ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
МІСЬКОГО ГОСПОДАРСТВА ІМЕНІ О. М. БЕКЕТОВА

Пояснювальна записка
до кваліфікаційної роботи бакалавра

на тему:

«Розробка візуального стилю та анімаційного контенту для 2D-гри на базі Unity»

Виконав: здобувач вищої освіти,
групи КН 2021-1
Спеціальності 122 – Комп’ютерні науки
Сергій БАТАЛОВ
Керівник: Юрій ЛЕВІКОВ
Рецензент: Наталія БРАТЕРСЬКА

м. Харків – 2025

Харківський національний університет міського господарства імені О. М. Бекетова
(повне найменування закладу вищої освіти)
Навчально-науковий Інститут енергетичної, інформаційної та транспортної інфраструктури
Кафедра комп'ютерних наук та інформаційних технологій
Рівень вищої освіти перший (бакалаврський)
Спеціальність 122 Комп'ютерні науки
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри КНтаІТ

 Марина НОВОЖИЛОВА
«26» 06 2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Баталову Сергію Денисовичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Розробка візуального стилю та анімаційного контенту для 2D-гри на базі Unity»

керівник роботи Левіков Юрій Володимирович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом закладу вищої освіти від «09» травня 2025 р. №341-03

2. Термін подання студентом роботи 21.06.2025

3. Вихідні дані до роботи Проектування та реалізація візуального стилю та анімаційного контенту для 2D-гри на базі Unity

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): Аналіз предметного середовища, дослідження сучасних підходів до побудови графічного контенту, аналіз існуючих аналогів, реалізація інтерфейсів користувача та анімаційного контенту.

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

Презентація – 19 аркушів

6. Консультанти розділів роботи

Розділ	Ім'я та Прізвище, посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Юрій ЛЕВІКОВ, старший викладач кафедри КН та ІТ	13.05.2025	23.05.2025
2	Юрій ЛЕВІКОВ, старший викладач кафедри КН та ІТ	26.05.2025	05.06.2025
3	Юрій ЛЕВІКОВ, старший викладач кафедри КН та ІТ	03.06.2025	08.06.2025
4	Вікторія МАЛИШЕВА, к. т. н., доцент кафедри ОП та БЖ	09.06.2025	13.06.2025

7. Дата видачі завдання 12.05.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вибір теми дипломної роботи	13.05.2025	виконано
2	Затвердження тем, наукових керівників, завдань та календарного плану підготовки кваліфікаційної роботи	13.05.2025	виконано
3	Написання I розділу	23.05.2025	виконано
4	Написання II розділу	05.06.2025	виконано
5	Написання III розділу	08.06.2025	виконано
6	Написання IV розділу	13.06.2025	виконано
7	Подання дипломної роботи керівнику	17.06.2025	виконано
8	Робота по усуненню зауважень керівника, уточнення і доповнення практичного матеріалу, оформлення додатків до роботи	19.06.2025	виконано
9	Подання доопрацьованого варіанту роботи керівнику	21.06.2025	виконано
10	Офіційний захист матеріалів дипломної роботи на засіданні екзаменаційної комісії	27.06.2025	виконано

Студент _____ (підпис) Сергій БАТАЛОВ (ім'я та прізвище)

Керівник роботи _____ (підпис) Юрій ЛЕВІКОВ (ім'я та прізвище)

АНОТАЦІЯ

Структура та обсяг роботи.

Пояснювальна записка кваліфікаційної роботи бакалавра студента групи КН 2021-1 спеціальності 122 – Комп'ютерні науки Баталова Сергія Денисовича на тему «Розробка візуального стилю та анімаційного контенту для 2D-гри на базі Unity» містить 4 розділи, включає 5 рисунків, 10 таблиць, 30 джерел та 20 додатків.

Предмет дослідження: технічні аспекти створення, інтеграції та оптимізації візуального стилю й анімаційного контенту у грі.

Мета роботи: розробка та інтеграція візуального стилю й анімаційного контенту для 2D-гри в середовищі Unity.

Методи дослідження: включають аналіз літератури та документації, проєктування програмної системи, програмування на C#, тестування продуктивності та оптимізацію графічних ресурсів.

Програмне забезпечення: Unity 2022.3, Visual Studio 2022, Adobe Photoshop 2024, Krita 5.2.9, PureRef 2.0.

Результати: розроблено інтерфейси користувача, реалізовано повний набір покадрових анімацій персонажів, макети та стилізовані локації, реалізовано паралакс-ефект, створено візуальне наповнення для міні-ігор.

Рекомендації щодо використання результатів: отримані результати можуть бути використані для створення інтерфейсів та анімацій у навчальних 2D-проєктах, а також у розробці ігор інді-формату.

Галузь застосування: ігрова індустрія, навчальні й інді-проєкти.

Значущість роботи та висновки: робота сприяє розвитку практичних навичок інтеграції візуального контенту в ігрове середовище, поглиблює навички взаємодії в командній розробці, демонструє шляхи оптимізації ресурсів та створення емоційно-залученого візуального наративу.

Ключові слова: 2D-ГРА, UNITY, ВІЗУАЛЬНИЙ СТИЛЬ, АНІМАЦІЯ, СПРАЙТИ, OPTIMIZATION, UI DESIGN.

ANNOTATION

Structure and scope of work.

Explanatory note of the qualification work of the bachelor's student of the group KN 2021-1 specialty 122 - Computer Science Batalov Sergey Denisovich on the topic "Development of visual style and animation content for a 2D game based on Unity" contains 4 sections, includes 5 figures, 10 tables, 30 sources and 20 appendices.

Subject of research: technical aspects of creating, integrating and optimizing visual style and animation content in this game.

Purpose of work: development and integration of visual style and animation content for a 2D game in the Unity environment.

Research methods: include analysis of literature and documentation, software system design, C# programming, performance testing and optimization of graphic resources.

Software: Unity 2022.3, Visual Studio 2022, Adobe Photoshop 2024, Krita 5.2.9, PureRef 2.0.

Results: user interfaces were developed, a full set of frame-by-frame character animations, mockups and stylized locations were implemented, a parallax effect was implemented, visual content for mini-games was created.

Recommendations for using the results: the results can be used to create interfaces and animations in educational 2D projects, as well as in the development of indie games.

Field of application: gaming industry, educational and indie projects.

Significance of the work and conclusions: the work contributes to the development of practical skills in integrating visual content into the gaming environment, deepens interaction skills in team development, demonstrates ways to optimize resources and create an emotionally engaging visual narrative.

Keywords: 2D GAME, UNITY, VISUAL STYLE, ANIMATION, SPRITES, OPTIMIZATION, UI DESIGN.

ЗМІСТ

ВСТУП

На сьогодні ігри є одним з найвпливовіших феноменів сучасної цифрової епохи, що поєднують в собі передові технології, креативність і складні інженерні рішення для створення інтерактивних віртуальних світів.

Розробка ігор – це багатогранний і міждисциплінарний процес, який охоплює аналіз вимог, проєктування ігрової логіки, створення візуального, звукового та наративного контенту, програмування, тестування й оптимізацію. Цей процес вимагає співпраці фахівців із комп'ютерних наук, графічного дизайну, звукорежисури та маркетингу, а також глибокого розуміння програмування та оптимізації.

У сучасному світі комп'ютерні ігри вже давно перестали бути простої розвагою. Сьогодні вони є потужним інструментом соціального, освітнього, культурного та економічного впливу. Завдяки ним з'являються нові способи взаємодії та комунікації, сприяючи розвитку глобальних спільнот через кіберспортивні турніри, багатокористувацькі платформи та віртуальні світи[1]. Такі ігри як «Minecraft» або «Counter-Strike», наочно демонструють, що віртуальні простори можуть об'єднувати мільйони людей по всьому світу, створюючи унікальні зв'язки.

Гейміфікація охоплює і сферу освіти, роблячи її більш інтерактивною та викликаючи зацікавленість серед молоді[1]. Гарним прикладом є симуляційні ігри , які використовуються для підготовки пілотів, медиків і інженерів. У медицині ігри можуть використовуватися для реабілітації, психотерапії та тренування когнітивних навичок, наприклад у лікуванні розладів уваги чи відновленні після травм.

Культурно ігри стають новим видом мистецтва, поєднуючи візуальну естетику, музику та наратив для передачі складних ідей і емоцій, як це видно в таких проєктах, як «Journey» чи «Gris». Ці аспекти підкреслюють, що ігри є не лише продуктом технологій, але й культурним феноменом, який відображає суспільні цінності та впливає на формування сучасної ідентичності.

Поява таких інструментів розробки, як Unity та Unreal Engine вивела розробку ігор на новий рівень, зробивши її доступною для розробників по всьому світу. Вони змогли відкинути необхідність у фінансуванні та великій команді спеціалістів. Займатись розробкою ігор може будь-хто, потрібно лише мати бажання. Таким чином на ринку з'явилися indie-розробники, які часто об'єднані любов'ю до ігор та бажанням відобразити свої емоції та ідеї у грі.

Серед різноманітних сегментів ігрової індустрії 2D-ігри займають особливе місце, залишаючись актуальними попри бурхливий розвиток 3D-технологій. 2D-сегмент вирізняється своєю економічністю, доступністю для невеликих команд і унікальною естетичною привабливістю, яка дозволяє створювати виразні візуальні світи з мінімальними ресурсами. Такі ігри, як «Hollow Knight», «Stardew Valley» чи «Valiant Hearts: The Great War», демонструють, що 2D-формат здатен конкурувати з 3D-проєктами, пропонуючи глибокий ігровий досвід і сильний емоційний вплив.

Метою роботи є розробка та інтеграція візуального стилю й анімаційного контенту для 2D-гри в середовищі Unity,. У процесі створення гри основна задача полягала в інтеграції графічних активів у середовище розробки, що включало створення функціонального головного меню, налаштування логіки відображення спарйтів персонажів та оточення залежно від вхідних данних користувача, а також оптимізацію графічних елементів для підвищення продуктивності гри.

Об'єктом дослідження є 2D-гра, створена в середовищі Unity.

Предметом дослідження виступають технічні аспекти створення, інтеграції та оптимізації візуального стилю й анімаційного контенту в цій грі.

Методи дослідження включають аналіз літератури та документації, проєктування програмної системи, програмування на C#, тестування продуктивності та оптимізацію графічних ресурсів. У процесі роботи використано ігрове середовище розробки Unity, мову програмування C#, а також інструменти для обробки спрайтів.

РОЗДІЛ 1
ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1 Опис предметного середовища

2D гра – це жанр відеоігор, де графіка контент складається із двовимірних зображень. Графічне оформлення таких ігор використовує спрайти, піксельну графіку або векторну графіку[2]. Ігрові об’єкти задаються координатами на плоскому полотні, а взаємодія з гравцем та логіка визначена у двовимірному просторі. Рендеринг зазвичай виконується за допомогою 2D-графічних бібліотек або рушіїв, а ігрова логіка базується на двовимірних фізичних розрахунках та перетворенні координат. Серед успішних 2D проєктів можна виділити: «Hollow Knight», «Valiant Heart: The Great War», «Cuphead» та інші[3].

Розробка 2D гри – це процес створення 2D гри, що включає в себе велику кількість процесів та потребує участі фахівців різних галузей.

Гра, що розробляється буде виконана у жанрі пригодницького квесту з елементами інтерактивної драми. Така гра потребує хорошої роботи над наративними елементами та емоційною складовою сюжету. Серед особливостей цього жанру можна виділити наступні елементи:

– Наративна основа: найбільшу цінність цього жанру несе в собі саме сюжет, який часто піднімає важливі теми, дає гравцю можливість прив’язатися до героїв, знайти в них асоціації із реальним життям, викликає емоційний відгук та спонукає його до роздумів на важливі моральні дилеми.

– Головоломки та дослідження: одними з найважливіших геймплейних складових пригодницького квесту є розв’язання головоломок, пошук предметів в рамках квестів, взаємодія з оточенням та іншими персонажами. До таких головоломок можуть відноситись прості задачі, наприклад: комбінування предметів, розшифровка кодів або вирішення ситуаційних задач, які виникають в ході сюжету.

– Атмосферне оточення: локації мають підкреслювати необхідну атмосферу та відповідати обраному сеттингу гри. Гравець повинен мати змогу взаємодіяти з цим світом, наприклад, через використання деяких механізмів, замків або іншими елементами оточення. Правильне використання спрайтів та гарно підібраний дизайн локацій може краще занурити гравця у всесвіт гри та передати йому особливості цього світу.

– Емоційна залученість: особливості інтерактивної драми можуть передаватися завдяки діалоговим системам, через які користувач зможе дізнатися історії інших героїв або отримати нову інформацію щодо місця, у якому знаходиться його персонаж. Деякі елементи оточення можуть підкреслювати особливості історії всесвіту гри, викликаючи у гравця емоційну прив’язаність та створюючи відчуття того, що цей світ не просто ігрове зображення, а реальний та має свої закони.

Об’єктом дослідження даної роботи постає створення графічного оформлення для 2D гри у жанрі пригодницького квесту з елементами інтерактивної драми[4].

Основні етапи роботи з ігровою графікою:

1) Розробка концепції та стилю: стиль гри має відповідати вимогам обраного жанру, щоб викликати у гравця максимальний відгук та понурити його у необхідну атмосферу.

2) Дизайн персонажів і локацій: дизайн персонажів та локацій має бути виконаний у єдиному стилі, щоб гра виглядала органічно.

3) Анімація: анімації створюються з використанням кадрів. Створюються послідовності кадрів для рухів, взаємодій та елементів оточення. Анімації мають бути синхронізовані з ігровими подіями та оптимізованими для забезпечення стабільної продуктивності, зберігаючи візуальну якість.

4) Інтеграція графіки в середовище розробки: графічні елементи, серед яких є спрайти, тайли, фони та анімації мають бути коректно імпортовані в ігрове середовище розробки, щоб забезпечити їх правильне відображення та взаємодію у грі. На цьому етапі налаштовується послідовність шарів у сцені,

масштабування та позиціонування, щоб усі елементи виглядали гармонійно та відповідали задумці

5) Програмування та налаштування логіки відображення анімацій та оточення: логіка анімацій і елементів оточення має бути запрограмована таким чином, щоб вони реагували на дії гравця та створювали ілюзію живого світу. Наприклад, залежно від того, у якому стані знаходиться персонаж відображається різна анімація. У налаштування також входить синхронізація анімацій із кодом та керування тригерами.

6) Оптимізація ігрової графіки: для забезпечення найкращої продуктивності, вся графіка, що є у грі, має пройти процес оптимізації. Для цього можуть бути використанні методи зменшення розміру спрайтів, раціональне використання графічних елементів та зменшення кількості кадрів у анімаціях.

На першому етапі розробку необхідно визначити, як має виглядати майбутня гра, розібратися з її стилем та дизайном оточення і персонажів для обрання правильних інструментів розробки спрайтів та кадрів для анімацій. Для цього необхідно добре розібратися у сюжеті гри та оцінити існуючи аналоги.

На наступному кроці відбувається розробка графіки для гри. Реалізуються всі необхідні спрайти для побудови локацій та створюються унікальні дизайни для персонажів, які зможуть підкреслити їх характер та особливості[4]. Необхідні інструменти обираються в залежності від обраного типу графіки. Основними стилями графіки для 2D ігор є:

– Мальована графіка: являє собою ручну ілюстрацію з високим рівнем деталізації. Реалізується з використанням растрової графіка у вигляді спрайтів, які в поєднані створюють одну велику картинку[5].

– Векторна графіка: це чіткі зображення, які реалізуються з використанням математичних виразів. Такі зображення не втрачають якість при масштабуванні. У даному випадку, нас цікавить не сама векторна графіка,

а стиль в іграх з векторною графікою, який часто складається з простих геометричних форм та не має великої деталізації[6].

– Піксель арт: це графіка з низькою роздільною здатністю, що дозволяє бачити чіткі пікселі. Такі ігри мають різний рівень деталізації, залежно від обраних розмірів спрайтів. Технічно, це мальована графіка з використанням растровової графіки, але її особливістю є значно менший розмір спрайтів для імітації ретро-стилю[7].

Далі потрібно інтегрувати створені графічні елементи у обране середовище розробки, де будуть проводитися подальші маніпуляції з графікою, побудова локацій, створення необхідних скриптів для прив’язки анімацій та налаштування їх правильного масштабування та швидкості відображення.

В кінці роботи необхідно бути розробити методи оптимізації графіки для забезпечення максимальної продуктивності ігрового додатку. Це може бути зменшення розміру спрайтів, оптимізації кількості кадрів при анімації або створення ігрових атласів для економії місця та зменшення ваги спрайтів.

Більшість маніпуляцій будуть вимагати близької взаємодії членів команди розробки. Для відтворення найбільш влучного дизайну для гри необхідно постійно тримати контакт з геймдизайнером, який є основним стовпом у створенні всесвіту гри та побудові правильної атмосфери для обраного наративу. Всі необхідні анімації та елементи інтерфейсу програмного додатку мають бути визначенні разом з іншими учасниками команди.

1.2 Огляд наявних аналогів

В процесі роботи було проведено пошук та аналіз схожих за своєю концепцією та ідеєю наявних інтерактивних розважальних додатків, які можуть дати натхнення та продемонструвати зацікавленість гравців у

подібних додатках. Основними іграми, які використовувались в якості прикладів в ході розробки є:

- «Valiant Hearts: The Great War»
- «Deponia»
- «When The Past Was Around»

Кожен з цих проєктів мав свій вплив на процес розробки гри. Найбільшу увагу було приділено грі «Valiant Hearts: The Great War», яка має схожі геймплейні механіки та наративні прийоми. Розглянемо кожен з цих проєктів окремо та проведемо аналіз.

«Valiant Hearts: The Great War» була створена студією Ubisoft Montpellier у 2014 році[8]. Дії гри розвертаються під час Першої світової війни. Сюжет гри розповідає історію чотирьох персонажів, які мають спільну мету – допомогти молодому німецькому солдату знайти свою кохану. Геймплей гри поєднує в собі розв’язання різноманітних головоломок, дослідження та стелс. Візуально гра виконана в коміксному стилі з використанням мальованої графіки. Кольорова палітра – приглушена, із землистими відтінками і акцентами на червоному та синьому для емоційних моментів. Після виходу гра отримала високі оцінки як від критиків, які поставили грі 81 бал, так і від гравців, що показує 95% позитивних відгуків у Steam. Виділяють історичну достовірність, дивовижний візуальний стиль та музикальний супровід. У 2014 році вона була номінована на Best Narrative у The Game Awards.

Рисунок 1.1 – Візуальний стиль гри «Valiant Hearts: The Great War»

«Deponia» була розроблена у Германії студією Daedalic Entertainment[9]. Представляє собою класичний Point-and-Click пригодницький квест із гумористичним сюжетом. Дії гри відбуваються на вигаданій планеті Депонія, де головний герой Руфус намагається знайти шлях до утопічного міста Елізіум. Особливостями гри є виразний візуальний стиль, складні головоломки та комедійний і саркастичний сюжет, підкріплений харизмою головного героя. Кольорова політра – яскрава, із контрасними відтінками, що підкреслюють комедійний та постапокаліптичний сетинг. «Deponia» отримала чисельні нагороди, зокрема за найкращу пригодницьку гру від німецьких критиків, і стала початком успішної серії з чотирьох частин. Гра отримала любов гравців та має велику кількість фанатів по всьому світу, що підтверджують 87% позитивних відгуків у Steam.

Рисунок 1.2 – Візуальний стиль гри «Deponia»

«When The Past Was Around», розроблена невеликою інді-студією Mojiken Studio в Індонезії, розповідає зворушливу історію про любов, втрату та прийняття[10]. У грі ми спостерігаємо за історією Еди – головної героїні, яка в ході сюжету має досліджувати свої спогади, розв'язуючи головоломки, щоб розкрити свою історію кохання. Увага гри зосереджена на емоційній оповіді без використання діалогових систем, яка підтримується атмосферним музикальним супроводом та мінімалістичним м'яким візуалом. Кольорова палітра – пастельна, що створює ліричну атмосферу. Гра отримала нагороди за найкращу арт-дирекцію та наратив на індонезійських The Lazy Game Awards у 2023 році. Гравці тепло прийняли гру та дали їй 95% позитивних відгуків у Steam. Цей проєкт є чудовим прикладом того, що не обов'язково робити великі багатогодинні ігри, щоб розповісти велику історію, адже історія у грі знайома кожному і не потребує пояснень.

Рисунок 1.3 – Візуальний стиль гри «When The Past Was Around»

Отже, провівши аналіз можемо зробити коротке порівняння кожної з ігор у вигляді таблиці (табл. 1.1).

Таблиця 1.1 – Порівняння наявних аналогів

	Valiant Hearts: The Great War	Deponia	When The Past Was Around
Жанр	Пригодницька гра, пазл-адвенчура	Класична point-and-click адвенчура	Point-and-click адвенчура, пазл
Геймплей	Поєднання пазлів, головоломок, стелс-елементів та дослідження. Прості механіки, акцент на наратив.	Класичний point-and-click: вирішення головоломок, взаємодія з предметами, діалоги	Прості point-and-click головоломки, зосереджені на взаємодії з об’єктами для розкриття історії.
Візуальний стиль	2D-стиль, схожий на мальовану акварель, з комікс-панелями.	Мультяшний 2D-стиль, яскравий і деталізований.	М’який, ручний малюнок у пастельних тонах
Анімації	Скелетні анімації.	Покадрова анімація.	Покадрова анімація.
Тон і атмосфера	Серйозна, зворушлива, з елементами гумору. Акцент на історичних подіях і людських долях.	Саркастична, абсурдно-комедійна, з легкими романтичними нотками.	Меланхолійна, поетична, з акцентом на емоційну глибину без діалогів.

Кожна з цих ігор має свої особливості та сильні сторони. Можна чітко побачити, як різниця дизайнів задає різність атмосфер кожної гри. Також варто приділити увагу стилю анімацій. «Valiant Hearts: The Great War» навідміну від інших проєктів використовую скелетні анімації. Такі анімації можуть дуже сильно зекономити час, так як не потребують створення кожного кадру анімації вручну. Проблемою таких анімацій є необхідність великого досвіду та вміння створювати правильні дизайни персонажів, які можна буде анімувати з використанням скелету.

Для покадрових анімацій кожен кадр малюється вручну, що робить процес більш довгим. Перевагою такого типу анімацій є живі та логічні рухи персонажів, адже малювання кожного кадру дає змогу позиціонувати героя в просторі будь яким чином, не маючи обмежень, як у скелетній анімації.

Тому подальша робота буде полягати в тому, щоб досягти задовільного рівня візуального аспекту та анімацій у грі, а також оптимізувати всю графіку у середовищі розробки, щоб забезпечити стабільну та коректну роботу графічних елементів створюваного додатку.

1.3 Постановка задачі

Основною метою даної роботи є проєктування та реалізація візуального стилю та анімаційного контенту для 2D-гри, що забезпечить створення цілісного та захопливого ігрового досвіду. Візуальний стиль має відповідати концептуальним вимогам ігрового додатку, підкреслювати особливості тематики та жанру, а також сприяти формуванню унікальної атмосфери. Анімаційний контент у свою чергу повинен забезпечити динамічність ігрового світу, плавність взаємодії та зворотній зв’язок для гравця. Дана робота виконується в рамках командного проєкту, де кожен учасник відповідає за певний аспект розробки, що вимагає узгодженості між візуальними анімаційними та іншими елементами гри, а також тісної взаємодії учасників розробки.

Досягнення цієї мети передбачає створення всіх необхідних графічних елементів для персонажів, локацій, елементів інтерфейсу та головного меню, а також анімацій, які будуть відповідати сучасним стандартам якості 2D-ігор і враховують обмеження продуктивності цільових платформ. Створенні анімації та графічні елементи будуть потребувати програмного налаштування, шляхом написання необхідних скриптів та доопрацювання скриптів, створених програмістом на початкових етапах для досягнення необхідного функціоналу та правильної логіки роботи програмного продукту.

Процес роботи можна поділити на наступні етапи:

1) Концептуалізація та створення графічного контенту:

– Визначення візуального стилю;

– Планування функціональності головного меню та інтерфейсу;

– Дизайн графічних елементів.

2) Розробка та налаштування анімацій:

– Розробка кадрів анімацій;

– Створення анімацій у середовищі розробки;

– Налаштування анімаційних контролерів для зміни станів персонажів.

3) Інтеграція та програмна підтримка

– Інтеграція графічних елементів головного меню та створення скриптів функціональних кнопок;

– Доопрацювання скриптів для керування персонажем та прив’язання до них файлів анімацій;

– Інтеграція локацій у сцени та налаштування шарів для коректного відображення елементів оточення;

– Створення допоміжних скриптів для візуалізації міні-ігор.

4) Тестування, оптимізація та фіналізація:

– Тестування відображення графічних елементів, виявлення наявних дефектів;

– Оптимізація графічного контенту;

– Передача додатку на доопрацювання іншим членам команди.

Висновки до розділу

У розділі було проведено комплексний аналіз предметного середовища, пов’язаного з розробкою візуального стилю та анімаційного контенту для пригодницького квесту з елементами інтерактивної драми. Було надано визначення 2D ігор та процесу розробки 2D ігор. Були описані основні складові обраного жанру та детально розглянуто процес діяльності для успішної реалізації програмного продукту.

У рамках огляду наявних аналогів було визначено актуальність створюваного розважального додатку, актуальність наративної складової сюжету та різність підходів до візуального оформлення ігор. Порівняння аналогів дозволило зробити обґрунтований вибір середовища розробки гри та визначити необхідний графічний стиль.

На основі попереднього аналізу було визначено постановку задачі, яка включає призначення створюваного додатку та цілі і задачі розробки, до яких входить покроковий план розробки візуального стилю проєкту та технічного налаштування графіки у середовищі розробки.

РОЗДІЛ 2
ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Аналіз предметної області

Предметна область даної роботи охоплює створення макетів для інтерфейсу 2D-гри, скриптів для реалізації логіки роботи UI-елементів, створення та інтеграцію спрайтів для відтворення персонажів та оточення в ігровому просторі, а також реалізацію префабів для зручної роботи з міні-іграми.

2.1.1 Концепції роботи інтерфейсів користувача

Інтерфейси користувача являють собою набір елементів, які дозволяють користувачу керувати можливостями гри, до яких можуть входити налаштування гри, можливість зберегти наявний прогрес або вийти з ігрового додатку. Усі елементи мають бути реалізовані логічно та лаконічно розміщуватись на екрані користувача. Для досягнення цієї цілі спочатку будуть реалізовані описи кожного меню для повного розуміння функціоналу та логіки роботи кожного з елементів після чого можна буде перейти до концептуального проєктування систем.

2.1.1.1 Головне меню

Головне меню буде першим, що побачить користувач при запуску інтерактивного додатку. Воно має бути провідником, яке дасть змогу налаштувати гру відповідно до системних особливостей користувача, розпочати гру шляхом вибору одного з доступних рівнів або продовжити гру з останньої контрольної точки. Відповідно до задач проєкту було визначено необхідність створення таких кнопок головного меню як: «Play», «Settings», «Credits», «Report bug», «Quit» (табл. 2.1).

Таблиця 2.1 – Опис функціоналу головного меню.

Кнопка	Функція
Play	Переводить користувача до меню вибору рівня.
Settings	Переводить користувача до меню налаштувань.
Credits	Переводить користувача до меню титрів, де будуть перелічені розробники гри.
Report bug	Надає користувачу посилання на форму заповнення інформації про знайдені баги або технічні проблеми.
Quit	Завершує гру та закриває ігровий додаток.

2.1.1.2 Меню вибору рівня

Меню вибору рівня необхідно для того, щоб користувач мав змогу запустити необхідний рівень. На початку гри доступним буде лише перший рівень. Всі інші будуть відкриватися в ході проходження сюжету гри. Дане меню має містити в собі кнопки, кожна з яких буде відповідати за відповідний ігровий рівень. Також мають бути реалізовані кнопки «Back» та «Reset» для зручного переходу між вікнами головного меню та скидання прогресу у грі (табл. 2.2).

Таблиця 2.2 – Опис функціоналу меню вибору рівнів.

Кнопка	Функція
Вибір рівня	Кількість кнопок вибору рівнів буде залежати від кількості рівнів у грі. Повний обсяг рівнів буде відомим тільки у кінці розробки проєкту. Для початку було реалізовано 6 кнопок для вибору рівнів, з яких доступною є лише одна.
Back	Повертає користувача на екран головного меню
Reset	Дає змогу користувачу скинути наявний прогрес у грі та почати проходження сюжету спочатку.

2.1.1.3 Меню налаштувань

Меню налаштувань є необхідним для повноцінної роботи проєкту. Воно дає змогу користувачу змінювати налаштування гри для отримання найбільш

якісної картинки або зміни рівня гучності музичного супроводу та звукових ефектів. На даному етапі проєкту було вирішено розробити два блоки налаштувань: «Volume Settings» та «Screen Settings». Кожен блок буде містити базові функції, які дають змогу налаштувати гру відповідно до вподобань користувача, але програмна реалізація блоку з налаштуваннями звукового супроводу буде передано учаснику проєкту відповідному за звуковий супровід (табл. 2.3).

Таблиця 2.3 – Опис функціоналу меню налаштувань.

Блок налаштувань	Поле налаштувань	Функція
Volume Setting	Master Volume	Повзунок, що відповідає за загальну гучність гри.
	Music Volume	Повзунок, що відповідає за гучність музики у грі.
	AFX Volume	Повзунок, що відповідає за гучність звуків оточення.
	SFX Volume	Повзунок, що відповідає за гучність звуків ефектів.
Screen Settings	Resolution	Випадаючий список, який дає користувачу змогу обирати необхідне розширення екрану.
	FullScreen	Перемикач, що відповідає за те чи знаходиться гра у вікні або розгорнута на весь екран.
	VSync	Перемикач, що відповідає за включення вертикальної синхронізації.

2.1.1.4 Меню паузи

Це меню буде викликатися під час гри для того, щоб поставити гру на паузу, перейти у вікно налаштувань, щоб змінити параметри не перериваючи ігровий процес або повернутися на екран головного меню. Для реалізації необхідного фукнціоналу будуть створенні наступні кнопки: «Continue», «Settings» та «Exit» (табл. 2.4).

Таблиця 2.4 – Опис функціоналу меню паузи

Кнопка	Функція
Continue	Закриває вікно меню паузи та повертає користувача до гри.
Settings	Переводить користувача до меню налаштувань.
Exit	Зупиняє ігровий процес та повертає користувача до головного меню.

2.1.2 Концепція зміни станів анімацій

Усі анімації головного персонажа та неігрових персонажів будуть реалізовані за допомогою кадрових анімацій. Кадри будуть створені вручну за допомогою програмного забезпечення для малювання після чого вони будуть імпортовані у середовище розробки Unity для подальшого налаштування та прив’язки до скриптів.

2.1.2.1 Анімації персонажів

Створення анімацій для ігрових персонажів багатокроковий процес. Перш за все необхідно створити унікальні дизайни для кожного персонажа. Дизайн кожного персонажа має бути обговорено зі сценаристом для найбільш вдалого влучення у характер персонажу. Після створення необхідного дизайну можна приступати до реалізації кадрів анімацій, які знадобляться в подальшому. Кожен кадр створюється вручну та має бути достатньо деталізованим для отримання якісної картинки у грі. Далі кадри імпортуються у Unity, де мають бути перетворені у повноцінні анімації з розширенням .anim. Таким чином створюються анімації, які в подальшому будуть використовуватися для відображення дій користувача. Цей процес має бути виконаний для всіх анімацій, необхідних у грі (табл. 2.5).

Таблиця 2.5 – Опис анімація персонажа

Анімація	Опис
1	2
Стан спокою	Базова анімація, яка буде програватись завжди, коли персонаж знаходиться на місці.

Продовження таблиці 2.5

1	2
Пересування	Анімація, що буде програватись, коли персонаж буде змінювати своє положення відносно осі абсцис.
Взяти	Анімація підбору/взаємодії з предметами у грі.
Стан спокою(Присід)	Базова анімація, яка буде програватись, коли персонаж знаходиться у стані присіду та знаходиться на одному місці.
Пересування(Присід)	Анімація, що буде програватись, коли персонаж знаходиться у стані присіду та змінювати своє положення відносно осі абсцис.
Піднятися(Невелика перешкода)	Анімація, що буде програватися при взаємодії гравця з невеликою перешкодою.
Стан спокою(Драбина)	Анімація, яка буде програватися, коли персонаж буде взаємодіяти з драбиною та знаходитись на одному місці.
Пересування(Драбина)	Анімація, що буде програватись, коли персонаж буде знаходитись на драбині та змінювати своє положення відносно осі ордината.
Піднятися(Драбина)	Анімація, що буде програватися при досягнені персонажем верхньої точки на драбині.
Підняти предмет	Анімація, що буде програватися при взаємодії персонажа з інтерактивними елементами оточення.
Діалог	Анімація, що буде програватись при взаємодії з іншими персонажами.

Подальша робота буде потребувати створення у середовищі розробки Unity компоненту Controller, в якому буде відбуватися налаштування змін станів для персонажу в залежності від дій гравця. Для коректної роботи зміни станів під час анімацій необхідно буде створити параметри, які будуть об’єднувати скрипти для керування персонажем та анімації, що будуть програватися відповідно до дій користувача.

2.1.2.2 Діалогова система

В рамках роботи з системою діалогів задача буде полягати в налаштуванні коректної роботи скрипта QuestGiverInteraction, створеного іншим учасником проєкту. В ході налаштування до скрипта будуть під’єднанні необхідні аніматори для відображення анімацій під час діалогів, а також створенні діалогові вікна, які будуть слугувати джерелом передачі інформації для користувача.

2.1.3 Концепція ігрового простору

До елементів ігрового простору будуть відноситися всі елементі, що будуть оточувати головного персонажа у всесвіті гри. До таких елементів належать локації, в яких будуть проходити ігрові події, елементи, з якими буде взаємодіяти гравець та міні-ігри. Для більшого занурення гравця у всесвіт гри та створення ефекту глибини на деяких локаціях буде реалізовано скрипт для відтворення ефекту паралаксу. Цей ефект буде доречно використовувати на відкритих локаціях, наприклад, у місті.

2.1.3.1 Локації

Всі спрайти для побудови рівнів та локацій будуть намальовані вручну, як і анімації. Для цього будуть створені макети локацій разом із геймдизайнером, На основі яких будуть реалізовані дизайни локацій. В подальшому готові локації будуть розбиватися на менші елементи та імпортуватися в середовище розробки. У Unity спрайти пройдуть етапи підготовки та будуть розміщені у сценах, які являють собою ігрові рівні.

Для зручної роботи необхідно буде розподілити всі спрайти на класи, які відіграють роль шарів. Так як середовище розробки 2D-гри має обмеження у вигляді відсутності третьої координати, то відповідно у грі відсутня глибина у кадрі і всі об’єкти розміщуються на одній площині. Тому для того, щоб запобігти некоректного рендерингу елементів на локаціях необхідно буде призначити кожному спрайту пріоритет, який буде визначати, який з елементів має відображатися поверх інших.

На даному етапі вже з’являється можливість керувати анімованим персонажем, пересуваючись у готових локаціях ігрового простору. Далі можна перейти до реалізації ефекту паралаксу, який зробить зображення живим та об’ємним. Для досягнення цієї цілі необхідно буде реалізувати скрипти, які будуть працювати разом, будуть модульними та універсальними завдяки чому можна буде використовувати їх на будь яких об’єктах. Ця система буде складатися з трьох основних компнонентів:

– Скрипт для відстеження руху камери: цей скрипт буде відповідати за визначення зміщення камери відносно осі абсцис та буде передавати цю інформацію іншим компонентам.

– Скрипт для окремих фонових шарів: цей скрипт буде прив’язуватися до кожного шару та визначатиме швидкість, з якою ці шари будуть рухатися відносно камери. Це дозволить індивідуально налаштовувати кожен шар для досягнення найкращого результату.

– Скрипт для керування шарами: цей компонент має координувати взаємодію між камерою та всіма шарами, забезпечуючи їх синхронний рух.

2.1.3.2 Міні-ігри

Розробка міні-ігор буде відбувати паралельно з іншими учасниками команди. Задачею при створені міні-ігор буде реалізація префабів для полегшення подальшої розробки. Робота буде проходити над двома міні-іграми, які мають назви «RushHour» та «TurnOnOff». Для створення префабу необхідно створити окремий об’єкт у середовищі Unity, який потім зберігається як заготівля для майбутній об’єктів такого ж типу. Це полегшить роботу над реалізацією міні гри та забере необхідність налаштовувати кожен окремий об’єкт в подальшому.

Для гри «RushHour» будуть створюватися префаби блоків різної розмірності, з який потім будуть будуватися рівні різної складності. Всього будуть створені п’ять таких елементів: «Блок 1x2», «Блок 1x3», «Блок 2x1», «Блок 3x1» та «Ігровий блок». До кожного блока буде додаватися основний скрипт міні-гри, у якому прописана логіка роботи. Це потрібно, щоб

забезпечити коректну роботу програмної частини міні гри та її візуальної складової. Після реалізації префабів, буде написаний скрипт, який дозволить динамічно малювати сітку, на якій будуть розміщені блоки для гри. У скрипті буде реалізовано можливості налаштування кількості стовпців та рядків, розміри полів, відстані між полями та товщину лінії сітки.

Для гри «TurnOnOff» буде реалізовано лише один префаб, який представляє собою елемент «Тумблер», який буде мати стани «On» та «Off». Залежно від поточного стану тумблеру має відображатися відповідний спрайт, який буду надавати гравцю інформацію щодо його дій. Після створення префабів буде реалізовано скрипт для коректної поведінки тумблера. Він має підтримувати зміну стану між «On» та «Off», а також змінювати стан інших пов’язаних з ним перемикачів.

2.2 Проєктування системи

До проєктування системи входить побудова концептуальної моделі для відображення логіки роботи всіх створюваних елементів. Будуть розроблені UML-діграми для візуального подання логіки роботи елементів меню, а також зв’язки зміни станів для всіх анімацій ігрового персонажу, що буде реалізовано у компоненті Animation Controller. Цей процес включає визначення необхідних параметрів для реалізації зв’язків, їх найменування та роботу зі скриптами.

2.2.1 Проєктування інтерфейсів користувача

Проєктування інтерфейсів користувача включає в себе розробку логіки роботи усіх елементів меню, наявних у інтерактивному додакту. Меню розроблені головні інтерфейси для взаємодії користувача з грою, що забезпечують доступ до основних функцій додатку. Кожне меню має свій функціонал та стилізовані дизайни, що відповідають стилю гри. Для наочного розуміння логіки роботи майбутніх інтерфейсів була розроблена UML-

діаграма, що дасть змогу полегшити розробку кожного меню в подальшому (див. додаток А).

Дана UML-діаграма демонструє логіку роботи всіх елементів, що будуть реалізовані у інтерфейсі користувача.

Реалізація меню буде відбуватися у середовищі розробки Unity за допомогою системи Canvas, із використанням UI-компонентів, які в подальшому будуть налаштовуватися, та мальованих спрайтів. У середовищі розробки Unity реалізовано стандартний набір UI-елементів, який буде використовуватися для реалізації інтерфейсів (табл. 2.6).

Таблиця 2.6 – Стандартні UI-елементі Unity

UI-елемент	Опис	Типові застосування
1	2	3
Canvas	Кореневий об’єкт, що містить усі UI-елементи. Відповідає за визначення способу рендерингу.	Організація та структуризація всіх UI-елементів на сцені.
Panel	Контейнер для групування інших UI-елементів. Містить у собі компонент Image, як виступає фоновим зображенням.	Групування елементів меню або діалогів.
TextMeshPro – Text	Текстовий елемент із широкими можливостями форматування тексту.	Створення тексту у високій якості. Можу використовуватись для найменування кнопок, або іншого тексту
Image	Елемент, що відображає графічні зображення.	Використовується для графічного оформлення кнопок, логотипів або фонів
Button	Інтерактивна кнопка	Використовується для реалізації кнопок або інших елементів меню
Toggle	Перемикач.	Налаштування, які потребують лише два положення.

Продовження таблиці 2.6

1	2	3
Slider	Повзунок для вибору значень в діапазоні.	Налаштування гучності звуку або яскравості зображення.
Scrollbar	Смуга прокрутки.	Прокрутка списків.
Dropdown	Випадаючий список	Вибір розмірності вікна додатку
Input Field	Поле введення тексту користувачем.	Форми заповнення та інші елементі інтерфейсу, які потребують введення тексту.

Структура створюваного інтерфейсу буде складатися з наступних елементів:

– Canvas: основний контейнер, що буде містити в собі всі меню, наявні у сцені та створені у них UI-елементи. Налаштований у режимі «Scale With Screen Size» для адаптації до роздільних здатностей відповідно до розміру монітору.

– Panel: цей елемент буде створюватися для кожного окремого меню. Кожен Panel буде зберігати в собі елементі відповідного меню. Наприклад Panel для головного меню буде містити в собі всі елементи визначені під розробки концепції головного меню (див. табл. 2.1).

– UI Button: UI-елементи, що відіграють роль кнопок. Будуть потребувати подальшого налаштування таких компонентів як Image та Button, щоб надати елементу необхідний зовнішній вигляд та функціонал. Всі елементи типу Button було визначено заздалегідь у діаграмі інтерфейсів користувачів (див. додаток А).

– UI Image: UI-елементи, що являють собою зображення. До таких зображень будуть відноситися логотипи та фонові зображення. Потребують налаштування компонентів Image, де будуть призначені необхідні зображення та проведені налаштування для коректного відображення заданих елементів.

– TextMeshPro – Text: буде міститися майже в кожному UI-елементі, щоб користувач наочно розумів, зо що відповідає кожна кнопка, повзунок або

перемикач. Містить в собі компонент TextMeshPro, в якому шрифти будуть налаштовані для забезпечення задовільного вигляду.

– Toggle: перемикач переважно буде використовуватися у меню налаштувань для реалізації таких функцій, як «Full Screen» та «VSync» (див. додаток А).

– Slider: повзунок буде використовуватися у меню налаштувань для реалізації функцій налаштувань гучності звуків (див. додаток А). Реалізація функціоналу повзунків буде передана учаснику проєкту відповідному за створення звукового супроводу.

– Scrollbar: може знадобитися в тому випадку, якщо деякі елементи не будуть вміщуватися на екран користувача. Наприклад, смугу прокрутки можна використати у випадаючому списку налаштування розміру вікна екрану.

– Dropdown: випадаючий список буде використано у меню налаштувань для реалізації опції вибору розмірності вікна застосунку (див. додаток А).

– Input Field: поля введення тексту не буде застосовано на даному етапі проєкту.

Створивши всю необхідну ієрархію елементів, інтерфейси можуть бути реалізовані у сценах Unity відповідно до макетів, які будуть розроблені в подальшому. Далі всі створенні елементи мають бути прив’язані до менеджерів MenuManager та SettingsManager, які містять в собі компоненти Menu Controller та Settings Manager відповідно. Скрипти для цих компонентів були створені іншим учасником проєкту та потребують прив’язки і налаштування UI-елементів для коректної роботи.

2.2.2 Проєктування змін станів анімацій

До проєктування змін станів анімацій відноситься створення логіки роботи анімацій та їх заміни залежно від дій, які виконує гравець. Для налаштування коректної зміни анімацій необхідно буде створити параметри, що будуть перевіряти, чи виконуються умови для зміни анімації. На даному

етапі будуть визначені зв’язки для виконання анімацій, визначено, що таке параметри та які вони бувають, а також створено концептуальну модель у вигляді UML-діаграми для наочного представлення роботи логіки анімацій.

Animator у Unity є дуже потужним інструментом для створення анімацій для ігрових персонажів та об’єктів. Для налаштування переходів між анімаціями у Animator існують параметри – змінні, які створюються у Animator Controller та дозволяють керувати логікою переходів між анімаціями або впливати на самі анімації. Вони діють, як посилання для скриптів або умов переходу, дозволяючи гнучко налаштовувати анімацію залежно від стану гри.

Всього Animator підтримує чотири типи параметрів:

– Float: використовується для значень із плаваючою точкою, наприклад, для визначення швидкості персонажа.

– Int: використовується для цілих чисел. Може використовуватися для вибору конкретного стану або індексу анімації.

– Bool: приймає значення true або false. Використовується для ввімкнення або вимкнення певного стану, наприклад, для визначення чи знаходиться персонаж у положенні присіду.

– Trigger: спеціальний тип, який діє як одноразовий сигнал для запуску переходу. Після спрацювання автоматично скидається. Використовується для разових подій, наприклад, для програвання анімації пострілу, підйому на перешкоду, взаємодії з предметом та інші.

Після створення концептуальної моделі зв’язків для переходів між анімаціями буде визначено кількість та типи необхідних параметрів.

Концептуальна модель у вигляді UML-діаграми буде створюватися на основі концепції анімацій персонажу (див. табл. 2.5). У ній будуть представлені всі необхідні для реалізації анімації та зв’язки зміни станів між ними (рис. Б.1).

Таким самим чином була розроблена концептуальна модель для противників, яка складається з кількох анімацій (рис. Б.2).

На даному етапі розробки гри противники мають вміти патрулювати та спостерігати за оточенням в пошуках головного героя. Для цих цілей були створенні лише два стани, які підтримують знаходження ігрової моделі на одному місці та програвання анімації пересування, що буде відбуватися під час патрулювання.

Відповідно до створених концептуальних моделей було визначено необхідність створення наступних параметрів: HorizontalMove, Climb, Crouch, OnLadder, ClimbSpeed, OnTopClimb та PickUp (табл. 2.7).

Таблиця 2.7 – Перелік параметрів для реалізації логіки анімацій

Назва	Тип	Опис
1	2	3
HorizontalMove	Float	Параметр анімації, який відображає інтенсивність горизонтального руху гравця. Він використовується для перемикання між станами анімації в залежності від того, чи гравець рухається та з якою швидкістю.
Climb	Trigger	Тригер, який ініціює анімацію підйому персонажа на невелику перешкоду.
Crouch	Bool	Булевий параметр у Animator, який синхронізує анімаційний стан присідання з внутрішньою логікою персонажа. Наприклад, коли персонаж присідає, цей параметр набуває значення «true», запускаючи відповідну анімацію.
OnLadder	Bool	Булевий параметр у Animator, який синхронізує стан знаходження гравця на драбині з анімаційною системою. Він встановлюється в «true», коли гравець взаємодіє з драбиною, щоб почати по ній лізти, і повертається в «false», коли він злазить або залишає зону драбини.
ClimbSpeed	Float	Параметр анімації, який відображає інтенсивність горизонтального руху гравця у стані присіду. Він використовується для перемикання між станами анімації в залежності від того, чи гравець рухається та з якою швидкістю.

Продовження таблиці 2.7

1	2	3
OnTopClimb	Trigger	Тригер, який ініціює анімацію підйому персонажа, коли він досягає крайньої точки драбини.
PickUp	Trigger	Тригер, який ініціює анімацію підбору предмета, коли гравець взаємодіє з об’єктом.

Дані параметри будуть використанні при налаштуванні елементів Transitions, що будуть перевіряти, яку анімацію необхідно програвати в залежності від дій гравця.

2.2.3 Проєктування ігрового простору

2.2.3.1 Інтеграція та налаштування спрайтів у середовищі розробки

Ігровий простір буде створюватися у середовищі Unity з використанням спрайтів, що будуть створенні при розробці дизайну локацій та рівнів. Побудова локацій у Unity відбувається шляхом додавання елементів до сцени, де вони розміщуються відповідно до макету та налаштовуються для коректного відображення. Так як інтерактивний додаток створюється у 2D середовищі, то він потребує правильного налаштування шарів, щоб елементи локацій не перекривали один одного. Для цього у середовищі розробки існують компонент Sprite Renderer, що містить в собі можливості сортування шарів Sorting Layer та параметр пріоритету для елементів на кожному з шарів Order in Layer.

Для Sorting Layer будуть створенні шари для створення логічної ієрархії об’єктів у сцені та також коректного відображення елементів під час гри. В ході розробки було визначено необхідність створення наступної структури Sorting Layer:

1) Default: шар існуючий у середовищі розробки за замовченням.
2) Backround: шар, що буде містити в собі елементів заднього плану, наприклад, фонові зображення, стіни, пейзажі та інше.

3) NPC: шар, на якому будуть розташовані неігрові персонажі, з якими зможе взаємодіяти гравець.

4) Cover: шар, що буде зберігати в собі елементи укриття, в яких гравець зможе сховатися, щоб запобігти зустрічі з противником або сховатися від небезпеки.

5) PlayerWhistle: шар для зберігання спрайтів свисту гравця, яким можна буде відволікти ворожих персонажів.

6) Interior: шар, для зберігання елементів інтер'єру. У ньому будуть зберігатися більшість елементів сцени.

7) InteractibleObjects: виконує таку ж саму функцію, що і шар для елементів інтер'єру, але елементи у цьому шарі будуть мати можливість взаємодії для гравця. До таких елементів можуть відноситися об'єкти для виконання квестових завдань.

8) Player: на цьому шарі будуть відображатися спрайти головного героя, до яких відносяться всі кадри анімацій.

9) Enemy: шар для зберігання спрайтів ворожих персонажів.

10) FrontGround: шар для відображення елементів переднього плану. Вони призначення для створення об'ємності сцени, шляхом додавання ілюзії глибини.

Кожен шар Sorting Layer буде мати свій власний пріоритет елементів Order in Layer, який буде підлягати налаштуванню під час створення локацій.

2.2.3.2 Проєктування системи паралакс ефекту

У рамках розробки інтерактивного ігрового додатку буде реалізовано систему паралакс ефекту – візуальної ілюзії глибини, яка створюється шляхом руху фонових шарів із різною швидкістю відносно руху камери. Така система дозволяє візуально розширити простір і зробити сцену динамічнішою, не використовуючи повноцінної тривимірної графіки.

Для досягнення цієї мети на етапі проєктування було прийнято рішення розділити систему на три основні компоненти, кожен з яких виконує

конкретну функцію. Це дозволяє дотримуватися принципу розділення обов’язків і спрощує подальшу реалізацію та підтримку коду.

До компонентів системи відносяться три скрипта, що будуть мати свої функції:

– Контролер камери (ParallaxCamera): скрипт, що буде відповідати за відстеження горизонтального руху камери. Його завдання – фіксувати зміщення позиції по осі абсциса та передавати відповідний сигнал іншим компонентам через механізм подій. Таким чином забезпечується незалежність логіки камери від логіки руху фону.

– Шари паралаксу (ParallaxLayer): кожен об’єкт фону, який повинен брати участь у паралакс-ефекті, реалізується як окремий шар. Основним параметром цього компонента буде коефіцієнт паралаксу (parallaxFactor), який визначає, наскільки швидко об'єкт рухається при зміщенні камери. Чим менше значення коефіцієнта, тим повільніше рухається шар, і тим далі він візуально сприймається.

– Менеджер шарів (ParalaxBackground): цей компонент відповідає за координацію усієї системи. Він підписується на подію зміщення камери, знаходить усі дочірні шари (ParallaxLayer) на сцені та передає їм відповідну інформацію. Таким чином, коли камера буде змінювати своє положення шари будуть автоматично оновлювати свої позиції.

На основі визначених функціональних обов’язків було сформовано концептуальну модель у вигляді діаграми класів для формування чіткої структури майбутньої системи (рис. Б.3).

Проєктована система є модульною, масштабованою та легко конфігурованою. Її реалізація дозволить ефективно створити глибину сцени у 2D-просторі без зайвих витрат на обчислення. Подієва модель взаємодії між компонентами забезпечує низький рівень залежності та високу гнучкість, що є важливою вимогою для сучасних ігрових систем.

2.2.3.3 Проєктування скрипта для міні гри Rush Hour

У рамках розробки користувацького інтерфейсу для міні-гри постала необхідність реалізації графічного елемента, здатного виводити сітку з динамічними параметрами. Такий елемент повинен мати можливість візуалізації рядків і стовпців, що формують прямокутну сітку, а також забезпечувати налаштування розміру клітинок, міжрядкових відступів і товщини ліній. Крім того, важливо забезпечити повну сумісність з UI-системою Unity, що зумовлює вибір спадкування від базового класу Graphic. У зв’язку з цим заплановано реалізацію кастомного компонента GridGraphic, який буде відповідати усім зазначеним вимогам.

Перед створенням концептуальної моделі необхідно визначити набір функціональних можливостей, які має підтримувати майбутній компонент (табл. 2.8).

Таблиця 2.8 – Набір функціональних можливостей компоненту GridGraphic

Назва параметру	Призначення
gridWidth	Кількість вертикальних ліній, що визначає кількість стовпців сітки
gridHeight	Кількість горизонтальних ліній, що визначає кількість рядків сітки
cellSize	Розмір клітинки
spacing	Відстань між клітинками
lineThickness	Товщина ліній, якими будується сітка
color	Колір ліній

Ключова задача компонента – в межах заданої області побудувати двовимірну сітку з ліній, що візуально формують клітинки. Вся побудова здійснюватиметься в методі OnPopulateMesh, шляхом додавання ліній як прямокутників у VertexHelper.

Для кращого розуміння структури та взаємозв’язків між частинами майбутньої реалізації побудовано концептуальну модель у вигляді UML-діаграми (рис. Б.4).

Створена модель зможе надати допомогу в подальшій реалізації скрипта.

2.2.3.4 Проєктування скрипта для міні гри TurnOnOff

Для можливості реалізації міні гри, заснованої на механіці взаємодії з перемикачами необхідно розробити скрипт SwitchToggle, який дозволить змінювати стан об’єкта при взаємодії з ним, а також буде впливати на інші пов’язані перемикачі.

Функціонал даного компоненту буде включати:

- зміну стану елементу при натисканні;
- візуальне оновлення спрайту залежно від стану;
- взаємодія з іншими перемикачами;
- виклик глобальної перевірки стану через менеджер.

Перед створенням концептуальної моделі необхідно визначити набір функціональних можливостей, які має підтримувати майбутній компонент (табл. 2.9).

Таблиця 2.9 – Набір функціональних можливостей компоненту SwitchToggle

Назва параметру	Призначення
isOn	Логічна змінна, яка зберігає поточний стан перемикача
onSprite, offSprite	Спрайти, що відображають стан перемикача
connectedToggles	Список інших перемикачів, на які впливає цей перемикач при натисканні
Toggle()	Метод, який змінює стан перемикача та оновлює його вигляд
UpdateVisual()	Змінює спрайт в залежності від стану isOn
OnSwitchClicked()	Обробник натискання: перемикає себе, пов’язані елементи і викликає перевірку стану гри

Передбачається, що кожен перемикач буде інтерактивною кнопкою з візуальним представленням у вигляді Image, яка реагує на події натискання. Це має працювати наступним чином:

1) поточний перемикач змінює стан на протилежний;

2) усі зв’язані перемикачі також змінюють стан;

3) перемикачі автоматично оновлюють зображення;

4) викликається метод CheckVictory() з зовнішнього менеджера SwitchPanelManager.

Таким чином, реалізується головоломка, в якій взаємозалежність перемикачів дозволяє створювати цікаві ігрові сценарії.

Для кращого розуміння структури та взаємозв’язків між частинами майбутньої реалізації побудовано концептуальну модель у вигляді UML-діаграми (рис. Б.5).

Компонент SwitchToggle дозволяє реалізувати модульну систему перемикачів для міні-ігрових механік. Гнучкість структури дозволяє легко налаштовувати взаємозв’язки між перемикачами у редакторі Unity без зміни коду. Така архітектура значно спрощує подальше створення головоломок і забезпечує повторне використання компонента у різних контекстах.

2.3 Оптимізація графічного контенту у 2D-грі

У процесі розробки відеоігор, зокрема 2D-проєктів, значна увага приділяється не лише створенню якісного візуального оформлення, але й забезпеченню ефективної роботи гри на різних пристроях. Це є особливо актуальним для мобільних платформ або пристроїв із обмеженими технічними можливостями, де надмірне навантаження на графічний процесор (GPU) може призвести до зниження частоти кадрів, зависань та погіршення загального користувацького досвіду.

Оптимізація графіки – це процес зменшення обсягу ресурсів, необхідних для виведення зображення на екран, без істотного погіршення візуальної

якості. На сьогодні розроблено велика кількість методів оптимізації ігор, серед яких є і ті, що використовуються в 2D проєктах. До таких методів відносяться: використання атласів текстур, зменшення роздільної здатності спрайтів, компресія текстур та інші. Даний підрозділ буде присвячено визначенню та аналізу кожного з цих методів для визначення плюсів та мінусів, які їх супроводжують.

2.3.1 Атласи текстур

Атлас текстур (Sprite Atlas) – це технологія об'єднання багатьох окремих спрайтів в одну велику текстуру. Unity автоматично створює та керує такими атласами, дозволяючи відображати окремі спрайти на основі координат у загальному зображенні. Основна мета цієї техніки полягає в оптимізації рендерингу за рахунок зменшення кількості перемикань текстур (draw calls) під час виводу зображень на екран (рис. 2.7).

Рисунок 2.7 – Приклад використання атласу для зберігання великої кількості кадрів для анімацій

Draw call – це запит до GPU на відображення конкретного об'єкта з певною текстурою. Якщо кожен спрайт має власну текстуру, таких запитів

може бути сотні, що значно навантажує графічний процесор. Використовуючи атласи текстур, велика кількість об'єктів рендериться в одному або кількох draw call, що суттєво підвищує продуктивність.

Застосування спрайтових атласів особливо ефективне у таких випадках:

– анімації персонажів, де кілька кадрів зберігаються як частини одного зображення;

– користувацький інтерфейс (UI), де багато дрібних іконок або кнопок можуть бути об'єднані в один атлас;

– повторювані фонові або декоративні об'єкти (дерева, предмети інтер’єру тощо).

Крім того, атласи полегшують керування ресурсами: художник має змогу централізовано налаштовувати стиснення, фільтрацію, роздільність тощо.

2.3.2 Зменшення роздільної здатності спрайтів

У більшості 2D-проєктів рендеринг візуальних об'єктів на екрані відбувається у зменшеному вигляді порівняно з роздільністю їхніх текстур. Випадки, коли великі спрайти (наприклад, 2048×2048 пікселів) використовуються для відображення об'єкта розміром 200×200 пікселів, є типовими прикладами неефективного використання ресурсів.

Зменшення роздільної здатності спрайтів до реальних потреб дає змогу зменшити обсяг відеопам’яті, необхідної для зберігання текстур у VRAM, прискорити завантаження сцен та поліпшити загальну стабільність гри, особливо на слабких пристроях.

Рекомендується використовувати текстури з розмірами, кратними ступеням двійки (наприклад, 128×128, 512×512). Це пояснюється тим, що більшість графічних процесорів (GPU), а також графічні програмні інтерфейси, такі як OpenGL або DirectX, оптимізовані саме для роботи з такими текстурами. Внутрішня структура відеопам’яті та алгоритми обробки зображень побудовані так, що дані з текстур із «стандартними» розмірами

зчитуються швидше, з меншими затримками і без потреби в додаткових обчисленнях або вирівнюванні пам’яті.

Звісно, зменшення роздільної здатності повинно бути зваженим – важливо зберегти баланс між економією ресурсів і прийнятною візуальною якістю. Зазвичай такий підхід доцільно застосовувати до фонових об’єктів, декоративних елементів, які не є фокусними та другорядних NPC або об'єктів, що з'являються епізодично.

2.3.3 Компресія текстур

Компресія текстур – це ще один важливий інструмент оптимізації, який дозволяє значно зменшити обсяг графічних ресурсів у збірці гри. Unity підтримує низку форматів текстурної компресії, кожен із яких призначений для певної платформи.

Основні формати:

– ETC (Ericsson Texture Compression) використовується для Android, є стандартом компресії для багатьох пристроїв;

– ASTC (Adaptive Scalable Texture Compression) – сучасний формат із гнучкими параметрами якості та ефективності;

– PVRTC (PowerVR Texture Compression) використовується для iOS;

– DXT (S3TC) використовується для ПК-платформ.

Компресія дозволяє зменшити розмір фінальної збірки, пришвидшити завантаження сцен, знизити навантаження на GPU та уникнути перевантаження текстурного кешу.

Компресія може призводити до невеликої втрати якості, однак у більшості випадків це непомітно для гравця, особливо якщо йдеться про другорядні об'єкти або фонові елементи. У Unity компресію можна налаштовувати окремо для кожної платформи, що дозволяє досягти максимальної ефективності на різних пристроях.

Висновки до розділу

У другому розділі було проаналізовано структуру ключових ігрових механік, які формують взаємодію користувача з ігровим середовищем. Особливу увагу приділено реалізації таких елементів, як система діалогів і квестів, механіка стелсу, інвентар, зміна станів анімацій, а також міні-ігри, які урізноманітнюють геймплей.

Визначено загальну архітектуру взаємодії між об’єктами гри, створено логіку переходів між станами, реакції на дії гравця та внутрішню послідовність виконання сценаріїв. Окреслено роль кожного компонента у підтримці цілісності ігрового процесу.

У результаті аналізу сформовано структуровану модель функціонування ігрової системи, яка є технічною основою для подальшої реалізації графічного та анімаційного контенту в межах проєкту.

РОЗДІЛ 3
ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Засоби розробки

Для створення 2D-гри в жанрі пригодницького квесту з елементами інтерактивної драми було використано набір програмних продуктів, що забезпечують реалізацію візуального стилю, анімаційного контенту, програмної логіки та оптимізації. Кожен інструмент мав певний вплив на процес розробки та відповідав за певний етап: створення графіки, написання коду, інтеграція активів, тестування.

Unity 2022.3 LTS є основним середовищем розробки гри, що забезпечує створення 2D-сцен, інтеграцію графічних активів, анімацій і програмної логіки [11].

Visual Studio 2022 Community Edition використано як основне середовище для написання, тестування та налагодження коду на мові програмування C# [12]. Воно інтегровано з Unity для створення скриптів, які реалізують логіку руху персонажів, діалогову систему, ефект паралаксу та міні-ігри.

Krita 5.2.9 – інструмент із відкритим кодом для цифрового малювання, використано для створення мальованої графіки гри, включаючи спрайти персонажів, фони та об’єкти оточення [13]. За допомогою Krita створено покадрові анімації.

Adobe Photoshop 2024 використано для підготовки макетів для інтерфейсів користувача: головного меню, меню вибору рівнів, меню налаштувань, меню паузи та інші [14]. Також були створенні UI-елементи, які використовувалися для реалізації меню у середовищі розробки Unity.

PureRef 2.0 – допоміжна програма для організації референсів під час створення графіки [15]. Вона дозволила зібрати та систематизувати

зображення, ескізи, кольорові палітри для персонажів, локацій та UI, забезпечуючи узгодженість візуального стилю.

Використання зазначених засобів забезпечило комплексний підхід до розробки гри, від створення графічних активів і анімацій до їх інтеграції, програмування логіки та оптимізація їх продуктивності.

3.2 Вимоги до технічного та програмного забезпечення

Розробка візуального стилю та анімаційного контенту для 2D-пригодницької гри потребувала відповідного технічного та програмного забезпечення, яке забезпечує стабільну роботу як на етапі створення графіки, так і під час тестування всіх компонентів у Середовищі розробки Unity. Вимоги сформовані з урахуванням особливостей ручного опрацювання графічних елементів, роботи з анімаційними параметрами Animator Controller, а також забезпечення сумісності з платформою Windows.

Фінальна версія гри оптимізована для запуску на пристроях із середніми та слабкими характеристиками. Мінімальні вимоги для користувача визначені на основі тестування й включають:

- процесор із двома ядрами та чотирма потоками з частотою 2.5 ГГц;
- оперативну пам’ять обсягом 4 ГБ DDR3;
- інтегровану відеокарту з 64 МБ відеопам’яті та підтримкою DirectX 11;
- 1 ГБ вільного простору на накопичувачі;
- клавіатуру, мишу та дисплей із роздільною здатністю не нижче 800×600.

Ці параметри забезпечать стабільний запуск створюваного інтерактивного розважального додатку.

Для повноцінної роботи з графічними редакторами, налаштування анімацій, компіляції та тестування проєкту використовувалося наступне апаратне забезпечення:

- процесор, що має шість ядер та дванадцять потоків (3.6 ГГц);
- 16 ГБ оперативної пам’яті DDR4 (3200 МГц);

- дискретна відеокарта з 12 ГБ відеопам’яті;
- SSD-накопичувач обсягом 1 ТБ.

Такі характеристики дозволяли одночасно працювати в середовищі Unity, обробляти графічні ресурси та виконувати збірку й запуск гри без затримок.

3.3 Опис програмної реалізації

Програмна реалізація розважального додатку включає в себе створення необхідних макетів та UI елементів за допомогою програмного пакету «Adobe Photoshop», дизайнів персонажів, анімацій пересування, елементів оточення та локацій за допомогою інструменту для цифрового малювання «Krita», а також імпортування та подальше налаштування всіх створених графічних активів у середовищі розробки «Unity». Після чого будуть реалізовані необхідні скрипти та компоненти, що зроблять графічний контент функціональним та готовим для взаємодії з гравцем.

3.3.1 Реалізація інтерфейсів користувача

Реалізація інтерфейсів користувача включає в себе всі етапи створення таких інтерфейсів як: «Головне меню», «Меню вибору рівня», «Меню налаштувань», «Меню паузи». Перед початком реалізації інтерфейсів користувача у середовищі розробки необхідно створити макети, які мають бути затверджені іншими учасниками розробки проєкту. Далі потрібно реалізувати необхідний графічний контент, який потім буде імпортуватися до середовища розробки та налаштовуватися відповідно до концептуальної моделі (див. додаток А).

3.3.1.1 Розробка макетів

Головне меню є основним інтерфейсом користувача, від дизайну якого залежав дизайн інших меню. На початку проєктування було створено декілька

макетів на вибір. Найбільшу кількість голосів набрав перший варіант, який сподобався всім учасникам команди (рис. В.1).

Створений макет стане основою, на яку можна буде спиратись при розробці дизайнів для інших інтерфейсів користувача, щоб забезпечити цілісність візуального стилю створюваного додатку.

Тепер, маючи конкретно визначений стиль дизайну інтерфейсу на основі макету головного меню можна перейти до реалізації інших меню. Наступним на черзі буде макет «Меню вибору рівня», який має містити в собі всі елементи визначенні на етапі концептуалізації (див. табл. 2.2) та відповідати графічному оформленню «Головного меню» (рис. В.2).

Всі елементи макети вже являються UI-елементами, які в подальшому будуть перенесені у середовище розробки Unity для налаштування функціоналу.

Створення «Меню налаштувань» потребує більших зусиль так як воно містить велику кількість інтерактивних елементів, які мають відповідати за конкретні функції налаштування майбутньої гри. Всі необхідні елементи налаштування були визначені на етапі концептуалізації (див. табл. 2.3). Макет буде створено відповідно до візуального стилю «Головного меню» та містити всі визначені інтерактивні елементи (рис. В.3).

Наступний макет буде виконано на фоні із суцільною заливкою, так як «Меню паузи» на відміну від інших інтерфейсів користувача буде з’являтися лише під час гри та не потребує фонового оформлення. Елементи для «меню паузи» було визначено на етапі концептуалізації (див. табл. 2.4). Макет має мінімалістичний стиль та містить лише необхідні кнопки для взаємодії (рис. В.4).

Всі створені макети мають єдиний стиль та відповідають концепції гри [17]. Подальша робота потребує розбиття створених макетів на окремі UI-елементи, які будуть імпортовані у середовище розробки «Unity».

3.3.1.2 Імпортування UI-елементів у середовищі розробки

Кожен створений макет був розбитий на окремі UI-елементи та імпортований у середовище розробки. Після чого у Unity було створено сцену «MainMenu» для реалізації інтерфейсів користувача на головному меню, до цих інтерфейсів входять: «Головне меню», «Меню вибору рівня», Меню налаштувань». Першим було реалізовано «Головне меню» (рис. Г.1).

Для створення ієрархії інтерфейсів користувача у сцені «MainMenu» було створено компонент Canvas, що буде містити в собі UI-елементи. Далі було створено три компоненти Panel, які являють собою контейнери для зберігання UI-елементів для кожного окремого меню та компонент Image для відображення фону головного меню.

В середині компоненту MainMenu (Panel) було додано п’ять елементів типу Button відповідно до концептуальної моделі (див. додаток А), елемент типу Image для відображення логотипу гри, а також елемент Panel для створення напівпрозорої підкладки, що візуально відділяє кнопки від фонового зображення.

Реалізація «Меню вибору рівня» потребувало створення у середині компоненту Play(Panel) восьми елементі типу Button відповідно до концептуальної моделі (див. додаток А) серед яких шість були призначені для створення кнопок рівнів, одна реалізації функції повернення на головний екран та одна для реалізації функції скидання прогресу гравця (рис. Г.2).

Останнім інтерфейсом головного меню є «Меню налаштувань». Його реалізація потребує використання великої кількості різноманітних UI-елементів, які були визначені на концептуальній моделі (див. додаток А). Елементи були поділені на блоки Volume та Screen для відділення налаштувань гучності від налаштувань зображення та більш зручного пошуку необхідних налаштувань гравцем (рис. Г.3).

«Меню паузи» має бути внутрішнім інтерфейсом, яким гравець зможе користуватись безпосередньо під час гри без необхідності виходити до головного меню. Воно має бути реалізовано на кожній сцені, які будуть

зберігати в собі ігрові рівні. Таким чином, для початку реалізації «Меню паузи» було створено нову сцену, яка має назву «Level1» та буде середовищем для реалізації першого рівня створюваної гри.

У сцені було створено елемент типу Panel під назвою PauseGameMenu(Panel). Цей контейнер зберігає в собі три елементи типу Button для реалізації функціоналу визначеного у концептуальній моделі (див. додаток А) та елемент типу Panel для створення напівпрозорої підкладки, що візуально відділяє кнопки від ігрового середовища (рис. Г.4).

Створенні інтерфейси, ще не мають в собі функціоналу та не відрізняються від звичайних макетів. Тому подальша робота полягає у прив’язці UI-елементів до скриптів, створених іншим учасником проєкту.

3.3.1.3 Реалізація функціоналу UI-елементів

Всього було створено три скрипти: MenuController, SettingsManager, PauseManager. Вони являють собою модульні менеджери до яких можна під’єднати компоненти безпосередньо в середовищі розробки Unity.

Скрипт MenuController реалізує логіку взаємодії з головним меню гри. Його основне призначення забезпечити зручну навігацію між основними розділами інтерфейсу: головним меню, меню вибору рівнів та меню налаштувань. У стартовому методі відбувається ініціалізація прогресу гравця та прив’язка дій до кнопок інтерфейсу таких як запуск прологу, скидання прогресу чи повернення до головного меню.

При натисканні клавіші Escape, скрипт дозволяє швидко повернутись до головного екрана, що зручно для користувача під час навігації між підменю. Також у скрипті передбачено перевірку та оновлення доступних рівнів залежно від того, які з них були вже відкриті гравцем. Це реалізовано за допомогою масиву кнопок рівнів, які активуються відповідно до даних про прогрес.

Скрипт SettingsManager відповідає за керування користувацькими налаштуваннями гри через інтерфейс. До сфери його відповідальності входять налаштування гучності (загальна, музика, ефекти, фонові звуки), роздільної

здатності екрана, повноекранного режиму та вертикальної синхронізації (V-Sync).

При першому запуску гри скрипт встановлює стандартні значення параметрів і зберігає їх у системі PlayerPrefs. Надалі ці значення використовуються як початкові під час кожного запуску гри. Через слайдери, випадаючі списки та перемикачі гравець може в реальному часі змінювати параметри, що стосуються аудіо. Зміни негайно відображаються завдяки використанню аудіомікшера, а графічні параметри застосовуються після збереження. Система відслідковує, чи були змінені якісь параметри, і відповідно активує або деактивує кнопку збереження.

Скрипт PauseMenu відповідає за реалізацію функціональності паузи під час проходження гри. Він дозволяє гравцеві у будь-який момент зупинити гру натисканням клавіші Escape. У разі активації паузи гра зупиняється – це досягається встановленням значення Time.timeScale рівним нулю, що блокує оновлення ігрової логіки. Водночас на екрані з’являється відповідна панель інтерфейсу з можливістю продовження гри, переходу до меню налаштувань або виходу в головне меню.

Якщо ж гравець вирішує повернутись до гри, скрипт знову активує ігровий час та приховує панель паузи. Вихід у головне меню передбачає автоматичне скидання значення часу, щоб уникнути помилок при наступному запуску сцени.

3.3.2 Реалізація анімаційного контенту

3.3.2.1 Створення дизайну та анімацій персонажа

Створення дизайну головного героя потребувало співпраці зі сценаристом та геймдизайнером. Процес створення дизайну головного героя відбувався у програмі для цифрового малювання Krita. Але на початку роботи ля зручного редагування та тестування анімацій був створений простий персонаж, з яким будуть відбуватися всі подальші маніпуляції (рис. Д.1).

Отримавши необхідний простий концепт персонажу, який не буде вимагати багато часу для подальших редагувань переходимо до створення

переліку необхідних кадрів, що будуть використання для реалізації анімацій у середовищі розробки Unity. Повний обсяг необхідних анімацій було визначено під час проєктування системи та представлено на рисунку Б.1 (рис. Д. 3).

Після проведення всіх необхідних тестувань та досягнення задовільного результату роботи скриптів та анімацій з тестовим персонажем, його буде замінено фінальним дизайном головного героя (рис. Д.2).

В ході подальшої розробки гри також будуть перероблені всі існуючі анімації, щоб замінити тестового персонажа на фінальний дизайн головної героїні. На даний момент було реалізовано дві анімації, які представлені на рисунку Д.4 та рисунку Д.5.

У середовищі розробки Unity кадри анімації використовуються для створення анімацій – файлів з розширенням «.anim». Цей процес виконується для створення всіх необхідних анімацій. Після чого можна переходити до реалізації логіки роботи зміни станів анімацій.

Також для наповнення локацій було створено дизайни для NPC, які будуть населяти всесвіт гри. Вони будуть провідниками для гравця, які будуть давати різні квести, створювати наративні ситуації та підтримувати сюжет. З усіма реалізованими дизайнами NPC можна ознайомитись у додатку Е.

3.3.2.2 Створення та налаштування Controller

Компонент Controller є основним етапом для реалізації логіки анімацій пересування у Unity. Створення анімаційної системи розпочалась з ініціалізації двох Animator Controller: PlayerAnimator для головного героя та NPCAnimator для неігрових персонажів [16]. Кожен контролер призначено відповідному ігровому об’єкту через компонент Animator, який додано до об’єктів Player та Enemy у сцені гри.

У вікні Animator для PlayerAnimator створено стани, що відповідають кожному анімаційному кліпу, при цьому стан PlayerIdle встановлено як початковий (Default State). Для NPCAnimator створено два стани: Enemy_Idle за замовчуванням та Enemy_Walk.

На основі таблиці 2.7 визначено та створено параметри для керування переходами між анімаційними станами (рис. Ж.1).

Після завершення етапу підготовки до якого входило створення файлів анімації, компоненту Controller та параметрів можна переходити до налаштування переходів між станами відповідно до концептуальної моделі зміни станів анімації (див. рис. Б.1). Для цього всі анімації переносяться у вікно контролеру, де для них вручну створюються зв'язки, які налаштовуються за допомогою параметрів (рис. Ж.2).

Подальша робота зі створенням функціональної системи пересування буде пов'язана з написанням програмного коду на основі напрацювань іншого учасника проєкту (див. додаток З).

3.3.2.3 Реалізація скриптів пересування та діалогової системи

У межах розробки гри було реалізовано візуальну частину механік керування головним героєм і поведінки ворогів. Основна мета полягала у забезпеченні візуальної відповідності між діями гравця та анімаційного відгуку на них, а також коректному відображені станів персонажа та ворогів у грі.

За логіку пересування ігрового персонажа по осі абсциса відповідає скрипт PlayerMovement.cs. У скрипті PlayerMovement.cs було реалізовано підтримку базової анімації для ругу персонажа, яка синхронізується з параметрами Animator. Зокрема, в методі Awake() відбувається ініціалізація компонента Animator, необхідного для керування візуальними станами:

```
private void Awake()
{
    // Отримуємо компоненти Rigidbody2D та Animator
    rb = GetComponent<Rigidbody2D>();
    animator = GetComponent<Animator>();
}
```

Під час кожного кадру, у методі Update(), до Animator передається абсолютне значення горизонтального вводу. Це значення використовується

для параметра HorizontalMove, який визначає, яка анімація повинна відтворюватись:

```
private void Update()
{
    if (!canMove)
    {
        // Якщо рух заблоковано, обнуляємо параметр анімації
        if (animator != null)
        {
            animator.SetFloat("HorizontalMove", 0f);
        }
        return;
    }

    // Зчитування горизонтального вводу
    moveInput = Input.GetAxis("Horizontal");

    // Передача значення в Animator
    if (animator != null)
    {
        animator.SetFloat("HorizontalMove", Mathf.Abs(moveInput));
    }
}
```

Додатково, у випадках, коли рух персонажа блокується (наприклад, під час діалогів, карабкання або іншої взаємодії), необхідно також зупинити відповідну анімацію. Для цього в методі SetCanMove() передбачено обнулення швидкості анімації, коли параметр canMove встановлюється у false:

```
public void SetCanMove(bool value)
{
    canMove = value;
    Debug.Log($"SetCanMove called with value: {value}, canMove is now:
{canMove}");

    if (!value)
    {
        rb.velocity = Vector2.zero;
        if (animator != null)
        {
            animator.SetFloat("HorizontalMove", 0f);
        }
    }
}
```

Усі ці зміни спрямовані на те, щоб візуальні стани персонажа узгоджувались з його логічним станом у грі.

Повний лістинг коду PlayerMovement.cs надано у додатку К. Для подальшої роботи та фіналізації скрипту, код було передано звукорежисеру (див. додаток З).

Механіка карабкання передбачає не лише зміну положення персонажа у просторі, але й відповідну візуальну підтримку цього процесу. У скрипті PlayerClimbing було реалізовано кілька важливих змін, що забезпечують правильну анімаційну поведінку персонажа під час взаємодії з об'єктами, по яких можна карабкатися.

Перш за все, при активації процесу карабкання, у методі StartClimbing() викликається анімаційний тригер Climb, який активує відповідну візуальну анімацію:

```
private void StartClimbing()
{
    // Блокуємо управління і фізику, активуємо анімацію
    isClimbing = true;
    playerMovement.SetCanMove(false);
    rb.isKinematic = true;
    rb.velocity = Vector2.zero;

    if (animator != null)
    {
        animator.SetTrigger("Climb");
    }
}
```

Ця анімація не просто візуальна – вона використовує механізм root motion, що дозволяє керувати переміщенням персонажа через саму анімацію, а не через фізичні розрахунки. Для цього реалізовано метод OnAnimatorMove(), у якому додається зсув до позиції гравця на основі animator.deltaPosition:

```
private void OnAnimatorMove()
{
    // Применяємо root motion для руху персонажа під час карабкання
    if (isClimbing && animator != null)
    {
        Vector3 delta = animator.deltaPosition;
        delta.z = 0f; // Ігноруємо зміщення по осі Z
        transform.position += delta;
    }
}
```

Після завершення анімації, яка зазвичай відтворюється один раз (наприклад, гравець перелазить через перешкоду), система має повернути гравцеві можливість рухатися. Цей перехід виконується в методі OnClimbAnimationEnd().

Повний лістинг коду PlayerClimbing.cs надано у додатку Л. Для подальшої роботи та фіналізації скрипту, код було передано звукорежисеру (див. додаток З).

Основною візуальною ознакою присідання є перемикання відповідного булевого параметра в Animator. У методі Crouch() при переході персонажа в стан сидіння активується параметр Crouch = true, що запускає відповідну анімацію:

```
public void Crouch()
{
    isCrouching = true;

    // Зменшуємо швидкість
    playerMovement.speed = crouchSpeed;

    // Змінюємо колайдер на "присідальний"
    if (playerCollider != null)
    {
        playerCollider.size = crouchColliderSize;
        playerCollider.offset = crouchColliderOffset;
    }

    // Активуємо анімацію
    if (animator != null)
    {
        animator.SetBool("Crouch", true);
    }

    Debug.Log("Персонаж ПРИСІВ (Crouch).");
}
```

Відповідно, у методі Uncrouch() параметр Crouch скидається (false), що означає повернення персонажа до стандартного візуального стану стояння:

```
public void Uncrouch()
{
    isCrouching = false;

    // Відновлюємо швидкість
    playerMovement.speed = originalSpeed;
```

```
    // Відновлюємо розміри колайдера
    if (playerCollider != null)
    {
        playerCollider.size = originalColliderSize;
        playerCollider.offset = originalColliderOffset;
    }

    // Вимикаємо анімацію присідання
    if (animator != null)
    {
        animator.SetBool("Crouch", false);
    }

    Debug.Log("Персонаж ВСТАВ (Uncrouch).");
}
```

Окрім перемикання анімаційного стану, важливо також забезпечити синхронізацію руху в режимі присідання з анімаційною швидкістю. Тому в методі Update() реалізовано динамічну передачу поточної горизонтальної швидкості (rb.velocity.x) до параметра HorizontalMove, коли персонаж перебуває в присіданні. Це дозволяє, наприклад, плавно показувати рух у присіді:

```
private void Update()
{
    // ...
    if (isCrouching && rb != null && animator != null)
    {
        float horizontalSpeed = Mathf.Abs(rb.velocity.x);
        animator.SetFloat("HorizontalMove", horizontalSpeed);
    }
}
```

Ці зміни дозволяють досягти максимальної візуальної відповідності між фізичним станом персонажа та його виглядом у грі. Присідання не просто зменшує колайдер або швидкість руху, а також відображається через відповідні анімаційні стани, що забезпечує візуальну виразність дій гравця.

Повний лістинг коду PlayerCrouch.cs надано у додатку М. Для подальшої роботи та фіналізації скрипту, код було передано звукорежисеру (див. додаток З).

Скрипт PlayerLadder відповідає за складну ігрову механіку лазіння по драбинах, включаючи як вертикальний рух гравця, так і спеціальні переходи

на верхню платформу або з неї. Для повноцінного сприйняття цієї взаємодії було важливо забезпечити відповідну візуальну анімаційну підтримку, яка відображає стани гравця в кожен момент лазіння.

У момент початку лазіння, тобто коли гравець наближається до драбини та натискає відповідну клавішу, активується метод StartClimbing(). У цьому методі встановлюється булевий параметр OnLadder = true, що запускає основну анімацію лазіння:

```
private void StartClimbing()
{
    if (isClimbing) return;

    if (playerCrouch != null && playerCrouch.IsCrouching)
    {
        playerCrouch.Uncrouch();
    }

    isClimbing = true;
    playerMovement.SetCanMove(false);
    rb.gravityScale = 0f;
    rb.velocity = Vector2.zero;

    if (playerCollider != null)
    {
        playerCollider.size = climbingColliderSize;
        playerCollider.offset = climbingColliderOffset;
    }

    if (currentLadderCollider != null)
    {
        float centerX = currentLadderCollider.bounds.center.x;
        transform.position = new Vector2(centerX, transform.position.y);
    }

    ForceFaceClimbDirection();

    if (animator != null)
    {
        animator.SetBool("OnLadder", true);
    }

    Debug.Log("StartClimbing: гравець почав лазити.");
}
```

Після завершення лазіння або виходу з драбини, у методі StopClimbing() параметр OnLadder скидається (false), що припиняє відтворення відповідної анімації:

```
public void StopClimbing()
```

```
{
    if (!isClimbing) return;

    isClimbing = false;
    rb.gravityScale = originalGravity;
    playerMovement.SetCanMove(true);

    if (playerCollider != null)
    {
        playerCollider.size = normalColliderSize;
        playerCollider.offset = normalColliderOffset;
    }

    if (animator != null)
    {
        animator.SetBool("OnLadder", false);
    }

    Debug.Log("StopClimbing: гравець припинив лазіння.");
}
```

Під час перебування на драбині важливо передавати швидкість вертикального вводу в параметр ClimbSpeed, що дозволяє синхронізувати інтенсивність анімації лазіння з реальним рухом. Це реалізовано у методі FixedUpdate():

```
private void FixedUpdate()
{
    if (isClimbing)
    {
        if (isFromTopSequence)
        {
            rb.velocity = Vector2.zero;
            if (animator != null)
            {
                animator.SetFloat("ClimbSpeed", 0f);
            }
        }
        else
        {
            float verticalInput = Input.GetAxis(verticalAxis);
            rb.velocity = new Vector2(0f, verticalInput * climbSpeed);

            if (animator != null)
            {
                animator.SetFloat("ClimbSpeed", Mathf.Abs(verticalInput));
            }
        }
    }
}
```

Окрему увагу було приділено двом анімаційним послідовностям, коли персонаж виходить на верхню платформу або спускається згори. У першому

випадку, активується метод StartOnTopSequence(), який запускає тригер OnTopClimb:

```
public void StartOnTopSequence(Vector2 topDestination)
{
    onTopFinalPosition = topDestination;
    isOnTopSequence = true;
    StopClimbing();
    playerMovement.SetCanMove(false);

    if (animator != null)
    {
        animator.SetTrigger("OnTopClimb");
    }

    Debug.Log("StartOnTopSequence: запуск анімації OnTopClimb.");
}
```

Аналогічно, для анімації спуску згори реалізовано метод StartFromTopSequence(), який запускає тригер FromTopClimb:

```
public void StartFromTopSequence(Vector2 ladderTopPosition)
{
    fromTopFinalPosition = ladderTopPosition;
    isFromTopSequence = true;

    if (animator != null)
    {
        animator.SetTrigger("FromTopClimb");
    }

    Debug.Log("StartFromTopSequence: запуск анімації FromTopClimb.");
}
```

Обидва ці тригери прив’язані до Animation Event, які викликають відповідні методи (OnTopAnimationFinished() та OnFromTopAnimationFinished()), що фіналізують положення гравця в просторі після завершення анімації.

У результаті впроваджених змін візуальне оформлення лазіння стало повноцінним і достовірним: гравець бачить логічно та візуально правильний перехід по драбині, включаючи початок, активне переміщення, вихід і завершення. Усі дії синхронізовані з фізикою, рухом та напрямком спрайта, що забезпечує цілісне сприйняття взаємодії.

Повний лістинг коду PlayerLadder.cs надано у додатку Н. Для подальшої роботи та фіналізації скрипту, код було передано звукорежисеру (див. додаток З).

Скрипт EnemyController відповідає за поведінку ворожого персонажа, зокрема патрулювання, реагування на гравця або шум. Для забезпечення візуальної виразності руху та орієнтації ворога мною було реалізовано кілька ключових змін, спрямованих на взаємодію зі складовою Animator, а також коректне відображення індикаторів над ворогом.

Насамперед, у методі Update() після виконання логіки переміщення або реагування на події здійснюється оновлення параметра HorizontalMove в Animator. Це значення задається відповідно до абсолютної величини змінної moveSpeed, яка оновлюється залежно від стану ворога. Це дозволяє анімаційному контролеру плавно перемикати ворога між станами "стоїть", "йде" або "біжить":

```
private void Update()
{
    moveSpeed = 0f;

    if (isGoingToLastPos || isWaitingAtLastPos)
    {
        HandleLostPlayerState();
    }
    else if (isRespondingToNoise || isWaitingAtNoise)
    {
        HandleNoiseResponse();
    }
    else if (isPlayerDetected && player != null)
    {
        HandleChasePlayer();
    }
    else
    {
        HandlePatrol();
    }

    // Оновлення параметра анімації руху
    if (animator != null)
    {
        animator.SetFloat("HorizontalMove", Mathf.Abs(moveSpeed));
    }
}
```

Окрім відтворення анімацій, над ворогом розміщено індикатор, який показує стан ворога, наприклад, попередження або виявлення гравця.

Важливо, щоб цей індикатор не розвертався разом зі спрайтом ворога, тому його положення й орієнтація постійно оновлюються в методі LateUpdate():

```
private void LateUpdate()
{
    if (detectionCanvas != null)
    {
        detectionCanvas.position = transform.position + canvasOffset;
        detectionCanvas.rotation = Quaternion.identity;

        Vector3 scale = detectionCanvas.localScale;
        scale.x = Mathf.Abs(scale.x); // Завжди позитивний масштаб по X
        detectionCanvas.localScale = scale;
    }
}
```

Щоб правильно візуально орієнтувати ворога при зміні напрямку руху, у функціях, де виконується переміщення масштаб ворога інвертується по осі X, якщо він рухається в протилежний бік. Це дозволяє зберігати відповідність між напрямком руху та візуальним виглядом ворога:

```
if (flipXForDirection && direction.x != 0)
{
    transform.localScale = new Vector3(
        Mathf.Sign(direction.x) * Mathf.Abs(transform.localScale.x),
        transform.localScale.y,
        transform.localScale.z
    );
}
```

Усі ці зміни спрямовані на підвищення візуальної достовірності поведінки ворога. Завдяки передачі швидкості в анімаційний контролер, інверсії масштабу при зміні напрямку та стабілізації індикатора над головою ворога створюється чітке, послідовне візуальне представлення стану NPC. Це дозволяє гравцеві інтуїтивно зчитувати наміри ворога та орієнтуватись у просторі гри не лише за логікою, але й за візуальним відображенням спрайтів.

Повний лістинг коду EnemyController.cs надано у додатку П. Для подальшої роботи та фіналізації скрипту, код було передано звукорежисеру (див. додаток З).

3.3.3 Реалізація ігрового простору

3.3.3.1 Створення дизайну та інтеграція локацій до Unity

Реалізація ігрового простору є одним з найважливіших етапів розробки 2D-гри, оскільки вона забезпечує створення середовища, яке підтримує наратив та взаємодію гравця з ігровим світом. Процес реалізації ігрового простору в середовищі розробки має схожий алгоритм дій із реалізацією анімаційного контенту. Робота на оточенням буде охоплювати створення першої ігрової локації.

Дизайн локацій був виконаний у програмі для цифрового малювання Krita, відповідно до наративних вимог, визначених спільно з геймдизайнером. Локації спроєктовано з акцентом на інтерактивність, що включає в себе створення та виділення на загальному плані об’єктів для взаємодії та зони для активації діалогів чи міні-ігор (рис. И.1).

Далі створений дизайн має бути розбитий на окремі спрайти та імпортований до середовища розробки Unity. У Unity об’єкти додаються до сцени «Level1», яка була створена в ході реалізації «Меню паузи» (див. рис. Г.3). Спрайти розміщено як елементи GameObject із компонентами Sprite Renderer, організовано в ієрархію для зручного керування. Після чого для сцени було налаштовано об’єкт Main Camera із прив’язкою до позиції гравця, що забезпечує коректне відображення локацій під час руху.

Для коректного відображення елементів було використано систему Sorting Layers, відповідно до проєктування ігрового простору (рис. И.2).

Параметр Order in Layer налаштовано для забезпечення правильного порядку рендерингу, наприклад, персонажі відображаються поверх фону, але позаду елементів переднього плану.

3.3.3.2 Реалізація ефекту паралаксу

Для створення візуального ефекту глибини під час переміщення камери у 2D-сцені була розроблена власна система паралаксу, що складається з трьох основних скриптів: ParallaxCamera, ParallaxBackground та ParallaxLayer [18]. Ця система дозволяє динамічно зсувати фонові шари з різною швидкістю

залежно від їхньої віддаленості, формуючи відчуття простору та перспективи під час гри.

Скрипт ParallaxCamera відповідає за визначення зміщення камери по горизонталі та повідомлення про це іншим об’єктам. У ньому використано делегат ParallaxCameraDelegate, який повідомляє всіх підписників про зміну положення камери. На старті гри зберігається поточна позиція камери:

```
void Start()
{
    oldPosition = transform.position.x;
}
```

У методі Update() перевіряється, чи змінилася позиція камери. Якщо так, обчислюється величина зміщення (delta), після чого викликається делегат onCameraTranslate, передаючи цю величину підписаним об’єктам:

```
void Update()
{
    if (transform.position.x != oldPosition)
    {
        if (onCameraTranslate != null)
        {
            float delta = oldPosition - transform.position.x;
            onCameraTranslate(delta);
        }

        oldPosition = transform.position.x;
    }
}
```

Цей скрипт є ключовим джерелом даних для всіх шарів, які беруть участь у паралаксі. Повний лістинг коду надано у додатку Р.

Скрипт ParallaxBackground прикріплюється до батьківського об’єкта, який містить усі фонові шари. Завданням скрипту є знаходження дочірніх об'єктів, що мають компонент ParallaxLayer, і передавати їм значення зміщення, отримане з ParallaxCamera. На старті відбувається автоматичне підключення до основної камери:

```
void Start()
{
    if (parallaxCamera == null)
        parallaxCamera = Camera.main.GetComponent<ParallaxCamera>();
    if (parallaxCamera != null)
        parallaxCamera.onCameraTranslate += Move;
    SetLayers();
}
```

Метод SetLayers() проходиться по всіх дочірніх об’єктах та додає лише ті, які мають компонент ParallaxLayer, до списку:

```
void SetLayers()
{
    parallaxLayers.Clear();
    for (int i = 0; i < transform.childCount; i++)
    {
        ParallaxLayer layer =
transform.GetChild(i).GetComponent<ParallaxLayer>();

        if (layer != null)
        {
            layer.name = "Layer-" + i;
            parallaxLayers.Add(layer);
        }
    }
}
```

При кожному зсуві камери викликається метод Move(float delta), який передає величину зсуву всім активним шарам:

```
void Move(float delta)
{
    foreach (ParallaxLayer layer in parallaxLayers)
    {
        layer.Move(delta);
    }
}
```

Повний лістинг даного скрипту надано у додатку С.

Скрипт ParallaxLayer містить логіку зсуву одного шару. Кожен шар має свій коефіцієнт паралаксу parallaxFactor, який визначає, наскільки повільно або швидко шар рухається відносно камери. Чим менше значення, тим шар "далі", і рухається повільніше. Основна логіка реалізована в методі Move(float delta), який обчислює нову позицію шару, зменшуючи її на величину delta, помножену на parallaxFactor:

```
public void Move(float delta)
{
    Vector3 newPos = transform.localPosition;
    newPos.x -= delta * parallaxFactor;

    transform.localPosition = newPos;
}
```

Цей метод забезпечує плавне переміщення кожного шару при русі камери, створюючи ілюзію глибини. Лістинг повного коду надано у додатку Т.

Уся система є незалежною від конкретної сцени, легко масштабована, підтримує роботу в режимі редагування та дозволяє розміщувати шари без жорсткого зв'язку з кодом. Завдяки цьому паралакс-ефект досягається без використання сторонніх пакетів або жорстко зашитих значень, повністю контрольований із Unity Editor. Така реалізація забезпечує гнучкість, читабельність і візуальну глибину в ігровому середовищі.

3.3.3.3 Реалізація скрипту GridGraphics для міні гри «RushHour»

У межах побудови користувацького інтерфейсу для гри було реалізовано власний компонент GridGraphic, який дозволяє візуалізувати гнучку сітку у UI-просторі. Цей скрипт успадковується від базового класу Graphic (простір імен UnityEngine.UI) і дозволяє відображати сітку з довільними параметрами.

У компоненті визначено кілька відкритих параметрів із відповідними атрибутами для зручного редагування через інспектор Unity:

```
[Header("Налаштування сітки")]
[Tooltip("Кількість комірок по горизонталі")]
public int gridWidth = 6;

[Tooltip("Кількість комірок по вертикалі")]
public int gridHeight = 6;

[Tooltip("Розмір комірки (в одиницях UI)")]
public float cellSize = 100f;

[Tooltip("Відстань між комірками (в пікселях)")]
public float spacing = 10f;

[Tooltip("Товщина ліній сітки (в пікселях)")]
public float lineThickness = 2f;
```

Ці параметри дозволяють дизайнеру або художнику оперативно налаштовувати вигляд сітки без редагування коду.

Ключовий метод, що відповідає за побудову візуального мешу це перевизначення вбудованого методу OnPopulateMesh(VertexHelper vh). Цей метод очищає поточну геометрію та додає нові лінії за допомогою прямокутників (quads), сформованих з 4-х вершин. Спочатку зчитується встановлений колір із властивості color базового класу Graphic:

```
Color lineColor = color;
```

Для врахування відступів між комірками обчислюється розмір ячейки:

```
float cellFullSize = cellSize + spacing;
```

Після цього відбувається побудова вертикальних ліній сітки:

```
for (int x = 0; x <= gridWidth; x++)
{
    Vector2 start = new Vector2(x * cellFullSize, 0);
    Vector2 end = new Vector2(x * cellFullSize, gridHeight * cellFullSize);
    AddLine(vh, start, end, lineColor);
}
```

А також горизонтальних:

```
for (int y = 0; y <= gridHeight; y++)
{
    Vector2 start = new Vector2(0, y * cellFullSize);
    Vector2 end = new Vector2(gridWidth * cellFullSize, y * cellFullSize);
    AddLine(vh, start, end, lineColor);
}
```

Метод AddLine() реалізує побудову однієї лінії у вигляді прямокутника з заданою товщиною. Спочатку визначається напрямок і нормаль до лінії:

```
Vector2 direction = (end - start).normalized;
Vector2 normal = new Vector2(-direction.y, direction.x);
Vector2 offset = normal * (lineThickness * 0.5f);
```

Потім розраховуються координати чотирьох кутів прямокутника:

```
Vector2 v1 = start - offset;
Vector2 v2 = start + offset;
Vector2 v3 = end + offset;
Vector2 v4 = end - offset;
```

І відповідно формуються вершини та два трикутники, що утворюють полосу:

```
int currentIndex = vh.currentVertCount;
UIVertex vertex = UIVertex.simpleVert;
vertex.color = lineColor;

vertex.position = v1;
vertex.uv0 = Vector2.zero;
vh.AddVert(vertex);

vertex.position = v2;
vertex.uv0 = Vector2.up;
vh.AddVert(vertex);

vertex.position = v3;
vertex.uv0 = Vector2.one;
vh.AddVert(vertex);

vertex.position = v4;
vertex.uv0 = Vector2.right;
vh.AddVert(vertex);

vh.AddTriangle(currentIndex, currentIndex + 1, currentIndex + 2);
vh.AddTriangle(currentIndex, currentIndex + 2, currentIndex + 3);
```

Скрипт GridGraphic.cs надає розробнику гнучкий засіб для відображення сітки без використання текстур. За рахунок повної побудови геометрії через VertexHelper, реалізовано можливість динамічного контролю над усіма аспектами сітки. Лістинг повного коду надано у додатку У.

3.3.3.4 Реалізація скрипту SwitchToggle для міні гри SwitchOnOff

Скрипт SwitchToggle реалізує поведінку візуального перемикача, який змінює стан при натисканні. Цей перемикач використовується як частина міні-гри або інтерфейсного елементу, де гравцю потрібно змінювати стани кількох об’єктів одночасно. Окрім зміни власного стану, компонент підтримує автоматичне перемикання пов’язаних елементів, а також виклик перевірки логіки перемоги.

Для зручності налаштування з інспектора в Unity передбачено кілька параметрів:

```
[Header("Налаштування стану")]
[Tooltip("Початковий стан перемикача: true - увімкнено, false - вимкнено")]
public bool isOn;

[Header("Спрайти для станів")]
public Sprite onSprite;
public Sprite offSprite;

[Header("Пов'язані перемикачі")]
[Tooltip("Список перемикачів, стан яких зміниться при натисканні на цей")]
public List<SwitchToggle> connectedToggles;
```

Параметри дозволяють встановити початковий стан, визначити візуальні спрайти для активного й неактивного станів, а також задати інші перемикачі, що мають бути змінені одночасно з цим.

У методі Awake() відбувається автоматичне отримання компонентів Button та Image, після чого до події натискання додається метод OnSwitchClicked(). Відразу ж викликається UpdateVisual() для синхронізації вигляду перемикача з поточним станом:

```
private void Awake()
{
    button = GetComponent<Button>();
    image = GetComponent<Image>();

    if (button != null)
        button.onClick.AddListener(OnSwitchClicked);

    UpdateVisual();
}
```

При натисканні на перемикач викликається метод OnSwitchClicked(), у якому реалізована наступна логіка:

1) перемикається власний стан за допомогою методу Toggle();
2) перемикаються всі пов'язані перемикачі зі списку connectedToggles;
3) викликається глобальна перевірка перемоги через менеджер SwitchPanelManager.

```
private void OnSwitchClicked()
{
    // Перемикаємо власний стан
    Toggle();

    // Перемикаємо пов'язані перемикачі
    foreach (SwitchToggle other in connectedToggles)
    {
        if (other != null)
            other.Toggle();
    }

    // Викликаємо перевірку перемоги
    SwitchPanelManager.Instance.CheckVictory();
}
```

Метод Toggle() інвертує логічне значення isOn і викликає UpdateVisual() для оновлення зовнішнього вигляду:

```
public void Toggle()
{
    isOn = !isOn;
    UpdateVisual();
}
```

У методі UpdateVisual() перевіряється наявність компонента Image, після чого змінюється спрайт на onSprite або offSprite залежно від поточного стану, що дозволяє гнучко візуально відображати стан перемикача.

```
private void UpdateVisual()
{
    if (image != null)
    {
        image.sprite = isOn ? onSprite : offSprite;
    }
}
```

Скрипт SwitchToggle реалізує зручну систему керованих перемикачів із візуальним супроводом та підтримкою логіки перемоги. Його гнучка архітектура дозволяє створювати міні-ігри, що базуються на принципі взаємозалежних натискань, подібно до механіки "світлових панелей", де кожне натискання змінює не лише поточний елемент, але й пов'язані з ним.

У поєднанні з менеджером (SwitchPanelManager) цей компонент дає змогу будувати складні логічні структури та перевірки, зберігаючи просту й зрозумілу внутрішню реалізацію.

Лістинг повного коду надано у додатку Ф.

3.4 Тестування

Після завершення створення графічних елементів і реалізації анімаційного контенту було проведено тестування, яке мало на меті перевірити коректність функціонування візуальної частини гри та її інтеграцію з ігровими механіками. Враховуючи, що гра розроблялася невеликою командою, тестування стало важливою складовою процесу перевірки узгодженості між художніми компонентами й програмною логікою.

У межах тестування було приділено увагу головному меню гри, яке є першою точкою взаємодії користувача з продуктом[19]. Перевірялося коректне відображення всіх кнопок, а також перевірялися переходи між екранами, що викликаються цими кнопками. Важливим аспектом стало підтвердження стабільності поведінки меню при різних роздільних здатностях та під час повернення до головного меню з ігрової сцени. В результаті перевірок було визначено, що створене меню функціонує коректно.

Наступним етапом тестування стала перевірка правильності відображення спрайтів персонажів у різних станах анімації. Усі зображення перевірялись безпосередньо в ігровому середовищі за допомогою попереднього перегляду в редакторі Unity. У випадках, коли було виявлено невідповідність положення спрайтів ці недоліки одразу виправлялися шляхом уточнення координат або редизайну конкретного кадру у програмі Krita.

Окрему увагу було приділено анімаційним станам, які пов’язані з логікою гри. У кожному скрипті, де використовувався Animator, тестувалася коректність зміни параметрів. У процесі перевірки здійснювався запуск відповідних дій гравцем, наприклад, присідання або початок лазіння, після

чого відслідковувалося, чи змінюються анімаційні стани відповідно до очікуваної логіки. Усі ці параметри були заздалегідь налаштовані в Animator Controller, і тестування підтвердило, що виклики з коду працюють коректно.

Також було перевірено роботу компонентів, що відповідають за взаємодію користувача з інтерфейсом. Зокрема, у рамках тестування сітки GridGraphic перевірялося, як вона масштабуються залежно від параметрів у редакторі, чи коректно розраховуються розміри комірок та ліній і чи немає візуальних артефактів при зміні роздільної здатності. Всі перевірки проходили в межах редактора UI, а також у режимі Play. Аналогічним чином тестувалася система перемикачів SwitchToggle, що використовувалася у міні-грі. Кожен перемикач мав змінювати свій стан при натисканні, а також змінювати стани пов’язаних з ним елементів. У результаті вдалося досягти повної узгодженості між візуальною реакцією перемикача та логікою перевірки виграшу.

Додатково, у процесі тестування перевірялась система фону з ефектом паралаксу. Камера рухалась по сцені, а задні шари зсувались із різною швидкістю, що забезпечувало візуальну глибину. Було підтверджено, що всі ParallaxLayer коректно реагують на зміну позиції камери, а плавність руху фону зберігається незалежно від продуктивності сцени.

Проведене тестування дозволило впевнено оцінити якість реалізованого візуального контенту, а також його стабільність у рамках гри. Виявлені незначні недоліки були виправлені на місці, а фінальний результат підтвердив відповідність створених елементів технічному завданню та загальній художній концепції проєкту.

Висновки до розділу

У межах третього розділу було реалізовано візуальний стиль гри, включаючи дизайн персонажів, середовища та інтерфейсу. Визначена художня концепція з приглушеною палітрою й реалістичними пропорціями забезпечила стилістичну цілісність проєкту.

Розроблено повний набір анімацій для персонажів і ворогів, інтегрованих із логікою гри через параметри Animator. Також створено графічні компоненти інтерфейсу: сітку, перемикачі та фоновий ефект паралаксу.

У результаті тестування підтверджено коректну роботу візуального контенту, його узгодженість із ігровими механіками та стабільність у всіх ключових сценаріях. Реалізовані рішення відповідають поставленим завданням і забезпечують візуальну основу гри.

РОЗДІЛ 4
ОХОРОНА ПРАЦІ

4.1 Регулювання питань охорони праці на законодавчому рівні

Охорона праці в Україні регулюється системою законодавчих та нормативно-правових актів, що спрямовані на створення безпечних і нешкідливих умов праці для всіх категорій працівників. Головним нормативним документом, що визначає правові, організаційні та соціально-економічні засади охорони праці, є Закон України «Про охорону праці» №2694-XII [20]. Він закріплює право працівника на безпечні умови праці, належне санітарно-гігієнічне середовище, забезпечення засобами індивідуального захисту та обов’язкове проходження інструктажів, навчання з питань охорони праці. Також закон визначає відповідальність роботодавця за створення належних умов праці та забезпечення функціонування системи управління охороною праці на підприємстві.

Окрім цього закону, важливими нормативними актами є Кодекс законів про працю України (КЗпП) [21], який регулює трудові відносини та містить окремі положення щодо охорони праці, Типове положення про службу охорони праці (НПАОП 0.00-4.21-04) [22], яке визначає структуру та завдання цієї служби на підприємствах, а також ДСТУ ISO 45001:2019, що встановлює вимоги до системи управління охороною здоров’я і безпекою праці з урахуванням міжнародного досвіду [23]. У контексті інтеграції європейських норм також варто згадати Директиву Ради Європейських Співтовариств 89/391/ЄЕС [24], яка зобов’язує роботодавців вживати заходів щодо запобігання професійним ризикам, навчання працівників і врахування безпеки вже на стадії проєктування робочих процесів.

Реалізація вимог законодавства з охорони праці передбачає участь різних суб’єктів. Центральну роль відіграє роботодавець, який відповідальний за організацію безпечного виробничого процесу, розробку інструкцій,

навчання працівників, забезпечення засобами захисту та впровадження профілактичних заходів. Фахівець з охорони праці (за його наявності) контролює дотримання вимог охорони праці, проводить інструктажі, бере участь у розслідуванні нещасних випадків і аналізує професійні ризики. Працівник зобов'язаний дотримуватися встановлених вимог безпеки, виконувати інструкції, не створювати загроз для себе та інших. Державний нагляд за дотриманням законодавства здійснює Державна служба України з питань праці (Держпраці), яка має повноваження щодо перевірки підприємств, накладення санкцій і надання приписів.

У сфері інформаційних технологій, де основні робочі процеси пов'язані з використанням комп'ютерної техніки, актуальними є нормативні акти, що регламентують ергономіку робочого місця, рівень освітленості, мікроклімат, рівні шуму та електромагнітного випромінювання. Зокрема, застосовуються такі документи, як ДСН 3.3.6.042-99 «Санітарні норми мікроклімату виробничих приміщень» [25], ДСанПіН 3.3.6.096-2002 «Державні санітарні норми і правила при роботі з джерелами електромагнітних полів» [26], ДБН В.2.5-28:2018 «Природне і штучне освітлення» [27] тощо. Дотримання вимог цих норм сприяє не лише підвищенню ефективності праці, а й збереженню здоров'я працівників у довготривалій перспективі.

4.2 Виявлення потенційних небезпек стосовно об'єкту проєктування

У процесі виконання робіт, пов'язаних із розробкою програмного забезпечення, працівники ІТ-сфери переважно здійснюють діяльність в умовах офісного середовища або в домашніх робочих просторах, обладнаних для виконання професійних завдань. Незважаючи на відсутність складних виробничих процесів, на таких робочих місцях можуть бути присутні як шкідливі, так і небезпечні фактори, що становлять загрозу для здоров'я, продуктивності або навіть життя працівників [28].

Фізичні небезпечні та шкідливі фактори на робочому місці програміста охоплюють насамперед вплив мікрокліматичних умов. Зокрема, йдеться про недостатню вентиляцію або кондиціювання повітря, відсутність регуляції температури в приміщенні, що може призвести до перегрівання або переохолодження працівника [29]. Вплив має також освітлення – при неправильному або недостатньому освітленні погіршується зір, зростає втомлюваність, виникають головні болі. Сюди ж слід віднести електромагнітні випромінювання, які генеруються комп'ютерами, моніторами, роутерами та іншими пристроями, особливо при тривалому перебуванні біля джерел випромінювання без належного екранування. До шкідливих факторів також належить шумове навантаження у приміщеннях із великою кількістю працюючої техніки або відкритим плануванням, що ускладнює концентрацію уваги.

Психофізіологічні фактори також мають суттєве значення. До них належать тривале сидіння в статичній позі за комп'ютером, яке викликає надмірне статичне навантаження на хребет і м'язи шиї, спини, зап'ястків. Частим наслідком цього є розвиток захворювань опорно-рухового апарату (остеохондроз, тунельний синдром тощо). Високе навантаження на зорову систему спричиняє зорову втому, сухість очей, подразнення. Крім того, IT-фахівці перебувають під впливом нервово-психічних перевантажень: це емоційне вигорання, монотонність праці, інформаційне перенасичення, жорсткі дедлайни, тривале інтелектуальне напруження, робота в умовах стресу [30]. До загальних небезпек, характерних для офісних умов, належать:

– ризик ураження електричним струмом внаслідок використання пошкодженого або неякісно заземленого обладнання;

– пожежна небезпека, пов'язана з надмірною кількістю електроніки та недотриманням правил пожежної безпеки;

– ризик травматизму внаслідок розміщення кабелів на підлозі, нестійких меблів, слизького покриття тощо.

Окремо слід виділити військову загрозу як чинник, що набув актуальності в сучасних умовах в Україні. У випадках повітряної тривоги, ракетних ударів або інших надзвичайних ситуацій працівники повинні мати доступ до укриттів, а робочі приміщення – бути облаштованими відповідно до вимог цивільного захисту.

Таким чином, серед основних груп небезпечних і шкідливих факторів, що можуть бути присутніми на робочому місці розробника, можна виділити:

– Фізичні: нераціональне освітлення, недотримання параметрів мікроклімату, шум, електромагнітні випромінювання, електрична небезпека.

– Психофізіологічні: статичні навантаження, розумове перенапруження, стрес, емоційне вигорання.

– Хімічні (опосередковано): забруднення повітря в замкнених приміщеннях при недостатній вентиляції.

– Загальнонебезпечні: пожежна загроза, військова загроза, ураження струмом, побутовий травматизм.

Більшість виявлених факторів можуть бути частково або повністю усунуті шляхом впровадження організаційних, технічних та профілактичних заходів, які будуть розглянуті у наступному підрозділі.

4.3 Дослідження ризику реалізації небезпек та рекомендації

Оцінювання ризиків є важливою складовою системи управління охороною праці та передбачає визначення ймовірності виникнення небезпечних ситуацій, а також рівня їх потенційних наслідків. Основною метою оцінювання ризиків є своєчасне виявлення загроз для життя та здоров’я працівників, а також розробка і впровадження ефективних профілактичних заходів для їх запобігання або мінімізації.

Процедура оцінювання ризиків включає кілька етапів: ідентифікацію небезпек, аналіз умов, у яких ці небезпеки можуть реалізуватись, оцінювання ймовірності їх виникнення, визначення тяжкості можливих наслідків,

класифікацію ризиків за рівнем небезпеки та ухвалення управлінських рішень щодо їх усунення або контролю. Задачами цієї процедури є формування безпечного середовища праці, зниження рівня професійної захворюваності, запобігання аваріям, нещасним випадкам та зменшення витрат, пов'язаних із ліквідацією їхніх наслідків.

Для практичного аналізу ризику реалізації потенційних небезпек доцільно скористатись методом «дерева небезпеки», який передбачає побудову логічної схеми, що демонструє причини виникнення певної небажаної події.

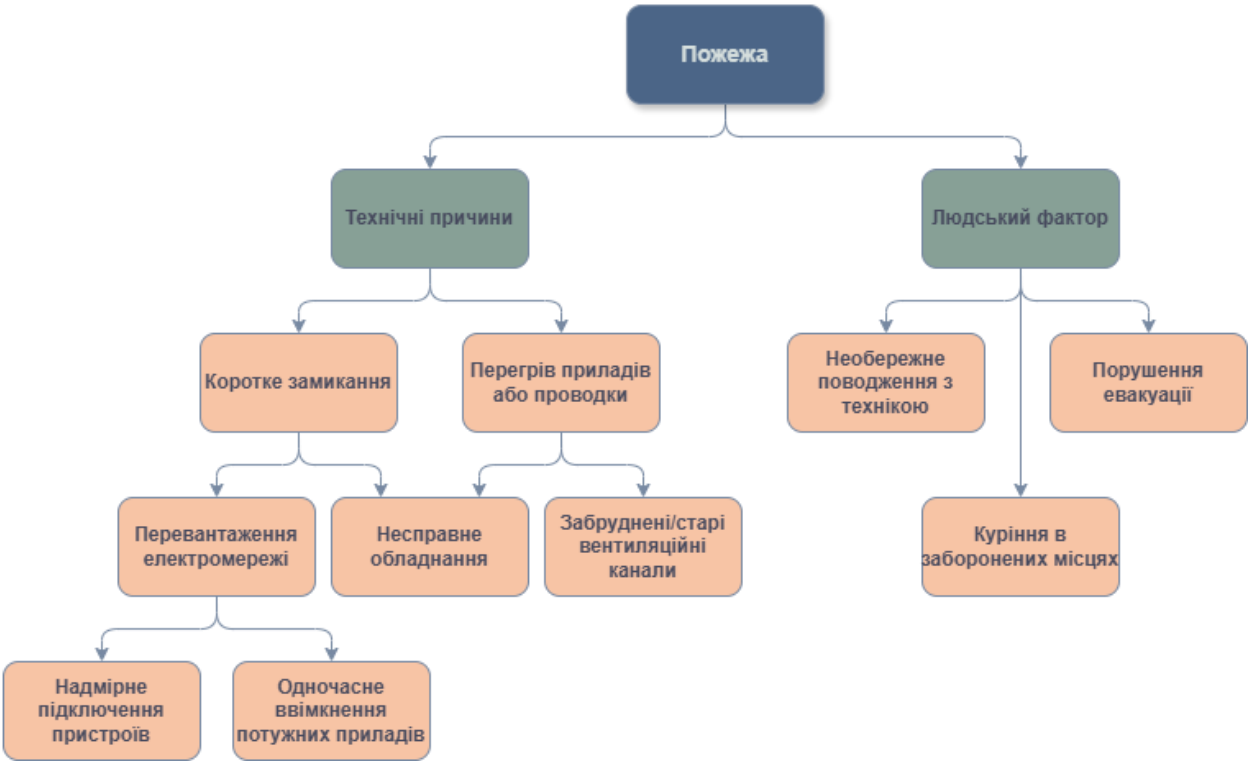


Рисунок 4.1 – Дерево небезпек для небезпеки «Пожежа в офісному приміщені»

У цій логічній схемі подія «Пожежа» є кінцевою. Вона може бути спричинена як технічними, так і організаційними чинниками. Технічні причини – це коротке замикання через перевантаження або пошкодження мережі. Людський фактор включає необережне поводження з

електроприладами, наприклад, залишення зарядних пристроїв у розетках, використання несправної техніки або порушення заборони на куріння.

Для зниження ризиків, пов’язаних із виявленими небезпеками, доцільно впровадити такі заходи:

1) Організаційні заходи:

– розробити та впровадити інструкції з охорони праці для працівників офісу;

– проводити регулярні інструктажі з пожежної безпеки та цивільного захисту;

– запровадити контроль за виконанням правил використання електроприладів.

2) Технічні заходи:

– забезпечити офісне приміщення справною електромережею з пристроями захисного відключення (УЗО);

– обладнати офіс засобами пожежогасіння (вогнегасники, пожежні сповіщувачі);

– здійснювати регулярне технічне обслуговування електроприладів та систем вентиляції;

– впровадити автоматичну систему сигналізації та оповіщення про надзвичайні ситуації.

3) Ергономічні та санітарно-гігієнічні заходи:

– організувати робочі місця відповідно до вимог освітлення, температури, вологості, шумового навантаження;

– забезпечити наявність ергономічних меблів, моніторів із налаштуванням яскравості та контрасту;

– впровадити регламентовані перерви для запобігання перенавантаженням та зоровій втомі.

4) Засоби цивільного захисту:

– позначити шляхи евакуації, встановити аварійне освітлення;

– передбачити укриття для персоналу в умовах воєнної загрози згідно з рекомендаціями органів цивільного захисту.

Висновки до розділу

У розділі «Охорона праці» було розглянуто ключові аспекти безпечної організації умов праці на об’єкті проєктування, яким у межах цієї роботи виступає офісне середовище для розробників програмного забезпечення. На законодавчому рівні визначено, що охорона праці є обов’язковим елементом діяльності будь-якого суб’єкта господарювання та передбачає дотримання відповідних норм, таких як Закон України «Про охорону праці», КЗпП, державні санітарні норми, технічні регламенти та міжнародні стандарти.

Проведений аналіз потенційних небезпек дозволив ідентифікувати як специфічні для ІТ-сфери фактори (наприклад, зорове й психоемоційне навантаження, статична поза, електромагнітне випромінювання), так і загальні загрози (електротравматизм, пожежі, військові дії). Застосування методу аналізу «дерева відмов» до небезпеки виникнення пожежі допомогло наочно представити структуру причин і підпричин, які можуть призвести до надзвичайної ситуації.

У результаті оцінки було сформовано перелік конкретних організаційних і технічних заходів, спрямованих на зменшення ризику та підвищення безпеки працівників. Це включає облаштування робочого місця відповідно до ергономічних вимог, інструктажі, наявність засобів пожежогасіння та системи оповіщення, а також дотримання норм мікроклімату й освітлення.

Таким чином, впровадження комплексного підходу до питань охорони праці забезпечує зниження рівня професійних ризиків, збереження здоров’я персоналу, підвищення продуктивності праці та відповідність вимогам чинного законодавства.

ЗАГАЛЬНІ ВИСНОВКИ

У ході виконання дипломної роботи на тему «Розробка візуального стилю та анімаційного контенту для 2D-гри на базі Unity» було досягнуто основної мети – створення завершеного візуального оформлення та відповідного анімаційного супроводу, що відповідає жанровим особливостям пригодницького квесту з елементами інтерактивної драми.

Було проведено аналіз предметного середовища, досліджено сучасні підходи до побудови графічного контенту та вивчено приклади успішних аналогів, що дозволило визначити актуальний стиль і засоби реалізації візуальної складової гри. У процесі реалізації проєкту створено унікальні графічні ресурси, які охоплюють як стилізацію персонажів і локацій, так і елементи інтерфейсу користувача, головного меню та міні-ігор.

Усі графічні елементи були інтегровані у середовище розробки Unity із врахуванням вимог до продуктивності та оптимізації. Була реалізована система зміни станів анімацій із використанням Animator Controller, налагоджено взаємодію між анімаційною системою та програмною логікою гри, а також створено модульну систему паралакс-руху фону, яка підсилює глибину ігрового простору.

У межах командної співпраці забезпечено взаємодію з геймдизайнером та програмістом, що дозволило ефективно адаптувати створений графічний контент до наративної структури гри та її технічних обмежень. Отримані результати засвідчили доцільність обраного підходу та технологій, продемонструвавши можливість реалізації якісного візуального наповнення навіть у межах інді-розробки.

Виконана робота не лише дозволила створити завершену графічну частину ігрового проєкту, але й сприяла набуттю важливих практичних навичок у сфері інтеграції, оптимізації та адаптації художнього контенту для інтерактивних програмних продуктів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Жерновий М.О., Баталов С.Д., Братерська Н. М. Кіберспорт у вищих навчальних закладах: розвиток та можливості // Комп'ютерні ігри та мультимедіа як інноваційний підхід до комунікації - 2023 / Матеріали III Всеукраїнської науково-технічної конферен-ції молодих вчених, аспірантів і студентів, Одеса, 28-29 жовтня 2023 р. - Одеса, Видавництво ОНТУ, 2023 р. – с.47-49 – URL: https://ontu.edu.ua/download/konfi/2023/Abstracts-Computer-games-and-multimedia-as-an-innovative-approach-to-electronic-communication-23.pdf. (дата звернення: 13.05.2025).

2. Unity Technologies. 2D Game Development in Unity // Unity Documentation. – 2023. – URL: https://docs.unity3d.com/Manual/Unity2D.html (дата звернення: 13.05.2025).

3. The Best 2D Games On Steam // Thegames – 2024. – URL: https://www.thegamer.com/2d-video-games-best-steam-according-metacritic/ (дата звернення: 13.05.2025).

4. What is 2D Art? Discovering Styles, Types, and Quality Standards // Argentics – 2024 – URL: https://www.argentics.io/what-is-2d-game-art (дата звернення: 13.05.2025).

5. Vector vs Raster: When to use which in Game UX/UI Design // LinkedIn – 2024 – URL: https://www.linkedin.com/pulse/vector-vs-raster-when-use-which-game-uxui-design-punchevgroup-ikajf/ (дата звернення: 13.05.2025).

6. Indie Game Art Styles // Inlingogames – 2025 – URL: https://inlingogames.com/blog/indie-game-art-styles/ (дата звернення: 13.05.2025).

7. Why Pixel Art Games Have Become Widely Popular // Rocketbrush – 2021 – URL: https://rocketbrush.com/blog/pixel-art-games-popular (дата звернення: 13.05.2025).

8. Valiant Hearts: The Great War // Steam Store – 2025. – URL: https://store.steampowered.com/app/260230/Valiant_Hearts_The_Great_War__Soldats_Inconnus__Mmoires_de_la_Grande_Guerre/ (дата звернення: 17.05.2025).

9. Deponia // Steam Store – 2025 – URL: https://store.steampowered.com/app/214340/Deponia/ (дата звернення: 17.05.2025).

10. When The Past Was Around // Steam Store. – 2023. – URL: https://store.steampowered.com/app/1164050 (дата звернення: 17.05.2025).

11. Unity Technologies. Unity 2D Game Development Manual – 2023 – URL: https://docs.unity3d.com/Manual/Unity2D.html (дата звернення: 24.05.2025)

12. Microsoft. Visual Studio 2022 Community Edition Documentation – 2023 – URL: https://learn.microsoft.com/en-us/visualstudio/releases/2022 (дата звернення: 24.05.2025)

13. Krita Foundation. Krita User Manual – 2024 – URL: https://docs.krita.org/en/index.html (дата звернення: 24.05.2025)

14. Adobe Inc. Adobe Photoshop User Guide – 2024 – URL: https://helpx.adobe.com/photoshop/user-guide.html (дата звернення: 24.05.2025)

15. Idyllic Pixel. PureRef Overview and Shortcuts – 2023 – URL: https://www.pureref.com/ (дата звернення: 24.05.2025)

16. Unity Technologies. Animator Controller – 2023 – URL: https://docs.unity3d.com/Manual/class-AnimatorController.html (дата звернення: 24.05.2025)

17. Unity Technologies. UI Toolkit and UI Elements in Unity – 2023 – URL: https://docs.unity3d.com/Manual/UIElements.html (дата звернення: 24.05.2025)

18. Паралакс – 2024 – URL: https://uk.wikipedia.org/wiki/Паралакс (дата звернення: 24.05.2025)

19. Unity Technologies. Testing and Debugging in Unity – 2023 – URL: https://docs.unity3d.com/6000.0/Documentation/Manual/WindowsDebugging.html (дата звернення: 24.05.2025)

20. Закон України «Про охорону праці» №2694-XII – 1992 – URL: https://zakon.rada.gov.ua/laws/show/2694-12#Text (дата звернення: 28.05.2025)

21. Кодекс законів про працю України (КЗпП) – 1971 – URL: https://zakon.rada.gov.ua/laws/show/322-08#Text (дата звернення: 28.05.2025)

22. НПАОП 0.00-4.21-04 «Типове положення про службу охорони праці» – 2004 – URL: https://zakon.rada.gov.ua/laws/show/z1526-04#Text (дата звернення: 28.05.2025)

23. ДСТУ ISO 45001:2019 «Системи управління охороною здоров'я та безпекою праці» – 2019 – URL: https://online.budstandart.com/ua/catalog/doc-page.html?id_doc=88004 (дата звернення: 28.05.2025)

24. Директива Ради ЄС 89/391/ЄЕС – 1989 – URL: https://zakon.rada.gov.ua/laws/show/994_b23#Text (дата звернення: 28.05.2025)

25. ДСН 3.3.6.042-99 «Санітарні норми мікроклімату виробничих приміщень» – 1999 – URL: https://online.budstandart.com/ua/catalog/doc-page.html?id_doc=14283 (дата звернення: 28.05.2025)

26. ДСанПіН 3.3.6.096-2002 «Робота з джерелами електромагнітних полів» – 2002 – URL: https://zakon.rada.gov.ua/laws/show/z0203-03#Text (дата звернення: 28.05.2025)

27. ДБН В.2.5-28:2018 «Природне і штучне освітлення» – 2018 – URL: https://online.budstandart.com/ua/catalog/doc-page.html?id_doc=79885 (дата звернення: 28.05.2025)

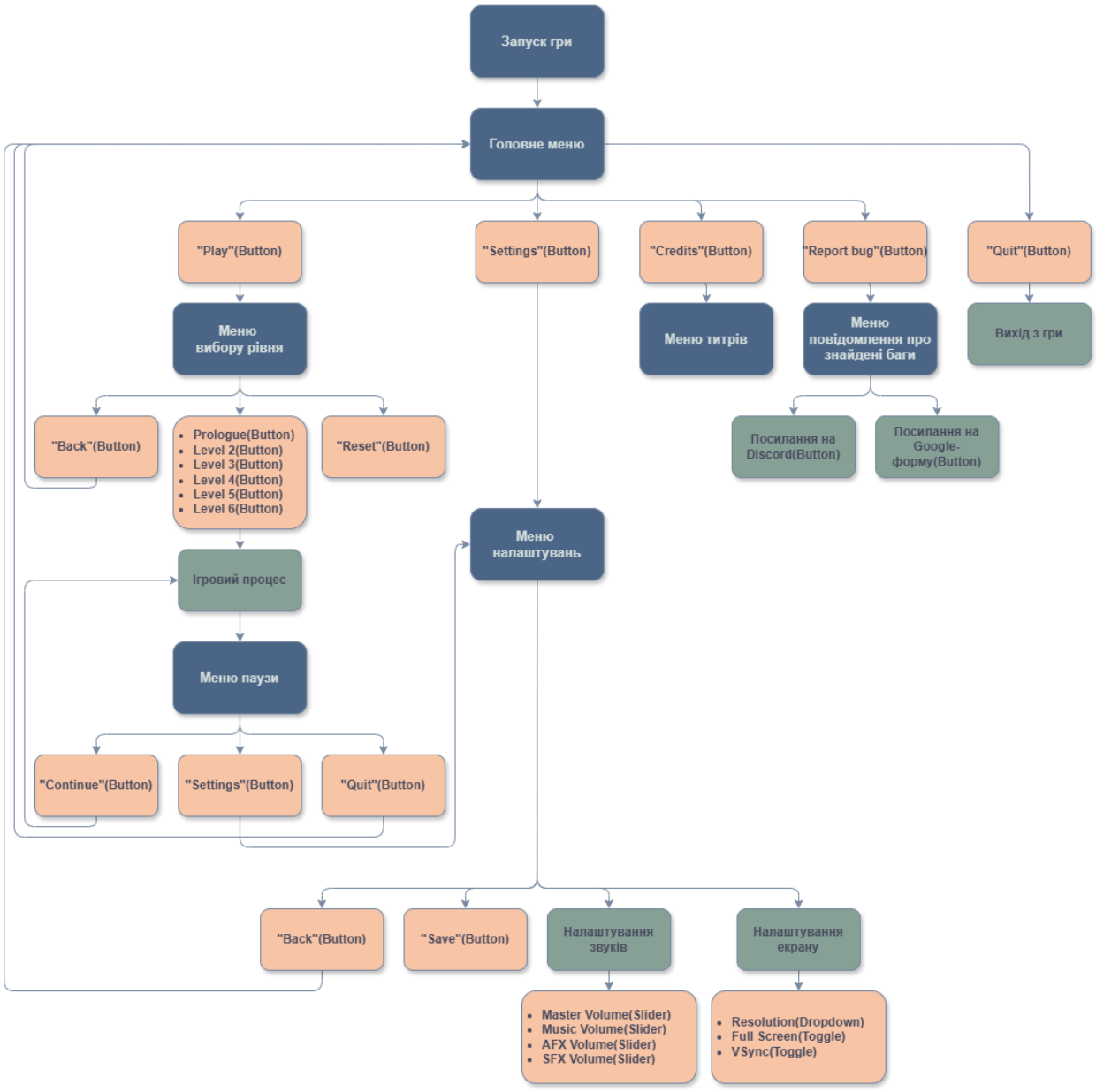
28. Класифікація шкідливих та небезпечних факторів виробничого середовища – 2021 – URL: https://oppb.com.ua/articles/klasyfikaciya-nebezpechnyh-i-shkidlyvyh-vyrobnychyh-faktoriv (дата звернення: 28.05.2025)

29. Робота в офісі: основні санітарно-гігієнічні вимоги – 2018 – URL: https://te.dsp.gov.ua/robota-v-ofisi-osnovni-sanitarno-gigiyenichni-vymogy/ (дата звернення: 28.05.2025)

30. Як знизити психосоціальні ризики на робочому місці – 2024 – URL: https://phc.org.ua/news/yak-zniziti-psikhosocialni-riziki-na-robochomu-misci-0 (дата звернення: 28.05.2025)

ДОДАТОК А

UML-діаграма інтерфейсів користувача



ДОДАТОК Б

Діаграми до проєктування системи

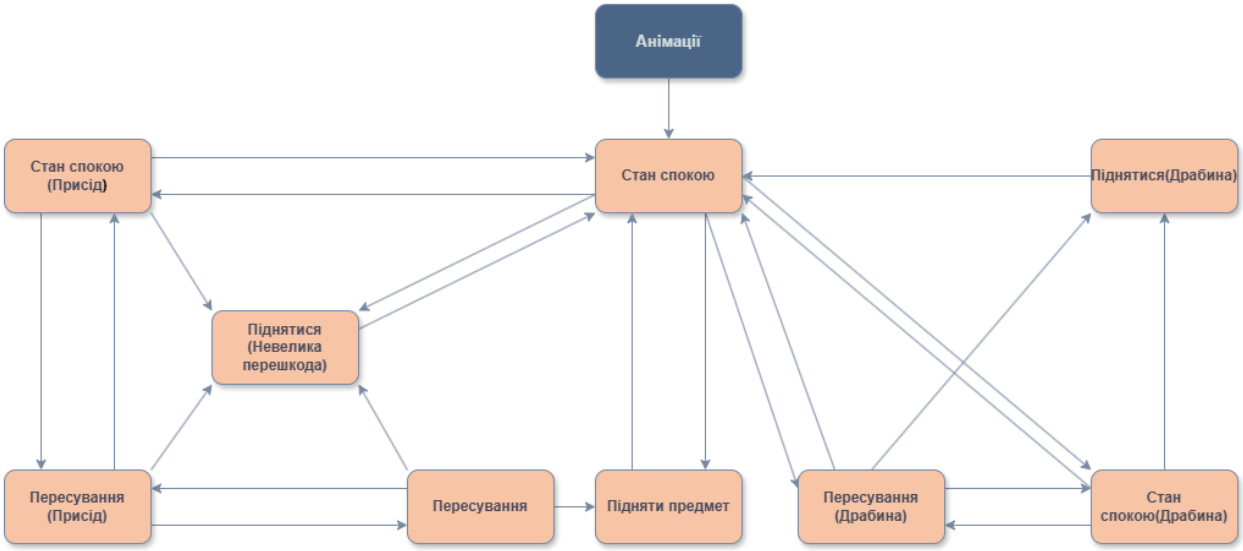


Рисунок Б.1 – UML-діаграма зміни станів анімацій для головного героя

Рисунок Б.2 – UML-діаграма зміни станів анімацій для противників

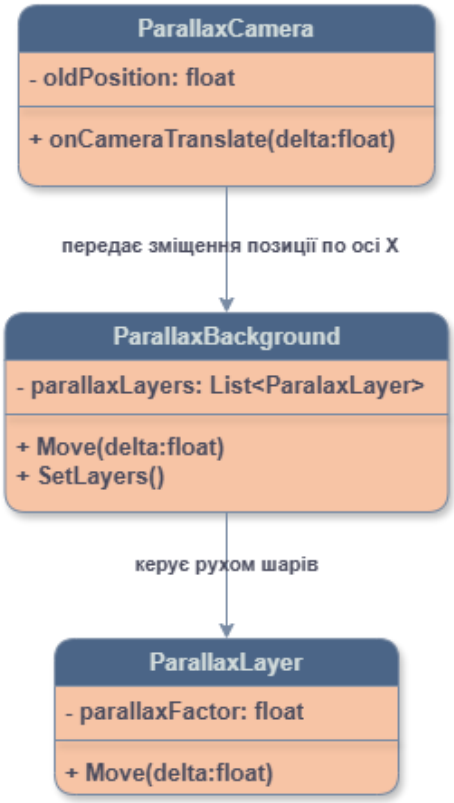


Рисунок Б.3 – Діаграма класів системи паралакс-ефекту

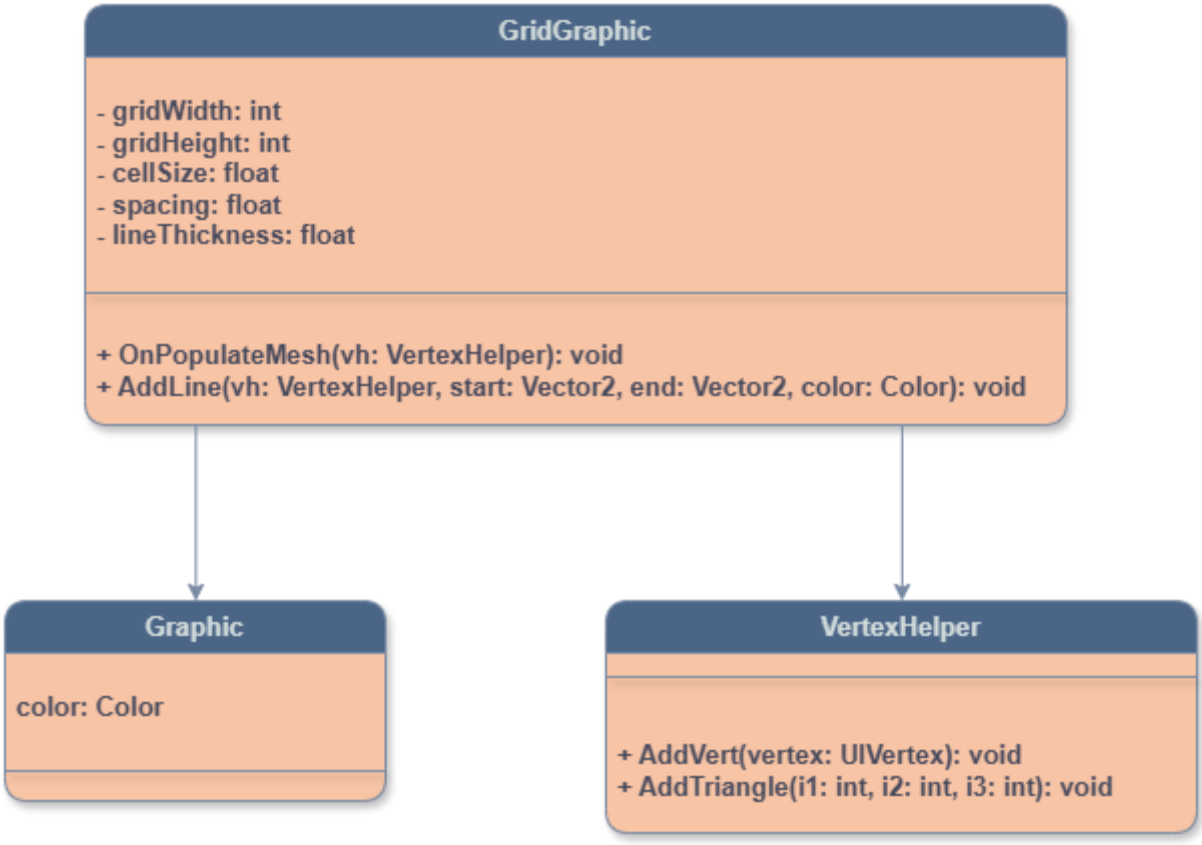


Рисунок Б.4 – Діаграма класів компонента GridGraphic

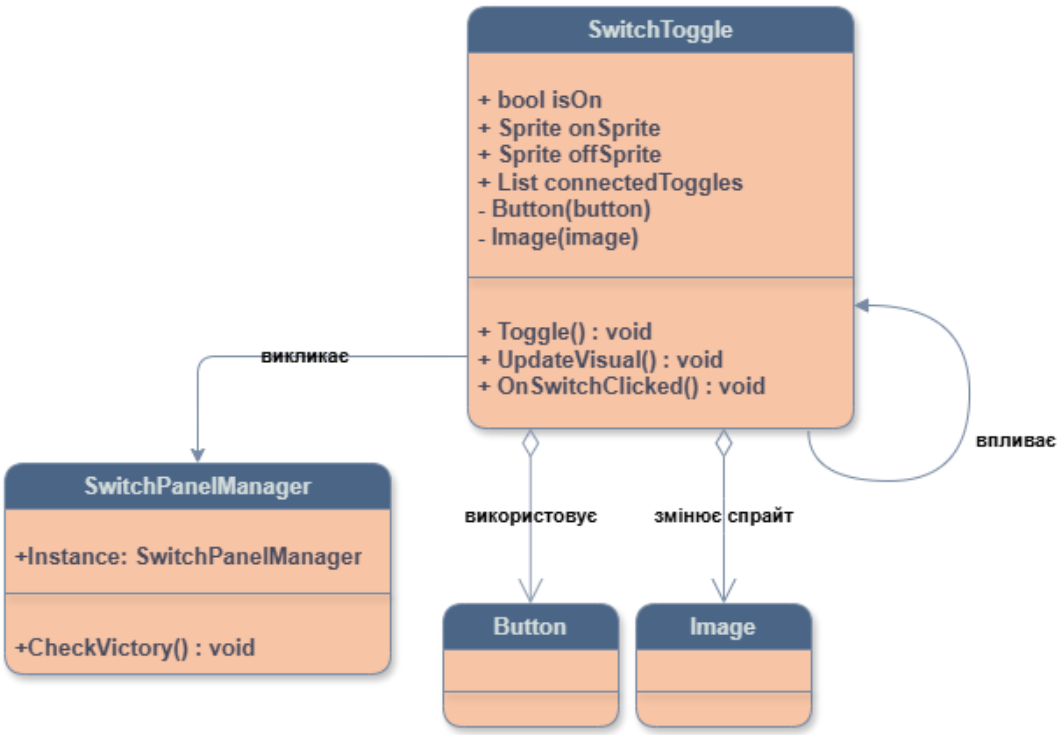


Рисунок Б.5 – Діаграма класів компонента SwitchToggle

ДОДАТОК В

Макети до інтерфейсів користувача у програмі Adobe Photoshop

Рисунок В.1 – Макет «Головного меню» у програмі «Adobe Photoshop»

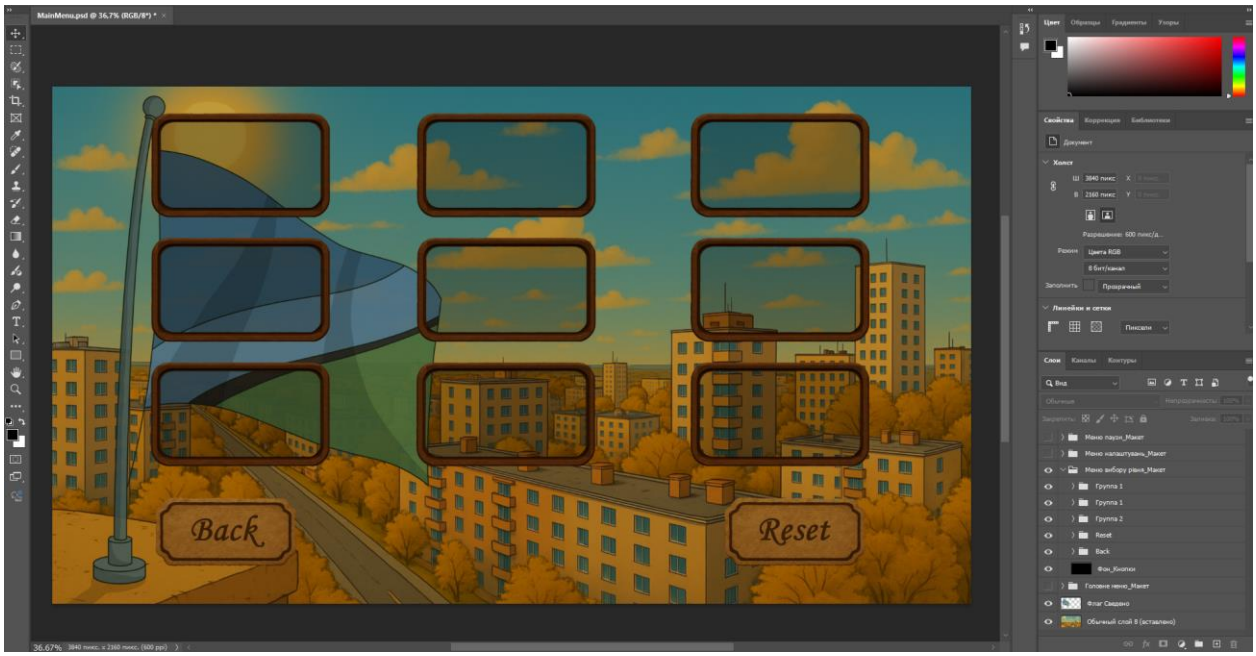


Рисунок В.2 – Макет «Меню вибору рівня» у програмі «Adobe Photoshop»

Рисунок В.3 – Макет «Меню налаштувань» у програмі «Adobe Photoshop»

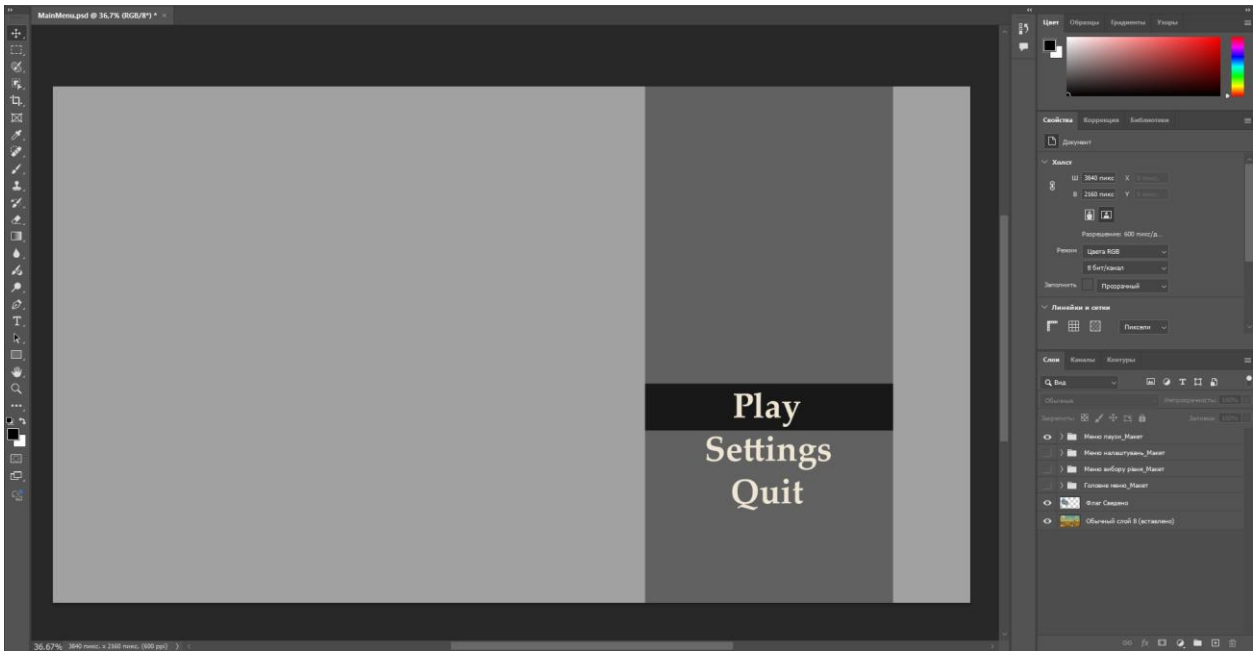


Рисунок В.4 – Макет «Меню паузи» у програмі «Adobe Photoshop»

ДОДАТОК Г

Інтерфейси користувача у середовищі розробки

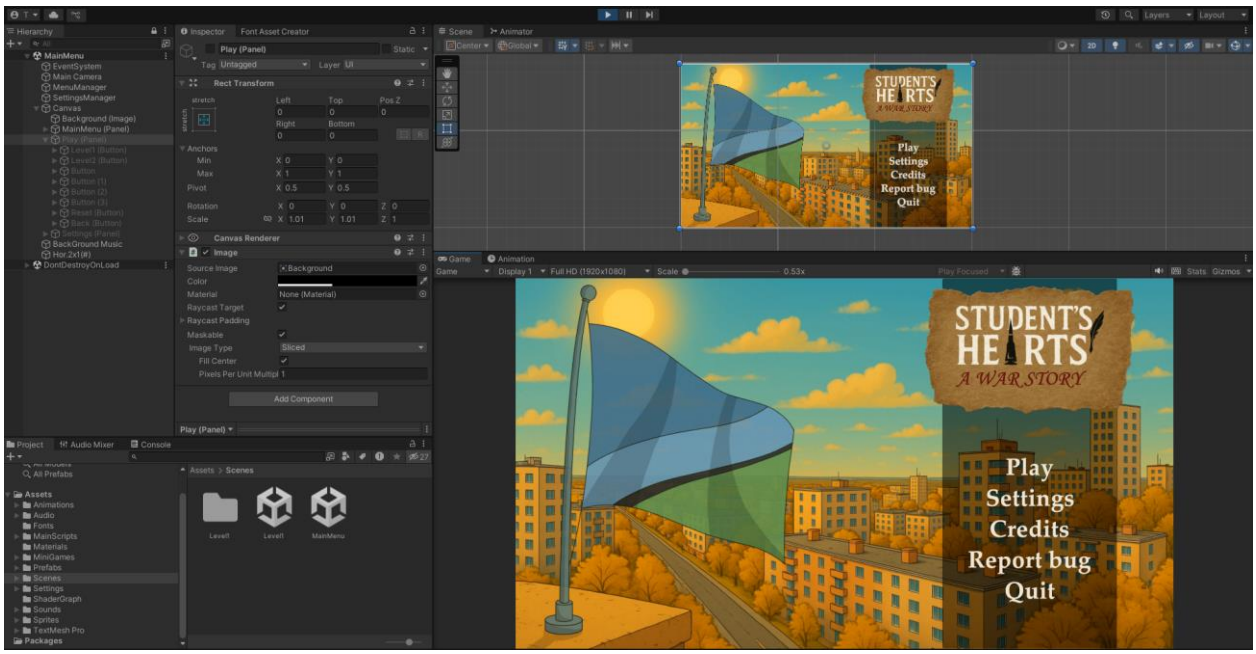


Рис. Г.1 – «Головне меню» у середовищі розробки Unity

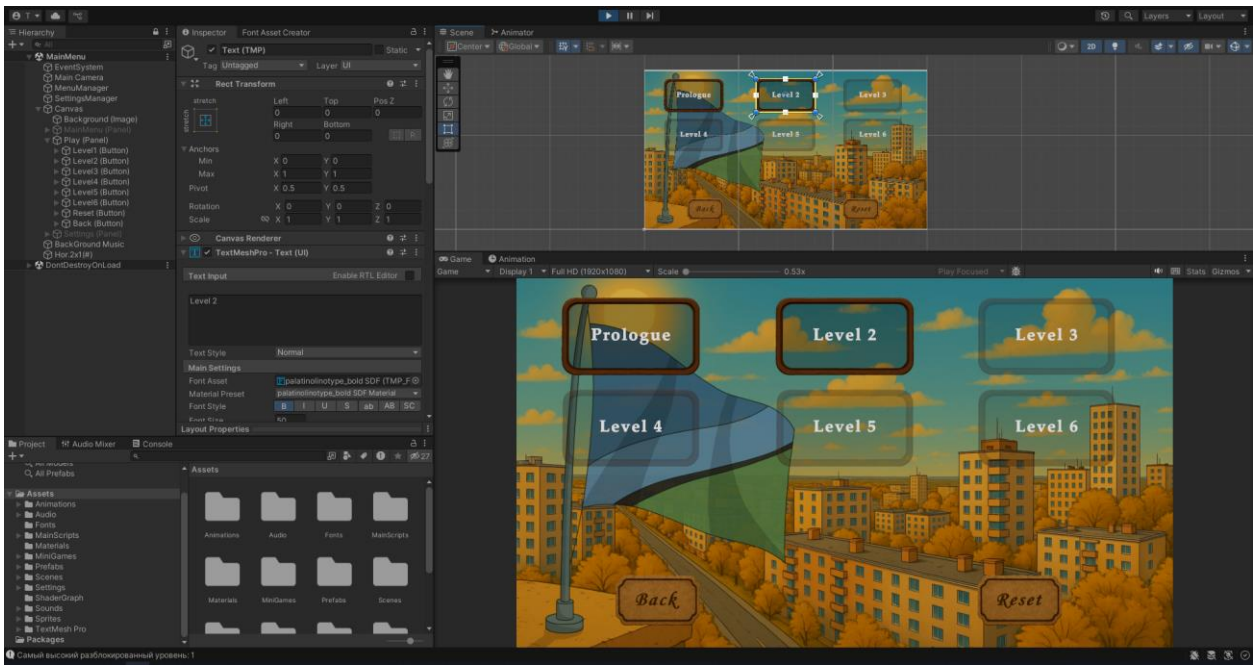


Рисунок Г.2 – «Меню вибору рівня» у середовищі розробки Unity

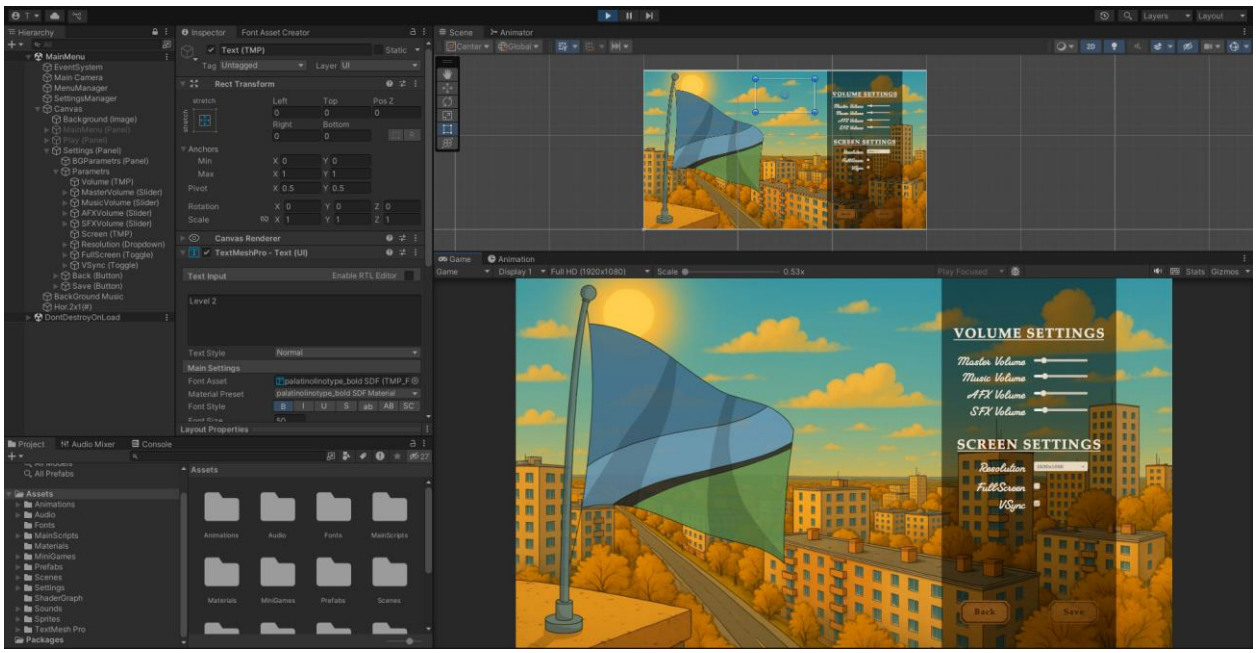


Рисунок Г.3 – «Меню налаштувань» у середовищі розробки Unity

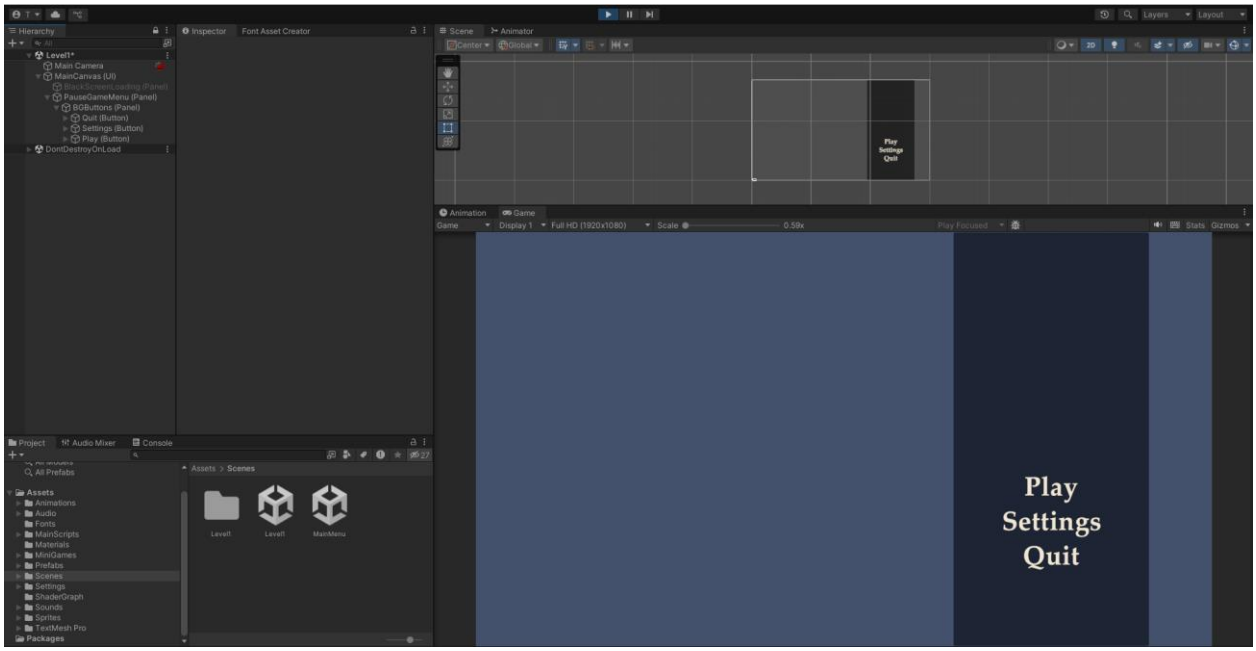


Рисунок Г.4 – «Меню паузи» у середовищі розробки Unity

ДОДАТОК Д

Процес створення анімаційного контенту

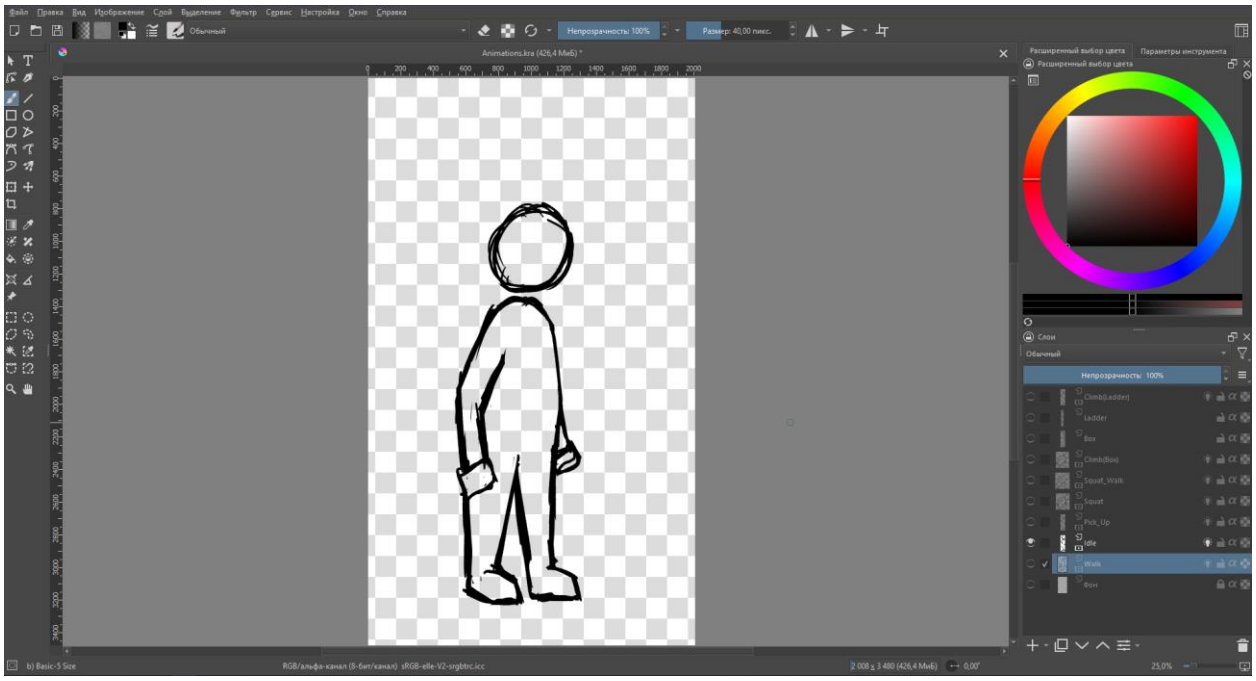


Рисунок Д.1 – Дизайн тестового персонажу у середовищі Krita

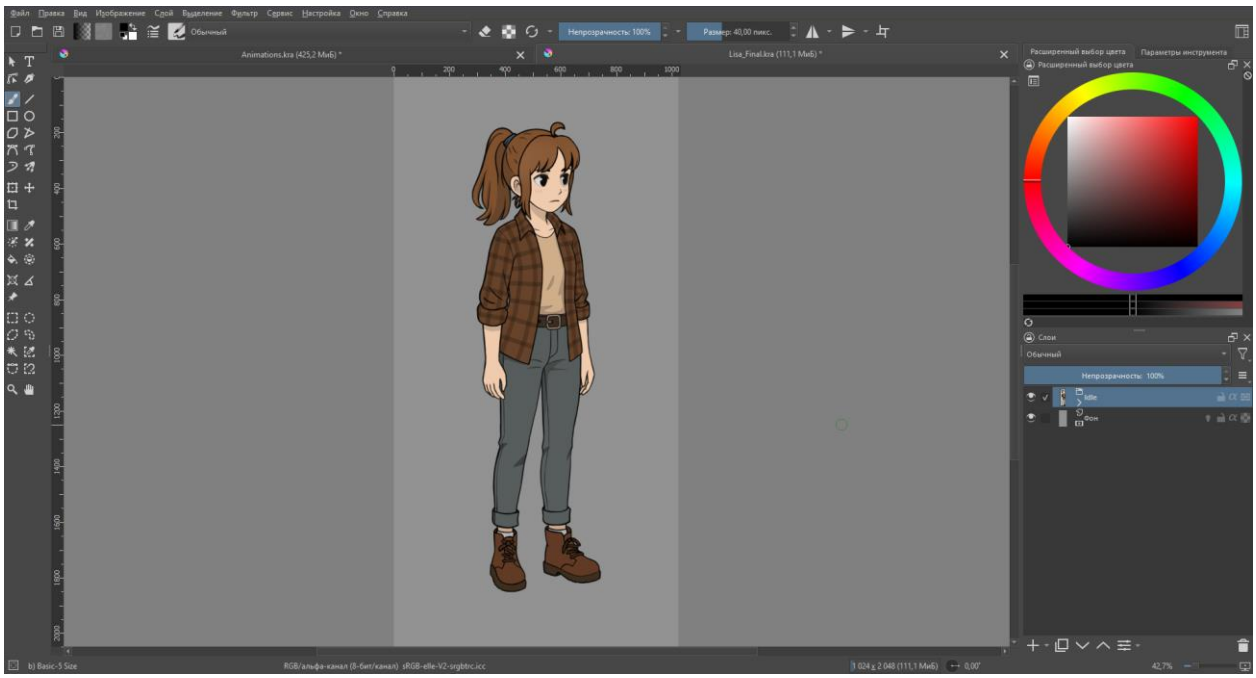


Рисунок Д.2 – Фінальний дизайн головної героїні

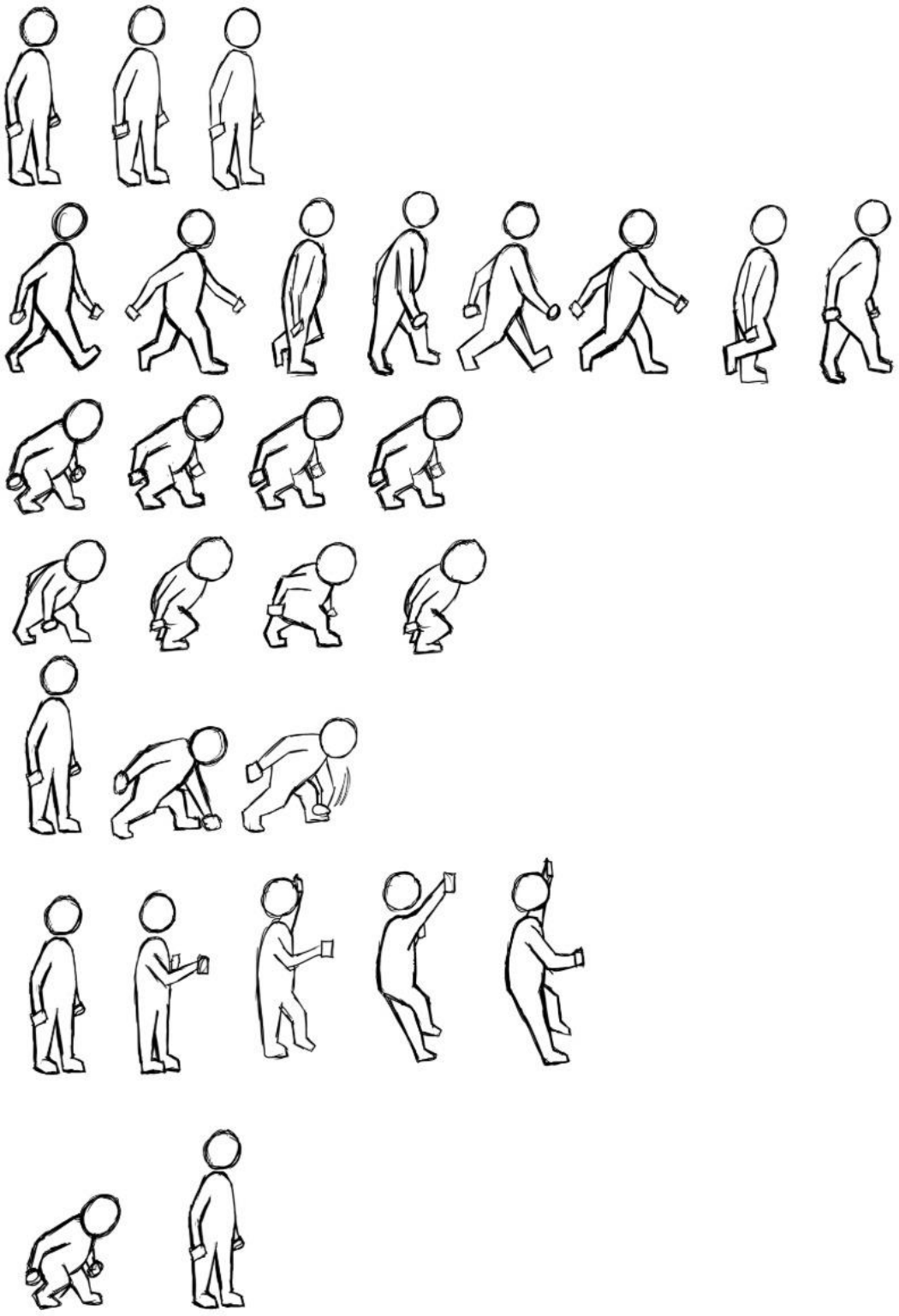


Рисунок Д.3 – Кадри анімацій для тестового персонажа

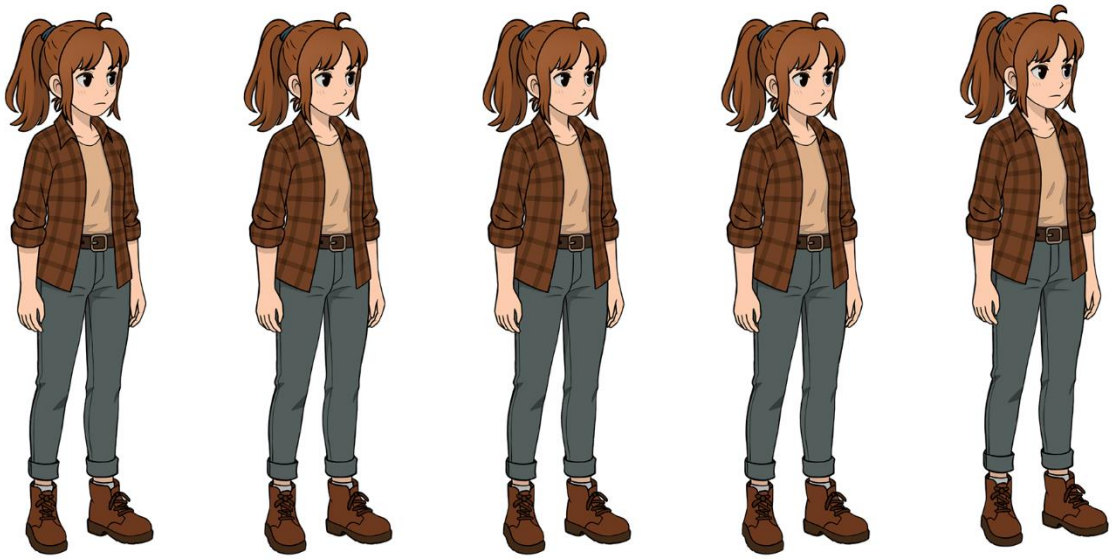


Рисунок Д.4 – Фінальний дизайн кадрів анімації «Idle» для головної героїні

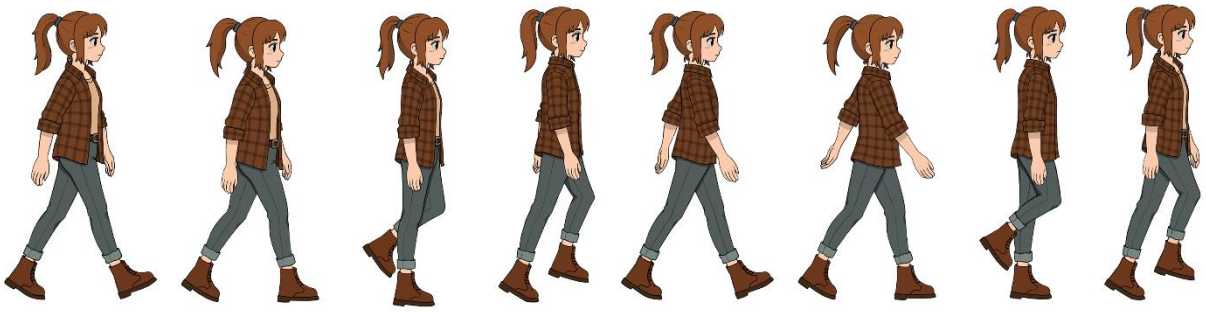


Рисунок Д.5 – Фінальний дизайн кадрів анімації «Walk» для головної героїні

ДОДАТОК Е

Дизайни NPC у програмі Krita

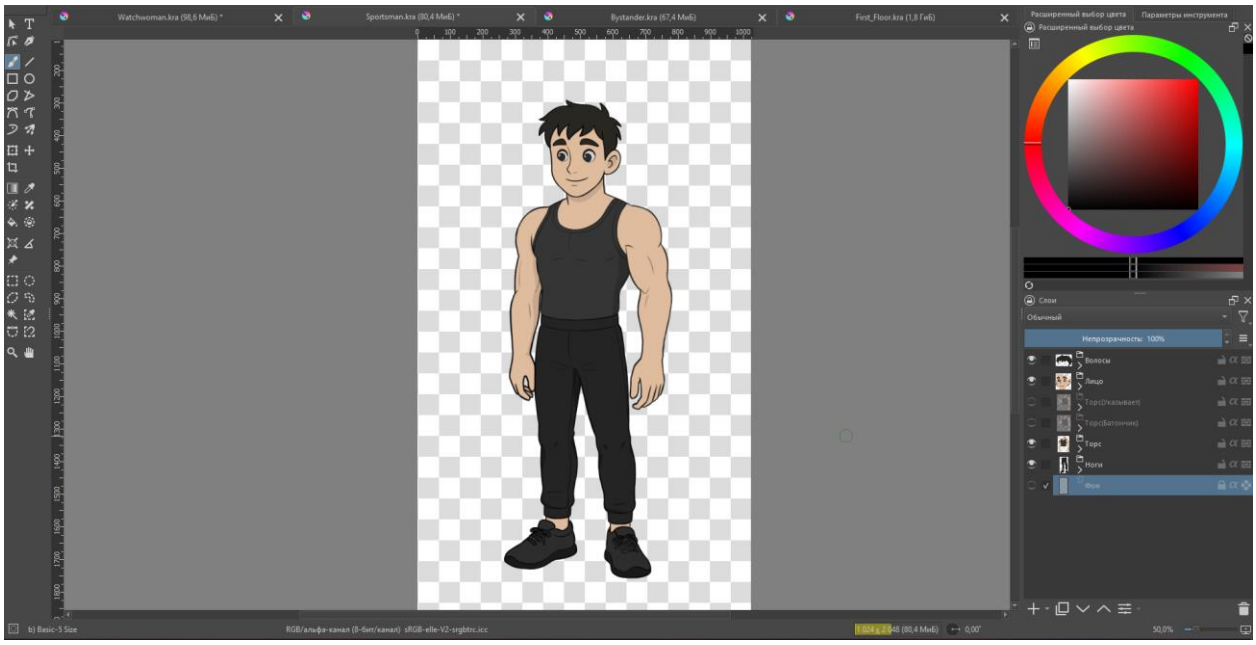


Рисунок Е.1 – Дизайн спортсмена у програмі Krita

Рисунок Е.2 – Дизайн нового жильця у програмі Krita

Рисунок Е.3 – Дизайн вахтерки у програмі Krita

ДОДАТОК Ж

Створення та налаштування Animator Controller

Рисунок Ж.1 – Параметри для Controller у середовищі розробки Unity

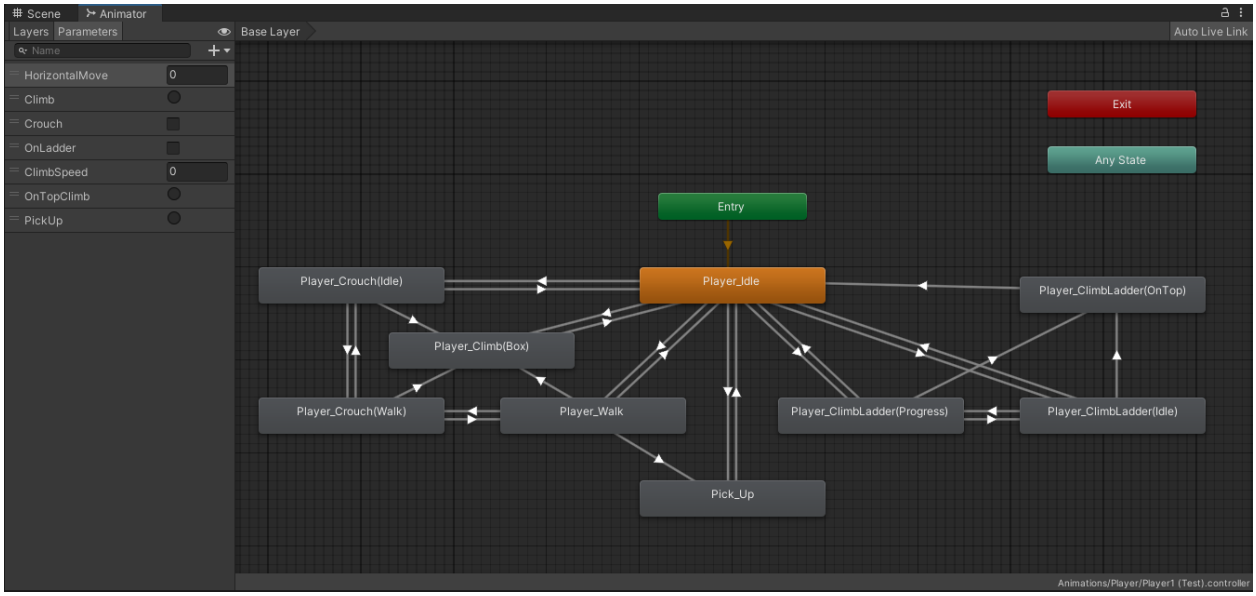
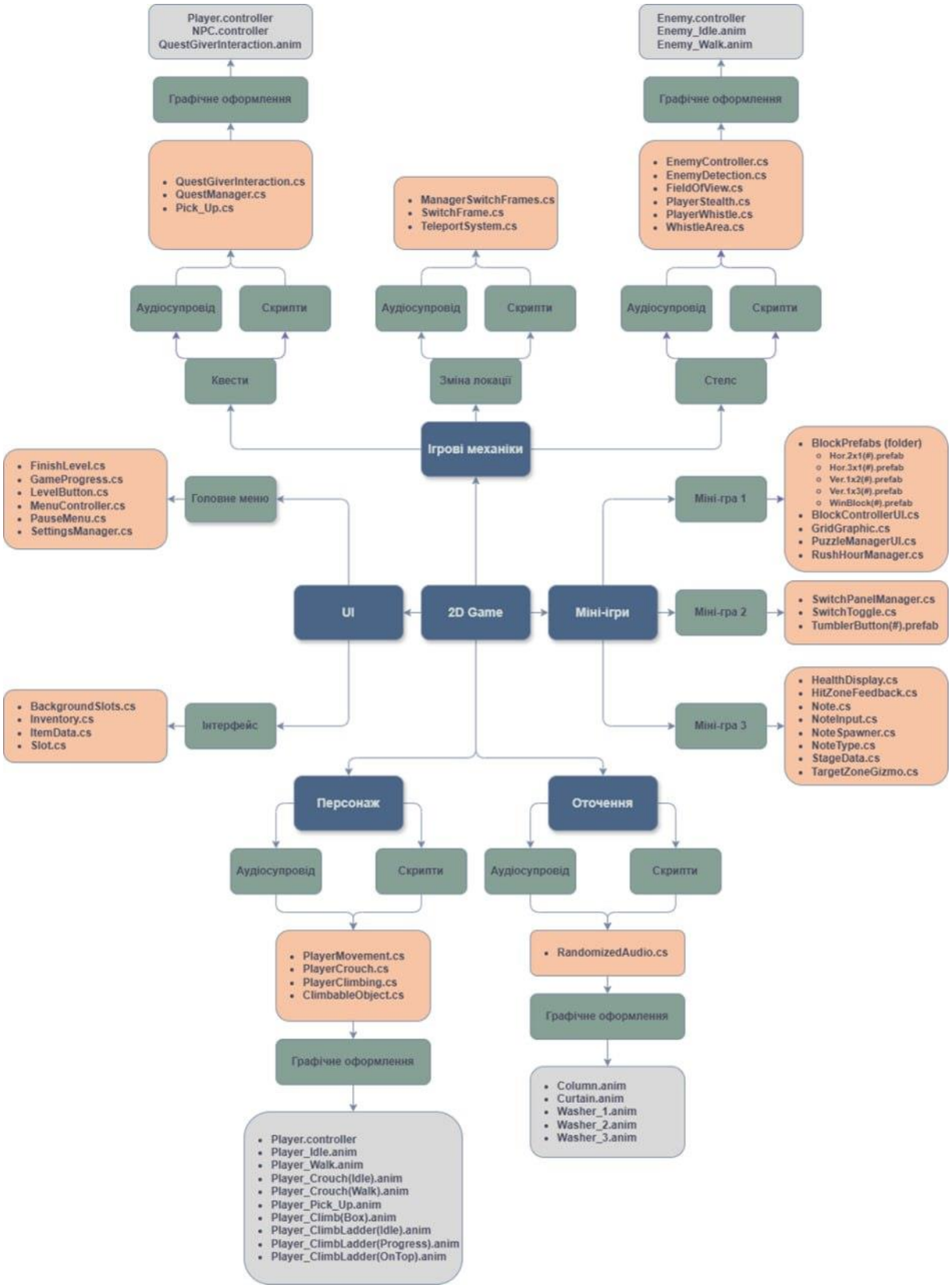


Рисунок Ж.2 – Налаштований Animator Controller у середовищі розробки Unity

ДОДАТОК З

Діаграма ієрархії підсистем із назвами скриптів



ДОДАТОК И

Дизайн ігрового простору

Рисунок И.1 – Дизайн першої локації у програмі Krita

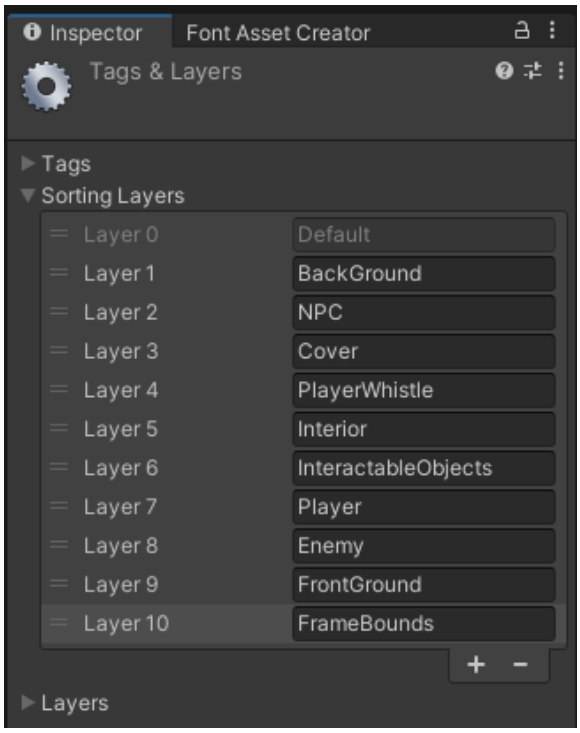


Рисунок И.2 – Налаштування системи Sorting Layers у Unity

ДОДАТОК К

Лістинг коду PlayerMovement.cs

```
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D))]
public class PlayerMovement : MonoBehaviour
{
    [Header("Movement")]
    public float speed = 5f;           // Швидкість руху
    private float moveInput;           // Значення осі "Horizontal"

    private bool facingRight = true;  // Напрямок, в якому дивиться персонаж
    private bool canMove = true;       // Прапорець дозволу руху

    private Rigidbody2D rb;            // Компонент Rigidbody2D для фізики
    private Animator animator;         // Компонент Animator для керування
анімаціями

    private void Awake()
    {
        // Отримуємо компоненти Rigidbody2D та Animator
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
    }

    private void Update()
    {
        if (!canMove)
        {
            // Якщо рух заблокований — скидаємо анімацію руху
            if (animator != null)
            {
                animator.SetFloat("HorizontalMove", 0f);
            }
            return;
        }

        // Зчитування горизонтального вводу
        moveInput = Input.GetAxis("Horizontal");

        // Передача значення руху в аніматор
        if (animator != null)
        {
            animator.SetFloat("HorizontalMove", Mathf.Abs(moveInput));
        }
    }

    private void FixedUpdate()
    {
        if (!canMove) return; // Якщо рух заборонено — вихід

        // Рух персонажа по осі X
        rb.velocity = new Vector2(moveInput * speed, rb.velocity.y);

        // Зміна напрямку, якщо потрібно
        if (!facingRight && moveInput > 0) Flip();
        else if (facingRight && moveInput < 0) Flip();
    }

    /// <summary>
    /// Дозволяє або забороняє рух персонажа.
```

```
    /// Якщо заборонено — обнуляє швидкість та скидає анімацію.
    /// </summary>
    public void SetCanMove(bool value)
    {
        canMove = value;
        Debug.Log($"SetCanMove called with value: {value}, canMove is now:
{canMove}"); // Відлагодження

        if (!value)
        {
            rb.velocity = Vector2.zero;
            if (animator != null)
            {
                animator.SetFloat("HorizontalMove", 0f);
            }
        }
    }
    /// <summary>
    /// Викликається через Animation Event в анімації підбору.
    /// Уповідомлює скрипт Pickup про підбір предмета.
    /// </summary>
    public void OnPickupItem()
    {
        Debug.Log("OnPickupItem called"); // Відлагодження
        Pickup.OnItemPickedUp(); // Повідомляємо про підбір предмета
    }

    /// <summary>
    /// Перевіряє, чи дивиться персонаж вправо.
    /// </summary>
    public bool IsFacingRight()
    {
        return facingRight;
    }

    /// <summary>
    /// Здійснює віддзеркалення персонажа по осі X.
    /// </summary>
    private void Flip()
    {
        facingRight = !facingRight;
        Vector3 scaler = transform.localScale;
        scaler.x *= -1;
        transform.localScale = scaler;
    }

    /// <summary>
    /// Примусовий переворот для анімації на драбині.
    /// </summary>
    public void FlipForLadder()
    {
        Flip();
    }

    /// <summary>
    /// Повертає стан можливості руху.
    /// </summary>
    public bool GetCanMove()
    {
        return canMove;
    }
}
```

ДОДАТОК Л

Повний лістинг коду PlayerClimbing.cs

```
using UnityEngine;

[RequireComponent(typeof(PlayerMovement))]
public class PlayerClimbing : MonoBehaviour
{
    [Header("Налаштування лазіння")]
    public float rayDistance = 1f; // Відстань променя для виявлення поверхні
    public LayerMask climbableLayer; // Шар об'єктів, по яких можна лазити
    public KeyCode climbKey = KeyCode.Space; // Клавіша для початку лазіння

    private PlayerMovement playerMovement; // Компонент керування рухом
    private bool isClimbing = false; // Прапорець стану лазіння
    private Animator animator; // Посилання на Animator
    private Rigidbody2D rb; // Посилання на Rigidbody2D

    private void Awake()
    {
        // Ініціалізація компонентів
        playerMovement = GetComponent<PlayerMovement>();
        animator = GetComponent<Animator>();
        rb = GetComponent<Rigidbody2D>();
    }

    private void Update()
    {
        // Якщо гравець вже лазить — нічого не робимо
        if (isClimbing)
            return;

        // Визначаємо напрямок променя залежно від того, куди дивиться
персонаж
        Vector2 direction = playerMovement.IsFacingRight() ? Vector2.right :
Vector2.left;

        // Випускаємо промінь для виявлення climbable поверхні
        RaycastHit2D hit = Physics2D.Raycast(
            transform.position,
            direction,
            rayDistance,
            climbableLayer
        );

        Debug.DrawRay(transform.position, direction * rayDistance,
Color.yellow);

        // Починаємо лазіння, якщо знайдено поверхню і натиснуто клавішу
        if (hit.collider != null && Input.GetKeyDown(climbKey))
        {
            StartClimbing();
        }
    }

    private void StartClimbing()
    {
        // Блокуємо керування та фізику, активуємо анімацію
        isClimbing = true;
        playerMovement.SetCanMove(false);
        rb.isKinematic = true;
        rb.velocity = Vector2.zero;
```

```
            if (animator != null)
            {
                animator.SetTrigger("Climb");
            }
        }

        public void OnClimbAnimationEnd()
        {
            // Завершуємо лазіння, повертаємо керування і фізику
            isClimbing = false;
            rb.isKinematic = false;
            playerMovement.SetCanMove(true);
            Debug.Log("Лазіння завершено, керування повернуто.");
        }

        private void OnAnimatorMove()
        {
            // Застосовуємо root motion для переміщення гравця під час лазіння
            if (isClimbing && animator != null)
            {
                Vector3 delta = animator.deltaPosition;
                delta.z = 0f; // Ігноруємо зміщення по осі Z
                transform.position += delta;
            }
        }
    }
```

ДОДАТОК М

Повний лістинг коду PlayerCrouch.cs

```
using UnityEngine;

[RequireComponent(typeof(PlayerMovement))]
public class PlayerCrouch : MonoBehaviour
{
    [Header("Налаштування присідання")]
    [Tooltip("Клавіша, за допомогою якої персонаж буде присідати")]
    public KeyCode crouchKey = KeyCode.C;

    [Tooltip("Швидкість руху персонажа у режимі присідання")]
    public float crouchSpeed = 2f;

    [Header("Налаштування колайдера")]
    [Tooltip("Колайдер гравця (використовується один і той самий)")]
    public BoxCollider2D playerCollider;

    [Tooltip("Розмір колайдера під час присідання")]
    public Vector2 crouchColliderSize = new Vector2(1f, 0.5f);

    [Tooltip("Зміщення колайдера під час присідання")]
    public Vector2 crouchColliderOffset = new Vector2(0f, -0.25f);

    [Header("Перевірка землі")]
    [Tooltip("Точка для перевірки того, що персонаж стоїть на землі")]
    public Transform groundCheck;
    [Tooltip("Радіус сфери для перевірки землі")]
    public float checkRadius = 0.2f;
    [Tooltip("Шар, що позначає землю")]
    public LayerMask groundLayer;

    private PlayerMovement playerMovement;
    private Rigidbody2D rb;
    private Animator animator;

    private float originalSpeed;
    private Vector2 originalColliderSize;
    private Vector2 originalColliderOffset;

    // <-- Приватне поле
    private bool isCrouching = false;

    // <-- Публічна властивість для перевірки стану присідання з інших скриптів
    public bool IsCrouching
    {
        get { return isCrouching; }
    }

    private void Awake()
    {
        // Зберігаємо посилання на скрипт, що відповідає за рух
        playerMovement = GetComponent<PlayerMovement>();

        // Отримуємо Rigidbody2D і Animator для роботи зі швидкістю та
анімаціями
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();

        // Зберігаємо початкову швидкість для повернення після присідання
        originalSpeed = playerMovement.speed;
```

```
        // Зберігаємо початкові розміри та зміщення колайдера
        if (playerCollider != null)
        {
            originalColliderSize = playerCollider.size;
            originalColliderOffset = playerCollider.offset;
        }
        else
        {
            Debug.LogWarning("playerCollider не призначений в інспекторі!");
        }
    }

    private void Update()
    {
        // Перевірка натиснення клавіші та контакту з землею
        if (Input.GetKeyDown(crouchKey) && IsGrounded())
        {
            // Перемикаємо стан: якщо вже сидимо — встаємо, інакше — присідаємо
            if (isCrouching)
            {
                Uncrouch();
            }
            else
            {
                Crouch();
            }
        }

        // Якщо персонаж присів, але втратив контакт із землею — підводимось
        if (isCrouching && !IsGrounded())
        {
            Debug.Log("Персонаж втратив контакт із землею, вихід із
присідання.");
            Uncrouch();
        }

        // Якщо персонаж присів — оновлюємо параметр анімації
        if (isCrouching && rb != null && animator != null)
        {
            float horizontalSpeed = Mathf.Abs(rb.velocity.x);
            animator.SetFloat("HorizontalMove", horizontalSpeed);
        }
    }

    /// <summary>
    /// Перевіряє, чи торкається персонаж землі.
    /// </summary>
    private bool IsGrounded()
    {
        if (groundCheck == null)
        {
            Debug.LogWarning("Не призначено groundCheck! Перевірка землі не
працюватиме коректно.");
            return false;
        }

        // OverlapCircle повертає true, якщо сфера перетинає землю
        return Physics2D.OverlapCircle(groundCheck.position, checkRadius,
groundLayer);
    }

    /// <summary>
    /// Логіка при початку присідання.
```

```
    /// </summary>
    public void Crouch()
    {
        isCrouching = true;

        // Встановлюємо зменшену швидкість
        playerMovement.speed = crouchSpeed;

        // Змінюємо розмір і зміщення колайдера
        if (playerCollider != null)
        {
            playerCollider.size = crouchColliderSize;
            playerCollider.offset = crouchColliderOffset;
        }

        // Встановлюємо параметр анімації "Crouch" у true
        if (animator != null)
        {
            animator.SetBool("Crouch", true);
        }

        Debug.Log("Персонаж ПРИСІВ. Швидкість зменшена, колайдер змінено,
анімація активна.");
    }

    /// <summary>
    /// Логіка при виході з присідання.
    /// </summary>
    public void Uncrouch()
    {
        isCrouching = false;

        // Повертаємо початкову швидкість
        playerMovement.speed = originalSpeed;

        // Повертаємо початкові розміри та зміщення колайдера
        if (playerCollider != null)
        {
            playerCollider.size = originalColliderSize;
            playerCollider.offset = originalColliderOffset;
        }

        // Встановлюємо параметр анімації "Crouch" у false
        if (animator != null)
        {
            animator.SetBool("Crouch", false);
        }

        Debug.Log("Персонаж ВСТАВ. Швидкість і колайдер відновлено, анімація
вимкнена.");
    }
}
```

ДОДАТОК Н

Повний лістинг коду PlayerLadder.cs

```
using UnityEngine;

[RequireComponent(typeof(PlayerMovement))]
[RequireComponent(typeof(Rigidbody2D))]
public class PlayerLadder : MonoBehaviour
{
    [Header("Налаштування драбини")]
    public float climbSpeed = 3f;           // Швидкість підйому по драбині
    public string verticalAxis = "Vertical";// Назва осі вводу (зазвичай
"Vertical")
    public KeyCode jumpKey = KeyCode.Space; // Клавіша стрибка для зістрибування
з драбини

    [Header("Налаштування колайдера гравця")]
    public BoxCollider2D playerCollider;
    public Vector2 normalColliderSize = new Vector2(1f, 2f);
    public Vector2 normalColliderOffset = new Vector2(0f, 0f);
    public Vector2 climbingColliderSize = new Vector2(1f, 2f);
    public Vector2 climbingColliderOffset = new Vector2(0f, 0f);

    private bool isOnLadder = false;          // Чи перебуває гравець у зоні
драбини
    private bool isClimbing = false;        // Чи відбувається фактичне лазіння
(OnLadder в Animator)
    private bool isOnTopSequence = false;    // Прапорець активної анімації
"OnTopClimb" (вихід догори)
    private bool isFromTopSequence = false;  // Прапорець анімації
"FromTopClimb" (спуск згори)

    private PlayerMovement playerMovement;
    private Rigidbody2D rb;
    private PlayerCrouch playerCrouch;
    private Animator animator;

    private float originalGravity;
    private BoxCollider2D currentLadderCollider;

    public LadderZone.LadderClimbDirection climbDirection;

    private Vector2 onTopFinalPosition;
    private Vector2 fromTopFinalPosition;

    private void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
        playerMovement = GetComponent<PlayerMovement>();
        playerCrouch = GetComponent<PlayerCrouch>();
        animator = GetComponent<Animator>();
    }

    private void Start()
    {
        originalGravity = rb.gravityScale;
    }

    private void Update()
    {
        if (isOnLadder)
        {
```

```
            float verticalInput = Input.GetAxis(verticalAxis);

            if (!isClimbing && !isOnTopSequence && !isFromTopSequence &&
Mathf.Abs(verticalInput) > 0.1f)
            {
                StartClimbing();
            }

            if (isClimbing && Input.GetKeyDown(jumpKey) && !isOnTopSequence &&
!isFromTopSequence)
            {
                StopClimbing();
            }
        }
        else
        {
            if (isClimbing)
            {
                StopClimbing();
            }
        }
    }

    private void FixedUpdate()
    {
        if (isClimbing)
        {
            if (isFromTopSequence)
            {
                rb.velocity = Vector2.zero;
                if (animator != null)
                {
                    animator.SetFloat("ClimbSpeed", 0f);
                }
            }
            else
            {
                float verticalInput = Input.GetAxis(verticalAxis);
                rb.velocity = new Vector2(0f, verticalInput * climbSpeed);

                if (animator != null)
                {
                    animator.SetFloat("ClimbSpeed", Mathf.Abs(verticalInput));
                }
            }
        }
    }

    private void StartClimbing()
    {
        if (isClimbing) return;

        if (playerCrouch != null && playerCrouch.IsCrouching)
        {
            playerCrouch.Uncrouch();
        }

        isClimbing = true;
        playerMovement.SetCanMove(false);
        rb.gravityScale = 0f;
        rb.velocity = Vector2.zero;

        if (playerCollider != null)
        {
```

```
            playerCollider.size = climbingColliderSize;
            playerCollider.offset = climbingColliderOffset;
        }

        if (currentLadderCollider != null)
        {
            float centerX = currentLadderCollider.bounds.center.x;
            transform.position = new Vector2(centerX, transform.position.y);
        }

        ForceFaceClimbDirection();

        if (animator != null)
        {
            animator.SetBool("OnLadder", true);
        }

        Debug.Log("StartClimbing: гравець почав лазіння.");
    }

    public void StopClimbing()
    {
        if (!isClimbing) return;

        isClimbing = false;
        rb.gravityScale = originalGravity;
        playerMovement.SetCanMove(true);

        if (playerCollider != null)
        {
            playerCollider.size = normalColliderSize;
            playerCollider.offset = normalColliderOffset;
        }

        if (animator != null)
        {
            animator.SetBool("OnLadder", false);
        }

        Debug.Log("StopClimbing: гравець припинив лазіння.");
    }

    public void ForceStartClimbing(LadderZone.LadderClimbDirection direction)
    {
        climbDirection = direction;
        if (!isClimbing)
        {
            StartClimbing();
        }
    }

    public void StartOnTopSequence(Vector2 topDestination)
    {
        onTopFinalPosition = topDestination;
        isOnTopSequence = true;

        StopClimbing();
        playerMovement.SetCanMove(false);

        if (animator != null)
        {
            animator.SetTrigger("OnTopClimb");
        }
```

```
        Debug.Log("StartOnTopSequence: запущено анімацію OnTopClimb.");
    }

    public void OnTopAnimationFinished()
    {
        if (isOnTopSequence)
        {
            transform.position = onTopFinalPosition;
            playerMovement.SetCanMove(true);
            isOnTopSequence = false;

            if (animator != null)
            {
                animator.ResetTrigger("OnTopClimb");
            }

            Debug.Log("OnTopAnimationFinished: гравець завершив підйом.");
        }
    }

    public void StartFromTopSequence(Vector2 ladderTopPosition)
    {
        fromTopFinalPosition = ladderTopPosition;
        isFromTopSequence = true;

        if (animator != null)
        {
            animator.SetTrigger("FromTopClimb");
        }

        Debug.Log("StartFromTopSequence: запущено анімацію FromTopClimb.");
    }

    public void OnFromTopAnimationFinished()
    {
        if (isFromTopSequence)
        {
            transform.position = fromTopFinalPosition;
            isFromTopSequence = false;

            if (animator != null)
            {
                animator.ResetTrigger("FromTopClimb");
            }

            Debug.Log("OnFromTopAnimationFinished: гравець телепортований до
вершини драбини.");
        }
    }

    public bool IsClimbing()
    {
        return isClimbing;
    }

    public bool IsFromTopSequence()
    {
        return isFromTopSequence;
    }

    public void SetCurrentLadderCollider(BoxCollider2D col)
    {
        currentLadderCollider = col;
    }
```

```
    public void SetClimbDirection(LadderZone.LadderClimbDirection direction)
    {
        climbDirection = direction;
    }

    private void ForceFaceClimbDirection()
    {
        bool isCurrentlyRight = playerMovement.IsFacingRight();

        if ((climbDirection == LadderZone.LadderClimbDirection.Right &&
!isCurrentlyRight) ||
            (climbDirection == LadderZone.LadderClimbDirection.Left &&
isCurrentlyRight))
        {
            playerMovement.FlipForLadder();
        }
    }

    public void SetIsOnLadder(bool onLadder)
    {
        isOnLadder = onLadder;

        if (!onLadder && isClimbing)
        {
            StopClimbing();
        }
    }
}
```

ДОДАТОК П

Повний лістинг коду EnemyController.cs

```
using UnityEngine;

[System.Serializable]
public class PatrolPoint
{
    public Transform pointTransform; // Точка патрулювання
    public float waitTime;           // Час очікування на цій точці
}

public class EnemyController : MonoBehaviour
{
    #region Змінні

    [Header("Патрулювання")]
    public PatrolPoint[] patrolPoints;      // Масив точок патрулювання
    public float patrolSpeed = 2f;          // Швидкість руху
    public bool loopPatrol = true;          // Чи патрулювати по колу
    public bool flipXForDirection = true;   // Візуально віддзеркалювати спрайт

    private int currentPatrolIndex = 0;
    private bool patrolForward = true;
    private bool isWaiting = false;
    private float waitTimer = 0;
    private float moveSpeed = 0f;

    [Header("Виявлення гравця")]
    public Transform player;
    private PlayerStealth playerStealth;
    public bool isPlayerDetected { get; private set; } = false;

    [Header("Реакція на шум")]
    private Vector3 noisePosition = Vector3.zero;
    private bool isRespondingToNoise = false;
    private bool isWaitingAtNoise = false;
    public float noiseWaitTime = 3f;
    private float noiseWaitTimer = 0f;

    [Header("Втрачення гравця")]
    public float lostPlayerWaitTime = 3f;
    private Vector3 lastKnownPlayerPos;
    private bool isGoingToLastPos = false;
    private bool isWaitingAtLastPos = false;
    private float lostWaitTimer = 0f;

    [Header("Canvas над ворогом")]
    [Tooltip("Canvas, який не повинен фліпатись разом зі спрайтом.")]
    public Transform detectionCanvas;
    [Tooltip("Зміщення Canvas.")]
    public Vector3 canvasOffset = new Vector3(0, 1.5f, 0);

    [Header("Анімація")]
    [Tooltip("Компонент Animator для керування анімацією ворога.")]
    private Animator animator;

    #endregion

    #region Ініціалізація

    private void Start()
```

```
{
    if (player != null)
    {
        playerStealth = player.GetComponent<PlayerStealth>();
    }

    animator = GetComponent<Animator>();
    if (animator == null)
    {
        Debug.LogError($"{name}: Відсутній компонент Animator!");
    }

    if (detectionCanvas != null)
    {
        detectionCanvas.SetParent(null, true);
    }
}

private void Update()
{
    moveSpeed = 0f;

    if (isGoingToLastPos || isWaitingAtLastPos)
    {
        HandleLostPlayerState();
    }
    else if (isRespondingToNoise || isWaitingAtNoise)
    {
        HandleNoiseResponse();
    }
    else if (isPlayerDetected && player != null)
    {
        HandleChasePlayer();
    }
    else
    {
        HandlePatrol();
    }

    if (animator != null)
    {
        animator.SetFloat("HorizontalMove", Mathf.Abs(moveSpeed));
    }
}

private void LateUpdate()
{
    if (detectionCanvas != null)
    {
        detectionCanvas.position = transform.position + canvasOffset;
        detectionCanvas.rotation = Quaternion.identity;
        Vector3 scale = detectionCanvas.localScale;
        scale.x = Mathf.Abs(scale.x);
        detectionCanvas.localScale = scale;
    }
}

#endregion

#region Публічні методи

public void OnPlayerDetected(Transform playerTransform)
{
    player = playerTransform;
```

```
            playerStealth = player.GetComponent<PlayerStealth>();
            isPlayerDetected = true;
            lastKnownPlayerPos = player.position;
            Debug.Log("Виявлено гравця! Починаємо переслідування.");
        }

        public void OnPlayerLost()
        {
            isPlayerDetected = false;
            isGoingToLastPos = true;
            isWaitingAtLastPos = false;
            lostWaitTimer = lostPlayerWaitTime;
            Debug.Log("Гравець втрачений. Рухаємось до останнього місця.");
        }

        public void MoveToNoise(Vector3 position)
        {
            if (isPlayerDetected) return;
            noisePosition = position;
            isRespondingToNoise = true;
            isWaitingAtNoise = false;
            Debug.Log($"Реакція на шум у точці {position}");
        }

        #endregion

        #region Приватна логіка

        private void HandleNoiseResponse()
        {
            if (isWaitingAtNoise)
            {
                noiseWaitTimer -= Time.deltaTime;
                if (noiseWaitTimer <= 0f)
                {
                    isWaitingAtNoise = false;
                    Debug.Log("Повернення до патрулювання після шуму.");
                }
                return;
            }

            if (isRespondingToNoise)
            {
                Vector3 enemyPos = transform.position;
                Vector3 targetPosXOnly = new Vector3(noisePosition.x, enemyPos.y,
enemyPos.z);
                Vector3 direction = (targetPosXOnly - enemyPos).normalized;
                moveSpeed = direction.x * 2f;
                transform.position += direction * 2f * Time.deltaTime;

                if (flipXForDirection && direction.x != 0)
                {
                    transform.localScale = new Vector3(
                        Mathf.Sign(direction.x)                                    *
Mathf.Abs(transform.localScale.x),
                        transform.localScale.y,
                        transform.localScale.z
                    );
                }

                if (Mathf.Abs(transform.position.x - targetPosXOnly.x) < 0.3f)
                {
                    isRespondingToNoise = false;
                    isWaitingAtNoise = true;
```

```
                noiseWaitTimer = noiseWaitTime;
                moveSpeed = 0f;
                Debug.Log($"Підійшли до шуму. Чекаємо {noiseWaitTime}с.");
            }
        }
    }

    private void HandleLostPlayerState()
    {
        if (isWaitingAtLastPos)
        {
            lostWaitTimer -= Time.deltaTime;
            if (lostWaitTimer <= 0f)
            {
                isWaitingAtLastPos = false;
                Debug.Log("Очікування завершено. Повертаємось до патруля.");
            }
            return;
        }

        if (isGoingToLastPos)
        {
            Vector3 enemyPos = transform.position;
            Vector3 targetPosXOnly = new Vector3(lastKnownPlayerPos.x,
enemyPos.y, enemyPos.z);
            Vector3 direction = (targetPosXOnly - enemyPos).normalized;
            moveSpeed = direction.x * 2f;
            transform.position += direction * 2f * Time.deltaTime;

            if (flipXForDirection && direction.x != 0)
            {
                transform.localScale = new Vector3(
                    Mathf.Sign(direction.x)                                    *
Mathf.Abs(transform.localScale.x),
                    transform.localScale.y,
                    transform.localScale.z
                );
            }

            if (Mathf.Abs(transform.position.x - targetPosXOnly.x) < 0.3f)
            {
                isGoingToLastPos = false;
                isWaitingAtLastPos = true;
                lostWaitTimer = lostPlayerWaitTime;
                moveSpeed = 0f;
                Debug.Log($"Дійшли до останньої позиції. Чекаємо
{lostWaitTimer}с.");
            }
        }
    }

    private void HandleChasePlayer()
    {
        if (playerStealth != null && playerStealth.isHidden)
        {
            Debug.Log("Гравець сховався. Припиняємо переслідування.");
            OnPlayerLost();
            return;
        }

        lastKnownPlayerPos = player.position;
        Vector3 direction = (player.position - transform.position).normalized;
        moveSpeed = direction.x * 2f;
        transform.position += direction * 2f * Time.deltaTime;
```

```
        if (flipXForDirection && direction.x != 0)
        {
            transform.localScale = new Vector3(
                Mathf.Sign(direction.x) * Mathf.Abs(transform.localScale.x),
                transform.localScale.y,
                transform.localScale.z
            );
        }
    }

    private void HandlePatrol()
    {
        if (patrolPoints == null || patrolPoints.Length == 0) return;

        if (isWaiting)
        {
            waitTimer -= Time.deltaTime;
            if (waitTimer <= 0f)
            {
                isWaiting = false;
            }
            return;
        }

        PatrolPoint pp = patrolPoints[currentPatrolIndex];
        Vector3 targetPos = pp.pointTransform.position;
        Vector3 direction = (targetPos - transform.position).normalized;
        moveSpeed = direction.x * patrolSpeed;
        transform.position += direction * patrolSpeed * Time.deltaTime;

        if (flipXForDirection && direction.x != 0)
        {
            transform.localScale = new Vector3(
                Mathf.Sign(direction.x) * Mathf.Abs(transform.localScale.x),
                transform.localScale.y,
                transform.localScale.z
            );
        }

        if (Vector3.Distance(transform.position, targetPos) < 0.1f)
        {
            if (pp.waitTime > 0f)
            {
                isWaiting = true;
                waitTimer = pp.waitTime;
                moveSpeed = 0f;
                Debug.Log($"{name}     очікує     {waitTimer}с     на     точці
{pp.pointTransform.name}");
            }

            if (loopPatrol)
            {
                currentPatrolIndex     =     (currentPatrolIndex     +     1)     %
patrolPoints.Length;
            }
            else
            {
                if (patrolForward)
                {
                    currentPatrolIndex++;
                    if (currentPatrolIndex >= patrolPoints.Length)
                    {
                        currentPatrolIndex = patrolPoints.Length - 1;
```

```
                        patrolForward = false;
                    }
                }
                else
                {
                    currentPatrolIndex--;
                    if (currentPatrolIndex < 0)
                    {
                        currentPatrolIndex = 0;
                        patrolForward = true;
                    }
                }
            }
        }
    }

    #endregion
}
```

ДОДАТОК Р

Повний лістинг коду ParallaxCamera.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[ExecuteInEditMode]
public class ParallaxCamera : MonoBehaviour
{
    // Делегат, який викликається при зсуві камери
    public delegate void ParallaxCameraDelegate(float deltaMovement);
    public ParallaxCameraDelegate onCameraTranslate;

    private float oldPosition;

    void Start()
    {
        oldPosition = transform.position.x;
    }

    void Update()
    {
        if (transform.position.x != oldPosition)
        {
            if (onCameraTranslate != null)
            {
                float delta = oldPosition - transform.position.x;
                onCameraTranslate(delta);
            }

            oldPosition = transform.position.x;
        }
    }
}
```

ДОДАТОК С

Повний лістинг коду ParallaxBackground.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[ExecuteInEditMode]
public class ParallaxBackground : MonoBehaviour
{
    public ParallaxCamera parallaxCamera;
    List<ParallaxLayer> parallaxLayers = new List<ParallaxLayer>();

    void Start()
    {
        if (parallaxCamera == null)
            parallaxCamera = Camera.main.GetComponent<ParallaxCamera>();
        if (parallaxCamera != null)
            parallaxCamera.onCameraTranslate += Move;
        SetLayers();
    }

    void SetLayers()
    {
        parallaxLayers.Clear();
        for (int i = 0; i < transform.childCount; i++)
        {
            ParallaxLayer                          layer                          =
transform.GetChild(i).GetComponent<ParallaxLayer>();

            if (layer != null)
            {
                layer.name = "Шар-" + i; // ← Перекладено
                parallaxLayers.Add(layer);
            }
        }
    }

    void Move(float delta)
    {
        foreach (ParallaxLayer layer in parallaxLayers)
        {
            layer.Move(delta);
        }
    }
}
```

ДОДАТОК Т

Повний лістинг коду ParralaxLayer.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[ExecuteInEditMode]
public class ParallaxLayer : MonoBehaviour
{
    public float parallaxFactor; // Коефіцієнт паралаксу (швидкість зміщення шару)

    public void Move(float delta)
    {
        Vector3 newPos = transform.localPosition;
        newPos.x -= delta * parallaxFactor; // Зсув шару залежно від руху камери

        transform.localPosition = newPos;
    }
}
```

ДОДАТОК У

Повний лістинг коду GridGraphics.cs

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.Sprites;
using System.Collections.Generic;

[RequireComponent(typeof(CanvasRenderer))]
public class GridGraphic : Graphic
{
    [Header("Налаштування сітки")]
    [Tooltip("Кількість комірок по горизонталі")]
    public int gridWidth = 6;
    [Tooltip("Кількість комірок по вертикалі")]
    public int gridHeight = 6;
    [Tooltip("Розмір комірки (в одиницях UI)")]
    public float cellSize = 100f;
    [Tooltip("Відстань між комірками (у пікселях)")]
    public float spacing = 10f;
    [Tooltip("Товщина ліній сітки (у пікселях)")]
    public float lineThickness = 2f;

    // Перевизначаємо метод, який відображає меш для UI
    protected override void OnPopulateMesh(VertexHelper vh)
    {
        vh.Clear();

        // Використовуємо колір, заданий у властивості color компонента Graphic
        Color lineColor = color;

        // Повний розмір комірки з відступом
        float cellFullSize = cellSize + spacing;

        // Малювання вертикальних ліній
        for (int x = 0; x <= gridWidth; x++)
        {
            Vector2 start = new Vector2(x * cellFullSize, 0);
            Vector2 end = new Vector2(x * cellFullSize, gridHeight *
cellFullSize);
            AddLine(vh, start, end, lineColor);
        }

        // Малювання горизонтальних ліній
        for (int y = 0; y <= gridHeight; y++)
        {
            Vector2 start = new Vector2(0, y * cellFullSize);
            Vector2 end = new Vector2(gridWidth * cellFullSize, y *
cellFullSize);
            AddLine(vh, start, end, lineColor);
        }
    }

    /// <summary>
    /// Додає лінію від start до end у VertexHelper як прямокутник (quad).
    /// </summary>
    void AddLine(VertexHelper vh, Vector2 start, Vector2 end, Color lineColor)
    {
        // Обчислюємо напрямок лінії та перпендикулярний вектор для товщини
        Vector2 direction = (end - start).normalized;
        Vector2 normal = new Vector2(-direction.y, direction.x);
        Vector2 offset = normal * (lineThickness * 0.5f);
```

```
            // Визначаємо 4 вершини для лінії у вигляді прямокутника
            Vector2 v1 = start - offset;
            Vector2 v2 = start + offset;
            Vector2 v3 = end + offset;
            Vector2 v4 = end - offset;

            int currentIndex = vh.currentVertCount;
            UIVertex vertex = UIVertex.simpleVert;
            vertex.color = lineColor;

            vertex.position = v1;
            vertex.uv0 = Vector2.zero;
            vh.AddVert(vertex);

            vertex.position = v2;
            vertex.uv0 = Vector2.up;
            vh.AddVert(vertex);

            vertex.position = v3;
            vertex.uv0 = Vector2.one;
            vh.AddVert(vertex);

            vertex.position = v4;
            vertex.uv0 = Vector2.right;
            vh.AddVert(vertex);

            vh.AddTriangle(currentIndex, currentIndex + 1, currentIndex + 2);
            vh.AddTriangle(currentIndex, currentIndex + 2, currentIndex + 3);
        }
    }
```

ДОДАТОК Ф

Повний лістинг коду SwitchToggle.cs

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class SwitchToggle : MonoBehaviour
{
    [Header("Налаштування стану")]
    [Tooltip("Початковий стан перемикача: true — увімкнено, false — вимкнено")]
    public bool isOn;

    [Header("Спрайти для станів")]
    public Sprite onSprite;
    public Sprite offSprite;

    [Header("Пов'язані перемикачі")]
    [Tooltip("Список перемикачів, стан яких зміниться при натисканні на цей")]
    public List<SwitchToggle> connectedToggles;

    private Button button;
    private Image image;

    private void Awake()
    {
        button = GetComponent<Button>();
        image = GetComponent<Image>();

        if (button != null)
            button.onClick.AddListener(OnSwitchClicked);

        UpdateVisual();
    }

    // Викликається при натисканні на перемикач
    private void OnSwitchClicked()
    {
        // Перемикаємо стан самого перемикача
        Toggle();

        // Перемикаємо стан пов'язаних перемикачів
        foreach (SwitchToggle other in connectedToggles)
        {
            if (other != null)
                other.Toggle();
        }

        // Перевірка умови перемоги
        SwitchPanelManager.Instance.CheckVictory();
    }

    // Метод для зміни стану
    public void Toggle()
    {
        isOn = !isOn;
        UpdateVisual();
    }

    // Оновлення зображення перемикача відповідно до стану
    private void UpdateVisual()
    {
```

```
            if (image != null)
            {
                image.sprite = isOn ? onSprite : offSprite;
            }
        }
    }
```

ДОДАТОК Х

Апробація результатів роботи

Міністерство освіти і науки України
Одеський національний технологічний університет
Вінницький національний технічний університет
Інститут комп`ютерної інженерії, автоматизації,
робототехніки та програмування ім.П.Н.Платонова

ПРОГРАМА

ІІІ ВСЕУКРАЇНСЬКОЇ
НАУКОВО – ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ
МОЛОДИХ ВЧЕНИХ, АСПІРАНТІВ
ТА СТУДЕНТІВ

«КОМП'ЮТЕРНІ ІГРИ І МУЛЬТИМЕДІА
ЯК ІННОВАЦІЙНИЙ ПІДХІД
ДО КОМУНІКАЦІЇ - 2023»

28-29 вересня 2023 р.
ОДЕСА

УДК 004.42+37.06

КІБЕРСПОРТ У ВИЩИХ НАВЧАЛЬНИХ ЗАКЛАДАХ: РОЗВИТОК ТА МОЖЛИВОСТІ

ЖЕРНОВИЙ М.О., БАТАЛОВ С.Д., БРАТЕРСЬКА Н.М.
(mykyta.zhernovyi@kname.edu.ua, serhii.batalov@kname.edu.ua, nataliia.braterska@kname.edu.ua)
Харківський національний університет міського господарства імені О. М. Бекетова

Навчання в XXI столітті є невід'ємною частиною шляху становлення пристойної людини, але конкретно система навчання бере початок ще за часів "Київської Русі". Вона не була переробена, а тільки доповнювалася, тому в наші дні пересічний навчальний заклад має такі ж методи навчання, як і тисячоліття тому. Але "середня температура по лікарні" не відображає усього спектру можливостей нашого часу, бо існують винятки, тенденції та новаторські ідеї, причиною яких є загальний розвиток цивілізації. Одним з багатьох новаторств є кіберспорт та його застосування в "закостенілій" системі освіти.

За весь час розвитку геймингу було створено багато ігор, деякі з них мають у своєму корені систему командної гри, в якій 2 або більше команд гравців змагаються один з одним. Саме такі ігри можуть входити до кіберспортивних дисциплін. Для прикладу, найпопулярнішими дисциплінами є:

- League of Legends – гра жанру MOBA(Multiplayer Online Battle Arena), в якій ціль кожної з команд полягає в контролю над картою та базою суперника.
- Dota 2 – така ж сама MOBA, як і League of Legends, ціль гри домінування на карті та знищення бази суперника
- Counter-Strike: Global Offensive – командна гра, в якій дві команди виконують ролі терористів та анти-терористів. Є шутером, в якому задача полягає в знищенні команди суперника, встановленні бомби або порятунку заручників.

Для участі у змаганнях з кіберспорту гравцю необхідно досконало знати особливості гри та вміти працювати в команді.

Кіберспорт як вид змагань серед гравців бере свій початок це з 1970-х років. Тоді спортивна зацікавленість була у отриманні та утриманні рекордів, але це не був саме кіберспорт, в сьогоденному розумінні. Те саме поняття «кіберспорт» з'явилося в США ще 1997 року внаслідок створення CPL (The Cyberathlete Professional League) – професійної Ліги з кіберспорту, призначенням якої була організація перших турнірів з Quake.

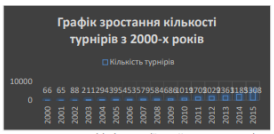


Рисунок 1 – Графік створено на основі інформації з сайту esportsearnings.com

Це дало потужний поштовх у потужний поштовх у розвитку геймінгу та кіберспорту в цілому. Трохи пізніше Counter Strike стане ключовою дисципліною кіберспортивного геймінгу, а починаючи з 2000-х почнуть з'являтися професійні ліги та турніри.

На даний момент кіберспорт є великою індустрією та охоплює велику кількість ігрових дисциплін,мільйони глядачів. Команди, які приймають участь у змаганнях мають с розвиток технологій та інтернет дав змогу проводити турніри, за якими спостерігають спонсорські контракти з великими компаніями та визнання у геймерському просторі.

Набуваючи все більшу популярність у світі кіберспорт почав привертати увагу освітян-практиків та дослідників. Тому кількість освітніх закладів, які впроваджують в кіберспорт як дисципліну зростає з кожним роком. Наприклад, у 2019 році кількість середніх шкіл, які беруть