МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Державне некомерційне підприємство

«Державний університет» Київський авіаційний інститут»

Факультет комп'ютерних наук та технологій

Кафедра інженерії програмного забезпечення

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_____ Олена ГРІНЕНКО

«_____» _____ 2025 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ЗДОБУВАЧА ОСВІТНЬОГО СТУПЕНЯ «МАГІСТР»
ЗАОЧНА ФОРМА ЗДОБУТТЯ ОСВІТИ

Тема: Методика розроблення комп'ютерної гри жанру 3D-платформер з елементами паркуру та процедурними анімаціями

Виконавець: Новіков Євгеній Олександрович

Керівник: к. е. н., доцент Ткаченко Костянтин Олександрович

Нормоконтролер: к. е. н., доцент Ткаченко Костянтин Олександрович

Київ 2025

Державне некомерційне підприємство
«Державний університет» Київський авіаційний інститут»

Факультет комп’ютерних наук та технологій
Кафедра інженерії програмного забезпечення
Спеціальність 121 «Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення»
Заочна форма здобуття освіти

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Олена ГРІНЕНКО
«_____» _____ 2025 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи студента
Новікова Євгенія Олександровича

1. Тема кваліфікаційної роботи: «Методика розроблення комп’ютерної гри жанру 3D-платформер з елементами паркуру та процедурними анімаціями» затверджена наказом ректора від «17» листопада 2025 р. № 2449/ст.

2. Термін виконання проекту: з 29.10.2025 р. по 14.12.2025 р.

3. Вихідні дані до роботи: прототип ігрового проекту з елементами паркуру, що включає системи процедурної анімації, контролю персонажа та відладки.

4. Зміст пояснювальної записки:
   1. Дослідження предметної області проекту.
   2. Аналіз та розробка підсистем ігрового продукту.
   3. Реалізація прототипу ігрового продукту.

5. Перелік обов’язкового графічного (ілюстративного) матеріалу:
   1. Демонстрація алгоритмів процедурної анімації.
   2. Демонстрація інструментів відладки (інтерфейс, 3D-візуалізації).
   3. Демонстрація реалізованих станів гравця, анімацій та паркур-трюків.

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1.	Розробка та затвердження графіка роботи.	29.09-01.10.2025	виконано
2.	Ознайомлення з постановкою задачі, вивчення інформаційних джерел та складання плану роботи.	02.10-05.10.2025	виконано
3.	Проведення аналізу, розробка алгоритмів програмного продукту.	06.10-12.10.2025	виконано
4.	Оформлення 1 розділу та подання його керівнику.	13.10-19.10.2025	виконано
5.	Розробка та тестування програмного рішення.	20.10-30.10.2025	виконано
6.	Оформлення 2 розділу на основі розроблених систем, подання розділу керівнику.	31.10-10.11.2025	виконано
7.	Завершення реалізації програмного продукту. Оформлення 3 розділу на основі деталей реалізації ігрового прототипу та подання його керівнику.	11.11-24.11.2025	виконано
8.	Редагування та остаточне оформлення тексту пояснювальної записки.	25.11-30.11.2025	виконано
9.	Представлення роботи для перевірки на академічну доброчесність. Проходження нормоконтролю.	01.12.2025	виконано
10.	Підготовка презентації та тексту доповіді.	02.12-10.12.2025	виконано
11.	Рецензування кваліфікаційної роботи.	15.12-18.12.2025	виконано
12.	Здача секретарю ЕК пояснювальної записки.	19.12.2025	
13.	Захист кваліфікаційної роботи перед екзаменаційною комісією.	23.12.2025	

Дата видачі завдання 29.09.2025 р.

Керівник кваліфікаційної роботи:

к. е. н., доцент Костянтин ТКАЧЕНКО

Завдання прийняв до виконання: Євгеній НОВІКОВ

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Методика розроблення комп’ютерної гри жанру 3D-платформер з елементами паркуру та процедурними анімаціями»: 135 сторінок, 52 рисунки, 11 використаних джерел, 2 додатки.

Об’єкт дослідження – процеси розробки та відладки комп’ютерних ігор зі складними контролерами персонажів та процедурними анімаціями.

Мета кваліфікаційної роботи – розробка підсистем процедурної 3D-анімації та контролера персонажа, інструментів відладки та ігрового прототипу, що їх використовує.

Методи дослідження – аналіз літературних джерел та існуючих комерційних ігор; планування, моделювання та дизайн ігрових систем; розробка, оптимізація та плейтестинг ігрового програмного продукту.

Результати роботи можуть бути використані при:

- впроваджені складних механік руху у комп’ютерних іграх;
- впроваджені елементів, або повноцінних систем процедурної анімації у 3D-додатках;
- розробці гнучких, підтримуваних та розширюваних контролерів персонажів у комп’ютерних іграх;
- розробці інструментів аналізу та відладки 3D-додатків.

Розробка та дослідження проводилися під управлінням ОС Windows 11. Розробка програми проводилася у середовищах Visual Studio Community 2022 Preview 17.14.10 Preview 1.0, JetBrains Rider 2025.2.3 та Godot Engine v4.4.1.stable.mono.official, на мові програмування C# (.NET 8).

КОМП’ЮТЕРНА ГРА, ПРОЦЕДУРНА АНІМАЦІЯ, КОЛІЗІЇ, ІНВЕРСНА КІНЕМАТИКА, ІГРОВА ФІЗИКА, 3D АНІМАЦІЯ, ІГРОВИЙ РУШІЙ, C#, GODOT, ПАТЕРНИ, ПАТЕРН СТАНУ, ПАТЕРН КОМПОНЕНТУ, ВІДЛАДКА, ДОМІШКИ.

ABSTRACT

Explanatory note to the qualification thesis "Methodology for Developing a 3D Platformer Video Game with Parkour Elements and Procedural Animations": 135 pages, 52 figures, 11 citations, 2 appendices.

Object of study – the processes of developing and debugging videogames with complex character controllers and procedural animations.

Aim of the qualification thesis – the development of procedural 3D animation and character controller subsystems, debugging tools, and a game prototype that uses them.

Research methods – analysis of literary sources and existing commercial games; planning, modeling and designing game systems; development, optimization and playtesting of the game software product.

Results of the work can be used for:

- implementing complex movement mechanics in videogames;
- implementing elements or full-fledged systems of procedural animation in 3D applications;
- developing flexible, maintainable, and extensible character controllers in videogames;
- developing tools for analysis and debugging of 3D applications.

Development and research were performed using the Windows 11 operating system. Software development was done in Visual Studio Community 2022 Preview 17.14.10 Preview 1.0, JetBrains Rider 2025.2.3 and Godot Engine v4.4.1.stable.mono.official, using the C# (.NET 8) programming language.

VIDEOGAME, PROCEDURAL ANIMATION, COLLISIONS, INVERSE KINEMATICS, GAME PHYSICS, 3D ANIMATION, GAME ENGINE, C#, GODOT, PATTERNS, STATE PATTERN, COMPONENT PATTERN, DEBUGGING, MIXINS.

ЗМІСТ

ТЕРМІНИ ТА СКОРОЧЕННЯ

API	–	Набір визначених засобів взаємодії з програмним модулем.
Домішка (Mixin)	–	Клас з методами, призначеними для використання іншими класами без необхідності спадкування.
Інверсна кінематика (IK)	–	Алгоритм позиціонування ланок кінематичного ланцюга виходячи з позиції кінцевої ланки.
Ігрові механіки (Game mechanics)	–	Набір певних правил, функцій та явищ, що разом формують ігровий процес гри.
Трансформація (Transform)	–	Позиція (Position), оберт (Rotation) та масштаб (Scale) об’єкта в 3D просторі.
Ресурс (Resource)	–	В рушії Godot – клас, об’єкти якого можуть бути (де)серіалізовані та збережені на диску як файли.
Рейкастинг/шейпкастинг (Raycast/shapecast)	–	Операції сканування простору у фізичному рушії.

ВСТУП

Актуальність теми. Одним з найпопулярніших напрямків розробки програмного забезпечення на сьогоднішній день є розробка комп'ютерних ігор. Комп'ютерні, мобільні та консольні ігри є одним з ключових елементів індустрії розваг, приносячи мільярди доларів прибутку та даючи мільйонам людей нові способи розважатись, знаходити друзів та заробляти на життя.

Розробка комп'ютерних ігор також є значним рушієм прогресу в сфері програмної та комп'ютерної інженерії. Вона сприяє розвитку технологій:

- рендерингу та графічних API;
- фізичної симуляції;
- доповненої та віртуальної реальності (AR/VR);
- оптимізації ПЗ;
- кросплатформеного програмування;
- обробки аудіо, процедурного та просторового звуку;
- хмарних обчислень, мережевої та веб-інфраструктури.

Враховуючи корисність та інноваційність процесу розробки ігор, було прийнято рішення провести дослідження саме в цій галузі. Конкретний напрямок дослідження (контролер гравця, 3D-анімації та фізика) був обраний враховуючи популярність таких жанрів комп'ютерних ігор як First-Person Shooter, Third-Person Shooter, Action Platformer та Movement Shooter.

Розробка технологій, пов'язаних з цією категорією ігор, буде сприяти покращенню та оптимізації як і внутрішніх процесів розробки комп'ютерних ігор, так і привабливості ігор та ігрових процесів для їх користувачів.

Мета і завдання виконання кваліфікаційної роботи. Дослідити існуючі реалізації ігрових систем контролера, процедурної анімації та колізії гравця. Розробити нові рішення цим проблемам, покращуючи їх функціонал та зручність користування. Розробити інструменти відладки, доречні для використання у 3D-грі.

Розробити ігровий прототип, поєднавши створені системи в одне ціле та продемонструвавши їх ефективність.

Об’єктом дослідження є процеси розробки та відладки комп’ютерних ігор.

Предметом дослідження є такі складові процесу розробки ігор: контролер гравця, 3D-анімація гуманоїдних персонажів, колізія гравця, способи руху гравця, інструменти дослідження стану гри та візуалізації її даних.

Методи дослідження. Методами дослідження в даній кваліфікаційній роботі є:

1. Аналіз літературних джерел (технічної документація ігрових рушіїв, наукових праць, статей та публікацій від розробників ігор).
2. Дослідження існуючих комерційних ігор з метою визначення їх підходів до реалізації складних ігрових систем.
3. Планування, моделювання та дизайн ігрових систем.
4. Розробка, оптимізація та плейтестинг ігрового програмного продукту.

Наукова новизна отриманих результатів. На основі проведеного дослідження та розробки, пропонується низка архітектур ігрових модулів, а саме контролер гравця на основі патернів State та Component, система процедурної анімації для 3D-моделей, а також підходи до реалізації колізії та руху гравця на їх основі. Розроблені рішення вдосконалюють існуючі системи та алгоритми, створюючи основу для ефективної розробки нових ігрових застосунків, та для подальших досліджень.

РОЗДІЛ 1
АНАЛІЗ ТА ПРОБЛЕМАТИКА ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Використання State Pattern та Component Pattern у контролерах гравця

При розробці ігрових додатків часто постає задача створення певних сутностей – гравець, ворог, бот тощо. Написання логіки таких сутностей може здатись легкою задачею, але зазвичай з часом стає очевидно що це зовсім не так. Одна з багатьох проблем, з якою стикаються розробники є складність організації самого коду. Класи, які на початку складалися з пари змінних та методів, багатократно виростають в об’ємі, і додавати нові функції стає важче і важче.

Першим кроком до вирішення цієї проблеми є впровадження концепції “стану” [1]. Ми визначаємо, що в будь-який момент часу, наша сутність знаходиться в певному стані. Для прикладу можна взяти уявну сутність Потяг у грі-симуляторі залізниці. Потяг може перебувати в одному з чотирьох станів: Рухається, Чекає (на станції), Зупинений (в депо) та Ремонтується.

Для впровадження цих станів можна використати перерахування (enum), наприклад:

```
public enum TrainState {
    Moving,
    Waiting,
    Off,
    UndergoingMaintenance
}
```

Використавши конструкцію switch, можна змінювати поведінку контролера в залежності від поточного стану:

```
switch(trainState) {
    case Moving:
      ...
    case Off:
      ...
}
```

Цей підхід покращує організованість коду, але з часом цього буде недостатньо. Наступним кроком буде доречне використання State Pattern [2]. При використанні патерну станів, кожний стан представляється у вигляді окремого класу, наприклад:

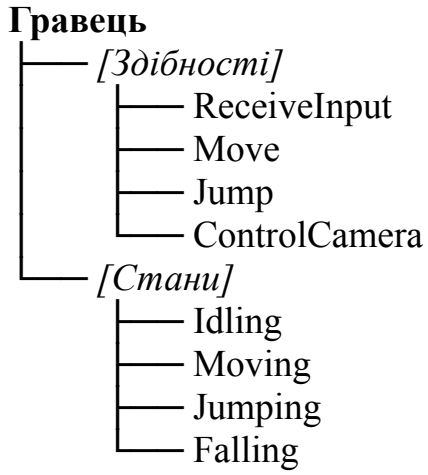
```
public abstract class TrainStateBase {}
public class Moving : TrainStateBase {}
public class Waiting : TrainStateBase {}
```

Перевагою такого підходу є відокремлення змінних та логіки кожного стану, що значно покращує читабельність коду, та робить легшим його підтримку. Недоліком цього методу є те, що певний функціонал (методи, властивості) контролера може знадобитись більш ніж одному стану – тоді його доведеться залишити в основному класі.

Існує також інший підхід для відокремлення логіки класу – композиція [3]. При впровадження композиції, функціонал класу виноситься в окремі, незалежні компоненти. Далі всі компоненти, необхідні для функціонування класу, приєднуються до нього за допомогою певного елементу інфраструктури, що також надає API для взаємодії з цими компонентами.

Для створення складного контролеру персонажів, ми можемо поєднати ці техніки в одну, створивши систему, де є компоненти різних видів. Базовий компонент може бути приєднаний до сутності і буде надавати їй властивості та методи. Для охайності публічного інтерфейсу, його можна унаслідувати в компонент-здібність (Ability), що також дозволить додати до нього методи-утиліти. Компоненти-стани (States), що наслідують від базових компонентів, поділяють весь функціонал компонентів, але з одним обмеженням: лише один об’єкт стану може бути активний в будь-який момент часу.

В якості прикладу візьмемо концепцію простої двовимірної гри-платформера. Гравець має можливість переміщатись вліво та вправо, а також стрибати, і як результат – падати. Використавши концепцію контролера станів та здібностей, ці механіки можна реалізувати за наступною схемою:



В стані Idling гравець стоїть на місці. В стані Moving гравець рухається вліво чи вправо, в залежності від вектору вводу. Якщо під гравцем не знайдено підлоги, стан переключається на Falling. В стані Jumping гравець прискорюється вверх, поступово сповільнюючись. Коли вертикальна швидкість досягає 0, стан переключається на Falling. В стані Falling гравець прискорюється вниз, а при торканні підлоги – стан переключається на Idling.

Здібність ReceiveInput зчитує ввод з клавіатури та передає команди іншим здібностям та станам. Здібність Move відповідає за переключення стану з Idling на Moving при наявності вводу руху. Здібність Jump переключає стан з Idling або Moving на Jumping при наявності вводу стрибка. Здібність ControlCamera позиціонує камеру на гравці.

Таким чином ми змогли створити одночасно просту та модульну систему керування гравцем, використовуючи дрібні компоненти станів та здібностей, які можна легко розширювати, заміняти та підтримувати в ході розробки ігрового продукту.

1.2. Традиційні та процедурні методи комп’ютерної анімації

Традиційно, анімації ігрових персонажей створюються у вигляді кліпів. Кліп це файл або структура даних, де описується зміна координат, обертів та інших властивостей моделі і її компонентів з часом [4]. Під час виконання гри, анімаційний рушій послідовно зчитує ці дані та присвоює ці значення моделі, анімуючи її. Цей вид анімацій широко використовується в ігровій, фільмовій та інших індустріях, так

як він дає аніматорам абсолютний контроль над візуальним станом моделі під час анімації.

Основна проблема цього підходу саме з цього і випливає – створені анімації не мають гнучкості. Зміна одного параметру анімації часто означає необхідність ручного створення аніматором нової версії цієї анімації. Також, при її відтворенні важливо щоб початковий та кінцевий стан персонажа в точності відповідав очікуванням аніматора.

Сучасні відеоігри ж є дуже динамічними. Коли аніматор створює анімацію Ходьби (Walk Cycle, рис. 1.1) він розраховує що під гравцем є рівна, широка підлога. Якщо ж гравець в цей момент йде, наприклад, по тонкій балці, буде здаватись що він йде по повітрю. Це можна вирішити, створивши альтернативну анімацію – Ходьба з балансуванням. Але проблеми на цьому насправді не закінчаться. Що якщо гравець йде впоперек схилу? Одна нога буде ступати по повітрю, інша – проходити крізь землю.

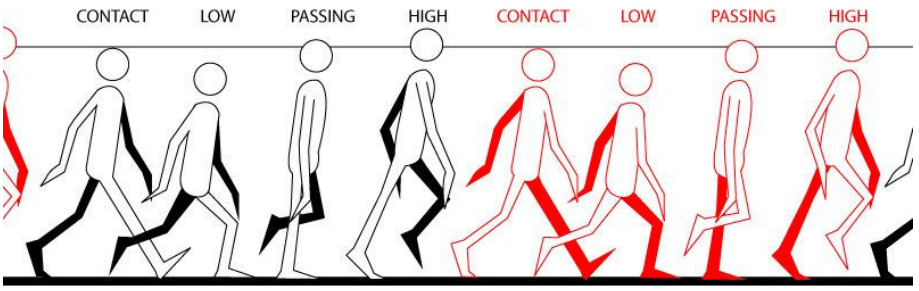


Рис. 1.1. Цикл ходьби

Створити всі можливі анімації для таких ситуацій буде фізично неможливо. Для більшості ігор це не є великою проблемою – гравці рідко приділяють увагу до такого роду подробиць. Але, це не означає що цю проблему не потрібно вирішувати. Сучасні ігри стають все більше і більше реалістичними. Розробники докладають багато зусиль щоб поглибити ефект “занурення” в гру. Тому створення нового виду анімацій – процедурних, динамічних – є популярним напрямом дослідження в сфері розробки ігор.

Одним із найпростіших елементів динамічних анімацій є Inverse Kinematics (IK) [5]. Система IK виконує обертання кісток скелету моделі, щоб задовольнити певну умову (solving, “вирішення”). Найчастіше ціллю вирішення ланцюга кісток є розміщення фінальної кістки якнайближче до позиції так званого “ефектора”.

В якості прикладу можна навести IK руки персонажа. Ланцюг кісток починається з кістки плеча, а закінчується кісткою кисті. Позиція ефектора такої IK системи буде визначати, де має розміщуватись кисть персонажа в 3D просторі. Такого роду IK системи зазвичай використовуються “поверх” стандартних анімацій. Тобто спочатку персонаж розташовується згідно поточного кадру анімації, а далі IK редагує фінальну позу, частково змінюючи її.

Також була розроблена інша технологія, що дозволяє покращити візуальну якість переходів між анімаціями – Blend Trees [6]. Blend Trees – це спеціальні структури даних, що описують можливі переходи між анімаціями у вигляді дерева (графа). В залежності від вхідних параметрів, визначених розробником і призначених через код, дерево обере необхідні анімації та проведе поступову інтерполяцію між ними, створивши плавний перехід. Таким чином можна створити переходи від, наприклад, анімації бігу до анімації ходьби.

Комбінація цих підходів є достатньою для багатьох ігор. Ми ж на цьому не зупинимось і спробуємо розробити повністю процедурну систему, з використанням IK, кривих та алгоритмів згладжування.

1.3. Паркур в відеоіграх

Паркур – мистецтво ефективного подолання перешкод – почав використовуватись в іграх ще в 80-х роках, коли з’явилися перші платформери, по типу Prince of Persia, де вертикальний рух, стрибки та акробатика вже стали одним з головних елементів геймплею. На глобальному рівні впровадження складних паркур-механік у відеоігри почалося із появою більш потужних технологій та переходом індустрії до 3D.

Популярними іграми, що впровадили такі механіки в повному обсязі є серії Assassin’s Creed та Mirror’s Edge. В обох іграх гравець може безперервно рухатися

дахами, стінами та вузькими вуличками. В Assassin’s Creed ця механіка працює на основі контекстних анімацій: при натисканні кнопок стрибку та напрямку руху персонаж автоматично підбирає оптимальний рух – стрибок, хват чи підйом (рис. 1.2) [7].

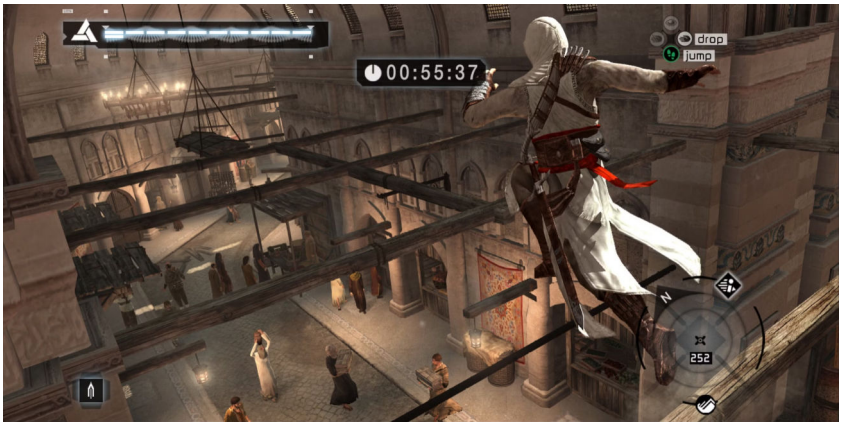


Рис. 1.2. Кадр з гри Assassin’s Creed

Це відрізняє Assassin’s Creed від більшості екшен-ігор того часу, де акробатика була обмежена заскриптованими ділянками рівня, та від більшості сучасних екшенів – таких як Mirror’s Edge, де гравець має більш повний контроль над напрямом та довжиною стрибків та інших рухів (рис. 1.3). Це робить паркур менш технічним, але більш доступним і плавним. Пізніші ігри серії почали більше схилятись до стилю RPG – вони мають великі відкриті світи та менше можливостей для вертикального руху. А нові ігри серії намагаються знайти баланс між свободою RPG-епохи та елегантністю класичних механік, повертаючи гравцям відчуття, що кожен рух має вагу і сенс.

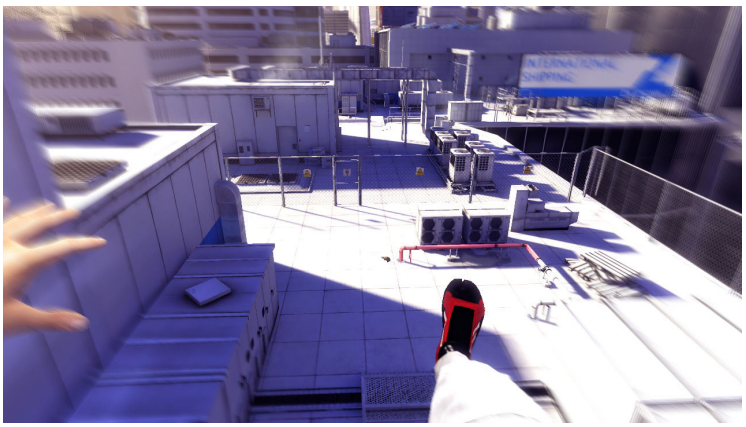


Рис. 1.3. Кадр з гри Mirror’s Edge

Деякі ігри мають інший підхід до паркуру. Наприклад система руху в The Legend of Zelda: Breath of the Wild широко використовує фізичний рушій та робить акцент на послідовності та свободі гравця [8]. Замість того щоб покладатися на заздалегідь визначені точки для лазіння або заскриптовані анімації, BOTW реалізує механіку “універсального лазання”: на майже будь-яку поверхню у світі можна залізти, а обмеженням виступає лише запас витривалості персона (рис. 1.4).

Рис. 1.4. Лазання в грі Legend of Zelda: BOTW

Той же самий запас витривалості використовується для бігу, плавання та планування. Рушій гри також враховує тертя, кут нахилу поверхні та погодні умови (наприклад, дощ робить стіни слизькими), що динамічно впливає на мобільність. З технічної точки зору, це перетворює пересування на своєрідну головоломку. Гравці, замість того, щоб слідувати заздалегідь прокладеним маршрутам, або покладатися на свою реакцію та точність вводу дій, взаємодіють безпосередньо з фізикою світу та слідкують за шкалою витривалості, вигадуючи власні рішення для дослідження світу.

1.4. Реалізація руху персонажа через фізику та через власний код

При розробці 2D та 3D ігор часто постає питання вибору способу переміщення гравця у просторі. В ігрових рушіях основними методами переміщення є переміщення через фізичний рушій та переміщення через власні обрахунки і пряму модифікацію просторових властивостей сутності [10].

Переміщення через фізику означає використання просторового тіла (Rigidbody, PhysicsBody тощо) і маніпуляція його станом через фізичні властивості (Velocity) або методи (AddForce, AddImpulse). Далі фізичний рушій проведе необхідні розрахунки та перемістить тіло у просторі. Великою перевагою такого методу є можливість додаткового використання інших частин фізичного рушія, наприклад колізії (обробка зіткнень з об’єктами, стінами тощо). Недоліком використання фізики є відсутність повного контролю над розташуванням персонажа у просторі, адже фізичний рушій в більшості випадків є мінливим та неточним у своїх розрахунках, і йому можуть бути притаманні ефекти “ковзання”. Деякі ігрові механіки, наприклад телепортація або сповільнення часу можуть працювати некоректно, адже фізичні рушії намагаються імітувати фізику реального світу і такі явища на даний час вважаються неможливими.

Альтернативним способом переміщення є переміщення через пряму модифікацію властивостей сутності. Якщо під час кожної ітерації ігрового циклу додавати певне зміщення до координатів гравця то створюється ілюзія його переміщення у просторі. Цей підхід дозволяє розробнику мати повний контроль над позицією гравця. Такі явища як крива розгону або дуга (вертикальна траєкторія) стрибка можуть бути скрупулезно налаштовані для ідеального ігрового досвіду (див. рисунки 1.5 та 1.6). Основним недоліком є необхідність створення деякого функціоналу “з нуля”, наприклад згаданих раніше колізій.

Рис 1.5. Приклади кривих розгону гравця

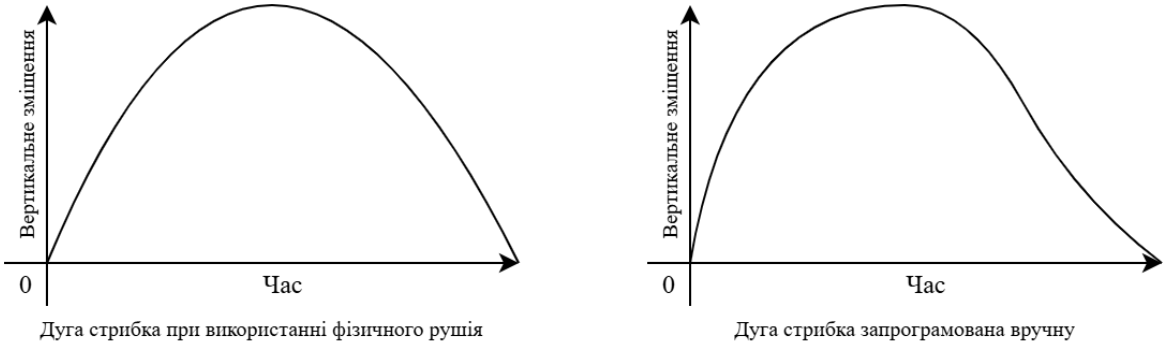


Рис 1.6. Приклади дуг стрибку гравця

Поєднавши цей аналіз з вимогами нашої гри, можна дійти висновку, що гібридний підхід буде найкращим. Гібридний підхід означає що ми використовуємо фізичні тіла, але в окремих випадках відключаємо їх фізичий функціонал та маніпулюємо їх властивостями напряму.

Стани, які представляють собою фіксовані в просторі анімації будуть використовувати просту логіку для переміщення гравця в кінцеву точку анімації, якщо така є. Якщо ж стан представляє собою рух у просторі (ходьба, стрибки, падіння, ковзання тощо) будуть використовувати функціонал фізичного рушія.

1.5. Динамічні колайдери

Оскільки було прийняте рішення використовувати фізичний рушій для переміщення гравця, наступним постає питання вибору форми його колізії. Колізією (collision) називають використання невидимих геометричних об’єктів (колайдерів) фізичним рушієм для обробки зіткнень [9]. Під цим розуміють унеможливлення проходження двох об’єктів крізь один одного, а також можливості сканування 3D-простору для визначення розташування об’єктів, їх форм та розмірів.

Використання колайдерів необхідне для забезпечення швидкодії рушія, адже математичне моделювання зіткнень двох примітивів (сфер, кубів, циліндрів тощо) є набагато ефективнішим за моделювання зіткнень повних 3D моделей з тисячами вершин та полігонів. Порівнявши рисунки 1.7. та 1.8. можна побачити різницю між зовнішньою формою об’єктів сцени, та можливою формою їх колізії, що використовується фізичним рушієм.

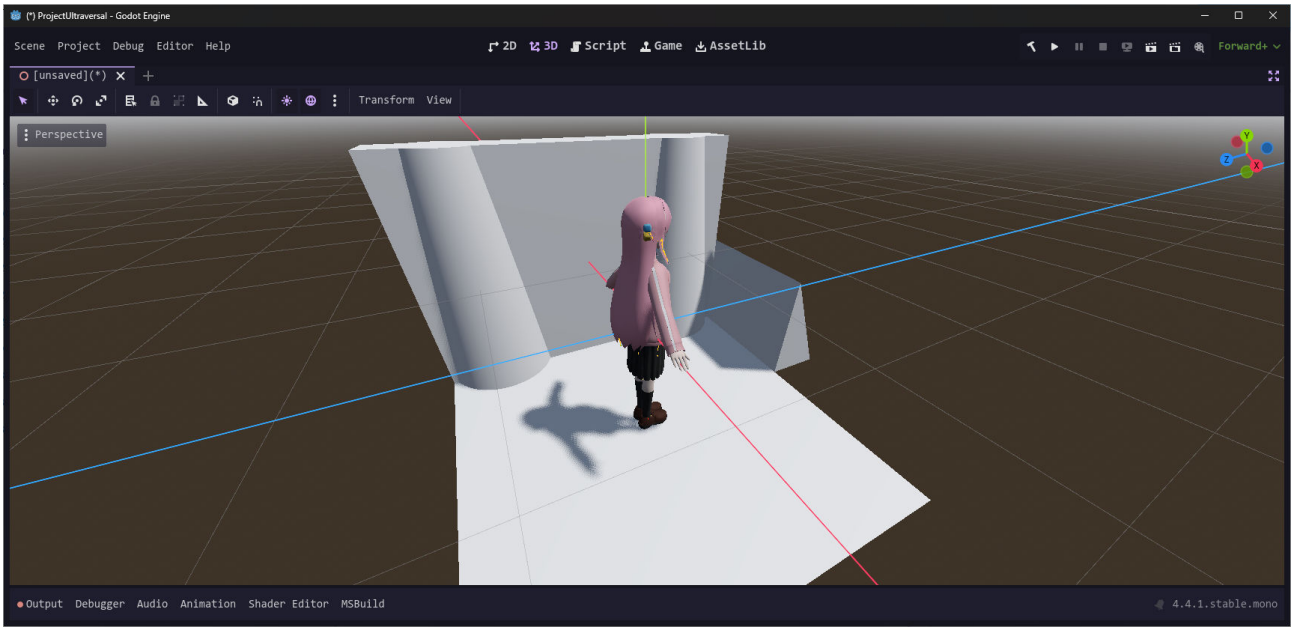


Рис. 1.7. Приклад ігрової сцени у редакторі Godot

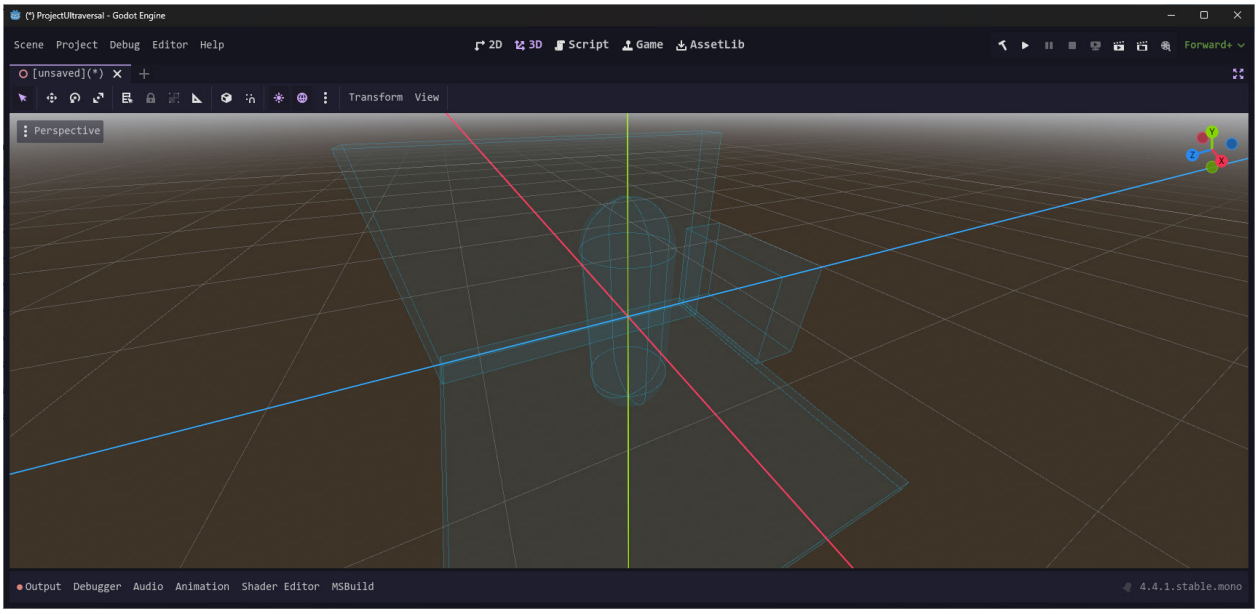


Рис. 1.8. Візуалізація колайдерів сцени

Вибір колайдера в більшості випадків не є складною задачею, адже його невідповідність моделі часто є непомітною. Якщо колайдер повністю охоплює 3D-простір моделі та не виступає занадто далеко від неї, гравцю цього буде достатньо для обходу перешкоди, а витрачати час зіштовхуюсь з нею та перевіряючи точну відповідність кожного сантиметру гравці не будуть.

Втім, існують випадки, коли одна геометрична фігура не може реалістично використовуватись для всіх можливих станів гравця (рис. 1.9 - 1.11).

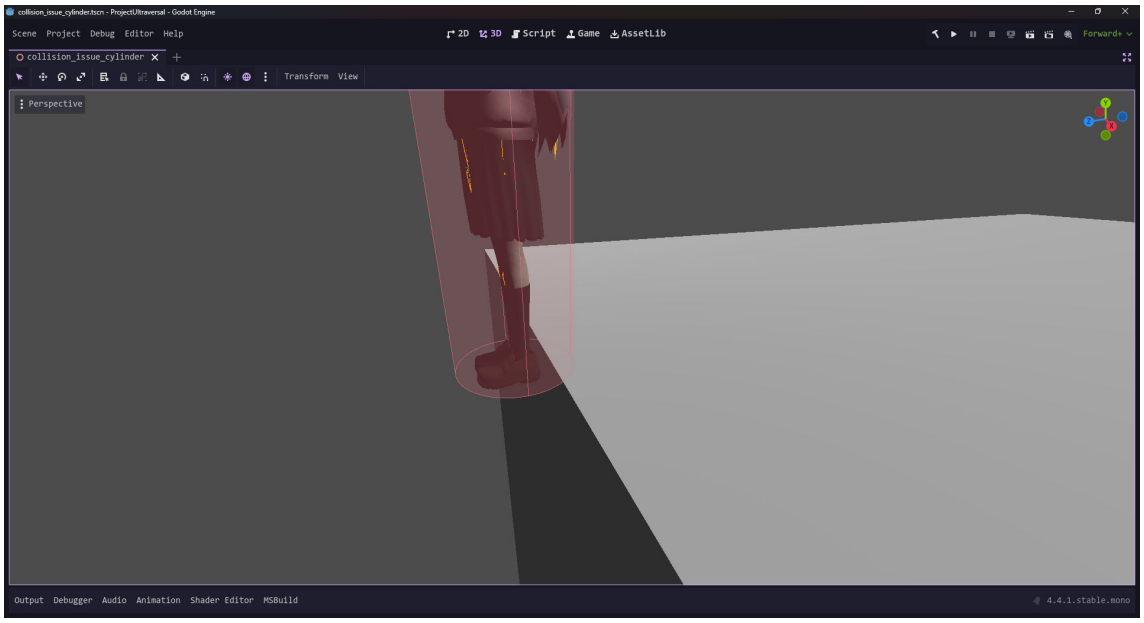


Рис. 1.9. Візуалізація недоліку циліндричного колайдера гравця

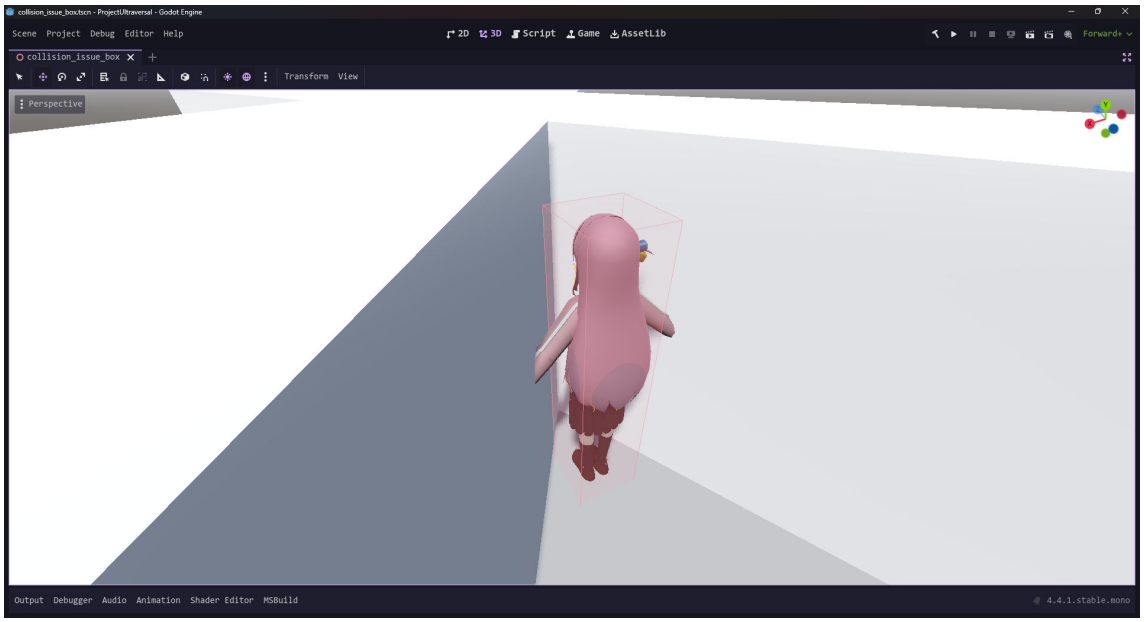


Рис. 1.10. Візуалізація недоліку кубічного колайдера гравця

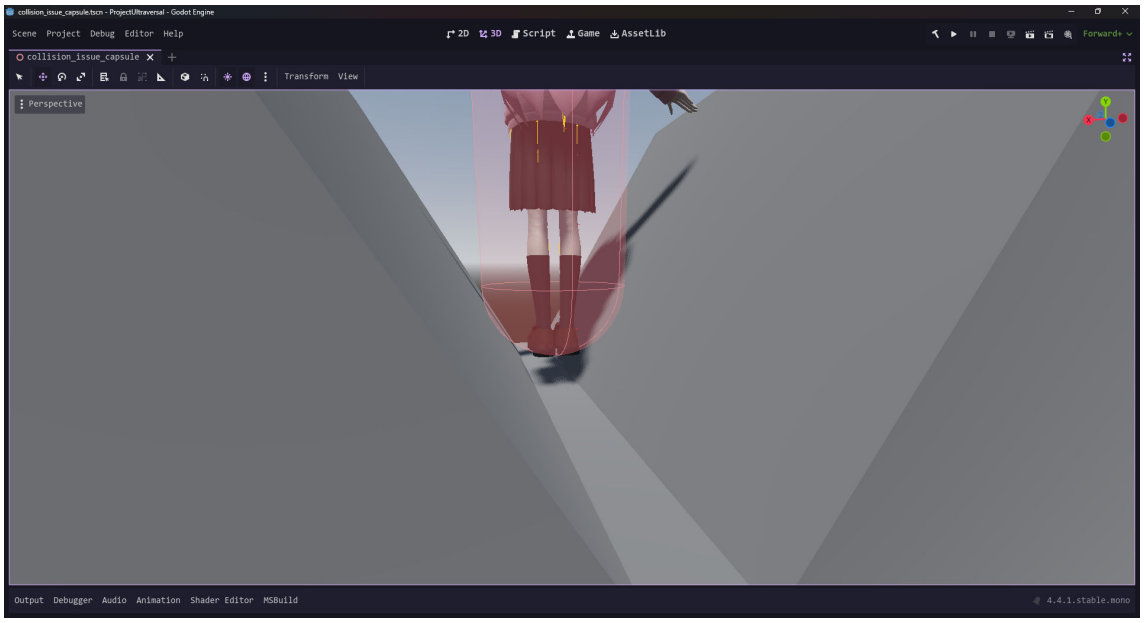


Рис. 1.11. Візуалізація недоліку капсульного колайдера гравця

Виходячи з цього було прийнято рішення розробити динамічну систему, що дозволяє змінювати поточний колайдер в залежності від стану гравця.

Висновок

В першому розділі був проведений аналіз деяких нюансів та проблем, з якими зустрічаються розробники при створенні комп'ютерних ігор. Було проаналізовано:

1. Способи реалізації складних програмних модулів через патерни State та Component.
2. Сучасні підходи до комп'ютерної анімації, в тому числі її використання в комп'ютерних іграх.
3. Роль дисципліни паркуру в відеоіграх.
4. Можливі підходи до реалізації руху гравця в відеоіграх.
5. Проблеми використання простих фіксованих форм колізії в 3D-іграх.

В результаті цього було виявлено декілька напрямків дослідження, що будуть детально розглянуті в наступних розділах роботи.

Першим напрямом є дизайн та розробка гнучкого фреймворку контролера персонажа, з подальшою реалізацією конкретного функціоналу у контексті

паркур-гри. Ключовою властивістю розробленого контролера повинна бути чиста, модульна архітектура, що забезпечить легку підтримку та розширення функціоналу гравця. Він також має підтримувати можливість налаштування динамічних колізій.

Другим напрямком є створення повноцінної системи анімації ігрового персонажа через код, з підтримкою технік ІК. Система має забезпечити анімацію по таким категоріям:

1. Вид анімації:
    a. згладження (інтерполяція) до кінцевого значення;
    b. контроль значень через код;
    c. анімація в часі на основі кривих.
2. Ціль анімації:
    a. тулуб моделі (основа та хребет);
    b. голова моделі (оберт шиї та голови);
    c. кінцівки моделі (ІК рук та ніг).

Додатковим напрямом дослідження є система повтору та пов’язані з нею інструменти відладки. Розроблена система дозволить переглядати та аналізувати стан гри за певний проміжок часу.

РОЗДІЛ 2
АРХІТЕКТУРА ТА РОЗРОБКА ІГРОВИХ СИСТЕМ

2.1. Вибір ігрового рушія

Вибір ігрового рушія є одним з перших питань яке має вирішити розробник, або команда розробників ігрового продукту. Від цього вибору будуть залежати такі ключові моменти як мова програмування проекту, вартість та ліцензії користування рушієм, наявні бібліотеки, а також інтернет-ресурси, необхідні для розробки (документація, форуми та спільноти, посібники тощо).

На даний момент, найбільш досконалими та широко використовуваними ігровими рушіями, відкритими для всіх розробників, є Unity, Unreal Engine та Godot [11].

Unity – кросплатформовий ігровий рушій, створений у 2005 році компанією Unity Technologies. Він швидко став популярним завдяки зручному інтерфейсу, великій кількості інструментів для 2D і 3D-розробки та активній спільноті. Unity використовує мову програмування C# і пропонує площадку Asset Store для придбання готових моделей, скриптів і ресурсів. Unity це комерційний продукт: є безкоштовна версія для невеликих студій, а для більших компаній – платні підписки.

Unreal Engine – рушій, розроблений компанією Epic Games, перша версія якого вийшла в 1998 році разом із грою Unreal. Сьогодні він відомий своїм потужним графічним ядром, яке широко застосовують у великих AAA-проєктах, архітектурній візуалізації та фільмовій індустрії. Основна мова програмування – C++, а також є система візуального програмування Blueprints. Unreal Engine розповсюджується безкоштовно, але після досягнення певного доходу з гри починається стягнення комісії.

Godot Engine – почав розроблятися Хуаном Лінецьким у 2007 році, а з 2014 року став доступним як проект з відкритим кодом (MIT-ліцензія). Він відзначається легкістю, модульністю та орієнтацією на інді-розробників. Godot підтримує 2D і 3D, власну мову програмування GDScript (схожу на Python), а також C# і C++. Завдяки своїй відкритості, безкоштовності та прості використання, Godot привертає все більше уваги як майбутній провідний рушій для маленьких та середніх ігрових

компаній. Особливим поштовхом стала серія контроверсійних змін в ToS та бізнес-моделі Unity в 2023 році, через що велика кількість його користувачів стала задумуватись над переходом до інших рушіїв.

З огляду на це, для реалізації цього проекту було обрано саме Godot, адже розвиток FOSS є важливим елементом сучасного пейзажу розробки ігор та ПЗ.

2.2. Структура проекту

Проект було створено на основі ігрового рушія Godot, найновішої на даний момент версії 4. Для реалізації проекту було обрано мову програмування C# (.NET 8). Для контролю версій використовується система Git. Репозиторій з вихідним кодом доступний за посиланням: https://github.com/Metal-666/ProjectUltraversal.

Файлова структура проекту має наступний вигляд:

```
res://
├── addons/ – встановлені розширення
│   ├── godot_entity_controller/ – розроблений фреймворк контролера гравця
│   ├── imgui-godot/ – бібліотека рендерингу інтерфейсу
│   ├── linear_stairs/ – розширення генерації сходів
│   └── proto-csgs/ – розширення з геометричними фігурами для прототипування
├── Data/ – ресурси (ігрові дані)
│   └── Player/ – ресурси гравця
│       ├── DriverLayers/ – драйверні шари
│       ├── GaitAnimations/ – анімації ходьби та бігу
│       └── MovementConfigurations/ – налаштування ходьби та бігу
├── Models/ – 3D-моделі
│   └── Player/ – 3D-моделі, текстури та анімації моделі гравця
├── Scenes/ – файли сцени гри
│   ├── Debugging/ – неігрові сцени для відладки
│   ├── Player/ – сцени гравця, гравця повтору тощо
│   └── World/ – ігровий світ
│       ├── Areas/ – локації ігрового світу
│       ├── Interactables/ – інтерактивні об’єкти світу
│       └── Objects/ – базові та неінтерактивні об’єкти світу
└── Scripts/ – код проєкту
    ├── Player/ – код гравця
    │   ├── Abilities/ – компоненти здібностей
    │   │   └── Mixins/ – домішки здібностей
```

2.3. Контролер персонажа (гравця)

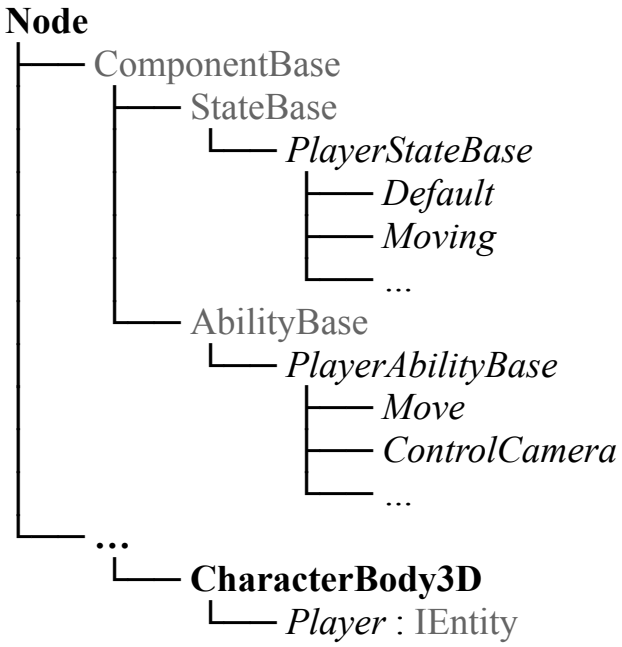
Запропонована в цій роботі архітектура контролера гравця використовує комбінацію паттернів State та Component. Ідея патерну State заключається в передачі контролю над об’єктом (в нашому випадку – гравцем) спеціальному об’єкту стану. При необхідності, об’єкт стану може бути замінений на інший, змінюючи логіку контролера. Цей паттерн добре підходить для представлення можливих станів гравця – Default, Moving, Jumping, Falling тощо, проте використовуючи його напряму постає проблема: куди помістити логіку, що має працювати незалежно від поточного стану, або лише в окремих станах?

2.3.1. Компоненти, стани та здібності

Для вирішення цієї проблеми пропонується замість використання станів напряму, використати патерн Component, і зробити Стан одним з підвидів компоненту. Тоді для виконання логіки незалежно від стану можна використати інший підвид компоненту, наприклад Здібність.

Для реалізації цього було створено поняття “активованості” компонента. Коли компонент є активованим, він присутній в дереві сцени і виконує свою логіку. Активованість Здібностей та Станів розроблена відповідно до запропонованої архітектури. Здібності можуть бути індивідуально активовані та деактивовані у будь який момент. Натомість у випадку Станів, лише один стан може бути одночасно активований.

Ієрархія класів розробленого рішення матиме наступний вигляд:



Базовим класом для всіх розроблених класів є Node – це вбудований клас Godot, що представляє собою елемент дерева сцени без ніякого особливого функціоналу. ComponentBase це базовий клас-компонент. Він містить всю внутрішню логіку керування компонентами, тобто ініціалізація, переключення станів та здібностей тощо. StateBase та AbilityBase це його підкласи, що містять більш вузьку логіку саме для станів та здібностей. Також наявний порожній інтерфейс IEntity, що використовується для позначення класу-сутності.

Станам та Здібностям доступні методи та властивості, створені в клас ComponentBase:

- CurrentState
- EnabledAbilities та DisabledAbilities
- Can<TAbility>()

- SwitchState<TState>() та QueueStateSwitch<TState>()
- ToggleAbility<TAbility>(), EnableAbility<TAbility>() та DisableAbility<TAbility>()

Використовуючи ці властивості та методи, конкретні класи станів та здібностей можуть оцінювати поточний стан контролера (наприклад визначати який Стан є поточним та які Здібності є активованими) та змінювати його (переключати Стан на інший, вмикати або вимикати Здібності).

Для створення самого контролера, спочатку необхідно унаслідувати три класи: базовий клас-стан, базовий клас-здібність та клас сутності. Оскільки в цій роботі розглядається в першу чергу контролер гравця, було відповідно створено класи PlayerStateBase, PlayerAbilityBase та Player. Конкретні класи станів та здібностей розглянуто детально в Розділі 3.

Структура сцени, що містить контролер, модель та інші об’єкти гравця, зображена на рисунку 2.1.

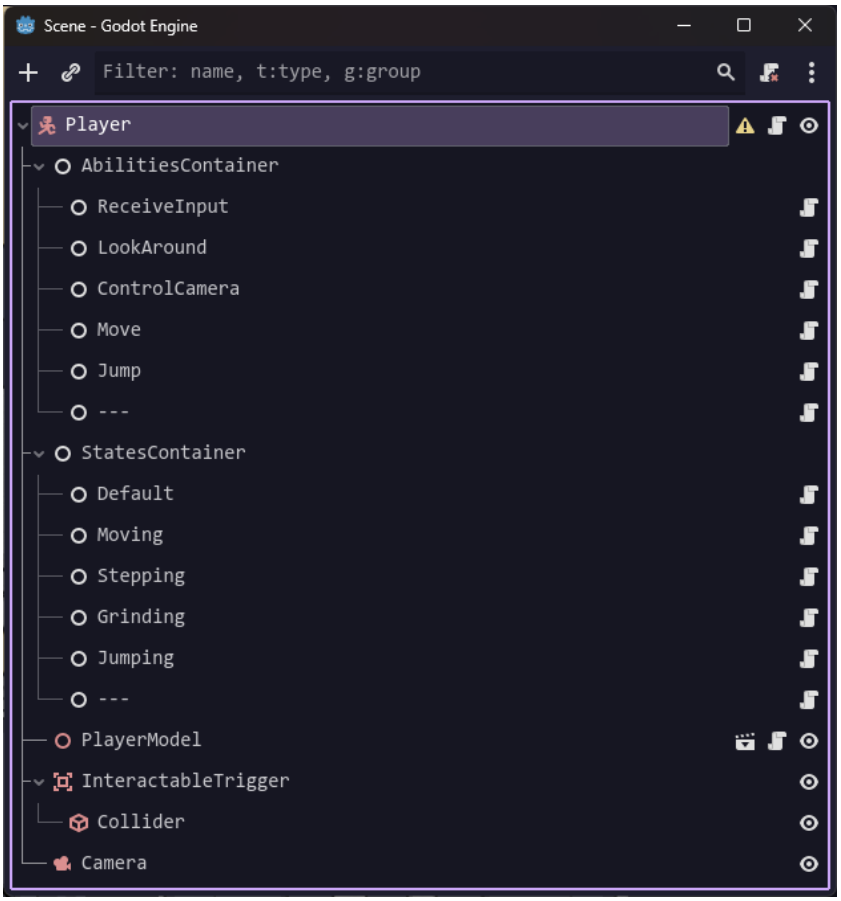


Рис. 2.1. Структура сцени гравця

2.3.2. Домішки (Mixins)

Як і компоненти, домішки є засобом для інтеграції спільної логіки (функцій, обробників подій) у класи програмного рішення без використання наслідування. В нашій архітектурі вони використовуються для реалізації спільної логіки між компонентами станів та здібностей.

Домішки було реалізовано за допомогою інтерфейсів та відображення (Reflection). Спочатку було створено базовий інтерфейс IMixin, який домішки повинні унаслідувати, щоб система їх розпізнала.

```
public interface IMixin<TEntity> where TEntity : Node, IEntity {
    TEntity Entity { get; }
    ComponentBase<TEntity> This => (ComponentBase<TEntity>) this;
}
```

В конструкторі базового компонента ComponentBase перевіряється, чи імплементує даний клас якісь інтерфейси, основані на IMixin. Якщо так, ці інтерфейси скануються за допомогою відображення та в них знаходяться методи-“хуки”. Знайдені хуки додаються в масив, і подалі викликаються при ключових подіях.

Для перетворення простих методів у хуки було створено атрибут MixinHookAttribute, енумерацію HookType та HookEvent.

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false, Inherited = true)]
public class MixinHookAttribute(HookEvent HookEvent, HookType HookType =
HookType.Post) : Attribute {
    public virtual HookEvent HookEvent { get; } = HookEvent;
    public virtual HookType HookType { get; } = HookType;
}
```

```
public enum HookType {
    Pre = -1,
    Post = 1
}
```

```
public enum HookEvent {
    Ready,
```

```
    Process,
    PhysicsProcess,
    Enter,
    Exit,
    Enabled,
    Disabled
}
```

Позначивши метод атрибутом MixinHook з необхідними полями HookEvent та HookType, компонент буде знати, коли цей метод-хук повинен викликатись. В базовому компоненті ComponentBase це було реалізовано за допомогою проксі методів-подій, наприклад:

```
public override void _PhysicsProcess(double delta) {
    base._PhysicsProcess(delta);
    _InvokeHooks(HookEvent.PhysicsProcess, HookType.Pre);
    OnPhysicsProcess(delta);
    _InvokeHooks(HookEvent.PhysicsProcess, HookType.Post);
}

protected virtual void OnPhysicsProcess(double delta) { }
```

Таким чином можна зареєструвати виклик методу-хуку до та після виклику відповідної події (в цьому прикладі – фізичного кадру) у самому компоненті.

2.3.3. Реалізація динамічних колізій

Спершу було створено класи ресурсів PlayerColliderSet та PlayerCollider, що містять дані про колізію гравця. Клас PlayerCollider містить властивості індивідуального колайдера, що може бути присвоєний гравцю, а саме:

- Shape3D Shape – об’єкт фігури колайдеру.
- Slot Slot – необов’язкове значення з енумерації Slot, що дозволяє прив’язати колайдер до частини тіла моделі персонажа (слоту), унаслідуючи її трансформацію.
- Vector3 Offset – зсув колайдера відносно трансформації Slot або відносно координат гравця, якщо слот не встановлено.

- Vector3 Rotation – оберт колайдера відносно трансформації Slot або відносно початкового оберту, якщо слот не встановлено.
- Vector3 Scale – масштаб колайдера.

Цей клас також містить методи для створення колайдеру на основі цих даних та присвоєння його гравцю за допомогою API фізичного рушія Godot.

```
public partial class PlayerCollider : Resource {
#nullable disable
    [Export] public virtual Shape3D Shape { get; set; }
    [Export] public virtual Slot Slot { get; set; }
    [Export] public virtual Vector3 Offset { get; set; }
    [Export(PropertyHint.Range, "-360,360,1,radians_as_degrees")]
    public virtual Vector3 Rotation { get; set; }
    [Export] public virtual Vector3 Scale { get; set; } = Vector3.One;
#nullable enable
    public virtual uint OwnerId { get; protected set; }
    public virtual Player? Player { get; protected set; }
    public virtual Transform3D LastColliderTransform { get; protected set; }

    public virtual void Register(Player player) {
        if(player != Player && IsInstanceValid(Player)) {
            Player.RemoveShapeOwner(OwnerId);
        }
        Player = player;
        OwnerId = Player.CreateShapeOwner(null);
        Player.ShapeOwnerAddShape(OwnerId, Shape);
    }
    public virtual void Update(Dictionary<Slot, Transform3D> slots) {
        if(!IsInstanceValid(Player)) {
            return;
        }
        if(!slots.TryGetValue(Slot, out Transform3D modelSlotTransform)){
            return;
        }
        Transform3D localTransform =
            new(Basis.FromEuler(Rotation).Scaled(Scale), Offset);
        Player.ShapeOwnerSetTransform(OwnerId, LastColliderTransform =
modelSlotTransform * localTransform);
    }
    public virtual void Enable() => Player?.ShapeOwnerSetDisabled(OwnerId, false);
```

```
    public virtual void Disable() => Player?.ShapeOwnerSetDisabled(OwnerId, true);
}
```

Клас ресурсу PlayerColliderSet містить масив Colliders ресурсів PlayerCollider, тобто він є колекцією індивідуальних колайдерів, що може бути присвоєна певним станам. Він містить методи Register, Update, Enable та Disable що виконують вказані операції на всіх колайдерах в масиві Colliders.

```
public partial class PlayerColliderSet : Resource {
#nullable disable
    [Export] public virtual PlayerCollider[] Colliders { get; set; }
#nullable enable

    public virtual void Register(Player player) {
        foreach(PlayerCollider playerCollider in Colliders) {
            playerCollider.Register(player);
        }
    }
    public virtual void Update(Dictionary<Slot, Transform3D> slots) {
        foreach(PlayerCollider playerCollider in Colliders) {
            playerCollider.Update(slots);
        }
    }
    public virtual void Enable() {
        foreach(PlayerCollider playerCollider in Colliders) {
            playerCollider.Enable();
        }
    }
    public virtual void Disable() {
        foreach(PlayerCollider playerCollider in Colliders) {
            playerCollider.Disable();
        }
    }
}
```

Для інтеграції цих даних в стани, в базовому стані PlayerStateBase було створено властивість ColliderSet, в яку присвоюється необхідний набір колайдерів для цього стану. Під час події OnReady (стан ініціалізовано) викликаються методи Register та Disable на об’єкті ColliderSet, що в свою чергу ініціалізує та вимикає всі індивідуальні об’єкт-колайдери. Також додається обробник до події SkeletonUpdated

моделі гравця, що в свою чергу викликає метод Update на об’єктів ColliderSet, передаючи йому оновлені трансформації слотів.

Під час подій Enter та Exit відповідно викликаються методи Enable та Disable на ColliderSet, таким чином активуючи та деактивуючи колайдери під час зміни стану.

```
protected override void OnReady() {
    base.OnReady();
    ColliderSet?.Register(Entity);
    ColliderSet?.Disable();
    Entity.Model.Skeleton.SkeletonUpdated += OnSkeletonUpdated;
}
protected override void Enter() {
    base.Enter();
    ColliderSet?.Enable();
}
protected override void Exit() {
    base.Exit();
    ColliderSet?.Disable();
}
protected virtual void OnSkeletonUpdated() =>
    ColliderSet?.Update(Entity.Model.Animators
                               .Select(modelAnimator => modelAnimator.SlotTransforms)
                               .Aggregate((slotTransforms1, slotTransforms2) =>
slotTransforms1.Concat(slotTransforms2).ToDictionary())
                               .Append(new(Slot.None, Entity.GlobalTransform))
                               .ToDictionary(pair => pair.Key, pair =>
Entity.GlobalTransform.AffineInverse() * pair.Value));
```

Приклад конфігурації стану в редакторі, виконаної за допомогою такої системи, наведено на рисунку 2.2.

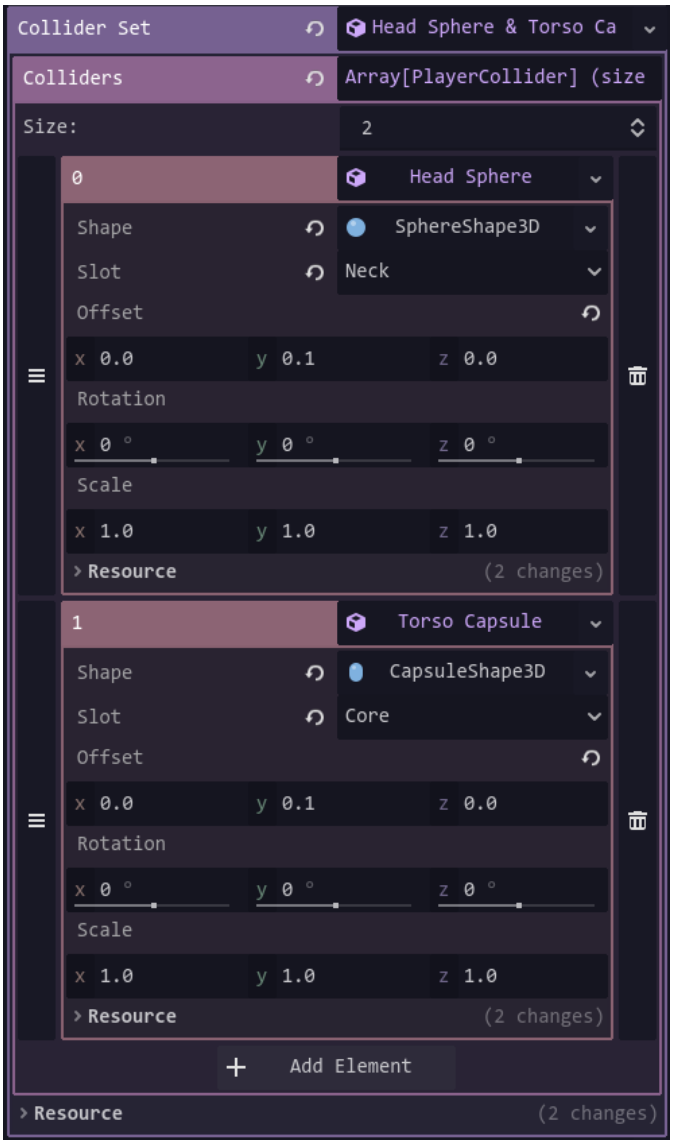


Рис 2.2. Приклад налаштування колізії стану

Властивості Colliders об’єкту PlayerColliderSet було присвоєно 2 об’єкти PlayerCollider. Перший описує сферу, прив’язану до голови персонажа, другий – капсулу, прив’язану до тулуба персонажа. Візуалізація цих колайдерів в запущеній грі продемонстрована на рисунку 2.3.

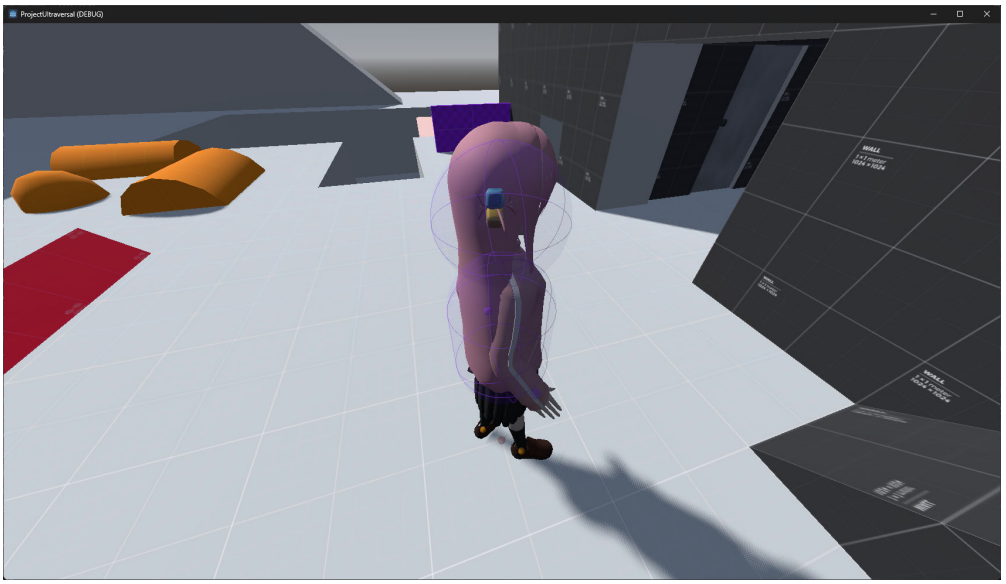


Рис. 2.3. Візуалізація колайдерів в запущеній грі

2.4. 3D-модель гравця

При анімації 3D-моделей в комп’ютерній графіці використовуються поняття “скелета” та “кісток”. Кістками називають невидимі стрижні, розміщені всередині моделі. Кожен стрижень має координати початку та кінця, та може бути приєднаним до іншого стрижня, унаслідуючи його трансформацію в просторі (розташування, оберт, масштаб) та створюючи ієрархію. Повна ієрархія кісток моделі і називається скелетом (рис. 2.4.). Далі, за допомогою засобів 3D-моделювання, частини моделі прив’язуються до кісток таким чином, що рухаючи (а саме обертаючи) кістки можна змінювати положення, наприклад, кінцівок 3D персонажа.

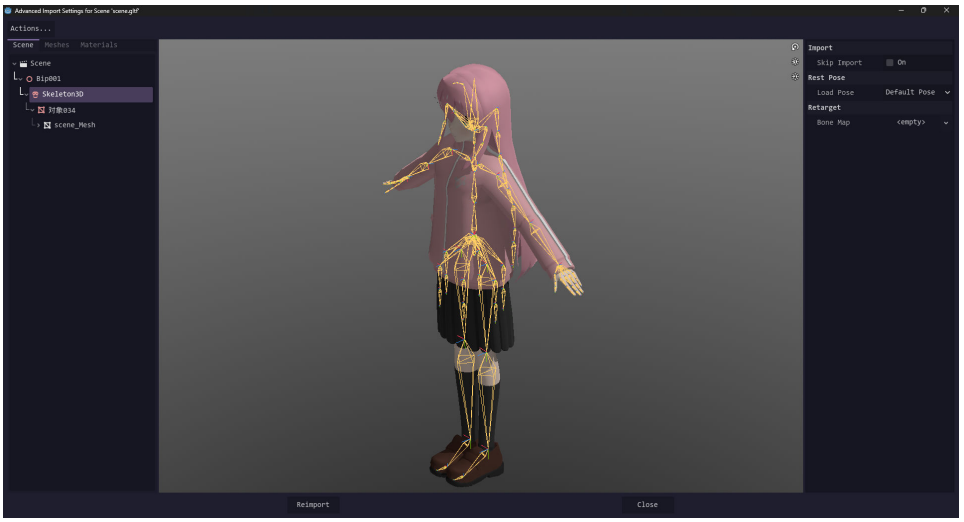


Рис 2.4. Візуалізація кісток скелету моделі

Виходячи з цього, задачею процедурної анімації є розробка алгоритмів, а також API для ефективного керування моделлю.

Для вирішення цієї проблеми було прийнято рішення розділити код керування моделлю на окремі підкласи: HeadAnimator, LimbAnimator та TorsoAnimator. Ці класи наслідують клас ModelAnimatorBase, що в свою чергу наслідує SkeletonModifier3D – вбудований клас Godot призначений для маніпуляції частинами скелету.

2.4.1. TorsoAnimator

TorsoAnimator відповідає за контроль тулуба персонажа, а саме кісток тазу (Core, коренева кістка скелету) та хребта (LowerSpine, MiddleSpine, UpperSpine). Управління ними відбувається за допомогою набору параметрів, частина з яких встановлюється завчасно в редакторі, а частина є публічним API. Всі параметри є числами з рухомою комою (float). Оскільки коренева кістка Core є батьківською для всіх інших кісток, її зміщення та оберт є аналогічним до трансформації всієї моделі.

Параметри публічного API:

1. CoreXOffset – горизонтальний зсув моделі (вліво-вправо).
2. CoreYOffset – вертикальний зсув моделі.
3. CoreZOffset – горизонтальний зсув моделі (вперед-назад).
4. CoreYaw – оберт моделі навколо вертикальної осі Y (також впливає на параметри Core*Offset).
5. CoreTilt – нахил моделі вперед-назад (навколо осі X).
6. CoreLean – нахил моделі вліво-вправо (навколо осі Z).
7. SpineTwist – комбінований оберт кісток хребта навколо осі Y.
8. SpineCurl – комбінований оберт кісток хребта навколо осі X.

Параметри конфігурації:

1. DefaultCoreYOffset – стале значення, що додається до всіх модифікацій CoreYOffset.

2. TwistLowerSpineInfluence, TwistMiddleSpineInfluence, TwistUpperSpineInfluence – множники впливу SpineTwist на конкретні кістки хребта.
3. CurlLowerSpineInfluence, CurlMiddleSpineInfluence, CurlUpperSpineInfluence – множники впливу SpineCurl на конкретні кістки хребта.

При розрахунку стану тулуба виконується наступний алгоритм:

1. Розраховується кватерніон оберту тулуба навколо осі Y (yRotation), де кут оберту – значення CoreYaw.
2. Розраховується повний кватерніон оберту тулуба методом обертання кватерніону нахила Lean кватерніоном нахилу Tilt і обертання результату кватерніоном yRotation. Кути Lean та Tilt це значення CoreLean та CoreTilt.
3. Розраховується повний зсув тулуба комбінуючи зсуви по трьох осях (CoreXOffset, CoreYOffset + DefaultCoreYOffset, CoreZOffset) у вектор та обертаючи його кватерніоном yRotation.
4. Розраховується повний відносний оберт кожної кістки хребта шляхом оберту кватерніона Twist кватерніоном Curl, де кути Twist та Curl це значення SpineTwist та SpineCurl помножені на відповідні для кісток коефіцієнти Twist*SpineInfluence та Curl*SpineInfluence.

Демонстрація поз, які можна створити на основі наведених восьми параметрів, зображена на рисунку 2.5.

Всі параметри за замовчуванням

- CoreYOffset: -0.1
- CoreYaw: 45°
- CoreLean: -20°

- CoreYOffset: -0.05
- CoreZOffset: -0.15
- SpineTwist: 45°
- SpineCurl: -60°

- CoreYOffset: -0.3
- CoreTilt: 30°
- SpineTwist: -45°

Рис 2.5. Демонстрація різних поз моделі

2.4.2. HeadAnimator

HeadAnimator відповідає за контроль головою та шиєю персонажа. Для цього використовується параметр LookAngle – двовимірний вектор, де перша компонента – кут оберту погляду навколо осі Y (вліво-вправо), а друга – кут оберта погляду навколо осі X (вверх-вниз). Також HeadAnimator містить параметр Camera. Якщо до нього передати об’єкт камери, HeadAnimator буде автоматично оновлювати його трансформацію, таким чином синхронізуючі перспективу від першої особи з реальною позицією голови (очей) гравця.

Головним параметром публічного API є LookAngle – кути погляду по осях X та Y. Параметрами конфігурації є LookYAngleHeadInfluenceCurve та LookYAngleNeckInfluenceCurve – криві, що описують характер залежності обертів голови та шиї навколо осі X.

Оскільки комбінований оберт здійснюється по двох осях, важливо визначити порядок окремих операцій оберту: першим має здійснюватись оберт навколо осі X (тангаж); після цього – навколо осі Y (рискання). Це дозволяє уникнути появи додаткового нахилу крену, що є дезорієнтуючим в перспективі від першої особи.

З цієї ж причини оберт по осі Y здійснюється лише для шиї. А навколо осі X було розроблено алгоритм обертання і шиї, і голови. Для цього розраховується різниця між тангажем погляду і тулуба, та використовуються параметри LookYAngleHeadInfluenceCurve та LookYAngleNeckInfluenceCurve (рис. 2.6).

Рис. 2.6. Криві конфігурації HeadAnimator

Якщо різниця дорівнює 0, то 0 відсотків кута тангажу передається голові, а 100 відсотків – шиї. Якщо гравець дивиться, наприклад, прямо вгору, то 50 відсотків тангажу передається шиї (45°), а 30 відсотків – голові. Це забезпечує плавний розподіл оберту між шиєю та головою, також уберігаючи візуальний стан від неприродніх кутів нахилу.

Повний алгоритм проводиться наступним чином:

1. Розраховується різниця lookYAngleToBendDifference між кутом нахилу тулуба та кут оберта погляду навколо осі X.
2. Розраховується відносний оберт шиї:
    a. Розраховується кут оберту навколо осі X. Для цього друга компонента LookAngle конвертується в межі [-180°; 180°] та множиться на значення кривої LookYAngleNeckInfluenceCurve в точці lookYAngleToBendDifference.
    b. Створюється кватерніон оберту навколо осі Y, де кут оберту – перша компонента LookAngle.
    c. Кватерніон оберту навколо осі X обертається кватерніоном оберту навколо осі Y; результатом є повний оберт шиї.
3. Аналогічно розраховується відносний оберт голови навколо осі Y; навколо осі X оберт не проводиться.

Демонстрація можливих обертів голови та шиї наведена на рисунку 2.7.

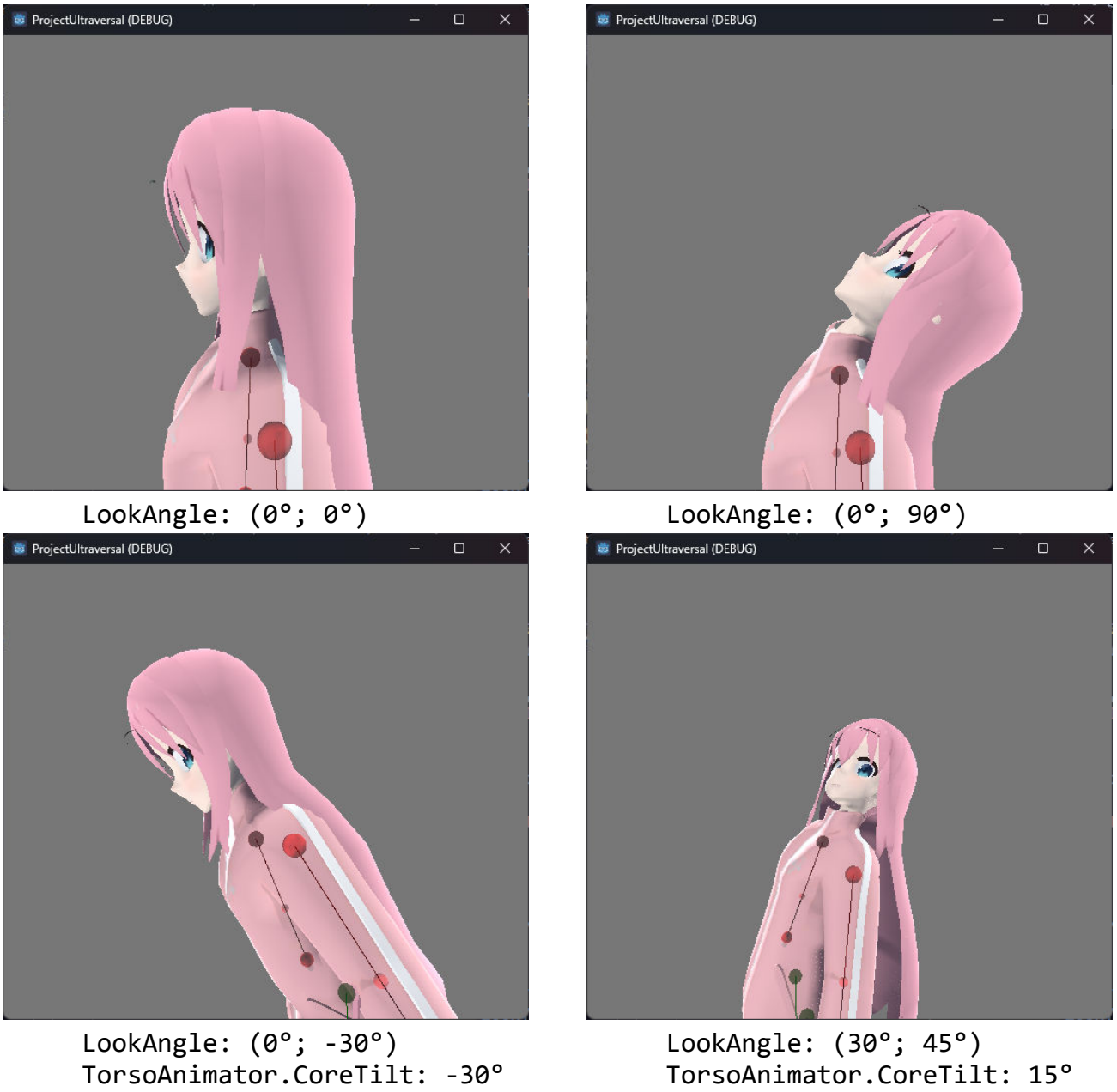


Рис 2.7. Демонстрація можливих обертів голови

2.4.3. LimbAnimator

LimbAnimator відповідає за контроль кінцівок моделі персонажа. Керування кінцівками здійснюється по принципу інверсної кінематики – знаючи положення кінцевої кістки скелету моделі, алгоритм повертає всі кістки в ланцюгу кінцівки так, щоб кінцева кістка співпала з цільовим положенням.

На відміну від існуючих алгоритмів IK, що повинні підтримувати ланцюги кісток будь-якої довжини, наш алгоритм був спеціально розроблений для ланцюгів з

трьома кістками, адже це відповідає спрощеній структурі кінцівок людини, яка використовується в 3D-анімації.

Розроблений алгоритм враховує два нюанси оберту суставів:

1. Другий сустав ланцюга (лікоть, коліно) може обертатись лише навколо однієї осі.
2. Оберт третього суставу (рука, ступня) може здійснюватись в двох режимах, глобальному та локальному. В глобальному режимі оберт суставу встановлюється вручну логікою стану. В локальному режимі основний оберт наслідується від другої кістки ланцюга, а встановлене значення оберту використовується як додатковий оберт в локальному просторі.

Алгоритм складається з трьох стадій:

1. Підготовка та обрахунок початкових значень.
2. Побудова трикутника в 3D просторі, вершини якого є координатами трьох суглобів ланцюга.
3. Розрахунок обертів кісток, у відповідності до знайдених координатів ланцюга у 3D просторі та особливостей фізіології людських кінцівок.

Визначені стадії дозволяють розбити задачу IK на легші для вирішення підпроблеми, а також досягти балансу між потребами ігрової логіки та складністю реальної фізіології людських кінцівок. Алгоритм буде пояснено взявши за основу кінцівку руки.

Перша стадія

Під час першої стадії обраховуються сторони трикутника та вектори, що формують ланцюг руки. Довжини плечової та ліктьової кісток (бічних сторін трикутника) обраховуються на основі координат суглобів скелету у нейтральній позі. Основа трикутника (“вектор кінцівки”) формується вектором, що поєднує суглоб плеча з цільовою позицією, даною алгоритму. Довжина відрізка кінцівки

обмежується максимальної довжиною витягнутої кінцівки та мінімальною довжиною, що дозволяє побудувати трикутник з даних сторін.

Друга стадія

Наступним етапом є побудова трикутника в 3D просторі, виходячи з вектора кінцівки та довжин двох кісток руки. Побудова трикутника зводиться до проблеми знаходження координат ліктя. Спочатку знаходиться висота трикутника:

$$p = \frac{len_{upperArm} + len_{forearm} + v_{limb}}{2}$$

$$S = \sqrt{p \times (p - len_{upperArm}) \times (p - len_{forearm}) \times (p - v_{limb})}$$

$$h = \frac{2 \times S}{v_{limb}}$$

Далі знаходиться позиція основи висоти трикутника:

$$weight = \frac{\sqrt{len_{upperArm}^{\ 2} - h^2}}{v_{limb}}$$

$$pos_{elbowBase} = lerp(pos_{shoulder}, pos_{hand}, weight)$$

Далі знаходиться вектор висоти трикутника v_{elbow}, тобто лінія, що поєднує знайдену основу та позицію ліктя:

1) Знаходиться кватерніон Q_{limb}, що обертає вектор “вниз” (0, -1, 0) у вектор v_{limb}.
2) Знаходиться кватерніон $Q_{elbowSpin}$, що обертає навколо вектору v_{limb} на задану алгоритму кількість градусів jointSpinAngle.
3) Знаходиться кватерніон $Q_{rootTwist}$, що обертає навколо вектору “вниз” на CoreYaw градусів.
4) Знаходиться вектор v', що дорівнює вектору згину кінцівки $v_{limbBend}$ (задається окремо для кожної кінцівки) оберненому оберненим кватерніоном $Q_{rootTwist}$.
5) Знаходиться вектор v'', що дорівнює вектору v', оберненому кватерніоном Q_{limb}.

6) Знаходиться вектор v''', що дорівнює вектору v'', оберненому кватерніоном $Q_{elbowSpin}$.
7) Знаходиться вектор v_{elbow}, що дорівнює вектору v''', помноженому на скаляр h.

Після цього позиція ліктя $pos_{elbow} = pos_{elbowBase} + v_{elbow}$. Знаючи позиції ліктя, плеча та руки, ми можемо знайти вектори кісток руки $v_{upperArm}$ та $v_{forearm}$. Візуалізація координат та векторів другого етапу наведена на рисунку 2.9.

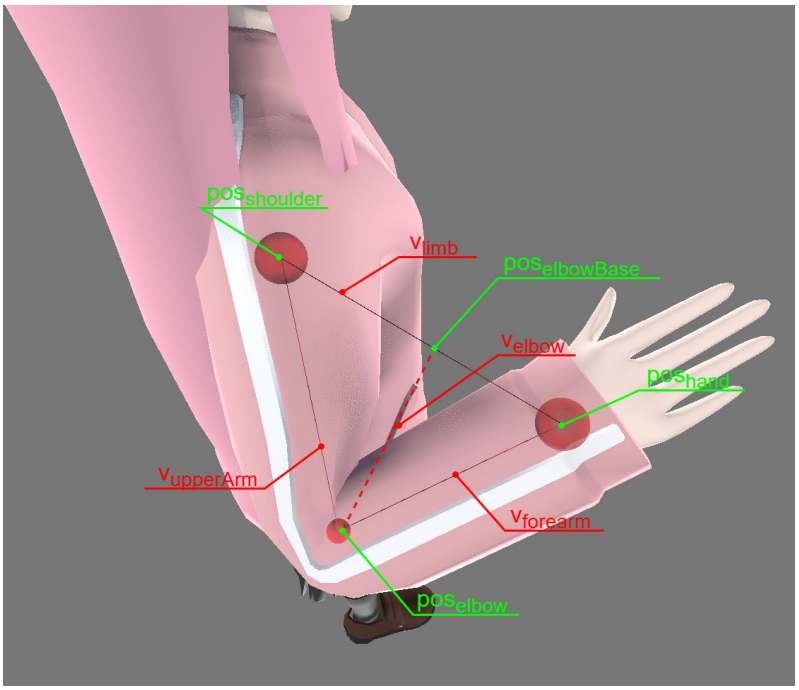


Рис. 2.8. Координати та вектори другого етапу алгоритму

Третя стадія

Третім етапом алгоритму є обрахунок обертів кісток ланцюга. Він зводиться до поступового розрахунку векторів та кватерніонів обертів, з їх подальшою комбінацією:

1. Розраховується крен $upperBoneRoll$ плечової кістки. Це оберт навколо осі $rollAxis$ від вектору “вгору” до відкидання вектора $upperArm$ з віссю $rollAxis$. Результатом першого кроку є вектор $upperBoneAfterRoll$ що дорівнює вектору “вгору” оберненому кватерніоном $Q_{upperBoneRoll}$ (рис. 2.9).

2. Знаходиться вісь $swingAxis$ шляхом оберту початкової осі $swingAxis_i$ кватерніоном $Q_{upperBoneRoll}$. Визначається кут оберту $upperBoneSwing$ навколо цієї осі між векторами $upperBoneAfterRoll$ та $upperBoneVector$. Розраховується кватерніон $Q_{upperBoneSwingRoll}$ шляхом множення кватерніону $Q_{upperBoneSwing}$ на кватерніон $Q_{upperBoneRoll}$ (рис. 2.10).

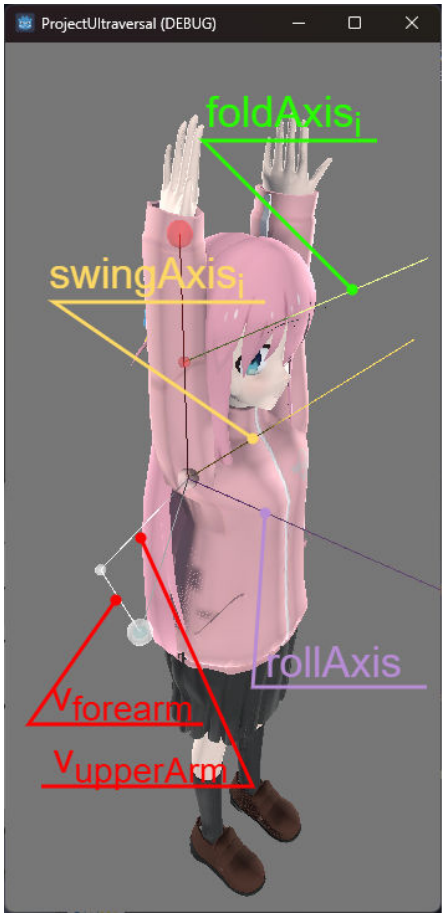


Рис. 2.9. Перший крок третього етапу

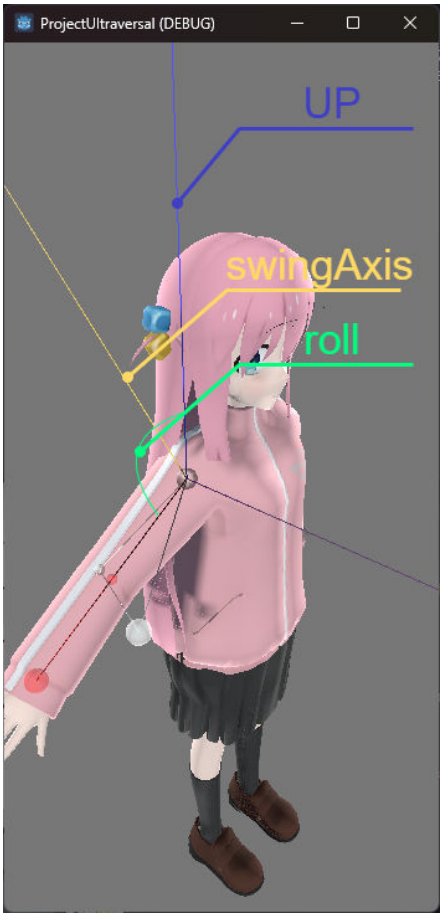


Рис. 2.10. Другий крок третього етапу

3. Розраховується вектор $twistReferenceVector$ шляхом оберту вектора “вперед” кватерніоном $Q_{upperBoneSwingRoll}$. Знаходиться кут оберту $twistAngle$ між відкиданням вектору $twistReferenceVector$ від $upperBoneVector$ та відкиданням вектору $lowerBoneVector$ від $upperBoneVector$. Кватерніон $Q_{upperBoneTwist}$ визначається як оберт навколо вектору “вгору” на кут $twistAngle$. Це дозволяє обчислити повний оберт плечової кістки

$Q_{upperBoneSwingRollTwist}$ шляхом множення кватерніону $Q_{upperBoneSwing}$ на $Q_{upperBoneRoll}$ та на $Q_{upperBoneTwist}$ (рис. 2.11).

4. Знаходиться вісь оберту ліктьової кістки $foldAxis$ шляхом оберту осі $foldAxis_i$ кватерніоном $Q_{upperBoneSwingRollTwist}$. Після цього можна знайти кут оберту кістки між векторами $upperBoneVector$ та $lowerBoneVector$ навколо знайденої осі. Це дозволяє обчислити повний оберт ліктьової кісти в просторі, що дорівнює добутку кватерніонів $Q_{lowerBoneFold}$ та $Q_{upperBoneSwingRollTwist}$ (рис. 2.12).

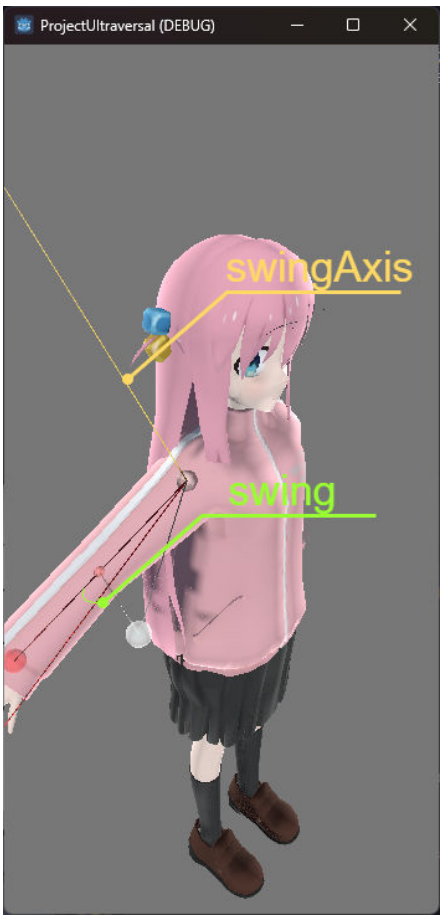


Рис. 2.11. Третій крок третього етапу

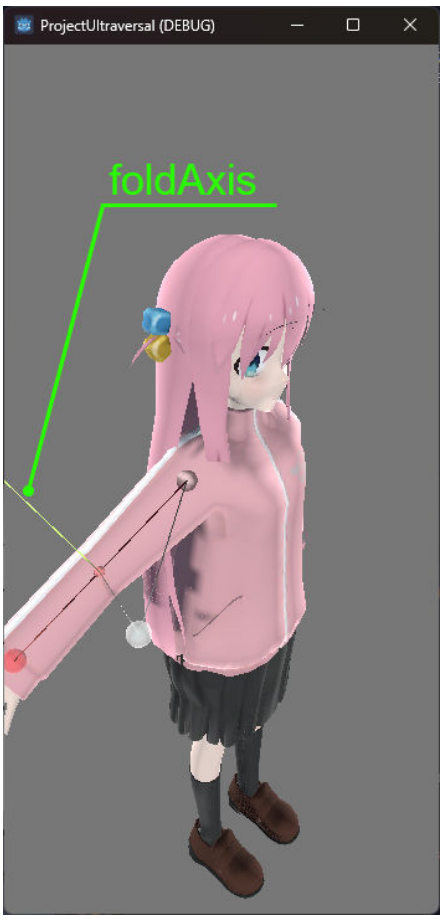


Рис. 2.12. Четвертий крок третього етапу

Початкові осі задані наступним чином:

1. $rollAxis$ – вектор “вперед” (мінус Z).
2. $swingAxis_i$ – вектор “вліво” (мінус X).
3. $foldAxis_i$ – вектор “вліво” (мінус X).

Рис. 2.13. Кінцевий результат виконання алгоритму

2.5. Система процедурної анімації

Для реалізації процедурних анімацій було створено домішку IAnimator та додано напряму до PlayerStateBase. Це дозволяє виконувати логіку анімації після всіх інших обчислень стану (Post PhysicsProcess), а також анімувати модель гравця напряму, додавши домішку до інших класів та вручну викликавши необхідні функції, адже домішка також є інтерфейсом. Це дозволяє створювати спеціальні сцени відладки, що напряму анімують модель.

Для створення анімацій була розроблена система драйверів (Drivers) та драйверних шарів (Driver Layers).

2.5.1. Drivers

Драйвер це клас-ресурс, що відповідає за анімацію однієї властивості моделі, наприклад числового значення повороту голови або векторного значення зміщення руки. Ключовим методом драйверу є метод Drive.

```
public abstract T Drive(T current, double delta);
```

Почергово викликавши цей метод на кожному драйвері для кожної властивості моделі, IAnimator отримує новий набір значень що передаються відповідним компонентам PlayerModel.

Також у драйвера є властивість DefaultTarget, що дозволяє встановлювати цільове значення через код. В залежності від реалізації драйвера це цільове значення може бути використано напряму (повернуто з функції Drive), або може бути додатково оброблене, враховуючи поточний стан цієї властивості.

Було розроблено декілька видів реалізації драйверу, представлених інтерфейсами IFunctionChainDriver, IFixedTimeInterpolationDriver та ITransformerDriver.

IFunctionChainDriver проводить обробку властивості моделей за допомогою ланцюга функцій. Кожна функція є об’єктом, що містить єдиний метод Execute.

```
public abstract T Execute(T target, T current, double delta);
```

Перша функція в ланцюгу отримує DefaultTarget як цільове значення target, а також значення current та delta передані драйверу. Після виконання, функція повертає значення, що стає новим ціловим значенням, яке далі передається до наступної функції. Після виконання всіх функцій цільове значення повертається драйвером як фінальний результат.

Функції було розроблено для двох типів драйверів: float (Single) та Vector3. Деякі функції можливо реалізувати з використанням generic параметрів, спрощуючи ієрархію, проте ігровий рушій Godot такі об’єкти не підтримує, отже було реалізовано окремі версії. Далі наведено реалізовані функції для типу Single.

1. Функція константи.

```
[GlobalClass]
public partial class SingleConstantFunction : SingleFunctionBase {
#nullable disable
    [Export] public virtual float Value { get; set; }
#nullable enable
```

```
    public override float Execute(float target, float current, double delta) =>
        Value;
}
```

Повертає стале значення Value, встановлене в редакторі, ігноруючи значення target, current та delta.

2. Функція згладжування.

```
[GlobalClass]
public partial class SingleDampingFunction : SingleFunctionBase {
#nullable disable
    [Export] public virtual float Smoothness { get; set; } = 20;
#nullable enable

    public override float Execute(float target, float current, double delta) =>
        current.Damp(target, Smoothness, delta);
}
```

Ця функція проводить інтерполяцію від поточного значення до цільового, враховуючи параметри згладжування та дельти часу. Інтерполяція виконується за алгоритмом, розробленим Фреєю Холмер:

$a' = lerp(a, b, 1 - 2^{-\Delta t/\lambda})$, де:

a – поточне значення

b – цільове значення

Δt – дельта часу

λ – коефіцієнт згладжування

3. Функція кривої (за часом).

```
[GlobalClass]
public partial class SingleTimeCurveFunction : SingleFunctionBase,
ITimeCurveFunction {
#nullable disable
    [Export] public virtual Curve ValueOverTime { get; set; }
#nullable enable

    public virtual float Progress { get; set; }
```

```
    public override float Execute(float target, float current, double delta) =>
        ValueOverTime.Sample(Progress);
}
```

Схожа на функцію константи, ця функція повертає значення, взяте з кривої що описує зміну значення з часом.

4. Функція перетворення в радіани.

```
[GlobalClass]
public partial class SingleToRadFunction : SingleFunctionBase {
    public override float Execute(float target, float current, double delta) =>
        target.ToRad();
}
```

Ця функція перетворює цільове значення в радіани.

Драйвер IFixedTimeInterpolationDriver використовується для плавного переходу до нового значення. Замість конкретного значення або ланцюга функцій, цей драйвер містить об’єкт вкладеного драйверу. Він також містить властивість Duration, що встановлюється в редакторі та вказує, за який проміжок часу драйвер повинен повністю “синхронізуватись” зі вкладеним драйвером.

```
public interface IFixedTimeInterpolationDriver<T> where T : struct {
    float Duration { get; }
    float TimeLeft { get; set; }

    DriverBase<T> GetInnerDriver();
    T LerpValue(T current, T target, float weight);
    T DriveInnerWithInterpolation(T current, double delta) {
        T result = LerpValue(current, GetInnerDriver().Drive(current, delta),
                        1 - (TimeLeft / Duration));
        TimeLeft = TimeLeft < 0 ? 0 : TimeLeft - (float) delta;
        return result;
    }
    void ResetTimeInterpolation() =>
        TimeLeft = Duration;
}
```

Під час виклику функції Driver, він викликає цю функцію на вкладеному драйвері, і потім проводить лінійну інтерполяцію від поточного значення до цільового. Інтерполяція стає сильнішою з плином часу, і коли спливає більше часу ніж Duration, драйвер починає напряму повертати значення вкладеного драйверу.

ITransformerDriver використовує вкладений драйвер для отримання значення, і далі проводить його обробку.

```
public interface ITransformerDriver { }
public interface ITransformerDriver<T> : ITransformerDriver where T : struct {
    DriverBase<T> GetInnerDriver();
    T TransformInput(T input);
    T TransformOutput(T output);
    T DriveInner(T current, double delta) =>
        TransformOutput(GetInnerDriver().Drive(TransformInput(current), delta));
}
```

Це дозволяє використовувати існуючі алгоритми, але з певними змінами. Одним з головних сценаріїв використання є переведення поточного значення в локальний простір, його обробка внутрішнім драйвером, і подальше переведення назад в глобальний простір. Конкретним прикладом такого драйверу є Vector3LocalSpaceTransformer.

```
[GlobalClass]
public partial class Vector3LocalSpaceTransformer : Vector3TransformerDriverBase {
#nullable disable
    [Export] public virtual bool CancelOutMovement { get; set; } = true;
#nullable enable

    public virtual Transform3D GlobalTransform { get; set; }
    protected virtual Transform3D? PreviousGlobalTransform { get; set; }

    public override Vector3 TransformInput(Vector3 input) =>
        ((PreviousGlobalTransform.HasValue && CancelOutMovement) ?
            PreviousGlobalTransform.Value :
            GlobalTransform).AffineInverse() * input;
    public override Vector3 TransformOutput(Vector3 output) {
```

```
        Vector3 result = GlobalTransform * output;
        PreviousGlobalTransform = GlobalTransform;
        return result;
    }
    public override void Reset() {
        base.Reset();
        PreviousGlobalTransform = null;
    }
}
```

Цей драйвер використовує присвоєний йому параметр GlobalTransform щоб конвертувати поточне значення вектору позиції у локальний простір моделі (вхідна трансформація), далі обробити його внутрішнім драйвером та конвертувати назад у глобальну координату (вихідна трансформація).

Це необхідно для створення анімацій, адже аніматор не може знати завчасно, де у світі знаходиться гравець, тому анімацій робляться відносно його початкової трансформації, у вигляді зсувів. Цей драйвер дозволяє конвертувати ці зсуви в реальні координати, відносно гравця в просторі.

Додатково драйвер також запам’ятовує попереднє значення глобальної трансформації. При встановленні значення true властивості CancelOutMovement, він буде використовувати його при вхідній трансформації, таким чином виключаючи з алгоритму фактор переміщення гравця у просторі. Це важливо для анімацій кінцівок, адже вони завжди рухаються разом з тілом гравця, і такі операції як згладжування рухів, не повинні враховувати переміщення гравця у світі.

2.5.2. Driver Layers

Для ефективного присвоєння драйверів станам та реалізації комплексних анімацій було створено систему “драйверних шарів” – Driver Layers. Драйверний шар це ресурс, що містить повний набір драйверів для всіх властивостей моделі.

Базовою класом для драйверних шарів є DriverLayerBase, що структурований наступним чином:

```
[GlobalClass]
public abstract partial class DriverLayerBase : Resource {
```

```
    public virtual DriverBase<float> CoreXOffset =>
        GetCoreXOffsetOverride() ?? SingleDriver.DampToZero;
    public virtual DriverBase<float> CoreYOffset =>
        GetCoreYOffsetOverride() ?? SingleDriver.DampToZero;

    ...

    public abstract DriverBase<float>? GetCoreXOffsetOverride();
    public abstract DriverBase<float>? GetCoreYOffsetOverride();
    ...
}
```

Для кожної властивості моделі було створено відповідний абстрактний метод перевизначення драйверу та властивість-getter. Getter є частиною публічного API та використовується домішкою IAnimator для отримання об'єктів драйверів. Стани також можуть змінювати параметри цих властивостей, наприклад для керування значеннями напряму. Метод перевизначення драйверу повинен бути мати конкретну реалізацію у класах-шарах. Він є частиною внутрішнього API драйверних шарів.

Конкретними реалізаціями DriverLayerBase є DriverLayer, BinaryChoiceDriverLayer, DriverLayerSet та ToggleableDriverLayer.

DriverLayer це найпростіша реалізація, в якій кожна властивість моделі може бути напряму присвоєна через редактор, наприклад:

```
[GlobalClass]
public partial class DriverLayer : DriverLayerBase {
#nullable disable
    [ExportGroup("Torso")]
    [Export] public virtual SingleDriverBase CoreXOffsetOverride { get; set; }
    [Export]  public virtual SingleDriverBase CoreYOffsetOverride { get; set; }
    ...
    public override DriverBase<float>? GetCoreXOffsetOverride() =>
        CoreXOffsetOverride;
    public override DriverBase<float>? GetCoreYOffsetOverride() =>
        CoreYOffsetOverride;
    ...
}
```

BinaryChoiceDriverLayer містить в собі два об’єкта драйверних шарів, та булеву змінну, що визначає, який з них є активним.

```
[GlobalClass]
public partial class BinaryChoiceDriverLayer : DriverLayerBase {
#nullable disable
    [Export] public virtual DriverLayerBase LayerA { get; set; }
    [Export] public virtual DriverLayerBase LayerB { get; set; }
#nullable enable

    public virtual bool SelectedLayer { get; set; }

    public override DriverBase<float>? GetCoreXOffsetOverride() =>
        (SelectedLayer ? LayerB : LayerA).GetCoreXOffsetOverride();
    public override DriverBase<float>? GetCoreYOffsetOverride() =>
        (SelectedLayer ? LayerB : LayerA).GetCoreYOffsetOverride();

    ...
}
```

DriverLayerSet містить масив драйверних шарів. При отриманні драйверу властивості він ітерує по кожному шару і повертає перший ненульовий драйвер. Ітерація проводиться в зворотньому напрямку, починаючи з найнижчого в списку шару.

```
[GlobalClass]
public partial class DriverLayerSet : DriverLayerBase {
#nullable disable
    [Export] public virtual DriverLayerBase[] Layers { get; set; }
#nullable enable

    protected virtual IEnumerable<DriverLayerBase> ReversedLayers =>
        (Layers as IEnumerable<DriverLayerBase>).Reverse();

    public override DriverBase<float>? GetCoreXOffsetOverride() =>
        ReversedLayers.FirstOrDefault(layer => layer.GetCoreXOffsetOverride() !=
null)?.GetCoreXOffsetOverride();
    public override DriverBase<float>? GetCoreYOffsetOverride() =>
```

```
        ReversedLayers.FirstOrDefault(layer => layer.GetCoreYOffsetOverride() !=
null)?.GetCoreYOffsetOverride();

    ...
}
```

ToggleableDriverLayer наслідує DriverLayer, з додатковою можливістю “виключити” його функціонал за допомогою булевої властивості.

```
[GlobalClass]
public partial class ToggleableDriverLayer : DriverLayer {
    public virtual bool IsEnabled { get; set; } = true;

    public override DriverBase<float>? GetCoreXOffsetOverride() =>
        IsEnabled ? CoreXOffsetOverride : null;
    public override DriverBase<float>? GetCoreYOffsetOverride() =>
        IsEnabled ? CoreYOffsetOverride : null;

    ...
}
```

2.6. Система повтору (replay)

Система повтору означає можливість збереження та відтворення подій гри за певний період часу. За допомогою такої системи:

- Розробники можуть:
    1. Проводити відладку
    2. Запрограмовувати анімації NPC
- Гравці можуть:
    1. Ділитись моментами зі своєї гри з іншими гравцями
    2. Створювати контент для соціальних платформ (монтажі, кліпи тощо)
    3. Відправляти баг-репорти розробникам

Ключовими елементами системи повтору є дані реплею, публічний API для збору цих даних та власне код що цим керує.

2.6.1. Дані

Основним контейнером для даних повтору є клас Replay. Він містить в собі колекцію об’єктів ReplayFrame. Кожен об’єкт ReplayFrame містить снапшот (snapshot) стану гри в конкретний момент часу (кадр). Оскільки система повинна підтримувати різні сценарії використання, було прийнято рішення зберігати окремі категорії даних в різних підоб’єктах.

Всі класи-категорії, а також сам клас ReplayFrame містять методи для ініціалізації об’єкту на основі поточних даних гри, такі як Snapshot, CreateFrom та Create.

```
[MessagePackObject]
public class ReplayFrame {
    [Key(0)] public virtual required ModelData? ModelData { get; init; }
    [Key(1)] public virtual required PlayerData? PlayerData { get; init; }
    [Key(2)] public virtual required GameData? GameData { get; init; }

    public static ReplayFrame Snapshot() => CreateFrom(Instance);
    public static ReplayFrame CreateFrom(Player.Player player) =>
        new() {
            ModelData = ModelData.CreateFrom(player),
            PlayerData = PlayerData.CreateFrom(player),
            GameData = GameData.Create()
        };
}
```

Критичним для відтворення реплею є клас ModelData, що містить мінімальний набір властивостей, які дозволяють відтворити візуальний стан сутності та спостерігати за її діями від першого та третього лиця:

1. Vector3 Position – координати розташування сутності в просторі.
2. Dictionary<int, Transform3D> BoneTransforms – трансформація кожної кістки скелета.
3. Transform3D CameraTransform – трансформація камери.
4. float CameraFov – поле зору камери.

Інші категорії даних будуть розглянуті в пункті про інструменти відладки.

2.6.2. Відтворення та керування

Для відтворення реплеїв були створені класи ReplayController, ReplayState та ReplayPlayer.

ReplayState це клас-обгортка навколо Replay, що зберігає дані поточного стану реплею. Він містить наступні поля:

1. Replay Replay – контейнер даних реплею.
2. Mode Mode – енумерація режиму реплею: Recording (запис) або Playback (відтворення).
3. State State – енумерація стану реплею: Active (відтворюється/записується) або Paused (призупинено).
4. int CurrentFrame – при відтворенні реплею вказує, який кадр має наразі бути відображений.
5. TimeSpan CurrentSpan – довжина (тривалість) реплею від початку до CurrentFrame.
6. int Length – повна довжина реплею.
7. TimeSpan LengthSpan – повна тривалість реплею.

Цей клас також містить методи для зчитування та запису даних реплею:

1. PushFrame – вставляє новий кадр в кінець реплею.
2. ReadFrame – повертає кадр за індексом CurrentFrame.
3. Advance – якщо кінця реплея ще не досягнуто, збільшує CurrentFrame на 1 та повертає true; в іншому випадку повертає false.
4. Seek – встановлює значення CurrentFrame.
5. Create – ініціалізує новий екземпляр ReplayState з пустим реплеєм та заданим режимом.
6. From – ініціалізує новий екземпляр ReplayState з заданим реплеєм та режимом.

Клас ReplayPlayer, та відповідний йому елемент в ігровому світі (реплейний гравець) представляє собою контейнер для моделі персонажа, без додаткових скриптів справжньої сутності гравця.

ReplayController це елемент сцени, що відповідає за візуалізацію реплею, на основі даних реплею. Він містить поля CurrentReplayState, в якому зберігається об’єкт класу ReplayState для керування поточним повтором.

```
public override void _PhysicsProcess(double delta) {
    base._PhysicsProcess(delta);
    if(CurrentReplayState == null || CurrentReplayState.State != State.Active) {
        return;
    }
    switch(CurrentReplayState.Mode) {
        case Mode.Recording: {
            Invoke.Deferred(RecordFrame, 999);
            break;
        }
        case Mode.Playback: {
            PlayFrame();
            if(!CurrentReplayState.Advance()) {
                CurrentReplayState.State = State.Paused;
                CurrentReplayState.Seek(0);
                break;
            }
            break;
        }
    }
}
```

Під час кожного ігрового кадру, ReplayController виконує набір дій в залежності від стану повтору. Якщо об’єкт стану повтору відсутній, або не є Active, ReplayController в цьому кадрі нічого не робить. В іншому випадку він діє в залежності від режиму повтору.

В режимі Recording він реєструє відкладений запис кадру, тобто виконання кадру після завершення стадії Process та перед стадією малювання кадру. Під час запису кадру використовуються раніше описані методи ReplayFrame.Snapshot та ReplayState.PushFrame.

```
protected virtual void RecordFrame() =>
    CurrentReplayState?.PushFrame(ReplayFrame.Snapshot());
```

В режимі Playback виконується відтворення кадру реплею, та просування вперед на один кадр в стані реплею. Для відтворення кадру створюється новий, або береться існуючий об’єкт реплейного гравця в світі, і йому присвоюються значення з кадру.

```
protected virtual void PlayFrame() {
    if(CurrentReplayState == null) {
        return;
    }
    ReplayFrame? currentReplayFrame = CurrentReplayState.ReadFrame();
    CurrentReplayPlayer ??= ReplayPlayerScene.Instantiate<ReplayPlayer>()
                                .AddTo(ReplayPlayersContainer);
    ReplayPlayer player = CurrentReplayPlayer;
    ModelData? modelData = currentReplayFrame?.ModelData;
    if(modelData == null) {
        return;
    }
    player.GlobalPosition = modelData.Position;
    player.ModelStateWriter.BoneTransforms = modelData.BoneTransforms;
}
```

2.6.3. Інтерфейс

Після запуску відтворення повтору, відображається його інтерфейс (рис. 2.14). Інтерфейс контролюється скриптом ReplayControls, що під час кожного кадру перевіряє наявність активного повтору, та відповідно показує або ховає елементи керування.

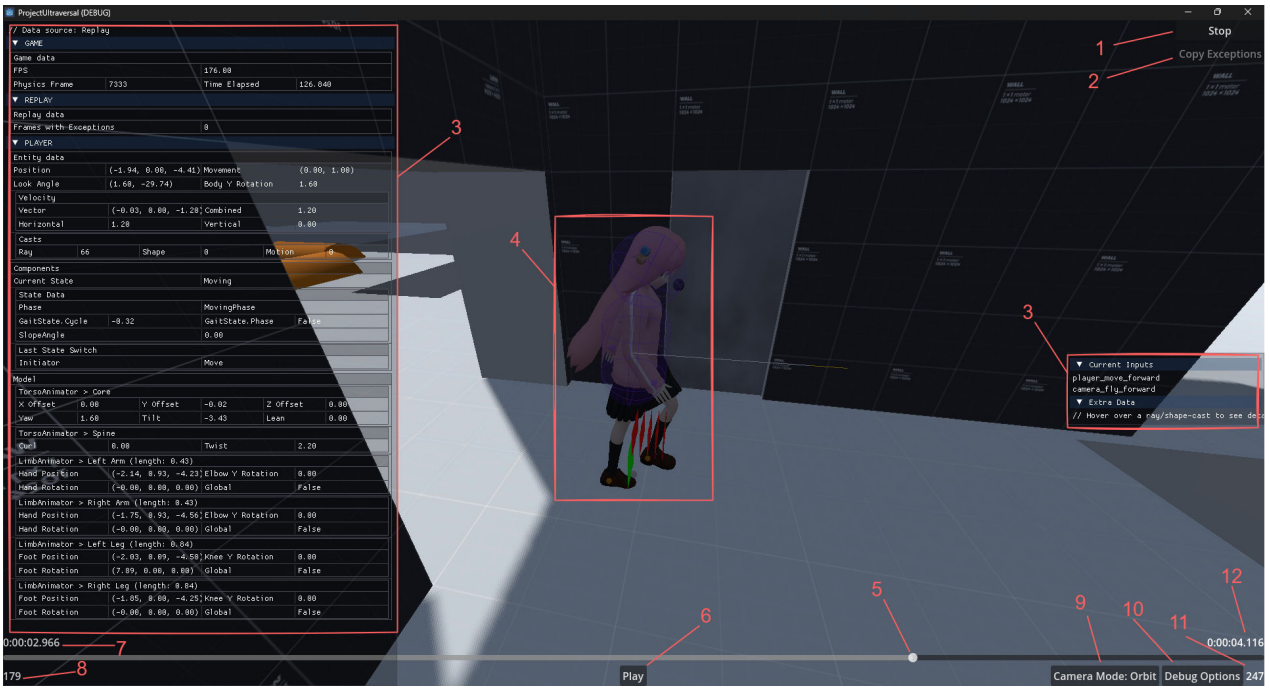


Рис. 2.14. Інтерфейс системи повтору з інструментами відладки

Елементи інтерфейсу системи повтору:

1. Кнопка завершення перегляду повтору.
2. Кнопка копіювання інформації про помилки.
3. Інструменти відладки.
4. Гравець повтору.
5. Повзунок відтворення.
6. Кнопка паузи.
7. Поточний час відтворення.
8. Поточний кадр відтворення.
9. Кнопка переключення режиму камери.
10. Кнопка спадного меню опцій відладки.
11. Повна кількість кадрів у повторі.
12. Повна тривалість повтору.

Для зручності перегляду було реалізовано три режими камери. В режимі Orbit, камера слідує за гравцем повтору. Вона завжди направлена на позицію гравця, але її можна обертати навколо нього, а також регулювати відстань колесиком миші. В

режимі FreeFlight користувач може вільно переміщувати та обертати камеру. В режимі POV позиція та оберт камери синхронізуються з позицією камери, збереженою в даних повтору, відтворюючи оригінальний вид від першого лиця.

Спадне меню Debug Options (рис. 2.15) дозволяє вмикати та вимикати частини інструментів відладки, описаних в наступному пункті.

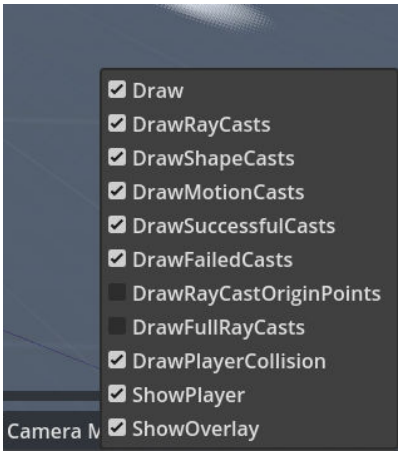


Рис. 2.15. Спадне меню опцій відладки

2.6.4. Інструменти відладки

На відміну від традиційного програмного забезпечення, проведення відладки в відеоіграх є складнішим процесом. Це зумовлено форматом вхідних та вихідних даних і часовою характеристикою ігрових процесів.

Головними форматами даних у 3D відеоіграх є вектори, кватерніони, базиси та трансформації. Самі по собі вони не дають розробнику достатньо інформації про те, що відбувається в грі під час помилки, адже вони насправді є лише вхідними даними для ігрових підсистем графіки та фізики. Аналогічно, вихідні дані гри – зображення на екрані та результати ігрового процесу – самі по собі не містять достатньо корисної інформації для проведення відладки.

Під часовою характеристикою мається на увазі циклічна структура ігрової логіки. В традиційному ПЗ, алгоритми в більшості випадків мають точку початку і точку кінця. Зупинивши програму у початковій точці та розглянувши крок за кроком дію алгоритму, розробник може виявити, де виникла помилка. В ігрових продуктах такий підхід не працює, адже більшість алгоритмів є нескінченними циклами, що

обробляють динамічні та моментальні дані, такі як ввод гравця клавіатурою або миші.

Проаналізувавши проблему, було прийнято рішення використати систему повторів для збору та представлення даних у форматі, зручному для відладки ігор. Для цього до об’єкту ReplayFrame були додані необов’язкові категорії даних відладки: PlayerData та GameData. PlayerData містить наступний набір даних:

- Vector3 GlobalPosition – координати гравця в ігровому просторі.
- Vector2 Movement – вектор вводу руху.
- Vector2 LookAngle – вектор кута погляду (оберту камери).
- float BodyYRotation – кут рискання тіла гравця.
- Vector3 Velocity – швидкість гравця (згідно фізичному рушію).
- string State – назва поточного стану.
- List<List<string>> StateData – вільний набір текстових даних відладки, отриманий з поточного об’єкту стану.
- StateDebugData? StateDebugData – набір додаткових статично-типізованих даних відладки стану.
- Type? LastStateSwitchInitiator – клас, відповідальний за останнє переключення стану.
- string[] Abilities – назви активних об’єктів Здібностей.
- List<CastBase> PerformedCasts – список кастів, здійснених у фізичному просторі від час цього кадру.
- TorsoAnimatorData ModelTorsoAnimatorData – дані TorsoAnimator.
- LimbAnimatorData ModelLimbAnimatorData – дані LimbAnimator.
- List<PlayerColliderData> PlayerColliders – дані (розташування, форма) поточних елементів колізії гравця.

GameData містить наступний набір даних:

- double Framerate – орієнтовна частота кадрів гри в цей момент часу.
- ulong PhysicsFrame – кількість фізичних циклів, що відбулись з моменту запуску гри.

- ulong MicrosecondTick – кількість мікросекунд що сплинули з моменту запуску гри.
- Dictionary<string, bool> ActionInputs – стан ігрового вводу (натиснуті клавіші).
- List<ExceptionData> Exceptions – список помилок, що відбулись під час цього кадру.

Далі була інтегрована система повтору відладки за принципом буфера. На початку гри автоматично починається запис реплею відладки. Коли довжина реплею досягає 1800 кадрів (30 секунд при частоті 60 кадрах в секунду), старі кадри починають видалятись. Натиснувши визначену клавішу, повтор відладки відтворюється, а буфер очищується. Таким чином розробник в будь який момент може відтворити та вивчити поведінку гри за період останніх 30 секунд.

Після цього система повтору була доповнена засобами малювання фігур в 3D просторі, на основі таких даних як PerformedCasts та PlayerColliders. Було створено набір класів, що представляють собою різні види об’єктів, які можна малювати у просторі. Базовим класом є DrawObjectBase.

```
public abstract class DrawObjectBase<T>
    where T : Node {
    public virtual float Duration { get; init; }
    public virtual Action<T>? OnReady { get; init; }

    public abstract T Create();
    public abstract void Draw(T node);
}
```

На його основі був створений базовий клас DrawMeshInstance3D що дозволяє малювати 3D-об’єкти.

```
public abstract class DrawMeshInstance3D : DrawObjectBase<MeshInstance3D> {
    public override MeshInstance3D Create() => new();

    protected virtual StandardMaterial3D UpdateMaterialProperties(StandardMaterial3D
material, bool alwaysOnTop, params Color[] colors) {
        material.NoDepthTest = alwaysOnTop;
        material.Transparency =
            colors.Any(color => !Extensions.Math.IsEqualApprox(color.A, 1)) ?
```

```
                BaseMaterial3D.TransparencyEnum.Alpha :
                BaseMaterial3D.TransparencyEnum.Disabled;
        return material;
    }
}
```

Прикладом класу малювання 3D-об’єкта є клас DrawSphere.

```
public class DrawSphere : DrawMeshInstance3D {
    public virtual Vector3 Origin { get; init; }
    public virtual float Radius { get; init; }
    public virtual Color? Color { get; init; }
    public virtual bool AlwaysOnTop { get; init; } = true;

    public override void Draw(MeshInstance3D meshInstance) {
        Color color = Color ?? Colors.White;
        meshInstance.GlobalPosition = Origin;
        meshInstance.Mesh = new SphereMesh() {
            Radius = Radius,
            Height = Radius * 2,
            Material = UpdateMaterialProperties(new() { AlbedoColor = color, },
AlwaysOnTop, color)
        };
    }
}
```

Викликаючи спеціально створені методи розширення, ігрові скрипти можуть реєструвати дії малювання фігур у просторі.

```
this.DebugDraw(new DrawSphere() { Origin = playerPosition, Radius = 0.025f, Color =
new(Colors.Brown, 0.5f)
});
```

Виклики малювання реєструються в чергу, виконання якої прив’язується до головного циклу поточної сцени. Після того як фігури були створені вони видаляються з черги.

```
public static List<DrawCall> DrawCalls { get; set; } = [];
private static Callable ExecuteCallable { get; } = Callable.From(Execute);

public static void Queue<T>(Node source, DrawObjectBase<T> drawObject)
    where T : Node {
    DrawCalls.Add(new DrawCall<T>(source, drawObject));
```

```
    SceneTree sceneTree = source.GetTree();
    if(DebugShapesContainer == null ||
        !GodotObject.IsInstanceValid(DebugShapesContainer)) {
        DebugShapesContainer = new() { Name = "DebugDraw" };
        sceneTree.Root.AddChild(DebugShapesContainer);
    }
    if(sceneTree.IsConnected(SceneTree.SignalName.ProcessFrame, ExecuteCallable)) {
        return;
    }
    _ = sceneTree.Connect(SceneTree.SignalName.ProcessFrame, ExecuteCallable);
}
private static void Execute() {
    if(!IsEnabled) {
        DebugShapesContainer?.QueueFree();
        DebugShapesContainer = null;
        DrawCalls.Clear();
        return;
    }
    if(DebugShapesContainer == null) {
        return;
    }
    for(int i = DrawCalls.Count - 1; i >= 0; i--) {
        DrawCall drawCall = DrawCalls[i];
        if(!drawCall.Execute(DebugShapesContainer)) {
            DrawCalls.RemoveAt(i);
        }
    }
}
```

Для цілей відладки, було налаштовано виклики малювання для візуалізації наступних даних:

1. Сфери координати гравця.
2. Лінії напрямку руху гравця.
3. Ліній вводу руху гравця.
4. Сфер координат кінцівок.
5. Ліній успішних raycast-ів.
6. Ліній неуспішних raycast-ів.

7. Сфер координат початку raycast-ів.
8. Фігур shapecast-ів.
9. Анімованих фігур motioncast-ів.
10. Фігур колізії гравця.
11. Додаткових фігур на основі даних поточного стану гравця.

Демонстрація малювання наведена на рисунку 2.16.

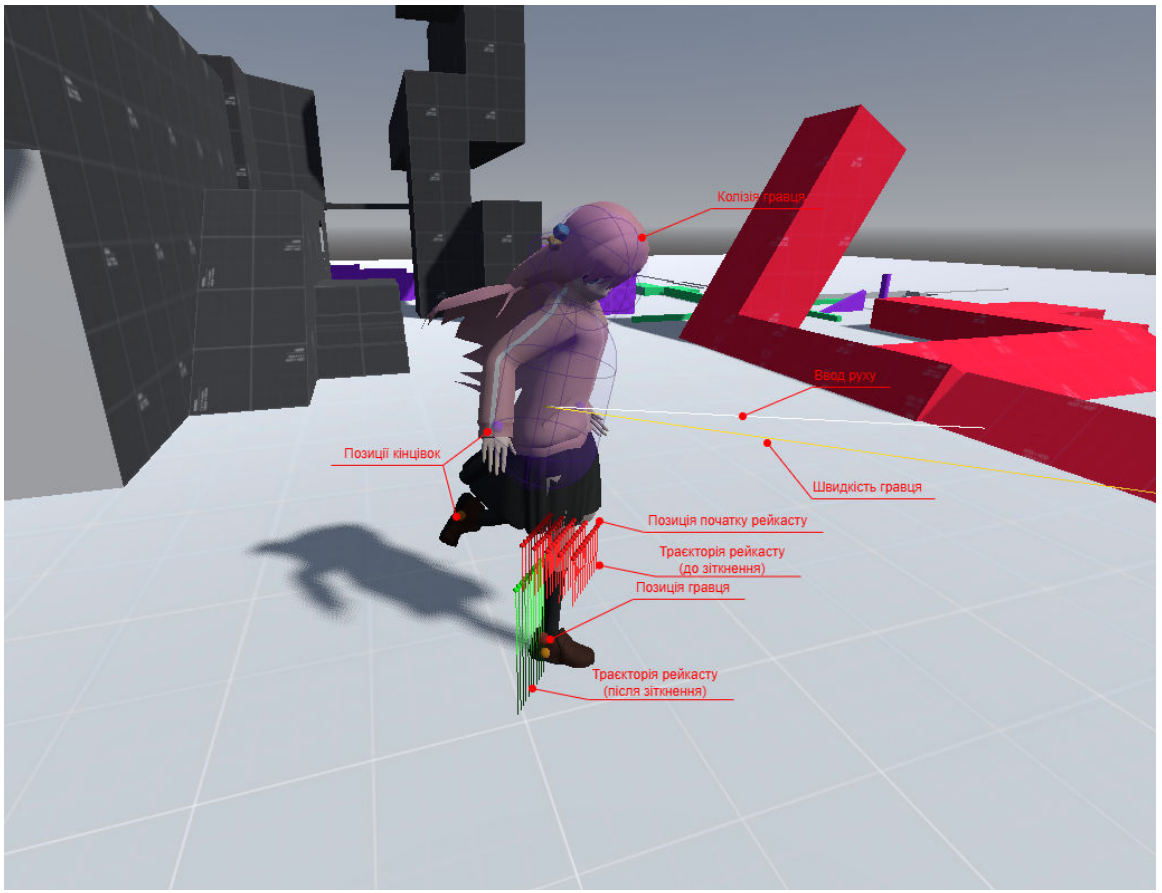


Рис. 2.16. Малювання фігур відладки в ігровому просторі

Другим ключовим елементом системи відладки є DebugOverlay (рис. 2.17, 2.18).

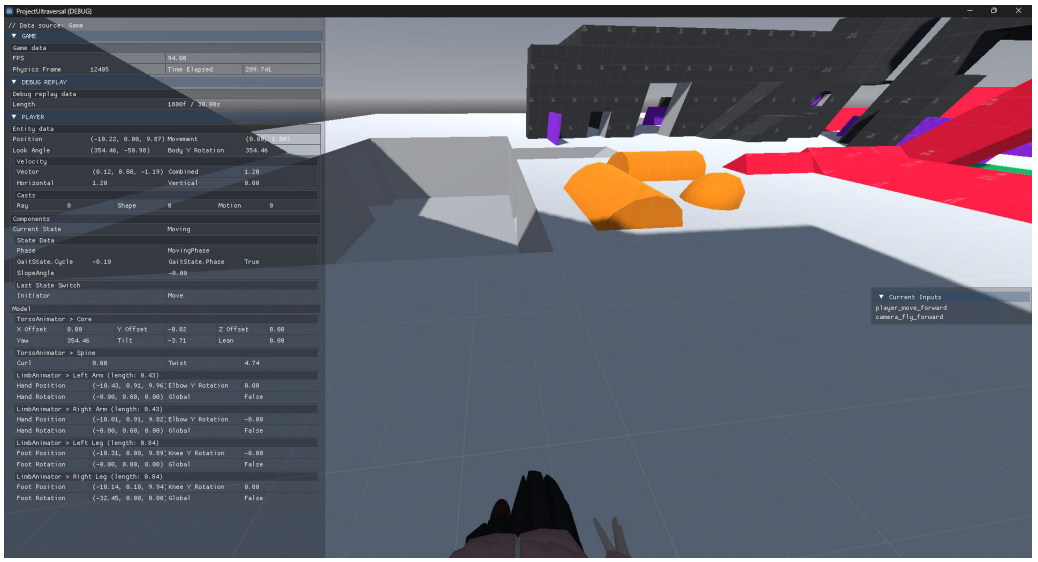


Рис. 2.17. Демонстрація оверлею відладки 1

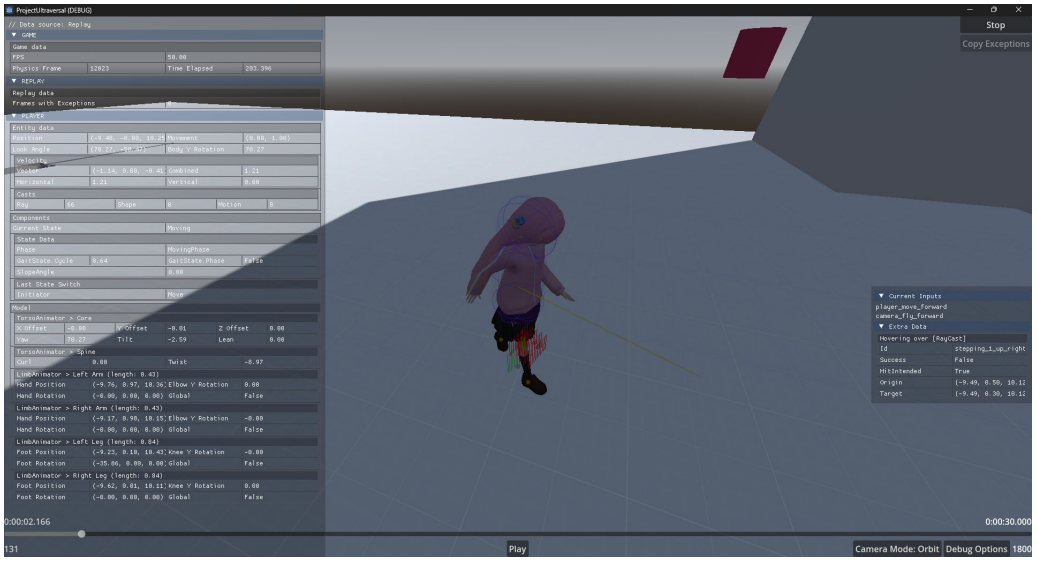


Рис. 2.18. Демонстрація оверлею відладки 2

DebugOverlay складається з двох панелей інтерфейсу, що оновлюються під час кожного кадру та відображають дані гри. Дані беруться з об’єкту ReplayFrame, джерелом якого є поточний повтор, або безпосередньо запущений екземпляр гри. Побудова інтерфейсу здійснюється безпосередньо в коді, зверху до низу, що дозволяє ефективно змінювати дані та стиль їх відображення в процесі розробки.

Головна панель оверлею містить дві головні секції – Game та Player – та одну додаткову секцію – Replay/Debug Replay. Додаткова секція змінюється, в залежності від джерела даних. Якщо дані беруться з гри, то в ній відображається довжина

повтору відладки, який записується в даний момент. Якщо дані беруться з повтору, то відображається кількість помилок, що відбулися під час його запису.

Додаткова панель оверлею містить секції Current Inputs та Extra Data. Current Inputs містить перелік всіх дій вводу в даний момент часу. Якщо під час перегляду реплею навести курсор миші на рейкаст, в секції Extra Data буде відображена детальна інформація про цей рейкаст.

Висновок

Результатами досліджень та розробки, проведених у другому розділі, було реалізовано ключові системи контролеру персонажа, процедурної анімації та повтору/відладки.

Розроблений контролер використовує комбінацію патернів State та Component, що дозволяє реалізувати модульну структуру керування будь-яким ігровим персонажем. В основі системи лежить патерн Component – частини контролера реалізуються як індивідуальні компоненти.

Першим видом компоненту є State – стан гравця. Ключовою особливістю цього компоненту є те, що лише один стан може бути активний одночасно. Таким чином реалізується патерн State. Для керування станами використовує другий вид компоненту – Ability. Здібності можуть бути індивідуально вимкнені та ввімкнені, що дозволяє легко включати та виключати частини контролера.

Додатково було імплементовано концепцію домішок, що дозволяє додавати спільний функціонал до компонентів, уникаючи проблем, пов’язаних зі складними ієрархіями наслідування, що часто зустрічаються в ООП модулях. В стани також було додано підтримку динамічних колізій, що є необхідним для реалізації складних паркур-механік.

Розроблена система процедурної анімації складається з двох компонентів: PlayerModel API та Drivers API. Керування моделлю гравця здійснюється трьома класами, що відповідають за оберти тулуба, голови та кінцівок.

До оберту тулуба входять можливість керування обертом основи моделі по трьох осях, а також обертом кісток хребта для додаткових анімацій нахилу. Оберт

голови пов’язаний з обертом кістки шиї, що дозволяє реалізувати природні їх анімації. Керування кінцівками здійснюється через систему інверсної кінематики, розробленої для реалістичного позиціонування кісток рук та ніг.

Drivers API інтегрується зі станами та API моделі гравця, та дозволяє задавати кожному стану необхідну анімацію за допомогою спеціальних об’єктів конфігурації в редакторі ігрового рушія. Була створена можливість різних видів анімації та переходів властивостей моделі, включаючи анімацію по кривих у часі, згладжування (інтерполяцію), контроль значень через код, фіксоване приведення до цільового значення, встановлення константного значення, а також перетворення значення з локального простору у глобальний.

Також була створена підтримка шарів драйверів (DriverLayers) що дозволяє накладати декілька наборів драйверів анімації один на одного та керувати їх виконанням через код стану.

Третьою створеною системою є система повтору, з інтегрованими інструментами відладки. Система повтору дозволяє покадрово записувати стан гри під час ігрового процесу, та відтворювати його. Інтерфейс системи повтору дозволяє покадрово переглядати геймплей, з можливістю фіксації камери на гравці. Інструменти відладки працюють разом з цією системою та складаються з двох компонентів: оверлей та малювання в просторі. Оверлей відображається поверх інтерфейсу гри та містить інформацію про її поточний стан, або про її стан в поточному кадрі повтору. Засоби малювання використовуються для візуалізації рейкастів, колізії, векторів та інших ігрових даних, що дозволяє розробнику ефективно досліджувати стан гри в будь який момент часу.

На основі розроблених систем, в наступному розділі буде розроблено повноцінний прототип ігрового продукту, в якому буде реалізовано набір складних паркур-трюків та різні види переміщення в просторі.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ СТАНІВ ГРАВЦЯ (РУХ ТА ТРЮКИ)

3.1. Реалізація спільної логіки станів за допомогою домішок

Для реалізації компонентів станів та здібностей на основі розробленого контролера персонажа було створено набір класів-домішок. Домішками станів є IAnimator, ICrouching, IUsesGateState, IGait, INaturalBodyRotation, IPhysics, ITimeDuration та IRecordPlayerProperties.

3.1.1. IAnimator

Домішка IAnimator поєднує між собою систему Drivers та код керування моделлю гравця, та прив’язує їх до поточного стану. Він містить властивість DriverLayer, в якій зберігається головний об’єкт драйверного шару для поточного стану, методи AnimateInternal, GetDriversRecursive та UpdateDriversOfType, що є частиною його API, а також хуки що використовують ці API.

```
public interface IAnimator : IMixin<Player> {
    DriverLayerBase DriverLayer { get; }

    [MixinHook(HookEvent.PhysicsProcess, HookType.Post)]
    protected void OnPostPhysicsProcess() =>
        AnimateInternal(Entity.Model, This.GetPhysicsProcessDeltaTime(),
Entity.BodyYRotation, Entity.LookAngle, Entity.BodyTransform, This as
ITimeDuration);
    [MixinHook(HookEvent.Enter)]
    protected virtual void OnPostEnter() {
        foreach(DriverBase driver in DriverLayer.Drivers) {
            driver.Reset();
        }
    }
    public virtual void AnimateInternal(PlayerModel model, double delta, float
bodyYRotation, Vector2 lookAngle, Transform3D bodyTransform, ITimeDuration?
timeDuration = null) {
        HeadAnimator headAnimator = model.HeadAnimator;
        TorsoAnimator torsoAnimator = model.TorsoAnimator;
        LimbAnimator limbAnimator = model.LimbAnimator;
        UpdateDriversOfType<Vector3LocalSpaceTransformer>(transformer =>
transformer.GlobalTransform = bodyTransform);
```

```
        UpdateDriversOfType<IFunctionChainDriver>(functionChainDriver => {
            if(timeDuration != null) {
                foreach(ITimeCurveFunction timeCurveFunction in
functionChainDriver.GetFunctionChain().OfType<ITimeCurveFunction>()) {
                    timeCurveFunction.Progress = timeDuration.Progress;
                }
            }
        });
        torsoAnimator.CoreYaw = bodyYRotation;
        torsoAnimator.CoreXOffset =
DriverLayer.CoreXOffset.Drive(torsoAnimator.CoreXOffset, delta);
        torsoAnimator.CoreYOffset =
DriverLayer.CoreYOffset.Drive(torsoAnimator.CoreYOffset, delta);
```

...

```
    }
    public virtual IEnumerable<DriverBase> GetDriversRecursive() =>
        DriverLayer.Drivers.SelectMany(driver => (IEnumerable<DriverBase>) [
            driver, ..driver.GetNestedDrivers()
        ]);
    public virtual void UpdateDriversOfType<T>(Action<T> action) {
        foreach(T driver in GetDriversRecursive().OfType<T>()) {
            action(driver);
        }
    }
}
```

3.1.2. ICrouching

Ця домішка використовується станами, де гравець присідає і забезпечує виявлення низької стелі на головою гравця. Якщо така є, нічого не відбувається. Якщо стелі немає, виконується метод UnCrouch, що дозволяє стану виконати переключення на інший стан, наприклад зі стану Crawling автоматично перейти до стану Moving, як тільки над головою гравця є достатньо місця.

Перед викликом методу UnCrouch виконується перевірка булевої змінної ShouldUnCrouch. Вона може бути встановлена здібністю Crouch при наявності вводу “присісти”, що запобігає автоматичному “вставанню” гравця.

```
public interface ICrouching : IMixin<Player> {
    bool ShouldUnCrouch { get; set; }
    Shape3D StandUpCheckShape { get; }
    Transform3D StandUpCheckTransform { get; }
    Vector3 StandUpCheckMotion { get; }

    void UnCrouch();
    [MixinHook(HookEvent.Enter, HookType.Post)]
    protected void OnPostEnter() => ShouldUnCrouch = false;
    [MixinHook(HookEvent.PhysicsProcess, HookType.Post)]
    protected void OnPostPhysicsProcess() {
        bool canStandUp = !Entity.TryCastMotion(StandUpCheckTransform,
StandUpCheckMotion, StandUpCheckShape, false, "stand_up_check");
        if(ShouldUnCrouch && canStandUp) {
            UnCrouch();
        }
        ShouldUnCrouch = true;
    }
}
```

3.1.3. IUsesGateState

Ця домішка містить властивість GaitState типу GaitState, та забезпечує її збереження при переключенні між станами. Властивість GateState містить дані, пов’язані з анімаціями та розрахунками ходьби та бігу. Якщо стан, що використовує цю властивість, переключається на інший стан, який теж її використовує, доречно оновити об’єкт GaitState у новому стані на основі GaitState попереднього стану (продовжуючи анімацію та розрахунки). Якщо ж попередній стан не є станом, що використовує GateState, тоді необхідно очистити дані властивості нового стану (почати анімацію/розрахунки з нуля).

```
public interface IUsesGaitState : IMixin<Player> {
    GaitState GaitState { get; set; }

    [MixinHook(HookEvent.Enter, HookType.Pre)]
    protected void OnPreEnter() {
        PlayerStateBase.CurrentStateSwitchData? currentStateSwitchData =
PlayerStateBase.Global.GetCurrentStateSwitchData(Entity);
```

```
        PlayerStateBase? previousState = currentStateSwitchData?.CurrentState as
PlayerStateBase;
        GaitState = (previousState is IUsesGaitState usesGaitState) ?
usesGaitState.GaitState : new();
    }
}
```

3.1.4. IGait

Ця домішка надає стану методи для відтворення анімацій ходьби та бігу. Для створення анімацій використовуються набори кривих, що описують координати частин тіла у відповідності до циклу ходьби, та об’єкт GateState. Криві анімації зберігаються в ресурсі GaitAnimation.

```
[GlobalClass]
public partial class GaitAnimation : Resource {
#nullable disable
    [ExportGroup("Torso")]
    [Export] public virtual DampingCurve CoreYOffsetCurve { get; set; }
    [Export] public virtual DampingCurve CoreXOffsetCurve { get; set; }
      ...
    [ExportGroup("Feet")]
    [Export] public virtual Curve StanceFootYOffsetCurve { get; set; }
      ...
    [ExportGroup("Hands")]
    [Export] public virtual Curve HandXOffsetCurve { get; set; }
      ...
}
```

Було створено два ресурси GaitAnimation: WalkAnimation та RunAnimation, для анімацій ходьби та бігу відповідно. В домішці IGait було створено методи-утиліти для інтерполяції між двома анімаціями, щоб забезпечити плавний перехід від ходьби до бігу та навпаки.

```
protected float SampleCurveBlend(Curve curve1, Curve curve2, float offset, float
weight) => weight switch {
        <= 0 => curve1.Sample(offset),
        >= 1 => curve2.Sample(offset),
        _ => curve1.Sample(offset).Lerp(curve2.Sample(offset), weight)
    };
```

Для моделювання циклу ходьби використовується об’єкт GaitState. GaitState описує поточний етап циклічної анімації ходьби за допомогою наступних змінних:

- Cycle – значення в межах [-1; 1], що вказує на позиції ніг. При значенні -1 права нога знаходиться позаду. При значенні 1 – ліва нога знаходиться позаду.
- Phase – булева змінна, що вказує на ролі ніг. При значенні true ліва нога є опорною, права здійснює крок. При значенні false ролі змінюються.
- CycleCount – лічильник циклів анімації.
- StanceSwitchCount – лічильник змін стійки.
- CurrentStepLength – довжина кроку, внутрішнє значення.
- QueuedStepLength – наступна довжина кроку, публічний API.

Для оновлення стану в ньому міститься метод Advance, що проводить інкремент або декремент значення Cycle в залежності від значення Phase, та за необхідності інкрементує лічильники та оновлює значення CurrentStepLength. Для забезпечення плавності анімації, змінній CurrentStepLength присвоюється значення QueuedStepLength лише в момент переходу до нового циклу. Для зручного користування API було також створено версію методу Advance, що обраховує значення інкременту/декременту на основі вхідних даних ходьби, за формулою:

$x = \frac{v}{stepLength} \times fwd \times \Delta t$, де

v – швидкість руху

$stepLength$ – довжина кроку CurrentStepLength

fwd – 1 при русі вперед, -1 при русі назад

Δt – дельта часу

Для безпосереднього виконання анімації в домішці IGait є методи AnimateHands, AnimateFeet та AnimateTorso. Кожний з них використовує властивості об’єкту GaitState для знаходження необхідних значень кривих у поточному моменті циклу анімації, наприклад:

```
DriverLayer.CoreYOffset.DefaultTarget =
    SampleCurveBlend(WalkAnimation.CoreYOffsetCurve, RunAnimation.CoreYOffsetCurve,
                     GaitState.StepProgress, torsoAnimator.CoreYOffset,
```

```
                        delta, dampen, blendWeight);
DriverLayer.SpineTwist.DefaultTarget =
    SampleCurveBlend(WalkAnimation.SpineTwistCurve, RunAnimation.SpineTwistCurve,
                        GaitState.Cycle, torsoAnimator.SpineTwist,
                        delta, dampen, blendWeight);
```

3.1.5. INaturalBodyRotation

Ця домішка використовується станами для автоматичного оберту тіла гравця навколо вертикальної осі Y згідно його куту погляду. Характер оберту конфігурується ресурсом NaturalBodyRotation, що містить властивості конфігурації HeadFollowAngleSoftThreshold, HeadFollowAngleHardThreshold та RotationSmoothness.

HeadFollowAngleSoftThreshold визначає мінімальне відхилення кута погляду від кута оберту тіла, при якому тіло має почати обертатись. HeadFollowAngleHardThreshold визначає максимальне відхилення. Якщо різниця кутів перевищує це значення то тіло буде моментально повернуте до необхідного кута без згладжування. RotationSmoothness визначає коефіцієнт згладжування оберту.

```
public interface INaturalBodyRotation : IMixin<Player> {
    NaturalBodyRotation NaturalBodyRotation { get; }
    float TargetBodyYRotation { get; set; }

    [MixinHook(HookEvent.Process, HookType.Pre)]
    protected virtual void OnPostProcess() {
        double delta = Entity.GetProcessDeltaTime();
        Vector2 lookAngle = Entity.LookAngle;
        if(lookAngle.X.AngleDifferenceAbs(TargetBodyYRotation) >
            NaturalBodyRotation.HeadFollowAngleSoftThreshold) {
            TargetBodyYRotation = lookAngle.X;
        }
        Entity.BodyYRotation = Entity.BodyYRotation.DampAngle(TargetBodyYRotation,
NaturalBodyRotation.RotationSmoothness, delta);
        float hardThreshold = NaturalBodyRotation.HeadFollowAngleHardThreshold;
        Entity.BodyYRotation = Entity.BodyYRotation.ClampRad(lookAngle.X -
hardThreshold, lookAngle.X + hardThreshold);
    }
    [MixinHook(HookEvent.Enter, HookType.Post)]
```

```
    protected virtual void OnPostEnter() =>
        TargetBodyYRotation = Entity.LookAngle.X;
}
```

3.1.6. IPhysics

Ця домішка використовується станами, яким необхідно використовувати фізичний рушій для переміщення гравця, реалізуючи описаний раніше гібридний підхід до руху гравця.

```
public interface IPhysics : IMixin<Player> {
    [MixinHook(HookEvent.PhysicsProcess, HookType.Post)]
    protected void OnPostPhysicsProcess() => Entity.MoveAndSlide();
}
```

3.1.7. ITimeDuration

Ця домішка використовується станами, що мають фіксовану тривалість часу. Він відслідковує час, що сплив з моменту входження в стан та надає властивості для отримання інформації про прогрес та сплив часу.

```
public interface ITimeDuration : IMixin<Player> {
    float Duration { get; }
    float TimeLeft { get; set; }

    public float Progress => 1 - (TimeLeft / Duration);
    public bool IsElapsed => TimeLeft < 0;
    public float TimeElapsed => Duration - TimeLeft;

    [MixinHook(HookEvent.Process, HookType.Pre)]
    protected void OnPreProcess() => TimeLeft -= (float) This.GetProcessDeltaTime();
    [MixinHook(HookEvent.Enter, HookType.Post)]
    protected void OnPostEnter() => TimeLeft = Duration;
}
```

3.1.8. IRecordPlayerProperties

Цю домішку було додано до базового класу стану PlayerStateBase. Його роль полягає у збереженні останніх значень позиції та швидкості гравця, адже це необхідно для багатьох алгоритмів його руху.

```
public interface IRecordPlayerProperties : IMixin<Player> {
    [MixinHook(HookEvent.PhysicsProcess, HookType.Post)]
    protected virtual void OnPostPhysicsProcess() {
        Entity.PreviousGlobalPosition = Entity.GlobalPosition;
        Entity.PreviousVelocity = Entity.Velocity;
    }
}
```

3.2. Базовий рух

Базовий рух гравця складається з наступних станів:

1. Default – стоїть на місці.
2. Moving – ходьба та біг.
3. Stepping – переступання.
4. Grinding – ковзання стоячи (гальмування).
5. Jumping – стрибок.
6. Falling – падіння.
7. LandingContact – момент приземлення.
8. LandingHarshly – просте невдале приземлення.
9. Crouching – в присяді.

Цей набір рухів дозволяє вільно переміщатися по рівній та похилій землі, а також долати невеликі перешкоди. Далі детально розглянемо реалізацію цих станів, та класи-здібності, що ними керують.

3.2.1. Default

В цьому стані гравець просто стоїть на місці. Якщо до цього гравець рухався, його швидкість плавно впаде до нуля. Оскільки цей стан поділяє всю свою логіку зі станом Crouching, було створено базовий клас StationaryStateBase. Домішками стану є: IPhysics, INaturalBodyRotation.

Під час кожного фізичного циклу, горизонтальна швидкість гравця змінюється за формулою згладжування, цільовим значенням якої є нульовий вектор, а коефіцієнт встановлюється в редакторі через властивість StopSmoothness. Вертикальна швидкість встановлюється 0.

Після цього виконується алгоритм визначення характеру поверхні під гравцем. Було створено клас FloorMatrix, що представляє собою двовимірну матрицю 3x3, де кожна комірка – секція підлоги під, та довкола гравця. Центральна комірка матриці – це значення безпосередньо по координатах гравця. Значення по периметру матриці лежать по колу навколо центральної позиції. Значення комірок є тривімірними векторами, що вказують на координати комірок в просторі. Якщо комірка не має значення, то в цьому місці немає підлоги.

Для визначення координатів використовуються 9 вертикальних рейкастів. Горизонтальна позиція центрального рейкасту співпадає з координатами гравця у світі. Радіус кола, по якому робляться 8 додаткових рейкастів дорівнює властивості StanceWidth / 2, тобто стандартній ширині стійки для даної моделі. Вертикальне зміщення кожного рейкасту є відносним вертикальній координаті гравця та було визначено експериментальним шляхом. Якщо в результаті рейкасту була знайдена позиція, і її відхилення від вертикальної осі не перевищує 45°, ця позиція додається до FloorMatrix.

3D-візуалізація променів, реалізована за допомогою інструментів відладки, наведена на рисунку 3.1.

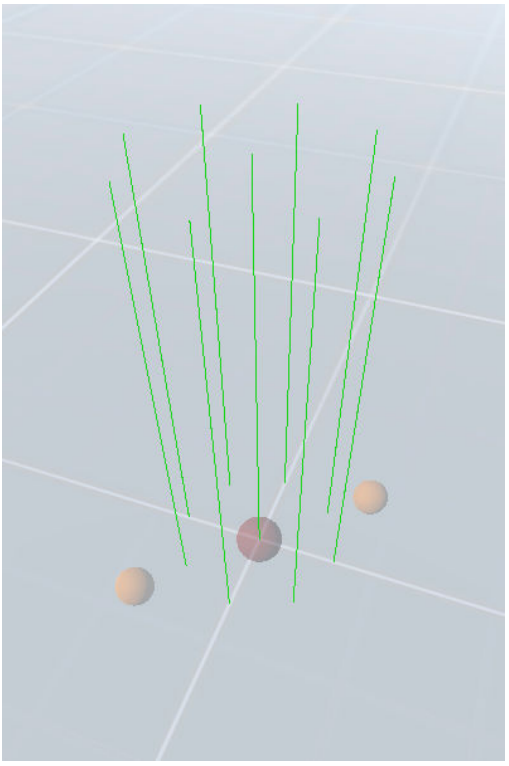


Рис. 3.1. Візуалізація променів сканування підлоги та знайдені координати

Отримавши заповнену FloorMatrix, ми можемо визначити характер підлоги під гравцем. Для того щоб гравець мав можливість стояти на місці, під ним має бути наявною або центральна комірка, або пара протилежних комірок по периметру. При наявності декількох підходящих позицій, першою в списку пріоритетів є діагональна пара позицій під кутом 45° по колу. Якщо такої пари нема, обирається будь яка інша пара позицій. Якщо жодної пари позицій знайдено не було, обирається центральна позиція. Якщо центральна позиція теж відсутня, то вважається що підлога під гравцем не є придатною і здійснюється переключення стану в Falling. Результати визначення підлоги продемонстровано на рисунку 3.2.

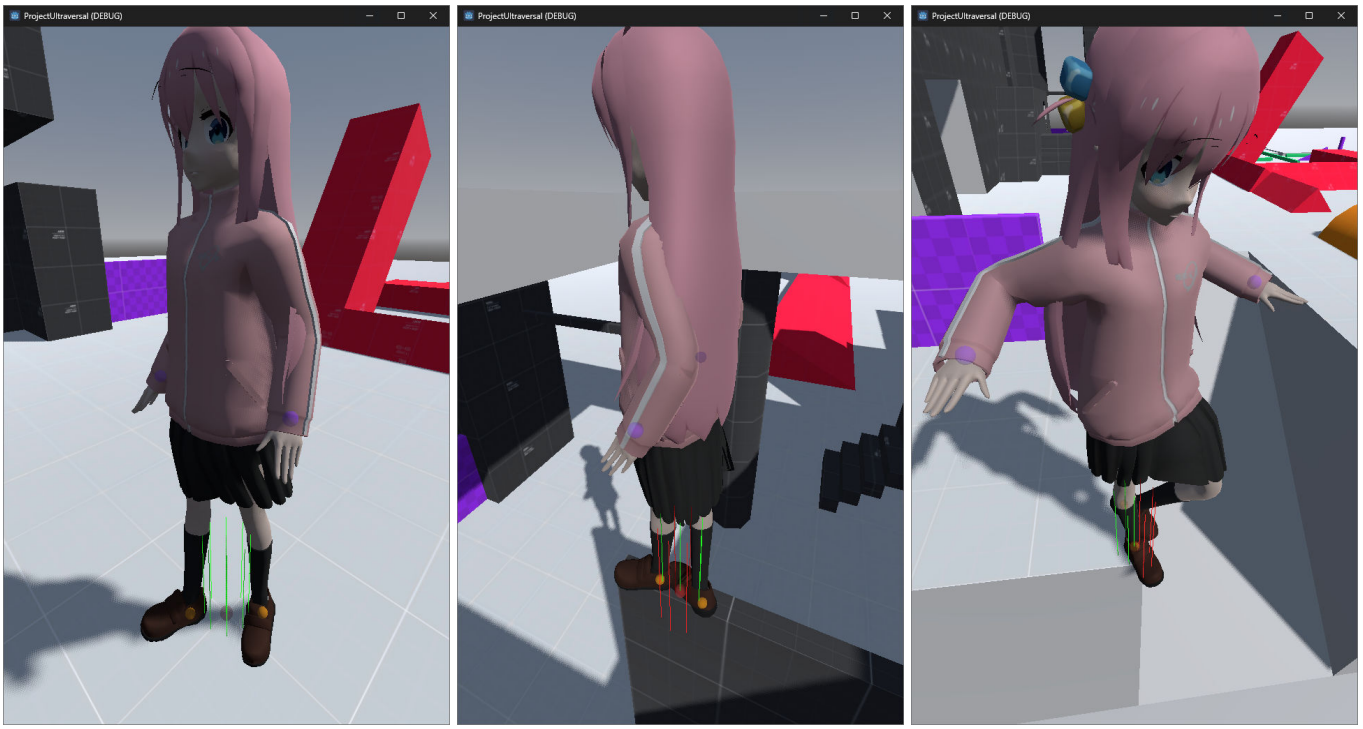


Рис. 3.2. Демонстрація адаптації моделі до неповної підлоги

Також в цьому стані відбувається зміщення моделі гравця виходячи зі знайдених координатів ніг на підлозі. Вертикальна координата гравця обраховується як середнє арифметичне вертикальних координат підлоги:

$$yPosition = \frac{yPosition_{leg1} + yPosition_{leg2}}{2}$$

Додатково відбувається коригування вертикального зміщення моделі CoreYOffset. Це гарантує, що обидві ступні зможуть торкнутися підлоги.

$$yAdjustment = min(yPosition_{leg1}, yPosition_{leg2}) - yPosition$$

Знайдене зміщення далі передається всім знайденим драйверам спеціалізованого типу SingleUnevenGroundTransformer. Результати корекції ніг зображено на рисунку 3.3.

Рис. 3.3. Демонстрація адаптації моделі до похилої підлоги

Драйверна ієрархія була побудована наступним чином:

- DriverLayerSet
  - DriverLayer (Base)
    - **Core Y Offset**
      - SingleUnevenGroundTransformer
        - SingleDriver
          - *SingleConstantFunction*
          - *SingleDampingFunction*
    - **Core Tilt**
      - SingleDriver
        - *SingleAngleDampingFunction { Smoothness: 10 }*
    - **Left Foot Position**
      - Vector3Driver
        - *Vector3DampingFunction*

- **Right Foot Position**
  - Vector3Driver
    - *Vector3DampingFunction*
- **Left Hand Position**
  - Vector3LocalSpaceTransformer
    - Vector3Driver
      - *Vector3ConstantFunction { Value: (-0.225, 0.95, 0) }*
      - *Vector3DampingFunction { Smoothness: 10 }*
- **Right Hand Position**
  - Vector3LocalSpaceTransformer
    - Vector3Driver
      - *Vector3ConstantFunction { Value: (0.25, 0.95, -0.055) }*
      - *Vector3DampingFunction { Smoothness: 10 }*

- ToggleableDriverLayer (Balancing)
  - **Left Foot Position**
    - Vector3LocalSpaceTransformer
      - Vector3Driver
        - *Vector3ConstantFunction { Value: (0, 0.2, 0.1) }*
        - *Vector3DampingFunction*
  - **Right Foot Position**
    - Vector3LocalSpaceTransformer
      - Vector3Driver
        - Vector3ConstantFunction
        - *Vector3DampingFunction*
  - **Left Knee Y Rotation**
    - SingleDriver
      - *SingleAngleConstantFunction { Value: -30° }*
      - *SingleAngleDampingFunction*
  - **Left Hand Position**
    - Vector3LocalSpaceTransformer
      - Vector3Driver
        - *Vector3ConstantFunction { Value: (-0.4, 1.2, 0) }*
        - *Vector3DampingFunction { Smoothness: 10 }*
  - **Right Hand Position**
    - Vector3LocalSpaceTransformer
      - Vector3Driver
        - *Vector3ConstantFunction { Value: (0.4, 1.2, 0) }*
        - *Vector3DampingFunction { Smoothness: 10 }*

3.2.2. Moving

В стані Moving гравець йде або біжить по рівній або похилій поверхні. Домішками стану є: IPhysics, IGait, IUsesGaitState. Для реалізації цього була створена система конфігурацій (Movement Configuration) та анімацій (Gait Animation). Gait Animation використовує домішку IGait, ресурси GaitAnimation та GaitState.

MovementConfiguration є ресурсом, що містить набір даних, які описують технічні аспекти ходьби:

- TopSpeed – максимальна швидкість.
- SpeedOverSlopeAngle – крива залежності швидкості від куту нахилу поверхні.
- StepLengthRange – мінімальна та максимальна довжина кроку.
- AccelerationSpeed – доданок у функції прискорення.
- AccelerationFactor – множник у функції прискорення.
- VelocityVectorSmoothness – параметр функції згладжування вектору швидкості.
- LeanFactor – коефіцієнт нахилу тіла при повороті.
- TurnAroundEnterSpeedThreshold – мінімальна швидкість, необхідна для фази розвороту.
- TurnAroundDuration – тривалість фази розвороту.
- TurnAroundExitSpeed – швидкість при виході з фази розвороту.
- DecelerationFactor – коефіцієнт сповільнення при вході в фазу зупинки.
- DecelerationSmoothness – згладжування сповільнення.

Було створено два ресурси MovementConfiguration – WalkConfiguration та RunConfiguration. Стан Moving автоматично обирає необхідний набір даних в залежності від декількох факторів. Якщо гравець має можливість бігти (булева властивість AllowRunning) і ввод вказує на рух вперед, вліво чи вправо, дані RunConfiguration будуть використані. В іншому випадку будуть використані дані WalkConfiguration. Дані анімації обираються алгоритмом змішування з двох наборів на основі швидкості гравця:

$$blend = invlerp(topSpeed_w, topSpeed_r, clamp(\|v\|, topSpeed_w, topSpeed_r))$$

Логіка стану Moving була розділена на окремі фази, представлені класами-записами:

- StartingMovingPhase – гравець починає рухатись. Його швидкість збільшується відповідно значенням AccelerationSpeed та AccelerationFactor. Параметри:
    - Speed – базова поточна швидкість.
- MovingPhase – гравець рухається. Його вектор швидкості постійно приводиться до вектору руху згідно вводу та значенню VelocityVectorSmoothness. В залежності від різниці векторів розраховується нахил тіла гравця. Параметри:
    - TargetVelocity – цільова швидкість.
- TurningAroundPhase – гравець розвертається. Його швидкість приводиться до 0. Параметри:
    - InitialVelocity – початкова швидкість.
    - TimeElapsed – скільки часу спливло з моменту входу в фазу.
- StoppingPhase – гравець зупиняється. Параметри:
    - TargetVelocity – цільова швидкість.

При відсутності вводу, гравець починає зупинятись. Це відбувається одним з трьох способів, в залежності від швидкості гравця:

- При значній швидкості (швидкість більша за GrindSpeedThreshold) стан переключається на Grinding.
- При низькій швидкості (швидкість менша за StepStopThreshold) стан переключається на Default.
- При середній швидкості фаза переключається на Stopping, і після здійснення одного кроку стан переключається на Default.

Логіка переключення фаз зображена на рисунку 3.4.

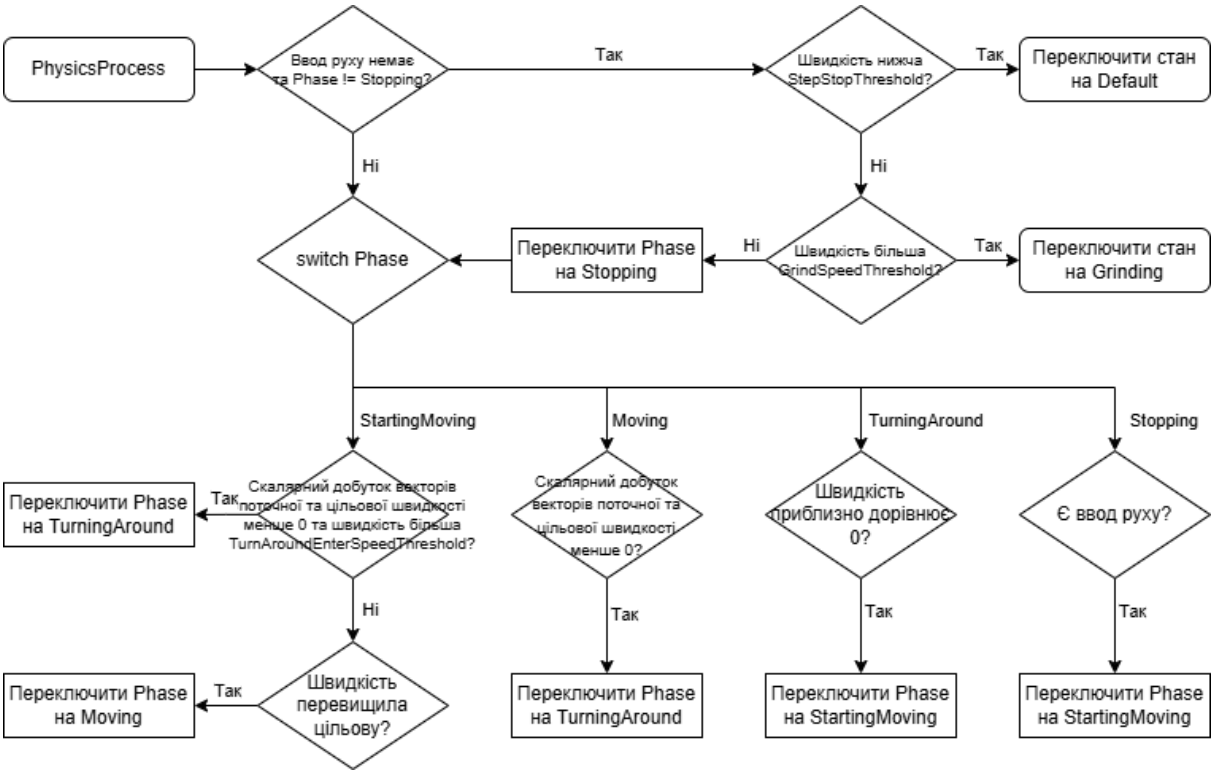


Рис. 3.4. Діаграма логіки переключення фаз

Розкадровки анімацій бігу та ходьби наведені на рисунках 3.5 та 3.6.

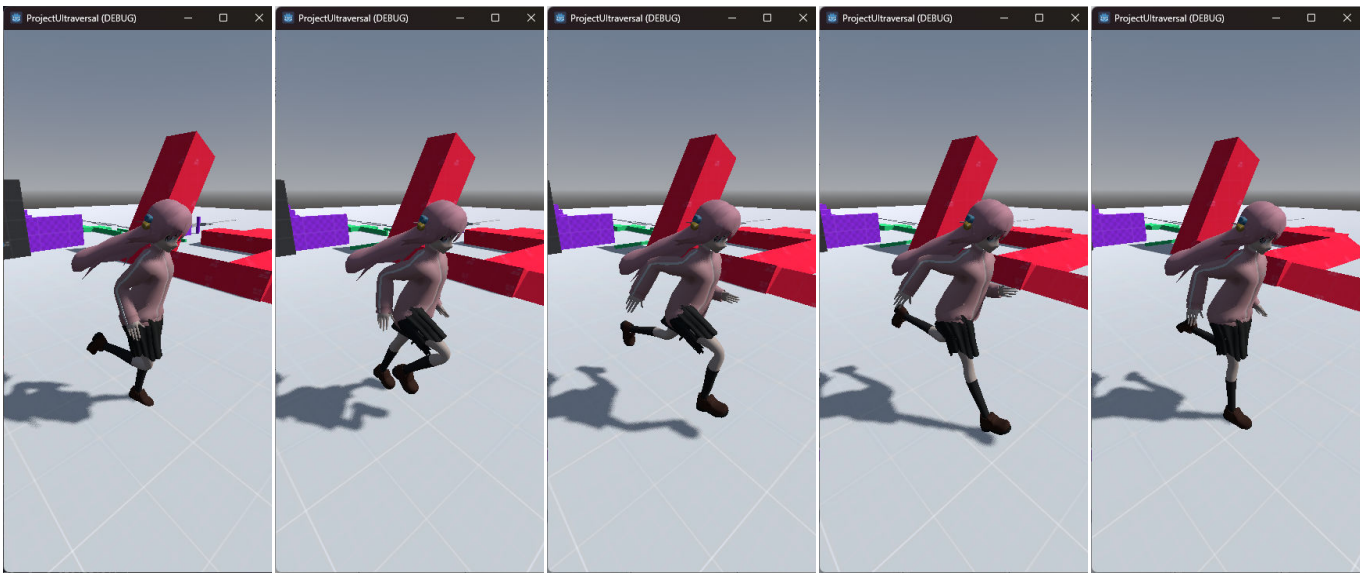


Рис. 3.5. Розкадровка анімації бігу

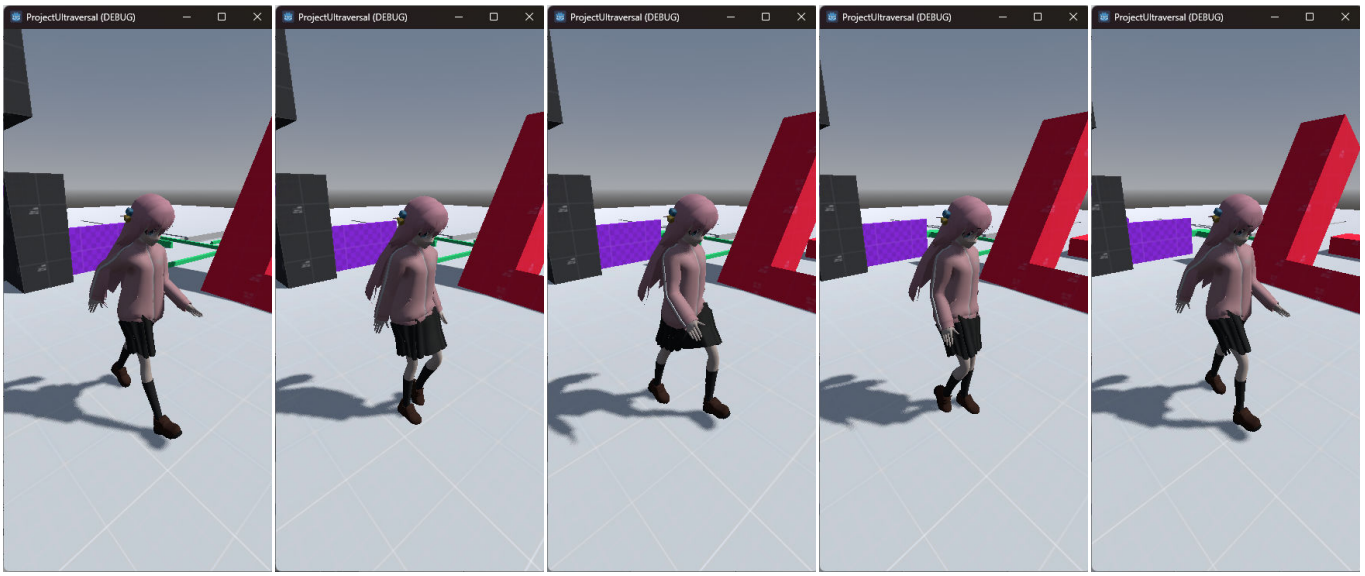


Рис. 3.6. Розкадровка анімації ходьби

3.2.3. Stepping

В стані Stepping гравець переступає через невеликі перешкоди. Цей стан було створено для забезпечення плавності анімацій при переміщенні по нерівній поверхні. Домішками стану є: ITimeDuration, IGait, IUsesGaitState. Параметрами конфігурації стану є:

- SteppingData SteppingData – спільний ресурс даних стану.
- MovementConfiguration WalkConfiguration – ресурс налаштувань ходьби.
- MovementConfiguration RunConfiguration – ресурс налаштувань бігу.
- Vector2 DurationRange – максимальна та мінімальна тривалість стану.
- Curve StepCurve – крива переміщення тіла.
- float ExitSpeedLimitWhenMovementIsZero – вихідна швидкість при відсутності вводу.
- GaitAnimation WalkAnimation – ресурс анімації ходьби.
- GaitAnimation RunAnimation – ресурс анімації бігу.

При вході в стан визначається початкова та кінцева позиція гравця. Початкова позиція береться з поточної, а кінцева – з ресурсу SteppingData, найближча до початкової. Після цього розраховується тривалість стану за формулою:

$t = lerp(t_{min}, t_{max}, invlerp(vr_{max}, vw_{max}, v))$, де:

t_{min} – мінімальна тривалість стану;

t_{max} – максимальна тривалість стану;

vr_{max} – максимальна швидкість бігу RunConfiguration.TopSpeed;

vw_{max} – максимальна швидкість ходьби WalkConfiguration.TopSpeed;

v – поточна швидкість руху.

Також визначається якою ногою здійснюється переступання. При русі вперед це значення GaitState.Phase, при русі назад – обернене значення GaitState.Phase.

Під час кожного кадру продовжується анімація руху за допомогою виклику GaitState.Advance, а позиція гравця інтерполюється від початкової до кінцевої згідно кривої StepCurve. Якщо тривалість стану сплинула та є ввод руху, стан переключається на Moving. Якщо вводу немає, стан переключається на Default, а вектор швидкості обмежується значенням ExitSpeedLimitWhenMovementIsZero.

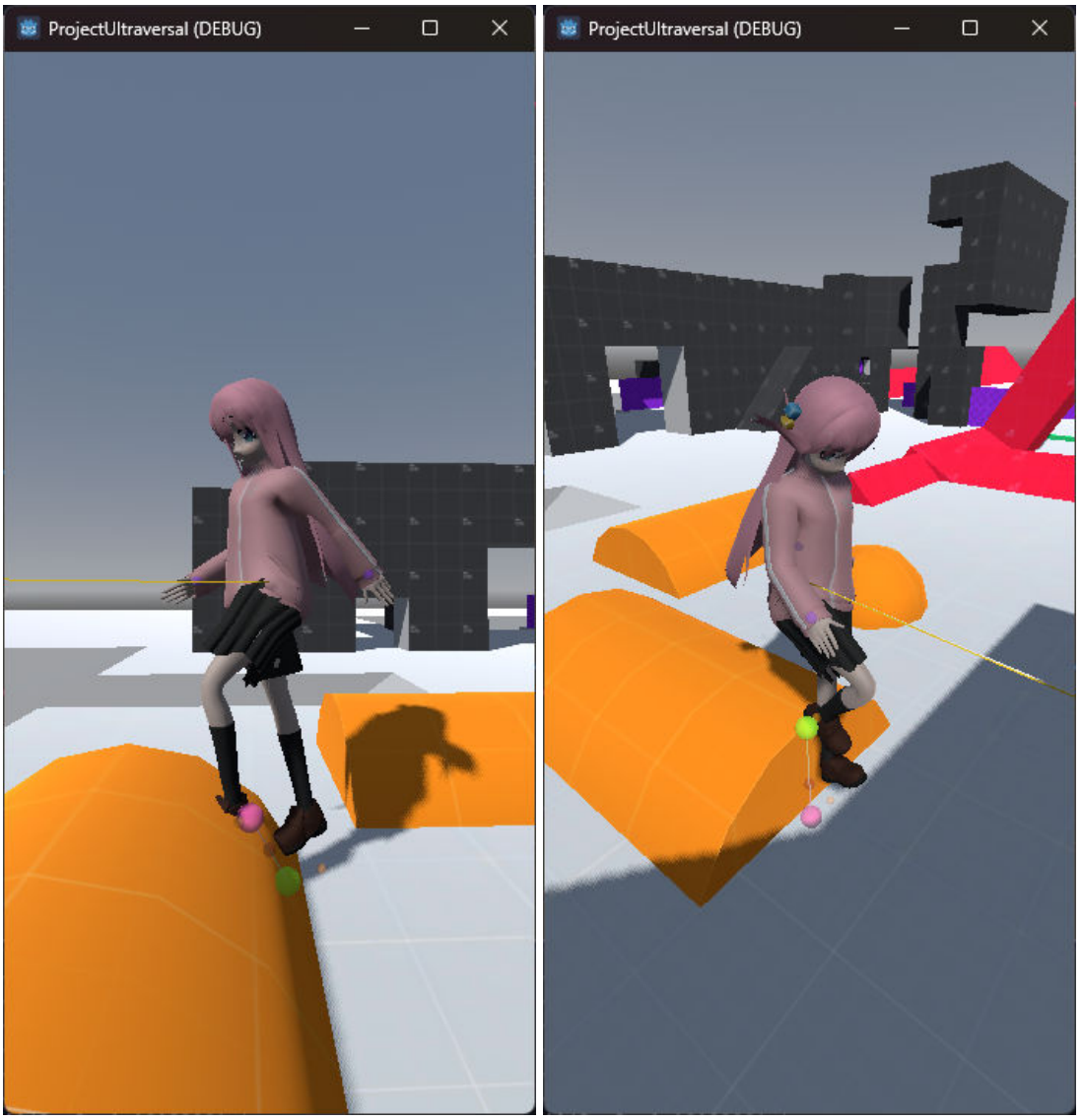


Рис. 3.7. Демонстрація анімацій стану Stepping

Анімації наведено на рисунку 3.7. Драйверна ієрархія була побудована наступним чином:

- DriverLayer (Base)
  - **Left Foot Position**
    - Vector3Driver
      - *Vector3DampingFunction*
  - **Right Foot Position**
    - Vector3Driver
      - *Vector3DampingFunction*
  - **Left Hand Position**
    - Vector3LocalSpaceTransformer
      - Vector3PassThroughDriver
  - **Right Hand Position**
    - Vector3LocalSpaceTransformer
      - Vector3PassThroughDriver

3.2.4. Grinding

В цьому стані гравець зупиняється, ковзаючи по поверхні. Домішками стану є: IPhysics. Параметрами конфігурації стану є:

- VelocitySmothness – параметр функції згладжування швидкості.
- FootRayCastVRange – мінімальний та максимальний вертикальний зсув рейкастів поверхні.
- MaxSlopeAngle – максимальний кут нахиилу поверхні.
- SlopeSpeedAdjustmentCurve – крива залежності сповільнення гравця від куту нахилу поверхні.
- BodyZOffsetRange – мінімальне та максимальне значення зсуву тіла гравця.
- BodyZOffsetSpeed – швидкість зсуву тіла гравця.

Під час входу в стан, занотовується напрямок руху та кут оберту тіла навколо осі Y. Це необхідно для визначення ведучої ноги та коректного відображення анімації.

Під час кожного кадру перевіряється чи є ввод. Якщо так, то стан переключається на Moving. Далі розраховуються позиції ніг та рук відповідно до напрямку руху, початкового оберту тіла та оберту погляду камери. Існують чотири варіанти стану гравця:

1. Права нога є ведучою, гравець дивиться в напрямку руху (рис. 3.8).
2. Ліва нога є ведучою, гравець дивиться в напрямку руху (рис. 3.9).
3. Права нога є ведучою, гравець дивиться назад (рис. 3.10).
4. Ліва нога є ведучою, гравець дивиться назад (рис. 3.11).

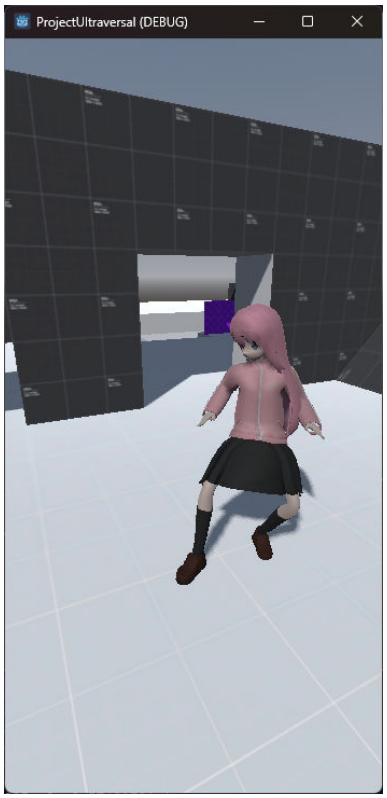


Рис. 3.8. Анімація Grinding, варіант 1

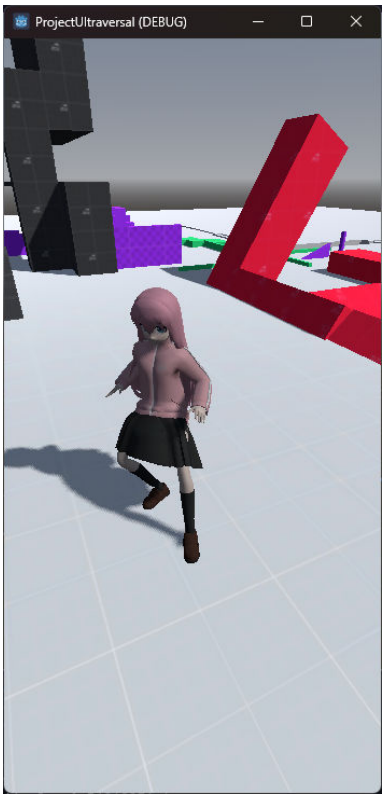


Рис. 3.9. Анімація Grinding, варіант 2

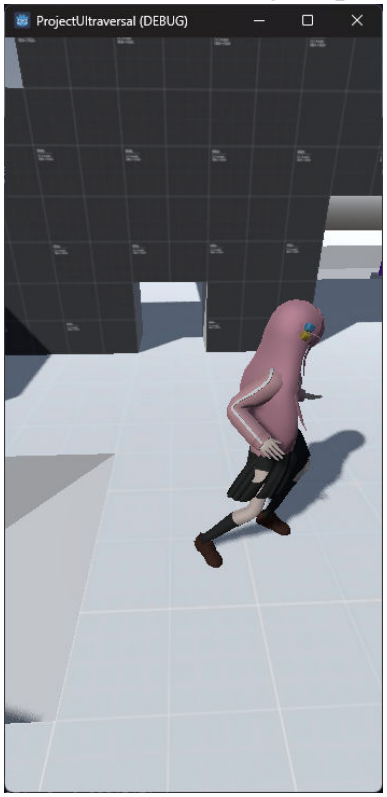


Рис. 3.10. Анімація Grinding, варіант 3

Рис. 3.11. Анімація Grinding, варіант 4

Знаючи цільові координати ніг в горизонтальній площині, виконуються два рейкасти та визначаються вертикальні координати (рис. 3.12).

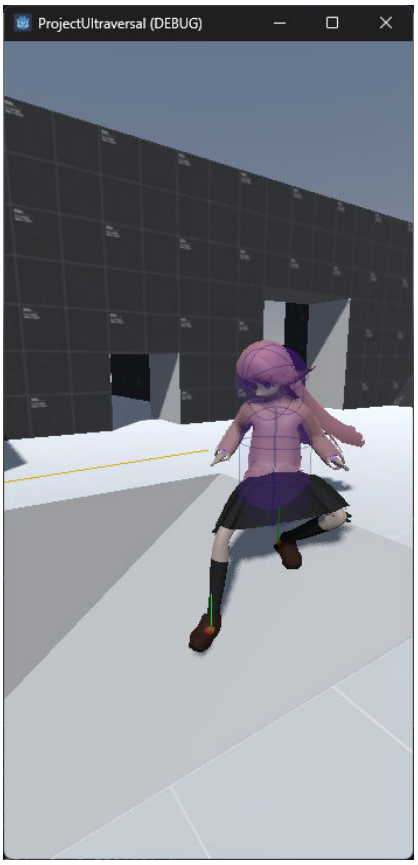


Рис. 3.12. Адаптація моделі до нахилу поверхні

Драйверна ієрархія була побудована наступним чином:

- DriverLayer (Base)
  - **Core X Offset**
    - SingleDriver
      - *SingleDampingFunction { Smoothness: 10 }*
  - **Core Z Offset**
    - SingleDriver
      - *SingleDampingFunction { Smoothness: 10 }*
  - **Core Tilt**
    - SingleDriver
      - *SingleConstantFunction { Value: -0.2 }*
      - *SingleAngleDampingFunction { Smoothness: 10 }*
  - **Left Foot Position**
    - Vector3FixedTimeInterpolationDriver
      - Vector3CodeControlledDriver
  - **Right Foot Position**
    - Vector3FixedTimeInterpolationDriver
      - Vector3CodeControlledDriver
  - **Left Knee Y Rotation**
    - Single Driver

        - *SingleAngleDampingFunction*
- **Right Knee Y Rotation**
    - Single Driver
        - *SingleAngleDampingFunction*
- **Left Hand Position**
    - Vector3FixedTimeInterpolationDriver
        - Vector3CodeControlledDriver
- **Right Hand Position**
    - Vector3FixedTimeInterpolationDriver
        - Vector3CodeControlledDriver

3.2.5. Jumping

В цьому стані гравець стрибає вверх. Домішками стану є: IPhysics, INaturalBodyRotation. Параметрами конфігурації стану є:

- JumpStrength – початкова вертикальна швидкість.
- ForwardBoost – додаткова горизонтальна швидкість при стрибку вперед.
- ForwardBoostThreshold – поріг, необхідний для активації горизонтального пришвидшення.
- Gravity – вертикальне сповільнення.
- MovementManipulationFactor – коефіцієнт контролю над рухом в повітрі.
- HorizontalVelocityDamping – горизонтальне сповільнення.

Під час входу в стан відбуваються дві дії:

1. Обраховується базова горизонтальна швидкість гравця BaseVelocityH. Якщо скалярний добуток поточної горизонтальної швидкості та вектору вводу лежить у межах порогу ForwardBoostThreshold, то до горизонтальної швидкості додається ForwardBoost.
2. Встановлюється вертикальна швидкість гравця зі значення JumpStrength.

Під час кожного кадру відбуваються наступні дії:

1. BaseVelocityH приводиться ближче до нуля за формулою згладжування. Фактор – HorizontalVelocityDamping.
2. Горизонтальна швидкість гравця встановлюється як BaseVelocityH + MovementInput * MovementManipulationFactor.

3. До вертикальної швидкості гравця додається значення Gravity помножене на дельту часу.
4. Якщо вертикальна швидкість перетнула 0 та стала негативною – гравець досяг вершини стрибка, і стан переключається на Falling.

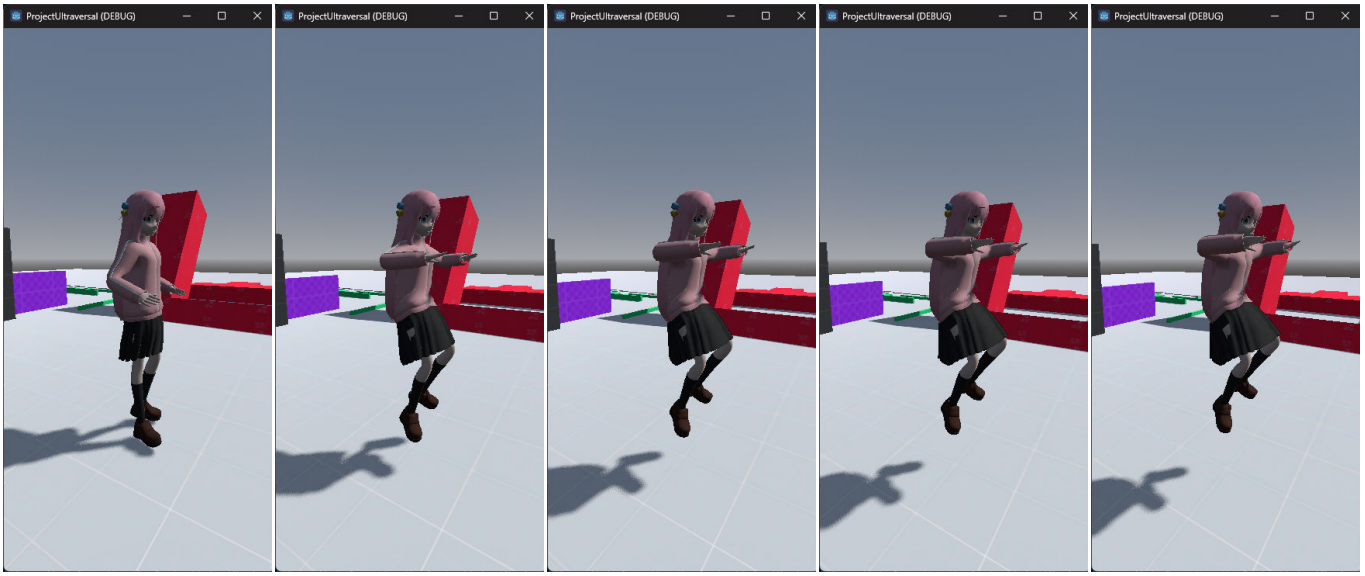


Рис. 3.13. Демонстрація анімації стану Jumping

Розкадровку анімації наведено на рисунку 3.13. Драйверна ієрархія була побудована наступним чином:

- DriverLayer (Base)
  - **Core Y Offset**
    - SingleDriver
      - *SingleDampingFunction { Smoothness: 30 }*
  - **Core Tilt**
    - SingleDriver
      - *SingleAngleDampingFunction { Smoothness: 10 }*
  - **Spine Curl**
    - Single Driver
      - *SingleAngleConstantFunction { Value: -40° }*
      - *SingleAngleDampingFunction { Smoothness: 5 }*
  - **Left Foot Position**
    - Vector3LocalSpaceTransformer
      - Vector3Driver
        - *Vector3ConstantFunction { Value: (-0.1, 0.3, 0) }*
        - *Vector3DampingFunction*
  - **Right Foot Position**
    - Vector3LocalSpaceTransformer
      - Vector3Driver
        - *Vector3ConstantFunction { Value: (0.1, 0.2, 0) }*

            - *Vector3DampingFunction*
    - **Left Knee Y Rotation**
        - SingleDriver
            - *SingleAngleConstantFunction { Value: -15° }*
            - *SingleAngleDampingFunction*
    - **Right Knee Y Rotation**
        - SingleDriver
            - *SingleAngleConstantFunction { Value: 15° }*
            - *SingleAngleDampingFunction*
    - **Left Hand Position**
        - Vector3LocalSpaceTransformer
            - Vector3Driver
                - *Vector3ConstantFunction { Value: (-0.3, 1.4, -0.3) }*
                - *Vector3DampingFunction { Smoothness: 30 }*
    - **Right Hand Position**
        - Vector3LocalSpaceTransformer
            - Vector3Driver
                - *Vector3ConstantFunction { Value: (0.3, 1.4, -0.3) }*
                - *Vector3DampingFunction { Smoothness: 30 }*
    - **Left Elbow Y Rotation**
        - SingleDriver
            - *SingleAngleConstantFunction { Value: 60° }*
            - *SingleAngleDampingFunction*
    - **Right Elbow Y Rotation**
        - SingleDriver
            - *SingleAngleConstantFunction { Value: -60° }*
            - *SingleAngleDampingFunction*

3.2.6. Falling

В цьому стані гравець падає вниз. Домішками стану є: IPhysics, INaturalBodyRotation. Параметрами конфігурації стану є:

- Gravity – вертикальне прискорення.
- MaxYVelocity – ліміт вертикальної швидкості.
- MovementManipulationFactor – коефіцієнт контролю над рухом в повітрі.
- HorizontalVelocityDamping – горизонтальне сповільнення.
- FloorRayCastStartYOffset – вертикальне зміщення рейкастів знаходження підлоги.
- MaxFloorAngle – максимальний кут нахилу підлоги.

Під час входу в стан, горизонтальна швидкість гравця записується в змінну BaseVelocityH.

Під час кожного кадру відбуваються наступні дії:

1. Аналогічно до стану Jumping, горизонтальна швидкість наближається до нуля та до неї додається ввод.
2. Вертикальна швидкість обраховується за формулою:

$$vVelocity = max(vVelocity + (g * \Delta), vVelocity_{max})$$

3. Використовується метод LocateFloor для знаходження підлоги. Початкове вертикальне зміщення рейкастів встановлюється параметром FloorRayCastStartYOffset. Кінцеве вертикальне зміщення обраховується за формулою:

$$FloorRayCastEndYOffset = \frac{vVelocity}{PhysicsTicksPerSecond} - 0,025$$

4. Якщо була знайдена центральна комірка або хоча б одна пара комірок периметру, стан змінюється на LandingContact.

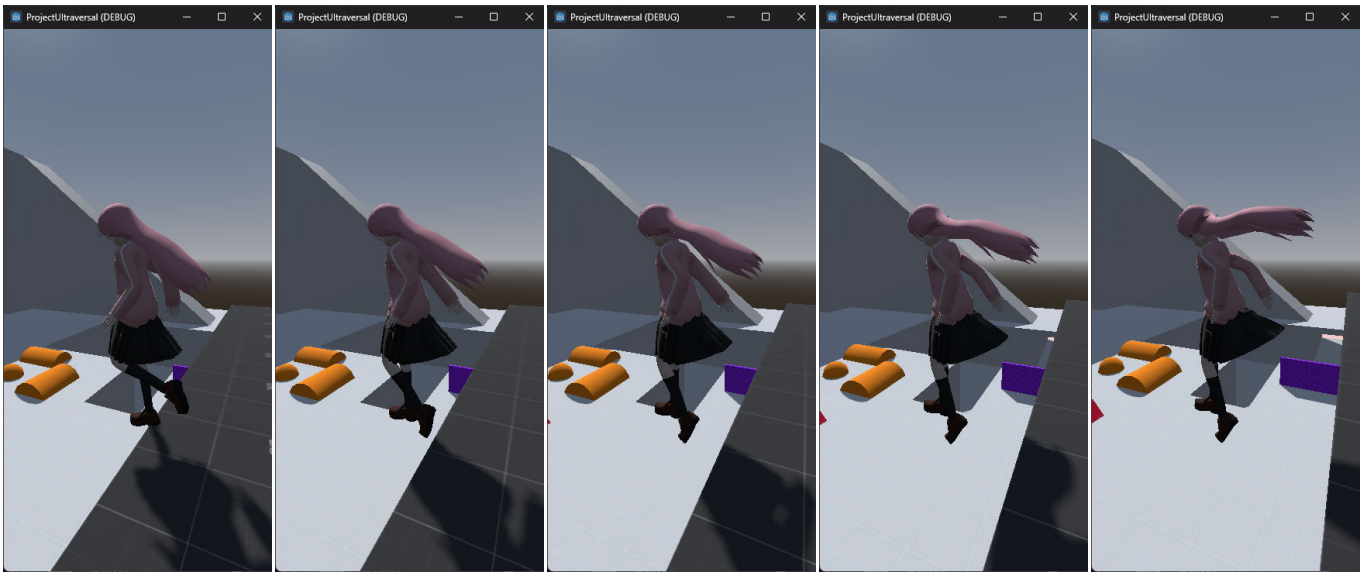


Рис. 3.14. Демонстрація анімації стану Falling

Розкадровку анімації наведено на рисунку 3.14. Драйверна ієрархія була побудована наступним чином:

- DriverLayer (Base)
  - **Core Y Offset**
    - SingleDriver
      - *SingleDampingFunction { Smoothness: 5 }*

- **Core Tilt**
  - SingleDriver
    - *SingleAngleConstantFunction { Value: -20° }*
    - *SingleAngleDampingFunction { Smoothness: 3 }*
- **Spine Curl**
  - Single Driver
    - *SingleAngleDampingFunction { Smoothness: 6 }*
- **Left Foot Position**
  - Vector3LocalSpaceTransformer
    - Vector3Driver
      - *Vector3ConstantFunction { Value: (-0.1, 0, 0) }*
      - *Vector3DampingFunction*
- **Right Foot Position**
  - Vector3LocalSpaceTransformer
    - Vector3Driver
      - *Vector3ConstantFunction { Value: (0.1, 0, -0.2) }*
      - *Vector3DampingFunction*
- **Left Hand Position**
  - Vector3LocalSpaceTransformer
    - Vector3Driver
      - *Vector3ConstantFunction { Value: (-0.3, 1, 0.1) }*
      - *Vector3DampingFunction { Smoothness: 30 }*
- **Right Hand Position**
  - Vector3LocalSpaceTransformer
    - Vector3Driver
      - *Vector3ConstantFunction { Value: (0.3, 1, 0.1) }*
      - *Vector3DampingFunction { Smoothness: 2 }*
- **Left Elbow Y Rotation**
  - SingleDriver
    - *SingleAngleConstantFunction { Value: 40° }*
    - *SingleAngleDampingFunction { Smoothness: 7 }*
- **Right Elbow Y Rotation**
  - SingleDriver
    - *SingleAngleConstantFunction { Value: -40° }*
    - *SingleAngleDampingFunction { Smoothness: 7 }*

3.2.7. LandingContact

В цьому стані гравець починає торкатись підлоги після падіння. В залежності від вводу та швидкості руху, стан буде переключено на інший. Домішкою стану є ITimeDuration. Параметрами конфігурації стану є:

1. MaxDuration – максимальна тривалість стану, якщо немає вводу.
2. ExcessiveYVelocityThreshold – поріг надмірної вертикальної швидкості.

3. CriticalYVelocityThreshold – поріг критичної вертикальної швидкості.

Під час кожного кадру відбуваються наступні дії:

1. Якщо вертикальна швидкість не перевищує надмірний поріг ExcessiveYVelocityThreshold, стан одразу перемикається на Moving, дозволяючи гравцю одразу почати рух після невеликих падінь.
2. Очікується сплив тривалості стану MaxDuration. В цей період здібності мають можливість переключити стан на інший, якщо є необхідний ввід.
3. Після спливу встановленого часу стан перемикається або на LandingHarshly (якщо вертикальна швидкість перевищує CriticalYVelocityThreshold) або на Moving, але з врахуванням нівечення, нанесеного надмірною швидкістю.

3.2.8. LandingHarshly

Перехід до стану LandingHarshly відбувається автоматично після невдалого приземлення (спливу часу стану LandingContact). Домішкою стану є ITimeDuration. Цей стан має фіксовану тривалість, задану параметром конфігурації Duration. Після спливу часу, стан переключається на Crouching.

Рис. 3.15. Демонстрація анімації стану LandingHarshly

Розкадровку анімації наведено на рисунку 3.15. Драйверна ієрархія була побудована наступним чином:

- DriverLayer (Base)
  - **Core Y Offset**
    - SingleDriver
      - *SingleConstantFunction { Value: -0.65 }*
      - *SingleDampingFunction { Smoothness: 5 }*
  - **Core Tilt**
    - SingleDriver
      - *SingleAngleConstantFunction { Value: -20° }*
      - *SingleAngleDampingFunction*
  - **Spine Curl**
    - Single Driver
      - *SingleAngleConstantFunction { Value: -90° }*
      - *SingleAngleDampingFunction*
  - **Left Foot Position**
    - Vector3FixedTimeInterpolationDriver
      - Vector3LocalSpaceTransformer
        - Vector3Driver
          - *Vector3ConstantFunction { Value: (0.0, 0.1, 0.25) }*
  - **Right Foot Position**
    - Vector3FixedTimeInterpolationDriver
      - Vector3LocalSpaceTransformer
        - Vector3Driver
          - *Vector3ConstantFunction { Value: (0.1, 0.0, -0.1) }*
  - **Left Knee Y Rotation**
    - SingleDriver
      - *SingleAngleConstantFunction { Value: -25° }*
      - *SingleAngleDampingFunction*
  - **Left Hand Position**
    - Vector3FixedTimeInterpolationDriver
      - Vector3LocalSpaceTransformer
        - Vector3Driver
          - *Vector3ConstantFunction { Value: (-0.1, 0.0, -0.2) }*
  - **Right Hand Position**
    - Vector3FixedTimeInterpolationDriver
      - Vector3LocalSpaceTransformer
        - Vector3Driver
          - *Vector3ConstantFunction { Value: (0.1, 0.45, -0.2) }*
  - **Right Elbow Y Rotation**
    - SingleDriver
      - *SingleAngleConstantFunction { Value: -40° }*
      - *SingleAngleDampingFunction*

3.2.9. LandingSoftly

М’яке приземлення використовується атлетами для погашення вертикальної швидкості при приземленні з помірної висоти. Перехід до стану LandingSoftly здійснюється Здібністю LandSoftly.

Під час кожного кадру здібність перевіряє наявністю трьох умов:

1. Наявний ввод “присісти”.
2. Гравець знаходиться в стані LandingContact.
3. Вертикальна швидкість гравця не перевищує ліміт (landingContactState.IsYVelocityExcessive = false).

Якщо ці умови виконуються, стан переключається в LandingSoftly. Стан LandingSoftly має фіксовану тривалість. Домішкою стану є ITimeDuration. Його параметрами конфігурації є:

1. float Duration – тривалість стану.
2. Curve BodyMovementCurve – крива переміщення тіла гравця.
3. float ExitSpeed – швидкість на виході зі стану.

При вході в стан поточна позиція гравця вносить в змінну InitialPosition. Під час кожного кадру, позиція гравця розраховується за формулою:

$pos = lerp(pos_i, pos_i + (dir \times C(1)), C(1 - \frac{t_l}{t_d}))$, де:

pos_i – вектор початкової позиція гравця;

dir – вектор напрямку руху гравця;

C – функція кривої BodyMovementCurve;

t_l – залишок тривалості стану;

t_d – повна тривалість стану.

Далі перевіряється чи сплив час стану. Якщо так, стан гравця переключається на Moving, а швидкість гравця розраховується як:

$vel = dir \times v_e$, де:

v_e – швидкість виходу зі стану ExitSpeed.

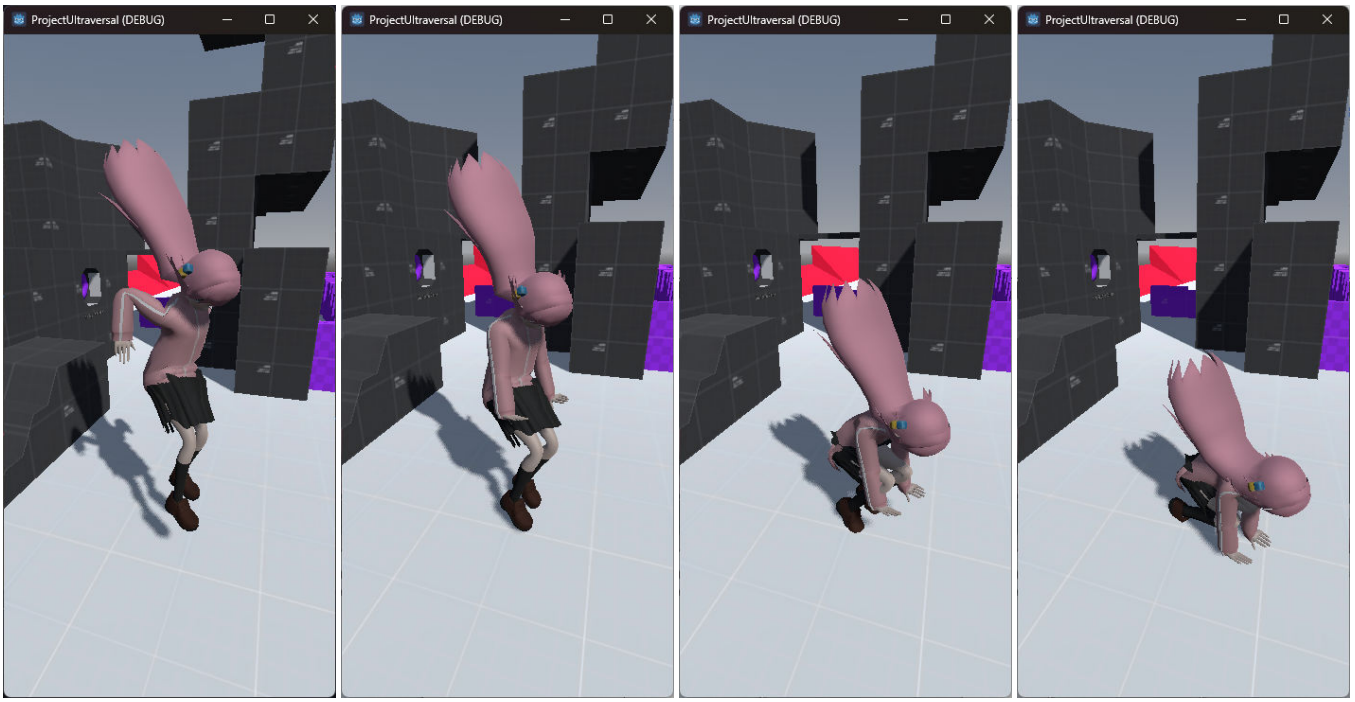


Рис. 3.16. Демонстрація анімації стану LandingSoftly

Розкадровку анімації наведено на рисунку 3.16. Драйверна ієрархія була побудована наступним чином:

- DriverLayerSet
  - DriverLayer (Base)
    - **Core Y Offset**
      - SingleDriver
        - *SingleTimeCurveFunction*
    - **Core Z Offset**
      - Single Driver
        - *SingleTimeCurveFunction*
    - **Core Tilt**
      - SingleDriver
        - *SingleTimeCurveFunction*
        - *SingleToRadFunction*
    - **Spine Curl**
      - SingleDriver
        - *SingleTimeCurveFunction*
        - *SingleToRadFunction*
    - **Left Foot Position**
      - Vector3LocalSpaceTransformer
        - Vector3Driver
          - *Vector3ConstantFunction { Value: (-0.1, 0.05, -0.1) }*
          - *Vector3DampingFunction*
    - **Right Foot Position**

        - Vector3LocalSpaceTransformer
            - Vector3Driver
                - *Vector3ConstantFunction { Value: (0.1, 0.05, -0.1) }*
                - *Vector3DampingFunction*
    - **Left Hand Position**
        - Vector3LocalSpaceTransformer
            - Vector3Driver
                - *Vector3ConstantFunction { Value: (-0.1, 0, -0.5) }*
                - *Vector3DampingFunction*
    - **Right Hand Position**
        - Vector3LocalSpaceTransformer
            - Vector3Driver
                - *Vector3ConstantFunction { Value: (0.1, 0, -0.5) }*
                - *Vector3DampingFunction*

3.2.10. Crouching

Як і Default, цей стан наслідує клас StationaryStateBase. В ньому персонаж стоїть на місці присівши. Домішкою стану є ICrouching.

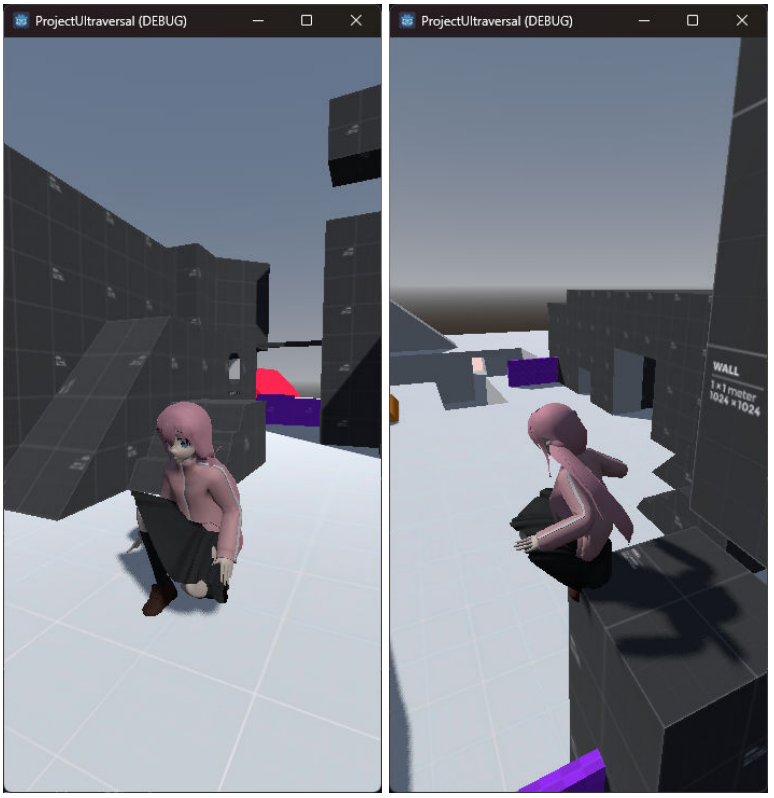


Рис. 3.17. Демонстрація анімацій стану Crouching

Розкадровку анімації наведено на рисунку 3.17. Драйверна ієрархія була побудована наступним чином:

- DriverLayerSet

- DriverLayer (Base)
  - **Core Y Offset**
    - SingleUnevenGroundTransformer
      - SingleDriver
        - *SingleConstantFunction { Value: -0.65 }*
        - *SingleDampingFunction { Smoothness: 9 }*
  - **Core Z Offset**
    - Single Driver
      - *SingleConstantFunction { Value: 0.1 }*
      - *SingleDampingFunction { Smoothness: 8 }*
  - **Core Tilt**
    - SingleDriver
      - *SingleAngleDampingFunction { Smoothness: 9 }*
  - **Spine Curl**
    - Single Driver
      - *SingleAngleConstantFunction { Value: -45° }*
      - *SingleAngleDampingFunction { Smoothness 5 }*
  - **Left Foot Position**
    - Vector3Driver
      - *Vector3DampingFunction*
  - **Right Foot Position**
    - Vector3Driver
      - *Vector3DampingFunction*
  - **Left Knee Y Rotation**
    - Single Driver
      - *SingleAngleConstantFunction { Value: -15° }*
      - *SingleAngleDampingFunction*
  - **Right Knee Y Rotation**
    - Single Driver
      - *SingleAngleConstantFunction { Value: 15° }*
      - *SingleAngleDampingFunction*
  - **Left Hand Position**
    - Vector3LocalSpaceTransformer
      - Vector3Driver
        - *Vector3ConstantFunction { Value: (-0.225, 0.95, 0) }*
        - *Vector3DampingFunction { Smoothness: 10 }*
  - **Right Hand Position**
    - Vector3LocalSpaceTransformer
      - Vector3Driver
        - *Vector3ConstantFunction { Value: (0.3, 0.3, -0.055) }*
        - *Vector3DampingFunction { Smoothness: 10 }*
- ToggleableDriverLayer (Balancing)
  - **Left Foot Position**

        - Vector3LocalSpaceTransformer
            - Vector3Driver
                - *Vector3ConstantFunction { Value: (-0.1, 0.1, 0) }*
                - *Vector3DampingFunction*
    - **Right Foot Position**
        - Vector3LocalSpaceTransformer
            - Vector3Driver
                - Vector3ConstantFunction *{ Value: (0.1, 0.1, 0) }*
                - *Vector3DampingFunction*
    - **Left Hand Position**
        - Vector3LocalSpaceTransformer
            - Vector3Driver
                - *Vector3ConstantFunction { Value: (-0.4, 0.5, 0) }*
                - *Vector3DampingFunction { Smoothness: 10 }*
    - **Right Hand Position**
        - Vector3LocalSpaceTransformer
            - Vector3Driver
                - *Vector3ConstantFunction { Value: (0.4, 0.5, 0) }*
                - *Vector3DampingFunction { Smoothness: 10 }*

3.3. Складні рухи та трюки

Складний рух гравця складається з наступних станів:

1. Vaulting – гравець долає перешкоду.
2. Rolling – перекат.
3. Sliding – підкат.
4. Crawling – повзання.
5. CatchingLedge – гравець чіпляється за виступ.
6. Clinging – гравець висить, упираючись ногами.
7. Hanging – гравець висить.
8. HangingRecovering – гравець висить, оговтуючись.
9. WallJumping – гравець відштовхується від стіни.
10. ClimbingUpMuscleUp – гравець підтягується з висячого положення.
11. ClimbingUpTopOut – гравець залазить на перешкоду після підтягування.

3.3.1. Vaulting

В цьому стані гравець долає перешкоду за допомогою стандартного трюка “monkey”. Домішкою стану є ITimeDuration. Перехід до стану Vaulting керується Здібністю Vault. Під час кожного кадру Vault перевіряє, чи знаходиться гравець в стані Jumping та наявність вводу руху вперед. Якщо так, то Vault виконує серію рейкастів, з метою знаходження позицій рук при виконанні трюку. Знайшовши підходящу позицію для лівої та правої руки (позиція не може бути всередині об’єкту та нахил поверхні не має перевищувати задане значення MaxSurfaceSteepness), Vault виконує сферичний shapecast по координатах гравця та сферичний motioncast по напрямку руху трюку, щоб визначити, чи є трюк можливим (рис. 3.18).

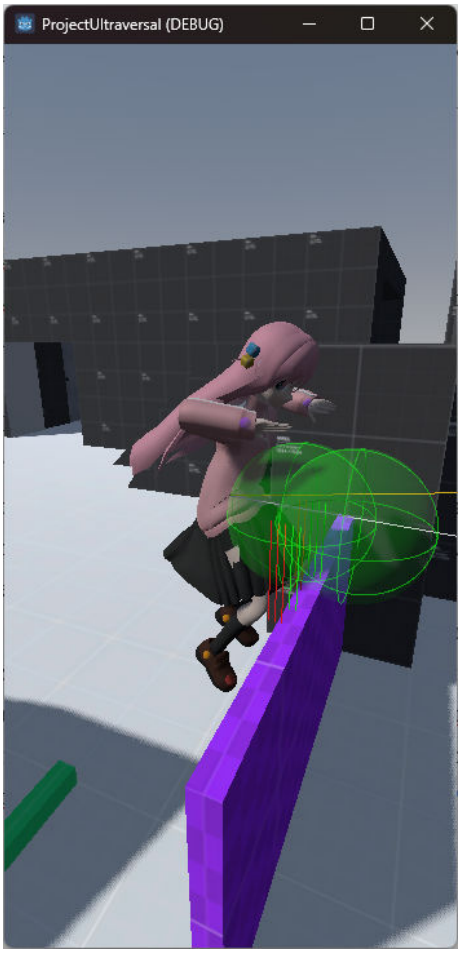


Рис. 3.18. Візуалізація raycast-ів та motioncast-ів

Якщо всі умови виконання трюку були виконані, знайдені позиції рук вносяться в об’єкт VaultData, що поділяється здібністю та станом, та стан переключається на Vaulting.

Стан Vaulting має фіксовану тривалість. При вході в стан Vaulting обраховується початкова та кінцева позиція гравця та встановлюються координати рук моделі. Під час кожного кадру перевіряється чи сплив час стану. Якщо так, то за допомогою motion-касту перевіряється наявність місця під гравцем. Якщо місця немає, стан переключається на Crouching, тобто гравець приземляється вприсядку на перешкоду. Якщо місце є, то стан переключається на Falling, тобто гравець повністю долає перешкоду та починає падати.

Якщо час ще не сплинув, виконується новий кадр анімації Vaulting. Позиція гравця лінійно інтерполюється від початкової до кінцевої, пропорційно пройденого часу та загальної тривалості стану.

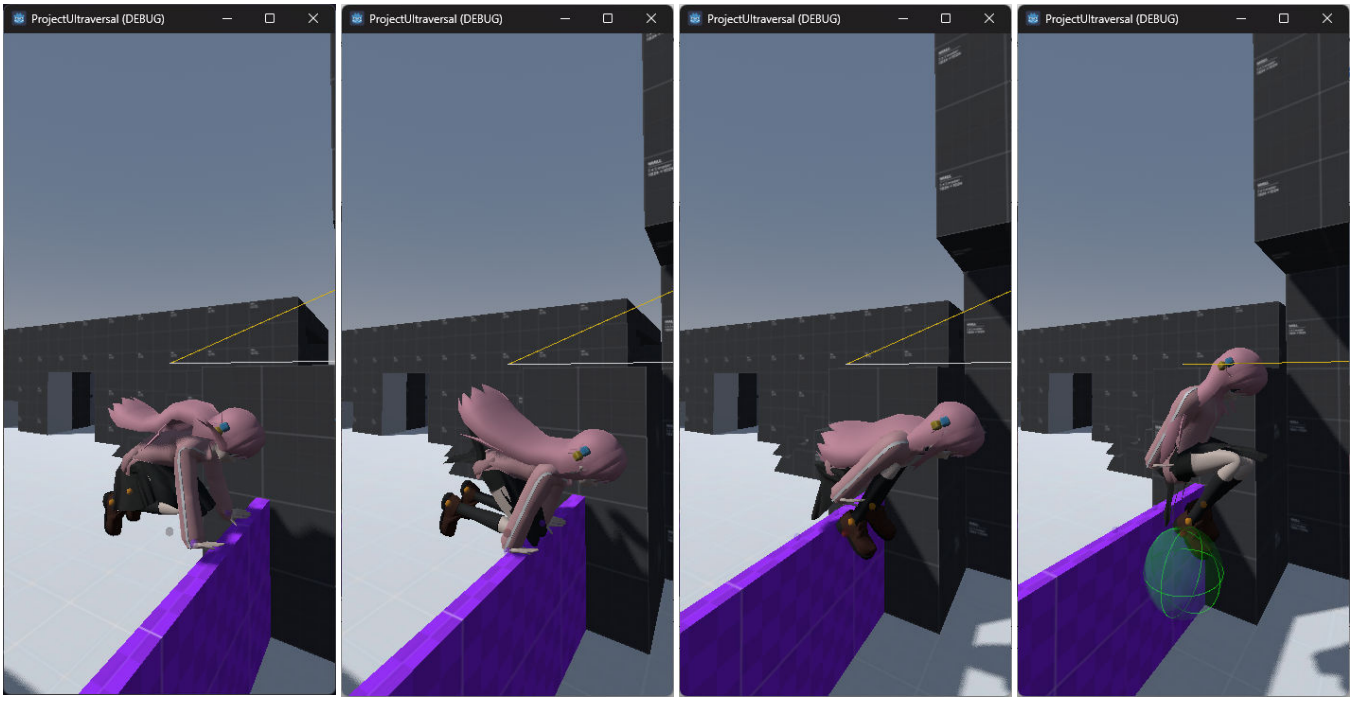


Рис. 3.19. Демонстрація анімації стану Vaulting

Розкадровку анімації наведено на рисунку 3.19. Драйверна ієрархія була побудована наступним чином:

- DriverLayer (Base)
  - **Core Y Offset**
    - SingleDriver
      - *SingleTimeCurveFunction*
  - **Core Z Offset**
    - SingleFixedTimeInterpolationDriver
      - SingleDriver
        - *SingleTimeCurveFunction*

- **Core Tilt**
  - SingleDriver
    - *SingleTimeCurveFunction*
    - *SingleToRadFunction*
- **Spine Curl**
  - SingleDriver
    - *SingleTimeCurveFunction*
    - *SingleToRadFunction*
- **Left Foot Position**
  - Vector3FixedTimeInterpolationDriver { Duration: 0.05 }
    - Vector3LocalSpaceTransformer
      - Vector3Driver
        - *Vector3TimeCurveFunction*
- **Right Foot Position**
  - Vector3FixedTimeInterpolationDriver { Duration: 0.05 }
    - Vector3LocalSpaceTransformer
      - Vector3Driver
        - *Vector3TimeCurveFunction*
- **Left Hand Position**
  - Vector3Driver
    - *Vector3DampingFunction*
- **Right Hand Position**
  - Vector3Driver
    - *Vector3DampingFunction*

3.3.2. Rolling

Стан Rolling це анімація перекату, який гравець може виконати або під час приземлення, або зі стану Crouching. Домішками стану є: IPhysics, ITimeDuration. Перехід до стану контролюється Здібностями Roll та LandRoll.

Здібність Roll дозволяє виконати перекат з сидячого положення. Під час кожного кадру перевіряється чи сидить гравець, та чи є ввод руху вперед. Якщо ввод руху вперед було натиснуто другий раз, стан переключається на Rolling. Здібність конфігурується властивістю MaxDoubleTapDelay.

Здібність LandRoll дозволяє виконати перекат після приземлення зі значної висоти, що використовується в паркурі для погашення швидкості при приземленні. Під час кожного кадру перевіряється чи знаходиться гравець в стані Falling або LandingContact. Якщо це так, та перевіряється ввод дії перекату. Після цього запускається таймер, що перевіряє чи є поточний стан станом LandingContact, та чи

перевищено критичну вертикальну швидкість. Якщо так, то стан переключається на Rolling. Такий алгоритм дозволяє налаштувати тривалість таймеру таким чином, щоб врахувати неточність (поспішність) вводу гравця. Це конфігурується властивістю MaxDurationAfterInput.

Стан Rolling має фіксовану тривалість. Він конфігурується наступними властивостями:

- float Duration – тривалість перекату.
- Curve SpeedCurve – крива швидкості гравця.
- Curve LookYAngleCurve крива оберту кута погляду гравця.
- float LookYAngleSyncDuration – час для інтерполяції з поточного кута погляду.

Під час входу в стан, оберт тіла синхронізується з кутом погляду. Під час кожного кадру, перевіряється чи сплив час тривалості стану. Якщо так, то стан переключається на Crouching. Якщо ні, то виконується анімація перекату, швидкість гравця встановлюється відповідно кривої SpeedCurve, а кут погляду синхронізується з кривою LookYAngleCurve.

```
float targetLookYAngle = LookYAngleCurve.Sample(ITimeDuration.Progress).ToRad();
float lookYAngleSyncProgress = ITimeDuration.TimeElapsed.Clamp(0,
LookYAngleSyncDuration) / LookYAngleSyncDuration;
Entity.LookAngle = Entity.LookAngle.CopyWith(y:
InitialLookYAngle.Lerp(targetLookYAngle, lookYAngleSyncProgress));
```

Синхронізація куту погляду є необхідною, адже при вході в стан гравець може дивитись під будь-яким кутом, а під час анімації погляд має співпадати з ротацією тіла гравця, тобто робити оберт в 360 градусів.

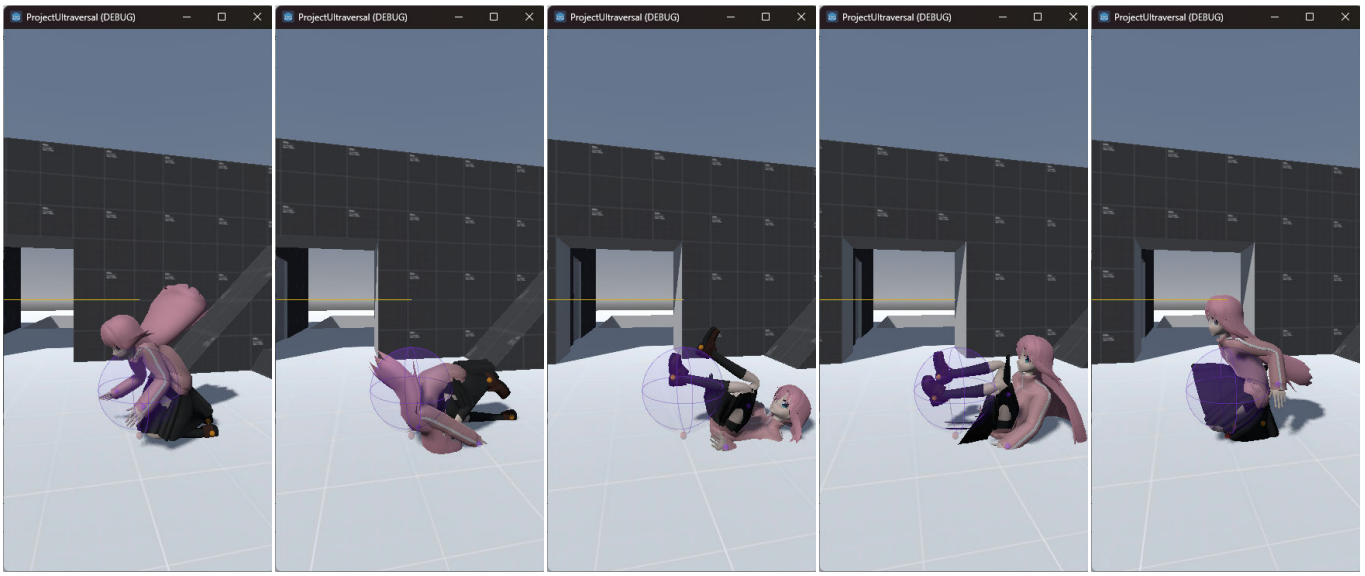


Рис. 3.20. Демонстрація анімації стану Rolling

Розкадровку анімації наведено на рисунку 3.20. Драйверна ієрархія була побудована наступним чином:

- DriverLayer (Base)
  - **Core Y Offset**
    - SingleDriver
      - *SingleTimeCurveFunction*
  - **Core Z Offset**
    - SingleFixedTimeInterpolationDriver { Duration: 0.05 }
      - SingleDriver
        - *SingleTimeCurveFunction*
  - **Core Tilt**
    - SingleDriver
      - *SingleTimeCurveFunction*
      - *SingleToRadFunction*
  - **Core Lean**
    - SingleDriver
      - *SingleTimeCurveFunction*
      - *SingleToRadFunction*
  - **Spine Twist**
    - SingleDriver
      - *SingleTimeCurveFunction*
      - *SingleToRadFunction*
  - **Spine Curl**
    - SingleDriver
      - *SingleTimeCurveFunction*
      - *SingleToRadFunction*
  - **Left Foot Position**
    - Vector3FixedTimeInterpolationDriver

            - Vector3LocalSpaceTransformer
                - Vector3Driver
                    - *Vector3TimeCurveFunction*
- **Right Foot Position**
    - Vector3FixedTimeInterpolationDriver
        - Vector3LocalSpaceTransformer
            - Vector3Driver
                - *Vector3TimeCurveFunction*
- **Left Hand Position**
    - Vector3FixedTimeInterpolationDriver { Duration: 0.05 }
        - Vector3LocalSpaceTransformer
            - Vector3Driver
                - *Vector3TimeCurveFunction*
- **Right Hand Position**
    - Vector3FixedTimeInterpolationDriver { Duration: 0.05 }
        - Vector3LocalSpaceTransformer
            - Vector3Driver
                - *Vector3TimeCurveFunction*

3.3.3. Sliding

Підкат дозволяє гравцю ефективно долати перешкоди, прослизнувши під ними, а також швидко долати відстані, ковзаючи по похилих поверхнях. Домішками стану є: IPhysics, ITimeDuration. Перехід до стану Sliding контролюється Здібністю Slide. Під час кожного кадру перевіряються три умови: гравець знаходиться в стані Moving, швидкість гравця більша або дорівнює параметру конфігурації MinSpeed, та кнопка вводу слайду натиснута.

Параметрами конфігурації стану є:

- float MinDuration – мінімальна тривалість слайду.
- float HorizontalVelocityDamping – сповільнення.
- float StopVelocityThreshold – нижній ліміт швидкості для виходу зі стану.
- float TopSpeed – базова максимальна швидкість слайду.
- float SpeedUpSmoothness – прискорення.
- float MaxSlopeAngle – максимальний кут нахилу поверхні.
- float ExitVelocityLimit – максимальна швидкість після виходу зі стану.
- float MaintainSpeedSlopeAngle – мінімальний кут нахилу поверхні, при якому швидкість слайду зберігається.

Під час кожного кадру в стані Sliding, виконується спроба знайти поверхню під гравцем. Для цього виконується серія вертикальний рейкастів, перпендикулярно до напрямку руху гравця (рис. 3.21). Якщо кількість успішних рейкастів дорівнює 0, стан переключається на Falling (поверхні немає, гравець починає падати). Якщо їх кількість більша за 0 але менша за встановлене значення SuccessfulRayCastThreshold, стан переключається на Crouching (поверхня не підходить для ковзання).

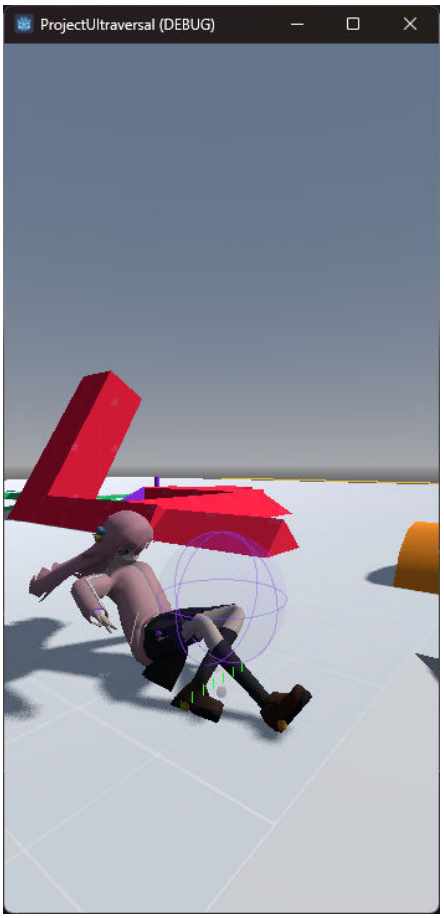


Рис. 3.21. Raycast-и та колізія стану

Далі найвища точка контакту з поверхнею порівнюється з попередньою позицією гравця. Якщо вона вища (гравець натрапив на виступ), стан переключається на Crouching. Якщо всі попередні умови виконані успішно, вертикальна позиція гравця синхронізується з найвищою точкою контакту та слайд продовжується. При цьому зчитується значення тертя (friction) з матеріалу поверхні. Надалі воно буде використовуватись для корекції швидкості гравця на слизьких та шорстких поверхнях.

Далі знаходиться кут нахилу поверхні, виходячи з поточної та попередньої позиції гравця.

```
CurrentSlopeAngle =
    Math.HalfPi - Vector3.Down.AngleTo(Entity.PreviousGlobalPosition
                                                .VectorTo(bodyPosition));
```

Якщо кут нахилу поверхні більший за мінімальний кут нахилу, необхідний для підтримки постійної швидкості, швидкість гравця поступово приводиться до TopSpeed, враховуючи коефіцієнт тертя та значення SpeedUpSmoothness.

```
Entity.Velocity = bodyVelocity = bodyVelocity.Damp(bodyVelocity.Normalized() *
TopSpeed / friction.Value, SpeedUpSmoothness, delta);
```

Якщо кут нахилу менший, то швидкість гравця поступово приводиться до 0, враховуючи кут нахилу, коефіцієнт тертя та параметр конфігурації HorizontalVelocityDamping.

```
float dampingFactor = Mathf.Max(0, Mathf.InverseLerp(MaintainSpeedSlopeAngle, 0,
CurrentSlopeAngle));
dampingFactor *= friction.Value;
Entity.Velocity = bodyVelocity = bodyVelocity.MoveToward(Vector3.Zero,
HorizontalVelocityDamping * dampingFactor);
```

Якщо після всіх розрахунків швидкість стала менша за StopVelocityThreshold, стан переключається на Crouching.

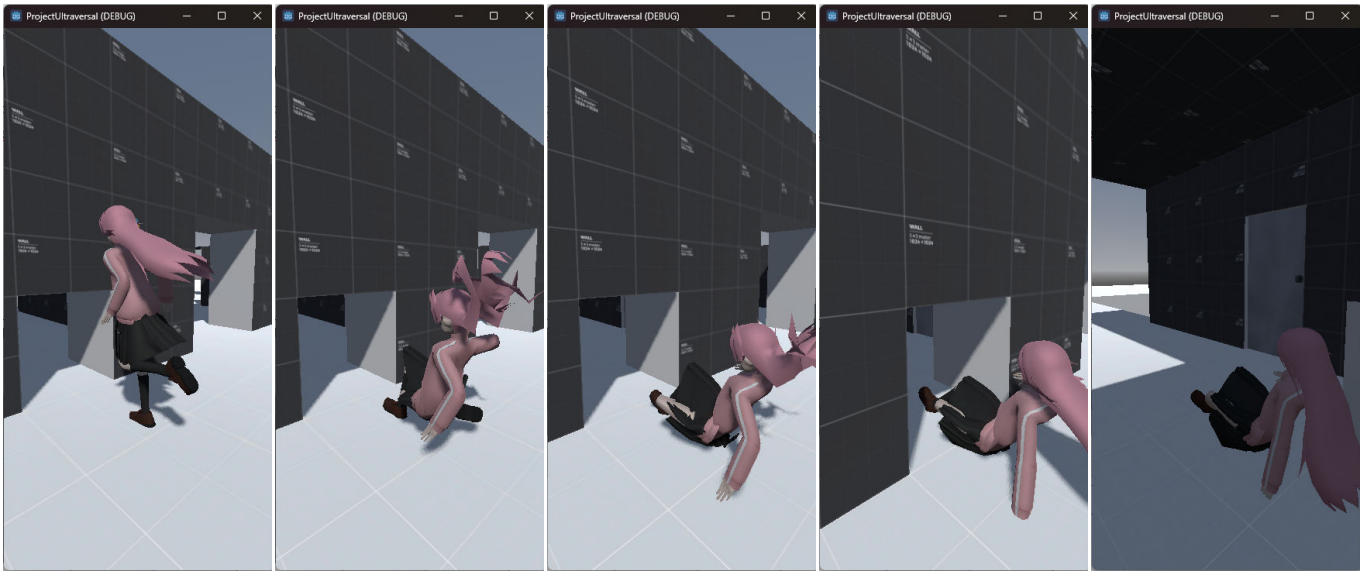


Рис. 3.22. Демонстрація анімації стану Sliding

Розкадровку анімації наведено на рисунку 3.22. Драйверна ієрархія була побудована наступним чином:

- DriverLayer (Base)
  - **Core Y Offset**
    - SingleDriver
      - *SingleConstantFunction { Value: -0.8 }*
      - *SingleDampingFunction { Smoothness: 30 }*
  - **Core Z Offset**
    - SingleDriver
      - *SingleConstantFunction { Value: 0.4 }*
      - *SingleDampingFunction*
  - **Core Tilt**
    - SingleSlopeAngleTransformer
      - SingleDriver
        - *SingleAngleConstantFunction { Value: 45° }*
        - *SingleAngleDampingFunction*
  - **Left Foot Position**
    - Vector3LocalSpaceTransformer
      - Vector3SlopeAngleTransformer
        - Vector3Driver
          - *Vector3ConstantFunction { Value: (0.2, 0, 0) }*
          - *Vector3DampingFunction*
  - **Right Foot Position**
    - Vector3LocalSpaceTransformer
      - Vector3SlopeAngleTransformer
        - Vector3Driver
          - *Vector3ConstantFunction { Value: (0.05, 0, -0.4) }*
          - *Vector3DampingFunction*
  - **Left Knee Y Rotation**
    - SingleDriver
      - *SingleAngleConstantFunction { Value: -20° }*
      - *SingleAngleDampingFunction*
  - **Left Hand Position**
    - Vector3LocalSpaceTransformer
      - Vector3SlopeAngleTransformer
        - Vector3Driver
          - *Vector3ConstantFunction { Value: (-0.3, 0, 0.7) }*
          - *Vector3DampingFunction*
  - **Right Hand Position**
    - Vector3LocalSpaceTransformer
      - Vector3SlopeAngleTransformer
        - Vector3Driver
          - *Vector3ConstantFunction { Value: (0.4, 0.4, 0.35) }*
          - *Vector3DampingFunction*

- **Right Elbow Y Rotation**
  - Single Driver
    - *SingleAngleConstantFunction { Value: -75° }*
    - *SingleAngleDampingFunction*

Драйвери SingleSlopeAngleTransformer та Vector3SlopeAngleTransformer це спеціалізовані драйвери для корекції оберту частин моделі при русі по похилій поверхні.

```
[GlobalClass]
public partial class SingleSlopeAngleTransformer : SingleTransformerDriverBase,
ISlopeAngleTransformer {
    public virtual float SlopeAngle { get; set; }

    public override float TransformInput(float input) => input + SlopeAngle;
    public override float TransformOutput(float output) => output - SlopeAngle;
}
```

```
IAnimator.UpdateDriversOfType<ISlopeAngleTransformer>(slopeAngleTransformer =>
slopeAngleTransformer.SlopeAngle = CurrentSlopeAngle);
```

3.3.4. Crawling

В стані Crawling гравець повзає по поверхні на чотирьох кінцівках. Домішками стану є: ICrouching, IPhysics. Це дозволяє ефективно пробиратись через важкодоступні місця, що часто зустрічаються в відеоіграх: обвали, вентиляційні шахти, печери тощо. Вхід в стан контролюється здібністю Crawl. Під час кожного кадру перевіряється наявністю вводу руху. Якщо гравець знаходиться в стані Crouching (присівши), стан одразу переключається на Crawling. Якщо гравець знаходиться в іншому підходящому стані (Default, Moving) то перевіряється наявність вводу дії присідання. Якщо так, стан переключається на Crawling.

Для реалізації логіки повзання було створено базовий клас LimbDataBase та два унаслідовані класи GroundedLimbData та SuspendedLimbData, а також енумерацію Limb. Ці класи дозволяють представити поточний стан кінцівок у процесі повзання.

```
public abstract record LimbDataBase { }
```

```
public record SuspendedLimbData(Vector3 InitialExtremityPosition, float TimeLeft,
Vector3? LastGoodRestPosition = null) : LimbDataBase;
public record GroundedLimbData(Vector3 ExtremityPosition, bool IsDeadlocked = false)
: LimbDataBase;
```

```
public enum Limb {
    LeftArm,
    RightArm,
    LeftLeg,
    RightLeg
}
```

Під час кожного кадру проводиться ітерація по кожній кінцівці. В залежності від стану кінцівки (GroundedLimbData чи SuspendedLimbData) виконуються різні дії:

- GroundedLimbData – кінцівка статична, знаходиться на поверхні. Спершу перевіряється наскільки далеко кінцівка знаходиться від її розташування за замовчуванням. Якщо відстань не перевищує задане значення ExtremityDisplacementThreshold, цикл переходить до наступної кінцівки. Якщо перевищує, то вважається що гравець знаходиться занадто далеко від кінцівки, і її необхідно “підтягнути” ближче.

  Тоді перевіряється стан суміжних кінцівок. Якщо одна з них знаходиться в повітрі, вважається що рух кінцвки неможливий, і встановюється флажок IsDeadlocked = true. Якщо всі суміжні кінцівки знаходяться на землі, поточний об’єкт кінцівки замінюється на SuspendedLimbData, тобто кінцівка переходить в рух.
- SuspendedLimbData – кінцівка знаходиться в русі у повітрі. Спершу перевіряється, чи є під кінцівкою поверхня, підходяща для переходу в стан Grounded. Якщо така поверхня знайдена, її координати вносяться у властивість LastGoodRestPosition. Якщо ні, то перевіряється існуюче значення LastGoodRestPosition. якщо його немає, і хоча б одна зі суміжних кінцівок є GroundedLimbData та має флажок IsDeadlocked, вважається що поточний стан є неможливим для повзання та стан переключається на Falling.

Якщо переключення стану не відбулось, LastGoodRestPosition вважається цільовою позицією кінцівки для наступних розрахунків. Далі властивість TimeLeft зменшується згідно дельти часу. Якщо час переміщення кінцівки вичерпано, об’єкт кінцівки замінюється на GroundedLimbData, тобто кінцівка успішно повертається на поверхню.

Для коректного відображення нахилу тіла гравця та розрахунку його позиції використовується властивість DetectedFloor. Вона представляє собою об’єкт Plane (площина), що є уявною поверхнею, розрахованою на основі даних розташування кінцівок. DetectedFloor оновлюється в момент переходу будь-якої кінцівки зі стану Suspended до Grounded, тобто в момент контакту з поверхнею. Площина будується на основі трьох точок, де дві точки – це координати кінцівки, що торкнулась землі, та протилежної до неї кінцівки, а третя точка це друга плюс вектор, перпендикулярний до напрямку руху гравця та вектора “вверх”.

Використовуючи горизонтальні координати гравця X та Z та площину DetectedFloor можна знайти вертикальну координат гравця Y за формулою:

$pos_y = \frac{D - normal_x \times pos_x - normal_z \times pos_z}{normal_y}$, де

$normal$ – нормаль площини;

D – відстань від початку координат до площини.

Для анімації кінцівок використовуються пари кривих SnapCurve та SnapYOffsetCurve. SnapCurve контролює інтерполяцію від початкової позиції (при переході в стан Suspended) до цільової. SnapYOffsetCurve контролює додаткове вертикальне зміщення. Зсув по кривих розраховується за формулою:

$progress = \frac{ExtremitySnapDuration - TimeLeft}{ExtremitySnapDuration}$, де

$ExtremitySnapDuration$ – параметр конфігурації стану.

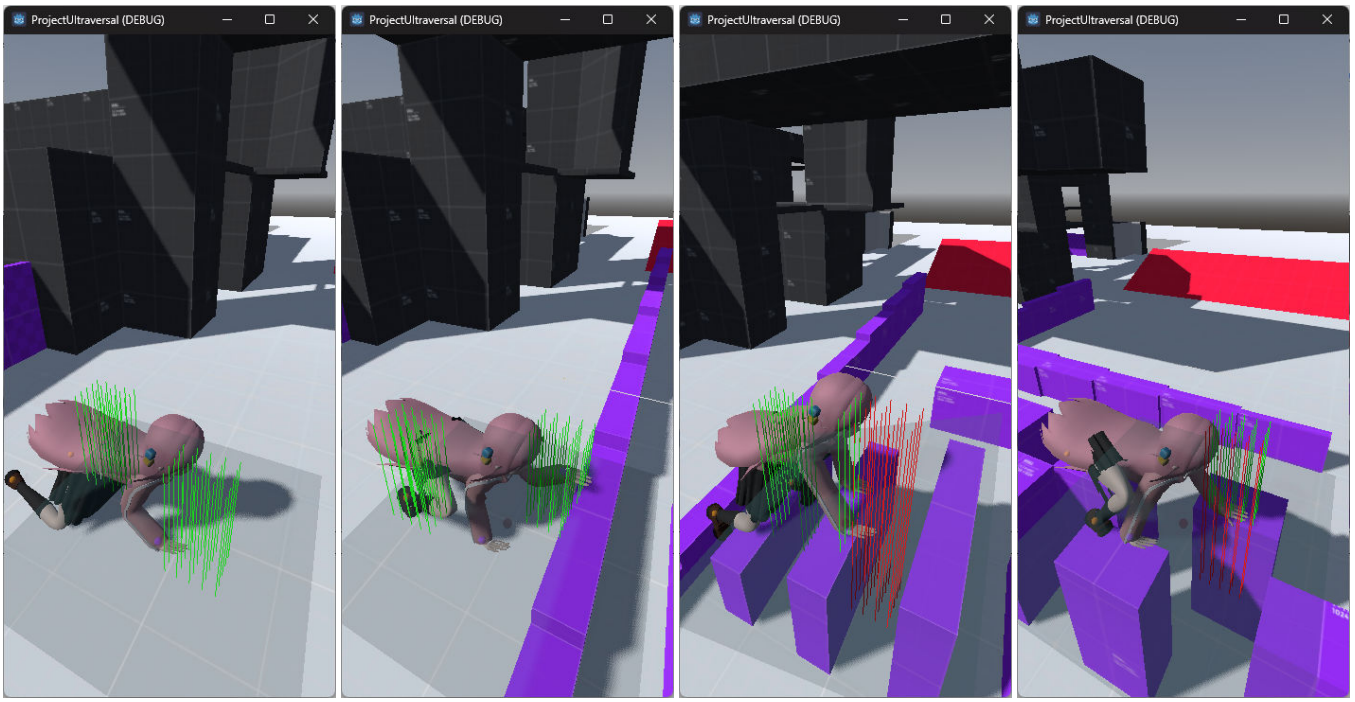


Рис. 3.23. Демонстрація анімації та raycast-ів стану Crawling

Розкадровку анімації наведено на рисунку 3.23. Драйверна ієрархія була побудована наступним чином:

- DriverLayerSet
  - DriverLayer (Base)
    - **Core Y Offset**
      - SingleDriver
        - *SingleDampingFunction { Smoothness: 5 }*
    - **Core Z Offset**
      - SingleDriver
        - *SingleDampingFunction*
    - **Core Tilt**
      - SingleDriver
        - *SingleAngleAdditionFunction { Value: -80° }*
        - *SingleAngleDampingFunction { Smoothness: 12 }*
    - **Spine Curl**
      - SingleDriver
        - *SingleAngleConstantFunction { Value: -20° }*
        - *SingleAngleDampingFunction*
    - **Left Foot Position**
      - Vector3FixedTimeInterpolationDriver
        - Vector3CodeControlledDriver
    - **Right Foot Position**
      - Vector3FixedTimeInterpolationDriver
        - Vector3CodeControlledDriver
    - **Left Knee Y Rotation**

    - SingleDriver
        - *SingleAngleConstantFunction { Value: -30° }*
        - *SingleAngleDampingFunction*
- **Right Knee Y Rotation**
    - SingleDriver
        - *SingleAngleConstantFunction { Value: 30° }*
        - *SingleAngleDampingFunction*
- **Left Hand Position**
    - Vector3LocalSpaceTransformer
        - Vector3Driver
            - *Vector3ConstantFunction { Value: (-0.225, 0.95, 0) }*
            - *Vector3DampingFunction { Smoothness: 10 }*
- **Right Hand Position**
    - Vector3LocalSpaceTransformer
        - Vector3Driver
            - *Vector3ConstantFunction { Value: (0.25, 0.95, -0.055) }*
            - *Vector3DampingFunction { Smoothness: 10 }*
- **Left Elbow Y Rotation**
    - SingleDriver
        - *SingleAngleConstantFunction { Value: 5° }*
        - *SingleAngleDampingFunction*
- **Right Elbow Y Rotation**
    - SingleDriver
        - *SingleAngleConstantFunction { Value: -5° }*
        - *SingleAngleDampingFunction*

ВИСНОВКИ

В процесі виконання даної кваліфікаційної роботи була розроблена низка алгоритмів та програмних систем, а також прототип ігрового продукту, що їх використовує. Розробленими системами є:

1. Контролер гравця.
2. Система анімації гуманоїдних 3D-моделей.
3. Система повтору.

Контролер гравця був побудований на основі двох патернів програмування: State та Component. Контролер складається з набору компонентів, які в свою чергу бувають двох видів: компонент-стан та компонент-здібність. Стани реалізують концепцію патерну State – лише один такий компонент може бути активним одночасно. Здібності можуть бути ввімкнуті та вимкнуті індивідуально, забезпечуючи можливість реалізації логіки, незалежної від поточного стану, та логіки, що керує станами.

Додатково було реалізовано можливість створювати домішки – спеціальні інтерфейси, що дозволяють додавати логіку в компоненти без необхідності використовувати наслідування. Логіка домішок також може бути автоматично підключена до подій компоненту.

Система анімації складається з двох підсистем – керування моделлю та драйвери анімації. Керування моделлю здійснюється за допомогою трьох класів – TorsoAnimator, HeadAnimator та LimbAnimator. Кожен з них реалізовує унікальний алгоритм, що забезпечує баланс між простотою використання та гнучкістю анімації.

TorsoAnimator дозволяє керувати тулубом моделі за допомогою параметрів CoreXOffset, CoreYOffset, CoreZOffset, CoreYaw, CoreTilt, CoreLean, SpineTwist та SpineCurl, що представляються собою кути нахилу та значення зміщення кісток тулубу.

HeadAnimator дозволяє контролювати оберт голови та шиї за допомогою параметру LookAngle та кривих LookYAngleHeadInfluenceCurve та LookYAngleNeckInfluenceCurve.

LimbAnimator реалізовує спеціальний алгоритм інверсної кінематики, що дозволяє розрахувати оберти кісток кінцівок виходячи з векторного значення позиції кінця ланцюга (руки/ступні).

Для створення анімації в часі було розроблено систему, що складається з драйверів та драйверних шарів. Драйвери являються ресурсами, що реалізовують функції трансформації значень з часом. Драйвери дозволяють встановлювати константе значення, згладжувати поточне значення до кінцевого, переводити значення з локального простору до зовнішнього, а також отримувати значення з кривих або напряму з ігрової логіки.

Драйверні шари групують набори драйверів та асоціюють їх з параметрами API керування моделлю. Окремі види шарів дозволяють змішувати, накладати та перемикати вкладені набори шарів, що дозволяє реалізувати складі накладені анімації.

Система драйверів анімації була інтегрована в базовий компонент стану гравця. Додатково в нього було інтегровано систему динамічної зміни колізії гравця, що дозволяє присвоїти кожному стану окрему форму колайдера.

Розроблена система повтору дозволяє зчитувати дані гри під час кожного кадру, зберігати їх у пам’яті або файлі, та потім відтворювати. Режим відтворення має свій користувацький інтерфейс, що дозволяє переглядати повтор покадрово, а також змінювати режим камери.

В систему повтору були додатково інтегровані інструменти відладки, що значно полегшують процес розробки 3D-ігор швидкого темпу. Розроблені інструменти складаються з двох ключових компонентів – оверлею та малювання. Оверлей показує набір значень в поточний момент часу – будь то кадр повтору, чи

поточний екземпляр гри. Інструменти малювання дозволяють створювати геометричні фігури у 3D-просторі, що в свою чергу дозволяє розробникам візуалізувати вектори, координати та інші невидимі властивості гри.

Побудований ігровий прототип поєднує в собі всі створені системи, та реалізує набір складних анімацій та переміщення гравця в просторі, основаних на принципах дисципліни паркуру.

Було успішно реалізовано наступні стани гравця:

1. Default
2. Moving
3. Stepping
4. Grinding
5. Jumping
6. Falling
7. LandingContact
8. LandingHarshly
9. LandingSoftly
10. Couching
11. Vaulting
12. Rolling
13. Sliding
14. Crawling

Цим було продемонстровано ефективність запропонованих систем, а також створено підґрунтя для розробки більш складних ігор та програмних продуктів, що включають складний рух у 3D-просторі, процедурні анімації та інструменти перегляду, аналізу та відладки стану застосунку в певний проміжок часу.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. State. *Game Programming Patterns* / R. Nystrom. URL: https://gameprogrammingpatterns.com/state.html (дата звернення: 02.09.2025).

2. State. *Dive Into Design Patterns* / A. Shvets. URL: https://refactoring.guru/design-patterns (дата звернення: 02.09.2025).

3. Animation Blend Trees. *Unity Documentation*. *Unity Manual*. 25.09.2025. URL: https://docs.unity3d.com/6000.2/Documentation/Manual/class-BlendTree.html (дата звернення: 03.09.2025).

4. Ahlström E., Holmqvist L., Goswami P. Comparing Traditional Key Frame and Hybrid Animation. *ACM SIGGRAPH Symposium on Computer Animation*. 2017.

5. Inverse Kinematics Techniques in Computer Graphics: A Survey / A. Aristidou та ін. *COMPUTER GRAPHICS forum*. 2017. С. 1–24.

6. Nierstrasz O., Tsichritzis D. Object-Oriented Software Composition. Campus 400, Maylands Avenue, Hemel Hempstead, Hertfordshire, HP2 7EZ : Prentice Hall International (UK) Ltd, 1995.

7. Phan N. Breath of the Wild Open World Analysis: Gravity to go Forward. *GameDeveloper*. *Blogs*. 01.04.2019. URL: https://www.gamedeveloper.com/design/breath-of-the-wild-open-world-analysis-gravity-to-go-forward (дата звернення: 04.09.2025).

8. Regan E. Assassin’s Creed: The Parkour Mechanics Of Every Game, Ranked. *GameRant*. 08.10.2023. URL: https://gamerant.com/assassins-creed-games-best-parkour/ (дата звернення: 04.09.2025).

9. The Great Debate: Rigidbody vs Character Controller for Player Movement in Unity. *Medium*. 25.05.2025. URL: https://sivakumar-prasanth.medium.com/the-great-debate-rigidbody-vs-character-controller-for-player-movement-in-unity-c551a52ff340 (дата звернення: 26.11.2025).

10. Basile P. J. Unity vs Unreal vs Godot: Finding Your Perfect Game Engine in 2025. *DEV Community*. 22.07.2025. URL:

https://dev.to/philipjohnbasile/unity-vs-unreal-vs-godot-finding-your-perfect-game-engine-in-2025-4ecg (дата звернення: 26.11.2025).
11. Souto N. Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects. *Toptal*. *Developers*. 21.04.2025. URL: https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects (дата звернення: 26.11.2025).

Додаток А

Код класу ComponentBase

```
using DynamicData; using DynamicData.Alias; using Godot; using System; using
System.Collections.Generic; using System.Diagnostics.CodeAnalysis; using System.Linq; using
System.Reactive.Linq; using System.Reflection;
namespace GEC;
public abstract partial class ComponentBase<TEntity> : Node where TEntity : Node, IEntity {
    protected static Action? StateQueueProcessor { get; set; }
#nullable disable
    public virtual TEntity Entity { get; protected set; }
    protected virtual Node Container { get; set; }
    protected virtual IObservableList<StateBase<TEntity>> States { get; set; }
    protected virtual IObservableList<AbilityBase<TEntity>> Abilities { get; set; }
#nullable enable
    protected virtual bool _ShouldActivateAfterInitialization { get; }
    protected virtual bool _IsActive => GetParent() == Container;
    protected virtual StateBase<TEntity> CurrentState =>
        States.Items.First(state => state.IsCurrent);
    protected virtual IEnumerable<AbilityBase<TEntity>> EnabledAbilities =>
        Abilities.Items.Where(ability => ability.IsEnabled);
    protected virtual IEnumerable<AbilityBase<TEntity>> DisabledAbilities =>
        Abilities.Items.Where(ability => ability.IsDisabled);
    protected virtual double TimeElapsedSinceActive { get; set; }
    protected virtual MixinHook[] _MixinHooks { get; init; }
    public ComponentBase() {
        if(this is IMixin<TEntity>) {
            Type[] interfaces = GetType().GetInterfaces();
            List<Type> mixins = [];
            foreach(Type @interface in interfaces) {
                if(!@interface.IsAssignableTo(typeof(IMixin<TEntity>))) { continue;}
                if(@interface == typeof(IMixin<TEntity>)) { continue; }
                if(mixins.Contains(@interface)) { continue; }
                mixins.Add(@interface);
            }
            _MixinHooks = [ ..mixins.SelectMany(mixin =>
mixin.GetMethods(BindingFlags.Instance | BindingFlags.Public |
BindingFlags.NonPublic).Where(methodInfo => !methodInfo.IsStatic && !methodInfo.IsAbstract
&& !methodInfo.ContainsGenericParameters && !methodInfo.IsConstructor &&
!methodInfo.IsSpecialName && methodInfo.ReturnType == typeof(void) &&
methodInfo.GetParameters().Length == 0 &&
methodInfo.GetCustomAttribute<MixinHookAttribute>(true) != null)).Select(methodInfo => {
```

```
MixinHookAttribute mixinHookAttribute =
methodInfo.GetCustomAttribute<MixinHookAttribute>(true)!;
return new MixinHook(mixinHookAttribute.HookEvent, mixinHookAttribute.HookType, () =>
methodInfo.Invoke(this, null), methodInfo.DeclaringType!); }) ];
        } else {
            _MixinHooks = [];
        }
    }
    #region Node Events
    public override void _Ready() {
        base._Ready();
        ProcessMode = ProcessModeEnum.Disabled;
        {
            Node parent = GetParent();
            Container = parent;
            void FindEntityRecursive() {
                if(parent is TEntity entity) {
                    Entity = entity;
                    return;
                }
                parent = parent.GetParent();
                FindEntityRecursive();
            }
            FindEntityRecursive();
        }
        Global._RegisterComponent(Entity, this);
        States = Global.GetComponents<StateBase<TEntity>>(Entity);
        Abilities = Global.GetComponents<AbilityBase<TEntity>>(Entity);
        Callable.From(_LateReady).CallDeferred();
        _InvokeHooks(HookEvent.Ready, HookType.Pre);
        OnReady();
        _InvokeHooks(HookEvent.Ready, HookType.Post);
        Log($"Component '{Name}' ({GetType().Name}) is ready!");
        if(_MixinHooks.Length > 0) {
            Log($"> Loaded mixins: {string.Join(", ", _MixinHooks.Select(mixinHook =>
mixinHook.Source).Distinct().Select(mixin => mixin.Name))}");
        }
    }
#pragma warning disable IDE1006 // Naming Styles
    protected virtual void _LateReady() {
        if(_ShouldActivateAfterInitialization) {
            ((IActivateableComponent) this)._OnActivated();
        } else {
```

```
            _DeactivateInternal();
        }
        ProcessMode = ProcessModeEnum.Inherit;
    }
#pragma warning restore IDE1006 // Naming Styles
    public override void _Process(double delta) {
        base._Process(delta);
        _InvokeHooks(HookEvent.Process, HookType.Pre);
        OnProcess(delta);
        _InvokeHooks(HookEvent.Process, HookType.Post);
        TimeElapsedSinceActive += delta;
    }
    public override void _PhysicsProcess(double delta) {
        base._PhysicsProcess(delta);
        _InvokeHooks(HookEvent.PhysicsProcess, HookType.Pre);
        OnPhysicsProcess(delta);
        _InvokeHooks(HookEvent.PhysicsProcess, HookType.Post);
    }
    #endregion
    protected virtual void OnReady() { }
    protected virtual void OnProcess(double delta) { }
    protected virtual void OnPhysicsProcess(double delta) { }
    protected virtual TState GetState<TState>()
        where TState : StateBase<TEntity> => GetStateOrNull<TState>()!;
    protected virtual TState? GetStateOrNull<TState>()
        where TState : StateBase<TEntity> =>
            States.Items.OfType<TState>().FirstOrDefault(state => state.GetType() ==
typeof(TState));
    protected virtual TAbility GetAbility<TAbility>()
        where TAbility : AbilityBase<TEntity> => GetAbilityOrNull<TAbility>()!;
    protected virtual TAbility? GetAbilityOrNull<TAbility>() where TAbility :
AbilityBase<TEntity> => Abilities.Items.OfType<TAbility>().FirstOrDefault(ability =>
ability.GetType() == typeof(TAbility));
    protected virtual bool Can<TAbility>() where TAbility : AbilityBase<TEntity> =>
            GetAbilityOrNull<TAbility>()?.IsEnabled ?? false;
    protected virtual void SwitchState<TState>() where TState :StateBase<TEntity> =>
            _SwitchStateInternal<TState>(true, default);
    protected virtual void QueueStateSwitch<TState>(IfAStateIsAlreadyQueued
ifAStateIsAlreadyQueued = IfAStateIsAlreadyQueued.DoNothing)
        where TState : StateBase<TEntity> =>
            _SwitchStateInternal<TState>(false, ifAStateIsAlreadyQueued);
    protected virtual void SwitchState(StateBase<TEntity> newState) =>
        _SwitchStateInternal(newState, true, default);
```

```
    protected virtual void QueueStateSwitch(StateBase<TEntity> newState,
IfAStateIsAlreadyQueued ifAStateIsAlreadyQueued = IfAStateIsAlreadyQueued.DoNothing) =>
_SwitchStateInternal(newState, false, ifAStateIsAlreadyQueued);
    protected virtual void _SwitchStateInternal<TState>(bool immediate,
IfAStateIsAlreadyQueued ifAStateIsAlreadyQueued) where TState : StateBase<TEntity> {
        TState? newState = GetState<TState>();
        if(newState == null) {
            LogError($"Failed to switch state by type '{typeof(TState).Name}': state not
found!");
            return;
        }
        _SwitchStateInternal(newState, immediate, ifAStateIsAlreadyQueued);
        return;
    }
    protected virtual void _SwitchStateInternal(StateBase<TEntity> newState, bool immediate,
IfAStateIsAlreadyQueued ifAStateIsAlreadyQueued) {
        QueuedStateSwitchData queuedStateSwitchData = new(this, newState);
        Log($"'{queuedStateSwitchData.Initiator.Name}' requested <{(immediate ? "immediate"
: "delayed")}> state switch from '{CurrentState.Name}' to
'{queuedStateSwitchData.TargetState.Name}'...");
        if(immediate) {
            _PerformStateSwitch(CurrentState, queuedStateSwitchData);
            return;
        }
        if(Global.IsStateQueuedFor(Entity)) {
            switch(ifAStateIsAlreadyQueued) {
                case IfAStateIsAlreadyQueued.DoNothing: { return; }
                case IfAStateIsAlreadyQueued.Overwrite: { break; }
                case IfAStateIsAlreadyQueued.Throw: { throw new Exception("Failed to queue
state switch: another state switch is already queued!"); }
            }
        }
        Global._SetQueuedStateSwitchData(Entity, queuedStateSwitchData);
        if(StateQueueProcessor != null) { return; }
        Callable.From(StateQueueProcessor = () => {
            StateQueueProcessor = null;
            foreach(TEntity entity in Global._Entities.Keys) {
                if(Global.TryGetQueuedStateSwitchData(entity, out QueuedStateSwitchData?
queuedStateSwitchData)) {
                  _PerformStateSwitch(Entity.CurrentState(), queuedStateSwitchData);
                }
            }
            Global._ClearQueuedStateSwitchDatas();
```

```
        }).CallDeferred();
    }
    protected virtual void _PerformStateSwitch(StateBase<TEntity> currentState,
QueuedStateSwitchData queuedStateSwitchData) {
        StateBase<TEntity> newState = queuedStateSwitchData.TargetState;
        if(currentState == newState) {
            LogError($"Failed to switch state to '{newState.Name}': state already active!");
            return;
        }
        CurrentStateSwitchData currentStateSwitchData =
            new(queuedStateSwitchData.Initiator, currentState, newState);
        Global._SetCurrentStateSwitchData(newState.Entity, currentStateSwitchData);
        ((IDeactivateableComponent) currentState)._OnDeactivating();
        currentState._DeactivateInternal();
        ((IDeactivateableComponent) currentState)._OnDeactivated();
        ((IActivateableComponent) newState)._OnActivating();
        newState._ActivateInternal();
        ((IActivateableComponent) newState)._OnActivated();
        Global._SetCurrentStateSwitchData(newState.Entity, null);
        Global._SetLastStateSwitchData(newState.Entity, currentStateSwitchData);
        Log($"State switched to '{newState.Name}'!");
    }
    protected virtual bool ToggleAbility<TAbility>() where TAbility : AbilityBase<TEntity> {
        TAbility? ability = GetAbility<TAbility>();
        if(ability == null) {
            LogError($"Failed to toggle ability by type '{typeof(TAbility).Name}': ability
not found!");
            return false;
        }
        return ToggleAbility(ability);
    }
    protected virtual bool ToggleAbility(AbilityBase<TEntity> ability) {
        if(ability.IsEnabled) {
            DisableAbility(ability);
        } else {
            EnableAbility(ability);
        }
        return ability.IsEnabled;
    }
    protected virtual TAbility EnableAbility<TAbility>()
        where TAbility : AbilityBase<TEntity> {
        TAbility? ability = GetAbility<TAbility>();
        if(ability == null) {
```

```
            LogError($"Failed to enable ability by type '{typeof(TAbility).Name}': ability
not found!");
            return null!;
        }
        _EnableAbilityInternal(ability);
        return ability;
    }
    protected virtual void EnableAbility(AbilityBase<TEntity> ability) =>
        _EnableAbilityInternal(ability);
    protected virtual void _EnableAbilityInternal(AbilityBase<TEntity> ability) {
        if(ability.IsEnabled) {
            LogError($"Failed to enable ability '{ability.Name}': ability is already
enabled!");
            return;
        }
        ((IActivateableComponent) ability)._OnActivating();
        ability._ActivateInternal();
        ((IActivateableComponent) ability)._OnActivated();
    }
    protected virtual TAbility DisableAbility<TAbility>()
        where TAbility : AbilityBase<TEntity> {
        TAbility? ability = GetAbility<TAbility>();
        if(ability == null) {
            LogError($"Failed to disable ability by type '{typeof(TAbility).Name}': ability
not found!");
            return null!;
        }
        _DisableAbilityInternal(ability);
        return ability;
    }
    protected virtual void DisableAbility(AbilityBase<TEntity> ability) =>
        _DisableAbilityInternal(ability);
    protected virtual void _DisableAbilityInternal(AbilityBase<TEntity> ability) {
        if(ability.IsDisabled) {
            LogError($"Failed to disable ability '{ability.Name}': ability is already
disabled!");
            return;
        }
        ((IDeactivateableComponent) ability)._OnDeactivating();
        ability._DeactivateInternal();
        ((IDeactivateableComponent) ability)._OnDeactivated();
    }
    protected virtual void _ActivateInternal() {
```

```
        Node? parent = GetParent();
        if(parent == Container) {
            LogError($"Failed to deactivate component '{Name}': component is already inside
it's container!");
            return;
        }
        parent?.RemoveChild(this);
        Container.AddChild(this);
        TimeElapsedSinceActive = 0;
    }
    protected virtual void _DeactivateInternal() {
        Node? parent = GetParent();
        if(parent != Container) {
            LogError($"Failed to deactivate component '{Name}': component is already removed
from it's container!");
            return;
        }
        parent.RemoveChild(this);
        TimeElapsedSinceActive = 0;
    }
    protected virtual void _InvokeHooks(HookEvent hookEvent, HookType hookType) {
        foreach(Action hook in _MixinHooks.Where(mixinHook => mixinHook.HookEvent ==
hookEvent && mixinHook.HookType == hookType).Select(mixinHook => mixinHook.Hook)) {
            hook();
        }
    }
    #region Logging
    protected virtual string ProcessLog(string message) =>
        $"[GEC][{Engine.GetPhysicsFrames()}] {message}";
    protected virtual void Log(string message) => GD.Print(ProcessLog(message));
    protected virtual void LogError(string message) => GD.PushError(ProcessLog(message));
    #endregion
    public static class Global {
        public static Dictionary<TEntity, EntityData> _Entities { get; } = [];
        public static void _RegisterComponent(TEntity entity, ComponentBase<TEntity>
component) => _GetOrCreateEntityData(entity).Components.Add(component);
        public static void _DeregisterComponent(TEntity entity, ComponentBase<TEntity>
component) => _GetOrCreateEntityData(entity).Components.Remove(component);
        public static IObservableList<T> GetComponents<T>(TEntity entity)
            where T : ComponentBase<TEntity> =>
                _GetOrCreateEntityData(entity).Components.Connect(component => component is
T).Transform(component => (T) component).AsObservableList();
```

```
        public static List<T> ListComponents<T>(TEntity entity) where T :
ComponentBase<TEntity> => [ .._GetOrCreateEntityData(entity).Components.Items.OfType<T>() ];
        public static QueuedStateSwitchData? GetQueuedStateSwitchData(TEntity entity) =>
_GetEntityData(entity)?.QueuedStateSwitchData;
        public static bool TryGetQueuedStateSwitchData(TEntity entity, [NotNullWhen(true)]
out QueuedStateSwitchData? queuedStateSwitchData) => (queuedStateSwitchData =
GetQueuedStateSwitchData(entity)) != null;
        public static void _SetQueuedStateSwitchData(TEntity entity, QueuedStateSwitchData
queuedStateSwitchData) =>
_GetOrCreateEntityData(entity)._SetQueuedStateSwitchData(queuedStateSwitchData);
        public static bool IsStateQueuedFor(TEntity entity) =>
            GetQueuedStateSwitchData(entity) != null;
        public static void _ClearQueuedStateSwitchData(TEntity entity) =>
            _GetEntityData(entity)?._SetQueuedStateSwitchData(null);
        public static void _ClearQueuedStateSwitchDatas() {
            foreach(TEntity entity in _Entities.Keys) {
                _ClearQueuedStateSwitchData(entity);
            }
        }
        public static void _SetCurrentStateSwitchData(TEntity entity,
CurrentStateSwitchData? currentStateSwitchData) =>
_GetOrCreateEntityData(entity)._SetCurrentStateSwitchData(currentStateSwitchData);
        public static CurrentStateSwitchData? GetCurrentStateSwitchData(TEntity entity) =>
_GetEntityData(entity)?.CurrentStateSwitchData;
        public static void _SetLastStateSwitchData(TEntity entity, CurrentStateSwitchData
currentStateSwitchData) =>
_GetOrCreateEntityData(entity)._SetLastStateSwitchData(currentStateSwitchData);
        public static CurrentStateSwitchData? GetLastStateSwitchData(TEntity entity) =>
_GetEntityData(entity)?.LastStateSwitchData;
        public static EntityData? _GetEntityData(TEntity entity) =>
            _Entities.GetValueOrDefault(entity);
        public static bool _TryGetEntityData(TEntity entity, [NotNullWhen(true)] out
EntityData? entityData) => (entityData = _GetEntityData(entity)) != null;
        public static EntityData _GetOrCreateEntityData(TEntity entity) {
            if(!_Entities.TryGetValue(entity, out EntityData? entityData)) {
                _Entities[entity] = entityData = new();
            }
            return entityData;
        }
        public class EntityData {
            public virtual SourceList<ComponentBase<TEntity>> Components { get; } = new();
            public virtual QueuedStateSwitchData? QueuedStateSwitchData { get; protected
set; }
```

```
            /// <summary>
            /// Can be accessed in <see cref="StateBase{TEntity}.Enter"/> and <see
cref="StateBase{TEntity}.Exit"/> of the relevant states.
            /// </summary>
            public virtual CurrentStateSwitchData? CurrentStateSwitchData { get; protected
set; }
            public virtual CurrentStateSwitchData? LastStateSwitchData { get; protected set;
}
            public virtual void _SetQueuedStateSwitchData(QueuedStateSwitchData?
queuedStateSwitchData) => QueuedStateSwitchData = queuedStateSwitchData;
            public virtual void _SetCurrentStateSwitchData(CurrentStateSwitchData?
currentStateSwitchData) => CurrentStateSwitchData = currentStateSwitchData;
            public virtual void _SetLastStateSwitchData(CurrentStateSwitchData?
lastStateSwitchData) => LastStateSwitchData = lastStateSwitchData;
        }
    }
    public record QueuedStateSwitchData(ComponentBase<TEntity> Initiator, StateBase<TEntity>
TargetState);
    public record CurrentStateSwitchData(ComponentBase<TEntity> Initiator,
StateBase<TEntity> CurrentState, StateBase<TEntity> TargetState);
    public enum IfAStateIsAlreadyQueued { DoNothing, Overwrite, Throw }
}
```

Код класу Player

```
using GEC; using Godot; using Metal666.GodotUtilities.Architecture; using
Metal666.GodotUtilities.Extensions; using ProjectUltraversal.Player.Model; using System;
using System.Collections.Generic; using System.Linq; using static
Metal666.GodotUtilities.Extensions.Physics;
namespace ProjectUltraversal.Player;
[Singleton] public partial class Player : CharacterBody3D, IEntity {
#nullable disable
    [Export(PropertyHint.Layers3DPhysics)]
    public virtual int SolidGroundLayerMask { get; set; }
    [Export] public virtual PlayerModel Model { get; set; }
    [Export] public virtual Area3D InteractableTrigger { get; set; }
    [Export] public virtual Camera3D Camera { get; set; }
#nullable enable
    public virtual Vector2 VelocityH {
        get => Velocity.FlattenY();
        set => Velocity = Velocity.OverwriteXZ(value);
    }
    public virtual float VelocityV {
        get => Velocity.Y;
        set => Velocity = Velocity.CopyWith(y: value);
    }
    public virtual float BodyYRotation { get; set; }
    public virtual Vector3 DirectionVector => RotateVectorWithBody(Vector3.Forward);
    public virtual Quaternion DirectionQuaternion => new(Vector3.Up, BodyYRotation);
    public virtual Transform3D BodyTransform => new(DirectionQuaternion.ToBasis(),
GlobalPosition);
    public virtual Vector2 LookAngle { get; set; }
    public virtual Vector2 Movement { get; set; }
    public virtual float MovementToVelocityScalar =>
        MovementToVelocity(Movement.Normalized()).Dot(VelocityH);
    public virtual Vector3 PreviousGlobalPosition { get; set; }
    public virtual Vector3 PreviousVelocity { get; set; }
    protected List<RayCastSnapshot> performedRayCasts = [];
    public virtual IEnumerable<RayCastSnapshot> PerformedRayCasts => performedRayCasts;
    protected List<ShapeCastSnapshot> performedShapeCasts = [];
    public virtual IEnumerable<ShapeCastSnapshot> PerformedShapeCasts =>
        performedShapeCasts;
    protected List<MotionCastSnapshot> performedMotionCasts = [];
    public virtual IEnumerable<MotionCastSnapshot> PerformedMotionCasts =>
        performedMotionCasts;
    #region Node Events
    public override void _Ready() {
```

```
        base._Ready();
        PreviousGlobalPosition = GlobalPosition;
        PreviousVelocity = Velocity;
    }
    public override void _Process(double delta) {
        base._Process(delta);
        Movement = Vector2.Zero;
    }
    public override void _PhysicsProcess(double delta) {
        base._PhysicsProcess(delta);
        Invoke.Deferred(() => {
            performedRayCasts.Clear();
            performedShapeCasts.Clear();
            performedMotionCasts.Clear();
        }, 1000);
    }
    #endregion
    public virtual Vector2 MovementToVelocity(Vector2 movement) =>
        movement.FlipY().Rotated(-LookAngle.X);
    public virtual Vector2 MovementToVelocity() =>
        MovementToVelocity(Movement.Normalized());
    public virtual Vector3 LocalOffsetToGlobalPosition(Vector3 offset) =>
        GlobalPosition + RotateVectorWithBody(offset);
    public virtual Vector3 RotateVectorWithBody(Vector3 vector) =>
        vector.Rotated(Vector3.Up, BodyYRotation);
    public virtual Quaternion RotateWithBody(Quaternion rotation) =>
        new Quaternion(Vector3.Up, BodyYRotation) * rotation;
    public virtual RayCastArrayResults RayCastArray(RayCastArrayConfiguration configuration)
{
        List<RayCastArrayResult?> results = [];
        static float CalculateOffset(int index, RayCastArrayConfiguration.AxisConfiguration
axisConfiguration) {
            float offset = index * axisConfiguration.Spacing;
            return axisConfiguration.OffsetBehaviour switch {
                RayCastArrayConfiguration.AxisOffsetBehaviour.Centered =>
                    offset - ((axisConfiguration.Count - 1) * (axisConfiguration.Spacing /
2f)),
                RayCastArrayConfiguration.AxisOffsetBehaviour.Negative => -offset,
                _ => offset
            };
        }
        for(int i = 0; i < configuration.ZAxis.Count; i++) {
            float zOffset = CalculateOffset(i, configuration.ZAxis);
```

```
            for(int j = 0; j < configuration.XAxis.Count; j++) {
                float xOffset = CalculateOffset(j, configuration.XAxis);
                Vector3 rayCastOffset = configuration.Rotation *
                        new Vector3(xOffset, 0, zOffset);
                Vector3 rayCastOrigin = configuration.Position + rayCastOffset;
                Vector3 rayCastTarget = rayCastOrigin + (configuration.Rotation *
(Vector3.Up * configuration.Length));
                RayCastResult3D? result = RayCast(rayCastOrigin, rayCastTarget, true, true,
                            $"{configuration.IdPrefix}_{i}_{j}");
                if(configuration.BreakWhen != null &&
                    configuration.BreakWhen(result)) {
                    return new(results);
                }
                results.Add(result != null ? new(new(xOffset, zOffset), result) : null);
            }
        }
        return new(results);
    }
    public class RayCastArrayConfiguration {
        public virtual required Vector3 Position { get; init; }
        public virtual required float Length { get; init; }
        public virtual Quaternion Rotation { get; init; } = Quaternion.Identity;
        public virtual AxisConfiguration XAxis { get; init; } = new();
        public virtual AxisConfiguration ZAxis { get; init; } = new();
        public virtual required string IdPrefix { get; init; } = "";
        [Obsolete]
        public virtual Func<RayCastResult3D?, bool>? BreakWhen { get; init; } = null;
        public record AxisConfiguration(int Count = 1, float Spacing = 0,
AxisOffsetBehaviour OffsetBehaviour = AxisOffsetBehaviour.Centered);
        public enum AxisOffsetBehaviour { Centered, Positive, Negative }
    }
    public virtual RayCastResult3D? RayCast(Vector3 from, Vector3 to, bool hitFromInside,
bool hitIntended, string rayCastId = "", uint? collisionMask = null) {
        collisionMask ??= (uint) SolidGroundLayerMask;
        RayCastResult3D? result =
            this.RayCast(new() { From = from, To = to, HitFromInside = hitFromInside,
CollisionMask = collisionMask.Value });
        performedRayCasts.Add(new(rayCastId, from, to, hitIntended ? (result != null) :
(result == null), hitIntended, result?.Position, result?.Normal));
        return result;
    }
    public virtual bool TryShapeCast(Transform3D transform, Shape3D shape, bool hitIntended,
string shapeCastId = "", uint? collisionMask = null) {
```

```
        collisionMask ??= (uint) SolidGroundLayerMask;
        List<ShapeCastResult3D> result = this.ShapeCast(new() { Transform = transform, Shape
= shape, CollisionMask = collisionMask.Value });
        bool hit = result.Count > 0;
        performedShapeCasts.Add(new(shapeCastId, transform, shape, hitIntended ? hit : !hit,
hitIntended));
        return hit;
    }
    public virtual bool TryCastMotion(Transform3D transform, Vector3 motion, Shape3D shape,
bool hitIntended, string shapeCastId = "", bool intersectFirst = true, uint? collisionMask =
null) {
        collisionMask ??= (uint) SolidGroundLayerMask;
        bool hit = ( intersectFirst && this.TryShapeCast(new() { Transform = transform,
Shape = shape, CollisionMask = collisionMask.Value }, out _, 1) ) || this.CastMotion(new() {
Transform = transform, Shape = shape, Motion = motion, CollisionMask = collisionMask.Value
}, out _, out _);
        performedMotionCasts.Add(new(shapeCastId, transform, motion, shape, hitIntended ?
hit : !hit, hitIntended));
        return hit;
    }
    public static implicit operator PlayerModel(Player player) => player.Model;
    public class RayCastArrayResults(List<RayCastArrayResult?> results) {
        protected virtual List<RayCastArrayResult?> Results { get; } = results;
        public virtual List<RayCastArrayResult> NonNull { get; } = [
..results.OfType<RayCastArrayResult>() ];
        public virtual bool AllMiss => NonNull.Count == 0;
        public virtual bool AllHit => NonNull.Count == Results.Count;
        public virtual bool AnyMiss => NonNull.Count != Results.Count;
        public virtual bool AnyHit => NonNull.Count != 0;
        public virtual bool AnyInside => NonNull.Any(result => result.Result.IsInside);
        public virtual bool AnyMissOrInside => AnyMiss || AnyInside;
        public virtual bool AnyTooSteep(float maxAngle, Vector3? up = null) =>
            NonNull.Any(result => result.Result.IsTooSteep(maxAngle, up));
        public virtual bool AllTooSteep(float maxAngle, Vector3? up = null) =>
            NonNull.All(result => result.Result.IsTooSteep(maxAngle, up));
        public virtual IEnumerable<RayCastArrayResult> OrderByXOffset =>
            NonNull.OrderBy(result => result.Offset.X);
        public virtual RayCastArrayResult MinByXOffset => OrderByXOffset.First();
        public virtual IEnumerable<RayCastArrayResult> OrderByYPosition =>
            NonNull.OrderBy(result => result.Result.Position.Y);
        public virtual RayCastArrayResult MaxByYPosition => OrderByYPosition.Last();
        public virtual RayCastArrayResult ClosestTo(Vector3 position) =>
            NonNull.MinBy(result => result.Result.Position.DistanceSquaredTo(position))!;
```

```
    }
    public record RayCastArrayResult(Vector2 Offset, RayCastResult3D Result);
    public record RayCastSnapshot(string Id, Vector3 From, Vector3 To, bool Success, bool
HitIntended, Vector3? Position, Vector3? Normal);
    public record ShapeCastSnapshot(string Id, Transform3D Transform, Shape3D Shape, bool
Success, bool HitIntended);
    public record MotionCastSnapshot(string Id, Transform3D Transform, Vector3 Motion,
Shape3D Shape,  bool Success, bool HitIntended);
}
```