Галицький фаховий коледж імені В’ячеслава Чорновола
відділення комп’ютерних та видавничих технологій
циклова комісія інформатики та комп’ютерних дисциплін

ДОПУСТИТИ ДО ЗАХИСТУ

Завідувач відділенням
комп’ютерних та видавничих
технологій

Чубей О.О. /_____/
підпис

«___»_____202_ р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

до дипломного проєкту
освітньо-кваліфікаційного рівня «молодший спеціаліст»
зі спеціальності 122 «Комп’ютерні науки»
на тему: «Ігровий двигун для побудови 3D графіки на мові С++»

Студент групи КН-41 Гуменний М.А. _____
(підпис)

Керівник проєкту Івасьєв С.В. _____
(підпис)

Консультанти:

з техніко-економічного
обґрунтування Меленчук Л.І. _____
(підпис)

нормо контролер Гавришків Н.Г. _____
(підпис)

Тернопіль – 2022

Галицький фаховий коледж імені В’ячеслава Чорновола

відділення комп’ютерних та видавничих технологій

циклова комісія інформатики та комп`ютерних дисциплін

ЗАТВЕРДЖУЮ
Завідувач відділенням
комп’ютерних та видавничихтехнологій

Чубей О.О. /_____/
підпис
«_ __»_____2021 р.

ЗАВДАННЯ

на дипломне проєктування
на здобуття освітньо-кваліфікаційного рівня «молодший спеціаліст»

студенту _____
(прізвище, ім’я та по-батькові студента)

1. Тема проєкту_____
_____
_____
_____

затверджена наказом по коледжу від 01.10.2021 р., № 178-н

2. Термін здачі студентом завершеного проєкту “_____” _____2022 р

3. Вихідні дані до проєкту _____
_____
_____
_____

4. Перелік питань, які повинні бути розроблені в проєкті: _____

а) основна частина _____
_____
_____
_____

б) техніко-економічного обґрунтування _____
_____
_____
_____
_____

5. Перелік графічного матеріалу_____
_____
_____

6. Консультанти проєкту: _____

Розділ	Консультанти	Підпис, дата	
		Завдання видано	Завдання прийнято
з техніко-економічного обґрунтування	Меленчук Л.І. (вчена ступень, звання П.І.Б. консультанта)		

КАЛЕНДАРНИЙ ПЛАН
дипломного проєктування

№ п/п	Найменування етапу	Терміни	
		початку	завершення
1.	Вибір теми, ознайомлення з вимогами до дипломного проєктування.	28.09.21 р.	01.10.21 р.
2.	Огляд типових рішень та написання відповідного розділу ПЗ	06.12.21 р.	26.01.22 р.
3.	Дослідження технологій реалізації та написання відповідного розділу ПЗ	26.01.22 р.	14.02.22 р.
4.	Розробка функціональних вимог до проєкту та робота над структурою програмного продукту. Написаннявідповідного розділу ПЗ	14.02.22 р.	02.03.22 р.
5.	Встановлення на налаштування середовища реалізації та написання відповідного розділу ПЗ	02.03.22 р.	16.03.22 р.
6.	Проєктування програмного засобу (функціоналу, інтерфейсу, бази даних продукту) та написання відповідного розділу ПЗ	16.03.22 р.	17.04.22 р.
7.	Реалізація та налаштування програмного засобу та написання відповідного розділу ПЗ	17.04.22 р.	03.05.22 р.
8.	Доопрацювання модулів	05.05.22 р.	18.05.22 р.
9.	Тестування на налагодження програмного продукту та написання відповідного розділу ПЗ	18.05.22 р.	01.06.22 р.
10.	Опрацювання економічного розділу дипломного проєкту та оформлення спеціального розділу	20.05.22 р.	05.06.22 р.
11.	Робота над оформленням пояснювальної записки	05.06.22 р.	12.06.22 р.
12.	Попередній захист дипломного проєкту, доопрацювання	15.06.22 р.	
13.	Підготовка до захисту дипломного проєкту	16.06.22 р.	22.06.22 р.
14.	Захист дипломного проєкту	25.06.22 р.	26.06.22 р.

7. Дата видачі завдання ”____” _____ 2021 р.

Керівник _____ / _____

Завдання прийняв до виконання_____ / _____

Реферат

Ігровий двигун для побудови 3D графіки на мові C++. Дипломний проєкт. Гуменний Максим. Галицький коледж імені В’ячеслава Чорновола відділення комп’ютерних та видавничих технологій. Спеціальність 122 «Комп’ютерні науки». ГК, 2021, сторінок - 98, рисунків - 33, таблиці - 2, додатків - 1.

Об’єктом дослідження є застосування нових технологій для побудови графіки та пришвидшення процесу розробки. Для цього проаналізовано існуючі ігрові движки, а саме: «Godot», «Marmelade», «Corona», «Unity». Мета проєкту – створення власного ігрового двигунця на основі проаналізованих, усунувши виявлені в них недоліки. Розробка власної логіки та зручного програмного інтерфейсу додатку. Завданням проєкту є розробка ігрового двигунця. Результат – ігровий движок для побудови 3D графіки, що повністю готовий до експлуатації. Зручний та швидкий застосунок для швидкої та якісної розробки 3D комп’ютерних ігор.

Для розробки додатку було обрано середовище розробки Visual Studio, що спеціалізується на створенні програмних додатків для комп’ютерів на базі операційної системи Windows. Visual Studio підтримує такі мови програмування: C, C++, C# та JavaScript. Мовою на якій буде реалізовано програмний продукт є C++, причиною є зрозумілий синтаксис, велика кількість програмної документації та реалізованих бібліотек. Для створення основної бібліотеки було обрано середовище розробки Visual Studio. Моделі розроблялись за допомогою Blender. Організація моделей реалізовувалась на основі технології JSON. Програмування відбувається за допомогою мови C++. Функціями двигунця є можливість швидкої організації ігрових ресурсів та графічне їх відображення з можливістю подальшої взаємодії.

ІГРОВИЙ ДВИГУНЕЦЬ, РОЗРОБКА ІГОР, VISUAL STUDIO, ОПЕРАЦІЙНА СИСТЕМА, C++, JSON, 3D ГРАФІКА, МОВА ПРОГРАМУВАННЯ.

Abstract

Game engine for building 3D graphics in C++. Diploma project. Humenny Maksim. Vyacheslav Chornovil Galician College department of computer and publishing technologies. Specialty 122 "Computer Science". GC, 2021, pages - 98, figures - 33, tables - 2, applications - 1.

The object of research is the application of new technologies to build graphics and speed up the development process. To do this, we analyzed the existing game engines, namely: "Godot", "Marmelade", "Corona", Unity. The purpose of the project is to create its own game engine on the basis of the analyzed ones, eliminating the shortcomings identified in them. Development of own logic and user-friendly application programming interface. The task of the project is to develop a game engine. The result is a game engine for building 3D graphics, which is completely ready for use. Convenient and fast application for fast and high-quality development of 3D computer games.

Visual Studio, a development environment that specializes in creating software applications for computers based on the Windows operating system, was chosen to develop the application. Visual Studio supports the following programming languages: C, C ++, C # and JavaScript. The language in which the software will be implemented is C ++, the reason is clear syntax, a large number of software documentation and implemented libraries. The Visual Studio development environment was chosen to create the main library. Models were developed using Blender. The organization of models was implemented on the basis of JSON technology. Programming is done using the C ++ language. The functions of the engine are the ability to quickly organize game resources and their graphical display with the possibility of further interaction.

GAME ENGINE, GAME DEVELOPMENT, VISUAL STUDIO, OPERATING SYSTEM, C++, JSON, 3D GRAPHICS, PROGRAMMING LANGUAGE.

Зміст

Вступ…….. 6

1 Аналіз ринку ігрових рушіїв…….. 8

1.1 Аналіз існуючих аналогів…….. 8

1.2 Вибір програмного забезпечення…….. 15

1.3 Постановка задачі…….…… 19

2 Технічні аспекти реалізації ігрових рушіїв…….…. 24

2.1 Визначення основних елементів двигунця…….. 24

2.2 Архітектура рушія…….…… 27

2.3 Інструменти для налагодження……..…….. 35

3 Реалізація та тестування……. 43

3.1 Вибір бібліотек…… 43

3.2 Реалізація функцій рендерингу……. 47

3.3 Реалізація камери…… 53

3.4 Реалізація мешів…….. 56

3.5 Реалізація моделей……. 60

3.6 Тестування розробленого програмного забезпечення…… 63

4 Техніко-економічне обгрунтування…….. 67

4.1 Аналіз ринку…….. 67

4.2 Розрахунок витрат на проєктування……. 68

4.3 Обгрунтування необхідності розробки…… 69

Висновки…….. 71

Перелік джерел посилання…… 72

Додатки…… 73

					ДП.КН.22.460.26.000 ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата				
Розробив		Гуменний М.А.			Ігровий двигун для побудови 3D графіки на мові C++	Літ.	Арк.	Акрушів
Перевірив		Івасьєв С.В.					5	96
Рецецнз.		Кульчинська Н.З.				ГФК.ВКВТ.ЦКІКД КН-41		
Н. Контр.		Гавришків Н.Г.						
Затверд.		Чубей О.О.						

ВСТУП

Програми для графічного відтворення об’єктів зараз широко затребувані в різних сферах, особливо в сфері розваг. Це мабуть найбільша та найпотужніша галузь розвитку даних технологій, де вони розкривають весь свій потенціал.

Ігровий рушій слугує для спрощення розробки ігор. Замість того щоб прописувати все вручну, що дуже довго і ресурсоємно, доцільніше було б організувати все через цей самий рушій, який включає в себе всі її аспекти і ресурси, це графіка, аудіоматеріали, вся ігрова логіка. Можливості рушія напряму залежать від апаратних можливостей пристрою на якому він працює. Через неймовірний розвиток саме апаратної частини ПК за останні роки з’явилось дуже багато нових цікавих технологій які дають можливість більш кращої реалізації поставлених задач та планів для багатьох компаній. Оскільки технології завжди розвиваються то і ігрові двигунці повинні вдосконалюватись щоб не втратити актуальність.

На даний момент практично кожна ігрова студія прагне створити власний двигунець під свої потреби на специфіку власних проєктів, це дозволяє впроваджувати нові цікаві рішення а також отримати повний контроль над тим що відбувається в процесі розробки і як все працює. Це не дивно адже ті двигунці які позиціонуються як універсальні та великі, такі як Unity 3D чи Unreal Engine, справді можуть надати великий спектр інструментів для розробки, їх може бути достатньо для поточних цілей, але в перспективі накладають певні обмеження. Тому краще використовувати власне розроблене середовище спеціально під свої цілі з можливістю досить швидко, легко і дешево інтегрувати додатковий функціонал в проєкт.

Метою дослідження є аналіз та застосування графічних бібліотек для створення власного графічного рушіїв.

Об’єктом дослідження є процеси обчислення шейдерів та спрайтів для ігрових рушіїв.

Предметом дослідження є існуючі засоби розробки ігрових рушіїв.

Для досягнення мети сформульовано наступні задачі:

- Дослідити сучасні ігрові рушії.
- Проаналізувати складові ігрових рушіїв.
- Дослідити технології для створення рушіїв.
- Провести аналіз готових рішень для рушіїв.
- Створити програмний засіб для реалізації функцій рушіїв.

1 АНАЛІЗ РИНКУ ІГРОВИХ РУШІЇВ

1.1 Аналіз існуючих аналогів

Ігровий рушій – це середовище розробки програмного забезпечення, яке призначенен призначене для створення відеоігор. Серед них можна виділити декілька найпопулярніших (рисунок 1.1).

Рисунок 1.1 – Графічний інтерфейс Unreal Engine 4

Unreal Engine 4 — це повний набір інструментів для розробки ігор, створених розробниками ігор для розробників ігор. Unreal Engine 3 був надзвичайно впливовим движком, здатним надати інструменти для розробки будь-якої гри, а тепер його наступник, Unreal Engine 4, активізував роботу. Незважаючи на те, що UE4 все ще знаходиться в розробці і все ще не завершений, він складається з надійного арсеналу інструментів і зручностей, визнаних одним з найбільш повних інструментів для розробки 3D і 2D [1] на багатьох платформах, таких як Windows, Mac, Xbox, Playstation, Android і браузер. Мова візуального програмування Blueprint є одним із багатьох доповнень до останнього Unreal Engine і поширюється від ігрового процесу

до мережевого програмування, дозволяючи розробникам швидко створювати прототип будь-чого, що спадає на думку. Ця функція одночасно зручна і легка в освоєнні, що дозволяє досягти результату навіть початківцю розробнику. UE4 також надає доступ до вихідного коду движка, надаючи розробникам можливість розширити будь-які попередні обмеження, знайдені в Blueprint. Розробник може створювати бібліотеки Blueprint за допомогою програмування на C++, які відповідають їхнім потребам, відкриваючи двері для тих, кому ця мова незручна, але потрібна функціональність, яка доступна через неї. Для забезпечення рівномірної гнучкості проєкти можна розробляти виключно на Blueprint або C++ або використовувати їх комбінацію.

UE4 Community є надзвичайно корисним ресурсом, який, можливо, можна вважати так само корисним для багатьох основних функцій самого двигуна. Будь-який розробник може скористатися перевагами магазину в стилі Google play/Apple App Market, який містить цілу низку проєктів, ігор, рівнів, креслень, арт-об'єктів, музики, аудіо, інструментів та плагінів, які розроблені іншими та легко доступні. Здається, там, де двигуна може не вистачати, в UE4 Community скоріш за все існує рішення[2]. Швидка публікація проблеми на форумі майже завжди призведе до зручного користувацького вирішення plug and play, яке збільшить функціональність рушія. Щоб надати додаткову підтримку цій уже активній групі, користувачі також можуть підключатися та говорити через службу обміну миттєвими повідомленнями, яку надає Epic. Unreal Engine 4 пропонує вражаюче освітлення, простоту використання та можливість втілити в життя будь-який дизайн за короткий час. Завдяки тому, що все в одному місці, UE4 стає потужним джерелом розвитку, оскільки користувачі швидко створюють неймовірно красиві світи, наповнені деталями, штучним інтелектом, цікавими механіками та інтерфейсом користувача без особливих зусиль або головного болю. Очевидно, що Epic провела інтенсивну роботу, щоб переконатися, що їхній рушій є простим у використанні, і з додаванням

розширених посібників, документації та демонстрацій, навіть ті, хто вперше вирішили поглибитись в індустрію розробки ігор можуть створити щось працююче в короткі терміни (рисунок 1.2).

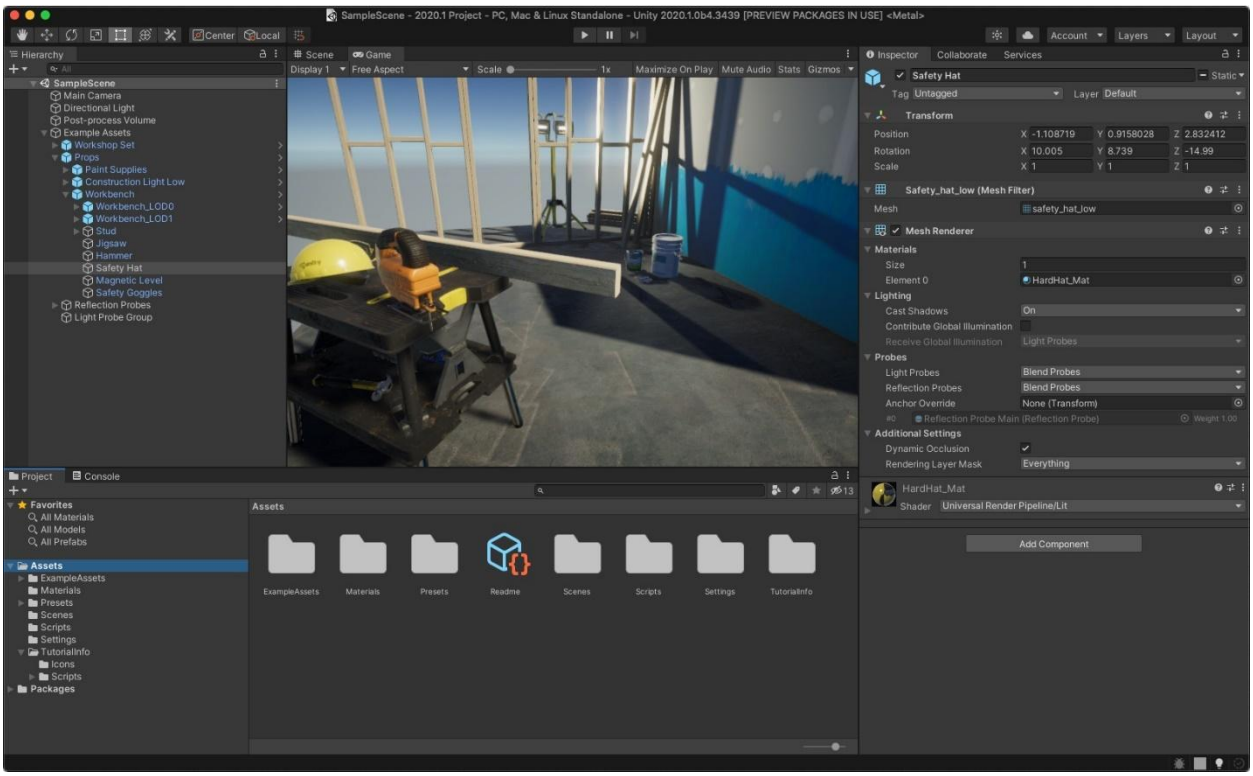


Рисунок 1.2 – Графічний інтерфейс Unity

Unity — це гнучка та потужна платформа розробки для створення крос-пластформових 3D та 2D ігор та інтерактивних програм. Це повноцінна екосистема для всіх, хто прагне побудувати бізнес на створенні високоякісного контенту та зв’язку зі своїми найбільш лояльними гравцями та клієнтами. Рушій Unity може похвалитися вражаючою кількістю лояльних прихильників та незалежних розробників, також до них приєднуються стабільні студії. Двигун спрощує процес розробки, задовольняючи всі типи архітектури проєктів. Користувачі можуть в будь-який момент втілити в життя декілька проєктів, що дозволяє їм оцінити та визначити, який проєкт буде остаточним. Unity також дозволяє користувачам зануритися в розробку

як 2D, так і 3D на кількох платформах, не згинаючи конвеєр для кожної з них[3]. Замість цього готову гру можна портувати на різні платформи, просто визначивши процес збірки без зміни коду гри. Хоча Unity не обмежується C#, він, безумовно, використовує цю мову високого рівня та рідну функціональність, яку вона пропонує, позбавляючи можливих витрат часу, пов'язаних із створенням власного. Навчитися створювати сценарії в Unity відносно просто, спостерігаючи за тим, як розробники швидко рухаються по функціональності інтерфейсу. Функції для кількох гравців і аналітика легко доступні в Unity і можуть бути інтегровані у проєкт за короткий час. Багато активів, включаючи фрагменти коду, завершені проєкти, аудіо, музику та розширення редакторів, також можна придбати в онлайн магазині двигунця. Поєднайте все вищесказане з дизайнером інтерфейсу користувача, набором анімацій, розширеними параметрами аудіо, фізикою, стандартним штучним інтелектом, високоякісною графікою та затіненнями, і ви отримаєте один з найкращих інструментів розробки ігор, здатний втілити в життя будь-яку ідею та дасть можливість розробникам гнучко розробляти свої проєкти (рисунок 1.3).

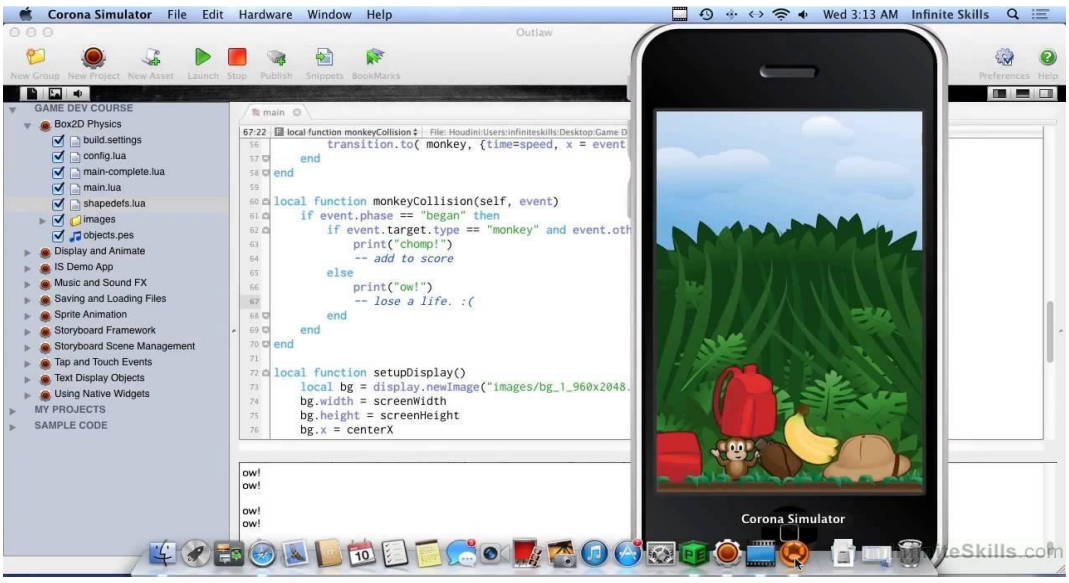


Рисунок 1.3 – Процес створення гри за допомогою Corona

API Corona забезпечує все, від анімації до мережі, за допомогою всього кількох рядків коду. Незалежно від того, створюються ігри чи бізнес-додатки, миттєво можна побачити зміни в Corona Simulator і можна дуже швидко виконувати ітерації. CoronaSDK — це ексклюзивний движок для 2D-розробки, який має на меті найбільш скоротити час розробки. CoronaSDK має велику бібліотеку API Lua, що дозволяє розробникам швидко та легко отримувати доступ до всіх його рідних функцій[4]. Використовуючи прості лінії, користувачі можуть активувати мережу, фізику та аудіо, не ставлячи галочки або оголошуючи включення. Corona також надзвичайно зручна для користувачів під час монетизації гри, оскільки забезпечує вбудовану підтримку покупок у програмі та рекламних мереж. Движок також дозволяє тестувати ігри в емуляторі, який імітує портативний пристрій із інструментами та налагодженням (рисунок 1.4).

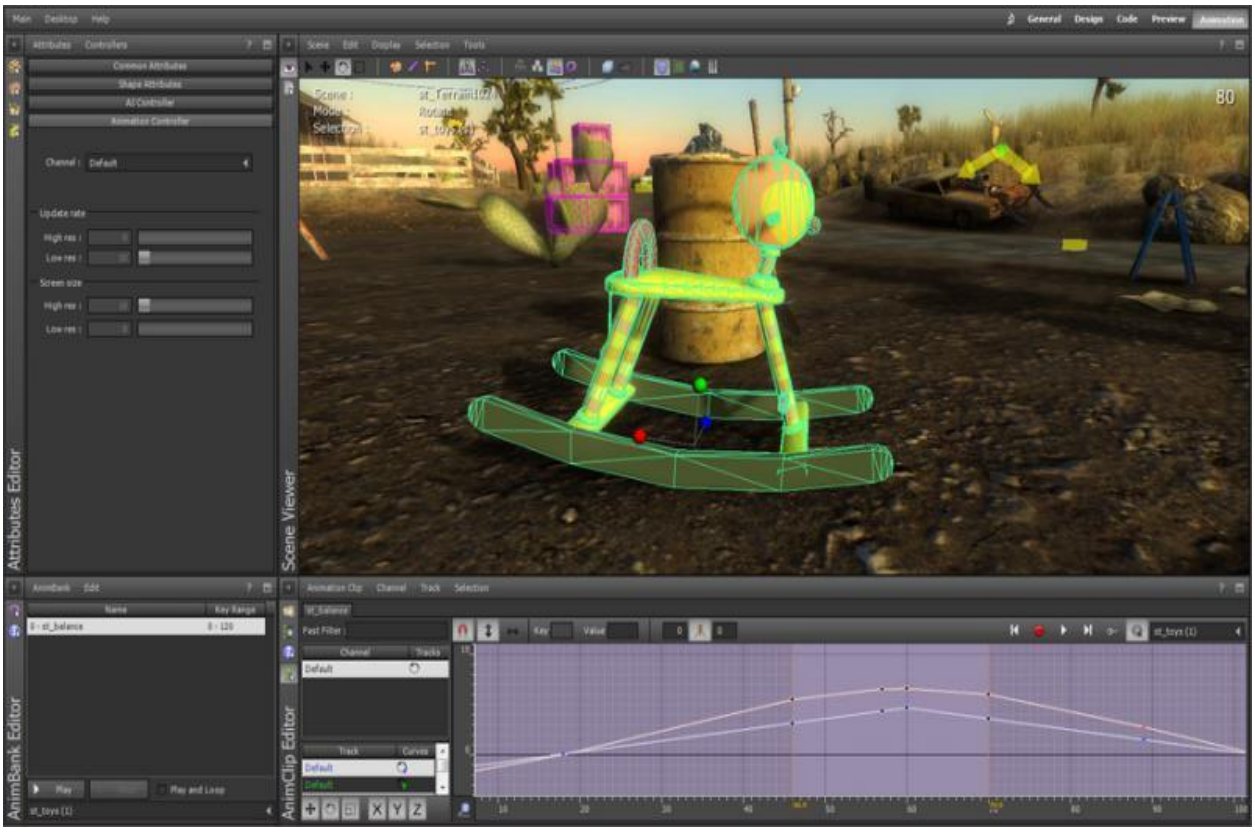


Рисунок 1.4 – Графічний інтерфейс Marmalade

Цей механізм також підтримує всі основні платформи для мобільних пристроїв, де його фокус залишається, однак незабаром вони впроваджують підтримку для Windows і Mac. Використовуючи CoronaSk, розробники можуть створювати неймовірні 2D-ігри з блискучим інтерфейсом користувача, мережею та монетизацією в найкоротші терміни. Marmalade — провідне крос-платформне рішення для розробників ігор. Ця платформа поєднує в собі SDK і сервіси. Marmalade полегшує розробку одного проєкту для кількох платформ, маючи можливість розгорнути єдину базу коду на всіх них. Користувачі можуть розробляти за допомогою Marmalade Core, який використовує розробку на C++, що гарантує максимальну гнучкість і продуктивність, або Marmalade Quick, який використовує Lua, прискорюючи розробку за допомогою потужної бібліотеки API, яка дозволяє розробникам виконувати широкий спектр власних методів, які ідеально підходять для швидкого створення прототипів[5].

Змішування та поєднання компонентів також є простим, будь то сторонній чи власний розробник, цей процес дуже потужний та гнучкий. Ці компоненти можуть зарядити проєкт, що розвивається, пропонуючи потужні API, які можна легко викликати та використовувати без проблем або головного болю. Рушій також має додаткові зручності, які ще більше полегшують процес розробки. Наприклад, симулятор відображає портативні пристрої та кнопкову упаковку, яка виконує всю важку роботу.

Marmalade також розміщує власний магазин активів, продаючи анімацію, 2D та 3D-активи, аудіо, послуги та проєкти. Проєктами також можна керувати в одному зручному центрі, що дозволяє розробникам стежити за навчальними посібниками, переглядати демонстрації та створювати пакети (рисунок 1.5).

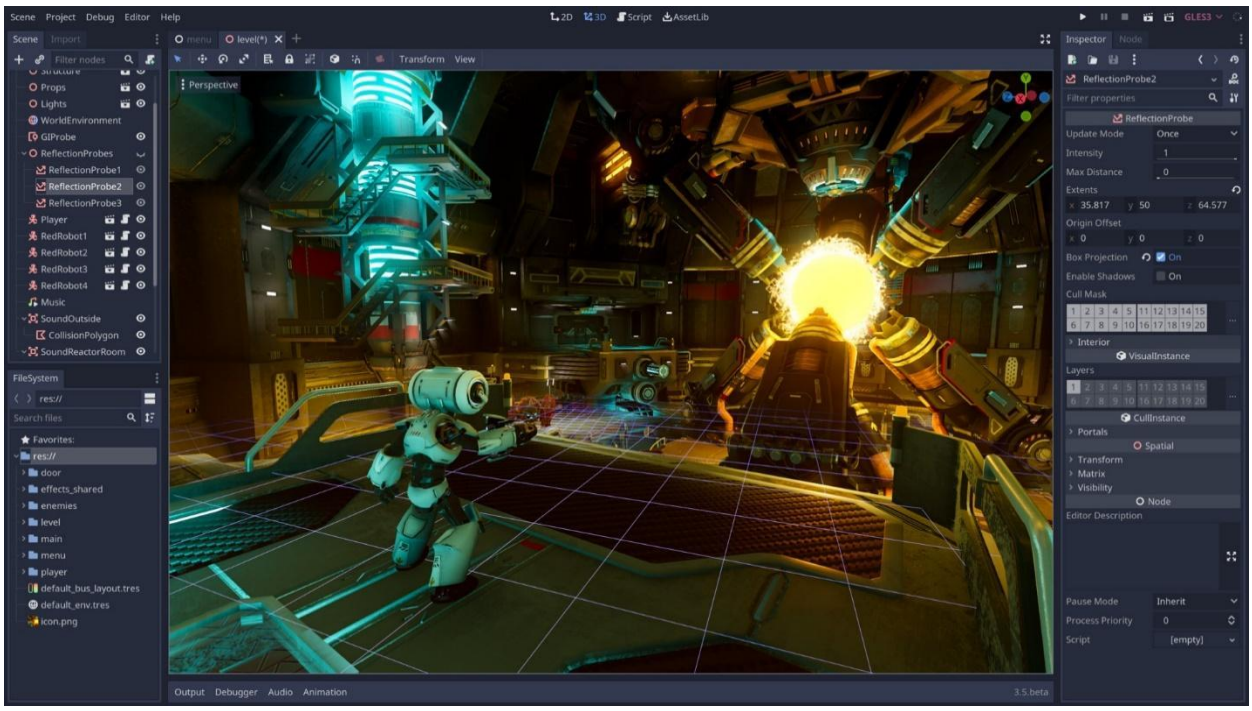


Рисунок 1.5 – Графічний інтерфейс Godot

Godot — відкритий крос-платформовий 2D та 3D гральний рушій під ліцензією MIT, що розробляється співавторством Godot Engine Community. До публічного релізу у вигляді відкритого ПЗ рушій використовувався всередині деяких компаній Латинської Америки. Оточення розробника працює на Windows, Linux, OS X, BSD і Haiku та може експортувати ігрові проєкти на ПК, консолі, мобільні та веб платформи. Задача двигунця — бути максимально інтегрованим та самодостатнім середовищем для розробки ігор. Середовище дозволяє розробникам створювати ігри з нуля, не користуючись більше ніякими інструментами окрім тих, що потрібні для створення ігрового контенту (елементи графіки, музичні треки тощо)[6]. Процес програмування також не потребує зовнішніх інструментів (хоча при необхідності використовувати зовнішній редактор, це зробити досить легко). Загальна архітектура рушія побудована навколо концепції дерева з наслідуваних "сцен". Кожен елемент сцени, в будь-який момент сам може стати повноцінною сценою. Тож при розробці можна легко змінювати повністю всю архітектуру проєкту, розширювати її елементи в будь-яку сторону та

працювати із комплексними сценами на рівні простих абстракцій. Всі ігрові ресурси, від скриптів до графічних ассетів та ігрових сцен, зберігаються в теці проєкту як звичайні файли, та не є частиною складної бази даних проєкту[7]. Ресурси, що не представляють собою комплексних даних, зберігаються у простих текстових форматах (наприклад скрипти та сцени, на відміну від моделей та текстур). Ці рішення дозволяють значно спростити різним командам розробників працювати з системами керування версій.

1.2 Вибір програмного забезпечення

Всі двигунці виконують декілька ключових функцій:

- Ініціалізація системи – відкриває вікно, загружає дані.
- Рендеринг – практично всі ігри використовують графіку, рушій відповідає за її відображення на екрані, шейдери, буфери вершин, багатобуферну чи однобуферну промальовку(залежить від типу графіки).
- Управління ігровими об’єктами та сценаріями – один із ключових механізмів в рушії, він управляє логікою гри.
- Аудіо – звукові ефекти та музика розділені, хоч і належать до цього розділу однаково. Основні необхідні функції це запуск і зупинка звукових циклів та відтворення звукових ефектів від початку і до кінця. Цим аудіо не обмежується але цього досить для основної роботи з аудіоматеріалами.
- Завантаження та управління файлами – файли загружають всі ігри, ця підсистема необхідна для автоматичного та комфортного запуску готового проєкту.

Серед даного списку варто почати з основи, а саме з ініціалізації та графіки. Основну підсистему можна написати використовуючи готові бібліотеки для роботи з графічними ядрами та процесором. Залежно від операційної системи вибір буде коливатись, бо різні технології зазвичай створюються під певні умови, що можуть відрізнятись один від одної, тому

часто такі технології між собою не сумісні і вимагають додаткових зусиль та проміжного програмного забезпечення для коректної роботи. Найпопулярніші бібліотеки для поставленої задачі будуть:

− DirectX — це інтерфейс програмування прикладних програм (API), розроблений Microsoft для Windows і Xbox. Графічний API — це посередник, який полегшує надсилання інструкцій з програмного забезпечення на обладнання всередині ПК. На початку розвитку комп'ютерів інструкції йшли безпосередньо до апаратного забезпечення. однак, оскільки ігри стають все складнішими, а заходи безпеки більш прямими, API є основою, яка вказує графічному процесору, що робити.

− OpenGL (Open Graphics Library) — це стандартний інтерфейс прикладних програм комп'ютерної індустрії для визначення двовимірних і тривимірних графічних зображень. До OpenGL будь-яка компанія, яка розробляла графічний додаток, зазвичай повинна була переписати його графічну частину для кожної платформи операційної системи, а також була обізнана з графічним обладнанням. За допомогою OpenGL програма може створювати ті самі ефекти в будь-якій операційній системі, використовуючи будь-який графічний адаптер, що підтримує OpenGL.

Оскільки OpenGL є крос-платформовим то це дає більше простору для використання написаного на його основі продукту. Також серед переваг можна виділити зручність використання та легкість інтегрування в проєкт. Основні можливості:

− Геометричні та растрові примітиви. На основі геометричних та растрових примітивів будуються всі об'єкти. З геометричних примітивів бібліотека надає: крапки, лінії, полігони. З растрових: бітовий масив (bitmap) та образ (image).

− Використання сплайнів. B-сплайн використовуються для малювання кривих по опорних точках.

− Видові та модельні перетворення. З допомогою цих перетворень можна

розташовувати об'єкти у просторі, обертати їх, змінювати форму, і навіть змінювати становище камери з якої ведеться спостереження.

− Робота із кольором. OpenGL надає розробнику можливість працювати з кольором у режимі RGBA, тобто червоний-зелений-синій-альфа, або використовувати індексний режим, де колір обирається з палітри.

− Видалення невидимих поверхонь та ліній і Z-буферизація.

− Подвійна буферизація. OpenGL надає як подвійну, так і одинарну буферизацію. Подвійна буферизація використовується у тому, щоб усунути мерехтіння при зміні кадрів. Зображення кожного кадру спочатку малюється в другому (невидимому) буфері, а потім, коли кадр повністю намальований, весь буфер відображається на екрані.

− Накладання текстури. Дозволяє надавати об'єктам реалістичності. На об'єкт, наприклад кулю, накладається текстура (просто якесь зображення), внаслідок чого наш об'єкт тепер виглядає не просто як куля, а як різнокольоровий м'ячик.

− Згладжування. Згладжування дозволяє приховати ступінчастість, властиву растровим дисплеям. Згладжування змінює інтенсивність і колір пікселів біля лінії, при цьому лінія виглядає на екрані без жодних сходинок.

− Освітлення. Дозволяє ставити джерела світла, їх розташування, інтенсивність і т.д.

− Атмосферні ефекти. Наприклад, туман, дим. Все це також дозволяє надати об'єктам або сцені реалістичність, а також відчути глибину сцени.

− Прозорість об'єктів.

− Використання списків зображень.

Незважаючи на те, що бібліотека OpenGL (GL) надає дуже багато можливостей для моделювання та відтворення тривимірних сцен, деякі з можливих функцій, які необхідні для роботи з графікою, відсутні в стандартній бібліотеці. Наприклад, щоб задати положення та напрямок камери, з якої спостерігатиметься сцена, потрібно власноруч розраховувати

модельну матрицю, а це важко та потрібні знання та навички. Тому для OpenGL є так звані допоміжні бібліотеки.

Перша бібліотека називається GLU. Ця бібліотека стала стандартом і постачається разом із головною бібліотекою OpenGL. До її складу увійшли більш складні функції, наприклад для того, щоб визначити диск або циліндр знадобиться тільки одна команда. Також у бібліотеку увійшли функції для роботи зі сплайнами, реалізовано додаткові види проекцій та операцій над матрицями.

Наступна бібліотека, що також широко використовується - це GLUT. freeglut — це безкоштовна альтернатива програмному забезпеченню з відкритим вихідним кодом бібліотеці OpenGL Utility Toolkit (GLUT). Спочатку GLUT був написаний Марком Кілгардом для підтримки зразків програм у другому виданні OpenGL «RedBook». З тих пір GLUT використовувався в широкому спектрі практичних застосувань, оскільки він простий, широко доступний і дуже портативний. GLUT виконує всі специфічні для системи роботи, необхідні для створення вікон, ініціалізації контекстів OpenGL та обробки подій введення, щоб забезпечити портативні програми які використовують OpenGL. freeglut випускається за ліцензією X-Consortium.

Є ще одна бібліотека схожа на GLUT, вона називається GLAUX. Ця бібліотека розроблена фірмою Microsoft для операційної системи Windows. Вона багато в чому схожа на бібліотеку GLUT, але трохи відстає від неї за своїми можливостями. І ще один недолік полягає в тому, що бібліотека GLAUX призначена тільки для Windows, тоді як GLUT підтримує багато операційних систем.

Існують інші додаткові бібліотеки для OpenGL. Всі вони додають щось своє або орієнтовані на якусь платформу. Наприклад, існує така бібліотека як GLX - це розширення для X Windows, що дозволяє використовувати в X

Windows OpenGL. GLX надає не лише локальний рендеринг, а й рендеринг через мережу.

1.3 Постановка задачі

Для прикладу двигунець можна використати щоб створити гру, яка використовує нову технологію, яку наразі не підтримують жодні інші движки, або яку не можна легко зробити для підтримки в їхньому поточному стані. Це може означати якусь масштабну симуляцію, яка вимагає деякого якусь спеціальну річ, яка не вписується в жодну існуючу бібліотеку, або бажання націлитись на дивне обладнання, яке поточні двигуни не підтримують, або будь-яке число інших речей. Це вагома причина зробити свій власний двигун, тому що в подібних випадках немає іншого виходу. Також з його допомогою можна оптимізувати свій робочий процес створення ігор. Якщо не потрібні всі функції, включені в комерційний ігровий движок, і є можливість зробити свій конвеєр ресурсів або ж редактор рівнів. Це вимога для створення власного двигуна, якщо не спеціалізуватись на ньому та не розуміти для чого конкретно потрібно створювати двигунець, необхідно розібратись для початку, чи доцільно цим займатись.

Власний двигунець створюють компанії або невеликі групи розробників щоб не залежати від чужих технологій у довгостроковій перспективі. Стимули та цінності такої компанії, як Unity або Epic, не завжди будуть співпадати з іншими, і коли є бажання контролювати свої власні технології, можливість самостійно виправляти помилки замість того, щоб чекати й сподіватися. Знаючи, що оновлення не зруйнує повністю поточний проєкт, можна займатись розробкою з більшим комфортом. Якщо дозволити витрати на розробку власних технологій, в довгостроковій перспективі це буде досить ефективно, тому що не доведеться постійно міняти код залежно від примх гігантських компаній.

Мотивація створення власного двигуна здебільшого пов’язана з новими технологіями та спеціалізацією. Наприклад, жоден із існуючих механізмів не підтримував імпорт флеш-анімації, тому єдиним варіантом було зробити це самому. Надзвичайно приємно мати можливість просто вставляти файли .swf у папку ресурсів і миттєво мати доступні анімації для використання в коді гри, не потребуючи жодних проміжних кроків для їх експорту в аркуші спрайтів чи щось подібне. Це не єдині причини, чому може виникнути необхідність створити власний ігровий движок. У нього є маса переваг і недоліків, і те, чи переважать плюси над мінусами, залежить багато від того, який досвід має розробник в створенні ігор та з низькорівневим кодуванням. Досить зручно мати власний програмний продукт, і приємно, що не потрібно постійно шукати в пошукових системах підручники, які можуть бути застарілими з реалізаціями того, як щось робити всередині проєкту. Зручно мати можливість налагодити внутрішні елементи гри, якщо щось піде не так. Але це також може погано, якщо зробити пару поганих варіантів дизайну, і все не працюватиме повністю, а рішень в інтернеті вже знайти не вдасться, щоб допомогти вирішити проблему. Потрібно добри оцінювати всі плюси та мінуси перш ніж братись за розробку подібного програмного забезпечення.

Насправді існує багато відмінностей ігрових двигунцях. Деякі з них є лише основою для відображення графіки, але роблять для полегшення розробки досить мало, це Flash або Pico-8. Деякі з них є в основному цілими іграми самі по собі або, принаймні, дуже спеціалізованими для певного жанру, додаючи масу загальної логіки гри в сам движок, наприклад, RPGMaker чи Ren’Py. Самі ігрові движки також, як правило, будуються на основі фреймворків нижчого рівня, таких як SDL і OpenGL, і включають багато бібліотек спеціального призначення для таких речей, як аудіо, фізика, математика та все, що є корисним для таких задач. Створення спеціального

ігрового движка не означає написання кожного його елементу самостійно, особливо щось таке стандартне і корисне, як SDL.

Складання правильного набору існуючих бібліотек для необхідного варіанту використання також є частиною створення двигуна, і існують бібліотеки для майже всіх систем, які вам можуть знадобитися. Найнеобхідніше, це мінімум, який знадобиться, перш ніж почати створювати гру:

− Ініціалізація системи: відкриття вікна, отримання контексту OpenGL/DirectX/Vulkan, ініціалізація аудіо. SDL впорається з усім цим, тому можна просто використати SDL. Насправді, SDL на даний момент є промисловим стандартом для користувацьких двигунів, немає причин робити цю частину самостійно.

− Контроль синхронізації кадрів:якщо потрібно, щоб гра працювала зі швидкістю 60 кадрів в секунду, необхідний якийсь таймер і цикл, який контролює, коли відбуваються оновлення та візуалізації.

− Вхід: Необхідно мати можливість реагувати на натискання кнопок. Існує багато різних способів реагування на натискання кнопок, можливо, просто потрібно отримати можливість запитати поточний стан кнопки, або зареєструвати події, це не має значення. SDL повідомить про введення кнопок, якщо вже використовується це для ініціалізації системи, також необхідно використовувати його тут. На додаток до цього можна створити дійсно потужну та гнучку систему введення, але спочатку не потрібно робити нічого, крім основ.

− Відтворення: більшість, ймовірно, принаймні 75% ігор, використовують графіку якимось чином, і це абсолютно те, за що має бути відповідальним двигун. Якщо розробляється 2D-гра, мінімальний інструмент візуалізації повинен мати можливість відображати на екрані прості текстуровані квадрати. Шейдери, вершинні буфери, цілі візуалізації, сітки, матеріали та багато іншого - все це чудово, але вони можуть бути

зробленими пізніше, по мірі необхідності. Якщо потрібно скористатись OpenGL чи Vulkan чи чим завгодно, можна абсолютно не використовувати власними функціями рендереру, але не має нічого поганого в використанні наявної бібліотеки, як-от Ogre3D, для покриття візуалізації. Це цілком залежить цілей і потреб, а також від того, які проблеми насправді необхідно вирішити.

− Математика та інші утиліти: це те, що ймовірно потрібно в системі мати як бібліотеку утиліт, до якої мають доступ як движок, так і код гри. Плюс будь-які інші випадкові корисні функції та формули, які виявляться під час розробки. STB — це неймовірно чудовий ресурс для випадкових корисних функцій, які можуть знадобитися.

Також кілька систем, які є необхідними, але вам не варто додавати їх до тих пір, поки дійсно вони не знадобляться:

− Управління ігровими об’єктами і сценою: кодування всього в одній великій функції оновлення насправді не так погано, як здається, але якщо гра почне ставати хоч трохи складнішою, знадобиться якась система для обробки окремих ігрових об’єктів і їх колекції. У певному сенсі це виявляється найважливішою системою у двигуні, оскільки саме це визначає логіку гри. Якщо об'єкти великі спадкові дерева чи вони складаються з менших компонентів, або ж використовується ECS, потрібно чітко визначити, на які типи подій вони реагують, як запитується інший об’єкт для взаємодії. Чи добре регулюється розташування в пам’яті. Для цього існують деякі бібліотеки, особливо для чистої ECS, але оскільки це система, яка найбільше впливає на те, як виглядає код гри, мабуть більш доцільним буде принцип «зроби сам». Покладатися на існуюче рішення змусить постійно думати про те, як вписати свою логіку гри в цю форму. Це рівно протилежне тому що б варто було робити насправді. Потрібно щоб движок був розроблений так, щоб підтримувати те, як виражається логіка гри, а не навпаки.

− Аудіо: звукові ефекти та музика тут є окремими системами, хоча обидві зведені під «Аудіо». Основні функціональні можливості, які потрібні, це можливість запускати та зупиняти аудіоцикли (музику), а також можливість відтворювати окремі звукові ефекти повністю. У аудіо є набагато більше, ніж просто це, але можна піти дуже далеко, використовуючи лише основи. Погано, що стандартні аудіо-фреймворки FMod і Wwise є комерційними та мають купу ліцензійних обмежень, але багато з відкритих вихідних кодів дуже незручні у використанні. FAudio є досить простим і чудовим, і можна використати його як основу для побудови складнішої поведінки.

− Завантаження та керування файлами: усі ігри мають завантажувати файли. Потрібен спосіб завантаження файлів. Було б не дуже добре перезавантажувати/декодувати файли, які вже були завантажені, тому потрібен якийсь базовий менеджер, який оброблятиме це все. Можливо, потрібно буде створити підтримку модів або динамічне завантаження або вивантаження чи щось інше, якщо у є основи і файли завантажуються лише через файловий менеджер, можна легко додати будь-які інші функції пізніше. Немає необхідності додавати це відразу, оскільки можна використовувати вбудоване завантаження файлів як заповнювач, але в кінцевому підсумку було б непогано мати систему менеджера файлів/ресурсів, як тільки використовуватимуться файли частіше.

Це основний набір систем, які знадобляться, щоб мати те, що можна вважати ігровим движком. Інші системи, такі як фізика, серіалізація, анімація та інтерфейс користувача тощо, насправді є необов’язковими. У більшості движків вони є, тому що вони досить поширені, що варто включити, але вони не потрібні для створення гри.

2 ТЕХНІЧНІ АСПЕКТИ РЕАЛІЗАЦІЇ ІГРОВИХ РУШІЇВ

2.1 Визначення основних елементів двигунця

Визначення двигуна може бути неоднозначним. Це може бути механізм додатків, керований такими представленнями, як Apple UIKit, React Native або сучасна веб-платформа, механізм візуалізації без складної логіки для пристроїв введення та створений для досліджень рендерингу в реальному часі, таких як NVIDIA Falcor або Microsoft MiniEngine, шейдерна іграшка, програма для демонстрації, або масивний який складається з різноманітних компонентів, таких як Unreal, Unity, Godot.

Ігрові движки складаються з систем вищого рівня, побудованих як:

− Ієрархія програми зі сценою акторів або колекцією компонентів перегляду та компонентами цих акторів, такими як логічні контролери, сітки, текстури зображень, аудіо тощо. Це часто називають шаблоном проєктування акторів, де незалежні актори виконують свої власні завдання.

− Об’єкти Singletons, доступні в усій програмі, які зберігають стан або керують загальними системами, такими як введення, стан програми, інтерфейси користувача.

− Проміжне програмне забезпечення, яке може підписатися на інше проміжне програмне забезпечення, наприклад, проміжне програмне забезпечення Renderer, яке підписується на проміжне програмне забезпечення Windowing, і системи низького рівня, такі як актори, щоб підписатися на них для доступу до стану програми.

Акторів можна створити за допомогою:

− Класичні ієрархії, де такі компоненти, як перетворення, сітки тощо, описуються через успадкування.

− Система компонентів сутності, де декларативні компоненти зберігають стан своїх даних і мають зовнішні системи, такі як проміжне програмне забезпечення або синглтони для обробки цього стану.

Або навіть комбінація цих проєктів, хоча використання двох різних методів може в кінцевому підсумку зробити код більш складним і важким для розуміння. Зрештою, ці обмеження та методи організації служать лише способом розділити проблеми та полегшити розуміння вашої програми в цілому.

Синглтони неймовірно корисні при розробці додатків, оскільки будь-яка система може негайно отримати доступ до даних із синглтона. Unity інтенсивно використовує синглтони, що можна побачити, наприклад, у тому, як вони обробляють свій синглтон введення (рисунок 2.1).

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour
{
    void Update()
    {
        if (Input.GetKeyDown((KeyCode.Space))
        {
            print("space key was pressed");
        }
    }
}
```

Рисунок 2.1 – Синглтон введення в Unity

ImGui використовує синглтон для малювання графічних інтерфейсів користувача безпосереднього режиму (рисунок 2.2).

```
ImGui::Text("Hello, world %d", 123);

if (ImGui::Button("Save"))
{
    // 💾 Save the application
}

const char* buf[1024];
ImGui::InputText("string", buf, IM_ARRAYSIZE(buf));

ImGui::SliderFloat("float", &f, 0.0f, 1.0f);
```

Рисунок 2.2 – Відмальовка графічного інтерфейсу ImGui

Одним з підходів до моделювання систем високого рівня є використання проміжного програмного забезпечення, ідея полягає в тому, щоб зберегти ядро рушія відносно невеликим і дозволити проміжному програмному забезпеченню заповнювати прогалини в стані, які були б відсутні за замовчуванням. Актори можуть підписатися на проміжне програмне забезпечення, якщо воно існує, і звідти зчитувати дані з вікон ОС, стан графічного інтерфейсу, або все, що розробники вважають за потрібне, щоб передати до акторів.

Проміжне програмне забезпечення буде підписане суб'єктами, яким потрібен доступ до стану різних систем, на яких запущена програма. Переваги цього полягають у тому, що глобальний простір імен більше не використовується, що призводить до більш ізольованого коду, але це відбувається за рахунок накладних витрат на підписку з боку акторів, на додаток до локальних покажчиків на передплачені дані. Інженер також має певний рівень «технічних витрат», оскільки більш ізольований код означає усвідомлення того, на яке проміжне програмне забезпечення їм потрібно буде підписатися, щоб вирішити свої проблеми. Також проміжне програмне забезпечення відповідає шаблону проєктування абонентів, і, таким чином, існує початкова вартість налаштування для кожного актора, який хоче підписатися на проміжне програмне забезпечення поза ієрархією сцени (порядку $O(N)$). Хоча цю вартість можна пом'якшити, розв'язуванням покажчика на основі constexpr під час компіляції, передплатники все одно повинні включати код.

Архітектура компонентів сутності дозволяє сутності утримувати компоненти, які зазвичай містять лише дані. Ці дані потім обробляються системами високого рівня, які керують візуалізаціями, зіткненнями, пошуком шляху чи фізикою. Це дуже добре вписується в ідею проєктування, орієнтованого на дані. В основному, в інтересах програми згрупувати подібні дані разом, щоб допомогти кеш-пам'яті ЦП рідше хибити пропускаючи дані.

Графічний движок досить великий і складний, щоб на нього поширювалися правила проєктування великої бібліотеки з використанням об'єктно-орієнтованого програмування. Об’єкти для управління, що важливо мати основний набір автоматичних служб, на які можуть використати розробники. Поліморфізм забезпечує абстрагування функціональності. Виклик поліморфної функції можна зробити незалежно від справжнього типу об’єкта, що викликає. Але бувають випадки, коли потрібно знати тип поліморфного об’єкта або визначити, чи вибраний тип походить від заданого, наприклад, для безпечного введення регістру вказівника базового класу на вказівник похідного класу. Цей процес називається динамічним приведенням типів. Об'єктно-орієнтована система з одним успадкуванням складається з набору спрямованих дерев де вузли дерева представляють класи, а гілки дерева представляють дочірні класи.

2.2 Архітектура рушія

При створені проєкту був використаний одиночний клас «Game», який контролює запуск і завершення роботи всіх менеджерів:

- DataManager.
- GLFWManager.
- InputManager.
- LightManager.
- SceneManager.
- TimeManager.

Кожен менеджер не був єдиним, а динамічно викликався, що свідчило про те що конструктор викликається лише один раз. Це означало, що лише одна логічна зміна повинен бути в сегменті даних замість цілого класу. Синглтонна конструкція дозволила отримати доступ до будь-якого з менеджерів з будь-якої точки мого коду, який включав заголовний файл. Це усунуло необхідність передавати всі ці менеджери як аргументи по всьому

коду. Щоразу, коли потрібно було викликати функцію менеджера, достатньо було просто написати «Game->get()::time_manager.time_elapsed» (рисунок 2.3).

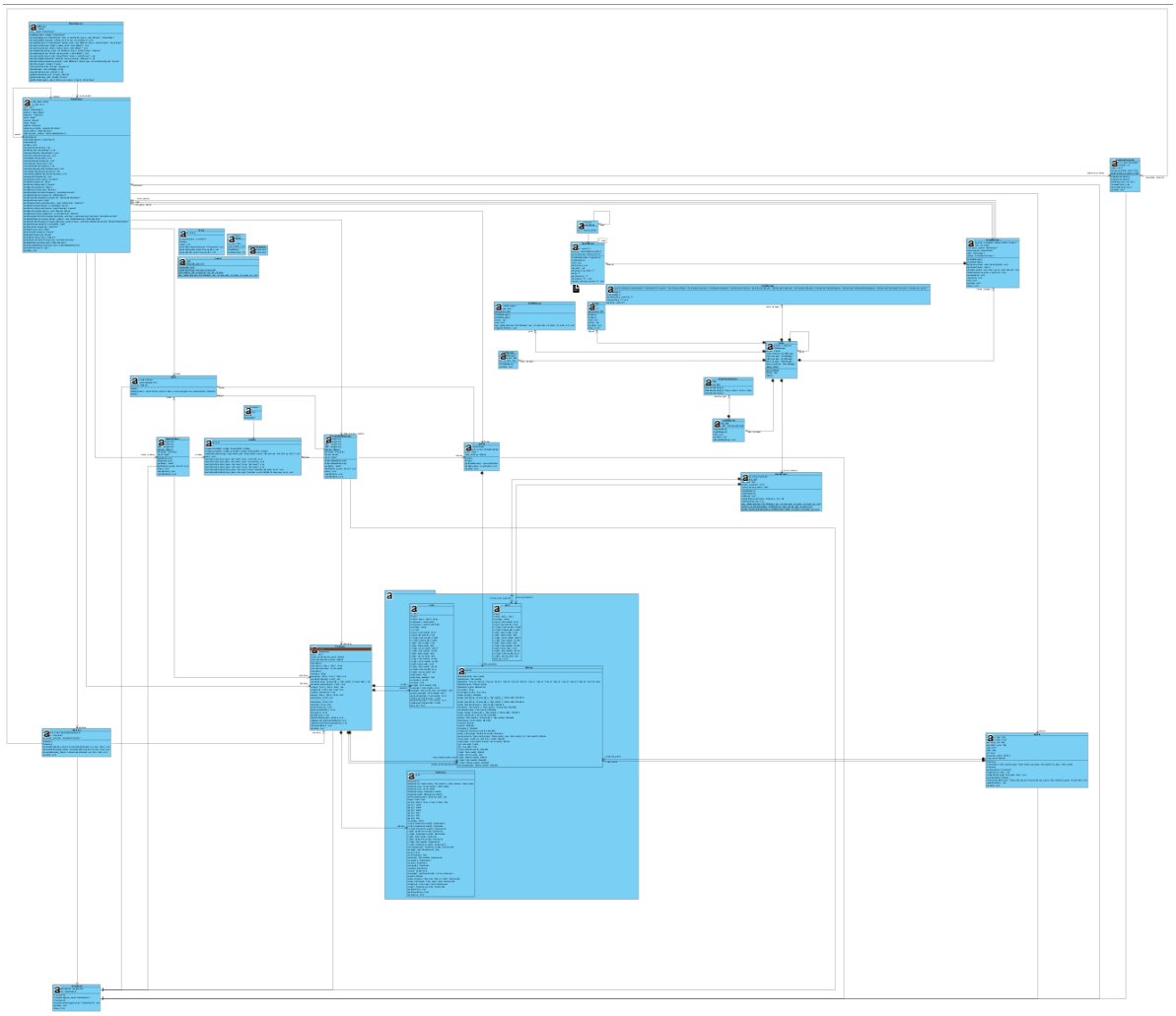


Рисунок 2.3 – Схема класів проєкту

Щоб дозволити імпортувати різні формати моделей було використано Assimp для аналізу файлів. Потім проаналізувавши внутрішні структури даних Assimps. Assimp не є ідеальною програмою, тому з деякими моделями, які намагався імпортувати, було багато проблем. Якби довелося це зробити знову, варто було б розробити сучасний формат моделі. Більша частина роботи була виконана тут, завантажуючи модель, щоб заповнити систему GameObject-Component інформацією про скелетну анімацію (рисунок 2.4).

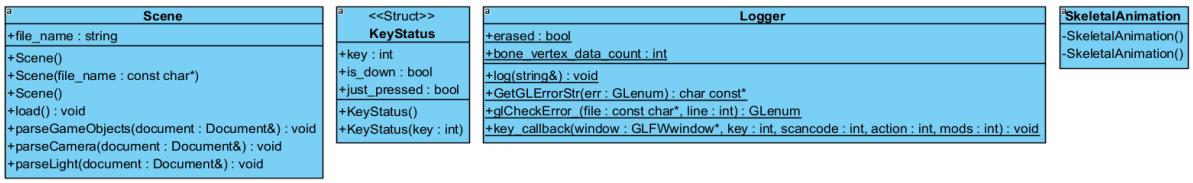


Рисунок 2.4 – Детальніша схема незалежних класів

Скелетна анімація концептуально проста, але реалізацію може бути складно описати. Модель, яка використовує скелетну структуру, є ієрархією вузлів. Кожне з'єднання має Transform відносно свого батьківського елемента та компонент Skeleton, який зберігає ключові кадри анімації для цього з'єднання. Воно може мати від нуля до багатьох сіток (рисунок2.7).

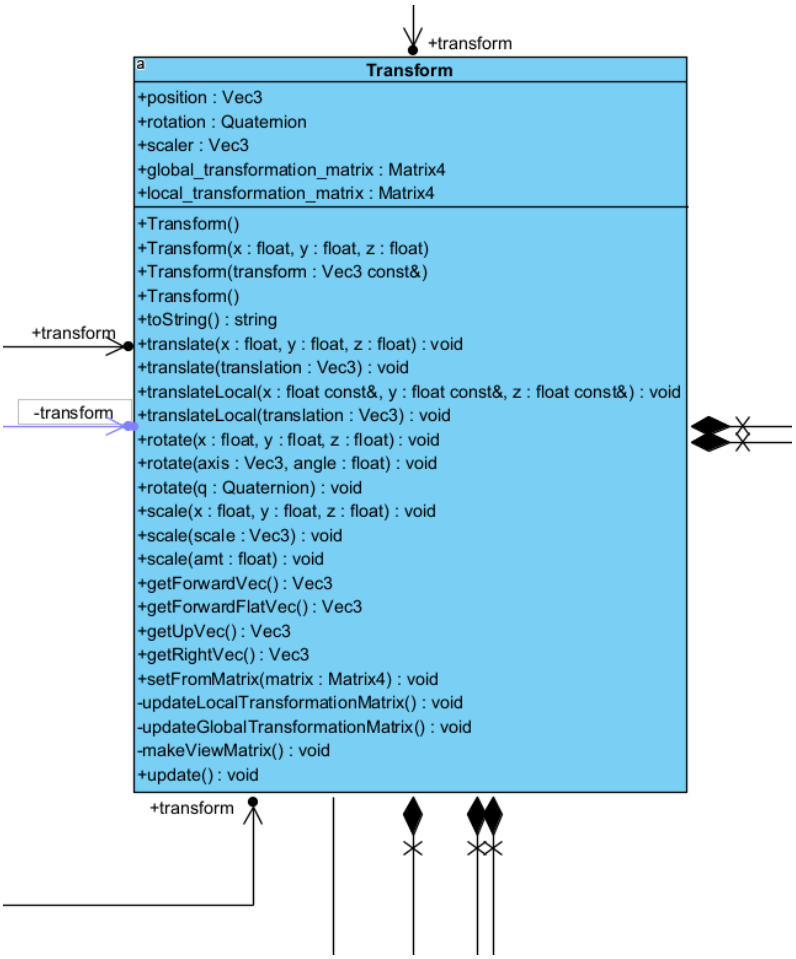


Рисунок 2.7 – Детальніша схема класу Transform

Кожна сітка є GameObject і дочірнім до спільного GameObject. Меші мають Transform відносно спільного GameObject. У них також є Mesh, SkeletalMeshRenderer і компонент Bones. На кожну сітку впливає лише певна кількість суглобів у скелетній ієрархії.

Компонент Bones містить вектор кісток. Кожна кістка містить вказівник на зазначений суглоб. Компонент Bones також містить вектор VertexBoneData.

Це говорить нам, які кістки впливають на певну вершину і наскільки. Якщо на вершину впливають кілька суглобів, вплив кожного суглоба або «вага», буде сумуватись до 1.

Загальний консенсус полягає в тому, що максимум чотири кістки впливатимуть на одну вершину.

Матриця кісток для кожної кістки обчислюється для кожного кадру таким чином:

− Кожна модель має AnimationController, який контролює, яка анімація відтворюється, а також містить таймер для поточної анімації.

− Оновлює структуру скелета.

− Компонент Skeleton перевіряє таймер AnimationController, щоб визначити, між якими двома ключовими кадрами LERP (положення, масштаб) і SLERP (поворот).

− Отримане положення, масштаб, і обертання замінює компонент Transform.

− Остаточна матриця кісток розраховується для кожної кістки.

− Матриця перетворення сітки * Матриця трансформації кістки * Матриця пози зв'язування кістки.

Кожна сутність у ігровому движку називається GameObject (рисунок 2.5).

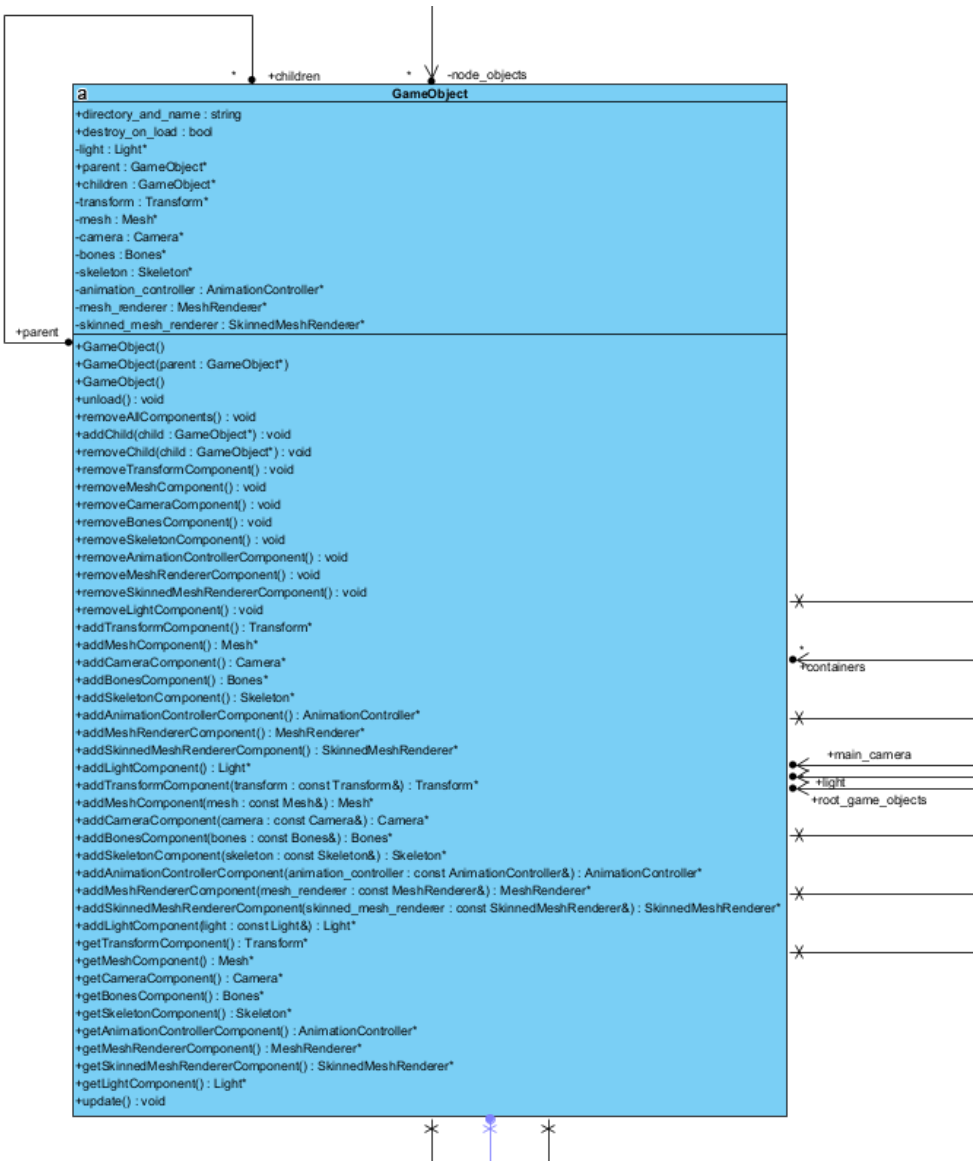


Рисунок 2.5 – Детальніша схема основного класу ігрових об’єктів

GameObject - це порожня оболонка, де містить одного - багато компонентів. Компоненти включають:

- Transform – Required.
- Mesh.
- Camera.
- Bones.
- Skeleton.
- AnimationController.
- MeshRenderer.

− SkinnedMeshRenderer.

Клас DataManager містив PoolAllocators для кожного типу компонента. Ця система надзвичайно корисна як з організаційної, так і з точки зору продуктивності. Щоразу, коли я потрібно було відображати на екрані модель, достатньо було просто перебрати усі MeshRenderers, які послідовно зберігалися в пам’яті в PoolAllocator, щоб збільшити продуктивність кешу. ModelLoader є узагальненням WaveFrontLoader, який надає можливість розробляти завантажувачі для інших форматів файлів у майбутньому, завжди відповідаючи одному і тому ж інтерфейсу. GameObject також може прийняти зображення, яке буде відображено на моделі. Якщо деякі поля створені в такому інструменті, як Blender, правильно розміщені над моделлю та експортовані в окремий файл Wavefront, GameObject може підібрати їх за допомогою класу BoundingBoxes і забезпечити деяке базове виявлення колізій. Renderer може відображати моделі, надані GameObjects. Він використовує клас Image або для утримання текстур, які будуть зіставлені з моделями, або для візуалізації у вигляді окремих прямокутників. Ці прямокутники самі працюють як об’єкти сцени і можуть використовуватися для представлення землі, неба, заставки тощо. Клас Text можна використовувати для завантаження тексту та відображення його на екрані за допомогою Renderer. Класи Exception і Logger використовуються в усьому механізмі для повідомлення про помилки та ведення журналів, як випливає з їх назв. Їх також можна використовувати в коді кожної гри, що розробляється разом із движком. Незважаючи на те, що я уникав використання великої кількості заздалегідь розроблених ігрових засобів, деякі залежності від бібліотеки були необхідні. Ось як виглядала б типова гра стосовно движка та цих компонентів, на які посилаються.

SceneManager містить вектор ігрових об’єктів верхнього рівня. Коли гра оновлюється, викликається кожна функція оновлення GameObjects верхнього рівня. Цей GameObject оновлює своїх дочірніх елементів у

глибину. Компоненти трансформації є відносно батьківських GameObjects. Ієрархія трансформацій є важливою частиною скелетної анімації (рисунок 2.6).

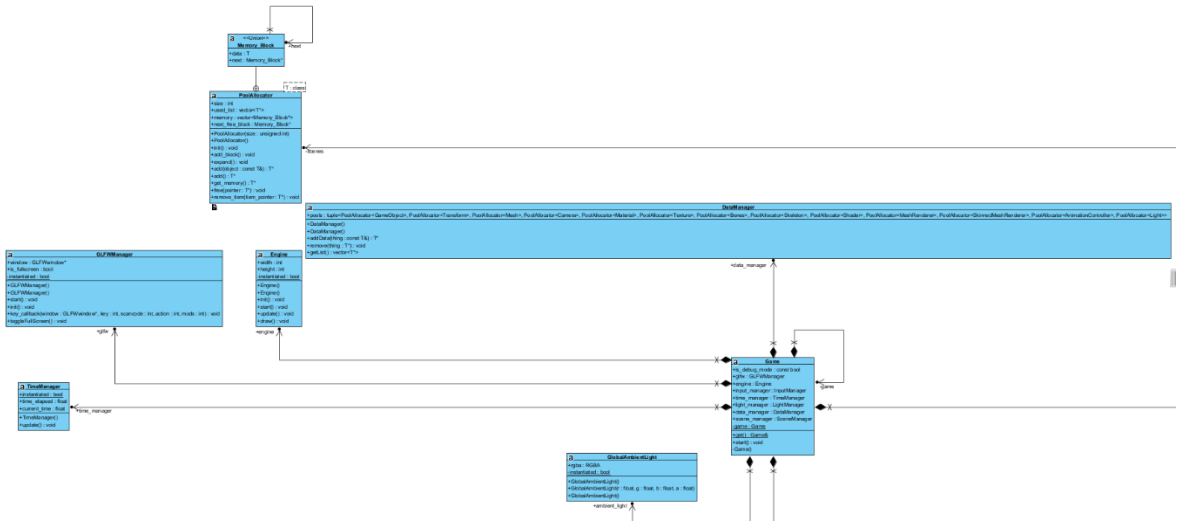


Рисунок 2.6 – Детальніша схема класів для передачі даних та розподільника пам’яті

Розподільник має шаблони для будь-якого типу T. Спочатку він відокремлює частину пам’яті.

Пам'ять є вектором типу об'єднання або між вказівником на наступну вільну адресу пам'яті, або фактичним об'єктом типу T. Це дозволяє звільняти об'єкти в будь-якому порядку для повторного використання пізніше. Якщо розподільник заповнено, він заблокує ще один шматок пам’яті, який вдвічі перевищує початковий розмір. Розподільник зберігає список усієї неефективної пам’яті, щоб він міг звільнити всю пам’ять під час деконструкції.

Існує другий вектор, який є списком покажчиків на використані адреси пам'яті. Цю функцію було додано, щоб я міг перебирати об’єкти, які потрібно оновлювати в кожному кадрі (рисунок 2.7).

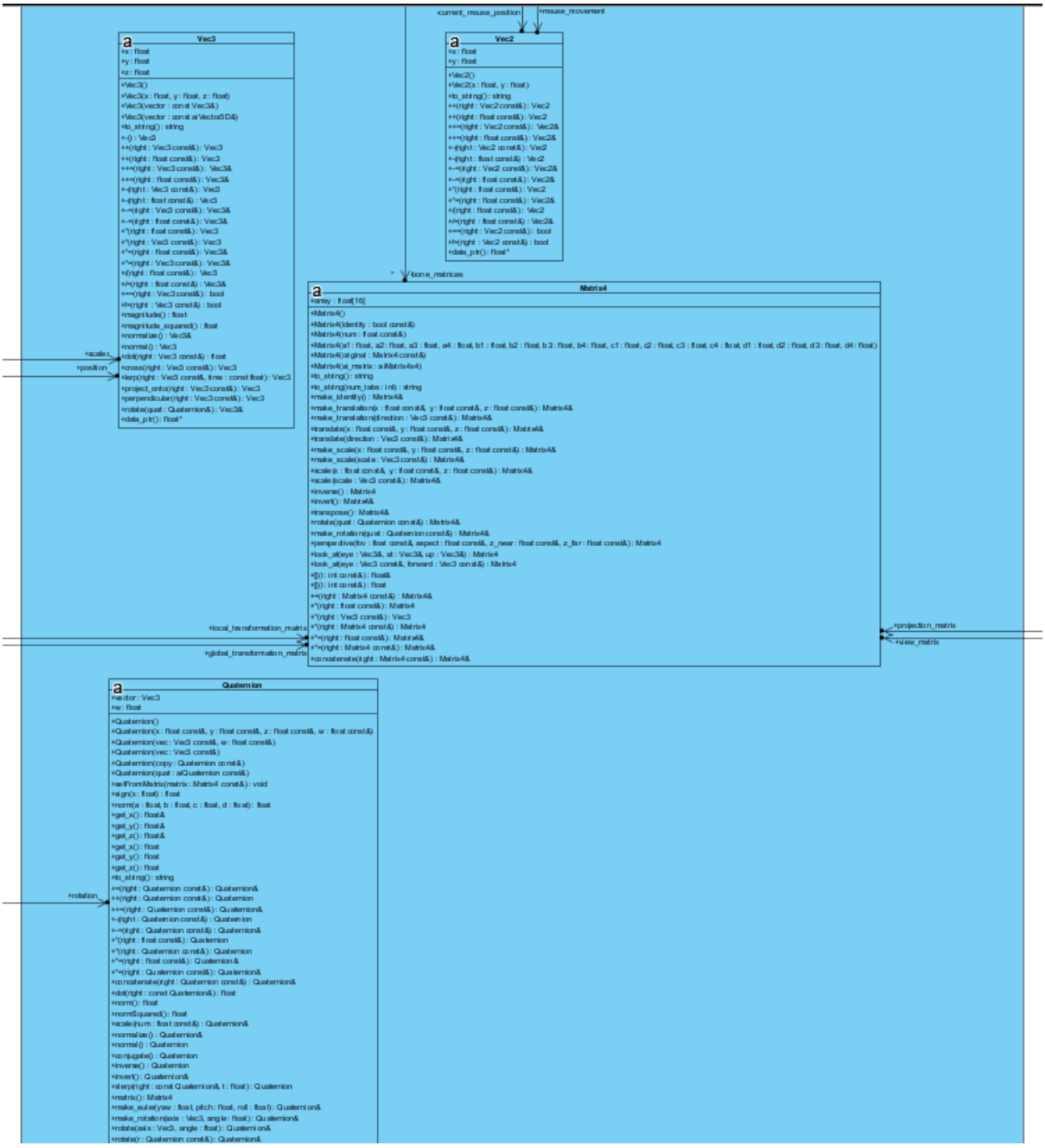


Рисунок 2.7 – Детальніша схема математичних класів

Математична бібліотека складається з класів Vec2, Vec3, Matrix4 та Quaternion у просторі імен «mtx». Вся математика, що використовується в ігровому движку, написана власноруч. Формули були взяті з різних математичних сайтів, а деякі більш складні операції, такі як інверсія матриці, були адаптовані з онлайн коду. Це досить велика кількість рядків коду, тому він досить надійний і містить багато функцій, які не використовуються у

своєму ігровому движку. Не використовувалась також SIMD математика, оскільки багато функцій спочатку містили помилки, і виправити їх набагато легше без SIMD. Тепер, коли стало можливим підтвердити правильність математичної бібліотеки, наступним кроком стане SIMD.

InputManager дозволяє додавати дії, які викликаються введенням з клавіатури або миші. Замість того, щоб код запитував, чи натиснуто певну клавішу, наприклад, «w», він запитує дію руху вперед. InputManager дозволяє легко переприв'язувати елементи керування та додавати більше елементів керування під час виконання. Дія має 3 етапи: просто натиснути, утримувати та відпустити. Усі три етапи дії можна запросити окремо.

2.3 Інструменти для налагодження

У той момент, коли неправильно використовується OpenGL, наприклад, налаштовується буфер без попереднього прив'язування, він зреагує і згенерує один або кілька прапорців помилок за кадром. Можна запитувати ці позначки помилок за допомогою функції glGetError, яка перевіряє набір прапорів помилок і повертає значення помилки, якщо OpenGL було використано неправильно. У момент виклику glGetError він повертає або позначку помилки, або її відсутність. У документації щодо функцій OpenGL ви завжди можете знайти коди помилок, які функція генерує в момент її неправильного використання. Наприклад, якщо ознайомитись з документацією функції glBindTexture, можна знайти всі коди помилок користувача, які вона може створити в однойменному розділі. У момент встановлення прапорця помилки, жодні інші позначки помилок не повідомлятимуться. Крім того, у момент виклику glGetError він очищає всі прапорці помилок, або лише один, якщо в розподіленій системі. Це означає, що якщо викликається glGetError один раз в кінці кожного кадру і він повертає помилку, не можна сказати, що це була єдина помилка, і джерело помилки могло бути де-небудь у кадрі. Чудова особливість glGetError

полягає в тому, що він дозволяє відносно легко визначити, де може бути помилка, і перевірити правильність використання OpenGL. Наприклад, з'являється чорний екран, і не відомо, що це спричинило, чи не правильно встановлений кадровий буфер, чи не зв'язана текстура. Викликаючи glGetError по всій базі коду, можливо швидко зловити перше місце, де починає з'являтися помилка. За замовчуванням функція друкує лише номери помилок, які важко зрозуміти, якщо не запам'ятовувати коди помилок. Часто має сенс написати невелику допоміжну функцію, щоб легко роздрукувати рядки помилок разом із місцем виклику функції перевірки помилок.

Директиви препроцесора __FILE__ і __LINE__, ці змінні під час компіляції замінюються відповідним файлом і рядком, у якому вони були скомпільовані. Якщо вирішуємо вставити велику кількість цих викликів в код, буде корисно точніше знати, який виклик функції повернув помилку. glGetError не дуже допомагає, оскільки інформація, яку він повертає, досить проста, але часто допомагає вловити помилки або швидко визначити, де у коді щось пішло не так. Простий, але ефективний інструмент у наборі інструментів для налагодження.

Менш поширеним, але більш корисним інструментом, ніж glCheckError, є розширення OpenGL, яке називається виведенням налагодження, яке стало частиною основного OpenGL з версії 4.3. З розширенням налагодження вихідних даних OpenGL безпосередньо надсилає користувачеві повідомлення про помилку або попередження з набагато більшою кількістю деталей порівняно з glCheckError. Він не тільки надає більше інформації, це також може допомогти вам ловити помилки саме там, де вони виникають, розумно використовуючи налагоджувач. Щоб почати використовувати вихідні дані налагодження, потрібно запитати контекст виводу налагодження від OpenGL під час нашого процесу ініціалізації. Цей процес залежить від того, яку віконну систему ви

використовуєте; тут ми обговоримо його налаштування на GLFW, але можна знайти інформацію про інші системи в додаткових ресурсах в кінці глави.

Запит контексту налагодження в GLFW напрочуд простий, оскільки все, що потрібно зробити, це передати GLFW підказку про те, що ми хотіли б мати вихідний контекст налагодження. Після того, як зініціалізується GLFW, потрібно мати контекст налагодження, якщо використовуємо OpenGL версії 4.3 або новішої, якщо ні, система може бути не в змозі запитати контекст налагодження. В іншому випадку необхідно запросити вихід налагодження, використовуючи розширення OpenGL. Принцип роботи налагодження полягає в тому, що переглядається OpenGL зворотний виклик функції реєстрації помилок, подібно до вхідних зворотних викликів GLFW, а у функції зворотного виклику ми можемо обробляти дані про помилки OpenGL так, як вважаємо за потрібне. В даному випадку будемо відображати корисні дані про помилки на консолі. З огляду на великий набір даних, які ми маємо на розкритті, ми можемо створити корисний інструмент друку помилок.

Щоразу, коли налагодження виявляє помилку OpenGL, він викликає цю функцію зворотного виклику, і ми зможемо роздрукувати велику кількість інформації про помилку OpenGL. Зауважте, що ми ігноруємо кілька кодів помилок, які зазвичай не відображають нічого корисного, наприклад, 131185 у драйверах NVidia, який повідомляє, що буфер було успішно створено. Тепер, коли є функція зворотного виклику, настав час ініціалізувати вихідні дані налагодження. Тут ми кажемо OpenGL увімкнути вихід налагодження. Виклик glEnable GL_DEBUG_SYNCRHONOUS повідомляє OpenGL безпосередньо викликати функцію зворотного виклику в момент, коли сталася помилка. За допомогою glDebugMessageControl ви можете потенційно відфільтрувати типи помилок, від яких ви хочете отримувати повідомлення. Було вирішено вирішили не фільтрувати жодне з джерел, типів чи рівня серйозності. Якби потрібно було показувати лише повідомлення з API OpenGL, які є помилками та мають високий рівень

серйозності. Ще один чудовий трюк із виведенням налагодження полягає в тому, що ви можете відносно легко визначити точний рядок або викликати помилку.

Встановивши точку зупинки в DebugOutput для певного типу помилки, або у верхній частині функції, якщо це не суттєво, налагоджувач вловить що виникла помилка, і можна перейти вгору по стеку викликів до будь-якої функції, що викликала відправку повідомлення. Це вимагає деякого ручного втручання, але якщо ви приблизно знати, що шукати, надзвичайно корисно швидко визначити, який виклик спричиняє помилку. Окрім читання повідомлень, також можна надсилати повідомлення до системи виводу налагодження за допомогою glDebugMessageInsert. Це особливо корисно, якщо підключатись до іншої програми або коду OpenGL, який використовує вихідний контекст налагодження. Інші розробники можуть швидко визначити будь-яку повідомлену помилку, яка виникає у користувацькому коді OpenGL. Підсумовуючи, вивід налагодження, якщо ви можете ним скористатися, надзвичайно корисний для швидкого виявлення помилок і вартий зусиль під час налаштування, оскільки заощаджує значний час розробки. Тут можна знайти приклад вихідного коду з налаштованим контекстом виведення glGetError і налагодження.

Коли справа доходить до GLSL, така функція як glGetError стає недоступною, і не має можливості переходити через код шейдера. Коли отримується чорний екран або зовсім неправильні зображення, часто важко зрозуміти, чи щось не так з кодом шейдера. Є звіти про помилки компіляції, які повідомляють про синтаксичні помилки, але виявлення семантичних помилок – це ще один звір. Один часто використовуваний прийом, щоб з’ясувати, що не так з шейдером, це оцінити всі відповідні змінні в програмі шейдера, відправивши їх безпосередньо на вихідний канал фрагментного шейдера. Виводячи змінні безпосередньо на вихідні кольорові канали, можна передавати цікаву інформацію, перевіряючи візуальні

результати. Наприклад, скажімо, що ми хочемо перевірити, чи має модель правильні вектори нормалей. Ми можемо передати їх, перетворені або нетрансформовані, з вершинного шейдера до фрагментного шейдера, де ми потім виведемо нормалі наступним чином. Виводячи змінну до вихідного колірного каналу таким чином, ми можна швидко перевірити, наскільки змінна, відображає правильні значення. Якщо, наприклад, візуальний результат повністю чорний, то зрозуміло, що вектори нормалі неправильно передаються шейдерам і коли вони відображаються, відносно легко перевірити, чи правильні вони чи ні.

У кожного драйвера є свої недоліки та переваги, наприклад, драйвери NVIDIA є більш гнучкими і, як правило, ігнорують деякі обмеження щодо специфікації, тоді як драйвери ATI/AMD, як правило, краще дотримуються специфікації OpenGL. Потрібно мати на увазі, що валідатор мови GLSL визначає тип шейдера за списком фіксованих розширень:

- vert: вершинний шейдер;
- frag: фрагментний шейдер;
- geom: шейдер геометрії;
- tesc: шейдер керування теселяцією;
- tese: шейдер для оцінки теселяції;
- comp: обчислювальний шейдер.

Якщо він не виявляє помилки, він не повертає результат. Тестування компілятора GLSL на зламаному вершинному шейдері дає наступний результат (рисунок 2.8).

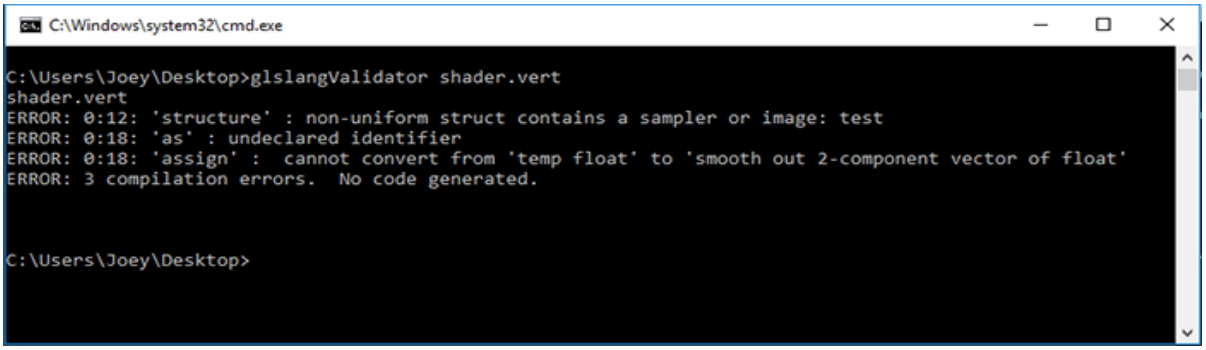


Рисунок 2.8 – Результат запуску шейдерів

Результатом цього є те, що шейдери на одній машині можуть не працювати на іншій через відмінності драйверів. Враховуючи двійковий GLSL lang validator, можна легко перевірити свій код шейдера, передавши його як перший аргумент двійкового файлу. Він не покаже тонких відмінностей між компіляторами AMD, NVidia або Intel GLSL, і не допоможе повністю захистити шейдери від помилок, але принаймні допоможе перевірити шейдери на відповідність прямій специфікації GLSL.

Ще один корисний інструмент для набору інструментів налагодження — відображення вмісту фреймового буфера в певній попередньо визначеній області екрана. Можливо, досить часто використовуються буфери кадрів, і оскільки більшість їхньої магії відбувається за лаштунками, іноді важко зрозуміти, що саме йде не так. Відображення вмісту фреймбуфера на екрані є корисним, щоб швидко перевірити, чи все запускається правильно. Використовуючи простий шейдер, який відображає лише текстуру, можна легко написати невелику допоміжну функцію для швидкого відображення будь-якої текстури у верхньому правому куті екрана.

Немає обмежень щодо того, щоб код гри проходив лише через движок для всього, для чого він розроблений. Це забезпечує гнучкість і, власне кажучи, іноді необхідно використовувати деякі функції з бібліотек безпосередньо. Наприклад, двигун не забезпечує можливості введення користувача. Бібліотека SDL2, на яку посилаються, дуже хороша в цьому, тому розробник може використовувати її безпосередньо.

Проблема візуалізації полягає в тому, що потрібно знати багато речей про OpenGL і сам GLSL, перш ніж написати код, який дійсно щось робить. І потім, як тільки зібрано деякі інструкції для ЦП і ГП, які повинні працювати, багато речей може піти не так, як-от помилки від неправильних типів даних, які використовуються для просування вершин до графічного процесора, неправильне позиціонування або використані неправильної матриці. І єдиний спосіб дізнатися, що не так у багатьох випадках, має бути написаний код, який збирає помилки з графічного процесора, тому що вони не легко виводяться на ваш екран. Цікавою особливістю, з якою можна поекспериментувати, є управління залежностями. Можна скористатись сервісом під назвою Biicode, який дозволить це зробити. Biicode може отримувати проєкти, які підтримують CMake, з незначними і якщо все зроблено добре, ненав'язливими змінами їх CMakeFile.txt. Кожен проєкт може посилатися на інші проєкти, надавши вихідний код бібліотеки, і Biicode аналізуватиме залежності та автоматично завантажуватиме та компілює їх під час збірки.

Все, що розробник повинен зробити, це додати оператор #include з адресою потрібного хедера проєкту, розміщеного в бібліотеці, а Biicode зробить усе інше. Можна сказати, що це еквівалент Nuget або Maven, але для C++. Причиною, чому було обрано цей сервіс, хоча він і є відносно новим, була швидкість розробки.

CMake сам по собі також дуже зручний, але налаштування та підключення бібліотек займає багато часу. Це дуже трудомістка процедура, особливо під час роботи між платформою або перемикання між налагодженням і випуском збірок. Оскільки Biicode виявить потрібні файли з кожної бібліотеки, завантажить і компілює їх на льоту, розробник не поглиблюється у відповідні тонкощі налаштування проєкту. Biicode є проєктом з відкритим вихідним кодом, тому навіть якби служба в її нинішньому вигляді стала недоступною в якийсь момент, можна б було

з’ясувати, як налаштувати її локально, повернувшись б до звичайного ванільного CMake, можливо, за допомогою ExternalProject_Add, що все одно було б більше обмеженою функцією, або можна б було знайти інший менеджер залежностей. Але на даний момент, це найкраще рішення для невеликого проєкту.

Однією з проблем, про яку не було згадано раніше, є фактично запуск OpenGL з однієї кодової бази на різних платформах. Існують різні бібліотеки, на які можна використати. Крім того, у Windows і Linux легко перевірити, яка версія доступна, і вибрати її. Однак на Mac для цього потрібно зробити припущення щодо версії, оскільки деякі можливості виявлення відсутні, принаймні, наскільки вдалося з’ясувати. Всі ці речі попередньо налаштовані та запропоновані через бібліотеку з менеджера залежностей, це може бути хорошою ідеєю. Може бути недоцільно це робити, але, якщо просто додати оператор #include, що вказує на якийсь розміщений код візуалізації, і бути готовим до програмування на трьох операційних системах, не роблячи нічого іншого, можна розробляти проєкт цим шляхом. Інша річ, яка є дуже хорошою в менеджері залежностей, — це розділення проблем.

Таким чином функціональність візуалізації може бути охоплена та налаштована одним розробником, інші можуть підтримувати інші корисні бібліотеки. Кожен новий проєкт може робити більше речей, заощаджуючи час, повторно використовуючи те, що є, і, якщо сам проєкт є новою бібліотекою, додаючи більше корисних функцій, які можуть використовувати розробники. Зберігаючи бібліотеки невеликими та цілеспрямованими, створюється пул постійно зростаючих можливостей повторного використання коду без зайвої мети. Наприклад, потрібно покращити small3d, але невідомо, чи необхідно додавати до нього більше функцій. Якщо потрібно створити гру-платформер, замість того, щоб додавати її багаторазові елементи до самого small3d, можна створити іншу бібліотеку під назвою small3d_platformer. Інший розробник може зробити

small3d_shooter. Це не є новим у тому сенсі, що повторне використання бібліотеки все одно працює таким чином, але наявність у мережі з менеджером залежностей для C++ є перевагою. Це робить повторне використання коду набагато швидшим, а також є гарантією того, що різні бібліотеки завжди будуть взаємодіяти, оскільки завжди зберігається запис про зв'язки між конкретними версіями. Кожен раз, коли хтось використовує одну частину «ланцюга», це перевіряє, що вона працює, або отримує повідомлення, що її потрібно виправити.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Вибір бібліотек

Для розробки двигунця та роботи з графікою необхідно встановити графічну бібліотеку. З її допомогою можна малювати різноманітні об’єкти та забезпечити відображення їх на екран. В двигунці використовуватиму OpenGL.

OpenGL в основному вважається API (інтерфейсом прикладного програмування), який надає нам великий набір функцій, які ми можемо використовувати для маніпулювання графікою та зображеннями. Однак OpenGL сам по собі не є API, а просто специфікацією, розроблено та підтримується Групою Хронос. Специфікація OpenGL точно визначає, яким має бути результат/виведення кожної функції та як вона повинна виконуватися. Тоді розробники, які впроваджують цю специфікацію, повинні знайти рішення щодо того, як ця функція повинна працювати. Оскільки специфікація OpenGL не дає нам деталей реалізації, фактично розроблені версії OpenGL можуть мати різні реалізації, якщо їх результати відповідають специфікації (і, таким чином, однакові для користувача). Люди, які розробляють бібліотеки OpenGL, зазвичай є виробниками відеокарт. Кожна відеокарта, яку ви купуєте, підтримує певні версії OpenGL, які є версіями OpenGL, розробленими спеціально для цієї карти. При використанні системи Apple бібліотека OpenGL обслуговується самими Apple, а під Linux існує комбінація версій графічних постачальників і адаптацій цих бібліотек для любителів. Це також означає, що всякий раз, коли OpenGL демонструє дивну поведінку, яку він не повинен, швидше за все, це вина виробників відеокарт або того, хто розробив чи підтримував бібліотеку.

Раніше використання OpenGL означало розробку в негайному режимі (часто званий конвеєром фіксованих функцій), який був простим у використанні методом для малювання графіки. Більшість функціональних

можливостей OpenGL було приховано всередині бібліотеки, і розробники не мали великого контролю над тим, як OpenGL виконує свої обчислення. Згодом розробники захотіли більшої гнучкості, і з часом специфікації стали більш гнучкими; розробники отримали більше контролю над своєю графікою. Негайний режим дійсно простий у використанні та розумінні, але він також надзвичайно неефективний. З цієї причини специфікація почала забороняти функціональність безпосереднього режиму з версії 3.2 і почала мотивувати розробників розвиватися в режимі основного профілю OpenGL, який є розділом специфікації OpenGL, який видалив всю стару застарілу функціональність. При використанні основного профілю OpenGL, OpenGL змушує нас використовувати сучасні методи. Щоразу, коли ми намагаємося використати одну з застарілих функцій OpenGL, OpenGL викликає помилку і припиняє малювання. Перевага навчання сучасному підходу в тому, що він дуже гнучкий та ефективний. Однак цьому також важче навчитися. Безпосередній режим досить багато абстрагував від фактичних операцій, які виконував OpenGL, і хоча його було легко навчитися, було важко зрозуміти, як насправді працює OpenGL. Сучасний підхід вимагає від розробника дійсного розуміння OpenGL і графічного програмування, і хоча це трохи складно, він забезпечує набагато більшу гнучкість, більше ефективності і, головне: набагато краще розуміння графічного програмування. Хоча це і складніше, але це вартує зусиль.

Відмінною особливістю OpenGL є його підтримка розширень. Щоразу, коли графічна компанія придумує нову техніку або нову велику оптимізацію для візуалізації, це часто зустрічається в розширенні, реалізованому в драйверах. Якщо апаратне забезпечення, на якому працює програма, підтримує таке розширення, розробник може використовувати функціональні можливості, надані розширенням, для більш просунутої та ефективної графіки. Таким чином, розробник графіки все ще може використовувати ці нові методи візуалізації, не чекаючи, поки OpenGL включить

функціональність у свої майбутні версії, просто перевіривши, чи підтримує розширення відеокартою. Часто, коли розширення популярне або дуже корисне, воно в кінцевому підсумку стає частиною майбутніх версій. Сама по собі бібліотека є великим кінцевим автоматом. Набір змінних, які визначають, як OpenGL має працювати на даний момент. Стан зазвичай називають контекстом OpenGL. Використовуючи цю бібліотеку, ми часто змінюємо його стан, встановлюючи деякі параметри, маніпулюючи деякими буферами, а потім відтворюємо, використовуючи поточний контекст. Кожного разу, коли ми говоримо OpenGL, що ми тепер хочемо малювати лінії замість трикутників, наприклад, ми змінюємо стан, змінюючи деяку змінну контексту, яка встановлює, як OpenGL має малювати. Як тільки ми змінимо контекст, сказавши OpenGL, що він повинен малювати лінії, наступні команди малювання тепер малюватимуть лінії замість трикутників. Працюючи в середовищі, ми зустрінемо кілька функцій зміни стану, які змінюють контекст, і кілька функцій використання стану, які виконують деякі операції на основі поточного стану OpenGL.

Бібліотеки OpenGL написані на C і допускають багато виведень іншими мовами, але в своїй основі вона залишається C-бібліотекою. Оскільки багато мовних конструкцій C погано перекладаються на інші мови вищого рівня, OpenGL було розроблено з урахуванням кількох абстракцій. Однією з таких абстракцій є об'єкти.

Щоб встановити OpenGL нам потрібен компілятор C/C++, або GCC (Колекція компіляторів GNU) від MinGW або Cygwin, або Visual C/C++ Compiler чи інші. Нам потрібні наступні набори бібліотек для програмування OpenGL:

− Core OpenGL (GL): складається з сотень функцій, які починаються з префікса "gl" (наприклад, glColor, glVertex, glTranslate, glRotate). Core OpenGL моделює об’єкт за допомогою набору геометричних примітивів, таких як точка, лінія та багатокутник.

− Бібліотека допоміжних програм OpenGL (GLU): побудована на основі ядра OpenGL для надання важливих утиліт та інших моделей будівель (наприклад, qradric поверхонь). Функції GLU починаються з префікса "glu" (наприклад, gluLookAt, gluPerspective).

− OpenGL Utilities Toolkit (GLUT): забезпечує підтримку взаємодії з операційною системою (наприклад, створення вікна, обробка введення клавіш і миші); і більше будівельних моделей (наприклад, сфери та тора). Функції GLUT починаються з префікса «glut» (наприклад, glutCreatewindow, glutMouseFunc). GLUT призначений для створення програм OpenGL малих і середніх розмірів. Хоча GLUT добре підходить для вивчення OpenGL і розробки простих додатків OpenGL, GLUT не є повнофункціональним набором інструментів, тому великі програми, які потребують складних інтерфейсів користувача, краще використовувати рідні набори інструментів віконної системи. GLUT простий, легкий і невеликий. Альтернатива GLUT включає SDL.

− OpenGL Extension Wrangler Library (GLEW): GLEW — це кросплатформна бібліотека завантаження розширень C/C++ із відкритим кодом. GLEW забезпечує ефективні механізми виконання для визначення, які розширення OpenGL підтримуються на цільовій платформі.

Кожен пакет програмного забезпечення складається з:

− Файл заголовка: "gl.h" для основного OpenGL, "glu.h" для GLU і "glut.h" (або "freeglut.h") для GLUT, зазвичай зберігається в каталозі "include\GL".

− Статична бібліотека: наприклад, у Win32 «libopengl32.a» для ядра OpenGL, «libglu32.a» для GLU, «libglut32.a» (або «libfreeglut.a» або «glut32.lib») для GLUT, зазвичай зберігається в каталозі "lib".

− Додаткова спільна бібліотека: наприклад, "glut32.dll" (для "freeglut.dll") для GLUT під Win32, зазвичай зберігається в "bin" або "c:\windows\system32".

Щоб правильно налаштувати середовище програмування OpenGL, важливо знайти шлях до каталогу та фактичні назви цих заголовних файлів і бібліотек у вашій операційній платформі.

Бібліотека glad дуже схожа на glLoadGen, вона генерує завантажувач для ваших потреб на основі офіційних специфікацій від Khronos SVN. Він був написаний таким чином, що можна легко розширити його на інші мови. Можна використовувати веб-сайт glad, щоб створити завантажувач для різних потреб, завантажити його та використовувати у своєму проєкті. Інший спосіб використання glad - це клонування чи завантаження репозиторію та створення власного завантажувача. Сам інструмент досить простий у використанні. Також можна включити джерело безпосередньо у проєкт CMake.

C++, будучи надмножиною C, підтримує велику кількість корисних математичних функцій. Ці функції доступні в стандартних C++ і C для підтримки різних математичних обчислень. Замість того, щоб зосередитися на реалізації, ці функції можна безпосередньо використовувати для спрощення коду та програм. C++ надає великий набір математичних функцій, для їх використання потрібно включити заголовний файл - <math.h> або <cmath>.

3.2 Реалізація функцій рендерингу

У основній функції ми спочатку ініціалізуємо GLFW за допомогою glfwInit, після чого ми можемо налаштувати GLFW за допомогою glfwWindowHint. Перший аргумент glfwWindowHint говорить нам, яку опцію ми хочемо налаштувати, де ми можемо вибрати параметр із великого переліку можливих параметрів із префіксом GLFW_. Другим аргументом є ціле число, яке встановлює значення нашого параметра. Список усіх можливих опцій та відповідних їм значень можна знайти в документації з роботи з вікнами GLFW. Якщо ви спробуєте запустити програму зараз, і вона

дає багато невизначених довідкових помилок, це означає, що ви не успішно зв'язали бібліотеку GLFW. Таким чином GLFW може зробити належні домовленості під час створення контексту OpenGL. Це гарантує, що, якщо користувач не має належної версії OpenGL, GLFW не запуститься. Було встановлено встановили як основною, так і другорядною версією третю. Також варто зазначити GLFW, що хочемо явно використовувати профіль ядра. Повідомлення GLFW про те, що ми хочемо використовувати профіль ядра, означає, що ми отримаємо доступ до меншої підмножини функцій OpenGL без зворотно-сумісних функцій, які нам більше не потрібні. Потрібно звернути увагу, що в Mac OS X потрібно додати glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); до коду ініціалізації, щоб він працював (рисунок 3.1).

```
glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
//glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
```

Рисунок 3.1 – Ініціалізація GLFW

Далі нам потрібно створити об'єкт вікна. Цей об'єкт вікна містить усі віконні дані і потрібен для більшості інших функцій GLFW (рисунок 3.2).

```
GLFWwindow* window = glfwCreateWindow(800, 600, "LearnOpenGL", NULL, NULL);
if (window == NULL)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window);
```

Рисунок 3.2 – Створення нового вікна

GLAD керує вказівниками функцій для OpenGL, тому ми потрібно ініціалізувати GLAD, перш ніж викликати будь-яку функцію OpenGL (рисунок 3.3).

```
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
}
```

Рисунок 3.3 – Ініціалізація Glad

Передаємо функцію GLAD, щоб завантажити адресу покажчиків функції OpenGL, яка є специфічною для ОС. GLFW дає нам glfwGetProcAddress, яка визначає правильну функцію на основі якої ОС ми компілюємо.

Перш ніж буде можлива візуалізація, нам потрібно зробити останню річ. Потвібно вказати OpenGL розмір вікна візуалізації, щоб OpenGL знав, як слід відобразити дані та координати вікна. Можна встановити ці розміри за допомогою функції glViewport. Перші два параметри glViewport задають розташування нижнього лівого кута вікна. Третій і четвертий параметри встановлюють ширину і висоту вікна візуалізації в пікселях, які встановлюються рівними розміру вікна GLFW. Можливо б було насправді встановити розміри вікна перегляду на значення, менші за розміри GLFW. Тоді весь рендеринг OpenGL буде відображатися в меншому вікні, і могли б, наприклад, відобразити інші елементи за межами вікна перегляду OpenGL. Однак у момент, коли користувач змінює розмір вікна, область перегляду також має бути налаштована. Ми можемо зареєструвати функцію зворотного виклику у вікні, яка викликається щоразу, коли вікно змінюється. Функція розміру буфера кадру приймає GLFWwindow як свій перший аргумент і два цілі числа, що вказують нові розміри вікна. Щоразу, коли розмір вікна

змінюється, GLFW викликає цю функцію та заповнює потрібні аргументи для обробки (рисунок 3.4).

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}
```

Рисунок 3.4 – Функція динамічного виведення вікна

Коли вікно вперше відображається, framebuffer_size_callback також викликається з отриманими розмірами вікна. Для дисплеїв Retina ширина та висота будуть значно вищими за вихідні введені значення. Існує багато функцій зворотного виклику, які ми можемо встановити для реєстрації наших власних функцій. Наприклад, ми можемо створити функцію зворотного виклику, щоб обробляти зміни введення джойстика, обробляти повідомлення про помилки тощо. Ми реєструємо функції зворотного виклику після того, як ми створили вікно і до початку циклу візуалізації.

Не потрібно, щоб програма малювала одне зображення, а потім негайно завершувалась та закривала вікно. Потрібно, щоб програма продовжувала малювати зображення та обробляла введення користувача, поки програмі не буде явно вказано зупинитися. З цієї причини потрібно створити цикл while, який називається циклом візуалізації, який продовжує працювати, поки явно не надійде вказівка GLFW зупинитися. Наступний код показує дуже простий цикл візуалізації (рисунок 3.5).

```
while (!glfwWindowShouldClose(window))
{
    WndResize(width, height);

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Рисунок 3.5 – Цикл візуалізації

Функція glfwWindowShouldClose перевіряє на початку кожної ітерації циклу, чи GLFW отримав інструкцію закрити. Якщо так, функція повертає true, і цикл візуалізації перестає виконуватися, після чого ми можемо закрити програму. Функція glfwPollEvents перевіряє, чи спрацьовують якісь події, наприклад, введення з клавіатури або події руху миші, оновлює стан вікна та викликає відповідні функції які ми можемо зареєструвати за допомогою методів зворотного виклику. GlfwSwapBuffers поміняє місцями буфер кольорів, великий двовимірний буфер, який містить значення кольорів для кожного пікселя у вікні GLFW, який використовується для візуалізації під час цієї ітерації візуалізації, і відображатиме його як вихід на екран. Як тільки відбувається вихід з циклу візуалізації, потрібно правильно очистити і звільнити всі ресурси GLFW, які були виділені. Можна зробити це за допомогою функції glfwTerminate, яка викликається в кінці основної функції. Це очистить всі ресурси та належним чином та закриє програму.

Потрібно також реалізувати певну форму контролю введення даних в GLFW, цього можливо досягнути за допомогою кількох функцій введення GLFW. Використовуватимемо функцію glfwGetKey GLFW, яка приймає вікно як вхід разом із ключем. Функція повертає, чи натиснута ця клавіша в даний момент. Створимо функцію Player_Move, щоб весь вхідний код був організованим (рисунок 3.6).

```
void Player_Move()
{
    Camera_MoveDirection( GetKeyState('W') < 0 ? 1 : GetKeyState('S') < 0 ? -1 : 0
                         ,GetKeyState('D') < 0 ? 1 : GetKeyState('A') < 0 ? -1 : 0
                         , 0.1);
    Camera_AutoMoveByMouse(400, 400, 0.2f);
}
```

Рисунок 3.6 – Функція введення

Також додана функція перевірки, чи натиснув користувач клавішу escape, якщо вона не натиснута, glfwGetKey повертає GLFW_RELEASE. Якщо користувач натиснув клавішу escape, закривається GLFW, встановлюючи для його властивості WindowShouldClose значення true за допомогою glfwSetwindowShouldClose. Наступна перевірка умов основного циклу while завершиться невдачею, і програма закриється. Це дає простий спосіб перевірити наявність певних натискань клавіш і відповідно реагувати на кожен кадр. Ітерацію циклу візуалізації частіше називають фреймом.

Потрібно помістити всі команди візуалізації в цикл візуалізації, оскільки було б непогано виконувати всі команди візуалізації на кожній ітерації або кадрі циклу. Це виглядатиме приблизно так (рисунок 3.7).

```
while (!glfwWindowShouldClose(window))
{
    WndResize(width, height);

    glEnable(GL_DEPTH_TEST);

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    Enabled_Light();

    processInput(window);
    Camera_Apply();
    Player_Move();

    paintGL();

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Рисунок 3.7 – Виконання команд в кожному фреймі

Тож зараз було підготовано все, щоб заповнити цикл візуалізації великою кількістю викликів рендеринга.

3.3 Реалізація камери

Камера для перегляду - це всі координати вершин, які розглядаються з точки зору камери, як джерело сцени: матриця перегляду перетворює всі координати світу в координати огляду, які є відносно положення камери. Щоб визначити камеру, нам потрібно її положення у просторі, напрямок, у який вона дивиться, вектор, спрямований праворуч, і вектор, спрямований вгору від камери. Потрібно створити систему координат з трьома перпендикулярними одиничними осями з положенням камери в якості початку координат. Отримати положення камери легко. Положення камери — це вектор у просторі, який вказує на положення камери (рисунок 3.8).

```
void Camera_Apply()
{
    glRotatef(-camera.Xrot, 1,0,0);
    glRotatef(-camera.Zrot, 0,0,1);
    glTranslatef(-camera.x, -camera.y, -camera.z);
}
```

Рисунок 3.8 – Функція визначення камери

Наступний необхідний вектор – це напрямок камери. Наразі ми дозволимо камері вказати на походження нашої сцени. Якщо відняти два вектори один від одного, то отримаємо вектор, який є різницею цих двох векторів. Таким чином, віднімання вектора положення камери від вектора початку сцени призводить до того, що нам потрібно. Для системи координат матриці перегляду необхідно, щоб її z-вісь була додатною, і оскільки за умовою в OpenGL камера спрямована на від'ємну вісь z, потрібно зробити

від’ємним вектор напрямку. Якщо змінити порядок віднімання, отримаємо вектор, спрямований на додатню вісь z камери. Далі потрібен правий вектор, який представляє додатню вісь x положеня камери. Оскільки результатом перехресного добутку є вектор, перпендикулярний до обох векторів, ми отримаємо вектор, який вказує на додатний напрямок осі x, якщо ми змінимо порядок скалярного добутку, ми отримаємо вектор, який вказує на від’ємну вісь x. Тепер, коли є і вектор осі x, і вектор осі z, отримати вектор, який вказує на додатню вісь y камери. Це відносно легко, потрібно отримати скалярний добуток правого вектора та вектора напрямку. За допомогою перехресного продукту та кількох прийомів вдалося створити всі вектори, які утворюють положення камери. Цей процес також відомий як процес Грама-Шмідта в лінійній алгебрі. Використовуючи ці вектори камер, тепер стало можливим створити матрицю LookAt, яка виявляється дуже корисною для створення камери (рисунок 3.9).

```
void Camera::updateRotation() {

    mtx::Quaternion yaw_quaternion = mtx::Quaternion();
    yaw_quaternion.make_rotation(mtx::Vec3(0.0f, 1.0f, 0.0f), yaw);
    mtx::Quaternion pitch_quaternion = mtx::Quaternion();
    pitch_quaternion.make_rotation(mtx::Vec3(1.0f, 0.0f, 0.0f), pitch);
    mtx::Quaternion roll_quaternion = mtx::Quaternion();
    roll_quaternion.make_rotation(mtx::Vec3(0.0f, 0.0f, 1.0f), roll);
    getTransform()->rotation *= yaw_quaternion * pitch_quaternion * roll_quaternion;
    getTransform()->rotation.normalize();
}
```

Рисунок 3.9 – Функція ротації камери

Чудова особливість матриць полягає в тому, що якщо визначити координатний простір за допомогою трьох перпендикулярних або нелінійних осей, можна створити матрицю з цими трьома осями плюс вектор трансляції, і можна перетворити будь-який вектор до цього координатного простору, помноживши його на ця матрицю. Це саме те, що робить матриця LookAt,

коли є три перпендикулярні осі та вектор положення для визначення простору камери, стає можливим створити власну матрицю LookAt:

LookAt=[RxRyRz0UxUyUz0DxDyDz00001]∗[100−Px010−Py001−Pz0001]

де R — правий вектор, U — вектор угору, D — вектор напрямку, а P — вектор положення камери (рисунок 3.10).

```
float time_elapsed = Game::get().time_manager.time_elapsed;

if (Game::get().input_manager.keys.at("forward").is_down) {
    getTransform()->translateLocal(0.0f, 0.0f, -sensitivity_move * time_elapsed);
}
if (Game::get().input_manager.keys.at("backward").is_down) {
    getTransform()->translateLocal(0.0f, 0.0f, sensitivity_move * time_elapsed);
}
if (Game::get().input_manager.keys.at("right").is_down) {
    getTransform()->translateLocal(sensitivity_move * time_elapsed, 0.0f, 0.0f);
}
if (Game::get().input_manager.keys.at("left").is_down) {
    getTransform()->translateLocal(-sensitivity_move * time_elapsed, 0.0f, 0.0f);
}
if (Game::get().input_manager.keys.at("up").is_down) {
    getTransform()->translateLocal(0.0f, sensitivity_move * time_elapsed, 0.0f);
}
if (Game::get().input_manager.keys.at("down").is_down) {
    getTransform()->translateLocal(0.0f, -sensitivity_move * time_elapsed, 0.0f);
}
if (Game::get().input_manager.keys.at("roll_left").is_down) {
    roll += Game::get().time_manager.time_elapsed * sensitivity_roll;
}
if (Game::get().input_manager.keys.at("roll_right").is_down) {
    roll -= Game::get().time_manager.time_elapsed * sensitivity_roll;
}
yaw += sensitivity_yaw * Game::get().input_manager.mouse_movement.x;
pitch += sensitivity_pitch * Game::get().input_manager.mouse_movement.y;

updateRotation();
```

Рисунок 3.10 – Функція переміщення камери

Частини обертання (ліва матриця) і трансляції (права матриця) інвертовані, оскільки ми хочемо обертати та перекладати світ у напрямку, протилежному тому, куди ми хочемо, щоб рухалася камера. Використання цієї матриці LookAt як нашої матриці перегляду ефективно перетворює всі координати світу в простір перегляду, який ми щойно було зазначено. Потім

матриця LookAt створює матрицю перегляду, яка дивиться на задану ціль. GLM вже виконує всю цю роботу за нас. Нам потрібно лише вказати положення камери, цільове положення та вектор, який представляє вектор вгору у світовому просторі (вектор вгору, який ми використовували для розрахунку правого вектора). Потім GLM створює матрицю LookAt, яку ми можемо використовувати як нашу матрицю перегляду.

3.4 Реалізація мешів

За допомогою Assimp ми можемо завантажувати багато різних моделей у програму, але після завантаження всі вони зберігаються в структурах даних Assimp. Потрібно перетворити ці дані у формат, який розуміє OpenGL, щоб можна було відображати об’єкти. Сітка являє собою єдиний об’єкт, який можна витягнути. Визначимо те, що мінімально повинен мати меш як структура даних. Меш повинен мати набір вершин, де кожна вершина містить вектор положення, вектор нормалі та вектор координат текстури. Меш також повинен містити індекси для індексованого малювання та дані про матеріал у вигляді текстур (дифузних або дзеркальних карт). Коли було встановлено мінімальні вимоги до класу мешу, можна визначити вершину в OpenGL. Ми зберігаємо кожен з необхідних атрибутів вершин у структурі під назвою Vertex. Поруч із структурою Vertex ми також необхідно організувати дані текстури в структурі Texture (рисунок 3.11).

```
struct Texture
{
    unsigned int id;
    std::string type;
};

struct Vertex
{
    glm::vec3 Position;
    glm::vec3 Normal;
    glm::vec2 TexCoords;
};
```

Рисунок 3.11 – Структури даних для мешів

Ми зберігаємо ідентифікатор текстури та її тип, наприклад, дифузна або дзеркальна текстура. Знаючи фактичне представлення вершини та текстури, ми можемо почати визначати структуру класу мешу (рисунок 3.12).

```
class Mesh: public Component {
public:
    Mesh();
    Mesh(std::vector<Vertex> const &vertices, std::vector<unsigned int> const &indices, Material* material);

    ~Mesh();

    std::vector<Vertex> vertices;
    std::vector<unsigned int> indices;
    Material* material;
};
```

Рисунок 3.12 – Структура мешу

Клас не надто складний. У конструкторі ми надаємо сітці всі необхідні дані, ініціалізуємо буфери і малюємо меш. Зверніть увагу, що ми надаємо. Передаючи шейдер до мешу, можна встановити кілька уніформ перед малюванням, наприклад, зв'язувати семплери з блоками текстури. Зміст функції конструктора досить простий. Встановлюються відкриті змінні класу за допомогою відповідних змінних аргументів конструктора. Завдяки конструктору ми тепер маємо великі списки даних сітки, які ми можемо використовувати для візуалізації. Тепер потрібно налаштувати відповідні буфери та вказати макет вершинного шейдера за допомогою покажчиків атрибутів вершин. Наразі не повинно виникнути проблем із цими поняттями, але цього разу ми трохи доповнили це введенням даних вершин у структури. Структури мають чудову властивість у C++, що їхнє розташування пам'яті є послідовним. Тобто, якби ми представили структуру як масив даних, вона міститиме лише змінні структури в послідовному порядку, який безпосередньо перетворюється на масив із плаваючою чисельністю, який необхідний для буфера масиву.

Остання функція, яку нам потрібно визначити, щоб клас Mesh був повним, це його функція рендеру. Перш ніж відтворювати меш, спочатку

потрібно зв’язати відповідні текстури, перед викликом glDrawElements (рисунок 3.13).

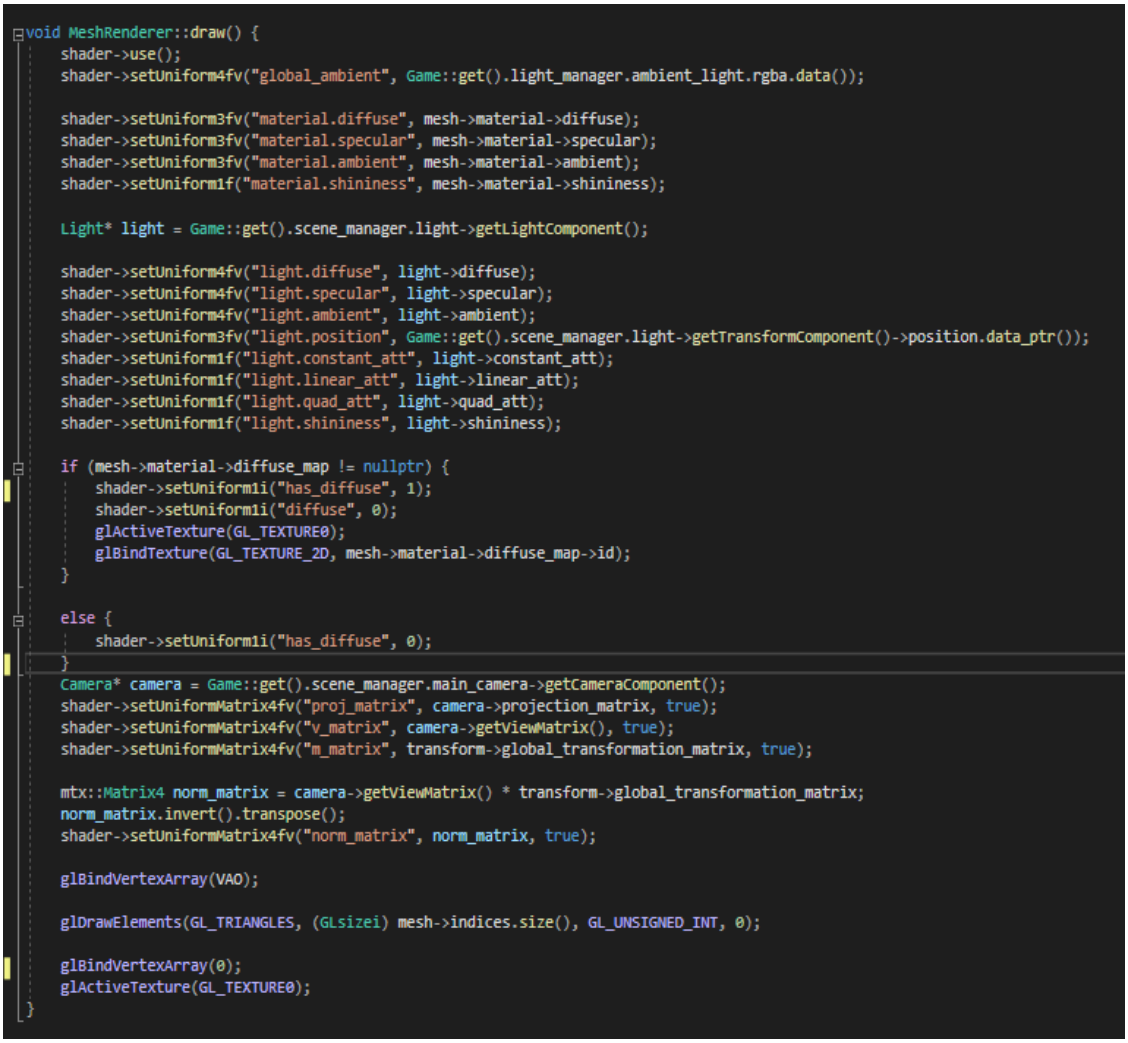
```
MeshRenderer::MeshRenderer() :
    VAO(0), VBO(0), EBO(0),
    mesh(nullptr)
{
    if (shader == nullptr) {
        Shader new_shader = Shader(std::string("Shaders/mesh_shader.vert"), std::string("Shaders/mesh_shader.frag"));
        shader = Game::get().data_manager.addData(new_shader);
    }
}

MeshRenderer::~MeshRenderer() {
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    glDeleteBuffers(1, &EBO);
}
```

Рисунок 3.13 – Конструктор та деструктор класу рендеру мешів

Однак це складно, оскільки з самого початку не відомо, скільки текстур у сітки і якого типу вони можуть мати. Щоб вирішити проблему припустимо певний шаблон щодо імен: кожна дифузна текстура має назву texture_diffuseN, а кожна дзеркальна текстура повинна мати назву texture_specularN, де N — будь-яке число від 1 до максимальної кількості дозволених семплерів текстур. Відповідно до цього стандарту можна визначити скільки завгодно семплерів текстур у шейдерах (до максимуму OpenGL), і якщо сітка дійсно містить стільки текстур, стає відомо, як вони будуть називатися. Відповідно до цієї конвенції ми можемо обробляти будь-яку кількість текстур на одному меші, і розробник шейдерів може вільно використовувати їх скільки завгодно, визначивши відповідні зразки (рисунок 3.14).

Потім ми знаходимо відповідний семплер, надаємо йому значення розташування, яке відповідає поточному активному блоку текстури, і прив’язуємо текстуру. Це також причина, чому нам потрібен шейдер у функції рендеру.



```
void MeshRenderer::draw() {
    shader->use();
    shader->setUniform4fv("global_ambient", Game::get().light_manager.ambient_light.rgba.data());

    shader->setUniform3fv("material.diffuse", mesh->material->diffuse);
    shader->setUniform3fv("material.specular", mesh->material->specular);
    shader->setUniform3fv("material.ambient", mesh->material->ambient);
    shader->setUniform1f("material.shininess", mesh->material->shininess);

    Light* light = Game::get().scene_manager.light->getLightComponent();

    shader->setUniform4fv("light.diffuse", light->diffuse);
    shader->setUniform4fv("light.specular", light->specular);
    shader->setUniform4fv("light.ambient", light->ambient);
    shader->setUniform3fv("light.position", Game::get().scene_manager.light->getTransformComponent()->position.data_ptr());
    shader->setUniform1f("light.constant_att", light->constant_att);
    shader->setUniform1f("light.linear_att", light->linear_att);
    shader->setUniform1f("light.quad_att", light->quad_att);
    shader->setUniform1f("light.shininess", light->shininess);

    if (mesh->material->diffuse_map != nullptr) {
        shader->setUniform1i("has_diffuse", 1);
        shader->setUniform1i("diffuse", 0);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, mesh->material->diffuse_map->id);
    }

    else {
        shader->setUniform1i("has_diffuse", 0);
    }
    Camera* camera = Game::get().scene_manager.main_camera->getCameraComponent();
    shader->setUniformMatrix4fv("proj_matrix", camera->projection_matrix, true);
    shader->setUniformMatrix4fv("v_matrix", camera->getViewMatrix(), true);
    shader->setUniformMatrix4fv("m_matrix", transform->global_transformation_matrix, true);

    mtx::Matrix4 norm_matrix = camera->getViewMatrix() * transform->global_transformation_matrix;
    norm_matrix.invert().transpose();
    shader->setUniformMatrix4fv("norm_matrix", norm_matrix, true);

    glBindVertexArray(VAO);

    glDrawElements(GL_TRIANGLES, (GLsizei) mesh->indices.size(), GL_UNSIGNED_INT, 0);

    glBindVertexArray(0);
    glActiveTexture(GL_TEXTURE0);
}
```

Рисунок 3.14 – Функція рендеру мешів

Спочатку ми обчислюємо N-компонент для кожного типу текстури та об’єднуємо його з рядком типу текстури, щоб отримати відповідне ім’я уніформи. Ми також додали «матеріал». до отриманого уніфікованого імені, оскільки ми зазвичай зберігаємо текстури в структурі матеріалу (це може відрізнятися в залежності від реалізації).

3.5 Реалізація моделей

В цій задачі знадобиться Assim, він дозволить відвантажувати моделі. Метою цього підрозділу є створення іншого класу, який представляє модель у повному обсязі, тобто модель, яка містить декілька сіток, можливо, з кількома текстурами. Будинок, який містить дерев'яний балкон, вежу, і,

можливо, басейн, зробити так щоб можна було завантажити це як одну модель. Завантажимо модель через Assimp і переведемо її в кілька об’єктів Mesh, які ми створили в попередньому розділі. Структура класу Model (рисунок 3.15).

```
class ModelImporter {
public:
    GameObject* loadModel(const std::string& path);

private:
    static int MAX_BONES;
    std::string directory;
    std::vector<GameObject*> node_objects;

    GameObject* processNode(GameObject* parent, const aiNode* node, const aiScene* scene);
    void processNodeMeshes(aiNode const* node, const aiScene* scene);
    GameObject* processMesh(GameObject* parent, const aiMesh* assimp_mesh, const aiScene* scene);
    void processVertices(Mesh* mesh, const aiMesh* assimp_mesh);
    void processIndices(Mesh* mesh, const aiMesh* assimp_mesh);
    Material* processMaterial(const aiMesh* assimp_mesh, const aiScene* scene);
    void processBones(Bones* bones, const aiMesh* assimp_mesh);
    void processAnimations(GameObject* root_node, const aiScene* scene);
    void loadAssimpMaterial(Material* material, aiMaterial *assimp_material);
    Texture* loadAssimpTextures(const aiMaterial* assimp_material, const aiTextureType& texture_type);
    Texture* loadTexture(const std::string& path);
    unsigned int TextureFromFile(const std::string& path);
    std::string filterPath(const aiString& path);

    void showNodeName(aiNode* node);
    Material* getMaterial(const std::string& material_path);
    Texture* getTexture(const std::string& texture_path);
    GameObject* getNodeObject(const std::string& game_object_directory_and_name);
};
```

Рисунок 3.15 – Клас ModelImporter

Клас Model містить вектор об’єктів Mesh і вимагає, щоб ми вказали йому розташування файлу в його конструкторі. Потім він відразу завантажує файл за допомогою функції loadModel, яка викликається в конструкторі. Усі приватні функції призначені для обробки частини процедури імпорту Assimp, і ми розглянемо їх незабаром. Ми також зберігаємо каталог шляху до файлу, який нам знадобиться пізніше під час завантаження текстур. Функція Draw не є чимось особливим і в основному обходить кожну з мешів, щоб викликати відповідну функцію Draw.

Щоб імпортувати модель і перевести її в нашу власну структуру, нам спочатку потрібно включити відповідні заголовки Assimp. Перша функція, яка викликається безпосередньо з конструктора, це loadModel. У loadModel

використовується Assimp для завантаження моделі в структуру даних, яка називається об’єктом сцени. Отримавши об’єкт сцени, ми зможемо отримати доступ до всіх необхідних даних із завантаженої моделі. Чудова особливість Assimp полягає в тому, що він акуратно абстрагується від усіх технічних деталей завантаження всіх різних форматів файлів і робить все це за допомогою одного рядка. Спочатку оголошується об’єкт Importer з простору імен Assimp, а потім викликається функція ReadFile. У якості другого аргументу функція очікує шлях до файлу та кілька параметрів постобробки. Assimp дозволяє нам вказати кілька параметрів, які змушують його виконувати додаткові обчислення або операції з імпортованими даними. Встановлюючи aiProcess_Triangulate, ми повідомляємо, що якщо модель повністю не складається з трикутників, вона повинна спочатку перетворити всі примітивні форми моделі в трикутники. aiProcess_FlipUVs повертає координати текстури на вісь y, де це необхідно під час обробки (рисунок 3.16).

```
Texture* ModelImporter::loadAssimpTextures(const aiMaterial* assimp_material, const aiTextureType& texture_type) {

    std::string type_string;
    if (Game::get().is_debug_mode) {
        switch (texture_type) {
        case aiTextureType_DIFFUSE: type_string = std::string("DIFFUSE"); break;
        case aiTextureType_SPECULAR: type_string = std::string("SPECULAR"); break;
        case aiTextureType_NORMALS: type_string = std::string("NORMAL"); break;
        case aiTextureType_HEIGHT: type_string = std::string("HEIGHT"); break;
        }
    }

    aiString aiPath;
    assimp_material->GetTexture(texture_type, 0, &aiPath);
    std::string texture_filename = filterPath(aiPath);


    if (texture_filename.length() > 0) {

        Texture* texture = getTexture(directory + texture_filename);

        if (texture != nullptr) {

            if (Game::get().is_debug_mode)  Logger::log("        " + type_string + " already loaded at path  " + directory + texture_filename);

            return texture;
        } else {
            if (Game::get().is_debug_mode) Logger::log("        " + type_string + " loading at path " + directory + texture_filename);

            return loadTexture(texture_filename);
        }
    } else {

        if (Game::get().is_debug_mode)  Logger::log("        " + type_string + " not specified ");

        return nullptr;
    }
}
```

Рисунок 3.16 – Функція загрузки Assimp Текстури

Після завантаження моделі перевіряємо, чи сцена та кореневий вузол сцени не є нульовими, і перевіряємо один з її прапорців, щоб перевірити, чи повернуті дані неповні. Якщо будь-яка з цих умов помилки виконується, ми повідомляємо про помилку, отриману з функції GetErrorString імпортера, і повертаємо. Ми також отримуємо шлях до каталогу заданого шляху до файлу. Якщо нічого не пішло не так, ми хочемо обробити всі вузли сцени. Перший вузол (кореневий вузол) передаємо рекурсивній функції processNode. Оскільки кожен вузол (можливо) містить набір дочірніх, ми хочемо спочатку обробити відповідний вузол, а потім продовжити обробку всіх дочірніх вузлів і так далі. Це відповідає рекурсивній структурі, тому ми визначимо рекурсивну функцію. Рекурсивна функція — це функція, яка виконує певну обробку і рекурсивно викликає ту саму функцію з різними параметрами, доки не буде виконана певна умова. У нашому випадку умова виходу виконується, коли всі вузли оброблені. Як ви пам’ятаєте зі структури, кожен вузол містить набір індексів меша, де кожен індекс вказує на певну сітку, розташовану в об'єкті сцени. Таким чином, ми хочемо отримати ці індекси меша, отримати кожен меш, обробити кожен меш, а потім зробити все це знову для кожного з дочірніх вузлів. Більшість сцен повторно використовують кілька їхніх текстур на кількох мешах. Якщо подумати про будинок, стіни якого мають гранітну текстуру. Цю текстуру можна також застосувати до підлоги, її стелі, сходів, можливо, столу, і, можливо, навіть маленький колодязь поблизу. Завантаження текстур — недешева операція, і в нашій поточній реалізації нова текстура завантажується та генерується для кожної сітки, навіть незважаючи на те, що та сама текстура могла бути завантажена кілька разів раніше. Це швидко стає вузьким місцем реалізації вашої моделі завантаження. Весь код програмного засобу приведений в додатку Б.

3.6 Тестування розробленого програмного забезпечення

Після завершення розробки основних компонентів двигунця його необхідно протестувати. В данному випадку буде застосовуватись «димне тестування». Воно зазвичай перевіряє всю систему, або принаймні деяку її більшу частину. Це те, що також можна назвати тестуванням повної збірки, оскільки тести будуть виконуватися на основі всієї гри. Це найдешевша форма тестування, але з неї може бути важко почати. Потрібно мати можливість якось створювати сценарій або відтворювати вхідні дані у грі. Також необхідно мати можливість визначити, що робить гра під час тестування, це легко зробити, якщо є журнал і метрики, виведені з гри. Цей результат може бути перевірений структурою тестування, щоб переконатися, що гра виконує все що повинна робити. Цей вид тестування може бути обширним або досить поверхневим, будь-який рівень такого тестування матиме величезний вплив на те, щоб гра з часом працювала достатньо добре.

Для початку тестування програми необхідно загрузити об'єкти які потрібно відобразити та помістити їх в папку Models. Після чого достатньо додати їх у файл сцени, це відбувається у форматі JSON (рисунок 3.17).

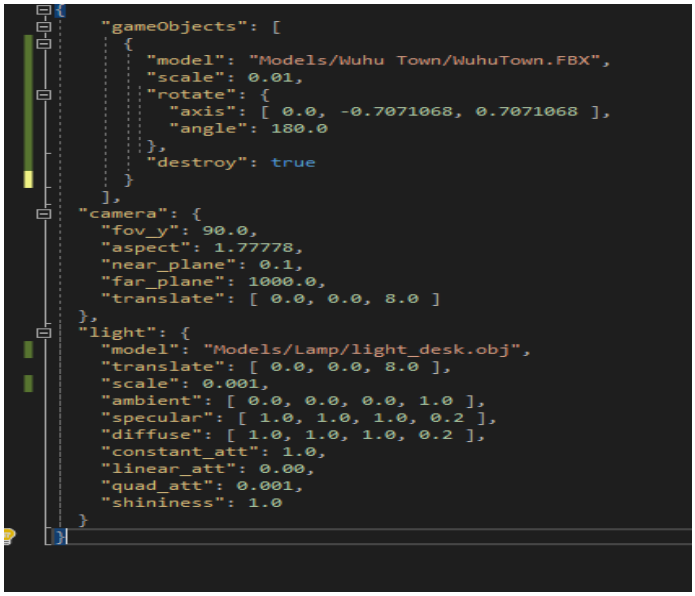


Рисунок 3.17 – Додавання об'єктів на сцену

Додавши об'єкти та запустиши гру, після нетривалої загрузки, можна побачити результат роботи програми (рисунок 3.18).

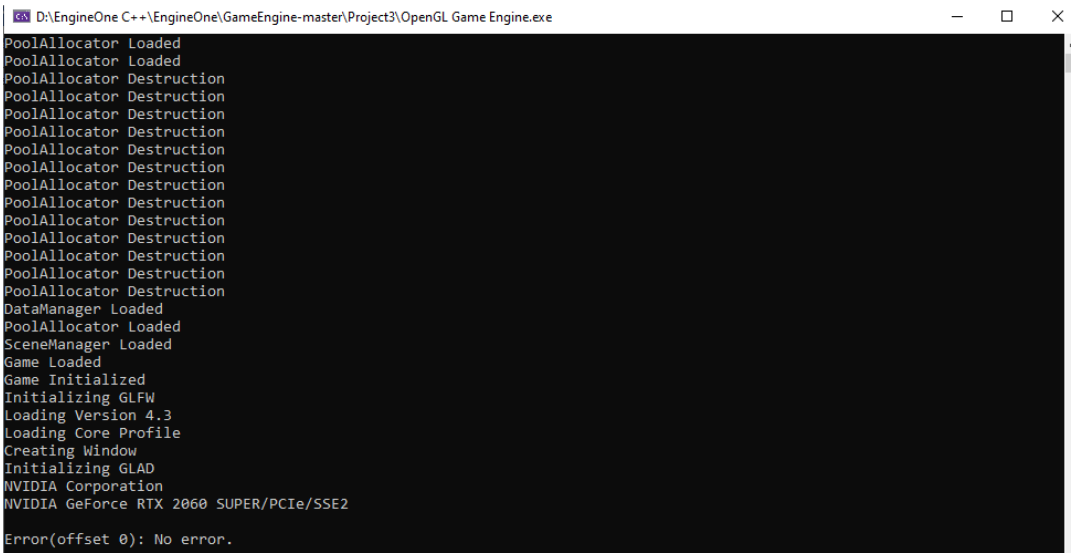


Рисунок 3.18 – Консольний вивід інформації про загрузку

Після запуску запуститься консоль в якій відображатиметься вся інформація про завантаження елементів і в випадку невдачі виведеться відповідне повідомлення про помилку (рисунок 3.19).

Рисунок 3.19 – Завантажена сцена гри

У файлі загрузки було додано також камеру, яку можна вільно переміщати по карті за допомогою клавіатури а також джерело світла, щоб всі об’єкти було видно (рисунок 3.20).

Рисунок 3.20 – Можливості повороту камери

За допомогою клавіш Q та E можна повертати камеру не тільки в певних напрямках а ще й під кутом (рисунок 3.21).

Рисунок 3.21 – Можливості переміщення камери

Після того як всі функції були випробувані і не зіткнувшись з помилками, можна сказати що програма працює правильно та виконує всі свої основні функції без проблем. Завантаження відбувається достатньо швидко, текстури та об’єкти відображаються коректно, управління теж працює так як повинно.

4 ТЕХНІКО-ЕКОНОМІЧНЕ ОБГРУНТУВАННЯ

4.1 Аналіз ринку

Основним фактором, що впливає на ринок ігрових рушіїв, є зростаючий попит з боку таких галузей, як автомобільна промисловість, BFSI та роздрібна торгівля, де вони використовуються для гейміфікації. За оцінками, поширення мобільних ігор підвищить попит на ігрові движки в прогнозований період. Наприклад, за даними Асоціації інтерактивних розваг Великобританії (UKIE). З 1 кварталу 2017-2018 років з Apple App Store та Google Play було завантажено близько 21,8 мільярда додатків для мобільних ігор. Це на 15,3% більше, ніж за підсумками першого кварталу 2016–2017 років (UKIE 2018). Крім того, зростаюча популярність доповненої реальності (AR) і віртуальної реальності (VR) також оцінюється як важливий фактор, що сприяє розвитку ринку. За даними Value Market Research, обсяг глобального ринку ігрових движків у 2020 році оцінювався приблизно в 2150 мільйонів доларів США і, за оцінками, зростатиме приблизно на 13. 5% протягом прогнозного періоду 2021-2027 років. Величезний попит на ігрові движки в автомобільному секторі для розробки дизайну UI та UX можна передбачити, щоб покращити попит на ринку в прогнозований період.

Крім того, блокування Covid-19 покращило взаємодію користувачів із кіберспортом та відеоіграми. Це тому, що ігри відволікали користувачів у час соціального дистанціювання. З початку карантину в ігровій індустрії спостерігається величезний час гри та продажі ігор. Наприклад, за оцінками Всесвітнього економічного форуму, у 2020 році світовий ринок відеоігор оцінюватиметься в 159 мільярдів доларів. Він дорівнює приблизно втричі доходу музичної індустрії (57 мільярдів доларів у 2019 році) і приблизно в чотири рази доходу від касових зборів (43 мільярди доларів у 2019 році). Азіатсько-Тихоокеанський регіон займає майже 50% ринку ігор за вартістю, за ним слідує Північна Америка, на яку припадає чверть доходу. однак,

висока вартість програмного забезпечення для постачальників може зашкодити світовому ринку ігрових двигунів у довгостроковій перспективі. Крім того, постійно мінливий попит клієнтів і технології, що швидко змінюються, негативно впливають на ринок. Сучасні тенденції галузі, Очікується, що розширення ігрової індустрії зі зростанням мобільних ігрових додатків, доступних у магазині додатків, надасть вигідні ринкові можливості для гравців ринку. Ринок ігрових движків величезний, з великою кількістю місцевих і глобальних гравців.

Ключові лідери дотримуються кількох стратегій для покращення своїх позицій на ринку, таких як злиття, поглинання, розширення, інновації продуктів та розширення асортименту продуктів, щоб збільшити свою частку ринку на світовому ринку. Наприклад, у березні 2020 року Unity Technologies придбала Artomatix. Це Ірландія - компанія, що базується на програмному забезпеченні, яка використовує AI та нейронні мережі для раціоналізації 3D-художніх процесів. Важливими гравцями, включеними до звіту, є Chukong Tech, Epic Games, Valve Corporation, The Game Creators, Leadwerks Software, Corona Labs, Amazon, Scirra, Sony, GameSalad, Silicon Studio Corp, Unity Technologies, YoYo Games, Marmalade Tech і Crytek.

За типом звіт поділяється на 2D ігрові движки, 3D ігрові движки та 2,5D ігрові движки. 3D-ігрові движки домінують у сегменті типів у 2019 році з максимальною часткою ринку 80%. Це пов’язано з широким використанням 3D-типу в іграх або рольовому моделюванні, моделюванні сценаріїв, 3D-движку та системі частинок. Крім того, 3D-ігрові движки дозволяють ігровим компаніям зменшити витрати, час і робочу силу, оскільки розробники ігор можуть використовувати легко зрозумілі функції движка, таким чином стимулюючи сегмент 3D-ігрових движків у прогнозований період.

4.2 Розрахунок витрат на проєктування

Створення провідного стартапу вимагає ретельного планування та організації. На ранній стадії розвитку важливо створити фіксований місячний або річний бюджет. Для більшості стартапів дотримання цього бюджету є важливою складовою успіху.

Витрати на людські ресурси — виплата заробітної плати, навчання, пільги — можуть становити значну частину загального місячного бюджету проєкту. Насправді стартапи витрачають до 25 відсотків загального доходу на витрати на людські ресурси. Для будь-якого стартапу виділення відповідних коштів на людські ресурси є критичною складовою довгострокового та стійкого успіху. Рахунок витрат які стосуються робітників проєкту приведено в додатку А в таблиці 4.1.

Витрати на розробку проєкту, заробітня плата оформлена в додатку А в таблиці 4.2.

Фінансові підрахунки витрат та прибутку свідчать про рентабельність проєкту. Це дає змогу заробити, а значить на даний рід програмного забезпечення є попит і потенційний клієнт зацікавлений в розвитку подібних технологій. Вкладені інвестиції будуть ефективними що дозволить розвивати проєкт і надалі.

4.3 Обгрунтування необхідності розробки

Основним рушієм розвитку компанії є інвестиції та інвестори що їх приносять. Інвестувати є сенс тільки в прибуткові компанії які можуть примножити вкладені туди кошти. Прибуток можуть принести клієнти які зацікавлені в придбанні послуг або програмного забезпечення, яке вирішує тим чи іншим чином їх проблеми чи вгамовує потреби. Оскільки попит на ігрові двигунці є досить великий, частка ринку цієї галузі тільки росте, а капіталізація компаній в ній збільшується, значить розробка таких двигунців може принести чималий прибуток, залучити інвесторів та дати змогу

розвиватись. Якщо розвивати власні унікальні технології та впроваджувати все більше інновацій ця справа стає дуже вигідною, бо попит на такі проєкти не зменшується а тільки набирає обертів.

Під час написання документації було представлено можливість створення власного унікального ігрового рушія та приведено порівняльну роботу між потенційними конкурентами.

В першу чергу рушій було створено для зручності створення власних ігор та можливістю інтерактивної роботи з 3D графікою. Можливість налаштувати та адаптувати його під власні потреби і задачі є безумовно одною з найбільших його переваг. Відсутність зайвих функцій які б навантажували систему але в проєкті практичного застосунку не мали, та впровадження специфічного функціоналу чи спеціального режиму відтворення даних вигідно вирізняє рушій серед інших для окремих специфічних задач, які часто виникають в геймдеві. Звісно і інші можна адаптувати згідно поставленої задачі, але інші великі рушії набагато важче піддаються модифікації та не такі гнучкі як особистий невеликий настроюваний рушій для невеликих ігор з малим бюджетом чи незалежної розробки.

ВИСНОВКИ

Сучасний темп технологічного розвитку не оминув і ігрові рушії. Це надзвичайно важлива складова величезної ігрової індустрії і обставини змушують винаходити нові та унікальні речі, що могли б надати перевагу над конкурентами. Все більш цікаві технології рендерингу, згладжування за допомогою нейромереж або ж відображення світлових променів з повною їх симуляцією як в реальності роблять цю сферу надзвичайно привабливою та цікавою з точки зору не тільки інвестицій а й реалізації творчого потенціалу та всіх технічних навичків.

Під час написання дипломного проєкту було проведено аналіз ринку, виявлено основні проблеми сучасних рішень які там представлені. Попит на різноманітні ігрові рушії мотивував на написання даного проєкту, в такій специфічній сфері важко підібрати найбільш комфортне і незалежне рішення для реалізації власних ідей.

Створивши детальний план розробки, постановивши задачу, функціонал та принципи роботи проєкту було вирішено які компоненти туди будуть входити а також використання сторонніх бібліотек та інструментів. Для написання логіки було використано мову програмування C++ а також бібліотеку OpenGL.

Створено повноцінний ігровий рушій з повним набором необхідних функцій. Ключовим етапом розробки ПЗ було створення архітектури та способу взаємодії елементів всередині системи. Проведено тестування розробленого програмного засобу, що дозволило виявити та виправити помилки.

В подальшій перспективі можливе розширення функціоналу двигунця, впровадження нових шейдерів та методів пост-обробки, вдосконалення взаємодії з створюваними об’єктами.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Власний Game Engine: як і навіщо створювати ігровий рушій. *Dou*: вебсайт. URL: https://gamedev.dou.ua/blogs/how-and-why-to-create-a-game-engine/ (дата звернення: 06.06.2022);
2. OpenGL SDK – Tutorials. *OpenGL*: вебсайт. URL: https://www.opengl.org/sdk/docs/tutorials/ (дата звернення: 10.04.2022);
3. Algorithms Library. CppReference: вебсайт. URL: https://en.cppreference.com/w/cpp/algorithm (дата звернення: 12.05.2022);
4. C++ в сучасному світі. *Habr*: вебсайт. URL: https://habr.com/ru/company/pvs-studio/blog/259777/ (дата звернення: 22.05.2022);
5. Патерни проєктування. *Refactoring.guru*: вебсайт. URL: https://refactoring.guru/uk/design-patterns/cpp (дата звернення: 03.04.2022);
6. Visual Studio 2019 Документація. *Microsoft.com:* вебсайт: URL: https://docs.microsoft.com/ru-ru/cpp/get-started/tutorial-console-cpp?view=msvc-170&viewFallbackFrom=vs-2022 (дата звернення: 19.04.2022);
7. Вивчення нових можливостей. *Godotengine*: вебсайт. URL: https://docs.godotengine.org/uk/stable/getting_started/introduction/learning_new_features.html (дата звернення: 23.05.2022).

ДОДАТКИ

Додаток А

Розрахунок витрат на реплізацію проєкту

Таблиця А.1 – Основна заробітня плата розробників

№ п/п	Посада виконавця	Оклад, грн/міс.	Відрахування грн/міс.	Кількість чол.	Кількість місяців	Сума, з/п, грн.
1	C++ розробник	6492	1039.06	1	6	5452.40
2	Системний адміністратор	6492	1039.06	1	6	5452.40
3	Девопс	6492	1039.06	1	6	5452.40
4	Тестувальник	6492	1039.06	1	6	5452.40

Таблиця А.2 – Кошторис витрат на проєктування

Найменування статей витрат	Сума, грн	Обґрунтування
1. Зарплата проєктувальників	25968	Оптимальний час виконання проєкту сягає 5 місяців при злагодженій роботі трьох спеціалістів. Посадові оклади узгоджені із тарифним коефіцієнтом кожної посади.

Продовження таблиці А.2

2. Відрахування на соціальні потреби	5712.96	Сума відрахувань від зарплати на соціальні потреби визначається по встановленому нормативу у відсотках від суми заробітної плати (6492 грн/особа) = 22%.
3. Контрагентські роботи і послуги	-	-
4. Витрати на відрядження	-	-
5. Інші прямі витрати	1100	Витратні матеріали та канцелярські товари.
6. Усього прямих витрат	32780.96	Підсумкова вартість витрат пункту 1-5.
7. Накладні витрати	5832	Оплата за опалення, електроенергію та інші супутні послуги протягом 6 місяців роботи над проєктом.

Продовження таблиці А.2

8. Планові накопичення	7722.59	20% від суми прямих і накладних витрат, яка спрямована на преміювання виконавчих осіб проєкту.
9. Усього, кошторисна вартість проєкту	46335.55	Загальна вартість прямих і накладних витрат та планових накопичень.
10. Податок на додану вартість	9267.11	Сума, яка сягає 20% від кошторисної вартості проєкту (визначається по діючому нормативу).
11. Загалом, договірна ціна розробки ЗП	55602.66	Вказує на суму кошторисної вартості роботи та податку на додану вартість і визначає ті витрати на проєктування, що несе підприємство-замовник.

Додаток Б

Програмний код

Лістинг Б1 – Код класу GameObject

```
#include "GameObject.h"
#include "Transform.h"
#include "Mesh.h"
#include "Camera.h"
#include "Bones.h"
#include "Skeleton.h"
#include "AnimationController.h"
#include "MeshRenderer.h"
#include "SkinnedMeshRenderer.h"
#include "Game.h"
GameObject::GameObject() :
	parent(nullptr),
	transform(nullptr),
	mesh(nullptr),
	camera(nullptr),
	bones(nullptr),
	skeleton(nullptr),
	animation_controller(nullptr),
	mesh_renderer(nullptr),
	skinned_mesh_renderer(nullptr),
	light(nullptr),
	destroy_on_load(true)
{}
```

```
GameObject::GameObject(GameObject* parent) :
	parent(parent),
	transform(nullptr),
	mesh(nullptr),
	camera(nullptr),
	bones(nullptr),
	skeleton(nullptr),
	animation_controller(nullptr),
	mesh_renderer(nullptr),
	skinned_mesh_renderer(nullptr),
	light(nullptr),
	destroy_on_load(true)
{}
GameObject::~GameObject() {
}
void GameObject::unload() {
	if (parent == nullptr) {
		for (GameObject* child : children) {
			child->parent = nullptr;

	Game::get().scene_manager.root_game_objects.push_back(child
);
		}
	} else {
		parent->removeChild(this);
		for (GameObject* child : children) {
			child->parent = parent;
			parent->addChild(child);
```

```
            }
        }
        removeAllComponents();
    }
    void GameObject::removeAllComponents()
    {
        if (transform != nullptr) removeTransformComponent();
        if (mesh != nullptr) removeMeshComponent();
        if (camera != nullptr) removeCameraComponent();
        if (bones != nullptr) removeBonesComponent();
        if (skeleton != nullptr) removeSkeletonComponent();

        if (animation_controller != nullptr)
removeAnimationControllerComponent();
        if (mesh_renderer != nullptr)
removeMeshRendererComponent();
        if (skinned_mesh_renderer != nullptr)
removeSkinnedMeshRendererComponent();
        if (light != nullptr) removeLightComponent();
    }
    void GameObject::addChild(GameObject* child) {
        children.push_back(child);
    }
    void GameObject::removeChild(GameObject* child) {
        children.erase(std::remove(children.begin(),
children.end(), child), children.end());
    }
    void GameObject::removeTransformComponent() {
        if (transform->removeContainer(this)) {
```

```
            Game::get().data_manager.remove(transform);
        }
        transform = nullptr;
    }
    void GameObject::removeMeshComponent() {
        if (mesh->removeContainer(this)) {
            Game::get().data_manager.remove(mesh);
        }
        mesh = nullptr;
    }
    void GameObject::removeCameraComponent() {
        if (camera->removeContainer(this)) {
            Game::get().data_manager.remove(camera);
        }
        camera = nullptr;
    }
    void GameObject::removeBonesComponent() {
        if (bones->removeContainer(this)) {
            Game::get().data_manager.remove(bones);
        }
        bones = nullptr;
    }
    void GameObject::removeSkeletonComponent() {
        if (skeleton->removeContainer(this)) {
            Game::get().data_manager.remove(skeleton);
        }
        skeleton = nullptr;
    }
```

```
void GameObject::removeAnimationControllerComponent() {
     if (animation_controller->removeContainer(this)) {
          Game::get().data_manager.remove(animation_controller);
     }
     animation_controller = nullptr;
}
void GameObject::removeMeshRendererComponent() {
     if (mesh_renderer->removeContainer(this)) {
          Game::get().data_manager.remove(mesh_renderer);
     }
     mesh_renderer = nullptr;
}
void GameObject::removeSkinnedMeshRendererComponent() {
     if (skinned_mesh_renderer->removeContainer(this)) {

     Game::get().data_manager.remove(skinned_mesh_renderer);
     }
     skinned_mesh_renderer = nullptr;
}
void GameObject::removeLightComponent() {
     if (light->removeContainer(this)) {
          Game::get().data_manager.remove(light);
     }
     light = nullptr;
}
Transform* GameObject::addTransformComponent() {
     transform = Game::get().data_manager.addData(Transform());
     transform->containers.push_back(this);
```

```
	return transform;
}
Mesh* GameObject::addMeshComponent() {
	mesh = Game::get().data_manager.addData(Mesh());
	mesh->containers.push_back(this);
	return mesh;
}
Camera* GameObject::addCameraComponent() {
	camera = Game::get().data_manager.addData(Camera())
	camera->containers.push_back(this);
	return camera;
}
Bones* GameObject::addBonesComponent() {
	bones = Game::get().data_manager.addData(Bones());
	bones->containers.push_back(this);
	return bones;
}
Skeleton* GameObject::addSkeletonComponent() {
	skeleton = Game::get().data_manager.addData(Skeleton());
	skeleton->containers.push_back(this);
	return skeleton;
}
AnimationController*
GameObject::addAnimationControllerComponent() {
	animation_controller =
Game::get().data_manager.addData(AnimationController());
	animation_controller->containers.push_back(this);
	return animation_controller;
```

```
}

MeshRenderer* GameObject::addMeshRendererComponent() {

     mesh_renderer =
Game::get().data_manager.addData(MeshRenderer());

     mesh_renderer->containers.push_back(this);

     return mesh_renderer;

}

SkinnedMeshRenderer*
GameObject::addSkinnedMeshRendererComponent() {

     skinned_mesh_renderer =
Game::get().data_manager.addData(SkinnedMeshRenderer());

     skinned_mesh_renderer->containers.push_back(this);

     return skinned_mesh_renderer;

}

Light* GameObject::addLightComponent() {

     light = Game::get().data_manager.addData(Light());

     light->containers.push_back(this);

     return light;

}

Transform* GameObject::addTransformComponent(const Transform&
transform) {

     this->transform =
Game::get().data_manager.addData(transform);

     this->transform->containers.push_back(this);

     return this->transform;

}

Mesh* GameObject::addMeshComponent(const Mesh& mesh) {

     this->mesh = Game::get().data_manager.addData(mesh);

     this->mesh->containers.push_back(this);

     return this->mesh;
```

```
}
Camera* GameObject::addCameraComponent(const Camera& camera) {
     this->camera = Game::get().data_manager.addData(camera);
     this->camera->containers.push_back(this);
     return this->camera;
}
Bones* GameObject::addBonesComponent(const Bones& bones) {
     this->bones = Game::get().data_manager.addData(bones);
     this->bones->containers.push_back(this);
     return this->bones;
}
Skeleton* GameObject::addSkeletonComponent(const Skeleton&
skeleton) {
     this->skeleton =
Game::get().data_manager.addData(skeleton);
     this->skeleton->containers.push_back(this);
     return this->skeleton;
}
AnimationController*
GameObject::addAnimationControllerComponent(const
AnimationController& animation_controller) {
     this->animation_controller =
Game::get().data_manager.addData(animation_controller);
     this->animation_controller->containers.push_back(this);
     return this->animation_controller;
}
MeshRenderer* GameObject::addMeshRendererComponent(const
MeshRenderer& mesh_renderer) {
     this->mesh_renderer =
Game::get().data_manager.addData(mesh_renderer);
```

```
    this->mesh_renderer->containers.push_back(this);
    return this->mesh_renderer;
}
SkinnedMeshRenderer*
GameObject::addSkinnedMeshRendererComponent(const
SkinnedMeshRenderer& skinned_mesh_renderer) {
    this->skinned_mesh_renderer =
Game::get().data_manager.addData(skinned_mesh_renderer);
    this->skinned_mesh_renderer->containers.push_back(this);
    return this->skinned_mesh_renderer;
}
Light* GameObject::addLightComponent(const Light& light) {
    this->light = Game::get().data_manager.addData(light);
    this->light->containers.push_back(this);
    return this->light;
}
Transform* GameObject::getTransformComponent() {
    return transform;
}
Mesh* GameObject::getMeshComponent() {
    return mesh;
}
Camera* GameObject::getCameraComponent() {
    return camera;
}
Bones* GameObject::getBonesComponent() {
    return bones;
}
Skeleton* GameObject::getSkeletonComponent() {
```

```
	return skeleton;

}

AnimationController*
GameObject::getAnimationControllerComponent() {

	return animation_controller;

}

MeshRenderer* GameObject::getMeshRendererComponent() {

	return mesh_renderer;

}

SkinnedMeshRenderer*
GameObject::getSkinnedMeshRendererComponent() {
```

Продовження лістингу Б1

```
	return skinned_mesh_renderer;

}

Light* GameObject::getLightComponent() {

	return light;

}

void GameObject::update() {

	if(Game::get().is_debug_mode)
Logger::log(directory_and_name + " updating");

	if (animation_controller != nullptr) animation_controller-
>update();

	if (camera != nullptr) camera->update();

	if (transform != nullptr) transform->update();

	if (mesh != nullptr) mesh->update();

	if (bones != nullptr) bones->update();

	if (skeleton != nullptr) skeleton->update();

	if (mesh_renderer != nullptr) mesh_renderer->update();

	if (skinned_mesh_renderer != nullptr)
skinned_mesh_renderer->update();
```

```
    if (light != nullptr) light->update();

    if (Game::get().is_debug_mode)
Logger::log(directory_and_name + " finished updating");

    for (GameObject* child : children) {

        child->update();

    }

    if (Game::get().is_debug_mode)
Logger::log(directory_and_name + " children finished updating");

    }
```

Лістинг Б2 – Код класу Mtx

```
    #include "Mtx.h"
    #include <string>
    #include <sstream>
    #include <iomanip>
    #include <cmath>
    #include <assimp/scene.h>
    #include "Logger.h"
    # define M_PI           3.14159265358979323846
    float mtx::to_radians(float const &radians) {
        return radians / 360 * (float)M_PI * 2;
    }
    mtx::Vec2::Vec2() : x(0), y(0) { }

    mtx::Vec2::Vec2(float x, float y) : x(x), y(y) { }
    std::string mtx::Vec2::to_string() const {
        std::string rtn;
        rtn.reserve(50);
        std::string str = std::to_string(x);
        str.erase(str.find_last_not_of('0')        +        1,
    std::string::npos);
        str.erase(str.find_last_not_of('.')        +        1,
    std::string::npos);
        rtn += str + "\n";
        str = std::to_string(y);
```

```
        str.erase(str.find_last_not_of('0')        +        1,
    std::string::npos);
        str.erase(str.find_last_not_of('.')        +        1,
    std::string::npos);
        rtn += str + "\n";
        return rtn;
    }
    mtx::Vec2 mtx::Vec2::operator+(Vec2 const &right) const {
        return Vec2(x + right.x, y + right.y);
    }
    mtx::Vec2 mtx::Vec2::operator+(float const &right) const {
        return Vec2(x + right, y + right);
    }
    mtx::Vec2& mtx::Vec2::operator+=(Vec2 const &right) {
        x += right.x;
        y += right.y;
        return *this;
    }
    mtx::Vec2& mtx::Vec2::operator+=(float const &right) {
        x += right;
        y += right;
        return *this;
    }
    mtx::Vec2 mtx::Vec2::operator-(Vec2 const &right) const {
```

Продовження лістингу Б2

```
        return Vec2(x - right.x, y - right.y);
    }
    mtx::Vec2 mtx::Vec2::operator-(float const &right) const {
        return Vec2(x - right, y - right);
    }
    mtx::Vec2& mtx::Vec2::operator-=(Vec2 const &right) {
        x -= right.x;
        y -= right.y;
        return *this;
    }
```

```
mtx::Vec2& mtx::Vec2::operator-=(float const &right) {
     x -= right;
     y -= right;
     return *this;
}
mtx::Vec2 mtx::Vec2::operator*(float const &right) const {
     return Vec2(x * right, y * right);
}
mtx::Vec2& mtx::Vec2::operator*=(float const &right) {
     x *= right;
     y *= right;
     return *this;
}
mtx::Vec2 mtx::Vec2::operator/(float const &right) const {
     return Vec2(x / right, y / right);
}
mtx::Vec2& mtx::Vec2::operator/=(float const &right) {
     x /= right;
     y /= right;
     return *this;
}
bool mtx::Vec2::operator==(Vec2 const &right) const {
     return x == right.x && y == right.y;
}
bool mtx::Vec2::operator!=(Vec2 const &right) const {
     return x != right.x || y != right.y;
}
float* mtx::Vec2::data_ptr() {
     return &x;
}
mtx::Vec3::Vec3() :x(0), y(0), z(0) {}
mtx::Vec3::Vec3(float x, float y, float z) : x(x), y(y),
z(z) {}
mtx::Vec3::Vec3(const Vec3 &vector) : x(vector.x),
y(vector.y), z(vector.z) {}
```

```
mtx::Vec3::Vec3(const aiVector3D& vector) : x(vector.x),
y(vector.y), z(vector.z) {}
std::string mtx::Vec3::to_string() const {
     std::string rtn;
     rtn.reserve(50);
     std::string str = std::to_string(x);
     str.erase(str.find_last_not_of('0')+1,
std::string::npos);
     str.erase(str.find_last_not_of('.')+1,
std::string::npos);
     rtn += "[ x: " + str + ", ";
     str = std::to_string(y);
     str.erase(str.find_last_not_of('0')+1,
std::string::npos);
     str.erase(str.find_last_not_of('.')+1,
std::string::npos);
     rtn += "y: " + str + ", ";
     str = std::to_string(z);
     str.erase(str.find_last_not_of('0')        +        1,
std::string::npos);
     str.erase(str.find_last_not_of('.')        +        1,
std::string::npos);
     rtn += "z: " + str + " ]";
     return rtn;
}
mtx::Vec3 mtx::Vec3::operator-() const {
     return Vec3(-x, -y, -z);
}
mtx::Vec3 mtx::Vec3::operator+(Vec3 const &right) const {
     return Vec3(x + right.x, y + right.y, z + right.z);
}
mtx::Vec3 mtx::Vec3::operator+(float const &right) const {
     return Vec3(x + right, y + right, z + right);
}
mtx::Vec3& mtx::Vec3::operator+=(Vec3 const &right) {
```

```
        x += right.x;
        y += right.y;
        z += right.z;
        return *this;
    }
    mtx::Vec3& mtx::Vec3::operator+=(float const &right) {
        x += right;
        y += right;
        z += right;
        return *this;
    }
```

Продовження лістингу Б2

```
    mtx::Vec3 mtx::Vec3::operator-(Vec3 const &right) const {
        return Vec3(x - right.x, y - right.y, z - right.z);
    }
    mtx::Vec3 mtx::Vec3::operator-(float const &right) const {
        return Vec3(x - right, y - right, z - right);
    }
    mtx::Vec3& mtx::Vec3::operator-=(Vec3 const &right) {
        x -= right.x;
        y -= right.y;
        z -= right.z;
        return *this;
    }
    mtx::Vec3& mtx::Vec3::operator-=(float const &right) {
        x -= right;
        y -= right;
        z -= right;
        return *this;
    }
    mtx::Vec3 mtx::Vec3::operator*(float const &right) const {
        return Vec3(x * right, y * right, z * right);
    }
    mtx::Vec3 mtx::Vec3::operator*(Vec3 const &right) const {
```

```
    return Vec3(x * right.x, y * right.y, z * right.z);
}
mtx::Vec3& mtx::Vec3::operator*=(float const &right) {
    x *= right;
    y *= right;
    z *= right;
    return *this;
}
mtx::Vec3& mtx::Vec3::operator*=(Vec3 const &right) {
    x *= right.x;
    y *= right.y;
    z *= right.z;
    return *this;
}
mtx::Vec3 mtx::Vec3::operator/(float const &right) const {
    return Vec3(x / right, y / right, z / right);
}
mtx::Vec3& mtx::Vec3::operator/=(float const &right) {
    x /= right;
    y /= right;
    z /= right;
```

Продовження лістингу Б2

```
    return *this;
}
bool mtx::Vec3::operator==(Vec3 const &right) const {
    return x == right.x && y == right.y && z == right.z;
}
bool mtx::Vec3::operator!=(Vec3 const &right) const {
    return x != right.x || y != right.y || z != right.z;
}
float mtx::Vec3::magnitude() {
    return std::sqrt(x * x + y * y + z * z);
}
float mtx::Vec3::magnitude_squared() {
    return x * x + y * y + z * z;
```

```
}
mtx::Vec3& mtx::Vec3::normalize() {
     float mag = magnitude();
     if (mag > 0) {
          *this /= mag;
     }
     return (*this);
}
```

Продовження лістингу Б2

```
mtx::Vec3 mtx::Vec3::normal() {
     float mag = magnitude();
     if (mag > 0) {
          return Vec3(x / mag, y / mag, z / mag);
     }
     else {
          return Vec3(*this);
     }
}
float mtx::Vec3::dot(Vec3 const &right) const {
     return x * right.x + y * right.y + z * right.z;
}
mtx::Vec3 mtx::Vec3::cross(Vec3 const &right) const {
     return Vec3(
          y * right.z - z * right.y,
          z * right.x - x * right.z,
          x * right.y - y * right.x);
}
mtx::Vec3  mtx::Vec3::lerp(Vec3  const  &right,  float  const
time) {
     return  (*this) * (1 - time) + right * time;
}
mtx::Vec3 mtx::Vec3::project_onto(Vec3 const &right) {
     return right * (dot(right) / magnitude_squared());
}
mtx::Vec3& mtx::Vec3::rotate(Quaternion &quat) {
```

```
        quat.normalize();
        *this     =     (quat     *     Quaternion(*this)     *
quat.conjugate()).vector;
        return (*this);
}
float* mtx::Vec3::data_ptr() {
        return &x;
}
mtx::Matrix4::Matrix4() : array{ 0.0f, 0.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,
0.0f, 0.0f } {}
mtx::Matrix4::Matrix4(bool const &identity) : array{ 1.0f,
0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f,
0.0f, 0.0f, 0.0f, 0.0f, 1.0f } {
}
mtx::Matrix4::Matrix4(float const &num) : array{ num, num,
num, num, num, num, num, num, num, num, num, num, num, num,
num, num } {}
mtx::Matrix4::Matrix4(float a1, float a2, float a3, float
a4, float b1, float b2, float b3, float b4, float c1, float
c2, float c3, float c4, float d1, float d2, float d3, float
d4) :
        array{ a1, a2, a3, a4, b1, b2, b3, b4, c1, c2, c3, c4,
d1, d2, d3, d4 } {}
mtx::Matrix4::Matrix4(Matrix4 const &original) : array() {
        for (int i = 0; i < 16; i++) {
                array[i] = original[i];
        }
}
mtx::Matrix4::Matrix4(aiMatrix4x4 ai_matrix) {
        array[0] = ai_matrix.a1;
        array[1] = ai_matrix.a2;
        array[2] = ai_matrix.a3;
        array[3] = ai_matrix.a4;
        array[4] = ai_matrix.b1;
```

```
        array[5] = ai_matrix.b2;
        array[6] = ai_matrix.b3;
        array[7] = ai_matrix.b4;
        array[8] = ai_matrix.c1;
        array[9] = ai_matrix.c2;
        array[10] = ai_matrix.c3;
        array[11] = ai_matrix.c4;
        array[12] = ai_matrix.d1;
        array[13] = ai_matrix.d2;
        array[14] = ai_matrix.d3;
        array[15] = ai_matrix.d4;
    }
    std::string mtx::Matrix4::to_string() {
        std::ostringstream rtn;
        std::string numbers[16];
        int length = 0;
        int length_r = 0;
        for (int i = 0; i < 16; i++) {
            numbers[i] = std::to_string(array[i]);
            numbers[i].erase(numbers[i].find_last_not_of('0')
    + 1, std::string::npos);
            numbers[i].erase(numbers[i].find_last_not_of('.')
    + 1, std::string::npos);
            if ((i + 1) % 4 != 0 && numbers[i].length() >
    length) {
                length = numbers[i].length();
            }
            else if (numbers[i].length() > length_r) {
                length_r = numbers[i].length();
            }
        }
        for (int i = 0; i < 16; i++) {
            if (i % 4 == 0) {
                rtn << "[ ";
            }
```

```
            if ((i + 1) % 4 != 0) {
                rtn << std::setw(length + 1) << std::left <<
numbers[i];
            }
            else {
                rtn << std::setw(length_r + 1) << std::left
<< numbers[i];
                rtn << "]\n";
            }
        }
        return rtn.str();
}
std::string mtx::Matrix4::to_string(int num_tabs) {
        std::ostringstream rtn;
        std::string numbers[16];
        int length = 0;
        int length_r = 0;
        for (int i = 0; i < 16; i++) {
            numbers[i] = std::to_string(array[i]);
            numbers[i].erase(numbers[i].find_last_not_of('0')
+ 1, std::string::npos);
            numbers[i].erase(numbers[i].find_last_not_of('.')
+ 1, std::string::npos);
            if ((i + 1) % 4 != 0 && numbers[i].length() >
length) {
                length = numbers[i].length();
            }
            else if (numbers[i].length() > length_r) {
                length_r = numbers[i].length();
            }
        }
        for (int i = 0; i < 16; i++) {
            if (i % 4 == 0) {
                for (int j = 0; j < num_tabs; j++) {
                    rtn << "  ";
```

```
                    }
                    rtn << "[ ";
               }
               if ((i + 1) % 4 != 0) {
                    rtn << std::setw(length + 1) << std::left <<
numbers[i];
               } else {
                    rtn << std::setw(length_r + 1) << std::left
<< numbers[i];
                    rtn << "]\n";
               }
          }
          return rtn.str();
     }
```