Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет навчально-науковий центр заочної форми навчання
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

АТЕСТАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Модель кросплатформеного низькорівневого програмного модуля VulkanKit

(тема)

Виконав:
студент II курсу, групи СПзм-18-2
Мельніченко Ф.О.
(прізвище, ініціали)

Спеціальність
123 – Комп’ютерна інженерія
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: проф. Каргін А.О.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ _____ Коваленко А.А.
(підпис) (прізвище, ініціали)

2020 р.

Харківський національний університет радіоелектроніки

Факультет навчально-науковий центр заочної форми навчання

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 – Комп'ютерна інженерія
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
"____" _____ 20___ р.

ЗАВДАННЯ

НА АТЕСТАЦІЙНУ РОБОТУ

студентові Мельніченку Федіру Олександровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Модель кросплатформеного низькорівневого програмного модуля VulkanKit

затверджена наказом по університету від "30" березня 2020 р. № 43 Стз

2. Термін подання студентом роботи до екзаменаційної комісії 18 травня 2020 р.

3. Вхідні дані до роботи

1. Програмний модуль VulkanKit
2. Мова програмування C++

4. Перелік питань, що потрібно опрацювати в роботі

1. Особливості Vulkan
2. Реалізація VulkanKit
3. Архітектура компонентів VulkanKit
4. Результати досліджень та розробки
5. Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) Демонстраційні матеріали. Плакати – № 18 - арк. ф. А4

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
	залишити пустим, якщо кон-		
	сультантом є керівник роботи		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз матеріалів джерел	31.03.20 – 17.04.20	
2	Побудова та розробка концептуальної моделі	18.04.20 – 2.05.20	
3	Проведення досліджень	3.05.20 – 7.05.20	
4	Підготовка та редагування роботи	8.05.20 – 11.05.20	
5	Подання атестаційної роботи керівникові та її	12.05.20 – 13.05.20	
	попередній захист		
6	Подання атестаційної роботи на рецензування	14.05.20 – 15.05.20	

Дата видачі завдання 30 березня 2020 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Каргін А.О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка атестаційної роботи: 69 с., 11 рис., 0 табл., 1 дод., 16 джерел.

API, OpenGL, VULKAN, VULKANKIT, PARENT-CHILD, GPU, SPIR-V, KHRONOS, C++.

Метою атестаційної роботи є розробка та представлення концепції дизайну програмного модуля.

У ході виконання атестаційної роботи було проаналізовано програмний модуль VulkanKit та створено власну концепцію дизайну програмного модуля VulkanKit для. спрощення розробки на Vulkan шляхом надання об’єктно-орієнтованого дизайну, безпеки типів на етапі компіляції, автоматизованих методів управління часом життя об’єктів, спрощеного та явного програмного інтерфейсу (API) для запитів функціоналу, обмежень та розширень, дизайну «Parent-Child» для компонентів, прямий виклик функцій Vulkan, що має привести до більшої продуктивності в великомасштабних проектах.

ABSTRACT

Master’s thesis: 69 pages, 11 figures, 0 tables,1 appendices, 16 sources.

API, OpenGL, VULKAN, VULKANKIT, PARENT-CHILD, GPU, SPIR-V, KHRONOS, C++.

The major goal of this thesis is the development and presentation of the design concept of the software module.

During the attestation work, the VulkanKit software module was analyzed and the own design concept of the VulkanKit software module for. simplification of Vulkan development by providing object-oriented design, compiler-type type security, automated object lifecycle management methods, simplified and explicit programming interface (API) for functionality requests, constraints and extensions, Parent-Child design for components, direct call to Vulkan features, which should lead to greater productivity in large-scale projects.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API –Application Programming Interface

GPU –Graphics Processing Unit

OpenGL – Open Graphics Library

SPIR-V – Standard Portable Intermediate Representation

ВСТУП

У даній роботі представлена концепція дизайну програмного модуля VulkanKit. VulkanKit – програмний модуль на базі мови програмування C++ для Vulkan.

Основна мета VulkanKit – спростити розробку на Vulkan шляхом надання об’єктно-орієнтованого дизайну, безпеки типів на етапі компіляції, автоматизованих методів управління часом життя об’єктів, спрощеного та явного програмного інтерфейсу (API) для запитів функціоналу, обмежень та розширень, дизайну «Parent-Child» для компонентів, прямий виклик функцій Vulkan, що має привести до більшої продуктивності в великомасштабних проектах. Увесь цей функціонал повинен зробити код коротшим і простішим для розуміння, ніж нативний код Vulkan.

1 VULKAN

У 2015 році Khronos Group, відкритий консорціум провідних компаній програмного забезпечення та апаратних засобів, анонсував новий відкритий стандарт – API Vulkan. «Vulkan – важлива ініціатива Khronos, яка надає розробникам вибір найсучаснішого GPU API, відкритого та портативного на багатьох платформах, у той час, коли різноманітність платформ збільшується», – повідомляє Ніл Треветт, президент Khronos Group та віце-президент NVIDIA. «Khronos буде керувати екосистемою Vulkan за допомогою тестових компонентів з відкритим вихідним кодом та зразкових реалізацій компіляторів, що використовують SPIR-V для найбільш ефективного вкладення інвестицій спільноти апаратного забезпечення в оптимізовані драйвери[1, 2 5]. Vulkan розширює сімейство 3D API від Khronos та доповнює OpenGL та OpenGL ES, які всі разом сьогодні забезпечують доступ до мільярдів графічних процесорів, і надалі розвиватимуться та підтримуватимуться для задоволення потреб галузі».

Отже, Vulkan – це кросплатформний інтерфейс програмування нового покоління (API) для розмаїття графічних і обчислювальних пристроїв. Наразі, Khronos Group є основним розробником Vulkan, однак, будь-яка компанія може брати участь у постійному розвитку Vulkan, приєднавшись до Khronos. Вперше технічний огляд API Vulkan був представлений на Конференції Розробників Ігор (GDC) у Сан-Франциско[2,6]. Vulkan також забезпечує пряме керування графічним процесором, чого вимагають вдосконалені ігрові двигуни, програмне забезпечення та додатки з перехресними можливостями постачальника та функціональною портативністю внаслідок більш простих і прогнозованих драйверів, тож, Vulkan отримав визнання у розробці ігор, оскільки . «Промислові стандартні API, такі як Vulkan, є важливою частиною, яка дозволяє розробникам приносити найкращий досвід клієнтам на багатьох платформах», –

повідомляє Гейб Ньюелл від Valve, – «Valve та інші члени Khronos наполегливо працюють заради того, щоб цей високопродуктивний графічний інтерфейс був доступним якомога ширше, і ми розглядаємо його як критичний компонент SteamOS та майбутніх ігор Valve»[1, 7, 8, 9].

Основою для Vulkan був Mantle API від AMD, однієї з провідних компаній-виробників інтегрованої електроніки. Основна мета Vulkan – продуктивність, задля цього для додатків був забезпечений безпосередній контроль над прискоренням GPU. Vulkan представляє безліч нових функцій і концепцій, які можуть бути незнайомі навіть досвідченим програмістам. Vulkan – це явний API. Прикладні програмісти зобов’язані повністю контролювати управління пам’яттю, синхронізувати стан ресурсів, виконання команд і т.д. Це створює багато проблем для програмістів, що, в свою чергу, вимагає глибоких знань Vulkan і його ефективного застосування. В іншому випадку потрібно бути готовим до зниження продуктивності свого застосунку. Vulkan API зроблений на C і має об’єктну модель, яка забезпечує ідеальний і явний контроль над ресурсами Vulkan з точки зору розробки програмного забезпечення. Vulkan (як і Direct3D 12) [13, 15] дозволяє планувати робоче навантаження для кожної доступної черзі в графічному процесорі для одночасного виконання, яке називається Async Compute. Крім того, Vulkan був розроблений для багатоядерних процесорів, що дозволяє розробникам більш ефективно завантажувати процесори.

На сьогоднішній день існують версії Vulkan 1.0, Vulkan 1.1 та Vulkan 1.2. Khronos Group продовжувала сумлінно дослухатися до потреб розробників після випуску Vulkan 1.0, тож у березні 2018 року консорціум анонсував вихід нової версії API – Vulkan 1.1, що має розширений функціонал ядра, таких як операції підгрупи (subgroup operations) із інтегрованим широким спектром перевірених розширень для Vulkan 1.0. AMD, Arm, Imagination, Intel Corporation, NVIDIA та Qualcomm випустили сумісні з Vulkan 1.1 драйвери[9, 11, 16]. На даний момент Vulkan постачається практично на всі платформи з підтримкою GPU, включаючи

Windows 7, 8.X, 10, Android 7.0+ та Linux, а також Khronos нещодавно анонсував інструменти з відкритим кодом, для забезпечення переносимості Vulkan 1.0 додатків переносилася на macOS та iOS. Vulkan отримав широку підтримку у провідних ігрових двигунах, включаючи Unreal, Unity, Source 2 від Valve, id Tech, Serious Engine від CroTeam, CryEngine та Xenko. Vulkan використовується в більш ніж 30 найсучасніших іграх на різних настільних і мобільних платформах, включаючи Doom, Quake, Roblox, The Talos Principle, Dota 2, і є ексклюзивним API, що використовується в AAA іграх, таких як Wolfenstein II і Doom VFR[6].

1.1 Особливості Vulkan

Як згадувалося вище, Vulkan належить до сімейства API нового покоління разом з Direct3D 12 (Microsoft), Metal (Apple), але в будь-якому випадку Vulkan більш низькорівневий і більш явний, як показують деякі живі тести. У порівнянні з API попереднього покоління, такими як OpenGL та Direct3D 11, Vulkan дає менше навантаження на процесор (рисунок 1.1). Цього можна домогтися, надавши розробнику додатка відповідальність за управління володінням, пам’яттю і синхронізацією. У випадку OpenGL драйвер контролює такого роду проблеми, що, в свою чергу, вимагає досить великих накладних витрат на стороні процесора[11, 16]. Як і OpenGL, Vulkan – це кросплатформена система, яка доступна для настільних, мобільних, вбудованих і консольних систем. Vulkan API має дизайн об’єктної моделі, в той час як OpenGL має дизайн кінцевого автомата[8].

Для програмування шейдерів в Vulkan використовується мова програмування GLSL. В порівнянні з OpenGL, синтаксис GLSL для Vulkan став більш розширеним для адресації ресурсів в шейдері. Vulkan не компілює код шейдера на пряму[11, 15]. Для цього було розроблено технологію SPIR (Standard Portable Intermediate Representation). SPIR-V дозволяє мовним інтерфейсам високого рівня випускати програми в стандартизованій

проміжній формі для завантаження драйверами Vulkan, OpenGL або OpenCL[13]. SPIR-V усуває необхідність в високорівневих мовних інтерфейсних компіляторах в драйверах пристроїв, значно зменшуючи складність драйверів, дозволяє широкому спектру мовних та каркасних зовнішніх інтерфейсів працювати на різноманітних апаратних архітектурах та підтримує живу екосистему аналізу з відкритим вихідним кодом. інструменти портирования, налагодження та оптимізації. SPIR-V також надає можливість використовувати мову програмування шейдерів HLSL.

Нова функціональність у Vulkan 1.1 включає операції підгруп, які дозволяють високоефективно обмінюватися даними та маніпулювати між різними завданнями, що працюють паралельно на GPU. Vulkan 1.1 також дозволяє програмам виконувати операції візуалізації та відображення з використанням ресурсів, до яких вони не можуть отримати доступ або скопіювати – для безпечного відтворення та відображення захищеного мультимедійного вмісту.

Крім того, інтегровано широкий діапазон розширень Vulkan 1.0, що приносить значну перевірену функціональність в ядро Vulkan 1.1, включаючи: одночасну візуалізацію декількох зображень, використання декількох графічних процесорів в одній системі та взаємоперемісність міжпроцесового API для розширеного візуалізації і складання операцій, які часто використовуються в складних програмах, таких як віртуальна реальність[8, 11, 16]. Ці основні функціональні можливості також включають в себе розширені обчислення з 16-бітовим доступом до пам’яті та підтримку макетів пам’яті HLSL, а також відображення, обробку та композицію відеопотоків шляхом прямої вибірки текстур кольорового формату YCbCr, створених багатьма відео кодеками.

На додачу до випуску Vulkan 1.1 з’явилась нова специфікація SPIR-V 1.3, яка розширює можливості проміжного представлення шейдерів Vulkan для підтримки операцій підгруп та дозволяє розширити оптимізацію компілятора. Екосистема інструментів SPIR-V продовжує набирати значних

обертів разом із компіляторами для підключення як для GLSL, так і для HLSL, а також розширює підтримку інструментарію низького рівня від проекту SPIRV-Tools з відкритим кодом.

1.2 Підтримка Vulkan у галузі

«Прямо впливаючи на декілька поколінь апаратних та програмних архітектур для платформ від міліватних до кіловатних, OpenGL, безперечно, є найуспішнішим в галузі 3D-графіки API», – сказав Раджа Кодурі, віце-президент корпорації, візуальні та перцептивні обчислення, AMD. «Vulkan – це перетворення на OpenGL, який забезпечує захоплюючі можливості з низькими накладними витратами, щоб домогтися переконливого підвищення продуктивності та енергоефективності, зберігаючи продуктивність розробника».

«З часів допомоги у заснуванні Khronos, ARM прагнула підвищити ефективність стандартів та поставляти переконливу графіку на мобільні пристрої з мінімальним споживанням енергії», – заявив Джем Девіс, віце-президент ARM Ltd., один з найбільших розробників та ліцензіарів ARM архітектури. «Vulkan – це великий крок вперед, що дозволяє нашій екосистемі розробників розкрити можливості новітньої технології GPU GPU»[12, 13, 16].

«Codeplay схвильована, що функція Vulkan SPIR-V дозволить використовувати нові мови та інструменти у графіці. Шляхом стандартизації способу додавання нових мов для графіки, відкривається ціла нова екосистема інструментів», – Ендрю Річардс, генеральний директор Codeplay[3].

«Як один з найбільших виробників цифрових автомобільних кластерних приладів та інформаційно-розважальних систем, Continental повністю підтримує нову ініціативу Vulkan від Khronos», – доктор Ульріх Кабатек, головний технічний експерт графічних систем та 3D-візуалізації в

Continental Automotive.

«Imagination з гордістю може бути головним розробником нового стандарту Vulkan. Завдяки цьому новому API поряд з OpenGL ES ми вважаємо, що зараз у компанії Khronos є правильний набір API для надання надзвичайної продуктивності та основних напрямків ринку графічного програмування», – стверджує Петер МакГіннес, директор з маркетингу технологій Imagination Technologies.

«NVIDIA активно підтримує OpenGL, і ми активно беремо участь у розробці Vulkan», – висловлюється Бартольд Ліхтенбельт, старший директор Tegra graphics software в NVIDIA. «Фокус Vulkan на забезпеченні наявності портативних, високопродуктивних ігор та двигунів дозволить переносити передовий контент у межах усіх ігрових платформ NVIDIA, включаючи ПК, мобільні та хмарні платформи»[6-8].

1.3 Архітектура Vulkan

Докладне пояснення дизайну і концепцій Vulkan, а також способів його ефективного використання – все це вимагає величезної кількості часу, але в цій частині ми спробуємо зробити невеликий огляд Vulkan.

По суті, Vulkan має багаторівневу архітектуру (рисунок 1.2), де основним компонентом є завантажувач (Loader).

Завантажувач відповідає за завантаження драйверів, взаємодію додатків з встановленими клієнтськими драйверами (Installable Client Driver, ICD) та додатковими компонентами. ICD підтримує один або декілька графічних пристроїв (GPU), які представлені як VkPhysicalDevice. Завантажувач відповідає за виявлення доступних в системі Vulkan ICD, що дає можливість перераховувати доступні графічні процесори і надавати інформацію про них[8, 10].

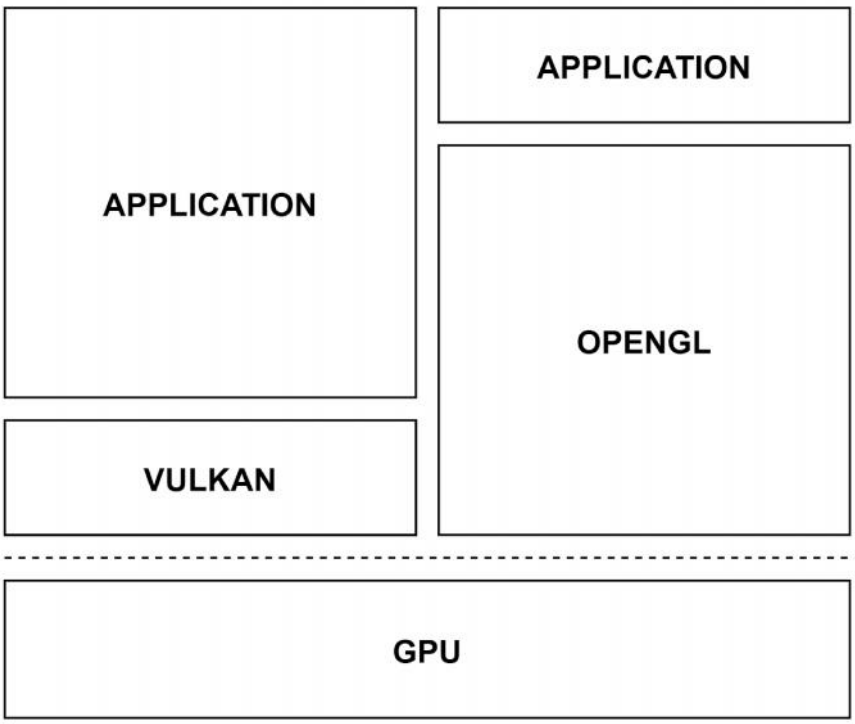

Рисунок 1.1 – Схематичне порівняння Vulkan та OpenGL

Як опціональні компоненти в Vulkan було впроваджено шари (Layers). Вони реалізовані у вигляді плагінів, які можуть бути включені по-різному. Існує два типи шарів: явний і неявний. Для використання явних шарів розробник програми повинен явно включити необхідні шари. Неявні шари завантажуються через завантажувач з системи автоматично. Шари часто використовуються для валідації API, налагодження або зображення додаткового накладення на вікні додатку. Багатошаровий дизайн Vulkan дає можливість багатьом незалежним виробникам електронних пристроїв (Independent Hardware Vendors, IHV) долучитися до загальної розширюваної архітектури для перевірки коду, налагодження та профілювання під час розробки, не впливаючи на показники продуктивності; Очікується, що така гнучкість за рахунок шарів сприятиме значним іноваціям в інструментах виробників GPU[15].

Є також розширення. Розширення є не обов’язковим функціоналом, який може надавати завантажувач, шар або ICD. Розширення змінюють або

розширюють основний функціонал Vulkan. Завантажувач дозволяє нам перераховувати доступні розширення для Instance та Device.

Vulkan має спеціальні розширення, такі як розширення Windows System Integration (WSI). Розширення WSI надають функціональні можливості і абстракції для подання/відображення результату рендеринга додатку Vulkan. По суті, розширення WSI можна розділити на два типи: платформо-залежні та платформо-незалежні, що забезпечують розробникам додатків достатню гнучкість (рисунок 1.3).

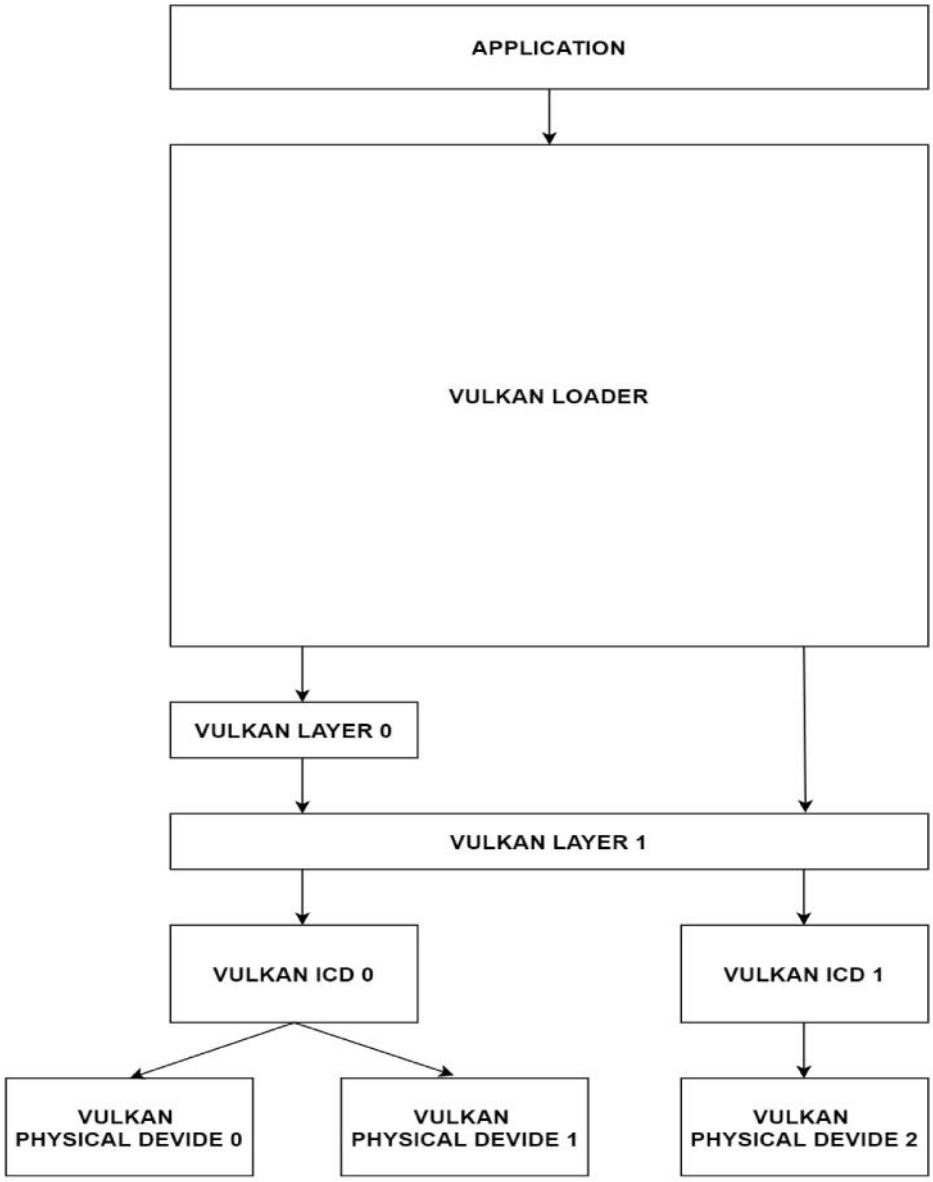

Рисунок 1.2 – Діаграма Vulkan архітектури

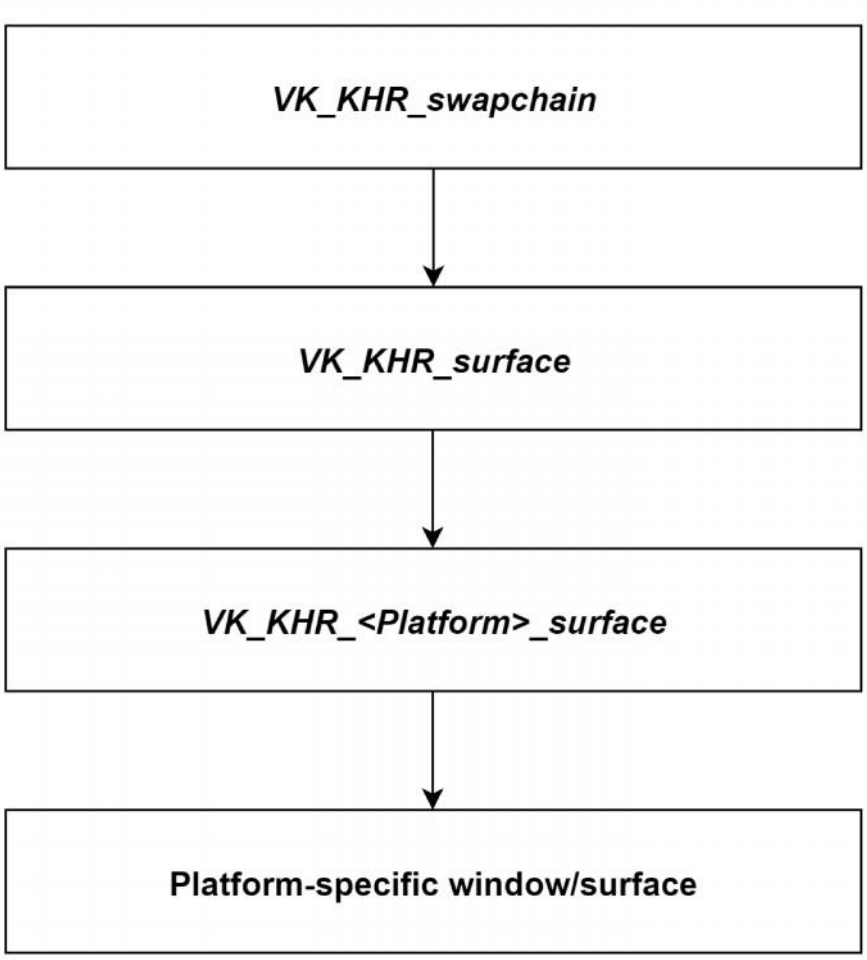

Рисунок 1.3 – Діаграма взаємодії WSI розширень

1.4 Vulkan компоненти

Vulkan включає ієрархію функціональних суб’єктів, починаючи з верхнього рівня, котрий виражений екземпляром, який об’єднує всі пристрої, що підтримує Vulkan. Тоді кожен пристрій виставляє одну або декілька черг. Ці черги виконують роботу, яку вимагає програма розробника.

Екземпляр Vulkan – це конструктор програмного забезпечення, який логічно відокремлює стан вашої програми від інших програм або бібліотек, що працюють у контексті вашої програми. Фізичні пристрої в системі представлені як члени екземпляра, кожен з яких має певні можливості, включаючи вибір наявних черг[5].

Фізичний пристрій зазвичай являє собою єдиний набір апаратних засобів або набір взаємопов’язаних апаратних засобів . У будь-якій системі існує фіксована кінцева кількість фізичних пристроїв. Логічний пристрій, створений екземпляром, являє собою конструкцію програмного забезпечення навколо фізичного пристрою і виконує резервування ресурсів, пов’язаних з певним фізичним пристроєм[8, 10, 16]. Це включає в себе можливу підмножину доступних черг на фізичному пристрої. Можна створити кілька логічних пристроїв, що представляють один фізичний пристрій.

Рисунок 1.4 ілюструє ієрархію суб’єктів. Можемо бачити, що додаток створив два екземпляри Vulkan. У системі є три фізичні пристрої, доступні для обох випадків. Після перерахування додаток створює на логічному пристрої перший фізичний пристрій, два логічних пристрої для другого пристрою та інший – для третього[1, 3].

Кожен логічний пристрій надає підмножину відповідних черг для фізичного пристрою. Але на практиці більшість додатків Vulkan не будуть такими складними.

Екземпляр Vulkan (VkInstance) – це перший об’єкт, який необхідно створити для забезпечення роботи Vulkan у додатку. Він являє собою з’єднання між створюваною програмою та середовищем виконання Vulkan, і тому він повинен існувати лише один раз у додатку, що розробляється. Також зберігається всі прикладні налаштування та інформація, необхідна для використання Vulkan. Тому ви повинні вказати всі шари (наприклад, Layer Validation) і всі розширення, які потрібно включити під час створення екземпляра[4].

Фізичний пристрій Vulkan (VkPhysicalDevice) являє собою конкретний пристрій, як графічна карта. Після перерахування фізичних пристроїв з екземпляру, після чого з’являється можливість запиту цих даних для своїх постачальників ідентифікаторів, пристроїв і підтримуваних функцій, а також інших властивостей та обмежень.

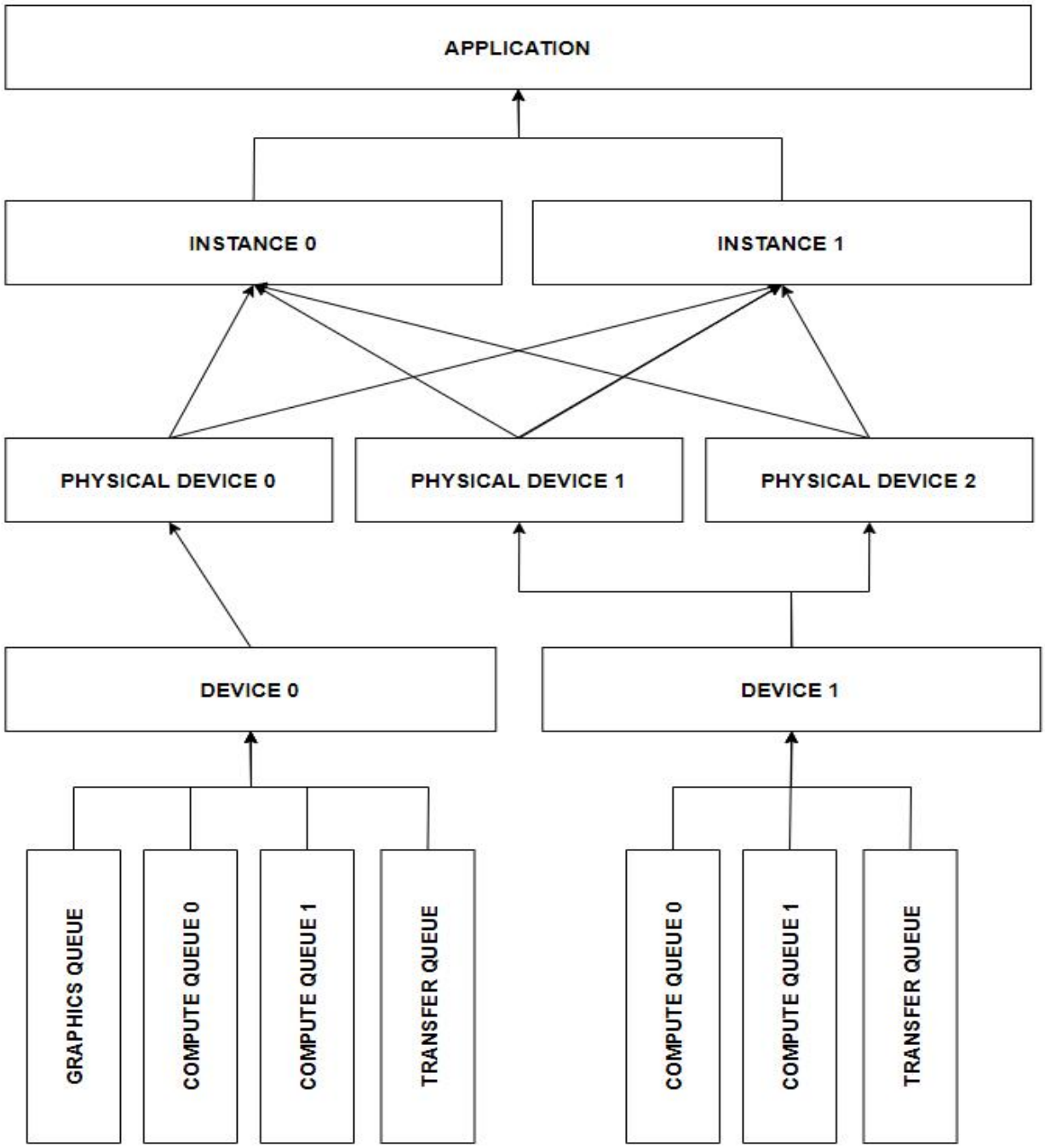

Рисунок 1.4 – Ієрархія суб’єктів Vulkan

Приклад створення VkInstance:

```
VkApplicationInfo applicationInfo{};
applicationInfo.pApplicationName = desc.applicationName;
applicationInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
applicationInfo.pEngineName = "<your_engine_name>";
applicationInfo.engineVersion = VK_MAKE_VERSION(0, 0, 1);
applicationInfo.apiVersion = VK_API_VERSION_1_0;
```

```
::VkInstanceCreateInfo instanceCreateInfo{};
instanceCreateInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
instanceCreateInfo.ppEnabledExtensionNames = nullptr;
instanceCreateInfo.enabledExtensionCount = 0;
instanceCreateInfo.ppEnabledLayerNames = nullptr;
instanceCreateInfo.enabledLayerCount = 0;
instanceCreateInfo.pApplicationInfo = &applicationInfo;

VkInstance instance{ nullptr };
auto const result = vkCreateInstance(
    &instanceCreateInfo,
    nullptr,
    &instance);

if (result != VK_SUCCESS)
{
// Handle error
}
```

Приклад 1.1 – Створення VkInstance

Фізичний пристрій може надати всі доступні типи сімейства черг. Черга є головною, але у вас також можуть бути додаткові компоненти, які підтримують лише паралельні розрахунки або трансформацію даних[9].

Фізичний пристрій також може надати типи пам’яті та типи пам’яті, котрі він має. Пам’ять купи являє собою конкретний пул оперативної пам’яті. Пул може абстрагувати наявну системну RAM на материнській платі або певний об’єм пам’яті у відеопам’яті на графічному процесорі або будь-якій іншій пам’яті хоста або пристрою, яку хоче виставити програма. Виділяючи пам’ять, потрібно вказати тип пам’яті. Тип пам’яті містить спеціальні вимоги до блоку пам’яті, як це бачать власники, узгоджене (між центральним процесором і графічним процесором) та кешування. Там може бути довільна їх комбінація, залежно від драйвера пристрою. Приклад перерахування фізичних пристроїв:

```
std::uint32_t physicalDeviceCount{ 0 };
vkEnumeratePhysicalDevices(instance, &physicalDeviceCount, nullptr);

auto const memSize = sizeof(VkPhysicalDevice) * physicalDeviceCount;
auto const memAlign = alignof(VkPhysicalDevice);

auto                          physicalDevices                          =
static_cast<VkPhysicalDevice*>(aligned_alloc(memSize, memAlign));
```

```
vkEnumeratePhysicalDevices(
    instance, &physicalDeviceCount, physicalDevices);
```

Приклад 1.2 – Приклад перерахування фізичних пристроїв

Черга (VkQueue) – це об’єкт, що представляє собою чергу команд, що виконуються на пристрої. Всі фактичні роботи, які потрібно виконати на графічному процесорі, запитуються заповненням буферу команд та поданням їх до черг, використанням функції vkQueueSubmit. Для випадку наявності кількох черг у створюваному додатку, таких як головна графічна черга та обчислювальна черга, існує можливість надсилання кожній з них різних буферів команд[11].

Таким чином, з’являється можливість увімкнення асинхронного обчислення, що може призвести до істотного зростання швидкості виконання, якщо це буде зроблено правильно. Приклад отримання черги з логічного пристрою:

```
::VkQueue queue{ nullptr };
::vkGetDeviceQueue(device, queueFamilyIndex, queueIndex, &queue);
```

Приклад 1.3 – Приклад отримання черги з логічного пристрою

Логічний пристрій (VkDevice) є головним об’єктом, який створює всі інші об’єкти. Це подібне поняття для об’єкта Device у Direct3D. Під час створення пристрою вам потрібно вказати, які функції ви хочете включити, і деякі з них є фундаментальними, як анізотропна фільтрація текстури. Ви також повинні вказати всі черги, які будуть використовуватися, їх кількість та їхні сімейства. Приклад створення логічного пристрою:

```
::VkResult result{};
::VkPhysicalDeviceFeatures supportedFeatures{};
::VkPhysicalDeviceFeatures requiredFeatures{};
```

```
::vkGetPhysicalDeviceFeatures(physicalDevices[0], &supportedFeatures);
requiredFeatures.multiDrawIndirect =
    supportedFeatures.multiDrawIndirect;
requiredFeatures.tessellationShader = VK_TRUE;
requiredFeatures.geometryShader = VK_TRUE;
const ::VkDeviceQueueCreateInfo deviceQueueCreateInfo
{
    VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO,
    nullptr,
    0,
    0,
    1,
    nullptr
};
const ::VkDeviceCreateInfo deviceCreateInfo
{
    VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO,
    nullptr,
    0,
    1,
    &deviceQueueCreateInfo,
    0,
    nullptr,
    0,
    nullptr,
    &requiredFeatures
};
::VkDevice device{ nullptr };
::vkCreateDevice(physicalDevices[0],
    &deviceCreateInfo,
    nullptr,
    &device);
```

Приклад 1.4 – Приклад створення логічного пристрою

Пул команд (VkCommandPool) – це простий об’єкт, який використовується для виділення командних буферів. Пул команд пов’язаний з певним сімейством черг.

Командний буфер (VkCommandBuffer) виділяється із пулу команд. Він являє собою буфер різних команд для виконання логічним пристроєм. Ви можете викликати різні функції в командному буфері, всі вони починаються з vkCmd<OperationName>. Вони використовуються для того, щоб вказати порядок, тип та параметри завдань, які повинні виконуватися, коли командний буфер передається до черги і нарешті виконується пристроєм.

Семплер (VkSampler) не пов’язаний ні з одним конкретним зображенням. Це набір параметрів стану, таких як режим фільтрування (найближчий чи лінійний) або режим адресації (повторити, затискати до

краю, затискати до межі тощо).

Vulkan має два типи основних ресурсів: буфер та зображення. Також є спеціальні підтипи ресурсів як Sparse Resource та Protected. Ці специфічні типи вимагають спеціальні тип пам’яті та апаратного забезпечення.

Буфер (VkBuffer) – найпростіший ресурс. Це контейнер для будь-яких бінарних даних, що мають лише довжину, виражену в байтах.

Зображення (VkImage) являє собою набір пікселів. Цей об’єкт відомий в інших графічних API як текстура. Є багато інших параметрів, необхідних для визначення створення зображення. VkImage може бути одновимірним, двовимірним або тривимірним , мати різні формати пікселів (наприклад, R8G8B8A8_UNORM або R32_SFLOAT), а також може складатися з багатьох дискретних зображень, оскільки може мати кілька шарів масивів або рівнів MIP (або обидва). Зображення являє собою окремий тип об’єкта, оскільки воно не обов’язково складається з простого лінійного набору пікселів, до яких можна звернутися безпосередньо. Зображення можуть мати інший внутрішній формат, який керується графічним драйвером.

Створення буфера певної довжини або зображення з певними розмірами автоматично не виділяє пам’ять для нього. Це процес із трьох етапів, який потрібно виконувати вручну:

- виділити пам’ять на GPU (VkDeviceMemory);
- створити буфер або зображення;
- зв’язати їх разом за допомогою функції vkBindBufferMemory або vkBindImageMemory.

Через наявність таких особливостей створення буфера виникає природна необхідність створити об’єкт типу VkDeviceMemory. Такий об’єкт являє собою блок пам’яті, виділений з певного типу пам’яті (як це підтримується VkPhysicalDevice) з певною довжиною в байтах. Суттєве зауваження: у розробника немає потреби виділяти окремий VkDeviceMemory для кожного буфера або зображення . Замість цього слід виділити більші шматки пам’яті та застосувати їх частинами для своїх буферів і зображень.

Виділення пам’яті є дорогою операцією, існує обмеження на максимальну кількість виділень, всі з яких можуть бути виконані вашим фізичним пристроєм.

Буфери й зображення не завжди використовуються безпосередньо у візуалізації. На вершині їх є ще один шар, званий як View. BufferView – це об’єкт, створений на основі конкретного буфера. Під час створення ви можете передавати зміщення та діапазон, щоб обмежити перегляд лише підмножиною буферних даних. Аналогічно, ImageView – це набір параметрів, що відносяться до певного зображення. Тут ви можете інтерпретувати пікселі як такі, що мають інший (сумісний) формат, закривати будь-які компоненти та обмежувати перегляд певному діапазону рівнів MIP або шарів масивів.

Шляхи доступу до цих ресурсів (буферів, зображень та пробників) здійснюються через дескриптори. Дескриптори не існують самостійно, але завжди згруповані в набори дескрипторів. Але перш ніж створювати набір дескрипторів, його макет повинен бути вказаний шляхом створення VkDescriptorSetLayout, який веде себе як шаблон для набору дескрипторів.

Вам також потрібно створити VkDescriptorPool. Це простий об’єкт, який використовується для виділення наборів дескрипторів. Під час створення пулу дескрипторів необхідно вказати максимальну кількість дескрипторних наборів та дескрипторів різних типів, які ви збираєтесь виділити з нього.

Нарешті, створюємо VkDescriptorSet. Вам потрібно як VkDescriptorPool, так і VkDescriptorSetLayout. VkDescriptorSet являє собою пам’ять, яка містить фактичні дескриптори, і її можна налаштувати таким чином, щоб дескриптор вказував на конкретний буфер, буфер зображення, зображення або пробник, використовуючи функцію vkUpdateDescriptorSets.

Кілька VkDescriptorSets можуть бути пов’язані як активні набори в командному буфері, які будуть використовуватися при відтворенні команд. Для цього використовуйте функцію vkCmdBindDescriptorSets. Ця функція

потребує іншого об’єкта – VkPipelineLayout, оскільки може бути декілька пов’язаних VkDescriptorSets, і Vulkan хоче заздалегідь знати, скільки і які типи їх слід очікувати. VkPipelineLayout являє собою конфігурацію конвеєра рендерингу з точки зору того, які типи дескрипторних наборів будуть пов’язані з командним буфером. Ви створюєте його з масиву VkDescriptorSetLayouts.

В інших графічних інтерфейсах ви можете скористатись найближчим підходом до режиму та просто перетворювати все, що відбувається у вашому списку. Це неможливо у Vulkan. Замість цього вам потрібно заздалегідь планувати рендеринг кадру та організувати його в пасах (Pass) та субпасах (Subpass).

Буфер кадру (VkFramebuffer) являє собою посилання на фактичні зображення, які можна використовувати як вкладення для рендерингу. Ви створюєте об’єкт VkFramebuffer, вказавши VkRenderPass і набір VkImageView. Звичайно, їхні номери та формати повинні відповідати специфікації VkRenderPass. VkFramebuffer – це ще один шар у верхній частині зображень і в основному об’єднує ці VkImageView разом, щоб бути пов’язаними як вкладення під час відтворення певного VkRenderPass. Кожного разу, коли ви починаєте рендеринг, потрібно викликати функцію vkCmdBeginRenderPass та передати VkFramebuffer.

Стан конвеєру (VkPipeline) є великим суб’єктом, оскільки він складається з більшості об’єктів, перерахованих вище. Він являє собою конфігурацію всього ковеєру і має багато параметрів. Одним з них є VkPipelineLayout. Є два типи конвеєру – Compute та Graphics. Compute – це найпростіший спосіб, оскільки всі підтримувані програми – це лише обчислювальні програми (іноді їх називають обчислювальними шейдерами). Graphics набагато складніший, оскільки він включає в себе всі параметри, такі як вершина, фрагмент, геометрія, обчислення та тесселяція, а також такі речі, як атрибути вершин, топологія, вилучення на задній поверхні та режим накладання. Всі ті параметри, що раніше використовувалися для окремих

налаштувань у багатьох застарілих графічних інтерфейсах (DirectX 9, OpenGL), згодом були згруповані у меншу кількість головних об’єктів, коли проходили API (DirectX 10 та 11). Для кожного окремого набору параметрів, необхідних під час рендеринга, потрібно створити новий конвеєр. Потім ви можете встановити його за допомогою функції vkCmdBindPipeline.

Компіляція шейдерів – це багатоетапний процес у Vulkan. По-перше, Vulkan не підтримує жодної шейдерної мови високого рівня, такої як GLSL або HLSL. Замість цього, Vulkan приймає проміжний формат SPIR-V, який може випускати будь-яка мова вищого рівня. Буфер, заповнений даними в SPIR-V, використовується для створення VkShaderModule. Цей об’єкт являє собою частину шейдерного коду, але все це не може виконується графічний процесор. Лише при створенні конвеєру для кожного типу шейдера, ви вказуєте VkShaderModule плюс назву функції точки вводу (наприклад, «main»).

Фенс (VkFence) повідомляє хосту, що завершення виконання завдання. Його можна чекати, опитати та вручну не сигналізувати на хості. Він не має своєї командної функції, але він проходить при виклику vkQueueSubmit. Після того, як представлена черга заповнює відповідний фенс, подано сигнал.

Семафор (VkSemaphore) використовується для керування доступом до ресурсів у кількох чергах. Про це можна сигналізувати або чекати як частину подання командного буфера, через функцію vkQueueSubmit, і це може бути сигналізовано в одній черзі (наприклад, compute) і очікується на інші (наприклад, graphics).

Подія (VkEvent) також створюється без параметрів. Її можна очікувати або повідомити на GPU окремо, передану в командний буфер, використовуючи функції vkCmdSetEvent, vkCmdResetEvent і vkCmdWaitEvents. VkCmdPipelineBarrier також може бути використаний для подібної мети, якщо синхронізація відбувається в одній точці на графічному процесорі.

1.5 Існуючий інструментарій Vulkan

Найвідоміший інструмент Vulkan є Vulkan SDK. Він має багато компонентів для розробки, таких як Vulkan Loader (прямий зв’язок під час компіляції, тому вам не потрібно завантажувати функціональність Vulkan вручну), шари перевірки, Vulkan Runtime (VulkanRT), інструменти SPIR-V, інструменти трасування та інше. Vulkan SDK підтримується на багатьох платформах і системах[1, 6, 11].

Також існує порт-реалізація MoltenVK, що імплементує API Vulkan для платформ macOS і iOS на базі технології Apple Metal.

Існує також безліч службових бібліотек, таких як мовна прив’язка або оболонки. Одним з них є Vulkan-Hpp, який забезпечує прив’язку мови програмування C++ для Vulkan C API. Він додає так званий синтаксичний цукор C++, такий як безпеку типів, використання контейнерів зі стандартної бібліотеки шаблонів STL, виключення, більш простий перелік та інше. Умовно це байдинг (binding) Vulkan API C до мови програмування C++.

1.6 Приклад Vulkan-додатку

Для кращого розуміння роботи Vulkan API пропоную розглянути базову реалізацію додатку, котрий створює головний контекст та опитує характеристики графічних адаптерів в системі.

На початку програми потрібно завантажити динамічний модуль, який є точкою входу основного завантажувача (Vulkan Loader). У кожній операційній системі цей модуль має різну назву, в Windows він називається vulkan-1.dll. З цього модуля ми маємо можливість завантажити функції для виконання додатку:

```
#include <Windows.h>
#include <vulkan/vulkan.h>

int main(int argc, const char** argv)
{
    auto vulkanModule = ::LoadLibrary(«vulkan-1.dll»);
    assert(vulkanModule);

    auto pfnGetInstanceProcAddr = (::PFN_vkGetInstanceProcAddr)
        ::GetProcAddress(vulkanModule, «vkGetInstanceProcAddr»);
    auto pfnCreateInstance = (::PFN_vkCreateInstance)
        pfnGetInstanceProcAddr(nullptr, «vkCreateInstance»);

    return 0;
}
```

Приклад 1.5 – Завантаження функцій завантажувача.

Для запиту доступних графічних адаптерів за допомогою Vulkan потрібно створити головний контекст, який виражений об’єктом VkInstance:

```
#include <Windows.h>
#include <vulkan/vulkan.h>

int main(int argc, const char** argv)
{
    auto vulkanModule = ::LoadLibrary(«vulkan-1.dll»);
    assert(vulkanModule);

    auto pfnGetInstanceProcAddr = (::PFN_vkGetInstanceProcAddr)
        ::GetProcAddress(vulkanModule, «vkGetInstanceProcAddr»);
    auto pfnCreateInstance = (::PFN_vkCreateInstance)
        pfnGetInstanceProcAddr(nullptr, «vkCreateInstance»);
    ::VkApplicationInfo applicationInfo
    {
        VK_STRUCTURE_TYPE_APPLICATION_INFO
    };
    applicationInfo.pApplicationName = «<your_application_name>«;
    applicationInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
    applicationInfo.pEngineName = «<your_engine_name>«;
    applicationInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
    applicationInfo.apiVersion = VK_MAKE_VERSION(1, 0, 0);
    ::VkInstanceCreateInfo instanceCreateInfo
    {
        VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO
    };
    instanceCreateInfo.pApplicationInfo = &applicationInfo;

    ::VkInstance instance{ nullptr };
    const ::VkResult result = pfnCreateInstance(
        &instanceCreateInfo, nullptr, &instance);

    return 0;
}
```

Приклад 1.6 – Створення VkInstance

За допомогою створеного контекста спробуємо запитати список графічних адаптерів та вивести ім’я кожного адаптера. Для цього існує 2 функції vkEnumeratePhysicalDevices та vkGetPhysicalDeviceProperties:

```
#include <Windows.h>
#include <vulkan/vulkan.h>

int main(int argc, const char** argv)
{
    auto vulkanModule = ::LoadLibrary(«vulkan-1.dll»);
    assert(vulkanModule);

    auto pfnGetInstanceProcAddr = (::PFN_vkGetInstanceProcAddr)
        ::GetProcAddress(vulkanModule, «vkGetInstanceProcAddr»);
    auto pfnCreateInstance = (::PFN_vkCreateInstance)
        pfnGetInstanceProcAddr(nullptr, «vkCreateInstance»);

    ::VkApplicationInfo applicationInfo
    {
        VK_STRUCTURE_TYPE_APPLICATION_INFO
    };
    applicationInfo.pApplicationName = «<your_application_name>«;
    applicationInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
    applicationInfo.pEngineName = «<your_engine_name>«;
    applicationInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
    applicationInfo.apiVersion = VK_MAKE_VERSION(1, 0, 0);

    ::VkInstanceCreateInfo instanceCreateInfo
    {
        VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO
    };
    instanceCreateInfo.pApplicationInfo = &applicationInfo;

    ::VkInstance instance{ nullptr };
    const ::VkResult result = pfnCreateInstance(
        &instanceCreateInfo, nullptr, &instance);

    auto pfnEnumeratePhysicalDevices =
        (PFN_vkEnumeratePhysicalDevices)
            pfnGetInstanceProcAddress(
                instance, «vkEnumeratePhysicalDevices»);
    auto pfnGetPhysicalDeviceProperties =
        (PFN_vkGetPhysicalDeviceProperties)
            pfnGetInstanceProcAddress(
                instance, «vkGetPhysicalDeviceProperties»);

    ::uint32_t physicalDeviceCount{ 0u };
    pfnEnumeratePhysicalDevices(
        instance, &physicalDeviceCount, nullptr);
    auto physicalDevices = (::VkPhysicalDevice*)malloc(
        sizeof(::VkPhysicalDevice) * physicalDeviceCount);
    pfnEnumeratePhysicalDevices(
        instance, &physicalDeviceCount, physicalDevices);

    for (::uint32_t it{ 0u }; it != physicalDeviceCount; ++it)
    {
        ::VkPhysicalDeviceProperties properties{};
        pfnGetPhysicalDeviceProperties(
```

```
                physicalDevices[it], &properties);

            ::std::cout << «GPU[« << it << «]: « << properties.deviceName;
        }

        ::free(physicalDevices);
        ::FreeLibrary(vulkanModule);

        return 0;
    }
```

Приклад 1.7 – vkEnumeratePhysicalDevices та vkGetPhysicalDeviceProperties

У кінці додатку не забуваємо звільнити контекст та ресурси. Особливу увагу притягує подвійний виклик функції vkEnumeratePhysicalDevices. Перший виклик дає можливість дізнатися кількість об’єктів; далі розробник повинен виділити достатню ділянку пам’яті для другого виклику функції vkEnumeratePhysicalDevices, щоб драйвер та завантажувач Vulkan мали можливість записати потрібну інформацію про графічні адаптери. Це дуже поширений дизайн для низькорівневих програмних інтерфейсів, шо надає розробнику додатку самому обирати спосіб виділення пам’яті, Vulkan API не є виключенням[9]. Код додатку добре демонструє екосистему та гнучкість Vulkan. Але також демонструє, що розробник має виконувати перед кожною командою підготовку потрібного функціоналу, шо не є дуже зручним для розробки кінцевого продукту, тому більшість розробників використовують Vulkan SDK, котрий бере на себе відповідальність про завантаження функціоналу, але ціною невеликого навантаження на центральний процесор (CPU) в момент виклику функцій[11]. Подібних випадків Vulkan має дуже багато, тому що головною метою цієї технології є продуктивність та гнучкість, тому розробник додатку має взяти відповідальність за контроль завантаження функціоналу для досягнення більшої продуктивность додатку.

На грунті цих проблем, котрі спричиняють проблеми контролю функціоналу виникла ідея створення самостійної технології, котра внесе новий погляд та дизайн на розробку високочутливого програмного забезпечення до швидкості виконання, особливо для систем реального масштабу часу[13].

2 РЕАЛІЗАЦІЯ VULKANKIT

VulkanKit – це експериментальна розробка, яка представляє новий програмний інтерфейс та його реалізацію з відкритим програмним кодом. Першочерговою ціллю даної технології є надання зручного інтерфейсу та інструментів для розробки програмного забезпечення, а також зручного та мінімалістичного інструментарію для відладки, трейсингу, динамічного розпізнавання функціоналу, бо кожна графічна система має свої налаштування, характеристики, тощо. Це допоможе стороннім розробникам зосередитися на реалізації важливого функціоналу – реалізації масивних обчислень на графічному адаптері, або адаптерах. Тому це також вимагає від VulkanKit мінімальні накладні витрати.

VulkanKit спроектований таким чином, що розробник має можливість використовувати VulkanKit як статичну бібліотеку та також як динамічну. Така варіативність використання даної бібліотеки надає розробникам самостійно вирішувати шлях завантаження функціоналу VulkanKit. При цьому розробнику додатку не потрібно змінювати основний код додатку.

Як і кожна самостійна програмна технологія, VulkanKit несе свою філософію та підхід в програмуванні. VulkanKit має гарну модульну та компонентну структуру, що надає більш чисте розуміння взаємодії тих чи інших суб’єктів в модулі та самого Vulkan. Давайте розглянемо більш детально.

2.1 Вибір інструментів

Головною мовою програмування було обрано C++. Сама C++ на сьогодні залишається однією із популярних мов програмування для низькорівневих рішень після C. В порівнянні з мовою програмування C, C++ має свої переваги та унікальний функціонал. Сюди входять об’єктно-

орієнтоване програмування (класи, методи, рівні доступу), шаблони, метапрограмування, автоматизоване управління життям об’єктів (ідіома RAII), функтори, безпека типів на етапі компіляції та безпечне приведення типів, простір імен. VulkanKit використовує та потребує C++ версії 17. На сьогодні це остання та стабільна версія C++ на всіх компіляторах.

Оскільки VulkanKit це проект кросплатформний, модульний та обіцяє вийти в open-source, то потребує загального рішення для збірки на тій чи іншій платформі. Тому було обрано систему CMake.

CMake – це кросплатформне сімейство інструментів з відкритим програмним кодом, призначене для створення, тестування і упаковки програмного забезпечення. CMake використовується для управління процесом компіляції програмного забезпечення з використанням простих файлів конфігурації, які не залежать від платформи і компілятора, і для створення власних make-файлів і проектних файлів, які можна використовувати в середовищі компілятора за вашим вибором. Набір інструментів CMake був створений Kitware. На сьогодні це одна з найпопулярніших систем для генерації та збірки кросплатформних проектів на C++ в особливості[4]. Він використовується разом з нативними середовищами розробки програмного забезпечення, такими як Make, Qt Creator, Ninja, Xcode Apple, Microsoft Visual Studio та інші. Він має мінімальні залежності, такі як лише компілятор C++ у власній системі розробки.

2.2 Дизайн модулів VulkanKit

Як вже було описано вище, VulkanKit має модульний та субмодульний дизайн. Кожний модуль має субмодулі (у свою чергу субмодуль може мати субмодулі також), інтерфейси суб’єктів та компоненти. Усі модулі та субмодулі поділяються на три класи: публічний, приватний, гібридний. До класу публічних модулів відносяться усі модулі, котрі відображають

загальний інтерфейс взаємодії з технологією VulkanKit та Vulkan. Приватний клас модулів містить усі інтерфейси та компоненти, котрі виконують мету імплементації для VulkanKit.

В VulkanKit було встановлено конвенцію реалізації модулів. Кожному модулю та субмодулю відповідає свій простір імен (namespace) та директорія в файловій системі проекту:

```
      #include <VulkanKit/API/Core/PhysicalDevice.hpp>
      #include <VulkanKit/API/Core11/PhysicalDevice.hpp>
      ...

      ::VulkanKit::API::Core::PhysicalDevice* physicalDevice10 =
          instance->GetPhysicalDevices()[0u];
::VulkanKit::API::Core11::PhysicalDevice* physicalDevice11 =
          physicalDevice10->As<::VulkanKit::API::Core11::PhysicalDevice>();
```

Приклад 2.1 – API VulkanKit

У перших ітераціях проектування, VulkanKit мав чотири головних модуля: Core, Extensions, Utility, System. Модуль Core містив усі інтерфейси, котрі описують функціонал взаємодії з суб’єктами Vulkan 1.0. Модуль Extensions містив субмодулі з інтерфейсами розширень для субєктів Vulkan 1.0. Модуль Utility це библиотека службових класів та шаблонів таких як смарт поінтерів, строки, базових алгоритмів та контейнерів. Модуль System мав компоненти, які імплементують платформо-незалежні суб’єкти для роботи з сервісами операційної системи.

В кінцевому варіанті було прийнято більш глибокий дизайн. Було впроваджено модуль API, котрий тримає субмодулі Core, Core11, Core12, Extensions (рисунок 2.1). Як можна побачити, модулі Utils (Utility) та System залишилися без змін, але Core потерпів де-яких змін (занесено до модулю API). Також було додано субмодулі Core11 та Core12, котрі мають інтерфейси для функціоналу Vulkan Core 1.1 та 1.2 відповідно. Це надає більш чіткої структури з роздробленням інтерфейсів по версіям ядра Vulkan та його розширенням для кращого сприйняття, того який функціонал

доступний на тій чи іншій системі. У свою чергу модуль API має базові інтерфейси та абстракції, котрі використовуються в його субмодулях Core, Core11, Core12, Extensions.

2.3 Мультиінтерфейс

Для початку трохи поговоримо про те як саме здійснюється взаємодія з суб’єктами Vulkan. Умовно існує два способи для взаємодії з суб’єктами: через вказівник (Handle) та порядковий індекс. Взаємодія через вказівник здійснюється з явними суб’єктами, тобто під час створення об’єкта того чи іншого суб’єкта розробник отримує вказівник. Цей підхід самий поширений в Vulkan API.

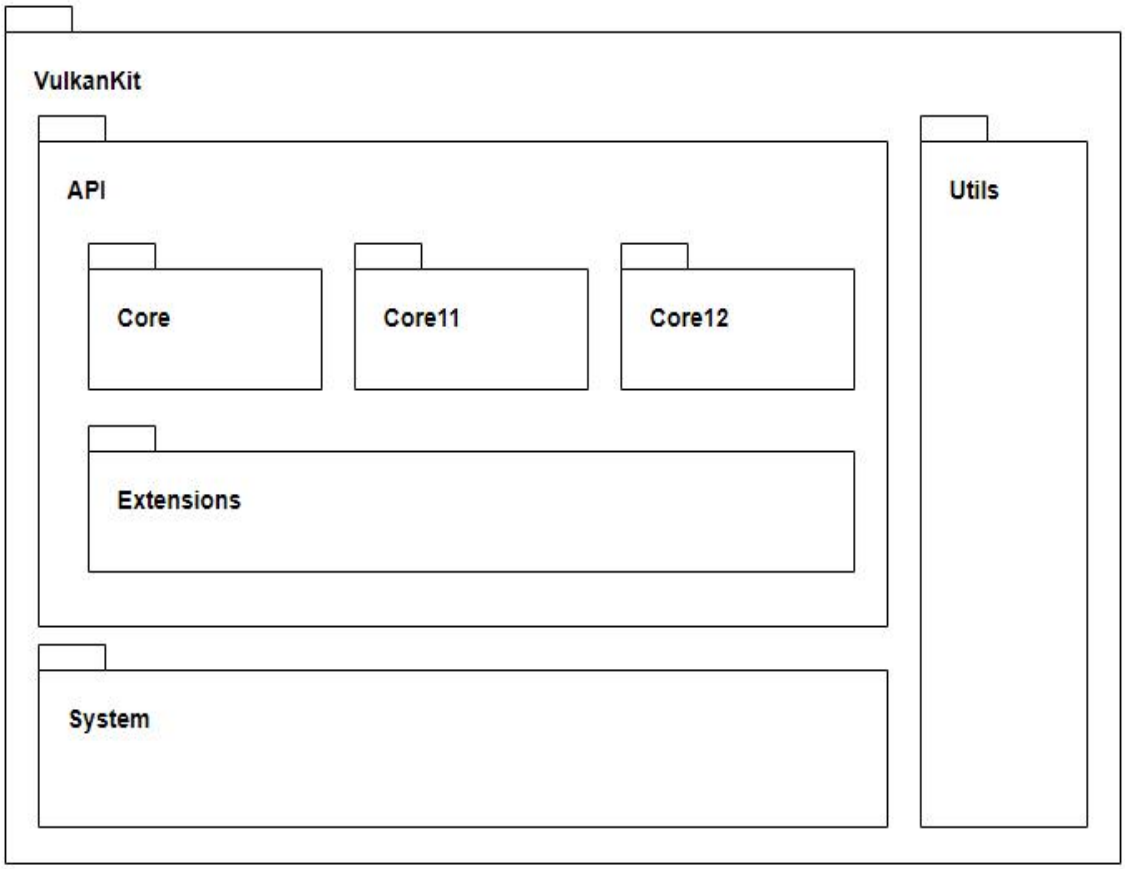

Рисунок 2.1 – Діаграма модулів VulkanKi

Розглянемо приклад з використанням вказівників в момент створення VkInstance:

```
::VkInstanceCreateInfo instanceCreateInfo{};
instanceCreateInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
::VkInstance instance = nullptr;
const VkResult result = vkCreateInstance(
    &instanceCreateInfo, nullptr, &instance);
```

Приклад 2.2 – Приклад з використанням вказівників в момент створення VkInstance

У свою чергу, порядкові індекси використовуються для неявних суб’єктів таких як QueueFamily, MemoryHeap, MemoryType, Query. Розглянемо приклад з використанням QueueFamily в момент створення черги:

```
uint32_t FindGraphicsQueueFamily(VkPhysicalDevice device)
{
std::vector<::VkQueueFamilyProperties> properties{};
vkGetPhysicalDeviceQueueFamilyProperties(device, ...)

uint32_t index{ 0u };
for (index; index != properties.size(); ++index)
{
      if                    (IsQueueFamilyCompatible(properties[index])
      {
      return index;
}
}

      return                                                          ~0u;
}

::VkDeviceQueueCreateInfo queueCreateInfo{};
      queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
      queueCreateInfo.queueFamilyIndex =
          FindGraphicsQueueFamily(physicalDevice);
```

Приклад 2.3 – Приклад з використанням QueueFamily

Як ми можемо бачити з даного прикладу, використання даного методу не є очевидним навіть для більшості досвідчених розробників.

VulkanKit впроваджує новий підхід та дизайн взаємодії з суб’єктами –

інтерфейси. Інтерфейс (чи протокол) з точки зору об’єктно-орієнтованого програмування та дизайну – це опис дій, які об’єкт може виконувати. Наприклад, коли ви натискаєте вимикач світла, світло вимикається, вам все одно, як, просто так і відбувається. Це один із способів описання абстракцій в мовах програмування, котрі мають об’єктно–орієнтовану філософію програмування. В мовах програмування такі як Java, Swift, C# інтерфейси відкривають великі можливості в проектуванні та імплементації програмного забезпечення. Клас має можливість реалізувати декілька інтерфейсів (мультиінтерфейс). Але мова програмування C++ не має такого поняття як інтерфейс на рівні синтаксису мови. Існує ідіома програмування на C++ як Pure Virtual Interface. Це клас, котрий має лише абстрактні методи (тобто не мают реалізації). Але все ж таки реалізувати одразу декілька таких “інтерфейсів” в одному класі через наслідування в мові програмування C++ це небезпечна справа. Чому? Існує проблема ромбовидного наслідування, котра в мові C++ призводить до неконтрольованих результатів, в залежності від вибору компілятора.

Це призвело до дослідження існуючих архітектурних рішень та створення дизайну для вирішення даної задачі. Після декількох ітерацій проектування, реалізації прототипів та тестування, було прийнято рішення реалізувати мультиінтерфейсний дизайн на основі композиції. Ключова ідея полягає в тому, що кожен інтерфейс реалізується в окремому класі, котрі у свою чергу компонуються в головному класі суб’єкта (рисунок 2.2). Схожий підхід використовується в Component Object Model (COM) від Microsoft та CORBA. Кожен об’єкт-реалізація має вказівник на головний об’єкт, що надає можливість об’єктам–реалізаціям створювати поведінку як одного цілого суб’єкта.

Так, як можна прийняти до уваги той факт, що виклик віртуального методу в C++ є трохи повільнішим, а ніж прямий виклик методу, або функції. В середньому виклик віртуального методу додає 2-4 асемблерні команди (це залежить від того як викликають метод та компілятора).

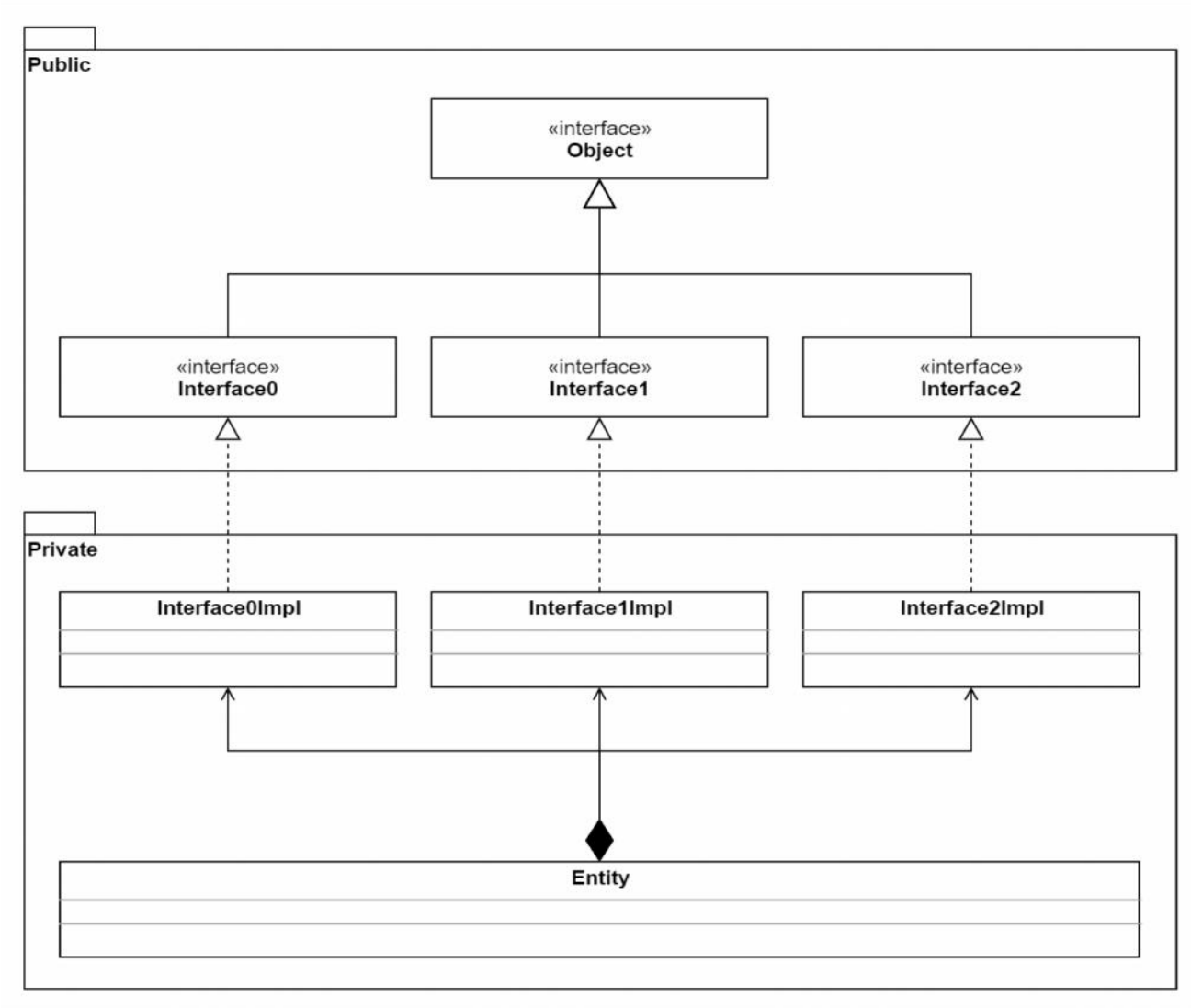

Рисунок 2.2 – Концепція імплементації мультиінтерфейсу

Але як вже було сказано вище, подібний підхід використовується в Component Object Model для DirectX та також на базі даного архітектурного рішення був реалізований програмний інтерфейс драйверів для платформ MacOS та iOS від компанії Apple, шо добре демонструє даний підхід дуже ефективним в проектуванні та імплементації низькорівневих систем та систем реального масштабу часу.

На діаграмі класів можна побачити, що інтерфейси одного суб’єкта знаходяться на одному рівні. В момент проектування додатку може виникнути потреба використати суб’єкт в іншому інтерфейсі. Стандартними операторами приведення типів в мовах програмування C та C++ не є

безпечним варіантом використання. Тому було додано в базовий інтерфейс ::VulkanKit::API::Object метод As(). Він має 2 варіації: приватну та публічну:

```
class VULKANKIT_INTERFACE Object : Utils::Noncopyable
{
public:
virtual ObjectType VULKANKIT_CALL GetObjectType() const noexcept = 0;
template < class Interface >
const Interface* As() const noexcept
{
   return static_cast<const Interface*>(As(Interface::InterfaceHash));
}

template < class Interface >
Interface* As() noexcept
{
   return static_cast<Interface*>(As(Interface::InterfaceHash));
}

private:
virtual const Object* VULKANKIT_CALL As(Word32 hash) const noexcept = 0;
virtual Object* VULKANKIT_CALL As(Word32 hash) noexcept = 0;
};
```

Приклад 2.4 – Інтерфейс VulkanKit::API::Object

Приватна варіація метода дає змогу отримати об’єкт по хешу інтерфейса. Публічний варіант отримує хеш інтерфейса на етапі компіляції та передає до приватного виклику метода As():

```
      ::VulkanKit::API::Core::PhysicalDevice* physicalDevice10 =
          instance->GetPhysicalDevices()[0u];
::VulkanKit::API::Core11::PhysicalDevice* physicalDevice11 =
          physicalDevice10->As<::VulkanKit::API::Core11::PhysicalDevice>();
```

Тому кожен інтерфейс має свій згенерований хеш у форматі CRC32:

```
class VULKANKIT_INTERFACE SomeInterface :
    public VulkanKit::API::Object
{
public:
    static constexpr Word32 InterfaceHash = 0x<Value>u;
};
```

Приклад 2.5 – Виклик метода As()

2.4 Архітектура компонентів VulkanKit

VulkanKit повторює ієрархію компонентів Vulkan (рисунок 2.3), але також впроваджує свої компоненти та переосмислює використання нативних компонентів Vulkan таких як QueueFamily, MemoryHeap, MemoryType, DeviceGroup, Query, вводячи для кожного з них свій інтерфейс.

Головним компонентом VulkanKit є Runtime. Введення даного компонента сприяло в результаті на створення нового рівня в організації ієрархії компонентів. Тому VulkanKit має чотирьох рівневу ієрархію в порівнянні з нативним Vulkan, котрий має лише три рівні.

Створення екземпляру компонента Runtime сприяє на завантаження драйвера Vulkan (якщо система не завантажила раніше) та завантажує таблицю функцій глобального рівня (GlobalDispatchTable). Runtime має функціонал для запиту повного списку розширень та плагінів для Instance, запиту актуальної версії Instance та створення самого Instance. Як опція, в момент створення Runtime, можна передати спеціалізований аллокатор пам’яті, котрий реалізує інтерфейс SystemAllocator. Інтерфейс SystemAllocator є доволі простим, має 3 ключові методи для алокації, реалокації та звільнення пам’яті. Цей інтерфейс надає змогу розробникам додатків створювати свою стратегію керування пам’яттю, особливо це критичне питання для вбудованих систем. Також розробники мають взяти відповідальність за вирівнювання пам’яті, в противному випадку це призведе до деградації продуктивності.

Компоненти VulkanKit мають одну особливість – зв’язок типу Parent-Child. Тобто кожен компонент тримає посилання на батьківський компонент (виключення Runtime). Такий дизайн програмного інтерфейсу достатньо відрізняється від нативного програмного інтерфейсу Vulkan. Розглянемо приклад взаємодії VkDevice та VkCommandPool, де перший є «батько», другий – «дитина»:

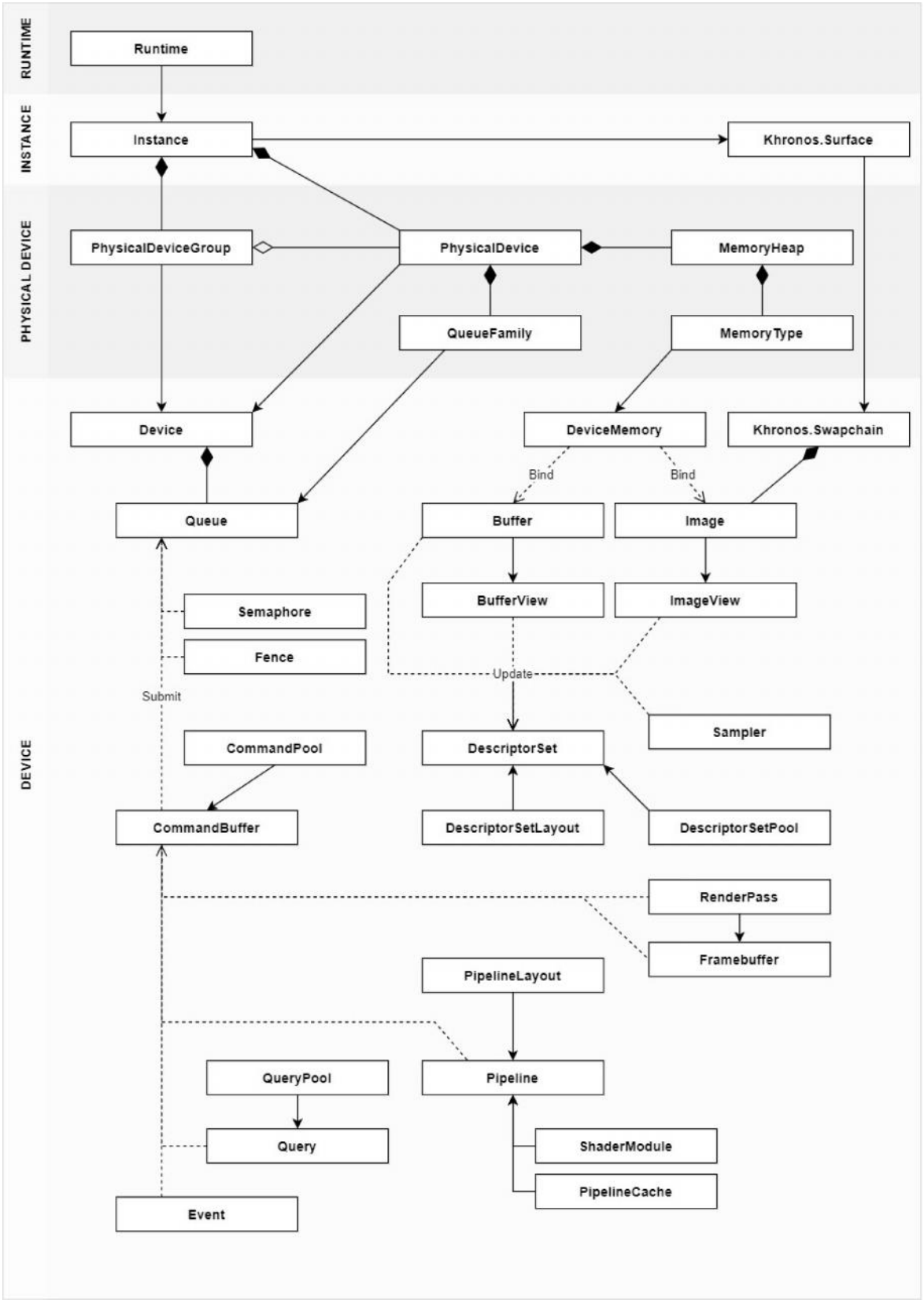

Рисунок 2.3 – Архітектура компонентів VulkanKit

```
      ::VkCommandPoolCreateInfo cmdPoolCreateInfo{};
      cmdPoolCreateInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
      cmdPoolCreateInfo.flags                                                 =
VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;

      ::VkCommandPool commandPool{ nullptr };
      vkCreateCommandPool(device, &cmdPoolCreateInfo, nullptr, &commandPool);

       // using device & command pool

vkDestroyCommandPool(device, commandPool, nullptr);
```

Приклад 2.6 – Створення Command Pool

Ми бачимо, створення та звільнення об’єкта вимагає об’єкт батьківського компонента, тому це створює ще одну відповідальність для розробника додатку в питанні менеджменту життя об’єктів.

VulkanKit вирішує це питання та бере на себе відповідальність за контроль взаємодії між компонентами. VulkanKit має 5 типів батьківських копонентів: Runtime, Instance, PhysicalDevice, Device, CommandPool. Для зворотної взаємодії з батьківськими компонентами VulkanKit впроваджує розширені інтерфейси:

- RuntimeSubObject;
- InstanceSubObject;
- PhysicalDeviceSubObject;
- DeviceSubObject;
- CommandPoolSubObject.

Кожен із вище сказаних інтерфейсів має таку структуру коду:

```
class VULKANKIT_INTERFACE <Component>SubObject :
    public VulkanKit::API::Object
{
public:
    virtual const <Component>& Get<Component>() const noexcept = 0;
    virtual <Component>& Get<Component>() noexcept = 0;
};
```

Приклад 2.7 – Шаблон інтерфейсів для субоб’єктів

В результаті створення такого дизайну, усі компоненти створюють

дерево залежностей, тому майже кожен компонент має доступ до верхніх компонентів по ієрархії.

Такий дизайн дає можливість переосмислити нативний інтерфейс Vulkan. Код для порівняння приведено нижче:

```
// Vulkan C
// VkDevice is a parent of VkSwapChainKHR
uint32_t outImageIndex{};
::vkAcquireNextImageKHR(device, swapchain,
    timeout, semaphore, nullptr,
    &outImageIndex);

// VulkanKit C++
// Swapchain keeps a reference to Device
currentImageIndex = swapchain->AcquireNextImage(timeout, semaphore,
nullptr);

або

Result result{};
currentImageIndex = swapchain->AcquireNextImage(timeout, semaphore,
nullptr, result);
```

Приклад 2.8 – Використання Swapchain в Vulkan та VulkanKit

Також спрощує звільнення об’єктів:

```
// Vulkan C
::VkSwapchainKHR swapchain{ nullptr };
::vkCreateSwapchainKHR(device, &swapchainCI, nullptr, &swapchain);

::vkDestroySwapchainKHR(device, &swapchain, nullptr);

// VulkanKit C++
auto swapchain = device->CreateSwapchain(swapchainCI);
swapchain = nullptr; // RAII idiom works
```

Приклад 2.9 – Створення Swapchain в Vulkan та VulkanKit

2.5 Таблиці функцій Vulkan

В даній роботі ми вже не раз бачили приклади використання функціоналу Vulkan шляхом завантаження функцій. Це є найефективніший спосіб використання функцій, але давайте розберемось як влаштований

функціонал Vulkan. Умовно весь функціонал Vulkan поділений на три рівні: глобальний рівень, рівень Instance, рівень Device (рисунок 2.4).

Для чого це зроблено? Відповідь полягає в тому, як ланцюжок викликів функцій Instance реалізується в порівнянні з ланцюжком викликів функцій Device. Vulkan Instance – це високорівнева конструкція, яка використовується для надання інформації на системному рівні Vulkan. Через це функції Instance необхідно транслювати на кожен доступний ICD в системі. На рисунку 2.5 показано приблизна конструкція ланцюжка викликів Instance з 3 включеними рівнями опціональних шарів.

Так само виглядає ланцюжок викликів функцій Vulkan Device, якщо ви запитуєте її за допомогою vkGetInstanceProcAddr. З іншого боку, функції Device не потрібно турбуватися про трансляцію, оскільки вона точно знає, на якому пов’язаному ICD і на якому фізичному пристрої повинен завершитися виклик. Через це завантажувач не повинен втручатися між будь-якими включеними шарами і ICD. Таким чином, якщо ви використовували функцію Vulkan Device, що експортується з завантажувача, ланцюжок викликів в тому ж сценарії, що і вище, буде виглядати так як на рисунку 2.6.

Ще кращим рішенням було б здійснити виклик vkGetDeviceProcAddr для всіх функцій Vulkan Device. Це додатково оптимізує ланцюжок викликів, видаляючи рівні завантажувачв в більшості сценаріїв (рисунок 2.7). Крім того, зверніть увагу, що якщо ніякі шари не включені, вказівник функції вашої програми буде вказувати безпосередньо на ICD. Якщо їх викликати досить, то менша кількість викликів може привести до підвищення продуктивності.

Тому для досягнення максимальної продуктивності, розробник має завантажувати потрібний функціонал на кожному рівні за допомогою нативних функцій vkGetInstanceProcAddr, vkGetDeviceProcAddr.

VulkanKit впроваджує на рівні імплементації три типи таблиць функцій: GlobalDispatchTable, InstanceDispatchTable, DeviceDispatchTable.

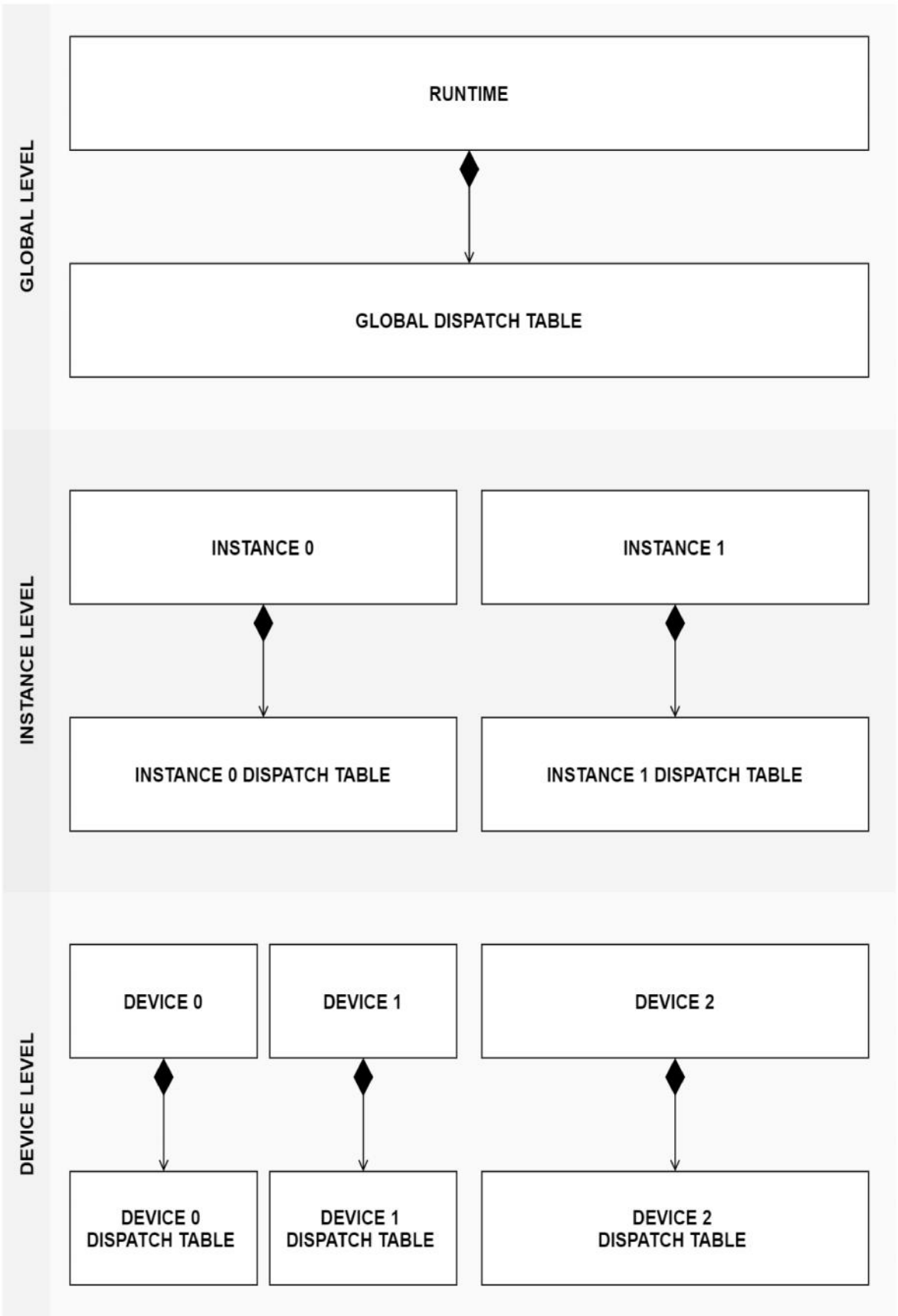

Рисунок 2.4 – Схема залежностей таблиць функцій

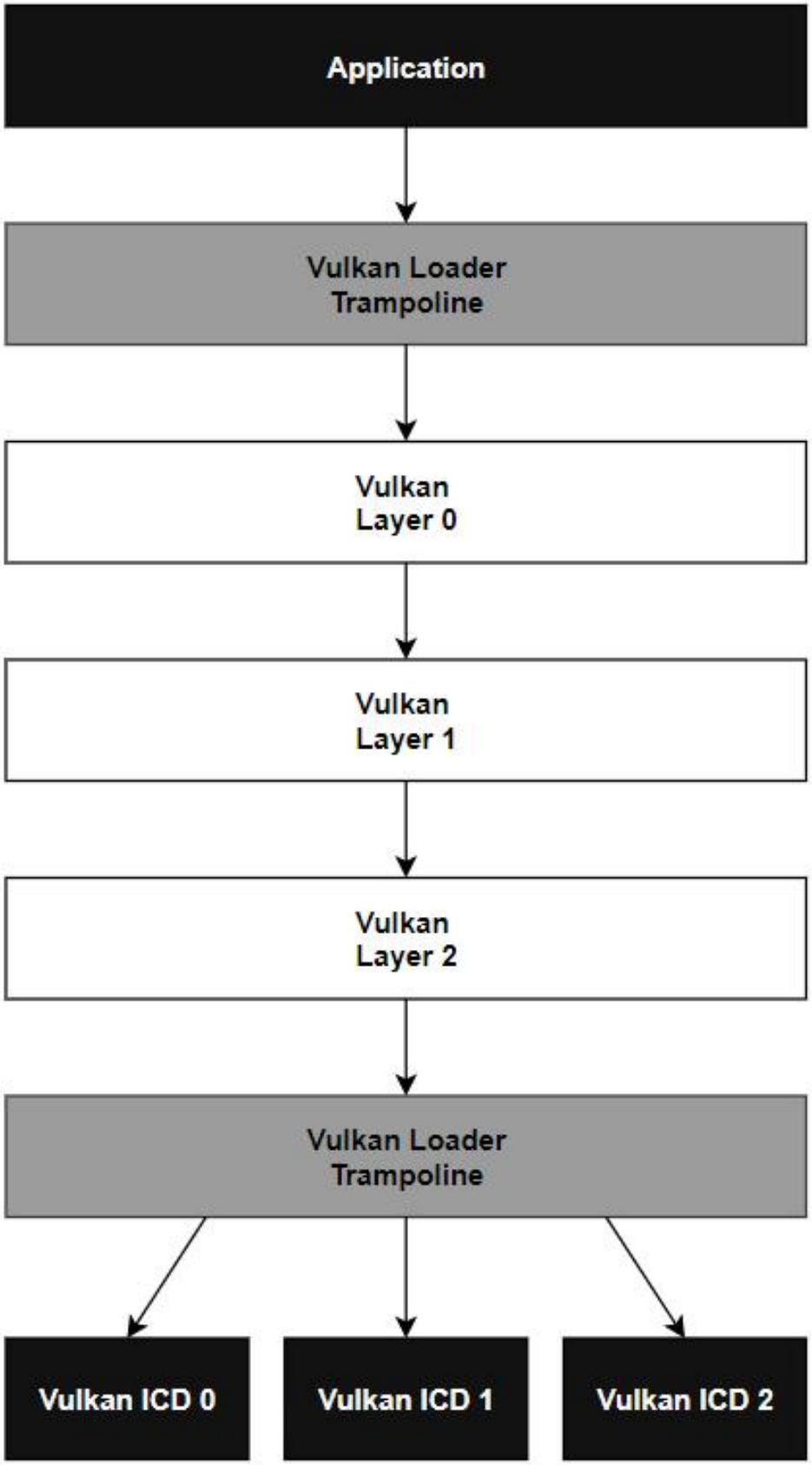

Рисунок 2.5 – Діаграма стандартного ланцюга викликів .

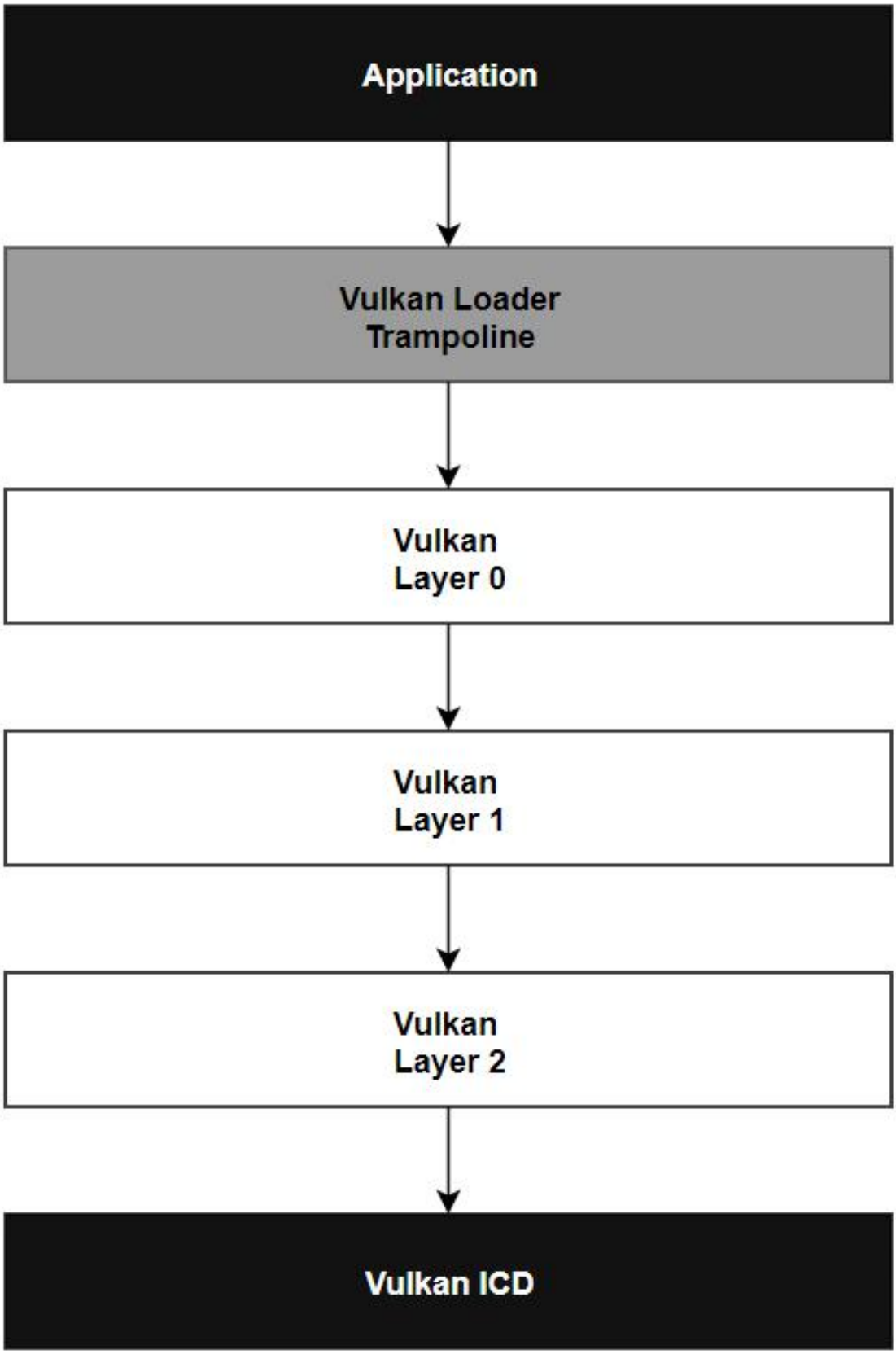

Рисунок 2.6 – Ланцюг викликів через vkGetInstanceProcAddr.

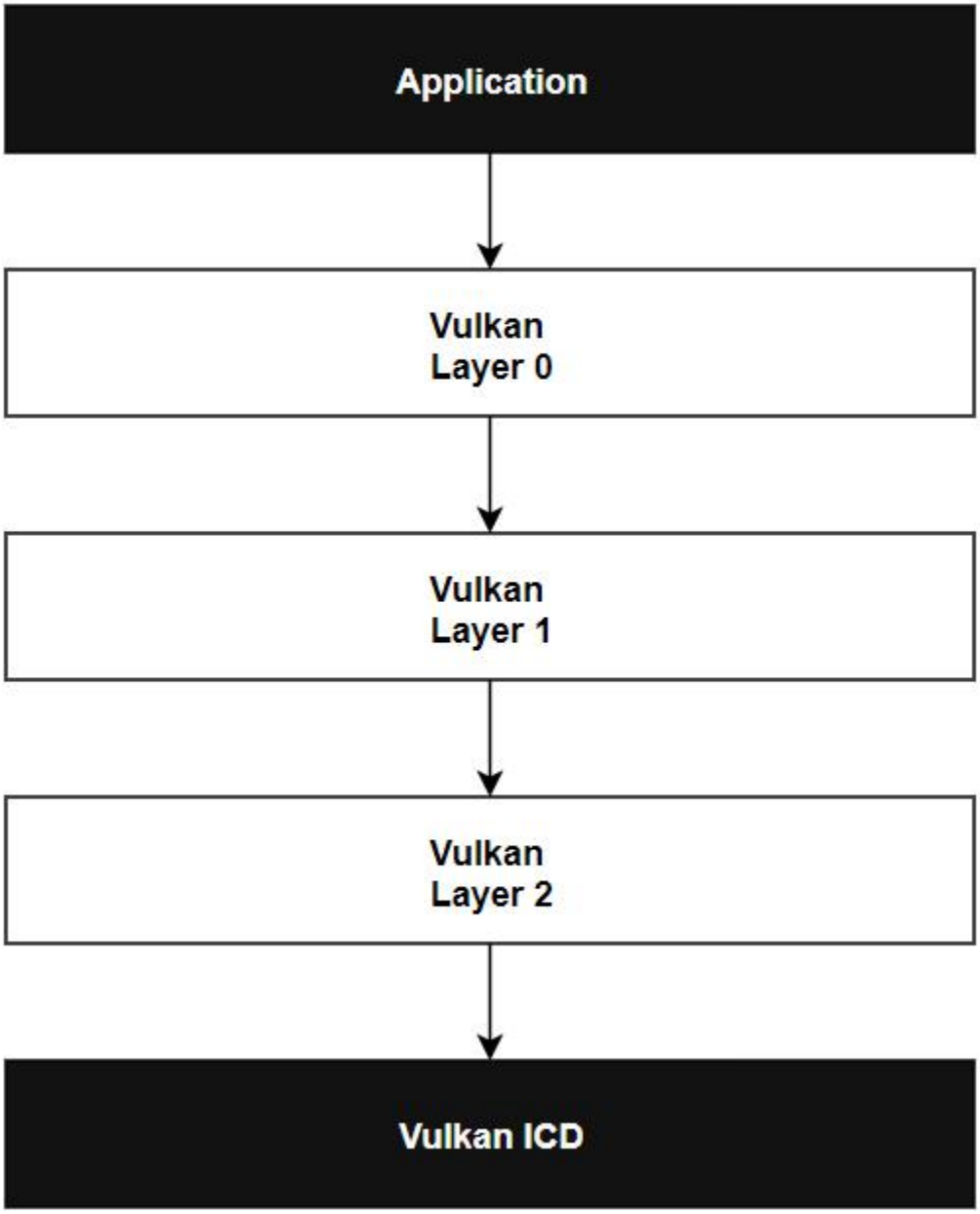

Рисунок 2.7 – Ланцюг викликів через
vkGetInstanceProcAddr та vkGetDeviceProcAddr.

GlobalDispatchTable завантажується в момент створення компонента Runtime, InstanceDispatchTable – в момент створення Instance, DeviceDispatchTable – в момент створення Device. Очікується, що такий підхід зменшить рівень накладних витрат в момент виклику функцій на 2-7% в залежності від драйвера та системи.

2.6 Субмодуль Utils

Кожна мова програмування має стандартну бібліотеку функцій, класів, шаблонів, тощо. C++ не виключення і має свою стандартну бібліотеку шаблонів STL (Standard Templates Library). Це універсальна бібліотека, котра має дуже багатий функціонал і пропонує вже готові базові рішення такі як контейнери, алгоритми; це прискорює час розробки додатку та кодингу. STL дуже гарно зарекомендувала себе в Software Development, але також має великі недоліки. Уявимо, що ваш код роками був у розробці для однієї платформи і після отримуєте запит на перенос існуючої кодобази на нову платформу, котра також має сучасний C++ компілятор та останню версію стандартної бібліотеки шаблонів. Після першої спроби побудувати додаток отримуєте великий список помилок компіляції. Так, якщо побудувати типовий Hello World, то помилок ви не отримаєте, хоча і тут не можна бути впевненим до кінця. Але якщо розглядати реальний продакшн код, то це велика проблема. Чому вона виникає? Одна із головних причин є те, що кожен компілятор для мови програмування C++ мають свою реалізацію стандартної бібліотеки шаблонів. Також на жаль не всі компілятори та системи побудови рішень на C++ мають абсолютно однаковий функціонал, особливо якщо йдеться мова про останні версії мови програмування C++.

Тому існує два шляхи вирішення цієї проблеми. Перший шлях це використання сторонніх подібних бібліотек таких, як Boost, STLPort, EASTL замість стандартного STL. Другий шлях це створення свого рішення, котре буде оптимізоване для всіх потрібних платформ та систем.

Для VulkanKit було обрано другий шлях, бо потреби програмного модуля VulkanKit невеликі та не потребує додаткових залежностей від сторонніх рішень. Результатом такого вибору став субмодуль Utils (в перших ітераціях проектування мав назву Utility).

Субмодуль Utils має невеликий функціонал базових шаблонів до списку якого входять смарт-вказівник Owner, автоматизована робота з

бітовими масками (Bitmask), захист від копіювання об’єктів (Noncopyable), фіксовані рядки (FixedString), тощо.

3 РЕЗУЛЬТАТИ ДОСЛІДЖЕНЬ ТА РОЗРОБКИ

Під час розробки програмного модулю VulkanKit було проведено аналіз існуючих технології та імпелементації на наявність своїх переваг та недоліків. На особливу увагу було взято Vulkan SDK, vulkan-hpp, Vulkano (Rust), PowerVR SDK (PVRVK), так як більшість з них використовуються в реальному продакшині. Кожне з цих рішень має свої якості, які вплинули на кінцевий дизайн VulkanKit.

Також було проведено дослідження властивостей мови програмування C++ версії 17 на підтримку компіляторами, сумісність скомпільованого коду на бінарному рівні (ABI compatibility) між компіляторами та платформами. Порівнювалися результати компіляції та лінковки таких компіляторів як MSVC (Microsoft Visual C++ Compiler), MinGW 32, Cygwin, Clang (для Microsoft Visual Studio та Apple Xcode), GCC (Xcode). На жаль, мова програмування C++ досі немає стандартного бінарного інтерфейсу (Application Binary Interface – ABI), що не дозволяє створити один бінарний об’єкт для використання різними компіляторами на одній платформі. Тому це спонукало до створення іншого дизайну, де програмний інтерфейс ядра модуля виконаний на мові програмування C, бо C має чудові властивості підтримки сумісності на бінарному рівні. Після створення першого прототипу на C стало видно, що такий підхід ускладнює розробку та розуміння структури модуля, тому в кінцевому варіанті відмовились від варіанту дистрибуції модулю в бінарному вигляді в угоду чіткої модульності та продуктивності. Це одна із причин використання Cmake як головної системи побудови проекту та дистрибуції проекту в open-source.

Для демонстрації використання програмного інтерфейсу VulkanKit нижче буде приведено програмний код програми, котра виводить інформацію кожного графічного адаптера в системі (ім’я, версія драйвера, індекс вендора). Цього буде достатньо для того, щоб продемонструвати

можливості та переваги VulkanKit з точки зору дизайну програмного забезпечення перед Vulkan SDK та утілітами як vulkan-hpp.

Але для початку зверніть увагу на реалізації даної програми на нативному Vulkan та Vulkan SDK:

```
void NativeVulkanImplementation()
{
    auto vulkanModule = ::LoadLibraryA(«vulkan-1.dll»);
    assert(vulkanModule);

    auto                    pfnGetInstanceProcAddr                    =
(::PFN_vkGetInstanceProcAddr)::GetProcAddress(vulkanModule,
«vkGetInstanceProcAddr»);
    auto                      pfnCreateInstance                       =
(::PFN_vkCreateInstance)pfnGetInstanceProcAddr(nullptr, «vkCreateInstance»);
    auto                  pfnEnumerateInstanceVersion                 =
(::PFN_vkEnumerateInstanceVersion)pfnGetInstanceProcAddr(nullptr,
«vkEnumerateInstanceVersion»);

    const           char*           extensions[]           =           {
«VK_KHR_get_physical_device_properties2», «VK_KHR_driver_properties» };

    uint32_t instanceApiVersion = VK_MAKE_VERSION(1, 0, 0);
    if (pfnEnumerateInstanceVersion)
    {
        pfnEnumerateInstanceVersion(&instanceApiVersion);
    }

    ::VkApplicationInfo applicationInfo{};
    applicationInfo.apiVersion = instanceApiVersion;

    ::VkInstanceCreateInfo instanceCreateInfo{};
    instanceCreateInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
    instanceCreateInfo.pApplicationInfo = &applicationInfo;
    instanceCreateInfo.ppEnabledExtensionNames = extensions;
    instanceCreateInfo.enabledExtensionCount = ::std::size(extensions);

    ::VkInstance instance{ nullptr };
    pfnCreateInstance(&instanceCreateInfo, nullptr, &instance);
    assert(instance);

    auto                  pfnEnumeratePhysicalDevices                 =
(::PFN_vkEnumeratePhysicalDevices)pfnGetInstanceProcAddr(instance,
«vkEnumeratePhysicalDevices»);
    auto             pfnGetPhysicalDeviceMemoryProperties2            =
(::PFN_vkGetPhysicalDeviceMemoryProperties2)pfnGetInstanceProcAddr(instance,
«vkGetPhysicalDeviceMemoryProperties2»);
    if (!pfnGetPhysicalDeviceMemoryProperties2)
    {
        pfnGetPhysicalDeviceMemoryProperties2                         =
(::PFN_vkGetPhysicalDeviceMemoryProperties2)pfnGetInstanceProcAddr(instance,
«vkGetPhysicalDeviceMemoryProperties2KHR»);
    }
    assert(pfnGetPhysicalDeviceMemoryProperties2);

    uint32_t physicalDeviceCount{};
    pfnEnumeratePhysicalDevices(instance,         &physicalDeviceCount,
```

```
nullptr);
          ::std::vector<::VkPhysicalDevice>                    physicalDevices{
physicalDeviceCount };
          pfnEnumeratePhysicalDevices(instance,          &physicalDeviceCount,
physicalDevices.data());

          for (auto index{ 0u }; index != physicalDeviceCount; ++index)
          {
              ::VkPhysicalDeviceDriverProperties driverProperties{};
              driverProperties.sType                                          =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES;

              ::VkPhysicalDeviceProperties2 properties{};
              properties.sType                                                =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2;
              properties.pNext = &driverProperties;

              pfnGetPhysicalDeviceProperties2(physicalDevices[index],
&properties);

              ::std::clog << «GPU[« << index << «]: «
                  << properties.properties.deviceName << «, «
                  << driverProperties.driverName << « «
                  << driverProperties.driverInfo << «, «
                  << properties.properties.vendorID << «\n»;
          }

          auto                        pfnDestroyInstance                      =
(::PFN_vkDestroyInstance)pfnGetInstanceProcAddr(nullptr,
«vkDestroyInstance»);
          pfnDestroyInstance(instance, nullptr);

          ::FreeLibrary(vulkanModule);
      }

      ////////////////////////////////////////////////////////////////

      void VulkanSDKImplementation()
      {
          auto                  pfnEnumerateInstanceVersion                   =
(::PFN_vkEnumerateInstanceVersion)::vkGetInstanceProcAddr(nullptr,
«vkEnumerateInstanceVersion»);

          const           char*           extensions[]           =           {
«VK_KHR_get_physical_device_properties2», «VK_KHR_driver_properties» };

          uint32_t instanceApiVersion = VK_MAKE_VERSION(1, 0, 0);
          if (pfnEnumerateInstanceVersion)
          {
              pfnEnumerateInstanceVersion(&instanceApiVersion);
          }

          ::VkApplicationInfo applicationInfo{};
          applicationInfo.apiVersion = instanceApiVersion;

          ::VkInstanceCreateInfo instanceCreateInfo{};
          instanceCreateInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
          instanceCreateInfo.pApplicationInfo = &applicationInfo;
          instanceCreateInfo.ppEnabledExtensionNames = extensions;
          instanceCreateInfo.enabledExtensionCount = ::std::size(extensions);
```

```
          ::VkInstance instance{ nullptr };
          ::vkCreateInstance(&instanceCreateInfo, nullptr, &instance);
          assert(instance);

          uint32_t physicalDeviceCount{};
          ::vkEnumeratePhysicalDevices(instance,         &physicalDeviceCount,
nullptr);
          ::std::vector<::VkPhysicalDevice>                   physicalDevices{
physicalDeviceCount };
          ::vkEnumeratePhysicalDevices(instance,         &physicalDeviceCount,
physicalDevices.data());

          for (auto index{ 0u }; index != physicalDeviceCount; ++index)
          {
              ::VkPhysicalDeviceDriverProperties driverProperties{};
              driverProperties.sType                                          =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DRIVER_PROPERTIES;

              ::VkPhysicalDeviceProperties2 properties{};
              properties.sType                                                =
VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_PROPERTIES_2;
              properties.pNext = &driverProperties;

              ::vkGetPhysicalDeviceProperties2(physicalDevices[index],
&properties);

              ::std::clog << «GPU[« << index << «]: «
                  << properties.properties.deviceName << «, «
                  << driverProperties.driverName << « «
                  << driverProperties.driverInfo << «, «
                  << properties.properties.vendorID << «\n»;
          }

          ::vkDestroyInstance(instance, nullptr);

      }
```

Приклад 3.1 – Реалізації програм для запиту графічних адаптерів та їх налаштування на Vulkan.

Як ми бачимо, версія VulkanSDKImplementation є трохи коротшою та прозорою, а ніж нативна реалізація. Але все ж таки VulkanSDK вимагає роботи деяку підготовку опціонального функціоналу. Давайте тепер глянемо реалізацію даної програми за допомогою VulkanKit:

```
using namespace VulkanKit::API;

void VulkanKitImplementation()
{
  auto runtime = Core::CreateRuntime({});
  VulkanKitAssert(runtime);

  const ::VulkanKit::Char* extensions[] =
```

```
  {
    »VK_KHR_get_physical_device_properties2»,
    »VK_KHR_driver_properties»
  };

  Core::InstanceCreateInfo instanceCreateInfo{};
  instanceCreateInfo.enabledExtensionNames = extensions;

  auto instance = runtime->CreateInstance(instanceCreateInfo);
  VulkanKitAssert(instance);

  for (const auto physicalDevice : instance->GetPhysicalDevices())
  {
    Core::PhysicalDeviceProperties properties{};
    physicalDevice->GetProperties(properties);

    const auto extendedPhysicalDevice = physicalDevice->As<
Extensions::DriverProperties::PhysicalDevice>();

    Extensions::DriverProperties::DriverProperties driverProperties{};
    extendedPhysicalDevice->GetDriverProperties(driverProperties);

    ::std::clog << «GPU: « << properties.name << «, «
        << driverProperties.driverName << « «
        << driverProperties.driverInfo<< «, «
        << properties.vendor << «\n»;
  }
}
```

Приклад 3.2 – Реалізації програми для запиту графічних адаптерів та їх налаштування на VulkanKit.

В приведених програмних кодах спеціально жирним шрифтом виділені виклики Vulkan та VulkanKit. Як можна побачити програмний код в виконанні VulkanKit є на багато компактним та прозорим для розуміння навіть в порівнянні з імплементацією на Vulkan SDK. Також VulkanKit не вимагає додаткових витрат в час виконання та залежностей від інших технологій.

Отже VulkanKit - програмний модуль на базі мови програмування C++ для Vulkan, основною метою якого є спрощення розробки на Vulkan шляхом надання об’єктно-орієнтованого дизайну, безпеки типів на етапі компіляції, автоматизованих методів управління часом життя об’єктів, спрощеного та явного програмного інтерфейсу (API) для запитів функціоналу, обмежень та розширень, дизайну «Parent-Child» для компонентів, прямий виклик функцій Vulkan, що має привести до більшої продуктивності в великомасштабних

проектах. Увесь цей функціонал дозволяє створювати код коротшим і простішим для розуміння, ніж нативний код Vulkan.

Також було проведено дослідження існуючих архітектурних рішень та створення дизайну для вирішення даної задачі. Після декількох ітерацій проектування, реалізації прототипів та тестування, було прийнято рішення реалізувати мультиінтерфейсний дизайн на основі композиції. Ключова ідея полягає в тому, що кожен інтерфейс реалізується в окремому класі, котрі у свою чергу компонуються в головному класі суб'єкта. Схожий підхід використовується в Component Object Model (COM) від Microsoft та CORBA. Кожен об'єкт-реалізація має вказівник на головний об'єкт, що надає можливість об'єктам-реалізаціям створювати поведінку як одного цілого суб'єкта.

В результаті в VulkanKit було впроваджено новий підхід та дизайн взаємодії з суб'єктами – інтерфейси. Інтерфейс (чи протокол) з точки зору об'єктно-орієнтованого програмування та дизайну – це опис дій, які об'єкт може виконувати. Це один із способів описання абстракцій в мовах програмування, котрі мають об'єктно-орієнтовану філософію програмування. В мовах програмування такі як Java, Swift, C# інтерфейси відкривають великі можливості в проектуванні та імплементації програмного забезпечення. Клас має можливість реалізувати декілька інтерфейсів (мультиінтерфейс), що надеє великий рівень гнучкості та модульності в одному компоненті.

Для підвищення продуктивності було реалізовані спеціальний функціонал у вигляді віртуальних таблиць функцій драйвера. Це зроблено тому що структура драйвера спроектована таким чином, що ланцюжок викликів функцій Instance реалізується іншим шляхом в порівнянні з ланцюжком викликів функцій Device. Так як Vulkan Instance – це високорівнева конструкція, яка використовується для надання інформації на системному рівні Vulkan. Через це функції Instance необхідно транслювати на кожен доступний ICD в системі. Найкращим рішенням було це здійснювати виклик vkGetDeviceProcAddr для всіх функцій Vulkan Device. Це додатково

оптимізує ланцюжок викликів, видаляючи рівні завантажувачв в більшості сценаріїв Крім того, якщо ніякі шари не включені, вказівник функції вашої програми буде вказувати безпосередньо на ICD. Якщо їх викликати у правильній черзі, то менша кількість викликів може привести до підвищення продуктивності. Тому для досягнення максимальної продуктивності, розробник має завантажувати потрібний функціонал на кожному рівні за допомогою нативних функцій vkGetInstanceProcAddr, vkGetDeviceProcAddr. Тому для VulkanKit було впроваджено на рівні імплементації три типи таблиць функцій: GlobalDispatchTable, InstanceDispatchTable, DeviceDispatchTable. GlobalDispatchTable завантажується в момент створення компонента Runtime, InstanceDispatchTable – в момент створення Instance, DeviceDispatchTable – в момент створення Device. В порівнянні з використанням Vulkan SDK такий підхід зменшує рівень накладних витрат в момент виклику функцій в середьому до 5% в залежності від драйвера та системи. Це особливо гарний показчик для мобільних та консольних систем.

ВИСНОВКИ

Оглядаючись на розглянуті приклади використання Vulkan API та VulkanKit, опишемо переваги новоствореної бібліотеки. Завдяки наявності прозорої модульної структури і використанню об’єктно-орієнтованого підходу, потенційно полегшується створення додатків з використанням Vulkan.

У якості автономної C++ бібліотеки VulkanKit надає експериментальний API, що явно застосовує переваги програмної ідіоми C++ Resource Acquisition Is Initialization (RAII), безпеку типів та шаблони мови C++ як засіб забезпечення узагальненого коду.

Очікується, що VulkanKit надає можливість значно скоротити час, витрачений для розробки та налагодження роботи програмного додатку, і разом з тим, загальний розмір кодової бази. Це дозволяє розробникам приділяти більше уваги критичним частинам програми, які зазвичай вимагають більшої ретельності та акуратності від творця.

У якості додаткової переваги, на деяких окремих драйверах можна зменшити накладні витрати CPU і покращити продуктивність, оскільки VulkanKit самостійно завантажує функціонал Vulkan.

Бібліотека VulkanKit поширюється як програмне забезпечення з відкритим кодом під ліцензією Apache 2.0.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Vulkan. Khronos group [Електронний ресурс] – Режим доступу до ресурсу: https://www.khronos.org/vulkan/.

2. Vulkan : Graphics and compute Belong Together [Електронний ресурс]. – 2015. – Режим доступу до ресурсу: https://www.khronos.org/assets/uploads/developers/library/overview/2015_vulkan_v1_Overview.pdf.

3. Vulkan Applications Enabled on Apple Platforms [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://www.khronos.org/news/press/vulkan-applications-enabled-on-apple-platforms.

4. One of Mantle’s Futures: Vulkan [Електронний ресурс]. – 2015. – Режим доступу до ресурсу: https://community.amd.com/community/gaming/blog/2015/05/12/one-of-mantles-futures-vulkan.

5. Vulkan API официально поддерживает multi-GPU на Windows 10, Windows 7, Windows 8.1 и Linux [Електронний ресурс]. – 2017. – Режим доступу до ресурсу: https://www.playground.ru/misc/vulkan_api_ofitsialno_podderzhivaet_multi_gpu_na_windows_10_windows_7_windows_8_1_i_linux-242819?.

6. AMD’s Mantle Lives On In Vulkan – Lays The Foundation For The Next OpenGL [Електронний ресурс]. – 2014. – Режим доступу до ресурсу: https://wccftech.com/mantle-vulkan-amd/.

7. Селлерс Г. Vulkan. Руководство разработчика = Vulkan. Programming Guide. — ДМК Пресс, 2017. — 394 с.

8. Quake 1 Ported To Run On Vulkan [Електронний ресурс]. – 2016. – Режим доступу до ресурсу: https://www.phoronix.com/scan.php?page=news_item&px=Quake-1-On-Vulkan.

9. Vulkan API (glNext) от Khronos Group [Електронний ресурс]. – 2016.

– Режим доступу до ресурсу: https://habr.com/ru/post/283490/.

10. Is AMD Mantle Dead As We Have Known It? Vulcan API Uses Mantle Technology for OpenGL [Електронний ресурс]. – 2015. – Режим доступу до ресурсу: https://www.legitreviews.com/amd-mantle-dead-known-vulcan-api-uses-mantle-technology-opengl_159339.

11. Khronos Group Releases Vulkan Ray Tracing [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://www.khronos.org/news/press/khronos-group-releases-vulkan-ray-tracing.

12. https://www.mcvuk.com/development-news/glnext-revealed-as-vulkan-graphics-api/203867/ [Електронний ресурс]. – 2015. – Режим доступу до ресурсу: glNext revealed as Vulkan graphics API.

13. Quick Look: Vulkan Performance on The Talos Principle [Електронний ресурс]. – 2016. – Режим доступу до ресурсу: https://www.anandtech.com/show/10047/quick-look-vulkan-performance-on-the-talos-principle.

14. AMD открыла исходный код V-EZ, оболочки кроссплатформенного низкоуровневого Vulkan API [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://habr.com/ru/post/421361/.

15. Vulkan 1.2 Deepens HLSL Support [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://www.khronos.org/blog/hlsl-first-class-vulkan-shading-language.

16. Offline Compilation of OpenCL Kernels into SPIR-V Using Open Source Tooling [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://www.khronos.org/blog/offline-compilation-of-opencl-kernels-into-spir-v-using-open-source-tooling.