МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД «УНІВЕРСИТЕТ «КРОК»

Фаховий коледж Університету «КРОК»

ДИПЛОМНА РОБОТА

за темою

«Розробка гри на платформі Unity»

Студент 4 курсу групи Іпз 20к-1

Керівник дипломної роботи

_______Доцент_________

(посада керівника)

_Богуш Роман Васильович___

(прізвище, ім'я та по-батькові студента)

___Добришин Ю.Є________

(прізвище, ім'я та по-батькові керівника)

___До захисту___

(резолюція «До захисту»)

_______________ 11.06.2024 _______________

(підпис студента) (дата) (підпис викладача)

Київ, 2024 рік

Зміст

Вступ 3
Розділ 1. Середа розробки Unity 4
1.1 Типи ігор та етапи їх розробки 4
1.1.1 Типи ігор 4
1.1.2 Етапи розробки ігор 7
1.2 Функціонал Unity 10
1.2.1 Основні компоненти інтерфейсу 11
1.2.2 Робота з 2D та 3D графікою 12
1.2.3 Анімація 14
1.2.4 Скриптинг 15
1.3 Порівняння Unity та інших ігрових рушіїв 16
Розділ 2. Розробка гри на рушії Unity 26
2.1 Підготовка Unity Hub до роботи 26
2.2 Розробка гри 28
2.2.1 Підготовка графічних ассетів. 29
2.2.2 Дизайн об'єктів 31
2.2.3 Левел Дизайн 33
2.2.4 Програмування скриптів 36
2.2.5 Робота за файлами 38
Розділ 3. Готова гра 41
3.1 Персонаж 41
3.2 Вороги 42
3.3 Перехід між сценами 42
3.4 Усі інші скрипти 42
Висновки 43
Література 44
Додаток 45

Вступ

Останні 20 років індустрія відеоігор розвивається семимильними кроками, кожного року з'являються та зникають ігрові студії, кожен місяць випускаються десятки ігор найрізноманітніших жанрів. І все це стало доступним саме завдяки тому, що за останні 20 років все більше людей отримало доступ до персональних комп'ютерів та глобальної мережі інтернет. Саме з появою нових технологій, з'явився попит на новий вид розваг. Відеоігри можуть дати вам новий, незабутній досвід, та допомогти втілити раніше недосяжні бажання. Ви можете спробувати себе у ролі фермера, пілота Формули-1, або навіть у ролі відважного лицаря чи мудрого чарівника в фентезійному світі. І над тим щоб допомогти вам втілити ваші бажання, працюють десятки програмістів, менеджерів, дизайнерів. Бо насправді створення навіть простої гри може зайняти не один десяток годин, а по справжньому великі проекти розробляються не один рік.

Але історія відеоігор почалася задовго до 2000х років. Першими стали аркадні автомати, наприклад "Nimatron" – електромеханічний комп'ютер для гри з ним, який був створений в 1939-40х роках. Потім, в 1960-1970х роках створювалися маленькі ігри для тренувань та просто розваг студентів. Першими масовими засобами для розваг стали ігрові приставки під які, як і під аркадні автомати писалися ігри. Справжнім початком ігрової індустрії стали 1980 роки. Space Invaders, Duck Hunt, Super Mario, Pac-Man та інші ігри на довгі роки задали тренд тому як мають виглядати ігри. Найцікавіше те, що в ті часи лідируючу позицію по створенню ігор тримали японці, а саме Nintendo. В той час ігри писались в основному на Ассамблері або ж на С. Ще одним переломним моментом став вихід Wolfenstein 3D та Doom. Треба подякувати Джону Кармаку, Джону Ромеро та всій команді id Software за популяризацію шутерів від першого обличчя та розширення кордонів графічних технологій. Doom називають першим справжнім 3D шутером, хоча насправді він ним не є, а лише імітує ефект 3D. На той час це було проривом.

Але технології покращувалися, захопленість відеоіграми росла і почалися з'являтися ігрові движки крім Id Tech 1, який ще відомий як Doom Engine. На сьогоднішній день існує дуже багато ігрових движків на яких роблять найрізноманітніші ігри, та також сама розробка ігор стала більш доступною. Одним з найпопулярніших движків є Unity.

Розділ 1. Середа розробки Unity

1.1 Типи ігор та етапи їх розробки

1.1.1 Типи ігор

Розробка ігор це довгий та трудомісткий процес який займає як багато часу, так і багато інших ресурсів. Великі студії роками розроблять одну гру, витрачаючи мільйони доларів. Але в наш час розробка ігор стала більш доступна навіть для маленьких, незалежних розробників. Тому розробити свою гру може навіть необізнана в цій сфері людина, або ж людина без великої кількості ресурсів. Все що вам треба це час, бажання, завзятість, та YouTube. Але допустимо ви все ж вирішили розробити гру своєї мрії, яка представить іншим вашу думку в такій формі. Але ж з чого почати?

Комп'ютерна гра – це складний продукт, який містить в собі величезну кількість деталей: ігрова механіка, фізика, логіка, сюжет, дизайн рівнів та персонажів, графіка та анімація, штучний інтелект та персонажі. В основному ігри діляться на 3 типи за рівнем своєї реалізації:

1. Високобюджетні, або ж AAA проєкти - це масштабні проєкти від відомих студій, на створення яких витрачаються роки та десятки мільйонів доларів. Над продуктом працюють сотні чи навіть тисячі різних фахівців: гейм-дизайнери, художники, розробники, тестувальники, маректологи, піарники. Великі студії (такі як Electronic Arts, Ubisoft, Nintendo) винаймають чимало розробників, часто мають власні стажування та навчання, але конкуренція надзвичайно висока – потрапити в компанію поталанить не кожному. Основними характеристиками таких ігор є:

a. Графіка та звук: ААА ігри відрізняються високоякісною графікою, реалістичною анімацією та професійно записаними звуковими доріжками. Графіка таких ігор може включати детально пророблені текстури, складні моделі персонажів та оточення, а також передові технології освітлення і тіней. Наприклад, "The Witcher 3: Wild Hunt" використовує високоякісні текстури та складну систему освітлення, що дозволяє створити реалістичний і живий світ. Звукові ефекти і музика також відіграють важливу роль, створюючи атмосферу ігрового світу.
b. Сюжет та геймплей: Вони часто мають глибокий сюжет, добре розроблені персонажі та багатошаровий геймплей. Історія таких ігор може включати складні сюжетні лінії, численні персонажі з власними мотиваціями і історіями, а також вибір гравця, що впливає на хід гри. Наприклад, у "Red Dead Redemption 2" гравець стикається з моральними виборами, які можуть змінити розвиток сюжету і вплинути на кінцівку гри.
c. Приклади: The Witcher 3: Wild Hunt", " The Elder Scrolls V: Skyrim", "Metro Exodus". Ці ігри відомі своєю масштабністю, високоякісною графікою і складними сюжетами, що робить їх популярними серед гравців і критиків.

2. Середньобюджетні, або ж АА проєкти - такі продукти мають менший бюджет, до їх створення долучається менше людей. Середньобюджетні ігри можуть мати менше системних вимог, бути більш дешевими для кінцевого користувача та менш ризикованими для авторів. Студії зі створення середньобюджетних ігор також винаймають чимало розробників початкового рівня, аби передати їм найпростіші та рутинні завдання. Основними характеристиками таких ігор є:

    a. Графіка та звук: АА ігри мають менш вражаючу графіку порівняно з ААА іграми, але все одно пропонують гарну візуальну якість.

Вони можуть використовувати менш деталізовані текстури та простіші моделі, але завдяки креативному підходу до дизайну, такі ігри можуть бути дуже привабливими. Наприклад, гра "Hellblade: Senua's Sacrifice" використовує передову технологію захоплення руху для створення реалістичної анімації.

b. Сюжет та геймплей: AA ігри часто мають інноваційний геймплей та цікаві сюжети, що відрізняють їх від стандартних проектів. Вони можуть експериментувати з новими механіками та підходами, що робить їх привабливими для гравців, які шукають щось нове. Наприклад, "A Plague Tale: Innocence" пропонує унікальну комбінацію стелсу та історичного контексту.

c. "Hellblade: Senua's Sacrifice", "A Plague Tale: Innocence", "GreedFall". Ці ігри демонструють, що навіть з обмеженим бюджетом можна створити високоякісні та інноваційні проекти, які отримують визнання як серед гравців, так і критиків.

3. Інді-ігри - це проєкти, які створюються групою ентузіастів або навіть соло-розробником. Головна відмінність інді-ігор – це відсутність залучення коштів інвесторів чи видавництва для створення продукту. Незважаючи на (зазвичай) незначний бюджеті, інді-ігри часто мають цікавий сюжет, авторську атмосферну графіку та складну ігрову логіку. Основними характеристиками таких ігор є:

    a. Бюджет: Зазвичай мають обмежений бюджет, що стимулює креативність та експерименти. Розробники інді ігор часто змушені використовувати інноваційні підходи та нестандартні рішення для досягнення своїх цілей. Наприклад, гра "Hollow Knight" була створена невеликою командою, але завдяки своїй креативності та відданості проекту, вони змогли створити гру з унікальним стилем та атмосферою.

    b. Геймплей: Орієнтовані на унікальний та інноваційний геймплей. Інді ігри часто експериментують з новими ігровими механіками та

ідеями, які можуть не бути комерційно успішними у великих проектах. Наприклад, "Celeste" пропонує гравцям захоплюючий платформер з складними рівнями та емоційною історією.

c. "Hollow Knight", "Celeste", "Undertale". Ці ігри демонструють, що навіть з обмеженим бюджетом можна створити захоплюючі ігри, які отримують визнання гравців та критиків.

Також нещодавно з'явився термін AAAA гри або AAA+. Цей термін був введений компанією Ubisoft для гри Skull and Bones, щоб виділитись на фоні інших, показавши що їх гра тягне на надновий рівень де у бюджету та якості рамок немає. Хоча насправді гра була провальною та досить неякісною, тому цей термін треба сприймати як рекламний хід ніж реальний показник якості гри.

1.1.2 Етапи розробки ігор

Розробка гри, як і розробка будь-якого продукту моє свої етапи. Етапи розробки ігор є ключовими складовими процесу створення якісного та успішного ігрового продукту. Розуміння цих етапів дозволяє краще усвідомити складність і глибину роботи, яку вкладають розробники в кожен проект. Від створення концепту до підтримки гри після релізу, кожен етап має свої завдання та виклики, що впливають на кінцевий результат. Етапи розробки:

1. Створення концепту – це початок будь-якого ігрового проєкту, тому що створити гру не маючи чіткої уяви що ти створюєш неможливо. Цей етап включає в себе наступні процеси:
   a. Творчо-аналітичний процес: На цьому етапі беруть участь гейм-дизайнери та проектні менеджери, які затверджують ключові аспекти майбутньої гри. Обговорюються основні механіки гри, сюжет, дизайн персонажів, ігровий світ і інші важливі елементи. Результатом є концептуальний документ, який служить основою для подальшої розробки.

b. Комерційна пропозиція: Підготовка документів для залучення інвесторів. Це включає опис гри, її унікальні особливості, аналіз ринку, цільову аудиторію, бюджет та прогнозовані доходи. Мета цього етапу — переконати інвесторів у цінності проекту.

2. Виготовлення прототипу – це наступний важливий етап який дозволяє перевірити основні механіки гри на практиці. Тут приймають участь програмісти, художники, та графічні дизайнери. Основні задачі на цьому етапі:
   a. Створення прототипу: Чернетка гри з основними механіками. Прототип зазвичай має просту графіку і базовий функціонал, але повинен демонструвати ключові аспекти гри. Це допомагає команді зрозуміти, як буде працювати ігрова механіка і чи варто продовжувати розробку в поточному напрямку.
   b. Тестування ідей: Оцінка життєздатності концепцій та виявлення слабких місць. Прототип тестується як всередині команди, так і за участю фокус-груп. Зворотний зв'язок від тестування допомагає виявити проблеми на ранньому етапі і внести необхідні зміни.
3. Вертикальний зріз – цей етап йде після оцінки прототипу. В ньому команда створює 1-2 повноцінних рівня з достатньою деталізацією для тестування. Це допомагає команді побачити, як працюють деякі аспекти гри і як можна з нею взаємодіяти.
4. Виробництво контенту – це найдовший етап серед усіх, на ньому усі команди задіяні в розробці гри починають працювати на повну:
   a. Графіка: Створення моделей, текстур, анімацій. Графічні дизайнери створюють візуальні елементи гри, такі як персонажі, оточення, інтерфейс користувача. Важливо забезпечити, щоб графіка відповідала стилю гри та була привабливою для цільової аудиторії.
   b. Програмування: Написання коду для ігрової логіки та механік. Програмісти реалізують ігрову логіку, обробку подій, фізику та

інші аспекти гри. Наприклад, вони можуть написати код для управління персонажем, взаємодії з об'єктами та обробки колізій.

c. Звуки: Запис та інтеграція музики, звукових ефектів. Звукові дизайнери створюють звукові ефекти та музичні композиції, що доповнюють ігровий досвід. Важливо забезпечити, щоб звуки відповідали атмосфері гри та підкреслювали ключові моменти геймплею.

5. Тестування – цей етап наступає коли більша частина гри же зроблена і саме час перевірити її на наявність багів, критичних помилок та проблем ігрової логіки:

   a. Пошук багів: Процес виявлення та виправлення помилок. Тестувальники грають у гру, шукають баги та повідомляють про них розробникам.

   b. Оптимізація: Підвищення продуктивності гри. Програмісти працюють над оптимізацією коду, графіки та інших елементів, щоб забезпечити плавну роботу гри на різних пристроях. Наприклад, вони можуть зменшити використання пам'яті або покращити обробку графіки.

   c. Бета-тестування: Запуск гри для обмеженої аудиторії для збору відгуків. Бета-тестери грають у гру, надають відгуки та пропозиції щодо покращення гри. Це допомагає виявити проблеми, які могли бути пропущені під час внутрішнього тестування.

6. Софт-лонч – це випуск гри в обмеженому регіоні для тестування її в умовах ринку:

   a. Виявлення проблем: Виявляються помилки та баги які не були помічені на більш ранніх етапах.

   b. Аналіз поведінки користувачів: збір та аналіз даних про поведінку користувачів для оптимізації процесів гри.

7. Реліз - це презентація продукту усім охочим. На цьому етапі гра вже починає приносити дохід інвесторам або видавцю, а команда повністю працює на підтримку. Під час цього етапу стаються такі процеси:
   a. Маркетингова кампанія: Створення рекламних кампаній для привернення уваги гравців. Маркетингова команда працює над просуванням гри через соціальні мережі, рекламні кампанії, трейлери та інші засоби.
   b. Розповсюдження: Випуск гри на різних платформах (Steam, App Store, Google Play). Розробники працюють над тим, щоб гра була доступна для завантаження та покупки на різних платформах.

Також до 8 етапу можна внести підтримку та оновлення гри. На цьому етапі команда розробників випускає патчі для виправлення помилок та багів, і покращує продуктивність гру після релізу. Також займається створенням нового контенту для гри у вигляді нових івентів, персонажів, рівнів, або ж режими та місії. І найголовнішою частиною цього етапу є зворотній зв'язок до користувачів, де розробники слухають відгуки та побажання. Але так як цей етап не відноситься до розробки гри, а наступає після релізу, він знаходиться окремо.

1.2 Функціонал Unity

І ось якщо ви твердо вирішили створювати свою гру, ознайомилися з етапами розробки, а також по бажанню знайшли команду, можна починати вибирати рушій для розробки. Найпопулярнішими рушіями є: Unity, Unreal Engine, Godot, та GameMaker. У кожного рушія є свої плюси та мінуси, які ми розглянемо пізніше. Зараз пропоную зупинитись на Unity, та оглянути його функціонал.

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

графіці, та на пристроях віртуальної чи доповненої реальності. Застосунки, створені за допомогою Unity, підтримують DirectX та OpenGL.

1.2.1 Основні компоненти інтерфейсу

Основні компоненти інтерфейсу – інтерфейс Unity складається з кількох основних вікон, що значно полегшує розробку гри.

1. Вікно проєкту(The Project Window): Це вікно містить всі ресурси вашого проекту, такі як моделі, текстури, скрипти, звуки та інші активи. Організувати активи можна за допомогою папок, що дозволяє легко знаходити та керувати файлами. Наприклад: У вікні проекту можна створити папки для моделей, текстур та скриптів, що допоможе підтримувати порядок у проекті.
2. Вікно сцени(The Scene View): Головне робоче поле, де ви можете бачити і маніпулювати всіма об'єктами, що входять до гри. Сцена дозволяє розташовувати об'єкти, налаштовувати їх позиції, ротації, масштаби та інші властивості.

   Наприклад: Уявіть сцену як полотно, на якому ви малюєте свій ігровий світ. Якщо ви створюєте платформер, то на сцені ви розміщуєте персонажа, платформи, ворогів та інші об'єкти.
3. Вікно ієрархії(The Hierarchy Window): У цьому вікні відображається список всіх об'єктів на сцені. Вони організовані у вигляді дерева, що дозволяє швидко знаходити і вибирати необхідні елементи. Наприклад: В ієрархії ви можете створити структуру з батьківськими та дочірніми об'єктами.

   Наприклад: персонаж може бути батьківським об'єктом, а його зброя або інвентар — дочірніми.
4. Вікно інспектора(The Inspector Window): Модуль для перегляду і зміни властивостей вибраного об'єкта. Інспектор дозволяє змінювати матеріали, додавати або видаляти компоненти, налаштовувати фізичні властивості та інші параметри об'єкта.

Наприклад: Якщо вибраний об'єкт — це персонаж, в Інспекторі ви можете змінити його матеріал, додати компоненти, такі як Rigidbody для фізичної взаємодії, або анімаційний контролер для управління анімаціями.

5. Панель інструментів(The Toolbar): Панель, яка містить кнопки для основних функцій, таких як запуск та зупинка гри, збереження сцени, перемикання між режимами редагування та інші корисні команди.

   Наприклад: Використовуючи панель інструментів, ви можете швидко запускати та зупиняти гру для тестування, перемикатися між режимами перегляду та редагування об'єктів на сцені.

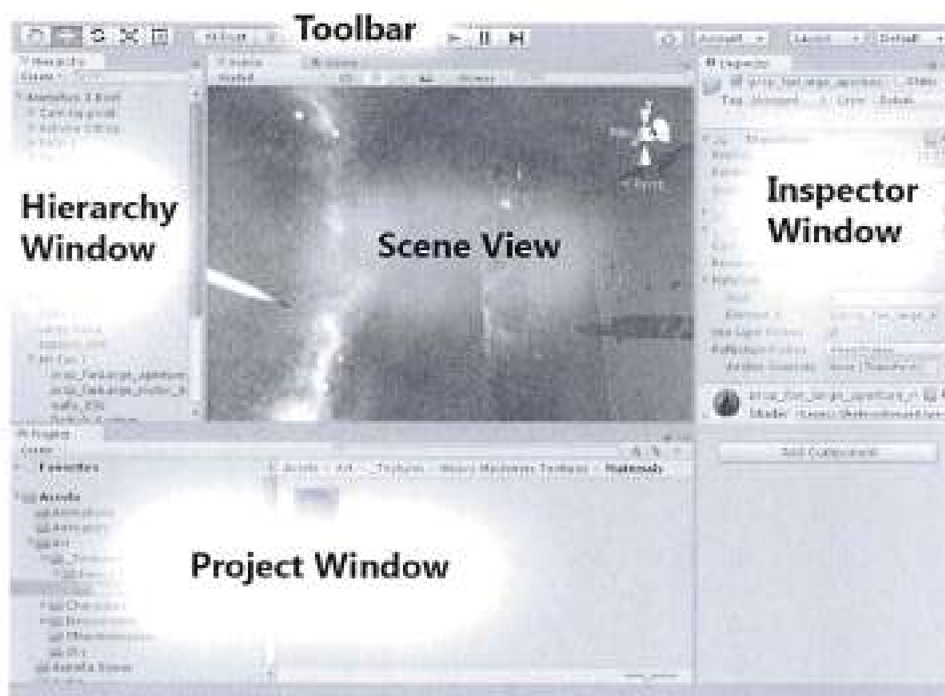

Рисунок 1.1 – Основні компоненти інтерфейсу Unity

1.2.2 Робота з 2D та 3D графікою

Unity підтримує як 2D, так і 3D графіку, що дозволяє створювати різноманітні ігри. 2D графіка: Unity надає інструменти для роботи зі спрайтами, анімаційними тайлами, 2D фізикою та іншими аспектами 2D графіки. Це включає використання компонентів Sprite Renderer для відображення спрайтів, Tilemap для створення тайлових карт, Collider 2D для обробки колізій у 2D просторі та інших.

Приклади: При створенні 2D платформера ви можете використовувати Sprite Renderer для відображення персонажа та оточення, Tilemap для створення рівнів з тайлів, та Collider 2D для обробки зіткнень персонажа з платформами. Інструменти для 2D графіки:

1. Рендеринг спрайтів(Sprite Renderer): Використовується для відображення спрайтів на сцені. Це основний компонент для роботи зі спрайтами в Unity.
   Наприклад: Додайте компонент Sprite Renderer до об'єкта і виберіть зображення спрайта для його відображення.
2. Tilemap: Інструмент для створення тайлових карт. Це дозволяє легко створювати великі рівні з повторюваних елементів.
   Наприклад: Створіть тайлову карту для рівня платформера, використовуючи різні тайли для землі, перешкод та інших елементів.
3. Колайдер 2D(Collider 2D): Компонент для обробки колізій у 2D просторі. Він дозволяє об'єктам взаємодіяти один з одним через зіткнення.
   Наприклад: Додаючи Collider 2D до персонажа та платформ, налаштуйте зіткнення, щоб персонаж міг ходити по платформам і взаємодіяти з іншими об'єктами.

Unity надає широкі можливості для роботи з 3D графікою, включаючи імпорт 3D моделей, налаштування матеріалів, освітлення, тіней, шейдерів та інших елементів:

1. Рендеринг сітки(Mesh Renderer): Використовується для відображення 3D моделей на сцені. Це основний компонент для роботи з 3D графікою в Unity.
   Наприклад: Додайте компонент Mesh Renderer до об'єкта і виберіть 3D модель для його відображення.

2. Матеріали та текстури(Materials and Textures): Налаштування матеріалів і текстур дозволяє створювати реалістичні поверхні для 3D об'єктів. Ви можете налаштовувати колір, блиск, прозорість та інші параметри матеріалу.

   Наприклад: Створіть матеріал для 3D моделі будівлі, налаштовуючи текстуру і параметри для досягнення реалістичного вигляду.
3. Освітлення та тіні(Lighting and Shadows): Освітлення і тіні додають глибини і реалізму до 3D сцени. Ви можете налаштувати різні типи освітлення, такі як точкові, направлені та світильники, а також параметри тіней для створення атмосфери.

   Наприклад: Додайте направлене світло до сцени, щоб імітувати сонячне світло, і налаштуйте параметри тіней для створення реалістичних тіней від об'єктів.
4. Шейдер(Shaders): Шейдери дозволяють створювати спеціальні ефекти для 3D об'єктів, такі як відблиски, прозорість, випромінювання та інші.

   Наприклад: Застосуйте шейдер для створення ефекту відблисків на воді або світіння для магічних об'єктів.

1.2.3 Анімація

Unity містить достатньо непогані інструменти для створення різних анімацій:

1. Анімаційний редактор(Animation Editor): Інструмент для створення анімаційних кліпів та налаштування анімаційних переходів. Анімаційний редактор дозволяє створювати анімації для персонажів та об'єктів, змінювати ключові кадри та налаштовувати плавні переходи між анімаціями. Наприклад: Ви можете створити анімацію для персонажа, що ходить, бігає або стрибає. Використовуючи Animation Editor, ви можете налаштувати ключові кадри для цих анімацій та забезпечити плавні переходи між ними.
2. Механізм анімаційних переходів(Animator Controller): Система, що дозволяє керувати анімаційними станами і переходами між ними на

основі різних умов. Наприклад, можна налаштувати анімації ходьби, бігу та стрибків для персонажа.
Наприклад: Використовуючи Animator Controller, ви можете створити логіку переходів між анімаціями персонажа залежно від швидкості руху, натискання клавіш або інших умов.

1.2.4 Скриптинг

Скриптинг на Unity здійснюється на мові програмування C#. Основні можливості включають:

1. Керування об'єктами: Створення та маніпуляція об'єктами на сцені. За допомогою C# скриптів розробники можуть створювати нові об'єкти, змінювати їх властивості, переміщувати їх на сцені та взаємодіяти з іншими об'єктами.
   Наприклад: За допомогою скрипта ви можете створити логіку руху персонажа. Наприклад, скрипт може обробляти введення користувача і змінювати позицію персонажа на сцені.
2. Обробка подій: Реакція на взаємодії користувача, такі як натискання клавіш або зіткнення об'єктів. Скрипти дозволяють обробляти різні події, що виникають під час гри, і виконувати відповідні дії.
   Наприклад: Ви можете створити скрипт, який реагує на натискання кнопки "Стрибок" і змушує персонажа стрибати. Також можна обробляти зіткнення персонажа з іншими об'єктами.

Вище був представлений основний функціонал ігрового рушія Unity. Звісно там було представлено не все, бо щоб зрозуміти усі тонкощі ігрового рушія, його треба спочатку випробувати самому. Але особисто для себе на цей раз я обрав Unity, саме через його функціонал та простоту. Але далі я спробую порівняти Unity з іншими найпопулярнішими рушіями, щоб зрозуміти. А який рушій все ж таки найкращий?

1.3 Порівняння Unity та інших ігрових рушіїв

Упродовж останніх років серед великих ігрових студій помітна тенденція до відмови від рушіїв власного виробництва. По приклади далеко йти не доведеться. CD Projekt RED відправила в архіви REDengine, а Cyberpunk 2077: Phantom Liberty стала останнім проєктом із застосуванням цієї технології. Respawn Entertainment вирішила не використовувати Frostbite під час створення Apex Legends та дилогії Star Wars Jedi. Водночас S.T.A.L.K.E.R. 2 будується без умовного послідовника X-Ray Engine.

Кожен з цих прикладів об'єднує перехід на **Unreal Engine**. І насправді список можна продовжувати: BioWare наймала фахівців, знайомих з цим рушієм, Konami заснувала студію для роботи над Silent Hill з використанням UE, і навіть вихідці з Blizzard Entertainment, які об'єдналися у Frost Giant Studios, вирішили створювати духовного наступника StarCraft на технології Epic Games.

Здається, ніби індустрія поступово рухається до спільного знаменника в цій категорії. Проте по інший бік барикад перебуває **Unity**. І хоча осінній скандал з монетизацією зіпсував репутацію компанії-розробника, не можна заперечувати, що кожного року виходять десятки прекрасних ігор, побудованих на цьому рушії.

Переваги Unity:

Почнемо з основної — простоти в опануванні. На противагу Unreal Engine, де для досягнення високого рівня майстерності потрібно розібратися з багатьма нюансами, в Unity **низький поріг входу**.

1 Розробники подбали про велику кількість документації для інструментів, реалізованих у рушієві.

2 Навколо Unity сформувалася велика та сильна спільнота. Творці завжди можуть знайти матеріали для навчання, посібники, присвяченні окремим можливостям, покрокові гайди тощо.

3 Мова програмування C# легша у вивченні, ніж C++. Це відіграє важливу роль на початковому етапі, а от щодо подальшого впливу думки розробників розділилися. Дехто вважає, що на позиціях Middle та Senior фахівцям у будь-якому напрямі IT байдуже, на чому писати. А на думку інших, Senior-розробник на Unity має глибше розбиратися в тонкощах C#, ніж його колега, який спеціалізується на Unreal Engine, у C++.

Unity створювали як продукт для широкої аудиторії. Розробники рушія подбали не лише про простоту його використання, а й про універсальність. Кожен з опитаних нами фахівців згадав, що рушій дозволяє випускати ігри та додатки різних форматів і жанрів для більшості доступних платформ. На Unity можна успішно реалізувати як простий 2D-проєкт, так і масштабний реліз. І водночас затрати коштів будуть меншими, якщо порівнювати з конкурентами.

Unity продовжують дуже активно розвивати. Розробники рушія регулярно впроваджують різноманітні новації, прислухаючись до потреб користувачів. Хоча комунікація не завжди відбувається швидко й компанія може відреагувати з запізненням, розвиток — це все-таки плюс.

У редактор рушія можна додавати новий функціонал, і це легший процес у порівнянні з конкурентами. Наявних інструментів також удосталь для розробки складних модифікацій чи власних рішень для Unity. Це важливо, коли виникають специфічні завдання.

На Unity легко ітегрувати проєкт. Можна швидко реалізувати ідею та подивитися, чи працює вона. Це спрощує процес формування фінальної концепції гри.

З активної спільноти, сформованої навколо Unity, витікає ще одна перевага — дуже багатий маркетплейс. Розробники створюють безліч асетів, плагінів та рішень, які можуть спростити створення проєкту. Частина матеріалів розповсюджується безплатно, хоча більшість виставлена на продаж. Наявність такої кількості пропозицій на маркетплейсі допомагає

заощадити час і просто придбати потрібні ресурси. На цьому наголосили декілька розробників, з якими ми поспілкувалися.

Unity має зручний користувацький інтерфейс. Усі потрібні інструменти та функції — на своєму місці. А ситуації, коли необхідно шукати, де і що вимкнути, не виникають. За словами одного з розробників, це доказ орієнтації рушія на широку аудиторію.

Як і у випадку з Unreal Engine, Unity підтримує інтеграцію за сторонніми рішеннями партнерів. Найпростіші приклади — Blender та Maya, які активно застосовують для реалізації моделей, анімацій, візуальних ефектів тощо. Імпорт файлів з двох згаданих програм утілено на рівні функціоналу, і для цього треба вибрати лише декілька пунктів у меню.

Монетизацію Unity, попри на вересневий скандал, частина розробників все ще зараховує як перевагу. До змін у фінансовій політиці рушій був найвигіднішим для розробників. Фахівці віддавали перевагу купівлі ліцензії за фіксовану вартість на противагу відрахуванням, як у випадку з Unreal Engine. Проте й зараз монетизація Unity — це перевага, особливо для невеликих інді-команд.

Тим паче, що спільноті вдалося своєю критикою домогтися позитивних рішень. Наприклад, тариф Unity Personal лишається безплатним, і платити за Unity Runtime на ньому непотрібно. Ігри з доходом менше як $1 мільйон за 12 місяців, починаючи з вересня 2022 року, не будуть обкладати комісіями. А після введення нових правил розробники зможуть вибирати між фіксованою платою та динамічною на основі кількості завантажень проєкту.

На Unity є безліч можливостей для роботи з 2D. Помітно, що під час розробки технології цьому напряму приділяли окрему увагу. Це підтверджують і двомірні ігри на рушієві, які стали справжніми хітами: дилогія Ori, Cuphead, Hollow Knight, Dave the Diver та інші.

Unity працює на будь-якій сучасній системі. Основні вимоги рушія — це операційна система не нижче як Windows 7, процесор з підтримкою

архітектури x64 та DirectX 10-ї версії як мінімум. Тобто флагманське залізо для Unity не потрібне.

Недоліки Unity:

Як і у випадку з Unreal Engine, розробники навели значний перелік недоліків Unity. Головні з них пов'язані з глобальною стратегією розвитку рушія. Причому невдоволення фахівці виражають активніше, ніж у випадку з UE.

За словами розробників, Unity часто випускає нові інструменти в наполовину готовому стані. А потім вони можуть довгий час залишатися в умовному ранньому доступі. Багато рішень досі відчуваються сирими. Як висловився один з опитаних фахівців, розробники рушія створили справжній бардак з новими можливостями.

В Unity надто багато багів. За останні п'ять років кількість помилок значно зросла. Наприклад, редактор може закритися з критичною помилкою, коли видаляєш об'єкти зі сцени комбінацією Ctrl+Z. А ще спостерігаються проблеми з використанням пам'яті на ПК. Нові важливі плагіни теж виходять з багами, що створює додаткові труднощі для розробників. А самостійно вичистити інструменти від помилок дуже складно.

Оновлення Unity можуть поламати роботу важливих функцій. Після цього доводиться довго виправляти проблеми й витрачати на це додатковий час. А найгірше, що у свіжих патчах може не бути нічого корисного для конкретних фахівців. І тоді наявність поламок нічим не компенсується.

Команда Unity намагається постійно збільшити кількість доступних інструментів. Проте частина рішень, які інтегруються в рушій, виявляються нікому непотрібними. Як приклад розробники навели систему контролю версій.

Створення багатокористувацьких ігор на Unity складніше, якщо порівнювати з Unreal Engine. Рушій апріорі немає інструментарію для розробки онлайн-

проєктів. Через це поріг входу вищий, хоча реалізація мультиплеєра цілком можлива.

В Unity є одночасно три системи рендерингу. Це дозволяє дібрати ту, яка найкраще підходить під запланований продукт у питаннях графіки та продуктивності. Проблема в тому, що ці системи максимально різні, тобто не уніфіковані.

Unity — це рушій із закритим вихідним кодом. Через це позбутися проблем, які виникають на боці рушія, буває складно, а процес займає більше часу, ніж у випадку з Unreal.

Репутаційний удар через зміну політики монетизації Unity турбує розробників. Нагадаємо, що у вересні компанія вирішила переглянути фінансову політику свого рушія. Вона хотіла ввести комісії за використання Runtime на основі кількості встановлень гри. Спільнота виступила з активною критикою, і Unity прислухалася до користувачів. Проте серед розробників усе ще відчувається осад.

Розробники загалом підкреслюють, що накопичилося дуже багато запитань до Unity та керівництва компанії. Через це вони не впевнені в подальшому позитивному розвитку рушія.

Зараз давайте розглянемо плюси та мінуси рушія Unreal Engine, та вирішимо де все ж таки краще розробляти ігри.

Переваги Unreal Engine:

Найбільшим плюсом цього рушія є наявність Blueprints Visual Scripting, або просто блупринтів. Якщо коротко, то це візуальна мова написання скриптів, яка дозволяє сформувати логіку гри без застосування програмування. Для розробників це важливо з декількох причин:

Можливість швидкого створення прототипів та тестування механік.

Зручність під час проведення експериментів, коли виникає необхідність перевірити певну ідею, проте не хочеться витрачати час на її написання з використанням C++.

Ефективність за потреби швидко реалізувати та оцінити задум у парі з геймдизайнером.

На рівні з Blueprints фахівці відзначають готові рішення «з коробки» для різноманітних проєктів — як сюжетних, так і багатокористувацьких. До багатьох з цих інструментів є відповідна документація. А за потреби розробники можуть покращити та масштабувати їх. Завдяки цій перевазі пришвидшується і спрощується створення великих проєктів. У підсумку на виробництво йде менше коштів.

Unreal Engine входить у технологічний авангард. Розробники виділяють великий спектр графічних та системних можливостей, які створюють перевагу над іншими рушіями. Epic Games постійно впроваджує нові інструменти в UE та розширює свою технологічну інфраструктуру. Особливо розробники відзначають два рішення:

Lumen — динамічну систему глобального освітлення та відображень.

Nanite — технологію автоматичного масштабування геометричної деталізації оточення в режимі реального часу залежно від відстані до об'єкта.

Розробники навели приклади й інших імплементованих у рушій технологій, які надають переваги: Chaos, MetaSounds, оновлена система Animation Blueprints, Control Rig, Niagara. А ще Epic Games додає в інструментарій Unreal Engine цікаві рішення з маркетплейсу.

Unreal Engine готовий для роботи з консолями дев'ятого покоління — PlayStation 5 та Xbox Series X/S. Підтримка реалізована на базовому рівні, тому для розробників, які хочуть випускати ігри на всіх актуальних платформах, це важливий фактор.

Unreal Engine — це рушій з відкритим вихідним кодом. Тобто розробники можуть будь-якої миті вносити в нього правки відповідно до потреб гри. А ще це значно спрощує локалізацію та виправлення помилок, бо фахівцям доступний моніторинг виконання процесів на боці рушія. До того ж відкритий код у парі з готовими рішеннями дозволяє зекономити на кількості спеціалістів у команді.

Epic Games пропонує вигідну модель монетизації Unreal Engine для розробників. Студії платять 5% відрахувань з кожної проданої копії. Проте комісії починають діяти, лише коли дохід від проєкту перевищує $1 мільйон. Це особливо вигідно незалежним командам і дозволяє виділитися на фоні конкурентів. Хоча розробники зазначають, що для ААА-проєктів така цінова політика підходить не завжди.

Фахівці, які вибрали Unreal Engine, мають змогу напряму контактувати з Epic Games. Для цього створено спеціалізований форум Unreal Developer Network, який доступний ліцензованим розробникам. Якщо виникнуть специфічні проблеми, можна звернутися по рішення та пораду до творців рушія.

Unreal Engine має широку індустріальну підтримку. Тобто рушій сумісний зі сторонніми рішеннями, підтримку яких реалізувала Epic Games. Наприклад, розробники можуть додавати асети з Blender просто під час сеансу в UE. Для цього необхідно скористатися двома додатковими надбудовами.

Популярність Unreal Engine мотивує фахівців вивчати рушій. Завдяки цьому на ринку з'являється багато розробників, знайомих з технологією. І в підсумку студіям легше знайти фахівців для створення гри на UE.

Виходить таке собі замкнуте коло, у якому переплелися відомі проєкти на Unreal Engine та кваліфіковані спеціалісти, що знаються на рушієві. Кількість перших і других взаємопов'язана, бо зростає пропорційно. Саме тому зараз немає проблем з пошуком розробників для проєкту на UE. Для багатьох студій це стає додатковим позитивним фактором.

Недоліки Unreal Engine:

На дев'ять переваг Unreal Engine знайшлося майже стільки ж проблем. Деякі з них стосуються локальних аспектів, проте є й вагомі недоліки, які можуть вплинути на рішення щодо вибору рушія. З останніх і почнемо, поступово переходячи до менш значущих мінусів.

Велику кількість рішень «з коробки» та передових технологій розробники записали до переваг. Проте в цьому ж аспекті приховані й одні з найвідчутніших проблем.

Epic Games намагається створити універсальний рушій, який зможе задовольнити будь-які потреби. В окремих випадках це стає проблемою, бо фахівцям доводиться миритися з безліччю непотрібних конкретно для їхніх завдань інструментів. Вони здатні впливати на рендер або займати зайвий обсяг. Якщо ж спробувати цього позбутися, можна створити собі додатковий і доволі сильний головний біль.

Готові рішення «з коробки» завжди мають недоліки. Їх потрібно самостійно виправляти та налаштовувати під себе. Можливість зробити це — однозначний плюс. А от необхідність витрачати додатковий час — помітна проблема.

Чимало інструментів Unreal Engine потрібно вміти правильно використовувати. Особливо це стосується нових технологій, як-от Lumen та Nanite. У 2023 році вийшло багато проєктів на UE5 з рівнем графіки, якого можна було досягнути й на UE4, проте з напрочуд поганою оптимізацією. Серед них, наприклад, Immortals of Aveum та Remnant II. А в RoboCop: Rogue City технічний стан хоч і непоганий, проте візуальна складова відчувається застарілою.

Такі проблеми виникають саме через нові інструменти, які розробники не встигли достатньо вивчити й проаналізувати, щоб доцільно

використовувати. Технічної документації для цих рішень поки що бракує, а тому переважно доводиться на практиці шукати правильний підхід для кожного випадку. І на поточному етапі це призводить до проблем з оптимізацією навіть за відсутності передової графіки.

Epic Games намагається регулярно покращувати Unreal Engine, однак нові версії зазвичай нестабільні й мають різноманітні баги. Більшість помилок розробники швидко виправляють, проте деякі лишаються надовго.

Unreal Engine 5 сильно зав'язаний на використанні одного потоку процесора для ігрової логіки. Рушій має інтерфейс для виконання завдань і в багатопотоковому та асинхронному режимі. Проте велика кількість обчислень відбувається саме на одному потоці. Через це зростає складність для розробників, які планують працювати з UE5.

Усі переваги Unreal Engine стосуються 3D-проєктів. Якщо ж необхідно створити 2D-гру, то рушій здасться просто незручним. Потрібні для цього інструменті є, проте Epic Games не фокусувалася на двомірній графіці. Одним з варіантів може бути об'єднання 3D та 2D, однак створення простих 2D-проєктів на Unreal Engine ризикує виявитися складнішим та дорожчим, ніж з використанням інших рушіїв.

На нашому форумі про це вдало висловилася розробниця під псевдонімом Mary R. Вона поділилася досвідом переходу на Unreal Engine після Unity й швидко помітила описану проблему.

На мобільних платформах гра на Unreal Engine матиме розмір приблизно на 70-100 Мб більший ніж при створенні на Unity. В окремих випадках такий обсяг може стати негативним фактором для проєкту.

Для Unreal Engine 5 потрібно мати потужний персональний комп'ютер. Системні вимоги рушія — це чотириядерний процесор з тактовою частотою від 2,5 ГГц, 8 Гб оперативної пам'яті та відеокарта NVIDIA GeForce RTX 2000-ї серії або AMD Radeon 6000-ї серії. І це лише необхідний мінімум,

бо команда Epic розробляє рушій на значно потужніших ПК з AMD Ryzen Threadripper Pro 3975WX (32 ядра / 64 потоки), RTX 3080 і 128 Гб DDR4-3200.

Компіляція C++ та розгортання проєкту на Unreal Engine займають певний час. Розробники не вважають це проблемою технології, але зазначають, що доведеться звикнути.

Ось я провів порівняльну характеристику і ви можете спитати так який рушій краще?

Правильної відповіді на запитання, який з цих рушіїв кращий, немає. Завжди потрібно обирати відповідно до завдань на проєкті. Як Unity, так і Unreal Engine будуть по своєму вигіднішими в конкретних випадках.

Та якщо конкретизувати, то вивчення Unity — це простіший шлях у геймдев. Низький поріг входу й велика спільнота дають змогу відносно швидко здобути необхідні вміння та почати працювати над проєктами. Велика кількість готових матеріалів на маркетплейсі дозволяє заощаджувати час. А для створення 2D-проєктів це найкращий вибір завдяки широкому інструментарію.

Тобто Unity люблять за простоту, універсальність і легкість в опануванні. Проте в рушія є проблеми з репутацією та подальшим розвитком. У розробників складається враження, ніби команда хапається за все одразу: намагається розширити функціонал та вводить непотрібні рішення, ламаючи справді важливі. А нещодавній скандал з фінансовою політикою тільки погіршив ситуацію. Та й для AAA-проєктів рушій не дуже підходить, і розробники це окремо підкреслюють.

Натомість Unreal Engine намагається стати галузевим стандартом. Epic Games хоче створити потужний рушій з дуже багатим функціоналом та високою зручністю завдяки Blueprints і відкритому коду. Компанія постійно реалізовує технологічні рішення, які мають спростити та здешевити розробку. А ще — підтримує зв'язок зі своїми клієнтами.

Водночас в Unreal Engine вистачає своїх проблем. Мала кількість документації для нових технологій породжує їх неправильне застосування, через що кульгає оптимізація. Готові рішення потрібно налаштовувати під себе й дописувати, а свіжі інструменти не те щоб завжди є корисними для розробників. Та й для реалізації певних ігрових концепцій Unreal Engine просто не підходить.

Розділ 2. Розробка гри на рушії Unity

2.1 Підготовка Unity Hub до роботи

Перед початком розробки гри треба встановити рушій на якому ви вирішили розробляти гру. Візьмемо за приклад Unity на якому розроблялася гра яка буде представлена далі. Актуальну версію Unity ви зможете знайти та завантажити на офіційному сайті Unity: https://unity.com. Після завантаження та встановлення його на ваш комп'ютер, зайдіть в Unity Hub та проведіть усі потрібні вам налаштування: Крок 1 – Якщо у вас немає акаунту Unity, створіть його на сайті Unity, або ж через сам Unity Hub, все що вам буде потрібно буде знаходитися на основній панелі зліва:

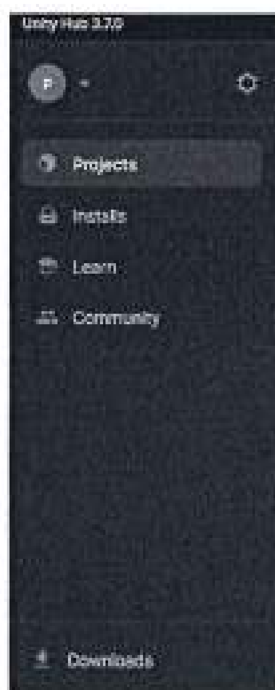

Рисунок 2.1 – Панель інструментів Unity Hub

Крок 2: Активація ліцензії - У меню "Preferences" перейдіть до розділу "License Management". Виберіть тип ліцензії (безкоштовна Personal або платна Pro/Plus) та активуйте її. Меню Preferences виглядає як шестерня на панелі інструментів у правому верхньому кутку. Перед тим як вибрати версію для вашого Unity, ознайомтесь з інформацією на офіційній сторінці Unity.

Крок 3: Встановлення Unity Editor – Оберіть бажану версію Unity для вас, рекомендую обирати останню. Для цього У Unity Hub перейдіть на вкладку "Installs". Натисніть кнопку "Add" і виберіть бажану версію Unity Editor. Виберіть додаткові модулі та платформи, які вам потрібні (наприклад, підтримка Android або iOS). Натисніть "Next" і дочекайтеся завершення завантаження та встановлення обраної версії Unity Editor.

Крок 4: Створення нового проєкту - Перейдіть на вкладку "Projects" у Unity Hub. Ця вкладка знаходиться справа від панелі інструментів і є найбільшою вкладкою в Unity Hub.

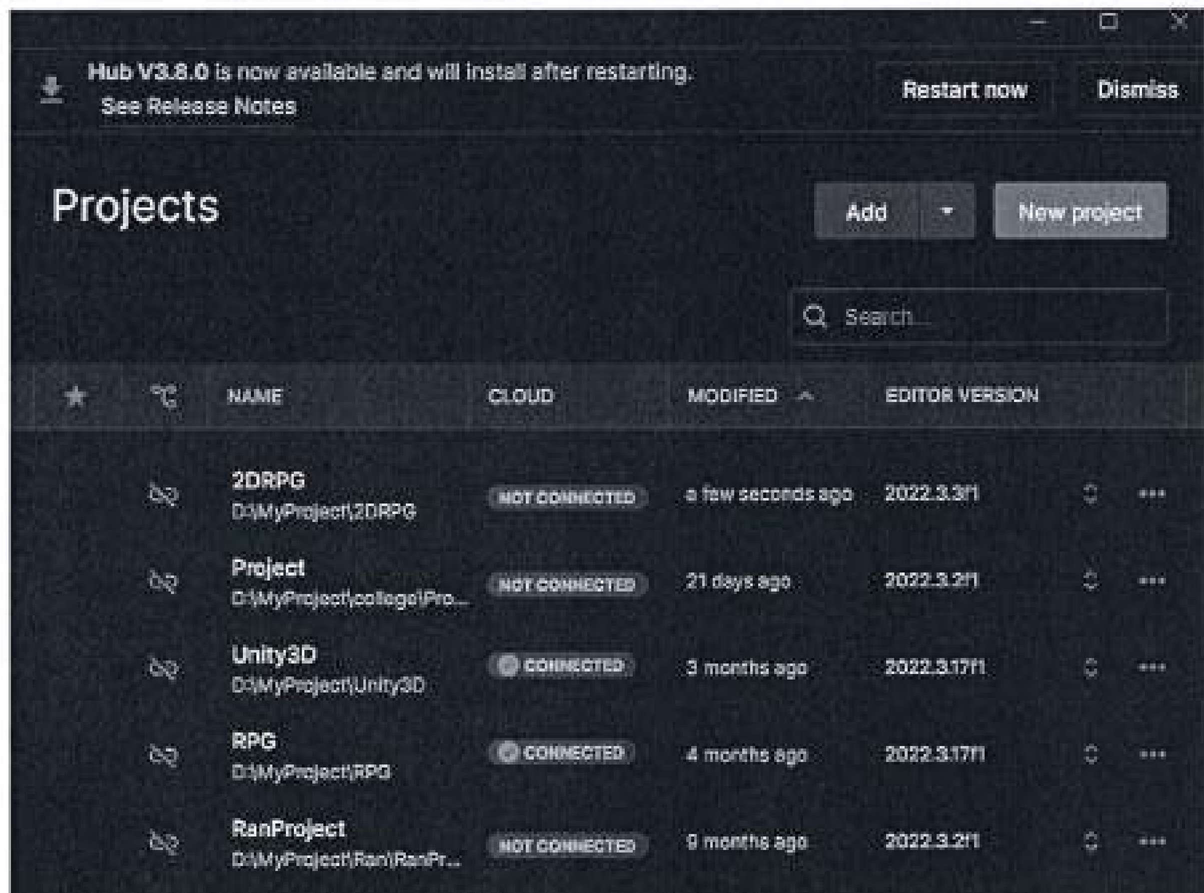

Рисунок 2.2 – Вкладка Projects

Далі натисніть кнопку "New Project". Виберіть шаблон проекту (наприклад, 2D, 3D, HDRP, URP). Вкажіть ім'я проекту та шлях для збереження. Натисніть "Create" для створення нового проекту. Дочекайтеся завершення створення та відкриття проекту в Unity Editor.

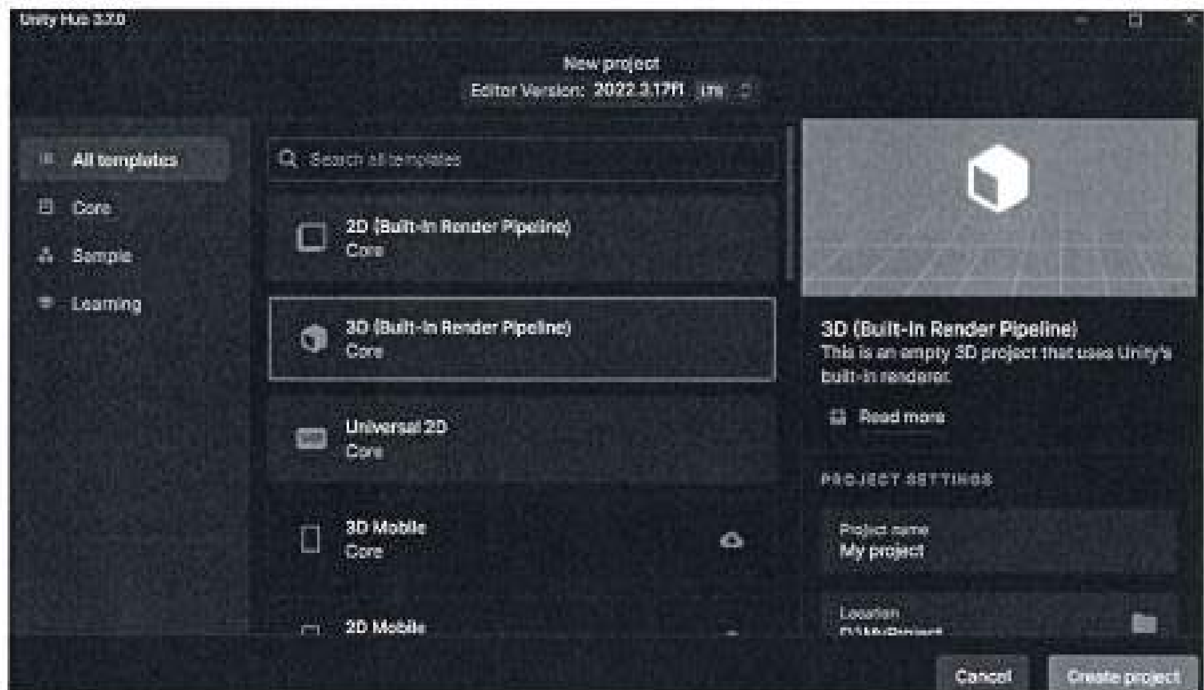

Рис 2.3 – Вкладка створення проєкту

На даному етапі основне налаштування Unity Hub та створення вашого проєкту було завершено, далі можете приступати до основної роботи.

2.2 Розробка гри

Коли проект був створений та завантажений у вас відкриється Unity Editor, детальніше про інтерфейс Unity Editor було написано в Розділі 1. Спочатку може бути нічого не зрозуміло, тому що Unity не славиться найпростішим інтерфейсом, і на мою думку він є трошки застарілим, але насправді розібратись буде дуже легко. Коли з основним інтерфейсом все стане зрозуміло, ви можете перейти до налаштування самого проєкту.

У меню “Edit” виберіть “Project Settings”.

1. Graphics: Налаштування графічних параметрів, включаючи якість та рендеринг.
2. Physics: Налаштування фізичних параметрів, таких як гравітація та обробка колізій.
3. Input: Налаштування вводу/виводу, включаючи налаштування клавіш та контролерів.

Але на самому початку не раджу змінювати будь-які налаштування якщо ви не ознайомились уважно з інструкцією.

Рисунок 2.4 – Project Settings

Коли основні налаштування були зроблені, саме час перейти безпосередньо до створення гри.

2.2.1 Підготовка графічних ассетів.

Розробка гри розпочинається з пошуку та підготовки потрібних вам ассетів. Але що таке ассети? Ігровий ассет (від англ. game asset) - це будь-який елемент, що використовується в процесі створення ігрового проекту. Ігрові ассети включають:

1. Графічні елементи: 2D спрайти, 3D моделі, текстури, анімації.

2. Аудіо компоненти: звукові ефекти, музика, голосові доріжки.
3. Скрипти та коди: програмні коди для управління ігровою логікою, поведінкою об'єктів.
4. Інші ресурси: шейдери, карти рівнів, налаштування фізики.

Ігрові ассети є базовими будівельними блоками, які використовуються для створення ігрових сцен, персонажів, середовища та всіх інших аспектів гри. Але на даному етапі давайте додамо графічні елементи, а саме текстури. Я брав безкоштовний текстур пак з сайту https://itch.io/. Але для початку раджу взяти текстур паки з самого сайту Unity, зараз покажу як це робити:

1. За допомогою інтерфейсу Unity, переходимо у вкладку вікна. У цій вкладці натискаємо на Asset store.

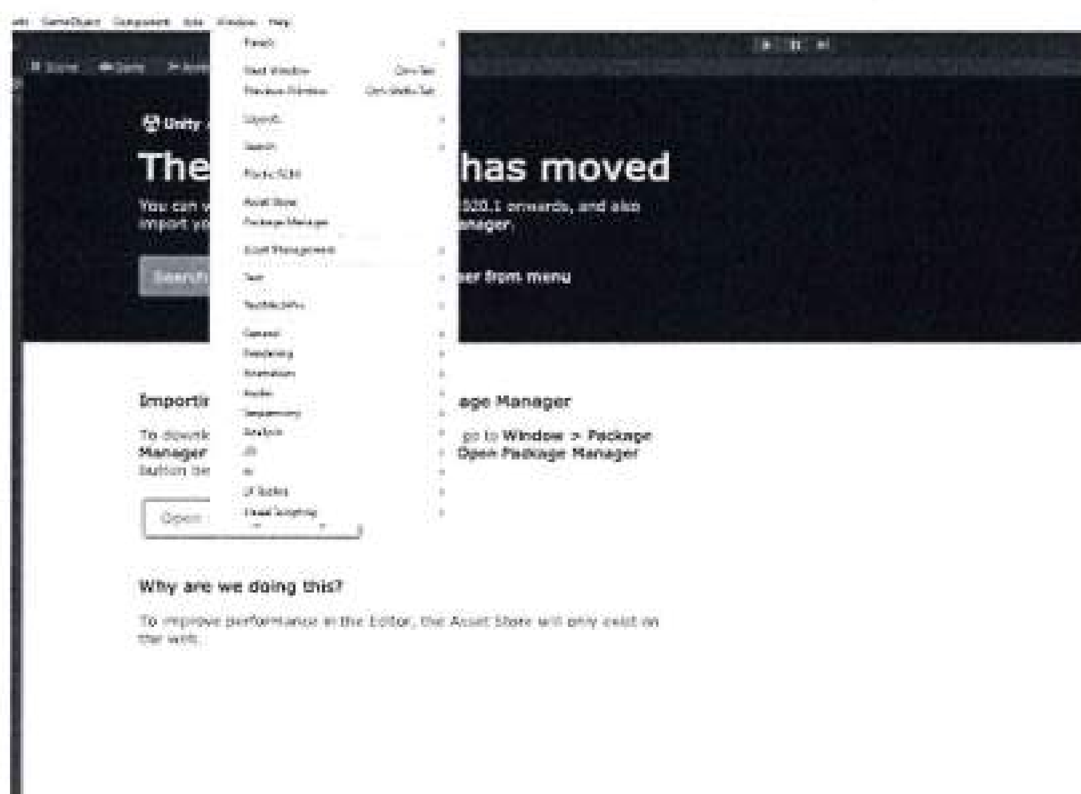

Рисунок 2.5 – Вкладка вікна

2. Далі ми відчиняємо браузер та менеджер пакетів натискаючи по черзі на дві сині клавіші у відчиненому вікні.

Рисунок 2.6 – Менеджер пакетів

3. Далі знаходимо та обираємо необхідний асет, встановлємо його з браузера і натискаємо download у менеджері пакетів.
4. Встановивши ассет, імпортуємо його в проєкт, натискаючи клавішу імпорт
5. Після недовго очікування, заходимо в папку яка знаходиться серед ассетів. Відчиняємо її і розпочинаємо ретельний огляд файлів.
6. Під час огляду видаляємо файли які нам не потрібні, а також переміщуємо їх в основну папку проєкту.

2.2.2 Дизайн об'єктів

Об'єкти в Unity - це основні елементи, які складають сцени ігрового світу. Вони включають в себе:

1. GameObjects: Основні будівельні блоки в Unity, які можуть представляти персонажів, реквізит, камери, світла тощо. Вони можуть містити різні компоненти, такі як фізичні властивості або скрипти.
2. Components: Складові частини GameObjects, які додають функціональність. Наприклад, компоненти фізики, рендерингу, аудіо тощо.
3. Assets: Ресурси, які використовуються для створення об'єктів, такі як моделі, текстури, звуки і скрипти.

Об'єкти можуть бути створені, налаштовані і керовані в Unity Editor, що дозволяє розробникам створювати складні ігрові світи.

Після того як ассети були встановлені та додані у папку, ми створюємо об'єкти. Для цього треба обрати необхідний спрайт для створення якогось елементу, наприклад головного героя. Далі створюємо необхідний ігровий об'єкт в полі проєкту. Для цього натискаємо правою клавішею в колонці ієрархії. Обираємо тип об'єкту, а також бажаний шаблон.

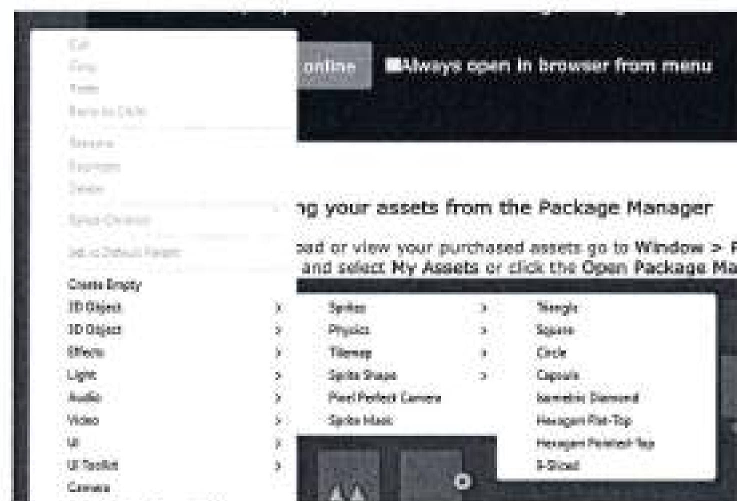

Рисунок 2.7 – Поле проєкту

Створивши об'єкт, починаємо додавати до нього необхідні компоненти у вікні інспектора. Такі як спрайт, скрипт, колайдери та інші.

Рисунок 2.9 – Вікно інспектора

Виконавши всі ці дії, додаємо елемент до префабів і якщо потрібно очікуємо реалізації внесених компонентів. (написання скрипту, знаходження аудіо треку, створення анімації).

2.2.3 Левел Дизайн

Левел-дизайн - це процес створення рівнів (сцен) для відеоігор. Він включає в себе планування, макетування та розробку ігрових середовищ, у яких гравці будуть взаємодіяти. Основні аспекти левел-дизайну включають:

1. Створення макетів: Розробка схем та планів рівнів, визначення розташування об'єктів, маршрутів гравців та зон взаємодії.
2. Балансування геймплею: Забезпечення рівномірного рівня складності, розподіл викликів і нагород.

3. Атмосфера і естетика: Визначення візуального стилю та атмосфери рівня, налаштування освітлення, звукових ефектів та інших елементів, що створюють настрій.

Використовуючи ассети які ми завантажили, та інструментарій Unity, ми почнемо створення ігрового рівня:

1. Перше що треба зробити це обрати поле, розмір карти для майбутнього рівня. Для цього треба зайти в вкладку сцена і віддалити камеру для найбільш зручного розташування.

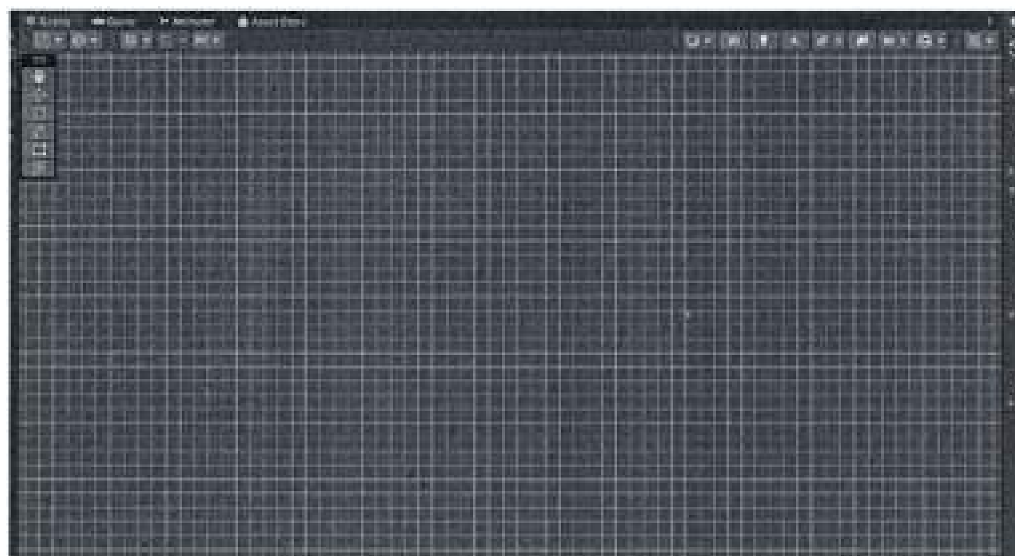

Рисунок 2.10 – Вкладка Scene

2. Наступним кроком буде створення необхідних елементів для подальшого створення рівня. Наприклад Tilemap (background). Для цього треба натиснути правою клавішею в порожньому полі ієрархії, Та обрати об'єкт tilemap.

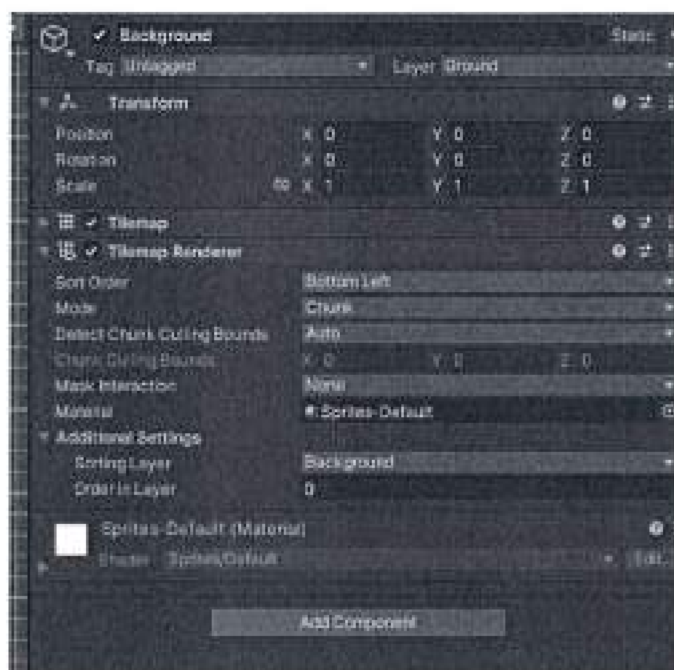

Рисунок 2.11 - Tilemap

3. Переходимо до підготовки спрайтів.
4. Серед спрайтів обираємо необхідні для рівня, наприклад terrain.
5. Відчиняємо спрайт і одразу обираємо мод спрайтів і кількість пікселів.
6. Наступним кроком буде перехід в sprite editor. Далі ріжемо спрайт на необхідні квадрати, компоненти рівня. В sprite editor натискаємо на Grid by sells size. Обираємо розмір квадратиків на які буде розрізаний спрайт, далі натискаємо клавішу slice і apply.
7. Тепер можна переносити готовий спрайт на необхідне поле tilemap.
8. Після збереження, можна переміщувати спрайти на сцену, створюючи рівень.
9. При створенні рівня використовуємо вже готові префаби, переміщуючи їх в місця які є найбільш актуальними для цього.

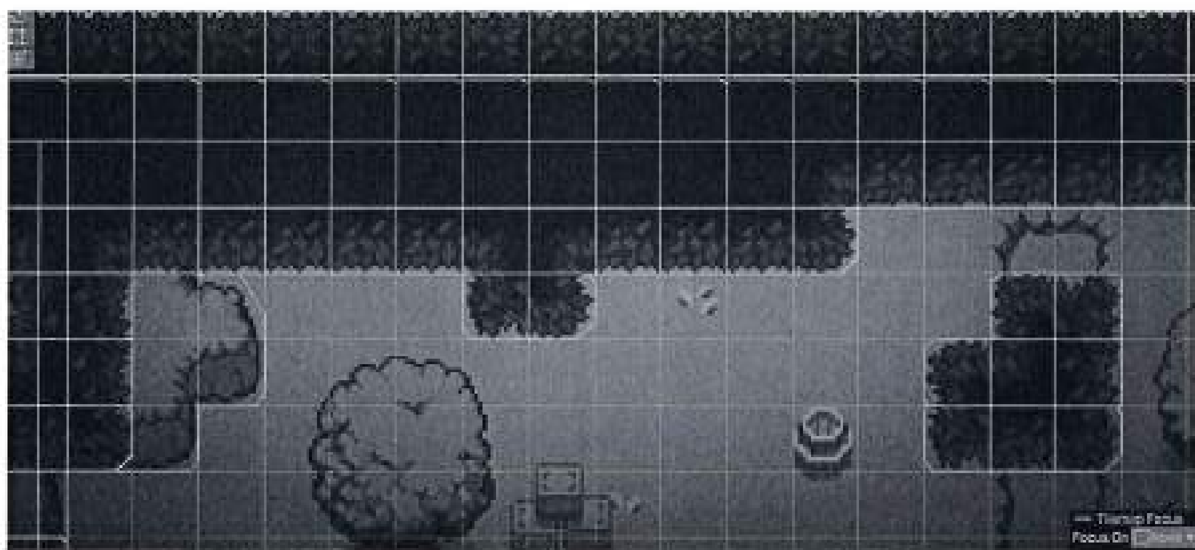

Рисунок 2.12 – Створений рівень

Створення рівня є однією з найважливіших задач при створені гри. Тому я раджу як швидше створити перший тестовий рівень для того щоб вам було простіше тестувати інші аспекти гри.

2.2.4 Програмування скриптів

Далі коли у вас є перший тестовий рівень, та моделька персонажа, або інших істот, саме час "оживити" вашу гру, а саме – зайнятися програмуванням скриптів.

Скрипт у програмуванні - це набір інструкцій, написаних мовою програмування, що виконується для автоматизації завдань, створення ігрової логіки, або керування об'єктами. У контексті Unity скрипти найчастіше пишуться мовою програмування C#.

1. Керування об'єктами: Зміна властивостей об'єктів, таких як позиція, ротація, масштаб.
2. Реакція на події: Взаємодія з гравцем, колізії, тригери.
3. Ігрова логіка: Реалізація механік гри, штучного інтелекту, керування рівнями.

Створення скриптів це дуже важкий та довгий процес, але в своїй послідовності дуже простий. У вас є завдання наприклад зробити щоб

персонаж міг рухатися, і ви використовуючи свої знання з C# та Unity, пишете для цього скрипт використовуючи логіку:

1. Спочатку треба визначити який скрипт вам потрібен, давайте візьмемо базовий скрипт для руху персонажу.

2. Створюємо скрипт та відчиняємо його у зручному для вас середовищі розробки. Наприклад візьмемо Visual Studio Code.

Рисунок 2.13 – Середовище розробки

3. Створюємо необхідні функції та елементи.

Рисунок 2.14 – Приклад скрипту

4. Далі після написання скрипту потрібно перейти до тестування написаного та виправлення помилок. Для цього переходимо на необхідну сцену з рівнем де є кнопка. Або ж заходимо на тестову сцену

яку ми створили раніше і додаємо елемент кнопки вручну. Для цього переходимо в префаби та обираємо елемент, а потім переносимо його на сцену(рис 2.15).

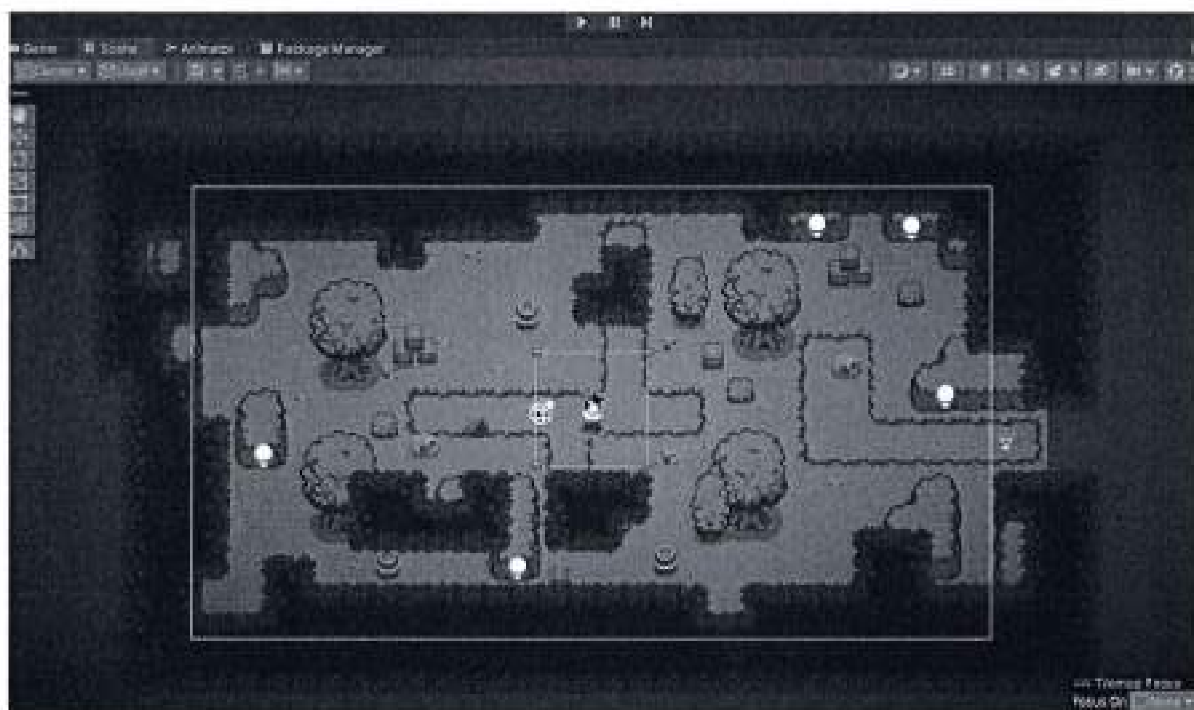

Рисунок 2.15 – Готова сцена гри

5. Запускаємо гру і розпочинаю тестування. Для цього треба натиснути стрілочку в верхній частині екрану

Ось і все, як було сказано вище – створення скриптів дуже проста дія в своїй послідовності, але важка в реалізації.

2.2.5 Робота за файлами

І ось коли ви опанували майже усі кроки потрібні в розробці гри, передостаннім кроком буде упорядкування вашої роботи задля комфорту та оптимізації вашої праці. Тому що набагато приємніше коли все упорядковано по папкам, а ніж коли все знаходиться в купі. Багато хто вважає цей крок непотрібним, але завжди треба після себе залишати чистоту.

1. Почнімо з відкриття вікна проєкту, для цього зайдемо у вкладку вікна, і оберемо проєкт. Або натисніть Ctrl+5.

2. Далі перевіримо сцени на порядок елементів. Зазвичай всі елементи знаходяться нагорі і тому зачиняють практично все видиме місце ієрархії(рис 2.16).

Рисунок 2.16 – Упорядкування елементів

3. Створимо батьківські елементи за якими можна буде розсортувати гру елементи гри. Наприклад: кнопки, інструменти, речі, ворогів. Створити їх можна натискаючи правою клавішею на порожньому полі ієрархії і обираючи порожній об'єкт(рис 2.17).

Рисунок 2.17 – Створення пустого об'єкту

4. Розміщуємо спільні елементи в новий об'єкт та змінюємо його назву.
5. В кінці перевіряємо щоб всі об'єкти ієрархії були на своїх місцях і не перешкоджали одне одному. Тобто щоб елементи були в необхідних батьківських елементах і при програванні функцій таких як увімкнення, вимкнення чи знищення, не були випадково задіяні елементи які не мають бути задіяні в цих функціях. Далі можемо скрити всі об'єкти натиснувши на маленьку стрілочку яка вказує до низу
6. У відчиненому вікні проєкту маємо оглянути і так само зробити сортування всіх файлів з перевіркою їх розширення. Якщо файли мають не прийняте для нас розширення (тобто можуть мати більшу вагу, або гіршу якість) то треба їх замінити.

Після упорядкування елементів і закінчення вашої роботи, гру треба протестувати задля знаходження помилок. Тестування проходить достатньо легко. Вам треба запустити проєкт. Почніть з першого рівня, і з цього моменту ви маєте натискати різні клавіші, оглядаючи як вони працюють і чи не призводять до збоїв. Робіть ті дії які були задумані в грі і спробуйте зробити ті які не були задумані. Запам'ятайте що користувачі не мають інструкції і бачення того як треба проходити ігри у всіх різне. Ходіть в одне та в інше місце. Збирайте різні предмети, намагайтеся вбити персонажа дивлячись де він з'явиться, а також перевіряйте наскільки важко пройти рівень. Якщо все добре, то перевірка закінчена і рівень помічається як повністю завершений. З цього моменту робота над рівнем закінчується.

На цьому етапі ми знаємо достатньо щоб спробувати створити першу гру.

Розділ 3. Готова гра

Моїм завданням було створити свою гру на рушії Unity. Я обрав створити 2D Roguelike RPG:

1. Рольова гра (RPG) – це жанр відеоігор, в яких гравці беруть на себе ролі персонажів у вигаданому світі. Основні елементи RPG включають розвиток персонажа, виконання завдань, збирання предметів, взаємодію з іншими персонажами та світом. Сюжет і вибір гравця часто мають велике значення для гри. Приклади: "The Witcher", "Final Fantasy", "Skyrim".
2. Roguelike – це піджанр RPG, що характеризується процедурною генерацією рівнів, постійною смертю персонажа (перманентна смерть), і покроковим геймплеєм. Ігри цього жанру часто мають високу складність і значну частку випадковості. Приклади: "Rogue", "NetHack", "The Binding of Isaac".

Тому задля цього я створив декілька тестових рівнів з різними ворогами. Така, як здається, маленька гра, містить в собі близько 40 скриптів, та багато годин плідної праці. Але давайте розберемо основі механіки які присутні в даній роботі.

3.1 Персонаж

В даній грі ми граємо за безіменного головного героя(рисунок 3.1) який опинився один в таємничому лісі повному монстрів і для того щоб герой зміг рухатися треба прописати йому скрипт руху. Насправді в Unity існує "Input Action Assets". Input Action Assets – це компонент системи введення в Unity, який використовується для визначення та керування ввідними діями гри. Вони дозволяють розробникам зручно налаштовувати та обробляти різні типи введення (клавіатура, миша, контролери) за допомогою спеціального інтерфейсу. За допомогою Input Action Assets можна легко створювати комплексні схеми введення, які адаптуються під різні платформи та пристрої, забезпечуючи гнучкість і контроль над ввідними подіями в грі. І зазвичай всі

дії я прописую в різних скриптах для легшої навігації між ними, але розробник ігор має опанувати ігровий рушій. Саме тому в цей раз я використав Input Action Assets. Перейшовши за цим посиланням ви зможете навчитись цим компонентом:

https://docs.unity3d.com/Packages/com.unity.inputsystem@0.9/manual/ActionAssets.html. Використовуючи цей компонент я зміг зробити скрипт для руху, битви та інвентаря героя. Далі був написаний скрипт для відновлення здоров'я, нанесення пошкодження ворогам, витривалості та ривку. І ще декілька незначних як анімація удару та перемикання між зброєю. Усі скрипти будуть представлені в додатку.

3.2 Вороги

Жодна гра не може обійтись без виклику гравцям і в RPG іграх таким викликом стають вороги. Вороги – це зазвичай неігрові персонажі які керуються штучним інтелектом. Для своєї невеличкої гри я обрав за ворогів 2 типи слизі: синя та фіолетова. Їх різниця буде складати не лише в кольорі а ще й в типах атак, сині атакують в ближньому бою коли фіолетові стріляють виноградинами. Для того щоб вороги могли функціонувати їм треба прописати загальний скрипт, який буде відповідати за рух. Далі був написаний скрипт здоров'я, слідування за персонажем, атаки. Усі скрипти будуть представлені в додатку

3.3 Перехід між сценами

Під створенням кількох рівнів мається на увазі створення кількох сцен. Тому що, насправді ігровий рівень це і є ігрова сцена, тому якщо ми хочемо перейти з точки А до точки Б треба створити другу сцену, налаштувати її і потім створити скрипт переходу. Таких скриптів буде декілька, скрипт початку рівня, скрипт переходу на інший рівень.

3.4 Усі інші скрипти

Під усіма іншими скриптами мається на увазі скрипт камери та різні скрипти UI які дуже необхідні для того щоб гра була повноцінною.

Висновки

Розробка ігор є однією з найперспективніших галузей у сучасному світі технологій. Це не тільки можливість проявити свою творчість і створити щось унікальне, але й шанс долучитися до індустрії, яка стрімко розвивається та приносить значні прибутки. Однак, шлях розробника ігор не є простим. Процес створення гри вимагає значних зусиль, часу, знань та навичок у різних сферах, включаючи програмування, дизайн, анімацію, звукорежисуру та управління проектами.

Розробка гри часто супроводжується безліччю викликів: технічні труднощі, необхідність вирішення креативних завдань, багатозадачність та необхідність постійного навчання. Незважаючи на ці труднощі, кожен етап процесу - від початкового задуму до остаточного релізу - може бути надзвичайно задовольняючим. Успішна розробка гри демонструє, що старання та відданість можуть перемогти всі перешкоди.

У цьому контексті важливо підкреслити, що кожен, хто має бажання та натхнення, може досягти успіху в цій галузі. Незалежно від того, чи ви новачок, чи вже маєте певний досвід, почати створювати свою гру можна в будь-який момент. Важливо лише мати пристрасть до цього процесу і готовність навчатися та вдосконалюватися.

Закликаю всіх, хто відчуває внутрішній поклик до створення ігор, не відкладати цю мрію на потім. Почніть з малого - навчіться основ програмування, досліджуйте різні інструменти для розробки, знайомтеся з досвідом інших розробників. Крок за кроком ви зможете перетворити свої ідеї на реальність. Пам'ятайте, що навіть найвідоміші та найуспішніші ігри починалися з маленького задуму, який був втілений у життя завдяки старанню та наполегливості їх творців. Тому не бійтеся почати свою подорож у світ розробки ігор - результати можуть перевершити всі ваші очікування.

Література

1. Офіційна документація Unity [Електроний ресурс] – Режим доступу до ресурсу: https://docs.unity.com/
2. Офіційний форум Unity [Електроний ресурс] – Режим доступу до ресурсу: https://forum.unity.com/
3. Learning C# by Developing Games with Unity 2021 / Harrison Ferrone
4. Mastering Unity 2D Game Development / Simon Jackson
5. Unity in Action: Multiplatform Game Development in C# / Joe Hocking
6. YouTube канал з гайдом [Електроний ресурс] – Режим доступу до ресурсу: https://www.youtube.com/@AbdelmoumeneUnity.
7. YouTube канал Brackeys [Електроний ресурс] – Режим доступу до ресурсу: https://www.youtube.com/@Brackeys

Додаток А

Скрипти персонажа

Основний скрипт:

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

public partial class @PlayerControls: IInputActionCollection2, IDisposable
{
public InputActionAsset asset { get; }
public @PlayerControls()
{
asset = InputActionAsset.FromJson(@"{
""name"": ""Player Controls"",
""maps"": [
{
""name"": ""Movement"",
""id"": ""03b13461-fc52-4aae-a953-31ebb16269bd"",
""actions"": [
{
""name"": ""Move"",
""type"": ""PassThrough"",
""id"": ""4188624a-1332-4844-9cd1-ca553dfd590e"",
""expectedControlType"": ""Vector2"",
""processors"": """",
""interactions"": """",
""initialStateCheck"": false
```

}
],
""bindings"": [
{
""name"": ""2D Vector"",
""id"": ""549480a3-6195-416c-9d0c-0ff1859609e2"",
""path"": ""2DVector"",
""interactions"": """",
""processors"": """",
""groups"": """",
""action"": ""Move"",
""isComposite"": true,
""isPartOfComposite"": false
},
{
""name"": ""up"",
""id"": ""015a2d7f-19d3-4af9-90bf-4083b9a67c14"",
""path"": ""<Keyboard>/w"",
""interactions"": """",
""processors"": """",
""groups"": """",
""action"": ""Move"",
""isComposite"": false,
""isPartOfComposite"": true
},
{
""name"": ""down"",
""id"": ""8fd9c14f-380a-4cbf-9302-9ce2f754555e"",

""path"": ""<Keyboard>/s"",

""interactions"": """",

""processors"": """",

""groups"": """",

""action"": ""Move"",

""isComposite"": false,

""isPartOfComposite"": true

},

{

""name"": ""left"",

""id"": ""4db10dd8-4d4b-411c-8514-91455db6244e"",

""path"": ""<Keyboard>/a"",

""interactions"": """",

""processors"": """",

""groups"": """",

""action"": ""Move"",

""isComposite"": false,

""isPartOfComposite"": true

},

{

""name"": ""right"",

""id"": ""f5b73001-4a90-47b0-a161-7ba00cab50d1"",

""path"": ""<Keyboard>/d"",

""interactions"": """",

""processors"": """",

""groups"": """",

""action"": ""Move"",

""isComposite"": false,

""isPartOfComposite"": true
}
]
},
{
""name"": ""Combat"",
""id"": ""6f857a41-2ea8-4c63-889e-8505127b2d6a"",
""actions"": [
{
""name"": ""Attack"",
""type"": ""Button"",
""id"": ""37f709e8-1846-4d8b-9fc9-34760de1f766"",
""expectedControlType"": ""Button"",
""processors"": """",
""interactions"": """",
""initialStateCheck"": false
},
{
""name"": ""Dash"",
""type"": ""Button"",
""id"": ""4c34d031-b49c-4489-a4af-ba47dcba307b"",
""expectedControlType"": ""Button"",
""processors"": """",
""interactions"": """",

arogantní pavouk, [20.05.2024 19:04]

""initialStateCheck"": false
}

],
""bindings"": [
{
""name"": """",
""id"": ""9769d740-b48f-4eb5-bc65-0d85c0cded02"",
""path"": ""<Mouse>/leftButton"",
""interactions"": """",
""processors"": """",
""groups"": """",
""action"": ""Attack"",
""isComposite"": false,
""isPartOfComposite"": false
},
{
""name"": """",
""id"": ""2090d226-78b6-4c6f-86e5-7d0efe781205"",
""path"": ""<Keyboard>/space"",
""interactions"": """",
""processors"": """",
""groups"": """",
""action"": ""Dash"",
""isComposite"": false,
""isPartOfComposite"": false
}
]
},
{
""name"": ""Inventory"",

""id"": ""6166874e-b22c-4ed9-ba26-505cb38a8a3c"",
""actions"": [
{
""name"": ""Keyboard"",
""type"": ""Value"",
""id"": ""ff9b92a7-db35-4060-adad-938cb46de9e6"",
""expectedControlType"": """",
""processors"": """",
""interactions"": """",
""initialStateCheck"": true
}
],
""bindings"": [
{
""name"": """",
""id"": ""79670d46-7d7f-467a-bc3b-b5e965788f12"",
""path"": ""<Keyboard>/1"",
""interactions"": """",
""processors"": ""Scale"",
""groups"": """",
""action"": ""Keyboard"",
""isComposite"": false,
""isPartOfComposite"": false
},
{
""name"": """",
""id"": ""1c5b0249-1b49-4f55-8eeb-3a7fb53092cd"",
""path"": ""<Keyboard>/2"",

""interactions"": """",

""processors"": ""Scale(factor=2)"",

""groups"": """",

""action"": ""Keyboard"",

""isComposite"": false,

""isPartOfComposite"": false

},

{

""name"": """",

""id"": ""e00cc44d-a70b-4f42-a2a1-e352faa2e0bd"",

""path"": ""<Keyboard>/3"",

""interactions"": """",

""processors"": ""Scale(factor=3)"",

""groups"": """",

""action"": ""Keyboard"",

""isComposite"": false,

""isPartOfComposite"": false

},

{

""name"": """",

""id"": ""86d47e26-5ed2-42b0-a521-983d232d13e8"",

""path"": ""<Keyboard>/4"",

""interactions"": """",

""processors"": ""Scale(factor=4)"",

""groups"": """",

""action"": ""Keyboard"",

""isComposite"": false,

""isPartOfComposite"": false

```
},
{
""name"": """",
""id"": ""ba631437-c6ed-430d-ae20-e7d2e6ee8f52"",
""path"": ""<Keyboard>/5"",
""interactions"": """",
""processors"": ""Scale(factor=5)"",
""groups"": """",
""action"": ""Keyboard"",
""isComposite"": false,
""isPartOfComposite"": false
}
]
}
],
""controlSchemes"": []
}");
// Movement
m_Movement = asset.FindActionMap("Movement", throwIfNotFound: true);

arogantní pavouk, [20.05.2024 19:04]
m_Movement_Move = m_Movement.FindAction("Move", throwIfNotFound:
true);
// Combat
m_Combat = asset.FindActionMap("Combat", throwIfNotFound: true);
m_Combat_Attack = m_Combat.FindAction("Attack", throwIfNotFound:
true);
m_Combat_Dash = m_Combat.FindAction("Dash", throwIfNotFound: true);
// Inventory
```

```
m_Inventory = asset.FindActionMap("Inventory", throwIfNotFound: true);
m_Inventory_Keyboard = m_Inventory.FindAction("Keyboard", throwIfNotFound: true);
}

public void Dispose()
{
UnityEngine.Object.Destroy(asset);
}

public InputBinding? bindingMask
{
get => asset.bindingMask;
set => asset.bindingMask = value;
}

public ReadOnlyArray<InputDevice>? devices
{
get => asset.devices;
set => asset.devices = value;
}

public ReadOnlyArray<InputControlScheme> controlSchemes => asset.controlSchemes;

public bool Contains(InputAction action)
{
return asset.Contains(action);
}
```

```
public IEnumerator<InputAction> GetEnumerator()
{
return asset.GetEnumerator();
}

IEnumerator IEnumerable.GetEnumerator()
{
return GetEnumerator();
}

public void Enable()
{
asset.Enable();
}

public void Disable()
{
asset.Disable();
}

public IEnumerable<InputBinding> bindings => asset.bindings;

public InputAction FindAction(string actionNameOrId, bool throwIfNotFound =
false)
{
return asset.FindAction(actionNameOrId, throwIfNotFound);
}
```

```
public int FindBinding(InputBinding bindingMask, out InputAction action)
{
return asset.FindBinding(bindingMask, out action);
}

// Movement
private readonly InputActionMap m_Movement;
private List<IMovementActions> m_MovementActionsCallbackInterfaces =
new List<IMovementActions>();
private readonly InputAction m_Movement_Move;
public struct MovementActions
{
private @PlayerControls m_Wrapper;
public MovementActions(@PlayerControls wrapper) { m_Wrapper =
wrapper; }
public InputAction @Move => m_Wrapper.m_Movement_Move;
public InputActionMap Get() { return m_Wrapper.m_Movement; }
public void Enable() { Get().Enable(); }
public void Disable() { Get().Disable(); }
public bool enabled => Get().enabled;
public static implicit operator InputActionMap(MovementActions set) {
return set.Get(); }
public void AddCallbacks(IMovementActions instance)
{

arogantní pavouk, [20.05.2024 19:04]
if (instance == null
m_Wrapper.m_MovementActionsCallbackInterfaces.Contains(instance)) return;
m_Wrapper.m_MovementActionsCallbackInterfaces.Add(instance);
@Move.started += instance.OnMove;
```

```
@Move.performed += instance.OnMove;
@Move.canceled += instance.OnMove;
}

private void UnregisterCallbacks(IMovementActions instance)
{
@Move.started -= instance.OnMove;
@Move.performed -= instance.OnMove;
@Move.canceled -= instance.OnMove;
}

public void RemoveCallbacks(IMovementActions instance)
{
if (m_Wrapper.m_MovementActionsCallbackInterfaces.Remove(instance))
UnregisterCallbacks(instance);
}

public void SetCallbacks(IMovementActions instance)
{
foreach (var item in m_Wrapper.m_MovementActionsCallbackInterfaces)
UnregisterCallbacks(item);
m_Wrapper.m_MovementActionsCallbackInterfaces.Clear();
AddCallbacks(instance);
}
}
public MovementActions @Movement => new MovementActions(this);

// Combat
```

```
private readonly InputActionMap m_Combat;
private List<ICombatActions> m_CombatActionsCallbackInterfaces = new
List<ICombatActions>();
private readonly InputAction m_Combat_Attack;
private readonly InputAction m_Combat_Dash;
public struct CombatActions
{
private @PlayerControls m_Wrapper;
public CombatActions(@PlayerControls wrapper) { m_Wrapper = wrapper; }
public InputAction @Attack => m_Wrapper.m_Combat_Attack;
public InputAction @Dash => m_Wrapper.m_Combat_Dash;
public InputActionMap Get() { return m_Wrapper.m_Combat; }
public void Enable() { Get().Enable(); }
public void Disable() { Get().Disable(); }
public bool enabled => Get().enabled;
public static implicit operator InputActionMap(CombatActions set) { return
set.Get(); }
public void AddCallbacks(ICombatActions instance)
{
if (instance == null
m_Wrapper.m_CombatActionsCallbackInterfaces.Contains(instance)) return;
m_Wrapper.m_CombatActionsCallbackInterfaces.Add(instance);
@Attack.started += instance.OnAttack;
@Attack.performed += instance.OnAttack;
@Attack.canceled += instance.OnAttack;
@Dash.started += instance.OnDash;
@Dash.performed += instance.OnDash;
@Dash.canceled += instance.OnDash;
}
```

```
private void UnregisterCallbacks(ICombatActions instance)
{
@Attack.started -= instance.OnAttack;
@Attack.performed -= instance.OnAttack;
@Attack.canceled -= instance.OnAttack;
@Dash.started -= instance.OnDash;
@Dash.performed -= instance.OnDash;
@Dash.canceled -= instance.OnDash;
}

public void RemoveCallbacks(ICombatActions instance)
{
if (m_Wrapper.m_CombatActionsCallbackInterfaces.Remove(instance))
UnregisterCallbacks(instance);
}

public void SetCallbacks(ICombatActions instance)
{
foreach (var item in m_Wrapper.m_CombatActionsCallbackInterfaces)
UnregisterCallbacks(item);
m_Wrapper.m_CombatActionsCallbackInterfaces.Clear();
AddCallbacks(instance);
}
}
public CombatActions @Combat => new CombatActions(this);
```

arogantní pavouk, [20.05.2024 19:04]

```
// Inventory
private readonly InputActionMap m_Inventory;
private List<IInventoryActions> m_InventoryActionsCallbackInterfaces = new
List<IInventoryActions>();
private readonly InputAction m_Inventory_Keyboard;
public struct InventoryActions
{
private @PlayerControls m_Wrapper;
public InventoryActions(@PlayerControls wrapper) { m_Wrapper = wrapper;
}
public InputAction @Keyboard => m_Wrapper.m_Inventory_Keyboard;
public InputActionMap Get() { return m_Wrapper.m_Inventory; }
public void Enable() { Get().Enable(); }
public void Disable() { Get().Disable(); }
public bool enabled => Get().enabled;
public static implicit operator InputActionMap(InventoryActions set) { return
set.Get(); }
public void AddCallbacks(IInventoryActions instance)
{
if (instance == null ||
m_Wrapper.m_InventoryActionsCallbackInterfaces.Contains(instance)) return;
m_Wrapper.m_InventoryActionsCallbackInterfaces.Add(instance);
@Keyboard.started += instance.OnKeyboard;
@Keyboard.performed += instance.OnKeyboard;
@Keyboard.canceled += instance.OnKeyboard;
}

private void UnregisterCallbacks(IInventoryActions instance)
{
@Keyboard.started -= instance.OnKeyboard;
```

```
@Keyboard.performed -= instance.OnKeyboard;
@Keyboard.canceled -= instance.OnKeyboard;
}

public void RemoveCallbacks(IInventoryActions instance)
{
if (m_Wrapper.m_InventoryActionsCallbackInterfaces.Remove(instance))
UnregisterCallbacks(instance);
}

public void SetCallbacks(IInventoryActions instance)
{
foreach (var item in m_Wrapper.m_InventoryActionsCallbackInterfaces)
UnregisterCallbacks(item);
m_Wrapper.m_InventoryActionsCallbackInterfaces.Clear();
AddCallbacks(instance);
}
}
public InventoryActions @Inventory => new InventoryActions(this);
public interface IMovementActions
{
void OnMove(InputAction.CallbackContext context);
}
public interface ICombatActions
{
void OnAttack(InputAction.CallbackContext context);
void OnDash(InputAction.CallbackContext context);
}
```

```
public interface IInventoryActions
{
void OnKeyboard(InputAction.CallbackContext context);
}
}
```

Скрипт здоров'я:

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

public class PlayerHealth : Singleton<PlayerHealth>
{
public bool isDead { get; private set; }

[SerializeField] private int maxHealth = 3;
[SerializeField] private float knockBackThrustAmount = 10f;
[SerializeField] private float damageRecoveryTime = 1f;

private Slider healthSlider;
private int currentHealth;
private bool canTakeDamage = true;
private Knockback knockback;
private Flash flash;

const string HEALTH_SLIDER_TEXT = "Health Slider";
const string TOWN_TEXT = "Scene1";
```

```
readonly int DEATH_HASH = Animator.StringToHash("Death");

protected override void Awake() {
base.Awake();

flash = GetComponent<Flash>();
knockback = GetComponent<Knockback>();
}

private void Start() {
isDead = false;
currentHealth = maxHealth;

UpdateHealthSlider();
}

private void OnCollisionStay2D(Collision2D other) {
EnemyAI enemy = other.gameObject.GetComponent<EnemyAI>();

if (enemy) {
TakeDamage(1, other.transform);
}
}

public void HealPlayer() {
if (currentHealth < maxHealth) {
currentHealth += 1;
UpdateHealthSlider();
```

```
}
}

public void TakeDamage(int damageAmount, Transform hitTransform) {
if (!canTakeDamage) { return; }

ScreenShakeManager.Instance.ShakeScreen();
knockback.GetKnockedBack(hitTransform, knockBackThrustAmount);
StartCoroutine(flash.FlashRoutine());
canTakeDamage = false;
currentHealth -= damageAmount;
StartCoroutine(DamageRecoveryRoutine());
UpdateHealthSlider();
CheckIfPlayerDeath();
}

private void CheckIfPlayerDeath() {
if (currentHealth <= 0 && !isDead) {
isDead = true;
Destroy(ActiveWeapon.Instance.gameObject);
currentHealth = 0;
GetComponent<Animator>().SetTrigger(DEATH_HASH);
StartCoroutine(DeathLoadSceneRoutine());
}
}

private IEnumerator DeathLoadSceneRoutine() {
yield return new WaitForSeconds(2f);
```

```
Destroy(gameObject);
SceneManager.LoadScene(TOWN_TEXT);
}

private IEnumerator DamageRecoveryRoutine() {
yield return new WaitForSeconds(damageRecoveryTime);
canTakeDamage = true;
}

private void UpdateHealthSlider() {
if (healthSlider == null) {
healthSlider =
GameObject.Find(HEALTH_SLIDER_TEXT).GetComponent<Slider>();
}

healthSlider.maxValue = maxHealth;
healthSlider.value = currentHealth;
}
}
```

Скрипт нанесення пошкодження:

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

public class DamageSource : MonoBehaviour
{
private int damageAmount;
```

```
private void Start() {
MonoBehaviour currenActiveWeapon =
ActiveWeapon.Instance.CurrentActiveWeapon;
damageAmount = (currenActiveWeapon as
IWeapon).GetWeaponInfo().weaponDamage;
}

private void OnTriggerEnter2D(Collider2D other) {
EnemyHealth enemyHealth =
other.gameObject.GetComponent<EnemyHealth>();
enemyHealth?.TakeDamage(damageAmount);
}
}
```

Скрипт атаки:

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

public class ActiveWeapon : Singleton<ActiveWeapon>
{
public MonoBehaviour CurrentActiveWeapon { get; private set; }

private PlayerControls playerControls;
private float timeBetweenAttacks;

private bool attackButtonDown, isAttacking = false;

protected override void Awake() {
base.Awake();
```

```
playerControls = new PlayerControls();
}

private void OnEnable()
{
playerControls.Enable();
}

private void Start()
{
playerControls.Combat.Attack.started += _ => StartAttacking();
playerControls.Combat.Attack.canceled += _ => StopAttacking();

AttackCooldown();
}

private void Update() {
Attack();
}

public void NewWeapon(MonoBehaviour newWeapon) {
CurrentActiveWeapon = newWeapon;

AttackCooldown();
timeBetweenAttacks = (CurrentActiveWeapon as
IWeapon).GetWeaponInfo().weaponCooldown;
}
```

```
public void WeaponNull() {
CurrentActiveWeapon = null;
}

private void AttackCooldown() {
isAttacking = true;
StopAllCoroutines();
StartCoroutine(TimeBetweenAttacksRoutine());
}

private IEnumerator TimeBetweenAttacksRoutine() {
yield return new WaitForSeconds(timeBetweenAttacks);
isAttacking = false;
}

private void StartAttacking()
{
attackButtonDown = true;
}

private void StopAttacking()
{
attackButtonDown = false;
}

private void Attack() {
if (attackButtonDown && !isAttacking && CurrentActiveWeapon) {
AttackCooldown();
```

```
(CurrentActiveWeapon as IWeapon).Attack();
}
}
}
```

Додаток Б

Скрипти ворогів

ШІ ворога:

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

public class EnemyAI : MonoBehaviour
{
[SerializeField] private float roamChangeDirFloat = 2f;
[SerializeField] private float attackRange = 0f;
[SerializeField] private MonoBehaviour enemyType;
[SerializeField] private float attackCooldown = 2f;
[SerializeField] private bool stopMovingWhileAttacking = false;

private bool canAttack = true;

private enum State {
Roaming,
Attacking
}

private Vector2 roamPosition;
private float timeRoaming = 0f;

private State state;
private EnemyPathfinding enemyPathfinding;
```

```
private void Awake() {
enemyPathfinding = GetComponent<EnemyPathfinding>();
state = State.Roaming;
}

private void Start() {
roamPosition = GetRoamingPosition();
}

private void Update() {
MovementStateControl();
}

private void MovementStateControl() {
switch (state)
{
default:
case State.Roaming:
Roaming();
break;

case State.Attacking:
Attacking();
break;
}
}

private void Roaming() {
```

```
timeRoaming += Time.deltaTime;

enemyPathfinding.MoveTo(roamPosition);

if (Vector2.Distance(transform.position,
PlayerController.Instance.transform.position) < attackRange) {
state = State.Attacking;
}

if (timeRoaming > roamChangeDirFloat) {
roamPosition = GetRoamingPosition();
}
}

private void Attacking() {
if (Vector2.Distance(transform.position,
PlayerController.Instance.transform.position) > attackRange)
{
state = State.Roaming;
}

if (attackRange != 0 && canAttack) {

canAttack = false;
(enemyType as IEnemy).Attack();

if (stopMovingWhileAttacking) {
enemyPathfinding.StopMoving();
} else {
```

```
enemyPathfinding.MoveTo(roamPosition);
}

StartCoroutine(AttackCooldownRoutine());
}
}

private IEnumerator AttackCooldownRoutine() {
yield return new WaitForSeconds(attackCooldown);
canAttack = true;
}

private Vector2 GetRoamingPosition() {
timeRoaming = 0f;
return new Vector2(Random.Range(-1f, 1f), Random.Range(-1f,
1f)).normalized;
}
}
```

Скрипт життя:

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

public class EnemyHealth : MonoBehaviour
{
[SerializeField] private int startingHealth = 3;
[SerializeField] private GameObject deathVFXPrefab;
[SerializeField] private float knockBackThrust = 15f;
```

```
private int currentHealth;
private Knockback knockback;
private Flash flash;

private void Awake() {
flash = GetComponent<Flash>();
knockback = GetComponent<Knockback>();
}

private void Start() {
currentHealth = startingHealth;
}

public void TakeDamage(int damage) {
currentHealth -= damage;
knockback.GetKnockedBack(PlayerController.Instance.transform,
knockBackThrust);
StartCoroutine(flash.FlashRoutine());
StartCoroutine(CheckDetectDeathRoutine());
}

private IEnumerator CheckDetectDeathRoutine() {
yield return new WaitForSeconds(flash.GetRestoreMatTime());
DetectDeath();
}

public void DetectDeath() {
if (currentHealth <= 0) {
```

```
Instantiate(deathVFXPrefab, transform.position, Quaternion.identity);
GetComponent<PickUpSpawner>().DropItems();
Destroy(gameObject);
}
}
}
```

Скрипт дистанційної атаки:

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

public class EnemyPathfinding : MonoBehaviour
{
[SerializeField] private float moveSpeed = 2f;

private Rigidbody2D rb;
private Vector2 moveDir;
private Knockback knockback;
private SpriteRenderer spriteRenderer;

private void Awake() {
spriteRenderer = GetComponent<SpriteRenderer>();
knockback = GetComponent<Knockback>();
rb = GetComponent<Rigidbody2D>();
}

private void FixedUpdate() {
if (knockback.GettingKnockedBack) { return; }
```

```
rb.MovePosition(rb.position + moveDir * (moveSpeed *
Time.fixedDeltaTime));

if (moveDir.x < 0) {
spriteRenderer.flipX = true;
} else if (moveDir.x > 0) {
spriteRenderer.flipX = false;
}
}

public void MoveTo(Vector2 targetPosition) {
moveDir = targetPosition;
}

public void StopMoving() {
moveDir = Vector3.zero;
}
}
```

Додаток В

Скрипт зміни сцен

Скрипт початку сцени:

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

public class AreaEntrance : MonoBehaviour
{
[SerializeField] private string transitionName;

private void Start() {
if (transitionName == SceneManagement.Instance.SceneTransitionName) {
PlayerController.Instance.transform.position = this.transform.position;
CameraController.Instance.SetPlayerCameraFollow();
UIFade.Instance.FadeToClear();
}
}
}
```

Скрипт переходу на іншу сцену:

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

public class AreaExit : MonoBehaviour
{
[SerializeField] private string sceneToLoad;
```

```
[SerializeField] private string sceneTransitionName;

private float waitToLoadTime = 1f;

private void OnTriggerEnter2D(Collider2D other) {
if (other.gameObject.GetComponent<PlayerController>()) {
SceneManagement.Instance.SetTransitionName(sceneTransitionName);
UIFade.Instance.FadeToBlack();
StartCoroutine(LoadSceneRoutine());
}
}

private IEnumerator LoadSceneRoutine() {
while (waitToLoadTime >= 0)
{
waitToLoadTime -= Time.deltaTime;
yield return null;
}

SceneManager.LoadScene(sceneToLoad);
}
}
```

Скрипт контролю камери:

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

```
public class CameraController : Singleton<CameraController>
{
private CinemachineVirtualCamera cinemachineVirtualCamera;

private void Start() {
SetPlayerCameraFollow();
}

public void SetPlayerCameraFollow() {
cinemachineVirtualCamera =
FindObjectOfType<CinemachineVirtualCamera>();
cinemachineVirtualCamera.Follow = PlayerController.Instance.transform;
}
}
```