МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВАДИМА ГЕТЬМАНА
Навчально-науковий інститут
«Інститут інформаційних технологій в економіці»
Кафедра інформаційних систем в економіці

ОСВІТНЬО ПРОФЕСІЙНА ПРОГРАМА «КОМП'ЮТЕРНІ НАУКИ»
галузь знань 12 «Інформаційні технології»
спеціальність 122 «Комп'ютерні науки»

Форма навчання: денна

КВАЛІФІКАЦІЙНИЙ БАКАЛАВРСЬКИЙ ПРОЕКТ

на тему

Розробка карткової Rogue-Like відеогри

здобувача Демиденка Дмитра Вікторовича _____

Науковий керівник:
к.е.н., доцент
_____Козак І.А.

Проект допущений до захисту перед екзаменаційною комісією з атестації здобувачів вищої освіти
завідувач кафедри:
к.е.н., доцент.
_____Тішков Б.О.

Київ 2024

Міністерство освіти і науки України
Київський національний економічний університет імені Вадима Гетьмана
Навчально-науковий інститут «Інститут інформаційних технологій в економіці»

Кафедра інформаційних систем в економіці

ОСВІТНЬО_ПРОФЕСІЙНА ПРОГРАМА «КОМП'ЮТЕРНІ НАУКИ»
галузь знань 12 «Інформаційні технології»
спеціальність 122 «Комп'ютерні науки»

ПОГОДЖЕНО:	ЗАТВЕРДЖУЮ:
Керівник проектної групи(гарант)	
освітньо-професійної програми	Завідувач кафедри
_____Іванченко Г.Ф.	_____Тішков Б.О.
"____" _____2024 р.	"____" _____2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

здобувача вищої освіти ***Демиденка Дмитра Вікторовича***

очної (денної) форми навчання

на підготовку кваліфікаційного бакалаврського проекту
на тему: «Розробка карткової Rogue-Like відеогри»

Тему затверджено наказом ректора Університету від «11» березня 2024 р. № 529 ст

Кваліфікаційний бакалаврський проект виконується на матеріалах

виконується на матеріалах предметної області та літературних джерел

План кваліфікаційного бакалаврського проекту

Розділ І Охарактеризувати та проаналізувати галузь розробки ігор пов'язану з колекціонними картковими іграми_____
_____

Розділ ІІ Розробити вимоги для проектування цифрової карткової гри_____
_____

Розділ ІІІ Реалізація карткової гри згідно встановлених раніше проектувальних вимог_____
_____

Об'єкт дослідження Індустрія розробки відеоігор._____
_____

Предмет дослідження карткова Rogue-like гра

Мета кваліфікаційного бакалаврського проекту Дослідження процессу розробки комп'ютерних та мобільних ігор

Конкретні завдання, які студент повинен виконати для досягнення поставленої мети:

У розділі I Провести детальний аналіз схожих проектів, порівняти між собою вже існуючих та більш менш популярних представників жанру. Охарактеризувати архітектурні та гейм-дизайнерські проблеми що можуть виникнути під час розробки гри.

У розділі II Провести аналіз та специфікувати вимоги до інформаційної системи гри. Змоделювати концепцію та архітектуру гри на базі поставлених вимог, визначитися з конкретними задачами та алгоритмом їх вирішення.

У розділі III Реалізувати основні компоненти системи такі як інформаційне забезпечення, технічне забезпечення, програмне забезпечення. Закінчити реалізацію інформаційної системи.

Завдання підготував

науковий керівник _____ **Козак Ірина Антонівна**

"15" березня 2024 р.

Завдання одержав
студент _____ **Демиденко Дмитро Вікторович**

"15" березня 2024 р.

АНОТАЦІЯ

Кваліфікаційного бакалаврського проекту студента 4 курсу Навчально-наукового інституту «Інститут інформаційних систем в економіці» Демиденка Дмитра Вікторовича, виконаної на тему: «Розробка карткової Rogue-Like відеогри»
Київ: кафедра інформаційних систем в економіці, 2024 р.

Метою даного проекту є розробка карткової комп'ютерної гри у жанрі rogue-like, яка буде поєднувати елементи стратегії, пригод та випадковості. Головною метою гри є надання гравцеві можливості досліджувати випадково генеровані локації, збирати ресурси, зброю та здобувати здібності в процесі, боротися зі злочинцями та монстрами, а також розвивати свого персонажа.

Дослідження жанру rogue-like: Проведено аналіз існуючих карткових комп'ютерних ігор у жанрі rogue-like, вивчено їхні особливості та механіки гри для подальшого використання в проекті.

Проектування геймплею: Сформульовано основні правила гри, визначено головні механіки та інтерфейс гравця. Розроблено концепцію випадково генерованих локацій та персонажів. Описано проблеми створення геймплею та способи їх подолання.

Графічний дизайн та анімація: Створено графічний дизайн інтерфейсу гравця, а також анімації для персонажів, монстрів та інших об'єктів у грі.

Тестування та оптимізація: Проведено тестування гри на предмет виявлення помилок та недоліків у геймплеї. Після цього проведено оптимізацію гри для підвищення її продуктивності та стабільності.

Випуск та розповсюдження: Гра розроблена для платформ PC та мобільних пристроїв. Планується розповсюдження гри через цифрові магазини та інші канали розповсюдження.

Висновок: У результаті розробки карткової комп'ютерної гри у жанрі rogue-like було досягнуто успішних результатів у більшості аспектів проекту. Гра відповідає основним стандартам жанру, має цікавий геймплей та добре виглядає, що робить її привабливою для широкого кола гравців.

РЕФЕРАТ

Кваліфікаційний бакалаврський проект містить **78** сторінок, **8** таблиць, **42** рисунки, список літератури з **7** найменувань, **3** додатки.

“Розробка карткової Rogue-Like відеогри”

Перелік ключових слів: Відеогра, Скрипти, Штучний інтелект, Програмна архітектура, Геймплей, Схема, Прототип.

Предметом дослідження є карткова Rogue-like гра.

Об’єктом дослідження виступає: Індустрія розробки відеоігор.

Мета кваліфікаційного бакалаврського проекту полягає в: Дослідження процессу розробки комп’ютерних та мобільних ігор.

Завданнями кваліфікаційного бакалаврського проекту є: Розробці концептуально нової карткової Rogue-like гри.

Апаратні та програмні засоби, що використовувались при проектуванні: Ігровий рушій Unity, растровий графічний редактор Krita, простір 3D моделювання Blender.

Результати досягнуті в процесі роботи (також їх новизна та ступінь упровадження): Результати помірні. новизна абсолютна, практично не має аналогів. Ступінь упровадження – помірний.

Одержані результати можуть бути використані (рекомендації щодо практичного застосування здобутих результатів, із зазначенням ефективності, сфери використання): Під час проектування та розробки комп’ютерних та мобільних ігор.

Рік виконання кваліфікаційного бакалаврського проекту: 2023-2024.

Рік захисту кваліфікаційного бакалаврського проекту: 2024

ЗМІСТ

ВСТУП

У сучасному світі ігрова індустрія займає величезну нішу серед великого багатомільярдного бізнесу розваг та відпочинку. Проте ігри можуть не лише розважати, але й стати потужним інструментом для навчання, соціалізації та розвитку креативності. У цьому контексті, розробка ігор стає актуальним завданням, оскільки вони можуть сприяти розвитку стратегічного мислення та логіки. Ця бакалаврська робота присвячена розробці карткової rogue-like гри. Мета проекту полягає в створенні цікавої гри, яка сприятиме розвитку когнітивних навичок, творчого підходу та аналітичних здібностей гравців. В рамках роботи буде детально розглянуто процес проектування, вибір теми та розробку механік гри. Особлива увага буде приділена аналізу існуючих аналогів карткових ігор та їх впливу на розвиток ігрової індустрії. Акцент буде на механіки гри, на способи їх реалізації, на когнітивний шлях який необхідно пройти аби зрозуміти яка саме архітектура повиння бути реалізована та як вона повинна бути імплементована. Також буде проанілізовано та обгрунтовано вибір того чи іншого способу імплементації системи з точки зору модульності та варітивності внутрішнього ігрового досвіду гравця, та зручності подальшої модифікації та розвитку проекту. Крім того, робота розгляне технічні аспекти розробки, такі як вибір програмних інструментів, платформи розгортання та оптимізація для різних пристроїв. Завершуючи вступ, слід відзначити, що розробка карткової гри – це захоплюючий, творчий і дуже складний процес, який об'єднує в собі технічні навички та вміння створювати унікальний геймплей. Наукова і практична цінність цього проекту полягатиме в обґрунтуванні виборів при розробці та виявленні перспектив для подальших досліджень у галузі розробки карткових ігор та цифрових ігор загалом.

РОЗДІЛ 1 ХАРАКТЕРИСТИКА ТА АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Характеристика предметної галузі та об’єкта дослідження

Предметна область даної роботи охоплює розробку та функціонування карткових ігор. А саме ігор, основною механікою яких є необхідність вибору найкращого варіанту розвитку подій з урахуванням наявних ресурсів. Звісно ж мова зараз йде не про звичайні гральні карти, а про значно більш комплексні системи які використовують значно більшу кількість змінних які необхідно враховувати при розігруванні карт, та витраті ресурсів

Основні аспекти предметної області: Історія та еволюція карткових ігор: Вивчення розвитку карткових ігор дозволяє з'ясувати їхню еволюцію від традиційних народних ігор до сучасних глобальних технологічних проектів.

Типи та жанри карткових ігор: Різноманіття карткових ігор включає в себе стратегічні, колекційні, рольові та інші жанри, що визначають їхні особливості та механіку.

Вплив карткових ігор на розвиток гравців: Аналіз впливу гри на когнітивний розвиток, соціальні навички та творчість гравців є важливим аспектом для розуміння корисності карткових ігор.

Технічні аспекти розробки карткових ігор: Вивчення програмних та апаратних аспектів дозволяє розуміти технічні виклики та можливості при створенні гри.

Інновації у карткових іграх: Врахування сучасних тенденцій та інновацій у галузі дозволяє створювати унікальні та привабливі проекти.

Об'єктом дослідження є конкретна карткова гра, розробка якої буде вивчатися в межах даної бакалаврської роботи. Обрана гра слугуватиме прикладом для аналізу та дослідження різних аспектів предметної області, таких як геймдизайн, відеоігрові механіки, взаємодія гравців, елементи навчання та технічні рішення. Проведення детального аналізу конкретного проекту надасть можливість отримати практичні знання та зробити необхідні

висновки для подальшого, більш ефективного, вдосконалення та розробки відео ігор.

1.2 Аналіз літературних джерел та практичного досвіду використання ІС і технологій в предметній галузі

Щож, порівняти програмні системи в класичному розумінні цього слова в нас не вийде, оскільки вихідний код більшості (99%) ігор відсутній у відкритому доступі. Проте я можу порівняти геймплей та ігрові механіки.

Якщо ми будемо порівнювати карткові ігри то серед них можно визначити всього 3 основні стовпи геймплейних систем.

Спочатку потрібно визначитися чим ми боремось. Тут мається на увазі що саме карти репрезентують у нашій грі, якщо карти це юніти і крім юнітів в грі більше нічого не використовується то це UnitBasedSystem. Якщо в грі карти репрезентують закляття, елементи або чаклунства то це SpellBasedSystem. Якщо ж в грі ми можемо користуватися і юнітами і закляттями то це MixedSystem. **[5]** Порівняння бойових систем карткових ігор подано в табл.1

Далі треба визначити за рахунок чого ми боремось. В будь якій грі має бути певний важіль що буде сповілнювати або обмежувати гравця, таким чином змушуючи його думати наперед та роблячи гру складнішою, цікавішою та комплекснішою. Якщо у нас в грі необмежена кількість карт але для їх використання нам необхідно витрачати будь який сторонній ресурс то це ManaBasedSystem. Якщо у нас немає якогось виокремленого ресурсу який би витрачався при використанні карт але карти у нас не безлімітні то це DeckBasedSystem. Якщо у нас карти безлімітні але після використання вони стають неактивними на певний час то це CooldownBasedSystem. В цьому пункті виокремити базу дуже складно оскільки карткові ігри комбінують данні механіки в абсолютно довільному порядку, або ж взагалі обходяться без них. **[5]** Порівняння системи ворогів карткових ігор подано в табл.2

І звісно треба розуміти з чим ми боремось. Тут що дивно варіантів всього два. Коли ми боремося з кимось хто має такі ж свмі або трохи інші карти як і

ми, це може бути як ШІ так і інша людина проти якої ми боремося онлайн, це називається DuelBasedSystem. Якщо ж ми боремося проти будь чого що має власні унікальні здібності, і не користується картами на зразок наших, то це MonsterBasedSystem. Таким чином ми можемо побудувати таблицю порівняннь де будуть відмічені використовувані геймплейні системи. **[5]** Порівняння ресурсних систем карткових ігор подано в табл.3

Таблиця 1. Порівняння бойових систем карткових ігор.

Fight System			
Назва гри	***Назва використаної системи***		
	UnitBasedSystem	SpellBasedSystem	MixedSystem
Slay The Spire		X	
Hearthstone			X
Magic:The Gathering			X
Legends Of Runeterra			X
Yu-Gi-Oh! Master Duel		X	
Inscryption	X		
Gvint			X
Faeria			X

Таблиця 2. Порівняння системи ворогів карткових ігор.

Enemy System		
Назва гри	***Назва використаної системи***	
	DuelBasedSystem	MonsterSystem
Slay The Spire		X
Hearthstone	X	

Таблиця 2. (продовження)

Magic:The Gathering	X	
Legends Of Runeterra		X
Yu-Gi-Oh! Master Duel	X	
Inscryption	X	
Gvint	X	
Faeria		X

Таблиця 3. Порівняння ресурсних систем карткових ігор.

Resource System			
Назва гри	***Назва використаної системи***		
	ManaBased System	DeckBased System	CooldownBased System
Slay The Spire	X		
Hearthstone	X	X	
Magic:The Gathering	X		X
Legends Of Runeterra	X		X
Yu-Gi-Oh! Master Duel	X	X	
Inscryption		X	
Gvint		X	
Faeria	X		X

1.3 Характеристика архітектурних та геймплейних проблем при розробці системи

Перед тим як вирішувати яку саме архітектуру використовувати необхідно зрозуміти яку геймплейну структуру має гра. Геймплейна структура – це набір пов’язаних між собою ігрових механік які формують ігровий досвід шляхом взаємодії з гравцем, наданням певних умов і правил, а також інструментів для взаємодії з оточенням, те наскільки умови та інструменти тісно пов’язані між собою і визначає якість структури геймплею.

Найпоширенішим спосособами досягнення структурованості геймплею є прогрессія і бути вона може Фізична або Ментальна.

Варто акцентувати увагу що ці види прогрессії не є взаємовиключними, вони можуть, і на мою думку повинні, співіснувати, і у випадку грамотної реалізації підвищать якість геймплейної структури.

Фізична – тип прогрессії в якій досвід та розумові здібності гравця мають значно меньше значення ніж проведений у грі час та заздалегідь проведені розрахунки. Фізична прогрессія завжди прив’язана до цифр і цифри напряму впливають на успіх гравця. Як правило деякі елементи геймплею (або ж одразу всі) в такій грі можуть перестати бути цікавими якщо зробити один чи декілька числових параметрів (змінних) занадто високими або занадто низькими, тому створення ігор що орієнтуються на фізичну систему прогрессії потребує скурпульозного математичного аналізу та розрахунків для досягнення адекватного балансу.

Ментальна – тип прогрессії в якому досвід гравця, його розумові здібності, інтуїція та допитливість грають основну та ключову роль у тому чи зможе гравець пройти гру, і як легко чи навпаки складно йому буде. Такий тип прогрессії по праву вважається значно складнішим з точки зору як гравця так і розробника, оскільки хоч фізичний тип і потребує хорошого знання мтематики, проте фізичний тип прогрессії можна зробити об’єктивно справедливим, а головне контрольованим. А ментальний тип прогресії

перекладає майже всю відповідальність за структурування геймплею на голову гравця та на його сприйняття.

Якщо брати за приклад класичні колекційні карткові ігри, такі як Hearthstone то в них тип прогресії буде класично ментальним. Здавалося б, Hearthstone розривається від нагромадження чисел та розрахунків, проте... Сам розмір цих чисел нічого не означає та й взагалі є константним. Без вміння добре розіграти карти ти не зможеш перемогти одними числами, крім того основною відмінною рисою фізичного типу прогрессії є так званий Min/Max, термін в ігровому жаргоні що використовують для описання процесу підбирання найкращого можливого спорядження чи комбінації спорядження для найефективнішого проходження гри. Подібний підхід абсолютно неможливо застосувати в грі де є нескінченна комбінація карт з унікальними здібностями і унікальним сприйняттям цих здібностей гравцем збиравшим колоду. Підібрати найоптимальнішу колоду для будь-якої ситуації не вийде.

Якщо заглибитися у тему то карткових ігор з фізичним типом прогрессії по суті немає. Це дуже просто пояснити, карткові ігри потребують концентрації та мають достатній рівень структурованості геймплею самі по собі. Фізичний тип прогрессії як привило характерний для динамічних ігор в реальному часі таких як RPG наприклад, в яких персонажа буквально описано набором змінних. Фізичний тип прогрессії буде надмірним так як внесе дуже великі зміни в баланс, при цьому потреби які будуть виставлені для розкриття і доцільної інтеграції цієї системи прогрессії будуть дуже великими. У всякому випадку так воно є для класичних колекціонних карткових ігор.

Як вже було згадано основною причиною чому карткових ігор з фізичною системою прогрессії немає це те, що класичні карткові ігри самі по собі достатньо комплексні і цілісні і потребують обдумування кожної дії. Таким чинм ми приходимо до висновку що фізична система в класичних іграх цього жанру буде п’ятим колесом. Саме тому мною було прийняте рішення створити карткову гру в якій використання фізичної системи буде доцільним.

1.3.1 Рішення щодо архітектури системи

Визначившись із геймплейною структурою можна братись за саму архітектуру. Чому нам було важливо спочатку розробити геймплейну концепцію, бо архітектура повинна передбачати розширення та модуляцію, на початкових етапах розробки під час імплементації деяких основних механік ми можемо занадто сильно специфікувати деякі системи, що може призвести до того що ми не зможемо змінити заздалегідь не передбачені сценарії поведінки тих чи інших об’єктів чи сутностей, через що в майбутньому нам доведеться перероблювати деякі системи майже повністю. Тому аби не робити зайвої роботи необхідно ще “на березі” визначитися хоча б з деякими системами.

Тепер нам треба визначити структуру, а саме як різні модулі програми будуть спілкуватися між собою. Взагалі в ідеальній архітектурі різні модулі програми мають взагалі не знати про існування одне одного, а користуються виключно результатами проходження вхідної інформації через послідовне виконання внутрішніх функцій кожного модуля. Але така архітектура була б доцільною в класичних обчислювальних ІС, в нашому вападку у нас відеогра. Які взагалі типи програмних архітектур використовуються в відеоіграх?

Ні для кого не секрет, що у великих іграх досить складна архітектура. Складна сутність, складна взаємодія між класами. Використання стандартного ООП підходу тягне за собою постійні переробки коду, внаслідок чого час розробки значно збільшиться. Проблема ховається у спадкуванні. Точніше проблема крихких базових класів, коли неможливо змінити реалізацію початкового типу, не порушивши корректність роботи дочірніх типів. Оскільки я роблю свою гру не на власному рушії а на вже готовому, то і обирати сильно не доводиться. В Unity імплементовано так звану систему КОП.

КОП (компонентно-орієнтовне програмування) було сформовано саме як вирішення цієї проблеми. Коротко, принцип роботи КОП такий:

Є клас-контейнер, а також клас-компонент, який можна додати в клас-контейнер. На **(Рис. 1 – Об’єкт з компонентами)** зображено об'єкт що складається з контейнера та компонентів у цьому контейнері. В контейнері знаходиться 3 компонента, 1 базовий та 2 побічні. Компонент Transform є базовам, він визначає положення об’єкта в просторі та його розміри. Компонент Animator є вбудованим побічним компонентом Unity що дає змогу призначати об’єкту анімації та керувати ними. Компонент Enemy є користувацьким побічним класом, розробленим мною, він необхідний для зберігання та відстежування характеристик ворога а також для застосування поведінки ворога.

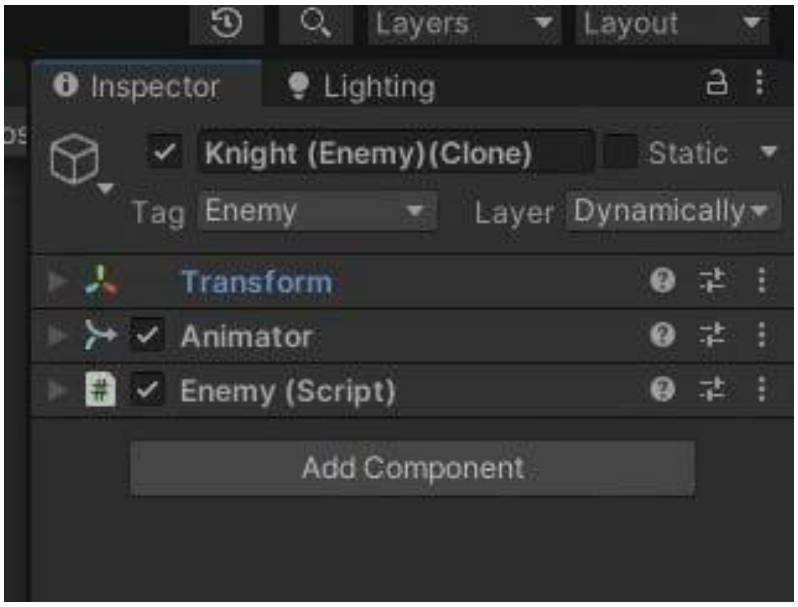


Рисунок 1. Об’єкт з компонентами.

Компоненти в чомусь схожі на інтерфейси. При цьому інтерфейси дають можливість тільки виділити у класів загальну сигнатуру функцій і властивостей, а компоненти дозволяють окремо винести загальну реалізацію класів. У підході ООП об'єкт визначається приписаним йому класом. У підході КОП об'єкт визначають компоненти, з яких він складається. Не важливо, який це об'єкт, важливо з яких компонентів він складається, і що він вміє виконувати. Працюючи з КОП, можна спростити собі завдання повторного використання вже написаного коду, адже ви просто вставляєте готовий компонент в різні об'єкти. Завдяки цьому комбінуючи різні компоненти, можна зібрати новий тип об'єкта. Візьмемо, наприклад, об'єкт “Персонаж”.

Створюючи його через ООП, ви отримали б один великий клас, можливо, успадкований від чогось. Для КОП це комбінація компонентів, з яких і складається об'єкт “Персонаж”. Наприклад: Управління персонажем Character (CharacterHandler), обробник зіткнень CharacterCollision (CharacterCollisionHandler), характеристики персонажа компонент Stats (StatsHandler). Оскільки при використанні класичного успадкування нам довелося би робити декілка рівнів, ми підемо іншим шляхом. Ми не будемо повністю відмовлятися від успадкування, а просто використаємо його як спосіб комунікації між окремими обробниками певних модулей системи.

На **(Рис. 2 – Модель архітектури)** зображено схематичну архітектурну ієрархію функціональних модулів гри. Головний розподілювач на схемі бере на себе роль хабу, який буде створювати екземпляри обробників та запускати їх. Самі обробники будуть виступати як хаби для компонентів однієї групи, наприклад ящо у нас є декілька видів карт то CardHandler (Картковий Обробник) буде нести в собі всю інформацію про карти, їх статус, та будь які інші параметри та змінні що напряму впливають безпосередньо на взаємодію з самими картами. Модуль під назвою посилання, також в майбутньому відомий як GameStorage - це просто збірник усіх моделей сунтностей та текстур які ми будемо створювати та якими програмні модулі будуть маніпулювати в процесі гри, для запуску власне кажучи самої гри.

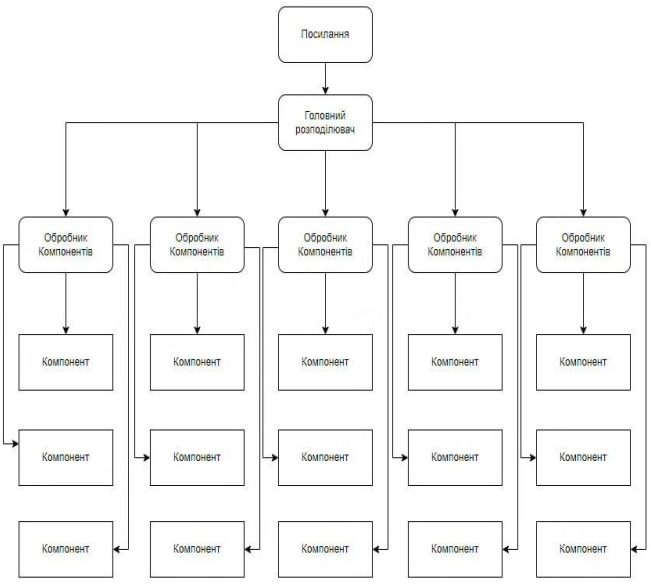


Рисунок 2. Модель архітектури.

РОЗДІЛ 2 РОЗРОБКА ВИМОГ І МОДЕЛЮВАННЯ ІНФОРМАЦІЙНОЇ ПІДСИСТЕМИ

2.1 Специфікація вимог до інформаційної підсистеми

Будь-яка систем потребує перелік вимог **(Рис. 3 – Діаграма вимог до ІС)** для корректного функціонування. Ігри не просто не є виключенням, а на відміну від деяких інших областей програмної розробки, не зможуть пройти шлях від задумки до реалізації навіть на половину, без добре сформованої системи вимог.

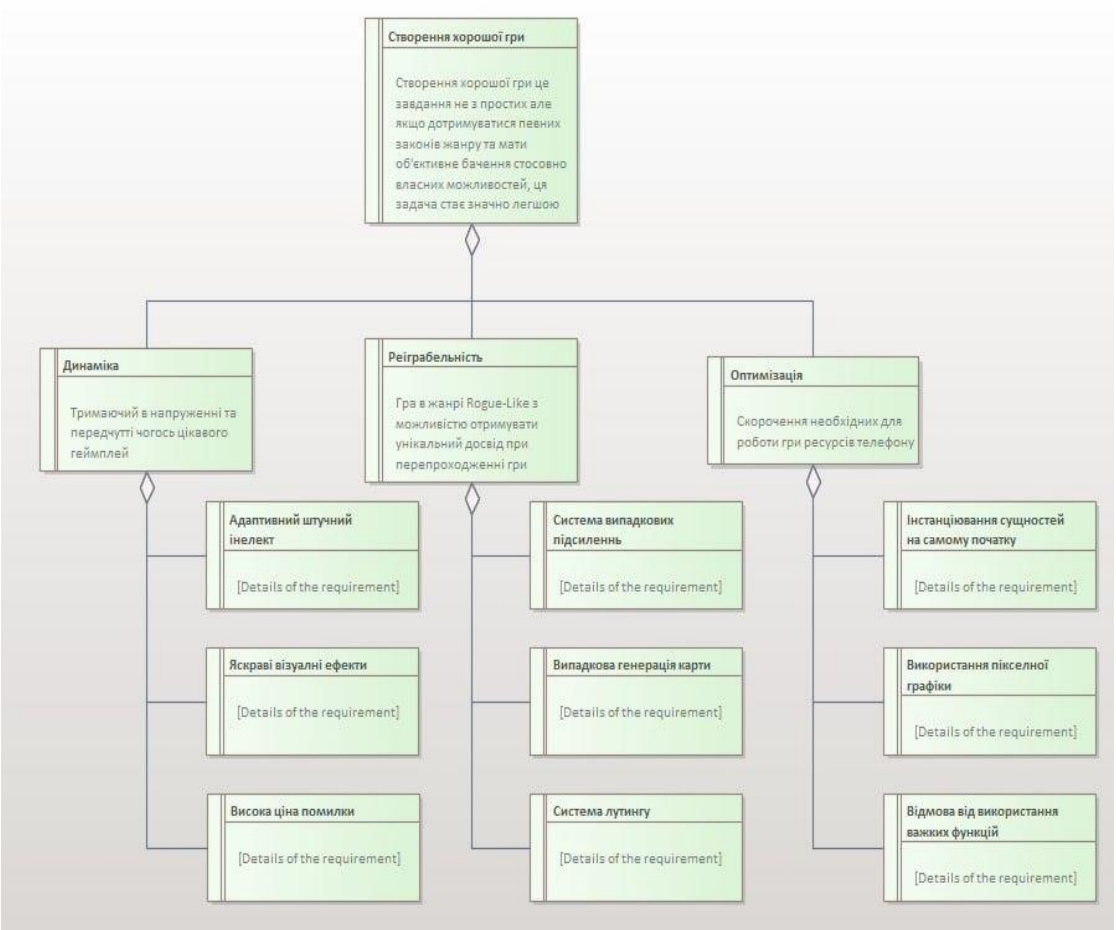


Рисунок 3. Діаграма вимог до ІС.

На цій **(Рис. 4 – Таблиця параметрів специфікацій вимог)** таблиці зображені базові вимоги до моєї гри. Базові не означає що вони не в повному переліку, на цій діграммі якраз таки зображені всі ті вимоги які зроблять гру

такою якою я її бачу, але вони були зкомпоновані в більш тісні блоки, оскільки вимог до гри може бути і тисяча, і одночасно стежити за виконанням усіх поставлених цілей може бути м’яко кажучи нераціонально. І зкомпоновані вони були таким чином аби їх виконання паралельно охоплювало максимально велику кількість необхідних для їх виконання підзадач за якими окремо слідкувати було б важко а в складі одного зі скомпонованих блоків, цілком адекватно.

Item	Type	Identifier	Status	Difficulty	Priority	Phase
Створення хорошої гри			Approved	Medium	High	0
Динаміка			Approved	Medium	Medium	1
Адаптивний штучний інелект			Approved	High	Low	1
Висока ціна помилки			Approved	Low	Low	1
Яскраві візуалні ефекти			Approved	High	Low	1
Реіграбельність			Validated	Low	Medium	1
Випадкова генерація карти			Validated	Medium	Medium	1
Система випадкових підсиленнь			Validated	Medium	High	1
Система лутингу			Validated	Medium	High	1
Випадкова генерація карти			Validated	Medium	Medium	1
Система випадкових підсиленнь			Validated	Medium	High	1
Система лутингу			Validated	Medium	High	1
Оптимізація			Proposed	Medium	High	1
Використання пікселної графіки			Proposed	Medium	Medium	1
Відмова від використання важких функцій			Proposed	Medium	Medium	1
Інстанціювання сущностей на самому початку			Proposed	Medium	Medium	1

Рисунок 4. Таблиця параметрів специфікацій вимог.

- *Адаптивний штучний інтелект:*

Пріоритет адаптивного штучного інтелекту низький, оскільки основна задача яку вона має вирішувати це бойова іммерсивність. Досягти подібного в будь якій грі доволі важко, а тим паче в мобільній грі яку розроблює одина людина. Штучний інтелект доволі вимогливий до обчислювалних потужностей, яких як правило у телефонів немає, а складність розробки цього ШІ майже співставна до розробки усієї гри цілком, тому є висока вірогідність що ШІ залишиться на тому рівні на якому він функціонує зараз, але якщо його робота буде негативно впливати на працесздатність гри, його буде спрощено та оптимізовано.

- *Висока ціна помилки:*

Одна з найкласичніших геймплейних механік, чи коректніше сказати сукупність геймплейних механік яка робить будь-які дії гравця в грі надпотужними, та надає їм наймовірної ваги. Саме так, коли будь-яка помилка в грі може наймовірно сильно знизити шанси гравця на перемогу це перетворює прийняття будь якого рішення гравцем в малесеньку адреналінову гірку, а втілення прийнятого рішення, і його успіх/провал відчувається як справжній бій вживу. Тим не менш варто пам’ятати що не всі люди люблять, чи хоча б здатні витримувати, подібний рівень напруги впродовж певного часу, тому гравцям завжди необхідно давати вибір, як правило цей вибір пряцює за формулою (ризик / час = нагорода), таким чином чим меньше гравець витрачає часу на прийняття рішення та чим більшому ризику він в цей момент піддає себе, тим більша нагорода повинна бути, аби вмілі гравці відчували значимість своїх стараннь а невмілі гравці мали стимул вдосконалюватися.

- *Яскраві візуальні ефекти:*

Дії гравця повинні мати не лише емоційну віддачу, а й візуальну, поєднання цих двох типів двосторонньої взаємодії гравця і гри дозволяє досягти дуже високого рівня іммерсивності та зацікавленності. Головне завдання будь- яких візуальних ефектів в грі – зробити процес максимально

приємним та дозволити гравцю відчути віддачу від власних дій, що сильно підвищує бажання продовжувати грати в гру.

- *Випадкова генерація карти:*

Ця механіка дозволяє гравцю вибирати свій стиль гри, коли в тебе є карта з кімнатами, то ти можеш вибирати, чи хочеш ти продовжувати досліджувати поверх, витрачаючи час але отримуючи більше ресурсів та нагород, чи піти на наступний поверх де вороги будуть складніші але й нагорода за них буде більшою. Також на карті є кімнати в яких буде складно на будь-якому поверсі – кімнати випробувань, чи кімнати які можуть підсилити тебе без ризику натрапити на сильного ворога – арсенал, чи кімнати в яких ти зможеш обміняти одні ресурси на інші – торгівні пости і тп. Що збільшує необхідність аналізувати своє переміщення та розраховувати ресурси.

- *Система випадкових підсилень:*

Система підсилень покликана підвищити реіграбелність за рахунок унікальності стилю гри, причому який саме стиль гри буде у вас у наступній вашій спробі вибираєте ви самі, в залежності від того яким елементом ви частіше користуєтесь.

- *Система лутингу:*

Систем лутингу зробить бій з ворогами не просто прокладкою між отриманням підсилень, бо при перемозі над ворогом буде шанс що вам випаде з нього частина його екіпірування яку можна буде одягнути на себе, чи використати в бою. Звісно що з кожного ворога з певним шансом буде випадати свій предмет/предмети, таким чином зустріч з певними ворогами буде не менш захоплюючою ніж отримання якогось підсилення.

- *Використання піксельної графіки:*

При розробці мобільної гри необхідно задумуватись навіть про такі речі як вага текстур. Тому мною, як і більшістю незалежних розробників, була обрана піксельна графіка. Текстури мало займають і як правило не мають артефактів а також не потребують моделей та швидше відмальовуються графічною картою, ідеальний варіант для мобільної платформи.

- Відмова від використання важких функцій

В Unity є багато зручних але дуже важких функцій. Не дивлячись на те що їх використання дозволило б зробити архітектуру гри наймовірно кампактною та маштабованою, в деяких моментах гра мала б використовувати ці функції по декілька разів на процессорний тік, а це дуже сильно б’є не лише по кількості кадрів в грі, але й по температурі і рівню заряду девайсу. Саме тому мені доведеться розробити власну архітектуру яка допоможе мені уникнути використання цих функцій при цьому залишивши максимальну компактність та модульність.

- Інстанціювання сутностей на самому початку:

Під час звантаження рівня гра буде створювати абсолютно всі об’єкти що можуть фігурувати на сцені а потім за допомогою спеціальної системи ці об’єкти будуть виставлятися на ігровому полі, таким чином ми уникнемо постійного створення ігрових об’єктів, пришвидшимо завантаження кімнат за рахунок використання оперативної пам’яті та знизимо навантаження на процесор.

2.2 Постановка задачі та алгоритм розв’язання задачі

Призначення розв'язання даної задачі не є очевидним з економічної точки зору, отримння вигоди не є першочерговою ціллю створення даної системи. Задача полягає в симуляції віртуального ігрового процесу з метою розважити, морально розвантажити чи навпаки змусити думати ту людину, що вирішила скористатися “системою”. Потреба використання ЕОМ для виішення задачі більш ніж очевидна, велика кількість розрахунків що має проходити миттєво, штучний інтелект ворогів та графічна складова що має заохочувати користувачів продовжувати користуватися “системою”, це все безперечно задачі що під силу виключно ЕОМ.

Вихідна інформація в грі допомагає гравцю зрозуміти що взагалі відбувається, на відміну від білшості класичних ІС, де вихідна інформація не несе на собі сильного графічного навантаження, в відеоіграх вихідна інформація це буквально кожен кадр, бо кожний оброблений ЕОМ кадр є

цінною для нас інформацією, і у випадку недостатньої кількості цих “вихідних повідомлень”, система буде не придатна до використання. Тому можна сказати що виконання задачі не припиняється ніколи за виключення тих випадків коли гравець самостійно вирішує вийти з системи. Тим не менш вихідна інформація в більш класичному розумінні слова в іграх звісно також наявна.

Такі системи як відеоігри в 99% випадків не складаються з однієї задачі, а є кластером задач, виконання яких формується в ігровий процесс. Таких задач дуже багато, вони бувають одночасно і вбудовані в середовище розробки, і написані сторонніми розробниками. Найголовнішою вбудованою автоматизованою задачею, без якої неможливо уявити жодну відеогру це – графіка, а саме симуляція 2D або 3D простору на екрані ЕОМ. На **(Рис. 5 – Стандартний процесс виведення зображення)** можна побачити через які етапи проходить автоматизована система обробки даних, що потрапляють у об’єктив ігрової камери, для виведення зображення на екран.

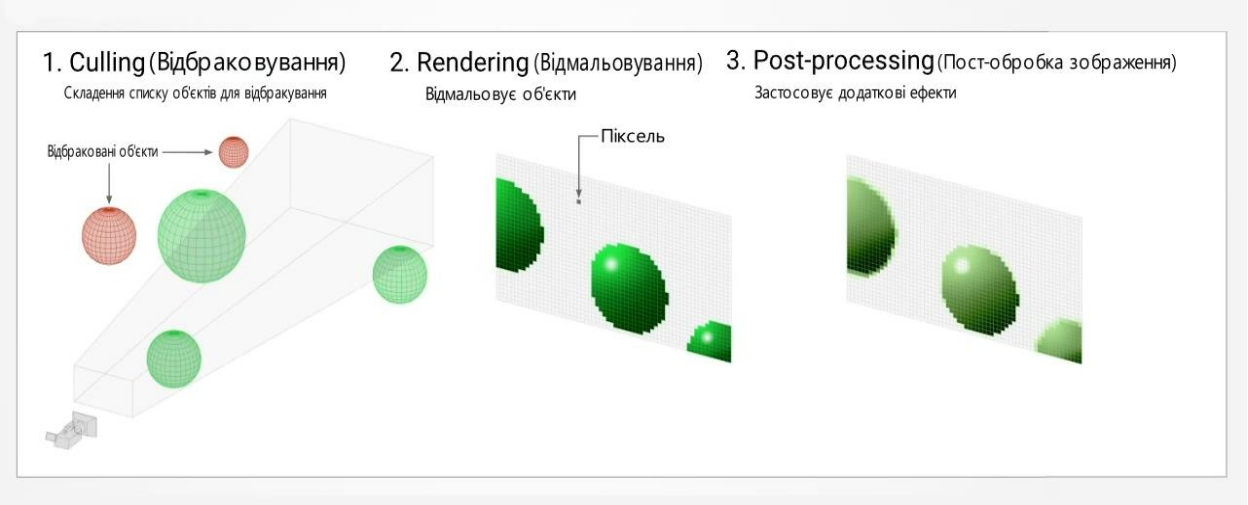


Рисунок 5. Стандартний процесс виведення зображення. [6]

Дана система виведення є автоматизованою та вбудованою в сам ігровий рушій. Ця процедура відбувається кожен кадр в незалежності від того відбувається щось на екрані чи ні, для ЕОМ на екрані завжди щось відбувається. Це самий базовий та необхідний з вбудованих процесів що відбуваються під час гри.

Якщо казати про класичну інформаційну модель то сформувати подібну модель для гри було б дуже складно зробити, якщо взагалі можливо, оскільки ігри описуються в динаміці і залежать не просто від вводу, а ще й від тривалості вводу, це означає що ми не можемо оцінити властивості виведеної інформації по одному десяти чи навіть сотні виводів, ми маємо отримувати сотні виводів на секунду напротязі відносно довгого проміжку часу аби зрозуміти що собою представляє візуальна та цифрова вихідна інформація, одним повідомленням описати всю вихідну інформацію що гра видає за навіть один цикл виводів неможливо, а описувати один єдиний вивід немає сенсу, так як він не буде відображати зв'язок між ланками системи. Проте ми можемо розробити інформаційну модель яка буде ставити так званих "слухачів". Тобто беручи до уваги те, що перебіг інформаційного потоку, а в наслідок і кінцевий вивід системи, може бути перервано чи змінено у будь-який момент ми сформуємо модель яка буде брати до уваги таку можливість, і "перенаправляти" або "зациклювати" потік інформації у випадку якщо "слухач" у зоні своєї дії почув зміну введення. На **(Рис. 6 – Схематична модель гри Cardiarm v0.05a)** зображено модель відеогри з урахуванням корегування вихідної інформації в процесі її обробки.

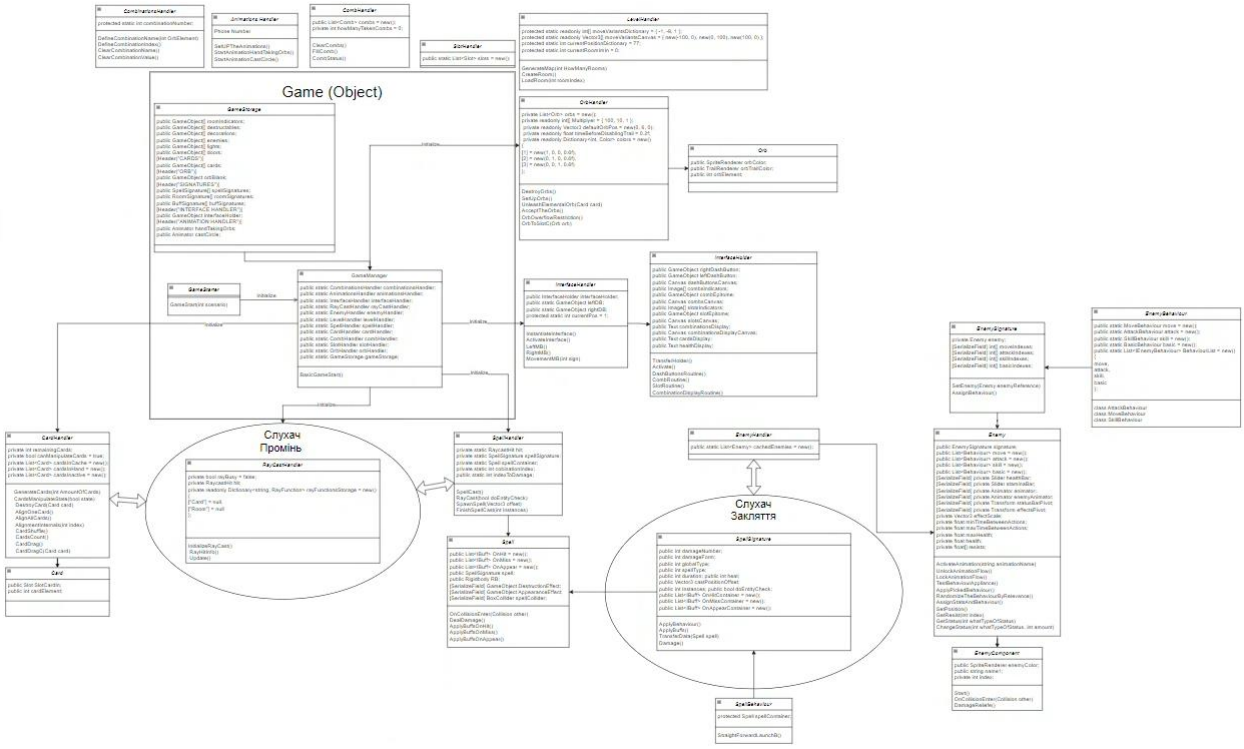


Рисунок 6. Схематична модель гри Cardiarm v0.05a.

Для того щоб ми змогли визначитися як саме буде функціонувати гра та які нам потрібно створювати класи, нам потрібне ядро геймплею. Ядром геймплею будь-якої гри є її керування, або по іншому Input System. Для того аби гра взагалі була грою гравцю необхідно надати інструменти взаємодії зі створеним нами цифровим світом. Оскільки гра в основному орієнтована на мобільні пристрої то в якості керуючої периферії буде виступати екран, і більше нічого. Сенсорні телефони існують доволі давно і способів розрізнення введення існує доволі багато. В табл. 4 наведено приклади існуючих методів введення інформації на сенсорних екранах.

Таблиця 4. Методи вводу.

Назва	*Опис*
Tap (Натиск)	*Самий звичайний натиск, просто відстеження доторку до екрану девайсу*
Multi-Tap (Багаторазовий Натиск)	*Два або більше натисків на екран в одному місці у швидкій послідовності*
Swipe (Гортання)	*Доторк до екрану який неодмінно має бути подовжено невідривним від екрану рухом пальця у певному напрямку, після чого палець обов'язково якомога швидше треба відвести від екрану, інакше введення не буде вважатися завершеним*
Hold (Затискання)	*Доторк до екрану який не повинен бути перерваний відведенням пальця від екрану певний проміжок часу*

Таблиця 4. (продовження)

Drag (Перетягування)	*Доторк до екрану який може бути подовжений переміщенням пальця по екрану і також може бути перерваний у будь який момент*

Як видно з наведеної вище таблиці способів вводу інформація використовуючи лише сенсорний екран достатньо, це не беручи до уваги що є ігри та програми що користуються всього декількома з наведених способів, а то й всього одним.

Щодо вихідної інформації, вона може відрізнятися у різних ігор доволі сильно, проте є так званий “Золотий стандарт”, вихідна інформація що є в абсолютній більшості динамічних ігор. В табл. 5 буде наведено базові типи вхідної інформації яка буде у грі в незалежності від того які дизайнерські та архітектурні рішення мною будуть прийняті у майбутньому.

Таблиця 5. Типи вихідної інформації.

Назва	*Опис*
Damage Dealt (Нанесення пошкоджень)	*Візуальне відображення кількості завданої шкоди*
Combination Name (Назва комбінації)	*Візуалне відображення складеної з елементів комбінації*
Player Statistics (Характеристики гравця)	*Графічне відображення характеристик гравця, як пасивних так і динамічних*
Eney Statistics (Характеристики ворога)	*Графічне відображення характеристик ворога, як пасивних так і динамічних*
Buffs Active (Активні підсилення)	*Візуально відображає активні на даний момент підсилення*

Оскільки моя гра не використовує БД для запису та ще й не є класичною ІС то і повідомлення що надсилються в систему, теж класичними бути не можуть. Всього глобально в грі на даний момент існує 2 типа введення, задопомогою променя і задопомогою базових UI елементів Unity, таких як наприклад кнопка. В табл. 6 наведено типи повідомлень що допомагають гравцю взаємодіяти з грою. Ці “повідомлення” використовуються як спосіб маніпулювання персонажем та його спорядженнням в грі.

Таблиця 6. Перелік і опис вхідних повідомлень.

З / П	Назва вхідного повідомлення	Ідентифікатор	Форма подання	Термін і частота надходження	Джерело
1	rayFunctionCard	R_FUNC_CARD	Input.MouseButton(0)	Залежить від користувача	RayCastHandler
2	rayFunctionRoom	R_FUNC_ROOM	Input.MouseButton(0)	Залежить від користувача	RayCastHandler
3	ButtonUI	BTTN	IPointerClickHandler	1-5 сек.	UnityInputOutputSystem

Також треба дати перелік вихідних повідомлень, хоча у моїй грі немає логу повідомлень, тим не менш його можна сформувати, хоч з практичної точки зору така таблиця ніяк не опише функціонування системи, проте полегшить роботу тестувальникам та програмістам які захочуть перевірити корректність роботи гри. В табл. 8 зображена таблиця повідомлень що вспливають у консолі у випадку вдалого відпрацювання важливих системних функцій, таких як складання комбінації, відпрацювання ворожого ШІ та ін.

Таблиця 7. Перелік і опис вихідних повідомлень.

З / П	Назва вихідного повідомлення	Ідентифікатор	Форма подання і вимоги до неї	Періодичність видання	Термін видання і допустимий час затримки	Користувачі інформації
1	Нанесена Шкода	DMG_DLT	Float (#.##)	Залежить від введення	Миттєво, Затримки неприпустимі	Гравець
2	Отримана Шкода	DMG_RCV	Float (#.##)	Залежить від введення	Миттєво, Затримки неприпустимі	Гравець
3	Виконана Ворогом Дія	AI_ACT	String	1-5 сек.	Меньше 0.1 сек., 0.2 сек.	Гравець
4	Зібрана Комбінація	COM_NAME	String	Безперервна	Миттєво, 0.1 сек.	Гравець
5	Кількість Карт	CARDS	Int (+)	Безперервна	Миттєво, 0.1 сек.	Гравець
6	Кількість Здоров'я	HLT	Float (#.##)	Безперервна	Миттєво, 0.1 сек.	Гравець
7	Кількість Досвіду	EXP	Int (+)	Безперервна	Миттєво, 0.1 сек.	Гравець

Згідно отриманих даних можна сформувати більш класичний варіант на цей раз інформаційної моделі гри **(Рис. 7.1 - Інформаційна модель гри Cardiarm v0.05a.)**.

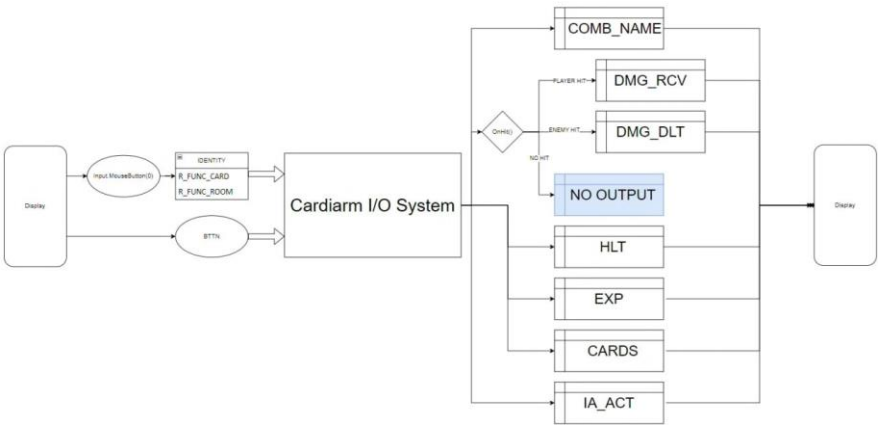


Рисунок 7.1 Інформаційна модель гри Cardiarm v0.05a.

2.2.1 Проблеми побудови геймплею

- *Проблема 1:*

Ми вже знаємо що основними проблемами для побудови карткової гри на базі фізичної систем прогресії є надмірність та складність сприйняття.

Складність сприйняття виражається в тому факті що кожа карта має унікальні здібності і як правило у добре сформованій гравцем колоді кожна карта має декілька комбінацій використання з іншими картами щоб зробити максимальну кількість комбінацій карт у руці корисною, чи наприклад мати змогу підлаштуватись під якомога більшу кількість ігрових ситуацій. Кажучи простіше, гравцю треба подумати. А надмірність виражається в тому що якщо зверху ще й додати прив'язку до фізичної прогрессії з властивою їй купою характеристик кожна з яких має свої коефіцієнти множення та принципи взаємодії то гра перетвориться на домашню роботу з алгебри, а в першу чергу гра має бути веселою.

Зрозуміло, потрібно зменшити когнітивне навантаження на гравця, залишивши суть в розіграванні карток, тобто складання комбінацій карт з тих що лежать у руці з метою отримати найбільшого профіту. Головною складовою карткового геймплею, що і забирає найбільшу кількість часу на роздуми, є текстовий опис здібностей карт. Я хочу позбавитися від цієї механіки, для того щоб дозволити гравцю звертати увагу не тільки на те що лежить у нього в руці а і на те що відбувається перед обличчям.

Для цього мною була розроблена система *елементального сплетення*, докладніше про те як вона буде реалізована написано у концепт документі **(Розділ 2.3)**, зараз же ми розуміємо які саме задачі така система вирішує. Вона дозволяє гравцю зрозуміти які в нього є варіати та які комбінації він може скласти, зробивши всього один погляд на карти в руці **(Рис. 7 Вигляд елементальних карт на момент версії 0.06a)**. Таким чином також кількість карт в руці необхідно обмежити п'ятьма. Для того щоб гравцю не потрібно було рахувати кількість карт в руці, а положення карт було завжди таким яким воно було при першому погляді гравця на них.

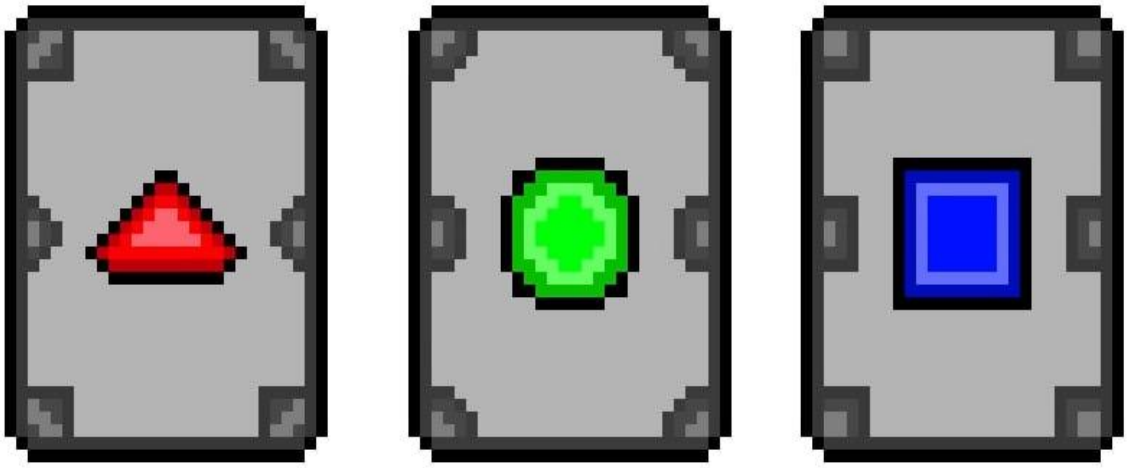


Рисунок 7. Вигляд елементальних карт на момент версії 0.06а.

І тут вступає в діло *перший принцип* побудови геймплею – постійна маніпуляція відчуттям терміновості, концентрація гравця повинна постійно піддаватися раціональним сплескам, а геймплей має бути нерівномірним в плані емоціонального навантаження.

З цього принципу випливає що тепер нам потрібно заповнити прогалину в геймплеї. Ми звільнили час необхідний гравцю для оцінки своїх можливостей і прогнозування дій, таким чином ми зняли велику порцію когнітивного навантаження на гравця, гра стала легшою, і стала потребувати меньше концентрації для отримання бажаного результату.

Зазвичай в карткових іграх ворог нікуди не намагається втікти і не намагається скоротити з нами дистанцію, в абсолютно всіх іграх подібного жанру ворог - це, по суті, такий самий гравець як і ми, тільки карти в нього інші, а бій проходить по принципу покрокової дуелі. Навіть чудовиська, з якими ми боремось, розігрують карти. Це є доцільною геймплейною механікою коли карти це лише зручний та зрозумілий спосіб описання та структурування внутрішньоігрових здібностей та механік. А репрезентують вони цілком реальні дії, будь то удар ногою, мечем, палицею чи вогняним закляттям, просто описані в такій манері. В своїй грі я хочу відійти від цього концепту і цим вбити одразу двох зайців. Імплементувавши в гру систему переміщення та унікальних атак.

- *Проблема 2:*

Ця проблема випливає з вирішення першої, якщо ми маємо ворога який може переміщуватись, і має унікальні анімовані таки то нам необхідно створити для нього штучний інтелект. Звісно в класичник ККІ (колекційних карткових іграх) у так званих ботів є ШІ, але ці ШІ представляють собою набір найпростіших закономірностей та послідовностей які можуть описуватися буквально декількома одновимірними масивами. У нас же з вами йде мова саме про ворога дії якого є фізичними, тобто вони напряму взаємодіють не тільки з гравцем чи ігровим полем, а з усім ігровим оточенням. Крім того гравець безумовно теж отримає можливість переміщуватися, що ставить вимоги для розробки дійсно якісного штучного інтелекту на значно вищий рівень. І тепер нам необхідно придумати як саме цей штучний інтелект реалізувати. Насправді у нас є лише два самі реалістичні шляхи розробки ШІ - сценарна та вагова моделі.

- *Сценарна модель ШІ:*

Сценарна модель це мій варіант способу повторити штучний інтелект що використовується в ККІ, про який я згадував вище. Суть його в тому що ми маємо певнй набір послідовних чисел, що заздалегідь записані в якусь змінну (Action Array). Кожне число репрезентує якусь дію яку повинен виконати ворог. На початку відпрацювання скрипту ворога запускається цикл котрий починає перебирати числа яким в скрипті ворога відповідають певні дії, після чого проходить превірка чи виконання дії можливе, наприклад:

Атака:

- Чи можливо дістати до ворога цією атакою?
- Чи не перешкоджає що-небудь влучанню атаки?

Переміщення:

- Чи можливе переміщення у цьому напрямку?

Здібність:

- Чи буде використання здібності релевантним?

Також універсальною перевіркою для усіх дій є чи достатньо у ворога поінтів дії, кожна дія витрачає певну кількість цих поінтів, з часом вони

відновлюються, така система необхідна аби обмежити використання ворогом декількох дуже сильних атак поспіль. Крім того кожна дія має свою унікальну перевірку релевантності, завдяки чому досягається високий рівень заглиблення в ігровий процес, оскільки ворог реагує на твої дії.

Якщо виконати дію можливо, то саме це і відбувається після чого починає відпрацьовувати метод що перевіряє:

- Чи досі даний сценарій дій є релевантним.
- Чи не закінчився сценарій.

Якщо сценрій перестав бути релевантним або закінчився то функція поверне нас до початку методу, де почнеться вибір нового сценарію. Якщо ж сценарій досі релевантний то буде віддана на перевірку наступна дія з масиву, і так по колу.

Релевантність сценарію, як власне і сам сценарій (послідовність дій) я, як розробник, маю визначити сам. Тобто для кожного ворога я повинен буду власноруч написати послідовність дій які ворог має виконати, у тих чи інших ситуаціях, і подібного роду сценарії може у одного ворога бути декілька десятків, якщо ми хочемо щоб супротивник дійсно становив хоч якусь загрозу.

Цей метод є більш легким не тільки в плані архітектурному, але і в плані навантаження на систему, по суті ми просто по черзі запускаємо анімації ворога, перед цим перевіряючи, чи можливо їх програти **(Рис. 8 – Схема відпрацювання сценарної моделі ШІ)**.

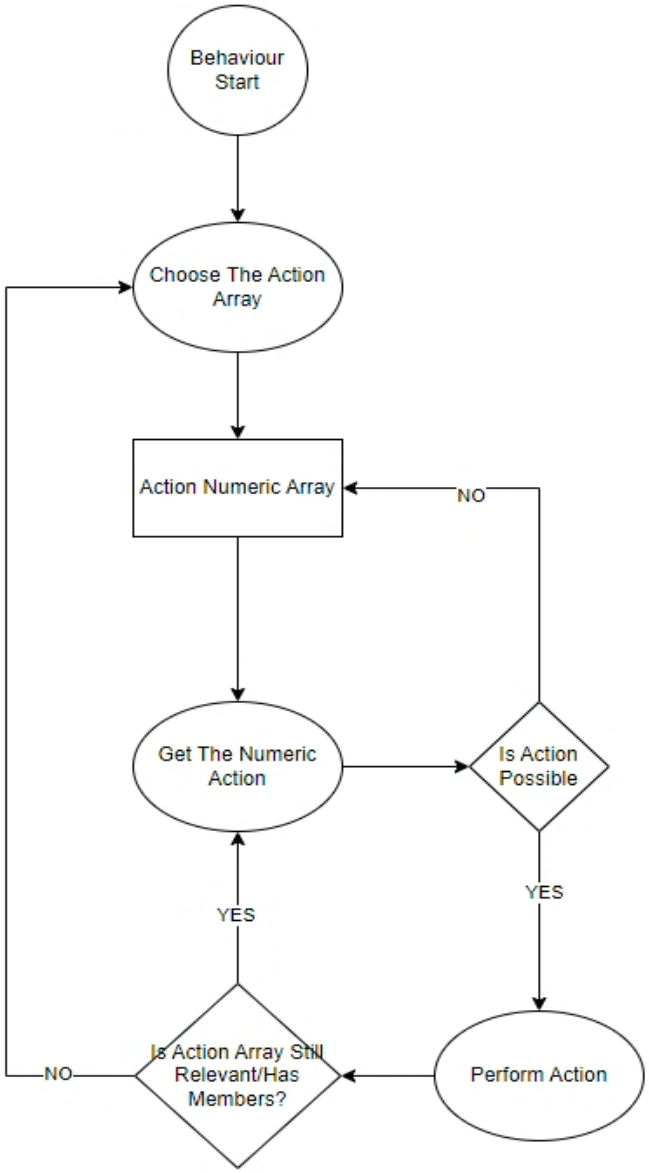


Рисунок 8. Схема відпрацювання сценарної моделі ШІ

- *Вагова модель ШІ:*

Дана модель **(Рис. 9 та 10 - Вручну змодельована схема відпрацювання вагової моделі ШІ та Автоматично змодельована схема відпрацювання вагової моделі ШІ)** представляє собою повноцінний штучний інтелект просто в зменшеному варіанті. Суть в тому, що перевірка на так звану вагу (релевантність) відбувається для абсолютно кожної дії, з тих, що доступні ворогу, після чого випадковим чином вибирається дія, але шанс вибору напряму залежить від ваги цієї дії. Звучить просто, але на справді це значно більш складна з архітектурної точки зору система, яка також потребує значно більше обчислювальних потужностей.

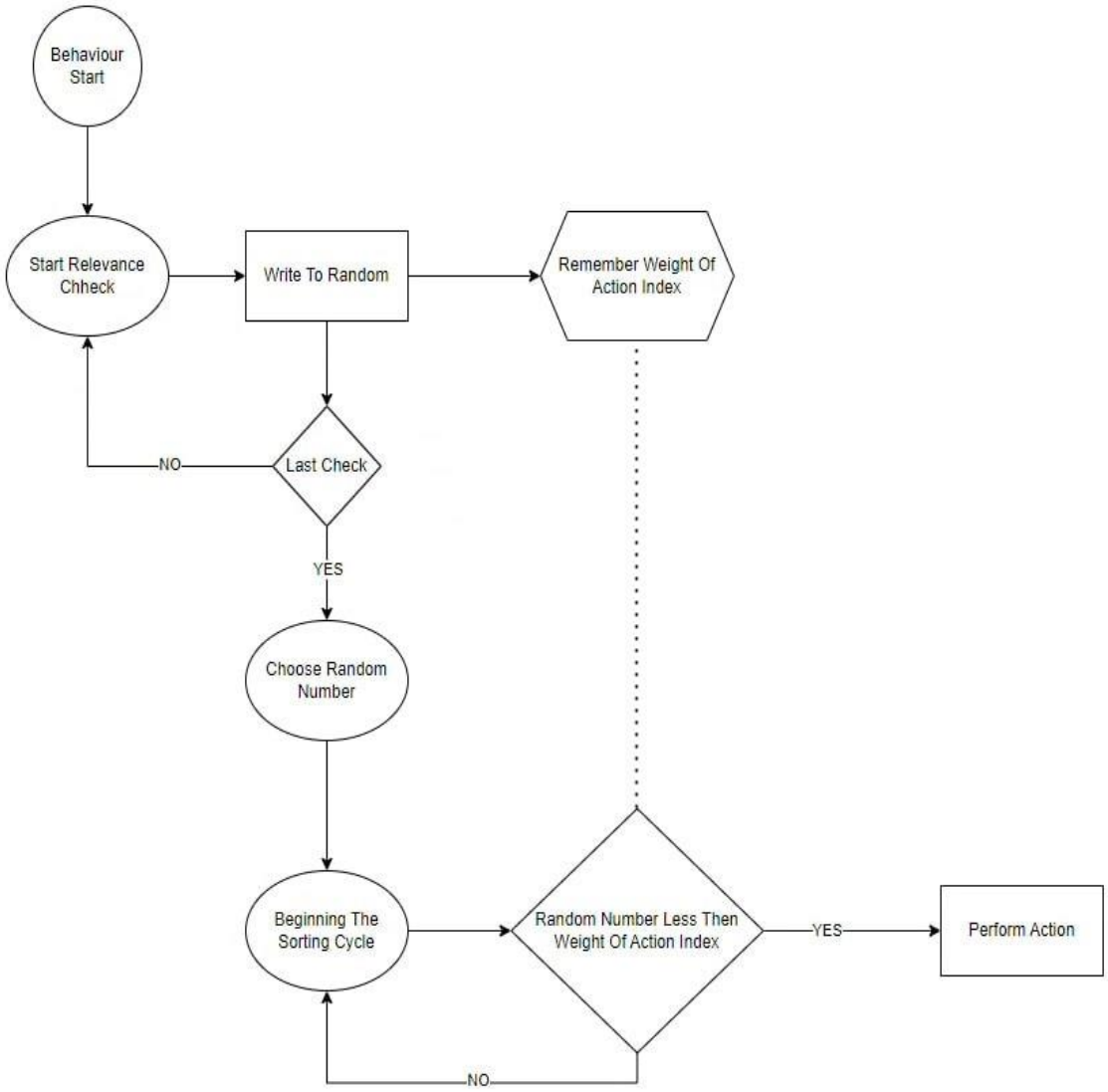


Рисунок 9. Вручну змодельована схема відпрацювання вагової моделі ШІ.

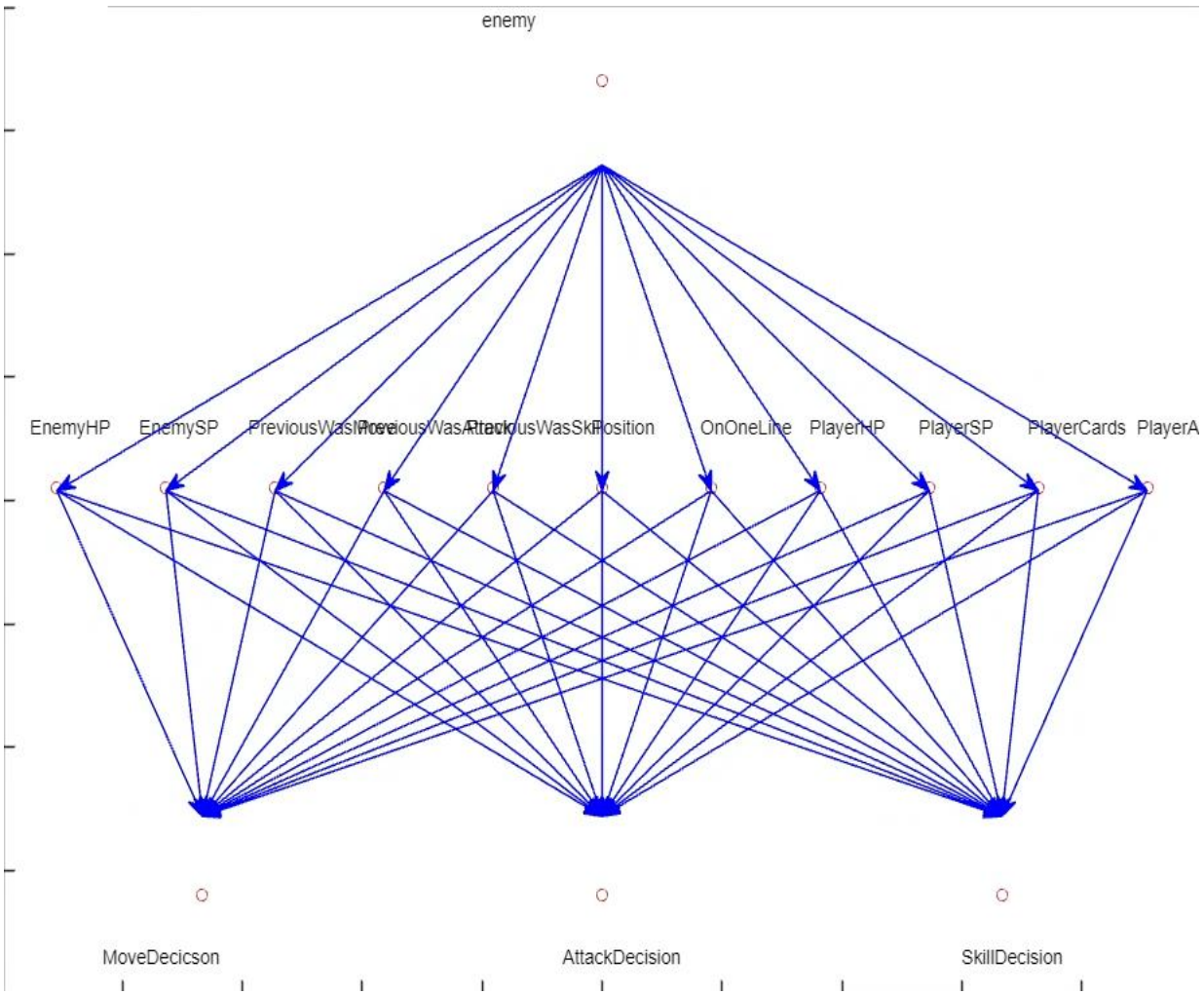


Рис 10. Автоматично змодельована схема відпрацювання вагової моделі ШІ.

Нижче наведено код автоматично змодельованої схеми відпрацювання вагової моделі.

```
SN=SNnew;
SN=SNaddORnode(SN, 'MoveDecicson','AttackDecision','SkillDecision');
SN=SNaddANDnode(SN, 'enemy', 'EnemyHP', 'EnemySP', 'PreviousWasMove',
'PreviousWasAttack', 'PreviousWasSkill', 'Position', 'OnOneLine', 'PlayerHP',
'PlayerSP', 'PlayerCards', 'PlayerAttackState');
SN=SNaddrelation(SN, 'enemy', 'has', 'EnemyHP');
SN=SNaddrelation(SN, 'enemy', 'has', 'EnemySP');
SN=SNaddrelation(SN, 'enemy', 'has', 'PreviousWasMove');
SN=SNaddrelation(SN, 'enemy', 'has', 'PreviousWasAttack');
SN=SNaddrelation(SN, 'enemy', 'has', 'PreviousWasSkill');
SN=SNaddrelation(SN, 'enemy', 'has', 'Position');
SN=SNaddrelation(SN, 'enemy', 'has', 'OnOneLine');
SN=SNaddrelation(SN, 'enemy', 'has', 'PlayerHP');
```

```
SN=SNaddrelation(SN, 'enemy', 'has', 'PlayerSP');
SN=SNaddrelation(SN, 'enemy', 'has', 'PlayerCards');
SN=SNaddrelation(SN, 'enemy', 'has', 'PlayerAttackState');
SN=SNaddrelation(SN, 'EnemyHP', 'type', 'AttackDecision');
SN=SNaddrelation(SN, 'EnemyHP', 'type', 'SkillDecision');
SN=SNaddrelation(SN, 'EnemyHP', 'type', 'MoveDecicson');
SN=SNaddrelation(SN, 'EnemySP', 'type', 'AttackDecision');
SN=SNaddrelation(SN, 'EnemySP', 'type', 'SkillDecision');
SN=SNaddrelation(SN, 'EnemySP', 'type', 'MoveDecicson');
SN=SNaddrelation(SN, 'PreviousWasMove', 'type', 'AttackDecision');
SN=SNaddrelation(SN, 'PreviousWasMove', 'type', 'SkillDecision');
SN=SNaddrelation(SN, 'PreviousWasMove', 'type', 'MoveDecicson');
SN=SNaddrelation(SN, 'PreviousWasSkill', 'type', 'AttackDecision');
SN=SNaddrelation(SN, 'PreviousWasSkill', 'type', 'SkillDecision');
SN=SNaddrelation(SN, 'PreviousWasSkill', 'type', 'MoveDecicson');
SN=SNaddrelation(SN, 'PreviousWasAttack', 'type', 'AttackDecision');
SN=SNaddrelation(SN, 'PreviousWasAttack', 'type', 'SkillDecision');
SN=SNaddrelation(SN, 'PreviousWasAttack', 'type', 'MoveDecicson');
SN=SNaddrelation(SN, 'Position', 'type', 'AttackDecision');
SN=SNaddrelation(SN, 'Position', 'type', 'SkillDecision');
SN=SNaddrelation(SN, 'Position', 'type', 'MoveDecicson');
SN=SNaddrelation(SN, 'OnOneLine', 'type', 'AttackDecision');
SN=SNaddrelation(SN, 'OnOneLine', 'type', 'SkillDecision');
SN=SNaddrelation(SN, 'OnOneLine', 'type', 'MoveDecicson');
SN=SNaddrelation(SN, 'PlayerHP', 'type', 'AttackDecision');
SN=SNaddrelation(SN, 'PlayerHP', 'type', 'SkillDecision');
SN=SNaddrelation(SN, 'PlayerHP', 'type', 'MoveDecicson');
SN=SNaddrelation(SN, 'PlayerSP', 'type', 'AttackDecision');
SN=SNaddrelation(SN, 'PlayerSP', 'type', 'SkillDecision');
SN=SNaddrelation(SN, 'PlayerSP', 'type', 'MoveDecicson');
SN=SNaddrelation(SN, 'PlayerCards', 'type', 'AttackDecision');
SN=SNaddrelation(SN, 'PlayerCards', 'type', 'SkillDecision');
SN=SNaddrelation(SN, 'PlayerCards', 'type', 'MoveDecicson');
SN=SNaddrelation(SN, 'PlayerAttackState', 'type', 'AttackDecision');
SN=SNaddrelation(SN, 'PlayerAttackState', 'type', 'SkillDecision');
SN=SNaddrelation(SN, 'PlayerAttackState', 'type', 'MoveDecicson');
SNplot(SN, 'hierarchy');
figure;
SNplot(SN, 'circle');
figure;
SNplot(SN, 'circle');
SN1=SN;
```

```
SN1=SNdelnode(SN1, 'EnemyHP', 'EnemySP', 'PreviousWasMove', 'PreviousWasAttack',
'PreviousWasSkill', 'Position', 'OnOneLine', 'PlayerHP', 'PlayerSP', 'PlayerCards',
'PlayerAttackState');
SN1=SNaddANDnode(SN1, '?');
SN1=SNaddrelation(SN1, 'enemy', 'type', '?');
Res=SNfind(SN, SN1);
```

На цій схемі зображено спрощену модель функціонування даного ШІ, оскільки не береться в розрахунок величезна кількість формул релевантності, та внутрішня реалізація системи розподілених дій M.A.S.B (Move Attack Skill Basic) через що у спрощеному вигляді продемонстрована система перебору масивів з індексами дій для визначення приналежності дії до вибраного випадкового індексу.

Дана модель ШІ є наймовірно гнучкою, адаптивною та модульною, до неї навіть можна додати модуль самонавчання. Так чи інакше, як вже було сказано, дана модель дозволяє ворогу відпрацьовувати не по шаблонам а у відповідності з діями гравця у реальному часі, що робить цю модель моїм фаворитом. Хоча треба зазначити що гра не тестувавалася на слабких девайсах, і використання змішаної версії цих двох моделей може знадобитися, саме тому я розповів про обидві не дивлячись на те що з самого початку знав яка краща.

На момент дописування цього бакалаврського проекту мною вже була спроектована більш проста та не менш ефективна система перебору що не прив'язана напряму до M.A.S.B, але вона в основному сфокусована на оптимізації, читабельності та легкості обслуговування коду і досі знаходиться на випробувальному терміні.

- *Проблема 3:*

Наступна проблема є універсальною для будь-якого розробника будь-якого ПО, але для мобільного ПО ця проблем є одним зі стовпів ігнорування якого приведе до того що проект просто не буде мати ніякої цінності, так, я зараз кажу про оптимізацію. І в цьому відрізку роботи я би хотів поговорити про оптимізацію самої важкої для мобільних платформ, та й не тільки для них,

частини гри – графіки та інстансингу (створення сутностей).

Моя гра має інтерактивну карту **(Рис. 11 - Робочий прототип інтерактивної карти рівня)**. Карта складається з окремих підсвічених та не підсвічених клітиночок кожна з яких репрезентує кімнату на рівні, і через карту в цю кімнату можна зайти.

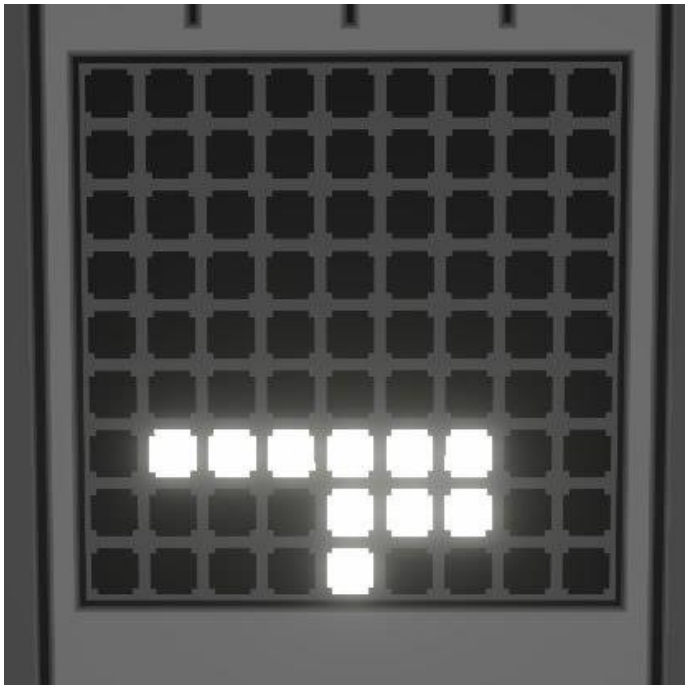


Рисунок 11. Робочий прототип інтерактивної карти рівня.

І постає питання, яким же чином на мобільному девайсі оптимізувати подібного типу механіку з кімнатами. Ми очевидно не можемо собі дозволити згенерувати всі кімнати, всіх ворогів всю внутрішню фурнітуру кімнати і просто розташувати це все на рівні. Абсолютна більшість мобільних девайсів не володіє достатньою кілкістю оперативної пам’яті для того щоб забезпечити комфортну гру в таких умовах. Також ми не можемо генеруваи кожен раз кімнату з нуля при переході з кімнати в кімнату, бо навіть якщо мобільні девайси і зможуть кожен раз відмаловувати нову кімнату, то те як телфон буде нагріватись навіть страшно уявити, що зробить гру максимально некомфортною. У зв’язку з цим була придумана система *“Teamp”*, що вона собою представляє: Ми не будемо створювати одразу всі об’єкти що є на рівні, ми створимо одну кімнату, і ті об’єкти які є на цьому рівні в кількості

максимально необхідній для складання з них будь-якої кімнати на рівні. Тобто якщо у нас на рівні так вийшло що немає кімнат в яких більше 2 дверей, то на початку рівня буде створено 2 двері. Якщо в кімнаті одні двері, то одна з 2-х заготовлених дверей стане на місце тих одних дверей. Таким чином у нас не буде 30 записаних у пам’яті дверей, у нас буде рівно стільки дверей скільки нам потрібно для того щоб ми могли зібрати будь яку кімнату на рівні. Таким чином ігровий простір перетворюється на так званий театр, в якому коли закривається ширма (відбувається завантаження наступної кімнати), всі непотрібні об’єкти прибираються зі сцени вбік а всі необхідні виставляються на показ, відповідно як і декорації **(Рис. 12 – Спрощена модель роботи системи “Театр”)**.

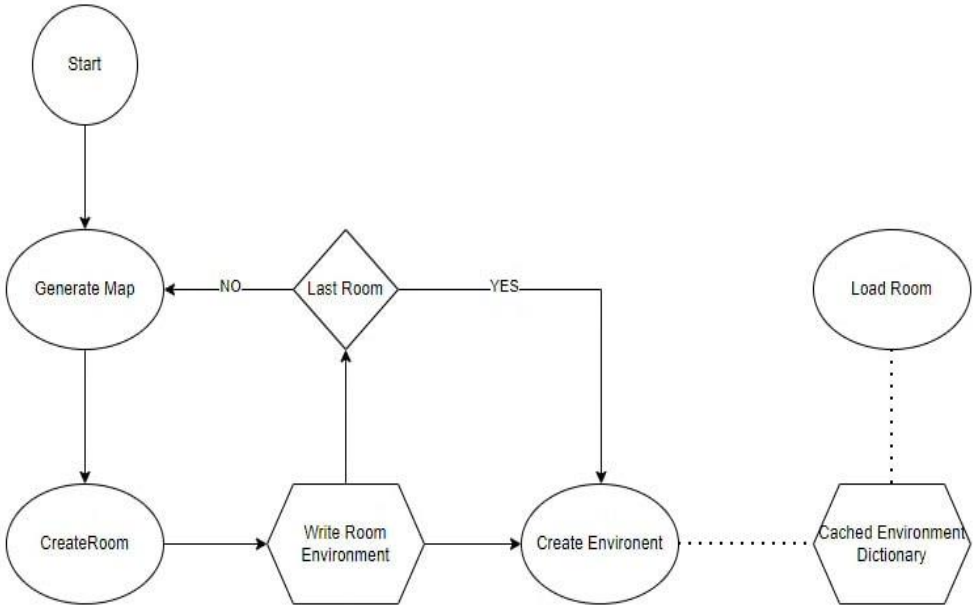


Рисунок 12. Спрощена модель роботи системи “Театр”.

Кожний елемент інтер’єру має свій власний індекс у спеціальному масиві (Environment), у нас буде другий такий самий за індексацією подвійний масив (Cached Environment Dictionary), тільки в ньому ми будемо зберігати не фізичну заготовку, а посилання на вже створені об’єкти які будемо викликати у випадку необхідності.

2.2.2 Математичне забезпечення та алгоритми розв’язання задач

На **(Рис. 13 – Потік ігрового процесу)** зображено організацію ігрового потоку, тобто схему того як організовано взаємодію гравця з грою, перетікання одних етапів у інші, вхідні та вихідні точки програми, а також логіку деяких транзицій між етапами геймплею.

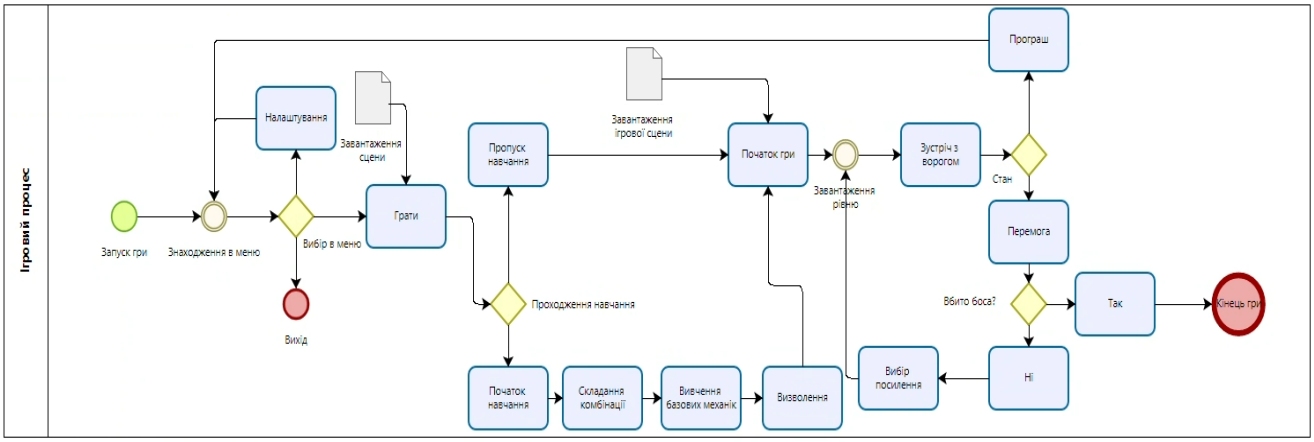


Рисунок 13. Потік ігрового процесу

На **(Рис. 14 – Організація переміщення сфер)** зображено організацію переміщення елементальної сфери у слот у вигляді графіку умовної конструкції.

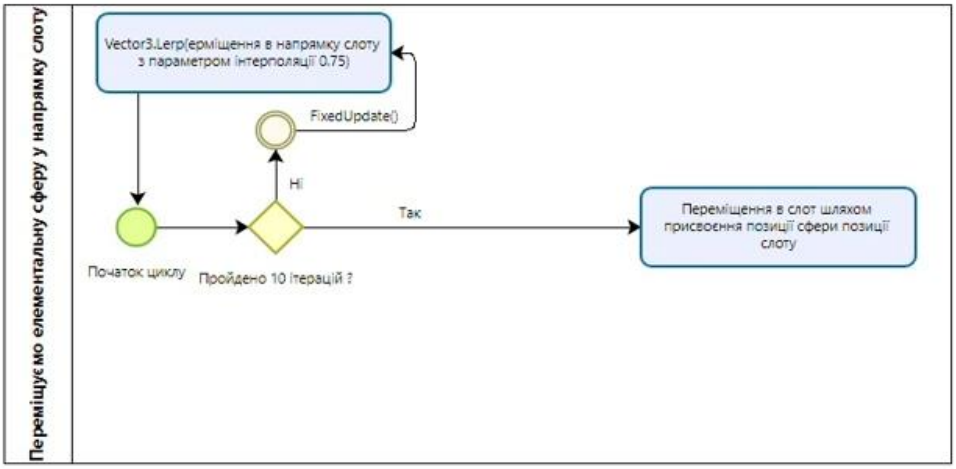


Рисунок 14. Організіція переміщення сфери

Послідовність функції виконання переміщення сфери, цікава не з точки зору що вона робить, а як вона це робить, можна побачити, що по завершенню циклу коли в нас всі 10 ітерацій пройшли ми прирівнюємо положення слоту і положення сфери, таким чином у нас сфера має те саме положення що і слот,

раніше ця операція була обов’язковою оскільки робилася перевірка на те чи дійшла сфера до слоту і чи можна зараховувати її як частину комбінації, це було зроблено для того аби гравець не міг використати комбінацію до того як сфера візуально зайде у слот. Проте така система була надлишковою та не ефективною, тому вона з часом була оптимізована і замінена на систему підрахунку слотів. Але чому прирівняння залишилось, все просто, функція інтерполяції не здатна прирівняти рухомий об’єкт та точку в напрямку якої він рухається, значення будуть наближтися один до одного поки кількість знаків після коми не перетне вмістимість пам’яті змінної. Саме тому у даного циклу фіксована кількість ітерацій. І завдяки прирівнюванню сфери і слоту, в не залежності від того де сфера була створена, за десять кадрів вона обов’язково опинется в слоті, а інтерполяція зробить цей рух рівномірним. Також дуже корисну функцію виконує затримка FixedUpdate(). В грі кількість кадрів може змінюватись в залежності від конфігурації системи на якій її запустили, а обмежувати кількість кадрів може бути поганою ідеєю так як на моніторах з більшою частотою оновлення ніж кількість кадрів в грі буде дуже рвана картинка. Якби у нас було не 60 кадрів а 300, то даний цикл програвався б не за 1/6 секунди а за 1/30, тобто на більш потужних девайсах гра відчувалася б в 5 разів швидшею. але FixedUpdate() відпрацьовує рівно 60 кадрів на секунду, таким чином ми очікуємо між ітераціями 1/60 секунди і нам байдуже в скільки кадрів йде сама гра, візуально та фізично сфера буде переміщуватися з однаковою швидкістю при будь якій кількості кадрів.

Найцікавішим на мою думку моментом цього процесу є використання змінної HowManySlots, не дивлячись на те що це буквально int змінна, вона дозволяє зекономити фантастичну кількість процессорного часу та полегшити задачу визначення комбінації **(Рис. 15 та 16 - Принцип роботи змінної HowManySlots та процес визначення назви комбінації)**. Оскільки раніше доводилося брати за допомогою важкої функції GetComponent положення слоту, дана оперерація була доволі повільною. HowManySlots дозволив помістити всі слоти в масив, а значення HowManySlots тепер виступає індексом необхідного слоту.

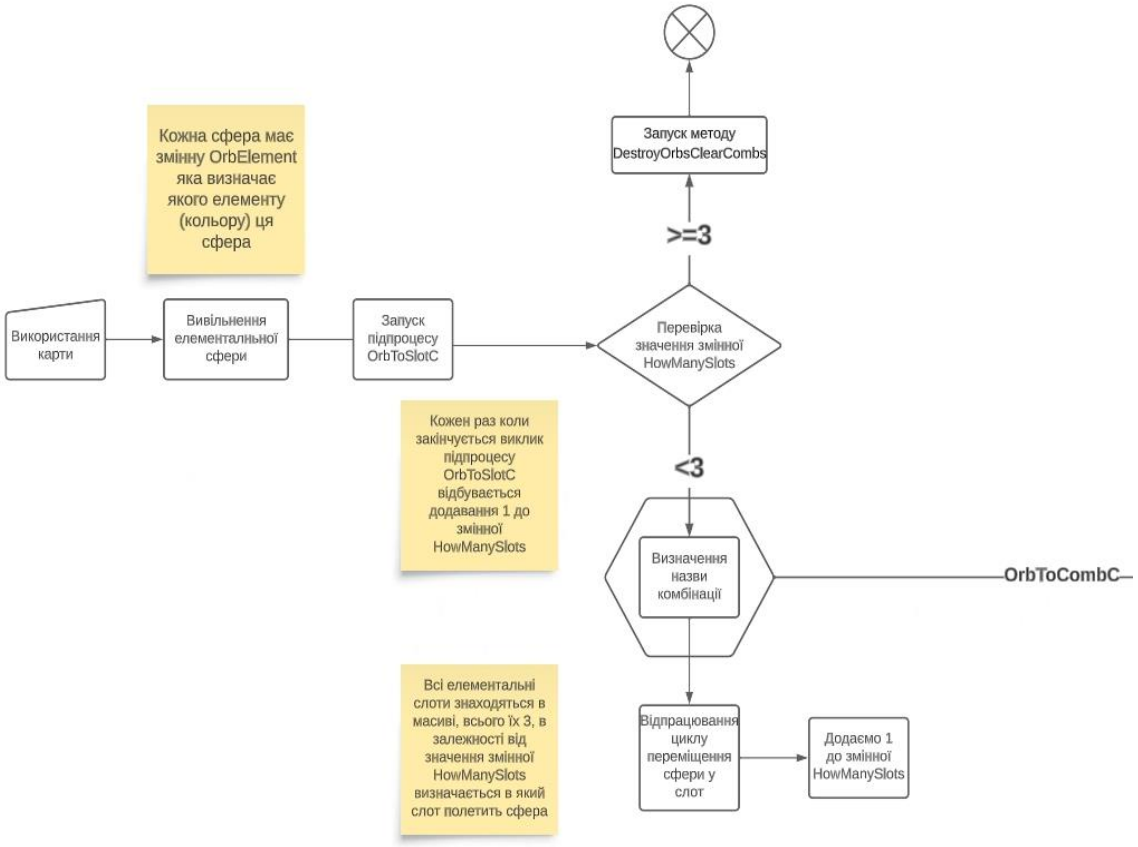


Рисунок 15. Принцип роботи змінної HowManySlots.

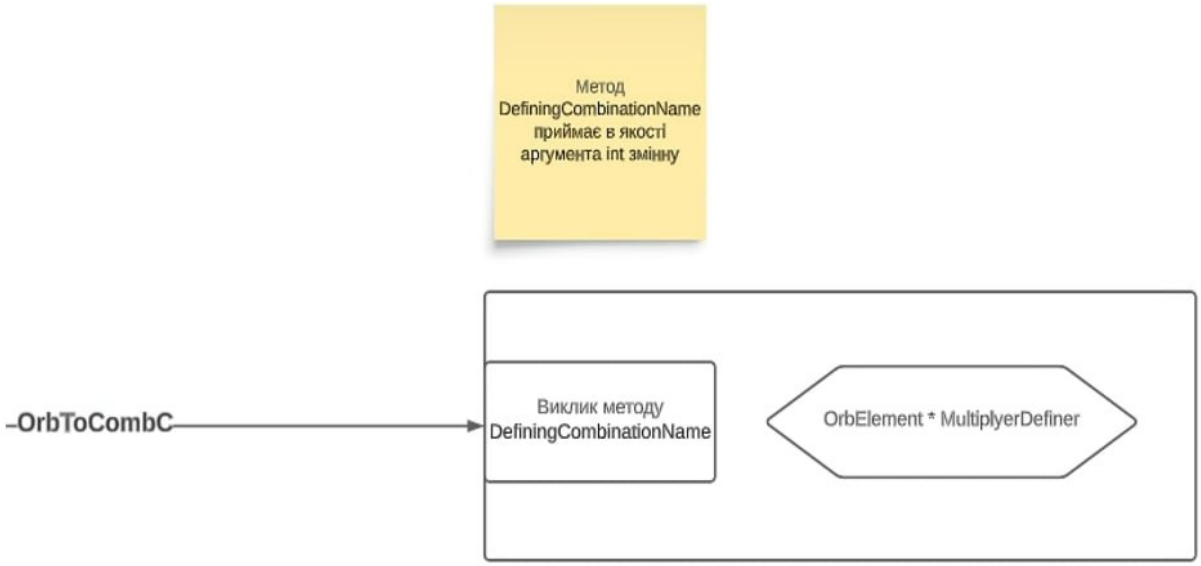


Рисунок 16. Процес визначення назви комбінації.

За допомогою змінної HowManySlots ми ще й можемо формувати запит для отримання комбінації.

За допомогою цієї формули яка знаходиться в методі MultiplyerDefiner:

$$X * 100 /10^Y$$

Де:

X – OrbElement (int);

Y – HowManySlots (int);

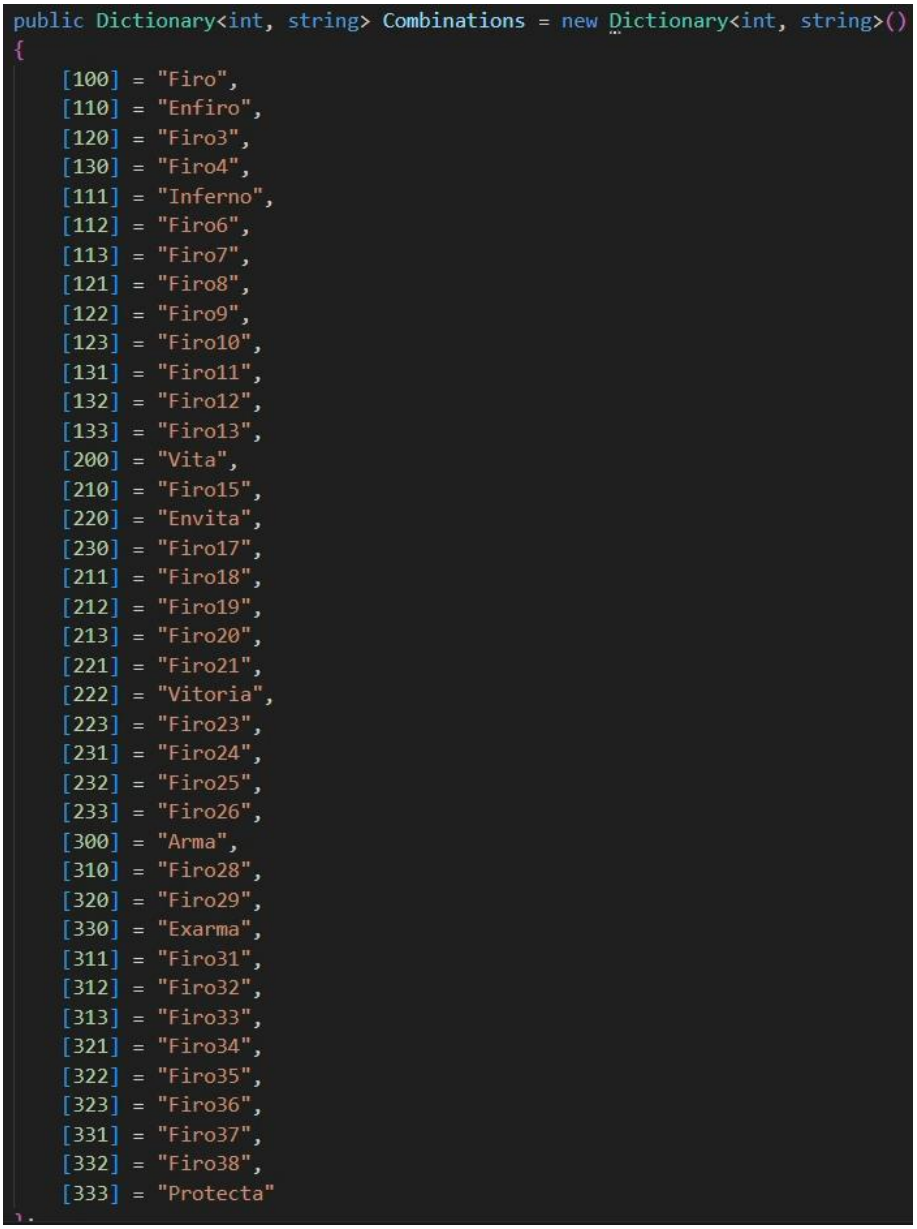
Суть в тому що нам необхідно отримати 3-х значне число в якому Одиниця – Елемент Фіро, Двійка – Елемент Віта, Трійка – Елемент Арма, Нуль – Пустий Слот. Таким чином:

Якщо у нас першим йде Фіро то ми отримуємо 100;

Якщо у нас другим йде Арма то ми отримуємо 20;

Якщо у нас третім йде Арма то ми отримуємо 2;

Після чого ми додаємо ці значення і отримуємо комбінацію з порядковим номером 122, після чого метод DefiningCobinationName проводить надшвидкий пошук по Словнику(Dictionary) Cobinations **(Рис. 17 - Список усіх на даний момент можливих комбінацій)** просто порівнюючи значення з Хеш-Таблицею. Такм чином ми отримаємо комбінацію під тимчасовою кодовою назвою “Firo9”



```
public Dictionary<int, string> Combinations = new Dictionary<int, string>()
{
    [100] = "Firo",
    [110] = "Enfiro",
    [120] = "Firo3",
    [130] = "Firo4",
    [111] = "Inferno",
    [112] = "Firo6",
    [113] = "Firo7",
    [121] = "Firo8",
    [122] = "Firo9",
    [123] = "Firo10",
    [131] = "Firo11",
    [132] = "Firo12",
    [133] = "Firo13",
    [200] = "Vita",
    [210] = "Firo15",
    [220] = "Envita",
    [230] = "Firo17",
    [211] = "Firo18",
    [212] = "Firo19",
    [213] = "Firo20",
    [221] = "Firo21",
    [222] = "Vitoria",
    [223] = "Firo23",
    [231] = "Firo24",
    [232] = "Firo25",
    [233] = "Firo26",
    [300] = "Arma",
    [310] = "Firo28",
    [320] = "Firo29",
    [330] = "Exarma",
    [311] = "Firo31",
    [312] = "Firo32",
    [313] = "Firo33",
    [321] = "Firo34",
    [322] = "Firo35",
    [323] = "Firo36",
    [331] = "Firo37",
    [332] = "Firo38",
    [333] = "Protecta"
};
```

Рисунок 17. Список усіх на даний момент можливих комбінацій (у більшості комбінацій назва все ще тестова)

Ефект руйнування карт

Коли прототип тільки був у планах мені було необхідно подивитися на реалізацію деяких механік та візуальних ефектів. На той час я ще не в повній мірі освідомлював на наскілки великий шмат роботи я замахнувся, тому якщо не брати до уваги візуальну реалізацію елементалних сфер то ефект розпадання карт це єдиний візуалний ефект який на той момент був в грі, і зараз я розповім як я відтворив його всередині Unity без використання сторонніх програм **(Рис. 18 та 19 – Вигляд об’єкту в редакторі та його**

ієрархія). Суть ефекту була в тому щоб імітувати знищення та розліт шматків картки при її перетаскуванні в зону заліку.

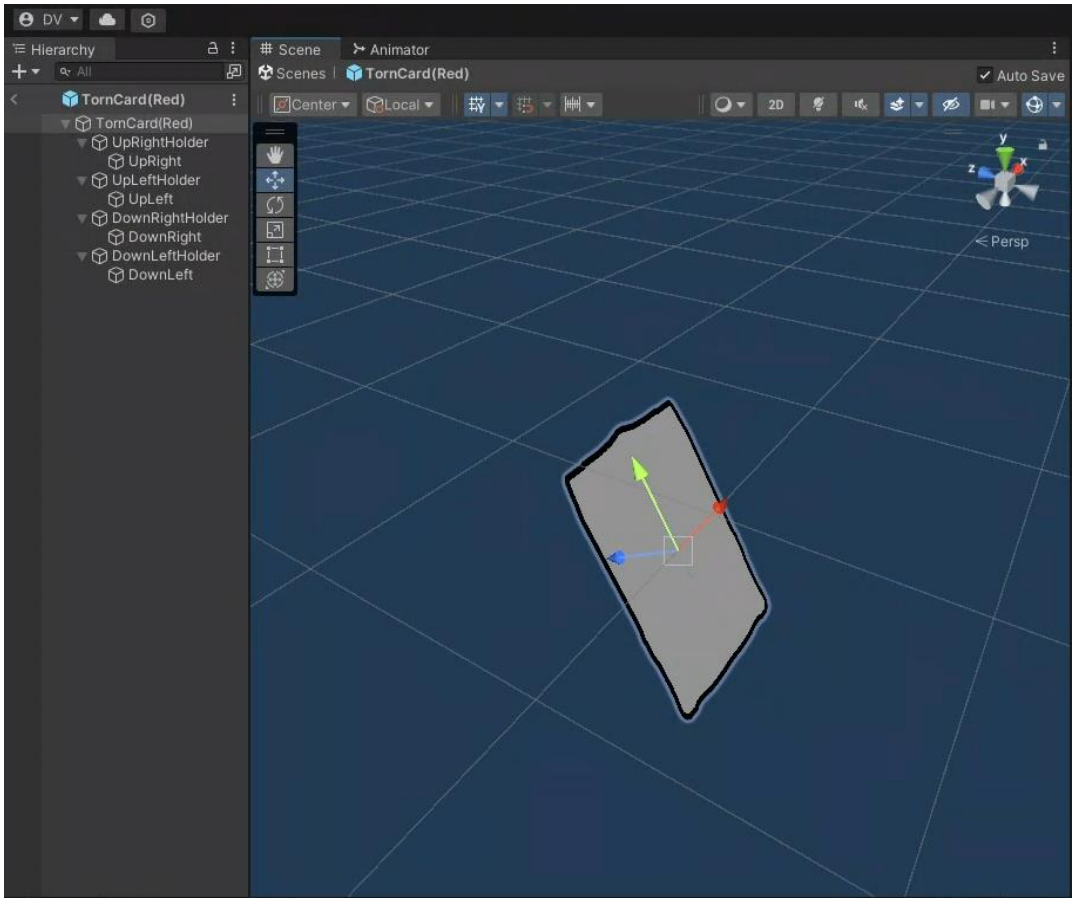


Рисунок 18. Вигляд об’єкту в редакторі.

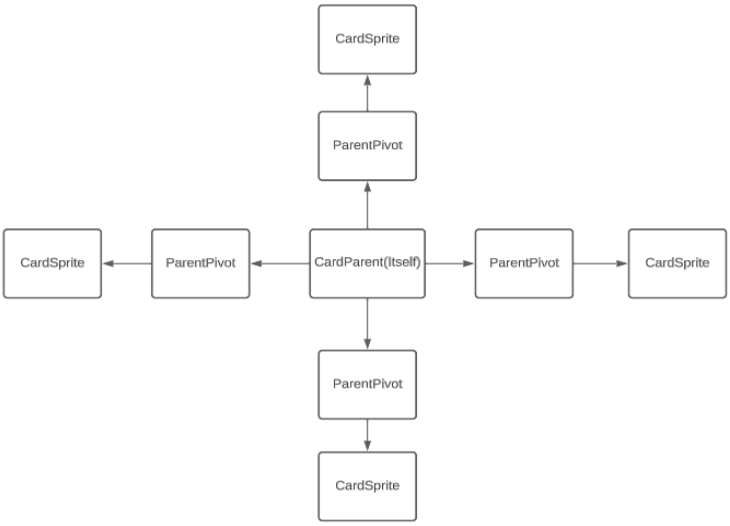


Рисунок 19. Ієрархія об’єкту.

CardParent – це пустий ігровий обєкт який виступає в якості скріплення для всіх інших.

CardSprite – це одна з 4 часточок картки яка і буде відлітати від центру, імітуючи вибух.

ParentPivot – ось тут цікаво, через особливості створення шматків карток, центр кожного шматочка був в одному й тому ж місці з центром всього об’єкту, але для того щоб імітувати розліт картки від вибуху нам потрібно отримати напрямок в якому ми будемо застосовувати сили, і кожен зі шматочків повинен відлетіти в свою сторону. Це було реалізовано так:

Для кожного шматочка ми створюємо батьківський об’єкт з центром координат в центрі шматочка і при створенні всього обєкту ми застосовуємо силу до кожного шматочка в залежності від положення цього центра у відношенні до центру всього об’єкта шляхом віднімання 3-х вимірних векторів.Ось цією формулою:

*RB.AddForce((transform.parent.position - Parent.position) * 10,ForceMode.VelocityChange);*

2.3 Моделювання концепції та архітектури інформаційної підсистеми

Вище я описав основні вимоги та способи реалізіції до деяких основних, імплементованих на даний момент в гру, систем. Настав час поговорити про архітектуру, тип архітектури я назвав “Кластер”. Ця архітектура буде мінятися ще багато разів в процессі розробки гри, бо як виявилось моделювати архітектуру гри, не протестувавши перед цим основні та базові механіки, не визначившись з масштабами проекту, не оцінивши можловості та навички програмістів, не взявши до уваги особливості програмного середовища та рушія за допомогою яких буде розроблюватися гра... Іншими словами як правило ніхто не проектує архітектуру гри до моменту коли почнеться повномаштабна розробка, в процесі якої природним шляхом буде виведена концепція найбільш підходящої для гри архітектури. Це стається в наслідок

недостатньої обізнанності окремих спеціалістів в суміжних сферах. Бо програмний архітектор може бути недостатньо обізнаним в тонкощах написання коду на тій платформі для якої він проектує гру, програміст який буде будувати на цьому каркасі гру може бути не обізнаним у тому як робляться цікаві ігрові механіки та як правильно їх пов’язувати, а геймдизайнер гадки не має як та навіщо будується програмна архітектура. І цей кругообіг некомпетентності приводить до того що, або:

- Гра розроблюється багато років, і в процесі розробки всі ці спеціалісти засобом постійної комунікації та сварок, таки через кров та піт видають хорошу гру.
- Некомпетентність та неспроможність досягти компромісу приводять до того що кожен починає робити те що сам вважвє правильним і в кінці кінців, виходить Cyberpu... кхм... погана гра.

2.3.1 Моделювання концепції архітектури

Суть архітектури “Кластер” **(Рис. 20 - Архітектура типу “Кластер” версії 0.3.)** складається в тому що за кожний елемент в грі відповідає певний обробник (Handler), всі ці обробники наслідуються від класу Ігрового менеджеру (GameManager) Який в свою чергу стежить за тим аби всі обробники мали звязок між собою, і саме в менеджері відбувається запуск та контроль ігрових послідовностей. Класс GameStorage несе в собі посилання на всі елементи упрвління та взаємодії які знаходяться на самій сцені. GameStarter відповідає за передачу екземпляра GameStorage у GameManager, таким чином ми отримуємо замкнуту систему, так зване ядро керування, від якого може відходити будь яка кількість обробників.

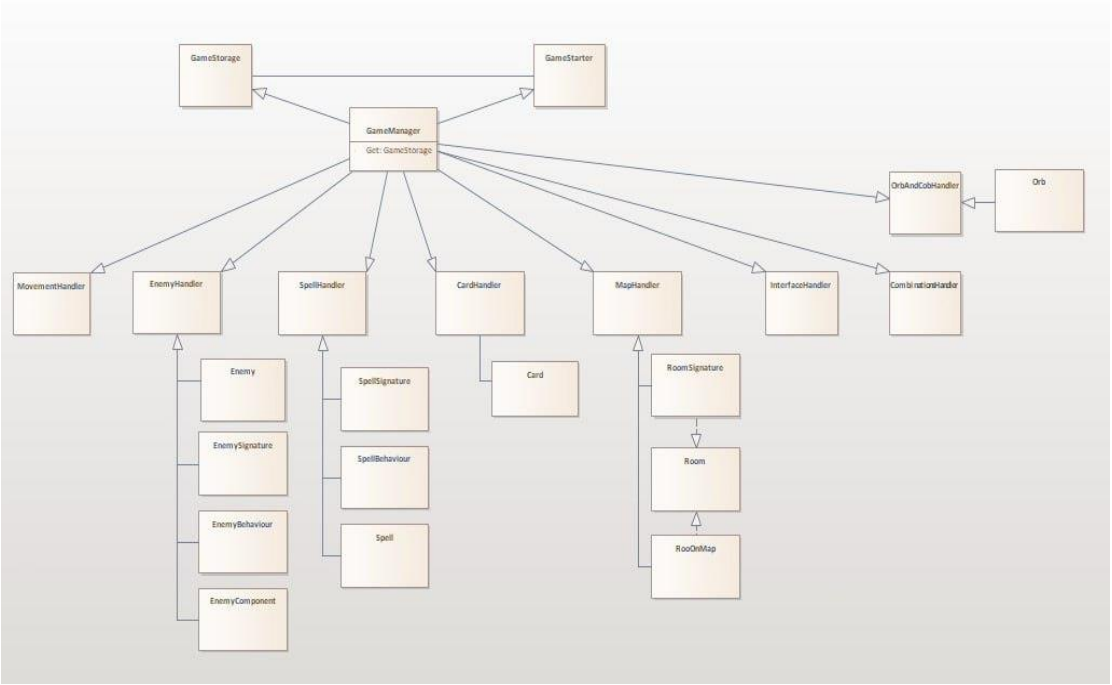


Рисунок 20. Архітектура типу “Кластер” версії 0.3.

Нижче я більш детально розберу сценарії використання кожного з обробників та їх основний функціонал:

- *InterfaceHandler:*

Відповідає за всі елементи інтервейсу та їх вплив на гру. Цей обробник ще поки не допрацьований але вже несе в собі функціонал виклику основного меню, вихід з гри та вхід у гру та функціональну інтерактивну карту **(Рис. 21 - Основний інтерфейс на момент версії 0.06.a)**. В майбутньому цей обробник буде зрощено з Map Handler.

Рисунок 21. Основний інтерфейс на момент версії 0.06.a.

- *Map Page:*

На цій вкладинці як можна побачити вже знаходиться інтерактивна карта за допомогою якої можна переміщатися по рівню. В майбутньому тут також буде глобальна карта яка буде показувати прогресс усього проходження а також складність локації і вірогідність зустрічання тих чи інших типів ворогів.

- *Inventory Page:*

На цій вкладинці буде знаходитись інформація про всі предмети які має гравець, тут можна буде подивитися повну інформацію про предмети та їх ефекти, екіпірувати їх чи викинути.

- *Gear Page:*

На цій вкладинці можна буде подивитись які предмети прямо зараз екіпіровані на персонажа, які вони дають бонуси усі разом, та окремо. Також тут можна буде подивитися поточний рівень персонажа, які бонуси були активовані та які бонуси будуть на наступному рівні персонажа.

- *Other Page:*

На цій сторінці буде знаходись загальна інформація така як альманах ворогів, локацій та предметів, деякі налаштування, можна буде подивитися поточну прогресію по глобалному дереву здібностей та ін.

- *OrbAndCombHandler:*

Відповідає за все що пов’язано з елементальними сферми та їх переміщенням у відповідні слоти. Цей обробник несе в собі інформацію про слоти в які встають сфери та про саме переміщення сфер.

- *Enemy Handler ma Spell Handler:*

Ці два класи дуже схожі між собою оскільки мають найбільшу варіативність при створенні сутностей за допомогою компонентних підкласів Spell та Enemy. Ці два класи (Spell та Enemy) є фабричними. Фабричний клас — це породжувальний патерн проектування, який визначає загальний інтерфейс для створення об’єктів у суперкласі, дозволяючи підкласам змінювати тип створюваних об’єктів. **[4]**

Патерн Фабричний метод пропонує відмовитись від безпосереднього створення об’єктів за допомогою оператора new, замінивши його викликом особливого фабричного методу. Об’єкти все одно будуть створюватися за допомогою new, але робити це буде фабричний метод. На перший погляд це може здатись безглуздим — ми просто перемістили виклик конструктора з одного кінця програми в інший. Проте тепер ви зможете перевизначити фабричний метод у підкласі, щоб змінити тип створюваного об’єкту. **[4]**

Щоб ця система запрацювала, всі об’єкти, що повертаються, повинні мати спільний інтерфейс. Підкласи зможуть виготовляти об’єкти різних класів, що відповідають одному і тому самому інтерфейсу.

Наприклад:

Класи Вантажівка і Судно реалізують інтерфейс Транспорт з методом доставити. Кожен з цих класів реалізує метод по-своєму: вантажівки перевозять вантажі сушею, а судна — морем. Фабричний метод класу ДорожноїЛогістики поверне об’єкт-вантажівку, а класу МорськоїЛогістики — об’єкт-судно. **[4]**

Клієнт фабричного методу не відчує різниці між цими об’єктами, адже він трактуватиме їх як якийсь абстрактний Транспорт. Для нього буде важливим, щоб об’єкт мав метод доставити, а не те, як конкретно він працює.**[4]**

Обробник Spell несе в собі логіку та поведінку заклятть які в свою чергу через класс Spell Behaviour можуть бути призначені будь якому закляттю в залежності від його типу дії. Таким чином ми отримуємо справжній конструктор **(Рис. 22 - Діаграма модульної системи заклятть)**.

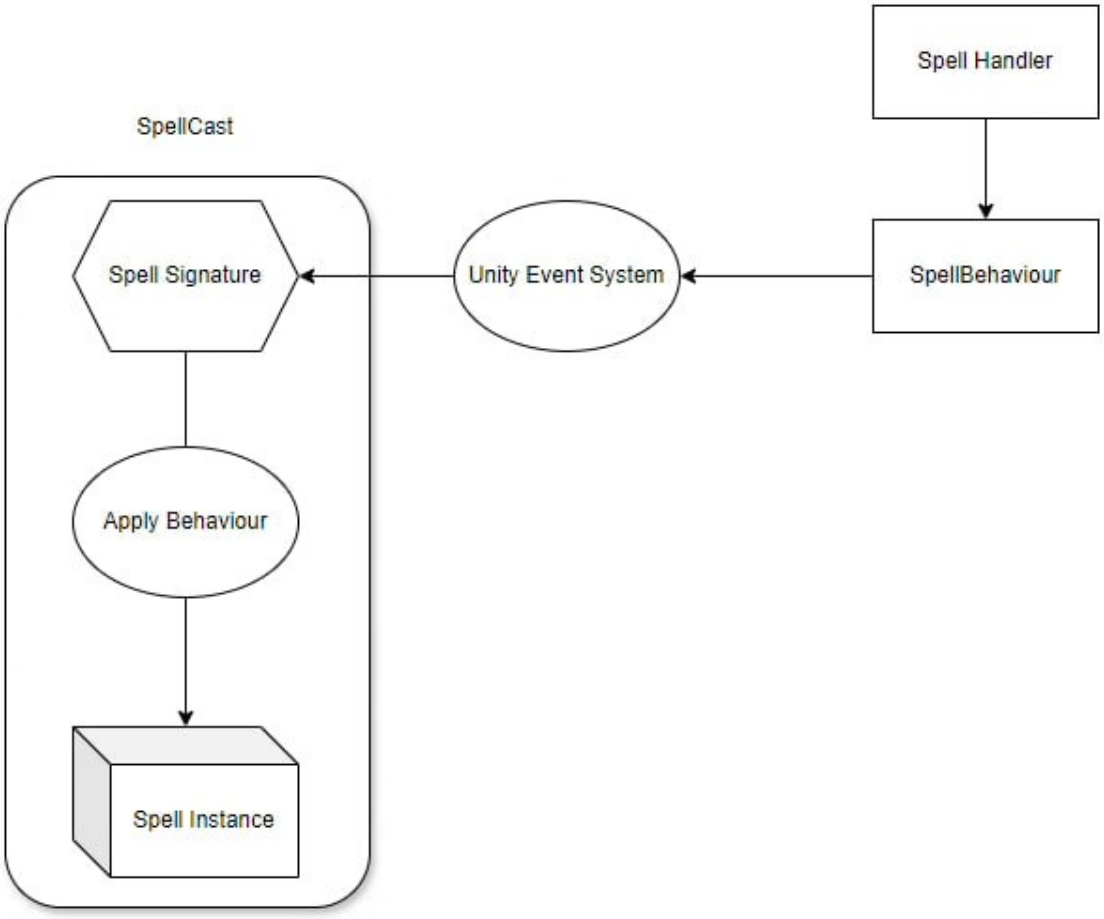


Рисунок 22. Діаграма модульної системи заклятть.

Задопомогою Unity Event System ми можемо передавати делеговані в SpellBehaviour методи до сигнатури закляття яка в свою чергу несе в собі абсолютно всю логіку функціонування закляття. А одразу після появи екземпляру закляття на ігровому полі вся ця інформація передається в екземпляр, за рахунок чого закляття і починає себе поводити так як очікується.

На відміну від Spell Handler обробник Enemy несе в собі лише функції ініціалізації, призначаючи кожному ворогові порядковий номер у масиві з якого буде діставатися посилання на самого ворога при отриманні пошкодженнь та ін. **(Рис. 23 - Діаграма модульної системи ворогів)**.

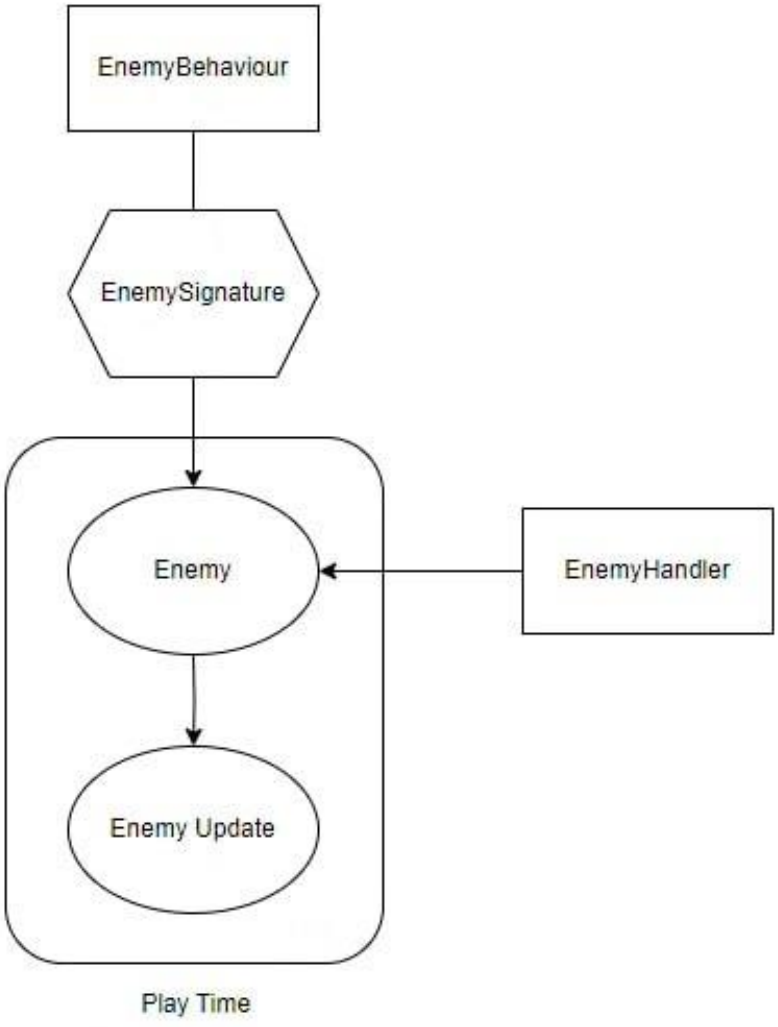


Рисунок 23. Діаграма модульної системи ворогів.

На відміну від обробника заклятть обробник ворогів не несе в собі

модульні функції, це тому що всі здібності ворога закладені в інтерактивні анімації, тому класу з модулями достатньо просто зберігати в собі активатори певних видів анімацій, цього буде цілком достатньо для того аби мати змогу налаштовувати поведінку ворогів так як нам потрібно без зайвих витрат часу.

- *Card Handler*

Цей обробник несе в собі всю функціональну інформацію пов’язану із картами та є батьківським класом для класу Card, який вже несе логіку самої карти та її специфіку і елемент.

- *Map Handler*

На даний момент цей обробник є самим комплексним у грі оскілки займається ініціалізацією всього рівня починаючи від індикаторів кімнат на карті, закінчуючи розташуванням воргів та предметів у самій кімнаті, в майбутньому цей обробник буде роздроблено на два класи Level Handler та MapHandler, та спрощено. Принцип дії основного набору функцій що формує систему рівнетворення та завантаження “Театр” вже було описано вище. Також варто сказати що цей обробник тісно пов’язаний з класом сиротою Room, в майбутньому цей клас буде зрощено з класом RoomOnMap який є дочірнім до обробника MapHandler.

- *Combination Handler*

Обробник комбінацій несе в собі всі індекси можливих комбінацій а також функції необхідні для роспізнавання індексів комбінацій, і конвертації цих індексів у посилання на самі закляття та їх характеристики.

2.3.2 Концепт-документ

- *Вступ (Сюжет):*

Ти оговтуєшся у підземеллі за гратами старого середньовічного підвалу. Ти зовсім ні в чому не винний, то все ті дурні карти що тобі підсунула та стара відьма, за які тебе звинуватили в чаклунстві, щоб її... Хм... Дивно... В тебе не забрали колоду. Ти вже сам дивитися не можеш на ті карти, взяв той малий шкіряний портфельчик в якомоу вони лежать та й киданув їх з усієї сили прямо у коридор за грати. Що це..? Колода рухається, і не аби куда, а прямо до тебе. Так само було й коли ти зустрів відьму. Вона казала що карти вибрали тебе. І ось ти сидиш один на холодній кам'яній бруківці, а в руках тремтить колода карт... Що ж, ти не в тому положенні щоб просто сидіти та жалкуватися на життя, і ти вирішуєш подивитися на що ці карти здатні...

- *Жанр та аудиторія:*

Це карткова гра в реальному часі (не покрокова), цей жанр можна назвати новаторським. Стиль напів піксельне 3D стилізоване під ранне "псевдо-3D" (DOOM, Wolfenstein).

Гра орієнтована на широку аудиторію, але не є казуальною, потребує певного рівня концентрації уваги та аналітичної реакції. Більший приціл йде саме на мобільну аудиторію, але гра буде росповсюджуватись і на ПК також.

- *Основні особливості:*
  - Новаторський жанр в якому вам доведеться комбінувати елементи карт в особливому порядку для того щоб проводити ті чи інші дії (атаки, захист, лікування, тощо.)
  - Дуже скромні вимоги до систем не дивлячись на те що гра буде яскравою та динамічною.
  - Rogue-Like механіки що дють змогу проходити гру багато разів отримуючи різний досвід від проходження.
  - Механіка комбінації елементів що буде потребувати швидкого та раціонального аналізу сутуації на полі битви стане цікавим та не занадто складним випробуванням для усіх вікових груп.

- *Опис гри:*

Самим першим завданням гравця буде визволення з-за грат, для цього необхідно буде скомбінувати карти таким чином щоб в тебе вийшла дія атаки яка повинна знищити сталеві прути.

- *Основна механіка ч1:*

Комбінація елементів карт є основною механікою в грі, комбінації складаються за рахунок вибору карт різного елементу у певному порядку.

Фіро: Агресивний елемент руйнування. Вибір першим елементом комбінації саме Фіро і зробить дію – атакою. Гравець має бути обережним з цим елементом, бо типи атак Фіро бувають найрізномнітніші і не досвідчений гравець може здійснити атаку яка нашкодить йому самому або ж зовсім не нашкодить ворогу.

Віта: Елемент життя та сили. Вибір першим елементом комбінації Віта зробить дію типу “Buff” тобто посилення гравця. Проте слід пам’ятати що різні види посилень діють певний час або ж взагалі спрацьовують моментально всього один раз. Тому наприклад немає сенсу накладати на себе посилення ще раз до того як пройде дія попереднього.

Арма: Елемент стійкості та захисту. Вибір першим елементом комбінації Арма зробить дію типу – захист. Захистити себе можна не тільки поглинувши шкоду щитом, але ще й просто не дозволяючи супротивнику себе атакувати, скувавши його, чи взагалі перейти у контр-випад одразу після блоку. Іншими словами Арма безпрограшний варіант для людей які люблять довго подумати чи відчути себе в безпеці, якщо звісно кількість залишившихся карток дозволяє.

Ітак, ми знаємо про три основні елементи і комбінції, але все ще не зрозуміло як саме їх створювати, ті комбінації.

Дуже просто: В будь-який момент часу в руці у гравця може бути всього 5 елементальних карт (В грі всі 5 карт будуть просто перед обличчям гравця), всього в колоді по 8 карток кожного елементу. На початку бою гравець дістає з колоди 5 випадкових карт. Кожен раз коли гравець використовує карту для

участі у комбінації - із колоди одразу ж дістається наступна карта випадкового елементу та кладеться на місце використаної. Самі комбінаїї можуть складатися мінімум з 1, максимум з 3 елементів і кожен раз коли гравець використовує карту - 1 з 3 слотів комбінації заповнюється елементом що відповідає використаній карті, слоти заповнюються по черзі в порядку вибраних гравцем карт. Коли гравець використовує першу карту на волю вивільнюється велика кілкість нестабільної енергії що не здатна існувати у елементальному вигляді довго, тому як тільки заповнюється перший слот комбінації гравець має встигнути заповнити ті слоти що залишилися до того як елементальна енергія перетвориться на дію. Під час бою карти які ви використаєте дезинтегруються, проте кожний певний проміжок часу вони будуть відновлюватися і знову з'являться в колоді. По завершенню бою всі використані в процессі карти не віновляться одразу, таймер відновлення буде продовжувати затримувати карти від портапляння в колоду, тому гравцю потрібно буде прогнозувати свої дії та не розкидуватись закляттями без нагальної необхідності.

- *Вороги та бій:*

Бій як правило буде проходити у форматі 1 на 1 в маленьких статичних локаціях, вся увага гравця буде сконцентрована на своїх діях та на діях ворога. Вороги представляють собою анімовані 2D спрайти що будуть з певним інтервалом атакувати гравця, або використовувати свої здібності. Також більшість ворогів будуть мати певного роду особливості, наче несприйнятливість до певних типів атак чи напраклад можливість пробивати блоки. Гравцю доведеться вивчати гру аби максимально ефективно використовувати свій арсенал карт.

- *Rogue-Like механіки:*

Рог-лайк також відомий пошановувачам цього жанру як "Рогалик" - це гра в якій є певний елемент випадковості при проходженні, як правило видів випадковостей в грі достатня кілкість для того щоб майже кожне наступне проходження гри було не схожим на попередне.

В Cardiarm елемент випадковості організований таким чином:

- Різноманіття ворогів – ти ніколи не заєш з яким саме ворогом тобі доведеться зустрітися у наступній локації, навіть якщо пере-проходиш гру, а різноманіття ворогів не дасть тобі легко дізнаватися можливий порядок шляхом віднімання з “пулу”, навіть якщо ти вже знаєш усі можливі види ворогів.
- Елементальне поглинання – при перемозі над ворогом, карти гравця поглинають його сутність, та конвертують її в свою силу, звісно ж сутність у кожного різна, тому і сила яку набудуть карти може бути надзвичайно різноманітна. Бонус отримує або комбінація або тип завдаваємої шкоди, а не якась окрема карта. Бонуси в свою чергу наймовірно варіативні.

- *Основна механіка ч2:*

Цього не було сказано раніше, але кольори є дуже важливою складовою гри, оскільки саме кольори визначають тип елементалної шкоди, посилення або захисту який використовуватиме гравець. Це важливо оскільки в процесі гри гравець буде отримувати певні посилення та бонуси стосовно певних комбінацій та типів шкоди, і якщо гравець наприклад не знає як саме складається та виглядвє комбінація Аppo, то і ефективно користувтися посиленням чи бонусом він не зможе.

Приклад бонуса для (комбінації) Аppo:

[При застосуванні комбінації Аppo гравець отримує бонус на 3 наступні дії що при атаці дає шанс у 50% нанести зверху ще додаткові 30% від вже нанесеної шкоди у вигляді типу Аppo]

Я вважаю що слід швидко пробігтися по елементальним типам шкоди 2-го рівня які створюються в процесі комбінації 2 основних елементів:

Енфіро = Фіро + Фіро

Енвіта = Віта + Віта

Ексарма = Арма + Арма

Всі комбінації що є результатом складання 2-х базових елементів мають однаковий основний принцип. Поперше цілі які сприйнятливі до базових типів шкоди будуть отримувати додатково +50% шкоди від дій що є комбінацією

2-х елементів того ж відповідного типу. Також бонуси що спрацьовують під час використання дій з базовими типами будуть спрацьовувати двічі за одну дію.

Фіта = Фіро + Віта

Віро = Віта + Фіро

Віма = Віта + Арма

Арта = Арма + Віта

Арро = Арма + Фіро

Фіма = Фіро + Арма

В залежності від порядку складання комбінації повністю змінюється механіка її використання та взаємодії з навколишнім світом, проте від порядку не змінюється тип елементалної шкоди, Треба пам'ятати що *назва комбінації і тип елементалної шкоди це не завжди одне й теж*, насправді вони співпадають лише у базових елементів т їх комбінацій самих з собою. У шкоди що є комбінацією різних базових елементів є свої окремі назви. Таким чином:

Фіта

та мають тип шкоди що називається = *Некро*;

Віро

Віма

та мають тип шкоди що називається = *Екзо*;

Арта

Арро

та мають тип шкоди що називається = *Фортас*;

Фіма

За таким принципом можливо активувати бонуси якихось певних типів шкоди при тому, що самі закляття будуть повністю відрізнятися за механікою своєї дії, що може дати стратегічну перевагу або ж навпаки зіпсувати комбінацію. Наприклад маємо бонус:

[При складанні елементальної комбінації з типом шкоди *Сонна* максимальне здоров’я супротивника зменшується на 10%]

Комбінація Фіта закрита елементом Віта (Фіро + Віта + Віта = Фівіта) накладає на ворога “Дебафф” – анти-бонус, який зменшує його максимальне здоров’я на 10%. Таким чином ми отримуємо комбінацію при якій можемо зменшити здоров’я ворога до якогось адекватного для нас рівня і вже після цього почати наносити шкоду, чудова комбінація.

Комбінація Віро закрита елементом Віта (Віта + Фіро + Віта = Віфіта) накладає на нас посилення яке дозволяє нам при кожній атаці додатково наносити ворогові шкоду яка буде складати 5% від максималного здоров’я ворога. Таким чином ми дійсно отримуємо комбінацію... анти-комбінацію, оскільки бонусом від *Сонна* ми зменшили максимальне здоров’я супротивника. І якщо наприклад у ворога було повне здоров’я то це ще пів біди, а якщо ж у ворога було меньше половини здоров’я (тобто зменшення максимальної кількості здоров’я ніяк не вплинуло на ситуацію), то ми буквально просто зменшили кілкість наносимої нами шкоди по ворогу.

- *Основні храктеристики:*

Основні характеристики ворогів це Здоров’я, Резисти та Шкода. “Резисти” це ступені супротиву різним типам шкоди, резисти можуть варіюватися від ворога до ворога, або їх може не бути взагалі.

Гравець має всі тіж самі основні характеристики, на які може впливати за допомогою підсилень та бонусів, за тим виключенням того що шкода яку наносить гравець поділяється на всі існуючі комбінації, і для кожного окремого типу, шкода вираховується індивідуально.

- *Основна механіка ч3:*

Якщо включати 1-но елементні та 2-х елементні комбінації - всього в грі 39 комбінацій, і кожна з них має свою унікальну анімацію та принцип дії, під словами “принцип дії” розуміється: як, коли та куди вони летять. Вже зібрані комбінації які почали діяти називаються закляттями, саме їх ми збираємо коли формуємо комбінації, та саме вони відповідальні за накладання ефектів та нанесення шкоди, якщо влучать звісно.

Вороги цілком можуть не стояти на місці, а рухатись, закриватись щитом чи намагатися ухилитися. Іноді завданням гравця буде необхідність правильно оцінити час та місце коли краще всього зібрати кобінацію аби вона не полетіла “у молоко”. Або ж наприклад скувати ворога аби він припинив рухатись, чи наприклад вибити в нього з рук щит. Якщо ворог тримає важкий башенний щит, то можна не тільки підірвати його комбінацією 3-х Фіро, а просто направити **летючі** закляття йому у голову, яка так зручно виглядає з-поза щита.

Слово **летючі** не просто так виділено, деякі закляття розповсюджуються по землі а деякі наприклад взагалі не мають напрямку, через що далеко не всі закляття можна запульнути саме туди куди хочеться гравцю, ці, й нетільки ці, тонкощі гравцю доведеться вивчити вже підчас самої гри.

РОЗДІЛ 3 ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ КОМПОНЕНТІВ СИСТЕМИ

3.1 Інформаційне забезпечення

- *База даних:*

Збереження даних відбувається шляхом запису їх на локальному пристрої користувача, і тільки так. Я планую і далі займатися розробкою своєї гри але на даному етапі не планую її монетизацію. Для впровдження масштабної робочої бази даних необхідно щоб система була архітектурно підпорядкована певним правилам, а для того щоб підпорядкувати її певним правилам необхідно спочатку її закінчити, бо який сенс робити БД під користувачів в не дороблений та не протестованій грі. Але тим не менш це не є головною причиною чому я вирішив не прив’язувати гру до БД, як я вже сказав я планую гру дороблювати, а це означає що я повинен буду орендувати сервери за гроші аби зберігати дані користувачв які захочуть зберегти їх на серверах, в мене просто немає на це грошей, не бачу сенсу розробляти систему якою не буду користуватись. Та і не має великої необхідності в цьому, гра в жанрі Rogue-Like, а не на проходження, в грі є лише тимчасова прогрессія на того персонажа яким ти граєш зараз, після загибелі персонажа твій прогрес все одно, за законом жанру, буде втрачено.

- *Структура інформаційних масивів:*

У своїй грі я використовую масиви для зберігання статичних екземплярів різних ігрових об'єктів таких як, карти, закляття, вороги, предмети які можна руйнувати, предмети декору та ін. В свою чергу кожний об'єкт має свої певні характеристики, записані в компоненті що відповідає його сутності. І типами даних масивів є саме компоненти, оскільки нам необхідно знати характеристики ворога та якому саме ворогу вони належать для того аби призначити їх ворогові. Те ж саме стосується і всіх інших об'єктів та сутностей. В табл. 9 наведено усі наявні масиви що зберігають у собі сигнатури та характеристики більшості об'єктів у грі. Крім того в грі також є

статичні масиви що несуть в собі статичну інформацію таку як назви комбінацій, початкова шкода заклятть, множники типів шкоди та ін.

Також в таблиці можна буде побачити таку колонку як безпекове забезпечення, це спосіб в який було захищено цілісність системних масивів, та захищено їх від перезаписування та редагування, робиться це таким чином:

- Задопомогою зашифрованого JSON файлу методом Advanced Encryption Standard (AES), під час завантаження гри цей файл розшифровується та співставляються його дані з даними поточних значень константних об’єктів, якщо десь дані не збігаються то вони перезаписуються у відповідності з зашифрованим файлом.
- Unity не дозволить редагувати приватні поля ззовні свого редактору, таким чином приватним полям які можна змінювати лише в процесі гри нічого не загрожує. Зробити так з усіма полями і об’єктами в грі не вийде оскільки доволі велика кількість скриптів та функцій потребує якщо не постійний, то дуже частий доступ до деяких об’єктів, через що їх не можна зробити приватними.

Таблиця 8. Константні сигнатурні масиви.

Найменування	Ідентифікатор	Тип	Об'єм масиву та макс. Об'єм	Безпекове забезпеченн я
cards	c	Card	[3], безмежний	JSON
enemies	e	Enemy	[2], безмежний	JSON
spells	s	Spell	[39], безмежний	JSON
spellSignatures	sS	spellSigna ture	[3], безмежний	JSON
combs	-	Comb	[3], [4]	Privating

Таблиця 8. (продовження)

slots	-	Slot	[5], [5]	Privating
rooms	r	Room	[5], безмежний	JSON
TypesDamage	-	Float (#.##)	[39], безмежний	Privating
TypesGlobal	-	Float (#.##)	[18], безмежний	Privating
CobinationsIndex	-	Int (+)	[39], безмежний	Privating
Combinations	-	String		Privating

- *Система прогресії та збереженнь:*

В грі організована система прогресії, за перемогу над ворогами передбачена нагорода у вигляді поінтів досвіду, за кожну перемогу над ворогом, в залежності від того наскільки він вважається складним ці поінти видаються а точніше заповнюють шкалу прогресії. При досягненні певного максимального на даний момент рівня цієї шкали передбачені бонусі в вигляді підсилень та підвищень характеристик тому персонажу за якого ви зараз граєте. В якості інформаційного забезпечення системи прогресії виступає запис закодованої інформації про гравця в файл формату JSON. Серіалізовані дані приймають такий вигляд **(Рис. 24 – Вигляд серіалізованих даних)**. Загальна схема серіалізації **(Рис. 25 – Схема сериализації даних)** представляє собою схему в якій є класс типу Singletone що несе в собі інформацію про всі можливі характеристики всіх об’єктів. Знаходиться цей класс під інтерфейсом IPersistence, який в свою чергу дозволяє зберігати в спеціально виділених масивах всі екземпляри обєктів які ми можемо зберегти, для проведення в подальшому з ними операцію серіалізації.

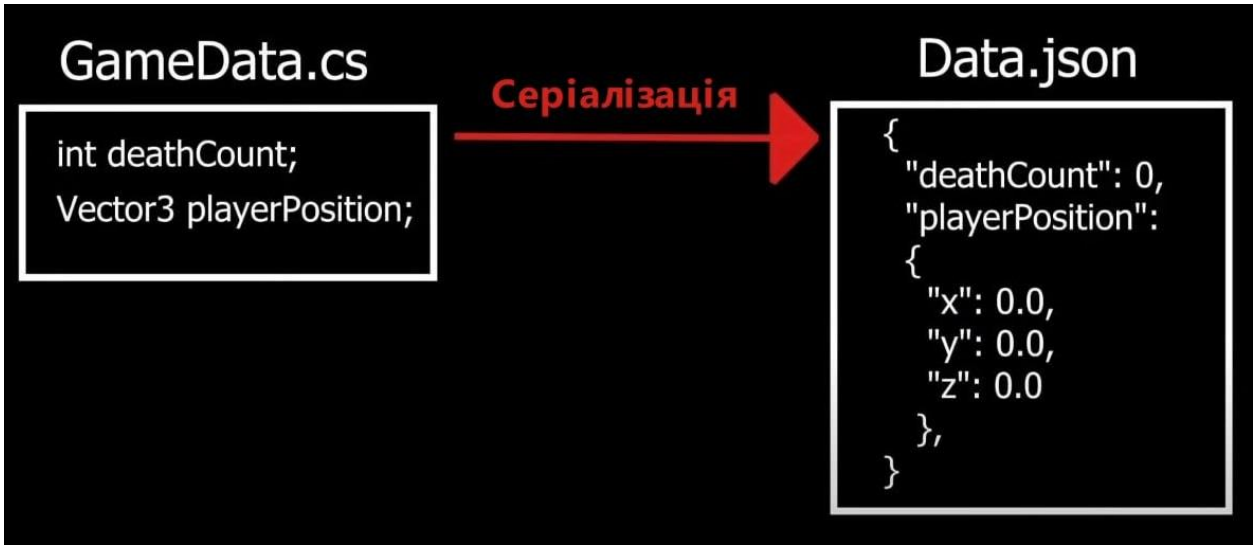


Рисунок 24. Вигляд серіалізованих даних.

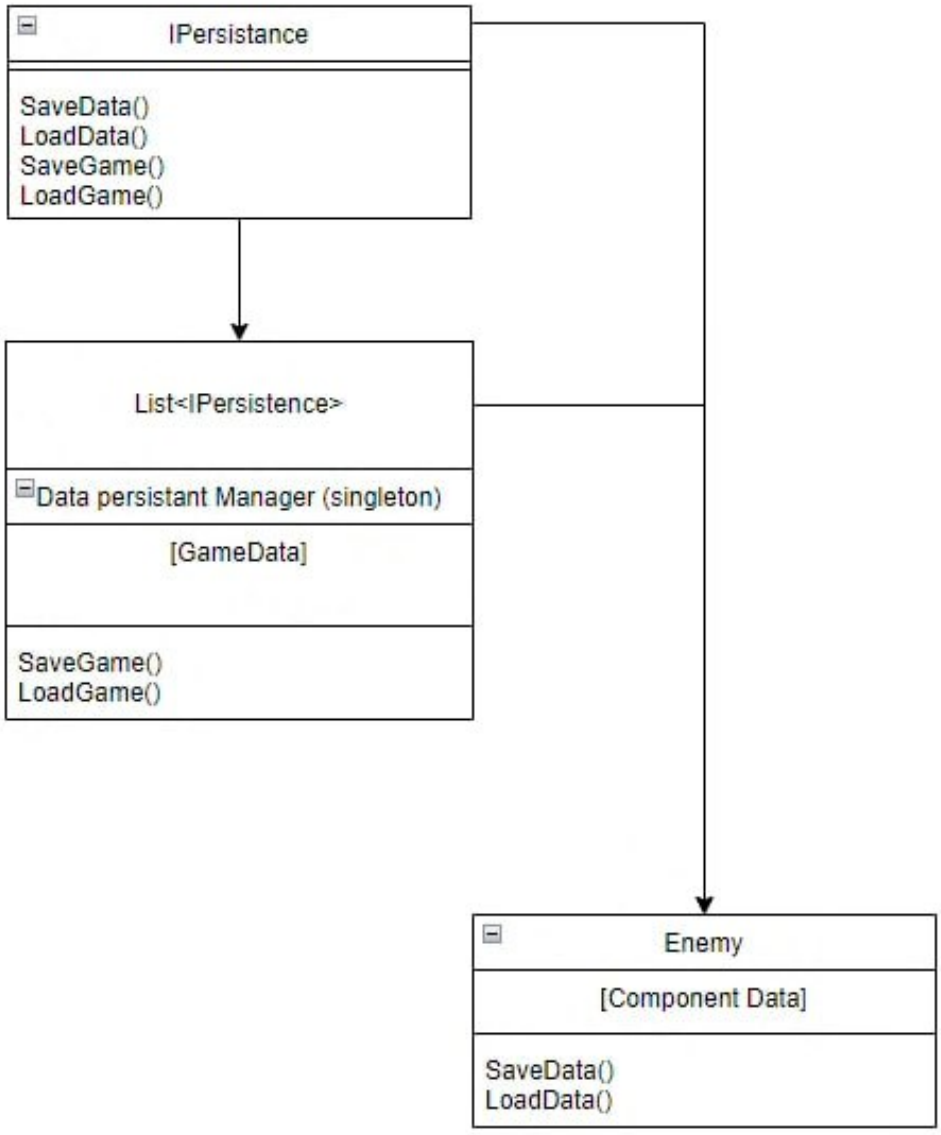


Рисунок 25. Схема серіалізації даних.

- *Штучний інтелект:*

На данний момент в грі реалізовано самий базовий інтелект ворога який прописаний вручну, в майбутньому ворог зможе приймати рішення на основі тих данних, що він зможе отримати, але кореляція буде не жорсткою. Це означає що дії ворога не будуть змінюватися одразу при перетені якогось певного значення а в залежності від значень буде змінюватися шанс на певні дії та їх послідовність.

Структурно система застосування логіки поведінки супротивника виглядає так **(Рис. 26 - Модель організації застосування поведінки ворога)**. В EnemyBehaviour знаходиться вся логіка вибору поведінки ворога. EnemySignature це ScriptableObject, тобто клас який може мати необмежену кількість варіацій, цей клас відповідає за збереження у собі інформаціїї про те якою логікою має керуватися ворог, робить він це через делегати. Коли EnemyHandler створює ворога він бере усю записану в сигнатуру логіку та перекладає її у ворога та його компоненти, якщо вони у ворога є.

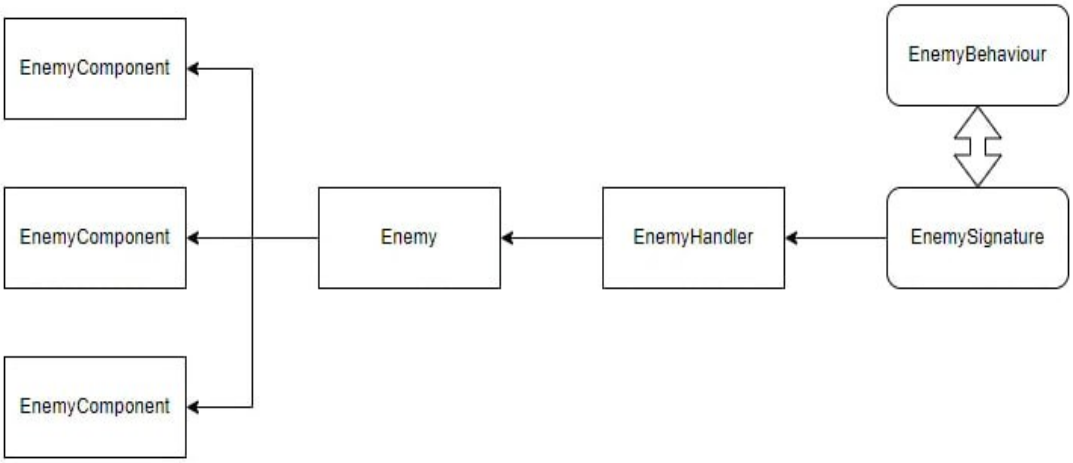


Рисунок 26. Модель організації застосування поведінки ворога.

- *Система КОП:*

На **(Рис. 27, 28 та 29 – Компоненти класу ворога, карти та загальна модель компонентної системи)** схематично зображено реалізацію системи компонентів, якими я доповнив базовий список компонентів Unity. На даний момент в грі є 4 сталі компонентні класи:

- EnemyHandler(EnemyBehaviour,EnemySignature,EnemyComponen)
- MapHandler (RoomOnMap, RoomSignature, Room)
- CardHandler (Orb)
- SpellHandler (SpellBehaviour, SpellSignature, Spell)

Насправді користувацьких компонентник класів значно більше ніж 4, проте я зазначив лише ті класи які вже мають сталу конструкцію, оскільки всі інші класи знаходяться на етапі допрацювання і продовжують допрацьовуатися навіть в момент коли ви це читаете, крім того абсолютна більшість з них виконує абсолютно тривіальні задачі, такі як утримання посиланнь на статичні об’єкти чи просто ініціалізацію константних змінних.

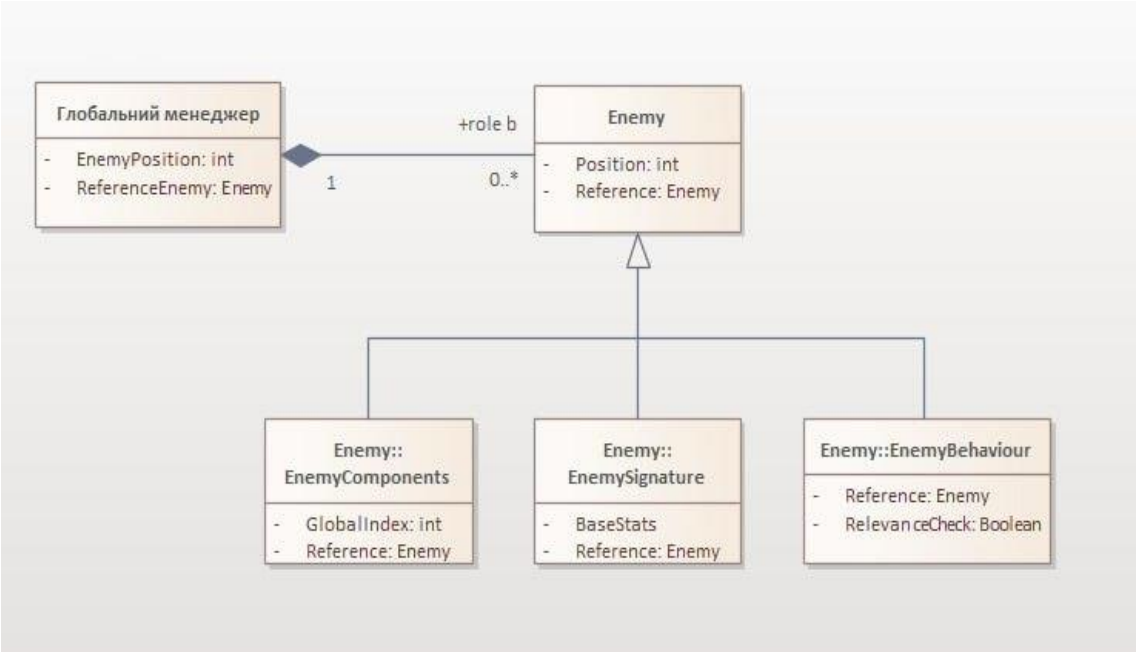


Рисунок 27. Компоненти класу ворога.

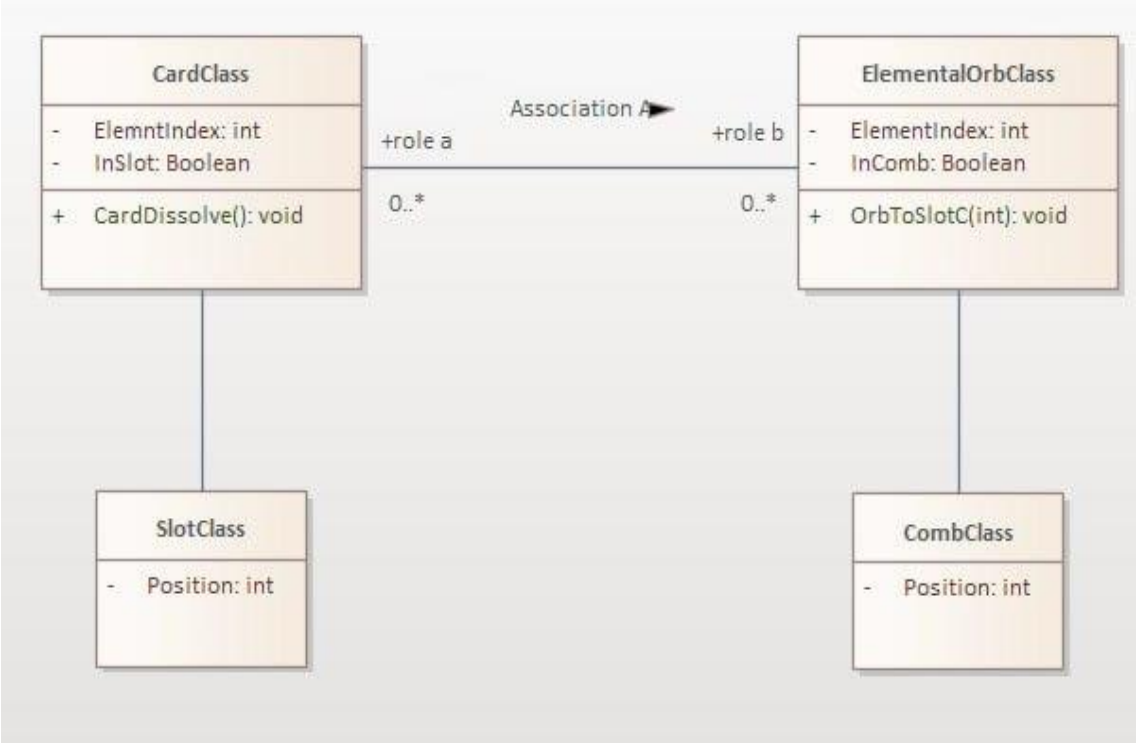


Рисунок 28. Компоненти класу карти.

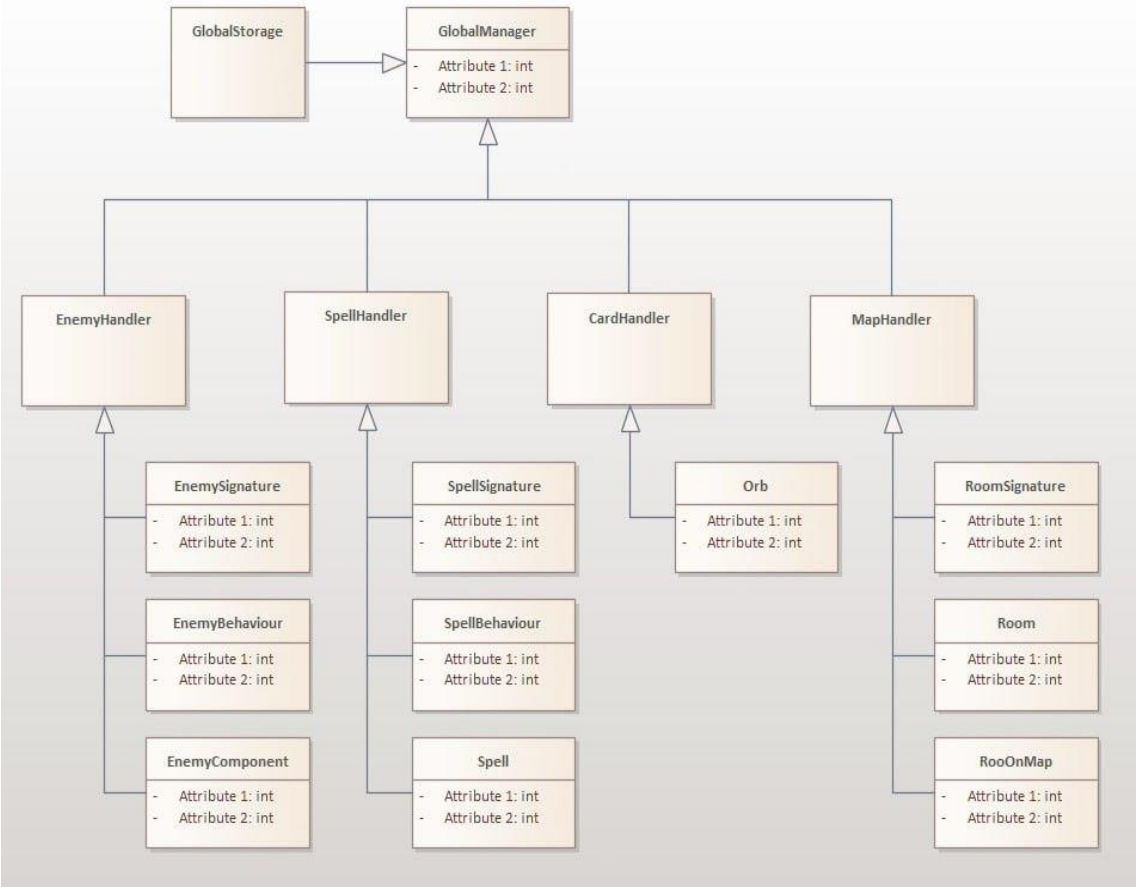


Рисунок 29. Загальна модель компонентної системи.

3.2 Технічне забезпечення

- *Системні вимоги гри:*

Дуже важливим питанням для будь-якої гри є питання мінімальних системних вимог. Особливо актуальним це питання є у зв’язку з тим що ігри це найбільші по об’єму споживання ресурсів комп’ютера програми, після серед моделювання та рендерингу звісно. Але нам не обов’язково працювати за комп’ютером під час оброблення створених нами сцен в середі моделювання, на відміну від ігор, в яких ми “зобов’язані” приймати участь у ігровому процесі, а отже ЕОМ повинна працювати стабільно в процесі роботи гри, і не допускати підторможень. Дізнатися системні вимоги гри насправді не так вже й легко, бо як мінімум для того щоб їх дізнатися необхідно завершити розробку гри та портувати її на певні девайси, після чого на цих девайсах запустити її. Саме так, просто подивитись на циферки в аналізаторі завантаженості не вийде, кожна конфігурація системи унікальна, навіть не описати наскільки часто трапляється ситуація коли на слабкіших ЕОМ гра видає більше кадрів на секунду ніж на більш потужних. Все діло в доцільності використання грою ядер процесора та відеопам’яті, підлаштувати гру так щоб вона сама визначла що в ЕОМ краще та навантажувала саме цей компонент системи – неможливо. Ну як неможливо, занадто безглуздо й складно, скоріше так, бо окрім самої гри нам для цього доведеться ще й писати програму яка буде вираховувати коефіцієнти потужності та злагодженості роботи системи після чого обирати один з багатьох варіантів заздалегідь заготовлених варіантів реалізації гри, що роздує складність, розмір та вартість проекту до неадекватних цифр. Тому й досі самим ліквідним та надійним методом знаходження рекомендованих системних вимог гри залишаєть просто-напросто запуск на якнайбільшій кількості девайсів.

Оскільки я не володію повноцінною студією з інвесторами та не маю друзів, я зміг протестувати гру всього на 4 телефонах:

- Samsung Galaxy A24 6/128GB
- Xiaomi Redmi Note 8 Pro 6/128Gb

- Infinix Zero 30 4G 8/256GB
- Xiaomi Redmi Note 13 8/256GB

Усі вище перелічені моделі телефонів протягом 1 години без підторможувань дозволяли грі йти в 60 кадрів на секунду, ні на одному з девайсів не було помічено аномально швидкого розряду батареї чи підвищення температури. Можна сказати що гра, в її нинішньому стані, більш ніж успішно пройшла первинне тестування. Найслабкішим з усіх вище перечислених девайсів є Xiaomi Redmi Note 8 Pro 6/128Gb, з найслабшим процессором MediaTek Helio G90T. 8 ядер, 2 ядра Cortex-A76 на 2050 МГц та 6 ядер Cortex-A55 на 2000 МГц **[7]**. Телефон впорався не гірше ніж білш сучасні аналоги, тому його можна записивута якщо не в рекомендовані девайси,то в мінімально припустимі точно.

- *Графіка та аудіо:*

В грі графіка буде 3D але все оточення буде спрайтовим, тобто не 3-х вимірні моделі а білборди, виставлені завжди “обличчям” до гравця таким чином буде витримано певний стиль а сама гра буде мало важити та не сильно напружувати графічні карти, це особливо актуально коли ми згадуємо про те шо гра орієнтована на мобільні пристрої.

Аудіо забезпечення в грі на даний момент немає, це буде однією з фінальних задач в розробці гри оскільки більшість людей не слухають звуки в мобільній грі, грають без навушників та з вимкненим звуком тому акцент на аудіо буде зроблений мінімальний.

- *Вибір середовища розробки:*

Створення будь якої гри це наймовірно комплексна та трудомістка задача. Вона потребує знання в одразу багатьох професійних сферах. Але перед тим як почати розробляти гру потрібно визначитись з середовищем розробки. В моєму випадку це був ігровий рушій Unity.

В виборі рушія я в першу чергу керувався його інформційною доступністю. Я зараз кажу саме про те яка кількість навчальних та методичних матеріалів що до роботи з рушієм є у відкритому доступі. Як це можна дізнатися? Можна подивитися кількість випадаючих запитів в Google, але

очевидно, що будь яка платформа має хоч якісь навчальні матеріали для початківців, а як же дізнатися який саме рушій має найбільшу кількість методичних матеріалів що дійсно допоможуть мені як розробнику закінчити свій проект. Тобто таких матеріалів які показують не окремі ситуації та функції, а ті які опишуть повний шлях від ідеї до випуску гри. Щож, на мою думку найкращим відображенням саме цього параметру стане кількість випущених ігор на цьому рушії. Візьмемо платформу Steam як саму на данний момент популярну серед людей що грають в ігри та займаються росповсюдженням своїх. А також один з найбільших онлайн сервісів для розміщення, продажу та завантаження інді-ігор (інді-ігри від Англ. – independent – незалежні проекти які як правило розробляються маленькою командою чи взагалі однією людиною, і як правило не мають прямих інвесторів чи видавництв) в інтернеті. Ось що каже статистика.

В першу 10-ку ігрових рушіїв по кількості завантажених ігор на пратформи Steam та itch.io:

- *Unity (27148)*
- *Unreal (6869)*
- *GameMaker: Studio (2806)*
- *RPGMaker (1938)*
- *Ren'Py (1235)*
- *XNA (572)*
- *Adobe AIR (398)*
- *Godot (384)*
- *Cocos2d (326)*
- *MonoGame (280)*

Перевага Unity над конкурентами очевидна навіть неозброєним оком, а оскільки ми вирішили що критерієм буде виступати кількість завершених та випущених ігор то усі рушії що мають показник меньше 2000 ігор відпадають. Таким чином ми отримуємо три варіанти на вибір Unity, Unreal та

GameMaker:Studio. Хоч навіть серед 3 найбільш популярних Unity виходить далеко вперед. Вважаю доцільним пояснити чому в моєму конкретному випадку Unity буде краще не тільки за основним критерієм. Оскільки всі три рушія мають безкоштовні версії вони будуть розглядатися окремо від своєї цінової моделі.

Unreal Engine – надпотужний рушій здатний створювати фотореалістичні інтерактивні світи та надає найширший, з на даний момент представлених на ринку, інструментаріїв для оптимізації та створення ААА проектів (ААА проект, він же “тріпл-ей”, triple-A - умовна підмножина відеоігор, створюваних й розповсюджуваних середніми й великими видавництвами, що зазвичай мають більше коштів на розробку й рекламу. Чітких мірил належності певної гри до підмножини ААА немає, тож зазвичай ідеться про клас передових високобюджетних ігор, розробка яких пов'язана зі значним економічним ризиком і потребою високих показників збуту задля забезпечення прибутковості.).

Як можна зрозуміти з опису, Unreal є надлишковим рушієм, розробка малого проекту на ньому буде схожа на “стрілянину з гармати по горобцях”, крім того не дивлячись на свою популярність і кількість випущених ігор Unreal це переважно рушій високобюджетних ігрових компаній і локальних спеціалістів. І знання про те як ним правильно користуватись переважно знаходяться на сторінках товстезних англомовних мануалів. Також з надмірної потужності випливає надмірне споживання ресурсів комп’ютера, мій ПК не є слабким але навіть так, при експлуатації періодично відчувалась нестача потужностей.

GameMaker: Studio – популярний ігровий рушій для багатьох двовимірних ігор, проте має низку обмежень таких як наприклад випуск ігор в безкоштовній версії тільки на платформу Opera GX.games та дуже слабкий вбудований 3D рушій.

Оскільки гра має бути виконана в стилізованому 3D а випуск планувався на мобільних девайсах то рушій GameMaker відпадає сам собою.

Unity – багатоплатформовий інструмент для розроблення відеоігор і застосунків, і рушій, на якому вони працюють. Створені за допомогою Unity програми працюють на настільних комп'ютерних системах, мобільних пристроях та гральних консолях у дво- та тривимірній графіці, та на пристроях віртуальної чи доповненої реальності.

Нашим фаворитом став Unity, насправді я вже маю доволі обширний досвід роботи з Unity і на певному рівні володію мовою C#, але таким чином я пояснив чому Unity був би для мене нйкращим варіантом в не залежності від досвіду використання інших рушіїв.

3.3 Програмне забезпечення

Створення відеогри вимагає використання різноманітних програмних засобів для розробки, тестування, графічного та звукового дизайну, анімації, маркетингу та інших аспектів. Не всі з вище перечислених є актуальними для мене прямо зараз але є ті які необхідно зазначити.

- Інтегроване розробницьке середовище (IDE):

Unity, вище я вже розписав чому саме він, на мою думку один з найкраших ігрових рушіїв у співвідношенні якість-можливості-легкість освоєння.

Коли ви встановлюєте Unity на Windows і macOS, за замовчуванням Unity також встановлює Visual Studio або Visual Studio для Mac відповідно. Ви можете виключити цей пункт під час вибору компонентів для завантаження та встановлення. За замовчуванням зовнішній редактор скриптів (меню: Unity > Preferences > External Tools > External Script Editor) налаштовано на Visual Studio. Коли ви вмикаєте цю опцію, Unity запускає Visual Studio та використовує його як редактор за замовчуванням для всіх файлів скриптів. Unity інтегрується з Microsoft Visual Studio через пакет редактора коду для Visual Studio **(Рис. 30 - VSE у вікні менеджеру пакетів)**. Цей пакет попередньо встановлено під час встановлення Unity. Якщо під час інсталяції

Unity встановлено Visual Studio, Unity використовує Visual Studio для відкриття та редагування скриптів за замовчуванням. **[3]**

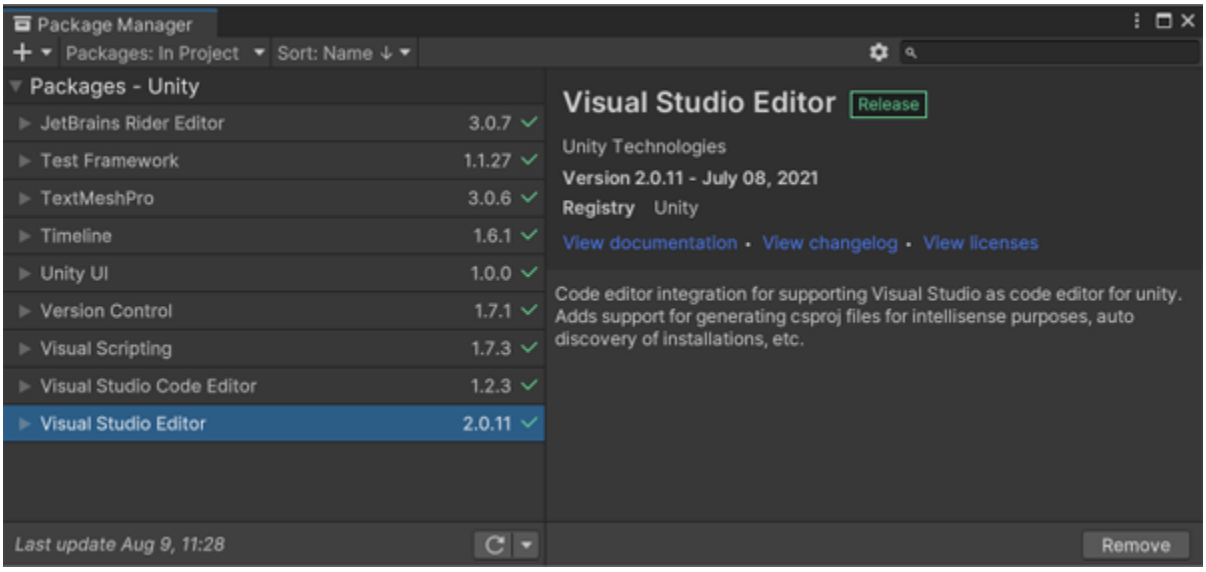


Рисунок 30. VSE у вікні менеджеру пакетів. [3]

Щоб встановити редактор скриптів за замовчуванням у ручному режимі, необхідно перейти до Edit > Preference у головному меню, щоб відкрити вікно параметрів. Відкрийте меню зовнішніх інструментів (External Tools). Клацніть спадне меню External Script Editor **(Рис. 31 – Налаштування зовнішніх інструментів)** і виберіть Microsoft Visual Studio. Вигляд цього параметра змінюється залежно від версії Microsoft Visual Studio, яку ви встановили. **[3]**

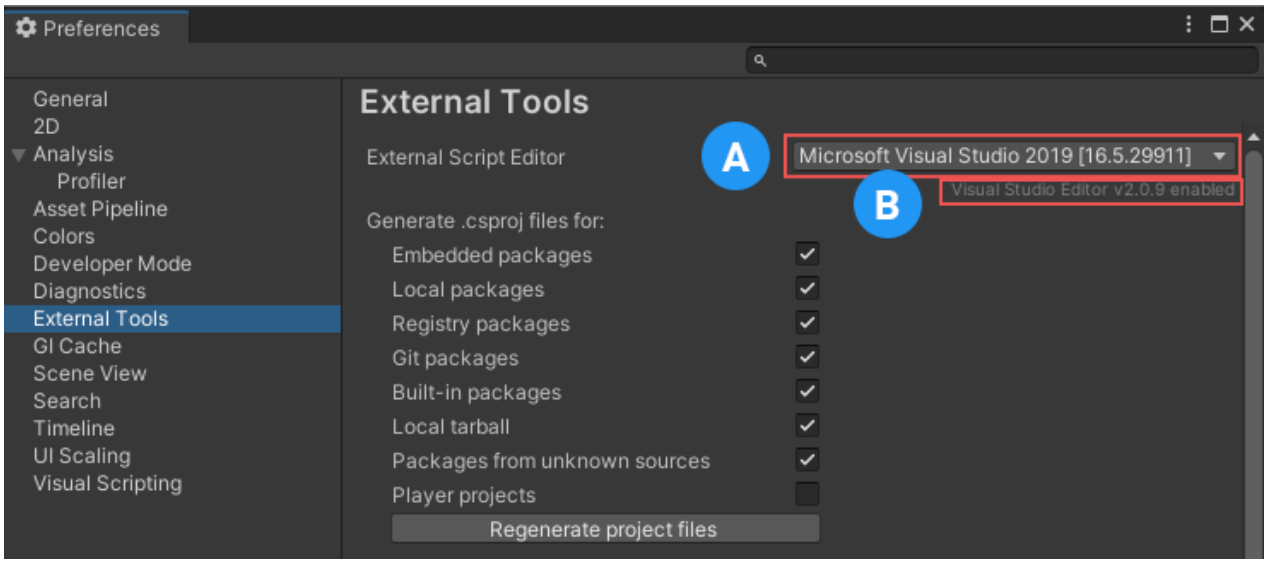


Рисунок 31. Налаштування зовнішніх інструментів. [3]

Unity використовує компілятор C# Visual Studio для компіляції скриптів. Коли ви використовуєте пакет редактора Visual Studio разом з Visual Studio, і Unity, і Visual Studio відображають деталі будь-яких помилок у ваших скриптах. Unity автоматично створює та підтримує файл Visual Studio .sln і .csproj. Ви можете контролювати, коли, куди та як Unity генерує файли .csproj для певних елементів вашого проекту в меню «External Tools» у вікні «Preferences», як на знімку екрана вище. Увімкніть або вимкніть прапорці, щоб увімкнути чи вимкнути генерацію файлів .csproj для певного параметра. Unity повторно генерує файли .sln і .csproj у вашому проекті кожного разу, коли ви вносите зміни в стан файлу, наприклад, редагуючи існуючий файл або створюючи новий. Ви також можете додавати файли до свого рішення з Visual Studio. Unity імпортує будь-які нові файли, і наступного разу, коли Unity знову створює файли проекту, вони створюються з новими файлами. **[3]**

Я під час розробки користувався більш легкою версією Visual Studio – Visual Studio Code. Він налаштовується так само як і повномасштабна версія. В додаток до самого редактору мною використовувалися деякі розширення для більш комфортної розробки, такі як:

- Unity for Visual Studio Code
- .NET Install Tool
- C# for Visual Studio Code
- Unity Code Snippets

- Мова програмування:

C# (Sharp) - це мова програмування загального призначення високого рівня, що підтримує більшість існуючих парадигм програмування. C# охоплює статичну типізацію, об’єктно-орієнтоване (на основі класів) і компонентно-орієнтоване програмування.

Очевидно що C# буде моїм вибором бо гра робиться на Unity, проте мені особисто більше імпонує C++, якби я робив свій рушій він би абсолютно точно був би на C++.

- Графічні редактори:

Krita — це потужний і багатофункціональний графічний редактор, який має ряд переваг, особливо для художників, ілюстраторів, аніматорів та розробників ігор **(Рис. 32 – Графічний редактор Krita)**.

Поперше Krita є повністю безкоштовним програмним забезпеченням без прихованих платежів чи підписок, що для нас є ледь не головною причиною вибору саме її**.** Програма має відкритий код, що дозволяє користувачам вносити власні зміни та завантажувати скрипти інших користувачів.

Krita пропонує більше 100 професійно розроблених кистей для різних стилів малювання а також має функцію створення та налаштування власних кистей і текстур. Зручний та налаштовуваний інтерфейс, який можна адаптувати під власні потреби, панелі інструментів та плагіни можна налаштовувати і переміщувати для максимального комфорту. Підтримка шарів, шарів-масок, шарів-фільтрів, а також шарів для векторної графіки а також має будовані фільтри для створення різних художніх ефектів. Підтримка покадрової анімації, таймлайну, а також функції onion skin для зручності роботи з анімацією. Велика кількість навчальних матеріалів, туторіалів та уроків, які доступні онлайн, більше того Krita має велику спільноту користувачів та розробників, які також діляться досвідом, створюють навчальні матеріали та можуть підтримати новачків на спеціалізованих форумах.

Рисунок 32. Графічний редактор Krita.

Blender — це спеціалізоване ПО для 3D моделювання, яке має численні переваги як для художників-аніматорів так і для розробників ігор та дизайнерів візуальних ефектів **(Рис. 33 – Середовище моделювання Blender)**.

Так само як і Krita Blender абсолютно безкоштовний та має відкритий код, що також дозволяє створювати власні аддони та завантажувати сторонні прямо з GitHub. Blender постійно оновлюється, додаючи нові функції та виправляючи помилки і також має велику спільноту користувачів які регулярно створюють навчальні матеріали та різного роду корисні плагіни.

Підтримка різних технік моделювання, включаючи полігональне моделювання, скульптинг, NURBS та субдівізійне моделювання. Має інструменти для малювання прямо на моделях, UV-розгортка, проекційне текстурування, покраска текстур та багато іншого. Також має потужний інструментарій для покадрової анімації, інверсної кінематики (IK), формоанімування задопомогою shape keys та багато іншого, наприклад інструменти для симуляції фізики рідин, диму, вогню, тканин та твердих тіл.

Потужний інструментарій для композитингу з підтримкою вузлів або ж ще кажуть нодів, що дозволяє створювати складні візуальні ефекти та налаштовувати їх прямо у процесі створення або запису анімації, та має будований нелінійний редактор відео, який дозволяє редагувати відео та створювати анімації безпосередньо в Blender.

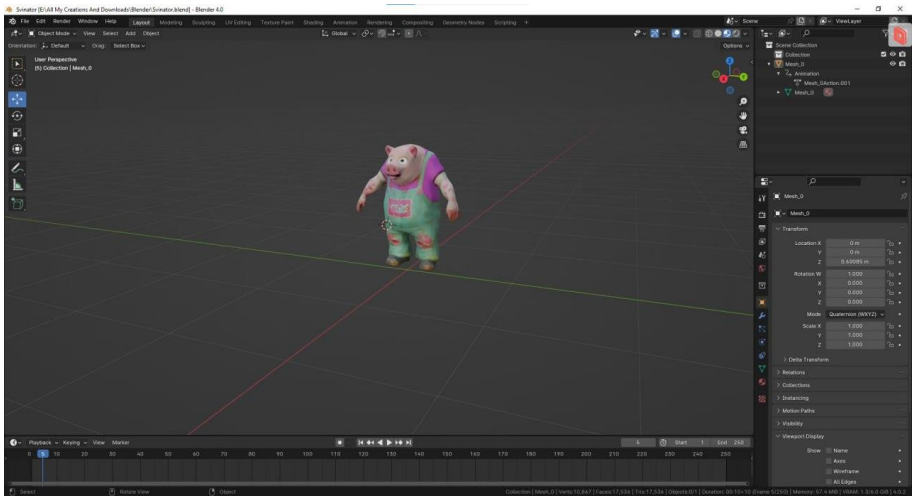


Рисунок 33. Середовище моделювання Blender

3.4 Результати реалізації інформаційної підсистеми

Тут коротко в програмному вигляді опишу послідовність відпрацювання скриптів гри.

Спочатку нам необхідно вибрати карту, елемент якої ми хочемо додати до комбінації **(Рис. 34 – Функція що відповідає за визначення комбінації та переміщення сфери)**, після чого ми приймаємо рішення продовжувати складати комбінацію або...

```
private IEnumerator EffectRealizeC()
{
    if (HowManyTakenSlots < 3)
    {
        DefiningCombinationName(Effect.GetComponent<Orb>().OrbElement * MultiplyerDefiner());//GetComponent

        _Effect = Instantiate(Effect, transform.position, transform.rotation);

        Orbs.Add(_Effect.GetComponent<Orb>());//GetComponent

        for (int j = 0; j < 10; j++)
        {
            _Effect.transform.position =
            Vector3.Slerp(Combs[HowManyTakenSlots].transform.position, _Effect.transform.position, 0.75f);
            yield return new WaitForFixedUpdate();
        }

        _Effect.transform.position = Combs[HowManyTakenSlots].transform.position;

        HowManyTakenSlots++;
    }
    else
    {
        DestroyOrbsClearCombs();
    }
}
```

Рисунок 34. Функція що відповідає за визначення комбінації та переміщення сфери.

...активуємо комбінацію завдяки натисканню на прозору панель що покриває всю зону заліку, в цей момент відбувається програвання анімації скриптом **(Рис. 35 – Функція початку залікової анімації.)**, того як ми буквально хапаємо елементи рукою та формуємо з них закляття, екран же в свою чергу загоряється кольоровою рамкою яка інформує про те що закляття готове до використання.

```
public void StartAnimationHandGrabbin()
{
    if (HowManyTakenSlots > 0)
    {
        ReadyPanel.SetActive(false);
        animationManager.GrabbinHandSwitch();
    }
}
```

Рисунок 35. Функція початку залікової анімації.

В момент коли з’явиться кольорова рамка в зоні заліку активується так звана панель променю RayCastPanel **(Рис. 36 – Вікно налаштуваннь зони заліку)** при натисканні на котру у нас буде відбуватися проектування променя в напрямку тієї точки в яку було натиснуто, у випадку якщо закляття мало небойовий характер, або просто є ненаправленим, то натискання на панель променя просто активує закляття, у випадку ж якщо закляття портебує для своєї повноцінної роботи напрямок руху, воно **(Рис. 37 – Функція Shot в коді)** візьме ту точку в напрямку якої гравень натиском пустив промінь як орієнтир, і полетить саме туди.

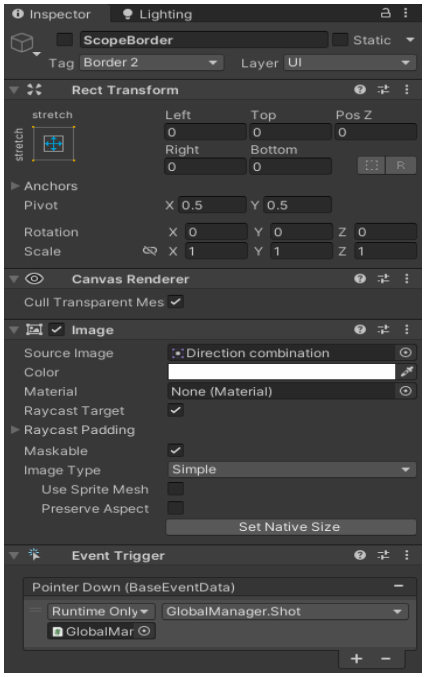


Рисунок 36. Вікно налаштуваннь зони заліку.

```
public void Shot()
{
    RayMouse = mainCamera.ScreenPointToRay(Input.mousePosition);
    Physics.Raycast(RayMouse.origin, RayMouse.direction, out hit, 5000);
    SpawnCombinationSpell(new Vector3(0,0.5f,0.5f));
    DenieCombinationLock();
    DeactivateBlocker();
}
```

Рисунок 37. Функція Shot в коді.

У випадку якщо гравець промаже снаряд просто розірветься, а у випадку якщо снаряд влучить у ворога буде запущений спеціальний скрипт **(Рис. 38 - Функція DamageRecieve в коді)** що підрахує з урхуванням усіх резистів та бафів шкоду яку має отримати ворог і відніме цю цифру від його здоров'я, а індикатор над головою супротивника відобразить скільки шкоди було нанесено та скільки ще необхідно нанести.

```
public void DamageRecieve(int Type, int Form, int FormDamage, int TypeDamage)
{
    health -= resists[Form] * DamageForms[FormDamage] * resists[Type] * DamageTypes[TypeDamage];

    healthBar.value = health;

    if (health < 0)
    {
        StopAllCoroutines();
        animator.SetBool("Death", true);
        //Destroy(gameObject);
    }

    Debug.Log(health);
}
```

Рисунок 38. Функція DamageRecieve в коді.

Зараз більш предметно розглянемо що саме нам необхідно для того аби грати в гру, а саме Input Data. Оскільки гра націлена на мобільні платформи то головну роль будуть грати натиски та затискання пальцем (мишею).Також є підвид введення даних типу затискання, так зване перетягування. Основна відмінність перетягування від затискання в тому що затискання може бути статичним, тобто може не потребувати переміщення обєкту для того щоб мати

сенс, перетягування ж це затискання що потребує від вас руху курсором/пальцем для використання функціоналу утиліти в повному обсязі.

В грі сам натиск виконує декілька задач, поперше активує закріплення комбінації, для того щоб пізніше таким самим натиском можна було запустити закляття. Затискання виконує одну функцію- перетягування карт у зону заліку, в якій вони будуть розпадатися та вивільнювати елементальну сферу що в свою чергу буде заповнювати слоти комбінації. Розглянемо реалізацію збору Input Data для них:

- Натиск реалізується вбудованим методом OnMouseDown який спрацювиє при натисканні на коллайдер об’єкту. Я не користуюся цією функцією так як її в моєму випадку можна замінити методом OnMouseDrag.
- Затискання це більш складна та комплексна операція яка складається з двох вбудованих методів: OnMouseDrag **(Рис. 39 - Функція OnMouseDrag в коді)** та OnMouseUp **(Рис. 40 - Функція OnMouseUp в коді)**.

```
private void OnMouseDrag()
{
    //EventSystem.current.IsPointerOverGameObject()==false
    if (rememberedCombinationNumber==0)
    {
        Vector3 mousePoint = Input.mousePosition;
        mousePoint.z = 1.2f;
        transform.position = Camera.main.ScreenToWorldPoint(mousePoint);
    }
}
```

Рисунок 39. Функція OnMouseDrag в коді.

Спочатку при натисканні на карту спрацьовує метод OnMouseDrag який кожен кадр, поки ми не відпускаємо клавішу буде виконувати написаний всередині метода код. Код буде переміщувати карту туди де буде знаходитись курсор чи палець.

```
private void OnMouseUp()
{
    if (transform.position.y >= 1.3)
    {
        SlotCardIn.Taken = false;
        _TornCard = Instantiate(TornCard, transform.position, transform.rotation);
        _TornCard.GetComponent<OrbSequence>().Respawn(gameObject);
        transform.position = RespawnPosition;
        gameObject.SetActive(false);
    }
    else
    {
        transform.position = SlotCardIn.transform.position;
    }
}
```

Рисунок 40. Функція OnMouseUp в коді.

У випадку якщо ми відпустимо клавішу миші або ж палець, спрацює метод OnMouseUp який проводить перевірку, якщо карта знаходиться поза зоною заліку то вона повернеться до слоту, у випадку якщо карта буде в зоні заліку, то її слот стане знову вільним, сама карта буде тимчасово відімкнена та запуститься скрипт елементалної сфери Orb.

Зона заліку - це зона що знаходиться над картами та над слотами для елементальних сфер. Ця зона займає весь простір що вище за 1.3 по осі Y. Сама гра побудована так щоб не залежати від швидкості обробки даних в основному потоці, та від потреби перевіряти кожен кадр на предмет виконання умов певних дій. Гра по суті InputBased, іншими словами поки гравець не натисне на екран в основному потоці нічого не буде відбуватися, за рахунок цього гра має дуже маленьку затримку відгуку (InputLag), і в перспективі її буде доволі складно “зламати”, мається на увазі викликання збоїв у роботі гри.

Впродовж усього циклу розробки гра доволі часто змінювала свій зовнішній вигляд а іноді навіть і цілі геймплейні концепції. На **(Рис. 41 - Вигляд гри на версії v.0.02a, v.0.04a та v.0.06a)** зліва-направо відповідно зображено як змінювалася гра упродовж розробки, зміни відносно суттеві, проте найбільші зміни відбувалися саме в коді, велика кількість запланованих механік так і не була імплементована, натомість були імплементовані деякі мехініки яких не було на початку. На даному етапі розробки в гру вже можна

пограти але її ні в якому разі не можна назвати завершеною. Ця гра була моєю першою спробою створити власну гру, і хоч я вже мав певний досвід роботи з ігровим рушієм Unity, за той проміжок часу що я займався цією грою мені довелось власноруч відшукати та вивчити більше інформації ніж за весь період мого знайомства з Unity. В силу моєї недосвідченості я натикався на самі різноманітні технічні та геймплейні проблеми вирішення яких могло займати тижні. Тим не менш мені вдалося довести гру до грабельного стану, а найголовніше, я отримав наймовірну кількість безцінного досвіду який в майбутньому допоможе мені в доопрацюванні Cardiarm та під час розробки інших ігор.

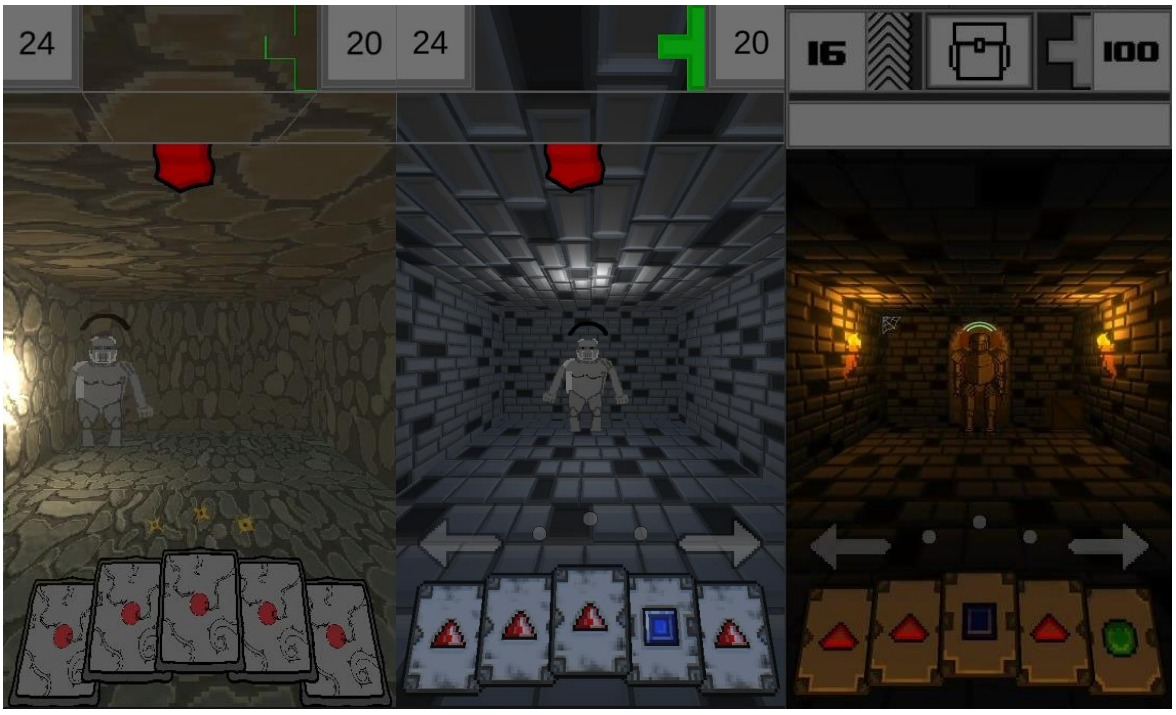


Рисунок 41. Вигляд гри на версії v.0.02a, v.0.04a та v.0.06a.

ВИСНОВКИ

У ході виконання бакалаврської роботи було проведено проектування та виконано початковий етап розробки карткової комп'ютерної гри в жанрі Rogue-like. Проект включав в себе вивчення основних принципів цього жанру, аналіз існуючих ігор, а також створення прототипу власної гри на основі отриманих під час самонавчання знань.

Початковий етап роботи передбачав створення концепції гри, визначення основних механік і функціоналу. Застосування ідей Rogue-like, таких як випадково генеровані рівні, перманентна смерть та непередбачуваність подій, в майбутньому дозволить створити унікальний геймплей, який відзначиться великою різноманітністю і викликами для гравця.

В другій частині, процес розробки включав в себе аналіз сучасних Rogue-like ігор, вивчення їхніх механік та імплементацію власних варіантів з урахуванням особливостей карткового формату, також я пояснив свій вибір тих чи інших ігрових механік, пояснив чому саме їх я вибрав для імплементації, і який ментальний шлях проходив для досягнення розуміння того що грі дійсно потрібно.

Процес програмування включав в себе використання сучасних інструментів і технологій, щоб забезпечити оптимальну продуктивність та зручний інтерфейс гри. Застосування мов програмування та фреймворків, специфічних для галузі геймдеву, дозволив ефективно реалізувати визначені на данному етапі завдання та досягти високого рівня геймдизайну. Крім того, були використані сучасні підходи до програмування, такі як об’єктно-орієнтоване програмування та паттерни проектування, для забезпечення гнучкості та розширюваності коду. Я імплементував основні механіки гри, такі як генерація рівнів, управління гравцем, бойова система та система прогресу.

У процесі тестування гри вдалося виявити та усунути численні помилки та недоліки, що поліпшило якість та стабільність гри, та дало неоціненний досвід який буде максимально ефективно застосовано в подальшому.

В цілому, розробка карткової комп'ютерної гри у жанрі Rogue-like була цікавим та важливим досвідом, який дозволив здобути практичні навички в галузі геймдеву, розширити розуміння основних принципів геймдизайну та програмування, а також випробувати себе у творчому процесі створення власної гри.

Результатом роботи став прототип гри з набором складних геймплейних механік, яка випробовує навички гравця та надає велику кількість варіантів стратегій.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) https://spacelab.ua/articles/prijomi-pid-chas-proyektuvannya-arhitekturi-igor/

2) https://gamemaker.io/en/blog/stages-of-game-development

3) https://docs.unity.com/

4) https://refactoring.guru/uk/design-patterns/factory-method

5) https://en.wikipedia.org/wiki/Collectible_card_game

6)https://docs.unity3d.com/2023.1/Documentation/Manual/BestPracticeLightingPipelines.html

7) https://www.mediatek.com/

ДОДАТКИ

Додаток А

Екземпляр коду головного ініціалізуючого скрипта (GameManager)

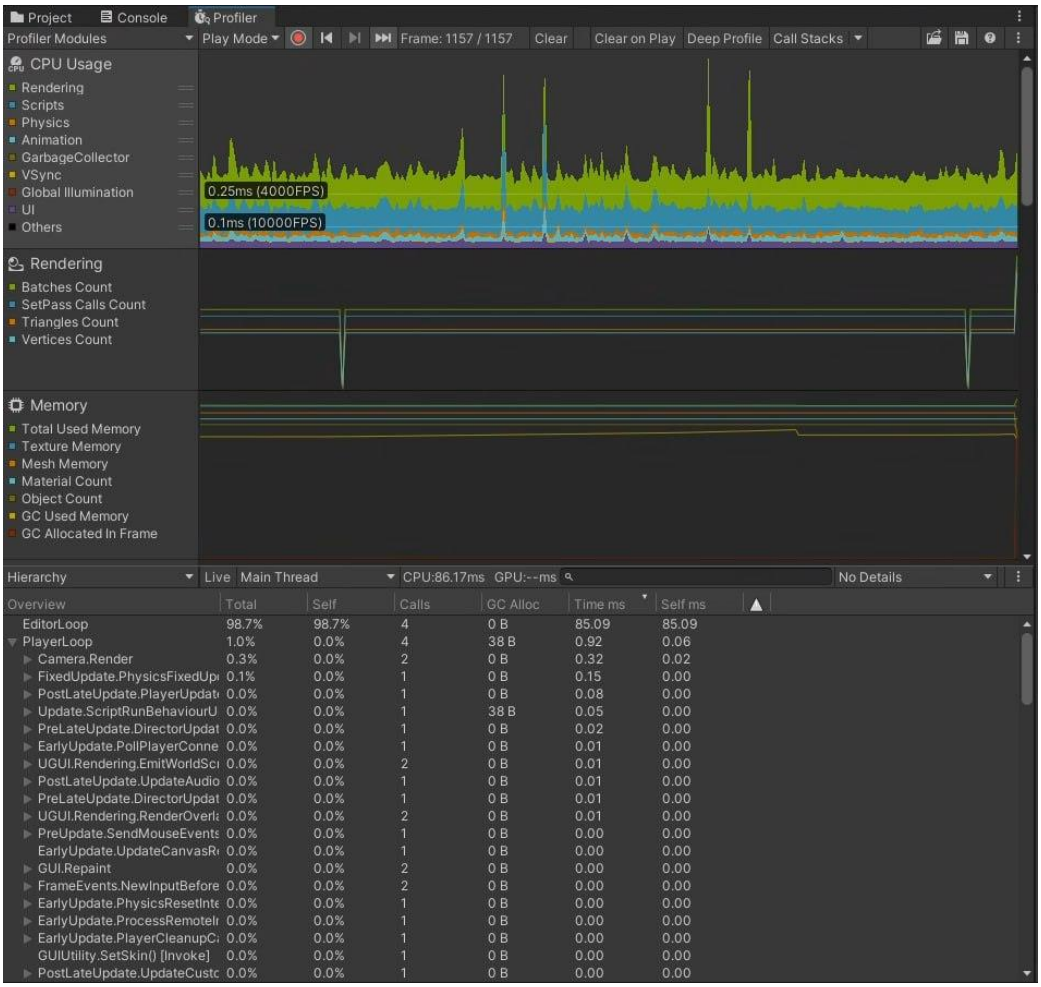
```
using System.Collections.Generic;
using UnityEngine;

public class GameManager : MonoBehaviour
{

    public static CombinationsHandler combinationsHandler;
    public static AnimationsHandler animationsHandler;
    public static InterfaceHandler interfaceHandler;
    public static RayCastHandler rayCastHandler;
    public static EnemyHandler enemyHandler;
    public static LevelHandler levelHandler;
    public static SpellHandler spellHandler;
    public static CardHandler cardHandler;
    public static CombHandler combHandler;
    public static SlotHandler slotHandler;
    public static OrbHandler orbHandler;

    public static GameStorage gameStorage;

    private void Update()
    {
        if(Input.GetKey(KeyCode.Q))
        Time.timeScale = 0.01f;
    }

    public void BasicGameStart()
    {
        StorageWorkflow();
        HandlersWorkflow();
        InterfaceWorkflow();
        CardWorkflow();
        RayCastWorkflow();
        OrbWorkflow();
    }

    private void StorageWorkflow()
    {
        gameStorage = gameObject.GetComponent<GameStorage>();
    }

    private void HandlersWorkflow()
    {
        combinationsHandler = gameObject.AddComponent<CombinationsHandler>().GetComponent<CombinationsHandler>();
        animationsHandler = gameObject.AddComponent<AnimationsHandler>().GetComponent<AnimationsHandler>();
        interfaceHandler = gameObject.AddComponent<InterfaceHandler>().GetComponent<InterfaceHandler>();
        rayCastHandler = gameObject.AddComponent<RayCastHandler>().GetComponent<RayCastHandler>();
        enemyHandler = gameObject.AddComponent<EnemyHandler>().GetComponent<EnemyHandler>();
        levelHandler = gameObject.AddComponent<LevelHandler>().GetComponent<LevelHandler>();
        spellHandler = gameObject.AddComponent<SpellHandler>().GetComponent<SpellHandler>();
        cardHandler = gameObject.AddComponent<CardHandler>().GetComponent<CardHandler>();
        combHandler = gameObject.AddComponent<CombHandler>().GetComponent<CombHandler>();
        slotHandler = gameObject.AddComponent<SlotHandler>().GetComponent<SlotHandler>();
        orbHandler = gameObject.AddComponent<OrbHandler>().GetComponent<OrbHandler>();
    }

    private void InterfaceWorkflow()
    {
        interfaceHandler.InstantiateInterface();
        interfaceHandler.ActivateInterface();
    }

    private void CardWorkflow()
    {
        cardHandler.GenerateCards(10);
        cardHandler.CardShuffle();
        cardHandler.CardsCount();
        cardHandler.AlignAllCards();
    }

    private void RayCastWorkflow()
    {
        rayCastHandler.InitializeRayCast();
    }

    private void OrbWorkflow()
    {
        orbHandler.SetUpOrbs();
    }
```

Додаток Б

Фрагмент коду скрипта детекції та взаємодії (RayCastHandler)

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

public delegate void RayFunction();

public class RayCastHandler : GameManager
{
    private bool rayBusy = false;
    private RaycastHit hit;

    public void InitializeRayCast()
    {
        rayFunctionsStorage["Card"] += cardHandler.CardDrag;
    }

    public RaycastHit RayHitInfo()
    {
        return hit;
    }

    private void Update()
    {
        if (Input.GetMouseButton(0) && !rayBusy)
        {
            Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition), out hit);

            if (hit.collider != null)
            {
                rayBusy = true;
                rayFunctionsStorage[hit.transform.tag]?.Invoke();
            }
        }

        if(!Input.GetMouseButton(0))
        {
            rayBusy = false;
        }
    }

    private readonly Dictionary<string, RayFunction> rayFunctionsStorage = new()
    {
        ["Card"] = null,
        ["Room"] = null
    };
}
```

Тест навантження на систему вбуд. утилітою UnityProfiler



Додаток В

Екземпляр коду відповідального за логіку роботи карт (CardHandler)

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

public class CardHandler : GameManager
{
    private int remainingCards;
    private bool canManipulateCards = true;
    private List<Card> cardsInCache = new();
    private List<Card> cardsInHand = new();
    private List<Card> cardsInactive = new();

    public void CardsManipulateState(bool state)
    {
        canManipulateCards = state;
    }

    private void DestroyCard(Card card)
    {
        cardsInHand.Remove(card);
        cardsInactive.Add(card);
        card.transform.localPosition = new(0, -3, 0);
        card.SlotCardIn.Taken = false;
    }

    private void AlignOneCard()
    {
        for (int i = 0; i < 5; i++)
        {
            //Debug.Log("Oppa");
            if (SlotHandler.slots[i].Taken == false)
            {
                AlignmentInternals(i);
                break;
            }
        }
    }

    private void AlignmentInternals(int index)
    {
        SlotHandler.slots[index].Taken = true;
        cardsInCache[0].SlotCardIn = SlotHandler.slots[index];
        cardsInCache[0].transform.position = SlotHandler.slots[index].transform.position;
        cardsInHand.Add(cardsInCache[0]);
        cardsInCache.Remove(cardsInCache[0]);
    }

    public void CardShuffle()
    {
        for (int i = 0; i < cardsInCache.Count; i++)
        {
            cardsInCache.Insert(Random.Range(0, cardsInCache.Count - 1), cardsInCache[cardsInCache.Count - 1]);
            cardsInCache.RemoveAt(cardsInCache.Count - 1);
        }
    }

    public void CardsCount()
    {
        remainingCards = cardsInCache.Count + cardsInHand.Count;
        interfaceHandler.interfaceHolder.cardsDisplay.text = remainingCards.ToString();
    }

    public void CardDrag()
    {
        for (int i = 0; i < cardsInHand.Count; i++)
        {
            if (cardsInHand[i].transform == rayCastHandler.RayHitInfo().transform)
            {
                StartCoroutine(CardDragC(cardsInHand[i]));
            }
        }
    }

    private IEnumerator CardDragC(Card card)
    {
        while (Input.GetMouseButton(0) && canManipulateCards)
        {
            Vector3 mousePoint = Input.mousePosition;
            mousePoint.z = 1;

            card.transform.position = Camera.main.ScreenToWorldPoint(mousePoint);

            yield return new WaitForEndOfFrame();
        }

        if (card.transform.position.y < combHandler.combs[1].transform.position.y)
        {
            card.transform.position = card.SlotCardIn.transform.position;
        }
        else
        {
            orbHandler.UnleashElementalOrb(card);
            DestroyCard(card);
            CardsCount();

            if (cardsInCache.Count > 0)
            {
                AlignOneCard();
            }
        }
    }
}
```