Міністерство освіти і науки України

Український державний університет науки і технологій

Навчально-науковий інститут
«Український державний хіміко-технологічний університет»
(назва навчального-наукового інституту)

Факультет комп'ютерних наук та інженерії
(повна назва факультету)

Кафедра комп'ютерно-інтегрованих технологій та робототехніки
(повна назва кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

бакалавр
(освітній рівень)

на тему: Розробка комп'ютерної гри у жанрі метроідванія з використанням рушія Unity

Виконав студент 4 курсу, групи КІ
спеціальності
123 «Комп'ютерна інженерія»
(код і назва спеціальності)

Студент Богданов Є. Є.
(прізвище, ім'я, по-батькові) (підпис)

Керівник Хорошилов С. В.
(прізвище, ім'я, по-батькові) (підпис)

Рецензент Дубовик Т. М.
(прізвище, ім'я, по-батькові) (підпис)

Дніпро – 2026 рік

Державний вищий навчальний заклад

«Український державний хіміко-технологічний університет»

(повне найменування вищого навчального закладу)

Факультет, відділення комп'ютерних наук та інженерії

Кафедра комп'ютерно-інтегрованих технологій та робототехніки

Освітній рівень бакалвр

Спеціальність 123 комп'ютерна інженерія

(код і назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри

канд. техн. наук, доц. Левчук І. Л.

«12» червня 2026 р.

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТУ

Богданову Євгенію Євгенійовичу

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) Розробка комп'ютерної гри у жанрі метроідванія з використанням рушія Unity

керівник проекту (роботи) Хорошилов Сергій Вікторович док. техн. наук, професор,

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від 02.03.2026 №77-к

2. Строк подання студентом проекту (роботи) 08.06.2026 р.

3. Вихідні дані до проекту (роботи) Матеріали виробничої практики, дані технічної літератури

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

Загальний розділ, Основний розділ, Розрахунковий розділ, Охорона праці, Організвційно-економічний розділ.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Презентація до дипломної роботи

6. Консультанти розділів роботи

Розділ	Ініціали, прізвище та посада консультанта	Підпис, дата
завдання видав	завдання прийняв	10.02.2026 р. 10.05.2026 р.
Охорона праці та безпека в надзвичайних ситуаціях	С.В. ХОРОШИЛОВ, докт. техн. наук, професор	10.02.2026 р. 10.05.2026 р.
Організаційно-економічний розділ	С.В. ХОРОШИЛОВ, докт. техн. наук, професор	10.03.2026 р. 10.05.2026 р.

7. Дата видачі завдання «14» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Бібліографічний огляд за темою	лютий	виконано
2	Робота над розділом «Загальний розділ»	лютий	виконано
3	Робота над розділом «Основний розділ»	березень	виконано
4	Робота над розділом «Розрахунковий розділ»	квітень	виконано
5	Робота над розділом «Охорона праці та безпека в надзвичайних ситуаціях»	травень	виконано
6	Робота над розділом «Організаційно-економічний розділ»	травень	виконано
7	Розробка презентації	травень	виконано
8	Перевірка роботи на плагіат	08.06.2026	виконано

Студент _____ Є.Є. БОГДАНОВ
(підпис) (ініціали, прізвище)

Керівник роботи _____ С.В. ХОРОШИЛОВ
(підпис) (ініціали, прізвище)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи на тему: «Розробка комп’ютерної гри у жанрі метроідванія з використанням рушія Unity».

Обсяг пояснювальної записки – 214 сторінок, 68 рисунків, 4 таблиць, 2 додатків, 15 використаних джерел.

Об’єкт дослідження – процес розробки комп’ютерних ігор жанру Metroidvania.

Мета роботи – розробка комп’ютерної гри у жанрі Metroidvania з використанням рушія Unity.

У процесі виконання дипломної роботи проведено аналіз особливостей жанру Metroidvania, досліджено сучасні ігрові рушії для розробки двовимірних комп’ютерних ігор та обґрунтовано вибір рушія Unity як середовища розробки.

У результаті виконання роботи розроблено програмний прототип комп’ютерної гри жанру Metroidvania, у якому реалізовано систему керування персонажем, бойову систему, систему анімацій, систему здоров’я, штучний інтелект противників, механізм збору монет, систему контрольних точок, переходи між сценами та користувацький інтерфейс.

Практична цінність роботи полягає у створенні працездатного програмного продукту, який демонструє можливості використання рушія Unity для розробки комп’ютерних ігор та може бути використаний як основа для подальшого розвитку повноцінного ігрового проєкту.

Ключові слова: UNITY, METROIDVANIA, КОМП’ЮТЕРНА ГРА, ІГРОВИЙ РУШІЙ, C#, ПРОГРАМУВАННЯ, ШТУЧНИЙ ІНТЕЛЕКТ, АЛГОРИТМ, ІГРОВИЙ ПЕРСОНАЖ, ПРОГРАМНИЙ ПРОДУКТ.

ПЕРЕЛІК СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

AI (Artificial Intelligence) – штучний інтелект.

API (Application Programming Interface) – програмний інтерфейс застосунку.

CPU (Central Processing Unit) – центральний процесор.

FPS (Frames Per Second) – кількість кадрів за секунду.

HP (Health Points) – показник здоров’я персонажа.

NPC (Non-Player Character) – неігровий персонаж.

UI (User Interface) – користувацький інтерфейс.

Unity – ігровий рушій для створення двовимірних та тривимірних комп’ютерних ігор.

C# – об’єктно-орієнтована мова програмування, що використовується для розробки програмного продукту.

Tilemap – система створення ігрових рівнів із використанням тайлової графіки.

Sprite – двовимірне графічне зображення, що використовується в ігровому середовищі.

Rigidbody2D – компонент Unity для реалізації фізичних взаємодій у двовимірному просторі.

Collider2D – компонент Unity, призначений для обробки зіткнень між об’єктами.

Зміст

РЕФЕРАТ 3
ПЕРЕЛІК СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ 4
ВСТУП 7
ЗАГАЛЬНИЙ РОЗДІЛ 10
1.1 Комп’ютерні ігри як різновид програмних систем 10
1.2 Особливості жанру Metroidvania 13
1.3 Аналіз сучасних представників жанру Metroidvania 18
1.4 Hollow Knight 19
1.5 Ori and the Blind Forest 20
1.6 Blasphemous 22
1.7 Порівняльний аналіз аналогів 23
1.8 Ігрові рушії для розробки двовимірних комп’ютерних ігор 25
1.9 Unity 25
1.10 Unreal Engine 26
1.11 Godot 27
1.12 Порівняння сучасних ігрових рушіїв 28
1.13 Обґрунтування вибору рушія Unity 29
ОСНОВНИЙ РОЗДІЛ 31
2.1 Загальна концепція програмного продукту 31
2.2 Архітектура програмного продукту 33
2.3 Структура проєкту та організація ресурсів у середовищі Unity 40
2.4 Реалізація системи керування персонажем 46
2.5 Реалізація бойової системи 53
2.6 Реалізація системи анімацій 61
2.7 Реалізація системи здоров'я персонажа 68
2.8 Реалізація переходів між сценами 74
РОЗРАХУНКОВИЙ РОЗДІЛ 81
3.1 Розробка штучного інтелекту противника 81
3.2 Реалізація системи збору монет 86
3.3 Реалізація контрольних точок 90
3.4 Реалізація здібності Dash 93
3.5 Організація користувацького інтерфейсу 97
3.6 Тестування програмного продукту 103
ОХОРОНА ПРАЦІ 108
4.1 Аналіз умов праці розробника програмного забезпечення 108

4.2 Освітлення робочого місця 111
4.3 Мікроклімат приміщення 114
4.4 Електробезпека 117
4.5 Пожежна безпека 121
4.6 Режим праці та відпочинку 124
ОРГАНІЗАЦІЙНО-ЕКОНОМІЧНИЙ РОЗДІЛ 128
5.1 Обґрунтування доцільності розробки 128
5.2 Оцінка трудомісткості розробки 130
5.3 Розрахунок вартості розробки 132
5.4 Розрахунок заробітної плати розробника 134
5.5 Оцінка економічної ефективності 137
ВИСНОВКИ 140
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 142
ДОДАТОК А 143
ДОДАТОК Б 170

ВСТУП

Комп’ютерні ігри є одним із найбільш динамічних напрямів розвитку сучасної індустрії програмного забезпечення. Завдяки стрімкому розвитку обчислювальної техніки, графічних технологій та програмних засобів розробка комп’ютерних ігор перетворилася на складний процес створення інтерактивних програмних систем, що поєднують програмування, комп’ютерну графіку, моделювання фізичних процесів, проєктування користувацького інтерфейсу та штучний інтелект.

Сучасна ігрова індустрія займає важливе місце серед галузей інформаційних технологій та демонструє стабільне зростання. Комп’ютерні ігри використовуються не лише для розваг, але й у навчанні, тренуванні професійних навичок, візуалізації складних процесів та популяризації цифрових технологій. У зв’язку з цим розробка програмних продуктів ігрового призначення є актуальним напрямом підготовки фахівців у галузі комп’ютерної інженерії та програмної інженерії.

Одним із популярних напрямів сучасної ігрової індустрії є жанр Metroidvania. Особливістю даного жанру є поєднання платформних механік, бойової системи, дослідження ігрового світу та поступового відкриття нових можливостей персонажа. На відміну від лінійних ігор, жанр Metroidvania надає користувачеві можливість самостійно досліджувати ігрові локації та знаходити нові шляхи проходження після отримання спеціальних здібностей або предметів.

Популярність жанру підтверджується успіхом таких проєктів, як Hollow Knight, Ori and the Blind Forest, Ori and the Will of the Wisps, Blasphemous та інших. Дані ігри демонструють високий рівень зацікавленості користувачів у проєктах, які поєднують динамічний ігровий процес, елементи дослідження світу та розвиток персонажа.

Для створення сучасних комп’ютерних ігор широко використовуються спеціалізовані ігрові рушії. Одним із найбільш поширених інструментів є Unity. Даний рушій забезпечує широкий набір засобів для створення двовимірних і тривимірних програмних продуктів, підтримує компонентно-орієнтовану архітектуру, має інтегровані засоби роботи з фізикою, анімацією, звуком та користувацьким інтерфейсом. Крім того, Unity підтримує мову програмування C#, що забезпечує високу гнучкість при реалізації ігрової логіки.

Актуальність теми дипломної роботи обумовлена необхідністю дослідження сучасних підходів до розробки інтерактивних програмних систем та практичного застосування технологій створення комп’ютерних ігор із використанням ігрових рушіїв. Розробка власного програмного продукту дозволяє отримати практичний досвід проєктування архітектури програмного забезпечення, реалізації механік взаємодії користувача з системою, створення штучного інтелекту ігрових персонажів та організації користувацького інтерфейсу.

Об’єктом дослідження є процес розробки комп’ютерних ігор жанру Metroidvania.

Предметом дослідження є методи та програмні засоби створення двовимірної комп’ютерної гри із використанням рушія Unity.

Метою дипломної роботи є розробка комп’ютерної гри у жанрі Metroidvania з використанням рушія Unity.

Для досягнення поставленої мети необхідно вирішити такі завдання:

провести аналіз особливостей жанру Metroidvania та сучасних аналогів;

дослідити можливості сучасних ігрових рушіїв для створення двовимірних комп’ютерних ігор;

обґрунтувати вибір рушія Unity для реалізації програмного продукту;

розробити архітектуру програмного забезпечення гри;

реалізувати систему керування персонажем;

реалізувати бойову систему та систему взаємодії з противниками;

реалізувати систему анімацій персонажа та противників;

реалізувати систему переходів між ігровими локаціями;

реалізувати систему збору ігрових об’єктів та користувацький інтерфейс;

провести тестування розробленого програмного продукту.

Практична цінність роботи полягає у створенні працездатного програмного прототипу комп’ютерної гри жанру Metroidvania, який демонструє можливості використання рушія Unity для розробки інтерактивних програмних систем та може бути використаний як основа для подальшого розвитку повноцінного ігрового проєкту.

У результаті виконання дипломної роботи розроблено програмний прототип двовимірної комп’ютерної гри, що реалізує основні механіки жанру Metroidvania, включаючи переміщення персонажа, бойову систему, взаємодію з противниками, систему анімацій, збір ігрових об’єктів та переходи між локаціями.

ЗАГАЛЬНИЙ РОЗДІЛ

1.1 Комп’ютерні ігри як різновид програмних систем

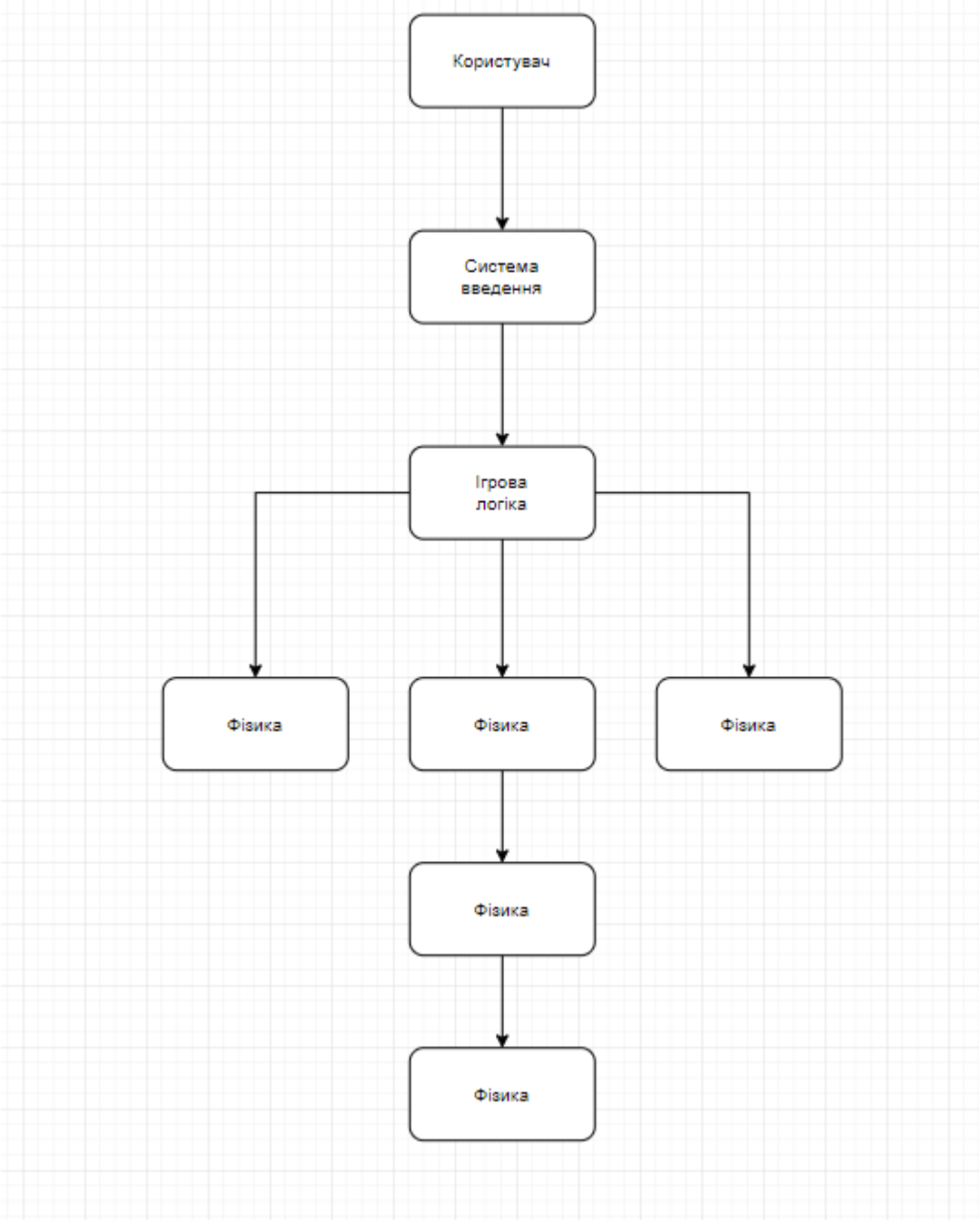


Рисунок 1.1 – Загальна структура сучасної комп’ютерної гри

Сучасні комп’ютерні технології охоплюють практично всі сфери діяльності людини. Одним із напрямів розвитку програмного забезпечення є створення інтерактивних мультимедійних систем, до яких належать комп’ютерні ігри. Протягом останніх десятиліть індустрія розробки комп’ютерних ігор перетворилася на окрему галузь інформаційних технологій, що об’єднує програмування, комп’ютерну графіку, математичне моделювання, штучний інтелект, цифрову обробку звуку та проєктування користувацьких інтерфейсів.

Комп’ютерна гра являє собою програмну систему, призначену для взаємодії користувача з віртуальним середовищем у режимі реального часу. На відміну від більшості традиційних програмних продуктів, комп’ютерна гра повинна не лише виконувати певний набір функцій, але й забезпечувати безперервну взаємодію з користувачем, швидке реагування на його дії та підтримувати високий рівень продуктивності.

Сучасні комп’ютерні ігри складаються з великої кількості взаємопов’язаних програмних компонентів. До основних компонентів належать система керування персонажем, система фізичних взаємодій, модуль штучного інтелекту, система анімації, графічна підсистема, система обробки введення користувача та користувацький інтерфейс.

Однією з ключових особливостей комп’ютерних ігор є необхідність обробки значного обсягу інформації в режимі реального часу. Під час роботи гри відбувається постійне оновлення стану ігрового світу, розрахунок фізичних процесів, обробка дій користувача, оновлення анімацій та відображення графічної інформації. Всі ці процеси повинні виконуватися з високою швидкістю для забезпечення плавного ігрового процесу.

Архітектура сучасних комп’ютерних ігор базується на принципах модульності та компонентного підходу. Це дозволяє розділити складну програмну систему на окремі незалежні модулі, кожен з яких виконує певний

набір функцій. Такий підхід спрощує розробку, тестування та подальшу підтримку програмного продукту.

Важливим елементом будь-якої комп’ютерної гри є система взаємодії користувача з ігровим світом. Вона включає механізми обробки натискань клавіш, руху миші, використання контролерів та інших пристроїв введення. Від швидкості та коректності роботи цієї системи значною мірою залежить комфорт користувача під час проходження гри.

Ще однією складовою сучасних ігор є система анімації. Анімація використовується для відображення рухів персонажів, роботи різноманітних механізмів, зміни стану об’єктів та створення візуальних ефектів. Використання анімації дозволяє зробити взаємодію з ігровим світом більш природною та зрозумілою для користувача.

Для забезпечення реалістичності ігрового процесу широко застосовуються фізичні моделі. Фізичний рушій відповідає за обчислення переміщення об’єктів, обробку зіткнень, моделювання сили тяжіння та інших фізичних явищ. Використання фізичного моделювання значно підвищує рівень занурення користувача в ігровий процес.

Особливе місце у структурі комп’ютерних ігор займає штучний інтелект. Основним завданням штучного інтелекту є керування поведінкою неігрових персонажів та створення відчуття взаємодії з розумним віртуальним середовищем. Для цього використовуються різноманітні алгоритми прийняття рішень, системи станів, дерева поведінки та інші методи.

Останніми роками спостерігається активний розвиток незалежної розробки комп’ютерних ігор. Завдяки появі сучасних ігрових рушіїв невеликі команди та окремі розробники отримали можливість створювати якісні програмні продукти без необхідності розробки власних технологічних платформ. Це сприяло появі великої кількості успішних

незалежних проєктів, які здобули популярність серед користувачів у всьому світі.

Для створення сучасних комп'ютерних ігор широко використовуються спеціалізовані ігрові рушії. Ігровий рушій являє собою програмну платформу, що забезпечує розробника готовими інструментами для реалізації графіки, фізики, анімації, звукового супроводу та інших елементів ігрового процесу. Використання рушіїв дозволяє значно скоротити час розробки та зосередити увагу на створенні безпосередньо ігрового контенту.

Серед найбільш популярних сучасних ігрових рушіїв можна виділити Unity, Unreal Engine та Godot. Кожен із них має власні особливості, переваги та сферу застосування. Unity отримав значне поширення серед незалежних розробників завдяки простоті використання, великій кількості навчальних матеріалів та підтримці розробки як двовимірних, так і тривимірних проєктів.

Розробка комп'ютерних ігор є складним багатокомпонентним процесом, що вимагає поєднання знань з різних галузей інформаційних технологій. Створення навіть відносно невеликого ігрового проєкту передбачає проєктування архітектури програмного забезпечення, реалізацію систем керування персонажем, створення штучного інтелекту, розробку користувацького інтерфейсу та забезпечення належного рівня продуктивності програмного продукту.

Таким чином, комп'ютерні ігри є складними програмними системами, які поєднують сучасні методи програмування, засоби комп'ютерної графіки та технології інтерактивної взаємодії з користувачем. Постійний розвиток апаратного забезпечення та програмних технологій сприяє подальшому вдосконаленню ігрових систем та розширенню можливостей їх застосування.

1.2 Особливості жанру Metroidvania

Жанр Metroidvania є одним із найбільш популярних напрямів у сфері двовимірних комп'ютерних ігор. Він сформувався в результаті поєднання ідей

двох відомих серій відеоігор — Metroid та Castlevania. Саме від назв цих серій походить термін «Metroidvania», який сьогодні використовується для позначення окремого жанру комп'ютерних ігор.

Перші ігри, що заклали основи даного жанру, з'явилися наприкінці XX століття. Особливу популярність концепція отримала після виходу гри Castlevania: Symphony of the Night, яка поєднала платформні механіки з нелінійним дослідженням ігрового світу. Надалі дана концепція активно розвивалася та стала основою для великої кількості успішних проєктів.

На відміну від традиційних платформерів, де гравець рухається по заздалегідь визначеному лінійному маршруту, ігри жанру Metroidvania пропонують великий взаємопов'язаний світ, який можна досліджувати в різних напрямках. Значна частина локацій залишається недоступною на початкових етапах гри та відкривається лише після отримання певних здібностей або предметів.

Однією з головних особливостей жанру є концепція поступового розвитку персонажа. У процесі проходження гри користувач отримує нові можливості, які дозволяють долати раніше недоступні перешкоди. Такий підхід створює відчуття постійного прогресу та мотивує гравця повертатися до вже відвіданих локацій для подальшого дослідження.

Рисунок 1.2 – Приклад системи прогресії персонажа

Важливою характеристикою жанру Metroidvania є нелінійність проходження. Після отримання нових здібностей користувач може самостійно обирати напрямок дослідження світу. Це дозволяє створити відчуття свободи та значно підвищує рівень залученості гравця в ігровий процес.

Ігровий світ у проєктах жанру Metroidvania зазвичай складається з великої кількості взаємопов’язаних локацій. Локації можуть бути об’єднані переходами, порталами, дверима або іншими механізмами переміщення між областями. Така структура світу забезпечує поступове відкриття нових територій та підтримує інтерес користувача протягом усього проходження гри.

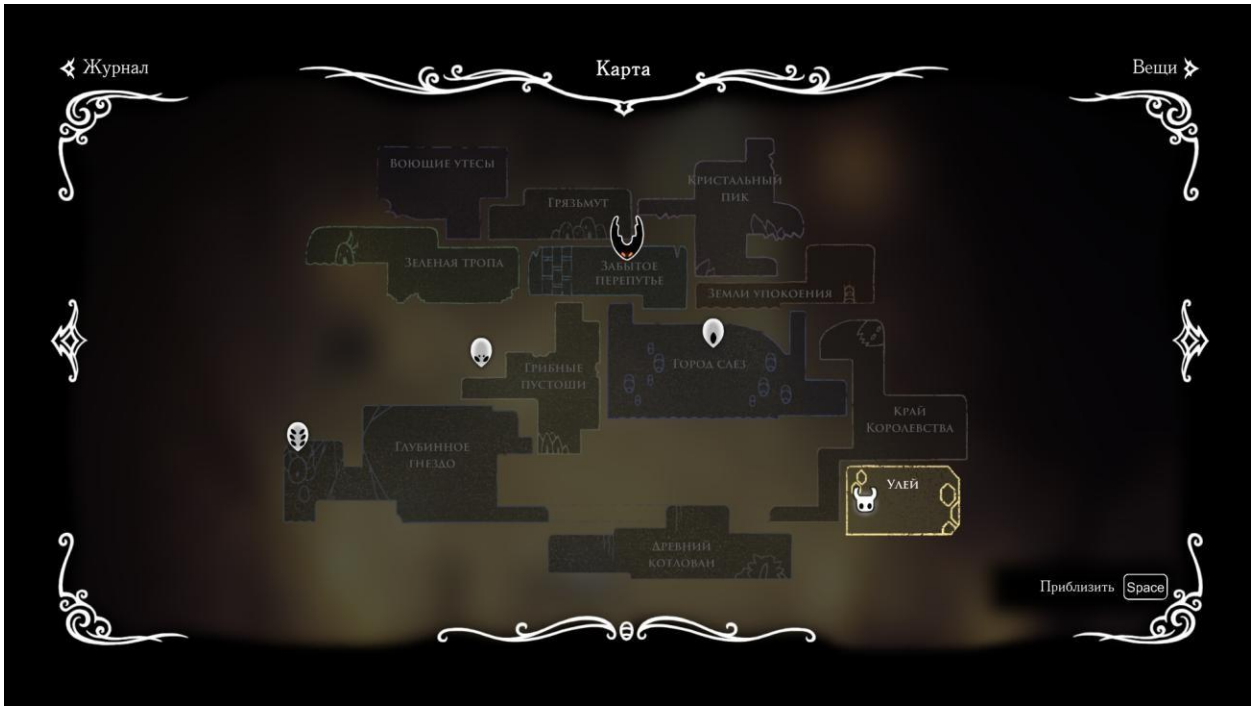


Рисунок 1.3 – Приклад взаємопов'язаного ігрового світу жанру Metroidvania

Однією з найбільш характерних механік жанру є система блокування доступу до певних зон. Для потрапляння до нової області гравець повинен отримати спеціальну здатність або предмет. Наприклад, високий уступ може

бути недоступним до моменту отримання подвійного стрибка, а зачинені двері можуть відкриватися лише після знаходження відповідного ключа.

Подібна система часто називається progression gating та є одним із ключових елементів дизайну Metroidvania. Вона дозволяє керувати порядком проходження гри, одночасно зберігаючи відчуття свободи дослідження.

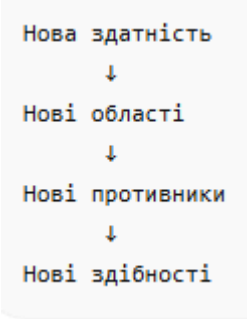


Рисунок 1.4 – Принцип progression gating у жанрі Metroidvania

Важливе місце в іграх жанру займає бойова система. У більшості випадків персонаж має можливість використовувати зброю ближнього бою або різноманітні спеціальні здібності для боротьби з противниками. Бойова система повинна бути достатньо динамічною та зрозумілою для користувача, оскільки значна частина ігрового процесу пов’язана саме з подоланням ворогів.

У сучасних представниках жанру часто використовуються комбінації атак, спеціальні удари, система ухилення від атак противника та різноманітні модифікації зброї. Такі механіки роблять бойову систему більш глибокою та різноманітною.

Ще однією характерною особливістю Metroidvania є велика кількість противників різних типів. Вороги можуть мати різну поведінку, швидкість пересування, дальність атаки та рівень складності. Для створення цікавого ігрового процесу широко використовуються системи

штучного інтелекту, які дозволяють противникам реагувати на дії гравця та змінювати свою поведінку залежно від ситуації.

Крім звичайних противників, у багатьох іграх жанру присутні боси. Бос являє собою особливого противника з підвищеним запасом здоров’я, складнішою поведінкою та унікальними атаками. Перемога над босом зазвичай відкриває доступ до нової області або надає гравцю нову здібність.

Система збереження прогресу також є важливим елементом жанру. У більшості випадків використовуються контрольні точки або спеціальні місця відпочинку, де користувач може зберегти свій прогрес та відновити запас здоров’я персонажа. Такий підхід дозволяє зменшити рівень фрустрації та зробити проходження гри більш комфортним.

Значну роль у створенні атмосфери відіграє візуальне оформлення. Багато сучасних незалежних проєктів жанру Metroidvania використовують двовимірну графіку та стиль Pixel Art. Подібний підхід дозволяє створити впізнаваний художній стиль та одночасно зменшити витрати на розробку графічного контенту.

Перевагою використання двовимірної графіки є також спрощення процесу розробки. У порівнянні з тривимірними проєктами двовимірні ігри потребують менше ресурсів як під час створення, так і під час виконання на комп’ютері користувача.

Важливою складовою жанру є система дослідження світу. Гравець постійно знаходить нові області, секретні кімнати, приховані проходи та додаткові предмети. Саме елемент дослідження значною мірою відрізняє Metroidvania від звичайних платформерів та бойових ігор.

Сучасні представники жанру часто включають систему колекційних предметів. До них можуть належати монети, ресурси, артефакти, покращення характеристик персонажа або інші об’єкти, які стимулюють більш детальне дослідження ігрового світу.

Ще однією особливістю є поступове ускладнення ігрового процесу. У міру проходження гри користувач зустрічає сильніших противників, складніші перешкоди та більш заплутані локації. Такий підхід забезпечує постійне зростання складності та підтримує інтерес до проходження.

З точки зору програмної реалізації ігри жанру Metroidvania поєднують велику кількість різноманітних систем. До них належать система керування персонажем, система бойової взаємодії, штучний інтелект противників, система анімацій, система переходів між локаціями, механізми збереження прогресу та користувацький інтерфейс.

Таким чином, жанр Metroidvania є складним напрямом розробки комп'ютерних ігор, який поєднує елементи платформера, пригодницької гри та бойової системи. Основними особливостями жанру є нелінійне дослідження світу, поступовий розвиток персонажа, відкриття нових здібностей, наявність бойової системи та взаємопов'язаної структури локацій. Саме ці особливості визначають вимоги до програмної реалізації комп'ютерної гри даного жанру.

1.3 Аналіз сучасних представників жанру Metroidvania

Для визначення основних вимог до програмної реалізації комп'ютерної гри жанру Metroidvania доцільно провести аналіз сучасних представників даного жанру. Вивчення успішних комерційних проєктів дозволяє визначити найбільш популярні механіки, підходи до проєктування ігрового світу, особливості бойової системи та способи реалізації прогресії персонажа.

Серед великої кількості ігор жанру Metroidvania особливу увагу заслуговують проєкти Hollow Knight, Ori and the Blind Forest та Blasphemous. Дані ігри отримали високу оцінку як від користувачів, так і від професійних оглядачів, а також суттєво вплинули на подальший розвиток жанру.

1.4 Hollow Knight

Рисунок 1.5 – скріншот гри Hollow Knight

Hollow Knight є однією з найуспішніших незалежних ігор жанру Metroidvania. Гра була розроблена австралійською студією Team Cherry та випущена у 2017 році. Проєкт швидко здобув популярність завдяки якісному художньому оформленню, великому взаємопов'язаному світу та глибокому ігровому процесу.

Події гри відбуваються у вигаданому підземному королівстві Hallownest, яке складається з великої кількості взаємопов'язаних локацій. Гравець керує персонажем, який досліджує світ, бореться з противниками та поступово відкриває нові можливості для подальшого проходження.

Однією з головних особливостей Hollow Knight є структура світу. На відміну від багатьох традиційних платформерів, усі локації гри об'єднані в єдину систему переходів. Отримання нових здібностей дозволяє повертатися до вже відвіданих областей та відкривати нові маршрути проходження.

Система розвитку персонажа побудована навколо отримання нових умінь. У процесі проходження користувач відкриває ривок, подвійний стрибок, можливість пересування по стінах та інші здібності. Кожна нова можливість відкриває доступ до раніше недоступних ділянок карти.

Бойова система гри заснована на використанні зброї ближнього бою та спеціальних магічних здібностей. Для успішного проходження необхідно не лише атакувати противників, але й своєчасно ухилятися від їхніх ударів. Значна увага приділяється вивченню поведінки ворогів та правильному вибору моменту для атаки.

Особливе місце у грі займають боси. Кожен бос має унікальний набір атак та власні особливості поведінки. Бої з босами є одними з найскладніших елементів ігрового процесу та виступають важливими контрольними точками розвитку персонажа.

Успіх Hollow Knight значною мірою пов’язаний із грамотним поєднанням дослідження світу, бойової системи та поступового розвитку персонажа. Саме ці механіки є основою більшості сучасних ігор жанру Metroidvania.

1.5 Ori and the Blind Forest

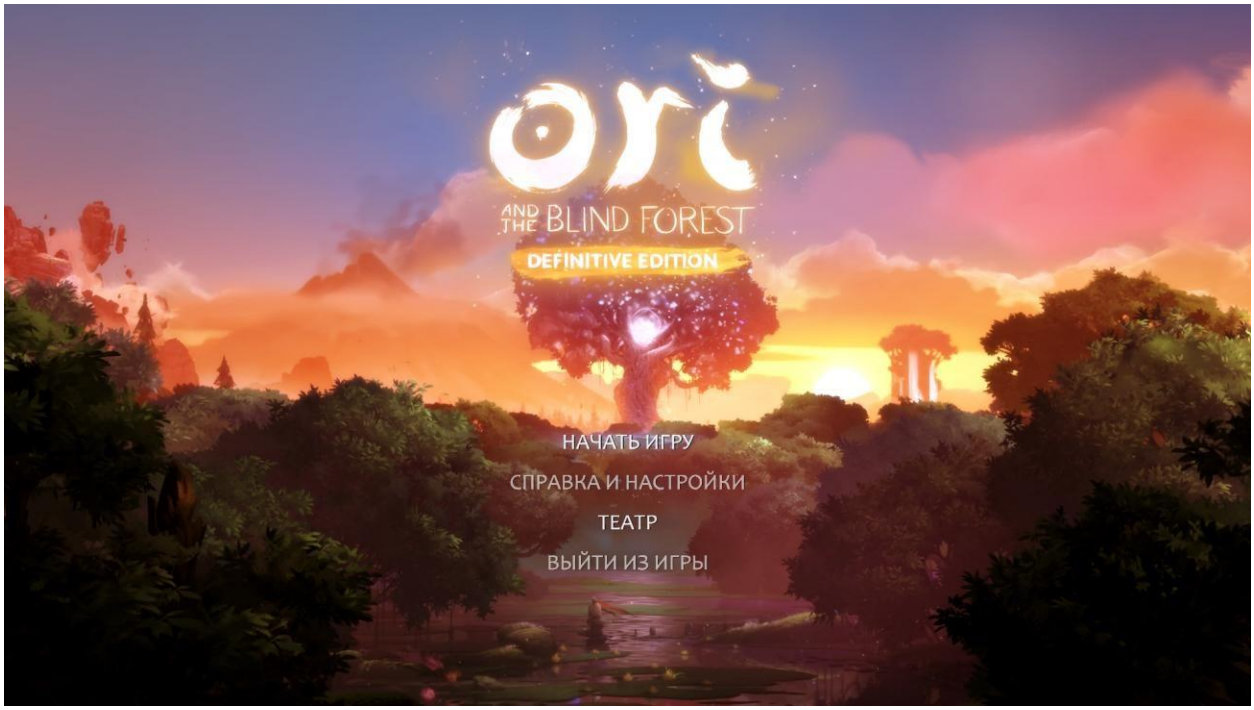


Рисунок 1.6 – скріншот гри Ori and the blind forest DEFIFNITIVE EDITION

Ori and the Blind Forest була розроблена студією Moon Studios та випущена у 2015 році. На відміну від багатьох інших представників жанру, дана гра робить акцент не лише на бойовій системі, але й на високій динаміці переміщення персонажа та художньому оформленні.

Основною особливістю гри є система платформінгу. Персонаж отримує нові здібності, які дозволяють долати складні перешкоди та відкривати нові області світу. Значна частина ігрового процесу пов’язана з точним виконанням стрибків та використанням спеціальних навичок пересування.

Світ гри складається з великої кількості взаємопов’язаних локацій, між якими користувач може вільно переміщуватися після отримання необхідних здібностей. Таким чином реалізується одна з ключових особливостей жанру Metroidvania — поступове відкриття нових територій.

Система розвитку персонажа передбачає отримання нових можливостей, серед яких подвійний стрибок, ривок, прискорене переміщення та інші

спеціальні навички. Кожна нова здатність суттєво впливає на доступність окремих частин ігрового світу.

Важливою особливістю проєкту є художнє оформлення. Гра використовує детально промальовані двовимірні локації, складні анімації та велику кількість візуальних ефектів. Завдяки цьому створюється унікальна атмосфера фантастичного світу.

Ще однією перевагою Ori and the Blind Forest є високий рівень плавності керування персонажем. Усі рухи персонажа виконуються швидко та передбачувано, що особливо важливо для платформних механік.

Проєкт демонструє важливість якісної реалізації системи керування персонажем та механік переміщення для створення захоплюючого ігрового процесу.

1.6 Blasphemous

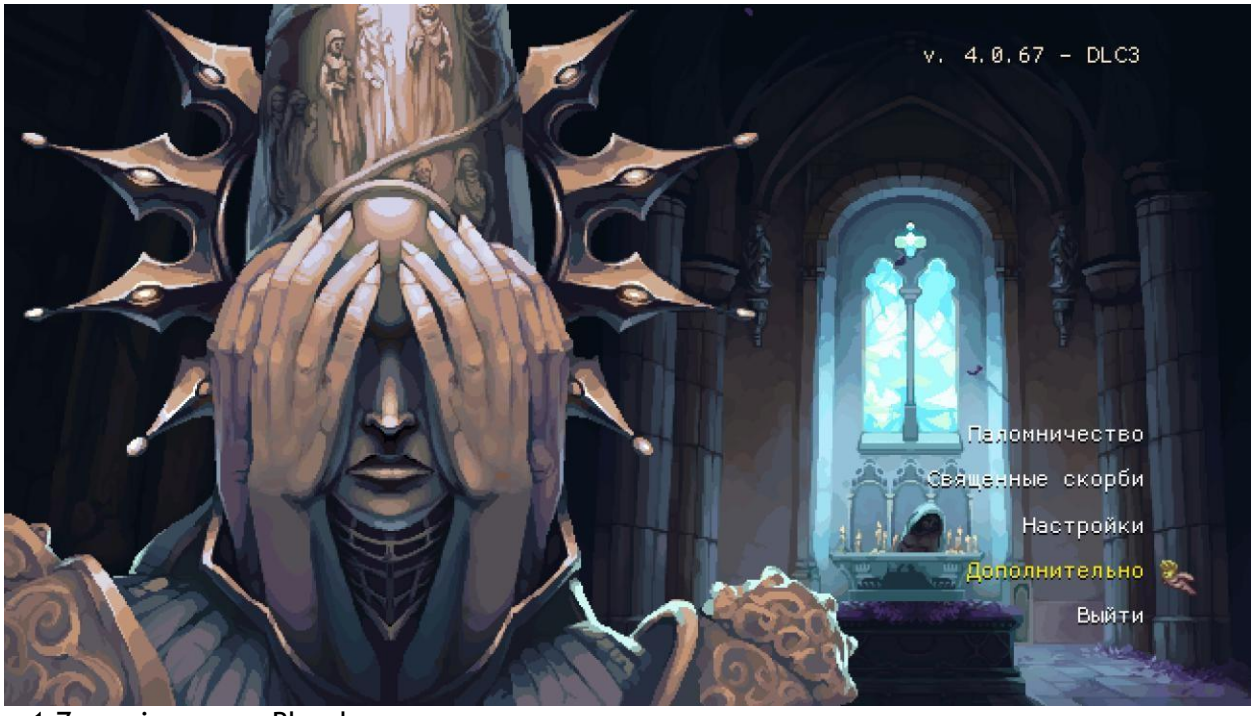


Рисунок 1.7 – скріншот гри Blasphemous

Blasphemous є двовимірною комп’ютерною грою жанру Metroidvania, розробленою іспанською студією The Game Kitchen. Гра була випущена у 2019 році та отримала популярність завдяки поєднанню жанру Metroidvania з елементами Souls-like.

Особливістю гри є похмура атмосфера середньовічного світу, натхненного релігійною символікою та готичною архітектурою. Візуальне оформлення виконане у стилі Pixel Art, що дозволило створити деталізовані локації та персонажів.

На відміну від Ori and the Blind Forest, основний акцент у Blasphemous зроблено на бойову систему. Гравець повинен уважно контролювати свої дії під час бою, вивчати поведінку противників та правильно використовувати доступні можливості персонажа.

Світ гри складається з взаємопов’язаних областей, які поступово відкриваються у процесі проходження. Для доступу до нових локацій необхідно отримувати спеціальні предмети та здібності.

Система противників включає як звичайних ворогів, так і складних босів. Кожен тип противника має власний набір атак та особливості поведінки. Для перемоги над сильними противниками користувач повинен ретельно вивчати їхні атаки та адаптувати власну тактику.

Blasphemous демонструє важливість поєднання атмосферного візуального оформлення, складної бойової системи та продуманого дизайну рівнів.

1.7 Порівняльний аналіз аналогів

На основі проведеного аналізу можна визначити спільні особливості сучасних представників жанру Metroidvania.

До таких особливостей належать:

– наявність великого взаємопов’язаного ігрового світу;

– нелінійна структура проходження;

– поступове відкриття нових здібностей персонажа;

– використання системи блокування доступу до окремих зон;

– наявність бойової системи;

– використання різних типів противників;

– наявність босів;

– система контрольних точок або збереження прогресу;

– дослідження світу та пошук секретів;

– збір колекційних предметів.

Проведений аналіз показує, що сучасні представники жанру Metroidvania поєднують дослідження світу, розвиток персонажа та бойову систему в єдину цілісну структуру. Саме ці механіки забезпечують високу популярність жанру серед користувачів та формують основні вимоги до розробки власного програмного продукту.

На основі розглянутих аналогів можна зробити висновок, що для створення комп’ютерної гри жанру Metroidvania необхідно реалізувати систему керування персонажем, бойову систему, систему анімацій, штучний інтелект противників, систему переходів між локаціями та механізми поступового розвитку персонажа. Дані принципи були враховані під час проєктування та реалізації програмного продукту, що розробляється в рамках даної дипломної роботи.

1.8 Ігрові рушії для розробки двовимірних комп’ютерних ігор

Рисунок 1.8 – логотипи

Розробка сучасних комп’ютерних ігор є складним процесом, який потребує використання спеціалізованих програмних засобів. Для спрощення створення ігрових проєктів використовуються ігрові рушії, що являють собою комплекс програмних інструментів для реалізації графіки, фізики, анімації, звукового супроводу та інших компонентів ігрового процесу.

Ігровий рушій можна визначити як програмну платформу, що забезпечує розробника готовим набором технологій для створення інтерактивних програмних продуктів. Використання рушіїв дозволяє уникнути необхідності створення базових систем з нуля та значно скоротити час розробки.

Сучасний ринок програмного забезпечення пропонує велику кількість ігрових рушіїв, які відрізняються функціональними можливостями, складністю використання, продуктивністю та умовами ліцензування. Найбільш популярними рішеннями для створення двовимірних комп’ютерних ігор є Unity, Unreal Engine та Godot.

1.6 Unity

Unity є одним із найпоширеніших ігрових рушіїв у світі. Його розробка була розпочата компанією Unity Technologies у 2005 році. За роки існування

рушій отримав широке поширення серед незалежних розробників, невеликих студій та великих компаній.

Однією з головних переваг Unity є універсальність. Рушій підтримує створення як двовимірних, так і тривимірних проєктів, а також забезпечує можливість експорту програмних продуктів на різні платформи. До таких платформ належать операційні системи Windows, Linux, macOS, Android, iOS та ігрові консолі.

Архітектура Unity побудована на компонентному підході. Будь-який об’єкт сцени може містити набір компонентів, які визначають його поведінку. Такий підхід значно спрощує розробку та підтримку програмного коду.

Для реалізації ігрової логіки використовується мова програмування C#. Дана мова характеризується високою продуктивністю, простотою синтаксису та великою кількістю доступних навчальних матеріалів.

Unity містить вбудовані засоби для роботи з фізикою, анімаціями, системами частинок, освітленням, аудіосистемами та користувацьким інтерфейсом. Крім того, рушій має власний магазин цифрових ресурсів Asset Store, який містить велику кількість готових моделей, текстур, звукових ефектів та програмних компонентів.

Для розробки двовимірних ігор Unity пропонує спеціалізовані інструменти, серед яких Tilemap, Sprite Renderer, 2D Physics та Animation System. Наявність даних компонентів значно спрощує створення платформерів та ігор жанру Metroidvania.

1.10 Unreal Engine

Unreal Engine є одним із найпотужніших сучасних ігрових рушіїв. Його розробником є компанія Epic Games. Рушій широко використовується при створенні високобюджетних комерційних проєктів.

Основною перевагою Unreal Engine є високий рівень графічних можливостей. Рушій забезпечує підтримку сучасних технологій візуалізації, реалістичного освітлення та складних графічних ефектів.

Для програмування використовується мова C++, яка забезпечує високу продуктивність, але водночас є більш складною для вивчення порівняно з C#. Також рушій підтримує систему візуального програмування Blueprint, яка дозволяє створювати логіку гри без написання програмного коду.

Попри значні переваги, Unreal Engine частіше використовується для створення тривимірних проєктів. Для невеликих двовимірних ігор можливості рушія часто є надлишковими, що призводить до ускладнення процесу розробки.

Ще одним недоліком є підвищені вимоги до апаратного забезпечення під час роботи редактора. Це може створювати додаткові труднощі для незалежних розробників та студентських проєктів.

1.11 Godot

Godot є безкоштовним ігровим рушієм з відкритим вихідним кодом. Проєкт активно розвивається міжнародною спільнотою розробників та поступово набирає популярність серед незалежних студій.

Однією з головних переваг Godot є його безкоштовність та відкритість. Розробники можуть використовувати рушій без сплати ліцензійних платежів незалежно від комерційного успіху проєкту.

Рушій має власну систему сцен та компонентів, яка певною мірою нагадує архітектуру Unity. Для програмування використовується мова GDScript, синтаксис якої схожий на Python. Також підтримуються C# та C++.

Godot забезпечує повноцінну підтримку двовимірної графіки та містить необхідні інструменти для створення платформерів, рольових ігор та інших жанрів.

Разом з тим, Godot поступається Unity за кількістю навчальних матеріалів, розміром спільноти та доступністю готових рішень. Для

студентських проєктів це може ускладнювати процес пошуку інформації та вирішення технічних проблем.

1.12 Порівняння сучасних ігрових рушіїв

Для вибору програмної платформи було проведено порівняння найбільш популярних рушіїв за основними характеристиками.

Основними критеріями оцінювання стали:

– підтримка двовимірної графіки;

– простота освоєння;

– продуктивність;

– наявність документації;

– кількість навчальних матеріалів;

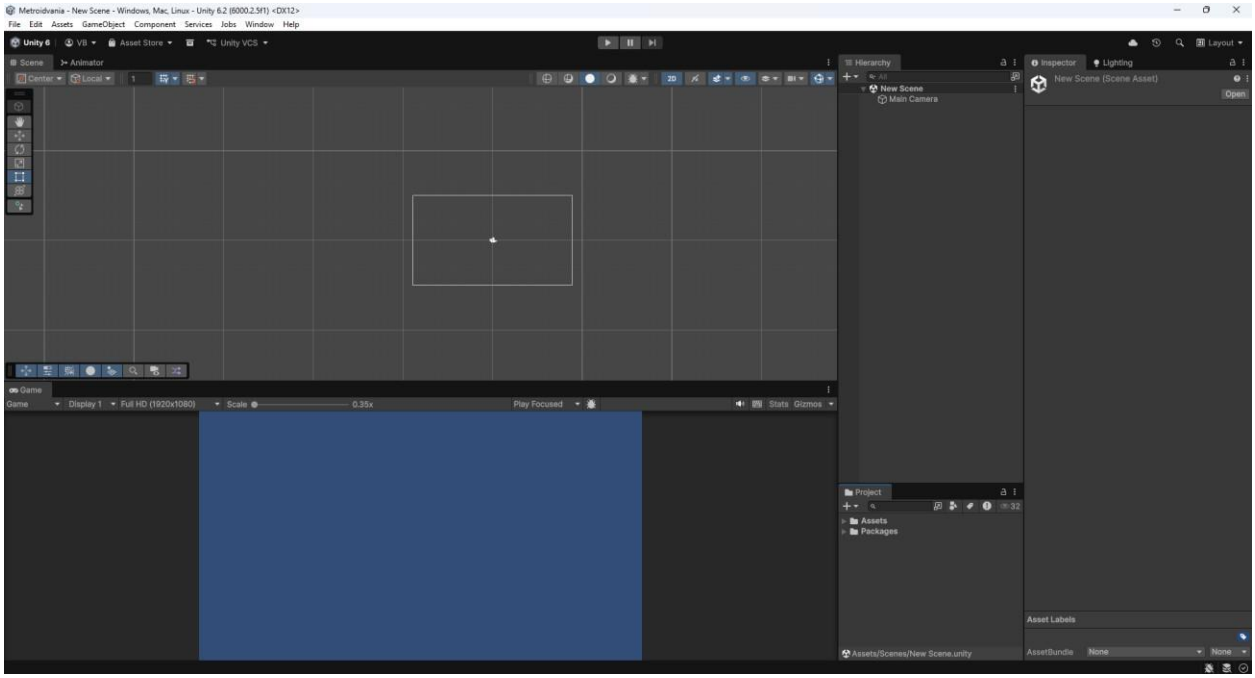
– підтримка мов програмування;

– можливість створення незалежних проєктів.

Проведений аналіз показав, що всі розглянуті рушії можуть бути використані для створення ігор жанру Metroidvania. Проте для навчального проєкту найбільш доцільним є використання Unity.

Рушій забезпечує оптимальне поєднання функціональних можливостей, простоти освоєння та кількості доступних навчальних ресурсів. Крім того, Unity має вбудовані інструменти для роботи з двовимірною графікою, що є важливим фактором при створенні комп’ютерної гри жанру Metroidvania.

1.13 Обґрунтування вибору рушія Unity



Одним із найважливіших етапів проєктування програмного продукту є вибір технологічної платформи, на основі якої буде здійснюватися розробка. Правильно обраний інструментарій дозволяє скоротити час створення програмного забезпечення, підвищити якість кінцевого результату та спростити подальшу підтримку проєкту.

Після аналізу сучасних ігрових рушіїв для реалізації комп’ютерної гри жанру Metroidvania було обрано Unity. Даний вибір обумовлений низкою технічних та практичних переваг.

Перш за все, Unity має повноцінну підтримку двовимірної графіки. Рушій містить спеціалізовані інструменти для роботи зі спрайтами, тайловими картами, анімаціями та фізичними взаємодіями. Це дозволяє реалізовувати основні механіки платформерів без використання сторонніх програмних рішень.

Важливою перевагою Unity є використання мови програмування C#. Дана мова є однією з найбільш популярних у сфері розробки програмного забезпечення та характеризується зрозумілим синтаксисом, високою продуктивністю і великою кількістю навчальних матеріалів.

Для реалізації механік гри рушій надає готові засоби роботи з фізикою, анімацією та користувацьким інтерфейсом. Завдяки цьому розробник може зосередитися безпосередньо на створенні ігрового процесу, а не на реалізації базових систем низького рівня.

Ще однією важливою перевагою є компонентно-орієнтована архітектура Unity. Кожен об’єкт сцени може містити набір незалежних компонентів, що відповідають за окремі функції. Такий підхід спрощує структуру проєкту та підвищує зручність супроводу програмного коду.

Для реалізації гри жанру Metroidvania особливе значення мають системи керування персонажем, бойової взаємодії, штучного інтелекту противників та анімацій. Unity містить усі необхідні інструменти для реалізації зазначених механік, що дозволяє створювати складні інтерактивні системи без використання додаткових програмних платформ.

Суттєвою перевагою є також велика кількість доступних навчальних матеріалів, офіційної документації та прикладів програмного коду. Це значно спрощує процес розробки та дозволяє швидко знаходити рішення технічних проблем.

Важливим фактором є підтримка сучасних версій операційної системи Windows, під яку орієнтований програмний продукт, що розробляється в рамках даної дипломної роботи.

Таким чином, проведений аналіз показав, що Unity найбільш повно відповідає вимогам розробки двовимірної комп’ютерної гри жанру Metroidvania. Наявність вбудованих інструментів для роботи з графікою, фізикою, анімаціями та користувацьким інтерфейсом, а також використання мови програмування C# стали основними причинами вибору даного рушія для реалізації програмного продукту.

ОСНОВНИЙ РОЗДІЛ

2.1 Загальна концепція програмного продукту

Метою даної дипломної роботи є розробка програмного прототипу комп'ютерної гри у жанрі Metroidvania з використанням рушія Unity та мови програмування C#.

Програмний продукт являє собою двовимірну комп'ютерну гру, виконану в стилі Pixel Art із середньовічним візуальним оформленням. Гра розрахована на запуск під операційною системою Windows та орієнтована на використання клавіатури як основного засобу керування персонажем.

Основною особливістю жанру Metroidvania є поєднання платформних механік, бойової системи та дослідження ігрового світу. Під час проєктування програмного продукту особлива увага приділялася реалізації базових механік, характерних для даного жанру.

У створеному програмному продукті користувач керує ігровим персонажем, який може переміщуватися ігровими локаціями, виконувати стрибки, атакувати противників, ухилятися від атак за допомогою перекату, збирати монети та взаємодіяти з елементами навколишнього середовища.

Ігровий процес побудований навколо дослідження локацій та подолання противників. Для створення динамічного ігрового процесу реалізована система ближнього бою, яка дозволяє виконувати серію атак та завдавати шкоди противникам.

У програмному продукті також реалізовано систему здоров'я персонажа. Гравець має обмежений запас життів, який зменшується при отриманні пошкоджень від противників. У разі втрати всіх одиниць здоров'я відображається екран завершення гри.

Для забезпечення більшої різноманітності ігрового процесу реалізована система здібностей персонажа. Під час проходження користувач може

отримувати нові можливості, які розширюють набір доступних дій та відкривають додаткові елементи взаємодії з ігровим світом.

Окрему увагу було приділено створенню противників. Для управління їх поведінкою використовується система станів, яка дозволяє реалізувати патрулювання території, переслідування персонажа, атаку та стан смерті.

Для організації переходів між ігровими локаціями реалізовано механізм зміни сцен із використанням плавного затемнення екрана. Це дозволяє створити більш комфортний перехід між різними областями ігрового світу.

Таким чином, створений програмний продукт містить основні механіки, характерні для жанру Metroidvania, та може виступати основою для подальшого розвитку повноцінної комп’ютерної гри.

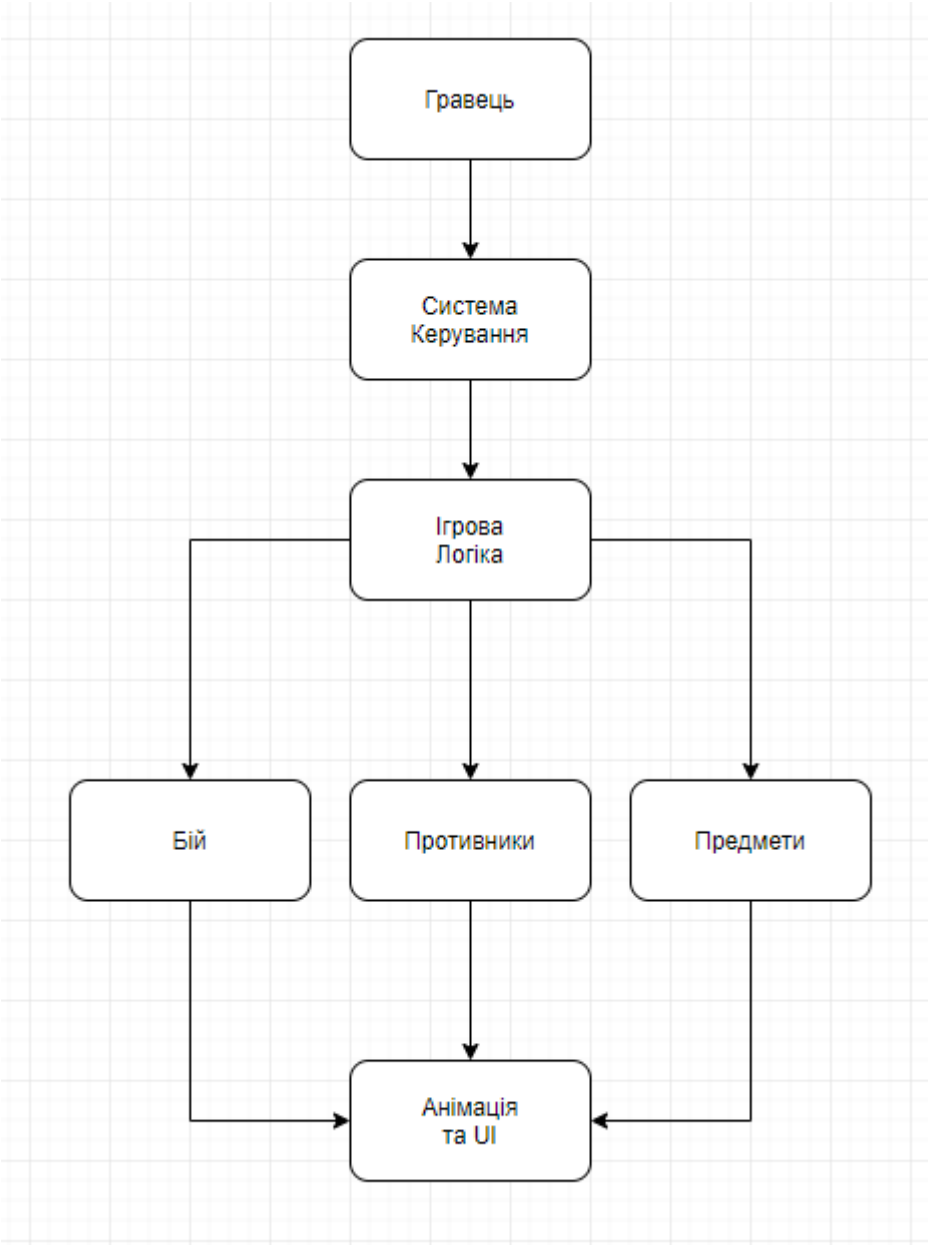


Рисунок 2.1 – Загальна структура програмного продукту

2.2 Архітектура програмного продукту

Архітектура програмного продукту є одним із найважливіших етапів проєктування комп'ютерної гри. Від правильності організації структури програмного коду залежить зручність подальшої розробки, масштабованість проєкту та простота внесення змін до вже реалізованого функціоналу.

Під час розробки комп'ютерної гри було використано компонентно-орієнтований підхід, який є стандартним для рушія Unity. Відповідно до даного

підходу кожен ігровий об’єкт містить набір компонентів, що відповідають за окремі функції. Подібна організація дозволяє розділити логіку програми на незалежні модулі та значно спрощує супровід програмного продукту.

У розробленій грі архітектура побудована навколо декількох основних підсистем:

– система керування персонажем;

– система бойової взаємодії;

– система здоров’я;

– система анімацій;

– система штучного інтелекту противників;

– система збору предметів;

– система контрольних точок;

– система переходів між сценами;

– система користувацького інтерфейсу.

Кожна із зазначених підсистем реалізована за допомогою окремих програмних компонентів, які взаємодіють між собою через публічні методи та властивості.

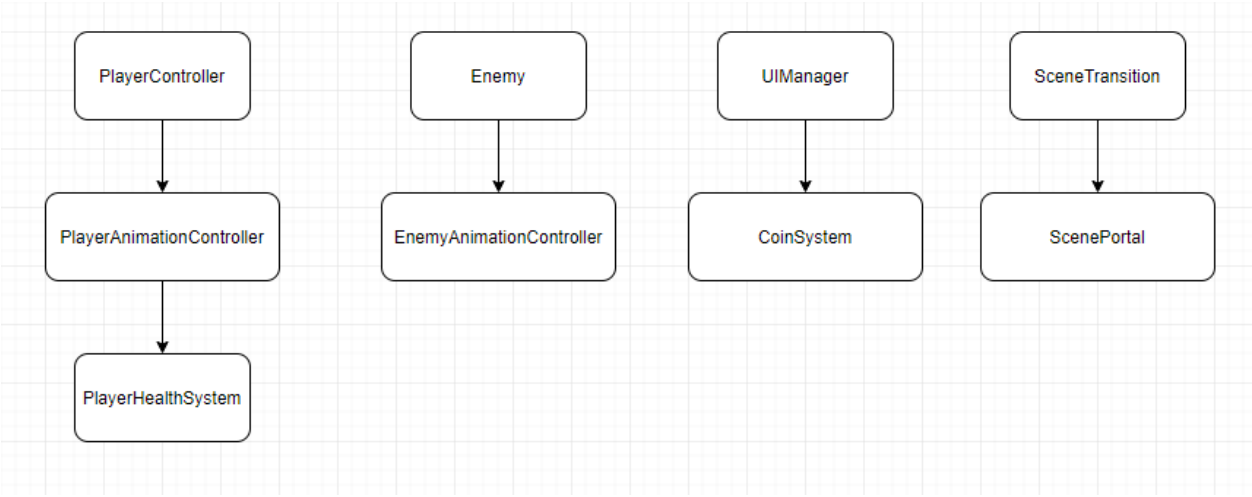


Рисунок 2.2 – Загальна архітектура програмного продукту

Архітектура керування персонажем

Центральним елементом програмного продукту є клас PlayerController. Даний компонент відповідає за обробку введення користувача та керування основними діями персонажа.

У межах даного класу реалізовано:

– горизонтальне переміщення персонажа;

– систему стрибків;

– систему перекату;

– механіку атак;

– отримання пошкоджень;

– взаємодію з іншими об'єктами сцени.

Для забезпечення коректної роботи фізичних взаємодій використовується компонент Rigidbody2D. Саме він відповідає за переміщення персонажа, вплив сили тяжіння та взаємодію з колайдерами ігрового світу.

Для обробки введення використовується Unity Input System. Завдяки цьому всі дії користувача централізовано обробляються в одному місці та можуть бути легко змінені при необхідності.

Архітектурне рішення із виділенням окремого контролера персонажа дозволяє відокремити логіку керування від інших частин програми.

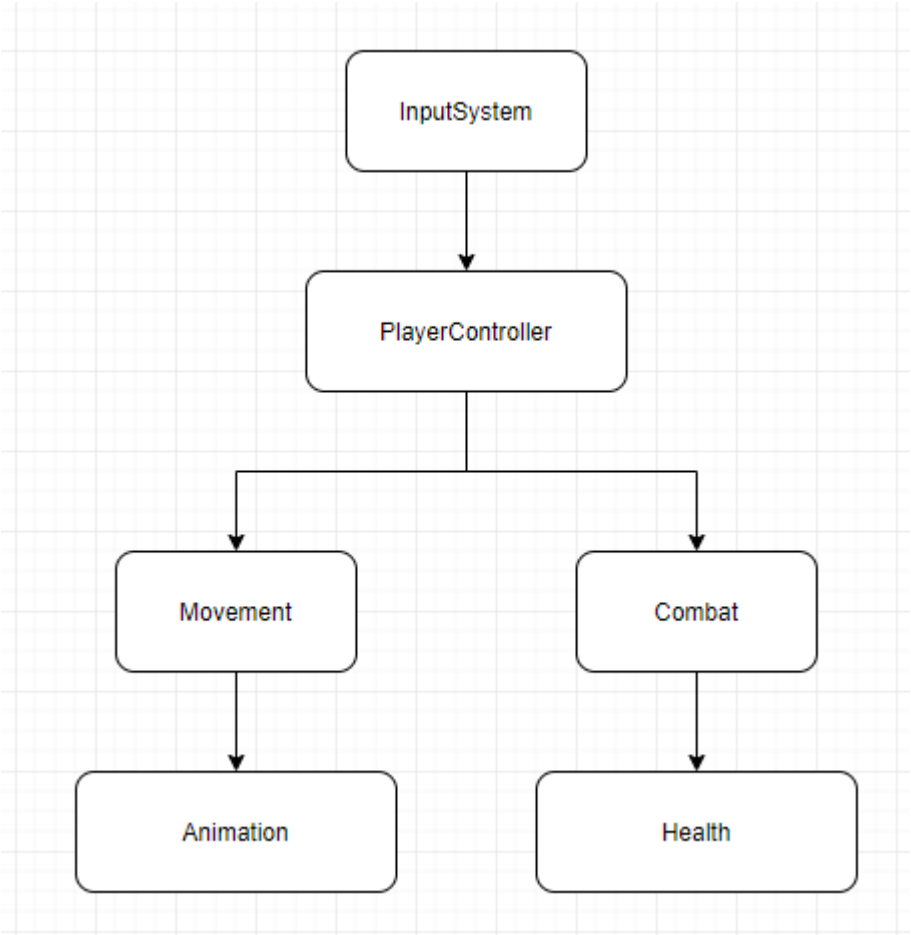


Рисунок 2.3 – Взаємодія компонентів персонажа

Архітектура системи анімацій

Для керування анімаціями використовується окремий компонент PlayerAnimationController.

Подібний підхід дозволяє відокремити логіку відображення анімацій від логіки керування персонажем.

Система анімацій відповідає за:

– анімацію руху;

– анімацію стрибка;

– анімацію перекату;

– анімацію атак;

– анімацію отримання пошкодження;

– анімацію смерті.

Перемикання між анімаційними станами здійснюється через параметри Animator Controller. Зміна параметрів відбувається автоматично залежно від поточного стану персонажа.

Використання окремого контролера анімацій дозволяє уникнути надмірного ускладнення основного класу керування персонажем.

Архітектура системи здоров'я

Для реалізації механіки отримання пошкоджень використовується клас PlayerHealthSystem.

Основними функціями даної підсистеми є:

– зберігання кількості життів персонажа;

– оновлення інтерфейсу здоров'я;

– відтворення анімації отримання шкоди;

– визначення моменту смерті персонажа.

Під час отримання пошкодження система автоматично оновлює стан індикаторів здоров'я та виконує відповідні анімації.

Після втрати останньої одиниці здоров'я персонаж переходить у стан смерті та запускається екран завершення гри.

Архітектура бойової системи

Бойова система побудована навколо взаємодії класів PlayerController та AttackCollider.

AttackCollider відповідає за визначення противників, які знаходяться в зоні ураження персонажа.

Під час виконання атаки система перевіряє наявність противника в області ураження та завдає відповідне пошкодження.

Для підвищення динамічності ігрового процесу реалізовано систему комбінованих атак, яка дозволяє виконувати послідовність ударів.

Завдяки розділенню логіки атак та визначення зіткнень вдається спростити структуру програмного коду та полегшити його подальше розширення.

Архітектура системи противників

Для реалізації поведінки противників використовується клас Enemy.

В основі роботи противника лежить кінцевий автомат станів.

У поточній реалізації використовуються такі стани:

– Patrol;

– Chase;

– Attack;

– Dead.

Стан Patrol використовується для патрулювання території.

Стан Chase активується після виявлення персонажа та запускає механізм переслідування.

Стан Attack відповідає за виконання атакуючих дій.

Стан Dead використовується після загибелі противника.

Подібний підхід є одним із найпоширеніших способів реалізації штучного інтелекту в комп’ютерних іграх завдяки простоті реалізації та високій наочності.

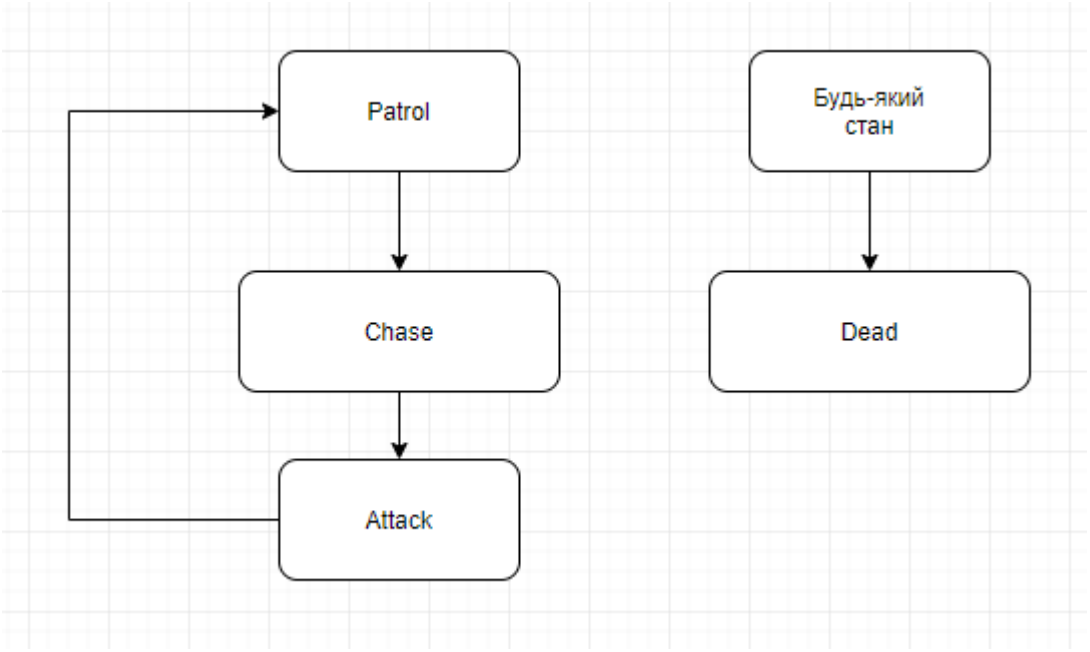


Рисунок 2.4 – Схема станів противника

Архітектура системи переходів між сценами

Для організації переходів між ігровими локаціями використовується система SceneTransitionManager.

Основними функціями підсистеми є:

– завантаження нових сцен;

– затемнення екрана;

– плавна поява нової сцени;

– позиціонування персонажа після переходу.

Використання окремого менеджера переходів дозволяє централізувати логіку роботи зі сценами та уникнути дублювання програмного коду.

Архітектура користувацького інтерфейсу

Користувацький інтерфейс реалізований за допомогою системи Canvas, що входить до складу рушія Unity.

Основними елементами інтерфейсу є:

– індикатори здоров’я;

– лічильник монет;

– екран завершення гри;

– службові елементи інтерфейсу.

Для керування елементами інтерфейсу використовується клас UIManager.

Усі зміни стану персонажа автоматично відображаються в інтерфейсі користувача, що забезпечує зручність взаємодії з програмним продуктом.

Таким чином, архітектура розробленої гри побудована на принципах модульності та компонентного підходу. Подібне рішення дозволяє забезпечити простоту підтримки програмного коду, можливість подальшого розширення функціоналу та зручність реалізації нових механік.

2.3 Структура проєкту та організація ресурсів у середовищі Unity

Одним із важливих етапів розробки програмного забезпечення є правильна організація структури проєкту. Грамотно побудована структура дозволяє спростити процес розробки, прискорити пошук необхідних ресурсів та полегшити подальше супроводження програмного продукту.

Під час створення комп’ютерної гри у жанрі Metroidvania було використано стандартну структуру проєкту Unity з додатковим логічним розподілом ресурсів за категоріями. Подібний підхід дозволяє підтримувати порядок у проєкті навіть при збільшенні кількості ігрового контенту.

Усі ресурси програмного продукту розміщуються в каталозі Assets, який є основним сховищем даних проєкту Unity.

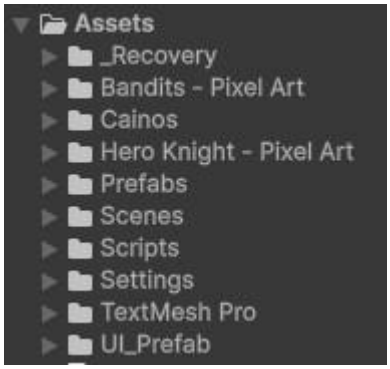


Рисунок 2.5 – Загальна структура каталога Assets

Організація програмного коду

Для зберігання програмного коду використовується каталог Scripts.

У даному каталозі розміщуються всі програмні компоненти, що відповідають за реалізацію логіки гри.

До основних програмних модулів належать:

– PlayerController;

– PlayerAnimationController;

– PlayerHealthSystem;

– Enemy;

– EnemyAnimationController;

– AttackCollider;

– Coin;

– CheckPoint;

– UIManager;

– ScenePortal;

– SceneTransitionManager.

Кожен із зазначених класів відповідає за окрему підсистему програмного продукту.

Подібний підхід дозволяє мінімізувати залежності між модулями та спрощує подальше розширення функціоналу гри.

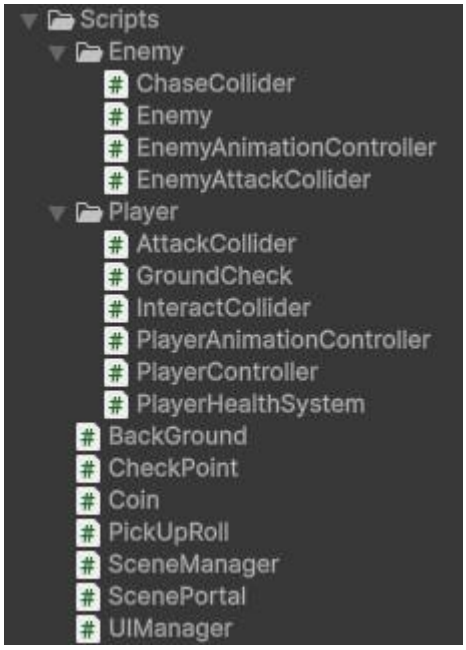


Рисунок 2.6 – Організація програмних компонентів

Організація графічних ресурсів

Для відображення об'єктів ігрового світу використовуються графічні ресурси, розташовані в каталозі Sprites.

До графічних ресурсів належать:

– спрайти персонажа;

– спрайти противників;

– елементи оточення;

– тайли рівнів;

– інтерфейсні елементи;

– колекційні предмети.

Оскільки програмний продукт створюється у стилі Pixel Art, усі графічні ресурси представлені у вигляді двовимірних растрових зображень.

Використання стилістики Pixel Art дозволяє зменшити складність створення графіки та забезпечує характерний зовнішній вигляд гри.

Організація анімацій

Для створення анімацій використовується каталог Animations.

У ньому розміщуються:

– Animation Clips;

– Animator Controller;

– допоміжні ресурси анімаційної системи.

Для персонажа та противників створюються окремі контролери анімацій.

Завдяки цьому кожен тип об'єктів може використовувати власний набір станів та переходів між ними.

До основних анімацій персонажа належать:

– Idle;

– Run;

– Jump;

– Roll;

– Attack;

– Hurt;

– Death.

Для противників використовуються анімації:

– Idle;

– Patrol;

– Attack;

– Death.

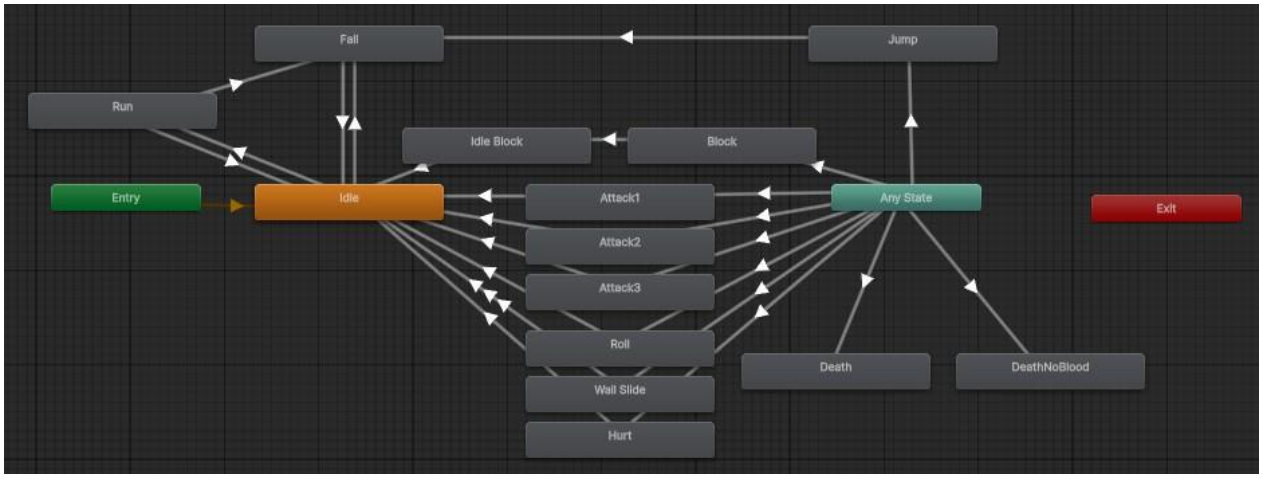


Рисунок 2.7 – Структура Animator Controller персонажа

Організація сцен

Для побудови ігрового світу використовується система сцен Unity.

Кожна сцена являє собою окрему частину ігрового світу та містить набір об'єктів, необхідних для функціонування відповідної локації.

Сцени розташовуються у каталозі Scenes.

Основними елементами сцени є:

– персонаж;

– противники;

– платформи;

– елементи інтерфейсу;

– контрольні точки;

– колекційні предмети;

– переходи між сценами.

Використання декількох сцен дозволяє створювати великі ігрові локації без перевантаження оперативної пам’яті.

Організація префабів

Для повторного використання ігрових об'єктів застосовуються префаби.

Префаб являє собою шаблон об'єкта, який може багаторазово використовуватися в різних сценах.

У програмному продукті префаби використовуються для:

– противників;

– монет;

– контрольних точок;

– елементів оточення;

– переходів між локаціями.

Основною перевагою використання префабів є можливість одночасного оновлення всіх екземплярів об'єкта після зміни вихідного шаблону.

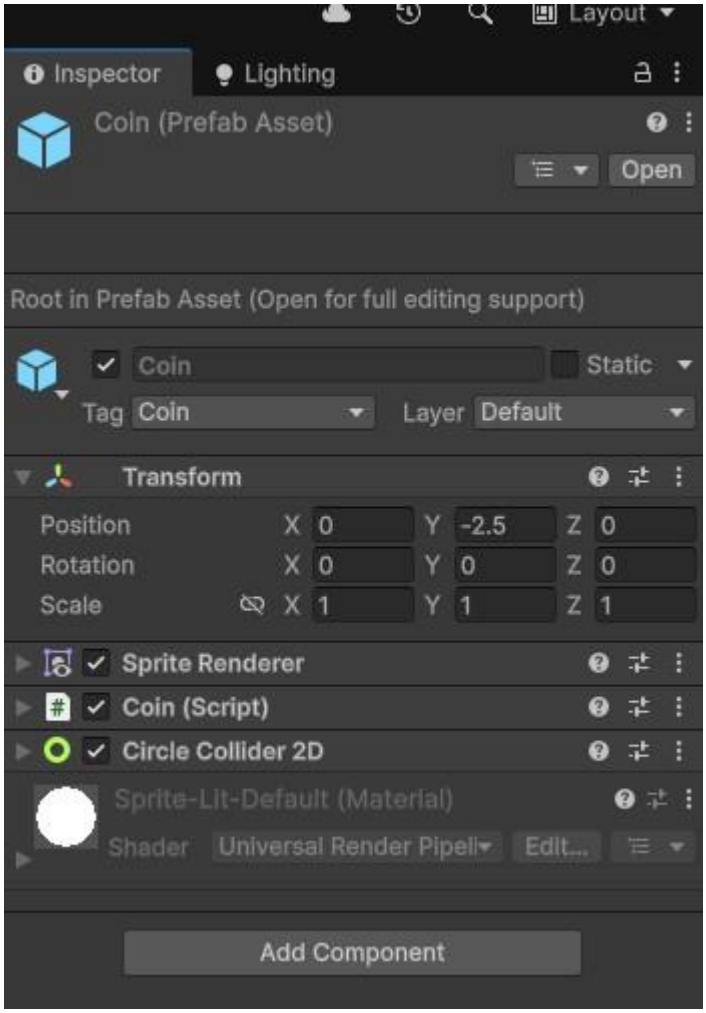


Рисунок 2.8 – Приклад префаба в середовищі Unity

Організація користувацького інтерфейсу

Для реалізації користувацького інтерфейсу використовується система Canvas.

Інтерфейс містить елементи, які відображають поточний стан гри та надають користувачу необхідну інформацію.

До основних елементів інтерфейсу належать:

– індикатори здоров’я;

– лічильник монет;

– екран завершення гри;

– елементи переходів між сценами.

Усі елементи інтерфейсу організовані таким чином, щоб забезпечити зручне сприйняття інформації під час ігрового процесу.

Організація ресурсів проєкту

Використання структурованого підходу до організації ресурсів дозволяє значно підвищити ефективність розробки.

Розділення ресурсів на окремі категорії забезпечує:

– швидкий доступ до необхідних файлів;

– спрощення супроводу проєкту;

– зменшення кількості помилок під час розробки;

– можливість подальшого масштабування програмного продукту.

Таким чином, структура проєкту була організована відповідно до рекомендацій Unity та принципів компонентно-орієнтованого програмування. Подібний підхід забезпечує зручність роботи з програмним кодом, графічними ресурсами та іншими елементами комп’ютерної гри.

2.4 Реалізація системи керування персонажем

Однією з найважливіших складових будь-якої комп’ютерної гри жанру Metroidvania є система керування персонажем. Саме через неї

користувач взаємодіє з ігровим світом, виконує переміщення, долає перешкоди та бере участь у бойових зіткненнях.

Під час розробки програмного продукту було реалізовано систему керування персонажем, яка забезпечує виконання основних дій, характерних для жанру Metroidvania. До таких дій належать переміщення персонажа, стрибки, атаки, перекат та взаємодія з навколишнім середовищем.

Основним компонентом підсистеми є клас PlayerController, який відповідає за обробку введення користувача та керування всіма базовими механіками персонажа.

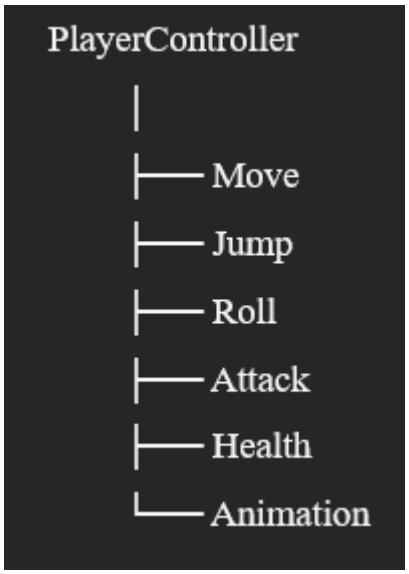


Рисунок 2.9 – Структура керування персонажем

Реалізація обробки введення користувача

Для обробки дій користувача використовується система Unity Input System.

Використання даної технології дозволяє централізовано керувати всіма діями персонажа та спрощує налаштування клавіш керування.

У межах проєкту реалізовано такі дії:

– переміщення персонажа;

– стрибок;

– атака;

– перекат.

Після запуску гри система автоматично створює набір дій користувача та активує їх для подальшого використання.

Подібний підхід забезпечує зручність розширення функціоналу та дозволяє легко додавати нові механіки без значних змін у структурі програмного коду.

Реалізація горизонтального переміщення

Переміщення персонажа є базовою механікою ігрового процесу.

Для реалізації руху використовується компонент Rigidbody2D, який входить до складу фізичної системи Unity.

Під час натискання клавіш руху система зчитує напрямок переміщення та змінює швидкість об'єкта по горизонтальній осі.

Швидкість переміщення задається окремим параметром, що дозволяє легко налаштовувати характеристики персонажа без зміни основного алгоритму роботи.

Використання фізичного компонента забезпечує коректну взаємодію з платформами, колайдерами та іншими об'єктами ігрового світу.

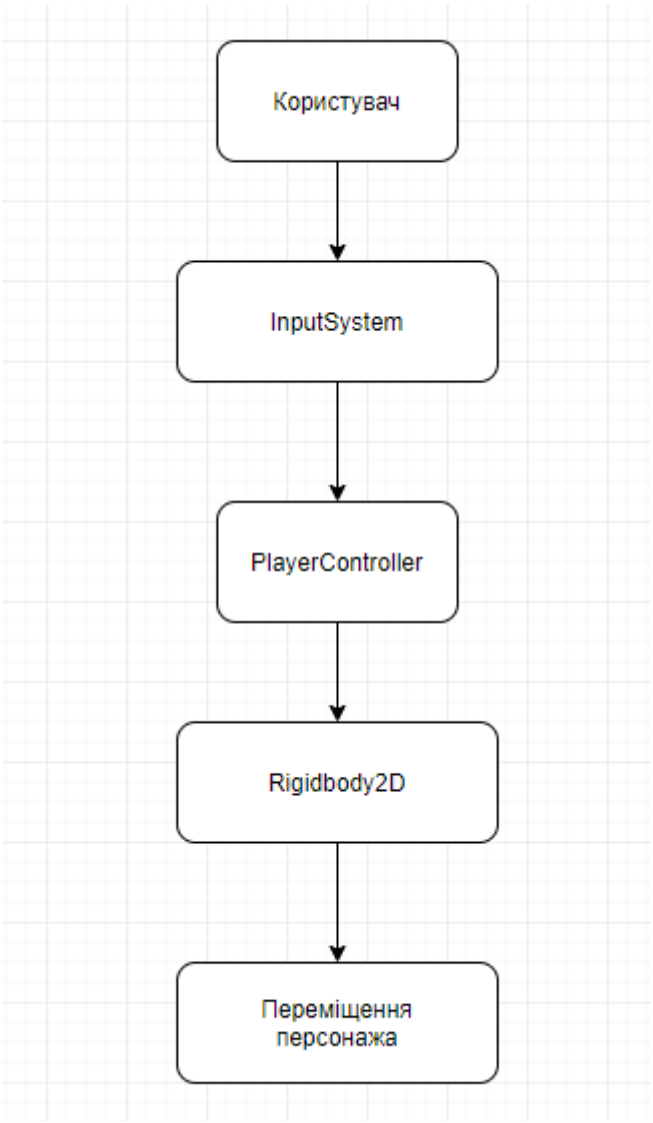


Рисунок 2.10 – Схема роботи системи переміщення

Реалізація системи стрибків

Стрибки є однією з основних механік платформних ігор.

Для реалізації даної можливості використовується взаємодія між класом PlayerController та допоміжним компонентом GroundCheck.

Перед виконанням стрибка система перевіряє наявність контакту персонажа із поверхнею.

У випадку підтвердження контакту персонажу надається вертикальний імпульс, який забезпечує виконання стрибка.

Для визначення факту знаходження персонажа на землі використовується окремий тригерний колайдер, розташований у нижній частині об'єкта.

Подібний підхід дозволяє надійно визначати момент торкання поверхні та запобігати виникненню помилок під час виконання стрибків.

Реалізація стрибка зі змінною висотою

Для підвищення якості керування була реалізована система стрибків зі змінною висотою.

На відміну від звичайного платформного стрибка, дана механіка дозволяє користувачу впливати на висоту підйому персонажа.

При короткому натисканні кнопки персонаж виконує невеликий стрибок.

При утриманні кнопки висота стрибка збільшується до встановленого значення.

Подібне рішення робить керування більш точним та дозволяє користувачу краще контролювати переміщення персонажа під час проходження складних ділянок рівня.

Реалізація розвороту персонажа

Для забезпечення правильного відображення напрямку руху реалізовано систему автоматичного розвороту персонажа.

Під час зміни напрямку переміщення виконується зміна масштабу об'єкта по горизонтальній осі.

Таким чином персонаж завжди повернутий у напрямку свого руху.

Даний підхід є одним із найбільш поширених способів реалізації розвороту двовимірних персонажів та характеризується високою продуктивністю.

Реалізація перекату

Для підвищення динамічності ігрового процесу в програмному продукті реалізована механіка перекату.

Перекат дозволяє персонажу швидко переміщатися на невелику відстань та уникати атак противників.

Під час виконання перекату персонаж отримує короткочасне прискорення в напрямку руху.

У процесі виконання даної дії тимчасово блокується можливість повторного використання перекату до завершення поточної анімації.

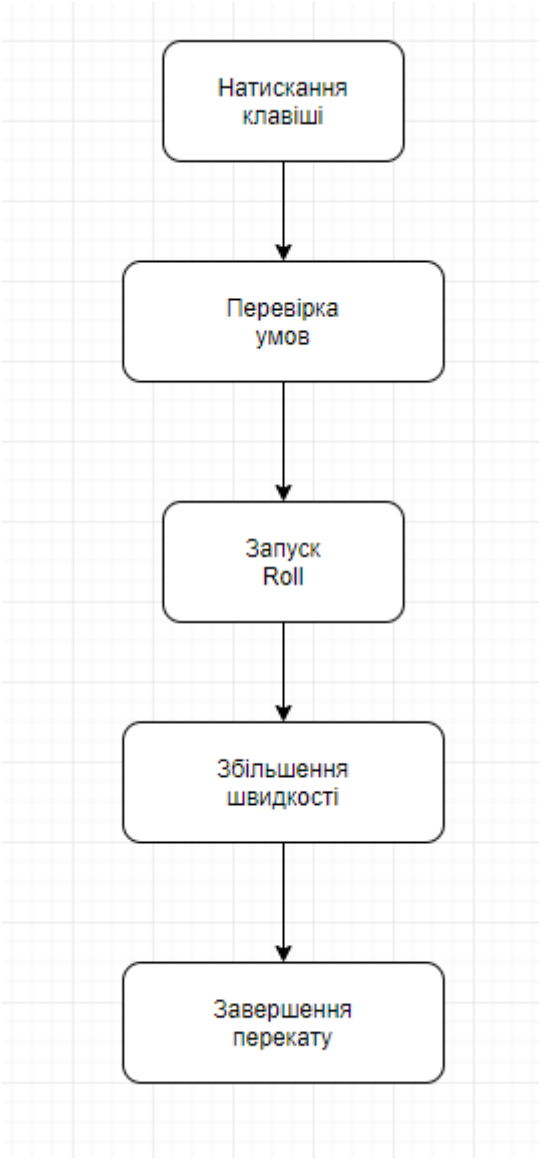


Рисунок 2.11 – Принцип роботи механіки перекату

Реалізація перекату дозволяє урізноманітнити бойову систему та зробити взаємодію з противниками більш динамічною.

Реалізація системи отримання здібностей

Однією з характерних особливостей жанру Metroidvania є поступове відкриття нових можливостей персонажа.

У поточній версії програмного продукту реалізовано механіку отримання здібності перекату.

Для цього використовується спеціальний об'єкт, взаємодія з яким активує відповідну можливість персонажа.

Після отримання здібності вона стає доступною для подальшого використання протягом усього проходження гри.

Подібна система створює основу для реалізації додаткових механік прогресії персонажа та відповідає принципам побудови ігор жанру Metroidvania.

Реалізація взаємодії з фізичною системою

Для забезпечення коректної роботи всіх механік персонажа використовується фізична система Unity.

Основним компонентом є Rigidbody2D, який відповідає за:

– переміщення персонажа;

– вплив сили тяжіння;

– зіткнення з платформами;

– взаємодію з противниками;

– обробку фізичних об'єктів сцени.

Використання стандартної фізичної системи Unity дозволяє уникнути необхідності створення власних алгоритмів обробки зіткнень та значно скорочує час розробки.

Висновки до підрозділу

У результаті реалізації системи керування персонажем було створено комплекс механік, необхідних для функціонування комп’ютерної гри жанру Metroidvania.

Реалізовано переміщення персонажа, систему стрибків зі змінною висотою, автоматичний розворот спрайта, механіку перекату та систему отримання нових здібностей.

Створена система забезпечує комфортне керування персонажем та формує основу для реалізації інших компонентів ігрового процесу.

2.5 Реалізація бойової системи

Бойова система є одним із ключових елементів комп’ютерних ігор жанру Metroidvania. Саме вона забезпечує взаємодію персонажа з противниками та значною мірою впливає на динаміку ігрового процесу.

Під час розробки програмного продукту було реалізовано систему ближнього бою, яка дозволяє персонажу атакувати противників, завдавати їм пошкодження та знищувати їх. Крім того, реалізовано механізми отримання шкоди від ворогів та систему смерті персонажа.

Основними компонентами бойової системи є:

– PlayerController;

– AttackCollider;

– Enemy;

– EnemyAttackCollider;

– PlayerHealthSystem.

Взаємодія зазначених компонентів забезпечує повний цикл бойового процесу.

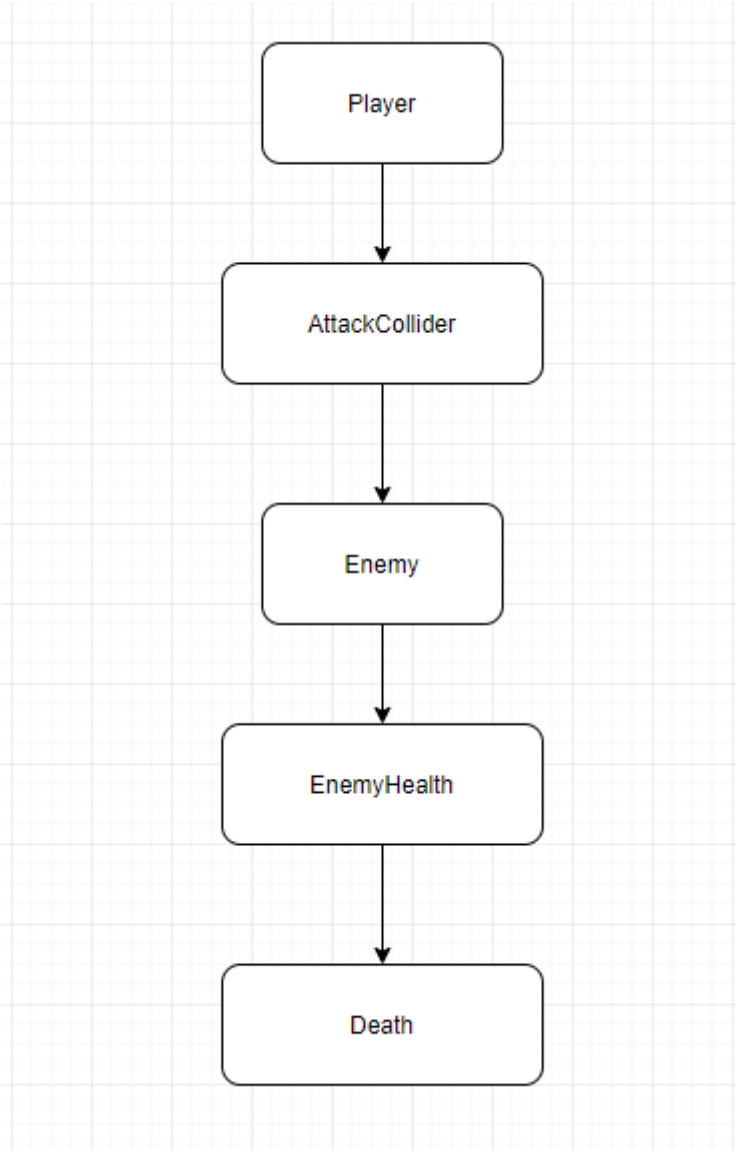


Рисунок 2.12 – Загальна структура бойової системи

Реалізація атаки персонажа

Основним способом взаємодії з противниками є використання атак ближнього бою.

Під час натискання відповідної клавіші користувач активує анімацію атаки персонажа. Одночасно запускається перевірка наявності противників у зоні ураження.

Для визначення цілей використовується спеціальний колайдер атаки, розташований перед персонажем.

Якщо в межах області атаки знаходиться противник, система передає інформацію про завдання пошкодження відповідному об'єкту.

Подібний підхід дозволяє розділити логіку відображення анімацій та логіку визначення зіткнень.

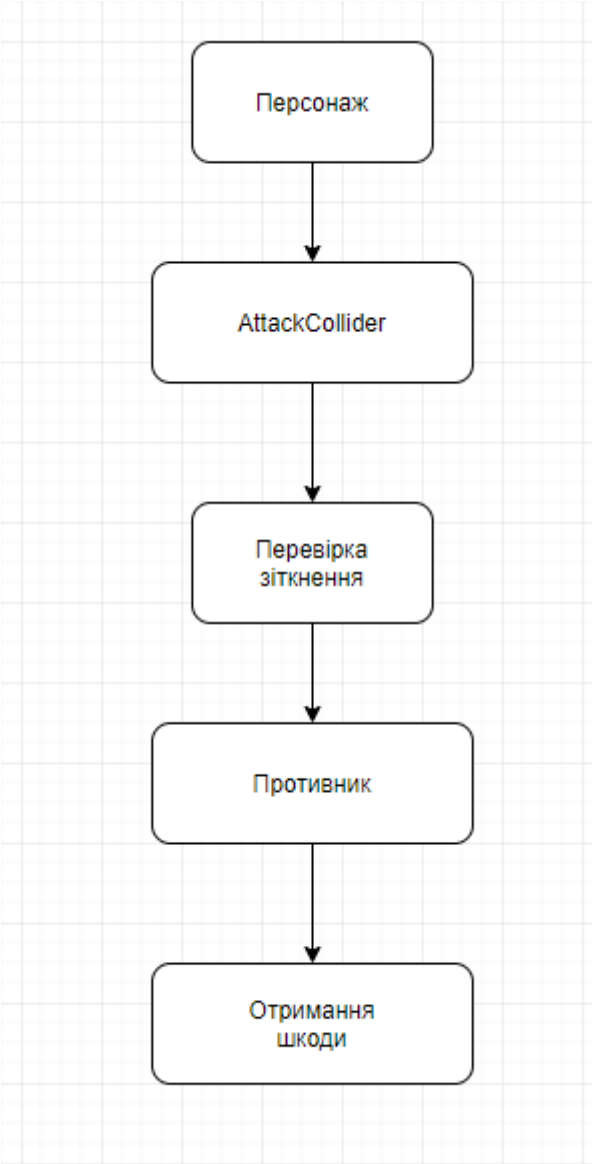


Рисунок 2.13 – Принцип визначення попадання атаки

Реалізація системи комбо-ударів

Для підвищення динамічності бойової системи реалізовано механіку комбінацій атак.

Після виконання першої атаки користувач має короткий проміжок часу для активації другої атаки.

Якщо протягом встановленого часу буде повторно натиснута кнопка атаки, запускається друга анімація удару.

У протилежному випадку комбінація завершується після виконання першої атаки.

Подібна механіка дозволяє зробити бойову систему більш цікавою та створює відчуття плавності бойових взаємодій.

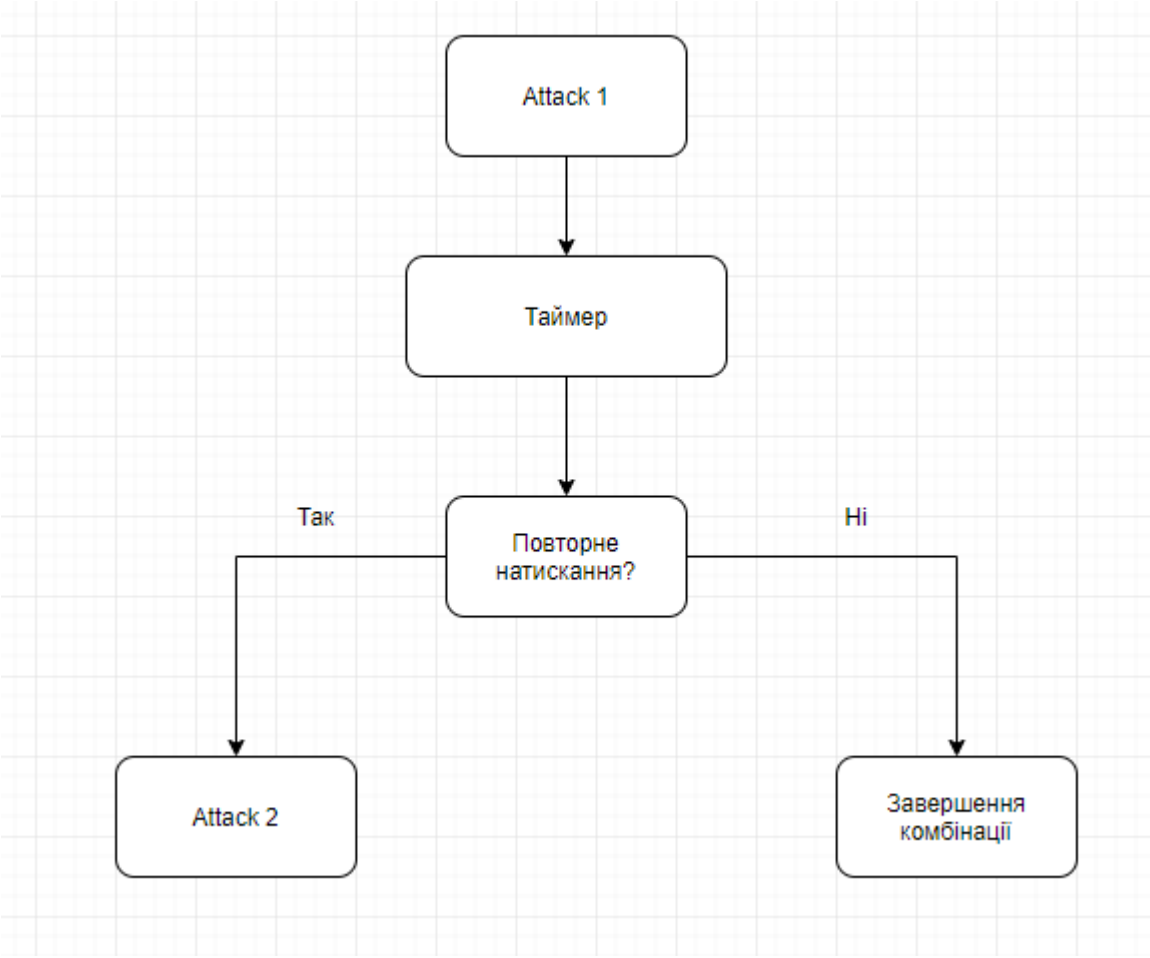


Рисунок 2.14 – Схема роботи комбо системи

Реалізація системи пошкоджень

Після успішного попадання атаки противнику завдається певна кількість пошкоджень.

У межах програмного продукту кожен противник має власний запас здоров’я.

При отриманні пошкодження запас здоров’я зменшується.

Одночасно запускаються візуальні ефекти та анімації, які повідомляють користувача про успішне попадання.

Використання окремої системи здоров’я дозволяє легко змінювати баланс гри та налаштовувати складність проходження.

Реалізація смерті противника

Після втрати всього запасу здоров’я противник переходить у стан смерті.

У даному стані виконується:

– запуск анімації смерті;

– блокування подальших атак;

– вимкнення взаємодії з персонажем;

– створення колекційного предмета.

Після завершення анімації об'єкт противника видаляється зі сцени.

Подібний підхід забезпечує коректне завершення життєвого циклу противника та звільняє ресурси системи.

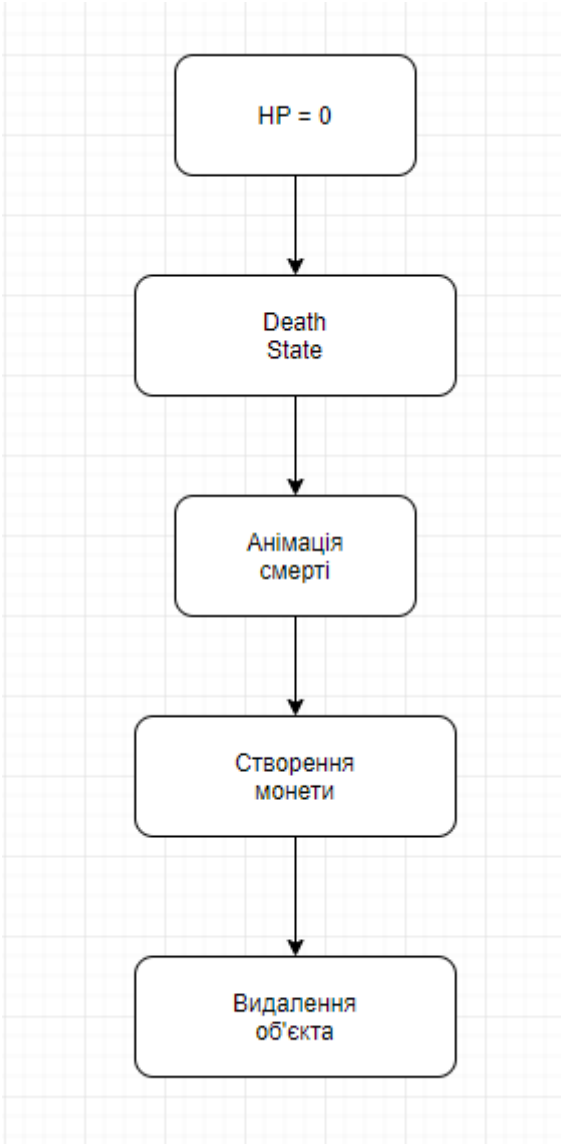


Рисунок 2.15 – Алгоритм смерті противника

Реалізація атаки противника

Для забезпечення повноцінної бойової взаємодії реалізовано систему атак противників.

Після виявлення персонажа противник переходить у режим переслідування.

Коли відстань між персонажем та противником стає достатньо малою, активується стан атаки.

Для визначення контакту використовується спеціальний тригерний колайдер.

Після входження персонажа в зону атаки запускається коротка затримка, що імітує підготовку до удару.

Подібне рішення дозволяє надати користувачу можливість вчасно відреагувати на атаку та використати перекат для ухилення.

Реалізація системи ухилення

Для уникнення пошкоджень використовується механіка перекату.

Під час виконання перекату персонаж може уникати атак противників, якщо встигає покинути небезпечну область до моменту нанесення шкоди.

Дана система створює додатковий тактичний елемент та заохочує користувача уважно стежити за поведінкою ворогів.

Реалізація отримання пошкоджень персонажем

При успішній атаці противника система виконує зменшення кількості здоров’я персонажа.

Одночасно запускаються:

– анімація отримання пошкодження;

– оновлення інтерфейсу здоров’я;

– перевірка умови смерті персонажа.

Відображення втрати здоров’я реалізовано через зміну стану графічних індикаторів у користувацькому інтерфейсі.

Подібне рішення дозволяє користувачу миттєво отримувати інформацію про поточний стан персонажа.

Реалізація смерті персонажа

У випадку втрати всіх одиниць здоров’я персонаж переходить у стан смерті.

Після цього:

– блокується можливість керування;

– запускається анімація смерті;

– відображається екран завершення гри.

Завдяки цьому користувач отримує зрозуміле повідомлення про завершення поточної спроби проходження.

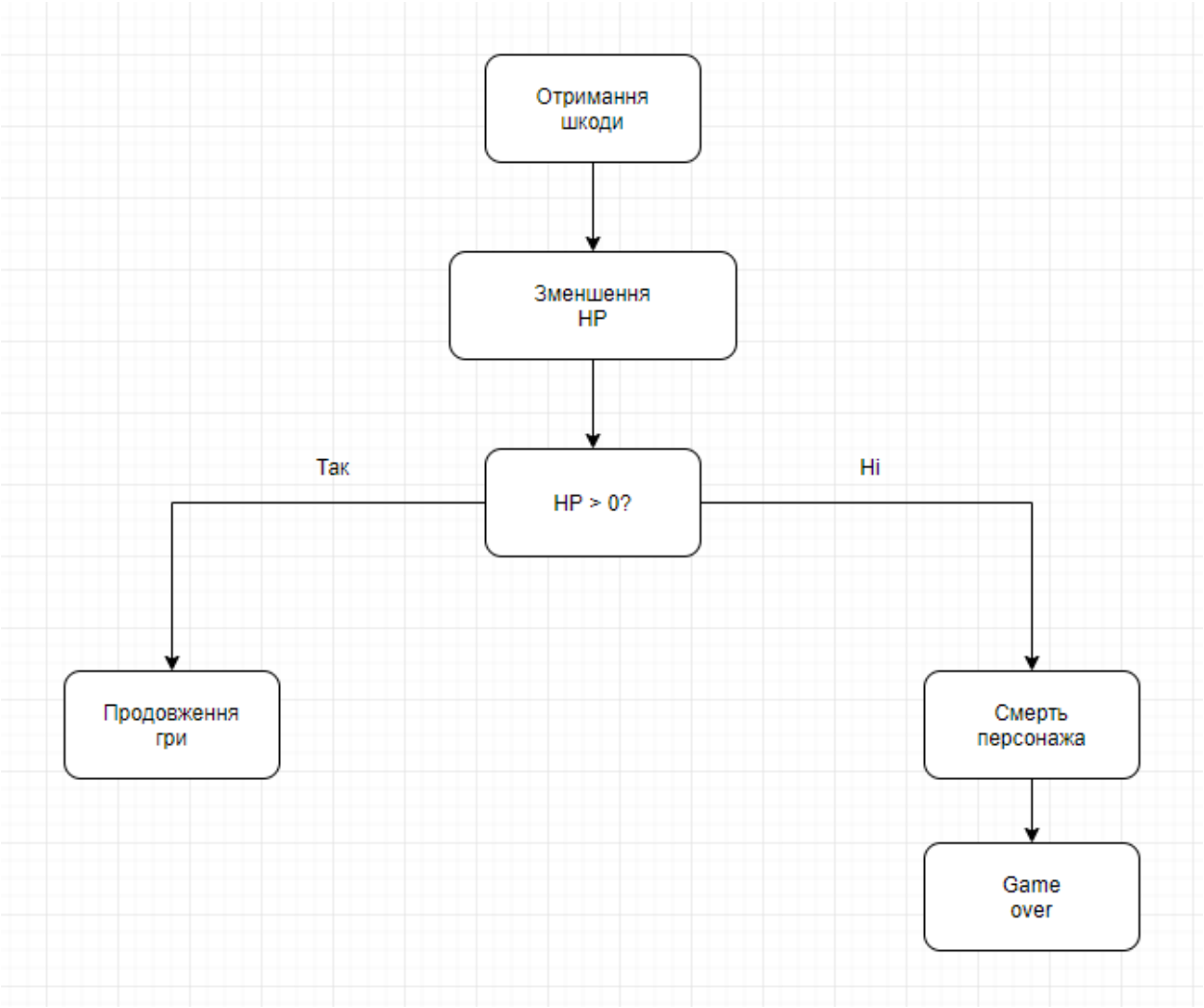


Рисунок 2.16 – Алгоритм завершення гри

Висновки до підрозділу

У результаті розробки бойової системи було реалізовано повний цикл взаємодії між персонажем та противниками.

Створено систему атак ближнього бою, механіку комбінацій ударів, систему отримання пошкоджень, механізми смерті персонажа та

противників, а також реалізовано можливість ухилення від атак за допомогою перекату.

Розроблена бойова система забезпечує необхідний рівень динаміки ігрового процесу та відповідає основним вимогам жанру Metroidvania.

2.6 Реалізація системи анімацій

Одним із важливих компонентів сучасних комп’ютерних ігор є система анімацій. Анімація забезпечує візуальне відображення дій персонажів та об'єктів, підвищує реалістичність ігрового процесу та сприяє покращенню сприйняття гри користувачем.

У розробленому програмному продукті система анімацій реалізована за допомогою вбудованих засобів рушія Unity. Для керування анімаціями використовуються компоненти Animator та Animation Controller.

Основними завданнями системи анімацій є:

– відображення руху персонажа;

– відображення бойових дій;

– відображення отримання пошкоджень;

– відображення смерті персонажа;

– відображення поведінки противників;

– синхронізація візуальної частини гри з програмною логікою.

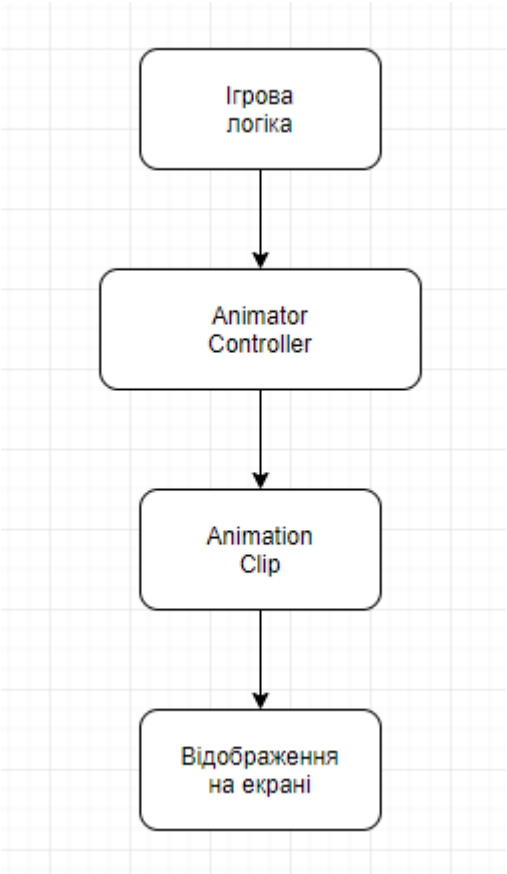


Рисунок 2.17 – Загальна структура системи анімацій

Реалізація анімацій персонажа

Для керування анімаціями головного персонажа використовується компонент PlayerAnimationController.

Даний компонент взаємодіє із системою керування персонажем та отримує інформацію про його поточний стан.

На основі отриманих даних здійснюється перемикання між відповідними анімаційними станами.

У поточній версії програмного продукту реалізовано такі анімації персонажа:

– Idle;

– Run;

– Jump;

– Roll;

– Attack1;

– Attack2;

– Hurt;

– Death.

Кожна з анімацій відповідає певному стану персонажа та запускається автоматично при виконанні відповідної дії.

Реалізація анімації очікування

Анімація Idle використовується у випадках, коли персонаж знаходиться у стані спокою та не виконує жодних дій.

Дана анімація є базовим станом Animator Controller та активується автоматично після завершення інших анімацій.

Використання окремої анімації очікування дозволяє зробити персонажа більш живим навіть за відсутності взаємодії користувача.

Реалізація анімації руху

Під час переміщення персонажа активується анімація Run.

Перехід до даного стану виконується автоматично після виявлення горизонтального руху.

Для визначення факту переміщення використовується значення горизонтального вводу користувача.

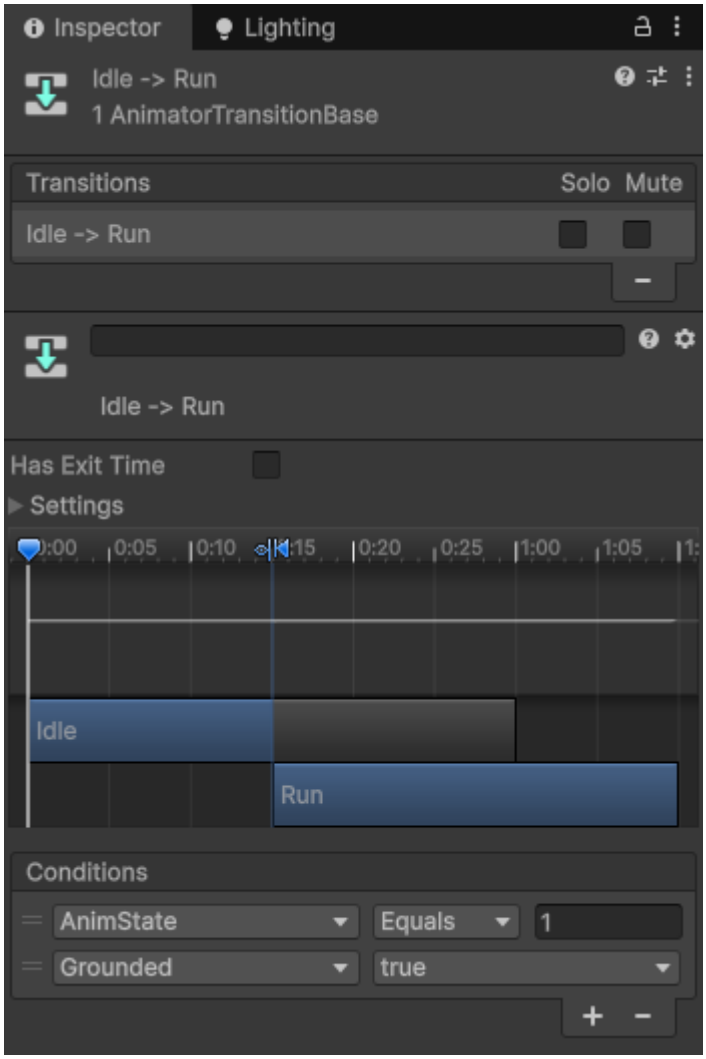


Рисунок 2.18 – Переходи між станами Idle та Run

Використання окремої анімації руху дозволяє забезпечити візуальний зв’язок між діями користувача та поведінкою персонажа.

Реалізація анімації стрибка

Однією з основних механік жанру Metroidvania є подолання платформних перешкод.

Для відображення даної дії реалізовано анімацію Jump.

Запуск анімації здійснюється після натискання кнопки стрибка.

Одночасно система передає інформацію до Animator про зміну вертикальної швидкості персонажа.

Для визначення стану польоту використовується спеціальний параметр AirSpeedY.

На основі його значення Animator автоматично перемикає відповідні анімаційні стани.

Подібний підхід забезпечує плавні переходи між анімаціями стрибка, підйому та падіння.

Реалізація анімації перекату

Для механіки ухилення використовується окрема анімація Roll.

Під час її відтворення персонаж виконує швидке переміщення в напрямку руху.

Одночасно відбувається блокування деяких інших дій для запобігання виникненню конфліктів між анімаційними станами.

Реалізація перекату через окрему анімацію дозволяє зробити дану механіку більш помітною та зрозумілою для користувача.

Реалізація бойових анімацій

Для відображення атак використовується система комбінацій із двох ударів.

У межах програмного продукту реалізовано:

– Attack1;

– Attack2.

Після натискання кнопки атаки запускається перша анімація удару.

Якщо користувач повторно натискає кнопку атаки протягом встановленого проміжку часу, система активує другу анімацію.

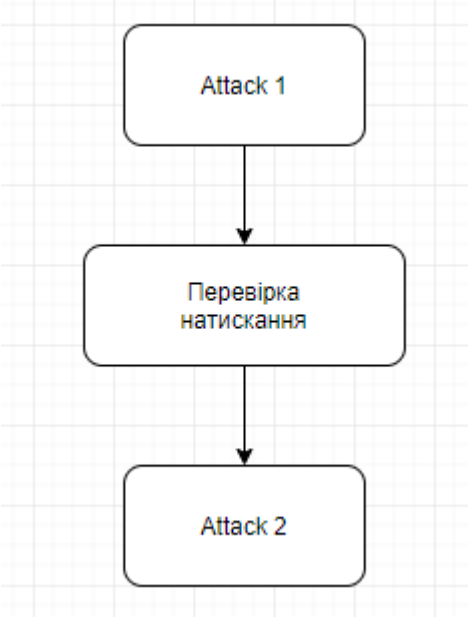


Рисунок 2.19 – Схема роботи бойових анімацій

Використання декількох анімацій ударів дозволяє зробити бойову систему більш динамічною та різноманітною.

Реалізація анімації отримання пошкодження

Після отримання шкоди персонажем запускається анімація Hurt.

Її основним призначенням є візуальне інформування користувача про факт отримання пошкодження.

Дана анімація активується автоматично під час виклику системи здоров’я та супроводжується зміною стану індикаторів життів.

Використання окремої анімації пошкодження покращує зворотний зв’язок між грою та користувачем.

Реалізація анімації смерті

У випадку втрати всього запасу здоров’я запускається анімація Death.

Після її активації персонаж більше не може виконувати рухи, атаки чи інші дії.

Даний стан є завершальним у життєвому циклі персонажа та використовується перед відображенням екрана завершення гри.

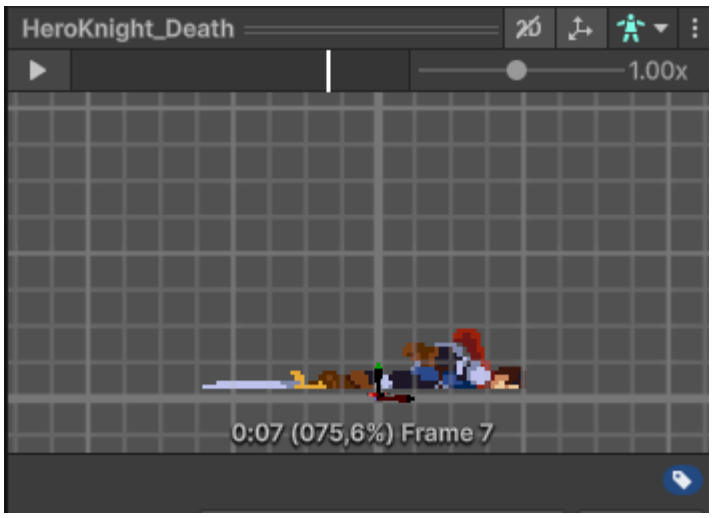


Рисунок 2.20 – Стан смерті персонажа

Реалізація анімацій противника

Для противників використовується окремий компонент EnemyAnimationController.

Подібне рішення дозволяє незалежно керувати анімаційними станами ворогів.

У поточній реалізації використовуються такі анімації:

– Idle;

– Patrol;

– Attack;

– Death.

Перемикання між анімаціями здійснюється на основі поточного стану противника.

Наприклад, при переході до режиму атаки автоматично запускається анімація Attack.

У випадку знищення противника активується анімація Death.

Синхронізація анімацій із логікою гри

Одним із важливих аспектів реалізації анімацій є їх синхронізація з програмною логікою.

У програмному продукті всі анімаційні переходи залежать від фактичного стану ігрових об'єктів.

Наприклад:

– рух персонажа запускає анімацію Run;

– натискання кнопки атаки запускає Attack;

– отримання шкоди запускає Hurt;

– смерть персонажа запускає Death.

Завдяки цьому забезпечується коректне відображення поточного стану гри та покращується сприйняття ігрового процесу користувачем.

Висновки до підрозділу

У результаті розробки системи анімацій було створено набір анімаційних станів для персонажа та противників.

Реалізовано анімації руху, стрибка, перекату, атак, отримання пошкоджень та смерті. Для керування анімаціями використано систему Animator Controller, яка забезпечує автоматичне перемикання між станами залежно від поточної ігрової ситуації.

Створена система анімацій дозволяє підвищити візуальну якість програмного продукту та забезпечує тісний зв’язок між діями користувача та відображенням подій на екрані.

2.7 Реалізація системи здоров'я персонажа

Однією з основних складових ігрового процесу є система здоров’я персонажа. Дана підсистема відповідає за облік отриманих пошкоджень,

відображення поточного стану персонажа та визначення моменту завершення гри у випадку втрати всіх одиниць здоров’я.

У розробленому програмному продукті система здоров’я реалізована за допомогою окремого програмного компонента PlayerHealthSystem. Використання окремого модуля дозволяє відокремити логіку обробки пошкоджень від інших підсистем гри та забезпечує зручність подальшого розширення функціоналу.

Основними завданнями системи здоров’я є:

– зберігання інформації про поточний запас здоров’я персонажа;

– обробка отримання пошкоджень;

– оновлення графічного інтерфейсу;

– запуск анімацій отримання шкоди;

– визначення факту смерті персонажа;

– передача інформації іншим підсистемам гри.

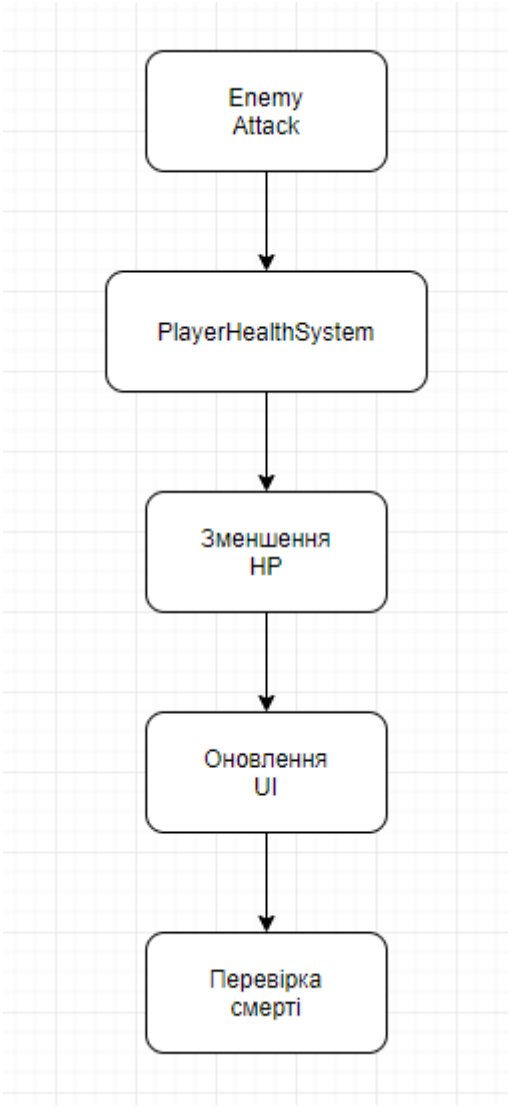


Рисунок 2.21 – Структура системи здоров’я

Організація запасу здоров’я

У поточній версії програмного продукту використовується система фіксованого запасу здоров’я.

На початку гри персонаж має три одиниці здоров’я, які відображаються у вигляді окремих графічних елементів користувацького інтерфейсу.

Подібне рішення широко використовується в платформерах та пригодницьких іграх завдяки своїй простоті та наочності.

Користувач може швидко оцінити поточний стан персонажа без необхідності аналізу числових значень або додаткових індикаторів.

Рисунок 2.22 – Відображення здоров’я персонажа

Реалізація отримання пошкоджень

Пошкодження персонажа відбувається під час успішної атаки противника.

Після отримання сигналу про нанесення шкоди система виконує перевірку поточного стану здоров’я та зменшує кількість активних одиниць життя.

Одночасно відбувається запуск додаткових механізмів, які повідомляють користувача про отримання пошкодження.

До таких механізмів належать:

– зміна стану індикаторів здоров’я;

– запуск анімації отримання шкоди;

– оновлення поточного стану персонажа.

Завдяки цьому користувач отримує миттєвий візуальний зворотний зв’язок щодо результату бойової взаємодії.

Відображення стану здоров’я

Для відображення кількості життів використовуються окремі графічні об’єкти інтерфейсу.

Кожен індикатор відповідає одній одиниці здоров’я персонажа.

При отриманні пошкодження один із графічних елементів вимикається, що візуально демонструє втрату життя.

Подібний підхід дозволяє реалізувати простий та зрозумілий інтерфейс без використання складних систем обчислення відсотків або числових значень.

Рисунок 2.23 – Алгоритм оновлення індикаторів здоров’я

Реалізація анімації отримання пошкодження

Для покращення сприйняття ігрового процесу система здоров’я інтегрована з анімаційною підсистемою.

При отриманні пошкодження автоматично запускається анімація Hurt.

Дана анімація дозволяє користувачу одразу визначити факт отримання шкоди навіть без спостереження за індикаторами здоров’я.

Використання подібного підходу є поширеною практикою у сучасних комп’ютерних іграх та сприяє підвищенню якості взаємодії між користувачем і програмним продуктом.

Реалізація смерті персонажа

Після втрати останньої одиниці здоров’я система переводить персонажа у стан смерті.

У даному стані:

– блокується подальше керування персонажем;

– запускається анімація смерті;

– припиняється виконання ігрових дій;

– активується екран завершення гри.

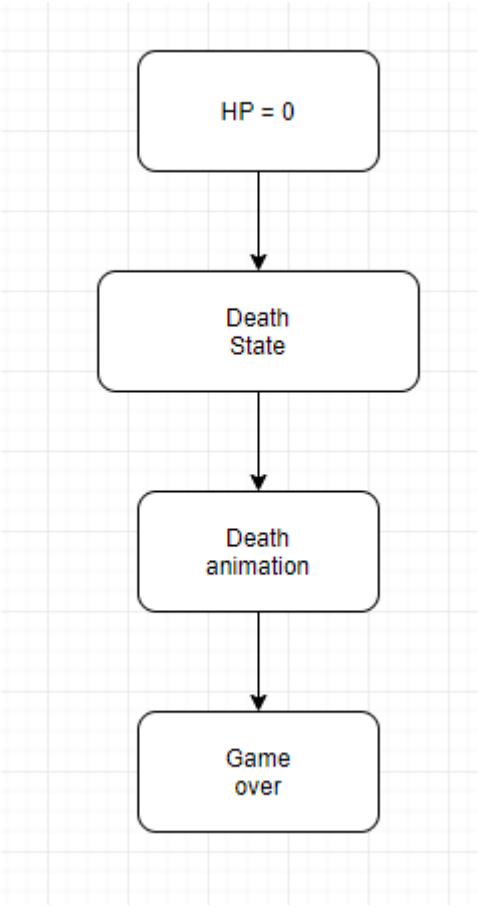


Рисунок 2.24 – Алгоритм смерті персонажа

Використання окремого стану смерті дозволяє коректно завершити всі активні процеси та уникнути виникнення помилок у роботі гри.

Взаємодія з іншими підсистемами

Система здоров’я взаємодіє з декількома компонентами програмного продукту.

Основними з них є:

– бойова система;

– система анімацій;

– користувацький інтерфейс;

– система завершення гри.

Подібна інтеграція забезпечує узгоджену роботу всіх елементів програмного продукту та дозволяє підтримувати цілісність ігрового процесу.

Висновки до підрозділу

У результаті реалізації системи здоров’я було створено механізм контролю стану персонажа, який забезпечує обробку отриманих пошкоджень, оновлення інтерфейсу та визначення моменту завершення гри.

Розроблена підсистема інтегрована з бойовою системою та системою анімацій, що дозволяє забезпечити коректне відображення результатів взаємодії між персонажем і противниками та покращує сприйняття ігрового процесу користувачем.

2.8 Реалізація переходів між сценами

Однією з характерних особливостей ігор жанру Metroidvania є наявність великого взаємопов’язаного світу, який складається з окремих локацій. Для забезпечення переміщення користувача між різними частинами ігрового світу необхідно реалізувати механізм переходу між сценами.

У розробленому програмному продукті для організації переходів між локаціями використовується система зміни сцен рушія Unity. Додатково реалізовано механізм плавного затемнення екрана, який

забезпечує більш комфортне сприйняття переходів між ігровими областями.

Основними компонентами підсистеми є:

– ScenePortal;

– SceneTransitionManager;

– система завантаження сцен Unity;

– інтерфейсний елемент Fade Panel.

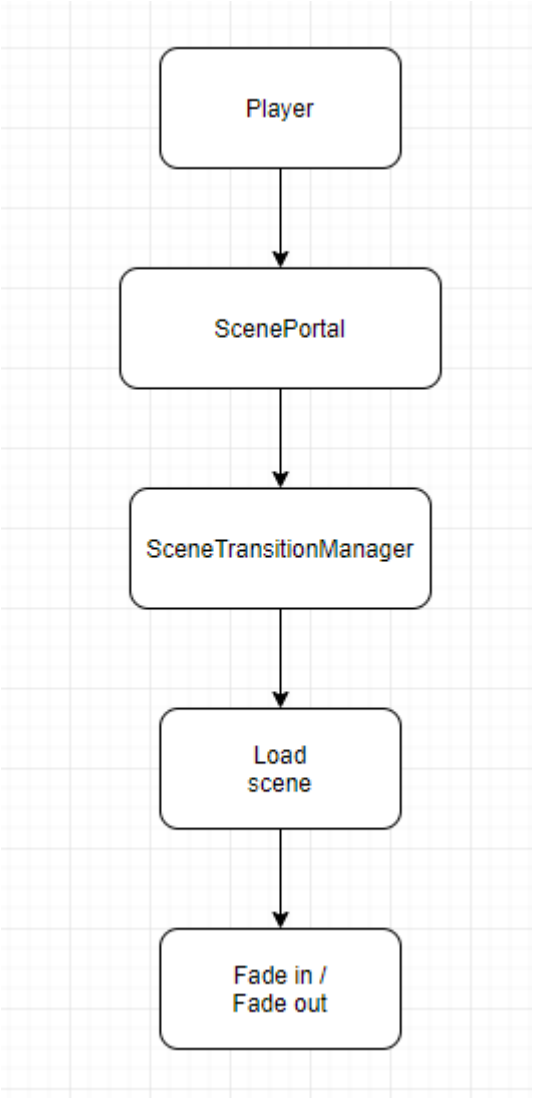


Рисунок 2.25 – Структура системи переходів між сценами

Організація переходів між локаціями

Для ініціації переходу між сценами використовуються спеціальні ігрові об'єкти — портали.

Портал являє собою тригерну область, яка відстежує входження персонажа.

Після входження гравця в область дії порталу активується механізм завантаження нової сцени.

Кожен портал містить інформацію про сцену, яка повинна бути завантажена після активації.

Подібний підхід дозволяє легко створювати переходи між будь-якими локаціями без зміни основної логіки гри.

Реалізація менеджера переходів

Для централізації логіки роботи зі сценами використовується окремий компонент SceneTransitionManager.

Основним завданням даного компонента є керування процесом переходу між сценами та синхронізація візуальних ефектів із процесом завантаження.

Менеджер переходів реалізований за принципом Singleton.

Подібний підхід дозволяє створити єдиний екземпляр компонента, доступний із будь-якої частини програмного продукту.

Використання шаблону Singleton забезпечує:

– централізоване керування переходами;

– уникнення дублювання компонентів;

– спрощення взаємодії між підсистемами.

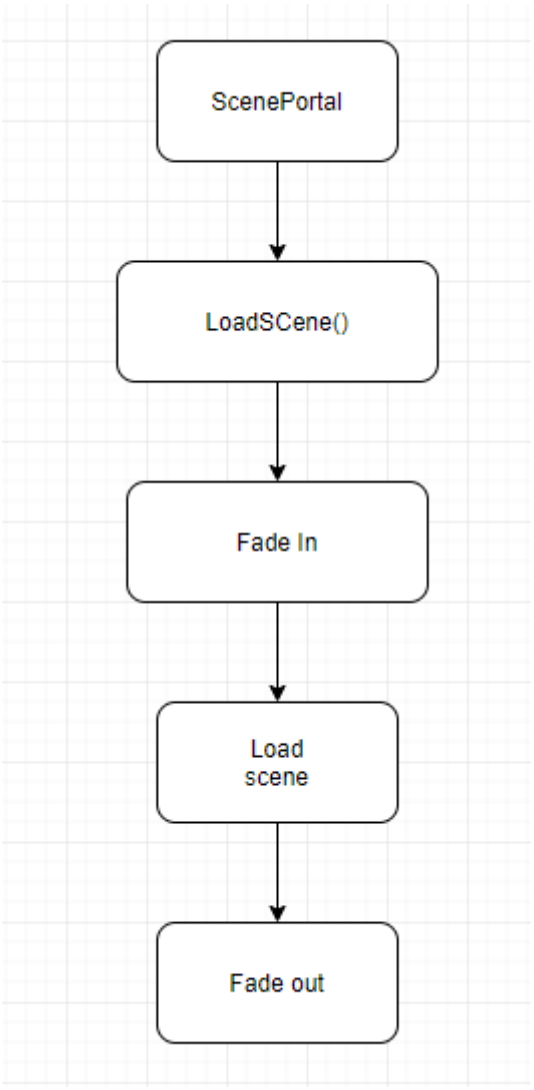


Рисунок 2.26 – Принцип роботи SceneTransitionManager

Реалізація затемнення екрана

Для підвищення якості користувацького досвіду реалізовано плавне затемнення екрана.

Під час переходу між сценами користувач спостерігає поступове затемнення зображення, після чого виконується завантаження нової локації.

Після завершення завантаження відбувається плавне повернення зображення до нормального стану.

Реалізація затемнення виконується шляхом зміни прозорості спеціального графічного елемента інтерфейсу.

Для цього використовується компонент Image та параметр Alpha кольору.

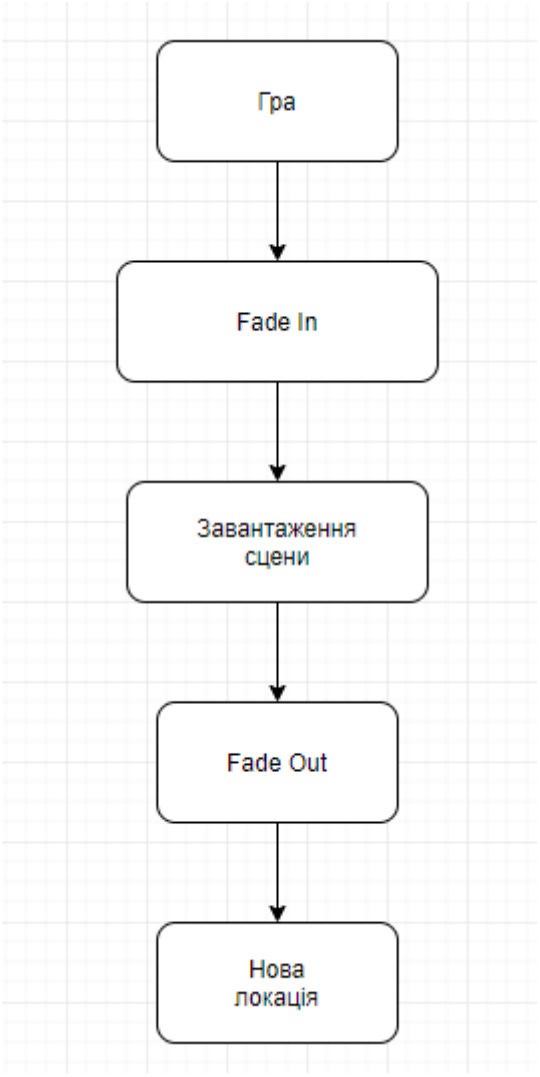


Рисунок 2.27 – Етапи переходу між сценами

Подібне рішення дозволяє приховати процес завантаження та зробити переходи більш плавними.

Завантаження нової сцени

Після завершення анімації затемнення виконується асинхронне завантаження нової сцени.

Використання асинхронного завантаження дозволяє уникнути блокування головного потоку програми та забезпечує стабільну роботу ігрового процесу.

Під час виконання завантаження система постійно контролює стан операції та очікує завершення процесу.

Після успішного завантаження нової сцени запускається процедура відновлення відображення.

Позиціонування персонажа

Після завершення завантаження сцени виконується позиціонування персонажа в новій локації.

У поточній версії програмного продукту використовується фіксована точка появи персонажа.

Подібне рішення було обране через простоту реалізації та достатність для поточного етапу розробки.

У подальшому дана система може бути розширена шляхом використання декількох точок появи залежно від напрямку переходу між локаціями.

Взаємодія з іншими підсистемами

Підсистема переходів взаємодіє з такими компонентами:

– системою керування персонажем;

– користувацьким інтерфейсом;

– менеджером сцен Unity;

– системою анімацій.

Спільна робота зазначених компонентів забезпечує коректне переміщення користувача між різними областями ігрового світу.

Переваги реалізованого рішення

Створена система переходів між сценами має низку переваг:

– простота реалізації;

– можливість масштабування;

– зручність використання;

– плавність переходів;

– централізоване керування логікою завантаження.

Завдяки цьому система може використовуватися як основа для подальшого розширення ігрового світу.

Висновки до підрозділу

У результаті реалізації системи переходів між сценами було створено механізм переміщення користувача між окремими локаціями ігрового світу.

Для цього використано спеціальні об'єкти переходу та централізований менеджер сцен. Додатково реалізовано систему плавного затемнення екрана, яка забезпечує комфортне сприйняття переходів між ігровими областями.

Створена підсистема відповідає вимогам жанру Metroidvania та дозволяє організувати структуру світу, що складається з декількох взаємопов’язаних локацій.

РОЗРАХУНКОВИЙ РОЗДІЛ

3.1 Розробка штучного інтелекту противника

Однією з важливих складових сучасних комп'ютерних ігор є система штучного інтелекту. Саме вона забезпечує керування поведінкою неігрових персонажів та формує значну частину ігрового процесу.

У межах розробленого програмного продукту реалізовано систему штучного інтелекту противника, яка дозволяє автоматизувати його поведінку та забезпечити взаємодію з персонажем користувача.

Основним завданням штучного інтелекту є виконання таких функцій:

– патрулювання території;

– виявлення персонажа;

– переслідування персонажа;

– виконання атаки;

– обробка стану смерті.

Для реалізації поведінки противника використовується кінцевий автомат станів (Finite State Machine).

Даний підхід широко застосовується в ігровій індустрії завдяки своїй простоті, наочності та високій ефективності.

Кожен стан описує певний режим роботи противника та визначає набір доступних дій.

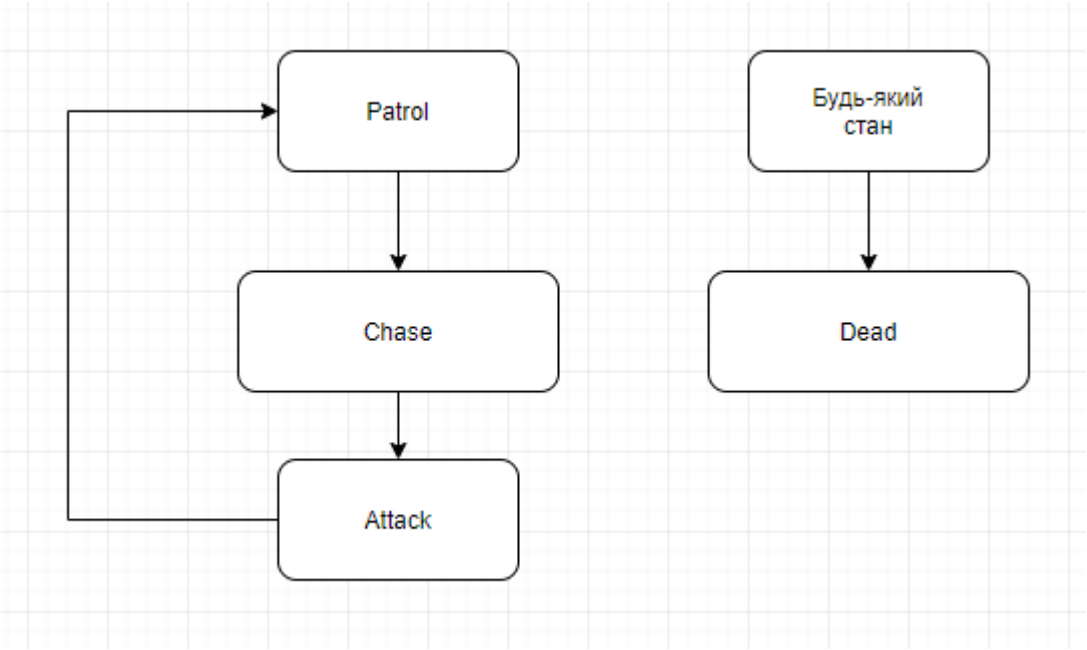


Рисунок 3.1 – Схема станів противника

Реалізація стану Patrol

Стан Patrol є базовим режимом роботи противника.

У даному режимі противник виконує переміщення між заздалегідь визначеними точками маршруту.

Основним призначенням патрулювання є створення відчуття активності ігрового світу та забезпечення постійної присутності ворогів на локації.

Під час перебування у стані Patrol противник не взаємодіє з персонажем та виконує лише переміщення за встановленим маршрутом.

Для реалізації даного механізму використовується система контрольних точок патрулювання.

Після досягнення чергової точки маршруту напрямок руху автоматично змінюється.

Подібне рішення дозволяє створити просту, але ефективну модель поведінки противника.

Реалізація виявлення персонажа

Для виявлення персонажа використовується спеціальна тригерна область.

Після входження гравця в зону виявлення система автоматично змінює поточний стан противника.

Таким чином реалізується механізм початку переслідування.

Для реалізації даної функції використовується окремий компонент ChaseCollider.

Перевагою такого підходу є відсутність необхідності постійного обчислення відстані між персонажем і противником.

Використання тригерів дозволяє зменшити навантаження на систему та підвищити продуктивність програмного продукту.

Реалізація стану Chase

Після виявлення персонажа противник переходить у стан Chase.

У даному режимі противник починає переміщатися в напрямку персонажа.

Основним завданням стану Chase є скорочення відстані між ворогом та гравцем.

Для визначення напрямку руху використовується поточне положення персонажа.

У процесі переслідування противник постійно оновлює напрямок руху та коригує власне положення.

Використання окремого стану переслідування дозволяє забезпечити більш природну поведінку противника.

Реалізація стану Attack

Після входження персонажа в область атаки активується стан Attack.

У цьому режимі противник припиняє переслідування та переходить до виконання бойових дій.

Для реалізації атаки використовується окремий компонент EnemyAttackCollider.

Після входження персонажа до зони ураження запускається коротка затримка перед нанесенням шкоди.

Дана затримка виконує одразу декілька функцій:

– забезпечує синхронізацію з анімацією атаки;

– надає користувачу можливість ухилитися від удару;

– підвищує реалістичність поведінки противника.

Після завершення затримки виконується перевірка поточного стану персонажа.

Якщо персонаж не використовує перекат та знаходиться в зоні ураження, йому завдається пошкодження.

Реалізація стану Dead

Стан Dead використовується після повного вичерпання запасу здоров’я противника.

У цьому режимі:

– припиняється патрулювання;

– вимикається переслідування;

– блокується можливість виконання атак;

– запускається анімація смерті.

Після завершення анімації противник видаляється зі сцени.

Одночасно створюється колекційний предмет у вигляді монети.

Подібний підхід дозволяє пов’язати бойову систему з системою збору ресурсів.

Реалізація анімацій штучного інтелекту

Для відображення поточного стану противника використовується окремий компонент EnemyAnimationController.

Даний модуль автоматично аналізує поточний стан противника та активує відповідну анімацію.

У межах програмного продукту реалізовано такі анімаційні стани:

– Idle;

– Patrol;

– Attack;

– Death.

Завдяки інтеграції системи анімацій із логікою штучного інтелекту забезпечується коректне відображення поведінки противника.

Переваги використання кінцевого автомата станів

Використання кінцевого автомата станів має ряд переваг:

– простота реалізації;

– зручність налагодження;

– висока продуктивність;

– можливість подальшого розширення;

– наочність структури поведінки.

У майбутньому дана система може бути доповнена новими станами без суттєвої зміни архітектури програмного продукту.

Висновки до підрозділу

У результаті виконання даного етапу було реалізовано систему штучного інтелекту противника на основі кінцевого автомата станів.

Створена система забезпечує патрулювання території, виявлення персонажа, переслідування, виконання атак та коректну обробку смерті противника.

Розроблене рішення дозволяє створити динамічну взаємодію між персонажем та ворогами і відповідає вимогам комп’ютерних ігор жанру Metroidvania.

3.2 Реалізація системи збору монет

Одним із важливих елементів ігрового процесу є система збору ресурсів. У багатьох комп’ютерних іграх колекційні предмети використовуються як засіб мотивації користувача до дослідження ігрового світу та взаємодії з навколишнім середовищем.

У розробленому програмному продукті реалізовано систему збору монет, яка дозволяє користувачу отримувати винагороду за знищення противників та дослідження локацій.

Монети виступають колекційними об'єктами, які можуть бути підібрані персонажем після взаємодії з ними.

Основними завданнями системи збору монет є:

– створення колекційних предметів;

– анімація монет;

– визначення факту підбору;

– збільшення лічильника монет;

– оновлення користувацького інтерфейсу.

Організація об'єкта монети

Монета реалізована у вигляді окремого ігрового об'єкта.

Для відображення використовується двовимірний спрайт, а для взаємодії з персонажем застосовується тригерний колайдер.

Після створення монета автоматично починає виконувати анімацію руху, що дозволяє привертати увагу користувача та підвищує помітність об'єкта на рівні.

Реалізація анімації монети

Для створення більш живого візуального ефекту реалізовано вертикальний рух монети.

Після появи об'єкт плавно переміщується вгору та вниз відносно початкового положення.

Даний ефект створює відчуття «плавання» предмета в повітрі та широко використовується в платформерах і пригодницьких іграх.

Для реалізації руху використовується програмний алгоритм, який періодично змінює напрямок переміщення між верхньою та нижньою точками.

Використання такого підходу дозволяє створити плавну анімацію без необхідності використання додаткових анімаційних ресурсів.

Реалізація підбору монет

Підбір монети виконується автоматично після контакту персонажа з об'єктом.

Для визначення взаємодії використовується тригерний колайдер.

Після входження персонажа в область монети система виконує такі дії:

– збільшує значення лічильника монет;

– оновлює користувацький інтерфейс;

– видаляє монету зі сцени.

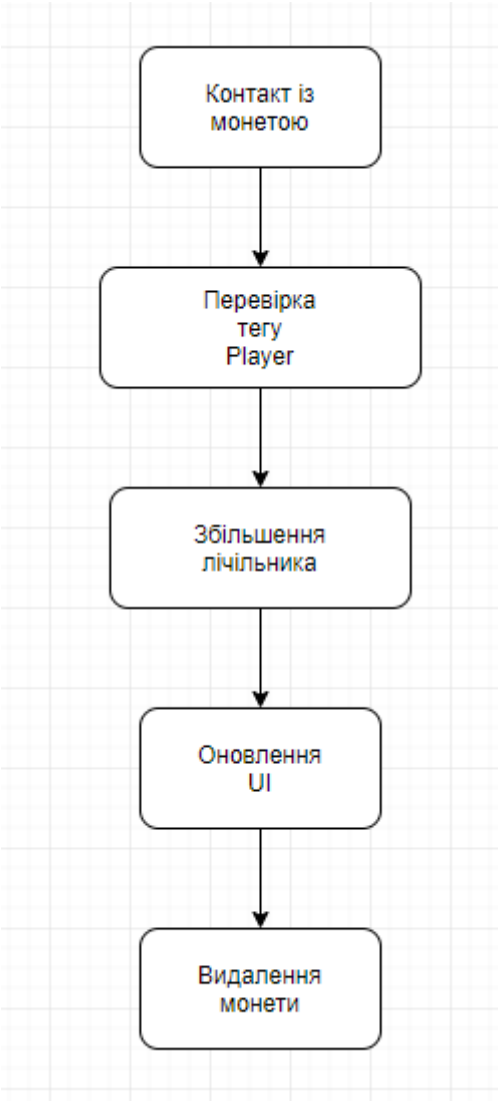


Рисунок 3.2 – Алгоритм підбору монети

Подібний підхід дозволяє забезпечити миттєвий зворотний зв’язок для користувача та спрощує взаємодію з колекційними предметами.

Інтеграція з бойовою системою

Система збору монет безпосередньо пов'язана з бойовою системою гри.

Після знищення противника створюється новий об'єкт монети.

Таким чином користувач отримує винагороду за перемогу над ворогами.

Подібне рішення стимулює активну взаємодію з противниками та підвищує зацікавленість користувача у проходженні гри.

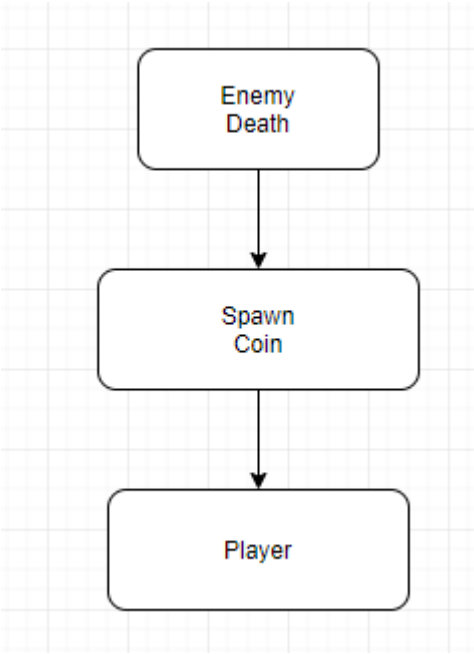


Рисунок 3.3 – Взаємодія бойової системи та системи монет

Відображення кількості монет

Для інформування користувача про кількість зібраних ресурсів використовується окремий елемент інтерфейсу.

Після підбору монети лічильник автоматично оновлюється та відображає актуальне значення.

Завдяки цьому користувач може постійно контролювати результати своєї діяльності під час проходження гри.

Переваги реалізованої системи

Створена система збору монет має ряд переваг:

– проста реалізація;

– низьке навантаження на систему;

– зручність використання;

– інтеграція з бойовою системою;

– можливість подальшого розширення функціоналу.

У майбутньому монети можуть використовуватися як ігрова валюта для придбання покращень або відкриття нових можливостей персонажа.

Висновки до підрозділу

У результаті розробки було створено систему збору монет, яка забезпечує взаємодію користувача з колекційними предметами та дозволяє отримувати винагороду за перемогу над противниками.

Реалізована система інтегрована з бойовою системою та користувацьким інтерфейсом, що забезпечує цілісність ігрового процесу та відповідає вимогам жанру Metroidvania.

3.3 Реалізація контрольних точок

Одним із важливих елементів сучасних комп'ютерних ігор є система контрольних точок. Її основним призначенням є збереження прогресу користувача та створення зручних умов для проходження ігрових локацій.

Особливо важливою дана механіка є для ігор жанру Metroidvania, оскільки користувач постійно досліджує великі взаємопов'язані області та може витрачати значний час на проходження окремих ділянок рівня.

У розробленому програмному продукті реалізовано систему контрольних точок, яка дозволяє фіксувати поточне положення персонажа та використовувати його як точку відродження після загибелі.

Основними функціями контрольних точок є:

– визначення факту активації;

– збереження координат персонажа;

– візуальне відображення активного стану;

– забезпечення можливості відновлення прогресу.

Організація контрольної точки

Контрольна точка реалізована у вигляді окремого ігрового об'єкта.

До її складу входять:

– Sprite Renderer;

– Collider2D;

– програмний компонент CheckPoint.

При контакті персонажа з контрольної точкою система виконує перевірку наявності взаємодії та активує відповідний механізм збереження.

Подібний підхід дозволяє створювати будь-яку кількість контрольних точок на різних локаціях без зміни програмної логіки.

Реалізація активації контрольної точки

Після входження персонажа до області контрольної точки система виконує її активацію.

У процесі активації відбувається:

– збереження поточної позиції;

– зміна візуального стану об'єкта;

– оновлення інформації про останню активну контрольну точку.

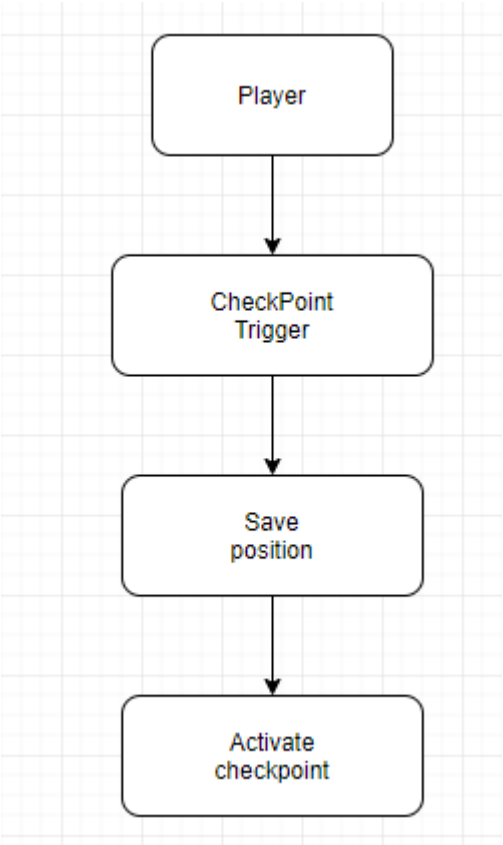


Рисунок 3.4 – Алгоритм активації контрольної точки

Завдяки цьому користувач отримує підтвердження успішної активації точки збереження.

Збереження позиції персонажа

Після активації контрольної точки система запам’ятовує координати персонажа.

Збережені координати можуть використовуватися для відновлення прогресу після смерті персонажа або повторного запуску локації.

У поточній версії програмного продукту реалізовано базову систему збереження позиції, яка може бути розширена в майбутньому.

Подібна архітектура дозволяє без значних змін інтегрувати повноцінну систему збережень.

Переваги реалізованої системи

Основними перевагами створеної системи є:

– простота реалізації;

– зручність використання;

– мінімальне навантаження на систему;

– можливість масштабування;

– відповідність особливостям жанру Metroidvania.

Наявність контрольних точок значно підвищує комфорт проходження гри та зменшує втрату прогресу після помилок користувача.

Висновки до підрозділу

У результаті виконання даного етапу було реалізовано систему контрольних точок, яка забезпечує фіксацію прогресу користувача під час проходження ігрових локацій.

Розроблена система дозволяє зберігати координати персонажа, візуально відображати активний стан контрольної точки та створює основу для подальшого впровадження повноцінної системи збережень.

3.4 Реалізація здібності Dash

Однією з характерних особливостей жанру Metroidvania є система поступового розвитку персонажа. У процесі проходження гри користувач отримує нові здібності, які дозволяють відкривати раніше недоступні ділянки ігрового світу та розширюють набір доступних механік.

У розробленому програмному продукті реалізовано здібність Dash, яка представлена у вигляді перекату персонажа. Дана механіка використовується як для подолання небезпечних ділянок рівня, так і для ухилення від атак противників.

Основними завданнями системи Dash є:

– надання персонажу додаткової мобільності;

– підвищення динамічності бойової системи;

– створення елемента прогресії персонажа;

– розширення можливостей дослідження ігрового світу.

Організація системи отримання здібності

На відміну від базових механік персонажа, здібність Dash стає доступною лише після взаємодії зі спеціальним ігровим об'єктом.

Після контакту персонажа з відповідним предметом система активує нову можливість та зберігає інформацію про її отримання.

Подібний підхід відповідає класичним принципам жанру Metroidvania, де нові здібності відкривають доступ до додаткових можливостей проходження.

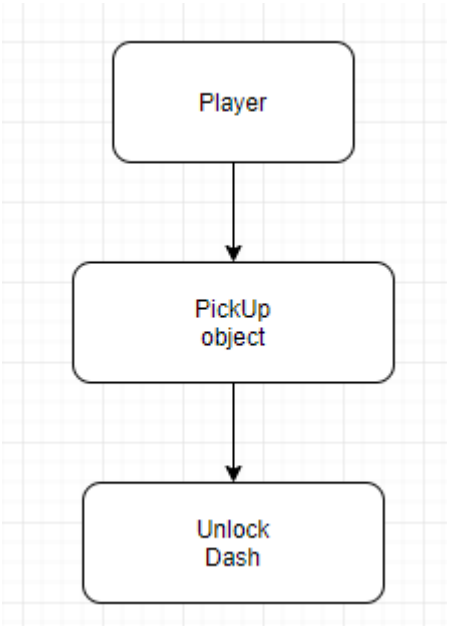


Рисунок 3.5 – Алгоритм отримання здібності Dash

Після отримання здібності користувач може використовувати її необмежену кількість разів протягом подальшого проходження гри.

Реалізація механіки Dash

Для активації здібності використовується окрема клавіша керування.

Після натискання відповідної клавіші система виконує перевірку доступності здібності та поточного стану персонажа.

Якщо всі умови виконані, запускається процедура перекату.

Під час виконання Dash персонаж отримує короткочасне прискорення в напрямку руху.

Рисунок 3.6 – Принцип роботи механіки Dash

Завдяки цьому користувач може швидко долати невеликі відстані та ефективніше реагувати на загрози.

Інтеграція з бойовою системою

Одним із головних призначень здібності Dash є використання під час бойових взаємодій.

При виконанні перекату персонаж може уникати атак противників та швидко змінювати власне положення на полі бою.

Подібна механіка значно підвищує глибину бойової системи та створює додаткові тактичні можливості.

Особливо важливою дана здібність є під час боротьби з сильними противниками та босами.

Інтеграція з системою анімацій

Для візуального відображення здібності використовується окрема анімація Roll.

Після активації Dash система автоматично запускає відповідний анімаційний стан.

Використання окремої анімації дозволяє забезпечити візуальний зв’язок між діями користувача та поведінкою персонажа.

Завдяки цьому механіка виглядає більш природною та зрозумілою для користувача.

Можливості подальшого розвитку

Створена система може бути розширена шляхом додавання нових здібностей персонажа.

До можливих напрямів розвитку належать:

– подвійний стрибок;

– ривок у повітрі;

– лазіння по стінах;

– використання спеціальних ключів для відкриття нових областей.

Подібний підхід дозволяє поступово збільшувати складність ігрового процесу та підтримувати інтерес користувача до дослідження світу.

Висновки до підрозділу

У результаті реалізації здібності Dash було створено додатковий механізм мобільності персонажа, який використовується як під час дослідження локацій, так і під час бойових взаємодій.

Розроблена система відповідає особливостям жанру Metroidvania та створює основу для подальшого розширення набору здібностей персонажа.

3.5 Організація користувацького інтерфейсу

Користувацький інтерфейс є важливою складовою будь-якої комп'ютерної гри. Саме через інтерфейс користувач отримує інформацію про поточний стан персонажа, результати своїх дій та перебіг ігрового процесу.

Під час розробки програмного продукту особлива увага приділялася створенню простого та зрозумілого інтерфейсу, який не перевантажує екран зайвою інформацією та дозволяє користувачу зосередитися на проходженні гри.

Для реалізації інтерфейсу використовувалися стандартні засоби рушія Unity, зокрема система Canvas та набір UI-компонентів.

Основними завданнями користувацького інтерфейсу є:

– відображення стану здоров'я персонажа;

– відображення кількості зібраних монет;

– відображення екрана завершення гри;

– забезпечення зворотного зв'язку між грою та користувачем.

Організація структури інтерфейсу

Усі елементи інтерфейсу розміщуються всередині Canvas.

Використання Canvas дозволяє коректно відображати інтерфейс незалежно від роздільної здатності екрана та параметрів пристрою користувача.

До структури інтерфейсу входять:

– блок відображення здоров’я;

– лічильник монет;

– панель завершення гри;

– допоміжні графічні елементи.

Рисунок 3.7 – Загальна структура користувацького інтерфейсу

Реалізація індикаторів здоров’я

Для відображення запасу здоров’я персонажа використовуються окремі графічні елементи у вигляді сердець.

Кожне серце відповідає одній одиниці здоров’я.

При отриманні пошкодження один із елементів автоматично вимикається.

Подібний спосіб відображення широко використовується в платформерах та пригодницьких іграх завдяки своїй наочності.

Користувач може швидко оцінити поточний стан персонажа без необхідності аналізувати числові показники.

Рисунок 3.8 – Відображення запасу здоров’я персонажа

Реалізація лічильника монет

Для відображення кількості зібраних монет використовується окремий текстовий елемент інтерфейсу.

Після підбору монети система автоматично оновлює значення лічильника.

Завдяки цьому користувач постійно бачить результати своєї діяльності під час проходження гри.

Використання лічильника також створює додаткову мотивацію до дослідження ігрових локацій та боротьби з противниками.

Рисунок 3.9 – Відображення лічильника монет

Реалізація екрана завершення гри

Одним із важливих елементів інтерфейсу є екран завершення гри.

Він відображається після втрати персонажем усіх одиниць здоров’я.

Основним призначенням даного елемента є інформування користувача про завершення поточної спроби проходження.

Для підвищення якості взаємодії реалізовано плавну появу панелі Game Over.

Поява здійснюється шляхом поступової зміни прозорості елемента інтерфейсу.

Подібний підхід дозволяє уникнути різких змін зображення та робить завершення гри більш візуально приємним.

Рисунок 3.10 – Екран завершення гри

Реалізація системи затемнення екрана

Для організації переходів між сценами використовується спеціальна панель затемнення.

Даний елемент є частиною користувацького інтерфейсу та використовується під час завантаження нових локацій.

Поступова зміна прозорості дозволяє створити ефект плавного переходу між сценами.

Завдяки цьому користувач не спостерігає процес завантаження та отримує більш цілісний ігровий досвід.

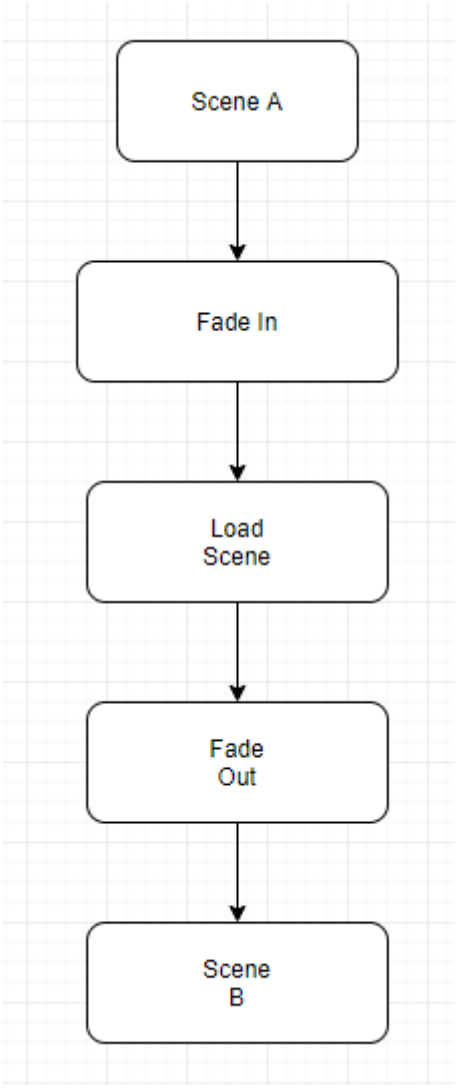


Рисунок 3.11 – Робота системи затемнення екрана

Взаємодія інтерфейсу з іншими підсистемами

Користувацький інтерфейс взаємодіє практично з усіма ключовими компонентами гри.

До них належать:

– система здоров’я;

– система збору монет;

– система переходів між сценами;

– система завершення гри.

Завдяки цьому користувач завжди отримує актуальну інформацію про поточний стан програмного продукту.

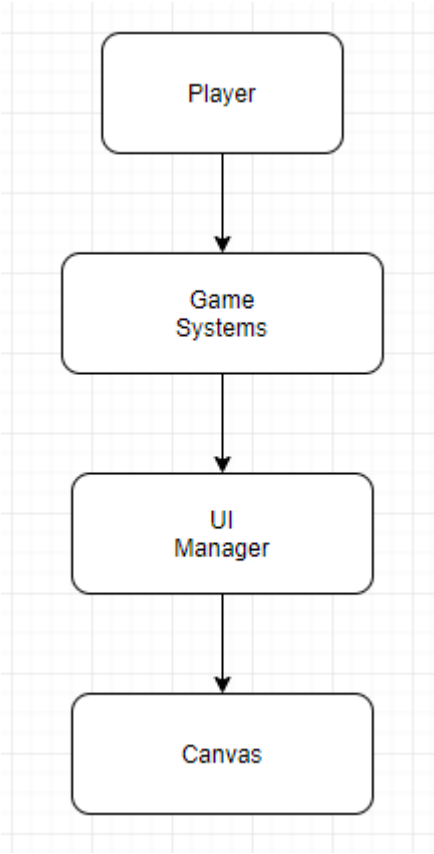


Рисунок 3.12 – Взаємодія UI з іншими компонентами гри

Переваги реалізованого інтерфейсу

Основними перевагами створеного інтерфейсу є:

– простота сприйняття;

– мінімалістичний дизайн;

– низьке навантаження на систему;

– швидкість оновлення інформації;

– зручність подальшого розширення.

Структура інтерфейсу дозволяє без значних змін додавати нові елементи та інтегрувати додаткові механіки.

Висновки до підрозділу

У результаті виконання даного етапу було реалізовано користувацький інтерфейс, який забезпечує відображення основної інформації про стан гри та персонажа.

Створена система містить індикатори здоров’я, лічильник монет, екран завершення гри та механізм затемнення екрана під час переходів між сценами.

Розроблений інтерфейс відповідає вимогам жанру Metroidvania та забезпечує зручну взаємодію користувача з програмним продуктом.

3.6 Тестування програмного продукту

Завершальним етапом розробки програмного забезпечення є тестування створеного програмного продукту. Основною метою тестування є перевірка коректності роботи реалізованих механік, виявлення можливих помилок та оцінка стабільності функціонування гри.

У межах даної дипломної роботи тестування проводилося після завершення реалізації основних підсистем програмного продукту.

Під час тестування перевірялася робота:

– системи керування персонажем;

– бойової системи;

– системи здоров’я;

– системи анімацій;

– системи збору монет;

– системи контрольних точок;

– системи переходів між сценами;

– користувацького інтерфейсу.

Мета тестування

Основною метою тестування було підтвердження працездатності програмного продукту та перевірка відповідності реалізованих механік початковим вимогам проєкту.

Для досягнення поставленої мети необхідно було перевірити:

– коректність виконання дій персонажа;

– правильність роботи противників;

– відсутність критичних помилок під час ігрового процесу;

– правильність взаємодії між окремими підсистемами.

Тестування системи керування персонажем

Першим етапом було проведено перевірку механік переміщення персонажа.

Під час тестування контролювалися:

– рух ліворуч;

– рух праворуч;

– стрибок;

– перекат;

– зміна напрямку руху.

Було встановлено, що персонаж коректно реагує на натискання клавіш керування та виконує всі передбачені дії.

Порушень у роботі системи керування виявлено не було.

Тестування бойової системи

Наступним етапом виконувалося тестування бойової системи.

Перевірялися:

– виконання атак;

– робота комбінацій ударів;

– нанесення пошкоджень противникам;

– отримання пошкоджень персонажем;

– механіка смерті противників.

У процесі тестування встановлено, що атаки коректно визначають зіткнення з противниками та завдають відповідне пошкодження.

Також підтверджено правильну роботу комбінації з двох ударів.

Критичних помилок у роботі бойової системи виявлено не було.

Тестування системи здоров’я

Для перевірки роботи системи здоров’я виконувалося багаторазове отримання пошкоджень персонажем.

У результаті тестування підтверджено:

– коректне оновлення індикаторів здоров’я;

– запуск анімації отримання пошкодження;

– правильне визначення моменту смерті персонажа;

– активацію екрана завершення гри.

Рисунок 3.13 – Тестування системи здоров’я

Система продемонструвала стабільну роботу за всіх перевірених сценаріїв.

Тестування системи збору монет

Для перевірки роботи колекційних предметів проводилося тестування механіки підбору монет.

У ході тестування перевірялися:

– створення монети після знищення противника;

– коректність підбору;

– оновлення лічильника монет;

– видалення об'єкта після взаємодії.

Було підтверджено правильну роботу всіх зазначених механізмів.

Тестування контрольних точок

Під час тестування контрольних точок перевірялася коректність їх активації та зміни візуального стану.

У результаті встановлено, що після контакту персонажа з контрольною точкою відбувається її успішна активація та зміна кольору спрайта.

Отримані результати підтвердили працездатність реалізованої системи.

Тестування переходів між сценами

Для перевірки роботи системи переходів виконувалося багаторазове переміщення між локаціями.

Під час тестування перевірялися:

– активація порталу;

– затемнення екрана;

– завантаження нової сцени;

– поява персонажа після переходу.

Проведені випробування показали стабільну роботу механізму зміни сцен.

Тестування користувацького інтерфейсу

Окремо виконувалася перевірка всіх елементів інтерфейсу.

Тестування включало:

– оновлення індикаторів здоров’я;

– зміну лічильника монет;

– відображення екрана завершення гри;

– роботу затемнення екрана.

Усі елементи інтерфейсу коректно реагували на зміни стану гри та відображали актуальну інформацію.

Результати тестування

За результатами проведеного тестування встановлено, що програмний продукт стабільно працює та коректно реалізує основні механіки жанру Metroidvania.

Критичних помилок, які унеможливлюють використання програмного продукту, виявлено не було.

Усі реалізовані підсистеми успішно пройшли перевірку та продемонстрували коректну взаємодію між собою.

Висновки до розділу

У межах розрахункового розділу було розглянуто особливості реалізації ключових компонентів програмного продукту.

Зокрема було описано систему штучного інтелекту противників, механізм збору монет, систему контрольних точок, реалізацію здібності Dash, організацію користувацького інтерфейсу та результати тестування гри.

Проведене тестування підтвердило працездатність створеного програмного продукту та відповідність реалізованих механік вимогам, поставленим на початковому етапі проєктування.

ОХОРОНА ПРАЦІ

4.1 Аналіз умов праці розробника програмного забезпечення

Розробка програмного забезпечення належить до видів діяльності, що характеризуються переважно розумовим навантаженням та тривалою роботою за комп'ютером. Незважаючи на відсутність значних фізичних навантажень, професійна діяльність програміста пов'язана з рядом факторів, які можуть негативно впливати на стан здоров'я працівника та його працездатність.

Основним робочим місцем розробника програмного забезпечення є комп'ютеризоване робоче місце, обладнане персональним комп'ютером, монітором, клавіатурою та іншими допоміжними пристроями.

Під час виконання професійних обов'язків працівник тривалий час перебуває у сидячому положенні, що створює додаткове навантаження на опорно-руховий апарат, зокрема на хребет, шийний відділ та м'язи спини.

Крім того, значне навантаження припадає на органи зору, оскільки більша частина робочого часу пов'язана зі спостереженням за інформацією на екрані монітора.

До основних шкідливих факторів, що можуть впливати на розробника програмного забезпечення, належать:

– тривала статична поза;

– перенапруження органів зору;

– нервово-емоційне навантаження;

– недостатня фізична активність;

– підвищене психічне навантаження;

– несприятливі параметри мікроклімату приміщення;

– недостатнє або надлишкове освітлення.

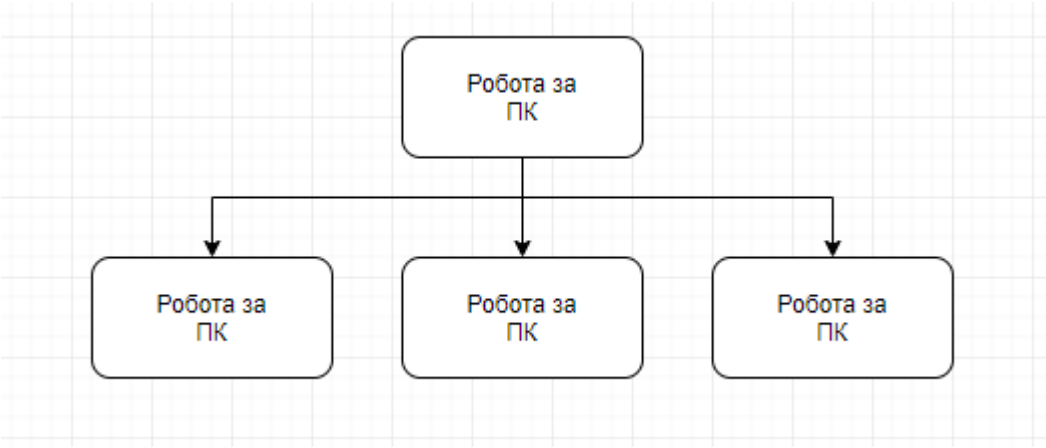


Рисунок 4.1 – Основні фактори впливу на розробника програмного забезпечення

Особливістю роботи програміста є необхідність тривалого зосередження уваги на вирішенні складних логічних задач. Це може призводити до підвищеного психологічного навантаження та швидкої розумової втоми.

Для забезпечення безпечних умов праці необхідно організувати робоче місце відповідно до чинних санітарних норм та вимог охорони праці.

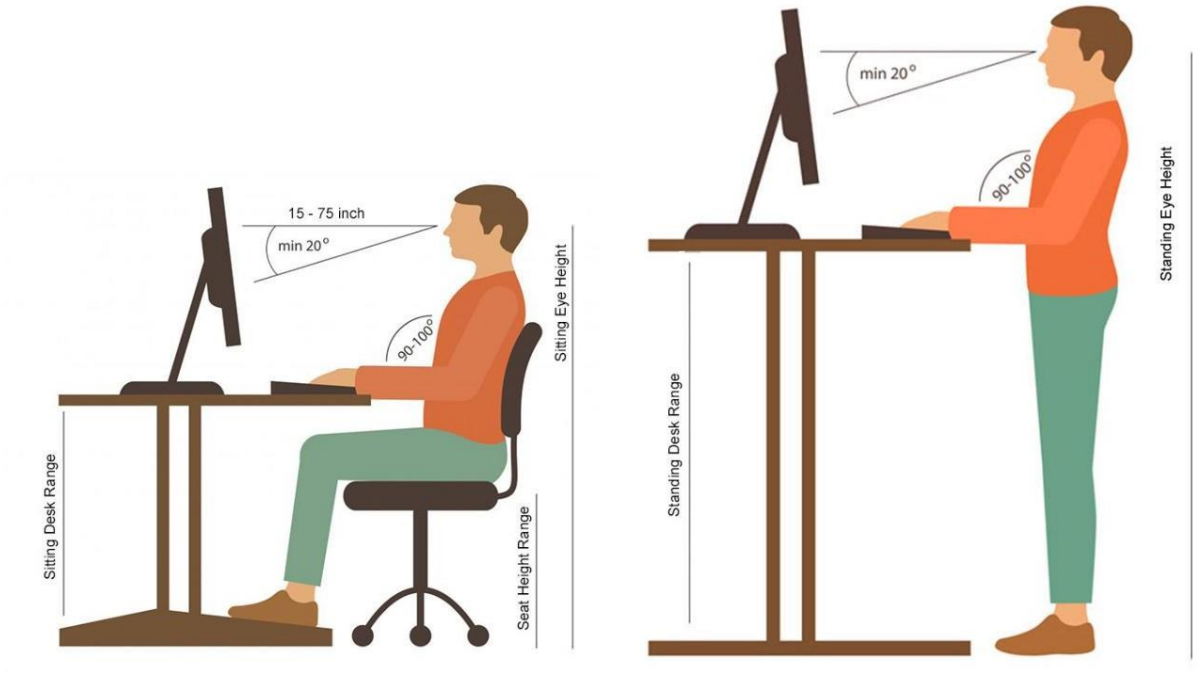
Робочий стіл повинен забезпечувати достатній простір для розміщення обладнання та комфортної роботи користувача.

Монітор рекомендується розташовувати на відстані 50–70 см від очей працівника. Верхня межа екрана повинна знаходитися приблизно на рівні очей або трохи нижче.

Клавіатура повинна бути розміщена таким чином, щоб руки працівника знаходилися у природному положенні без надмірного напруження м’язів.

Стілець повинен забезпечувати можливість регулювання висоти сидіння та нахилу спинки. Це дозволяє підтримувати правильне положення тіла під час роботи та зменшувати навантаження на хребет.

Рисунок 4.2 – Рекомендована організація робочого місця програміста



Для зниження негативного впливу на органи зору необхідно регулярно робити короткі перерви та виконувати спеціальні вправи для очей.

Також рекомендується дотримуватися правила «20–20–20», відповідно до якого кожні 20 хвилин необхідно переводити погляд на об’єкт, що знаходиться приблизно на відстані 20 футів (близько 6 метрів), протягом 20 секунд.

Значну роль відіграє правильна організація режиму праці та відпочинку. Регулярні перерви дозволяють знизити рівень втоми та підтримувати високу продуктивність праці протягом робочого дня.

Таким чином, діяльність розробника програмного забезпечення характеризується рядом факторів, що можуть негативно впливати на здоров’я працівника. Для забезпечення безпечних умов праці необхідно правильно організувати робоче місце, дотримуватися санітарних норм та забезпечувати оптимальний режим праці і відпочинку.

4.2 Освітлення робочого місця

Одним із найважливіших факторів, що впливають на працездатність розробника програмного забезпечення, є освітлення робочого місця. Недостатній рівень освітленості може призводити до швидкої втоми органів зору, зниження продуктивності праці та погіршення загального самопочуття працівника.

Оскільки професійна діяльність програміста пов’язана з постійною роботою за комп’ютером та обробкою великого обсягу інформації, питання організації освітлення робочого місця має особливе значення.

Основними вимогами до освітлення є:

– забезпечення достатнього рівня освітленості;

– рівномірний розподіл світлового потоку;

– відсутність сліпучої дії світильників;

– мінімізація відблисків на поверхні монітора;

– забезпечення комфортних умов для тривалої роботи.

Освітлення робочих приміщень може бути природним, штучним або комбінованим.

Природне освітлення

Природне освітлення забезпечується сонячним світлом, яке надходить через віконні прорізи.

Його основною перевагою є позитивний вплив на зорову систему людини та зниження втоми під час роботи.

При організації робочого місця бажано розташовувати монітор таким чином, щоб світло від вікна падало збоку від працівника.

Не рекомендується розміщувати екран безпосередньо навпроти вікна або спиною до нього, оскільки це може призводити до появи відблисків та погіршення видимості інформації.

Для регулювання рівня природного освітлення можуть використовуватися жалюзі або світлорозсіювальні штори.

Штучне освітлення

У вечірній час або за недостатнього природного освітлення використовується штучне освітлення.

Для приміщень, у яких виконуються роботи з використанням комп'ютерної техніки, рекомендується застосовувати світлодіодні світильники з нейтральною температурою світла.

Використання сучасних світлодіодних джерел освітлення дозволяє:

– забезпечити стабільний світловий потік;

– зменшити енергоспоживання;

– уникнути мерехтіння світла;

– створити комфортні умови праці.

Загальне освітлення повинно забезпечувати рівномірний розподіл світла по всій площі приміщення.

Крім загального освітлення, за необхідності можуть використовуватися локальні джерела світла.

Нормування освітленості

Згідно з рекомендаціями щодо організації робочих місць користувачів персональних комп'ютерів, освітленість робочої поверхні повинна становити приблизно 300–500 лк.

Такого рівня освітлення достатньо для комфортної роботи з текстовою інформацією, програмним кодом та графічними редакторами.

Недостатня освітленість може призводити до:

– підвищеної втоми очей;

– погіршення концентрації уваги;

– зниження швидкості роботи;

– появи головного болю.

Надлишкове освітлення також негативно впливає на працездатність людини та може викликати дискомфорт через надмірну яскравість.

Відблиски та їх вплив на роботу

Однією з найбільш поширених проблем під час роботи за комп’ютером є поява відблисків на екрані монітора.

Причинами виникнення відблисків можуть бути:

– пряме сонячне світло;

– неправильно розташовані світильники;

– глянцеві поверхні в приміщенні.

Для усунення даної проблеми рекомендується:

– використовувати монітори з антивідблисковим покриттям;

– правильно розташовувати робоче місце;

– застосовувати жалюзі або штори;

– використовувати світильники з розсіювачами світла.

Зменшення кількості відблисків позитивно впливає на якість сприйняття інформації та знижує навантаження на органи зору.

Висновки до підрозділу

Правильно організоване освітлення робочого місця є важливим фактором забезпечення безпечних та комфортних умов праці розробника програмного забезпечення.

Поєднання природного та штучного освітлення, дотримання нормативних значень освітленості та запобігання появі відблисків дозволяють знизити втому працівника, підвищити продуктивність праці та забезпечити комфортну роботу за комп’ютером протягом тривалого часу.

4.3 Мікроклімат приміщення

Важливим фактором забезпечення безпечних та комфортних умов праці розробника програмного забезпечення є мікроклімат робочого приміщення. Параметри мікроклімату безпосередньо впливають на самопочуття працівника, його працездатність та ефективність виконання професійних обов'язків.

Під мікрокліматом приміщення розуміють сукупність фізичних параметрів внутрішнього середовища, які впливають на теплообмін організму людини з навколишнім середовищем.

До основних параметрів мікроклімату належать:

– температура повітря;

– відносна вологість повітря;

– швидкість руху повітря;

– температура поверхонь навколишніх предметів.

Для працівників, діяльність яких пов'язана з виконанням робіт за персональним комп'ютером, створення оптимальних параметрів мікроклімату є особливо важливим через тривале перебування у закритому приміщенні.

Температура повітря

Температура повітря є одним із головних факторів, що визначають комфортність умов праці.

Для приміщень, у яких виконуються роботи, пов'язані з програмуванням та обробкою інформації, рекомендується підтримувати температуру в межах 22–24 °C у холодний період року та 23–25 °C у теплий період.

За надто низької температури працівник може відчувати дискомфорт, зниження концентрації уваги та підвищену втому.

Надмірно висока температура також негативно впливає на працездатність, викликаючи сонливість, погіршення концентрації та загальне зниження продуктивності праці.

Відносна вологість повітря

Вологість повітря визначає кількість водяної пари, що міститься в повітрі робочого приміщення.

Для комфортної роботи користувачів персональних комп'ютерів рекомендована відносна вологість становить 40–60 %.

Знижена вологість може викликати:

– сухість слизових оболонок;

– подразнення очей;

– погіршення самопочуття;

– підвищення втоми.

Надмірна вологість також негативно впливає на комфорт працівника та може сприяти розвитку грибкових мікроорганізмів у приміщенні.

Для підтримання оптимальної вологості можуть використовуватися системи кондиціонування та зволожувачі повітря.

Швидкість руху повітря

Швидкість руху повітря впливає на тепловий комфорт людини.

Для офісних приміщень рекомендоване значення швидкості руху повітря становить не більше 0,1–0,2 м/с.

Надмірний рух повітря може викликати відчуття протягів та дискомфорт під час роботи.

Водночас повна відсутність циркуляції повітря також небажана, оскільки це може призводити до погіршення якості повітря в приміщенні.

Для підтримання оптимальних параметрів використовуються системи вентиляції та кондиціонування.

Організація вентиляції приміщення

Однією з необхідних умов забезпечення сприятливого мікроклімату є наявність ефективної вентиляції.

Основними завданнями вентиляції є:

– подача свіжого повітря;

– видалення надлишкового тепла;

– підтримання нормативної вологості;

– забезпечення комфортних умов праці.

У сучасних офісних приміщеннях можуть використовуватися природна або механічна вентиляція.

Найбільш ефективним рішенням є використання комбінованих систем вентиляції та кондиціонування повітря.

Вплив мікроклімату на продуктивність праці

Параметри мікроклімату безпосередньо впливають на ефективність роботи програміста.

Комфортні умови праці сприяють:

– підвищенню концентрації уваги;

– зменшенню рівня втоми;

– покращенню самопочуття;

– збільшенню продуктивності праці.

Недотримання нормативних параметрів може призводити до погіршення результатів роботи та підвищення ризику виникнення професійних захворювань.

Саме тому забезпечення оптимального мікроклімату є одним із важливих завдань охорони праці в сфері інформаційних технологій.

Висновки до підрозділу

Оптимальні параметри мікроклімату є необхідною умовою ефективної роботи розробника програмного забезпечення.

Підтримання нормативних значень температури, вологості та швидкості руху повітря дозволяє забезпечити комфортні умови праці, знизити втому працівника та підвищити продуктивність виконання професійних завдань.

Застосування сучасних систем вентиляції та кондиціонування повітря сприяє підтриманню необхідних параметрів мікроклімату та забезпечує безпечні умови праці у приміщеннях, де здійснюється розробка програмного забезпечення.

4.4 Електробезпека

Під час виконання професійної діяльності розробник програмного забезпечення постійно працює з електронним обладнанням, яке підключене до електричної мережі. Незважаючи на відносно низький рівень ризику у порівнянні з виробничими підприємствами, недотримання правил електробезпеки може призвести до травмування працівника, пошкодження обладнання або виникнення аварійних ситуацій.

Електробезпека являє собою систему організаційних та технічних заходів, спрямованих на захист людини від небезпечного впливу електричного струму, електричної дуги, електромагнітного поля та статичної електрики.

У процесі розробки програмного забезпечення основними джерелами електричної енергії є:

– персональний комп’ютер;

– монітор;

– мережеве обладнання;

– периферійні пристрої;

– джерела безперебійного живлення;

– системи освітлення робочого місця.

Основні небезпечні фактори

Під час роботи з електричним обладнанням можливий вплив наступних небезпечних факторів:

– ураження електричним струмом;

– пошкодження ізоляції електропроводки;

– коротке замикання;

– перегрів електрообладнання;

– виникнення пожежі;

– накопичення статичної електрики.

Найбільш небезпечним фактором є безпосередній контакт людини зі струмопровідними частинами електрообладнання.

Навіть відносно невеликі значення сили струму можуть негативно впливати на організм людини та викликати порушення роботи серцево-судинної системи.

Вимоги до електрообладнання

Усе обладнання, що використовується під час розробки програмного забезпечення, повинно відповідати вимогам чинних стандартів електробезпеки.

До основних вимог належать:

– справний технічний стан обладнання;

– відсутність пошкоджень ізоляції;

– наявність захисного заземлення;

– використання сертифікованих джерел живлення;

– відповідність електромережі встановленим нормам.

Особливу увагу необхідно приділяти стану електричних кабелів та мережевих фільтрів.

Пошкоджена ізоляція може стати причиною короткого замикання або ураження працівника електричним струмом.

Захисне заземлення

Одним із найбільш ефективних засобів забезпечення електробезпеки є захисне заземлення.

Заземлення використовується для відведення небезпечного електричного потенціалу від корпусів обладнання до землі.

У випадку виникнення несправності це дозволяє зменшити ризик ураження людини електричним струмом.

У сучасних комп’ютерних системах використання захисного заземлення є обов’язковою умовою безпечної експлуатації обладнання.

Захист від короткого замикання

Коротке замикання є одним із найпоширеніших видів аварійних ситуацій в електромережах.

Основними причинами виникнення короткого замикання можуть бути:

– пошкодження ізоляції;

– потрапляння вологи;

– перевантаження електромережі;

– несправність обладнання.

Для захисту від коротких замикань використовуються:

– автоматичні вимикачі;

– запобіжники;

– мережеві фільтри;

– джерела безперебійного живлення.

Застосування зазначених засобів дозволяє зменшити ризик пошкодження обладнання та виникнення пожежі.

Статична електрика

Під час роботи комп’ютерного обладнання можливе накопичення статичної електрики.

Особливо часто дане явище спостерігається в приміщеннях із низькою вологістю повітря.

Статичні заряди можуть негативно впливати на електронні компоненти та викликати збої в роботі обладнання.

Для зменшення рівня статичної електрики рекомендується:

– підтримувати оптимальну вологість повітря;

– використовувати антистатичні матеріали;

– забезпечувати належне заземлення обладнання.

Дії у разі ураження електричним струмом

У випадку ураження людини електричним струмом необхідно діяти швидко та відповідно до встановлених правил безпеки.

Основні дії включають:

– негайне відключення джерела живлення;

– звільнення потерпілого від дії струму;

– оцінку стану потерпілого;

– виклик медичної допомоги;

– надання домедичної допомоги при необхідності.

Своєчасне виконання зазначених дій дозволяє мінімізувати наслідки нещасного випадку та підвищує шанси на успішне відновлення потерпілого.

Висновки до підрозділу

Електробезпека є важливою складовою організації робочого місця розробника програмного забезпечення.

Дотримання вимог до експлуатації електрообладнання, використання захисного заземлення, контроль технічного стану електромереж та знання правил поведінки у надзвичайних ситуаціях дозволяють забезпечити безпечні умови праці та знизити ризик виникнення аварійних ситуацій під час роботи за персональним комп’ютером.

4.5 Пожежна безпека

Пожежна безпека є важливою складовою системи охорони праці та спрямована на запобігання виникненню пожеж, захист життя та здоров’я працівників, а також збереження матеріальних цінностей.

Незважаючи на те, що діяльність розробника програмного забезпечення не пов’язана з використанням відкритого вогню або горючих технологічних процесів, ризик виникнення пожежі повністю виключити неможливо.

Основними причинами виникнення пожеж у приміщеннях, де використовується комп’ютерна техніка, можуть бути:

– коротке замикання електромережі;

– перевантаження електропроводки;

– несправність електрообладнання;

– пошкодження ізоляції кабелів;

– неправильна експлуатація електроприладів;

– використання несправних подовжувачів або мережевих фільтрів.

Категорія приміщення за пожежною небезпекою

Робоче приміщення, в якому здійснюється розробка програмного забезпечення, належить до приміщень з відносно низьким рівнем пожежної небезпеки.

Основними горючими матеріалами є:

– пластик корпусів обладнання;

– ізоляція кабелів;

– меблі;

– паперова документація;

– пакувальні матеріали.

Незважаючи на невелику кількість горючих речовин, наявність великої кількості електрообладнання потребує дотримання встановлених вимог пожежної безпеки.

Основні причини виникнення пожеж

Однією з найпоширеніших причин виникнення пожеж у комп'ютерних класах та офісних приміщеннях є несправність електричного обладнання.

Перегрівання блоків живлення, пошкодження кабелів або короткі замикання можуть призвести до займання окремих елементів обладнання.

Для мінімізації ризику виникнення пожеж необхідно регулярно проводити перевірку технічного стану електрообладнання та електромереж.

Засоби пожежогасіння

Для ліквідації можливих осередків займання в приміщенні повинні бути передбачені первинні засоби пожежогасіння.

До них належать:

– порошкові вогнегасники;

– вуглекислотні вогнегасники;

– пожежні щити;

– системи пожежної сигналізації.

Для приміщень із комп’ютерною технікою найбільш доцільним є використання вуглекислотних вогнегасників.

Основною перевагою даного типу вогнегасників є можливість гасіння електрообладнання без його значного пошкодження.

Евакуація працівників

У разі виникнення пожежі необхідно забезпечити швидку та безпечну евакуацію працівників.

Для цього приміщення повинно бути обладнане:

– евакуаційними виходами;

– покажчиками напрямків руху;

– планом евакуації;

– аварійним освітленням.

Працівники повинні бути ознайомлені з маршрутом евакуації та порядком дій у надзвичайних ситуаціях.

Профілактика пожеж

Для забезпечення належного рівня пожежної безпеки необхідно дотримуватися наступних заходів:

– регулярно перевіряти стан електромережі;

– не допускати перевантаження розеток;

– використовувати лише справне обладнання;

– підтримувати порядок на робочому місці;

– забезпечувати вільний доступ до засобів пожежогасіння.

Виконання зазначених заходів дозволяє суттєво знизити ймовірність виникнення пожежі та підвищити загальний рівень безпеки приміщення.

Висновки до підрозділу

Пожежна безпека є важливою складовою організації робочого місця розробника програмного забезпечення.

Основними джерелами пожежної небезпеки є електрообладнання та електромережа. Дотримання вимог експлуатації технічних засобів, використання первинних засобів пожежогасіння та наявність ефективної системи евакуації дозволяють забезпечити належний рівень безпеки працівників та мінімізувати наслідки можливих надзвичайних ситуацій.

4.6 Режим праці та відпочинку

Раціональна організація режиму праці та відпочинку є важливим фактором забезпечення безпечних умов праці розробника програмного забезпечення. Робота програміста пов’язана з тривалим використанням персонального комп’ютера, високою концентрацією уваги та значними розумовими навантаженнями.

Недотримання оптимального режиму праці та відпочинку може призводити до перевтоми, зниження продуктивності праці, погіршення стану здоров’я та виникнення професійних захворювань.

Основною метою організації режиму праці та відпочинку є:

– підтримання високої працездатності працівника;

– запобігання розвитку перевтоми;

– зменшення навантаження на органи зору;

– профілактика захворювань опорно-рухового апарату;

– забезпечення комфортних умов праці.

Особливості праці розробника програмного забезпечення

Професійна діяльність програміста належить до робіт із переважним розумовим навантаженням.

Протягом робочого дня працівник виконує:

– розробку програмного коду;

– аналіз інформації;

– пошук та усунення помилок;

– тестування програмного забезпечення;

– роботу з технічною документацією.

Виконання зазначених завдань потребує високого рівня концентрації уваги та значних інтелектуальних зусиль.

Тривала безперервна робота за комп'ютером може викликати:

– втому очей;

– головний біль;

– зниження концентрації уваги;

– дискомфорт у шиї та спині;

– загальну втому організму.

Саме тому під час організації роботи необхідно передбачати регулярні перерви для відпочинку.

Регламентовані перерви

Для працівників, які більшу частину робочого часу використовують персональний комп'ютер, рекомендується організовувати регламентовані перерви.

Під час перерв рекомендується:

– виконувати вправи для очей;

– змінювати положення тіла;

– виконувати легку фізичну активність;

– здійснювати короткі прогулянки приміщенням.

Регулярні перерви дозволяють зменшити рівень втоми та підтримувати стабільну працездатність протягом робочого дня.

Профілактика зорової втоми

Одним із найбільш навантажених органів під час роботи за комп’ютером є зоровий апарат.

Для профілактики зорової втоми рекомендується:

– підтримувати оптимальну відстань до монітора;

– використовувати достатній рівень освітлення;

– регулярно виконувати вправи для очей;

– уникати відблисків на поверхні екрана.

Широко використовується правило «20–20–20».

Відповідно до даного правила кожні 20 хвилин необхідно переводити погляд на об’єкт, який знаходиться приблизно на відстані 20 футів (близько 6 метрів), протягом 20 секунд.

Дотримання даних рекомендацій дозволяє знизити навантаження на органи зору та зменшити ризик виникнення професійних захворювань.

Профілактика захворювань опорно-рухового апарату

Тривале перебування у сидячому положенні може негативно впливати на стан хребта та м’язів спини.

Для профілактики захворювань опорно-рухового апарату рекомендується:

– використовувати ергономічні меблі;

– підтримувати правильну поставу;

– виконувати розминку під час перерв;

– періодично змінювати положення тіла.

Регулярна фізична активність позитивно впливає на загальний стан здоров’я працівника та сприяє підтриманню високої працездатності.

Організація робочого дня

Раціональний режим праці передбачає правильне планування робочого часу.

Найбільш складні завдання рекомендується виконувати в періоди максимальної працездатності.

Також доцільно чергувати різні види діяльності для зменшення монотонності роботи.

Подібний підхід дозволяє підтримувати високу концентрацію уваги та підвищувати ефективність виконання професійних обов’язків.

Висновки до підрозділу

Раціональна організація режиму праці та відпочинку є важливою умовою забезпечення безпечної та ефективної роботи розробника програмного забезпечення.

Регулярні перерви, профілактика зорової втоми, підтримання фізичної активності та правильне планування робочого часу дозволяють зменшити негативний вплив професійної діяльності на здоров’я працівника та забезпечують високий рівень працездатності протягом усього робочого дня.

ОРГАНІЗАЦІЙНО-ЕКОНОМІЧНИЙ РОЗДІЛ

5.1 Обґрунтування доцільності розробки

Ринок комп’ютерних ігор є однією з найбільш динамічних галузей сучасної цифрової економіки. Щороку збільшується кількість користувачів персональних комп’ютерів та мобільних пристроїв, а попит на інтерактивні розважальні продукти продовжує стабільно зростати.

Особливе місце серед сучасних комп’ютерних ігор займають проєкти жанру Metroidvania. Даний жанр поєднує елементи платформера, пригодницької гри та дослідження великого взаємопов’язаного світу. Популярність жанру пояснюється високим рівнем залучення користувача в ігровий процес та можливістю поступового розвитку персонажа.

Серед найбільш відомих представників жанру можна виділити такі проєкти:

– Hollow Knight;

– Ori and the Blind Forest;

– Dead Cells;

– Bloodstained: Ritual of the Night.

Успішність зазначених проєктів демонструє високий інтерес користувачів до ігор даного жанру та підтверджує перспективність створення подібних програмних продуктів.

Розробка комп’ютерної гри у жанрі Metroidvania є доцільною з декількох причин.

По-перше, подібні проєкти мають відносно невисокі вимоги до апаратних ресурсів порівняно із сучасними тривимірними іграми, що дозволяє зменшити витрати на розробку та розширити коло потенційних користувачів.

По-друге, використання рушія Unity значно прискорює процес створення програмного продукту завдяки наявності великої кількості вбудованих інструментів для роботи з графікою, фізикою, анімацією та користувацьким інтерфейсом.

По-третє, двовимірний формат гри дозволяє реалізувати повноцінний ігровий проєкт навіть невеликій команді розробників або окремому розробнику.

Розроблений програмний продукт може використовуватися як:

– самостійна комп’ютерна гра;

– навчальний проєкт для вивчення технологій розробки ігор;

– демонстраційний проєкт для формування портфоліо розробника;

– основа для подальшого розвитку комерційного програмного продукту.

Використання рушія Unity також є економічно обґрунтованим рішенням. Безкоштовна версія рушія містить усі необхідні інструменти для створення повноцінних двовимірних ігор та дозволяє суттєво скоротити початкові витрати на розробку.

Крім того, жанр Metroidvania характеризується високою популярністю серед користувачів цифрових платформ розповсюдження ігор. Це створює передумови для подальшого розвитку проєкту та потенційної комерціалізації після завершення основного етапу розробки.

Створений програмний продукт дозволяє отримати практичний досвід розробки ігрових механік, роботи з рушієм Unity, створення користувацького інтерфейсу, реалізації штучного інтелекту противників та організації взаємодії між різними підсистемами гри.

Таким чином, розробка комп’ютерної гри у жанрі Metroidvania з використанням рушія Unity є технічно та економічно доцільною. Створений програмний продукт може бути використаний як основа для подальшого розвитку повноцінного ігрового проєкту та має потенціал для подальшого вдосконалення і можливого комерційного використання.

Висновки до підрозділу

Проведений аналіз показав, що розробка комп’ютерної гри у жанрі Metroidvania є актуальним та перспективним напрямком. Використання рушія Unity дозволяє скоротити витрати на створення програмного продукту, а популярність жанру створює передумови для подальшого розвитку та можливого комерційного використання розробленої гри.

5.2 Оцінка трудомісткості розробки

Трудомісткість розробки програмного продукту є одним із основних показників, що характеризують обсяг виконаних робіт та витрати часу на створення програмного забезпечення.

Оцінка трудомісткості дозволяє визначити загальний обсяг ресурсів, необхідних для реалізації проєкту, а також використовується під час подальших економічних розрахунків.

Розробка комп’ютерної гри у жанрі Metroidvania включала виконання комплексу робіт, пов’язаних із проєктуванням архітектури програмного продукту, створенням ігрових механік, реалізацією графічної складової, тестуванням та підготовкою документації.

До основних етапів розробки належать:

– аналіз предметної області;

– проєктування структури програмного продукту;

– створення ігрових механік;

– програмування основних підсистем;

– створення користувацького інтерфейсу;

– тестування програмного продукту;

– підготовка документації.

Для оцінки трудомісткості кожного етапу було визначено приблизний обсяг часу, необхідний для виконання відповідних робіт.

Таблиця 5.1 – Оцінка трудомісткості розробки

	Етап розробки	Трудомісткість, год
	Аналіз предметної області	12
	Проєктування архітектури гри	18
	Створення структури проєкту Unity	10
	Реалізація системи керування персонажем	30
	Реалізація бойової системи	24
	Реалізація системи анімацій	16
	Реалізація штучного інтелекту противників	20
	Реалізація системи збору монет	8
	Реалізація контрольних точок	8
0	Реалізація системи переходів між сценами	12
1	Розробка користувацького інтерфейсу	12
2	Тестування програмного продукту	20
3	Підготовка документації	20
	Разом	210

Отримані результати свідчать про те, що найбільший обсяг часу був витрачений на реалізацію основних ігрових механік, які формують основу програмного продукту.

Найбільш трудомісткими етапами стали:

– реалізація системи керування персонажем;

– створення бойової системи;

– розробка штучного інтелекту противників;

– тестування програмного продукту.

Подібний розподіл часу є характерним для розробки комп’ютерних ігор, оскільки саме ігрові механіки визначають якість та функціональні можливості програмного продукту.

Отримана оцінка трудомісткості дозволяє перейти до визначення економічних показників розробки та виконати розрахунок вартості створення програмного продукту.

Висновки до підрозділу

У результаті проведеної оцінки встановлено, що загальна трудомісткість розробки комп’ютерної гри становить приблизно 210 людино-годин.

Найбільшу частку витрат часу займає програмна реалізація основних ігрових механік та тестування програмного продукту. Отримані результати можуть бути використані для подальших розрахунків вартості розробки та економічної ефективності створеного програмного продукту.

5.3 Розрахунок вартості розробки

Вартість розробки програмного продукту визначається на основі витрат трудових ресурсів, використання програмного забезпечення, комп’ютерної техніки та інших ресурсів, необхідних для виконання робіт.

Для оцінки вартості створення комп’ютерної гри у жанрі Metroidvania використовується трудомісткість, визначена в попередньому підрозділі.

Загальна трудомісткість розробки становить:

Т = 210 людино-годин.

Для виконання розрахунків приймемо середню погодинну оплату праці розробника програмного забезпечення на рівні 250 грн за годину.

Тоді основна заробітна плата розробника визначається за формулою:

Зосн = Т × С,

де:

Зосн – основна заробітна плата;

Т – трудомісткість розробки, год;

С – погодинна ставка розробника, грн/год.

Підставляючи значення, отримаємо:

Зосн = 210 × 250 = 52 500 грн.

Крім основної заробітної плати необхідно врахувати додаткові витрати, пов’язані з використанням обладнання, електроенергії та програмного забезпечення.

Для спрощеного розрахунку приймемо їх у розмірі 20 % від основної заробітної плати.

Додаткові витрати:

Здод = 0,2 × 52 500 = 10 500 грн.

Тоді загальні витрати на розробку складуть:

Ззаг = Зосн + Здод

Ззаг = 52 500 + 10 500 = 63 000 грн.

Таблиця 5.2 – Структура витрат на розробку

	Стаття витрат	Сума, грн
	Основна заробітна плата	52 500

	Стаття витрат	Сума, грн
	Витрати на обладнання та електроенергію	10 500
	Разом	**63 000**

Отримане значення характеризує орієнтовну собівартість створення програмного продукту на етапі розробки.

Слід зазначити, що наведений розрахунок є орієнтовним та може змінюватися залежно від складності проєкту, тривалості розробки, кількості учасників команди та інших факторів.

Висновки до підрозділу

У результаті проведених розрахунків встановлено, що орієнтовна вартість розробки комп’ютерної гри у жанрі Metroidvania становить 63 000 грн.

Основну частину витрат складає оплата праці розробника, що є характерним для більшості проєктів у сфері програмної інженерії.

5.4 Розрахунок заробітної плати розробника

Заробітна плата є однією з основних складових вартості створення програмного забезпечення. У сфері інформаційних технологій саме витрати на оплату праці розробників формують найбільшу частину собівартості програмних продуктів.

Для визначення заробітної плати розробника необхідно врахувати трудомісткість виконаних робіт та погодинну оплату праці.

У попередньому підрозділі було визначено, що загальна трудомісткість розробки становить:

Т = 210 людино-годин.

Для виконання розрахунків приймається середня погодинна ставка розробника програмного забезпечення:

С = 250 грн/год.

Основна заробітна плата визначається за формулою:

Зосн = Т × С,

де:

Зосн – основна заробітна плата розробника;

Т – трудомісткість виконаних робіт;

С – погодинна ставка оплати праці.

Після підстановки значень отримаємо:

Зосн = 210 × 250 = 52 500 грн.

Крім основної заробітної плати працівник може отримувати додаткову оплату за виконання окремих робіт, премії або компенсаційні виплати.

Для розрахунків приймемо додаткову заробітну плату на рівні 15 % від основної.

Додаткова заробітна плата визначається за формулою:

Здод = Зосн × 0,15

Здод = 52 500 × 0,15 = 7 875 грн.

Загальна заробітна плата становитиме:

Ззп = Зосн + Здод

Ззп = 52 500 + 7 875 = 60 375 грн.

Таблиця 5.3 – Розрахунок заробітної плати розробника

	Показник	Значення
	Основна заробітна плата	52 500 грн
	Додаткова заробітна плата	7 875 грн
	Разом	**60 375 грн**

Аналіз отриманих результатів показує, що основна частина виплат припадає на основну заробітну плату, яка безпосередньо залежить від обсягу виконаних робіт та складності програмного продукту.

Подібна структура витрат є характерною для більшості проєктів у сфері розробки програмного забезпечення.

Використання погодинної системи розрахунку дозволяє досить точно оцінювати витрати на створення програмного продукту та використовувати отримані результати під час планування майбутніх проєктів.

Висновки до підрозділу

У результаті проведених розрахунків встановлено, що загальна заробітна плата розробника під час створення комп’ютерної гри у жанрі Metroidvania становить 60 375 грн.

Отримані результати підтверджують, що витрати на оплату праці є основною складовою собівартості програмного продукту та мають вирішальний вплив на загальну вартість розробки.

5.5 Оцінка економічної ефективності

Оцінка економічної ефективності є завершальним етапом організаційно-економічного аналізу розробленого програмного продукту. Вона дозволяє визначити доцільність витрат на створення програмного забезпечення та оцінити перспективи його подальшого використання.

Для комерційних програмних продуктів економічна ефективність зазвичай визначається шляхом порівняння витрат на розробку та можливого прибутку від реалізації продукту.

У випадку даної дипломної роботи розроблений програмний продукт являє собою навчально-дослідницький проєкт, який демонструє практичні навички створення комп’ютерних ігор із використанням рушія Unity.

Разом із тим створена гра має потенціал для подальшого розвитку та комерційного використання після завершення процесу розробки.

Потенційні напрями використання програмного продукту

Розроблена комп’ютерна гра може використовуватися в декількох напрямках:

– як самостійний ігровий продукт;

– як демонстраційний проєкт для формування портфоліо розробника;

– як навчальний приклад для вивчення технологій розробки ігор;

– як основа для створення більш масштабного комерційного проєкту.

Наявність готової архітектури програмного продукту значно скорочує обсяг робіт, необхідних для подальшого розвитку гри.

Переваги використання рушія Unity

Важливим фактором економічної ефективності є використання сучасного рушія Unity.

Основними перевагами Unity є:

– безкоштовна версія для індивідуальних розробників;

– велика кількість навчальних матеріалів;

– підтримка багатьох платформ;

– широкий набір готових інструментів;

– можливість швидкого масштабування проєкту.

Завдяки використанню Unity вдалося зменшити витрати часу на реалізацію основних підсистем гри та скоротити загальну вартість розробки.

Оцінка перспектив розвитку проєкту

Однією з переваг створеного програмного продукту є можливість його подальшого вдосконалення.

У майбутньому до гри можуть бути додані:

– нові ігрові локації;

– додаткові противники;

– система розвитку персонажа;

– нові здібності;

– сюжетна складова;

– система збережень;

– боси та додаткові ігрові механіки.

Подібне розширення функціональних можливостей дозволить підвищити конкурентоспроможність програмного продукту та збільшити його потенційну цінність для користувачів.

Аналіз отриманих результатів

У попередніх підрозділах було визначено:

– трудомісткість розробки – 210 людино-годин;

– орієнтовну вартість розробки – 63 000 грн;

– загальну заробітну плату розробника – 60 375 грн.

Отримані показники свідчать про те, що створення програмного продукту не потребує значних фінансових вкладень порівняно з великими комерційними ігровими проєктами.

Застосування сучасних інструментів розробки дозволяє одному розробнику створювати повноцінні ігрові продукти з відносно невеликими витратами.

Загальна оцінка економічної доцільності

Аналіз отриманих результатів дозволяє зробити висновок про економічну доцільність виконаної розробки.

Створений програмний продукт має практичну цінність, може використовуватися як основа для подальшого розвитку та дозволяє отримати значний досвід у сфері створення комп’ютерних ігор.

Використання рушія Unity забезпечує скорочення витрат на розробку та спрощує процес реалізації основних механік гри.

У разі подальшого розвитку проєкту існує можливість його адаптації для комерційного використання та розповсюдження через цифрові платформи.

Висновки до підрозділу

Проведена оцінка економічної ефективності показала, що розробка комп’ютерної гри у жанрі Metroidvania з використанням рушія Unity є економічно доцільною.

Використання сучасних інструментів розробки дозволило зменшити витрати часу та ресурсів на створення програмного продукту. Створена гра має потенціал для подальшого розвитку та може бути використана як основа для майбутніх ігрових проєктів.

ВИСНОВКИ

У результаті виконання дипломної роботи на тему «Розробка комп’ютерної гри у жанрі метроідванія з використанням рушія Unity» було досягнуто поставленої мети та вирішено основні завдання, пов’язані зі створенням програмного продукту.

Під час виконання роботи було проведено аналіз сучасних підходів до розробки комп’ютерних ігор та досліджено особливості жанру Metroidvania. Встановлено, що даний жанр залишається популярним серед користувачів завдяки поєднанню платформних механік, дослідження ігрового світу та поступового розвитку персонажа.

Для реалізації програмного продукту було обрано рушій Unity, який надає широкий набір інструментів для створення двовимірних ігор та забезпечує ефективну організацію процесу розробки. Використання Unity дозволило реалізувати необхідні механіки гри та забезпечити можливість подальшого розширення функціоналу програмного продукту.

У процесі виконання дипломної роботи було розроблено архітектуру програмного продукту та створено структуру проєкту, яка забезпечує взаємодію між окремими підсистемами гри. Реалізовано систему керування персонажем, що включає переміщення, стрибки та використання спеціальної здібності Dash.

Також було створено бойову систему, яка забезпечує взаємодію між персонажем та противниками. Реалізовано механіку атак, отримання пошкоджень, систему здоров’я персонажа та механізм знищення противників.

Важливою частиною програмного продукту стала система анімацій, яка забезпечує візуальне відображення дій персонажа та противників. Для цього було використано можливості Animator Controller, що дозволило

реалізувати анімації руху, стрибків, атак, отримання пошкоджень та смерті.

У межах роботи було реалізовано систему штучного інтелекту противників на основі кінцевого автомата станів. Створена система забезпечує патрулювання території, виявлення персонажа, переслідування та виконання атак. Використання такого підходу дозволило створити логічну та передбачувану поведінку противників.

Додатково було реалізовано систему збору монет, механізм контрольних точок, систему переходів між сценами та користувацький інтерфейс, який забезпечує відображення основної інформації про стан гри.

На завершальному етапі виконано тестування програмного продукту. Проведені перевірки підтвердили працездатність основних механік гри та коректну взаємодію між окремими підсистемами. Критичних помилок, що унеможливлюють використання програмного продукту, виявлено не було.

У межах розділу «Охорона праці» було розглянуто питання організації безпечних умов праці розробника програмного забезпечення. Проаналізовано вимоги до освітлення робочого місця, параметрів мікроклімату, електробезпеки, пожежної безпеки та режиму праці й відпочинку.

В організаційно-економічному розділі виконано оцінку трудомісткості розробки програмного продукту, визначено орієнтовну вартість створення гри та проведено аналіз економічної доцільності виконаної роботи. Отримані результати свідчать про можливість створення повноцінного ігрового продукту з використанням сучасних інструментів розробки без значних фінансових витрат.

Таким чином, поставлена мета дипломної роботи була досягнута. Розроблено комп’ютерну гру у жанрі Metroidvania з використанням рушія Unity, яка реалізує основні механіки жанру та може слугувати основою для подальшого розвитку, вдосконалення та розширення функціональних можливостей програмного продукту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Unity Technologies. Unity User Manual. URL: https://docs.unity3d.com/Manual/index.html (дата звернення 15.04.2026)

2. Microsoft. C# Documentation. URL: https://learn.microsoft.com/dotnet/csharp (дата звернення 18.04.2026)

3. Unity Learn. Official Learning Platform. URL: https://learn.unity.com (дата звернення 24.04.2026)

4. Hollow Knight Official Website. URL: https://www.hollowknight.com (дата звернення 30.04.2026)

5. Ori and the Blind Forest Official Website. URL: https://www.orithegame.com (дата звернення 30.04.2026)

6. Blasphemous Official Website. URL: https://thegamekitchen.com/blasphemous (дата звернення 30.04.2026)

7. Закон України «Про охорону праці». URL: https://zakon.rada.gov.ua (дата звернення 12.05.2026)

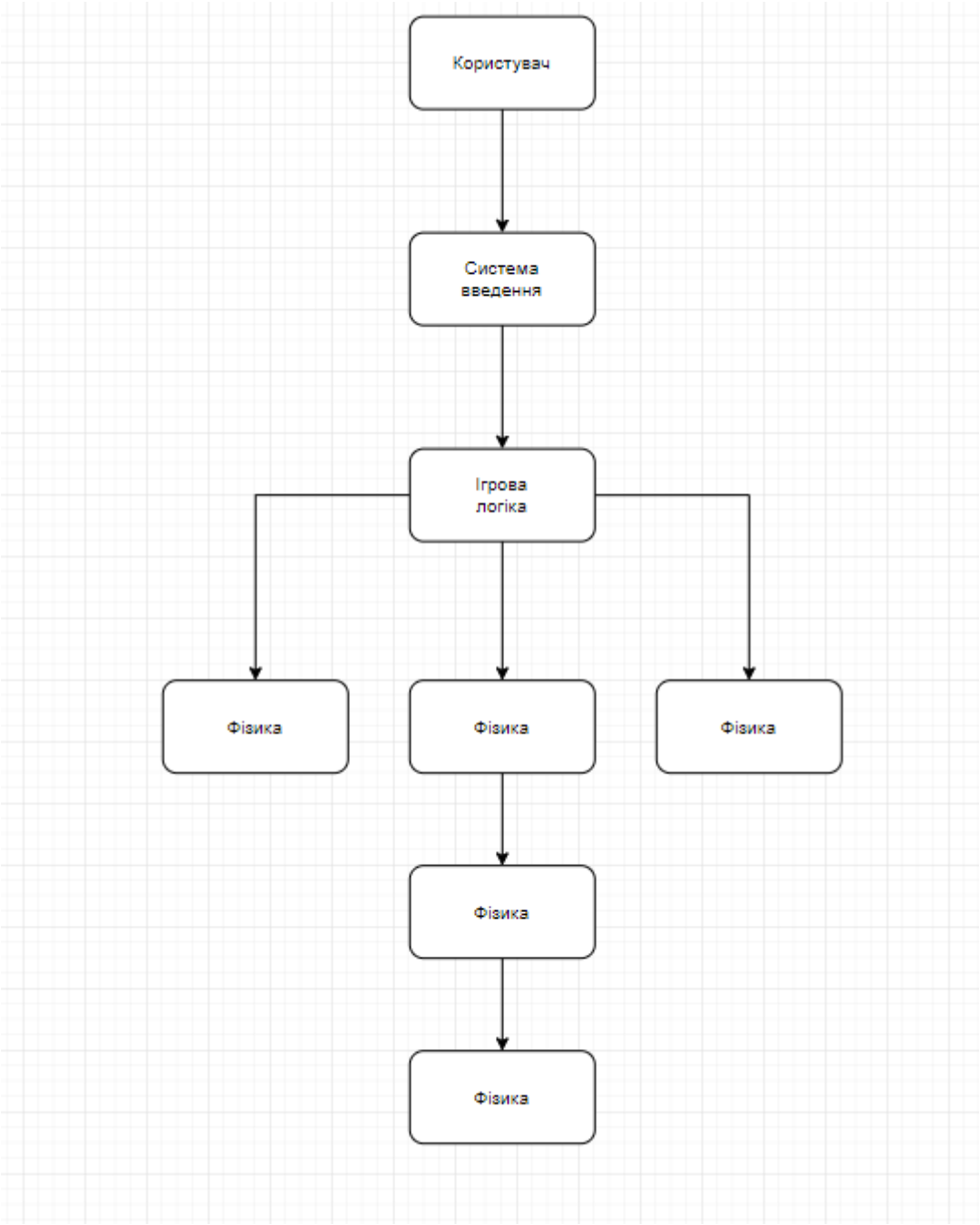
8. Кодекс законів про працю України. URL: https://zakon.rada.gov.ua (дата звернення 15.05.2026)

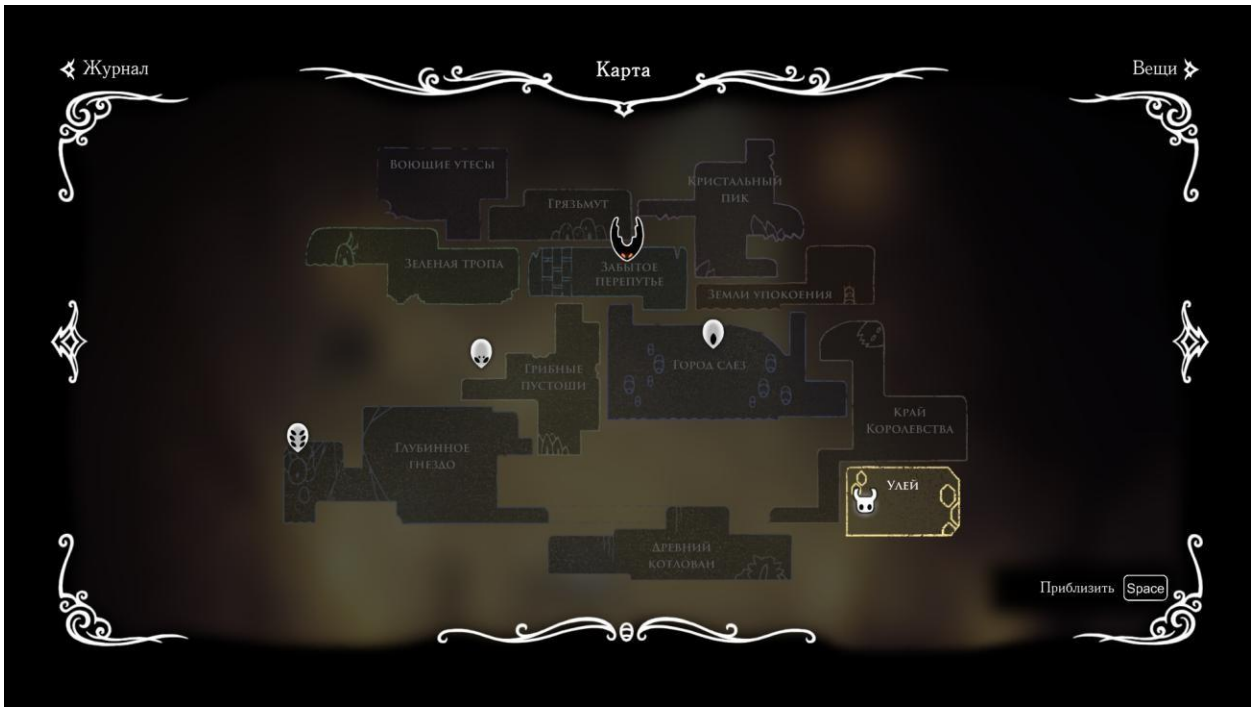
9. Правила пожежної безпеки в Україні. URL: https://zakon.rada.gov.ua (дата звернення 17.05.2026)

10. Microsoft. Visual Studio Documentation. URL: https://learn.microsoft.com/visualstudio (дата звернення 15.04.2026)

11. Unity Technologies. 2D Game Development Workflow. URL: https://docs.unity3d.com (дата звернення 21.05.2026)

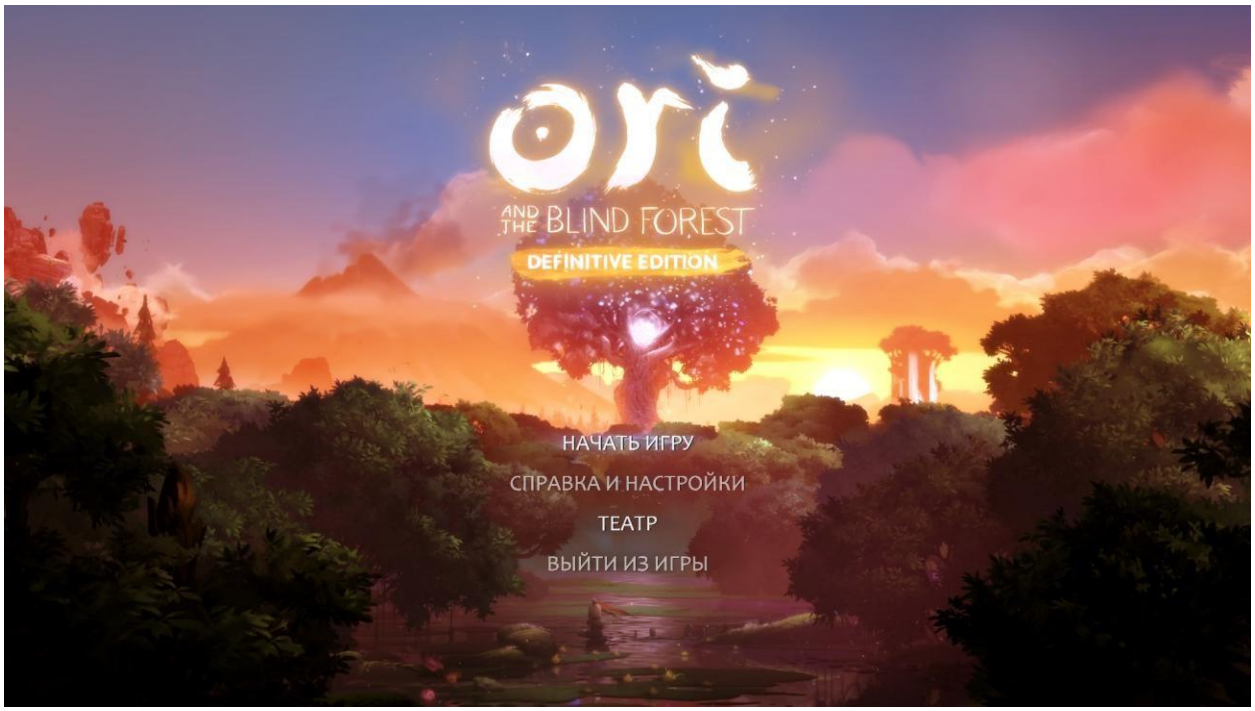
ДОДАТОК А

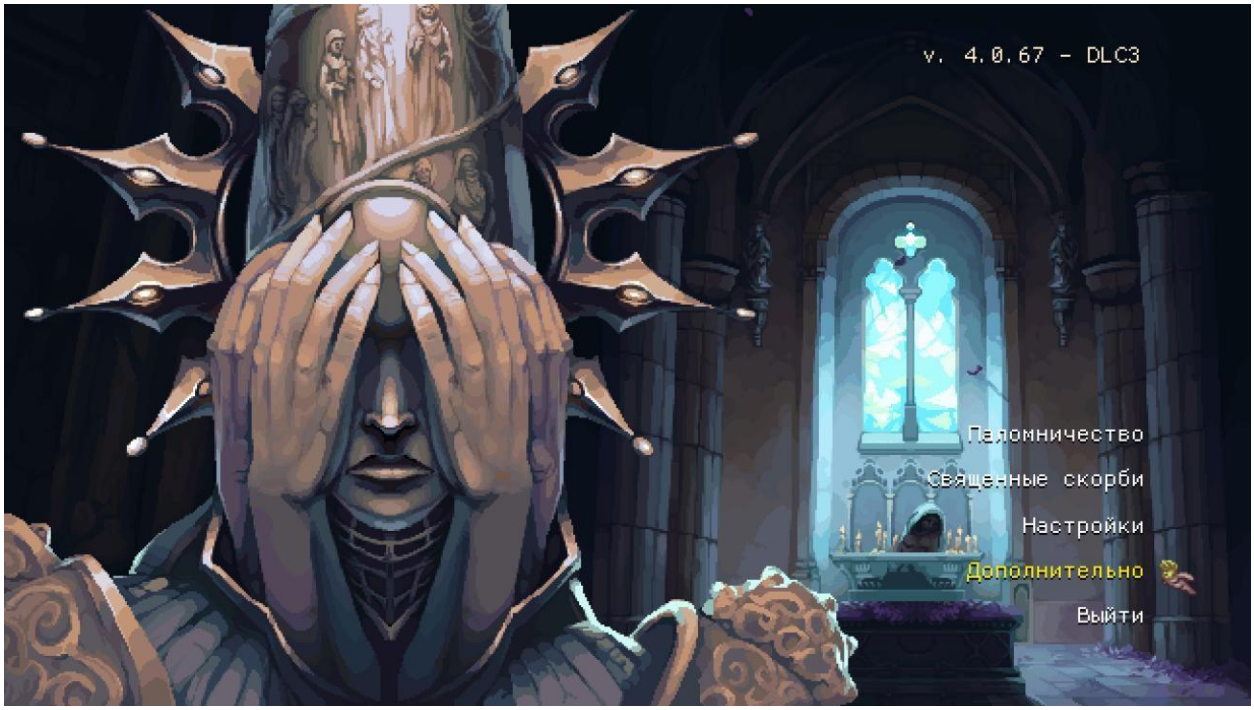

Користувач
Система введення
Ігрова логіка
Фізика
Фізика
Фізика
Фізика
Фізика



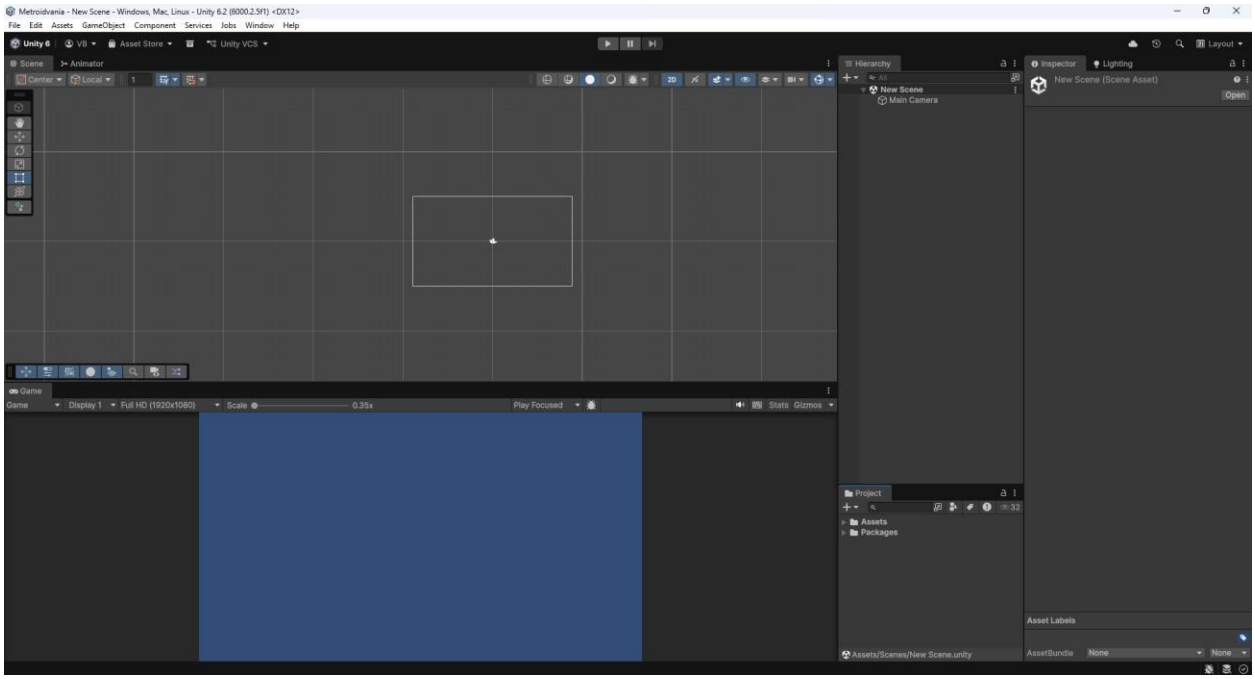

```
Нова здатність
      ↓
Нові області
      ↓
Нові противники
      ↓
Нові здібності
```

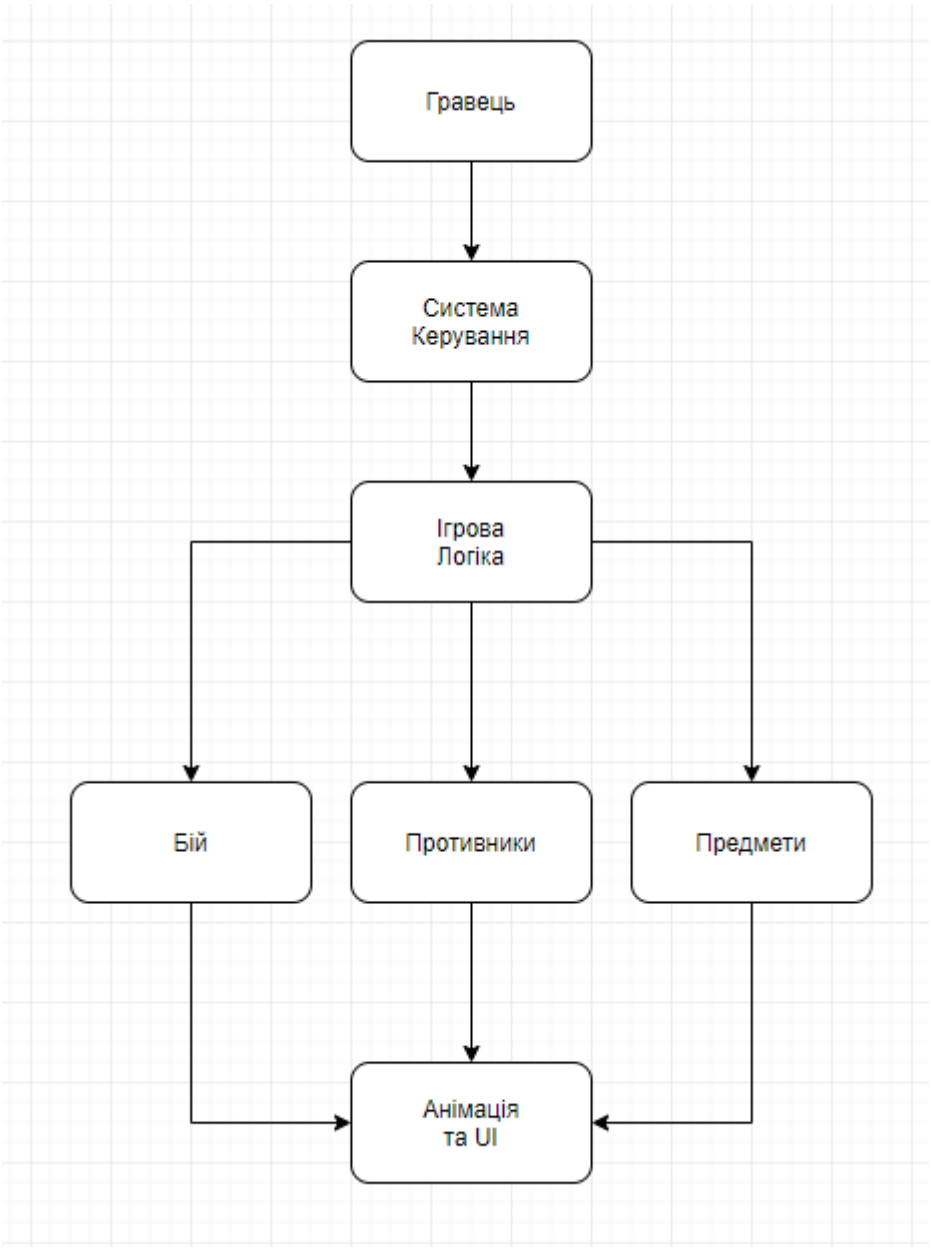

HOLLOW KNIGHT
Начать игру
Настройки
Достижения
Дополнительно
Выход
1.5.12620
team cherry

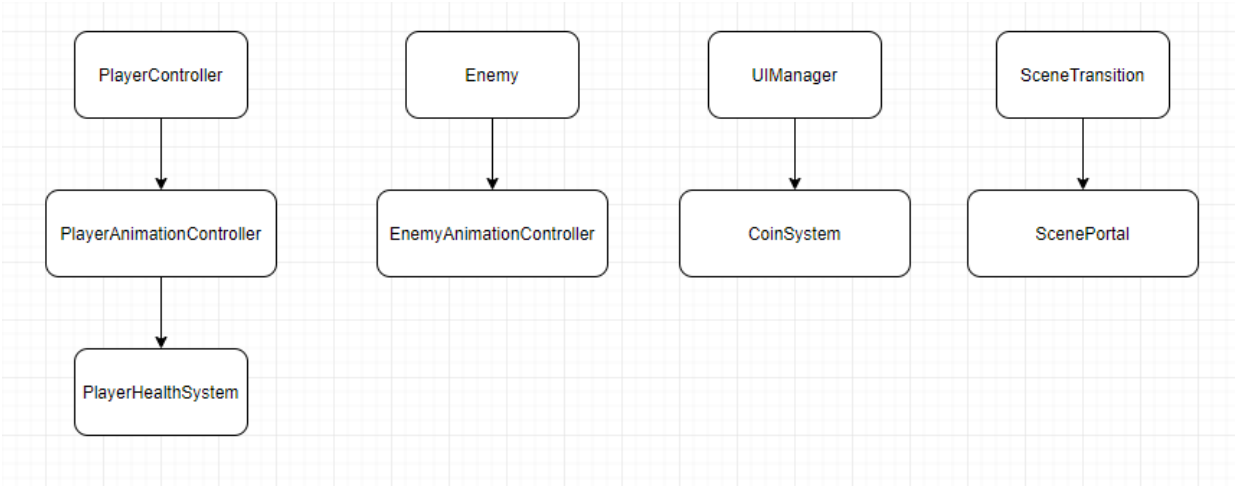

ori
AND THE BLIND FOREST
DEFINITIVE EDITION
НАЧАТЬ ИГРУ
СПРАВКА И НАСТРОЙКИ
ТЕАТР
ВЫЙТИ ИЗ ИГРЫ

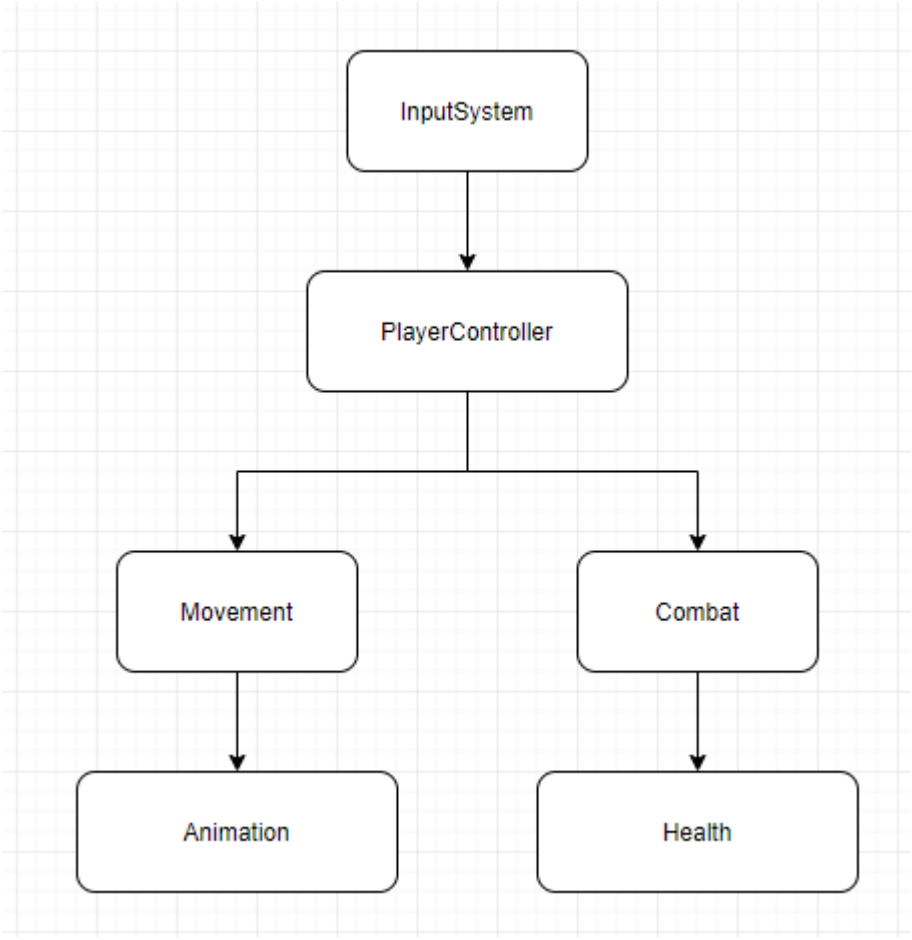

v. 4.0.67 - DLC3
Паломничество
Священные скорби
Настройки
Дополнительно
Выйти

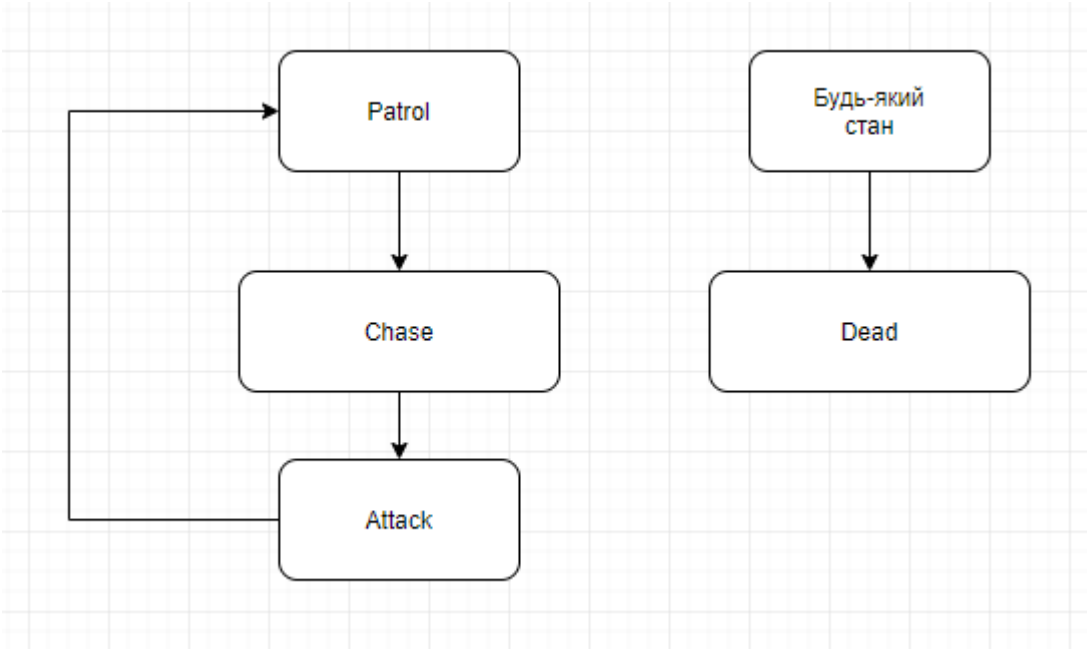

UNREAL
ENGINE
Unity
GODOT

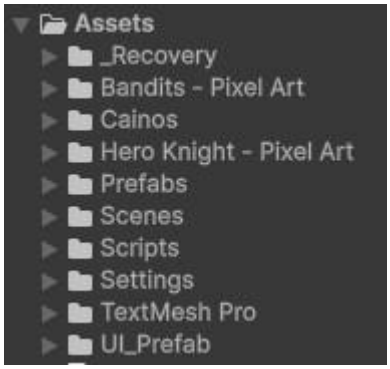

Metroidvania - New Scene - Windows, Mac, Linux - Unity 6.2 (6000.2.5f1) <DX12>
File Edit Assets GameObject Component Services Jobs Window Help
Unity 6
VB
Asset Store
Unity VCS
Layout
Scene
Animator
Center
Local
2D
Hierarchy
New Scene
Main Camera
Inspector
Lighting
New Scene (Scene Asset)
Open
Game
Display 1
Full HD (1920x1080)
Scale
0.35x
Play Focused
Stats
Gizmos
Project
Assets
Packages
Asset Labels
Assets/Scenes/New Scene.unity
AssetBundle
None
None

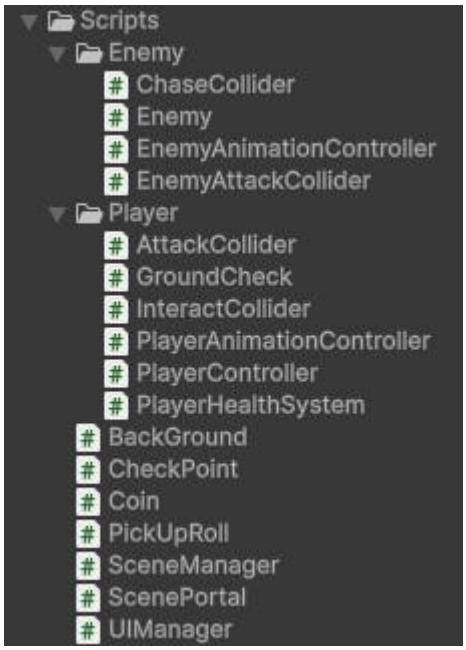

Гравець
Система Керування
Ігрова Логіка
Бій
Противники
Предмети
Анімація та UI

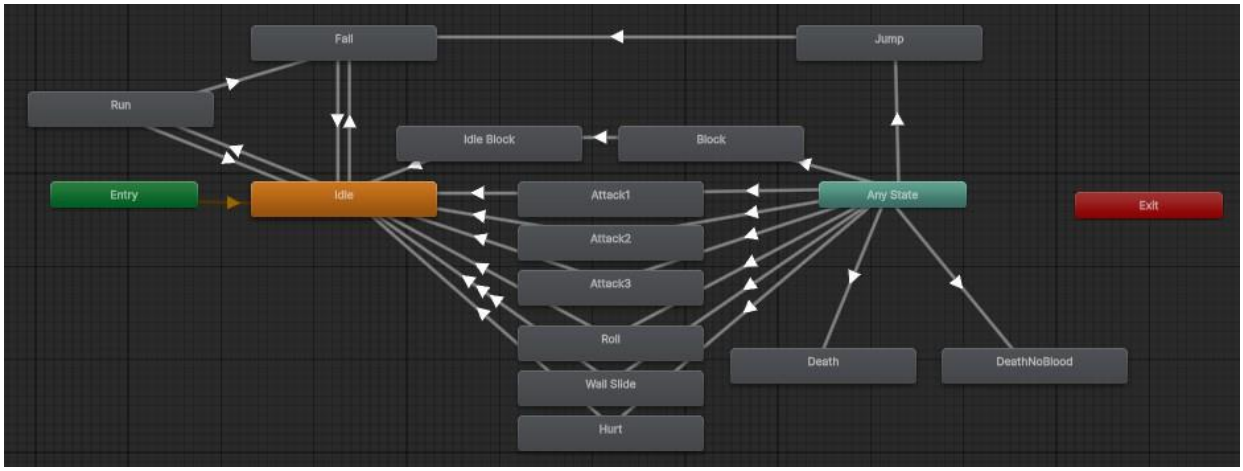

PlayerController
PlayerAnimationController
PlayerHealthSystem
Enemy
EnemyAnimationController
UIManager
CoinSystem
SceneTransition
ScenePortal

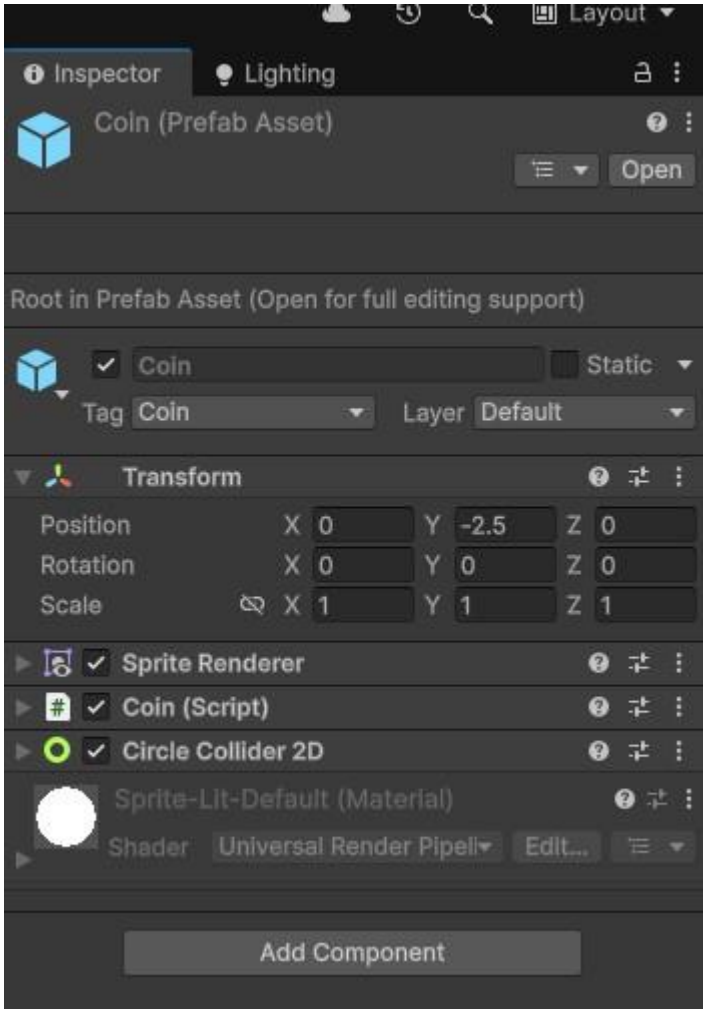

InputSystem
PlayerController
Movement
Combat
Animation
Health

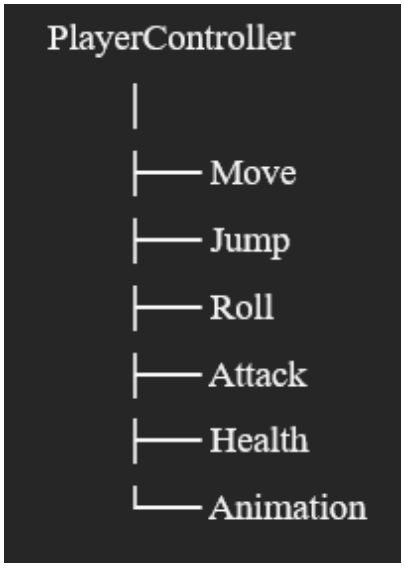

Patrol
Chase
Attack
Будь-який стан
Dead

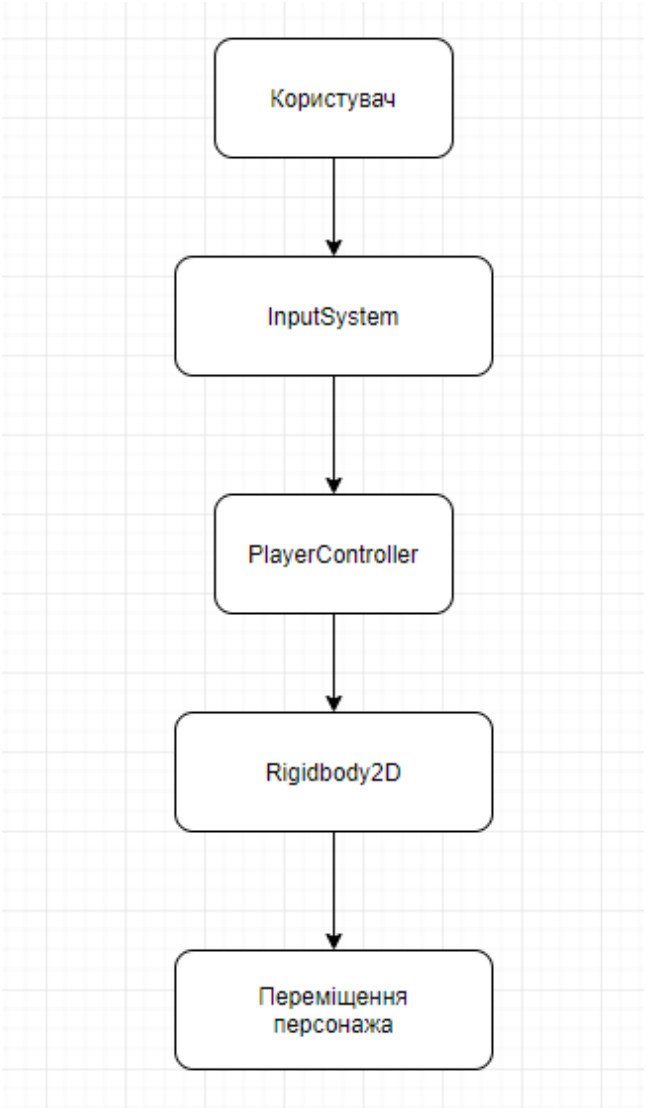

Assets
_Recovery
Bandits - Pixel Art
Cainos
Hero Knight - Pixel Art
Prefabs
Scenes
Scripts
Settings
TextMesh Pro
UI_Prefab

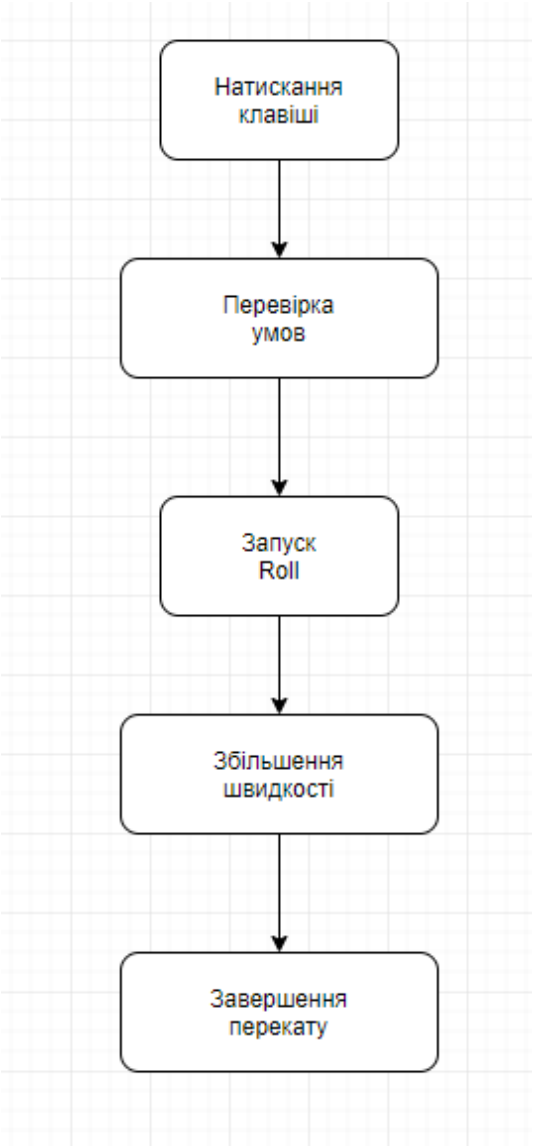

Scripts
Enemy
ChaseCollider
Enemy
EnemyAnimationController
EnemyAttackCollider
Player
AttackCollider
GroundCheck
InteractCollider
PlayerAnimationController
PlayerController
PlayerHealthSystem
BackGround
CheckPoint
Coin
PickUpRoll
SceneManager
ScenePortal
UIManager

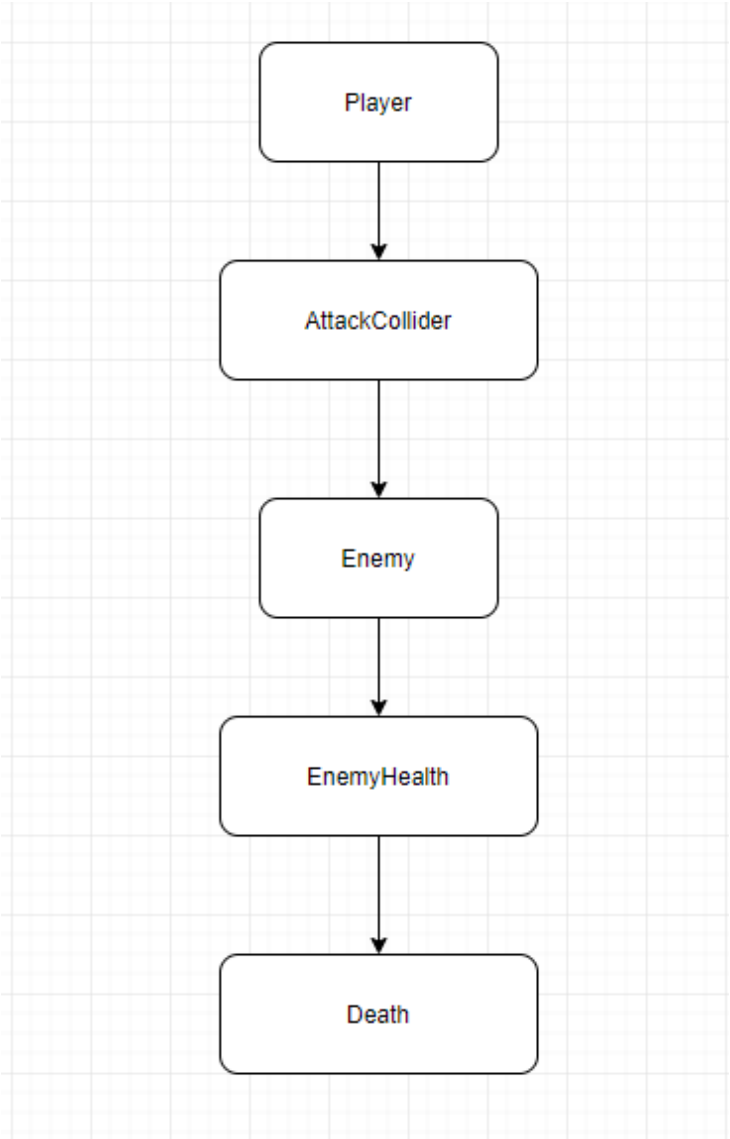

Fall
Jump
Run
Idle Block
Block
Entry
Idle
Attack1
Any State
Exit
Attack2
Attack3
Roll
Death
DeathNoBlood
Wall Slide
Hurt

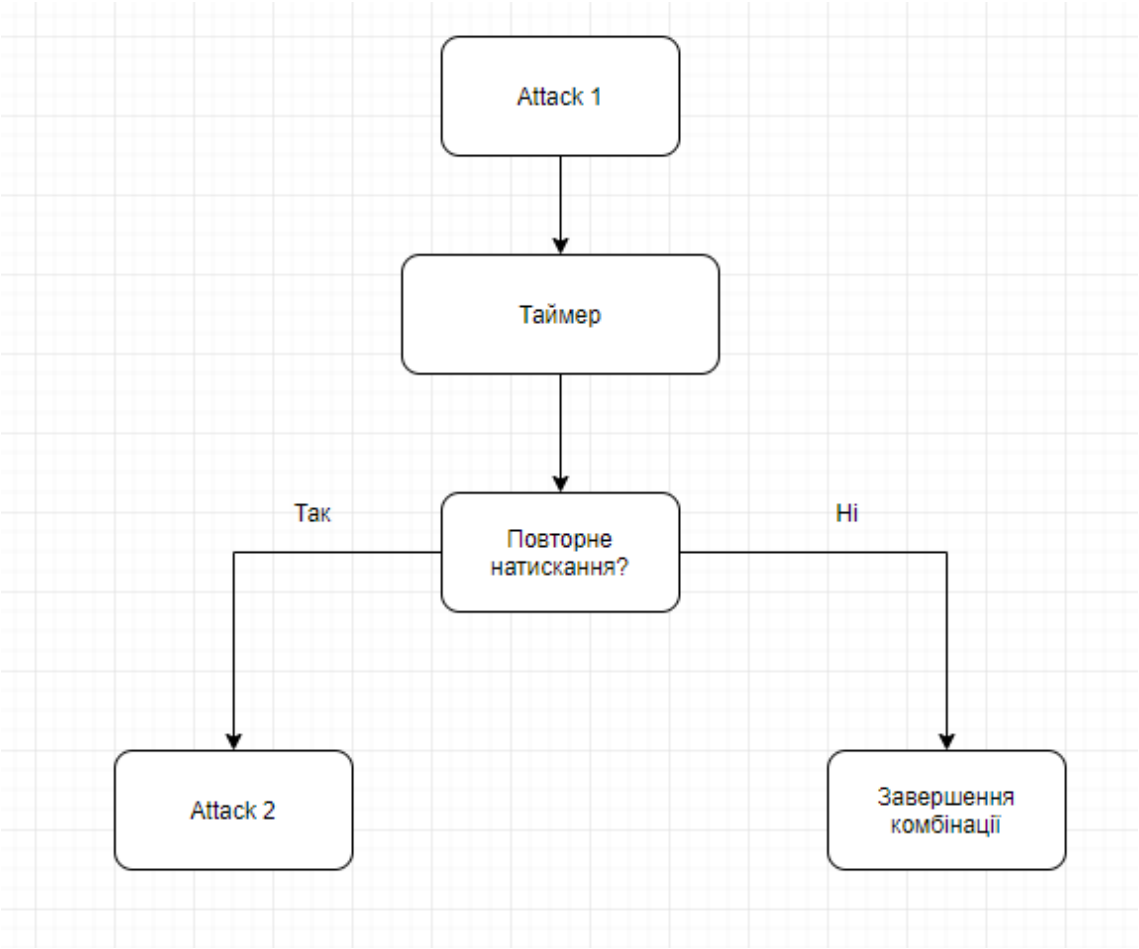

Layout
Inspector
Lighting
Coin (Prefab Asset)
Open
Root in Prefab Asset (Open for full editing support)
Coin
Static
Tag Coin
Layer Default
Transform
Position X 0 Y -2.5 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Sprite Renderer
Coin (Script)
Circle Collider 2D
Sprite-Lit-Default (Material)
Shader Universal Render Pipeli
Edit...
Add Component

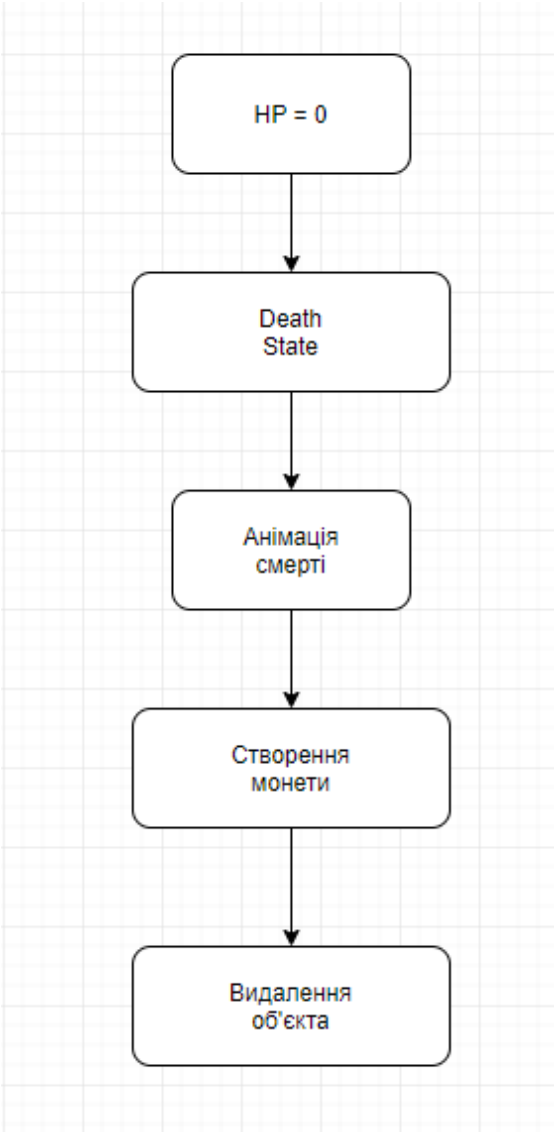

PlayerController
Move
Jump
Roll
Attack
Health
Animation

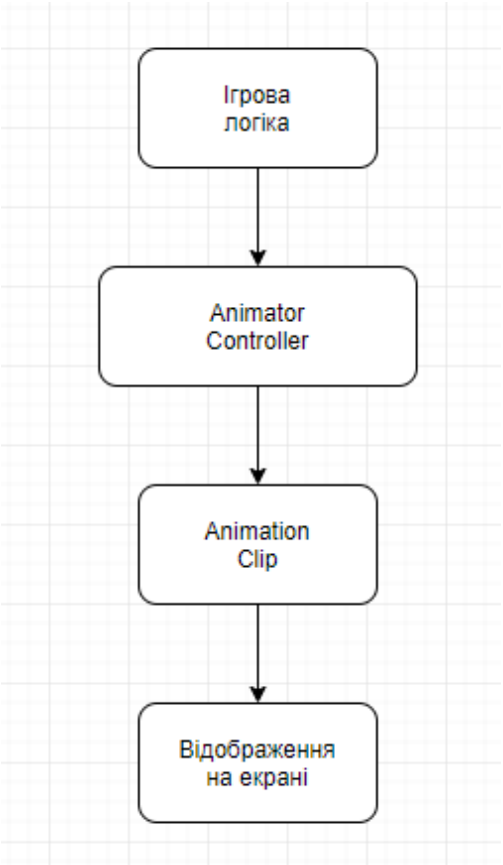

Користувач
InputSystem
PlayerController
Rigidbody2D
Переміщення
персонажа

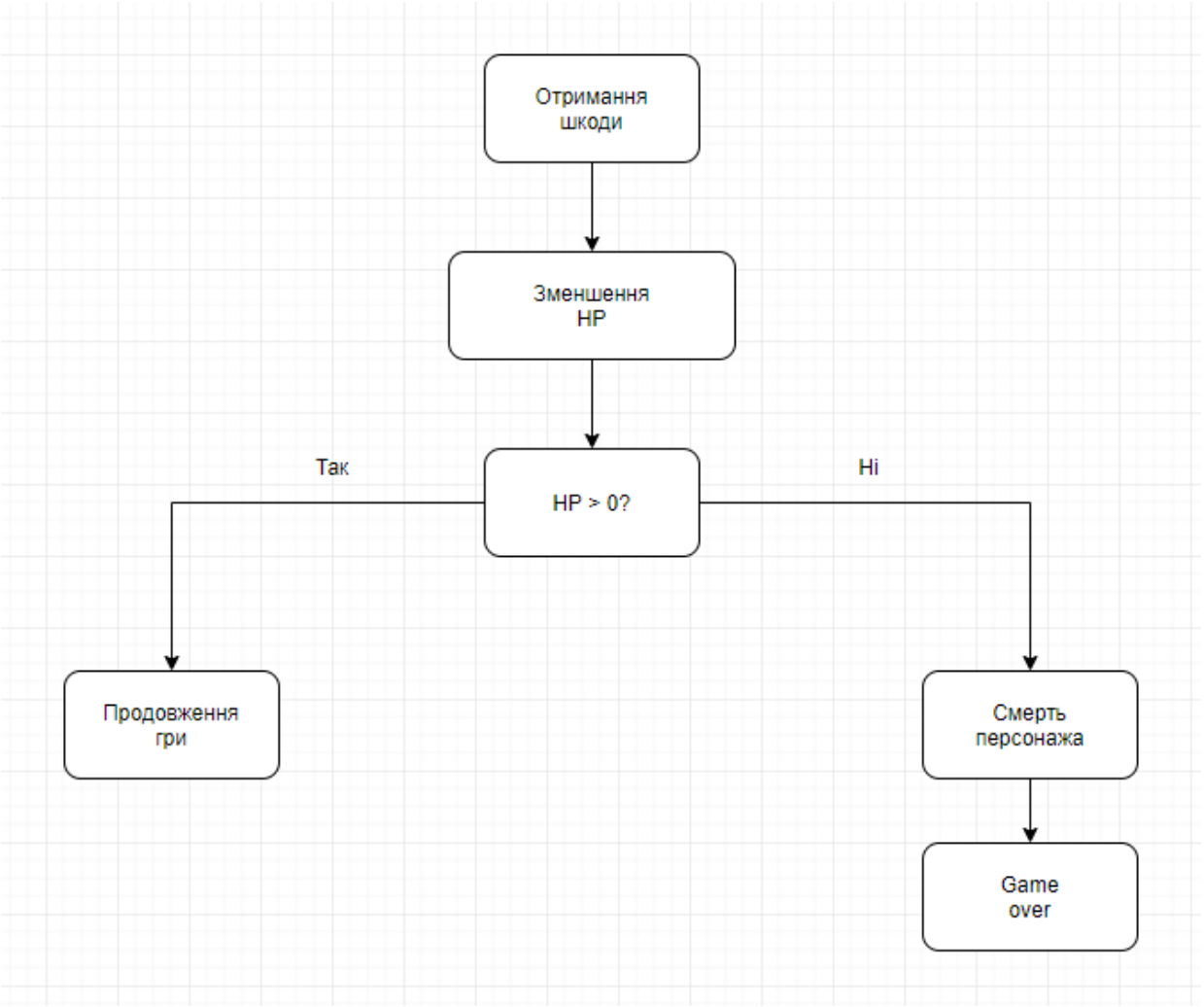

Натискання клавіші
Перевірка умов
Запуск Roll
Збільшення швидкості
Завершення перекату

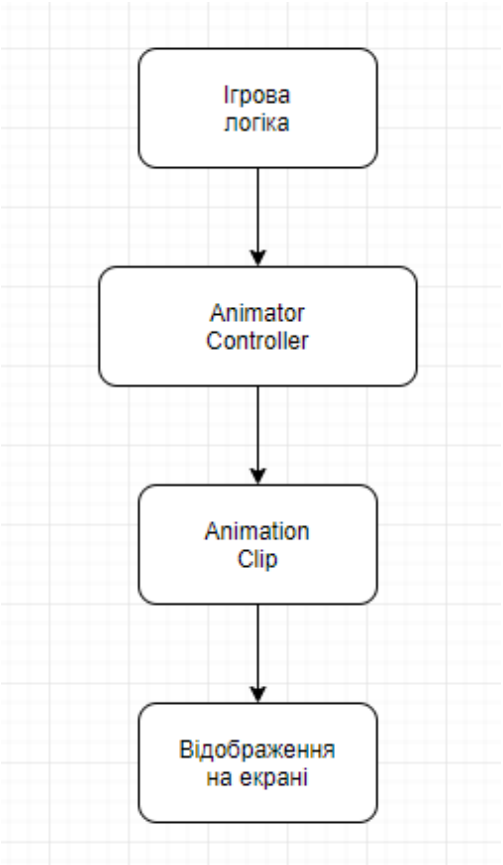

Player
AttackCollider
Enemy
EnemyHealth
Death

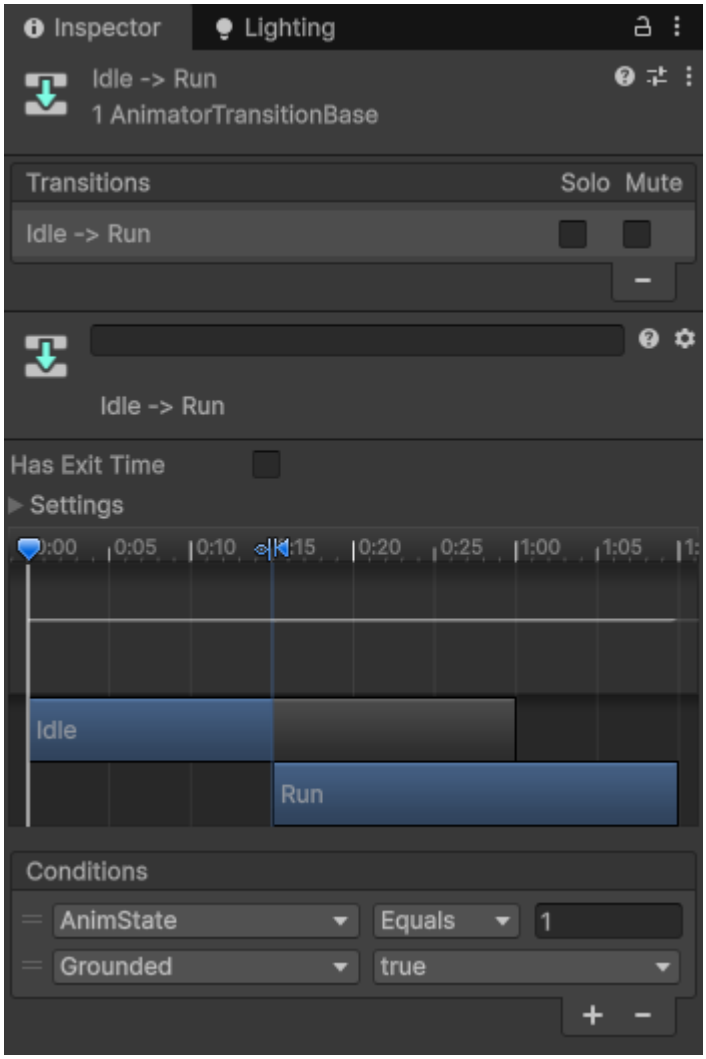

Attack 1
Таймер
Так
Повторне натискання?
Ні
Attack 2
Завершення комбінації

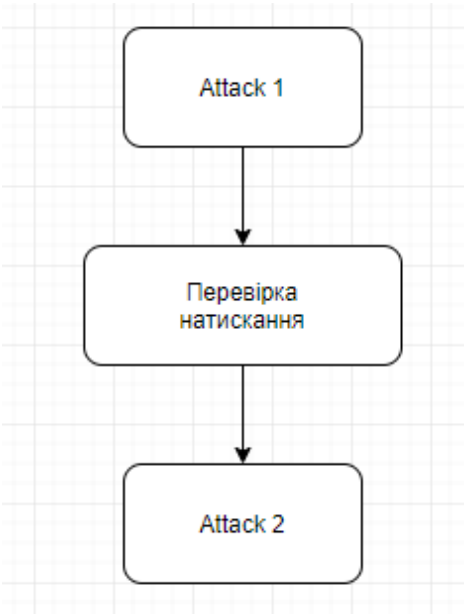

HP = 0
Death
State
Анімація
смерті
Створення
монети
Видалення
об'єкта

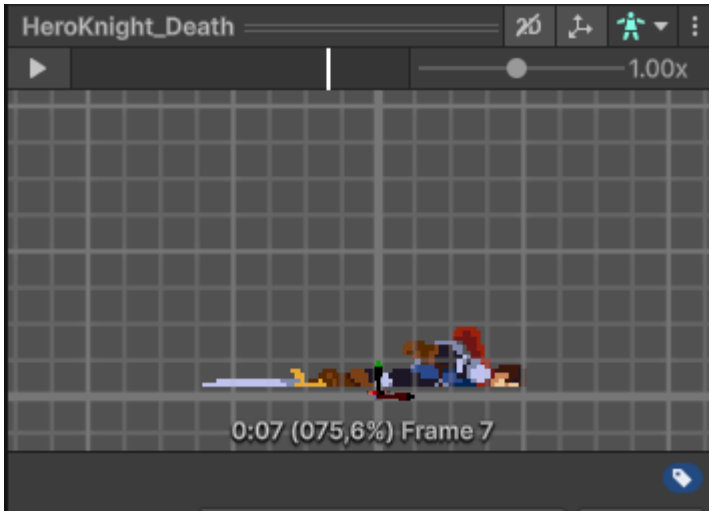

Ігрова
логіка
Animator
Controller
Animation
Clip
Відображення
на екрані

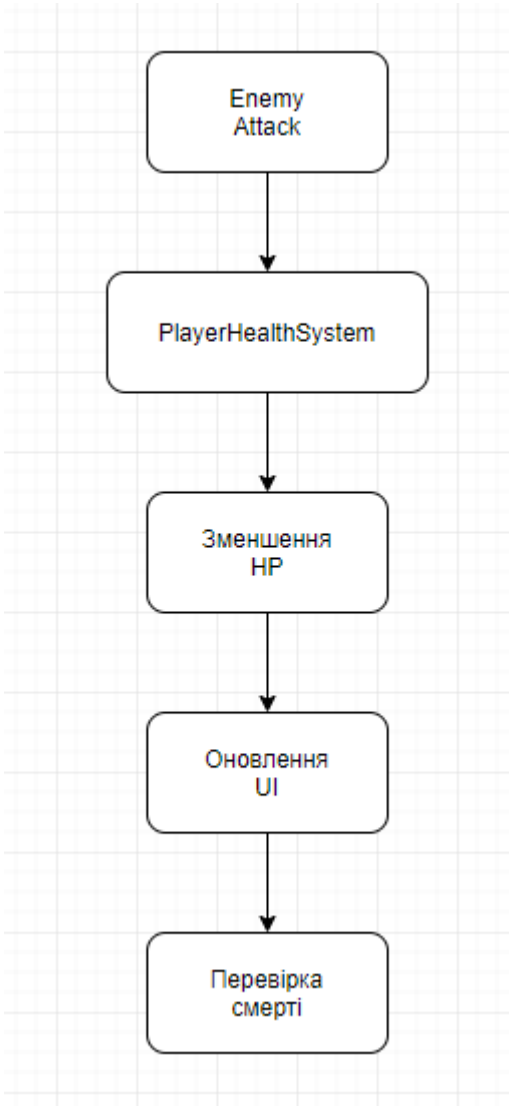

Отримання шкоди
Зменшення HP
HP > 0?
Так
Ні
Продовження гри
Смерть персонажа
Game over


Ігрова
логіка
Animator
Controller
Animation
Clip
Відображення
на екрані

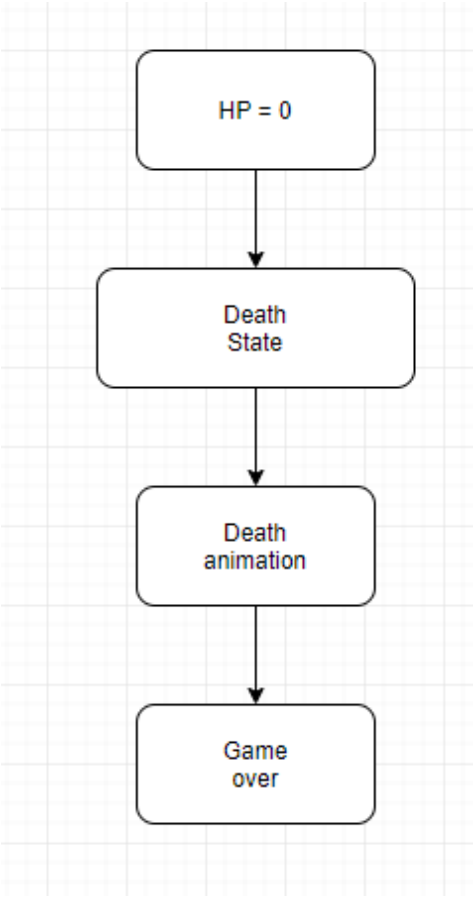

Inspector
Lighting
Idle -> Run
1 AnimatorTransitionBase
Transitions
Solo
Mute
Idle -> Run
Idle -> Run
Has Exit Time
Settings
:00
0:05
0:10
15
0:20
0:25
1:00
1:05
Idle
Run
Conditions
AnimState
Equals
1
Grounded
true

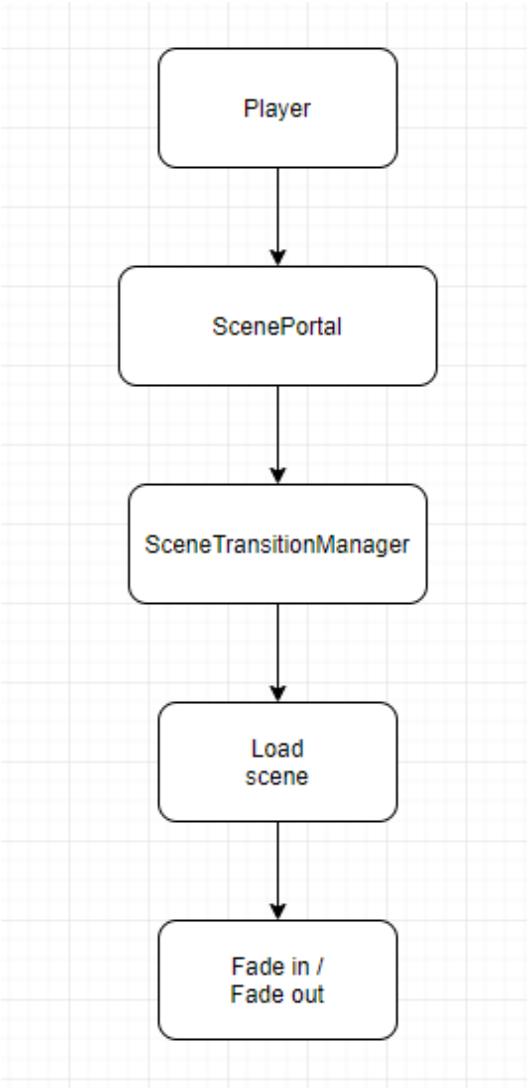

Attack 1
Перевірка
натискання
Attack 2

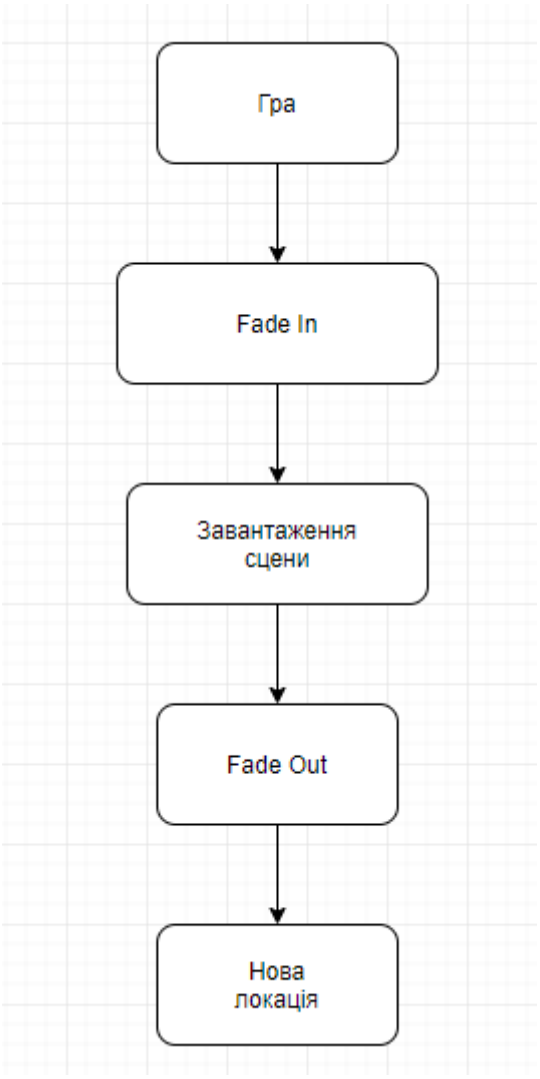

HeroKnight_Death
1.00x
0:07 (075,6%) Frame 7

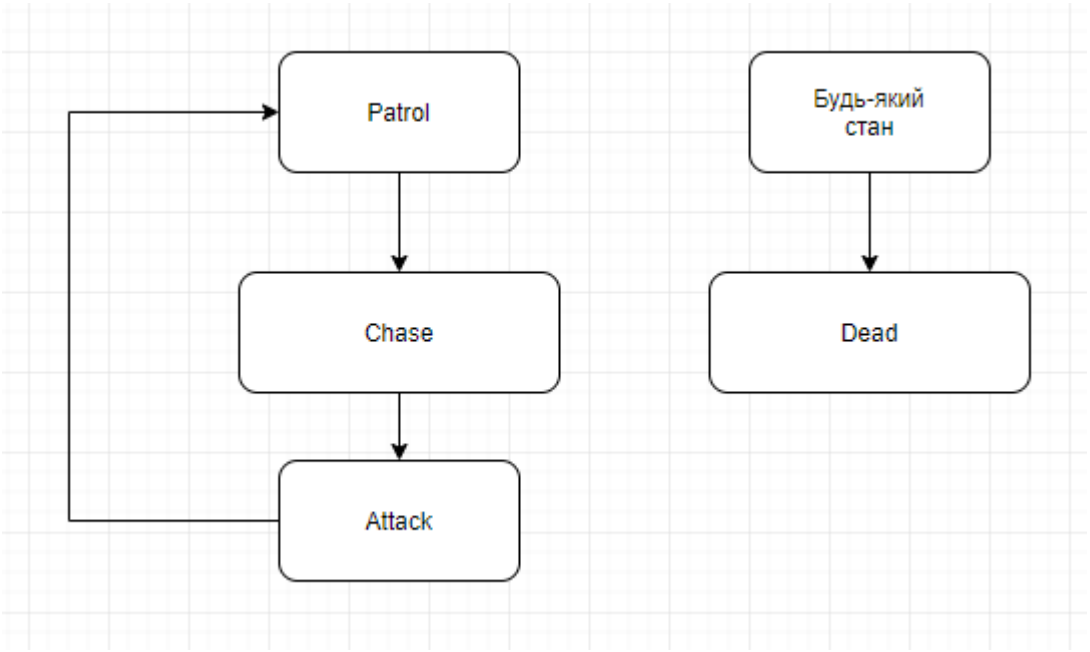

Enemy Attack
PlayerHealthSystem
Зменшення HP
Оновлення UI
Перевірка смерті

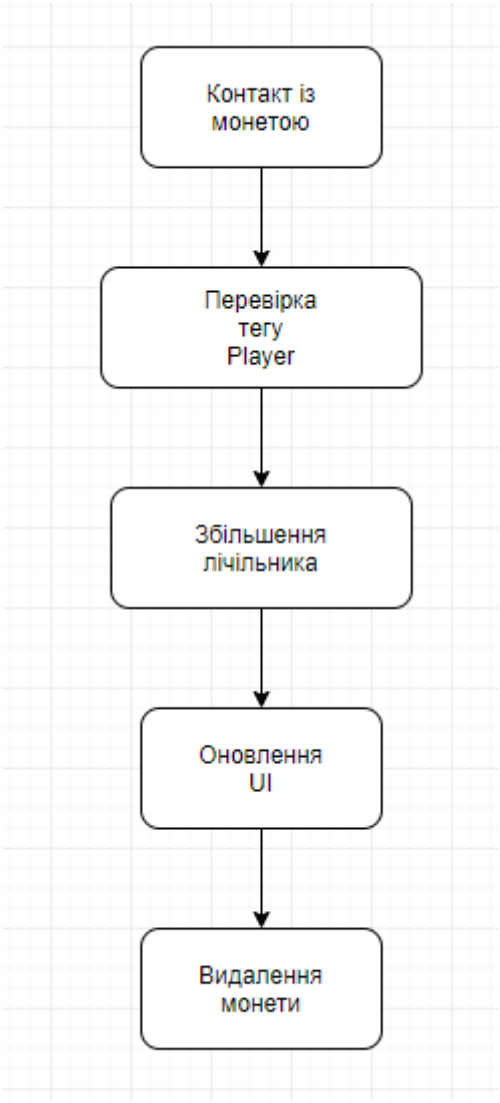

Отримання шкоди
Зменшення HP
Вимкнення індикатору
Оновлення інтерфейсу

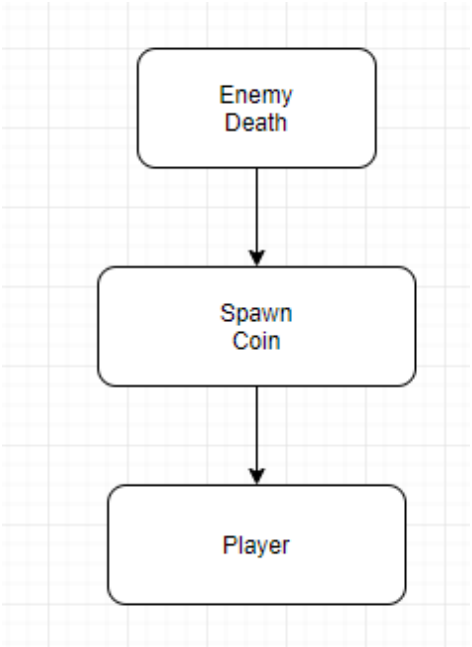

HP = 0
Death
State
Death
animation
Game
over


Player
ScenePortal
SceneTransitionManager
Load
scene
Fade in /
Fade out


Гра
Fade In
Завантаження сцени
Fade Out
Нова локація


Patrol
Chase
Attack
Будь-який стан
Dead

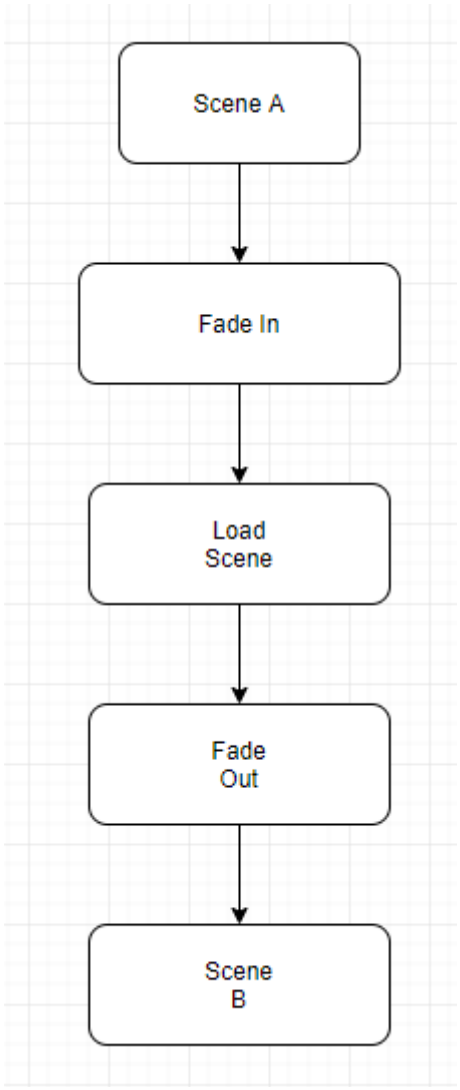

Контакт із монетою
Перевірка тегу Player
Збільшення лічільника
Оновлення UI
Видалення монети

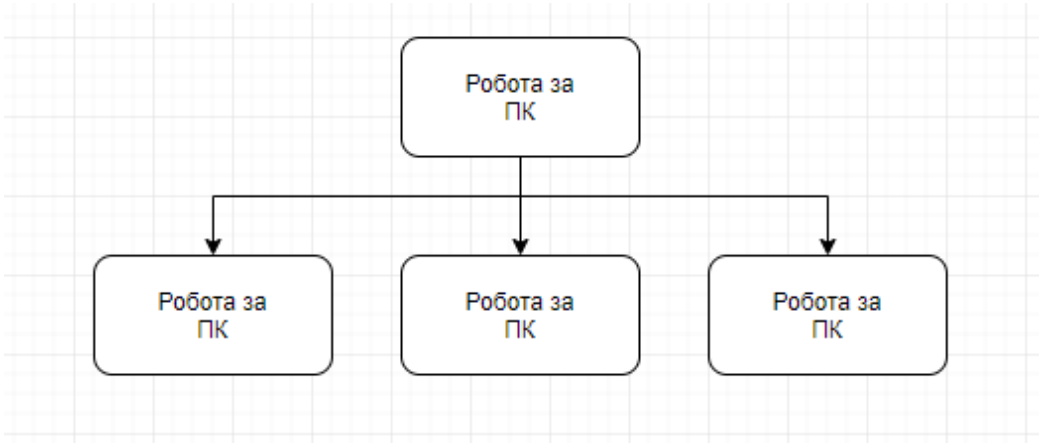

Enemy Death
Spawn Coin
Player

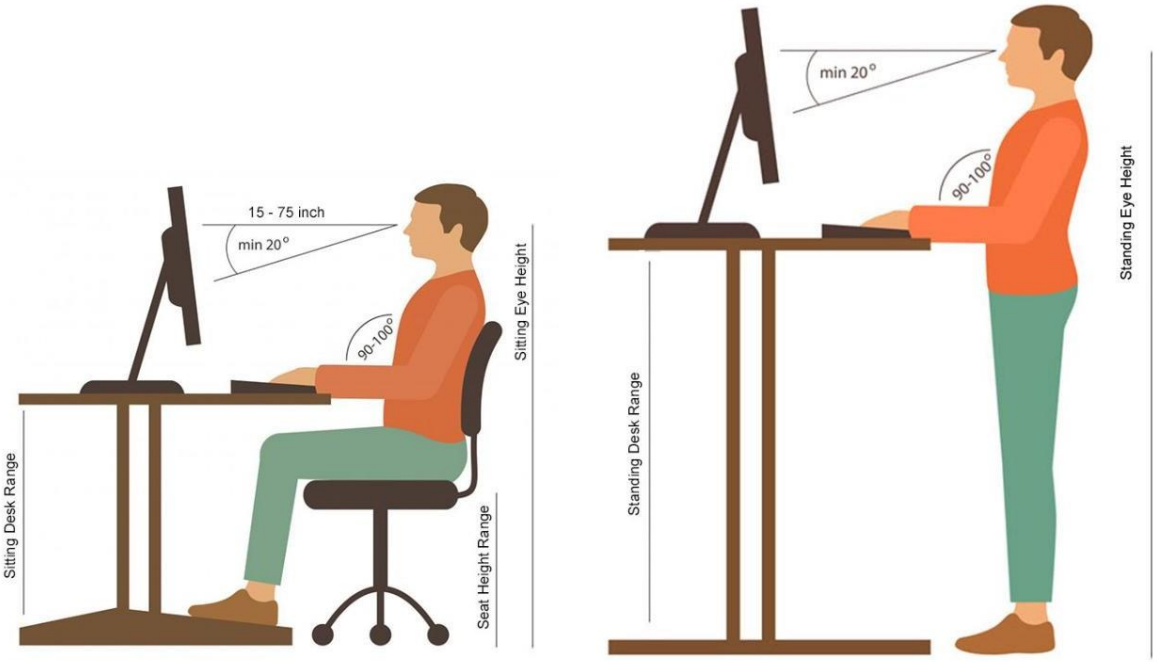

Canvas
Health
Health (1)
Health (2)
LoadingPanel
GameOverPanel
Coins


0 X 5


You Are Dead
Try Again


Scene A
Fade In
Load
Scene
Fade
Out
Scene
B


Робота за
ПК
Робота за
ПК
Робота за
ПК
Робота за
ПК


15 - 75 inch
min 20°
90-100°
Sitting Eye Height
Sitting Desk Range
Seat Height Range
min 20°
90-100°
Standing Eye Height
Standing Desk Range

ДОДАТОК Б

```
using UnityEngine;
using TMPro;

public class UIManager : MonoBehaviour
{
    public TextMeshProUGUI coinsCounterText;

    private Enemy enemy;
    private PlayerHealthSystem healthSystem;
    private PlayerController player;

    private int coinsCount;

    public CanvasGroup canvasGroup;

    public bool startAnimation;

    private void Awake()
    {
        DontDestroyOnLoad(gameObject);
    }

    private void Start()
    {
        enemy =
GameObject.Find("Enemy").GetComponent<Enemy>();
```

```
        healthSystem =
GameObject.Find("PlayerHealthSystem").GetComponent<PlayerHealthS
ystem>();
        player =
GameObject.Find("Player").GetComponent<PlayerController>();

        coinsCount = 0;

        startAnimation = false;

        HideGameOver();
    }

    private void Update()
    {
        if (startAnimation)
        {
            ShowGameOver();
        }

        coinsCounterText.text = coinsCount + " X ";
    }

    public void PlusMoney()
    {
        coinsCount++;
        Debug.Log("Plus money");
    }
```

```
public void ShowGameOver()
{
    canvasGroup.alpha = 1;
    canvasGroup.interactable = true;
    canvasGroup.blocksRaycasts = true;
}

public void HideGameOver()
{
    canvasGroup.alpha = 0;
    canvasGroup.interactable = false;
    canvasGroup.blocksRaycasts = false;
}

public void RestartButton()
{
    Debug.Log("BUTTON WORK");

    player.transform.position = player.checkpointPos;

    player.isDead = false;

    startAnimation = false;

    healthSystem.AddHP();
    enemy.currenttHealth = enemy.maxHealth;

    HideGameOver();
```

```
    }
}

using UnityEngine;

public class InteractCollider : MonoBehaviour
{
    private PlayerController player;

    private void Start()
    {
        player =
GameObject.Find("Player").GetComponent<PlayerController>();
    }

    private void Update()
    {
        if (player.inputActions.Player.Interact.triggered && player.canSave)
        {
            player.Checkpoint();
        }
    }
}
```

```
using UnityEngine;

public class CheckPoint : MonoBehaviour
{
    private PlayerController player;

    private void Start()
    {
        player =
GameObject.Find("Player").GetComponent<PlayerController>();
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            player.canSave = true;
        }
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            player.canSave = false;
        }
```

```
    }
}

using System.Collections;
using UnityEngine;

public enum EnemyState {Patrol, Chase, Attack, Dead}

public class Enemy : MonoBehaviour
{
    public PlayerController playerController;

    public GameObject coin;
    public GameObject enemyVisual;

    public bool isAttacking;
    public bool isDead;
    public bool timerStarted;

    public EnemyState currentState;

    private EnemyAnimationController animationController;
    private BoxCollider2D hitBox;
    private PlayerHealthSystem healthSystem;

    [SerializeField] private float speed;

    public int maxHealth;
```

```
        public int currenttHealth;

        [SerializeField] private int restTime;
        [SerializeField] private int patrollingX;

        [SerializeField] private bool isCoinExist;
        [SerializeField] private bool onFirstPoint;
        [SerializeField] private bool onSecondPoint;
        [SerializeField] private bool isMovingLeft;

        private Vector2 targetPoint1;
        private Vector2 targetPoint2;

        private int attackTimerCounter;

        private void Start()
        {
            hitBox = GetComponent<BoxCollider2D>();
            animationController =
GameObject.Find("EnemyVisual").GetComponent<EnemyAnimationCon
troller>();
            playerController =
GameObject.Find("Player").GetComponent<PlayerController>();
            healthSystem =
GameObject.Find("PlayerHealthSystem").GetComponent<PlayerHealthS
ystem>();

            currentState = EnemyState.Patrol;
```

```
        targetPoint1 = new Vector2(transform.position.x - patrollingX,
transform.position.y);
        targetPoint2 = new Vector2(transform.position.x + patrollingX,
transform.position.y);

        isCoinExist = false;
        isAttacking = false;
        isDead = false;
        onFirstPoint = false;
        onSecondPoint = false;
        timerStarted = false;
        isMovingLeft = false;

        currenttHealth = maxHealth;
        attackTimerCounter = 0;
    }

    private void Update()
    {
        switch (currentState)
        {
            case EnemyState.Patrol:
                EnemyVisualRotate();
                Patrolling();
                break;

            case EnemyState.Chase:
                EnemyVisualRotate();
                Chasing();
```

```
                break;

            case EnemyState.Attack:
                Attack();
                break;

            case EnemyState.Dead:
                Death();
                break;
        }

        if (playerController.isDead)
        {
            currentState = EnemyState.Patrol;
        }

        if (currenttHealth <= 0)
        {
            if (!isCoinExist)
            {
                Instantiate(coin, transform.position, Quaternion.Euler(0,
0, 0));
                isCoinExist = true;
            }
            currentState = EnemyState.Dead;
        }
    }

    public void DamagePlayer()
```

```
{
    if (!playerController.isInvincible)
    {
        StartCoroutine(invicibleTimer());
        playerController.Hurt();
        healthSystem.ChangeHP();
    }
}

private void Death()
{
    isDead = true;
    hitBox.enabled = false;
}

private void Attack()
{
    isAttacking = true;
    if(attackTimerCounter == 0)
        StartCoroutine(attackTimer());
}

private void Chasing()
{
    if (playerController.transform.position.x < transform.position.x)
    {
        isMovingLeft = true;
    }
```

```
            else
            {
                isMovingLeft = false;
            }
            transform.position = Vector2.MoveTowards(transform.position,
new Vector2(playerController.transform.position.x, transform.position.y),
speed * Time.deltaTime);


        }


        private void Patrolling()
        {
            if (onFirstPoint || !onFirstPoint && !onSecondPoint)
            {
                transform.position =
Vector2.MoveTowards(transform.position, targetPoint2, speed *
Time.deltaTime);
                isMovingLeft = false;


                if (Vector2.Distance(transform.position, targetPoint2) < 0.05f
&& !timerStarted)
                    StartCoroutine(waitToWalk());
            }
            else if (onSecondPoint)
            {
                transform.position =
Vector2.MoveTowards(transform.position, targetPoint1, speed *
Time.deltaTime);
                isMovingLeft = true;
```

```
            if (Vector2.Distance(transform.position, targetPoint1) < 0.05f &&
!timerStarted)
                StartCoroutine(waitToWalk());
            }
        }

        private void EnemyVisualRotate()
        {
            if (isMovingLeft && !isDead)
                enemyVisual.transform.rotation = Quaternion.Euler(new
Vector3(0, 0, 0));
            else if (!isDead)
                enemyVisual.transform.rotation = Quaternion.Euler(new
Vector3(0, -180, 0));
        }

        public void TakeDamage()
        {
            currenttHealth--;
        }

        private void OnTriggerEnter2D(Collider2D collision)
        {
            if (collision.CompareTag("Player") && hitBox.IsTouching(collision)
&& !playerController.isRolling)
            {
                DamagePlayer();
            }
        }
```

```
IEnumerator invicibleTimer()
{
    playerController.isInvincible = true;
    yield return new WaitForSeconds(1);
    playerController.isInvincible = false;
}

IEnumerator attackTimer()
{
    attackTimerCounter = 1;
    yield return new WaitForSeconds(restTime);
    currentState = EnemyState.Chase;
    animationController.isAttackTriggered = false;
    attackTimerCounter = 0;
}

IEnumerator waitToWalk()
{
    timerStarted = true;
    yield return new WaitForSeconds(2);

    if (Vector2.Distance(transform.position, targetPoint2) < 0.05f)
    {
        onFirstPoint = false;
        onSecondPoint = true;
    }
    else if (Vector2.Distance(transform.position, targetPoint1) <
0.05f)
    {
```

```
            onFirstPoint = true;
            onSecondPoint = false;
        }

        timerStarted = false;
    }
}
```

```
using UnityEngine;

public class EnemyAnimationController : MonoBehaviour
{
    private Enemy enemy;
    private Animator animator;

    private string ATTACK = "Attack";
    private string DEATH = "Death";
    private string ANIMSTATE = "AnimState";

    public bool isAttackTriggered;

    private void Start()
    {
        enemy = GameObject.Find("Enemy").GetComponent<Enemy>();
```

```
            animator = GetComponent<Animator>();

            isAttackTriggered = false;
        }

        private void Update()
        {
            if (enemy.currentState == EnemyState.Attack &&
!isAttackTriggered)
            {
                isAttackTriggered = true;
                animator.SetTrigger(ATTACK);
            }

            if (!enemy.timerStarted)
            {
                animator.SetInteger(ANIMSTATE, 2);
            }
            else
            {
                animator.SetInteger(ANIMSTATE, 0);
            }

            if (enemy.isDead)
            {
                animator.SetTrigger(DEATH);
            }
        }
    }
```

```
using System.Collections;
using UnityEngine;

public class EnemyAttackCollider : MonoBehaviour
{
    private Enemy enemy;
    private void Start()
    {
        enemy = GameObject.Find("Enemy").GetComponent<Enemy>();
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player") && !enemy.isDead)
        {
            enemy.currentState = EnemyState.Attack;
            StartCoroutine(prepareAttack());
        }
    }

    IEnumerator prepareAttack()
    {
        yield return new WaitForSeconds(0.5f);
```

```
        if(!enemy.playerController.isRolling)
            enemy.DamagePlayer();
    }
}

using UnityEngine;

public class ChaseCollider : MonoBehaviour
{
    public GameObject enemyObj;

    private Enemy enemyScript;

    private void Start()
    {
        enemyScript = enemyObj.GetComponent<Enemy>();
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player") && !enemyScript.isDead)
        {
            enemyScript.currentState = EnemyState.Chase;
        }
    }
}
```

```
using System.Collections;
using UnityEngine;
using UnityEngine.InputSystem;

public class PlayerAnimationController : MonoBehaviour
{
    public GameObject groundChecker;

    private PlayerController playerController;
    private GroundCheck groundCheckScript;
    public Animator anim;

    public string HURT = "Hurt";

    private string DEATH = "Death";
    private string ATTACK1 = "Attack1";
    private string ATTACK2 = "Attack2";
    private string ROLL = "Roll";
    private string AIRSPEEDY = "AirSpeedY";
    private string JUMP = "Jump";
    private string ANIMSTATE = "AnimState";
    private string GROUNDED = "Grounded";
```

```
        private bool isDeathTriggered;
        private bool checkAttack;
        private bool nextAttack;

        private void Start()
        {
            playerController =
GameObject.Find("Player").GetComponent<PlayerController>();
            groundCheckScript =
groundChecker.GetComponent<GroundCheck>();
            anim = GetComponent<Animator>();
            anim.SetInteger(ANIMSTATE, 0);

            isDeathTriggered = false;
            checkAttack = false;
            nextAttack = false;
        }

        private void Update()
        {
            if (!playerController.isDead)
            {
                if (playerController.inputActions.Player.Roll.triggered &&
groundCheckScript.OnGround && !playerController.isRolling &&
playerController.hasDash)
                {
                    anim.SetTrigger(ROLL);
                }
```

```
            AttackChacker();

            if (playerController.inputActions.Player.Attack.triggered &&
nextAttack)
            {
               anim.SetTrigger(ATTACK2);
               nextAttack = false;
            }
            else if (playerController.inputActions.Player.Attack.triggered)
            {
               anim.SetTrigger(ATTACK1);
               StartCoroutine(attackTimer());
            }

            anim.SetFloat(AIRSPEEDY, playerController.linearVelocityY);

            if (playerController.inputActions.Player.Jump.triggered &&
!playerController.isRolling)
            {
               anim.SetTrigger(JUMP);
            }

            if (playerController.moveInput.x != 0)
            {
               anim.SetInteger(ANIMSTATE, 1);
            }
            else
            {
               anim.SetInteger(ANIMSTATE, 0);
```

```
            }

            if (groundCheckScript.OnGround)
            {
                anim.SetBool(GROUNDED, true);
            }
            else
            {
                anim.SetBool(GROUNDED, false);
            }
        }
        else if (!isDeathTriggered)
        {
            isDeathTriggered = true;
            //anim.SetTrigger(DEATH);
        }
    }

    private void AttackChacker()
    {
        if (checkAttack &&
playerController.inputActions.Player.Attack.triggered)
        {
            nextAttack = true;
        }
    }

    IEnumerator attackTimer()
    {
```

```
        checkAttack = true;
        yield return new WaitForSeconds(0.3f);
        checkAttack = false;
    }
}

using UnityEngine;
using static UnityEngine.GraphicsBuffer;

public class PlayerController : MonoBehaviour
{
    public GameObject attackCollider;
    public GameObject groundCheckObject;
    public GameObject playerVisual;

    public PlayerInputActions inputActions;

    public float linearVelocityY;

    public bool canSave;
    public bool hasDash;
    public bool isDead;
    public bool isRolling;
    public bool maxJumpReached;
    public bool isInvincible;

    public Vector2 moveInput;
    public Vector2 checkpointPos;
```

```
        private BoxCollider2D hitBox;
        private AttackCollider attackCollScript;
        private GroundCheck groundCheck;
        private Rigidbody2D playerRB;

        private Vector2 rollStopPoint;

        private int backstepDir;

        [SerializeField] private int holdJumpCounter;

        [SerializeField] private float rollDistance;
        [SerializeField] private float rollSpeed;
        [SerializeField] private float jumpVelocity;
        [SerializeField] private float maxJumpVelocity;
        [SerializeField] private float speed;
        [SerializeField] private float jumpForce;

        private void Awake()
        {
            inputActions = new PlayerInputActions();

            inputActions.Player.Move.performed += ctx => moveInput =
ctx.ReadValue<Vector2>();
            inputActions.Player.Move.canceled += ctx => moveInput =
Vector2.zero;
            DontDestroyOnLoad(gameObject);
        }
```

```
        private void Start()
        {
            hitBox = GetComponent<BoxCollider2D>();
            playerRB = GetComponent<Rigidbody2D>();
            attackCollScript = attackCollider.GetComponent<AttackCollider>();
            groundCheck =
groundCheckObject.GetComponent<GroundCheck>();

            checkpointPos = transform.position;

            canSave = false;
            hasDash = false;
            isDead = false;
            maxJumpReached = false;

            holdJumpCounter = 0;
        }

        private void Update()
        {
            if (!isDead)
            {
                Attack();
                linearVelocityUpdate();
                PlayerMove();
                PlayerVisualRotate();
                Jump();
                Roll();
            }
```

```
        }

        void OnEnable()
        {
            inputActions.Enable();
        }

        void OnDisable()
        {
            inputActions.Disable();
        }
        public void Checkpoint()
        {
            checkpointPos = transform.position;
            Debug.Log("Save Working");
        }

        public void Hurt()
        {
            playerRB.linearVelocity += new Vector2(0, jumpForce *
Time.deltaTime * 5);
            if (backstepDir == 0)
            {
                playerRB.linearVelocityX += jumpForce * Time.deltaTime *
5;
            }
            else
            {
```

```
            playerRB.linearVelocityX += -1 * jumpForce * Time.deltaTime * 5;
        }
    }

    private void Attack()
    {
        if (inputActions.Player.Attack.WasPressedThisFrame() &&
attackCollScript.enemyInZone)
        {
            attackCollScript.enemy.GetComponent<Enemy>().TakeDamage();
            Debug.Log("attack");
        }
    }

    private void Roll()
    {
        if (inputActions.Player.Roll.triggered && groundCheck.OnGround
&& !isRolling && hasDash)
        {
            playerRB.gravityScale = 0;
            hitBox.enabled = false;

            float direction = playerVisual.transform.rotation.y == 0 ? 1 : -1;

            rollStopPoint = new Vector2(transform.position.x + rollDistance *
direction, transform.position.y);
            isRolling = true;
        }
```

```
            if (isRolling)
            {
                transform.position =
Vector2.MoveTowards(transform.position, rollStopPoint, rollSpeed *
Time.deltaTime);

                if (Vector2.Distance(transform.position, rollStopPoint) <
0.05f)
                {
                    playerRB.gravityScale = 1;
                    hitBox.enabled = true;
                    isRolling = false;
                }
            }
        }

        private void linearVelocityUpdate()
        {
            linearVelocityY = playerRB.linearVelocityY;
        }

        private void PlayerVisualRotate()
        {
            if (moveInput.x < 0)
            {
                backstepDir = 0;
                playerVisual.transform.rotation = Quaternion.Euler(new
Vector3(0,-180,0));
                attackCollider.transform.position = transform.position + new
Vector3(-1.1f, 0, 0);
```

```
            }
            else if (moveInput.x > 0)
            {
                backstepDir = 1;
                playerVisual.transform.rotation = Quaternion.Euler(new
Vector3(0, 0, 0));
                attackCollider.transform.position = transform.position;
            }
        }

        private void PlayerMove()
        {
            if (!isRolling)
            {
                Vector3 move = new Vector3(moveInput.x, 0, moveInput.y);
                transform.Translate(move * speed * Time.deltaTime);
            }
        }

        private void Jump()
        {
            jumpVelocity = playerRB.linearVelocity.y;

            if (inputActions.Player.Jump.IsPressed() && !maxJumpReached &&
holdJumpCounter == 0 && !isRolling)
            {
                if (jumpVelocity < maxJumpVelocity)
                {
```

```
                playerRB.linearVelocity += new Vector2(0, jumpForce *
Time.deltaTime);
            }
            else
            {
                playerRB.linearVelocity += new Vector2(0, 0);
                maxJumpReached = true;
                holdJumpCounter++;
            }
        }

        if (inputActions.Player.Jump.WasReleasedThisFrame())
        {
            holdJumpCounter = 0;

            if (!groundCheck.OnGround)
            {
                playerRB.linearVelocity += new Vector2(0, 0);
                maxJumpReached = true;
            }
        }
    }
```

```
using UnityEngine;

public class PlayerHealthSystem : MonoBehaviour
{
    private UIManager uiManager;
    private PlayerController playerController;
    private PlayerAnimationController playerAnimationController;

    public GameObject player;
    public GameObject playerVisual;
    public GameObject health1;
    public GameObject health2;
    public GameObject health3;

    private void Start()
    {
        uiManager =
GameObject.Find("Canvas").GetComponent<UIManager>();
```

```
            playerAnimationController =
playerVisual.GetComponent<PlayerAnimationController>();
            playerController = player.GetComponent<PlayerController>();

            health1.SetActive(true);
            health2.SetActive(true);
            health3.SetActive(true);
        }

        public void AddHP()
        {
            health1.SetActive(true);
            health2.SetActive(true);
            health3.SetActive(true);
        }

        public void ChangeHP()
        {
            if(health3.activeSelf)
            {

playerAnimationController.anim.SetTrigger(playerAnimationController.
HURT);
                health3.SetActive(false);
            }
            else if(health2.activeSelf)
            {

playerAnimationController.anim.SetTrigger(playerAnimationController.
HURT);
```

```
            health2.SetActive(false);
        }
        else if(health1.activeSelf)
        {
            health1.SetActive(false);
            playerController.isDead = true;
            uiManager.startAnimation = true;
        }
    }
}


using UnityEngine;

public class Coin : MonoBehaviour
{
    private UIManager uiManager;

    [SerializeField] private float height = 0.5f;
    [SerializeField] private float speed = 2f;

    private Vector3 startPosition;

    private void Start()
    {
        uiManager =
GameObject.Find("Canvas").GetComponent<UIManager>();
        startPosition = transform.position;
    }
```

```
    private void Update()
    {
        transform.position = startPosition +
            Vector3.up * Mathf.Sin(Time.time * speed) * height;
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            uiManager.PlusMoney();
            Destroy(gameObject);
        }
    }
}
```

```
using UnityEngine;

public class BackGround : MonoBehaviour
{
```

```
    [SerializeField] private SpriteRenderer sprite;

    private PlayerController player;

    private Color backgroundColor;

    public bool isBlack;

    private void Start()
    {
        player =
GameObject.Find("Player").GetComponent<PlayerController>();
        backgroundColor = sprite.color;
        isBlack = false;
    }

    private void Update()
    {
        if (player.transform.position.x > -12)
        {
            isBlack = false;
        }

        if (isBlack)
        {
            sprite.color = Color.Lerp(sprite.color, Color.black, Time.deltaTime
* 5f);
        }
        else
```

```
            {
                sprite.color = Color.Lerp(sprite.color, backgroundColor,
Time.deltaTime * 5f);
            }
        }

        private void OnTriggerEnter2D(Collider2D collision)
        {
            if (collision.CompareTag("Player"))
            {
                isBlack = true;
            }
        }
    }
    using UnityEngine;

    public class PickUpRoll : MonoBehaviour
    {
        private PlayerController player;

        private void Start()
        {
            player =
GameObject.Find("Player").GetComponent<PlayerController>();
        }
        private void OnTriggerEnter2D(Collider2D collision)
        {
            if (collision.CompareTag("Player"))
            {
```

```
            player.hasDash = true;

            Destroy(gameObject);
        }
    }
}
```

```
using UnityEngine;

public class ScenePortal : MonoBehaviour
{
#if UNITY_EDITOR
    [SerializeField] private UnityEditor.SceneAsset targetScene;
#endif

    [SerializeField] private string sceneName;

#if UNITY_EDITOR
    private void OnValidate()
    {
        if (targetScene != null)
```

```
        {
            sceneName = targetScene.name;
        }
    }
#endif

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            SceneTransitionManager.Instance.LoadScene(sceneName);
        }
    }
}
using System.Collections;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class SceneTransitionManager : MonoBehaviour
{
    public static SceneTransitionManager Instance;

    [SerializeField] private Image fadePanel;
    [SerializeField] private float fadeDuration = 1f;

    private PlayerController player;
```

```
    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }

    public void LoadScene(string sceneName)
    {
        StartCoroutine(LoadSceneRoutine(sceneName));
    }

    private IEnumerator LoadSceneRoutine(string sceneName)
    {
        yield return StartCoroutine(FadeIn());

        AsyncOperation operation =
SceneManager.LoadSceneAsync(sceneName);

        while (!operation.isDone)
        {
            yield return null;
        }
```

```
    player = FindFirstObjectByType<PlayerController>();

    if (player != null)
    {
        player.transform.position =
            new Vector2(-7f, player.transform.position.y);
    }

    yield return StartCoroutine(FadeOut());
}

private IEnumerator FadeIn()
{
    float time = 0f;
    Color color = fadePanel.color;

    while (time < fadeDuration)
    {
        time += Time.deltaTime;

        color.a = Mathf.Lerp(0f, 1f, time / fadeDuration);
        fadePanel.color = color;

        yield return null;
    }

    color.a = 1f;
```

```
        fadePanel.color = color;
    }

    private IEnumerator FadeOut()
    {
        float time = 0f;
        Color color = fadePanel.color;

        while (time < fadeDuration)
        {
            time += Time.deltaTime;

            color.a = Mathf.Lerp(1f, 0f, time / fadeDuration);
            fadePanel.color = color;

            yield return null;
        }

        color.a = 0f;
        fadePanel.color = color;
    }
}
```

```
using UnityEngine;

public class AttackCollider : MonoBehaviour
{
    public GameObject enemy;

    public bool enemyInZone;

    private string ENEMY = "Enemy";

    private void OnTriggerStay2D(Collider2D collision)
    {
        if (collision.CompareTag(ENEMY))
        {
```

```
            enemyInZone = true;
            enemy = collision.gameObject;
        }
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag(ENEMY))
        {
            enemyInZone = false;
            enemy = null;
        }
    }
}
using UnityEngine;

public class GroundCheck : MonoBehaviour
{
    public GameObject playerObject;

    public bool OnGround;

    private string GROUND = "Ground";

    private PlayerController playerScript;

    private void Start()
    {
```

```
            playerScript =
playerObject.GetComponent<PlayerController>();
        }

        private void OnTriggerEnter2D(Collider2D collision)
        {
            if (collision.gameObject.CompareTag(GROUND))
            {
                OnGround = true;
                playerScript.maxJumpReached = false;
            }
        }

        private void OnTriggerExit2D(Collider2D collision)
        {
            if (collision.gameObject.CompareTag(GROUND))
            {
                OnGround = false;
            }
        }
    }
```