Міністерство освіти і науки України
Кам’янець-Подільський національний університет імені Івана Огієнка
Фізико-математичний факультет
Кафедра комп’ютерних наук

Дипломна робота бакалавра
з теми: **«Дослідження можливостей рушіїв Unity та Blender для створення ігрових застосунків»**

Виконав: здобувач вищої освіти групи KN1-B20 спеціальності 122 Комп’ютерні науки
Цинчик Дмитро Олегович

Керівник: Пилипюк Тетяна Михайлівна, кандидат фізико-математичних наук, доцент, доцент кафедри комп'ютерних наук

Рецензент: Громик Андрій Петрович, кандидат технічних наук, доцент, доцент кафедри інформаційних технологій, фізико-математичних та безпекових дисциплін Закладу вищої освіти «Подільський державний університет»

м. Кам’янець-Подільський – 2024 р.

АНОТАЦІЯ

У даній роботі проаналізовано можливості та інструментарій двох потужних програмних середовищ – Unity та Blender – для створення ігрових застосунків. Основні напрямки дослідження – моделювання та анімація в Blender, а також інтеграція створеного контенту в Unity для подальшої розробки гри. Також успішно реалізовано прототип 3D гри, який включає такі компоненти як: анімовані та текстуровані моделі; контролери анімацій; систему введення користувацьких команд; інтерактивні елементи.

Результати дослідження свідчать, що комбінація можливостей Blender і Unity є ефективним підходом для створення 3D ігрових застосунків. Рекомендується поглиблене вивчення можливостей обох інструментів, розширення прототипу додаванням нових ігрових механік, а також оптимізація продуктивності гри для різних платформ.

Ключові слова: Unity, Blender, 3D моделювання, анімація, ігровий рушій, інтерактивні елементи, прототип гри.

ABSTRACT

The possibilities and tools of two powerful software for creating game applications – Unity and Blender – are analyzed in this work. The main areas of study are modeling and animation in Blender and integration of created content in Unity for further game development. A 3D game prototype was also successfully implemented, which includes such components as: animated and textured models; animation controllers; user command entry system; interactive elements.

The research results show that the combination of Blender and Unity capabilities is an effective approach for creating 3D game applications. An in-depth study of the capabilities of both tools, expansion of the prototype by adding new game mechanics, and optimization of game performance for different platforms are recommended.

Keywords: Unity, Blender, 3D modeling, animation, game engine, interactive elements, game prototype.

ЗМІСТ

ВСТУП

Актуальність теми. У сучасному світі ігрова індустрія є однією з галузей інформаційних технологій, що динамічно розвивається. Вона не лише впливає на розваги та дозвілля, але також знаходить застосування в освіті, тренуваннях та навіть в медичних програмах. Одним з ключових аспектів створення ігор є вибір відповідного програмного забезпечення, яке дозволяє реалізувати концепції розробників, забезпечуючи високу якість графіки, анімації та загальну функціональність ігрових застосунків.

У даній роботі особливу увагу приділено дослідженню можливостей двох популярних інструментів для створення 3D-ігор: Unity та Blender. Unity є одним з найпоширеніших ігрових рушіїв, відомий своєю гнучкістю та можливістю кросплатформенної розробки. Blender, у свою чергу, надає потужні інструменти для 3D-моделювання, анімації та рендерингу, що робить його невід'ємною частиною процесу створення контенту для ігор.

Дослідження можливостей Unity та Blender для створення ігрових застосунків є актуальним завданням, оскільки воно дозволяє розкрити потенціал цих інструментів та розробити ефективні методи їх використання у різних етапах розробки ігрових проектів.

Об’єкт та предмет дослідження. Об’єктом дослідження є процес розробки 3D ігрових застосунків із використанням Unity та Blender. Предметом дослідження є методи і технології моделювання, анімації, текстурування у Blender та інтеграціїя створеного контенту в Unity для створення повнофункціональних ігрових застосунків.

Мета дослідження: аналіз потенціалу та можливостей рушіїв Unity та Blender у контексті створення ігрових застосунків: оцінка, оптимізація та практичне використання.

Завдання дослідження:

- дослідити інтегроване середовище UNITY розробки програмного забезпечення для створення ігрових застосунків;

- визначити ключові характеристики та можливості програмного забезпечення BLENDER;
- здійснити практичну реалізацію застосування стеку технологій UNITY/BLENDER, реалізацію прототипу 3D застосунку.

Методи дослідження: аналіз літературних джерел та документації щодо функціональних можливостей Unity та Blender; емпіричні методи, включаючи експериментальне моделювання та анімацію у Blender, а також інтеграцію ігрових елементів у Unity; порівняльний аналіз для оцінки переваг та недоліків використання зазначених інструментів.

Апробація результатів дослідження. Результати досліджень були оприлюднені на науковій конференції здобувачів вищої освіти за підсумками НДР у 2023-2024 навчальному році 9-10 квітня 2024 року, м. Кам’янець-Подільський [5].

Практичне значення. Результати дослідження можуть бути використані розробниками ігор для оптимізації процесу створення 3D ігрових застосунків. Запропоновані методи інтеграції контенту, створеного у Blender, у Unity, дозволяють значно підвищити ефективність розробки та якість кінцевого продукту. Результати також можуть бути застосовані у навчальних програмах з розробки ігор та 3D моделювання.

Структура роботи. Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків.

РОЗДІЛ 1. ІНТЕГРОВАНЕ СЕРЕДОВИЩЕ UNITY РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ СТВОРЕННЯ ІГРОВИХ ЗАСТОСУНКІВ

Unity – це інтегроване середовище розробки програмного забезпечення, що використовується для створення і розгортання ігрових застосунків та інтерактивних віртуальних середовищ. Розроблене Unity Technologies, це програмне забезпечення знаходить широке застосування у сфері розваг, навчання, архітектурного дизайну, візуалізації даних, медицини та інших галузях.

Це один з найпопулярніших та потужних інструментів для розробки ігор, віртуальної реальності, розширеної реальності та інших інтерактивних застосунків. Unity надає розробникам доступ до різноманітних функціональних можливостей, що допомагають в створенні вражаючих та динамічних ефектів:

- Графічний редактор: Unity надає зручний графічний інтерфейс для створення та редагування сцен, об'єктів, анімацій, матеріалів та інших компонентів гри.
- Скриптування: дозволяє розробникам програмного забезпечення додавати функціональність до своїх проектів за допомогою скриптів на мовах програмування, таких як C# або JavaScript.
- Фізичне моделювання: Unity має вбудовану систему фізичного моделювання, яка дозволяє створювати реалістичну поведінку об'єктів у віртуальному просторі, включаючи гравітацію, зіткнення та інші фізичні властивості.
- Архітектура гри: Unity надає структуровану архітектуру гри, яка дозволяє легко управляти об'єктами та компонентами, організувати логіку гри та взаємодіяти з різними системами.
- Розгортання на різних платформах: Unity дозволяє розгортати створені проекти на різних платформах, таких як Windows, MacOS, Android, iOS та

інші, що робить його універсальним інструментом для розробки крос-платформових застосунків.

Переваги та недоліки ігрового рушія Unity подамо у таблиці 1.1.

Таблиця 1.1

Характеристика ігрового рушія Unity [5]

Переваги Unity	*Недоліки Unity*
Кросплатформеність: Unity дозволяє розробляти ігри для різних платформ, використовуючи одну кодову базу. Велика спільнота: Unity має велику активну спільноту розробників, яка надає підтримку, ресурси та допомогу. Візуальний редактор: Інтуїтивно зрозумілий візуальний редактор дозволяє швидко створювати ігрові об'єкти, налаштовувати їх параметри та взаємодіяти з ними. Asset Store: Unity має велику бібліотеку готових ресурсів, які можна використовувати в проектах, що значно полегшує процес розробки.	Вартість: Повна версія Unity може бути дорога, особливо для незалежних розробників або малих студій. Продуктивність: Для великих проєктів чи ігор з високим рівнем деталізації можуть виникати проблеми з продуктивністю, особливо на мобільних пристроях.

1.1. Фізичне моделювання та Class Physics в Unity

Фізика в Unity – це система, яка дозволяє симулювати реалістичну поведінку об'єктів у віртуальному середовищі. Ця система включає в себе різні компоненти та функції для взаємодії об'єктів, їхнього руху, зіткнень та інших фізичних явищ.

Основні складові системи фізики в Unity:

- Колізії (Colliders). Колайдери – це компоненти, які визначають форму об'єкта для взаємодії з іншими об'єктами у просторі. Unity має різні типи колайдерів, такі як коробка, куля, капсула, меш (модель), які можна встановлювати на об'єкти.
- Фізичне тіло (Rigidbody). Ріджидболі – це компонент, який додає фізичні властивості до об'єкта, такі як маса, швидкість, обертання. Використовується для симуляції реалістичного руху об'єкта від дії сили тяжіння та інших сил.
- Фізичні матеріали (Physics Materials). Фізичні матеріали визначають фізичні властивості поверхні об'єкта, такі як тертя, пружність і сковзання. Вони використовуються для визначення того, як об'єкти взаємодіють між собою при зіткненні.
- Система детекції зіткнень (Collision Detection). Unity має різні методи детекції зіткнень, такі як дискретна (discrete) та неперервна (continuous). Дискретна детекція використовується для статичних або повільних об'єктів, а неперервна – для швидких об'єктів або об'єктів з великими швидкостями.
- Фізичні сили (Physics Forces). В Unity є можливість застосовувати різні фізичні сили, такі як тяжіння, сила тиску, сила тертя і т.д., для моделювання різних фізичних явищ.

Клас Physics в Unity – це статичний клас, який містить різноманітні методи і властивості, пов'язані з фізичною системою гри. Цей клас дозволяє взаємодіяти з фізичними об'єктами, отримувати інформацію про зіткнення, застосовувати фізичні сили та багато іншого.

- Raycasting. Методи, такі як Physics.Raycast та Physics.RaycastAll, використовуються для визначення, чи перетинає промінь певний колайдер. Це часто використовується для виявлення об'єктів у зазначеному напрямку.

- Overlap. Методи, такі як Physics.OverlapSphere і Physics.OverlapBox, дозволяють знаходити всі колайдери, які перетинають або знаходяться всередині певної області в просторі.
- Collision Detection Mode. За допомогою властивості Physics.defaultCollisionDetectionMode можна встановлювати режим детектування зіткнень за замовчуванням для всіх об'єктів у сцені. Це дозволяє керувати тим, як Unity виявляє зіткнення між об'єктами.
- Layer Masks. Physics.LayerMask дозволяє визначити, з якими шарами фізики взаємодіють об'єкти. Це дозволяє вам налаштувати, які об'єкти взаємодіють між собою у фізичній системі.
- Сили і фізичні властивості. Клас Physics також містить різні методи для роботи з фізичними силами, такими як Physics.ApplyForce, Physics.ApplyTorque і т. д., що дозволяє застосовувати сили до фізичних об'єктів.

1.2. Графічний редактор Unity та його інтерфейс

Графічний редактор Unity представляє собою інтегроване середовище розробки програмного забезпечення, що дозволяє створювати візуально зручні та функціонально складні ігрові та інтерактивні застосунки. Його інтерфейс розроблений з урахуванням принципів ергономіки та логічного розташування елементів для максимальної продуктивності розробника.

Інтерфейс графічного редактора Unity (рис. 1.1) складається з кількох ключових елементів, які забезпечують управління різними аспектами розробки.

- Панель ієрархії (Hierarchy Panel). Ця панель відображає ієрархію всіх об'єктів у поточній сцені. Розробники можуть організовувати об'єкти у відповідний спосіб та керувати їхнім розташуванням та структурою.
- Вікно інспектора (Inspector Windows). Це вікно відображає властивості та компоненти вибраного об'єкта. Розробники можуть змінювати параметри

об'єкта, додавати нові компоненти та взаємодіяти з ними безпосередньо через це вікно.

– Вікна сцени (Scene Windows). Вікна сцени надають можливість перегляду та редагування об'єктів у поточній сцені. Розробники можуть переміщати, обертати та масштабувати об'єкти, встановлювати їх позиції та взаємодіяти з ними візуально.
– Панель проектів (Project Panel). Ця панель відображає всі файли та ресурси, які використовуються в поточному проекті Unity. Розробники можуть керувати ресурсами, організовувати їх у папки та імпортувати нові файли.
– Панель ассортименту (Asset Store Panel). Ця панель дозволяє розробникам переглядати та завантажувати додаткові ресурси, такі як моделі, текстури, аудіо та інші активи, з магазину Unity Asset Store. На рис. 1.2. – вікно мереджера активів.

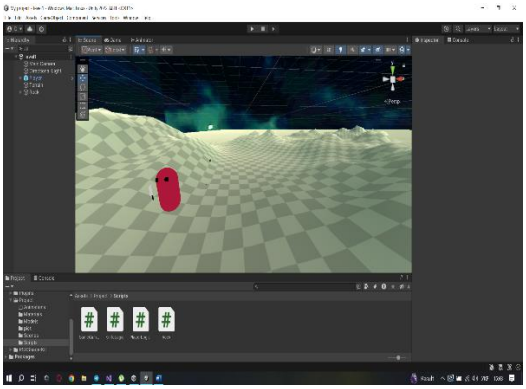

Рис. 1.1. Інтерфейс редактора

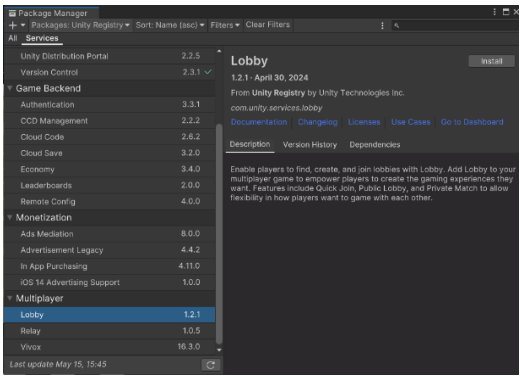

Рис. 1.2. Менеджер активів

1.3. Методології реалізації руху об'єктів

Зважаючи на характеристики віртуальних ігрових середовищ, рух об'єктів в рамках Unity може бути реалізований за допомогою різних методологій. Ці методи можуть включати: обробку введення користувача для контролю руху за допомогою клавіатури або контролера, використання фізичних двигунів для симуляції реалістичного фізичного взаємодії з оточуючими об'єктами, використання анімацій для передачі різноманітних рухів та дій гравця, і

використання алгоритмів штучного інтелекту для управління поведінкою персонажів в залежності від оточення та мети гравця. Вибір конкретного методу реалізації руху об'єктів визначається специфічними вимогами і особливостями кожного конкретного проекту.

У контексті віртуальних ігрових середовищ, керування рухом персонажа за допомогою клавіатури або контролера є одним із основних способів взаємодії гравця з віртуальною обстановкою. Цей метод передбачає застосування алгоритмів обробки введених даних, що надходять від зовнішнього контролюючого пристрою (клавіатури або контролера), з метою визначення та виконання відповідних дій, які впливають на рухові можливості персонажа в ігровому світі.

Введені дані, що надходять від користувача через вибраний контролер, перетворюються в цифрові сигнали, які система обробляє згідно з заздалегідь визначеними правилами та логікою гри. Цей процес може включати інтерпретацію введених команд та їх перетворення на конкретні дії відповідно до поточного стану гри та параметрів персонажа. Наприклад, натискання клавіші "вперед" може призвести до збільшення швидкості пересування персонажа у відповідному напрямку, або натискання кнопки "стріляти" може викликати дію по вистрілу зі зброї персонажа.

Крім того, керування персонажем за допомогою клавіатури або контролера може бути реалізоване з урахуванням принципів ергономіки та відповідно до звичайних концепцій інтерфейсу взаємодії з користувачем, з метою забезпечення зручного та інтуїтивно зрозумілого досвіду гри.

Система введення Unity (Unity Input System) є сучасним механізмом для обробки введення користувачів у програмному середовищі Unity, що надає розробникам більш гнучкі та масштабовані інструменти для інтеграції різноманітних пристроїв введення, включаючи клавіатури, миші, геймпади та сенсорні екрани. Введена як заміна старої системи введення, нова Input System забезпечує покращену обробку подій введення, конфігурацію пристроїв і розширені можливості для налаштування введення.

Основні Компоненти такі:

- Input Action Asset. Основний компонент системи введення, що слугує контейнером для визначення дій та їх прив'язок до конкретних пристроїв введення. Він дозволяє розробникам визначати різні контексти введення, такі як ігровий процес, меню або UI.
- Action Maps. Логічні групи дій, що представляють різні стани введення або режими, в яких можуть перебувати гравці. Це дозволяє легко перемикатися між різними наборами дій, наприклад, між керуванням персонажем та керуванням меню.
- Input Actions (рис. 1.3). Окремі дії, які представляють конкретні події введення, такі як натискання клавіші, рух миші або переміщення аналогового стика. Вони можуть бути налаштовані для обробки різних типів введення, таких як цифрові, аналогові або векторні значення.
- Control Schemes. Набори пристроїв введення, що визначають, які саме пристрої використовуються для обробки дій у певному контексті. Це дозволяє легко підтримувати різноманітні конфігурації пристроїв, такі як клавіатура з мишею або геймпад.

Input Actions у системі введення Unity представляють собою абстракції подій введення, які використовуються для зв’язування фізичних дій користувача (натискання кнопок, рух миші, маніпуляції джойстиком тощо) з логікою гри. Це забезпечує більш інтуїтивний і масштабований підхід до обробки введення порівняно з традиційними методами.

Input Actions розроблені для створення розширюваної, кросплатформової системи введення, яка забезпечує гнучкість і зручність використання. Вони дозволяють абстрагувати фізичні дії введення від логіки гри, що сприяє більш чистій і модульній архітектурі коду.

На рис. 1.4 – вікно з налаштованим Input Handler.

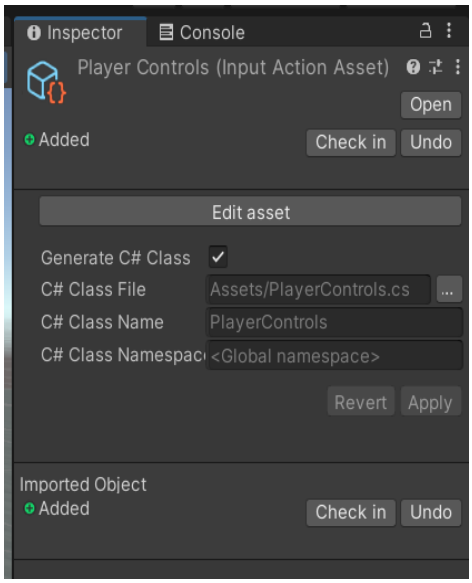

Рис. 1.3. Створений Input Action у вікні Inspector

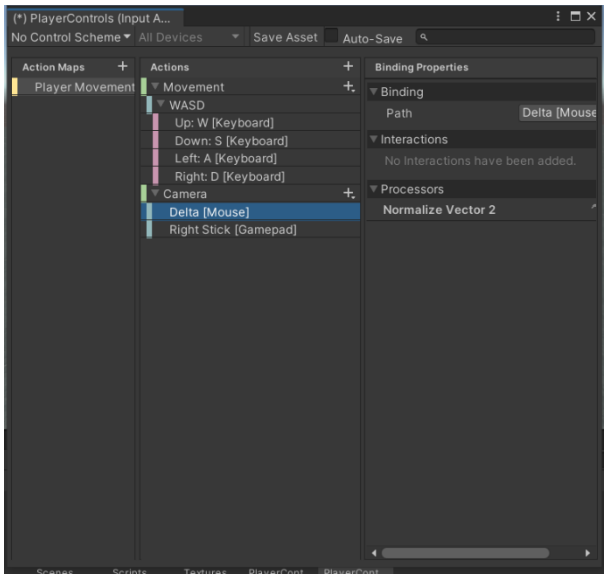

Рис. 1.4. Налаштований Input Handler

Використання скриптів є ключовим елементом для реалізації руху гравця. Скрипти, що пишуться на мові програмування C# або UnityScript (Javascript), виступають як основний інструмент для визначення поведінки об'єктів у віртуальному середовищі.

В процесі розробки, програміст зазвичай створює скрипти, які прив'язуються до об'єктів, що відповідають гравцеві у грі. Ці скрипти визначають різні аспекти руху гравця, такі як швидкість, напрямок та інші параметри, що впливають на його переміщення відносно віртуального середовища.

Скрипти руху гравця можуть включати в себе різні елементи програмної логіки, такі як обробка введення користувача з клавіатури або контролера, обчислення нової позиції гравця на основі цього введення та інших факторів, які впливають на рух віртуального персонажа.

Під час написання скриптів (рис. 1.5), розробник враховує фізичні закони віртуального світу, такі як колізії з іншими об'єктами, гравітація та інерція, що можуть впливати на рух персонажа. Використовуючи функціональні можливості Unity API, розробник взаємодіє з об'єктами гри та визначає їх поведінку відповідно до потреб проекту.

Рис. 1.5. Створені скрипти у графічному інтерфейсі

У Unity, для реалізації керування персонажем з клавіатури, існують декілька методів, які можна використовувати для обробки введення від користувача та визначення руху персонажа відповідно до натискання клавіш. Декілька основних методів включають у себе:

- Метод Input.GetKey/GetKeyDown/GetKeyUp. Ці методи використовуються для перевірки стану клавіші в поточному кадрі, визначення часу натискання та звільнення клавіші. Наприклад, метод Input.GetKeyDown(KeyCode.W) може бути використаний для визначення часу натискання клавіші "W" для переміщення персонажа вперед.
- Метод Input.GetAxis. Цей метод дозволяє отримати значення осі з введення клавіатури або контролера, які можуть бути використані для плавного керування рухом персонажа. Наприклад, Input.GetAxis("Horizontal") може повернути значення від -1 до 1 для горизонтального руху персонажа.
- Робота з фізикою. У деяких випадках, для реалістичного руху персонажа можуть використовуватися фізичні компоненти, такі як CharacterController або Rigidbody, які можна керувати з клавіатури або контролера за допомогою скриптів.

Ці методи, використовуючи програмні скрипти, обробляють введення від клавіатури та визначають, яким чином персонаж повинен реагувати на це введення, включаючи його рух, обертання, або взаємодію з іншими об'єктами віртуального середовища. Такі методи взаємодіють з іншими компонентами та системами Unity для створення плавного та реалістичного досвіду руху гравця у віртуальному світі.

Активація фізичної моделі в середовищі розробки Unity включає в себе інтеграцію компонентів фізики, що дозволяють симулювати реальні фізичні

властивості об'єктів та їх взаємодії відповідно до правил і законів механіки. Для активації фізики у Unity необхідно використовувати фізичні компоненти, що надають засоби моделювання реального руху, колізій та динамічних взаємодій об'єктів у віртуальному просторі.

Під час активації фізики в Unity, розробники можуть використовувати різноманітні компоненти, такі як Rigidbody (рис. 1.6), Collider (рис. 1.7), Joint і інші, для задання фізичних властивостей об'єктів та їх взаємодій. Наприклад, компонент Rigidbody дозволяє встановлювати масу, швидкість, силу та інші параметри об'єкту, що впливають на його рух. Компонент Collider визначає форму та область колізій об'єкту, що визначає його взаємодію з іншими об'єктами у сцені.

За допомогою скриптів програмування, розробники можуть контролювати поведінку фізичних об'єктів у реальному часі, використовуючи різні методи та функції, що надаються Unity API. Це дозволяє створювати складні фізичні симуляції, такі як симуляція гравітації, колізії, руху тіл під впливом сил і т. д.

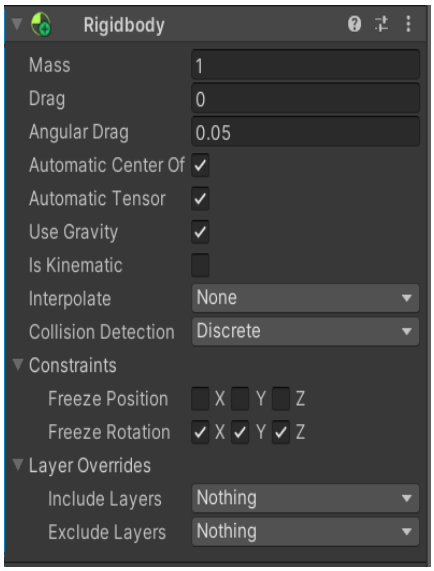

Рис. 1.6. Компонент Rigidbody, доданий до ігрового об’єкту в інспекторі, та його налаштування

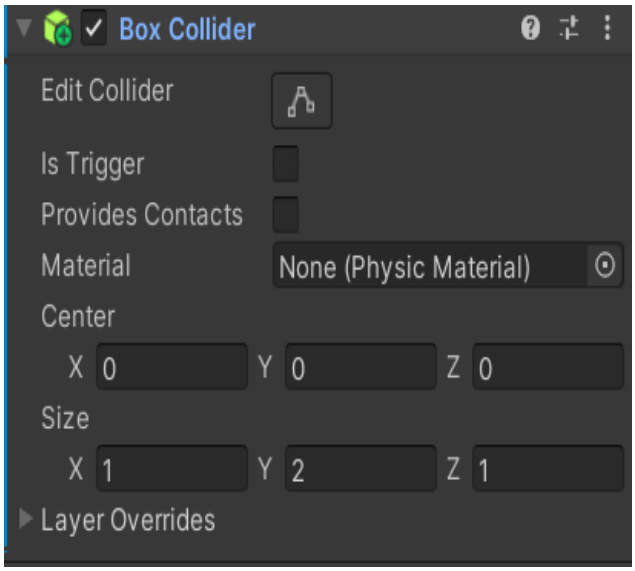

Рис. 1.7. Компонент Box Collider, доданий до ігрового об’єкту в інспекторі, та його налаштування

Для контролю руху персонажа використовується Velocity.

Velocity, визначена як векторна величина швидкості, є ключовим поняттям у фізиці, що застосовується до руху персонажу в середовищі розробки Unity. У контексті програмування та симуляції фізики, вектор швидкості, відомий як velocity, представляє собою об'єктну властивість, яка визначає швидкість та напрямок руху персонажа у тривимірному просторі.

Визначення параметрів швидкості:

- Velocity в Unity визначається як тривимірний вектор, що включає компоненти швидкості по координатам *x*, *y* та *z*.
- За замовчуванням, швидкість руху персонажа задається у метрах на секунду (m/s) або іншій відповідній масштабній одиниці.

Управління рухом за допомогою velocity:

- Швидкість руху персонажа контролюється за допомогою встановлення відповідних значень для компонентів velocity.
- Зміна швидкості може відбуватися за допомогою зміни властивостей velocity відповідно до умов або подій у грі.

Приклади використання:

- Під час руху персонажа вперед, velocity може бути налаштована на позитивні значення в компоненті зі швидкістю вздовж вісі *x*.
- Для стрибка, компонент зі швидкістю вздовж вісі *y* може бути налаштований на певне значення, щоб забезпечити рух персонажа вгору.

Щодо динамічного управління, то velocity може бути змінена в реальному часі шляхом застосування сил або імпульсів, що моделюють зовнішні чи внутрішні впливи на персонажа.

Реалізація можливості моделі гравця озиратись по сторонах у віртуальному середовищі відбувається через програмне керування обертанням тіла або голови гравця, що моделюється як частина геймплейної механіки. Ця функціональність дозволяє гравцеві досліджувати своє оточення, спостерігати за подіями та

реагувати на них, підвищуючи ступінь іммерсії та взаємодії у грі або іншому віртуальному досвіді.

Моделювання обертання гравця:

- гравець може бути представлений як модель, що складається з різних частин, таких як тіло, голова та інші елементи;
- через програмне керування обертанням цих частин можна досягти ефекту озирання гравця по сторонах.

Для зчитування введення користувача, що вказує на бажання озирнутись, використовуються методи або події вводу відповідно до використаної платформи (клавіатура, миша, джойстик тощо).

Відповідно до введення користувача, змінюється значення обертання голови чи інших частин моделі гравця. Це може бути здійснено шляхом зміни значень кутів ейлера моделі або застосування повороту до неї.

Обертання моделі оновлюється в реальному часі в залежності від введення користувача. Це забезпечує плавний та реалістичний ефект озирання по сторонах. За потреби можуть бути встановлені обмеження на обертання моделі, щоб уникнути неочікуваних результатів чи конфліктів з грою.

Для реалізації обертання моделі гравця використовують клас Quaternion та його методи.

Quaternion – це тип даних, що використовується для представлення обертань об'єктів в тривимірному просторі в середовищі розробки Unity. Цей математичний об'єкт використовується для зберігання і обчислення обертань об'єктів навколо вісей координат, що дозволяє точно та ефективно керувати їхньою орієнтацією в просторі.

Quaternion використовується для представлення обертань в тривимірному просторі. Він зберігає інформацію про кут повороту та вісь обертання.

Quaternion може бути представлений як чотиривимірний вектор, який складається з трьох компонент, що відповідають вектору обертання, та одного скалярного компоненту, який визначає кут повороту.

Quaternion також має ряд математичних властивостей, які роблять його корисним для роботи з обертаннями, такі як асоціативність, комутативність та інші.

Quaternion використовується в Unity для представлення обертань об'єктів, таких як об'єкти гри, камери та інші. Він є невід'ємною частиною API Unity і використовується в багатьох функціях та методах, пов'язаних з переміщенням та обертанням об'єктів в просторі.

Quaternion використовується для обчислення нового положення та обертання об'єктів під час їхньої трансформації в просторі.

Для реалізації обертання моделі гравця є методи класу Quaternion.

Методи Quaternion.Lerp та Quaternion.Euler є важливими засобами управління обертаннями об'єктів в тривимірному просторі у середовищі розробки Unity. Дозволяють вони здійснювати зручне та точне керування орієнтацією об'єктів у грі чи іншому віртуальному досвіді.

Метод Quaternion.Lerp використовується для лінійної інтерполяції між двома кватерніонами, що дозволяє плавно переміщувати об'єкт від одного обертання до іншого.

Лінійна інтерполяція забезпечує плавний та контрольований перехід між станами об'єкта, що покращує іммерсію гравця.

Метод Quaternion.Euler використовується для створення кватерніона на основі кутів ейлера – кутів повороту об'єкта навколо кожної з трьох осей (*X*, *Y*, *Z*). Цей метод дозволяє зручно задавати орієнтацію об'єкта у просторі за допомогою зрозумілих інтуїтивно кутів.

Використання:

- Обидва методи широко використовуються для керування обертаннями об'єктів у графічній анімації, віртуальних середовищах та іграх.
- Quaternion.Lerp дозволяє плавно анімувати обертання об'єктів з одного стану у інший.
- Quaternion.Euler дозволяє зручно задавати кут повороту об'єкта на основі його кутів ейлера.

Синтаксис:

- Синтаксис методу Quaternion.Lerp виглядає так: Quaternion.Lerp(from, to, t), де from – початковий кватерніон, to – кінцевий кватерніон, t - параметр часу інтерполяції.
- Синтаксис методу Quaternion.Euler виглядає так: Quaternion.Euler(*x*, *y*, *z*), де x, y, z – кути ейлера навколо відповідних осей.

1.4. Камера в середовищі розробки Unity

Камера в середовищі розробки Unity є важливою складовою для відтворення та візуалізації віртуального світу, яка надає можливість перегляду сцени з точки зору гравця чи іншого об'єкта. Камера відображає графічний вихід гри на екрані, а також може контролювати панораму, фокус та інші аспекти графіки та відображення.

Камера в Unity має різноманітні параметри, які визначають її поведінку та відображення. Сюди входять позиція, орієнтація, поле зору (FOV), відстань віддалення, аспектний відношення та інші.

Камера використовується для відображення графічного виходу гри або іншого віртуального досвіду на екрані.

Вона може бути розміщена в будь-якій точці сцени для створення різних ефектів камери, таких як перегляд з перспективи гравця, зображення зверху, слідування за об'єктом тощо.

Камера може рухатись у тривимірному просторі в залежності від програмної логіки або дій гравця. Це може бути здійснено через програмне керування позицією та орієнтацією камери.

Камера може бути анімована або модифікована для створення різних ефектів, таких як зум, трансформація між сценами, ефекти руху камери та багато іншого.

У грі камера може використовуватися для створення різних сценаріїв гри, включаючи визначення області видимості гравця, слідування за героєм, відтворення відео та інше.

Використання скрипту, що містить логіку слідування камери за гравцем в середовищі Unity, дозволяє створювати ефективні та реалістичні ефекти перспективи гравця. Цей скрипт керує рухом камери, так щоб камера завжди знаходилася в потрібному положенні відносно гравця, що полегшує навігацію та створює зручний іммерсивний досвід для користувача.

1. Отримання посилання на об'єкт гравця.

Спочатку скрипт отримує посилання на об'єкт гравця в сцені Unity. Це може бути здійснено через публічне поле скрипту, в яке можна присвоїти об'єкт гравця через інтерфейс користувача Unity або програмно.

2. Розрахунок нової позиції камери.

За допомогою логіки, вбудованої в скрипт, визначається нова позиція камери. Це може бути зроблено на основі поточної позиції гравця та параметрів, таких як відстань, висота, кут огляду тощо.

3. Зміна позиції та орієнтації камери.

Після розрахунку нової позиції камери, скрипт встановлює позицію та орієнтацію камери відповідно до розрахованих значень. Це може бути здійснено через доступ до компоненту Transform камери та встановлення нових значень позиції та орієнтації.

4. Оновлення камери.

Логіка слідування камери за гравцем повинна викликатися кожен кадр для забезпечення постійного оновлення позиції та орієнтації камери відносно гравця.

5. Додаткові функції.

Залежно від конкретних вимог, скрипт може містити додаткові функції, такі як обмеження області переміщення камери, згладжування руху, реалізація плавних переходів між положеннями камери тощо.

Висновок до розділу 1

Фізична система Unity дозволяє створювати реалістичні симуляції руху, зіткнень та інших фізичних явищ, що робить її потужним інструментом для розробки ігор та інтерактивних застосунків.

Використання скриптів у Unity для реалізації руху гравця є важливим елементом програмування в ігровому середовищі, що дозволяє створювати реалістичні та динамічні взаємодії між гравцем та віртуальним світом гри.

Система введення Unity представляє собою потужний та гнучкий інструмент для обробки введення користувачів, що відповідає сучасним вимогам до розробки інтерактивних застосунків. Завдяки своїй масштабованості та можливостям налаштування, вона забезпечує розробників усім необхідним для створення високоякісних ігор та застосунків з підтримкою різноманітних пристроїв введення.

РОЗДІЛ 2. КЛЮЧОВІ ХАРАКТЕРИСТИКИ ТА МОЖЛИВОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ BLENDER

2.1. Моделювання в Blender

Blender (рис. 2.1) – це вільне, відкрите програмне забезпечення для моделювання, анімації, рендерингу, композитингу та редагування відео та 3D-графіки. Він створений фондом Blender Foundation та підтримується спільнотою розробників і користувачів по всьому світу.

- Моделювання. Blender надає широкі можливості для моделювання об'єктів у тривимірному просторі. Він підтримує різні техніки моделювання, такі як полігональне, скульптурне, руководне та інші.
- Анімація. Blender дозволяє створювати складні анімації для об'єктів, персонажів та сцен. Він має інструменти для кадрування, ключової анімації, скелетної анімації та інших технік анімації.
- Рендеринг. Blender має потужний вбудований рендерер, який дозволяє створювати візуально захоплюючі зображення та анімації. Він підтримує різні движки рендерінгу, такі як Cycles та EEVEE, які надають різні можливості та ефекти.
- Композитинг та редагування відео. Blender має інструменти для композитингу та редагування відео, що дозволяє об'єднувати різні шари, ефекти та редагувати відео-контент.
- Відкритий код та спільнота. Однією з ключових переваг Blender є те, що він є вільним та відкритим програмним забезпеченням, що означає, що він доступний для використання та модифікації безкоштовно. Велика та активна спільнота розробників та користувачів надає підтримку, навчальні матеріали та різноманітні ресурси для розвитку вмінь та знань.

Моделювання в Blender – це процес створення тривимірних об'єктів та їх редагування у віртуальному середовищі. Blender надає розробникам широкий спектр інструментів для створення різних типів об'єктів – від простих

геометричних форм до складних персонажів та архітектурних споруд. Ось деякі ключові аспекти моделювання в Blender:

- Створення об'єктів. У Blender ви можете створювати різні типи об'єктів, такі як куби, сфери, циліндри, конуси тощо, за допомогою вбудованих інструментів. Крім того, Blender підтримує створення складних об'єктів за допомогою моделювання з мешем (полігонами), більш складних кривих або використання різних модифікаторів для формування об'єктів.
- Редагування об'єктів. Після створення об'єктів ви можете їх редагувати, переміщуючи, обертаючи або масштабуючи їх за допомогою вбудованих інструментів редагування. Ви також можете видаляти, додавати або змінювати окремі елементи об'єктів, такі як вершини, ребра або грани.
- Моделювання персонажів. Blender надає інструменти для створення складних тривимірних моделей персонажів. Це включає в себе створення основної форми персонажа, нанесення деталей, таких як обличчя, волосся або одяг, а також роботу з анімацією скелета (ріга) та морфом (шейп ключів).
- Топологія моделей. При моделюванні важливо враховувати топологію моделі, тобто розподіл вершин, ребер та граней, який визначає форму та структуру об'єкта. Правильна топологія дозволяє забезпечити гладкість об'єкта, підтримку анімації та оптимальне моделювання.
- Модифікатори. Blender має ряд модифікаторів, які дозволяють змінювати форму об'єктів без зміни їх базової геометрії. Це дозволяє вам створювати складні форми шляхом комбінування та застосування різних модифікаторів.

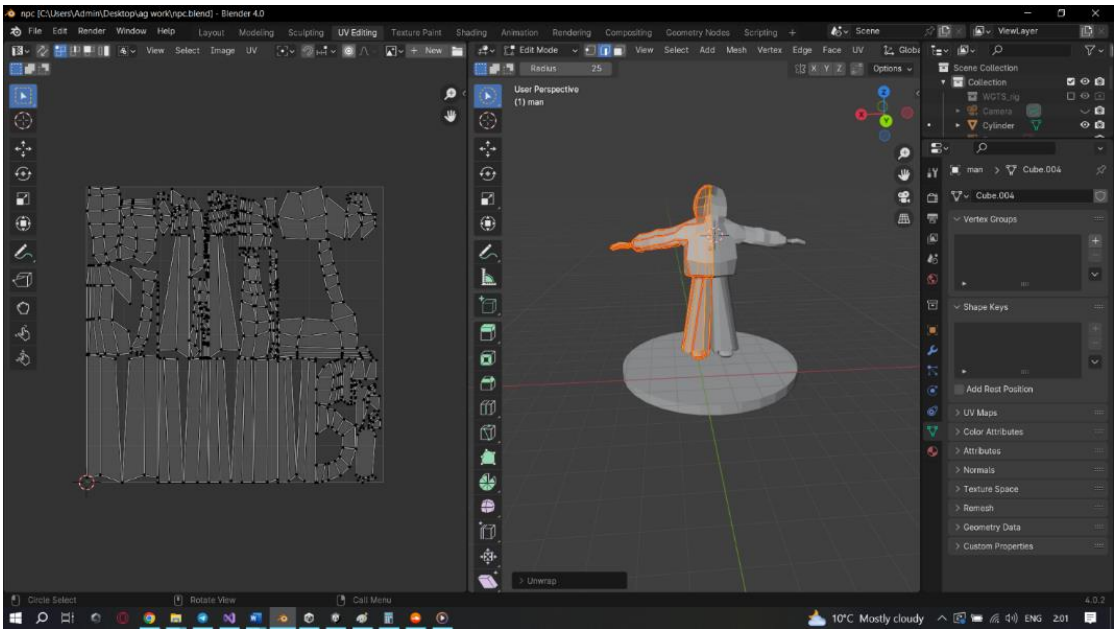

Рис. 2.1. Інтерфейс редактора

Переваги та недоліки програмного пакету Blender подамо у таблиці 2.1 [5].

Таблиця 2.1

Характеристика програмного пакету Blender

Переваги Blender	*Недоліки Blender*
Безкоштовність: Blender є повністю безкоштовним програмним забезпеченням, що робить його доступним для широкого кола розробників. Різноманітність функцій: Blender має широкі можливості для створення 3D-моделей, анімації, текстур та рендерингу, що дозволяє реалізувати різноманітні ідеї. Розширюваність: За допомогою додаткових модулів та плагінів, Blender може бути розширений для підтримки різних функцій та потреб розробки.	Вивчення: Blender має дещо високий поріг входження, і для новачків може знадобитися час, щоб зрозуміти всі його можливості. Не стільки орієнтований на розробку ігор: Blender не є спеціалізованим інструментом для розробки ігор, тому може бути менш підходящим для складних ігрових проектів порівняно з Unity.

2.2. Анімації у Blender

Анімації у Blender відображають процес створення рухомих зображень та сцен, який здійснюється за допомогою комп'ютерної програми Blender. Цей процес включає в себе створення та редагування рухів об'єктів, персонажів та камери для створення вражаючих анімаційних сцен.

Основні етапи процесу анімації в Blender включають:

- Створення ключових кадрів. Початковим етапом анімації є визначення ключових кадрів, які визначають початкове та кінцеве положення об'єкта чи персонажа. Ці ключові кадри встановлюються на певних кадрах у таймлайні анімації.
- Проміжні кадри (інтерполяція). Після встановлення ключових кадрів, розробник може додавати проміжні кадри, які визначають рух об'єкта між ключовими кадрами. Цей процес включає в себе використання різних методів інтерполяції, таких як лінійна, квадратична або кубічна, для плавного переходу між кадрами.
- Редагування траєкторії руху. Розробник може редагувати траєкторію руху об'єктів та персонажів, визначаючи шлях їх руху в просторі. Цей процес включає в себе використання інструментів траєкторії та графіків для точного налаштування руху.
- Додавання анімаційних ефектів. Після створення основної анімації розробник може додати різноманітні анімаційні ефекти, такі як зміна форми, деформація, вибухи, частинкові ефекти тощо. Blender надає широкий вибір інструментів для створення різноманітних анімаційних ефектів.
- Рендеринг та експорт. Після завершення процесу анімації розробник може здійснити рендеринг анімаційної сцени та експортувати її у відповідний формат для подальшого використання або публікації.

Для створення анімаційних скелетних систем для тривимірних об'єктів, що дозволяє їм рухатися, деформуватися та взаємодіяти у віртуальному просторі використовують **Rigging** у Blender – процес створення анімаційних скелетних

систем для тривимірних об'єктів, що дозволяє їм рухатися, деформуватися та взаємодіяти у віртуальному просторі. Цей процес є ключовим етапом у створенні анімаційних сцен та персонажів у Blender.

Основні компоненти риггінгу в Blender включають:

- Створення скелету. Риггінг починається зі створення скелетної системи, що складається з кісток та спеціальних контрольних елементів. Ці кістки дозволяють визначити структуру та рухи об'єкта під час анімації.
- Прив'язка скелету до моделі. Після створення скелетної системи, її необхідно прив'язати до моделі. Це забезпечує зв'язок між кістками та відповідними частинами моделі, такими як руки, ноги, тіло тощо.
- Налаштування контрольних точок. Риггінг також включає налаштування контрольних точок, які використовуються для керування рухом та деформацією об'єкта під час анімації. Ці контрольні точки дозволяють аніматорам зручно керувати об'єктом та його скелетною системою.
- Встановлення обмежень руху. Риггінг також включає встановлення обмежень руху для кожної кістки, що дозволяє обмежити її рух у певних напрямках та областях. Це допомагає уникнути неприродних деформацій та забезпечити реалістичний рух об'єкта.
- Тестування та налагодження. Після завершення риггінгу важливо провести тестування скелетної системи та відповідності моделі, а також внести необхідні корективи для досягнення оптимального результату.

2.3. UV-відображення (розгортка)

UV-відображення (або UV-розгортка) у Blender – це процес призначення двовимірних координат текстури тривимірному об'єкту. Кожній точці на поверхні об'єкта присвоюється відповідна точка на текстурі, що дозволяє програмі знати, яку частину текстури потрібно нанести на кожну точку об'єкта. Це дуже важливо для створення реалістичних та деталізованих моделей, оскільки текстури

дозволяють надати об'єктам візуальну інформацію, таку як кольори, текстури та інші деталі.

UV-відображення використовується для різних цілей, включаючи нанесення деталей, кольорів, шаблонів та інших графічних ефектів на модель. Цей процес є важливим кроком у створенні реалістичних та деталізованих тривимірних об'єктів у візуальних застосунках, таких як ігри, візуалізації або анімація.

У відомих графічних програмах, таких як Blender та Unity, користувачі можуть маніпулювати координатами UV для кожної точки моделі, редагуючи їх, щоб досягти потрібного візуального ефекту. Це може включати розміщення та масштабування текстур для кращого відображення, а також створення складних текстурних ефектів, таких як рельєфи, малюнки та текстурні переходи.

Створення UV-розгортки означає призначення двовимірних координат текстури об'єкту. У Blender це може бути зроблено різними способами, включаючи автоматичне розгортання, ручне розміщення або використання різних альтернативних методів, таких як Smart UV Project.

Після створення UV-розгортки можна використовувати різні інструменти для її оптимізації та редагування. Це може включати переміщення, обертання, масштабування або вирізання різних частин текстури, щоб забезпечити оптимальне використання простору UV-координат.

Висновок до розділу 2

Моделювання у Blender відображає важливий етап у процесі створення тривимірних об'єктів та сцен для використання у відеоіграх, анімації, візуалізації та інших сферах комп'ютерної графіки. Володіючи різноманітними інструментами та техніками, Blender дозволяє розробникам створювати складні та захоплюючі тривимірні об'єкти з високим ступенем деталізації.

Анімації у Blender є важливим етапом у процесі створення візуально захоплюючих та динамічних анімаційних сцен. Завдяки розширеним

можливостям та інструментам, що надає Blender, розробники можуть створювати складні анімаційні ефекти та динамічні рухи з високим ступенем деталізації.

Rigging у Blender – процес створення анімаційних скелетних систем для тривимірних об'єктів, що дозволяє їм рухатися, деформуватися та взаємодіяти у віртуальному просторі є ключовим етапом у створенні анімаційних сцен та персонажів у Blender.

У вирішенні проблеми UV-відображення важливу роль відіграють алгоритми розгортання текстур, які автоматично оптимізують розміщення UV-координат на поверхні моделі. Це допомагає уникнути перекривання текстур та інших артефактів і забезпечує високоякісне візуальне представлення моделі в графічних застосунках.

РОЗДІЛ 3. ЗАСТОСУВАННЯ СТЕКУ ТЕХНОЛОГІЙ UNITY/BLENDER

Використання Blender разом з Unity є популярним підходом у галузі розробки відеоігор та візуалізації. Blender, як потужний програмний пакет для 3D-моделювання, анімації та рендерингу, забезпечує розробникам можливість створення складних та деталізованих об'єктів, а також анімаційних сцен для використання в ігровому середовищі.

Unity, з іншого боку, є інтегрованим середовищем розробки (IDE) для створення ігор та інших інтерактивних застосунків. Його можливості включають у себе імпорт та обробку 3D-моделей, створення фізики, програмування геймплейних логік та візуалізацію.

Переваги:

- Експорт у формати, сумісні з Unity: Blender підтримує експорт моделей у формати, які можна безпосередньо імпортувати в Unity, такі як FBX або OBJ, зберігаючи структуру об'єктів, текстури та анімацію.
- Інтеграція з іншими аспектами розробки в Unity: Після імпортування моделей у Unity, розробники можуть додати їх до сцен, налаштувати колізії, фізичну поведінку та інші параметри, щоб інтегрувати їх в готову гру або інтерактивний застосунок.

3.1. Експорт у формати, сумісні з Unity

Експорт у формати, сумісні з Unity, представляє собою процес перетворення тривимірних моделей та даних із формату, що підтримується Blender, у формати, який може бути безпосередньо імпортований у середовище розробки Unity. Цей процес має важливе значення для інтеграції створених об'єктів, сцен та анімацій у розроблену гру чи інший інтерактивний застосунок.

У зв'язку з тим, що Unity підтримує різноманітні формати файлів для тривимірної графіки, такі як FBX, OBJ та інші, експорт із Blender в ці формати є

переважною практикою. Це забезпечує сумісність моделей, текстур, анімацій та інших елементів, створених у Blender, з середовищем розробки Unity.

Процес експорту у формати, сумісні з Unity, включає наступні етапи:

- Розробник повинен попередньо підготувати модель у Blender, включаючи налаштування текстур, матеріалів та анімацій.
- Після завершення підготовки моделі в Blender, розробник вибирає опцію експорту та обирає формат файлу, який підтримується Unity, наприклад, FBX.
- Під час експорту розробник може налаштувати параметри, такі як масштаб, координати та інші властивості моделі, щоб забезпечити правильне відображення у Unity.
- Отриманий файл імпортується у середовище розробки Unity, де розробник може продовжити роботу з моделлю, додавати її до сцени, налаштовувати фізику та інші параметри відповідно до вимог гри чи застосунку.

3.2. Інтеграція Blender зі сценами, скриптами, колізіями, фізичною поведінкою в Unity

Інтеграція Blender зі сценами, скриптами, колізіями, фізичною поведінкою та іншими аспектами розробки в Unity відображає суттєвий етап процесу створення відмінної візуальної та функціональної гри чи інтерактивного застосунку. Ця інтеграція включає в себе злагоджену взаємодію різноманітних елементів, що об'єднуються для створення злагодженого виробу.

- Сцени і об'єкти. Blender дозволяє розробникам створювати складні тривимірні моделі та сцени, які потім імпортуються у Unity. Ці сцени та об'єкти можуть бути легко розміщені в сценах Unity, де розробники можуть додавати додаткові ефекти, освітлення та інші компоненти для досягнення бажаного результату.
- Скрипти та програмування. Unity надає можливості для програмування різноманітних функцій та поведінки гри за допомогою скриптів, написаних

мовами програмування, такими як C# або JavaScript. Розробники можуть інтегрувати скрипти для управління рухом об'єктів, взаємодії з гравцем та багатьма іншими аспектами гри.

- Колізії та фізика. Після імпортування об'єктів з Blender у Unity, розробники можуть легко додавати колізії та налаштовувати фізичну поведінку об'єктів. Це дозволяє створювати реалістичну взаємодію об'єктів у грі, таку як зіткнення та взаємодія з навколишнім середовищем.
- Анімація та рухи. Blender забезпечує потужні інструменти для створення анімаційних сцен та персонажів. Ці анімації можуть бути експортовані у форматі, що підтримується Unity, і інтегровані у гру за допомогою скриптів та інших засобів розробки.

3.3 Реалізація прототипу 3D застосунку

На практиці успішно реалізовано прототип 3D гри, який включає наступні компоненти:

1. Анімовані та текстуровані моделі.

У Blender створено та анімовано декілька 3D моделей, які після цього були текстуровані для більшої реалістичності.

Процес створення моделі:

- Створення базової форми. Використання інструментів моделювання для створення базової форми моделі.
- Деталізація. Додавання деталей за допомогою скульптингу та модифікаторів.

На рис. 3.1 відображено створену та деталізовану модель.

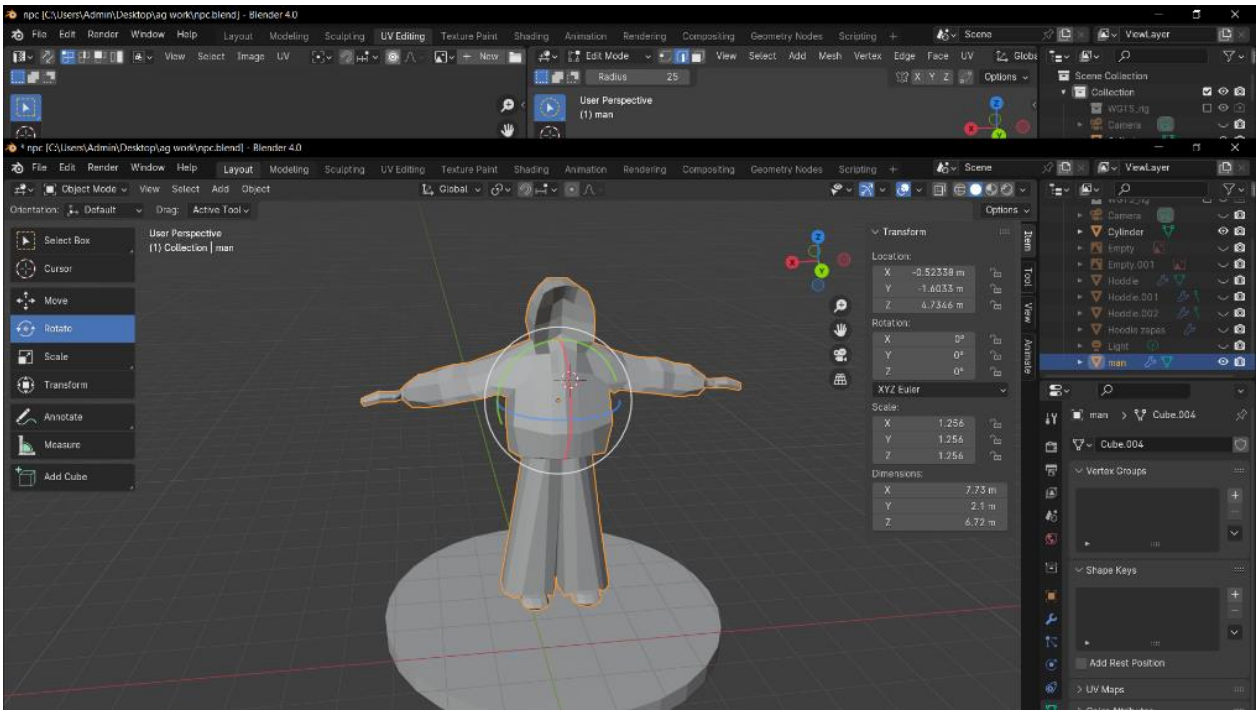

Рис. 3.1. Створена та деталізована модель

– UV-розгортка. Створення UV-розгортки для подальшого текстурування.
– Текстурування. Нанесення текстур з використанням різних шарів та матеріалів.

На рис. 3.2. відображено текстуровану модель.

Рис. 3.2. Текстурована модель

Процес анімації:

– Створення рігів (скелетів). Налаштування скелетної структури для моделі.
– Анімація рігів (рис. 3.3). Використання рігів для створення анімаційних послідовностей, таких як ходьба, біг, стрибки тощо.

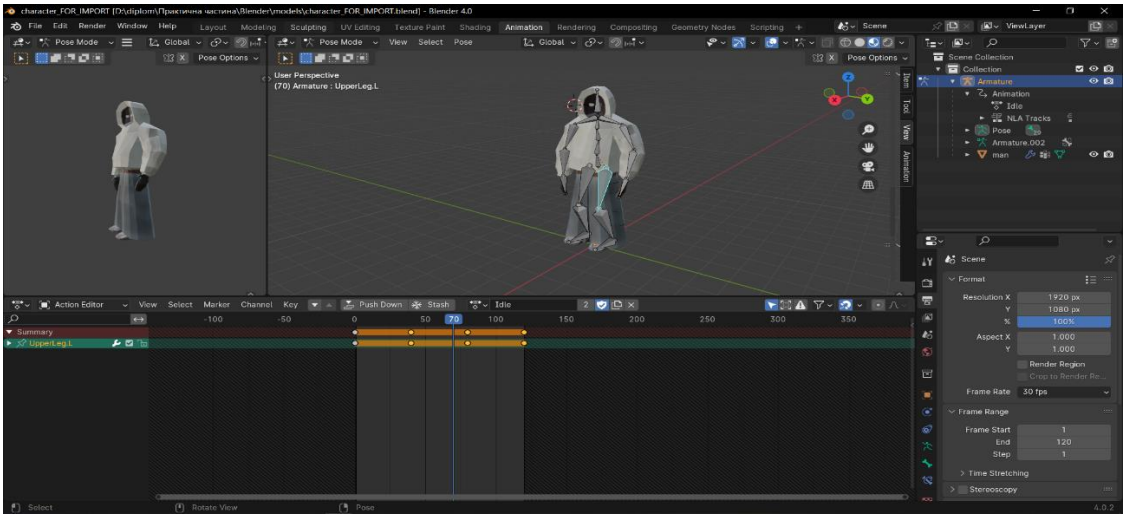

Рис. 3.3. Анімована модель з прив’язаним рігом

2. Контролери анімацій.

У Unity здійснено налаштування контролерів анімацій, які забезпечують плавні переходи між різними станами анімацій.

Налаштування контролера анімації:

– Створення анімаційного контролера. В Unity створено контролер, який керує анімаційними станами.
– Створення станів та переходів. Визначення різних анімаційних станів (наприклад, Idle, Walk) та налаштування переходів між ними.
– Налаштування параметрів. Використання параметрів для керування переходами між станами (наприклад, швидкість, напрямок).

На рис. 3.4 – вікно з налаштованим контролером анімацій.

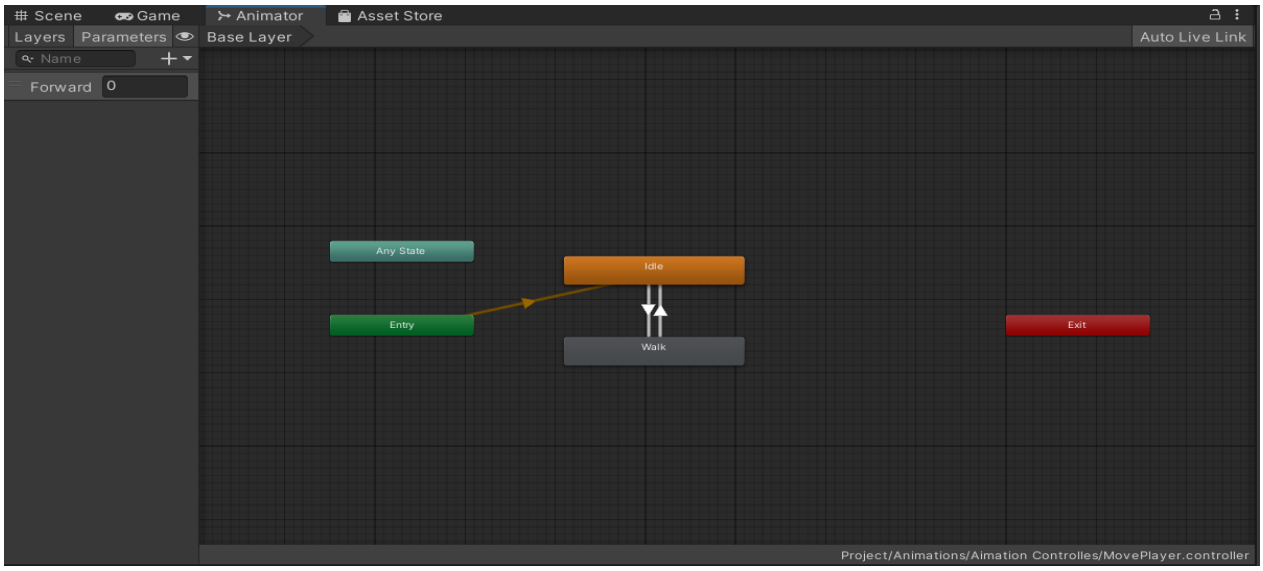

Рис. 3.4. Налаштований контролер анімацій

3. User Input та система введення

Реалізовано систему введення користувацьких команд, яка включає обробку введення з клавіатури та миші за допомогою нової системи введення Unity Input System.

Налаштування системи введення:

– Створення Input Actions (рис. 3.5). Визначення дій, які повинні бути виконані при натисканні клавіш або кнопок миші.
– Прив'язка до скриптів. Зв'язування введення користувача зі скриптами, які керують поведінкою ігрових об'єктів.

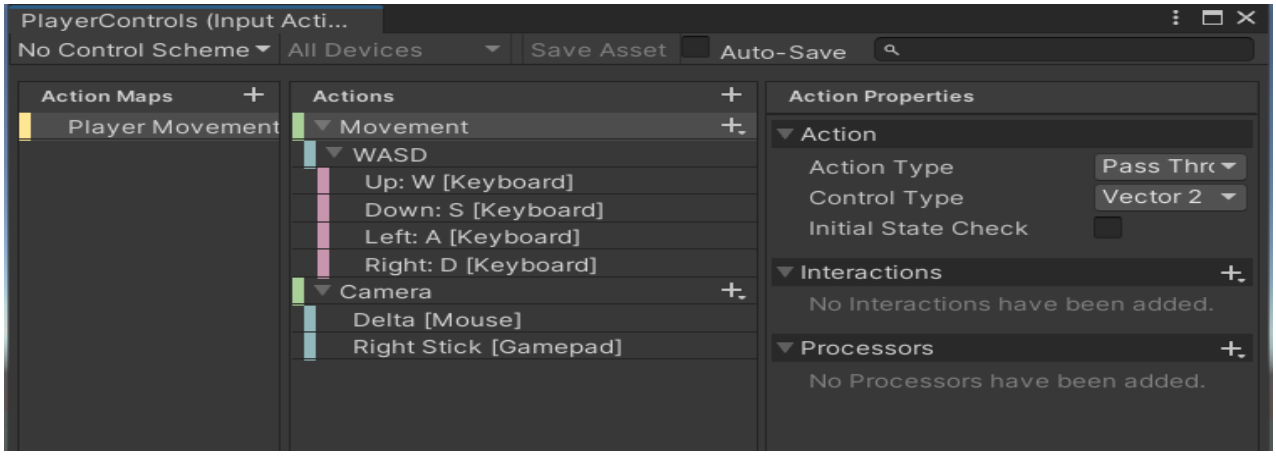

Рис. 3.5. Налаштований Input Action

Висновок до розділу 3

Інтеграція Blender з сценами, скриптами, колізіями, фізичною поведінкою та іншими аспектами розробки в Unity є ключовим етапом у процесі створення ігор та інтерактивних застосунків. Ця інтеграція дозволяє розробникам створювати складні та захоплюючі візуальні та ігрові ефекти, використовуючи потужність обох платформ.

Через комбінацію Blender та Unity розробники створюють вражаючі 3D-середовища та ігрові досвіди, поєднуючи потужність інструментів моделювання та анімації з можливостями програмування та візуалізації, що пропонує Unity.

Завдяки процесу експорту у формати, сумісні з Unity, розробники можуть ефективно інтегрувати моделі, анімації та інші графічні ресурси, створені у Blender, у процес розробки в Unity, сприяючи таким чином швидкому та продуктивному створенню ігор та інтерактивних застосунків.

ВИСНОВКИ

У ході даного дослідження було проаналізовано можливості та інструментарій двох потужних програмних середовищ – Unity та Blender – для створення ігрових застосунків. Основними напрямками дослідження були моделювання та анімація в Blender, а також інтеграція створеного контенту в Unity для подальшої розробки гри.

На практиці було успішно реалізовано прототип 3D гри, який включає: анімовані та текстуровані моделі, контролери анімацій, User Input та систему введення, інтерактивні елементи.

У Blender було створено та анімовано декілька 3D моделей, які після цього були текстуровані для більшої реалістичності.

У Unity було налаштовано контролери анімацій, які забезпечують плавні переходи між різними станами анімацій.

Реалізовано систему введення користувацьких команд, яка включає обробку введення з клавіатури та миші за допомогою нової системи введення Unity Input System.

У прототипі реалізовано базову взаємодію між ігровими об’єктами та гравцем, що дозволяє демонструвати основні можливості ігрового рушія Unity.

Дослідження показало, що комбінація Blender і Unity є ефективним підходом для створення 3D ігрових застосунків. Blender забезпечує високий рівень контролю над моделями та анімаціями, тоді як Unity надає потужні інструменти для інтеграції цих елементів у функціональний ігровий прототип.

У підсумку, використання Blender та Unity дозволяє ефективно реалізовувати складні ігрові проєкти, надаючи розробникам потужні інструменти для створення якісного контенту та його інтеграції в ігрове середовище.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Unity Documentation URL: https://docs.unity.com/ (Дата звернення: 19.05.2024)
2. Unity User Manual URL: https://docs.unity3d.com/Manual/index.html (Дата звернення: 19.05.2024)
3. Blender Documentation URL: https://docs.blender.org/ (Дата звернення: 19.05.2024)
4. Blender 4.1 Manual https://docs.blender.org/manual/en/latest/# (Дата звернення: 19.05.2024)
5. Цинчик Д. Дослідження можливостей рушіїв UNITY та BLENDER для створення ігрових застосунків : Збірник матеріалів наукової конференції за підсумками науково-дослідної роботи здобувачів вищої освіти фізико-математичного факультету Кам'янець-Подільського національного університету імені Івана Огієнка у 2023-2024 н.р., 9-10 квітня 2024 року [Електронний ресурс]. Кам'янець-Подільський : Кам'янець-Подільський національний університет імені Івана Огієнка, фізико-математичний факультет, 2024. С. 102-105.
6. Pluralsight Learning Unity: https://www.pluralsight.com/paths/unity (Дата звернення: 19.05.2024)
7. CG Cookie. Blender Fundamentals: https://www.cgcookie.com/learn-blender (Дата звернення: 19.05.2024)
8. Unity Blog. "Latest Updates in Unity": https://blog.unity.com/ (Дата звернення: 19.05.2024)
9. Blender Artists. "Blender Community Projects and News": https://blenderartists.org/ (Дата звернення: 19.05.2024)
10. Gregory, Jason. Game Engine Architecture. A K Peters/CRC Press, 2017.
11. Goldstone, Will. Unity 2018 Game Development in 24 Hours, Sams Teach Yourself: The Official Guide to Learning Unity. Sams Publishing, 2018.

12. Hess, Roland. Blender Foundations: The Essential Guide to Learning Blender 2.6. Focal Press, 2010.
13. Cook, Nick. "The Use of Unity for Game Development". Journal of Game Development, vol. 5, no. 2, 2019, C. 45-58.
14. Petrov, Ivan. "Blender's Role in Modern Game Design". International Journal of Computer Graphics, vol. 7, no. 3, 2020,C. 67-82.
15. Unity Forum. Available at: https://forum.unity.com/ (Дата звернення: 19.05.2024)
16. Blender Artists Forum. Available at: https://blenderartists.org/ (Дата звернення: 19.05.2024)
17. Stack Overflow. "Questions and Answers on Unity Development": https://stackoverflow.com/questions/tagged/unity3d (Дата звернення: 19.05.2024)
18. Reddit. "Subreddit for Blender Users": https://www.reddit.com/r/blender/ (Дата звернення: 19.05.2024)

Додаток А

Опис базового класу усіх колайдерів в Unity

Цей код - частина простору імен UnityEngine і описує базовий клас всіх колайдерів у Unity - Collider.

Простори імен і визначення класу:

- Код знаходиться в просторі імен UnityEngine.
- Клас Collider є базовим класом для всіх колайдерів у Unity.

Атрибути класу:

- [RequireComponent(typeof(Transform))]: Цей атрибут вказує, що компонент Transform обов'язково повинен бути доданий до того ж об'єкта, на якому розміщений цей колайдер.
- [RequiredByNativeCode]: Цей атрибут вказує, що клас є необхідним для внутрішнього коду Unity.
- [NativeHeader("Modules/Physics/Collider.h")]: Вказує, що інформація про клас Collider знаходиться в заголовковому файлі Collider.h модуля фізики Unity.

Властивості:

- enabled: Вказує, чи є колайдер активним.
- isTrigger: Вказує, чи є колайдер тригером.
- contactOffset: Відстань між колайдерами, на якій вони зіштовхуються.

Методи:

- ClosestPoint(Vector3 position): Повертає найближчу точку на колайдері до заданої позиції.
- ClosestPointOnBounds(Vector3 position): Повертає найближчу точку на межах колайдера до заданої позиції.
- Raycast(Ray ray, out RaycastHit hitInfo, float maxDistance): Виконує променеве виявлення зіткнення з колайдером.

Виклики методів, реалізованих нативно:

- Багато методів помічені атрибутом [MethodImpl(MethodImplOptions.InternalCall)], що вказує на їхню реалізацію на рівні мови програмування C++ або C#.
- Деякі методи помічені [MethodImpl(MethodImplOptions.InternalCall)] і extern, що вказує на виклик зовнішньої функції.

Є виклики методів, які зберігаються нативно.

Отже, цей код описує основні характеристики та функціональність класу Collider в Unity, такі як активація, тригери, променеві виявлення зіткнень та інше.

```
using System.Runtime.CompilerServices;
using UnityEngine.Bindings;
using UnityEngine.Scripting;

namespace UnityEngine
{
    //
    //     A base class of all colliders.
    [RequireComponent(typeof(Transform))]
    [RequiredByNativeCode]
    [NativeHeader("Modules/Physics/Collider.h")]
    public class Collider : Component
    {
        //
        //     Enabled Colliders will collide with other Colliders, disabled Colliders
won't.
        public extern bool enabled
        {
            [MethodImpl(MethodImplOptions.InternalCall)]
            get;
            [MethodImpl(MethodImplOptions.InternalCall)]
            set;
        }

        //
        // Сводка:
        //     The rigidbody the collider is attached to.
        public extern Rigidbody attachedRigidbody
        {
            [MethodImpl(MethodImplOptions.InternalCall)]
            [NativeMethod("GetRigidbody")]
            get;
        }

        //
        // Сводка:
        //     The articulation body the collider is attached to.
        public extern ArticulationBody attachedArticulationBody
        {
            [MethodImpl(MethodImplOptions.InternalCall)]
            [NativeMethod("GetArticulationBody")]
            get;
        }

        //
        //     Specify if this collider is configured as a trigger.
        public extern bool isTrigger
        {
            [MethodImpl(MethodImplOptions.InternalCall)]
            get;
            [MethodImpl(MethodImplOptions.InternalCall)]
            set;
        }

        //
        //     Contact offset value of this collider.
        public extern float contactOffset
        {
```

```
            [MethodImpl(MethodImplOptions.InternalCall)]
            get;
            [MethodImpl(MethodImplOptions.InternalCall)]
            set;
        }

        //
        //     The world space bounding volume of the collider (Read Only).
        public Bounds bounds
        {
            get
            {
                get_bounds_Injected(out var ret);
                return ret;
            }
        }

        //
        //     Specify whether this Collider's contacts are modifiable or not.
        public extern bool hasModifiableContacts
        {
            [MethodImpl(MethodImplOptions.InternalCall)]
            get;
            [MethodImpl(MethodImplOptions.InternalCall)]
            set;
        }

        //
        //     Whether or not this Collider generates contacts for
Physics.ContactEvent.
        public extern bool providesContacts
        {
            [MethodImpl(MethodImplOptions.InternalCall)]
            get;
            [MethodImpl(MethodImplOptions.InternalCall)]
            set;
        }
        //     A decision priority assigned to this Collider used when there is a
conflicting
        //     decision on whether a Collider can contact another Collider.
        public extern int layerOverridePriority
        {
            [MethodImpl(MethodImplOptions.InternalCall)]
            get;
            [MethodImpl(MethodImplOptions.InternalCall)]
            set;
        }

        //     The additional layers that this Collider should exclude when deciding
if the
        //     Collider can contact another Collider.
        public LayerMask excludeLayers
        {
            get
            {
                get_excludeLayers_Injected(out var ret);
                return ret;
            }
            set
            {
                set_excludeLayers_Injected(ref value);
            }
        }

        //
```

```
        //     The additional layers that this Collider should include when deciding
if the
        //     Collider can contact another Collider.
        public LayerMask includeLayers
        {
            get
            {
                get_includeLayers_Injected(out var ret);
                return ret;
            }
            set
            {
                set_includeLayers_Injected(ref value);
            }
        }

        //
        //     The shared physic material of this collider.
        [NativeMethod("Material")]
        public extern PhysicMaterial sharedMaterial
        {
            [MethodImpl(MethodImplOptions.InternalCall)]
            get;
            [MethodImpl(MethodImplOptions.InternalCall)]
            set;
        }
        //     The material used by the collider.
        public extern PhysicMaterial material
        {
            [MethodImpl(MethodImplOptions.InternalCall)]
            [NativeMethod("GetClonedMaterial")]
            get;
            [MethodImpl(MethodImplOptions.InternalCall)]
            [NativeMethod("SetMaterial")]
            set;
        }

        //     Returns a point on the collider that is closest to a given location.
        //     Location you want to find the closest point to.
        //     The point on the collider that is closest to the specified location.
        public Vector3 ClosestPoint(Vector3 position)
        {
            ClosestPoint_Injected(ref position, out var ret);
            return ret;
        }

        private RaycastHit Raycast(Ray ray, float maxDistance, ref bool hasHit)
        {
            Raycast_Injected(ref ray, maxDistance, ref hasHit, out var ret);
            return ret;
        }

        public bool Raycast(Ray ray, out RaycastHit hitInfo, float maxDistance)
        {
            bool hasHit = false;
            hitInfo = Raycast(ray, maxDistance, ref hasHit);
            return hasHit;
        }

        [NativeName("ClosestPointOnBounds")]
        private void Internal_ClosestPointOnBounds(Vector3 point, ref Vector3 outPos,
ref float distance)
        {
            Internal_ClosestPointOnBounds_Injected(ref point, ref outPos, ref
distance);
```

```
        }


        //     The closest point to the bounding box of the attached collider.
        //
        // Параметры:
        //   position:
        public Vector3 ClosestPointOnBounds(Vector3 position)
        {
            float distance = 0f;
            Vector3 outPos = Vector3.zero;
            Internal_ClosestPointOnBounds(position, ref outPos, ref distance);
            return outPos;
        }

        [MethodImpl(MethodImplOptions.InternalCall)]
        private extern void ClosestPoint_Injected(ref Vector3 position, out Vector3
ret);

        [MethodImpl(MethodImplOptions.InternalCall)]
        [SpecialName]
        private extern void get_bounds_Injected(out Bounds ret);

        [MethodImpl(MethodImplOptions.InternalCall)]
        [SpecialName]
        private extern void get_excludeLayers_Injected(out LayerMask ret);

        [MethodImpl(MethodImplOptions.InternalCall)]
        [SpecialName]
        private extern void set_excludeLayers_Injected(ref LayerMask value);

        [MethodImpl(MethodImplOptions.InternalCall)]
        [SpecialName]
        private extern void get_includeLayers_Injected(out LayerMask ret);

        [MethodImpl(MethodImplOptions.InternalCall)]
        [SpecialName]
        private extern void set_includeLayers_Injected(ref LayerMask value);

        [MethodImpl(MethodImplOptions.InternalCall)]
        private extern void Raycast_Injected(ref Ray ray, float maxDistance, ref bool
hasHit, out RaycastHit ret);

        [MethodImpl(MethodImplOptions.InternalCall)]
        private extern void Internal_ClosestPointOnBounds_Injected(ref Vector3 point,
ref Vector3 outPos, ref float distance);
    }
}
```

Додаток Б

Логіка керування гравцем

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Security.Cryptography.X509Certificates;
using UnityEngine;

public class PlayerLogic : MonoBehaviour
```

```
{
    //заголовки над полями в Інспекторі для зручності відлагодження
    [Header("Visuals")]
    public GameObject playerModel;
    public float rotatingSpeed = 2f;
    public Animator playerAnimator;

    [Header("Equipment")]
    public gunLogic gun;
    public KnifeLogic knife;
    public GameObject sandThrowPrefab;
    public float throwingSpeed;
    public int sandAmount = 5;
    public int bulletsAmount = 40;

    [Header("Movement")]
    public float movingVelocity;
    public float jumpingVelocity;

    private bool canJump;

    //створення об'єкту класу Rigidbody
    private Rigidbody playerRigidBody;
    private Vector3 originalPlayerAnimatorPosition;



    //  Метод Start викликається перед першим оновленням кадру (frame)
    void Start()
    {
        //запис в обєкт класу компоненту
        playerRigidBody = GetComponent<Rigidbody>();
        originalPlayerAnimatorPosition = playerAnimator.transform.localPosition;
    }



    // Метод Update викликається один раз на кадр
    void Update()
    {
        //перевірка чи можна стрибати
        RaycastHit hit;
        if (Physics.Raycast(transform.position, Vector3.down, out hit, 1.35f))
        {
            canJump = true;
        }


        void ProcessInput()
        {
            playerRigidBody.velocity = new Vector3(
            0,
            playerRigidBody.velocity.y,
            0);
        }
        ProcessInput();

        //Утримання аніматору персонажу на місці
        playerAnimator.transform.localPosition = originalPlayerAnimatorPosition;

        //стрибок
        if (canJump && Input.GetKeyDown("space"))
        {
            canJump = false;
```

```
            playerRigidBody.velocity = new Vector3(
                playerRigidBody.velocity.x,
                jumpingVelocity,
                playerRigidBody.velocity.y);
        }

        bool isPlayerMoving = false;

        //Рух на WASD
        if (Input.GetKey("d"))
        {
            playerRigidBody.velocity = new Vector3(
                movingVelocity,
                playerRigidBody.velocity.y,
                playerRigidBody.velocity.z
                );

            playerModel.transform.rotation =
Quaternion.Lerp(playerModel.transform.rotation,
Quaternion.Euler(0,90,0),Time.deltaTime *rotatingSpeed);
            isPlayerMoving = true;
        }
        if (Input.GetKey("a"))
        {
            playerRigidBody.velocity = new Vector3(
                -movingVelocity,
                playerRigidBody.velocity.y,
                playerRigidBody.velocity.z
                );

            playerModel.transform.rotation =
Quaternion.Lerp(playerModel.transform.rotation, Quaternion.Euler(0, 270, 0),
Time.deltaTime * rotatingSpeed);
            isPlayerMoving = true;
        }
        if (Input.GetKey("w"))
        {
            playerRigidBody.velocity = new Vector3(
                playerRigidBody.velocity.x,
                playerRigidBody.velocity.y,
                movingVelocity
                );

            playerModel.transform.rotation =
Quaternion.Lerp(playerModel.transform.rotation, Quaternion.Euler(0, 0, 0),
Time.deltaTime * rotatingSpeed);
            isPlayerMoving = true;
        }
        if (Input.GetKey("s"))
        {
            playerRigidBody.velocity = new Vector3(
                playerRigidBody.velocity.x,
                playerRigidBody.velocity.y,
                -movingVelocity
                );

            playerModel.transform.rotation =
Quaternion.Lerp(playerModel.transform.rotation, Quaternion.Euler(0, 180, 0),
Time.deltaTime * rotatingSpeed);
            isPlayerMoving = true;
        }

        playerAnimator.SetFloat("Forward", isPlayerMoving ? 1f : 0f);
```

```
        if (Input.GetKey(KeyCode.LeftShift) && Input.GetMouseButtonDown(0))
        {
            knife.StrongAttack();
        }
        if (Input.GetMouseButtonDown(0))
        {

            knife.Attack();
        }

        if (Input.GetKeyDown("x"))
        {
            ThrowSand();
        }

        if (Input.GetKeyDown("c"))
        {
            if (bulletsAmount>0)
            {
                gun.Attack();
                bulletsAmount--;
            }

        }
    }
    private void ThrowSand()
    {
        if (sandAmount <= 0)
        {
            return;
        }
        GameObject sandObject = Instantiate(sandThrowPrefab);
        sandObject.transform.position = transform.position +
playerModel.transform.forward;
        Vector3 throwingDirection = (playerModel.transform.forward).normalized;

sandObject.GetComponent<Rigidbody>().AddForce(throwingDirection*throwingSpeed);
        sandAmount--;
    }
}
```

Додаток В

Логіка слідування камери за гравцем

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

public class NewBehaviourScript : MonoBehaviour
{
    //ціль до якої привязується камера і слідує за нею
    public GameObject target;
    //зміна яка визначає положення камери відносно цілі
    public Vector3 offset;
    //зміна для зглажування руху камери за ціллю
    public float focusSpeed = 1f;
    // Start is called before the first frame update
    void Start()
    {
```

```
    }

    // Update is called once per frame
    void Update()
    {
        //оновлення камери викликається кожен кадр
        transform.position = Vector3.Lerp(transform.position,
target.transform.position + offset, Time.deltaTime*focusSpeed);
    }
}
```