НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри

Комп'ютерних систем, мереж та кібербезпеки

_____ Касаткін Д.Ю., к.пед.н., доц.

(підпис) (ПІБ, вчене звання і ступінь)

«___»_____2025 р.

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

На тему: «Розроблення комп'ютерної системи автоматизації процесу підготовки персоналу критичних сфер безпеки»

Спеціальність 123 «Комп'ютерна інженерія»

Гарант освітньої програми

к.фіз.-мат.н., доц. _____ / Нікітенко Є.В. /

(підпис) (ПІБ)

Керівник дипломного проекту: _____ / Шкарупило В.В. /

(підпис) (ПІБ)

Виконав: _____ / Олексійчук О.Ю./

(підпис) (ПІБ)

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

«ЗАТВЕРДЖУЮ»
завідувач кафедри
комп'ютерних систем, мереж та кібербезпеки
_____/ Касаткін Д.Ю., к.пед.н., доц. /
(підпис) (ПІБ, вчене звання і ступінь)
«__» _____ 20__ р.

З А В Д А Н Н Я

ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ БАКАЛАВРСЬКОЇ СТУДЕНТУ

Олексійчука Олексія Юрійовича
(прізвище, ім'я, по батькові)
Спеціальність (напрям підготовки): комп'ютерна інженерія
Тема кваліфікаційної бакалаврської роботи: «Розроблення комп'ютерної системи автоматизації процесу підготовки персоналу критичних сфер безпеки»
затверджена наказом ректора НУБіП України від "16" 12 2024 р. № 2250 "С"
Термін подання завершеної роботи на кафедру _____
Вихідні дані до кваліфікаційної бакалаврської роботи проектування та розробка комп'ютерної системи автоматизації процесу підготовки процесу підготовки персоналу крити
Перелік питань, що підлягають розробці:

1. Аналіз існуючих систем автоматизації процесу підготовки персоналу
2. Вибір програмного забезпечення для розробки системи автоматизації процесу підготовки персоналу критичних сфер безпеки
3. Розробка програмного забезпечення

Перелік графічного матеріалу (за потреби) _____

Дата видачі завдання "_____" _____2024 р.

Керівник кваліфікаційної роботи _____ _____
(підпис) (прізвище та ініціали)
Завдання прийняв до виконання _____ _____Олексійчук О.Ю.
(підпис) (прізвище та ініціали студента)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Аналіз технічного завдання	14.02.2025 р - 16.02.2025 р	Виконано
2	Проектування системи	01.03.2025 р - 19.03.2025 р	Виконано
3	Реалізація системи	20.04.2025 р - 15.05.2025 р	Виконано
5	Оформлення пояснювальної записки	11.05.2025 р - 20.05.2025 р	Виконано

Студент **О.Ю. Олексійчук**

(підпис) (ініціали та прізвище)

Керівник проекту (роботи) **В.В. Шкарупило**

(підпис) (ініціали та прізвище)

РЕФЕРАТ

Пояснювальна записка: 64 сторінки, 6 рисунків, 11 лістингів, 12 джерел

КОМП'ЮТЕРНА СИСТЕМА, UNITY, C#, FPS, ТРЕНАЖЕР, ГРА-ТРЕНАЖЕР, ІМІТАЦІЯ, ПРИЦІЛЮВАННЯ, НАВЧАННЯ, ВІЗУАЛІЗАЦІЯ, РЕАЛІЗМ

Об'єкт розроблення - процес розроблення комп'ютерної системи автоматизації процесу підготовки персоналу критичних сфер безпеки.

Мета роботи - розроблення комп'ютерної системи автоматизації процесу підготовки персоналу критичних сфер безпеки.

Проект складається з трьох розділів.

Перший розділ присвячено аналізу вимог до програмного забезпечення, огляду існуючих аналогів та визначенню їхніх недоліків.

В другому розділі сформовано обґрунтування вибору платформи, мови програмування, рушія Unity, а також розглянуто основні інструменти, використані для досягнення поставлених цілей.

В третьому розділі представлено реалізацію системи: побудова сцени, розміщення мішеней, логіка стрільби, імплементація FPS-камери, пояснення структури проєкту, частини коду, скріншоти з результатами.

В результаті створено робочий прототип гри-тренажера, придатний для подальшого розширення або адаптації під реальні сценарії навчання.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

C# - Мова програмування, що використовується для створення сценаріїв у середовищі Unity,

FPS - прогама система-тренажер від першої особи (First Person Shooter), жанр комп'ютерних ігор,

GameObject - Об'єкт сцени у рушії Unity,

IDE - Інтегроване середовище розроблення (Integrated Development Environment,

PC - Персональний комп'ютер (Personal Computer),

Raycast - Метод визначення перетинів променя з об'єктами у віртуальному середовищі,

Scene - сцена, ігровий рівень або середовище взаємодії у Unity,

Script - Програмний сценарій, написаний мовою C#,

Target - Мішень для прицілювання у тренажері,

UI - Інтерфейс користувача (User Interface) ,

Unity - Програмне середовище (ігровий рушій) для створення дво- та тривимірних комп'ютерних ігор,

Build - Зібрана (скомпільована) версія програмного продукту для запуску на цільовій платформі.

ВСТУП

Сучасні засоби підготовки персоналу для критичних сфер безпеки все частіше потребують інноваційних рішень, які дозволяють моделювати складні сценарії в безпечних умовах. Одним із таких підходів є використання ігрових технологій та симуляторів, що можуть точно імітувати умови реальної ситуації та вимагати від користувача миттєвої реакції, концентрації та навичок прийняття рішень.

Гра-тренажер від першої особи (FPS) виявивилася особливо зручною для цього завдання. Вона дозволяє зосередити увагу користувача на точності прицілювання, влучності, швидкості реагування та просторовій орієнтації. Тому розроблення прототипу навчального симулятора на базі FPS-механік виглядає як цілком логічне і технічно виправдане рішення.

У цій дипломній роботі розглянуто процес створення комп'ютерної системи, що імітує базові навички стрільби у віртуальному середовищі. Проєкт реалізовано з використанням рушія Unity, що забезпечив гнучке керування об'єктами, зручну роботу зі сценою та можливість швидкої побудови ігрової логіки.

Метою цієї роботи є побудова ігрового прототипу, який може бути використаний як основа для подальшого розвитку більш складних тренажерів або адаптований до конкретних сценаріїв професійної підготовки. Під час розробки основну увагу приділено модульності, простоті підтримки та розширюваності системи.

1 АНАЛІЗ ТЕХНІЧНОГО ЗАВДАННЯ

1.1 Вимоги до розроблення системи

Щоб система навчального симулятора на основі FPS виконувала свою функцію ефективно, необхідно чітко визначити, які саме завдання вона має вирішувати. В даному випадку мова йде про базове тренування стрільби, орієнтації в просторі та взаємодії з простими цілями у віртуальному середовищі. Тобто симулятор повинен забезпечити умови, максимально наближені до тренувального полігону, але в безпечному цифровому форматі.

Враховуючи це, до системи висуваються наступні основні вимоги:

– Можливість точного прицілювання: гравець повинен мати контроль над камерою з відчуттям повного занурення, як у класичних FPS;

– Реалістична механіка пострілу: кожен постріл має вираховуватись з урахуванням напрямку погляду, позиції зброї та можливого розсіювання;

– Взаємодія з мішенями: об'єкти на сцені повинні реагувати на попадання та відображати результат (наприклад, анімацією чи ефектом);

– Візуальна простота: графічна складова не є пріоритетною, тож основна сцена має бути побудована з простих форм, щоб не відволікати гравця;

– Гнучкість та масштабованість: структура проєкту має дозволяти додавання нових функцій або цілей без суттєвої переробки базової логіки;

– Мінімальні технічні вимоги: оскільки тренажер розробляється для персональних комп'ютерів, бажано оптимізувати його для середніх систем.

Ці вимоги є основою для подальшого вибору інструментів, побудови архітектури і всієї логіки взаємодії в проєкті.

1.2 Огляд існуючих систем

Коли стоїть завдання створити симулятор для тренування стрільби, логічно спочатку поглянути, що вже існує. Сучасний ринок пропонує безліч FPS-рішень — від простих шаблонів для Unity до складних військових симуляторів. Частина з них доступна відкрито, інші — комерційні або навіть закриті. Але більшість мають щось спільне: фокус на стрільбі, реакції та просторовій орієнтації.

До прикладу, існують комерційні проєкти, які моделюють стрільбу в умовах полігону або закритих приміщень. Деякі з них реалізовані на рушії Unreal Engine або Unity й мають спеціальні модулі для обробки попадань, візуального зворотного зв'язку, підрахунку результатів. Проте такі системи часто потребують дорогого обладнання, специфічної технічної підтримки або не відкриті для модифікацій.

Unity FPS Microgame

Це, мабуть, найвідоміший стартовий шаблон. Його головна мета — показати новачку як працює базовий FPS в Unity. Є рух, стрільба, мішені. Все просто і доступно.

Переваги: простота запуску, гнучкість, інтеграція з Unity Learn.

Недоліки: мінімальний функціонал, необхідна доробка під конкретні задачі.

3D Aim Trainer

Працює прямо в браузері. Придуманий не для навчання персоналу, а для геймерів, які хочуть покращити прицілювання. Проте його структура добре підходить для тренування реакції та її швидкості.

Переваги: простий доступ, висока швидкість запуску.

Недоліки: немає логіки взаємодії з мішенями, вузький сценарій.

NeoFPS (для Unity)

Платний модуль, який дає змогу побудувати свою програму систему-тренажер майже з нуля, але без написання всього самостійно. Він модульний, має систему зброї, прицілювання, навіть перезарядки.

Переваги: модульна архітектура, готові компоненти зброї, анімації, взаємодії.

Недоліки: ліцензія платна, потрібна технічна підготовка для інтеграції.

Aimlabs

Цей продукт давно використовується геймерами, і хоча його головна мета — тренування точності, по суті це тренажер. В ньому є режими стрільби по мішенях, аналітика влучності, налаштування складності.

Переваги: велика кількість сценаріїв, статистичний аналіз.

Недоліки: не призначений для симуляції умов, наближених до професійної підготовки.

Engagement Skills Trainer (EST)

Використовується військовими, має фізичні модулі, екранні мішені, контроль поведінки гравця. Така реалізація підходить для навчальних центрів, а не для дому.

Переваги: високий рівень реалістичності, багатокористувацьке навчання.

Недоліки: закрита система, висока вартість, не адаптована для цивільного використання.

2 ПРОЕКТУВАННЯ СИСТЕМИ

2.1 Обґрунтування вибору рушія Unity

Перед початком розробки важливо обрати відповідні інструменти, які дозволять ефективно реалізувати всі поставлені функції, зберігаючи при цьому гнучкість та масштабованість проєкту. Враховуючи жанр гри-тренажера, технічні обмеження та обсяг задач, особливу увагу було зосереджено на виборі рушія, що має зручний візуальний редактор, розвинену спільноту та підтримку необхідної функціональності “з коробки”.

2.1.1 Unity як платформа для навчальних симуляторів

Серед сучасних ігрових рушіїв, які активно використовуються для розроблення інтерактивних систем у реальному часі, Unity посідає особливе місце завдяки своїй відкритості, багатофункціональності та широкій підтримці платформ. Обрання саме цього середовища пояснюється сукупністю інженерних переваг, зокрема гнучкою архітектурою, потужним редактором сцен і підтримкою компонентно-орієнтованого програмування, що критично важливо для побудови розширюваної структури FPS-тренажера.

Unity реалізує компонентну архітектуру через концепцію GameObject–Component, що дозволяє кожен елемент сцени описувати як об'єкт із набором поведінкових або фізичних модулів. Кожен компонент є підкласом UnityEngine.Component, що дозволяє доступ до таких життєво важливих інтерфейсів, як Transform, Collider, Rigidbody, а також до високорівневих засобів на кшталт Animator або AudioSource.

Під час ініціалізації сцени рушій послідовно інстанціює об'єкти у вигляді деревоподібної структури, що відповідає ієрархії GameObject'ів. Кожен з них

проходить через життєвий цикл, у межах якого викликаються події Awake(), Start(), Update() і OnDestroy(). Завдяки цим подіям можлива тонка настройка взаємодії між об'єктами, що особливо корисно при реалізації поведінки ворогів, стрільби або фізичних взаємодій у тренажерах.

Одним із визначальних чинників на користь Unity є його інтегрований фізичний рушій PhysX від NVIDIA. Він забезпечує детальну симуляцію зіткнень, сили тяжіння та імпульсів. Наприклад, при реалізації стрільби використовуються методи Physics.Raycast() або Rigidbody.AddForce(), що дозволяє створити правдоподібну взаємодію пострілу з мішенями. Unity автоматично виконує квантування симуляцій на основі FixedUpdate() із заздалегідь визначеним кроком часу, що є критично важливим для узгодженості між логікою гри та фізичним розрахунком.

Крім того, Unity забезпечує розвинену систему обробки вводу. Починаючи з версій 2018+, підтримується як класична система Input.GetAxis(), так і нова Input System на базі подій і дій, що дозволяє абстрагуватися від конкретного типу пристрою (клавіатура, геймпад, VR-контролер). Це забезпечує гнучке керування, яке легко масштабувати під майбутні сценарії — наприклад, додавання режиму віртуального тренування з використанням шолома або контролера.

Не менш важливим є Unity Editor API, який дозволяє створювати кастомні інструменти безпосередньо в редакторі. Це відкриває можливість розробити, наприклад, редактор конфігурацій мішеней або інструмент для візуального сценарію дій під час тренування.

Переваги Unity також включають:

- кросплатформеність (PC, WebGL, VR, Android, iOS, консолі);
- вбудована система освітлення (Baked + Realtime GI);
- підтримка анімації через Mecanim і State Machine;
- редактор матеріалів і постобробки через URP/HDRP;
- інтеграція з CI/CD (через Unity Cloud Build або CLI).

Отже, вибір рушія Unity є технічно виправданим для задачі створення навчального FPS-тренажера. Його архітектура, інструментальні засоби та модульність створюють надійне середовище для розробки, налагодження й подальшого масштабування системи.

2.1.2 Обґрунтування вибору C#

C# (вимовляється сі-шарп) — це об'єктно-орієнтована, типізована мова високого рівня, що була розроблена компанією Microsoft як складова екосистеми .NET. Її архітектура ґрунтується на принципах безпеки, типового контролю, строгого синтаксису, а також підтримки сучасних парадигм: інкапсуляції, успадкування, поліморфізму, делегування та подійного програмування. Саме ця сукупність ознак робить C# стратегічно доцільним вибором для створення надійного, масштабованого та підтримуваного коду в проєктах на базі Unity.

У середовищі Unity C# працює поверх Mono/IL2CPP-інтерпретаторів, що виконують компіляцію проміжного байт-коду (CIL, Common Intermediate Language) у платформозалежний машинний код під час запуску (JIT) або під час збирання (AOT, для iOS/WebGL). При написанні скриптів Unity розробник фактично створює класи, що успадковуються від базового типу MonoBehaviour, який є адаптером до рушія: він викликає зарезервовані методи життєвого циклу (наприклад, Start(), Update(), FixedUpdate(), OnTriggerEnter() тощо).

Завдяки статичній типізації, компілятор C# (Roslyn або mcs) ще на етапі збірки виявляє типові помилки: неініціалізовані змінні, порушення видимості, некоректні операції приведення типів. Це підвищує надійність та дозволяє швидше ідентифікувати дефекти на ранніх етапах.

Однією з ключових переваг C# є підтримка автоматичного керування пам'яттю через Garbage Collector (GC). Звільнення неактивних об'єктів

здійснюється неявно, що суттєво знижує ризик витоків пам'яті, однак вимагає усвідомленого проєктування архітектури — особливо при роботі з анонімними функціями, івентами та делегатами, які можуть утворювати замикаючі контексти.

C# в Unity підтримує інтерфейси (interfaces), події (events), делегати (delegates) та лямбда-функції, що дозволяє використовувати шаблони проєктування, такі як спостерігач (Observer), фабрика (Factory), командний об'єкт (Command) або інверсія керування (IoC). Завдяки цьому реалізація внутрішньої логіки гри може залишатися незалежною, ізольованою та придатною для модульного тестування.

Також важливо зазначити, що C# підтримує атрибутивну систему метаданих, яка широко використовується Unity для серіалізації та створення редакторських інтерфейсів. Наприклад, атрибути SerializeField, HideInInspector, Range() дозволяють розробнику впливати на відображення полів класу в інспекторі редактора без порушення інкапсуляції.

Переваги використання C# у контексті Unity включають:

– інтеграцію з Visual Studio / Rider, включно з IntelliSense, дебагом і рефакторингом;

– підтримку асинхронності через async/await, що дозволяє безпечно керувати затримками, завантаженням ресурсів та мережею;

– розширюваність за рахунок використання Partial класів, Extension методів, Generic-контейнерів;

– захист від небезпечних операцій на відміну від C++: відсутність прямої роботи з покажчиками (окрім unsafe-режиму), перевірка масивів, контроль меж індексації.

З урахуванням наведених особливостей, вибір C# як мови програмування для реалізації системи FPS-тренажера є не лише технологічно обґрунтованим,

але й практично ефективним. Її строгість, сумісність з рушієм Unity, а також здатність до абстрагування та архітектурної масштабованості дозволяють зберігати контроль над логікою проєкту навіть за умови розширення його функціоналу.

2.2 Структура та логіка ігрової сцени

2.2.1 FPS-камера та керування

У тренажерах, створених на основі програми системи-тренажер від першої особи, система камери відіграє ключову роль у формуванні відчуття присутності. Гравець повинен мати змогу швидко оглядатися, орієнтуватися у просторі та прицілюватися без зайвих зусиль. Від цього безпосередньо залежить ефективність тренування.

Класична система огляду від першої особи ґрунтується на поділі керування камерою на два незалежні компоненти: вертикальний рух (нахил камери вгору/вниз) та горизонтальне обертання (поворот усього тіла гравця ліворуч/праворуч). Це дозволяє зберігати стабільність огляду та запобігає виникненню помилок, пов'язаних з некоректною трансформацією координат (наприклад, ефект “перекручування” при надмірному повороті).

У більшості реалізацій камера фізично закріплюється в межах об'єкта гравця — зазвичай як дочірній елемент, який автоматично переміщується разом із ним. Логіка обробки обертання камери реалізується через перехоплення введення користувача (рух миші) та перетворення його на зміну кутів огляду. Щоб уникнути перегляду “через голову”, вертикальний кут зазвичай

обмежується у певному діапазоні (наприклад, від -80° до +80°).

Для гнучкості та масштабованості бажано реалізовувати логіку огляду як окремий модуль або режим, який може бути легко замінений. Такий підхід дозволяє в майбутньому реалізувати додаткові сценарії — наприклад, огляд із кабіни транспорту, кінематографічні камери, або автоматичне слідування за об'єктом.

Параметри чутливості огляду, інверсія осей, початковий напрямок погляду та обмеження обертання — усе це має бути доступним для налаштування через окремі властивості або конфігураційні файли. Це дозволяє адаптувати систему під користувача, його обладнання (миша, трекпад, контролер) та сценарій навчання.

У підсумку, система керування камерою в програмі системі-тренажер від першої особи має бути реалізована як окремий, ізольований функціональний блок, який не залежить напряму від фізики гравця або решти ігрової логіки. Така модульність забезпечує зручність в адаптації, тестуванні та подальшому масштабуванні симулятора.

2.2.2 Побудова ігрового середовища

Ігрове середовище в симуляторах від першої особи відіграє не лише роль візуального оформлення, але й виступає важливою складовою тренувального простору. Його завдання — створити логічну, інтуїтивну й водночас технічно просту зону, у якій користувач зможе відпрацьовувати поставлені дії (прицілювання, орієнтація, стрільба) без зайвих відволікаючих факторів.

У запропонованому рішенні середовище реалізовано як мінімалістичне полігонне поле, що складається з базових геометричних форм (куби, площини) з використанням простих матеріалів. Такий підхід дозволяє не перевантажувати сцену, зберігаючи зосередженість користувача на мішенях і діях.

Основні об'єкти сцени:

– Гравець, що розміщується на стартовій позиції та включає

систему огляду, контролер руху і камеру;

– Мішені або статичні об'єкти, що виконують роль цілей для стрільби (реалізовані через розміщення кубічних примітивів або префабів без складної логіки);

– Елементи взаємодії, зокрема візуальні ефекти (сліди від куль, уламки) — реалізовані через окремі префаби, які спавняться на точці контакту;

– Просторове оформлення, яке включає основну підлогу, стіни, освітлення та камеру, закріплену в ігровому об'єкті.

Позиціонування об'єктів здійснюється без використання процедурної генерації — всі ключові елементи сцени розміщено вручну з урахуванням їх функціонального призначення. Це дає змогу забезпечити контрольоване середовище для перевірки конкретних механік.

Середовище адаптоване для подальшого масштабування: можна розширити поле, додати динамічні цілі, змінити умови освітлення або побудувати сценарії із декількома зонами стрільби без зміни базової архітектури сцени. Отже, середовище відповідає основним вимогам: простота, функціональність та адаптивність.

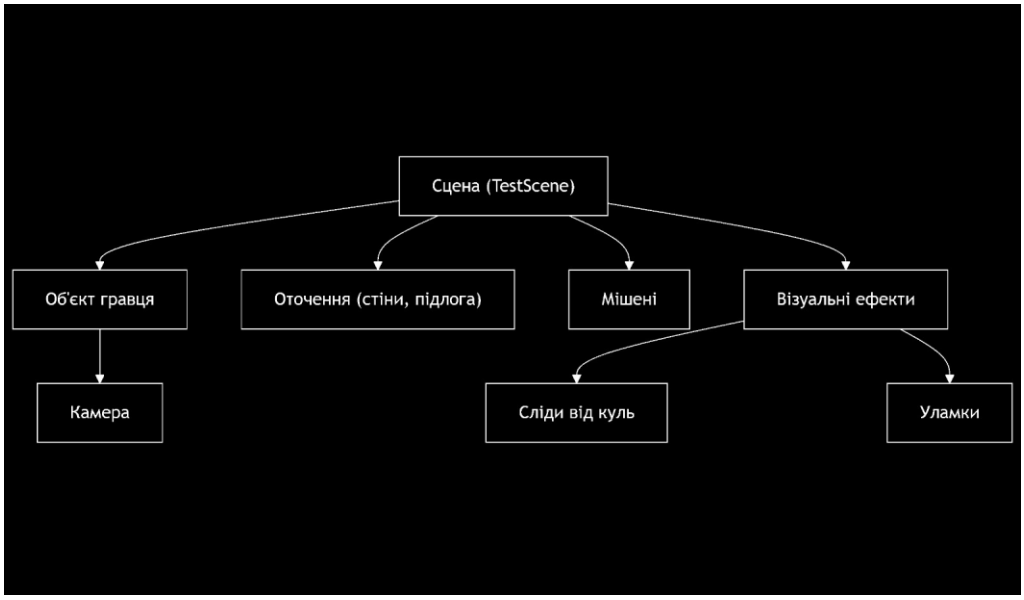

Рисунок 2.1 — блок-схема із компонентами для побудови сцени.

2.3 Алгоритм стрільби

Один із ключових елементів у програмі система-тренажер — це система стрільби. Вона не лише відповідає за запуск снаряду, або моделювання пострілу, але й повинна враховувати реалістичні аспекти: точність, реакцію середовища, обробку боєзапасу, візуальні ефекти тощо. Особливо це важливо в умовах, в яких від користувача очікується виконання конкретних дій — прицілювання, стабілізації погляду та контролю ситуації.

2.3.1 Реалізація механіки пострілу

У рамках симулятора постріл реалізовано як взаємодію кількох ізольованих компонентів: системи керування зброєю, моделі боєприпасів та візуального об'єкта снаряду. Вся логіка стрільби зосереджена в окремому модулі, що реагує на події, пов'язані з натисканням відповідної клавіші або тригера.

Після отримання сигналу на активацію пострілу система ініціює серію дій:

– зменшується кількість доступних набоїв у магазині;

– визначається напрямок вогню з урахуванням поточної похибки (spread);

– створюється новий снаряд, який розміщується у визначеній точці на моделі зброї.

Параметр розсіювання визначається динамічно і залежить від стану гравця. Якщо користувач рухається або стоїть — точність нижча. У разі присідання або нерухомого стану — похибка зменшується. Зміна цього параметра виконується поступово, через згладжування значення з використанням математичної інтерполяції.

Снаряд не створюється через пряме викликання методу Instantiate(), а отримується з пулу об'єктів. Це дозволяє уникнути витрат на часте створення/знищення ресурсів, що є важливим для підтримки стабільної частоти кадрів під час активної стрільби.

Обертання створеного снаряда залежить не лише від положення гравця, а

й від випадкового відхилення, яке симулює реалістичну стрілецьку похибку. Це забезпечує змінну точність стрільби навіть при однаковому напрямку прицілу, імітуючи людський фактор.

Підіб'ємо підсумки, механіка пострілу побудована за принципами автономності, оптимізації та масштабованості. Її структура дозволяє легко адаптувати систему до інших типів зброї або сценаріїв взаємодії, не змінюючи базову архітектуру проєкту.

2.3.2 Налаштування об'єктів та колізій

Щоб симуляція стрільби виглядала переконливо й коректно з точки зору фізики, важливо правильно налаштувати взаємодію між снарядом і середовищем. У подібних системах колізії виконують одразу кілька ролей: вони визначають факт попадання, фіксують точку контакту, запускають візуальні ефекти та можуть впливати на об'єкти в зоні ураження. У проєкті кожен снаряд створюється як фізичний об'єкт, що має власний колайдер і компонент пересування (наприклад, фізичний або логічний рух за напрямком).

При досягненні об'єкта, який має колайдер, ініціюється обробник зіткнення. Цей обробник фіксує точку дотику, напрямок удару, а також — тип об'єкта, з яким відбулось зіткнення. На основі цієї інформації створюється слід від попадання або уламок (наприклад, дірка в поверхні або візуальний відблиск). Також важливою частиною налаштування є правильне маркування об'єктів середовища. Це дозволяє, наприклад, відрізняти мішень від декоративних елементів або встановлювати різну реакцію залежно від матеріалу поверхні (метал, дерево, скло). Така реалізація зазвичай базується на використанні тегів, шарів або типів фізичних матеріалів.

Колізії не лише інформують про попадання — вони також можуть змінювати поведінку об'єкта. Наприклад, мішень може зреагувати на влучання падінням, зміною кольору або зниженням “життів”, якщо реалізована система здоров'я. З технічної точки зору, всі ці процеси виконуються за допомогою подій

зіткнення (наприклад, OnCollisionEnter) або променевої перевірки (Raycast), залежно від типу снаряду (фізичний чи умовний). Такий підхід дозволяє легко масштабувати систему — додаючи нові типи реакцій або типи цілей, не змінюючи базової логіки пострілу.

2.4 Модульність та можливості розширення

Модульний підхід, покладений в основу архітектури програмного комплексу, забезпечує високий ступінь ізольованості між функціональними блоками, що значно спрощує подальше масштабування та адаптацію системи до змінних умов експлуатації. У межах цієї архітектури кожен компонент виконує строго визначену роль і взаємодіє з іншими виключно через публічні інтерфейси або подійні механізми, що дозволяє виключити небажані побічні ефекти при його модифікації. Стрільба, керування камерою, логіка гравця, візуалізація влучань та генерація допоміжних ефектів реалізовані як окремі автономні модулі. Кожен із них функціонує незалежно, що дає змогу не лише змінювати його реалізацію без впливу на інші частини системи, але й забезпечує можливість подальшої інтеграції альтернативних механізмів без зміни структури основного коду.

Ключовим аспектом розширюваності є використання шаблонної обробки подій, яка дозволяє зв'язувати окремі компоненти на основі реактивної логіки. Таким чином, поява нових сценаріїв — наприклад, специфічних режимів тренування, альтернативних типів вогню або особливих форм зворотного зв'язку — не потребує рефакторингу існуючих класів, а лише підключення додаткових обробників або підсистем.

Архітектурна відкритість дає змогу не тільки змінювати поведінку на рівні окремих об'єктів, але й переналаштовувати загальну логіку взаємодії між елементами. Завдяки чітко витриманому принципу інверсії залежностей і використанню абстракцій замість жорстких зв'язків, система лишається стійкою до змін, зберігаючи при цьому прогнозовану поведінку.

Отже, модульна структура реалізованої системи визначає її придатність

для подальшого функціонального збагачення, а також для адаптації під специфіку практичних завдань, які можуть виникнути під час застосування симулятора у різних критичних сферах.

2.5 Архітектура системи

Архітектура розробленої системи побудована за принципами ієрархічної структурованості, компонентної ізоляції та подійно-орієнтованої взаємодії. Такий підхід забезпечує зрозуміле логічне розділення обов'язків між модулями, а також дозволяє підтримувати внутрішню узгодженість при масштабуванні функціональності. Кожен функціональний блок реалізується у вигляді спеціалізованого компонента, що виконує строго визначену роль у рамках загального процесу. Взаємодія між ними здійснюється через абстраговані інтерфейси або подійну систему, яка забезпечує реактивну поведінку компонентів без їхньої прямої залежності одне від одного. Це, у свою чергу, дозволяє уникнути циклічних зв'язків, знизити когезію та підвищити керованість логіки. Центральну частину архітектури становить ігровий об'єкт користувача, який містить модулі контролю руху, управління камерою, обробки дій (зокрема стрільби) та реакцій на зовнішні події.

Ці модулі, попри просторову локалізацію в межах одного об'єкта, реалізовані незалежно та оперують лише визначеними зонами відповідальності. Зокрема, камера не контролює логіку переміщення, а стрілецький модуль не обробляє зіткнення чи інтерфейсні події.

Допоміжні системи — обробка зіткнень, виведення ефектів, керування боєприпасами, реакції мішеней — винесено в окремі підсистеми, які взаємодіють із ядром гри за запитом або через генерацію подій. Наприклад, створення снаряду супроводжується автоматичним запуском сліду від влучання, який реалізовано як підписаний обробник у пулі об'єктів. Загальна структура системи дозволяє віднести її до категорії відкритих архітектур, що підтримують інверсію управління та полегшують інтеграцію нових компонентів без ризику порушення

існуючої логіки. Це створює передумови для використання системи як основи для більш складних симуляторів або адаптації під інші тренувальні задачі.

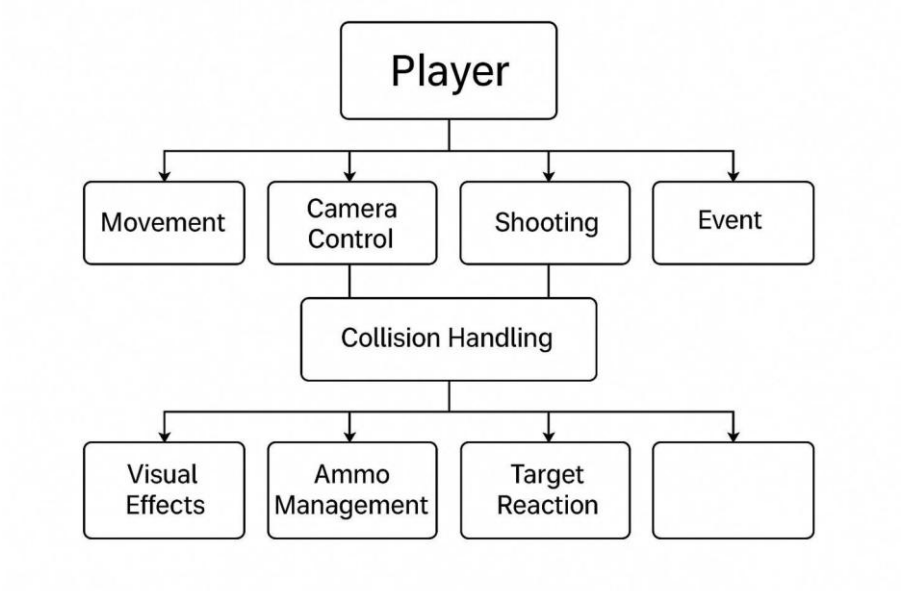

Рисунок 2.2 — Узагальнена архітектура модульної системи симулятора

3 РЕАЛІЗАЦІЯ СИСТЕМИ

Розроблення функціонального прототипу тренажера здійснювалося на основі попередньо спроєктованої модульної архітектури з акцентом на простоту розгортання, логічну структурованість та можливість поступового розширення. Реалізація охоплює як створення інтерактивного середовища, так і технічну імплементацію ключових механік, необхідних для відтворення базового сценарію навчання. В рамках цього етапу було сформовано ігрову сцену з мінімалістичним полігоном, реалізовано логіку переміщення гравця та систему огляду у форматі гри-тренажера від першої особи. Окрема увага приділялася модулю стрільби, в якому поєднано обробку введення, динамічне розрахування точності та реакцію середовища на влучання. Кожен з компонентів створено у вигляді окремої функціональної одиниці з чіткими зонами відповідальності. Такий підхід дозволив досягти стабільної поведінки симулятора навіть за інтенсивного навантаження та забезпечив основу для подальшого

функціонального розширення.

У цьому розділі розглянуто поетапну реалізацію основних елементів гри, наведено ключові фрагменти коду, скріншоти інтерфейсу та результати роботи.

3.1 Створення проєкту в Unity

3.1.1 Налаштування середовища розробки

Проєкт реалізовано у середовищі Unity версії 2018.3.0f2, що забезпечує сумісність зі стандартною архітектурою Unity-програм, підтримку C# 6.0 та доступ до необхідного інструментарію для розробки тривимірних інтерактивних середовищ. Вибір цієї версії обумовлено стабільністю, широкою підтримкою спільноти та відсутністю критичних змін, характерних для більш пізніх релізів.

Для написання скриптів використовувалась IDE Visual Studio, яка інтегрується з Unity та забезпечує зручні засоби для навігації по проєкту, автодоповнення, налагодження та контроль за структурою коду. Структура проєкту є логічно впорядкованою. Основні скрипти зосереджено в директорії: Assets/OsFPS/Code

Усередині неї розподіл реалізовано за функціональними блоками:

– Player: модулі руху, механіки та парамерів гравця;

– Weapons: логіка стрільби, керування зброєю, постріли та ефекти;

– Cameras: (через інтеграцію з модулем UnityTK) — окремі реалізації камерних режимів;

– UI: елементи інтерфейсу користувача.

Кожен функціональний модуль має власну папку зі скриптами та метаданими. Наприклад, логіку стрільби реалізовано в класі

WeaponProjectileShooter, який підписується на події зброї через:

```
this.weapon.weaponFire.onStart += this.OnFire;
```

Таким чином, класи не зв'язані жорстко з ігровим циклом (Update()), а реагують на події, що підвищує гнучкість системи. Префаби, матеріали та графічні ресурси згруповано в директорії:

Assets/OsFPS/Resources Assets/OsFPS/Shading

Сцена, яка використовується як тестове середовище, розташована в Assets/OsFPS/Scenes/TestScene.unity. Саме ця сцена є основною для демонстрації механік, оскільки в ній представлено мішені, початкову позицію гравця та налаштоване оточення.

У системі тегів і шарів (Layers and Tags) реалізовано стандартні позначення для ключових об'єктів (мішень, гравець, інтерфейс), що спрощує керування зіткненнями та видимістю. Для побудови системи огляду та контролю було додатково використано зовнішній модуль UnityTK, який інтегрується через Git-субмодуль. Цей фреймворк забезпечує розширювану архітектуру для роботи з камерою та іншими службовими компонентами. Він дозволяє використовувати типову логіку FPS-огляду, ізольовано від решти системи. Конфігурація проєкту також включає використання object pooling для снарядів, що дозволяє уникнути динамічного створення об'єктів під час стрільби. Приклад створення снаряда з пулу:

```
var p = PrefabPool.instance.GetInstance(this.projectile);
p.transform.position = this.projectileOrigin.position;
```

Цей підхід значно зменшує навантаження на пам'ять при багаторазових пострілах, що особливо важливо в умовах високої частоти оновлень.

Таким чином, налаштування середовища розробки забезпечує чітку організацію проєкту, модульність коду та високий ступінь готовності до подальшої адаптації, розширення або повторного використання компонентів у подібних навчальних системах.

3.1.2 Імплементація FPS-контролера

У межах розробки системи керування від першої особи (FPS-контролера), було реалізовано комплексну ієрархію взаємодіючих компонентів, кожен із яких інкапсулює вузькоспеціалізовану функціональність, що відповідає за окремі аспекти ігрової логіки, пов'язаної з рухом, обертанням, взаємодією та обробкою подій життєвого циклу ігрової сутності гравця.

Контролер гравця побудовано за принципами модульності, з дотриманням інверсії залежностей, що забезпечує високий ступінь гнучкості при масштабуванні функціоналу. Нижче наведено детальний опис функціональних модулів, які безпосередньо формують поведінкову модель гравця.

Компонент FirstPersonController.

Центральним обчислювальним ядром системи введення користувача є компонент FirstPersonController. Він реалізує патерн делегування поведінки до допоміжних сервісів, що дозволяє досягти низької когезії між модулями.

Лістинг 3.1 — Реалізація керування

```
public class FirstPersonController : MonoBehaviour
{
private RigidbodyMotor motor; private FirstPersonLook look;

void Update() {
Vector2 input = new Vector2(Input.GetAxis("Horizontal"),
```

```
Input.GetAxis("Vertical"));
      motor.Move(input); look.HandleLook();
          }
      }
```

Зчитування векторів введення з клавіатури відбувається в координатах локальної площини, після чого дані передаються до модулю RigidbodyMotor — моторного шару обчислень.

Компонент RigidbodyMotor

Механізм просторової трансляції гравця реалізований на основі кінематичної моделі, що враховує фізичні властивості Rigidbody. Нижче подано узагальнену модель руху:

Лістинг 3.2 — Реалізація переміщення користувача

```
public class RigidbodyMotor : MonoBehaviour {
public float moveSpeed = 5f;
private Rigidbody rb;


public void Move(Vector2 input) {
Vector3 velocity = transform.forward * input.y +
transform.right * input.x;
rb.velocity = velocity.normalized * moveSpeed + new
Vector3(0, rb.velocity.y, 0);
}
}
```

У цьому фрагменті реалізовано одночасну підтримку поступального руху та гравітаційного впливу, що імітує реалістичну кінематику тіл у тривимірному просторі.

Компонент FirstPersonLook.

Для забезпечення свободи обертання камери відносно голови ігрової сутності реалізовано компонент FirstPersonLook. Він інкапсулює обробку руху миші, реалізуючи окреме обертання по вісі X (камера) і Y (ігровий об'єкт):

Компонент RigidbodyMotor

Механізм просторової трансляції гравця реалізований на основі кінематичної моделі, що враховує фізичні властивості Rigidbody. Нижче подано узагальнену модель руху:

Лістинг 3.2 — Реалізація переміщення користувача

```
public class RigidbodyMotor : MonoBehaviour {

public float moveSpeed = 5f;
private Rigidbody rb;


public void Move(Vector2 input) {

Vector3 velocity = transform.forward * input.y +
transform.right * input.x;
rb.velocity = velocity.normalized * moveSpeed + new
Vector3(0, rb.velocity.y, 0);
    }

}
```

У цьому фрагменті реалізовано одночасну підтримку поступального руху та гравітаційного впливу, що імітує реалістичну кінематику тіл у тривимірному просторі.

Компонент FirstPersonLook.

Для забезпечення свободи обертання камери відносно голови ігрової сутності реалізовано компонент FirstPersonLook. Він інкапсулює обробку руху миші, реалізуючи окреме обертання по вісі X (камера) і Y (ігровий об'єкт):

Лістинг 3.3 — Рух камери навколо власних осей

```
public class FirstPersonLook : MonoBehaviour {
public float sensitivity = 2f; private float pitch = 0f;

void Update() {
float mouseX = Input.GetAxis("Mouse X")* sensitivity;
float mouseY = Input.GetAxis("Mouse Y") * sensitivity;

pitch -= mouseY;
pitch = Mathf.Clamp(pitch, -85f, 85f);
transform.localEulerAngles = new Vector3(pitch, 0, 0);
transform.parent.Rotate(Vector3.up * mouseX);
    }
}
```

Такий підхід забезпечує повну декомпозицію логіки обертання по вертикалі та горизонталі, що відповідає практикам побудови сучасних FPS-систем.

Компонент FirstPersonEnity.

FirstPersonEnity виступає як інтегруючий клас, який об'єднує фізичне представлення, логіку управління, а також обробники пошкоджень, інвентаризації та взаємодії. Він реалізує парадигму композиції поведінки:

Лістинг 3.4 — Об'єднання різних I/O команд від користувача

```
public class FirstPersonEntity : MonoBehaviour {

public FirstPersonController controller;

public FirstPersonWeaponHandler weaponHandler;

public EntityDamageHandler damageHandler;
void Awake() {
controller = GetComponent<FirstPersonController>();
weaponHandler = GetComponent<FirstPersonWeaponHandler>();
damageHandler = GetComponent<EntityDamageHandler>();

    }

}
```

Цей компонент виступає центральним посередником, що агрегує всі інші модулі, створюючи єдину функціональну сутність.

Компоненти звукової взаємодії та візуального зворотного зв'язку

– EntityFootstepSound та EntityFootsteps відповідають за обробку звукових подій під час пересування.

– EntityDamageHandler відстежує рівень здоров'я, здійснює його декремент та ініціює сценарії смерті.

Ці модулі не лише покращують імерсію користувача, але й формують повноцінну функціональну модель фізіології ігрової сутності.

Обробка зброї через FirstPersonWeaponHandler.

Для реалізації динаміки стрільби використано клас FirstPersonWeaponHandler, що взаємодіє з анімаційною підсистемою, відтворює звукові ефекти та відповідає за стан патронів:

Лістинг 3.5 — Імплементація стільби та ефектів

```
public class FirstPersonWeaponHandler : MonoBehaviour {
```

```
public void Fire() {
    if (CanShoot())
    {
        PlayMuzzleFlash();
        SpawnProjectile();
    }
}
}
```

Його структура дозволяє гнучко розширювати арсенал зброї або додавати нові типи візуального супроводу (наприклад, сліди від куль).

Система інвентаризації: SimpleInventory

Компонент SimpleInventory забезпечує базову логіку зберігання і перемикання об'єктів, які може нести гравець. Його використання дозволяє масштабувати гру за рахунок введення додаткових об'єктів взаємодії без зміни основної архітектури.

Враховуючи все вищесказане, імплементація контролера від першої особи реалізована як система компонентів, що ізольовано обробляють рух, орієнтацію, стрільбу, інвентаризацію, пошкодження та взаємодію з навколишнім середовищем. Така структура відповідає принципам SOLID та дозволяє легко підтримувати, тестувати і розширювати функціонал у межах архітектури Unity.

3.2 Реалізація механіки стрільби

Функціональна підсистема стрільби виконує одну з ключових ролей у контексті інтерактивної симуляції, оскільки забезпечує моделювання реакції на

дію користувача, обробку взаємодії з об'єктами віртуального середовища та генерацію відповідних візуальних і звукових ефектів. Її конструкція прямо впливає як на загальне відчуття керованості в симуляторі, так і на можливість відтворення умовних бойових сценаріїв із заданим рівнем складності. З технічної точки зору механізм стрільби слугує посередником між компонентом введення та системою фізичної моделі. Саме ця підсистема відповідає за фіксацію факту ініціації пострілу, обчислення параметрів вогню з урахуванням стану об'єкта (наприклад, прискорення або положення гравця), створення відповідних снарядів або променів у просторі сцени та обробку результатів їх взаємодії з іншими об'єктами. Реалізація даної механіки побудована з урахуванням принципів модульності та інверсії управління. Логіка пострілу винесена в окремий компонент, що функціонує у взаємодії з загальною системою керування зброєю. Це дозволяє забезпечити ізоляцію відповідальностей, спростити підтримку та відкриває можливості для масштабування — наприклад, додавання різних типів зброї, варіантів боєприпасів або умов застосування.

У наступних підрозділах буде розглянуто деталізовану реалізацію обчислювальної логіки пострілу, а також механізми конфігурації об'єктів середовища, відповідальних за фізичну взаємодію з уражаючими елементами.

3.2.1 Реалізація механіки пострілу

Основна логіка пострілу реалізована в скрипті

WeaponProjectileShooter.cs (Додаток А), який виступає окремим компонентом, прикріпленим до об'єкта зброї. Він взаємодіє з базовим класом Weapon, через який отримує доступ до подій, патронів та точки спавну снарядів. Така побудова дозволяє ізолювати поведінку пострілу від інших частин системи

та легко адаптувати її до різних конфігурацій зброї. Ключовий етап взаємодії — підписка на подію onStart, яка виконується в методі Awake():

Лістинг 3.6 — Налаштування виклику пострілу

```
public void Awake()
{
this.weapon.weaponFire.onStart += this.OnFire;
}
```

Це забезпечує активацію методу OnFire() у момент, коли користувач ініціює постріл (наприклад, натискає кнопку миші). Сам метод OnFire() зменшує кількість набоїв і створює снаряд з урахуванням похибки прицілювання:

Лістинг 3.7 — Зменшення кількості набоїв та похибка прицілювання

```
private void OnFire()
{
this.weapon.ammoInClip--; var euler = this.projectileOrigin.rotation.eulerAngles;
float offset = ((Random.value * 2f) - 1f) / 2f; euler.x += this.spread * offset;
euler.y += this.spread * offset;

var p = PrefabPool.instance.GetInstance(this.projectile);
p.transform.position = this.projectileOrigin.position;
p.transform.rotation = Quaternion.Euler(euler);
}
```

Значення spread, що впливає на точність, динамічно оновлюється в методі Update() залежно від стану гравця. Якщо гравець рухається або стоїть, розсіювання збільшується, при присіданні — зменшується. Для досягнення плавності зміни використовується інтерполяція:

Лістинг 3.8 — Застосування похибки пострілу

```
public void Update()
{
float spreadTarget = this.baseSpread;
if (this.weapon.fpEntity.model.stance.IsCrouching) 
spreadTarget *= this.crouchedSpreadModifier;
if 
(!this.weapon.fpEntity.model.characterController.isGrounded
|| this.weapon.fpEntity.model.characterController.velocity.s
qrMagnitude > 0.5f)
spreadTarget *= this.movingSpreadModifier;

this.spread = Mathf.Lerp(this.spread, spreadTarget, 
Time.deltaTime * this.spreadLerpIntensity);
}
```

Окремої уваги заслуговує спосіб створення снарядів: замість прямого використання Instantiate() застосовується механізм пулінгу (PrefabPool.instance.GetInstance(...)), що значно знижує витрати на створення/знищення об'єктів під час активної стрільби. Це підвищує продуктивність, особливо при високій частоті вогню.

Завдяки чіткій структурі, окремому компоненту для стрільби та параметричній моделі точності, реалізація цієї механіки забезпечує не лише реалістичну поведінку, а й технічну ефективність і зручність масштабування. Компонент WeaponProjectileShooter легко адаптується під інші види зброї або

альтернативні сценарії стрільби.

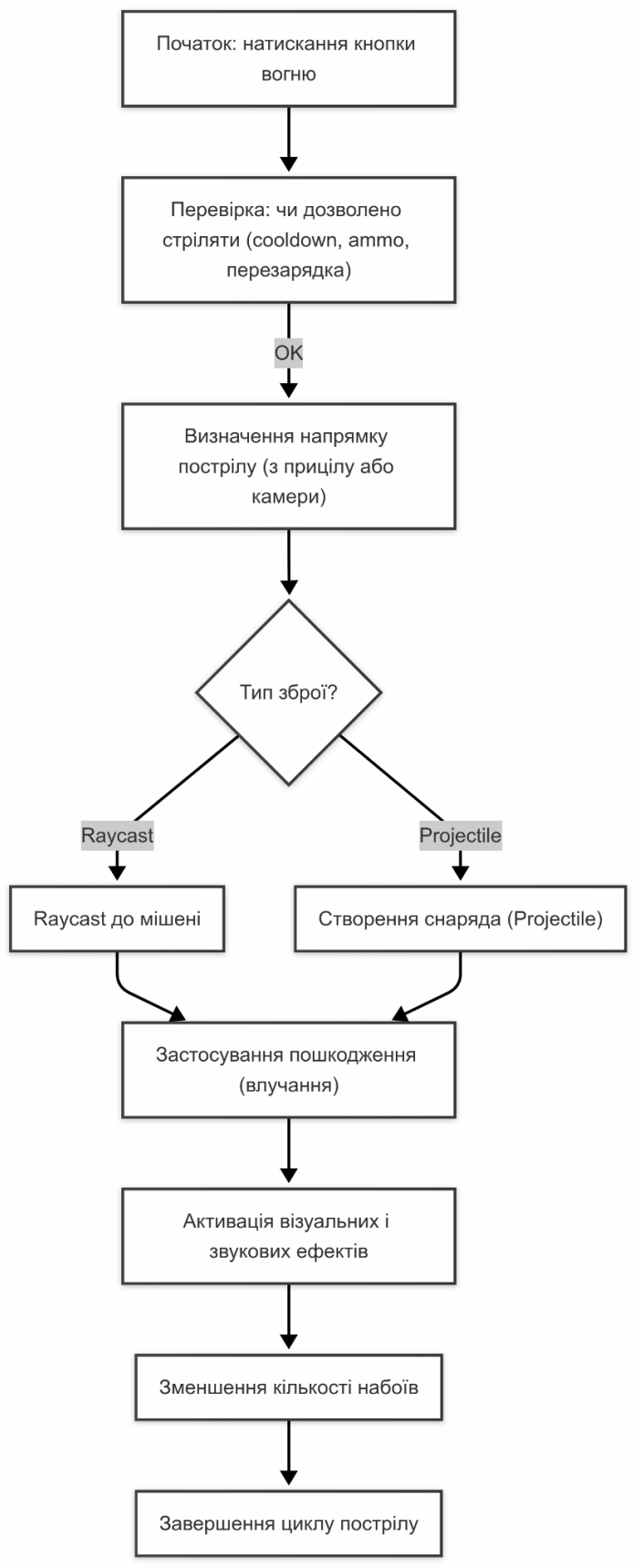

Рисунок 3.1 — Реалізація алгоритму пострілу

3.2.2 Налаштування об'єктів та колізій

Для забезпечення реалістичної симуляції влучання у віртуальному

середовищі необхідно коректно налаштувати фізичні властивості об'єктів та логіку обробки їхньої взаємодії. У проєкті механізм виявлення зіткнень реалізований шляхом комбінації вбудованих фізичних компонентів Unity (Collider, Rigidbody) та кастомного скрипта Projectile.cs, що прикріплюється до снарядів.

Конфігурація снарядів

Кожен снаряд створюється як prefab з обов'язковими компонентами:

– Rigidbody, що забезпечує фізичне переміщення;

– SphereCollider або CapsuleCollider, що виконує роль активного зіткнення;

– скрипт Projectile.cs, який керує подальшою логікою.

Ці об'єкти також мають налаштування шару (Layer) і мітки (Tag), які дозволяють фільтрувати зіткнення через систему Layer Collision Matrix. Наприклад, снаряд може бути налаштований так, щоб ігнорувати колізії з гравцем, але реагувати на поверхні, мішені та динамічні об'єкти.

Обробка влучань

У компоненті Projectile.cs реалізований метод OnCollisionEnter, який спрацьовує при першому фізичному контакті з іншим об'єктом. У межах цього методу визначається точка контакту (contact point) і матеріал поверхні, після чого запускається візуальна реакція:

Лістинг 3.9 — Імплементація колізій із об'єктами

```
    private void OnCollisionEnter(Collision collision)
    {
        ContactPoint contact = collision.contacts[0];
        GameObject decal = Instantiate(decalPrefab,
contact.point,
Quaternion.LookRotation(contact.normal)); Destroy(decal,
```

```
            5f); Destroy(this.gameObject);
        }
```

Окрім візуального ефекту (залишку на поверхні), цей фрагмент також демонструє одноразове використання об'єкта — після зіткнення снаряд видаляється із ігрової сцени, що забезпечує стабільність симуляції, оптимізовує використання ресурсів системи та запобігає їх накопиченню.

Взаємодія із мішенями

Об'єкти, які мають реагувати на постріли (наприклад, мішені або моделі противника), позначені тегами або містять власні компоненти, що перевіряються у collision.gameObject. При потребі можна ініціювати іншу поведінку — наприклад, зниження очок "життя", падіння або зміна візуального стану:

Лістинг 3.10 — Виклик реакції мішені, яка підпадає під тег Target

```
if (collision.gameObject.CompareTag("Target"))
{
collision.gameObject.GetComponent<TargetBehavior>()?.On
Hit();
}
```

Цей підхід дозволяє реалізовувати спеціалізовану поведінку об'єктів без зміни загальної логіки стрільби.

Оптимізація взаємодії

Для запобігання зайвим обчисленням у фізичному рушії використовується:

– фільтрація зіткнень через Layer Collision Matrix;

– обмеження кількості активних снарядів за допомогою PrefabPool;

– видалення або деактивація об'єктів одразу після влучання.

Завдяки цим рішенням забезпечується висока продуктивність навіть у разі великої кількості активних елементів у сцені, що є критично важливим для інтерактивних симуляторів із багатьма цільовими об'єктами.

4 ТЕСТУВАННЯ ТА ВІЗУАЛІЗАЦІЯ РЕЗУЛЬТАТІВ

Після завершення імплементації основних функціональних модулів системи необхідним етапом стало проведення тестування та візуального аналізу результатів її роботи. Основною метою цього етапу є верифікація коректності реалізованих механік, оцінка стабільності поведінки під час взаємодії з динамічними елементами середовища, а також фіксація результатів симуляції для подальшого аналізу.

Тестування проводилося в умовах, наближених до практичних сценаріїв використання системи: з урахуванням переміщення гравця, динамічної стрільби, реакції мішеней та взаємодії з об'єктами. Особлива увага приділялась перевірці взаємодії між модулями, таких як стрілецька підсистема, система виявлення колізій та управління виведенням візуальних ефектів.

У межах цього розділу представлено результати тестування ключових функціональних блоків, а також надано приклади фрагментів візуального інтерфейсу та поведінки системи в момент активації основних механік.

4.1 Тестування та налагодження

Тестування є завершальним етапом життєвого циклу програмної системи та має на меті перевірку відповідності реалізованої функціональності очікуваним

вимогам. У контексті інтерактивного симулятора, що реалізує механіку FPS, тестування відіграє критичну роль, оскільки дозволяє перевірити цілісність поведінки системи, її взаємозв'язки та стійкість до неочікуваних сценаріїв.

Класифікація методів тестування

Статичне тестування

Статичне тестування (англ. static testing) — це процес виявлення помилок, дефектів або слабких місць у програмному забезпеченні без його виконання. Воно є попереднім етапом до динамічного тестування та застосовується ще на стадії аналізу, проєктування або написання коду.

Головна мета — ідентифікувати проблеми на ранньому етапі, коли їх виправлення є найменш ресурсозатратним.

Основні цілі статичного тестування:

– виявлення синтаксичних, логічних та структурних помилок;

– забезпечення відповідності стилю коду, архітектурних стандартів і принципів розробки;

– аналіз залежностей, потоків даних, мертвого коду;

– підвищення читабельності, підтримуваності та повторного використання компонентів.

Методи статичного тестування:

– огляд коду (англ. code review): формальна або неформальна перевірка коду іншими членами команди або незалежним експертом;

– аналіз відповідності стандартам: використання форматерів і лінтерів для перевірки дотримання угод про стиль коду (naming conventions, indentation, comments тощо);

– статичний аналіз коду: застосування автоматизованих інструментів (наприклад, SonarQube, ESLint, ReSharper), що дозволяють

виявляти потенційні помилки (null refrence, неперевизначені змінні), надлишкові залежності, порушення інкапсуляції;

– моделювання контролю та потоків даних: аналіз шляху виконання програмного потоку без запуску, з урахуванням змін стану змінних.

Інструменти статичного аналізу:

– C#/.NET: Roslyn Analyzer, FxCop, StyleCop, ReSharper;

– Python: pylint, flake8, mypy;

– Загальні: SonarLint, SonarQube (для CI/CD).

Переваги:

– раннє виявлення помилок, ще до етапу виконання;

– зменшення вартості виправлень на пізніх етапах;

– підвищення якості коду на рівні архітектури;

– зменшення навантаження на тестувальників і QA-етап. Обмеження:

– не гарантує логічної коректності програми;

– не виявляє помилки, що виникають лише під час виконання;

– потребує значного залучення розробників до перевірок;

– у випадку складних систем може давати надмірну кількість хибнопозитивних результатів.

Динамічне тестування

Динамічне тестування (англ. dynamic testing) — це тип тестування, за якого програмне забезпечення перевіряється під час виконання. Метою є виявлення помилок, що проявляються лише в процесі реального функціонування системи: логічних, функціональних, поведінкових або продуктивних.На відміну від статичного, яке вивчає структуру коду, динамічне тестування дозволяє перевірити фактичну поведінку системи, її реакцію на вхідні дані, зміни середовища та взаємодію між компонентами.

Основні цілі динамічного тестування:

– перевірка відповідності реалізації функціональним вимогам;

– виявлення помилок, пов'язаних із обробкою даних, обчисленнями, умовами та сценаріями використання;

– оцінка стійкості системи до зовнішніх факторів;

– перевірка взаємодії між модулями, компонентами або сервісами.

Класифікація динамічного тестування:

– функціональне тестування: перевіряє, чи реалізовані функції відповідають специфікаціям (наприклад, чи виконується дія при натисканні кнопки);

– нефункціональне тестування: охоплює перевірку продуктивності, часу відгуку, навантаження, надійності та масштабованості;

– чорний ящик (англ. black-box): тестування з позиції користувача — без знання внутрішньої структури;

– білий ящик (англ. white-box): тестування з повним доступом до коду: перевірка логіки, умов, циклів;

– сірий ящик (англ. gray-box): комбінований підхід, коли тестувальник має часткове розуміння внутрішньої структури.;

Методи динамічного тестування:

– ручне тестування: перевірка поведінки вручну, за заздалегідь підготовленими сценаріями;

– автоматизоване тестування: виконання тест-кейсів за допомогою фреймворків (Selenium, NUnit, JUnit);

– тестування на рівні юнітів: модульні перевірки окремих функцій або класів (unit tests);

– інтеграційне тестування: перевірка взаємодії модулів між

собою;

– системне тестування: повна перевірка продукту як цілісного комплексу;

Інструменти динамічного тестування:

– Unity: Test Runner, Play Mode Tests, Profiler, Debug.Log, Gizmos;

– C#: NUnit, xUnit, MSTest;

– Інші: Selenium, Appium, Postman (для API), JMeter (навантаження).

Переваги:

– виявляє реальні помилки, які можуть виникати у користувача;

– дає змогу побачити програму “в дії”;

– тестує як функціональність, так і взаємодію між модулями;

– дозволяє легко створювати нові сценарії відповідно до специфіки.

Обмеження:

– вимагає більше часу та ресурсів, ніж статичне;

– ефективність залежить від якості тест-кейсів і сценаріїв;

– важко досягти повного покриття коду;

– можливі помилкові позитивні або пропущені помилки у складних системах.

Модульне тестування

Модульне тестування (англ. unit testing) — це тип динамічного тестування, що передбачає перевірку окремих, ізольованих одиниць програмного забезпечення. Під "модулем" зазвичай мається на увазі найменша автономна частина системи: функція, метод або клас. Метою модульного

тестування є встановлення коректності внутрішньої логіки кожної одиниці при заданих вхідних даних і очікуваних результатах, без залежності від інших компонентів системи.

Основні задачі модульного тестування:

– перевірити правильність обчислень, умов і логіки обробки;

– виявити локальні помилки на ранньому етапі;

– забезпечити стабільність коду при рефакторингу;

– дозволити повторне використання компонентів без побічних ефектів.

Типове модульне тестування включає три основні етапи. Спершу здійснюється підготовка вхідних даних, що відповідає фазі Arrange. На завершальному етапі виконується перевірка результату, що позначається як Assert. Такий підхід до тестування відомий під абревіатурою AAA (Arrange, Act, Assert).

Інструменти для модульного тестування:

– C#/.NET: NUnit, MSTest, xUnit;

– Python: unittest, pytest;

– Unity: Unity Test Framework (раніше — Unity Test Runner).

Переваги:

– дозволяє локалізувати помилки;

– прискорює рефакторинг (регресія одразу помітна);

– дає змогу писати код згідно з принципами TDD (розробка через тестування);

– підтримує CI/CD інтеграцію. Обмеження:

– не виявляє проблем на рівні взаємодії модулів;

– ускладнене тестування у випадку сильно зв'язаного коду;

– потребує додаткового часу на написання і підтримку тестів;

– у графічних або подієво-орієнтованих системах (як-от Unity) не завжди легко виділити модульну логіку без залежності від середовища.

Інтеграційне тестування

Інтеграційне тестування (англ. integration testing) — це тип тестування, що полягає у перевірці взаємодії між окремими модулями або підсистемами. Якщо модульне тестування фокусується на ізольованих компонентах, то інтеграційне — на правильності обміну даними, викликів і логіки зв'язку між ними. Інтеграційне тестування є наступним етапом після модульного і передує системному тестуванню. Воно особливо важливе у проєктах, де модулі мають чітко виражені інтерфейси та залежать від спільних ресурсів (зовнішні API, бази даних, подієві системи, менеджери станів).

Цілі інтеграційного тестування:

– переконатися в узгодженій взаємодії модулів;

– виявити помилки на стиках — передавання параметрів, форматів, станів;

– виявити проблеми ініціалізації, циклічних залежностей або конфліктів;

– перевірити ланцюги викликів, що охоплюють декілька компонентів.

Існує кілька підходів до інтеграційного тестування, кожен з яких має свої особливості. Інкрементальне тестування передбачає поступове об'єднання модулів, де кожна нова інтеграція проходить окрему перевірку. Цей підхід реалізується у двох варіантах: Top-down, коли тестування починається з головного модуля і поступово охоплює підлеглі (з використанням заглушок — stubs), та Bottom-up, де тестування здійснюється від підлеглих модулів до основного (із застосуванням драйверів — drivers).

Іншим підходом є неефективне (big bang) тестування, коли всі модулі інтегруються одночасно. Цей метод є простішим у реалізації, але менш

стабільним і складним у виявленні джерел помилок. Окремо виділяється функціональне інтеграційне тестування, яке орієнтується не на окремі модулі, а на перевірку логічних сценаріїв використання системи (наприклад, user flows або pipelines).

Інструменти для інтеграційного тестування:

– Unity: Play Mode Tests, інтеграційні сцени, моніторинг логів;

– .NET: NUnit з Mocking-фреймворками (Moq), AutoFixture;

– Загальні: TestContainers, Docker для симуляції залежностей, Postman (для інтеграції API).

Переваги:

– дозволяє виявити помилки, які не видно при модульному тестуванні;

– забезпечує цілісність логіки на рівні функціональних ланцюгів;

– дає впевненість у працездатності складених систем. Обмеження:

– складніше локалізувати джерело помилки;

– потребує налаштування середовища або симуляторів залежностей;

– може мати побічні ефекти, якщо модулі неправильно ізольовані;

– важко реалізувати для сильно взаємозалежних компонентів без рефакторингу.

Системне тестування

Системне тестування (англ. system testing) — це вид динамічного тестування, що полягає у перевірці повністю інтегрованого програмного продукту як єдиного цілого. Мета — визначити, чи система виконує всі специфіковані вимоги та чи функціонує належним чином у передбаченому

середовищі. На цьому етапі тестувальник поводиться як кінцевий користувач: система вже не розглядається як набір окремих компонентів, а як завершене, працездатне програмне забезпечення. Системне тестування є ключовим етапом перед розгортанням, доставкою або демонстрацією.

Основні цілі системного тестування:

– перевірити повноту реалізації функціональних вимог;

– оцінити поведінку системи в типових і граничних умовах;

– виявити дефекти, які проявляються лише при повному запуску системи;

– перевірити готовність продукту до інтеграції у зовнішнє середовище.

Системне тестування поділяється на кілька підвидів залежно від цілей і аспектів перевірки. Функціональне системне тестування спрямоване на верифікацію всіх функціональних вимог, включно зі сценаріями користувача, логікою обробки та взаємодією з інтерфейсом. У межах нефункціонального системного тестування перевіряються характеристики, пов'язані зі стабільністю, продуктивністю, зручністю інтерфейсу, локалізацією та безпекою системи.

Регресійне системне тестування передбачає повторну перевірку всієї системи після внесення змін, таких як рефакторинг або додавання нових функцій, з метою виявлення можливих порушень існуючої функціональності. Окремо виділяється тестування, орієнтоване на сценарії (end-to-end), яке охоплює повний користувацький шлях — від запуску програми до досягнення очікуваного результату.

Інструменти системного тестування:

– Unity: Play Mode, візуальна перевірка в Editor, Unity Profiler;

– CI-системи: автоматичне збирання і запуск тестових збірок;

– Зовнішнє середовище: тестові стенди, sandboxes, емулятори (у мобільних, AR/VR-сценаріях).

Переваги:

– дозволяє перевірити повноцінну, готову до використання систему;

– виявляє комплексні помилки, що виникають на рівні всієї програми;

– формує фінальну впевненість у готовності до публікації або захисту;

– зручне для демонстрації результату.

Обмеження:

– висока вартість виявлення помилки на пізньому етапі;

– складність відтворення деяких сценаріїв без автоматизації;

– потребує підготовленого середовища та повної збірки проєкту;

– може маскувати проблеми, які слід було виявити ще на рівні модулів.

Ручне тестування

Ручне тестування (англ. manual testing) — це форма динамічного тестування, у межах якої усі перевірки функціональності, логіки та інтерфейсів здійснюються без використання автоматизованих засобів, безпосередньо людиною — тестувальником або розробником. Це один з найдавніших та найгнучкіших методів, який не потребує попередньої автоматизації або написання скриптів.

На практиці ручне тестування найчастіше застосовується у невеликих проєктах, під час ранніх ітерацій розробки, при тестуванні змін, що складно автоматизуються, або для перевірки візуальних аспектів (UI, UX, анімацій, графіки).

Основні задачі ручного тестування:

– перевірка поведінки системи при взаємодії з

кінцевим користувачем;

– виявлення помилок, які можуть бути пропущені автоматикою (особливо у візуальному інтерфейсі);

– забезпечення гнучкого реагування на непередбачені ситуації;

– верифікація сценаріїв, що складно піддаються формалізації.

Існують різні типи ручного тестування, які використовуються залежно від цілей перевірки та наявної документації. Дослідницьке тестування (exploratory) проводиться без чітко визначеного сценарію: тестувальник вільно вивчає поведінку системи, формулює й перевіряє гіпотези. У сценарному тестуванні (scripted) дії виконуються відповідно до заздалегідь підготовлених тест-кейсів та інструкцій.

Ad-hoc тестування є спонтанним процесом без документації, що часто використовується розробниками для оперативної перевірки після змін у коді. Окрему категорію становить UI/UX-тестування, яке зосереджене на перевірці елементів інтерфейсу, їх розташування, стилістики, адаптивності та загальної зручності користування.

Інструменти:

– Unity: Scene View, Play Mode, Console, Inspector, Debug.Log;

– Допоміжні: чеклісти, таблиці тест-кейсів, скріншоти для фіксації поведінки;

– Нотатки: Google Docs, Excel, TestRail — для відслідковування пройдених сценаріїв.

Переваги:

– не потребує попередньої підготовки інфраструктури;

– дозволяє виявляти складні візуальні або контекстуальні

помилки;

– підходить для перевірки унікальних, рідкісних сценаріїв;

– добре комбінується з дослідницькими підходами.

Обмеження:

– суб'єктивність результатів (залежить від досвіду та уважності тестувальника);

– висока вартість часу, особливо при великій кількості сценаріїв;

– відсутність повторюваності без детальної документації;

– складно масштабувати та інтегрувати в CI/CD.

Автоматизоване тестування

Автоматизоване тестування (англ. automated testing) — це процес перевірки роботи програмного забезпечення за допомогою спеціально написаних скриптів або фреймворків, які автоматично виконують задані сценарії без участі людини. Основна мета — зменшити ручну працю, забезпечити повторюваність перевірок та інтегрувати тестування в життєвий цикл проєкту. Автоматизація найефективніша при великій кількості регресійних, критичних або повторюваних тестів, а також у проєктах з тривалим життєвим циклом або CI/CD-процесами.

Основні задачі автоматизованого тестування:

– забезпечити стабільну, багаторазову перевірку функціональності;

– виявляти помилки одразу після змін у коді;

– скоротити час регресійного тестування;

– мінімізувати людський фактор і суб'єктивність.

Автоматизоване тестування поділяється на кілька основних типів, кожен з яких охоплює певний рівень перевірки системи. Unit-тестування передбачає

автоматичне виконання модульних перевірок, зосереджених на логіці окремих функцій або класів. Integration-тестування реалізується у вигляді скриптованих сценаріїв для перевірки взаємодії між окремими модулями системи. UI-тестування (end-to-end) автоматизує взаємодію з графічним інтерфейсом, включаючи натискання кнопок, введення тексту та перевірку відображення. Regression Testing здійснюється для регулярної перевірки працездатності системи після внесення змін, з метою виявлення порушень у вже протестованому функціоналі. Нарешті, Smoke Testing виконується для швидкої перевірки базової функціональності системи одразу після її збірки.

Інструменти:

– Unity: Unity Test Framework (включає Unit + Play Mode тести), NUnit;

– Серверні системи: Unity Test Framework (включає Unit + Play Mode тести), NUnit;

– UI automation: Selenium (для веб), Appium (для мобільних), TestComplete, Playwright;

– Mocking-фреймворки: Moq, FakeItEasy (для ізоляції залежностей).

Переваги:

– швидкість і повторюваність виконання;

– висока точність і об'єктивність результатів;

– зручна інтеграція в CI/CD;

– суттєве зменшення витрат на довготривалому проєкті.

Обмеження:

– потребує початкових інвестицій часу і ресурсів;

– складно тестувати візуальні та креативні аспекти (анімації, UX);

– змінення логіки часто вимагає оновлення тестів (висока вартість підтримки);

– автоматичне тестування не замінює повністю ручне — особливо в прототипах або креативних іграх.

Методи, використані в межах даної роботи

У межах цього симулятора основну увагу було приділено ручному динамічному тестуванню, доповненому елементами модульного та інтеграційного аналізу. Основні сценарії перевірки включали:

Модульне тестування компонентів FPS-контролера: перевірка реакції на введення, функціональності руху, точності повороту камери.

– Інтеграційне тестування взаємодії між модулями стрільби, колізій та візуальних ефектів.

– Системне тестування поведінки симулятора при запуску повної сцени — із перевіркою ініціалізації, стабільності частоти кадрів, коректності завантаження об'єктів.

Під час розробки активно використовувались можливості Unity Editor для діагностики та налагодження ігрового процесу. Зокрема, вкладка Console застосовувалась для аналізу повідомлень і виявлення помилок, а Inspector дозволяв у реальному часі відстежувати значення змінних. Для перевірки напрямку руху снарядів використовувались візуальні Gizmos, що значно спрощувало відлагодження. Крім того, інструмент Profiler допомагав виявляти перевантаження CPU, особливо під час інтенсивної стрільби або численних зіткнень об'єктів.

Також застосовувались прості засоби виводу налагоджувальної інформації, зокрема Debug.Log().

Лістинг 4.1 — Приклад використання Debug.Log() для переконання коректності стрільби

```
Debug.Log("Projectile fired at position: " +
```

```
transform.position);
```

Це дозволяло в реальному часі відслідковувати параметри снарядів, напрямок стрільби, кількість боєприпасів та фіксувати, чи коректно спрацьовують події зіткнення.

Лістинг 4.2 — Використання Debug.Log() для перевірки коректності роботи колізій

```
private void OnCollisionEnter(Collision collision)
{
Debug.Log("Collision with: " + collision.gameObject.name);
}
```

Таким чином, тестування програмного забезпечення в контексті розроблення інтерактивного FPS-тренажера потребує застосування методів, які відповідають як природі системи, так і її архітектурним особливостям.

Серед наявних підходів до тестування найбільш доцільним виявилось комбіноване використання статичного, модульного, інтеграційного, системного та ручного тестування. Ці методи забезпечують як виявлення потенційних помилок на ранніх етапах розробки, так і контроль функціонування всієї системи у цілому.

Вибір саме цих підходів обумовлений низкою факторів: по-перше, специфіка проєкту не вимагала глибокої автоматизації або CI/CD-процесів, що дозволило зосередитися на ручному тестуванні та візуальній перевірці; по-друге, модульність системи спростила локалізацію помилок за допомогою модульного та інтеграційного підходів; по-третє, графічна й подієво-орієнтована природа середовища Unity значною мірою ускладнює повну автоматизацію перевірок, тому ручне тестування залишилось основним методом валідації. В результаті вдалося охопити усі критичні аспекти функціонування системи та гарантувати її стабільність у середовищі виконання.

4.2 Візуалізація результатів роботи

З метою підтвердження працездатності основних функціональних модулів системи було виконано фіксацію візуальних результатів у середовищі виконання. Нижче наведено серію ілюстрацій, що демонструють ключові аспекти поведінки симулятора під час взаємодії з гравцем та середовищем. Усі скріншоти зроблені безпосередньо в режимі Play Mode в редакторі Unity.

На початку запуску користувач розташовується у заздалегідь визначеній позиції з активною камерою від першої особи. Навколишнє середовище включає тестовий полігон, кілька об'єктів (мішеней) та візуально відокремлену зону для орієнтації.

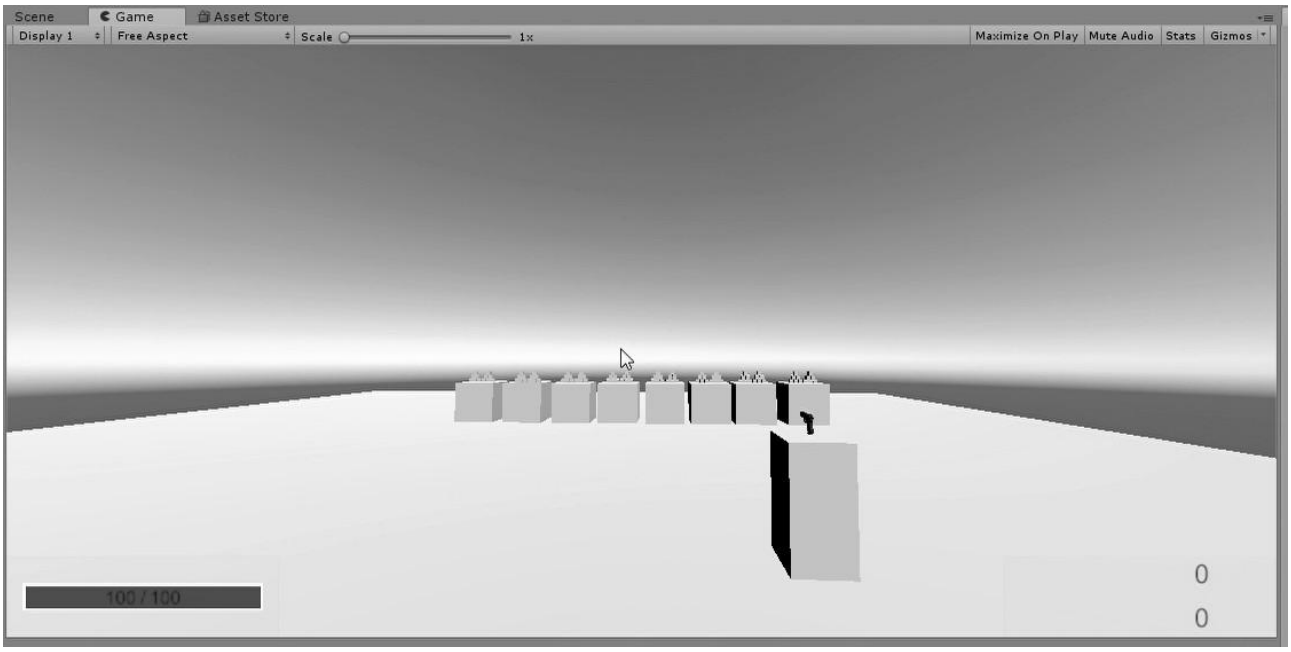

Рисунок 4.1 – Стартове положення гравця в ігровій сцені

У момент активації стрільби (натискання лівої кнопки миші) відбувається генерація візуального ефекту пострілу та ініціація створення снаряда. Видимий ефект містить спалах та об'ємне джерело руху — реалізовано через об'єкт Projectile, що створюється в точці походження.

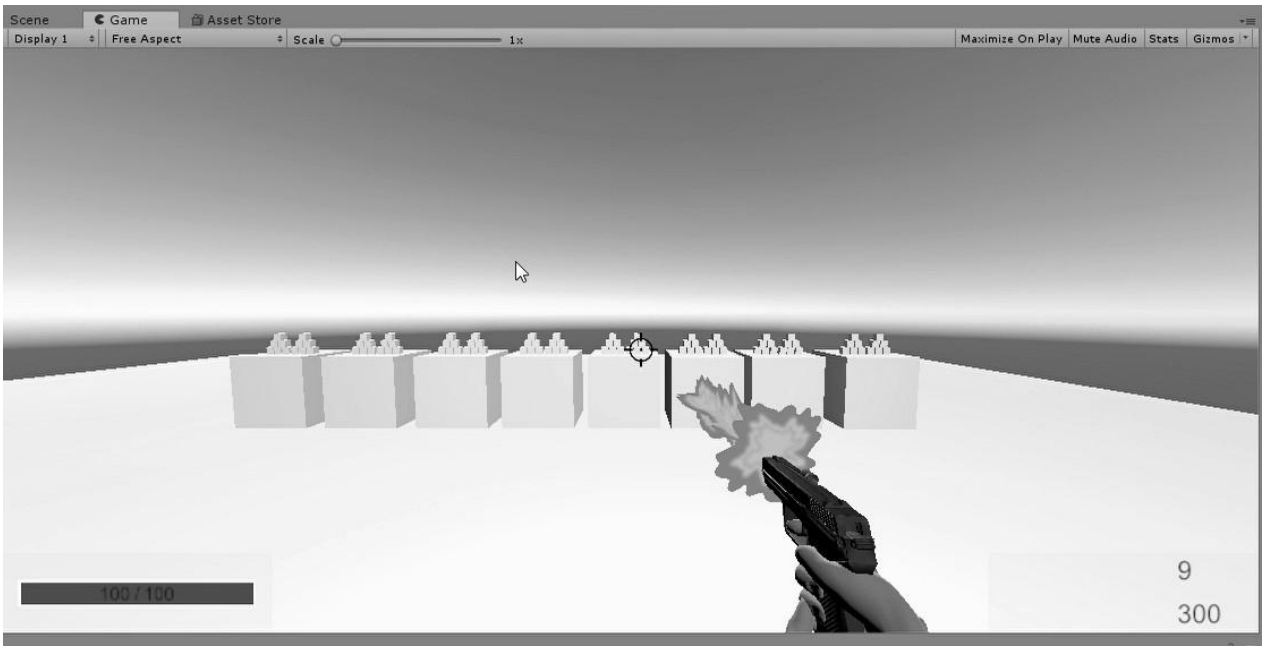

Рисунок 4.2 – Візуалізація пострілу з використанням ефекту вогню

Після контакту снаряда з мішенню відбувається обробка колізії, виведення ефекту зіткнення та, за потреби, виклик відповідного зворотного зв'язку. Після виконання кількох пострілів на сцені залишаються ознаки взаємодії гравця з навколишнім середовищем. Це може включати множинні сліди влучань, зміну положення мішеней або візуальне нагромадження елементів у зоні активності.

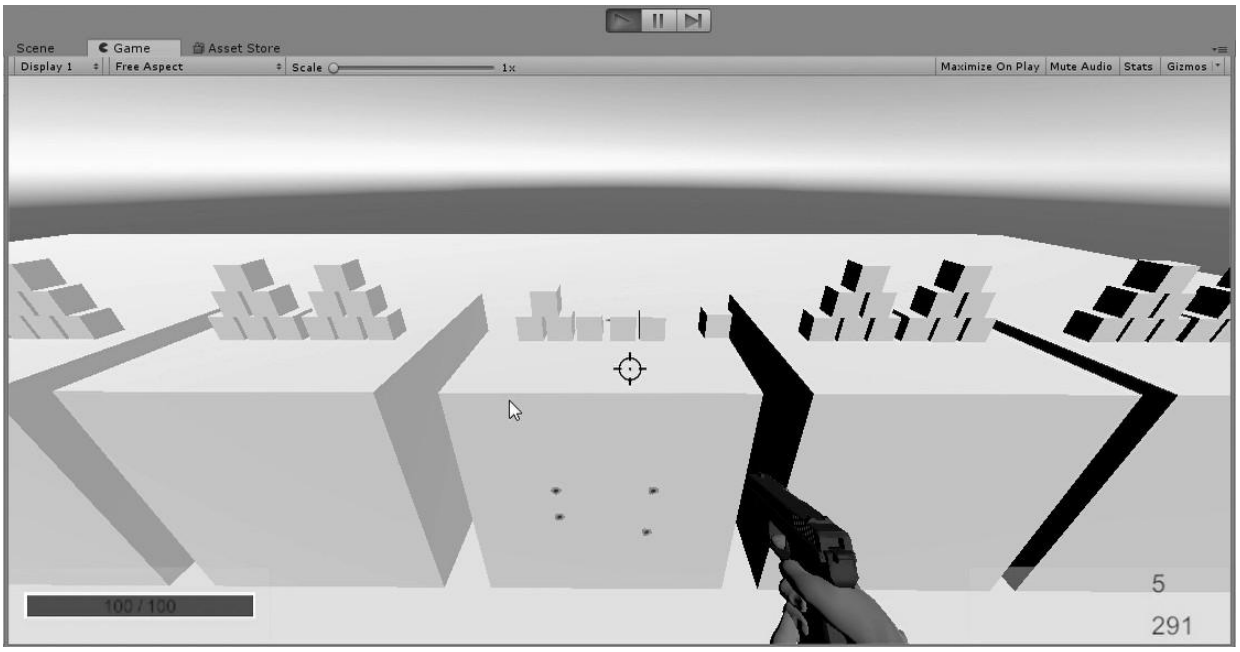

Рисунок 4.3 – Сліди влучань на поверхнях після серії пострілів

Представлені зображення наочно ілюструють працездатність основних

модулів симулятора, зокрема логіки керування, стрільби та обробки колізій. Візуальний зворотний зв'язок дозволяє користувачу миттєво сприймати результати дій, що є критично важливим в умовах тренажерної системи. Отримані результати свідчать про відповідність реалізованої системи вимогам функціональності та базової інтерактивності.

ВИСНОВКИ

У результаті виконаної роботи було реалізовано комп'ютерну систему симуляції тренування персоналу у вигляді 3D-тренажера з видом від першої особи, орієнтовану на розвиток навичок прицілювання, просторової орієнтації та реакції. Проєкт має потенціал практичного застосування в умовах попередньої підготовки кадрів у сферах, що потребують точної моторики та швидкої обробки візуальної інформації.

У ході дослідження було здійснено аналіз технічного завдання та наявних рішень у сфері віртуального навчання. Обґрунтовано вибір рушія Unity як технічної платформи, враховуючи його підтримку фізичних симуляцій, систему компонентів та інтегровані засоби розробки. Визначено ключові вимоги до ігрового середовища, моделі управління персонажем, реалізації стрільби та обробки колізій.

У рамках розробки імплементовано модульну архітектуру проєкту, що охоплює окремі компоненти для управління рухом гравця, керування камерою, взаємодії зі зброєю, стрільби, виявлення зіткнень та виведення ефектів. Особливу увагу приділено реалістичному візуальному зворотному зв'язку, що підсилює ефект занурення та сприяє підвищенню якості навчання.

На етапі тестування було використано комбінацію статичних, модульних, інтеграційних, системних і ручних підходів. Це дозволило виявити та усунути низку логічних і поведінкових помилок, а також оптимізувати взаємодію між підсистемами. Результати тестування підтверджують коректність реалізованих алгоритмів та стабільність роботи симулятора в середовищі виконання.

Таким чином, створена система відповідає поставленим вимогам та демонструє функціональну готовність до подальшого вдосконалення. Проєкт має потенціал для масштабування, розширення функціоналу (додавання нових сценаріїв, мішеней, зброї, механік) та адаптації під конкретні потреби замовника або навчального закладу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Unity Technologies. Unity User Manual 2018.3. [Електронний ресурс]. – Режим доступу: https://docs.unity3d.com/2018.3/Documentation/Manual/index.html

2. Unity Technologies. Scripting API Reference. [Електронний ресурс]. – Режим доступу: https://docs.unity3d.com/ScriptReference/

3. Гамма Е., Хелм Р., Джонсон Р., Влісідес Дж. Шаблони проєктування. – К.: Діалектика, 2007. – 368 с.

4. Meszaros G. xUnit Test Patterns: Refactoring Test Code. – Boston: Addison-Wesley, 2007. – 896 p.

5. Fowler M. Refactoring: Improving the Design of Existing Code. – Boston: Addison-Wesley, 2019. – 448 p.

6. Unity Learn. Create with Code Course. [Електронний ресурс]. – Режим доступу: https://learn.unity.com/course/create-with-code

7. Microsoft Docs. C# Programming Guide. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/dotnet/csharp/

8. W3C. HTML5 Specification. [Електронний ресурс]. – Режим доступу: https://www.w3.org/TR/html5/

9. Майєр Б. Об'єктно-орієнтоване програмування. Принципи, технології, мова. – К.: Видавництво "BHV", 2010. – 704 с.

10. СонарК'юб. SonarQube Documentation. [Електронний ресурс]. – Режим доступу: https://docs.sonarsource.com/

11. Unity Forum. Best practices for modular FPS architecture. [Електронний ресурс]. – Режим доступу: https://forum.unity.com

ДОДАТКИ

Додаток А

```
using UnityEngine;
using System.Collections;
using UnityTK;

namespace OsFPS
{
    /// <summary>
    /// Weapon behaviour that binds to <see
cref="Weapon.weaponFire"/> in order to handle firing a bullet.
    /// </summary>
    [RequireComponent(typeof(Weapon))]
    public class WeaponProjectileShooter : MonoBehaviour
    {
        public Weapon weapon
        {
            get
            {
                if (this._weapon == null)
                    this._weapon = GetComponent<Weapon>();
                return this._weapon;
            }
        }
        private Weapon _weapon;

        public float baseSpread { get { return
this.weapon.weaponDefinition.spread; } }
        [Header("Spread")]
        public float movingSpreadModifier = 4;
        public float crouchedSpreadModifier = 0.4f;
        /// <summary>
        /// Spread will not instantly change state after leaving
movement state, instead it will linearly interpolate using lerp.
        /// This intensity is multiplied with Time.deltaTime and
```

```
used as lerp parameter t.
        /// </summary>
        public float spreadLerpIntensity = 10;

        [Header("Projectile")]
        public Transform projectileOrigin;

        /// <summary>
        /// The prefab for projectile fired by this shooter.
        /// </summary>
        public GameObject projectile;

        [Header("Debug")]
        [SerializeField]
        private float spread;

        public void Update()
        {
            float spreadTarget = this.baseSpread;

            // States
            if
(this.weapon.weaponHandler.entity.model.crouch.IsActive())
                spreadTarget *= this.crouchedSpreadModifier;

            if
(this.weapon.weaponHandler.entity.model.motorMovement.Get().magnit
ude > 0.01f)
                spreadTarget *= this.movingSpreadModifier;

            // Lerp spread
            this.spread = Mathf.Lerp(this.spread, spreadTarget,
Time.deltaTime * this.spreadLerpIntensity);
        }
```

```
        public void Awake()
        {
            this.weapon.weaponFire.onStart += this.OnFire;
            this.spread = this.baseSpread;
        }

        private void OnFire()
        {
            // Debug spread
            Debug.DrawLine(this.projectileOrigin.position,
this.projectileOrigin.position + (this.projectileOrigin.forward *
15f), Color.red, 50f);

            // Consume ammo
            this.weapon.ammoInClip--;

            // Calculate spread
            Quaternion rot = this.projectileOrigin.rotation;
            float spreadRngBy2 = ((Random.value * 2f) - 1f) / 2f;
            Vector3 euler = rot.eulerAngles;
            euler.x += this.spread * spreadRngBy2;
            euler.y += this.spread * spreadRngBy2;
            rot = Quaternion.Euler(euler);

            // Debug spread
            Debug.DrawLine(this.projectileOrigin.position,
this.projectileOrigin.position + (rot * Vector3.forward * 15f),
Color.green, 50f);

            // Spawn projectile
            var p =
PrefabPool.instance.GetInstance(this.projectile);
            p.transform.position = this.projectileOrigin.position;
            p.transform.rotation = rot;
        }
    }
```

```
}
```