МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВОДНОГО ГОСПОДАРСТВА ТА ПРИРОДОКОРИСТУВАННЯ
Навчально-науковий інститут автоматики, кібернетики та обчислювальної техніки
Кафедра комп'ютерних наук та прикладної математики

«До захисту допущена»
Завідувач кафедри комп'ютерних наук
та прикладної математики
_____ Турбал Ю.В.
«_____» _____ 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Розробка рушія для створення комп'ютерних ігор на основі специфікацій OpenGL»

Виконав: Мишенко Владислав Олександрович _____
(прізвище, ім'я, по батькові) (підпис)

група ІПЗ-41

Керівник: старший викладач Харів Н.О. _____
(науковий ступінь, вчене звання, посада, прізвище та ініціали) (підпис)

Рівне – 2023

Національний університет водного господарства та природокористування

Навчально-науковий інститут автоматики, кібернетики та обчислювальної техніки
Кафедра **комп'ютерних наук та прикладної математики**
Рівень вищої освіти **бакалавр**
Галузь знань **12 «Інформаційні технології»**
Спеціальність **121 «Інженерія програмного забезпечення»**

«ЗАТВЕРДЖУЮ»
Завідувач кафедри
комп'ютерних наук та прикладної математики
д.т.н., професор Турбал Ю.В.

«____» _____2023 року

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Мишенко Владислав Олександрович
(прізвище, ім'я, по батькові)

1. Тема роботи *«Розробка рушія для створення комп'ютерних ігор на основі специфікацій OpenGL»*

керівник роботи *Харів Наталія Олексіївна, старший викладач кафедри комп'ютерних наук та прикладної математики*

затверджені наказом вищого навчального закладу від « 19 » *квітня 2023 року С* №-449

2. Термін здачі студентом закінченої роботи 29.05.2023

3. Вихідні дані до роботи*: технічні вимоги до застосунку*

4. Зміст розрахунково-пояснювальної записки *Перший розділ. Постановка задачі та дослідження структури рушіїв. Другий розділ. Методи реалізації систем рушія на основі проаналізованих даних. Третій розділ. Опис реалізації систем та підсистем рушія, а також огляд користування та підключення створеного рушія.*

5. Перелік графічного матеріалу *мультимедійна презентація*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Розділ 1	*Харів Н.О., ст. викладач*		
Розділ 2	*Харів Н.О., ст. викладач*		
Розділ 3	*Харів Н.О., ст. викладач*		

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	*Опрацювання літератури на задану тему.*		
2	*Вивчення теоретичного матеріалу, аналіз наявних рушіїв*		
3	*Аналіз вивченого матеріалу, створення алгоритмів та методу реалізації рушія.*		
4	*Програмна реалізація з врахуванням поставлених завдань.*		
5	*Тестування та виправлення помилок.*		
6	*Оформлення теоретичної частини*		

Студент _____ (*Мишенко В.О.*)

Керівник кваліфікаційної роботи _____ (Харів Н.О)

ЗМІСТ

РЕФЕРАТ

використання сучасних технологій для написання ігрового рушія для розробки комп’ютерних ігор.

Кваліфікаційна бакалаврська робота на здобуття освітнього ступеня бакалавр за спеціальністю 121 «Інженерія Програмного Забезпечення». – Національний університет водного господарства та природокористування. – Рівне, 2023.

Бакалаврська робота: n сторінок, n рисунка, n таблиць та n літературних джерел.

Метою роботи є написання ігрового рушія для розробки комп’ютерних ігор з використанням сучасних технологій.

В роботі розглянуто основні методи та інструменти розробки рушія та їхні специфікації. Наведені математичні формули при обчислення необхідних даних, таких, наприклад, які необхідні при відмальовці об’єкта на сцені. Наведені приклади різних інтерфейсів для 3D і 2D графіки та їх порівняння.

Ключові слова: OpenGL; GLEW; GLFW; Cmake; GLSL; овнер; API, буфер, вертекс, шейдер.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

OpenGL – Open Graphics Library

GLEW – The OpenGL Extension Wrangler Library

GLFW – Graphics Library Framework

GLSL – OpenGL Shading Language

API – Application Programming Interface

VS - vertex shader

FS - fragment (pixel) shader

VAO - Vertex Arrays Object

VBO - Vertex Buffer Object

IBO - Index Buffer Object

GPU - графічний процесор

CPU - центральний процесор

ВСТУП

Ігри є невід'ємною частиною дозвілля сучасних людей. Сотні мільйонів людей одночасно грають в ігри на різних платформах: будь-то телефони, комп'ютери чи консолі. Це величезна та багато мільярдна індустрія, яка з кожним роком росте та розвивається.

Основною платформою для ігор є персональні комп'ютери, так я багато людей мають їх у власності не тільки для розваг, а у вільний від роботи/навчання час проводять час в цьому виді дозвілля. Розвиток комп'ютерних технологій дуже цікавий та обширний і одним із чинників прогресу є саме комп'ютерні ігри, які змушують виробників збільшувати продуктивність комплектуючих, тим самим збільшуючи можливість розвитку наступних поколінь ігор для розробників.

Сучасні ігри вражають своєю гіпер реалістичною графікою, швидкістю відгуку, звуковими ефектами, вражаючою фізикою і тому подібного. Кількість людей, які працюють над грою можуть досягати декількох тисяч осіб і потрібний інструмент, який допоможе оптимізувати всю роботи та уніфікувати процеси та методи розробки для кожного, що дасть змогу легко підтримувати та масштабувати проект. Таким інструментом є ігровий рушій, який об'єднує безліч інструментів та дозволяє швидко розробляти ігри, маючи в собі стандартні можливості, які можна розширювати.

Базою кожного рушія є графічне API, а саме певний інтерфейс, який незалежно від мови програмування, буде звертатись до апаратної складової для рендеру простих примітивів, які в свою чергу можуть складати складні фігури чи моделі. Кожне графічне API має свої переваги та недоліки, тому при виборі потрібно детально врахувати кожну деталь.

Ця бакалаврська робота має на меті заглибитись у сферу розробки ігрових рушіїв, зосередившись на використанні можливостей графічного API OpenGL для рендерингу в реальному часі та інтерактивного ігрового процесу. Беручись за цей проект, ми прагнемо отримати всебічне розуміння фундаментальних концепцій і методів, пов'язаних з побудовою ігрового рушія, і дослідити, як

OpenGL, широко використовувана графічна бібліотека з відкритим вихідним кодом, може слугувати надійною основою для цього. А також дослідити вплив на продуктивність та оптимізацію, пов'язані з використанням різних методів розробки на базі графічного API OpenGL. Проводячи ретельне тестування та аналіз, буде представлене дослідження різних технік рендерингу, програмування шейдерів та методи відображення текстур для досягнення оптимального рівня продуктивності.

Проект має за ціль дослідити виклики та міркування, пов'язані зі створенням крос-платформного ігрового рушія. Універсальність графічного API OpenGL дозволяє розробляти ігри, які можуть без проблем працювати на різних операційних системах, включаючи Windows, macOS та Linux. А також в результаті дана робота зробить внесок в існуючий обсяг знань в області розробки ігрових рушіїв. Документуючи проектні рішення, деталі реалізації та оцінки продуктивності, ця робота має на меті надати цінний ресурс для майбутніх розробників, зацікавлених у вивченні розробки ігрових рушіїв за допомогою OpenGL.

РОЗДІЛ 1
ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕНЬ

Перш за все потрібно чітко визначити цілі та окреслити межі враховуючи запити та потреби, крім того проаналізувати та оцінити можливості аналогічних рушіїв, які представлені на ринку. Крім того, потрібно обрати інструменти та методи розробки - все це буде описано в даному розділі.

1.1 Задача розробки ігрового рушія

Метою даної дипломної роботи є розробка ігрового рушія на основі відкритого графічного інтерфейсу, а також дослідити та порівняти наявні рушії.

Об’єктом дослідження є інструменти та комп’ютерні технології для розробки рушія.

Предметом дослідження є методи розробки основних систем ігрового рушія.

Для вирішення поставленої задачі - сформовані наступні завдання:

- Проаналізувати існуючі графічні інтерфейси, інструменти та технології, які використовуються в сучасних рушіях.
- Обрати основні системи, необхідні для програмного продукту
- Створити структуру програми.
- Розробити рушій для спрощення розробки ігор.

Потенційним користувачем даного ігрового рушія можуть бути маленькі інді студії чи команди, у яких немає великих бюджетів та вимоги до гри набагато менші ніж в ААА ігор.

Ігровий рушій розроблений за допомогою мови програмування C++, на основі 2017 стандарту в інтегрованому середовищі розробки Visual Studio 2019.

1.2 Характеристика та функції ігрового рушія

1.2.1 Графічний конвеєр та основа комп’ютерної графіки

Графічний конвеєр – це низка етапів і процесів, які виконує система комп'ютерної графіки (графічне API) для перетворення абстрактних геометричних і візуальних даних у кінцеве зображення, що рендериться. Він являє собою потік даних і операцій, пов'язаних з перетворенням 3D-моделей, текстур та інших графічних даних у пікселі на екрані. Кожне зображення складається з елементів, а саме графічних примітивів.

У комп'ютерній графіці примітив - це базова геометрична форма або елемент, який використовується як будівельний блок для побудови більш складних об'єктів або сцен. Примітиви - це фундаментальні об'єкти, які формують основу комп'ютерних зображень.

Існує кілька типів примітивів, які зазвичай використовуються в комп'ютерній графіці, зокрема:

- Точки - найпростіший примітив, представлений однією координатою у просторі. Точки часто використовуються для представлення частинок, джерел світла або як будівельні блоки для більш складних об'єктів.
- Лінії - утворені поєднанням двох точок. Лінії можуть бути прямими або кривими і використовуються для створення каркасних моделей, контурів або штрихів.
- Багатокутники - складаються із замкненої фігури з прямими краями, утвореної з'єднанням декількох вершин. Трикутники та чотирикутники є найпоширенішими полігональними примітивами. (Рисунок 1.1)
- Криві - визначають плавні, безперервні контури. Криві зазвичай використовуються для створення плавних ліній або контуріві. До популярних типів кривих належать криві Безьє та сплайнові криві.

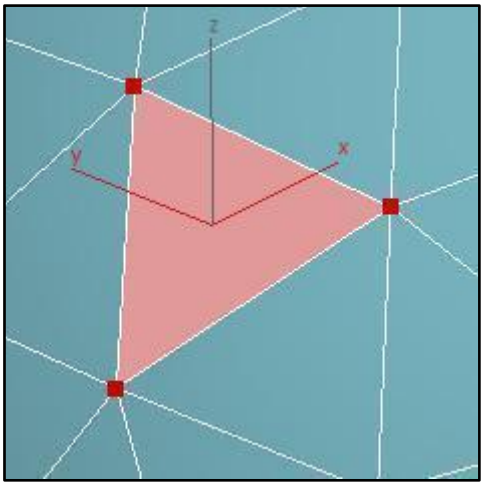

Рис. 1.1. Один із примітивів - трикутник (полігон)

Шейдери - це невеликі програми, що виконуються на графічному прискорювачі (GPU). Ці програми виконуються для кожної конкретної ділянки графічного конвеєра. Якщо описувати шейдери найпростішим способом, то шейдери - це не більше ніж програми, що перетворюють входи у виходи. Шейдери зазвичай ізольовані один від одного, і не мають механізмів комунікації між собою, крім згаданих вище входів і виходів. (Рисунок 1.2)

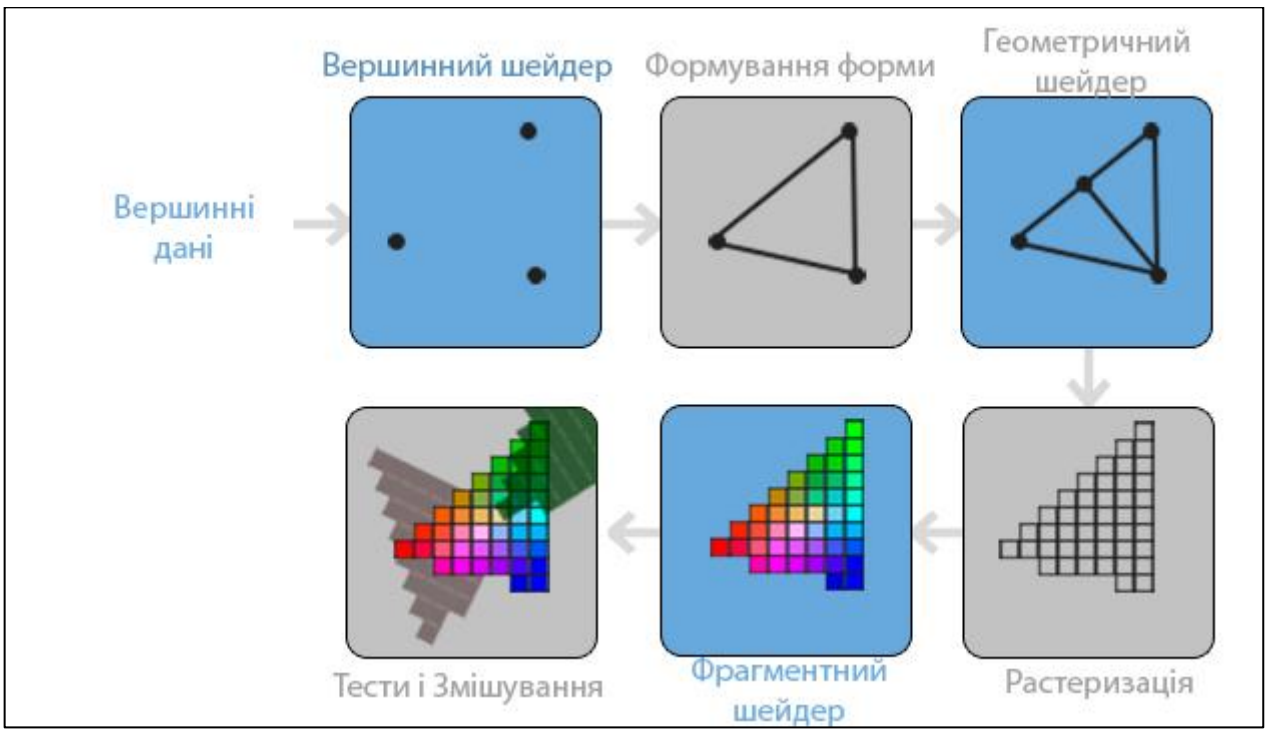

Рис. 1.2. Графічний конвеєр шейдерів

Процес побудови примітиву вимагає лише визначення розташування точок цього самого примітиву та відповідно написаних шейдерів, в залежності від того, що саме ми повинні зробити з цим примітивом: унікальні ефекти для прикладу, коли хочемо візуалізувати ефект хвиль, то потрібно обраховувати полігональну сітку (сітку, що складається з трикутних примітивів).

1.2.1 Графічне API

Графічні API (інтерфейси прикладного програмування) - це важливі програмні інтерфейси, які дозволяють розробникам взаємодіяти з графічним обладнанням комп'ютерної системи та використовувати його. Вони надають стандартизований набір функцій, протоколів та інструментів, які абстрагуються від складнощів базового графічного обладнання і дозволяють створювати та маніпулювати графікою у додатках.

Графічні API слугують мостом між програмним додатком і графічним обладнанням, надаючи розробникам можливість спілкуватися з графічним процесором (GPU) і використовувати його потужність для рендерингу 2D і 3D графіки, застосування шейдерів, обробки текстур і матеріалів та виконання інших операцій, пов'язаних з графікою.

Існує кілька популярних графічних API, зокрема OpenGL, DirectX, Vulkan. Кожен API має свої особливості, можливості та сумісність з різними платформами та обладнанням.

OpenGL – це широко використовуваний графічний API з відкритим вихідним кодом, який надає низькорівневий, крос-платформний інтерфейс для рендерингу 2D і 3D графіки (лого API можна оглянути на рисунку 1.3). Він пропонує широкий спектр можливостей та широку апаратну сумісність. OpenGL дозволяє ефективно використовувати ресурси графічного процесора, що робить його придатним для додатків рендерингу в реальному часі. Однак його низькорівнева природа вимагає ручного керування графічним конвеєром і може мати круту криву навчання.

Рис 1.3. Лого графічного API OpenGL

DirectX – розроблений компанією Microsoft, DirectX - це графічний API, що використовується переважно на платформах Windows (лого API можна оглянути на рисунку 1.4). Він надає повний набір інструментів та бібліотек для розробки ігор, включаючи Direct3D для 3D-рендерингу, Direct2D для 2D-графіки та DirectCompute для обчислень на базі графічного процесора. DirectX пропонує високу продуктивність і тісну інтеграцію з системами Windows, але йому бракує крос-платформної підтримки, що може обмежувати портативність ігор.

Рис. 1.4. Лого графічного API DirectX

Vulkan – це сучасний крос-платформний графічний API, розроблений для забезпечення високопродуктивного доступу до можливостей графічного

процесора. Він пропонує низькорівневий контроль над графічним обладнанням, що дозволяє розробникам оптимізувати конвеєри рендерингу для максимальної ефективності (лого API можна оглянути на рисунку 1.5). Vulkan відомий завдяки зниженому навантаженню на процесор і підтримці багатопотоковості, що робить його придатним для критично важливих до продуктивності додатків. Однак його складність і крива навчання можуть бути складними для початківців і можуть вимагати більш просунутих знань програмування.

Рис. 1.5. Лого графічного API Vulkan

Для того щоб обрати графічне API потрібно зважити всю плюси та мінуси кожного з них і передусім скласти список необхідностей, які постають перед створення гри і які мають бути її можливості.

1.3 Існуючі рішення рушіїв

Сьогодні на ринку представлена велика кількість рушіїв для розробки ігор і кожен з них має свої особливості, функціонал і внутрішні інструменти.

Unity - один з найпопулярніших і найпоширеніших ігрових рушіїв, доступних сьогодні. Він надає повний набір інструментів та функцій для розробки ігор на різних платформах, включаючи настільні, мобільні та консолі. Unity підтримує широкий спектр мов програмування, пропонує великий магазин ресурсів та має потужну спільноту для підтримки та ресурсів. Однак Unity може бути ресурсоємним і має круту криву навчання для початківців.

Рис. 1.6. Логотип Unity

Unreal Engine - розроблений Epic Games, Unreal Engine відомий своєю приголомшливою графікою та візуальною точністю. Він пропонує потужний набір інструментів, гнучку систему візуальних скриптів і велику документацію. Unreal Engine чудово підходить для створення висококласних, візуально вражаючих ігор і підтримує різні платформи. Однак його складна архітектура та оптимізація продуктивності можуть створювати проблеми для новачків, і він вимагає більш глибокого розуміння програмування.

Рис 1.7 - Логотип Unreal Engine

CryEngine - це потужний ігровий рушій, відомий своїми розширеними можливостями рендерингу та реалістичною візуалізацією. Він надає широкий спектр можливостей, включаючи просунуте фізичне моделювання, динамічне освітлення та інтуїтивно зрозумілий редактор рівнів. CryEngine ідеально підходить для створення візуально приголомшливих середовищ у відкритому світі, але може мати крутішу криву навчання порівняно з іншими рушіями.

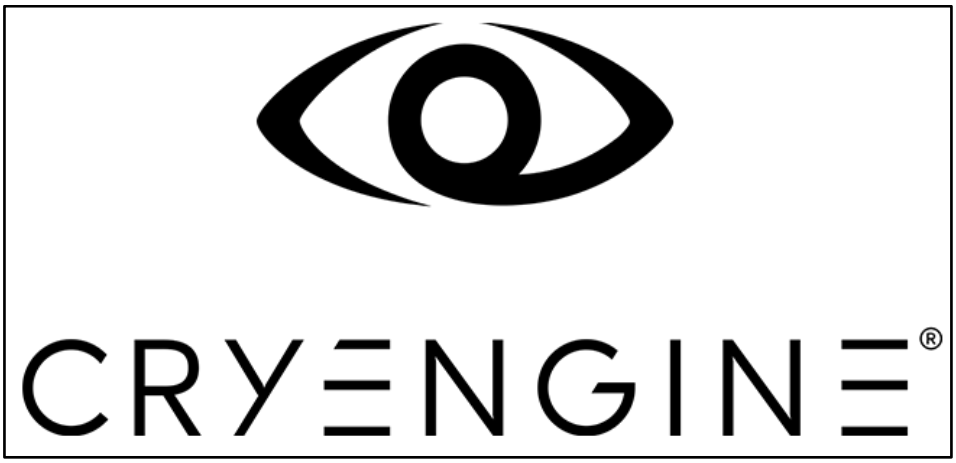

Рисунок 1.8 - Логотип CryEngine

Таблиця 1.1 - Порівняльна характеристика ігрового рушія з наявними аналогами.

Характеристика	Unity	Unreal Engine	CryEngine	TestOpenGL
Мови програмування	C#, Java	C++	C++	C++
Автоматична генерація проекта в залежності від машини	+	+	+	+
Серіалізація даних	повна	повна	повна	часткова

Підтримка мультиплатформ	+	+	+	-
Продуктивність	обмежена через наявну архітектуру та мови програмування	бездоганна	бездоганна	бездоганна
Поріг входу	середній	високий	високий	середній
Ціна	наявна безкоштовна версія, але з обмеженим функціоналом; від 40$ на місяць	наявна безкоштовна версія, але з обмеженим функціоналом; від 19$ за місяць. 5% роялті з продажів гри	від 9.90$ на місяць. 5% роялті з продажів гри	безкоштовно

Проаналізувавши таблицю можна зробити висновки, що розроблений ігровий рушій є в певній мірі унікальним, так як забезпечує базовий функціонал та можливості для створення гри, але головною його перевагою є безкоштовність, що виділяє його від інших рушіїв.

1.4 Засоби реалізації

1.4.1 Графічне API OpengGL

Проаналізувавши всі доступні графічні API вибір був зроблений на користь OpenGL, оскільки він має підтримку кросплатформності, а також відносно легший в освоєнні та використанні ніж Vulkan. Загалом весь процес рендеру (графічний конвеєр) має ось такий вигляд на рисунку 1.9.

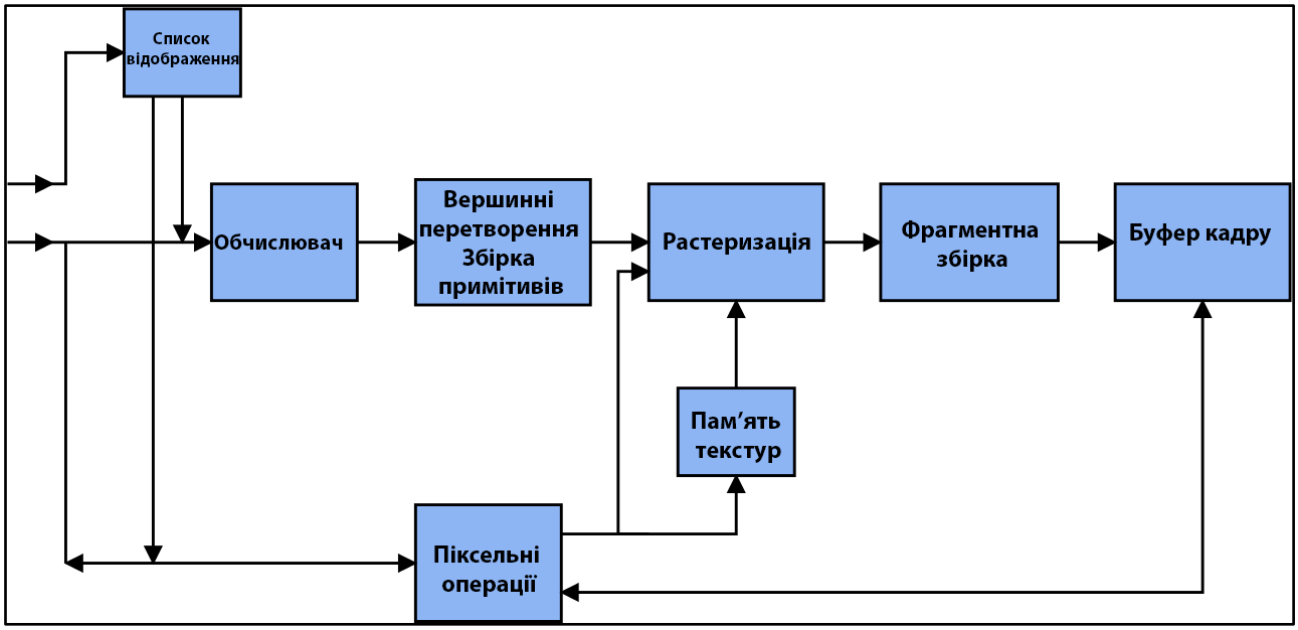

Рис. 1.9. Графічний конвеєр графічного API OpenGL

Так як OpenGL це лише всього API, тобто інтерфейс або іншими словами специфікація, то для того щоб використовувати її нам потрібні відповідні бібліотеки з потрібними для нас функціями.

Однією з таких допоміжних бібліотек є GLFW - це легка бібліотека з відкритим вихідним кодом, яка надає простий і незалежний від платформи інтерфейс для створення вікон, контекстів і обробки подій введення в додатках OpenGL. Вона слугує основою для створення крос-платформних додатків на основі OpenGL, абстрагуючись від специфічних для платформи деталей і надаючи узгоджений API. GLFW пропонує ряд можливостей і функцій, необхідних для розробки програм на OpenGL, включаючи створення і керування вікнами, створення контексту OpenGL, обробку вводу (клавіатура, миша і джойстик), а також обробку системних подій, таких як зміна розміру вікна і закриття. Він також підтримує роботу з декількома моніторами та надає функції керування часом і курсором.

Знову ж оскільки OpenGL - лише інтерфейс, то реалізація лягає на плечі розробників відеокарт. З цієї причини, оскільки існує безліч реалізацій OpenGL, реальне розташування OpenGL функцій не є доступним на етапі компіляції і їх

доводиться отримувати на етапі виконання. Фактично отримання адрес функцій лягає на плечі програміста. Процес отримання адрес специфічний для кожної платформи, для Windows це виглядає приблизно так:

```
// Визначаємо прототип функції
typedef void (*GL_GENBUFFERS) (GLsizei, GLuint*);
// Знаходимо цю функцію в реалізації та зберігаємо покажчик на неї
GL_GENBUFFERS glGenBuffers = (GL_GENBUFFERS)wglGetProcAddress("glGenBuffers");
// Тепер ми можемо нормально викликати цю функцію
GLuint buffer;
glGenBuffers(1, &buffer);
```

Необхідність отримувати адресу для кожної OpenGL функції робить цей процес просто болісним. Але, на щастя, існують бібліотеки, що реалізують це динамічне лінкування, а саме GLEW.

GLEW - бібліотека, яка спрощує процес завантаження та керування розширеннями OpenGL. Вона надає крос-платформний API для доступу до найновіших функцій та розширень OpenGL, що підтримуються графічним обладнанням.

Розширення OpenGL - це додаткові функції та можливості, які виходять за межі базової функціональності стандарту OpenGL. Ці розширення надають доступ до вдосконалених методів рендерингу, нової функціональності шейдерів та інших спеціалізованих можливостей. Він абстрагується від специфічних для платформи деталей і надає функції для запиту і завантаження розширень OpenGL під час виконання.

1.4.2 Мова програмування та середовище розробки

Так як необхідні бібліотеки підтримуються мову C++ до того ж вона є однією з найкращих мов програмування для розробки складних проектів, які потребують великих затрат потужностей, оскільки надає можливість гнучкого програмування та менеджменту пам’яті. До того ж усі ігрові рушії написані на цій мові. Тому вибір був доволі очевидний.

C++ - це мова програмування загального призначення, яка була розроблена як розширення мови програмування C. Розроблена Б'ярном Страуструпом (англ. Bjarne Stroustrup) у 1979 році. Вона відома своєю ефективністю, гнучкістю та потужними можливостями. C++ поєднує в собі процедурну парадигму програмування мови C з додаванням можливостей об'єктно-орієнтованого програмування (ООП).

Основним середовищем розробки будь-яких програм на C++ є інтегроване середовище розробки Visual Studio.

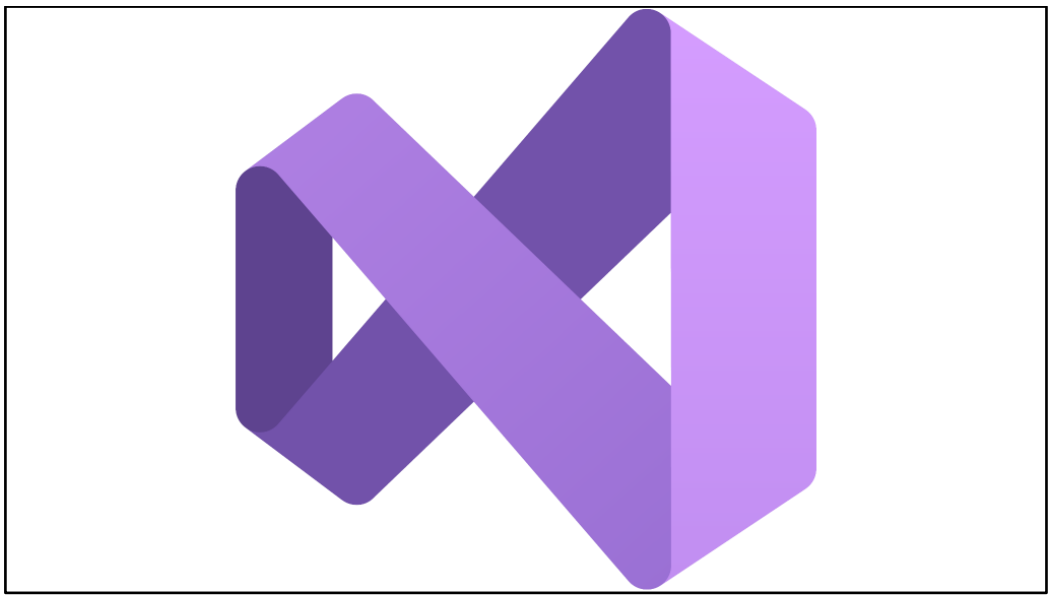

Рис. 1.10. Логотип IDE Visual Studio 2019

Інтегроване середовище розробки (IDE) - це програмне забезпечення, яке надає повний набір інструментів і функцій для полегшення розробки

програмного забезпечення. Воно слугує центральним центром для написання, редагування, компіляції, налагодження та керування кодом у межах єдиного інтерфейсу, що спрощує процес розробки.

Оскільки написання рушія потребує не тільки написання коду на C++, а також створення та редагування різних текстових файлів різного розширення з різними мовами. Для цього завдання чудово підходить швидкий та зручний текстовий редактор Visual Code, який має вбудований магазин з доповненнями, які допомагають в роботі з будь якими текстовими файлами. Для прикладу написання шейдерів на спеціальній мові GLSL.

GLSL - мова високого рівня для програмування шейдерів. Синтаксис мови базується на мові програмування ANSI C.

1.4.3 Інструмент генерації проекту

Використання інструментів генерації полягає у спрощенні процесу створення та управління складними програмними проектами. Ці інструменти автоматизують етапи конфігурації та компіляції, полегшуючи роботу з залежностями, специфічними для платформи відмінностями та організацією проекту.

Інструменти генерації, надають незалежний від платформи спосіб визначення процесу збірки програмного проекту. Замість того, щоб писати специфічні для платформи скрипти збірки або make-файли, розробники можуть використовувати мову конфігурації вищого рівня для визначення вимог проекту та цілей збірки. Це дає змогу гнучкого і автоматичного управління проектом, тим самим зберігаючи в системі контролю версії лише вихідні файли і комплювати проекти в залежності від машини розробника.

Вибір був зроблений на користь CMake, оскільки він має такі переваги над іншими інструментами генерації:

- Кросплатформенна підтримка: CMake абстрагується від платформних відмінностей і створює файли збірки, сумісні з різними платформами. Це спрощує процес збирання проектів на різних операційних системах, заощаджуючи час та зусилля.
- Керування залежностями: CMake надає механізми для роботи з залежностями, такі як пошук і підключення зовнішніх бібліотек. Це спрощує процес включення та керування сторонніми бібліотеками, забезпечуючи належну інтеграцію у проект.
- Модульна структура проекту: CMake підтримує організацію складних проектів у вигляді окремих модулів або бібліотек. Це дозволяє розробникам визначати та керувати залежностями між різними модулями, покращуючи модульність коду та можливість повторного використання.
- Гнучкість та масштабованість: CMake пропонує гнучку та масштабовану систему збірки. Вона може працювати з проектами різного розміру та складності, дозволяючи кастомізувати та адаптуватись до вимог проекту, що змінюються.

1.4.4 Допоміжні бібліотеки

Кожна бібліотека містить в собі необхідний функціонал для того, щоб не створювати велосипед кожного разу та не марнувати час на розробку того чи іншого функціоналу, якщо вже існують відповідні бібліотеки. Ось перелік та короткий опис бібліотек, що використані в даному ігровому рушію:

- boost - це широко використовувана і високо оцінена колекція бібліотек C++ з відкритим вихідним кодом. Вона надає повний набір бібліотек, які розширюють та покращують функціональність мови програмування C++. Boost має на меті доповнити стандартну бібліотеку C++, пропонуючи додаткові

можливості та абстракції. Деякі з реалізацій цієї бібліотеки були прийняти до різних стандартів C++.

- stb - проста бібліотека для завантаження графічних файлів (картинок) різного розширення та преображення їх в формат даних, які піддаються обробці для подальшого використання в OpenGL
- pugi - легка бібліотека, яка надає функціонал для простої роботи з xml файлами в C++
- RTTR - це потужна бібліотека C++, яка надає можливості самоаналізу та рефлексії типів під час виконання. Вона дозволяє перевіряти, запитувати та маніпулювати типами, класами, методами та властивостями під час виконання, навіть для класів, які не відомі під час компіляції.

РОЗДІЛ 2
АЛГОРИТМИ ТА МЕТОДИ СТВОРЕННЯ БАЗОВОГО ФУНКЦІОНАЛУ ІГРОВОГО РУШІЯ

Весь ігровий рушій починається з Game Loop (ігровий цикл) це звичайний цикл, який відбувається постійно і зупиняється якщо були виконані відповідні операції по типу виходу з гри або аварійне завершення.

Вся логіка відбувається саме в цьому циклі та всі наступні системи, які будуть працювати в реальному часу чи то графічний конвеєр чи то система управління ресурсами виконуються з цієї точки.

Він має приблизно ось такий вигляд:

```
while(ShouldClose())
{
    calculateDeltaTime();
    update();
    draw();
}
```

2.1 Рендеринг

Для успішної імплементації рендерингу за допомогою OpenGL потрібно використати відповідні послідовні кроки, що включають в собі певні алгоритми.

- Налаштування контексту рендерингу: Передбачає ініціалізацію OpenGL та створення контексту рендерингу, який керує станом рендерингу. Він включає створення вікна або поверхні для рендерингу та налаштування контексту OpenGL з потрібними параметрами.

- Підготовка даних вершини: Перед рендерингом необхідно підготувати дані вершин для об'єктів у сцені. Це

передбачає визначення геометрії об'єктів шляхом визначення вершин та їх атрибутів, таких як положення, нормалі, координати текстур та кольори. Дані про вершини зазвичай зберігаються у буферних об'єктах вершин (VBO) або масивах вершин.

- Програмування шейдерів:

В OpenGL дозволено лише перепрограмування двох типів шейдерів.

  - Вершинний шейдер трансформує вхідні вершини, застосовуючи такі перетворення, як трансляція, обертання та масштабування.
  - Фрагментний шейдер визначає колір кожного пікселя, враховуючи освітлення, текстурування та інші ефекти.

- Компіляція та компонування шейдерів:

Щоб створити виконувані об'єкти шейдерів, програми шейдерів потрібно скомпілювати і зв'язати з програмою OpenGL. Це включає компіляцію вихідного коду шейдерів, перевірку на наявність помилок, приєднання скомпільованих шейдерів до програми та зв'язування програми для створення остаточної програми шейдерів.

- Текстурування:

Текстурування додає об'єктам деталізації поверхні шляхом застосування зображень або візерунків. Координати текстури використовуються для накладання текстури на геометрію об'єкта. Текстури потрібно завантажити в пам'ять графічного процесора, прив'язати до текстурних блоків і застосувати у фрагментному шейдері під час рендерингу.

- Тестування глибини та відбраковування:

Для забезпечення коректного рендерингу об'єктів у 3D-сцені використовується тестування глибини, яке визначає, які фрагменти слід рендерити на основі їхніх

значень глибини. Відсікання задніх граней виконується для того, щоб відкинути грані об'єктів, які не є видимими.

- Матриці та перетворення: Матриці використовуються для перетворення геометрії та положення об'єктів у сцені. Найпоширеніші перетворення включають трансляцію, обертання, масштабування та проекціювання. Матриці застосовуються у вершинному шейдері для перетворення позицій вершин.

2.2 Камера та трансформації

OpenGL передбачає, що всі вершини, які ми хочемо побачити, після запуску шейдера будуть у нормалізованих координатах пристрою (NDC - normalized device coordinates). Це означає, що x, y і z координати кожної вершини мають бути між -1.0 і 1.0; координати за межами цього діапазону видно не буде. Зазвичай ми вказуємо координати в діапазоні, який налаштовуємо самостійно, а у вершинному шейдері перетворюємо ці координати в NDC. Потім, ці NDC передаються растеризатору для перетворення їх у двовимірні координати/пікселі вашого екрану.

Для перетворення координат з одного простору в інший ми використовуватимемо кілька матриць трансформації, серед яких найважливішими є матриці Моделі, Виду та Проекції. Координати наших вершин починаються в локальному просторі як локальні координати, і надалі перетворюються на світові координати, потім на координати вигляду, відсікання, і, нарешті, все закінчується екранними координатами. Наступне зображення (рисунок 2.1) показує цю послідовність, і те, що робить кожне перетворення:

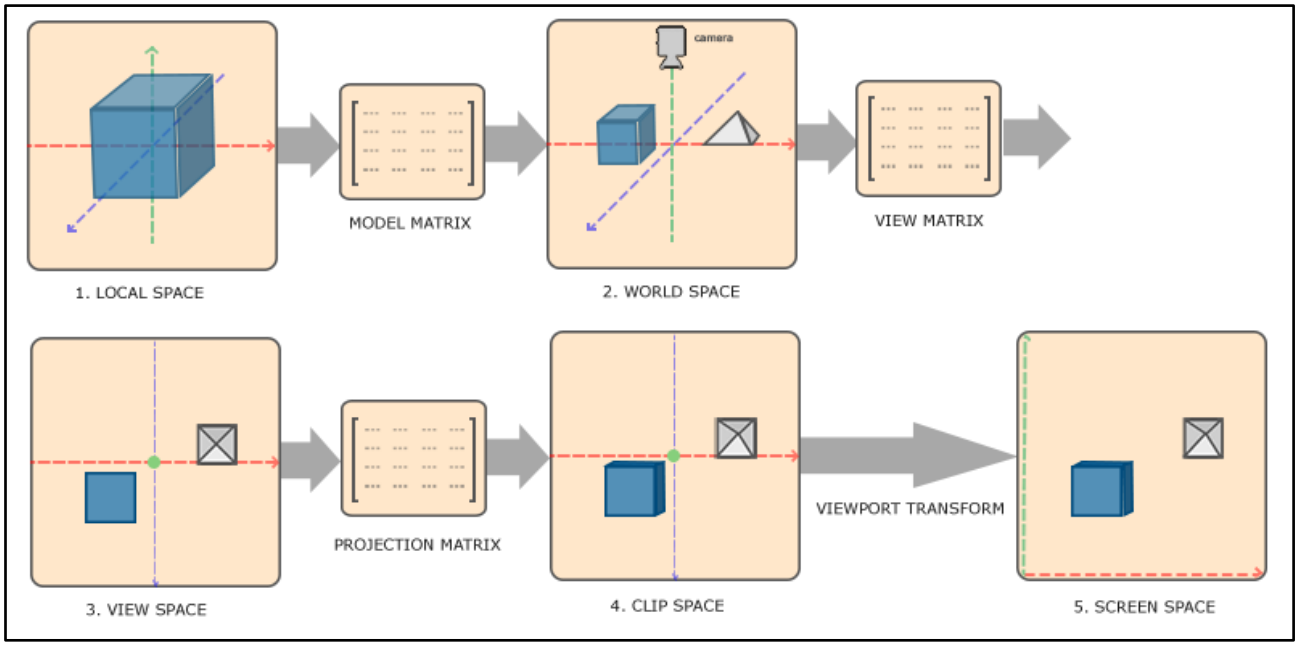

Рис. 2.1. Схема перетворення координат з локальних в екранні

1. Локальні координати - це координати вашого об'єкта, які вимірюються відносно точки відліку, розташованої там, де починається сам об'єкт.
2. На наступному кроці локальні координати перетворюються на координати світового простору, які за змістом є координатами більшого світу. Ці координати вимірюються відносно глобальної точки відліку, єдиної для всіх інших об'єктів, розташованих у світовому просторі
3. Далі ми трансформуємо світові координати в координати простору Виду таким чином, що кожну вершину стає видно так, наче на неї дивляться з камери або з точки зору спостерігача
4. Після того, як координати було перетворено в простір Виду, ми хочемо спроектувати їх у координати Відсікання. Координати Відсікання є дійсними в діапазоні від -1.0 до 1.0 і визначають, які вершини з'являються на екрані.
5. І, нарешті, у процесі перетворення, який ми назвемо трансформацією області перегляду, ми перетворимо координати відсікання від -1.0 до 1.0 в область екранних координат, задану функцією glViewport.

В OpenGL немає такого поняття як камера, оскільки при русі сам гравець не переміщується, а весь світ змінює свою трансформацію. Тому це визначення насправді відносне.

Нище описані кроки реалізації камери.

- Визначити параметри камери:

Положення камери, вектор направлення і вектор вгору (визначає орієнтацію камери). Ці параметри будуть використані для побудови матриці огляду.

- Обчислити матрицю огляду:

Матриця огляду являє собою перетворення, яке позиціонує та орієнтує камеру в сцені. Зазвичай вона створюється за допомогою параметрів камери, згаданих вище. Матрицю огляду можна побудувати за допомогою функцій або бібліотек, які обробляють матричні операції.

- Застосувати матрицю вигляду:

У циклі рендерингу, перед рендерингом будь-яких об'єктів, застосуємо матрицю вигляду до поточного стеку матриць OpenGL. Це можна зробити за допомогою функцій glLoadMatrix(), glUniformMatrix() або встановивши матрицю як однорідну у шейдері.

- Налаштувати матрицю проекцій:

Виберемо тип проекції, наприклад, перспективну або ортогональну, і визначимо параметри проекції, такі як поле зору, співвідношення сторін та ближні і дальні площини відсікання. Використовуємо ці параметри для створення матриці проекцій.

- Застосувати матрицю проекцій:

У циклі рендерингу, перед рендерингом будь-яких об'єктів, застосуємо матрицю проекцій до поточного стеку матриць OpenGL. Подібно до матриці вигляду, це

можна зробити за допомогою відповідних функцій або задати матрицю як однорідну в шейдері.

- Об'єднання матриць вигляду та проекцій: Створимо матрицю перетворення для кожного з вищезазначених кроків: моделі, вигляду і матриці проекції. Координата вершини перетворюється в координати простору відсікання таким чином: $V_{clip} = M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}$

Потрібно звернути увагу, що порядок множення матриць зворотний (множення матриць потрібно читати справа наліво). Отриману координату вершини потрібно присвоїти у вершинному шейдері вбудованій змінній gl_Position, після чого OpenGL автоматично виконає перспективний розподіл і відсікання.

2.3 Структура компонентів рушія

Основна структура рушія представляє собою ієрархію базових об’єктів з яких складається сцена (рівень гри). Основною одиницею на сцені повинен виступати SceneObject (об’єкт сцени), який має свою транформацію (позицію, масштаб та поворот) та містить в собі компонент, кожен з яких може містити в собі дитячі компоненти.

Кожен компонент містить в собі дві базові функціональні структури:

- Drawable - відповідає за візуальне представлення об’єкта та відмальовку його на сцені. Може бути лише одним в кожному компоненті.
- Behaviour - є поведінковою структурою, де міститься всі необхідні методи для створення логіки компоненту/об’єкта на сцені. В одному компоненті можуть міститись декілька поведінкових структур.

2.4 Система генерації проекту та структура файлів

Основний або ж початковий файл генерації буде знаходитись в корінній папці. Послідовність налаштувань, які повинні бути реалізовані мають виглядати приблизно ось так:

Створюємо проект з назвою рушія, цю назву проекту, як змінну буде використовувати система генерації.

Виставляється точка запуску проекту (проект, який буде компілюватись IDE Visual Studio), а саме щойно створений проект.

Налаштовуємо компіляційні флажки для того, щоб кожне попередження реєструвалось системою як помилка, оскільки проект повинен бути без будь-яких помарок.

Підключаємо кожну залежну бібліотеку, в кожній папці якої повинний лежати файл з вказівками, як правильно її реєструвати. Кожна така бібліотека повинна знаходитись в окремо виділеній папці з відповідною назвою (до прикладу dependecies - залежності), звідки система генерації буде видирати потрібні їй файли.

Запускаємо цикл, який буде збирати файл контенту (шейдери, текстури і т.п.) та вихідні файли (розширення .h та .cpp), додаючи їх до виконуваного проекту.

Реєструємо наш проект, як статичну бібліотеку, оскільки її повинні підключати ігри, які будуть використовувати рушій.

Визначаємо стандарт C++, який буде використовуватись.

І на кінець вимикаємо консоль при запуску.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ

Реалізація програмного забезпечення - це процес перетворення програмного дизайну у виконуваний код. Він включає в себе написання та налаштування коду, інтеграцію необхідних бібліотек і фреймворків, а також тестування програмного забезпечення, щоб переконатися, що воно функціонує так, як задумано. Програмна реалізація є найважливішим етапом у розробці програмного забезпечення, на якому дизайн і функціональність програмної системи втілюються в життя.

Для успішної реалізації рушія потрібно перед усім створити набір основних систем, які будуть формувати загальну структуру рушія та його функціональність. Даний розділ детально описує архітектуру розробленого рушія та його використання.

3.1 Архітектура рушія

Архітектуру ігрового рушія можна розглядати як складну мережу взаємопов'язаних компонентів, кожен з яких відповідає за певний аспект створення та виконання гри. Ці компоненти включають системи рендерингу, фізичне моделювання, управління звуком, обробку вводу, управління ресурсами, організацію сцен і багато іншого. Добре продумана архітектура забезпечує безперешкодну інтеграцію цих компонентів, що дозволяє розробникам ігор зосередитися на створенні захопливого ігрового процесу та контенту. Розуміння базової архітектури має вирішальне значення для використання повного потенціалу ігрового рушія та створення захоплюючих інтерактивних світів.

Розроблений ігровий рушій містить в собі невід’ємні системи та підсистеми, кожна з яких має свою ціль та функціонал, що забезпечує загальну

роботоспроможність рушія. Загальна структура рушія представлена в візуальному оформленні на рисунку 3.1:

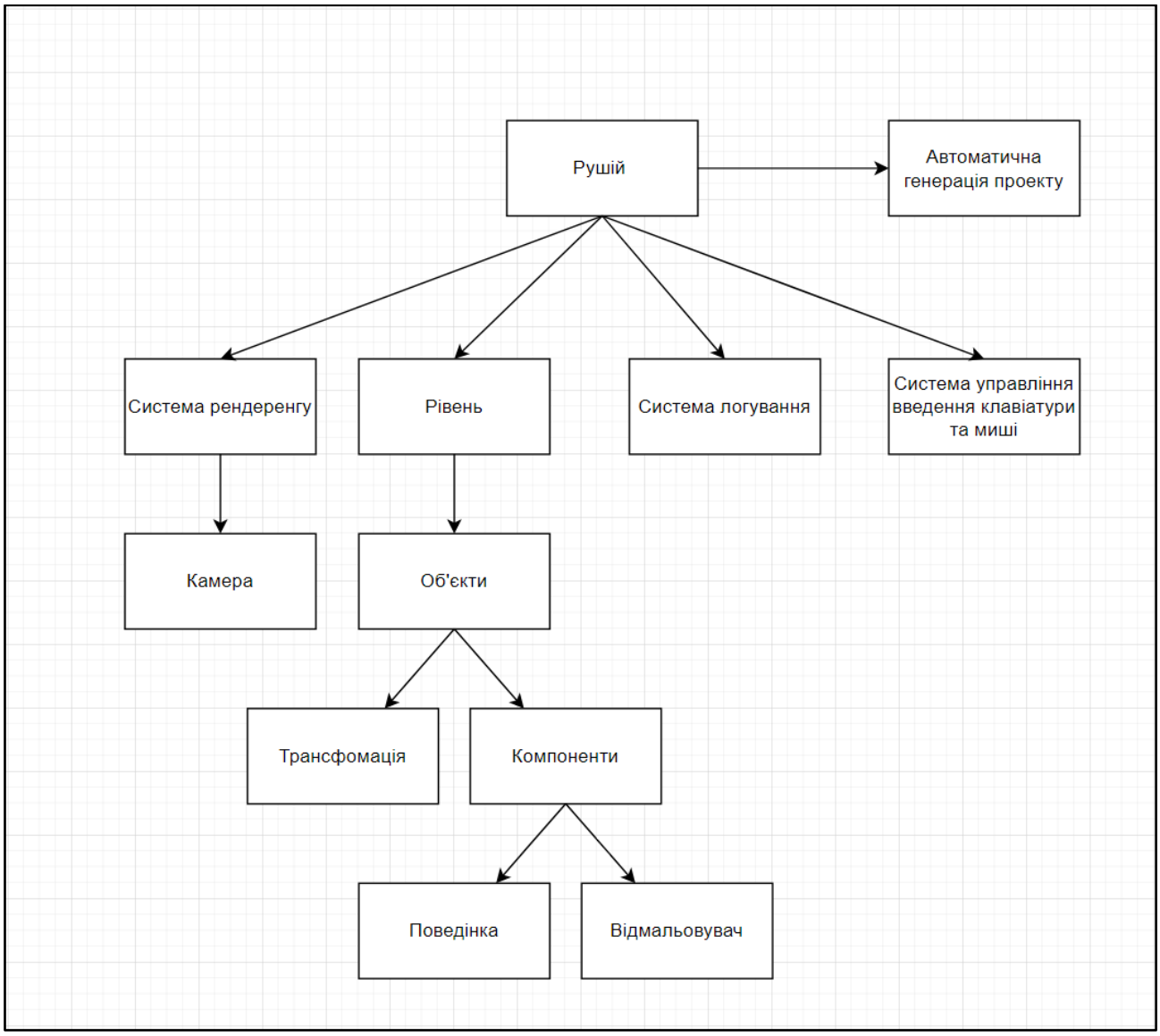

Рис. 3.1. Загальна структура рушія

3.1.2 Діаграма класів

Діаграма класів описує реалізацію розробленого рушія, за допомогобю uml-діаграми з використанням парадигми об'єктно орієнтованого програмування. Діаграма базових класів наведена на рисунку 3.2 та 3.3.

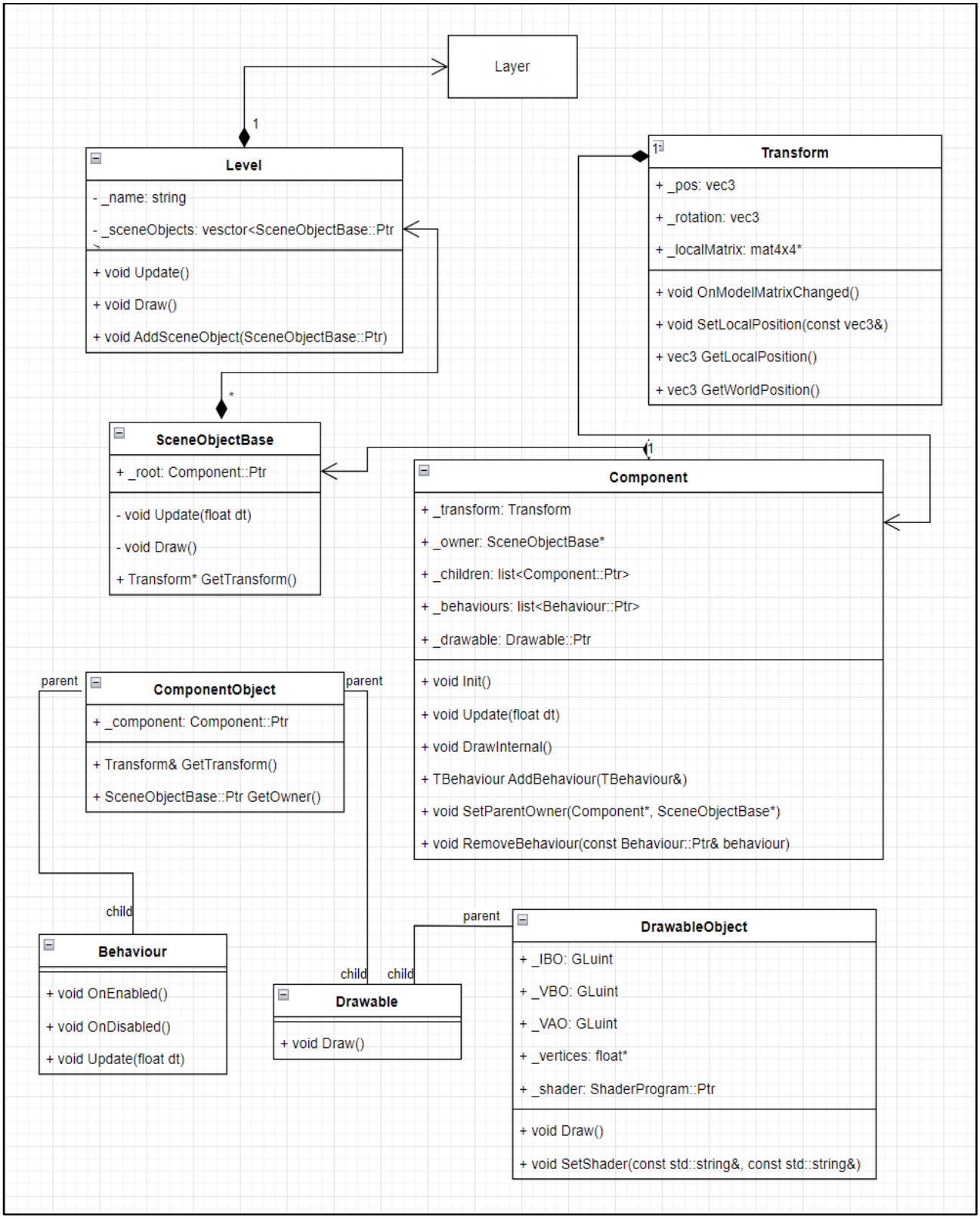

Рис. 3.2. Діаграма базових класів частина 1

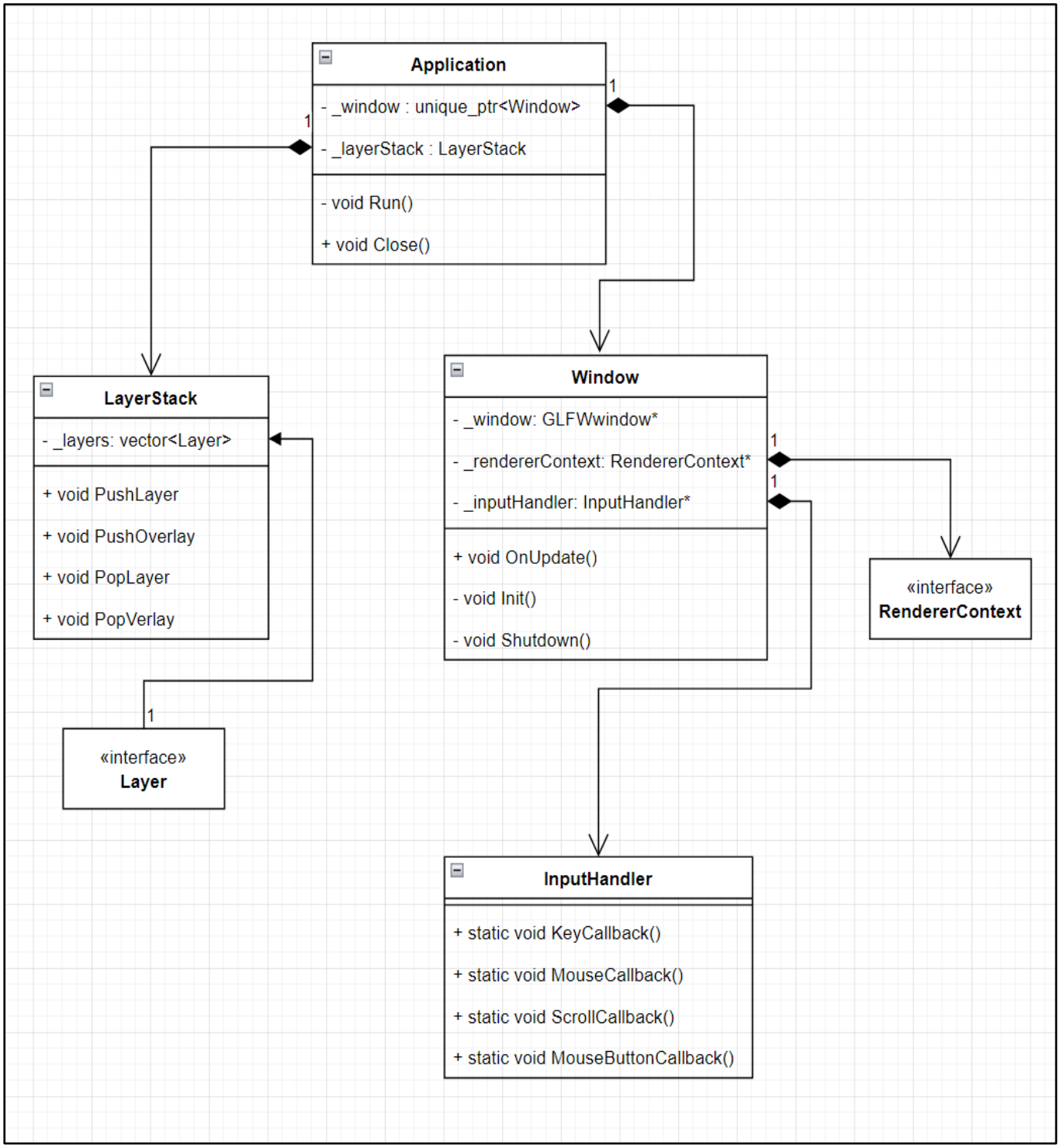

Рис. 3.3. Діаграма базових класів частина 2

Короткий опис наявних класів та їх функціональність:

- Application - основний клас, що уособлює в собі гру. Це точка відліку всієї програми.

- Window - вікно гри, що містить відповідні налаштування (розмір, чи ввімкнена вертикальна синхронізація тощо). Напряму підпорядковується Application.
- Log - містить статичні методи для використання системи логування повідомлень
- Camera - клас камери, що містить відповідний функціонал та налаштування.
- Layers - базовий клас, що відповідає за окреме управління рішуєм, до прикладу відмалювання допоміжної сітки в редакторі, в окремому шарі редактора, оскільки це не є частиною гри.
- Layers stack - колекція, що відповідає за те в якій послідовності повинні виконуватись виклики шарів з базового класу додатку.
- InputHandler - менеджер управлінням введенням користувача.
- RefCounter - менеджер підрахунку сильних та слабких посилань для правильного видалення даних з кучі; необхідний для використання intrusive_ptr з бібліотеки boost.
- Texture - описує текстуру з її даними та містить необхідний функціонал для її використання.
- ShaderProgram - описує шейдер, містить дані про нього та функціонал для його використання.
- RendererContext - абстрактний клас, що описує конктект, який буде використовуватись. В даному рушії представлений лише OpenGLRendererContext.
- FrameBuffer - абстрактний клас, що описує буфер кадру. В даному рушії представлений лише OpenGLFrameBuffer.
- Transform - клас, що описує трансформацію сущності та має простий функціонал для управління.
- Level - рівень гри/редактора.
- SceneObjectBase - базова одиниця гри, що містить в собі компонент та трансформацію.

- Component - корінний клас SceneObjectBase, містить об’єкти поведінки та відмальовки. Може містити в собі дочірні компоненти.
- ComponentObject - базовий клас об’єкта компонента.
- Drawable - об’єкт компонента, який відповідає за відмальовку.
- Behaviour - об’єкт компонента, який відповідає за логіку/поведінку.
- Canvas - об’єкт сцени, що відповідає за інтерфейс.
- ResourceLoader - менеджер ресурсів.
- CubeDrawable - клас, що малює куб з відповідними параметрами, та текстурою.
- Timer - звичайний клас таймеру з використанням функціоналу бібліотеки chrono.
- Singleton - шаблонний клас, що реалізує патерн.

3.2 Камера

Коли ми говоримо про простір камери/виду, ми маємо на увазі вид усіх вершин із погляду камери, положення якої в цьому просторі є базовою точкою початку координат: матриця виду трансформує світові координати в координати виду, що вимірюються відносно розташування та напрямку камери. Для однозначного математичного опису камери нам необхідне її положення у світовому просторі, напрямок, у якому вона дивиться, вектор, що вказує вправо, і вектор, що вказує вгору.

За допомогою бібліотеки glm, ми можемо створити дві різні типи камери: перспективна та ортогональна за допомогою вхідних даних, а саме:

- fov - який ми перетворюємо в радіани для правильних обчислень
- якщо у нас перспективна камера, то ми визначаємо Ratio за допомогою ділення ширини вікна на висоту.
- ближча межа
- дальня межа

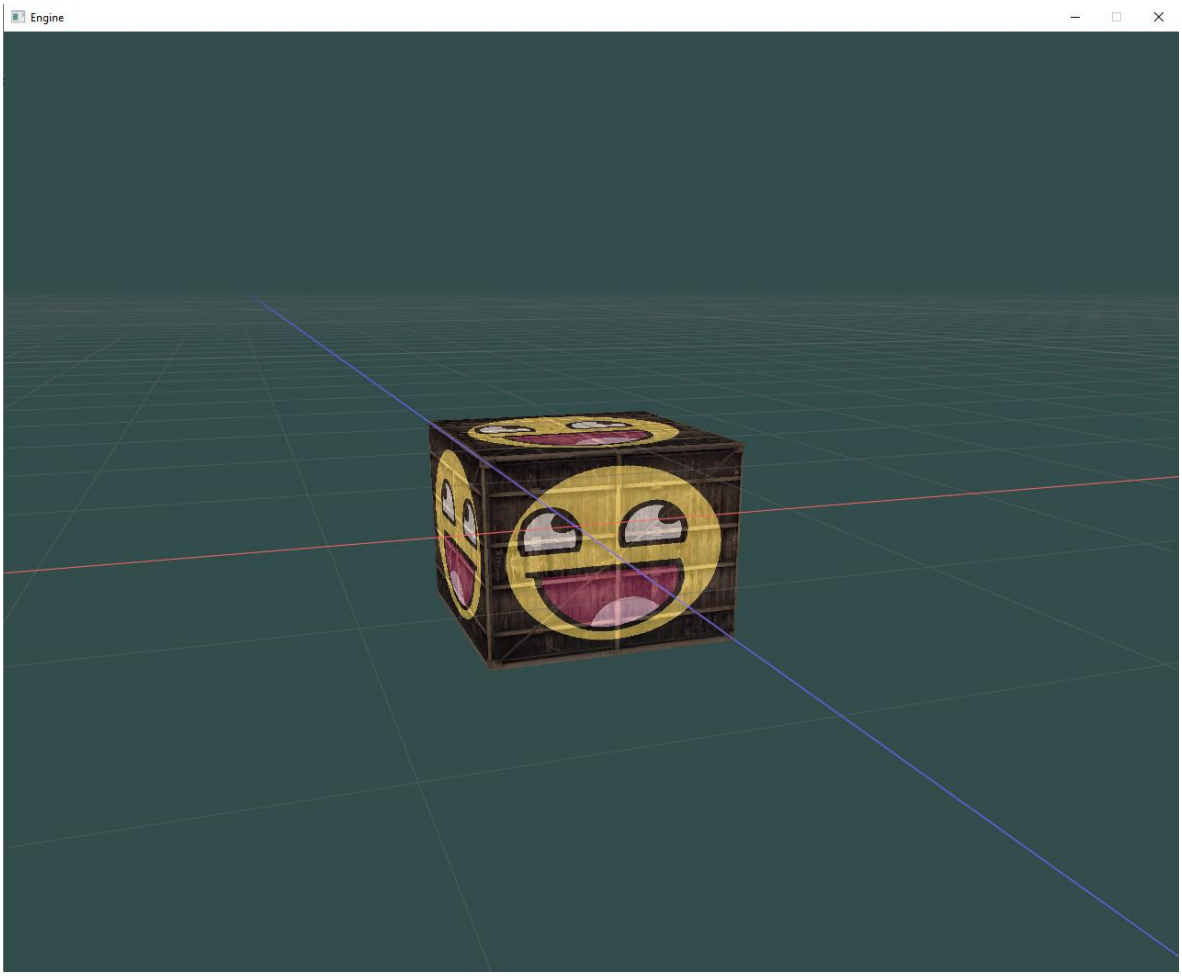

Рис. 3.4. Візуалізація перспективної камери

Матриці мають властивість, завдяки якій можна визначити координатний простір з трьома перпендикулярними (або лінійно незалежними) осями. Побудувавши матрицю з використанням векторів цих осей і додаткового вектора зсуву, будь-які задані вектори можуть бути перетворені в заданий координатний простір за допомогою множення. Ця точна функціональність є саме тим, що досягається за допомогою матриці LookAt.

Розрахунок матриці LookAt виглядає приблизно ось так як зображено на рисунку 3.5.

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Рис. 3.5. Обчислення матриці LookAt

Де R - правий вектор, U - вектор, що вказує вгору, D - вектор напрямку камери, а P - позиція камери. Зверніть увагу на те, що вектор положення камери інвертований, оскільки в кінцевому підсумку ми змінюватимемо світові координати в напрямку, протилежному руху камери. Використання матриці LookAt як матриці вигляду дає змогу ефективно перетворити всі світові координати у щойно заданий нами простір. Матриця LookAt робить саме те, про що говорить її назва: вона створює матрицю вигляду, яка дивиться на задану ціль.

3.3 Управління в грі та редакторі

Введення в редакторі - це спосіб, у який користувач взаємодіє з інтерфейсом редактора для виконання дій і надання команд або даних. Він включає в себе різні методи введення, такі як введення з клавіатури чи миші, які дозволяють користуву маніпулювати і контролювати функціональність редактора. Введення в редакторі відіграє життєво важливу роль у полегшенні взаємодії з користувачем, дозволяючи йому ефективно створювати, змінювати та керувати контентом у середовищі редагування.

Основним менеджером який відповідає за введення є InputHandler. Він сповіщає систему про те, що відбулась якась певна дія і розробник може створити реакцію на ту чи іншу дію, а також визначити унікальні умови цієї дії, для прикладу: клавіша була натиснута чи відпущена, активація якщо клавіша зажата певний період часу, тощо.

В редакторі представлено основне управління, яке неможливо змінити. Ось перелік можливостей, які керуються за допомогою клавіатури чи миші:

- Для того щоб змістити фокус ОС Windows, потрібно виділити вікно і натиснути клавішу “E”, для повернення фокус потрібно ще раз натиснути.
- Прокрутка середньої клавіша миші або ж колеса відповідає за швидке пересування камери редактору вперед чи назад вздовж осі напряму камери.
- Затиснути середню клавішу миші та рухаючи мишу дає змогу рухати камеру редактора вздовж осі XY.
- WASD - основні клавіші, що змінюються положення камери вперед/вліво/назад/вправо.
- Якщо при русі за допомогою WASD затиснути клавішу “Shift” відбудеться прискорення зміни положення.
- Рухаючи мишу ми можемо змінювати нахил камери редактора. Так як миша може рухатись по двум осям, то і камера змінює лише Yaw та Pitch, а також обмежує зміну повороту, якщо значення переходить за 90 градусів.

3.4 Інтерфейс редактора

Інтерфейс користувача в редакторі - це візуальні елементи та інтерактивні компоненти, які дозволяють користувачам взаємодіяти з програмним забезпеченням для редагування. Він охоплює меню, кнопки, панелі, панелі інструментів та інші графічні елементи, які дозволяють користувачам виконувати дії, переміщатися по функціях і маніпулювати вмістом у редакторі. Інтерфейс редактора має на меті забезпечити інтуїтивно зрозумілий і зручний інтерфейс, який сприяє ефективному робочому процесу та покращує загальний досвід редагування.

За допомогою бібліотеки ImGui створено зручний інтерфейс для управління об’єктами на сцені: видалення, додавання, зміна параметрів сцен об’єкта. За це відповідає шар редактору, який створює відповідні вікна.

- App - головне меню для взаємодії з редактором та сценою. (див. рисунок 3.6)

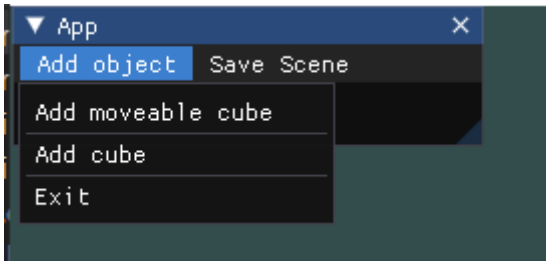

Рис. 3.6. Головне меню редактора

- SceneObjects - містить список всіх наявних об’єктів на сцені. При натисканні на об’єкт з’являється вікно налаштувань
- Properties - вікно налаштувань вибраного об’єкта на сцені. На разі представляє можливість зміни лише позиції та повороту.

Рис. 3.7. Інтерфейс зміни параметрів об’єктів на сцені

3.5 Автоматична генерація проекту та налаштування залежностей

Кожна бібліотека була розміщена в окремому каталозі - dependecies і має свій CmakeLists.txt, що містить правила збірки кожної бібліотеки відповідно. У кожної бібліотеки можуть міститись унікальні правила, але загальний інтерфейс підключення виглядає ось так:

- Підключення файлів оголошення (.h) та вихідних файлів (.cpp).
- Створення статичної бібліотеки з відповідною назвою.
- Підключення бібліотеки до проекту рушія.

3.6 Об’єкти сцени та компоненти

SceneObject - базова одиниця на сцені. Реалізує два основні методи: Draw та Update. Кожний такий об’єкт має клас Component, що виступає корінною структурою. При створення SceneObject ми змушуємо корінну структуру присвоїти собі цей об’єкт як головного власника та батька, оскільки Component може бути дочірнім для іншого, але в даному випадку він є корінним, тому така умова необхідна.

Draw - запускає ланцюжок відмальовки компонента, який в свою чергу змушує відмальовуватись об’єкти компонента (Drawable).

Update - також запускає ланцюжок і сигналізує про повне проходження ігрового циклу і зміну кадру, за яким повинно відбутися зміна логіки гри. Параметром цього методу є dt (delta time) - різниця між поточним та попереднім кадром для використання в різних обчисленнях фізичної поведінки тощо.

За відмальовку як уже було сказано відповідає клас Drawable, що містить в собі Vertex Arrays Object, Vertex Buffer Object, Index Buffer Object, масив вершин, та шейдер і відповідні методи для ініціалізації цих даних.

В рушії уже є декілька реалізованих відмальовувачів. До прикладу:

- CubeDrawable - малює куб, що містить в собі дві текстури, що за допомогою можливості OpenGL змішуються за допомогою відповідного шейдера, де визначається наскільки сильно інша текстура витісняє іншу.
- LineDrawable - відповідає за відмалювання лінії в просторі з вказаною початковою та кінцевою точкою, а також кольором.

Для того щоб створити свій відмальовувач: достатньо створити клас та наслідуватись від класу Drawable, що знаходить в просторі імен Visual. Після цього потрібно визначити метод Draw та зробити відповідну ініціалізацію даних (текстури, шейдери тощо).

Behaviour - поведінковий клас, що реалізує метод Update, який викликається з власника цього об’єкту.

Так само як і для Drawable, щоб створити свою поведінку достатньо створити клас, який наслідується від Behaviour та реалізувати метод Update. До прикладу можна розглянути клас MoveBehaviour, що рухає його власника по осі Y, використовуючи тип руху PingPong - тобто з плином часу він повертається на попереднє місце.

3.7 Використання рушія для розробки гри

Розробка рушія це лише одне, а його правильна інтеграція з проектом гри - зовсім іншого рівня завдання. Перш за все було поставлено за мету зробити процес інтеграції максимально зручним для користувача.

Для того щоб почати розробляти гру за допомогою рушія, достатньо перенести репозиторій з рушієм в кореневу папку гри. Далі створюємо файл Cmake, де додаємо наш рушій як статичну бібліотеку, а після цього потрібно створити скрипт, який буде генерувати солюшен. Після всіх цих взаємодій у нас готовий проект з усіма підключеними залежностями.

Після цього нам потрібно створити сам об’єкт гри, який буде наслідуватись від базового класу - Application, а також додати визначення функції CreateApplication (вона позначена як extern, що дає змогу визначити її в іншому проекті) всередині якої ми створюємо новий додаток (new “назва класу гри”). Щоб це зробити достаньо підключити файл EntryPoint.h.

Оскільки в C++ точкою відліку є метод main(), то потрібно ізолювати цю функцію для користувача, щоб рушій сам правильно відповідав за старт виконання.

Щоб це було можливо в класі Application функцію main(), де створюється сам Application, було зроблено як приватно дружня, що дає змогу у ній викликати приватний метод нашої гри, а саме метод Run, який відповідає за старт застосунку та запуск ігрового циклу.

ВИСНОВКИ

Метою даної дипломної роботи було створення зручного в користуванні, швидкого в вивченні та простого ігрового рушія для розробки ігор, а також дослідити методи реалізації архітектури рушіїв.

При реалізації та виконанні основних завдань можна констатувати результати:

- Реалізовано автоматичну генерацію проекту за допомогою Cmake, що значно спрощує роботу над майбутніми проектами та модифікацією версій рушія, без додаткових втручань.
- Створено систему рендеренгу, на основі графічного API OpenGL.
- Додане зручне управління редактором, що спрощує роботу над рівнями.
- Реалізовані базові класи, які мають простий функціонал, що можна змінювати для забезпечення своїх цілей.

На закінчення, розробка ігрового рушія для мого дипломного проекту була складною, але важливою справою. Завдяки обширним дослідженням, реалізації та тестуванню, я успішно створив функціональний ігровий рушій, який забезпечує міцну основу для розробки ігор. Рушій включає в себе ключові компоненти, такі як рендеринг, системи введення, генерації проекту та інші, що дозволяє створювати захоплюючі та інтерактивні ігри. Хоча завжди є місце для вдосконалення та подальшого доопрацювання, цей проект надав цінне розуміння складності архітектури ігрового рушія та тонкощів побудови надійної програмної системи. Загалом, цей дипломний проект поглибив моє розуміння принципів розробки ігор і забезпечив мене цінними навичками, які будуть безцінними в моїх майбутніх починаннях у цій галузі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Fast and effective occlusion culling for 3D holographic displays: веб-сайт. URL:https://www.researchgate.net/figure/Projection-modes-a-orthographic-projection-and-b-perspective-projection_fig2_265693452
2. OpenGL Projection Matrix: веб-сайт. URL: http://www.songho.ca/opengl/gl_projectionmatrix.html
3. 3D Graphics with OpenGL Basic Theory: веб-сайт. URL: https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
4. Засіб створення UML діаграм: веб-сайт. URL: https://app.diagrams.net/
5. Cmake документація: веб-сайт. URL: https://cmake.org/cmake/help/latest/guide/tutorial/index.html
6. Документація C++: веб-сайт. URL: C++ reference - cppreference.com
7. UnrealEngine: веб-сайт. URL: The most powerful real-time 3D creation tool - Unreal Engine
8. http://ce.eng.usc.ac.ir/files/1511334027376.pdf
9. Документація по boost: веб-сайт. URL: https://www.boost.org/
10. GLFW документація: веб-сайт. URL: https://www.glfw.org/docs/3.3/input_guide.html
11. Бібліотека SPDLOG: веб-сайт: URL: https://github.com/gabime/spdlog/wiki/1.-QuickStart
12. Unity: веб-сайт: URL: https://unity.com/
13. OpenGL документація: веб-сайт: URL: https://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/Chapter_3.pdf
14. Khronos OpenGL® and OpenGL® ES: веб-сайт: URL: https://registry.khronos.org/OpenGL-Refpages/
15. Glew: веб-сайт: URL: https://glew.sourceforge.net/
16. GLM: веб-сайт: URL: http://glm.g-truc.net/0.9.5/index.html

17. Intro to the trigonometric ratios : веб-сайт: URL:https://www.khanacademy.org/math/geometry-home/right-triangles-topic/intro-to-the-trig-ratios-geo/v/basic-trigonometry
18. Vertex Buffer Objects: веб-сайт: URL: https://antongerdelan.net/opengl/vertexbuffers.html
19. DirectX: веб-сайт: URL: https://support.microsoft.com/uk-ua/topic/%D0%B7%D0%B0%D0%B2%D0%B0%D0%BD%D1%82%D0%B0%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D0%B0-directx-%D1%96-%D0%B9%D0%BE%D0%B3%D0%BE-%D1%96%D0%BD%D1%81%D1%82%D0%B0%D0%BB%D1%8F%D1%86%D1%96%D1%8F-d1f5ffa5-dae2-246c-91b1-ee1e973ed8c2
20. API Vulkan: веб-сайт: URL: https://www.vulkan.org/
21. Документація imgui: веб-сайт: URL: https://github.com/ocornut/imgui/wiki#general-documentation
22. CryEngine: веб-сайт: URL: https://www.cryengine.com/
23. Introduction to Computer Graphics: веб-сайт: URL: https://www.geeksforgeeks.org/introduction-to-computer-graphics/
24. Документація з створення діаграм: веб-сайт: URL: https://venngage.com/blog/class-diagram/
25. Use _crtBreakAlloc to debug a memory allocation: веб-сайт: URL: https://learn.microsoft.com/en-us/troubleshoot/developer/visualstudio/cpp/libraries/use-rtbreakalloc-debug-memory-allocation
26. Типи бібліотек: веб-сайт: URL: https://domiyanyue.medium.com/c-development-tutorial-4-static-and-dynamic-libraries-7b537656163e

ДОДАТОК А

Додаток наводить вихідний код основних класів та об’єктів ігрового рушія написаного на C++. Даний код демонструє структуру та дає можливість краще ознайомитись з архітектурою рушія.

Реалізація класу камери:

```
void Camera::UpdateProjection()
        {
            const       Engine::Application*       app       =
Engine::Application::Get();
            uint32_t Height = GlobalSettings::HEIGHT;
            uint32_t Width = GlobalSettings::WIDTH;

            // TODO zNearPlane and zFarPlane move outside from
GlobalSettings
            _projection = glm::perspective(glm::radians(_fov),

static_cast<float>(Width / Height),

GlobalSettings::zNearPlane, GlobalSettings::zFarPlane);
        }

        void Camera::UpdateCameraVectors()
        {
            // Calculate the new Front vector
            glm::vec3 front;
            front.x      =      cos(glm::radians(_yaw))      *
cos(glm::radians(_pitch));
            front.y = sin(glm::radians(_pitch));
            front.z      =      sin(glm::radians(_yaw))      *
cos(glm::radians(_pitch));

            _front = glm::normalize(front);
```

```
            // Also re-calculate the Right and Up vector
            // Normalize the vectors, because their length gets
closer to 0 the more you look up or down which results in slower
movement.
            _right     =     glm::normalize(glm::cross(_front,
_worldUp));

            _up = glm::normalize(glm::cross(_right, _front));
        }
void Camera::SetPitch(GLfloat pitch)
        {
            if (pitch < -89.0f)
            {
                _pitch = -89.0f;
            }
            else if (pitch > 89.0f)
            {
                _pitch = 89.0f;
            }
            else
            {
                _pitch = pitch;
            }
            UpdateCameraVectors();
        }

        void Camera::SetFov(GLfloat fov)
        {
            if (fov >= 1.0f && fov <= 120.0f)
            {
                _fov = std::clamp(fov, 1.0f, 120.0f);
                Camera::Instance().UpdateProjection();
            }
        }
```

Реалізація сітки в редакторі за допомогою шейдерів. Вертексний шейдер:

```
#version 410 core

layout (location = 0) in vec3 position;
out float near; //0.01
out float far; //100
out vec3 nearPoint;
out vec3 farPoint;
out mat4 fragView;
out mat4 fragProj;
uniform mat4 projection;
uniform mat4 view;
uniform float zNear;
uniform float zFar;

vec3 UnprojectPoint(float x, float y, float z, mat4 view, mat4
projection)
{
    mat4 viewInv = inverse(view);
    mat4 projInv = inverse(projection);
    vec4 unprojectedPoint =  viewInv * projInv * vec4(x, y, z,
1.0);
    return unprojectedPoint.xyz / unprojectedPoint.w;
}

void main()
{
    vec3 p = position;

    nearPoint   =   UnprojectPoint(p.x,   p.y,   0.0,   view,
projection).xyz;
    farPoint    =   UnprojectPoint(p.x,   p.y,   1.0,   view,
projection).xyz;

    near = zNear;
    far = zFar;

    fragView = view;
```

```
    fragProj = projection;

    gl_Position = vec4(p, 1.0);
}
```

Реалізація game loop:

```
void Application::Run()
    {
       while (_isRunning && !glfwWindowShouldClose(_window->GetNativeWindow()))
        {
            const float time = static_cast<float>(glfwGetTime());
            _deltaTime = time - _lastFrame;
            _lastFrame = time;

            glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

            for (Layer* layer : _layerStack)
            {
                layer->OnUpdate(_deltaTime);
            }

            _imGuiLayer->Begin();
            {
                for (Layer* layer : _layerStack)
                {
                    layer->OnImGuiRender();
                }
            }
            _imGuiLayer->End();

            _window->OnUpdate();
```