ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка відеогри в жанрі “Мандрівний бойовик”»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Олександр КОВАЛЬ
(підпис)

Виконав: здобувач вищої освіти групи ПД-44

Олександр КОВАЛЬ

Керівник: Олесь ДІБРІВНИЙ
доктор філософії (PhD)

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Ковалю Олександру Сергійовичу

1. Тема кваліфікаційної роботи: «Розробка відеогри в жанрі “Мандрівний бойовик”»
керівник кваліфікаційної роботи доктор філософії (PhD) Олесь ДІБРІВНИЙ,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи розробки ігор на рушії Unity, опис методів імплементації ігрових механік та візуальної складової, технічна документація Unity.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд жанру “Мандрівний бойовик” та аналіз існуючих ігор з подібними жанровими ознаками й ігровими механіками, визначення вимог до програмного забезпечення.
2. Аналіз засобів реалізації тривимірних відеоігор.
3. Проєктування архітектури програмного забезпечення гри.
4. Програмна реалізація та тестування продукту.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма варіантів використання.
5. Діаграма машини кінцевих станів противника
6. Діаграма класів-компонентів гравця.
7. Екранні форми.
8. Апробація результатів дослідження

6. Дата видачі завдання «25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Огляд жанру “мандрівний бойовик” та аналіз його ознак, визначення вимог до програмного забезпечення	14.03-20.03.2025	
4	Аналіз засобів реалізації тривимірних відеоігор	21.03-22.03.2025	
5	Проєктування архітектури програмного забезпечення гри.	23.03-25.03.2025	
6	Програмна реалізація та тестування продукту	26.03-28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
8	Розробка демонстраційних матеріалів	12.05-18.05.2025	
9	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Олександр КОВАЛЬ
(підпис)

Керівник
кваліфікаційної роботи _____ Олесь ДІБРІВНИЙ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 49 стор., 1 табл., 28 рис., 20 джерел.

Мета роботи – покращення ігрового досвіду за рахунок поєднання динамічного геймплею та елементів планування.

Об'єкт дослідження – процес накопичення ігрового досвіду в однокористувацьких відеоіграх жанру Мандрівний бойовик.

Предмет дослідження – механіки, які впливають на повторюваність досвіду в грі жанру Мандрівний бойовик.

Короткий зміст роботи: У роботі проаналізовано жанр Action Roguelike та особливості його реалізації. Проведено огляд ігрових механік, які сприяють підвищенню реіграбельності, зокрема через використання випадкових елементів та менеджменту ресурсів. Розроблено однокористувацьку відеогру з бойовою системою, що базується на використанні грального карт і покерних комбінацій як засобу нанесення шкоди ворогам. Реалізовано механіку уповільнення часу для стратегічного прийняття рішень під час бою, а також систему випадкових покращень колоди після проходження рівнів. У процесі створення гри використано рушій Unity, JetBrains Rider як середовище розробки та Blender зі сторонніми моделями для побудови низькополігонального візуального стилю.

Сферою використання розробленої гри є розважальні застосунки для персональних комп'ютерів.

КЛЮЧОВІ СЛОВА: ВІДЕОГРА, UNITY, C#, ACTION, ROGUELIKE, 3D, LOW-POLY.

ЗМІСТ

ВСТУП

Актуальність: останнім часом зростає популярність інді-ігор, які роблять ставку не на масштабність чи графіку, а на цікаві механіки, здатні утримувати увагу гравця на багаторазове проходження. Особливо це стосується ігор, що поєднують простоту освоєння з певною стратегією, де кожен забіг може дати новий досвід. Сучасні однокористувацькі екшн-ігри дедалі частіше використовують елементи roguelike-жанру – варіативність, перманентну смерть, змінювані ресурси – аби зробити геймплей більш насиченим і непередбачуваним. Це не тільки підвищує реіграбельність, але й дозволяє з меншими затратами створити гру, яку захочеться проходити знову і знову. Для інді-розробників із обмеженим бюджетом і командою це стає практично ідеальним підходом. У таких умовах особливо цікаво досліджувати, як поєднати динаміку бою, елементи випадковості та стратегічного планування в одному ігровому циклі.

Об'єктом дослідження є процес накопичення ігрового досвіду в однокористувацьких відеоіграх жанру Мандрівний бойовик.

Предметом дослідження є механіки, які впливають на повторюваність досвіду у відеогрі жанру Мандрівний бойовик.

Метою роботи є покращення ігрового досвіду за рахунок поєднання динамічного геймплею та елементів планування.

Методи дослідження: Першим етапом дослідження було проведено огляд відомих ігор у жанрі *Roguelike*, а також простих інді-проєктів, що поєднують елементи цього жанру з нестандартними або новими механіками. Метою цього етапу було сформувати загальне уявлення про те, що саме робить такі ігри привабливими, глибокими та реіграбельними.

На основі зібраного аналізу було розроблено власну ідею гри, сформульовано ключові механіки та створено дизайн-документ з визначенням функціональних і нефункціональних вимог. Наступним кроком став огляд

інструментів і технологій, які відповідають вимогам до реалізації проєкту. У якості рушія було обрано Unity, що використовує мову C# для написання ігрової логіки. Для написання коду застосовувалося середовище JetBrains Rider, для базової роботи з 3D-моделями – Blender, а для контролю версій та збереження історії змін – GitHub.

Дослідження реалізації й тестування компонентів гри проводилися паралельно в процесі розробки.

Наукова новизна роботи полягає у поєднанні кількох на перший погляд несумісних механік – варіативності грального карт, їх стратегічного використання у бою, елементів менеджменту ресурсів та динамічного геймплею. Гравець не просто використовує карти як зброю, а будує комбінації з урахуванням поточних умов, при цьому впливаючи на імовірність випадіння певних карт через систему покращень.

Такий підхід дозволяє створити ігровий цикл, що забезпечує унікальний досвід при кожному проходженні.

1 АНАЛІЗ ОСОБЛИВОСТЕЙ ІГОР ЖАНРУ ACTION ROGUELIKE

1.1 Визначення жанру та його характеристик

Жанр Action Roguelike виник як еволюція класичних roguelike-ігор, поєднуючи їхні ключові елементи з динамічним екшн-геймплеєм. Цей піджанр зберігає основні характеристики roguelike, такі як процедурна генерація рівнів і перманентна смерть персонажа, але додає реальний час, швидкий темп та акцент на бойових механіках.

Термін "roguelike" походить від гри Rogue (1980), яка заклала основи жанру: випадково згенеровані рівні, покроковий геймплей, перманентна смерть та ASCII-графіка. Ці характеристики були формалізовані в "Берлінській інтерпретації" 2008 року, яка визначила ключові елементи roguelike-ігор, включаючи процедурну генерацію, перманентну смерть, покроковий геймплей та складність [1, 2].

З часом з'явилися ігри, які відходили від традиційних канонів, зберігаючи лише деякі елементи roguelike. Ці ігри отримали назву "roguelite" або "action roguelike", де геймплей відбувається в реальному часі, але зберігається перманентна смерть та процедурна генерація.

В українській локалізації платформи Steam жанр Action Roguelike перекладено як "Мандрівний бойовик". Цей термін підкреслює поєднання елементів пригодницької гри ("мандрівний") та динамічного бойового геймплею ("бойовик"). Такий переклад відображає сутність жанру: гравець подорожує через випадково згенеровані рівні, стикаючись з різноманітними ворогами та викликами, де кожне проходження є унікальним.

Характерні риси жанру "Мандрівний бойовик" включають:

– Процедурну генерацію рівнів: кожне проходження гри пропонує новий досвід, оскільки рівні створюються випадковим чином.

– Перманентну смерть: смерть персонажа означає початок гри з самого початку, що додає виклику та стимулює гравця до вдосконалення своїх навичок.

– Динамічний бойовий геймплей: битви відбуваються в реальному часі, вимагаючи від гравця швидких реакцій та стратегічного мислення.

– Складність: Цілі повинні мати кілька можливих рішень - аспект, який тісно пов'язаний з цими іграми. Це також означає, що управління ресурсами має вирішальне значення, оскільки вони обмежені [3].

1.2 Повторюваність досвіду в іграх

Повторюваність (реіграбельність) – це здатність гри залишатися цікавою при багаторазовому проходженні. Це важливий аспект, особливо для інді-розробників, які працюють з обмеженими ресурсами. Загалом, щоб досягти таких характеристик, використовуються певні елементи жанру roguelike.

Для інді-розробників, які часто мають обмежений бюджет і ресурси, створення гри з високою повторюваністю є стратегічно важливим. Замість інвестування в об'ємний контент, вони можуть зосередитися на глибоких ігрових механіках, що забезпечують різноманітний досвід у кожному проходженні. Це дозволяє утримувати інтерес гравців без значних витрат на розробку [4].

Форми повторюваності які зустрічаються найчастіше:

– Випадковість і варіативність: Використання процедурної генерації рівнів, випадкових подій або предметів створює унікальний досвід у кожному проходженні. Наприклад, у грі The Binding of Isaac кожен забіг відрізняється завдяки випадковим комбінаціям предметів, що підтримує інтерес гравців .

– Розгалуженість сюжету та вибір гравця: Ігри, що надають гравцеві можливість впливати на розвиток подій, стимулюють повторне проходження для дослідження альтернативних сценаріїв.

– Різноманітність стилів гри: Наявність різних класів персонажів, стратегій або підходів до проходження дозволяє гравцям експериментувати та відкривати нові аспекти гри.

– Поступовий прогрес та розблокування контенту: Механіки, що передбачають відкриття нових можливостей або контенту з кожним проходженням, мотивують гравців повертатися до гри.

1.3 Цільова аудиторія

Розроблювана гра орієнтована на гравців із щонайменше середнім рівнем досвіду у відеоіграх. Ігровий процес передбачає базову динаміку екшн-руху та систему бою, що вимагає реакції, але при цьому доповнена механікою сповільнення часу (slow motion), яка дозволяє гравцю приймати рішення в спокійнішому темпі. Такий підхід робить гру доступною для ширшої аудиторії.

Окрему зацікавленість гра може викликати у гравців, знайомих з настільними або картковими іграми, зокрема з базовими покерними комбінаціями, адже ігрова механіка побудована на їхньому стратегічному використанні в бою.

1.4 Аналіз існуючих ігор з аналогічними жанрами

Існує багато успішних ігор у жанрі Roguelike, що поєднують унікальні механіки з динамічним геймплеєм або стратегічними елементами але проаналізованими будуть лише ті, які найбільше вплинули на концепцію проєкту. У межах цього дослідження я не прагну критично оцінювати ці ігри, оскільки всі обрані приклади, на мою думку, реалізовані на високому рівні і всі вони по своєму унікальні та довершені. Вони слугують радше натхненням та прикладами вдалого поєднання жанрів і механік і вдалого виконання. Нижче наведено короткий аналіз кількох таких ігор:

1.4.1 Balatro

Гра, що поєднує елементи покеру та roguelike. У ній гравець використовує покерні комбінації для подолання викликів, при цьому кожен забіг відрізняється

завдяки випадковості карток, бонусів а також різними тактичними варіантами. Саме ця гра стала головним джерелом натхнення для використання гральних карт у бою та створення глибокої, але інтуїтивно зрозумілої механіки [5]. На рисунку 1.1 зображено головний екран геймплею гри Balatro.

Аналіз основних механік:

– Покерна основа: гравець формує класичні покерні комбінації з карт, щоб набрати достатню кількість фішок для проходження раунду.

– Спеціальні модифікатори: особливі карти які можуть змінювати правила гри, додаючи бонуси або змінюючи умови для комбінацій.

– Магазин між раундами: після кожного раунду гравець може купувати нові карти, модифікатори або покращення, що впливають на подальший геймплей.

– Випадкові події: деякі раунди мають особливі умови, які ускладнюють гру, наприклад, обмеження на типи комбінацій.

– Roguelike елементи: гра складається з кількох етапів, кожен з яких має зростаючу складність; програш у будь-якому з них призводить до початку гри спочатку.

Рис. 1.1 Основний екран геймплею гри Balatro

1.4.2 Heck Deck

Унікальне поєднання bullet hell-шутера та карткової гри з елементами deckbuilding – рисунок 1.2. Гра вирізняється тим, що діє в реальному часі, але для прийняття рішень використовується механіка зупинки часу. Це дало ідею використати уповільнення часу як інструмент, що балансує між динамікою бою та стратегічним мисленням [6].

Аналіз основних механік:

– Картки: усі снаряди в грі представлені у вигляді карт, які гравець повинен уникати або збирати для використання.

– Механіка часу: час у грі рухається лише тоді, коли гравець рухається, що дозволяє планувати дії в умовах інтенсивного бою.

– Обмежені ресурси: кожна зібрана карта коштує гравцеві одне очко життя, що змушує обережно підходити до вибору карт.

– Різноманітність: гра містить 5 унікальних рівнів з власними темами, музикою та наборами ворогів і босів.

– Магазин: між рівнями гравець може купувати або продавати карти, що дозволяє адаптувати колоду до майбутніх викликів.

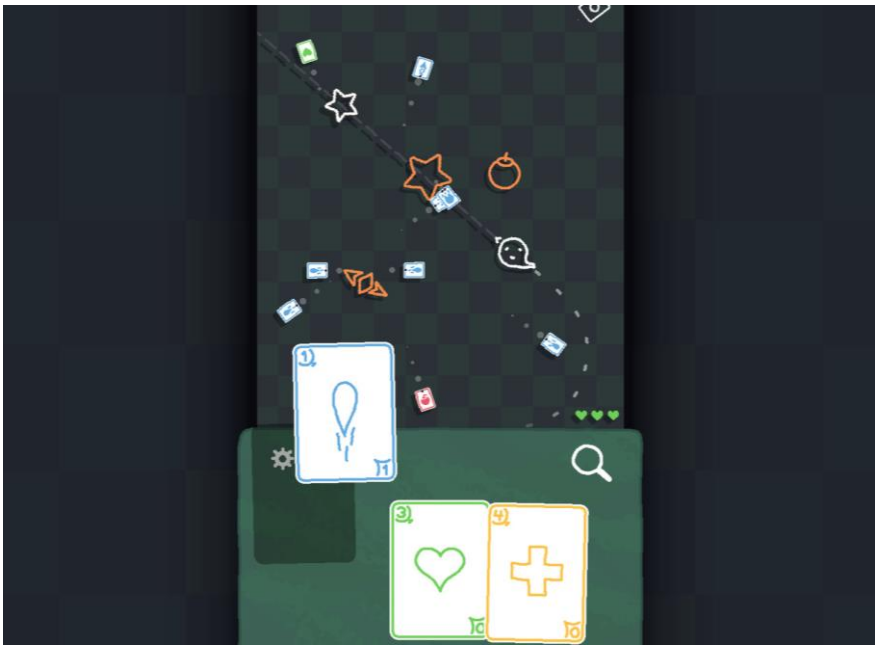


Рис. 1.2 Основний екран геймплею гри Heck Deck

1.5 Визначення вимог до програмного забезпечення гри

Оглянувши особливості жанру Action Roguelike, приклади ігор з подібними механіками, а також орієнтуючись на цільову аудиторію, було сформульовано перелік основних вимог до гри. Ці вимоги поділяються на функціональні та нефункціональні й визначають базові принципи, за якими реалізується проєкт.

Функціональні вимоги:

– Керування персонажем здійснюється за допомогою клавіатури з мишкою або контролера.

– Кількість ворогів збільшується з кожним новим раундом.

– Інвентар гравця формується випадковим чином з доступного пулу карт.

– HUD (інтерфейс користувача) відображає здоров’я, ресурси та вміст інвентарю.

– Гравець взаємодіє з об’єктами в грі (стіл та магазин) через натискання визначеної клавіші.

– Відображення пулу (колоди) карт під час паузи;

– Функція уповільнення часу при відсутності руху.

Нефункціональні вимоги:

– Усі текстові елементи інтерфейсу та повідомлень повинні бути англійською мовою.

– Візуальна постобробка повинна включати bloom, vignette та color adjustments.

– Візуальний стиль всіх сутностей повинен відповідати простій low-poly стилістиці.

2 ЗАСОБИ РЕАЛІЗАЦІЇ ВІДЕОГРИ

2.1 Ігровий рушій

Ігровий рушій – це програмне забезпечення, яке надає набір інструментів для створення відеоігор. Він забезпечує основну функціональність, необхідну для розробки ігор, включаючи графіку, фізику, звук, штучний інтелект, мережеву взаємодію та інтерфейс користувача. Основна мета ігрового рушія – спростити процес створення ігор, абстрагуючи складні технічні деталі та надаючи розробникам можливість зосередитися на творчих аспектах гри.

У цьому проєкті використовується Unity – один із найпопулярніших ігрових рушіїв, який підтримує розробку 2D та 3D ігор для різних платформ, включаючи ПК, мобільні пристрої та консолі, нижче наведені головні особливості:

1) Низький поріг входу.

Unity використовує популярну мову програмування C# та має величезну базу документації, посібників і відеоуроків, що дозволяє навіть новачкам швидко освоїти основи й почати створювати власні проєкти.

2) Універсальність.

Рушій не обмежується певним жанром або типом гри – на ньому можна створювати як 2D-платформери, так і повноцінні 3D-екшн або VR-проєкти.

3) Розвиток і підтримка.

Unity постійно оновлюється, додає нові функції та реагує на потреби спільноти, хоча іноді з деякою затримкою. Попри це, рушій залишається актуальним і технологічно гнучким.

4) Гнучкість модифікацій.

Система компонентів у Unity дозволяє легко додавати новий функціонал або змінювати вже існуючий, адаптуючи гру під власні потреби без необхідності переписувати все з нуля.

5) Маркетплейс Asset Store.

Unity має багатий магазин готових ресурсів: від 3D-моделей і анімацій до скриптів і повноцінних систем, що значно пришвидшує розробку.

6) Швидке ітерування.

Гру можна запускати прямо з редактора, швидко тестувати зміни та вносити правки без тривалих процесів збирання проєкту, що зручно для інді-розробки.

7) Зручний інтерфейс.

Інтерфейс Unity інтуїтивно зрозумілий – більшість елементів можна налаштовувати візуально, що зменшує кількість рутинної роботи з кодом.

8) Інтеграція сторонніх рішень.

Рушій підтримує безліч плагінів і SDK від інших сервісів (аналітика, реклама, хмарні сервіси тощо), що спрощує підключення додаткових функцій.

9) Платформна незалежність.

Unity можна встановити та використовувати на будь-якій сучасній операційній системі – Windows, macOS або Linux.

10) Монетизація.

Попри критику після оновлень політики ліцензування, умови для незалежних розробників залишаються досить вигідними, а безкоштовна версія рушія дозволяє працювати без вкладень на ранніх етапах [7].

2.2 Середовище розробки

Середовище розробки – це програмне забезпечення, яке надає розробникам інструменти для написання, редагування, компіляції та налагодження коду. Воно об'єднує текстовий редактор, компілятор, засоби налагодження та інші інструменти в одному інтерфейсі, що спрощує процес створення програмного забезпечення.

У цьому проєкті використовується JetBrains Rider — кросплатформенне інтегроване середовище розробки (IDE), спеціально розроблене для .NET-розробки, яке також має глибоку інтеграцію з Unity, що робить його зручним вибором для розробників ігор.

Особливості JetBrains Rider:

1. Підтримка Unity та інших ігрових рушіїв.

JetBrains Rider має спеціальні плагіни та інструменти для інтеграції з Unity, що дозволяє розробникам легко працювати з ігровими проєктами, використовуючи знайомі інструменти та функції.

2. Потужні інструменти для аналізу та рефакторингу коду.

Інтеграція з ReSharper надає розробникам потужні інструменти для аналізу коду, виявлення помилок та автоматичного виправлення, що сприяє підтримці високої якості коду.

3. Кросплатформеність.

Rider працює на Windows, macOS та Linux, що дозволяє розробникам працювати на зручній для них операційній системі без втрати функціональності.

4. Інтеграція з системами контролю версій.

Вбудована підтримка Git та інших систем контролю версій дозволяє легко керувати змінами в коді, об'єднувати гілки та відслідковувати історію змін безпосередньо з IDE.

5. Розширюваність та підтримка плагінів.

Rider підтримує широкий спектр плагінів, які розширюють функціональність IDE, дозволяючи налаштувати середовище розробки відповідно до потреб проєкту [8].

2.3 Мова програмування

Мова програмування – це засіб, за допомогою якого розробники можуть «спілкуватися» з комп’ютером, даючи йому чіткі інструкції для виконання дій. Програми, які ми створюємо, складаються з таких інструкцій, і саме мова

програмування дозволяє виразити логіку, обробку даних та управління поведінкою додатку в зручному для людини вигляді.

C# (сі-шарп) – це сучасна об’єктно-орієнтована мова програмування, створена компанією Microsoft у 2000 році як частина платформи .NET. Вона поєднує елементи зручності з потужністю мов високого рівня та дозволяє розробляти широкий спектр програм – від настільних застосунків до ігор, вебсайтів і мобільних додатків.

Однією з головних переваг C# є читабельний синтаксис, що нагадує мови сімейства C (наприклад, C++ або Java), але є простішим і більш послідовним. Завдяки цьому C# вважається хорошою мовою для вивчення, особливо в поєднанні з інструментами, які його підтримують – наприклад, Unity або Visual Studio.

C# активно використовується в розробці ігор – головним чином через рушій Unity, який підтримує цю мову на всіх рівнях: від керування об'єктами до реалізації ігрової логіки. Завдяки високій продуктивності та потужній обробці подій, C# дозволяє ефективно реалізовувати складні ігрові механіки, зберігаючи при цьому зручність написання коду.

Мова також підтримує об’єктно-орієнтоване програмування, що дозволяє будувати гнучку й модульну архітектуру проєктів. У C# також є підтримка таких сучасних можливостей, як асинхронне програмування, лямбда-вирази, LINQ-запити та багато інших інструментів, що спрощують роботу з даними та подіями.

У поєднанні з Unity, C# є одним із найпопулярніших виборів серед інді-розробників, оскільки дозволяє створювати якісні ігри навіть невеликими командами або соло [9].

2.4 Система контролю версій

Система контролю версій – це інструмент, який дає змогу зберігати історію змін у проєкті, відстежувати правки, повертатися до попередніх версій та зручно працювати в команді. Вона особливо важлива у процесі розробки

програмного забезпечення, де часто вносяться зміни до коду, і потрібно уникати втрати даних.

GitHub – це найпопулярніша у світі платформа для зберігання, обміну та керування версіями коду, побудована на системі Git. Вона дозволяє розробникам розміщувати свої проєкти в хмарі, співпрацювати з іншими учасниками команди, коментувати зміни та керувати процесом розробки з будь-якої точки світу [10].

Одна з головних переваг GitHub – зручна інтеграція з іншими інструментами, такими як середовище розробки (наприклад, JetBrains Rider) або платформи для тестування й автоматизації. Для індивідуального розробника GitHub також корисний як резервне сховище, де можна безпечно зберігати свій проєкт та при потребі відкотити зміни.

Сервіс підтримує візуальне порівняння змін у коді, роботу з гілками (branches) та спільну розробку через pull-запити, що робить його корисним навіть у невеликих командах або соло-проєктах з довготривалим циклом розробки.

2.5 Засоби створення графіки та візуального оформлення

Для розробки візуальної складової гри використовуються спеціалізовані програми, які дозволяють створювати та редагувати 2D і 3D-елементи. Вони дають змогу розробнику підготувати графічні ресурси, які надалі імпортуються до ігрового рушія та взаємодіють із геймплеєм.

2.5.1 GIMP

GIMP (GNU Image Manipulation Program) – це безкоштовний редактор растрової графіки з відкритим кодом. Він використовується для створення та обробки текстур, іконок, інтерфейсних елементів та іншої 2D-графіки, необхідної для гри [11].

GIMP підтримує роботу з шарами, прозорістю, масками та фільтрами, що робить його функціональним аналогом Adobe Photoshop для інді-розробника.

Програма дозволяє створювати графіку з нуля або редагувати вже наявні зображення, зберігаючи файли у форматах, сумісних з Unity (.png, .jpg тощо).

2.5.2 Blender

Blender – це безкоштовне програмне забезпечення з відкритим кодом для створення та редагування 3D-графіки. Воно широко використовується як в індустріальній, так і в інді-розробці ігор для моделювання, анімації та експорту 3D-об’єктів [12].

Blender дозволяє створювати low-poly моделі, які ідеально підходять для оптимізованої інді-гри. Крім цього, він має потужні інструменти для розгортання UV-розгорток, оформлення текстур та експорту моделей у форматах, що підтримуються Unity (наприклад, .fbx або .obj).

3 ПРОЄКТУВАННЯ ГРИ

3.1 Концепт

3.1.1 Короткий зміст гри

Однокористувацька екшн-гра з елементами roguelike, в якій гравець використовує комбінації гральних карт, щоб знищувати ворогів. Основна ідея полягає у поєднанні динамічного геймплею з тактичним плануванням, де кожне рішення має значення.

3.1.2 Огляд ігрового процесу

Гравець потрапляє в статичну замкнуту локацію, де постійно з’являються вороги. У центрі знаходиться Стіл із гральною колодою. Гравець може витягнути з нього декілька карт, якими можна кидатися у ворогів, але кожна нова спроба тягне за собою втрату одного очка здоров’я. Якщо кинута карта або кілька карт влучають у ворога то застрягаючи вони утворюють відповідну комбінацію, якщо ця послідовність відповідатиме визначеній то активується шкода – чим сильніша комбінація, тим більша шкода.

Після перемоги над ворогами гравець збирає монети та переходить до етапу модифікації вмісту колоди. Таким чином, гравець не просто реагує на ситуації в бою, а й поступово формує власну довгострокову стратегію. Це створює ефект персоналізації стилю гри. На рисунках 3.1, 3.2 та 3.3 зображено проєктування механік та геймплею гри у форматі ментальної карти.

Механіка slow motion (сповільнення часу) активується під час зупинки гравця, що дозволяє не втрачати контроль у напружених моментах і дає час для планування. Таким чином досягається баланс між динамікою та стратегічним мисленням. Для візуального зображення геймплею на рисунку 3.4 зображено діаграму варіантів використання.

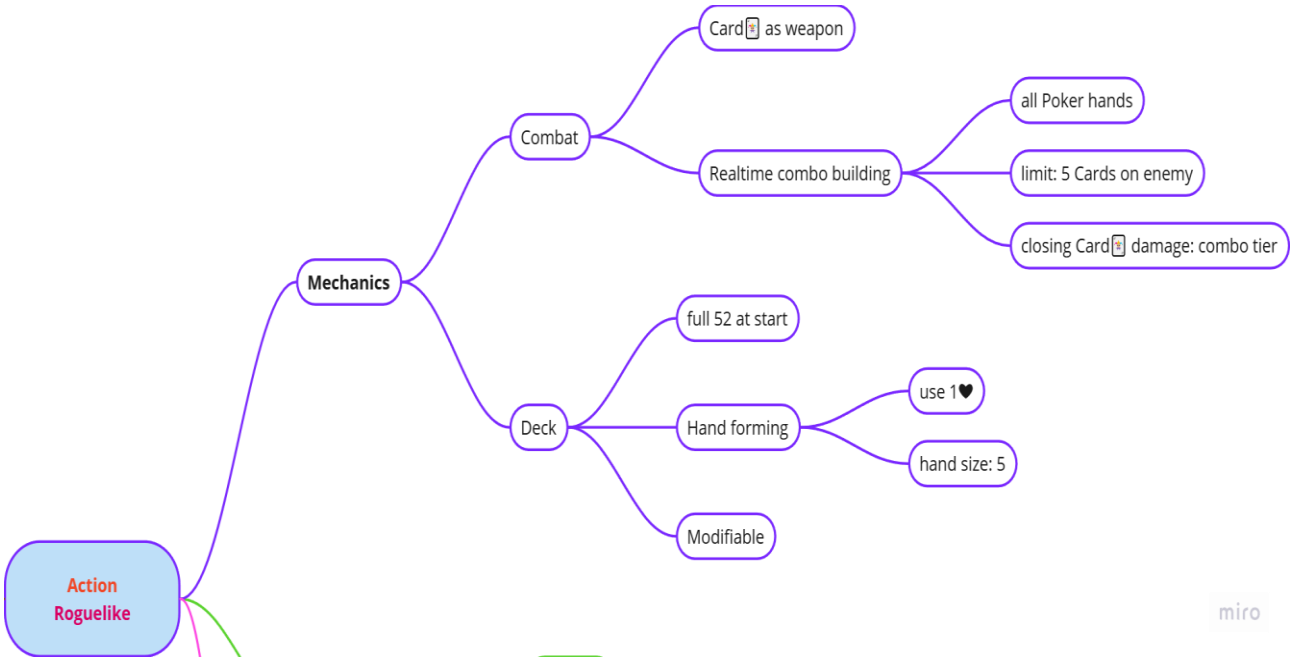


Рис. 3.1 Проєктування механік гри у вигляді Mindmap

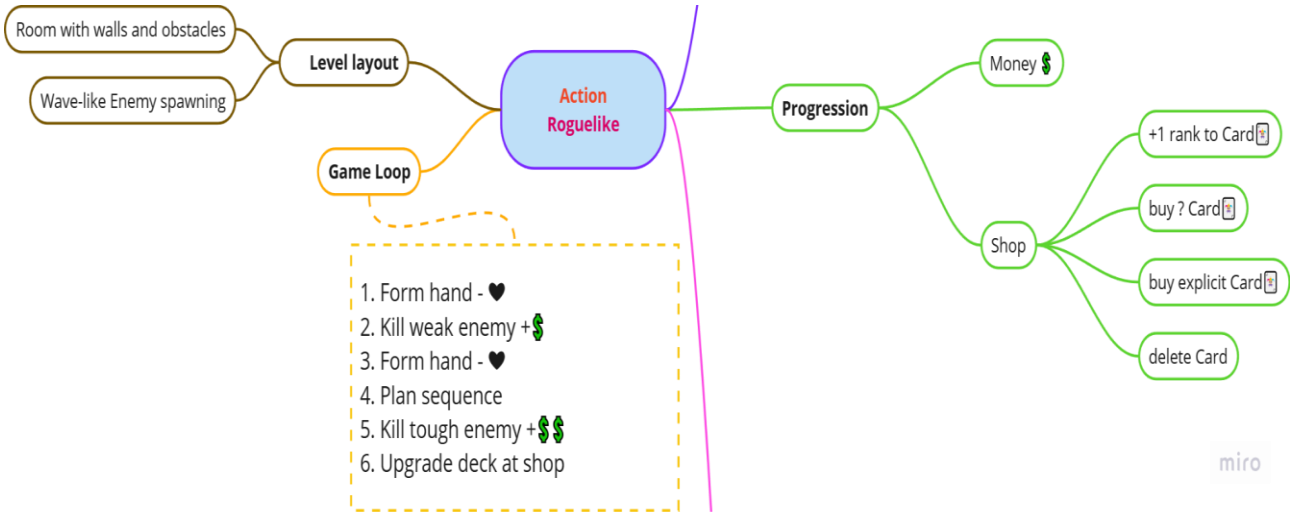


Рис. 3.2 Проєктування геймплею гри у вигляді Mindmap

Рис. 3.3 Проєктування сутностей гри у вигляді Mindmap

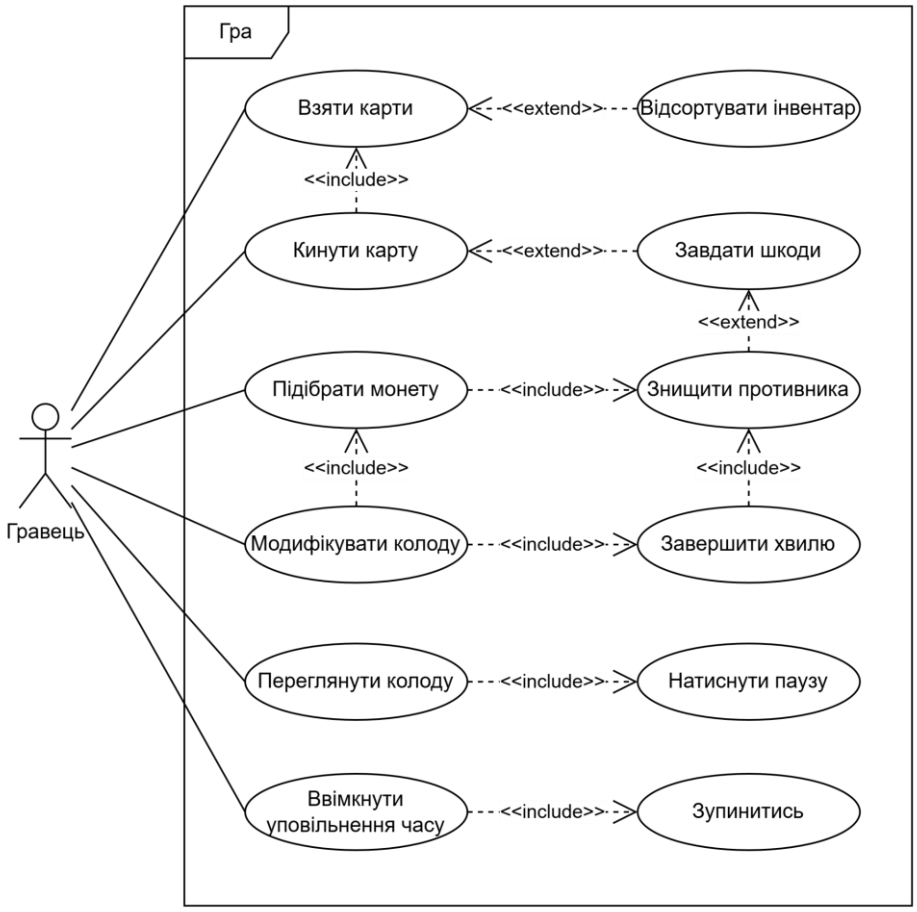


Рис. 3.4 Діаграма варіантів використання

3.2 Структура бойової системи

Бойова система в грі High Card поєднує елементи динамічного метання карт із перевіркою комбінацій, заснованих на покерних правилах. Цей підхід дозволяє надати гравцеві як тактичну глибину, так і відчуття екшну в реальному часі. Нижче розглянуто ключові елементи цієї системи.

3.2.1 Динаміка карт

У грі карти використовуються як основна метальна зброя. При натисканні кнопки атаки гравець кидає одну з карт у напрямку атаки. Рисунок 3.5 ілюструє, що кидки мають кутовий сектор розкиду, тобто карта не летить строго по лінії прицілу, а в межах визначеного діапазону – що додає динаміки та вимагає точнішого прицілювання замість швидкого “спаму”.

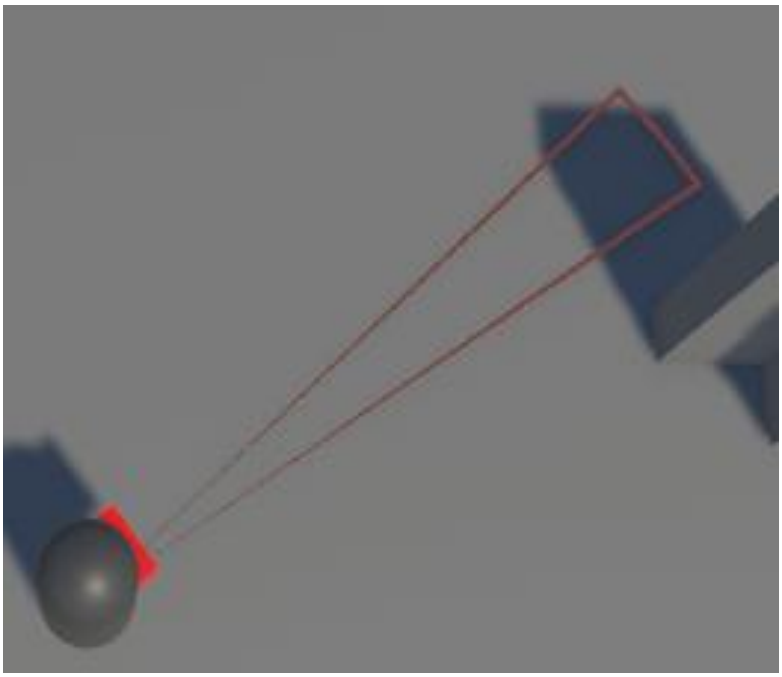


Рис. 3.5 Сектор розкиду (прототип)

Картки мають фізичні властивості: швидкість польоту, напрям, колізії з ворогами та середовищем. На рисунку 3.6 зображено, що після зіткнення з ворогом карта прилипає до нього, а її назва відображається над тілом ворога. Це

дозволяє гравцеві бачити, які карти вже потрапили у ворога, і спланувати подальші атаки.

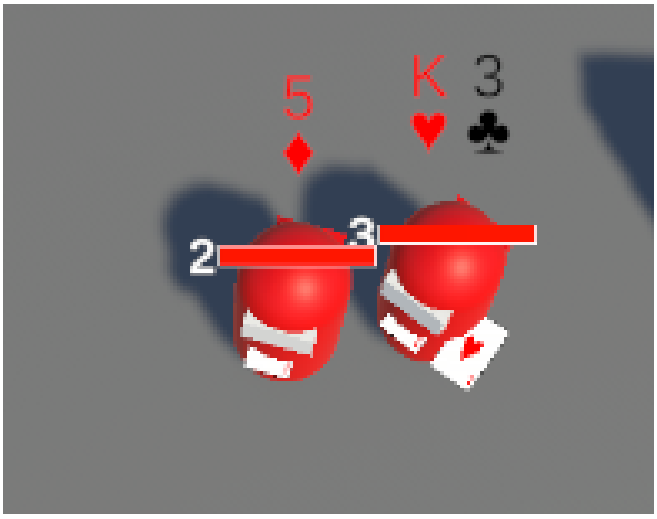


Рис. 3.6 Відображення значень карток над ворогами (прототип)

3.2.2 Логіка зарахування шкоди

Після кожного влучання у ворожий об’єкт, карта "прилипає" до нього. Після кожної атаки, відбувається перевірка комбінації, після чого ворогу завдається відповідна кількість шкоди та, як видно на рисунку 3.7, назва візуально зображується поряд з противником.

Принцип нарахування шкоди:

Якщо не утворена жодна комбінація – шкода не завдається (0).

Якщо утворена покерна комбінація – шкода визначається її рангом, наприклад:

– High Card (1);

– One Pair (2);

– Two Pair (3);

– Full House, Straight – значно більша кількість шкоди.

Усі комбінації перевіряються відповідно до стандартних покерних правил.

Рис. 3.7 Приклад отримання шкоди ворогом простою комбінацією (прототип)

3.3 Механіка колоди та її модифікацій

У грі колода виконує роль основного пулу карт, з якого формується бойовий інвентар гравця. У будь-який момент бою гравець може витратити одне очко здоров’я (HP), щоб отримати максимум 5 карт, випадково обраних із поточної колоди в свій інвентар. Одна й та сама карта не може бути обрана повторно. Нові карти не видаються, якщо в інвентарі вже є 5 карт. Таким чином, гравець має планувати свої дії так, щоб мінімізувати кількість доборів.

Після проходження рівня гравцю пропонується змінити свою колоду за допомогою одного з 4 типів модифікацій. Ці зміни дозволяють налаштовувати й покращувати шанс на отримання бажаних комбінацій у майбутньому.

Типи модифікацій:

1) Видалити карту

– дозволяє прибрати одну небажану карту з колоди, що збільшує шанс випадіння цінніших.

2) Додати нову карту

– додає до колоди звичайну карту.

3) Додати невідому карту

– значення карти стає відомим лише після покупки. Цей варіант створює додатковий елемент ризику.

4) Покращити карту

– дозволяє підвищити ранг певної карти, що збільшує її потенціал для створення високорівневих покерних комбінацій.

3.4 Поведінка противників

Оскільки в грі використовується один статичний рівень, поява ворогів реалізована у вигляді хвиль. З кожною новою хвилею складність поступово зростає: збільшується кількість ворогів, а також їх запас здоров’я. Це дозволяє підтримувати постійний ритм гри та поступово нарощувати виклик для гравця.

Логіка поведінки ворогів реалізована за допомогою машини кінцевих станів (FSM – Finite State Machine). Такий підхід робить структуру коду більш читабельною, а саму логіку – зручною для масштабування. Кожна нова поведінка або зміна може бути додана як окремий стан без порушення вже існуючої архітектури [13, 14]. На рисунку 3.8 зображено діаграму кінцевих станів противника в розроблюваній грі.

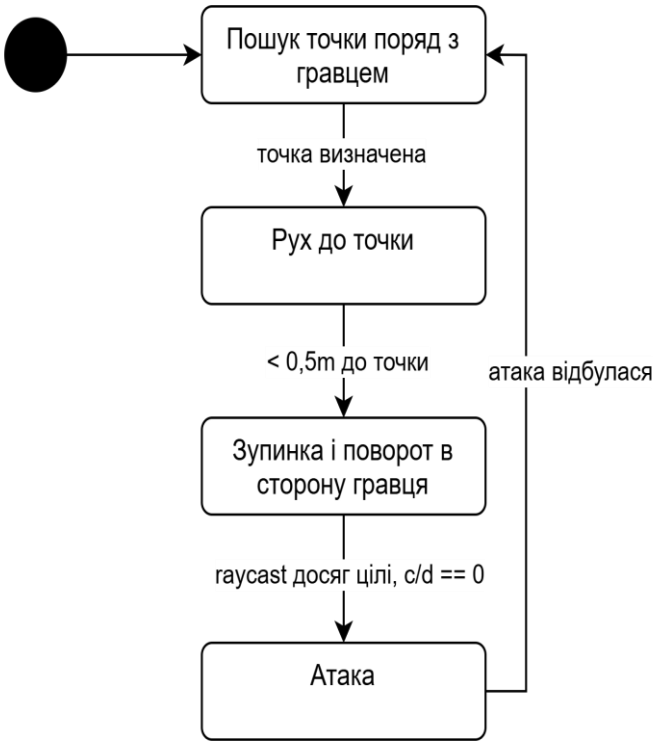


Рис. 3.8 Діаграма машини станів противника

Поведінка ворога складається з послідовних станів:

1) Пошук цільової точки поруч із гравцем – використовується спеціальний алгоритм, який обирає точку не прямо перед гравцем, а в певному радіусі навколо.
2) Рух до знайденої точки – ворог переміщується до неї, оминаючи інші об'єкти.
3) Зупинка та поворот у бік гравця – при досягненні точки ворог розвертається до гравця.
4) Постріл – якщо гравець у полі зору та перезарядка завершена, ворог виконує атаку.

Після виконання останнього стану цикл поведінки починається спочатку. Такий підхід формує менш передбачуваний патерн поведінки, на відміну від примітивного переслідування.

3.5 Елементи інтерфейсу та управління гравцем

Головне меню є стартовим екраном, з якого гравець починає гру. Воно містить кнопку “Почати нову гру” та “Вийти з гри”.

Під час активного геймплею гравець має доступ до HUD (heads-up display) – інформаційної панелі, яка відображає:

– Здоров’я (HP) гравця;

– Кількість зібраних монет;

– Номер поточної хвилі;

– Кількість живих ворогів на рівні.

Після завершення хвилі гравець може взаємодіяти з інтерфейсом магазину, який пропонує змінити колоду. У ньому відображаються:

– Три випадкові пропозиції модифікацій;

– Частина колоди, з якої гравець може обрати карту для взаємодії.

Цей інтерфейс є важливою частиною стратегії, бо дозволяє впливати на шанси випадіння потрібних карт у майбутніх боях.

Під час гри гравець може поставити гру на паузу, після чого відкривається меню паузи з:

– Кнопкою виходу до головного меню;

– Поточним станом колоди.

Управління гравцем реалізовано за допомогою системи InputActions [14], що дозволяє легко налаштувати управління як для клавіатури з мишею, так і для геймпада.

Управління за допомогою клавіатури та миші:

- WASD / стрілки — переміщення гравця;
- Shift — ривок;
- Рух миші — прицілювання;
- Затискання ЛКМ — звуження сектору розкиду;
- Відпускання ЛКМ — атака (кидок карти).

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ГРИ

4.1 Реалізація основної ігрової логіки

4.1.1 Рух гравця та прицілювання

Увесь ввід у грі реалізовано за допомогою Input System – офіційного пакета від Unity. Завдяки цьому інструменту можна легко створювати систему управління одразу для кількох платформ, не витрачаючи час на дублювання коду або окрему реалізацію під кожен пристрій [15].

Призначення клавіш та їх багатоплатформених відповідників здійснюється через Input Action Asset, який створюється в файловій системі Unity через контекстне меню (ПКМ → Create → Input Actions). У цьому файлі налаштовується мапа дій (Action Map), до якої додаються конкретні дії (Actions), що відповідають за певні типи вводу. Дію руху в Input Action Asset даного проєкту зображено на рисунку 4.1.

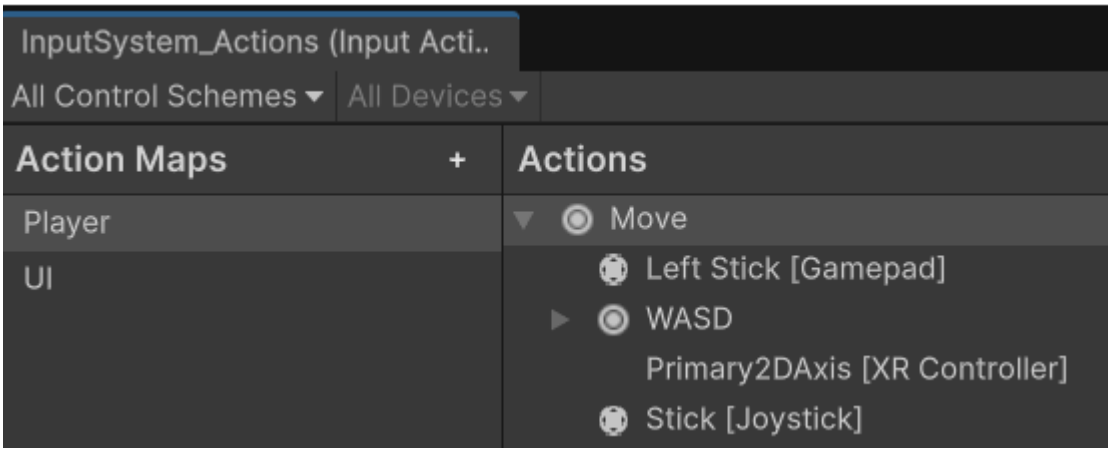


Рис. 4.1 Дія “Move” (рух) для різних платформ

Після цього у скрипті можна зчитувати дані з пристроїв введення незалежно від типу контролера – чи то клавіатура, миша, чи геймпад. Це дозволяє обробляти взаємодію гравця єдиним способом для всіх платформ, без створення окремих сценаріїв.

У даному випадку зчитується двовимірне векторне значення руху, яке множиться на коефіцієнт швидкості для розрахунку фінального вектора руху:

```
private InputAction _moveAction;
Vector2 direction = _moveAction.ReadValue<Vector2>();
Vector3 moveVelocity = new Vector3(direction.x, 0, direction.y).normalized * moveSpeed;
```

Оскільки рух реалізовано через фізичну систему, отримане значення присвоюється лінійній швидкості компонента Rigidbody об'єкта гравця:

```
_rb.linearVelocity = new Vector3(moveVelocity.x, -1, moveVelocity.z);
```

Обробка вводу для атаки здійснюється за аналогічним принципом, однак, оскільки гра реалізована в тривимірному просторі, виникає потреба визначити положення курсора миші на площині ігрового світу. Для цього використовується трасування променів – механізм, що дозволяє "прострілювати" сцену з камери у вказаному напрямку і дізнаватися, що було перетнуто.

Unity надає готовий інструмент для цього – метод Physics.Raycast, який виконує перевірку, чи перетинає промінь певний об’єкт у сцені [16, 17]. Нижче наведено реалізацію, яка дозволяє визначити точку прицілювання на землі:

```
Ray ray = _mainCamera.ScreenPointToRay(Input.mousePosition);
if (Physics.Raycast(ray, out RaycastHit raycastHit, 999f, groundLayerMask)){
    _targetPosition = raycastHit.point;
    _targetPosition.y = transform.position.y;
    _aimDirection = _targetPosition - transform.position;}
```

Цей підхід дозволяє точно визначати напрямок атаки в 3D-просторі залежно від положення курсора, навіть якщо поверхня ігрового рівня має складну форму. У результаті гравець завжди кидає картку в ту точку, на яку наведений курсор, що робить управління інтуїтивним і точним.

4.1.2 Камера та слідкування

У грі використовується вид від третьої особи з оглядом зверху (top-down). Такий ракурс дозволяє гравцеві контролювати ситуацію навколо персонажа,

бачити ворогів, карту і пересування в межах видимого екрану – що цілком достатньо для динамічної бойової системи, орієнтованої на реакцію і позиціонування.

Для реалізації камери було використано Cinemachine – офіційний пакет (package) від Unity, призначений для гнучкого управління камерою без потреби ручного кодування. Cinemachine підтримує слідування за об’єктами, зум, згладжування рухів камери, автоматичну орієнтацію, обмеження руху тощо [18, 19].

Для початку роботи у сцену додати об’єкт камери типу Cinemachine Camera (меню ієрархії → Cinemachine → Cinemachine Camera) та розмістити сам об’єкт на сцені в потрібному положенні (над гравцем).

Для визначення цілі у полі Tracking Target потрібно перетягнути об'єкт гравця (наприклад, Player) – це буде ціль, за якою камера буде слідкувати.

Далі необхідно додати компонент Cinemachine Follow, щоб камера почала рухатись. Як показано на рисунку 4.2, значення параметрів Position Damping компонента Cinemachine Follow встановлені на 1, що забезпечує м’яке і комфортне слідування камери за гравцем.

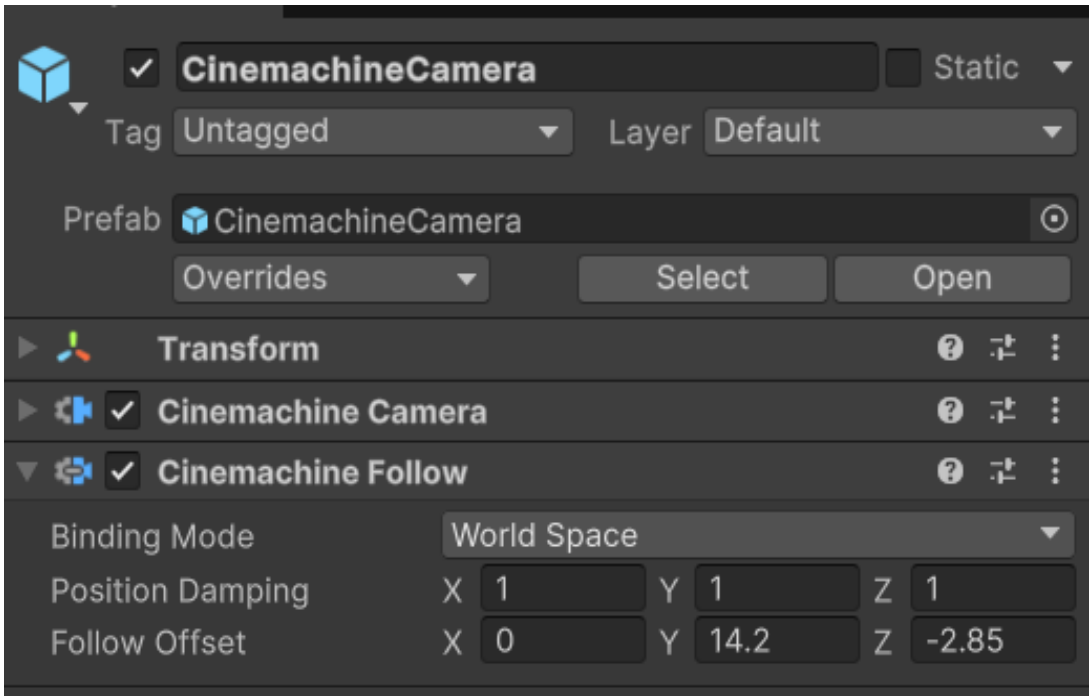


Рис. 4.2 Об’єкт з двома компонентами Cinemachine Camera і Follow

4.1.3 Механіка уповільнення часу

Оскільки частина ігрового процесу полягає в плануванні комбінацій, то з реальном плином часу та за великої кількості ворогів геймплей буде занадто важким. Щоб вирішити дану проблему було створено систему уповільнення часу, яка спрацьовує як тільки гравець перестає рухатись.

Для цього було створено окремий об’єкт на сцені який виконуватиме роль менеджера часу з спеціальним скриптом:

```
private float slowMoTimeScale = 0.1f;
public void DoTimeStop(){
    Time.timeScale = slowMoTimeScale;
    Time.fixedDeltaTime = Time.timeScale * 0.02f;}
public void DoTimeNormal(){
    Time.timeScale = 1;
    Time.fixedDeltaTime = Time.timeScale * 0.02f;}
```

Ці два методи змінюють властивість timeScale (масштаб, у якому протікає час) класу Time на значно менше число, щоб уповільнити і на 1 відповідно, щоб повернути звичний плин часу. Додатково потрібно змінити значення властивості fixedDeltaTime яке відповідає за те скільки часу проходить між кадрами методу FixedUpdate щоб вся логіка фізики була плавною зі зміною масштабу часу.

Далі можна звертатися до цього менеджера по посиланню з класу який відповідає за рух гравця:

```
if (direction == new Vector2(0, 0)) timeManager.DoTimeStop();
else timeManager.DoTimeNormal();
```

Таким чином коли гравець зупинятиметься, значення вектора direction (напрям руху), яке береться зі значень вводу дорівнюватиме (0, 0) і виконуватиметься функція уповільнення часу.

На рисунку 4.3 зображено основні створені класи, які використовуються як компоненти для гравця.

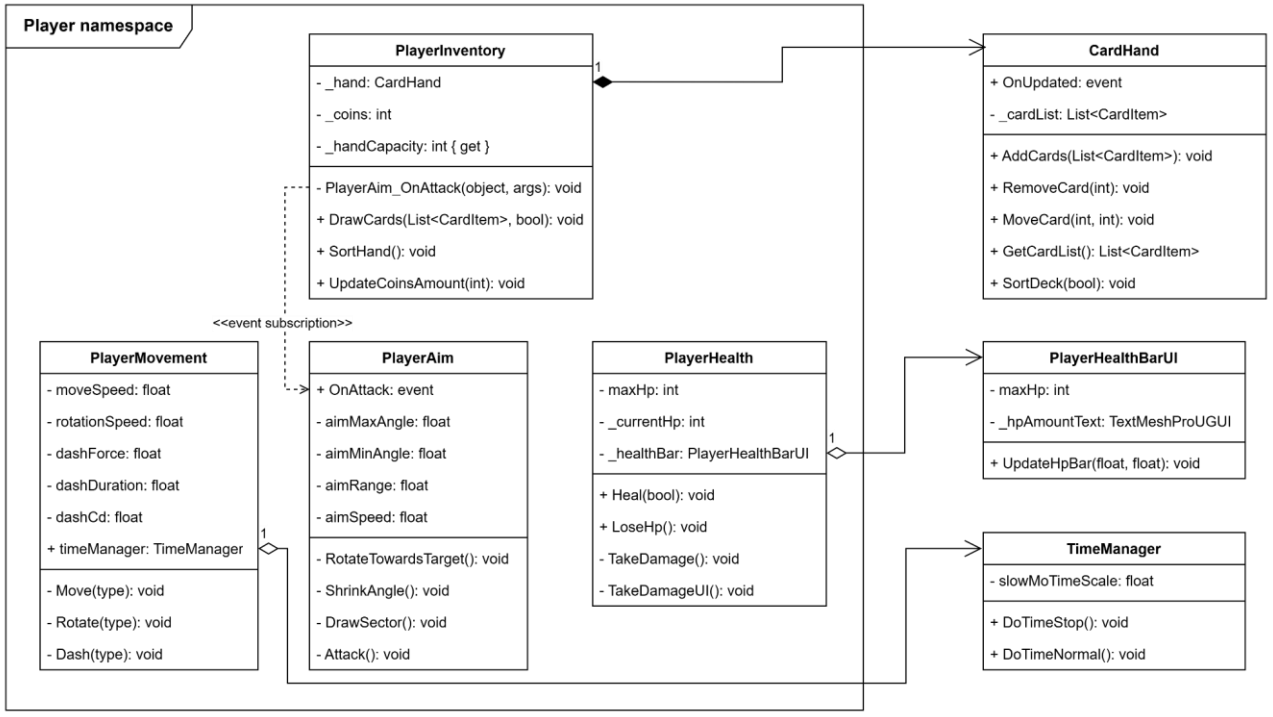


Рис. 4.3 Діаграма основних класів-компонентів гравця

4.2 Бойові механіки

4.2.1 Сектор розкиду та фізика польоту карт

Алгоритм атаки реалізовано наступним чином: після спрацювання дії "Attack" у системі введення визначається напрям атаки. Проте, щоб атака виконалась, потрібно відслідковувати момент відпускання відповідної клавіші, а не її натискання. Для цього використовується підписка на подію завершення дії:

_attackAction.canceled += OnAttackReleased;

Коли спрацьовує цей тригер, викликається метод атаки:

OnAttack?.Invoke(this, new AttackEventArgs{AimDirection = _aimDirection, SpreadAngle = _currentAngle});

У подію передається поточний напрям атаки та актуальне значення кута розкиду.

Механіка розкиду працює так: на початку затиску клавіші значення кута максимальне (у цьому випадку – 40°), і поступово зменшується впродовж часу:

float dt = Time.unscaledDeltaTime;

```
_currentAimSpeed += aimAcceleration * dt;
_currentAngle -= _currentAimSpeed * dt;
if (_currentAngle < aimMinAngle) _currentAngle = aimMinAngle;
```

Це означає, що якщо гравець трохи почекає, кидок буде більш точним. Такий підхід знижує ефективність бездумного спаму атакою і стимулює гравця діяти точніше та виважено.

Для візуалізації сектору розкиду в ігровому середовищі було використано компонент LineRenderer. Цей компонент дозволяє динамічно будувати лінії у реальному часі.

Використовується метод, який будує сектор у вигляді віяла. Спочатку визначається центр цього сектору, далі розраховується кут розкиду, який поділяється на рівні частини. Метод створює набір точок, що утворюють дугу – від лівого краю до правого, – і з’єднує їх між собою.

Як показано на рисунку 4.4, було виконано налаштування товщини лінії, щоб вона не залишалась однаковою по всій довжині.

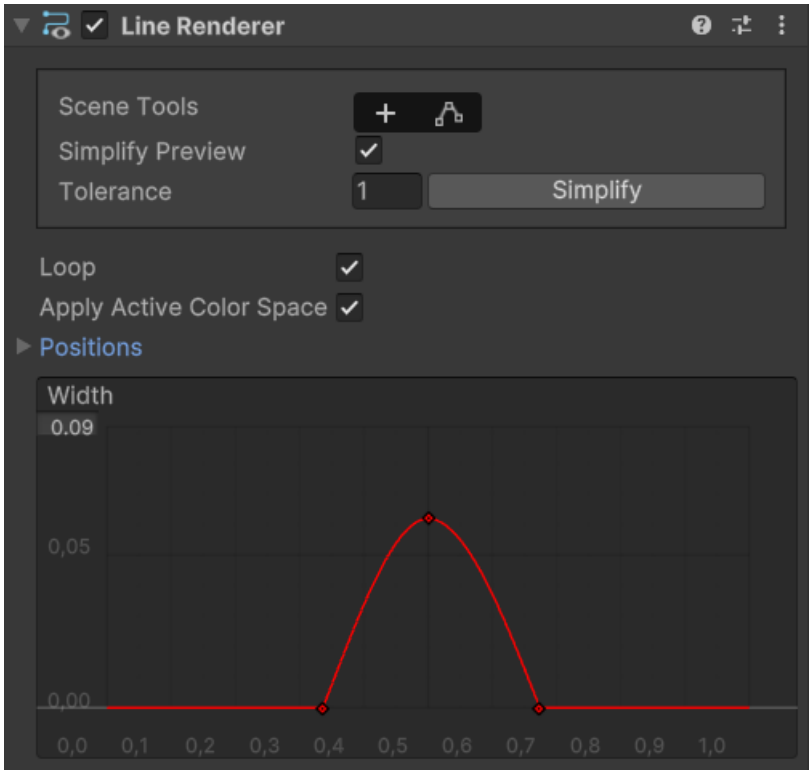


Рис. 4.4 Налаштування компоненту LineRenderer

Таким чином, на екрані з’являється візуальний сектор, який показує напрямок і ширину можливого розкиду прицілу – рисунки 4.5, 4.6.

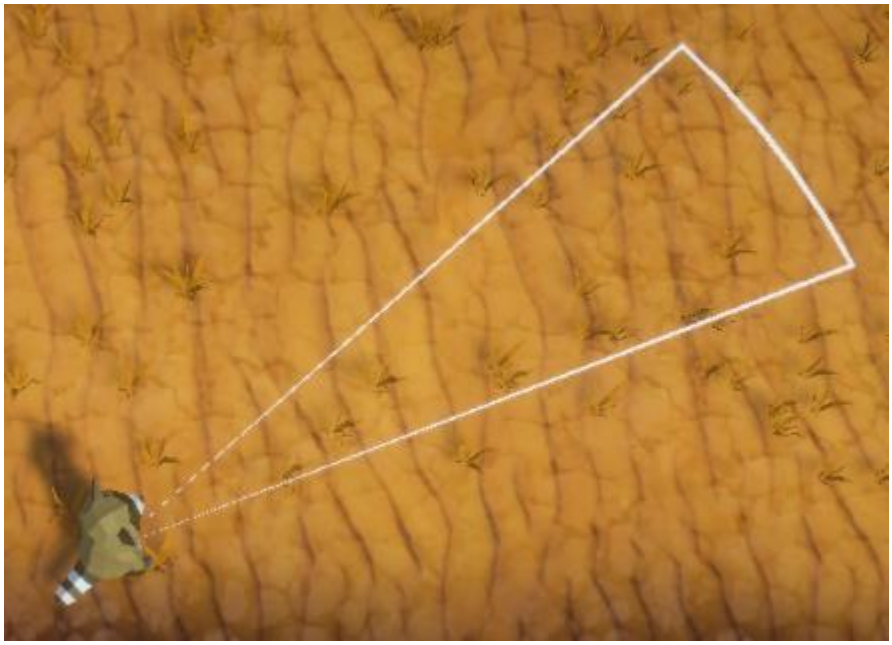


Рис. 4.5 Початковий сектор розкиду

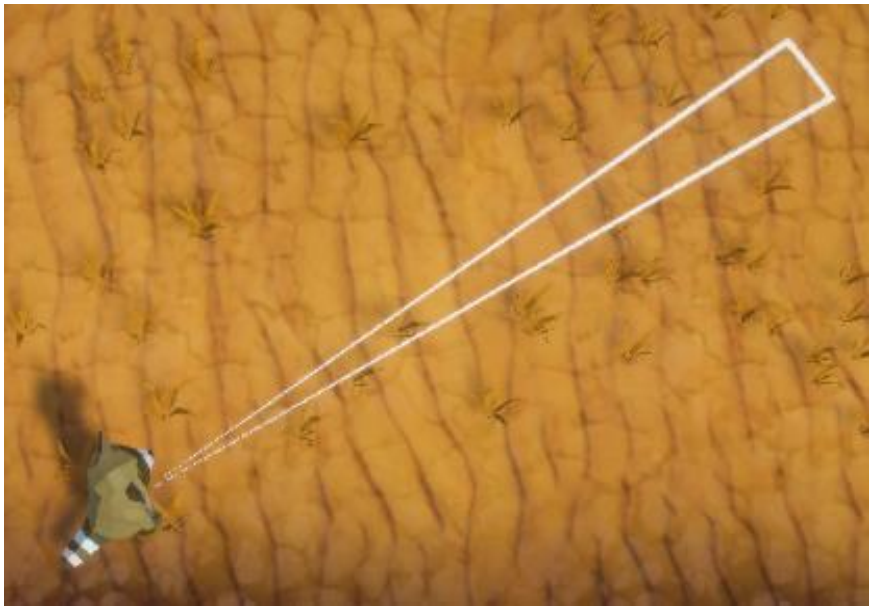


Рис. 4.6 Сектор розкиду після певного часу

Для запуску атаки створюється копія картки на заздалегідь визначеній позиції, після чого їй передаються параметри – зокрема напрям атаки та кут розкиду.

var cardWorld = CardWorld.SpawnCardWorld(cardSpawnPoint.position, _hand.GetCardList()[cardIndexToThrow]);

cardWorld.GetComponent<CardMovementPhysics>().Setup(e.AimDirection, e.SpreadAngle);

Усередині класу CardMovementPhysics випадковим чином обирається точний напрям руху в межах заданого сектора навколо основного напрямку, після чого фізичному компоненту картки (RigidBody) задається початкова швидкість, яка і запускає її в політ. Таким чином реалізується динамічна атака з варіативною траєкторією.

4.2.2 Підбір спрайтів і відображення

У грі кожна картка, незалежно від того, знаходиться вона в інтерфейсі чи у світі, має власний спрайт (візуальне зображення), який відповідає її значенню – рисунок 4.7. Для цього використано окремий об’єкт, що виконує роль менеджера спрайтів і надає доступ до зображень з будь-якого іншого класу.

Під час ініціалізації (у методі Awake) менеджер завантажує всі спрайти з папки Resources/Sprites/Cards за допомогою Resources.LoadAll<Sprite>(). Потім кожен спрайт додається до словника, де ключем виступає його назва:

_spriteDict[sprite.name] = sprite;

Коли потрібно отримати спрайт певної картки, інші об'єкти звертаються до цього менеджера через метод GetCardSprite, передаючи масть і ранг картки. Ці параметри поєднуються у ключ типу "Hearts 10", за яким відбувається пошук у словнику:

string key = $"{suit} {(int)rank}";

Якщо спрайт з таким ключем знайдено, він повертається. Якщо ні – виводиться попередження в консоль. Такий підхід дозволяє централізовано зберігати і легко призначати візуальні зображення для будь-якої картки у грі.

Рис. 4.7 Об’єкти карток зі своїми відповідними значенню спрайтами

4.2.3 Формування інвентарю

Коли гравець атакує, то з інвентарю видаляється обрана або остання карта, і її параметри передаються об’єкту в ігровому середовищі. Тому гравцеві необхідно якось створювати картки в своєму інвентарі. В грі інвентар формується випадково вибираючи 5 карток із загального пулу – колоди, після взаємодії з об’єктом “Стіл”. Систему інвентарю реалізовано 3-ма класами: CardHand – тимчасовий інвентар, Deck – колода та CardItem для збереження параметрів однієї картки. Обробку між ними здійснено в компоненті що розташований на гравці:

```
public void DrawCards(List<CardItem> cards, bool takeDamage = true){
int availableSlots = handCapacity - _hand.GetCardList().Count;
_hand.AddCards(cards.Take(availableSlots).ToList());
if(availableSlots > 0 && takeDamage) GetComponent<PlayerHealth>().LoseHp();}
```

Після взаємодії з об’єктом “Стіл” він викликає публічний метод який визначає кількість вільних слотів інвентарю і додає необхідну кількість карток у вигляді списку з CardItem. Також bool прапором можна ввімкнути отримання шкоди гравцем.

4.2.4 Логіка зарахування шкоди

У грі вся логіка перевірки комбінацій карт та нарахування шкоди реалізована на боці цілі – у спеціальному класі, що прикріплений до кожного противника. Кожного разу, коли картка влучає у ворога, її внутрішнє представлення (CardItem) додається до локального списку в цьому класі.

Використовуючи методи типу CheckForPair(), CheckForFullHouse() тощо, система по черзі перевіряє всі можливі комбінації. Реалізація цих перевірок ґрунтується на LINQ-запитах, які дозволяють легко аналізувати елементи списку, групувати їх за мастю або рангом та перевіряти умови для конкретних комбінацій.

Наприклад, у методі CheckForHighCard() перевіряється, чи є остання додана карта найстаршою серед усіх у списку. Якщо так – враховується комбінація "High Card" і присвоюється відповідний рівень шкоди:

```
bool isLastHighest = _cardItems
.Take(_cardItems.Count - 1)
.All(x => x.cardRank.GetHashCode() < _cardItems[^1].cardRank.GetHashCode());
```

Після успішного визначення комбінації система формує об’єкт події ComboEventArgs, у який передає:

– значення заподіяної шкоди (TotalDamage);

– назву комбінації (CurrentCombo);

– індекси карт, що формують комбінацію (для візуального виділення).

Далі викликається подія OnComboUpdated, на яку підписані інші скрипти які, як зображено на рисунку 4.8, керують UI-елементами, що відображають назву комбінації:

```
OnComboUpdated?.Invoke(this, new ComboEventArgs {
   TotalDamage = _totalDamage,
   CurrentCombo = _currentCombo,
   ComboIndexes = _comboIndexes});
```

Після цього всі службові змінні скидаються, і клас готовий до обробки нових влучань.

Такий підхід дозволяє чітко відокремити відповідальність за логіку перевірки комбінацій, а також легко масштабувати систему, додаючи нові типи комбінацій або модифікатори шкоди.

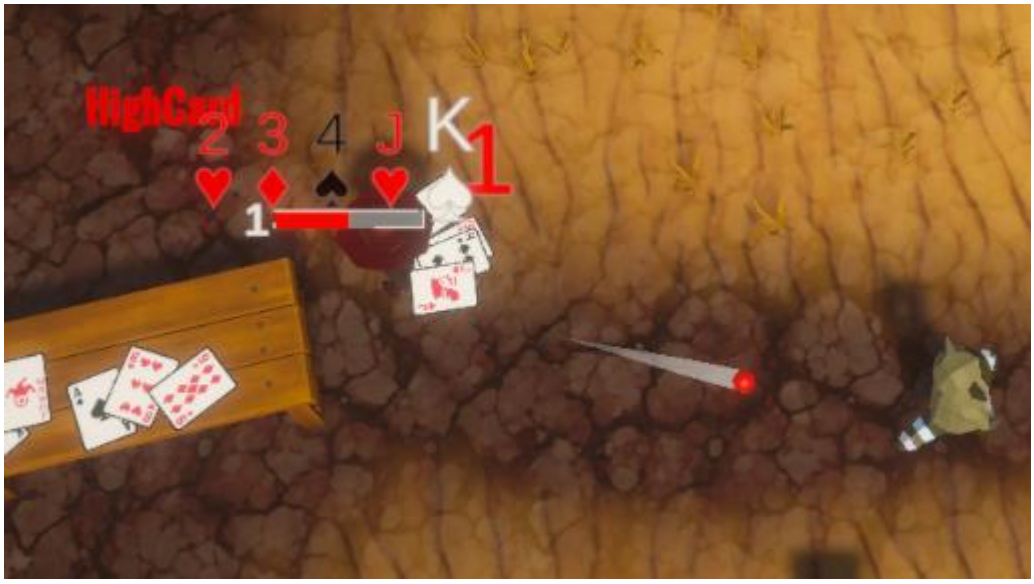


Рис. 4.8 Спрацювання комбінації High Card (старша карта) на цілі

4.3 Механіки прогресії та економіки

4.3.1 Монети

У грі реалізовано базову економічну систему, у якій основним ресурсом є монети, що випадають після знищення ворогів. Цей процес повністю контролюється з боку класу ворога, який відповідає за обробку шкоди та смерть об’єкта.

Після того, як кількість здоров’я (_currentHp) ворога зменшується до нуля або нижче, відбувається запуск логіки смерті. У першу чергу викликається подія OnHpLost, яка може бути використана іншими системами. Потім відбувається відписка від події оновлення комбінації, щоб уникнути помилок, якщо кілька карток влучають одночасно:

_cardComboHandler.OnComboUpdated -= TakeDamage;

Одразу після цього запускається метод SpawnCoins, який відповідає за створення монет у точці смерті ворога:

SpawnCoins(1);

Destroy(gameObject);

У самому методі SpawnCoins(int amount) виконується цикл на потрібну кількість монет, де кожна створюється через Instantiate() на поточній позиції ворога:

Instantiate(GameAssets.Instance.pfCoin, transform.position, Quaternion.identity);

4.3.2 Формування пропозицій магазину

Після завершення кожного бою гравцеві пропонуються різні варіанти модифікації колоди, які можна придбати в магазині. Вся логіка формування пропозицій реалізована в окремому об’єкті гри – так званому "візку-магазині", що активується у відповідній фазі гри.

Після активації об'єкта (OnEnable) запускається метод RefreshShop(), який очищує попередні елементи інтерфейсу, якщо вони були присутні, і готує місце для нових. Далі викликається метод SpawnItems(), який створює певну кількість слотів – згідно з заданим параметром slotsAmount.

Кожен слот заповнюється випадковим типом пропозиції, що обирається через Random.Range із чотирьох можливих варіантів:

– Додавання випадкової карти

– Додавання конкретної карти

– Видалення карти з колоди

– Підвищення рангу карти

Відповідно до вибраного типу викликається один із методів InstantiateSlot_*(), які створюють відповідний шаблон інтерфейсу, підключають обробники натискання миші та встановлюють ціну. Наприклад, метод InstantiateSlot_Delete() створює слот з іконкою видалення карти та виводить відповідну вартість:

```
var item = Instantiate(_deleteCardTemplate, _itemSlots);
item.GetComponentInChildren<ShopItem>().OnPointerClickEvent += OnDeleteCardClick;
```

Таким чином, при кожному вході в магазин гравець отримує новий, випадковий набір модифікацій, що дозволяє формувати власну колоду в залежності від обраної стратегії. Така система забезпечує варіативність ігрового процесу та підтримує повторюваність гри навіть при статичній структурі рівнів.

На рисунку 4.9 показано налаштування компоненту Shop Manager де кожному типу товару призначена своя ціна а також загальна кількість слотів пропозицій. На рисунку 4.10 в свою чергу показано створені пропозиції магазину відповідно налаштуванням.

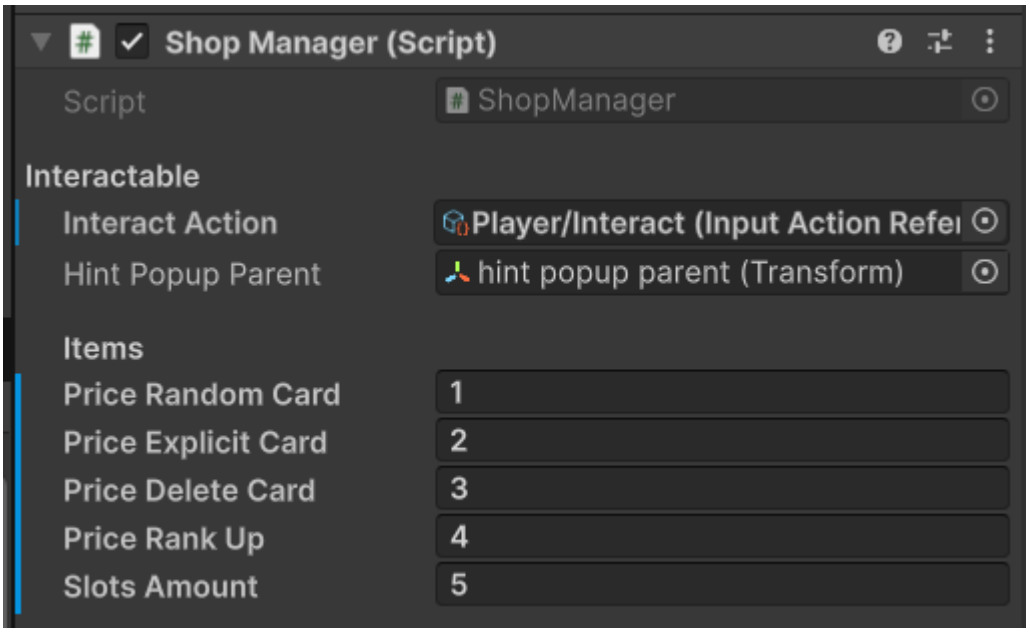


Рис. 4.9 Налаштування компоненту магазина в інспекторі

Рис. 4.10 Пропозиції магазину відповідно налаштуванням

4.4 Ігрова петля та вороги

4.4.1 GameLoopManager та етапи гри

Загальний цикл гри керується окремим об’єктом на сцені, на якому розміщено скрипт GameLoopManager. Він виконує роль менеджера ігрових фаз, контролюючи перехід між двома основними станами: бойовим етапом та магазином.

На старті (Start) запускається метод SetCombatPhase(), який активує бойову фазу: дозволяє взаємодію з "ігровим столом", виконує початкову роздачу карт та приховує інтерфейс магазину. Кількість хвиль при цьому відображається у відповідному UI елементі.

Після того, як гравець знищує всіх ворогів, спрацьовує обробник події AllEnemiesDied_Event(), який переводить гру в режим магазину (_isShopMode = true). У цьому режимі гравцеві недоступна взаємодія зі столом, натомість активується інтерфейс покупки карт та модифікацій. Додатково гравець отримує повне відновлення здоров’я:

player.GetComponent<PlayerHealth>().Heal(true);

Коли гравець готовий перейти до наступної хвилі, він виконує відповідну дію (nextAction), яка обробляється методом StartNextWave(). Цей метод перевіряє, чи активний режим магазину, і якщо так – повертає гру до бойової фази, оновлює номер хвилі та викликає спавн нових ворогів через компонент EnemySpawner.

_enemySpawner.SpawnNextWave();

4.4.2 Система хвиль

Противники в грі з’являються за допомогою системи хвиль, яка автоматично масштабує складність з кожним новим раундом. Її логіка реалізована у спеціальному класі, який відповідає за спавн ворогів та відстеження їх кількості.

Основним методом є SpawnEnemies(), який викликається під час початку нової хвилі. У цьому методі на випадкових точках спавну створюється потрібна кількість ворогів. Для кожного з них визначається об’єм здоров’я залежно від номера хвилі: останній ворог у списку отримує повне значення, рівне WaveCount - 1, а інші – випадкове менше значення:

```
if(i == maxEnemies-1) enemyHealth.AddHp(_waveCount-1);
else enemyHealth.AddHp(Random.Range(0, _waveCount-1));
```

Знищення кожного ворога обробляється через подію OnHpLost, яка підключається безпосередньо під час створення об’єкта. Після смерті ворога метод HandleEnemyDestroyed() зменшує лічильник активних супротивників. Коли їх не залишається, викликається подія OnAllEnemiesDied, яка сигналізує GameLoopManager'у про перехід до наступного етапу гри.

Кількість ворогів у хвилі (maxEnemies) поступово зростає: кожні три раунди до неї додається +1 ворог за допомогою методу AddToMaxEnemies():

```
if(_waveCount % 3 == 0) AddToMaxEnemies(1);
```

Таким чином, із кожною хвилею гра стає складнішою не лише за рахунок кількості супротивників, а й через збільшення їхнього здоров’я, що створює прогресивну криву складності та стимулює гравця вдосконалювати стратегію і колоду. Лічильник ворогів постійно оновлюється на екрані через метод UpdateEnemyCounterTMP(), що дозволяє гравцеві орієнтуватися в поточній ситуації на полі бою – рисунок 4.11.

Рис. 4.11 Значення лічильника ворогів на 6-й хвилі

4.4.3 FSM-логіка ворогів

Щоб зробити поведінку супротивників більш непередбачуваною і унікальною, було використано патерн машини кінцевих станів (FSM – Finite State Machine). Цей підхід дозволяє розділити логіку ворога на чіткі стани з умовами переходу між ними, що підвищує читабельність коду та спрощує розширення в майбутньому.

У кожного ворога, зокрема у класі EnemyShooter, є власна FSM-система, яка керує трьома основними станами:

1) Пошук точки навколо гравця (SearchForPointNearTarget) – ворог обирає випадкову точку в певному радіусі від гравця, щоб обійти його або зайняти вигідну позицію для стрільби – рисунок 4.12.
2) Рух до обраної точки (MoveToPoint) – за допомогою NavMeshAgent ворог прямує до визначеної позиції, використовуючи анімації переміщення.
3) Постріл (Shoot) – після досягнення точки супротивник обертається в бік гравця і виконує постріл.

Переходи між станами задаються через умови у вигляді функцій. Наприклад, ворог переходить із пошуку точки до руху, коли знайдено дійсну цільову позицію, і з руху – до стрільби, коли достатньо наблизився до точки:

```
At(search, moveToTarget, () => DistanceFromTargetToDest != 0);
At(moveToTarget, shoot, ReachedTarget);
```

Після виконання пострілу ворог повертається до стану пошуку точки, що створює цикл поведінки, де ворог постійно змінює позицію та періодично стріляє.

Метод Shoot() виконує трасування (Raycast) в напрямку погляду ворога і, якщо гравець знаходиться в полі зору, створює снаряд (EnemyProjectile), який летить точно в обрану точку:

```
EnemyProjectile bullet = Instantiate(...);
bullet.SetDirection(hit.point - gunPoint.position);
```

Окремо враховується динаміка дистанції до точки: радіус цільової зони поступово збільшується, якщо ворог знаходиться в межах зони стрільби, щоб

уникнути ситуацій, коли ворог безкінечно намагається дістатись до недоступної точки.

Рис. 4.12 Візуалізація логіки пошуку точки

4.5 Візуальне оформлення та інтерфейс

4.5.1 Інтерфейс користувача

У грі інтерфейс користувача побудований на базі стандартних UI-компонентів Unity, які розміщуються всередині об’єкта типу Canvas. Canvas – це спеціальний контейнер, що дозволяє відображати UI-елементи у 2D-просторі поверх 3D-сцени або безпосередньо в самому 3D-середовищі (через World Space). Уся взаємодія між користувачем та ігровим інтерфейсом здійснюється саме через Canvas [20].

HUD (Head-Up Display або прозорий дисплей), як показано на рисунку 4.13, містить такі елементи: шкалу здоров’я, що реалізована через компонент Image, та числові значення монет, хвилі й кількості ворогів, які представлені через TextMeshPro. Ці елементи закріплені у Canvas, що працює в режимі Screen Space - Overlay, щоб завжди залишатися видимими поверх ігрової сцени.

Рис. 4.13 Елементи HUD

Головне меню гри реалізовано як окрема сцена, що завантажується першою при запуску гри. У ній знаходяться дві основні кнопки (Button UI): “Нова гра” та “Вихід” – рисунок 4.14. На відміну від інтерфейсу на сцені геймплею, Canvas у головному меню, в якому знаходяться кнопки, працює в режимі World Space, тобто прив’язаний до простору сцени, що дозволяє розміщати UI-елементи у самому ігровому світі.

Рис. 4.14 Елемент UI (Button), який відображається в режимі World Space

Меню паузи також розташовується в Canvas і активується під час натискання відповідної клавіші. У цьому меню відображається поточна колода гравця та кнопка виходу до головного меню. Цей інтерфейс дозволяє гравцеві оцінити свій прогрес та змінити рішення, не виходячи з гри.

Відображення інвентарю та взаємодія з картками реалізовані за допомогою спеціального скрипту, який динамічно створює в Canvas копії карток, що є UI-елементами з власними скриптами. Ці елементи реалізують інтерфейси IBeginDragHandler, IDragHandler, IEndDragHandler та IPointerClickHandler, що

дозволяє гравцеві перетягувати картки, обирати їх мишею та взаємодіяти з ними у звичний спосіб – рисунок 4.15.

Рис. 4.15 Перетягування UI елементу за допомогою реалізації інтерфейсів

4.5.2 Візуальне наповнення сцени

Для створення візуального наповнення гри були використані вже готові 3D-моделі та текстури, завантажені з відкритих електронних ресурсів. Основними джерелами стали Unity Asset Store, open3dmodel, CGTrader та OpenGameArt. Через ці платформи було підібрано моделі для оточення, персонажів, рослинності, а також поверхні сцени. Пошук та підбір шрифтів для інтерфейсу здійснювався через ресурс 1001 Fonts.

Частина моделей потребувала комбінування чи редагування – для цього використовувалася програма Blender. Наприклад, модель супротивника було об'єднано з окремою моделлю зброї, після чого імпортовано до Unity як один об’єкт.

У результаті на основі зібраних і адаптованих моделей було створено замкнуту ігрову сцену, стилізовану під сетинг дикого заходу, що відповідає загальній стилістиці гри й підтримує атмосферу обраного жанру.

Для анімації персонажів використовувалися готові рухи з сайту Mixamo, з можливістю подальшого редагування. Наприклад, для гравця в Unity було додатково налаштовано поворот голови відповідно до напрямку прицілювання.

Для покращення візуальної атмосфери сцени у грі застосовано post-processing ефекти за допомогою компонента Global Volume. У ньому були активовані такі ефекти, як:

Bloom – додає м’яке сяйво до яскравих об'єктів;

Vignette – затемнює краї екрану для зосередження уваги на центрі;

Color Adjustments – корекція контрасту та насиченості кольору;

White Balance – регулювання температури зображення для кращої кольорової гармонії.

Окрім того, в налаштуваннях Lighting (освітлення) було увімкнено Fog (туман), який додає глибини сцені та створює легкий шар атмосфери, особливо помітний на далеких об'єктах. Відмінності у вигляді сцени до та після постобробки зображено на рисунках 4.16 та 4.17.

Рис. 4.16 Вигляд сцени до постобробки

Рис. 4.17 Вигляд сцени після постобробки

4.6 Тестування

4.6.1 Ручне тестування ігрового циклу

Для перевірки коректної роботи основних фаз гри було проведено ручне тестування повного ігрового циклу, що включає чергування бойових етапів, переходи до магазину та початок нових хвиль. Мета тестування – виявити помилки у переходах між станами гри, відображенні інтерфейсу та коректній взаємодії між системами.

Етапи тестування наведено в таблиці 4.1:

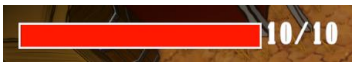
Таблиця 4.1

Тестові сценарії етапів гри

№ етапу	Назва етапу	Очікуваний результат	Фактичний результат
1	Запуск гри та початок першої хвилі	HUD відображає: шкалу здоров’я, кількість монет, лічильник хвиль та ворогів	
		Спавн противника	
		Можливість взаємодії зі столом (спливаюча підказка “E”)	

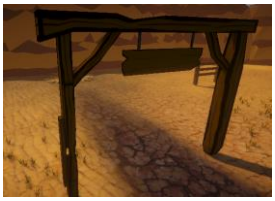
Продовження таблиці 4.1

Тестові сценарії етапів гри

№ етапу	Назва етапу	Очікуваний результат	Фактичний результат
2	Знищення всіх ворогів	Спавн об’єкту магазину (синій візок)	
		Відключення взаємодії з бойовим столом	
		Повне відновлення здоров’я	
3	Виконання покупки в магазині	Відображення 5-ти карт-пропозицій	
		Обрана карта зникає з пропозицій (4-та)	
		Списується відповідна кількість монет	

Продовження таблиці 4.1

Тестові сценарії етапів гри

№ етапу	Назва етапу	Очікуваний результат	Фактичний результат
4	Початок наступної хвилі	Об’єкт магазину зникає	
		Повертається можливість взаємодіяти зі столом	
		Оновлюється лічильник хвилі	

Результати:

Жодних критичних помилок під час переходів між фазами не виявлено. Всі системи працюють узгоджено, інтерфейси оновлюються вчасно, хвилі прогресивно ускладнюються згідно з логікою.

4.6.2 Тестування логіки ворогів через Debug Tools

Для перевірки коректності роботи логіки поведінки ворогів, яка реалізована через FSM (Finite State Machine), використовувався простий та ефективний метод – виведення повідомлень у консоль за допомогою функції Debug.Log().

Кожен окремий стан супротивника (пошук точки, рух, атака) має власний метод Tick(), у якому розміщується відповідний виклик Debug.Log(). Це дозволяє в реальному часі бачити в консолі Unity, в якому стані знаходиться ворог у даний момент – рисунок 4.18.

Таким чином, можна впевнено відслідковувати послідовність переходів між станами та переконатися у коректності умов.

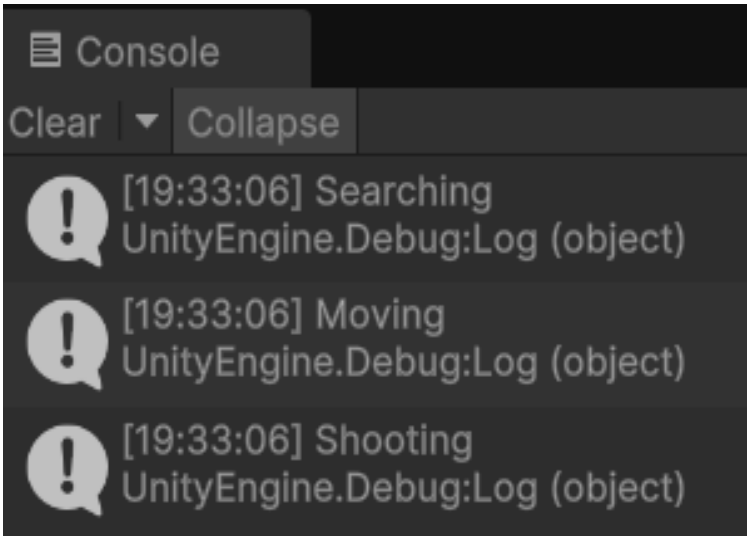


Рис. 4.18 Виведення в консоль роботи всіх станів

Як показано на рисунку 4.18 послідовність спрацювання станів виконується в правильному порядку.

ВИСНОВКИ

1. Проведено аналіз особливостей жанру Roguelike, що дозволило визначити ключові механіки, які впливають на повторюваність досвіду: варіативність предметів, зростаючу складність, перманентність прогресу та керовану випадковість.

2. Розглянуто приклади ігор зі схожими жанрами, зокрема Balatro та HeckDeck, що стали джерелом ідеї концепту для проєкту. Було виокремлено ідею поєднання динамічного геймплею з елементами стратегічного планування через механіку уповільнення часу.

3. Сформовано функціональні та нефункціональні вимоги, що лягли в основу архітектури гри. Основні функціональні вимоги стосувалися реалізації випадкової вибірки карт в інвентарі та випадковості пропозицій у магазині.

4. Проведено аналіз засобів розробки програмного забезпечення, в результаті чого обрано рушій Unity, середовище JetBrains Rider, Blender для роботи з моделями та інші допоміжні інструменти.

5. Створено концепцію гри, у якій поєднано кілька на перший погляд несумісних механік – варіативність гральних карт, їх стратегічне використання, та динамічні бої. Гравець формує комбінації на основі поточних умов, при цьому змінює ймовірність випадіння карт через систему модифікацій, що створює унікальний досвід під час кожного проходження.

6. Здійснено проєктування архітектури гри, зокрема реалізовано базові класи для гравця, механіку формування інвентарю, та систему поведінки ворогів на основі FSM (машини кінцевих станів).

7. Реалізовано ігровий прототип з використанням рушія Unity, що дозволило протестувати основні ідеї, механіки та оцінити загальний цикл гри.

8. Проведено ручне тестування ключових механік, зокрема тестування циклу бою та переходів між фазами, а також логіки поведінки супротивників через Debug.Log, що підтвердило стабільність роботи систем.

Робота пройшла апробацію на двох наукових конференціях, за результатами апробації опубліковано тези доповідей:

1) Коваль О.С., Дібрівний О.А. Інтеграція roguelike елементів як засобу підвищення реіграбельності. Застосування програмного забезпечення в інформаційно-комунікаційних технологіях : Матеріали VI Всеукраїнської науково-технічної конференції, Київ, 24 квіт. 2025. ДУІКТ. С. 261–263.
2) Коваль О.С., Дібрівний О.А. Забезпечення безпеки даних у відеогрі. Інформаційні технології – 2025 : Матеріали XII Всеукраїнської науково-практичної конференції молодих учених, Київ, 15 трав. 2025. КСУ ім. Бориса Грінченка. С. 284–285.

ПЕРЕЛІК ПОСИЛАНЬ

1. What is a Roguelike? The Beginner's Guide. *Lifewire*. URL: https://www.lifewire.com/what-are-roguelikes-4117411.
2. Berlin Interpretation. *Roguebasin*. URL: https://www.roguebasin.com/index.php/Berlin_Interpretation.
3. 5 Elements of Roguelike Video Games. *Masterclass*. URL: https://www.masterclass.com/articles/roguelike-game-guide.
4. Why replayability is better than length in indie game development. *Wardrome*. URL: https://wardrome.com/why-replayability-is-better-than-length-in-indie-game-development/.
5. Balatro. *Steam*. URL: https://store.steampowered.com/app/2379780/Balatro/.
6. Heck Deck. *Steam*. URL: https://store.steampowered.com/app/1606210/Heck_Deck/.
7. Unity проти Unreal Engine. *Gamedev DOU*. URL: https://gamedev.dou.ua/articles/unity-or-unreal-engine.
8. Get started. *JetBrains Rider*. URL: https://www.jetbrains.com/help/rider/Introduction.html.
9. C#. *Microsoft*. URL: https://dotnet.microsoft.com/en-us/languages/csharp.
10. Why GitHub. *GitHub*. URL: https://github.com/why-github.
11. *GIMP*. URL: https://gimp.org.
12. *Blender*. URL: https://blender.org.
13. Lecture 03 – Finite State Machines. *Edirlei*. URL: https://edirlei.com/aulas/game-ai/GAME_AI_Lecture_03_Finite_State_Machines_2018.pdf.
14. FSM tutorial. *Toptal*. URL: https://www.toptal.com/unity/unity-ai-development-finite-state-machine-tutorial.

15. Input System. *Unity3D manual*. URL: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/Actions.html.
16. Physics.Raycast. *Unity3D Documentation*. URL: https://docs.unity3d.com/6000.1/Documentation/ScriptReference/Physics.Raycast.html.
17. Raycast In Unity Made Easy. *GameDevBeginner*. URL: https://gamedevbeginner.com/raycasts-in-unity-made-easy/.
18. Cinemachine. *Unity3D Packages*. URL: https://docs.unity3d.com/Packages/com.unity.cinemachine@3.1/manual/index.html.
19. Cinemachine. *Documentation*. URL: https://documentation.help/Cinemachine/documentation.pdf.
20. Canvas. *Unity3D Packages*. URL: https://docs.unity3d.com/Packages/com.unity.ugui@3.0/manual/UICanvas.html.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка відеогри в жанрі “Мандрівний бойовик”

Виконав студент 4 курсу
групи ПД-44
Коваль Олександр Сергійович
Керівник роботи
доктор філософії (PhD), доцент кафедри ІПЗ Дібрівний Олесь Андрійович

Київ – 2025

ПОРІВНЯЛЬНА ТАБЛИЦЯ

Параметр	Balatro	Heck Deck	***High Card***
Базова механіка	Формування покерних комбінацій у колоді	Кидання карт у ворогів з різними ефектами	Комбінації карт у бою, що формуються через кидки у ворогів
Тип боїв	Покрокова, спокійна	Динамічна з паузами	Динамічна з сповільненням
Рівень стратегічності	Середній, базується на колодобудуванні	Мінімальний, рішення здебільшого ситуативні	Більше стратегічного планування через керування колодою та розподіл карт
Випадковість	Висока – багато випадкових подій та карт	Помірна – переважно у магазині	Помірна – випадкові модифікації, контрольована побудова колоди
Особливості	Покерна тема, нестандартні синергії	Колодоскладальна динамічна	Поєднання динамічного бою та планування

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

– Керування персонажем здійснюється за допомогою клавіатури з мишкою або контролера.

– Кількість ворогів збільшується з кожним новим раундом.

– Інвентар гравця формується випадковим чином з доступного пулу карт.

– HUD (інтерфейс користувача) відображає здоров'я, ресурси та вміст інвентарю.

– Взаємодія гравця з об'єктами в грі (стіл та магазин) через натискання відповідної клавіші.

– Відображення пулу (колоди) карт під час паузи;

– Функція уповільнення часу при відсутності руху.

Нефункціональні вимоги:

– Усі текстові елементи інтерфейсу та повідомлень повинні бути англійською мовою.

– Візуальна постобробка повинна включати bloom, vignette та color adjustments.

– Візуальний стиль всіх моделей повинен відповідати low-poly стилістиці (<10 000 полігонів)

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Unity

JetBrains Rider

GitHub Desktop

GIMP

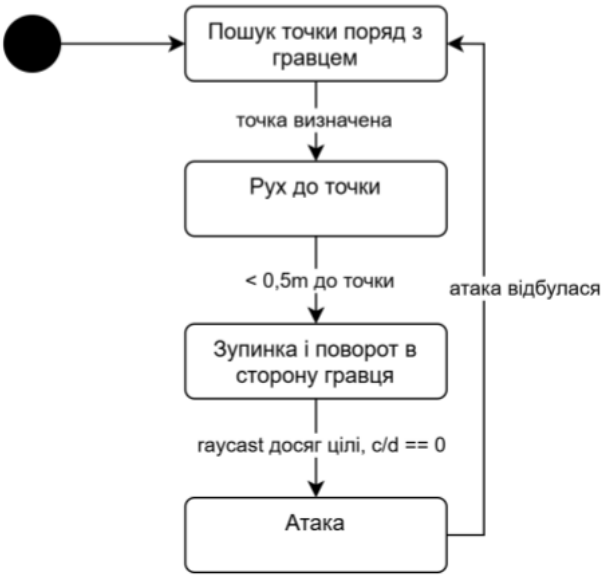
Blender

6

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ

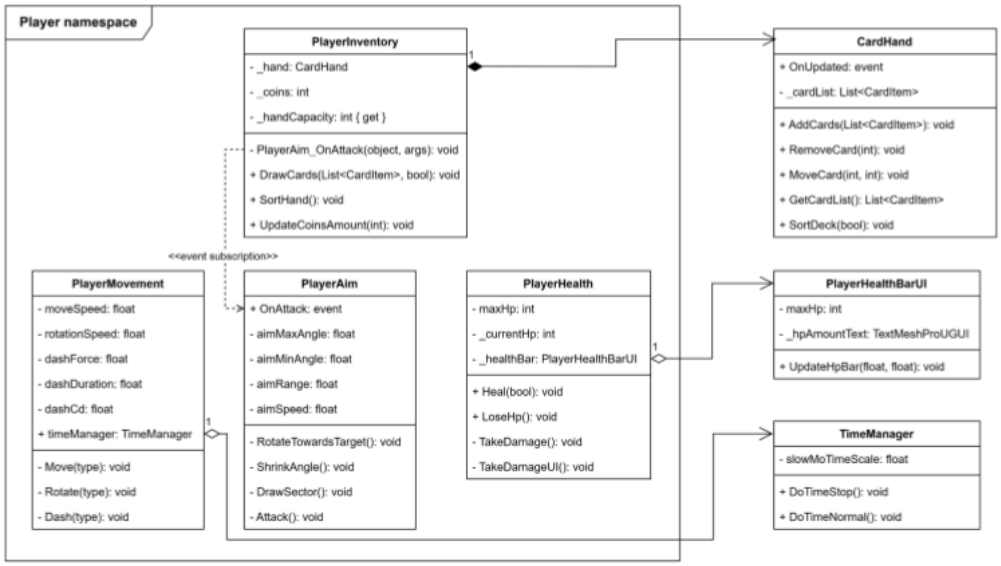
7

ДІАГРАМА МАШИНИ СТАНІВ (стани противника)



8

ДІАГРАМА КЛАСІВ (простір імен Player)



ЕКРАННА ФОРМА

Головне меню

ЕКРАННА ФОРМА

Основний геймплей 11

ЕКРАННА ФОРМА

Магазин 12

ЕКРАННА ФОРМА

Меню паузи

13

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Коваль О. С., Дібрівний О. А. Інтеграція roguelike елементів як засобу підвищення реіграбельності. Застосування програмного забезпечення в інформаційно-комунікаційних технологіях : Матеріали VI Всеукраїнської науково-технічної конференції, Київ, 24 квіт. 2025. ДУІКТ. С. 261–263.

2. Коваль О. С., Дібрівний О. А. Забезпечення безпеки даних у відеогрі. Інформаційні технології – 2025 : Матеріали XII Всеукраїнської науково-практичної конференції молодих учених, Київ, 15 трав. 2025. КСУ ім. Бориса Грінченка. С. 284–285.

14

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

Вихідний код класу PlayerMovement

```
[Header("Move")]
    [SerializeField] private float moveSpeed = 5;
    [Range(0f, 1f)]
    [SerializeField] private float rotationSpeed = 0.5f;

    [Header("Dash")]
    [SerializeField] private float dashForce;
    [SerializeField] private float dashDuration;
    [SerializeField] private float dashCd;
    private float _dashCdTimer;
    private bool _dashing;

    [Header("Refs")]
    [SerializeField] private TimeManager timeManager;
    [SerializeField] private InputActionAsset
inputActionAsset;
    private Animator _animator;

    private InputAction _moveAction;
    private Rigidbody _rb;
    private TrailRenderer _trailFX;
    private static readonly int Running =
Animator.StringToHash("running");


    private void Awake()
    {
        _rb = GetComponent<Rigidbody>();
        _trailFX = GetComponent<TrailRenderer>();
        _animator = GetComponentInChildren<Animator>();
    }

    private void Start()
    {
        _moveAction =
inputActionAsset.FindAction("Move");

        _trailFX.enabled = false;
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.LeftShift)) Dash();

        if (_dashCdTimer > 0) _dashCdTimer -=
Time.deltaTime;
    }

    private void FixedUpdate()
    {
        Move();
        Rotate();
    }

    private void Move()
    {
        Vector2 direction =
_moveAction.ReadValue<Vector2>();
        Vector3 moveVelocity = new Vector3(direction.x, 0,
direction.y).normalized * moveSpeed;
        if(_moveAction.inProgress)
_animator.SetBool(Running, true);
        else _animator.SetBool(Running, false);

        if(_dashing) return;
        _rb.linearVelocity = new Vector3(moveVelocity.x, -
1, moveVelocity.z);

        if (direction == new Vector2(0, 0))
timeManager.DoTimeStop();
        else timeManager.DoTimeNormal();
    }

    private void Rotate()
    {
        Vector2 lookDirection =
_moveAction.ReadValue<Vector2>();

        if (_moveAction.inProgress)
        {
            Quaternion targetRotation =
Quaternion.LookRotation(new Vector3(lookDirection.x, 0,
lookDirection.y));
            transform.rotation =
Quaternion.Slerp(transform.rotation, targetRotation,
rotationSpeed);
        }
    }

    private void Dash()
    {
        if (_dashCdTimer > 0) return;
        _dashCdTimer = dashCd;
        _dashing = true;

        Vector3 forceToApply = new
Vector3(_moveAction.ReadValue<Vector2>().x,
_rb.linearVelocity.y,
_moveAction.ReadValue<Vector2>().y) * dashForce;
        _rb.AddForce(forceToApply, ForceMode.Impulse);

        _trailFX.enabled = true;

        Invoke(nameof(ResetDash), dashDuration);
    }

    private void ResetDash()
    {
        _dashing = false;

        Invoke(nameof(DisableTrailFX), 0.2f);
    }

    private void DisableTrailFX()
    {
        _trailFX.enabled = false;
    }
```

Вихідний код класу PlayerAim

```
public event EventHandler<AttackEventArgs> OnAttack;
public class AttackEventArgs : EventArgs
{
```

```
        public Vector3 AimDirection;
        public float SpreadAngle;
    }

    [Header("Cursor Raycast")]
    [SerializeField] private LayerMask groundLayerMask;
    [SerializeField] private List<GameObject>
blockingAimUIObjects;

    [Header("Aim")]
    [SerializeField] private float aimMaxAngle = 90f;
    [SerializeField] private float aimMinAngle = 5f;
    [SerializeField] private float aimRange = 5f;
    [SerializeField] private int aimSectorResolution = 50;
    [SerializeField] private float aimSpeed = 45f;
    [SerializeField] private float aimAcceleration = 0.1f;
    private Vector3 _aimDirection;
    private float _currentAngle;
    private float _currentAimSpeed;
    private Vector3 _targetPosition;

    [Header("Refs")]
    [SerializeField] private InputActionAsset
inputActionAsset;

    // Refs
    private Camera _mainCamera;
    private InputAction _attackAction;
    private LineRenderer _lineRenderer;


    private void Awake()
    {
        _mainCamera = Camera.main;
        _attackAction =
inputActionAsset.FindAction("Attack");

        _attackAction.canceled += OnAttackReleased;
    }

    private void OnDestroy()
    {
        _attackAction.canceled -= OnAttackReleased;
    }

    void Start()
    {
        _lineRenderer =
GetComponentInChildren<LineRenderer>();
        _lineRenderer.positionCount = aimSectorResolution
+ 2;
        _lineRenderer.useWorldSpace = false;
        _lineRenderer.enabled = false;

        _currentAngle = aimMaxAngle;
        _currentAimSpeed = aimSpeed;
    }

    private void Update()
    {
        if(!_attackAction.IsPressed()) return;

        FindAimDirection();
        ShrinkAngle();
    }

    private void FixedUpdate()
    {
        if(!_attackAction.IsPressed()) return;

        DrawSector();
    }

    private void
OnAttackReleased(InputAction.CallbackContext context)
    {
        Attack();
    }


    private void FindAimDirection()
    {
        Ray ray =
_mainCamera.ScreenPointToRay(Input.mousePosition);
        if (Physics.Raycast(ray, out RaycastHit raycastHit,
999f, groundLayerMask))
        {
            _targetPosition = raycastHit.point;
            _targetPosition.y = transform.position.y;

            _aimDirection = _targetPosition -
transform.position;

            Quaternion rotation =
Quaternion.LookRotation(_aimDirection.normalized);
            _lineRenderer.transform.rotation = rotation;
        }
    }

    private void ShrinkAngle()
    {
        float dt = Time.unscaledDeltaTime;
        _currentAimSpeed += aimAcceleration * dt;
        _currentAngle -= _currentAimSpeed * dt;
        if (_currentAngle < aimMinAngle) _currentAngle =
aimMinAngle;
    }

    private void DrawSector()
    {
        Vector3 center = new Vector3(0, 0.3f,0);

        float halfAngle = _currentAngle / 2f;
        float stepAngle = _currentAngle /
aimSectorResolution;

        if(!IsPointerOverIncludedUI())_lineRenderer.enabled
= true;
        _lineRenderer.positionCount = aimSectorResolution
+ 2;
        _lineRenderer.SetPosition(0, center); // Центр
сектору

        for (int i = 0; i <= aimSectorResolution; i++)
        {
            float angle = -halfAngle + stepAngle * i; // Від -
halfAngle до +halfAngle
            float radians = Mathf.Deg2Rad * angle;

            Vector3 point = new Vector3(
                Mathf.Sin(radians) * aimRange,
                center.y,
                Mathf.Cos(radians) * aimRange
            );

            _lineRenderer.SetPosition(i + 1, point);
        }
    }

    private void Attack()
```

```
        {
            if (!IsPointerOverIncludedUI())
            {
                OnAttack?.Invoke(this, new
AttackEventArgs{AimDirection = _aimDirection,
SpreadAngle = _currentAngle});
            }

            _currentAngle = aimMaxAngle;
            _currentAimSpeed = aimSpeed;
            _lineRenderer.enabled = false;
        }

        private bool IsPointerOverIncludedUI()
        {
            PointerEventData pointerData = new
PointerEventData(EventSystem.current)
            {
                position = Input.mousePosition
            };

            List<RaycastResult> raycastResults = new
List<RaycastResult>();
            EventSystem.current.RaycastAll(pointerData,
raycastResults);

            foreach (RaycastResult result in raycastResults)
            {
                if
(blockingAimUIObjects.Contains(result.gameObject))
                {
                    return true;
                }
            }

            return false;
        }
```

Вихідний код класу PlayerInventory

```
[Header("Hand")]
    [SerializeField] private CardHolder handHolder;
    [SerializeField] private TextMeshProUGUI
cardCountText;
    private CardHand _hand;

    [Header("Other")]
    [SerializeField] private Transform pfCardWorld;
    [SerializeField] private TextMeshProUGUI
coinsCountText;
    [SerializeField] private Transform cardSpawnPoint;

    private int _selectedCardIndex;
    private bool _isSelectedCard;
    private int _coins;

    private int handCapacity => 5;


    private void Awake()
    {
        _hand = new CardHand();

        _hand.OnUpdated += (_, _) =>
        {

handHolder.UpdateInventory(_hand.GetCardList());
            cardCountText.text = _hand.GetCardList().Count +
" / " + handCapacity;
        };
        handHolder.OnCardSelected +=
OnCardSelected_Hand;
        handHolder.OnCardMove += (_, @event) =>
_hand.MoveCard(@event.From, @event.To);

        GetComponent<PlayerAim>().OnAttack +=
PlayerAim_OnAttack;
    }

    private void Start()
    {
        //handHolder.UpdateInventory(_hand.GetCardAll());
        _coins += 100;
        coinsCountText.text = "$ " + _coins;
    }

    public void DrawCards(List<CardItem> cards, bool
takeDamage = true)
    {
        int availableSlots = handCapacity -
_hand.GetCardList().Count;


_hand.AddCards(cards.Take(availableSlots).ToList());
        if(availableSlots > 0 && takeDamage)
GetComponent<PlayerHealth>().LoseHp();
    }

    private void PlayerAim_OnAttack(object sender,
PlayerAim.AttackEventArgs e)
    {
        if(!handHolder.gameObject.activeSelf) return;

        int cardIndexToThrow = _selectedCardIndex;
        if (!_isSelectedCard || _selectedCardIndex < 0)
cardIndexToThrow = _hand.GetCardList().Count - 1;
        if (cardIndexToThrow < 0) return;

        CardWorld cardWorld =
CardWorld.SpawnCardWorld(cardSpawnPoint.position,
_hand.GetCardList()[cardIndexToThrow]);

cardWorld.GetComponent<CardMovementPhysics>().Setup(
e.AimDirection, e.SpreadAngle);

        _hand.RemoveCard(cardIndexToThrow);

        _isSelectedCard = false;
    }

    private void OnCardSelected_Hand(object sender,
CardHolder.SelectedCardEvent e)
    {
        _selectedCardIndex = e.SelectedCardIndex;
        _isSelectedCard = true;
    }

    private void OnCollisionEnter(Collision other)
    {
        if (!other.transform.CompareTag("Coin")) return;
        _coins++;
        coinsCountText.text = "$ " + _coins;

        Destroy(other.gameObject);
    }

    public int GetCoinsAmount()
    {
```

```
        return _coins;
    }

    public void UpdateCoinsAmount(int amount)
    {
        _coins += amount;
        coinsCountText.text = "$ " + _coins;
    }

    private bool isAscending;
    public void SortHand()
    {
        _hand.SortDeck(isAscending);
        isAscending = !isAscending;
    }

    public void SwitchHandUI(bool active)
    {
        if (!handHolder) return;
        var obj = handHolder.transform.parent.gameObject;
        obj.SetActive(active);
    }
```

Вихідний код класу PlayerHealth

```
public event EventHandler OnPlayerDeath;

    [SerializeField] private int maxHp = 10;

    [Header("Refs")]
    [SerializeField] private GameObject healthUI;
    [SerializeField] private Image ouchUI;
    private PlayerHealthBarUI _healthBar;
    private Color _ouchUIColor;

    private int _currentHp;


    private void Awake()
    {
        _currentHp = maxHp;

        if(healthUI != null)_healthBar =
healthUI.GetComponent<PlayerHealthBarUI>();
        if (ouchUI != null)
        {
            ouchUI.gameObject.SetActive(false);
            _ouchUIColor = ouchUI.color;
        }
    }

    private void Start()
    {
        if (_healthBar == null)
        {
            enabled = false;
            return;
        }
        _healthBar.UpdateHpBar(_currentHp, maxHp);
    }

    public void Heal(bool healFull = false)
    {
        if (_currentHp == maxHp) return;
        _currentHp += healFull ? maxHp - _currentHp : 1;
        _healthBar.UpdateHpBar(_currentHp, maxHp);
    }

    public void LoseHp()
    {
        TakeDamage(1);
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Offensive"))
        {
            TakeDamage(1);
        }

        if (other.CompareTag("EnemyBullet"))
        {
            EnemyProjectile projectile =
other.GetComponentInParent<EnemyProjectile>();
            TakeDamage(projectile.GetDamageAmount());

            Destroy(projectile.gameObject);
        }
    }

    private void Update()
    {
        if(!ouchUI.gameObject.activeSelf) return;

        _ouchUIColor.a -= Time.unscaledDeltaTime * 0.7f;
        ouchUI.color = _ouchUIColor;

        if (_ouchUIColor.a <= 0)
        {
            _ouchUIColor.a = 1;
            ouchUI.color = _ouchUIColor;
            ouchUI.gameObject.SetActive(false);
        }
    }

    private void TakeDamage(int damageAmount)
    {
        _currentHp -= damageAmount;
        _currentHp = Mathf.Clamp(_currentHp, 0, maxHp);

        TakeDamageUI();

        if (_currentHp > 0) return;
        OnPlayerDeath?.Invoke(this, EventArgs.Empty);
        ouchUI.gameObject.SetActive(false);
        DestroySelf();
    }

    private void TakeDamageUI()
    {
        ouchUI.gameObject.SetActive(true);
        _healthBar.UpdateHpBar(_currentHp, maxHp);
    }

    private void DestroySelf()
    {
        Destroy(gameObject);
    }
```

Вихідний код класу Target

```
private Material _material;
    private Color _mainColor;
    private Transform _cardInitialsPopupParent;
```

```
    private List<CardInitialsPopup> _cardInitialsPopups;
    private CardComboHandler _cardComboHandler;
    private List<CardItem> _stuckCardItems;
    private List<int> _comboIndexes;
    private List<Transform> _collidedCardsTransforms;

    private void Awake()
    {
        _cardInitialsPopupParent =
transform.Find("cardInitialsPos");
        _cardComboHandler =
GetComponent<CardComboHandler>();

        _material =
GetComponentInChildren<Renderer>().material;
        _mainColor = _material.color;
        _cardInitialsPopups = new
List<CardInitialsPopup>();
        _collidedCardsTransforms = new List<Transform>();
        _stuckCardItems = new List<CardItem>();

        _cardComboHandler.OnComboUpdated +=
TakeDamage;
    }

    private void OnTriggerEnter(Collider other)
    {
        if (!other.transform.CompareTag("Card")) return;
        if
(!other.GetComponentInParent<CardWorld>().enabled)
return;


_stuckCardItems.Add(other.GetComponentInParent<CardW
orld>().GetCardItem());

_cardComboHandler.AddCardToList(_stuckCardItems.Last()
);

_collidedCardsTransforms.Add(other.transform.parent); //
томущо колайдер це дочірній об'єкт то беремо
батьківській об'єкт карти

        other.transform.parent.SetParent(transform);
    }
    private void TakeDamage(object sender,
CardComboHandler.ComboEventArgs e)
    {
        StartCoroutine(TakeHitVisual());

        CreatePopups(e.TotalDamage, e.CurrentCombo,
e.ComboIndexes);
    }

    private IEnumerator TakeHitVisual()
    {
        _material.color = Color.red;
        yield return new WaitForSeconds(0.1f);
        _material.color = _mainColor;
    }

    private void CreatePopups(int damageAmount,
CardComboHandler.ComboName comboName, List<int>
comboIndexes)
    {
        _comboIndexes = comboIndexes;

        var cardInitialsPopup =
CardInitialsPopup.Create(_cardInitialsPopupParent,
_stuckCardItems.Last().cardRank,
_stuckCardItems.Last().cardSuit);
        _cardInitialsPopups.Add(cardInitialsPopup);
        if (_cardInitialsPopups.Count > 5)
        {
            _cardInitialsPopups[0].DestroySelf();
            _cardInitialsPopups.RemoveAt(0);
        }

        UpdateCardPopupHighlight();

        DamagePopup.Create(transform.position +
Vector3.one, damageAmount, comboName);

        if (comboName !=
CardComboHandler.ComboName.None)
ComboNamePopup.Create(transform.position + new
Vector3(-1,1,1), comboName);

    }

    private void UpdateCardPopupHighlight()
    {
        foreach (var popup in _cardInitialsPopups)
        {
            popup.ClearHighlight();
        }

        foreach (int i in _comboIndexes)
        {
            _cardInitialsPopups[i].Highlight();
        }
    }
```

Вихідний код класу CardComboHandler

```
public event EventHandler<ComboEventArgs>
OnComboUpdated;

    public class ComboEventArgs : EventArgs
    {
        public int TotalDamage;
        public ComboName CurrentCombo;
        public List<int> ComboIndexes;
    }

    private List<CardItem> _cardItems;
    private int _totalDamage;
    private ComboName _currentCombo;
    private List<int> _comboIndexes;

    private void Awake()
    {
        _cardItems = new List<CardItem>();
        _comboIndexes = new List<int>();
    }

    public void AddCardToList(CardItem cardItem)
    {
        _cardItems.Add(cardItem);
        if(_cardItems.Count > 5) _cardItems.RemoveAt(0);

        CheckForHighCard();
        CheckForPair();
        CheckForTwoPair();
        CheckForThree();
```

```
        CheckForStraight();
        MakeAcesHigh();
        CheckForStraight();

        CheckForFlush();
        CheckForFullHouse();
        CheckForFour();

        OnComboUpdated?.Invoke(this, new
ComboEventArgs{TotalDamage = _totalDamage,
CurrentCombo = _currentCombo, ComboIndexes =
_comboIndexes});

        _totalDamage = 0;
        _currentCombo = ComboName.None;
        _comboIndexes.Clear();
    }

    public enum ComboName
    {
        None,
        HighCard,
        Pair,
        TwoPair,
        Three,
        Straight,
        Flush,
        FullHouse,
        Four
    }

    private void MakeAcesHigh()
    {
        foreach (var cardItem in _cardItems)
        {
            if (cardItem.cardRank == CardItem.CardRank.Ace)
cardItem.cardRank = CardItem.CardRank.HighAce;
        }
    }

    private void CheckForHighCard()
    {
        if(_cardItems.Count < 5) return;
        MakeAcesHigh();

        bool isLastHighest = _cardItems
            .Take(_cardItems.Count - 1)
            .All(x => x.cardRank.GetHashCode() <
_cardItems[^1].cardRank.GetHashCode());

        if (!isLastHighest) return;

        _currentCombo = ComboName.HighCard;
        _totalDamage = _currentCombo.GetHashCode();

        _comboIndexes.Add(_cardItems.Count - 1);
    }

    private void CheckForPair()
    {
        if (_cardItems.Count < 2) return;
        var groups = _cardItems.GroupBy(card =>
card.cardRank);

        bool isValid = groups.Count(group => group.Count()
== 2) == 1 &&
                    groups.All(group => group.Count() <= 2)
&&
                    groups.Any(group => group.Key ==
_cardItems[^1].cardRank && group.Count() == 2);

        if (!isValid) return;

        var validGroup = groups.FirstOrDefault(group =>
group.Count() == 2);

        _currentCombo = ComboName.Pair;
        _totalDamage = _currentCombo.GetHashCode();

        _comboIndexes = _cardItems
            .Select((card, index) => new { card, index })
            .Where(x => x.card.cardRank == validGroup.Key)
            .Select(x => x.index)
            .ToList();

    }

    private void CheckForTwoPair()
    {
        if (_cardItems.Count < 4) return;
        var groups = _cardItems.GroupBy(card =>
card.cardRank);

        bool isValid = groups.Count(group => group.Count()
== 2) == 2 &&
                    groups.All(group => group.Count() <= 2)
&&
                    groups.Any(group => group.Key ==
_cardItems[^1].cardRank && group.Count() == 2);

        if (isValid)
        {
            var validGroups = groups
                .Where(group => group.Count() == 2)
                .ToList();

            _comboIndexes = validGroups
                .SelectMany(group =>
                    _cardItems.Select((card, index) => new {
card, index })
                        .Where(x => x.card.cardRank ==
group.Key)
                        .Select(x => x.index)
                )
                .ToList();

            _currentCombo = ComboName.TwoPair;
            _totalDamage = _currentCombo.GetHashCode();
        }
    }

    private void CheckForThree()
    {
        if (_cardItems.Count < 3) return;
        var groups = _cardItems.GroupBy(card =>
card.cardRank);

        bool isValid = groups.Count(group => group.Count()
== 3) == 1 &&
                    groups.All(group => group.Count() <= 3)
&&
                    groups.All(group => group.Count() != 2)
&&
                    groups.Any(group => group.Key ==
_cardItems[^1].cardRank && group.Count() == 3);

        if (isValid)
        {
            var validGroup = groups.FirstOrDefault(group =>
group.Count() == 3);
```

```
            _comboIndexes = _cardItems
                .Select((card, index) => new { card, index })
                .Where(x => x.card.cardRank ==
validGroup.Key)
                .Select(x => x.index)
                .ToList();

            _currentCombo = ComboName.Three;
            _totalDamage = _currentCombo.GetHashCode();
        }
    }

    private void CheckForStraight()
    {
        if (_cardItems.Count < 5) return;

        var selectedCards = _cardItems
            .Select(card => (int)card.cardRank)
            .OrderBy(rank => rank)
            .ToList();

        for (int i = 1; i < selectedCards.Count; i++)
        {
            if (selectedCards[i] != selectedCards[i - 1] + 1)
            {
                return;
            }
        }
        _comboIndexes = new List<int> { 0, 1, 2, 3, 4 };
        _currentCombo = ComboName.Straight;
        _totalDamage = _currentCombo.GetHashCode();
    }

    private void CheckForFlush()
    {
        if (_cardItems.Count < 5) return;

        var selectedCards = _cardItems
            .Select(card => card.cardSuit)
            .ToList();

        foreach (var cardSuit in selectedCards)
        {
            if (selectedCards[0] != cardSuit) return;
        }

        _comboIndexes = new List<int> { 0, 1, 2, 3, 4 };
        _currentCombo = ComboName.Flush;
        _totalDamage = _currentCombo.GetHashCode();
    }
    private void CheckForFullHouse()
    {
        if (_cardItems.Count < 5) return;

        var rankGroups = _cardItems.GroupBy(card =>
card.cardRank)
            .Select(group => new { Rank = group.Key, Count
= group.Count() })
            .ToList();
        if (rankGroups.All(g => g.Count != 3) ||
rankGroups.All(g => g.Count != 2)) return;

        _comboIndexes = new List<int> { 0, 1, 2, 3, 4 };
        _currentCombo = ComboName.FullHouse;
        _totalDamage = _currentCombo.GetHashCode();
    }

    private void CheckForFour()
    {
        if (_cardItems.Count < 3) return;

        var groups = _cardItems.GroupBy(card =>
card.cardRank);
        if (groups.Any(group => group.Key ==
_cardItems[^1].cardRank && group.Count() != 4)) return;

        _comboIndexes = _cardItems
            .Select((card, index) => new { card, index })
            .Where(x => x.card.cardRank ==
_cardItems[^1].cardRank)
            .Select(x => x.index)
            .ToList();

        _currentCombo = ComboName.Four;
        _totalDamage = _currentCombo.GetHashCode();
    }
```