

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалавр

(ступінь вищої освіти)

на тему ПРОГРАМНА РЕАЛІЗАЦІЯ КЕРУВАННЯ АКТИВНИМ
РЕГДОЛОМ ДЛЯ САМОБАЛАНСУВАННЯ І ПЕРЕМІЩЕННЯ
У ІГРОВІЙ МЕХАНІЦІ

SOFTWARE IMPLEMENTATION OF ACTIVE RAGDOLL
CONTROL FOR SELF-BALANCING AND MOVEMENT
IN GAMEPLAY MECHANICS

Виконав(ла): студент(ка) 4 курсу, групи КНТ-112

Спеціальності 121 Інженерія програмного

(код і найменування спеціальності)

забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

ГЛЄБОВ М.М.

(ПРИЗВИЩЕ та ініціали)

Керівник ЛЕОЩЕНКО С.Д.

(ПРИЗВИЩЕ та ініціали)

Рецензент ПОЛЯКОВ М.О.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Факультет КНТ
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 Спеціальність 121 Інженерія програмного забезпечення
 (код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
 (назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
 “ ” 2026 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ГЛЄБОВА Максима Максимовича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Програмна реалізація керування активним регдолом для самобалансування і переміщення у ігровій механіці. Implementation of Active Ragdoll Control for Self-balancing and Movement in Gameplay Mechanics

керівник проєкту (роботи) доктор філософії, доцент, ЛЕОЩЕНКО Сергій Дмитрович

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

затверджені наказом закладу вищої освіти від “ 07 ” квітня 2026 року № 139

2. Строк подання студентом проєкту (роботи) 10 червня 2026 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз проблеми та постановка завдань дослідження. 2. Матеріали і методи. 3. Опис програми. 4. Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)

Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ЛЕОЩЕНКО С.Д., доцент		
Нормоконтроль	КАЛІНІНА М.В., асистент		

7. Дата видачі завдання « 20 » квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій	2 тиждень	Розділ 2
	розробки.		
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження	5 тиждень	Розділ 4
	програмного забезпечення.		
7	Оформлення пояснювальної записки та документів	6 тиждень	Додатки
	до неї.		
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)

(підпис) Максим ГЛЄБОВ
(Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Сергій ЛЕОЩЕНКО
(Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 118 с., 3 табл., 45 рис., 3 дод., 48 джерел.

АКТИВНИЙ РЕГДОЛ, ФІЗИЧНА СИМУЛЯЦІЯ, PID-РЕГУЛЯТОР, САМОБАЛАНСУВАННЯ, ПРОЦЕДУРНА АНІМАЦІЯ, ІГРОВИЙ РУШІЙ, UNITY.

Об'єкт дослідження – процес розроблення програмного забезпечення для керування фізично симульованими персонажами у відеоіграх.

Предмет дослідження – методи та програмні засоби керування активним регдолом на основі PID-регуляторів, пружинних сил і м'язових моментів для забезпечення самобалансування та переміщення персонажа.

Мета роботи – підвищення реалістичності поведінки персонажа та його взаємодії з оточенням у відеоіграх шляхом розроблення програмних засобів керування активним регдолом на основі фізичної симуляції.

Матеріали, методи та технічні засоби: об'єктно-орієнтоване програмування, фізична симуляція твердих тіл, PID-регулятори, зворотна кінематика (ІК), алгоритми навігації на основі NavMesh, мова програмування C#, ігровий рушій Unity, середовище розробки Visual Studio Code, персональний комп'ютер з процесором AMD Ryzen 7 7800X3D під управлінням операційної системи Microsoft Windows 11.

Результати. Розроблено програмну систему керування активним регдолом (NPAR), що реалізує фізично коректне самобалансування та переміщення персонажа в реальному часі засобами ігрового рушія Unity.

Висновки. Розроблена система демонструє практичну придатність підходу до керування активним регдолом на основі комбінування PID-регуляторів, м'язових моментів і процедурної анімації кінцівок для вирішення задач самобалансування та переміщення у фізичній симуляції.

Галузь використання – розробка відеоігор, симуляція фізично керованих персонажів, процедурна анімація.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 118 pages, 3 tables, 45 figures, 3 appendixes, 48 sources.

ACTIVE RAGDOLL, PHYSICAL SIMULATION, PID CONTROLLER, SELF-BALANCING, PROCEDURAL ANIMATION, GAME ENGINE, UNITY.

Object of study is the process of software development for controlling physically simulated characters in video games.

Subject of study is methods and software tools for active ragdoll control based on PID controllers, spring forces, and muscle torques to ensure character self-balancing and movement.

The purpose of the work is to increase the realism of character behavior and its interaction with the environment in video games by developing software tools for active ragdoll control based on physical simulation.

Materials, methods, and tools: object-oriented programming, rigid body physical simulation, PID controllers, inverse kinematics (IK), NavMesh-based navigation algorithms, C# programming language, Unity game engine, Visual Studio Code development environment, personal computer with an AMD Ryzen 7 7800X3D processor running Microsoft Windows operating system.

Results. A software system for active ragdoll control (NPAR) has been developed, implementing physically correct self-balancing and character movement in real time using the Unity game engine.

Conclusions. The developed system demonstrates the practical viability of the active ragdoll control approach based on combining PID controllers, muscle torques, and procedural limb animation for solving self-balancing and movement tasks in physical simulation.

The field of use is video game development, simulation of physically controlled characters, and procedural animation.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	9
Вступ.....	10
1 Аналіз проблеми та постановка завдань дослідження	12
1.1 Підходи до анімації персонажів у відеоіграх	12
1.2 Концепція активного регдолу	14
1.3 Огляд існуючих реалізацій активного регдолу	15
1.4 Постановка завдань дослідження	20
1.5 Висновки до розділу 1	21
2 Матеріали і методи.....	23
2.1 Unity як середовище розробки.....	23
2.2 Фізична симуляція твердих тіл.....	25
2.3 PID-регулятори.....	27
2.4 М'язові моменти та пружинні сили	31
2.5 Зворотна кінематика	33
2.6 Методи керування постановкою ніг.....	35
2.7 Навігація на основі NavMesh	37
2.8 Висновки до розділу 2	38
3 Опис програми.....	40
3.1 Загальна архітектура системи	40
3.2 Система пресетів і налаштування поведінки	45
3.3 Машина станів та прийняття рішень	48
3.4 Контролери фізичної симуляції.....	52
3.5 Контролер навігації.....	68
3.6 Висновки до розділу 3	72
4 Експлуатація, тестування та експериментальне дослідження програми	74
4.1 Середовище тестування.....	74
4.2 Результати функціонального тестування	75

4.3 Узагальнені результати.....	83
Висновки	86
Перелік джерел посилення	88
Додаток А Технічне завдання	94
Додаток Б Текст програми	97
Додаток В Слайди презентації.....	108

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

CoM	– Center of Mass;
FK	– Forward Kinematics;
IK	– Inverse Kinematics;
NavMesh	– Navigation Mesh;
NPAR	– Non-Playable Active Ragdoll;
PD	– Proportional-Derivative;
PID	– Proportional-Integral-Derivative;
UE	– Unreal Engine;
ПЗ	– програмне забезпечення;
ПК	– персональний комп'ютер.

ВСТУП

Сучасна ігрова індустрія висуває дедалі вищі вимоги до реалістичності поведінки персонажів. Гравець відчуває занурення у гру тоді, коли ігровий світ справляє враження живого та послідовного – коли персонажі переконливо взаємодіють з оточенням, реагують на фізичні впливи та поведуться природно. Порушення цього враження – будь то неприродна анімація, проходження персонажа крізь поверхні або відсутність реакції на зіткнення – миттєво руйнує імерсивність і погіршує користувацький досвід.

Традиційні підходи до анімації персонажів, зокрема ключова анімація та motion capture, забезпечують високу якість і реалістичність руху, проте є статичними за своєю природою: анімація відтворюється незалежно від стану фізичного оточення. Процедурна анімація та фізично керовані персонажі, навпаки, динамічно взаємодіють із середовищем, адаптуючись до нерівної поверхні, зіткнень та зовнішніх сил у реальному часі. Разом з тим досягнення при цьому рівня природності руху, порівнянного з motion capture, залишається актуальною науково-технічною задачею. Одним із перспективних підходів до її вирішення є концепція активного регдолу – фізично симульованого персонажа, керованого м'язовими силами та регуляторами, що утримують тіло у рівновазі та забезпечують цілеспрямоване переміщення.

Метою роботи є дослідження та програмна реалізація методів керування фізично симульованими персонажами на основі PID-регуляторів, пружинних сил і м'язових моментів для досягнення стійкого балансу та природного переміщення.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- проаналізувати існуючі підходи до анімації та керування персонажами у відеоіграх;

- розробити архітектуру програмної системи керування активним регдолом;
- реалізувати алгоритми самобалансування та переміщення фізично симульованого персонажа;
- дослідити поведінку розробленої системи в ігровому середовищі.

Об'єктом дослідження є методи та алгоритми керування фізично симульованими персонажами в ігровому середовищі.

Предметом дослідження є програмна система керування активним регдолом на основі PID-регуляторів, пружинних сил і м'язових моментів для забезпечення самобалансування та переміщення персонажа.

Практична цінність роботи полягає у розробці системи, що може бути застосована у відеоіграх для підвищення імерсивності та реалістичності поведінки персонажів, а також у задачах процедурної анімації та симуляції фізично керованих об'єктів.

1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАВДАНЬ ДОСЛІДЖЕННЯ

1.1 Підходи до анімації персонажів у відеоіграх

Анімація персонажів є одним із ключових чинників, що визначають імерсивність відеогри. Спосіб, у який персонаж рухається, реагує на оточення та взаємодіє з фізичним світом, безпосередньо впливає на сприйняття гравцем ігрового середовища як достовірного. Впродовж розвитку ігрової індустрії сформувалося кілька принципово різних підходів до анімації персонажів, кожен з яких має характерні переваги та обмеження.

1.1.1 Ключова анімація

Ключова анімація є найпоширенішим підходом у сучасних відеоіграх [1]. Аніматор вручну визначає положення кісток скелета персонажа у ключових кадрах, а рушій автоматично інтерполює проміжні стани. Такий підхід дає художнику повний контроль над виразністю та стилістикою руху, дозволяє досягати як реалістичних, так і стилізованих результатів, і є відносно економічним з точки зору обчислювальних ресурсів.

Проте ключова анімація є статичною: готові анімації відтворюються незалежно від поточного стану фізичного оточення. Персонаж може ковзати по поверхні, не торкаючись її ногами, або не реагувати на зіткнення у спосіб, що відповідав би фізичним законам. Усунення таких артефактів вимагає або створення великої кількості варіативних анімацій, або поєднання ключової анімації з іншими методами.

1.1.2 Захоплення руху (Motion Capture)

Технологія захоплення руху передбачає запис реальних рухів людини або тварини за допомогою спеціалізованого обладнання з подальшим перенесенням отриманих даних на скелет ігрового персонажа. Результатом є

анімація з високим ступенем природності та реалістичності, яку складно відтворити вручну [2]-[3].

Разом з тим motion capture має ті самі фундаментальні обмеження, що й ключова анімація: записані рухи є фіксованими і не адаптуються до динамічного оточення. Крім того, виробництво motion capture є значно дорожчим і технічно складнішим процесом, що робить його малодоступним для невеликих команд розробників.

1.1.3 Процедурна анімація

Процедурна анімація генерується алгоритмічно у реальному часі на основі поточного стану ігрового світу [4]. На відміну від ключової анімації та motion capture, вона не є заздалегідь записаною: положення кінцівок та тіла персонажа обчислюється кожного кадру з урахуванням геометрії оточення, фізичних впливів та інших чинників. Це дозволяє персонажу природно адаптуватися до нерівної поверхні, динамічно розміщувати кінцівки та реагувати на непередбачені ситуації.

Основним недоліком процедурної анімації є складність досягнення тієї виразності та природності руху, яку забезпечує ручна робота аніматора або motion capture. Алгоритмічно згенерований рух може виглядати механічно або неприродно, якщо не доповнений додатковими засобами корекції.

1.1.4 Фізична симуляція та пасивний регдол

Фізична симуляція передбачає керування персонажем безпосередньо через закони механіки: кожна частина тіла є твердим тілом з масою та інерцією, а їхня взаємодія визначається фізичним рушієм. Найпростішою формою такого підходу є пасивний регдол – модель, у якій тіло персонажа повністю підпорядковане зовнішнім силам без жодного активного керування.

Пасивний регдол широко використовується для симуляції смерті або непритомності персонажа, коли м'язовий контроль повністю втрачається [5]-[6].

Попри фізичну коректність поведінки, пасивний регдол не здатний самостійно підтримувати рівновагу або виконувати цілеспрямовані рухи, що унеможлиблює його використання як основного методу керування живим персонажем. Подолання цього обмеження є мотивацією для розвитку концепції активного регдолу, розглянутої у наступному підрозділі.

1.2 Концепція активного регдолу

Активний регдол є розвитком ідеї фізичної симуляції персонажа, у якому тіло не лише підпорядковується зовнішнім силам, а й активно чинить опір їм за допомогою внутрішніх керуючих сил. Подібно до того, як м'язи живої істоти утримують тіло у рівновазі та забезпечують цілеспрямований рух, активний регдол використовує алгоритмічно генеровані сили та моменти, що діють на суглоби фізичної моделі персонажа.

Ключовою архітектурною особливістю активного регдолу є використання двох паралельних скелетних моделей. Перша модель – анімаційна – відтворює цільову позу персонажа та слугує орієнтиром для керування. Друга модель – фізична – є набором твердих тіл, з'єднаних суглобами, і безпосередньо бере участь у симуляції фізичного рушія. Керуюча система зчитує цільові положення кісток анімаційної моделі кожне оновлення фізичної симуляції та застосовує до суглобів фізичної моделі сили, що прагнуть відтворити цю позу. Таким чином, анімаційна модель визначає бажану поведінку, а фізична – реалізує її в умовах реального фізичного оточення [7]-[8].

Важливо, що анімаційна модель сама по собі може використовувати будь-який з описаних у підрозділі 1.1 підходів або їхню комбінацію. У даній роботі застосовується гібридний підхід: анімаційна модель поєднує ключові

анімації для базових станів персонажа з процедурно генерованими рухами кінцівок та тулуба. Таке поєднання дозволяє зберегти виразність ручної анімації коли це необхідно, і водночас забезпечити динамічну адаптацію до оточення засобами процедурної генерації.

Порівняно з пасивним регдолом, активний регдол здатний підтримувати рівновагу, протидіяти зовнішнім збуренням та виконувати цілеспрямоване переміщення, залишаючись при цьому повністю фізично симульованим. Це забезпечує природну взаємодію персонажа з оточенням, яку неможливо досягти засобами статичної анімації.

При цьому допустиме використання штучних зовнішніх сил для підтримування рівноваги активного регдолу. Але чим краще буде реалізоване керування цільовою позою анімаційної моделі, тим менше буде потреба в використанні допоміжних зовнішніх сил.

1.3 Огляд існуючих реалізацій активного регдолу

Концепція активного регдолу реалізована у ряді комерційних продуктів та ігрових рушіїв з різним ступенем повноти та складності. Розглянемо найбільш показові приклади, що охоплюють широкий спектр підходів – від спеціалізованих промислових рішень до відносно простих реалізацій на загальнодоступних рушіях.

1.3.1 Euphoria (Natural Motion)

Euphoria є спеціалізованою middleware-бібліотекою компанії NaturalMotion, призначеною для інтеграції у сторонні ігрові рушії [9]-[10]. На відміну від самостійного рушія, Euphoria не керує рендерингом чи іншими аспектами гри – її єдиною задачею є симуляція поведінки фізично керованих персонажів, що дозволило досягти в цій галузі виняткового рівня якості.

В основі Euphoria лежить нейром'язова симуляція: персонаж моделюється як система м'язів і суглобів, що активно протидіють зовнішнім силам і прагнуть підтримувати рівновагу [10]-[11]. Система у реальному часі обчислює відповідні керуючі зусилля для кожного суглоба тіла залежно від поточної ситуації – падіння, удару, втрати опори тощо. Завдяки цьому персонажі переконливо спотикаються, намагаються утриматися на ногах під час зіткнень, інстинктивно виставляють руки при падінні та реагують на кожен удар індивідуально, а не відтворюють заздалегідь записану анімацію.

Euphoria застосовувалася у таких іграх, як Grand Theft Auto IV, Grand Theft Auto V та Red Dead Redemption, де фізична поведінка персонажів стала однією з найбільш помітних рис цих проєктів [9]-[11]. Рівень достовірності та варіативності поведінки, досягнутий за допомогою Euphoria, досі залишається еталоном у галузі активної фізичної симуляції персонажів і практично недосяжний для відтворення у рамках окремої незалежної розробки. Приклад роботи Euphoria представлено на рис. 1.1 [12].

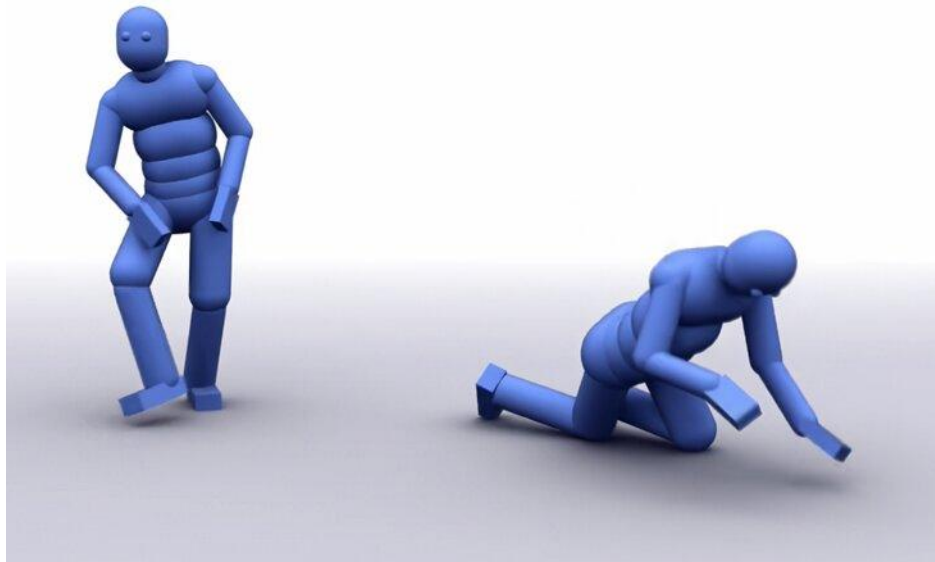


Рисунок 1.1 – Приклад роботи Euphoria [12]

1.3.2 Unreal Engine Physical Animation

Unreal Engine надає вбудований інструментарій Physical Animation, що дозволяє застосовувати елементи фізичної симуляції до скелетних персонажів безпосередньо у середовищі рушія. Підхід ґрунтується на змішуванні анімаційного та фізично симульованого станів кісток скелета: для кожної кістки або групи кісток задається сила, з якою фізична симуляція прагне відтворити цільову позу поточної анімації [13]-[14]. Чим вища ця сила, тим точніше фізична модель слідує анімації; при нижчих значеннях кістки більше підпорядковуються зовнішнім фізичним впливам.

Така архітектура концептуально близька до активного регдолу і дозволяє, наприклад, зберігати анімований рух верхньої частини тіла, водночас дозволяючи ногам або рукам фізично реагувати на зіткнення. Проте Physical Animation є передусім інструментом часткової фізичної реакції, а не повноцінного керування тілом: система не реалізує активного балансування, не управляє центром мас і не здатна самостійно утримувати персонажа у вертикальному положенні під дією зовнішніх збурень. Повноцінна реалізація активного регдолу засобами Unreal Engine вимагає значних додаткових розробок поверх наявного інструментарію. Роботу інструментарію можна побачити на рис. 1.2 [15].

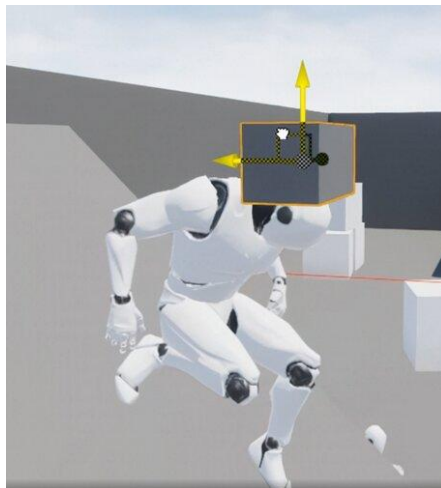


Рисунок 1.2 – Приклад роботи Unreal Engine Physical Animation [15]

1.3.3 Human: Fall Flat

Гра «Human: Fall Flat» є одним із найбільш показових прикладів використання концепції активного регдолу як фундаментальної основи ігрового процесу [16]-[17]. На відміну від систем на кшталт Euphoria, які орієнтовані на досягнення максимальної реалістичності поведінки, або гібридних підходів рушія Unreal Engine, розробники «Human: Fall Flat» застосували повністю фізично керовану модель персонажа, відмовившись від традиційних ключових анімацій для базових рухів.

Керування персонажем у цій грі здійснюється шляхом застосування фізичних сил та крутних моментів безпосередньо до з'єднань твердих тіл, з яких складається модель. Переміщення та підтримання рівноваги реалізуються за допомогою внутрішніх математичних регуляторів, які генерують керуючі впливи для утримання центру мас персонажа над площею опори. Алгоритми переміщення процедурно розраховують траєкторії для ніг, створюючи кроки шляхом докладання імпульсів до відповідних кінцівок у реальному часі.

Особливістю реалізації є те, що гравець має прямий контроль над руками персонажа, що дозволяє виконувати процедурне захоплення об'єктів та взаємодіяти з оточенням [18]. Оскільки руки також є частиною фізичної симуляції, вага піднятих предметів, їх інерція та точки зіткнення безпосередньо впливають на баланс усього тіла, змушуючи систему самобалансування динамічно адаптуватися до нових умов.

Хоча такий підхід цілеспрямовано не забезпечує біомеханічної достовірності рухів, пропонуючи натомість дещо комічну та незграбну фізику, він створює надзвичайно високий рівень емерджентності. Персонаж здатний динамічно адаптуватися до рельєфу будь-якої складності, фізично коректно реагувати на перешкоди та формувати унікальні рухи під час кожного проходження. Ця реалізація яскраво демонструє, як активний регдол може слугувати не лише допоміжним інструментом анімації, але й ключовою

механікою, що повністю визначає взаємодію гравця зі світом та дизайн головоломок. Геймплей гри можна побачити на рис. 1.3 [19].



Рисунок 1.3 – Приклад роботи активного регдолу у Human: Fall Flat [19]

1.3.4 Порівняння існуючих реалізацій активного регдолу

Існуючі реалізації активного регдолу охоплюють широкий спектр підходів – від промислових рішень із нейром’язовою симуляцією до спрощених фізично керованих систем на загальнодоступних рушіях. Спільною рисою повноцінних реалізацій є активне балансування та висока адаптивність до оточення, тоді як доступність для незалежних розробників залишається обмеженою. Детальне порівняння існуючих реалізацій активного регдолу представлено в таблиці 1.1.

Таблиця 1.1 – Порівняння існуючих реалізацій активного регдолу

Критерій	Euphoria	UE Physical Animation	Human: Fall Flat
1	2	3	4
Спосіб керування суглобами	Нейром’язова симуляція	Звичайні пружини у суглобах	PD-подібні регулятори

Продовження таблиці 1.1

1	2	3	4
Реакція на зовнішні сили	Повна, з урахуванням усього тіла	Часткова, налаштовується по кістках	Повна, для всього тіла
Активне балансування	Так, повноцінне	Ні	Так, спрощене
Адаптивність до оточення	Висока	Часткова	Висока
Реалістичність руху	Висока	Залежить від анімації	Навмисно спрощена
Доступність для розробників	Комерційна ліцензія	Вбудовано в Unreal Engine	Реалізація на Unity

Порівняння підтверджує доцільність розробки власної системи керування активним регдолом на основі відкритого інструментарію, здатної забезпечити повноцінне самобалансування та переміщення персонажа

1.4 Постановка завдань дослідження

На основі проведеного аналізу підходів до анімації персонажів та існуючих реалізацій активного регдолу можна сформулювати конкретні задачі, які необхідно вирішити в рамках даної роботи.

Аналіз існуючих рішень показав, що повноцінна система керування активним регдолом повинна забезпечувати: підтримання вертикального положення тіла в умовах зовнішніх збурень, цілеспрямоване переміщення у просторі, природну поведінку кінцівок під час руху, а також здатність відновлюватися після втрати рівноваги. Досягнення цих властивостей вимагає вирішення ряду взаємопов'язаних задач.

Відповідно до мети роботи визначено такі завдання дослідження:

- проаналізувати існуючі підходи до анімації та керування персонажами у відеоіграх, визначити їхні переваги та обмеження;

- дослідити методи активного керування фізично симульованими тілами, зокрема PID-регулятори, пружинні сили та м'язові моменти, як основу для реалізації системи;
- розробити архітектуру програмної системи керування активним регдолем, що забезпечує модульність та можливість гнучкого налаштування поведінки;
- реалізувати алгоритми самобалансування тіла персонажа на основі відстеження анімаційної моделі та корекції відхилень фізичними силами;
- реалізувати процедурні алгоритми керування кінцівками персонажа під час переміщення, зокрема постановку ніг та рух рук;
- реалізувати систему навігації, що забезпечує планування маршруту та фізично коректне керування швидкістю переміщення;
- дослідити поведінку розробленої системи в ігровому середовищі та оцінити відповідність результатів поставленим вимогам.

1.5 Висновки до розділу 1

У першому розділі було проведено аналіз існуючих підходів до анімації персонажів у відеоіграх та розглянуто концепцію активного регдолу як перспективного методу забезпечення імерсивної та фізично коректної поведінки персонажів.

Встановлено, що традиційні методи – ключова анімація та захоплення руху – забезпечують високу якість і виразність руху, проте є статичними за своєю природою і не здатні динамічно адаптуватися до фізичного оточення. Процедурна анімація усуває цей недолік, проте поступається традиційним методам за природністю та виразністю руху. Пасивний регдол забезпечує фізично коректну реакцію на зовнішні сили, але не здатний до активного керування тілом.

Активний регдол поєднує переваги фізичної симуляції з можливістю цілеспрямованого керування персонажем завдяки архітектурі подвійної скелетної моделі, у якій анімаційна модель слугує динамічною ціллю для фізично симульованого тіла. Огляд існуючих реалізацій показав, що повноцінні системи активного регдолу або є недоступними комерційними рішеннями, або обмежуються частковою фізичною симуляцією.

На підставі проведеного аналізу сформульовано завдання дослідження, що охоплюють розробку архітектури системи, реалізацію алгоритмів самобалансування, керування кінцівками та навігації, а також дослідження поведінки системи в тестовому середовищі.

2 МАТЕРІАЛИ І МЕТОДИ

2.1 Unity як середовище розробки

Unity – це кросплатформний ігровий рушій, розроблений компанією Unity Technologies, що набув широкого застосування як у комерційній розробці відеоігор, так і в академічних та дослідницьких проєктах [20]. Рушій підтримує розробку для широкого спектру платформ, зокрема Windows, macOS, Linux, Android, iOS та ігрових консолей, що робить його одним із найбільш універсальних інструментів у галузі [21].

Завдяки розвиненій екосистемі, великій спільноті розробників та доступній документації Unity є поширеним вибором для проєктів різного масштабу – від невеликих інді-розробок до великих комерційних продуктів. Саме ця доступність у поєднанні з потужним набором вбудованих інструментів визначила вибір Unity як середовища розробки для даної роботи.

2.1.1 Компонента архітектура та життєвий цикл об'єктів

Архітектура Unity ґрунтується на компонентній моделі: кожен об'єкт сцени є екземпляром класу `GameObject`, функціональність якого визначається набором прикріплених до нього компонентів. Власна логіка реалізується через компоненти-скрипти, що успадковують базовий клас `MonoBehaviour`. Така архітектура сприяє модульності коду та чіткому розподілу відповідальності між підсистемами.

Особливо важливим для реалізації системи керування активним реголом є чітко визначений життєвий цикл оновлення об'єктів [22]. Unity розрізняє три основні методи оновлення:

- `FixedUpdate` – викликається з фіксованою частотою, незалежною від частоти кадрів, і є основним місцем для всіх обчислень, пов'язаних із фізичною симуляцією;

- `Update` – викликається щокадру і використовується для логіки, що залежить від частоти кадрів;
- `LateUpdate` – викликається щокадру після завершення всіх викликів `Update`, що дозволяє виконувати операції, які мають відбуватися після оновлення стану всіх інших компонентів.

У розробленій системі `FixedUpdate` використовується для застосування фізичних сил та оновлення стану регдолу, тоді як `LateUpdate` – для синхронізації положення анімаційної моделі з фізичною.

2.1.2 Фізичний рушій PhysX

Для симуляції фізики Unity використовує бібліотеку NVIDIA PhysX – один із найпоширеніших фізичних рушіїв у ігровій індустрії [23], [24]. PhysX забезпечує симуляцію твердих тіл, виявлення та обробку колізій, а також роботу з суглобами та обмеженнями. Детальніше можливості PhysX у контексті реалізації системи розглядаються у підрозділі 2.2.

2.1.3 Система анімації

Unity надає розвинену систему анімації на основі компонента `Animator` та анімаційних кліпів `AnimationClip` [25], [26]. `Animator` керує відтворенням анімацій через граф станів, у якому переходи між анімаціями визначаються програмно заданими параметрами [27]. Система підтримує змішування анімацій, інверсну кінематику та процедурні модифікації пози, що робить її придатною як основу для анімаційної моделі активного регдолу.

2.1.4 Мова програмування C#

Скриптування в Unity здійснюється мовою C# – строго типізованою об'єктно-орієнтованою мовою платформи .NET [28]. C# надає розробнику

повний набір засобів об'єктно-орієнтованого програмування, зокрема інкапсуляцію, спадкування та поліморфізм, а також підтримує сучасні можливості мови, такі як узагальнені типи, лямбда-вирази та події. Ці засоби були активно використані при проєктуванні модульної архітектури системи керування активним регдолом.

2.2 Фізична симуляція твердих тіл

2.2.1 Тверді тіла

Твердим тілом у контексті фізичної симуляції називається об'єкт, форма якого вважається незмінною під дією зовнішніх сил. У Unity тверде тіло представлене компонентом `Rigidbody`, що надає об'єкту фізичні властивості: масу, момент інерції, лінійне та кутове затухання [29]. `PhysX` щокроку симуляції інтегрує рівняння руху для кожного твердого тіла, обчислюючи нові положення та швидкості на основі прикладених сил і моментів.

У системі керування активним регдолом кожна кістка фізичного скелета персонажа є окремим твердим тілом із власною масою та параметрами затухання. Сукупність цих тіл, з'єднаних суглобами, утворює фізичну модель персонажа, поведінка якої визначається як зовнішніми силами оточення, так і внутрішніми керуючими силами системи.

2.2.2 Виявлення та обробка колізій

`PhysX` виявляє зіткнення між колайдерами – спрощеними геометричними формами, що апроксимують межі фізичних об'єктів. При виявленні перетину двох колайдерів рушій обчислює контактні точки та нормалі, після чого застосовує імпульсні сили, що розштовхують тіла та симулюють пружну або непружну взаємодію залежно від налаштованих

фізичних матеріалів. Коректна обробка колізій є необхідною умовою природної взаємодії персонажа з поверхнями оточення.

2.2.3 Режими виявлення колізій

PhysX підтримує кілька режимів виявлення колізій, що відрізняються точністю та обчислювальною вартістю [30].

Режим Discrete є найпростішим і найшвидшим: зіткнення перевіряються лише у дискретних точках часу – на початку та наприкінці кожного кроку симуляції. Це означає, що об'єкти з високою швидкістю теоретично можуть проникати крізь тонкі перешкоди між кроками, проте для об'єктів із помірною швидкістю цей режим забезпечує стабільну та передбачувану поведінку.

Режим Continuous виконує безперервну перевірку траєкторії об'єкта між кроками симуляції, що запобігає проникненню крізь перешкоди при високих швидкостях. Проте цей режим є значно дорожчим обчислювально і, як показало тестування, менш стабільним при роботі з суглобами: реакції на колізії стають менш передбачуваними, що є критичним недоліком для системи, що спирається на точність фізичної симуляції з'єднань скелета.

Режим Continuous Dynamic є розширенням попереднього і застосовує безперервну перевірку як для самого об'єкта, так і для його взаємодії з іншими об'єктами у відповідних режимах.

У даній роботі для всіх твердих тіл фізичного скелета використовується режим Discrete. Оскільки жоден із компонентів системи не передбачає переміщення з високою швидкістю, ризик проникнення крізь перешкоди є незначним. Натомість режим Discrete забезпечує стабільнішу та передбачуванішу поведінку суглобів, вищу продуктивність симуляції та кращу сумісність із високою частотою оновлень фізики, що у сукупності робить його обґрунтованим вибором для даного проєкту.

2.2.4 Суглоби та ступені свободи

З'єднання між кістками фізичного скелета реалізуються за допомогою компонента `ConfigurableJoint` – найгнучкішого типу суглоба у `PhysX`. `ConfigurableJoint` дозволяє незалежно налаштовувати обмеження для кожного з шести ступенів свободи твердого тіла: трьох осей лінійного переміщення та трьох осей обертання [31]. Для кожної осі можна задати режим обмеження – вільний, заблокований або обмежений діапазоном кутів – що дозволяє точно відтворити анатомічні обмеження кожного суглоба скелета.

Крім обмежень, `ConfigurableJoint` підтримує суглобові приводи – пружинно-демпферні системи, що прагнуть привести суглоб до заданої цільової позиції або орієнтації. Привід генерує силу, пропорційну відхиленню від цілі, та демпферну силу, пропорційну швидкості наближення до неї, що забезпечує плавне відстеження цільової позиції без коливань. Саме суглобові приводи є механізмом, через який система керування передає м'язові зусилля фізичній моделі персонажа, що детальніше розглядається у підрозділі 2.4.

2.3 PID-регулятори

2.3.1 Керування зі зворотним зв'язком

Задача керування динамічною системою, що прагне утримувати певний параметр на бажаному рівні попри зовнішні збурення, є фундаментальною у теорії автоматичного керування. Ключовим поняттям тут є зворотний зв'язок – принцип, за яким керуюча система безперервно вимірює поточний стан об'єкта керування та порівнює його з бажаним. Різниця між бажаним і поточним значенням називається помилкою керування і слугує основою для формування керуючого впливу.

У контексті керування активним регдолем задача полягає у тому, щоб утримувати фізично симульоване тіло персонажа у заданому положенні та

орієнтації попри дію гравітації, зовнішніх сил та інерції. Саме для вирішення таких задач широко застосовуються пропорційно-інтегрально-диференціальні регулятори.

2.3.2 Структура PID-регулятора

PID-регулятор формує керуючий вплив $u(t)$ як суму трьох складових за формулою (2.1) [32]-[33]:

$$u(t) = K_p \times e(t) + K_i \times \int e(t) dt + K_d \times \frac{de(t)}{dt}, \quad (2.1)$$

де $e(t)$ – поточна помилка керування;

K_p , K_i , K_d – коефіцієнти пропорційної, інтегральної та диференціальної складових відповідно.

Пропорційна складова $K_p \times e(t)$ формує керуючий вплив, пропорційний поточній помилці. Чим більше відхилення від бажаного стану, тим сильніший вплив. Ця складова забезпечує основну реакцію регулятора, проте сама по собі не здатна повністю усунути сталу помилку.

Інтегральна складова $K_i \times \int e(t) dt$ накопичує помилку з часом і формує вплив, що поступово зростає, доки помилка не буде усунена. Це дозволяє компенсувати постійно діючі збурення, такі як гравітація або спротив повітря, які пропорційна складова не може повністю подолати.

Диференціальна складова $K_d \times \frac{de(t)}{dt}$ реагує на швидкість зміни помилки. Якщо помилка швидко зменшується, диференціальна складова протидіє надмірному перерегулюванню; якщо помилка швидко зростає – посилює реакцію. Ця складова фактично реалізує передбачення поведінки системи на основі її поточної динаміки.

В реалізованій системі також використано вбудований простий захист від перенакопичення інтегральної складової. Після кожного кроку оновлення регулятора вихід максимально допустимою кількістю сили, а інтегральна складова перестає накопичувати дані поза лімітом. Цей простий механізм є достатнім захистом від перерегулювання в складних умовах.

2.3.3 Інкрементальна форма PID-регулятора

Класична позиційна форма PID-регулятора, описана у попередньому пункті, обчислює абсолютне значення керуючого впливу на кожному кроці. Проте існує альтернативне формулювання – інкрементальна форма, у якій регулятор обчислює не абсолютне значення виходу, а його приріст відносно попереднього кроку за формулою (2.2), потім додаючи його до попереднього кроку за формулою (2.3) [34]-[35]:

$$\Delta u(t) = \frac{K_p \times (e(t) - e(t-1)) + K_i \times e(t) \times \Delta t + K_d \times (e(t) - 2e(t-1) + e(t-2))}{\Delta t}, \quad (2.2)$$

$$u(t) = u(t-1) + \Delta u(t), \quad (2.3)$$

де $e(t)$, $e(t-1)$, $e(t-2)$ – помилки керування на поточному, попередньому

та передпопередньому кроках відповідно;

Δt – крок часу симуляції.

Ключова відмінність від позиційної форми полягає у ролі кожної складової. Пропорційна складова у інкрементальній формі реагує не на абсолютну помилку, а на її зміну між кроками – тобто фактично є похідною помилки. Інтегральна складова реагує на поточну абсолютну помилку і таким чином виконує роль, аналогічну пропорційній складовій позиційної форми, стаючи основним рушієм керуючого впливу. Диференціальна складова

реагує на другу похідну помилки – прискорення її зміни – забезпечуючи додаткове демпфування при різких змінах. Все це робить інкрементальну форму дуже зручною при потребі в постійній підтримці деякої швидкості активного регдола, навіть якщо цільова швидкість дорівнює нулю.

Важливою практичною перевагою інкрементальної форми є вбудований захист від інтегрального насичення: оскільки вихід формується як накопичена сума приростів, обмеження абсолютного значення виходу на кожному кроці автоматично запобігає неконтрольованому зростанню інтегральної складової [36]-[37]. У реалізованій системі це досягається затискуванням сумарної сили до максимально допустимого значення після кожного оновлення, причому затиснуте значення використовується як база для наступного кроку, що є ефективним механізмом протидії перерегулюванню.

2.3.4 Переваги над простою пружиною

Проста пружина, що застосовується у фізичній симуляції, є аналогом виключно пропорційної складової регулятора: вона генерує силу, пропорційну відхиленню від положення рівноваги. Додавання демпфера відповідає введенню диференціальної складової – сили, протилежної швидкості руху [38]. Таким чином, стандартний пружинно-демпферний елемент відповідає PD-регулятору без інтегральної складової.

PID-регулятор має ряд переваг над такою системою. По-перше, інтегральна складова дозволяє усувати сталу помилку, що виникає під дією постійних збурень. По-друге, диференціальна складова реагує не лише на поточне відхилення, а й на його динаміку, що дозволяє отримати більш плавну та точну реакцію. По-третє, три незалежні коефіцієнти надають значно більшу гнучкість налаштування порівняно з простою парою жорсткість-демпфер.

2.3.5 Застосування PID до м'язових моментів

Теоретично PID-регулятори могли б бути застосовані й для керування м'язовими моментами суглобів фізичного скелета, забезпечуючи точніше відстеження цільової пози анімаційної моделі. Проте суглобові приводи `ConfigurableJoint` вже реалізують власну пружинно-демпферну систему, що є функціональним еквівалентом PD-регулятора. Нашарування PID-регулятора поверх цього механізму створило б дві взаємодіючі системи керування з потенційно нестабільною та важко передбачуваною поведінкою.

Крім того, обчислення PID для кожного суглоба кожного кроку фізичної симуляції вносить додаткове навантаження на продуктивність, яке не є виправданим з огляду на достатню якість керування, що забезпечується суглобовими приводами. З цих причин PID-регулятори у даній роботі застосовуються для керування на рівні всього тіла – балансування та переміщення – тоді як керування окремими суглобами реалізується через механізм приводів, описаний у наступному підрозділі.

2.4 М'язові моменти та пружинні сили

2.4.1 Концепція м'язового керування суглобами

Біологічний м'яз діє шляхом скорочення, створюючи момент сили навколо суглоба та змінюючи кут між сусідніми кістками. У фізичній симуляції аналогічна поведінка досягається через суглобові приводи `ConfigurableJoint`, які генерують момент обертання, що прагне привести суглоб до заданої цільової орієнтації. Сукупність таких приводів на всіх суглобах скелета утворює систему, функціонально подібну до м'язового апарату.

Кожен суглоб фізичного скелета у розробленій системі характеризується двома параметрами приводу: жорсткістю, що визначає силу, з якою суглоб прагне до цільової орієнтації, та демпфуванням, що

протидіє кутовій швидкості та забезпечує плавне, без коливань, наближення до цілі. Ці параметри масштабуються пропорційно до коефіцієнта маси відповідної кістки, забезпечуючи фізично коректне співвідношення між масою кістки та силою м'яза, що її контролює.

2.4.2 Цільова орієнтація та простір обертань

Цільова орієнтація кожного суглоба визначається поточною позою анімаційної моделі: щокадру система зчитує обертання відповідної кістки анімованого скелета та передає його як ціль суглобовому приводу фізичної кістки. Таким чином, анімаційна модель безперервно «тягне» фізичне тіло до своєї поточної пози.

Важливим аспектом є простір, у якому задається цільова орієнтація суглоба. Локальний простір визначає цільовий кут відносно батьківської кістки, що є природним для більшості суглобів скелета. Світовий простір задає орієнтацію відносно глобальних осей сцени, що може бути корисним для кісток, які повинні зберігати певну орієнтацію незалежно від положення тіла. У розробленій системі кожній кістці призначається параметр змішування між локальним та світовим просторами, що дозволяє гнучко налаштовувати поведінку окремих частин скелета відповідно до потреб конкретного стану персонажа.

2.4.3 Масштабування м'язової сили

Система підтримує глобальне масштабування м'язової сили через єдиний коефіцієнт зусилля у діапазоні від нуля до одиниці. При нульовому значенні всі суглобові приводи вимкнені і тіло поводить себе як пасивний регдол, повністю підпорядкований зовнішнім силам. При одиничному значенні приводи працюють з повною силою, забезпечуючи максимально точне відстеження анімаційної пози.

Окремим механізмом є статична напруга – мінімальний рівень м'язового зусилля, нижче якого суглоб не розслабляється навіть при низькому загальному коефіцієнті. При активації статичної напруги цільова орієнтація суглоба фіксується у нейтральному положенні, а привід підтримує мінімальну жорсткість. Це запобігає повному безконтрольному звисанню окремих частин тіла у станах з низьким м'язовим зусиллям, зберігаючи мінімальну м'язову готовність.

2.5 Зворотна кінематика

2.5.1 Пряма та зворотна кінематика

Кінематика скелетної моделі описує математичний зв'язок між кутами суглобів та положенням кінцівок у просторі [4], [39]. Розрізняють два підходи до розв'язання цього зв'язку.

Пряма кінематика визначає положення кінцівки на основі заданих кутів суглобів ланцюга. Кожна кістка трансформується відносно батьківської, і результуюче положення кінцівки є наслідком усіх послідовних обертань у ланцюгу. Цей підхід є природним для керування тулубом та хребтом персонажа, де аніматор або система керування задає кути окремих сегментів хребта, а положення верхньої частини тіла є результатом їхньої композиції.

Зворотна кінематика вирішує обернену задачу: задано бажане положення кінцівки у просторі, необхідно знайти кути суглобів, що його забезпечують. Цей підхід є природним для керування руками та ногами, де система знає, куди повинна потрапити стопа або кисть, і потребує відповідних кутів суглобів для досягнення цієї позиції. Саме зворотна кінематика використовується у розробленій системі для розміщення стоп на поверхні оточення та керування положенням рук.

2.5.2 Двокісткова ІК

Найпоширенішим випадком зворотної кінематики в анімації персонажів є двокістковий ланцюг: стегно – гомілка – стопа для ноги або плече – передпліччя – кисть для руки [40]-[41]. Для такого ланцюга існує аналітичне рішення на основі геометричних співвідношень.

Задано положення кореня ланцюга A , цільове положення кінцівки C та довжини кісток l_1 і l_2 . Відстань між коренем та ціллю $d = |AC|$ визначає кут згину у серединному суглобі B через теорему косинусів за формулою (2.4) [40]-[41]:

$$\cos(\angle ABC) = \frac{l_1^2 + l_2^2 - d^2}{2l_1 l_2}, \quad (2.4)$$

Знаючи цей кут, можна однозначно визначити положення серединного суглоба B , а отже й орієнтації обох кісток ланцюга. Напрямок згину визначається додатковим вектором підказки, що вказує у бік бажаного положення коліна або ліктя.

2.5.3 Застосування до керування кінцівками

Двокісткову ІК доречно застосовувати як для ніг, так і для рук персонажа. Для кожної кінцівки система керування може обчислювати цільову позицію кінцевої точки – положення стопи на поверхні або положення кисті у просторі – після чого ІК визначатиме необхідні кути суглобів для досягнення цієї позиції. Отримані орієнтації кісток передаються анімаційній моделі, яка у свою чергу слугує ціллю для суглобових приводів фізичного скелета.

Такий підхід дозволяє природно адаптувати положення кінцівок до геометрії оточення: стопи коректно розміщуються на нерівній поверхні, а руки реагують на зміну рівноваги тіла, не потребуючи заздалегідь записаних анімацій для кожної можливої ситуації.

2.6 Методи керування постановкою ніг

2.6.1 Реактивна модель постановки ніг

Реактивна модель постановки ніг ґрунтується на принципі зворотного зв'язку за відхиленням: крок ініціюється тоді, коли поточне положення стопи відхиляється від ідеального на величину, що перевищує допустимий поріг. Ідеальне положення стопи визначається динамічно на основі поточного положення та орієнтації тіла персонажа. Відхилення оцінюється як за лінійною відстанню між поточним та ідеальним положеннями стопи, так і за кутовою різницею орієнтацій, що дозволяє реагувати не лише на зміщення, а й на розворот тіла.

Для забезпечення стабільності ходи реактивна модель підтримує чергування кроків між ногами: у штатному режимі нова нога починає крок лише після того, як інша завершила свій. Проте за умови критичного відхилення обох стоп одночасно система переходить у режим невідкладного кроку, що дозволяє обом ногам робити крок паралельно для швидкого відновлення рівноваги.

Реактивна модель є особливо ефективною при малих швидкостях та стоячи на місці, коли кроки мають нерегулярний характер і визначаються переважно збуреннями рівноваги, а не ритмом ходи.

2.6.2 Циклічна модель постановки ніг

Циклічна модель ґрунтується на принципі фазового автомата: час поділяється на рівні цикли фіксованої тривалості, і кожна нога виконує крок

у заздалегідь визначену фазу циклу [42], [43], [44]. Дві ноги зсунуті за фазою на половину циклу відносно одна одної, що забезпечує природне чергування кроків при ходьбі та бігу. Тривалість циклу та параметри кроку адаптуються до поточної швидкості руху, дозволяючи системі природно масштабувати ритм ходи від повільної ходьби до бігу.

На відміну від реактивної моделі, циклічна не потребує явного контролю чергування кроків – він забезпечується самою фазовою структурою циклу. Це робить циклічну модель більш ритмічною та передбачуваною, що є перевагою при регулярному русі з постійною швидкістю.

2.6.3 Гібридне змішування моделей

Реактивна та циклічна моделі мають взаємодоповнювальні характеристики. Реактивна модель забезпечує точну адаптацію до збурень та нерівностей оточення, проте при високих швидкостях може генерувати нерівномірний, аритмічний ритм кроків. Циклічна модель забезпечує природний ритмічний рух при постійній швидкості, проте є менш гнучкою при різких змінах напрямку або зовнішніх збуреннях.

Гібридний підхід поєднує переваги обох моделей шляхом плавного змішування їхніх сигналів залежно від поточного режиму руху. При малих швидкостях домінує реактивна модель, забезпечуючи адаптивну реакцію на оточення. При збільшенні швидкості система поступово переходить до циклічної моделі, забезпечуючи ритмічність та природність ходи. Такий підхід дозволяє системі коректно поводитися у широкому діапазоні режимів руху без різких переходів між ними.

2.7 Навігація на основі NavMesh

2.7.1 Навігаційна сітка

Навігаційна сітка, або NavMesh, є полігональним представленням прохідного простору ігрової сцени [45]. Вона будується на основі геометрії статичних об'єктів сцени шляхом аналізу поверхонь, доступних для переміщення агента з заданими розмірами та можливостями. Результатом є набір опуклих полігонів, що покривають прохідні ділянки сцени та утворюють зв'язний граф, яким може рухатися навігаційний агент.

NavMesh абстрагує складну тривимірну геометрію сцени до плоского графу, що значно спрощує задачу планування маршруту. Кожен полігон сітки характеризується набором властивостей, зокрема типом поверхні та вартістю переміщення, що дозволяє враховувати різні умови прохідності при побудові маршруту.

2.7.2 Планування маршруту

Пошук маршруту на NavMesh зводиться до задачі пошуку найкоротшого шляху у графі полігонів між поточним положенням агента та цільовою точкою. Для вирішення цієї задачі широко застосовується алгоритм A*, що поєднує пошук у ширину з евристичною оцінкою відстані до цілі, забезпечуючи ефективне знаходження оптимального маршруту [46].

Результатом планування є послідовність точок маршруту – вершин коридору на NavMesh, що визначають напрямок руху агента. Агент послідовно рухається до кожної наступної точки маршруту, і при досягненні поточної точки активується наступна.

2.7.3 Особливості застосування до фізично керованого персонажа

Стандартний навігаційний агент у більшості ігрових рушіїв є кінематичним – він може миттєво змінювати швидкість та напрямок руху відповідно до маршруту [47]. Фізично керований персонаж, натомість, підпорядковується законам інерції: зміна швидкості та напрямку вимагає часу та залежить від прикладених сил.

Це породжує принципову відмінність у підході до навігації. Якщо кінематичний агент може рухатися з максимальною швидкістю аж до повороту і різко змінити напрямок, фізично керований персонаж повинен завчасно знижувати швидкість перед поворотами, щоб уникнути втрати рівноваги. Чим гостріший кут повороту – тим раніше і інтенсивніше необхідне гальмування. Таким чином, планування маршруту для активного регдолу нерозривно пов'язане з плануванням швидкості вздовж цього маршруту з урахуванням фізичних обмежень персонажа. Для апроксимації гальмувальної відстані можна використати формулу (2.5) [48]:

$$d = \frac{v^2 - v_t^2}{2a}, \quad (2.5)$$

де d – гальмівна відстань, м;

v – поточна швидкість персонажа, м/с;

v_t – цільова швидкість кластера, м/с;

a – параметр розрахункового сповільнення, м/с².

2.8 Висновки до розділу 2

У другому розділі розглянуто теоретичну основу та методи, що були використані при розробці системи керування активним регдолом.

Визначено Unity як середовище розробки, що надає необхідний інструментарій для реалізації системи: фізичний рушій PhysX з підтримкою твердих тіл та суглобів, розвинену систему анімації та чітко визначений життєвий цикл оновлення, що дозволяє коректно розділити фізичні обчислення та анімаційні оновлення між FixedUpdate та LateUpdate.

Розглянуто фізичну симуляцію твердих тіл як основу фізичної моделі персонажа. Показано, що ConfigurableJoint є ключовим компонентом з'єднання кісток скелета, що надає гнучке керування ступенями свободи та підтримує суглобові приводи для відтворення цільової пози. Обґрунтовано вибір режиму Discrete для виявлення колізій як найбільш стабільного та передбачуваного рішення для системи з великою кількістю з'єднаних твердих тіл.

Досліджено PID-регулятори як основний інструмент керування на рівні всього тіла. Розглянуто позиційну та швидкісну форми регулятора, їхні відмінності у ролі складових та практичні переваги кожної. Показано переваги PID над простою пружинно-демпферною системою та обґрунтовано рішення не застосовувати PID для керування окремими суглобами.

Описано механізм м'язових моментів через суглобові приводи як спосіб передачі цільової пози анімаційної моделі фізичному скелету, розглянуто роль простору обертань та масштабування м'язової сили.

Розглянуто зворотну кінематику як метод керування кінцівками: показано відмінність від прямої кінематики та описано аналітичне рішення для двокісткового ланцюга.

Описано реактивну та циклічну моделі постановки ніг як взаємодоповнювальні підходи, що поєднуються у гібридній системі для забезпечення коректної поведінки у широкому діапазоні режимів руху.

Розглянуто навігацію на основі NavMesh та обґрунтовано необхідність планування швидкості вздовж маршруту для фізично керованого персонажа як принципову відмінність від стандартного кінематичного агента.

3 ОПИС ПРОГРАМИ

3.1 Загальна архітектура системи

3.1.1 Принцип подвійного скелету

Основою архітектури системи є принцип подвійного скелета: кожен персонаж NPAR одночасно містить дві незалежні скелетні моделі, що функціонують паралельно. Перша – анімована модель – керується компонентом Animator рушія Unity та слугує динамічним джерелом цільових поз. Друга – фізична модель – являє собою ланцюг твердих тіл Rigidbody, з'єднаних суглобами ConfigurableJoint, і безпосередньо бере участь у симуляції фізичного рушія. Анімована модель ніколи не стикається з об'єктами оточення і не підпорядковується гравітації – вона існує виключно як орієнтир для фізичної. Схему синхронізації подвійного скелету можна побачити нижче (рис. 3.1).

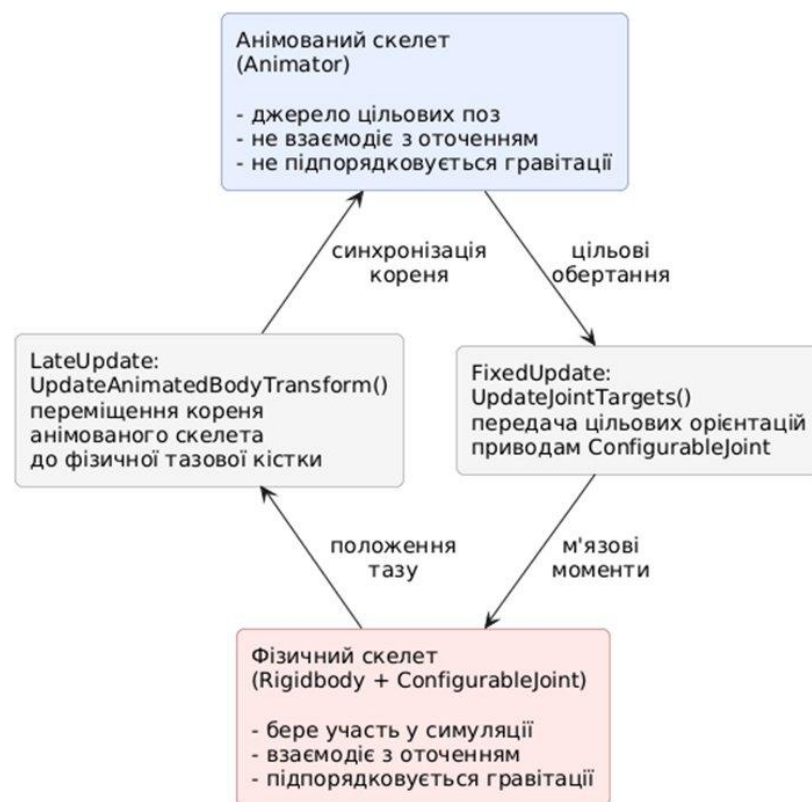


Рисунок 3.1 – Принцип подвійного скелета (схема синхронізації)

Зв'язок між відповідними кістками двох скелетів описується структурою `RagdollBodyBoneData`. Кожен екземпляр цієї структури зберігає посилання на `Transform` (позиція, поворот, розмір) анімованої кістки, `Transform` фізичної кістки, компонент `Rigidbody`, компонент `ConfigurableJoint`, а також коефіцієнт маси кістки та параметр змішування простору обертань. Список таких структур для всіх кісток скелета зберігається у `NPARMainController`. Під час ініціалізації він перетворюється на словник з ключем типу `HumanBodyBones`, що забезпечує швидкий доступ до даних будь-якої кістки.

Синхронізація відбувається у два етапи кожного фізичного кадру. Спочатку, у `FixedUpdate`, метод `UpdateJointTargets` зчитує поточне обертання кожної анімованої кістки та передає його як цільову орієнтацію відповідному приводу `ConfigurableJoint` фізичної кістки. Привід генерує момент сили, що прагне привести фізичну кістку до цієї орієнтації, протидіючи при цьому гравітації та зовнішнім впливам. Таким чином, анімаційна поза безперервно «тягне» фізичне тіло до себе через м'язові зусилля суглобів. Потім, у `LateUpdate`, вже після завершення фізичної симуляції, `UpdateAnimatedBodyTransform` переміщує корінь анімованого скелета так, щоб його тазова кістка збігалася з положенням фізичної тазової кістки. Завдяки цьому анімована модель завжди залишається поряд із фізичним тілом і продовжує задавати коректні цільові пози для суглобів наступного кадру.

Загальна маса персонажа задається єдиним параметром і розподіляється між кістками фізичного скелета пропорційно до їхніх коефіцієнтів маси під час ініціалізації. Це дозволяє налаштовувати фізичну поведінку окремих частин тіла, зберігаючи загальну масу персонажа незмінною. Візуалізацію анімованого скелета поверх фізичного тіла представлено на рис. 3.2.



Рисунок 3.2 – Анімований скелет поверх фізичного тіла

3.1.2 Ієрархія компонентів та потік даних

Система NPAR реалізована як набір окремих компонентів MonoBehaviour, що розміщуються на одному ігровому об'єкті Unity. Центральним з них є NPARMainController – він є єдиним обов'язковим компонентом і виступає точкою входу для всієї системи. Решта контролерів є необов'язковими: система коректно функціонує за їхньої відсутності, поступово втрачаючи відповідну функціональність.

Під час ініціалізації NPARMainController самостійно відшукує на об'єкті всі наявні контролери та послідовно їх ініціалізує, передаючи кожному посилання на себе та спільний словник кісткових даних. Порядок ініціалізації є фіксованим: спочатку NPARMuscleController, потім NPARBodyController, NPARArmsController, NPARLegsController, NPARNavigationController, NPARAnimationController і, останнім, NPARBrain.

Такий порядок гарантує, що до моменту ініціалізації NPARBrain усі контролери, від яких він залежить – NPARPresetManager, NPARNavigationController та NPARAnimationController – вже готові до роботи. Компонентну діаграму представлено на рис. 3.3, а власне компоненти у редакторі рушія на рис. 3.4.

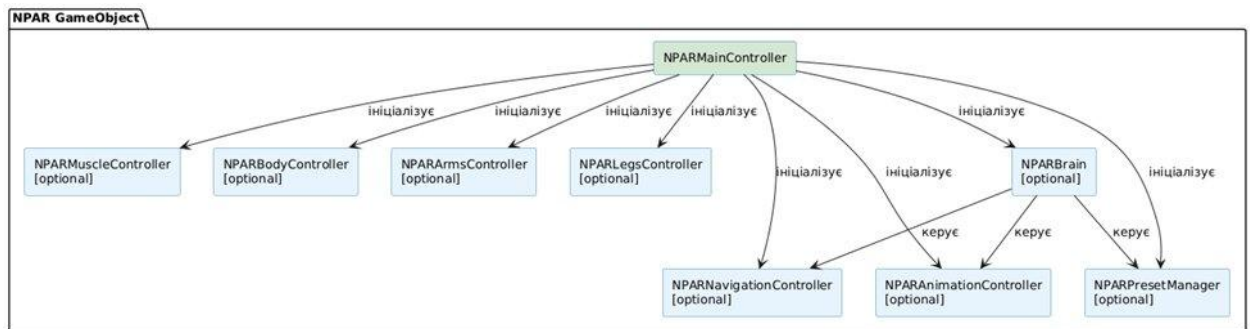


Рисунок 3.3 – Компонентна діаграма системи NPAR

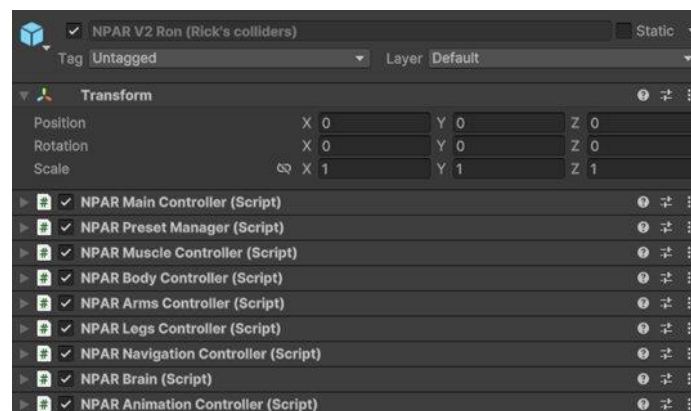


Рисунок 3.4 – Повна система NPAR у редакторі рушія

Потік даних у кожному фізичному кадрі організований у кілька послідовних етапів. У FixedUpdate NPARMainController першим оновлює спільний стан, наприклад, центр мас, вектори напрямку, точки балансу. Ці значення зберігаються як публічні властивості і є основним джерелом інформації для всіх контролерів. Після цього NPARMainController застосовує фізичні сили до тазової кістки та оновлює цільові орієнтації суглобів. Щоб завжди бути впевненим у виконанні NPARMainController першим, він має пріоритет у Script Execution Order рушія Unity. Паралельно у своїх власних

викликах `FixedUpdate` виконуються незалежні контролери – `NPARMuscleController` масштабує жорсткість м'язових груп, `NPARBodyController` керує поворотом хребта, `NPARArmsController` обчислює цільові позиції рук, а `NPARLegsController` оновлює постановку стоп. `NPARNavigationController` також оновлюється у `FixedUpdate` – він перераховує маршрут, напрямки погляду і переміщення та швидкість, які `NPARMainController` читає наступного кадру. `NPARBrain` виконується в `Update`, оскільки його завдання – прийняття рішень і зміна станів – не прив'язані до фізичного кроку симуляції.

У `LateUpdate`, після завершення всіх фізичних обчислень і рендерингу, `NPARMainController` переміщує корінь анімованого скелета до поточного положення фізичних стегон та оновлює трансформ центру мас. Весь потік даних системи представлено на рис. 3.5.

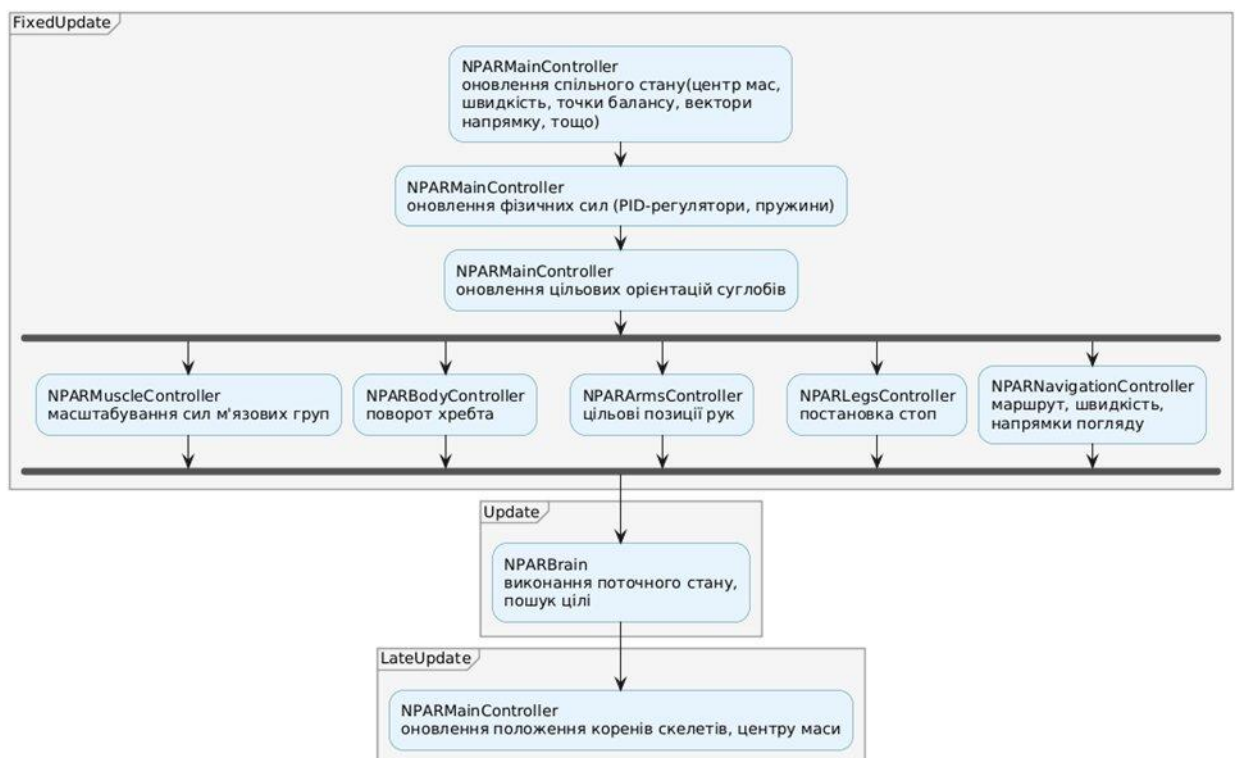


Рисунок 3.5 – Потік даних системи NPAR за один кадр симуляції

Такий розподіл відповідальності дозволяє легко вмикати і вимикати окремі підсистеми, а також замінювати їхню поведінку через систему

пресетів, описану у підрозділі 3.2. Більше про роль кожного контролеру фізичної симуляції описано у підрозділі 3.4.

3.2 Система пресетів і налаштування поведінки

Усі числові параметри кожного контролера зосереджені в окремому класі даних – `XxxControllerData` (наприклад, `MainControllerData`, `MuscleControllerData`). Це звичайні серіалізовані класи `C#`, не `MonoBehaviour` і не `ScriptableObject`. Вони успадковують абстрактний базовий клас `ControllerDataBase<T>`, що визначає єдиний обов’язковий метод `ApplyFrom` – копіювання значень з іншого екземпляра даних у поточний. Кожен контролер зберігає один живий екземпляр свого класу даних у публічній властивості `Data` і звертається до нього під час кожного кадру.

Пресет – це `ScriptableObject`, що успадковує `PresetBase<T>`. Він містить лише два поля: екземпляр відповідного класу даних та час застосування `ApplyTime`. Таким чином, пресет є суто контейнером для значень без будь-якої власної логіки. На рівні файлової системи проєкту пресети зберігаються як ассети Unity та можуть редагуватися в інспекторі без зміни коду. Інтерфейс налаштувань пресета в редакторі рушія представлено на рис. 3.6.

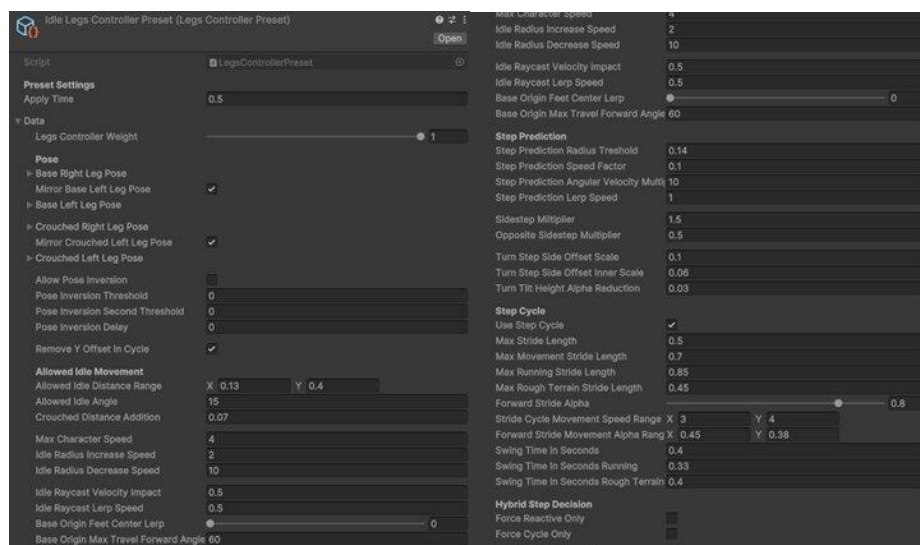


Рисунок 3.6 – Приклад пресета контролера ніг в редакторі рушія

Для одночасного перемикання поведінки всього персонажа використовується NPARBehaviorPresetSO – ScriptableObject, що об'єднує по одному пресету для кожного контролера: MainControllerPreset, MuscleControllerPreset, BodyControllerPreset, ArmsControllerPreset та LegsControllerPreset. Застосування одного такого поведінкового пресету одночасно оновлює фізичні коефіцієнти, силу м'язів, параметри ІК та поведінку ніг без жодних змін у коді. Структуру даних системи пресетів представлено на рис. 3.7, а конфігурацію пресета поведінки в редакторі рушія на рис. 3.8.

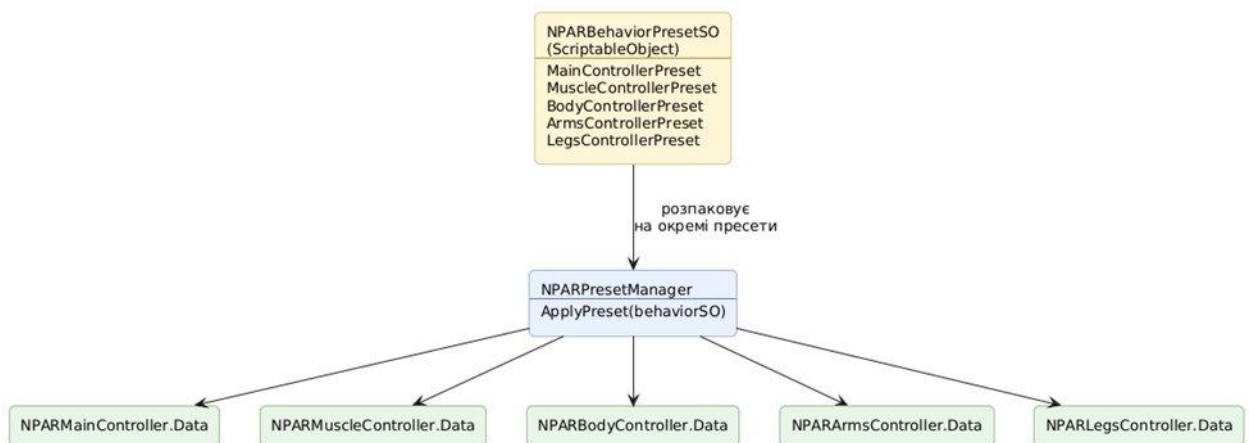


Рисунок 3.7 – Структура даних системи пресетів

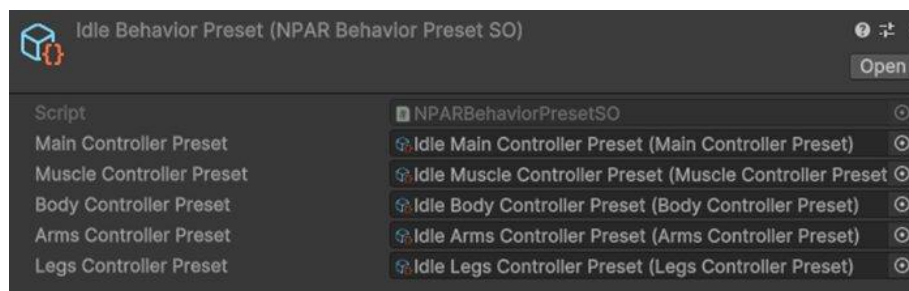


Рисунок 3.8 – Приклад NPARBehaviourPresetSO в редакторі рушія

Диспетчером застосування пресетів є NPARPresetManager. Він зберігає посилання на всі контролери та поточно активні пресети для кожного з них. При виклику ApplyPreset(NPARBehaviorPresetSO) менеджер

розпаковує поведінковий пресет на окремі пресети контролерів, а потім для кожного викликає `ApplyPreset`, що у свою чергу викликає `Data.ApplyFrom` відповідного контролера.

Якщо передано прапорець `lerp: true`, `ApplyFrom` не копіює значення миттєво, а запускає окрему корутину для кожного параметра через `PresetUtils.StartLerp`. Кожна така корутина інтерполює числове значення від поточного до цільового протягом часу `ApplyTime`, заданого у пресеті. Корутини ідентифікуються рядковим ключем – іменем поля – що дозволяє при повторному застосуванні пресету зупинити вже активну корутину для того самого параметра і замінити її новою. Завдяки цьому перехід між пресетами є плавним навіть якщо він відбувається до завершення попереднього переходу.

Застосування пресетів захищене прапорцем `allowedToChangePreset` у `NPARBrain`. Якщо він вимкнений, виклики `ApplyPreset` ігноруються, якщо тільки не передано параметр `forced: true`. Це дозволяє зовнішнім системам все одно внести зміни в поведінку NPAR, наприклад при налаштуванні параметрів.

`NPARPresetManager` дозволяє керувати пресетами не тільки під час запуску програми, але й в редакторі рушія. При зовнішньому керуванні пресетів завжди ставляться прапорці `lerp: false`, `forced: true`, тобто розробник може в ручному форматі завжди змінити поточний пресет, навіть якщо програма не виконується щоб провести повноцінне інтерполювання. Інтерфейс менеджера пресетів в редакторі рушія представлено на рис. 3.9.

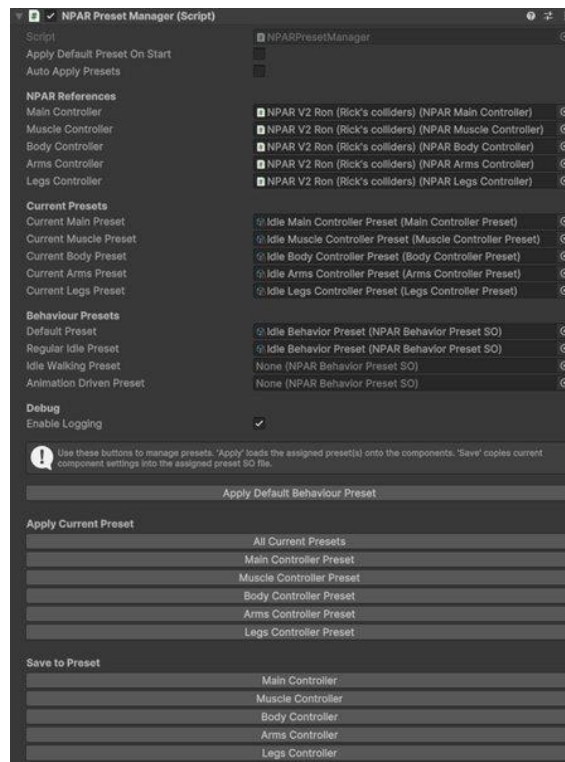


Рисунок 3.9 – NPARPresetManager у редакторі рушія

3.3 Машина станів та прийняття рішень

3.3.1 Стани системи та переходи між ними

Поведінка персонажа керується простою машиною станів, реалізованою в NPARBrain. Кожен стан є серіалізованим класом, що реалізує інтерфейс `IState` з трьома методами: `Enter`, `Execute` та `Exit`. Базовий клас `StateBase` надає спільні захищені посилання на основні компоненти системи. Залежності передаються у стани через структуру `StateDependencies` під час ініціалізації – це забезпечує відокремлення станів від конкретних посилань на сцені.

Система містить три стани. `IdleState` є початковим станом – він застосовує пресет стояння, встановлює навігаційний намір за замовчуванням і, якщо задана точка маршруту, надсилає її навігаційному контролеру. `FallState` активується при втраті стабільності й відповідає за весь цикл падіння та підйому. `FightState` відповідає за поведінку при наявності ворога, але він нерелевантний до теми роботи, тому далі його буде ігноровано.

Переходи між станами перевіряються у методі `Execute` поточного стану щокадру. `IdleState` переходить у `FallState` при `brain.IsStable == false`. `FallState` повертається в `IdleState` після завершення циклу підйому із затримкою, що дає персонажу час стабілізуватися після вставання. Зміна стану захищена прапорцем `allowedToChangeState` у `NPARBrain` – якщо він вимкнений, виклик `ChangeState` ігнорується.

`FallState` внутрішньо поділяється на чотири фази: `Fall`, `Lie`, `PrepareToGetUp` та `GetUp`. У фазі `Fall` система перевіряє швидкість центру мас та кутову швидкість стегон, очікуючи повної зупинки персонажа. Додатково, якщо середня кутова швидкість перевищує поріг `midFallGroupUpThreshold`, запускається анімація групування, що зменшує ризик неприродного відтворення падіння при обертанні. Після зупинки система переходить у `Lie`, де очікує фіксований час, потім у `PrepareToGetUp` для поступового відновлення м'язового тону, і нарешті в `GetUp`, де запускається анімація вставання і через `stateSwitchDelayAfterGetUp` секунд відбувається повернення в `IdleState`. Правила переходу прості, але дозволяють інтеграції у майбутньому, наприклад, з системою здоров'я. Деталі системи можна побачити на діаграмі нижче (рис. 3.10).

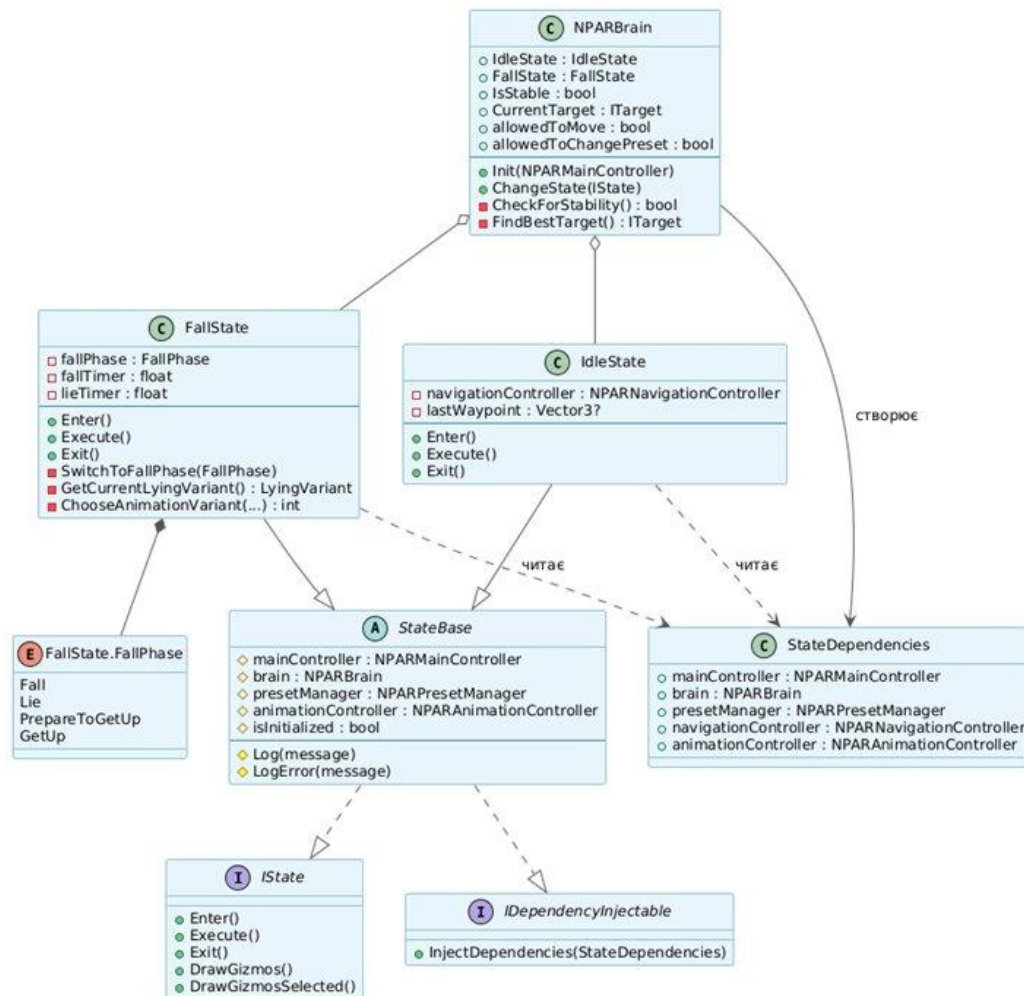


Рисунок 3.10 – Діаграма класів машини станів NPAR

Вибір анімаційного варіанту на кожній фазі залежить від поточного стану персонажа. Напрямок падіння визначається за вектором швидкості центру мас відносно PseudoForward: при русі вперед обирається варіант падіння на живіт, назад – на спину, при бічній або хаотичній траєкторії – варіант групування. Положення на підлозі для фаз Lie, PrepareToGetUp та GetUp визначається за кутом між вектором правого боку стегон та вертикаллю: якщо кут менший за sideAngleThreshold, персонаж вважається таким, що лежить на боці, інакше обирається спина або живіт залежно від того, який із векторів – передній чи задній – ближчий до вертикалі. Детальніше зв'язки між станами можна побачити на рис. 3.11.

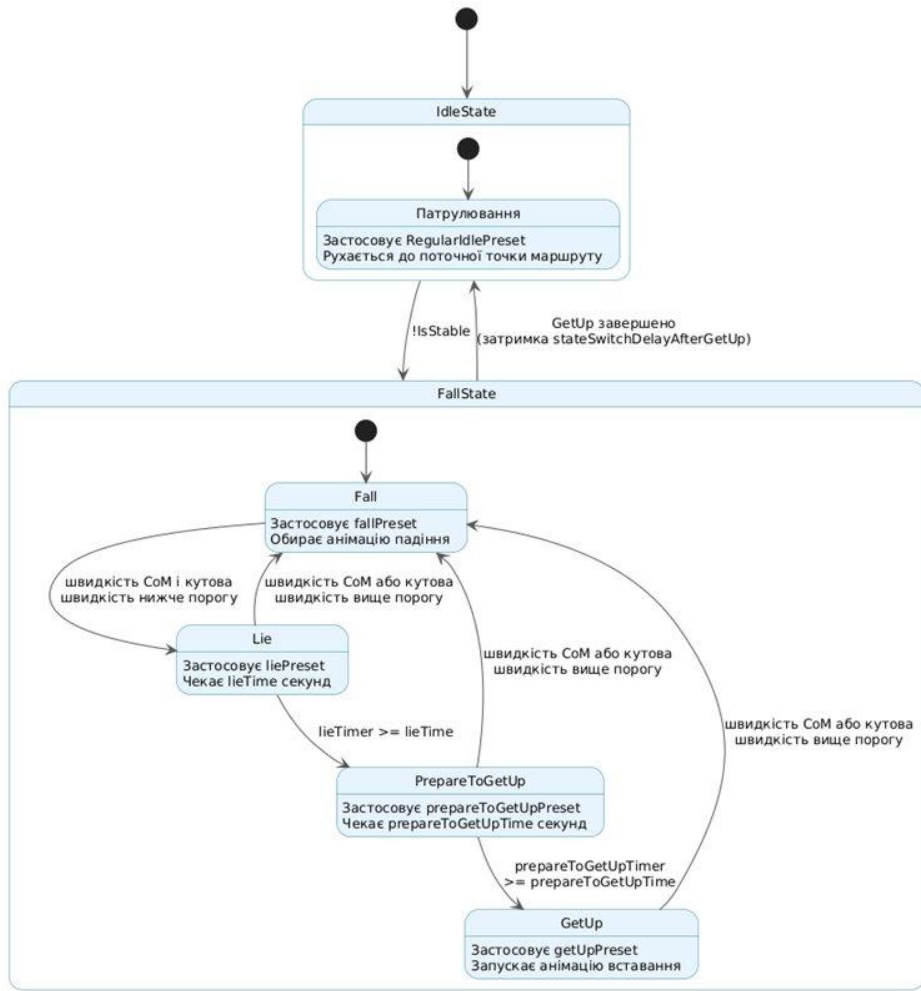


Рисунок 3.11 – Діаграма станів NPAR

IdleState кожен кадр перевіряє IsStable, який оновлює NPARBrain, і переключається на FallState, якщо IsStable: false. Уся послідовність фаз FallState виконується за рахунок переключення між відповідними пресетами і варіантами анімацій. В цьому випадку система NPARPresetManager сяє і дозволяє динамічно змінювати кожен аспект поведінки NPAR в залежності від оточення і ситуації. На кожному кроці послідовності підіймання на ноги FallState продовжує перевіряти поріг швидкості, який якщо перевищити, фаза послідовності скидається назад до FallPhase.Fall.

3.2.2 Оцінка стабільності

Стабільність персонажа оцінюється у FixedUpdate методом CheckForStability і зберігається у властивості IsStable. Оцінка ґрунтується на п'яти метриках, кожна з яких відстежується через TimedValueTracker – структуру ковзного вікна, що зберігає значення за останній відрізок часу і дозволяє отримати як середнє значення, так і його дельту між кроками. Метрики такі: горизонтальна швидкість центру мас, вертикальна швидкість центру мас, відстань між центром мас і точкою балансу, кутова швидкість стегон по осях нахилу (pitch/roll) та по осі повороту (yaw).

Перевірка виконується у два рівні. Перший рівень – прості умови, що не потребують часового вікна: якщо персонаж у повітрі, кутова швидкість мала, а нахил тіла перевищує airborneAllowedTiltAngleForLanding, система вважає, що приземлення буде невдалим. Також перевіряється кут відхилення між фізичним і анімованим векторами від стегон до верху тіла – якщо він перевищує maxPhysicalAnimatedBodyDeviationAngle, система вважає тіло нестабільним. Другий рівень перевіряє середні значення кожної метрики за часовим вікном відносно абсолютних порогів, а також дельти цих середніх для виявлення різких змін, наприклад від раптового удару.

3.4 Контролери фізичної симуляції

3.4.1 Головний контролер (NPARMainController)

NPARMainController є центральним компонентом системи – він ініціалізує всі підсистеми, обчислює спільний стан персонажа кожного фізичного кадру та застосовує фізичні сили до тазової кістки. Усі інші контролери читають обчислені ним дані через публічні властивості.

Щокадру контролер обчислює три вектори, що описують орієнтацію та рух персонажа. PseudoForward – горизонтальна проєкція напрямку вперед фізичних стегон, що коректно обробляє перевернутий стан тіла: при падінні

на спину або живіт вектор плавно переходить до напрямку вздовж тіла, не роблячи різких стрибків. FaceForward – напрямок, у який персонаж хоче дивитися; плавно слідує за DesiredFacingRotation з навігаційного контролера. TravelForward – фактичний напрямок руху; при русі спиною вперед автоматично інвертується з гістерезисом, щоб уникнути мерехтіння на межі переключення.

Центр мас обчислюється як зважена сума позицій усіх твердих тіл скелета з урахуванням їхніх мас. Точка балансу BalanceCenter визначається як середня точка між позиціями стоп із доданим зміщенням, що залежить від поточного пресету та ступеня присідання. Саме відхилення проєкції центру мас від точки балансу є основним сигналом помилки для PID-регуляторів.

У кожному FixedUpdate до тазової кістки послідовно застосовується кілька сил. Позиційна пружина притягує фізичні стегна до анімованих. Обертальна пружина реалізована через привід ConfigurableJoint тазової кістки, цільова орієнтація якого встановлюється за анімованими стегнами. Вертикальна сила компенсує просідання персонажа під час кроків; при опорі лише на одну ногу вона масштабується окремим коефіцієнтом. Але усі ці сили залишаються як допоміжні про всяк випадок, основна робота лежить на PID-регуляторах.

NPAR користується трьома різними PID-регуляторами, які відповідають за різні аспекти контролю тілом персонажа. Позиційний PID-регулятор прагне утримати проєкцію центру мас над точкою балансу, швидкісний – забезпечити цільову швидкість руху від навігаційного контролера. Обертальний PID-регулятор утримує орієнтацію стегон відповідно до анімованої моделі. При активному русі ($\text{MovementLerp} > 0.5$) коефіцієнти всіх трьох PID-регуляторів автоматично перемикаються на рухові значення через механізм перевизначень. Схему контурів зворотнього зв'язку PID-регуляторів можна побачити нижче (рис. 3.12).

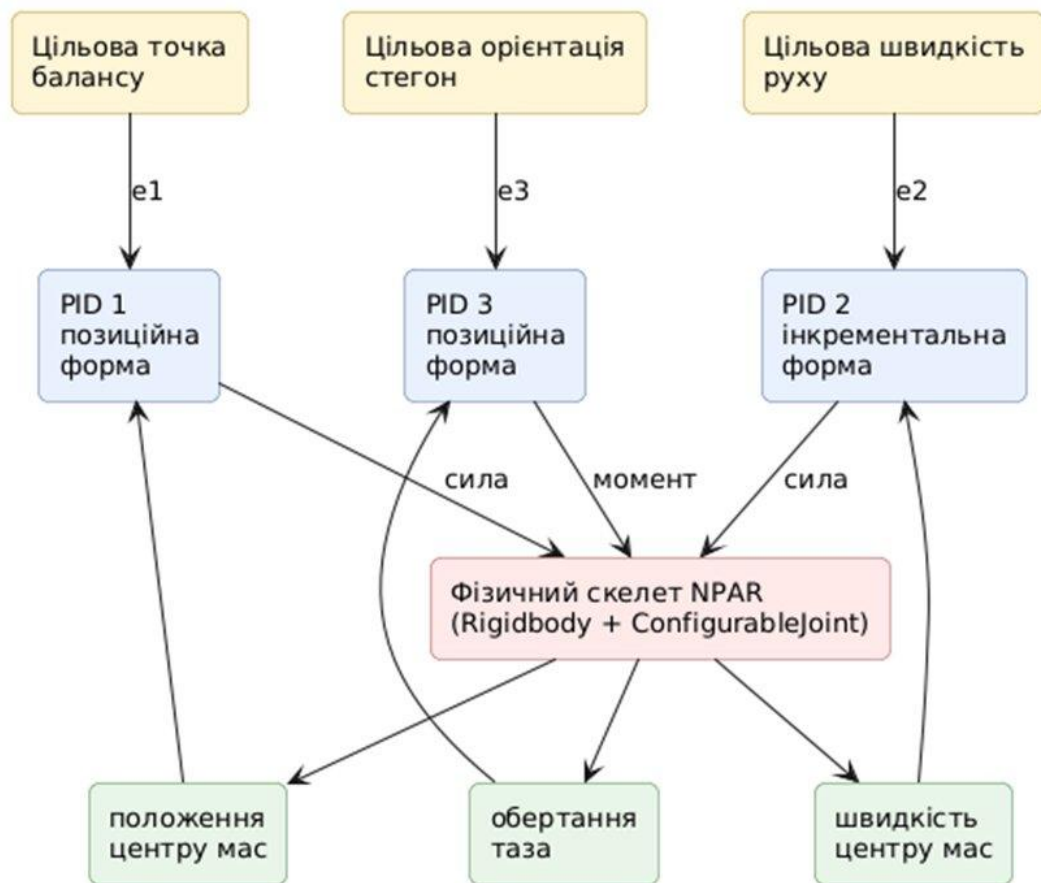


Рисунок 3.12 – Схема контурів зворотнього зв'язку PID-регуляторів

У LateUpdate контролер переміщує корінь анімованого скелета так, щоб його тазова кістка збігалася з фізичною. Режим позиціонування налаштовується через AnimatedHipsPositionMode і може відслідковувати фізичні стегна, центр стоп, точку балансу або залишатися на місці. Режим обертання налаштовується через AnimatedHipsRotationMode: режим Balance обчислює цільовий нахил тіла на основі відхилення центру мас від точки балансу та нормалізованої сили PID-регулятора, додає нахил у повороті та компенсацію кроку тазом; режим StandDynamic слідує за PseudoForward, використовуючи тільки поворот навколо осі Y; режим StandFixed працює так само, як і попередній, але запам'ятовує поворот навколо осі Y тільки один раз; режим MatchPhysical слідує за поворотом фізичної тазової кістки; MatchTarget відтворює задану ззовні орієнтацію напрямку. Розробницьку відмальовку головного контролера можна побачити нижче (рис. 3.13).

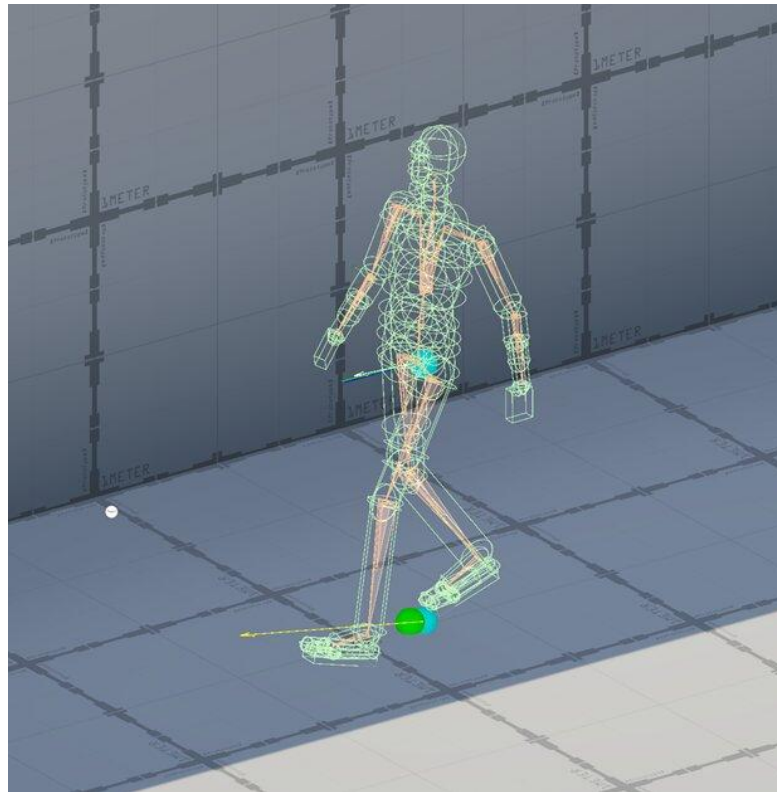


Рисунок 3.13 – Розробницька відмальовка головного контролера в рушії

3.4.2 Контролер м'язів (NPARMuscleController)

М'язовий контролер відповідає за глобальне масштабування жорсткості (сили) всіх суглобів фізичного скелета. Його архітектура навмисно мінімальна: контролер не приймає рішень щодо конкретних поз чи рухів, а лише керує тим, наскільки сильно фізичне тіло в цілому прагне відтворити анімаційну позу.

Контролер оперує списком груп м'язів `RagdollBodyMuscleGroup`, кожна з яких об'єднує кілька кісток зі спільними налаштуваннями жорсткості та демпфування. Щокадру у `FixedUpdate` обчислюється єдиний скалярний коефіцієнт `MuscleStrength` у діапазоні від нуля до одиниці на основі параметра `Effort` із поточних даних пресету. Цей коефіцієнт передається кожній групі, яка масштабує ним жорсткість і демпфування приводів своїх суглобів.

При `MuscleStrength` близькому до нуля суглобові приводи майже вимкнені і тіло поводить як пасивний регдол, повністю підпорядкований зовнішнім силам. При значенні близькому до одиниці приводи працюють з повною силою і фізичне тіло щільно слідує анімаційній позі. Перемикання між пресетами з різними значеннями `Effort` через механізм інтерполяції, описаний у підрозділі 3.2, дозволяє плавно змінювати загальний м'язовий тонус персонажа – наприклад, поступово розслабляти тіло при переході у стан падіння.

Окремим механізмом є статична напруга окремих кісток: при її активації кістка має мінімальну напругу, при якій цільова орієнтація суглоба фіксується у нейтральному положенні незалежно від поточної анімації, що запобігає безконтрольному звисанню кінцівок при низькому загальному `MuscleStrength`. Деталі можна побачити на діаграмі класів контролера м'язів на рис. 3.14.

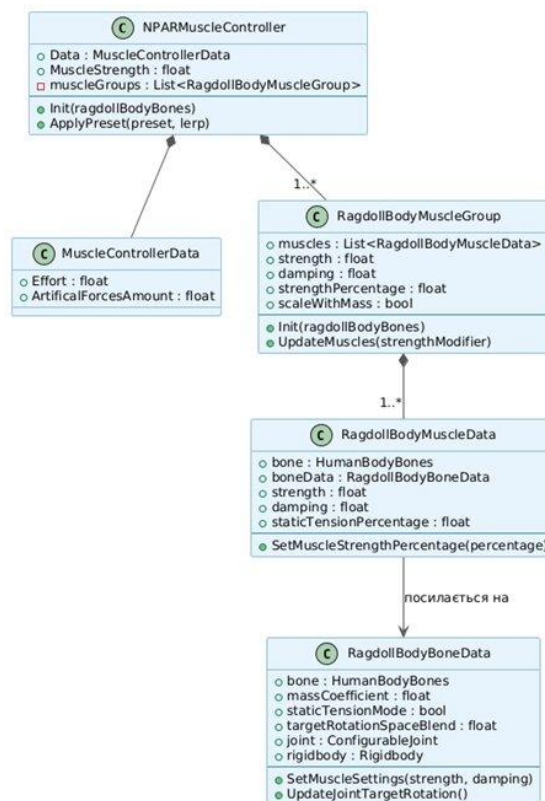


Рисунок 3.14 – Діаграма класів контролера м'язів

3.4.3 Контролер тіла (NPARBodyController)

Контролер тіла відповідає за обертання хребта персонажа. Його завдання — нахилити верхню частину тіла у відповідь на баланс та напрямок погляду, розподіляючи загальний нахил між трьома сегментами хребта.

Структура хребта. Контролер оперує трьома сегментами – waist, spine01, spine02 – кожен з яких має власний MultiRotationConstraint в анімаційному ризі та параметр rotationScale, що визначає яку частку загального нахилу поглинає цей сегмент. Завдяки цьому нахил тіла розподіляється природно по всьому хребту, а не концентрується в одному суглобі.

Обчислення цільового обертання. Щокадру у FixedUpdate накопичується сумарне цільове обертання targetTotalRotation з двох складових. Перша – AnimatedHipsRotationTargetOffset – зчитує AnimatedHipsBaseRotation з головного контролера, нормалізує кути Ейлера до діапазону $[-180, 180]$, масштабує кожну вісь окремо через PerAxisTiltScale та змішує результат через AnimatedHipsRotationScale. Це дозволяє верхній частині тіла реагувати на нахил стегон, компенсуючи або посилюючи його залежно від налаштувань пресету. Друга складова – FaceForwardOffset – додає поворот хребта навколо вертикальної осі, коли кут між PseudoForward та FaceForward перевищує поріг FaceForwardRotationThreshold. Це дозволяє верхній частині тіла повертатися у бік руху раніше, ніж повернуться стегна. Перехід реалізований з гістерезисом – вмикається при перевищенні верхнього порогу і вимикається лише при падінні нижче нижнього.

Компенсація кроку. Перед застосуванням до сегментів до сумарного обертання додається компенсація кроку: обертання, протилежне PelvisStrideRotation, масштабоване коефіцієнтом StrideCompensationScale. Це є компенсування поворотної швидкості стегон під час ходи.

Розподіл по сегментах. Згладжене через SmoothDamp сумарне обертання розподіляється між сегментами через SlerpUnclamped від

нейтрального положення до повного обертання з коефіцієнтом `rotationScale` кожного сегмента. Результат записується як локальне обертання цільового трансформу відповідного `MultiRotationConstraint`.

3.4.4 Контролер рук (`NPARArmsController`)

Контролер рук керує положенням кистей персонажа через двокісткову ІК, описану у підрозділі 2.5. Він має два режими роботи: локомоційний та бойовий. Оскільки бойовий режим не є предметом цієї роботи, далі описується лише локомоційний.

На початку кожного кадру контролер обчислює два кути нахилу тіла відносно `PseudoForward`: `bodyTiltX` – нахил вперед/назад та `bodyTiltZ` – бічний нахил. Вони отримуються як кути Ейлера різниці між вектором від стегон до верху анімованого та фізичного скелетів у локальному просторі `PseudoForward`. Ці два значення є спільним входом для всіх подальших обчислень кадру.

Базова траєкторія рук. Положення рук вздовж базової траєкторії визначається параметром `leftToRightSwing` – значенням у діапазоні $[0, 1]$, що відображає взаємне розташування стоп вздовж `FaceForward`: коли ліва нога попереду, значення наближається до 1, коли права – до 0. Контролер обчислює позицію кожної руки, оцінюючи криву профілю `WalkPathProfile` в точці `leftToRightSwing` для лівої руки та $1 - \text{leftToRightSwing}$ для правої. Паралельно оцінюється вертикальний профіль `VerticalWalkPathProfile`, і результати змішуються залежно від різниці висот стоп, швидкості та ступеня присідання. Завдяки цьому при русі сходами або нерівною поверхнею руки природно піднімаються вище.

Балансові зміщення. Поверх базової траєкторії адитивно накладаються три незалежні корекції. Бічний підйом: обидві руки піднімаються в сторони, коли магнітуда нахилу тіла перевищує `SideLiftBodyTiltRange` або при присіданні. Кут підйому плавно наближається

до цільового через `MoveTowards`, а його нормалізоване значення `currentSideLiftAlpha` масштабує дві наступні корекції. Асиметричне вертикальне зміщення: одна рука піднімається, інша опускається залежно від бічного нахилу `bodyTiltZ` та бічної компоненти нормалізованої сили PID-регулятора. Симетричне переднє зміщення: обидві руки рухаються вперед при від'ємному `bodyTiltX` або при високій передній компоненті PID-сили, а також при присіданні.

Також присутні кругові махи руками при критичних обставинах. Ступінь кругових махів рук визначається параметром `currentSwingBlend`: він зростає коли `bodyTiltX` перевищує поріг `SwingBodyTiltRange`, та примусово встановлюється в 1 при відриві від землі. При ненульовому `currentSwingBlend` обидві руки обертаються в протифазі навколо спільного конуса: права рука розміщується під кутом `currentSwingCircleAngle`, ліва – під кутом $180^\circ - \text{currentSwingCircleAngle}$. Швидкість обертання `SwingSpeed` масштабується знаком `bodyTiltX`, що автоматично синхронізує напрямок маху з напрямком нахилу. Позиція кисті реконструюється як $\text{pivot} + \text{swungDir} * \text{armLength}$, що зберігає довжину руки незалежно від кута конуса.

Кожна рука описується окремим екземпляром класу `ArmData`, що зберігає всі необхідні посилання та внутрішній стан. З посилань ключовими є: `TwoBoneIKConstraint` – компонент Unity Rigging, що виконує безпосередній ІК-розрахунок; `MultiRotationConstraint` на ключиці – для керування плечем; `TwoBoneIKHintController` – визначає напрямок згину ліктя. Окремо зберігаються три анкери: `AnchorClavicle` – точка відліку для позиції ІК-цілі, `AnchorChest` – точка відліку для обертання, `StaticAnchorUpperArm` – позиція плеча у позі спокою, прив'язана до грудей і ніколи не переміщується ІК. Позиції цілей рук зберігаються у локальному просторі відносно `AnchorClavicle` (позиція) та `AnchorChest` (обертання), що робить їх незалежними від світової позиції і поточної пози рук персонажа. При ініціалізації кешуються довжини сегментів руки `upperArmLength`, `lowerArmLength` та `totalArmLength`.

Плечовий суглоб керується через `MultiRotationConstraint` на ключиці, що зміщує її відносно анімованої пози. Метод `ApplyShoulderRotation` обчислює кутове зміщення у двох площинах: горизонтальній (вперед/назад) та вертикальній (вгору/вниз). Для цього порівнюється напрямок від ключиці до `StaticAnchorUpperArm` – положення плеча у позі спокою – з напрямком від ключиці до передбаченої позиції ліктя. Різниця розкладається на кути Ейлера, кожен з яких незалежно зважується, затискається в межах та згладжується через `SmoothDamp`. Використання `StaticAnchorUpperArm` замість живої позиції плечової кістки є дуже важливим: якби для розрахунку використовувалась жива кістка, результат попереднього кадру впливав би на вхід поточного, що призводило б до нестабільності.

Метод `UpdateArm` виконує два проходи. У першому проході плечовий `MultiRotationConstraint` оновлюється на основі поточної позиції плечової кістки – ІК ще не виконувався, тому кістка знаходиться у стані попереднього кадру. У другому проході обчислюється де опиниться плечова кістка після застосування щойно розрахованого зміщення ключиці, і ця передбачена позиція використовується для розрахунку фінального повороту кисті.

Метод `SolveArmRotation` виконує три кроки. Спочатку базове обертання кисті обчислюється як добуток обертання `AnchorChest` та вхідного повороту з пресету. Потім передбачається позиція ліктя через теорему косинусів із урахуванням напрямку підказки від `TwoBoneIKHintController`. На основі передбаченого вектора передпліччя обчислюється корекційне обертання, що вирівнює вісь «вгору» кисті вздовж передпліччя – при цьому компонента *Z* обнуляється, щоб уникнути небажаного крену. Нарешті до результату додається параметр `additionalX` для тонкої ручної корекції нахилу кисті. Це робиться щоб полегшити задання натуральних позицій для кісті. Нахил кісті «вгору» майже завжди виглядає ненатурально при розслабленому контролі руками, тому, не зважаючи на цільову позицію зворотної кінематики, нахил уздовж цієї осі винесено в окремий параметр.

Розробницьку відмальовку контролера рук можна побачити нижче (рис. 3.15).

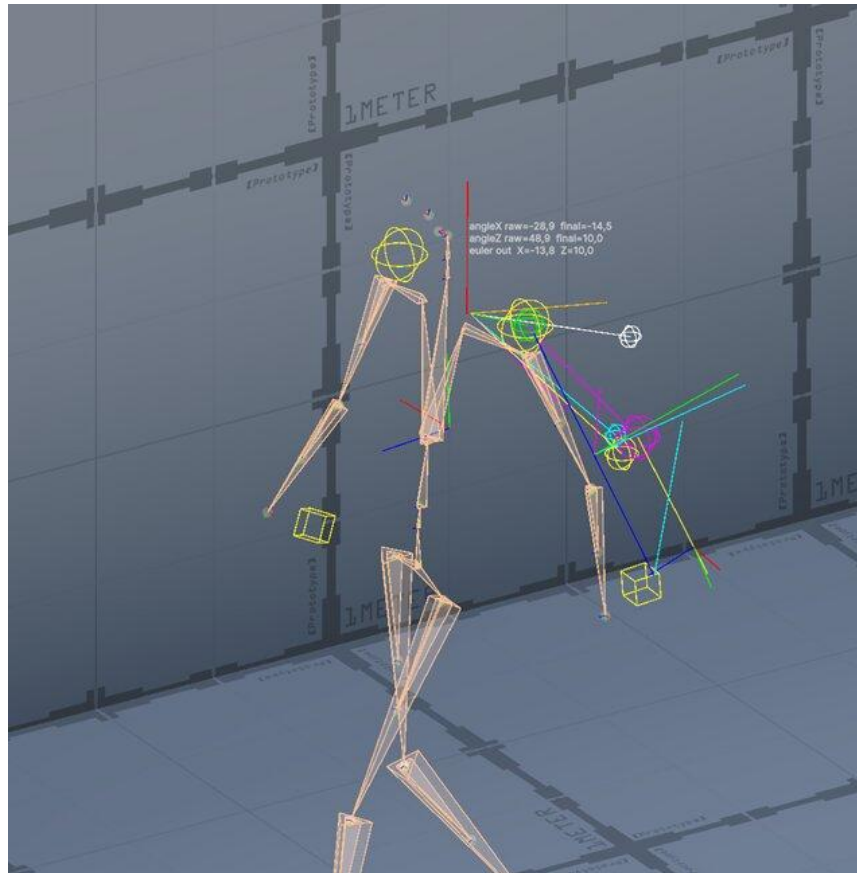


Рисунок 3.15 – Розробницька відмальовка контролера рук в рушії

3.4.5 Контролер ніг (NPARLegsController)

Контролер ніг є найбільшим і найскладнішим компонентом системи. Він відповідає за постановку стоп на поверхню, виконання кроків, виявлення виступів, присідання та низку коригувальних ефектів.

Кожна нога описується екземпляром класу `LegData`, що зберігає весь поточний стан ноги. З ІК-компонентів: `TwoBoneIKConstraint` для основного ІК ланцюга, `MultiRotationConstraint` для пальців стопи, `DistanceConstraintNonAnim` для обмеження довжини ноги. З трансформів: ІК-ціль `ikTarget` у точці балансу стопи, точки п'яти та носка. Стан кроку зберігається у полях `isStepping`, `startPosition`, `endPosition`, `stepLinearAlpha`,

swingSpline та інших. Стан землі – у `isGrounded`, `isAirborne`, `groundNormal`, `groundTransform`. Поточна ціль ІК поділяється на два рівні: `rawTargetPosition` – обчислена логікою контролера позиція без корекцій, та `finalIKPosition` – результат після додавання `positionOffset`, який накопичує всі пост-ефекти.

Кожний фізичний кадр обробляється у п'ять послідовних етапів. Перший – підготовчий: кешується стан реґдола, перевіряється контакт стоп із землею, оновлюється допустима відстань холостого положення та поточна дистанція рейкасту, обчислюється цикл. Другий – обчислення цільових станів: висоти ніг і ступінь присідання, базові та холості початки рейкастів, холості трансформи стоп. Третій – логіка кроків: рішення про крок, прогрес кроку та його завершення. Четвертий – пост-ефекти: довжини ніг, протидія швидкості, притиснення стоп, ефект приземлення, перестрілення обертання, обмеження висоти стоп, обертання за відстанню. П'ятий – застосування ІК. Діаграму цього життєвого циклу можна побачити нижче (рис. 3.16).

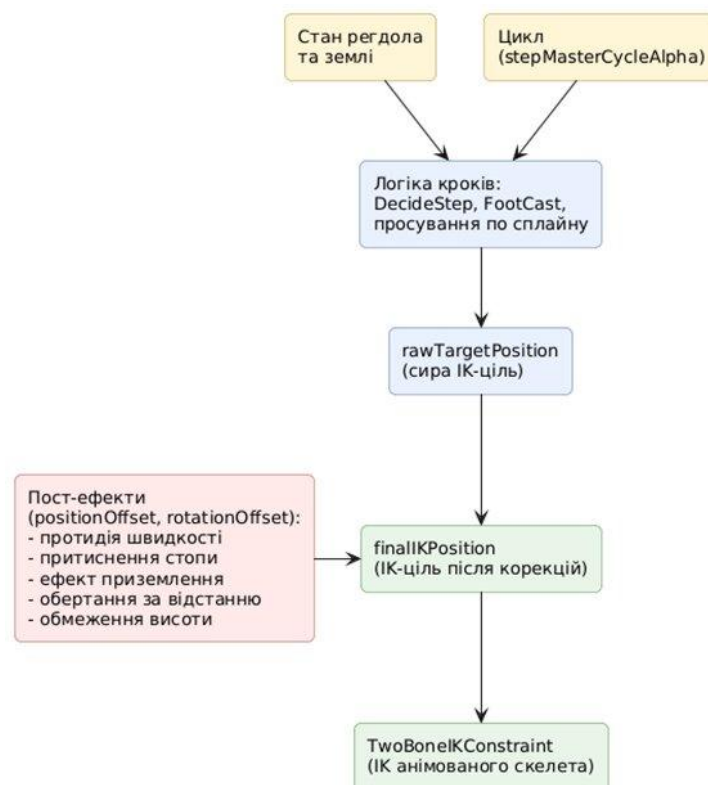


Рисунок 3.16 – Архітектура контролера ніг: сиря ІК-ціль і пост-ефекти

Для кожної ноги щокадру обчислюється базовий початок рейкасту `baseRaycastOrigin` – горизонтальна позиція, куди нога повинна стояти в поточному стані. Вона визначається як зміщена від проєкції центру мас (з можливим змішуванням у бік центру стоп) позиція, повернута в напрямку `PseudoForward` з поступовим підворотом до `TravelForward` на кут до `BaseOriginMaxTravelForwardAngle`. До базового початку додається зміщення поточної пози ноги. Холостий початок `idleRaycastOrigin` переміщується плавніше: він слідує за базовим через `MoveTowards`, а між кадрами зміщується пропорційно до горизонтальної швидкості центру мас. При обох ногах у повітрі він одразу відповідає базовому.

Метод `FootCast` визначає де нога може стати, виконуючи перевірку поверхні у кілька фаз. Перша фаза – `VoxCast` розміром колайдера стопи вниз для отримання нормалі поверхні та базової висоти. Друга – серія індивідуальних рейкастів із набору точок навколо периметра стопи, орієнтованих за нормаллю з першої фази. Третя фаза аналізує результати: обчислюється середня позиція та нормаль успішних влучань, а також найвища та найнижча висоти для виявлення виступу. Якщо різниця висот перевищує `ledgeDetectionThreshold` або деякі рейкасти не влучили, виступ вважається виявленим. У четвертій фазі система приймає рішення: якщо крок нагору або вниз є реалізовним, обирається найкращий кластер влучань за кривою висотних переваг `heightPreferenceCurve` і виконується рекурсивний `FootCast` зі скоригованим початком; інакше нога ставиться перед виступом. У п'ятій фазі виконується спрощена корекція виступу без рекурсії: фінальна точка зміщується за вектором краю. Шоста фаза конструює структуру `FootCastHit` з результуючою позицією, нормаллю та інформацією про виступ. Діаграму алгоритму `FootCast` можна побачити на рисунку нижче (рис. 3.17), а код функції у Додатку А.21.

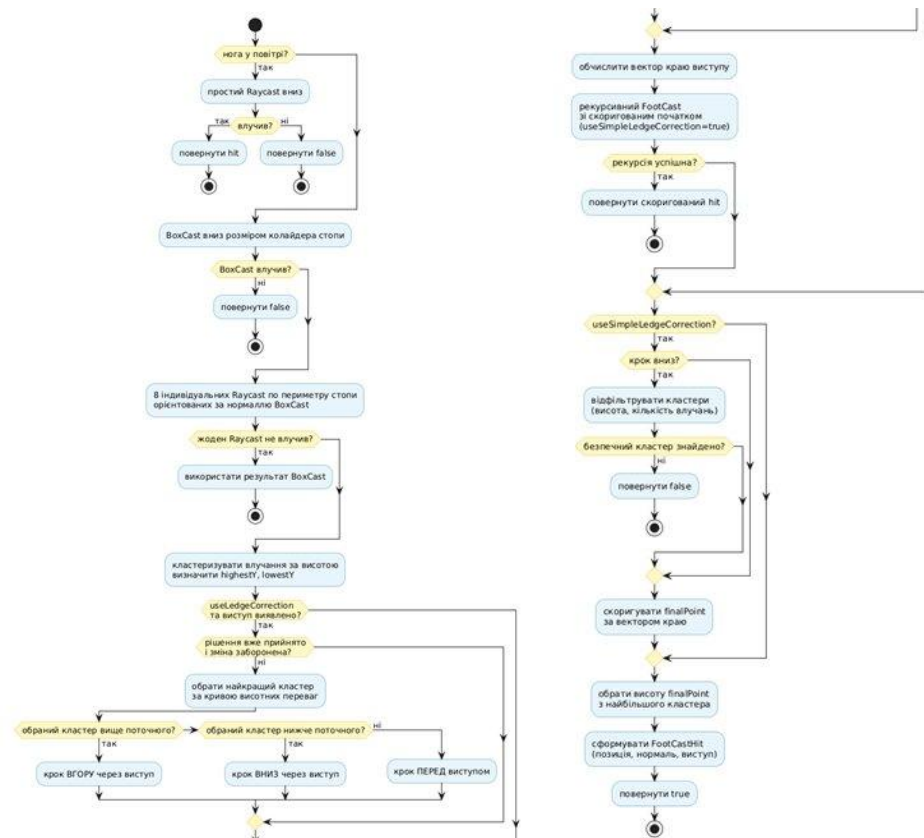


Рисунок 3.17 – Діаграма алгоритму FootCast

Система змішує два режими вирішення кроків через параметр `cycleBlendAlpha`, що плавно зростає від 0 до 1 зі збільшенням швидкості. При низьких значеннях домінує реактивний режим, при високих – циклічний. Майстер-цикл `stepMasterCycleAlpha` просувається кожний кадр пропорційно до $\frac{1}{\text{cycleLengthInSeconds}}$, де $\text{cycleLengthInSeconds} = \text{swingTime} + \text{strideLength} / \text{comSpeed}$. Кожна нога отримує власну фазу зі зсувом у половину циклу. При переході між режимами майстер-цикл поступово синхронізується з поточними реактивними кроками, щоб уникнути різких подвоєних кроків в момент підвищення швидкості. Схему гібридної моделі прийняття рішення представлено на рис. 3.18.

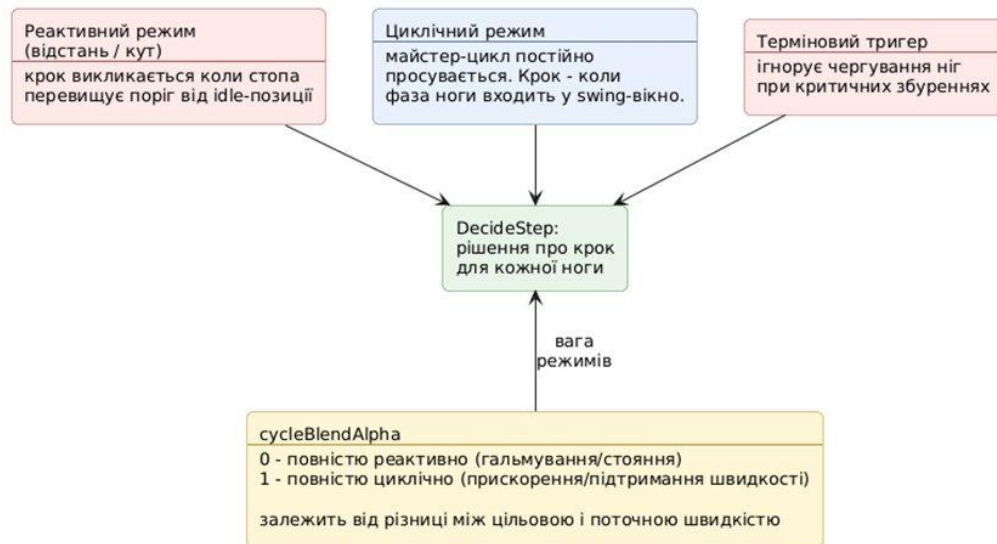


Рисунок 3.18 – Гібридна модель прийняття рішення

Для кожної ноги щокадру метод `DecideStep` приймає рішення про необхідність кроку. Циклічний тригер активується коли фаза ноги потрапляє у вікно $\text{swingTimeAsAlpha} * \text{cycleBlendAlpha}$, тобто чим менше `cycleBlendAlpha`, тим вужче вікно і тим рідше цикл самостійно ініціює кроки. Реактивний тригер перевіряє горизонтальну та вертикальну відстань стопи від холостого положення, а також кутове відхилення. Терміновий тригер ігнорує обмеження на чергування ніг, якщо відхилення перевищило критичний множник `UrgencyThresholdMultiplier` або обидва рейкасти провалились. Чергування регулюється через `AllowedForStep`: нова нога може зробити крок тільки якщо минув `StepSelfDelay` з моменту її останнього кроку, минув `OtherLegDelay` з моменту кроку іншої ноги, та ця нога відстоїть від холостого положення більше або рівно як інша.

Метод `PredictStepBlendedTarget` обчислює позицію, куди нога має стати, змішуючи реактивний та циклічний початки рейкасту. Реактивний початок: базова позиція + горизонтальна швидкість, масштабована `StepPredictionSpeedFactor` та зменшена при боковому кроці всередину. Циклічний початок: базова позиція + вектор кроку ($\text{strideLength} * \text{forwardStrideAlpha} * \text{comVelocity}$) + поправка на залишковий час до кінця фази. Обидва змішуються пропорційно до `cycleBlendAlpha`. Окремо до

результату додається бічне зміщення при повороті: зовнішня нога відсувається назовні пропорційно до бічної компоненти PID-сили та швидкості, внутрішня – злегка стягується. Попередній початок рейкасту зберігається між кадрами і плавно наближається до нового через MoveTowards, щоб уникнути різких стрибків цілі під час кроку.

Після визначення початкової та кінцевої позицій будується В-сплайн через CalculateFootSpline. Контрольні точки задаються у пресеті відносно напрямку кроку та масштабуються за швидкістю. Додатково застосовується уникання іншої стопи: якщо траєкторія проходить надто близько до неї, контрольні точки зміщуються назовні. Уникання перешкод піднімає контрольні точки якщо капсульна перевірка між початком і кінцем кроку виявляє геометрію на шляху. Позиція стопи під час кроку обчислюється як swingSpline.EvaluateUniform(stepLinearAlpha) – рівномірна параметрична оцінка по дузі сплайну, що забезпечує стабільну швидкість руху стопи незалежно від форми кривої і відстані між контрольними точками.

Після виконання кроків до positionOffset та rotationOffset кожної ноги адитивно застосовується ряд корекцій. Протидія швидкості зміщує стопи у напрямку, протилежному нормалізованій PID-силі, компенсуючи тенденцію тіла вперед. Притиснення стоп (ApplyFeetPushdown) опускає ІК-ціль на PushdownDistance вниз із наростанням через pushdownProgressAlpha, тримати персонажа в стоячому положенні. При увімкненій компенсації кута коліна сила притиснення модулюється через $\sin(kneeAngle/2)^{power}$: максимальна при куті близькому до 90°, що поступово зменшується при повному випрямленні. Обмеження висоти стоп (ApplyFeetHeightLimits) обчислює мінімально допустиму висоту ІК-цілі відносно стегна та забороняє стопі підніматися вище або нижче. При повороті внутрішня нога отримує знижений ліміт висоти через TurnTiltHeightAlphaReduction, що дозволяє тілу нахилитися у повороті. Обертання за відстанню (ApplyDistanceRotation) нахиляє стопу навколо п'яти або носка залежно від того, чи стегно

знаходиться спереду чи позаду стопи – імітуючи природний переكات стопи при ходьбі.

Всю систему в роботі можна побачити на рисунках нижче (рис. 3.19-3.20).

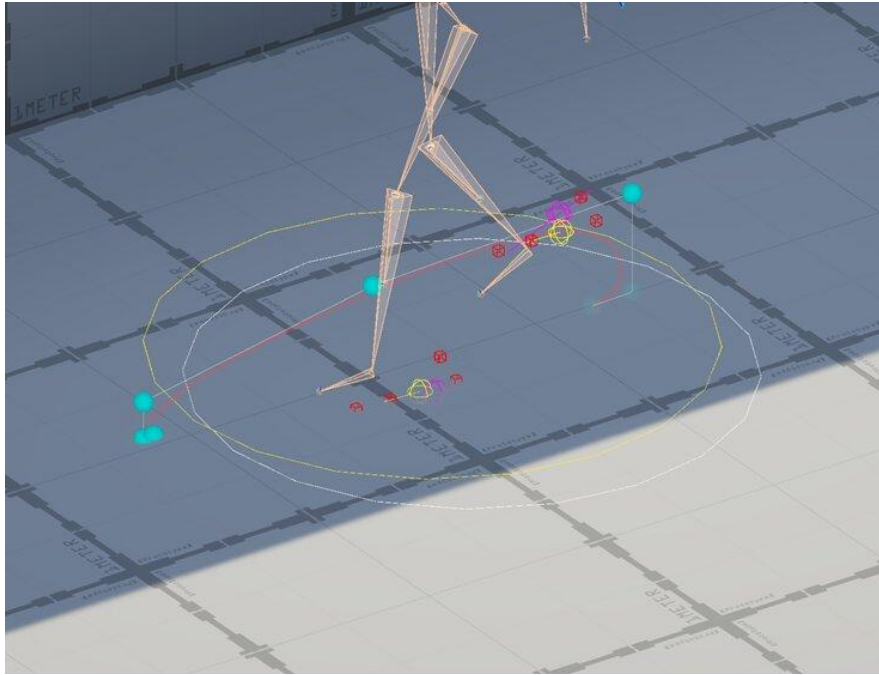


Рисунок 3.19 – Розробницька відмальовка контролеру ніг на рівній поверхні

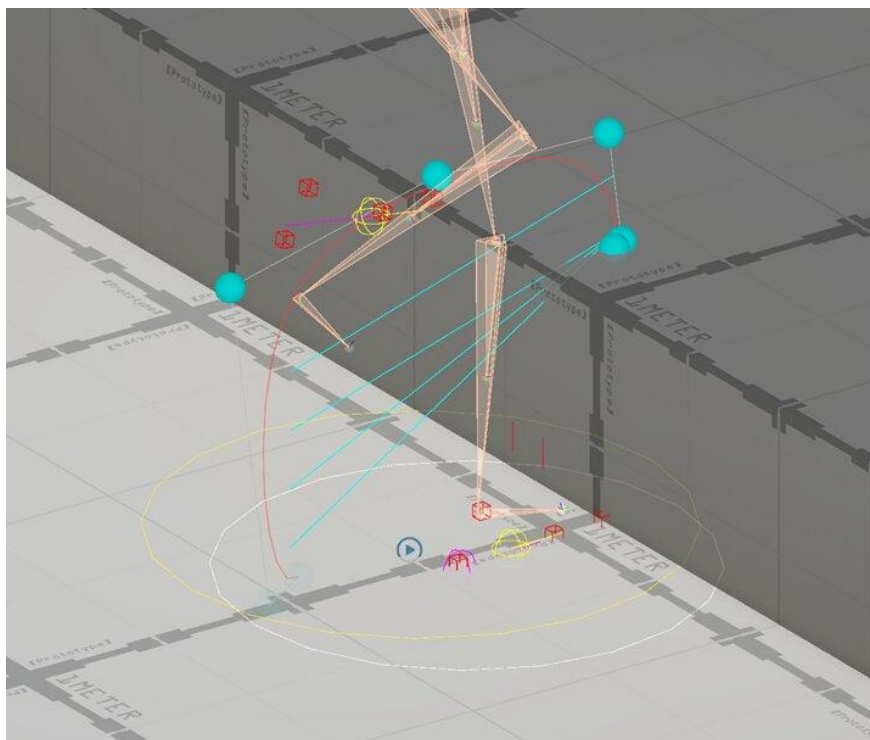


Рисунок 3.20 – Розробницька відмальовка контролеру ніг перед перешкодою

3.5 Контролер навігації

Контролер навігації відповідає за прокладання маршруту по NavMesh, планування швидкості вздовж маршруту та обчислення бажаного напрямку погляду. Він виводить три значення, що читаються головним контролером та іншими підсистемами: MovementDirection, Speed та DesiredFacingRotation. Діаграму алгоритму планування швидкості можна побачити нижче (рис. 3.21).



Рисунок 3.21 – Алгоритм планування швидкості навігаційним контролером

Поведінка контролера визначається структурою `NavigationIntent`, що встановлюється зовнішнім власником – зазвичай поточним станом `NPARBrain` – через атомарний виклик `SetIntent`. Намір містить режим обертання (`Uncontrolled`, `KeepCurrentRotation`, `TowardsWaypoint`, `TowardsCustomPoint`), прапорець збереження обертання на місці, точку для режиму `TowardsCustomPoint`, прапорець обмеження швидкості за кутом між поглядом та рухом, дозвіл на рух та бажана швидкість. Такий підхід забезпечує повне відокремлення логіки навігації від логіки станів: контролер не знає нічого про стани, а стани не змінюють внутрішніх параметрів контролера напрямку.

Стандартний `NavMeshAgent` використовується лише для побудови маршруту – його кінематичне керування позицією та обертанням вимкнено. Щокадру агент вручну переміщується до проєкції центру мас персонажа. Маршрут будується до заданої цілі або трансформу, після чого контролер самостійно читає масив кутів маршруту та виконує всі подальші обчислення.

Оскільки фізично керований персонаж не може зупинитися миттєво, контролер завчасно обчислює необхідне гальмування перед кутами маршруту. Метод `RebuildBrakingClusters` щокадру виконує три кроки. Спочатку для кожного кута маршруту обчислюється кут повороту між вхідним і вихідним сегментами; для кінцевої точки кут завжди дорівнює 180° , що відповідає мінімальній швидкості. Потім сусідні кути у межах `clusterRadius` об'єднуються у кластери з накопиченням кутів. Кожному кластеру через `angleToSpeedCurve` призначається цільова швидкість. Нарешті виконується зворотна індукція від останнього кластера до першого: якщо досягти цільової швидкості наступного кластера неможливо фізично з урахуванням `brakingDeceleration` та відстані між кластерами, цільова швидкість поточного кластера знижується.

Під час руху відстежується пікова швидкість `peakSpeed` у вільній зоні перед активним кластером. У зоні гальмування швидкість плавно зменшується від `peakSpeed` до цільової швидкості кластера, причому рампа

займає лише `brakingRampPortion` від повної гальмівної відстані — решта тримає цільову швидкість. Окрім кластерного планування, на швидкість впливають: кут між `TravelForward` та `PseudoForward` через `facingAngleToSpeedCurve`, ступінь присідання та дані сканування рельєфу.

Щокадру вздовж шляху попереду виконується сканування рельєфу — кілька рейкастів вниз у рандомних точках смуги шириною `terrainScanStripWidth`. Вибірки накопичуються у ковзному буфері з терміном придатності `terrainSampleLifetime`. До буфера щокадру підганяється лінійна регресія по осі руху, що одночасно дає два показники: нахил рельєфу як кутовий коефіцієнт регресії та шорсткість як розмах залишків навколо лінії тренду. Таке розділення дозволяє рівному пандусу мати низьку шорсткість, а нерівній горизонтальній поверхні — низький нахил, незалежно один від одного.

`DesiredFacingRotation` обчислюється щокадру залежно від режиму наміру. У режимі `TowardsWaypoint` система дивиться вперед уздовж маршруту на відстань `facingLookaheadDistance`, що дозволяє персонажу починати повертатися до повороту завчасно. Аналогічне попередження на відстань `travelLookaheadDistance` застосовується для `TravelLookaheadDirection`, що використовується головним контролером для обчислення `TravelForward`. Якщо персонаж стоїть і намір не вимагає збереження обертання, обидва вектори відповідають `PseudoForward`.

Приклади розрахунку гальмувальної відстані (рис. 3.22) і сканування рельєфу (рис. 3.23) можна побачити нижче.

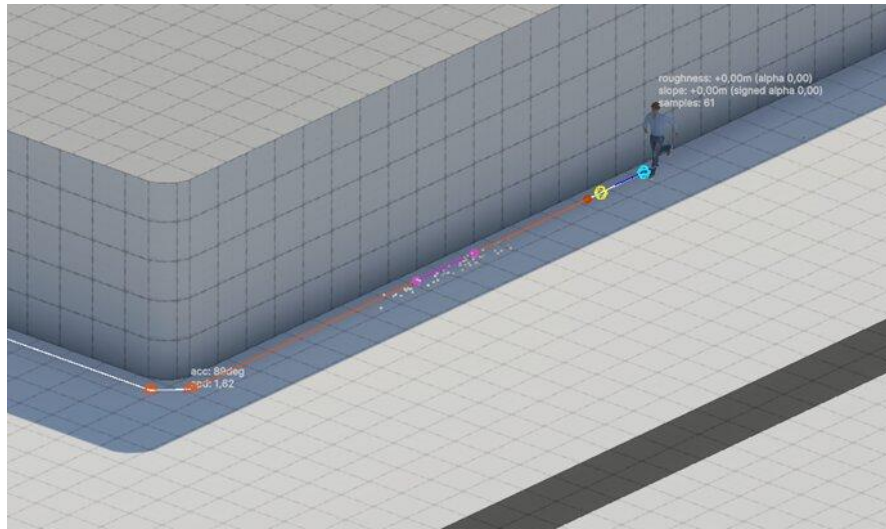


Рисунок 3.22 – Приклад розрахунку гальмівної відстані до кута



Рисунок 3.23 – Приклад сканування шорсткого рельєфу

3.6 Висновки до розділу 3

У третьому розділі описано програмну реалізацію системи керування активним регдолом NPAR.

Розглянуто загальну архітектуру системи, що ґрунтується на принципі подвійного скелета: анімована модель слугує динамічним джерелом цільових поз, а фізична модель реалізує їх в умовах реального фізичного оточення через суглобові приводи. Зв'язок між кістками обох скелетів описується структурою `RagdollBodyBoneData`, а синхронізація відбувається у два етапи – передача цільових орієнтацій у `FixedUpdate` та переміщення анімованого кореня у `LateUpdate`.

Описано систему пресетів, що дозволяє змінювати поведінку персонажа шляхом підміни екземплярів класів даних без зміни коду. Поведінковий пресет `NPARBehaviorPresetSO` об'єднує налаштування всіх контролерів і застосовується через `NPARPresetManager` з опціональною інтерполяцією параметрів у часі.

Описано машину станів та логіку прийняття рішень у `NPARBrain`. Оцінка стабільності ґрунтується на п'яти метриках з ковзними часовими вікнами та перевіркою як абсолютних значень, так і їхніх дельт. `FallState` реалізує повний цикл падіння через чотири фази з вибором анімаційного варіанту на основі поточного стану персонажа.

Описано сім контролерів фізичної симуляції. `NPARMainController` обчислює спільний стан системи, керує трьома PID-регуляторами та кількома режимами позиціонування й обертання анімованого скелета. `NPARMuscleController` глобально масштабує жорсткість суглобів через єдиний коефіцієнт зусилля. `NPARBodyController` розподіляє нахил тіла між трьома сегментами хребта. `NPARArmsController` керує руками через двокісткову ІК з балансовими зміщеннями та двопрохідним розрахунком плечового суглоба. `NPARLegsController` реалізує гібридну систему постановки стоп зі змішуванням реактивного та циклічного режимів,

багатофазним алгоритмом FootCast з виявленням виступів та набором пост-ефектів. NPARNavigationController забезпечує кластерне планування швидкості з зворотною індукцією та скануванням рельєфу.

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

4.1 Середовище тестування

Тестування системи NPAR виконувалось у спеціально побудованій сцені рушія Unity. Апаратне забезпечення: процесор AMD Ryzen 7 7800X3D, 32 ГБ оперативної пам'яті, операційна системи Microsoft Windows 11. Середня частота кадрів у тестовій сцені складала 120 кадрів на секунду. Частота оновлень фізичної симуляції становила 120 оновлень на секунду.

Сцена поділена на сім незалежних зон, кожна з яких спрямована на перевірку окремого аспекту поведінки системи. Перелік зон та їх призначення наведено у таблиці 4.1.

Таблиця 4.1 – Тестові зони та їх призначення

№	Назва зони	Призначення
1	Зона ходьби	Перевірка стабільності циклу ходи
2	Зона бігу	Перевірка циклу бігу та планування швидкості на поворотах
3	Зона нерівного рельєфу	Перевірка адаптації до різноманітних перешкод
4	Зона балансування	Перевірка стійкості під ударами на рівній поверхні
5	Зона балансування на одній нозі	Перевірка стійкості на одній нозі під ударами
6	Зона балансування на нерівній поверхні	Перевірка стійкості та відновлення після падіння на складному рельєфі
7	Зона падіння з висоти	Перевірка поведінки в польоті та вибору варіанту падіння

4.2 Результати функціонального тестування

4.2.1 Зона ходьби

У першій зоні NPAR виконує замкнений маршрут по колу з цільовою швидкістю у 1,5 м/с. Мета – перевірити стабільність циклу ходи в умовах тривалого рівномірного руху.

Спостерігається стабільний і ритмічний цикл ходи без відхилень від маршруту (рис. 4.1). Кожен шаг має проходити рівномірну відстань і має однаковий час, що складається в стабільний цикл. Верхня частина тіла природно реагує на рух нижньої через компенсацію поворотного моменту тілом і руками. Система майже не демонструє нестабільності впродовж тестування, окрім рідких випадків, коли персонаж спотикається на різких поворотах.

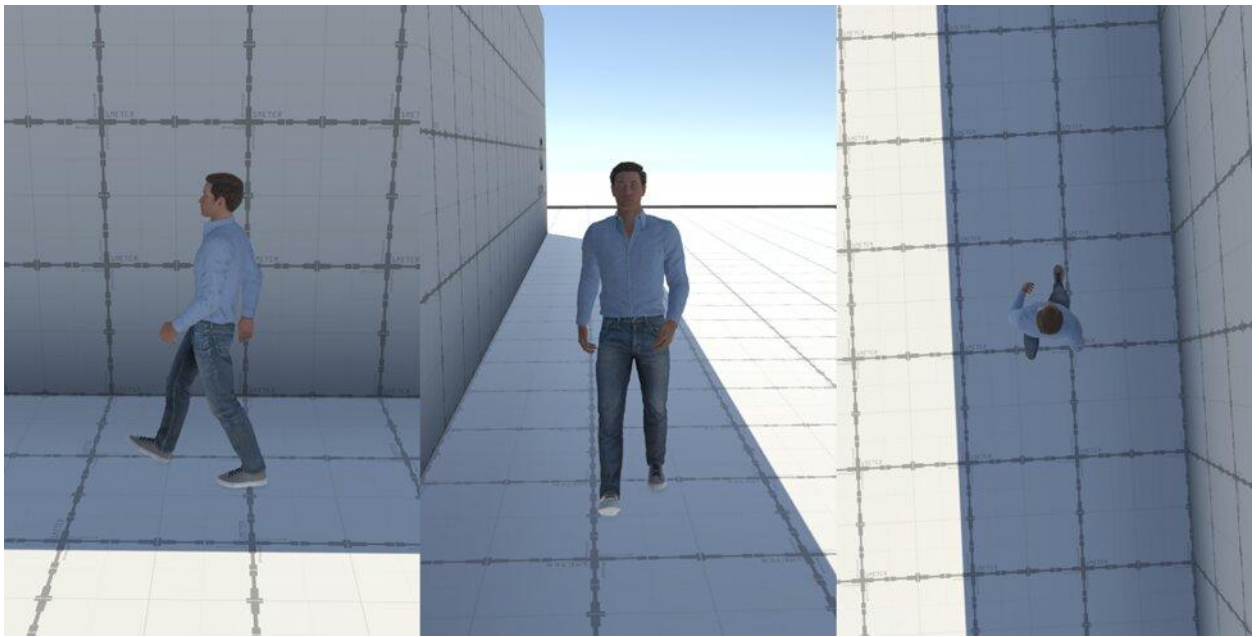


Рисунок 4.1 – NPAR під час ходьби

4.2.2 Зона бігу

У другій зоні NPAR виконує замкнений маршрут з максимальною швидкістю. Маршрут містить два повороти з кутом приблизно 90 градусів,

один 30-45 градусів і один приблизно 150 градусів, що дозволяє спостерігати планування швидкості при різних умовах.

При русі на прямолінійних ділянках NPAR досягає максимальної швидкості 5,2 м/с (рис. 4.2). На крутих поворотах NPAR починає сповільнюватися завчасно (рис. 4.3), проходячи їх в темпі звичайної ходи. На пологому повороті NPAR визначає його як недостатній для суттєвого гальмування, тож NPAR проходить поворот на близькій до максимальної швидкості, нахилиючи корпус у бік повороту (рис. 4.4). Загалом система демонструє плавний і стабільний цикл бігу, тільки іноді спотикаючись на крутих поворотах.

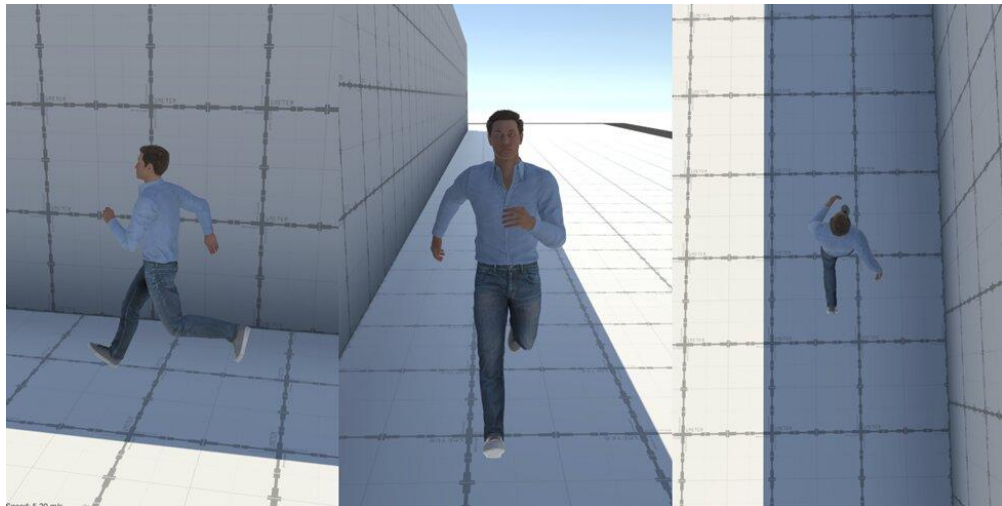


Рисунок 4.2 – NPAR під час бігу на максимальній швидкості

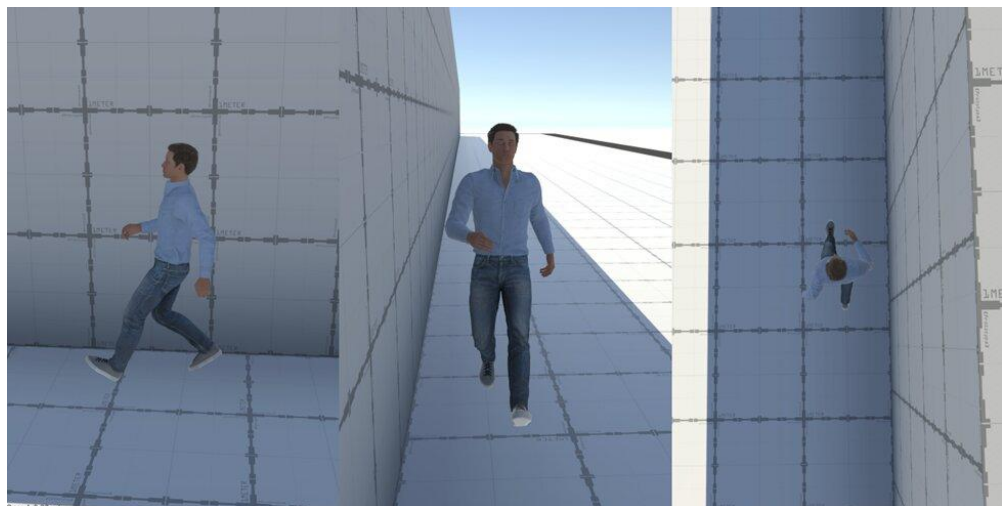


Рисунок 4.3 – NPAR під час гальмування перед крутим поворотом

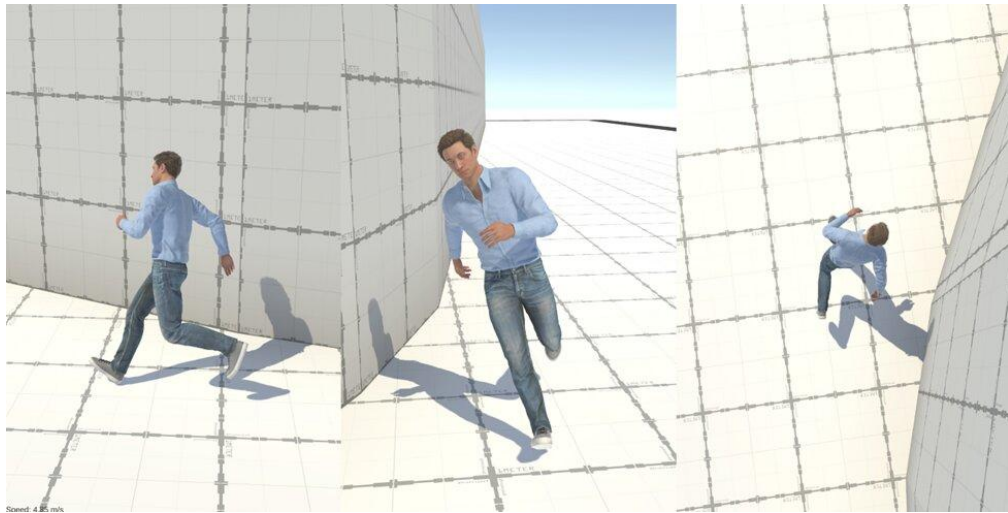


Рисунок 4.4 – NPAR на швидкісному повороті під час бігу

4.2.3 Зона нерівного рельєфу

У третій зоні NPAR проходить лінійний маршрут через різноманітні перешкоди: об'єкти на землі, рампи, сходи, горбиста поверхня.

Контролер ніг коректно виявляє геометрію поверхні та адаптує постановку стоп відповідно до рельєфу. Перед кожною перешкодою NPAR сповільнюється в залежності від її складності. На рампах і нерівній поверхні стопи підлаштовуються за нормаллю. На сходах алгоритм FootCast розумно виявляє виступи та обирає коректну точку постановки стопи. На об'єктах, що лежать на землі, система або переступає через них, або обходить залежно від висоти. Загальна стійкість зберігається впродовж усього маршруту, проте NPAR іноді має складнощі з особо високими перешкодами, крутими нахилами на горбистій поверхні і на сходах. Також, в деяких випадках фізична нога не може досягти обраної цільової позиції з гарною точністю. Це обумовлено використанням простих пружин для м'язових моментів замість повноцінного PID-регулятора. Не зважаючи на проблеми, після кількох спроб NPAR все одно проходить перешкоду успішно, що є очікуваною поведінкою на особливо складних перешкодах. Приклади перешкод в зоні можна побачити на рис. 4.5, рис. 4.6, рис. 4.7 та рис. 4.8.

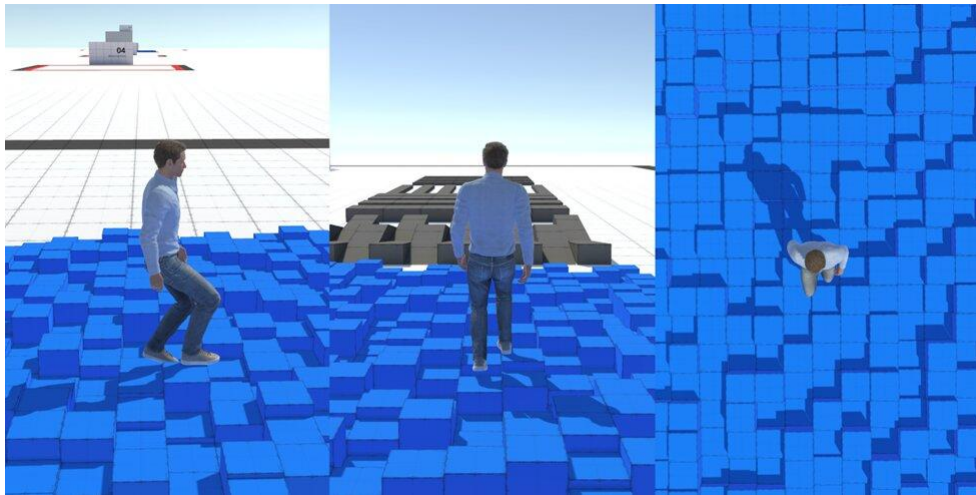


Рисунок 4.5 – NPAR на нерівній поверхні

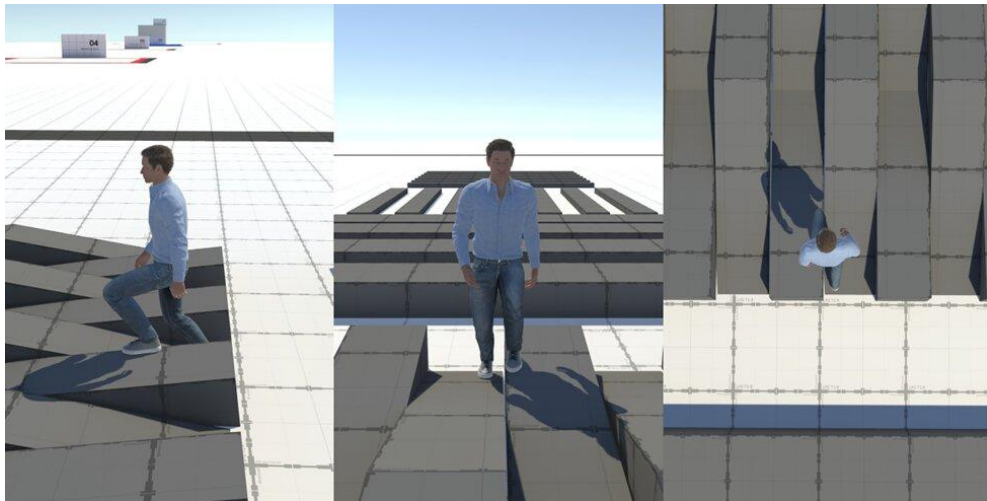


Рисунок 4.6 – NPAR на високих перешкодах

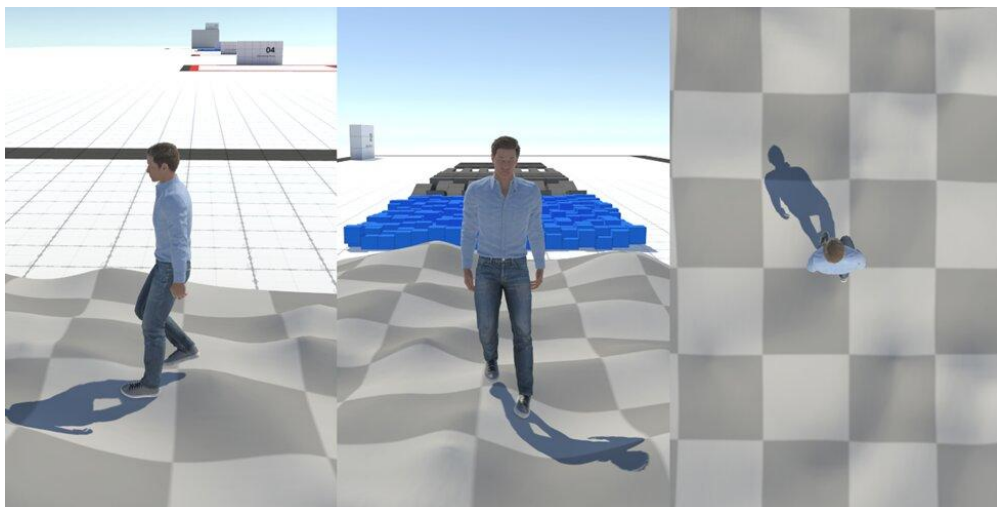


Рисунок 4.7 – NPAR на горбистій поверхні

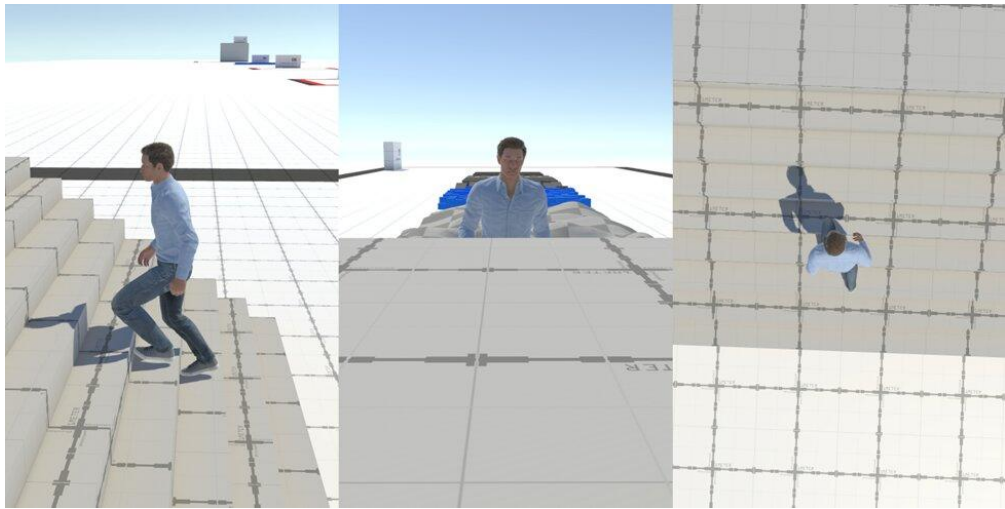


Рисунок 4.8 – NPAR на сходах

4.2.4 Зона балансування

У четвертій зоні у NPAR кидають об'єкти середньої ваги, поки він стоїть на рівній поверхні.

NPAR успішно утримує рівновагу в більшості випадків. Падіння виникало приблизно раз на кожні 7-10 кидків. Завдяки динамічному скануванню рельєфу, NPAR навіть іноді використовує об'єкти, що кидаються в нього, як опору для утримання балансу. Приклад NPAR, що намагається утримати баланс після удару, можна побачити на рисунку нижче (рис. 4.9). Після падіння система коректно переходить у цикл підйому та відновлює стоячу позицію (рис. 4.10).

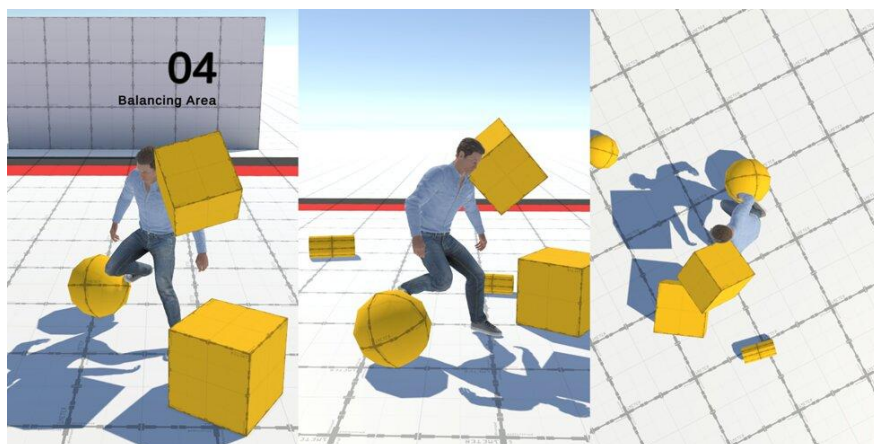


Рисунок 4.9 – NPAR під ударом об'єкта

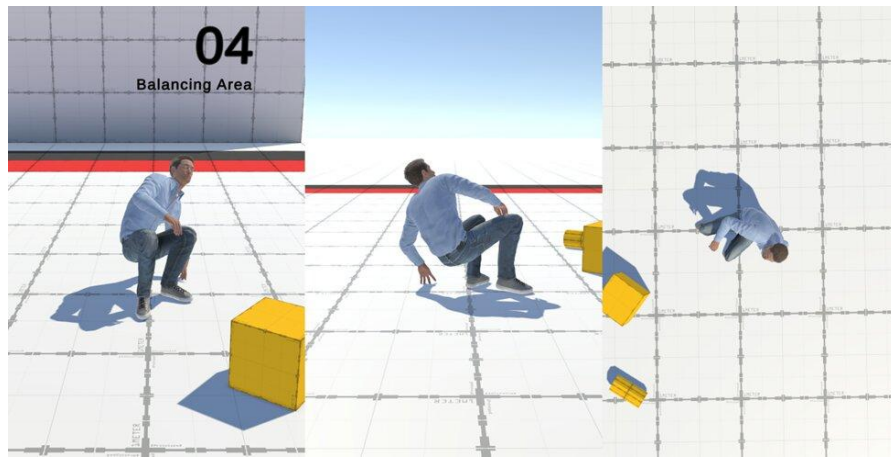


Рисунок 4.10 – NPAR у процесі підйому після падіння

4.2.5 Зона балансування на одній нозі

П'ята зона аналогічна четвертій, але NPAR стоїть на одній нозі. Маса об'єктів, що кидаються, менша з огляду на знижену стійкість конфігурації.

Попри значно складніші умови рівноваги, NPAR демонструє хорошу стійкість, порівняну з четвертою зоною: падіння виникало приблизно раз на кожні 7-10 кидків. Це свідчить про те, що система балансування ефективно адаптується до опорної площі. Найважливішу роль в балансуванні на одній нозі мають контролери тіла і рук, які повинні постійно компенсувати прискорення NPAR. Приклад їх роботи представлено на рис. 4.11. Проте все ж таки в цьому сценарії іноді виникають ситуації, де NPAR утримує баланс ненатурально.

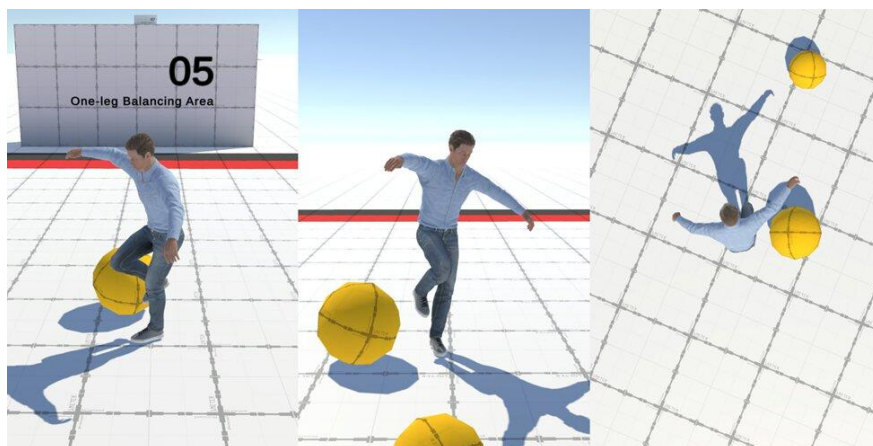


Рисунок 4.11 – NPAR на одній нозі одразу після удару

4.2.6 Зона балансування на нерівній поверхні

Шоста зона поєднує складний нерівний рельєф та об'єкти більшої ваги. Це найбільш навантажений тест на стійкість.

В цих умовах NPAR утримував рівновагу рідше: падіння виникало приблизно раз на кожні 4-7 кидків. NPAR все одно успішно вибирав стабільні точки постановки стоп, але через хаотичні умови, все ж таки спотикався (рис. 4.12). Водночас система стабільно виконувала підйом після падінь навіть на складному рельєфі, коректно вибираючи варіант анімації та завершуючи цикл відновлення до стоячої позиції. NPAR при підйомі на ноги можна побачити на рис. 4.13 та рис. 4.14.

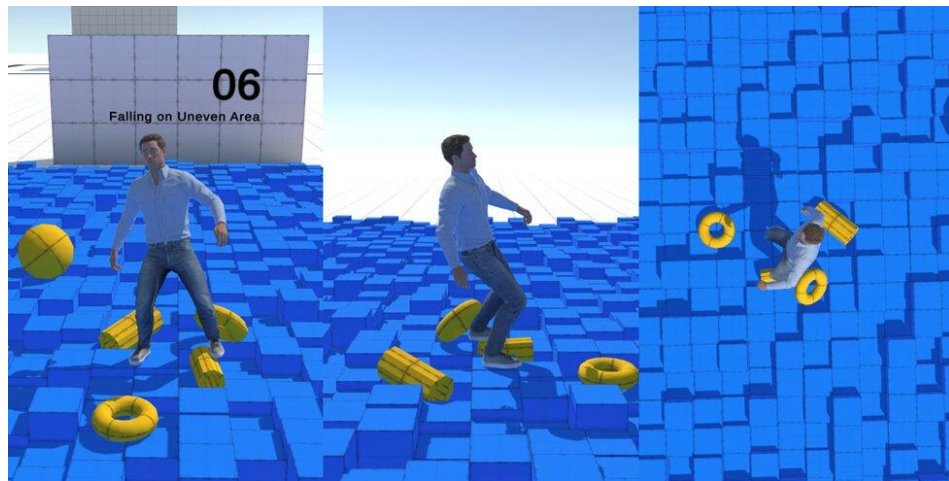


Рисунок 4.12 – NPAR після удару на нерівній поверхні

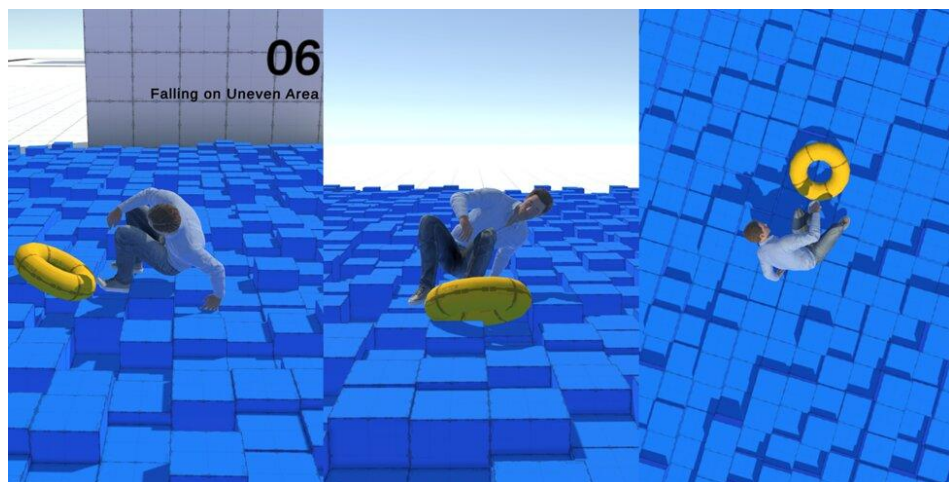


Рисунок 4.13 – NPAR у процесі вставання на нерівній поверхні (на спині)

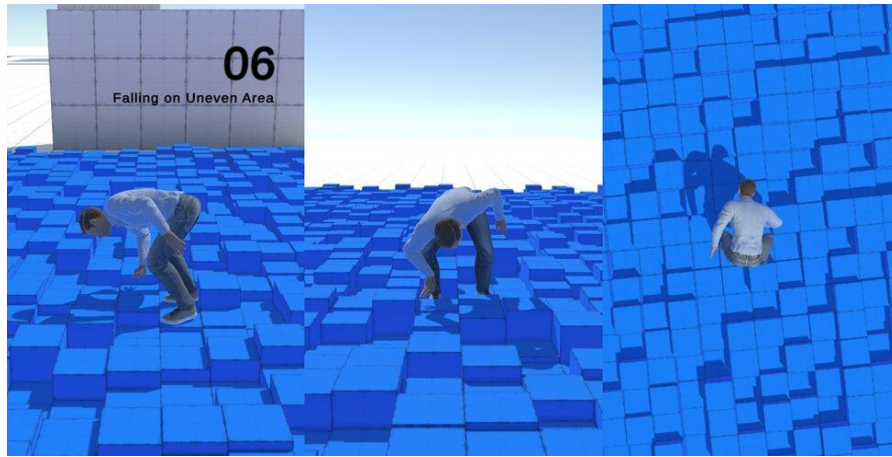


Рисунок 4.14 – NPAR у процесі вставання на нерівній поверхні (на животі)

4.2.7 Зона падіння з висоти

У сьомій зоні NPAR стоїть на краю підвищеної поверхні. Об'єкт, що штовхає, застосовує зусилля під різним вертикальним кутом і швидкістю, а початкова орієнтація NPAR є випадковою. Це забезпечує різноманіття початкових умов падіння.

Залежно від напрямку та швидкості удару система обирає відповідний варіант анімації падіння: на живіт, на спину або інші – відповідно до локального вектора швидкості центру мас відносно PseudoForward, як описано у підрозділі 3.3.1. Після приземлення система коректно визначає положення тіла та виконує відповідний цикл підйому. Приклади польоту після поштовху в спину (рис. 4.15), в живіт (рис. 4.16) і в бік (рис. 4.17) можна побачити нижче.

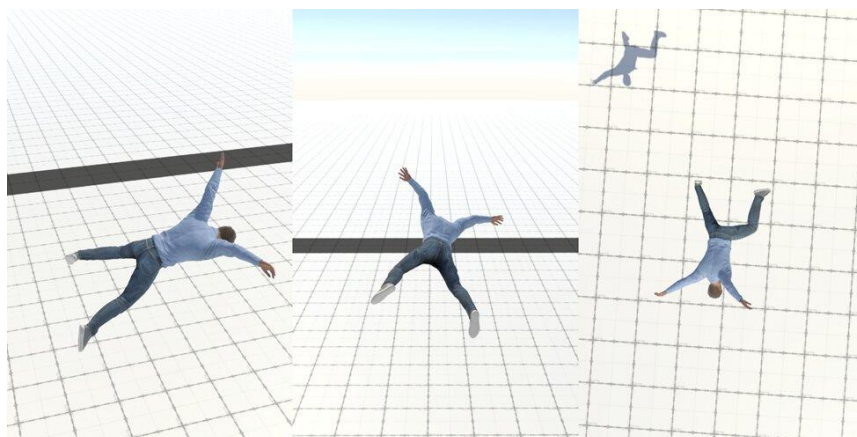


Рисунок 4.15 – NPAR в польоті після поштовху в спину

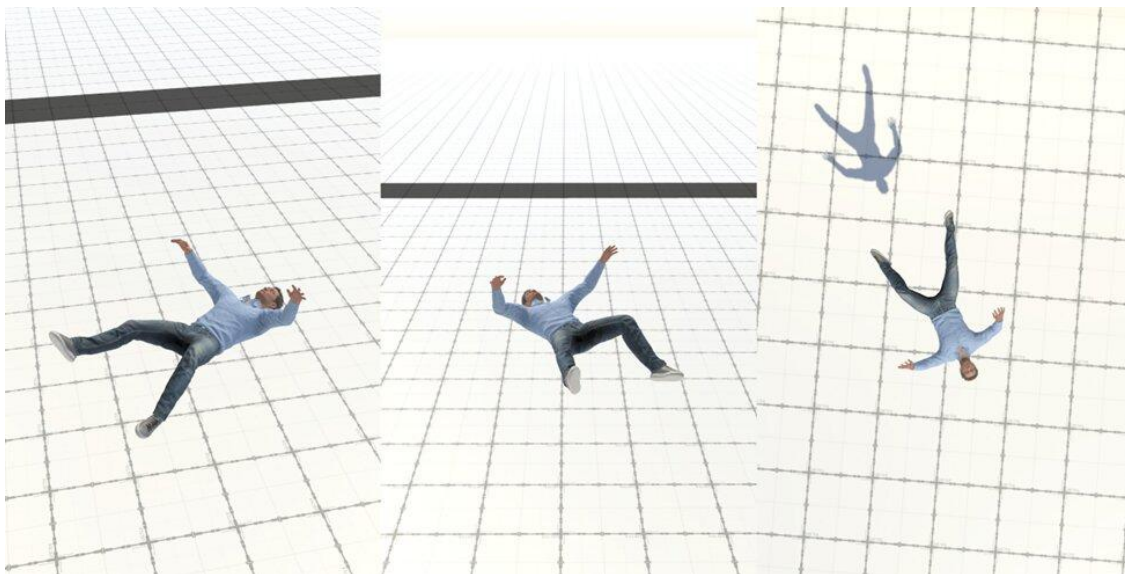


Рисунок 4.16 – NPAR в польоті після поштовху в живіт

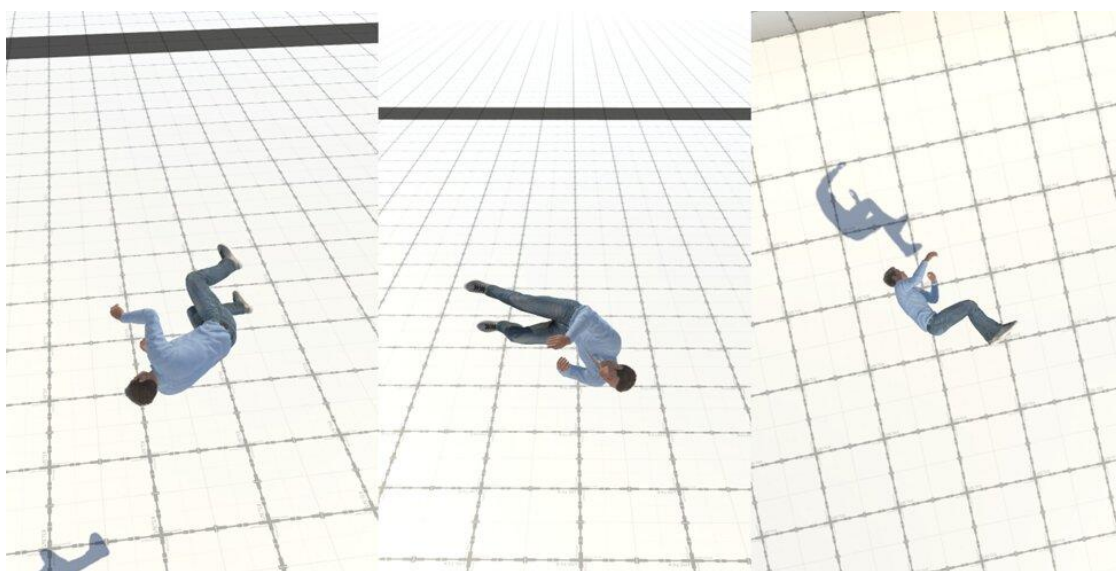


Рисунок 4.17 – NPAR в польоті після поштовху в бік

4.3 Узагальнені результати

За результатами тестування у всіх семи зонах система NPAR продемонструвала здебільшого коректну поведінку відповідно до поставлених задач. Узагальнені результати наведено у таблиці 4.2.

Таблиця 4.2 – Узагальнені результати тестування

Зона	Перевірялось	Результат
Ходьба	Стабільність циклу ходи	Стабільний ритмічний цикл без відхилень. Нестабільність на крутих поворотах.
Біг	Цикл бігу, планування швидкості на поворотах	Максимальна швидкість 5,2 м/с; пологий поворот проходиться без суттєвого гальмування з нахилом корпусу. Нестабільність на крутих поворотах.
Нерівний рельєф	Адаптація до перешкод	Коректна постановка стоп на всіх типах перешкод. Іноді неточне повторення цільових рухів фізичним тілом, нестабільність на схилах.
Балансування	Стійкість під ударами	Падіння раз на 7-10 кидків; підйом коректний. Іноді занадто пружний корпус.
Балансування на одній нозі	Стійкість на одній нозі під ударами	Падіння раз на 7-10 кидків; підйом коректний. Іноді ненатуральне підтримання балансу.
Балансування на нерівній поверхні	Стійкість та відновлення на складному рельєфі	Падіння раз на 4-7 кидків; підйом коректний.
Падіння з висоти	Поведінка в польоті, вибір варіанту падіння	Коректний вибір варіанту падіння та наступного підйому.

Отримані результати підтверджують практичну придатність реалізованої системи до виконання задач самобалансування та переміщення у фізичній симуляції в різноманітних ігрових умовах. Хоча й система має ряд недоліків, їх можна виправити додатковим налаштуванням параметрів або додаванням нових функцій, якщо ціль зробити дуже реалістичну поведінку персонажа.

Якщо цільове ігрове середовище є більш аркадним, комічним та незграбним, то поточний стан системи NPAR є більш ніж задовільним. Більше того, система дозволяє точно налаштувати розмір штучних сил, стиль процедурних рухів і набір використовуваних анімацій, а також прибрати та додавати нові компоненти. Це все робить систему дуже гнучкою і універсальною.

ВИСНОВКИ

У роботі проведено дослідження та виконано програмну реалізацію системи керування активним регдолом на основі PID-регуляторів, пружинних сил і м'язових моментів для забезпечення самобалансування та переміщення персонажа в ігровому середовищі.

За результатами аналізу існуючих підходів до анімації персонажів встановлено, що традиційні методи – ключова анімація та захоплення руху – не здатні динамічно адаптуватися до фізичного оточення, тоді як пасивний регдол позбавлений можливості активного керування тілом. Концепція активного регдолу поєднує переваги фізичної симуляції та цілеспрямованого керування через архітектуру подвійного скелета. Огляд існуючих реалізацій показав, що повноцінні системи активного регдолу є або недоступними комерційними рішеннями, або обмежуються частковою фізичною симуляцією, що підтвердило доцільність розробки власної системи.

Розроблено модульну архітектуру системи NPAR, що реалізує принцип подвійного скелета: анімована модель слугує динамічним джерелом цільових поз, а фізична модель відтворює їх в умовах реального фізичного оточення через суглобові приводи `ConfigurableJoint`. Модульність забезпечується організацією підсистем у вигляді незалежних компонентів `MonoBehaviour` з єдиним центром координації `NPARMainController` та системою пресетів, що дозволяє змінювати поведінку персонажа без зміни коду шляхом підміни даних.

Реалізовано алгоритми самобалансування на основі трьох PID-регуляторів, що відповідають за утримання проєкції центру мас над точкою балансу, забезпечення цільової швидкості руху та відстеження орієнтації стегон. Оцінка стабільності ґрунтується на п'яти метриках з ковзними часовими вікнами та перевіркою як абсолютних значень, так і їхніх дельт для виявлення раптових збурень.

Реалізовано процедурні алгоритми керування кінцівками: контролер рук на основі двокісткової ІК із балансовими зміщеннями та контролер ніг із гібридною системою постановки стоп, що поєднує реактивний і циклічний режими. Алгоритм FootCast забезпечує виявлення виступів і адаптивний вибір точки постановки стопи на складному рельєфі. Реалізовано також контролер навігації з кластерним плануванням швидкості на основі зворотної індукції та скануванням рельєфу для фізично коректного сповільнення перед поворотами.

За результатами функціонального тестування у семи незалежних зонах система продемонструвала практичну придатність до виконання поставлених задач. На рівних поверхнях забезпечується стабільний ритмічний цикл ходи та бігу з максимальною швидкістю 5,2 м/с. На нерівному рельєфі контролер ніг коректно адаптує постановку стоп до геометрії поверхні. Система витримує зовнішні удари з частотою падіння близько одного разу на 7-10 кидків і стабільно відновлює стоячу позицію після падіння на будь-якому рельєфі. Виявлені обмеження – нестабільність на крутих поворотах та неточне відтворення окремих рухів фізичним тілом – є усувними засобами додаткового налаштування параметрів або розширення функціональності окремих контролерів.

Практична цінність роботи полягає у розробці гнучкої та розширюваної системи, придатної до застосування у відеоіграх різних жанрів для підвищення імерсивності та реалістичності поведінки персонажів, а також у задачах процедурної анімації та симуляції фізично керованих об'єктів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Ołdakowska K. Character animation for games. Explain Visually. URL: <https://explainvisually.co/en/character-animation-for-games/> (date of access: 07.05.2026).
2. How motion capture is used in video game development. VICON. URL: <https://www.vicon.com/resources/blog/how-motion-capture-continues-to-revolutionize-game-development/> (date of access: 07.05.2026).
3. GDC Festival of Gaming. Animation bootcamp: an indie approach to procedural animation, 2017. YouTube. URL: <https://www.youtube.com/watch?v=LNidsMesxSE> (date of access: 07.05.2026).
4. Zucconi A. An introduction to procedural animations. Alan Zucconi. URL: <https://www.alanzucconi.com/2017/04/17/procedural-animations/> (date of access: 07.05.2026).
5. Contributors to Wikimedia projects. Ragdoll physics - Wikipedia. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Ragdoll_physics (date of access: 07.05.2026).
6. Chandler N. How ragdoll physics works. howstuffworks. URL: <https://electronics.howstuffworks.com/ragdoll-physics.htm> (date of access: 07.05.2026).
7. Abreu S. How to make active ragdolls in unity. Medium. URL: <https://sergioabreu-g.medium.com/how-to-make-active-ragdolls-in-unity-35347dcb952d> (date of access: 07.05.2026).
8. Lang P. PuppetMaster overview. Home - RootMotion. URL: <http://root-motion.com/puppetmasterdox/html/page3.html> (date of access: 07.05.2026).
9. Contributors to Wikimedia projects. Euphoria (software) - Wikipedia. Wikipedia, the free encyclopedia. URL: [https://en.wikipedia.org/wiki/Euphoria_\(software\)](https://en.wikipedia.org/wiki/Euphoria_(software)) (date of access: 07.05.2026).
10. Contributors to Wikimedia projects. NaturalMotion - Wikipedia.

Wikipedia, the free encyclopedia. URL: <https://en.wikipedia.org/wiki/NaturalMotion> (date of access: 07.05.2026).

11. Boyer B. Product: Grand Theft Auto IV using NaturalMotion's Euphoria. Game Developer. URL: <https://www.gamedeveloper.com/game-platforms/product-i-grand-theft-auto-iv-i-using-naturalmotion-s-euphoria> (date of access: 07.05.2026).

12. BackbreakerFootball. NaturalMotion's Euphoria, 2010. YouTube. URL: <https://www.youtube.com/watch?v=HauN98naZ9U> (date of access: 07.05.2026).

13. Physics-Based Animation | Unreal Engine 4.27 Documentation | Epic Developer Community. Epic Games Developer. URL: https://dev.epicgames.com/documentation/unreal-engine/physics-based-animation?application_version=4.27 (date of access: 07.05.2026).

14. Applying a Physical Animation Profile in Unreal Engine | Unreal Engine 5.7 Documentation | Epic Developer Community. Epic Games Developer. URL: <https://dev.epicgames.com/documentation/unreal-engine/applying-a-physical-animation-profile-in-unreal-engine> (date of access: 07.05.2026).

15. codehawk64. Fun with physics animation harassment. Reddit. URL: https://www.reddit.com/r/unrealengine/comments/ufd6b4/fun_with_physics_animation_harassment/ (date of access: 07.05.2026).

16. Contributors to Wikimedia projects. Human: Fall Flat - Wikipedia. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Human:_Fall_Flat (date of access: 07.05.2026).

17. Replicating the look of Human Fall Flat's realtime shading. Unity Discussions. URL: <https://discussions.unity.com/t/replicating-the-look-of-human-fall-flats-realtime-shading/1551102> (date of access: 07.05.2026).

18. Aubrey D. Human Fall Flat Interview With Tomas Sakalauskas, CEO Of No Brakes Games. TheGamer. URL: <https://www.thegamer.com/human-fall-flat-interview-tomas-sakalauskas/> (date of access: 07.05.2026).

19. DemonRebuilt. Awkward Ragdoll Party - Human: Fall Flat

Multiplayer - Best Moments, 2018. YouTube. URL: <https://www.youtube.com/watch?v=PtLSSmTIEYw> (date of access: 07.05.2026).

20. Unity. Unity. URL: <https://unity.com> (date of access: 07.05.2026).

21. Maximize Multiplatform Game Development | Unity. Unity. URL: <https://unity.com/features/multiplatform> (date of access: 07.05.2026).

22. Unity - Manual: Event function execution order. Unity - Manual: Unity 6.4 User Manual. URL: <https://docs.unity3d.com/Manual/execution-order.html> (date of access: 07.05.2026).

23. Unity - Manual: Built-in 3D physics. Unity - Manual: Unity 6.4 User Manual. URL: <https://docs.unity3d.com/6000.3/Documentation/Manual/PhysicsOverview.html> (date of access: 07.05.2026).

24. Physics solutions for game development | Unity. Unity. URL: <https://unity.com/solutions/programming-physics> (date of access: 07.05.2026).

25. Unity - Manual: Animator Controller Asset. Unity - Manual: Unity 6.4 User Manual. URL: <https://docs.unity3d.com/6000.0/Documentation/Manual/Animator.html> (date of access: 07.05.2026).

26. Unity - Manual: Animation clips. Unity - Manual: Unity 6.4 User Manual. URL: <https://docs.unity3d.com/6000.2/Documentation/Manual/AnimationClips.html> (date of access: 07.05.2026).

27. Unity - Manual: Animation state machine. Unity - Manual: Unity 6.4 User Manual. URL: <https://docs.unity3d.com/6000.3/Documentation/Manual/AnimationStateMachines.html> (date of access: 07.05.2026).

28. C# Guide - .NET managed language - C#. Microsoft Learn: Build with answers in reach. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/> (date of access: 07.05.2026).

29. Unity - Manual: Rigidbody component reference. Unity - Manual:

Unity 6.4 User Manual. URL:
<https://docs.unity3d.com/6000.6/Documentation/Manual/class-Rigidbody.html>
 (date of access: 07.05.2026).

30. Unity - Manual: Choose a collision detection mode. Unity - Manual:
 Unity 6.4 User Manual. URL:
<https://docs.unity3d.com/6000.3/Documentation/Manual/choose-collision-detection-mode.html> (date of access: 07.05.2026).

31. Unity - Manual: Configurable Joint component reference. Unity -
 Manual: Unity 6.4 User Manual. URL:
<https://docs.unity3d.com/6000.2/Documentation/Manual/class-ConfigurableJoint.html> (date of access: 07.05.2026).

32. Introduction: PID Controller Design. University of Michigan. URL:
<https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction§ion=ControlPID> (date of access: 07.05.2026).

33. Zhuge S. PID “Proportional, Integral, and Derivative” Control Theory.
 Crystal Instruments. URL:
<https://www.crystallinstruments.com/blog/2020/8/23/pid-control-theory> (date of access: 07.05.2026).

34. Proportional Integral Derivative (PID). APMonitor Optimization Suite. URL:
<https://apmonitor.com/pdc/index.php/Main/ProportionalIntegralDerivative> (date of access: 07.05.2026).

35. Perform Common Process Loop Control Algorithms. Smart Manufacturing Industrial Automation | Rockwell Automation | US. URL:
https://literature.rockwellautomation.com/idc/groups/literature/documents/wp/logix-wp008_-en-p.pdf (date of access: 07.05.2026).

36. Integral (Reset) Windup, Jacketing Logic and the Velocity PI Form – Control Guru. Control Guru – Practical Process Control. URL:
<https://controlguru.com/integral-reset-windup-jacketing-logic-and-the-velocity-pi-form/> (date of access: 07.05.2026).

37. Taylor D. PID Anti-windup Techniques. Erdos Miller: MWD Technology. URL: <https://info.erdosmiller.com/blog/pid-anti-windup-techniques> (date of access: 07.05.2026).
38. PD-controllers. MatthewPeterKelly. URL: <https://www.matthewpeterkelly.com/tutorials/pdControl/> (date of access: 07.05.2026).
39. Apperson L. Beginning Game Development: Inverse Kinematics. Medium. URL: <https://medium.com/@lemapp09/beginning-game-development-inverse-kinematics-00177650c4b2> (date of access: 07.05.2026).
40. Juckett R. Analytic Two-Bone IK in 2D. RyanJuckett.com. URL: <https://www.ryanjuckett.com/analytic-two-bone-ik-in-2d/> (date of access: 07.05.2026).
41. Kaufmann M. Procedural Animation: Inverse Kinematics. Little Polygon Game Dev Blog. URL: <https://blog.littlepolygon.com/posts/twobone/> (date of access: 07.05.2026).
42. Kaufmann M. Procedural Animation: Locomotion (Part 1). Little Polygon Game Dev Blog. URL: <https://blog.littlepolygon.com/posts/loco1/> (date of access: 07.05.2026).
43. Lecluyse B. Exploring procedural animation in Trifox - Trifox. Trifox. URL: <https://www.trifox-game.com/exploring-procedural-animation-in-trifox/> (date of access: 07.05.2026).
44. Margison L. Procedural animation for humans in Unreal Engine 5. Udemy. URL: <https://www.udemy.com/course/procedural-animation/?couponCode=MT260504G1> (date of access: 07.05.2026).
45. AI Navigation | AI Navigation | 2.0.12. Unity - Manual: Unity 6.4 User Manual. URL: <https://docs.unity3d.com/Packages/com.unity.ai.navigation@2.0/manual/> (date of access: 07.05.2026).
46. Red Blob Games: Introduction to the A* Algorithm. Red Blob Games. URL: <https://www.redblobgames.com/pathfinding/a-star/introduction.html> (date of

access: 07.05.2026).

47. Creating a NavMesh Agent | AI Navigation | 1.1.7. Unity - Manual:
 Unity 6.4 User Manual. URL:
<https://docs.unity3d.com/Packages/com.unity.ai.navigation@1.1/manual/CreateNavMeshAgent.html> (date of access: 07.05.2026).

48. Stopping Distance Formula: Definition, Derivation and More. Embibe Exams. URL: <https://www.embibe.com/exams/stopping-distance-formula/> (date of access: 07.05.2026).

ДОДАТОК А
Технічне завдання

A.1 Підстава для розробки

Підставою для розробки є завдання на дипломну кваліфікаційну роботу на тему «Програмна реалізація керування активним регдолом для самобалансування і переміщення у ігровій механіці», затверджене наказом Національного університету «Запорізька політехніка» № 139 від 7 квітня 2026 р.

A.2 Призначення розробки

Програмна система призначена для керування фізично симульованим персонажем у ігровому середовищі.

A.3 Основні вимоги до програми, що розробляється

A.3.1 Вимоги до функціональних характеристик

Програмна система керування активним регдолом забезпечує виконання таких функцій:

- самостійне підтримання вертикального положення тіла персонажа;
- протидія зовнішнім збуренням та утримання рівноваги;
- цілеспрямоване переміщення персонажа по заданому маршруту;
- адаптація швидкості руху до характеру маршруту та рельєфу;
- коректна постановка стоп на нерівній поверхні та перешкодах;
- падіння та самостійний підйом до стоячої позиції у разі втрати рівноваги;
- гнучке налаштування поведінки персонажа без модифікації вихідного коду.

Програмна система містить тестове середовище для перевірки роботи у типових сценаріях.

A.3.2 Вимоги до надійності

Програмна система керування активним регдолом повинна забезпечити надійне функціонування у режимі реального часу.

A.3.3 Умови експлуатації

Для експлуатації програмної системи керування активним регдолом необхідна наявність персонального комп'ютера.

A.3.4 Вимоги до складу та параметрів технічних засобів

Як апаратно-технічні засоби для експлуатації системи повинен використовуватися персональний комп'ютер з характеристиками не нижче:

- процесор – Intel Core i5 10400 чи AMD Ryzen 5 3600;
- обсяг оперативної пам'яті – 8 ГБ;
- обсяг відеопам'яті – 4 ГБ;
- відеокарта – NVIDIA GeForce RTX 2070;
- операційна система – Microsoft Windows 10.

ДОДАТОК Б
Текст програми

Б.1 Текст жизненного цикла NPARMainController.cs

```
private void FixedUpdate()
{
    if (!IsInitialized) return;

    // --- 0. Sin Wobble Left-Right ---
    ApplySinWobble();

    // --- 1. Update Core State ---
    UpdateCoreRagdollState(); // Calculates Pseudo-Forward, Balance Data,
    CoM, Critical State

    // --- 2. Apply Physical Forces & Torques ---
    ApplyPhysicalForces(); // Applies PID, Springs, Upward Force, Damping

    // --- 3. Update Joint Targets ---
    UpdateJointTargets(); // Sets target rotations for individual limb joints
}

private void LateUpdate()
{
    if (!IsInitialized) return;

    // --- 4. Update Animated Body Transform ---
    UpdateAnimatedBodyTransform();

    // --- 5. Move Physical Body Root to projected CoM (for LOD) ---
    UpdatePhysicalBodyRootPosition();

    // --- 6. Set Center of Mass Transform (CoM, PseudoForward) ---
    UpdateCenterOfMassTransform();
}
```

Б.2 Текст файла PIDPositionController.cs

```
using NaughtyAttributes;
using UnityEngine;
using SHLP.Utils;
using System;
```

```
#pragma warning disable CS0414 // Suppress warning with simulation variable. It
is not used in this script, but in the editor script
```

```

[System.Serializable]
public class PIDPositionController : PIDController
{
    private Vector3 targetPosition = Vector3.zero;

    [Header("Limits")]
    [SerializeField, Min(0.001f)] private float maxForce = 50f;

    [Header("Debug")]
    [SerializeField] private bool drawGizmos = false;

    [Header("Graph Simulation")]
    [SerializeField] private float targetSimulationMagnitude = 0.5f;

    // Internal PID Variables
    private Vector3 previousIntegral = Vector3.zero;
    private Vector3 previousError = Vector3.zero;

    // Runtime overrides, null means no override
    private float? overrideMaxForce;
    private float EffectiveMaxForce => overrideMaxForce ?? maxForce;

    // Public Properties
    public Vector3 CorrectionForce { get; private set; }
    public Vector3 CorrectionForceNormalized => CorrectionForce / maxForce;

    public PIDPositionController(Rigidbody rigidbody)
    {
        base.rigidbody = rigidbody;
        targetPosition = base.rigidbody.position;
    }
    public PIDPositionController(Rigidbody rigidbody, Vector3 targetPos)
    {
        base.rigidbody = rigidbody;
        targetPosition = targetPos;
    }
    public PIDPositionController(PIDPositionController other)
    {
        rigidbody = other.rigidbody;
        targetPosition = other.targetPosition;

        proportionalGain = other.proportionalGain;
        integralGain = other.integralGain;
        derivativeGain = other.derivativeGain;
    }

```

```

    maxForce = other.maxForce;

    targetSimulationMagnitude = other.targetSimulationMagnitude;
}

public Vector3 Update(float deltaTime)
{
    if (rigidbody == null)
    {
        Debug.LogError("PID Controller doesn't have rigidbody linked. Can't
update.");
        return Vector3.zero;
    }

    // Calculate error
    Vector3 positionError = GetCurrentError();

    // Proportional term
    Vector3 pTerm = positionError * EffectiveP;

    // Integral term
    if (CorrectionForce.sqrMagnitude < (EffectiveMaxForce *
EffectiveMaxForce)) // Anti windup (stop increasing integral if force exceeds max)
        previousIntegral += positionError * deltaTime;
    Vector3 iTerm = previousIntegral * EffectiveI;

    // Derivative term
    Vector3 derivative = (positionError - previousError) / deltaTime;
    Vector3 dTerm = derivative * EffectiveD;

    // Calculate total force
    Vector3 pidForce = pTerm + iTerm + dTerm;

    // Clamp force to maximum
    CorrectionForce = Vector3.ClampMagnitude(pidForce, EffectiveMaxForce);

    // Store error for next frame's derivative calculation
    previousError = positionError;

    if (drawGizmos)
    {
        Debug.DrawRay(rigidbody.position, pTerm / EffectiveMaxForce,
Color.blue, Time.fixedDeltaTime);
        Debug.DrawRay(rigidbody.position, iTerm / EffectiveMaxForce,
Color.green, Time.fixedDeltaTime);
    }
}

```

```

        Debug.DrawRay(rigidbody.position, dTerm / EffectiveMaxForce,
Color.red, Time.fixedDeltaTime);
    }

    return CorrectionForce;
}

// Setters
public void SetTargetPosition(Vector3 newTarget)
{
    targetPosition = newTarget;
}
public void SetMaxForce(float newMaxForce)
{
    maxForce = Mathf.Max(newMaxForce, 0.001f);
}

// Getters
public Vector3 GetTargetPosition() => targetPosition;
public Vector3 GetCurrentError() => targetPosition - rigidbody.position;
public float GetMaxForce() => maxForce;
public float GetEffectiveMaxForce() => EffectiveMaxForce;

public void CopySettingsFrom(PIDPositionController other)
{
    base.CopySettingsFrom(other);

    maxForce = other.maxForce;
    targetSimulationMagnitude = other.targetSimulationMagnitude;
}

public void CopySettingsFromWithPresetLerp<TData>(
    MonoBehaviour coroutineRunner,
    ControllerBase<TData> target,
    PIDPositionController other,
    float applyTime,
    string key
) where TData : ControllerBase<TData>
{
    base.CopySettingsFromWithPresetLerp(coroutineRunner, target, other,
applyTime, key);

    PresetUtils.StartLerp(coroutineRunner, target, string.Join('.', key,
nameof(maxForce)), maxForce, other.maxForce, applyTime, value => maxForce =
value);

```

```

    targetSimulationMagnitude = other.targetSimulationMagnitude;
}

#region Overrides

public override void SetGainOverrides(PIDControllerOverrides overrides)
{
    base.SetGainOverrides(overrides);
    overrideMaxForce = overrides.LimitOverride;
}

public override void ClearGainOverrides()
{
    base.ClearGainOverrides();
    overrideMaxForce = null;
}

#endregion
}

```

Б.3 Текст аплікації пресету в NPARPresetManager.cs

```

/// <summary>
/// Apply a preset to one of the controllers. Controller type is chosen based on
preset type.
/// </summary>
/// <param name="preset">Preset to apply. </param>
/// <param name="lerp">Interpolate over time? </param>
/// <param name="forced">Ignore preset application restrictions? </param>
public void ApplyPreset<TData>(PresetBase<TData> preset, bool lerp = true,
bool forced = false) where TData : ControllerBase<TData>
{
    if (preset == null) return;
    if (!forced && MainController.Brain != null &&
!MainController.Brain.allowedToChangePreset) return;

    if (preset is MainControllerPreset mainControllerPreset)
    {
        if (!MainController) return;
        MainController.ApplyPreset(mainControllerPreset, lerp);
        EditorUtility.SetDirty(MainController);
    }
    if (preset is MuscleControllerPreset musclePreset)
    {

```

```

        if (!MuscleController) return;
        MuscleController.ApplyPreset(musclePreset, lerp);
        EditorUtility.SetDirty(MuscleController);
    }
    else if (preset is BodyControllerPreset bodyPreset)
    {
        if (!BodyController) return;
        BodyController.ApplyPreset(bodyPreset, lerp);
        EditorUtility.SetDirty(BodyController);
    }
    else if (preset is ArmsControllerPreset armsPreset)
    {
        if (!ArmsController) return;
        ArmsController.ApplyPreset(armsPreset, lerp);
        EditorUtility.SetDirty(ArmsController);
    }
    else if (preset is LegsControllerPreset legsPreset)
    {
        if (!LegsController) return;
        LegsController.ApplyPreset(legsPreset, lerp);
        EditorUtility.SetDirty(LegsController);
    }
}

```

Б.4 Текст жизненного цикла NPARLegsController.cs

```

private void FixedUpdate()
{
    if (!isInitialized) return;

    if (Data.LegsControllerWeight > 0f)
    {
        // Prepare for new frame (update heel toes ratio and reset offsets)
        legsList.ForEach(leg => leg.UpdateIKTargetHeelToesRatio());

        leftLeg.ResetOffsets();
        rightLeg.ResetOffsets();

        // --- 1. Update Input Data & Ragdoll State ---
        CacheRagdollState();
        UpdateGroundedStatus();
        UpdateLegVelocities();
        UpdateAllowedIdleDistance();
        UpdateCurrentRaycastDistance();
        CalculateCycle();
    }
}

```

```

// --- 1.5. Sin Microjumps for Fight State ---
ApplySinMicrojumps();

// --- 2. Calculate Target States ---
UpdateLegHeightsAndCrouch();
UpdateCurrentLegPoses();
UpdateBaseRaycastOrigins();
UpdateIdleRaycastOrigins();
legsList.ForEach(leg => CalculateIdleTransform(leg));

// --- 3. Decision Making & Steps ---
HandleSteppingLogic();
legsList.ForEach(leg => leg.UpdateRawTargetFootPoints());

// --- 4. Apply Corrections & Effects ---
CalculateLegLengths();
ApplyVelocityCounteraction();
ApplyFeetPushdown();
ApplyLandingOffset();
ApplyRotationOvershoot();
ApplyFeetHeightLimits();
legsList.ForEach(leg => ApplyDistanceRotation(leg));
}
else
{
    // Update basic things while controller weight is 0
    legsList.ForEach(leg => leg.UpdateIKTargetHeelToesRatio());

    leftLeg.ResetOffsets();
    rightLeg.ResetOffsets();

    CacheRagdollState();

    UpdateBaseRaycastOrigins();

    legsList.ForEach(leg =>
    {
        leg.idleRaycastOrigin = leg.baseRaycastOrigin;
        CalculateIdleTransform(leg);

        leg.rawTargetPosition = leg.idlePosition;
        leg.rawTargetRotation = leg.idleRotation;

        leg.isStepping = false;
    });
}

```

```

    });
}

// --- 5. Apply IK ---
leftLeg.ApplyIK(Data.LegsControllerWeight);
rightLeg.ApplyIK(Data.LegsControllerWeight);
}

private void LateUpdate()
{
    // Doing the sync after Animator IK updated
    leftLeg.SyncRawTargetTransform();
    rightLeg.SyncRawTargetTransform();
}

```

Б.5 Текст розрахунку циклу ходи в NPARLegsController.cs

```

private void CalculateCycle()
{
    // --- Cycle Blend Alpha ---
    if (Data.ForceReactiveOnly)
    {
        cycleBlendAlpha = 0f;
    }
    else if (Data.ForceCycleOnly)
    {
        cycleBlendAlpha = 1f;
    }
    else
    {
        float targetCycleBlendAlpha =
Mathf.InverseLerp(Data.CycleBlendMinSpeed, Data.CycleBlendMaxSpeed,
comSpeed);
        float diff = targetCycleBlendAlpha - cycleBlendAlpha;
        cycleBlendAlpha += Mathf.Clamp(diff, -Data.CycleBlendLerpSpeed *
Time.fixedDeltaTime, Data.CycleBlendLerpSpeed * Time.fixedDeltaTime);
        cycleBlendAlpha = Mathf.Clamp01(cycleBlendAlpha);
    }

    // --- Master Cycle Timings ---
    cycleLengthInSeconds = EffectiveSwingTimeInSeconds;
    float effectiveMaxStrideLength = EffectiveMaxStrideLength;
    cycleLengthInSeconds += effectiveMaxStrideLength / Mathf.Max(comSpeed,
0.5f);
}

```

```

    swingTimeAsAlpha = EffectiveSwingTimeInSeconds /
cycleLengthInSeconds;

    // --- Master Cycle Sync ---
    // While not fully in cycle mode, nudge the master alpha toward what the legs'
current
    // states correlate to. This makes sure, that when engaging cycle it lings up
with reactive steps.
    // Only sync when no leg is currently doing a cycle-triggered step.
    // Skipped entirely at cycleBlendAlpha == 0 — in pure reactive there's
nothing to align to yet,
    // and pulling master around based on reactive steps can stuff legs into the
cycle trigger window
    // once blend rises, causing double steps.
    bool anyCycleStepInProgress = legsList.Any(leg => leg.isStepping &&
leg.stepTriggeredByCycle);
    if (!anyCycleStepInProgress && cycleBlendAlpha > 0f)
    {

        float masterAlphaSum = 0f;
        int validLegs = 0;

        for (int i = 0; i < legsList.Count; i++)
        {
            LegData leg = legsList[i];
            if (!leg.firstStepDone) continue;
            if (!leg.isStepping) continue;

            float impliedLegCycleAlpha;

            // Reactive step in progress: map step progress (0->1) into swing phase
(0->swingTimeAsAlpha)
            impliedLegCycleAlpha = leg.stepLinearAlpha * swingTimeAsAlpha;

            // Back-calculate what master alpha this leg implies (undo the per-leg
phase offset)
            float impliedMasterAlpha = (impliedLegCycleAlpha - i * 0.5f + 1f) %
1f;

            masterAlphaSum += impliedMasterAlpha;
            validLegs++;
        }

        if (validLegs > 0)

```

```

    {
        float targetMasterAlpha = masterAlphaSum / validLegs;

        // Shortest-path, framerate-independent move on the circular [0, 1)
domain.    // delta in [-0.5, 0.5] is the signed shortest distance from current to target.
        float delta = ((targetMasterAlpha - stepMasterCycleAlpha + 1.5f) % 1f) -
0.5f;
        float syncStrength = Mathf.Clamp01(Data.CycleSyncLerpSpeed *
Time.fixedDeltaTime);
        syncStrength = 1f;
        stepMasterCycleAlpha = (stepMasterCycleAlpha + delta * syncStrength
+ 1f) % 1f;
    }
}

// --- Advance Master Cycle ---
stepMasterCycleAlpha = (stepMasterCycleAlpha + Time.fixedDeltaTime /
cycleLengthInSeconds) % 1.0f;

float cycleOffset;
for (int i = 0; i < legsList.Count; i++)
{
    LegData leg = legsList[i];
    cycleOffset = 0.5f;
    if (!leg.firstStepDone) cycleOffset = 0f;

    leg.stepCycleAlpha = (stepMasterCycleAlpha + i * cycleOffset) % 1.0f;
}
}

```

ДОДАТОК В
Слайди презентації

Національний університет «Запорізька політехніка»
Кафедра програмних засобів
Дипломна кваліфікаційна робота бакалавра

Програмна реалізація керування активним регдолом для самобалансування і переміщення в ігровій механіці

Виконав:

ст. групи КНТ-132

Максим ГЛІБОВ

Керівники:

доктор філософії, доцент

Сергій ЛЕОЩЕНКО

1

Рисунок В.1 – Слайд 1

Мета, об'єкт, предмет і задачі дослідження

- Мета – підвищення реалістичності поведінки персонажа та його взаємодії з оточенням у відеоіграх шляхом розроблення програмних засобів керування активним регдолом на основі фізичної симуляції.
- Об'єкт – процес розроблення програмного забезпечення для керування фізично симульованими персонажами у відеоіграх.
- Предмет – методи та програмні засоби керування активним регдолом на основі PID-регуляторів, пружинних сил і м'язових моментів для забезпечення самобалансування та переміщення персонажа.
- Завдання:
 - проаналізувати існуючі підходи до анімації персонажів;
 - розробити архітектуру програмної системи керування активним регдолом;
 - реалізувати алгоритми самобалансування та переміщення;
 - дослідити поведінку розробленої системи в ігровому середовищі.

2

Рисунок В.2 – Слайд 2

Існуючі реалізації активного регдолу

Критерій	Euphoria	UE Physical Animation	Human: Fall Flat
Спосіб керування суглобами	Нейром'язова симуляція	Звичайні пружини у суглобах	PD-подібні регулятори
Реакція на зовнішні сили	Повна, з урахуванням усього тіла	Часткова, налаштовується по кістках	Повна, для всього тіла
Активне балансування	Так, повноцінне	Ні	Так, спрощене
Адаптивність до оточення	Висока	Часткова	Висока
Реалістичність руху	Висока	Залежить від анімації	Навмисно спрощена
Доступність для розробників	Комерційна ліцензія	Вбудовано в Unreal Engine	Реалізація на Unity

3

Рисунок В.3 – Слайд 3

Вибір ігрового рушія

Критерій	Unity	Unreal Engine	Godot
Мова скриптів	C#	C++/Blueprints	GDScript/C#
Фізичний рушій	PhysX	Chaos	Jolt/Godot Physics
Гнучкість налаштування фізики	високий контроль	вищий контроль на низькому рівні	базові налаштування
Робота з анімаціями	Animator + Animation Rigging, окремий пакет ІК	Animation Blueprint + Control Rig, єдина графова система	Animation Tree + SkeletonIK, спрощена модель
Якість документації та спільноти	велика база туторіалів, офіційна документація з прикладами	велика, але орієнтована на AAA-розробку	менша спільнота, документація розвивається
Особистий досвід розробника	значний	базовий	відсутній

4

Рисунок В.4 – Слайд 4

Вибір середовища розробки

Критерій	Visual Studio Code	Visual Studio	JetBrains Rider
Підтримка C# для Unity	C# Dev Kit +Unity-розширення	Вбудована, через Unity Workload	Плагін Unity Support
Швидкість запуску та споживання ресурсів	мінімальне, миттєвий запуск	високе, тривалий запуск	високе, повільніший запуск
Розширення	вбудований Marketplace, одна кнопка	окремий менеджер у Tools	менеджер плагінів JetBrains
Ліцензія	вільна	Community безкоштовна / Pro платна	платна
Особистий досвід розробника	значний	значний	відсутній

Рисунок В.5 – Слайд 5

Концепція активного регдолу: принцип подвійного скелета

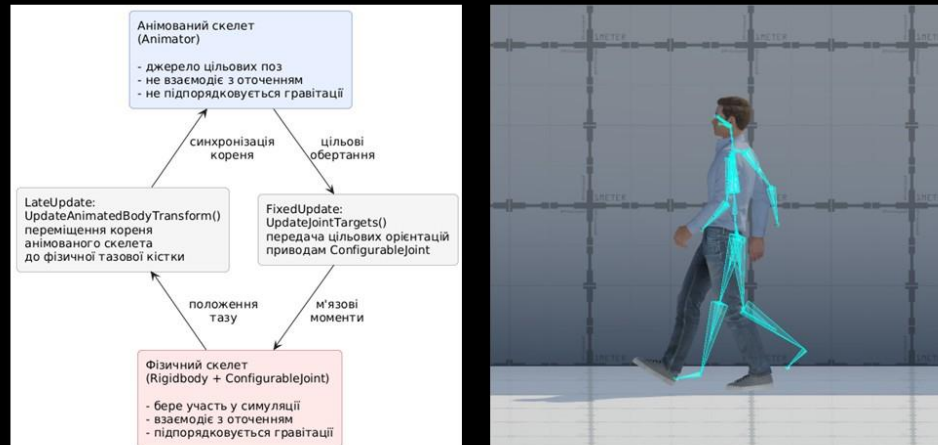
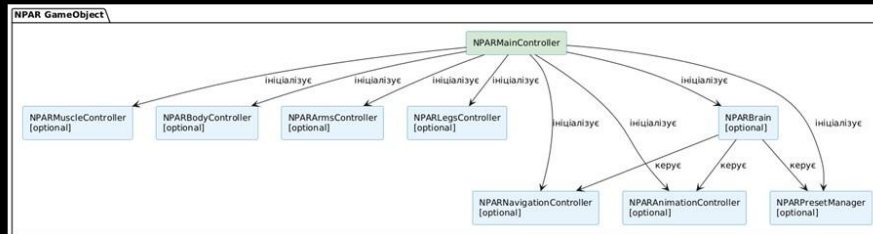


Рисунок В.6 – Слайд 6

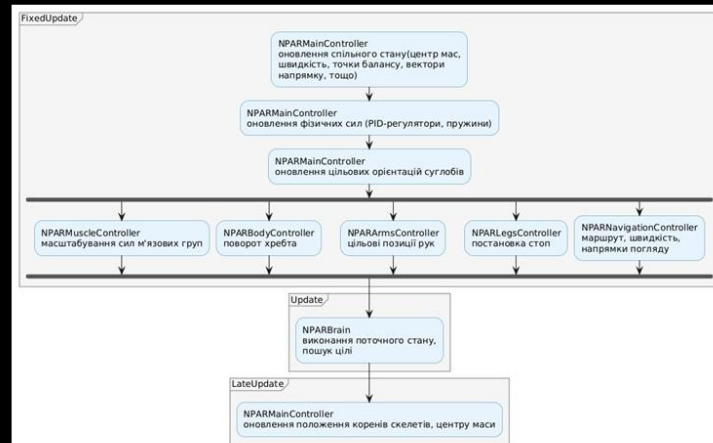
Архітектура системи NPAR



7

Рисунок В.7 – Слайд 7

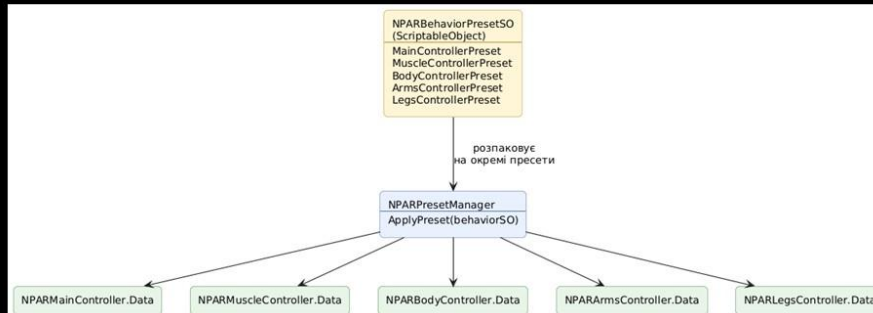
Архітектура системи NPAR



8

Рисунок В.8 – Слайд 8

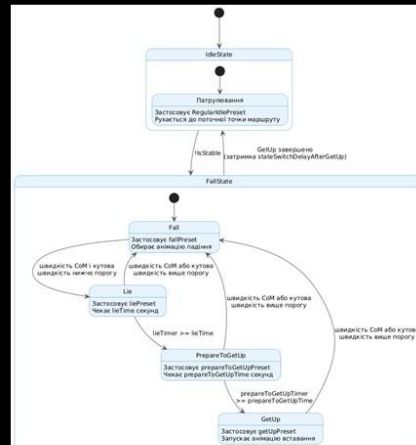
Системи пресетів і налаштування поведінки



9

Рисунок В.9 – Слайд 9

Машина станів та прийняття рішень



10

Рисунок В.10 – Слайд 10

PID-регулятор: позиційна та інкрементальна форми

Позиційна форма:

$$u(t) = K_p \times e(t) + K_i \times \int e(t)dt + K_d \times \frac{de(t)}{dt}$$

Інкрементальна (швидкісна) форма:

$$\Delta u(t) = \frac{K_p \times (e(t) - e(t-1)) + K_i \times e(t) \times \Delta t + K_d \times (e(t) - 2e(t-1) + e(t-2)))}{\Delta t}$$

$$u(t) = u(t-1) + \Delta u(t)$$

11

Рисунок В.11 – Слайд 11

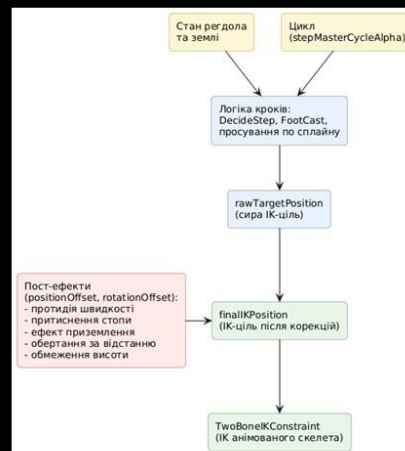
Застосування PID для самобалансування



12

Рисунок В.12 – Слайд 12

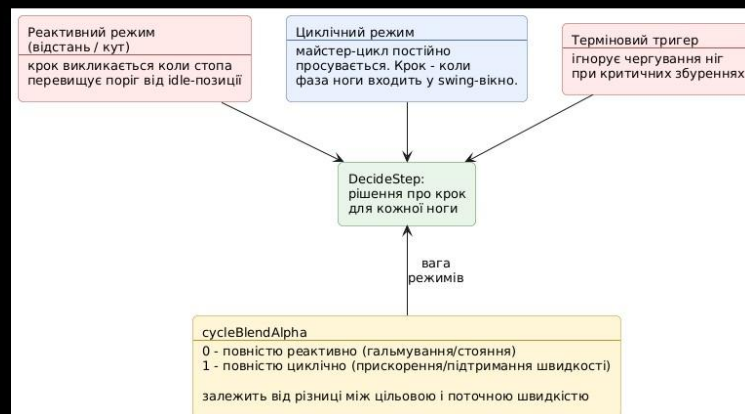
Контролер ніг: архітектура та пост-ефекти



13

Рисунок В.13 – Слайд 13

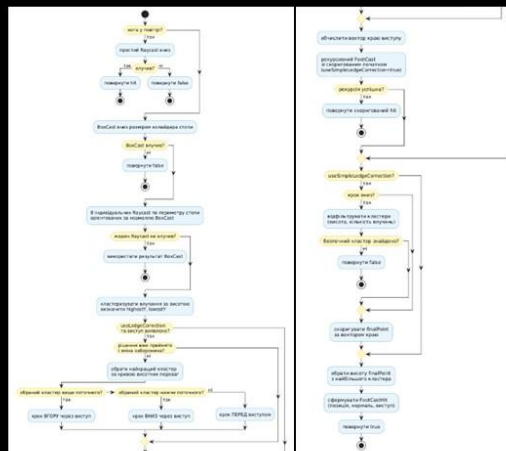
Контролер ніг: гібридна модель прийняття рішень



14

Рисунок В.14 – Слайд 14

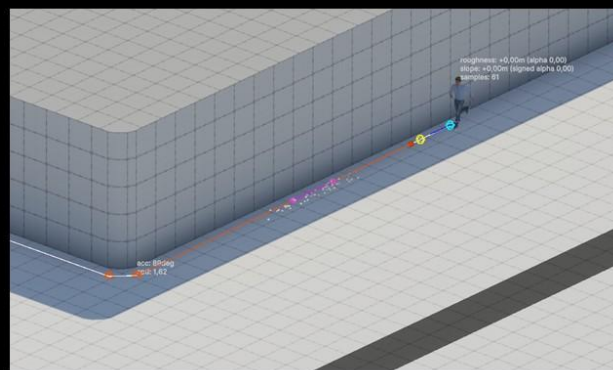
Контролер ніг: алгоритм FootCast



15

Рисунок В.15 – Слайд 15

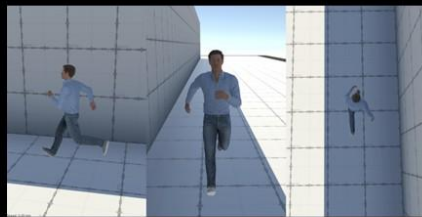
Контролер навігації: планування швидкості



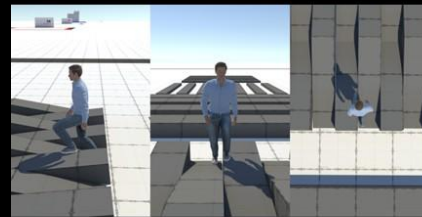
16

Рисунок В.16 – Слайд 16

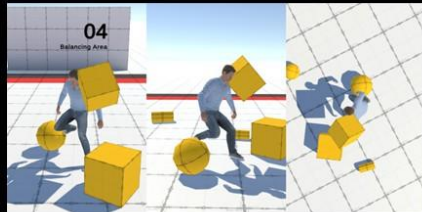
Результати тестування



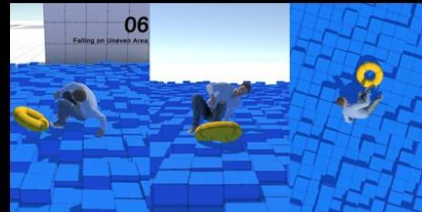
Біг



Високі перешкоди



Під ударом об'єктів



Підйом після падіння

17

Рисунок В.17 – Слайд 17

Узагальнені результати

Зона	Перевірялось	Результат
Ходьба	Стабільність циклу ходи	Стабільний ритмічний цикл без відхилень. Нестабільність на крутих поворотах.
Біг	Цикл бігу, планування швидкості на поворотах	Максимальна швидкість 5,2 м/с; пологий поворот про-ходитьс-ся без суттєвого гальмування з нахилом кор-пусу. Нестабільність на крутих поворотах.
Нерівний рельєф	Адаптація до перешкод	Коректна постановка стоп на всіх типах перешкод. Іноді неточне повторення цільових рухів фізичним тілом, неста-більність на сходах.
Балансування	Стійкість під ударами	Падіння раз на 7-10 кидків; підйом коректний. Іноді занадто пружний корпус.
Балансування на одній нозі	Стійкість на одній нозі під ударами	Падіння раз на 7-10 кидків; підйом коректний. Іноді ненатуральне підтримання балансу.
Балансування на нерівній поверхні	Стійкість та від-новлення на складному рельєфі	Падіння раз на 4-7 кидків; підйом коректний.
Падіння з висоти	Поведінка в польоті, вибір варіанту падіння	Коректний вибір варіанту падіння та наступного підйому.

18

Рисунок В.18 – Слайд 18

Висновки

У дипломній роботі розроблено програмну систему керування активним реґолом NPAR, що забезпечує самобалансування та переміщення фізично симульованого персонажа в ігровому середовищі. Система побудована на принципі подвійного скелета з модульною архітектурою з семи контролерів, де поведінка налаштовується підміною пресетів без зміни коду. Балансування реалізоване через три незалежні PID-регулятори, доповнені пружинно-демпферними м'язовими моментами на рівні суглобів.

Контролер ніг поєднує процедурну постановку стоп через гібридну реактивно-циклічну модель кроків і алгоритм для адаптації до рельєфу. Контролер навігації застосовує фізично-обізнане планування швидкості з кластеризацією поворотів і зворотною індукцією.

Експериментальне дослідження у семи тестових зонах підтвердило практичну придатність системи: NPAR демонструє стабільну ходу та біг до 5,2 метра на секунду, коректну адаптацію до нерівного рельєфу та надійне відновлення після падінь. Розроблена система є придатною до застосування у відеоіграх для підвищення імерсивності та реалістичності поведінки персонажів і може слугувати основою для подальших розширень.