m-math_2025_021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРНІВЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЮРІЯ ФЕДЬКОВИЧА

Факультет математики та інформатики
кафедра математичного моделювання

РОЗРОБКА ІГРОВОГО РУШІЯ ДЛЯ СТВОРЕННЯ 2D-ІГОР ІЗ ВІЗУАЛЬНИМ РЕДАКТОРОМ

Кваліфікаційна робота

Рівень вищої освіти – другий (магістерський)

Виконав:
студент 2 курсу, 601 групи
Яковець Дмиро Григорович
Керівник:
кандидат фізико-математичних наук,
асистент Дорош А.Б.

До захисту допущено
на засіданні кафедри
протокол № 5 від 26 листопада 2024 р.
Зав. кафедрою _____ проф. Черевко І.М.

Чернівці – 2024

АНОТАЦІЯ

У даній роботі було розроблено власний ігровий рушій для створення комп'ютерних 2D-ігор зі зручним графічним інтерфейсом користувача, який спрощує процеси розробки гри. Редактор дозволяє розробникам швидко, редагувати сутності, додавати компоненти, створювати скрипти, керувати ресурсами та компілювати гру.

Ключові слова: Редактор, WPF, C#, C++, Vulkan

ABSTRACT

In this work, a custom game engine was developed for creating 2D computer games with a user-friendly graphical interface that simplifies the game development process. The editor allows developers to quickly edit entities, add components, create scripts, manage resources, and compile the game.

Keywords: Editor, WPF, C#, C++, Vulkan

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

_____ Д.Г. Яковець
(підпис)

Зміст

ВСТУП

Актуальність роботи. Нестандартні ігрові рушії вимагають спеціальних редакторів, оскільки робота з файлами самого рушія та API може зайняти багато часу та бути складною.

Мета роботи. Мета цього проєкту полягала в тому, щоб розробити зручний візуальний редактор для створеного нами раніше ігрового рушія, який би надавав зручні іструменти для роботи зі сценами, керування ресурсами та тестування функцій гри.

Завдання роботи – розширити розроблений раніше рушій 2D-ігор зручним візуальним редактором. Для досягнення поставленої мети було виконано наступні завдання:

- Розроблена модульна та масштабована архітектура для редактора, що дозволяє легко розширювати та налаштовувати його.
- Реалізовано інтеграцію рушія, що забезпечує візуалізацію сцени в режимі реального часу.
- Розроблені інструменти для завантаження, упорядкування та редагування аудіо - та графічних ресурсів.
- Реалізовано інтуїтивний інтерфейс для розміщення об'єктів, налаштування властивостей і визначення логіки гри.
- Створив систему для створення інтерфейсів користувача, такі як кнопки, меню та інші елементи.
- Систему скриптів.
- Додано можливість скомпілювати проєкт у готову гру.

Наукова новизна та практична цінність використовуючи переваги передових технологій, таких як C++ і WPF C#, цей проєкт містить спеціалізований редактор, розроблений спеціально для власного двовимірного ігрового рушія.

Практичне значення. З практичної точки зору, редактор скорочує час, необхідний для розробки гри, спрощуючи та прискорюючи процес. Розробники можуть з легкістю створювати, тестувати та покращувати ігри за допомогою

цього інструменту, зменшуючи витрати та час на розробку. Дозволяє швидко та просто сворити дестирибутив гри.

РОЗДІЛ 1. Використані технології для реалізації проєкту

1.1 C++

C++ – це високорівнева мова програмування, яка має підтримку об’єктно-орієнтованого, процедурного та загального програмування. Вона була створена Б'ярном Страуструпом у 1983 році як розширення мови C. Головною метою розробки C++ було надання розробникам можливості використовувати об’єктно-орієнтовані підходи разом з усіма перевагами, які надавала мова C, зберігаючи її ефективність і низькорівневий доступ до апаратного забезпечення.

C++ є багатопарадигмальною мовою, що робить її гнучкою для різних типів завдань. Мова дозволяє використовувати як процедурний, так і об’єктно-орієнтований стиль програмування, що робить її ідеальною для створення як низькорівневих системних додатків, так і складних програмних систем.
Основні можливості мови:

1. Об’єктно-орієнтоване програмування (ООП)
    - Класи та об’єкти. Основою ООП є концепції класів і об’єктів. Клас є шаблоном для створення об’єктів, які мають дані (поля) і функції (методи), що маніпулюють цими даними.
    - Інкапсуляція. Завдяки механізму інкапсуляції, C++ дозволяє ховати внутрішню реалізацію класів, надаючи доступ до даних лише через спеціально визначені методи. Це забезпечує контроль доступу до даних і підвищує безпеку коду.
    - Наслідування. C++ дозволяє створювати нові класи на основі вже існуючих, використовуючи механізм наслідування. Це сприяє повторному використанню коду та створенню ієрархічних структур.
    - Поліморфізм. Поліморфізм дозволяє функціям і методам обробляти об’єкти різних типів через єдиний інтерфейс. C++ підтримує два типи поліморфізму: компіляційний (через шаблони) та виконуваний (через віртуальні функції).
2. Шаблони (Templates) Шаблони в C++ дозволяють створювати узагальнені класи та функції, які працюють з будь-якими типами даних. Це робить код більш гнучким і дозволяє уникати дублювання коду. Шаблони є основою для створення STL (Standard Template Library), яка містить узагальнені алгоритми і структури даних.
3. Стандартна бібліотека шаблонів (STL) STL – це потужний інструмент для роботи з контейнерами, ітераторами і алгоритмами. Вона містить базові структури даних, такі як списки, вектори, множини та асоціативні контейнери, а також функції для обробки цих структур.

4. Керування пам'яттю C++ дозволяє програмісту повністю контролювати виділення і звільнення пам’яті, що дає можливість писати високоефективний код, але водночас вимагає особливої уваги, оскільки неправильне керування пам’яттю може призвести до витоків пам’яті чи помилок у роботі програми. У C++ існують два основні способи керування пам'яттю:
    - Автоматичне виділення пам'яті. Стекові змінні автоматично звільняються після виходу з області видимості.
    - Динамічне виділення пам'яті. Оператори new і delete. використовуються для виділення і звільнення пам'яті в купі.
5. Указівники та низькорівнева робота з пам’яттю Указівники – це змінні, що зберігають адреси в пам’яті. Вони дозволяють програмістам працювати безпосередньо з пам’яттю, що надає великі можливості для оптимізації, але вимагає особливої обережності.
    - Указівники дозволяють реалізувати структури даних, такі як динамічні масиви, зв’язані списки і дерева.
    - Робота з указівниками може спричинити критичні помилки, такі як доступ до невиділеної області пам’яті або витік пам’яті.

1.2 WPF

WPF (Windows Presentation Foundation) – це сучасна технологія, розроблена компанією Microsoft для створення настільних додатків із багатими користувацькими інтерфейсами (UI) для операційної системи Windows. WPF була представлена у .NET Framework 3.0 як частина платформи .NET і забезпечує потужні інструменти для розробки динамічних, інтерактивних та візуально привабливих додатків. Можливості WPF:

1. Векторна графіка
    - WPF використовує векторну графіку для відображення елементів інтерфейсу, що дозволяє масштабувати вміст без втрати якості. Завдяки цьому інтерфейс додатків може виглядати чітким незалежно від роздільної здатності екрана.
    - Це також спрощує створення графічних ефектів, таких як обертання, масштабування та анімації.
2. XAML (Extensible Application Markup Language)
    - XAML – це мова розмітки, що використовується для опису візуальних елементів і макетів у WPF. Вона дозволяє розробникам розділяти логіку інтерфейсу від логіки програмного коду, що полегшує розробку та підтримку великих додатків.

3. Прив'язка даних (Data Binding)
    - Однією з найпотужніших функцій WPF є підтримка прив'язки даних, яка дозволяє з'єднувати елементи інтерфейсу з даними, що зберігаються у джерелах даних (наприклад, у базі даних, файлах або об'єктах програми).
    - Прив'язка даних забезпечує автоматичне оновлення елементів UI при зміні значень у моделі даних, що сприяє створенню гнучких і динамічних інтерфейсів.
4. Стилі та шаблони (Styles and Templates)
    - WPF дозволяє розробникам налаштовувати зовнішній вигляд елементів UI за допомогою стилів і шаблонів. Це дозволяє створювати єдиний стиль для всього додатку або окремих елементів, що значно покращує читабельність коду.
    - Стилі визначають властивості елементів (наприклад, кольори, розмір шрифтів), а шаблони визначають, як елемент буде виглядати і як його вміст буде відображатися.
5. Анімація
    - WPF має вбудовану підтримку анімації для створення привабливих візуальних ефектів. Анімації можна використовувати для зміни властивостей елементів, таких як розмір, положення, колір тощо.
    - Анімації в WPF легко визначаються за допомогою XAML або програмно, що дозволяє створювати як прості, так і складні візуальні ефекти.
6. Система макетів (Layout System)
    - WPF пропонує гнучку систему макетів для створення динамічних інтерфейсів, які адаптуються до різних розмірів і орієнтацій вікон.
    - Основні панелі макетів включають:
        - Grid: дозволяє розбивати інтерфейс на рядки та стовпці.
        - StackPanel: розміщує елементи вертикально або горизонтально.
        - WrapPanel: розміщує елементи у рядок, який переноситься, якщо закінчується простір.
        - Canvas: дозволяє точно позиціонувати елементи за координатами.
7. Ресурси (Resources)
    - У WPF можна визначати ресурси, такі як стилі, шрифти, та використовувати їх у різних частинах додатку. Це дозволяє легко змінювати зовнішній вигляд всього додатку, змінивши лише одне місце в коді.
    - Ресурси можуть бути визначені на рівні додатку, вікна або окремого елемента.
8. Підтримка 3D-графіки

- WPF має вбудовану підтримку 3D-графіки, що дозволяє створювати об’ємні моделі і інтерактивні тривимірні інтерфейси. 3D-графіка в WPF базується на використанні векторних об'єктів та дозволяє працювати з геометрією, освітленням і камерами.

9. Потужна модель подій (Event System)
    - WPF використовує покращену модель подій порівняно з традиційними Windows Forms. Існують такі типи подій, як маршрутизовані події (routed events), які дозволяють обробляти події на рівні контейнерів, навіть якщо джерело події – вкладений елемент.
    - Події можуть бути оброблені у вигляді піднімальних подій (bubbling events), коли подія проходить через дерево елементів від дочірнього до батьківського, або тунельних подій (tunneling events), коли подія проходить у зворотному напрямку – від батьківського до дочірнього.

1.3 Vulkan

Vulkan API є одним із найбільш передових графічних та обчислювальних інструментів для створення додатків з високими вимогами до продуктивності. Його низькорівневий підхід надає розробникам майже повний контроль над GPU, дозволяючи використовувати апаратні ресурси максимально ефективно. Це включає оптимізацію роботи з пам’яттю, точний розподіл обчислювальних завдань і налаштування шейдерів, що критично важливо для графічно інтенсивних задач.

Одна з ключових переваг Vulkan — це можливість ручного керування пам’яттю. Розробники мають змогу визначати, як і коли виділяти або звільняти ресурси, що дозволяє уникнути непотрібних накладних витрат, притаманних автоматичному керуванню. Це відкриває двері для точних оптимізацій, які є неможливими при використанні традиційних високорівневих API.

Vulkan дозволяє організовувати команди в черги та працювати з командними буферами, які потім передаються на виконання GPU. Це дає змогу максимально ефективно використовувати ресурси GPU та CPU, розподіляючи роботу між ними. Наприклад, команда рендерингу може бути виконана одночасно з підготовкою нових даних для обчислень.

На відміну від старіших API, таких як OpenGL, Vulkan спеціально створений для підтримки багатопотокового середовища. Розробники можуть використовувати всі ядра процесора, розподіляючи обчислювальні завдання між

потоками. Це особливо корисно в ігрових додатках, де завдання фізики, анімації та рендерингу можна виконувати паралельно, зменшуючи затримки.

Vulkan має вражаючу мультиплатформенну сумісність, дозволяючи працювати як на потужних настільних ПК, так і на мобільних пристроях. Завдяки MoltenVK, Vulkan також підтримується на macOS, що робить його універсальним інструментом для кросплатформеної розробки.

Окрім графічного рендерингу, Vulkan підтримує обчислювальні задачі завдяки Compute Shaders. Це дозволяє використовувати API не лише для ігор, а й для інших графічно інтенсивних завдань, таких як машинне навчання, обробка зображень, симуляції та візуалізація даних.

Vulkan API має модульну архітектуру, що дозволяє додавати нові функції через розширення. Це означає, що розробники можуть додавати або використовувати специфічні можливості для конкретних пристроїв або задач. Наприклад, виробники GPU, такі як NVIDIA та AMD, можуть пропонувати власні розширення для покращення продуктивності або додавання нових функцій.

На відміну від OpenGL, Vulkan API забезпечує більшу передбачуваність і стабільність. Він не допускає багатьох можливих помилок у процесі виконання завдяки обов’язковій валідації та попередньому налаштуванню всіх ресурсів. Це дає розробникам впевненість у стабільності роботи додатку на різних пристроях.

Vulkan використовує SPIR-V, проміжний мова для шейдерів, що забезпечує кращу продуктивність порівняно з традиційними мовами, такими як GLSL. SPIR-V дозволяє компілювати шейдери на етапі розробки, зменшуючи витрати на компіляцію під час виконання програми.

Основні обєкти Vulkan:

1. **Інстанс (Instance) -** Інстанс є головним об’єктом Vulkan, який представляє додаток і ініціалізує взаємодію з API. Через інстанс додаток отримує доступ до фізичних пристроїв (GPU), розширень та шарів валідації. Інстанс також виступає точкою входу для налаштування середовища виконання Vulkan.
2. **Фізичний пристрій (Physical Device) -** Фізичний пристрій представляє конкретне апаратне забезпечення, таке як GPU або інтегрований графічний процесор. Через фізичний пристрій розробники можуть отримувати інформацію про можливості GPU, доступні формати

зображень, типи пам’яті та підтримувані функції, такі як трасування променів або багатопотоковість.

3. **Логічний пристрій (Logical Device) -** Логічний пристрій — це об’єкт, який дозволяє додатку працювати з вибраним фізичним пристроєм. Через логічний пристрій розробники створюють черги команд, керують ресурсами та виконують операції рендерингу або обчислення. Логічний пристрій забезпечує доступ до специфічних функціональних можливостей фізичного пристрою.
4. **Командний буфер (Command Buffer) -** Командний буфер використовується для зберігання команд, які GPU виконує під час рендерингу або обчислювальних задач. Команди можуть включати малювання геометрії, копіювання даних між буферами або синхронізацію ресурсів. Цей об’єкт дозволяє розподіляти завдання між потоками та оптимізувати обчислення.
5. **Черга команд (Command Queue) -** Черга команд представляє чергу для виконання команд, записаних у командні буфери. Vulkan дозволяє використовувати кілька черг, які можуть виконуватись паралельно, залежно від можливостей апаратного забезпечення. Це дозволяє одночасно виконувати рендеринг, обчислення та операції з передачею даних.
6. **Буфери та зображення (Buffers and Images) -** Буфери та зображення є основними об’єктами для зберігання даних у Vulkan. Буфери використовуються для зберігання інформації, наприклад вершин або індексів, тоді як зображення зберігають текстури, карти глибини або кольорові дані. Розробники мають повний контроль над їхньою пам’яттю та організацією.
7. **Пам'ять (Memory) -** Vulkan API надає розробникам прямий доступ до керування пам’яттю. Об’єкт пам’яті прив’язується до буферів або зображень для виділення ресурсів. Завдяки цьому можна досягти максимальної ефективності через оптимізацію розподілу пам’яті для специфічних задач.
8. **Семплери (Samplers) -** Семплери використовуються для визначення, як текстури застосовуються до поверхонь. Вони налаштовують параметри фільтрації, відображення текстур у різних масштабах і поведінку при виході за межі текстури. Це дозволяє отримати контроль над виглядом матеріалів і текстур у рендерингу.
9. **Шейдерні модулі (Shader Modules) -** Шейдерні модулі є контейнерами для коду шейдерів, які виконуються на GPU. У Vulkan шейдери пишуться на SPIR-V, що забезпечує їхню ефективну обробку. Шейдери можна використовувати для завдань, таких як вершинна трансформація, розрахунок освітлення або обчислення фізичних симуляцій.

10. **Рендер-проходи та підпроходи (Render Passes and Subpasses) -** Рендер-проходи визначають порядок і способи рендерингу, зокрема, які цілі (кольорові буфери, буфери глибини) використовуються. Підпроходи дозволяють організувати виконання складних рендеринг-завдань, наприклад постобробки, у межах одного рендер-проходу.
11. **Синхронізація (Synchronization) -** Синхронізація у Vulkan здійснюється за допомогою об’єктів, таких як семафори та фехенси. Вони забезпечують правильну послідовність виконання команд і дозволяють координувати обчислювальні процеси між CPU та GPU або між різними чергами команд.

1.4 C#

C# (Сі Шарп) – це сучасна мова програмування, розроблена компанією Microsoft у 2000 році під керівництвом Андерса Гейлсберга. Вона була створена як частина платформи .NET і призначена для створення різноманітного програмного забезпечення: від простих консольних додатків до масштабних веб-сервісів, мобільних ігор і корпоративних систем. C# поєднує в собі простоту та елегантність синтаксису мов високого рівня, таких як Java, із потужними інструментами для низькорівневого програмування, притаманними C++.

C# є повністю об’єктно-орієнтованою мовою програмування, що означає, що основою для створення додатків є об’єкти та класи. Класи визначають шаблони для створення об’єктів, які інкапсулюють дані (поля) та поведінку (методи). Це дозволяє створювати програми, які легше підтримувати, розширювати та тестувати. Наприклад, використання інкапсуляції допомагає приховувати деталі реалізації та відкривати лише необхідні функції через публічний інтерфейс, забезпечуючи захист даних. Поліморфізм і наслідування спрощують розробку складних ієрархій класів, дозволяючи створювати гнучкі та багаторазово використовувані компоненти.

Однією з головних переваг C# є автоматичне керування пам’яттю через вбудований механізм збирача сміття (Garbage Collector, GC). Це дозволяє уникати типових помилок, пов’язаних із ручним керуванням пам’яттю, таких як витоки пам’яті або подвійне звільнення ресурсу. Збирач сміття автоматично звільняє об’єкти, які більше не використовуються, що забезпечує ефективне використання пам’яті та стабільну роботу програми. Це спрощує процес розробки, оскільки розробники можуть зосередитися на функціональності додатку, не турбуючись про керування ресурсами.

C# є строго типізованою мовою, що забезпечує виявлення помилок ще на етапі компіляції. Типи даних у C# поділяються на значимі (value types) та посилальні (reference types). Значимі типи, такі як int, float чи bool, зберігаються у стеку, що забезпечує швидкий доступ, тоді як посилальні типи, наприклад string чи об'єкти класів, розміщуються у купі. Завдяки цьому підходу C# забезпечує високу безпеку коду, зменшуючи ймовірність непередбачених помилок, наприклад через некоректні перетворення даних.

C# надає широкий набір інструментів для реалізації об’єктно-орієнтованих принципів. Абстрактні класи дозволяють створювати базові класи, які можна успадковувати, але не можна створити екземпляр, тоді як інтерфейси визначають набір методів, які клас повинен реалізувати. Крім того, віртуальні методи дозволяють перевизначати поведінку методів у похідних класах, що забезпечує динамічний поліморфізм. Також C# підтримує властивості (properties), що слугують для зручного доступу до приватних полів, дозволяючи реалізувати механізми перевірки та обчислення.

LINQ — це одна з найбільш зручних і потужних функцій C#, яка дозволяє писати запити до колекцій даних у стилі SQL. За допомогою LINQ розробники можуть ефективно працювати з масивами, списками, базами даних і навіть XML-документами. Наприклад, LINQ-запити можуть фільтрувати, сортувати або групувати дані, роблячи код більш читаємим і зрозумілим. Це значно скорочує кількість коду, який потрібно писати для виконання складних операцій із даними.

Асинхронність у C# реалізована через ключові слова async і await, які дозволяють створювати програми, що не блокують головний потік виконання. Це особливо корисно для роботи з мережею, базами даних або великими файлами, де очікування відповіді може займати тривалий час. Асинхронне програмування робить додатки більш масштабованими, оскільки вивільняє потоки CPU для виконання інших завдань, що особливо важливо для веб-додатків або мобільних застосунків із високим навантаженням.

Делегати — це спеціальні об’єкти, які можуть зберігати посилання на методи. Вони широко використовуються для реалізації зворотних викликів (callbacks) або передачі поведінки в методи. Події у C# реалізують механізм публікації-підписки, дозволяючи класам сповіщати інші об’єкти про зміни стану. Наприклад, у графічних інтерфейсах події повідомляють про натискання кнопок. Лямбда-вирази надають компактний синтаксис для створення анонімних методів, що робить код коротшим і зручнішим для читання.

З появою .NET Core C# став повноцінною кросплатформеною мовою. Це дозволяє запускати додатки на Windows, Linux та macOS без модифікації коду. Також .NET Core забезпечує високу продуктивність і дозволяє створювати контейнери для програм, що спрощує їх розгортання в хмарних середовищах, таких як Azure чи AWS.

Маршалізація типів у C# — це процес перетворення даних із керованого середовища .NET у формат, зрозумілий некерованому коду, і навпаки. Вона є критично важливою при взаємодії з нативними бібліотеками, використанні P/Invoke або інтеграції з COM-компонентами. Простий приклад автоматичної маршалізації — передача типів, таких як int або string, до функцій некерованого коду. Однак у складніших випадках, наприклад для структур чи масивів, потрібно вказувати, як саме дані мають бути перетворені. Для цього використовуються атрибути, наприклад MarshalAs. Крім того, клас Marshal у бібліотеці System.Runtime.InteropServices дозволяє виконувати ручну маршалізацію, надаючи інструменти для виділення пам’яті, копіювання даних і керуванння ресурсами. Маршалізація є потужним, але потенційно ризикованим процесом, який потребує точності й розуміння архітектури.

Unsafe-код у C# дозволяє виконувати низькорівневі операції з пам’яттю, такі як прямий доступ до адреси змінної через покажчики. Використання unsafe-коду є необхідним у випадках, коли потрібна максимальна продуктивність або потрібно працювати з апаратним забезпеченням чи некерованими API. Щоб позначити метод або блок коду як небезпечний, використовується ключове слово unsafe. Наприклад, за допомогою вказівників можна змінювати значення змінної безпосередньо в пам'яті. Цей підхід дозволяє уникати накладних витрат, властивих механізмам CLR, але також виходить за межі безпеки, забезпечуваної середовищем .NET. Використання unsafe-коду вимагає обережності та вмикання спеціального прапора /unsafe під час компіляції.

Unmanaged структури складаються лише зі значимих типів даних, таких як int, float або інші unmanaged структури, і не містять посилань на керовані об’єкти. Такі структури можуть бути безпосередньо використані для взаємодії з некерованим кодом. Завдяки цьому вони ідеально підходять для передачі даних у функції, написані на C або C++, або для роботи з нативними API операційних систем. Зберігаючись у стеку, unmanaged структури забезпечують високу продуктивність і просте керування пам’яттю. Прикладом може бути структура, яка представляє координати точки в просторі (Point). Unmanaged структури часто використовуються в P/Invoke, дозволяючи передавати дані у форматі, який розуміє некерований код.

StructLayoutAttribute дозволяє контролювати розташування полів у пам'яті для структур і класів. Це необхідно для забезпечення сумісності між керованим і некерованим кодом, де порядок і вирівнювання полів мають бути строго визначені. Атрибут підтримує три основні режими:

- **LayoutKind.Sequential** — поля розташовуються у пам’яті в порядку їх оголошення.
- **LayoutKind.Explicit** — розробник вказує точні зсуви кожного поля через атрибут FieldOffset.
- **LayoutKind.Auto** — CLR сам вирішує порядок розташування (не використовується для некерованого коду).

Наприклад, для структур, які передаються у функції нативного коду, використання LayoutKind.Sequential забезпечує, що поля будуть інтерпретовані коректно. Якщо потрібно вручну контролювати розташування, використовується LayoutKind.Explicit. Це корисно при роботі з бінарними файлами або апаратними інтерфейсами, де кожен байт даних має конкретне призначення.

1.5 P/Invoke

P/Invoke — це технологія, яка використовується у середовищі .NET для виклику функцій з некерованих бібліотек, написаних на таких мовах, як C або C++. Вона дозволяє керованому коду взаємодіяти з системними API або сторонніми бібліотеками, які не реалізовані на .NET. У контексті розробки ігор або додатків на C#, ця технологія стає необхідною, коли потрібно отримати доступ до низькорівневих функцій операційної системи чи до апаратних ресурсів, які недоступні через стандартні бібліотеки .NET.

Для використання P/Invoke розробник повинен оголосити сигнатуру некерованої функції у вигляді статичного методу в класі C#, використовуючи ключове слово extern і атрибут DllImport. Цей атрибут вказує, з якої бібліотеки буде імпортовано функцію. При виклику такої функції P/Invoke автоматично здійснює маршалізацію даних, перетворюючи типи з керованого середовища .NET у відповідні типи некерованого коду і навпаки.

Важливим аспектом роботи з P/Invoke є правильна маршалізація типів даних. Оскільки .NET і некерований код мають різні способи зберігання даних у пам’яті, неправильно визначені типи можуть призвести до помилок або некоректної роботи програми. P/Invoke надає механізми для явного керування

маршалізацією, такі як специфікація MarshalAs для уточнення, як потрібно обробляти складні типи.

Розділ 2. Опис розробленого програмного продукту

2.1 Постановка задачі

У ході даної роботи було розроблено візуальний редактор для власного ігрового рушія, який забезпечить зручну та інтерактивну роботу з налаштуванням сцени, створенням і керуванням ігровими об’єктами, а також додаванням скриптової логіки.

Основні функціональні можливості Редактора:

1. Інтерфейс для керування об’єктами сцени:
    - Інтуїтивна панель для створення, видалення та керування ігровими об'єктами.
    - Підтримка ієрархічної структури об’єктів (наприклад, створення батьківських і дочірніх об’єктів).
    - Панель властивостей, де можна змінювати параметри об’єкта (позиція, обертання, масштаб, компоненти тощо).
2. Редагування компонентів об’єктів:
    - Можливість додавання або видалення компонентів об’єктів через редактор (наприклад, додавання спрайта, фізичного тіла, джерела звуку).
    - Панель властивостей компонентів для налаштування параметрів (наприклад, вибір текстури для спрайта, налаштування маси для фізичного тіла, вибір звукового файлу для джерела звуку).
    - Візуалізація та редагування анімацій через редактор анімаційних кліпів.
3. Візуальний редактор сцени:
    - Поле сцени з підтримкою функцій перетягування (drag-and-drop) для розміщення об'єктів.
    - Підтримка масштабування та навігації по сцені.
    - Інтерактивне редагування об'єктів: переміщення, обертання та масштабування об'єктів у режимі реального часу.
4. Редагування користувацького інтерфейсу (UI):
    - Інструмент для створення та налаштування елементів інтерфейсу (кнопок, текстових полів, вікон).
    - Підтримка прив’язки подій до елементів UI, таких як кліки або введення тексту.
    - Можливість налаштування стилів і розташування елементів інтерфейсу.
5. Менеджер ресурсів:
    - Система перегляду та керування графічними та аудіоресурсами.

- Можливість додавання нових ресурсів до проєкту (текстури, аудіофайли, анімації).
- Панель попереднього перегляду ресурсів з можливістю швидкого тестування (наприклад, попередній перегляд текстури або прослуховування звукового файлу).

6. Система тестування та запуску сцени:
   - Можливість запуску сцени безпосередньо з редактора для швидкого тестування ігрових механік.
   - Інтерактивна панель для налагодження (дебагінгу) з інформацією про активні об'єкти, значення змінних та інші важливі параметри.
   - Пауза, відновлення та повний перезапуск сцени під час тестування.
7. Редактор анімацій:
   - Інструмент для створення і редагування анімаційних кліпів. Підтримка налаштування тривалості анімацій, порядку кадрів і циклічності.
   - Візуалізація кадрів анімації та можливість ручного налаштування їхніх параметрів.

2.2 Опис редактора

Панель проєктів (Projects Panel): Ця панель призначена для керування проєктами у програмі. Вона відображає список доступних проєктів із зазначенням їхніх назв, шляхів до файлів і дати останнього редагування. Інтерфейс дозволяє швидко вибрати проєкт для відкриття або створення нового.

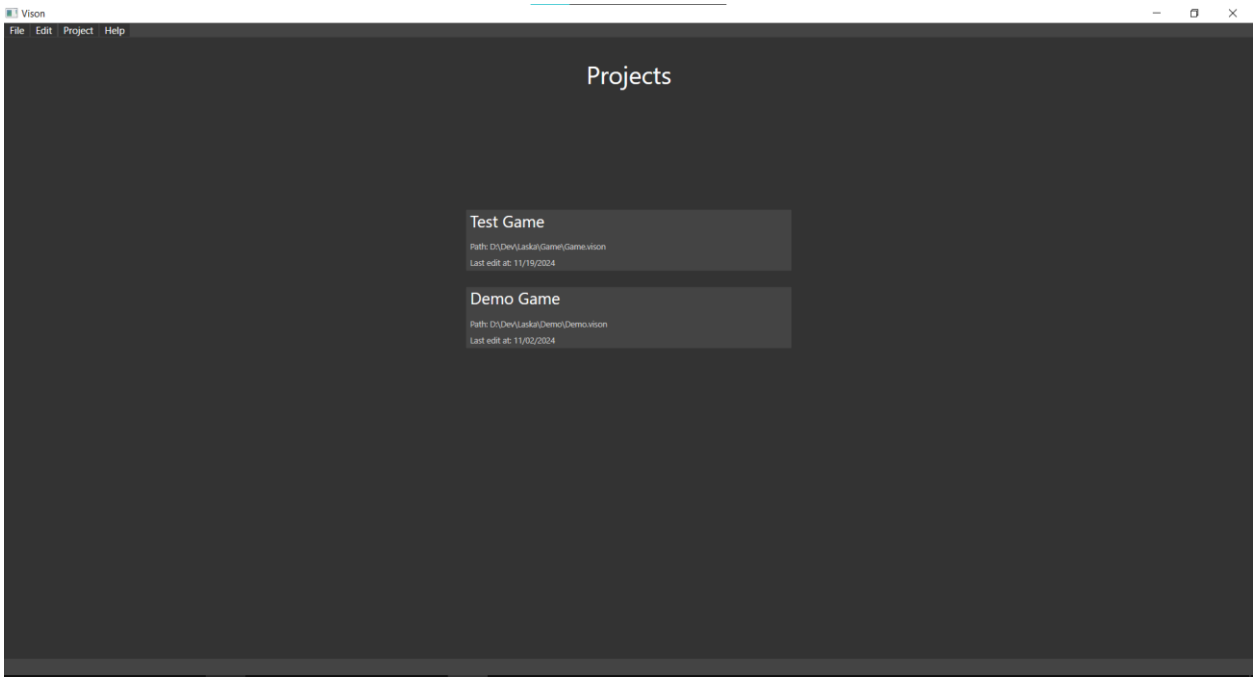

Рисунок 2.2.1 – Список проєктів

Панель ієрархії (Hierarchy Panel): У цій панелі відображаються всі об’єкти сцени у вигляді списку. Панель дозволяє швидко вибирати сутності, редагувати їх і змінювати їхні властивості. Додатково, можна легко організовувати об’єкти, додаючи нові компоненти для розширення їх функціоналу.

Панель редактора компонентів (Component Editor Panel): Ця панель дозволяє змінювати параметри в компонентах обраної сутності, додавати та видаляти компоненти.

Панель сцени (Scene Panel): Панель сцени є робочою областю, де відображаються всі об'єкти поточної сцени. Тут можна переміщати сутності, обертати їх та змінювати масштаб за допомогою миші. Координатна сітка допомагає з точним розташуванням об’єктів. Панель сцени також надає можливість спостерігати за змінами в реальному часі.

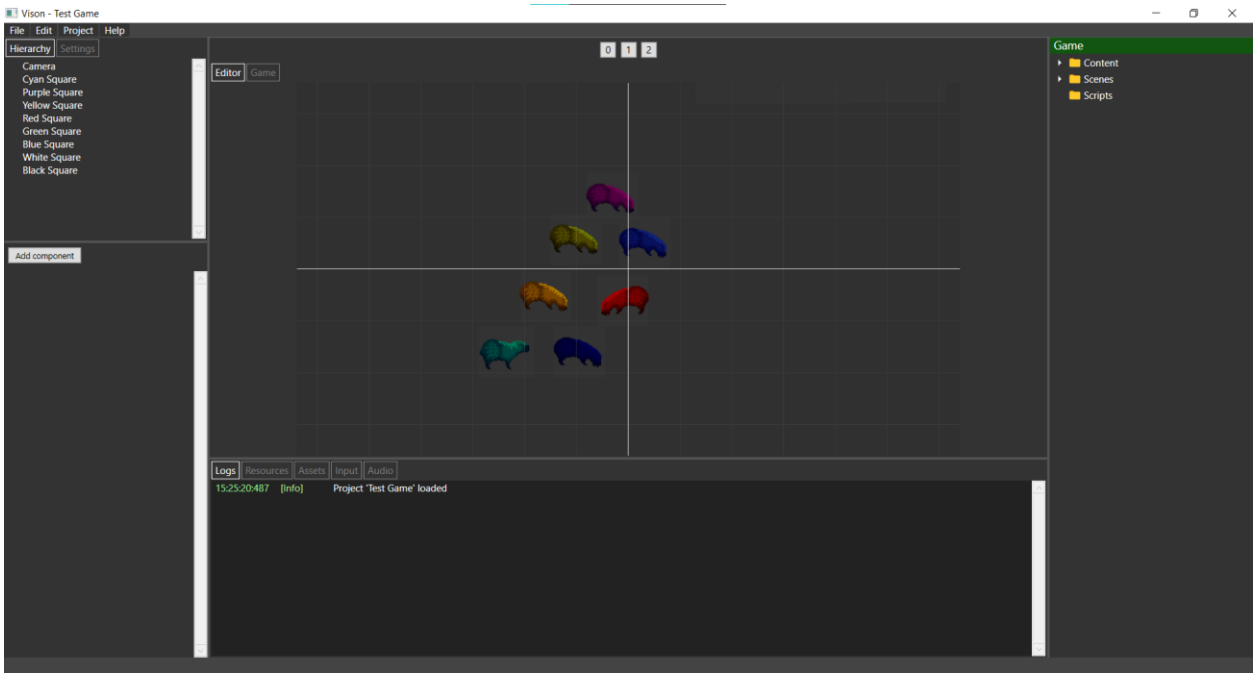

Рисунок 3.2.2 – Вікно редактора

Панель гри (Game Panel): У цій панелі відображається структура проєкту, що включає три основні розділи: *Content*, *Scenes* і *Scripts*. Вона дозволяє керувати ресурсами проєкту, створювати нові сцени, видаляти існуючі, керувати скриптами (C#, C++).

Панель логів (Log Panel): Панель логів призначена для сповіщень від редактора рушія, системи скриптів тощо.

Розділ 3. Опис архітектури

3.1. Зміни в рушії

3.1.1 Реєстр

У процесі розробки були внесені значні зміни в архутектурі реєстра. Було впроваджено систему архетипів. Архетипи визначають групи сутностей з однаковими наборами компонентів. У кожному архетипі компоненти зберігаються у вигляді сторінкових списків (PageList), які забезпечують компактне використання пам’яті та швидкий доступ до даних.

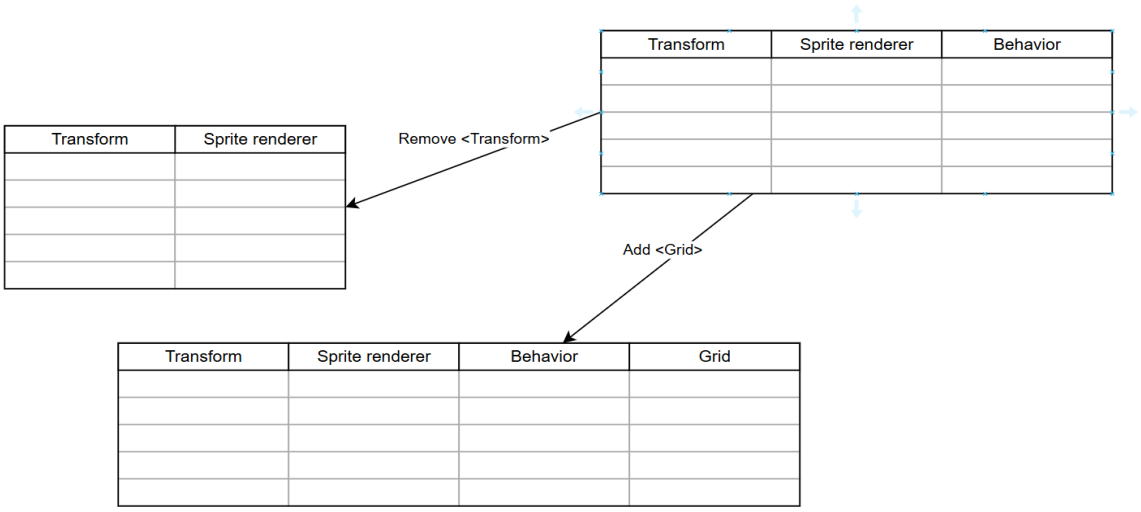

Рисунок 3.1.1 – Приклад прототипів

Сутності представлені структурою EntityInfo, яка містить унікальний ідентифікатор, теги, перемикачі, інформацію про архетип, до якого належить сутність, і посилання на дані її компонентіи. Дані про всі сутності зберігаються в масиві, організованому у вигляді списку.

Додавання компонентів до сутності є складним процесом. Реєстр перевіряє, чи існує архетип з оновленим набором компонентів. Якщо такого архетипу немає, він створюється динамічно. Дані сутності копіюються з її старого архетипу до нового. Для нового компонента виділяється місце в пам'яті, й ініціалізуються його початкові значення. Після цього сутність переноситься до нового архетипу, а посилання на старий архетип оновлюються. Реєстр також викликає обробники подій, які повідомляють інші системи про зміну стану сутності.

Видалення компонентів працює аналогічно, але зі зворотним ефектом. Якщо сутність втрачає компонент, реєстр шукає або створює архетип із зменшеним набором компонентів. Дані компонента, що видаляється, звільняються, а інші компоненти копіюються у новий архетип. Сутність оновлює свій зв’язок із новим архетипом, а реєстр викликає обробники подій, які повідомляють про вилучення компонентів.

Для обходу сутностей, які відповідають заданому набору компонентів, використовується ітератор (RegistryIterator). Ітератор створюється, коли потрібно обробити всі сутності, що мають певний набір компонентів. Під час ініціалізації ітератор перевіряє всі архетипи, визначаючи, які з них містять потрібні компоненти. Коли знайдено відповідний архетип, ітератор переходить до його сутностей і послідовно обробляє їх. Для кожної сутності ітератор надає доступ до даних її компонентів, використовуючи мапу компонентів архетипу, яка дозволяє швидко знайти потрібну інформацію у сторінкових списках. Ітератор також підтримує можливість повторного обходу сутностей, що зручно для багатократної обробки тих самих даних у різних контекстах. Крім того, ітератор може працювати у зв'язці з функціями зворотного виклику, які застосовуються до кожної сутності, забезпечуючи реакцію счистем на події повязані із сутностями.

Реєстр також включає функціональність для збереження і відновлення стану через знімки (Snapshot), що дає змогу зберігати весь стан ECS у компактному вигляді і відновлювати його у будь-який момент. Це особливо важливо для ігор, які потребують можливості збереження прогресу, а також для використання в редакторі при тестуванні гри. Після тесту можна дуже легко повернути сцену до вихідного стану.

3.1.2 В'юпорти

Система в'юпортів у цьому проєкті забезпечує створення та керування окремими областями відображення, які можуть використовуватися як для виведення графіки на екран, так і для обчислень поза межами екрану. В'юпорт виступає абстракцією, що підтримує два основних типи:

- віконний - прив’язаний до вікна
- позавіконний - результати зберігаються у вигляді піксельних даних

Ініціалізація системи в'юпортів відбувається через інтеграцію з іншими модулями. Наприклад, якщо в'юпорт має тип "вікно", то він створюється за допомогою WindowRegistry. При цьому налаштовуються такі параметри, як

прив’язка до подій, використання вертикальної синхронізації та доступ до черг подій. Якщо в'юпорт має тип "позавіконний", то створюється лише внутрішня структура без візуального виводу, оскільки такі в'юпорти зазвичай використовуються для рендерингу в текстури.

3.1.3 Графічні API

Система абстракцій над графічними API побудована для забезпечення універсального інтерфейсу роботи з графічними технологіями, такими як Vulkan, Direct3D чи OpenGL. Її основна мета — приховати специфіку реалізації окремих API, надаючи єдиний стандарт взаємодії. Це дозволяє легко замінювати або комбінувати різні графічні технології без внесення значних змін у структуру проєкту.

Ключовим елементом системи є базовий клас Graphics, який визначає основні методи роботи з графікою, такі як Init, Dispose, CreateInstance та GetInstance. Цей клас забезпечує створення графічного контексту, керування його життєвим циклом і надає універсальний спосіб отримання доступу до екземпляру графічного API.

3.1.4 Рендеринг

Нова система побудована навколо концепції рендер-графу, який складається з вузлів (RenderGraphNode). Кожен вузол представляє конкретну задачу, наприклад рендеринг певного виду об'єктів або обробку окремої області екрану. Граф дозволяє чітко впорядкувати та оптимізувати всі процеси, включаючи обробку текстур, геометрії, і виконання шейдерів. Для кожного вузла створюються відповідні командні буфери, що забезпечує плавність і паралельність виконання задач. Вузол включає наступні поля:

- Type: визначає тип вузла (RenderNodeType), наприклад, вузол для відображення на екран або для рендерингу в текстуру.
- Camera: указує на камеру, пов'язану з цим вузлом.
- Viewport: посилання на область відображення, яка визначає розмір і місце виводу.
- Groups: список груп пайплайнів (PipelineGroup), які виконують основну частину рендерингу.
- CommandBuffer: буфер команд, що зберігає інструкції для GPU.
- RenderTargetDescriptor та Framebuffer: визначають, куди саме рендеряться результати (наприклад, екран чи текстура).

- Dependencies та WaitSemaphores: керують синхронізацією між вузлами графа.
- SwapChain та Surface: забезпечують рендеринг для віконних вузлів (типу "Present").
- UniformBuffer та Uniforms: дані для шейдерів, наприклад, матриці трансформацій або параметри камери.

PipelineGroup - ця структура організовує роботу з пайплайнами, які визначають, як обробляти дані для рендерингу:

- Pipeline: указує на сам пайплайн (IPipeline), який містить шейдери та їх конфігурації.
- UniformSet: набір ресурсів для шейдерів, таких як уніформ-буфери та текстури.
- Steps: список рендер-етапів (PipelineStep), які деталізують операції малювання в рамках цієї групи.

PipelineStep - ця структура описує конкретний рендер-етап у пайплайні:

- Mode: режим рендерингу (RenderStepMode), наприклад, рендеринг вершин чи індексів.
- TextureSet: текстурний набір (SetRef), який використовується в цьому етапі.
- IndexBuffer, InstanceBuffer, VertexBuffer: буфери для геометрії, екземплярів та індексів, які використовуються в рендері.
- InstanceCount, VertexCount, IndexCount: кількість оброблюваних екземплярів, вершин та індексів.
- FirstInstance, FirstVertex, FirstIndex: початкові зсуви для даних, які рендеряться.

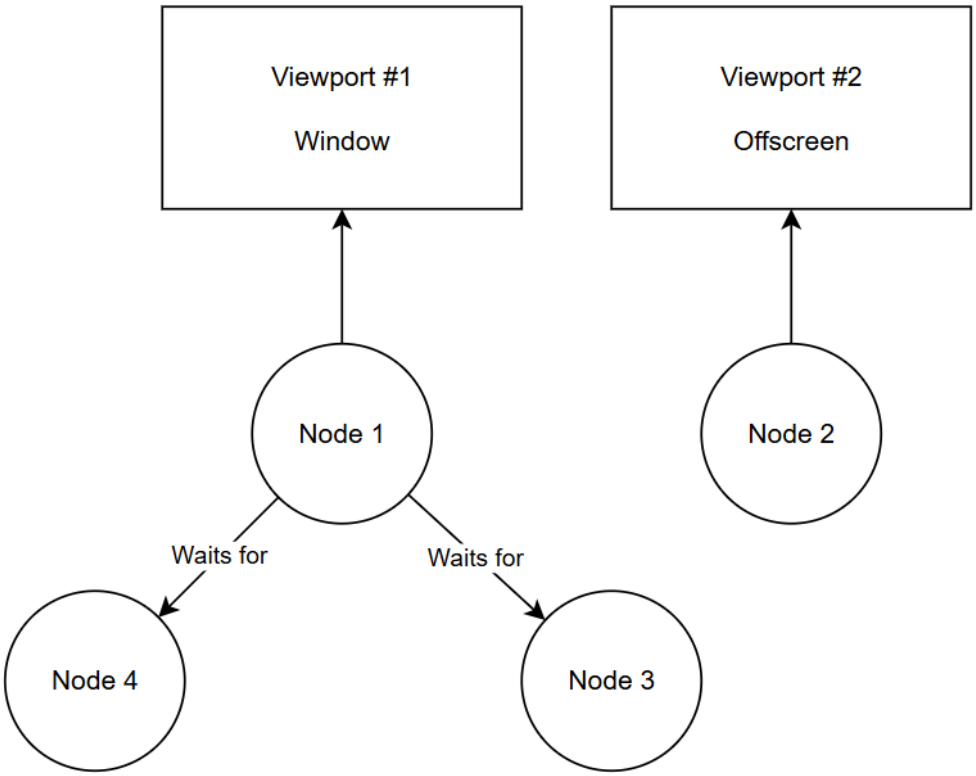

Рисунок 3.1.3 – Схема Render Graph

Важливою частиною рендерера є підтримка батчів – об’єднання об’єктів у групи для одночасного рендерингу. Це дозволяє значно зменшити кількість викликів до GPU, що позитивно впливає на продуктивність. Крім цього, реалізовано динамічне керування текстурами через систему наборів текстур. Вона забезпечує завантаження та оновлення тільки тих текстур, які використовуються в даний момент, зберігаючи при цьому мінімальні затрати пам’яті.

Для промальовування використовується розділення на кроки рендерингу, які визначають послідовність обробки даних. Наприклад, для об’єкта можуть бути виконані кроки підготовки геометрії, прив’язки текстур та запуску шейдерів. Також введено механізм синхронізації між етапами через використання семафорів і бар'єрів, що запобігає конфліктам у доступі до ресурсів..

На початку роботи рендерера відбувається етап ініціалізації, під час якого завантажуються необхідні ресурси: графічні шейдери, текстури, буфери та пайплайни. Також формується рендер-граф, який визначає, як і в якій послідовності будуть оброблятися всі елементи сцени.

У момент початку сцени викликається метод BeginScene, який налаштовує рендерер для роботи з конкретною камерою та об'єктами сцени. Після цього гра починає додавати об'єкти до рендерингу. Кожен об'єкт проходить через процес групування за типом шейдерів і текстур. Це дозволяє оптимізувати кількість викликів до GPU, об'єднуючи об'єкти з однаковими характеристиками в батчі.

Дані об'єктів зберігаються у відповідних буферах: VertexBuffer, InstanceBuffer, IndexBuffer. Буфери оновлюються тільки тоді, коли об'єкти змінюються, що економить ресурси системи. Якщо текстура використовується вперше, рендерер додає її до активного набору текстур, перевіряючи, чи є вільні місця для нових текстур. Якщо набір переповнений, відбувається примусове виконання промальовування поточної групи.

Після завершення всіх підготовчих операцій рендерер переходить до виконання рендер-графу. Командні буфери на строні GPU створюються для кожного вузла, записуючи всі необхідні операції: налаштування ресурсів, виконання шейдерів, передачу даних.

На фінальному етапі відбувається виведення кадру на екран. Для цього використовується команда представлення, яка виводить готове зображення на екран або у визначений буфер. Після цього рендерер завершує роботу сцени викликом методу EndScene, очищуючи всі тимчасові дані та підготовлюючи систему до наступного кадру.

3.1.5 Ресурси

У попередній реалізації система ресурсів базувалася на простому списку файлів, що використовувались для завантаження та відвантаження ресурсів у міру потреби. Така структура мала низьку гнучкість і не дозволяла ефективно керувати ресурсами при роботі з великими проєктами. Нова архітектура, заснована на концепції *банків ресурсів* надає кращу продуктивність.

Система банків ресурсів включає три основні типи банків: *файловий банк*, *банк у пам'яті* та *мережевий банк*. Банки виступають як абстракція для керування ресурсами, кожен зі своєю унікальною функціональністю.

1. Файловий банк **(**File Bank**)**
    - Опис: Файловий банк є абстракцією, що представляє бінарний файл, у якому зберігаються закодовані ресурси. Замість зберігання ресурсів у розрізнених окремих файлах, всі ресурси збираються і упаковуються у єдиний бінарний файл.

- Принцип роботи: При завантаженні ресурсу система звертається до файлу банку, зчитує необхідні дані і декодує ресурс для подальшого використання.

2. Банк у пам’яті (In-Memory Bank)
   - Опис: Банк у пам'яті є абстракцією для зберігання ресурсів безпосередньо в оперативній пам’яті. Це ідеально підходить для часто використовуваних ресурсів, таких як текстури, звукові ефекти або моделі.
   - Принцип роботи: Ресурси завантажуються і залишаються в оперативній пам’яті, забезпечуючи миттєвий доступ без затримок. Це дозволяє уникати повторного завантаження з диска чи мережі.
3. Мережевий банк (Network Bank)
   - Опис: Мережевий банк є абстракцією для завантаження ресурсів через мережу. Це корисно для онлайн-ігор або додатків, де ресурси зберігаються на віддалених серверах.
   - Принцип роботи: При зверненні до мережевого банку ресурс завантажується з віддаленого сервера і може бути кешований у пам’яті або збережений у локальному файловому банку. Завантаження відбувається асинхронно, щоб не блокувати основний потік гри.

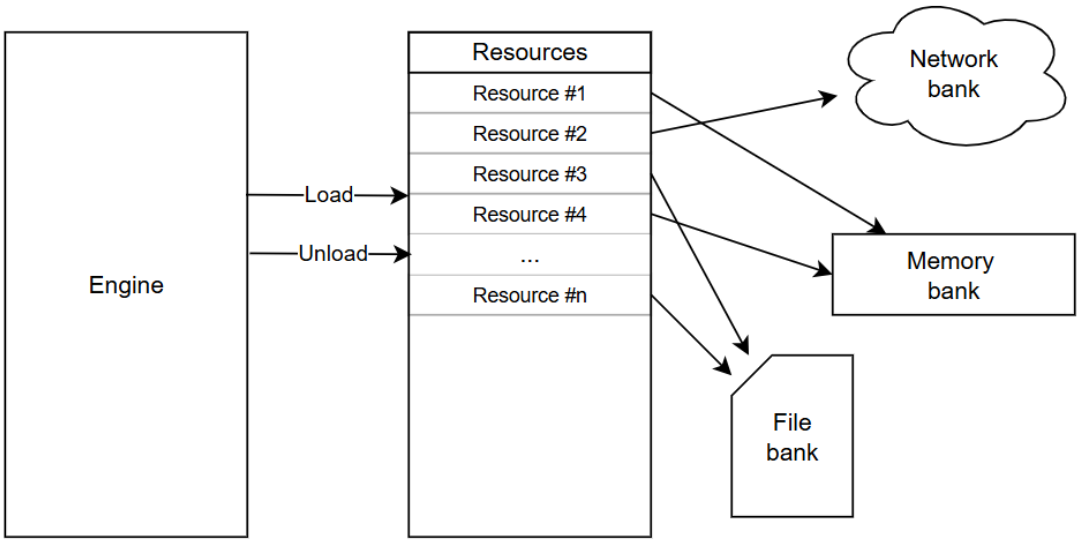

Рисунок 3.1.3 – Система ресурсів

3.1.6 .NET

У новій версії бекенду .NET реалізовано API для роботи з полями та масивами, що розширює функціонал. Завдяки цьому API можна працювати з полями як базових, так і складних типів, а також легко маніпулювати масивами.

Нові методи SetField і GetField забезпечують зручну роботу з полями скриптів.

1. GetField - дозволяє отримувати значення полів з об'єкта, використовуючи універсальний тип Any, що абстрагує реальний тип даних.
2. SetField - у свою чергу, забезпечує встановлення значень у поля об'єктів, включаючи складні типи. Зокрема, це підтримка компонентів, таких як Sprite, Animator, Transform та інших.

Процес мапінгу значення з типу Any у реальний тип поля скрипта реалізовано через функцію MapToObject. Основна ідея функції полягає в тому, щоб визначити, як обробляти дані на основі значення BehaviorFieldDataType.

Для простих типів, таких як int, float, bool, значення безпосередньо конвертується у відповідний базовий тип. Наприклад, якщо BehaviorFieldDataType дорівнює Int, функція повертає значення data->Value.Int.

При роботі зі складними типами, такими як Vector2, Vector3 або Matrix4x4, функція зчитує необхідні дані зі структури BehaviorFieldData, конвертуючи їх у формат, який підходить для використання у скриптах.

Для компонентів (SpriteRenderer, Transform, Animator тощо) логіка передбачає створення або використання існуючих об'єктів відповідних класів обгорток. Їхній стан зв’язується з вказівником, отриманим із BehaviorFieldData.

Додатково функція враховує специфіку роботи з масивами, дозволяючи динамічно отримувати або створювати об'єкти для кожного індексу масиву.

Зворотний мапінг, тобто конвертація реального типу поля в Any, реалізовано через функцію MapToAny. Вона відповідає за перетворення значень полів об'єкта у формат Any, який може бути переданий у систему.

Для числових типів (int, float, double) функція напряму заповнює відповідні поля структури Any (наприклад, result.Int, result.Float).

При роботі зі складними типами, такими як вектори (Vector2, Vector3) або матриці (Matrix4x4), значення конвертуються у відповідний формат, наприклад result.Float2 або result.Float3.

Якщо поле є компонентом гри, наприклад RigidBody або SpriteRenderer, функція додає вказівник на цей компонент у поле result.IntPtr.

Для роботи з масивами додано методи GetArrayInfo, GetFieldArray і SetFieldArray:

1. GetArrayInfo дозволяє отримувати метаінформацію про масив, наприклад його розмір.
2. GetFieldArray забезпечує доступ до елементів масиву за індексом.
3. SetFieldArray дозволяє змінювати значення елементів.

3.2 Архітектура редактора

3.2.1 Контекст проєкту

Контекст проєкту – контекст який керує всіма підконтекстами запущеного проєкту. Він має наступні підконтексти:

1. SceneContext (Контекст сцени): Відповідає за керування станом сцени, її об'єктами, системами та режимами роботи. Він забезпечує організацію життєвого циклу сутностей, дозволяючи перемикатися між режимами, такими як редагування або ігровий процес, і викликає необхідні системи та підмодулі для їх виконання.
2. SelectionContext (Контекст вибору): Керує вибором об’єктів на сцені, зберігаючи активні сутності, обрані користувачем. Він надає інтерфейси для обробки взаємодій, таких як переміщення, обертання або видалення об’єктів, і тісно інтегрується з інструментами редагування.
3. ResourceManager (Менеджер ресурсів): Забезпечує керування ресурсами проєкту, такими як текстури, моделі та звукові файли. Він працює з банками ресурсів (файловими, пам'яттєвими та мережевими), оптимізуючи доступ і завантаження ресурсів у пам'ять або кеш.
4. AssetManager (Менеджер активів): Організовує активи проєкту, включаючи метадані, які описують ресурси, та їх зв’язки з фізичними файлами. Він інтегрується з ResourceManager, відстежуючи залежності між активами і забезпечуючи їх доступність для редагування або виконання.
5. ImageOverlay (Накладання зображень): Відповідає за накладання візуальних елементів поверх сцени або інтерфейсу редактора. Він використовується для відображення допоміжних сіток, маніпуляторів, відлагоджувальних елементів та інших графічних підказок, які полегшують редагування.
6. EditorToolBox (Панель інструментів редактора): Надає набір інструментів для редагування об’єктів сцени, таких як переміщення, обертання, масштабування, малювання та інші операції. Він забезпечує зручний інтерфейс для роботи з обраними об’єктами через SelectionContext та ImageOverlay.
7. DotnetHost (Хост .NET): Виконує користувацький код, написаний на .NET, забезпечуючи інтеграцію між логікою проєкту та рушієм. Він компілює та запускає скрипти в реальному часі, обробляючи помилки та передаючи дані між рушієм і кодом.
8. EngineHost (Хост рушія): Є ядром рушія, яке відповідає за запуск ігрового циклу, виконання основних систем, таких як рендеринг, фізика та звук, а

також керування взаємодією між сценою, ресурсами та інструментами редактора.

3.2.2 Контекст сцени

Контекст сцени (SceneContext) виконує роль основного модуля для керування станом і процесами, що стосуються поточної сцени. Він керує данними про сутності, компоненти, ресурси та вибрані об’єкти. Контекст сцени дозволяє створювати, видаляти та змінювати сутності, додаючи або прибираючи компоненти, та відстежує їх зв'язки, зберігаючи інформацію про них у структурованому вигляді. Крім того, SceneContext забезпечує синхронізацію між вибором об’єктів у редакторі та їхнім станом у сцені, підтримуючи повну синхронізацю між двома доменами. Він відповідає за завантаження та ініціалізацію сцени з файлів, обробку архетипів і компонентів, а також підготовку всіх даних до виконання.

3.2.3 Контекст вибору

SelectionContext — це модуль, який керує вибором об'єктів у сцені. Він зберігає список вибраних сутностей і дозволяє змінювати або очищати вибір. Якщо вибір змінюється, викликається подія SelectionChanged, яка повідомляє інші системи про оновлення. Через властивість Selection можна отримати поточний вибір або встановити новий. Є також метод Clear, який швидко скидає вибір.

3.2.4 Контексти ресурсів та активів

AssetManager відповідає за керування активами проєкту, такими як текстури, моделі, аудіофайли чи шейдери. Він забезпечує створення, видалення та отримання активів на основі їхніх унікальних ідентифікаторів (UUID), а також зберігання метаданих у файлі активів (AssetsFile). Кожен активі створюється через низькорівневий інтерфейс (AssetManagerRaw) і додається до внутрішнього списку об'єктів, де він може бути доступний для подальшого використання іншими частинами рушія. AssetManager також дозволяє отримувати інформацію про активи, наприклад їхній тип, ім'я чи метадані, через спеціальні методи доступу. При створенні нового активу його дані додаються у файл активів, що забезпечує їхнє збереження навіть після завершення роботи програми. Завдяки події OnEvent система відстежує зміни стану активів, наприклад їхнє створення чи видалення, що дозволяє синхронізувати ці дії з іншими компонентами рушія.

ResourceManager — це модуль, який керує екземплярами ресурсів у проєкті, такими як текстури, моделі чи звукові дані, забезпечуючи їхнє завантаження у пам’ять або вивантаження для звільнення ресурсів. На відміну від AssetManager, який зберігає лише метаінформацію про активи, ResourceManager працює безпосередньо з фізичними даними ресурсів. Ресурси можуть бути створені через низькорівневий інтерфейс (ResourceManagerRaw), а також завантажені з банків або файлів ресурсів (ResourcesFile). При створенні або завантаженні ресурсу інформація про нього зберігається у внутрішньому списку, а доступ до даних забезпечується через унікальні ідентифікатори. ResourceManager підтримує різні типи ресурсів і джерел, зокрема ресурси у вигляді файлів, у пам’яті або завантажені з мережі. Крім того, цей модуль дозволяє динамічно змінювати стан ресурсу, наприклад перевантажувати текстури або змінювати їхній режим використання, інтегруючись з іншими підсистемами рушія.

3.2.5 Система оверлею

Система оверлею — це ключовий компонент редактора, який відповідає за відображення графічних накладень поверх сцени. Її основна мета — забезпечити користувачеві зручний і зрозумілий інтерфейс для роботи з об'єктами та інструментами редагування. Вона відображає такі елементи, як маніпулятори для переміщення, обертання чи масштабування, а також допоміжні сітки, полігони та текстури. Завдяки інтеграції з камерою редактора (EditorCamera) оверлей автоматично адаптується до змін масштабу та зсуву, забезпечуючи точне позиціонування накладень у сцені.

Система оверлею складається з кількох важливих компонентів. ImageOverlay керує накладеннями, використовуючи набір інструментів із ToolBox. Цей клас викликає методи малювання для активних інструментів і забезпечує їхню синхронізацію з поточним станом сцени. Графічні елементи рендеряться через OverlayDrawer, який відповідає за створення базових форм, таких як лінії, прямокутники, кола чи текстури. Додатково, у системі використовується логічний буфер (_inkBuffer), який дозволяє відслідковувати стан накладень, наприклад, виділення об'єктів або активні області взаємодії.

Користувач може взаємодіяти з оверлеєм через різні режими роботи, які задаються в ToolBoxMode. Наприклад, у режимі переміщення (Move) активуються відповідні інструменти, що дозволяють змінювати положення об'єктів у сцені, а в режимі роботи з UV-мапами (UVs) користувач може редагувати текстури. Режими легко перемикаються через гарячі клавіші, такі як M, S або R, забезпечуючи швидкий доступ до необхідних функцій. Сітка

(GridTool) також може вмикатися або вимикатися залежно від потреб користувача, що допомагає при вирівнюванні об'єктів.

Наведемо список інструментів, реалізованих у редакторі:

PolygonTool — це інструмент редактора, призначений для роботи з багатокутниками у сцені. Він дозволяє створювати, редагувати та переміщати вершини багатокутника, забезпечуючи точний контроль над геометрією. Інструмент відображає багатокутник на сцені, дозволяючи взаємодіяти з його вершинами за допомогою миші.

Коли користувач наводить курсор на вершину багатокутника, інструмент визначає, яку саме вершину обрано, використовуючи логічний буфер (`ink`) для відстеження стану. Якщо вершина активована, вона виділяється спеціальним кольором, а під час перетягування використовується інший колір. При натисканні лівої кнопки миші вершина фіксується, і її положення можна змінити, переміщуючи мишу. Координати вершини оновлюються відповідно до переміщення, враховуючи масштаб та зсув камери редактора.

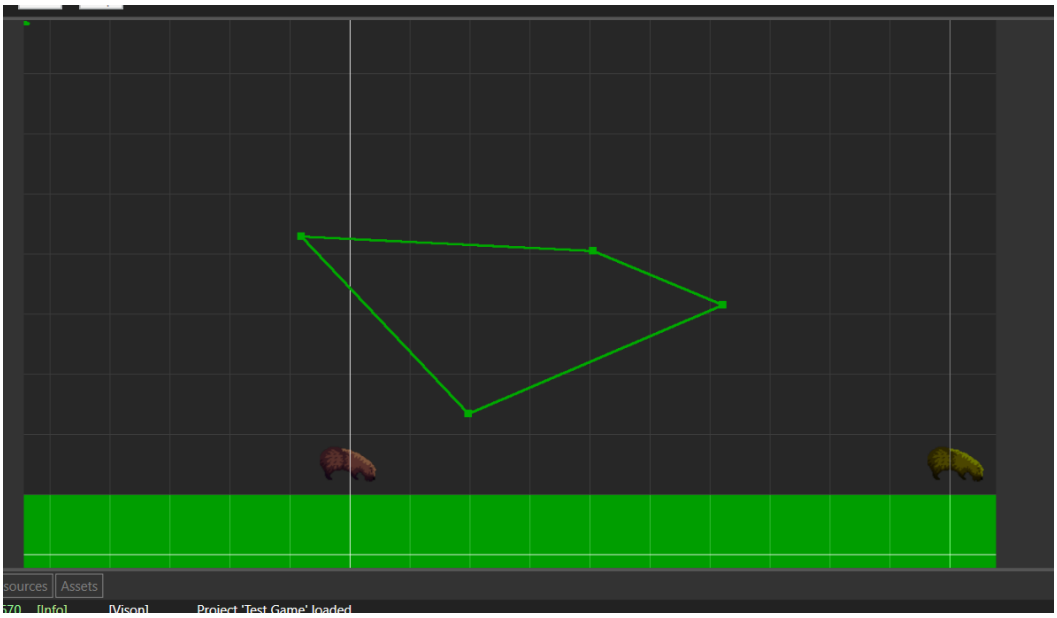

Рисунок 3.2.1 – Polygon tool

Для візуалізації `PolygonTool` малює багатокутник і його вершини, використовуючи методи `OverlayDrawer`. Лінії між вершинами малюються для підкреслення структури, а самі вершини відображаються як маленькі прямокутники, які реагують на взаємодії.

GridTool — це інструмент редактора, який відповідає за відображення сітки на сцені. Сітка допомагає користувачам орієнтуватися в просторі сцени, вирівнювати об'єкти та точніше позиціонувати їх. Інструмент автоматично адаптується до масштабу сцени та зсуву камери, що забезпечує його відповідність до поточного вигляду сцени.

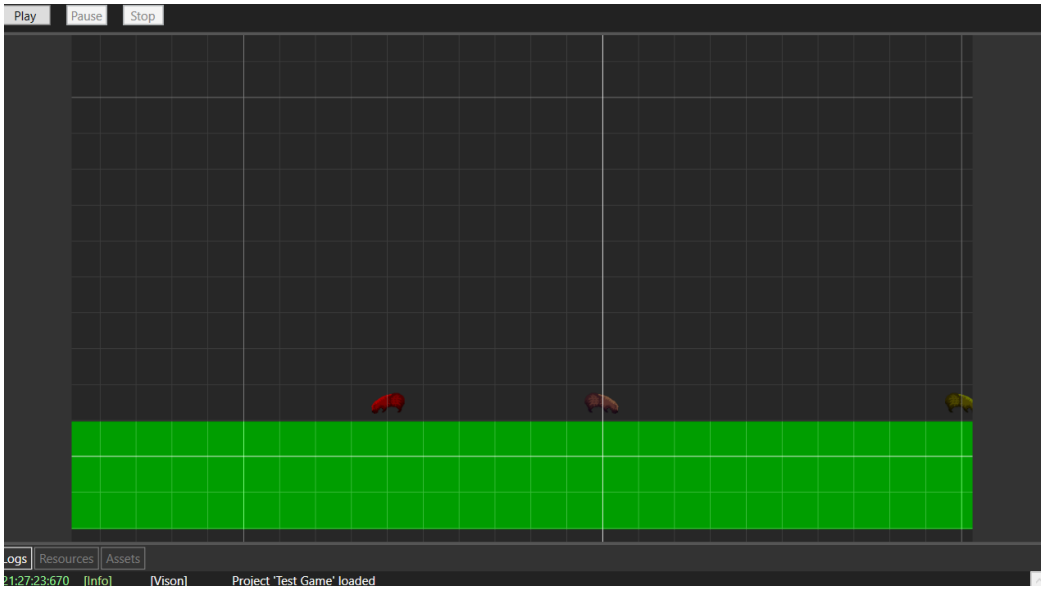

Рисунок 3.2.2 – Grid tool

Основний метод Draw викликається для рендерингу сітки. Він розраховує розміри та позиції сіткових ліній на основі поточного масштабу (Scale) та зміщення (X, Y) камери, переданих через OverlayTransform. GridTool малює три рівні сітки: дрібну, середню та велику, кожна з яких має свій колір для чіткої візуалізації. Дрібна сітка допомагає працювати з деталями, середня використовується для загального вирівнювання, а велика підкреслює основну структуру простору.

MoveTool — це інструмент редактора, призначений для переміщення об'єктів у сцені. Він надає користувачеві можливість змінювати позицію вибраного об’єкта за допомогою маніпуляторів на осях X, Y або одночасно по обох осях через зону захоплення (пад). Інструмент автоматично визначає взаємодію з користувачем за допомогою курсору миші та адаптує свої дії залежно від положення миші та активної осі.

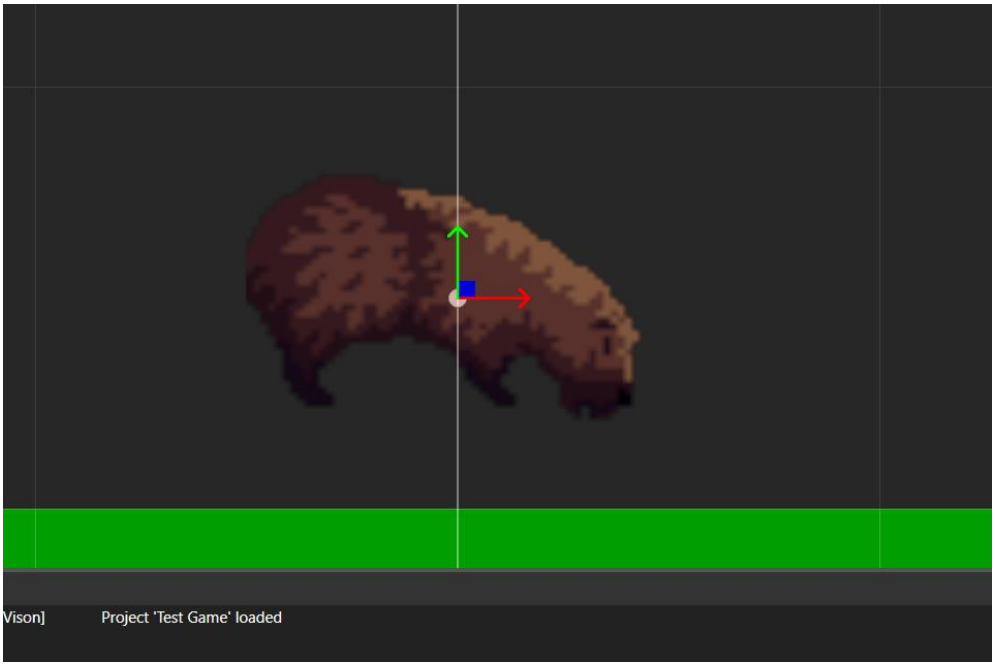

Рисунок 3.2.3 – Move tool

Інструмент малює маніпулятор у вигляді стрілок для осей X і Y та прямокутника для зони захоплення по обох осях (Z-пад). Стрілки та зона захоплення підсвічуються різними кольорами при наведенні миші, що дозволяє чітко розуміти, яка дія буде виконана. Кольори оновлюються в реальному часі, а курсор змінюється, відображаючи напрямок переміщення.

Коли користувач захоплює маніпулятор, наприклад, натискаючи на стрілку X, MoveTool починає обробляти переміщення об’єкта. Інструмент враховує зміщення миші від початкової точки захоплення та масштабу камери (EditorCamera) для точного розрахунку нової позиції. Якщо обрано Z-пад, переміщення виконується одночасно по обох осях. Після завершення переміщення (відпускання миші) MoveTool фіксує зміни та оновлює прив’язані властивості об’єкта.

Маніпулятор адаптується до змін масштабу та зсуву сцени, що дозволяє завжди підтримувати його пропорційність та зручність використання. Інструмент також інтегрується з контекстом вибору (SelectionContext) і може автоматично перемикатися в режим виділення, якщо об'єкт не активний.

ScaleTool — це інструмент для редагування сцени, який дозволяє змінювати масштаб об’єктів по осях X, Y в обох напрямках. Він надає користувачеві візуальні маніпулятори для взаємодії, включаючи стрілки для

кожної осі та прямокутну зону для комбінованого масштабування. Інструмент інтегрується з OverlayDrawer, що дозволяє відображати маніпулятори з урахуванням масштабу та зсуву камери редактора.

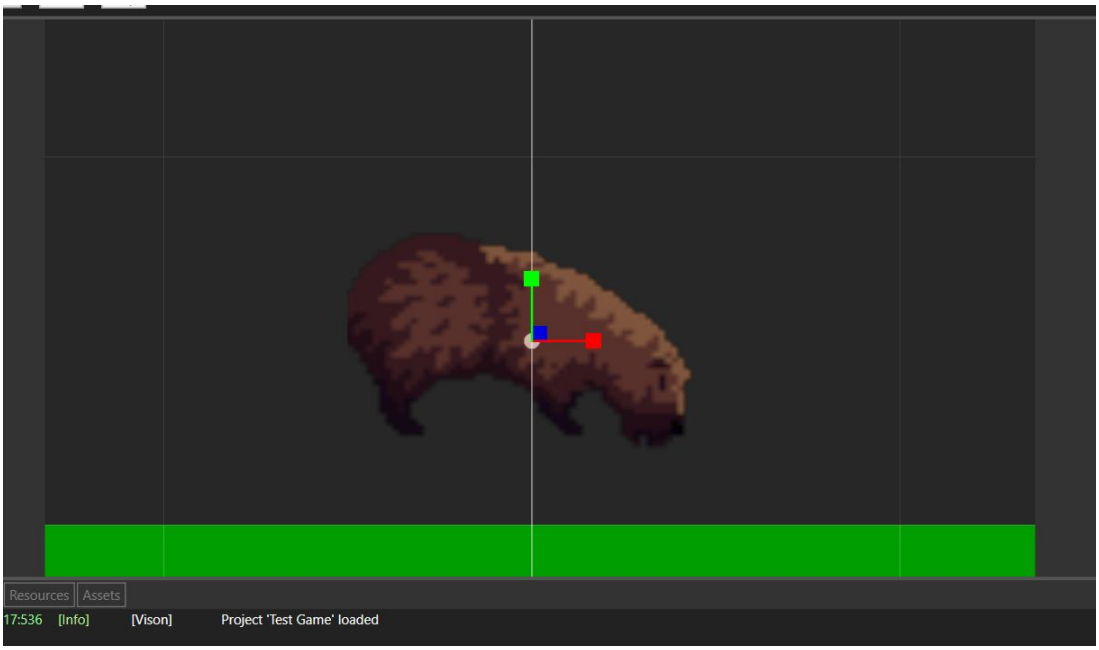

Рисунок 3.2.4 – Scale tool

Під час роботи з мишею ScaleTool визначає, на якому маніпуляторі наведено курсор, використовуючи логічний буфер. Активні області підсвічуються різними кольорами: червоним для осі X, зеленим для осі Y та синім для комбінованої області (Z-пад). При натисканні лівої кнопки миші інструмент фіксує початкові координати курсору та масштабу об’єкта, що дозволяє точно змінювати розміри об’єкта залежно від переміщення миші.

Малювання маніпуляторів виконується з урахуванням зсуву та масштабу сцени. Стрілки осей та прямокутник для комбінованого масштабування малюються з відповідними кольорами. Інструмент також реагує на зміну масштабу та положення камери, адаптуючи відображення маніпуляторів для підтримання їх пропорційності.

RotateTool — це інструмент редактора, який використовується для обертання об’єктів у сцені навколо осі Z. Інструмент надає візуальний маніпулятор у вигляді кільця, що дозволяє користувачеві інтуїтивно зрозуміти, як буде виконуватись обертання. При наведенні курсору на активну область маніпулятор підсвічується, використовуючи логічний буфер для відстеження взаємодій.

Взаємодія з RotateTool відбувається за допомогою миші. Під час натискання на маніпулятор інструмент зберігає початкову позицію курсору і кут обертання об’єкта. Потім, при переміщенні миші, обчислюється новий кут, заснований на векторному співвідношенні між початковим і поточним напрямками. Використовуючи скалярний і векторний добутки, інструмент визначає величину і напрямок зміни кута.

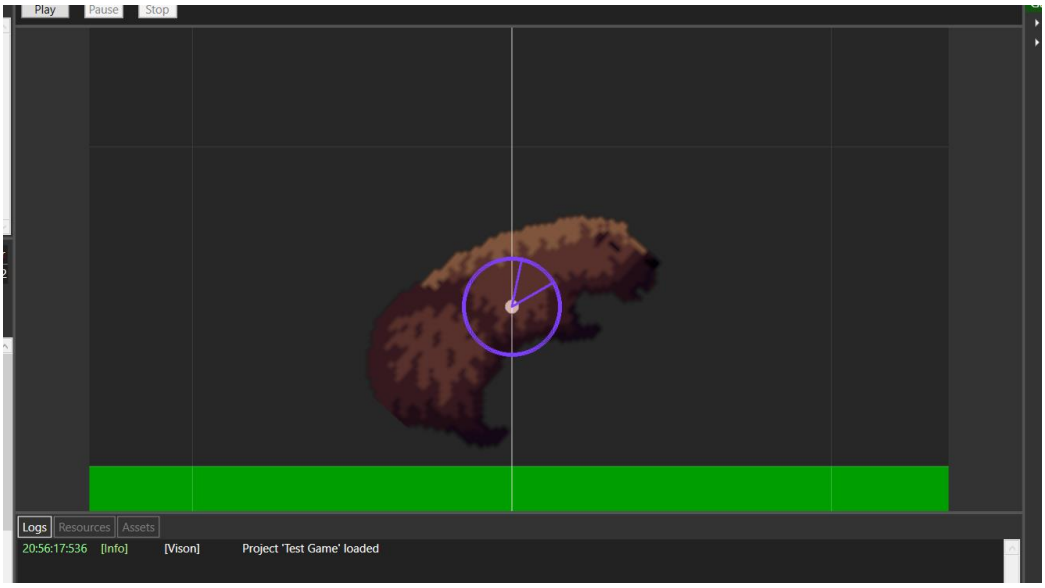

Рисунок 3.2.5 – Rotate tool

Під час обертання RotateTool візуалізує процес, показуючи лінії, які позначають початковий і поточний напрямки. Інструмент також враховує масштаб і зсув сцени, автоматично адаптуючи розмір і положення маніпулятора, щоб зберегти пропорційність і зручність використання незалежно від масштабу перегляду.

UVsTool — це інструмент редактора, який використовується для редагування UV-координат текстур. Він дозволяє змінювати положення та розмір UV-прямокутника, працюючи як із вершинами, так і з краями.

Інструмент представляє UV-прямокутник у вигляді чотирьох вершин (верхній лівий, верхній правий, нижній лівий, нижній правий) та чотирьох країв, які з'єднують ці вершини. Користувач може змінювати положення вершин, тим самим змінюючи розмір прямокутника, або пересувати окремі краї для зміни його форми. Взаємодія відбувається через логічний буфер, який визначає, яку частину UV-прямокутника обрав користувач. При наведенні миші на вершини або краї вони підсвічуються.

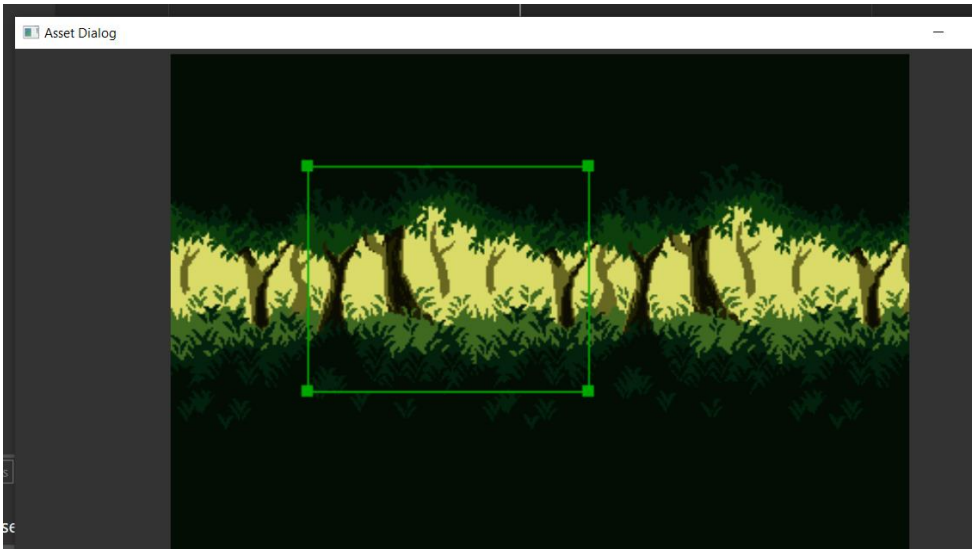

Рисунок 3.2.6 – UVs tool

Редагування UV-прямокутника здійснюється через натискання миші на вершину або край. Інструмент фіксує початкову позицію курсору та параметри прямокутника (координати, ширину і висоту), після чого обчислює зміни залежно від переміщення миші. Це дозволяє точно змінювати розміри та положення прямокутника, враховуючи всі взаємодії у реальному часі. Якщо ширина або висота прямокутника стають негативними, вони автоматично виправляються на мінімальні значення.

Метод Draw відповідає за візуалізацію UV-прямокутника. Лінії між вершинами утворюють контури прямокутника, а вершини малюються у вигляді маленьких квадратів. Інструмент використовує різні кольори для підсвічування активних частин: базовий колір для нерухомих елементів, інший — для виділених, та ще один для переміщуваних.

CameraViewportTool — це інструмент для керування в'юпортом камери, що дозволяє редагувати положення, масштаб і зону огляду камери безпосередньо у вікні редактора. Інструмент надає користувачу інтерактивні елементи керування, такі як рамки огляду та маніпулятори для масштабування і переміщення. Він інтегрується з системою відображення редактора, автоматично враховуючи зміни масштабу або положення камери.

Під час взаємодії з CameraViewportTool користувач може навігаційно змінювати зону огляду камери. Коли курсор миші наводиться на межі в'юпорту, активуються відповідні маніпулятори, які підсвічуються для зручності взаємодії. Наприклад, стрілки для переміщення камери підсвічуються червоним

і зеленим кольорами для осей X та Y відповідно. Для масштабування надаються інтерактивні маркери на кутах в'юпорту, які дозволяють пропорційно збільшувати або зменшувати поле зору камери.

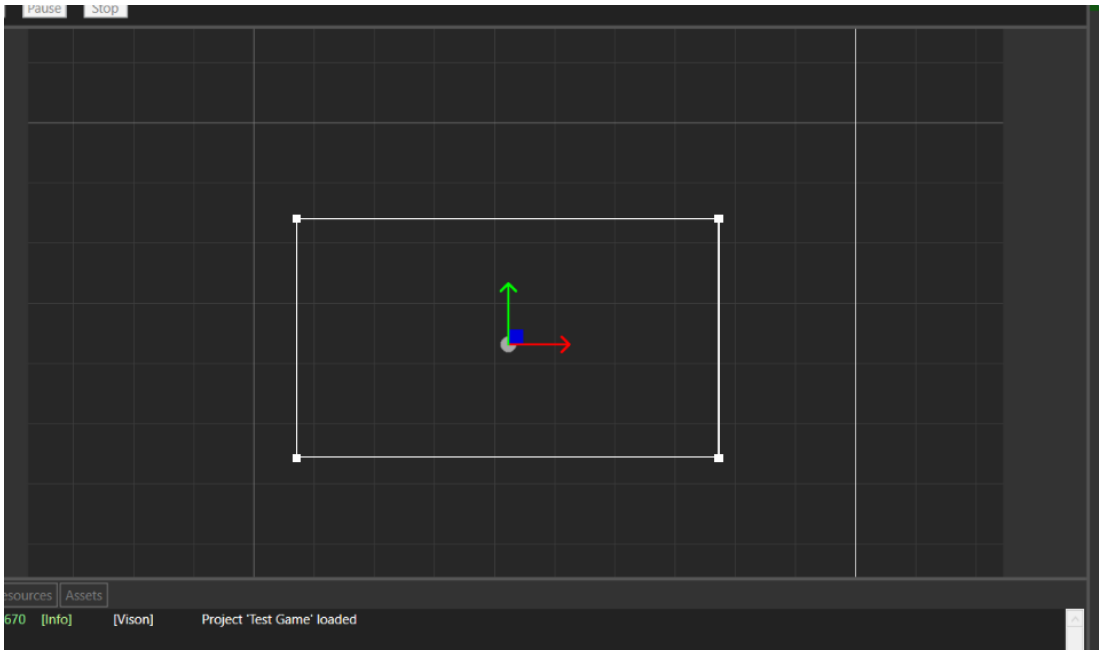

Рисунок 3.2.7 – Camera tool

Малювання маніпуляторів CameraViewportTool відбувається з урахуванням зсуву, масштабу та співвідношення сторін в'юпорту. Це забезпечує правильне відображення зони огляду незалежно від масштабу сцени. Крім того, інструмент синхронізується зі змінами положення або розміру вікна редактора, щоб завжди коректно відображати рамки й елементи керування.

Інструмент враховує перспективу та пропорції камери, дозволяючи точно налаштовувати її положення і масштаб. Крім того, CameraViewportTool інтегрується з системами подій редактора, що дозволяє використовувати його разом з іншими інструментами, такими як маніпулятори переміщення або обертання.

BoxColliderTool — це інструмент для редагування параметрів прямокутного колайдера (box collider), який забезпечує взаємодію об'єктів у фізичній системі. Інструмент дозволяє користувачеві змінювати розміри, положення та орієнтацію колайдера, інтерактивно налаштовуючи його параметри прямо в редакторі сцени.

Інструмент відображає рамку, що визначає межі колайдера, а також маніпулятори для редагування його розмірів і положення. Коли курсор миші наводиться на краї або кути колайдера, активуються відповідні маніпулятори, які підсвічуються для чіткої візуалізації. Наприклад, тягнучи за край рамки, можна збільшувати чи зменшувати ширину або висоту, тоді як кути дозволяють змінювати розміри одночасно по обох осях.

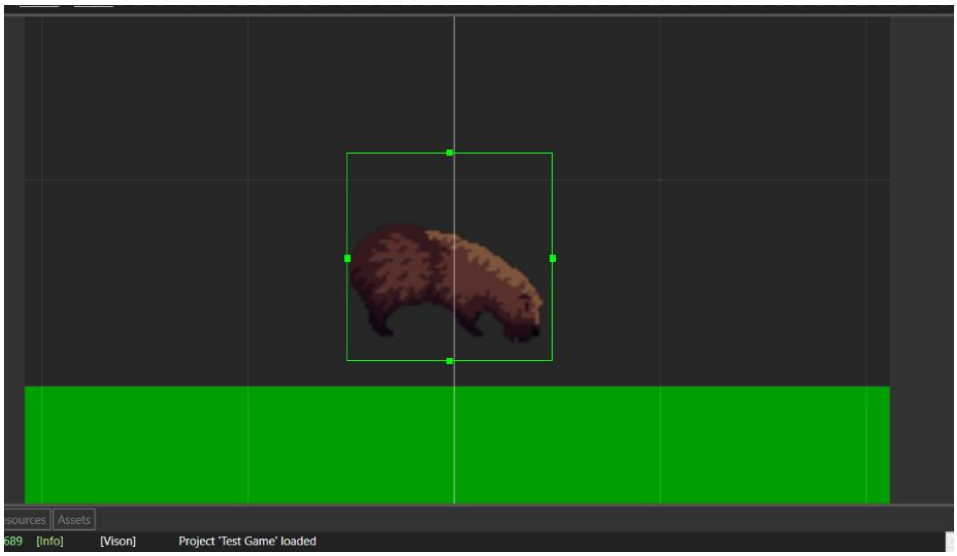

Рисунок 3.2.8 – Box Collider tool

Малювання маніпуляторів та рамки BoxColliderTool враховує масштаби та позицію камери сцени, що дозволяє завжди чітко бачити колайдер навіть у великих чи сильно зменшених сценах. Інструмент адаптується до орієнтації об'єкта: якщо об'єкт повертається, рамка колайдера також коректно відображає його нахил. Це важливо для точного налаштування колайдерів у складних сценах, де геометрія має нестандартні кути.

Під час редагування параметрів BoxColliderTool зберігає синхронізацію з фізичною системою, автоматично оновлюючи дані у відповідних буферах. Це дозволяє миттєво бачити результати змін.

CircleColliderTool — це інструмент для редагування кругових колайдерів (circle collider), які забезпечують обчислення фізичних взаємодій для об’єктів круглої форми. Він дозволяє легко налаштовувати основні параметри, такі як радіус і центр колайдера.

Інструмент візуалізує колайдер у вигляді кола з видимим радіусом і центром. Взаємодія з мишею дозволяє редагувати його параметри:

перетягування центральної точки змінює положення, а тягнення за межу кола дозволяє змінювати радіус. При наведенні курсору на активні області (центр або контур) вони підсвічуються, що допомагає чітко бачити, який параметр змінюється.

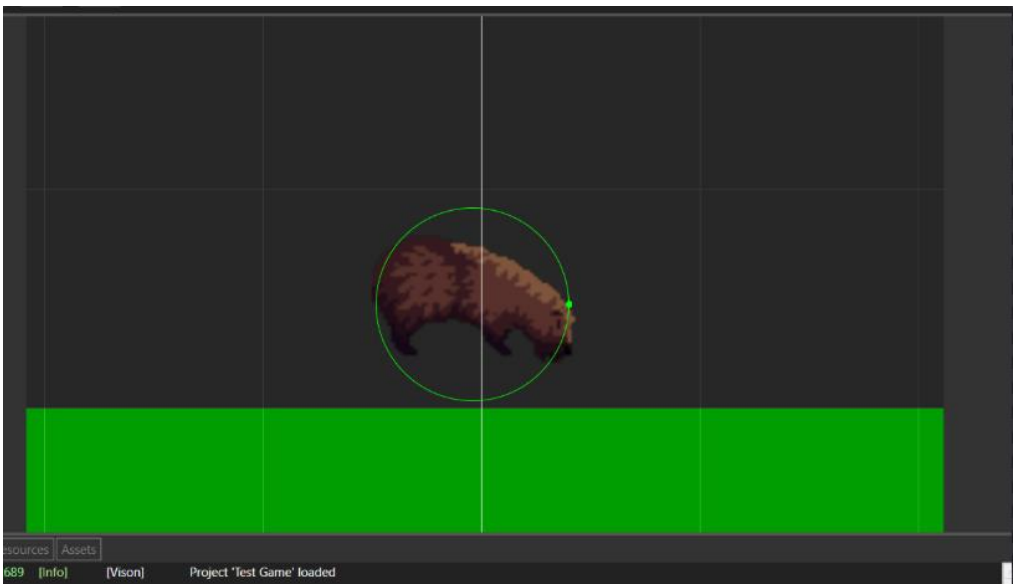

Рисунок 3.2.9 – Circle collider tool

CircleColliderTool автоматично адаптується до масштабу та позиції камери, забезпечуючи правильне відображення навіть у сценах із великими або дуже дрібними деталями. Інструмент також підтримує інтеграцію з системами трансформацій об’єктів, зберігаючи відповідність між геометрією сцени та фізичними параметрами. Якщо об'єкт змінює своє положення чи розмір, колайдер автоматично слідує за цими змінами.

PointLightTool — це інструмент для редагування параметрів точкового джерела світла, що використовується для створення локального освітлення в сцені. Він дозволяє змінювати основні властивості світла, такі як позиція, радіус впливу та інтенсивність, забезпечуючи користувачеві інтерактивні маніпулятори для точного налаштування.

Інструмент відображає точкове джерело світла у вигляді кола, яка відображає область його впливу. Користувач може перетягувати джерело світла для зміни його позиції або масштабувати коло, щоб налаштувати радіус освітлення. Візуальне підсвічування активних зон (центру або меж кола) допомагає чітко зрозуміти, які параметри наразі редагуються.

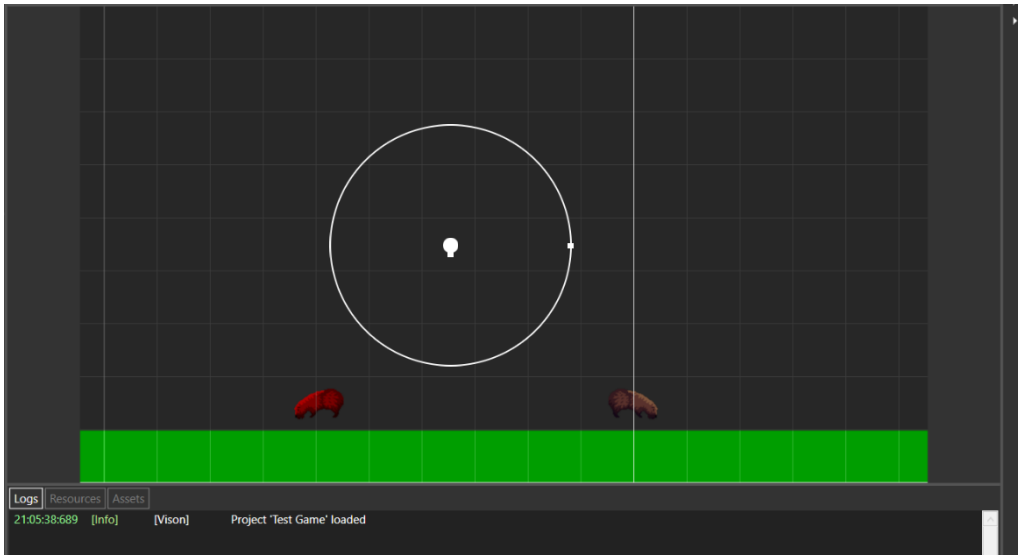

Рисунок 3.2.10 – Point light tool

PointLightTool інтегрується із системою камери та враховує зсув і масштаб сцени, щоб світло завжди відображалося у правильному контексті.

3.2.6 .NET host

Модуль DotnetHost забезпечує керування .NET-збірками в контексті проєкту. Він виконує ключові функції, такі як завантаження та перезавантаження .NET-модулів, інтеграцію їх із рушієм і обробку скриптів, визначених у проєкті.

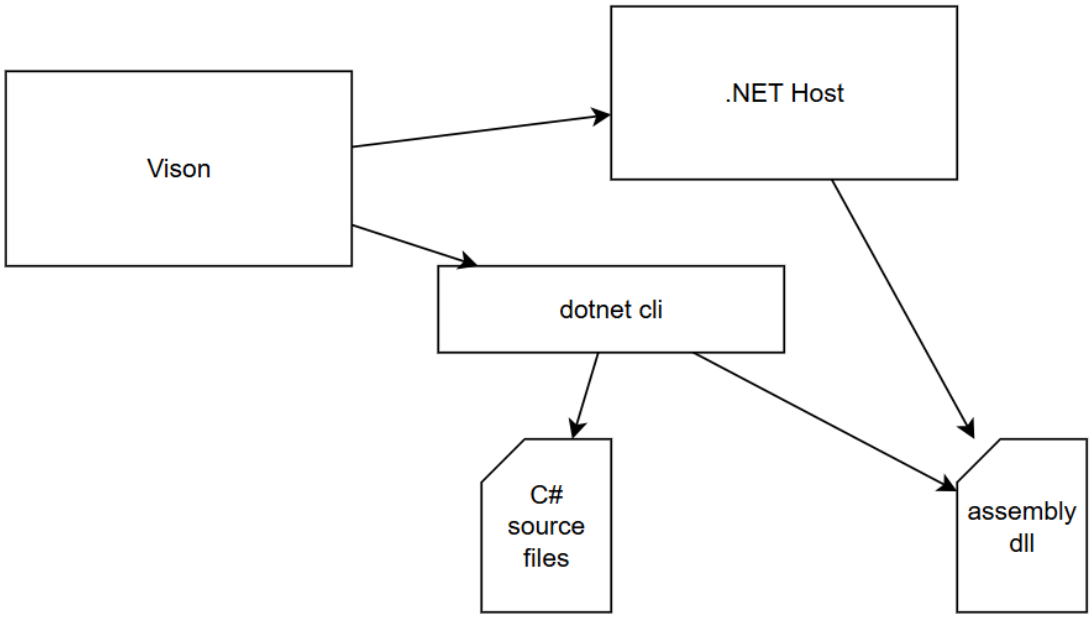

Рисунок 3.2.4 – .NET система

Компіляція здійснюється за допомогою dotnet cli. Модуль викликає окремий процес для створення нової збірки, після чого аналізує отримані логи на предмет помилок чи попереджень. Знайдені помилки та попередження обробляються та записуються у внутрішній логгер для подальшого використання. Після успішної компіляції .NET-збірка завантажується в спеціальний контекст виконання (DotnetExecutionContext), що надає ізольованість і можливість її перезавантаження без перезапуску програми.

Однією з важливих функцій модуля є рефлексія типів у згенерованій збірці. Після завантаження збірки інструмент Reflector аналізує скрипти, визначені в проєкті, і передає їх у систему рушія через внутрішній модуль ScriptBackend. Це дозволяє автоматично оновлювати доступні типи для використання в проєкті, додаючи їх до списку скриптів для відображення в

редакторі. Модуль також інтегрується з рушієм через API, забезпечуючи перезавантаження типів у системі при зміні збірки.

3.2.7 .Engine host

EngineHost — це ядро системи керування рушієм, що відповідає за запуск, конфігурацію, та виконання основних функцій рушія. Модуль інтегрується з нативними компонентами рушія через API, дозволяючи працювати зі сценами, ресурсами, ігровою логікою, а також забезпечує обробку рендерингу та взаємодію з редактором.

Основними завданнями модуля є ініціалізація рушія, керування ігровими сценами та забезпечення плавного ігрового циклу. EngineHost дозволяє створювати нові сцени, активувати їх та здійснювати оновлення стану гри у реальному часі. Модуль виконує ігровий цикл в окремому потоці. Керування потоком здійснюється через створений контролер потоку.

EngineHost дає змогу отримувати доступ до параметрів сцени, налаштовувати відображення, а також динамічно змінювати властивості вікон, такі як розмір або батьківський елемент.

Окрім запуску рушія, EngineHost надає інструменти для завершення роботи або перезапуску системи, що дозволяє адаптуватися до змін в конфігурації проєкту або оновлення внутрішніх компонентів.

3.2.8 EntityList

EntityList — це клас, який забезпечує інтерактивне керування списком сутностей поточної сцени. Він працює як спостерігаюча колекція, дозволяючи динамічно відстежувати зміни в реєстрі сутностей і автоматично синхронізувати стан з UI або іншими підсистемами. Клас інтегрується з низькорівневими подіями з реєстру через некерований код (unsafe).

Клас EntityList автоматично синхронізує свої дані з реєстром сутностей, переданим у конструктор. У методі Fetch відбувається ітерація по сутностях реєстру, і кожна сутність додається у список через CreateEntityInternal. Цей процес включає:

1. Ініціалізацію EntityListItem, яка зберігає інформацію про сутність.
2. Автоматичне визначення компонентів сутності через її архетип і додавання цих компонентів у список.

Це дозволяє динамічно будувати актуальну модель сутностей, яка відображає всі зміни в ECS у реальному часі.

Методи для додавання та видалення компонентів, такі як AddComponentInternal і RemoveComponentInternal, забезпечують оновлення інформації про компоненти сутностей. Додавання компонентів включає:

- Визначення компонентів на основі архетипу сутності.
- Ініціалізацію кожного компонента і створення його візуального представлення.

Клас інтегрується з низькорівневими подіями, такими як створення сутностей чи зміна компонентів, через делегати і колбеки. Наприклад:

- OnEntitiesCreatedCallback: реагує на створення нових сутностей у реєстрі. Сутності додаються в колекцію через диспатчер UI (Dispatcher), забезпечуючи синхронізацію з графічним інтерфейсом.
- OnComponentAddCallback і OnComponentRemoveCallback: обробляють додавання чи видалення компонентів.

3.2.9 BufferDrawer

BufferDrawer — це клас для малювання графічних примітивів на базовому рівні, який працює з прямим доступом до пам’яті. Основний принцип його роботи базується на маніпуляції буфером пікселів у форматі uint* (BGRA), що дозволяє змінювати кольори окремих пікселів екрану. Використання небезпечного коду забезпечує максимальну продуктивність, але вимагає від розробника дотримання обережності при доступі до даних.

BufferDrawer отримує на вхід посилання на буфер пікселів, а також його ширину та висоту. Усі операції малювання відбуваються шляхом прямого запису у цей буфер. Для запобігання помилкам, методи малювання перевіряють коректність координат (наприклад, щоб не вийти за межі буфера) за допомогою внутрішніх функцій SafeSetPixel і SafeSetPixelBlend. Це забезпечує захист від пошкодження пам’яті. Базові методи малювання, такі як DrawCircle, DrawRect або DrawLineWithStroke, побудовані на класичних алгоритмах графіки, таких як алгоритм Брезенхема.

Ось список основних методів класу BufferDrawer для малювання примітивів:

1. void SafeSetPixel(int x, int y, uint color)
   Установлює піксель у буфері з перевіркою коректності координат.

2. void SafeSetPixelBlend(int x, int y, uint color)
   Застосовує блендинг (накладання) кольору до пікселя з перевіркою коректності координат.
3. uint ReadPixel(int x, int y)
   Зчитує значення пікселя з буфера за заданими координатами.
4. void DrawLineWithStroke(int x0, int y0, int x1, int y1, int thickness, uint color)
   Малює лінію між двома точками з можливістю задавати товщину.
5. void DrawVerticalLine(int y0, int y1, int x, uint color)
   Малює вертикальну лінію між двома точками.
6. void DrawHorizontalLine(int x0, int x1, int y, uint color)
   Малює горизонтальну лінію між двома точками.
7. void DrawCircle(int centerX, int centerY, int radius, uint color)
   Малює контур кола з використанням алгоритму Брезенхема.
8. void DrawCircleFilled(int xc, int yc, int radius, uint color)
   Малює заповнене коло, використовуючи горизонтальні лінії.
9. void DrawCircleWithStroke(int centerX, int centerY, int radius, int stroke_width, uint color)
   Малює контур кола з вказаною товщиною обведення.
10. void DrawRect(int x, int y, int w, int h, int thickness, uint color)
    Малює контур прямокутника з заданою товщиною.
11. void DrawFilledRect(int x, int y, int w, int h, uint color)
    Малює заповнений прямокутник.
12. void DrawRectBlend(int x, int y, int w, int h, int thickness, uint color)
    Малює контур прямокутника з використанням блендингу кольорів.
13. void DrawRectFilledBlend(int x, int y, int w, int h, uint color)
    Малює заповнений прямокутник із блендингом кольорів.
14. void DrawPolygon(Vector2[] points, int stroke, uint color)
    Малює багатокутник на основі масиву точок.
15. void DrawBitmap(int x, int y, UnsafeBitmap bitmap)
    Малює растрове зображення, копіюючи його пікселі в буфер.
16. void DrawGrid(int cellSize, uint color, int dx = 0, int dy = 0)
    Малює сітку з клітинками заданого розміру.
17. void Fill(int v)
    Заповнює весь буфер заданим значенням кольору.

ВИСНОВКИ

У рамках даної роботи було вдосконалено створений раніше власний ігровий рушій для розробки 2D-ігор створенням зручного візуального редактора, який забезпечує інтерактивні засоби для роботи зі сценами, керування ресурсами та тестування ігрових механік. Використовуючи сучасні технології, такі як C++, C#, WPF і Vulkan, вдалося створити потужний інструмент, що дозволяє розробникам швидко створювати, налаштовувати й удосконалювати 2D-ігри.

Редактор надає модульну архітектуру, інтеграцію з рушієм для візуалізації сцени в реальному часі, систему для редагування аудіо та графічних ресурсів, а також інструменти для створення скриптів і графічних елементів інтерфейсу.

Внесено зміни в сам рушій, наприклад, реалізація рендер-графу та нової системи ресурсів підвищили продуктивність і гнучкість рушія, дозволяючи ефективно працювати з великими проєктами. Розширення .NET API для скриптів з підтримкою масивів та полів, чого бракувало в попередній версії рушія.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. .NET documentation [Електронний ресурс]. Режим доступу: https://docs.microsoft.com/en-us/dotnet.
2. Vulkan tutorial [Електронний ресурс]. Режим доступу: https://vulkan-tutorial.com.
3. Vulkan C++ examples and demos by Sascha Willems [Електронний ресурс]. Режим доступу: https://github.com/SaschaWillems/Vulkan
4. Dotnet native hosting [Електронний ресурс]. Режим доступу: https://github.com/dotnet/runtime/blob/main/docs/design/features/native-hosting.md
5. Vulkan Programming Guide: The Official Guide to Learning Vulkan by Graham Sellers, John Kessenich – Addison-Wesley Professional October 31, 2016

ДОДАТКИ

ДОДАТОК А

Реалізація редактору

ProjectContext.cs

```
using System;
using System.Linq;
using System.Windows.Input;
using System.Collections.Generic;
using System.Runtime.InteropServices;

using Laska;
using Laska.Dotnet;
using Laska.Interop.Utils;

using Vison.Controls.Components;
using Vison.ViewModels;
using System.Windows.Threading;
using System.Windows;
using System.Windows.Interop;
using System.IO;


namespace Vison.Infrastructure
{
    public sealed class ProjectContext : IDisposable
    {
        public event SceneChangedHandler SceneChanged;
        public event ContextLoadedHandler Loaded;

        public Dispatcher Dispatcher { get; set; }

        public Logger Logger { get; set; }

        public ProjectViewModel ViewModel { get; set; }

        public VisonFile Config { get; set; }
        public SceneContext CurrentScene { get; set; }

        public Parameters Parameters { get; set; }
        public SelectionContext Selection { get; set; }
        public ResourceManager Resources { get; set; }
        public AssetManager Assets { get; set; }
        public ImageOverlay Overlay { get; set; }
        public EditorToolBox ToolBox { get; set; }

        public DotnetHost Dotnet { get; set; }
        public EngineHost Engine { get; set; }


        private IntPtr _parent;
        private FileSystemWatcher _watcher;

        public ProjectContext(VisonFile file, nint parent)
        {
```

```
            Config = file;
            ViewModel = new(this);
            _parent=parent;
        }

        public unsafe void Load()
        {
            Logger = new(Dispatcher);



            Engine = new(Logger);

            SetupEngineHost();

            Selection = new();
            ToolBox = new();
            ToolBox.Mode = ToolBoxMode.Select;
            ToolBox.Selection = Selection;
            Parameters = new();
            Overlay = new();
            Overlay.ToolBox = ToolBox;
            Overlay.SetEditorCamera(Engine.GetEditorCamera());
            Resources = ResourceManager.ParseFile(Config.Content.Resources,
Config.Content.Path, Engine.GetModule(LaskaModule.ResourceManager));
            Assets = AssetManager.ParseFile(Config.Content.Assets,
new(Engine.GetModule(LaskaModule.AssetManager)));
            Dotnet = new(Logger, Config.Scripts.Dotnet,
Engine.GetModule(LaskaModule.DotNet));

            ViewModel.Selection = Selection;
            ViewModel.EditorOverlay = Overlay;
            ViewModel.EditorCamera = Engine.GetEditorCamera();
            ViewModel.GameViewPort = Engine.GetViewport(0);
            ViewModel.EditorViewport = Engine.GetViewport(1);
            ViewModel.Tree = CreateProjectTree();
            ViewModel.Resources = Resources;
            ViewModel.Assets = Assets;


            Selection.SelectionChanged += Selection_SelectionChanged;

            App.Window.KeyDown += MainWindow_KeyDown;
            App.Window.KeyUp += MainWindow_KeyUp;

            Dotnet.Rebuild();

            Loaded?.Invoke(this, new());

            Logger.Info($"Project '{Config.Name}' loaded");

            // Console.SetOut(Logger.TextWriter);

            ViewModel.IsLoading = false;
        }

        private unsafe void SetupEngineHost()
        {
```

```
            Engine.Configure(builder =>
            {
                builder.WithWindow(window =>
                {
                    window.SetParent(0);
                    window.SetResolution(1280, 720);
                    window.SetTitle("WND");
                })
                .WithEditor(editor =>
                {

                })
                .WithViewports(viewports =>
                {
                    viewports.AddViewport(new()
                    {
                        Type = ViewportType.Window,
                        Parent = 0,
                        Height = 720,
                        Width = 1280,
                        IsValid = true,
                        Name =
(char*)MemoryUtils.ConvertCSharpStringASCIICString("Game"),
                        Resized = true,
                    });

                    viewports.AddViewport(new()
                    {
                        Type = ViewportType.Offscreen,
                        Height = 720,
                        Width = 1280,
                        IsValid = true,
                        Name =
(char*)MemoryUtils.ConvertCSharpStringASCIICString("Vison"),
                        Resized = true,
                    });
                })
                .WithGame(game =>
                {

                })
                .WithDotnet(dotnet =>
                {
                    delegate* unmanaged[Cdecl]<DotnetInit*, void> ptr = (delegate*
unmanaged[Cdecl]<DotnetInit*, void>)&Bridge.Init;

                    dotnet.WithUseExternalDotnet(true);
                    dotnet.WithInitHook(ptr);
                });
            });
            Engine.Init();
        }

        public void Save()
        {
            Assets.Save();
            Resources.Save();
            CurrentScene.Save();
```

```
        }

        private void MainWindow_KeyUp(object sender, KeyEventArgs e)
        {
            Overlay.OnKeyUp(e.Key);
            ToolBox.OnKeyUp(e.Key);

            if (e.KeyboardDevice.Modifiers == ModifierKeys.Control && e.Key ==
Key.S)
            {
                Save();
            }
        }

        private void MainWindow_KeyDown(object sender, KeyEventArgs e)
        {
            Overlay.OnKeyDown(e.Key);
            ToolBox.OnKeyDown(e.Key);
        }

        private void Selection_SelectionChanged(object sender,
SelectionChangedEventArgs e)
        {
            if (e.Entities.Length > 0)
            {
                EntityListItem entity = e.Entities[0];


                ToolBox.Move.BindEntity(entity);
                ToolBox.Scale.BindEntity(entity);
                ToolBox.Rotate.BindEntity(entity);
                ToolBox.EntityMark.BindEntity(entity);




                ToolBox.SetMode(ToolBoxMode.Move);
            }
        }

        public unsafe void LoadScene(int index)
        {
            ProjectSceneInfo scene = Config.Scenes[index];

            if (scene is null)
            {
                return;
            }

            LoadScene(scene);
        }

        public unsafe void LoadScene(string name)
        {
            ProjectSceneInfo scene = Config.Scenes.FirstOrDefault(s => s.Alias ==
name);

            if (scene is null)
```

```
            {
                return;
            }

            LoadScene(scene);
        }

        public SceneContext LoadScene(ProjectSceneInfo scene)
        {
            SceneContext context = SceneContext.ParseSceneFile(scene.Path, this);
            // context.Mode = SceneMode.Preview;

            CurrentScene = context;

            Engine.PlayScene(context.Scene);

            return context;
        }

        public IEnumerable<IProjectNode> CreateProjectTree()
        {
            _watcher = new FileSystemWatcher(Config.Content.Path)
            {
                IncludeSubdirectories = true,
                EnableRaisingEvents = true,
            };

            _watcher.NotifyFilter = NotifyFilters.Attributes
                     | NotifyFilters.CreationTime
                     | NotifyFilters.DirectoryName
                     | NotifyFilters.FileName
                     | NotifyFilters.LastAccess
                     | NotifyFilters.LastWrite
                     | NotifyFilters.Security
                     | NotifyFilters.Size;

            FolderProjectNode content = new(new
DirectoryInfo(Config.Content.Path));

            VirtualProjectNode scripts = new("Scripts");

            VirtualProjectNode scenes = new("Scenes");
            scenes.AddChildren(Config.Scenes.Select(s => new
VirtualProjectNode(s.Alias, "\\Icons\\ProjectTree\\file_empty_icon.png")));


            IProjectNode[] nodes = new IProjectNode[]
            {
                content,
                scenes,
                scripts,
            };



            return nodes;
        }
```

```
        public void Dispose()
        {

        }

        public static ProjectContext ParseVisonFile(string path, IntPtr parent)
        {
            VisonFile file = VisonFile.LoadFromPath(path);
            ProjectContext context = new(file, parent);


            return context;
        }
    }
}
```

ImageOverlay.cs

```
using System;
using System.Collections.Generic;
using System.Numerics;
using System.Diagnostics.CodeAnalysis;
using System.Windows.Documents;
using Laska;
using System.Windows.Media.Media3D;
using System.Windows.Input;

namespace Vison.Infrastructure
{
    public struct Box2d
    {
        public Vector2 V1;
        public Vector2 V2;
        public Vector2 V3;
        public Vector2 V4;
    }

    public interface IImageOverlay
    {

    }

    public unsafe sealed class ImageOverlay : IImageOverlay
    {
        public ToolBox ToolBox { get; set; }

        private EditorCamera _camera;
        private OverlayDrawer _drawer;


        public ImageOverlay()
        {
            _drawer = new();

            ToolBox = new();
        }

        public void Draw()
```

```
{
    OverlayTransform transform = default;

    if (_camera.Camera != null)
    {
        float scale = _camera.Camera->Scale;
        float dx = _camera.Transform->Position.X;
        float dy = _camera.Transform->Position.Y;

        transform = new(dx, dy, scale);
    }

    _drawer.ClearLogicInkBuffer();

    foreach (var (uuid, tool) in ToolBox.Tools)
    {
        if (tool.Active)
        {
            tool.Draw(transform, _drawer);
        }
    }
}

public void SetEditorCamera(EditorCamera camera)
{
    _camera = camera;

    foreach (var (uuid, tool) in ToolBox.Tools)
    {
        tool.SetCamera(camera);
    }
}

public void OnViewportMove(float x, float y)
{
    foreach (var (uuid, tool) in ToolBox.Tools)
    {
        if (tool.Active)
            tool.OnViewportMove(x, y);
    }
}

public void OnViewportScale(float scale)
{
    foreach (var (uuid, tool) in ToolBox.Tools)
    {
        if (tool.Active)
            tool.OnViewportScale(scale);
    }
}

public void OnMouseMove(int x, int y)
{
    foreach (var (uuid, tool) in ToolBox.Tools)
    {
        if (tool.Active)
            tool.OnMouseMove(x, y, _drawer);
    }
```

```
}

public void OnMouseDown(int x, int y, MouseButton button)
{
    foreach (var (uuid, tool) in ToolBox.Tools)
    {
        if (tool.Active)
            tool.OnMouseDown(x, y, button, _drawer);
    }
}

public void OnMouseUp(int x, int y, MouseButton button)
{
    foreach (var (uuid, tool) in ToolBox.Tools)
    {
        if (tool.Active)
            tool.OnMouseUp(x, y, button, _drawer);
    }
}

public void OnMouseEnter(int x, int y)
{
    foreach (var (uuid, tool) in ToolBox.Tools)
    {
        if (tool.Active)
            tool.OnMouseEnter(x, y, _drawer);
    }
}

public void OnMouseLeave(MouseEventArgs e)
{
    foreach (var (uuid, tool) in ToolBox.Tools)
    {
        if (tool.Active)
            tool.OnMouseLeave(e, _drawer);
    }
}

public void OnMouseWheel(MouseWheelEventArgs args)
{
    foreach (var (uuid, tool) in ToolBox.Tools)
    {
        if (tool.Active)
            tool.OnMouseWheel(args, _drawer);
    }
}

public void OnKeyDown(Key key)
{
    foreach (var (uuid, tool) in ToolBox.Tools)
    {
        if (tool.Active)
            tool.OnKeyDown(key);
    }
}

public void OnKeyUp(Key key)
{
```

```
            foreach (var (uuid, tool) in ToolBox.Tools)
            {
                if (tool.Active)
                    tool.OnKeyUp(key);
            }
        }

        public void BindBuffer(uint* buffer, int width, int height)
        {
            _drawer.BindBuffer(buffer, width, height);
        }

        public void BindCamera(EditorCamera camera)
        {
            _camera = camera;
        }
    }
}
```

SceneContext.cs

```
using Laska;
using Laska.Interop.Utils;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.InteropServices;
using Vison.ViewModels;

namespace Vison.Infrastructure
{
    public sealed unsafe class SceneContext
    {
        public SceneMode Mode
        {
            get => _sceneMode;
            set => SetMode(value);
        }

        public EngineHost Engine { get; }
        public SceneFile File { get; }
        public Scene Scene { get; }
        public EntityList Entities { get; set; }
        public SelectionContext Selection { get; set; }
        public SceneViewModel ViewModel { get; set; }
        public DotnetHost Dotnet { get; set; }


        private readonly AssetManager _assets;
        private readonly ProjectContext _projectContext;

        private SceneMode _sceneMode = SceneMode.Play;

        public SceneContext(ProjectContext project, SceneFile sceneFile, Scene
scene)
        {
            _projectContext = project;
```

```
            _assets = project.Assets;

            Engine = project.Engine;
            Dotnet = project.Dotnet;
            File = sceneFile;
            Scene = scene;
            Entities = new(
                registry: &scene.Proxy->_registry,
                dispatcher: System.Windows.Threading.Dispatcher.CurrentDispatcher
            );

            Entities.OnComponentAdd += Entities_OnComponentAdd;
            Entities.OnComponentAdded += Entities_OnComponentAdded;

            Entities.OnComponentRemove += Entities_OnComponentRemove;

            Entities.OnEntityCreate += Entities_OnEntityCreate;
            Entities.OnEntityCreated += Entities_OnEntityCreated;

            Entities.OnEntityDestroy += Entities_OnEntityDestroy;

            Entities.Fetch();

            ViewModel = new(this);
        }

        private void SetMode(SceneMode mode)
        {
            LaskaNative.LaskaSceneSetMode((IntPtr)Scene.Proxy, mode);
            _sceneMode = mode;
        }

        private void Entities_OnComponentAdd(EntityList list, EntityListItem
entity, EntityListItemComponent component)
        {
            if (!File.Entities.TryGetValue(entity.Id, out var item))
            {
                return;
            }

            component.Project = _projectContext;

            string key = ComponentRegistry.GetName(component.Id);

            if (!item.ComponentsData.TryGetValue(key, out ComponentData data))
            {
                data = ComponentData.CreateNative(component.Id);

                item.ComponentsData.Add(key, data);
            }

            component.RecreateView(data);
        }
        private void Entities_OnComponentAdded(EntityList arg1, EntityListItem
arg2, EntityListItemComponent arg3)
        {

        }
```

```
        private void Entities_OnComponentRemove(EntityList arg1, EntityListItem
entity, EntityListItemComponent component)
        {
            if (!File.Entities.TryGetValue(entity.Id, out var item))
            {
                return;
            }

            string key = ComponentRegistry.GetName(component.Id);

            item.ComponentsData.Remove(key);
        }


        private void Entities_OnEntityCreate(EntityList arg1, EntityListItem arg2)
        {
            if (File.Entities.ContainsKey(arg2.Id))
            {
                return;
            }

            var item = new HierarchyItem();

            File.Entities.Add(arg2.Id, item);
        }

        public void Save()
        {
            File.Save();
        }


        public ulong CreateEntity(ulong archetype, bool save = true)
        {
            var uuud = Scene.CreateEntity(archetype);


            return uuud;
        }

        public void DestroyEntity(ulong entity)
        {
            Scene.DestroyEntity(entity);
        }

        public void AddComponent(ulong entity, ulong id)
        {
            var ptr = Scene.AddComponent(entity, id);
        }

        public void AddComponent<T>(ulong entity)
            where T : IComponent, new()
        {
            AddComponent(entity, ComponentRegistry.GetIndex<T>());
        }

        public IntPtr GetComponent<T>(ulong entity)
```

```
            where T : IComponent, new()
        {
            IntPtr ptr = Scene.GetComponent(entity,
ComponentRegistry.GetIndex<T>());

            return ptr;
        }

        public void RemoveComponent<T>(ulong entity)
            where T : IComponent, new()
        {
            Scene.RemoveComponent(entity, ComponentRegistry.GetIndex<T>());
        }



        public static SceneContext ParseSceneFile(string path, ProjectContext
project)
        {
            var file = SceneFile.LoadFromFile(path);

            SceneCreateInfo info = default;
            info.RequiredHandles =
MemoryUtils.AllocateSpan<ulong>((ulong)file.Assets.Count);
            info.Entities =
MemoryUtils.AllocateSpan<SceneEntityCreateInfo>((ulong)file.Hierarchy.Count);


            if (file.Archetypes.Count > 0)
            {
                info.Archetypes =
MemoryUtils.AllocateSpan<ArchetypeInfo>((ulong)file.Archetypes.Count);

                var i = 0;

                foreach (var archetype in file.Archetypes)
                {
                    info.Archetypes[i].Components =
MemoryUtils.AllocateSpan<ushort>((ulong)archetype.Components.Count);

                    int j = 0;

                    foreach (var component in archetype.Components)
                    {
                        info.Archetypes[i].Components[j] =
(ushort)ComponentRegistry.GetIndex(component);

                        j++;
                    }

                    i++;
                }
            }


            List<ComponentFieldBinding> bindings = new();

            for (int i = 0; i < file.Assets.Count; i++)
```

```
            {
                info.RequiredHandles[i] = file.Assets[i];
            }

            for (int i = 0; i < file.Hierarchy.Count; i++)
            {
                HierarchyItem item = file.Hierarchy[i];

                ref SceneEntityCreateInfo entity = ref info.Entities[i];

                entity.Id = item.Id;
                entity.Name = (char*)Marshal.StringToHGlobalAnsi(item.Name);
                entity.Prototype = item.Prototype;
                entity.Components =
MemoryUtils.AllocateSpan<SceneEntityComponentInfo>((ulong)item.Components.Count);

                int j = 0;

                foreach (var (key, value) in item.ComponentsData)
                {
                    if (ComponentDataLoaders.TryGetByName(key, out var loader))
                    {
                        entity.Components[j] = loader.ToComponentInfo(value);

                        foreach (var binding in value.Bindings)
                        {
                            binding.Value.Asset =
project.Assets.FirstOrDefault((a) => a.Id == binding.Value.Id);
                        }

                        j++;

                        var bngs = loader.GetBindings(value, entity.Id);

                        foreach (var binding in bngs)
                        {
                            bindings.Add(binding);
                        }
                    }
                }
            }

            if (bindings.Count > 0)
            {
                info.Bindings =
MemoryUtils.AllocateSpan<ComponentFieldBinding>((ulong)bindings.Count);

                for (int i = 0; i < bindings.Count; i++)
                {
                    info.Bindings[i] = bindings[i];
                }
            }

            Scene scene = project.Engine.CreateScene(ref info);

            var contex = new SceneContext(project, file, scene);
            contex.Selection = project.Selection;
```

```
            return contex;
        }
    }
}
```