КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

Кваліфікаційна робота

на здобуття освітнього рівня бакалавра

за спеціальністю 121 Інженерія програмного забезпечення

на тему:

РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ СТВОРЕННЯ ІГРОВИХ СИСТЕМ

Виконав студент 4-го курсу
Максим БОНДАР ___________
(підпис)

Науковий керівник:
доцент, кандидат фіз-мат. наук
Євген ІВАНОВ ___________
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _________
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри інтелектуальних програмних систем
«____»________________2021р.,
протокол №
Завідувач кафедри
Олександр ПРОВОТАР _________
(підпис)

Київ - 2021

РЕФЕРАТ

Обсяг роботи 46 сторінок, 8 ілюстрацій, 9 таблиць, 11 посилань на джерела.

Метою роботи є створення нового програмного продукту для зручного створення ігрових систем. Отримання знань в сфері розробки ігрових рушіїв, рендеру.

Методи розроблення: мова програмування – C++ [1] [2], IDE – QtCreator [3]. Система написана за домогою Vulkan API [4] [5]. Графічне відображення системи створено за допомогою багатоплатформового API GLFM [6]. Мова написання скриптів – Python.

Результати роботи: в ході проведеної роботи вдалося реалізувати продукт, здатний виконувати скрипти на мові Python, здійснювати інтеграцію інтерпретатора мови Python в код C ++ програми, виконувати рендеринг тривимірних моделей за допомогою графічного API Vulkan. У процесі роботи були отримані знання графічного конвеєра, такі як растеризація, теселяція, створення фреймбуфферів, пулу дескрипторів, командного пулу, створення синхронізації процесора та графічного процесора, вибір найкращого пристрою GPU серед імен у системі. Для даної задачі був розроблений формат тривимірних моделей, для спрощення і стандартизації даних для завантаження в програму.

ЗМІСТ

РЕФЕРАТ....2
Обсяг роботи 46 сторінок, 8 ілюстрацій, 9 таблиць, 11 посилань на джерела. 2
Метою роботи є створення нового програмного продукту для зручного створення ігрових систем. Отримання знань в сфері розробки ігрових рушіїв, рендеру....2
ЗМІСТ....3
ВСТУП 6
1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ....9
1.1 Актуальність та мета роботи....9
1.2 Опис API Vulkan 9
1.3 Ієрархія основних об'єктів API Vulkan 10
1.4 Домовленість про типи об'єктів та функцій 12
1.5 Управління пам'яттю 13
1.6 Багатопотоковість в Vulkan 14
1.7 Графічний конвеєр Vulkan....14
1.8 Скриптування за допомогою Python API....17
2. ТЕХНІЧНЕ ЗАВДАННЯ 18
2.1 Опис області використання та початкових даних додатка 18
2.2 Вимоги до апаратних, програмних і комунікаційних інтерфейсів 18
2.3 Вимоги до файлів формату *.kem....19
3. ОБГРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ ТА ПРОГРАМНИХ ЗАСОБІВ....21
3.1 Мова розробки 21
3.2 Бібліотека GLFW....21
3.3 Графічний API Vulkan 22
3.4 Інтерпретована скриптова мова 23
3.5 Інструменти для розробки....23
4. РОЗРОБКА БАГАТОПЛАТФОРМНОГО ІГРОВОГО РУШІЯ З ІНТЕРПРЕТАТОРОМ СКРИПТОВИХ КОМАНД 24
4.1 Структура проекту 24
4.2 Опис класів 24
5. Приклад створення гри на базі розробленого продукту....41
Висновок....42
Посилання на використані джерела....43

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

IDE – Integrated Design Environment, інтегроване середовище розробки;

API – Application Programming Interface, інтерфейс програмних застосунків;

CPU – Central Processing Unit, центральний процесор;

GPU – Graphics Processing Unit, графічний процесор;

Меш – polygon mesh, полігональна сітка;

Спрайт – Sprite, графічний об'єкт в комп'ютерній графіці;

SDK - Software Development Kit, набір із засобів розробки, утиліт та документації;

Vulkan - багатоплатформний прикладний програмний інтерфейс для тривимірної графіки та супровідних обчислень;

OpenGL - **Open G**raphics **L**ibrary, специфікація, що визначає незалежний від мови програмування багатоплатформний API;

DSP – Digital Signal Processor, цифровий процесор обробки сигналів;

Інді-геймдев - напрям геймдеву, в якому створюються ігрові програми невеличкими командами без фінансової підтримки видавництва;

Сі – мається на увазі мова програмування Деніса Рітчі - C

ВСТУП

Ще з часів ігрових консолей п'ятого покоління розробники ігор прагнули при створенні ігор не переписувати той же самий код безліч разів. Приблизно в той час зародилося поняття «ігровий рушій», який позначає собою фреймворк, спрямований на створення інтерактивних графічних додатків, в яких присутній рендеринг двомірних спрайтів і тривимірних мешів, звук, система для користувача введення, опціонально фізика, штучний інтелект, мережевий мультиплеєр.

З розвитком обчислювальних потужностей персональних пристроїв збільшувалися вимоги до можливостей і пропрацьованності відеоігор. Все частіше замість написання свого рушія розробники ліцензували чужий, особливо популярними стали в свій час рушії Quake Engine і Unreal Engine [7], які і понині популярні для використання у сторонніх студій.

До недавнього часу всі подібні рушії були пропрієтарними, з фіксованою вартістю ліцензії, але завдяки переходу Unity і Unreal Engine на безкоштовну модель (з певними обмеженнями та умовами), широкі маси розробників ігрових систем отримали доступ до професійних ігрових рушіїв.

Навіть в таких умовах доступності, все ж може виникнути необхідність написати свій рушій, що задовольняє особисті потреби і завдання.

Найбільш актуальними прикладами сучасних ігрових рушіїв є Unity і Unreal Engine 4 [7] (на рис. 1). Дані продукти мають недолік в тому, що величезна кількість абстракцій коду впливає на швидкодію, а також їх комерційне використання вимагає відрахування певної виплати розробникам, до того ж Unity не надає першокод, що вносить певні обмеження в спектр можливостей.

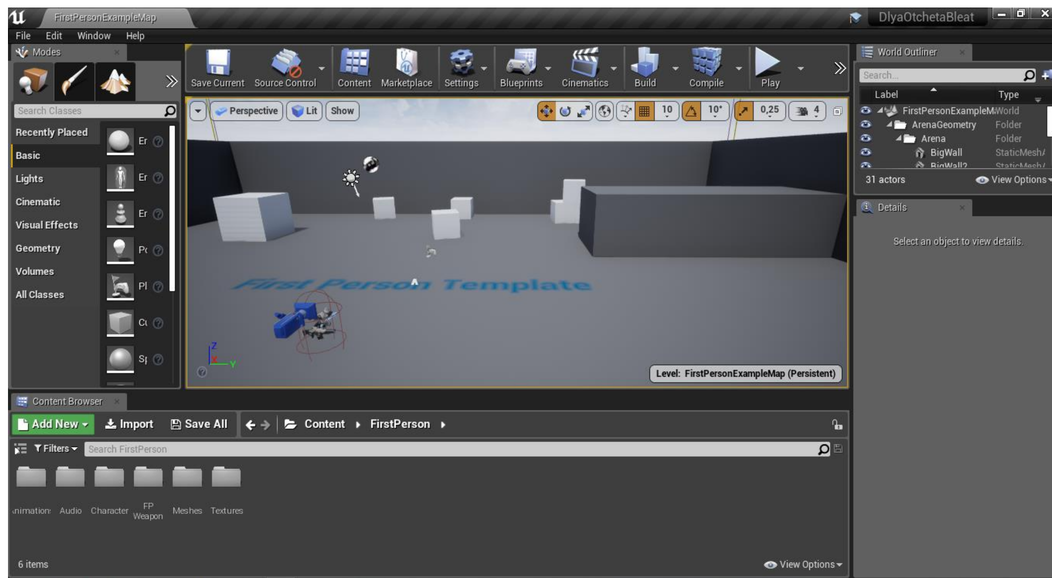

Рис. 1 – Інтерфейс рушія Unreal Engine 4

Ігровий рушій частіше всього має два режими роботи: режим розробки (зазвичай він буває у такому вигляді, що зберігається в SDK) та у відеоіграх-продуктах. У режимі розробки рушій повинен мати можливість створення інтерактивності за допомогою програмування скриптів, розміщення об'єктів на віртуальній сцені за допомогою відповідного інтерфейсу. У користувацькому режимі ігор-продуктів, рушій має завантажити налаштовану сцену ігрового світу та виконувати ігрові скрипти, опрацьовувати користувацькі дії(огляд, переміщення, вибір об'єктів, тощо).

Люди краще всього сприймають інформацію візуально, тому ключовий елемент ігрового рушія - це система отримання графіки. Враховуючи те, що ігри відрізняються від інших художніх творів інтерактивностю, то потребується система скриптів, що дозволяє ігровому світу реагувати на дії гравця.

Відповідно при розробці потрібно сфокусуватися на зручному графічному інтерфейсі та системі скриптів.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Актуальність та мета роботи

Метою даного проекту є створення ігрового рушія - умовно фрейморка, платформи для створення тривимірний ігор, інтерактивність яких забезпечується за допомогою скриптів на мові Python. Проект повинен використовувати для відтворення об'єктів графічне API Vulkan [5] [4]. Даний проект дозволить відпрацювати описані вище дослідницькі завдання.

Виходячи із зазначеної мети, можна виділити особисті завдання, поставлені в даній роботі:

1. Провести детальний аналіз предметної області та вибрати технічні засоби реалізації;
2. Розробити програму, що створює вікно для рендерингу, рендерить в нього тривимірний ігровий світ і оновлює його за допомогою логіки скриптів.

Актуальність даного завдання обґрунтовується тим, що відеоігри є одним з найбільш щвидко розвиваючуюся медіасферою, в тому числі одним з факторів розвитку як програмного оточення користувачів, так і обчислювальної потужністі, зокрема нових відеокарт компанією Nvidia, яка першою ввела в свої пристрої інтегровані блоки для обчислення трасування променів. У наслідок розвитку апаратної бази і вимог продуктів, підвищується необхідність у фахівцях, здатних на доопрацювання існуючих проектів або створення нових, орієнтованих на свою специфіку.

1.2 Опис API Vulkan

Vulkan [4] [5] - це програмний інтерфейс для управління такими пристроями, як графічні процесори. Vulkan є логічним наступником OpenGL, але незважаючи на це, сильно від нього відрізняється. Однією з таких відмінностей є його надлишковість. Для будь-яких дій вимагається заповнення структур, а

помітні зміни, на кшталт рендерингу, вимагають заповнення командного буфера і відправку його в графічну чергу. До того ж більшу частину того, що раніше робив драйвер OpenGL, тепер є обов'язком програміста. Це включає в себе синхронізацію, планування, управління пам'яттю і т. п. Також Vulkan був розроблений для того, щоб бути більше, ніж просто графічним API. Він може використовуватися для різних пристроїв, таких як GPU, DSP і обладнання, що використовує заздалегідь задану функціональність. Функціональність розділяється на кілька великих категорій. Поточна версія Vulkan визначає категорію перенесення, яка використовується для копіювання даних; обчислювальну категорію, яка використовується для виконання обчислювальних шейдеров; і графічну категорію, що включає растеризування, складання примітивів, змішування, тест глибини і трафарету і іншу функціональність.

Певною мірою підтримка кожної з цих категорій є необов'язковою, може бути пристрій на Vulkan, який взагалі не підтримує графіку. Як наслідок навіть API, призначений для виведення картинки на екран, не просто є необов'язковим, але і надано як розширення Vulkan, а не його базова частина.

1.3 Ієрархія основних об’єктів API Vulkan

Vulkan включає в себе ієрархію різних сутностей, що починаються зверху з екземпляру, що містить все підтримуючі Vulkan пристрої разом. Кожен пристрій надає одну, або кілька черг. Саме через ці черги і виконується вся запитувана додатками робота. Екземпляр Vulkan - це програмна конструкція, яка логічно відокремлює стан однієї програми від інших додатків або від бібліотек, які виконуються в контексті програми.

Фізичні пристрої в системі представлені як члени екземпляру, кожний з них має різні можливості, включаючи набір підтримуваних черг. Фізичний пристрій зазвичай представлено окремим пристроєм або набором пов'язаних між собою

пристроїв. У будь-якій заданій системі існує фіксоване і кінцеве число пристроїв, звичайно, якщо не підтримує переконфігурацію прямо на ходу.

Логічний пристрій, що створюється екземпляром - це програмна конструкція, що обертає фізичний пристрій і представляє зарезервований набір ресурсів, пов'язаних з конкретним фізичним пристроєм. Він включає в себе можливу підмножину доступних черг на фізичному пристрої. Можна створити кілька різних логічних пристроїв, що представляють один і той же фізичний пристрій, і саме з логічним пристроєм додаток і буде працювати більшу частину часу.

Наприклад, в системі є три фізичні пристрої, які доступних обом екземплярам. Додаток створює один логічний пристрій на першому фізичному пристрої, два логічних пристрої на другому фізичному пристрої і ще один пристрій на третьому. Кожний логічне пристрій задіює певну підмножину черг відповідного фізичного пристрою.

На рис. 2 зображений приклад вищеописаної ієрархії.

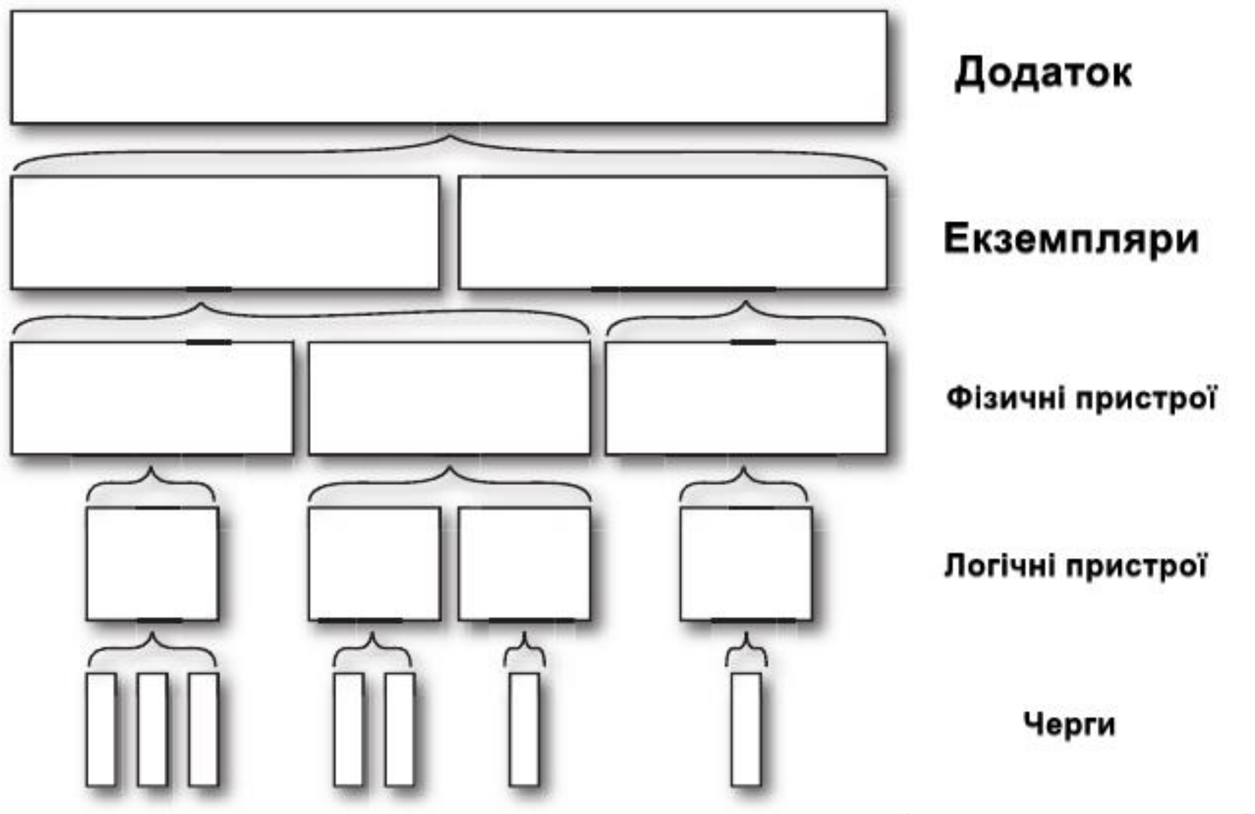

Рис. 2 – Ієрархія основних об'єктів Vulkan

1.4 Домовленість про типи об’єктів та функцій

Практично все в Vulkan представлено за допомогою об'єктів, і кожен об'єкт керуються за допомогою дескриптора. Всі дескриптори діляться на дві великі категорії: об’єкти, шо диспетчеризуються та ті, що не диспетчеризуються об'єкти. Об’єкти, що диспетчеризуються - це об'єкти, які всередині себе зберігають таблицю диспетчеризації. Це таблиця функцій, яка використовується різними компонентами для визначення того, які частини коду виконати, коли додаток викликає Vulkan. Такі об'єкти зазвичай є досить важкими і на даний момент складаються з екземпляру (VkInstance), фізичного пристрою (VkPhysicalDevice), логічного пристрою (VkDevice), буфера команд (VkCommandBuffer) і черги (VkQeue). Всі інші об'єкти вважаються недиспетчеризуємими.

Першим аргументом будь-якої функції Vulkan завжди є диспетчеризуємий об'єкт. Єдиними винятками з цього правила є функції, що служать для створення і ініціалізації екземпляру.

1.5 Управління пам’яттю

Vulkan надає два типи пам'яті - пам'ять CPU (host memory) і пам'ять GPU (device memory). Об'єкти, які створюються Vulkan API, зазвичай вимагають деякої кількості пам'яті CPU. Це те місце, де реалізація Vulkan буде зберігати стан об'єкта і всі дані, необхідні Vulkan API. Ресурсні об'єкти, такі як буфери і зображення, вимагають деякого об'єму пам'яті GPU. Це та пам'ять, де зберігаються дані цього ресурсу. Потрібно, щоб додаток саме керувало пам'яттю GPU. Кожен створюваний ресурс може повернути інформацію про обсяг і тип пам'яті, яка потрібна для його зберігання. Додаток саме має виділити правильну кількість пам'яті і приєднати його до ресурсного об'єкту перед його використанням.

У високорівневих API, таких як OpenGL, все це виконується драйверами від додатку. Однак деяким додаткам потрібно дуже велику кількість маленьких ресурсів, а іншим потрібно невелика кількість дуже великих ресурсів. Деякі додатки створюють і знищують ресурси під час своєї роботи, в той час як інші програми створюють всі свої ресурси на початку роботи програми і не звільняють їх до кінця роботи програми.

Стратегії для виділення пам'яті в цих випадках можуть сильно відрізнятися. Немає будь-якої універсальної стратегії, яка підійде всім. Драйвер OpenGL не знає, як додаток буде себе вести, і повинен підлаштовувати свої стратегії виділення в спробі підлаштуватися під роботу програми.

Важливо зауважити, що кожне виділення пам'яті, під час роботи програми, є важкою операцією. Тому важливо мінімізувати число виділених об'єктів. Рекомендується виділяти пам'ять GPU великими блоками.

1.6 Багатопотоковість в Vulkan

Підтримка багатопоточних додатків є важливою частиною дизайну Vulkan. Зазвичай Vulkan вважає, що додаток гарантує, що ніякі два потоку не будуть одночасно змінювати один і той же об'єкт. Це називається зовнішньою синхронізацією. Переважна більшість команд Vulkan, критичних для швидкодії (таких як побудова командних буферів), взагалі не надає ніякої синхронізації.

Для того щоб точно визначити, які вимоги різних команд Vulkan до поведінки окремих потів, кожен параметр, який повинен бути захищений від одночасного доступу CPU, позначається як зовні синхронізований.

У деяких випадках дескриптори об'єктів та інші дані вкладені в структури, в масиви або передаються якимось іншим неявним способом. Всі ці параметри також мають бути зовні синхронізовані.

Ця діяльність спрямована на те, що реалізації Vulkan ніколи не потрібен був внутрішній м'ютекс або якийсь інший синхронізаційних примітив для захисту даних. Це означає, що багатопотокові програми вкрай рідко стикаються з блокуваннями або очікуваннями потоку.

1.7 Графічний конвеєр Vulkan

Графічний конвеєр в Vulkan може розглядатися як виробнича лінія, де команди надходять в початок конвеєра і обробляються за стадіями конвеєра. Кожна стадія виконує певний вид перетворень: вона бере команди і пов'язані з

ними дані та перетворює їх в щось інше. В кінці конвеєра команди виявляються перетвореними в різнокольорові пікселі, що утворюють вихідну картинку.

Багато частин конвеєра є необов'язковими і можуть бути виключеними або навіть не підтримуватися конкретною реалізацією Vulkan. Єдиною частиною конвеєра, яку програма має включити, є вершинний шейдер.

Короткий опис кожної стадії конвеєра:

1) виведення примітивів: це те, де команди надходять на вхід графічного конвеєра Vulkan. Зазвичай невеликий процесор або спеціальна апаратна частина всередині пристрою Vulkan інтерпретує команди в командному буфері і безпосередньо взаємодіє з апаратурою для виконання роботи;
2) вхідна збірка: на цій стадії читаються буфери з індексами і вершинами, що містять інформацію про вершини, які виводяться;
3) вершинний шейдер: це те місце, де виконується вершинний шейдер. Він отримує властивості вершини і готує перетворені і підготовлені дані у вершини для наступної стадії;
4) керуючий шейдер теселяції: ця стадія відповідає за підготовку параметрів теселяції та інших даних, що відносяться до всього примітиву теселяції, які будуть використовуватися апаратним теселятором;
5) генерація примітивів теселяції: ця стадія використовує параметри теселяція, підготовлені керуючим шейдером теселяції, для розбиття вхідних примітивів на багато маленьких, простих примітивів, готових до обробки теселяційним обчислювальним шейдером;
6) теселяційний обчислювальний шейдер: ця стадія виконується для кожної вершини, створеної на стадії генерації примітивів теселяції. Вона працює аналогічно вершинному шейдеру, за винятком того, що вхідні вершини створюються, а не читаються з пам'яті;

7) геометричний шейдер: ця стадія працює з цілими примітивами. Примітиви можуть бути точками, відрізками, трикутниками або ж спеціальними їх варіантами, що включають в себе додаткові вершини поруч з ними. Ця стадія також має можливість змінювати в собі тип примітиву;
8) збірка примітивів: ця стадія групує вершини, створені на вершинних, теселяційних і геометричних стадіях, в примітиви, які підходять для растеризації. Вона також відкидає і відсікає примітиви і перетворює їх у відповідне вікно;
9) відсікання і відкидання: на цій фіксованій (непрограмованій) стадії визначається, які частини яких примітивів можуть вплинути на вихідне зображення і відкидаються частини, які не впливають на нього. Після цього відбувається надсилання потенційно видимих примитивів на растеризатор;
10) растеризатор: растеризация - це фундаментальна частина всієї графіки в Vulkan. Растеризатор бере зібрані примітиви, які все ще представлені послідовністю вершин, і перетворює їх в окремі фрагменти, які можуть стати пікселями, які формують зображення;
11) передфрагментні операції: деякі операції можуть бути виконані над фрагментами, як тільки їх положення стає відомим, але до визначення їх кольору. Ці передфрагментні операції включають в себе тести глибини і трафарету, коли вони включені
12) збирання фрагментів: стадія збірки фрагментів бере вихід з растеризатора разом з будь-якими даними фрагмента і посилає їх як групу на стадію фрагментного шейдера;
13) фрагментний шейдер: ця стадія виконує останній шейдер в конвеєрі, що відповідає за обчислення даних, які будуть надіслані на останні стадії конвеєра;

14) постфрагментні операції: в деяких випадках фрагментний шейдер змінює дані, які зазвичай використовуються на передфрагментних операціях. У цих випадках ці передфрагментні операції виконуються тут;
15) змішання кольорів: операції з кольорами беруть остаточні результати фрагментного шейдера і постфрагментних операцій і використовують їх для зміни вмісту фреймбуферу. Кольорові операції включають в себе змішання кольорів та логічні операції.

1.8 Скриптування за допомогою Python API

Пакет Python поширюється разом з набором заголовних файлів і бібліотек, націлених на розширення можливостей цієї мови шляхом взаємодії між скриптами цією мовою і модулів, написаних на C/C++. Інтерфейс SDK дозволяє як створювати модулі для Python, так і робити виклик з C/C ++ коду. Це реалізується шляхом включення в першокод заголовки Python.h та лінковки програми з відповідної бібліотекою. В результаті програміст має можливість оперувати об'єктами інтерпретатора Python і виробляти виклик скриптових функцій. Управління даної інтеграцією проводитися шляхом використання API функцій і маніпулювання об'єктами PyObject*, які можуть представляти будь-яку сутність мови Python - змінні різних типів, функції, цілі модулі. Недоліком даного скриптового мови є відсутність наївною підтримки багатопотоковості. Реалізований в мові Global Interpreter Lock (GIL - глобальний замок інтерпретатора) є захистом внутрішніх даних інтерпретатора від небезпечної зміни станів і допускає в межах виконання одного екземпляра інтерпретатора тільки один активний потік

2. ТЕХНІЧНЕ ЗАВДАННЯ

2.1 Опис області використання та початкових даних додатка

Створена програма призначена для створення на своїй базі ігрових програм, які мають вивід тривимірної графіки і реалізовані на базі мов Python. Областю застосування розроблюваного продукту є інді-геймдев, в ситуаціях, коли розробка власного рушія є занадто дорогим завданням, а великі продукти на кшталт Unreal Engine 4 [7] [8] занадто надлишкові, або не влаштовують умовою ліцензування.

Початковими даними для рушія є файли текстур в форматі *.tga, моделей в форматі *.kem, конфігурацій ігрових об'єктів в форматі *.class і скриптів в форматі *.py.

Розробити додаток, що виконує рендеринг тривимірних мешів, і виконує обробку скриптових команд на мові Python.

Для тривимірних моделей розробити власний формат зберігання моделей, написати плагін до редактора тривимірних моделей Blender. дозволяє здійснювати експорт в вищезгаданий формат.

2.2 Вимоги до апаратних, програмних і комунікаційних інтерфейсів

Розроблюваний продукт є багатоплатформним, тому доступний для використання на операційних системах Linux та Windows.

Для функціонування обов'язковим є підтримка API Vulkan відеокартою кінцевого пристрою і наявність відповідного драйверу.

2.3 Вимоги до файлів формату *.kem

Формат *.kem – це бінарний формат даних тривимірних геометричних моделей. Код плагіна експорту цього формату з редактора Blender наведено в додатку. Файл формату *kem містить:

1) константу - заголовок HEADER розміром 3 байта - завжди повинен мати шістнадцяткове значення 0x4b454d;
2) константу CONSTANT розміром 7 байт - завжди має значення шістнадцяткове значення 0x494c5518112015;
3) байт, що описує версію формату, яка використовується під час запису файлу;
4) п'ять зарезервований байт для майбутніх розширення можливостей формату;
5) послідовність блоків даних.

Кожний блок має наступну структуру:

1) байт, який описує тип блоку;
2) байт, для бітових прапорів
3) два зарезервованих байти для майбутнього розширення можливостей формату;
4) чотири байти для зберігання 32-бітного беззнакового цілого числа, яке описує розмір даних в блоці;
5) послідовність байтів даних, яка має фіксований розмір, зазначений в попередньому полі.

На даний момент (перша версія формату) блоки можуть мати такі типи:

1) 0x01 - крапки - поле DATA містить послідовність масивів [x, y, z] координат, де кожна з них є числом з плаваючою комою;
2) 0x02 - нормалі - поле DATA містить послідовність масивів [x, y, z] значення, де кожне є числом з плаваючою комою, разом описують вектор нормалі певної крапки;
3) 0x03 - UV - поле DATA містить послідовність [u, v] кординат, де кожна з них є числом з рухомою комою, і дані координати описують відображення точок моделей на двомірну текстуру;
4) 0x04 - меш - поле DATA містить масиви індексів, де кожні три індексу становлять трикутник. Всі індекси зберігаються як 4 байтові беззнакові цілі числа. Наприклад, послідовність індексів (0,1,2,1,3,2) містить два трикутника із загальним ребром.

Блоків кожного типу може бути кілька в файлі. В цьому випадку нумерація індексів, що містяться в них, буде продовжена щодо останнього блоку того ж типу. Поле прапорів у всіх випадках зарезервовано на майбутнє розширення можливостей формату.

3. ОБГРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ ТА ПРОГРАМНИХ ЗАСОБІВ

3.1 Мова розробки

Для розробки обрана мова програмування C ++, тому що вона є найбільш зручним інструментом для розробки низькорівневих продуктів, бо поєднує в собі наявність сучасної ООП парадигми і ручного управління пам'яттю. Дана мова має компілятори на всіх актуальних платформах, а його помірна частота оновлення стандарту дозволяє тримати код сучасним і зручним для читання без регулярних змін. Так само він має сумісність з мовою програмування Сі, який є стандартом до більшості API, що дозволяє інтегрувати в майбутньому будь-які сторонні рішення та бібліотеки.

3.2 Бібліотека GLFW

Можна декількома шляхами створити вікно графічної оболонки. З найбільш відомих бібліотек, для створення графічних вікон, можна назвати GLFW [6], SDL, GLUT [9], SFML і ручна реалізація за допомогою платформозалежних API.

GLUT [9] не підтримує Vulkan [4] [5], також він є застарілою технологією. SFML є високорівневою мультимедійної бібліотекою, яка абстрагує процес рендерингу від програміста, що суперечить меті даної роботи. Власне рішення не розглядається через що виникає складності при портуванні на операційну систему Linux, тому що вимагає підтримки одночасно декількох графічних серверів - Wayland і X11. Був обраний GLFW, бо остання є об'єктно-орієнтованою, а інтерфейс SDL написаний функціонально під мову Сі.

3.3 Графічний API Vulkan

Не дивлячись на те що існують готові бібліотеки для рендерингу об'єктів, більш оптимальним і відповідним поставленому завданню рішенням буде вибір графічного API, що дозволить підвищити продуктивність, на яку впливають сторонні бібліотеки. Порівняння GAPI з перерахуванням можна знайти в таблиці 3.1 [10]:

Таблиця 3.1 – порівняння графічних API

API	Платформы						
	Windows	Linux	MacOS	iOS	Android	XboxOne	PS4
OpenGL	+	+			+		
Vulkan	+	+			+		
DirectX	+					+	
Metal			+	+			
Gnm							+

Виходячи з цих даних, найпоширеніші графічні API це OpenGL та Vulkan. Vulkan на відміну від OpenGL розроблений як низькорівневий та багатопоточний, тому саме він обраний для розробки.

3.4 Інтерпретована скриптова мова

Для скриптовой системи було вирішено використовувати готовий інтерпретатор якої-небудь скриптової мови, що дозволить заощадити ресурси на розробку парсеру і супутню логіку.

Був обраний Python, бо він має готовий Сі API, який розповсюджується разом з основним пакетом. Важливим фактором є зручність самої мови, простота якого дозволяє легко створювати прототипи ігровий логіки.

3.5 Інструменти для розробки

Як IDE був обраний QtCreator [3], перевагою якого є багатоплатформеність та готова прив'язка до налагоджувача gdb, системі контролю версій Git і профайлером valgrind.

4. РОЗРОБКА БАГАТОПЛАТФОРМНОГО ІГРОВОГО РУШІЯ З ІНТЕРПРЕТАТОРОМ СКРИПТОВИХ КОМАНД

4.1 Структура проекту

Корінь проекту розбитий на тематичні директиви. У корені проекту розміщені використовувані ресурси. Логіка рушія передбачає обов'язкову наявність в кореневій директорії файлу main.py.

Папка BlenderPlugin містить код плагіна до редактора тривимірних моделей Blender [11], який дозволяє здійснювати експорт різної інформації - моделей, матеріалів, анімацій, допоміжних даних.

Папка Shaders містить використовувані рушієм вершинні і фрагментні шейдери на мові GLSL, які Vulkan компілює в байт код на мові SPIR-V.

Папка Source містить першокод рушія.

4.2 Опис класів

Таблиця 4.1 – Класи додатка

Назва класу	Опис класу
OptionsParser	Клас для парсингу параметрів з командного рядка та файлів конфігурацій.
Log	Клас для запису інформації до файлів логу. Потокобезпечний.
TickingObject	Абстрактний клас для класів, які повинні виконуватися з чітко заданої періодичністю.
MeshesLoader	Клас для завантаження даних з файлів формату (*.kem)

Продовження таблиці 4.1

Назва класу	Опис класу
ResourcesLoader	Абстрактний клас для класів, що виконують завантаження даних з жорсткого диска.
TgaLoader	Клас для завантаження зображень в форматі *.tga.
Buffer	Клас-обгортка над об'єктом VkBuffer.
DescriptorPoolManager	Клас-обгортка для аллоцірованія дескрипторів ресурсів
Device	Клас-обгортка над сукупністю діспетчерізуйомих сутностей API Vulkan.
Image	Клас-обгортка над об'єктами VkImage, vkDeviceMemory, VkImageView и VkSampler.
Mesh	Клас для зберігання даних геометрії.
Engine	Клас, керуючий усіма підсистемами рушія.
Renderer	Клас, що виконує відтворення зображення графіки.
ResoucesMaker	Клас, який створює ресурси для рендерера.
Shader	Клас-обгортка над об'єктом VkShaderModule.
SingleTimeCommandBuffer	Клас-обгортка над об'єктом VkCommandBuffer.
PythonModule	Клас, керуючий скриптами на мові Python.
MeshObject	Клас, який є контейнером для ряду властивостей, що описують один об'єкт ігрового світу.
World	Клас ігрового світу, який реалізує управління об'єктами і кешування ресурсів.

Клас OptionsParser призначений для збору параметрів з командного рядка або файлу конфігурації. Він реалізований за рахунок гетерогенних контейнерів, введених в мові програмування C++ версії C++17. Основні типи зібрані в typedef для контейнера std::variant під псевдонімом Variant, а аналогічні для цих типів покажчики в typedef контейнера під псевдонімом PointerVariant.

Клас OptionsParser містить в собі внутрішній клас Options, який в свою чергу має внутрішню структуру Option, яка складається з чотирьох полів: іменного ключа, рядки опису, контейнер значення і контейнер покажчика. Для роботи з гетерогенним контейнерами використовується функція std :: visit і два функтора - visitorPointer і visitorStream. Схематичне зображення взаємозв'язків описаних вище класів зображено на рисунку 4.1.

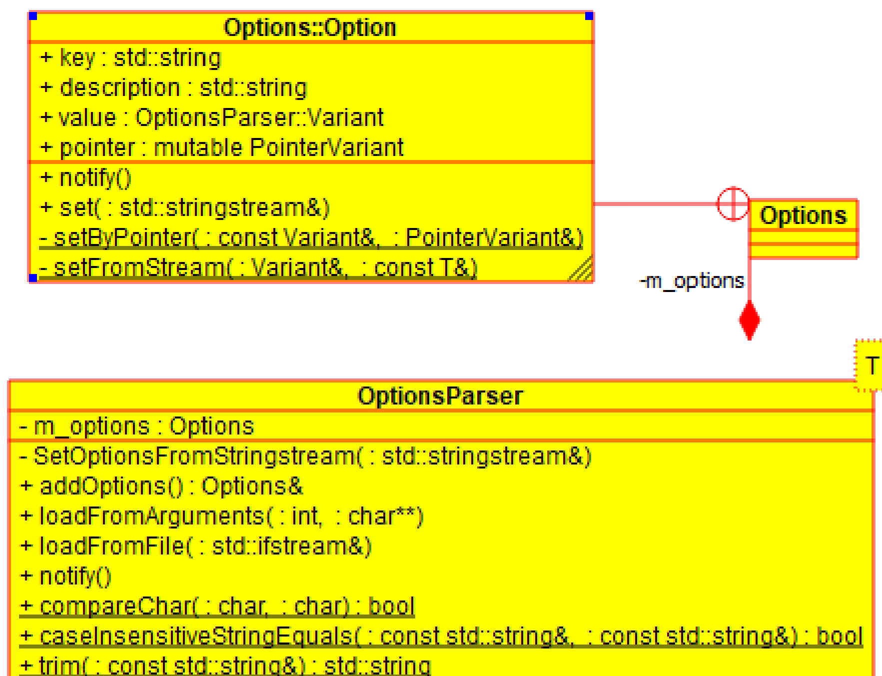

Рис. 4.1 – Взаємозв'язок класів OptionsParser. Options та Option

Опис полів і методів класу Option приведено в таблиці 4.2

Таблиця 4.2 – методи та поля класу Option

Атрибут класу	Опис атрибуту
key	Строковий ключ назви опції
description	Рядок-опис опції
value	Контейнер для гетерогенного зберігання значень параметра
pointer	Контейнер для гетерогенного зберігання покажчика на цільову змінну, в яку буде завантажений отримане значення поля value
notify	Функція, яка вивантажує значення в змінну за посиланням в поле pointer. використовує функцію std::visit та статичні функції, описані нижче
Set	Функція, отримує значення змінної з потоку
setByPointer	Шаблонова функція для присвоєння змінної по посиланню будь-якого значення, при чому обидва параметри повинні між собою бути T і T*
setFromStream	Шаблонна функція для присвоєння змінної по посиланню будь-якого значення, при чому обидва параметри повинні між собою бути T і T*

Клас Option використовується для заповнення вектора в класі Options, який грає роль контейнера для безлічі опцій. Опис полів і методів класу Options приведено в таблиці 4.3.

Атрибут класу	Опис атрибуту
m_options	Вектор опцій
keyIsBoolean	Метод для перевірки того, що ключ є булевим значенням
contains	Метод, який дозволяє перевірити наявність ключу серед наявних опцій
Options& operator()	Оператор, що дозволяє використовувати об'єкт цього класу як функтор, який заповнюється шляхом виклику себе, що підвищує читаність коду
setByKeyFromStream	Передає строковий потік зазначеної по рядку-ідентифікатором опції
notify	Сповіщає всі наявні опції про необхідність передати своє значення в зазначені ним покажчики.

Опис полів і методів класу OptionsParser приведено в таблиці 4.4.

Таблица 4.4 – методы и поля класса OptionsParser

Атрибут класу	Опис атрибуту
m_options	Об’єкт класу Options
SetOptionsFromStringstream	Передає потік об'єкту m_options
addOptions	Повертає об'єкт m_options для використання в якості функтора.
loadFromArguments	Завантажує параметри з командного рядка додатка
loadFromFile	Завантажує параметри з файлів, переданих як std::ifstream
notify	Повідомляє об'єкт класу m_options про необхідність передати значення в змінні, по збереженим в Option покажчикам.
compareChar	Порівнює два символи char
caseInsensitiveStringEquals	Не чутлива до регістру перевірка рядків на еквівалентність
trim	Функція для видалення з рядків зайвих прогалин на початку і кінці

Для логування системної, налагоджувальної та іншої інформації в рушії використовується клас Log. Схема класу наведена на рисунку 4.2.

```
Log
- m_logFile : std::ofstream
- m_mutex : std::mutex
+ getInstance() : Log&
+ callbackForGlfw( : int,  : const char*)
+ deleteDebugCallback()
+ operator <<(value : const T&) : Log&
+ operator <<(pf : OstreamManipulator) : Log&
+ ~ Log() «destructor»
- Log() «constructor»
```

Рисунок 4.2 – Схема класу Log

Опис полів і методів класу Log приведено в таблиці 4.5.

Таблиця 4.5 - методи і поля класу Log

Атрибут класу	Опис атрибуту
m_logFile	Представлений потоком ofstream файл, в який проводитиметься логування інформації
m_mutex	М'ютекс для синхронізації багатопотокового доступу до Сінглтон об'єкту класу
getInstance	Метод, який повертає посилання на Сінглтон об'єкт класу

Продовження таблиці 4.5

Атрибут класу	Опис атрибуту
callbackForGlfw	Метод для виведення інформації бібліотеки GLFW шляхом передачі покажчика на цю функцію для зворотного виклику
deleteDebugCallback	Видаляє всі присвоєні зворотні виклики
Перевантаження оператора <<	Встановлює через м'ютекс блокування від багатопоточного доступу і виводить в файл подані дані

Для реалізації класів, які повинні спрацьовувати зі встановленою періодичністю використовується абстрактний клас TickingObject. Він реалізований за допомогою методу start(), який створює новий потік. У цьому потоці код очікує заданий в конструкторі час, після чого викликає віртуальний метод tick(). Це продовжується до тих пір поки встановлений прапор об'єкта, який скидається викликом методу stop(). Прапор має тип atomic_bool, тому він безпечний для одночасного доступу з внутрішнього потоку і ззовні одночасно.

Схема класу наведена на рисунку 4.3.

TickingObject

- m_interval : std::chrono::milliseconds
- m_isRunning : std::atomic_bool

+ TickingObject(: const std::chrono::milliseconds&) «constructor»
+ ~ TickingObject() «destructor»
+ start()
+ stop()
- *tick()*

Рисунок 4.3 – Схема класу TickingObject

Опис полів і методів класу TickingObject приведено в таблиці 4.6.

Таблиця 4.6 - методи і поля класу TickingObject

Атрибут класу	Опис атрибуту
m_interval	Інтервал в мілісекундах, який повинен проходити між викликами методу tick(), задається в конструкторі
m_isRunning	Прапор, який визначає чи повинен цикл роботи всередині об'єкту тривати. Керується методами start () та stop ()
Start	Створює новий потік, в якому буде виконуватися виклик віртуального методу tick() поки значення прапора m_isRunning не буде встановлено в false
stop	Перемикає внутрішній прапор, тим самим завершуючи роботу об'єкту

Головним в ієрархії рушія є клас Engine. При ініціалізації він створює віконне додаток за допомогою бібліотеки GLFW, створює та налаштовує конфігурацію модулів, так само створює об'єкт класу World для передачі в ті модулі, які потребують знати стан ігрового світу. Схема класу наведена на рисунку 4.4

Engine
- m_instanceName : std::string - m_engineSettings : EngineConfig - m_renderer : Renderer::Renderer* - m_pythonModule : PythonModule* - m_glfwWindowHandler : GLFWwindow* - m_world : std::shared_ptr< World >
+ Engine(: const std::string&, : const EngineConfig&) «constructor» + ~ Engine() «destructor» + run() - keyCallback(: GLFWwindow*, : int, : int, : int, : int) - resizeCallback(: GLFWwindow*, : int, : int)

Рисунок 4.4 – Схема класу Engine

Опис полів і методів класу Engine наведено в таблиці 4.7.

Атрибут класу	Опис атрибуту
m_instanceName	Ім'я об'єкту, використовується для заголовка вікна і створення об'єкта класу VkInstance
m_engineSettings	Структура EngineConfig, передається через конструктор і визначає параметри роботи програми.

Продовження таблиці 4.7

Атрибут класу	Опис атрибуту
m_renderer	Об'єкт класу Renderer, що виконує рендеринг ігрового світу в вікно, створене через функції бібліотеки GLFW, так само містить об'єкти для створення ресурсів і роботи з пам'яттю GPU
m_pythonModule	Підмодуль, що запускає інтерпретатор скриптової мови Python і забезпечує взаємодію скриптів та ігрового світу
m_glfwWindowHandler	Платформозалежний хендлер вікна, що повертається функцією glfwCreateWindow. Використовується для зміни параметрів вікна, а також передається в клас Renderer для створення контексту рендеру
m_world	Клас, що описує ігровий світ. Після створення передається об'єктам класів Renderer і PythonModule. захист від зміни станів реалізована всередині самого класу World.

Закінчення таблиці 4.7

Атрибут класу	Опис атрибуту
run	Запускає ігровий цикл, в якому основний потік займається опитуванням платформозалежних подій (дані пристроїв введення, зміну розміру вікна), а всі підмодулі запускають власні незалежні потоки.
keyCallback	Функція зворотного виклику для бібліотеки GLFW, яка перехоплює натиснення клавіш
resizeCallback	Функція зворотного виклику для бібліотеки GLFW, яка перехоплює зміну розміру вікна програми

Взаємодія скриптів і розробленого рушія забезпечується за рахунок класу PythonModule. Той в свою чергу використовує клас PythonObject, який в свою чергу є обгорткою над типом PyObject і має всередині себе управління інкрементуванням і декрементуванням лічильника посилань на об’єкт, що зберігається, який використовується в Python для управління пам'яттю.

Схема класів наведена на рисунку 4.5.

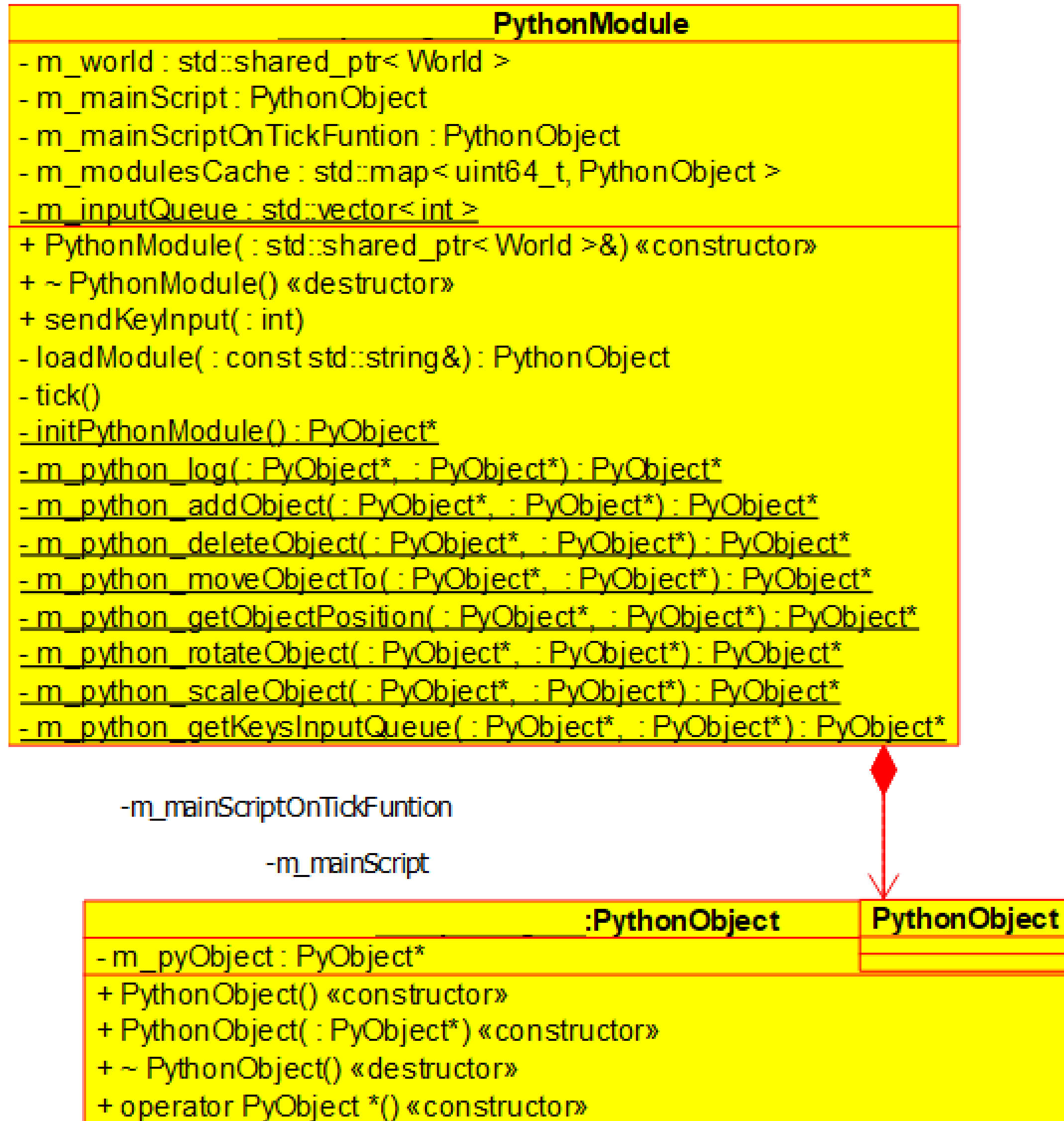

Рисунок 4.5

Опис полів і методів класу PythonObject приведено в таблиці 4.8

Атрибут класу	Опис атрибуту
m_pyObject	Об'єкт PyObject*, навколо чого і є обгорткою даний клас. При захопленні значення або виклику деструктора викликаються відповідні методи Py_DECREF та Py_INCREF
operator PyObject*	Повертає значення поля m_pyObject для використання у функціях Python C API, при цьому викликається інкремент лічильника посилань на цей об'єкт (Py_INCREF)

Таблиця 4.8

Описание полей и методов класса PythonModule приведено в таблице 4.9.

Атрибут класу	Опис атрибуту
m_world	Клас, що описує ігровий світ. Після створення передається об'єктам класів Renderer і PythonModule. захист від зміни станів реалізована всередині самого класу World.
m_mainScript	Об'єкт PythonObject, кешує в собі головний скрипт main.py, який за замовчуванням завантажується на старті програми і слугує для збереження глобальних змінних скрипта

Продовження таблиці 4.9

Атрибут класу	Опис атрибуту
m_mainScriptOnTickFuntion	Об'єкт PythonObject, що зберігає в собі об'єкт, який представляє функцію onTick зі скрипта main.py, яка виконується кожну ітерацію модуля, незалежно від наявності в ігровому світі об'єктів
m_modulesCache	Кеш, який зберігає в собі завантажені скрипти об'єктів індивідуально по ідентифікаторів об'єктів. Це дозволяє зберігати в скриптах глобальні значення змінних
m_inputQueue	Вектор чисел, які представляють собою чисельні коди клавіатурних клавіш і чекають обробки в скриптах.
sendKeyInput	Дозволяє ззовні додати в чергу на введення числової номер клавіші
loadModule	Завантажує по вказаному шляху модуль на мові Python. Якщо вказаний модуль не знайдено або він не коректний, виводиться відповідна помилка в лог і повертається nullptr.

Продовження таблиці 4.9

Атрибут класу	Опис атрибуту
initPythonModule	Ініціалізує інтерпретатор мовою Python і додає в нього модуль для зворотних викликів зі скриптів в статичні функції модуля PythonModule
m_python_log	Функція для виклику з скриптів Python, дозволяє виводити текстову інформацію в файл логування
m_python_addObject	Функція для виклику з скриптів Python, дозволяє додавати в ігровий світ додаткові об'єкти по імені їх ігрового класу. Повертає чисельний ідентифікатор об'єкта
m_python_deleteObject	Функція для виклику зі скриптів Python, дозволяє видаляти об'єкти з ігрового світу по їх чисельному ідентифікатору
m_python_moveObjectTo	Функція для виклику зі скриптів Python, дозволяє пересувати зазначені за чисельним ідентифікатором об'єкти на зазначені координати

Кінець таблиці 4.9

Атрибут класу	Опис атрибуту
m_python_getObjectPosition	Функція для виклику зі скриптів Python, дозволяє за чисельним ідентифікатором об'єкту отримати кортеж його координат
m_python_rotateObject	Функція для виклику зі скриптів Python, дозволяє обертати зазначені за чисельним ідентифікатором об'єкти на зазначені кути кожної з осей тривимірного простору. Не працює з кватерніонами, тому слід побоюватися проблеми шарнірного замку
m_python_scaleObject	Функція для виклику зі скриптів Python, дозволяє масштабувати зазначені за чисельним ідентифікатором об'єкти на зазначені пропорції щодо кожної з осей тривимірного координатного простору
m_python_getKeysInputQueue	Функція для виклику зі скриптів Python, дозволяє отримати список чекаючих обробки кодів клавіатурних клавіш

5. Приклад створення гри на базі розробленого продукту

Для перевірки працездатності продукту була розроблена примітивна гра Тетріс, що використовує тільки Python інтерфейс рушія (рисунок 5.1).

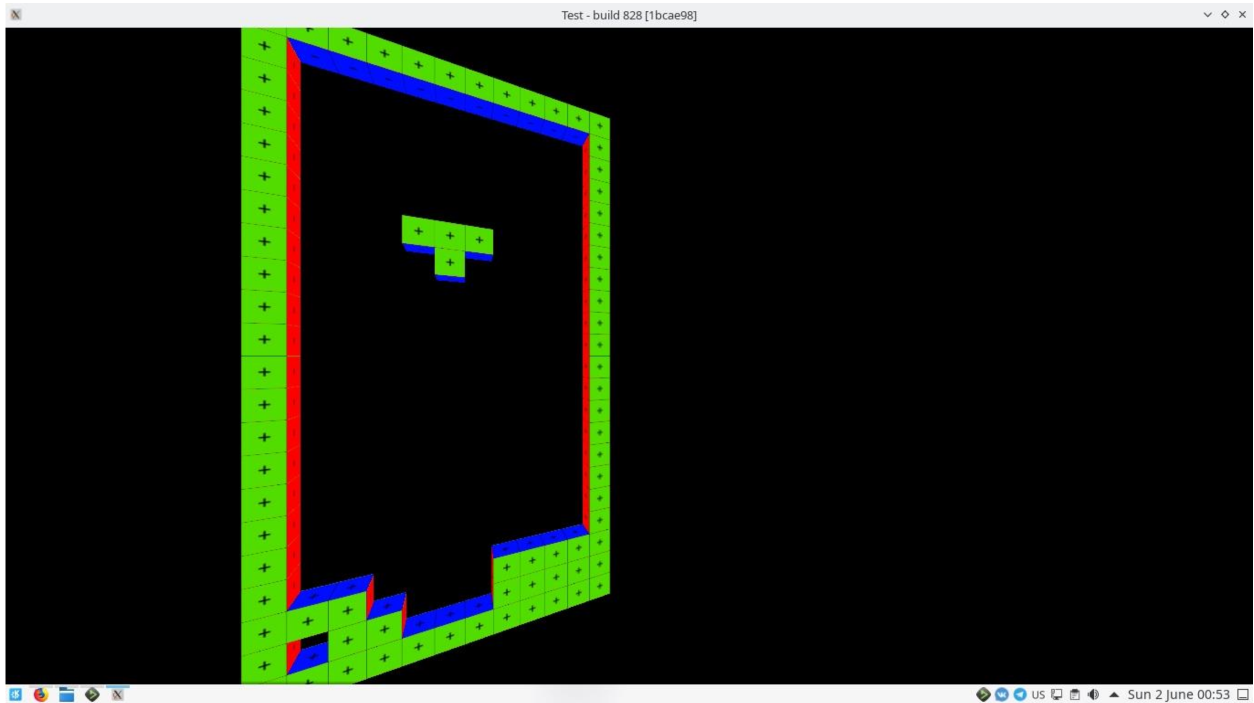

Рисунок 4.9 - Реалізація гри Тетріс на розробленому продукті

Висновок

В ході проведеної роботи вдалося реалізувати продукт, здатний виконувати скрипти на мові Python, здійснювати інтеграцію інтерпретатора мови Python в код C ++ програми, виконувати рендер тривимірних моделей за допомогою графічного API Vulkan.

Для даної задачі був розроблений формат тривимірних моделей, для спрощення і стандартизації даних для завантаження в програму. У процесі роботи були отримані знання графічного конвеєра, такі як растеризация, теселяція, створення фреймбуферів, пулу дескрипторів, командного пулу, створення синхронізації CPU і GPU, вибір найбільш підходящого пристрою GPU серед наявних в системі.

При подальшому розвитку рушія, він може використовуватися як інструмент для створення ігрових програм.

Посилання на використані джерела

[1] B. Stroustrup: Programming -- Principles and Practice Using C++ (Second Edition). May 2014.

[2] Мейерс С. Ефективне використання C ++. 55 вірних способів поліпшити структуру і код ваших програм. - М .: ДМК Прес, 2006. -301 с..

[3] "Офіційна документація Qt," [Online]. Available: https://doc.qt.io/.

[4] ««Vulkan® 1.1.178 - A Specification»,» [Онлайновий]. Available: https://www.khronos.org/registry/vulkan/specs/1.1-khr-extensions/html/chap36.html.

[5] Sellers G. Vulkan Programming Guide. - Addison-Wesley, 2017. -745 с..

[6] «Документація по фреймворку GLFW,» [Онлайновий]. Available: https://www.glfw.org/.

[7] Документація по рушію Unreal Engine 4.

[8] Gregory J. Game Engine Architecture. - NY.: «CRC Press», 2015. – 989 с..

[9] «Офіційна документація щодо GLUT,» [Онлайновий]. Available: https://www.opengl.org/resources/libraries/glut/glut-3.spec.pdf.

[10] V. D. M. G. Офіційна документація щодо платформ: OpenGL. [Онлайновий]. Available: https://www.opengl.org/sdk/docs/tutorials/
https://renderdoc.org/vkspec_chunked/index.html
https://microsoft.fandom.com/wiki/DirectX
https://developer.apple.com/documentation/metal
https://gdal.org/tutorials/gnm_api_tut.html.

[11] «Офіційний сайт Blender,» [Онлайновий]. Available: https://www.blender.org/.