Міністерство освіти і науки України Тернопільський національний педагогічний університет імені Володимира Гнатюка

Фізико-математичний факультет

Кафедра інформатики та методики її навчання

Кваліфікаційна робота

Розробка комп'ютерної гри жанру First Person Shooter

Спеціальність:

122 Комп'ютерні науки Освітня програма «Інженерія ігрових проєктів»

Здобувача вищої освіти освітньо-кваліфікаційного рівня «бакалавр»
Бровчука Крістіана Романовича

НАУКОВИЙ КЕРІВНИК:
Габрусєв Валерій Юрійович,
кандидат педагогічних наук, доцент кафедри інформатики та методики її навчання

РЕЦЕНЗЕНТ:
Габрусєва Ірина Юріївна,
кандидат технічних наук,
доцент кафедри вищої математики Тернопільського національного університету імені Івана Пулюя

Тернопіль – 2026

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

FPS (First Person Shooter) – шутер від першої особи, жанр комп'ютерних ігор, де гравець спостерігає за подіями з перспективи власного персонажа.

FSM (Finite State Machine) – скінченний автомат станів, математична модель обчислень, яка описує поведінку об'єкта через набір станів та правила переходів між ними.

GDD (Game Design Document) – ігровий дизайн-документ, формальний технічний документ, що описує концепцію, механіки та всі аспекти розроблюваної гри.

GDScript – мова програмування з Python-подібним синтаксисом, розроблена безпосередньо для рушія Godot Engine.

GPU (Graphics Processing Unit) – графічний процесор, спеціалізований електронний пристрій, призначений для прискорення операцій з обробки та виведення зображень.

HFSM (Hierarchical Finite State Machine) – ієрархічний скінченний автомат станів, розширена версія FSM з підтримкою вкладених станів.

MIT – ліцензія відкритого програмного забезпечення Массачусетського технологічного інституту, що надає широкі права на використання, копіювання та розповсюдження.

NPC (Non-Player Character) – непідконтрольний гравцеві персонаж, керований ігровою логікою або штучним інтелектом.

QOL (Quality of Life) – якість взаємодії, сукупність другорядних механік, які не змінюють ключових правил гри, однак суттєво підвищують зручність та комфорт ігрового досвіду.

Raycast (Raycasting) – метод трасування променя у просторі тривимірної сцени для виявлення перетинів із геометрією або колайдерами.

Vulkan – міжплатформний низькорівневий API для роботи з тривимірною графікою та обчисленнями на GPU, розроблений Khronos Group як наступник OpenGL.

ВСТУП

В сьогоденні, відеоігри являються такою ж невід'ємною частиною посякденного життя мільйонів людей, як і перегляд фільмів, серіалів, переслуховування музики онлайн чи споживання любого іншого цифрового контенту. Ринок відеоігор є одним із найбільш динамічно зростаючих секторів глобальної цифрової економіки на сьогодні, за оцінками дослідницьких агенцій, його обсяг у 2024 році перевищив 282 млрд доларів США, а прогнозована вартість ринку до 2030 року, за різними підрахунками, коливається від 600 до 721 млрд доларів при середньорічному темпі зростання понад 10% [7]. Такі показники свідчать про те, що відеоігри вже давно перестали бути суто розважальним явищем і перетворились на повноцінну культурну та технологічну індустрію, що породжує попит на широкий спектр фахівців – від програмістів і художників до звукових дизайнерів і нарратологів.

Серед багатьох жанрів, що утворюють сучасний пласт відеоігор, особливе місце займають шутери від першої особи (First Person Shooter, FPS). Жанр бере свій початок від ранніх тривимірних ігор початку 1990-х років, типу Wolfenstein 3D (1992) чи DOOM (1993), за понад три десятиліття перетворившись на один із найпопулярніших та найтехнічно вимогливіших напрямів ігрової розробки. Він охоплює мільярди гравців по всьому світу та щорічно генерує мільярди доларів доходу від продажів та монетизації. Ключовою характеристикою жанру є занурення гравця в ігровий простір через перспективу від першої особи, що виходить з назви жанру, та у поєднанні з реакційними механіками стрільби та динамічним переміщенням формує унікальний інтерактивний досвід.

Зростаючий ринок ігор обумовлює відповідний попит на фахівців із ігрової розробки, тому питання підготовки розробників, ознайомлення їх із сучасними інструментами та підходами є більш актуальним, ніж будь-коли. Одним із таких інструментів є відкритий ігровий рушій Godot Engine, який за останні роки демонструє стрімке зростання популярності: кількість ігор, випущених з його допомогою у Steam, зросла від 47 у 2020 році до понад 1500 у 2025 [21]. Відкритий вихідний код, ліцензія MIT, відсутність роялті та нативна підтримка

Vulkan-рендерера роблять Godot Engine привабливою платформою як для інді-розробників, так і для навчальних і дослідницьких проєктів.

Метою цієї кваліфікаційної роботи є розробка прототипу комп'ютерної гри жанру FPS засобами рушія Godot Engine версії 4.6+ із реалізацією базових механік жанру: системи керування персонажем, механіки застосування вогнепальної зброї, системи реєстрації пошкоджень та набору QOL-елементів, що підвищують рівень занурення в ігровий процес.

Об'єктом дослідження є процес проєктування та розробки тривимірних комп'ютерних ігор жанру шутер від першої особи, а також методи організації їх програмної архітектури та взаємодії основних ігрових підсистем. Особлива увага приділяється реалізації механік, що безпосередньо впливають на ігровий процес та користувацький досвід, зокрема систем керування персонажем, взаємодії зі зброєю, обробки пошкоджень і допоміжних механізмів, спрямованих на підвищення рівня занурення гравця в ігрове середовище.

Предметом дослідження виступають методи, алгоритми, архітектурні патерни та програмні засоби реалізації ключових механік FPS-ігор на базі рушія Godot Engine. До предметної області дослідження належать підходи до побудови систем керування персонажем із використанням скінченних автоматів станів, реалізація модульної системи зброї, механізмів стрільби та реєстрації влучань, організація взаємодії між окремими ігровими компонентами за допомогою сигналів і ресурсів рушія, а також способи реалізації динамічної камери та інших QOL-механік, що покращують сприйняття ігрового процесу користувачем.

Для досягнення поставленої мети в роботі вирішуються такі завдання: аналіз теоретичних основ жанру FPS та огляд ключових механік, що формують досвід гравця; порівняльна характеристика сучасних ігрових рушіїв та обґрунтування вибору Godot Engine; дослідження архітектурного патерну скінченного автомату станів стосовно керування персонажем; проєктування концептуальної моделі ігрового прототипу та складання технічного завдання; практична реалізація всіх ключових підсистем гри засобами GDScript; тестування прототипу та аналіз отриманих результатів.

Практичне значення роботи полягає у створенні функціонального прототипу тривимірної комп'ютерної гри жанру FPS із реалізованими базовими механіками, характерними для сучасних шутерів від першої особи. У межах проєкту розроблено систему керування персонажем, модульну систему зброї, механіку стрільби а також комплекс допоміжних систем, зокрема динамічну камеру та інші елементи, спрямовані на покращення користувацького досвіду. Результатом роботи є програмний продукт, який демонструє можливості використання рушія Godot Engine для створення інтерактивних тривимірних застосунків і може слугувати основою для подальшого розвитку повноцінної гри. Систематизований у роботі досвід використання рушія Godot Engine, мови програмування GDScript та архітектурних патернів розробки гри становить практичну цінність для студентів, початківців-розробників та всіх, хто вивчає сучасні технології створення комп'ютерних ігор. Матеріали роботи можуть бути використані як навчально-методична основа під час викладання дисциплін, пов'язаних із програмуванням, комп'ютерною графікою, проєктуванням програмного забезпечення та ігровою розробкою.

За темою кваліфікаційної роботи було опубліковано тези:

1. Бровчук К. Р. Розробка комп'ютерної гри жанру First Person Shooter засобами рушія Godot Engine / К. Р. Бровчук, В. Ю. Габрусєв // Матеріали конференції. – Тернопіль, 2026.

Структура роботи. Кваліфікаційна робота складається зі вступу, трьох розділів, висновків та списку використаних джерел. У першому розділі викладено теоретичні основи розробки ігор жанру FPS. Другий розділ присвячено проєктуванню ігрового прототипу. Третій розділ описує безпосередню реалізацію проєкту на рушії Godot Engine 4. Загальний обсяг роботи становить 61 сторінку, список джерел налічує 40 позицій.

РОЗДІЛ 1 ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ ІГОР ЖАНРУ FPS

1.1 Жанр FPS: генезис, класифікація та сучасний стан

Шутери від першої особи як самостійна жанрова категорія сформувались на рубежі 1980-х – 1990-х років, хоча передумови виникнення жанру можна відшукати ще в ранніх аркадних автоматах та текстових пригодницьких іграх, де гравцю пропонувалася суб'єктивна точка зору. Першою значущою грою, яку прийнято вважати родоначальником жанру у його сучасному розумінні, є Maze War (1973–1974), що дозволяла переміщатись лабіринтом від першої особи та знищувати суперників. Однак справжнє народження жанру пов'язане з виходом Wolfenstein 3D від id Software у 1992 році – гри, що вперше запропонувала технічно переконливий тривимірний простір, швидке переміщення та перестрілко-орієнтований геймплей, сформувавши тим самим шаблон для всіх наступних представників жанру.

Революційним поштовхом для подальшого розвитку FPS стала гра DOOM (1993), також розроблена id Software. Вона не лише значно підвищила технічну планку жанру, впровадивши псевдо-тривимірний рендеринг на основі трасування секторів, а й запровадила мережеву гру типу deathmatch, заклавши фундамент для всього жанру онлайн-шутерів. Quake (1996), наступний проєкт тієї самої студії, перейшов до повноцінного тривимірного рушія, відмовившись від обмежень двовимірних карт, і ввів у широкий ужиток поняття рух-на-мишці, що й донині залишається стандартом керування у жанрі.

Наприкінці 1990-х і на початку 2000-х років жанр збагатився новими піджанровими напрямами. Half-Life (1998) від Valve продемонстрував, як FPS може поєднувати екшн із наративом без переривання ігрового процесу катсценами. Counter-Strike (2000) увів у масову свідомість тактичний аспект командних шутерів, де ключовими стали не рефлекси, а координація та стратегія. Halo: Combat Evolved (2001) переосмислив жанр для консолей і показав, що FPS може органічно існувати за межами ПК. Серія Battlefield (2002–до сьогодні)

розвинула поняття масштабних багатокористувацьких битв із транспортними засобами та класовою системою. Детальніше про ключові етапи розвитку жанру наведено в таблиці (Таб. 1.1):

Таблиця 1.1: Ключові етапи розвитку жанру First Person Shooter (1973–2024)

Рік	Назва / Розробник	Платформа	Ключова інновація жанру
1973	Maze War / NASA Ames	Робоча станція Imlac	Перше переміщення від першої особи в 3D-лабіринті, концепція deathmatch
1992	Wolfenstein 3D / id Software	DOS, PC	Комерційний FPS зі швидкісним рейкастингом, стелс-елементи, шаблон боїв
1993	DOOM / id Software	DOS, PC	Псевдо-3D BSP-рушій, мережевий deathmatch, модифікаційна спільнота (WAD)
1996	Quake / id Software	DOS, Windows	Перший повноцінний 3D-рушій, QuakeWorld-мережа, 180° огляд мишею
1998	Half-Life / Valve	Windows	Безперервний наратив без катсцен, AI NPC-союзники, скриптові події
2000	Counter-Strike / Valve	Windows	Тактичний командний FPS, economy-система, кастомний мап-контент
2001	Halo: Combat Evolved / Bungie	Xbox	Повноцінний FPS-геймплей на консолях,
2004	Far Cry / Crytek	Windows	Відкритий світ у FPS, реалістична рослинність, CryEngine
2007	Team Fortress 2 / Valve	Windows, Xbox	Комерційне зародження жанру геройського шутеру

2009	Modern Warfare 2 / Infinity Ward	Multi	Killstreak-система, кастомні класи, масова популяризація жанру
2014	Overwatch / Blizzard Entertainment	Multi	Запровадження сучасного стандарту класичного геройського шутеру
2016	DOOM (rebooted) / id Software	Multi	Glory Kill, push-forward combat, arena-design, демонстрація Vulkan API
2020–н.ч.	Valorant, CS2, Apex Legends	PC	Ability-based агенти, live-service модель, esport-екосистема

Примітка: Таблицю складено на основі аналізу відкритих джерел та академічних публікацій.

Сучасний стан жанру відзначається його диверсифікацією та технологічним ускладненням. FPS-ігри сьогодні можна поділити на кілька основних напрямів: арена-шутери із симетричним геймплеєм (Quake Champions, Diabotical); тактичні шутери, де ключовою є командна взаємодія і висока ціна помилки (CS2, Valorant, Rainbow Six Siege); наративні сингплеєрні шутери з акцентом на сюжет і дослідження середовища (серія BioShock, Dishonored, Deathloop); battle royale – масові матчі із постійно звужуваним ігровим простором (Warzone, Apex Legends, PUBG); шутери-рогалики та лутер-шутери (Borderlands, Deep Rock Galactic). Кожен із цих піджанрів апелює до різних аудиторій, однак усі вони об'єднані спільним знаменником – перспективою від першої особи та центральним місцем стрілецьких механік у ігровому процесі.

З науково-дослідницької точки зору жанр FPS є предметом активного вивчення. Роботи у сфері ігрової психології та UX-дизайну встановлюють зв'язок між механічним дизайном FPS-ігор та психологічним станом гравця, зокрема феноменом потоку (flow) – стану повного занурення та концентрації, детально описаного Міхаєм Чиксентміхаї. Дослідження показують, що FPS-ігри є

особливо ефективними інструментами для введення гравця у стан потоку завдяки безперервному зворотному зв'язку, чіткій меті та поступовому нарощенню складності [13]. Це робить жанр цінним не лише як комерційний продукт, а й як об'єкт академічного аналізу в контексті людино-комп'ютерної взаємодії.

Аналіз того, яким чином FPS-ігри утримують гравців різного рівня навичок, свідчить, що ключовими факторами є: збалансована система нагороди, що дозволяє менш досвідченим гравцям відчути прогрес; варіативність ігрових ситуацій, яка запобігає одноманітності; та наявність глибоких механік, освоєння яких відкриває нові рівні майстерності для досвідчених гравців [1]. Ці принципи формують те, що дизайнери називають «легко навчитися, складно опанувати» – підхід, що є одним із стовпів успішного геймдизайну.

1.2 Ігрові рушії: класифікація, порівняльна характеристика

Ігровий рушій (game engine) є центральним інструментом сучасної ігрової розробки. З технічної точки зору рушій являє собою програмний фреймворк, що надає розробникам набір готових підсистем і забезпечує виконання типових ігрових завдань: рендеринг двовимірної та тривимірної графіки, симуляцію фізики, відтворення звуку, обробку введення, управління ресурсами та мережеву взаємодію. Використання рушія дозволяє абстрагуватись від низькорівневих деталей та зосередити зусилля безпосередньо на ігровій логіці й дизайні. Дослідження в галузі архітектури рушіїв підкреслюють, що правильне розуміння їхньої внутрішньої структури є необхідним для ефективного використання та оптимізації [16].

Загалом ігрові рушії класифікують за кількома ознаками. За архітектурним принципом виокремлюють рушії на основі компонентної системи сутностей (ECS – Entity Component System), що використовує такі рушії, як Unity та Bevy, та рушії на основі ієрархії сцен і вузлів, до яких належить Godot Engine. За цільовим призначенням розрізняють універсальні рушії загального призначення

(Unity, Unreal Engine, Godot) та спеціалізовані, орієнтовані на вузькі завдання (RPG Maker для рольових ігор, Ren'Py для візуальних новел). За моделлю ліцензування рушії поділяють на комерційні з роялті (Unreal Engine, що стягує 5% від доходу понад 1 млн доларів), комерційні з фіксованою підпискою (Unity), повністю безкоштовні з відкритим кодом (Godot) та безкоштовні для освіти.

На сьогодні, найпопулярнішими та найчастішими у використанні є три ігрові рушії: Unity, Unreal Engine та Godot Engine. Кожен із них має виражені переваги та обмеження, що зумовлюють доцільність їх застосування у різних сценаріях (Таб. 1.2). Порівняльний аналіз Unity та Godot у контексті навчальних проєктів показує, що Godot значно знижує поріг входу за рахунок простішого синтаксису GDScript та інтуїтивної структури проєкту [2].

Таблиця 1.2: Порівняльна характеристика ігрових рушіїв Unity 6, Unreal Engine 5 та Godot Engine 4 (станом на 2025 р.)

Критерій	Unity 6	Unreal Engine 5	Godot Engine 4
Ліцензія	Персональна безкоштовна / Plus від $185/міс (runtime fee скасовано у 2024)	Безкоштовна до $1 млн доходу; 5% роялті понад	MIT — повністю відкритий, без роялті та обмежень
Мова скриптингу	C# (.NET 8)	C++ / Blueprint (візуальний)	GDScript, C#, C++ (GDExt)
Поріг входу	Середній	Високий	Низький
Графічний API	DX11/12, Vulkan, Metal	DX12, Vulkan, Metal	Vulkan, DX12 (exp.), GL3/ES3
Рендерер для 3D	URP / HDRP	Lumen + Nanite	Forward+ / Mobile
Підтримка фізики	PhysX / Havok	Chaos Physics	Godot Physics / Jolt (4.3+)

Розмір інсталяції рушія	~2–4 ГБ	~60–80 ГБ	~70–120 МБ
Документація (якість)	Відмінна	Добра	Добра та зростає
Розмір спільноти (2025)	Найбільша	Велика	Середня, зростає
Ціна для комерц. релізу (інді)	0 (до $200K/рік)	0 (до $1M доходу)	0 (необмежено)

Примітка: Порівняльні дані складено на основі офіційної документації та порівняльних досліджень [2, 16].

Unity є найпоширенішим рушієм серед інді-розробників і малих студій завдяки широкій документації, великій спільноті та підтримці мобільних платформ. Проте у 2023 році компанія оголосила про зміну моделі ліцензування, що спричинило масовий відтік розробників та підвищений інтерес до альтернатив. Unreal Engine вирізняється кінематографічною якістю графіки та комплексним набором інструментів (зокрема, системою Blueprint для візуального програмування), однак пред'являє значно вищі вимоги до апаратного забезпечення та потребує глибоких знань від розробника. Порогом входу для Unreal є не лише технічна складність, а й необхідність роботи з масивними розмірами проєктів та тривалі процеси компіляції.

Godot Engine займає особливу нішу на ринку: будучи повністю безкоштовним відкритим рушієм під ліцензією MIT, він пропонує сучасний набір інструментів, достатньо потужних для розробки як інді-ігор, так і великих та комплексних проєктів. Знакова подія 2025 року – вихід Slay the Spire 2, розробленого на Godot після того, як студія відмовилась від двох років роботи на Unity, – зібрав понад 574 тисячі одночасних гравців у Steam та продемонстрував

комерційний потенціал рушія [21]. Це свідчить про те, що Godot вже вийшов за межі суто навчального інструменту і є повноцінною платформою для комерційної розробки.

З архітектурної точки зору усі три рушії реалізують конвеєр рендерингу реального часу, хоча використовують різні API. Vulkan став стандартом де-факто для сучасних рушіїв, тоді як DirectX 12 залишається прерогативою платформи Windows та Xbox. Godot Engine 4 перейшов на власний рендерер RenderingDevice, побудований поверх Vulkan, що забезпечило значний стрибок у продуктивності та якості графіки порівняно з третьою версією рушія. Сучасні графічні конвеєри передбачають кілька стадій обробки зображення: збирання геометрії, трансформацію вершин, растеризацію, виконання фрагментних шейдерів, пост-обробку – і кожна з них може бути налаштована або розширена засобами відповідного рушія [24].

1.3 Godot Engine як платформа для розробки тривимірних ігор

Godot Engine є відкритим ігровим рушієм, розробка якого розпочалась у 2007 році Хуаном Лінєцьким та Аріелем Манзуром як внутрішній інструмент їхньої аргентинської студії. У 2014 році рушій було опубліковано у відкритому доступі під ліцензією MIT, що ознаменувало початок його стрімкого поширення у спільноті незалежних розробників. Четверта основна версія рушія, Godot 4.0, вийшла у 2023 році і ознаменувала кардинальну переробку архітектури рендерингу, системи фізики та ряду базових підсистем. Станом на квітень 2026 року актуальною стабільною версією є Godot 4.6.2.

Принциповою архітектурною рисою Godot є ієрархія сцен та вузлів (Nodes). Кожна гра у Godot складається з дерева вузлів, де кожен вузол є самостійним об'єктом із власними властивостями, методами та сигналами. Сцени – це фрагменти цього дерева, збережені як окремі файли, що можуть бути повторно використані, успадковані та вкладені одна в одну. Такий підхід природно реалізує концепцію компонентної архітектури, де поведінка об'єктів

формується через їхнє місце у ієрархії та підключені сценарії. Вузлова архітектура Godot принципово відрізняється від монолітних компонентних систем Unity або ECS-підходу Unreal Engine, і ця відмінність суттєво впливає на спосіб мислення розробника [26].

Для тривимірних ігор Godot пропонує спеціалізований набір вузлів: CharacterBody3D для керованих персонажів із вбудованою підтримкою переміщення та виявлення зіткнень, RigidBody3D для фізично симульованих об'єктів, StaticBody3D для нерухомої геометрії, AnimationPlayer та AnimationTree для управління анімаціями. Система фізики в Godot 4 перероблена і використовує власний фізичний рушій Godot Physics, що підтримує дискретне та неперервне виявлення зіткнень. Крім того, рушій надає інструменти для запиту фізичної сцени: RayCast3D для трасування променів та ShapeCast3D для перевірки перекриттів об'ємних форм.

Мова програмування GDScript є ключовою частиною екосистеми Godot. Вона розроблялась із акцентом на простоту синтаксису та тісну інтеграцію з API рушія. GDScript 2.0, що з'явився у Godot 4, отримав типізацію, що перевіряється на етапі компіляції, власний асинхронний механізм через ключове слово await та покращене автодоповнення у редакторі. Завдяки Python-подібному синтаксису мова є доступною для початківців, водночас дозволяючи писати складні системи без зайвої синтаксичної громіздкості. Поряд із GDScript рушій підтримує C# та C++ через GDExtension, що відкриває можливості для перформансно-критичного коду.

Система сигналів у Godot реалізує патерн спостерігача (Observer) на рівні рушія. Кожен вузол може оголошувати сигнали – події, на які можуть підписатися інші вузли. Це забезпечує слабку зв'язність компонентів і є основним механізмом міжвузлової комунікації. Для тривимірних шутерів цей механізм особливо важливий: система зброї може випромінювати сигнал потрапляння, на який підписується компонент здоров'я цілі, – і жоден із них не знає безпосередньо про існування іншого.

Рендерер Godot 4+ базується на Vulkan і підтримує два режими: Forward+ (для потужних настільних систем) та Mobile (оптимізований для мобільних платформ та слабшого заліза). Forward+ підтримує кластерний рендеринг, що дозволяє ефективно обробляти велику кількість джерел світла без значних втрат продуктивності. Система матеріалів побудована на BaseMaterial3D, що підтримує PBR-рендеринг (фізично коректний рендеринг) та широкий набір параметрів поверхні. Дослідження архітектури рушіїв для рендерингу в реальному часі підтверджують, що використання Vulkan дозволяє досягати значно кращого розподілу навантаження між CPU та GPU завдяки явному управлінню пам'яттю та командними буферами [25].

Зростання популярності Godot відображає ширшу тенденцію у ігровій індустрії до вибору відкритих та незалежних інструментів. За даними GMTK Game Jam, частка проєктів на Godot зросла з 19% у 2024 до 40% у 2025 році [21]. Опитування серед інді-студій показують, що відкриті рушії скорочують цикл розробки на 20–30% завдяки простоті скриптингу та відсутності ліцензійних обмежень [12]. Ці числа є свідченням того, що Godot стає не просто освітнім інструментом, а виробничою платформою першого вибору для значної частини розробників.

1.4 Теоретичні засади занурення та ігрового потоку

Поняття занурення (immersion) є одним із центральних у теорії ігрового дизайну та ігровій психології. Під зануренням розуміється суб'єктивний стан гравця, при якому він відчуває себе «присутнім» у ігровому світі і сприймає події гри як значущі та реальні в контексті цього світу. Занурення не є бінарним явищем – це континуум, який залежить від безлічі факторів: якості графіки, переконливості ігрових систем, аудіального дизайну, наративних елементів та самого характеру взаємодії гравця з середовищем.

Дженіт та співавтори у своїй роботі 2008 року запропонували ієрархічну модель занурення з трьох рівнів: залученість (engagement), де гравець приділяє грі свою увагу; захоплення (engrossment), де гра починає домінувати у свідомості гравця; та повне занурення (total immersion), або присутність – стан, при якому

гравець «забуває» про реальний світ. Для шутерів від першої особи третій рівень є найбільш досяжним серед усіх ігрових жанрів саме завдяки перспективі від першої особи, яка мінімізує когнітивну дистанцію між гравцем та персонажем [9].

Суміжним і тісно пов'язаним поняттям є концепція потоку (flow), введена у психологічній науці Міхаєм Чиксентміхаї. Потік – це стан повної концентрації, при якому суб'єкт виконує завдання на межі своїх здібностей, відчуваючи внутрішню нагороду від самого процесу. У контексті відеоігор дослідження психофізіологічних реакцій гравців під час проходження FPS-ігор – включно з вимірюванням електроенцефалограми, електроміограми та шкірно-гальванічної реакції – демонструють чітку кореляцію між суб'єктивними оцінками занурення та об'єктивними показниками збудження нервової системи [13].

Для досягнення стану потоку у FPS-грі критичним є баланс між складністю завдань та навичками гравця. Якщо складність значно перевищує навички, гравець відчуває тривогу та фрустрацію. Якщо навички гравця значно переважають над складністю – виникає нудьга. Механіки, що дозволяють підтримувати цей баланс, включають адаптивну складність, систему прогресії, варіативність ситуацій та чіткий зворотний зв'язок на дії гравця. Аналіз стратегій утримання гравців у FPS-іграх показує, що найбільш ефективними є ті ігри, які надають відчуття прогресу та компетентності гравцям усіх рівнів майстерності [1].

З точки зору реалізації, занурення у FPS-грі формується численними технічними рішеннями. Вибір перспективи від першої особи є найбільш очевидним із них, однак не менш важливими є: наявність фізично достовірних анімацій зброї та рук персонажа; реакція камери на рух тіла (хитання голови при русі, нахил при повороті); звуковий дизайн, що відображає стан персонажа (дихання при бігу, ударний звук при отриманні шкоди); візуальні ефекти попадання та взаємодії зі середовищем. Сукупність цих дрібних деталей, які у дизайн-термінології об'єднуються під поняттям «ігровий сік» (game feel або

juice), має непропорційно великий вплив на сприйняття гри гравцем порівняно зі складністю їх реалізації.

Концепція «якості взаємодії» (Quality of Life, QOL) в ігровому контексті охоплює всі ті механіки та системи, що не є частиною основного ігрового циклу, але суттєво полегшують або збагачують взаємодію гравця з грою. У FPS-іграх до QOL-елементів відносять: «coyote time» – можливість пригати під час перших долей секунд знаходження персонажу в повітрі, плавне прискорення та уповільнення руху замість різкого старту та зупинки, індикатори попадань, що відображають напрямок загрози, систему підказок щодо наявності боєприпасів та зарядки зброї. Ці елементи знижують когнітивне навантаження на гравця і дозволяють йому зосередитись на виклику самої гри, а не на подоланні незручностей геймплею. Правильно реалізовані QOL-механіки є одним із найбільш потужних інструментів для підвищення рівня занурення без зміни базових правил гри.

Підсумовуючи теоретичний огляд, можна констатувати, що успішний прототип FPS-гри повинен реалізовувати три взаємопов'язані рівні досвіду: технічний (точне керування, достовірна фізика, коректна реєстрація пошкоджень), перцептивний (реакція камери, звукові та візуальні ефекти) та когнітивний (зрозумілі механіки, чіткий зворотний зв'язок між грою та гравцем). Саме з цих трьох рівнів складається той унікальний ігровий досвід, заради якому гравці повертаються до ігор жанру FPS знову і знову.

РОЗДІЛ 2 ПРОЄКТУВАННЯ ІГРОВОГО ПРОТОТИПУ

2.1 Вимоги до проєкту та концептуальна модель гри

Будь-яка розробка програмного продукту, у тому числі ігрового, розпочинається зі збору та формалізації вимог. У ігровій індустрії основним документом, що фіксує вимоги та концепцію, є дизайн-докумен (GDD) – всеохоплюючий проєктний документ, що функціонує для команди розробників умовним кресленням, даючи загальну ідею та напрямок розробки. Вміст ігрового дизайн-документу охоплює всі критичні елементи проєкту: механіки, сюжет, персонажів, візуальний стиль, рівні та досвід гравця [27]. Для поточного прототипу GDD виконує спрощену, але не менш важливу функцію: він фіксує межі реалізації, запобігаючи розмиванню обсягу проєкту.

Концептуальна модель прототипу визначається такими характеристиками. Жанр – тривимірний шутер від першої особи із одним гравцем. Ціль розробки – навчальний прототип, придатний для демонстрації базових FPS-механік і подальшого розширення. Ігровий простір – замкнений тестовий рівень, який включає різноманітні ландшафтні умови: відкриті простори, низькі чи замкнуті простори, похилий рельєф. Ігрові об'єкти – інтерактивні мішені з системою здоров'я, що реагують на попадання та руйнуються при вичерпанні очок міцності. Цільова платформа – персональний комп'ютер (Windows/Linux), що обумовлюється доступністю засобів розробки та цільової аудиторії.

Функціональні вимоги до прототипу можна систематизувати за підсистемами. Система керування персонажем повинна забезпечувати: переміщення у чотирьох напрямках відносно напрямку погляду; стрибок з урахуванням фізики гравітації; біг як альтернативний режим переміщення з підвищеною швидкістю та обмеженим витривалістю або без такого обмеження; присід зі зменшенням висоти колайдера персонажа; динамічний напівприсід у місцях із низькою стелею. Система зброї повинна підтримувати: постріл із точкою вильоту від місця розташування зброї; реєстрацію попадань методом

трасування променя; ефект відкату зброї при пострілі; скінченні боєприпаси, та систему перезарядки.

Нефункціональні вимоги охоплюють: підтримку частоти кадрів не нижче 60 FPS на апаратному забезпеченні середнього класу; час відгуку на дії гравця, що не перевищує одного кадру при стандартній частоті; модульність архітектури, що дозволяє відносно просто додавати нові типи зброї, ворогів чи ігрових механік; зрозумілість коду, документований GDScript із коментарями в критичних місцях.

Концептуальна модель ігрового процесу (core game loop) для поточного прототипу є лінійною та зосередженою на бойовому аспекті: гравець переміщується по рівню, знаходить та уражає цілі-мішені, отримує візуальне підтвердження попадань. Цикл не передбачає умов перемоги чи поразки, що відповідає статусу прототипу – наведені елементи є закономірним претендентом для розширення проєкту у майбутньому. Важливо зазначити, що відсутність явного ігрового циклу є усвідомленим рішенням, а не упущенням: в умовах обмеженого часу розробки концентрація зусиль на технічній якості базових механік є більш ціннішою, ніж поверхнева реалізація ширшого набору функцій.

2.2 Архітектурні рішення та патерн скінченного автомату станів

Вибір архітектурних патернів є, мабуть, найбільш визначальним рішенням у проєктуванні ігрових систем – помилки на цьому рівні надзвичайно дорого обходяться при подальшій розробці. Для системи керування персонажем у FPS-грі центральним архітектурним рішенням є використання скінченного автомату станів (Finite State Machine, FSM). FSM є одним із найстаріших та найбільш добре вивчених патернів у ігровій розробці: ще у класичних аркадних іграх, таких як PAC-MAN, поведінка персонажів описувалась через набір станів та умови переходів між ними [8].

Формально скінченний автомат станів описується п'ятіркою (S, Σ, δ, s_0, F), де S – скінченна множина станів; Σ – алфавіт вхідних символів (у ігровому контексті – події або умови); δ: S × Σ = S – функція переходів; s_0 – початковий стан; F – множина кінцевих станів. У практичній реалізації для ігрового персонажа кінцевих станів зазвичай не існує – автомат функціонує нескінченно протягом ігрової сесії, циклічно обходячи стани залежно від дій гравця та стану середовища (Рис. 2.1).

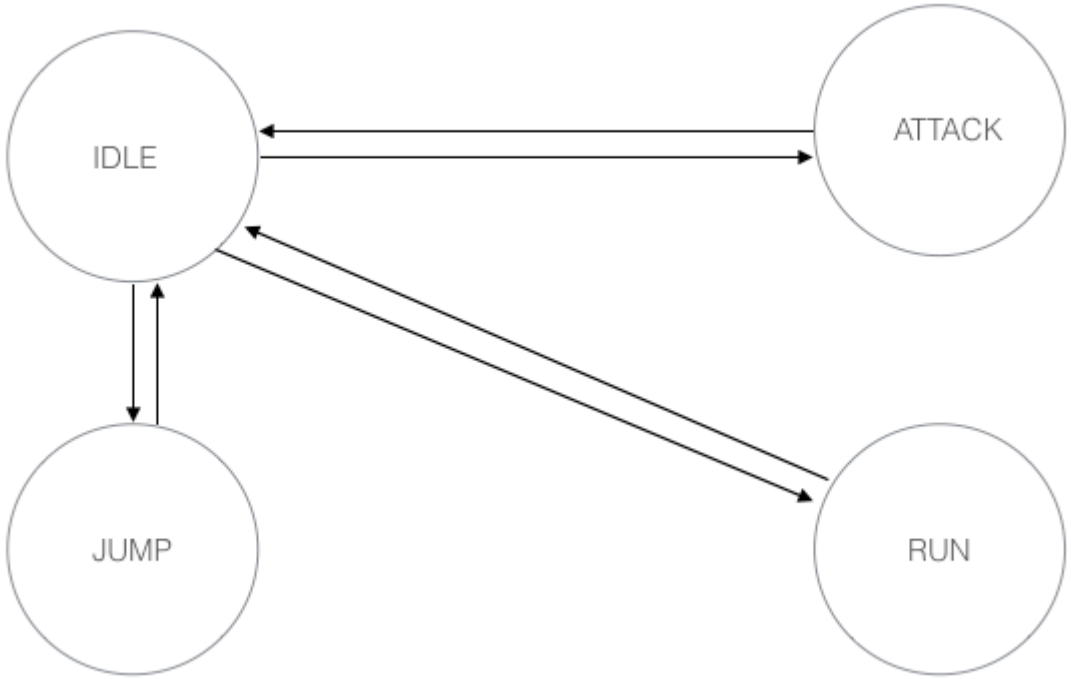

Рисунок 2.1: Схематичний приклад скінченного автомату станів для гри

Для персонажа FPS-гри визначаються такі стани: IDLE – персонаж нерухомий, відсутній жодний вхідний сигнал руху; WALK – персонаж переміщується з базовою швидкістю; SPRINT – персонаж переміщується з підвищеною швидкістю при затиснутій клавіші бігу; CROUCH – персонаж присів, висота колайдера зменшена, швидкість руху знижена; MIDAIR – персонаж перебуває у повітрі після стрибка або падіння. Умови переходів між станами визначаються комбінаціями вхідних подій (натискання клавіш) та станом середовища (наявність підлоги під ногами, наявність перешкоди над головою та інше).

Перевага FSM перед монолітним підходом (де вся логіка керування міститься в одній функції update) полягає у кількох аспектах. По-перше, кожен стан є ізольованою одиницею поведінки, що значно спрощує відлагодження – розробник завжди знає, у якому стані перебуває персонаж і чому. По-друге, додавання нових станів не вимагає модифікації існуючих. По-третє, переходи між станами є явними та задокументованими, а не прихованими в ланцюгах умовних операторів.

Ієрархічний автомат станів (HFSM – Hierarchical Finite State Machine) є розширенням базового FSM, де стани можуть містити підстати. Для поточного прототипу ієрархічний підхід застосовується опосередковано: стан CROUCH виведено в окремий контроллер, що накладається поверх існуючої машини станів, дозволяючи комбінувати режим присіду з іншими станами, без необхідності створення нових спеціалізованих станів, типу CROUCH_WALK чи CROUCH_SPRINT. Дослідження у галузі ігрового програмування відзначають, що HFSM є стандартом для складних персонажів у сучасних AAA-іграх, де дерево станів може налічувати десятки рівнів [8].

Окремою архітектурною проблемою є реалізація динамічного напівприсіду. Ця механіка виникає з вимоги, щоб персонаж не міг раптово збільшити свій розмір у замкненому просторі, що призводило б до «прокльовування» геометрії рівня. Рішення полягає у постійній перевірці наявності перешкод вище голови персонажа засобами ShapeCast3D, що проєктує об'ємну форму (капсулу повного зросту + безпечний відступ) вгору від поточної позиції персонажа. Якщо форма перетинає геометрію, перехід у вертикальне положення блокується, і персонаж залишається у стані CROUCH до тих пір, поки простір над головою не звільниться, разом з цим, розмір колайдера персонажа за певних умов динамічно змінюється, підлаштовуючись під актуальний вільний простір. Ця деталь є прикладом того, як технічна необхідність може бути реалізована у формі механіки, що одночасно вирішує технічну проблему та збагачує ігровий досвід.

Вузлова структура прототипу в Godot організована таким чином. Кореневим вузлом персонажа є CharacterBody3D, що надає вбудований метод move_and_slide() для переміщення з виявленням зіткнень. Дочірніми вузлами є: колізійна капсула, вузол камери, контроллери ефектів камери, контроллер поведінки персонажа, контроллер зброї, контейнер для візуалу та кілька інших допоміжних нод.

2.3 Проєктування системи зброї та реєстрації пошкоджень

Система зброї є, разом із системою керування персонажем, другим ключовим компонентом будь-якого FPS-прототипу. Її проєктування потребує вирішення кількох взаємопов'язаних проблем: звідки стріляє зброя; яким чином визначається точка попадання; як передається інформація про пошкодження цілі; яким чином реалізується відчуття правдоподібності поведінки зброї в руках персонажу.

Перша ключова проєктна проблема – точка вильоту пострілу. Існують два підходи. Перший – постріл із центру екрана: з центру камери вилітає об’єкт з колізією або трасується промінь у напрямку погляду персонажа. Цей підхід простий у реалізації та забезпечує передбачувану точність для гравця, однак є технічно некоректним: зброя, яку видно у нижній частині екрана, розташована не в центрі, і постріл від центру екрана не відповідає положенню ствола. Другий підхід – постріл від точки дула зброї – є технічно достовірнішим, але складнішим у реалізації, оскільки вимагає правильного обчислення напрямку ствола у просторі сцени. Для поточного прототипу обраний другий підхід як такий, що більш повно відповідає завданню створення фізично достовірної поведінки.

Реєстрація попадань реалізована методом трасування променя (Raycast). Raycast – це операція запиту до фізичного рушія, яка визначає першу точку перетину заданого променя з колайдерами у сцені [14]. При пострілі від точки дула зброї у напрямку, обчисленому з орієнтації прив'язки зброї, промінь трасується на максимальну відстань ефективного пострілу. Якщо промінь

перетинає колайдер, що має компонент здоров'я (HitBox або HealthComponent), цьому компоненту надсилається сигнал або викликається метод отримання пошкоджень. Якщо колайдер не має компонента здоров'я, реєструється попадання у статичний об'єкт середовища і відтворюється відповідний ефект (декаль удару, частинки).

Ефект інерції зброї (weapon lag) є одним із найбільш ефективних QOL-елементів для підвищення сприйняття «ваги» зброї. Принцип реалізації полягає у наступному: замість того, щоб орієнтація прив'язки зброї миттєво відповідала орієнтації камери, вона оновлюється з певною затримкою через лінійну інтерполяцію (lerp). При швидкому русі камерою між орієнтацією погляду та орієнтацією ствола виникає тимчасове розузгодження – зброя «відстає» від руху голови, що інтуїтивно читається як інерційна маса фізичного об'єкта. Ступінь ефекту регулюється коефіцієнтом інтерполяції: менше значення – більша інерція, більше – менша.

Відкат при пострілі (recoil) проєктується у двох вимірах: вертикальному та горизонтальному. Вертикальний відкат є передбачуваним та стабільним, що дозволяє досвідченому гравцеві компенсувати його. Горизонтальний відкат містить випадкову компоненту в межах заданого діапазону, що додає непередбачуваності. Реалізація відкату через безпосереднє обертання камери є технічно більш правильним підходом, ніж роздування кола розсіювання, оскільки гравець фізично бачить підняття прицілу і може свідомо компенсувати відкат мишею. Саме така модель відкату використовується у більшості сучасних тактичних шутерів.

Компонент здоров'я (HealthComponent) проєктується як незалежний вузол, що може бути доданий до будь-якого ігрового об'єкта. Він зберігає поточні та максимальні очки здоров'я і надає метод take_damage(amount: float), що зменшує поточне здоров'я та випромінює сигнали damaged(amount) та died(). Підписники можуть реагувати на ці сигнали: відтворювати анімацію отримання шкоди, змінювати колір матеріалу, відображати UI-індикатор здоров'я або видаляти об'єкт при загибелі.

2.4 Проєктування ігрового рівня та інтерактивних об'єктів

Тестовий рівень у контексті прототипу виконує подвійну функцію: з одного боку, він є середовищем для перевірки всіх реалізованих механік; з іншого – є демонстраційним простором, що дозволяє оцінити ігровий досвід та стан проєкту в цілому. Ефективний тестовий рівень повинен містити різноманітні геометричні умови: відкриті ділянки для тестування стрільби на дальній дистанції; вузькі простори для перевірки переміщення у стиснених умовах; ділянки з низькою стелею для тестування присіду; різниця висот для перевірки механіки стрибка та реакції на падіння. Проєктування рівня у Godot реалізується переважно через інструмент GridMap або через розміщення MeshInstance3D вузлів із відповідними колайдерами StaticBody3D та CollisionShape3D. GridMap є більш зручним для прямокутних рівнів у стилі арени, тоді як ручне розміщення меш-інстанцій надає більшу гнучкість у формуванні нестандартних геометричних форм. Для тестового рівня прийнятний спрощений підхід, з розміщенням у сцені об’єктів типу CSGBox3D, та вибудовою з них простого рівня.

Інтерактивні мішені є основними активними об'єктами тестового рівня. Кожна мішень є вузлом StaticBody3D або RigidBody3D з дочірніми вузлами: MeshInstance3D для візуального представлення, CollisionShape3D для реєстрації попадань та HealthComponent для обліку здоров'я. При отриманні пошкоджень мішень може змінювати колір або відтворювати анімацію, що надає гравцеві візуальний зворотний зв'язок. При вичерпанні здоров'я об'єкт видаляється зі сцени або переходить у «зламаний» стан із заміною меша. Розміщення мішеней на тестовому рівні може охоплювати декілька різних сценаріїв: статичні мішені на різних відстанях для перевірки дальності ефективного пострілу; мішені на різних висотах, включно із тими, що потребують пострілу догори або донизу; мішені за частковими укриттями для перевірки точності системи Raycast; мішені у вузьких коридорах для перевірки попадань у стиснених умовах.

Система освітлення тестового рівня виконує не лише візуальну, але й важливу функціональну роль. Правильно налаштоване освітлення допомагає гравцеві краще орієнтуватися в ігровому просторі, швидше знаходити ключові об'єкти та оцінювати відстань до цілей. Крім того, освітлення безпосередньо впливає на загальне сприйняття сцени, формуючи атмосферу та підкреслюючи особливості оточення. Ігровий рушій Godot Engine 4+ підтримує сучасну систему динамічного освітлення з використанням рендерера Forward+, що забезпечує коректну роботу джерел світла та тіней при збереженні високої продуктивності. Завдяки цьому навіть відносно прості сцени можуть виглядати значно реалістичніше без необхідності використання складних методів попереднього запікання освітлення. Для тестового рівня було прийнято рішення використати джерело світла типу DirectionalLight3D, яке імітує сонячне освітлення та забезпечує рівномірне освітлення всієї сцени. Такий підхід дозволяє створити природне освітлення відкритого простору та забезпечити чітке відображення об'єктів незалежно від їхнього розташування на рівні. Додатково було налаштовано генерацію тіней, що дозволило підвищити глибину сцени та покращити візуальне сприйняття навколишнього середовища.

Окрім основного джерела світла, для покращення зовнішнього вигляду рівня до списку об’єктів тестового рівня також ввійшла нода WorldEnvironment. Даний компонент відповідає за глобальні параметри середовища та дозволяє застосовувати різноманітні ефекти постобробки зображення. Використання WorldEnvironment дає змогу значно підвищити якість візуалізації без внесення змін до окремих об'єктів сцени.

У межах роботи над тестовим рівнем планувалося створити простий фоновий простір із небом, який забезпечував би природне заповнення сцени та усував ефект порожнечі за межами ігрової зони. Для цього гарно підходить вбудована система Sky, що генерує процедурне небо та формує більш правдоподібне освітлення оточення.

Для підсилення атмосфери також можна фикористати ефект об'ємного туману (Volumetric Fog). На відміну від звичайного туману, який лише приховує

віддалені об'єкти, об'ємний туман взаємодіє зі світлом та створює помітні світлові промені й м'які переходи між освітленими та затіненими ділянками сцени. Це дозволяє зробити середовище більш глибоким та візуально насиченим.

Додатково, для покращення візуального досвіду також можна використати ефекти SSAO (Screen Space Ambient Occlusion) та SSIL (Screen Space Indirect Lighting). Технологія SSAO відповідає за формування м'яких контактних тіней у місцях стику поверхонь та дрібних деталей геометрії. Завдяки цьому об'єкти виглядають менш пласкими та краще інтегруються в навколишній простір. Ефект SSIL, своєю чергою, моделює непряме відбиття світла від поверхонь, додаючи сцені більш природне освітлення та покращуючи передачу кольорів між об'єктами [28].

Сукупне використання DirectionalLight3D, системи WorldEnvironment, процедурного неба, об'ємного туману та ефектів SSAO і SSIL дозволяє створити візуально привабливий тестовий рівень з достатнім рівнем деталізації та реалістичності, забезпечуючи комфортне сприйняття ігрового простору, покращуючи навігацію гравця та водночас демонструє можливості сучасної системи рендерингу Godot Engine 4.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ІГРОВОГО ПРОЄКТУ НА РУШІЇ GODOT ENGINE

3.1 Налаштування середовища розробки та структура проєкту

Середовище розробки для поточного проєкту базується на Godot Engine версії 4.6+ Рушій доступний для завантаження як standalone-виконуваний файл без необхідності інсталяції, що є ще однією перевагою Godot порівняно з конкурентами, які потребують окремих пакетних менеджерів або великих інсталяторів. Приємним бонусом являється ще також те, що рушій можна безкоштовно завантажити через платформу Steam. Для розробки GDScript-коду достатньо вбудованого редактора всередині рушія, однак при бажанні можна використовувати зовнішній редактор, типу Visual Studio Code із розширенням godot-tools, що надає автодоповнення та підсвічування синтаксису.

Структура проєкту у Godot є файловою: усі ресурси – сцени (.tscn), сценарії (.gd), меш-файли (.glb/.obj), текстури та звуки – зберігаються у файловій системі і відображаються у панелі FileSystem редактора. Рекомендованою практикою є організація ресурсів за функціональними папками: /scenes для ігрових сцен, /scripts для скриптів GDScript, /assets для мешів та текстур, /audio для звукових файлів (Рис. 3.1).

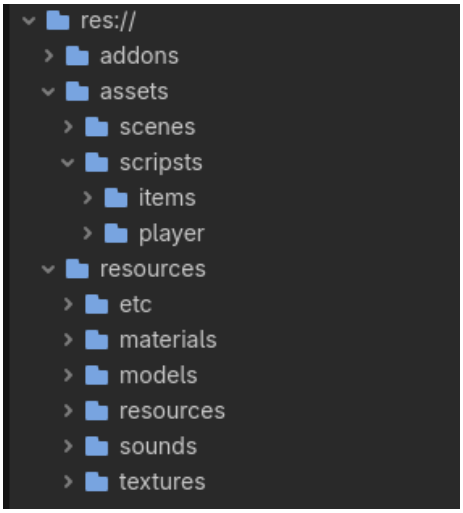

Рисунок 3.1: Зображення ієрархії директорій проєкту

Першим важливим кроком в розробці є налаштування проєкту. Для FPS-прототипу ключовими налаштуваннями є: прив'язка дій вводу (Input Map) та створення головної сцени. Input Map визначає дії що може виконувати персонаж, відповідно натиснутих клавіш, для проєкту було сторено мапу вводу з діями типу Forward, Backwards, Left та Right для безпосереднього руху, Jump, Sprint та Crouch для прижку, бігу та присідання відповідно, Use, Aim, Reload та Interact для взаємодії зі зброєю та ігровими предметами (Рис. 3.2).

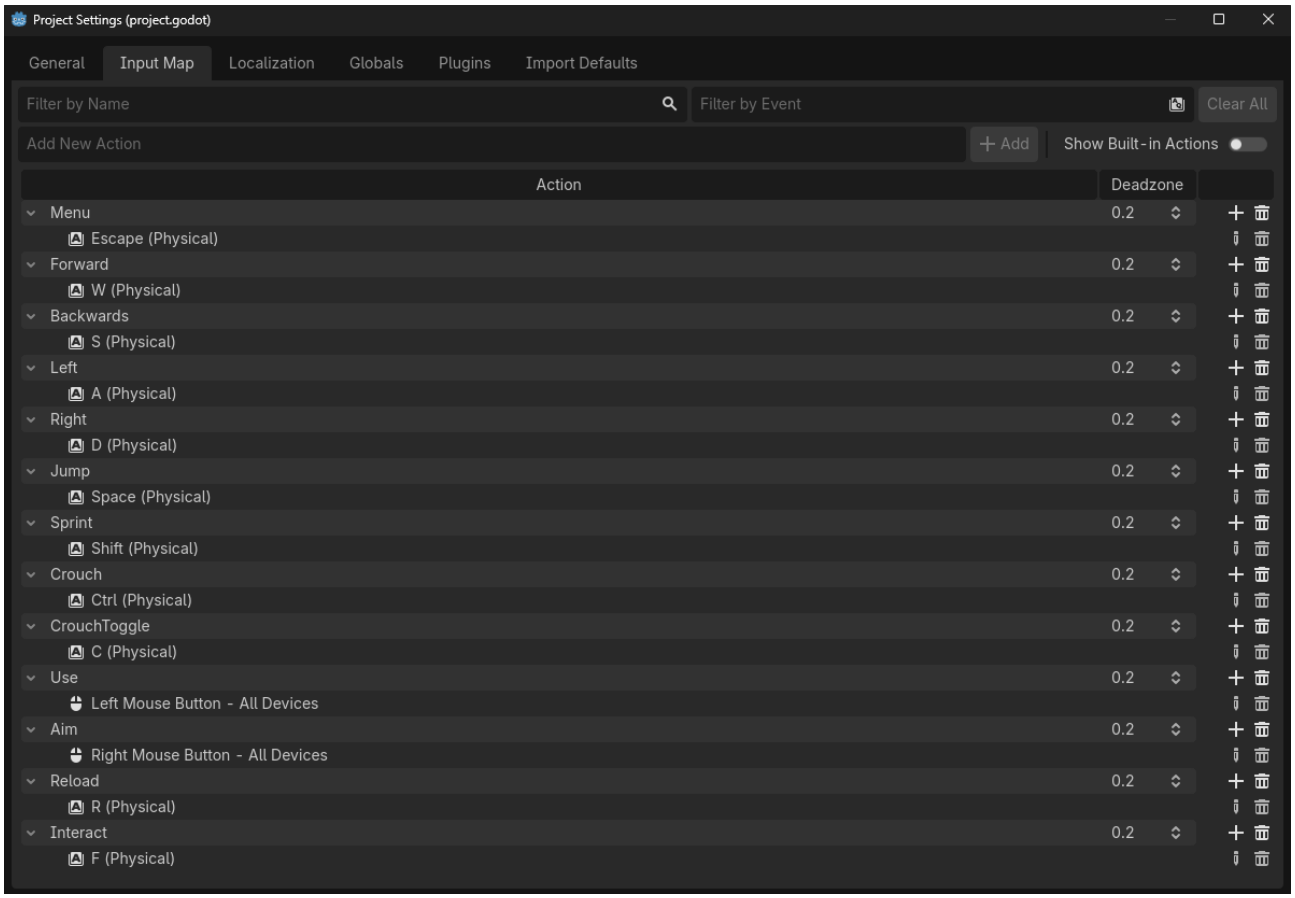

Рисунок 3.2: Input Map проєкту

Наступним кроком стала конфігурація фізики, поскільки Godot налічує декілька фізичних рушіїв, кожен з яких має свої особливості при роботі з фізичними об’єктами. Для проєкту було вирішено використовувати модель Jolt Physics, що видає кращі показники швидкодії в порівнянні зі стандартними налаштуваннями [29].

За цим була створена перша основна сцена, в якій в подальшому розмістились об’єкти тестового рівня та сам іговий персонаж. На початку персонаж представляв із себе максимально просту конструкцію з базових

необхідних компонентів, типу колізії та камери, та нод кнтейнерів, вміщаючих в собі реалізовані в подальшому контроллери персонажа, камери та зброї (Рис. 3.3).

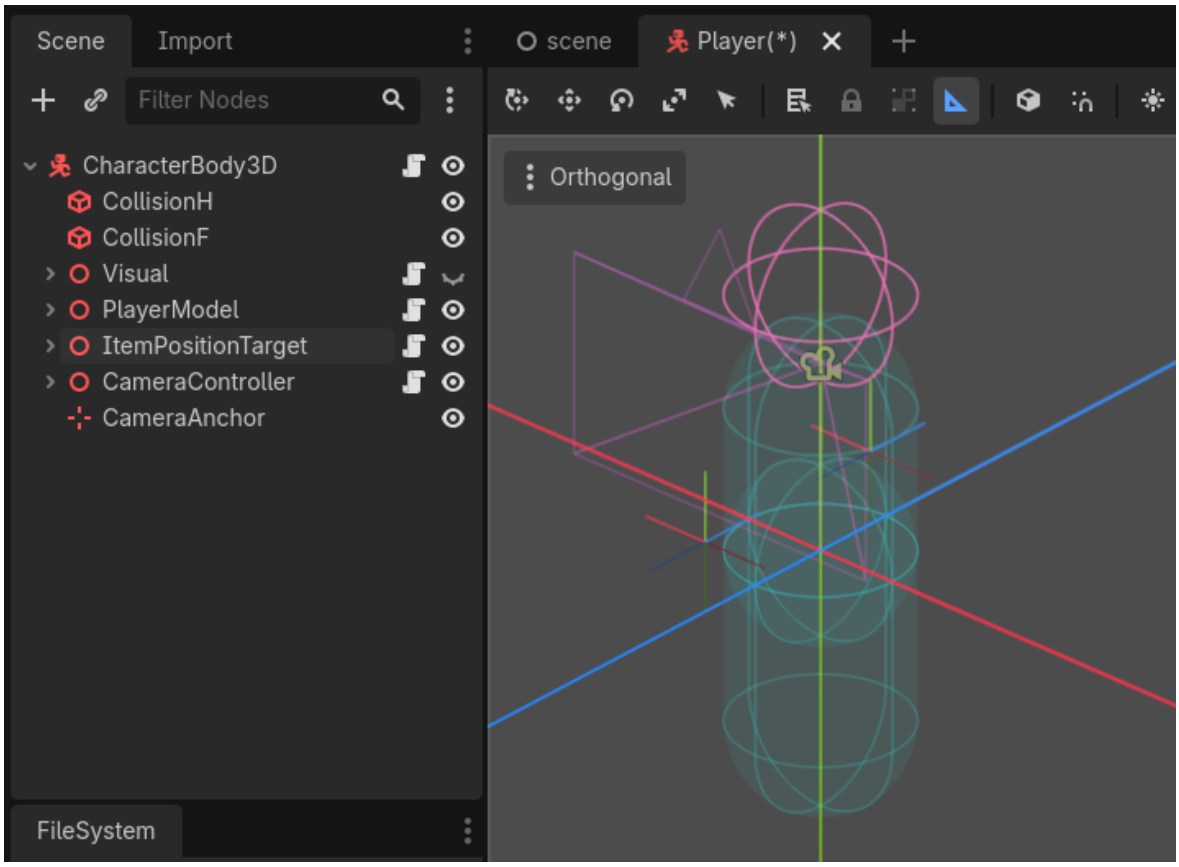

Рисунок 3.3: Вміст ігрового персонажу

3.2 Реалізація системи керування персонажем на базі FSM

Реалізація скінченного автомату станів у GDScript для системи керування персонажем може бути здійснена кількома способами: через enum та match-вираз, через окремі класи-стани або через вузлову ієрархію станів. У поточному прототипі застосований комбінований принцип: Створено основний FSM контроллер та загальний клас руху, на основі якого створено дочірні стани, кожен з яких має унікальний функціонал, ім’я та прив’язку до унікального вузла (Рис. 3.4). Такий підхід було використано для спрощення сприйняття системи та для легшого її розширення за потреби.

Рисунок 3.4: Ієрархія станів руху персонажа

Інформаційною базою для FSM контроллера є так званий InputPackage, пакет данник в якому записується поточний список дій які намагається виконати персонаж. Список основується на вводі з клавіатури та миші, який в свою чергу сприймаються та інтерпритується контроллером вводу, що конвертує сирий ввід в назви дій, та додає їх в InputPackage. Також в цей пакет данних передається напрямок руху персонажу та координати положення камери (Рис. 3.5).

input_pacage.gd — Online Docs

```
1  extends Node
2  class_name InputPackage
3  var actions : Array[String]
4  var item_actions : Array[String]
5  var input_direction : Vector2
6  var crouch : bool
7  var crouch_just_pressed : bool
8  var crouch_just_released : bool
9  var camera_basis : Basis = Basis()
```

Рисунок 3.5: Вміст InputPackage

Всередині FSM контроллера розписаний список рухів персонажа з прив’язкою їх до конкретних об’єктів сцени, тоді як базовий клас руху персонажа містить enum «moves_priority» що визначає приорітет виконуємих дій. Таке розділення потрібне для правильного сортування дій між собою, поскільки рух персонажа може бути непередбачуваним. Поточний стан зберігається у змінній current_move. За перехід між станами відпофідають створені функції update() та switch_to(), які порівнюють дії в списку, обирають самий значимий стан, та автоматично його активують (Рис. 3.6).

```
23  func update(input : InputPackage, delta : float):
24      var relevance = current_move.check_relevance(input)
25      if relevance != "okay":
26          switch_to(relevance)
27          print("Actions: ", input.actions)
28      mmc.check_modes(input)
29      current_move.update(input, delta)
30
31  func switch_to(state : String):
32      current_move.on_exit_state()
33      current_move = moves[state]
34      current_move.on_enter_state()
35      current_move.mark_enter_state()
36      print("State: ", state)
```

Рисунок 3.6: Функції update() та switch_to()

Метод update() викликається кожен фізичний кадр і є основним місцем обробки логіки стану: для станів руху, до прикладу, він викликає кастомний метод velocity_by_input() який зчитує вхідний вектор руху, застосовує гравітацію якщо персонаж у повітрі, обчислює бажану швидкість відповідно до поточного стану та викликає move_and_slide(), оновлюючи позицію персонажа в просторі. При кожному виклику, метод перевіряє умови переходів у порядку пріоритетності: найвищій приорітет мають стани прижку та падіння, маючи змогу перервати інші стани рухів, за ними йдуть більшість станів клавіш введення. Якщо поточний стан відрізняється від обчисленого нового стану, викликається метод switch_to(state), що виконує дії, пов'язані із входом у стан, типу on_exit_state() та on_enter_state() (Рис. 3.6).

Динамічний напівприсід є технічно найскладнішою частиною FSM. Для спрощення реалізації вся логіка присідання була виведена в окремий контроллер MMC (Movement Mode Controller), в задачу якого входить моніторинг та модифікація станів персонажа. Така реалізація значно спростила імплементацію присідання в стани руху, оскільки в іншому випадку було б потрібно створювати копії існуючих станів з ідмінними значеннями, типу WALK_ CROUCH, що суттєво збільшило б кількість станів та поскладнило б логіку переходів. Разом з цим відокремлення присідання дозволило реалізувати більш цікавий механізм самого присіду як такого, та виправити декілька непередбачених багів. Суть

помилок заключалась в просочуванні персонажа через об’єкти карти, при спробі встати з присіду під стелею, для виправлення проблеми була додана нода ShapeCast3D що трасує форму відповідну колайдеру персонажа у вертикальному положенні, а ціль трасування – точка безпосередньо над головою персонажа. Якщо ShapeCast3D.is_colliding() повертає true при спробі переходу зі стану CROUCH у STAND, стан змінюється на PARTIAL замість вертикального. Висота колайдера у стані PARTIAL є проміжною між висотою повного зросту та присіду і обчислюється на основі максимального вільного простору, доступного у поточній позиції (Рис. 3.7).

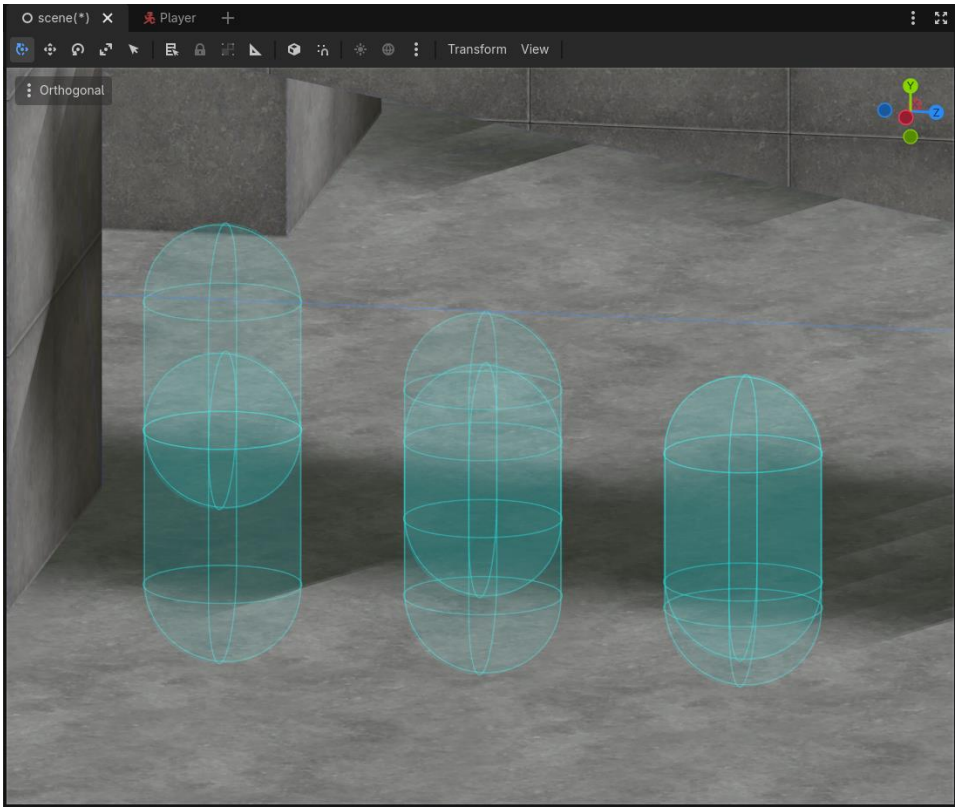

Рисунок 3.7: Візуалізація колайдера персонажа в різних стадіях присіду

Плавна зміна висоти колайдера є важливою деталлю реалізації. Замість миттєвої зміни значення height у CollisionShape3D, висота інтерполюється кожен кадр за допомогою lerp(). Аналогічно інтерполюється зміщення вузла камери по вертикалі, що забезпечує плавне зниження точки огляду при присіданні. Без цієї інтерполяції візуальний перехід виглядав би різким і руйнував би занурення.

Обробка стрибка у рамках FSM заслуговує окремого розгляду. Стрибок ініціюється лише якщо is_on_floor() є true та не активований стан CROUCH (або

PARTIAL за умови вільного простору над головою). При ініціалізації стрибка поточний стан переводиться у JUMP, а вертикальна компонента вектора швидкості встановлюється рівною JUMP_FORCE. По проходженню деякого короткого проміжку часу, стан прижку змінюється станом FALLING, після чого персонаж починає падати, допоки не приземлиться на підлогу (Рис. 3.8). Перехід зі стану FALLING назад у будь-який наземний стан відбувається при наступному виклику is_on_floor() == true, що і визначає приземлення.

```
11 func on_enter_state() -> void:
12     mmc.is_jumping      = true
13     player.velocity.y   = JUMP_FORCE
14
15 func on_exit_state() -> void:
16     mmc.is_jumping = false
17
18 func default_lifecycle(_input: InputPackage) -> String:
19     if works_longer_than(MAX_JUMP_TIME) and player.velocity.y <= 0:
20         return "fall"
21     return "okay"
22
23 func update(_input: InputPackage, delta: float) -> void:
24     player.velocity.y -= gravity * delta
25     player.move_and_slide()
```

Рисунок 3.8: Частина коду стану «Jump»

3.3 Реалізація системи зброї та фізичних ефектів

На відміну від системи керування персонажем, що базується на скінченному автоматі станів із фіксованим набором станів та механізмом пріоритетів, система зброї реалізована через окрему FSM зі значно спрощеною архітектурою. Таке рішення обумовлене характером поведінки зброї: послідовність її дій є значно більш передбачуваною та лінійною, ніж рух персонажа, тому використання складної системи пріоритетів було б надлишковим. Якщо під час керування персонажем одночасно можуть виникати десятки умов, які впливають на вибір поточного стану (біг, стрибок, присідання, перебування в повітрі тощо), то для зброї кількість можливих сценаріїв значно менша. У більшості випадків зброя перебуває у стані очікування, стрільби або

перезарядки, а переходи між ними відбуваються за чітко визначеними правилами.

Ключовою особливістю FSM зброї є її динамічна структура. Якщо у FSM персонажа перелік станів визначається під час розробки та залишається незмінним, то для зброї набір станів формується безпосередньо під час завантаження конкретного екземпляра зброї. Всередині FSM використовується словник станів, який початково є порожнім. Після екіпірування зброї він автоматично заповнюється станами, визначеними у відповідному ресурсі зброї. Такий підхід дозволяє кожному виду зброї мати власний набір станів, їх кількість та найменування без необхідності змінювати код самої FSM.

Подібне рішення забезпечує високий рівень масштабованості системи. Наприклад, для простої вогнепальної зброї може бути достатньо лише трьох базових станів, тоді як для більш складних зразків зброї можливо додати додаткові стани, такі як зміна режиму вогню, альтернативні типи стрільби, перегрів, заряджання по одному патрону або спеціальні анімаційні стани. При цьому базова реалізація FSM залишається незмінною, а всі особливості поведінки визначаються лише даними конкретного ресурсу зброї.

На відміну від FSM персонажа, де вибір наступного стану здійснюється шляхом перевірки списку можливих станів відповідно до їхнього пріоритету, у FSM зброї логіка переходів визначається безпосередньо всередині станів. Кожен стан самостійно вирішує, коли і до якого іншого стану необхідно переключитись, та за яких умов. Це спрощує реалізацію та підвищує гнучкість системи, оскільки різні види зброї можуть реалізовувати власну логіку роботи без впливу на інші компоненти. Крім того, такий підхід покращує читабельність коду, оскільки вся логіка, пов'язана з конкретною дією зброї, зосереджується в одному місці.

Для тестування системи було створено ресурс пістолета. Ресурс містить усі основні характеристики зброї: назву, тривимірну модель, максимальну кількість набоїв у магазині, величину завданої шкоди та список доступних станів. Фактично ресурс виступає контейнером даних, який описує вміст та поведінку

конкретного екземпляра зброї чи предмета, та може бути легко замінений іншим ресурсом без зміни основної логіки системи (рис. 3.9)..

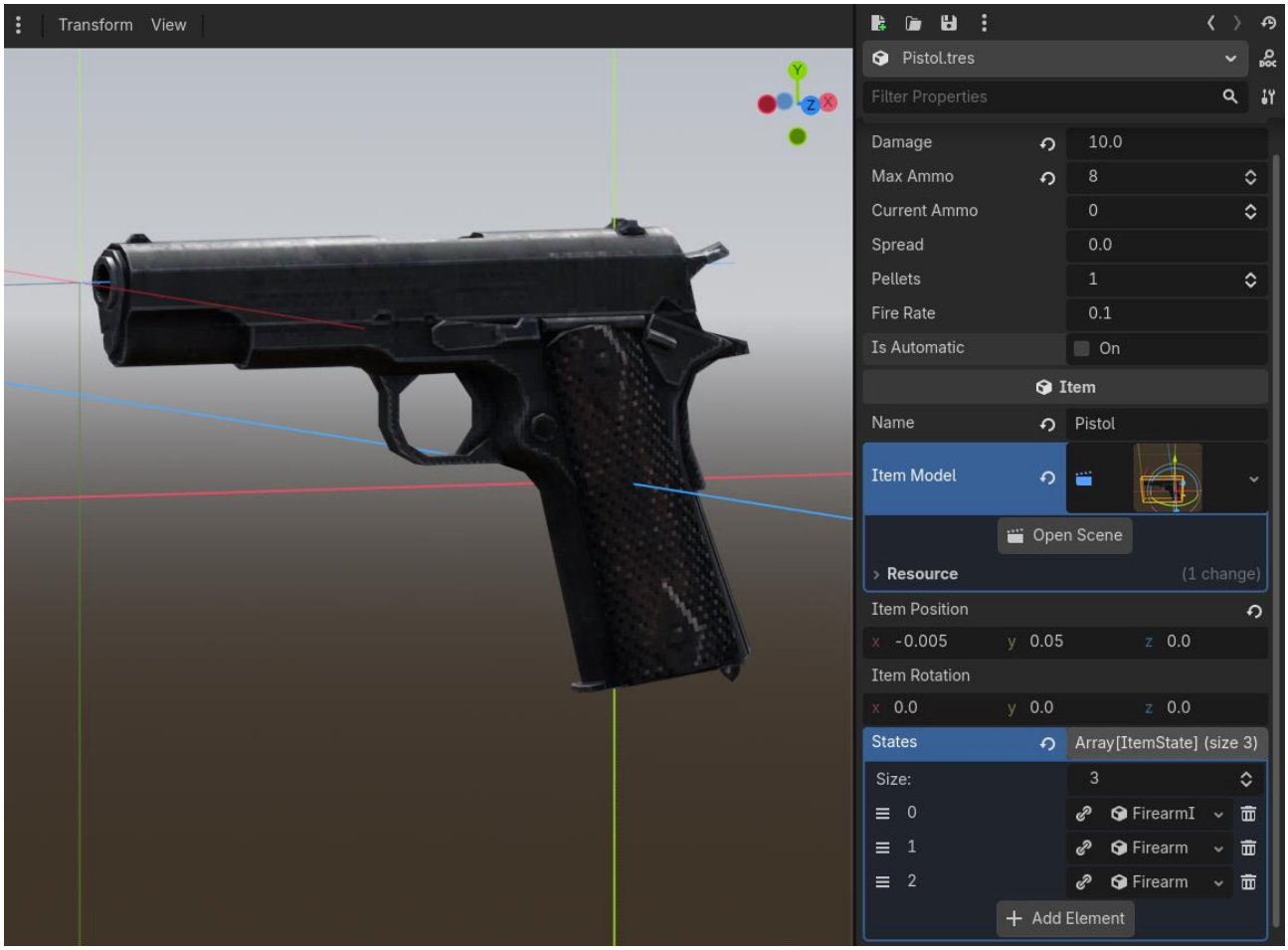

Рисунок 3.9: Ресурс зброї створеної для тестування механік стрільби

Для пістолета було реалізовано три базові стани: FirearmIdle, FirearmFire та FirearmReload. Цього набору достатньо для реалізації повного циклу роботи простої вогнепальної зброї. Кожен з станів описує окремий патерн поведінки, зі своєю логікою перемикання між іншими станами. Разом ці стани утворюють завершену модель поведінки зброї та демонструють можливості розробленої системи динамічної FSM.

Стан FirearmIdle є базовим станом спокою, в якому зброя перебуває більшу частину часу. У цьому стані визначено два можливі переходи. Якщо гравець натискає кнопку використання (ліва кнопка миші) та за умови, що в магазині присутні патрони, відбувається перехід до стану стрільби. Якщо ж боєприпаси відсутні, система виводить повідомлення про необхідність перезарядки. Другий можливий перехід пов'язаний із натисканням клавіші перезарядки. Якщо

поточна кількість патронів у магазині менша за максимально допустиму, активується стан перезарядки.

Стан FirearmFire відповідає за виконання пострілу. Під час його активації з магазину віднімається один патрон, після чого виконується перевірка попадання за допомогою трасування променя (Raycast). Для цього всередині моделі зброї розміщується спеціальний вузол-маркер Muzzle, який визначає точку вильоту кулі. Його глобальна позиція використовується як початок променя, а напрямок обчислюється на основі орієнтації ствола зброї. За допомогою фізичної системи рушія Godot виконується пошук першого об'єкта, з яким перетинається промінь. У разі успішного попадання у точці перетину створюється простий візуальний ефект у вигляді червоного куба (Рис. 3.10).

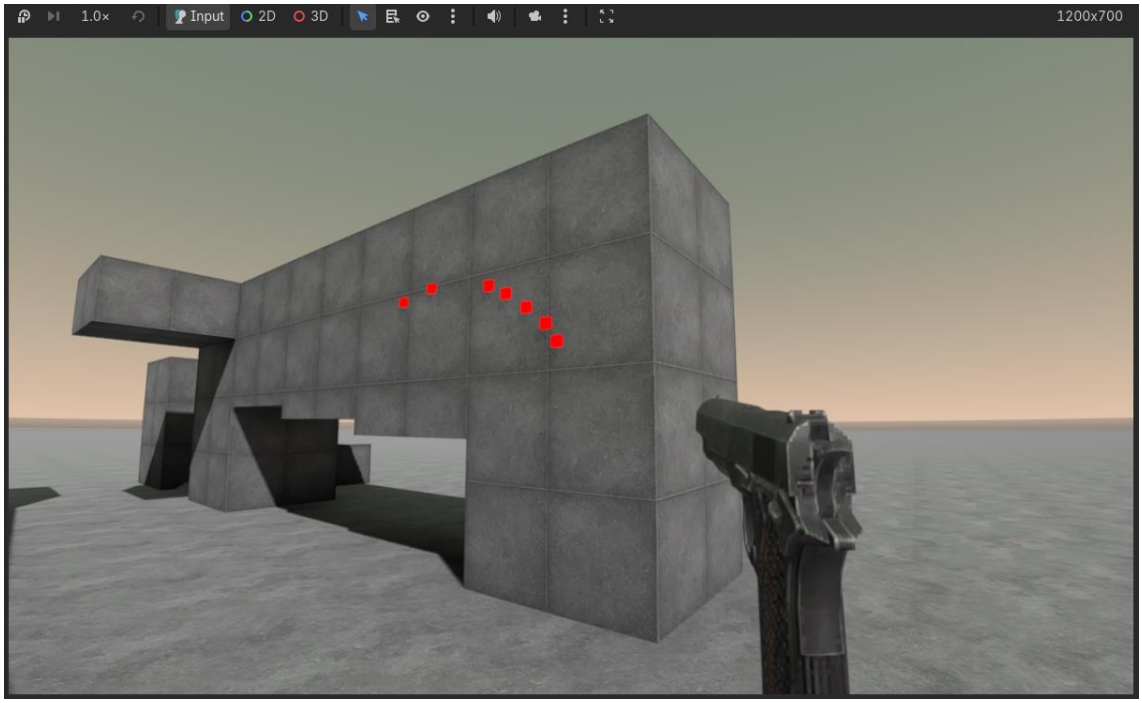

Рисунок 3.10: Демонстрація працездатності механік стрільби

Незважаючи на примітивність реалізації, такий підхід значно спрощує тестування механіки стрільби, оскільки дозволяє візуально контролювати точність трасування променя та коректність визначення місця попадання.

Стан FirearmReload відповідає за перезарядку зброї. Його логіка враховує особливості роботи вогнепальної зброї. Якщо в магазині залишається хоча б один патрон, після перезарядки магазин заповнюється до максимальної місткості. Якщо ж магазин був повністю порожнім, він заряджається на один

патрон менше від максимального значення. Така поведінка імітує реальну конструкцію більшості сучасної стрілецької зброї, де при частковій перезарядці один додатковий патрон може залишатися в патроннику (Рис. 3.11).

```
3  func on_enter() -> void:
4      print("Reloading")
5      var item = controller.current_item
6
7      if item.current_ammo == 0:
8          item.current_ammo = item.max_ammo - 1
9      else:
10         item.current_ammo = item.max_ammo
11
12     #controller.play_reload_effects()
13     controller.item_fsm.switch_to("idle")
```

Рисунок 3.11: Ланка коду для стану перезарядки зброї

Окремим елементом системи є реалізація ефекту weapon lag, що використовується для підвищення відчуття ваги та інерції зброї. Модель зброї розташовується всередині допоміжного вузла Node3D, який є дочірнім елементом камери гравця. Замість миттєвого повторення всіх поворотів камери цей вузол плавно інтерполює свою орієнтацію до цільового положення камери.

У кожному кадрі виконується обчислення нової орієнтації вузла зброї шляхом інтерполяції між його поточним положенням та положенням камери. Внаслідок цього зброя слідує за рухами гравця з невеликою затримкою, створюючи ефект інерційності. Інтенсивність запізнення регулюється окремим коефіцієнтом, що дозволяє легко налаштовувати поведінку різних видів зброї. Для легких пістолетів може використовуватися мінімальна затримка, тоді як важка зброя може мати більш виражений ефект інерції.

Запропонована архітектура дозволяє легко розширювати систему. Для створення нового виду зброї достатньо створити відповідний ресурс із необхідними характеристиками та набором станів. При цьому FSM автоматично підлаштовується під структуру нової зброї, що забезпечує високу масштабованість рішення та спрощує подальшу підтримку проєкту. Разом з цим розроблений контроллерзброї дозволяє реалізовувати не лише зброю, а й

інтерактивні предмети як такі: зброю ближнього бою, інструменти, розхідники, квестові предмети та предмети з унікальними властивостями та призначенням.

3.4 Динамічна камера та QOL-механіки

Динамічна камера є одним із ключових елементів, що впливають на комфорт сприйняття гри від першої особи. Навіть за наявності якісно реалізованого керування персонажем відсутність додаткових візуальних ефектів може робити пересування неприродним та «пласким». Саме тому в рамках прототипу було реалізовано набір допоміжних ефектів, що покращують візуальне сприйняття руху та створюють відчуття фізичної присутності персонажа в ігровому світі.

Архітектурно система розділена на два незалежні модулі: EffectController та FOVController. Таке розділення дозволяє ізолювати логіку різних візуальних ефектів та спрощує подальшу підтримку системи. Обидва контролери працюють незалежно один від одного та можуть бути відключені або замінені без впливу на роботу інших компонентів камери.

Модуль EffectController відповідає за локальні ефекти руху камери, а саме нахил камери під час руху в сторони (camera tilt) та коливання камери під час ходьби (head bob). Дані ефекти не впливають на ігрову механіку та виконують виключно візуальну функцію, покращуючи відчуття руху персонажа.

Ефект нахилу камери реалізований через аналіз локального напрямку руху персонажа. Якщо гравець рухається ліворуч або праворуч відносно свого поточного напрямку погляду, камера плавно нахиляється у відповідний бік навколо осі Z. Величина нахилу залежить від напрямку руху та інтерполюється до початкового положення після припинення бокового переміщення. Завдяки плавній інтерполяції рух камери виглядає природно та не створює дискомфорту для гравця (Рис. 3.12). Досить простий в реалізації, такий ефект робить пересування більш динамічним та додає відчуття інерції під час зміни напрямку руху.

```
39 func _update_tilt(delta: float) -> void:
40     if not mmc.is_walking:
41         vert.rotation.z = lerp(vert.rotation.z, 0.0, tilt_speed * delta)
42         return
43     var input_x = mmc.input_direction.x
44     var target_roll = -input_x * tilt_max_angle
45     vert.rotation.z = lerp(vert.rotation.z, target_roll, tilt_speed * delta)
```

Рисунок 3.12: Логіка функції нахилу камери при пересуванні

Другим ефектом, реалізованим у EffectController, є head bob – невелике періодичне коливання камери, що імітує природний рух голови людини під час ходьби або бігу. Реалізація базується на системі маркерів, які розміщуються у відповідних станах руху персонажа. Поки персонаж перебуває у станах ходьби або бігу, контролер отримує інформацію від цих маркерів та застосовує зміщення камери по вертикальній і горизонтальній осях. В інших станах, наприклад під час стояння, стрибка або падіння, ефект автоматично відключається.

Використання окремих маркерів дозволяє гнучко налаштовувати поведінку head bob для різних режимів пересування. Наприклад, для ходьби може використовуватися менша амплітуда коливань, тоді як для бігу – більша. Завдяки цьому візуально підкреслюється різниця між різними швидкостями руху персонажа.

За динамічну зміну кута огляду камери відповідає окремий модуль FOVController. Його основною задачею є зміна значення поля зору (Field of View, FOV) залежно від поточного режиму пересування персонажа. Для отримання інформації про зміну станів використовується система сигналів від контролера режимів руху Movement Mode Controller (MMC). Після отримання сигналу контролер визначає необхідне цільове значення FOV та запускає процес плавної інтерполяції. Для кожного режиму руху може бути визначене власне значення поля зору. Наприклад, під час бігу FOV дещо збільшується, створюючи додаткове відчуття швидкості. При переході у присід, навпаки, кут огляду стає трохи вужчим, що підкреслює більш обережний характер пересування. Додатково можуть використовуватися спеціальні значення для інших станів, зокрема для падіння (Рис. 3.13).

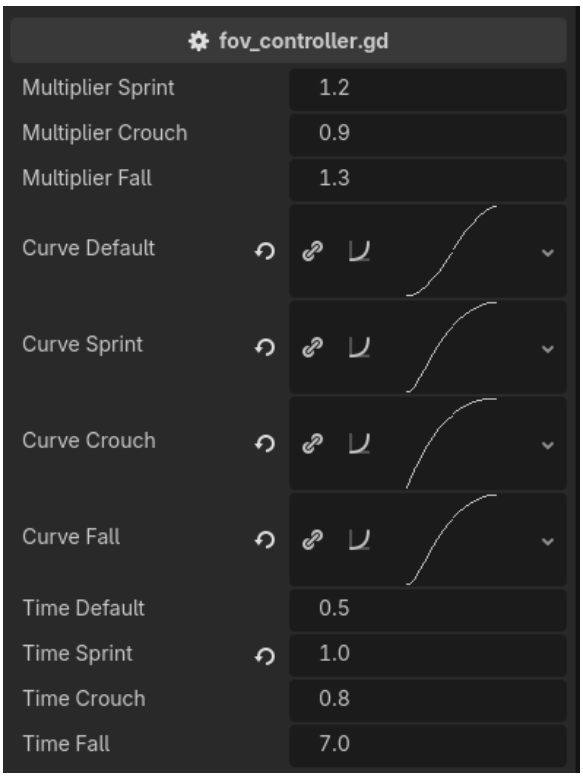

Рисунок 3.13: Налаштування мультиплікаторів та кривих для різних станів

Особливістю реалізації є використання окремих кривих інтерполяції для кожного режиму руху (Рис. 3.13). Це дозволяє не лише змінювати кінцеве значення FOV, а й контролювати характер переходу між станами. Наприклад, під час переходу з ходьби на біг зміна поля зору відбувається досить швидко, підкреслюючи різке прискорення персонажа. Водночас при падінні використовується більш високе значення часу зміни кута огляду, що створює враження поступового набору швидкостіз плином часу.

Поєднання EffectController та FOVController дозволяє створити комплексну систему динамічної камери, де кожен ефект виконує власну задачу. Нахил камери підкреслює напрямок руху, head bob додає відчуття кроків та контакту з поверхнею, а динамічна зміна FOV візуально передає зміну швидкості пересування. Разом ці механіки формують більш плавний та приємний користувацький досвід без внесення змін до основної ігрової логіки.

3.5 Тестування та результати

Тестування ігрового прототипу відрізняється від тестування традиційного програмного забезпечення у кількох аспектах. Поряд перформанс-тестуванням (чи досягається цільова частота кадрів) та із функціональним тестуванням (чи коректно працюють механіки), ігровий прототип потребує ігрового тестування (playtesting) – суб'єктивної оцінки того, чи є ігровий досвід переконливим та приємним. Усі три аспекти тестування є необхідними для повноцінної оцінки прототипу.

Перформанс-тестування проводилось шляхом вимірювання частоти кадрів за допомогою вбудованого монітора продуктивності Godot (Debug > Monitor) на апаратному забезпеченні тестової машини (Таб. 3.1).

Таблиця 3.1: Системні характиристики тестової машини

ОС:	Windows 11 Pro x64
Процесор:	Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz (2.40 GHz)
Графічна плата:	NVIDIA GeForce GTX 1660 Ti (6 ГБ)
ОЗП:	16,0 ГБ
Сховище:	1,61 ТВ

Ключовими метриками перформансу є: завантаження CPU та GPU у відсотках (Рис. 3.14); час рендерингу кадру (frame time) в мілісекундах; кількість draw calls за кадр; пікове споживання пам'яті (Рис. 3.15).

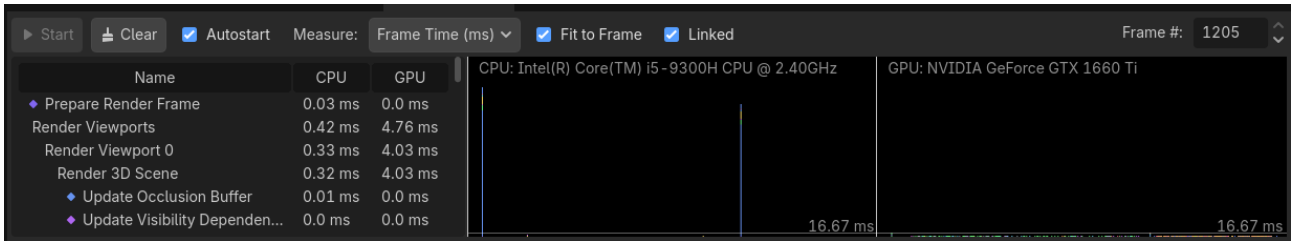

Рисунок 3.14: Перформанс тестування – вимірювання нагрузки на CPU та GPU

Рисунок 3.15: Перформанс тестування – вимірювання fps та розходу пам’яті

Функціональне тестування системи керування персонажем проводилось шляхом систематичного проходження всіх станів скінченного автомата (Finite State Machine, FSM) та перевірки коректності переходів між ними. Для кожного стану перевірялась правильність роботи логіки керування, відповідність анімацій поточному стану персонажа та відсутність конфліктів між окремими механіками.

Для проведення тестування розробленого прототипу було створено окремий тестовий рівень, призначений для перевірки всіх реалізованих ігрових механік. Рівень побудовано з використанням базових геометричних примітивів типу CSGBox3D, що дозволило швидко сформувати необхідні елементи оточення без створення складних 3D-моделей. На поверхні об'єктів були накладені прості матеріали з текстурою бетону для покращення візуального сприйняття середовища та полегшення орієнтації під час тестування (Рис. 3.16). Конфігурація рівня включала рівні майданчики, схили різного кута нахилу, вузькі проходи, тунелі з різною висотою стелі, вертикальні перешкоди та спеціально підготовлені ділянки для перевірки роботи системи стрільби.

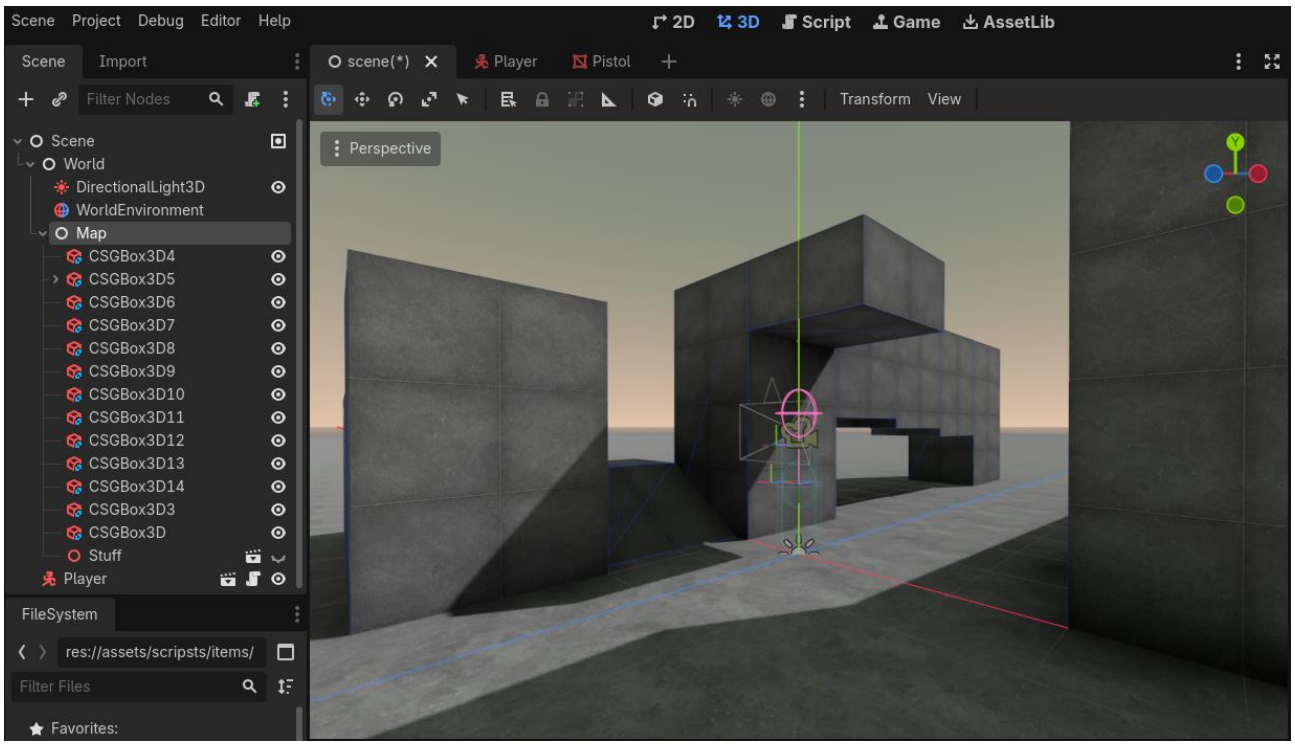

Рисунок 3.16: Створений всередині рушія тестовий рівень

Тестові сценарії охоплювали переміщення у всіх можливих напрямках із перевіркою відповідності анімацій руху поточному стану FSM, виконання стрибків як на рівній поверхні, так і під час руху по схилах різного нахилу, присідання у різноманітних геометричних умовах, включно з проходженням через тунелі різної висоти. Додатково перевірялися переходи між бігом та присіданням та робота механізму динамічного напівприсіду, який автоматично активується під час спроби персонажа піднятися під низькою стелею.

Особливу увагу було приділено перевірці крайніх випадків, які потенційно можуть спричиняти помилки в логіці FSM. До таких сценаріїв належали одночасне натискання клавіш стрибка та присідання, спроба виконати стрибок у стані PARTIAL, а також приземлення персонажа безпосередньо у стан присідання після стрибка. Перевірка цих випадків дозволила переконатися у стабільності роботи автомата станів та відсутності некоректних переходів між ними.

Функціональне тестування системи зброї було спрямоване на перевірку коректності роботи механізму Raycast та пов'язаних із ним підсистем. Зокрема, тестувалась правильність формування напрямку пострілу при різних

положеннях камери та зброї, коректність визначення точки зіткнення променя з об'єктами сцени, а також відображення візуальних ефектів попадання безпосередньо в точках контакту. Окремо перевірялась система обліку боєприпасів, включно з витрачанням патронів під час стрільби. Також проводились перевірки на предмет можливості виникнення неконсистентних станів, наприклад спроби стрільби під час перезарядки або перезарядки при повністю заповненому магазині.

Особливу увагу приділено тестуванню Raycast у складних геометричних ситуаціях. Було перевірено поведінку системи під час стрільби вздовж поверхні стіни під мінімальним кутом, пострілів через вузькі отвори між об'єктами та в точки перетину двох поверхонь. Такі сценарії дозволили оцінити точність реєстрації попадань і переконатися у відсутності помилок, пов'язаних із геометричними особливостями сцени.

Окрім формального функціонального тестування, проводилось playtesting, який є менш формалізованим, але надзвичайно важливим методом оцінювання якості ігрового прототипу. Під час кількох ігрових сесій оцінювались суб'єктивні відчуття від взаємодії з реалізованими механіками. Зокрема, аналізувалось, наскільки природним та чуйним сприймається керування персонажем, чи забезпечує система руху достатній рівень контролю над персонажем та чи не виникають труднощі під час виконання складних послідовностей дій.

Також оцінювалась якість реалізації стрільби та загальне відчуття від використання зброї. Перевірялось, чи є стрільба достатньо точною та передбачуваною, чи забезпечує вона необхідний рівень зворотного зв'язку для гравця, а також наскільки помітними та доречними є реалізовані ефекти покращення відчуття руху, зокрема weapon lag, head bob та camera tilt. Окремо аналізувалось, чи не викликають зазначені ефекти надмірного візуального навантаження або дискомфорту під час тривалої гри.

На основі проведеного тестування було виявлено та усунено кілька технічних проблем. Однією з найбільш суттєвих стала надмірна інтенсивність ефекту weapon lag у початковій реалізації. Через завелике значення коефіцієнта

згладжування зброя помітно відставала від рухів камери, що негативно впливало на процес прицілювання та ускладнювало точну стрільбу. Після серії експериментів параметри ефекту були відкалібровані до рівня, який забезпечує відчуття ваги зброї без втрати керованості.

Ще однією проблемою стала надмірна амплітуда та частота ефекту head bob. Під час тестування було встановлено, що інтенсивні коливання камери можуть викликати дискомфорт у частини користувачів, особливо під час тривалого переміщення персонажа. Для усунення цього недоліку параметри ефекту були скориговані шляхом зменшення амплітуди та частоти коливань.

Крім того, тестування виявило недоліки у початковій реалізації динамічного напівприсіду. Початкове положення камери в стані PARTIAL не забезпечувало достатньої видимості під час проходження через низькі тунелі та інші обмежені простори. Після аналізу результатів тестування було скориговано висоту розташування камери в цьому стані, що покращило оглядовість та загальний комфорт пересування.

Результати проведеної розробки та тестування свідчать про успішну реалізацію всіх запланованих механік. Створений прототип демонструє повністю функціональний FPS-контролер із коректно реалізованою FSM-системою керування персонажем, достовірною поведінкою зброї, надійною реєстрацією попадань за допомогою Raycast та набором додаткових QOL-механік, які суттєво покращують сприйняття ігрового процесу та рівень занурення гравця.

Створений тестовий рівень забезпечує достатню кількість сценаріїв для комплексної оцінки роботи всіх реалізованих підсистем та підтверджує їхню працездатність у різних умовах експлуатації. Отримані результати дозволяють зробити висновок, що розроблений прототип є технічно стабільною основою для подальшого розвитку проєкту та може бути використаний як фундамент для створення повноцінної гри жанру FPS.

ВИСНОВКИ

Результатом виконання кваліфікаційної роботи є розроблений функціональний прототип комп'ютерної гри жанру First Person Shooter на рушії Godot Engine 4, що реалізує повний набір ключових механік жанру та набір QOL-елементів для підвищення рівня ігрового занурення.

У теоретичній частині роботи систематизовано знання про жанр FPS: простежено еволюцію жанру від ранніх прикладів раннього тривимірного рендерингу до сучасних мультиплатформних тайтлів, проаналізовано піджанрову диверсифікацію та сучасний ринковий стан. Здійснено порівняльний аналіз ключових ігрових рушіїв – Unity, Unreal Engine та Godot Engine – за критеріями ліцензування, порогу входу, технічних можливостей та доцільності використання у навчальних і дослідницьких проєктах. Обґрунтовано вибір Godot Engine 4.6+ як платформи для реалізації прототипу. Розглянуто теоретичні засади ігрового занурення та концепцію потоку стосовно FPS-ігор, що сформувало теоретичне підґрунтя для проєктних рішень у частині QOL-механік.

У проєктувальній частині розроблено концептуальну модель прототипу та сформульовано функціональні та нефункціональні вимоги до системи. Спроєктовано архітектуру системи керування персонажем на основі скінченного автомату станів із сімома станами та розгалуженою системою умов переходів. Адаптовано патерн FSM для роботи з динамічною висотою стелі через ShapeCast3D запити – оригінальне технічне рішення, що вирішує проблему просторових обмежень без розширення набору базових станів. Спроєктовано систему зброї з реєстрацією попадань від точки дула та механізм передачі пошкоджень через слабозв'язаний компонент здоров'я.

У практичній частині реалізовано всі спроєктовані підсистеми засобами GDScript 2.0. Система керування персонажем коректно обробляє всі визначені стани та граничні умови переходів. Зброя реалізована з інерцією наведення на основі slerp-інтерполяції та відкатом камери при пострілі. Динамічна камера включає три незалежних ефекти: camera tilt, head bob та відкат. Тестовий рівень

містить інтерактивні мішені з компонентом здоров'я, що реагують на попадання та руйнуються. Проведено функціональне, перформанс- та ігрове тестування, на основі якого внесено ряд коригувань параметрів для покращення ігрового відчуття.

Практичне значення отриманих результатів полягає у створенні документованого, структурованого прототипу, що може використовуватись як навчальний матеріал у курсах ігрової розробки. Систематизований досвід застосування Godot Engine 4.6+ для розробки тривимірних шутерів – зокрема архітектурні рішення щодо FSM, реалізації Raycast від точки дула та QOL-механік камери – зафіксований у цій роботі і може бути відтворений та розширений іншими розробниками.

Напрями подальшого розвитку прототипу включають: реалізацію ворогів із AI на основі ієрархічного FSM або дерева поведінок; введення повноцінного ігрового циклу з умовами перемоги та поразки; розробку системи прогресії та дизайн повноцінних ігрових рівнів; реалізацію мультиплеєрного режиму засобами Multiplayer API Godot; розширення арсеналу зброї з унікальними механіками для кожного виду; впровадження адаптивної системи складності.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. An Analysis of How First-Person Shooting Games Retain Players of All Skill Levels. Proceedings of the 2nd International Conference on Social Psychology and Humanity Studies. 2024. URL: https://www.researchgate.net/publication/377603985 (дата звернення: 20.03.2026).

2. Alybaev A. Comparative Analysis of Unity and Godot for 2D Game Development. Preprints. 2025. URL: https://www.preprints.org/manuscript/202511.1981 (дата звернення: 22.03.2026).

3. Bradfield C. Godot 4 Game Development Projects: Build Five Cross-Platform 2D and 3D Games Using One of the Most Powerful Open Source Game Engines. 2nd ed. Packt Publishing, 2023. 452 p.

4. Godot Engine official documentation. ShapeCast3D. URL: https://docs.godotengine.org/en/stable/classes/class_shapecast3d.html (дата звернення: 25.03.2026).

5. Godot Engine official documentation. CharacterBody3D. URL: https://docs.godotengine.org/en/stable/classes/class_characterbody3d.html (дата звернення: 25.03.2026).

6. Godot Engine official documentation. RayCast3D. URL: https://docs.godotengine.org/en/stable/classes/class_raycast3d.html (дата звернення: 25.03.2026).

7. Grand View Research. Global Video Game Market Size & Outlook, 2025–2030. 2025. URL: https://www.grandviewresearch.com/industry-analysis/video-game-market (дата звернення: 10.04.2026).

8. Jagdale D. Finite State Machine in Game Development. International Journal of Advanced Research in Science, Communication and Technology. URL: https://www.ijarsct.co.in/Paper2062.pdf (дата звернення: 15.03.2026).

9. Jennett C. et al. Measuring and Defining the Experience of Immersion in Games. International Journal of Human-Computer Studies. 2008. Vol. 66, No. 9. P. 641–661.

10. Linietsky J., Manzur A. et al. Godot Engine. Version 4.6. Godot Foundation, 2024. URL: https://godotengine.org (дата звернення: 01.04.2026).

11. Mhatre T., Samel M., Chaudhari U., Banekar S. LostDune: A 2D Platformer Game Using Godot 4 Engine with Finite State Machine Architecture. IRE Journals. 2026. Vol. 9, No. 8. URL: https://doi.org/10.64388/IREV9I8-1714549 (дата звернення: 20.03.2026).
12. MoldStud Research Team. Future Trends in Game Design with Godot Engine Advantages. MoldStud, 2025. URL: https://moldstud.com/articles/p-the-future-of-game-design-why-you-should-start-using-godot-today (дата звернення: 05.04.2026).
13. Nacke L., Lindley C. A. Flow and Immersion in First-Person Shooters: Measuring the Player's Gameplay Experience. Proceedings of the 2008 Conference on Future Play. Toronto: ACM Press, 2008. P. 81–88.
14. Newcastle University. Physics – Raycasting Introduction. Game Technologies Masters Course Notes. URL: https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/physicstutorials/1raycasting/Physics%20-%20Raycasting.pdf (дата звернення: 18.03.2026).
15. On the Relevance of the Godot Engine in the Indie Game Development Industry. arXiv. 2024. URL: https://arxiv.org/pdf/2401.01909 (дата звернення: 12.04.2026).
16. Politowski C. et al. SyDRA: An Approach to Understand Game Engine Architecture. Entertainment Computing. 2025. Vol. 52. Art. 100832. URL: https://www.researchgate.net/publication/381307371_SyDRA_An_Approach_to_Understand_Game_Engine_Architecture (дата звернення: 13.04.2026)
17. Tatarchuk N. Advances in Real-Time Rendering in Games. SIGGRAPH 2025. URL: https://advances.realtimerendering.com/s2025/index.html (дата звернення: 15.04.2026).
18. Toward Real-Time Scalable Rigid-Body Simulation Using GPU-Optimized Collision Detection and Response. Mathematics. 2025. Vol. 13, No. 19. DOI: https://doi.org/10.3390/math13193230.
19. Ullmann G. C. et al. A Cross-Platform, WebGPU-Based 3D Engine for Real-Time Rendering and XR Applications. Proceedings of the 30th International Conference on 3D Web Technology. ACM, 2025. DOI: https://doi.org/10.1145/3746237.3746305.

20. Vanhove S. Learning GDScript by Developing a Game with Godot 4: A Fun Introduction to Programming in GDScript 2.0 and Game Development Using the Godot Engine. Packt Publishing, 2024. 378 p.
21. Ziva [pseudonym]. 5 Engineering Decisions That Made Godot the Fastest-Growing Game Engine. DEV Community. 2026. URL: https://dev.to/ziva/5-engineering-decisions-that-made-godot-the-fastest-growing-game-engine-5hgh (дата звернення: 14.04.2026).
22. Ziva [pseudonym]. Why Godot's Architecture Makes It the Best Engine for AI-Assisted Development. DEV Community. 2026. URL: https://dev.to/ziva/why-godots-architecture-makes-it-the-best-engine-for-ai-assisted-development-5e8f (дата звернення: 14.04.2026).
23. Flow and Immersion in First-Person Shooters: Measuring the Player's Gameplay Experience. Semantic Scholar. URL: https://www.semanticscholar.org/paper/Flow-and-immersion-in-first-person-shooters:-the-Nacke-Lindley/6407964757e4236ffbcf7789a541fe22c6b06031 (дата звернення: 16.03.2026).
24. Emerging Trends in Graphics Rendering Pipeline. International Journal of Computer Technology. 2025. URL: https://zenodo.org/records/17183463 (дата звернення: 20.03.2026).
25. Towards a Modern and Lightweight Rendering Engine for Dynamic Robotic Simulations. arXiv. 2024. URL: https://arxiv.org/html/2410.05095v1 (дата звернення: 20.03.2026).
26. Godot (game engine). Wikipedia. URL: https://en.wikipedia.org/wiki/Godot_(game_engine) (дата звернення: 15.05.2026).
27. Game design document. Wikipedia. URL: https://en.wikipedia.org/wiki/Game_design_document (дата звернення: 15.05.2026).
28. Screen-space effects (SSAO, SSIL, SSR, SSS). Godot Engine 4.6 documentation in English. URL: https://docs.godotengine.org/en/stable/engine_details/architecture/internal_rendering

_architecture.html#screen-space-effects-ssao-ssil-ssr-sss (дата звернення: 15.05.2026).
29. Using Jolt Physics. Godot Engine 4.6 documentation in English. URL: https://docs.godotengine.org/en/stable/tutorials/physics/using_jolt_physics.html (дата звернення: 15.05.2026).
30. Godot Engine official documentation. Using CharacterBody2D/3D. URL: https://docs.godotengine.org/en/stable/tutorials/physics/using_character_body_2d.html (дата звернення: 15.05.2026).
31. Godot Engine official documentation. Signals. URL: https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html (дата звернення: 16.05.2026).
32. Godot Engine official documentation. GPUParticles3D. URL: https://docs.godotengine.org/en/stable/classes/class_gpuparticles3d.html (дата звернення: 16.05.2026).
33. Godot Engine official documentation. Decal. URL: https://docs.godotengine.org/en/stable/classes/class_decal.html (дата звернення: 18.05.2026).
34. Godot Engine official documentation. Input and InputMap. URL: https://docs.godotengine.org/en/stable/tutorials/inputs/inputevent.html (дата звернення: 19.05.2026).
35. Godot Engine official documentation. Tween. URL: https://docs.godotengine.org/en/stable/classes/class_tween.html (дата звернення: 21.05.2026).
36. Godot Engine official documentation. AudioStreamPlayer3D. URL: https://docs.godotengine.org/en/stable/classes/class_audiostreamplayer3d.html (дата звернення: 16.05.2026).
37. Godot Engine official documentation. GridMap. URL: https://docs.godotengine.org/en/stable/classes/class_gridmap.html (дата звернення: 21.05.2026).

38. Swink S. Game Feel: A Game Designer's Guide to Virtual Sensation. Morgan Kaufmann Publishers, 2009. 376 p.
39. Rogers S. Level Up! The Guide to Great Video Game Design. 2nd ed. John Wiley & Sons, 2014. 512 p.
40. Sherif A., Sherif Y. Game Development with Godot 4.x. 2025. URL: https://arxiv.org/abs/2504.01598 (дата звернення: 21.05.2026).

ДОДАДКИ

Додаток А

```
extends Node3D
class_name PlayerModel

@onready var player = $".."
@onready var mmc : MovementModeController = $MMC

@onready var moves = {
	"idle" : $States/Idle,
	"walk" : $States/Walk,
	"sprint" : $States/Sprint,
	"jump" : $States/Jump,
	"fall" : $States/Falling,
}

var current_move : State

func _ready():
	current_move = moves["idle"]
	for move : State in moves.values():
		move.player = player
		move.mmc = mmc

func update(input : InputPackage, delta : float):
	var relevance = current_move.check_relevance(input)
	if relevance != "okay":
		switch_to(relevance)
		print("Actions: ", input.actions)
	mmc.check_modes(input)
```

```
        current_move.update(input, delta)

func switch_to(state : String):
        current_move.on_exit_state()
        current_move = moves[state]
        current_move.on_enter_state()
        current_move.mark_enter_state()
        print("State: ", state)
```

Додаток Б

```
class_name State
extends Node

var gravity = ProjectSettings.get("physics/3d/default_gravity")
var player  : CharacterBody3D
var mmc     : MovementModeController

@export var animation : String
var move_name         : String
var enter_state_time  : float

var has_forced_move : bool   = false
var forced_move     : String = ""

static var moves_priority : Dictionary = {
        "idle"   : 1,
        "walk"   : 2,
        "sprint" : 3,
        "jump"   : 10,
        "fall"   : 10,
}


# ___ lifecycle ___

static func moves_priority_sort(a: String, b: String) -> bool:
        return moves_priority[a] > moves_priority[b]

func top_action(input: InputPackage) -> String:
```

```
        input.actions.sort_custom(moves_priority_sort)
        return input.actions[0]

func check_relevance(input: InputPackage) -> String:
        if has_forced_move:
                has_forced_move = false
                return forced_move
        return default_lifecycle(input)

func default_lifecycle(_input: InputPackage) -> String:
        return "implement default_lifecycle in: " + animation

func update(_input: InputPackage, _delta: float) -> void:
        pass

func on_enter_state() -> void:
        pass

func on_exit_state() -> void:
        pass

func check_floor() -> String:
        return "fall" if not player.is_on_floor() else ""

# ___ timing management ___

func mark_enter_state() -> void:
        enter_state_time = Time.get_unix_time_from_system()

func get_progress() -> float:
```

```
        return Time.get_unix_time_from_system() - enter_state_time

func works_longer_than(time: float) -> bool:
        return get_progress() >= time

func works_less_than(time: float) -> bool:
        return get_progress() < time

func works_between(start: float, finish: float) -> bool:
        var p = get_progress()
        return p >= start and p <= finish



# ___ speed management ___

func mode_speed(base_speed: float) -> float:
        assert(mmc != null)
        return base_speed * mmc.speed_multiplier

func _apply_vertical(velocity: Vector3, delta: float) -> Vector3:
        velocity.y = -gravity * delta if not player.is_on_floor() else 0.0
        return velocity

func velocity_by_input(input: InputPackage, delta: float, speed: float) -> Vector3:
        var velocity   = player.velocity
        var horizontal = Vector3(velocity.x, 0.0, velocity.z)
        var direction  = (input.camera_basis * Vector3(
                input.input_direction.x, 0.0, input.input_direction.y
        )).normalized()
        horizontal = horizontal.move_toward(direction * speed, 15.0 * delta)
```

```
        velocity.x = horizontal.x
        velocity.z = horizontal.z
        return _apply_vertical(velocity, delta)

func apply_deceleration(delta: float) -> Vector3:
        var velocity   = player.velocity
        var horizontal = Vector3(velocity.x, 0.0, velocity.z)
        horizontal = horizontal.move_toward(Vector3.ZERO, 20.0 * delta)
        velocity.x = horizontal.x
        velocity.z = horizontal.z
        return _apply_vertical(velocity, delta)
```

Додаток В

```
extends Node3D
@onready var mmc = $"../PlayerModel/MMC"
@onready var camera : Camera3D = $Vert/Camera
@onready var fov_controller = $FOVController
@onready var player_controller = $".." as CharacterBody3D

@export var mouse_sens : float = 0.15
@export var camera_fov : float = 80

var _cam_height : float = CAM_STAND_Y
var cam_follow_speed : float = 20.0
const CAM_STAND_Y  : float = 0.5
const CAM_CROUCH_Y : float = 0.05

var rot_x : float
var rot_y : float
var rot_x_min : float = deg_to_rad(-80.0)
var rot_x_max : float = deg_to_rad(80.0)

var start_fov : float
var target_fov : float
var current_curve : Curve
var transition_time : float
var progress : float = 1.0

func _ready():
	camera.fov = camera_fov
	start_fov = camera_fov
	target_fov = camera_fov
```

```
        progress = 1.0
        Input.mouse_mode = Input.MOUSE_MODE_CAPTURED

func _physics_process(delta):
        _update_camera_transform(delta)

func _process(delta):
        if progress < 1.0:
                progress += delta / transition_time
                progress = clamp(progress, 0.0, 1.0)
                camera.fov = lerp(start_fov, target_fov, current_curve.sample(progress))

func _unhandled_input(event):
        if event is InputEventMouseMotion and Input.mouse_mode ==
Input.MOUSE_MODE_CAPTURED:
                rot_y -= deg_to_rad(event.relative.x * mouse_sens)
                rot_x -= deg_to_rad(event.relative.y * mouse_sens)
                rot_x = clamp(rot_x, rot_x_min, rot_x_max)

func request_fov_transition(new_target: float, curve: Curve, time: float) -> void:
        if new_target == target_fov:
                return
        start_fov       = camera.fov
        target_fov      = new_target
        current_curve   = curve
        transition_time = max(time, 0.001)
        progress        = 0.0

func _update_camera_transform(delta: float) -> void:
```

```
        var target_y = lerp(CAM_CROUCH_Y, CAM_STAND_Y,
mmc.crouch.current_ratio)
        _cam_height = lerp(_cam_height, target_y, cam_follow_speed * delta)
        var anchor_interp =
player_controller.camera_controller_anchor.get_global_transform()
        global_position = anchor_interp.origin + Vector3.UP * _cam_height
        global_rotation = Vector3(rot_x, rot_y, 0.0)
```