НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих

комп'ютерних систем

До захисту допущено

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

«___»_____2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою

«Системне програмування та спеціалізовані комп'ютерні системи»

спеціальності 123 «Комп'ютерна інженерія»

на тему: «Рушій для створення програмних інтерактивних агентно-орієнтованих комп'ютерних симуляцій м'якого реального часу»

Виконав:
студент IV курсу, групи КВ-11
Парієнко Віктор Володимирович _____

Керівник:
Доцент каф. СПіСКС, к.т.н., доцент
Тарасенко-Клятченко О. В. _____

Консультант з нормоконтролю:
Доцент каф. СПіСКС, к.т.н.
Клятченко Ярослав Михайлович _____

Рецензент:
к.т.н., доц., доц.каф.ПЗКС
Заболотня Т.М. _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2025 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет прикладної математики
Кафедра системного програмування і
спеціалізованих комп'юктерних систем

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Системне програмування та спеціалізовані комп'ютерні системи»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Віталій РОМАНКЕВИЧ
«___» _____2025 р.

ЗАВДАННЯ
на дипломний проєкт студенту
Парієнку Віктору Володимировичу

1. Тема проєкту «Рушій для створення програмних інтерактивних агентно-орієнтованих комп'ютерних симуляцій м'якого реального часу», керівник проєкту Тарасенко-Клятченко Оксана Володимирівна, доц. каф. СПСКС, к.т.н., затверджені наказом по університету від
«___»_____ 2025 р. №_____
2. Термін подання студентом проєкту _____
3. Вихідні дані до проєкту див. Технічне завдання
4. Зміст пояснювальної записки:
   - аналіз існуючих рішень та обґрунтування теми дипломного проєкту;
   - аналіз засобів реалізації;
   - розробка рушія.

5. Перелік графічного патеріалу (із зазначенням обов'язкових креслень, плакатів, презентацій тощо:

- ІАЛЦ.045490.005 Д1. Архітектура рушія. Схема структурна
- ІАЛЦ.045490.006 Д2. Структура ECS. Схема структурна
- ІАЛЦ.045490.007 Д3. Алгоритм збиральника сміття. Схема алгоритму
- ІАЛЦ.045490.008 Д4. Алгоритм оновлення світу. Схема алгоритму

6. Консультанти розділів проєкту*

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Клятченко Я.М. доц.каф.СПСКС, к.т.н.		

7. Дата видачі завдання «20» жовтня 2024 р.

Календарний план

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	Вивчення літератури за тематикою проєкту	15.11.2024	
2.	Розроблення та узгодження технічного завдання	01.12.2024	
3.	Аналіз існуючих рішень	05.12.2024	
4.	Вибір програмних засобів для реалізації проєкту	01.03.2025	
5.	Розробка програмного продукту	01.04.2025	
6.	Тестування та відлагодження програмного продукту	01.05.2025	
7.	Написання пояснювальної записки	15.05.2025	
8.	Підготовка графічної частини дипломного проєкту	20.05.2025	
9	Оформлення документації дипломного проєкту	25.05.2025	
10.	Попередній огляд матеріалів диплому на кафедрі	31.05.2025	

Студент Віктор Парієнко

Керівник проєкту Оксана Тарасенко-Клятченко

---

* Консультантом не може бути зазначено керівника дипломного проєкту.

АНОТАЦІЯ

Кваліфікаційна робота включає пояснювальну записку (91 с., 34 рис. 2 табл., 3 додатки).

Об’єкт розробки – рушій із підтримкою рендерингу тривимірної графіки, що надає інструменти та середовище для створення інтерактивних комп’ютерних симуляцій із взаємодією автономних сутностей (агентів), у режимі м’якого реального часу.

Рушій дозволяє: створювати проєкт симуляції та змінювати його налаштування; імпортувати файли тривимірних моделей та графічних зображень різних форматів; рендерити тривимірну сцену з джерелами світла; додавати ефекти постобробки до фінального зображення кадру. Передбачена можливість додавання об’єктам власної логіки поведінки та взаємодії за допомогою C++ компонентів. В процесі розробки було використано мову програмування C++ та графічний API OpenGL 4.6 Core для рендерингу графіки.

В ході розробки:

- Проведено аналіз технічних рішень що використовуються в існуючих рушіях;
- Сформульовані вимоги до можливостей та інструментарію рушія;
- Побудовано архітектуру рушія та визначено підтримувані технології;
- Розроблено рушій та інструментарій користувацького редактора;
- Розроблено систему взаємодії між користувацьким редактором та проєктом симуляції.

Упровадження цього рушія дозволить зручне та швидке створення інтерактивних комп’ютерних симуляцій з тривимірною графікою.

Ключові слова:

ГРАФІЧНИЙ РУШІЙ, ФІЗИЧНИЙ РУШІЙ, РЕНДЕРИНГ, СИМУЛЯЦІЯ, ГРАФІКА, СЦЕНА, ШЕЙДЕР, C++, OPENGL.

ABSTRACT

The qualification work includes an explanatory note (91 p., 34 fig. 2 tables, 3 appendices).

The object of development is an engine that supports rendering of three-dimensional graphics, which provides tools and an environment for creating interactive computer simulations with the interaction of autonomous entities (agents) in soft real-time.

The engine allows you to: create a simulation project and change its settings; import files of three-dimensional models and graphic images of various formats; render a three-dimensional scene with light sources; add post-processing effects to the final image of the frame. It is possible to add custom behavior and interaction logic to objects using C++ components. In the course of development, the C++ programming language and the OpenGL 4.6 Core graphics API for rendering graphics were used.

During the development:

- An analysis of technical solutions used in existing engines was conducted;
- Requirements for the engine’s capabilities and tools were formulated;
- The engine architecture was built and supported technologies were identified;
- The engine and user editor tools were developed;
- A system for interaction between the user editor and the simulation project was developed.

The implementation of this engine will allow convenient and fast creation of interactive computer simulations with three-dimensional graphics.

Keywords:

GRAPHICS ENGINE, PHYSICS ENGINE, RENDERING, SIMULATION, GRAPHICS, SCENE, SHADER, C++

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	примітки
	А4	ІАЛЦ.045490.002 ТЗ	Рушій для створення	4		
			програмних інтерактивних			
			агентно-орієнтованих			
			комп’ютерних симуляцій			
			м’якого реального часу			
			Технічне завдання			
	А4	ІАЛЦ.045490.003 ТП	Рушій для створення	2		
			програмних інтерактивних			
			агентно-орієнтованих			
			комп’ютерних симуляцій			
			м’якого реального часу			
			Відомість технічного			
			проєкту			
	А4	ІАЛЦ.045490.004 ПЗ	Рушій для створення	91		
			програмних інтерактивних			
			агентно-орієнтованих			
			комп’ютерних симуляцій			
			м’якого реального часу			
			Пояснювальна записка			

					ІАЛЦ. 045490.001 ОА	
Зм	***Лист***	***№ докум.***	***Підп.***	***Дата***		
Розроб.		Парієнко В. В.			Рушій для створення програмних інтерактивних агентно-орієнтованих комп’ютерних симуляцій м’якого реального часу ***Опис альбому***	***Літ.*** / ***Лист*** 1 / ***Листів*** 2
Перев.		Тарасенко- -Клятченко				
Н. контр.		Клятченко Я.М.				**НТУУ «КПІ ім. Ігоря Сікорського», ФПМ, КВ-11**
Зав. каф		Романкевич В.О.				

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	примітки
	А4	ІАЛЦ.045490.005 Д1	Архітектура рушія	1		
			Схема структурна			
	А4	ІАЛЦ.045490.006 Д2	Структура ECS	1		
			Схема структурна			
	А4	ІАЛЦ.045490.007 Д3	Алгоритм збиральника	1		
			сміття			
			Схема алгоритму			
	А4	ІАЛЦ.045490.008 Д4	Алгоритм оновлення світу	1		
			Схема алгоритму			
		Диск CD-ROM	Текст пояснювальної	1		
			записки.			
			Графічний матеріал			

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ РОЗРОБКИ

Назва розробки: «Рушій для створення програмних інтерактивних агентно-орієнтованих комп’ютерних симуляцій м’якого реального часу».

Галузь застосування: Створення інтерактивних віртуальних середовищ, для моделювання поведінки об’єктів і систем віртуального світу з можливістю керування ними.

2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на дипломне проектування на здобуття першого (бакалаврського) рівня вищої освіти, затверджене кафедрою системного програмування і спеціалізованих комп’ютерних систем Національного технічного університету України «Київський Політехнічний Інститут імені Ігоря Сікорського».

3. МЕТА І ПРИЗНАЧЕННЯ РОБОТИ

Метою даного проекту є створення рушія з комплексом систем, що дозволить створювати програмні інтерактивні симуляції в межах розробленого фреймворку.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом інформації є технічна та науково-технічна література, технічна документація, публікації у періодичних виданнях та електронні статті у мережі Інтернет.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до програмного продукту, що розробляється

- сумісність з операційною системою Windows 10/11
- можливість створення проєкту симуляції

- можливість редагування параметрів об’єктів на сцені використовуючи редактор
- можливість запуску симуляції в редакторі
- можливість рендерингу тривимірної освітленої сцени
- наявність системи рефлексії
- наявність системи компонентів сутностей
- наявність системи обробки подій введення

5.2. Вимоги до апаратного забезпечення

- Процесор: AMD Ryzen 5 3600 або відповідний аналог
- Оперативна пам’ять: 16 Гб
- Графічний процесор: NVIDIA GTX 1060 6GB або відповідний аналог

5.3. Вимоги до програмного та апаратного забезпечення користувача

- Операційна система Windows 10/11
- Драйвер графічного процесору з підтримкою OpenGL 4.6 API

6. ЕТАПИ РОЗРОБКИ

№ з/п	Назва етапів виконання дипломного проекту	Термін виконання етапів
1.	Вивчення літератури за тематикою проєкту	15.11.2024
2.	Розроблення та узгодження технічного завдання	01.12.2024
3.	Аналіз існуючих рішень	05.12.2024
4.	Вибір програмних засобів для реалізації проєкту	01.03.2025
5.	Розробка програмного продукту	01.04.2025
6.	Тестування та відлагодження програмного продукту	01.05.2025
7.	Написання пояснювальної записки	15.05.2025
8.	Підготовка графічної частини дипломного проєкту	20.05.2025

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ.045490.004 ПЗ	Рушій для створення	91		
			програмних інтерактивних			
			агентно-орієнтованих			
			комп’ютерних симуляцій			
			м’якого реального часу			
			Пояснювальна записка			
	А4	ІАЛЦ.045490.005 Д1	Архітектура рушія	1		
			Схема структурна			
	А4	ІАЛЦ.045490.006 Д2	Структура ECS	1		
			Схема структурна			

Зм	Лист	№ докум.	Підп.	Дата	***ІАЛЦ. 045490.003 ТП***		
Розроб.		Парієнко			Рушій для створення програмних інтерактивних агентно-орієнтованих комп’ютерних симуляцій м’якого реального часу ***Відомість технічного проекту***	*Літ.*	*Лист* 1 / 2
Перев.		Тарасенко-					
		-Клятченко О.В.				**НТУУ «КПІ ім. Ігоря Сікорського», ФПМ, КВ-11**	
Н. контр.		Клятченко Я.М.					
Затв.		Тарасенко В.П.					

Поз.	Формат	ПОЗНАЧЕННЯ	НАЙМЕНУВАННЯ	Кількість аркушів	№ прим.	Примітки
	А4	ІАЛЦ.045490.007 Д3	Алгоритм збиральника	1		
			сміття			
			Схема алгоритму			
	А4	ІАЛЦ.045490.008 Д4	Алогоритм оновлення світу	1		
			Схема алгоритму			
		Диск CD-ROM	Текст пояснюввальної	1		
			записки.			
			Графічний матеріал			

Зм	Лист	№ докум.	Підп.	Дата	ІАЛЦ.045490.003 ТП	Лист 2

ЗМІСТ

ДОДАТКИ

Додаток 1. Копії графічних матеріалів

- ІАЛЦ.045490.005 Д1. Архітектура рушія. Схема структурна
- ІАЛЦ.045490.006 Д2. Структура ECS. Схема Структурна
- ІАЛЦ.045490.007 Д3. Алгоритм збиральника сміття. Схема алгоритму
- ІАЛЦ.045490.008 Д4. Алгоритм оновлення світу. Схема алгоритму

Додаток 2. Фрагменти програмного коду

Додаток 2. Презентація

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

VeiM – рушій розроблений у рамках цього дипломного проєкту.

Графічний API – стандартизований програмний інтерфейс для керування графічним конвеєром.

Blueprints – інструмент візуального програмування в Unreal Engine.

Режим м’якого реального часу – харакетристика системи або симуляції у якій виконання звадань має відбуватись у межах визначених часових обмежень, але порушення цих обмежень не призводить до критичних наслідків для системи.

C# - мова програмування високого рівня.

C++ - низькорівнева мова програмування.

Unity Asset Store – офіційний маркетплейс готових рішень для розробки в Unity.

GLSL (OpenGL Shading Language) – високорівнева мова програмування шейдерів для OpenGL

HLSL (High-Level Shading Language) – високорівнева мова програмування шейдерів від Microsoft для Direct3D

PBR (Physically Based Rendering) – підхід до візуалізації, що імітує фізичні закони взаємодії світла з матеріалами

Оверкіл – надмірна складність або функціональність системи, що перевищує реальні потреби користувача.

Фреймворк – базовий каркас рушія, що визначає ключові системи для організації логіки та контенту.

Рендерпас – окремий етап візуалізації у графічному пайплайні.

Пайплайн – послідовність етапів обробки даних у графічному або обчислювальному процесі.

ECS (Entity-Component-System) – шаблон проєктування орієнтований на дані, який розподіляє завдання між сутноснятим, компоненатами і системами.

GDScript – скриптова мова рушія Godot.

OpenGL – кросплатформний графічний API для рендерингу тривимірної графіки.

Vulkan – низькорівневий графічний API нового покоління для рендерингу тривимірної графіки.

DirectX 12 – низькорівневий графічний API від Microsoft для рендерингу тривимірної графіки та обчислень.

GPU – графічний процесор, спеціалізований для паралельної обробки графіки і обчислень.

JVM – віртуальна машина Java, що виконує байт-код незалежно від операційної системи.

SDL – кросплатформна бібліотека для роботи з мультимедія, вікнами, аудіо та пристроями вводу.

SFML – кросплатформна мультимедіа бібліотека для створення графічних застосунків.

Allegro – кросплатформна бібліотека для мультимедійних задач.

WinAPI – набір фукнцій для програмування під Windows, що надає доступ до системних ресурсів і інтерфейсу операційно системи.

GLFW – кросплатформна бібліотека длястворення вікон, обробки вовду та управління контекстом OpenGL.

stb_image – бібліотека для завантаження зображеть різних форматів.

Dear ImGui – бібліотека для швидкого створення інтерактивних інтерфейсів користувача.

SIMD-інструкції – набір команд процесора для одночасної обробки кількох даних однією інструкцією.

GLM – математична бібліотека для роботи з комп’ютерною графікою.

Layout – структура розташування елементів інтерфейсу у вікні застосунку.

Drag-and-drop – механізм перетягування об’єктів інтерфейсу за допомогою миші для їх переміщення або копіювання.

XAML – декларативна мова розмітки для опису інтерфейсів користувача в застосунках Microsoft.

.NET – програмна платформа від Microsoft для розробки застосунків.

GC – автоматизована система управління пам’яттю, що звільняє непотрібні об’єкти.

RTTI – механізм мови програмування для визнаечння типу об’єкта під час виконання.

Меш – сітка з вершимн, ребер і граней, що описує форму тривимірного об’єкта.

HDR – технологія обробки зображень із розширеним динамічним діапазоном яскравості.

ВСТУП

В сучасному світі комп’ютерні технології стрімко розвиваються, і одинм з ключових напрямків цього розвитку є створення інтерактивних комп’ютерних симуляцій, які дозволяють моделювати складні системи з динамічною взаємодією об’єктів. Сфери застосування таких симуляцій охоплюють наукові дослідження, інженерію, освіту, розваги та спорт.

Сьогодні розробка інтерактивних симуляцій неможлива без використання спеціалізованого програмного забезпечення – рушіїв. Їх поява була обумовлена необхідністю повторного використання вже розроблених рішень у нових проєктах, що дозволяє суттєво знизити витрати часу та ресурсів на розробку. Сучасний рушій є частиною будь-якої симуляції створеної з його використанням, та забезпечує роботу основних систем що є універсальними для кожного продукту. Крім того, він часто включає інструменти, що спрощують процес розробки з використанням рушія.

Однак через велику різноманітність сфер застосування та індивідуальних потреб розробників існуючі рушії рідко повністю відповідають вимогам конкретного проєкту. В деяких випадках вони можуть бути занадто складними та ресурсомісткими, в інших – не містити необхідних функцій, що змушує розробників або адаптувати готові рішення, або створювати власні.

Тому метою даного проєкту є розробка рушія, призначеного для створення інтерактивних комп’ютерних симуляцій, який:

- матиме гнучку архітектуру;
- забезпечить оптимальну продуктивність у режимі «м’якого» реального часу;
- включатиме лише необхідні компоненти, що сприятиме оптимізації під специфіку проєктів;
- надасть зручні інструменти для використання рушія;

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБҐРУНТУВАННЯ ТЕМИ ДИПЛОМНОГО ПРОЕКТУ

1.1. Аналіз особливостей існуючих рушіїв

На сьогоднішній день існує велика кількість готових рушіїв, включаючи комерційні та відкриті, що широко використовуються в різних індустріях. Всі рушії мають свої переваги та недоліки, і підходять під різні потреби користувачів, тому потрібно проаналізувати існуючі рушії, акцентуючи увагу на їх особливостях.

Для аналізу вибрано наступні рушії:

1) Unreal Engine;
2) Unity;
3) OGRE.

Unreal Engine (рис. 1.1) – це рушій з підтримкою тривимірної графіки, розроблений компанією Epic Games. Він написаний мовою C++ та підтримує всі основні сучасні платформи, зокрема Windows, Linux, macOS, IOS та Android, а також графічні API: OpenGL, DirectX 11, DirectX 12, Vulkan.

Рушій активно використовується в різних галузях, зокрема в ігровій індустрії, кіноіндустрії, телебаченні, маркетингу, віртуальній реальності. Його можливості дозволяють створювати інтерактивні тривимірні сцени з високим рівнем фотореалізму, анімовану графіку, застусунки з використанням технологій віртуальної та доповненої реальності.

Для створення проектів в Unreal Engine використовується мова програмування C++, а також візуальна система скриптів Blueprints, що надає змогу працювати з рушієм користувачам без технічних знань у програмуванні - таким як художники, дизайнери, сценаристи або ентузіасти, зацікавлені в комп'ютерній графіці і розробці інтерактивного контенту. Важливою перевагою

є відктритий вихідний код рушія, доступний на умовах ліцензії, що дозволяє вільну модифікацію рушія під власні потреби.

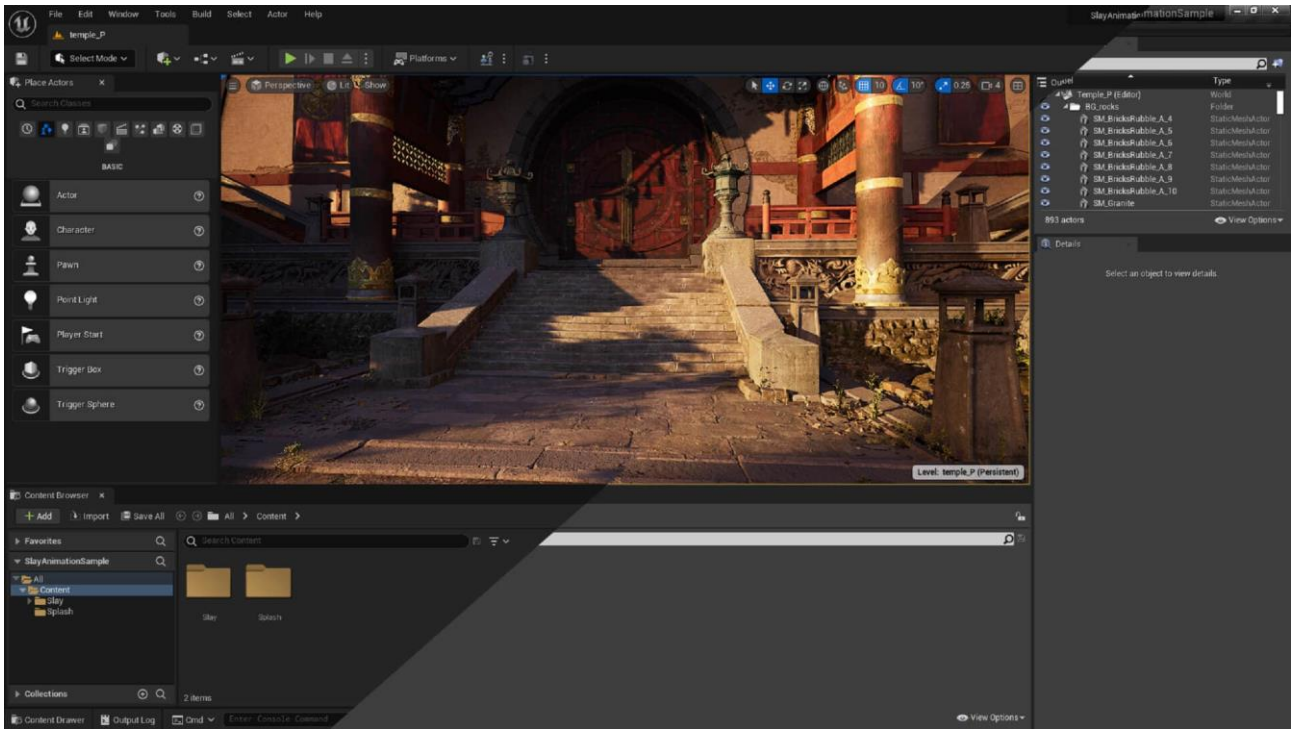


Рисунок 1.1 – Знімок екрану редактора Unreal Engine 4/5 з відкритою графічною сценою

Unreal Engine має модульну архітектуру з високим рівнем абстракції, та має широку функціональність та інструментарій. Він включає безліч готових підсистем, які не є основними, та можуть бути легко інтегрованими в проєкт користувачем, що значно зменшує витрати часу на розробку. Їх перевагою є надійність та гнучкість, підтримка кастомізації та готовність до мережевої взаємодії.

Однак попри свої численні переваги, Unreal Engine не завжди є оптимальним вибором. Рушій може виявитись занадто складним та «важким» для простих агентних симуляцій. Значна частина підсистем рушія є активною за замовчуванням і продовжує функціонувати навіть тоді, коли вони не потрібні, а також значну частину пам’яті використовують непотрібні дані. Наприклад,

система мережевої взаємодії глибоко інтегрована в архітектуру рушія, і навіть якщо розробник не має наміру створювати мережевий застосунок, рушій все одно розглядає проєкт як мережевий – з одним клієнтом, який також є сервером. Такий підхід хоч і не заважає процесу розробки, однак спричиняє додаткові витрати ресурсів і ускладнює оптимізацію.

Даний рушій має обмежений контроль над ключовими системами, чим є високий поріг входу для модифікації рушія на рівні вихідного коду, тому вищезазанчена можливість насправді недоступна для більшості розробників. Попри відкритість, складна структура рушія, велика кількість взаємозалежних компонентів та високий рівень абстракції роблять внесення змін у код доступним переважно лише досвідченим професіоналам.

Таким чином, Unreal Engine – це потужний, професійний інструмент для створення високоякісного інтерактивного контенту. Водночас його використання недоцільне для великої категорії користувачів, яким потрібен легкий, вузькоспеціалізований рушій з невеликим споживанням ресурсів і меншою складністю.

Unity (рис. 1.2) – це кросплатформний рушій для розробки ігор, розроблений компанією Unity Technologies. Він активно використовується для створення як двовимірних так і тривимірних інтерактивних графічних симуляцій. Рушій підтримує широкий спектр операційних систем і пристроїв, зокрема Windows, Linux, macOS, Android, iOS, WebGL, а також ігрові консолі.

Unity особливо популярний у сфері мобільної розробки, завдяки своїй ефективності, зручному середовищу розробки та великій кількості готових рішень. Однак він також знайшов застосування в інших галузях: кіноіндустрії, архітектурній візуалізації, автомобілебудуванні, будівництві, інженерії та системах військової підготовки збройних сил США.

Для програмування в Unity використовується мова C#, що дозволяє швидко опанувати інструмент навіть початківцям. Екосистема рушія

орієнтована на швидку розробку, використання стандартних компонентів, систем візуального налаштування поведінки об’єктів та зручний редактор сцени.

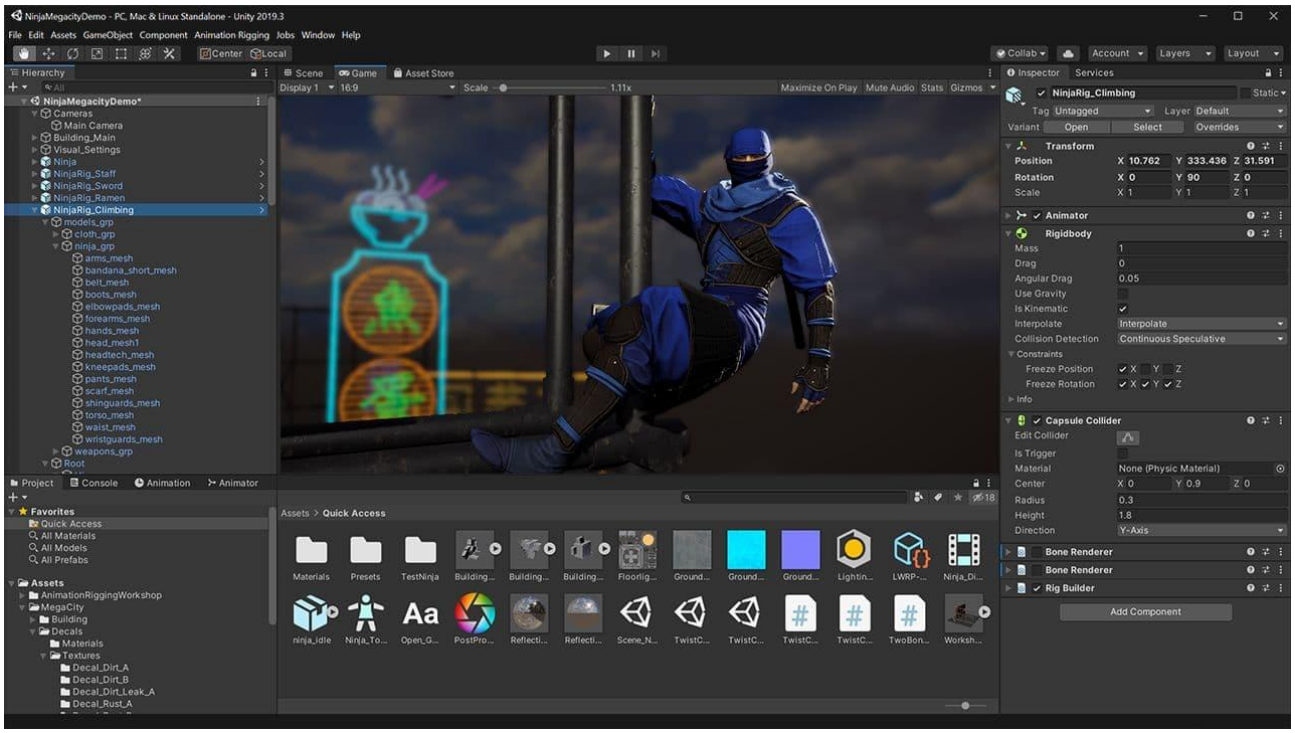


Рисунок 1.2 – Знімок екрану редактора Unity з відкритою графічною сценою

Unity має закритий вихідний код, що виключає можливість модифікації рушія загалом. Користувачі не мають прямого доступу до внутрішніх систем, і хоча багато аспектів проєкту можна налаштовувати через API, заміна або переробка фундаментальних підсистем, таких як система рендерингу, фізики чи сцени, є майже неможливою. Це особливо критично в тих випадках, коли розробник бажає реалізувати нетипові сценарії взаємодії агентів або нестандартну архітектуру. Це також виключає можилвість вивчення роботи рушія зсередини, що унеможливлює оптимізацію своїх систем для ефективної роботи з системами рушія.

Внутрішні процеси рушія непрозорі, а їхня поведінка часто залежить від прихованих механізмів. Це створює додаткові складнощі в налагодженні, особливо коли потрібна точна і передбачувана взаємодія між агентами.

Використання мови програмування C# для написання скриптів спрощує написання логіки об’єктів, проте призводить до гіршої продуктивності симуляції в м’якому реальному часі. Використання нативної мови C++ дозволило б симуляціям працювати швидше, але кожен користувач робить свій обмін між продуктивністю та простотою

Unity має беззаперечні переваги: активну спільноту, велику кількість документації, готові ресурси в Unity Asset Store, розширення функціоналу через плагіни, а також можливості інтеграції з хмарними сервісами. Це робить рушій придатним для комерційної розробки, швидкого прототипування і освітніх цілей. Однак його універсальність стає недоліком у випадках коли потрібна мінімалістична та контрольована платформа. У таких випадках доцільніше застосовувати спеціалізовані або власноруч створені рушії з нижчим рівнем абстракції, та більшим рівнем персоналізації для уникнення непотрібних накладних витрат та повним контролем над архітектурою.

OGRE (рис. 1.3) – Object-Oriented Graphics Rendering Engine – це кросплатформенний графічний рушій з відкритим вихідним кодом, орієнтований вкилючно на рендеринг тривимірної графіки. Рушій розробляється спільнотою з 2001 року, написаний мовою програмування C++. Він має модульну архітектуру та надає користувачу повний контроль над графічним пайплайном. OGRE не містить вбудованої системи симуляцій світу, фізики, звуку чи анімаційної логіки. Що робить його хорошим вибором для розробників з потребою в «легкому» рушії без зайвих систем, проте для декого це занадто радикальний вибір, який примушує розробляти багато систем власноруч.

Основна перевага OGRE полягає в «легкості», чистоті реалізації та розширюваності. OGRE не нав’язує специфічну архітектуру проєкту, не містить прихованих залежностей або систем, що працюють у фоновому режимі без прямого використання з боку розробника. Завдяки цьому він має перевагу в проєктах де важливе точне керування ресурсами та контроль над архітектурою,

і відсутність накладних витрат, що відрізняє його від великих універсальних рушіїв.

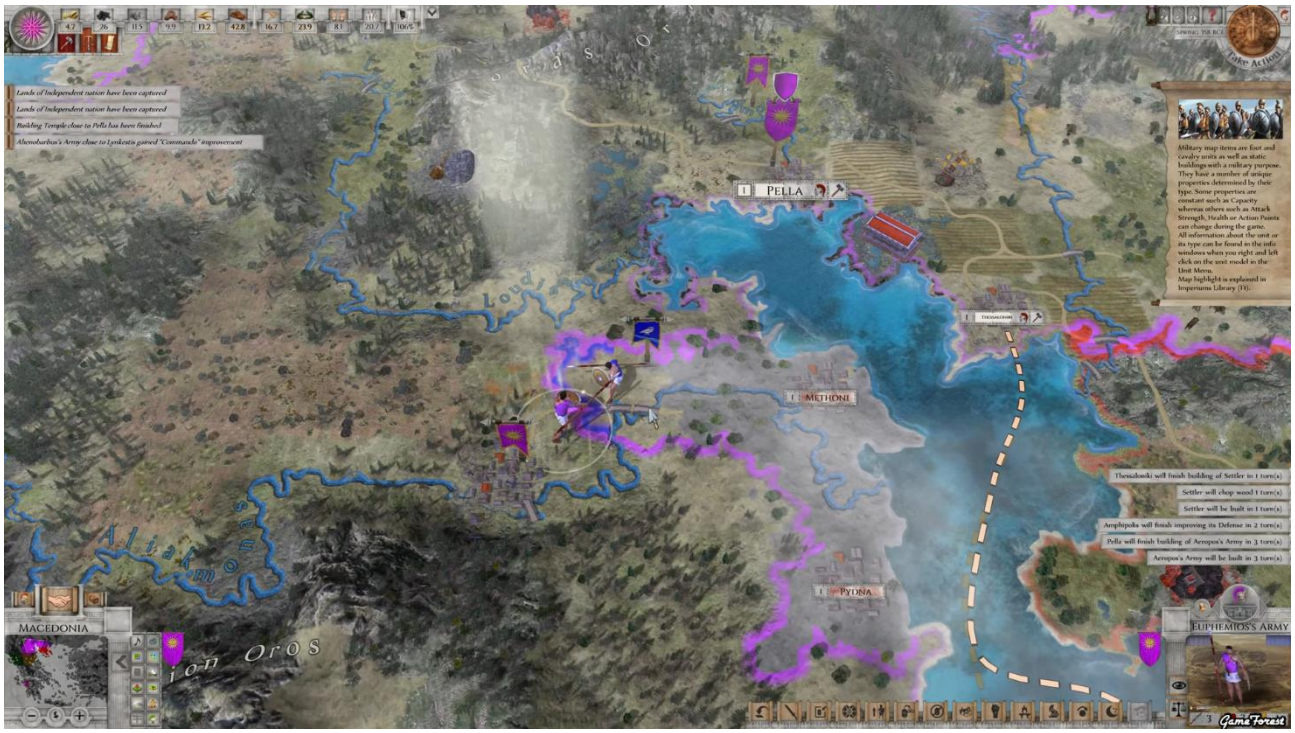


Рисунок 1.3 – Знімок екрану сцени візуалізованою рушієм OGRE

OGRE підтримує всі важливі графічні API, як OpenGL, Direct3D 9, Direct3D 11, та Vulkan, а також забезпечує кросплатформність для Windows, Linux, macOS, Android. Система матеріалів і шейдерів підтримує фіксований пайплайн і сучасні підходи через GLSL, HLSL одночасно. Новіші версії дозволяють використовувати PBR.

Наявність відкритого коду дозволяє вивчати рушій та модифікувати під свої потреби, це корисно у випадках де важливий прямий контроль над кожним кроком пайплайну.

OGRE не надає вбудованих рішень для логіки симуляції, таких як система об’єктів сцени, поведінка агентів і обробка подій. Користувач має самостійно реалізовувати ці частини або інтегрувати сторонні рішення, в разі їх потреби. Така гнучкість дозволяє будувати оптимізовані та специфічні системи без

обмежень, притаманних повнофункціональним рушіям, але вимагає значено більшого обсягу роботи.

OGRE часто застосовується в наукових візуалізаціях, інженерних симуляціях, освітніх інструментах та системах, де потрібно мати ізольовний графічний модуль. Його архітектура сприяє побудові кастомного рушія або фреймворку поверх графіки OGRE.

Таким чином OGRE добре відповідає поставленим вимогам «легкості» і спеціалізації, проте бракує системи яка забезпечувала б робочий процес між рушієм та симуляцією, інструментарію та фреймворку симуляції світу.

1.2. Обґрунтування теми і висновоки

Сучасні рушії, такі як Unreal Engine або Unity є потужними інструментами, але їхня «універсальність» робить їх занадто складними та ресурсоємними для багатьох спеціалізованих задач. Вони надають велику кількість вбудованих систем та високий рівень абстракції, які можуть бути надлишковими для певної категорії користувачів, в яких є потреба в легкості, гнучкості та високому рівні кастомізації і оптимізації. OGRE навпаки надає лише базові можливості рендерингу, чого більшості користувачів буде недостатньо.

Через відсутність оптимального рушія на поточному ринку, було прийнято рішення розробити власний рушій – VeiM Engine, що представляє з себе усереднення між оверкіл рушіями як Unreal Engine та обмеженими графічними рушіями з недостатнім функціоналом, такими як OGRE.

Переваги VeiM Engine включають мінімалізм та ясну архітектуру. В ньому запроваджений оптимальний рівень абстракції що дозволяє легко розширювати та модифікувати рушій, але не витрачає додаткових ресурсів. Він містить лише необхідні доробки що потрібні для розробки власних інтерактивних комп’ютерних симуляцій, проте без всіх складностей повноцінного ігрового рушія.

Серед них:

- ядро – основа для симуляції, що містить такі системи як: рефлексія, система логічного світу, рендеринг, серіалізація, введення, фреймворк для програмування об’єктів;
- редактор – набір інструментів що дозволяє працювати над створенням проєкту симуляції;
- інфраструктура – забезпечує створення проєкту і зручну взаємодію між ним та редактором рушія.

VeiM також надає гнучкість і простоту інтеграції. Навідміну від монолітних рушіїв, він пріоретизує невелике ядро, обмінюючи вбудовані системи на гнучкість. Компонетно-орієнтований підхід дозволяє будувати складні системи через композицію, але з можливістю використання спадкування за потреби.

В рушії пріоретизовано оптимізацію під спеціалізовані задачі. Оскільки даний рушій підтримує лише платформу Windows, його архітектура простіша та ефективніша, порівняно з крос-платформними рішеннями. Це дозволить оптимізовувати майбутні проєкти та зменшить використання дискового простору.

Таким чином, VeiM є оптимальним вибором для розробника, якому потрібен легкий, швидкий і повністю контрольований рушій, з базовими системами та зручними інструментами.

2. АНАЛІЗ ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1. Вибір мови програмування

Одним з найважливіших технічних рішень у процесі розробки рушія є вибір мови програмування. Від нього залежить не лише стиль і складність розробки, а й продуктивність системи, можливісті її масштабування, інтеграція з апаратним та програмними інтерфейсами, доступність інструментів, а також сумісність з іншими компонентами рушія – такими як графічні та фізичні API. У випадку створення власного рушія, який має бути гнучким, продуктивни і технічно розширюваним, питання вибору мови формує основу свієї архітектури.

На сьогоднішній день фактичним стандартом індустрії розробки рушіїв є мова C++. Цю мову для розробки ядра рушія обирають як великі студії для створення коменрційно успішних продуктів, так і невеликі команди розробників при створенні проектів з відкритим кодом. Наприклад, модульна структура, висока продуктивність, підтримка візуального програмування та інтеграція з редактором в Unreal Engine реалізовані за рахунок можливостей C++, зокрема ручного керування пам’ятттю, шаблонного програмування, метапрограмування тощо. Хоча багато рушіїв дозволяють створення логіки іншими мовами (наприклад Lua, C#, GDScript), ядро рушія використовує C++.

Причиною такої тотальної пристуності C++ є поєднання двох критично важливих характеристик: максимальної продуктивності та гнучкого контролю над низькорівневими процесами. Завдяки компіляції у машинний код і можливості точного керування пам’яттю, розробники можуть досягати високої швидкодії, що є особливо важливим у графічно насичених тривимірних сценах, де час обробки одного кадру обмежується мілісекундами. Окірм того, C++ дозволяє безпосередньо працювати з графічними API, такими як Vulkan, DirectX 12, що вимагають точного управління ресурсами GPU. Ще одним фактором є екосистема: величезна кількість готових бібліотек для роботи з

графікою, фізикою, звуком, мережами, та для інтеграції графічного інтерфейсу і внутрішніх систем.

Історично велику роль у розробці системного програмного забезпечення відігравала мова C. Вона вирізняється своєю простотою, прямим доступом до пам’яті, відсутністю зайвих абстракцій та чудовою продуктивністю. Через це C активно використовувалась у створенні графічних біблотек, операційних систем та драйверів. Існують приклади рушіїв або рушійних біблотек, створених на C, зокрема id Tech 1 та id Tech 2. Проте в сучасних умовах C вже не вважається зручним вибором для побудови повноцінного рушія. Її обмежені можливості в області абстракції та відсутність об’єктно-орієнтованого підходу значно ускладнюють розробку складних систем – таких як модульна архітектура рушія, система компонентів (ECS), щейдерна система або редактор матеріалів. В той час як C++ надає всі можливості мови C, але при цьому дозволяє створювати масштабовану, безпечнішу й ефективнішу архітектуру. Саме тому сучасні рушії практично повністю відмовились від використання C як основної мови розробки, надаючи перевагу C++ як більш гнучкому, структурованому та адаптованому до великих проєктів інструменту.

З меою повноти аналізу слід розглянути також альтернативні мови програмування, які викоистовують в індустрії для побудови рушіїв. Однією з найпоширеніших альтернатив є C#. Найвідомішим прикладом рушія з цією мовою, є Unity/ Хоча сам Unity Engine містить низькорівневі компоненти на C++, ядро високорівневої логіки та велика частина рушія реалізовані на C#. Ця мова забезпечує вищий рівень абстракції, зручні засоби об’єктно-орієнтованого програмування та потужну стандартну бібліотеку. Проте важливо зазначити, що C# працює через середовище виконання .NET або Mono, що додає проміжний шар між програмою і апаратним забезпеченням. Це призводить до втрати частини продуктивності, а також ускладнює контроль над системною пам’яттю та оптимізацією роботи з GPU. У рамках створення повноцінного рушія це стає критичним недоліком. Тому по при те що C# добре підходить для реалізації

логіки об’єктів, або створення продуктів на вже існуючих рушіях, для розробки ядра нового рушія вона не є достатньо ефективною.

Іншою цікавою сучасною мовою є Rust, що позиціонується як безпечна, продуктивна альтернатива C++, що не потребує збирача сміття. Вона довзоляє уникати класичних помилок роботи з пам’яттю за рахунок системи володіння ресурсами. Прикладом рушія, реалізованого на Rust, є Bevy – експериментальний ECS-орієнтований рушій з відкритим кодом. Однак, незважаючи на інноваційність екосистема Rust ще не досягла того рівня зрілості, що є у C++. У Rust бракує кількості стабільних, перевірених бібліотек для рендерингу та роботи з графікою; інтеграція з API таким як Vulkan є можливою, але складнішою. Використання Rust є неоптимальним для розробки ядра рушія.

Також варто згадати Java, яка історично використовувалась у рушіях для мобільних і веб-проєктів. Наприклад рушій jMonkeyEngine повністю написаний на Java. Він орієнтований переважно на навчання або незалежні проєкти, оскільки Java працює у віртуальній машині (JVM), що значно обмежує продуктивність. Збирач сміття у JVM може викликати паузи під час виконання, що критично для ситем реального часу. Через це Java фактично втратила свою конкурентноздатність у розробці рушіїв, де потрібна висока продуктивність.

Ще одна категорія мов – це скриптові мови, зокрема Python, Lua або JavaScript. Вони ніколи не використовуються для створення ядра рушія через низьку швидкодію та відсутність прямого доступу до апаратного рівня. Наприклад, Lua активно застосовується в рушіях CryEngine і Defold, але виключно для логіки сценаріїв або штучного інтелекту, тоді як ядро рушія залишається на C++.

Отже з урахуванням аналізу мов, що реально вискористовується для побудови рушіїв, можна зробити висновок, що C++ залишається єдиною повноцінною мовою, яка повністю відповідає технічним вимогам даного проєкту. Вона забезпечує баланс між максимальною продуктивністю, контролем, наявністю інструментів, підтримкою з боку спільноти та досвідом

індустрії. Саме тому C++ було вибрано для реалізації рушія у рамках цієї дипломної роботи.

2.2. Робота з графікою

У процесі створення тривимірного графічного рушія однією з найважливіших технічних складових є рендеринг – процес відтворення сцени у вигляді зображення на екрані. На відміну від створення невеликих ігор або візуалізацій симуляцій, де іноді достатньо високорівневих графічних біблотек на кшталт SDL, SFML чи Allegro, повноцінний тривимірний рушій вимагає глибшого котролю над графічним ковеєром. Використання бібліотек, які приховують низькорівневу реалізацію, обмежує можливості щодо продуктивності, доступу до буферів, шейдерів, оптимізації пам’яті GPU та багатьох інших складових, які критично важливі для сучасної графіки.

Саме тому для реалізації тривимірного рушія необхідно використовувати низькорівневий графічний API – набір функцій і структур, які безпосередньо взаємодіють із драйвераи відеокарти та GPU. Графічні API надають засоби створення буферів, опису геометрії, завантаження текстур, компіляції шейдерів і управління рендер-пайплайном. Серед сучасних API, які активно використовуються в індустрії, можна виокремити три основні: OpenGL, Vulkan і DirectX.

OpenGL було обрано як основне графічне API для реалізації рендерингу у рамках цієї роботи. Основною причиною такого вибору є відносна простота вивчення та використання. OpenGL має добре задокументовану, стабільну архітектуру, з великою кількістю навчальних матеріалів, прикладів відкритого коду. OpenGL функціонує за принципом детермінованого кінцевого автомату, де контекст зберігає параметри, які застосовуються до об’єктів рендерингу. Це дозволяє швидко отримати перші результати й зосередитись на розробці системи

рушія, а не витрачати значні ресурси на реалізацію низькорівневої логіки керування GPU.

OpenGL хоч і вважається застарілим у порівнянні з Vulkan, однак на практиці продовжує використовуватись у багатьох комерційних і незалежних проєктах, як основним так і альтернативним API. Універсальність і гнучкість OpenGL дозволяють реалізовувати навіть склдні техніки освітлення, тіней, постобробки та PBR-рендерингу. У поєднанні з GLSL він надає достатньо зособів для створення сучасного графічного ядра.

Окрім простоти, OpenGL також має потенціал для кросплатформенності – він підтримується на Windows, macOS, Linux, а також на мобільних платформах. Хоча поточна розробка рушія орієнтована виключно на Windows, у майбутньому OpenGL залишає можливість додавання підтримки інших платформ з мінімальними змінами рендер-системі. Це додає гнучкості майбутньому розвитку проєкту.

Для об’єктивного вибору OpenGL важливо порівняти з його сучасними альтернативами.

Vulkan – нове покоління графічного API, розроблене консорціумом Khronos Group. Vulkan надає розробнику повний контроль над GPU, багатопоточну обробку команд, асинхронну обробку ресурсів, ефективне управління пам’яттю й можливість тонкої оптимізації під конкретне обладнання. Він став ядром для багатьох сучасних рушіїв і використовується гігантами індустрії. Проте за всіма цими перевагами знаходиться головний недолік – надзвичайна склдність вивчення та впровадження. Vulkan має набагато склдніший і об’ємніший API порівняно з OpenGL. Розробнику необхідно вручну описувати більшість структур, синхронізацію, керування пам’яттю та інші компоненти, що значно збільшує обсяг коду навіть для найпростішої сцени. Крім того, широкі можливості Vulkan можуть призвести до гіршої продуктивності за рахунок неоптимального використання його інструментів через підвищену можливість здійснення помилок в реалізації. Для цієї роботи це є суттєвим

бар’єром, тому Vulkan є неоптимальним варіантом для розробки власного рушія в наявних умовах.

DirectX, зокрема його остання версія DirectX 12 – графічний API від компанії Microsoft, як є стандартом для Windows-платформи. DirectX активно використовується в великих рушіях. Він має розвинуту підтримку сучасних графічних технологій і забезпечує глибоку інтеграцію з Windows-драйверами. Проте DirectX є пропрієтарним та платформо-залежним API, тобто використання його обмежує можливості в подальшому перенесенні рушія на інші платформи. Окрім того, DirectX 12, подібно до Vulkan вимагає великого обсягу ручного кодування та управління ресурсами, що робить його менш придатним для проєкту.

Варто зазначити що вибір графічного API для рушія не є кінцевим, оскільки є можливим розширення для підтримки одразу декілької API в майбутньому, що часто зустрічається в сучасних рушіях. Через що для вибору першого графічного API важливішими критеріями є простота вивчення і використання, та швидкість розробки.

Таким чином, з урахуванням усіх факторів – простоти використання, наявності великої кількості навчальних матеріалів, придатності для швидкої реалізації рендер-системи, а також потенційної кросплатформенності – було прийнято рішення використовувати OpenGL як основний графічний API для реалізації рушія в межах даної роботи.

2.3. Управління вікнами

Одним із перших технічних етапів при розробці рушія є створення вікна програми та ініціалізація графічного контексту. Це фундаментальна складова, без якої неможливо розпочати роботу з графічним API чи візуалізацією роботи інших систем через графічний інтерфейс користувача. Вікно є посередником між операційною системою та рушієм: воно обробляє події клавіатури, миші, зміну

розміру буфера та інші взаємодії з користувачем. Тому вибір технології для створення вікна впливає не лише на інтеграцію графіки, а й на подальшу архітектуру рушія.

Найбільш прямолінійний підіхд на Windows – це використання нативного WinAPI. Windows API надає повний набір функцій для створення вікон, обробки повідомлень, підключення графічного контексту та управління опдіями. Деякі великі рушії особливо комерційні або історично давні, можуть використовувати WinAPI напряму, щоб отримати повний контроль над поведінкою вікна та інтегрувати глибоку підтримку Windows-специфічних особливостей. Проте використання WinAPI має кілька суттєвих недоліків для сучасної індивідуальної розробки. По-перше, цей підхід значно збільшує обсяг коду, який потрібно написати навіть для базової функціональності. Створення вікна, обробка повідомлень, підключення OpenGL-контексту –все це потребує ручної реалізації, часто з використанням низькорівневих макросів і структур. По-друге, цей підхід жорстко прив’язує рушій до платформи Windows, що суперечить навіть потенційній ідеї кросплатформенності в майбутньому.

Для цього проєкту було обрано використовувати готову бібліотеку GLFW (Graphics Library Framework) для створення вікон і обробки подій. GLFW – це кросплатформна бібліотека, яка значно спрощує роботу з OpenGL, створенням вікна, обробкою вводу з клавіатури та миші, а також управлінням контекстом рендерингу. Вона є відкритою та активно підтримується спільнотою. Завдяки GLFW створення вікна й ініціалізація OpenGL значно спрощується в порівнянні з WinAPI. Бібліотека також дозволяє легко налаштовувати параметри вікна – роздільну здатність, повноекранний режим, версію OpenGL, підтримку глибини та подвійної буферизації. Ще однією перевагою GLFW є простота інтеграції з іншими бібліотеками – наприклад, з бібліотекаою завантаження зображень stb_image, системами управління ресурсами, або з UI-бібліотеками на кшталт ImGui, що використовуються в інструментальній частині рушія. Завдяки

широкій популярності GLFW, існує велика кількість прикладів, документації й підтримки, що зменшує технічні ризики при розробці.

Отже, з огляду на простоту використання, зменшення обсягу платформозалежного коду, потенціал кросплатфрмності та активну підтримку спільноти, бібліотека GLFW є оптимальним вибором для реалізації вікнного інтерфейсу в менах даного рушія.

2.4. Математичне ядро

Математична бібліотека є важливим компонентом будь-якого рушія. Вона забезпечує базову функціональність, без якої неможлива реалізація графічного рендерингу, фізики, анімації, трансформацій та роботи з камерою. До основних елементів, які повнинна надавати така бібліотека, належать: вектори (2D, 3D, 4D), матриці (зокрема 4x4), кватерніони, функції обчислення довжини, нормалізації, добутків, перетворення координат, обертання, масштабування, проєкції, а також розширені математичні операції, як інтерполяція.

У більшості сучасних рушіїв використовується власна пропрієтарна математична бібліотека, що глибоко інтегрована в архітектуру рушія. Такий підхід забезпечує максимальну гнучкість, контроль над точністю обчислень та оптимізацією під специфічні задачі. Проте розробка подібної бібліотеки з нуля вимагає значних ресурсів і достатньої кваліфікації. Необхідно не лише реалізувати широкий спектр математичних операцій, але й провести ретельне тестуванн, щоб гарантувати їхню коректну та стабільну роботу в усіх можливих випадках.

Більше того, власна математична бібліотека має демонструвати кращі або принаймні не гірші результати, ніж сторонні рішення. Це стосується як продуктивності (швидкість обчислень, використання SIMD-інструкцій), так і точності, зручності API, відповідності математичним стандартам. Досягти такого результату, особливо в умовах обмеженого часу та людських ресурсів

майже неможливо. У зв’язку з цим у рамках даної роботи створення власної математичної бібліотеки вважається недоцільним.

Замість цього варто використати готову, перевірену спільнотою бібліотеку, яка вже має багаторічний досвід використання в розробці рушіїв та іншого графічного програмного забезпечення. Найбільш популярним і логічним кандидатом у цьому випадку є GLM(OpenGL Mathematics) – загальнодоступна C++ бібліотека, яка проєктувалась спеціально для роботи з OpenGL. Її API натхненне мовою шейдерів GLSL, що дозволяє писати код у схожому стилі та легко переносити логіку з грфічних шейдерів у процесорну частину рушія. GLM підтримує всі стандартні структури, а також широкий набір функцій для геометричних і алгебраїчних операцій.

GLM реалізована як «header-only» бібліотека, тобтно не потребує компіляції окремих об’єктних файлів. Це значно полегшує її інтеграцію в проєкт, не створюючи зщалежності від зовнішніх збірок. GLM також підтримує оптимізації через SIMD-інструкції, що дозволяє досягати високої продуктивності при масових обчисленнях, таких як оновлення трансформацій або фізичні симуляції.

Іншим менш поширеним варіантом є бібліотека DirectXMath, яка використовується в рушіях на базі DirectX. Вона також підтримує SIMD-оптимізації, але має API орієнтоване під Windows-платформу, і структуру, яка менш зручня для розробника, ніж у GLM. ЇЇ ліцензія та фокус на DirectX роблять її менш привабливою для рушія, що базується на OpenGL.

Врахочуючи простоту інтеграції, відповідність сучасним вимогам, активну підтримку, високу продуктивність та зручне API, GLM є оптимальним вибором для математичної підтримки рушія.

2.5. Графічний інтерфейс

Графічний інтерфейс (GUI) є важливою складовою інструментарію рушія, коли мова йде про створення редакторського середовища. Редактор – окрема утиліта або режим роботи рушія, який дозволяє взаємодіяти з проєктом через графічний інтерфейс. Через редактор здійснюється доступ до сцен, розміщення об’єктів, редагування їх властивостей, перегляд ресурсів, редагування шейдерів, запуск симуляцій в редакторі тощо. Графічний інтерфейс редактора виконує роль головного інструменту для взаємодії з рушієм, і від його реалізації залежить зручність використання всього рушія. Якісний GUI значно спрощує розробку продуктів на рушії, прискорює ітерації та робить роботу інтуїтивно зрозумілою.

В межах цієї роботи передбачено створення власного редактора для рушія, і, відповідно, постає потреба в інструменті для реалізації графічного інтерфейсу для нього. Для цього існує кілька можливих підходів, кожен із яких має свої переваги та недоліки.

Один з підходів полягає у створенні власної системи GUI з нуля, використовуючи можливості самого рушія. Це означає написання коду для рендерингу віджетів (вікон, кнопок, тексту, слайдерів), обробку подій введення, побудову логіки ієрархії елементів, системи розміщення (layout) тощо. Подібним шляхом пішли великі рушії, наприклад Unreal Engine має Slate, а Unity розробив UI Toolkit.

Переваги такого підходу:

- Повна гнучкість і контроль над зовнішнім виглядом і поведінкою інтерфейсу.
- Можливість глибокої інтеграції з рушієм, наприклад, виведення сцени у віджет, що є частиною layout.
- Використання цієї ж системи для рендерингу внутрішнього інтерфейсу симуляції
- Уніфікація стилю рушія та редактора.

Однак цей підхід вимагає великих ресурсів. Розробка системи GUI з усіма базовими функціями, і її тестування, може зайняти місяці або роки. Крім того, така система потребує регулярної підтримки, розширення, виправлення помилок. В існуючих умовах реалізація повноцінного власного GUI-фреймворку є нереалістичною, оскільки відволікає від основної задачі та вимагає забагато ресурсів.

Бібліотека Dear ImGui була створена спеціально для швидкої реалізації інструментів графічного інтерфейсу в застосунках реального часу. Вона використовує імперативний підхід – інтерфейс описується у вигляді функцій, що викликаються в кожному кадрі. Це робить бібліотеку дуже простою для інтеграції з будь-яким рушієм і графічним API.

ImGui має набір готових віджетів: кнопки, поля введення, дерева, таблиці, вкладки, графіки, плаваючі вікна і навіть можливість реалізації drag-and-drop систем. Вона не потребує складного опису структури GUI або декларативного програмування – усе будується безпосередньо у коду C++. Важливою перевагою також є висока продуктивність і низький поріг входу. Завдяки цьому бібліотека часто використовується у великих студіях для створення внутрішніх інструментів та редакторів, або інструментів для налагодження. Недоліками ImGui є обмежена гнучкість стилізації та відсутність сучасних layout систем, але для заадч редактора ці мінуси не є критичними. Таким чином, вибір ImGui є оптимальним компромісом між зручністю розробки, швидкістю реалізації та достатньою функціональністю.

Ще одним варіантом є використання зовнішніх GUI-фреймворків, таких як Windows Presentation Foundation (WPF) – сучасна платформа для створення графічних застосунків для Windows з використанням C# та XAML. ЇЇ перевагами є потужна декларативна система опису інтерфейсу, підтримка стилів, анімацій, шаблонів і інтеграція з .NET-інструментами. Проте для рушія на C++ використання WPF створює значні технічні складнощі, а саме: необхідість взаємодії між C++ і C#, що потребує міжмовної інтеграції і залежність від

Windows. Окрім технічних обмежень, важливою відмінністю є також підхід до оновлення інтерфейсу. ImGui є повінстю динамічною бібліотекою, яка оновлює графічний інтерфейс кожного кадру, що дозволяє одразу бачити результати будь-якийх змін. У випадку з WPF інтерфейс є переважно статичним, і зміни в неьому потребують виклику окремих оновлень, що ускладнює реалізацію інтерфейсу в динамічному середовищі рушія і призводить до помилок. Таким чином попри зручність для створення класичних UI-застосунків, WPF не підходить як основа для редактора рушія.

Таблиця 2.1

Порівняльна таблиця бібліотек графічного інтерфейсу

Критерій	Власний фреймворк	ImGui	WPF
Мова реалізації	C++	C++	C# + XAML
Простота впровадження	Висока складність	Низька складність	Середня складність
Гнучкість дизайну	Максимальна	Обмежена	Висока
Інтеграція з рушієм	Максимальний рівень	Високий рівень	Ускладнена
Кросплатформність	Можлива	Повна	Відсутня
Продуктивність	Висока	Висока	Низька
Зміни в реальному часі	Миттєве оновлення	Миттєве оновлення	Потребує ручного оновлення

З огляду на доступні ресурси та технічні цілі, найкращим вибором є використання ImGui. Це дозволяє швидко отримати функціональний, інтерактивний інтерфейс без значних витрат часу, з можливістю глибокої інтеграції.

2.6. Висновки

У цьому розділі дипломного проєкту було проведено комплексний аналіз засобів реалізації, необхідних для розробки рушія. На основі технічних та практичних критеріїв були обґрунтовані ключові фактори вибору, що забезпечують баланс між продуктивністю, зручністю розробки та перспективами масштабування проєкту. Вибрана мова програмування C++ є галузевим стандартом для створення рушіїв завдяки високій продуктивності, контролю над пам’яттю, а також активному екосередовищу. При виборі графічного API було визначено доцільність використання низькорівневого інтерфейсу замість абстрактних бібліотек, що дає змогу реалізувати власну гнучку систему рендерингу. Серед доступних API обрано OpenGL як найбільш оптимальний варіант для цього проєкту завдяки простоті освоєння, широкій документації та достатній функціональності і продуктивності. Хоча інші графічні API пропонують більше контролю і сучасні можливості, складність їх інтеграції та підтримки не виправдана. Для створення вікна та обробки вводу вибір зроблено на користь бібліотеки GLFW, яка значно спрощує створення OpenGL-контексту та обробку подій, у порівнянні з використанням нативного WinAPI. Для математичного ядра було обґрунтовано доцільність використання готового рішення, зокрема бібліотеки GLM. Хоча написання власної бібліотеки могло б дати більше контролю, це потребує значних ресурсів, а також ризикує поступитися продуктивністю й надійністю існуючим рішенням без гарантій успіху. У завданні побудови графічного інтерфейсу редактора було розглянуто три підходи, серед яких було обрано використання бібліотеки ImGui, як оптимальне рішення завдяки його динамічній природі, простоті інтеграції, та миттєвій реакції на оновлення в реальному часі.

Прийняті технологічні рішення дозволяють побудувати надійну основу для функціоналу рушія, використовуючи перевірені інструменти, які відповідають сучасним стандартам індустрії.

3. РОЗРОБКА РУШІЯ

3.1. Архітектура рушія

Архітектура рушія є фундаментом усієї системи і визначає, яким чином взаємодіють між собою основні компоненти, як організовано масштабування проєкту, модульність, повторне використання коду та підтримка функціональності редактора. В основі розробленого рушія закладена багаторівнева модульна структура, що дозволяє ізолювати логіку ядра, інструментів розробника та кінцевого продукту що розробляється на рушії, забезпечуючи гнучкість і керованість при розробці.

Найвищий рівень абстракції включає три основні модулі: ядро рушія, редактор, а також користувацький проєкт, який створюється на базі рушія. Такий розподіл дозволяє підтримувати чітку межу між інструментарієм програмної логіки та внутрішніми компонентами рушія, що забепзечують базову функціональність: управління ресурсами, сценою, рендерингом, логікою та параметрами об’єктів тощо.

3.1.1. Ядро рушія

Ядро – це центральний модуль рушія, який реалізує ключову функціональність незалежно від конфігурації збірки. Воно включає в себе всі низькорівневі підсистеми, які визначають роботу симуляційного світу: обробку сцени, об’єктів, компонентів, подій, часу, ресурсів, систему рендерингу, систему вводу тощо. Уся логіка, що має бути спільною як для конфігурації з редактором так і самостійної, зосереджена саме тут.

Ядро рушія розроблене таким чином, щоб його можна було збирати як статично, так і динамічно, залежно від потреби:

- У конфігурації для редактора ядро компілюється як динамічна бібліотека (DLL) і завантажується редактором під час виконання.

- У конфігурації без редактора, тобто у самостійній збірці симуляції, ядро компілюється як статична бібліотека і лінкується до застосунку симуляції на етапі компіляції.

Такий підхід дозволяє забезпечити максимальну ізоляцію ядра, зберігаючи можливість незалежного тестування, відлагодження та повторного використання.

3.1.2. Редактор

Редактор – це окремий застосунок, який збирається лише у відповідних конфігураціях, призначених для розробника. Він реалізує інтерфейс користувача, що дозволяє створювати, редагувати та тестувати сцени, працювати з ресурсами, матеріалами, шейдерами, компонентами, об’єктами та іншими сутностями рушія. Вся функціональність редактора базується на викликах API ядра, проти якого лінкується редактор.

Редактор не містить у собі жодної логіки сцен. Його призначення працювати з всім вмістом проєкту за допомогою системи рефлексії, що дозволяє динамічно взаємодіяти з типами, властивостями та об’єктами, які реалізовані в окремому незалежному від редактора модулі користувача. Ця система рефлексії буде розглянута в одному з наступних підрозділів.

Оскільки редактор є виключно інструментом розробки, він не потрапляє у фінальні збірки і не використовується на етапі запуску симуляції. Завдяки цьому, при складанні релізної версії можна значно зменшити обсяг застосунку, уникнувши непотрібних залежностей.

3.1.3. *Користувацький проєкт*

Третій головний модуль – це користувацький проєкт, який створюється розробником на основі рушія. Його основне призначення – реалізувати специфічну логіку сцени, за допомогою визначення логіки, взаємодій, і візуального стилю об’єктів і їх компонентів.

Користувацький проєкт залежить від ядра, використовуючи його API для взаємодії зі сценою, ресурсами, рендерингом, подіями тощо. При цьому проєкт користувача абсолютно незалежний від реадктора.

Збірка користувацького проєкту відбувається у двох основних режимах:

1) У режимі розробки, коли симуляція тестується через редактор, її код збирається в окрему динамічну бібліотеку, яка динамічно завантажується редактором під час його роботи.
2) У режимі релізу, коли симуляція збирається як окремий застосунок, проєкт компілюється у виконуваний файл, який лінкується з статично зібраним ядром.

Такий підхід дозволяє досягти високої роздільності та чистоти структури проєкту. Симуляція в релізі не містить жодного коду редактора, що зменшує обсяг та недозволяє використання ресурсоємних функцій редактора.

3.1.4. Конфігурації збірки

Рушій передбачає кілька конфігурацій збірки, що поділяються за типом призначення, та можуть комбінуватись:

1) «Editor» конфігурація.

   Включає виконуваний файл редактора, який лінкується з динамічною бібліотекою ядра та завантажує динамічну бібліотеку користувацького проєкту під час роботи. Це дозволяє оновлювати, відлагоджувати або перезапускати частини рушія чи проєкту без повної перезбірки всього застосунку.
2) Самостійна збірка.

   Створює самостійну збірку користувацького проєкту. У цьому режимі редактор не присутній, ядро лікується статично, і результатом є самостійний застосунок симуляції.
3) «Debug» збірка.

   Конфігурація що має відключені оптимізації та містить

відлагоджувальні символи для зручного відлагодження програми. Ця конфігурація має використовуватись лише для пошуку і виправлення помилок.

4) «Development» збірка.

   Конфігурація з включеними оптимізаціями, але все ще містить відлагоджувальні символи. Вона є основною конфігурацією, якою має користуватись розробник, що користується рушієм. Така конфігурація забезпечує високу продуктивність, що є максимально близькою до релізної, проте залишає можливість відлагодження.

5) «Shipping» збірка.

   Конфігурація що використовується для фінальної збірки користувацького проєкту, потрібна для поширення застосунку серед кінцевих користувачів.

Таким чином, всі типи конфігурацій утворють п’ять доступних конфігурацій збірки:

1) «Debug»;
2) «Debug Editor»;
3) «Development»;
4) «Development Editor»;
5) «Shipping».

3.1.5. Додаткові модулі

В архітектурі VeiM присутні додаткові модулі, що відносяться до інстурментарію рушія, і не присутні в збірках користувацького проєкту і редактора.

Модуль «DesktopPlatform» – розробницький модуль що реалізує функції рівня платформи, для взаємодії з API операційної системи.

Модуль «VeimManagerTool» – окрема утиліта, що використовує модуль «DesktopPlatform» і застосовується для менеджменту рушія і проєктів що

створюються на ньому. Ця утиліта використовує регістр Windows для реєстрації директорії рушія з ідентифікатором, для можливості управління одразу декількома версіями рушія, що одночасно встановлені користувачем, а також створює формат файлу «vmproject». Файл з форматом «vmproject» надає користувачу можливість управління власним проєктом який він розробляє з використанням рушія VeiM.

3.1.6. Підсумок

Архітектура рушія, побудована на принципах модульності, ізоляції відповідальностей та підтримки конфігурацій, створює надійну основу як для розробки, та і для масштабування проєктів. Чітке розділення ядра, редакторських інструментів та користувацького коду забезпечує не лише структурну чистоту, а й гнучкість у побудові робочого процесу.

Такий підхід дозволяє розробникам що використовують рушій VeiM ефективно змінювати, налагоджувати та розширювати функціональність без ризику порушення основних механізмів рушія. А підтримка різних типів збірок дає змогу адаптувати рушій до різних етапів життєвого циклу користувацького проєкту.

Впроваджена архітектура не лише полегшує підтримку рушія в довгостроковій перспективі, а й відкриває шлях до створення повноцінної екосистеми розробки з можливістю інтеграції нових модулів, автоматизації процесів та побудови гнучного інструментарію для кінцевого користувача.

3.2. Інфраструктура роботи з проєктами

Оскільки VeiM націлений на забезпечення зручного робочого процесу під час створення продуктів з його використанням, дуже важливим елементом рушія є інфраструктура що забезпечує роботу з проєктами і взаємодію проєктів з модулями ядра і редактора. Після розгляду загальної архітектури рушія в

попередньому підрозділі, було визначено що вона забезпечує можливість взаємодії ядра з модулем корстувацького проєкту. Мета цього підрозділу описати механізми, які дозволяють зручно створювати, відкривати й керувати такими проєктами. Ці механізми включають реалізацію системи управління проєктами на стороні редактора, а також динамічне завантаження і розвантаження користувацьких модулів, яке здійснюється на рівні рушія

3.2.1. Управління проєктами

Після запуску редаткора без зазначення параметрів командрного рядка, користувач бачить вікно вибору проєкту – браузер проєктів, що зображений на рис. 3.1. У цьому вікні є можливість або обрати вже існуючий проєкт, або створити новий. Цей процес максимально автоматизований і спрямований на зниження порогу входу: для створення нового проєкту достатньо вказати назву та місце розташування, після чого система ініціалізує структуру проєкту з типовим шаблоном вихідного коду, системними файлами для збірки, а також конфігураційним файлом проєкту з розширенням «.vmproject».

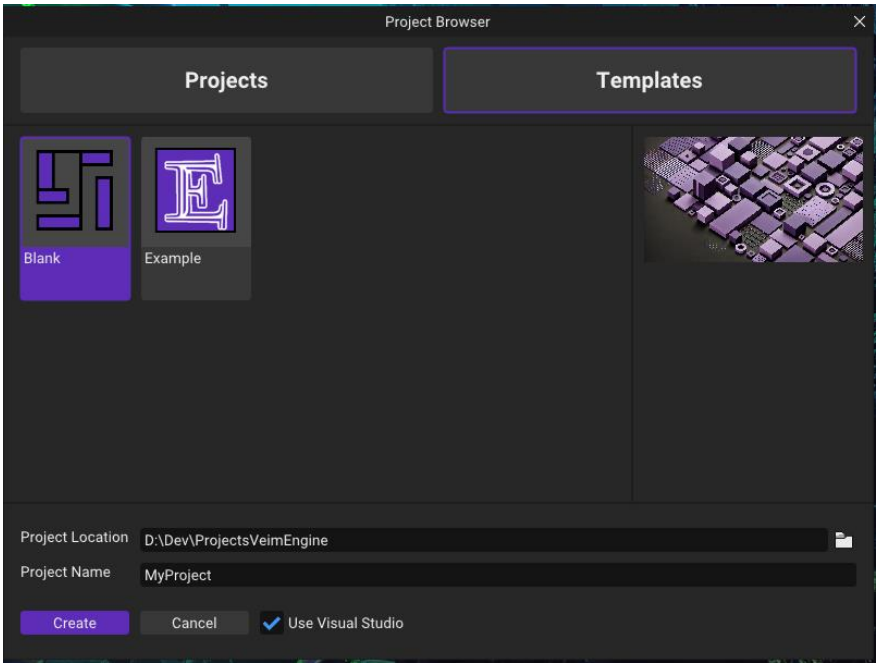


Рисунок 3.1 – Бразуер проєктів

Файл проєкту виконує кілька функцій:

- Слугує дескриптором проєкту при відкритті через редактор;
- Дозволяє відкривати проєкт в редакторі;
- Містить необхідну інформацію для генерації середовища розробки;
- Дозволяє генерувати файли середовища розробки;
- Дозволяє запускати симуляцію без запуску самого редактора.

Центральним елементом системи є спосіб, у який редактор взаємодіє з користувацьким проєктом. При виборі проєкту через браузер, редактор завершує поточний процес і запускає новий процес редактора, передаючи в командному рядку шлях до «.vmrpoject» файлу. Якщо ж запуск відбувається безпосередньо через файл проєкту, редактор одразу отримує цей шлях як аргумент командрного рядка, минаючи браузер проєктів. У будь-якому з цих випадків рушій завантажує відповідний користувацький модуль на основі заданого проєкту.

1.1.2. Менеджер модулів

Іншою частиною інфраструктури є механізм завантаження користувацького модуля симуляції. Він реалізований через клас ModuleManager (рис. 3.2), який відповідає за динамічне відключення бібліотек під час виконання. У контексті рушія користувацький модуль компілюється як динамічна бібліотека (DLL), яка потім завантажується редактором через стандартний WinAPI, за допомогою функції LoadLibraryW.

```cpp
#pragma once
#include "CoreDefines.h"

#include <unordered_map>

#if IS_UNIFIED
#define IMPLEMENT_MAIN_GAME_MODULE(Name) static VeiM::String s_AppName = #Name; static VeiM::String s_AppTitle = #Name;
#define VM_GAME_MODULE_IMPLEMENTATION <Application/EntryPoint.h>
#else
#define IMPLEMENT_MAIN_GAME_MODULE(Name)
#define VM_GAME_MODULE_IMPLEMENTATION "CoreDefines.h"
#endif

namespace VeiM
{
    // Used to load game dll for editor
    class CORE_API ModuleManager
    {
    public:
        ~ModuleManager();
        static ModuleManager& Get();

        void LoadModule(const std::wstring moduleName);
        void UnloadModule(const std::wstring moduleName);
        void SetGameBianariesDir(const fs::path& gameDir);

    private:
        ModuleManager() = default;
        ModuleManager(const ModuleManager&) = delete;
        ModuleManager& operator=(const ModuleManager&) = delete;
    private:
        std::unordered_map<std::wstring, void*> m_Modules;
    };
}
```

Рисунок 3.2 – Клас ModuleManager

Таким чином забезпечується модульність архітектури, коли ядро рушія не має жорсткої залежності від користувацького коду, і дозволяє завантажити будь-який проєкт, не перекомпілюючи редактор.

Перед завантаженням модуля редактор виконує формування повного іменті бібліотеки залежно від конфігурації та платформи, наприклад, «ProjectName-Win64-Debug.dll». Після чого перевіряє, чи вже було завантажено цю бібліотеку, і лише за відсутності відповідного запису здійснює її підключення. У випадку успішного завантаження, модуль зберігається у внутрішньому словнику, що дозволяє відстежувати його стан і, при необхідності, виконати розвантаження. Окрім того, перед завантаженням вказується директорія, в якій слід шукати динамічну бібліотеку, чого потребує WinAPI.

Завдяки такому підходу рушій отримує гнучкість, яка дозволяє:

- Підключати різні модулі користувацького проєкту до одного і того ж процесу редактора;
- Оновлювати симуляційну логіку незалежно від рушія;

- Завантажувати та вивантажувати користувацький модуль динамічно, без необхідності перезапуску всього редактора
- Надає можливість розширення системи, для підтримки одразу декількох користувацьких модулів в одному проєкті.

Важливою частиною цієї системи є рефлексія –механізм, що дозволяє створювати екземпляри класів, описаних у динамічно підключених модулях. Саме завдяки цьому редактор здатен працювати з типами, визначеними в користувацькому проєкті, не знаючи їх під час компіляції. Цей аспект буде розглянуто у наступному підрозділі.

Таким чином система користувацьких проєктів у рушії не лише спрощує початкове налаштування і запуск симуляцій, а й формує гнучкий фундамент для розділення рушія та користувацьких застосунків, що є важливою характеристикою сучасних рушіїв. Подібна реалізація довела свою ефективність в таких системах, як Unreal Engine та Godot, і дозволяє масштабувати проєкти без зміни внутрішньої архітектури рушія.

3.3. Система рефлексії

Однією з фундаментальних технологій рушія, що забезпечує гнучкість, масштабованість та розширюваність рушія є система рефлексії. Вона дозволяє отримувати метаінформацію про типи, властивості, методи й інші елементи програми під час виконання, а також створювати об’єкти динамічно на основі їх імен або метаданих. Рефлексія лежить в основі багатьох підсистем рушія VeiM, таких як серіалізація, редагування властивостей у редакторі, створення об’єктів під час запуску симуляції, модульність, UI-прив’язка і збиральник сміття.

У контексті архітектури рушія система рефлексії є обов’язковим компонентом для розмежування ядра та симуляційної логіки. Завдяки їй рушій, не знаючи наперед про типи, визначені у модулі користувача, здатен:

- Створити екземпляр класу за іменем;
- Отримати список усіх властивостей класу для їх серіалізації, відображення у редакторі чи UI;
- Зчитати або змінити значення властивостей за іменем;
- Реєструвати класи в системі

Основою системи є спеціальний реєстр класів, реалізований як глобальна таблиця на рівні ядра рушія. Кожен тип, що має бути досступним через рефлексію, реєструється у цьому реєстрі. При цьому до кожного типу додається метаінформація. Це дозволяє рушію оперувати типами незалежно від конкретної реалізації класів.

Процес реєстрації класів автоматизований за рахунок макросів. Під час компіляції виконуються макроси реєстрації, і відповідна метаінформація додається до реєстру. Таким чином, на відміну від стандартного C++, який не підтримує рефлексію вбудовано, система досягається за рахунок метапрограмування.

Ще одним важливим аспектом є підтримка динамічного створення об’єктів. Завдяки збереженій фукнції-конструктору для кожного типу, рушій може створити будь-який об’єкт за його ім’ям. Цей механізм використовується для створення сутностей на сцені, компонентів, і інших об’єктів.

Оскільки рефлексія дозволяє отримати доступ до структури класів у рантаймі, вона також є критичною для серіалізації – процесу збереження та завантаження стану об’єктів. Це включає як збереження сцен, так і конфігураційних файлів або збереження стану симуляції.

У розробленій системі рушія рефлексія виступає як міст між ядром рушія та динамічно підключеним користувацьким модулем. Після завантаження модуля, усі типи, які потрібно відображати у редакторі чи створювати під час

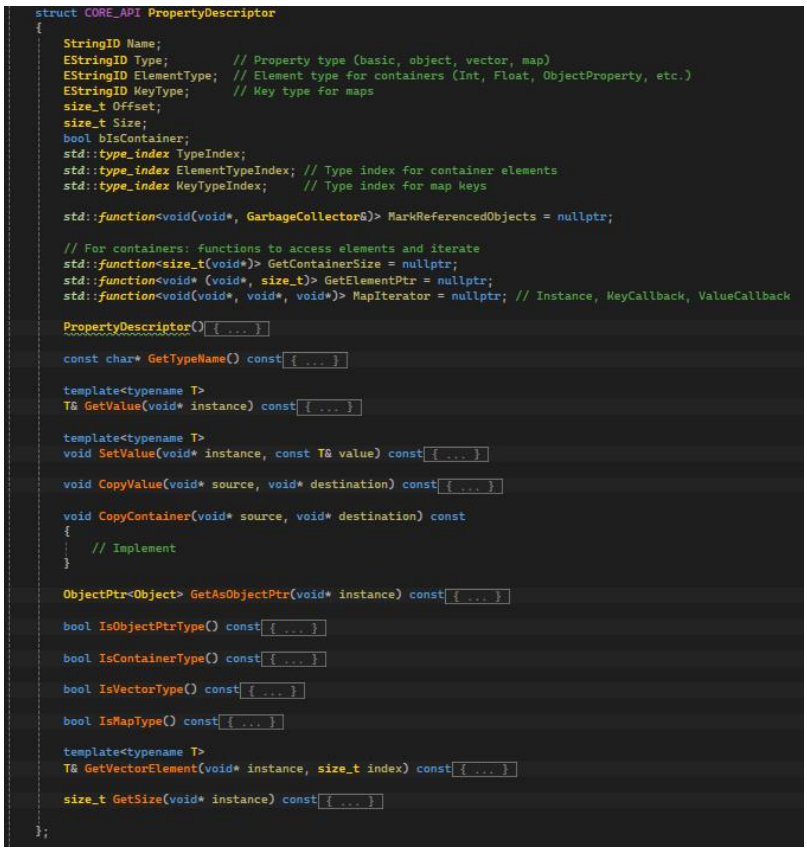
відтворення симуляції, реєструються у реєстрі, після чого доступні через підповідні API рушія.

Система рефлексії в рушії VeiM реалізована через три ключові компоненти: ClassRegistry, ClassDescriptor, PropertyDescriptor, які разом забезпечують динамічний доступ до метаданих класів і їх властивостей. Кожен клас що підтримує рефлексію, описується через окремий об'єкт метаінформації, відомий як дескриптор класу, в цьому випадку його роль виконує клас ClassDescriptor (рис. 3.3). Цей дескриптор містить усю необхідну інформацію для роботи з класом на рівні ядра рушія – зокрема, його ім'я, список властивостей та функцію створення екземпляра. Завдяки наявності цієї інформації рушій може інспектувати клас, серіалізувати його дані, відображати їх у UI, або створювати об'єкти динамічно без необхідності жорсткого зв'язування з конкретними типами на етапі компіляції.

Кожна властивість у класі описується дескприптором властивості, що реалізований класом PropertyDescriptor (рис. 3.4) і містить назву властивості, її тип у вигляді рядка, та позицію в пам'яті об'єкта. Це дає змогу отримати або змінити значення конкретного поля об'єкта за допомогою функцій GetValue, SetValue та CopyValue, маючи лише ім'я цієї властивості. Також доступні інші функції, для перевірки на приналежність властивості до певного типу, отримання розміру, імені типу.

```
class CORE_API ClassRegistry
{
    // Later add CDO
public:
    static void InitializeReflectionSystem() { ... }

private:
    static std::unordered_map<StringID, ClassDescriptor*>& GetRegistry() { ... }

    static std::vector<std::function<void()>>& GetRegistrationCallbacks() { ... }
public:
    template<typename T>
    static void AddRegistrationCallback() { ... }

    static void InitializeAllClasses() { ... }
public:
    static void RegisterClass(ClassDescriptor* classDesc) { ... }

    static ClassDescriptor* FindClass(StringID className) { ... }

    static std::unordered_map<StringID, ClassDescriptor*> GetAllClasses() { ... }
    template <typename T>
    static std::map<StringID, ClassDescriptor*, StringIDComparator> GetAllClassesOfClass() { ... }

    static std::map<StringID, ClassDescriptor*, StringIDComparator> GetAllClassesOfClass(ClassDescriptor* baseClass) { ... }
};
```

Рисунок 3.3 – клас ClassRegistry



```
struct CORE_API PropertyDescriptor
{
    StringID Name;
    EStringID Type;         // Property type (basic, object, vector, map)
    EStringID ElementType;  // Element type for containers (Int, Float, ObjectProperty, etc.)
    EStringID KeyType;      // Key type for maps
    size_t Offset;
    size_t Size;
    bool bIsContainer;
    std::type_index TypeIndex;
    std::type_index ElementTypeIndex; // Type index for container elements
    std::type_index KeyTypeIndex;     // Type index for map keys

    std::function<void(void*, GarbageCollector&)> MarkReferencedObjects = nullptr;

    // For containers: functions to access elements and iterate
    std::function<size_t(void*)> GetContainerSize = nullptr;
    std::function<void* (void*, size_t)> GetElementPtr = nullptr;
    std::function<void(void*, void*, void*)> MapIterator = nullptr; // Instance, KeyCallback, ValueCallback

    PropertyDescriptor() { ... }

    const char* GetTypeName() const { ... }

    template<typename T>
    T& GetValue(void* instance) const { ... }

    template<typename T>
    void SetValue(void* instance, const T& value) const { ... }

    void CopyValue(void* source, void* destination) const { ... }

    void CopyContainer(void* source, void* destination) const
    {
        // Implement
    }

    ObjectPtr<Object> GetAsObjectPtr(void* instance) const { ... }

    bool IsObjectPtrType() const { ... }

    bool IsContainerType() const { ... }

    bool IsVectorType() const { ... }

    bool IsMapType() const { ... }

    template<typename T>
    T& GetVectorElement(void* instance, size_t index) const { ... }

    size_t GetSize(void* instance) const { ... }
};
```

Рисунок 3.5 – клас PropertyDescriptor

Всі дескриптори класів реєструються в центральному реєстрі типів – глобальній структурі, доступній на рівні ядра рушія. Роль цього реєстру виконує клас ClassRegistry (рис. 3.5). Цей реєстр, що функціонує як словник типів, дозволяє здійснювати пошук класу за ім’ям та отримувати його дескриптор. Усі операції створення об’єктів за ім’ям, інспекції властивостей, серіалізації або UI-відображення відбуваються саме через взаємодію з цим реєстром.

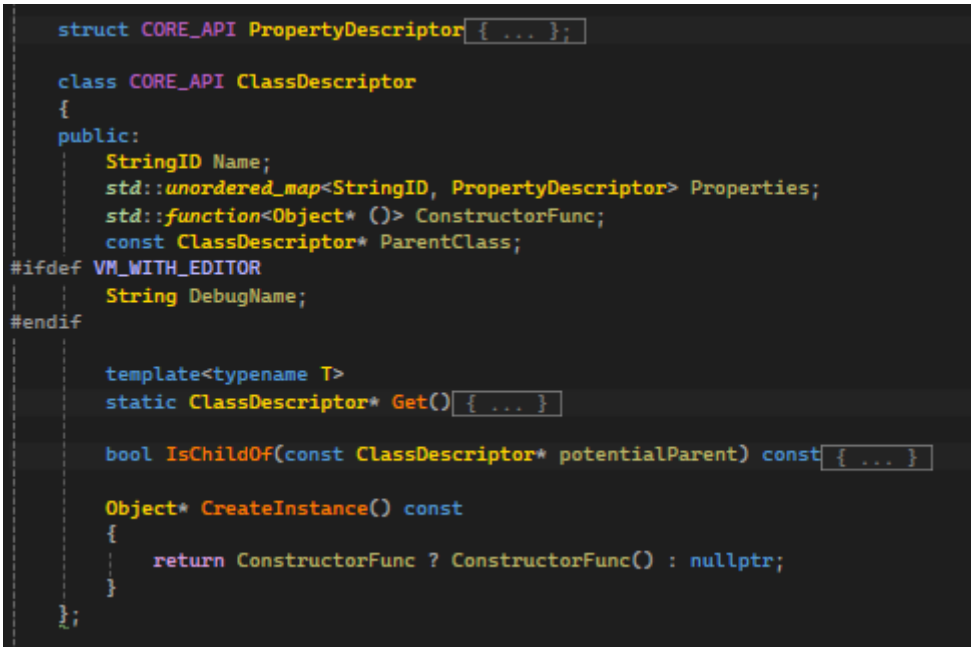


Рисунок 3.4 – клас ClassDescriptor

Процес реєстрації класів та властивостей реалізовано через набір макросів. Макрос DECLARE_CLASS (рис. 3.6) автоматизує оголошення необхідних методів для доступу до метаінформації всередині класу і оголошує метод RegisterProperties який необхідний для реєстрації властивостей оголошеногок класу. Крім цього оголошується статичний член класу s_ClassDescriptor що зберігає вказівник на дескриптор цього класу, який можна отримати за допомогою віртуальної функції GetClass, яка в свою чергу викликає статичною функцію StaticClass. Разом з цим макросом, для реєстрації класу необхідний макрос IMPLEMENT_CLASS. Він визначає метод StaticClass що повертає дескриптор цього класу, а також містить допоміжну функцію, що реалізує автоматичну реєстрацію класу під час статичної ініціалізації.

```
#define DECLARE_CLASS(Class, Parent) \
public: \
    using Super = Parent; \
    static VeiM::ClassDescriptor* StaticClass(); \
    virtual VeiM::ClassDescriptor* GetClass() const override { return Class::StaticClass();} \
private: \
    static VeiM::ClassDescriptor* s_ClassDescriptor; \
public: \
    static void RegisterProperties(VeiM::ClassDescriptor* classDesc);

#ifdef VM_WITH_EDITOR
    #define REFLECTION_DECLARE_DEBUG_NAME(DNAME) s_ClassDescriptor->DebugName = String(#DNAME);
#else
#define REFLECTION_DECLARE_DEBUG_NAME(DNAME)
#endif

#define IMPLEMENT_CLASS(Class) \
    VeiM::ClassDescriptor* Class::s_ClassDescriptor = nullptr; \
    namespace \
    { \
    bool _autoReg_##Class = []() \
        { \
        VeiM::ClassRegistry::AddRegistrationCallback<Class>(); \
        return true; \
        }(); \
    } \
    VeiM::ClassDescriptor* Class::StaticClass() \
    { \
        if (!s_ClassDescriptor) \
        { \
            s_ClassDescriptor = new VeiM::ClassDescriptor(); \
            s_ClassDescriptor->Name = VeiM::StringID(#Class); \
            REFLECTION_DECLARE_DEBUG_NAME(Class) \
            s_ClassDescriptor->ParentClass = Super::StaticClass(); \
            s_ClassDescriptor->ConstructorFunc = []() -> VeiM::Object* {return new Class(); }; \
            VeiM::ClassRegistry::RegisterClass(s_ClassDescriptor); \
            Class::RegisterProperties(s_ClassDescriptor); \
        VM_CORE_TRACE("[Reflection] Registered '{0}'", VeiM::String(#Class)); \
        } \
        return s_ClassDescriptor; \
    }
```

Рисунок 3.6 – Макороси DECLARE_CLASS і IMPLEMENT_CLASS

Макрос REGISTER_PROPERTY (рис. 3.7) застосовується для додавання опису властивостей до дескриптора класу, створюючи екземпляр класу PropertyDescriptor. Після цього цей об’єкт ініціалізується, і отримує лямбда функцію MarkReferencedObjects, що приймає участь в одній з стадій збиральника сміття, який буде розглянуто в наступному підрозділі. Інші макроси REGISTER_VECTOR_PROPERTY та REGISTER_MAP_PROPERTY мають ідентичне призначення, але розраховані на властивості типів vector та map.

Рисунок 3.7 – Макрос REGISTER_PROPERTY

При ініціалізації модуля або на початку виконання програми ці макроси забезпечують формування повної структури метаінформації для кожного типу що підтримує рефлексію.

Загалом, система рефлексії – це один із найскладніших, але найважливіших компонентів рушія, що забезпечує можливість створення інтерактивного середовища розробки, модульності та високого рівня автоматизації. Розроблена система рефлексії у VeiM забезпечує повну підтримку ключових задач рушія і є результатом ретельної інтеграції низькорівневих механізмів C++, шаблонів метапрограмування і макросів, що разом створюють потужну абстракцію. Її реалізація є основою для широкого спектра функціональності, і саме вона дає змогу реалізувати рушій не як моноліт, а як платформу для розробки гнучких, модульних симуляцій.

3.4. Збиральник сміття та ієрархія Object

Ієрархія Object у рушії VeiM грає ключову роль в системі сцен, що буде розглянута в одному з наступних підрозділів, але також застосовується поза її межами. Основна суть Object – відслідковування часу життя об’єктів цієї

ієрархії, і саме в керуванні часом життя об’єктів полягає роль збиральника сміття.

3.4.1. Збиральник сміття

Збиральник сміття є критично важливим компонентом будь-якого рушія, в тому числі VeiM. Це підсистема, що забезпечує автоматичне керування життєвим циклом об’єктів. Його основна мета – звільнення пам’яті від об’єктів, які більше не використовуються, без необхідності ручного втручання розробника. Це особливо важливо в довготривалих або динамічних симуляціях, де об’єкти постійно створюються та знищуються, а ризик витоку пам’яті зростає експоненційно. Така система вирішує цю проблему централізовано, безпечним і ефективним способом.

Система збираня сміття у рушії VeiM побудована за класичною моделлю «Mark-and-Sweep». Цей підхід поширений серед рушіїів, та зарекомендував свою ефективність. Він дозволяє легко керувати часом життя об’єктів уникаючи циклів, які притаманні системам що використовують розумні вказівники. І хоча розумні вказівники дозволяють розривати цикли за допомогою «weak» вказівників, такий підхід значно обмежує розробника у гнучкості системи що на них базується. В свою чергу «Mark-and-Sweep» алгоритм дозволяє уникнути циклів без будь-яких компромісів.

Алгоритм «Mark-and-Sweep» виконує 2 основні операції: фаза позначення і фаза очищення. Збиральник сміття що використовує такий алгоритм виділяє кілька основних етапів: реєстрація об’єктів, маркування досяжних об’єктів, очищення недосяжних, а також підтримки кореневих об’єктів та інтеграції з системою вказівників рушія. Усі об’єкти, що мають бути керованими, успадковують базовий тип Object, і автоматично реєструються в системі під час створення.

У серці системи знаходиться синглтон GarbageCollector (рис. 3.8), який надає глобальний інтерфейс для реєстрації об’єктів, за допомогою функції

RegisterObject, додавання об’єктів до коренів функцією AddRoot, ручного виклику збирання сміття функцією CollectGarbage та деяких інших функцій. Після створення об’єкт зберігається у множині m_Objects, яка представляє всі об’єкти, що знаходяться під контролем збиральника. Під час знищення об’єкт самостійно відписується з системи через UnregisterObject, що виключає ризик обробки неіснуючого об’єкта.

Окрема множина m_RootObjects містить кореневі об’єкти – тобто такі, які гарантовано мають бути живими і з яких починається пошук досяжних об’єктів. Ця концепція дає змогу побудувати дерево посилань і відстежити всі об’єкти, що напряму або опосередковано є такими, на які посилаються корені. Коренями зазвичай виступають такі об’єкти як сцени, активні сутності, або частини рушія, що постійно перебувають у пам’яті.

```
class CORE_API GarbageCollector
{
public:
    static GarbageCollector& Get();

    void CollectGarbage(bool bForce = false);

    void RegisterObject(Object* obj);
    void UnregisterObject(Object* obj); // Called by Object Destructor
    void AddRoot(Object* obj);
    void RemoveRoot(Object* obj);

    void MarkReachable(Object* obj);

    void SetCollectionThreshold(size_t min) { m_CollectionThreshold = min; }
    void SetAutomaticCollectionInterval(float seconds) { m_AutomaticCollectionIntervalSeconds = seconds; }

    uint32 GetObjectsCount() { return m_Objects.size(); }
    uint32 GetRootObjectsCount() { return m_RootObjects.size(); }
    uint32 GetLastCollectedCount() { return m_LastCollectedNonZeroCount; }
    uint32 GetStrongPtrRegisteredCount();
    uint32 GetWeakPtrRegisteredCount();

    std::unordered_set<Object*>& Debug_GetAllObjects();
private:
    void MarkPhase();
    void SweepPhase();
    void MarkRoots();
    void ProcessMark(Object* obj);
    void PrepareCollection();
    void FinalizeCollection();

    bool ShouldCollect() const;
public:
    template <typename T>
    void MarkReachableArray(const std::vector<T>& objects);

    template <typename T>
    void MarkReachableSet(const std::unordered_set<T>& objects);

private:
    GarbageCollector() = default;
    ~GarbageCollector();

private:
    std::unordered_set<Object*> m_Objects;
    std::unordered_set<Object*> m_RootObjects;
    std::vector<Object*> m_PendingKillObjects;

    size_t m_TotalObjectCount = 0;
    size_t m_LastCollectedCount = 0;
    size_t m_LastCollectedNonZeroCount = 0;
    size_t m_CollectionThreshold = 100;

    std::chrono::time_point<std::chrono::steady_clock> m_LastCollectionTime;
    float m_AutomaticCollectionIntervalSeconds = 30.0f;

    static GarbageCollector* s_Instance;
};
```

Рисунок 3.8 – Клас GarbageCollector

Збирання починається з етапу маркування, який реалізовано у функції MarkPhase, десистема обходить усі короневі об’єкти та позначає ті, що доступні, викликаючи ProcessMark для кожного. В свою чергу ця функція обходить всі властивості об’єкту, що зареєстровані системою рефлексії, і є вказівниками на інші об’єкти, після чого викликає зареєстрований делегат MarkReferencedObjects що був прив’язаний при реєстрації властивості. Функцію що прив’язується до цього делегату було розглянуто в розділі 3.3 у визначенні макроса REGISTER_PROPERTY. Ця функція викликає функцію збиральника сміття MarkReachable для властивості, де знову викликається ProcessMark, вже для нового об’єкту. Цей етап є рекурсивним і завдяки ньому зберігається повнаінформація про всі об’єкти, до яких існує прямий або непрямий доступ.

Після завершення маркування, система переходить до етапу очищення який реалізовує функція SweepPhase. На цьому етапі кожен об’єкт, що залишився в множині m_Objects, але не був помічений як досяжний, вважається недосяжним. Такі об’єкти поміщаються у m_PendingKillObjects, а потім знищуються централізовано у FinilizeCollection.

Збиральник підтримує автоматичне збирання з налаштованим інтервалом через m_AutomaticCollectionIntervalSeconds а також якщо кількість об’єктів перевищує поріг m_CollectionThreshold. Метод ShouldCollect виконує перевірку цих умов. Така автоматизація дозволяє запускати процес збирання без втручання розробника. Ця компібнація ручного і автоматичного керування робить систему адаптивною як для симуляцій з низькою інтенсивністю об’єктів, так і для високонавантажених симуляцій, де тисячі об’єктів створюються й зникають за секунду. Збирання сміття не було би повним без інтеграції з власною системою розумних вказівників, яка розглянута в пункті 3.4.2

Порівняно з ручним керуванням пам’яттю, система збиральника сміття реалізована в даному рушії має низку переваг:

- Висока безпека за рахунок відсутності втрачених вказівників і подвійного видалення.

- Гнучкість через автоматичне створення і знищення об’єктів.
- Простота використання
- Масштабованість

3.4.2. Система розумних вказівників

Рушій VeiM реалізує власну систему розумних вказівників, яка є важливою частиною для роботи збиральника сміття, відстеження життєвого циклу об’єктів та забезпечення безпечного керування пам’яттю у складних об’єктних структурах.

Система представлено двома основними шаблонними класами:

- ObjectPtr<T> – сильне посилання;
- WeakObjectPtr<T> – слабке посилання.

В обох випадках шаблонний параметр T повинен бути класом, що успадковує базовий тип Object.

Класична модель shared_ptr та weak_ptr не є достатньо ефективною для рушїв де:

- Життєвий цикл об’єктів може керуватися рушієм через відкладне знищення;
- Об’єкти мають складну ієрархію, властивості, які потрібно серіалізувати, або автоматично реєструвати в системах редактора, UI;
- Необхідна взаємодія з системою збирання сміття, яка виконує обхід графу об’єктів, та з системою рефлексії.

Саме тому у VeiM впроваджена спеціалізована система вказівників, яка дає змогу автоматично відстежувати всі активні посилання на об’єкти, дозволяє контролювати, які вказівники мають вплив на «життєздатність» об’єкта, і гнучко взаємодіє з класом GarbageCollector через реєстрацію вказівників у PointerRegistry.

ObjectPtr<T> – це основний тип вказівника, який гарантує коректне значення вказівника. Якщо об’єкт на який посилається ObjectPtr<T> буде видалено, внутрішній вказівник прийме значення nullptr. При кожному встановлені або копіюванні посилання, вказівник реєструється у PointerRegistry як сильний вказівник. При знищенні або зміні – автоматично відписується. Вказівники цього типу використовуються для властивостей, які утримують об’єкти «живими» у графі посилань

WeakObjectPtr<T> – це тип посилання, яке дозволяє взаємодіяти з об’єктом без участі у визначенні його життєвого циклу. Тобто, такий вказівник не зупиняє збирача сміття від зничення об’єкта. Єдина його відмінність від ObjectPtr<T> полягає в тому що властивість зареєстрована з WeakObjectPtr<T> не буде помічена як досяжна для збиральника сміття.

PointerRegistry – глобальний реєстратор вказівників, що асоціює конкретні екзепляри Object* із множинами вказівників на них. Цей клас використовується збирачем сміття для пошуку всіх посилань на об’єкт при виконанні обхідної фази. Завдяки цьому збиральник сміття може визначити, чи є живі сильні посилання на об’єкт, а також інвалідовувати посилання після знищення об’єкта.

3.4.3. Ієрархія Object

В сучасному рушії важливою є чітка ієрархія об’єктів, яка забезпечує узгоджене управління життєвим циклом, серіалізацію, відображення типів, роботу з ресурсами, збір сміття та компонентну архітектуру. У рушії VeiM роль такої базової одиниці виконує клас Object. Він є центральним елементом майже всіх систем. Ієрархія Object у VeiM служить фундаментом для всіх об’єктів, якими керує рушій.

Базовий клас Object реалізує фундаментальні сервіси для будь-якого рушієвого об’єкта:

- Ідентифікація і типізація через систему реєстрації типів. Цей клас має інтерфейс що дозволяє отримувати динамічний тип об’єкту функціями

GetClass і StaticClass також перевіряти динамічний тип об’єкту функцією IsA.

- Система імен, яка використовує StringID для швидкого доступу, збереження і порівняння імен. Цю систему розглянуто у підрозділі 3.8.
- Флагова система що забезпечує контроль життєвого стану об’єкта та його властивості.
- Підтримка ієрархії творення, яка забезпечує зв’язок між об’єктом і тим, хто його створив. Інформацію про творця об’єкту зберігає поле m_Creator.
- Сумісність з GC, реалізована через інтерфейс для збирача сміття. Частина цього інтерфейсу функція MarkReferencedObjects, яка позначає властивості об’єкту як досяжні для збиральника сміття.
- Компонетна модель, яка надає можливість створення дочірніх компонентів через функцію CreateComponent. Ця модель грає важливу роль у системі світу, що розглянута в підрозділі 3.5.
- Механізм клонування, реалізований функціями DuplicateObject і CopyProperties. Цей механізм є важливою частиною системи світу і дозволяє запускати симуляції прямо в редаткорі. Детальніше цей механізм розглянуто в підрозділі 3.5.

Ця модель дозволяє реалізувати як сутності сцени, так і системі елементи рушія з ждиною архітектурною логікою.

Однією з ключових особливостей є підтримка рантайм рефлексії через клас ClassDescriptor, який описано в підрозділі 3.3. Кожен об’єкт знає свій клас, і може отримати його використовуючи віртуальну функцію GetClass, і має доступ до статичного опису типу – StaticClass. Це дозволяє виконувати Runtime-Type-Information (RTTI) без використання стардартного typeid, що важливо для серіалізації і редакторських інструментів. Механізм IsA<T> дозволяє визначити, чи є об’єкт спадкоємцем певного класу, що особливо корисно при побудові динамічних систем на кшталт редакторів або геймплейних фреймворків.

Клас Object інтегрований із системою збирача сміття, через виклики MarkReferencedObjects, IsPendingKill, markPendingKill, StartDestroy, FInishDestroy. Це дозволяє реалізувати контрольоване знищення об’єктів, коли GC визнає, що вони більше не використовуються. Флаги, як FLAG_PENDING_KILL, FLAG_GARBAGE, FLAG_ROOT, вказують на стан об’єкта та визначають, чи повинен об’єкт зберігатися чи бути очищеним.

Важливою частиною є можливість маркування об’єкта як кореневого функцією MarkAsRoot та UnmarkAsRoot – такі об’єкти не можуть бути автоматично знищені, допоки не буде знято флаг FLAG_ROOT. Це корисно для глобальних менеджерів чи життєво важливих систем рушія.

Метод CreateComponent дозволяє об’єктам створювати дочірні об’єкти з автоматичним встановленням m_Creator. Це дозволяє компонуванню об’єктів, де наприклад об’єкт класу Entity, що є основним агентом сцени, створює різноманітні компоненти, що додають йому функціональність. Така архітектура – основа ECS-підходу або його гібридного варіанту.

Завдяки статичній функції NewObject<T> створення об’єктів стало централізованим і безпечним. Функція використовує ClassDescriptor, встановлює ім’я, творця, флаги й автоматично реєструє об’єкт. Це дозволяє легко створювати нові об’єкти на льоту без жорсткого зв’язування типів. CastObject<T> дозволяє безпечно перетворити об’єкт до іншого типу за допомогою системи типів рушія.

Щоб керувати життєвим циклом об’єктів у рушії з урахуванням їхнього походження, у VeiM реалізована спеціальна система – ObjectCreatorSystem. Вона виконує облік об’єктів за їхніми творцями (creator), що є важливим для управління власністю і внутрішніх систем рушія. Така система дозволяє знаходити всі об’єкти, створенні конкретним об’єктом, рекурсивно обходити ієрархії створення, знищувати вкладені об’єкти разом із їхнім творцем і ізолювати контексти. Система ObjectCreatorSystem тісно інтегрована з класом Object, який надає доступ до неї через функцію GetCreatorSystem.

Ієрархія Object слугує основою для багатьох внутрішніх систем рушія та забезпечує гнучку архітектуру придатну до масштабованості.

3.5. Система світу

Ієрархія Object дозволяє створити систему світу, що реалізує концепцію сцен. Ключовою частиною організації сцен є система компонентів.

3.5.1. Система компонентів

Сучасні рушії використовують різні підходи до організації об’єктів сцени. Одним із найбільш популярних є Entity-Component-System (ECS), однак у рушії VeiM реалізовано гібридний підхід, який поєднує гнучкість компонентної системи з об’єктно-орієнтованим підхододм.

ECS – це шаблон, який розділяє дані, логіку та об’єкти на незалежні частини, що дозволяє досягнти високої продуктивності, масштабованості та модульності. ECS був особливо популяризований в застосунках, де важливо обробляти велику кількість сутностей.

У класичному ECS використовується підхід орієнтований на дані, замість широко поширеного об’єктно-орієнтованого. Вся його логіка організована навколо трьох основних понять:

1) Entity

   Унікальний ідентифікатор, найчастіше просто число, що не містить даних і не має логіки.

2) Component

   Проста структура, що зберігає лише дані.

3) System

   Окрема частина коду, яка виконує логіку. Система отримує всі компоненти певного типу, обробляє їх і вносить зміни в дані.

Ключова ідея ECS – максимально розділити дані і поведінку. Це дозволяє системам працювати паралельно, кеш ефективно використовувати дані, і зменшити зв’язність між частинами коду. ECS чудово підходить для ситуацій, коли кількість об’єктів у симуляції велика, а логіка повинна бути максимально оптимізованою.

Проте разом із перевагами ECS має й свої недоліки: складність відлагдження. Низька гнучкість у створенні ієрархій об’єктів, ускладнене використання в редакторах, а також підвищена вартість розробки через втрату об’єктно-орієнтованої структури.

У рушії Veim реалізована гібридна система, що поєднує ідеї ECS з більш зручною для high-level логіки структурою. Основна ідея цього підходу: зберегти гнучкість ECS, але при цьому мати об’єкти, які можна безпосередньо описати в коді або відредагувати в інструментах редактора. Замість простого ідентифікатора, у VeiM використовується клас Entity, що є повноцінним об’єктом із життєвим циклом, методами і здатністю містити компоненти.

Опис основних складових гібридного ECS у рушії VeiM:

- Entity

  Основний об’єкт сцени, має ідентифіктор, трансформацію, методи оновлення, список компонентів, і може створювати компоненти. Це основна точка входу для логіки об’єктів у симуляції.

- Component

  Це логічний компонент, що додається до Entity. Компоненти можуть мати власну поведінку, лоігку оновлення, посилання на ресурси, властивості для редактора тощо.

- SceneComponent

  Спеціалізований підклас Component, який має просторову трансформацію і може бути частиною ієрархії сутності. Такі компоненти дозволяють будувати вкладені струкутри з візуальним відображенням.

В таблиці 3.1 наведені відмінності між класичним ECS та гібридним підходом що використовується у VeiM.

Таблиця 3.1

Порівняльна таблиця видів ECS

Характеристика	Класичний ECS	VeiM ECS
Entity	Простий ідентифікатор	Повноцінний об’єкт сцени
Component	Тільки дані	Дані і логіка
System	Централізовано обробляє дані	Поведінка розміщена у компонентах
Ієрархія сцени	Як правило, відсутня	Є через SceneComponent
Продуктивність	Максимізована	Збалансована зручність і ефективність
Підтримка редактора	Складно реалізувати	Добре інтегрується

1.5.2. Основні класи ієрархії Object

У сучасному рушії об’єкти ієрархії Object і їх життєвий цикл є основою всієї системи світу. Світ – це контекст, у якому існують усі елементи симуляції, від сутностей і компонентів до сцен і логічних систем. Основна мета системи світу – впорядкувати й організувати об’єкти так, щоб їх поведінка, рендеринг, вхідні дані та взаємодія могли відбуватися узгоджено і масштабовано.

В архітектурі рушія ця система побудована навкого фундаментальної ієрархії об’єктів. Базовий клас Object, як було розглянуто в попередньому підрозділі, слугує фундаментом для об’єктів рушія. Над ним створені абстрації вищого рівня: Component, Entity, World, Level тощо. Кожен із цих класів відповідає за окремий аспект логіки симуляції – від компонування властивостей до взаємодії з світом.

Клас Component

Клас Component наслідує базовий клас Object, що забезпечує механізми відкладеного знищення, створення та інші системні функції. Клас має типову ієрархію станів:

1) Ініціалізація

   Реалізований методом Initialize і викликається при першому налаштуванні компонента.

2) Початок симуляції

   Реалізований методом BeginPlay і активується під час запуску логіки симуляції, дозволяючи компоненту вступити в дію.

3) Оновлення

   Реалізований методом TickComponent і викликається кожного кадру, якщо компонент увімкнено для оновлення.

4) Завершення симуляції

   Реалізований методом EndPlay і викликається при завершенні симуляції або видаленні компонента.

5) Знищення

   Реалізований методом Destroy та StartDestroy і керує життєвим циклом та очищенням.

Цикл оновлення компоннета керується через структуру ComponentTickFunction, яка задає частоту оновлення, групу оновлення та прапорці властивостей функції. Компонент можна динамічно активувати або деактивувати через методи SetActive, ToggleActive. Далі розглянуто систему реєстрації функцій оновлення більш детально.

Компонент може бути зареєстрований у світі за допомогою функції RegisterInWorld опосередковано через власника Entity. Реєстрація компонентів напряму без власника не дозволяється. У процесі реєстрації викликаються методи OnRegister та RegisterTickFunction.

Компонент має асоційованого власника Entity, що зберігається у полі m_OwnerCached. Це дозволяє швидко отримати доступ до об’єкта сцени, до якого прикріплено компонент. Власник задається через SetOwner, а отримується через GetOwner. Також передбачено перевірку тегів компонента через HasTag, що дозволяє легко фільтрувати компоненти за ознаками.

Клас Component є абстрактною основою, яку розширюють інші класи. Ці класи розглянуто далі.

Клас Entity

У рушії VeiM основним базовим об’єктом, який представляє фізичну або логічну сутність у світі, є склас Entity. Його роль у системі – бути агентом сцени, до якого можна прикріплювати компоненти, трансформувати у просторі, обробляти логіку оновлення, взаємодіяти з системами вводу, контролерів, та управляти життєвим циклом об’єкта на сцені.

Клас Entity належить ієрархії Object, завдяки чому автоматично отримує базові можливості рушія. Основний цикл життя Entity включає:

1) Ініціалізація і післястворювальні методи: OnSpawnInitialize, PostConstruction.
2) Компонентні стадії: PreInitializeComponents, InitializeComponents, RegisterAllComponents, UnregisterAllComponents, UninitializeComponents.
3) Симуляційний цикл: BeginPlay, Tick, EndPlay, Destroy.

BeginPlay викликається після того, як сутність повністю створена, компоненти ініціалізовані, зареєстровані у світі й вона готова до участі в симуляції. Метод TickInWorld викликається зсередини світу й обробляє щокадрову логіку, включаючи оновлення компонентів і делегування віртуальному Tick, якщо ввімкнено оновлення.

Сутність підтримує повноцінну трансформацію у тривимірному просторі через RootComponent, який є похідним від SceneComponent. Клас

SceneComponent розглянуто у наступному підпункті. Всі функції трансформації делегуються RootComponent і включають:

- Абсолютні зміни: SetTransform, SetLocation, SetRotation, SetScale;
- Інкрементальні: AddWorldLocation, AddWorldRotation, AddLocalTransform;
- Відносні: SetRelativeLocation, SetRelativeRotation, SetRelativeScale.

Кожен об’єкт класу Entity володіє власною множиною компонентів – це об’єкти класу Component, які можна додавати як під час ініціалізації, створюючи їх за допомогою функції CreateComponent класу Object, що вже раніше була згадана, так і під час виконання через AddComponent. Відповідно, рушій підтримує нативні компоненти, і компоненти створені під час виконання. Вказівники на всі компоненти зберігаються в полі m_OwnedComponents. Вказівники на рантайм компоненти додатково розташовані в m_RuntimeComponents.

Сутність також визначає RootComponent, який виконує роль кореня ієрархії трансформацій. Всі інші SceneComponent-компоненти можуть бути прикріплені до нього або до інших підкомпонентів. Для зручності надано методи приєднання AttachToComponent, AttachToEntity, а також DetachFromEntity і DetachAllSceneComponents.

Типізація компонентів реалізована за допомогою шаблонних методів GetComponentByClass, що дозволяє безпечно отримати компоннет за типом, використовуючи StaticClass. Також реалізовано отримання всіх компонентів певного типу через GetComponents.

Клас Entity має точки входу для взаємодії з іншими системами рушія:

- Підтримка вводу, через динамічне створення InputComponent-компоненту. Доступні функції EnabgleInput і DisableInput для управління станом прийому вводу.
- Контекст, в якому сутність знаходиться. До контекту відносяться зв’язані об’єкти класів World, Level, GlobalgameState.

Так само як і компоненти, сутність має підтримку тегів, які дозволяють ідентифікувати об’єкти без класових перевірок.

Клас SceneComponent

Клас SceneComponent виконує ключову роль у сценовій ієрархії, трансформація і прикріпленні компонентів. Його головне завдання – зберігати позицію, обертання і масштаб відносно батьківського компоненту або світу, та забезпечувати синхронізацію цих даних у ієрархії сцени.

SceneComponent забезпечує транфсформацію в локальних і світових координатах, а також дозволяє організовувати ієрархічні зв’язки між компонентами через механізм прикріплення. Клас збергіає локальні трансформаційні дані: m_RelativeLocation, m_RelativeRotation, m_RelativeRotationQuat, m_RelativeScale, а також кешовану світову трансформацію в m_Transform. Бітові прапори з префіксом Absolute взначають, чи трансформації є абсолютними, тобто незалежними від батьківських. Також він зберігає вказівник на батьківський компонент m_Parent і масив вказівників на дочірні компоненти m_AttachChildren.

Механізм SetupAttachment, AttachToComponent, DetachFromComponent дає змогу будувати сценову ієрархію. Метод CalcWorldTransform враховує батьківський компонент, щоб обчислити глобальну трансформацію, а UpdateWorldTransform синхронізує її.

Трансформації можна змінювати як у локальній, так і в світовій системі координат. Методи SetRelativeLocation, AddLocalRotation, SetWorldTransform дозволяють повністю керувати положенням компонента. Функції GetForwardVector, GetRightVecetor, GetUpVector дозволяють отримати напрямки компонента, які базуються на поточному обертанніє

Клас SceneComponent формує основу просторової організації об’єктів у рушії. Він забезпечує контроль над трансформацією, прикріпленням, візуальною складовою і життєвим циклом компонентів у сцені.

Клас Level

Клас Level є представляє одну сцену та забезпечує повноцінне управління її вмістом. Він виступає контейнером для сутностей, а також точною, з якої відбувається контроль за їх ініціалізацією, активацією, обробкою вводу та копіюванням. На відміну від класу World, який відповідає за глобальний контекст сцени, Level фокусується виключно на локальному управлінні групою сутностей.

Головне призначення класу – це організація життєвого циклу сутностей, які належать до рівня. За це відповідають методу InitializeEntities, PreInitializeComponents, InitializeComponents, та за умови що світ уже перейшов у режим симуляції – ProcessBeginPlay. Крім того, Level гарантує що всі компоненти сутностей будуть зареєстровані або скинуті через методи UpdateAllComponents та ClearAllComponents.

Окреме місце займає система обробки вводу. У випадках, коли контролери ще не готові або не присутні під час створення певних сутностей, вони можуть бути тимчасово додані до внутрішнього списку, що зберігається в полі m_EntityInputList. Після того, як контролер з’являється у світі, метод ProcessNewInputInitWithInputList звіряє його індекс у списку контролерів світу та, у разі збігу, виконує прив’язку до відповідних сутностей. Роботу контролера розглянуто у наступних пунктах. Залежно від типу об’єкта, він може бути взятий під контроль або просто отримати досутп до вводу. Такий підіхд забезпечує гнучке і динамічне управління інтерактивною логікою у багатокористувацькому середовищі.

Метод Duplicate дозволяє створити повну копію рівня разом з усіма його стуностями, що важливо для інструментів редактора і динамічного створення копій рівнів.

Клас World

Клас World – це головний клас, який представляє симуляційний світ і виконує роль центрального контейнера для всіх сутностей, контролерів, служб

та пов’язаних підсистем. У рушії може існувати лише один активний екземпляр World для симуляції, і він зберігається в глобальному вказівнику g_World. Кожен світ має свій тип EWorldType який визначає його призначення.

Світ керує життєвим циклом сцени – від ініціалізації до знищення. Він відповідає за створення та знищення сутностей Entity, керування оновленням через Tick усіх об’єктів, а також оновлення та рендеринг компонентів. Крім того, World зберігає поточний рівень в CurrentLevel, і контролери клієнтів. Клас підтримує запуск симуляції методом BeginPlay, виконує делеговане оновлення служб і сутностей, а також дозволяє отримати доступ до глобального стану через GlobalGameState.

Клас GlobalGameState

Це об’єкт, що представляє глобальний стан сесії застосунку симуляції і зберігається протягом усього життєвого циклу програми. Клас ініціалізується як для звичайного запуску через InitForGame, так і для редакторського режиму InitForEditorPlay. Він створює новий світ через статичну функцію класу World – CreateWorld та зберігає посилання на нього.

Головне призначення GlobalGameState – керування клієнтами або гравцями, що реалізовані класом Player, та їх взаємодією зі світом. Він створює нових гравців і пов’язує їх із контролерами, що автоматично з’являються у світі. Окрім цього, класмає доступ до класу рушія Engine і може створювати об’єкти з його контексту.

Клас Agent

Клас Agent – це клас що успадковує клас Entity, і є базовим для всіх об’єктів, які можуть бути керовані гравцем у світі рушія. Агент представляє фізичну сутність у симуляції, яка здатна реагувати на введення і бути взятий під контроль класом Controller.

Агент має можливість приймати події введення, відслідковувати поточного контролера, створювати і реєструвати InputComponent, і братись під контроль. Перевизначивши віртуальну функцію SetupInputComponent,

користувач рушія має прив’язати делегати обробки введення до дій, осей, або конопок. Цей механізм реалізовано у класі InputComponent.

Клас InputComponent

Важливою складовою інтерактивності у програмних агентно-орієнтованих симуляціях є механізм обробки вводу від користувача. У запропонованій архітектурі рушя введення для агентів реалізується через спеціальний компонент InputComponent, що може бути прикріплений до будь-якого об’єкта-сутності. Його основне завдання полягає в надані сутності можливості підписуватися на події вводу та реалізовувати відповідні реакції, ґрунтуючись на конфігурації прив’язок до клавіш, дій та осей.

Компонент підтримує читири основні типи при’язок:

1) InputActionBinding;
2) InputAxisBinding;
3) InputKeyBinding;
4) InputAxisKeyBinding.

Реєстрація нових прив’язок здійснюється за допомогою шаблонних методів BindAction, BindAxis, BindKey та BindAxisKey.

Проєктування InputComponent дозволяє інкапсулювати ввід на рівні сутностей, не покладаючись на глобальні робробники, що відповідає сучасним принципам архітектури рушіїв.

Класи Player і Controller

Класи Player та Controller реалізують частину взаємодії між користувачем і симуляційним світом. Ці два класи працюють разом для представлення локального гравця, обробки введення, контролю об’єктів сцени, а також управління життєвим циклом.

Клас Playerє є представником конкретного користувача або гравця у системі рушія. Його основна задача – зберігати посилання на об’єкт Controller, який виконує фактичне керування агентом і обробку вводу.

Клас Controller є більш комплексним і забезпечує зв’язок між гравцем і контрольованим агентом, обробку вводу через систему InputManager, підтримку ієрархії компонентів вводу з пріоритетами, упрвління станом агента, якого контролю і взаємодію з об’єктами світу. Контролер успадковує Entity, отже може бути розміщений у світі і брати участь в оновленнях через Tick.

Основні функції Controller:

- Possess і Unposses – беруть агента під контроль або звільняють
- PushInputComponent і PopInputComponent – динамічне управління стеком компонентів вводу.
- InitInput – ініціалізація систем вводу, до якої входить створення InputManager, та передача контролера до рівня для додаткових налаштувань.
- InputKey – передача події вводу до системи.
- ProcessInput – побудова і обробка стеку InputComponent, у якому враховується ввід від агента, самого контролера та всіх додаткових компонентів.

1.5.2. Система реєстрації функцій оновлення

У рушії VeiM система оновлення об’єктів реалізована через абстрактну структуру TickFunction, яка є фундаментальним механізмом для визначення, коли і як має виконуватись логіка оновлення об’єктів під час симуляційного циклу.

Кожен об’єкт, що потребує регулярного оновлення, може реалізувати одну з похідних структур – EntityTickFunction або ComponentTickFunction, залежно від того, чи це сутність, чи компонент. Ці структури зберігають вказівник на відповідний об’єкт TargetObject і реалізують метод RunTick, в якому викликається оновлення логіки конкретного об’єкта.

У базовій структурі TickFunction є можливість налаштовувати частоту оновлення TickPeriod, увімкнення або вимкнення оновлення через SetEnable,

визначення групи оновлення TickGroup, а також керування тим, чи потрібно оновлювати об’єкт у паузі btickInPause. Метод Register при’язує функцію оновлення до рівня Level, після чого вона буде викликатись автоматично відповідно до графіку.

Прапорці bTickable та bTickOnStart визначають, чи має об’єкт взагалі брати участь в оновленні і чи повинен він починати оновлення одразу після створення. Внутрішній стан m_TickState дозволяє відстежувати, чи оновлення увімкнено, вимкнено або поставлно на паузу. Такий підхід дозволяє уникнути зайвих викликив функцій, відключивши оновлення, якщо воно не використовується.

Ця система забезпечує чітке розділення відповідальностей між реєстрацією, управлінням і виконанням оновлень, дозволяючи масштабувати проєкт і централізовано контролювати оновлення всіх активних об’єктів у симуляції.

1.5.3. Класи Engine

Для забезпечення функціонування системи світу, важливим компонентом архітектури рушія є центральна ієрархія класів, що реалізує основні механізми ініціалізації, симуляційного циклу та інтеграції редактора. У VeiM це реалізовано через базовий клас Engine та його нащадків GameEngine для самостійного застосунку та EditorEngine для редакторського.

Базовий клас Engine є точкою входу для запуску як симуляційного так і редакторського середовища. Він успадковує клас Object, підтримує систему рефелксії і є кореневим об’єктом для життєвого циклу рушія.

Основні методи інтерфейсу цього класу:

- Init – ініціалізує рушій, позначаючи себе як кореневий об’єкт;
- Start – точка запуску систем, яка може бути перевизначена нащадками;
- Tick – викликається кожен кадр, призначений для оновлення рушія;
- NotifyGCRunThisFrame – допоміжна логіка для інтеграції зі збиральником сміття.

Також Engine містить об’єкт GameViewportClient, який є інтерфейсом між внутрішньою логікою симуляції і візуалізацією.

Клас GameEngine розширює Engine і реалізує логіку, пов’язану із стандартним самостійним симуляційним запуском проєкту, тобто без редактора. Він призначений для створення глобального стану та гравців. Під час ініціалізації відбувається створення об’єкта GlobalGameState, та викликається його метод InitForGame, після цчого створюється перший гравець за допомогою CreatePlayer.

Клас EditorEngine розширює Engine та забезпечує роботу середовища редактора. Його основна мета – керування двома незалежними світами: редакторським в EditorWorld та симуляційним PlayWorld, які використовуються в режимах редагування та попередього перегляду відповідно.

Серед основних можливостей цього класу – запуск сесії симуляції в редакторі. Метод RequestPlaySession фіксує запит на початок симуляції. Метод StartPlaySession створює дубльовану копію редакторського світу через CreatePlayWorldDuplication, ініціалізує новий GlobalGameState, створює GameViewportClient, гравця, контролер, і передає керування у створений світ. Розділення світів на редакторський і симуляційний, та копіювання редакторського світу для ініціалізації симуляції дозволяє розділити сцену що редагується від сцени що симулюється, через що зміни зроблені під час симуляції не будуть присутні в редагованій сцені. Дублювання реалізується через функцію GetDuplicateForEditorPlay, яка вже згадувалась раніше. Механізм дублювання працює за рахунок визначення логіки копіювання самим об’єктами. Для цього клас Object має функцію DuplicateObject яка виконує базове копіювання в новий об’єкт, яке не залежить від специфіки об’єкту, після чого викликається віртуальний метод Duplicate, який за потреби перевизначається дочірніми класами, та реалізує логіку дублювання свого стану що залежить від специфіки об’єкту. Після чого метод DuplicateObject повертає продубльований об’єкт. Механізм дублювання є рекурсивним оскільки всередину функції Duplicate може

міститись виклик DuplicateObject для властивості поточного об’єкту. Наверху рекурсії знаходиться дублювання об’єкту World. Після копіювання об’єкту World, новий світ ініціалізується.

Для завершення сесії симуляції використовується метод EndPlaySession. Він виконує повне очищення симуляційного світу, припиняє його роботу і повертає активність у редакторський світ, після чого викликає збиральник сміття для свільнення ресурсів завчасно.

Для створення редакторського світу використовується метод NewMap, який створює новий World типу Editor, встановлює його як поточний і виконує початкову ініціалізацію і реєстрацію компонентів.

Метод Tick проводить оновлення симуляційногосвіту і контролює запити на запуск і завершення сесії, а також викликає очищення при потребі.

Присутній метод CloseEditor, що завершує роботу програми і забезпечує коректне завершенн всіх сесій та збереження.

Даний клас реалізує повний цикл життя редакторської сесії, включно з дублюванням світу, його ізольованим виконанням та подальшим очищенням.

1.6. Forward+ рендеринг

Рендеринг у тривимірних застосунках – це процес побудови зображення сцени на основі геоментрії, матеріалів, освітлення та положення камери. Вибір рендерингового підходу безпосередньо впливає на продуктивність, візуальну якість і гнучкість системи. Протягом років у графічних руіях сформувалися два основні методи рендерингу: Forward та Deferred. Кожен із них має свої сильні й слабкі сторони, і тому в залежності від типу сцени, скількості світлових джерел або необхідних візуальних ефектів рушії можуть використовувати різні стратегії.

Forward рендеринг є традиційним і простим підходом, при якому об’єкти сцени одразу освітлюються та відображаються у фінальний кадр. Однак при збільшенні кількості джерел світла цей метод стає неефективним. Deferred

рендеринг, навпаки, відкладає обчислення освітлення на пізніший етап, що дозволяє ефективніше обробляти багато джерел світла, але вимагає більше пам’яті та ускладнює підтримку деяких ефектів. У відповідь на ці обмеження з’явились гібридні підходи, зокрема Forward+, який поєднує простоту Forward рендерингу з масштабованістю Deferred. У цьому підрозділі розглянуто суть підходу Forward+, його архітектуру та переваги.

Для розуміння архітектури рендереру потрібно проаналізувати теоретичний підхід, який використовується цією парадигмою. Оскільки Forward+ рендеринг є покращеною версією Forward рендерингу, спочатку необхідно розібрати його. Суть Forward рендерингу полягає у виконанні всього пайплайну від початку до кінця за один підхід, один раз для кожної пари тривимірна сітка(меш) – світло. Для кожного об’єкту накопичується ефект від кожного джерела світла в один фреймбафер за декілька викликів малювання. Виклик малювання – початок виконання пайплайну, про яке було згадано раніше. Такий вид Forward рендерингу називається багатопрохідним, оскільки виконується декілька проходів для кожного мешу. Подібний підхід є занадто примітивним і ресурсозатратним, оскільки потрібно обраховувати вплив кожного джерела світла на кожен об’єкт, навіть якщо цей ефект буде нульовим.

Його перевагою є можливість простого рендерингу напівпрозорих об’єктів, для якого спочатку малюються непрозорі об’єкти, після чого малюються прозорі. Також він дуже ефективний для сцен з маленькою кількістю джерел світла, і дозволяє використовувати моделі з різними типами матеріалів, тобто такими що мають різні функції освітлення. Єдиним недоліком є значне зниження продуктивності при додаванні додаткових джерел світла.

Для оптимізації продуктивності Forward рендерингу можна використати техніки що зменшують кількість викликів малювання. Для цього змоделювати світло, спад яскравості якого не нескінченний, а все ж досягає нуля при певному радіусі. Це надає можливість обмежити межі впливу вітла, і замальовувати лише ті пікселі, які потрапляють в цю зону впливу світла.

Для сучасних апаратних засобів виклики малювання стали відносно дешевими з точки зору продуктивності, завдяки чому Forward рендеринг здатен ефективно обробляти велику кількість геометрії без суттєвого впливу на продуктивність. Натомість Deferred рендеринг споживає значний обсяг відеопам’яті через використання внутрішніх буферів та створює додаткове навантаження на шину PCIe під час передавання даних.

Forward+ рендеринг долає основний недолік класичного Forward підходу – обмежену масштабованість щодо кількості джерел світла. Це досягається шляхом розділення процесу на два етапи: кластеризації екранного простору та відбірки джерел світла, що впливають на кожен кластер.

Сцена ділиться на тривимірну сітку кластерів у просторі екрана і для кожного кластера визначається, які джерела світла на нього впливають. Цей етап виконується на обчислювальному шейдері до початку рендерингу геометрії. У результаті формується спеціальна структура даних – табилця освітлення для кластерів, яка дозволяє уникнути непотрібних обчислень під час рендерингу.

Після побудови цієї таблиці, геометрія рендериться звичайним Forward методом, але тепер фрагментний шейдер використовує дані з таблиці кластерів, щоб враховувати лише потрібні джерела світла для кожного пікселя. Це дає змогу ефективно обробляти десятки або навіть сотні джерел світла, зберігаючи переваги Forward рендерингу – зокрема підтримку прозорості, просту роботу з різними матеріалами та відсутність потреби у великих G-буферах, та споживати менше відеопам’яті.

Forward+ рендеринг інтегрується в архітектуру рушія як основна стратегія обробки освітлення. У структурі рушія передбачено виділення окремого етапу кластеризації екранного простору, що виконується за допомогою обчислювального шейдера. Кластеризація реалізується на основі поділу простору в проєкції на екран на сітку по координатах X, Y та Z, що забезпечує можливість визначення локального впливу джерел світла для кожного фрагмента сцени.

Для зберігання результатів обчислення використовуються структуровані буфери, доступні в шейдерах, які містять індекси джерел світла для кожного кластера. У фрагментному шейдері, який відповідає за остаточне обчислення освітлення, здійснюється вибірка лише тих джерел світла, що стосуються відповідного кластера.

1.7. Система подій введення

Система обробки вхідних подій є важливою частиною будь-якої інтерактивної програми, особливо у інтерактивних агентно-орієнтованих рушіях реального часу та графічних застосунках. Вона відповідає за перехоплення дій користувача та їх перетворення на події, які можуть бути оброблені логікою програми. У даному підрозділі розглядається реалізації такої системи. Деякі частини цієї системи були розглянуті раніше в класах Controller та InpuptComponent.

Система вхідних подій побудована на основі механізму зворотних викликів, які реєструються через GLFW. Кожен тип події має власну функцію зворотнього виклику, яка викликається при відповідній дії користувача. Ці зворотні виклики створюють об’єкти подій, які потім передаються до централізованого обробника подій застосунка рушія.

Початковим елементом системи є клас Window, який відповідає за управління головним вікном та реєстрацію зворотніх викликів. У методі SetEventCallbacks реєструються всі необхідні обробники для різних типів подій. Кожен зворотній виклик отримує відповідні параметри, наприклад, код кнопки або координати курсору і створює приватний об’єкт події, який передається далі через мезанізм делегування.

Після створення об’єкта події він передається до делегату EventCallback. Цьому делегату присвоєну функцію OnEvent класу Application, що є центральним класом застосунку рушія. Цей метод відповідає за додавання події

до черги вхідних подій застосунку. Черга реалізована у класі Application через контейнер vector<InputEventInternal*>, що дозволяє накопичувати події між кадрами та обробляти їх пізніше.

Обробка подій відбувається в методі ProcessInputEvents, який викликається у головному циклі програми. Цей метод ітерує через всі накопичені події, викликає для кожної метод ProcessInputEvent та звільняє пам’ять. Такий підхід гарантує, що події обробляються послідовно і немає втрати даних через перезапис у реальному часі.

Кожна подія має свій тип, який визначається переліком EInputEventType. У методі ProcessInputEvent відбувається розгалуження за типом події та її приведення до відповідного класу. Після цього викликаються спеціалізовані методи, такі як OnKeyDown, OnMouseMove або OnSizeChanged, які трансформують сирі дані події у більш зручний формат і передають їх до системи вводу.

Для клавіатурних подій додатково відстежується стан модифікаторів через клас ModifierState. Це дозволяє визначати, чи була натиснута клавіша разом із модифікатором, що важливо для гарячих клавіш та інших комбінацій. Події миші обробляються з урахуванням попереднього положення курсору, що дозволяє розраховувати зміщення для плавного управління.

Однією з ключових особливостей системи є механізм прив’язки дій до конкретних клавіш або їх комбінацій через клас InputManager. Він дозволяє зв’язувати абстрактні дії, наприклад, «переміщення вперед» з фізичними клавішами за допомогою InputActionMapping та InputAxisMapping. Кожне таке зв’язування може включати додаткові умови, такі як необхідність модифікаторів. Для аналогових дій, наприклад, безперервного руху використовуються InputAxisMapping, які дозволяють задати масштабування значення введення через параметр scale.

Ці прив’язки зберігаються у спеціальних контейнерах і можуть бути динамічно змінені під час роботи програми через методи AddActionMapping і

RemoveActionMapping. Для оптимізації пошуку прив’язок за іменем дії використовується механізм кешування у вигляді unordered_map<StringID, vector<InputActionMapping>>.

Клас InputComponent надає інструментарій для створення спеціалізованих обробників введення. Він дозволяє зв’язувати конкретні дії з метоадми об’єктів через систему делегатів. Наприклад, прив’язка дії «стрибок» до методу об’єкта.

Система тісно інтегрована з сиуляційною логікою через клас Controller. Приклад введення у симуляційному світі демонструє метод ViewportWidget::OnKeyDown, який передає події до PlayerController. Для аналогових дій використовується механізм накопичення зсувів з подальною їх передачею у вигляді параметрів осі.

Інструментом для роботи з фізичними клавішами та кнопками в системі є клас Key, який реалізує абстракцію над фізичними пристроями введення. Цей клас використовує ідентифікатор типу StringID для унікального позначення кожної клавіші, що дозволяє уніфікувати роботу з різними типами введення – клавіатурою та мишею. Клас надає методи для перевірки типу клавіші та її характеристик, що є важливим для коректної обробки аналогових пристроїв. Варто зауважити що в цій системі введення рух миші також вважається за клавішу Key, наприклад, MouseX та MouseY.

Для зручності роботи система включає статичний клас IKey, який містить стандартний набір констант для всіх підтримуваних клавіш. Ці константи ініціалізуються під час старту програми через статичний метод Initialize, який автоматично заповнює внутрішню мапу даних s_KeyMap інформацією про кожну клавішу. Наприклад, константи IKey::W або IKey::Space дозволяють зручно звертатися до конкретних клавіш без необхідності знання їх внутрішніх ідентифікаторів.

Клас KeyInfo виступає сховищем метаданих про кожну клавішу, зберігаючи такі властивості як належність до модифікаторів, належність до геймпаду, вимір вісі аналогового введення, необхідність постійного оновлення.

Ця інформація використовується системою для коректної обробки спеціальних випадків.

Клас KeyInputManager відповідає за взаємодію з апаратним рівнем, надаючи механізм перетворення фізичних кодів клавіш на внутрішній формат Key. Під час ініціалізації він викликає статичну функцію PopulateKeyArray класу PlatformInput для отримання списку всіх доступних клавіш конкретної платформи.

Для роботи зі складними комбінаціями клавіш система вводить структуру KeyWithMods, яка крім самої клавіші зберігає інформацію про стан модифікаторів. Вона використовується в InputKeyBinding для точного визначення умов активації дії. Методи Masks та Equal дозволяють порівнювати комбінації клавіш з урахуванням специфічних вимог до модифікаторів.

Для обробки станнів введення використовується клас KeyState, зберігаючи поточний стан кожної клавіші. Він містить поточне значення осі Value, необроблене значення RawValue, стани натиснутості bDown та bDownPrev, прапорець спожитості події bConsumed, та історію подій EventAccum.

Механізм накопичення подій EventAccum дозволяє обробляти ситуації, коли за кодин кадр відбувається кілька подій для однієї клавіші. Це забезпечує точність відтворення дій користувача незалежно від частоти оновлення програми.

Система пропонує кілька типів спеціалізованих прив’язок для різних сценаріїв використання:

- InputActionBinding;
- InputAxisBinding;
- InputKeyBinding;
- InputAxisKeyBinding.

Кожен тип прив’язки підтримує делегати, що дозволяє зв’язувати дії з методами будь-яких об’єктів. Для оптимізації продуктивності система

використовує механізм KeyToActionInfo, який кешує зв’язки між клавішами та діями. На рис. 3.9 зображено приклади використання цих прив’язок.

```
ic->BindKey(IKey::G, EInputEvent::Pressed, this, &CapucinoAgent::OnKeyJump);
ic->BindAxisKey(IKey::MouseX, this, &CapucinoAgent::LookYaw);
ic->BindAxisKey(IKey::MouseY, this, &CapucinoAgent::LookPitch);
ic->BindAction(StringID("Jump"), EInputEvent::Pressed, this, &CapucinoAgent::Jump);
ic->BindAxis(StringID("Move"), this, &CapucinoAgent::Move);
```

Рисунок 3.9 – Приклади використання InputBinding механізму

1.8. Додаткові системи

У VeiM пристуні системи що не є ключовими, проте допомагають підвищити зручність використання інших систем рушія, або його продуктивність. До таких систем відносяться «string interning» та делегати.

3.8.1. Система делегатів

Система делегатів у рушії VeiM забезпечує високопродуктивний і типобезпечний механізм зворотних викликів міє об’єктами. Делегати дозволяють інкапсулювати виклики методів, функцій, лямбда-виразів і std::function у зручну абстракцію, що може бути динамічно прив’язана та виконана в майбутньому.

Основними компонентами системи є шаблонні класи Delegate<Signature> та EventDelegate<Signature>. Перший дозволяє створити одиночну прив’язку до функції або методу з певним сигнатурним типом, другий – дозволяє підписатися кільком об’єктам на одне джерело події. Це створює зручну модель шаблону проектування спостерігача, коли різні частини рушія можуть реагувати на зміни стану або події без жорсткого зв’язуванняміж модулями.

Клас Delegate реалізує делегат, що може бути прив’язаний до:

- Статичної функції;

- Методу об’єкта;
- Константного методу об’єкта;
- std::function;
- лямбда-функції.

Прив’язка до методу класу автоматично використовує ObjectPtr, що гарантує безпечне керування життєвим циклом об’єкта-отримувача. Це дозволяє делегату не викликати метод, якщо об’єкт вже знищено, запобігаючи невизначеній поведінці. Для цього делегати перевіряють об’єкт функцією IsValid перед виконанням виклику.

Клас EventDelegate – це колекція делегатів, кожен з яких має унікальний DelegateHandle, який можна використати для відписки Unbind. Завдяки цьому механізму можливо реалізовувати складні системи подій. Подібна система спрощує побудову редактора, UI-подій, симуляційної логіки, синхронізації тощо.

Усі види об’єктів що викликаються обгортаються у нащадків абстрактного класу BoundFunctionBase, який визначає єдиний інтерфейс IsValid та Execute. Це дозволяє зберігати делегати у єдиній змінній незалежно від типу прив’язаної функції.

Система підтримує прив’язку методів під час компіляції, так і під час виконання з передачою вказівника на метод як параметру. Обидва варіанти забезпечують перевірку на те, що об’єкт походить від базового класу Object, тим самим захищаючи систему від спроб прив’язати несумісні об’єкти.

Система делегатів довзоляє створювати слабозв’язані системи, де компоненти можуть реагувати на події або взаємодіяти між собою без прямого доступу. Це підвищує гнучкість рушія, та швидкість розробки. Делекати широко застосовуються в рушії, особливо важливу роль відіграючи в системі подій вводу.

3.8.2. String interning

У сучасних рушіях важливо зменшити як обсяг оперативної пам’яті, так і вартість частих операцій з рядками, особливу у великих проєктах з великою кількістю ресурсів та іменованих сутностей. Рядки можуть забирати значний відсоток обчислювальної потужності, тому кожен рушій реалізує свій механізм string interning – техніку, що дозволяє уникнути повторного зберігання однакових рядків у пам’яті та прискорити їх порівняння.

Принцип string interning у VeiM реалізовано за допомогою класу StringID. Він дозволяє уникнути дублювання однакових рядків у пам’яті. Замість зберігання та порівняння самих рядків, кожен унікальний текст асоціюється з числовим ідентифікатором типу uint32, який використовується для швидких операцій порівняння та доступу. Такий підхід суттєво пришвидшує роботу з рядками та зменшує споживання пам’яті, особливо в системах де однакові рядки використовуються багаторазово. Для обчислення ідентифікаторів використовується швидкий і високоякісний алгоритм CityHash, що забезпечує рівномірний розподіл і продуктивність навіть для коротких рядків.

Клас StringID дозволяє створювати об’єкти як з «C-style» рядка, так і з елементів перерахування EStringID, який містить попередньо визначені ідентифікатори. Усі рядки зберігаються у внутрішньому пулі, де кожен унікальний рядок додається лише один раз. Після цього будь-яке створення StringID з тим же рядком призведе до повернення вже наявного ідентифікатора. Для підтримки фіксованих і стабільних ідентифікаторів, які не змінюються між перезапусками рушія, використовується окремий механізм реєстрації через RegisterCommonStrings.

Оператори порівняння перевантажені так, що порівняння StringID з іншими StringID або з EStringID виконується через зіставлення числових ID, що забезпечує ефективність. Крім того, реалізовано спеціалізований std::hash, що дозволяє використовувати StringID у хеш-таблицях в якості ключа. В якості

додаткової функціональності, підтримується повернення оригінального рядка через метод Get, а також зручне перетворення в тип String.

4. ТЕСТУВАННЯ

Метою тестування рушіє є перевірка коректності роботи рушія, стабільності, продуктивності та відповідності технічним вимогам. До першого етапу тестування входить перевірка інфраструктури користувацьких проєктів. Однією з головних цілей VeiM було забезпечення зручного використання рушія при взаємодії користувацького проєкту та редактора і автоматизація процесів інструментарію.

Тестування розпочалося із клонування репозиторію рушія з сервісу GitHub, командою «git clone --recursive https://github.com/SkaLe3/VeiM». Структуру завантажених файлів рушія зображено на рис. 4.1.

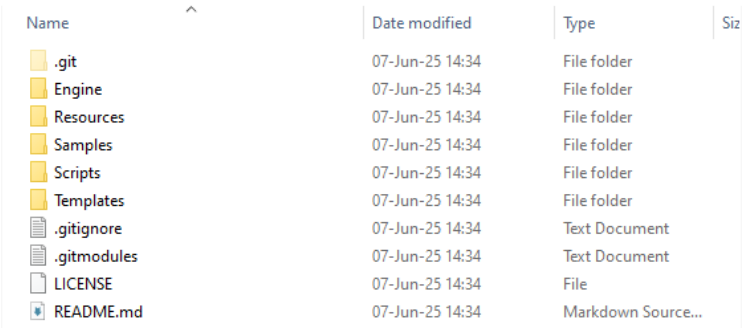


Рисунок 4.1 – Структура директорії з вихідним кодом рушія після клонування.

Для початку роботи з рушієм користувач має запустити скрипт Setup.bat в директорії Scripts, який запустить інструмент рушія VeimManagerTool з командою «/register». Ця комда виведе діалогове вікно (рис. 4.2) з пропозиціїєю зареєструвати типи файлів рушія VeiM, до яких відноситься тип «.vmproject». Незалежно від вибору корстувача, рушій буде зареєстровано в реєстрі Windows за шляхом «HKEY_CURRENT_USER\SOFTWARE\SkaLe Studio\VeiM Engine\Builds». Під

час тестування за цим шляхом було створено запис (рис. 4.3), що містить пару універсальний унікальний ідентифікатор – шлях до місця інсталяції рушія.

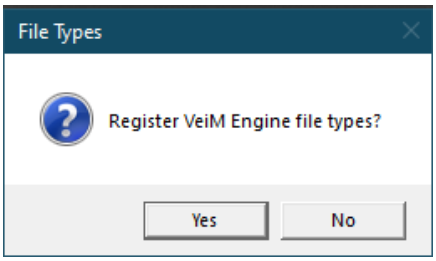


Рисунок 4.2 – Діалогове вікно реєстрації типів файлів.

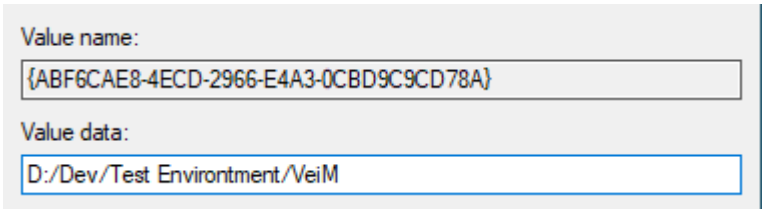


Рисунок 4.3 – Новий запис в реєстрі Windows.

У випадку якщо користувач образ зареєструвати типи файлів, в реєстрі з'явиться запис «VeiM.ProjectFile» за шляхом «HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.vmproject» та саме піддерево VeiM.ProjectFile з структурою що зображена на рис. 4.4. В цьому піддереві містяться записи що присвоюють файлам формату «.vmproject» певні властивості.

До цих записів належать:

1) DefaultIcon. Містить інформацію про шлях до виконуваного файлу VeimManagerTool, іконка якого має використовуватись для файлів формату «.vmproject».
2) open/command. Містить команду запуску VeimManagerTool з аргументом «/editor», виконання якої при запуску файлу проєкту призведе до запуску редактора і відкриття в ньому проєкту, до якого належить цей самий файл.

3) run/command. Містить команду запуску VeimManagerTool з аргументом «/game», виконання якої шляхом натиснення відповідного пункту контекстного меню призведе до відкриття користувацього застосунку редактором в мінімальному режимі без інтерфейсу редактора.
4) rungenproj. Містить команду запуску VeimManagerTool з аргументом «/projectfiles», виконання якої шляхом натиснення відповідного пункту контекстного меню призведе до генерації файлів середовища розробки для користувацького проєкту.

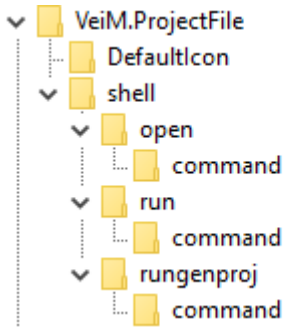


Рисунок 4.4 – Структура піддерева VeiM.ProjectFile

Після інсталяції рушія, користувач має запустити скрипт «Win-GenerateProjectFiles.bat» в директорії Scripts, який за допомогою утиліти premake згенерує файли середовища розробки, де користувач має зкомпілювати рушій в конфігурації Development_Editor. Рушій готовий до використання після компіляції редактора.

Для тестування управління проєктами слід почати з запуску редактора без аргументів командного рядка, що автоматично відриває лаунчер (рис. 4.5) – окреме вікно, яке виконує роль браузера проєктів. У ньому користувач може вибрати існуючий проєкт або створити новий.

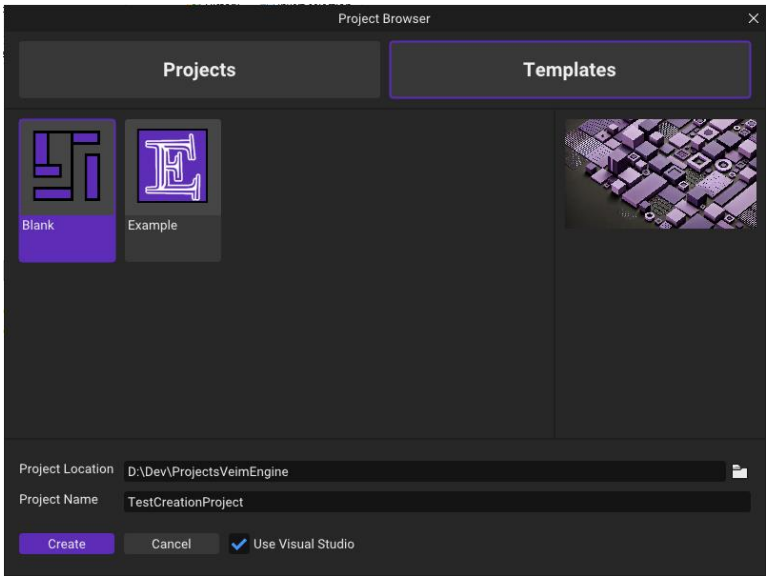


Рисунок 4.5 – Вікно лаунчера VeiM

Для створення нового проєкту користувачу необхідно перейти у вкладку Templates, та вибрати шаблон проєкту. В даному випадку було вибрано шаблон Blank що створює пустий проєкт. Після цього в поле Project Location користувач вводить або вибирає через графічний інтерфейс шлях в якому він бажає створити свій проєкт. В полі Project Name вводиться ім’я для нового проєкту, після чого натискається кнопка Create, внаслідок чого починається генерація нового проєкту. За вказаним шляхом лаунчер рушія створює нову папку з назвою проєкту, та копіює в неї файли шаблону, замінюючи плейсхолдери імені на справжнє ім’я проекту а також генерує додаткові файли для менеджменту проєкту. На рис. 4.6 зображено структуру директорії проєкту. Згенерований файл EngineRoot.lua зберігає шлях до встановленого рушія, яким було згенеровано проєкт, що потрібно для генерації файлів середовища розробки, та забезпечення можливості редагування вихідного коду рушія прямо в середовищі розробки під час написання користувацького проєкту.

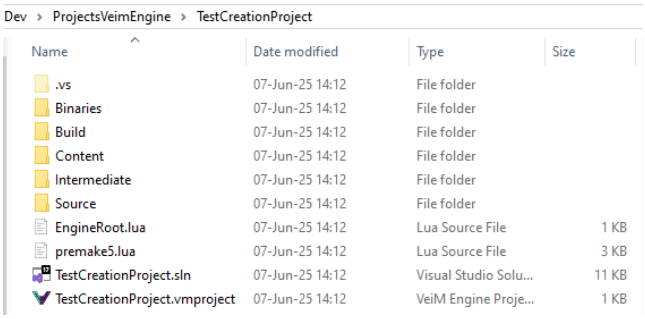


Рисунок 4.6 – Структура директорії проєкту

Після генерації директорії, лаунчер автоматично генерує файли середовища розробки, виконує компіляцію новоствореного проєкту, відкриває середовище розробки і запускає новий процес редактора з новим проєктом (рис. 4.7).

Центром керування проєктом є файл TestCreationProject.vmproject. Цей файл містить запис із ідентифікатором рушія, що використовується для ідентифікації конкретної інсталяції. Записи відповідності шляху і ідентифікатора містяться в реєстрі операційної системи Windows, сам вміст файлу наступний: «EngineAssociation: {6773411D-4B5F-5D7D-466E-5CA3FBA4131A}».

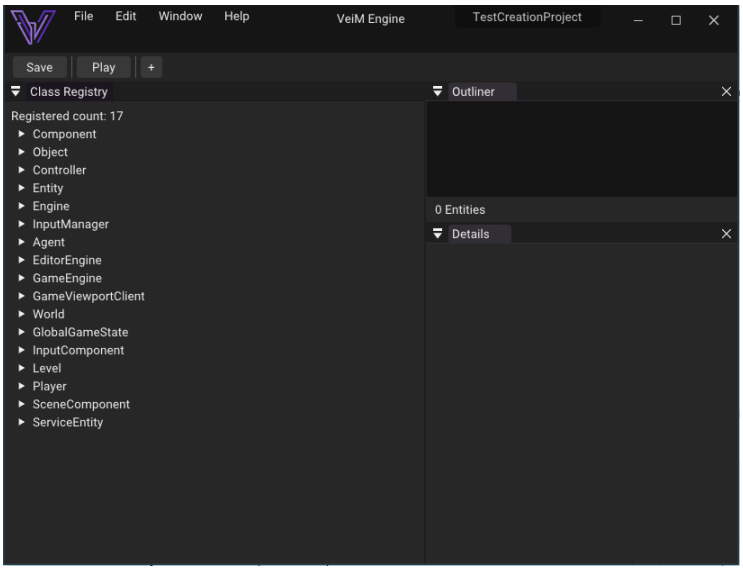


Рисунок 4.7 – Вікно редактора з відкритим проєктом TestCreationProject.

На рис. 4.8 зображено частину контекстного меню файлу TestCreationProject.vmproject з пунктами що були описані раніше в реєстрі.

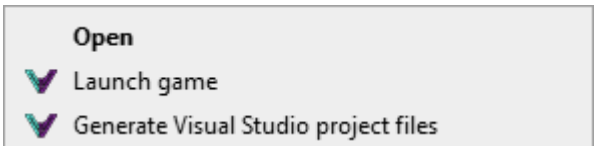


Рисунок 4.8 – Частина контекстного меню TestCreationProject.vmproject.

Після створення проєкту, він буде відображатись в лаунчері в категорії Projects. На рис. 4.9 видно наявність проєкту TestCreationProject.

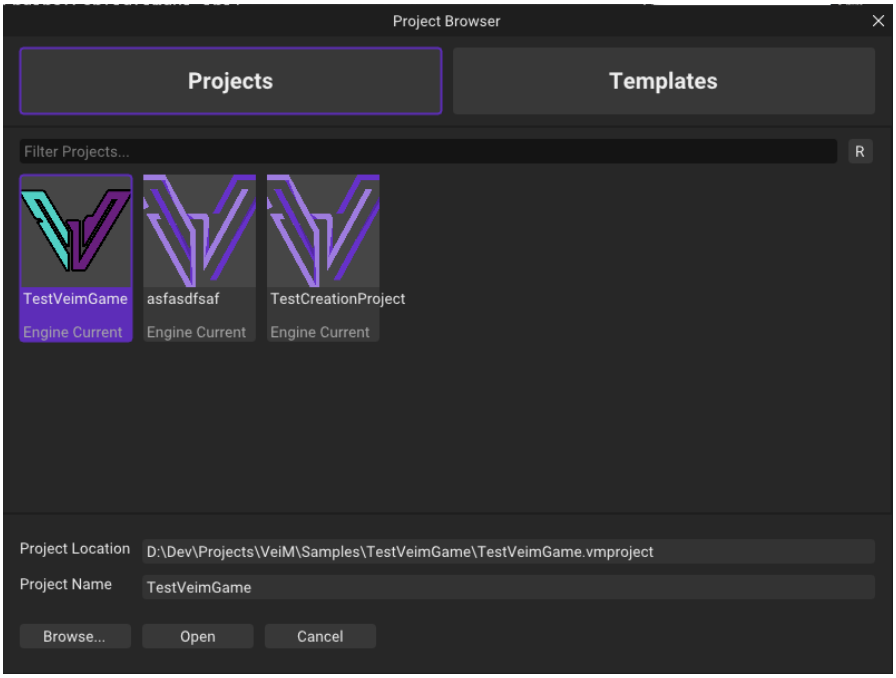


Рисунок 4.9 – Вікно лаунчеру з активною категорією Projects

Для тестування систем редактора було створено новий користувацький проєкт на основі шаблону Example, що містить демонстраційні класи та початкову логіку. Відкриття проєкту здійснювалося запуском редактора з передачею шляху до відповідного файлу конфігурації з розширенням «.vmproject» як аргументу командого рядка. Така передача параметрів відбувається автоматично під час виконання команди «open» новоствореного

файлу TestEngine.vmproject. Після запуску рушій успішно розпізнав конфігурацію проєкту, ініціалізував середовище та коректно підключив динамічну бібліотеку корситувацького модуля. Таким чином, була підтверджена стабільна робота всієї інфраструктури інтеграції з користувацькими проєктами. Рушій успішно підтримує повний базовий життєвий цикл роботи з користувацьким проєктом.

Після відкриття тестового проєкту стало можливим переглянути всі класи зареєстровані системою рефлексії. Для цього було використано вікно ClassRegistry, яке наочно відображає доступні типи. На рис. 4.10 показано відповідний інтерфейс, де зафіксовано 17 класів що відносяться до ядра рушія та 4 класи користувацького модуля: ExampleComponent, ExampleSceneComponent, ExampleEntity та ExampleAgent.

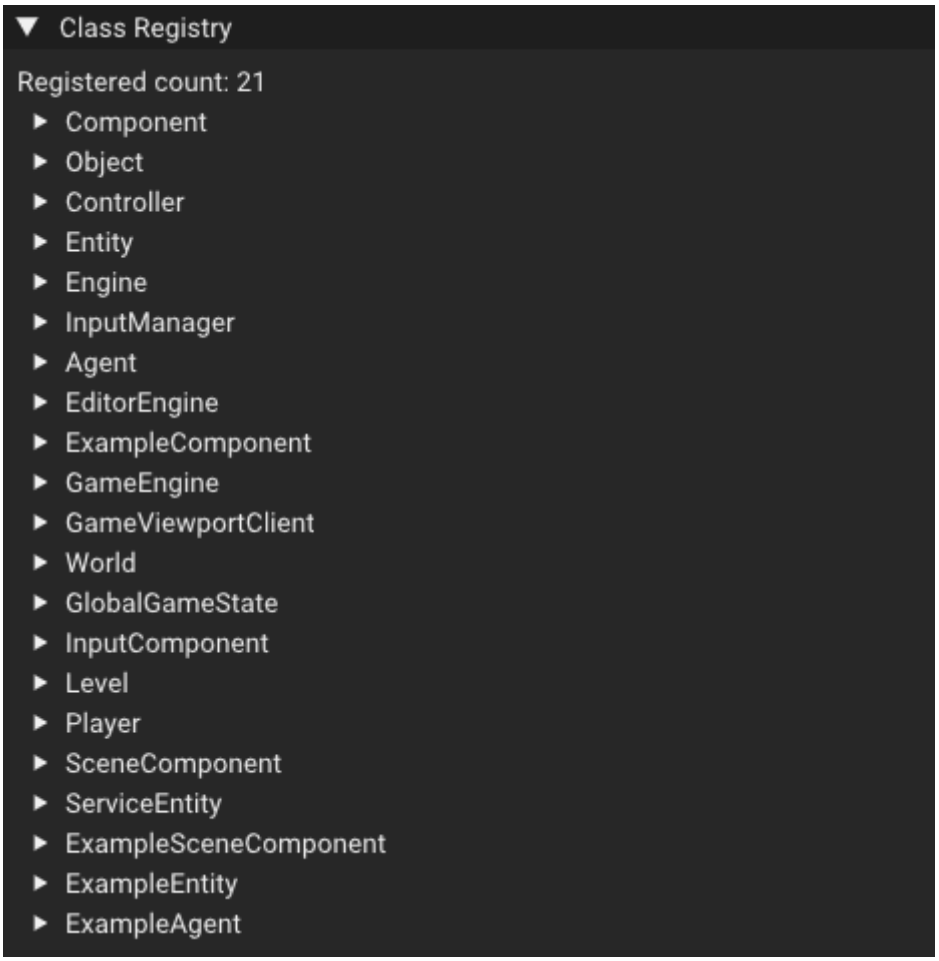


Рисунок 4.10 – Вікно Class Registry.

Додавання сутностей у сцену виконується через натискання кнопки «+», що відкриває відповідне меню(рис. 4.11). Створені сутності ExampleEntity та ExampleAgent після цього з’являються у вікні Outliner, яке відображає ієрархію сцени. На рис. 4.12 зображено ці сутності з прикладом ієрархічної структури компонентів ExampleEntity0. У нижній частині інтерфейсу видно елементи керування позицією компонентів сцени, які мають трансформацію.

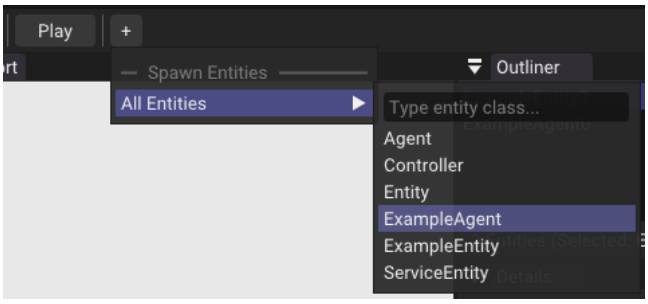


Рисунок 4.11 – Меню розміщення сутностей

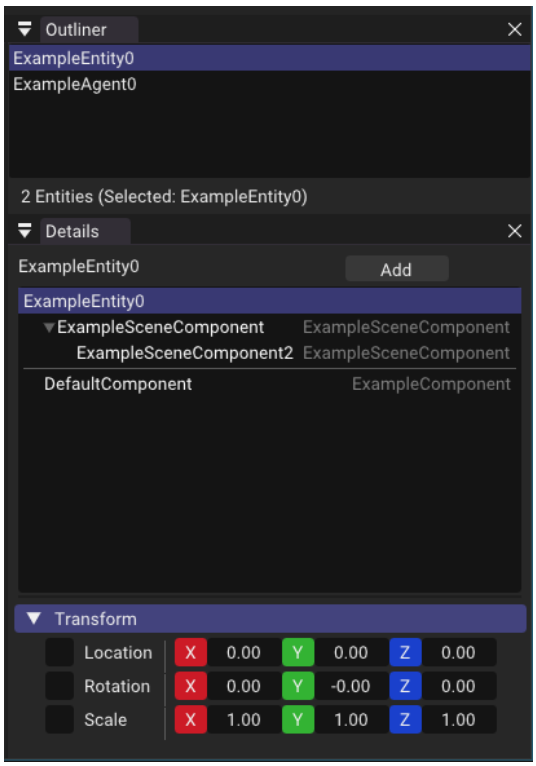


Рисунок 4.12 – Ієрархія сцени та компонентів

Запуск симуляції прямо в редакторі здійснюється натисканням кнопки «Play» (рис. 4.11), після чого рушій створює окремий екземпляр світу для симуляційної логіки та копією до нього всі сутності з редакторського середовища. На рис. 4.13 зображено новий стан сцени, де з'явилась нова сутність PlayerController. У цьому режимі автоматично створюється сутнстість класу Controller, для кожного гравця симуляції. За замовчуванням створюється лише один гравець, тому сутність PlayerController теж лише одна. Декілька гравців і відповідно декілька сутностей PlayerController можливі для застосування в локальному багатокористувацькому режимі. PlayerController забезпечує можливість взаємодії користувача з об'єктами сцени, завдяки чому сутність ExampleAgent0 може бути підконтрольна контролеру і отримувати події введення користувача через відповідну систему рушія. На рис. 4.14 зображено зразок можливого коду користувача, що виконує взяття під контроль сутності ExampleAgent0 сутністю PlayerController.

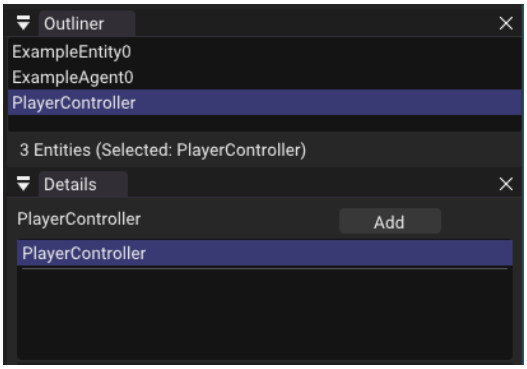


Рисунок 4.13 – Стан сцени після запуску

```
if (Controller* controller = GameStatics::GetController(0))
{
    controller->Posses(this);
}
```

Рисунок 4.14 – Код корстувача, що виконує взяття сутності під контроль

Користувач має налаштувати отримання подій введення та їх обробку. Для цього він має перевизначити функцію SetupInputComponent свого підкласу Agent, та додати прив'язки функцій до кнопок або подій. На рис. 4.15 зображено приклад такого налаштування в тестовому проєкті.

```
void ExampleAgent::SetupInputComponent(InputComponent* ic)
{
    Super::SetupInputComponent(ic);

    ic->BindAction(StringID("Jump"), EInputEvent::Pressed, this, &ExampleAgent::Jump);
    ic->BindAxis(StringID("Move"), this, &ExampleAgent::Move);
    ic->BindKey(IKey::G, EInputEvent::Pressed, this, &ExampleAgent::OnKeyJump);
    ic->BindAxisKey(IKey::MouseX, this, &ExampleAgent::LookYaw);
    ic->BindAxisKey(IKey::MouseY, this, &ExampleAgent::LookPitch);
}
```

Рисунок 4.15 – Приклад налаштування введення користувачем.

Для зручного перегляду результату до функцій обробки додано логування. Після запуску симуляції і виконання введення, в консоль було виведено повідомлення що підтверджують отримання введення миші і клавіатури (рис. 4.16). Користувач сам вирішує як далі його обробляти.

```
[18:38:06] VEIM: ExampleAgent::LookYaw: 1
[18:38:06] VEIM: ExampleAgent::LookPtich: 1
[18:38:08] VEIM: ExampleAgent::OnKeyJump
```

Рисунок. 4.16 – Логування введення.

Для тестування роботи системи оновлення, класи ExampleComponent, ExampleSceneComponent та ExampleEntity містять логування в функціях Tick. Після запуску симуляції з розміщеною сутністю ExampleEntity в консолі з'являється логування з цих методів. Логування проводиться кожен 600-ий, 700-ий та 800-ий виклик функції Tick відповідно. Результат логування зображено на рис. 4.17.

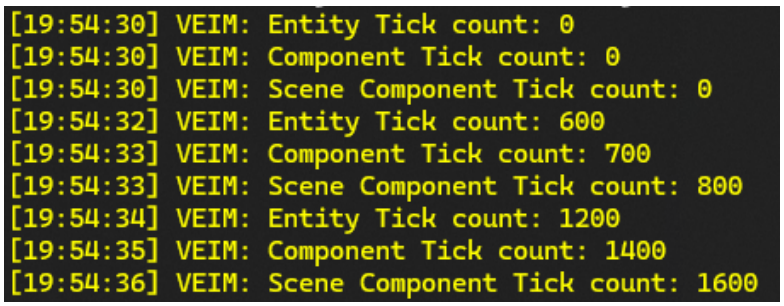


Рисунок 4.17 – Логування системи оновлення

Наступний етап тестування – перевірка працездатності підсистеми імпорту тривимірних моделей та їх рендерингу. У рамках цього процесу до проєкту було додато статичну тривимірну модель формату «.obj», яка складається з декількох геометричних мешів, а також має текстури і матеріали. Зокерма, для тесту було обрано високополігональну модель з UV-розгорткою та PBR-текстурами (diffuse, normal, roughness, specular).

Імпорт моделі здійснювався через внутрішній інструментарій рушія, який базується на бібліотеці assimp. У процесі імпорту засобами бібліотеки було зчинати модель, після чого сирі дані перетворено у внутрішній формат рушія. Для кращої демонстрації моделі, до сцени додано джерела світла двох типів, напрямне для імітації сонця, та точкове.

На рис. 4.18 зображено результат рендерингу імпортованої моделі в редакторі. Видно, що геометрія моделі відображається коректно, матеріали застосовано відповідно до оригінальних текстур, нормалі обчислюються правильно, а система освітлення коректно впливає на поверхні.

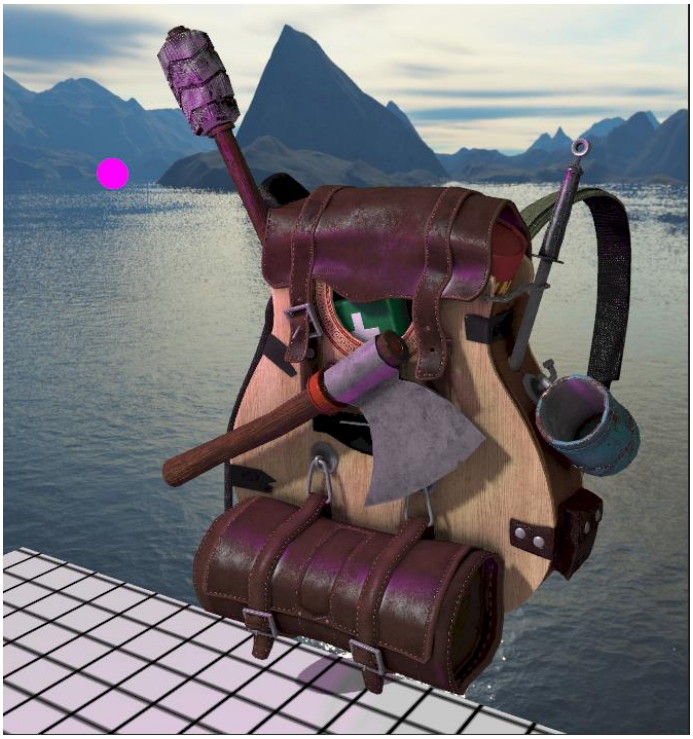


Рисунок 4.18 – Результат рендеру тривимірної моделі.

На рис. 4.19 зображено порівняння результатів рендеру моделі рюкзака з увімкненою та вимкненою мапою нормалі. На результатах рендеру видно зміни у відбитті світла на деяких частинах моделі що не мають змін геометрії. Це доводить коректність відображення матеріалу з мапою нормалі.

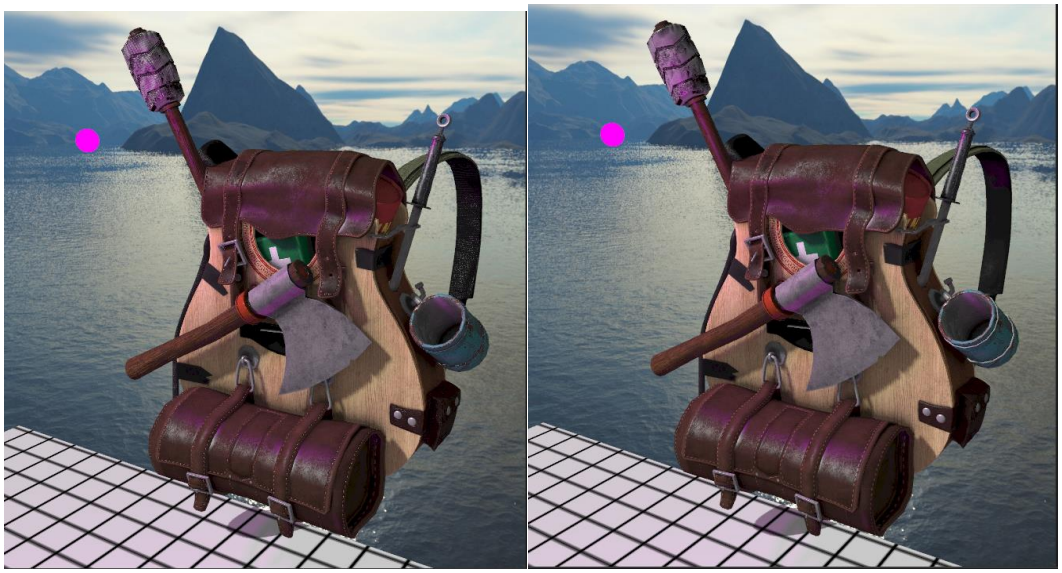


Рисунок 4.19 – Порівняння рендеру без мапи нормалі.

На рис. 4.20 зображено порівняння результатів рендеру моделі рюкзака з увімкненими та вимкненими тінями від джерел світла. Створення мапи тіней для обох видів джерел світла працює коректно.

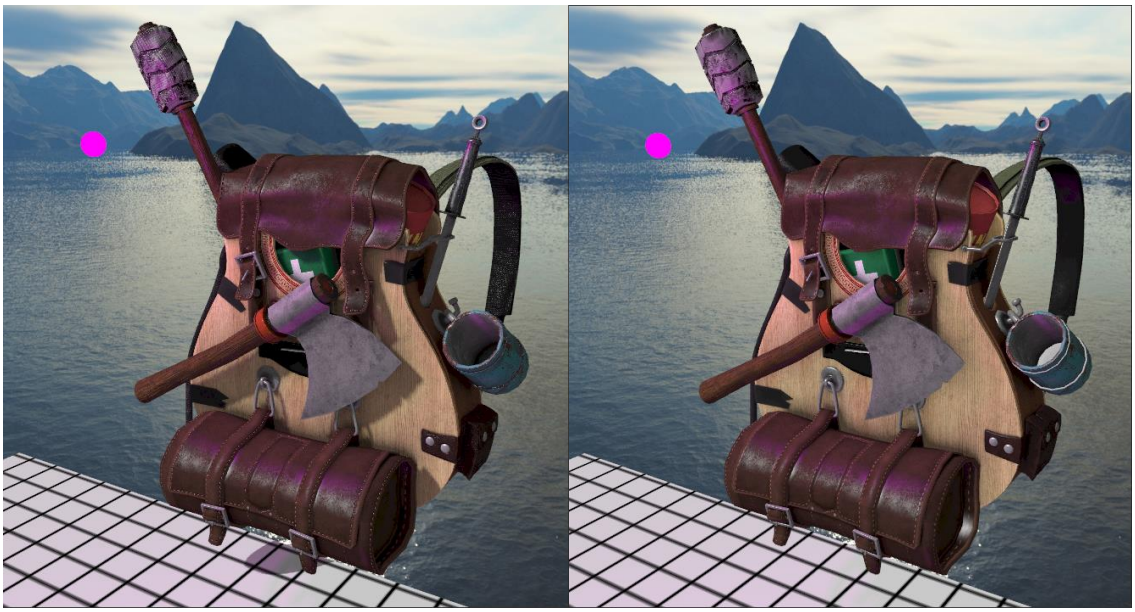


Рисунок 4.20 – Порівняння рендеру тіней

На рис. 4.21 зображено порівняння результатів рендеру моделі рюкзака з увімкненим та вимкненим згладжуванням MSAA 4X. Згладжування допомагає прибрати гострі кути на зображенні, роблячи його приємнішим для перегляду. Результат згладжування добре помітно на краях рюкзака, де перехід між заднім фоном та безпосередньо рюкзаком став плавнішим.

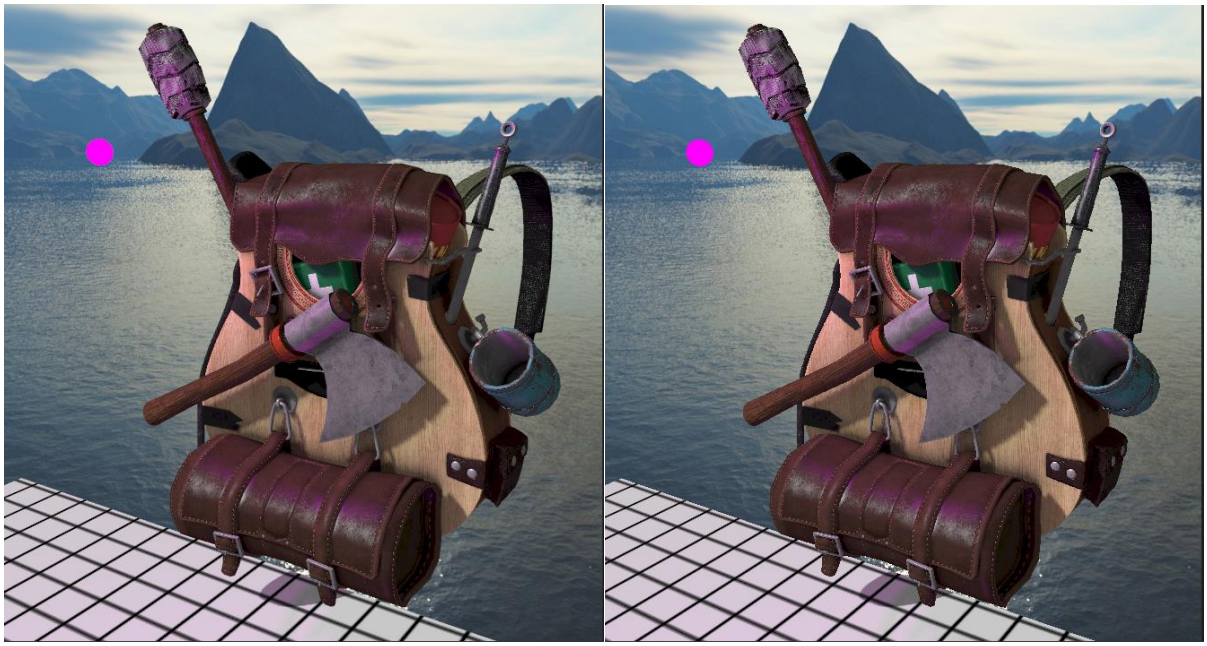


Рисунок 4.21 – Порівняння рендеру із згладжуванням.

На рис. 4.22 зображено порівняння результатів рендеру моделі рюкзака з увімкненим та вимкненим високим динамічним діапазоном (HDR). HDR дозволяє зберегти на картинці більше інформації за рахунок збереження значень поза межами стандартного діапазону. На етапі післяобробки ці значення переводяться в стандартний діапазон, в наслідок чого зображення зберігає більше інформації в дуже світлих та дуже темних регіонах. Можна побачити що застосування HDR працює коректно та робить зображення приємнішим для перегляду.

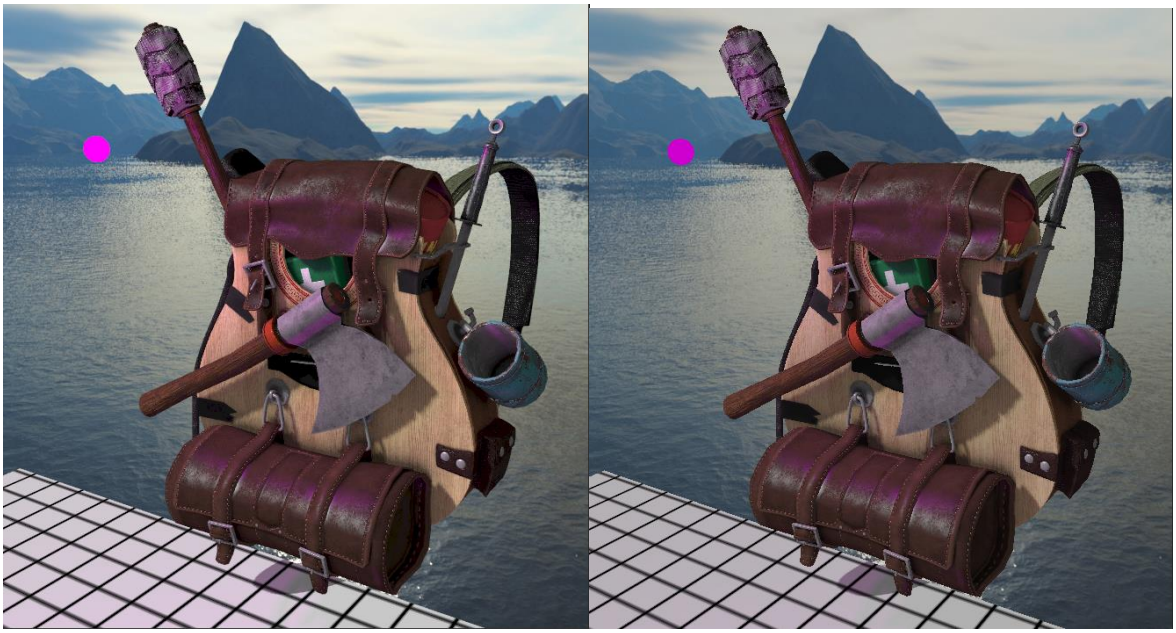


Рисунок 4.22 – Порівняння рендеру з HDR

Результати показали, що вся система імпорту й рендерингу працює стабільно: моделі зберігають свою геометричну і візуальну цілісність, а рендеринг відповідає очікуваному результату без графічних артефактів або збоїв.

Проведене тестування підтвердило, що рушій VeiM повністю відповідає вимогам до програмного продукту. Всі необхідні системи та інструменти працюють коректно та ефективно. Було успішно реалізовано і перевірено підтримку створення користувацьких проєктів, їх конфігурації та запуску. Розроблена система відповідає критеріям функціональнсоті системи користувацьких проєктів. Редактор забезпечує можливість додавання об’єктів на сцену, їх ієрархічну організацію, редагування параметрів і запуск симуляції в режимі реального часу. Робота системи компонентів, рафлексії та обробки введення також була підтверджена під час тестування, що демонструє стабільність і надійність архітектури рушія. Таким чином, рушій VeiM успішно виконує свої основні функції, демонструючи готовність до використання як платформи длястворення інтрактивних комп’ютерних симуляцій реального часу.

ВИСНОВОК

У даному дипломному проєкті було розроблено рушій тривимірної графіки для створення програмних інтерактивних агентно-орієнтованих комп'ютерних симуляцій м'якого реального часу. Розробка охоплює ключові аспекти архітектруи рушіїв, зокрема модульну структуру з підтримкою користувацьких проєктів.

У процесі роботи було здійснено глибокий аналіз існуючих рушіїв. Це дозволило виявити як їхні перваги, так і недоліки кожного з них, що стало основою для формування вимог до власного рушія. Жоден з розглянутих рушіїв не забезпечує одночасного виконання всіх вимог для користувача, який потребує від рушія таких якостей, як: простота використання, висока продуктивність, відкритий вихідний код, розширюваність, сводобода контролю над основними підсистемами, а також наявність усіх необхідних засобів для створення повноцінних інтерактивних агентно-орієнтованих комп'ютерних симуляцій.

На основі цих висновків у дипломному проєкті було реалізовано рушій, що поєднує переваги простоти, відкритості, продуктивності та гнучкості. Його архітектура дозволяє створювати окремі користувацькі проєкти, які забезпечують розділення логіки редактора та кінцевого застосунку. Рушій уникає складності та надмірної універсальності, притаманної великим рушіям, водночас залишаючи простір для масштабування та розширення функціональності.

Розроблений рушій є зручним інструментом для створення різноманітних інтерактивних агентно-орієнтованих симуляцій. Завдяки модульній структурі та відкритій архітектурі він забезпечує можливість подального розширення, адаптації до нових задач і перспективного використання як у навчальних, так і в комерційних проєктах.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Learn OpenGL – Graphics Programming [Електронний ресурс]. Режим доступу: https://learnopengl.com (дата доступу: 24.05.2025)
2. Forward and Deferred Rendering - Cambridge Computer Science Talks [Електронний ресурс]. Режим доступу: https://www.youtube.com/watch?v=n5OiqJP2f7w&ab_channel=BenAndrew
3. Jason Gregory. Game Engine Architecture. CRC Press Taylor & Francis Group: Boca Raton, 2019.
4. Mark-and-Sweep: Garbage Collection Algorithm [Електронний ресурс]. Режим доступу: https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/ (дата доступу: 24.05.2025).
5. Game++. String interning [Електронний ресурс]. Режим доступу: https://akhenatengame.squarespace.com/devblog/game-string-interning (дата доступу: 24.05.2025).
6. Engine Internals: Reflection [Електронний ресурс]. Режим доступу: https://medium.com/@heinapurola/engine-internals-reflection-f671ddbead7f (дата доступу: 24.05.2025).
7. Dear ImGui [Електронний ресурс]. Режим доступу: https://github.com/ocornut/imgui/wiki/ (дата доступу: 24.05.2025).
8. GLFW [Електронний ресурс]. Режим доступу: https://www.glfw.org/documentation (дата доступу: 24.05.2025).
9. OpenGL [Електронний ресурс]. Режим доступу: https://docs.gl/ (дата доступу: 24.05.2025).
10. Designing a Robust Input Handling System for Games [Електронний ресурс]. Режим доступу: https://www.gamedev.net/blog/355/entry-2250186-designing-a-robust-input-handling-system-for-games/ (дата доступу: 24.05.2025).

11. Architecture of a Game Engine [Електронний ресурс]. Режим доступу: https://grier.hashnode.dev/tinker-engine (дата доступу 24.05.2025).
12. Reliable and Efficient Agent-Based Modeling and Simulation [Електронний ресурс]. Режим доступу: https://www.jasss.org/27/2/4.html (дата доступу 24.05.2025).
13. Entity Component System: An Introductory Guide [Електронний ресурс]. Режим доступу: https://www.simplilearn.com/entity-component-system-introductory-guide-article (дата доступу 24.05.2025).
14. Mastering Unity: A Comprehensive Guide to Delegates and Event Handling for Seamless Game Development 2024 [Електронний ресурс]. Режим доступу: https://bleedingedge.studio/blog/mastering-delegates-and-event-handling-24/ (дата доступу 24.05.2025).

					ІАЛЦ. 045490.004 ПЗ		
Зм	*Лист*	*№ докум.*	*Підп.*	*Дата*			
Розроб.		Парієнко			Рушій для створення програмних інтерактивних агентно-орієнтованих комп'ютерних симуляцій м'якого реального часу ***Пояснювальна записка***	*Літ.*	*Лист* 1 / *Листів* 4
Перев.		Тарасенко-					
		-Клятченко О.В.				**НТУУ «КПІ ім. Ігоря Сікорського», ФПМ, КВ-11**	
Н. контр.		Клятченко Я.М.					
Затв.		Тарасенко В.П.					