VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

INTERAKTIVNÍ APLIKACE V API VULKAN

INTERACTIVE APPLICATION IN VULKAN API

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **RADEK BLAHOŠ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ MILET**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/20954/2017/xblaho05

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Blahoš Radek**
Obor: Informační technologie
Téma: **Interaktivní aplikace v API Vulkan**
Interactive Application in API Vulkan
Kategorie: Počítačová grafika

Pokyny:

1. Nastudujte metody 3D grafiky a knihovnu Vulkan.
2. Navrhněte nadstavbu GPUEngine v API Vulkan.
3. Implementujte nadstavbu GPUEngine a ukázkou aplikaci s využitím nadstavby. Aplikace bude interaktivní a bude zobrazovat animace.
4. Proměřte implementovanou aplikaci, zhodnoťte.
5. Vytvořte video s demonstrací odvedené práce.

Literatura:

- dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a kostra aplikace.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Milet Tomáš, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
L.S. 612 66 Brno, Božetěchova 2

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Práce pojednává o tvorbě nadstavbové knihovny geVk pro Vulkan API[1] a jejím využití při implementaci demonstrační aplikace. Představen je návrh knihovny, která se snaží o zjednodušení programování ve Vulkan API a zároveň o jeho co nejoptimálnějším využití. Text obsahuje výtah specifikace Vulkan nutný k čtenářovu lepšímu pochopení návrhu knihovny a případně jejímu efektivnějšímu využití při práci. Při popisu funkčnosti geVk knihovny jsou nastíněny různé strategie pro management paměti, zpracování GPU příkazů pomocí front (Queues) nebo optimalizované vytváření pipeline. Dále se v práci vyskytuje popis komponent využitých v rámci demonstrační aplikace – především rendereru, u nějž je rozebrán návrh jeho více-vláknové renderovací rutiny (vykreslování) a jeho propojení s Qt frameworkem.

Abstract

The goal of this bachelor thesis is creation of wrapper library over Vulkan API and its utilization during implementation of example aplication. Thesis proposes design of the library, which tries to simplify usage of Vulkan library and at the same time tries to use it in most optimal way as possible. Thesis contains extract of the Vulkan specification essential for reader to understand design of the geVk library and eventually for reader to be able to use it during programming his own graphic aplication. Description of geVk library presents memory managment, command buffer submiting or pipelines creation stategies. Thesis also suggests multi-threaded rendering strategy. Additionally thesis explains how to connect geVk library with Qt framework.

Klíčová slova

renderer, Vulkan, pomocná knihovna pro Vulkan, Qt rendering widget, optimalizace vykreslování ve a za pomocí Vulkan, optimalizované uložení scény pro Vulkan

Keywords

renderer, Vulkan, Vulkan wrapper library, Qt Vulkan Widget, Vulkan optimizations, Vulkan optimized scene storage

Citace

BLAHOŠ, Radek. *Interaktivní aplikace v API Vulkan.* Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Milet

[1] Application Programming Interface

Interaktivní aplikace v API Vulkan

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Mileta. Další informace mi poskytl Ing. Tomáš Starka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Radek Blahoš
16. května 2018

Poděkování

Tímto bych chtěl poděkovat především svým rodičům za materiální podporu, bez které bych se tak daleko nedostal, dále Adéle Jurčíkové za částečnou jazykovou korekturu a také Ing. Tomáši Miletovi a Ing. Tomáši Starkovi za nápomocné rady při tvorbě této práce.

Obsah

Kapitola 1

Úvod

Téměř každý člověk se již alespoň jednou v životě setkal s grafickým programem. Buď v podobě programu samotného, nebo s produktem, který by bez něj nemohl existovat. Počínaje volnočasovými aktivitami, jako je sledování filmů nebo hraní videoher, až po programy každodenně využívané v průmyslu, jako jsou programy pro návrh staveb či spotřebních produktů. Tato skutečnost dokazuje důležitost existence a velké možnosti uplatnění grafických programů na trhu a spolu s faktem, že mne tvorba grafických programů vždy fascinovala, tvoří důvody, proč je právě implementace grafického programu, resp. jeho nejdůležitější součásti – vykreslovacího jádra (rendereru), náplní mojí bakalářské práce. Od konkurenceschopného rendereru se očekává schopnost zadanou scénu vykreslit co nejrychleji, s maximálním využitím všech dostupných hardwarových prostředků a při uplatnění co největšího množství esteticky příjemných vizualizačních technik. Renderer dále musí umět objekty scény rozpohybovat. Jakým způsobem a do jaké míry se to daří mému programu je rozepsáno v kapitolách 5 a 6. K implementaci rendereru jsem využil aplikační rozhraní Vulkan, pro nějž jsem měl vytvořit pomocnou knihovnu, která s ním usnadní práci. Popisem výsledné knihovny se zabývá kapitola 4. Pro správné pochopení kapitoly 4 je třeba přečíst kapitolu 2. Zhodnocení celé práce pak obsahuje kapitola 8.

V téhle práci jsem si dal za cíl vytvořit kvalitní základ grafického jádra aplikace, jež se může s přidáváním dalších funkcionalit, jako je například editor scény, vyvinout v produkt, pomáhající s návrhem v průmyslu, tvorbou přídavných efektů ve filmech nebo tvorbou videoher. Mimo to, pro tuto práci vytvořené knihovny, hlavně knihovna pro práci s Vulkan, by se mohly stát pomocným prostředkem ostatních programátorů při vývoji jejich vlastních grafických aplikací.

Kapitola 2

Vulkan

Při psaní kapitoly jsem čerpal vědomosti z následující literatury [1], [3] a [2]. Tuto kapitolu jsem se snažil napsat podle toho, jak jsem danou problematiku pochopil a tak, aby byla srozumitelná i pro částečně počítačové grafice znalého čtenáře.

2.1 Motivace

Spolu s rostoucími možnostmi grafických karet rostla také potřeba jejich kontroly. Nabalování nové funkcionality na tehdejší API, tak aby byla zachována jejich zavedená struktura, ale zároveň bylo možné nové prvky naplno využít, byl úkol obtížný a někdy dokonce nemožný. Jako příklad je možné si představit specifikaci OpenGL před a po verzi 3.0., v níž bylo jeho rozhraní raději přepracováno, namísto vkládání nových prvků do verze předchozí. Navíc tím rozhraní prošlo optimalizací, díky níž API nabízelo větší výkonost. Bohužel to taky způsobilo nutnost vytvořit dva režimy práce s OpenGL - jeden zachovávající zpětnou kompatibilitu programů a druhý, o kompatibilitu se nestarající, schopen kompletně využít nové vlastnosti. Podobné řešení se však nedá aplikovat v případě vícevláknového přístupu. Již existující API byla navrhnuta jako vysokoúrovňová, což znamená, že nejnáročnější úlohy, jako je alokace paměti, vytváření a nahrávání příkazů pro GPU, či synchronizace, jsou delegovány na drivery grafických karet. Přijít se specifikací umožňující efektivní využívání paralelního přístupu a zároveň zachovávající jednoduchost používání API, se stalo nemožným. Drivery dodržující takovou specifikaci, by musely zvládat najít pro každý program vhodnou strategii zajišťující co nejmenší zadrhávání vícevláknového chodu v co nejkratším čase. Takové drivery jsou zatím hudbou budoucnosti. Jediným východiskem je tedy zformovat nové API vracející veškerou zodpovědnost za chod programu do rukou programátora. Synchronizace však nebyla jediným zádrhelem, který vývojáři grafických aplikací řešili. Další problém tkvěl v neustálém sledování a kontrole správnosti běhu programu i po tom, co už byl dostatečně odladěn. Drivery tím způsobovaly zbytečné a nechtěné vytížení procesoru. Navíc po celou dobu existence předchozí generace grafických API volala spousta vývojářů po možnosti ovládat GPU explicitněji. Drivery, na než padala veškerá tíha práce s GPU, činily exekuci programu nepřehlednou. Zatemňovaly některé aspekty práce s pipeline a kvůli vnitřním optimalizacím mohl programátor jenom těžko odhadovat kdy a která akce bude provedena, což je pro ladění a refaktoring naprosto nevhodné.
Bylo potřeba přijít s něčím novým...

O to se postarali inženýři v AMD, ve spolupráci s DICE[1], když roku 2013 přivedli na svět revoluční API – Mantle. Zakladatele rodiny moderních API, které konečně nabízelo nízkoúrovňový přístup k GPU, prostředky efektivní synchronizace mezi vlákny a GPU samotným a možnost vypnutí bezpočtu kontrol nad užíváním API. Vulkan je jeho přímým nástupcem.

2.2 Vlastnosti

Jedná se o **nízkoúrovňové** (velmi explicitní, například umožňující přímý management paměti daného zařízení), **multiplatformní** API. Právě schopnost pracovat nad co největším množstvím platforem (operačních systémů nebo hardware různých výrobců) odlišuje Vulkan od současné konkurence moderních s grafickými kartami pracujících API nejvíce. Vulkan je schopen pracovat nejenom s grafickými kartami, ale prakticky s čímkoli, co se dá považovat za koprocesor[2], například i s procesory zpracovávajícími signály[3]. Vulkan je, co se podpory různých operací týče, navržen flexibilně. Rozděluje funkcionalitu do tří základních skupin: Compute, Graphic, Transfer, z nichž Transfer operace musí umožnit všechna zařízení, ale podpora pro Compute a Graphic operace se může lišit hardware od hardwaru. Tím dovoluje obrovský počet zařízení, pro něž může být Vulkan využit. Vulkan výrazně **redukuje vytížení CPU** tím, že veškeré kontroly nad používáním API a sledování stavů objektů přesunul na tzv. **validační vrstvy**, které jsou využívány pouze při ladění aplikace. Poté, co je aplikace stabilní, jsou validační vrstvy vypnuty, aby zbytečně nevytěžovaly CPU. Na rozdíl od konkurence, je **shader kód** jenž předává Vulkan knihovna driverům jednotlivých zařízení, drivery **interpretován**, nikoli kompilován. Teprve až interpretací shader kódu driver vytváří kód jemuž dané zařízení rozumí. I když se to na první pohled může zdát, kvůli potřebě interpretu a pomalejší exekuci kódu, jako krok špatným směrem, ukázal se pravý opak. Interpretovaný kód odstraňuje režii spojenou s potřebou shader kód za běhu kompilovat (a s tím spojené kontroly syntaxe) a poté spustit. Čas potřebný k interpretaci kódu je menší než čas zabraný kompilací a následným spuštěním shader kódu. Navíc mezikód (bytecode) je lépe přenositelný mezi vícero zařízeními. Zavedením mechanismu **Rozšíření (Extensions)** představuje Vulkan elegantní způsob jak jednoduše přidávat API novou funkcionalitu, aniž by se musela struktura API nějak výrazně měnit. Vulkan je **objektové** API – definuje objekty, které mezi sebou vzájemně komunikují skrz Vulkan funkce. S Vulkan objekty se pracuje skrz jejich handle. Ty jsou buď unikátním id (číslem) mezi všemi vytvořenými Vulkan objekty anebo přímo adresy k objektům. Aktuální stavy objektů samotných stanovují, jakým způsobem by mělo zařízení pracovat. Ve Vulkan neexistuje žádný globální kontext jako například v OpenGL.

[1] Švédské herní studio, tvůrci série Battlefield a jednoho z nejmocnějších herních engine současnosti, Frostbite.

[2] Pomocný hardware sloužící k akceleraci nějaké činnosti: GPU, zvuková karta, DSP, ...

[3] Ty podporují pouze Compute a Transfer funkcionalitu API, zcela postrádají schopnost něco vyrenderovat.

2.3 Pojmy

Přichází pasáž přinášející vysvětlení základních pojmů a objektů v rámci API, poskytující tak základní přehled v rámci technologie.

2.3.1 Základní terminologie

Vysvětlení některých zavádějících pojmů využitých v následujících kapitolách.

- Host – zařízení na kterém běží aplikace.
- Zařízení – je externí hardware programovatelný skrz Vulkan API.
- Uživatel – programátor využívající knihovnu Vulkan.

2.3.2 Rozšíření (Extensions)

Aby zařízení splňovala Vulkan specifikaci, ale zároveň specifikace nenutila zařízení podporovat přebytečnou funkcionalitu, například zařízení sloužící pouze k akceleraci transferu dat, nemusí umět prezentovat na monitor, přichází Vulkan s konceptem **rozšíření**. Specifikace tedy definuje základní funkcionalitu, jejíž část musí všechna zařízení splňovat, a zbytek je považován za volitelné rozšíření. Jestliže chce programátor ve své aplikaci nějaké rozšíření využít, musí se nejprve skrz API dotázat na to, zda ho cílové zařízení podporuje, a poté ho explicitně zapnout. Tím dává driverům jasně najevo, které části zařízení bude využívat. Podle toho může driver věnovat pozornost jenom určitým komponentám a zbytek nechat bez povšimnutí (nemusí je vůbec aktivovat a tím snížít energickou náročnost zařízení). Rozšíření jsou zároveň způsob jak udržet Vulkan dopředu kompatibilní. Pokud se v budoucnu objeví nějaká převratná funkcionalita, okamžitě se může zabudovat do Vulkan ve formě rozšíření, aniž by se tím dotkla návrhu stávající specifikace. Vulkan rozlišuje dva typy rozšíření podle jejich platnosti v rámci objektů:

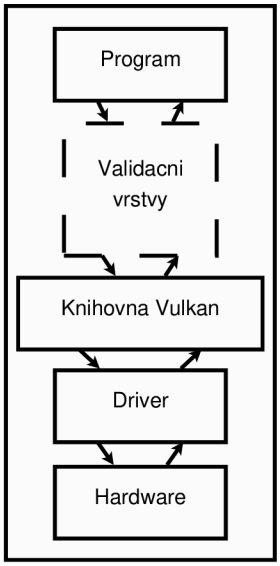
- *Instance* – aktivované pro celý program
- *Device* – dostupné pouze pro objekty z něj vytvořené.

Pro mou práci je nejdůležitější rozšíření Window System Integration, starající se o prezentaci image na obrazovku.

2.3.3 Vrstvy (Layers)

Programy, přes něž prochází invokované Vulkan funkce, nad kterými programy provádějí dodatečné operace. Nejčastěji se využívají validační vrstvy, mající za úkol dodatečnou kontrolu nad využíváním API. Slouží například ke kontrole parametrů funkcí, sledování životního cyklu objektů (kontrole správné dealokace objektů), sledování a kontrole práce vláken nad objekty nebo sledování funkčních volání pro profiling a debuging. Aby aplikace mohla zpracovávat hlášení z validačních vrstev, musí mít nadefinovanou callback funkci. Funkce se předává instance objektu, který skrz ni zpřístupňuje aplikaci výstup z validačních vrstev.

Aktivace ať už vrstev či rozšíření probíhá buď při vytváření instance nebo device, záleží na jejich typu (potřeba vyčíst z jejich dokumentace).



Obrázek 2.1: Obrázek byl převzat z [2]. Obrázek ukazuje komunikační cestu mezi programem a hardware. Jak již bylo zmíněno, ke zmírnění režie při běhu aplikace přenesl Vulkan celou tíhu kontroly využití API na validační vrstvy. Ty jsou zde zobrazeny šedivě, aby se zdůraznilo, že odladěná aplikace by už validační vrstvy používat neměla a její komunikace by měla probíhat přímo s knihovnou Vulkan. Vulkan dále předává požadavky programu driverům zařízení. Drivery se poté postarají o jejich doručení a vykonání hardwarem.

2.3.4 Formát zdrojů (Resource's Format)

Určuje jakým způsobem interpretovat data uložená v paměti zdrojů. Udává bitovou šířku a typ dat. Pro demonstraci například formát `VK_FORMAT_R4G4B4A4_UNORM_PACK16` udává, že data uložená v paměti zaberou dva byty, budou brána jako bezznaménková a v rámci těchto dvou bytů jsou uskladněny čtyři složky (hodnoty), z nichž každá se rozkládá na čtyřech bitech.

2.3.5 Shader

Program vykonávaný na zařízení (např. GPU), jehož funkcionalita je popsána uživatelovým kódem. Program může operovat buď nad vertexy, kontrolními body (tessellation control), tessellačními vertexy (tessellation evaluation), primitivy (geometry shader), fragmenty nebo pracovní skupině (workgroup v compute shader). Shadery tvoří základní kostru pipeline.

2.4 Objekty

Vysvětlení jednotlivých objektů API. Jejich výčet a vzájemné interakce jsou zobrazeny na 2.2.

2.4.1 Paměť zařízení (Device Memory)

Reprezentuje paměť, se kterou může zařízení operovat, a která je zároveň dostupná uživateli, aby ji mohl spravovat. Například v ní uživatel může ukládat obsah bufferů, image. Device memory může být jak paměť umístěná na zařízení, tak i paměť hosta[4]. Ve Vulkan je paměť rozdělena na několik typů, které mohou být alokovány z různých hromad (heap)[5]. Každý typ má své vlastnosti udávající, zda bude:

1. paměť viditelná i pro hosta (namapována v RAM).
2. ukládána v hostově cache.
3. alokována se zpožděním/na vyžádání (tzv. lazy allocation).
4. dodržována koherentnost (shodnost hodnot) dat mezi pamětí hosta a zařízení.

2.4.2 Instance

Uchovává stav aplikace v rámci API[6]. Zároveň tvoří pojítko mezi aplikací a Vulkan knihovnou. Představuje vrchol v hierarchii Vulkan objektů – všechny ostatní jsou z něj odvozeny. Při vytváření Instance se určuje, jak bude aplikace knihovnu Vulkan využívat. Definují se zde kompletní informace o aplikaci (jméno, verze, minimální potřebná verze API) a hlavně využívané rozšíření a vrstvy. Aplikace může vytvořit vícero instance, které jsou na sobě navzájem nezávislé. Instance odkrývá možnost manipulace s veškerým Vulkan podporujícím hardware na základní desce. V API se k němu dá dostat skrz objekt physical device.

2.4.3 Fyzické zařízení (Physical Device)

Reprezentuje jedno reálné fyzické zařízení umístěné na základní desce a dostupné k využití skrz API. Fyzické zařízení je jeden z Vulkan objektů, jenž se nevytváří. Ihned po vytvoření instance objektu jsou přes něj všechna dostupná fyzická zařízení k dispozici. Pak už je jen v rukou programátora, které fyzické zařízení zvolí pro vykonání své aplikace. Pokud má být fyzické zařízení dále využíváno, musí se nad ním vytvořit device objekt.

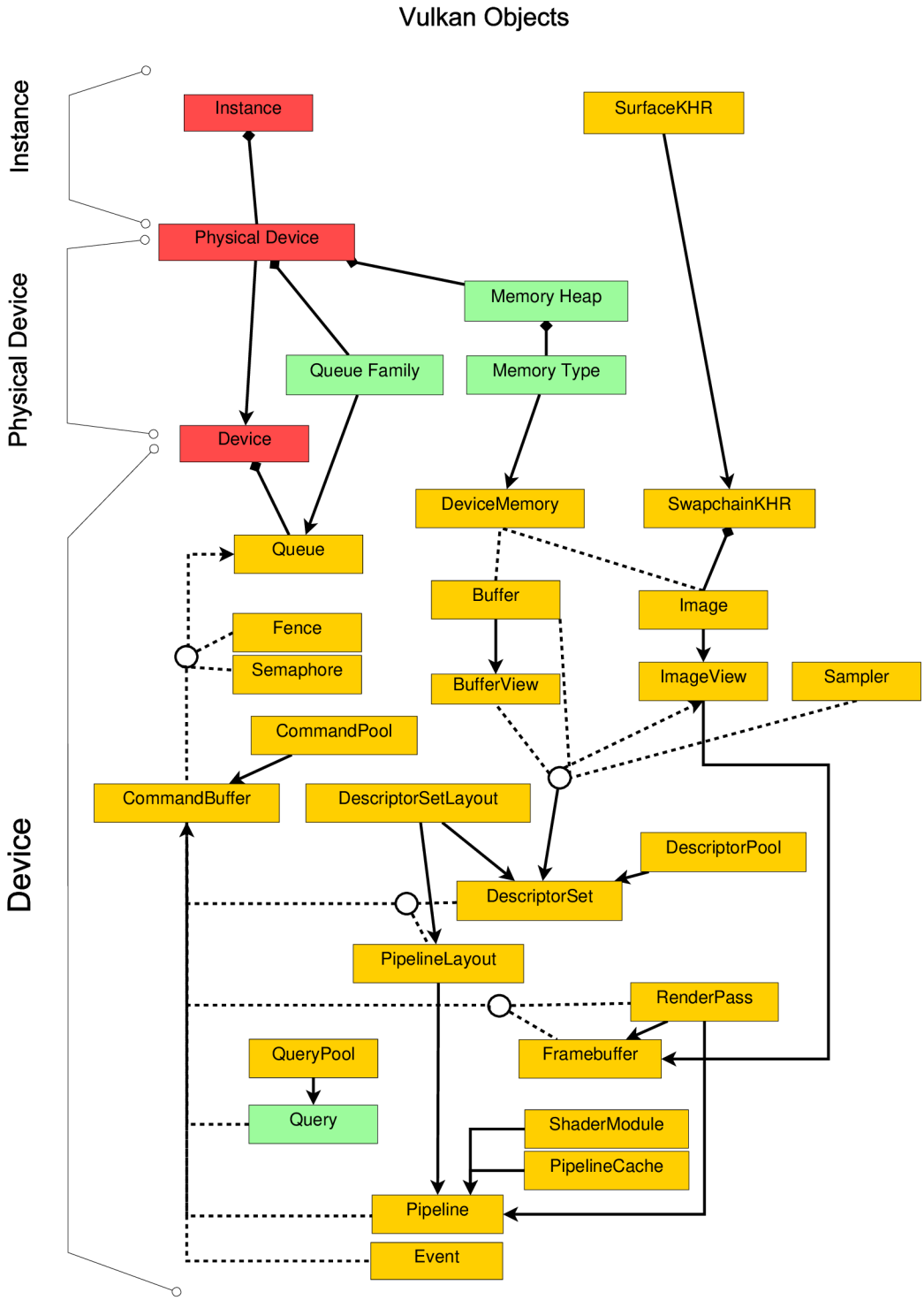
2.4.4 Zařízení (Device)

Výstižněji logické zařízení (Logical Device), představuje aplikační rozhraní k danému physical device. Zpřístupňuje veškeré zdroje, kterými hardware disponuje, ať už se jedná o paměť, fronty atd...

[4] Z ní si zařízení data vytáhne pomocí sběrnice do své paměti a pak si je přečte.

[5] Záleží na výrobci hardware, některé zařízení můžou mít jen jedinou hromadu, podporující všechny podporované typy zároveň.

[6] Stav aplikace fakticky vyjádřují objekty, které se z instance vytvoří.



Obrázek 2.2: představuje všechny Vulkan objekty a jejich vzájemné interakce. Každý Vulkan objekt je reprezentován svým typem, jež má předponu VkJménoObjektu, a hodnotou. Hodnota uložená v proměnné Vulkan objektu by se neměla vyhodnocovat jako obyčejné číslo nebo pointer. Mělo by se s ní nakládat jako s uzavřenou handle k objektu.

Obrázek 2.2: Objekty se zeleným zbarvením nemají své vlastní typy. Namísto toho reprezentují číselný index skrze který se lze k těmto objektům dostat z jejich rodičovského objektu, příkladem takových objektů jsou Query nebo Queue Family. Plné šipky ukazují pořadí vzniku objektů (aby se mohl vytvořit Command Buffer, musí existovat Command Pool, ze kterého bude naalokován). Plné čáry s diamantem místo šipky vyjadřují vztah kompozice. Kompozice ukazuje, které objekty již existují v rámci svých rodičovských objektů, skrz něž se k nim dá dostat, například z Instance objektu se dá přistoupit ke všem Physical Device. Přerušené čáry reprezentují ostatní závislosti mezi objekty, jako jsou nahrávání command bufferů do Queues nebo přiřazení jednoho objektu jinému, například přiřazení Device Memory Buffer objektu. Obrázek je rozdělen na tři části. Každá z nich má hlavní objekt vybarvený červeně. Všechny ostatní objekty dané části jsou z něj odvozeny (vytvořeny). Jako třeba u Pipeline, jejíž inicializační funkce potřebuje již existující Device, aby mohla být Pipeline zkonstruována. Kvůli zachování čitelnosti jsou v obrázku zanedbány závislosti mezi hlavními objekty a objekty z nich odvozenými. Diagram byl převzat z https://gpuopen.com/understanding-vulkan-objects/.

2.4.5 Fronta (Queue)

Fronta se stará o předání a vykonání veškerých příkazů aplikace na konkrétním physical device[7]. Příkazy nahrané do vícero front současně jsou vykonány paralelně. Fronty jsou součástí větších skupin (Queue Families), od nichž odvozují své vlastnosti. Jaké vlastnosti daná queue family reprezentuje a kolik různých skupin (families) bude, udává samotné zařízení, nikoli specifikace. Nicméně specifikace definuje obecnou množinu vlastností, které dané queue families mohou splňovat. Předně stanovuje jaké druhy operací bude daná fronta schopna vykonávat. Každá queue family podporuje minimálně jednu z množin operací (Graphic, Compute, Transfer).

2.4.6 Sampler

Spravuje stav vnitřní samplovací jednotky zařízení. Dovoluje nastavit různé filtrovací operace a transformace nad, ve shaderech čteným, image.

2.4.7 Shader Module

Jedna z částí pipeline. Obsahuje kód k shaderům, jež musí být ve formátu SPIR-V[8]. Výběr části modulu (kódu), který se má použít v dané exekuční části pipeline (pipeline stage), se děje na základě určení vstupního bodu[9].

2.4.8 Monitorování stavu (Query)

Jak již název napovídá, objekt se využívá ke sledování stavu zařízení. Na stav zařízení se dotazuje pomocí příkazů, jež se jako všechny ostatní nahrávají do command bufferu, a poté jsou předány do fronty. Výsledky dotazů jsou zasílány asynchronně a ukládají se v **QueryPool**, z něhož se výsledky získávají.

[7] Skrz physical device jsou fronty dostupné, nevytváří se.

[8] Jediný jazyk, kterému Vulkan zařízení rozumí. Jazyk je ve formě byte code. Naštěstí existují externí překladače z GLSL nebo HLSL do SPIR-V.

[9] Vstupní bod de facto znamená jméno funkce v rámci SPIR-V souboru.

2.4.9 Synchronizační primitiva

Všechny primitiva mohou nabývat pouze dvou stavů: signalizované/nesignalizované.

Fence

Slouží k synchronizaci práce mezi zařízením a aplikací. Zařízení objekt signalizuje, aplikace resetuje, dotazuje se na jeho současný stav nebo čeká na jeho signalizaci. Využíván je například k signalizaci ukončení práce zařízení nad společnými prostředky mezi zařízením a aplikací[10]. Obvykle bývá implementován s pomocí operačního systému, díky čemuž existuje způsob, jak jej konvertovat například na posix mutex.

Semafor (Semaphore)

Synchronizuje exekuci vícero command bufferů, předaných physical device frontám. Pro všechny fronty je jeho stav koherentní[11]. Pouze zařízení smí objekt signalizovat, číst, či resetovat jeho stav. Aplikace pouze určuje, které command buffery budou čekat na jeho resetování, či ukončení kterých command bufferů naopak způsobí jeho signalizaci.

Událost (Event)

Dovoluje synchronizaci mezi zařízením a aplikací nebo uspořádání exekuce několika command bufferů v rámci jedné fronty[12]. Stav události může být modifikován jak aplikací, tak zařízením. Využívá se třeba k synchronizaci exekuce odlišných částí stejné pipeline.

Bariéra (Barrier)

Nejedná se o objekt, nýbrž o příkaz. Umožňuje změnu přístupných operací nad zdroji, nebo výměnu vlastnictví zdrojů napříč frontami[13] mezi zadanými fázemi (stage) pipeline.

[10] Které byly definovány při nahrávání command bufferů.
[11] Ve stejný čas přečtou tu samou hodnotu.
[12] Pouze v rámci jediné fronty, jelikož stav události není koherentní vůči všem frontám.
[13] Pokud mají zdroje nadefinováno, že nemohou být současně využity více než v jedné frontě.

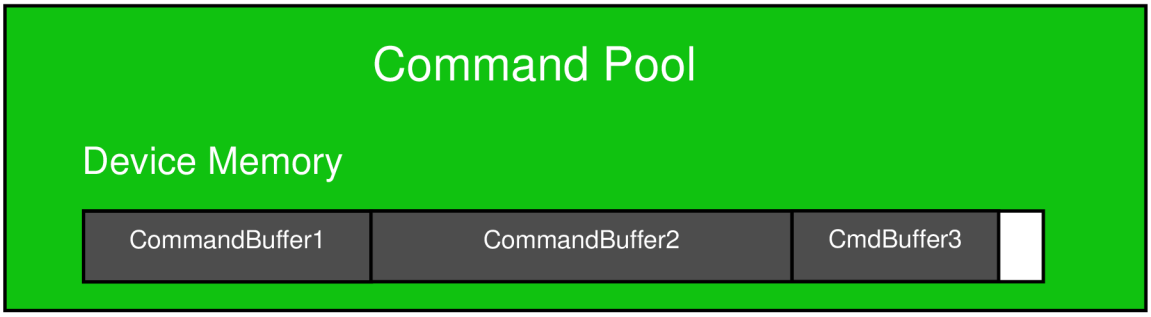
2.4.10 Nahrávání Příkazů

Command Pool

Spravuje paměť, z níž se alokují command buffery. Z jednoho poolu lze naalokovat a uvolnit několik bufferů zároveň. Tím se zmírňuje režie vznikající se spravováním životního cyklu každého command bufferu zvlášť. Command pool, a z něj alokované command buffery, musí mít definovanou queue family, se kterou budou kompatibilní. Jedině frontám patřící zvolené queue family lze z něj získané command buffery předávat. Při využívání poolu ve více vláknech je nutnost k němu zajistit výlučný přístup.

Command Buffer

Alokuje se z command poolu. Zaznamenává veškeré příkazy aplikace danému zařízení. K tomu, aby v něm uložené příkazy byly vykonány, je třeba command buffer nahrát do kompatibilní[14] fronty. Zároveň se v každém command bufferu uchovává aktuální stav exekuční části Vulkan, například jaká pipeline bude aktuálně použita a s jakými daty bude operovat. V jednom command bufferu může být nadefinováno víc exekučních stavů, mezi kterými se bude jeho postupným vykonáváním přecházet. Command bufferů existují dva typy, první, primární (primary) command buffery, slouží k přímému nahrávání příkazů a jejich následným předáním do fronty. Druhé, sekundární (secondary) command buffery, pak slouží spíše k uchování nahraných příkazů pro využití v jiných (primárních) command bufferech, například uchování příkazů, které jsou vícero primárním command bufferům společné. Aby mohly být příkazy uložené v sekundárních command bufferech vykonány, musí se nahrát do primárních command bufferů.



Obrázek 2.3: Obrázek předvádí, jakým způsobem se alokují command buffery z poolu. Přidělování paměti command bufferům funguje na principu lazy alokace. Každý command buffer začne s nějakým zanedbatelným množstvím přidělené paměti potřebné k uložení jeho metadat. Teprve v momentě, kdy se do něj mají nahrávat commandy, se mu přiřadí více paměti. Celková velikost command bufferu závisí na počtu v něm uložených příkazů. Obrázek ukazuje rozložení command bufferu v poolu pouze abstraktně. O tom, jak přesně jsou buffery ukládány, rozhodují drivery jednotlivých zařízení.

[14] S jakými frontami je command buffer kompatibilní určuje pool, ze kterého je obdržen.

2.4.11 Zdroje

Objekty sloužící k uchování programem využívaných dat. Aby zdroje mohly uchovávat nějaké data, musí jim být přiřazena paměť (reprezentovaná device memory objektem).

Buffer

Objekt nad souvislou částí paměti, která může a nemusí mít nějaký formát, využívanou k různým účelům, například jako vstupní data vertex shaderu, nebo jako uniform buffer[15].

Buffer View

Umožňuje vytvořit několik pohledů[16](views) nad jedním bufferem. Nemusí se tím vytvářet vícero bufferů s odlišnými formáty, nebo každou chvíli měnit jeho formát přes buffer barrier.

Image

K image se vážou následující pojmy které by bylo vhodné vysvětlit:

- **Texel** – Základní jednotka dat image, jejíž tvar je popsán resource formátem.
- **Image layout** – Udává, jaké operace jsou s image povoleny.
- **Uspořádání dat image (Images's Tiling)** – Určuje, jakým způsobem jsou data v image ukládána. Přípustné jsou dva způsoby:
  1. *Optimální* – texely jsou rozloženy v paměti takovým způsobem, aby s nimi zařízení pracovalo co nejrychleji.
  2. *Lineární* – jednotlivé texely jsou naskládány za sebou.

Image spravuje vícevrstvé (až 3 vrstvy) pole[17] dat, které musí mít pevně stanovený formát i layout. Využívá se třeba k ukládání textur, výsledků vykreslování (později přenesené na monitor) nebo jako vstup/výstup pokročilých operací (např. depth test). Aby se mohl image využít pro vstup/výstupní operace, musí být uveden v (nabindován) descriptor setu nebo jako attachment ve framebufferu.

Image View

Ve Vulkan se image v shaderech nepoužívají přímo. Místo toho se nad nimi vytvoří pohledy (views) obsahující jejich přídavná metadata. Kromě toho image view umožňuje práci s image podle ve view zvoleného formátu a s odlišným počtem vrstev (menší nebo roven). View však musí dodržet původní dimensionalitu image.

[15] Data, se kterými pracují předem stanovené části pipeline.
[16] Chování se k objektu podle odlišných kritérií, například jako by měl jiný formát.
[17] Polem se označuje souvislá části paměti.

2.4.12 Deskriptory (Descriptors)

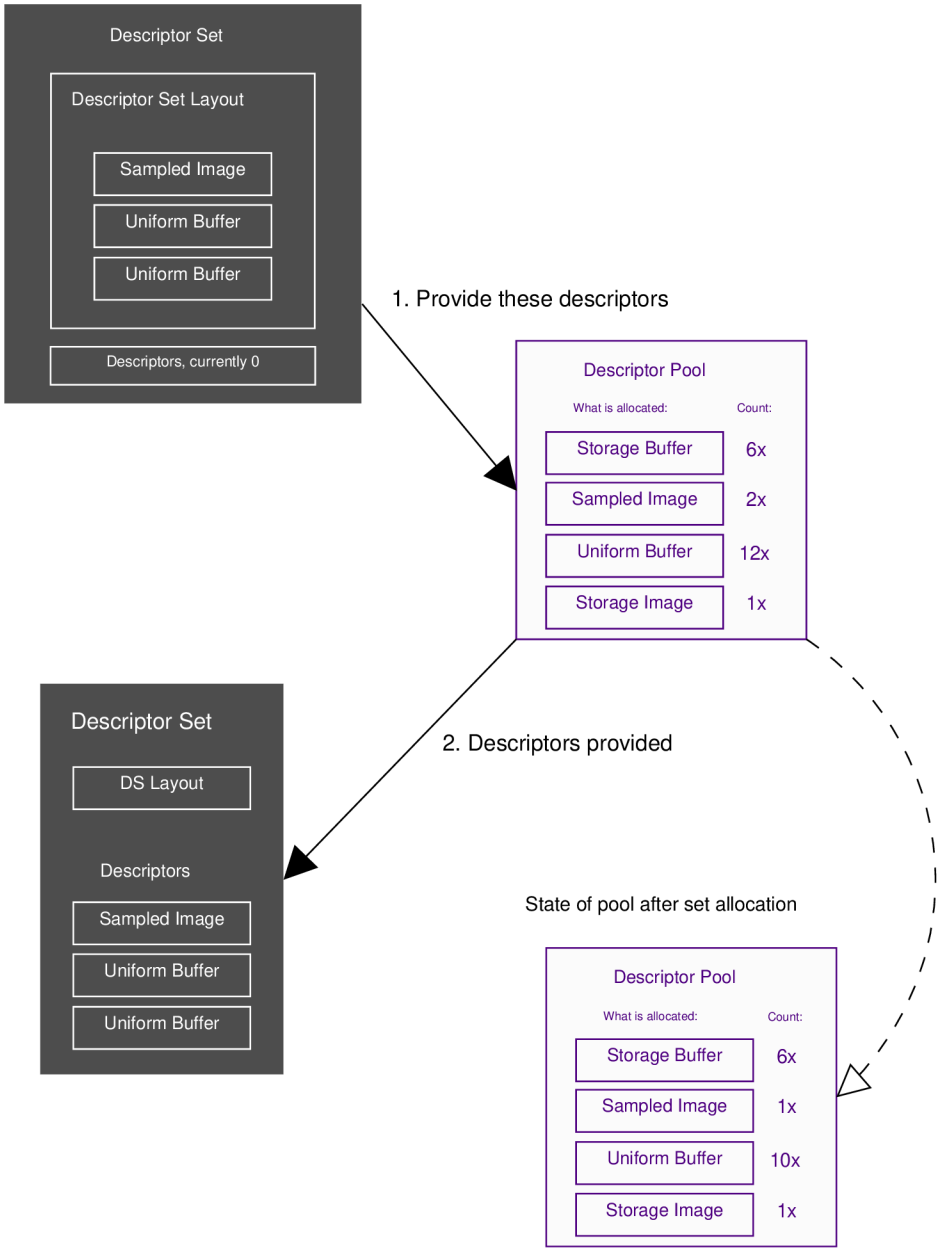
Zprostředkovávájí shaderům další zdroje, s nimiž mohou operovat. Deskriptory jsou speciální proměnné, kterými se mohou shadery k datům referencovat. Všem invokacím shaderů, jež mají nabindována data skrz deskriptory, jsou data přístupná v plném rozsahu (shadery mohou přistoupit k celé paměti zdroje dat) a všem invokacím stejně (všechny invokace shaderu se odkazují na stejná data).

Jakého typu můžou být data přiřazená (nabindována) deskriptorům, určuje **typ deskriptoru (Descriptor Type)**, například sampler, storage image, uniform buffer, storage buffer... Deskriptory jsou vždy sjednoceny do jednoho setu a jako s celým setem se s deskriptory jak v rámci aplikace tak i shaderů pracuje.

Teprve až přes **deskriptor sety (Descriptor Set)** se udává, s jakými deskriptory (potažmo daty, na které ukazují) pracují které shadery. Deskriptor setů může být pro jeden shader určeno více. Jelikož shadery tvoří jednu z hlavních částí pipeline, také deskriptor sety jsou součástí pipeline, tentokráte volitelnou. Aby se přes deskriptor sety mohly shaderům nabindovat jednotlivá data, musí se vědět s jakými typy deskriptorů shadery operují.

Ty se uvádí v **deskriptor set layout (Descriptor Set Layout)** jednotlivých deskriptor setů. Jelikož deskriptor sety potřebují pro své deskriptory naalokovat paměť, deskriptor set layout se též využívá při alokaci deskriptorů z deskriptor pool pro určený deskriptor set.

Deskriptor pool (Descriptor Pool) spravuje paměť, z níž se alokují deskriptor sety. Z jednoho poolu lze naalokovat a uvolnit několik setů zároveň. Každý deskriptor pool obsahuje určitý počet různých typů deskriptorů. Jaké, a kolik jich bude, se určuje během jeho inicializace. Jednotlivé deskriptory se následně přidělují deskriptor setům. Při využívání jednoho pool ve více vláknech je nutnost k němu zajistit výlučný přístup. Jak deskriptor sety zapadají do návrhu pipeline, je zobrazeno na obrázku 2.6.



Obrázek 2.4: Obrázek demonstruje, jakým způsobem probíhá alokace deskriptorů. Každý deskriptor pool obsahuje různý počet a různé typy deskriptorů. Ty se poté přerozdělují deskriptor setům. Alokace deskriptor setu se provede jedině tehdy, pokud je v poolu, po kterém se žádají prostředky, dostatek deskriptorů k přidělení. Jaké deskriptory deskriptor set požaduje, je uvedeno v jeho layout.

2.4.13 Push Konstanty (Push Constants)

Jsou dalším způsobem jakým zprostředkovat data shaderům. V rámci zařízení je shaderům vyhrazená velmi rychlá ale málo kapacitní paměť. Do ní lze přes Push konstanty ukládat malé množství dat – hodnot (proměnné, konstanty) jež shadery využívají. Přístup k hodnotám je pak mnohem rychlejší než v případě deskriptorů.

Aby deskriptory nebo push konstanty ovlivnily vstupní data shaderů, musí být v kódu jednotlivých shaderů nadefinováno, že shadery s nějakými deskriptory nebo push constants vůbec pracují.

2.4.14 Vykreslovací průchod (Render Pass)

Ve spolupráci s grafickými pipeline popisuje jeden vykreslovací průchod[18]. Render pass je tvořen dvěma seznamy. První seznam obsahuje seznam všech prostředků (attachmentů), se kterými bude render pass pracovat. U každého musí být uvedeno, jaké jsou jejich vlastnosti, například jaký mají mít formát, layout, a co se s nimi má stát na začátku a po skončení render pass. Attachmenty jsou nějaké image, které se během průchodu čtou, předávají dál mezi subpass nebo se do nich zapisuje. Nejčastějšími příklady attachmentů jsou color attachment (výsledný image celého vykreslování) a depth, stencil attachmenty, využité při depth/stencil testu. Druhý seznam obsahuje všechny podprůchody (subpass) vykonávající nad attachmenty nějakou činnost. Například technika Shadow mapping[19] by v jednom render pass obsahovala dva subpass. V prvním subpass by se výstupní color attachment využil jako shadow-depth mapa, která by se přivedla jako vstupní attachment druhému subpass. Ve druhém subpass by se shadow-depth mapa využila ke správnému vykreslení stínů do aktuálního color attachmentu (který již tentokrát bude prezentován na obrazovku). Mezi subpass se většinou uvádí závislosti (tzv. subpass dependencies) určující, jak se mají s přechody mezi subpass měnit layouty attachmentů.

Subpass

Část render pass popisující nějakou činnost nad attachmenty. Subpass jsou vykonány sériově v pořadí, v jakém jsou umístěny v seznamu předaném inicializační struktuře render pass. V každém subpass se definuje, jaké jsou jeho vstupní a výstupní (color, depth/stencil, resolve[20]) attachmenty, a které attachmenty mají zůstat jeho průběhem nedotčeny (preserve attachments). Každý subpass musí mít přiřazenou pipeline. Právě přiřazená pipeline udává, co se s attachmenty během subpass stane.

Příkazy pro render pass

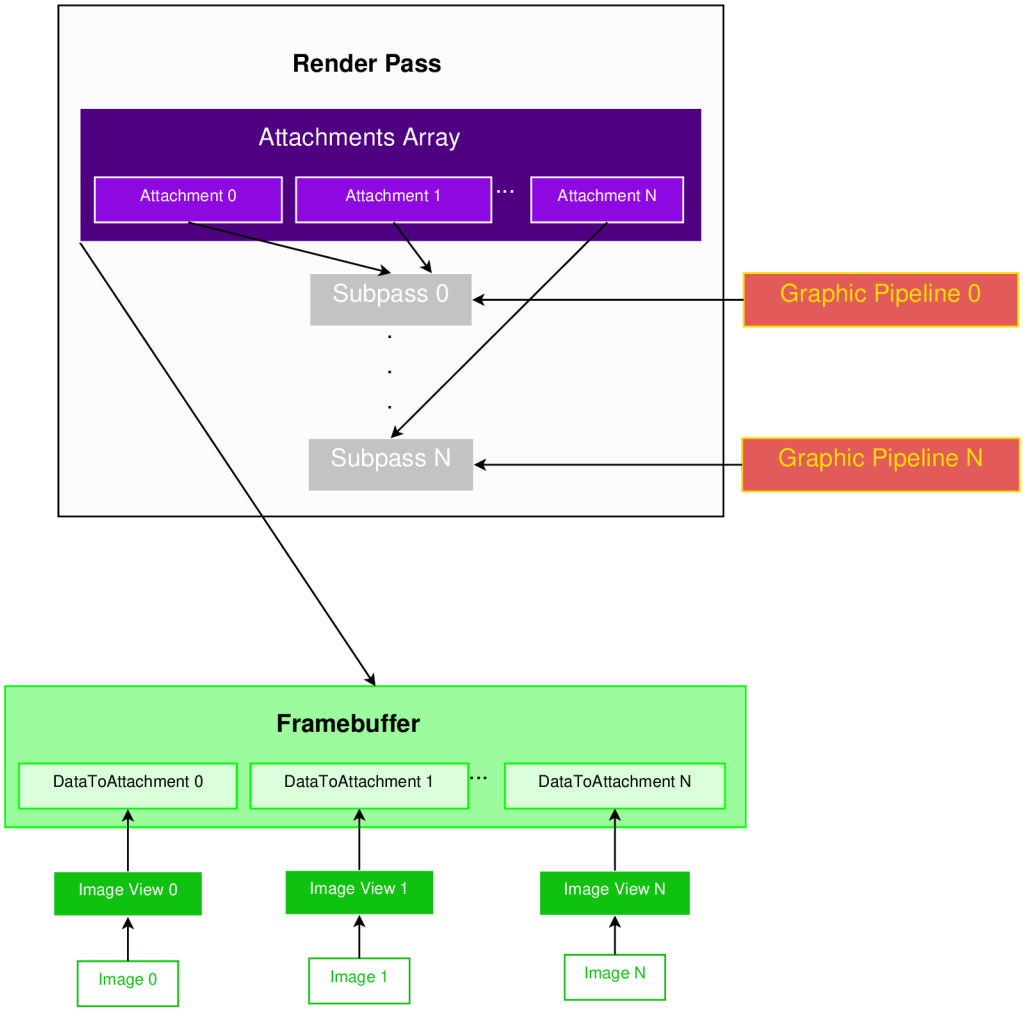
K zahájení provádění jednoho render pass slouží speciální příkazy určující kdy render pass začne, skončí a sloužící ke změně aktuálního subpass. Pouze mezi příkazy začátku a konce render pass se smí přiřazovat (bindovat) pipeline, descriptor sety, buffery (vertex, index) nebo nahrávat push constants.

Render pass popisuje attachmenty pouze abstraktně. Skutečná data, se kterými bude render pass pracovat, se přiřazují až ve Framebuffer objektu.

[18] Název objektu přesně vystihuje o co jde.
[19] Jeden ze způsobů jak simulovat stíny. Viz [6].
[20] Resolve attachmenty se využívají při multisamplingu.

2.4.15 Framebuffer

Určuje, jaké konkrétní existující image budou připojeny k render pass jako attachmenty. Tyto image pak budou během render pass modifikovány.



Obrázek 2.5: Obrázek ukazuje, jak spolu souvisí render pass, framebuffer a jaké další objekty jsou potřeba k vykonání jednoho render pass. Renderpass se skládá z několika subpass (minimálně jednoho) a attachmentů, s nimiž budou jednotlivé subpass operovat. Jaká činnost se v subpass provede, mu určí přiřazená pipeline. S jakými konkrétními objekty bude render pass pracovat, určuje framebuffer. Ten propojuje reálné objekty s popisy attachmentů v render pass.

2.4.16 Pipeline

Pipeline Cache

Umožňuje znovu použít prostředky vzniklé během předešlého vytváření pipeline. Cache dovoluje přenášet mezivýsledky v ní uložené napříč vytvářením dalších pipeline. Pokud vytvářené pipeline mezi sebou sdílí podobné vlastnosti, například stejný layout či shader moduly, může se recyklací prostředků vzniklých při předchozím vytváření pipeline tvorba nových pipeline značně urychlit.

Pipeline Layout

Určuje, jaká data přijdou na vstup jednotlivým shaderům. Činí tak za pomoci $0 \ldots n$ descriptor set layoutů a push konstant, které dohromady tvoří jeden celý pipeline layout objekt.

Ve Vulkan se vyskytují dva druhy pipeline, z nichž každá má jinou strukturu a plní odlišný účel:

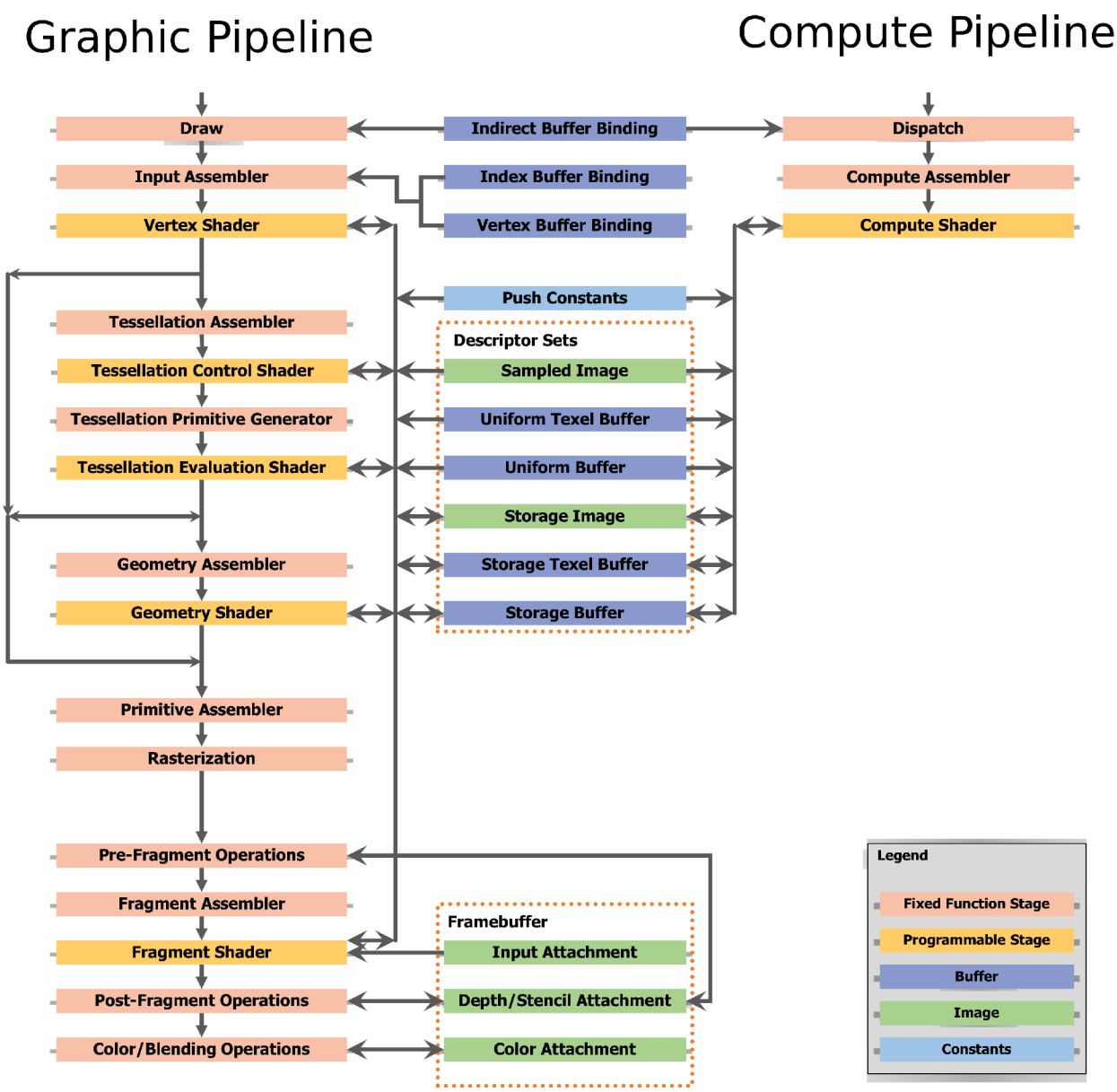
Compute Pipeline

Využívá se k vykonání různorodých paralelních výpočtů. Compute pipeline, respektive její shadery, slouží k akceleraci výpočtů, jež by na CPU zabraly spoustu času. Skládá se z jediného shader modulu a layoutu popisujících činnost a rozhraní pipeline. Z čeho všeho se compute pipeline skládá, popisuje obrázek 2.6.

Graphic Pipeline

Stará se o provádění různých grafických operací, nejčastěji o vykreslení vertexů tvořících nějakou scénu do image, který bude později prezentován na obrazovku. Vznikne spojením shaderů s fixed-function[21] částmi – oboje popisující chování pipeline a pipeline layoutu, které použitým shaderům vytváří rozhraní. Pipeline jednotlivé shadery a fixed-function části navzájem propojuje a tvoří z nich postupný exekuční řetězec. Ten se podobá produkční lince, po které postupují vstupní data, a každá část řetězce data nějakým odlišným způsobem transformuje. Spousta částí grafické pipeline je volitelná, nebo je hardware nemusí vůbec podporovat. Proto se aktivace některých shaderů (tesselation, geometry) musí vyžádat při vytváření device. Všechny fixed-function části grafické pipeline musí mít při jejím vytváření definovaný svůj stav. Důvodem, proč je třeba mít během vzniku pipeline všechno předem nastaveno, je zamezení nutnosti aby driver, po následném doplnění definice některé části, celou pipeline od znova přetvářel. To by zabralo příliš mnoho času. Avšak některé fixed-function částí mohou být za běhu přenastaveny, například rozměry renderovací oblasti (viewport). Jak vypadá úplná Vulkan grafická pipeline ukazuje obrázek 2.6.

[21] Části, kterým lze částečně nastavit stav, ale funkcionalita je pevně definována hardware.



Obrázek 2.6: Obrázek znázorňuje, z jakých částí se skládají jednotlivé pipeline a jaké jsou jejich vstupy a výstupy. Diagram byl převzat z Vulkan specifikace.

Vykonávání grafické pipeline začíná příkazem **Draw**. S jeho vyvoláním se okamžitě přechází do první části grafické pipeline **Nastavení vstupu (Input Assembly)**. Zde se seskupují vertexy tak, aby vytvořili požadovaná primitiva (body, úsečky, trojúhelníky). Dále se v Input Assembly připravují vstupní data pro vertex shader, získané z vertex bufferů. Input Assembly má pro každý vertex buffer určeno, kolik bytů z daného bufferu připadne na jednu invokaci vertex shaderu. Buffery se díky tomu dají indexovat. To, z kterého indexu přejdou data do aktuální invokace vertex shaderu, se určuje buď na základě hodnot v index bufferu, nebo podle sekvenčního pořadí. **Vertex shader** mu předaná data, podle svého naprogramování, transformuje a posílá dále. Následující čtyři části pipeline jsou volitelné. Pakliže jsou aktivovány tak, Tessellation control shader, Tessellation primitive generation a Tessellation evaluation shader vstupní data, která nyní interpretují jako celá primitiva, buď rozloží na vícero menších částí nebo jim nějaké další části přidají. Data poté postoupí do **Geometry shader**u, v němž se jedno primitivum může znásobit, či zcela odstranit. Posléze se vertexy ocitnou ve fázi **Sestavení primitiv (Primitive assembler)**. V ní dochází k převedení souřadnic vertexů do clip volume[22] a vyřazení vertexů spadajících mimo něj. Vertexy, jež nejsou odstraněny, přijdou do **Rasterizer**u. Ten se postará o jejich fragmentaci. Fragmenty putují do **Úvodních fragment operací (Prefragment operations)**. Tady podstoupí depth nebo stencil testy, pokud jsou zapnuty. Fragmenty, které jimi projdou, se následně dostanou do **Fragment shader**u. Fragment shader jednotlivé fragmenty, podle programátorem nadefinovaných operací, upraví a pošle dál. **Finální fragment operace (Postfragment operations)** se aplikují, pokud se zjistí, že fragment shader upravuje data, která už jednou prošla depth/stencil testem. Nad pozměněnými daty jednotlivé testy proběhnou ještě jednou. Poslední fáze je **zápis fragmentů** do framebufferu (Color/blending operations). Ta může finální framebuffer data ještě pozměnit, například smíchat současné fragmenty s již zapsanými. Pak už jenom dochází k zápisu dat do framebufferem definovaného color attachmentu.

2.4.17 Prezentace

Pro vykreslení obsahu color attachmentu na obrazovku je třeba se obrátit na WSI (Window System Integration) extension[23], vytvářející propojení mezi Vulkan a window systémem daného operačního systému. U WSI je především důležité zmínit následující pojmy:

WSI Platform

WSI Platform je Vulkan abstrakce window systému jednotlivých operačních systému (např. MS Windows, XCB, XLIB, Android...).

[22] Je pomyslný kvádr, mající střed v počátku souřadnicového systému a rozprostírající se od -1 do 1 v osách x,y. Na ose z je rozložen od 0 po 1.

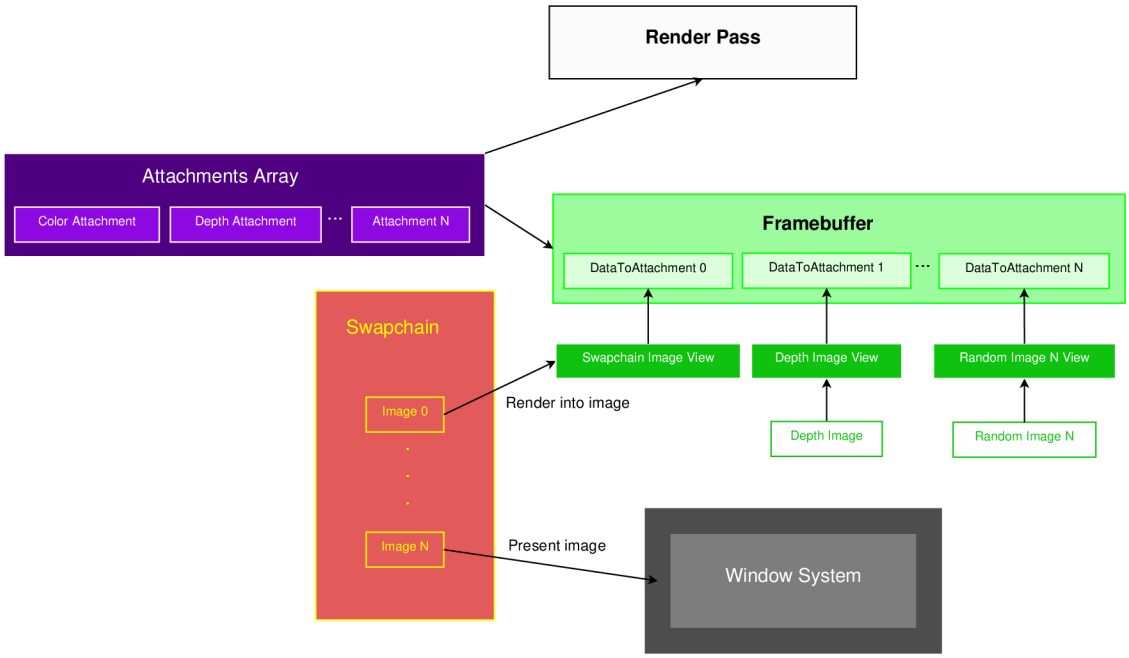
[23] Důvod, proč prezentace bufferu na obrazovku není ve Vulkan součástí jádra, je ten, že ne všechna zařízení musí umět pracovat s grafikou a velmi často má prezentaci na starosti spíše operační systém hosta než zařízení.

Surface

Surface je součást window systémů. Objekt, do něhož se zapisují data, která budou posléze operačním systémem promítnutá na obrazovku. Jelikož téměř každý OS implementuje vlastní window systém, bylo potřeba vytvořit univerzální surface objekt, který by byl kompatibilní se všemi. WSI surface vytváří nad jednotlivými surface WSI platformami obecné rozhraní, tak aby se s nimi dalo ve Vulkan pracovat jednotně – deklaruje jejich společné funkce a handle, kterým zpřístupňuje jejich datové struktury. Jelikož různé WSI surface můžou za svým handle vypadat úplně jinak, musí pro ně každá WSI platforma definovat vlastní konstruktor a všechny společné funkce určené standardem rozšíření.

Swapchain

Swapchain ze zadaného WSI surface vytváří a spravuje uživatelem stanovený počet image, se kterými poté může aplikace pracovat. Vícero swapchain image vytváří možnost tzv. multiple-bufferingu urychlující vykreslování. Multiple-buffering značí práci, v níž se současně pracuje minimálně se dvěma image. První, v němž je uložena podoba aktuálního snímku, je předán window systému k prezentaci. Do ostatních image probíhá renderování následujících snímků. Swapchain image se připojuje ve framebufferu jako color attachment, do něhož se zapisují výsledky renderování. Swapchain se následně stará o předání zpracovaného image operačnímu systému, aby jej vykreslil na obrazovku.



Obrázek 2.7: Obrázek vyjadřuje, jakým způsobem se pracuje se swapchain. Jeden z jeho image je přiřazen jako attachment render passu, do něhož se vykreslí aktuální snímek. Další z jeho image se současně předá window systému k prezentaci na obrazovku.

2.5 Shrnutí

Práce s Vulkan API je zpočátku velmi složitá, Vulkan rozhodně není jednoduché API jak na pochopení, tak na používání. K jeho zvládnutí potřebuje programátor spoustu času a zkušeností. Nicméně vytrvalého programátora odmění nebývalou svobodou při programování zařízení a kontrolou nad zařízením samotným.

Program psaný s využitím Vulkan zabírá spoustu řádků kódu: Jak již bylo několikrát zmíněno, Vulkan je explicitní API. Proto, než je vůbec možno cokoli vykreslit na obrazovku, byť jenom základní trojúhelník, musí se napsat kolem 7000 řádků kódu. To je způsobeno tím, jak API funguje. Vulkan je API vyvinuté pro jazyk C, z čehož vyplývá, že nejenom na zařízení, ale i na hostu, si musí uživatel všechno zařídit sám. Většinu objektů je potřeba explicitně vytvořit a odstranit, což je první věc, která přidává velké množství kódu. Další okolnost, která značně znásobí počet řádků kódu je to, že při vytváření jakéhokoli Vulkan objektu se o něm musí vyplňovat spousta informací. Totéž platí o jakékoliv větší operaci, jako je například nahrávání příkazů do front nebo prezentace Image na obrazovku. Z toho důvodu má každý Vulkan objekt pro svou inicializační funkci Create info strukturu, obsahující všechny podstatné atributy určující výsledné vlastnosti objektu. I každá důležitější Vulkan funkce dostává svou Info strukturu obsahující všechny k jejímu vykonání potřebné parametry. Ty se dále předají funkci jako jeden ze vstupních parametrů[24]. Všechny Vulkan funkce, až na výjimky v podobě funkcí nahrávajících příkazy do command bufferu, vrací návratovou hodnotu, značící úspěšnost jejího průběhu. Kontrola výsledku průběhu Vulkan funkcí je v tomto výčtu posledním důvodem, proč mají zdrojové soubory aplikace využívající knihovnu Vulkan tolik řádků kódu. Všechny napsané řádky Vulkan kódu jsou však nezbytné, respektive jsou daní za požadovanou explicitnost API.

Vulkan je i přes svou mohutnost skvěle navrženým API a skutečnost, že se snaží zachovávat pro všechny své funkce podobnou signaturu a obdobný způsob práce, z něj činí i API velmi intuitivní.

Jak vyplývá z předchozích řádků, Vulkan je jedno z nejmocnějších, k ovládání zařízení určených, API vůbec. Zároveň s tím i jedno z nejsložitějších. Uživateli zpřístupní téměř absolutní kontrolu nad zařízením, ale zároveň s tím na uživatele klade velkou zodpovědnost za to, jak zařízení využívá.

[24] Vulkan funkce obvykle obsahuje minimálně tři parametry. Prvním je objekt, s nímž zadaná funkce operuje, druhým je právě chování funkce určující Info struktura a třetím je objekt, v němž se uloží výsledek funkce.

Kapitola 3

Ostatní Teorie

3.1 Phong/Blinnův osvětlovací model

Jelikož Phong/Blinnův osvětlovací model nijak nerozšiřuji, ani mu nekonkuruji svým odlišným pohledem na věc, jen ho implementuji, nemusím ho podrobněji uvádět. Jenom zmíním, že se jedná o empirický model, snažící se modelovat šíření světla v každém bodě scény, který za stanovených světelných podmínek (počet a typ světelných zdrojů ovlivňujících scénu) spolu s definovanými vlastnostmi (normála ke každému bodu povrchu či jeho barva) povrchu určuje výslednou barvu modelů scény. Při jeho implementaci jsem se inspiroval teorií z [5], a [4].

Kapitola 4

Knihovna geVk

Spadá do sady knihoven pro práci s grafikou, vyvíjených programátory z Ústavu počítačové grafiky a multimédií na Fakultě informačních technologií VUT s názvem GPUEngine (ge) a její účelem je zjednodušit uživateli práci s Vulkan API (Vk). Knihovna je napsaná v jazyku C++.

Pokud dále nebude názvu objektu předcházet nějaký přívlastek (například Vulkan), popisuje se objekt knihovny geVk.

4.1 Klíčové vlastnosti

Knihovna geVk je vyvíjena s ohledem na maximální efektivitu a pohodlí při práci s Vulkan. Mezi její vlastnosti určené ke **zvýšení pohodlí práce** patří:

- Redukce množství kódu potřebného k využívání Vulkan objektů či jejich funkcí.
- Zjednodušení či kompletní přebrání zodpovědnosti za správu paměti Vulkan objektů, ať už se jedná o buffery, image, deskriptor sety nebo command buffery.
- Automatický úklid alokovaných Vulkan objektů.
- Usnadnění práce s frontami pomocí Command Processor objektu, starající se o nahrávání a přerozdělování command bufferů do front.
- Usnadnění ladění práce s geVk objekty uchováním jejich dodatečných informací.
- Zpřístupněním speciálního objektu Render Context obsahujícího veškeré k renderování potřebné Vk objekty.
- Zprostředkování výstupu validačních vrstev, pokud jsou aktivovány.

Efektivity se geVk snaží dosáhnout pomocí následujících vlastností:

- Snahou o vkládání příkazů do front tak, aby byly vykonány co nejdříve.
- Optimalizací vytváření pipeline – vytváření vícero pipeline současně je rozděleno do několika vláken a při jejich tvorbě se využívá pipeline cache. Z ní se jednak znovu využívají mezivýsledky předchozích vytváření pipeline a jednak se do ní mezivýsledky současné tvorby zapisují.

- Alokací většího množství paměti z jednoho heapu a následným přerozdělováním paměti prostředkům, které si o ni zažádají. Tím, že knihovna zavádí vlastní správu paměti, program využívající geVk knihovnu nevytěžuje během svého provádění zařízení neustálým žádáním o přidělení či uvolnění paměti[1].

4.2 Návrh

Knihovna geVk zapouzdřuje většinu Vulkan objektů do samostatného objektu, v němž se mimo Vulkan objekt samotný nachází ještě info objekt, uchovávající nejdůležitější atributy definující chování objektu. Takové objekty budou dále referovány jako **zapouzdřující (wrapper)** objekty. Příjemnou vlastností plynoucí ze zapouzdření Vulkan objektů wrappery je automatická dealokace Vulkan objektů se zánikem wrapperu. Ta je umožněna implicitním voláním destruktoru wrapper objektu C++ překladačem.

Ke každému wrapper objektu exisuje **Create info** objekt uchovávající všechny k jeho vytvoření potřebné atributy. Z Create info objektu se posléze vyjmou určité atributy, které se uloží do info objektu wrapperu. Knihovna geVk definuje dva typy těchto info objektů. První nazvaný stejně, **Info**, obsahuje pouze ty nejnutnější atributy, jež wrapper potřebuje ke své inicializaci či během své existence znát. Kdežto druhý objekt, **Further info**, obsahuje všechny atributy, které se daly v Create info, wrapperu nastavit[2].

Jelikož jsou Vulkan objekty skryty za svým handle, je za normálních okolností pro uživatele jediným způsobem, jak se dostat k hodnotám jeho atributů, použití specializovaného software jako je debugger grafických aplikací RenderDoc. Avšak geVk představuje způsob jak se k hodnotám dostat přímo z aplikace. A to přes info objekty uchované ve wrapperech. Info objekty zpřístupňují programátorovi možnost sledování vnitřního stavu Vulkan objektů, nad nimiž byl wrapper vystaven. Myslí se tím zejména jakých hodnot momentálně nabývají jeho modifikovatelné atributy (např. layout u image). Further info objekty obohacují monitorování ještě o možnost nahlédnout na to, s jakými vlastnostmi byly Vulkan objekty ve wrapperu uložené vytvořeny. Info objekty vznikly za účelem snadnějšího ladění aplikace s tím, že nejvíce informací o sobě prozradí wrapper za použití Further info objektů. Nicméně pokud je aplikace dostatečně odladěna, měly by se, kvůli úspoře paměti, do wrapperu ukládat pouze Info objekty.

Knihovna obsahuje i několik dalších info objektů (mimo těch uložených ve wrapper objektech), tentokráte určených k předávání parametrů Vulkan funkcím. Všechny info objekty (včetně Create info) mají své vzory v originálních Vulkan strukturách. Info objekty vznikly kvůli potřebě pozměnit originálním Vulkan strukturám jejich atributy. Kupříkladu mnoho Vulkan struktur obsahuje mezi svými atributy adresu na začátek dat a jejich počet, což volá po nahrazení takových atributů STL vectorem. Odpadnou tím uživatelovy starosti spojené s manuálním vyhrazením paměti a neustálým aktualizováním hodnoty počtu prvků pole ve Vulkan Info struktuře při každé potřebné změně velikosti pole. Info objekty jsou též prostředkem ke snížení počtu řádků aplikačního kódu. Spousta jejich atributů je knihovnou předdefinovaných na obecně využívané hodnoty, takže už je uživatel nemusí zadávat.

[1] Jedná se stále o rozpracovanou featuru. Na některých aspektech geVk memory managmentu je potřeba ještě zapracovat.

[2] Jak obšírné info se nakonec do objektu uloží, rozhoduje uživatel nastavením atributu abundantInfo v Create info při vytváření instance.

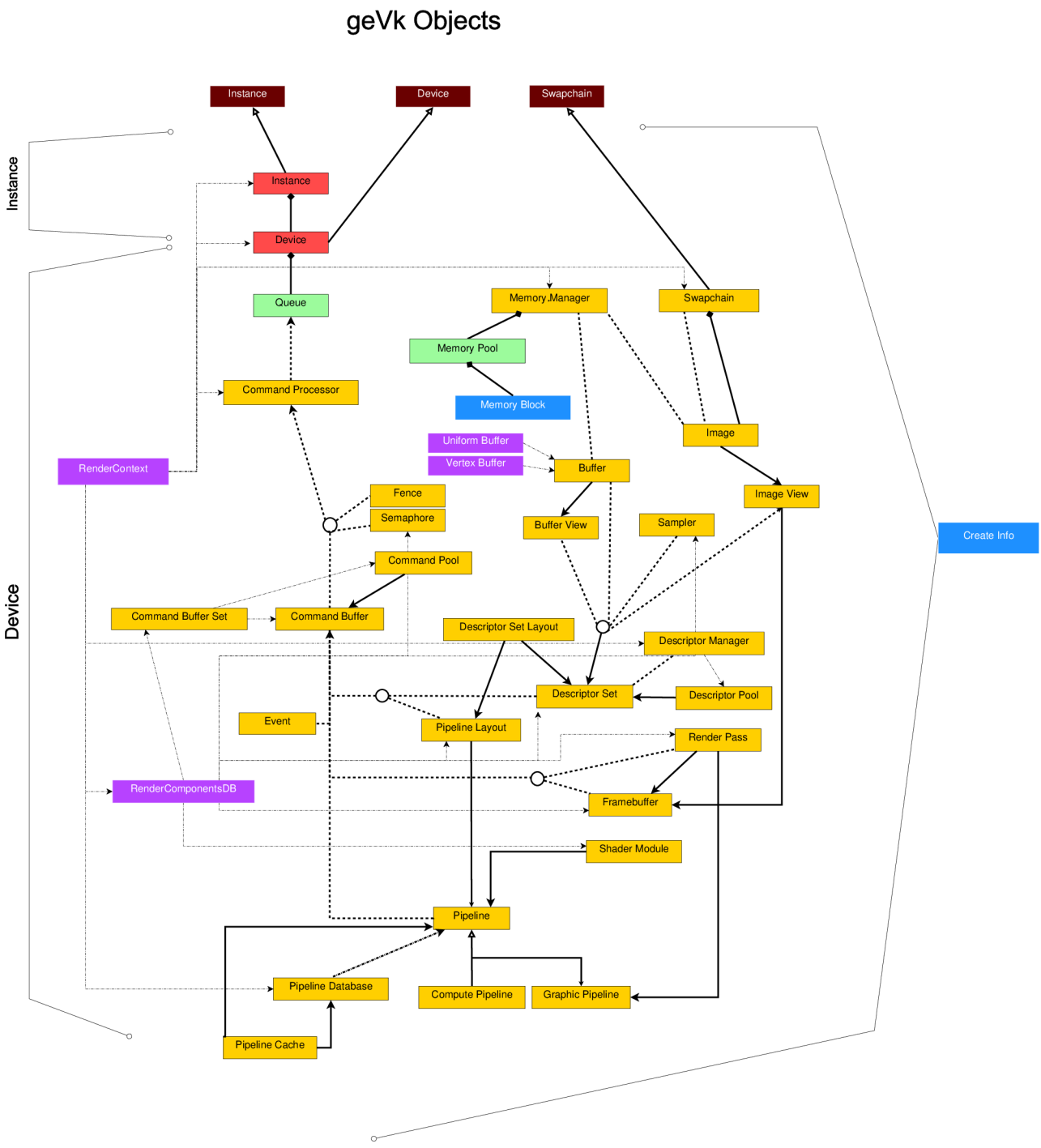
Tedy v geVk se nachází tyto skupiny objektů:

- **Wrappery** – vystavěné nad nějakým Vulkan objektem, zjednodušující s ním manipulaci. Pro všechny wrappery platí, že:
  - obsahují minimálně handle k jejich Vulkan protějšku a atribut description. Description je typu Info::JménoWrapperu a zachovává všechny důležité informace zadané během vytváření objektu.
  - implementují metody k jejich vytvoření a odstranění. Obsahují minimálně dva typy konstruktorů. Jeden vytvářející neinicializovaný objekt, připraven k pozdější inicializaci a druhý, vyžadující Create info k danému objektu, poskytující přímou inicializaci. Pro možnost pozdější inicializace s využitím Create info struktury, nabízí každý objekt init metodu[3]. K dealokaci Vulkan objektů wrapperu slouží deinit metoda, volaná též v destruktoru wrapper objektu.
  - mají několik getterů. Gettery jsou jimi poskytovány alespoň k jejich Vulkan objektům a Info objektu. U většiny objektů jsou atributy skrz gettery zpřístupněny pouze ke čtení.
  - se dají přetypovat na své Vulkan protějšky.

  Wrappery u kterých je z exekučního hlediska žádoucí[4] aby podporovali move operaci, ještě obsahují move konstruktor a operátor= (move provádějící).
- **Info objekty** – ty se dále dělí na:
  - Create info objekty, udávající attributy vytvářeného wrapperu. Dokážou se přetypovat na Create info struktury Vulkan objektů.
  - Info objekty, uchovávající všechny potřebné atributy daného wrapperu nebo parametry Vulkan funkcí.
  - Further info objekty, obsahující kompletní výčet všech nastavitelných atributů wrapperu.
- **Pomocné (Auxiliary) objekty** – uchovávají implementaci statických metod využitých ve více wrapperech.
- **Užitkové (Utility) objekty** – poskytující nějakou rozšířenou funkčnost nad wrappery. Například VertexBuffer nebo RenderContext, obsahující všechny k renderování potřebné objekty.

Většina geVk objektů má stejné jméno, stejnou funkcionalitu a pracuje se s nimi stejně, jako s jejich Vulkan protějšky. Dokonce se takové geVk objekty mohou předávat přímo Vulkan funkcím[5]. O těch platí víceméně to samé, co o Vulkan objektech stejného jména z kapitoly 2.

[3] Init metoda se využívá i v inicializačním konstruktoru, ten ji ve svém těle zavolá a skončí.
[4] Move operace má pro daný objekt uplatnění a nezabere příliš procesorového času.
[5] Pokud funkce vyžaduje objekt hodnotou, nikoli adresou.



Obrázek 4.1: Diagram představuje všechny důležité geVk objekty a jejich vzájemné interakce. Každý Vulkan objekt je reprezentován svým typem (třídou), který se nachází v namespace ge::Vk::JménoObjektu.

Obrázek 4.1: Objekty se zeleným pozadím jsou dostupné pouze skrz jejich rodičovské objekty, které řídí jejich existenci, například Queues nebo Memory Pool. Rámečky zbarvené do tmavě červené barvy symbolizují (Auxiliary) objekty. Ty nejsou závislé na žádných jiných objektech, pouze uchovávají implementaci metod, jež se můžou vyskytnout (podědit) ve vícero objektech. Fialově zbarvené rámečky reprezentují pomocné (Utility) objekty, obsahující nějaké ostaní geVk objekty poskytující nad nimi rozšířenou funcionalitu. Modré rámečky zastupují Info objekty. Ty uchovávají v rámci daného kontextu podstatné informace, například Create info objekty, v nichž jsou uložená data potřebná k vytvoření daného geVk objektu. V rámci přehlednosti jsou v diagramu všechny Create info objekty vyjádřeny jediným objektem, z něhož vystupují dvě šipky signalizující, že všechny ostatní objekty, které se mezi šipkami nachází, využívají svůj vlastní Create info objekt ke své inicializaci. Všechny objekty využívající Create info objekt obsahují vlastní Info objekt z Create info odvozený. V geVk je Info objektů daleko více, avšak aby se diagram příliš nekomplikoval, zůstala většina z nich opomenuta. Dále je kvůli zachování čitelnosti v diagramu zanedbáno přímé vyznačení závislostí mezi hlavními objekty a objekty z nich odvozenými. Závislosti jsou naznačeny sektory, kde každý z nich má hlavní objekt vybarvený červeně. Z hlavních objektů jsou odvozeny (vytvořeny) všechny ostatní objekty náležící stejnému sektoru. Jako třeba Memory Manager, který, aby mohl správně pracovat, potřebuje dostat již existující Device objekt. Plné šipky ukazují pořadí vzniku objektů (aby se mohl vytvořit Command Buffer, musí existovat Command Pool, ze kterého bude naalokován). Přerušované šipky s tečkami v mezerách označují vztah mezi objekty, kdy objekt, z něhož šipka vychází, vytváří šipkou označovaný objekt, ale šipkou označený objekt může být vytvořen i samostatně, například Pipeline Database vytvářející Pipeline. Plné čáry s diamantem místo šipky vyjadřují vztah kompozice. Kompozice ukazuje, které objekty již existují v rámci svých rodičovských objektů a skrz které se k nim dá dostat, například ze Swapchain se dá přistoupit k jejím Image. Přerušené čáry reprezentují ostatní závislosti, jako jsou nahrávání command bufferů do Command Processoru nebo přiřazení jednoho objektu jinému objektu, například přiřazení Image View Descriptor Setu.

4.2.1 Rozvinutý popis objektů

Následuje podrobnější rozbor objektů, které ve Vulkan buďto neexistují nebo mají odlišnou/rozšířenou funkcionalitu.

Device

Obsahuje jak zvolené VkPhysicalDevice, tak samotné VkDevice. Dále uchovává všechny vyžádané fronty (queues), ke kterým se jedině skrz device lze dostat. Objekt odstraňuje nutnost manuálně vyhledat vhodné physical device, posléze z něj vytvořit device a nakonec z něj získat handle k frontám. V Create Info pro device stačí pouze zadat, jaké vlastnosti se po physical device požadují, kolik front se má aktivovat a jaké mají mít fronty vlastnosti, či jaké extensions a layers se mají zapnout. O zbytek se postará konstruktor device objektu.

Buffer a Image

Chovají se stejně jako jejich vzory, nemusí se jim však přiřazovat paměť. O to se postará Memory Manager.

Vertex Buffer

Spojuje buffer s popisem určujícím rozložení jeho dat (Vertex Input description).

Uniform Buffer

Template objekt, který pro libovolnou strukturu vytvoří buffer o její velikosti. S uniform bufferem se pracuje úplně stejně[6] jako s objektem typu dané struktury, jen s tím rozdílem, že díky bufferu lze strukturu číst jak z hosta, tak z device. Nejčastější využití je nabindování jeho bufferu do deskriptor setu, čímž slouží jako vstup/výstupní data shaderům.

Pipeline Database

Slouží k uchování všech pipeline, které aplikace ke svému chodu potřebuje. Její nejdůležitější vlastností je schopnost vytvářet několik pipeline současně a ke zrychlení tvorby využívat pipeline cache. Při vytváření všech požadovaných pipeline se pipeline (resp. jejich Create Info) rozdělí do skupin o velikosti počtu k vytváření pipeline database vyhrazených vláken. Každá skupina bude obsahovat $\text{Pocet}_{(pipeline\ Create\ info)} / \text{Pocet}_{(dostupnych\ vlaken)}$ Create Info struktur, na jejichž základě se poté vytvoří pipeline. V každém vláknu se pouze zavolá Vulkan funkce schopna vytvořit několik pipeline zároveň jednou její invokací a zkontroluje se její návratová hodnota (zda se vytváření všech pipeline zdařilo). Vulkan funkce přitom využije pipeline cache.

Render Components DB

Uchovává objekty potřebné k renderování, jež jsou potřeba ve větším množství (mimo pipeline, ty už mají svůj vlastní úložní objekt), například shader moduly nebo render pass.

Render Context

Uchovává nejnutnější objekty potřebné k renderování. Její využití spočívá v jejím podědění externí třídou – znovuvyužitelnost ve vícero třídách.

Swapchain

Rozšiřuje základní VkSwapchain o framebuffery (vytvořené pro každý swapchain image) a depthbuffer tvoříce kompletní sestavu, do níž lze z pipeline přímo renderovat a jež může být následně prezentována.

Command Buffer

Oproti originálu obsahuje synchronizační primitivum, signalizující, zda-li je daný command buffer vykonáván frontou, či nikoli. Stále musí být alokován z nějakého comand pool.

[6] Lze přistupovat přímo k attributům struktury.

Command Buffer Set

Obsahuje několik command bufferů, command pool, z nějž jsou všechny jeho command buffery naalokovány[7] a pomocný registr, udávající dostupnost jednotlivých bufferů setu napříč celou aplikací (tedy jestli s command buffery momentálně některé vlákno či fronta pracuje nebo je možné je znova nahrát novými příkazy).

Command Processor

Stará se o nahrávání command bufferů do front. Pokud se nahrává vícero command bufferů, snaží se je vhodně přerozdělovat mezi vícero front tak, aby byly vykonány pokud možno současně (nejsou-li na sobě závislé[8]).

Descriptor Set

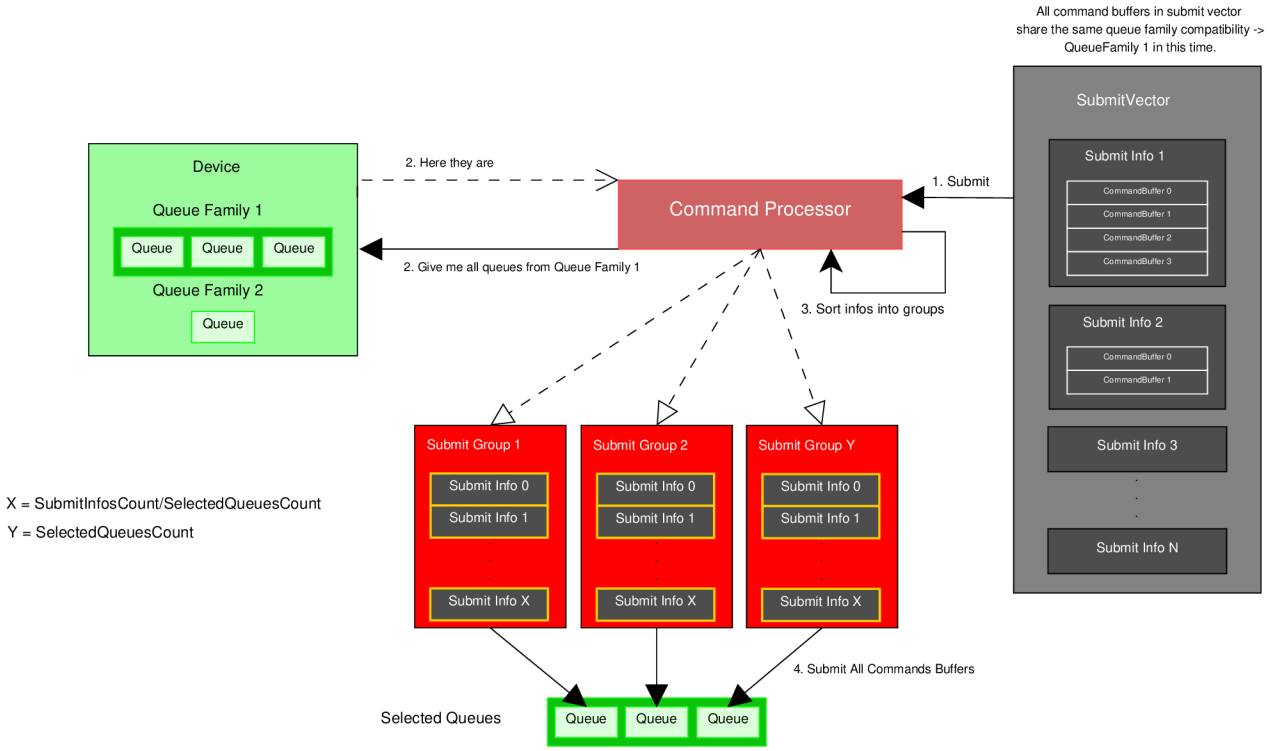
Na rozdíl od Vk objektu je geVk deskriptor setu paměť přiřazena automaticky Descriptor Managerem. Deskriptor set může obsahovat více deskriptor set layoutů, které jsou navzájem kompatibilní (mají společných prvních n deskriptorů), čímž lze nad descriptor sety vytvářet nové pohledy. Díky nim lze pracovat z částí deskriptorů deskriptor setu jako s úplně novým deskriptor setem vytvořeným podle daného pohled tvořícího layoutu. Tím se ušetří trocha paměti a nutnost, v některých případech, zbytečně vytvářet nové descriptor sety. Přitom decriptor set layout obsahující všechny deskriptory napříč layouty pohledy tvořících je hlavním layoutem. Právě přes něj se stanovenému setu alokují deskriptory.

Descriptor Manager

Správce paměti deskriptor setů. U každého deskriptor setu, který ho požádá o paměť, vezme Descriptor manager jeho hlavní layout a podle něj prochází jednotlivé jím spravované descriptor pooly a z nejvhodnějšího (toho z nějž je možné naalokovat všechny deskriptory uvedené v hlavním layoutu) alokuje pro daný descriptor set paměť. Při zániku descriptor setu vrací jím zabranou paměť zpět poolu, aby se paměť mohla později zase využít. Jeho činnost je zaznamenána na obrázku 4.3.

[7] Všechny jsou kompatibilní s jedinou queue family – lze je nahrát jenom do fronty, která z podporované queue family vychází.

[8] Závislý command buffer je ten, jehož alespoň jeden v něm uložený příkaz potřebuje běžet až po vykonání nějakého jiného příkazu uloženého v závisejícím command bufferu.



Obrázek 4.2: Diagram znázorňuje posloupnost událostí mezi objekty účastnícími se předávání příkazů frontám. Plné šipky označují inicializační zprávu, přerušené šipky bez plného hrotu označují odpověď. Přerušené šipky s plným hrotem značí výsledek jedné operace, konkrétně rozdělení Submit info do submit skupin:

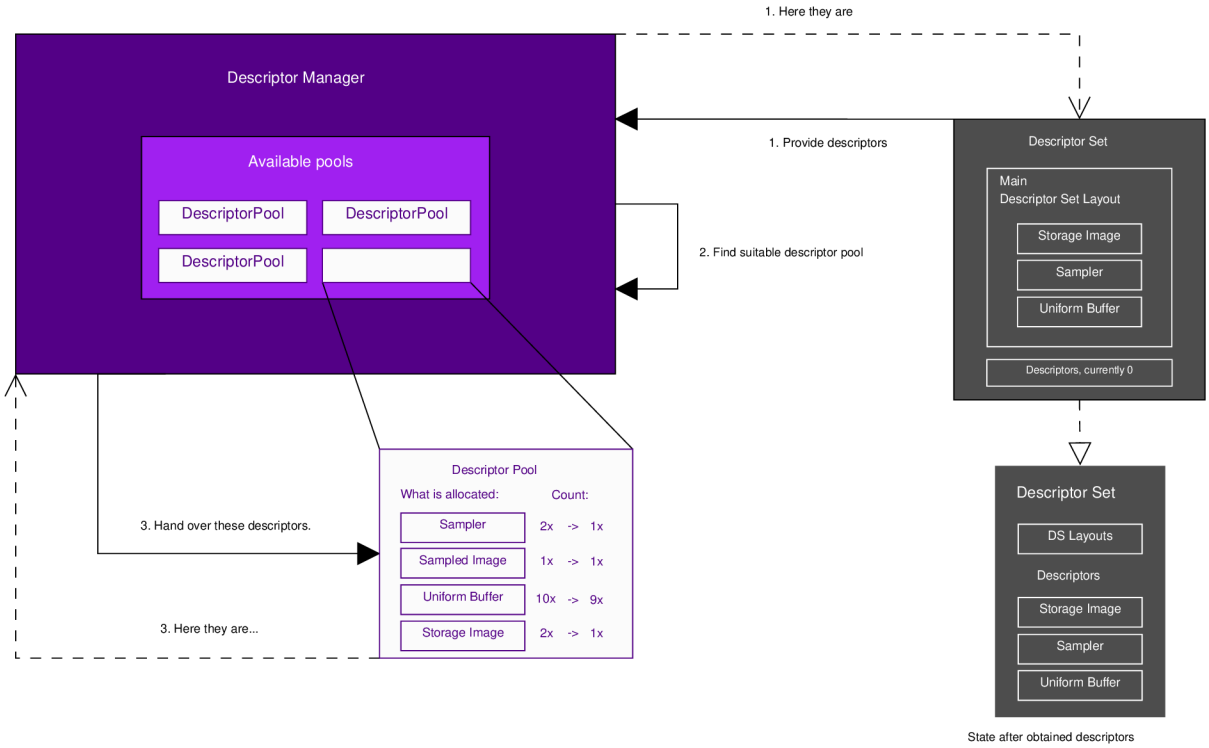
1. Aplikace vygeneruje několik command bufferů, jež je třeba předat zařízení k vykonání. Ty zapíše buď do jedné nebo několika Submit Info struktur – závisle na tom, jestli se mají command buffery nahrát do jedné nebo více front.

2. Aplikace předá vektor submit info struktur Command Processoru, který se postará o jejich přerozdělení dostupným frontám (na jednu frontu spadá $\text{Pocet}_{(submit\ info)} / \text{Pocet}_{(dostupnych\ front)}$ předaných submit info struktur.)

3. Command processor si vytáhne všechny dostupné fronty kompatibilní s předanými command buffery uvnitř submit info struktur a submit info struktury rozdělí do submit skupin.

4. Command processor předá submit skupiny jednotlivým frontám ke zpracování.

5. Aplikace běží dál, případně čeká na fence jednotlivých command bufferů, dokud s nimi zařízení nepřestane pracovat – nedokončí příkazy v nich uložené.



Obrázek 4.3

Obrázek 4.3: Diagram zaznamenává posloupnost událostí vedoucí k alokaci jednoho deskriptor setu.

1. Deskriptor set ve svém konstruktoru zavolá v Create Info uvedený descriptor manager a požádá ho o přidělení paměti deskriptorům.

2. Descriptor manager projde své předtím vytvořené deskriptor pooly, zda se z nějakého z nich nedají deskriptory naalokovat. Pokud se takový pool nenalezne, descriptor manager vytvoří nový pool, z něhož se paměť přidělí.

3. Až je vhodný deskriptor pool nalezen nebo vytvořen, descriptor manager z něj descriptor setu přiřadí paměť.

Memory Block

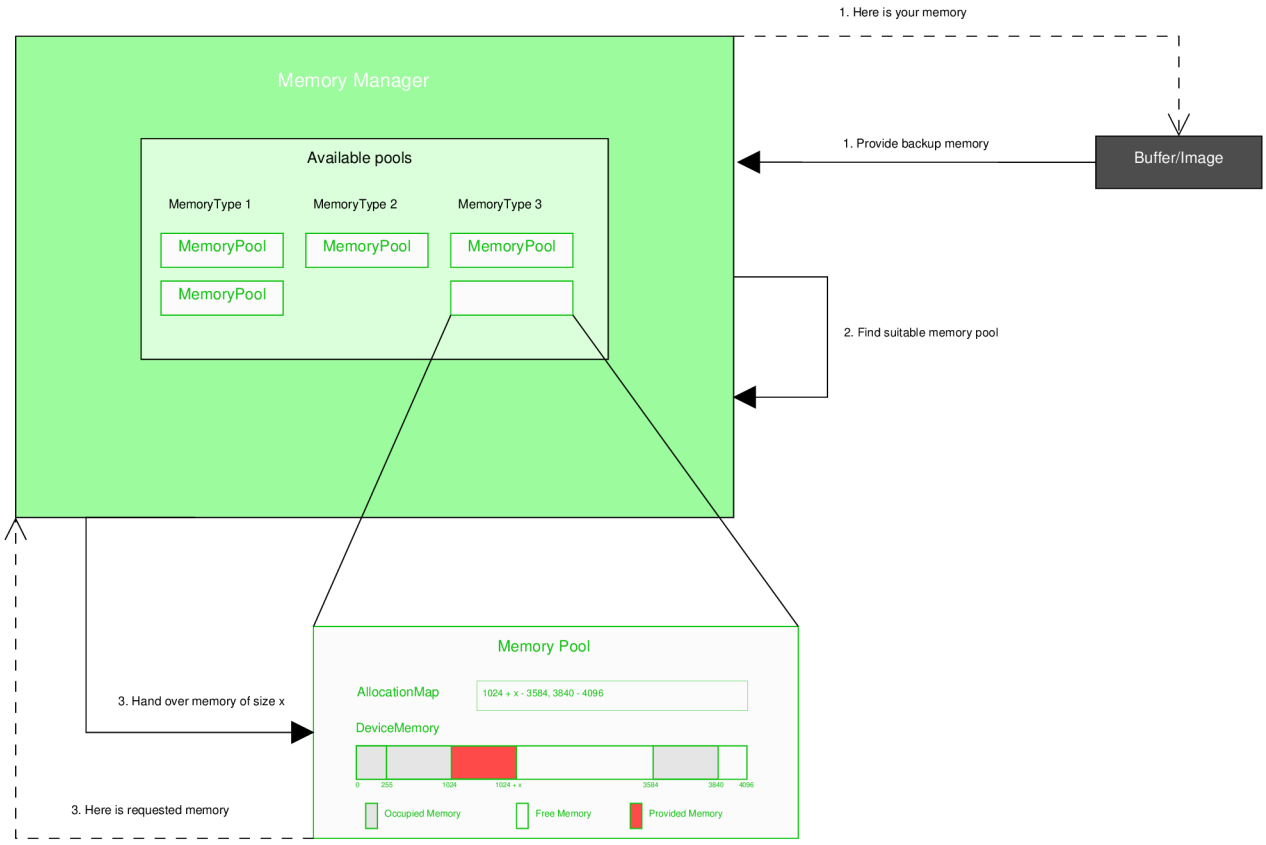
Uchovává informaci, ze kterého memory pool se paměť alokovala, její velikost a na kterém offsetu začíná.

Memory Pool

Objekt kontrolovaný Memory Managerem. Je mu přiřazena část VkDeviceMemory, kterou poskytuje zdrojům a u níž zaznamenává její využití.

Memory Manager

Správce paměti, kterou lze získat skrz Vulkan API. Uchovává všechny memory pools, z nichž je možné, až si o ni zdroje požádají, přidělovat paměť. Přidělování paměti je zachyceno na 4.4. Dealokace paměti probíhá podobně. Zdroje ve svém destruktoru řeknou jím přidělenému memory manageru, aby jim uvolnil paměť. Zdroj má ve své description struktuře uloženo, ze kterého pool byla paměť alokována, od kterého offsetu a kolik jí bylo. Tuto informaci si memory manager přečte a v cílovém pool vloží do pomocného objektu uchovávající informaci o volné paměti nový záznam obsahující offset a velikost alokace uvolňovaného zdroje. Tím označí oblast za neobsazenou.



Obrázek 4.4: Diagram objasňuje, jakým způsobem pracuje Memory manager.

1. Zdroje jako jsou buffery nebo image ve svém konstruktoru požádají určený Memory manager o přidělení paměti.

2. Memory manager projde své memory pooly a podívá se, ze kterého lze zdroji přiřadit paměť. Pokud se nenajde v žádném pool volné místo, vytvoří se pool nový.

3. Memory manager přiřadí zdroji jeho paměť. Přitom si v memory poolu upraví pomocnou strukturu určující od jakého bytu a kolik paměti je ještě dostupné.

4.3 Implementace

Knihovna je implementována čistě za pomoci C++ a Vulkan API. Při vytváření knihovny vznikla ještě jedna doprovodná knihovna. Doprovodnou knihovnu je důležité představit, jelikož bude v následující sekci zmíněna, protože je geVk knihovnou hojně využívaná.

Knihovna geAx (ge Auxiliary)

Pomocná knihovna sloužící k uchování pomocných objektů, funkcí, které nemají s Vulkan nic společného a daly by se využít i v jiných knihovnách. Obsahuje například třídy výjimek, objekt ke sledování využívání prostředků (UsageRegister) nebo funkce pro práci se soubory.

4.3.1 Významné části

Následuje část, ve které bude více popsána implementace některých aspektů geVk.

PtrVector (PointerVector)

Objekt z knihovny geAx, snažící se o odstranění problému plynoucího s ukládáním geVk objektů do vektorů. PtrVector je template objekt, který dědí z objektu STL `vector<T*>` (T – jméno typu). Přetěžuje remove metody (`clear, erase, pop_back`), aby spolu s pointery uklidily i objekty samotné. Ten samý účel plní i destruktor objektu.

Uniform Buffer

Vytváří buffer dosažitelný jak ze zařízení, tak i z hosta, díky čemuž si mohou snadno předávat data. Buffer viditelný pro hosta lze namapovat, čímž jednoduchým přetypováním pointeru, získaného buffer metodou map, na pointer cílové stuktury uvedeného v typename deklaraci template, lze s bufferem pracovat stejně jako s objektem typu dané struktury.

4.3.2 Implementační nesnáze

Největší problém, se kterým bylo třeba se během tvorby knihovny vypořádat, spočíval v používání STL vektrorů k uchování geVk objektů. Jelikož u žádného objektu nelze dopředu stanovit, kolik prvků daný vektor bude mít, vektory se v určité fázi musely realokovat. S tím je spojena destrukce všech v něm zatím uložených objektů, a po naalokování většího množství paměti vektoru jejich následná rekonstrukce. Zde nastane problém. Konstruktory objektů už nemají k dispozici původní Create Info struktury, skrz které byly nainicializovány, tudíž vzniknou prázdné[9] objekty, se kterými se v žádném případě nedá dále pracovat. Bohužel jsem při původním návrhu knihovny nevzal v potaz neefektivnost (s každou realokací destruování a rekonstrukce objektů) a nepraktičnost (nutnost předem znát velikost vektoru a tu už nikdy neměnit) tohoto způsobu. K vyřešení tohoto problému jsem použil následující postup – namísto objektů samotných uchovávám ve vektoru pouze jejich adresy. Pokud se vektor bude muset realokovat, adresy se překopírují správně, a jelikož se jedná o bázový typ, nemusí se nic rušit ani znovu vytvářet. Jediná nevýhoda spočívá v manuální dealokaci vytvořených objektů. Aby se dealokací uživatel nemusel pokaždé zaobírat, byl v pomocné knihovně geAx vytvořen PtrVector objekt (popsaný výše), jež dealokaci objektů, na něž se z vektoru odkazuje, řeší automaticky.

[9] Objekty s hodnotami inicializovanými na nedefinované nebo nulové hodnoty.

4.4 Shrnutí

4.4.1 Zhodnocení dosavadního stavu

Knihovna zapouzdřuje většinu Vulkan objektů do samostatných geVk objektů. Urychluje a usnadňuje tím tvorbu Vulkan objektů a sama se tím stará o jejich uvolnění. Dokáže buď poloautomaticky (programátor definuje určité parametry) nebo zcela automaticky spravovat alokaci paměti zdrojů. Dále zcela automaticky přiřazuje paměť deskriptor a command setům, čímž zbavuje uživatele starostí o alokaci jakékoli paměti. Zároveň uživateli nijak nebrání v tom, aby si paměť spravoval sám, bude-li chtít. Knihovna umožňuje efektivní vytváření pipeline a nabízí alespoň základní ulehčení práce s frontami přes command processor objekt. Tím, že jsem knihovnu použil k implementaci jednoduchého Dema, jsem si vyzkoušel, jak se s ní pracuje. Mohu říct, že opravdu snížila počet řádků kódu nejméně o 25%[10] a rozhodně práci s Vulkan, automatizací základních činností, ulehčila.

4.4.2 Návrhy k vylepšení

- Vylepšit management paměti. Vymyslet strategii rozumné alokace většího množství device memory z různých heapů, a tu poté přerozdělovat.
- Zlepšit managment paměti deskriptorů. V současném stavu Deskriptor Manager vytvoří pro každý požadovaný layout nový deskriptor pool[11]. Opět je třeba vymyslet nějakou strategii k vytvoření většího pool, z nějž lze většinu v aplikaci využitých deskriptorů získávat.
- Zdokonalit Command Processor, aby na základě Submit(ComandBuffer)Info dokázal přeskládat a vybrat k okamžitému provedení comand buffery, jež nezávisí na žádných jiných command bufferech.
- Dodělat podporu pro VkQuery a další objekty z Vulkan, které zatím nebylo potřeba využít.

[10] Možná to vypadá jako málo, ale pořád jsou spousty atributů, které je třeba pro správný běh Vulkan vyplnit.

[11] Zde vyvstává otázka, jestli je to až tak špatná strategie. Abych ušetřil čas na ostatní objekty této práce, příliš jsem se nezabýval hledáním ideálního řešení alokace deskriptorů.

Kapitola 5

geVk Renderer

5.1 Představení

GeVk Renderer tvoří jádro Demo aplikace, jež je výstupem této práce. Renderer je napsán za pomocí geVk knihovny, přes niž renderer komunikuje s knihovnou Vulkan. Renderer se snaží o efektivní vykreslování scény reprezentované objektem geVk Scene (viz 6.2.1). Celý proces vykreslování je popsán v sekci 5.3. Renderer vdechuje scéně život za pomocí Phongova osvětlovacího modelu, který je jednoduchý jak na implementaci, tak i na výpočet. Přesto nabízí přijatelnou modelaci světelných podmínek scény, a tím i o něco realističtější vykreslování. Více v sekci 3.1.

5.2 Speciální objekty

5.2.1 geVkScene

GeVkScene reprezentuje jednu celou ze souboru načtenou scénu. Uchovává si informaci o jediné kameře a prozatím jediném (na více světelných zdrojů bude potřeba upravit renderer) zdroji světla, které jsou umístěny ve scéně. GeVkScene využívá objekt scény definovaný v geSG knihovně, která scénu ukládá na hostu. GeSG scéna obsahuje obecný popis všech modelů (pozice jejich vertexů, cestu k jejím texturám) a taky obsahuje vlastní graf scény. GeVkScene tvoří prostředníka mezi geSG scénou a GPU. GeVkScene ukládá všechny data geSG scény na GPU za účelem rychlejšího přístupu k datům z Vulkan pipeline. Přesněji řečeno geVk Scene všechny mesh attributy umístí do bufferů, a ze všech cest nahraje textury a umístí je do image. U textur scéna kontroluje, aby byla každá textura nahrána maximálně jednou a poté, pokud bude třeba, byla referencována mezi objekty, které ji využívají.

5.3 Vykreslovací Rutina

Vykreslovací rutina má tři různé implementace. První dvě implementace představují dva rozdílné způsoby vykreslování zaměřující se na odlišné věci a v třetí implementaci se kombinují oba předešlé způsoby, tak aby byl geVk Renderer schopný vykreslit scénu uchovanou v grafu scény a s otexturovanými modely.

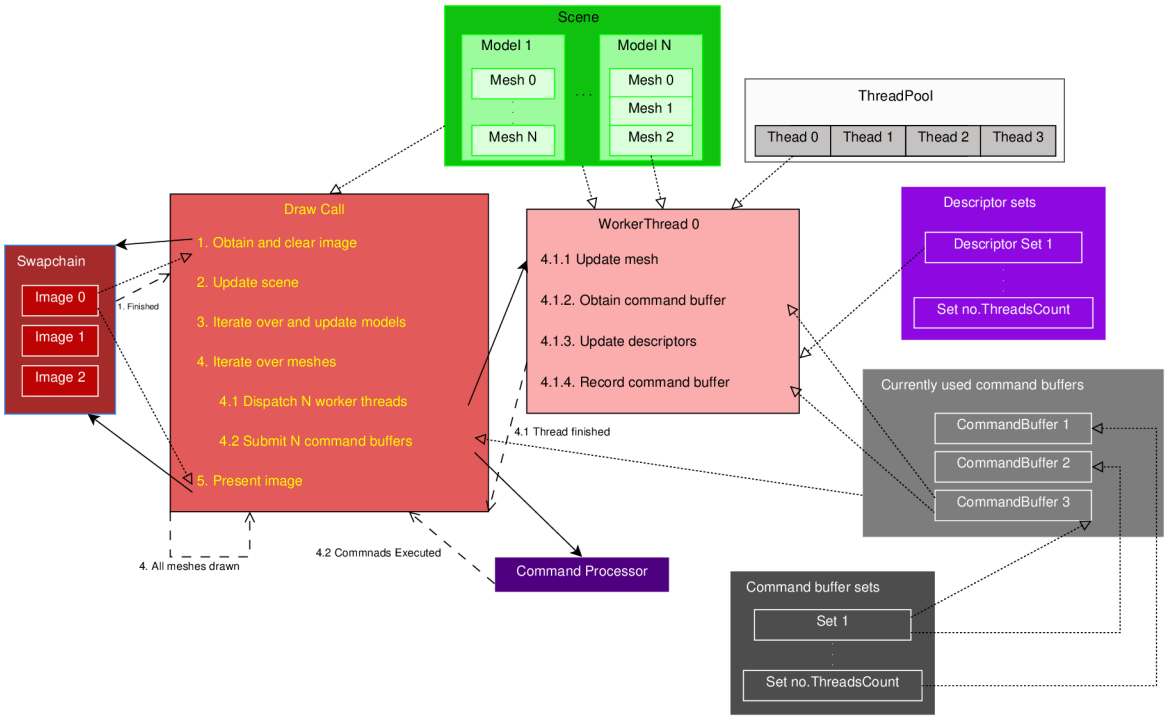
První rutina (**paralelní**) se snaží používat vícero vláken a vícero front k vykreslení několika mesh do swapchain image současně s cílem urychlit renderování scény. Popis její činnosti je vyjádřen na diagramu 5.1. Vzhledem k tomu, že první verze rutiny pracuje s

CPU i GPU prostředky souběžně, je potřeba práci nad nimi sesynchronizovat. Pro průběh renderování je nejpodstatnější vytvořit výlučný přístup nad framebufferem (v případě geVk Rendereru framebuffer uchovává současný swapchain image a depth buffer), kde se musí dbát na to, aby si prostředky v něm uložené paralelně běžící pipeline vzájemně nepřepisovaly. Vykreslovací rutina to řeší čekáním na k framebufferu přístup hlídajícímu semaforu v rámci úvodních fragment operací (prefragment operations). Další synchronizace je potřeba při práci s command buffery a deskriptor sety. U nich je třeba počkat, až s nimi GPU skončí práci. Vykreslovací rutina hlídá dokončení práce GPU s command buffery a prostředky v nich užitých za pomocí fence. K tomu, aby se dalo renderovat do image z více front, je potřeba pro každou jednu frontu vytvořit deskriptor set a command buffer set, čímž bude každá fronta schopna operovat nad odlišnými daty – vykreslovat jinou mesh. První verze rutiny se dá s úspěchem využít v případě, že uživatel požaduje zobrazit scénu obsahující mesh (množné číslo) lišící se navzájem svým popisem, neboli jedna mesh má zadané textury a indexy (určující pořadí zpracování jeho vertexů), druhá mesh nikoli. Nicméně existuje vlastnost, kterou musí mít mesh nadefinovanou vždy a pro každý vertex, a to normály, které se využívají při výpočtu Phongova osvětlovacího modelu. Kvůli tomu, že se v každé následující mesh mohou kompletně změnit parametry vykreslování (například má-li se využít textura či nikoli), mohou různé mesh vyžadovat při svém renderování odlišné prostředky. Třeba otexturované mesh k určení jejich barvy používají image (uchovávající jejich texturu) a sampler, kdežto neotexturované mesh si hodnotu své barvy vyčtou s, oproti texturované mesh přídavné, push konstanty. Z toho vyplývá, že je nezbytné mít pro otexturované a neotexturované mesh rozdílné shadery, a tím pádem i pipeline. Protože se s každou mesh mění data deskriptor setů, vstupní data vertex shaderu a použitá pipeline musí rutina před každým vykreslením mesh znovu nahrát command buffery, jejich příkazy budou odkazovat na objekty potřebné k vykreslení aktuální mesh.

Ukázalo se, že první způsob vykreslování má své nedostatky. Propojit první způsob vykreslování s procházením grafu scény se ukázalo jako velmi komplikovaný úkol, jehož splnění by vyžadovalo více času, než mi bylo dopřáno, a to zejména kvůli procházení grafu scény ve více vláknech. Navrhnout způsob, jak graf efektivně paralelně projít a zároveň nezpůsobit deadlock, je téma pro samostatnou bakalářskou práci.

Tudíž jsem pro práci s grafem scény navrhl v geVk Rendereru druhou (**předinicializační**) verzi vykreslovací rutiny. Ta usiluje o urychlení vykreslování scény nikoliv za použití vícero vláken či front, nýbrž odstraněním povinnosti command buffery před každým vykreslením neustále znovu nahrávat. Proto, na rozdíl od předchozího způsobu, se ukládá každý vertex attribut do jednoho společného bufferu a na místo přímého vykreslovacího příkazu se využívá příkaz nepřímý[1]. Tato rutina pracuje pouze s jediným deskriptor setem a jedinými (uniform) buffery, jenž jsou deskriptor setu nabindovány pouze jednou, dále už se nabindování deskriptorů po celou dobu vykreslování scény nemění. Všechny tyto úpravy dovolují jednorázové nahrání command bufferu při načítání scény a pak jeho znovuvyužití při vykreslování všech mesh scény. Přesněji řečeno se na začátku vytvoří command buffer pro každý swapchain image, respektive jeho framebuffer, tak, aby se pokryly všechny možnosti nastavení vykreslovacího Vulkan kontextu. Jediný důvod, proč poté znovu nahrávat command buffer, bude při změně velikosti okna, kdy se musí znovu sestavit celá swapchain a s tím nabindovat command bufferu nové reference na framebuffery. Druhá verze rutiny pracuje pouze v jediném vlákně a pouze s jedinou frontou.

[1]Parametry vykreslovacího příkazu se čtou z bufferu, nejsou zadány přímo ve volání příkazu.



Obrázek 5.1: Diagram znázorňuje průběh vykreslení jedné scény pomocí paralelní verze vykreslovací rutiny. Plné šipky označují inicializační zprávu, přerušené šipky bez plného hrotu označují odpověď. Tečkované šipky s prázdným hrotem značí vstupní parametry buď konkrétních akcí nebo objektů, například k vykonání pomocného vlákna je potřeba jedna mesh, scéna ze které mesh pochází a vlákno samotné. Kroky vykreslovací rutiny vypadají zhruba takto ... Ještě před začátkem vykreslovací rutiny se vytvoří pole deskriptor a command buffer setů o velikosti počtu současně využitelných vláken (N) za účelem aby každé vlákno mohlo pracovalo se svými vlastními prostředky. Tím se umožní souběžné nahrávání N command bufferů. Vykreslovací rutina začíná požádáním swapchain o vydání k vykreslení dostupného swapchain image, přičemž přikáže swapchain aby předchozí obsah image vymazala. Poté se aktualizují informace platné pro celou scénu, například pozice, typ a barva světelného zdroje. Následně se postupně iteruje skrz všechny modely scény a upravují se jejich atributy, například pozice v rámci scény. Pokračuje se iterací skrz všechny mesh scény. V jedné iteraci se postupně vybere předem stanovený počet N mesh a každá z N mesh se předá pomocnému vláknu. V rámci běhu vlákna se aktualizují vlastností mesh, například jejich lokální pozice vůči modelu samotnému. Poté se vlákno postará o získání volného command bufferu, do nějž by mohli být nahrány příkazy zajišťující vykreslení mesh do swapchain image. Následně pomocné vlákno zařídí nabindování správných zdrojů deskriptor setu použitého při vykreslování mesh. Ve finále se vlákno postará o nahrávání všech k vykreslení mesh potřebných příkazů do command bufferu. Pak se v hlavním vláknu počká na dokončení všech pomocných N vláken. Se skončením všech N vláken by mělo být v deskriptor setech a command bufferech uloženo všechno potřebné k současnému vykreslení N mesh do swapchain image. Ihned po ukončení pomocných vláken se všech N command bufferů předá Command Processoru k jejich vykonání. Nakonec hlavní vlákno vykreslovací rutiny počká až se provedou vykreslovací příkazy všech mesh. Poté vykreslovací rutina předá zpracovaný swapchain image k prezentaci.

Činnost předinicializační rutiny vypadá zhruba takto: Nejprve rutina získá volný a vyčištěný swapchain image, dále rutina upraví vlastnosti společné pro celou scénu (jako jsou například vlastnosti světelného zdroje), a pak rekurzivně nad celým grafem spustí metodu, která se postará o vykreslení jednotlivých mesh na scénu. Ve zmíněné metodě se prvně pro každou mesh vypočítá její transformační matice, přičemž se čeká na konec práce GPU se společnými vykreslovacími prostředky. Posléze se zapíše vypočítaná transformační matice mesh do uniformbufferu napojeného přes deskriptor set k vertex shaderu. Ihned poté se nastaví, zápisem do určeného bufferu, parametry nepřímého vykreslovacího příkazu, a následně se příslušný předem vytvořený command buffer pro aktuální swapchain image předá frontě k exekuci. Avšak u tohoto přístupu se zase vyskytl problém při určování textur, s nimiž má shader pracovat. Aby každá mesh mohla mít svou vlastní texturu, je nejjednodušším způsobem daný image, v němž je textura pro danou mesh umístěna, nabindovat na správný deskriptor před každým vykreslením mesh. To způsobí potřebu před každým vykreslením znovu nahrát command buffer, čemuž se snažím vyhnout. Řešením úlohy, pokoušející se o nahrávání command bufferů pouze jednou při načtení scény, by mohlo být nabindování přes deskriptor shaderu pole 2D Image (ve Vulkan 2D image s více vrstvami) uchovávajícího všechny textury, které mesh scény vyžadují. To by ale požadovalo (Protože shader potřebuje mít přesně stanoveno, s kolika texturami bude pracovat.) mít pro každou scénu vlastní shader. To je poněkud neobvyklý požadavek a žádný způsob reprezentace scény s ním nepočítá, takže jediná možnost je každé scéně nastavit shader manuálně, což je při nejmenším nepraktické.

Ve finále jsem tedy navrhl třetí verzi vykreslovací rutiny (**univerzální**), která neoplývá žádnými optimalizacemi, operace běží v jednom vlákně a na jedné frontě a před každým vykreslením mesh se musí nově nahrávat command buffery. I přesto se ukázalo, že běží rychleji než vykreslování paralelizující první verze. Třetí verze funguje na podobném principu jako druhá, pouze s modifikací, že se mezi aktualizací matic a jejich zápisem do uniform bufferu upraví deskriptor set tak, aby odkazoval na správné buffery a znova se nahraje command buffer, který se posléze odešle do fronty.

Ve výsledku, jak konečně potvrdilo i měření popsané v kapitole 6.4.2, se ukázalo, že ani jedna mnou zatím navrhnutá vykreslovací rutina není optimální. První rutina nevyhovuje kvůli své špatné efektivitě způsobené nešikovně navrženou kontrolou dokončování práce s objekty na GPU pomocí fence (Která způsobuje třeba i sekundové zaseknutí v běhu programu.), či kvůli režii spojené s tvorbou a čekáním na konec běhu vláken. Další věcí, jež činí první rutinu nepoužitelnou pro reálné nasazení, je nedokončené propojení s grafem scény. Druhá rutina je neaplikovatelná, jelikož postrádá, kvůli jednorázovému předzpracování command bufferů či deskriptor setů, možnost vykreslovat textury. Jejím dalším problémem je, že u ní lze během vykreslování použít maximálně jednu pipeline pro celou scénu. U třetího způsobu, i když umožňuje jak práci s grafem scény tak i texturami, avšak nepoužívá žádné optimalizace, lze očekávat optimálnost nejmíň ze všech.

Proto, aby mohl být geVk Renderer reálně použitelný, musí se jeho vykreslovací rutina přepracovat. Rád bych zde alespoň zjednodušeně představil návrh vylepšené vykreslovací rutiny. Avšak pro nedostatek zkušeností a kvůli nedostatečnému průzkumu již existujících řešení, kterými bych se mohl inspirovat[2], nechávám toto téma jako námět pro mé budoucí odborné práce.

[2]Protože, jak se ukázalo, stavit čistě na vlastním úsudku není úplně účinné řešení.

5.4 Shrnutí

5.4.1 Zhodnocení dosavadního stavu

GeVkRenderer ježto využívá knihovnu Assimp k načítání scény, zvládá nahrávat scény v nejběžnějších formátech jako jsou .obj, .dae, .blend, .3ds. V současném stavu je každá mesh limitována na maximálně jednu texturu a scéna může obsahovat maximálně jeden zdroj světla, což představuje do budoucna prostor k rozšíření.

5.4.2 Návrhy k vylepšení

- Navrhnout efektivnější variantu vykreslovací rutiny
- Přidaní shadow mappingu viz [6].
- Využití přídavných textur modelů, jako jsou normal, spectacular mapy určující vlastnosti povrchu modelů ovlivňující odraz světla. Zakomponování těchto textur při výpočtu osvětlovacího modelu přinese věrnější zobrazení scény.
- Umožnit na scéně více zdrojů světla.
- Přidání podpory pro kosterní animace.
- Při znovunahrávání command bufferů v rámci vykreslovací rutiny využít, tam kde to bude možné, k urychlení činnosti sekundární command buffery.

5.4.3 Použití

GeVkRenderer se dá využít jako samostatné vykreslovací jádro grafických programů. Jediné, co geVkRenderer potřebuje, aby mohl být využíván, je propojení s externí knihovnou starající se o výstup na obrazovku. Přesněji řečeno, potřebuje při své inicializaci dostat již předem vytvořený Vulkan surface.

Kapitola 6

Demo Aplikace

6.1 Představení

Demo aplikace je viditelným výstupem této práce. Při své činnosti zužitkovává všechny v předchozích kapitolách představené objekty.

6.2 Základní komponenty

Demo aplikace se skládá z několika základních komponent ...

6.2.1 geVkWidget

Propojuje geVk Renderer s QWindow, respektive umožňuje geVk Rendereru do QWindow vykreslovat. Oba objekty zapouzdřuje do jediného vykreslovacího widgetu, který je možné použít s ostatními Qt komponentami.

Dodatek k implementaci: Aby bylo možné využít externí Vulkan knihovnu s QWindow, je potřeba, aby byl vytvořen QInstance objekt, který má jako jediný objekt schopnost vytvořit v QWindow Vulkan surface, z něhož se posléze vytvoří swapchain image. Naštěstí QInstance dokáže adoptovat již předtím vytvořenou Vulkan instance, čímž vytváří způsob, jak i externí knihovny jako je geVk mohou vykreslovat do QWindow. Jednoduše se inicializuje instance podle externí knihovny, následně se vytvoří QInstance, jemuž je předána Vulkan instance vytvořená v externí knihovně, a poté QInstance objekt vytvoří v QWindow Vulkan surface. Nad surface externí knihovna vystaví swapchain, a pak už si externí knihovna pracuje podle sebe, aniž by se více musela o Qt starat. Je třeba dát pozor při úklidu objektů, které se musí uklidit v pořadí: swapchain, surface, a následně zbytek. V tomto případě swapchain, QWindow, až nakonec ostatní objekty z externí knihovny.

6.2.2 MainWindow

MainWindow obsahuje geVkWidget jako hlavní komponentu, která je ještě doplněna MenuBar a ToolBar, které kontrolují jeho chování. MenuBar a ToolBar vytváří komunikační rozhraní mezi uživatelem a geVk Rendererem. Na základě nabídek v MenuBar či ToolBar uživatel kontroluje, jaká scéna se momentálně zobrazí a jaká v ní bude pozice kamery či světla.

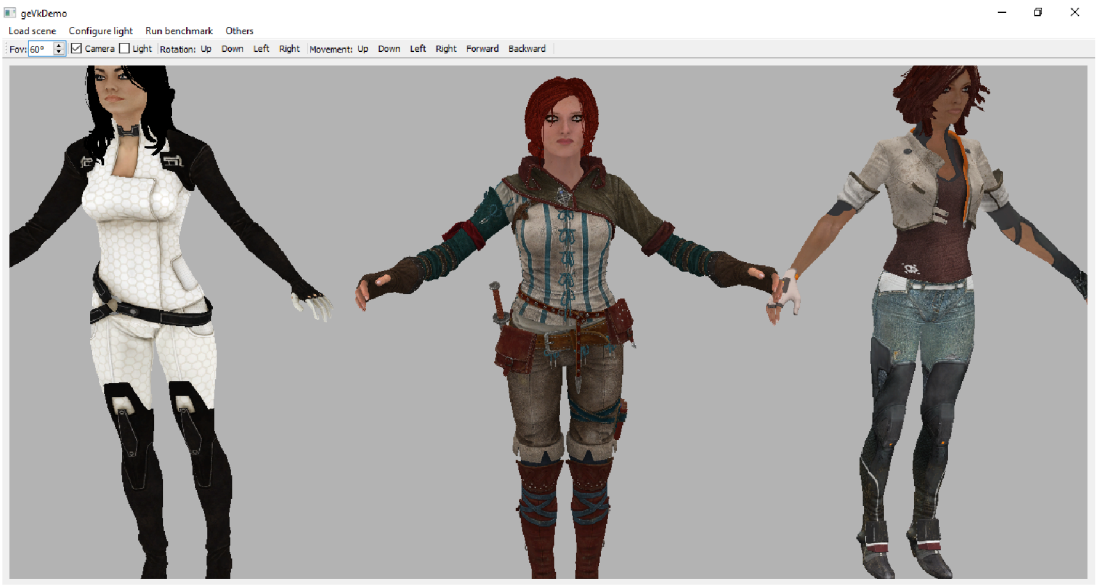
6.3 Benchmark

Demo aplikace nabízí možnost otestovat rychlost její implementace na dané platformě na základě jednoduchého benchmarku. 60krát po sobě se spustí renderovací rutina a pro každý její běh se naměří celková doba potřebná k zobrazení scény na obrazovku. Poté se ze všech měření vypočítá průměrná vykreslovací doba a vypočítá se hodnota FPS[1] podle vzorce $\frac{60}{BenchmarkTime}$. Výsledky se zanáší do souboru.

6.4 Shrnutí

6.4.1 Zhodnocení dosavadního stavu

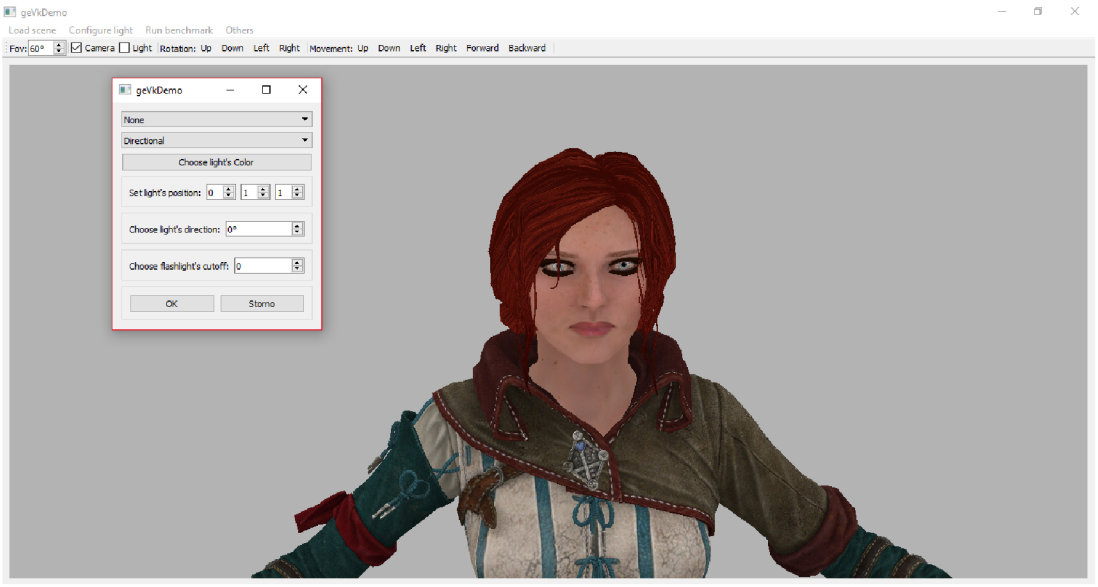
Jak aplikace momentálně vypadá, ukazují obrázky 6.1 a 6.2. Všechny modely scény viditelné na obrázku byly ještě minulý rok volně dostupné na `https://free3d.com`.



Obrázek 6.1

Obrázek 6.1: Obrázek zachycuje, jak vypadá celkové rozložení aplikace. Aplikaci vévodí většinu prostoru okna zabírající geVk Widget. Nad ním se nachází toolbar, který obsahuje tlačítka pro změnu FOV[2] a ovládání pozice kamery nebo zdroje světla. Úplně nahoře se rozprostírá menubar, nabízející tlačítka pro načtení scény, změny vlastností zdroje světla, spuštění benchmarku, či nabídku others, skrývající možnost vykreslení jednoduchého trojúhelníku za účelem rychlého otestování běhu Demo aplikace na různých platformách.

[1] frames per second



Obrázek 6.2: Obrázek předvádí, jak vypadá nabídka pro konfiguraci světelného zdroje.

6.4.2 Návrhy k vylepšení

- Vylepšit vzhled uživatelského rozhraní, například místo textu v toolbar použít ikony.
- Přepracovat zpracování vstupu, tak aby se četl i z klávesnice a myši.

Kapitola 7

Měření

Bohužel první verze vykreslovací rutiny, u níž by se daly měnit parametry potenciálně ovlivňující rychlost vykreslování, obsahuje data race, chybu, jenž nedeterministicky ovlivňuje čas exekuce vykreslování. Proto jsem se rozhodl, že pro paralelní verzi nebudu provádět ani samostatné měření, jež by ukázalo, jak se bude měnit rychlost vykreslování se změnou jeho parametrů, a tím pádem zde neuvádím ani výstup měření v podobě grafu.

Meření se provádělo v rozlišení 1366 x 768 pixelů, na sestavě obsahující AMD Phenom II X4 965 procesor, 8 GB RAM a GPU AMD Radeon HD 7850 s 1GB VRAM, přičemž na GPU byla k dispozici pouze jedna fronta. Měření proběhlo za použití scény, která obsahovala 1 334 892 vertexů, a všechny její mesh měly vlastní texturu.

Tabulka 7.1: Výsledky měření rychlosti vykreslování scény jednotlivých rutin

Vykreslovací rutina	1. Paralelní	2. Předinicializační	3. Univerzální
čas	$\cong 0.24\,s$	$\cong 0,015\,s$	$\cong 0,06\,s$
fps	± 4	± 66	± 16

Výsledky měření ukazují, že pokud je aplikaci ze zařízení poskytnuta pouze jediná fronta, je sériová vykreslovací rutina rychlejší než ta paralelní. Je to nejspíše proto, že neobsahuje režii spojenou s tvorbou a čekáním na ukončení paralelně běžících vláken. Výrazně nižší průměrný čas vykreslování u druhé vykreslovací rutiny je zajisté spojený s tím, že druhá rutina nepracuje s texturami. Nicméně, i tak se na výsledném čase projevilo předzpracování množství k renderování potřebných prostředků, jež nebylo nutné pořád dokola pro každou mesh nabindovávat.

Z výsledků měření je patrné, že ze současně dostupných vykreslovacích rutin je na tom nejlíp ta poslední. Sice běží asi 4x pomaleji než rutina předinicializační, ale na rozdíl od ní vyobrazuje textury. Obrovským zklamáním je rychlost paralelní rutiny, je však otázkou, jak by se rutina chovala, kdyby měla k dispozici vícero front a neobsahovala data race.

Kapitola 8

Závěr

Cílem bakalářské práce bylo vytvořit knihovnu ulehčující programátorovi práci s Vulkan API, a na ní postavit demonstrační aplikaci. To výsledná práce splňuje. Vzniklá pomocná knihovna pro Vulkan může směle sloužit jako pomocný prostředek pro tvorbu budoucích grafických aplikací. Knihovna umožňuje automatickou alokaci zdrojů, optimalizované zadávání práce GPU a značně redukuje množství kódu potřebného při práci s Vulkan. Výsledkem práce je nejenom velmi dobře použitelná knihovna, ale i referenční implementace rendereru (geVk Renderer), ukazující, jak by se s ní mělo zacházet. Spolu s knihovnou a rendererem byl vytvořen Qt Widget, jenž využívá geVk Renderer k vykreslení načtené scény do QWindow. Finální demonstrační aplikace používající Qt Widget dokáže vykreslovat libovolné modely za pomoci základních vizualizačních technik Phong/Blinnova osvětlení. Avšak cíl vytvořit kvalitní základ grafické aplikace, stanovený v úvodní kapitole, se mi v plné míře splnit nepodařilo. I když jsem se při tvorbě rendereru snažil klást důraz na efektivitu vykreslovacího procesu, v němž se dokonce vykresluje několik objektů současně, mnou navržený vykreslovací proces není natolik efektivní, aby mohl být nasazen v reálném prostředí. Proto se navržená aplikace nedá zcela považovat za kvalitní základ pro budoucí rozšiřování. Nicméně pokud by se vykreslovací rutina vylepšila, mohla by má práce sloužit jako spolehlivé jádro pokročilejší grafické aplikace.

Literatura

[1] Khronos: *Vulkan Specification*. [Online; navštíveno 14.05.2018].
URL https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html

[2] Lapinski, P.: *Vulkan Cookbook*. Packt Publishing, 2017, ISBN 978-1-78646-815-4.

[3] Sellers, G.; Kessenich, J.: *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Addison-Wesley, 2017, ISBN 0-13-446454-0.

[4] Vries, J.: *Learn OpenGL - Blinn reflection model*. [Online; navštíveno 27.03.2018].
URL https://learnopengl.com/Advanced-Lighting/Advanced-Lighting

[5] Vries, J.: *Learn OpenGL - Phong reflection model*. [Online; navštíveno 27.03.2018].
URL https://learnopengl.com/Lighting/Basic-Lighting

[6] Vries, J.: *Learn OpenGL - Shadow-Mapping*. [Online; navštíveno 8.04.2018].
URL https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping