МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
Факультет мехатроніки та комп'ютерних технологій
Кафедра комп'ютерних наук

ДИПЛОМНА БАКАЛАВРСЬКА РОБОТА

на тему

СТВОРЕННЯ КРОСПЛАТФОРМЕННОГО ТА МОДУЛЬНОГО ІГРОВОГО РУШІЯ

Виконав: студент групи БІТ-1-19
спеціальності 122 Комп'ютерні науки
Максим СІВЕРСЬКИЙ
Науковий керівник
Вікторія РЕЗАНОВА
Рецензент
Володимир ЩЕРБАНЬ

Київ 2023

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ

Факультет мехатроніки та комп’ютерних технологій

Кафедра комп’ютерних наук

Спеціальність 122 Комп’ютерні науки

Освітня програма Комп’ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри
комп’ютерних наук

_____ Володимир ЩЕРБАНЬ

«_____» _____ 20 __ р.

ЗАВДАННЯ

НА ДИПЛОМНУ БАКАЛАВРСЬКУ РОБОТУ

студенту
Сіверському Максиму Дмитровичу

1. Тема роботи: Створення кросплатформенного та модульного ігрового рушія Науковий керівник роботи: Резанова Вікторія Георгіївна, доцент кафедри комп'ютерних наук, затверджені наказом КНУТД від “08” листопада 2022 року № 224-уч.
2. Строк подання студентом дипломної роботи: 25.05.2023р.
3. Вихідні дані до дипломної бакалаврської роботи: Розробки кафедри комп’ютерних наук та технологій, рекомендована література, додатки
4. Зміст дипломної бакалаврської роботи: Розділ 1. Теоретичний, Розділ 2. Алгоритмічний, Розділ 3. Програмний.
5. Дата видачі завдання: 06.02.2023р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної бакалаврської роботи	Терміни виконання етапів	Примітка про виконання
1	Вступ	20.04.2023р.	
2	Розділ 1. Теоретична частина	25.04.2023р.	
3	Розділ 2. Алгоритмічна частина	05.05.2023р.	
4	Розділ 3. Програмна частина	15.05.2023р.	
5	Висновки	16.05.2023р.	
6	Оформлення дипломної бакалаврської роботи	20.05.2023р.	
7	Здача дипломної бакалаврської роботи на кафедру для рецензування	25.05.2023р.	
8	Перевірка дипломної бакалаврської роботи на наявність ознак плагіату		
9	Подання дипломної роботи (проекту) на затвердження завідувачу кафедри (за 7 днів до захисту)		

Студент _____ Максим СІВЕРСЬКИЙ

Науковий керівник роботи _____ Вікторія РЕЗАНОВА

Рецензент _____ Володимир ЩЕРБАНЬ

АНОТАЦІЯ

Пояснювальна записка дипломного проекту складається з трьох розділів, містить 2 додатки, 14 рисунків та 20 джерел – загалом 53 сторінки.

Об’єкт дослідження: операційні системи персональних комп’ютерів, ігрові рушії, графічні карти та драйвери до них.

Мета дипломного проекту: дослідження архітектури сучасних ігрових рушіїв та можливостей графічних карт на різних апаратних платформах.

У першому розділі описано дослідження наявних ігрових рушіїв та API взаємодії із графічними прискорювачами.

У другому розділі описана архітектура власного ігрового рушія та представлені дійсні приклади використання отриманих у ході дослідження знань.

У третьому розділі проведено тестування ігрового рушія на різних платформах.

У додатках наведено: похідний код ігрового рушія, необхідні для компіляції інструменти, відео з демонстрацією його роботи, патчі для запуску наявних Windows ігор на Linux та відео з демонстрацією їх роботи.

КЛЮЧОВІ СЛОВА: VULKAN API, ІГРОВИЙ РУШІЙ, ГРАФІЧНИЙ ПРИСКОРЮВАЧ, LINUX GAMING

ABSTRACTS

The explanatory note of the diploma project consists of three sections, contains 2 appendices, 14 figures and 20 sources - a total of 53 pages.

The object of research: operating systems of personal computers, game engines, graphic cards and drivers for them.

The goal of the diploma project: the study of the architecture of modern game engines and the capabilities of graphics cards on various hardware platforms.

The first chapter describes the study of existing game engines and APIs for interaction with graphics accelerators.

The second chapter describes the architecture of our own game engine and presents real examples of the use of the knowledge gained during the research.

In the third section, the game engine was tested on different platforms.

Appendices include: the source code of the game engine, the tools needed for compilation, a video demonstrating its operation, patches for running existing Windows games on Linux, and a video demonstrating their operation.

Keywords: Vulkan API, game engine, graphics accelerator, Linux gaming

ЗМІСТ

ВСТУП

Важко уявити життя сучасної людини без персонального комп'ютера чи смартфона. Дані пристрої можуть сильно полегшити наше життя або зробити його більш цікавим. Одним із таких захоплень є комп'ютерні ігри. Ця тема є досить популярною і не дивно, адже саме для ігор у більшості випадків купуються дорогі комп'ютери, а такі компанії як Microsoft наголошують на ігровій продуктивності в Windows і позиціонують це як головну перевагу своїх систем.

Однак, крім гравців, повинні бути і розробники комп'ютерних ігор. Для них існує велика кількість платних або відкритих для комерційного та некомерційного використання ігрових інструментів, кожен з яких допомагає так чи інакше спростити створення кінцевого продукту. Кажуть, що будь-яка людина, подивившись десяток лекцій на YouTube, може стати розробником комп'ютерних ігор, але чи все так просто насправді… Якщо говорити коротко, то так. Подивившись кілька уроків, можна на основі наявних технологій створити щось схоже. Проте складнощі починаються, коли потрібно створити щось нове. Те, що не дозволяє зробити обраний фреймворк або ігровий рушій, зазвичай так і залишається ідеєю і не отримує подальшого розвитку.

У рамках цієї роботи ми спробуємо зайти з іншого боку: розробити ігровий рушій з нуля, дослідивши всі підводні камені цього процесу, щоб у майбутньому не зіткнутися з вище описаними ситуаціями.

Мета дослідження: вивчити архітектуру сучасних графічних прискорювачів та механізми взаємодії з ними у різних операційних системах.

Завдання дипломного проєкту: розробити платформонезалежний ігровий рушій, який буде практичним посібником з використання графічних та системних API.

РОЗДІЛ 1. ТЕОРЕТИЧНИЙ

1.1. Огляд наявних рішень

Враховуючи шалену популярність комп’ютерних ігор, на теперішній час існує безліч різноманітних ігрових рушіїв. Усі вони мають різне призначення, та різні ліцензії. Дуже невелику кількість рушіїв можна використовувати безкоштовно, і ті лише в навчальних цілях. Серед них було виділено наступні ігрові рушії.

1.1.1. Unreal Engine 5

Сучасний Unreal Engine 5 є еталоном у світі ігрових рушіїв. Він підтримує майже усі платформи для ігор та має безліч можливостей. Ні для кого не секрет, що серед безкоштовних ігрових рушіїв Unreal Engine 5 має найбільш передову графіку та містить власну реалізацію апаратно-незалежного трасування променів. Серед недоліків можна зазначити, що навіть невелика гра на цьму ігровому рушії буде дуже вимоглива до ресурсів системи та займати багато місця на диску.

Слід зазначити, що Unreal Engine 5 можна використовувати безкоштовно лише у некомерційних проектах.

1.2. Методи прискорення математичних обчислень

Найбільш складною задачею ігрового рушія є моделювання реального світу за допомогою математичних абстракцій та вимогливих до ресурсів комп’ютера алгоритмів. Очевидно, що швидкість обчислень має великий вплив на продуктивність ігрового рушія, тому варто приділити цьому увагу ще на етапі проєктування.

1.2.1. Вбудовування функцій

Стандартна бібліотека математичних функцій мови Сі містить функції, які не можуть бути вбудовані до ігрового рушія, а часті звернення до бібліотеки, що розділяється між декількома програмами, може погано

позначитися на продуктивності. До того ж у стандартній бібліотеці використовуються універсальні функції, які не оптимізовані під архітектуру певного процесора. Проте, орієнтуючи ігровий рушій на сучасні комп'ютери та смартфони, можна увімкнути певні оптимізації під більшість сучасних моделей центральних процесорів. Також можливо цілком зібрати ігровий рушій під архітектуру власного процесора, використовуючи аргумент компілятора “-march=native”.

Згадуючи тему про вбудовування функцій, слід зазначити, що компілятор може вбудовувати одні функції в інші, якщо на його думку це підвищить швидкість виконання коду або зменшить розмір бінарного файлу. Такий процес може виконуватись навіть між функціями із різних файлів, при наявності параметру компілятора “-flto”. Але компілятор не може вбудовувати функції динамічних бібліотек. І не дивно, тому вони і називаються динамічними. Отже, в “гарячих” точках програми ми маємо використовувати власні статичні функції, що підвищить загальну швидкість обчислень.

1.2.2. Векторизація та суперскалярність

Алгоритми комп'ютерної графіки часто працюють із векторами. Виконуючи якусь дію, його необхідно застосувати до кількох компонентів вектора. Використання циклів для цих задач вже давно не актуальне, проте працює скрізь. Центральні процесори персонального комп'ютера та мобільного телефону мають спеціальні набори інструкцій, так звані SIMD. Вони можуть застосувати певну операцію відразу до всіх елементів вектора. Реалізовано це рахунок збільшення обчислювальних блоків у кожному ядрі процесора. Примітка: більшість алгоритмів, що працюють зі скалярними величинами не варто перетворювати на векторні з метою збільшення продуктивності, оскільки SIMD інструкції використовують власні регістри та обмін даними між основними та SIMD регістрами – досить часвитратна операція.

Наступним за ефективністю у використанні усіх обчислювальних блоків центрального процесора стоїть суперскалярне виконання інструкцій. Якщо залежності між даними зведені до мінімуму і центральний процесор має гарний апаратний планувальник завдань, то швидкість обчислень буде порівняння з векторними інструкціями, без необхідності вектиризувати код.

Автовекторизація компілятора – це ще один спосіб використовувати SIMD інструкції, не звертаючи уваги на архітектуру центрального процесора. Даний спосіб реалізований як одна з оптимізації компілятора.

Якщо говорити про довіру до оптимізацій компілятора, то можна виділити три різні підходи:

- Не думати про оптимізації взагалі, але під час розробки ігрового рушія це може призвести до значно меншої швидкості роботи у порівнянні з аналогами;
- Довіряти компілятору та лише в деяких випадках явно вказувати на використання певної апаратної інструкції. Це можна робити, наприклад, при багатопотоковій оптимізації, оскільки вона погано розвинена у всіх компіляторах;
- Не довіряти компілятору. Під цим мається на увазі явне використання апаратно-залежних функцій у межах програми. Даний підхід сильно обмежує портативність коду і він становиться підтверджений старінню, так як з виходом нових процесорів буде необхідно так само оновлювати і код під них.

У цій роботі ми схильні цілком довіряти компілятору та його оптимізаціям, тим паче, що є досить багато прикладів, коли скомпільований код працював набагато швидше, ніж нативний під певну архітектуру процесора.

Якщо обирати останній спосіб, то існує метод абстракції апаратно-залежних функцій від реалізації алгоритмів. Суть методу полягає у створенні заголовного файлу з універсальними статичними функціями виду: vec3_add,

vec4_mul, та універсальними типами: vec3_t, vec4_t і т. д. Такі файли потрібно створити для кожного набору інструкцій центрального процесора (SSE, AVX, NEON). Подібний метод дозволив би на етапі компіляції визначитися з оптимальним SIMD бекендом і зробив би більш читабельним код.

З іншого боку ми втрачаємо оптимальні платформозалежні функції, такі як бітове ABI на процесорах x86. Також універсальних підхід не враховує особливості архітектури деяких центральних процесорів, наприклад: одна операція над числом типу float на архцтектурі x86 займає стільки ж часу, що й 4 таких же операції над різними компонентами вектора SSE (float vec [4]), бо в останніх моделях скалярні блоки обчислювання float були видалені в економічних цілях.

1.3. Системи рендерингу

Графіка в комп'ютерних іграх з кожним роком прагне бути все більш реалістичною шляхом підвищення точності обчислень і ускладнення алгоритмів. Для швидкої обробки цих алгоритмів існує спеціальний пристрій - відеоприскорювач, або - відеокарта. На відміну від центральних процесорів, ABI відеокарт різних виробників відрізняються один від одного. Так склалося історично і навряд чи зміниться надалі.

Для універсальності коду були розроблені графічні бібліотеки: OpenGL, Direct3D, Vulkan і т.д. Вони містять у собі багато популярних функцій, які часто використовуються у графічних додатках. API даних бібліотек суворо стандартизовано і виробники відеокарт постачають їх реалізації разом із графічними драйверами.

1.3.1. Vulkan API

У лютому 2016 року команда Khronos Group, що складається з представників відомих світових компаній, опублікувала новий API для графічних обчислень. Справа у тому, що більш ранні API, такі як OpenGL та Direct3D працювали в такий спосіб: центральний процесор підготовляє дані

для подальшого обчислення та надсилає їх у відеопам’ять, графічний прискорювач обчислює. Однак із зростанням продуктивності графічних процесорів, підготовка та надсилання даних стали займати більше часу, ніж їх обчислення.

Таким чином потрібно було розробити рішення, що дозволяє гнучко налаштовувати графічний конвеєр і управляти передачею даних між пам'яттю центрального процесора і графічного прискорювача. Цим рішенням і став API Vulkan.

1.3.2. Розширення та налагодження додатків

Vulkan спочатку замислювалося як універсальне API широкого призначення, тому там відсутні будь-які функції для відображення, налагодження та профілювання. Однак все це і навіть більше представлено у вигляді розширень рівня екземплярів чи пристроїв.

Рівень екземплярів містить загальні розширення для поточної реалізації Vulkan, наприклад для виведення налагоджувальної інформації VK_EXT_DEBUG_UTILS_EXTENSION_NAME або для відображення в буфері віконної системи VK_KHR_WIN32_SURFACE_EXTENSION_NAME.

Розширення рівня пристрою є більш вузькоспеціалізованими для конкретного графічного прискорювача, наприклад, апаратне прискорення трасування променів VK_KHR_RAY_TRACING_EXTENSION_NAME, що останнім часом дуже популярне серед розробників ігор.

Якщо Vulkan призначений для рендерингу зображення, чому ж така функція як виведення зображення не є його частиною? Справа в тому, що Vulkan використовується виключно для рендерингу, а точкою виведення може бути не обов'язково дисплей. Наприклад, при хмарному геймінгу сервера не мають моніторів, відповідно позверхньою для виводу є просто потік відео, який відправляється по мережі гравцю.

1.3.3. Графічні пристрої

На сьогоднішній день існує чимало графічних прискорювачів. Виробники пристроїв використовують різні підходи в організації обчислювального конвеєра і відмінності в архітектурі можуть бути не тільки між двома виробниками відеокарт, а між двома поколіннями відеокарт одного виробника. До того ж, багато функцій відеокарт так і не вдалося стандартизувати під API Vulkan і доступ до них організований у вигляді розширень пристроїв. Більш старі графічні API, такі як OpenGL та Direct3D надавали єдиний загальний інтерфейс спілкування програми та графічної карти. Очевидно, що такий підхід був простішим для розробників, проте менш ефективним для відеокарт. У розділі 3.2 зазначено, що Vulkan API надає можливість одночасного використання кількох графічних прискорювачів. І для цього не обов'язково мати відеокарти однієї архітектури та одного виробника, як це було в API OpenGL та Direct3D. Також підтримка технологій SLI та CrossFier не є обов'язковою, проте вони надають швидкий інтерфейс для спілкування відеокарт, тому їх підтримка зайвою не буде.

1.3.4. Черги графічного пристрою

Механізмом надсилання команд на графічний прискорювач є черги. Черга - це абстракція над апаратним механізмом надсилання команд до графічного прискорювача. Черги, які мають однакові властивості, поєднують у сімейства черг. Важливо відзначити, що кількість черг не впливає на продуктивність відеокарти, як у випадку з кількістю ядер центрального процесора, адже черги – це лише механізм надсилання команд. Якщо у сімействі міститься кілька черг, це означає, що заповнювати їх можна з різних потоків програми, не переймаючись про синхронізацію. Запис з різних потоків в одну чергу так само є безпечним, так як синхронізація за допомогою м'ютексів відбувається всередині реалізації Vulkan, однак такий спосіб програє у швидкості використанню окремих апаратних черг на кожен програмний потік.

Кожен графічний прискорювач має зазвичай одне сімейство графічних черг (VK_QUEUE_GRAPHICS_BIT), яке поєднане з обчислювальними (VK_QUEUE_COMPUTE_BIT) і DMA (VK_QUEUE_TRANSFER_BIT) чергами. На старих прискорювачах може не бути обчислювальної черги зовсім (наприклад усередині SoC BCM2835 у перших ревізіях raspberry pi). На дискретних відеокартах зазвичай існує окреме сімейство черг (VK_QUEUE_TRANSFER_BIT), що має власний DMA контролер. Це зроблено спеціально для асинхронного обміну даними по шині PCI Express, щоб не навантажувати обчислювальні та графічні черги. На вбудованих графічних прискорювачах такого сімейства черг немає зі зрозумілих причин. Вбудовані у центральний процесор відеокарти він мають прямий доступ до оперативної пам'яті і за правильної організації даних у ній, копіювати нічого не доведеться. Як можна помітити, Vulkan API надає досить низькорівневий інтерфейс взаємодії з графічним прискорювачем порівняно з OpenGL і Direct3D. Підтримку різних графічних прискорювачів повинен брати на себе розробник програми, проте це дозволяє максимально гнучко налаштувати графічний конвеєр під свої завдання і у підсумку отримати менші затримки при обробці викликів малювання та передбачувані час кадру і навантаження на центральний процесор. Однак при неправильній організації обчислювального конвеєра ми отримаємо результати навіть гірше, ніж на більш старих API.

1.3.5. Шейдери та графічний конвеєр

Сучасна відеокарта є досить складним та універсальним пристроєм. На нього покладається велика кількість різноманітних завдань. Перші графічні прискорювачі були досить простими і мали фіксований графічний конвеєр із досить простим набором функцій. Однак вимоги до графічного обладнання зростали і в якийсь момент виробники драйверів не могли догодити усім користувачам. Рішенням стала поява програмованих шейдерів. Ця

можливість дозволяла змінювати функціонал деяких частин графічного конвеєра програмним шляхом.

Очевидно, що для вирішення графічних задач не підходила жодна з наявних на той час мов програмування. Тоді світу були представлені дві нові мови:

- GLSL (відкритий стандарт Khronos)
- HLSL (закритий стандарт від Microsoft)

Обидві мови були побудовані на базі мови програмування C. Однак у них були відсутні вказівники і були додані деякі специфічні для графічного прискорювача функції. Насправді HLSL мало чим відрізняється від GLSL, хіба що перший більш орієнтований на розробників комп'ютерних ігор. Швидше за все, ліцензія GLSL не дозволила використовувати його в API Direct3D, що змусило Microsoft створювати окрему мову програмування. Якщо говорити про OpenGL і Direct3D, то вони могли працювати у двох режимах: стандартний конвеєр та програмований.

API Vulkan не надає можливості вибору. Вершинний, фрагментний, тесселяційний та геометричний шейдери можуть бути лише програмованими, інші є стандартними. Таке рішення разом із переходом на бінарний формат шейдерів SPIR-V дозволило значно зменшити розмір коду драйверів для відеокарт.

Через велику кількість різноманітних графічних прискорювачів неможливо створити універсальний компілятор шейдерів. Тому кожен виробник графічних процесорів повинен був постачати власний компілятор усередині графічного драйвера, який перетворював код шейдера на зрозумілі для конкретного графічного прискорювача інструкції. Очевидно, що компіляція шейдерів під час виконання була вузьким місцем більшості програм. Особливо це позначалося на відгуку у іграх. Щоб максимально знизити затримки, розробники ігор виділяли окремі потоки процесора під компіляцію шейдерів, а у відкритих драйверах Mesa під Linux навіть існує

опція, яка виділяє вільний потік із пулу для компіляції шейдерів. Така проблема не залишилася непоміченою архітекторами Vulkan, і вони запропонували рішення, яке успішно продемонструвало себе у компіляторі clang. Вони розробили максимально наближену до асемблера відеокарт мову, яку дуже швидко можна перекласти в інструкції, що виконується на графічному процесорі. Такою мовою став SPIR-V. Він схожий на WebAssembly та Java Dex, проте орієнтований на графічні задачі. Так само поява SPIR-V примірила два ворожі табори GLSL і HLSL розробників, так як обидві ці мови можуть бути скомпільовані в SPIR-V відповідними компіляторами.

1.3.6. Буфери команд та синхронізація

Ще одним вузьким місцем старих графічних API було послідовне заповнення буфера команд та відправлення на пристрій. Тобто графічний прискорювач був без діла, поки додаток формував необхідний набір геометрії для малювання. Тобто додатку потрібно на кожному циклі говорити відеокарті, що робити і чекати відповіді від неї. Зі збільшенням кількості вершин геометрії опис команд для відеокарт почав займати більше часу, ніж її обробка. У API Vulkan це вирішується за допомогою командних буферів. Тобто додатку достатньо одного разу заповнити його і після цього постійно відправляти цей буфер на малювання. Такий підхід також знижує затримки, пов'язані із синхронізацією при надсиланні команд графічному прискорювачу, шляхом зменшення кількості цих команд.

Для ефективного використання пам'яті при виділенні буферів команд використовуються командні пули. Це наперед виділені блоки пам'яті та розбиті на фрагменти, що підходять під виділення командних буферів. Такий підхід дозволяє як економити пам'ять при частих виділеннях і звільненнях командних буферів, так і для розділення буферів різного призначення. Наприклад, можна розділяти командний пул рендерингу та обчислень на відеокарті для паралельної генерації та виконання команд. Слід зазначити, що

паралельно вони будуть виконуватись за наявності у графічному процесорі вільних обчислювальних блоків для виконання цих команд.

Якщо підсумувати, то Vulkan API надає широкі можливості для розпаралелювання завдань, не витрачаючи ресурси на синхронізацію. Це є можливістю API, а не його вимогою. Оскільки кожен програмний потік, командний пул чи буфер потребує додаткової пам'яті, якої багато не буває. До того ж на смартфонах з архітектурою BigLittle розпаралелювання завдань може призвести до перенесення планувальником потоків програми на велику кількість малопотужних ядер замість використання пари продуктивних ядер.

1.4. Висновки

У розділі був розглянутий ігровий рушій Unreal Engine 5. Його архітектура та функціонал є гарним прикладом для тих, хто створює власний ігровий рушій. Від надає можливість покласти усі зусилля на створення гри, а не на адаптацію її під безліч графічних карт.

Також у розділі було приділено багато уваги архітектурі та API сучасних графічних карт. Актуальним відкритим API на теперішній день є Vulkan, тому він був обраний як оновний під час розробки ігрового рушія. Усі подальші приклади коду будуть побудовані на базі цього API.

Не менше увари будо приділено і прискоренню математичних обчислень, що є важливим для будь якого ігрового рушія. Після теоретичного дослідження та проведення практичних експериментів було виділено три основних методи прискорення:

1 Використання власних статичних математичних функцій замість стандартних динамічних;

2 Використання апаратного прискорення векторних обчислення замість циклів для обчислення кожного компоненту вектора окремо.

РОЗДІЛ 2. АЛГОРИТМІЧНИЙ

2.1. Глобальні модулі

Глобальні модулі у проекті GameEngine існують для абстракції системних викликів з інших модулів проекту. Вони надають більш високорівневі функції до роботи з системними ресурсами. Розглянемо ці модулі докладніше.

2.1.1. Модуль обробки помилок

Під час роботи програми можуть виникати помилки і програма повинна вміти їх обробляти.

Некритичні помилки лише записуються в лог і додаток продовжує працювати у звичайному режимі. Прикладом некритичних помилок можуть бути налаштування графіки, деякі з яких можуть бути недоступні на певному обладнанні або операційній системі. Таким чином, GameEngine обирає відповідний параметр серед доступних і логує попередження.

Критичні помилки не дозволяють додатку продовжити роботу і негайно його завершують зі звільненням усіх системних ресурсів. Прикладом критичних помилок є некоректна робота віконної системи, графічного обладнання, нестача оперативної пам'яті або відсутність прав доступу до ресурсів. Також модуль обробки помилок перехоплює системні сигнали, найпопулярніші з яких SIGSEGV (помилка сегментації) або SIGFPE (некоректна арифметична операція).

При ініціалізації модуль обробки помилок реєструє функцію звільнення системних ресурсів, у разі критичної помилки викликає її (рис. 2.1).

Рис. 2.1. Єдиний механізм звільнення ресурсів системи

2.1.2. Модуль логування

Модуль логування також є важливою частиною будь-якої програми. Він відповідає за виведення та збереження інформації про роботу програми. Логічне питання, чому не використовувати стандартну функцію fprintf?

По суті так і є, модуль логування це обгортка над fprintf, яка додає підтримку:

1. Рівнів логування:

- RK_LOG_LEVEL_ERROR – завжди увімкнений;
- RK_LOG_LEVEL_WARNING – за умовчанням увімкнений;
- RK_LOG_LEVEL_INFO – за замовчуванням вимкнений;
- RK_LOG_LEVEL_DEBUG – за замовчуванням вимкнений;

Є можливість увімкнути та вимкнути логування повідомлень усіх рівнів, крім RK_LOG_LEVEL_ERROR.

2. Кольорового висновку:

Деякі консолі мають можливість виводити кольоровий текст, який легко сприймати візуально під час читання логів. Кольори кожного рівня логування задаються в файлі platforms/global/source/logger.c. Вимкнути цю функцію можна консольною командою.

3. Підтримка тегів:

Дозволяє вказувати один або кілька тегів повідомлення для простоти пошуку повідомлення по логах.

Підбиваючи підсумки, будь-які з цих функцій реалізуються парою строк коду, однак при таких невеликих вкладаннях ми отримуємо значно зручніше для використання API логування в порівнянні зі стандартним fprintf (рис. 2.2). Слід відзначити, що пункти 1 та 4 може закрити стандартна функція syslog в операційних системах Linux, однак в угоду більшої універсальності від неї не довелось відмовитися.

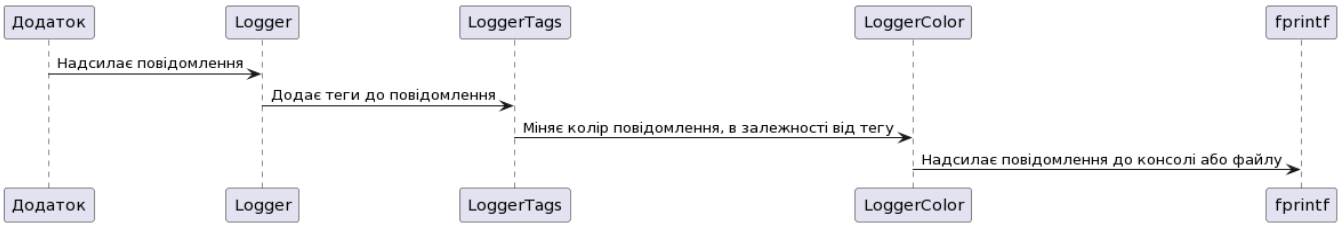


Рис 2.2. Алгоритм роботи модуля логування

2.1.3. Модуль парсингу параметрів командного рядка

Параметри командного рядка - це один із небагатьох інструментів, що дозволяють динамічно керувати параметрами роботи програми. Отже обробка цих аргументів – одна з найважливіших частин програми. Існує безліч підходів та реалізацій парсерів для цих задач, проте ми зупинимося на стандартному: getopt_long**.** Щоб уникнути колізій та непорозуміння між розробником та користувачем, прийнято рішення не використовувати однолітерні опції. Напроти було віддано перевагу довгим опціям, в яких явно в самій назві вказується, що вона робить.

Опції були розбиті на типи:

- RK_OPT_TYPE_HELP – не має параметрів, виводить список доступних команд;
- RK_OPT_TYPE_SELECT – дає вибір серед доступних варіантів;
- RK_OPT_TYPE_NUMBER – приймає число в певному діапазоні;
- RK_OPT_TYPE_STRING – рядок з певною мінімальною та максимальною довжиною.

Взаємодія з опціями реалізована через parser та helper:

- parser – отримує значення з параметра з командного рядка;

- helper – виводить опис параметра після команди --help.

Існують універсальні абстракції parser та helper для всіх типів опцій:

- typedef void (* RkOptParser)(const RkOpt * const restrict opt, const char * const arg);
- typedef void (* RkOptHelper) (const RkOpt * const restrict opt);

Відповідно є масиви, де індексом є номер типу аргументу, а значенням – обернені виклики parser чи helper. Реалізація обробника параметрів командного рядка (рис 2.3) знаходиться у файлі platforms/global/source/options.c

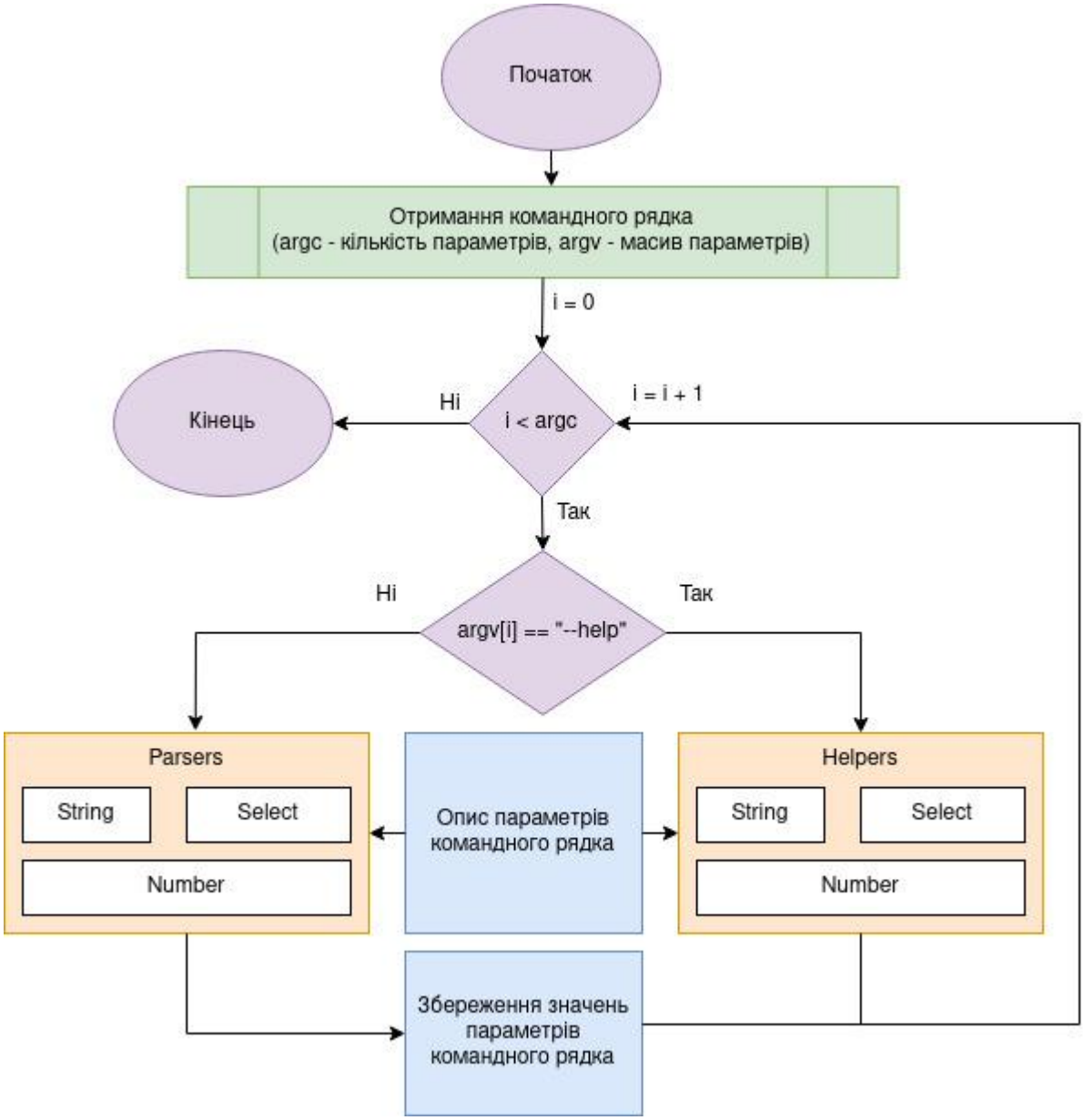


Рис 2.3. Алгоритм обробки параметрів командного рядка

2.1.4. Модуль скалярної математики

Ігрові рушії мають виконувати складні математичні обчислювання досить швидко, тому базові математичні функції мають бути якомога більше оптимізованими та вбудованими усередину алгоритму. Серед оптимізацій можна виділити наступні дії:

- Зменшення діапазону вхідних значень (має сенс, якщо реалізовувати деякі функції через розкладання у ряд Тейлора)
- Видалення додаткових перевірок (стандартна бібліотека Сі має додаткові перевірки, щоб запобігти виникненню критичної помилки, наприклад ділення на 0)

Розглянемо різницю між власними вбудованими функціями, та викликом функції із стандартної бібліотеки Сі на прикладі функції sqrt (підрахунок квадратного кореня) (рис. 2.4). Реалізацію усіх вбудованих математичних функцій можете подивитись у файлі platforms/global/source/simple-math.c

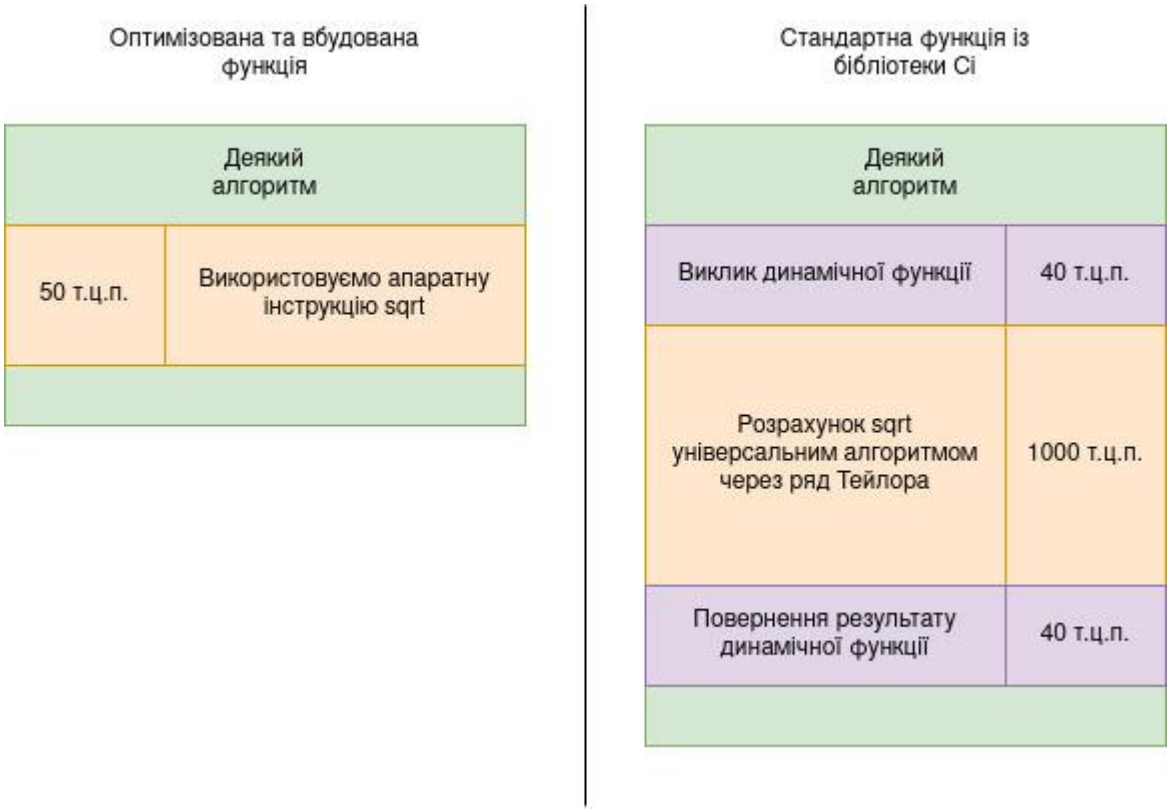


Рис. 2.4. Різниця між швидкістю виконання вбудованої функції sqrt у порівнянні з динамічною функцією sqrt із стандартної бібліотеки Сі (т.ц.п. - такт центрального процесору)

2.1.5. Модуль векторної математики

У алгоритмах комп'ютерної графіки більшість обчислень є векторними, тобто які виконують операції над векторами та більш складними структурами, такими як матриці.

Ці функції використовуються часто, тому мають бути оптимізованими, під наявні у центральному процесорі SIMD інструкції (детальніше у розділі 1.2.2). У GameEngine існує відповідний модуль, який реалізує базові операції над векторами та матрицями оптимальним для певної архітектури ЦП чином.

Був запропонований наступний компроміс: для процесорів x86_64 реалізація векторної математики використовує набори SIMD інструкцій SSE. Для решти архітектур, включаючи архітектуру ARM і всі її різноманітні варіації, використовується автовекторизація компілятора (рис 2.5).

Суб'єктивно кажучи, це рішення не виглядає найкращим, оскільки апаратні функції x86 незабаром застаріють, а автовекторизація не завжди може розкрити весь потенціал інших архітектур. Однак для невеликого проекту без тривалої підтримки такий метод чи не єдиний оптимальний за співвідношенням зусилля/результат. Переглянути реалізацію векторної математики можете у файлі platforms/global/source/vector-math.c. Реалізації апаратно-залежних функцій знаходяться у файлах platforms/global/source/vector-math-x86.c та platforms/global/source/vector-math-arm.c.

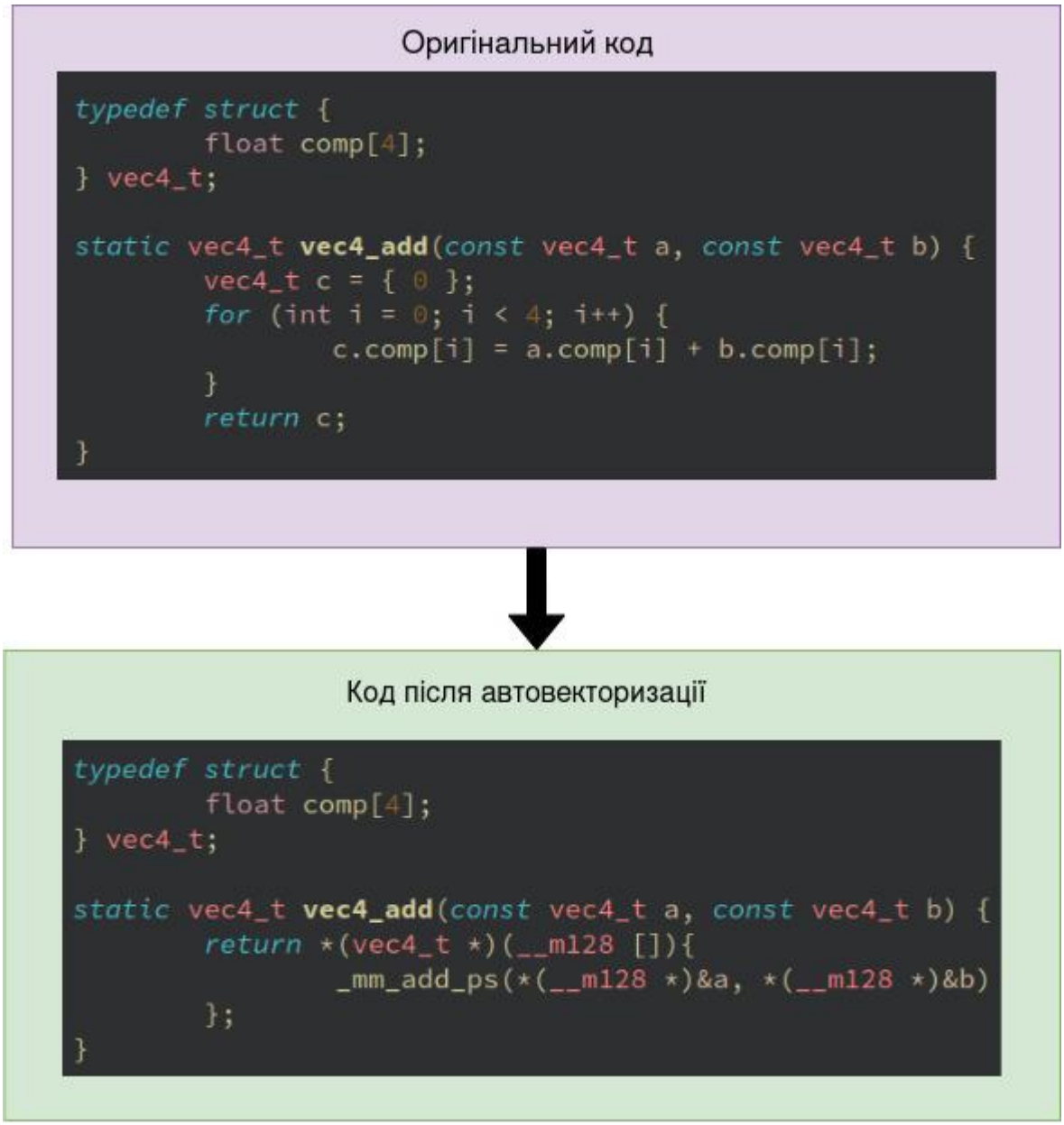


Рис 2.5. Приклад автовекторизації коду функції

2.1.6. Модуль unit тестування

Тема покриття коду unit тестами досить суперечна. У деяких випадках – це приносить свої плоди, а в інших є марною тратою часу. Якщо повністю ігнорувати unit тести, то на налагодження великого проекту може піти чимало часу. Якщо ж покривати тестами весь код, можна зустрітися з ситуацією, коли код пишеться під тести, а не тести під код.

У рамках проекту GameEngine було вирішено покрити тестами лише математичний модуль. Інші модулі є дуже гнучкими і неможливо

передбачити всі кейси використання. Під час написання тестів використовувалися наступні принципи тестування:

- якщо в результаті виконання двох абсолютно різних реалізацій однієї і тієї ж задачі ми отримали один і той же результат, то можна впевненіше казати, що обидві реалізації є вірними;
- якщо при однакових вхідних даних результати представлених функцій збігаються з результатами аналогічних перевірених функцій, то представлені функції також можна вважати перевіреними, за умови, що набір вхідних даних обраний випадковим чином.

Тестування модулів відбувається під час компіляції для персонального комп'ютера, і через емулятор qemu якщо відбувається кросскомпіляція під Android (на даний момент тільки при кросскомпіляції на Linux). Реалізація модуля unit тестування знаходиться у файлі platforms/global/test/tests.c Тести математичного модуля знаходяться у файлі platforms/global/test/math-tests.c. Алгоритм роботи модуля unit тестування зображений на рис. 2.6.

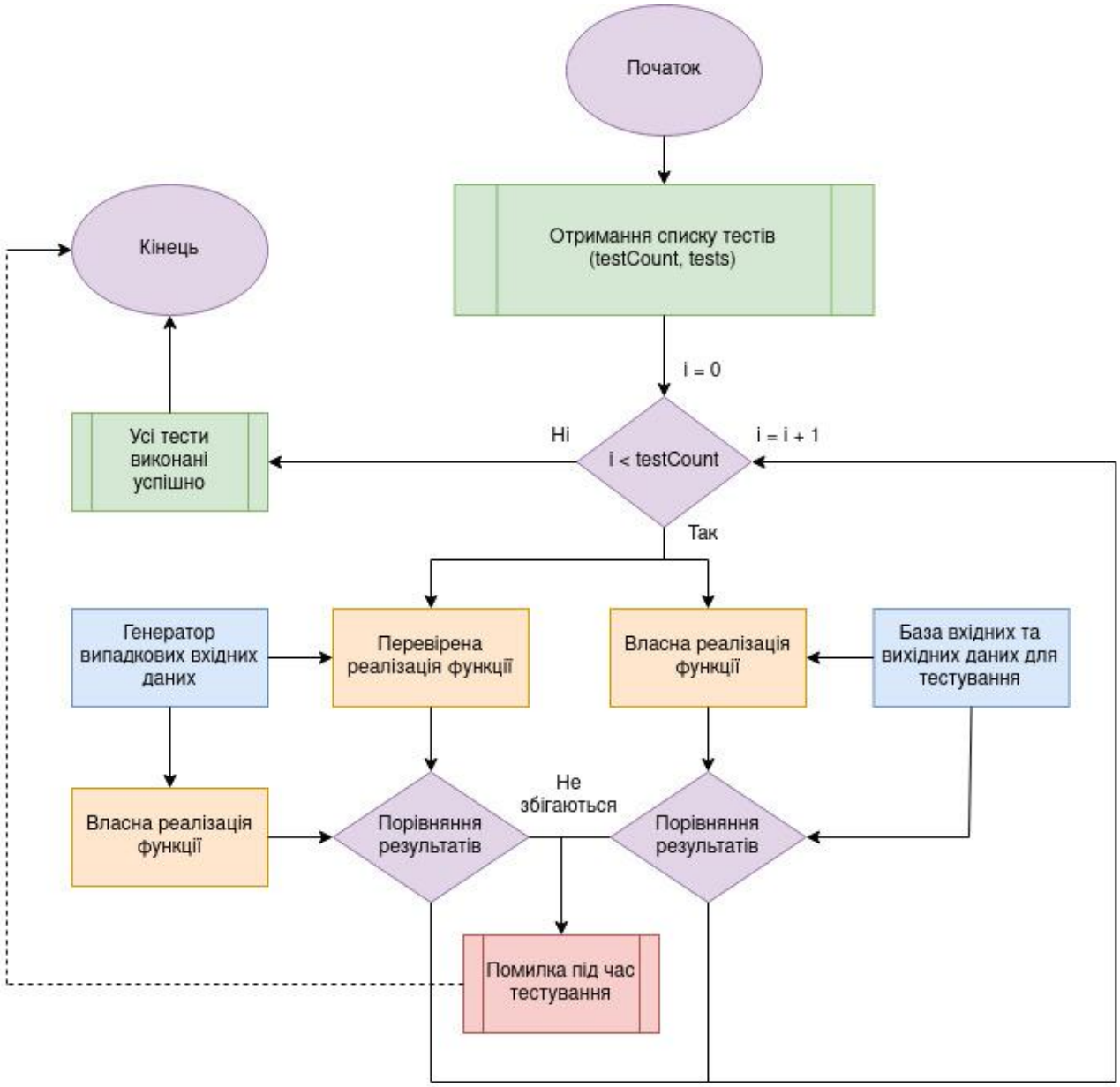


Рис. 2.6. Алгоритм модуля unit тестування

2.2. Модуль рендерингу

У ігрових рушіях найбільшу увагу приділяють графічний підсистемі. Це дуже складний модуль який має брати на себе наступні задачі:

- роботу з графічним обладнанням
- розподіл відеопам’яті (якщо цим не займається драйвер)
- планування та групування графічних команд (якщо цим не займається драйвер)
- оптимізація ігрової сцени (видалення скритих геометричних вершин)

Через складність цього модуля у прирівнянні з іншими, у ігровому рушії GameEngine файли модуля рендерингу знаходяться окремо від файлів інших модулів, для простішої орієнтації у коді.

API модуля рендерингу є абстрагованим від графічного бекенду, тобто можна обрати будь яке графічне API для рендерингу без необхідності перероблювати інші модулі ігрового рушія. На даний момент підтримується лише API Vulkan, але закладена можливість використання OpenGL та Direct3D.

2.2.1. Статичне та динамічне лінкіування Vulkan

Оскільки Vulkan є динамічною бібліотекою, існує кілька способів інтеграції їх у проект. Основною відмінністю Vulkan від інших графічних API є можливість використовувати кілька графічних прискорювачів для різних завдань одночасно. До того ж вони можуть бути навіть від різних виробників. Уявимо, що у нас є персональний комп'ютер із вбудованою графікою та дискретним графічним прискорювачем.

Більш старі API могли використовувати лише один із графічних прискорювачів і лише той, який виводить зображення на монітор. Vulkan натомість дозволяє, наприклад, на дискретній відеокарті обробляти зображення, а на вбудованій графіці виконувати фізичні розрахунки, тим самим розвантажуючи центральний процесор. Варто відзначити, що можливість комбінації різних реалізацій Vulkan є не в усіх операційних системах. Серед відомих - тільки GNU/Linux з відкритим стеком графічних драйверів Mesa3D.

Однак у Android та Linux існує спосіб власноруч підключити динамічну бібліотеку Vulkan під один або декілька GPU і використовувати незалежно один від одного. Такий спосіб ще цікавий тим, що з'являється можливість перемикання між графічними API без перекомпіляції проекту. У Linux/Android динамічно завантажити бібліотеку можна наступним чином (рис. 2.7).

```
void * const vk_lib = dlopen("/path/to/vulkan_xxx.so", RTLD_LOCAL);
const PFN_vkGetInstanceProcAddr vkGetInstanceProcAddr = dlsym(vk_lib, "vkGetInstanceProcAddr");
// Використовуємо  vkGetInstanceProcAddr для отримання інших функцій Vulkan
const PFN_vkCreateInstance vkCreateInstance =
        (PFN_vkCreateInstance)vkGetInstanceProcAddr(VK_NULL_HANDLE, "vkCreateInstance");
// ...
// Закриваємо бібліотеку, коли закінчили працювати з графічним пристроєм
dlclose(vk_lib);
```

Рис 2.7. Приклад динамічного лінкування бібліотеки Vulkan

(xxx - це виробник відеоприскорювача: intel, amd, adreno...)

На даний момент реалізація GameEngine не підтримує динамічне завантаження бібліотеки Vulkan, щоб не ускладнювати реалізацію. Однак така можливість архітектурно закладена і при необхідності може бути легко інтегрована. Розглянемо файл renderers/vulkan/source/loader.c, який містить функції vkCreateDebugUtilsMessengerEXT та vkDestroyDebugUtilsMessengerEXT. Це функції призначені для пошуку помилок у використанні Vulkan API і за умовчанням вони відсутні у реалізації Vulkan. Однак, увімкнувши необхідне розширення, і отримавши адрес функції у файлі ми можемо використовувати їх ніби вони знаходяться всередині бібліотеки Vulkan. Додавши аналогічний прошарок для інших функцій, ми можемо використовувати відеоприскорювачі різних виробників в обхід усіх обмежень, які накладає динамічне лінкування.

Для тестування окремо від проєкту GameEngine був динамічно підвантажений драйвер vulkan_intel.so (GNU/Linux 6.0.2, Mesa 23-devel, Wayland Session) і при малюванні трикутника на дисплеї використовувалося всього 20 МБ оперативної пам'яті (згідно з системним моніторингом htop), а при тих же параметрах системи , але з лінкуванням на етапі компіляції використовувалося 105 МБ пам'яті + реалізацією Vulkan було створено 2 додаткові потоки, які більшу частину часу перебували в замороженому стані. Як видно, звернення безпосередньо в драйвер, на деяких реалізаціях дає вагомі плюси, проте для драйверів іншого виробника або іншій версії драйвера від intel може бути все навпаки, тому розробники Mesa3D рекомендують лінкувати бібліотеку Vulkan природним шляхом.

2.2.2. Поверхні малювання для Vulkan

Vulkan є платформонезалежним API. У це поняття вкладається те, що він має однакові параметри та поведінку функцій на різних операційних системах та архітектурах центрального процесора. Кожна операційна система з графічним інтерфейсом може мати власну віконну систему, що відрізняється від інших. Деякі з них можуть навіть мати суцільно різні підходи до відтворення графіки. Ось наприклад, віконні системи Windowing System (в ОС Windows) і X.Org (у старих дистрибутивах Linux) оперують вікнами, у той час як Wayland (у нових дистрибутивах Linux) та SurfaceFlinger (в ОС Android) оперують поверхнями малювання. У Vulkan абстракцією поверхні малювання над усіма віконними системами є VkSurface. Його створення є платформою залежною функцією і в рамках проекту GameEngine знаходиться у файлах:

- platforms/android/source/window.c
- platforms/linux/source/window_xcb.c
- platforms/linux/source/window_wayland.c

На платформах Android та Windows створення VkSurface реалізовано в рамках функції vkCreateSurfaceWrap. Залежно від вибраної платформи компілюється відповідний файл з цією функцією. Однак для операційної системи Linux існує два можливі способи створення поверхні малювання:

- vkXcbCreateSurfaceWrap
- vkWaylandCreateSurfaceWrap

Вибір між ними здійснюється за допомогою параметру командного рядка “--wsi=<xcb/wayland>”. За замовчуванням використовується новий віконний менеджер Wayland, проте за бажанням можна використовувати більш старий XCB. Оскільки невідомо на етапі компіляції, яка віконна система використовуватиметься під час роботи, виклики віконних функцій реалізовані через колбеки, які задаються етапі запуску (рис. 2.8).

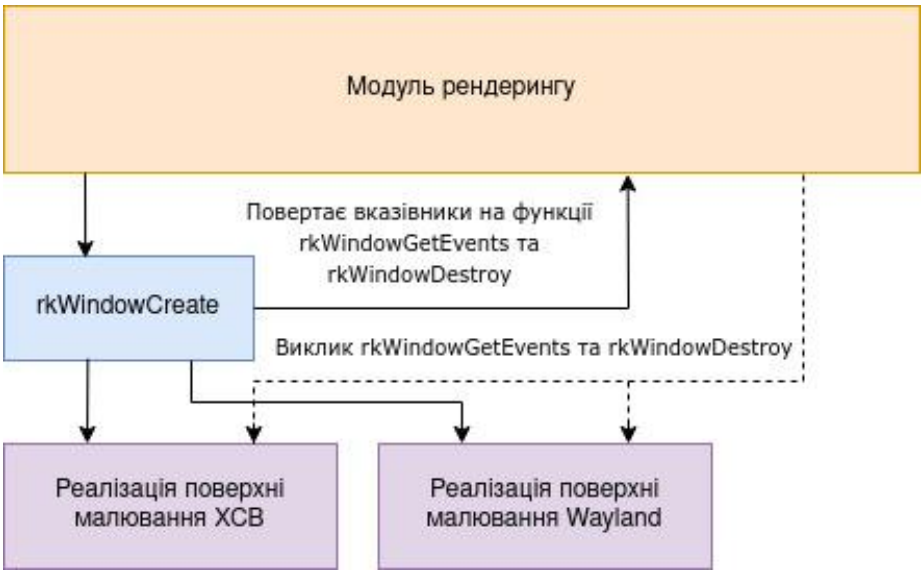


Рис 2.8. Абстракція поверхні малювання від модуля рендерингу

2.2.3. Вибір графічного пристрою

Розглянемо загальні правила вибору графічних прискорювачів із доступних у системі:

1 Графічним прискорювачем для рендерингу має бути той, що має прямий доступ до поверхні малювання. Як правило, це прискорювач, до якого підключений монітор. Також він повинен мати хоча б одну чергу рендерингу (VK_QUEUE_GRAPHICS_BIT);

2 Прискорювач обчислень повинен мати прямий доступ до пам'яті центрального процесора або центральний процесор повинен мати прямий доступ до пам'яті прискорювача обчислень. Це може бути вбудована в процесор графічна карта або дискретна, яка має видиму для процесора локальну пам'ять (як відеокарти AMD). Пам'ять графічного прискорювача повинна містити прапори: VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT та VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT і пристрій повинен мати хоча б одну обчислювальну чергу (VK_QUEUE_COMPUTE_BIT);

3 При виборі графічних прискорювачів Vulkan не надає інформації про їх продуктивність, проте можна спиратися на таке правило: спочатку шукаємо відповідну критеріям дискретну графічну карту, якщо не вдалося знайти, то шукаємо відповідну вбудовану у центральний процесор графічну карту, інакше вибираємо CPU як пристрій для обчислень та рендерингу. Пріоритети можна записати таким чином:

3.1VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU
3.2VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU
3.3VK_PHYSICAL_DEVICE_TYPE_CPU

Алгоритм вибору пристрою для рендерингу зображений на рис. 2.9. Подивитись його реалізацію можна у файлі renderers/vulkan/source/physical_device.c.

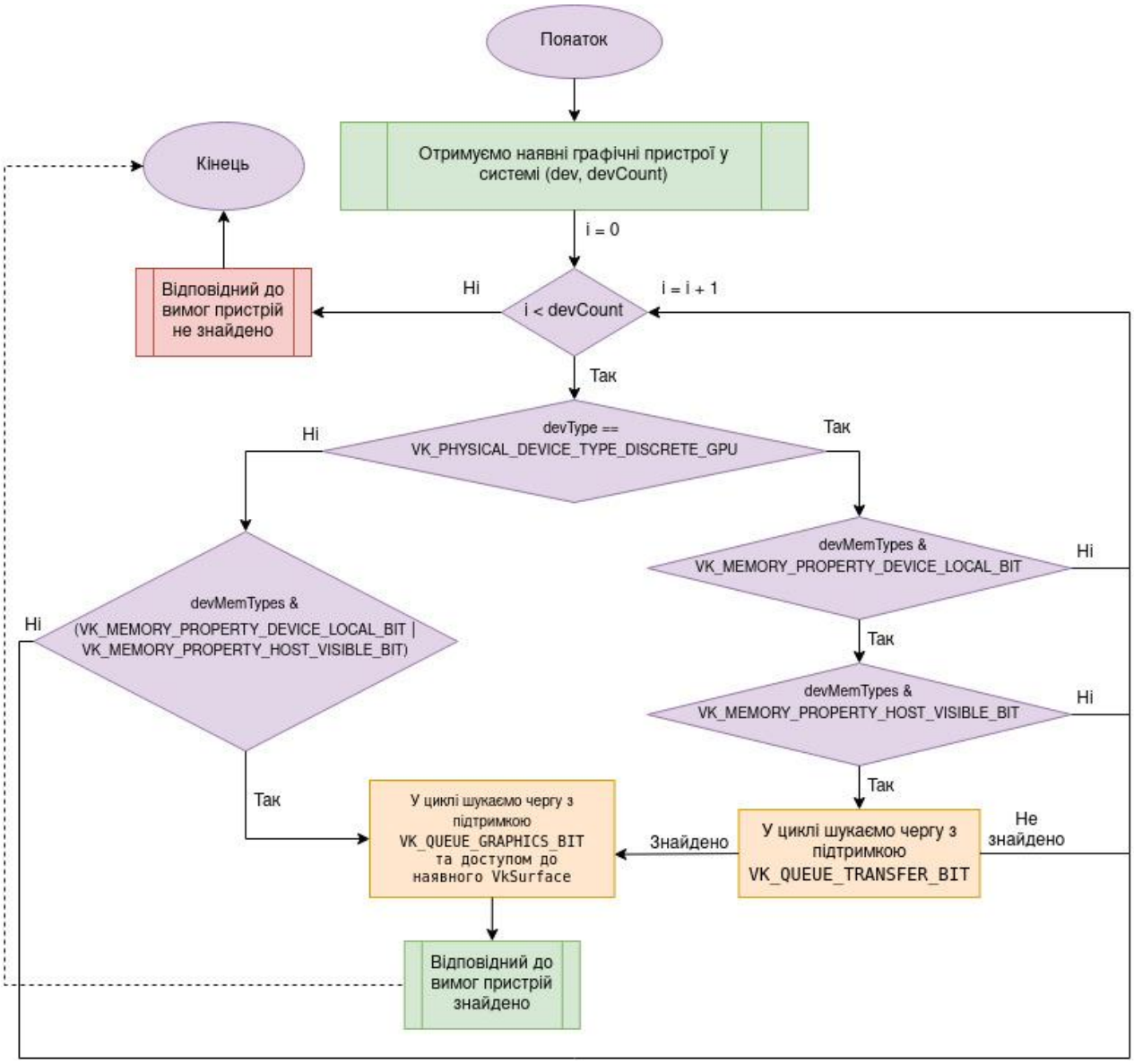


Рис. 2.9. Алгоритм вибору графічного пристрою для рендерингу

2.2.4. Компіляція шейдерів

У рамках проекту GameEngine для написання шейдерів було обрано мову GLSL версії 4.5 core. Це мінімальна версія, яку підтримує API Vulkan. На поточній стадії розвитку ігрового рушія немає необхідності використовувати чогось більш просунутого. Перейти на новішу можна просто змінивши перший рядок у файлі шейдера: #version 450 core.

Код шейдерів на етапі компіляції переводиться із мови програмування GLSL у SPIR-V (рис 2.10 та 2.11) та завантажується у файлі renderers/vulkan/source/shader.c. Компіляція шейдерів у інструкції для

графічного процесору відбувається при формуванні графічного конвертера у файлі renderers/vulkan/source/pipeline.c.

main.vert.glsl

```
#version 450 core

layout (location = 0) in vec3 inPosition;
layout (location = 1) in vec3 inColor;
layout (location = 0) out vec3 outColor;
layout (push_constant) uniform constants {
    mat4 view;
} inConstants;

void main(void) {
    gl_Position = inConstants.view * vec4(inPosition, 1.0);
    outColor = inColor;
}
```

Рис 2.10. Код вершинного шейдера на мові програмування GLSL

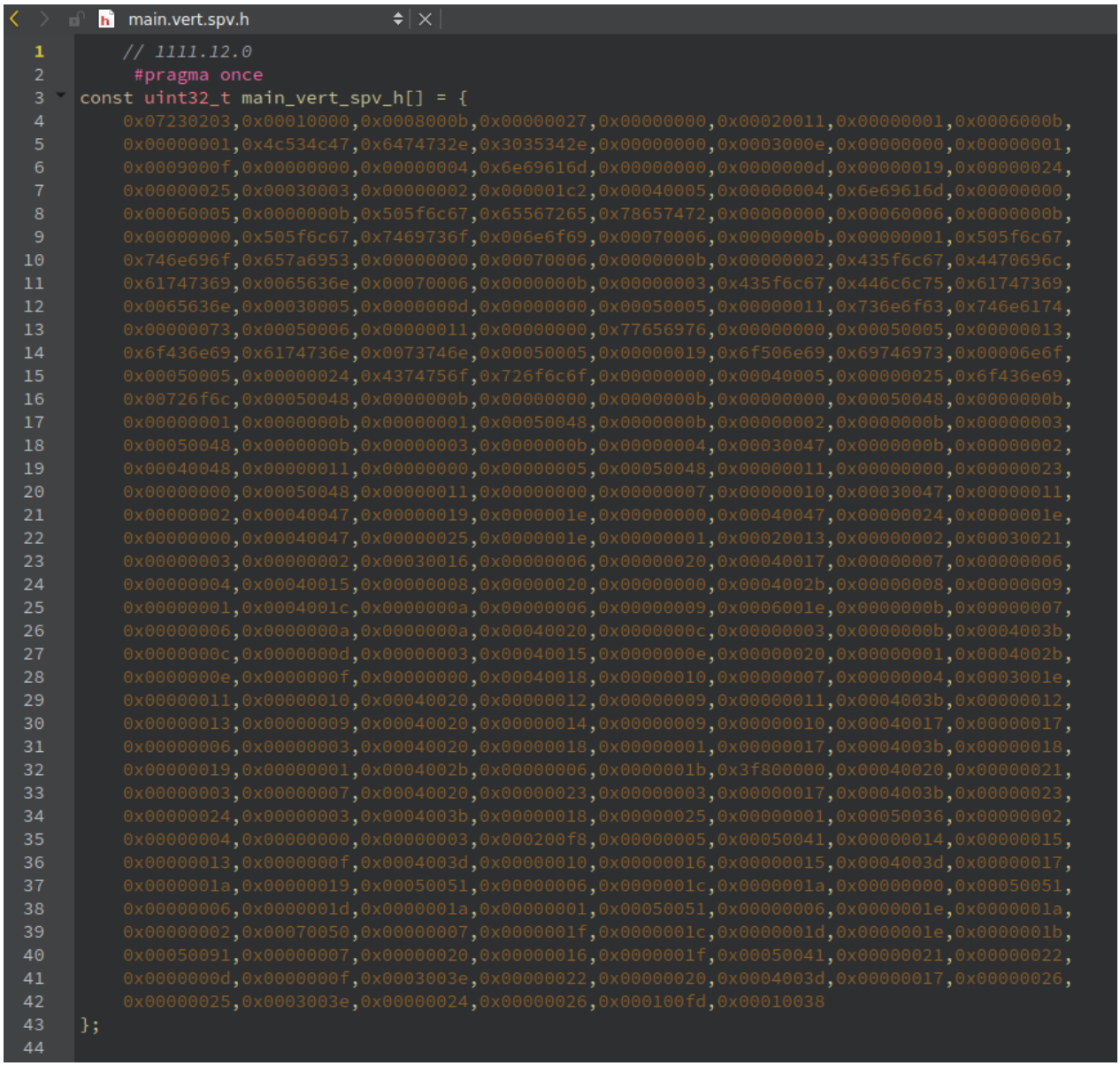


Рис 2.11. Вершинний шейдер у форматі SPIR-V

На теперішній час шейдери у проєкті GameEngine є максимально простими і не здатні взаємодіяти з матеріалами та системами частинок із програм 3D моделювання, таких як Blender. Проте технічно така можливість закладена.

2.3. Висновки

У даному розділі описана архітектура ігрового рушія GameEngine. Рушій складається з незалежних один від одного модулів. API усіх модулів складається з абстракцій високого рівня, тому кожна з підсистем рушія володіє лише тою інформацією, що їй має бути доступна. Представлена архітектура є платформонезалежною та може використовуватись у

різноманітних проєктах. Глобальні модулі ігрового рушія можуть бути використані окремо від нього у інших проєктах, без необхідності модифікації їх коду.

Представлений модуль рендерингу може відображати та рухати 3D об’єкти за допомогою функцій переміщення, обертання та масштабування по усім трьом осям.

Під розробки не приділялось уваги моделюванню фізичних законів, механік взаємодії гравця зі світом, штучного інтелекту NPC. У рамках даної роботи була досліджена графічна підсистема ігрових рушіїв та модулі, які необхідні для її роботи.

РОЗДІЛ 3. ПРОГРАМНИЙ

3.1. Тестування ігрового рушія

Сенс тестування полягає у дослідженні роботи ігрового рушія на платформах Windows, Linux та Android. У якості 3D сцени був обраний різнокольоровий куб. Щоб сцена не виглядала статичною для куба на кожному кадрі створюється матриця перетворення (рис. 3.1). Код цих функцій знаходиться у модулі векторної математики.

```
27
28
29      mat4x4_t view;
30      rkMathInitIdentityMat4x4(&view);
31      rkMathRotateX(&view, (float)(frameIndex) / 30.f);
32      rkMathRotateY(&view, (float)(frameIndex) / 40.f);
33      rkMathRotateZ(&view, (float)(frameIndex) / 50.f);
34
35
```

Рис. 3.1. Створення матриці перетворення для обертання куба (frameIndex — порядковий номер кадру)

Також під час тестування перевірялись:

- здатність роботи з різними віконними системами без перекомпіляції рушія
- можливість роботи у повноекранному режимі
- можливість міняти рівні буферизації кадрів

Тестування на відео проводилось в операційній системі Linux Ubuntu 20.04 LTS

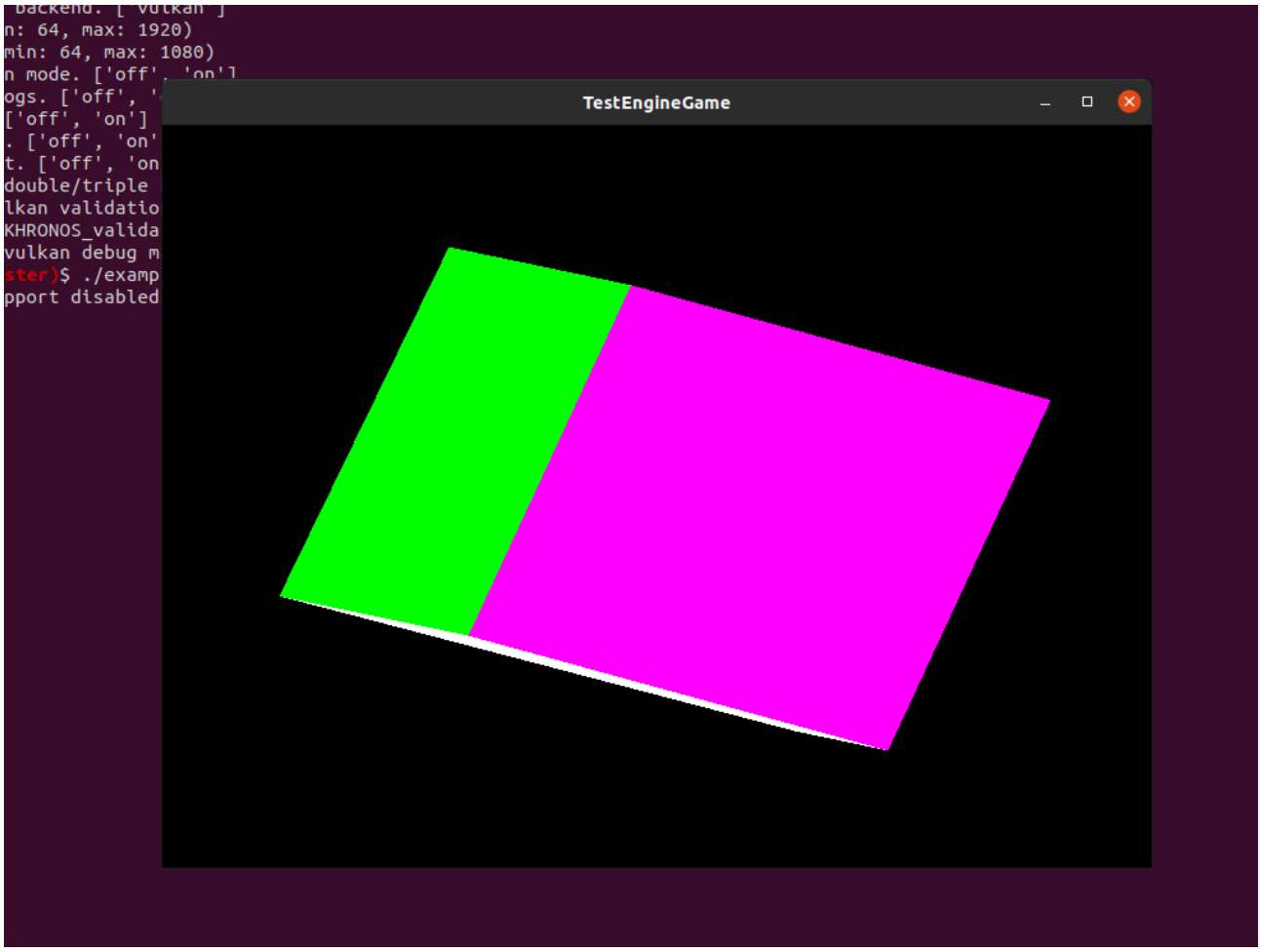


Рис. 3.2. Тестова сцена ігрового рушія “Різнокольоровий куб”

3.2. Завантаження 3D сцени у форматі GLTF

Для створення ігрових моделей існують спеціальні 3D редактори. Серед відомих можна виділити Blender та 3D Max. У них є можливість експортувати створені моделі у певний формат. Але, на жаль, ми не можемо просто завантажити файл 3D моделі у відеопам’ять і чекати, доки модель відобразиться на екрані. Тому для перетворення файлу 3D моделі у масив індексів та вершиш існують спеціальні бібліотеки.

TinyGLTF — це бібліотека для завантаження та інтерпретації 3D моделей у форматі GLTF. Для тестування була обрана саме ця бібліотека через мінімальну кількість залежностей. Також з інтернету була завантажена 3D модель для тестування. Ця модель розповсюджується за безкоштовною ліцензією. Результат ви можете бачити на рис. 3.3.

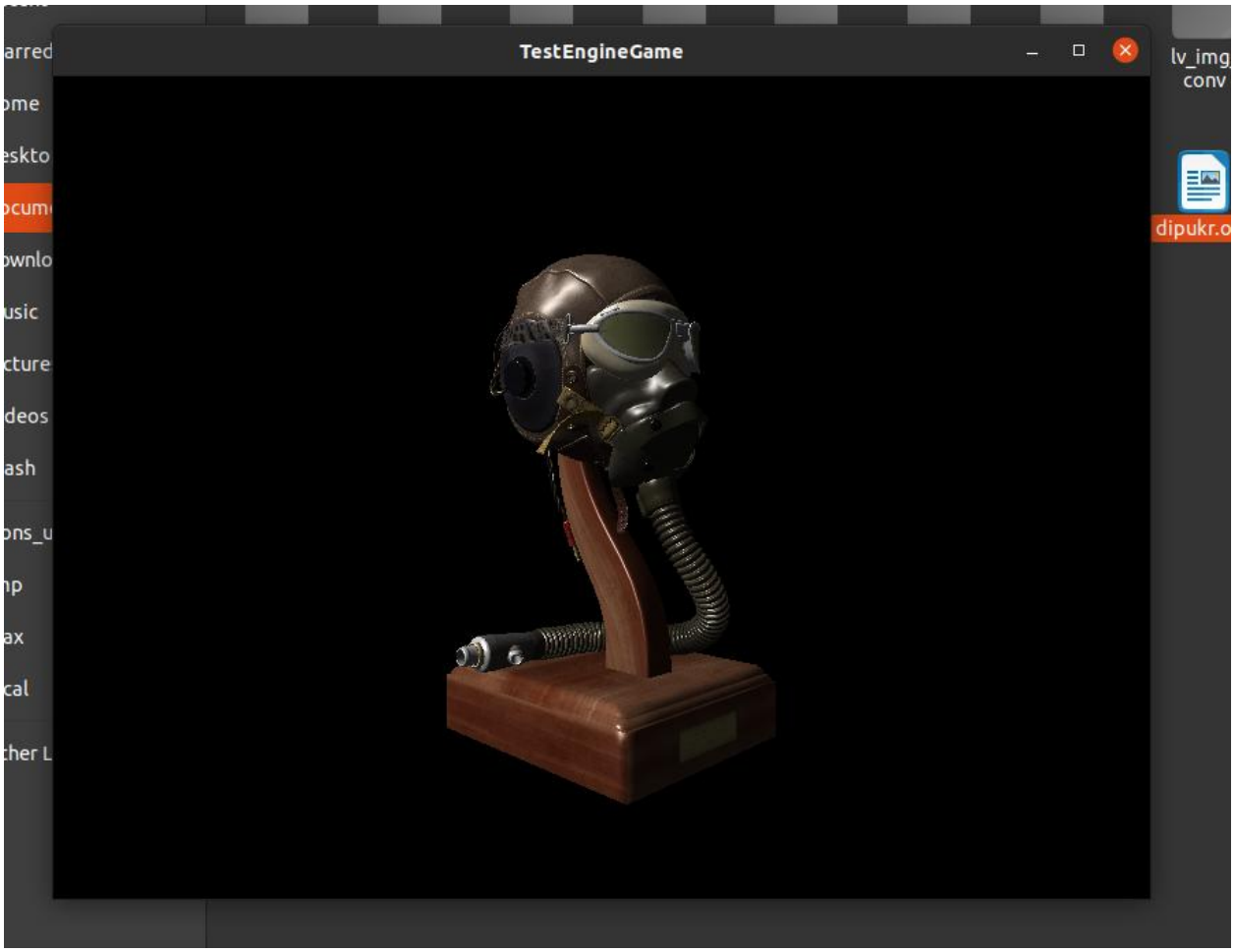


Рис 3.3. Завантаження 3D моделі

3.3. Застосування отриманих знань

Існує такий стереотип, що операційні системи Windows краще підходять для ігор, MacOS призначена для роботи, а Linux незрозуміло для чого взагалі існує. Це й не дивно, адже Linux, на відміну від інших, не має реклами і величезного штату маркетологів.. А вектор розвитку системи задають не великі компанії, а прості користувачі. Більшість коду, написаного під Linux є відкритим і ми можемо самостійно розширювати систему або відкидати непотрібні функції. А з повновісною інтеграцією системи складання meson, для цього навіть не потрібно бути програмістом, але розбиратися в структурі Linux все ж таки варто. Однак за нас це роблять інші люди, а потім із отриманих додатків формують дистрибутиви.

Одним з таких є Ubuntu, що має низький поріг входу для більшості користувачів. Аналогічно думала і компанія Valve, що створила Steam. Їх

бізнес побудований на іграх і донедавна був щільно зав'язаний на Windows. Як тільки компанія почала набирати обертів, Microsoft це помітила та почала вимагати грошових відрахувань. Отримавши відмову від Vlave, вона потихеньку почала вставляти ціпки в колеса проекту Steam і тоді Vlalve заявила про перенесення Steam на Linux.

Ми підтримуємо цю ідею і тому у рамках дипломного проекту пропонуємо набір патчів для таких ігор як:

- Minecraft – патч дозволяє запускати цю гру з-під графічного сервера Wayland і виправить баг подвійного прокручування коліщатка миші при виборі предмета в інвентарі
- Genshin Impcat – патч дозволяє запускати гру з-під графічного сервера Wayland та виправляє баг із білою формою авторизації. Також портована система античит з Windows на Linux. (поточна портована версія є актуальною для версії гри 3.6)

3.4. Висновки

У даному розділі був описаний процес тестування власного ігрового рушія, а також приклад застосування отриманих знань для запуску наявних Windows ігор на Linux.

Для тестування відображення 3D моделей була використана бібліотека TiniGLTF. Варто зазначити, що вона не є частиною ігрового рушія. Це було зроблено, щоб розробник сам міг обрати 3D формат для зберігання своїх моделей.

Також TiniGLTF завантажує данні 3D моделі значно повільніше ніж аналогічні завантажувачі із Unreal Engine 5. Для підвищення швидкості завантаження, треба використовувати власний формат, який буде оптимізований під певний ігровий рушій, але у рамках даної роботи такі задачі не ставилися.

ВИСНОВКИ

У ході роботи було розроблено простий платформонезалежний ігровий рушій GameEngine. На даний момент його використання в комерційних проектах не має сенсу, однак він стане у нагоді під час навчання. Код цього рушія можна використовувати як приклад реалізації алгоритмів, описаних у списку літератури до дипломного проекту.

Якщо вашою метою є створення ігор, то найкраще для цього використовувати Unreal Engine 5. Він також є платформонезалежним і має достатньо супровідної документації та відеоуроків. Ми у рамках роботи знайшли більш корисне застосування здобутим знанням, ніж створення чергового ігрового двигуна. Це патчі для коректної роботи найпопулярніших ігор під Linux. Для тих, хто вже використовує Linux – це буде приємним бонусом, а комусь може цього і не вистачало для повноцінного переходу на нову для себе операційну систему.

Подальші плани щодо розвитку:

- Тестування Windows ігор та програм під Linux, при необхідності випуск відповідних патчів з виправленнями;
- Тестування Linux додатків, які відсутні в репозиторіях дистрибутивів, при необхідності випуск відповідних патчів з виправленнями;
- розвиток проектів суспільства
  - https://github.com/varmd/wine-wayland
  - https://github.com/doitsujin/dxvk

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1 Graham Sellers and John Kessenich, Vulkan Programming Guide, 2017p., 394 ст.

2 Kaiwan N Billimoria, Linux Kernel Programming, 2021p., 754 ст.

3 Jason Gregory, Game Engine Architecture, 2021p., 1136 ст.

4 David Wolff, OpenGL 4 Shading Language Cookbook, 2018 p., 472 ст.

5 Vulkan-Samples [Електронний ресурс] - Режим доступу: https://github.com/KhronosGroup/Vulkan-Samples

6 Vulkan driver for Raspberry Pi [Електронний ресурс] - Режим доступу: https://github.com/Yours3lf/rpi-vk-driver

7 Vulkan Tutorial [Електронний ресурс] - Режим доступу: https://vulkan-tutorial.com

8 Android Vulkan Tutorials [Електронний ресурс] - Режим доступу: https://github.com/googlesamples/android-vulkan-tutorials

9 Android Developer Guide [Електронний ресурс] - Режим доступу: https://developer.android.com/docs

10 OpenGL Mathematics [Електронний ресурс] - Режим доступу: https://github.com/g-truc/glm

11 Wine Wayland [Електронний ресурс] - Режим доступу: https://github.com/Kron4ek/wine-wayland

12 Genshin Impact on Linux [Електронний ресурс] - Режим доступу: https://notabug.org/Kowalski/GI-on-Linux

13 DXVK [Електронний ресурс] - Режим доступу: https://github.com/doitsujin/dxvk

14 Makefile Tutorial [Електронний ресурс] - Режим доступу: https://makefiletutorial.com

15 KConfig Language Docs [Електронний ресурс] - Режим доступу: https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html

16 Intel Intrinsics Guide [Електронний ресурс] - Режим доступу: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

17 Обчислення на відеокартах [Відеокурс] - Режим доступу: https://www.youtube.com/playlist?list=PLlb7e2G7aSpTgwAm0GBkvn5XA0NokovJJ

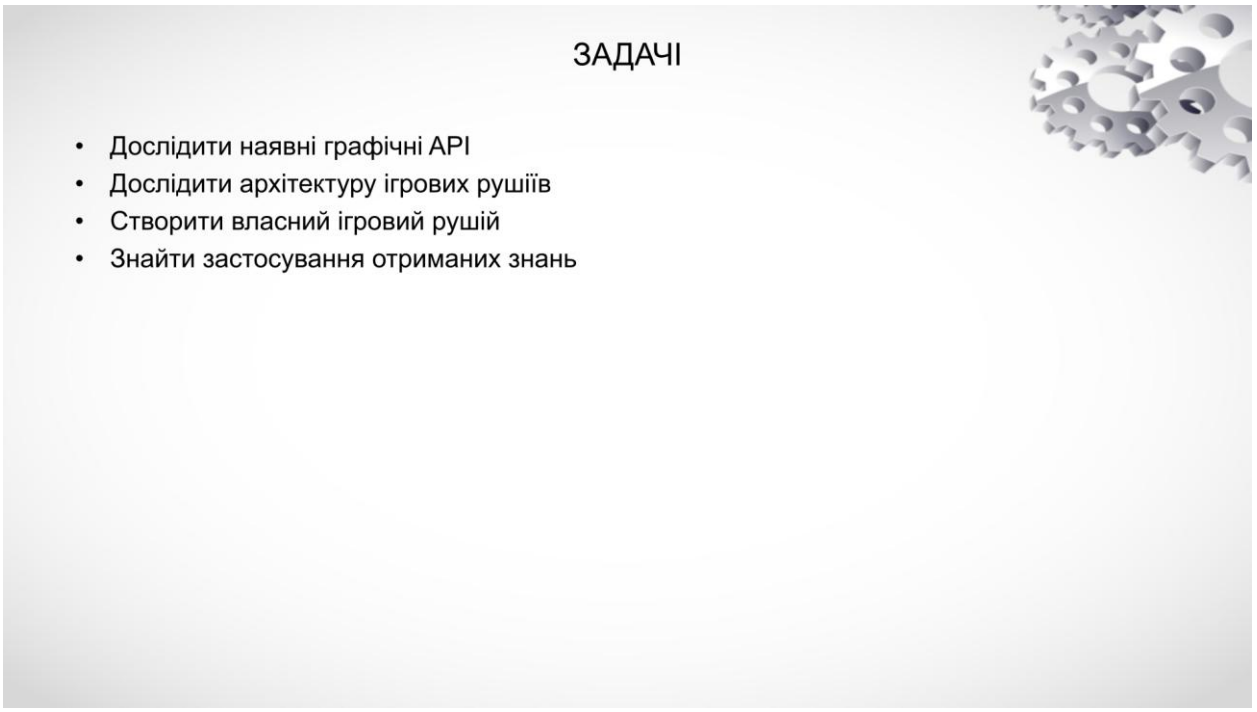
18 Що не вміє оптимізувати компілятор [Відеоурок] - Режим доступу: https://youtu.be/dAmhGElNUX0

19 Kohi Game Engine [Відеокурс] - Режим доступу: https://youtube.com/playlist?list=PLv8Ddw9K0JPg1BEO-RS-0MYs423cvLVtj

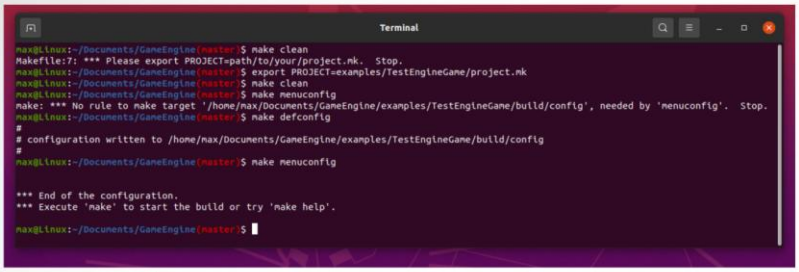
20 Vulkanised 2023 [Відеокурс] - Режим доступу: https://youtube.com/playlist?list=PLMLurvdlOpWOluO2PqfGO7XBU8FBlpDug

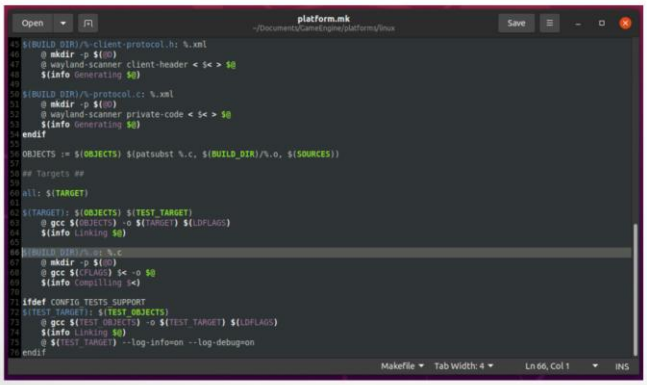
Додаток А

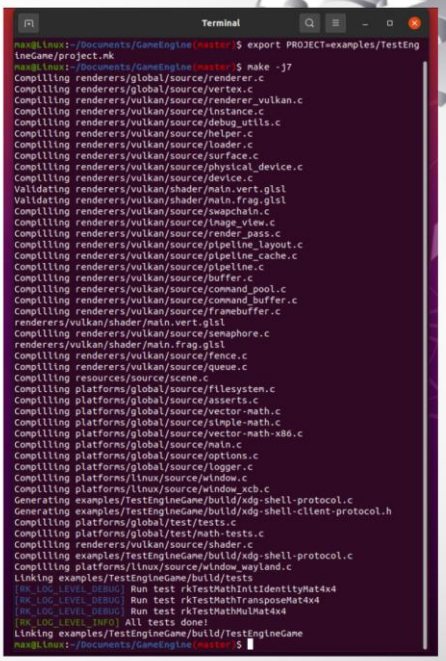
Презентація до захисту



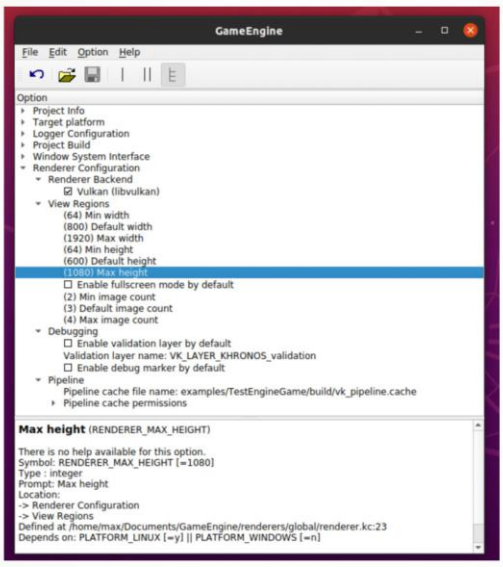
ІНСТРУМЕНТ АВТОМАТИЗАЦІЇ КОМПІЛЯЦІЇ MAKE

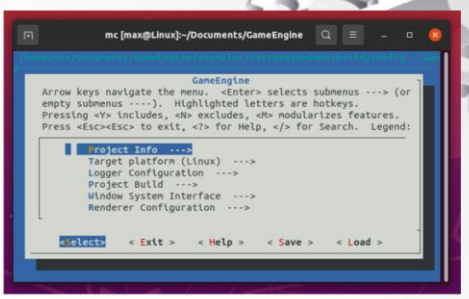






ІНСТРУМЕНТ НАЛАШТУВАННЯ ПАРАМЕТРІВ КОДУ KCONFIG





ПАРСИНГ АРГУМЕНТІВ КОМАНДНОГО РЯДКА

```c
static const RkOptSelectData rendererOptSelectData[] = {
#ifdef CONFIG_RENDERER_VULKAN
    RK_OPT_SELECT_DATA(RK_RENDERER_TYPE_VULKAN, "vulkan"),
#endif
    RK_OPT_SELECT_DATA(RK_RENDERER_TYPE_MAX, NULL),
};

static const RkOptSelectData boolOptSelectData[] = {
    RK_OPT_SELECT_DATA(RK_OPT_BOOL_FALSE, "off"),
    RK_OPT_SELECT_DATA(RK_OPT_BOOL_TRUE, "on"),
    RK_OPT_SELECT_DATA(RK_OPT_BOOL_MAX, NULL),
};

static const RkOpt cmdOpt[] = {
    RK_OPT_HELP(RK_OPT_KEY_HELP, "print this help message"),
#ifdef CONFIG_PLATFORM_LINUX
    RK_OPT_SELECT(RK_OPT_KEY_WSI, &cmdOptions.wsiType,
        wsiOptSelectData, "select window system interface"),
#endif
    RK_OPT_SELECT(RK_OPT_KEY_RENDERER, &cmdOptions.rendererType,
        rendererOptSelectData, "select renderer backend"),
    RK_OPT_NUMBER(RK_OPT_KEY_WIDTH, &cmdOptions.width, CONFIG_RENDERER_MIN_WIDTH,
        CONFIG_RENDERER_MAX_WIDTH, "set window width"),
    RK_OPT_NUMBER(RK_OPT_KEY_HEIGHT, &cmdOptions.height, CONFIG_RENDERER_MIN_HEIGHT,
        CONFIG_RENDERER_MAX_HEIGHT, "set window height"),
    RK_OPT_BOOL(RK_OPT_KEY_FULLSCREEN, &cmdOptions.fullscreen, "enable fullscreen mode"),
    RK_OPT_BOOL(RK_OPT_KEY_LOG_WARNING, &cmdOptions.logWarning, "enable warning logs"),
    RK_OPT_BOOL(RK_OPT_KEY_LOG_INFO, &cmdOptions.logInfo, "enable info logs"),
    RK_OPT_BOOL(RK_OPT_KEY_LOG_DEBUG, &cmdOptions.logDebug, "enable debug logs"),
#ifdef CONFIG_PLATFORM_LINUX
    RK_OPT_BOOL(RK_OPT_KEY_LOG_COLOR, &cmdOptions.logColor, "enable color output"),
#endif
    RK_OPT_NUMBER(RK_OPT_KEY_IMAGE_COUNT, &cmdOptions.imageCount, CONFIG_RENDERER_MIN_IMAGE_COUNT,
        CONFIG_RENDERER_MAX_IMAGE_COUNT, "set image count (double/triple buffering)"),
#ifdef CONFIG_RENDERER_VULKAN
    RK_OPT_BOOL(RK_OPT_KEY_VK_VALIDATION, &cmdOptions.vkValidation, "enable vulkan validation"),
    RK_OPT_STRING(RK_OPT_KEY_VK_VALIDATION_LAYER, &cmdOptions.vkValidationLayer,
        CONFIG_VULKAN_VALIDATION_LAYER_MIN, CONFIG_VULKAN_VALIDATION_LAYER_MAX,
        "vulkan validation layer name"),
    RK_OPT_BOOL(RK_OPT_KEY_VK_DEBUG_MARKER, &cmdOptions.vkDebugMarker,
        "enable vulkan debug marker, used by RenderDoc"),
#endif
};

static const struct option cmdOptionsInfo[] = {
    RK_OPT(RK_OPT_KEY_HELP, "help", no_argument),
#ifdef CONFIG_PLATFORM_LINUX
    RK_OPT(RK_OPT_KEY_WSI, "wsi", required_argument),
#endif
    RK_OPT(RK_OPT_KEY_RENDERER, "renderer", required_argument),
    RK_OPT(RK_OPT_KEY_WIDTH, "width", required_argument),
    RK_OPT(RK_OPT_KEY_HEIGHT, "height", required_argument),
```

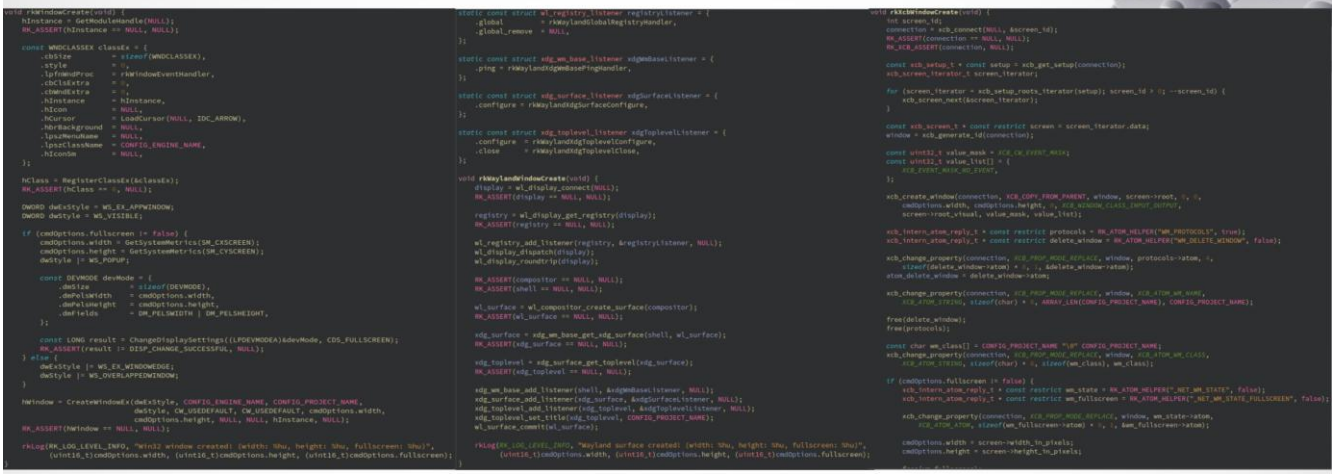
```
Terminal
max@Linux:~/Documents/GameEngine(master)$ ./examples/TestEngineGame/build/TestEngineGame --help
--help - print this help message.
--wsi=wayland - select window system interface. ['wayland', 'xcb']
--renderer=vulkan - select renderer backend. ['vulkan']
--width=800 - set window width. (min: 64, max: 1920)
--height=600 - set window height. (min: 64, max: 1080)
--fullscreen=off - enable fullscreen mode. ['off', 'on']
--log-warning=on - enable warning logs. ['off', 'on']
--log-info=off - enable info logs. ['off', 'on']
--log-debug=off - enable debug logs. ['off', 'on']
--log-color=on - enable color output. ['off', 'on']
--image-count=3 - set image count (double/triple buffering). (min: 2, max: 4)
--vulkan-validation=off - enable vulkan validation. ['off', 'on']
--vulkan-validation-layer=VK_LAYER_KHRONOS_validation - vulkan validation layer name. (minLength: 4, maxLength: 32)
--vulkan-debug-marker=off - enable vulkan debug marker, used by RenderDoc. ['off', 'on']
max@Linux:~/Documents/GameEngine(master)$
```

МОДУЛЬ ВЕКТОРНОЇ МАТЕМАТИКИ

```c
void rkMathRotateX(mat4x4_t * const restrict mat, const float angle) {
    mat4x4_t in;
    rkMathInitIdentityMat4x4(&in);

    const float cos = cosf(angle);
    const float sin = sinf(angle);

    in.comp[1][1] =  cos;
    in.comp[1][2] =  sin;
    in.comp[2][1] = -sin;
    in.comp[2][2] =  cos;

    mat4x4_t out;
    rkMathMulMat4x4(&out, mat, &in);
    *mat = out;
}

void rkMathRotateY(mat4x4_t * const restrict mat, const float angle) {
    mat4x4_t in;
    rkMathInitIdentityMat4x4(&in);

    const float cos = cosf(angle);
    const float sin = sinf(angle);

    in.comp[0][0] =  cos;
    in.comp[0][2] = -sin;
    in.comp[2][0] =  sin;
    in.comp[2][2] =  cos;

    mat4x4_t out;
    rkMathMulMat4x4(&out, mat, &in);
    *mat = out;
}

void rkMathRotateZ(mat4x4_t * const restrict mat, const float angle) {
    mat4x4_t in;
    rkMathInitIdentityMat4x4(&in);

    const float cos = cosf(angle);
    const float sin = sinf(angle);

    in.comp[0][0] =  cos;
    in.comp[0][1] =  sin;
    in.comp[1][0] = -sin;
    in.comp[1][1] =  cos;

    mat4x4_t out;
    rkMathMulMat4x4(&out, mat, &in);
    *mat = out;
}
```

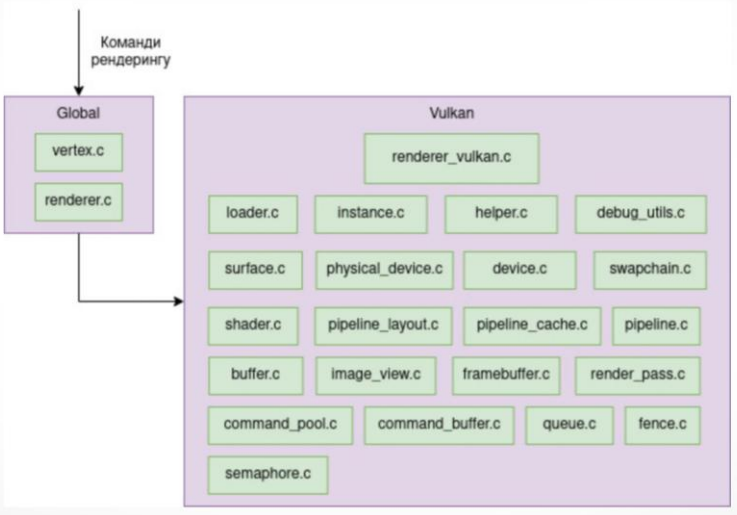
```c
void rkMathMulMat4x4(mat4x4_t * const restrict out, const mat4x4_t * const restrict in1,
                     const mat4x4_t * const restrict in2)
{
    const __m128 in2vec[] = {
        _mm_load_ps(in2->vec[0].comp),
        _mm_load_ps(in2->vec[1].comp),
        _mm_load_ps(in2->vec[2].comp),
        _mm_load_ps(in2->vec[3].comp),
    };

    for (uint_fast8_t i = 0; i < ARRAY_LEN(out->vec); i++) {
        const __m128 in1vec = _mm_load_ps(in1->vec[i].comp);
        const __m128 e0 = _mm_shuffle_ps(in1vec, in1vec, _MM_SHUFFLE(0, 0, 0, 0));
        const __m128 e1 = _mm_shuffle_ps(in1vec, in1vec, _MM_SHUFFLE(1, 1, 1, 1));
        const __m128 e2 = _mm_shuffle_ps(in1vec, in1vec, _MM_SHUFFLE(2, 2, 2, 2));
        const __m128 e3 = _mm_shuffle_ps(in1vec, in1vec, _MM_SHUFFLE(3, 3, 3, 3));

        const __m128 m0 = _mm_mul_ps(in2vec[0], e0);
        const __m128 m1 = _mm_mul_ps(in2vec[1], e1);
        const __m128 m2 = _mm_mul_ps(in2vec[2], e2);
        const __m128 m3 = _mm_mul_ps(in2vec[3], e3);

        const __m128 a0 = _mm_add_ps(m0, m1);
        const __m128 a1 = _mm_add_ps(m2, m3);
        const __m128 a2 = _mm_add_ps(a0, a1);
        _mm_store_ps(out->vec[i].comp, a2);
    }
}

void rkMathTransposeMat4x4(mat4x4_t * const restrict out,
                           const mat4x4_t * const restrict in)
{
    const __m128 v0 = _mm_load_ps(in->vec[0].comp);
    const __m128 v1 = _mm_load_ps(in->vec[1].comp);
    const __m128 v2 = _mm_load_ps(in->vec[2].comp);
    const __m128 v3 = _mm_load_ps(in->vec[3].comp);

    const __m128 t0 = _mm_shuffle_ps(v0, v1, _MM_SHUFFLE(1, 0, 1, 0));
    const __m128 t2 = _mm_shuffle_ps(v0, v1, _MM_SHUFFLE(3, 2, 3, 2));
    const __m128 t1 = _mm_shuffle_ps(v2, v3, _MM_SHUFFLE(1, 0, 1, 0));
    const __m128 t3 = _mm_shuffle_ps(v2, v3, _MM_SHUFFLE(3, 2, 3, 2));

    _mm_store_ps(out->vec[0].comp, _mm_shuffle_ps(t0, t1, _MM_SHUFFLE(2, 0, 2, 0)));
    _mm_store_ps(out->vec[1].comp, _mm_shuffle_ps(t0, t1, _MM_SHUFFLE(3, 1, 3, 1)));
    _mm_store_ps(out->vec[2].comp, _mm_shuffle_ps(t2, t3, _MM_SHUFFLE(2, 0, 2, 0)));
    _mm_store_ps(out->vec[3].comp, _mm_shuffle_ps(t2, t3, _MM_SHUFFLE(3, 1, 3, 1)));
}
```

СТВОРЕННЯ ВІКНА ІГРОВОГО РУШІЯ

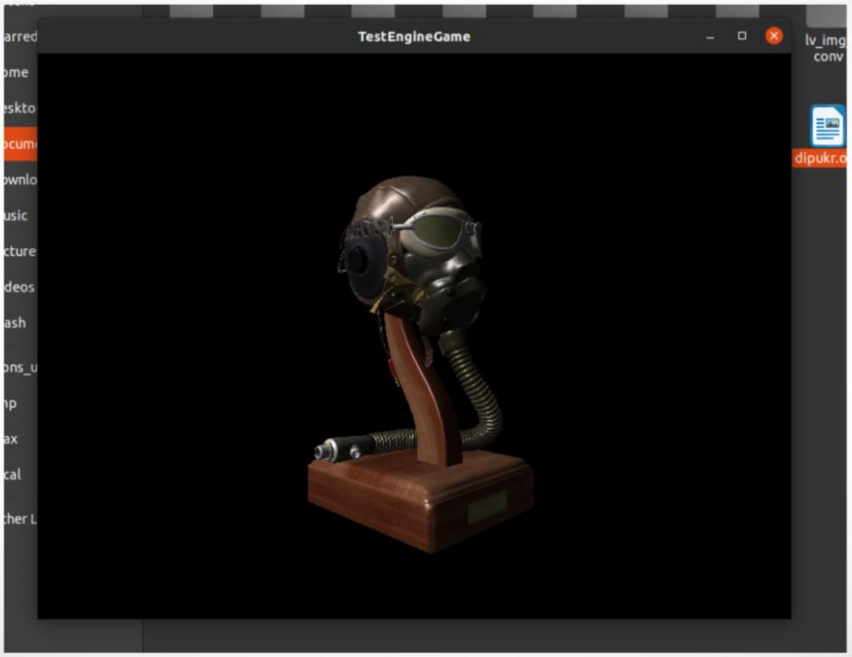


Windows | Linux (Wayland) | Linux (X.Org)

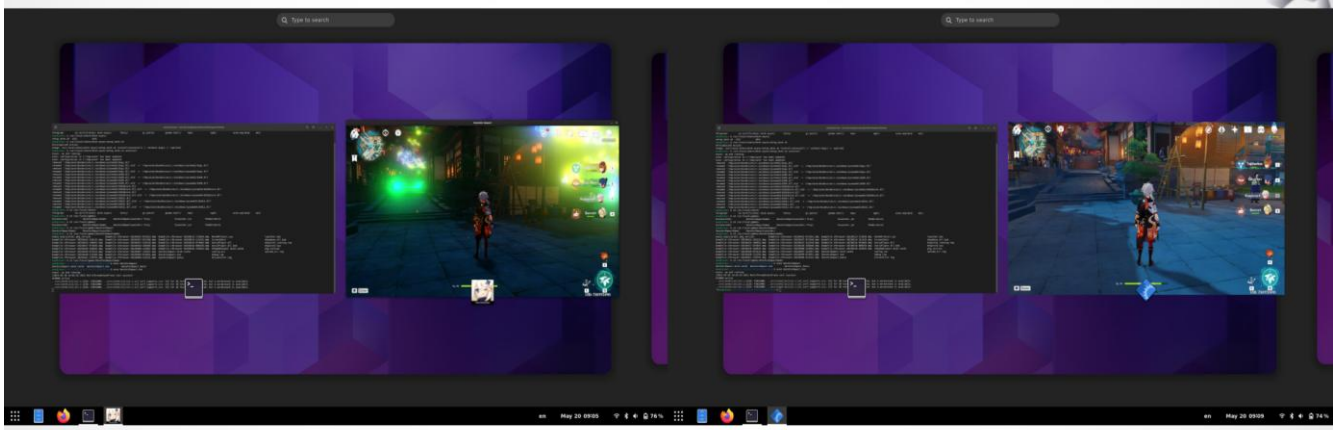
МОДУЛЬ РЕНДЕРИНГУ НА БАЗІ VULKAN API



ТЕСТУВАННЯ ІГРОВОГО РУШІЯ GAMEENGINE



ЗАПУСК WINDOWS ДОДАТКІВ НА LINUX



Конвертація Direct3D функцій в OpenGL

Конвертація Direct3D функцій в Vulkan

ВИСНОВКИ

- У ході роботи були досліджені можливості сучасних відеокарт та графічних API
- Створений власний ігровий рушій GameEngine, який може відображати та рухати 3D моделі
- Досліджені проблеми запуску Windows додатків на Linux, деякі з них були вирішені, а патчі з виправленнями були надіслані розробникам
- Отриманий опит у створенні платформонезалежних додатків, а деякі модулі ігрового рушія GameEngine можна використовувати у подальших проєктах

Додаток Б

Лістинг файлу pipeline.c (налаштування графічного конвеєра)

```
#define VK_SHADER(type, shader)                                                 \
{                                                                               \
        .sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO,           \
        .pNext = NULL,                                                          \
        .flags = 0,                                                             \
        .stage = type,                                                          \
        .module = shader,                                                       \
        .pName = "main",                                                        \
        .pSpecializationInfo = NULL,                                            \
}

#define VK_VERTEX_BINDING(index, size, rate)          \
{                                                                               \
        .binding = index,                                                       \
        .stride = size,                                                         \
        .inputRate = rate,                                                      \
}

#define VK_VERTEX_ATTRIBUTE(index, bindIndex, attrFormat, attrOffset)                   \
{                                                                               \
        .location = index,                                                      \
        .binding = bindIndex,                                                   \
        .format = attrFormat,                                                   \
        .offset = attrOffset,                                                   \
}

typedef enum {
        VK_GRAPHICS_PIPELINE_INDEX_MIN,
        VK_GRAPHICS_PIPELINE_INDEX_MAIN = VK_GRAPHICS_PIPELINE_INDEX_MIN,
        VK_GRAPHICS_PIPELINE_INDEX_MAX,
} VkGraphicsPipelineIndex;

typedef enum {
        VK_VERTEX_BINDING_INDEX_MIN,
        VK_VERTEX_BINDING_INDEX_MAIN = VK_VERTEX_BINDING_INDEX_MIN,
```

```
	VK_VERTEX_BINDING_INDEX_MAX,
} VkVertexBindingIndex;

typedef enum {
	VK_ATTRIBUTE_INDEX_MIN,
	VK_ATTRIBUTE_INDEX_POSITION = VK_ATTRIBUTE_INDEX_MIN,
	VK_ATTRIBUTE_INDEX_COLOR,
	VK_ATTRIBUTE_INDEX_MAX,
} VkAttributeIndex;

typedef enum {
	VK_COLOR_BLEND_ATTACHTMENT_INDEX_MIN,
	VK_COLOR_BLEND_ATTACHTMENT_INDEX_MAIN = VK_COLOR_BLEND_ATTACHTMENT_INDEX_MIN,
	VK_COLOR_BLEND_ATTACHTMENT_INDEX_MAX,
} VkColorBlendAttachmentIndex;

static const VkVertexInputBindingDescription inputBindingDescriptions[] = {
	VK_VERTEX_BINDING(VK_VERTEX_BINDING_INDEX_MAIN, sizeof(vertext_t),
VK_VERTEX_INPUT_RATE_VERTEX),
};

static const VkVertexInputAttributeDescription inputAttributeDescription[] = {
	VK_VERTEX_ATTRIBUTE(VK_ATTRIBUTE_INDEX_POSITION, VK_VERTEX_BINDING_INDEX_MAIN,
		VK_FORMAT_R32G32B32_SFLOAT, offsetof(vertext_t, inPosition)),
	VK_VERTEX_ATTRIBUTE(VK_ATTRIBUTE_INDEX_COLOR, VK_VERTEX_BINDING_INDEX_MAIN,
		VK_FORMAT_R32G32B32_SFLOAT, offsetof(vertext_t, inColor)),
};

static const VkPipelineVertexInputStateCreateInfo vertexInputInfo = {
	.sType                                                    =
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO,
	.pNext                                    = NULL,
	.flags                                    = 0,
	.vertexBindingDescriptionCount            = ARRAY_LEN(inputBindingDescriptions),
	.pVertexBindingDescriptions          = inputBindingDescriptions,
	.vertexAttributeDescriptionCount     = ARRAY_LEN(inputAttributeDescription),
	.pVertexAttributeDescriptions             = inputAttributeDescription,
};

static const VkPipelineInputAssemblyStateCreateInfo inputAssemblyInfo = {
```

```
	.sType                                  =
VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO,
	.pNext                                  = NULL,
	.flags                                  = 0,
	.topology                       = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST,
	.primitiveRestartEnable     = VK_FALSE,
};

static const VkPipelineViewportStateCreateInfo viewportInfo = {
	.sType                  = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO,
	.pNext                  = NULL,
	.flags                  = 0,
	.viewportCount          = 0,
	.pViewports             = NULL,
	.scissorCount           = 0,
	.pScissors              = NULL,
};

static const VkPipelineRasterizationStateCreateInfo rasterizationInfo = {
	.sType                                  =
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO,
	.pNext                                  = NULL,
	.flags                                  = 0,
	.depthClampEnable                       = VK_FALSE,
	.rasterizerDiscardEnable        = VK_FALSE,
	.polygonMode                            = VK_POLYGON_MODE_FILL,
	.cullMode                               = VK_CULL_MODE_BACK_BIT,
	.frontFace                              = VK_FRONT_FACE_CLOCKWISE,
	.depthBiasEnable                = VK_FALSE,
	.depthBiasConstantFactor        = 0.0f,
	.depthBiasClamp                 = 0.0f,
	.depthBiasSlopeFactor                   = 0.0f,
	.lineWidth                              = 1.0f,
};

static const VkPipelineMultisampleStateCreateInfo multisampleInfo = {
	.sType                                  =
VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO,
	.pNext                                  = NULL,
	.flags                                  = 0,
```

```
	.rasterizationSamples		= VK_SAMPLE_COUNT_1_BIT,
	.sampleShadingEnable		= VK_FALSE,
	.minSampleShading		= 1.0f,
	.pSampleMask			= NULL,
	.alphaToCoverageEnable	= VK_FALSE,
	.alphaToOneEnable		= VK_FALSE,
};

static const VkPipelineColorBlendAttachmentState colorBlendAttachments[] = {
	[VK_COLOR_BLEND_ATTACHTMENT_INDEX_MAIN] = {
		.blendEnable		= VK_FALSE,
		.srcColorBlendFactor	= VK_BLEND_FACTOR_ONE,
		.dstColorBlendFactor	= VK_BLEND_FACTOR_ZERO,
		.colorBlendOp		= VK_BLEND_OP_ADD,
		.srcAlphaBlendFactor	= VK_BLEND_FACTOR_ONE,
		.dstAlphaBlendFactor	= VK_BLEND_FACTOR_ZERO,
		.alphaBlendOp		= VK_BLEND_OP_ADD,
		.colorWriteMask	= VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT,
	},
};

static const VkPipelineColorBlendStateCreateInfo colorBlendInfo = {
	.sType			= VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO,
	.pNext			= NULL,
	.flags			= 0,
	.logicOpEnable		= VK_FALSE,
	.logicOp		= VK_LOGIC_OP_COPY,
	.attachmentCount	= ARRAY_LEN(colorBlendAttachments),
	.pAttachments		= colorBlendAttachments,
	.blendConstants[0]	= 0.0f,
	.blendConstants[1]	= 0.0f,
	.blendConstants[2]	= 0.0f,
	.blendConstants[3]	= 0.0f,
};

static const VkDynamicState dynamicStates[] = {
	VK_DYNAMIC_STATE_VIEWPORT,
	VK_DYNAMIC_STATE_SCISSOR,
};
```

```
static const VkPipelineDynamicStateCreateInfo dynamicInfo = {
	.sType				= VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO,
	.pNext				= NULL,
	.flags				= 0,
	.dynamicStateCount		= ARRAY_LEN(dynamicStates),
	.pDynamicStates		= dynamicStates,
};

VkPipeline mainPipeline = VK_NULL_HANDLE;

void vkCreateGraphicsPipelineWrap(void) {
	const VkPipelineShaderStageCreateInfo shaderStageInfo[] = {
		VK_SHADER(VK_SHADER_STAGE_VERTEX_BIT, vertexShader),
		VK_SHADER(VK_SHADER_STAGE_FRAGMENT_BIT, fragmentShader),
	};

	const VkGraphicsPipelineCreateInfo infos[] = {
		[VK_GRAPHICS_PIPELINE_INDEX_MAIN] = {
			.sType				= VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO,
			.pNext				= NULL,
			.flags				= 0,
			.stageCount			= ARRAY_LEN(shaderStageInfo),
			.pStages			= shaderStageInfo,
			.pVertexInputState		= &vertexInputInfo,
			.pInputAssemblyState		= &inputAssemblyInfo,
			.pTessellationState		= NULL,
			.pViewportState		= &viewportInfo,
			.pRasterizationState		= &rasterizationInfo,
			.pMultisampleState		= &multisampleInfo,
			.pDepthStencilState		= NULL,
			.pColorBlendState		= &colorBlendInfo,
			.pDynamicState		= &dynamicInfo,
			.layout				= mainPipelineLayout,
			.renderPass			= renderPass,
			.subpass			= VK_SUBPASS_INDEX_MAIN,
			.basePipelineHandle		= VK_NULL_HANDLE,
			.basePipelineIndex		= 0,
		},
```

```
	};

	const VkResult result = vkCreateGraphicsPipelines(
		mainDevice, mainPipelineCache, ARRAY_LEN(infos), infos, allocationCallbacks, &mainPipeline
	);
	RK_VULKAN_ASSERT(result, NULL);
	vkLog(VULKAN_TAG_INFO, "Main pipeline created!");

	// Shaders are no longer needed
	vkDestroyShaderModulesWrap();
}

void vkDestroyGraphicsPipelineWrap(void) {
	if (mainPipeline != VK_NULL_HANDLE) {
		vkDestroyPipeline(mainDevice, mainPipeline, allocationCallbacks);
		vkLog(VULKAN_TAG_INFO, "Main pipeline destroyed!");
		mainPipeline = VK_NULL_HANDLE;
	}
}
```