ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка плагіну для ігрового рушія з відкритим кодом для спрощення проектування перешкод для гравця»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Дмитро КОНОВАЛОВ
(підпис)

Виконав: здобувач вищої освіти групи ПД-44

Дмитро КОНОВАЛОВ

Керівник: Богдан ХУДІК
доктор філософії (PhD)

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Коновалову Дмитру Олексійовичу

1. Тема кваліфікаційної роботи: «Розробка плагіну для ігрового рушія з відкритим кодом для спрощення проектування перешкод для гравця»
керівник кваліфікаційної роботи доктор філософії (PhD) Богдан ХУДІК,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи обробки мови ігрового рушія, опис методів створення допоміжного функціоналу, технічна документація з описом бібліотек для створення асетів.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд та аналіз існуючих методів та технологій створення механіки польоту ворожих куль.
2. Проектування застосунку для автоматизованої побудови асетів ворожих атак на основі куль.
3. Програмна реалізація та опис функціонування застосунку для автоматизованої побудови асетів ворожих атак на основі куль.

4. Тестування застосунку.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма варіантів використання.
5. Алгоритм роботи застосунку.
6. Діаграма класів.
7. Екранні форми.
8. Апробація результатів дослідження

6. Дата видачі завдання «23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02-28.02.2026	
2	Аналіз та дослідження існуючих аналогів	01.03-07.03.2026	
3	Огляд існуючих алгоритмів створення шаблонів атак в різноманітних рушіях	08.03-10.03.2026	
4	Проектування застосунку для створення асетів для створення шаблонів атак	11.03-23.03.2026	
5	Програмна реалізація застосунку	24.03-04.04-2026	
6	Тестування застосунку	05.05-01.05.2026	
7	Оформлення роботи: вступ, висновки, реферат	02.05-07.05.2026	
8	Розробка демонстраційних матеріалів	08.05-10.05.2026	
9	Попередній захист роботи	11.05-22.05.2026	

Здобувач вищої освіти _____ Дмитро КОНОВАЛОВ
(підпис)

Керівник
кваліфікаційної роботи _____ Богдан ХУДІК
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 51 стор., 1 табл., 21 рис., 21 джерело.

Мета роботи – спрощення роботи зі створенням перешкод для гравця за рахунок принципів ігрових снарядів та розробці допоміжного забезпечення для покращення ефективності геймдизайну.

Об’єкт дослідження – процесс створення ассетів перешкод засновані на повендінці снарядів

Предмет дослідження – ігровий рушій Godot та методи створення плагінів всередині рушія.

Короткий зміст роботи: В роботі проаналізовано алгоритми та методи для створення асетів перешкод на основі польоту куль в контексті атак ворога. Проаналізовано інструментальні засоби для створення шаблонів атак: BulletUpHell, Danmakufu, Uni Bullet Hell. Розроблено алгоритм створення асету програмно реалізовані ключові функціональні можливості, зокрема: створення об’єкту шаблону, зміна параметрів шаблону, призначення списку правил руху кулі, створення ресурсів для опису польоту кулі, зміна правил поведінки кулі, відображення польоту куль у ігровому просторі. Проведено функціональне та модульне тестування додатку. В роботі використано бібліотеки Godot Engine та скрипт для малювання позначень у тривімірному просторі для візуалізації шаблону у редакотрі.

Сферою використання застосунку є ігровий рушій Godot з використанням 3D сцен.

КЛЮЧОВІ СЛОВА: ШАБЛОН, ПОВЕДІНКА КУЛІ, АСЕТ, ОБ’ЄКТ АСЕТУ, СЦЕНА, ІНСПЕКТОР, ПЛАГІН, GODOT, NODE, ПЛАГІН, ПАРАМЕТРИ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

GD — GDScript

2D — 2 Dimmensional

3D — 3 Dimmensional

NES — Nintendo Entertainment System

ДЗ — Допоміжне забезпечення

Shmup — Shoot ’em up

IDE — Integrated Development Enviorment

VR — Virtual Reality

GIMP - GNU Image Manipulation Program

FPS — Frames Per Second

ВСТУП

Актуальність: З підняттям популярності ігор, розроблених незалежними ігровими студіями, також збільшуються до вимог доступності ігрових рушіїв. Високодоступні ігрові рушії є відмінним рішенням початківців в ігровій індустрії, або студіям, які за певних рішень або бюджетних лімітів, не можуть дозволити собі розробку власного рушія. На сьогоднішній день в фгровій індустрії домінують чотири рушії: Unity, Unreal Engine, GameMaker Studio та Godot. Кожен рушій має власні особливості та переваги, але їх поєдує висока доступність та можливість доповнювати рушій додатковим забезпеченням, або плагінами для нового функціоналу.

Під час розробки гри, виникає певний етап, коли виступає потреба у створенні ворожих атак. Простим рішенням було б створити скрипт або асет, який би описував саму атаку і виконував їх у середовищі. Це відноситься навіть для таких фундаментальних дій як ворожий постріл: Достатньо створити об’єкт кулі, прив’язати простий скрипт до об’єкту з якого «вилітає» куля і висати в поведінку ворога. Але при потребі більш коплесної поведіки кулі (вплив зовнішніх сил на кулю або необхідність створювати кілька куль одночасно), виникає ідея для пошуку допоміжного забезпечення, який спрощює цей процес.

Проблема, яка буде вирішена цим застосунком є відсутність плагіну для створення кулевих перешкод для 3D просторів у рушії Godot. Плагін буде використовувати принципи ігор жанрів Shmup та існуючих альтернатив в інших рушіях для створення унікальних типів об’єктів для створення куль і організації їхнього польоту у реальному часі. Плагін працюватиме всередині редактору рушія і використовуватись через інтерфейс рушія (зокрема через Інспектор).

Об’єктом дослідження є процес створення скриптованих асетів гри з фокусом на створення та менеджмент куль в просторі.

Предмет дослідження: Ігровий рушій Godot з вбудованим функціоналом допоміжного програмного забезпечення.

Мета роботи: спрощення роботи зі створенням перешкод для гравця за рахунок принципів ігрових снарядів та розробці допоміжного забезпечення для покращення ефективності геймдизайну рівнів.

Методи дослідження: Пошук аналогічних рішень в альтерантивних ігрових рушіях і адаптація принципів створення перешкод у редакторі або через код.

Наукова новизна роботи визначається втіленням пріорітетно 2D механік у 3D середовищі і використання іструментів рушія для створення нових типів об’єктів з новим функціоналом.

Практична значущись: Вирішення технічної відсутності у ринку допоміжного забезпечення рушія Godot, оптимізація процесу створення асетів перешкод та покращення ефективності геймдизайну при втіленні перешкод та атак ворогів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

Механіки польоту куль є одним із ключових механік в багатьох жанрах відеоігор, зокрема в швидких іграх які фокусуються на вистрілах або будь яких рухомих перешкод. Ця механіка бере корені з ігор ери аркад (близько кінця 1970-х до середини 1980-х). Одією з перших ігор з цієї ери, яка включала у себе подібну механіку є аркадна гра Space Invaders 1978 року. Гри була дуже проста: з верхньої частини екрану спускались інопланетяни і вистрілювали кулі в сторону гравця. Гравець в свою чергу керував маленьким кораблем, який рухався тільки наліво або направо і вистрілював кулю в сторону інопланетян. Якщо куля потрапляла в ворога, той помирає. По тій самій логіці, якщо ворог попаде в гравця, гравець також помирає. Це стало базою для використання куль як механіки завдавання шкоди, як ворогу, так і гравцю.

З часом розвитку ігор в технічному плані, також і еволіціонувала механіка куль у методах їх втіленнях. Серед прикладів можна зазначити ігри з жанру Shmup, який став прямим наслідником Space Invaders. Цей жанр також фокусується на польоті ворогів і іхніх куль з верхньої частини екрану і задачі граця будучи пріорітетно на ухиляння від куль замість завдавання шкоди. На сьогоднішній день жанр досі має нові ігри, які випускаються незалежними (інді) студіями і збирають аудиторію. Жанр навіть зміг зробити свій піджанр Bullet Hell, де складність гри збільшувалась в кілька разів за рахунок великої кількості куль і дуже чутливого управління персонажем.

1.1 Hitscan та Projectile постріли

В той же час, жанри шутерів також мали власний прорив у використаннях куль. Одним із таких проривів є Quake — шутер від першої особи який став золотим стандартом для шутерів від першого лиця (як мінімум тих, які брали перевагу над швидкістю гри). Там можна визначити те, що постіли двух видів: фізичні (projectile)

та проєекційні (hitscan). Різниця між ними в тому, що hitscan постріли працюють за методом уявлення польоту кулі по прямій лінії і надавання «ефекту подпдання» першому об’єкту що потрапляє під проєкцію. Цей метод перкрасно підходить для зброї з швидкими кулями, такі як дробовики, лазерні гармати, пістлоети, ітд. Фізичні постріли в свою чергу запускають реальний об’єкт по прораховаій траєкторії і мають певну поведінку при стиканні з поверхнею. Це підходить для зброї, чиї постріли можуть бути вплинуті зовнішніми силами (куля з гранатомету падає під впливом гравітації) або якщо політ снаряду повільніший (ракета з ракетниці).

Саме ця механіка польоту снарядів стала базою для багатьох ігор зі зброєю, оскільки не тільки шутери використоввували цю логіку для польоту фізичних об’єктів. Серія ігор Portal, яка була головоломкою в рушії для шутерів (Source), використовує hitscan для основного ісрументу гри — портальної гармати. Але при цьому, в грі існують перешкоди які задають прямої шкоди гравцю пострілами: Турелі які стріляють швидкими hitscan пострілами та енергетичні сфери, які є projectile кулями що летять по заданій траєкторії.

1.2 Розвиток ігрової індустрії та інді-сектору

На початку життя відеоігрової індустрії, розробкою ігор займались лише компанії, які займались програмним забезпеченням, оскільки ще не було ніяких очікувань на потенціал віртуальних ігор щодо прибутку на ринку. Згодом, з’явились команії, які фокусувались виключно на розробці ігор та приладів для їх відтворенні. Одним із таких компаній була Atari, яка в свій час була гігантом відеоігор з домашньою консолю Atari 2600. Але незабаром після успіху Atari відбулась криза відеоігрової індустрії у 1983 році (яка впала в прибутку з 3.2 млрд. доларів до 100 млн.), причиною якого було перенасичення ринку неліцінзійованими іграми від третіх лиць поганої якості.

Після кризи 1983 року, компанії як Nintendo та SEGA почали свій старт в індустрії, бувально підіймаючи її з занепаду. Щоб не повторити провал Atari,

компаніям, які розробляли домашні консолі, такі як NES (або Famicom в Японії) та SEGA Mega Drive, також доводилось ліцензувати ігри які надходили до компаній і давати дозвіл на випуск ігор на їхніх пристроях. Правила були відносно жорстокі, від яких відбулись перші кейси піратства ігор в індустрії, але це задало стандарт в розробці ігор: студія розробляє гру, видавець контролює якість і випускає на своїй платформі. Сьогодні така методика випуску ігор видозмінена, оскільки тепер видавництва набагато більші ніж раніше і мають тенденцію викупати студії для свого видавництва, що не завжди є вигідним рішенням.

З часом, ігорва індустрія поступово розширялась і набувала нових властивостей і вимог щодо випуску продукту: віковий рейтинг, контроль якості, рецензії, маркетинг, цільові аудиторії, та ін. При цьому, доступність до інструментів розробки ігор теж змінювалась. Коли в 2000-х роках, щоб розробити гру треба було мати команду програмістів і інструменти для розробки ігор на певні платформи (або ігрові рушії). Сьогодні достатньо скачати один із популярних рушіїв (Unity, GameMaker, Unreal Engine, Godot), і можна робити гру мрії виключно самотужки.

Завдяки збільшення доступності ігрових рушіїв, в ігровій індустрії поступово виростав абсолютно новий сектор: інді. Ігри з цього сектору розробляються незалежними студіями з розробки (іноді навіть трьома людьми) і видаються на популярних платформах дистрибуції ігор. І оскільки кількість інді-ігор збільшувалась, паралельно зростала і аудиторія з попитом на цей сектор. Є шанс, що гравець який пройде маленьку інді-гру, надихнеться і захоче зробити свою власну гру.

1.3 Ігрові рушії і плагіни

Популярність таких ігрових рушіїв як Unity чатсково була спричинена саме вибухом популярності інді-ігор, на чому Unity Technologies і бере дохід за рахунок частки продаж гри, яка була розроблена в їхньому рушії. Також великою превагою цих рушіїв це можливість модифікувати сам рушій за допомогою плагінів —

допоміжного забезпечення, яке встановлюється всередину рушія і додає новий функціонал або набір інструментів для рушія. Ці плагіни можуть додавати як і нові об’єкти або іструменти для покращення ефективності роботи з рушієм, так і створювати нові фреймворки усередині уде існуючого фреймворку. Ці допоміжні програми зазвичай знаходяться окремому ринку, який виділений суто під плагіни для рушіїв (Unity Asset Store, Godot Asset Library, вкладка плагінів для Unreal Engine в Epic Games Store). Також є плагіни, які мають власні репозиторії (наприклад в GitHub) або поширюються через інші платформи в інтернеті (магазини ігор, архіви, тощо).

Одим із плюсів цих плагінів є те, що вони розроблюються незалежними розробниками, і підтримуються користувачами рушія. Використання плагінів під час розробки ігор сильно заохочується, оскільки це в рази спрощує процесс розробки певних частин гри, зменшує витрачений час який був би витрачений на розробку з чистого аркуша та пошуку інформації як імплементувати ці частини самотужки.

Проблема виникає коли для імплементування певної механіки, бракує потрібних плагінів або істрементів для автоматизації цього процесу. Така проблема існує у менш популярних рушіях, таких як Godot або GameMaker. В більшості випадків доводиться вкиористовувати те, що є і робити цю механіку власноруч, або шукати альтернативи які не поністю підходять до вимог гри. Або якщо знайдено рішення в іншому ігровому рушії, доводиться мігрувати проєкт з одного рушія в інший, що іноді вимагає написання коду з нуля.

Також є варінт розробити потрібний плагін власноруч, що дозволяють розробити самі грові рушії. Якщо рушій має вбудовані інструменти для розробки плагінів, цілком можливо розробити допоміжне забезпечення яке вирішує конкретну проблему, для якої не існує рішень в ринку допоміжного забезпечення рушія. Але при цьому, цей процес вимагає більше зусиль для аналізу, розробки, коригуванню та тестування плагіну, оскільки розроблюється те, чого в ринку плагінів майже не існує.

1.4 Аналіз аналогів

Проблема, яка буде вирішуватись в цій роботі є відсутність плагіна для створення асетів перешкод на основі projectile куль в ігровому рушії Godot. Проблема виникає з того, що гра або проєкт, на яку орієнтується планіг, є нетиповою сумішшю жанрів Shmup та шутерів з першого лиця з жанровою переважністю першого. Таке поєднання є нетиповим, оскільки Shmup жанр складається з переважно 2D ігор, коли шутери від першого лиця є переважно 3D іграми (є ігри які існують в 2D середовищах, але використовують приципи 3D середовищ для відображення глибини середовища). Із-за нетиповості поєднання і специфічності жанру Shmup, більшість рішень розраховуються на 2D ігри, або 3D іграми з видом зверху. Для цієї проблеми візьмемо припущення що камера гри не прив’язана до точки над полем для іллюзії 2D гри, а прив’язана до самого гравця для виду з першого лиця.

Аналоги які будуть розглядатися будуть використані як приклади заствосування механік Shmup в 2D середовищах для адаптації у 3D середовища. Один із цих аналогів буде використано як приклад втілення конфігурації асетів у рушії Godot через вкладку Інспектора в редакторі рушія. Після аналогу буде створенна приблизна термінологія для опису функціоналу механіки, об’єкти та процесси які виконуються під час створення та виконання асету в грі.

Використані аналоги мають бути допоміжним забезпеченням або повноцінними ігровими рушіями, які концентруються на механіка пострілу куль та їхньому менеджменту під час польоту. Оскільки специфіка жанру висагає переважно 2D середовища, ДЗ яке буде аналізуватись має працювати в 2D проєктах незалежно, чи може програма прцювати з 3D проектами.

Подальші розділи оглянуть приклади вже реалізованих рішень у вигляді ДЗ до ігрового рушія або фреймвворку ігрового рушія і визначать переваги та недоліки в контексті завдання.

1.4.1. Uni Bullet Hell

Uni Bullet Hell — плагін для ігрового рушія Unity, який дозволяє створювати комплексі атаки кулями, як для 2D, так і для 3D. Серед функціоналу є можливість налаштовувати комплексні траєкторії та формувати власні атаки за допомогою опцій.

Це один із багатьох ДЗ для рушія Unity, які імплементують механіки польоту куль з відносно простим інтерфейсом, та дозволяє створювати прості ворожі атаки з параметрами швидкості, прискорення, направлення польоту, швидкості обертання, тощо.

Простота використання є відносно середньою, оскільки для зручного використання достатньо купити плагін з сторінки Unity Asset Store, інсталювати ДЗ та створити потрібні об’єкти в проєкті. Сам плагін є чудовим рішенням для тих, хто збирається розроблювати ігри на рушії Unity, прикладом є рисунки 1.1 та 1.2. Але при цьому недоліком є те, що цей плагін доступний лише для цього рушія, і його не можна використовувати в інших рушіях.

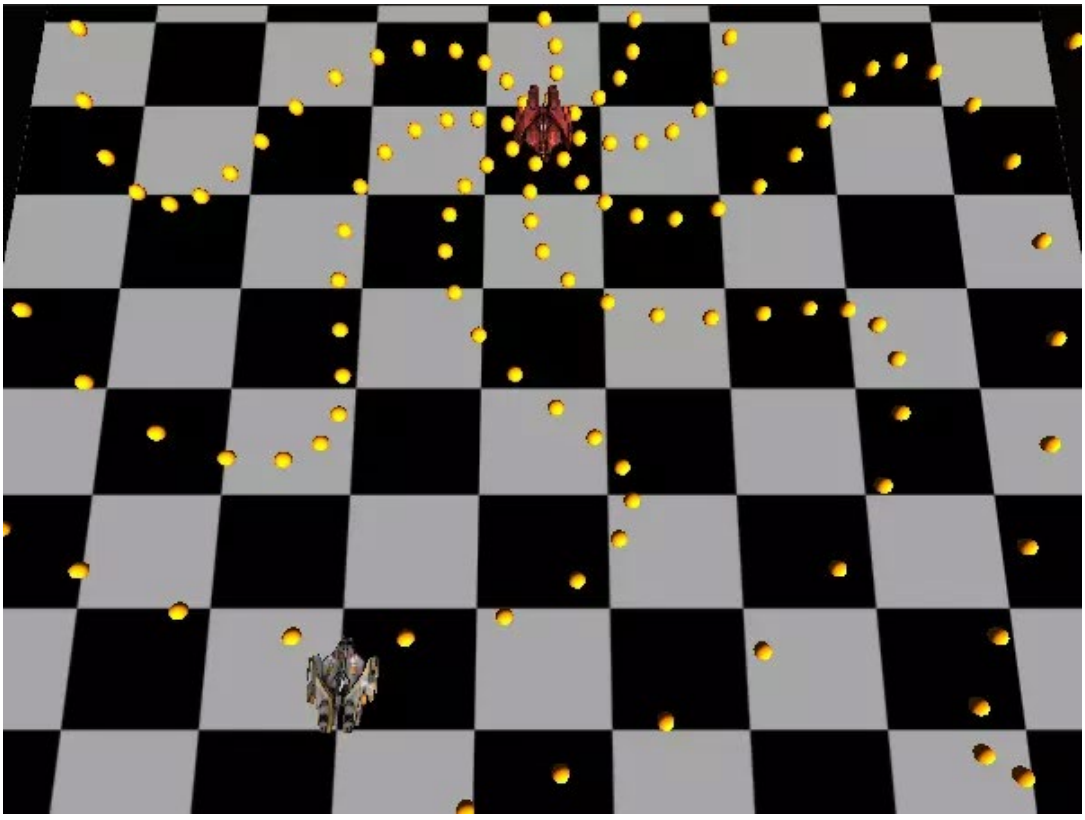


Рис. 1.1 Демонстрація Uni Bullet Hell в 3D

Рис. 1.2 Демо-сцена плагіну в 2D просторі

1.4.2 BulletUpHell BLAST

Аналогічним рішенням для рушія Godot є плагін BulletUpHell BLAST — повноціне додаткове забезпечення, яке виконує функції створення перешкод за допомогою куль, використовуючи набір власних типів об’єктів. Серед таких асетів можна віднести абстрактні об’єкти якорних точок, тригерів, характеристик куль та «шаблонів» - об’єктів, які займаються створенням куль за набором правил, описаних в його характеристиках. Об’єкти, які включаються у плагін є наступними:

Spawnpoint - точка створення куль та цетром об’єкта шаблону. Асет має характеристики в основному щодо створення куль. Створення куль можна встановлювати як на певний проміжок часу, так і на певні умови, які встановлюються асетом TriggerContainer.

TriggerContainer — асет який визначає умови для створення куль і підключається через поле умови в Spawnpoint. В асеті можна виставляти різні умови, такі як певний проміжок часу, колізія з поверхнею фізичного об’єетка, ітд. Це один із варіантів створення комплексних шаблонів, оскільки більш комплексні

перешкоди мають тенденцію змінювати поведінку куль під час польоту. Зміни цих куль залежать виключно від умов, які зазначені в асеті.

BulletProp — характеристики, які визначають яку поведінку має куля під час польоту. Це одна з найдетальніших асетів плагіну, оскільки туди включаються такі опції як швидкість польоту, траєкторія (яка може як і прямою лінією так і користувацькою кривою), поведінка під час знищення кулі, наведення на точку в просторі під час польоту, ітд. Є одним із ключових частин «шаблону» атаки, бо асет має увесь потрібний опис для поведінки кулі, як для протих пострілів, так і до комплексних атак.

Pattern — це функціональний асет який поєднує усе рахом в один шаблон. В ньому визначаються кількість куль, частота їх створення, початкова траєкторія (яка відрізняється від поведінки траєекторії в BulletProp), та інші властивості щодо організації створення куль. Це є основним елементом шаблону перешкоди, оскільки він зв’язує усі елементи в одному об’єкті і запускає сам процес польоту.

Плагін має високу гнучкість завдяки великій кількості налаштувань характеристик перешкоди і має безкоштовну демо-версію, яку можна завантажити з сайту ДЗ і встановити в рушій Godot. Із мінусів можна виділити те, що плагін підтримує лише 2D проєкти, що не вирішує поставлену проблему і певні баги при імпортуванні демо-версії в новіші версії рушія. Також треба зазначити що для того щоб подивитись роботу асету, треба запустити сцену гри, що робить перегляд в редакторі неможлививм, як на рисунку 1.3.

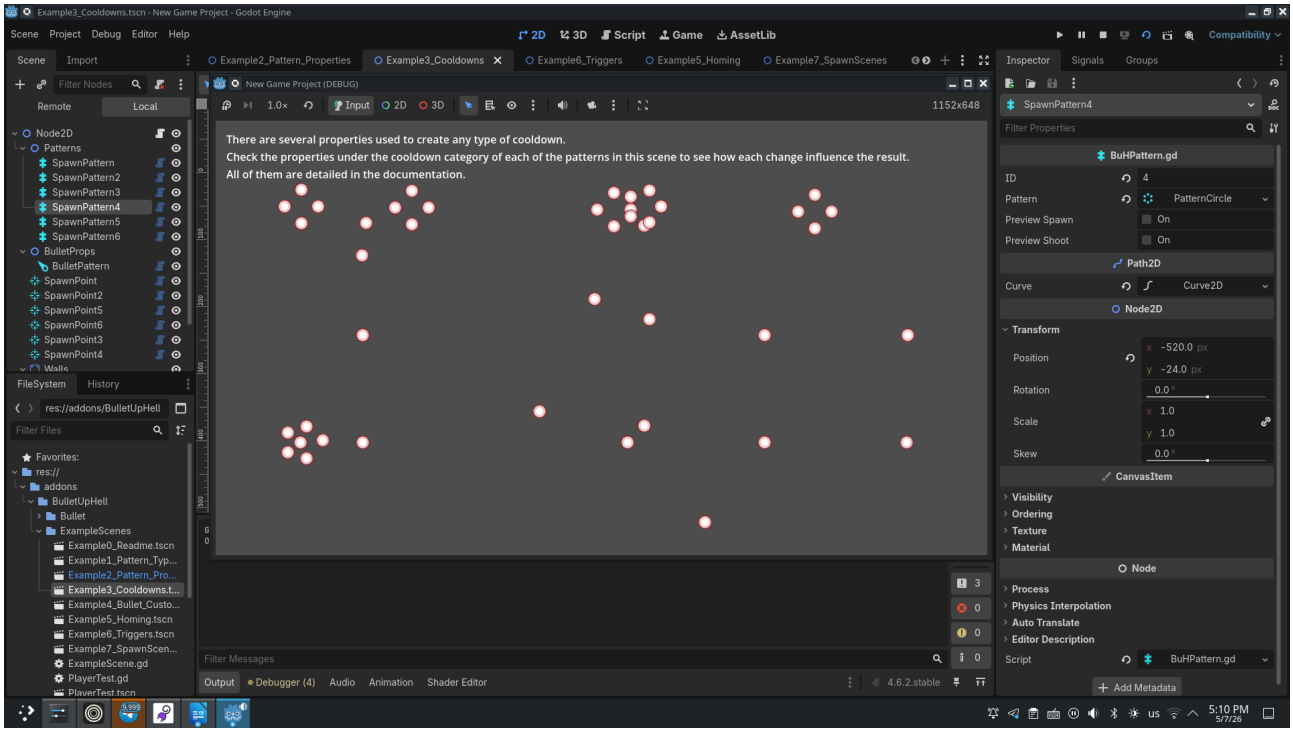


Рис. 1.3 Демонстрація демо-версії BulletUpHell в вікні гри

1.4.3 Danmakufu

Danmakufu можна назвати одним із нішових ігрових рушіїв, оскільки він розроблений в великим напрямом на жанри Shmup та Bullet Hell. Завдяки цьому, у рушія є дуже висока гнучкість створення перешкод та атак куль ворога, оскільки це його основний функціонал. Написання атак відбувається написанням скрипта сценарію, який ініціалізує ворога, надає йому характеристик та спрайт в грі, після чого прописується сценарій атаки ворога.

Для скриптів використовується власний фреймворк, який за граматикою сходий на C. Один скрипт може описувати як одну конкретну атаку або послідовність атак, що залежить від типу скрипта, який пишеться. Серед мінусів: Відсутній інтерфейс рушія, що означає що уся розробка гр на цьому рушії виконується в текстовому редакторі або IDE який використовується для написання скрипта.

З цього витікає наступна проблема, що пов’язана з запуском гри, яка розробляється, бо немає інструментів перевірки атаки під час написання скрипту, що не є інтуїтивним. Але якщо розробник хоче зробити дуже якісний Shmup і має

час на вивчення цього рушію, цей рушій теж можна рекомендувати, як зображено на рисунку 1.4.

Рис. 1.4 Демо-скрипт для рушія Danmakufu

Вище були описані аналоги, які набільше підходять під ціль проекту. Кожег з цих рішень маж свої переваги, такі як висока гнучкість створених асетів, чи простота використання під час процесу розробки. Але при цьому, кожен з них має і свої недоліки, які впливають як і на рішення конкретної проблеми, так і на рішення самого розробника на вибір інстркментів перед початком створення гри.

У таблиці 1.1 буде наведено порівняння наведених аналогів за категоріями простоти використання, підтримки 3D просторів, доступності функціоналу та гнучкості створених асетів. Ці критерії є найважливішими, оскільки ціль плагіну — легка автоматизація створення асетів куль під час гри з високою гнучкістю конфігурації атак та підтрика у 3D іграх.

Таблиця 1.1

Зведені результати аналізу характеристик методів розробки шаблонів атак

Критерій	**Uni Bullet Hell**	**Bullet Up Hell BLAST**	**Danmakufu**
Доступність функціоналу	Доступний лише з магазину плагінів Unity і не має безкоштовної демо-версії	Деякі функції недоступні у демо-версії плагіну	Доступний повний функціонал
Підтримка 3D	Підтримує 2D і 3D	Підтримується лише 2D	Підтримується лише 2D
Простота використання	Середня, потрібно розуміти принципи створення шаблонів	Середня, потрібно розбриратись в налаштуванні типів об’єктів плагіну	Висока, треба знати мову скриптів Danmakufu та вміти працювати без інтерфейсу
Гнучкість асетів	Низька, доступні лише кілька простих налаштувань. Неможливо робити комплексні атаки.	Висока, завдяки великій кількості параметрів та модульній системі шаблонів	Висока, завдяки власній мові написання скриптів
Допоміжна документація	Відсутня допоміжна документація	Наявна повна документація плагіну і навчальні відео	Повний навчальний блог використання рушію з демонстраціями роботи коду.

1.5 Визначення вимог

З наведеного аналізу аналогів, їхнього функціоналу, сособливостей рішень, переваг та недоліків, було сформовано перелік функціональних та нефункціональних вимог до розроблювалного плагіну зі створення асетів перешкод на основі польоту куль.

Функціональні вимоги:

1. базові шаблони для створення асетів перешкод (лінія, кільце, куб, сфера) з

характеристиками щодо кількості куль на шаблон, частота створення куль та час між килами створення потоку куль;

2. опції для показу точок створення куль в редакторі, ліній для показу початкової траєкторії куль та можливість змінювати відображуваний колір точок та лійній;відображення поведінки польоту куль безпосередньо у редакторі (без запуску гри) з метою тестування поведінки куль та коригування шаблону безпосередньо в редакторі;

3. можливість редагування шаблонів в редакторі без втручання в код плагіну, використовуючи вкладку інспектора в редакторі та надані абстактні об’єкти плагіна.

Нефункціональні:

1. чіткий процес створення об’єктів шаблонів в редакторі (створення об’єкту, зміна параметрів, назначення змін для поведінки куль тестування поведінки в середовищі, коригування неправильної поведінки);

2. використання плагіну без додаткової конфігурації плагіну (лише процес встановлення та увімкнення в редакторі);

3. незалежність від іншого допоміжного програмного забезпечення для використання у власному проєкті з самого початку розробки гри.

1.6 Інформаційна модель предметної галузі

Інформаційна модель предметної галузі — уявлення ключових сутностей та взаємозв’язків усередині програмного забезпечення в контексті плагіну зі створення перешкод на основі польоту куль. Метою побудови цієї моделі є визначення логіки плагіну, взаємодія змінних та об’єктів у просторі між собою та навколишнім середовищем під час роботи шаблону у грі.

Центральною сутністю є шаблон (Pattern) — абстрактний об’єкт який тримає в собі ключову інформацію про себе та ключовий функціонал самого шаблону. Шаблон тримає в собі інформацію про об’єкт, який він створює (Bullet Scene), швдкість створення (Spawn rate), час між періодами створення куль (Cooldown

Rate), ім’я самого шаблону (не є обов’язковим для визначення), стан відображення точок створення куль у просторі (Show Rig), список поведінки куль під час польоту (Bullet Behviour) і стан виконання шаблону в реальному часі (Display).

Ціль шаблону (не залежачи від типу самого шаблону) це створити список координат точок, з яких вилітають кулі і надавання створенним кулям список поведінки куль під час кожної нової ітерації створення куль. В залежності від типу шаблону (лінія, кільце, куб або сфера), виконуються обрахунки координат точок для рівномірного розподілення по «площині» шаблону. В деяких шаблонах за кількість створення куль відповідає оереме число (Amount), коли в деяких за це відповідає деталізація вершин шаблону (куб або сфера).

Нижче наведено діграму класів шаблонів з описом всіх доступних змінних для редагування та усі наявні функції для обрахунку координат і запуску куль з оброблених координат (Рис 1.5). Оскільки Обрахунки ліній та кубів не мають простих формул обрахунку, вони вимагають більше функцій для обрахунку координат.

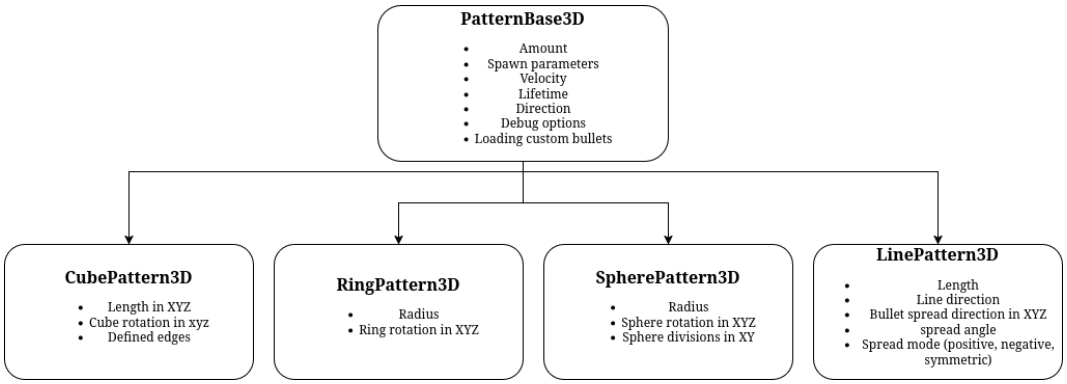


Рис. 1.5 Діаграма класів шаблонів для плагіна

Bullet Script — це скрипт, який автоматично прив’язується до кулі під час її створення.Скрипт відповідає за призначення кулі характеристик і пересування кулі у 3D просторі. Сам скрипт є прихованим і не може бути відреагованим в Інспекторі. Однак впиливати на характеристики кулі можна за допомогою списку Bullet Behaviour який має в собі список ресурсів типу BulletMove.

Список являє собою послідовність рухів кулі у просторі методом призначення нових характеристик з наступного Bullet Move. Коли в списку більше немає даних, куля автоматично знищується.

Bullet Move — це абстрактний ресурс, який тримає в собі інформацію про характеристики кулі під час певного етапу в послідовності Bullet Behaviour. Ресурс має в собі дані про швидкість (Velocity), тривалість цієї поведінки (Lifetime), логічна зміна для збігу траєкторій в одну точку (Converge) та ціль траєкторії польоту кулі (Aim At). Цією цілюю може бути як і конкретний внектор направлення, або інший об’єкт. Якщо Converge є позитивним і кулі летять в інший об’єкт, траєкторії куль збігаються у місцезнаходжені цього об’єкту. Або якщо Converge є хибним, кулі пропорційно відлітають від центру цілі.

Процес створення кулі, схемою якої є рисунок 1.6, відбувається у шаблоні, де істанціюється сцена кулі, яка буде літати. До цієї кулі додають скрипт Bullet, до якої також додають послідовність BulletBehaviour. Послідовність починається з індексу 0, коли відбувається перший рух Bullet Move. Коли час Lifetime спливає, скрипт автоматично змінює рух у послідовності і продовжує своє існування.

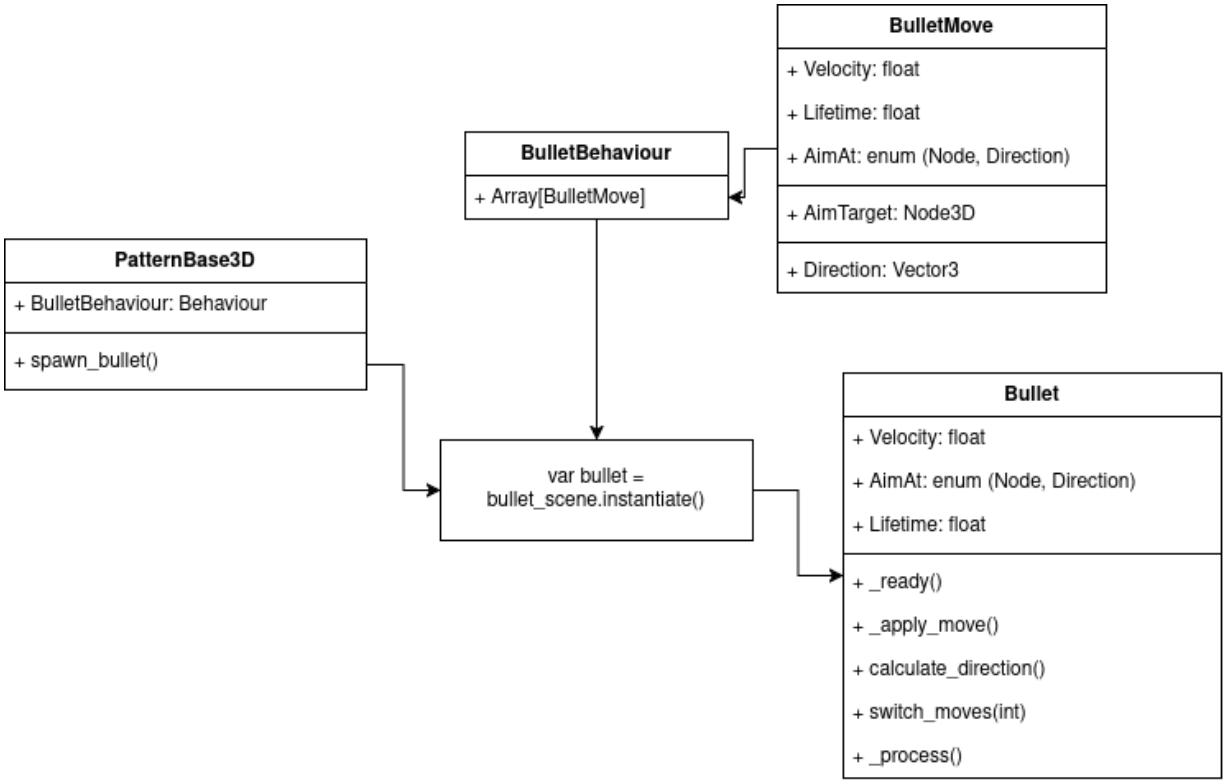


Рис. 1.6 Процес створення кулі

2 ЗАСОБИ РЕАЛІЗАЦІЇ ЗАСТОСУНКУ

2.1 Середовище розробки

Оскільки робота виконується в рамках ігрового рушія, для роботи використовувалися інструменти, надані самим рушієм. Забезпечення розроблюється для рушія Godot, тому будуть використані саме інструменти Godot. До цих інструментів також включається і вбудований IDE — забезпечення, яке надає розробникам інструменти для написання коду програми з інструментами для написання, налагодження та виправлення коду безпосередньо в рушії. Для роботи з рушієм Godot можна використовувати і сторонні IDE від інших компаній (наприклад Visual Studio Code), але саме IDE Godot підходить найкраще, бо він має усі потрібні бібліотеки встановлені в середовище і має вбудовані посилання на документацію.

Графічна частина роботи проводилась в GIMP — безкоштовному графічному редакторі. Робота в програмі була обмежена лише створенням піктограм нових об’єктів та форматування зображень у потрібний графічний формат.

2.1.1 Godot Engine

Godot Engine — один із найпопулярніших ігрових рушіїв на сьогоднійшній день, розроблений Godot Foundation. Рушій отримав великий приріст популярності завдяки декількох успішних інді ігор (Slay The Spire II, Buckshot Roulette, Cruelty Squad). Такод за допомогою рушія можна створювати власні програми та іструментами для покращення продуктивності, особливо в розробці ігор. Одним із таких інструментів, розроблених на рушії є Material Maker — графічний редактор шейдерів та текстур, що використовує діаграми для створення текстур, анімованих шейдерів, 2D анімацій, та ін.

Одним із переваг цього рушія є відкритий код. Це дозволяє не тільки повністю безкоштовне програмне забезпечення, але і повна свобода модифікації

програми під власні потреби. Рушій розповсюджується за ліцензією MIT, що надає повний безкоштовний доступ до рішуя. Якщо розробник створює модифікацію рушія, або випускає гру що розроблена в цьому рушії, він має познатчити використання MIT ліцензії в своєму продукті. Це єдине правило щодо використання ліцензії, яке дає повну свободу для розповсюдження власного продукту. Завдяки відкритості коду та наявним платформам розповсюдження, розробники можуть ділитися своїми плагінами та модифікаціями рушія, іноді навіть вносити вклад у розробку основного рушія.

Окрім створення ігрового контенту за допомогою самого рушія, Godot має великий іструментів для покращення ефективності розробки гри. До таких іструментів можна віднести: графічні редактори текстур та шейдерів, редактор анімацій, інструменти тестування, власний IDE, підтримка кількох мов програмування (в тому числі мови шейдерів). Також за допомогою плагінів, можна додавати власний функціонал, наприклад імпортування мап з гри Quake, створювання нових типів об’єктів з власним функціоналом, оптимізація інтерфесу рушія, робота з VR, тощо.

Godot має велику навчальну базу, включаючи документацію, яка детально описує повний функціонал рушія, використання функцій GD, рекомендації щодо використання певних об’єктів, тощо. Спільнота Godot також робить навчальні відео для початківців, які допомагають засвоїти рушій та принципи певних жанрів ігор, якої логіки та норм слід дотримуватися для хорошої гри.

2.1.2 Ієрархія об’єктів Godot

На відміну від Unity, де об’єкти в сцені існують незалежно один від одного, в Godot повноційна ієрархічна система де кожен об’єкт має бути дочірнім до іншого об’єкту. Основним об’єктом в Godot є нода (Node) — Абстрактний об’єкт який є базовим об’єктом усієї ієрархії рушія та від якого походять три основні ноди: Node2D, Node 3D, User Interface. Ці три основні ноди є ключовими нодами, оскільки вони є сценами або інших об’єктів.

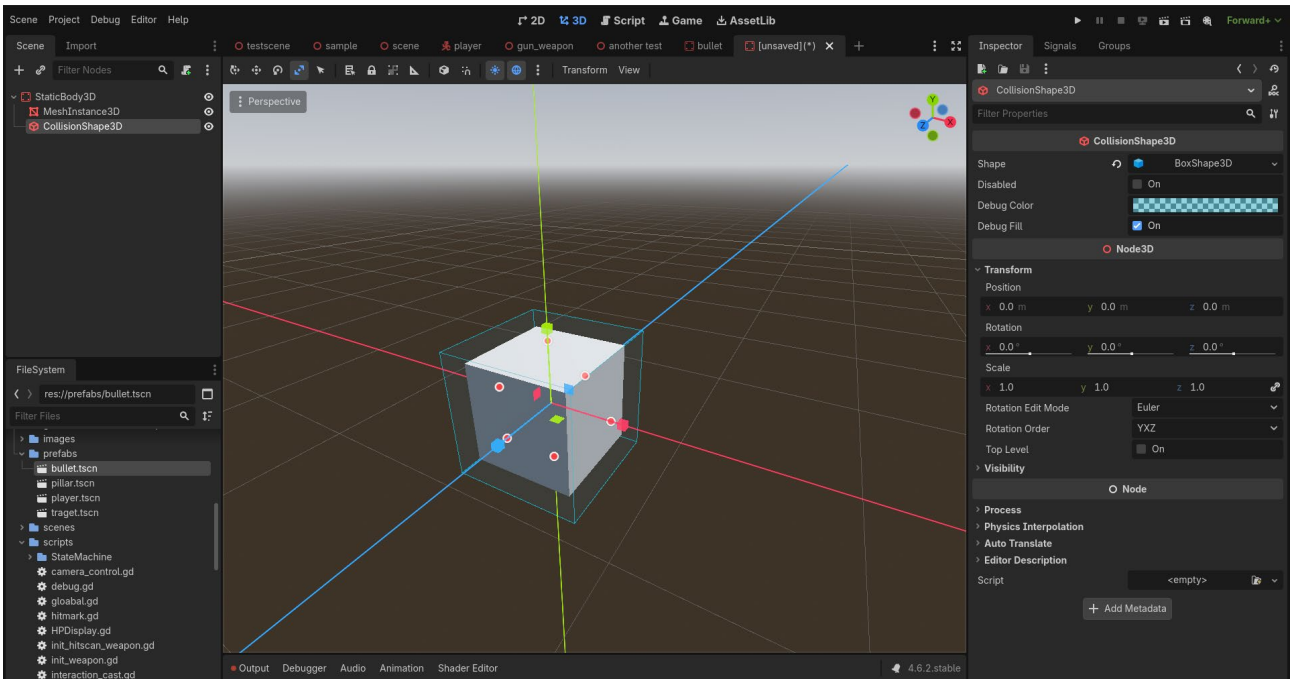


Рис. 2.1 Створення куба в Godot

Оскільки дочірний Node3D, може бути роміщений в 3D просторі завдяки властивостям, належним 3D об’єктам (розташування, обертання, розмір), воно може вважатись як окрема сцена. При такій модульній системі можна створювати комплексні мапи при використанні простих об’єктів, що покращує читаємість та реюзабіліті (можливість використання об’єктів в інших сценаріях) асетів, не залежачи від інших об’єктів що знаходяться в сцені.

На рисунку 2.1 зображено сцену куба, де основним об’єктом є StaticBody3D — об’єкт, який позначає властивості фізичних об’єктів з параметрами фізики об’єкта, стиканнями з іншими об’єктами (колізії) та типових властивостей Node3D. Цей об’єкт вважається фізичним об’єктом з фізичною оболонкою CollisionShape3D та візульною оболонкою MeshInstance3D.

Дочірні об’єкти CollisionShape3D та MeshInstance3D виступають як додаткові характеристики фізичного об’єкта. MeshInstance3D відповідає за візуальну репрезинтацію куба у 3D просторі, що робить його видимим у грі, коли CollisionShape3D відповідає за фізичну частину об’єкта, фігуруючи як хітбокс — тобто невидима фігура з фізичними властивостями колізії, що дозволяє фізичному об’єкту поводитись за законами фізики.

Визначення фігури колізії важливо для різних ситуацій: від визначення що

фігури перетнулись без фізичного впливу один на одного, або використання фізичного об’єкту як зовнішньої сили на інший об’єкт. Цей приклад є найпростішою формою фізичного об’єкта, який може стикатись з іншими фізичними об’єктами.

Усі об’єкти в Godot також можуть присвоїти скрипти для їхнього виконання в грі. Кожен об’єкт може присвоїти собі лише один скрипт. Якщо скрипт має посилання на інший об’єкт з скриптом (зазвичай цим виступають дочірні об’єкти), то об’єкт може виконувати скрипт який належить посланому об’єкту. Альтернативним рішенням є система сигналів — процес передачі певного сигналу з одного об’єкту в інший.

Ця система працює за наступним принципом: В скрипті Object1 робится сигнал Signal1, і через код надсилається на усю сцену в якій об’єкт знаходиться. Тим часом, інший об’єкт зі скриптом Object2 зі слухачем (listener), який очікує Signal1. Якщо Object2 отрмує Signal1 з іншого об’єкту, тоді він виконує функцію, яка залежала від цього сигналу.

Godot підтримує 2D і 3D простори, що вимагають різних об’єктів з різною поведінкою на площині. З цієї причини, неможливо розмістити спрайт, який є дочірнім від Node2D у тривірному просторі, якщо не надати їй 3D площині з одним лицем (Plane) для розташування спрайту як текстури цієї площини.
Набір об’єктів, з якими буде працювати розробник залежать від стилістики або жанрових вимог гри. Але це не означає що для певного жанру треба використовувати лише певні об’єкти. Колекційні карткові ігри можуть існувати як в 2D, так і в 3D стилістиці.

2.1.3 Godot IDE

Godot має вбудований IDE в рушії, що дозволяє писати скрипти безпосередньо в рушії гри (див. Рис. 2.2). Це не тільки прибирає потребу в окремому IDE для роботи з рушієм, але і дає усі потрібні інструменти для роботи з грою без істалювання додаткових бібліотек чи плагінів для роботи з рушієм. IDE використовується для написання скриптів, які можуть бути присвоєні об’єктам,

скриптів шейдерів та інших файлів, які не відносяться до самого рушія Godot (наприклад редагування json).

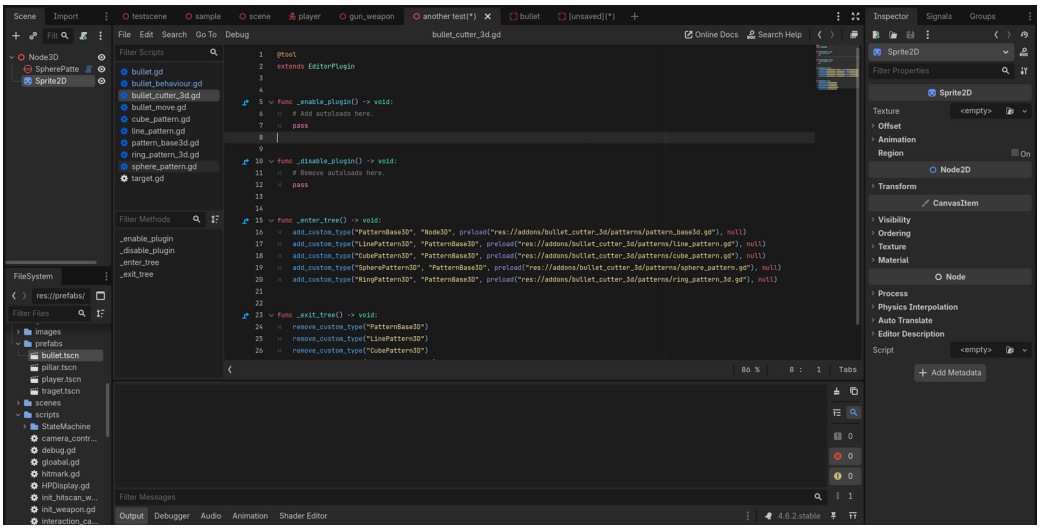


Рис. 2.2 Вигляд IDE всередині рушія Godot

IDE має мінімальний інтерфейс з панеллю відкритих скриптів, функціями наявні в скрипті, пошук слів в скрипті, та кнопки для пошуку інформації в документації. Якщо функція походить з функціоналу рушія, IDE виведе короткий опис функції та змінні які вона приймає.

2.1.4 GDScript

GDScript — власна мова програмування ігрового рушія Godot. Сам рушій також підтримує C#, але в більшості випадків використовується саме GDScript. За синтаксом, мова програмування схожна на Python, бо більшій частині форматуванням функцій та певними функціями, які перекочували напряму з Python. Але при цьому, в GD є власні особливості, які полегшують роботу з об’єктами і рушієм взагалом.

Одним із таких особливостей є використання власних аннотацій. Наприклад, для того щоб вивести змінну у поле Інспектора для більш зручного редагування, перед об’явленням змінної використовується тег @export. Це найпростіше використання тегу для виставлення змінної в Іспектор,оскілтк існують варіації цього тегу для більш точних значень (@export_range для виставлення певного

діапазону змінної, @export_group та @export_category та сортування змінних по функціям та категоріям).

Для використання скрипту усередині самого редактора рушія, використовується тег @tool. Цей тег каже рушію, що цей скрипт також виконується всередині редактора, що є корисним для тестування певних скриптів та об’єктів без запуску самої гри. Це допомагає розробляти плагіїни для рушія Godot безпосередньо в самому IDE, тут же перевіряти роботу плагіну і воносити правки усередині одного програмного забезпечення.

2.2 Графічний редактор GIMP

Потреба в графічній візуалізації може виникнути навіть коли розроблюється технічні рішення як плагін для ігрового рушія. Задля комфортності використання плагіну та читабельності об’єктів плагіну, об’єкти мають мати піктограми (icons) для позначення їхнього функціоналу і виділення їх поміж інших об’єктів. Для створення цих піктограм, рекомендовано використовувати графічний редактор, який може працювати з поширеними графічними файлами.

GIMP (GNU Image Manipulation Program) — графічний редактор з відкритим кодом, який є альтернативним рішенням популярних графічних редакторів, таких як Adobe Photoshop. Завдяки відкритості коду, програму теж можна модернізувати і змінювати під власний комфорт, що допамагає виправляти технічні або персональні проблеми з програмою. Деякі користувачі, особливо ті, які працюють з Adobe Photoshop, вже адаптувались до клавішних комбінацій цієї програми. Для цього існує метод заміни комбінацій клавіш GIMP на комбінації клавій Adobe Photoshop.

GIMP є хорошою альернативою для користувачів, які не хочуть робити підписку на сервіси Adobe, хочуть зекономити гроші, але не хочуть втрачати можливість редагувати фото на вищому рівні. На сьогоднішній день, альтернатив платних програм стає все більше, створюючи конкуренцію компаніям, які домінували певними ринками і розбиваючи стагнацію в розвитку ПЗ.

2.3 Git та контроль версій

Система контролю версій - це ключовий інструмент у процесі розробки програмного забезпечення, не залежачи відбувається в команді чи індивідуально. Вона дозволяє зберігати історію змін у проєкті, відслідковувати модифікації, повертатися до попередніх версій коду, і ефективно організовувати співпрацю між кількома розробниками. Для розробки подібного програмного забезпечення, особливо програм з відкритим кодом, дані проекти мають перевагу за рахунок вкладів з інших розробників. Якщо один розробник викласть свою програму в репохиторії, інший розробник може внести свій вклад в подальшу розробку програми за допомогою модифікацій або додаванням коментарів до вже існуючого коду.

Більшість систем контролю версій слідкують за принципом Git — розподільною системою контролю версією програми. Під час підготовки проекту для системи контрою, створюється репозиторій програми — директорії програми які слугують основною папкою програми. До основної папки репозиторію також додаються системні файли, такі як README.md для короткого опису та документації програми та .gitignore що дозволяє ігнорувати бібліотеки при додаванні файлів до репозиторію.

Після створення локального репозиторію, розробник може вносити нові файли до репозиторію, змінювати їх та видаляти з репозиторію. Кожна зміна в репозиторії повина помічатись комітом для підтвердження зміни (наприклад коміт видалення застарілих скриптів) та коментарем до коміту для короткого опису змін. Цього достатньо для індивідуальної розробки ПЗ та контролю програми на випадок якщо ПЗ перестане працювати методом повернення до минулої версії програми.

Для групових проєктів, репозиторій публікується на сайтах хостингу репозиторіїв (зазвичай це GitHub, але є альтернативи) і налаштовуються параметри доступності. Від цих параметір залежить, хто має доступ до репозиторію та має права вносити коміти до репозиторію. Відомою практикою є форкінг (forking) —

відгалуження програми на певному коміті у окрему гілку для розробки альтернативної версії ПЗ паралельно з іншими гілками. Така практика дозволяє тестувати певні функції ПЗ, розбиття завдань між членами команди та злиття гілок у нову версію програми. Це дозволяє швидко розроблювати ПЗ у команді, і при цьому фокусуватись на індивідуальних завданнях.

Для початку роботи з Git, достатньо завантажити середовище Git на комп’ютер де проводиться робота. Але не всі зможуть комфортно працювати з Git через термінал, оскільки Git працює лише через команди без інтерфейсу.

Для цього є два варіанти рішень: скачати застосунок для контролю версій як GitHub Desktop або Sourcetree, чи додати інтерфес до звичайної версії Git. Lazygit — це програмний пакет, який являє собою оболонку Git через інтерфейс в терміналі. Програми такого плану називають TUI (Terminal UI) програмами.

Це підходяще рішення для тих, хто хоче більше контролю над репозиторієм Git, але не хоче встановлювати додаткові програми. Навігація по інтерфесу відбувається за допомогою клавіатури з підказками в нижній частині інтерфейсу. На рисунку 2.3 зображено інтерфейс lazygit.

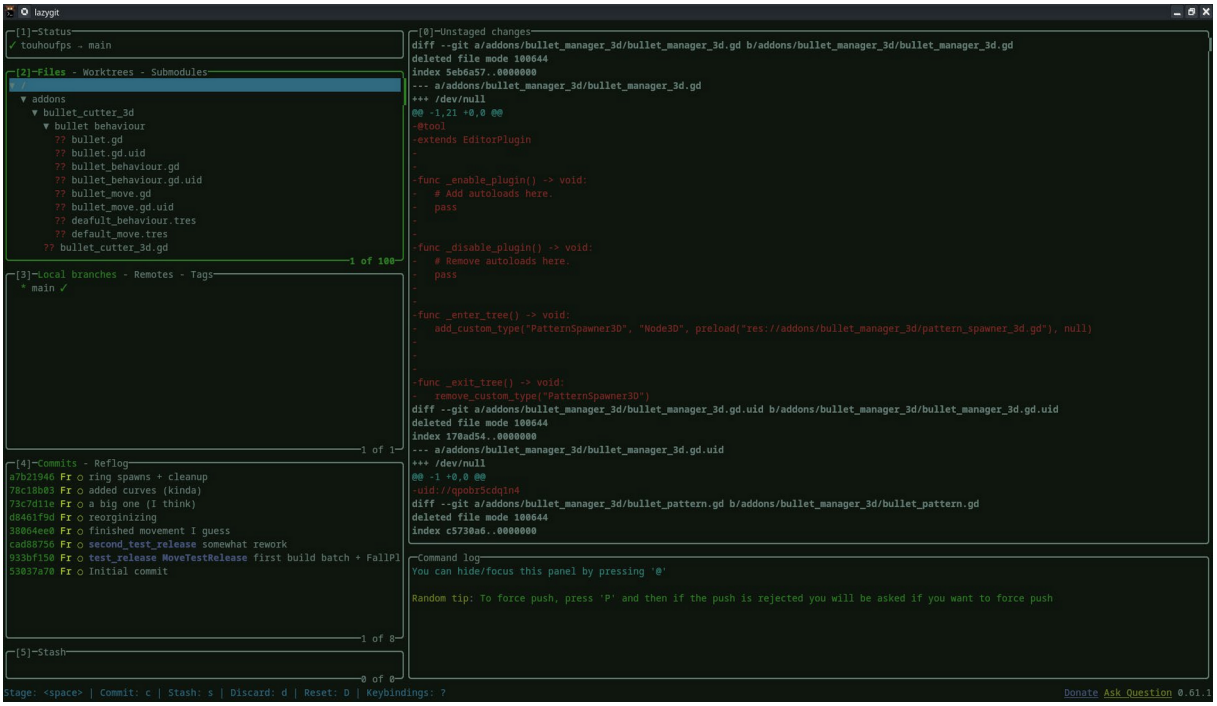


Рис. 2.3 Інтерфейс lazygit в терміналі

Під час роботи викотовувався саме цей набір програм для розробки прототипної версії плагіну. Для написання самого плагіну та тестування забезпечення у ділі використоввувався ігровий рушій Godot версії 4.6 та вбудований IDE для написання коду програми. Графічний дизайн піктограм був зроблений в графічному редакторі GIMP з параметрами комбінацій клавіш, схожиз з Adobe Photoshop. Для контролю версій та випуск першої пробної версії застосунку було використано Git з пакетом lazygit для комфортного контролю проекту.

Слід зазначити що усі використані ПЗ обирались за принципом лекої доступності та можливості модифікування програм, оскільки це дає розробникам більше контролю над самою програмою без бюджентих або ліцензійних обмежень. Сам рушій Godot був обраний завдяки великій потужності рушія та відносно швидкому зростанні забезпечення серед розробників. Створення цього плагіну вважається як вклад як в розвиток ПЗ на ринку плагінів застосунку.

3 ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Архітектура системи

Під час розробки програмного забезпечення, було приділенотувагу щодо архітектури плагіну для подальшого масштабування програми і втілення нових функцій. Але оскільки ДЗ існує всередині ігрового рушія, кожен користувач матиме доступ до коду програми. Для цього було обрано варіацію класичної багаторівневої архітектури, до основні об’єкти, з якими буде працювати розробник будуть лекодоступні та створенні з меню об’єктів Godot. При цьому, об’єкти, які виконують основні обрахунки та логіку плагіну будуть «приховані», тобто недоступні через меню об’єктів і вимагатимуть прямого втручання в код плагіну для зміни поведінки. Також буде створенно змінні-ресурси, які не будуть доступні через меню об’єктів, але можуть бути додані до шаблонів на сцені як змінні, що впливатимуть на поведінку об’єкта.

Такий підхід підходить за рахунок модульності плагіну і можливості додавати нові об’єкти та змінні. Це буде заохочувати колаборативність проєкту, де інші розробники за бажанням та можливості можуть внести свої змінні (новий функціонал для куль, тригери переходів між станами, ітд). Таким чином, програма матиме два рівня доступності: простий та складний. В простому рівні розробник використовує об’єкти та змінні які він додав в проєкт і користується лише ними. В свою чергу, складний рівень означатиме створення нових шаблонів, розширення функціоналу уже існуючих об’єктів та змінних та коміти в репозиторій плагіну (див. Рис. 3.1).

Зберігання інформації про створені об’єкти за допомогою плагіну будуть зберігатися локально в проекті, оскільки розробник створив їх на простому рівні і використовує їх лише в своєму проекті. Демо-сцени будуть створені на складному рівні, оскільки вони будуть додані до репозиторію в якості пробних або навчальних цілей.

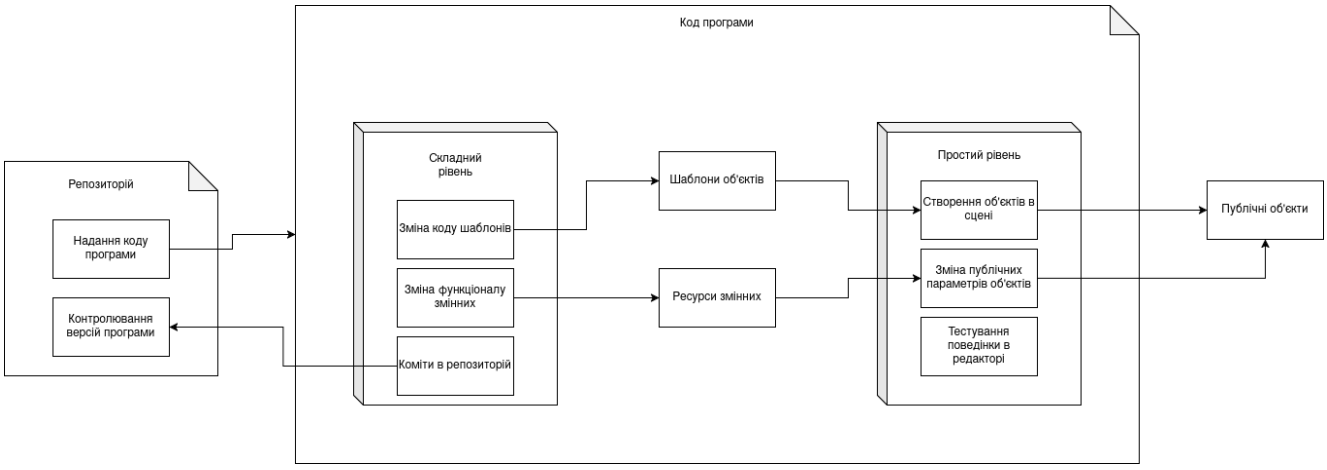


Рис. 3.1 Діаграма рівнів доступу до програми

3.2 Діаграма використання плагіну

Під час моделювання програми, важливо визначити поведінку кінцевого користувача плагіну і методи створення об’єктів за допомогою наданих іструментів під час розробки гри. В випадку цього проєкту, кінцевим користувачем є розробник гри або геймдизайнер. Їхня задача — створити перешкоду, яка буде стріляти кулі на певному інтервалі з визначеною швидкістю та направленням. Вони використовуватимуть об’єкти шаблонів, надані плагіном для створення об’єктів на сцені.

Після визначення кінцевого користувача та його цілей, сворюється діаграма поведінки користувача під час створення потрібного асету гри. Ця діаграма описуватиме наступний сценарій:

1. Відкриття меню об’єкта шаблону та додавання його у сцену. Тип шаблону залежить від вибраного об’єкту з меню.

2. Розміщення об’єкту шаблону у сцені три

3. Зміна параметрів частоти створення куль групами і тривалості між періодами створення груп куль

4. Додавання послідовності поведінки куль під час польоту за необхідності в унікальній поведінці польоту

5. Додавання рухів кулі у певний період і налаштовування параметрів

швидкості та траєкторії польоту разо з тривалістю періоду.

6. Тестування роботи шаблону у дії

7. Якщо розробник задовільнений результатом роботи і в сцені наявні усі бажані шаблони, розробник закінчує процес створення перешкод.

Наступна діаграма візуально показує процесс створення перешкод куль від створення першої перешкоди куль до завершення роботи з плагіном в конкретній сцені. Ця діаграма є важливою, оскільки вона описує поведіку геймдизайнера як кінцевого користувача плагіну і основним логічним процесом використання продукту:

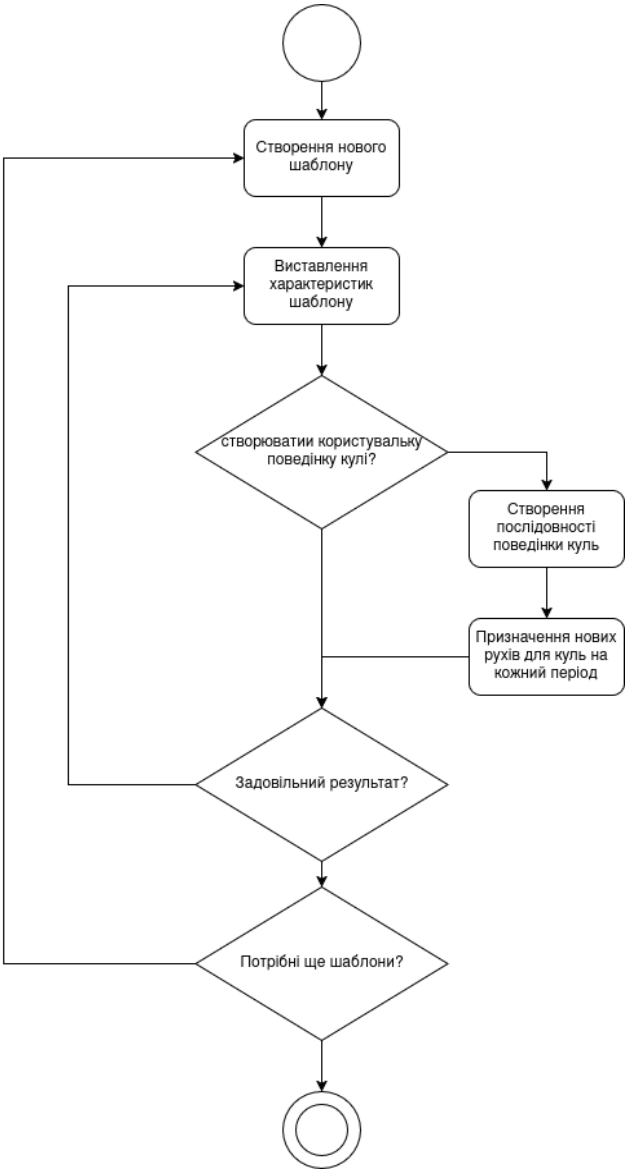


Рис. 3.2 Діаграма поведінки використання плагіну

3.3 Реалізація системи

Після того як буда визначена архітектура програми і поведінка користувача привикористанні плагіну, можна розбирати тезнічну частину розробки проекту. Плагін складається з двух частин: Шаблони і кулі. Шаблони — це об’єкти які доступні для створенні в сцені гри і займають роль розміщення точок, з яких вилітатимуть кулі, створення самих куль і присвоєння їм скриптів поведінки.

В свою чергу, частина куль складається з скрипту кулі, який автоматично присвоюється створеній кулі, послідовності поведінок куль, та самі данні поведінки куль, які тримають усю потрібну інформацю про пересування. Скрипт кулі буде займатись пересуванням кулі, до який він призначений на основі даних з активного на даний період руху кулі.

Увесь проект був написаний у вбудованому IDE рушія Godot з інформацією, взятий з офіційної документації про мову програмування GDScript та організовний всередині репозиторія гри, де розроблювався плагін. Контроль версій проекту виконувався за допомогою Git через інтерфейс lazygit.

3.3.1 Ініціалізація плагіну

Для початку роботи з плагіном, треба створити пустий плагін через вкладку Project > Project Settings > Plugins > Create New Plugin. Після заповнення інформації про назву, кореневий скрипт, автора та версію плагіну, в директорії гри створюється папка addons. Ця директорія зберігає усі плагіни, які додані до проєкту. Усередині цієї папки також створиться папка самого плагіну, усередині якої буде кореневий скрипт, як на рисунку 3.3.

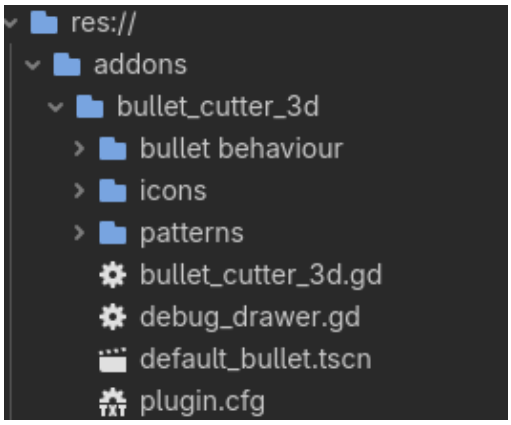


Рис. 3.3, Директорія плагіну в папці addons

Кореневий скрипт плагіну має ту же назву, що й назва самого плагіну, але за бажанням, її можна змінити. Скрипт є дочірнім класом від EditorPlugin, що дає йому наслідування усіх потрібних функцій для роботи всередині рушія. Аннтотація @tool означає, що скрипт буде працювати безпосередньо в редакторі рушія. Цю анотацію також мають усі створені класи в плагіні для можливості тестування роботи ПЗ всередині програми та задовільнення одного з вимог проєкту: Безпосереднє використання функцій плагіну всередині рушія без необхідності запускати гру.

3.3.2 PatternBase3D

Після початкової ініалізації плагіну, починається розробка основного шаблону — PatternBase3D. Це батьківський клас, який має усі основні дані про шаблон та основну поведінку шаблону під час його роботи. Завдяки цьому, виходить модульна система шаблонів, від якого йдуть дочірні класи.

Ці дочірні класи матимуть ту сму поведінку що й PatternBase3D, але матимуть різні форми розміщення точок створення куль та методи обрахування координат цих точок. Схему класів зображено на рисунку 3.4.

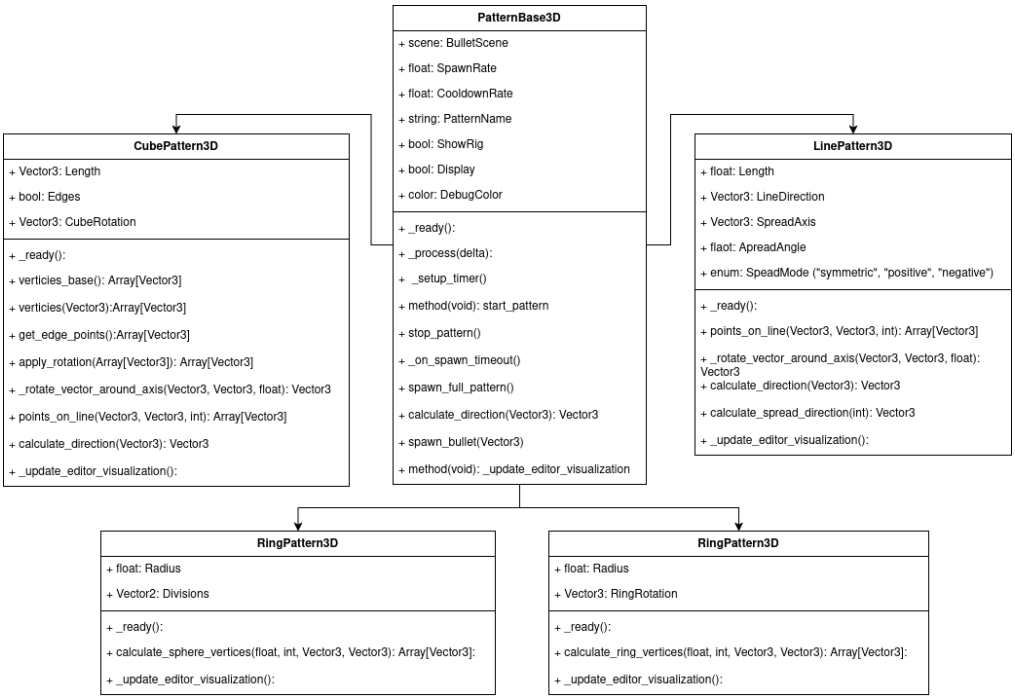


Рис. 3.4 Розширена діаграма класів шаблонів

PatternBase3D розподілений на три категорії: тестування (Debug), Посилання на об’єкт кулі (Reference) та самі параметри базового шаблону (Spawn). В категорії тестування наявні два логічні змінні для перегляду точок в сцені (Show Rig) і запуску дії шаблону в середовищі (Display). Також наявна змінна кольору відображення графів шаблону на сцені (Debug Color). В категорії посилання на об’єкт присутня лише одна змінна, яка приймає Node3D об’єкти для використання в якості кулі. Основною категорією є Spawn, в якому присутні значення для частоти створення куль (Spawn Rate) та проміжку часу між потоками куль (Cooldown Rate).

Функція _ready() є вбудованою функцією GDScript, яка викликається під час інілізації об’єкту, в якому знаходиться скрипт (зазвичай при пешому кадрі сцени). Функція Виконує початкову підготовку скрипту, зокрема встановлення внутрішнього таймеру та функція завантаження debug_loading(). Ця функція підключає скрипт з малювання ліній та точок у просторі, присвоює її до дочірнього об’єкта, та використовує функції цього скрипту для малювання точок за допомогою функції _update_editor_visualization().

```
func debug_loading() -> void:
	var DebugDrawerScript = preload("res://addons/bullet_cutter_3d/debug_drawer.gd")
	debug_drawer = Node.new()
	debug_drawer.set_script(DebugDrawerScript)
	add_child(debug_drawer)

func _ready() -> void:
	debug_loading()
	_setup_timer()
	if Engine.is_editor_hint():
		_update_editor_visualization()
```

_update_editor_visualization() - це одна з головних функцій класу, яка займається відображенням точок та ліній шаблону у просторі сцени. Функція виконується тільки за у мови якщо Show Rig стає позитивним. Базова версія підчищає уже існуючі точки, якщо вони були створені раніше. Функція бере список уже існуючих точок та ліній та видаляє їх з сцени за допомогою .clear() та .free_queue(). Цей метод є базовою функцією очищення зайвих або застарілих точок, що використовується і в дочірніх класах.

```
func _update_editor_visualization() -> void:
	if not Engine.is_editor_hint():
		return
	# Clear existing visualization
	for line in _editor_lines:
		if is_instance_valid(line):
			line.queue_free()
	for point in _editor_points:
		if is_instance_valid(point):
			point.queue_free()
	_editor_lines.clear()
	_editor_points.clear()

	if not show_rig:
		return
```

_process() також є функцією рушія Godot, яка викликається кожен кадр. В PatternBase3D, функція використовується для відрахунку внутрішніх таймерів, які слідкуються функцією _on_spawn_timeout(). В залежності від частота Spawn Rate дорівнює нулю, шаблон вистрелює усю групу куль і стартує відрахунок нового таймеру та часу між групами куль Cooldown Rate. Функція _process() виконується тільки за однієї з двух умов: якщо шаблон працює всередині запущеної гри або стан Display є позитивним. Якщо умови не задовільні, ця функція ігнорується.

```
func _process(delta: float) -> void:
	if Engine.is_editor_hint() and global_transform != _last_transform and show_rig:
		_update_editor_visualization()
		_last_transform = global_transform
	if Engine.is_editor_hint() and not display:
		return

	# Simple "Auto-fire" for testing/preview
	if display or not Engine.is_editor_hint():
		if _spawn_timer.is_stopped() and _cooldown_timer <= 0:
			start_pattern()
	# We still need to count down the cooldown before the next burst can start
	if _cooldown_timer > 0:
		_cooldown_timer -= delta
```

Коли таймер Cooldown досягає нулю, функція запускає кулі за допомогою функції spawn_full_pattern(). Сама функція spawn_full_pattern() проходить через список існуючих точок шаблону і запускає функцію spawn_bullet(), яка робить об’єкт кулі який був визначений в Bullet Scene, призначає йому скрипт bullet.gd та назначає кулі послідовність рухів BulletBehaviour. Цей процесс відбувається з кожною створеною кулею в групі, які з’являються зі швидкістю Spawn Rate:

```
func spawn_bullet(origin: Vector3 = global_position):
	if not bullet_scene: return

	var bullet = bullet_scene.instantiate()
	bullet.set_script(bullet_script)

	# Give it a fresh behaviour so each bullet tracks its own index independently
	var b := BulletBehaviour.new()
	b.BulletMoves = Behaviour.BulletMoves  # shared array ref is fine, moves are read-only
	bullet.behaviour = b

	get_parent().add_child(bullet)
	bullet.global_position = origin
```

PatternBase3D має дочірні класи шаблонів, які відповідають за створення куль впродовж лінії, кільця, куба та сфери. Ці об’єкти є єдиним, що доступні для додавання до сцени через меню об’єктів. Ці дочірні об’єкти виконують більшість функцій батьківського класу, разом з модифікаціями функції _update_editor_visualization() та власні функції для розрахунку координат точок.

3.3.3 LinePattern3D

Першим з дочірніх класів є LinePattern3D — шаблон, який створює кулі впродовж прямої лінії у 3D просторі. Окрім значень, яка були успадковані з PatternBase3D, клас також має власні значення для побудови лінії. Цими значеннями є вектор направлення Direction та довжина лінії Length. Також для класу наявні данні для рівномірного розкиду куль, що є потенційним методом розширення функціоналу у майбутніх версіях продукту.

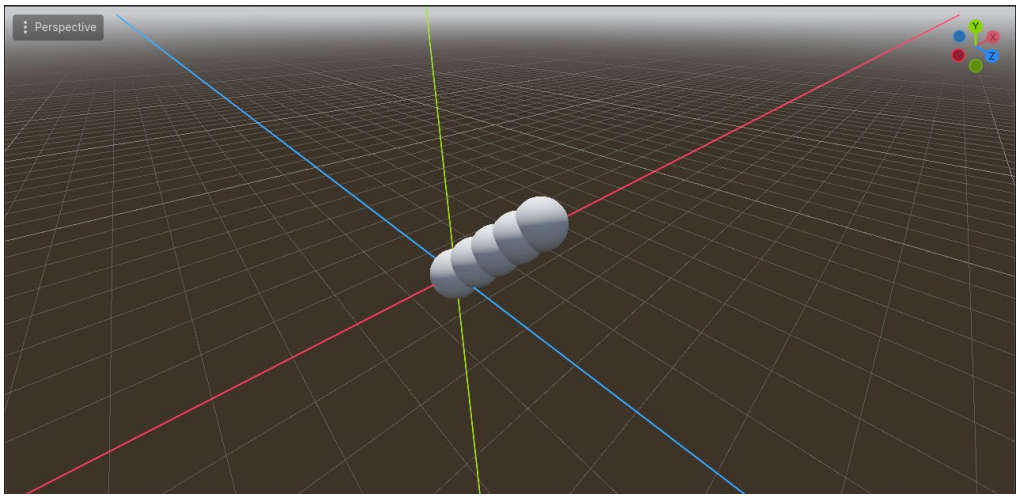


Рис. 3.5 LinePattern3D у редакторі

Для використання цього, так і інших дочірніх шаблонів, потрібно провести розрахунки координатів точок, з якиз будуть створюватись кулі. Для цього беруться данні з категорії LinePattern3D.

```
end_pos = global_position + line_direction.normalized() * length
```

Ця формула обчислює кінець відрізка методом додавання добутку вектору на довжину лінії та початкової точки, чим виступає global_position як координата центру шаблону. Після обрахування, відомо точки початку та кінеця відрізку.

Основною функцією класу LinePattern3D для обчислення точок шаблону є points_on_line(). Функція приймає початок та кінець відрізку разом з кількістю куль в шаблоні (що визначається значенням Amount класу PatternBase3D).

Якщо кількість точок дорівнює нулю, то функція ігнорується. В іншому випадку, якщо кількість точок більше нуля, позиції обраховуються за циклом:

```
for i in range(num_points):
        var t = float(i) / float(num_points - 1)  # 0.0 to 1.0
        points.append(start.lerp(end, t))
```

Функція проходить через набір чисел від нуля до числа кількості точок (num_points) та прораховує віносну позицію точки впродовж лінії. Наприклад, якщо у лінії всього три точки, дві точки стають початком та кінцем відрізку. Але третя точка стає посередині, оскільки вона пропорційна буде стояти на центрі відрізку.

Після формування нового списку, точок, запускається додаткова логіка _update_editor_visualization(), яка виконує логіку батьківського класу, після якої функція малює нові точки на місцях координат зі спику точок, що було отримано раніше.

3.3.4 RingPattern3D

На відміну від LinePattern3D, який має власні функції з ручними обрахунками координат, RingPattern3D має лише одну функцію яка обчислює координати точок методом обертання точки навколо осей XYZ. Клас бере радус кільця (Radius) та градуси обертання у 3D просторі (Ring Rotation).

Процес обрахунку координат точок відбувається наступним чином:

1.Створення нового списку точок для збереження координат і використання при побудові точок створення куль.

2. Конвертація значення обертання кільця з градусів в радіани за формулою 3.1.

$$rrrrrr = \frac{\pi\pi}{180}, \tag{3.1}$$

де rad — градуси обертання кільця в радіанах (зберігається в значенні Vector3), α — градус обертання в градусах.

1. Створюється матриця обертів у тривірному просторі розміром 3 на 3:

```
var basis = Basis()
	basis = basis.rotated(Vector3(1, 0, 0), rot_rad.x)  #Rotate around X
	basis = basis.rotated(Vector3(0, 1, 0), rot_rad.y)  #Rotate around Y
	basis = basis.rotated(Vector3(0, 0, 1), rot_rad.z)  #Rotate around Z
```

де rot_rad — градуси обертання в радіанах

2. Для кожної точки в кільці, виконуються обрахунок локального кута відносно площини XY

```
for i in range(num_vertices):
	# Angle for this vertex (in radians)
	var angle = (2.0 * PI * i) / num_vertices

	# Local position on the ring (in XY plane initially)
	var local_pos = Vector3(
		radius * cos(angle),
		radius * sin(angle),
		0.0
	)
```

3. Глобальний кут обраховується методом суми добутку локального кута на матрицю обертання і глобальної позиції шаблону.

```
var world_pos = basis * local_pos + position
ring_points.append(world_pos)
```

В результаті отримається список координат кільця, який обернений навколо осієй XYZ, з радіусом та деталізацією вершин визначених кількістю куль в шаблоні.

Взятий список потім оброблюється функцією _update_editor_visualization() для візуалізації в редакторі.

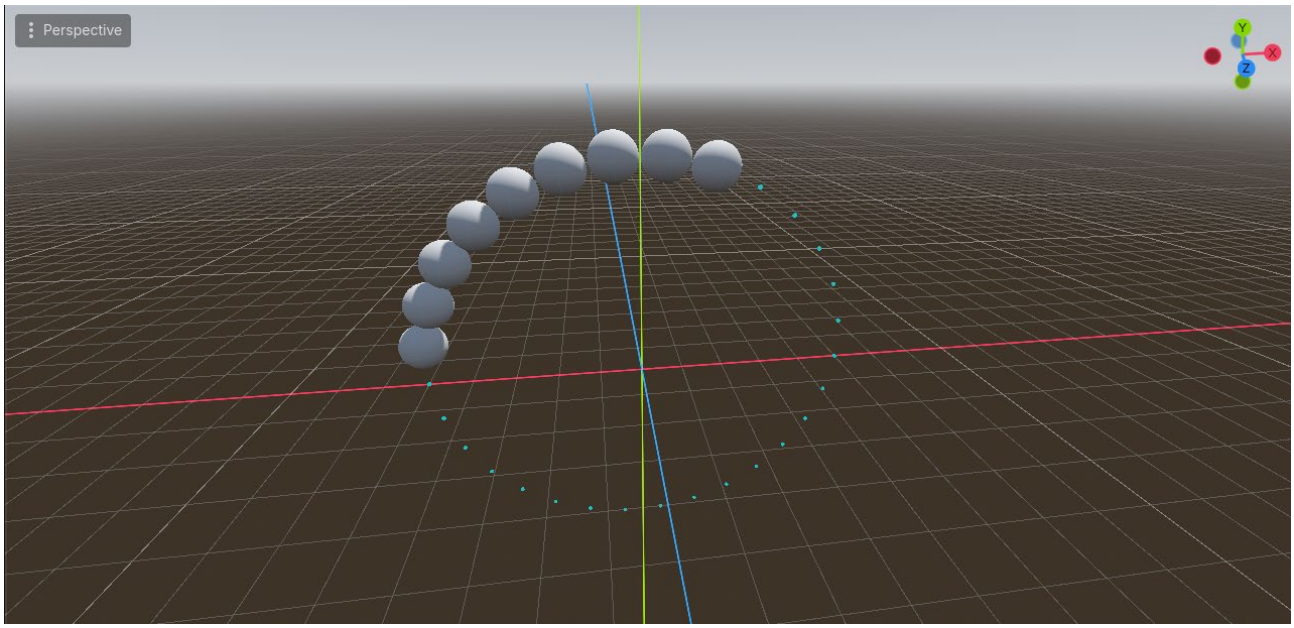


Рис. 3.6 RingPatter3D з поступовим створенням куль

3.3.5 CubePattern3D

CubePattern3D — цн один із двух комплексних шаблонів, який створює абстрактний куб з 8 вершинами як основними кулями та оціональними гранями, які випускають кулі в залежності від їх кількості. Кількість куль випущених з грані напряму залежить від велечини Amount класу PatternBase3D. Також куб можна обертати у 3D прсторі за допомогою велечини Rotation Degrees (Vector3).

Із усіх фігур в плагіні, куб є накомплекснішим у виконанні, оскільки в обрахування координат включаються початкові позиції 8 вершин, 12 граней вджовж яких роміщуються точки та обертання цих точок навколо осей XYZ. Але при цьому клас використовує функцію points_on_line() з LinePattern3D.

Спочатку, обраховуються 8 вершин куба методом створення списку вершин та додавання точок до списку з відступом, що дорівнює половині довжини:

```
# Bottom face (z = -hal_length.z)
verts.append(Vector3(-hal_length.x, -hal_length.y, -hal_length.z))  # 0
verts.append(Vector3( hal_length.x, -hal_length.y, -hal_length.z))  # 1
verts.append(Vector3( hal_length.x,  hal_length.y, -hal_length.z))  # 2
verts.append(Vector3(-hal_length.x,  hal_length.y, -hal_length.z))  # 3

# Top face (z = +hal_length.z)
verts.append(Vector3(-hal_length.x, -hal_length.y,  hal_length.z))  # 4
verts.append(Vector3( hal_length.x, -hal_length.y,  hal_length.z))  # 5
verts.append(Vector3( hal_length.x,  hal_length.y,  hal_length.z))  # 6
verts.append(Vector3(-hal_length.x,  hal_length.y,  hal_length.z))  # 7
```

Щоб зробити центр куба мати ті ж координати що і центр шаблону, при визначенні вершин точок, приймається відступ, довжиною з половини довжини куба взовж осі. Ще одною причиною використання половин довжин для відступів зберігання довжини сторін куба до велечин, заданих в Інспекторі (див. Рис. 3.7). Зсув куба до ценрту шаблону робить зміни в довжинах сторін більш натуральним і комфортним для використання шаблону та редагування в рушії.

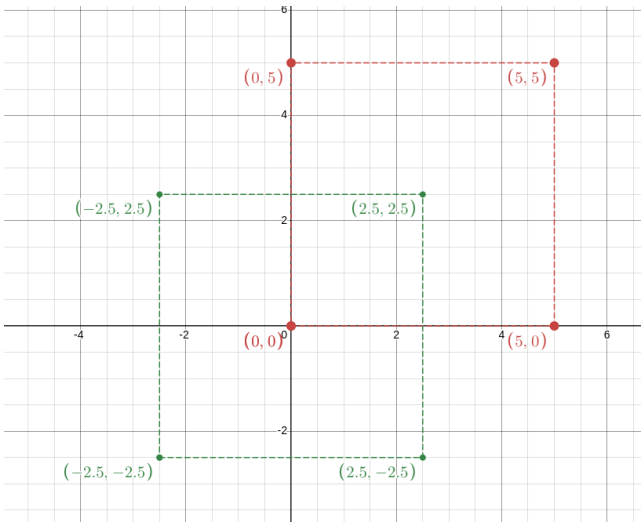


Рис. 3.7 Демонстрація централізації куба до нульових координат

Після визначення вершин куба, та позитивному значення відображення та обрахунок граней куба. Оскільки, вже відомо вершини куба з попередніх обрахунків, для обрахування точок вздовж гранец, використовується функція points_on_line() та додавання точок вершин, до списку точок створення куль.
Для обрахування обертання куба використовується функції apply_rotation(), що створюють список повернутих точок та окреме обертання цих точок навколо XYZ через функції rotate_around_axis(). Функція rotate_around_axis() виконує основні обрахунки обертання куба, методом конвертації градусів у радіани та нормалізації кутів обертання. Після нормалізації та конветрації кутів, використовується фомула Родрігеса:

$$\vec{v}_{rot} = \vec{v}\cos\theta + (\vec{k} \times \vec{v})\sin\theta + \vec{k}(\vec{k} \cdot \vec{v})(1 - \cos\theta) \quad (3.2)$$

де k — вісь обертання, v — ветор у , - кут повернення вектору навколо осі.

Обрахунок виконується за принципом обертання проєкції вектора на градусів по площині, що перпендикулярна осі k. Ця формула використовується при рендерах комп’ютерної графіки, особливо, коли виконуються тривимірні обертання об’єктів. Після виконаних обрахунків, клас отримує новий список точок для візуалізації через _update_editor_visualization().

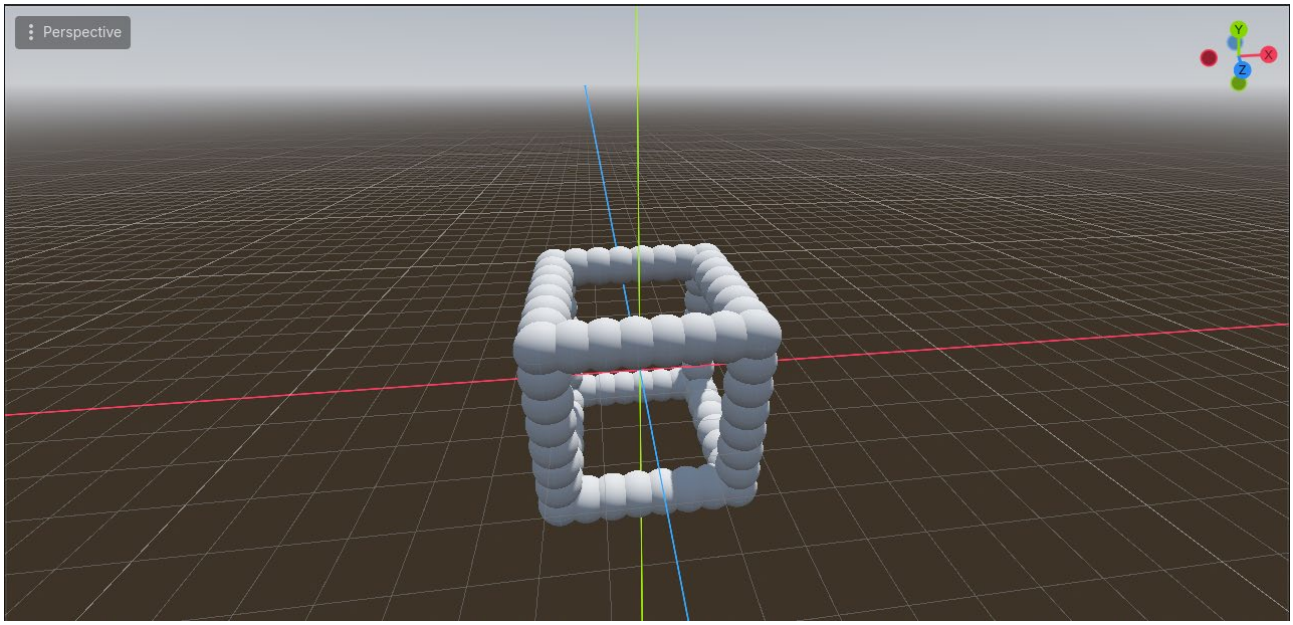


Рис. 3.8 CubePattern3D у Редакторі

3.3.6 SpherePattern3D

SpherePattern3D — це другий комплексний шаблон, який бере два числа для визначення кількості розрізів по вертикалі і горизонталі сфери (Divisions: Vector2) в додаток до радіусу сфери (Radius). Цей шаблон, як і RingPattern3D, має вього лиш одну функцію, оскільки для розрахунку сфери також використовується одна формула сферичних координат (див. Форм. 3.3 -3.7):

$$xx = rrvvvvvvvvvvss,ssrr \quad (3.3)$$
$$yy = rrvvssssvvvvss,ssrr \quad (3.4)$$
$$zz = rrvvvvvvrr \quad (3.5)$$

де x, y, z — координати точки яка обраховується, θ — це добуток π на відношення номеру вершини відносно кількості горизонтальних граней, а ϕ - добуток 2π на відношення номеру вершини відносно кількості вертикальних граней:

$$vv = \frac{ss}{rr_{xx}} * \pi\pi \quad (3.6)$$
$$rr = \frac{ss}{rr_{yy}} * 2\pi\pi \quad (3.7)$$

де i — це номер вершини відносно одного з розрізів по горизонталі (x) та вертикалі (y). rr_{xx} - кількість розрізів по горизонталі, rr_{yy} - кількість розрізів по вертикалі.

Після виконання обрахунків усіх вершин сфери та занесення координат до списку точок, _update_editor_visualization() оновлює відображення точок в редакторі.

Ці чотири дочірні шаблони є стартовою точкою в розробці модульної системи шаблонів. Саме ці чотири шаблони є найпростішими шаблонами при розробці ігор жанру Shmup і можуть бути використані поза межами жанру гри. Перевага цієї системи це висока гнучкість, яка дозволяє створювати велику кількість унікальних атак, використовуючи мінімум ресурсів.

При цьому, слід зауважити, що одних інструкцій щодо запуску куль буває недостатньо щоб змусити загадані кулі рухатись по сцені.

3.3.7 Система поведінки куль

Як було зазначено раніше, кожна атака чи перешкода скаладається з двух компонентів: шаблона та поведінки куль. Така модульна система спроєктована ддля того, щоб можна було легко замінювати частини асету без необхідності створювати асет наново. Шаблонна частина слідує за місцем свторення і швидкості

створення куль, коли поведінкова частина відповідає за власне поведінку самх куль, швидкості їхнього польоту та направлення їхнього польоту. Поведінковий модуль комплементує шаблонний модуль, бо саме поведінка кулі впливає на основну роботу шаблона призначенням властивостей кулі, яку створює шаблон. Шаблон може працювати без додаткового налаштування шаблонів у сцені (використовуючи базову dafault поведінку). При цьому, ресурси поведінки не можуть існувати незалежно у сцені, оскільки вони не описують кулі, до яких вони прив’язані.

Система поведінки куль складається з трьох частин: скрипт кулі (bullet_script), список послідовності поведінки (Bullet Behaviour), та ресурси поведінки (Bullet Move). Скрипт кулі відповідає за поєднання шаблону і кулі методом прив’язування скрипту і списку послідовності під час створення кулі. Список послідовності відповідає за організацію рухів куль на протязі її існування.

3.3.8 Ресурси BulletBehavior та BulletMove

Русурси в Godot — це абстрактні об’єкти, які мають функціонал зберігання данних і виконання коду. Вони використовуються як замінюючі об’єкти, яких можна міняти як значення. Наприклад, якщо у персонажа є ресурс зброї з кодом, який дозволяє стріляти та переряджати зброю, розробник може створити дочірні ресурси які наслідують цей код і мають конкретні значення зброї. Ресурси дозволяють робити системи модульними, для створення нового типу об’єкту, достатньо унаслідувати ресурс з новим значеннями.

В контексті цього плагіну, ресурси стають контейнерами, які тримають в собі інформацію про поведінку кулі під час польоту. Зокрема, за це відповідають ресурси BulletBehaviour та BulletMove. Вони є незалежними один від одного ресурсами, тому вони обидва наслідуються від базового Resource, що відповідає за саму логіку ресурсів. Єдиний зв’язок який вони мають, це сам список послідовності BulletBehaviour який приймає лише значення BulletMove. Оскільки ресурси по власній суті є контейнерами для данних, їх можна використовувати як окремі значення або змінні, які можна організовувати в інших ресурсах.

BulletMove в свою чергу, є більш комплексним ресурсом ніж BulletBehaviour, оскільки цей ресурс містить в собі інформацію про поведінку кулі під час певного періоду. Ресурс в собі має інформацію про швидкість (Velocity), час певного періоду поведіки кулі (Lifetime), та особливу варіацію для типу напряму кулі (AimAt). Ця змінна має лише два значення: направлення на інший об’єкт в сцені (Node) або направлення по вектору (Direction).

Тип направлення визначається за допомогою функції Godot _get_property_list(). Ця функція повертає список значень об’єкту, які видимі в меню Інспектора об’єкту. Цю функцію можна використовувати для більшої інтерактивності з Інспектором та додавання нових функцій до значень. В скрипті BulletMove значення цілі на об’єкт (Aim_target) та вектор направлення (Direction) не мають тегу @export. Ці два значення можна вважати прихованими, оскільки вони не відображаються в Інспекторі. Але при цьому, в функції _get_property_list() можна виставити умову на додавання цих параметрів, залежачи від типу напряму кулі AimAt і додавати потрібне поле залежно від обраного типу:

func _get_property_list():

```
var properties = []
# Show Aim_target only when aim_at is "Node"
if aim_at == "Node":
	properties.append({
		"name": "Converge",
		"type": TYPE_BOOL,
	})
	properties.append({
		"name": "Aim_target",
		"type": TYPE_OBJECT,
		"usage": PROPERTY_USAGE_EDITOR,
		"hint": PROPERTY_HINT_NODE_TYPE,
		"hint_string": "Node3D"
	})
# Show direction only when aim_at is "Direction"
if aim_at == "Direction":
	properties.append({
		"name": "direction",
		"type": TYPE_VECTOR3,
		"usage": PROPERTY_USAGE_DEFAULT | PROPERTY_USAGE_EDITOR
	})
return properties
```

Це дозволяє змінювати поля в інспекторі прямо під час налаштовування об’єкту, що робить налаштування поведінки кулі більш зручним завдяки компактності полей. Інтерактивність полей введення є дуже корисним прийомом, оскільки це дозволяє записувати велику кількість параметрів при лише кількох полях або підтримувати модульність об’єктів.

3.3.9 Скрипт кулі Bullet.gd

Саме Bullet.gd є зв’язуючим скриптом між шаблоном та послідовністю поведіки під час створення процесу кулі. Яак було зазначено раніше, скрипт прив’язується разом з BulletBehaviour до кожної кулі під час її ініціалізації. Сам скрипт має доступ до цього списку послідовності і саме у цьому скрипті проходить логіка обрахування траєкторії польоту і самого пересування куль. Двома основними функціями є _process() та calculate_direction():

```
func _process(delta: float) -> void:
	if behaviour == null or behaviour.BulletMoves.is_empty():
		return

	# Recalculate direction every frame if homing
	if move.aim_at == "Node" and move.Aim_target != null:
		if move.Converge:
			move_dir = (move.Aim_target.global_position - global_position).normalized()
		else:
			move_dir = (global_position - move.Aim_target.global_position).normalized()

	global_translate(move_dir * delta * move.velocity)
	rotate_y(move.rotation_speed * delta)

	_move_age += delta
	if _move_age >= move.lifetime:
		_move_index += 1
		if _move_index >= behaviour.BulletMoves.size():
			queue_free()
			return
		switch_moves(_move_index)
		_move_age = 0.0
```

У функції _process() відбувається основна логіка основного руху, де спочатку перевіряється чи присутні дані у шаблоні. Якщо шаблон має якісь значення BulletMove, запускається основна логіка. Якщо кулі цілять в якійсь об’єкт, то йде додаткова перевірка на збіжність траєкторій польоту (Converge). Якщо Converge є

позитивним, то кулі збігаються в одну точку, яка зазвичай є центорм об’єкту, до якої вони цілять. Але якщо Converge є негативним, то кулі навпаки, розбігаються від центру наданої точки, але при цьому летять в її сторону, як на рисунку 3.6.

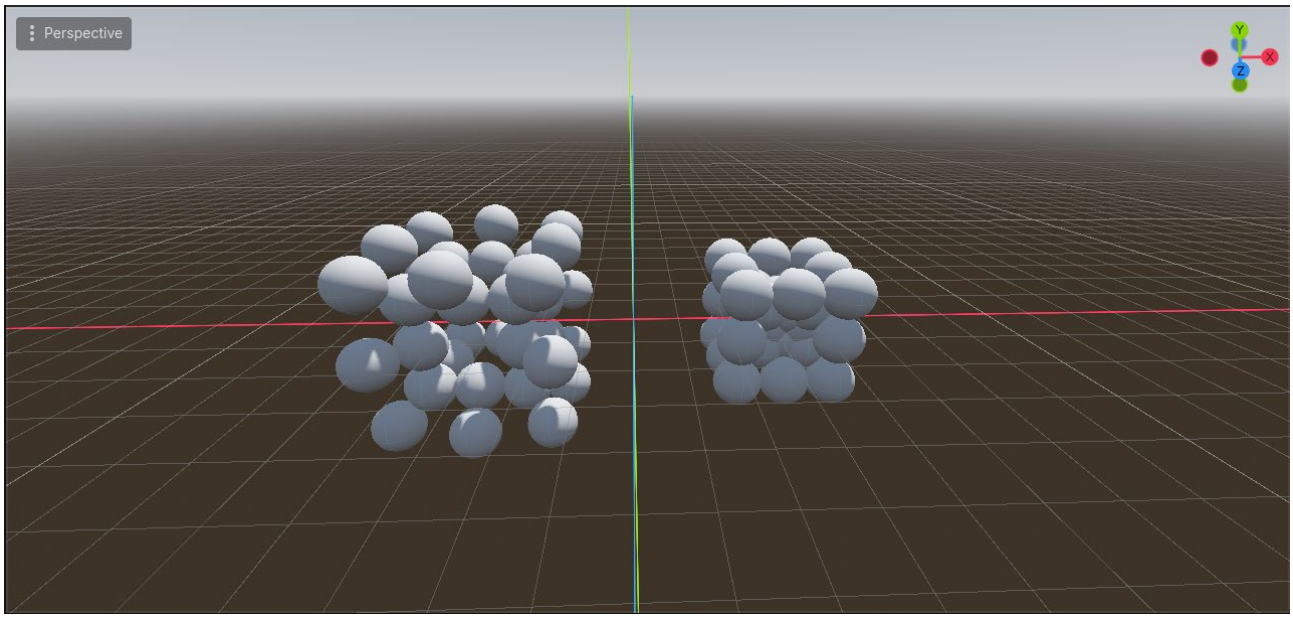


Рис. 3.6 Різниця логічних значень Converge

Для пересування самої кулі у просторі є два методи: зміна координтат чи використання зовнішніх сил. Обидва варанти мають свої застосування та переваги в певних ситуаціях. Якщо куля має падати вниз під силою гравітації (наприклад гармата), до для цієї задачі краще підійде метод зовнішніх сил, де куля є динамічним об’єктом зі своїм об’ємом та масою.

На цей динамічний об’єкт накладаються сили тяжіння та сила самого запуску кулі щоб змусити її рухітись. Метод зміни координат підходить для більш прямолінійних куль, до яких не діють зовнішні сили і мають власні правила пересування. Якщо умовно, ракета, яка летить по прямій лінії, рухається за допомогою методу зміни координат. Метод зміни координат підходить для куль, які летять за передбачуваною координатою, коли метод зовнішніх сил має свій елемент випадковості.

Пересування куль в скрипті Bullet.gd використовується методом зміни координат через функцію global_translate(). Функція приймає значення Vector3, та

приміняє її як відступ від оригінальної позиції об’єкта. Оскільки _process() викликається кожен кадр, ця функція робить ілюзію рухаючого об’єкта, коли насправді кожен кадр приміняється відступ від початкової позиції об’єкта. Для застосування цієї функції береться добуток вектору напряму Direction (або move_dir якщо ціллю є інший об’єкт), число delta що позначає час з першого кадру та та швидкість кулі Velocity:

Після того як виконались усі потрібні діїї для пресування, виконується збільшення пройденого часу під час цього ругу за допомогою move_age += delta. Якщо move_age досяг значення часу дії руху, відбувається перехід на наступний рух. По-перше ідекс руху збільшується на один і відбувається перевірка розмірку списку послідовності. Якщо число індексу входить до цього списку, запускається функція зміни руху switch_moves(), де застосовуються нові зміни для руху кулі і перераховуються направлення кулі через функцію calculate_direction(). Після того як змінився рух кулі, час періоду move_age скидується до нуля і процес починається заново.

Функція calculate_direction() є другою основною функціією скрипту кулі, яка бере на себе обрахунок нової траєкторії польоту. У функції є дві умови: Перша перевіряє тип направлення між вектором і ціллю. Якщо типом направлення є вектор, то направленням стає нормалізований вектор, який визначений в BulletMove. Нормалізація — це процесс приведення вектору до рівнопропорційних значень методом ділення компонентів XYZ вектора на його довжину, щоб запобігти нерівномірної швидкості по діагоналям, або як в випадку шаблону, щоб кулі мали свою власну траєкторію (див. Рис 3.7).

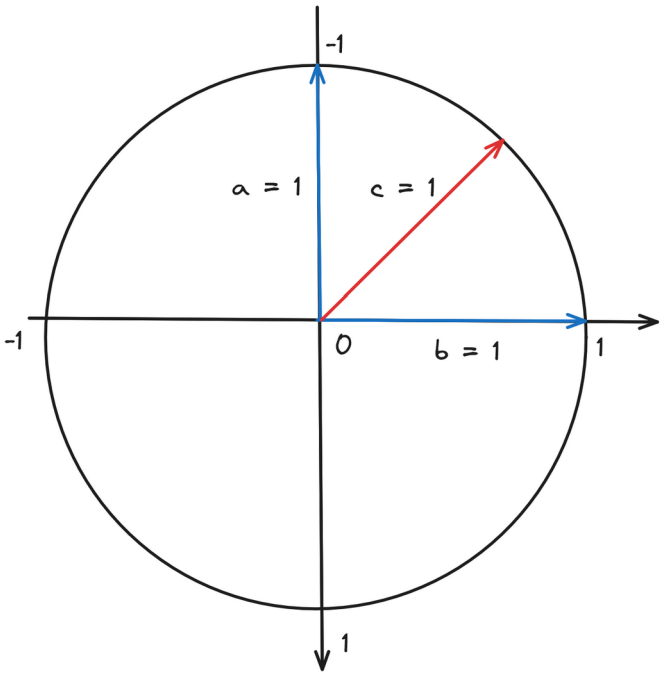


Рис 3.7 Нормалізація Вектора

В іншому випадку, якщо типом направлення є об’єкт, розраховується вектор методом різниці центру цілі об’єкта, в яку летить куля та самого розташування кулі, після чого вектор нормалізується. Таким чином виходить що усі кулі будуть збігатися в цетрі цілі, за умови якщо у цілі нема власної колізії або рухається (якщо Coverge є позитивним). Таким методом можна імітувати «вибух» шаблону, якщо назначити ціллю шаблону самого себе.

```
func calculate_direction() -> void:
	if move.aim_at == "Direction":
		move_dir = move.direction.normalized()
	else:
		if move.Aim_target != null:
			# Fly toward the target
			move_dir = (move.Aim_target.global_position - global_position).normalized()
		else:
			# No target — shoot inward toward pattern origin (null sentinel behaviour)
			move_dir = (global_position — get_parent().global_position).normalized()
```

4 ТЕСТУВАННЯ ТА ВИЗНАЧЕННЯ ПЛАНУ РОЗВИТКУ

Тестування плагіну зазвичай вібувається прямо під час розробки самого забезпечення. Рушій Godot дозволяє активно тестувати код усередині програми завдяки власним інструметам для тестування. До цього також включається і аннотація @tool, що дозволяє запускати код усередині редактора рушія. Ця функція корисна не тільки для самого геймдизайну, але і дозволяє активно виправляти баги під час внесення нових функцій.

Одним із прикладів є тестуванн під час зміни поведінки певної механіки. Якщо на початку розробки програми, одна з частин програми виконувалась одним чином (допустимо, обрахунок направлення польоту обраховувався в коді шаблону), при створенні нового коду чи адаптації старого коду може призвести до певних конфліктів зі старим кодом (певні функції не працюють або працюють некоректно). В таких випадках було визначено метод активного тестування асетів плагіну під час розробки.

Як приклад, буде взято кейс міграції обрахунку направлення польоту кулі. Перед створенням концептів BulletBehaviour та BulletMove, функція calculate_direction() знаходилась у PatternBase3D, яка мала додаткові дані, які відповідали за швидкість, направлення руху, та життя кулі. Цієї системи було достатньо для створення простих шаблонів атак, бо такий процес не вимагав від себе модульності. Проблема виникла в потребі цієї самої модульності після перегляду геймплею Shmup ігор, де атаки були комплесними і багатоступеневими. Із-за цієї потреби в модульності поведінки кулі, створились концепти BulletBehaviour та BulletMove.

Перед початком мігрування коду, слід зробити тестову сцену, в якій можна вільно розміщувати шаблони атак і коригувати зроблені шаблони під час тестування. Сцена не повинна мати надто складною, достатньо мати площину, по якій можна пересуватись персонажем в грі (якщо він є) і розмітки для замірювань. Для цього кейсу було зроблено невелику сітчасту площину з перешкодами і

додоано інтерфейс для перегляду статистик дебагінгу всередині гри, як на рисунку 4.1.

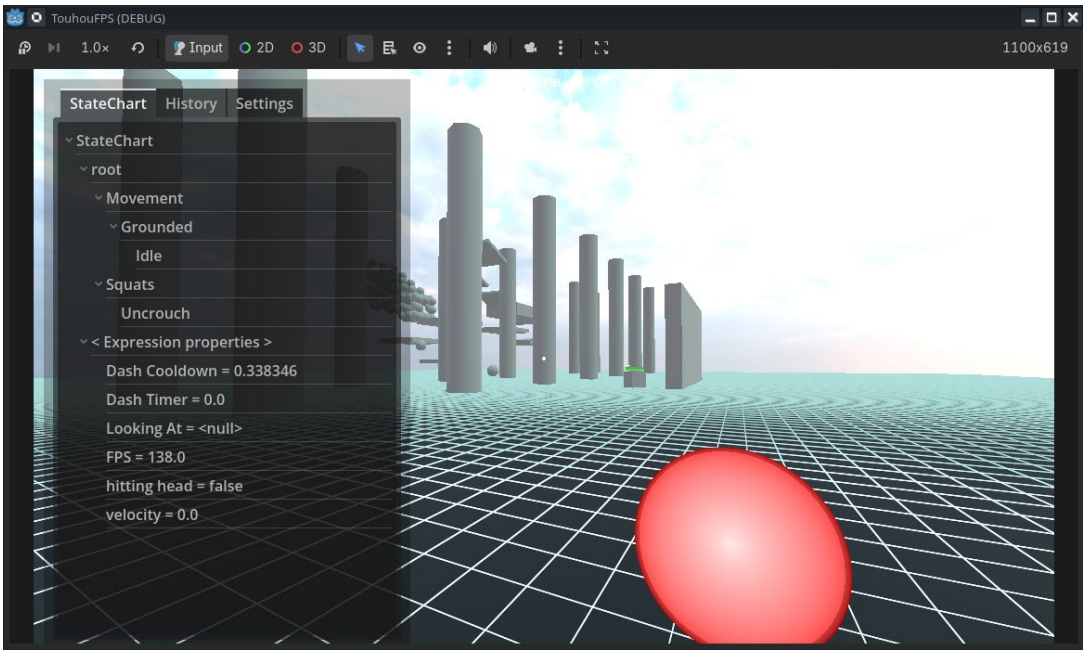


Рис. 4.1 Вікно гри в сцені для тестування

4.1 Інтерфейс статистик у грі

Інтерфейси для перегляду тестувальних статистик може допомагати у багатьох випадках, від тестування певних механік гри, як руз персонажа, так і загальна стабільність гри. Одним із прикладів тестових інтерфейсів є консоль — вікно терміналу усередині гри, яка дозволяє переглядати певну інформацію у грі або налаштовувати певні параметри, які недоступні в звичайних налаштуваннях під себе. Але оскільки проект доволі малий по розмірам і не має широких налаштувань на даному етапі розробці гри, достатньо просто мати невелике вікно для показу базових статистик: стан персонажу, швидкість персонажа, на який об’єкт дивиться, FPS, та ін.

Приклад наведено на рисунку 4.2.

Рис. 4.2 Вікно консолі в Team Fortress 2

Для розробки плагіну, подібні вікна дозволяють визначити, наскільки русурсномісткі процеси створення куль можуть бути. Такі шаблони як лінія та кільце не займають багато навантаження на роботу гри, коли комплексні атаки як сфера та куб при великому навантажені куль можуть понизити FPS навіть на 20 кадрів, що для деяких девайсів може бути катастрофічною втратою продуктивності гри і може негативно плинути на враження від гри.

Слід зауважити що не в усіх випадках результати можуть бути однакові, деякі комп’ютери можуть мати проблеми з обробкою великої кількості об’єктів, коли інші можуть не мати проблем взагалі. Тому рекомедується проводити тестування з іншими користувачами, бажано під час відеозв’язку. Таким чином, розробник буде бачити реакцію інших користувачів на гру або продукт і виностити певні висновки з тестування.

4.2 Результати тестування

Після виправлення усіх відомих багів в плагіні, було влаштовано невеликий огляд перешкод у грі. Перше що слід помітити, це ресурсозатратність певних шаблонів. Сферичні шаблони виявились набільш тяжкими для обробки, оскільки ці фігури є найкомплекснішими і мають найбільше кутів, з яких можуть вилітати

кулі. Перехід між кулями відбувається без проблем і не виникають логічні помилки, які впливають на гру.

Тестування в редакторі пройшли краще ніж у грі, оскільки кожен раз коли вносяться зміни у шаблон, скрипт автоматично корегує щоб дійні рухи куль співпадали очікуваним. Це означає те, що основною проблемою плагіну є оптимізація та високозатратність створення великої кількості куль. Оскільки плагін розраховується на 3D простір, проблема оптимізації є критичною.

4.3 План для наступних розробок плагіну

Плагін, який був розроблений під час цієї роботи вважається прототипом для перевірки можливості втілення рішення для існуючої проблеми. Для прототипного плагіну достатньо мати лише базові функції які будуть втілювати основний функціоназ ПЗ. Але якщо проект продовжить свою розробку у майбутньому, можна привернути до уваги наступні механіки:

1. Додавання нових опцій для BulletMove — оскільки системо контролю поведінки куль є модульною, є сенс розширити функціонал скрипту кулі та додванні нових даних до BulletMove. Внесення елементу випадковості до куль, обертаюча швидкість куль, тощо.

2. Створення шаблонів з інших шаблонів — деякі атаки можуть створювати нові атаки для більшої гнучкості перешкод. Наприклад, створення нових кільцевих шаблонів вздовж лінії.

3. Лазери як новий типу куль можуть мати власний тип поведінок, оскільки велика частина Shmup ігор також використовують лазери в якості атак. Оптимізації та виправлення нових та старих багів для більш гнучкого та приємного використання плагіном.

ВИСНОВКИ

У процессі виконання дипломної роботи було створено повноцінний прототип плагіну для створення асетів атак та перешкод на основі польоту куль для рушія Godot Engine. Під час виконання проекту було проаналізовано аналогічні рішення для 2D Shoot `em up ігор, виявлено сильні сторони рішень, обрано оптималні інструменти для вирішення проблеми, спроєктовано архітектуру плагіну та реалізовано функціональні компоненти з виконаним тестуванням.

1. Проведено аналіз рішень для інших ігрових рушіїв для створення шаблонів атак, що дозволило краще зрозуміти принципи роботи механіки, очікувані логічні процеси під час створення асетів та скриптів та особливості рішень які допомогли сформолювати функціональні та нефункціональні вимоги програмного забезпечення.

2. Проведено огляд та технологій, які використовуватимуться для вирішення проблеми. Було обрано ігровий рушій Godot Engine, оскільки плагін розроблюється саме для цього ігрового рушія і має усі необхідні інструменти для створення плагіну.

3. Розроблено архітектурну структуру плагіну за припципом двох рівнів взаємодії з плагіном та двох основних компонентів асету: шаблону для створення місь для куль та системи менеджменту поведінки куль під час їхнього польоту

4. Реалізовано основну функціональність: створення об’єктів шаблонів у сцені, перегляд точок створення куль у редакторі, створення послідовності поведінки куль та реалізація модуьної системи шаблонів.

5. Проведено тестування плагіну безпосередньо в ігровому рушії та середовищі гри під час розробки плагіну. Плагін задовільняє усі вимоги програмного забезпечення які були створенні на початку проекту.

6. Розроблено план розширення функціоналу плагіну на наступні версії при можливості розвитку ПЗ у майбутньому. Головною місією цього плану це додавання нових функцій до уже існуючих та оптимізація плагіну.

7. Загалом, усі поставленні задачі було виконано. Результатом роботи став працюючий прототип плагіну, який можна використовувати при розробці ігор з механіками Shoot `em up у 3D.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Коновалов Д.О., Худік Б.О. Розробка плагіну для ігрового рушія з відкритим кодом. Матеріали II Всеукраїнської науково-технічної конференції “Виклики та рішення в програмній інженерії”, 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025. С. 396-398.

2. Коновалов Д.О., Худік Б.О. Розробка плагіну для ігрового рушія з відкритим кодом. Матеріали VII Всеукраїнської науково-технічної конференції “Застосування програмного забезпечення в інформаційно-комунікаційних технологіях”, 23 квітня 2026 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2026. С. 166-169.

ПЕРЕЛІК ПОСИЛАНЬ

1. Why indie games are becoming so popular among gamers - Theology Pathfinder. Theology Pathfinder. URL: https://derekdemars.com/blog/why-indie-games-are-becoming-so-popular-among-gamers/ (date of access: 21.04.2026).

2. P J. What was the Great Video Game Crash of 1983?. *The BugSplat Blog*. URL: https://blog.bugsplat.com/great-video-game-crash-1983/ (date of access: 14.05.2026).

3. After Buying Up Studios, Xbox Says It Doesn't Have The Resources To Run Them - Kotaku. *Kotaku*. URL: https://kotaku.com/xbox-bethesda-studio-closings-arkane-too-many-games-1851464885 (date of access: 14.05.2026).

4. About Unity | The World's Leading Game Engine. *Unity*. URL: https://unity.com/our-company (date of access: 14.05.2026).

5. Showcase - Godot Engine. *Godot Engine*. URL: https://godotengine.org/showcase/ (date of access: 14.05.2026).

6. ANTHONY C. Développer des jeux avec Godot Game Engine. D-BOOKER, 2019. 300 p.

7. Craddock D. L. GameDev Stories : Volume 2: More Interviews about Game Development and Culture. Independently Published, 2019.

8. Futter M. The GameDev Budgeting Handbook: How to finish your game in time and on budget. Bithell Games, 2018. 182 p.

9 . Godot Engine Game Development in 24 Hours, Sams Teach Yourself: The Official Guide to Godot 3.0. Sams Publishing, 2018. 432 p.

10. Harris B. J. Console Wars: Sega, Nintendo, and the Battle that Defined a Generation. HarperCollins Audio and Blackstone Audio, 2014. 1 p.

11. Kalata K. Hardcore Gaming 101 Presents: Japanese Video Game Obscurities. Unbound, 2019. 208 p.

12. Kushner D. Masters of Doom. New York : Random House Publishing Group, 2003.

13. Trkulja M. GD Script: Godot 3. 1 Game Engine. Independently Published, 2019.

14. Godot Docs – 4.6 branch. *Godot Engine documentation*. URL: https://docs.godotengine.org/en/stable/ (date of access: 14.05.2026).

15. BulletUpHell BLAST. Bottled up Studio. Version 4.7.4. DarkPeace, 2024. URL: https://bottled-up-studio.itch.io/godot-bullethell-plugin.

16. Uni bullet hell. Unity Asset Store. Version 1.5.7. West Hill, 2023. URL: https://assetstore.unity.com/packages/tools/integration/uni-bullet-hell-19088.

17. Sparen's danmakufu ph3 tutorials - index. *Andrew Fan's Code Dump: The Embodiment of Code and Hacks (AFCDTECH)*. URL:

https://sparen.github.io/ph3tutorials/ph3tutorials.html (date of access: 21.04.2026).

18. Articles/Rodrigues' rotation/Rodrigues' rotation.md at master · EgoMoose/Articles. *GitHub*. URL:

https://github.com/EgoMoose/Articles/blob/master/Rodrigues'%20rotation/Rodrigues'%20rotation.md (date of access: 21.04.2026).

19. Richardson C. Defining a 3D cube: multiple coordinate spaces. URL: https://craigrichardsonportfolio.wordpress.com/2016/03/04/defining-a-3d-cube-multiple-coordinate-spaces/ (date of access: 01.10.2023).

20. Spherical Coordinates -- from Wolfram MathWorld. *Wolfram MathWorld: The Web's Most Extensive Mathematics Resource*. URL:

https://mathworld.wolfram.com/SphericalCoordinates.html (date of access: 14.05.2026).

21. Aecert Gaming. How to draw lines and points in 3D - godot 4 tutorial, 2022. Youtube. URL: https://www.youtube.com/watch?v=JnrhURF1jgM.(date of access: 01.10.2022).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка плагіну для ігрового рушія з відкритим кодом

Виконав студент 4 курсу
групи ПД-44
Коновалов Дмитро Олексійович
Керівник роботи
доцент кафедри ІПЗ, доктор філософії (PhD) Худік Богдан Олександрович
Київ – 2026

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: спрощення процесу створення перешкод для гравця за рахунок використання принципу ігрових снарядів.

Об’єкт дослідження: процес створення асетів («наборів») перешкод, який базується на поведінці снарядів.

Предмет дослідження: ігровий рушій Godot та методи створення плагінів всередині рушія.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Дослідити аналоги для вивчення методів управління кулями (снарядами);
2. розробити структуру ПЗ з описом поведінки розробника, рівнів доступу та взаємодії плагіна з рушієм;
3. створити прототипну версію плагіну для демонстрації роботи концепту;
4. протестувати розроблене програмне забезпечення та створити план на майбутні ревізії проекту.

3

АНАЛІЗ АНАЛОГІВ

	Uni Bullet Hell	Bullet Up Hell BLAST	Danmakufu	Bullet Cutter 3D (*власна розробка*)
Доступність функціоналу	Доступний лише з магазину плагінів Unity і не має демо-версії	Деякі функції недоступні у демо-версії плагіну	Доступний повний функціонал	Доступний повністю завдяки відкритому коду
Підтримка 3D	Підтримує 2D і 3D	Підтримується лише 2D	Підтримується лише 2D	Підтримка 3D просторів
Простота використання	Середня, потрібно розуміти принципи створення шаблонів	Середня, потрібно розбриратись в налаштуванні типів об'єктів плагіну	Висока, треба знати мову скриптів Danmakufu та вміти працювати без інтерфейсу	Низька, усі функції шаблону доступні в одному об'єкті
Гнучкість асетів	Низька, доступні лише кілька простих налаштувань. Неможливо робити комплексні атаки.	Висока, завдяки великій кількості параметрів та модульній системі шаблонів	Висока, завдяки власній мові написання скриптів	Висока, задяки модульній системі об'єктів і параметрів
Допоміжна документація	Відсутня допоміжна документація	Наявна повна документація плагіну і навчальні відео	Повний навчальний блог використання рушію з демонстраціями роботи коду і питаннями для перевірки знань	Допоміжна документація на сторінці проєкту

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні:

1. наявність базових шаблонів для створення асетів перешкод (лінія, кільце, куб, сфера);
2. наявність опцій для показу точок створення куль в редакторі;
3. відображення поведінки польоту куль безпосередньо у редакторі (без запуску гри) для тестування;
4. можливість редагування шаблонів в редакторі без втручання в код плагіну.

Нефункціональні:

1. процес створення об’єктів шаблонів в редакторі (створення об’єкту, зміна параметрів, тестування та коригування);
2. використання плагіну без додаткової конфігурації плагіну (лише процес встановлення та увімкнення в редакторі);
3. незалежність від іншого допоміжного програмного забезпечення для можливості використання у пустому проєкті.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Godot:

- Рушій для розробки ігор;
- влсний IDE;
- GDScript (модифікований Python);
- Відкритий код.

GIMP:

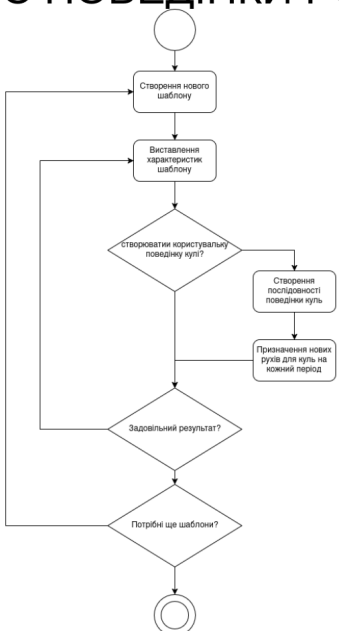
- Альтернативний графічний редактор;
- відкритий код;
- у дипломному проекті використовувався для дизайну піктограм (icons).

Git:

- Система контролю версій;
- функціонал для індивідуальних та групових робіт;
- використання системних файлів для документації (README.md).

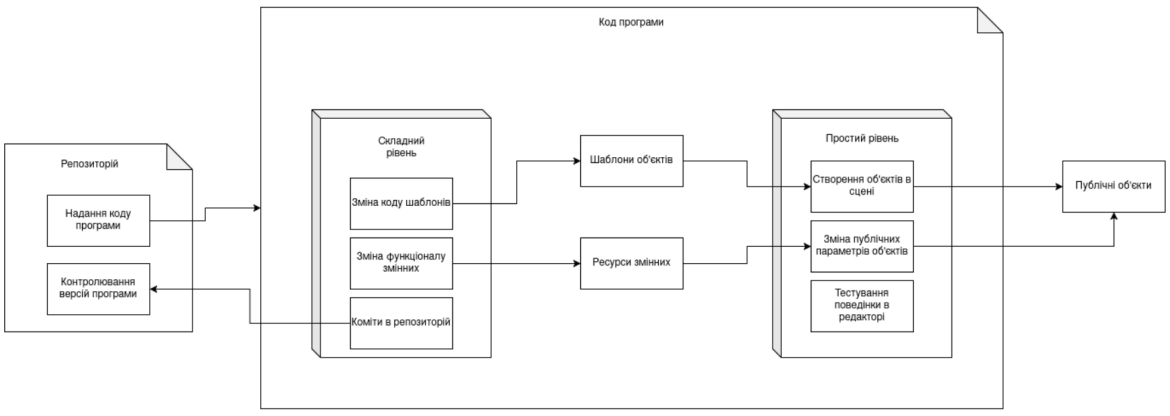
6

ОПИС ПОВЕДІНКИ РОРОБНИКА



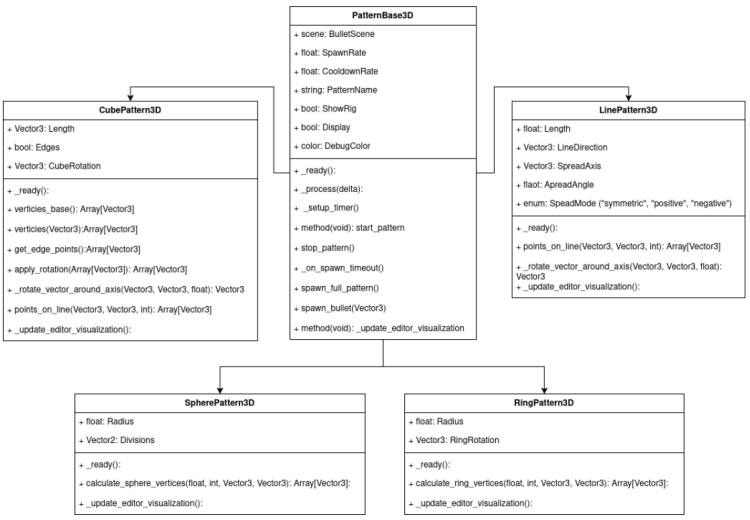
7

ОПИС РІВНІВ ВЗАЄМОДІЇ



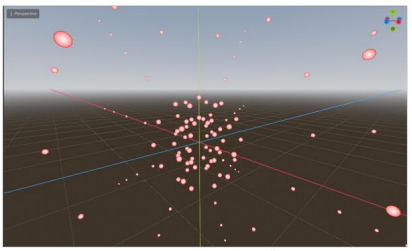
8

ОПИС КЛАСІВ ШАБЛОНІВ

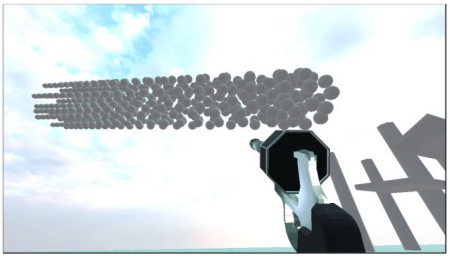


9

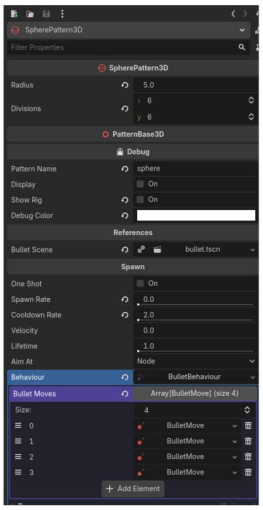
ЕКРАННІ ФОРМИ



Демонстрація шаблону в редакторі



Вигляд шаблону у грі



Налаштування об’єкту в Інспекторі

11

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

- Коновалов Д.О., Худік Б.О. РОЗРОБКА ПЛАГІНУ ДЛЯ ІГРОВОГО РУШІЯ З ВІДКРИТИМ КОДОМ. Матеріали II Всеукраїнської науково-технічної конференції “Виклики та рішення в програмній інженерії”, 26 листопада 2025 року, Київ, Державний Університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025. С 396-398.

- Коновалов Д.О., Худік Б.О. РОЗРОБКА ПЛАГІНУ ДЛЯ ІГРОВОГО РУШІЯ З ВІДКРИТИМ КОДОМ. Матеріали VII Всеукраїнської науково-технічної конференції “Застосування програмного забезпечення в інформаційно-комунікаційних технологіях”, 23 квітня 2026 року, Київ, Державний Університет інформаційно-комунікаційних технологій. Збірник тез. (подано до друку).

12

ВИСНОВКИ

1. Проведено ознайомлення з існуючими технологіями та методиками розробки перешкод в ігрових рушіях, виділено їхні сильні та слабкі сторони;
2. Розроблено схему взаємодії розробника з плагіном в ігровому рушії на основі двох рівнів доступу, з включенням доступу до репозиторію та опису циклу використання плагіна;
3. Створено прототипну версію плагіну та опублікована в окремому репозиторії проєкту;
4. Протестовано плагін під час процесу дизайну ворога та створено попередній план покращень для наступних версій програми.

13

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

debug_drawer.gd

```
# Props to https://www.youtube.com/@AecertGaming for this piece of code
@tool
extends Node

func line(pos1: Vector3, pos2: Vector3, color = Color.WHITE_SMOKE, persist_ms = 0):
	var mesh_instance := MeshInstance3D.new()
	var immediate_mesh := ImmediateMesh.new()
	var material := ORMMaterial3D.new()

	mesh_instance.mesh = immediate_mesh
	mesh_instance.cast_shadow = GeometryInstance3D.SHADOW_CASTING_SETTING_OFF

	immediate_mesh.surface_begin(Mesh.PRIMITIVE_LINES, material)
	immediate_mesh.surface_add_vertex(pos1)
	immediate_mesh.surface_add_vertex(pos2)
	immediate_mesh.surface_end()

	material.shading_mode = BaseMaterial3D.SHADING_MODE_UNSHADED
	material.albedo_color = color

	return await final_cleanup(mesh_instance, persist_ms)

func point(pos: Vector3, radius = 0.05, color = Color.WHITE_SMOKE, persist_ms = 0):
	var mesh_instance := MeshInstance3D.new()
	var sphere_mesh := SphereMesh.new()
	var material := ORMMaterial3D.new()

	mesh_instance.mesh = sphere_mesh
	mesh_instance.cast_shadow = GeometryInstance3D.SHADOW_CASTING_SETTING_OFF
	mesh_instance.position = pos

	sphere_mesh.radius = radius
	sphere_mesh.height = radius*2
	sphere_mesh.material = material

	material.shading_mode = BaseMaterial3D.SHADING_MODE_UNSHADED
	material.albedo_color = color

	return await final_cleanup(mesh_instance, persist_ms)

func square(pos: Vector3, size: Vector2, color = Color.WHITE_SMOKE, persist_ms = 0):
	var mesh_instance := MeshInstance3D.new()
	var box_mesh := BoxMesh.new()
	var material := ORMMaterial3D.new()

	mesh_instance.mesh = box_mesh
	mesh_instance.cast_shadow = GeometryInstance3D.SHADOW_CASTING_SETTING_OFF
	mesh_instance.position = pos

	box_mesh.size = Vector3(size.x, size.y, 1)
	box_mesh.material = material

	material.shading_mode = BaseMaterial3D.SHADING_MODE_UNSHADED
	material.albedo_color = color

	return await final_cleanup(mesh_instance, persist_ms)

## 1 -> Lasts ONLY for current physics frame
## >1 -> Lasts X time duration.
## <1 -> Stays indefinitely
func final_cleanup(mesh_instance: MeshInstance3D, persist_ms: float):
	get_tree().get_root().add_child(mesh_instance)
	if persist_ms == 1:
		await get_tree().physics_frame
		mesh_instance.queue_free()
	elif persist_ms > 0:
		await get_tree().create_timer(persist_ms).timeout
		mesh_instance.queue_free()
	else:
		return mesh_instance
```

PatternBase3D.gd

```
@tool
extends Node3D
class_name PatternBase3D

var bullet_script = load("res://addons/bullet_cutter_3d/bullet behaviour/bullet.gd")
var bullet_move: BulletMove = load("res://addons/bullet_cutter_3d/bullet behaviour/default_move.tres")
var deafult_behaviour: BulletBehaviour = load("res://addons/bullet_cutter_3d/bullet behaviour/deafult_behaviour.tres")
var _spawn_timer: Timer


@export_category("Debug")
@export var pattern_name: String
@export var display: bool = false:
	set(value):
```

```
                display = value
                if not display:
                        stop_pattern()
                _update_editor_visualization()
@export var show_rig: bool = true:
        set(value):
                show_rig = value
                _update_editor_visualization()
@export var debug_color: Color = Color.RED:
        set(value):
                debug_color = value
                _update_editor_visualization()
@export_category("References")
@export var bullet_scene: PackedScene =
preload("res://addons/bullet_cutter_3d/default_bullet.tscn")

@export_category("Spawn")
@export var Behaviour: BulletBehaviour
var amount: int = 8:
        set(value):
                amount = value
                _update_editor_visualization()
@export var one_shot: bool = false
@export_range(0,999) var spawn_rate: float = 1.0
@export_range(0,999) var cooldown_rate: float = 1.0
@export var velocity: float

func _get_property_list():
        var properties = []
        if not is_class("SpherePatten3D"):
                properties.append({
                        "name": "amount",
                        "type": TYPE_INT,
                        "usage":
PROPERTY_USAGE_DEFAULT |
PROPERTY_USAGE_EDITOR
                })
        return properties

var _time_acum := 0.0
var _cooldown_timer: float = 0.0
var _bullets_spawned := 0
var _editor_lines: Array[MeshInstance3D] = []
var _editor_points: Array[MeshInstance3D] =[]
var end_pos
var points
var dir
var _next_spawn_time: float = 0.0

var debug_drawer: Node



func debug_loading() -> void:
        var DebugDrawerScript =
preload("res://addons/bullet_cutter_3d/debug_drawer.gd")
        debug_drawer = Node.new()
        debug_drawer.set_script(DebugDrawerScript)
        add_child(debug_drawer)

var _last_global_position: Vector3 = Vector3.INF



func _ready() -> void:
        debug_loading()
        _setup_timer()
        if Engine.is_editor_hint():
                _update_editor_visualization()


func _setup_timer() -> void:
        _spawn_timer = Timer.new()
        add_child(_spawn_timer)
        _spawn_timer.timeout.connect(_on_spawn_timeo
ut)
        _spawn_timer.one_shot = true # We will manually
restart it

func start_pattern() -> void:
        if _spawn_timer.is_stopped():
                _on_spawn_timeout()

func stop_pattern() -> void:
        if _spawn_timer:
                _spawn_timer.stop()
        _bullets_spawned = 0
        _cooldown_timer = 0 # Optional: Reset cooldown
so it's ready to go again

func _on_spawn_timeout() -> void:
        if spawn_rate <= 0:
                # --- INSTANT BURST MODE ---
                spawn_full_pattern()
                _spawn_timer.start(cooldown_rate)
        else:
                # --- GRADUAL MODE ---
                if _bullets_spawned < amount:

        spawn_single_bullet(_bullets_spawned)
                        _bullets_spawned += 1
                        _spawn_timer.start(spawn_rate)
                else:
                        # Burst finished, start cooldown
                        _bullets_spawned = 0
                        # Start next burst immediately
without cooldown delay
                        _spawn_timer.start(0.0)  #
Changed from cooldown_rate to 0.0


func spawn_full_pattern() -> void:
        for p in points:
                spawn_bullet(p)

func spawn_single_bullet(index: int) -> void:
        spawn_bullet(points[index])
        pass

var _last_transform: Transform3D

func _process(delta: float) -> void:
        if Engine.is_editor_hint() and global_transform !=
_last_transform and show_rig:
                _update_editor_visualization()
                _last_transform = global_transform
        if Engine.is_editor_hint() and not display:
```

```
                return

        # Simple "Auto-fire" for testing/preview
        if display or not Engine.is_editor_hint():
                if _spawn_timer.is_stopped() and
_cooldown_timer <= 0:
                        start_pattern()
        # We still need to count down the cooldown before
the next burst can start
        if _cooldown_timer > 0:
                _cooldown_timer -= delta


func spawn_bullet(origin: Vector3 = global_position):
        if not bullet_scene: return

        var bullet = bullet_scene.instantiate()
        bullet.set_script(bullet_script)

        # Give it a fresh behaviour so each bullet tracks its
own index independently
        var b := BulletBehaviour.new()
        b.BulletMoves = Behaviour.BulletMoves  # shared
array ref is fine, moves are read-only
        bullet.behaviour = b
```

LinePattern3D.gd

```
@tool
@icon("res://addons/bullet_cutter_3d/icons/line_icon.png")
extends PatternBase3D
class_name LinePattern3D

#--- LINE
@export var length: float = 2.0:
        set(value):
                length = value
                _update_editor_visualization()

@export var line_direction: Vector3:
        set(value):
                line_direction = value
                _update_editor_visualization()

#--- SPREAD (ANGLE-BASED)
@export var spread_axis: Vector3 = Vector3.UP:
        set(value):
                spread_axis = value.normalized() if
value.length() > 0 else Vector3.UP
                _update_editor_visualization()

@export var spread_angle: float = 0.0:  # In degrees
        set(value):
                spread_angle = value
                _update_editor_visualization()

@export var spread_mode: String = "symmetric":  #
"symmetric", "positive", "negative"
        set(value):
                spread_mode = value
                _update_editor_visualization()

var spawn_rate_discrease: float = 0.0
        get_parent().add_child(bullet)
        bullet.global_position = origin


func _update_editor_visualization() -> void:
        if not Engine.is_editor_hint():
                return
        # Clear existing visualization
        for line in _editor_lines:
                if is_instance_valid(line):
                        line.queue_free()
        for point in _editor_points:
                if is_instance_valid(point):
                        point.queue_free()
        _editor_lines.clear()
        _editor_points.clear()

        if not show_rig:
                return

func _exit_tree():
        show_rig = false
        _update_editor_visualization()




func _ready() -> void:
        debug_loading()
        super()
        points = points_on_line(global_position, end_pos,
amount)
        if Engine.is_editor_hint():
                _update_editor_visualization()


func points_on_line(start: Vector3, end: Vector3,
num_points: int) -> Array[Vector3]:
        var points: Array[Vector3] = []

        if num_points <= 0:
                return points

        if num_points == 1:
                points.append(start.lerp(end, 0.5))
                return points

        for i in range(num_points):
                var t = float(i) / float(num_points - 1)  #
0.0 to 1.0
                points.append(start.lerp(end, t))

        return points



func _rotate_vector_around_axis(vector: Vector3, axis:
Vector3, angle_degrees: float) -> Vector3:
        """
        Rotates a vector around an axis by the specified
```

```
angle (in degrees).
	Uses Rodrigues' rotation formula.
	"""
	if axis.length() == 0 or angle_degrees == 0:
		return vector

	var axis_normalized = axis.normalized()
	var angle_rad = deg_to_rad(angle_degrees)

	# Rodrigues' rotation formula:
	# v_rot = v*cos(θ) + (k×v)*sin(θ) + k*(k·v)*(1-
cos(θ))
	var cos_angle = cos(angle_rad)
	var sin_angle = sin(angle_rad)

	var term1 = vector * cos_angle
	var term2 = axis_normalized.cross(vector) *
sin_angle
	var term3 = axis_normalized *
axis_normalized.dot(vector) * (1 - cos_angle)

	return term1 + term2 + term3
```

RingPattern3D.gd

```
@tool
@icon("res://addons/bullet_cutter_3d/icons/ring_icon.png")
extends PatternBase3D
class_name RingPattern3D

#--- RING
@export var radius: float = 1.0:
	set(value):
		radius = value
		if is_node_ready():
			_update_editor_visualization()
@export var ring_rotation: Vector3:
	set(value):
		ring_rotation = value
		if is_node_ready():
			_update_editor_visualization()


func _ready() -> void:
	super()
	debug_loading()
	points = calculate_ring_vertices(radius, amount,
global_position, ring_rotation)
	if Engine.is_editor_hint():
		_update_editor_visualization()



func calculate_ring_vertices(radius: float, num_vertices:
int, position: Vector3, rotation: Vector3) -> Array:
	var result: Array = []  # local array instead of
```

CubePattern3D.gd

```
modifying points directly

	var rot_rad = rotation * PI / 180.0
	var basis = Basis()
	basis = basis.rotated(Vector3(1, 0, 0), rot_rad.x)
	basis = basis.rotated(Vector3(0, 1, 0), rot_rad.y)
	basis = basis.rotated(Vector3(0, 0, 1), rot_rad.z)

	for i in range(num_vertices):
		var angle = (2.0 * PI * i) / num_vertices
		var local_pos = Vector3(
			radius * cos(angle),
			radius * sin(angle),
			0.0
		)
		result.append(basis * local_pos +
position)

	return result  # ← actually return it

func _update_editor_visualization() -> void:
	super()

	if not Engine.is_editor_hint() or not show_rig:
		return
	points = calculate_ring_vertices(radius, amount,
global_position, ring_rotation)
	for point in points:

	_editor_points.append(debug_drawer.point(point,
0.05, debug_color, 0))
```

```
func _update_editor_visualization() -> void:
	super()
	end_pos = global_position +
line_direction.normalized() * length

	if not Engine.is_editor_hint() or not show_rig:
		return

	points = points_on_line(global_position, end_pos,
amount)
	for i in range(points.size()):
		var point = points[i]

	_editor_points.append(debug_drawer.point(point,
0.05, debug_color, 0))

	var axis_start = global_position
	var axis_end = axis_start +
spread_axis.normalized() * length * 0.5
	_editor_lines.append(debug_drawer.line(axis_start
, axis_end, Color.YELLOW, 0))
```

```
@tool
@icon("res://addons/bullet_cutter_3d/icons/cube_icon.png"
)
extends PatternBase3D
class_name CubePattern3D

#turned out more complex than I expected lol

#--- LINE
@export var length: Vector3:
	set(value):
		length = value
		_update_editor_visualization()

@export var edges: bool:
	set(value):
		edges = value
		_update_editor_visualization()
#idk about that one, maybe implement later
#@export var faces: bool:
	#set(value):
		#faces = value
		#_update_editor_visualization()

#--- ROTATION
@export var rotation_deg: Vector3:
	set(value):
		rotation_deg = value
		_update_editor_visualization()

func _ready() -> void:
	#loading the editor stuff
	debug_loading()
	super()

	#define if we want add edges
	if edges:
		points = get_edge_points()
	else:
		points = verticies(global_position)
	if Engine.is_editor_hint():
		_update_editor_visualization()

func verticies_base() -> Array[Vector3]:
	# Returns an array of 8 verticies based on the
lengths set on xyz
	var verts: Array[Vector3] = []

	# Halving the length to center the cube
	var hal_length = length / 2

	# Define all 8 vertices as LOCAL OFFSETS (not
world positions)
	# This is important so rotation happens around
origin (0,0,0)
	# Bottom face (z = -hal_length.z)
	verts.append(Vector3(-hal_length.x, -hal_length.y,
-hal_length.z))  # 0
	verts.append(Vector3( hal_length.x, -hal_length.y,
-hal_length.z))  # 1
	verts.append(Vector3( hal_length.x,  hal_length.y,
-hal_length.z))  # 2
	verts.append(Vector3(-hal_length.x,  hal_length.y,
-hal_length.z))  # 3

	# Top face (z = +hal_length.z)
	verts.append(Vector3(-hal_length.x, -hal_length.y,
hal_length.z))  # 4
	verts.append(Vector3( hal_length.x, -hal_length.y,
hal_length.z))  # 5
	verts.append(Vector3( hal_length.x,  hal_length.y,
hal_length.z))  # 6
	verts.append(Vector3(-hal_length.x,  hal_length.y,
hal_length.z))  # 7

	return verts

func verticies(origin: Vector3) -> Array[Vector3]:
	# Taking the verticies we made and apply the
rotation to them
	var base_verts = verticies_base()
	var rotated_verts = apply_rotation(base_verts)

	# making a separate array for the points we rotated
	var world_verts: Array[Vector3] = []
	for vert in rotated_verts:
		world_verts.append(origin + vert)

	return world_verts

func get_edge_points() -> Array[Vector3]:
	# Generates extra points based on ammount of
bullets we point in BasePattern3D
	var verts = verticies(global_position)
	var edge_points: Array[Vector3] = []

	# Define the 12 edges as pairs of vertex indices
	var edges_list = [
		# Bottom face edges
		[0, 1],
		[1, 2],
		[2, 3],
		[3, 0],

		# Top face edges
		[4, 5],
		[5, 6],
		[6, 7],
		[7, 4],

		# Vertical edges
		[0, 4],
		[1, 5],
		[2, 6],
		[3, 7],
	]

	# For each edge, generate points along it
	for edge in edges_list:
		var start = verts[edge[0]]
		var end = verts[edge[1]]
		var points_on_edge =
points_on_line(start, end, amount)
		# Skip the first point on subsequent edges
to avoid duplicates at corners
		if edge_points.size() > 0:
```

```
                points_on_edge = points_on_edge.slice(1)

        edge_points.append_array(points_on_edge)

        return edge_points

func apply_rotation(vertices: Array[Vector3]) -> Array[Vector3]:
        # Apply rotation to vertices around the anchor point (0,0,0 in local space)
        var rotated: Array[Vector3] = []

        for vertex in vertices:
                var v = vertex

                # Rotate around X axis
                if rotation_deg.x != 0:
                        v = rotate_around_axis(v, Vector3.RIGHT, rotation_deg.x)

                # Rotate around Y axis
                if rotation_deg.y != 0:
                        v = rotate_around_axis(v, Vector3.UP, rotation_deg.y)

                # Rotate around Z axis
                if rotation_deg.z != 0:
                        v = rotate_around_axis(v, Vector3.FORWARD, rotation_deg.z)

                rotated.append(v)

        # Returning it back to verticies()
        return rotated

func rotate_around_axis(vector: Vector3, axis: Vector3, angle_degrees: float) -> Vector3:
        # Rotates a vector around an axis by the specified angle (in degrees).
        # Uses Rodrigues' rotation formula.

        # Skipping if there's a zero
        if axis.length() == 0 or angle_degrees == 0:
                return vector

        # Normalizing and converting to radeons
        var axis_normalized = axis.normalized()
        var angle_rad = deg_to_rad(angle_degrees)

        # Rodrigues' rotation formula:
        # v_rot = v*cos(θ) + (k×v)*sin(θ) + k*(k·v)*(1-cos(θ))
        var cos_angle = cos(angle_rad)
        var sin_angle = sin(angle_rad)
        var term1 = vector * cos_angle
        var term2 = axis_normalized.cross(vector) * sin_angle
        var term3 = axis_normalized * axis_normalized.dot(vector) * (1 - cos_angle)

        return term1 + term2 + term3

func points_on_line(start: Vector3, end: Vector3, num_points: int) -> Array[Vector3]:
        # Generates evenly spaced points between start and end (same as LinePattern3D)
        var line_points: Array[Vector3] = []

        if num_points <= 0:
                return line_points

        if num_points == 1:
                line_points.append(start.lerp(end, 0.5))
                return line_points

        for i in range(num_points):
                var t = float(i) / float(num_points - 1)  # 0.0 to 1.0
                line_points.append(start.lerp(end, t))

        return line_points


func _update_editor_visualization() -> void:
        super()
        if not Engine.is_editor_hint() or not show_rig:
                return

        var verts = verticies(global_position)

        # Draw corner points
        for vert in verts:

        _editor_points.append(debug_drawer.point(vert, 0.05, debug_color, 0))



        # Draw edge points if enabled
        if edges:
                var edge_pts = get_edge_points()
                for pt in edge_pts:

        _editor_points.append(debug_drawer.point(pt, 0.03, debug_color, 0))
                points = edge_pts
```

SpherePattern3D.gd

```
@tool
@icon("res://addons/bullet_cutter_3d/icons/sphere_icon.png")
extends PatternBase3D
```

```
class_name SpherePattern3D

var div_limit: int = 5

@export var radius: float = 2:
	set(value):
		radius = value
		_update_editor_visualization()


@export var divisions: Vector2i = Vector2(5,5):
	set(value):
		if value.x < div_limit: value.x = div_limit
		if value.y < div_limit: value.y = div_limit
		divisions = value
		_update_editor_visualization()

func _ready() -> void:
	super()
	debug_loading()
	points = calculate_sphere_vertices(radius, amount,
global_position, divisions)
	if Engine.is_editor_hint():
		_update_editor_visualization()

func calculate_sphere_vertices(radius: float, amount: int,
origin: Vector3, div: Vector2i):
	var sp_rot: Vector2 = div
	var sp_points = []
	var sin_tethra
	var cos_tethra
	var phi
	var sin_phi
	var cos_phi

	var vert = Vector3.ZERO
```

BulletMove.gd

```
@tool
@icon("res://addons/bullet_cutter_3d/icons/bullet_icon.png
")
extends Resource
class_name BulletMove

@export var velocity: float = 0.0
@export var lifetime: float = 1.0
@export var rotation_speed: float = 0.0

@export_enum ("Node", "Direction") var aim_at: String =
"Node":
	set(value):
		aim_at = value
		notify_property_list_changed()
var Aim_target: Node3D = null:
	set(value):
		if aim_at == "Node":
			Aim_target = value
			direction = Vector3.ZERO
		else:
			Aim_target = null
var Converge: bool = true
```

```
	for lat in range(0, sp_rot.x):
		var tethra = (lat/sp_rot.x) * PI
		sin_tethra = sin(tethra)
		cos_tethra = cos(tethra)

		for lon in range(0, sp_rot.y):
			phi = (lon/sp_rot.y)*2*PI
			sin_phi = sin(phi)
			cos_phi = cos(phi)

			vert.x = origin.x + radius *
sin_tethra * cos_phi
			vert.y = origin.y + radius *
cos_tethra
			vert.z = origin.z + radius *
sin_tethra * sin_phi

			sp_points.append(vert)


	return sp_points


func _update_editor_visualization() -> void:
	super()
	if not Engine.is_editor_hint() or not show_rig:
		return
	points =
calculate_sphere_vertices(radius,amount,global_position,di
visions)
	for point in points:

	_editor_points.append(debug_drawer.point(point,
0.05, debug_color, 0))
```

```
var direction: Vector3:
	set(value):
		if aim_at == "Direction":
			direction = value
			Aim_target = null
		else :
			direction = Vector3.ZERO


func _get_property_list():
	var properties = []
	# Show Aim_target only when aim_at is "Node"
	if aim_at == "Node":
		properties.append({
			"name": "Converge",
			"type": TYPE_BOOL,
		})
		properties.append({
			"name": "Aim_target",
			"type": TYPE_OBJECT,
			"usage":
PROPERTY_USAGE_EDITOR,
			"hint":
PROPERTY_HINT_NODE_TYPE,
```

```
                        "hint_string": "Node3D"
                    })
        # Show direction only when aim_at is "Direction"
        if aim_at == "Direction":
                properties.append({
                        "name": "direction",
                        "type": TYPE_VECTOR3,
                        "usage": PROPERTY_USAGE_DEFAULT | PROPERTY_USAGE_EDITOR
                    })
        return properties

var _age := 0
.0
```

Bullet.gd

```
@tool
extends Node3D
class_name Bullet

var behaviour: BulletBehaviour  # replaces single `move`

var _move_index := 0
var _move_age := 0.0
var move: BulletMove

var move_dir: Vector3

func _ready() -> void:
        if behaviour == null or behaviour.BulletMoves.is_empty():
                queue_free()
                return
        move = behaviour.BulletMoves[0]
        calculate_direction()



func calculate_direction() -> void:
        if move.aim_at == "Direction":
                move_dir = move.direction.normalized()
        else:
                if move.Aim_target != null:
                        # Fly toward the target
                        move_dir = (move.Aim_target.global_position - global_position).normalized()
                else:
                        # No target — shoot inward toward pattern origin (null sentinel behaviour)
                        move_dir = (global_position - get_parent().global_position).normalized()


func switch_moves(ind: int) -> void:
        move = behaviour.BulletMoves[ind]  # ← no `var`, assigns to instance variable
        calculate_direction()


func _process(delta: float) -> void:
        if behaviour == null or behaviour.BulletMoves.is_empty():
                return

        # Recalculate direction every frame if homing
        if move.aim_at == "Node" and move.Aim_target != null:
                if move.Converge:
                        move_dir = (move.Aim_target.global_position - global_position).normalized()
                else:
                        move_dir = (global_position - move.Aim_target.global_position).normalized()

        global_translate(move_dir * delta * move.velocity)
        rotate_y(move.rotation_speed * delta)

        _move_age += delta
        if _move_age >= move.lifetime:
                _move_index += 1
                if _move_index >= behaviour.BulletMoves.size():
                        queue_free()
                        return
                switch_moves(_move_index)
                _move_age = 0.0
```

BulletBehaviour.gd

```
@tool
@icon("res://addons/bullet_cutter_3d/icons/bulletbuh_icon.png")
extends Resource

class_name BulletBehaviour


@export var BulletMoves: Array[BulletMove] = []
```