СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки

Кафедра інформаційних технологій та програмування

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної випускної роботи

освітній ступінь бакалавр
спеціальність 121 «Інженерія програмного забезпечення»
(шифр і назва спеціальності)

на тему «Ігровий застосунок на рушії Unity»

Виконав: студент групи ІПЗ-21д _____ І. В. Щербак
(підпис) (ініціали і прізвище)

Керівник _____ В. Г. Іванов
(підпис) (ініціали і прізвище)

Завідувач кафедри _____ О.І. Захожай
(підпис) (ініціали і прізвище)

Рецензент _____В.О. Лифар_____

Київ – 2025

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки
Кафедра інформаційних технологій та програмування
Освітній ступінь бакалавр
спеціальність 121 „Інженерія програмного забезпечення”
(шифр і назва спеціальності)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Захожай О.І.
“___” _____ 2025 року

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

Щербак Іван Вячеславович
(прізвище, ім’я, по батькові)

1. Тема роботи: Ігровий застосунок на рушії Unity

керівник роботи Іванов Віталій Геннадійович
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджений наказом університету від “_27_” _травня_2025 року №67/15.15-С

2. Строк подання студентом роботи 16.06.2025р.
3. Вихідні дані до роботи: Об’єктом даної роботи є процес проєктування гри на рушії Unity

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): Вступ. Аналіз методів та засобів розробки ігрового застосунку Визначення основних вимог до функціоналу додатку Опис архітектурних рішень та процесу проєктування програмного забезпечення Опис процесу реалізації ігрового застосунку Висновки. Перелік використаних джерел.
5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслеників)

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 01.04.2025р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання кваліфікаційної випускної роботи	Строк виконання етапів	Примітка
1	Одержання завдання на виконання роботи	01.04.25	виконано
2	Укладання і погодження з керівником плану і етапів виконання роботи	06.04.25	виконано
3	Узагальнення даних літературних джерел, укладання розділу «Аналіз предметної галузі»	18.04.25	виконано
4	Аналіз шляхів виконання завдання. Вибір і погодження з керівником оптимального шляху	01.05.25	виконано
5	Укладання та тестування програмного продукту	02.06.25	виконано
6	Укладання, оформлення та погодження пояснювальної записки з керівником	12.06.25	виконано
7	Надання готової пояснювальної записки на кафедру	16.06.25	виконано
8	Укладання доповіді і презентації	16.06.25	виконано

Студент _____ І.В. Щербак
підпис (ініціали і прізвище)

Керівник роботи _____ В.Г. Іванов
підпис (ініціали і прізвище)

РЕФЕРАТ

Робота містить: 61 сторінка основного тексту, 18 сторінок додатків, 10 рисунків, 13 використаних джерел.

В бакалаврській дипломній роботі розглянуто процес створення прототипу ігрового застосунку з використанням рушія Unity. Була приділена увага архітектурі програмного забезпечення, реалізації ігрових механік та використанню сучасних підходів до розробки.

Для успішного досягнення поставленої мети був проведений аналіз успішних ігрових проєктів у жанрі roguelike та розглянуті різні архітектурні підходи. Було застосовано ігровий рушій Unity та мову програмування C#.

Результатом роботи став розширюваний прототип тривимірний ігрового застосунку с реалізованими системами управління персонажем, штучного інтелекту ворогів, бойової взаємодії, пошуку шляху.

Зміст

ВСТУП

Індустрія комп’ютерних ігор – це динамічне середовище, яке постійно розвивається та видозмінюється. Вона охоплює різні жанри, платформи та технології. Сучасні ігри стають усе складнішими та більш інтерактивними завдяки розвитку графічної техніки, штучного інтелекту та мережевих можливостей. Останнім часом ігрова індустрія збільшує свою частку на ринку і стає популярнішою.

Розробка ігор поєднує в собі багато різних галузей знань – програмування, дизайн, психологію, математику та інші. Виробництво гри – це систематизований процес, який має багато етапів, від ідеї до публікації. В роботі будуть проаналізовані ідеї з популярних проєктів та описані різні методи розробки.

При аналізі конкретних ігор важливо виділити їхні механіки, стиль управління, фізику та інше. Багато ігор використовують схожі механіки пересування, стрільби та взаємодії з об’єктами тощо. Аналіз популярних проєктів допомагає визначити ключові елементи, які роблять гру успішною, цікавою та захопливою для гравців.

Метою бакалаврської кваліфікаційної роботи є дослідження сучасних підходів до проєктування ігрових систем, архітектурних рішень і патернів, аналіз геймдизайну. У роботі буде розглянуто різні парадигми, методології та патерни, що дозволяють краще структурувати код і спрощують його підтримку. Також буде проаналізовано рішення, які використовуються в популярних ігрових проєктах.

Завданням роботи є створення на основі набутих знань прототипу ігрового застосунку з використанням сучасних підходів та інструментів. Необхідно побудувати гнучку архітектуру та, спираючись на аналіз популярних рішень, реалізувати різноманітні ігрові механіки.

1. АНАЛІТИЧНИЙ ОГЛЯД

1.1 Вибір рушія

Для спрощення створення ігор розробники використовують ігрові рушії, як основу для проєктів. Тому перш за все треба визначитись з середовищем розробки, бо від цього залежить ефективність розробки, здатність її підтримки у подальшій розробці та її кінцева якість. Середовище повинно бути гнучким та, бажано, мати великий функціонал який може бути необхідний для конкретних задач.

1.1.1 Godot Engine

Godot Engine - це багатофункціональний кросплатформний рушій для створення 2D та 3D ігор з єдиним інтерфейсом. Він надає повний набір загальних інструментів, щоб користувачі могли зосередитися на створенні ігор. Ігри можна експортувати на низку платформ, включаючи основні настільні платформи (Linux, macOS, Windows), мобільні платформи (Android, iOS), а також веб-платформи та консолі.

Основною перевагою godot є його безплатність і відкритий вихідний код з ліцензією permissive MIT license. Через це рушій став популярний серед спільноти інді розробників, які мають досить обмежений бюджет розробки , бо не має необхідності платити роялті з продажів. Розробка godot повністю незалежна та керується спільнотою, що дозволяє користувачам допомогти сформувати свій механізм відповідно до їхніх очікувань.

Загальна архітектура рушія побудована навколо концепції дерева з наслідуваних сцен. Кожен елемент сцени (вузол, нод), в будь-який момент сам може стати повноцінною сценою. Тож при розробці можна легко змінювати повністю всю архітектуру проєкту, розширювати її функціонал в будь-яку сторону та працювати із комплексними сценами на рівні простих абстракцій.

Всі ігрові ресурси, від скриптів до графічних ассетів та ігрових сцен, зберігаються в теці проєкту як звичайні файли та не є частиною складної бази даних проєкту. Ресурси, що не являють собою комплексних даних, наприклад скрипти та сцени, на відміну від моделей та текстур, зберігаються у простих текстових форматах. Ці рішення дозволяють значно спростити різним командам розробників працювати з системами керування версій.

Написання коду здійснюється на GDScript або на C#. GDScript є високорівневою інтерпретованою мовою з необов'язковою статичною типізацією. Синтаксис надихнутий Python, однак GDScript не оснований на ньому. Також є можливість розробки на C# з повною підтримкою синтаксису та функцій C# 12.0.

1.1.2 Unreal Engine

Unreal Engine – це ігровий рушій, що розроблений компанією Epic Games. Рушій підтримує розробку для різних платформ, включаючи ПК, консолі, мобільні пристрої та віртуальну реальність. З моменту свого першого випуску в 1998 році, Unreal Engine постійно еволюціонував, пропонуючи розробникам нові інструменти та можливості.

Головною перевагою Unreal Engine є вбудовані інструменти для створення високоякісної графіки, тому цей рушій часто використовується великими компаніями для AAA проєктів з високими вимогами до візуальної складової. До недоліків можна віднести необхідність для комфортної роботи з рушієм потужного апаратного забезпечення як зі сторони розробника, так і для гравців.

Логіку можна реалізовувати за допомогою Blueprints або C++. Blueprints – це візуальна скриптова система, яка дозволяє створювати логіку гри без написання коду. Хоча Blueprints спрощують процес розробки, для повного використання можливостей Unreal Engine бажано мати знання C++ та досвід у програмуванні. Усі елементи ігрового рушія представлені у вигляді об'єктів, що

мають набір характеристик і клас, який визначає доступні функції. Своєю чергою, будь-який клас є дочірнім класом object.

1.1.3 Cryengine

CryEngine – це потужний ігровий рушій, розроблений компанією Crytek, який використовується для створення високоякісних відеоігор із фотореалістичною графікою та складною фізичною симуляцією. Вперше представлений у 2002 році як технологічна основа для гри Far Cry, він згодом перетворився в автономний продукт, що постійно розвивається та адаптується до сучасних вимог ігрової індустрії.

Основною особливістю CryEngine є його графічні можливості, зокрема підтримка глобального освітлення, реалістичних тіней, складних систем частинок і передових ефектів рендерингу. Рушій активно впроваджує технологію трасування променів, що дозволяє досягати високого рівня деталізації та реалістичності у візуальному відображенні сцен. Завдяки використанню алгоритмів фізично коректного рендерингу (PBR) CryEngine забезпечує точне відтворення матеріалів і освітлення в режимі реального часу.

Ще однім важливим компонентом є вбудована система фізики, що базується на PhysX та власних технологіях Crytek. Вона підтримує розрахунок реалістичних взаємодій між об'єктами, руйнування структур та моделювання складних фізичних ефектів, що особливо важливо для шутерів, пригодницьких і ігор-симуляторів. Крім того, рушій має просунуту систему анімації, що підтримує як процедурну генерацію рухів, так і ручну обробку ключових кадрів.

У сучасних версіях CryEngine надає широкий набір інструментів для розробки, зокрема інтегроване середовище розробки (Sandbox), систему візуального програмування (Flowgraph) та підтримку зовнішніх мов програмування, таких як C++ і C#. Це робить його зручним для використання як професійними студіями, так і незалежними розробниками.

Попри значні переваги, CryEngine має певні недоліки, серед яких складність освоєння через велику кількість налаштувань та високі вимоги до апаратного забезпечення. Однак він залишається одним із провідних інструментів у сфері розробки відеоігор завдяки своїм графічним можливостям, гнучкості та постійним оновленням від Crytek.

1.1.4 Unity

Unity – це багатоплатформовий ігровий рушій, розроблений компанією Unity Technologies, який широко використовується для створення інтерактивних додатків та відеоігор. З моменту свого запуску у 2005 році, Unity здобув популярність серед розробників завдяки своїй доступності та широкому спектру можливостей. Unity як й інші перераховані рушії є багатоплатформовим і підтримує розробку для різних платформ, включаючи Windows, macOS, Linux, iOS, Android, а також ігрові консолі, такі як PlayStation та Xbox.

Майже половина всіх ігор на платформі Steam розроблені за допомогою рушія Unity, що свідчить про його популярність серед інді-розробників та великих студій. Unity має власний магазин ресурсів та плагінів, де розробники можуть придбати або безкоштовно завантажити різноманітні ресурси для своїх проєктів, що значно скорочує час розробки.

Скриптова система ігрового рушія підтримує декілька скриптових бекендів для компіляції скриптів. Підгримованою мовою програмування є C#.

Велика кількість напрацьованої за 20 років документації, уроків, модулів, бібліотек та ресурсів роблять Unity універсальним, простим в освоєнні та зручним рушієм, який обирають для проєктів невеликого та середнього розміру.

Всі вище описані середовища є популярними серед розробників. Кожне з них займає свою нішу й долю ринка. Ринок інструментів для розробки ігор

розвивається. Наприклад, ігрова компанія CD Projekt відмовилася від розробки та підтримки свого власного рушія REDengine в тому числі через цю причину, створювати нові проєкти компанія вирішила на легшому в освоєнні та використанні для нових робітників Unreal Engine. Тому для ігрових студій основними критеріями вибору рушія є наступні:

- Адаптація рушія для розробки на цільовій платформі;
- Наявність досвідчених фахівців з технології у штабі;
- Поріг входу рушія для найняття нових співробітників;
- Вбудований візуал;
- Додаткові послуги, наприклад, хмарні технології або засоби аналітики.

Базова комплектація рушія Unity не перевантажена великою кількістю функцій, які переходили від старих версій, але в ньому є все необхідне для початку роботи. Також за необхідності Unity можливо розкришити за допомогою сторонніх плагінів чи написати розширення самотужки. У рушія низький поріг входу, охоплює майже всі популярні платформи й підійде для проєкту над яким буде працювати невелика команда або одна людина. Тому рушій Unity володіє балансом між функціональністю та доступністю і відповідає вимогам майбутнього проєкту.

1.2 Методології розробки

Існують різні методи проєктування комп’ютерних ігор. Кожен має свої недоліки та переваги. При дотриманні однієї методології розробка розгортається швидко, але ймовірно, що кінцевий продукт буде менш якісним, або якщо, наприклад, слідувати іншій методології, то виникає проблема управління проєкт чи його терміни будуть стікати. Будуть описані та проаналізовані головні методи розробки ігор та існуючі проблеми розробки, які їх вирішують.

1.2.1 Водоспадна модель

Водоспадна модель, або каскадна модель, є однією з найстаріших і найвідоміших методологій розробки програмного забезпечення. Вона характеризується послідовним виконанням етапів, де кожен наступний етап починається лише після завершення попереднього. Основні фази цієї моделі включають: збір і аналіз вимог, проєктування системи, реалізацію, інтеграцію та тестування, розгортання та технічне обслуговування.

Перевагами водоспадної моделі є її проста структура та чітка послідовність етапів, що полегшує управління проєктом і контроль за його виконанням. Кожен етап має детальну документацію, що сприяє кращому розумінню проєкту та полегшує його супровід у майбутньому. Ця модель підходить для проєктів з чітко визначеними та стабільними вимогами, де зміни малоймовірні.

Однак водоспадна модель має і недоліки. Головним з них є жорстка фіксація вимог на початкових етапах, що ускладнює внесення змін у процесі розробки. Крім того, тестування відбувається лише на завершальних стадіях, що може призвести до виявлення критичних помилок пізно в процесі, збільшуючи витрати на їх виправлення. Це робить модель менш гнучкою та менш придатною для складних проєктів або тих, де вимоги можуть змінюватися.

Вибір водоспадної моделі доцільний для невеликих проєктів з добре визначеними вимогами та технологіями, що не зазнають частих змін. У випадках, коли проєкт вимагає більшої гнучкості або очікуються зміни в процесі розробки, варто розглянути інші методології.

1.2.2 Ітераційна модель

Ітераційна модель розробки програмного забезпечення передбачає поетапне створення системи через повторювані цикли – ітерації. Кожна ітерація включає всі основні стадії життєвого циклу: аналіз вимог, проєктування,

реалізацію та тестування. На завершення кожного циклу отримується робоча версія продукту з розширеною функціональністю, що дозволяє поступово наближатися до кінцевого результату.

Основною перевагою ітераційної моделі є її гнучкість: вона дозволяє вносити зміни на будь-якому етапі розробки, що особливо важливо для проєктів із неповністю визначеними або змінними вимогами. Крім того, регулярне тестування після кожної ітерації сприяє ранньому виявленню та виправленню помилок, що підвищує якість кінцевого продукту.

Проте ітераційна модель має і недоліки. Постійне внесення змін може призвести до збільшення часу та витрат на розробку. Крім того, необхідність частого тестування та інтеграції вимагає додаткових ресурсів і зусиль від команди розробників.

Загалом, ітераційна модель підходить для проєктів, де важлива гнучкість і можливість адаптації до змінних вимог, а також для складних систем, що потребують поступового вдосконалення та регулярного зворотного зв'язку від користувачів.

1.2.3 Спіральна модель

Спіральна модель розробки програмного забезпечення, запропонована Баррі Боемом у 1986 році, є однією з ключових методологій життєвого циклу програмних систем. Вона поєднує елементи ітеративного підходу та управління ризиками, що дозволяє ефективно розробляти складні та великомасштабні проєкти.

Структура спіральної моделі представлена у вигляді спіралі, де кожен виток відповідає одній ітерації процесу розробки. Кожен виток поділяється на чотири основні фази: визначення цілей, оцінка та аналіз ризиків, розробка та тестування, а також планування наступної ітерації. Такий підхід забезпечує

систематичне виявлення та мінімізацію потенційних ризиків на ранніх етапах проєкту.

Перевагами спіральної моделі є її гнучкість та адаптивність до змінних вимог, що особливо важливо в умовах швидко змінюваного ринку технологій. Крім того, акцент на управлінні ризиками сприяє підвищенню надійності та якості кінцевого продукту. Однак, застосування цієї моделі вимагає значних ресурсів та високої кваліфікації команди, що може обмежувати її використання в малих проєктах.

Спіральна модель є ефективним підходом до розробки програмного забезпечення, який поєднує ітеративність з управлінням ризиками, забезпечуючи тим самим високу якість та відповідність продукту вимогам замовника.

1.2.4 Швидка розробка застосунків

Також деякі ігрові компанії використовують методологію RAD. Швидка розробка застосунків (англ. Rapid Application Development, RAD) – це методологія розробки програмного забезпечення, яка акцентує увагу на швидкому створенні прототипів та отриманні зворотного зв'язку від користувачів з метою прискорення процесу розробки. На відміну від традиційних підходів, таких як водоспадна модель, RAD зосереджується на гнучкості та адаптивності, що дозволяє швидко реагувати на зміни у вимогах та середовищі.

Основні фази RAD включають:

- Визначення вимог – тісна співпраця з користувачами для збору та узгодження вимог до системи;
- Проєктування прототипу – швидке створення робочих моделей системи, які демонструють основні функціональні можливості;

- Конструювання – розробка та вдосконалення системи на основі зворотного зв'язку від користувачів, отриманого під час тестування прототипів;
- Перехід – впровадження та інтеграція готового продукту в реальне середовище експлуатації.

Переваги RAD включають скорочення часу розробки, підвищену гнучкість у реагуванні на зміни та активну участь користувачів у процесі створення продукту. Однак, цей підхід вимагає високої кваліфікації команди розробників та може бути менш ефективним для проєктів з жорсткими обмеженнями або для систем, що потребують високого рівня надійності та безпеки.

Загалом, швидка розробка застосунків є ефективною методологією для проєктів, де ключовими факторами є швидкість виведення продукту на ринок та здатність до адаптації в умовах змінних вимог.

1.2.5. Scrum Agile

Agile – це філософія управління проєктами, що базується на ітеративному та інкрементальному підході до розробки. Вона спрямована на швидке реагування на зміни та тісну співпрацю з замовником. Основні цінності Agile були сформульовані в "Маніфесті Agile", який наголошує на важливості людей і взаємодії над процесами та інструментами, працюючого програмного забезпечення над вичерпною документацією, співпраці з замовником над узгодженням умов контракту та реагуванні на зміни над дотриманням плану.

Scrum є однією з методологій, що реалізує принципи Agile. Це фреймворк для управління проєктами, який організовує роботу команди через серію фіксованих інтервалів часу, відомих як спринти, зазвичай тривалістю від одного до чотирьох тижнів. Кожен спринт завершується створенням потенційно готового до випуску продукту або його частини. Scrum передбачає чітко визначені ролі: власник продукту (Product Owner), який визначає

пріоритети та вимоги; Scrum-майстер (Scrum Master), який забезпечує дотримання процесу та усуває перешкоди; та команда розробки, яка безпосередньо виконує роботу.

Головна ідея Scrum полягає у гнучкості, постійному зворотньому зв'язку та самоорганізації команди. Це дозволяє швидко реагувати на зміни вимог, мінімізувати ризики та забезпечувати високу цінність для замовника на кожному етапі розробки.

Scrum є однією з багатьох методологій, що відповідають принципам Agile. Використання Agile та Scrum дозволяє командам ефективно організовувати робочий процес, швидко адаптуватися до змін та регулярно отримувати зворотний зв'язок, що сприяє підвищенню якості кінцевого продукту та задоволеності замовника.

1.3 Аналіз сучасних проєктів

Основою для реалізації проєкту є жанр ігор roguelike. Roguelike – це піджанр рольових відеоігор, що характеризується процедурною генерацією рівнів, перманентною смертю персонажа з втратою прогресу та покроковим ігровим процесом. З часом жанр еволюціонував, і з'явилися ігри, які поєднують елементи roguelike з іншими жанрами. Такі ігри часто називають roguelite. Основна відмінність між roguelike та roguelite полягає в тому, що в roguelite зазвичай передбачено збереження певного прогресу між спробами, наприклад, збереження частини здобутих ресурсів або покращень персонажа, тоді як у класичних roguelike після смерті весь прогрес втрачається.

Завдяки процедурній генерації, повторюваному ігровому циклу та високій складності ці ігри забезпечують гравцям динамічний і непередбачуваний досвід. Популярність жанру значно зросла за останні десятиліття, що зумовило появу нових механік та підходів до його реалізації. Різні проєкти пропонують унікальні варіанти реалізації: від класичних

покрокових боїв до швидкого ігрового процесу в реальному часі та глибоких наративних компонентів.

Будуть проаналізовані найуспішніші ігри у жанрі roguelike та roguelite, виявленні ключові розробницькі рішення, розглянуті бойові механіки, системи прогресу, інтеграція сюжету та підходи до дизайну та їх вплив на загальну привабливість ігрового процесу та те, що зробило ці проєкти популярними серед гравців і критиків.

1.3.1 Hades

Hades – це популярна roguelike-гра, розроблена студією Supergiant Games та випущена в повному релізі у 2020 році. Гра швидко здобула визнання критиків та гравців завдяки своїй глибокій ігровій механіці, захопливому наративу та унікальному підходу до жанру. Будуть розглянуті основні фактори, що зробили Hades успішною, та практики, які можуть бути корисними для розробки гри.

Однією з головних особливостей Hades є динамічний бойовий процес, що поєднує елементи hack and slash та roguelike. Гравець керує Загреєм, сином Аїда, який намагається втекти з підземного світу, долаючи численних ворогів і босів. Бойова система передбачає комбінацію швидких атак, ухилянь та спеціальних здібностей, а різноманітність зброї вимагає адаптації до нового стилю гри. Додатковий рівень тактичної глибини додають благословення богів: під час кожного забігу Загрей отримує випадкові бонуси від олімпійських богів, що дозволяє експериментувати з різними стратегіями. Попри неминучі невдачі, гра передбачає поступовий прогрес, оскільки гравець отримує ресурси для покращення персонажа, що мотивує повертатися й пробувати нові комбінації.

Однією з ключових інновацій Hades є інтеграція сюжетної частини в roguelike геймплей. На відміну від багатьох ігор жанру, де історія мінімальна, у Hades кожен забіг додає нові діалоги, розвиток персонажів та розширення лору світу. Це створює ефект безперервного прогресу навіть у разі повторних поразок. Гра не має традиційних кат-сцен – історія розповідається через

інтерактивні діалоги та реакції персонажів на дії гравця. Персонажі пам’ятають попередні дії гравця та реагують на них, що створює відчуття живого світу. Hades майстерно адаптує грецькі міфи, надаючи класичним персонажам унікальні характери та сучасний стиль спілкування.

Hades також продемонструвала ефективні розробницькі практики. Використання раннього доступу дозволило розробникам активно взаємодіяти з гравцями та вдосконалювати механіки на основі їхніх відгуків. Гнучкість балансу та величезна кількість можливих комбінацій зброї, здібностей і покращень забезпечили унікальність кожного забігу. Крім того, гра довела, що навіть у межах жанру roguelike можна реалізувати складний і емоційно насичений сюжет.

Hades стала однією з найвпливовіших ігор останніх років завдяки поєднанню продуманого геймплею, динамічного наративу та аудіовізуального стилю. Практики, використані Supergiant Games, можуть стати цінними для розробки гри з високим рівнем залученості гравців та довготривалим успіхом.

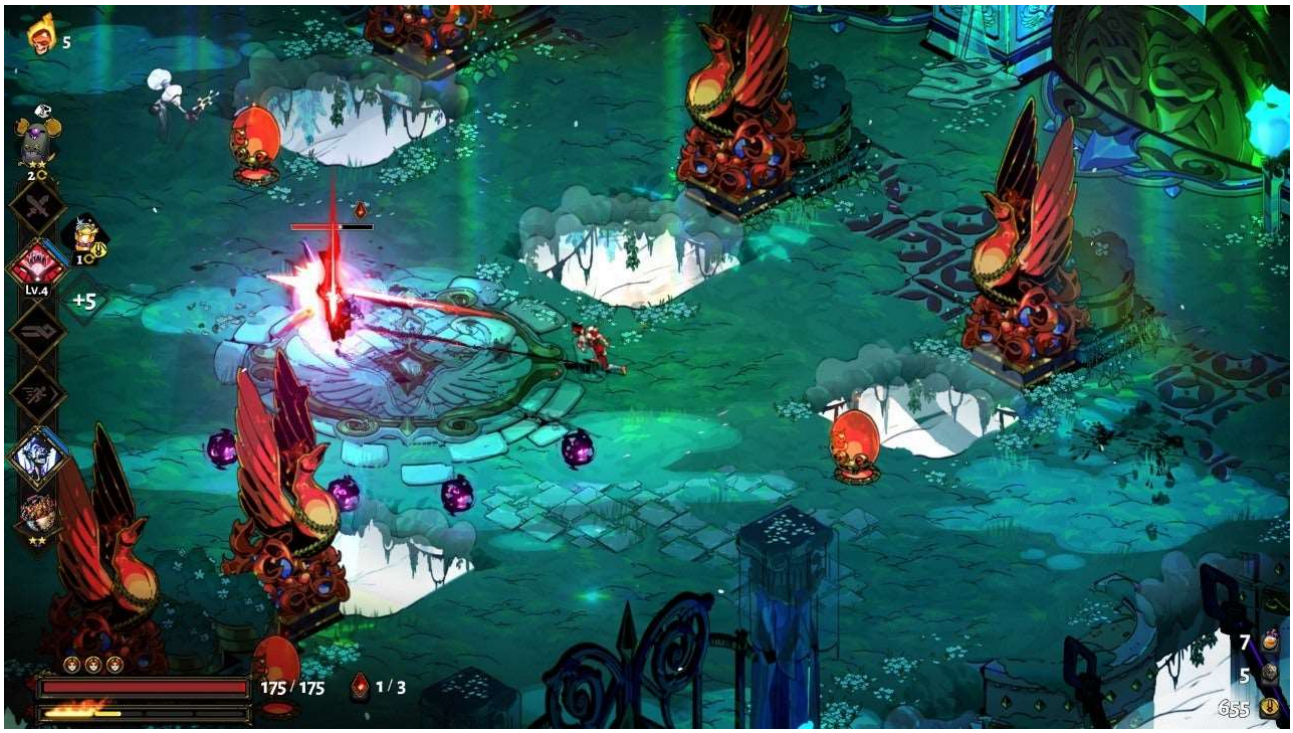

Рис. 1.1 Знімок екрана з гри Hades

1.3.2 Dead Cells

Dead Cells – це roguelike Metroidvania, розроблена Motion Twin та випущена у 2018 році. Гра отримала широке визнання завдяки динамічному бойовому процесу, процедурно генерованим рівням і глибокій системі прогресу.

Dead Cells поєднує швидкий екшен-платформер із глибокими елементами roguelike, пропонуючи гравцям інтенсивний і стратегічний бойовий процес. В грі динамічний бойовий геймплей, бо бойова система побудована навколо використання зброї ближнього та дальнього бою, а також додаткових засобів, таких як пастки та гранати. Це забезпечує гравцеві широку варіативність тактик, заохочуючи експерименти та адаптацію до різних ситуацій.

Однією з ключових особливостей гри є процедурна генерація рівнів, завдяки якій кожен забіг стає унікальним. Розташування ворогів і структура локацій змінюються, що сприяє повторюваності ігрового процесу без втрати інтересу. Прогрес між забігами створюється внаслідок використання системи покращення, гравець збирає "клітини", які можна використовувати для розблокування нової зброї та покращень між забігами. Таким чином, навіть після загибелі персонажа гравець відчуває поступовий розвиток, що мотивує до подальших спроб.

Наратив у Dead Cells подається мінімалістично, залишаючи багато простору для інтерпретацій. Світ гри розкривається через непрямі засоби, зокрема описи предметів, написи на стінах і взаємодію з NPC. Такий підхід стимулює дослідження та створює відчуття загадковості, що є важливим елементом атмосфери. Візуальний стиль і саундтрек підкреслюють постапокаліптичний характер світу, формуючи унікальний естетичний досвід.

Dead Cells також демонструє ефективні організаційні та розробницькі рішення, які сприяли її успіху. Випуск у ранньому доступі дозволив студії Motion Twin отримувати зворотний зв’язок від гравців і вдосконалювати баланс та контент. Глибока бойова система з великим вибором зброї та здібностей

сприяє різноманітності стилів гри, що розширює аудиторію проєкту. Завдяки процедурній генерації та поступовому розвитку персонажа гра підтримує високу мотивацію до повторних проходжень, що стало одним із ключових чинників її популярності.

Dead Cells стала еталоном серед roguelike-платформерів завдяки продуманому бойовому процесу, процедурній генерації рівнів та мотиваційній системі прогресу. Використані розробниками підходи можуть бути корисними для створення ігор, що повинні забезпечувати довготривалий інтерес гравців.

Рис. 1.2 Знімок екрана з гри Dead Cells

1.3.3 Vampire Survivors

Vampire Survivors – це інді-гра в жанрі roguelite, розроблена студією poncle та випущена у 2021 році. Попри просту графіку та мінімалістичний геймплей, гра швидко здобула величезну популярність завдяки своїй високій реіграбельності. Вона використовує елементи виживання, автоматичних атак та поступового розвитку персонажа, що робить її доступною, але водночас захопливою для гравців.

Гравець керує персонажем, який автоматично атакує ворогів, а завданням є виживати якомога довше на процедурно згенерованій карті. Основний акцент робиться на збиранні досвіду, отриманні нових здібностей і зброї, а також на грамотному виборі покращень, щоб створити потужні комбінації. Кожен забіг триває певний час, після чого настає фінальний виклик у вигляді масового напливу ворогів, який практично неможливо пережити.

Попри просте управління, гра пропонує значну глибину через систему розвитку. Гравець має можливість комбінувати різні здібності та зброю, що дає змогу створювати унікальні стратегії для кожного забігу. Успіх залежить не тільки від випадковості випадіння бонусів, а й від грамотного вибору покращень. Це створює відчуття контролю над хаосом і постійне бажання вдосконалювати свої результати.

Vampire Survivors показала, що інновації у геймплеї можуть бути важливішими за візуальні технології. Гра довела, що прості механіки з правильним балансом ігрового процесу можуть створити ефект сильної залученості, що є ключовим фактором її популярності. Вона також стала прикладом того, як невелика незалежна студія може досягти великого успіху без величезного бюджету, використовуючи ранній доступ і активну взаємодію з гравцями для покращення гри.

Vampire Survivors стала важливою віхою в жанрі roguelite, запропонувавши гравцям захопливий досвід виживання з мінімальним керуванням та глибокими механіками розвитку. Її успіх доводить, що добре спроєктований геймплей може стати головним рушієм популярності навіть за відсутності високоякісного візуала. Ця гра є чудовим прикладом того, як можна створювати ігри, що утримують увагу гравців завдяки грамотному дизайну прогресії та механіки випадкових, але стратегічних виборів.

Рис. 1.3 Знімок екрана з гри Vampire Survivors

1.3.4 Enter the Gungeon

Enter the Gungeon – це динамічний roguelike-шутер, розроблений студією Dodge Roll та випущений у 2016 році. Гра швидко завоювала популярність завдяки поєднанню стрімкого екшену, процедурно генерованих рівнів і величезної кількості зброї. Вона вдало використовує унікальну стилістику та гумор, що дозволило їй виділитися серед інших представників жанру.

Основна механіка Enter the Gungeon базується на швидких перестрілках, де гравець має ухилятися від куль, використовувати оточення та комбінувати різноманітну зброю. Кожен рівень створюється випадковим чином, що гарантує унікальні виклики під час кожного проходження. Система зброї та предметів відіграє ключову роль у створенні ігрового досвіду: гравці можуть знайти сотні унікальних гармат, кожна з яких має свої особливості та часто містить відсилання до попкультури.

Рівні поділені на кімнати, наповнені ворогами, пастками та секретами. Персонажі, доступні для гри, мають свої стартові набори зброї та здібностей, що впливає на стиль гри. Важливою механікою бойової системи є ухиляння за

допомогою спеціального перекату, який дозволяє тимчасово ставати невразливим. Гравці також можуть використовувати столи для укриття, перекидаючи їх у пошуках захисту.

Enter the Gungeon стала одним із найкращих представників roguelike-шутерів, встановивши високу планку для жанру. Розробники вдало поєднали хаотичний екшен, гумор і глибоку систему зброї, що забезпечило грі довговічність та високу реіграбельність. Використання процедурної генерації разом із ретельно продуманою бойовою системою створює відчуття унікального проходження щоразу. Гра також продемонструвала ефективний підхід до оновлень, оскільки розробники протягом кількох років після релізу продовжували підтримувати її новим контентом, що дозволило зберегти інтерес спільноти.

Enter the Gungeon завдяки швидкому геймплею, розмаїттю зброї та стильному візуальному оформленню змогла стати культовою грою в жанрі roguelike-шутерів. Її успіх базується на грамотному балансі між викликом і задоволенням від гри, а також на унікальному гумористичному стилі. Використані розробниками підходи можуть бути корисними для створення ігор, що націлені на довготривалу зацікавленість гравців.

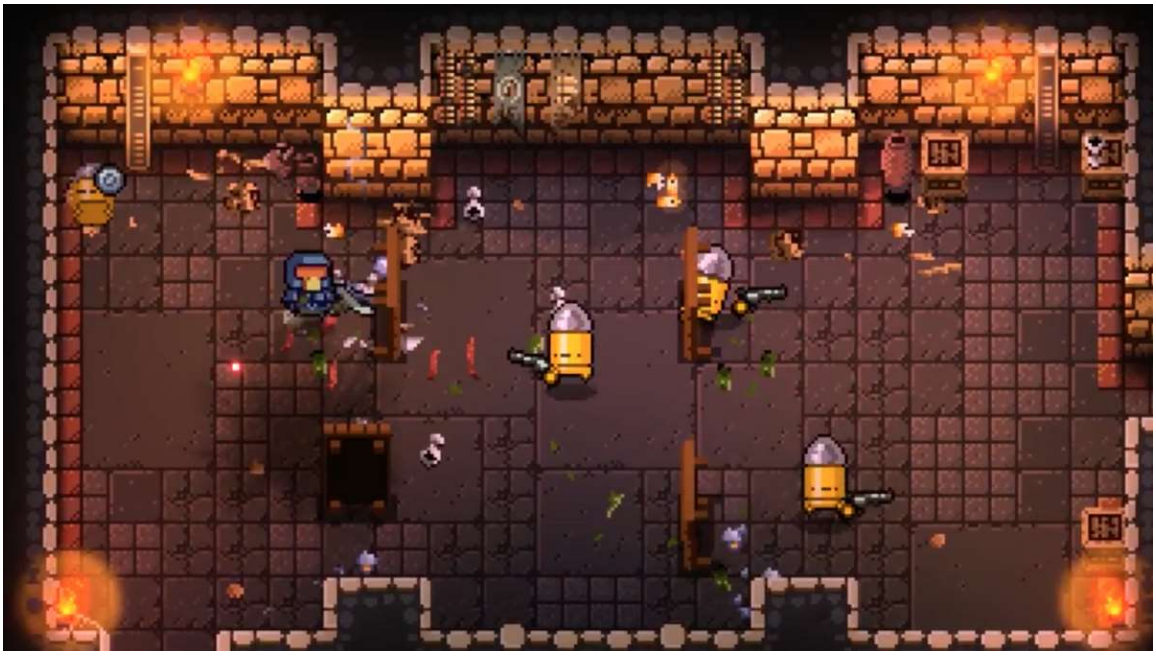

Рис. 1.4 Знімок екрана з гри Enter the Gungeon

1.4 Парадигми програмування

1.4.1 Об'єктно-орієнтоване програмування.

Об'єктно-орієнтоване програмування – це одна з парадигм програмування, яка розглядає програму як множину об’єктів, що взаємодіють між собою. Основу ООП складають такі концепції:

- Класи та об’єкти. Клас є шаблоном для створення об’єктів, визначаючи їх атрибути та методи. Об’єкт – це конкретний екземпляр класу з власним станом;
- Інкапсуляція – приховання внутрішніх даних компонентів і деталей його реалізації від інших компонентів програми та надання набору методів для взаємодії з ними. Інкапсуляція досягається шляхом вказування, які класи можуть звертатися до членів об'єкта. Як наслідок, кожен об'єкт надає кожному іншому класу певний інтерфейс, який доступний іншим класам;
- Успадкування – механізм, що дозволяє створювати нові класи на основі вже створених. Похідний клас успадковує властивості та методи базового класу, що сприяє повторному використанню коду та спрощує його модифікацію;
- Поліморфізм – здатність об'єктів різних класів реагувати на однакові повідомлення по-різному. Це дозволяє використовувати єдиний інтерфейс для взаємодії з об'єктами різних типів, підвищуючи гнучкість та розширюваність системи;
- Абстрагування – теоретичний спосіб дослідження, який дозволяє відсторонитися від деяких несуттєвих, у певному сенсі, властивостей досліджуваних явищ і виокремити суттєві та визначальні властивості.

Основними перевагами Об'єктно-орієнтованого програмування має декілька переваг. По-перше, модульність: код розділяється на незалежні класи, що спрощує його розуміння тестування та супровід. По-друге, повторне використання коду: наслідування дозволяє створювати нові класи на основі

створених, зменшуючи дублювання коду. По-третє, гнучкість та розширюваність: поліморфізм та абстракція сприяють легкому розширенню та модифікації функціональності програми без значних змін у коді.

Алан Кей, відомий як один із засновників об'єктно-орієнтованого програмування (ООП) та розробник мови Smalltalk, визначив кілька ключових принципів цієї парадигми:

- Усе є об'єктами. В ООП будь-яка сутність розглядається як об'єкт із власними властивостями та поведінкою;
- Обчислення здійснюються через взаємодію об'єктів. Об'єкти обмінюються повідомленнями, запитуючи один в одного виконання певних дій та передаючи необхідні аргументи;
- Кожен об'єкт має власну пам'ять. Ця пам'ять може складатися з інших об'єктів, що дозволяє будувати складні структури даних;
- Об'єкти є екземплярами класів. Клас визначає загальні властивості та поведінку групи об'єктів, які є його екземплярами;
- Поведінка об'єкта визначається його класом. Усі об'єкти одного класу можуть виконувати однакові дії, визначені в цьому класі;
- Класи організовані в ієрархію успадкування. Це означає, що властивості та поведінка, визначені в одному класі, автоматично доступні його підкласам, що сприяє повторному використанню коду та спрощує моделювання складних систем.

Наразі кількість прикладних мов програмування, що реалізують об'єкто-орієнтовану парадигму, є найбільшим стосовно інших парадигм. Усі інші далі розглянуті принципи й патерни опираються на ООП. Тому використання об'єктно-орієнтованого програмування у проєкті дозволить створити структуровану, легку для розширення систему.

1.4.2 Принципи SOLID

Принципи SOLID – це п’ять фундаментальних принципів об'єктно-орієнтованого програмування та дизайну, які були вперше сформульовані Робертом Мартіном. Дотримання цих принципів допомагає розробляти програмне забезпечення, яке є більш зрозумілим, гнучким, підтримуваним та розширюваним.

Розглянемо сутність кожного принципу, його значення для архітектури програмних систем:

- Принцип єдиної відповідальності (Single Responsibility Principle - SRP): Клас повинен мати лише одну причину для зміни. Іншими словами, клас повинен відповідати лише за одну конкретну частину функціональності. Якщо клас виконує кілька різних завдань, будь-яка зміна в одному з цих завдань може призвести до непередбачених наслідків в інших частинах класу. Розділення відповідальностей робить код більш модульним і зменшує ризик внесення помилок при модифікації;
- Принцип відкритості/закритості (Open/Closed Principle - OCP): Програмні сутності (класи, модулі, функції тощо) повинні бути відкритими для розширення, але закритими для модифікації. Це означає, що поведінку програмної сутності можна змінювати шляхом додавання нового коду, але не шляхом зміни написаного коду. Досягається це зазвичай за допомогою абстракцій (інтерфейсів, абстрактних класів) та поліморфізму. Таким чином, можна розширювати функціональність, не порушуючи стабільність вже написаного коду;
- Принцип підстановки Ліскoв (Liskov Substitution Principle - LSP): Об'єкти в програмі можуть бути замінениими їхніми нащадками без зміни коду програми. Цей принцип стверджує, що якщо клас B є підтипом класу A, то об'єкти типу A в програмі повинні бути замінені об'єктами типу B без зміни коректності програми. Це вимагає, щоб похідні класи наслідували поведінку базових класів таким чином, щоб вони могли використовуватися в будь-якому контексті, де очікується базовий клас;

- Принцип розділення інтерфейсу (Interface Segregation Principle - ISP): Клієнти не повинні залежати від інтерфейсів, які вони не використовують. Цей принцип говорить про те, що краще мати багато невеликих, специфічних інтерфейсів, ніж один великий, універсальний інтерфейс. Якщо клас реалізує великий інтерфейс, але використовує лише частину його методів, він стає залежним від методів, які йому не потрібні. Розділення інтерфейсів робить систему більш гнучкою та зменшує кількість непотрібних залежностей;
- Принцип інверсії залежностей (Dependency Inversion Principle - DIP): Модулі верхнього рівня не повинні залежати від модулів нижнього рівня. Обидва повинні залежати від абстракцій. Абстракції не повинні залежати від деталей. Деталі повинні залежати від абстракцій. Цей принцип передбачає, що залежності між модулями повинні бути інвертовані. Замість того, щоб модулі верхнього рівня залежали від конкретних реалізацій модулів нижнього рівня, обидва повинні залежати від абстракцій (інтерфейсів або абстрактних класів). Це дозволяє легко замінювати реалізації модулів без впливу на модулі вищого рівня, роблячи систему більш гнучкою та тестувальною.

Принципи SOLID дуже тісно корелюють з принципами об'єктно-орієнтованого програмування. Принцип підстановки Лісков фактично є практичною реалізацією успадкування, гарантуючи, що похідні класи можуть замінювати базові. Принцип інверсії залежностей розвиває ідею поліморфізму, роблячи код гнучкішим. А принцип єдиної відповідальності підсилює важливість інкапсуляції, зосереджуючи функціональність в окремих, чітко визначених одиницях. Дотримання принципів SOLID є важливою практикою для створення якісного, підтримуваного та розширюваного програмного забезпечення.

1.4.3 Entity Component System

Entity Component System – це архітектурний патерн, створений спеціально для розробки ігор, він гарно підходить для опису динамічного віртуального світу. Головною його особливістю є ділення проєкту на три концепції:

- Entity – сутність, абстрактний об’єкт, умовний контейнер для компонентів, що будуть визначати чим буде ця сутність. Найчастіше подається у вигляді ідентифікатора для доступу до даних;
- Component – компонент, характеризує об’єкт як такий, що володіє певним аспектом, і містить дані, необхідні для моделювання цього аспекту. Компоненти ECS повинні містити виключно чисті дані, без логіки. Реалізації зазвичай використовують структури, класи або асоціативні масиви;
- System – система, логіка обробки даних. Система це процес, який діє на всі сутності з потрібними компонентами. Наприклад, фізична система може запитувати сутності, що мають компоненти маси, швидкості та положення, і повторювати результати, виконуючи фізичні обчислення набору компонентів для кожної сутності. Системи ECS не повинні містити жодних даних, тільки логіка обробки даних.

ECS дотримується принципу композиції над успадкуванням, тобто кожна сутність визначається не ієрархією типів, а пов’язаними з нею компонентами. Невідмінну від класичного об’єктно-орієнтовного програмування поведінка об’єкта визначається не інтерфейсами, контрактами або публічним API, а наданими об’єкту властивостями з даними й окремо присутньою логікою обробки.

ECS має свої особливості які можуть бути корисними так і створювати складності, створюючи обмеження, які треба обходити. Перевагами є:

- Слабка зв'язність коду. В проєкті легше розширювати та проводити рефакторинг не змінюючи старий код;

– Модульність й тестування коду. Основа, яка побудована на даних дозволяє легко копіювати логіку з одного проєкту в інший та полегшує тестування, бо можливо запускати логіку на будь-яких вхідних даних;
– Захист від поганого проєктування. ECS менш вимогливий до архітектури, тому задає рамки з якими складніше створити поганий дизайн коду;
– Можливість легко комбінувати компоненти. Патерн гарно підходить для опису динамічних світів, бо компоненти можливо додавати до будь-якої сутності й системи будуть коректно на це реагувати;
– Приріст продуктивності. ECS може збільшити продуктивність проєкту через перевикористання логіки для різних даних. Патерн особливо оптимальний в контексті рушія Unity, бо такий підхід більш продуктивний ніж стандартна архітектура.

Entity-Component-System також має такі недоліки:

– Проблеми зі слабкою зв’язністю. З причини високої абстрактності сутностей не можна гарантувати конкретну комбінацію компонентів, що може викликати складності. Однак така особливість спонукає до передбачення кожного випадку для можливості подальших змін;
– Погана підтримка структур даних. Реалізація структур даних за допомогою ECS не є оптимальним його використанням;
– Відкритий доступ до будь-яких даних. Всі дані в ECS є публічними, що може створити проблеми при неправильному використанні.

Патерн Entity-Component-System набув значної популярності в розробці ігор завдяки своїй гнучкості та ефективності. ECS може стати гарним варіантом архітектури для створення ігор, але не є універсальним. Вибір повинен базуватися на специфіці проєкту та вимогах до нього. ECS може стати майбутнім розробки інтерактивних розваг, бо він має вигідні в контексті розробки ігор переваги.

1.5 Висновки до розділу

У рамках аналітичного розділу були розглянуті популярні ігрові рушії, зокрема Godot, Unreal Engine та CryEngine, були проаналізовані їх переваги та недоліки у контексті реалізації проєкту. Для рушія Unity, який буде обрано як оптимальне середовище розробки, були описані особливості, а також методології, які застосовуються в процесі створення ігрового програмного забезпечення. Unity надає гнучкі можливості для розробки кросплатформенних ігор, має низький поріг входу, потужну підтримку спільноти та широкий вибір інструментів і плагінів.

Були досліджені основні методології розробки програмного забезпечення, зокрема водоспадна, ітераційна, спіральна моделі, методологія RAD та фреймворк Scrum. Аналіз показав, що вибір методології має ґрунтуватися на специфіці проєкту, обсязі роботи, ресурсах команди та гнучкості вимог.

Також було описано актуальні підходи до архітектури програмного забезпечення, включаючи об’єктно-орієнтоване програмування, принципи SOLID та патерн ECS. Ці концепції сприяють створенню структурованої, масштабованої та легко підтримуваної системи, що є необхідним для сучасних ігрових продуктів.

Були переглянуті популярні та успішні проєкти й проаналізовані їх геймплейні рішення, механіки та системи, що дозволило виокремити ключові фактори успіху для майбутнього їх використання.

Отриманні знання необхідні для проєктування та реалізації майбутнього проєкту прототипу ігрового застосунку та сприяють глибшому розумінню процесів розробки ігор й формуванню власного підходу до проєктування ігрової системи.

2. МОДЕЛЬ ПРОЄКТУ ТА ПОСТАНОВКА ЗАДАЧІ

2.1 Визначення необхідних компонентів проєкту

Для початку роботи над прототипом проєкту необхідно визначити ключові механіки, які треба реалізувати. Головною ціллю є створення повної ігрової системи, що є самодостатньою для ознайомлення користувачами.

Архітектура проєкту повинна забезпечувати структурованість та можливості підтримки проєкту. Також потрібно використовувати алгоритми та архітектурні шаблони проєктування, щоб збільшити модульність компонентів системи, зробити можливим перевикористання коду та спростити його тестування.

Щоб забезпечити динамічний ігровий досвід, де гравець відчуває контроль над персонажем та комфорт від процесу гри необхідно детально проробити взаємодію об’єктів у просторі та реагування персонажа на сигнали з пристроїв вводу. Ворогам треба мати складну поведінку та повинні динамічно реагувати на зміну ситуації в грі. Вони повинні орієнтуватися в ігровому середовищі та ефективно переміщатися з урахуванням перешкод, координуючи переміщення між собою.

Також необхідно реалізувати бойовий процес, що є основою для цільового ігрового жанру. Він повинен виконувати коректну, різноманітну та гнучку взаємодію між великою кількістю об’єктів у реальному часі.

Тому для геймплейних та архітектурних потреб буде реалізовано наступні системи:

- Локатор сервісів: система, що структурує системи застосунку, систематизуючи їх взаємодію за допомогою реєстрації кожної системи та надання доступу до них;
- Шина подій: статичний об’єкт, що дозволяє централізовано обробляти події;

- Система керування персонажем: обробляє ввід гравця та на основі фізичної взаємодії рухає об’єкт персонажа;
- Поведінкова система штучного інтелекту: побудована на основі шаблону State Machine, що спрощує реалізацію складної логіки об’єктів, розділяючи її на різні стани та переходи між ними;
- Система навігації: використовує пошук найкоротшого шляху у графі для забезпечення навігації штучного інтелекту ворогів з метою пошуку гравця;
- Система ройової поведінки: алгоритм, що дозволяє ворогам координовано пересуватися та уникати зіткнень;
- Бойова система: обробляє взаємодію між ворогами та персонажем у тривимірному просторі, використовуючи фізичну модель ігрового рушія;
- Пул об’єктів: збільшує продуктивність застосунку за допомогою повторного використання вже створених об’єктів, що зберігаються в оперативній пам’яті пристрою;

2.2 Визначення технічних засобів

Після аналізу різних середовищ розробки ігор зважаючи на всі критерії було вирішено, що розробка прототипу ігрового застосунку буде виконуватись на рушії Unity (Рис 2.1).

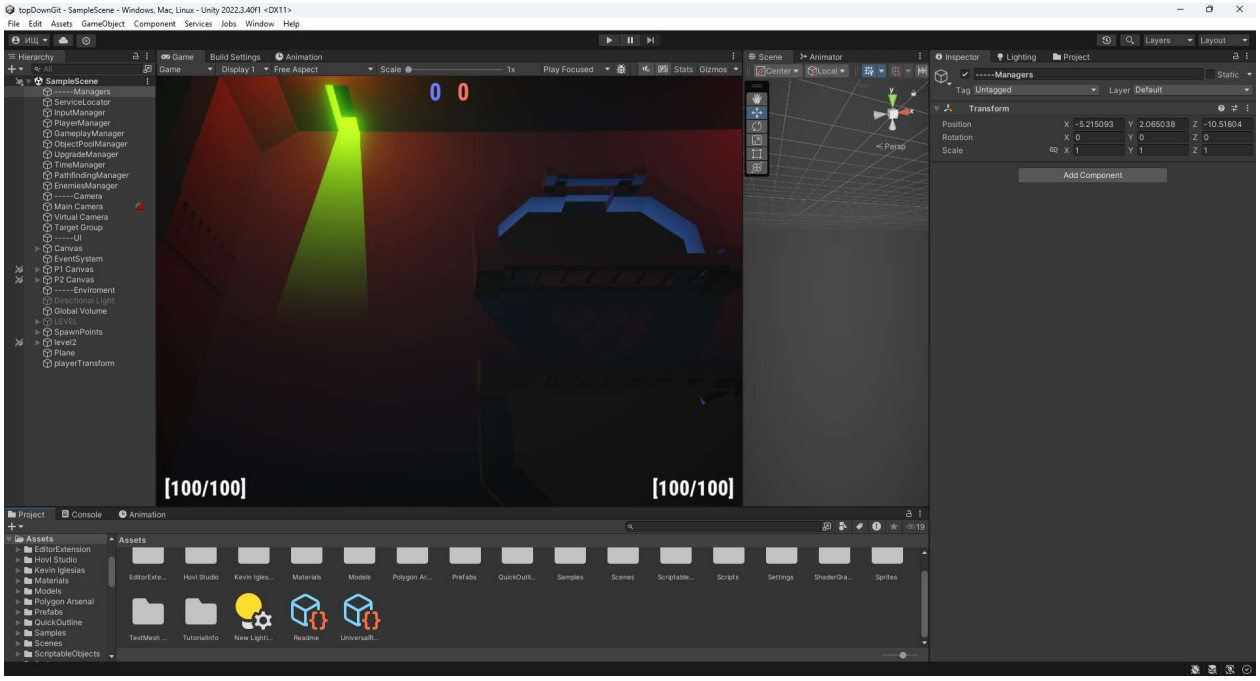

Рис. 2.1 Інтерфейс програми Unity

Цільовою платформою для проєкту стане комп’ютер на операційній системі Windows, а код гри написаний мовою програмування C# у програмі JetBrains Rider (Рис 2.2). JetBrains Rider 2023.1.3 – це інтегроване середовище розробки, що розроблений компанією JetBrains. JetBrains Rider працює з технологіями .NET, ASP.NET, .NET Core, Xamarin, а також використовується для розробки ігор на Unity та Unreal Engine.

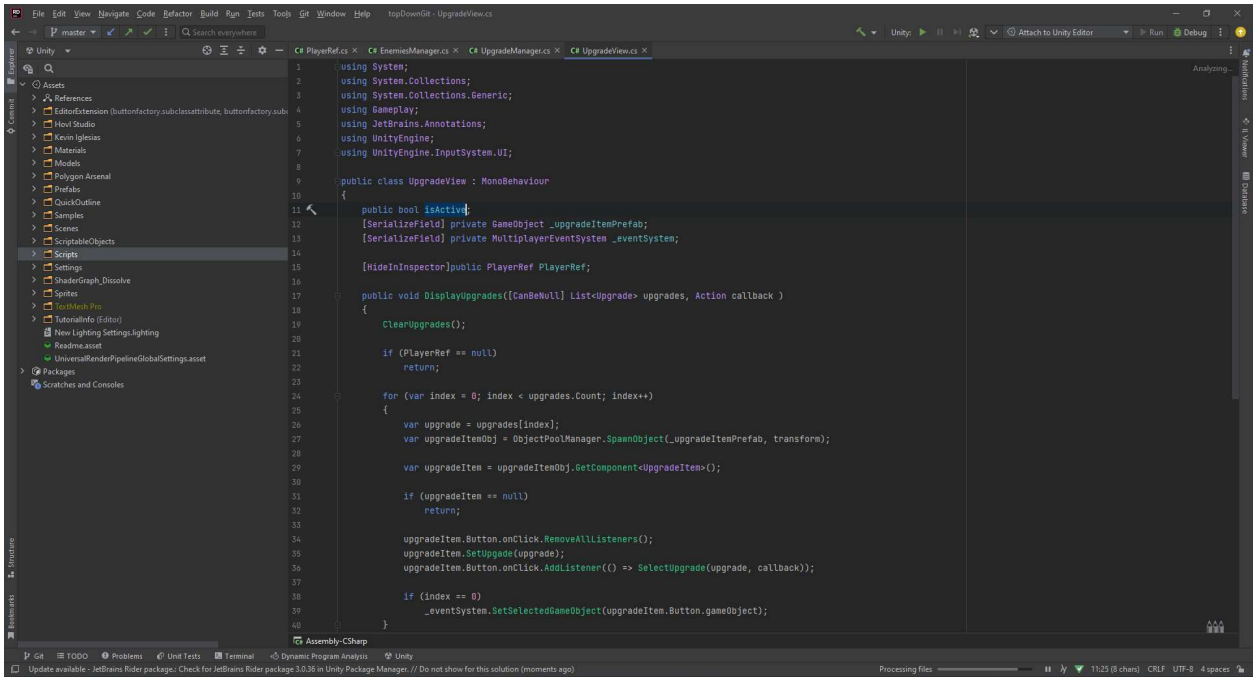

Рис. 2.2 Інтерфейс програми Rider

3. РЕАЛІЗАЦІЯ ІГРОВОГО ЗАСТОСУНКУ

3.1 Вбудований функціонал рушія

3.1.1 Клас MonoBehaviour

MonoBehaviour – це базовий клас від якого успадковуються класи, які є скриптами й повинні взаємодіяти з логікою рушія Unity. MonoBehaviour надає широкий спектр вбудованих функцій для управління ігровими об'єктами, реагування на введення гравця, управління рендерингом та багато іншого. Нижче наведені деякі з методів, що найчастіше використовуються в ігрових проєктах:

- Awake() : викликається під час ініціалізації об’єкта, до виклику інших методів. Він використовується для ініціалізації змінних, налаштування посилань на компоненти, а також для виконання будь-яких налаштувань, які мають відбутися до активації об'єкта.
- OnEnable() : викликається щоразу, коли об'єкт активується. Часто використовується для підписки на події або активації певної поведінки.
- Start() : викликається один раз при першому увімкненні об'єкта після Awake(). Зазвичай використовується для запуску логіки, яка потребує попередньої ініціалізації об'єкта або посилань.
- Update() : викликається кожен кадр, часто прив'язаний до частоти оновлення екрана. Цей метод зазвичай використовується для опрацювання введення гравця та для інтерполяції зміни положення, обертання чи інших параметрів, що змінюються в реальному часі.
- FixedUpdate() : викликається через рівні проміжки часу, незалежно від частоти кадрів (наприклад, 50 разів на секунду). Застосовується для фізичних оновлень, зокрема для руху за допомогою Rigidbody.
- LateUpdate() : викликається після Update() кожного кадру. Цей метод часто використовується для оновлення положення та обертання

ігрового об'єкта на основі положення інших ігрових об'єктів, наприклад камери, яка слідує за гравцем.

- OnDisable() — викликається щоразу, коли об'єкт дезактивується. Часто використовується для скасування підписок на події або зупинки певної поведінки.
- OnDestroy() — викликається перед остаточним знищенням об'єкта. Служить для очищення ресурсів, завершення з'єднань тощо.

Ініціювати об’єкти та системи можливо через методи *Start()* та Awake(), але при такому сценарії їх використання ми не контролюємо порядок виконання й не враховуємо єрархію об’єктів. Тому також є необхідність в структурі, яка буде контролювати в якому порядку ініціалізуватися системами та буде централізовано зберігати посилання на них.

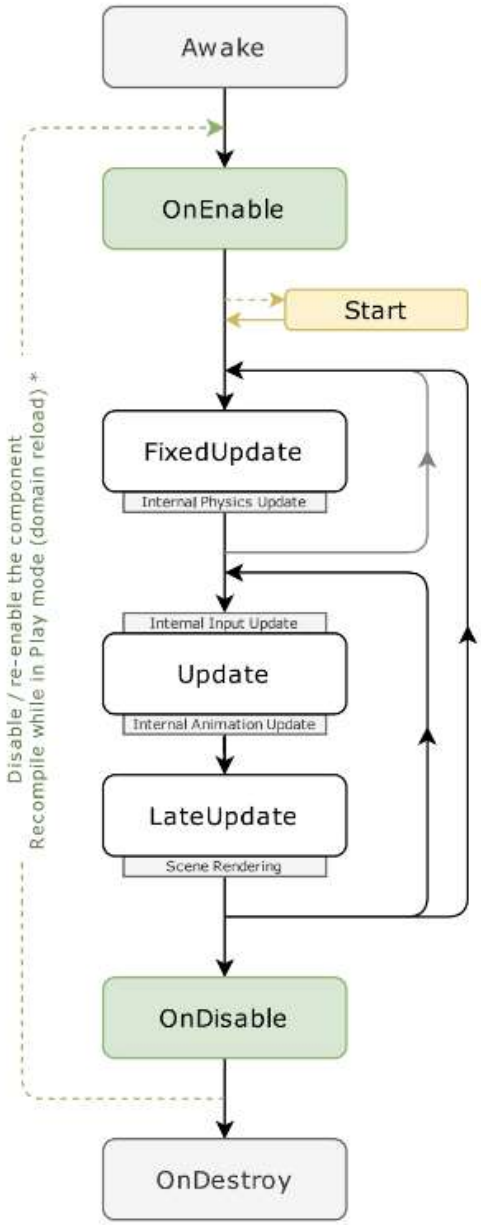

Рис. 3.1 Порядок виконання викликів методів

3.1.2 Користувацький Інтерфейс

Рушій Unity надає готове рішення для користувацького інтерфейсу, яке є зручним для інтеграції у проєкт. Основою побудови інтерфейсу в Unity є об’єкт Canvas, який слугує контейнером для всіх інших UI-елементів. У межах Canvas можуть розміщуватися текстові поля, зображення, кнопки, списки та інші елементи, необхідні для побудови інтерфейсу. Для роботи з текстовими

елементами найчастіше використовується компонент TextMeshPro, який дозволяє створювати гнучкий у налаштуванні текст.

Інтерактивні елементи, такі як кнопки, пов’язуються з логікою проєкту через систему подій. Наприклад, для обробки натискання кнопки достатньо призначити відповідний метод у скрипті, який буде викликано при активації події. Це можливо зробити як і безпосередньо у редакторі Unity, так і реалізуванням обробника події у коді скриптів.

Для того, щоб масштабувати елементи інтерфейсу відповідно до розміру або роздільної здатності екрана використовується компонент CanvasScaler. Він дозволяє налаштувати поведінку елементів при зміні екранного простору. У режимі. Існує три режими масштабування: Constant Pixel Size, при якому елементи зберігають фіксований піксельний розмір незалежно від роздільної здатності; Scale With Screen Size, що дозволяє адаптувати UI до різних екранів, зберігаючи пропорції завдяки заданій роздільній здатності; та Constant Physical Size, де розміри елементів визначаються у фізичних одиницях.

3.1.3 Робота з Анімаціями

Важливим елементом кожної гри анімації. Робота з анімаціями в рушії Unity відбувається з допомогою компонента Animator та Animator Controller. Для кожного об’єкта який необхідно анімувати необхідно створити екземпляр Animator Controller, який містить вбудовану машину станів. Кожен стан у такій машині, як правило, відповідає певному Animation Clip – наприклад, стан очікування, ходьбі або бігу, які мають інформацію про ключові кадри анімації об’єкта. Animator Controller зберігає посилання на кліпи та визначає між ними переходи. Animator Controller прикріплюється до GameObject через компонент Animator, і під час виконання гри відтворює кліп активного стану.

Transition в Animator – це засіб плавного перемикання анімації з одного стану в інший. Перехід визначає умови спрацьовування і тривалість змішування

між станами. Наприклад, перехід «Idle → Attack» може бути активований, коли спрацьовує тригер Attack. Також можливо налаштовувати швидкість переходу та його початок.

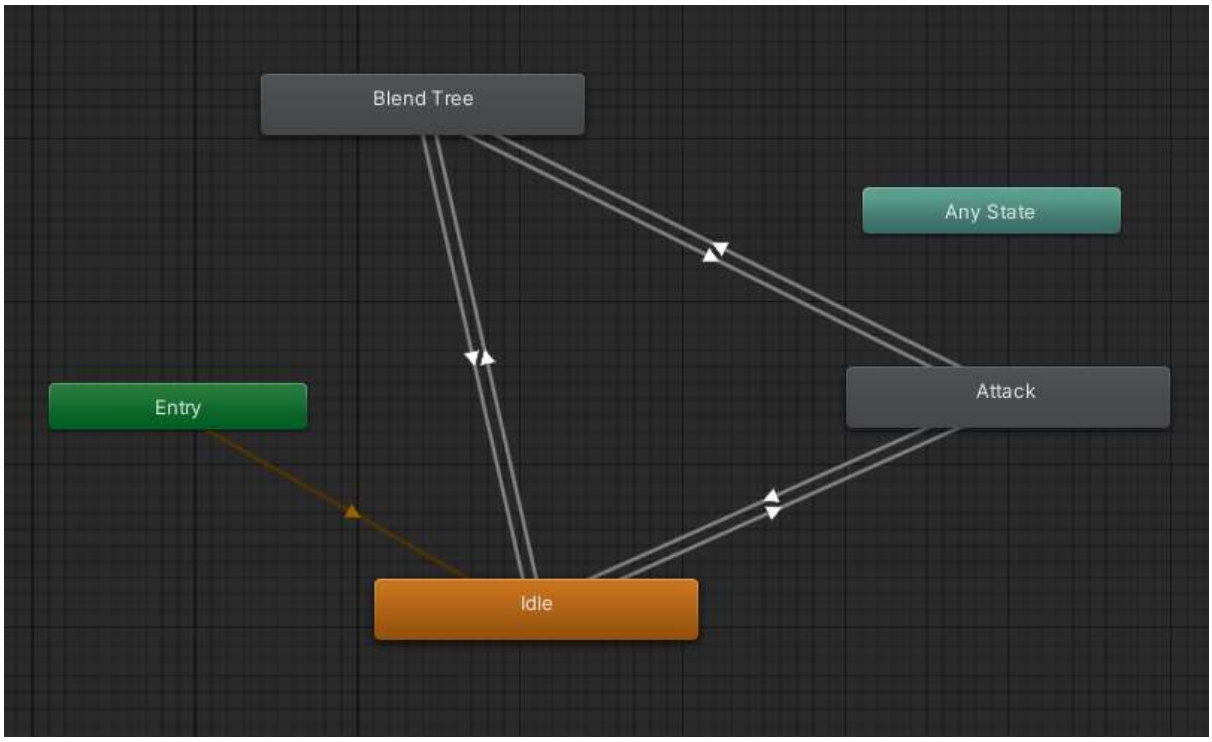

Рис. 3.2 Реалізація машини станів в компоненті Animator.

Керувати анімаціями в Animator Controller можливо безпосередньо у коді скриптів. Після отримання посилання на компонент Animator потрібно викликати необхідні для переходу методи. Наприклад, Animator.SetTrigger("Attack") робить виклик переходів, які були налаштовані на тригер Attack.

Для того, щоб плавно поєднувати декілька анімаційних кліпів на основі одного або кількох параметрів у рушії використовується Blend Tree. Цей механізм спрощує об’єднання схожих станів в один, який можливо гнучко налаштовувати. Blend Tree працює як окремий стан в Animator Controller, який усередині містить логіку змішування кількох анімацій. У проєкті Blend Tree використовується для плавного перемикання анімацій пересування персонажу у різних напрямах.

3.1.4 Скриптовані об'єкти

Для зберігання даних та більш гнучкого та зручного редагування великої кількості об’єктів в проєкті використовуються об’єкти Scriptable Object. Scriptable Objects є серіалізованими класами Unity, які функціонують як контейнери даних, що дозволяють зберігати значні обсяги даних незалежно від конкретних екземплярів скриптів.

На відміну від класів, які наслідуються від MonoBehaviours, вони не прикріплюються до ігрових об'єктів у сцені, а зберігаються як ресурси проєкту у файлах проєкту. Це дозволяє повторно використовувати їх у різних сценах та ігрових об'єктах не створюючи копії у кожному компоненті, а використовувати як посилання, що дозволяє ефективніше використовувати пам’ять та зменшує вагу проєкту.

Для використання Scriptable Objects в рушії Unity необхідно створити клас, який буде наслідуватися від класу ScriptableObject й за необхідності прописати для нього атрибут, який створить розділ для створення об’єкту в редакторі Unity. Наприклад, наступний код додасть опцію у розділі ScriptableObjects з назвою Upgrade для створення файлу-об’єкту Upgrade.

```
[CreateAssetMenu(fileName = "Upgrade", menuName =
"ScriptableObjects/Upgrade")]
```

Scriptable Objects допомагають зменшити зв’язність між об’єктами та відокремлюють дані від логіки, що позитивно впливає на архітектуру проєкту. Також Scriptable Object можна редагувати прямо в редакторі Unity, що зручно для редагування ігрового дизайну без змін в коді.

3.1.5 Співпрограми

Coroutines – це механізм співпрограм, який дозволяє виконувати відкладені дії в межах одного кадру або протягом кількох кадрів, не блокуючи основний потік виконання гри. Вони реалізуються як методи, що повертають

тип IEnumerator, і використовують ключове слово yield для тимчасового призупинення свого виконання. Завдяки цьому корутини дозволяють програмістам керувати часом виконання певної логіки без використання складних багатопотокових структур. Корутини запускаються за допомогою методу StartCoroutine, а зупинити їх можна через StopCoroutine або StopAllCoroutines.

В проєкті співпрограми застосовуються для реалізації таких сценаріїв, як затримки між подіями, поступове завантаження ресурсів, а також для створення поведінки, яка залежить від часу, за допомогою корутин можна реалізувати відкладену появу об'єкта після певного проміжку часу або інші подібні дії.

Корутини працюють в межах основного потоку виконання гри, тобто є однопоточними. Це означає, що, на відміну від багатопотокових процесів вони не створюють нових потоків, а лише призупиняють своє виконання на певний час або до настання визначеної умови, дозволяючи іншим частинам коду продовжити свою роботу. Такий підхід забезпечує безпечну взаємодію з елементами сцени, оскільки більшість об’єктів Unity не є потокобезпечними й будь-яка взаємодія з ними повинна відбуватися саме в головному потоці.

3.1.6 Таймер

Також для розширення функціоналу було написано клас Timer, який у деяких випадках практичніше використовувати для розв’язання проблем. Створений екземпляр класу відстежує актуальне значення часу у рушії й сповіщає коли проходить конкретний проміжок часу.

Клас містить приватні поля для збереження параметрів таймера: duration визначає тривалість таймера у секундах, startTime фіксує момент його запуску, isRunning сигналізує про активний стан таймера. Метод CreateFromSeconds(float seconds) дозволяє задати тривалість таймера у секундах

і запускає його, якщо той не був активним на момент виклику. Таким чином запобігається повторне ініціювання вже активного таймера.

Метод Reset() здійснює повторну ініціалізацію таймера, із можливістю задання нової тривалості. При цьому оновлюється час запуску і знову встановлюється активний стан. Метод IsFinished() виконує перевірку на завершення таймера шляхом порівняння поточного часу (Time.time) із сумою часу запуску та тривалості. У разі завершення таймера метод повертає true і деактивує таймер, в іншому випадку – false.

Метод GetRemainingTime() обчислює залишковий час до завершення таймера, повертаючи максимальне значення між нулем та різницею між тривалістю і вже витраченим часом. У випадку неактивного таймера повертається нуль. Метод GetProgress() надає нормалізоване значення прогресу у вигляді дробового числа в межах [0, 1], що відображає частку пройденого часу від загальної тривалості.

Клас Timer виконує роль зручного інструмента для реалізації відліку часу. Його можливо швидко перевикористовувати у різних місцях без повторного написання логіки, оскільки для різних задач потрібна аналогічна логіка.

3.1.7 Raycast

Для того, щоб прораховувати зіткнення у середовищі Unity без необхідності використання фізичних об'єктів з колайдерами існує механізм Raycast, який є частиною фізичної системи рушія. Raycast – це процес випускання віртуального променя з певної точки у певному напрямку з метою виявлення перетину з об'єктами сцени. Даний механізм базується на математичних обчисленнях між вектором напрямку та колайдерами об'єктів, що дозволяє визначити факт наявності перешкод, відстань до них, точку зіткнення, нормаль до поверхні тощо.

В Unity метод Raycast реалізується в рамках класу Physics і зазвичай має сигнатуру Physics.Raycast(), де першим параметром виступає початкова точка променя Vector3 origin, другим – напрямок Vector3 direction, а також можуть додатково передаватися такі параметри, як максимальна довжина променя float maxDistance, фільтр шарів LayerMask, та інші ознаки обробки зіткнень. У разі виявлення перетину променя з колайдером, метод повертає true, а додаткові дані (зокрема точку перетину та інформацію про об’єкт) можна зчитати через структуру RaycastHit.

В проєкті Raycast активно використовується для реалізації бойової системи, визначення перешкод та для отримання позиції курсора відносно персонажа за допомогою методу Camera.main.ScreenPointToRay(), що випускає промінь з камери в напрямку позиції курсора.

Перевагою для використання Raycast є то, що він не вимагає фізичної симуляції чи зміни положення тіл у просторі. Це забезпечує його ефективність у порівнянні з обробкою зіткнень колайдерів – виклики Physics.Raycast витрачають менше обчислювальних ресурсів, особливо при правильному використанні LayerMask для обмеження пошуку тільки релевантними шарами.

3.2 Архітектура проєкту

3.2.1 Локатор Сервісів

Однією з важливих проблем, яку потрібно вирішити в проєкті є взаємодія класів між собою. Наприклад, при старті гри потрібно завантажити карту, ввімкнути контроль персонажа, запустити генерацію ворогів й інші системи. Очевидним рішенням є переміщення всієї логіки ініціалізації в один клас, але це вимагатиме передачі багатьох залежностей через конструктор, що призведе до його розростання та ускладнення підтримки коду. Альтернативою може стати Singleton, але у цього підходу є свої недоліки. Щоб уникнути громіздких конструкторів і не перевантажувати код Singleton класами, можна скористатися Service Locator.

Singleton – це шаблон проєктування. Він гарантує, що створений за допомогою нього клас матиме тільки один екземпляр, і забезпечує глобальну точку доступу до цього екземпляра.

Service Locator – це патерн проєктування, що використовується в розробці програмного забезпечення для інкапсуляції процесів, пов'язаних з отриманням будь-якого сервісу з сильним рівнем абстракції. Його відмінність від звичайного Singleton полягає в тому, що Service Locator надає гнучкість управління сервісами, їхньою реєстрацією та видаленням. Цей шаблон надає централізований механізм, що використовує локатор сервісів, який за запитом повертає інформацію необхідну для виконання певного завдання. В деяких випадках використання Service Locator фактично є антипатерном.

В порівнянні з Singleton патерн Service Locator є кращим вибором і має наступні переваги:

- Працює через інтерфейси, що дозволяє легко змінювати конфігурації;
- Дозволяє відключати непотрібні сервіси в різних сценах;
- При використанні Singleton можуть з’являтися мертві посилання;

Головним конкурентом Service Locator є Dependency Injection. Впровадження залежностей — це шаблон проєктування, в якому залежності або сервіси впроваджуються, або передаються по посиланню в залежний об'єкт і стають частиною клієнтського стану. Шаблон відокремлює створення залежностей клієнта від власної логіки клієнта, що дозволяє компонентам бути слабко зв'язаними й дотримуватися принципів інверсії залежностей і єдиного обов'язку. Це суперечить антипатерном service locator, який дозволяє клієнтам знати про систему, що використовується для пошуку залежностей.

Хоча патерни Service Locator та Dependency Injection вирішують схожі завдання, але мають свої відмінності. До переваг Service Locator можна віднести наступне:

- Service Locator простіший у реалізації;
- Зручніший для швидкого розширення проєкту;

Dependency Injection має такі переваги:

- Dependency Injection наочніший і зручніший для тестування – усі залежності видно під час ініціалізації;
- Більш безпечний – будь-який клас в Service Locator може отримати доступ до будь-якого сервісу, що може ускладнювати контроль залежностей;
- Прозоріший – в Service Locator приховує залежності класу, викликаючи помилки під час виконання, а не помилки часу компіляції, коли за відсутності необхідних залежностей компілятор інформує про помилку;

Отже, Service Locator має низку переваг порівняно з Singleton, але також має свої недоліки й вважається антипатерном і загалом використовується в невеликих проєктах де складно або впровадити Dependency Injection або де він надлишковий, як гарна альтернатива Singleton.

В проєкті патерн Service Locator використовується для ініціалізації та доступу до сервісів в одному місці у будь-якій частині коду без прямого зв’язку між об'єктами.

Service Locator – це статичний об'єкт, що зберігає посилання на сервіси та надає їх іншим класам за запитом. У нашому випадку це Singleton, що містить статичне посилання на поточний екземпляр класу ServiceLocator і колекцію Dictionary, де ключ – це назва сервісу з типом string, а значення це самі сервіси, які збігаються за допомогою інтерфейсу IService.

Реєстрація сервісу, його видалення й отримання посилання на сервіс виконується за схожим принципом. Наприклад для реєстрації використовується наступний метод:

```
public T Get<T>() where T : IService
```

Це генеричний метод, який приймає параметр service типу T. “where T : IService“ – обмеження, яке вимагає, щоб тип T реалізовував інтерфейс IService. Це означає, що можна реєструвати тільки сервіси, які відповідають певному інтерфейсу. Далі отримуємо назву типу класу сервісу, яку використовуємо як ключ для елементу колекції сервісів і якщо сервіс цього типу ще не зареєстрований у словнику _services, то реєструємо його, а якщо ні то припиняємо виконання методу та сповіщаємо про помилку:

```
string key = typeof(T).Name;
if (!_services.ContainsKey(key))
{
    Debug.LogError($"{key} not registered with {GetType().Name}");
    throw new InvalidOperationException();
}

return (T)_services[key];
```

Щоб використати Service Locator в рушії Unity був написаний клас ServiceLocatorLoader_Gameplay, який є нащадком MonoBehaviour, щоб клас можливо було використовувати на сцені. Кожний сервіс успадковується від пустого інтерфейсу IService. ServiceLocatorLoader_Gameplay Зберігає посилання на кожен сервіс й займається їх реєстрацією в Service Locator, ініціалізацією сервіс локатора, ініціалізацією кожного сервісу й очищенням ServiceLocator від тих сервісів які наслідуються від інтерфейсу IDisposable і які повинні бути видалені при зміні поточного екземпляра класу ServiceLocator.

Для того, щоб звернутися до потрібного класу сервісу треба отримати посилання на нього в локаторі сервісів. Для цього треба звернутися до статичного класу ServiceLocator, і через поточний екземпляр класу за допомогою генеричного методу Get(), ввівши в параметр тип класу, який успадковується від інтерфейсу IService.

3.2.2 Шина подій

Шина подій – це механізм, який дозволяє різним компонентам взаємодіяти між собою, не маючи прямої залежності. Компонент, що використовує шину подій може повідомити про події, а інші компоненти, що мають доступ до шини подій можуть підписатися на ці події й реагувати на них.

Шина подій у проєкті являє собою глобальний статичний об’єкт, що зберігає у своїх полях події, які можуть бути викликані. Компоненти, які повинні відстежувати певну подію виконують підписку на неї за допомогою оператора +=, що додає посилання на метод в об’єкті до списку викликів делегата події. Для уникнення витоку пам’яті та проблем виконання логіки також необхідно відписувати об’єкт від події, наприклад, коли від знищується за допомогою оператора -=.

Коли потрібно повідомити, що подія сталася, викликається делегат. Виклик може виглядати наступним чином:

```
EventBus.OnPlayerJoined?.Invoke(this);
```

Спочатку відбувається звернення до шини подій у EventBus.OnPlayerJoined, потім за допомогою умовного оператора ? відбувається перевірка наявності підписників на подію. Метод Invoke, якому передаються необхідні параметри по черзі викликає всі методи, які підписані на цю подію, передаючи їм ці параметри.

3.3 Ігрові Системи

3.3.1 Система керування персонажем

Для керування персонажем в грі необхідна система обробки вводу та система, яка отримує ці вхідні дані та керує персонажем. В рушії Unity вже реалізована Input System і вона підходить під потреби проєкту. Ця система наступні основні переваги:

- Підтримує декілька пристроїв. система дозволяє обробляти ввід з клавіатур, мишок, геймпадів, тачскрінів, VR-контролерів тощо;
- Дозволяє створювати різні схеми керування, між якими можна динамічно перемикатися;
- Гравці можуть підключати чи відключати контролери під час гри.

Щоб використовувати Input System необхідно створити в проєкті файл Input Actions та налаштувати його під потреби проєкту в редакторі, в ньому задаються необхідні для гри дії (наприклад, “Move”, “Jump”) та сигнали вводу, які викликають ці дії. Об’єктам, які повинні використовувати керування треба додати компонент PlayerInput.

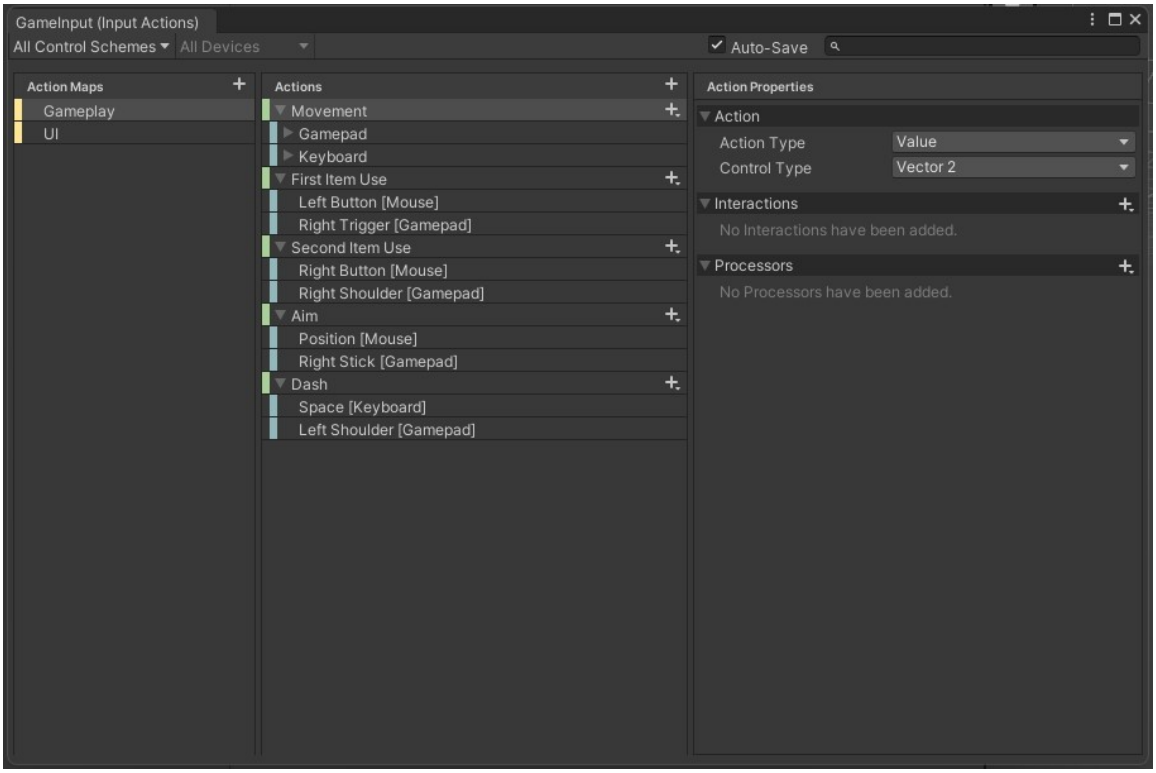

Рис. 3.3 Вікно налаштування Input Actions

На основі цієї системи та сервіс локатора для проєкту був написаний власний InputManager, який оброблює підключення та відключення пристроїв вводу та надає доступ до них іншим системам. InputManager оперує з класом InputsList який є обгорткою списку класів InputRef. InputRef зберігає в кожному елементі дані про пристрій та об’єкт гравця, яким керує цей пристрій.

Клас InputManager підписаний на події OnDeviceConnect та OnDeviceDisconnect й коли вони відбуваються, то додає пристрої до списку InputsList.

Для контролю гравця в Input Actions було створено різні дії, компонент PlayerInput оброблює проініційований для об'єкта гравця ввідні дані та підписує на їх виклик необхідні для логіки персонажу методи. Наступні методи підписуються в класі управління гравцем TopDownController: OnMove() – приймає двомірний вектор напряму руху, отриманий вектор зберігається в полі _movementInput; OnAim() – приймає двомірний вектор напряму погляду гравця, вектор зберігається в полі _aimInput; OnDash() – приймає бульове значення натиску на кнопку ривка, при виклику методу запускається співпрограма Dash().

Інформація про напрям руху оброблюється кожен кадр викликом CalculateMovementVector() в методі Update(). Також в цьому методі зчитується поточна схема керування, оновлюються параметри аніматора (MovementSpeed, Horizontal, Vertical) та залежно від контролю обирається метод повороту: мишкою – TurnToMousePosition(), геймпадом – TurnToStickDirection().

Метод CalculateMovementVector() перетворює вхідні дані з клавіатури або геймпада у світові координати камери й нормалізує їх. Цей метод існує для того, щоб залежно від напрямку повороту камери напрям руху персонажа був інтуїтивним, тобто, коли гравець натискає "вперед", а персонаж рухається вперед відносно камери, а не лише, наприклад, по глобальній осі X. Покроково цей метод працює наступним чином:

```
float h = _movementInput.x;
float v = _movementInput.y;


Vector3 cameraR = _cameraTransform.right;
Vector3 cameraF = _cameraTransform.forward;

cameraR.y = 0;
cameraF.y = 0;
```

Отримується вертикальна та горизонтальна частина двовимірному вектору, який надходить з Input System. Отримується вектор напрямку вперед та праворуч від камери, величина по осі Y ігнорується.

```
Vector3 movementVector = cameraF.normalized * v +
cameraR.normalized * h;
movementVector = Vector3.ClampMagnitude(movementVector, 1);
return movementVector;
```

Вектор руху складається з векторів напрямку вперед та праворуч від камери, помножених на відповідні значення v і h. Метод ClampMagnitude() обмежує довжину вектора до 1, щоби уникнути прискорення при русі по діагоналі. Отриманий в глобальних координатах вектор movementVector повертається в місце виклику методу. Оброблений вектор напряму руху помножується на задану налаштовану швидкість та застосовується на об’єкті через його компонент Rigidbody кожен виклик методу FixedUpdate().

Схожим чином працює вирахування напрямку погляду гравця при керуванні с геймпада. В методі TurnToStickDirection() дані з вектору напряму гравця помножуються на вектори напрямку вперед та праворуч від камери. Напрям погляду для керування мишею використовується метод Physics.Raycast, який створює промінь від позиції камери в напряму позиції курсора на екрані.

3.3.2 Навігація ворогів

При наявній у ворогів логіці поведінки все ще треба, щоб вороги мали можливість обходити перешкоди й отримували інформацію про те як рухатись по рівню. Розв’язати цю задачу можливо, якщо розбити рівень на точки (ноди) та з’єднати сусідні, щоб отримати з них сітку. Так отримана сітка буде являти собою граф, а задача пошуку шляху для ворогів буде розв’язана за допомогою задачі про найкоротший шлях у рамках теорії графів, яка розглядає визначення найкоротшого шляху між двома точками у великій мережі.

Існують різні алгоритми пошуку шляху, кожен з яких є гарним вибором для конкретних випадків. Будуть розглянуті деякі алгоритми і їх переваги та недоліки у контексті проєкту.

Пошук у ширину:

Пошук у ширину (Breadth-First Search) – це базовий алгоритм обходу або пошуку в графі. Його ключовою ідеєю є відстежування межі між пройденими та непройденими вершинами й розширення вздовж всієї ширини. Алгоритм циклічно досліджує вершини за рівнями, починаючи з початкової й проходить через всі доступні вершини. Пошук у ширину використовується не тільки для пошуку шляху, а й для інших задач, наприклад, для карт відстаней або виявлення циклів у графах.

Алгоритм починає з заданої початкової вершини й додає всі сусідні вершини поточної вершини до черги. Потім алгоритм послідовно відвідує вершини, витягуючи їх з черги, і додає до черги їхніх невідвіданих сусідів. Процес повторюється, поки черга не стане порожньою.

Алгоритм Дейкстри:

Для того, щоб було можливо враховувати вартість для кожної вершини потрібно використовувати складніший алгоритм. Алгоритм Дейкстри — це класичний алгоритм пошуку найкоротшого шляху у зважених графах без від’ємних ваг. Його розробив Едсгер Дейкстра у 1956 році. Порівняно з пошуком в ширину алгоритм Дейкстри відстежує витрати на переміщення, зберігаючи загальну вартість переміщення від початкового місця.

Спочатку алгоритм встановлює нескінченну відстань для кожної вершини, крім стартової вершини. Потім циклічно вибирає вершини з найменшою відстанню серед неперевірених і для кожного сусіда обраної вершини перевіряється, чи буде шлях через неї коротшим. Якщо шлях найкоротший – відстань до сусіда оновлюється, а відвідана вершина більше не обробляється. Цей цикл повторюється поки не будуть відвідані всі вершини.

Відмінність між пошуком у ширину та алгоритмом Дейкстри полягає в тому, що алгоритм Дейкстри може працювати з графами чиї ребра мають різну вартість і тому зазвичай у реалізаціях цього алгоритму замість звичайної черги використовується пріоритетна, що змінює спосіб розширення межі.

Жадібний пошук за найкращою вартістю:

Завдяки пошуку в ширину та алгоритму Дейкстри кордон розширюється в усіх напрямках. Така особливість є корисною, якщо необхідно знайти шлях до декількох місць. Однак поширеним випадком є пошук шляху лише до одного місця. Жадібний пошук за найкращою вартістю – це алгоритм пошуку, який розширює пошук у напрямку найближчому до цілі на основі евристичної функції.

Приблизну відстань до цілі можливо вирахувати багатьма способами, наприклад, евристичною функцією для прямокутної мережі вершин, наприклад, може бути манхеттенський метод. За ним відстань між двома точками дорівнює сумі модулів різниць їх координат. Для побудови шляху вибираються найближчі до цілі вузли.

Алгоритм пошуку A*:

Алгоритм A* – це евристичний алгоритм пошуку шляху, який знаходить найкоротший шлях між початковою і цільовою точкою у графі або на карті. Він поєднує в собі переваги алгоритму Дейкстри й жадібного пошуку за найкращою вартістю. Алгоритм Дейкстри добре працює для пошуку найкоротшого шляху, але він витрачає час на дослідження неперспективних напрямків. Жадібний пошук за принципом «найкращий перший шлях» досліджує перспективні напрямки, але може не знайти найкоротший шлях. Алгоритм A* використовує як фактичну відстань від початку, так і розрахункову відстань до цілі.

Алгоритм A* використовує функцію вартості для кожного вузла:

$$f(n) = g(n) + h(n)$$

– g(n) : фактична вартість шляху від початкової точки до поточної вершини n;

– h(n) : евристична оцінка вартості найкоротшого шляху від n до цільової вершини;

– f(n) : загальна оцінка, яка допомагає вибирати, куди йти далі.

Доки евристика h(n) не перевищує реальну відстань до цілі, A* знаходить оптимальний шлях, як це робить алгоритм Дейкстри. A* використовує евристику для зміни порядку вузлів, щоб збільшити ймовірність того, що цільовий вузол буде знайдено раніше.

Вибір оптимального алгоритму:

Алгоритм A* є гарним вибором якщо треба побудувати один шлях для однієї точки. Пошук в ширину можна використовувати для побудови шляху для кожної точки і зробити це швидше ніж за допомогою A*, тому був вибраний саме пошук в ширину. Для кожної вершини вираховується напрямок, який буде вести до наступної найближчої до цілі вершини.

Для проєкту логіка пошуку шляху була реалізована в об’єкті-сервісі PathfindingManager. Цей об’єкт створює сітку вузлів, зв’язує сусідні вузли генерує навігаційну мапу для кожного гравця та зберігає ці дані для подальшого використання іншими системами. Об’єкт має наступні поля: посилання на список точок на сцені рівня; список вузлів, з посиланнями на сусідні вузли; колекцію Dictionary, де клас гравця є ключем, а згенеровані для цього гравця навігаційні дані є даними.

Ініціалізація класу відбувається у методі Init(). Цей метод запускає наступні процеси: викликає метод CreateNodes(), який генерує сітку вузлів на рівні; викликає метод співпрограму UpdateNavigation(), який є нескінченним циклом який оновлює мапу навігації для гравців с заданою у полі _navigationUpdateRate частотою, підписується на події створення та знищення об’єкта гравця.

В методах OnPlayerJoined() та OnPlayerDied(), які викликаються відповідними подіями з шини подій, за допомогою сервіс локатора отримується список створених на рівні гравців, цей список синхронізується зі словником PlayerPaths.

Генерація навігаційної мапи відбувається в методі GenerateWightNavigation (Node start), який використовує алгоритм пошуку в ширину. Метод генерує найкоротший шлях від кожного вузлу у графі до вузла вказаного в аргументі методу. Результатом обчислень є словник, який інформацію про відстань і з якого вузла прийшли до нього. Код методу виглядає наступним чином:

- Створюється словник result, де кожному вузлу присвоюється PathData з максимально можливою початковою відстанню int.MaxValue;

```
var result = new Dictionary<Node, PathData>();
foreach (var node in nodeList)
    result[node] = new PathData(int.MaxValue);
```

- Кожен вузол зберігає дистанцію та посилання на сусідній вузол, який буде використовуватись в будуванні шляху. Початковий вузол, який заданий в параметрах методу, ініціалізується з нульовою відстанню та посиланням на себе, бо цей вузол і є ціллю пошуку.

```
result[start].Distance = 0;
result[start].CameFrom = start;
```

- Створюється черга, в неї додається початковий вузол та починає виконуватись цикл, доки всі елементи в черзі не будуть обійдені;

```
var queue = new Queue<Node>();
queue.Enqueue(start);

while (queue.Count > 0)
{
    ...
}
```

- В кожній ітерації циклу з черги виймається вузол і перебирається кожен його сусід. Якщо сусід ще не відвіданий, то: встановлюється відстань до поточного вузла; зберігається посилання на минулий вузол; сусід додається до черги. Результат зберігається в словник result та повертається методом;

```
while (queue.Count > 0)
{
    var current = queue.Dequeue();
    var currentData = result[current];

    foreach (var neighbor in current.connections)
    {
        var neighborData = result[neighbor];
        if (neighborData.Distance == int.MaxValue)
        {
            neighborData.Distance = currentData.Distance + 1;
            neighborData.CameFrom = current;
            queue.Enqueue(neighbor);
        }
    }
}

return result;
```

Методи FindNearestNode(Vector3 pos) та FindFurthestNode(Vector3 pos) виконують цикл, перевіряючи весь граф вузлів та повертають найближчий та найдальший вузол у мережі відповідно.

Інші системи в проєкті, наприклад, штучний інтелект ворога, може використовувати навігаційну мапу. Для цього треба звернутися до методу FindNearestNode(Vector3 pos) щоб отримати найближчий вузол до необхідного об’єкта, з посилання на наступний вузол на шляху, яке зберігається в класі вузла, отримати позицію наступного вузла та вирахувати нормалізований вектор напрямку руху від теперішньої позиції об’єкта до наступного вузла за формулою $\vec{d} = \frac{1}{\|\vec{v}\|} \cdot \vec{v}$, де $\vec{v}$ – це різниця між позицією наступного вузла та поточною позицією об’єкта, а $\vec{d}$ – це отриманий нормалізований напрям руху до наступного вузла.

3.3.3 Шаблон проєктування State Machine

State Machine – це поведінковий патерн проєктування, який дозволяє об'єкту змінювати свою поведінку залежно від внутрішнього стану. Зовні створюється враження, що змінився сам клас об'єкта. Цей патерн часто використовується для заміни громіздких конструкцій switch-case та if-else.

Патерн State Machine оснований на концепції машини станів, також відомої як стейт-машина або скінченний автомат. Скінчений автомат – це модель, що використовується для опису зміни стану об’єкта залежно від поточного стану та інформації отриманої ззовні. Поняття скінченного автомата було запропоновано як математичну модель дискретних приладів, оскільки будь-який такий прилад може мати тільки скінченну кількість станів.

Патерн State Machine ділить поведінку об’єкта на стани. Стан інкапсулює поведінку об'єкта в певний момент часу. Об’єкт може знаходитися в одному з кількох станів, які увесь час змінюють один одного. Кожен стан визначає власну поведінку об'єкта. Набір цих станів, а також переходів між ними, визначений наперед та скінчений.

Цей патерн широко використовується в розробці ігор, бо він зручний у ситуаціях, де об'єкт змінює поведінку залежно від стану, наприклад:

- Поведінка ворогів залежно від обставин (патрулювання, переслідування, атака);
- Управління користувацьким інтерфейсом, наприклад, в ситуаціях коли є елементи, які є несумісними один з одним;
- Покрокова механіка. Наприклад, зміна фаз бою, черговість ходів;
- Контролер анімацій в рушії Unity реалізований через цей патерн;
- Системи діалогів, де стан діалогу змінюється залежно від вибору гравця;

Замість громіздкої логіки з великою кількістю умов, програма представляється як набір станів із чіткими переходами між ними. Такий підхід робить код більш читабельним і підтримуваним.

Реалізація

Патерн Клас State Machine в проєкті використовується для реалізації різної поведінки для ворогів. Для цього була написана логіка універсального скінченного автомата, яку можливо використовувати у подальшій розробці й для інших систем. Абстрактна машина станів складається з класів FSM (finite state machine) та FSMState.

В класі FSM стани зберігаються у колекції Dictionary з назвою _states, де ключем є тип стану, а значеннями є самі стани типу FSMState. Також клас має посилання на поточний стан в StateCurrent. Клас FSM реалізовує методи AddState, SetState, Update та FixedUpdate. Метод AddState(FSMState state) додає стан до колекції _states. Метод SetState приймає в параметри узагальнений клас T, який повинен буди успадкований від класу, перевіряє, чи відрізняється він від поточного, викликає метод Exit() для попереднього стану, змінює поточний стан і викликає Enter() для нового.. Методи Update та FixedUpdate будуть використовуватися для передачі викликів від конкретної реалізації машини станів, яка, наприклад, може бути успадкована від MonoBehaviour.

За допомогою патерну State Machine у проєкті було реалізовано логіку ворогів. Для ворогів був створений клас FSMEnemyAI, який є компонентом об’єкта ворога. Він створює та керує екземпляром класу FSM.

Всі стани які може мати об’єкт додаються у словник _states за допомогою методу AddState(FSMState state) до класу FSM під час ініціалізації класу FSMEnemyAI. В конструктори станів передаються всі необхідні для їх роботи дані: посилання на клас FSM, посилання на базовий компонент ворога enemyRef, цільова швидкість переміщення, система навігації, список гравців та початковий навігаційний вузол. Після цього виконується Метод SetState <T>(), який приймає початковий стан ворога й ініціалізація завершується.

В методах Update() та FixedUpdate() класу FSMEnemyAI передаються необхідні виклики до класу машини станів. Логіка зміни поведінки об’єкта задана усередині класів станів та виконується автономно. Подібний спосіб використання патерну можливо реалізовувати й в інших станах.

3.3.4 Логіка ворогів

Основними складниками поведінки ворогів є патерн скінченний автомат, навігація ворогів за допомогою алгоритмів пошуку шляху та поведінковий алгоритм "Boids". Модель "Boids" розроблена Крейгом Рейнольдсом у 1986 році для симуляції колективної поведінки тварин, таких як зграї птахів або косяки риб. Модель Boids стала важливою у галузі комп'ютерної анімації та штучного життя. Кожен об’єкт в моделі самостійно дотримується трьох правил:

- Уникнення (Separation): уникати надмірного зближення з сусідами.
- Вирівнювання (Alignment): пристосовувати свій напрямок руху до середнього напрямку сусідів.
- Згуртованість (Cohesion): рухатися до середнього положення сусідів.

Ці правила застосовуються лише до сусідів у певному радіусі та куті огляду, що імітує обмежене сприйняття. Модель демонструє емерджентну поведінку, де складні глобальні патерни виникають із взаємодії простих локальних правил.

Стани наслідуються від базового абстрактного класу EnemyState, який наслідує базовий компонент патерну FSMState та забезпечує загальну логіку і залежності для всіх похідних станів. До таких залежностей належать посилання на компоненти ворога, поточний вузол навігаційної сітки _currentNode, менеджер пошуку шляху PathfindingManager та список гравців для переслідування _playerList. Також в базовому класі є метод TargetClosestPlayer(), що обирає найближчого гравця, спираючись на

попередньо розраховані шляхи до кожного з них. Цей метод може бути використаний в класах що його наслідують.

Логіка поведінки ворогів ділиться на три стани. IdleEnemyState – це початковий стан, який ініціюється в машині станів. Стан описує пасивну поведінку ворога, сутність очікує появи персонажа гравця і викликає стан переслідування при його появі. В початку виклику цього стану в методі Enter() клас звертається до аніматора ворога та викликає анімацію стану покою.

У ChaseEnemyState реалізовано поведінку переслідування, включаючи алгоритмічну підтримку руху в рою за допомогою моделі Boids, через розрахунок сил вирівнювання, відштовхування та згуртованості. У методі FixedUpdate() містить виклик пошуку найближчого вузла, обчислення напрямку руху до цілі _moveDirection, перевірки досяжності гравця, а також перемикання на стан AttackEnemyState при достатньому наближенні.

Для реалізації ройової поведінки було створено наступні методи:

- GetNeighbours(): за допомогою механізму Physics.OverlapSphere() повертає колекцію найближчих колайдерів сусідів;
- CalculateSeparationForce() описує розділення: розраховує силу, яка відштовхує поточний об’єкт від кожного сусіда. Метод вираховує відстань та напрямок до сусідів й додає отриману силу відштовхування до вектора _separationForce для подальших розрахунків. Чим ближчий сусід – тим сильніше відштовхування;

```
var dir = neighbour.transform.position - _rigidbody.position;
var distance = dir.magnitude;
var away = -dir.normalized;

if (distance > 0)
{
    _separationForce += (away / distance) * _separationWeight;
}
```

- ApplyAllignment() описує вирівнювання: метод отримує напрямок руху сусідів, збирає їх в один вектор та нормалізує його. Отриманий вектор

помножений на налаштовану величину додається до _separationForce для подальших розрахунків;

- ApplyCohesion() описує згуртованість: метод обчислює середнє положення сусідів, яке є центром мас і спрямовує об’єкт до цього центру. Напрямок помножений на налаштовану величину додається до _separationForce.

```
averagePosition /= neighbours.Length;

Vector3 cohesionDir = (averagePosition -
_rigidbody.position).normalized;
_separationForce += cohesionDir * _cohesionWeight;
```

Після виклику обчислень сили для коректування позиції об’єкта відносно сусідніх у методі FixedUpdate() відбувається вирахування напрямку найшвидшого шляху до гравця, який був повернутий методом TargetClosestPlayer().

За допомогою менеджера пошуку шляху отримується найближчий до поточної позиції об’єкта вузол. Якщо є цільовий шлях і кількість вузлів до гравця велика, вираховується напрямок до наступного вузла на шляху. Якщо гравець достатньо близько, то вираховується напрям до нього. Якщо ворог знаходиться на тому ж вузлі що й персонаж гравця або відстань до нього мала, то стан перемикається на стан атаки AttackEnemyState.

```
if (_targetPathData.Distance > 2)
{
    _moveDirection = (new Vector3(
        _targetPathData.CameFrom.transform.position.x,
        _rigidbody.position.y,
        _targetPathData.CameFrom.transform.position.z
    ) - _rigidbody.position).normalized;
}
else if (_targetPathData.Distance > 0)
{
    _moveDirection = (new Vector3(
        _targetPlayer.transform.position.x,
        _rigidbody.position.y,
        _targetPlayer.transform.position.z
    ) - _rigidbody.position).normalized;
}
else if (_targetPathData.Distance == 0)
```

```
{
    _fsm.SetState<AttackEnemyState>();
}

if (Vector2.Distance(new Vector2(_rigidbody.position.x,
_rigidbody.position.z),
        new Vector2(_targetPlayer.transform.position.x,
_targetPlayer.transform.position.z)) < 1.5f)
{
    _fsm.SetState<AttackEnemyState>();
}
```

Далі зануляється вертикальний складник зграйової сили та обчислюється комбінований напрям руху як сума руху до цілі та зграйового вектора. Вектор застосовується у компоненті _rigidbody.velocity, плавно змінюючи швидкість у напрямку combinedDirection. Також відбувається поворот ворога у напрямку руху.

```
_separationForce = new Vector3(_separationForce.x, 0f,
_separationForce.z);
var combinedDirection = _moveDirection * _speed +
_separationForce;

_rigidbody.velocity = Vector3.Lerp(_rigidbody.velocity,
combinedDirection, 0.15f);
_rigidbody.MoveRotation(Quaternion.Lerp(Quaternion.LookRotation(_m
oveDirection), _rigidbody.rotation, 0.9f));
```

AttackEnemyState реалізовує стан атаки гравця ворогом. При вході у цей стан у методі Enter() запускається анімація атаки ворога, створюється таймер для синхронізації анімації та атаки ворога й таймер, який відлічує час, коли ворог може повернутись до іншого стану.

В методі Update() ворог отримує за допомогою TargetClosestPlayer() найближчого гравця, слідкує за його позицією, обертається у його бік та атакує персонажа якщо дистанція до нього мала й таймер _startAttackTimer сплинув. У іншому випадку, коли дистанція до ворога велика, то стан перемикається на стан переслідування ChaseEnemyState. Далі, коли сплине таймер _endAttackTimer стан перемикається на нейтральний стан очікування IdleEnemyState.

3.3.5 Система здоров’я

Функцію компонента для системи здоров’я для різних ігрових сутностей виконує клас Health. Його призначення полягає у моделюванні життєвого циклу об’єкта, здатного зазнавати пошкоджень або зцілення. Також клас сповіщає про взаємодію зі здоров’ям іншим компонентам.

Властивість CurrentHealth зберігає поточний стан здоров’я, а IsAlive повертає логічне значення, що вказує на життєздатність об’єкта. Максимальне значення здоров’я задається через серіалізоване поле _maxHealth, що забезпечує його конфігурацію через редактор Unity. Система також включає механізм тимчасової невразливості після отримання шкоди, реалізований через об’єкт типу Timer та параметр _immortalDamageTime, який визначає тривалість цієї невразливості. Перевірка завершення невразливості виконується в методі Update(), що регулярно оновлюється в циклі гри.

Щоб обробляти шкоду, що надходить до компонента клас наслідує інтерфейс IHitTarget та реалізує метод цього інтерфейсу ProcessHit(), який приймає параметр ref HitData, що інкапсулює інформацію про пошкодження чи зцілення, їх силу та наслідки. У межах методу викликається ApplyHit, який змінює значення здоров’я відповідно до типу дії, а також запускає таймер невразливості у випадку пошкодження. Якщо після дії здоров’я падає до нуля або нижче, викликається метод OnDie, який є віртуальним і повинен бути перевизначений у похідних класах для реалізації конкретної логіки смерті. Події HitTaken та FatalHitTaken сигналізують про отримання будь-якого чи фатального удару відповідно, що дозволяє іншим системам гри реагувати на ці події, наприклад, запускати візуальні ефекти або відтворювати звуки.

Отриманий клас Health – це модульна система для керування здоров’ям об’єктів у грі. Його структура відокремлює логіку пошкодження від інших елементів гри, та сприяє повторному використанню коду та полегшує його підтримку.

3.3.6 Шаблон проєктування Декоратор

Декоратор – це структурний патерн проєктування, він є класичним прикладом об’єктно-орієнтованого підходу до розширення функціональності об’єктів без модифікації їхнього базового класу. Патерн дає змогу динамічно додавати об’єктам нову функціональність, загортаючи їх у корисні «обгортки». Обгортання класів – це агрегація, тобто, поміщення одного класу усередину іншого з метою розширення та зміни його поведінки. Патерн розділяє логіку на декілька об’єктів та має таку структуру:

- Component – базовий інтерфейс або абстрактний клас;
- ConcreteComponent – конкретна реалізація інтерфейсу;
- Decorator – абстрактний клас, який реалізує інтерфейс Component і містить посилання на інший об’єкт Component;
- ConcreteDecorator – конкретна реалізація декоратора, яка додає нову поведінку.

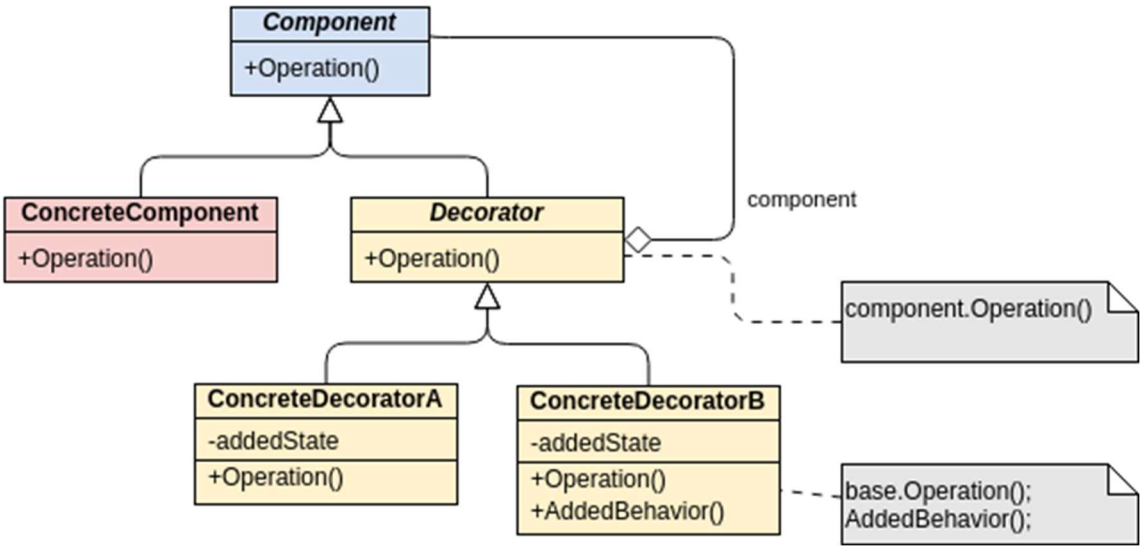

Рис. 3.4 UML схема патерну декоратор

Перевагами цього патерну є те, що він дозволяє змінювати поведінку об’єктів без успадкування, підтримує принцип відкритості/закритості та підвищує модульність коду. До недоліків можна віднести складність відстеження кількості декораторів та можливість створення великої кількості дрібних класів.

В ігрових проєктах патерн декоратор використовується для ряду задач, вирішення яких практично можливо лише за допомогою нього. Наприклад, у багатьох іграх з великою кількістю статусів, ефектів, модифікаторів шкоди та інших параметрів саме цей шаблон використовується для реалізації системи апгрейдів здібностей.

Окрім ігрової індустрії, декоратори застосовуються також у системному програмуванні, зокрема для модифікації поведінки драйверів або інших програмних компонентів. Наприклад, у розробці мережевих застосунків, якщо існує необхідність у тимчасовому логуванні результатів розрахунків, розробник може обгорнути модуль обчислень декоратором із логуванням, не змінюючи безпосередньо вихідний код. Коли потреба у відстежуванні зникає, обгортка легко знімається, а основна функціональність залишається недоторканою.

У проєктному коді декоратор застосовується до системи апгрейдів, зокрема для модульного додавання характеристик у процесі гри, наприклад, зміни шкоди від пострілу, додаткові умови використання предметів тощо.

Інтерфейс IDamage в структурі патерну реалізує об’єкт Component. IDamage контролює шкоду зброї та зв’язані з нею механіки. Декоратор може перевизначити функціонал методів інтерфейсу.В IDamage описано те, що клас який його реалізує повинен мати наступні методи:

```
void Init(IDamage baseDamage);
int GetDamage();
EDamageType GetDamageType();
void TakeDamage(IDamageReceiver damageReceiver);
object Clone();
```

- Init(IDamage baseDamage) ініціалізує об’єкт;
- GetDamage() повертає цифрове значення шкоди типу int;

- GetDamageType() повертає тип шкоди типу enum EDamageType;
- TakeDamage(IDamageReceiver damageReceiver) – метод, виклик якого передає об’єкту що використовує систему покращень оброблені дані про шкоду;
- Clone() використовується, щоб отримати об’єкт компонента знаслідують розширення функціональності усіма вкладеними декораторами.

Клас Damage – це ConcreteComponent, тобто, базовий клас, що реалізує логіку завдання звичайної шкоди. Він реалізовує IDamage, але не містить додаткових ефектів або бонусів і потрібен для початкової ініціалізації.

Абстрактний клас BaseDamageUpgrade є компонентом Decorator. Він також реалізовує інтерфейс IDamage. Цей клас делегує основні виклики методів базовому компоненту, які потім можуть перевизначатись в нащадках.

Одними із написаних компонентів структури декоратора ConcreteDecorator є класи AddDamageUpgrade та AdditionalDamageUpgrade, які наслідують реалізують розширення функціональності:

- AddDamageUpgrade змінює результат методу GetDamage, додаючи до значення базової шкоди певний фіксований бонус;
- AdditionalDamageUpgrade доповнює базову логіку заподіяння шкоди, завдаючи додаткової шкоди з іншим типом після виконання стандартного виклику TakeDamage.

Ці класи реалізують обгорткову структуру, де кожен наступний декоратор може розширити або змінити поведінку попереднього, зберігаючи при цьому уніфікований інтерфейс IDamage.

3.4 Висновки до розділу

У даному розділі було описано деталі реалізації прототипу ігрового застосунку із застосуванням рушія Unity та мови програмування C#. Було

використано вбудований функціонал рушія, зокрема системи анімацій, скриптовані об'єкти, користувацький інтерфейс, механізми Raycast і таймери.

Для зручності можливого майбутнього масштабування та підтримки було впроваджено шаблони проєктування локатор сервісів та шину подій. Вони дозволили структурувати взаємодію між системами та зменшити зв’язаність компонентів, що спростило обробку подій у проєкті.

Було описано реалізацію наступних ігрових систем: систему керування персонажем, систему навігації ворогів, систему поведінки ворогів, систему здоров’я та систему покращень. Написані ігрові системи створюють цілісну ігрову логіку та забезпечують передбачувану між об'єктами в грі.

Загалом реалізоаний прототип продемонстрував можливість створення якісної архітектури, заснованої на сучасних підходах до розробки, а також підтвердив доцільність використання обраних технологій у контексті розробки ігор.

ВИСНОВКИ

У ході виконання роботи було розроблено прототип гри. Була виконана мета, що полягала в створені гнучкої, масштабованої системи, що побудована за допомогою об’єктно-орієнтованого підходу з використанням ефективних архітектурних патернів.

Було здійснено аналіз ігрових рушіїв для розробки ігор, серед яких Unity Engine був обраний як оптимальне середовище, що є доступною, гнучкою платформою з підтримкою різних платформ та необхідним інтегрованим функціоналом. Рушій виявився зручним інструментом, що задовольнив потреби проєкту.

Під час дослідження було розглянуто методології розробки програмного забезпечення та доречність їх застосування в ігровій індустрії. Також були розглянуті архітектурні підходи, що відіграють важливу роль у структурі проєкту, зокрема принципи SOLID та патерн ECS. Використання ООП та принципів SOLID дозволило досягти високої модульності, слабкої зв’язаності та придатності прототипу до масштабування. Вони сприяли покращенню якості коду, підвищенню ефективності повторного використання компонентів та полегшенню тестування. Був проведений аналіз успішних ігрових проєктів у цільовому жанрі, що дозволило визначити ключові фактори залучення гравців. Ці ідеї та механіки частково були запозичені та адаптовані для реалізації в рамках власного проєкту.

Результатом роботи став розширюваний прототип, який може бути основою для повноцінної гри. Прототип реалізував різноманітні шаблони проєктування: State Machine, Service Locator, Event Bus, Decorator та інші. Поведінка ворогів реалізована через шаблон State Machine, з використанням алгоритму пошуку шляху та ройової логіки. Розроблено систему покращень, систему керування персонажем та інші небхідні для коректної роботи ігрової логіки компоненти. Отримані знання й практичні навички дозволили закріпити теоретичні знання з розробки програмного забезпечення, архітектури та управління проєктом.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Explore the Global Games Market in 2023. Newzoo. [Електронний ресурс] - Режим доступу: https://newzoo.com/resources/blog/explore-the-global-games-market-in-2023.
2. Дацко М., Семенів Г. Аналіз моделей життєвого циклу проектів галузі інформаційних технологій // Формування ринкової економіки в Україні. — № 18-2008. — С. 63-69.
3. Unreal Engine – Real-Time 3D Creation Tool. Unreal Engine. [Електронний ресурс] - Режим доступу: https://www.unrealengine.com/en-US/.
4. Unity Real-Time Development Platform. Unity. [Електронний ресурс] - Режим доступу: https://unity.com/.
5. Silverthorne V. What is rapid application development (RAD) ? | Definition from TechTarget. Search Software Quality. [Електронний ресурс] - Режим доступу: https://www.techtarget.com/searchsoftwarequality/definition/rapid-application-development.
6. Webster C. Entity Component Systems Usefulness, Literature Review // Journal of LaTeX Class Files. – 2015. – Vol. 14, No. 8. – P. 1–6.
7. Boehm B. A spiral model of software development and enhancement. – ACM SIGSOFT Software Engineering Notes, Vol. 11, No. 4, 1986. – P. 14–24.
8. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms (3rd ed.). MIT Press, 2009. — ISBN 978-0-262-03384-8.
9. Reynolds C. W. Flocks, Herds, and Schools: A Distributed Behavioral Model. Computer Graphics (SIGGRAPH '87 Proceedings), 1987. — [Електронний ресурс] - Режим доступу: https://www.red3d.com/cwr/boids/
10. Unity Technologies. Input System manual, 2025. — [Електронний ресурс] - Режим доступу: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.14/manual/index.html

11. Shalloway A., Trott J. R. Design Patterns Explained: A New Perspective on Object-Oriented Design. — 2-е видання. Boston, MA: Addison-Wesley Professional, 2004. — 480 с.
12. Seemann M., van Deursen S. Dependency Injection Principles, Practices, and Patterns. Manning Publications, 2019. — 552 p.
13. Госкрофт, Дж. Е., Мотвані, Р., Ульман, Дж. Д. (2001). Вступ до теорії автоматів, мов і обчислень (2-ге вид.). Addison–Wesley. С. 521.

ДОДАТКИ

ПРОГРАМНИЙ КОД

```
public class Timer
{
    private float duration;
    private float startTime;
    private bool isRunning;
    private bool isPaused;
    private float pauseTime;

    public Timer()
    {
        isRunning = false;
    }

    public void CreateFromSeconds(float seconds)
    {
        duration = seconds;
        if (!isRunning)
        {
            startTime = Time.time;
            isRunning = true;
        }
    }

    public void Reset(float newDuration = -1)
    {
        if (newDuration > 0) duration = newDuration;
        startTime = Time.time;
        isRunning = true;
    }

    public bool IsFinished()
    {
        if (Time.time >= startTime + duration)
        {
            isRunning = false;
            return true;
        }
        return false;
    }

    public float GetRemainingTime()
    {
        if (!isRunning) return 0;
        return Mathf.Max(0, duration - (Time.time - startTime));
    }

    public float GetProgress()
    {
        if (!isRunning) return 0;
        return Mathf.Clamp01((Time.time - startTime) / duration);
    }
}
public class ServiceLocator
{
    private ServiceLocator()
    {
```

```
    }

    private readonly Dictionary<string, IService> _services = new Dictionary<string,
IService>();

    public static ServiceLocator Current { get; private set; }

    public static void Initialize()
    {
        Current = new ServiceLocator();
    }

    public T Get<T>() where T : IService
    {
        string key = typeof(T).Name;
        if (!_services.ContainsKey(key))
        {
            Debug.LogError($"{key} not registered with {GetType().Name}");
            throw new InvalidOperationException();
        }

        return (T)_services[key];
    }

    public void Register<T>(T service) where T : IService
    {
        string key = typeof(T).Name;
        if (_services.ContainsKey(key))
        {
            Debug.LogError(
                $"Attempted to register service of type {key} which is already
registered with the {GetType().Name}.");
            return;
        }

        _services.Add(key, service);
    }

    public void Unregister<T>() where T : IService
    {
        string key = typeof(T).Name;
        if (!_services.ContainsKey(key))
        {
            Debug.LogError(
                $"Attempted to unregister service of type {key} which is not
registered with the {GetType().Name}.");
            return;
        }

        _services.Remove(key);
    }
}
public interface IDisposable
{
    public void Dispose();
}
public interface IService
{
}
public class ServiceLocatorLoader_Gameplay : MonoBehaviour
```

```
{
    [SerializeField] private InputManager _inputManager;
    [SerializeField] private PlayerManager _playerManager;
    [SerializeField] private GameplayManager _gameplayManager;
    [SerializeField] private UpgradeManager _upgradeManager;
    [SerializeField] private TimeManager _timeManager;
    [SerializeField] private PathfindingManager _pathfindingManager;
    [SerializeField] private EnemiesManager _enemiesManager;

    private List<IDisposable> _disposables = new List<IDisposable>();

    private void Awake()
    {
        Register();
        Init();
        AddToDisposables();
    }

    private void Init()
    {
        _inputManager.Init();
        _playerManager.Init();
        _gameplayManager.Init();
        _upgradeManager.Init();
        _pathfindingManager.Init();
        _enemiesManager.Init();
    }

    private void Register()
    {
        ServiceLocator.Initialize();

        ServiceLocator.Current.Register(_inputManager);
        ServiceLocator.Current.Register(_playerManager);
        ServiceLocator.Current.Register(_gameplayManager);
        ServiceLocator.Current.Register(_upgradeManager);
        ServiceLocator.Current.Register(_timeManager);
        ServiceLocator.Current.Register(_pathfindingManager);
        ServiceLocator.Current.Register(_enemiesManager);
    }

    private void AddToDisposables()
    {
        _disposables.Add(_inputManager);
        _disposables.Add(_playerManager);
        _disposables.Add(_gameplayManager);
        _disposables.Add(_upgradeManager);
        _disposables.Add(_pathfindingManager);
        _disposables.Add(_enemiesManager);
    }

    private void OnDestroy()
    {
        foreach (var disposable in _disposables)
        {
            disposable.Dispose();
        }
    }
}
public static class EventBus
```

```
{
    public static Action<PlayerRef, HitData> OnPlayerDied;
    public static Action<PlayerRef> OnPlayerJoined;
    public static Action<PlayerRef> OnUpgradeRequest;
    public static Action<EnemyRef, HitData> OnEnemyDied;
}
public class PlayerRef : MonoBehaviour, ITeamMember
{
    [SerializeField] public PlayerHealth playerHealth;
    [SerializeField] public TopDownController TopDownController;
    [SerializeField] public Inventory Inventory;
    [SerializeField] public Animator Animator;
    [SerializeField] public PlayerInput PlayerInput;
    [SerializeField] public Outline Outline;
    [SerializeField] public PlayerIndicators PlayerIndicators;
    [SerializeField] public UpgradeHandler UpgradeHandler;
    [SerializeField] public PlayerStatsController PlayerStatsController;
    public Team Team { get; set; }
    public int Id;
    public Transform CameraTransform;
    public Action<Timer> OnCooldown;
    public Action OnItemSwitch;
    public Action<UpgradeHandler> OnUpgradeReceive;


    public MonoBehaviour[] GetAllScripts()
    {
        MonoBehaviour[] scripts = new MonoBehaviour[] {playerHealth,
TopDownController, Inventory, PlayerInput, Outline };
        return scripts;
    }

    public void Start()
    {
        StartCoroutine(InitializeComponents());
    }

    private void OnDisable()
    {
        OnItemSwitch -= Outline.UpdateOutline;
    }

    private IEnumerator InitializeComponents()
    {
        yield return new WaitUntil(() => UpgradeHandler.Init(this));
        yield return new WaitUntil(() => PlayerStatsController.Init(this));
        TopDownController.Init(PlayerStatsController, CameraTransform);
        yield return new WaitUntil(() => Inventory.Init(this));
        Outline = gameObject.AddComponent<Outline>();
        Outline.OutlineWidth = 1f;
        OnItemSwitch += Outline.UpdateOutline;
        PlayerIndicators.Init(this, CameraTransform);
        EventBus.OnPlayerJoined?.Invoke(this);
    }
}
public class PlayerManager : MonoBehaviour, IService, IDisposable
{
    public List<PlayerRef> SpawnedPlayersList { get; private set; }

    [SerializeField] private Transform _mainCamera;
```

```
    [SerializeField] private GameObject _playerPrefab;
    [SerializeField] private GameObject _spawnEffectPrefab;
    [SerializeField] private List<Transform> _spawnPoints;
    [SerializeField] private CinemachineTargetGroup _targetGroup;
    [SerializeField] private float _respawnDelay = 3f;
    [SerializeField] private int _playerMaxCount = 2;

    private InputManager _inputManager;
    private InputsList _inputsList = new ();
    private InputsList _inGameDevicesList = new ();


    public void Init()
    {
        _inputManager = ServiceLocator.Current.Get<InputManager>();
        _inputManager.OnDeviceDisconnect += OnDeviceDisconnect;
        EventBus.OnPlayerDied += OnPlayerDie;
        SpawnedPlayersList = new List<PlayerRef>();
    }

    public void Dispose()
    {
        EventBus.OnPlayerDied -= OnPlayerDie;
    }

    private void OnDeviceDisconnect(InputDevice device)
    {
        _inGameDevicesList.RemoveInputRef(device);
    }

    private void Update()
    {
        bool isSpawned = false;
        _inputsList = _inputManager.InputsList;
        if (_inputsList.GetInputRefs().Count > 0)
        {
            foreach (var inputRef in _inputsList.GetInputRefs().ToList())
            {
                if (_inGameDevicesList.GetInputRefs().Count >= _playerMaxCount)
                    break;

                if (IsDevicePressed(inputRef.InputDevice) &&
SpawnedPlayersList.Count < _playerMaxCount)
                {
                    if (_inGameDevicesList.TryAddInputRef(inputRef))
                    {
                        _inputsList.GetInputRefs().Remove(inputRef);
                        SpawnCharacter(inputRef.InputDevice);
                        isSpawned = true;
                        break;
                    }
                }
            }
        }

        var inGameDevices = _inGameDevicesList.GetInputRefs();

        if (isSpawned == false && inGameDevices.Count > 0)
        {
            foreach (var inputRef in inGameDevices)
```

```
            {
                if (IsDevicePressed(inputRef.InputDevice) &&
SpawnedPlayersList.Count < _playerMaxCount)
                {
                    SpawnCharacter(inputRef.InputDevice);
                    break;
                }
            }
        }
    }

    private bool IsDevicePressed(InputDevice device)
    {
        if (device is Gamepad || device is Keyboard)
        {
            return device.allControls.OfType<ButtonControl>().Any(button =>
button.wasPressedThisFrame);
        }
        return false;
    }

    private void SpawnCharacter(InputDevice device)
    {
        var inputRef = _inGameDevicesList.GetInputRef(device);

        if (inputRef == null)
            return;

        if (inputRef.SpawnedPlayer != null)
            return;

        var PlayerID = _inGameDevicesList.GetInputRefs().IndexOf(inputRef);

        var spawnPoint = _spawnPoints[0].position;
        if(SpawnedPlayersList.Count <= _spawnPoints.Count)
            spawnPoint = _spawnPoints[PlayerID].position;

        GameObject character = Instantiate(_playerPrefab, spawnPoint,
Quaternion.identity);

        inputRef.SpawnedPlayer = character.GetComponent<PlayerRef>();

        inputRef.SpawnedPlayer.Team = (Team)PlayerID;
        inputRef.SpawnedPlayer.Id = PlayerID;

        inputRef.SpawnedPlayer.CameraTransform = _mainCamera;

        var components = inputRef.SpawnedPlayer.GetAllScripts();
        foreach (var component in components)
        {
            if (component is ISpawnable spawnable)
            {
                spawnable.OnSpawn();
            }
        }

        var playerInput = character.GetComponent<PlayerInput>();

        inputRef.PlayerInput = playerInput;
```

```
        if (playerInput != null && device is Gamepad)
        {
            playerInput.SwitchCurrentControlScheme(device);
        }

        ObjectPoolManager.SpawnObject(_spawnEffectPrefab,
character.transform.position, quaternion.identity);

        OnPlayerSpawn(inputRef.SpawnedPlayer);
    }

    private void OnPlayerDie(PlayerRef playerRef, HitData hitData)
    {
        var playerInput = playerRef.PlayerInput;
        _targetGroup.RemoveMember(playerInput.transform);

        SpawnedPlayersList.Remove(playerRef);
        Destroy(playerRef.gameObject);

        var inputRef = _inGameDevicesList.GetInputRef(playerInput);
        if (inputRef != null)
            inputRef.SpawnedPlayer = null;
    }

    private void OnPlayerSpawn(PlayerRef player)
    {
        _targetGroup.AddMember(player.transform, 1, 7);

        SpawnedPlayersList.Add(player);
    }


}

public class Node : MonoBehaviour
{
    public List<Node> connections;
}
public class PathData
{
    public PathData(int distance)
    {
        Distance = distance;
    }

    public Node CameFrom;
    public int Distance;
}
public class PathfindingManager : MonoBehaviour, IService, IDisposable
{
    public List<Transform> grid;
    public List<Node> nodeList;
    public Node nodePrefab;

    [SerializeField] private float _navigationUpdateRate = 0.5f;
    [SerializeField] private bool _hasNavUpdate = true;

    public  Dictionary<PlayerRef, PlayerPathMap> PlayerPaths = new();
    private bool _isInit;
```

```
    public void Init()
    {
        CreateNodes();
        StartCoroutine(UpdateNavigation());
        _isInit = true;
        EventBus.OnPlayerJoined += OnPlayerJoined;
        EventBus.OnPlayerDied += OnPlayerDied;
    }

    public void Dispose()
    {
        EventBus.OnPlayerJoined -= OnPlayerJoined;
        EventBus.OnPlayerDied -= OnPlayerDied;
    }


    private void OnPlayerJoined(PlayerRef playerRef)
    {
        var playerList =
ServiceLocator.Current.Get<PlayerManager>().SpawnedPlayersList;
        foreach (var player in playerList)
        {
            PlayerPaths.TryAdd(player, null);
        }

    }

    private void OnPlayerDied(PlayerRef playerRef, HitData hitData)
    {
        var playerList =
ServiceLocator.Current.Get<PlayerManager>().SpawnedPlayersList;
        foreach (var player in playerList)
        {
            PlayerPaths.Remove(player);

        }
    }

    public Vector2 GetDirection()
    {
        int choice = Mathf.FloorToInt(UnityEngine.Random.value * 3.99f);

        switch (choice)
        {
            case 0:
                return Vector2.down;
            case 1:
                return Vector2.up;
            case 2:
                return Vector2.left;
            case 3:
                return Vector2.right;
            default: return Vector2.zero;
        }
    }

    void CreateNodes()
    {
        foreach (var cell in grid)
        {
```

```
            Node node = Instantiate(nodePrefab, new Vector3(cell.position.x,
cell.position.y, cell.position.z), Quaternion.identity);
            node.transform.SetParent(transform);
            nodeList.Add(node);
        }
        CreateConnections();
    }

    void ConnectNodes(Node from, Node to)
    {
        if(from == to) { return; }

        from.connections.Add(to);
    }

    void CreateConnections()
    {
        for(int i = 0; i < nodeList.Count; i++)
        {
            for(int j = i + 1; j < nodeList.Count; j++)
            {
                if (Vector3.Distance(nodeList[i].transform.position,
nodeList[j].transform.position) <= 3f &&
Vector3.Distance(nodeList[i].transform.position, nodeList[j].transform.position) >
1f)
                {
                    ConnectNodes(nodeList[i], nodeList[j]);
                    ConnectNodes(nodeList[j], nodeList[i]);
                }
            }
        }
    }

    public PlayerPathMap GetPlayerPaths(PlayerRef player)
    {
        return PlayerPaths.TryGetValue(player, out var map) ? map : null;
    }

    private IEnumerator UpdateNavigation()
    {
        while (true)
        {
            yield return new WaitForSeconds(_navigationUpdateRate);

            if (!_hasNavUpdate)
                continue;

            foreach (var player in
ServiceLocator.Current.Get<PlayerManager>().SpawnedPlayersList)
            {
                var nearestNode = FindNearestNode(player.transform.position);
                UpdatePlayerPaths(player, nearestNode);
            }
        }
    }

    public void UpdatePlayerPaths(PlayerRef player, Node startNode)
    {
        var pathMap = GenerateWightNavigation(startNode);
        PlayerPaths[player] = new PlayerPathMap(pathMap);
```

```
}

public void GetDirectionToNearestPlayerByNode()
{

}

private Dictionary<Node, PathData> GenerateWightNavigation(Node start)
{
    var result = new Dictionary<Node, PathData>();
    foreach (var node in nodeList)
        result[node] = new PathData(int.MaxValue);

    result[start].Distance = 0;
    result[start].CameFrom = start;

    var queue = new Queue<Node>();
    queue.Enqueue(start);

    while (queue.Count > 0)
    {
        var current = queue.Dequeue();
        var currentData = result[current];

        foreach (var neighbor in current.connections)
        {
            var neighborData = result[neighbor];
            if (neighborData.Distance == int.MaxValue)
            {
                neighborData.Distance = currentData.Distance + 1;
                neighborData.CameFrom = current;
                queue.Enqueue(neighbor);
            }
        }
    }

    return result;
}

public Node FindNearestNode(Vector3 pos)
{
    Node foundNode = null;
    float minDistance = float.MaxValue;

    foreach(Node node in FindObjectsOfType<Node>())
    {
        float currentDistance = Vector3.Distance(pos, node.transform.position);

        if(currentDistance < minDistance)
        {
            minDistance = currentDistance;
            foundNode = node;
        }
    }

    return foundNode;
}

public Node FindFurthestNode(Vector3 pos)
{
```

```
        Node foundNode = null;
        float maxDistance = default;

        foreach (Node node in FindObjectsOfType<Node>())
        {
            float currentDistance = Vector3.Distance(pos, node.transform.position);
            if(currentDistance > maxDistance)
            {
                maxDistance = currentDistance;
                foundNode = node;
            }
        }

        return foundNode;
    }

    public Node[] AllNodes()
    {
        return nodeList.ToArray();
    }

    public class PlayerPathMap
    {
        public PlayerPathMap(Dictionary<Node, PathData> pathMap)
        {
            PathMap = pathMap;
        }

        public Dictionary<Node, PathData> PathMap = new();
    }
}
public class EnemiesManager : MonoBehaviour, IService, IDisposable
{
    public EnemyRef npc;
    public List<EnemyRef> EnemyList => _enemyList;

    [SerializeField] private int _maxEnemies = 0;

    private PathfindingManager _pathfindingManager;
    private List<EnemyRef> _enemyList = new();
    private  List<PlayerRef> _playerList = new();
    private bool _isInit;

    public void Init()
    {
        _pathfindingManager = ServiceLocator.Current.Get<PathfindingManager>();
        SpawnEnemies(_maxEnemies - _enemyList.Count);
        EventBus.OnPlayerJoined += OnPlayerJoined;
        EventBus.OnPlayerDied += OnPlayerDied;
        EventBus.OnEnemyDied += OnEnemyDied;
        _isInit = true;
    }

    public void Dispose()
    {
        EventBus.OnPlayerJoined -= OnPlayerJoined;
        EventBus.OnPlayerDied -= OnPlayerDied;
    }

    private void OnPlayerJoined(PlayerRef playerRef)
```

```
    {
        _playerList.Add(playerRef);
    }

    private void OnPlayerDied(PlayerRef playerRef, HitData hitData)
    {
        if (_playerList.Contains(playerRef))
            _playerList.Remove(playerRef);
    }

    public void SpawnEnemies(int count)
    {
        for (int i = 1; i < count; i++)
        {
            SpawnAI();
        }
    }

    private void SpawnAI()
    {
        var nodeList = _pathfindingManager.nodeList;
        if (nodeList.Count == 0)
            return;

        Node furthestNode;
        if (_playerList.Count > 0)
            furthestNode =
                _pathfindingManager.FindFurthestNode(_playerList[Random.Range(0,
_playerList.Count)].transform
                    .position);
        else
            furthestNode = nodeList[Random.Range(0, nodeList.Count)];
        Node randNode = furthestNode;

        EnemyRef newEnemy = Instantiate(npc, randNode.transform.position,
Quaternion.identity);

        _enemyList.Add(newEnemy);
        newEnemy.FsmEnemyAI.InitalNode = randNode;
    }

    private void OnEnemyDied(EnemyRef enemyRef, HitData hitData)
    {
        if (_enemyList.Contains(enemyRef))
        {
            _enemyList.Remove(enemyRef);
            StartCoroutine(DestroyEnemy(enemyRef.gameObject));
        }

        SpawnEnemies(_maxEnemies - _enemyList.Count);
    }

    private IEnumerator DestroyEnemy(GameObject enemy)
    {
        yield return new WaitForSeconds(1f);
        Destroy(enemy);
    }
}
public class EnemyRef : MonoBehaviour, ITeamMember
{
```

```
    [SerializeField] public Rigidbody Rigidbody;
    [SerializeField] public FSMEnemyAI FsmEnemyAI;
    [SerializeField] public Animator Animator;
    [SerializeField] public MeleeAttack MeleeAttack;
    [SerializeField] public Transform AttackTransform;
    [SerializeField] public EnemyHealth EnemyHealth;
    [SerializeField] public DissolveChilds Dissolve;
    public Team Team { get; set; }

    private PlayerManager _playerManager;
    private List<PlayerRef> _playerList = new();

    private void Start()
    {
        Team = Team.Enemies;
        _playerManager = ServiceLocator.Current.Get<PlayerManager>();
        _playerList = _playerManager.SpawnedPlayersList;
        FsmEnemyAI.Init(this, _playerList);
        EnemyHealth.Init(this);
    }
}
public class FSM
{
    private FSMState StateCurrent { get; set; }

    private Dictionary<Type, FSMState> _states = new Dictionary<Type, FSMState>();

    public void AddState(FSMState state)
    {
        _states.Add(state.GetType(), state);
    }

    public void SetState<T>() where T : FSMState
    {
        var type = typeof(T);

        if (StateCurrent != null && StateCurrent.GetType() == type)
            return;

        if (_states.TryGetValue(type, out var newState))
        {
            StateCurrent?.Exit();

            StateCurrent = newState;

            StateCurrent.Enter();
        }
    }

    public void Update()
    {
        StateCurrent?.Update();
    }

    public void FixedUpdate()
    {
        StateCurrent?.FixedUpdate();
    }
}
public abstract class FSMState
```

```
{
    protected readonly FSM _fsm;

    public FSMState(FSM fsm)
    {
        _fsm = fsm;
    }

    public virtual void Enter() {}
    public virtual void Exit() {}
    public virtual void Update() {}
    public virtual void FixedUpdate() {}
}
public class FSMEnemyAI : MonoBehaviour
{
public Node InitalNode { get; set; }

[SerializeField] private float _speed;

private PathfindingManager _pathfindingManager;
private FSM _fsm;
private bool _isInit;

public void Init( EnemyRef enemyRef, List<PlayerRef> playerList)
{
    _pathfindingManager = ServiceLocator.Current.Get<PathfindingManager>();

    _fsm = new FSM();
    _fsm.AddState(new IdleEnemyState(_fsm, enemyRef, _speed, _pathfindingManager,
playerList, InitalNode));
    _fsm.AddState(new ChaseEnemyState(_fsm, enemyRef, _speed, _pathfindingManager,
playerList, InitalNode));
    _fsm.AddState(new AttackEnemyState(_fsm, enemyRef, _speed, _pathfindingManager,
playerList, InitalNode));

    _fsm.SetState<IdleEnemyState>();

    _isInit = true;
}

private void Update()
{
    _fsm.Update();
}
public abstract class Health : MonoBehaviour, IHitTarget
{
    public event Action<HitData> HitTaken;
    public event Action<HitData> FatalHitTaken;

    public float   CurrentHealth { get; protected set; }
    public bool    IsAlive       => CurrentHealth > 0f;
    public float   MaxHealth     => _maxHealth;

    [SerializeField]
    protected float _maxHealth = 100f;

    [SerializeField]
    protected float _immortalDamageTime = 0.67f;

    protected Timer _immortalTimer;
```

```
public bool IsActive { get; }

public bool IsImmortal { get; protected set; }

public void Init()
{
    SetHealth(_maxHealth);
    _immortalTimer = new Timer();
}
public void ProcessHit(ref HitData hitData)
{
    ApplyHit(ref hitData);

    if (hitData.Amount == 0)
        return;

    if (IsAlive == false)
    {
        hitData.IsFatal = true;
        OnDie(hitData);
    }

    OnHitTaken(ref hitData);
}

protected float AddHealth(float amount)
{
    float previousHealth = CurrentHealth;
    SetHealth(CurrentHealth + amount);
    return CurrentHealth - previousHealth;
}

protected float RemoveHealth(float amount)
{
    float previousHealth = CurrentHealth;
    SetHealth(CurrentHealth - amount);
    return previousHealth - CurrentHealth;
}

protected void SetHealth(float health)
{
    CurrentHealth = Mathf.Clamp(health, 0, _maxHealth);
}

protected void OnHitTaken(ref HitData hitData)
{
    HitTaken?.Invoke(hitData);

    if (hitData.IsFatal)
    {
        FatalHitTaken?.Invoke(hitData);
    }
}

protected void Update()
{
    if (_immortalTimer.IsFinished())
    {
        IsImmortal = false;
```

```
        }
        else
        {
            IsImmortal = true;
        }
    }

    protected void ApplyHit(ref HitData hitData)
    {
        if (IsAlive == false || IsImmortal)
        {
            hitData.Amount = 0f;
            return;
        }

        if (hitData.Action == EHitAction.Damage)
        {
            hitData.Amount = RemoveHealth(hitData.Amount);
            _immortalTimer.CreateFromSeconds(_immortalDamageTime);
        }
        else if (hitData.Action == EHitAction.Heal)
        {
            hitData.Amount = AddHealth(hitData.Amount);
        }

    }

    protected virtual void OnDie(HitData hitData)
    {
    }
}
[Serializable]
public class UpgradeBase
{
   public virtual string GetDiscription()
   {
      return "";
   }
}
public interface IDamage
{
    void Init(IDamage baseDamage);
    int GetDamage();
    EDamageType GetDamageType();
    void TakeDamage(IDamageReceiver damageReceiver);
    object Clone();
}

public interface IDamageReceiver
{
    void GetHit(DamageData damageData);
}
public struct DamageData
{
    public int Damage;
    public EDamageType DamageType;

    public DamageData(int damage, EDamageType damageType)
    {
        Damage = damage;
```

```
        DamageType = damageType;
    }
}
public class BaseDamageUpgrade : UpgradeBase, IDamage
{
    [SerializeField] public WeaponSlots WeaponSlots;
    protected IDamage BaseDamage;

    public virtual void Init(IDamage damage)
    {
        BaseDamage = damage;
    }
    public virtual int GetDamage()
    {
        return BaseDamage.GetDamage();
    }

    public virtual EDamageType GetDamageType()
    {
        return BaseDamage.GetDamageType();
    }

    public virtual void TakeDamage(IDamageReceiver damageReceiver)
    {
        BaseDamage.TakeDamage(damageReceiver);
    }

    public object Clone()
    {
        var clone = MemberwiseClone();
        return clone;
    }
}
public class AddDamageUpgrade : BaseDamageUpgrade
{
    [SerializeField] private int addBaseDamage;
    public override int GetDamage()
    {
        return BaseDamage.GetDamage() + addBaseDamage;
    }

    public override void TakeDamage(IDamageReceiver damageReceiver)
    {
        DamageData damageData = new DamageData(GetDamage(),
BaseDamage.GetDamageType());
        damageReceiver.GetHit(damageData);
    }

}
public class AdditionalDamageUpgrade : BaseDamageUpgrade
{
    [SerializeField, Header("In player")] private int _additionalDamage;
    [SerializeField] private EDamageType _additionalDamageType;
    public override void TakeDamage(IDamageReceiver damageReceiver)
    {
        BaseDamage.TakeDamage(damageReceiver);
        DamageData damageData = new DamageData(_additionalDamage,
_additionalDamageType);
        damageReceiver.GetHit(damageData);
    }
```

```
}
public class UpgradeHandler : MonoBehaviour
{
    public ProjectileBase Projectile { get; private set; }
    private List<UpgradeBase> UpgradeList;
    private PlayerRef _playerRef;

    public bool Init(PlayerRef playerRef)
    {
        _playerRef = playerRef;
        return true;
    }

    public void ApplyUpgrade(Upgrade upgrade)
    {
        var upgradeList = upgrade.Upgrades.List;
        if (upgradeList != null)
            UpgradeList = upgradeList;

        var projectile = upgrade.ProjectileUpgrade;
        if (projectile != null)
            Projectile = projectile;

        _playerRef.OnUpgradeReceive?.Invoke(this);
    }

    public T GetUpgrade<T>() where T : class
    {
        foreach (var obj in UpgradeList)
        {
            if (obj is T interfaceImplementation)
            {
                return interfaceImplementation;
            }
        }
        return null;
    }
}
```