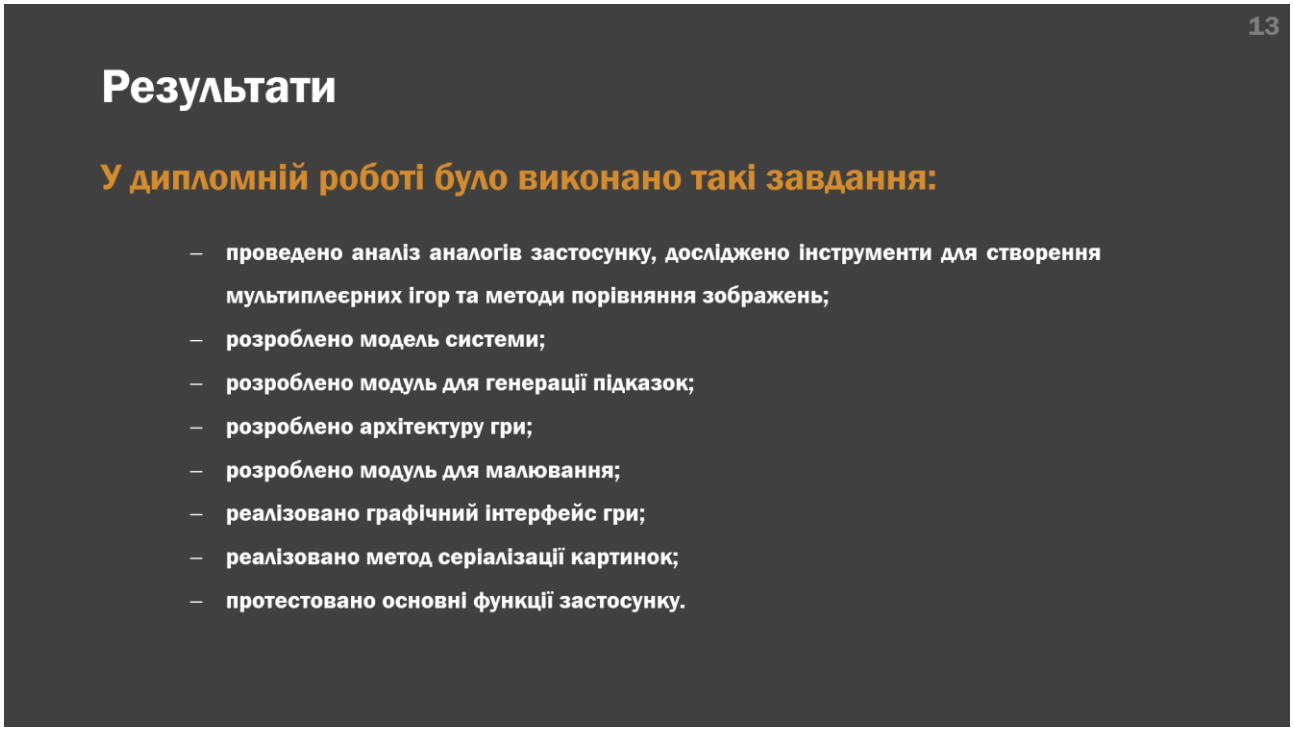
Г.10 – Евклідова відстань



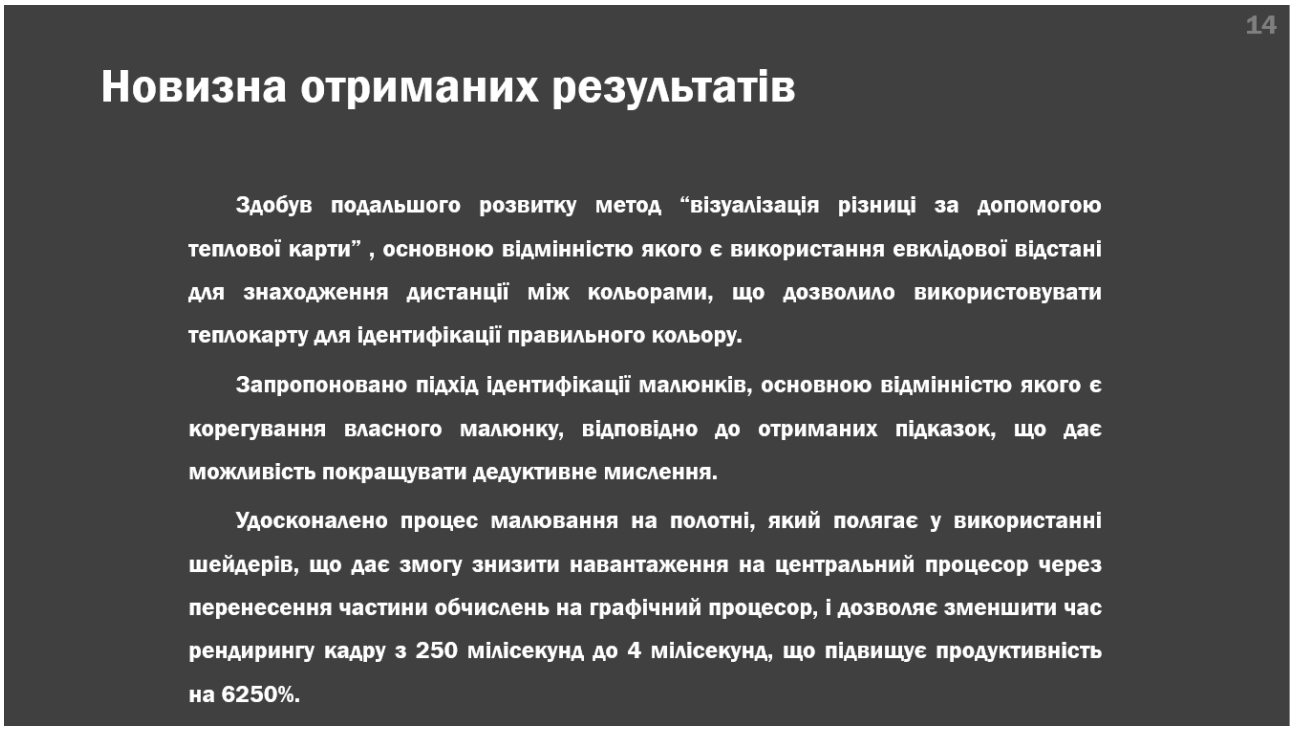
Г.11 – Генерація теплокарти



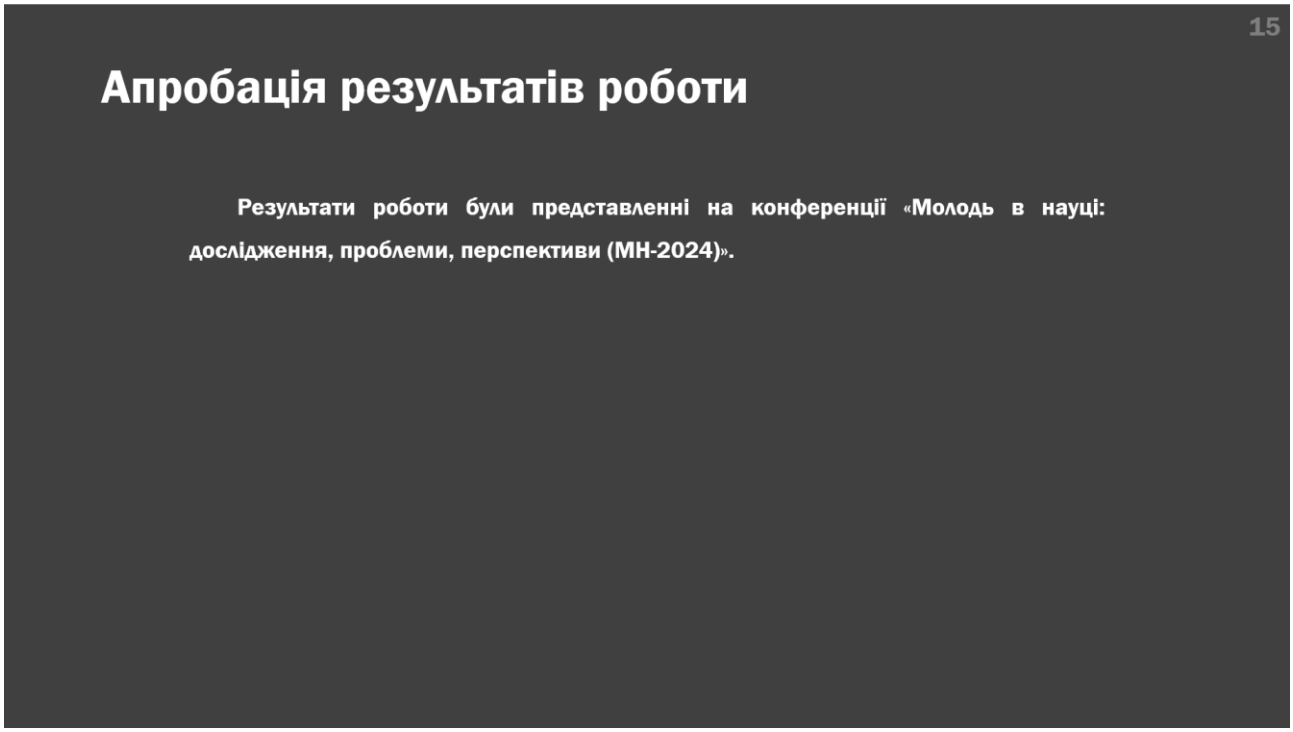
Г.12 – Алгоритм Mean Squared Error



Г.13 – Результати досліджень



Г.14 – Новизна отриманих результатів



Г.15 – Апробація результатів дослідження