Vulkan a OpenGL
porównanie wydajności na przykładach

(Vulkan and OpenGL — performance comparison based on examples)

Damian Dyńdo

Praca magisterska

Promotor: dr Andrzej Łukaszewski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

10 sierpnia 2017

Streszczenie

Vulkan to nowe API graficzne i obliczeniowe stworzone z myślą o najbardziej wymagających aplikacjach, którego głównym konceptem jest udostępnienie bardzo niskopoziomowego dostępu do GPU i przesunięcie odpowiedzialność za kluczowe aspekty obsługi sprzętu ze sterowników na twórców aplikacji. Vulkan z założenia przynieść ma spory wzrost wydajności dzięki niższemu narzutowi na CPU oraz udostępnieniu wielu możliwości dla optymalizacji. W pracy tej zbadam czy rzeczywiste wyniki potwierdzą zapewnienia twórców i czy korzystając z tego API możemy uzyskać lepszą wydajność w porównaniu z interfejsem OpenGL. Korzystać będę przy tym z dostępnych testów wydajnościowych porównujących oba API oraz przeprowadzę własne testy na autorskiej aplikacji napisanej na potrzeby tej pracy. Ponadto praca zawiera krótkie omówienie kluczowych elementów API Vulkan.

Vulkan is a new graphics and compute API created for performance-demanding applications. Its core concept is to expose very low-level access to GPU and shift responsibility of key aspects of managing hardware from drivers to application' creators. Vulkan should, by design, come with a performance gain due to its low overhead on CPU and exposition of more possibilities for optimizations. In this paper I will try to test if that is really the case and if we — by using Vulkan API — can gain performance comparing to OpenGL interface. To do this, I will analyze existing performance benchmarks comparing both APIs and perform my own tests using testing application created for this paper. Furthermore, this paper will include short overview of key Vulkan API elements.

Spis treści

Rozdział 1.

Wprowadzenie

1.1. Wstęp

Przez ostatnie kilkanaście lat rynek procesorów graficznych dynamicznie się rozwijał. Nowe modele kart graficznych obsługują coraz to nowszą i bardziej skomplikowaną funkcjonalność, która w założeniu oferować ma szersze możliwości oraz przyśpieszenie renderowania skomplikowanych wizualizacji trójwymiarowych. Programiści chcąc tworzyć przenośne aplikacje multimedialne zmuszeni są do korzystania z odpowiednich interfejsów programistycznych pozwalających na dostęp do tych urządzeń.

Interfejs programistyczny aplikacji (ang. *Application Programming Interface*, **API**) jest to zestaw reguł pozwalających na komunikowanie się różnych modułów we wspólny i zrozumiały dla siebie "język". Interfejs taki specyfikuje się na poziomie kodu źródłowego napisanego w odpowiednim języku programowania.

Istnieje kilka popularnych i szeroko wykorzystywanych interfejsów programistycznych pozwalających na korzystanie z procesorów graficznych do skomplikowanych celów, takich jak renderowanie zaawansowanej grafiki dwuwymiarowej i trójwymiarowej, czy wykonywanie skomplikowanych obliczeń wykorzystując przy tym równoległą architekturę procesorów graficznych. Najpopularniejszymi z nich są:

- Direct3D
- OpenGL
- Vulkan
- Metal

API te ciągle ewoluują by nadążać za zmianami wprowadzanymi w sprzęcie oraz by móc zaoferować największą wydajność i elastyczność. Każdy z tych interfejsów ma

swoje wady i zalety, zatem warto je znać, by wiedzieć, który z nich będzie najlepszy do danego zastosowania.

Na przestrzeni ostatnich kilkunastu lat zauważyć możemy trend w architekturze tych interfejsów pokazujący przenoszenie coraz to większej odpowiedzialności ze sterowników urządzeń na programistów. Dzieje się tak, gdyż to twórcy danej aplikacji wiedzą najlepiej jak dane zasoby zostaną użyte i nie muszą oni zgadywać jak dana aplikacja będzie się zachowywała.

1.2. Cel pracy

Celem pracy jest porównanie nowego API graficznego i obliczeniowego Vulkan z istniejącym wiele lat i szeroko wykorzystywanym interfejsem OpenGL. Kluczowymi elementami badanymi w tej pracy będą:

- wpływ API na wydajność aplikacji,
- subiektywna ocena opłacalności wyboru tego API.

Aby osiągnąć te cele, przejrzymy dostępne testy aplikacji wspierających oba interfejsy graficzne i przeanalizujemy je pod względem wydajności. Na potrzeby pracy stworzone zostały również autorskie testy skupiające się na porównaniu różnic wydajności obu API. Testy te składają się z kilku identycznych scen renderowanych w obu API i pozwalają zbadać między innymi ich narzut. Dodatkowo, postaram się opisać kluczowe elementy interfejsu Vulkan, ze szczególnym uwzględnieniem aspektów wydajnościowych.

1.2.1. Analiza wydajności

Najważniejszą zapowiadaną zaletą nowszego API jest możliwość uzyskania znacznie lepszej wydajności względem dostępnych dotychczas interfejsów. Twórcy Vulkana zapewniają, że samo korzystanie z niego będzie wiązać się z zyskiem wydajnościowym wynikającym z niższego narzutu na CPU przez *odchudzenie* sterowników. Kolejnym ważnym elementem jego architektury jest fakt, że został on zaprojektowany z myślą o efektywnym wykorzystywaniu w aplikacjach wielowątkowych. Vulkan udostępnia również bardzo niskopoziomowe mechanizmy do zarządzania zasobami urządzeń go obsługujących, co przełożyć ma się na lepsze ich wykorzystanie. To wszystko powinno pozwolić na otrzymanie zauważalnie wyższej wydajności, szczególnie w aplikacjach wykonujących dużą ilość obliczeń na głównym procesorze (CPU).

Vulkan niestety nie wprowadza dużych zmian w kwestii pracy wykonywanej przez procesor graficzny (GPU), co prawdopodobnie przełoży się na małe zyski lub ich brak w aplikacjach silnie ograniczonych przez moc GPU.

1.2.2. Analiza opłacalności

Istotnym elementem, od którego zależy tempo rozwoju i przyswojenia nowego API jest poziom trudności związany z jego nauką, zrozumieniem oraz efektywnym wykorzystywaniem w praktyce. By API odniosło sukces, powinno być relatywnie proste, zrozumiałe, a korzystanie z niego nie powinno wymagać znacznie większych nakładów pracy niż w przypadku korzystania z innych API. Pracując nad aplikacjami testującymi wydajność spróbuję subiektywnie ocenić pod tym względem jak nowy interfejs wypada na tle swojego poprzednika.

1.3. Omówienie rozdziałów

Rozdział 2. OpenGL

W tym rozdziale przypomnę pokrótce historię tego API, omówię na przykładach jak ewoluowało ono przez te wszystkie lata by stać się tym, czym jest dziś, oraz — na koniec — opowiem o tym jak wygląda ono i jak stosowane jest dziś.

Rozdział 3. Vulkan

Rozdział ten skupiony będzie na różnych aspektach tytułowego API. Omówię w nim jego architekturę, kluczowe elementy, które zostały wprowadzone — zwracając przy tym szczególną uwagę na te elementy, których nie znajdziemy w jego poprzedniku. Ponadto postaram się również porównać je ze starszym interfejsem i wskazać elementy wprowadzone z nastawieniem na wydajność.

Rozdział 4. Przegląd istniejących testów

Znajdą się tutaj omówienia dostępnych w sieci testów porównujących wydajność oraz wykorzystanie zasobów aplikacji wykorzystujących oba interfejsy. Analizowane testy przedstawione zostaną tu w trzech kategoriach:

1. Testy oparte na autorskim oprogramowaniu stworzonym w tym celu.
2. Testy oparte na silnikach graficznych.
3. Testy oparte na wydanych dotychczas grach komputerowych.

Rozdział 5. Przeprowadzone testy

W rozdziale tym znajdą się informacje na temat stworzonego przeze mnie na cele tej pracy projektu, którym posłużę się do przetestowania wydajności uzyskanej

przy zastosowaniu każdego z omawianych interfejsów. Omówiony będzie pokrótce sam projekt, a następnie szerzej opisany każdy stworzony test wraz zagadnieniami specyficznymi dla implementacji w każdym z testowanych API. Następnie podane zostaną informacje o sposobie przeprowadzenia testów oraz ich wyniki wraz z analizą.

Rozdział 6. Wnioski

Rozdział ten zawierać będzie podsumowanie informacji zawartych w tej pracy oraz wyników uzyskanych w trakcie jej tworzenia. Postaram się w nim zwięźle przedstawić wnioski wypływające z zebranych danych oraz odpowiedzieć na dwa kluczowe pytania zadane wcześniej w tym rozdziale, czyli:

1. Czy korzystanie z nowszego interfejsu pozwala zapewnić większą wydajność?
2. Jak wyglądają różnice pomiędzy nakładem pracy potrzebnym na stworzenie aplikacji w tych API?

Dodatek A. Projekt GL_vs_VK

Rozdział ten opisuje stworzony na potrzeby tej pracy projekt pozwalający na testowanie obu API graficznych w kilku przygotowanych scenariuszach testowych. Zawiera on krótki opis kodu źródłowego, użytych bibliotek oraz informacje o budowaniu i korzystaniu z projektu.

Rozdział 2.

OpenGL

2.1. Historia

Interfejs OpenGL został stworzony w roku 1992 przez firmę **Silicon Graphics**, aktualnie znaną pod nazwą **SGI**, jako odpowiedź na ówczesne zapotrzebowanie przenośnego i wspólnego API dla szerokiego grona dostępnych kart graficznych. Powstał on poprzez przekształcenie swojego protoplasty — interfejsu *IrisGL* — w otwarty standard oraz odchudzenie go ze zbędnych — z punktu widzenia renderowania grafiki — funkcjonalności. Jego główną zaletą była wówczas możliwość używania go na szerokiej klasie procesorów graficznych, co z czasem i pojawieniem się innych interfejsów przestało odgrywać tak wielką rolę. Interfejs ten ciągle ma wiele zalet w stosunku do jego aktualnej konkurencji, z których najważniejszą wydaje się być wieloplatformowość i przenośność na różne systemy operacyjne, w przeciwieństwie do innych swoich rywali, takich jak — wydanego przez firmę Microsoft — Direct3D, który działa tylko na systemach z rodziny Windows.

Aktualnie API rozwijane jest przez **Khronos Group**, która powstała w roku 2006 i zrzesza wiele firmy z branży takich jak **Advanced Micro Devices**, **Intel Corporation** czy **NVIDIA**. Organizacja ta skupia się na tworzeniu i rozwijaniu otwartych i darmowych standardów i interfejsów programistycznych.

Pierwsze wersje API były relatywnie proste i opierały się na stałym potoku graficznym oferując podstawowe możliwości i małą elastyczność. Z czasem, by dać programistom więcej swobody, udostępniono programowalny potok graficzny, który był swego rodzaju rewolucją w tym jak tworzono aplikacje graficzne. Zapewniał on również znacznie większą funkcjonalność co pozwoliło na implementację dużo bardziej skomplikowanych i efektownych algorytmów.

Aby przyśpieszyć rozwój interfejsu, wprowadzony został **mechanizm rozszerzeń**. Pozwalał on twórcom procesorów graficznych oraz firmom zaangażowanym w rozwój interfejsu na wprowadzanie własnych, niestandardowych rozszerzeń. Najlepsze i najprzydatniejsze rozszerzenia z czasem, już w stabilnej postaci, zostawały

wcielane do podstawowej wersji API. Dzięki temu mechanizmowi programiści mieli znacznie szybszy dostęp do funkcjonalności pojawiającej się w najnowszych generacjach sprzętu. Minusem tego rozwiązania była fragmentacja dostępnych elementów API pomiędzy konkretnymi procesorami graficznymi oraz sprzętem różnych firm, przez co pisanie wydajnych aplikacji sprowadzało się do implementowania podsystemów graficznych na kilka sposobów i wybraniu aktualnego w zależności od tego jakie elementy interfejsu są dostępne na danym sprzęcie.

OpenGL często przedstawiane jest jako wielka maszyna stanów, na którą można wpływać poprzez wywoływanie odpowiednich funkcji interfejs. Tak skonstruowane — ze względu na domyślną konfigurację wielu elementów — jest bardzo proste w użytkowaniu. Architektura taka jednak często była krytykowana i prowadziła do wielu problemów, których część nie została — i prawdopodobnie nie zostanie już nigdy — rozwiązana. Przykładem takiego problemu może być brak możliwości używania API na wielu wątkach i niezależnego jego wykorzystania przez kilka bibliotek jednocześnie.

2.2. Stały potok

Stały potok (ang. *fixed-function pipeline*) to proces renderowania grafiki dostępny w API OpenGL od pierwszych jego wersji. Polegał on na udostępnieniu programistom ustalonego procesu renderowania, na który mogli oddziaływać tylko w minimalny sposób poprzez ustawienie odpowiednich zmiennych. Po wywołaniu metody rysującej, procesor graficzny, posiadając wszystkie dane, przeprowadzał ustalone obliczenia zgodne z ustawieniami stałego potoku. Stały potok był chwalony za swoją prostotę i z tego względu często wykorzystywany w najprostszych aplikacjach graficznych. Wraz z wersją 3.0, twórcy API wprowadzili mechanizm deprecjonowania przestarzałej funkcjonalności, w tym również stałego potoku. Wersja 3.1 usunęła z głównego kontekstu wszystkie oznaczone tak elementy interfejsu, wymuszając przy tym na programistach przepisanie oprogramowania z wykorzystaniem bardziej elastycznego mechanizmu *programowalnego potoku*. Obsługa stałego potoku została jednak zachowana we wstecznie-kompatybilnym kontekście, przez co w większości implementacji jest dostępna w nim do dziś.

2.3. Programowalny potok

Z czasem metoda *stałego potoku* okazała się niewystarczająca do renderowania coraz to bardziej realistycznej grafiki i programistom udostępniono — początkowo w formie rozszerzeń, które oficjalnie zostały częścią API w wersji 2.0 — mechanizm programowalnych shaderów (ang. *programmable pipeline*). Są one małymi programami pisanymi w języku **GLSL**, który przypomina język **C**. Dzięki nim programista może wpływać na części potoku, przez które przechodziły wygenerowane wierzchołki oraz

fragmenty, co pozwoliło na stworzenie i implementację całkiem nowych i zdecydowanie bardziej skomplikowanych i efektownych algorytmów, których wykonywanie wcześniej nie było praktyczne. Idea programowalnych potoków zyskała na popularności i dziś są one dostępne w jakiejś postaci w praktycznie wszystkich najpopularniejszych interfejsach. Z biegiem czasu programiści uzyskali kontrolę nad większą ilością etapów potoku. Na dzień dzisiejszy programista może użyć następujących rodzajów shaderów:

- shader wierzchołków,
- shader fragmentów (lub pikseli),
- shader geometrii,
- shader teselacji.

Listing 2..1: Przykładowy kod shadera wierzchołków w języku GLSL.

```
#version 330 core

layout(location = 0) in vec4 input_position;
uniform mat4 MVP;

void main()
{
    gl_Position = MVP * input_position;
}
```

2.4. OpenGL dziś

API OpenGL znacznie ewoluowało od jego pierwszej wersji. W aktualnej wersji — na czas pisania jest to wersja 4.5 — programista ma do dyspozycji znacznie więcej niskopoziomowych mechanizmów, tak, by móc jak najlepiej wykorzystywać aktualne procesory graficzne i uzyskać na nich jak najlepszą wydajność. Funkcjonalność ta pojawiała się w formie rozszerzeń, z których spora ilość znalazła się już w standardzie wersji 4.5, i ciągle jest powiększana.

Dostępne są również rozszerzenia do API, które starają się udostępnić funkcjonalność podobną do tej znanej z Vulkana [1]) i DirectX 12, czy nawet wyłączyć pewne elementy walidacji, by uzyskać efekt podobny do warstwowej architektury nowszego API [2] .

Warto również wspomnieć, że wśród programistów wykształciły się pewne techniki programowania — znane między innymi jako **AZDO** (ang. *Approaching Zero*

[1] Przykładem może być rozszerzenie `NV_command_list`.
[2] Umożliwić to ma dopiero rozwijane rozszerzenie `KHR_no_error`.

Drier Overhead) — które mają na celu w znacznym stopniu zminimalizowanie lub nawet całkowite wyeliminowanie wad architektury tego interfejsu.

Rozdział 3.

Vulkan

3.1. Wprowadzenie

Vulkan to nowy interfejs graficzny i obliczeniowy stworzony przez **Khronos Group** jako następca interfejsu OpenGL. Udostępniony został publicznie w lutym 2016 roku na wielu platformach wspierając przy tym takie systemy operacyjne jak **Windows 7** i nowsze, **Linux**, **Android** oraz **Tizen**.

Głównymi motywacjami do stworzenia nowego API był brak istniejącego interfejsu dobrze modelującego architekturę aktualnie dostępnych procesorów graficznych, oraz stworzenie nowoczesnego i czystego API, które w łatwy sposób pozwalałoby na niskopoziomową komunikację z szeroką gamą urządzeń na różnych platformach. Bazuje on w dużej mierze na przekazanym przez AMD, własnościowym API nazwanym **Mantle**, jednak różni się on pewnymi elementami od swojego protoplasty.

Vulkan celuje w najbardziej zasobożerne i wymagające aplikacje multimedialne oraz gry. Przerzuca on sporą część odpowiedzialności dotychczas ciążącej na sterownikach na programistów, przez co uważany jest za dużo bardziej niskopoziomowe API niż OpenGL. Ma to na celu udostępnienie programiście możliwości do wykorzystanie dostępnych zasobów w najlepszy możliwy sposób zgodnie z zamierzonym sposobem ich użycia. Ważnym elementem jest również idea bardzo niskiego narzutu interfejsu, która wynika z odchudzenia sterowników. Sterowniki obsługujące to API bowiem mogą być znacznie lżejsze, gdyż nie muszą sprawdzać poprawności wszystkich zapytań oraz — wynikającą ze względu na pseudo-obiektową architekturę — aktualnego globalnego stanu, a jedynie odnoszą się do konkretnych obiektów. Ponadto dotychczas to sterowniki odpowiadały za takie operacje jak alokacja pamięci czy automatyczne czyszczenie zasobów, co — nie znając sposobu w jaki działa aplikacja — było ciężkim i wymagającym zasobów zadaniem. Vulkan wymaga od programisty sporej wiedzy na temat tego, co zostanie wykonane jeszcze przed wykonaniem czegokolwiek, dzięki czemu sterowniki mogą lepiej zaplanować pracę, którą będą wykonywać

oraz wykonać ją w pełni optymalnie.

Vulkan zaprojektowany jest w sposób warstwowy, dzięki czemu programista może włączyć dodatkowe warstwy walidujące sposób wykorzystania interfejsu. Pozwala to zaoszczędzić nakład pracy normalnie przeznaczony na sprawdzanie poprawności aplikacji w gotowym i sprawdzonym produkcie oraz włączenie ich tylko w trakcie rozwijania aplikacji, co przekłada się na znacznie niższy narzut. Dotychczasowo wyłączenie walidacji w istniejących API nie było możliwe.

3.2. Ogólny zarys API

Interfejs ten został zaprojektowany w nowoczesny i przemyślany sposób. Jego architektura przypomina dobrze znany programistom interfejs obiektowy, czego przykładem może być konieczność podania uchwytu do obiektu na którym będziemy wykonywać daną czynność jako pierwszy argument funkcji. Dzięki dobremu nazewnictwu elementów interfejsu, poruszanie się po nim jest stosunkowo proste. API to nie korzysta z ogromnej maszyny stanów oraz listy globalnie aktywowanych obiektów, a prawie każda operacja wykonywana jest z perspektywy wskazanego obiektu i wymaga podania uchwytów do wszystkich obiektów, do których się właśnie odnosimy.

Vulkan wprowadza wiele nowych obiektów modelujących w niskopoziomowy sposób elementy dostępnego sprzętu czy potoku renderującego. Część z tych elementów wprowadzona jest, by dać programiście większą elastyczność w tym co zamierza zrobić, a część wprowadzona została głównie by umożliwić lepsze wykorzystanie sprzętu w typowych zastosowaniach i pozwolić na lepszą optymalizację aplikacji. Interfejs jest przy tym spójny i zachowuje swoje konwencje w nazewnictwie, przez co poruszanie się po funkcjach, stałych czy wartościach wyliczeń jest proste i intuicyjne. Dodatkowo API to korzysta z dobrodziejstw nowszych standardów języków programowania oraz zalecanych wzorców projektowych, czego przykładem może być fakt rozdzielenia wszystkich wartości wyliczeniowych w osobne typy. O ile w języku **C** nie daje to kompletnego bezpieczeństwa typów, o tyle jest to zdecydowany krok naprzód względem swojego poprzednika — gdzie wszystkie wartości enumeracji należały do jednego, wspólnego typu — oraz pozwala to na stworzenie bibliotek opakowujących ten interfejs i uzyskujących pełne bezpieczeństwo typów.[1]

Podobnie jak OpenGL, Vulkan również wspiera mechanizm rozszerzeń, które mogą być zdefiniowane na różnych poziomach:

- poziom instancji — funkcjonalność dostępna dla całej aplikacji,
- poziom urządzenia — funkcjonalność dostępna na wybranym urządzeniu.

[1] Przykładem takiej biblioteki może być oficjalny *wrapper* API w języku **C++** o nazwie **VulkanHpp**. Wykorzystuje on obiektową architekturę interfejsu by ułatwić korzystanie z niego implementując znane wzorce projektowe.

3.2.1. Instancja

Każda aplikacja korzystająca z tego interfejsu musi stworzyć jej własną instancję Vulkana, która będzie skupiała w sobie cały swój stan oraz wiązała ze sobą wszystkie stworzone na potrzeby aplikacji obiekty. W stosunku do architektury OpenGL, wydaje się to być znacznie lepszym rozwiązaniem niż ogromna i ciężka w zrozumieniu maszyna stanów, w której łatwo się zgubić.

Tworząc instancję, programista musi sprecyzować z których rozszerzeń będzie korzystał oraz które warstwy powinny zostać włączone. Ponadto programista przekazuje informacje na temat uruchamianej aplikacji oraz silnika, z którego korzysta. Pozwala to w łatwy sposób na zidentyfikowanie przez sterownik graficzny uruchamianej aplikacji i wykorzystanie ewentualnych optymalizacji zaimplementowanych dla konkretnego silnika graficznego.

Mając własną instancję Vulkana, programista może sprawdzić jakie urządzenia fizyczne są dostępne i zacząć na nich pracę.

3.2.2. Urządzenia

Vulkan udostępnia programiście dwa typy obiektów związanych z urządzeniami, które mogą wykonywać polecania Vulkana:

- Urządzenia fizycznie,
- Urządzenia logiczne.

Urządzenia fizyczne modelują procesory graficzne i obliczeniowe, które obsługują sprzętowo lub programowo ten interfejs. Pozwala to programiście między innymi na wybór odpowiedniego urządzenia na systemach posiadających kilka takich urządzeń (na przykład zintegrowaną z procesorem kartę graficzną oraz dedykowaną kartę graficzną), co nie było możliwe w przypadku jego poprzednika. Z urządzeniami fizycznymi wiążą się odpowiednie struktury opisujące dostępną funkcjonalność oraz obowiązujące dla niego nim limity i wielkości dostępnych zasobów.

Urządzenia logiczne to abstrakcyjne uchwyty wykorzystywane w większości funkcji API. Odpowiadają one logicznemu urządzeniu, które wykorzystywać będzie aplikacja. Dzięki nim programista — po wybraniu danego urządzenia fizycznego — nie musi przejmować się jego szczegółami. W przyszłości możliwe będzie również stworzenie urządzenia logicznego, które grupuję zestaw urządzeń fizycznych, które będą rozdzielały między sobą pracę i dzieliły wspólnie zasoby. [2]

[2] Funkcjonalność taka wymagać będzie dostępnych kilku rozszerzeń, m.in. `VK_KHX_device_group`, które aktualnie są w trakcie rozwoju.

3.2.3. Warstwy

Jak wspomniano wcześniej, Vulkan zaprojektowany jest w sposób warstwowy. Sprowadza się to do dynamicznego wywoływania funkcji z API w zależności od włączonych warstw. Jeśli wszystkie warstwy są wyłączone, po wywołaniu funkcji z API sterowanie przechodzi bezpośrednio do sterownika. Jeśli jednak któreś warstwy są włączone, to sterowanie najpierw przechodzi przez wszystkie aktywne warstwy by w końcu trafić do sterownika. Dzięki temu możliwe jest uzyskanie efektów takich jak:

- Sprawdzanie czy API używane jest poprawnie,
- Mierzenie wydajności,
- Zapisywanie użytych poleceń,
- Wspomaganie debugowania aplikacji,
- I wiele innych...

Aktualnie dostępnych jest kilka warstw sprawdzających poprawność używania API oraz wspomagających debugowanie aplikacji. Dodatkowo programiści mogą tworzyć własne warstwy i rozszerzać istniejące o własną funkcjonalność. Wszystko to daje programiście ogromną elastyczność oraz pozwala na zaoszczędzenie zasobów, gdy aplikacja zostanie już w pełni przetestowana.

3.2.4. Kolejki

Kolejki to elementy API reprezentujące abstrakcyjne kolejki zadań do których programista może wysyłać swoje polecenia by zostały wykonane na urządzeniu. Każda kolejka może wspierać wybrany podzbiór następujących operacji:

- Operacje graficzne,
- Operacje obliczeniowe,
- Operacje transferowe,
- Operacje zarządzania zasobami rzadkimi (ang. *sparse*).

Kolejki pogrupowane są w **rodziny kolejek**, w których każda kolejka wspiera taki sam zestaw funkcjonalności. W praktyce rodziny kolejek odpowiadają zazwyczaj dostępnemu fizycznie sprzętowi, zatem programista mając do dyspozycji kilka kolejek, może wykonywać różne operacje jednocześnie. Częstą praktyką jest udostępnianie kolejki obsługującej wszystkie dostępne operacje oraz osobnej kolejki obsługujących tylko transfer danych, przez co programista może w optymalny sposób

przesyłać dane pomiędzy pamięcią RAM oraz VRAM jednocześnie wykonując inne obliczenia.

Warto dodać, że w przypadku obsługi przez sprzęt funkcjonalności *asynchronicznych obliczeń* (ang. *async compute*), implementacja jest w stanie wykonywać jednocześnie operacje obliczeniowe oraz operacje renderujące. Taka funkcjonalność wspierana jest aktualnie tylko przez najnowsze karty graficzne firmy AMD.[14, 15] [3]

3.2.5. Bufory komend

Wszystkie operacje wykonywanie na procesorze graficznym muszą zostać wysłane do kolejki danego urządzenia. Wysyłane zadania są pogrupowane wewnątrz buforów komend, a proces dodawania komend do bufora nazywa się „nagrywaniem bufora komend". Bufor taki to abstrakcyjna struktura opakowująca specyficzne dla danej implementacji struktury przechowujące dodane komendy. Dzięki nie programista może dodać kolejne komendy do wykonania na urządzeniu bez znajomości szczegółów implementacyjnych sterowników.

Nagrywanie komend do bufora odbywa się w trzech krokach:

1. Zasygnalizowanie rozpoczęcia nagrywania. Jeśli bufor był już nagrany, należy go wcześniej zresetować by mógł być znów używany. Ponadto jeśli bufor był wysłany do urządzenia, należy upewnić się przed jego modyfikacją, że urządzenie zakończyło wykonywanie operacji odwołujących się do tego bufora.

2. Wykonanie po stronie aplikacji funkcji dodających dane komendy do bufora. Każda komenda przyjmuje jako pierwszy argument uchwyt bufora, a wszystkie parametry użyte w komendach są kopiowane do bufora. Z tego względu zasoby stworzone na jej potrzeby mogą zostać zwolnione zaraz po wywołaniu danej funkcji. stworzone na jej potrzeby. Wykonywanie tych funkcji musi być synchronizowane przez aplikację — to znaczy dwa różne wątki nie mogą jednocześnie wykonywać operacji na tym samym buforze komend.

3. Zasygnalizowanie zakończenia nagrywania. Wszystkie ewentualne sprawdzenia błędów są odroczone do tego miejsca i użytkownik dowie się o ewentualnych problemach dopiero na tym etapie.

Tak nagrany bufor może zostać wysłany do kolejki, gdzie wszystkie zapisane w nim komendy zostaną asynchronicznie wykonane. Po jego wykonaniu, programista może go zniszczyć, zwalniając tym samym jego zasoby, zresetować, by móc ponownie go wypełnić innymi komendami, lub wysłać jeszcze raz te same komendy. Dzięki temu programista może wielokrotnie wykonywać polecenia nagrane tylko jeden raz.

[3] Funkcjonalność ta jest wspierana przez karty graficzne oparte o architekturę *Graphics Core Next* [13] użytą w wybranych modelach z serii *Radeon HD 7000* i nowszych.

Każdy bufor komend musi zostać stworzony z obiektu `VkCommandPool` reprezentującego abstrakcyjną pulę zasobów. Powodem tego jest chęć zminimalizowania ilość alokacji pamięci i przyśpieszenie tworzenia tych obiektów. Bardzo ważnym elementem tej architektury jest również fakt, iż dostęp do bufora komend oraz obiektu reprezentującego pulę zasobów, z których został utworzony, musi być synchronizowany przez programistę. Pozwala to programiście w środowisku wielowątkowym utworzyć osobne dla każdego wątku pule zasobów, z których będą tworzone bufory komend używane na danym wątku. Korzystając z interfejsu Vulkana w taki sposób, programista może wykonywać pracę na wielu wątkach bez korzystania z żadnych zewnętrznych mechanizmów synchronizacji zasobów.

Warto zauważyć, że nagrywanie buforów komend jest jedną z najbardziej wymagających operacji, zatem programista powinien upewnić się, że robi to w najbardziej optymalny sposób.

3.2.6. Potoki i deskryptory

We wcześniejszych interfejsach obiekty opisujące aktualnie wykonywany potok graficzny — lub obliczeniowy — były ustawiane częściowo poprzez ustawianie pewnych jego elementów. Wiązało się to z tym, że implementacja danego API nigdy nie wiedziała do końca co nastąpi za chwile i nie zawsze potrafiła w pełni zoptymalizować pracę. Vulkan wprowadza tu zmianę i udostępnia obiekt będący reprezentacją całego potoku, który z góry precyzuje wszystkie jego etapy oraz jawnie informuje o wszystkich wykorzystywanych w nim danych i sposobie ich przepływu pomiędzy kolejnymi etapami. Mając do dyspozycji pełną wiedzy na temat sposobu użycia interfejsu przez aplikację, sterownik może lepiej zoptymalizować pracę wykonywaną w takim potoku.

Zmienił się również sposób aktualizacji uniformów, które w starszych interfejsach jak OpenGL były ustawiane pojedynczo. W Vulkanie zgrupowane są one w zbiory deskryptorów co pozwala ustawić wszystkie jednocześnie. Pozwala to również w łatwy sposób pogrupować wykorzystywane uniformy według częstotliwości ich zmian i w jednej operacji zaktualizować tylko te, które zmieniły się od ostatniego polecenia rysowania.

By przyśpieszyć tworzenie deskryptorów, alokowane są one — tak jak bufory komend — z obiektów reprezentujących pewne pule zasobów.

3.2.7. Pamięć

Programista korzystający z Vulkana ma bezpośredni dostęp do wszystkich zasobów pamięci widocznych z punktu widzenia implementacji interfejsu. Różne rodzaje zasobów pamięci pogrupowane są w sterty, z których każda różni się rozmiarem oraz pewnymi cechami. Przykładowo jedna sterta może reprezentować pamięć VRAM, a

inna pamięć RAM. Niektóre pamięci mogą być widoczne bezpośrednio przez aplikację, a inne mogą być widoczne tylko dla urządzenia.

Każda sterta posiada kilkanaście typów pamięci, jaka może być z zaalokowana. Typy te zazwyczaj odpowiadają konkretnym zasobom, do jakich można je użyć. Dla przykładu obiekty obrazów mogą wymagać innego typu pamięci niż obiekty buforów. Typy również mogą różnić się tym, czy są *spójne*, czy może wymagają jawnych czyszczeń pamięci *cache*, o ile jej używają. Zasoby obrazów posiadają również stan nazywany *układem* (ang. *layout*). Ustawienie odpowiedniego układu dla konkretnej operacji jest wymagane, a korzystanie z zasobu będącego w złym stanie może prowadzić do ciężkich w zdiagnozowaniu błędów.

Programista odpowiada za poprawne alokowanie, używanie oraz zwalnianie pamięci i obiektów. Odpowiada również za wybór odpowiedniego typu z odpowiedniej sterty, co może mieć drastyczny wpływ na wydajność aplikacji. Dlatego więc kwestia odpowiedniego zarządzania pamięcią w Vulkanie jest bardzo istotna. Na koniec należy wspomnieć, że aplikacja ma pewne limity na to jak z pamięci korzysta. Przykładowo istnieje odgórny limit maksymalnej ilości alokacji pamięci. Z tego powodu faworyzuje się wykonywanie małej ilości dużych alokacji pamięci wewnątrz API oraz sub-alokowanie zasobów w ramach pozyskanych wcześniej pul zasobów.

3.2.8. Synchronizacja

Odpowiedzialność za synchronizacja w API Vulkan jest — podobnie jak zarządzanie pamięcią — w całości przerzucona na aplikację. Programista nie może zakładać, że kolejność wykonywania komend będzie taka sama, jak kolejność ich nagrywania i zgodna z kolejnością dodawania do kolejki buforów komend. Chcąc upewnić się, że kolejne komendy „widzą" efekty poprzednich, od których zależą, programista musi wskazać takie miejsca i jawnie w nich zsynchronizować dostęp do tych zasobów. Ponadto programista musi upewnić się, że Vulkan zakończył swoje obliczenia wykorzystujące dane obiekty, aby móc je zwolnić lub skorzystać z nich na nowo.

W tym celu API udostępnia programistom kilka typów *prymitywów synchronizujących* pozwalających na synchronizację pracy i dostępu do zasobów.

Obiekty grodzące

Obiekty grodzące (`VkFence`) wykorzystywane są do synchronizacji pomiędzy procesorem graficznym i procesorem ogólnym. Dzięki nim możemy poprosić o zatrzymanie pracy CPU do czasu, aż dana operacja zakończy się wykonywać na GPU. Ze względu na swój wysoki narzut, obiekty te powinny być stosowane tylko w sytuacjach, w których chcemy się upewnić, że możemy kontynuować pracę spodziewając

się, że operacje zostały już zakończone, lub wtedy, gdy czekać musimy niezależnie od tego faktu.

Przykładem wykorzystania tych obiektów może być chęć zwolnienia utworzonych wcześniej obiektów. Ze względu na manualne zarządzanie pamięcią to programista jest odpowiedzialny za zniszczenie utworzonych obiektów, ale może zrobić to, dopiero gdy wszystkie operacje wysłane do wykonania na GPU zostały zakończone. Zamiast czekać na zakończenie wszystkich operacji na GPU możemy poczekać tylko na te, które go wykorzystywały.

Semafory

Semafory (**VkSemaphore**) pozwalają na synchronizację pracy wykonywanej przez GPU zarówno wewnątrz pojedynczej kolejki jak i pomiędzy różnymi kolejkami. Precyzują one kiedy dany zasób zostanie udostępniony i na zakończenie jakich operacje dalsze wykonywanie powinno zaczekać. Programista ma tylko kilka dostępnych miejsc, w których może zlecić czekanie na dany semafor lub zlecić jego sygnalizację.

Przykładem wykorzystania semaforów może być zwykła prezentacja wyrenderowanego obrazu. Prezentacja powinna odbyć się dopiero po zakończeniu renderowania, zatem powinniśmy polecić zasygnalizowanie danego semafora w momencie zakończenia renderowania oraz wymusić czekanie na jego zasygnalizowanie zanim zaprezentujemy nową klatkę.

Zdarzenia

Zdarzenia (**VkEvent**) to obiekty przypominające obiekty grodzące, jednak w przeciwieństwie do nich:

- mogą one być używane tylko w obrębie jednej kolejki,
- mogą one być ustawiane i resetowane zarówno po stronie CPU jak i GPU,
- czekanie na sygnalizację może odbywać się tylko po stronie GPU.

Bariery pamięci

Bariery pamięci (**VkMemoryBarrier**) to kolejne obiekty pozwalające na synchronizację pracy wykonywanej na GPU. Dostępnych jest kilka rodzajów tych urządzeń, które różnią się rodzajem zasobów jakie chronią:

- bariery pamięci,
- bariery pamięci buforów,

- bariery pamięci obrazów.

Wykorzystywane są do precyzyjnego oznaczania zależności pomiędzy operacjami generującymi konkretne dane oraz operacjami, które dane te wykorzystują. Dzięki temu implementacja API dokładnie wie kiedy i na jakie dane musi zaczekać. Ponadto bariery pamięci obrazów mogą być wykorzystane do zmiany ich *układów*, których poprawne i dokładne ustawienie dla danej operacji, pozwala na szybszy dostęp do zasobów.

Ich poprawne użycie jest nieoczywiste, a chęć zachowania maksymalnej wydajności wymaga starannego przemyślenia gdzie powinny się one znaleźć oraz użycia minimalnej ilości potrzebnych barier z podanymi odpowiednimi etapami potoków tak, by praca GPU nie była wstrzymana dłużej niż jest to konieczne.

W porównaniu do pozostałych obiektów, pozwalają na bardzo dokładną synchronizację — z dokładnością do konkretnych etapów potoku — dzięki czemu są idealnym sposobem na synchronizację pomiędzy kolejnymi po sobie operacjami.

Przykładem sytuacji, w której wymagane są bariery pamięci jest renderowanie z algorytmem *deferred shading* [44, 45] — kolejne jego etapy zależą od poprzednich, a więc wymagają zakończenia nad nimi prac. Gdyby nie zastosowano żadnych mechanizmów synchronizujących, implementacja API mogłaby wykonać je jednocześnie lub w niepożądanej kolejności, co mogło by skutkować wyrenderowaniem „śmieci".

3.3. Podsumowanie

Jak widzimy, API Vulkan różni się znacząco od swojego poprzednika. W jego architekturze widzimy wzorowanie się na aktualnych trendach i uznanych wzorcach projektowania interfejsów, dzięki czemu API wydaje się przyjemne i łatwe w użyciu. Zauważyć od razu można też to, jak bardzo „opisowy" on jest — na każdym kroku wymaga od jego użytkownika sprecyzowania wszystkich zamiarów „do przodu". Nawet jeśli pewna funkcjonalność nie będzie używana, musimy o tym zawczasu poinformować implementację.

Interfejs rozwiązuje również wiele problemów znanych z jego poprzedników w sensowny sposób, co jest zdecydowaną zaletą. Ponadto znacznie lepiej modeluje aktualny sprzęt i udostępnia elementy pozwalające na lepsze wykorzystanie dostępnego sprzętu.

Znacznie wyższy poziom trudności poprawnego użycia, wynikający między innymi ze względu na ręczne zarządzanie pamięcią oraz synchronizację, może nie przypaść wszystkim do gustu i z tego powodu API prawdopodobnie nie będzie tak często używane — szczególnie w przypadku prostych i niewymagających wysokiej wydajności aplikacji.

Rozdział 4.

Przegląd istniejących testów

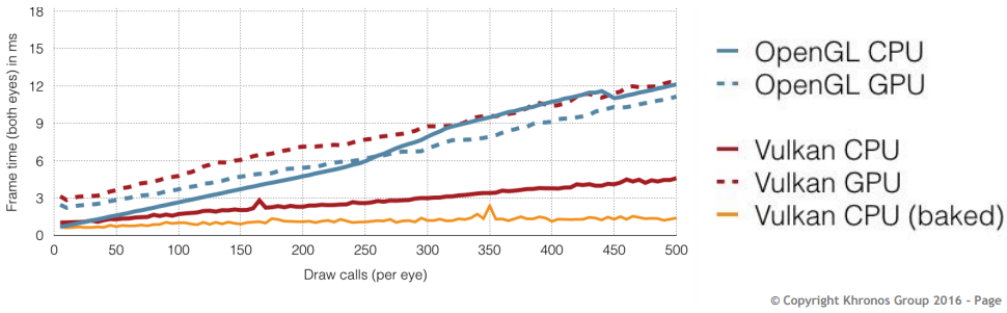
Od wydania Vulkana minął już ponad rok. Przez ten czas powstało wiele aplikacji z niego korzystających, a także wiele osób postanowiło zbadać jak na tle innych interfejsów wypada on w praktyce pod względem wydajności. Stworzone zostały liczne aplikacje testowe, silniki graficzne i gry obsługujące jednocześnie kilka interfejsów graficznych. W pracy tej skupię się na omówieniu wyników porównujących wydajność Vulkana z jego poprzednikiem — API OpenGL.

Warto jednak pamiętać, że część z tych testów została przeprowadzona na wczesnych wersjach sterowników, które nie są jeszcze tak zoptymalizowane jak sterowniki dostępne od ponad dwóch dekad dla interfejsu OpenGL. Ponadto część aplikacji wprowadziła obsługę nowego API jako *wrapper* na już zaimplementowane systemy renderujące korzystające ze starszych, już wykorzystywanych interfejsów, co może generować dodatkowy narzut negatywnie wpływający na wydajność i sprawiać, że aplikacja nie korzysta z wszystkich zalet Vulkana.

4.1. Przegląd autorskich testów

4.1.1. Test Khronos DevU w Seulu

W roku 2016, na warsztatach Khronos DevU zorganizowanych w Seulu przedstawiony został prosty test stworzony przez **Corta Strattona** — osobę pracującą nad niskopoziomowym kodem renderującym w konsolach PS3 oraz PS4. Test opisany w [30] składa się z jednej statycznej sceny — pokoju wypełnionego obracającymi się obiektami. Każdy obiekt składa się z około 800 wierzchołków, co przekłada się na około 1400 trójkątów. Każdy obiekt posiada unikalne zasoby takie jak tekstury i bufory danych. Wszystko to rysowane jest dwukrotnie aby zasymulować osobne renderowanie dla każdego oka. Ilość rysowanych obiektów zmienia się w przedziale od 5 do 500 obiektów, a czasy mierzone są jako średnia z 60 wyrenderowanych klatek.



Rysunek 4.1: Wyniki testu przedstawionego na warsztatach Khronos DevU w Seulu. [30]

Na wynikach możemy zaobserwować dwie rzeczy:

- Czas jaki GPU potrzebuje do wykonywana operacji jest bardzo zbliżony w obu API. Różnice są minimalne na niekorzyść Vulkana, co może sugerować gorszą optymalizację wówczas dostępnych sterowników.
- Wraz ze wzrostem ilości renderowanych obiektów, czas potrzebny na wykonanie operacji po stronie CPU rośnie w obu przypadkach liniowo, jednak dla API OpenGL rośnie on znacznie szybciej. Średni stosunek czasu wykonywania klatki po stronie CPU dla API OpenGL i Vulkan odczytany z powyższego wykresu wynosi 2.8 i wartość ta odpowiada wielkości narzutu starszego API względem swojego następnika.

Po wynikach pomiarów dla całych klatek zaprezentowano również wyniki pomiarów rysowania tylko jednego „oka", jednak różnica względna między wynikami w poszczególnych API jest bardzo zbliżona do wcześniej uzyskanych wyników. Na koniec zaprezentowany został również wykres przedstawiający czasy z wielokrotnym wykorzystaniem raz nagranego bufora komend. Wartości z tych pomiarów zaznaczone są na rysunku 4.1 pomarańczową linią. W tym przypadku czas wykonywania na CPU był praktycznie stały. Potwierdza to intuicję i pokazuje jak duży zysk potrafimy uzyskać wykorzystując w pełni możliwości nowego API.

4.1.2. Demo Gnome Horde

Gnome Horde to demo przygotowane przez firmę **PoverVR** w wersji wykorzystującej API OpenGL ES (mobilna wersja OpenGL-a) oraz Vulkan. Demo działa na platformie Android i zostało opublikowane w sierpniu 2015 roku w [31], zatem finalne rezultaty, które można osiągnąć wykorzystując nowe API mogą się różnić.

Głównym założeniem dema było wykonywanie podobnego kodu przy wykorzystaniu obu API. Dema nie korzystają z żadnych rozszerzeń ani mechanizmu instancjonowania — każda komenda rysująca mogłaby renderować osobną geometrię z

osobnymi materiałami czy teksturami i nie powinno to wpłynąć na czas wykonywania po stronie CPU.

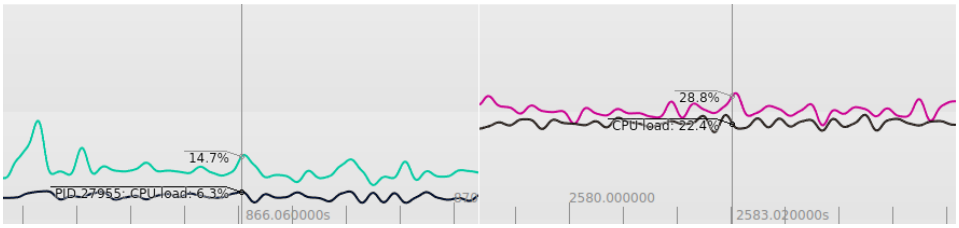
Demo składa się z ogromnej sceny po której porusza się kamera. Widoczna jest powierzchnia, na której znajduje się duża ilość kolorowych krasnali. Wnioski płynące z tego dema dla wersji wykorzystującej interfejs Vulkan możemy przedstawić jako:

- Zapewnienie płynnego odtwarzanie przez cały czas, podczas gdy konkurencyjna wersja korzystająca ze starszego API ma drastyczne spadki wydajności, szczególnie przy szybkiej pracy kamery kiedy to wydajność spada do kilku klatek na sekundę.

- Wersja wykorzystująca API Vulkan wykorzystuje w podobny sposób wszystkie 4 dostępne wątki rozkładając między nimi równomiernie pracę, podczas gdy wersja oparta o OpenGL ES zazwyczaj wykorzystuje tylko jeden z dostępnych rdzeni choć obciążenie pomiędzy nimi dynamicznie przemieszcza się pomiędzy różnymi rdzeniami.

- Wersja z Vulkanem nie wykorzystuje w pełni żadnego rdzenia zatem można stwierdzić, że jest ograniczana przez moc GPU. Wersja OpenGL ES przez większość czasu wykorzystuje w pełni któryś z rdzeni, co pozwala sugerować, że jest ograniczona przez moc CPU.

Rysunek 4.2: Kadr z pracy dema Gnome Horde przedstawiającego obciążenie CPU i wskaźnik FPS dla Vulkana (po lewej) i OpenGL ES (po prawej). [31]

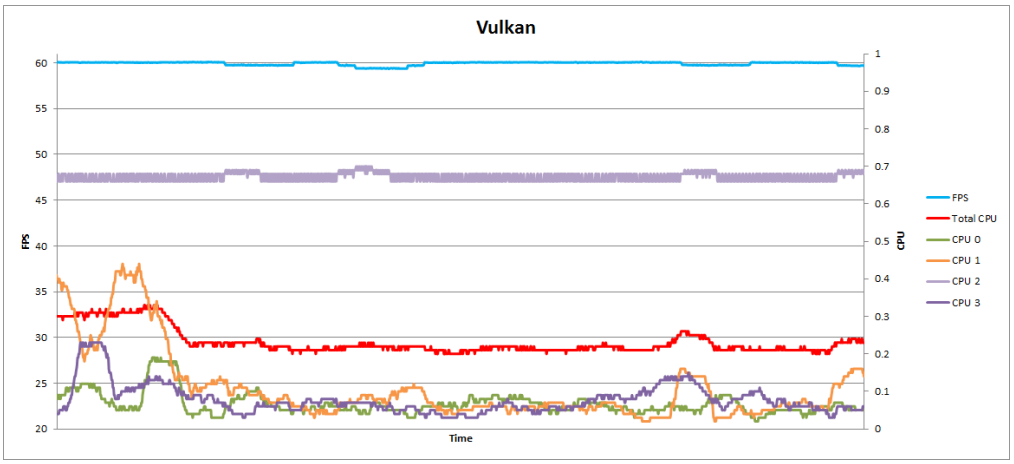


Rysunek 4.3: Obciążenie CPU i wskaźnik FPS podczas pracy dema Gnome Horde z użyciem Vulkana (po lewej) i OpenGL ES (po prawej). Dolna linia to obciążenie CPU przez aplikację, a górna to obciążenie całego systemu. [31]

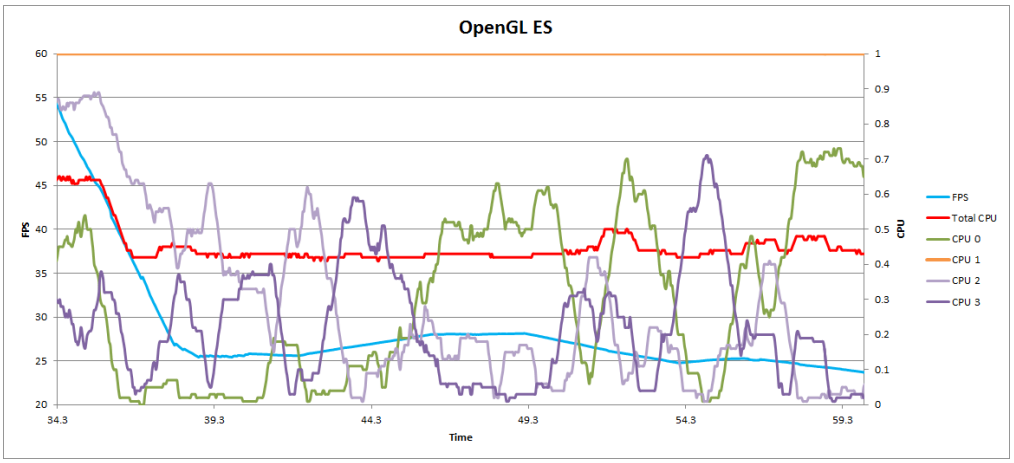
4.1.3. Demo Satelite navigation

Kolejne demo przygotowane przez **PowerVR** zostało opisane przez jego twórców w [32]. Tym razem demo przypomina prawdziwą, dostępną dla użytkowników końcowych, aplikację będącą nawigacją samochodową.

W testach przeprowadzonych przez autorów dema, implementacja oparta o API Vulkan zapewnia ciągłą płynność ze znacznie wyższą ilością klatek na sekundę bez zauważalnych spadków wydajnościowych spowodowanych ładowaniem nowych danych, zaś wersja wykorzystująca OpenGL ES wykonuje w każdej klatce znacznie więcej pracy na CPU. Jak twierdzą autorzy, dzieje się tak, gdyż implementacja wykorzystująca nowsze API potrafi raz nagrać dany fragment mapy, który się już nie zmienia, a następnie tylko wykonać nagrane komendy. Druga implementacja zmuszona jest do przekazywania wszystkich komend GPU za każdym razem.



Rysunek 4.4: Wydajność i obciążenie CPU dla API Vulkan. [32]



Rysunek 4.5: Wydajność i obciążenie CPU dla API OpenGL ES. [32]

4.1.4. Demo Stardust

Intel na konferencji SIGGRAPH 2015 zaprezentował swoje demo wykorzystujące API Vulkan oraz OpenGL nazwane Stardust[33], by pokazać zalety, jakie nowsze z nich może zaoferować względem poprzedników. Samo demo prezentuje system renderowania efektów cząsteczkowych.

Wersja OpenGL zapewnia około 25 klatek na sekundę oraz zauważalne wykorzystanie mocy CPU podczas gdy praca wykonywana jest prawie w całości tylko na jednym rdzeniu — pozostałe wykorzystywane są w znikomej ilości. Wykorzystywany rdzeń jest zajęty w 100 procentach, zatem możemy założyć, że aplikacja jest ograniczana przez CPU.

Po przełączeniu się na implementację wykorzystującą API Vulkan wydajność wzrasta dwukrotnie do około 50 stabilnych klatek na sekundę, przy czym obciążenie CPU spada o około 2/3 do niskiego poziomu. Zauważyć również można równomierne rozłożenie pracy na wszystkie dostępne rdzenie, które wykorzystywane są w około 20 procentach. To pozwala sugerować, że ta implementacja ograniczona jest przez moc GPU.

4.1.5. Test ARM

Test przeprowadzony przez firmę ARM i opisany w [34] skupia się na porównaniu wykorzystania procesora CPU oraz zużywanej energii przez układ *SoC*.

Na filmie [35] prezentującym wyniki testu po lewej stronie ekranu widzimy uruchomioną implementację wykorzystującą mobilną wersję API OpenGL, która wykorzystuje w sporym stopniu pierwszy z dostępnych rdzeni, gdy pozostałe są w spoczynku. Po prawej widzimy zaś implementację z Vulkanem, która to wykorzystuje wszystkie cztery rdzenie równomiernie w znacznie mniejszym stopniu. Autorzy testu sugerują, że dzięki temu system na którym została uruchomiona druga implementacja wykorzystał około 15 procent mniej energii poprzez zmniejszenie napięcia i taktowania rdzeni oraz możliwość uruchomienia na słabszych rdzeniach pobierających mniejszą ilość energii. [1]

4.2. Przegląd testów opartych na silnikach graficznych

4.2.1. Unity

W trzecim kwartale 2016 roku silnik Unity otrzymał wstępną wersję wsparcia dla omawianego API. Autorzy silnika w swoim artykule [38] opisują, że przejście na

[1] W urządzeniach mobilnych często spotykamy układy w systemie *big.LITTLE* który składa się z kilku wolnych rdzeni o niskim poborze prądu oraz kilku mocnych, które pobierają więcej energii.

to API zapewnia zysk w wydajności na poziomie około 35 procent na platformie Android nawet w trybie jednowątkowym.

Wersja ta jest testowa, zatem na rzetelne wyniki należy zapewne poczekać, aż prace nad implementacją obsługi Vulkana zostaną zakończone. W sieci dostępny jest jednak test, pokazujący jednocześnie pracę silnika na obu API. W teście tym zauważyć możemy, że wydajność w obu wersjach jest zbliżona, jednak na wykresach wskazujących obciążenie CPU zauważyć możemy znacznie mniejsze jego wykorzystanie w wersji wykorzystującej Vulkana.

4.2.2. Xenko

W materiale [39] zamieszczonym w serwisie YouTube przez twórców silnika widzimy porównanie prędkości renderowania sceny w obu API. Z filmiku wynika, że renderer oparty na Vulkanie jest wydajniejszy — w zależności od aktualnie rysowanej sceny — o od 50 do 300 procent niż wersja oparta na API OpenGL.

Niestety w teście widzimy również, że w danym momencie szybsza implementacja wykonuje o około 10 procent mniej poleceń rysujących, przez co ciężko ocenić jednoznacznie rzetelność przeprowadzonego testu, który wykonuje inną pracę w różnych trybach. Niemniej w mojej ocenie możemy wysnuć wnioski, że różnica na korzyść nowszego API jest zauważalna, a nawet spora.

4.2.3. Inne

Twórcy silników graficznych **Unreal Engine** oraz **CryENGINE** zapowiedzieli [40, 41] wsparcie dla API Vulkan, jednak na czas pisania tej pracy nie zostało ono jeszcze dodane. Prawdopodobnie po jego wprowadzaniu pojawią się testy oparte o te silniki graficzne. Biorąc pod uwagę popularność tych silników oraz ilość gier tworzonych na nich, wydaje się warte, by przyjrzeć się wówczas osiągniętym na nich wynikom.

4.3. Przegląd testów opartych na grach komputerowych

4.3.1. The Talos Principles

The Talos Principles była pierwszą grą, która miała zaimplementowany rendering z wykorzystaniem Vulkana. Twórcy zaimplementowali jego obsługę jako nakładkę na istniejący renderer wykorzystujący OpenGL 2.1 oraz Direct3D w wersji 9. Jak sami twierdzą ([42]) — uzyskana wydajność jest zadowalająca, ale nie jest to rozwiązanie idealne, gdyż nie wykorzystuje w pełni wszystkich możliwości nowego API. Twórcy ciągle pracują nad ulepszeniami silnika oraz chwalą się, że w aktualnej wersji wydajność jest od 50 do 100 procent większa.

Niezależne testy [43] najnowszej wersji — po naprawie błędów oraz aktualizacji sterowników — wskazują, że wersja wykorzystująca Vulkana jest szybsza o około 30 procent.

4.3.2. Doom 2016

Doom (2016) to remake popularnej serii gier stworzonej przez **id Software**. Firma ta zawsze chwalona była za dobrą optymalizację oraz umiejętność do wyciśnięcia pełnej mocy z każdego sprzętu. Gra ta otrzymała wsparcie dla API Vulkan po premierze w postaci łatki. Istotnym faktem jest również wykorzystanie — jeśli tylko jest to możliwe — najnowszej wersji API OpenGL. Biorąc to pod uwagę oraz doświadczenie programistów z id Software, można założyć, że implementacja na tym API jest świetnie zoptymalizowana.

Warto zauważyć, że gra jako jedna z pierwszych wspierać ma nową funkcjonalność dostępną w Vulkanie — asynchroniczne obliczenia (ang. *asynchronous compute*), czyli funkcjonalność pozwalająca na równoległe i asynchroniczne wykonywanie zadań obliczeniowych i graficznych jednocześnie. Funkcjonalność ta w chwili pisania tej pracy dostępna jest jedynie na najnowszych kartach graficznych firmy AMD.

Gra ta była — z racji obsługi obu API graficznych — często wykorzystywana do ich porównywania. W pracy tej odniosę się do testów przeprowadzonych przez redakcję serwisu PCGamer wykorzystującej stosunkowo aktualną na czas pisania tekstu wersję gry oraz sterowniki.

Z dostępnych testów [36, 37] możemy wysunąć następujące wnioski:

- Wydajność na kartach ze „stajni” AMD wzrasta ze zmianą API o około 25 do 30 procent wzwyż w większości testowanych kart graficznych.

- Wydajność w testach korzystających z kart graficznych firmy Nvidia wzrasta jedynie na najwydajniejszych konfiguracjach z takimi kartami graficznymi jak GTX 1080 czy GTX 1070. Pozostałe karty nie zanotowały większych różnic pomiędzy API. Należy zwrócić jednak uwagę na zwiększoną stabilność prędkości renderowania — w większości kart graficznych testy sprawdzające średnią z 3 procent najwolniejszych klatek wzrosły o około 10-20 procent, co przekłada się na brak chwilowych spadków wydajności z nowym API. W przypadku karty GTX 1080 był to wzrost rzędu 50 procent.

- Testy w wyższych rozdzielczościach (1440p, 2160p) pokazują utrzymujący się wzrost wydajności na kartach firmy AMD, gdzie wyniki u konkurencyjnej firmy pozostają bardzo zbliżone na obu API.

4.4. Podsumowanie

Jak dobrze widzimy na powyższych testach, nowe API potrafi dostarczyć programistom możliwości optymalizacji silników i gier. Niestety w większości — poza grą Doom 2016 — zyski te uzyskane są przede wszystkim tam, gdzie wydajność zależy w sporej mierze od szybkości naszego procesora głównego. Poza zwiększoną wydajnością, Vulkan potrafi zaoferować znacznie lepszy rozkład pracy na dostępne rdzenie, których w dzisiejszym sprzęcie jest coraz więcej. Skorzystać z tego mogą również urządzenia mobilne, na których powinniśmy zauważyć zmniejszony pobór mocy w takich sytuacjach.

Niestety tylko jeden z omawianych testów skupiony był na porównaniu samych interfejsów graficznych, przez co wyniki w sporej mierze zostały zaburzone przez jakość implementacji renderingu dla obu API oraz inne podzespoły silników gier, które również wymagały zasobów. Warto byłoby więc przeprowadzić więcej testów skupiających się na porównaniu samych API bez zewnętrznych zaburzeń wyników. Ponadto część z implementacji zapewne napisana została zgodnie z dotychczasową architekturą używanych silników, przez co prawdopodobnie nie wszystkie elementy nowego API wykorzystywane są w stu procentach.

Ciężko stwierdzić na tym etapie, które z mechanizmów Vulkana potrafią pozwolić na poprawę wydajności. Zaobserwowane wcześniej wyniki dają jednak motywację do stworzenia własnych testów, które skupiać się będą na pomiarach narzutu na CPU korzystając z obu API. Dzięki temu będziemy w stanie realnie oszacować jakie zyski i w jakich scenariuszach możliwe są do osiągnięcia. Warto zatem zbadać wydajność zarówno w scenariuszach ograniczonych przez moc CPU jaki i GPU, a także porównać wydajność korzystając z nowych mechanizmów wprowadzonych w API Vulkan jak na przykład *PushConstants*.

Rozdział 5.

Przeprowadzone testy

5.1. Projekt

5.1.1. Opis projektu

Na potrzeby tej pracy stworzyłem projekt o nazwie **GL_vs_VK**, w którym zaimplementowałem — z wykorzystaniem obu interfejsów graficznych — kilka różnych scen i testów w różnych wariantach. Każdy z nich zawiera część wspólną, niezależną od wykorzystywanego sprzętu tak, by zapewnić jednakowe warunki testowe. Część zależna od używanego API pisana jest ręcznie tak, by zapewnić możliwe maksymalną wydajność korzystając z danego podzbioru dostępnej funkcjonalności. Więcej informacji o projekcie znajduje się w Dodatku A: Projekt GL_vs_VK.

Motywacją do stworzenia tego projektu była chęć przeprowadzenia testów skupiających się na zmierzeniu narzutu na procesor CPU podczas korzystania z obu omawianych API. Dostępne bowiem testy skupiają się w większości na wydajności całej aplikacji, na którą składa się wiele modułów, co nie pozwala jasno przedstawić różnic w wydajności obu API. Chcemy również móc przeanalizować wydajność interfejsów zarówno w teście syntetycznym, który powinien pokazać pełne różnice w wydajności API, oraz w testach modelujących praktyczne zastosowanie interfejsów przy implementacji popularnych algorytmów graficznych. Dzięki temu będziemy w stanie odpowiedzieć na pytanie — ile maksymalnie możemy zyskać na wydajności, oraz w jakich warunkach uzyskamy największy zysk.

5.2. Testy

Projekt zawiera cztery testy, z których każdy ma za zadanie zasymulować inny scenariusz użycia API graficznego w aplikacji multimedialnej lub grze. Pierwsze trzy testy skupiają się na wydajności renderowania klatek. Czwarty test mierzy czas potrzebny na inicjalizację całego potoku graficznego.

5.2.1. Elementy wspólne

Pierwsze trzy testy skupiają się na mierzeniu wydajności renderowania klatek. W celu uzyskania maksymalnej wydajności, inicjalizacja wszystkich możliwych obiektów została przeniesiona do fazy ładowania, przez co faza renderująca wykonuje minimalną ilość operacji na każdą klatkę. Z tego względu wszystkie elementy takie jak bufory wierzchołków, programy cieniujące czy obiekty potoków ładowane są przed rozpoczęciem wykonywania pomiarów.

Faza renderowania polega na zaktualizowaniu stanu testów — na przykład pozycji obiektów — co odbywa się na CPU, aktualizacji buforów i uniformów, po czym wykonywane są polecenia rysujące. Na koniec aplikacja prosi o prezentację aktualnej klatki. Czas każdej iteracji tej fazy jest mierzony i przekazywany do modułu mierzącego wydajność.

Etap zwalniania zasobów wykonywany jest po zakończeniu pomiarów, zatem nie wpływa na wyniki testów.

Wersje wielowątkowe różnią się od wersji jednowątkowych tym, że aktualizacja stanu wykonywana jest na wszystkich dostępnych wątkach, zamiast tylko na jednym. Ponadto w wersji wykorzystującej API Vulkan, budowanie buforów komend wykonywane jest wielowątkowo. Ze względu na ograniczoną logikę, nie wszystkie testy w wersji z API OpenGL posiadają wersje wielowątkowe.

Należy dodać, że implementacje wykorzystujące interfejs Vulkan tworzą zawsze przynajmniej trzy bufory klatek, jeśli tylko jest taka możliwość. Każdy obraz posiada własny zestaw buforów komend tak, aby renderowanie danej klatki było niezależne od renderowania pozostałych.

By zmaksymalizować ilość renderowanych klatek, wszystkie wersje wykorzystują również tryb prezentacji z wyłączoną synchronizacją pionową.

5.2.2. Test 1 — scena statyczna z dużą ilością obiektów

Test polega na wyświetleniu ustalonej ilości obiektów na ekranie. Każdy z obiektów w przypadku tego testu jest kulą — współdzieląc przy tym bufor wierzchołków — o ustalonej szczegółowości, jednak obiekty różnią się od siebie pozycją oraz kolorem.

Test ten pozwala na wybranie ilości obiektów rysowanych na ekranie oraz tego, jak dokładne będą rysowane kule, a konkretniej z ilu wierzchołków będą się one składać. Dodatkowo test posiada zmienną pozwalającą decydować ile razy, na klatkę, stan każdej kuli ma zostać zaktualizowany. Dzięki temu możemy wymusić różne obciążenie procesora CPU symulując przy tym wykonywanie skomplikowanych obliczeń związanych z — na przykład — poruszaniem się czy kolizją obiektów. Zmienne te dostępne są w pliku `BaseBallsSceneTest.cpp` pod jego nagłówkiem.

By test symulował renderowanie różnych obiektów, nie są wykorzystywane techniki polegające na grupowaniu wspólnej geometrii lub innych optymalizacjach. Nie jest również wykorzystywane *instancjonowanie*. Dzięki temu największy narzut w tym teście — w zależności od ustawień — dają następujące jego elementy:

- ilość poleceń rysujących,
- ilość wierzchołków na każdy obiekt.

W przypadku użycia *instancjonowania*, narzut wynikający z ilości poleceń rysujących byłby znacznie mniejszy, gdyż wystarczyłoby tylko jedno polecenie rysujące, jeśli wszystkie obiekty są takie same. W przypadku różnych obiektów nie jesteśmy w stanie użyć tego mechanizmu.

Bazowy moduł testu odpowiada za wygenerowanie danych takich jak pozycja, prędkość poruszania oraz kolor dla wszystkich obiektów, oraz ich aktualizację. Zawiera również metodę, która aktualizuje tylko podzbiór obiektów w danym przedziale indeksów, która wykorzystywana jest w wersji wielowątkowej tego testu.

Każdy rysowany obiekt posiada przypisane do siebie dwie zmienne przekazywane do GPU:

- pozycja obiektu na ekranie,
- kolor obiektu.

W wersji korzystającej ze starszego interfejsu elementy te przekazywane są w postaci uniformów. Nowsze API udostępnia mechanizm *PushConstants*, który pozwala na szybką aktualizację małych uniformów i to on został wykorzystany jako mechanizm aktualizacji uniformów w wersji korzystającej z Vulkana.

Jest to jedyny test posiadający wersję wielowątkową zarówno w przypadku API OpenGL jak i Vulkana. Powodem tego jest wykonywanie potencjalnie sporej ilości obliczeń związanych z aktualizacją stanu na CPU, co może przełożyć się na zwiększoną wydajność gdy korzystamy z wielu wątków. Zbiór wszystkich obiektów dzielony jest wówczas na prawie równe zbiory i każdy zbiór aktualizowany jest przez osobny wątek. Dzięki takiej architekturze, nie są konieczne żadne mechanizmy synchronizujące podczas wykonywania aktualizacji stanu obiektów.

5.2.3. Test 2 — scena dynamiczna z terenem wykorzystującym LoD

Test ten polega na załadowaniu mapy wysokościowej terenu (ang. *heightmap*) z pliku obrazu o rozmiarze 1024x1024. Odpowiada to mapie złożonej z 1023 pól w obu wymiarach. Po załadowaniu mapy tworzone dla niej są:

- Bufor wierzchołków zawierający 4 wartości (x, y, z, w) dla każdego wierzchołka.

- Bufor indeksów wierzchołków, z którego pobierane będą indeksy odpowiednich wierzchołków danego pola.
- Drzewo czwórkowe, zawierające dla każdego pola na mapie ilość indeksów oraz ich pozycję (ang. *offset*) w pamięci bufora indeksów, potrzebnych do jego narysowania.

Implementację tego mechanizmu znaleźć można w klasie `TerrainLoD`. Klasa ta w trakcie rysowania wykorzystywana jest do wywoływania poleceń rysujących w danym API z wykorzystaniem zapisanych w drzewie czwórkowym wspomnianych wyżej wartości. Wykorzystuje ona rekursywny algorytm *level-of-detail* do poruszania się po stworzonym wcześniej drzewie czwórkowym, który możemy przedstawić w uproszczeniu jako następujące kroki:

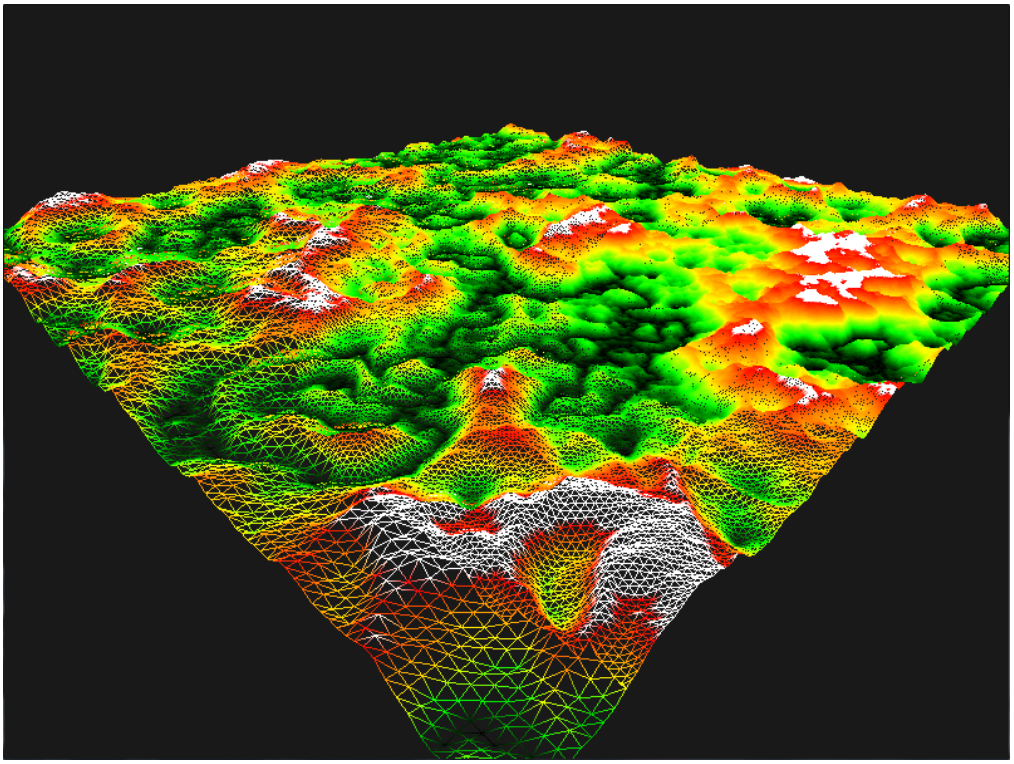
Listing 5..1: Uproszczony rekursywny algorytm *LoD* zaaplikowany do drzewa czwórkowego.

```
ExecuteRecursiveLoD(node, position, drawCallback) :=
    if (node.isLeaf()) then
        drawCallback(node.indexCount, node.indexOffset);
        return;

    d = distance(node.centerPosition(), position);
    s = node.maxSideSize();
    if (d / s >= CONST_LOD_FACTOR) then
        drawCallback(node.indexCount, node.indexOffset);
        return;

    for (subNode in node.childs()) do
        ExecuteRecursiveLoD(subNode, position, drawCallback);
```

Teren kolorowany jest wewnątrz shadera fragmentów na podstawie wysokości każdego wierzchołka tak, by przypominał mapę topograficzną. Dodatkowo teren rysowany jest w formie siatki niewypełnionych trójkątów (tryb *wireframe*), by łatwiej było zaobserwować działanie algorytmu *LoD*. Kamera ustawiona jest w stałym miejscu nad jednym z rogów mapy, zaś symulowane jest poruszanie się pozycji *gracza* po okręgu wokół środka mapy tak, by widoczne były zmiany rysowanych trójkątów.

Aktualizacja stanu sprowadza się do aktualizacji pozycji *gracza* oraz aktualizacji zależnych macierzy przekształceń, zatem nie obciąża ona w znaczący sposób procesora głównego. Z tego względu test nie posiada wersji wielowątkowej dla API OpenGL. Generowanie całego drzewa czwórkowego odbywa się w fazie ładowania, zatem nie wpływa na wyniki pomiarów.

Test w trakcie renderowania ustawia jeden uniform będący jedną macierzą przekształceń *Model-View-Projection* zawierającą wszystkie niezbędne przekształcenia do wyrenderowania trójwymiarowej sceny w danej klatce. Rysowanie odbywa się poprzez przechodzenie tym samym algorytmem *LoD* przez drzewo czwórkowe i dla każdego rysowanego pola mapy wołana jest funkcja rysująca.



Rysunek 5.1: Klatka z drugiego testu. Przedstawia siatkę terenu narysowaną przy użyciu algorytmu *LoD*.

Każdy wierzchołek drzewa czwórkowego w praktyce opisuje dwa trójkąty o odpowiednich rozmiarach i pozycji. Jeśli dany wierzchołek uznany jest za odpowiednio dokładny, wykonywana jest funkcja rysująca dany fragment. Jeśli jest on zbyt duży, algorytm rekursywnie wykonuje się dla wszystkich jego *dzieci*.

Wersja wielowątkowa zaimplementowana dla interfejsu Vulkan różni się od pozostałych tym, że budowanie bufora komend podzielone jest na dokładnie cztery wątki. Powodem tego jest specyficzna architektura drzewa czwórkowego. Początkowy podział korzenia drzewa wykonywany jest na głównym wątku po czym każde poddrzewo jest obsługiwane przez osobny wątek. W zależności od pozycji *gracza* może prowadzić to do sytuacji, w których podział ten jest nierównomierny, co może przekładać się na niestabilną ilość klatek.

Ciekawym wyzwaniem mogłoby być stworzenie algorytmu, który efektywnie i równomiernie rozdziela pracę na wszystkie dostępne wątki, jednak Nie zostało to zaimplementowane w obecnej wersji testu. Warto zauważyć, że przy obecnej implementacji największe różnice zauważylibyśmy tylko wtedy, gdy pozycja *gracza* ustawiona jest na któryś z rogów mapy. W wykonywanych testach pozycje te są ustawianie znacznie bliżej środka mapy, zatem wszystkie wątki powinny wykonywać sporą część pracy. Ponadto testy przeprowadzone są na komputerze, którego procesor obsługuje jednocześnie maksymalnie cztery wątki.

5.2.4. Test 3 — scena statyczna z mapowaniem cieni

Test trzeci polega na wyrenderowaniu statycznej sceny składają się z:

1. *Podłogi* przypominającej szachownicę. Każde pole składa się z sześcianu rysowanego naprzemiennie w ciemniejszym lub jaśniejszym kolorze.
2. Sześcianów rysowanych w nad *podłogą* w równych odstępach naprzemiennie wyżej i niżej.
3. Złożonej pod względem geometrii kuli o znacznie większych wymiarach niż pozostałe obiekty, znajdującej się samym środku sceny nad *podłogą*.

Wszystkie elementy rysowanej sceny renderowane są wraz z realistycznym i dynamicznym odwzorowaniem cieni. Programista, chcąc uzyskać taki efekt ma do dyspozycji kilka technik, z których najpopularniejszymi są:

- mapowanie cieni,
- cienie objętościowe.

Każda z tych technika ma wiele wariantów różniących się aspektami takimi jak:

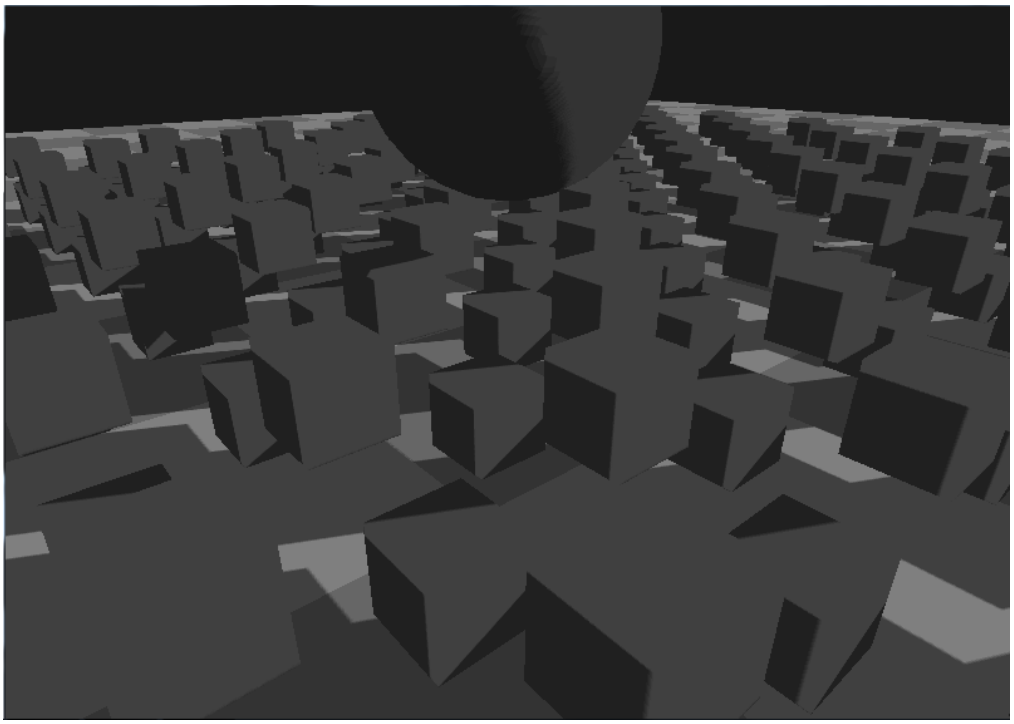
- jakość generowanych cieni,
- złożoność implementacji,
- narzut na wydajność,
- wsparcie dla efektów takich jak miękkie cienie.

W pracy tej zaimplementowano pierwszą ze wspomnianych technik — **mapowanie cieni** (ang. *shadow mapping*)[46]. Technika ta wyróżnia się możliwością uzyskania miękkich cieni, gdzie część fragmentów jest w półcieniu, oraz brakiem wrażliwości — w kontekście rysowanych cieni — na stopień złożoności geometrii. Jedną z jej poważniejszych wad jest natomiast możliwość uzyskania drobnych artefaktów przy wyznaczaniu fragmentów znajdujących się w cieniu.

Technika ta sprowadza się do wyrenderowania tej samej sceny dwukrotnie:

1. Scena renderowana jest z perspektywy światła rzucającego cień, a renderowane są tylko obiekty rzucające cień. Ponadto wykorzystywany jest tylko bufor głębokości, co sprawia, że etap ten jest wykonywany szybciej, niż gdyby rysować pełną klatkę.
2. Scena rysowana jest z perspektywy kamery, a każdy widoczny fragment testowany jest z użyciem wcześniej wygenerowanego bufora głębokości pod kątem jego widoczności ze źródła światła.

Końcowym elementem testu jest poruszająca się dookoła kuli kamera, dzięki której możemy obejrzeć całą scenę i zaobserwować jak rzucane są cienie na obiektach.



Rysunek 5.2: Klatka z trzeciego testu. Na scenie widoczne są cienie rzucane przez obiekty.

Wspomniany bufor głębokości to najczęściej dwuwymiarowa tekstura, której wielkość wpływa na jakość otrzymanych cieni. W przypadku tego testu zastosowano teksturę o rozmiarze 4096×4096 o precyzji 32 bitów. By poprawić jakość otrzymanych cieni, wyeliminować drobne artefakty na granicy fragmentów zaciemnionych i oświetlonych oraz uzyskać efekt rozmytych cieniu, zaimplementowałem w shaderze fragmentów algorytm rozmycia cieni *Percentage Closer Filtering*[47, 48], który polega na zebraniu wielu próbek z okolicy danego fragmentu i uśrednieniu otrzymanych wyników.

Ze względu na brak skomplikowanych aktualizacji stanu po stronie CPU, ten test również nie posiada wersji wielowątkowej z użyciem API OpenGL. W przypadku implementacji korzystającej z API Vulkan konieczne było zastosowanie bariery pamięci obrazu dla tekstury głębokości, gdyż w drugim etapie renderowania korzystamy z wyników wygenerowanych w pierwszym etapie. Ponadto ze względu na różny rozmiar bufora ramki, do którego rysujemy w obu etapach, konieczne było stworzenie dwóch obiektów potoku — po jednym na każdy z etapów. Z tego względu nie mogłem również wykorzystać mechanizmu *Multipass* wprowadzonego w nowszym API.

5.2.5. Test 4 — czas inicjalizacji dema

Motywacją do stworzenia tego testu była chęć zbadania jak dużo czasu zajmuje stworzenie i inicjalizacja obiektów potrzebnych do pełnoprawnego korzystania z API.

Choć w teorii zadanie jest proste, w praktyce okazuje się znacznie trudniejsze. Dzieje się tak ze względu na to, że spora część pracy wykonywana jest asynchronicznie, zatem do końca nie jesteśmy w stanie stwierdzić, czy dany obiekt jest już w pełni zainicjalizowany, czy sterownik odłożył sobie wykonanie tego zadania na później. Ponadto również inne elementy — takie jak komunikacja z silnikiem prezentującym bufory ramek na ekranie, czy brak skupienia się nad tym elementem optymalizacji sterowników przez ich twórców, ze względu na niski priorytet zadania — mogą zaburzać otrzymane wyniki.

Patrząc na wszystkie wspomniane wyżej powody, należy zaznaczyć, że wyniki otrzymane w tym teście mogą nie być miarodajne, ani nie pokrywać się z prawdziwymi wynikami na różnym sprzęcie. Pamiętać również trzeba, że inicjalizacja większości obiektów zazwyczaj dzieje się tylko raz, na początku działania aplikacji. Z tego powodu ewentualne zyski czy straty, o ile małe, nie powinny wpływać na nasze postrzeganie danego interfejsu.

By zminimalizować zaburzenia w otrzymanych wynikach test — poza inicjalizacją obiektów — rysuje również pierwszą klatkę oraz wysyła żądanie aktualizacji bufora ramki lub prezentacji aktualnej klatki. Klatka ta składa się z jednego, kolorowego trójkąta. Po tym wszystkim wykonywana jest metoda synchronizująca pracę wykonywaną na procesorze graficznym z pracą wykonywaną na CPU, by upewnić się, że wszystkie wysłane zadania zostały już wykonane. Warto również nadmienić, że czas wymagany na utworzenie okna wraz ze wszystkimi ustawieniami (kontekstem OpenGLa lub możliwościami prezentacji ramek Vulkana) również został włączony do pomiarów, ze względu na możliwość inicjalizacji pewnych obiektów API podczas tego zadania.

5.3. Metodologia testowania

Testy zostały przeprowadzone na laptopie Lenovo Y580 o następujących podzespołach:

- CPU: Intel i5 3210M
- GPU: Nvidia GTX 660M (sterowniki w wersji 382.33)
- RAM: 6GB DDR3

Testy zostały przeprowadzone na systemie operacyjnym `Windows 7 SP1 64-bit`. Aplikacja testowa uruchomiona była w trybie pomiarów — z przełącznikiem `-benchmark`.

5.4. Pomiary czasów

Aplikacja w trybie testującym mierzy średnią długość klatki jako czas pomiędzy początkiem renderowania pierwszej klatki i zakończeniem renderowania ostatniej klatki podzielony przez ilość wygenerowanych klatek. Taki sposób pomiarów pozwala na zmierzenie pełnego czasu klatki, zarówno po stronie CPU jak i GPU, oraz zniwelowanie różnych czynników zewnętrznych zaburzających wyniki, jak na przykład specyficzne optymalizacje odkładające wykonywanie danego zadania na osobny wątek.

Początkowo wartości te były porównywane również z wynikami generowanymi przez wbudowaną w Vulkana warstwę `VK_LAYER_LUNARG_monitor`, jednak z kilku względów wykorzystana została własna implementacja:

- Wyniki uzyskane we własnej implementacji pokrywają się z wynikami wygenerowanymi przez warstwę.
- Brak odpowiadającej wersji dla API OpenGL, zatem osobna implementacja również musiałaby być stworzona.
- Brak łatwego dostępu do wygenerowanych przez warstwę wyników. Zapisywane są one automatycznie w belce tytułowej uruchomionej aplikacji. Brak możliwości łatwego wykorzystania tych danych przez automatyczne skrypty.[1].

Korzysta ona z funkcji `glfwGetTime()` dostępnej w bibliotece `GLFW`, która zwraca możliwie dokładną na danej platformie wartość aktualnego czasu. Pozwala to uzyskać stabilne i dokładne wyniki niezależnie od wykorzystywanej platformy testowej i kompilatora.

Poza wartością średnią, podawane są również minimalne i maksymalne czasy generowania klatki na CPU, mierzone jako czas pomiędzy kolejnymi podmianami bufora klatki, w przypadku API OpenGL, oraz pomiędzy kolejnymi zapytaniami prezentacji aktualnej klatki, w przypadku Vulkana. Wartości uzyskane w ten sposób nie są do końca miarodajne, gdyż implementacja danego API jest w stanie odłożyć pracę wykonywaną z daną czynnością na później lub na inny wątek. Z tego względu otrzymane tak wyniki nie zostały podane w pracy.

Wszystkie podawane dalej wyniki są uśrednionymi czasami klatek spośród wszystkich pomiarów w zadanym przedziale czasowym wynoszącym 15 sekund. Ponadto wyniki z pierwszej sekundy pomiarów są pomijane, by zniwelować możliwe wahania spowodowane inicjalizacją czy wykonywaniem pewnych operacji po raz pierwszy.

[1] Mechanizm ten został użyty przez redakcję serwisu `Phoronix`, który wykorzystuje projekt `GL_vs_VK` jako jeden z testów dostępnych w ich narzędziach testujących [50].

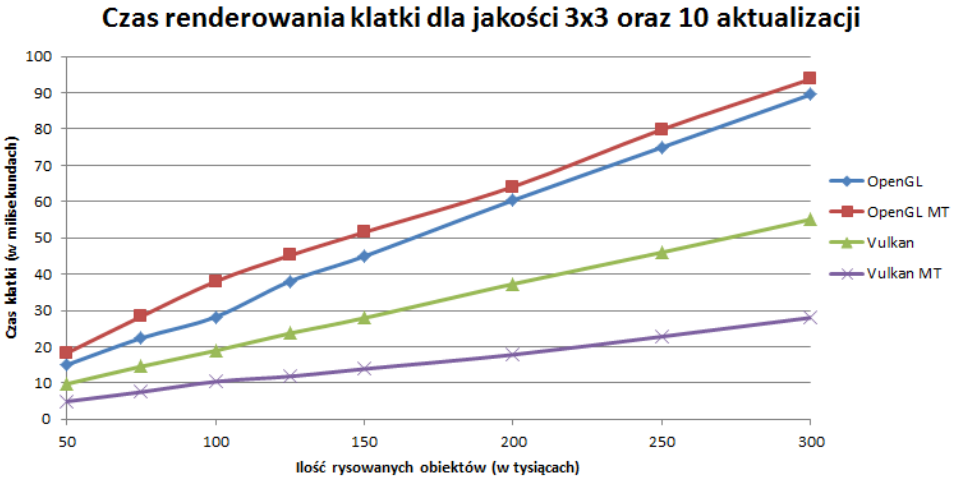
5.5. Wyniki

5.5.1. Wyniki testu 1

Test ten zawierał trzy wartości konfiguracji pozwalające na ustawienie:

- Ilości rysowanych obiektów,
- Jakości rysowanych obiektów,
- Ilości pracy wykonywanej na CPU dla każdego obiektu w każdej klatce.

Dzięki temu możliwe było zasymulowanie różnego różnych proporcji obciążenia CPU oraz GPU w testach. Z tego względu wykonałem kilka pomiarów, by przetestować każdy wariant. Jakość $A \times B$ odpowiada wygenerowaniu sfery składającej się z $2AB$ trójkątów i rozumiana jest przez podział siatki na A części w pionie i B w poziomie.

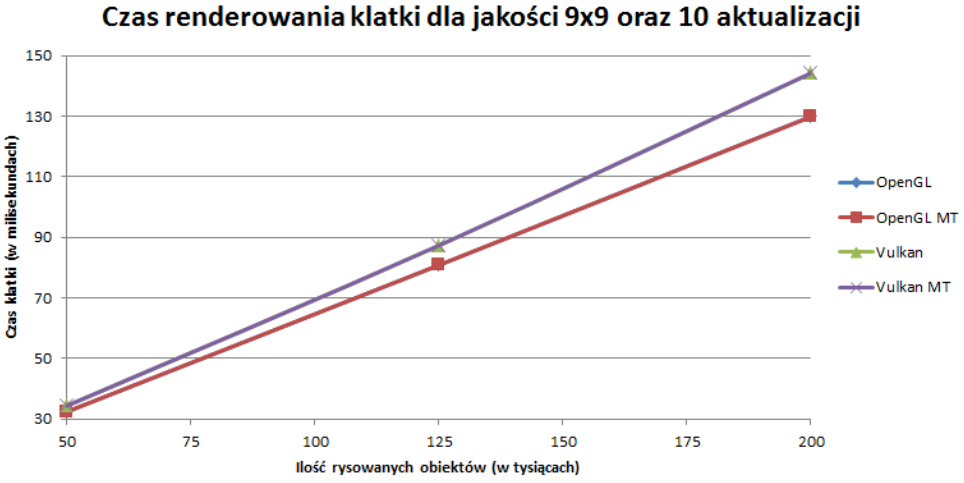


Rysunek 5.3: Wyniki pomiarów testu 1 dla jakości 3x3 oraz 10 aktualizacji.

Przedstawione na rysunku 5.3 wyniki odnoszą się do testów przeprowadzonych na konfiguracji wykorzystującej niższą jakość obiektów oraz mniejszą ilość pracy wykonywaną na CPU dla każdego obiektu. Na wynikach tych zaobserwować możemy znaczną przewagę wydajnościową API Vulkan, która powiększą się wraz z wykorzystaniem wielu wątków. Wersja oparta o OpenGL jest około dwukrotnie wolniejsza od zwykłej wersji korzystającej z Vulkana oraz około trzykrotnie wolniejsza od implementacji wielowątkowej. Co ciekawe, wielowątkowa wersja korzystająca z OpenGL jest nieznacznie wolniejsza, od wersji jednowątkowej. Sugeruje to, że w tej konfiguracji test ten jest ograniczony przez CPU i mały zysk zaoszczędzony na wielowątkowej aktualizacji obiektów nie daje zysków przy narzucie na wykorzystanie wielu wątków.

Na kolejnym rysunku (5.4) widzimy wyniki dla konfiguracji, która symulować ma duże obciążenie na procesorze graficznym oraz stosunkowo małe obciążenie na

procesorze głównym. Na tym wykresie widzimy, że wydajność wersji korzystającej z nowszego API jest minimalnie mniejsza, niż tej, opartej na sprawdzonym i przez wiele lat optymalizowanym API. Różnice wynoszą około 10 procent. Co ciekawe — w obu przypadkach wersje wielowątkowe posiadają prawie identyczną wydajność, co pozwala sugerować, że w tej konfiguracji, test ograniczony jest w całości przez moc GPU. Wyniki prawdopodobnie wynikają z minimalnie lepszej optymalizacji sterowników.

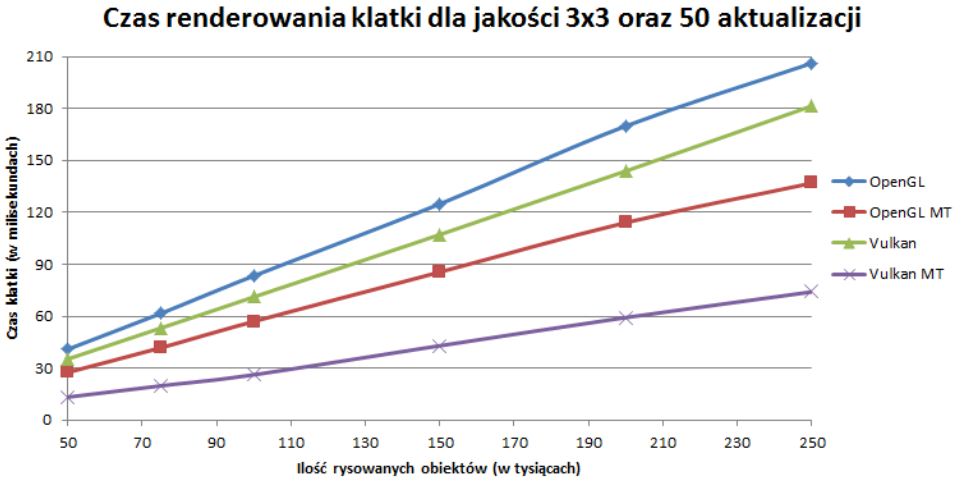


Rysunek 5.4: Wyniki pomiarów testu 1 dla jakości 9x9 oraz 10 aktualizacji. Zauważyć możemy jednakową wydajność dla wersji jednowątkowych i wielowątkowych.

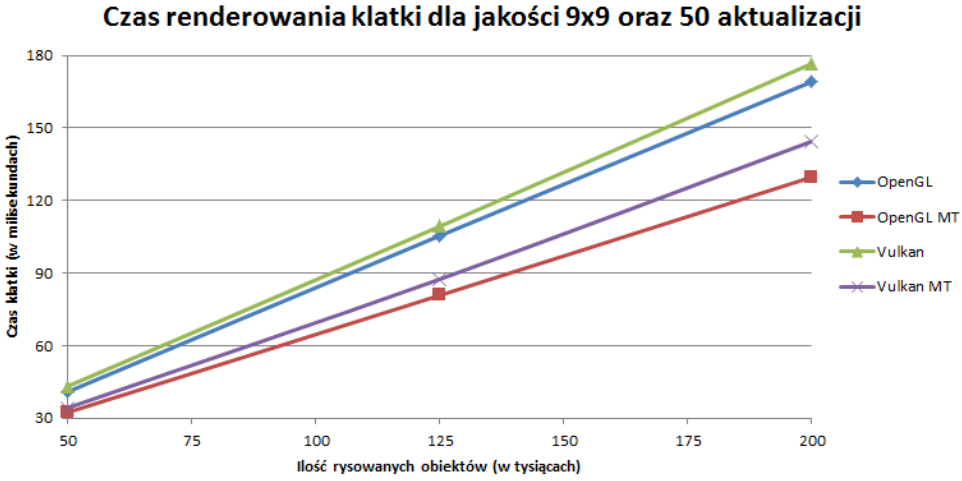
Kolejne dwa testy przeprowadzone były w konfiguracji o znacznie zwiększonej ilości pracy wykonywanej na CPU. Na pierwszym z wykresów (rys. 5.5) widzimy pomiary dla konfiguracji o niskiej jakości obiektów, zatem test powinien być zdominowany przez pracę wykonywaną na CPU. Przyjmując jako bazową implementację opartą o API Vulkan, z pomiarów wynika, że wersja oparta o starsze API jest wolniejsza o około 15 procent. Zgodnie z przewidywaniami widzimy tu, że wersje wielowątkowe są zauważalnie szybsze, niż wersje ograniczone do jednego wątku i obie są szybsze od bazowej wersji. Potrzebują one tylko 75 oraz 40 procent czasu, kolejno dla wersji OpenGL i Vulkan, by wyrenderować tę samą klatkę. Widzimy tu, że niezależnie od użytego API, jeśli aplikacja wykonuje duże obliczenia to warto je zrównoleglić jeśli tylko się da. Ponadto widzimy, że wielowątkowa implementacja oparta o Vulkan oferuje znacznie większe przyśpieszenie, niż podobna wersja dla API OpenGL, co wynika zapewne z faktu wielowątkowego korzystania z API.

Na ostatnim wykresie (rys. 5.6) widzimy wyniki dla konfiguracji, która symuluje dużo pracy wykonywanej na CPU oraz złożoną pracę wykonywaną GPU, wykonywaną ze stosunkowo małej ilości użytych poleceń. Przedstawia to zatem sytuację, w której Vulkan powinien zyskać najmniej spośród dotąd wykonanych testów.

Zgodnie z przewidywaniami, na wykresie zauważyć możemy, że bazowe wersje



Rysunek 5.5: Wyniki pomiarów testu 1 dla jakości 3x3 oraz 50 aktualizacji. Zauważyć możemy jednakową wydajność dla wersji jednowątkowych i wielowątkowych.



Rysunek 5.6: Wyniki pomiarów testu 1 dla jakości 9x9 oraz 50 aktualizacji.

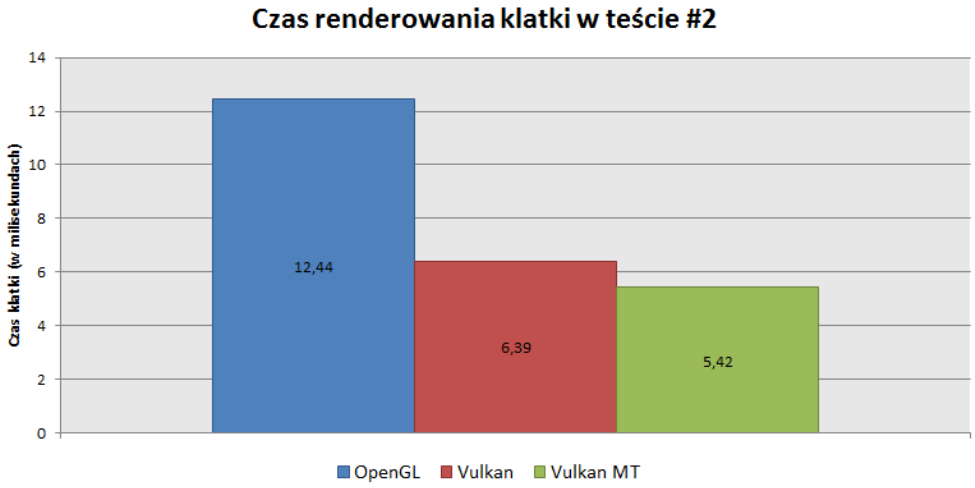
oparte na obu API zapewniają porównywalną wydajność, podobnie jak na wykresie 5.4, z małą różnicą na korzyść OpenGL. Co ciekawe, nawet w wersji wielowątkowej starsze API wygrywa. Pokazuje to nam, że w przypadku dużego obciążenia procesora graficznego oraz stosunkowo — względem pozostałej pracy — małej ilości wykonywanych poleceń rysowania na CPU, nowe API nie pozwala w dużym stopniu na zwiększenie wydajności aplikacji. Zastanawiające jest to, że różnice pomiędzy API rosną w przypadku wersji wielowątkowych. Intuicja bowiem podpowiada, że różnica wynikająca z nakładu pracy na GPU powinna być podobna w przypadku obu wersji.

Podsumowanie

Jak widzimy na otrzymanych wykresach, Vulkan potrafi znacząco przyśpieszyć renderowanie scen złożonych z dużej ilości obiektów oraz świetnie spisuje się w aplikacjach wielowątkowych. Zauważyć możemy również minimalnie gorszą wydajność w przypadku scen złożonych ze skomplikowanych obiektów. Może to świadczyć o słabszej optymalizacji dostępnych sterowników lub o braku pewnych optymalizacji w zaimplementowanym teście.

5.5.2. Wyniki testu 2

Na rysunku 5.7 przedstawiony jest wykres z wynikami dla testu drugiego. Test ten posiada tylko jedną konfigurację.

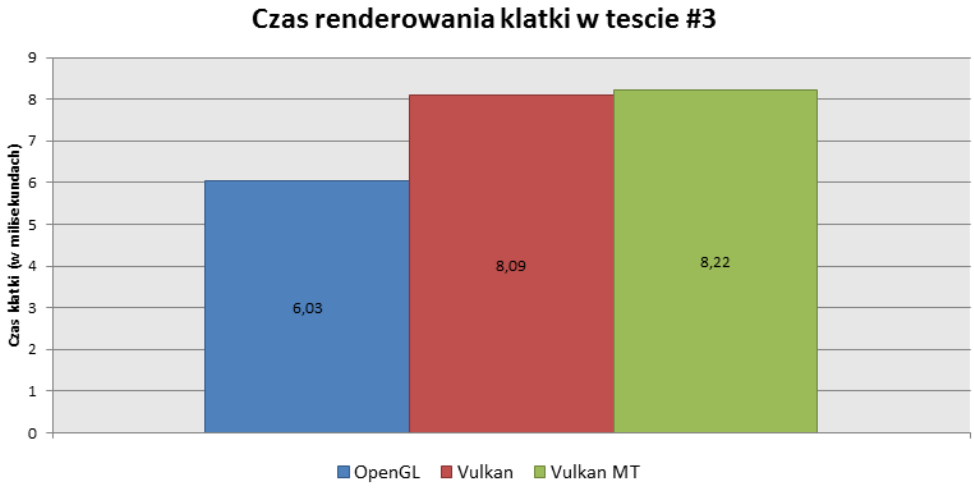


Rysunek 5.7: Wyniki pomiarów testu 2 w zależności od użytego API.

Jak wynika z powyższego wykresu, Vulkan w przypadku renderowania siatki terenu korzystając z mapy wysokościowej i algorytmu *Level-of-Detail* pozwala na znaczące przyśpieszenie aplikacji. Wersja bazowa oparta na nowszym API jest prawie dwukrotnie szybsza od wersji opartej o starszy interfejs. Wersja wielowątkowa oferuje małe — około 15 procentowe — przyśpieszenie względem wersji jednowątkowej. Wynika to zapewne ze skomplikowania algorytmu *LoD*, co przekłada się na nieoptymalny podział pracy na wszystkich wątkach. Niemniej jest to zauważalny zysk, który w przypadku korzystania z większych map wysokościowych zapewne byłby jeszcze większy.

5.5.3. Wyniki testu 3

Wyniki trzeciego z dostępnych testów przedstawione są rysunku 5.8. Z wykresu odczytać możemy, że wersja oparta o nowszy interfejs jest wolniejsza o około 30 procent od wersji korzystającej z OpenGL.



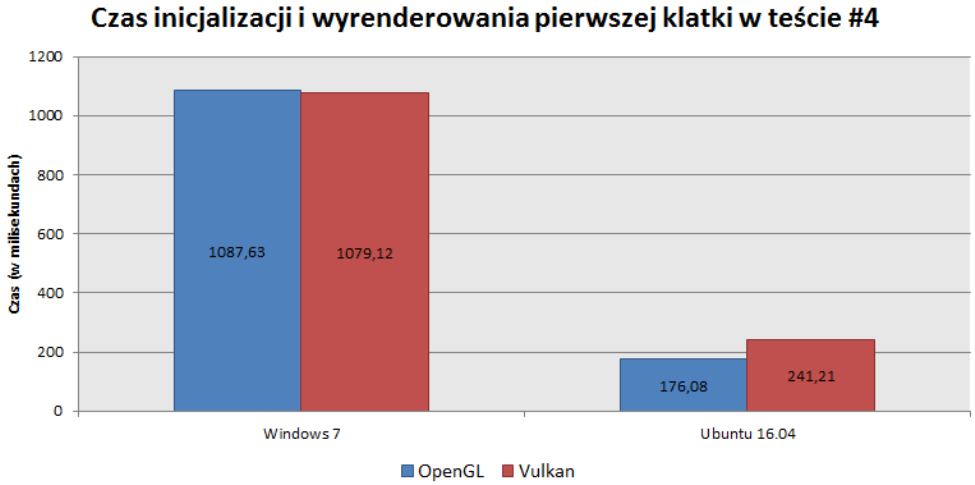
Rysunek 5.8: Wyniki pomiarów testu 3 w zależności od użytego API.

Zgodnie z intuicją, wersja wielowątkowa nie zyskuje, a wręcz minimalnie traci w tym teście, co spowodowane jest zapewne bardzo niskim wykorzystaniem CPU w tym teście. Tak duża różnica jest zastanawiająca, gdyż implementacja oparta o Vulkana była kilkakrotnie sprawdzana i nie zostały znalezione żadne potencjalne optymalizacje. Dodatkowo zauważyć można, że ten sam test na systemie Linux pokazuje porównywalną wydajność w przypadku API Vulkan, jednak wersja OpenGL jest tam drastycznie wolniejsza co pozwala zwyciężyć nowszemu API nawet w tym teście. Nie jest znane jak duży wpływ na takie wyniki mają sterowniki, jednak możliwe jest, że sterownik OpenGL wykonuje za plecami programisty bardziej agresywne optymalizacje. Warto również zauważyć, że algorytm mapowania cieni jest bardzo powszechnie stosowany, co być może zostało w jakiś sposób dodatkowo rozpoznawane i zoptymalizowane.

5.5.4. Wyniki testu 4

Na rysunku 5.9 przedstawiony jest wykres z wynikami czwartego testu. Test ten posiada tylko jedną konfigurację i nie jest dostępny w trybie wielowątkowym, jednak ze względu na pomiar czasowy elementów powiązanych z systemem operacyjnym i używanym systemem okien na danej platformie, testy zostały przeprowadzone również na systemie `Ubuntu 16.04 x64`.

Jak widzimy na powyższym rysunku, czas potrzebny na inicjalizację API, okna,



Rysunek 5.9: Wyniki pomiarów testu 4 w zależności od użytego API.

wszystkich używanych obiektów oraz wyrenderowanie pierwszej klatki zależy w sporej mierze od używanej platformy. Na systemie operacyjnym `Windows 7`, obie wersje potrzebowały około 1 sekundy na załadowanie i wygenerowanie pierwszej klatki, a różnice są marginalne. Na wynikach z systemu `Ubuntu` widzimy, że obie wersje potrzebują znacznie mniej czasu do wykonania testu, a także zauważyć możemy, ze wersja oparta o API OpenGL jest około 27 procent szybsza w tym zadaniu.

Wyniki te ciężko zinterpretować z wielu powodów:

- O ile wersje oparte o ten sam system operacyjny i system okienkowy powinny być miarodajne, o tyle porównywanie różnych platform nie ma większego sensu, gdyż różne systemy okienkowe mogą generować bardzo różniący się narzut. Taką sytuację widzimy na powyższym wykresie.
- Nie wiadomo do końca co robi sterownik OpenGL. Ciężko powiedzieć, czy bufor wierzchołków, podobnie jak w przypadku API Vulkan przenoszony jest do pamięci lokalnej urządzenia, by w następnych klatach korzystanie z niego było jak najszybsze. Nie wiemy również jaka praca została wykonana z utworzeniem buforów ramek, gdzie wersja korzystająca z Vulkana tworzy przynajmniej trzy zestawy buforów ramek z własnymi buforami komend.
- Nie wiadomo, czy sterownik OpenGL nie posiada w pamięci cache bazowej wersji domyślnie skonfigurowanego potoku graficznego, co znacząco przyśpieszałoby resetowanie oraz ładowanie nowych ustawień.
- Duży wpływ na uzyskane tu wyniki może mieć użyty sterownik graficzny. Możliwe, że jest to element który dopiero zostanie zoptymalizowany w przyszłości, gdyż programiści chcieli skupić się na uzyskaniu maksymalnej wydajności podczas pracy, a nie na etapie inicjalizacji.

- O ile wersja korzystająca z API Vulkan napisana jest tak, by w następnych klatkach wykonać jak najmniej pracy, a wykonana praca była jak najszybsza, o tyle sterownik starszego interfejsu może zoptymalizować pracę tej klatki kosztem pozostałych klatek i wykonać mniej pracy tutaj.

Jedynym, bezsprzecznym argumentem, który powinniśmy wyciągnąć z powyższych wyników jest to, że inicjalizacja niektórych obiektów, takich jak potoki graficzne czy programy cieniujące, są operacjami drogimi ze względu na czas i wykonywanie ich powinno być — jeśli to tylko możliwe — przeniesione do fazy ładowania aplikacji.

5.6. Wyniki redakcji serwisu Phoronix

Projekt `GL_vs_VK` tworzony na potrzeby tej pracy został zauważony przez redakcje serwisu **Phoronix**. **Michael Larabel**, założyciel serwisu, pokusił się o przetestowanie projektu na dostępnym sprzęcie i kilku sterownikach graficznych:

- Nvidia GeForce GTX 780 Ti (sterownik Nvidia 381.22),
- Nvidia GeForce GTX 1050 (sterownik Nvidia 381.22),
- Nvidia GeForce GTX 1060 (sterownik Nvidia 381.22),
- Nvidia GeForce GTX 1080 (sterownik Nvidia 381.22),
- AMD Radeon R9 Fury (sterowniki AMDGPU-PRO 17.10 oraz RADV Mesa 17.2-dev),
- AMD Radeon RX 580 (sterowniki AMDGPU-PRO 17.10 oraz RADV Mesa 17.2-dev),
- Intel HD Graphics 630 (sterownik Mesa 17.2-dev).

Dokładniejsze informacje, wraz z pełnymi wynikami można znaleźć na stronie testów [50].

Otrzymane wyniki są bardzo kompleksowe i oferują możliwość porównania wydajności na wielu kartach graficznych i różnych sterownikach. W większości testów Vulkan oferuje zysk wydajnościowy rzędu 50 do 200 procent, w zależności od testu i użytych sterowników. Warto jednak zauważyć, że testy zostały napisane w taki sposób, aby przedstawić pełny zakres możliwych wyników, czego przykładem mogą być ręcznie dobrane wartości konfiguracyjne pierwszego z dostępnych testów. Dzięki temu możliwe było uzyskanie na moim sprzęcie różnych wariantów, takich jak ograniczenie wydajności przez moc procesora lub karty graficznej. Testy przeprowadzone przez redakcję serwisu wykonane zostały na mocnym procesorze graficznym, przez

co większość testów prawdopodobnie była ograniczona mocą CPU, co przekłada się na — w pewnym stopniu — faworyzowanie Vulkana i jego zalet. Warto jednak pamiętać, że zaobserwowane różnice są jak najbardziej realne i możliwe zyski wydajnościowe badane w tej pracy zaobserwować można na różnym sprzęcie i różnych sterownikach.

Rozdział 6.

Wnioski

Zgodnie z zapowiedzią, w rozdziale tym postaram się odpowiedzieć na dwa zadane wcześniej pytania:

- Czy korzystanie z nowszego interfejsu pozwala zapewnić większą wydajność?
- Jak wyglądają różnice pomiędzy nakładem pracy potrzebnym na stworzenie aplikacji w tych API?

W sformułowaniu odpowiedzi na te pytania, posłużę się wiedzą uzyskaną podczas pisania tej pracy oraz wynikami — zarówno dostępnymi w sieci jak i uzyskanymi w projekcie stworzonym na potrzeby tej pracy.

6.1. Wydajność

Zgodnie z przypuszczeniami wysnutymi na początku pracy oraz zapewnieniami twórców nowego API, Vulkan, w pewnych sytuacjach, pozwala na uzyskanie zauważalnego wzrostu wydajności aplikacji. Wniosek taki jest całkowicie bezsprzeczny i potwierdzony może być wieloma testami dostępnymi w sieci oraz potwierdzają go również aplikacje stworzone na cele tej pracy. Zyski płynące z korzystania tego API są tym większe, im większy jest stosunek pracy wykonywanej na potrzeby przetwarzania zapytań API. Zyskamy zatem najwięcej w przypadku kiedy rysujemy mało skomplikowaną geometrię lub musimy często zmieniać stan.

Zyski, jakie możemy osiągnąć możemy podzielić na dwie kategorie:

- Zwiększenie ilości renderowanych klatek na sekundę,
- Zmniejszenie obciążenia CPU wykonując tą samą pracę.

Dzięki temu zyskać mogą zarówno aplikacje, które wykonują bardzo duże ilości poleceń rysowania czy zmiany stanu oraz te aplikacje, które wykonują również —

pomiędzy zapytaniami do API graficznego — dużo pracy na CPU. Mniejsze obciążenie CPU pozwala bowiem wykonać więcej pracy w tym czasie, co powinno przełożyć się na większą wydajność. Część zysków możemy przypisać również wprowadzonym w nowszym API mechanizmom takim jak *PushConstants*, które pozwalają na aktualizację małych fragmentów pamięci na GPU bez potrzeby uruchamiania całego mechanizmu transferu danych i synchronizacji dostępu do niej.

Z przeprowadzonych i dostępnych testów wynika jednak, że zyski są małe lub praktycznie niewidoczne, gdy aplikacja w całości ograniczona jest przez GPU. Nie pomoże ono zatem aplikacjom, których lwia część pracy wykonywana jest na procesorze graficznym.

Ciekawie prezentuje się wyniki czasu inicjalizacji dema. Znacznie gorsze rezultaty dla nowszego API mogą wynikać z konieczności ręcznej inicjalizacji całego potoku i wszystkich wykorzystywanych elementów interfejsu przez programistę, gdzie w starszym API OpenGL, wiele obiektów była używana w domyślnej postaci, co zapewne pozwala na pewne optymalizacje. Wpływ na wyniki może mieć również chęć bardziej agresywnych optymalizacji po stronie Vulkana, który posiadając znacznie szerszą wiedzę na temat zamiarów programisty, może próbować lepiej zoptymalizować posiadane zasoby podczas ich tworzenia, by przyśpieszyć późniejszą pracę. Tezę tą wspiera fakt, ze w API dostępnych jest kilka flag pozwalających na wyłączenie optymalizacji pewnych obiektów w przypadku jednokrotnego ich użycia.

6.2. Nakłady pracy

Vulkan jest zdecydowanie bardziej niskopoziomowym interfejsem niż OpenGL. Przekłada się to na zauważalnie większy potrzebny nakład pracy, by stworzyć podobne aplikacje z jego wykorzystaniem. Najcięższymi elementami do opanowania wydają się ręczna alokacja i zarządzanie pamięcią oraz konieczność poprawnej synchronizacji wykonywanej pracy. Zadania te nie są proste, a błędne ich wykonanie przełożyć może się na niepoprawne zachowanie aplikacji na niektórych urządzeniach czy gorszą wydajność.

Kolejną istotną kwestią w analizie nakładu pracy jest to, że choć API jest bardzo *rozwlekłe* i wymaga przedstawienia dość dokładnego opisu jego wykorzystania, zadanie to jest stosunkowo proste, ze względu na przemyślane zaprojektowanie interfejsu. Pisząc aplikacje korzystające z niego, mając podstawową wiedzę na temat najważniejszych obiektów w API łatwo zrozumieć czym są wymagane parametry i co opisują dane struktury.

Zauważyć ponadto musimy, że dostępne warstwy walidacji w dużym stopniu ułatwiają korzystanie z tego API oraz znacznie przyśpieszają pracę z tym API. W przypadku interfejsu OpenGL często występowały sytuacje, w których na ekranie nie widzieliśmy wyników zleconych operacji, jednak nie dostawaliśmy żadnych błędów

lub wskazówek, gdzie mogliśmy popełnić błąd. W tym temacie wspomniane warstwy spisują się znacznie lepiej — i choć ciągle są rozwijane — ich pokrycie interfejsu jest znaczne i wykrywa większość najczęstszych problemów. Fakt, że są one udostępnione jako oprogramowanie open-source, które może rozwijać każdy, pozwala wierzyć, że warstwy te będą rozszerzały się o coraz to nowe możliwości usprawniające pracę z tym API.

Istotnym elementem z punktu widzenia programisty jest również dostępność narzędzi i możliwość utworzenia nowych. Na przykładzie Vulkana widzimy, że twórcy API sporo nauczyli się od czasu wydania OpenGL i mają znacznie lepsze podejście w tym temacie. Przykładem może być udostępnienie wszystkich kluczowych elementów API — w tym samej specyfikacji API, dostępnych narzędzi, referencyjnych kompilatorów czy wspomnianych warstw walidacji — na zasadzie otwartej licencji, przez co programiści korzystający z tych rzeczy mają większy wpływ na kształtowanie się ekosystemu wokół omawianego interfejsu.

6.3. Podsumowanie

Vulkan to stosunkowo młode API. Stworzone zostało w konkretnych celach i cele te realizuje znacznie lepiej, niż jego poprzednik. Do celów tych nigdy nie należało zastąpienie jego poprzednika i patrząc na stopień jego skomplikowania szybko to prawdopodobnie nie nastąpi. Pozwala jednak na uzyskanie znacznie wyższej wydajności, zatem aplikacje czy silniki graficzne, które zmuszone są do jak najlepszego optymalizowania swojego kodu prawdopodobnie zyskają zauważalny wzrost wydajności po przejściu na nowsze API.

Choć API jest znacznie trudniejsze, korzystanie z niego — poza kilkoma aspektami jak synchronizacja — wcale nie wydaje się takie trudne. Wszystko to dzięki dobremu zaprojektowaniu interfejsu, który jest czytelny i łatwy w zrozumieniu. Sytuacja ta prawdopodobnie poprawi się wraz z powiększeniem bibliotek i projektów ułatwiających korzystanie z tego API.

Cieszy również fakt zdrowego podejścia do utrzymywania API przez twórców, wydania narzędzi oraz wspierania różnych systemów operacyjnych. Pozwala to wierzyć, że API będzie się stabilnie rozwijało, a zbiór dostępnych narzędzi będzie się stale powiększał, z których każde będzie ciągle udoskonalane i rozszerzane o nowe możliwości.

W mojej opinii Vulkan to przyszłość aplikacji nastawionych na maksymalną wydajność, a z czasem również i pozostałych aplikacji multimedialnych. Wraz z nadejściem bibliotek ułatwiających wykorzystywanie tego API oraz poprawą dostępnych sterowników, znikną ostatnie argumenty przeciwko wyborze tego interfejsu. Potwierdza to również stopień zaangażowania w prace nad API oraz rozwój swojego oprogramowania firm z branży.

Dodatek A

Projekt GL_vs_VK

A.1. Kod źródłowy

Projekt udostępniony jest na warunkach licencji MIT w publicznym repozytorium hostowanym w serwisie GitHub. Repozytorium to znajduje się pod adresem `https://github.com/RippeR37/GL_vs_VK`.

Kod napisany jest w języku `C++` i do jego kompilacji wymaga kompilatora zgodnego ze standardem `C++11`. Do kodu źródłowego dołączony jest plik projektu dla środowiska **Microsoft Visual Studio 2013** oraz skrypt budujący dla programu **CMake**. Projekt wspiera systemy z rodzin Microsoft Windows oraz Linux przy czym testowany był na następujących systemach operacyjnych:

- Microsoft Windows 7,
- Ubuntu 16.04,
- Ubuntu 17.04.

Kod do swojego działania wykorzystuje następujące biblioteki:

- `GLEW` — załadowanie API OpenGL,
- `GLFW` — stworzenie okna oraz obsługi wejścia/wyjścia,
- `GLM` — obliczenia matematyczne związane z macierzami i wektorami.

Projekt — jako submoduły systemu kontroli wersji GIT — zawiera w sobie powyższe biblioteki oraz dodatkowo udostępnia konkretną wersję nagłówków API Vulkan, oraz odniesienie do odpowiedniej wersji biblioteki `Vulkan-Hpp` tak, by być niezależnym od wersji zainstalowanej przez użytkownika.

Dodatkowo kod do kompilacji i działania wymaga zainstalowanych sterowników OpenGL oraz Vulkan. W przypadku chęci zmiany i rekompilacji shaderów wymagane jest zainstalowanie aplikacji `glslangValidator` dostępnej między innymi w oprogramowaniu `Vulkan SDK`.

Kod podzielony jest na trzy moduły:

- `base` — moduł odpowiedzialny za bazową obsługę podstawowych i często wykorzystywanych obiektów. Wewnątrz tego modułu znajdują się dwa submoduły — `gl` oraz `vkx` które skupiają się na udostępnieniu wysokopoziomowego dostępu do podstawowych elementów API OpenGL oraz Vulkan, między innymi pozwalając na stworzenie okna obsługującego dane API.
- `framework` — moduł odpowiedzialny za funkcjonalność frameworku uruchamiającego testy i mierzącego ich wydajność.
- `tests` — moduł zawierający wszystkie zaimplementowane testy w osobnych katalogach. W każdym katalogu testu znajdują się pod-katalogi dla każdego z testowanych API zawierające implementację danego testu wykorzystującą dane API.

A.2. Kompilacja

Projekt jest w całości niezależny i kompiluje większość potrzebnych bibliotek wraz ze sobą. Dzięki temu nie jest wymagana żadna ingerencja w system użytkownika. Dodatkowo, jeśli skrypt CMake wykryje obecność w systemie biblioteki GLEW, to zamiast budować ją od podstaw — użyje dostępnej wersji. Dokładniejsze informacje na temat budowy znajdują się w pliku `README.md`.

A.2.1. Kompilacja na platformie Windows

Dla systemów z rodziny Windows wspierany jest system budowy wykorzystujący środowisko Microsoft Visual Studio w wersjach 2013 lub nowszych. Proces budowy sprowadza się tutaj do:

1. Inicjalizacji submodułów Gita,
2. Wypakowaniu zawartości pliku `glew-win-src.zip` z folderu `GL_vs_VK/third_party/glew-win/` do folderu go zawierającego,
3. Uruchomieniu pliku projektu znajdującego się w katalogu `GL_vs_VK/project/msvc/`,
4. Zbudowaniu projektu przy użyciu środowiska Visual Studio.

Budowa projektu z użyciem innego kompilatora oraz skryptu CMake powinna być możliwa, jednak w tym wypadku należy własnoręcznie skompilować wszystkie zależności oraz ręcznie podać do nich ścieżki. W tym wypadku polecam skorzystać z oprogramowania **CMake GUI**, które powinno ułatwić ten proces.

A.2.2. Kompilacja na platformie Linux

Na systemach operacyjnych z rodziny Linux kompilacja odbywa się z wykorzystaniem dołączonego skryptu budującego wykorzystującego oprogramowanie CMake. Budowanie na tej platformie polega na:

1. Inicjalizacji submodułów Gita,
2. Wygenerowaniu projektu programem CMake,
3. Zbudowanie wygenerowanego projektu przy użyciu odpowiedniej komendy.

Przykładowymi poleceniami budującymi projekt mogą być:

```
cd GL_vs_VK
git submodule update --init
mkdir build && cd build
cmake .. && make
```

A.3. Uruchomienie

Po zbudowaniu projektu, w katalogu `bin` znajdzie się plik wykonywalny o nazwie `GL_vs_VK`. Aby uruchomić któryś z testów, należy uruchomić ten program z odpowiednimi argumentami.

Dostępne argumenty:

- `-t [N]` — precyzuje który test ma zostać uruchomiony,
- `-api [API]` — wybiera które API ma zostać użyte (dostępne: `gl` i `vk`),
- `-m` — opcjonalny przełącznik włączający wersję wielowątkową (gdy dostępna)
- `-benchmark` — włącza tryb pomiaru, który sprawia, że dany test włączony będzie tylko przez określony czas i po jego zakończeniu wyświetlone zostaną na standardowym wyjściu statystyki z jego działania.
- `-time [T]` — zmienia domyślny czas wykonywania testu na podaną wartość. Argument ten jest ignorowany, jeśli nie podano przełącznika `-benchmark`.

Przykładowe, chcąc uruchomić trzeci test w wersji wielowątkowej wykorzystującej API Vulkan w trybie przeprowadzenia pomiarów z domyślnym czasem testów, należy wykonać polecenie:

```
./GL_vs_VK -t 3 -api vk -m -benchmark
```

Bibliografia

[1] Wikipedia, *Application programming interface*, https://en.wikipedia.org/wiki/Application_programming_interface.

[2] Wikipedia, *Vulkan*, https://en.wikipedia.org/wiki/Vulkan_(API).

[3] Wikipedia, *OpenGL*, https://en.wikipedia.org/wiki/OpenGL.

[4] Wikipedia, *Mantle*, https://en.wikipedia.org/wiki/Mantle_(API).

[5] Khronos Group, *History of OpenGL*, https://www.khronos.org/opengl/wiki/History_of_OpenGL.

[6] Khronos Group, *OpenGL 4.5 core profile specification*, https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf/

[7] Wikipedia, *Fixed-function*, https://en.wikipedia.org/wiki/Fixed-function.

[8] Khronos Group, *Fixed Function Pipeline*, https://www.khronos.org/opengl/wiki/Fixed_Function_Pipeline.

[9] Khronos Group, *Rendering Pipeline Overview*, https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview.

[10] Wikipedia, *OpenGL Shading Language*, https://en.wikipedia.org/wiki/OpenGL_Shading_Language.

[11] Khronos Group, *OpenGL Shading Language*, https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language.

[12] Cass Everitt, *OpenGL Efficiency: AZDO*, Konferencja GDC, San Francisco, 2014, https://www.khronos.org/assets/uploads/developers/library/2014-gdc/Khronos-OpenGL-Efficiency-GDC-Mar14.pdf.

[13] Wikipedia, *Graphics Core Next*, https://en.wikipedia.org/wiki/Graphics_Core_Next.

[14] Joel Hruska, *Asynchronous compute, AMD, Nvidia, and DX12: What we know so far?*, https://www.extremetech.com/extreme/213519-asynchronous-shading-amd-nvidia-and-dx12.

[15] Khalid Moammer, *AMD Improves DirectX 12 Performance By Up To 46% With Asynchronous Compute Engines*, http://wccftech.com/amd-improves-dx12-performance-45-gpu-asynchronous-compute-engines/

[16] Sellers Graham, Kessenich John M., Kessenich John, *Vulkan Programming Guide: The Official Guide to Learning Vulkan*, Addison Wesley, 2016.

[17] Khronos Group, *Vulkan 1.0 — A Specification*, https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html.

[18] NVIDIA, *Vulkan Training Day*, https://developer.nvidia.com/nvidia-vulkan-developer-day.

[19] Christoph Kubisch, *Transitioning from OpenGL to Vulkan*, https://developer.nvidia.com/transitioning-opengl-vulkan.

[20] Christoph Kubisch, *Engaging the Voyage to Vulkan*, https://developer.nvidia.com/engaging-voyage-vulkan.

[21] Christoph Kubisch, *OpenGL like Vulkan*, https://developer.nvidia.com/opengl-vulkan.

[22] Chris Hebert, *Vulkan Memory Management*, https://developer.nvidia.com/vulkan-memory-management.

[23] Matthaeus Chajdas, *Vulkan barriers explained*, http://gpuopen.com/vulkan-barriers-explained/.

[24] Graham Sellers, *Vulkan Renderpasses*, http://gpuopen.com/vulkan-renderpasses/.

[25] Alexander Overvoorde, *Vulkan Tutorial*, Maj 2017, https://vulkan-tutorial.com/Overview.

[26] Dominik Witczak, Daniel Rakos, Derrick Owens, *The most common Vulkan mistakes*, Prezentacja na Uniwersytecie Łódzkim, 2016, http://32ipi028l5q82yhj72224m8j.wpengine.netdna-cdn.com/wp-content/uploads/2016/05/Most-common-mistakes-in-Vulkan-apps.pdf.

[27] Soowan Park, *Vulkan Game Development in Mobile*, Konferencja GDC, 2017, https://www.khronos.org/assets/uploads/developers/library/2017-gdc/GDC_Vulkan-on-Mobile_Vulkan-Game-Development-in-Mobile-Samsung_Mar17.pdf.

[28] Timothy Lottes, Graham Sellers, Dr. Matthäus G. Chajdas, *Vulkan fast paths*, Konferencja GDC, 2016, http://32ipi028l5q82yhj72224m8j.wpengine.netdna-cdn.com/wp-content/uploads/2016/03/VulkanFastPaths.pdf.

[29] Dan Ginsburg, Valve, *Porting Source2 to Vulkan*, Konferencja SIGGRAPH, 2015, http://nextgenapis.realtimerendering.com/presentations/6_Ginsburg_Source2.pdf.

[30] Cort Stratton, *OpenGL/Vulkan Performance Test*, Konferencja Khronos DevU Seoul, Seul 2016, http://www.khronos.org/assets/uploads/developers/library/2016-vulkan-devu-seoul/7-Vulkan-GL-Performance-Comparison.pdf.

[31] Ashley Smith, *Gnomes per second in Vulkan and OpenGL ES*, https://www.imgtec.com/blog/gnomes-per-second-in-vulkan-and-opengl-es/.

[32] Robin Britton, *Vulkan vs OpenGL ES for a 3D satellite navigation app on PowerVR*, https://www.imgtec.com/blog/vulkan-3d-satnav-app-powervr/.

[33] Mark Tyson, *Khronos Vulkan API runs demo nearly twice as fast as OpenGL*, http://hexus.net/tech/news/software/85985-khronos-vulkan-api-runs-demo-nearly-twice-fast-opengl/.

[34] ARM, *Vulkan vs OpenGL ES on ARM mobile*, https://developer.arm.com/graphics/vulkan/vulkan-demos/vulkan-vs-opengl-es-on-arm-mobile.

[35] ARM, *First comparison of Vulkan API vs OpenGL ES API on ARM*, https://www.youtube.com/watch?v=rvCD9FaTKCA.

[36] Jarred Walton, *Doom benchmarks return: Vulkan vs. OpenGL*, http://www.pcgamer.com/doom-benchmarks-return-vulkan-vs-opengl/2/.

[37] John Williamson, *Doom OpenGL VS Vulkan Graphics Performance Analysis*, https://www.eteknix.com/doom-opengl-vs-vulkan-graphics-performance-analysis/.

[38] Mikko Strandborg, *Introducing the Vulkan renderer preview*, https://blogs.unity3d.com/2016/09/29/introducing-the-vulkan-renderer-preview/.

[39] Xenko, *Xenko 1.8 - Mutithreading: OpenGL vs Vulkan*, https://www.youtube.com/watch?v=sJ2p982cZFc.

[40] Dana Cowley, *Epic Games Unveils ProtoStar at Samsung Galaxy Unpacked*, https://www.unrealengine.com/en-US/blog/epic-games-unveils-protostar-at-samsung-galaxy-unpacked.

[41] Crytec, *CryENGINE roadmap*, https://www.cryengine.com/roadmap.

[42] Dean Sekulić, *Getting Serious with Vulkan*, Konferencja Khronos UK Vulkanised, 2017, https://www.khronos.org/assets/uploads/developers/library/2017-khronos-uk-vulkanised/005-Vulkanised-Dean-Sekulic-Getting-Serious-with-Vulkan_May17.pdf.

[43] Liam Dawe, *Testing The Talos Principle OpenGL vs Vulkan again, now that Valve have fixed the Steam Overlay*, https://www.gamingonlinux.com/articles/testing-the-talos-principle.7049.

[44] Michael F. Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, Neil Hunt, The Triangle Processor and Normal Vector Shader: a VLSI System for High Performance Graphics, ACM SIGGRAPH Computer Graphics, v.22 n.4, p.21-30, Aug. 1988.

[45] Shawn Hargreaves, Mark Harris, *Deferred Shading*, Prezentacja NVIDIA Developer Conference: 6800 Leagues Under the Sea, Londyn, 2004, http://http.download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf.

[46] Lance Williams, *Casting curved shadows on curved surfaces*, ACM SIGGRAPH Computer Graphics 12,3 (sierpień 1978), 270-274.

[47] William T. Reeves, David H. Salesin, Robert L. Cook, *Rendering Antialiased Shadows with Depth Maps*, Konferencja SIGGRAPH, 1987, http://dl.acm.org/citation.cfm?id=37435.

[48] Louis Bavoil, *Advanced Soft Shadow Mapping Techniques*, Konferencja GDC, 2008, http://developer.download.nvidia.com/presentations/2008/GDC/GDC08_SoftShadowMapping.pdf.

[49] Khronos Group, *Khronos Videos and Presentations*, https://www.khronos.org/developers/library/.

[50] Michael Larabel, *GL_vs_VK: A Micro-Benchmark Looking At The Overhead Of OpenGL vs. Vulkan APIs*, czerwiec 2017, http://www.phoronix.com/scan.php?page=article&item=gl-vs-vk.

[51] Damian Dyńdo, *GL_vs_VK: Comparison of OpenGL and Vulkan API in terms of performance*, Repozytorium kodu, https://github.com/RippeR37/GL_vs_VK.