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

ROZŠÍŘENÍ PROGRAMU VRUT O ZOBRAZOVACÍ PLUGIN V ROZHRANÍ VULKAN

VULKAN RENDERING PLUGIN FOR VRUT SOFTWARE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MARTIN KÁČERIK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JOZEF KOBRTEK**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/19029/2016/xkacer00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Káčerik Martin, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **Rozšíření programu VRUT o zobrazovací plugin v rozhraní Vulkan**
Vulkan Rendering Plugin for VRUT Software
Kategorie: Počítačová grafika

Pokyny:

1. Seznamte se s rozhraním Vulkan pro tvorbu 3D aplikací.
2. Seznamte se s programem VRUT, jeho architekturou a možnostmi rozšíření.
3. Implementujte odobný plugin jako RenderGL3 v rozhraní Vulkan.
4. Zmeřte výkon vašeho řešení v porovnání s ostatními zobrazovacími pluginy programu VRUT.
5. Zhodnoťte dosažené výsledky.

Literatura:

- https://www.khronos.org/vulkan/

Při obhajobě semestrální části projektu je požadováno:

- Bodu 1 a 2.

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

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% 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: **Kobrtek Jozef, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

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

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Táto diplomová práca sa zaoberá zobrazovaním trojrozmerných CAD modelov v reálnom čase pomocou rozhrania Vulkan. Zároveň skúma možnosti prepojenia tohoto rozhrania s aplikáciou VRUT, komplexným riešením pre zobrazovanie modelov vyvíjaným v spoločnosti ŠKODA AUTO a.s. Predstavuje návrh takéhoto spojenia vo forme zobrazovacieho modulu založeného na rozhraní Vulkan v prostredí aplikácie VRUT. Výkon navrhnutého modulu je porovnaný s iným dostupnými modulom, založeným na iných technológiach zobrazovania.

Abstract

The master's thesis submitted deals with realtime rendering of three-dimensional CAD data using Vulkan API. The thesis also covers possibilities of connecting the API with VRUT, complex solution for renderig developed by ŠKODA AUTO a.s. Design of such connection is presented in form of Vulkan rendering plugin for VRUT application. Performance of the designed module is compared with another rendering module, based on different rendering technologies.

Kľúčové slová

3D grafika, zobrazovanie v reálnom čase, vizualizácia CAD dát, Vulkan, postOpenGL, VRUT

Keywords

3D graphics, realtime rendering, CAD data visualization, Vulkan, postOpenGL, VRUT

Citácia

KÁČERIK, Martin. *Rozšíření programu VRUT o zobrazovací plugin v rozhraní Vulkan.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Kobrtek Jozef.

Rozšíření programu VRUT o zobrazovací plugin v rozhraní Vulkan

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Jozefa Kobrtka. Ďalšie informácie mi poskytol pán Mgr. Antonín Míšek, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Martin Káčerik
24. mája 2017

Poďakovanie

Chcel by som sa poďakovať svojmu vedúcemu pánu Kobrtkovi a svojmu konzultantovi zo spoločnosti ŠKODA AUTO a.s. pánu Míškovi za všetku poskytnutú odbornú pomoc.

Obsah

Kapitola 1

Úvod

Súčasný priemyselný dizajn je jedna z mnohých oblastí, ktorá výrazne profituje zo stále rastúceho výkonu dostupného grafického hardvéru. S rastúcim výkonom je možné nie len zahrnúť do výpočtu čím ďalej tým viac fyzikálnych podrobností pre vernú reprezentáciu sveta, ale prichádzajú s ním aj nové zobrazovacie zariadenia. Virtuálna realita sa stala fenoménom nie len v oblasti automobilového priemyslu, a je to hlavný dôvod, prečo aj v tomto prostredí rastie dopyt po vysoko výkonných zobrazovacích aplikáciach.

Spoločnosť ŠKODA AUTO a.s (ďalej ŠKODA). vlastní platformu, ktorá do značnej miery spĺňa jej vnútorné požiadavky v tejto oblasti. Pre dosiahnutie čo možno najlepších výsledkov je však ochotná investovať do experimentov v oblasti najmodernejších technológií, a tak táto práca vzniká ako súčasť takého experimentu.

Cieľom práce je vytvoriť program schopný vizualizovať trojrozmerné CAD dáta v reálnom čase. Tento program bude figurovať ako samostatný plugin pre aplikáciu VRUT a na zobrazovanie bude využívať aplikačné programové rozhranie Vulkan.

K naplneniu stanoveného cieľa je nutné vykonať viacero krokov, ktoré táto práca v logickom slede dokumentuje.

V nasledujúcej kapitole je teoretickej analýze podrobené nové API Vulkan a aplikácia VRUT, vyvíjaná a využívaná v rámci ŠKODA. Sú zhodnotené základné vlastnosti oboch technológií a skúmajú sa možnosti ich spolupráce.

Tretia kapitola predstavuje konkrétne požiadavky na riešenie a predkládá návrh aplikácie vychádzajúci z vykonanej analýzy. Návrh delí aplikáciu na jednotlivé podčasti a postupne sa im podrobne venuje. Na záver je popísaný predpokladaný beh programu, ktorý spája predstavené podčasti do výsledného celku.

Po spracovaní návrhu bolo možné pristúpiť k jeho realizácii, čo popisuje kapitola s poradovým číslom štyri. V jej prvej časti sú zhrnuté všetky použité technológie a dôležité programátorské techniky, zatiaľ čo jej druhá časť sa venuje testovaniu výsledného programu, jeho porovnaniu s alternatívami a komentuje získané výsledky.

Posledná kapitola obsahuje zhodnotenie práce ako celku a predkladá myšlienky na jej ďalšie použitie alebo rozšírenie.

Kapitola 2

Teoretický úvod

V tejto kapitole najskôr predstavíme rozhranie Vulkan — multiplatformné API pre grafický a výpočtový hardvér, v súčasnosti vyvíjané pod taktovkou konzorcia Khronos.

V jej druhej časti sa budeme venovať nástroju VRUT, ktorý je v spoločnosti ŠKODA používaný na vizualizáciu trojrozmerných CAD modelov.

2.1 Vulkan

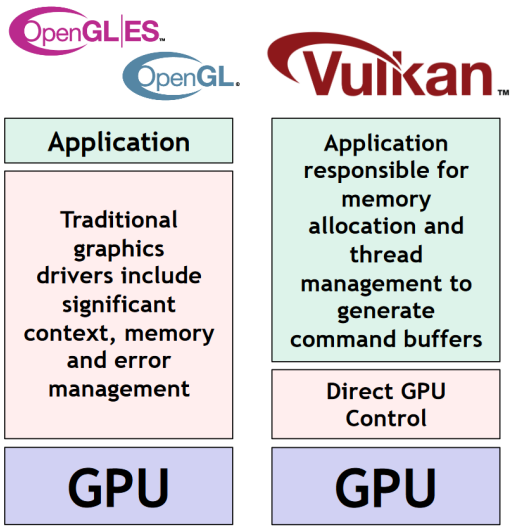
Ako bolo zmienené, Vulkan je nové multiplatformné rozhranie vytvorené pre potreby moderného hardvéru. Bližšie popíšeme hlavné koncepty jeho funkcionality a porovnáme ho s jeho rozšíreným a známym predkom, OpenGL.

2.1.1 História

Korene rozhrania Vulkan sa viažu na API nesúce meno Mantle. Spoločnosť AMD začala svoje proprietálne API Mantle vyvíjať v roku 2013 ako tzv. *low-overhead* renderovacie API so zameraním na 3D videohry. S príchodom roku 2015 sa však medzi iným na trh dostal produkt DirectX 12 spoločnosti Microsoft a AMD oznámilo zmenu svojej stratégie. Mantle sa stal základom pre novovznikajúci otvorený štandard ohlásený konzorciom Khronos na konferencii GDC s názvom Vulkan [10]. Šestnásteho februára roku 2016 vychádza špecifikácia Vulkanu verzia 1.0 a spolu s ňou aj open-source Vulkan SDK [2]. V súčasnosti je k dispozícii verzia špecifikácie 1.0.50.

2.1.2 Základné informácie

Pred zverejnením názvu Vulkan sa novovznikajúci štandard označoval ako *glNext*, čo v zásade vyjadrovalo jeho ambíciu stať sa nasledovníkom OpenGL. OpenGL má za sebou 25 rokov vývoja a jeho architektúra pracujúca na vyššej úrovni abstrakcie mu neumožňuje naplno využiť potenciál moderného hardvéru. Obrázok 2.1 znázorňuje, v čom spočíva potenciál Vulkanu — značná časť zodpovednosti sa presúva z rúk ovládača do rúk programátora. Znížením úrovne abstrakcie má programátor, za cenu zvýšenia nárokov na jeho prácu, výrazne vyššiu kontrolu nad aplikáciou samotnou.



Obr. 2.1: Pomer zodpovednosti v rukách ovládača a programátora v rozhraniach OpenGL a Vulkan [1].

Napriek tomu Vulkan zdieľa s OpenGL niekoľko zásadných vlastností [2]:

- je nezávislý od použitej platformy

  podpora na Windows, Linux, Android, Tizen, iOS (a macOS)

- je nezávislý od výrobcu grafického hardvéru

  podpora u AMD, NVIDIA, Intel, Imagination, Qualcomm

Väčšina jeho kľúčových vlastností sa však od OpenGL líši:

- nízka pridaná záťaž spôsobená ovládačom [1]
- kód shaderu je vo formáte SPIR-V a je predkompilovaný [1]
- natívna podpora viacvláknového prístupu, dobrá škálovateľnosť [5]

Súradnicový systém

Ďalšou dôležitou novinkou sú zmeny v súradnicovom systéme, ktoré je nevyhnutné zohľadniť najmä ak preberáme na zobrazenie scény optimalizované pre rozhranie OpenGL. Všetky zmeny sú popísané v špecifikácii (viď. [3]) a sú prehľadne zhrnuté v článku [11]. Z hľadiska prípravy dát je zaujímavá hlavne zmena orientácie normalizovaného súradnicového systému zariadenia (ang. *normalized device coordinates*, známe aj ako *NDC*).

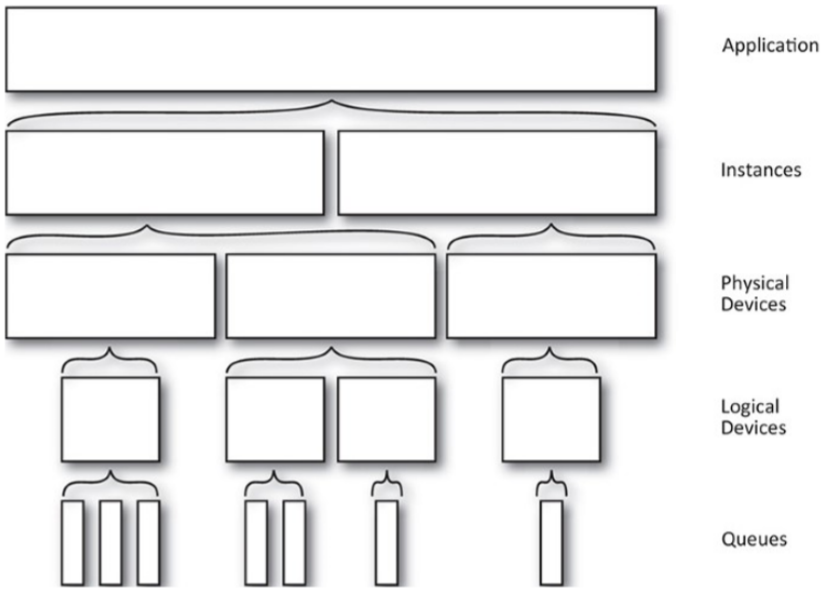
Súradnicový systém vo Vulkane predstavuje os Y orientovanú smerom nadol, čo efektívne znamená pretočenie implicitného ľavotočivého súradnicového systému z OpenGL na nový pravotočivý systém.

Preto je nevyhnutné pre všetky 3D súradnice objektov scény vyjadrenej v ľavotočivom priestore otočiť ypsilonové súradnice ich vynásobením číslom -1.

2.1.3 Model fungovania

Vulkan, na rozdiel od OpenGL, nepozná globálny stav a s ním zviazaný kontext. Pre udržanie informácií o svojom stave za behu aplikácie zavádza objektovo založený hierarchický model. Na najvyššej úrovni tohoto modelu sa nachádza inštancia (ang. *instance*), ktorá zastrešuje všetky výpočtové zariadenia s podporou Vulkanu pripojené k systému. Každý kus takéhoto hardvéru je reprezentovaný ako jedno fyzické zariadenie (ang. *physical device*). Fyzické zariadenie nesie so sebou informácie o svojich výpočtových možnostiach a ponúka jednu alebo viac front (ang. *queue*) umožňujúcu vykonať istý typ práce na zariadení.

Inštancia umožňuje nad fyzickými zariadeniami vytvoriť logické zariadenia (ang. *logical device*). Logické zariadenie je softvérová abstrakcia reprezentujúca rezervované zdroje (fronty) na fyzickom zariadení a slúži ako komunikačné rozhranie medzi aplikáciou a týmto fyzickým zariadením. Popísané hierachické vzťahy sú vizualizované na obrázku 2.2.



Obr. 2.2: Hierarchia objektového modelu rozhrania Vulkan [9].

Fronty, ktoré sú na zariadení k dispozícii, sa delia do rodín (ang. *families*). Každá rodina podporuje jednu alebo viac typov funkcionality a obsahuje jednu alebo viac podobných front. Špecifikácia definuje tieto 4 typy funkcionality: grafická, výpočtová, transferová a manažment riedkej pamäte. Fronty v rámci jednej rodiny sú považnované za kompatibilné (ang. *compatible queues*) a práca určená pre túto rodinu môže byť vykonaná na ktorejkoľvek z nich. Je však možné, že rôzne rodiny na jednom zariadení podporujú rovnakú funkcionalitu — fronty v nich však nie sú považované za kompatibilné.

Aplikácia využívajúca Vulkan následne riadi sadu zariadení pomocou tzv. *command buffers*, do ktorých sú pomocou Vulkan volaní nahraté príkazy pre jednotlivé zariadenia. Po skonštruovaní môže byť command buffer raz alebo opakovane zaslaný do fronty na vykonanie [3][9].

2.1.4 Správa pamäte

Pamäť je z pohľadu Vulkanu rozdelená na dve časti:

- hostiteľskú pamäť (ang. *host memory*)
- a pamäť zariadenia (ang. *device memory*)

Toto delenie je pomerne ľahko uchopiteľné z pohľadu typickej desktopovej architektúry, kde hostiteľská pamäť reprezentuje RAM pamäť fyzicky pripojenú k CPU a pamäť zariadenia je samostatná pamäť pripojená ku GPU. Iné, najmä mobilné, architektúry však môžu k pamäti pristupovať iným spôsobom a fyzicky môžu mať k dispozícii len jeden typ pamäte slúžiaci pre všetky účely.

V hostiteľskej pamäti si Vulkan udržiava vnútorné informácie o stave svojich objektov. Alokačný model použitý pre tento typ pamäte je súčasťou implementácie Vulkanu, je však možné nahradiť ho modelom vlastným.

Do pamäte zariadenia sa ukladajú dáta potrebné pre beh aplikácie samotnej. Za jej alokáciu preberá plnú zodpovednosť programátor [3][4].

Správa pamäte je oblasť, do ktorej sa vo veľkej miere premieta nízkoúrovňovosť celého rozhrania. V rozhraniach pracujúcich na vyššej úrovni, ako je napríklad OpenGL, neexistuje požiadavka na vytvorenie subsystému správy pamäte zariadenia. Túto správu zaobstará ovládač, ktorý však nedokáže určiť účelnosť dát tak spoľahlivo, ako programátor konkrétnej aplikácie. Preto sú často volené neoptimálne alokačné stratégie, čo má za následok plytvanie dostupnými zdrojmi. Pokiaľ však tento subsystém vytvorí priamo niekto, kto vie presne určiť dátový tok aplikácie, je schopný vytvoriť ideálny scenár pre správu pamäťových zdrojov (viď. 2.1.7) [9].

Pamäť zariadenia

Vulkan operuje nad dátami a všetky jeho súčasti sú tomu podriadené. Problematike pamäte zariadenia je v literatúre [9] aj v špecifikácii Vulkanu [3] venovaných niekoľko desiatok strán. Pre účely návrhu sú na tomto mieste zhrnuté základy, pre hlbšie porozumenie je odporúčané preštudovať uvedené zdroje.

Za pamäť zariadenia obecne považujeme pamäť dostupnú alebo mapovateľnú zo zariadenia, pričom táto pamäť nemusí byť priamo fyzicky spojená so zariadením. Táto pamäť je k dispozícii vo forme jednej alebo viacerých hromád (ang. *memory heaps*). Ich počet a charakteristiky sa líšia podľa typu zariadenia a sú vyjadrené sadou príznakov. Zariadenia, na ktorých sa predpokladá beh modulu VRender, budú typicky obsahovať minimálne jednu hromadu s príznakom *device local*, ktorá je hierarchicky bližšie k jadru GPU a poskytuje vyšší výkon, ako aj minimálne jednu hromadu s príznakom *host visible*, ktorá je síce pomalšia, ale je mapovateľná zo strany CPU. Z dostupných hromád môžeme pre účely aplikácie alokovať pamäť potrebnú pre pamäťové zdroje.

Keďže GPU, ako je obecne známe, pracuje na vysoko paralelnej úrovni, pre optimálny prístup k dátam vyžaduje zarovnané bloky pamäte. Veľkosť takéhoto bloku sa líši, je daná konkrétnym zariadením a určeným použitím daného bloku. Rozhranie Vulkan umožňuje svojimi volaniami zistiť konkrétne hodnoty pre použité fyzické zariadenie.

Pamäťové zdroje

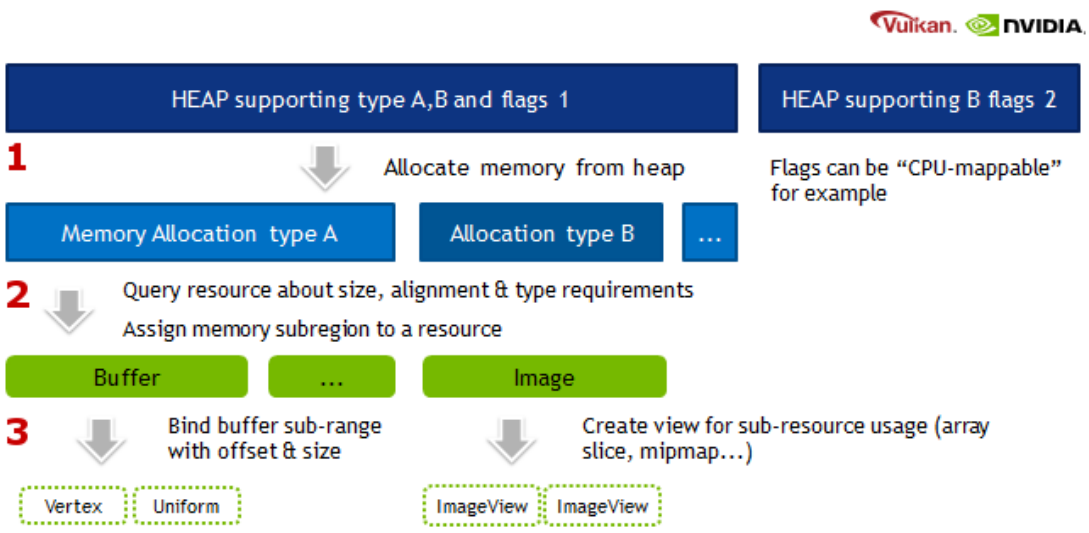
Pamäťové zdroje sú v prostredí Vulkan reprezentované dvoma fundamentálnymi typmi:

- *Buffer*
  Špecifikuje lineárny blok dát, hodí sa pre uchovanie akéhokoľvek typu informácie. K dátam je možné pristupovať priamo.

- *Image*
  Štrukturovaný, multidimenzionálny typ, podporuje sadu špecializovaných operácií pre čítanie a zápis dát. K dátam teda nie je možné pristupovať priamo, ale pomocou tzv. pohľadov (ang. *image views*). Pohľad určuje požadovanú interpretáciu dát.

  Pre každý image navyše existuje *image layout*, ktorý definuje aktuálne fyzické usporiadanie dát v pamäti. Je tak možné meniť *layout* jedného *image*, a tým umožňovať optimálnu prácu s pamäťou pre rôzne operácie nad ním.

Hierarchia alokácií



Obr. 2.3: Hierarchia pamäte zariadenia v prostredí Vulkan [4].

Obecne platí, že z jednej hromady môžeme alokovať viacero pamäťových blokov, jeden blok môže pomocou pamäťového offsetu slúžiť viacerým pamäťovým zdrojom a jeden skutočný zdroj môže pomocou pamäťového offsetu reprezentovať viacero virtuálnych zdrojov. Túto hierarchiu prehľadne znázorňuje obrázok 2.3.

Hierarchický prístup k správe pamäte je nie len možný, ale aj odporúčaný postup (viď. [4]). Pri správnom návrhu totiž umožňuje efektívne využívať možnosti kešovania pamäte, keď logicky súvisiace, ale nezávislé bloky pamäte (typicky napríklad vrcholy a indexy vrcholov pri indexovanom vykresľovaní) budú v pamäti riadene uložené fyzicky vedľa seba.

2.1.5 Multithreading

Podpora práce z viacerých vlákien hostiteľa je jedna z kľúčových súčastí dizajnu rozhrania Vulkan. Na rozdiel od OpenGL 4 (ktorý bol pôvodne navrhnutý pre jednojadrové archi-

tektúry na strane hostiteľa a až následne rozšírený o podporu viacerých vlákien) poskytuje Vulkan aj solídnu škálovateľnosť výkonu v závislosti od počtu použitých vlákien [5].

Všetky Vulkan príkazy podporujú súčasné volanie z viacerých vlákien, niektoré parametre alebo ich časti však môžu vyžadovať samostatnú synchronizáciu prístupu. Vulkan z pohľadu synchronizácie rozlišuje tri typy objektov:

- *immutable* objekty
- *mutable* objekty synchronizované interne
- *mutable* objekty synchronizované externe

Pomerne veľké množstvo objektov je tzv. *immutable* — po vytvorení v nich už nemôžu nastať žiadne zmeny. Tieto objekty nevyžadujú externú synchronizáciu. Je však nutné zaručiť, že objekt nebude zničený, zatiaľ čo s ním iné vlákno pracuje.

Tzv. *mutable* objekty s internou synchronizáciou (teda synchronizáciou vnútri samotného príkazu, v rámci implementácie Vulkanu) sa vyskytujú zriedkavo. Taktiež nevyžadujú externú synchronizáciu. Zvyšok objektov (zoznam konkrétnych parametrov konkrétnych volaní je k dispozícii v špecifikácii) vyžaduje externú synchronizáciu v réžii programátora [3].

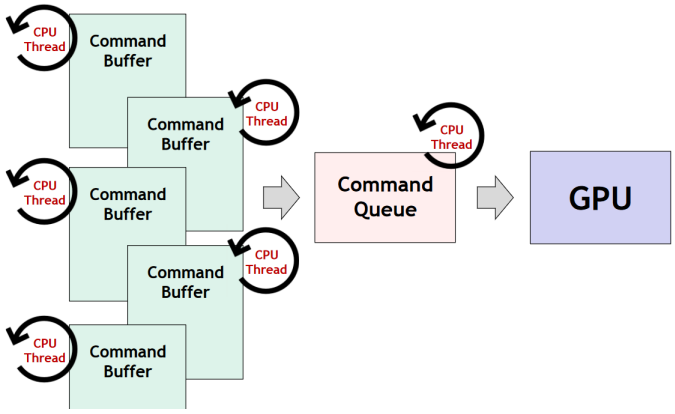
Je možné paralelne budovať niekoľko command bufferov z viacerých vlákien aplikácie, tak ako to ilustruje obrázok 2.4. Následná synchronizácia na úrovni hostiteľ–zariadenie je taktiež v réžii programátora [9].

2.1.6 Rozšírenia a vrstvy

Rozšírenia (ang. *extensions*) a vrstvy (ang. *layers*) dopĺňajú základnú funkcionalitu jadra o voliteľné časti.

Rozšírenia

Koncept rozšírení je známy už z prostredia OpenGL. Vo forme rozšírení štandardne prichádzajú do API inovácie a novinky, ktoré sa následne, pokiaľ ich prax dostatočne preverí, môžu pretaviť do súčastí jadra. Rozlišujeme rozšírenia na úrovni inštancie a na úrovni zariadenia, ktoré rozširirujú funkcionalitu jadra na adekvátnej úrovni. Rozšírenia je nutné povoliť pri vytvorení danej inštancie alebo daného zariadenia v aplikácii a ďalej sa k nim



Obr. 2.4: Paralelné budovanie viacerých command bufferov [1].

pristupuje už ako ku štandardnej súčasti API. Rozšírenia môžu obsahovať nové príkazy či štruktúry.

Ako samostnané rozšírenia v rozhraní Vulkan stoja aj viaceré platformne závislé súčasti, spojené najmä s prezentáciou výsledkov na obrazovku. Prezentácia je tesne zviazaná so systémom okien, a ten je vždy súčasťou konkrétneho operačného systému.

Keďže rozšírenia môžu nie len pridávať novú funkcionalitu, ale aj meniť správanie súčasného stavu, ich súčasťou by mali byť aj príslušné vrstvy [3].

Vrstvy

Počas behu aplikácie má prioritu efektivita budovania command bufferov a ich odosielanie na spracovanie. Samotné jadro tak neuchováva a neposkytuje prakticky žiadne ďalšie informácie, potrebné hlavne pri vývoji aplikácie, umožňujúce jej ladenie či profilovanie.

Systém vrstiev umožňuje voliteľne zapínať a vypínať prídavnú funkcionalitu, a to vo forme rozšírenia už existujúcich volaní. Týmto spôsobom nie je možné pridávať žiadne nové funkcie. Implementácia vrstiev v zásade zodpovedá návrhovému vzoru dekorátor. Tento systém je typicky použitý k rozšíreniu jadra o prídavné informácie k jednotlivým volaniam, a to pre umožnenie efektívneho ladenia, profilovania či validácie výsledkov.

Opäť rozlišujeme vrstvy na úrovni inštancie a na úrovni zariadenia a je nutné ich povoliť pri vzniku týchto objektov [3][9].

2.1.7 Správa zdrojov, rendering a prezentácia

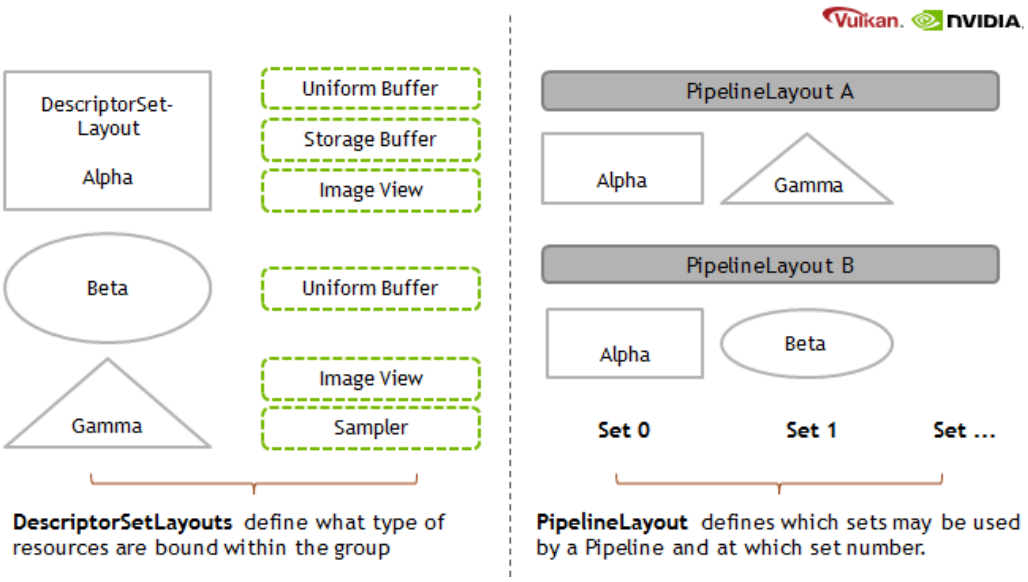
Proces zobrazovania v rozhraní Vulkan je pomerne dlhý a zúčastňuje sa ho veľké množstvo Vulkan objektov. Tie nadôležitejšie z hľadiska budúceho návrhu sú stručne popísané na tomto mieste. Ich popis vychádza prevažne zo štúdia [3] a [9].

Descriptor Sets

Pre efektívne použitie dát na grafickej karte je nutné tieto dáta nielen fyzicky umiestniť na zariadenie, ale poskytnúť aj vhodné rozhranie, cez ktoré k nim budú jednotlivé shadery vykonávané v rámci pipeline (viď. 2.1.8) pristupovať. Obecne sa dáta shaderu sprístupnia pripojením (ang. *binding*) požadovaného pamäťového bloku. Táto operácia je však relatívne drahá a je preto nevyhnutné v maximálnej možnej miere znížiť jej výskyt v programe. Ako riešenie zavádza API Vulkan systém pripájania po skupinách, nazývaných *descriptor sets*.

Každý descriptor set má presne definované rozhranie, ktoré určuje, aké zdroje je možné pomocou tohoto setu pripojiť. Rozhranie sa nazýva *descriptor set layout* a nesie tiež informáciu o tom, v akej časti pipeline (pre ktorý konkrétny shader) má byť ktorý pamäťový zdroj prístupný.

Podstatou rozdelenia zdrojov do descriptor setov je fakt, že pokiaľ v nejakej časi pipeline pripojíme jeden set, tento set ostane pripojený pre všetky ďalšie volania danej časti, čím sa redukuje počet pripojení v kritickej renderovacej slučke. Je potom zrejmé, že hlavnou prioritou pri navrhovaní štruktúry descriptor setov v aplikácií je správne určenie frekvencie zmeny dát za behu vykresľovania. Preto budú typicky dáta s vysokou frekvenciou zmeny (napr. transformačné matice, ktoré sa menia pre každý jeden renderovaný objekt) umiestnené do jedného setu, zatiaľ čo dáta s nižšou frekvenciou zmien (napr. materiál, ktorý sa pri vhodnom zoradení objektov v scéne mení každých N renderovaných objektov) budú umiestné do iného setu.



Obr. 2.5: Vzťah rozhraní systému descriptor setov pre ich použitie v pipeline [6].

Aby bolo možné v pipeline pristupovať paralelne k viacerým setom, je nutné ich definovať v rozhraní pipeline, tzv. *pipeline layout*. Vzťah spomínaných *layouts* je zhrnutý v obrázku 2.5.

Uniform data binding

Na obrázku 2.5 je možné pozorovať, že súčasťou descriptor setu môžu byť aj uniformné dáta, ktoré môžme chápať ako parametre pre program shaderu. Na rozdiel od iných typov existuje pre uniformné dáta niekoľko spôsobov, ako ich pre shader sprístupniť. Tak, ako je zhrnuté v článku [6], je možné využiť znalosť o vyvíjanom programe a použiť najvhodnejšiu z nasledujúcich možností:

- *Uniform buffer binding*
  Samostatné uniformné dáta sú uložené do samostatných uniformných bufferov, každý buffer je samostatne pripojený pre použitie v shaderoch.
- *Uniform buffer dynamic binding*
  Tento spôsob umožňuje nahrať viaceré dáta do jedného uniformného bufferu, ktorý sa pripojí jeden krát a následne sú mu dynamicky posielané offsety ukazujúce na požadované dáta. Je nevyhnutné zachovať správne zarovnanie pamäte, aj za cenu nevyužitého miesta v každom bloku.
- *Push constants*
  Tento spôsob nepoužíva k svojej realizácií descriptor sety, ale umožňuje dáta uložiť do command bufferu a tak ich za behu využiť na GPU. Za špecifických okolností bude tento spôsob najrýchlejší.

Render pass

Výsledný výstupný obraz vzniká jedným alebo viacerými prechodmi scénou a grafickou pipeline, kde každý takýto prechod môže pripravovať inú časť scény alebo aplikovať rôzne

efekty. Súbor týchto prechodov nazývame *renderpass*, samostatný prechod z tohoto súboru potom nazývame *subpass*. Vytvorením Vulkan objektu renderpass zadefinujeme celý proces vzniku výsledného obrazu, pričom tento môže byť jednoduchý, jednokrokový renderpass bez dátových závislostí, ale aj komplikovaný renderpass s množstvom na sebe závislých subpassov.

Renderpass a dátové závislosti jednotlivých častí je možné modelovať ako orientovaný acyklický graf. Tento graf sa s použitím pokročilých renderovacích techník (napr. *deferred shading*, tvorba pokročilých tieňov, priehľadnosť, ...) značne rozrastá.

Framebuffer

Framebuffer je časť pamäte, reprezentovaná rovnomenným Vulkan objektom *framebuffer*. Tento obsahuje sadu obrázkov (reprezentovaných práve pamäťovým zdrojom image z podkapitoly 2.1.4). Do týchto obrázkov sú počas renderovania ukladané potrebné dáta, preto musi zloženie framebufferu zodpovedať nadefinovaným vzťahom a dátovým závislostiam použitého renderpassu.

Grafická pipeline

Po stanovení formy a vytvorení dátového priestoru pre vykresľovanie zostáva stanoviť priebeh samotného renderingu. Požadovanú funkcionalitu dosiahneme implementáciou celej grafickej pipeline z prílohy B. Je pravda, že pre jej minimalistický priebeh by stačilo povoliť len *vertex shader*, pre reálnu aplikáciu sa však požaduje nastavenie alebo implementácia väčšiny krokov.

Pipeline je možné vytvoriť ako jeden statický objekt, ale množstvo objektov v scéne môže požadovať rozdielne nastavenia častí tejto pipeline. Vytvorenie a prepínanie celej pipeline za behu aplikácie je možné, ale je to drahá operácia. Preto sa dajú niektoré aspekty pipeline nastaviť ako dynamické a za behu nastavovať požadované hodnoty len im.

Pipeline je pevne zviazaná s rozlíšením, v akom renderuje výsledný obraz, a pri jeho zmene (typicky pri zmene veľkosti okna) sa teda musí vybudovať znovu. Ako bolo spomínané, jedná sa o drahú operáciu. Pre zvýšenie efektivity tohoto procesu sa zavádza objekt *pipeline cache*, ktorý umožňuje uchovávať a znovupoužívať časti existujúcich pipeline pri vytváraní potrebných nových.

Swapchain

Plocha, na ktorú sa zobrazí výsledok, je reprezentovaná Vulkan objektom surface z rovnomenného rozšírenia. Podmienky jeho vytvorenia sú popísané v podkapitole 3.2.1. Za predpokladu, že sa podarilo vytvoriť požadovanú plochu, k úspešnému zobrazeniu ešte potrebujeme získať sadu špeciálnych obrázkov, ktoré sú schopné poskytnúť svoje dáta tejto ploche. Takéto obrázky sú však obdobne ako plocha v okne úzko spojené s okenným systémom. Na ich získanie a správu sa dá využiť Vulkan objekt *swapchain*.

Swapchain zaobstará od operačného (okenného) systému jeden alebo viac obrázkov priamo prezentovateľných na surface. Ich počet závisí od použitého spôsobu bufferovania videopamäte (none, double-buffering, triple-buffering). Implementácia takéhoto bufferovania je tiež úzko zviazaná s použitým prezentačným módom, ktorý riadi synchronizáciu zobrazovania s okenným systémom (a skrz neho priamo s fyzickým zobrazovacím zariadením).

Podľa špecifikácie (viď. [3]) swapchain ponúka nasledujúce módy:

- *Immediate*
  Prezentácia obrazu prebehne čo najskôr po zaslaní obrázku do fronty zariadenia. Výsledkom je maximálne možné FPS za cenu možného výskytu artefaktu roztrhnutia (ang. *tearing*) obrazu (vodorovné posunutie v obraze v dôsledku zobrazovania infromácií patriacich rôznym obrázkom).

- *Mailbox*
  Obrázok zaslaný do fronty je označený ako čakajúci a zobrazí sa na požiadanie okenného systému. Pokiaľ počas čakania na zobrazenie príde do fronty novší obraz, nahradí už čakajúci a ten je zmazaný. To umožňuje efektívne využiť tento mód pri implementácii triple-bufferingu.

- *FIFO*
  Obrázky prichádzajúce na prezentáciu sú ukladané v internej FIFO fronte, odkiaľ sú postupne v pravidelnom intervale (typicky vo frekvencii vertikálnej synchronizácie) zobrazované. V prípade zaplnenia fronty aplikácia čaká.

- *Relaxed FIFO*
  Upravuje správanie sa fronty z predchádzajúceho prípadu tak, že pokiaľ je fronta prázdna a požaduje sa od nej obrázok, tak novoprichádzajúci obraz nie je zaradený na čakanie na pravidelný interval, ale je poskytnutý na zobrazenie okamžite.

Keďže swapchain spravuje obrázky, je zodpovedný aj za ich rozlíšenie. Z toho priamo vyplýva, že pri zmene rozlíšenia (typicky pri zmene veľkosti okna použitého pre renderovanie) je nutné celý swapchain vybudovať znova so správnymi parametrami. Táto operácia je pomerne drahá, ale keďže nový swapchain bude s pôvodným zdieľať väčšinu nastavení, je možné použiť určitú formu dedičnosti a celý proces značne urýchliť.

2.1.8 Programátorský pohľad

Vulkan je z programátorského hľadiska často opisovaný slovom *verbose*, čo vyjadruje nutnosť veľkého množstva napísaného kódu pre každý požadovaný detail implementácie.

Z hľadiska podpory renderovania je dôležitá pracovná *pipeline* rozhrania Vulkan, najmä jej grafická (väčšinová) časť. Jej implementácia by spolu s manažmentom pamäte tvorila dominantnú časť renderovacej aplikácie. Táto pipeline sa myšlienkovo v zásade neodlišuje od pipeline použitej v rozhraní OpenGL 4 a je zobrazená v prílohe B.

Spolu so špecifikáciou vyšlo aj oficiálne SDK, a to v réžii spoločnosti LunarG. Medzi jeho súčasti patrí:

- Vulkan loader
- implementácia ladiacich vrstiev známa ako *validation layers*
- nástroje spojené so štandardom SPIR-V (napr. prekladač z jazyka GLSL)
- Vulkan runtime knižnice
- dokumentácia, demá a príklady použitia

Verzia SDK vždy číselne korešponduje s verziou špecifikácie. Obsahuje hlavičkové súbory jazyka C, od verzie 1.0.24 obsahuje aj hlavičkové súbory pre jazyk C++, pre prekladače podporujúce štandard C++11 [2].

Jeho súčasťou samozrejme nie je samotný Vulkan ovládač — ten vzniká v réžii jednotlivých výrobcov hardvéru.

2.2 VRUT

Názov aplikácie VRUT je skratkou z anglického *Virtual Reality Universal Toolkit*. Z názvu samotného je možné určiť aj jej účel — jedná sa o komplexný multiplatformný nástroj na editáciu a vizualizáciu trojrozmerných dát. Pri jeho vývoji v súčasnosti spolupracujú ŠKODA AUTO a.s., České vysoké učení technické v Praze, Masarykova univerzita, Mendelova univerzita v Brně, Západočeská univerzita v Plzni, Vysoká škola báňská Ostrava a v súčasnosti už aj Vysoké učení technické v Brně. Je nasadená v produkčnom prostredí viacerých oddelení spoločnosti ŠKODA a taktiež je prístupná pre školy na výuku a vlastné experimenty [8].

2.2.1 História

Základ aplikácie začal vznikať v roku 2008 v rámci diplomovej práce Václava Kybu na Fakulte elektrotechnickej Českého vysokého učení technického v Praze. V rámci tejto práce vznikol model jadra VRUTu tak ako ho poznáme doteraz a spolu s ním aj niekoľko základných modulov (viď. 2.2.2) [7]. Postupom času pribúdali ďalšie moduly, napríklad podpora prostredia pre virtuálnu realitu CAVE či podpora behu na HPC clusteri spoločnosti.

2.2.2 Architektúra

VRUT je navrhnutý ako modulárna aplikácia, preto môžme o jej funkčných častiach hovoriť ako o *hlavnej aplikácii* (alebo ekvivalentne *jadro*) a *moduloch*. Vzhľadom na stanovené požiadavky je jadro v rámci možností minimálne a z užívateľského pohľadu neposkytuje žiadne možnosti. Väčšina funkcionality je zabezpečená práve v samostatne stojacich moduloch, ktoré je potrebné k jadru pripojiť [8].

Jadro

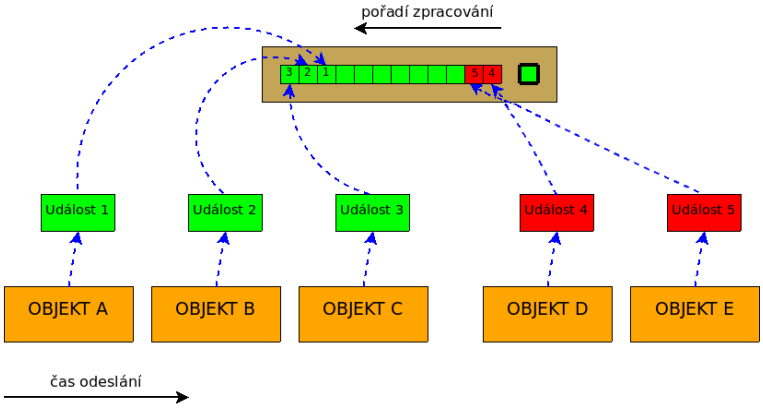
Medzi kľúčové funkcie jadra patria:

- manažment modulov
- manažment udalostí
- manažment grafických dát (graf scény)
- GUI, podpora pre ladenie

Správa modulov v sebe zahŕňa zaistenie dostupnosti a kompatibility modulov s jadrom a ich korektné spojenie s jadrom. Vzhľadom na existenciu viacerých kategórií modulov (viď. 2.2.2)) vyžadujúcich rôznu úroveň spolupráce s jadrom je k dispozícii samostatný správca pre každú kategóriu.

Keďže VRUT za behu tvorí isté množstvo viac alebo menej nezávislých objektov, ktoré potrebujú spolu komunikovať, súčasťou jadra je aj správca udalostí. Tento správca funguje

na bežných princípoch udalosťami riadeného systému a zastrešuje distribúciu udalostí v rámci celého systému. Na spracovanie prichádzajúcich udalostí využíva manažér frontu typu FIFO, v ktorej sa prichádzajúce udalosti radia podľa času príchodu. V rámci VRUTu je možné zasielať tzv. prioritné udalosti, ktoré sa vždy zaradia do fronty pred udalosti bez priority. Poradie spracovania je znázornené na obrázku 2.6.



Obr. 2.6: Príjem a spracovanie udalostí v čase. Bežné udalosti sú zelené, udalosti s prioritou sú červené [8].

Aplikácia VRUT obsahuje proprietálnu implementáciu grafu scény. Slúži ako štruktúrované úložisko grafických dát, ktoré sú spoločné pre všetky aktívne moduly aplikácie. S grafom scény aplikácia komunikuje pomocou udalostí, a to oboma smermi. Pomocou udalostí je možné správne synchronizovať prístupy do grafu scény, keďže sa jedná o zdieľanú dátovú oblasť. Napriek tomu, že existuje niekoľko otvorených implementácií grafu scény fungujúcich na veľmi podobných princípoch, z dôvodu vyššej kontroly nad projektom a zjednodušenia experimentovania bola pripravená implementácia vlastná [8].

Vrstva grafického užívateľského rozhrania je spojená s knižnicou wxWidgets, ktorá umožňuje podporu grafického rozhrania a ďalších platformne závislých súčastí aplikácie na rôznych operačných systémoch.

Architektúra jadra aplikácie VRUT je zachytená na obrázku 2.7

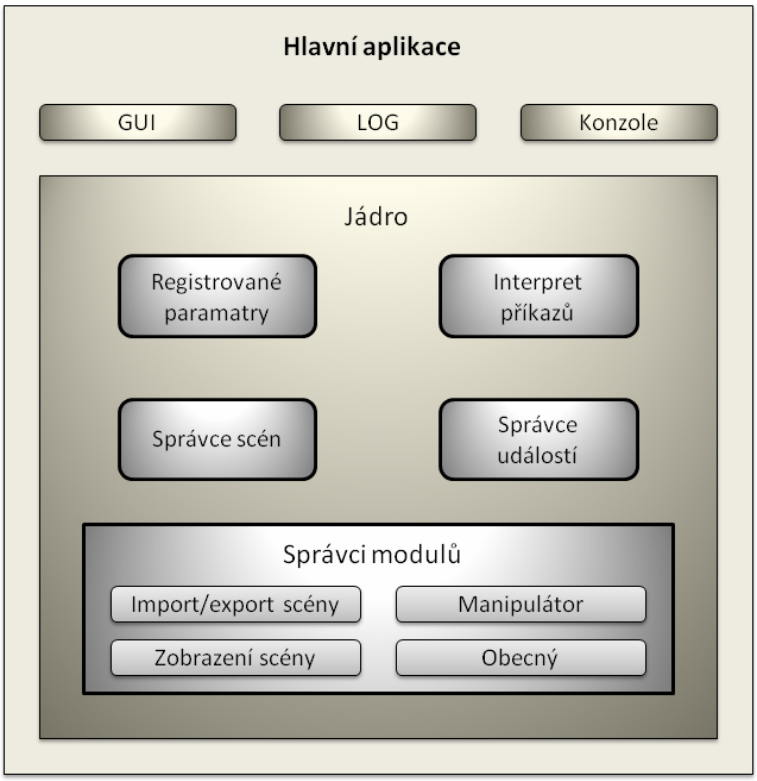
Moduly

Moduly poskytujú aplikácii jej požadovanú funkcionalitu pre spracovanie rôznych úloh. V jeden okamih môžu byť v aplikácii aktívne všetky dostupné moduly a od každého modulu môže súčasne bežať ľubovoľný (v rámci dostupných zdrojov) počet jeho inštancií. Každá takáto inštancia beží samostatne vo svojom vlastnom vlákne.

Komunikácia

Ako bolo spomínané v sekcii 2.2.2, jednotlivé moduly (respektíve ich rôzne inštancie) komunikujú s jadrom aj s ostatnými modulmi pomocou systému udalostí. Toto spojenie je z princípu asynchrónne a prebieha sprostredkovane skrz správcu komunikácie v jadre. V špeciálnych prípadoch je možné na komunikáciu s jadrom využiť aj synchrónne volania.

Zodpovednosťou každého modulu je spracovať akúkoľvek udalosť k nemu distribuovanú a zabezpečiť plynulý chod bez fatálnych následkov.



Obr. 2.7: Architektúra jadra aplikácie VRUT [8].

Kategórie modulov

Podľa požiadaviek na úroveň spolupráce s jadrom rozlišujeme nasledovné kategórie modulov dostupných pre aplikáciu VRUT [8]:

- Obecný modul – *Module*
  Modul bez špeciálnych požiadaviek, komunikuje štandardným rozhraním pre udalosti.
- Modul scény – *SceneModule*
  Modul vychádzajúci z obecného modulu, navyše definuje GUI prvok pre definíciu scény. Hodí sa pre rôzne operácie nad scénou, ako je napríklad detekcia kolízií.
- Modul pre import a export – *IOModule*
  Modul vychádzajúci z modulu scény, umožňuje import a export dát z daného formátu, napríklad OBJ alebo FHS.
- Zobrazovací modul – *RenderModule*
  Modul vychádzajúci z modulu scény, výstupom jeho činnosti je grafický obsah prezentovaný do GUI, ktorého správa je v kompetencii jadra VRUTu.
- Manipulátor – *ManipulatorModule*
  Modul vychádzajúci z modulu scény, navyše definuje GUI prvok pre výber manipulovaného uzlu. Hodí sa pre moduly interagujúce so scénou.
- Manipulátor kamery – *CameraModule*
  Modul vychádzajúci z modulu scény, navyše definuje GUI prvok pre výber renderovacieho okna, s ktorým je spojený. Umožňuje napríklad prechádzanie scénou.

2.2.3 Existujúce zobrazovacie moduly

V súčasnosti sú v aplikácii VRUT k dispozícii tieto plne funkčné zobrazovacie moduly [8]:

- *RenderGL*
  Je to pôvodný a najstarší zobrazovací modul. Je implementovaný pomocou OpenGL a zodpovedá špecifikácii verzie 2. Využíva fixnú grafickú pipeline.

- *RenderGL3*
  Novší modul, kde číslo 3 značí tretiu generáciu modulov založených na OpenGL. Tento modul už využíva programovateľnú grafickú pipeline a ďalšie moderné postupy predstavené v špecifikácii OpenGL 4. Plne nahradil svojho predchodcu, *RenderGL*.

- *RayTracer*
  Zobrazovací modul využívajúci metódu *ray tracing*, implementovanú na CPU. Napriek sade optimalizácií ako je progresívne vykresľovanie alebo manuálna vektorizácia výpočtovo náročných častí, je stále príliš pomalý, aby plne nahradil moduly založené na OpenGL. Pri riešení špecifických problémov je však nasadzovaný a preferovaný aj v jeho aktuálnej forme.

VRUT obsahuje podporu pre špecializovaný hardvér pre virtuálnu realitu, ako je CAVE alebo head-mounted displeje a každý spomínaný modul je schopný poskytnúť adekvátny požadovaný výstup pre daný typ zobrazovania. Táto podpora je kladená ako požiadavka aj na každý prípadný nový zobrazovací modul.

2.2.4 Možnosti rozšírenia

Z podkapitoly 2.2.2 je zrejmé, že pre obecné rozšírenie funkcionality aplikácie VRUT je možné pomerne priamočiarym spôsobom, a to implementáciou nového modulu, ktorý bude spĺňať požadované komunikačné rozhranie.

Pri rozšírení funkcionality o zobrazovací modul využívajúci rozhranie Vulkan predstavené v kapitole 2.1 však bohužiaľ dochádza ku konfliktu. Knižnica wxWidgets, pomocou ktorej je spracované grafické užívateľské rozhranie, a teda aj prezentácia výsledkov zobrazovacích modulov, rozhranie Vulkan nepodporuje a pravdepodobne v dohľadnej dobe ani podporovať nebude[1].

Je preto stále predmetom diskusie, ako tento typ komplikácií riešiť. Nedá sa vylúčiť, že kvôli nasadeniu takéhoto modulu bude nutné nie len pripraviť samotný modul, ale aj zasiahnuť do jadra VRUTu.

[1] https://forums.wxwidgets.org/viewtopic.php?t=42577

Kapitola 3

Návrh riešenia

Obsahom tejto kapitoly je predstavenie návrhu zobrazovacieho pluginu pre VRUT v rozhraní Vulkan. Určuje konkrétne rozhrania prepájajúce obe technológie predstavené v kapitole 2 a špecifikuje vnútornú architektúru modulu spolu s logickým odôvodnením tohoto členenia.

Pre modul bol v rámci zachovania konzistentnosti vnútorného názvoslovia aplikácie VRUT (s prihliadnutím na fonetický aspekt veci) určený názov *VRender*. Celý nasledujúci text bude preto ďalej pracovať s týmto názvom.

3.1 Požiadavky na riešenie

Modul VRender má v prvom rade slúžiť ako vizualizačný nástroj profesionálnych CAD dát. Pre zaručenie korektného výsledku sú na riešenie kladené viaceré požiadavky, ktoré môžeme na základe ich povahy rozdeliť na dve skupiny: požiadavky funkčné, definujúce úlohy, ktorých riešenie musí aplikácia poskytovať a požiadavky plynúce z povahy použitých technológií.

3.1.1 Funkčné požiadavky

Pre účely plnenia požadovaných funkcií musí výsledný modul VRender spĺňať nasledujúce požiadavky:

- načítanie dát z modulu scény (viď. sekcia 2.2.2) aplikácie VRUT
- rendering získaných dát v rozhraní Vulkan
- prezentácia výsledného renderu užívateľovi

Takto definované požiadavky triviálne popisujú základné prípady použitia plánovaného systému. Pre jednotlivé body zadania ďalej formulujeme detailnejší popis, z ktorého budeme vychádzať pri tvorbe návrhu.

Načítanie dát

Scéna v prostredí aplikácie VRUT je reprezentovaná acyklickým grafom, kde jednotlivé uzly zodpovedajú prvkom scény. Každý uzol obsahuje transformačnú maticu ovplyvňujúcu svojich potomkov, graf takto reflektuje transformačnú hierarchiu v scéne.

Dôležité sú najmä uzly kamier, uzly s geometriou a uzly s osvetlením. Okrem nich scéna obsahuje aj špeciálne entity — geometrie a materiály, ktoré obsahujú konkrétne dáta. Uzly s geometriou môžu na tieto entity odkazovať, pričom viacero uzlov môže odkazovať na rovnakú geometriu alebo materiál. Z vyššie uvedeného teda plynie, že modul VRender musí vedieť spracovať nasledovné dáta:

- Uzly v scéne
  Pre každý uzol je nevyhnutné spočítať výslednú transformačnú maticu a správne priradiť ním odkazované dáta (geometrie, materiály, osvetlenie alebo projekčnú maticu kamery).

- Geometrie
  Dáta jednotlivých geometrií, medzi ktoré patria dáta o vrcholoch, normály a textúrovacie súradnice.

- Materiály
  Dáta jednotlivých materiálov, medzi ktoré patria dáta o farbách a vlastnostiach materiálu, informácie o priehľadnosti, hĺbkovom teste a cesta k prípadným použitým textúram.

Rendering

Z pohľadu renderingu je kľúčová požiadavka na renderovanie náročných CAD scén vo vysokom rozlíšení v reálnom čase. Optimálna hranica je v tomto prípade daná špecifikáciou dostupného hardvéru pre virtuálnu realitu (konkrétne HMD HTC Vive) a dá sa vyčísliť ako 180 (90 pre každé oko) snímkov v rozlíšení 1200x1080 za sekundu. Vzhľadom na náročnosť zobrazovaných scén je toto číslo na hranici možností najmodernejšieho grafického hardvéru.

Na samotný proces tvorby obrazu nie sú kladené žiadne výnimočné nároky a nevyžadujú sa pokročilé renderovacie techniky. Modul by mal zvládnuť základné zobrazenie pevných materiálov zastrešený Phongovým osvetľovacím modelom.

Modul by mal byť tiež pripravený na prípadné budúce rozšírenia, ako renderovanie zrkadiel, tieňov či pokročilejší osvetľovací model.

Prezentácia

Modul musí byť schopný zobraziť výsledok svojej práce do okna aplikácie VRUT na to určeného, ktoré bolo modulu priradené. Modul taktiež musí zvládať prácu v režime full-screen. Tieto úlohy musí zvládať v súlade s používanými knižnicami pre správu okien prostredia VRUTu.

3.1.2 Požiadavky na architektúru

Hlavný dôraz v tejto oblasti je kladený na zapuzdrenosť výsledného modulu, najmä s ohľadom na možné znovupoužitie či rozšírenie celej práce. V praxi to vyjadruje potrebu vhodne navrhnutého komunikačného rozhrania — či už vstupného (napr. podpora iných implementácií scény) alebo výstupného (podpora iných knižníc pre správu okna/celého užívateľského rozhrania).

3.2 Architektúra modulu VRender

Na základe definovaných požiadaviek z podkapitoly 3.1 bol vytvorený model architektúry modulu VRender. Je popísaný pomocou diagramu tried jazyka UML a ako celok je k nahliadnutiu v prílohe C. Tento diagram popisuje finálny stav návrhu, pripravený na implementáciu. Vznikol ako produkt mnohých iterácií v cykle vývoja aplikácie.

V nasledujúcej časti práce vzniknutý model rozdelíme na logicky súvisiace časti plniace špecifické úlohy a pripojíme k nim detailný návrh riešenia týchto úloh.

3.2.1 Riadiaca časť

Táto časť obsluhuje komunikáciu s jadrom VRUTu a zabezpečuje korektnú inicializáciu Vulkanu a ostatných submodulov.

Inicializácia

Po spustení modulu sa tento pokúsi inicializovať Vulkan, a to v súlade s pravidlami popísanými v 2.1.3. Pokúsi sa vytvoriť inštanciu Vulkanu, a to s nasledovnými požiadavkami na vrstvy a rozšírenia (viď. 2.1.6):

- Ladenie
  sada *standard validation* vrstiev, rozšírenie *debug report*
- Prezentácia výsledku do okna
  obecné rozšírenie *surfaceKHR*, špecifické rozšírenie *surface* podľa použitého operačného systému (podpora pre win32 a XLIB, viď. 2.2.4)

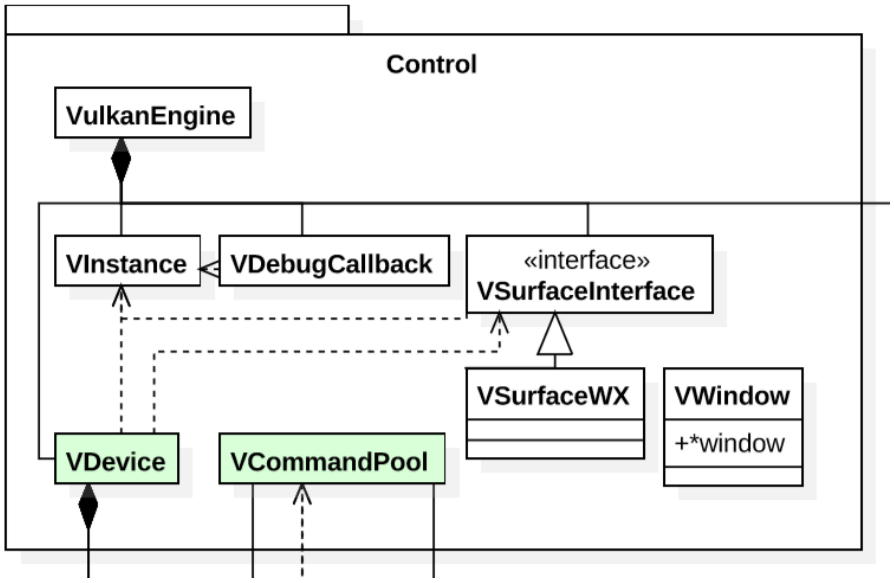
Po úspešnom vytvorení inštancie sa v prípade nutnosti ladenia pripraví podpora pre potrebné výstupy, tzv. *debug callback*, inak sa rovno pokračuje na vytvorenie plochy pre vykresľovanie, tzv. *surface*. Surface je priamo zviazaný s oknom, a keďže okná sú súčasťou operačného systému, je nevyhnutné od neho získať parametre tohoto okna. Obdobne ako príprava ladiaceho výstupu, aj táto časť inicializácie sa môže preskočiť, a to v prípade, že od modulu nebudeme vyžadovať priamu prezentáciu renderu do okna.

Posledným krokom inicializácie v riadiacej časti je výber fyzického zariadenia a vytvorenie logického zariadenia, nad ktorým budú bežať všetky ďalšie úlohy v procese renderovania. Zo zoznamu dostupných fyzických zariadení sa vyberie také, ktoré podporuje grafické operácie a požadované rozšírenie (*swapchainKHR*, viď. 3.2.5) pre prípadnú prezentáciu výsledku priamo na obrazovku. Nad ním je následne vytvorené logické zariadenie a spolu s ním sa pripravia aj fronty pre zasielanie príkazov — *command buffers* na GPU.

Po úspešnom vytvorení zariadenia nasleduje inicializácia zvyšných submodulov, popísaných ďalej v tejto kapitole.

MultiGPU rendering

Pôvodne návrh počítal s podporou renderovania na viacerých zariadeniach súčasne, čo vo fáze jeho prípravy nebolo zo strany API priamo podporované. Vulkan je však v aktívnom vývojovom procese a špecifikácia verzie 1.0.42 priniesla rozšírenie *device group*, ktoré malo za následok stratu významu pôvodne plánovaných krokov. Napriek dostupnosti nového rozšírenia sa však od podpory multiGPU renderovania v tejto verzii modulu VRender pre značnú komplikovanosť daného problému upustilo.



Obr. 3.1: Redukovaný diagram tried vyjadrujúci vzťahy v riadiacej časti. Pre úplný diagram viď. príloha C.

Command Pool

Command buffery predstavené v podkapitole 2.1.3 nie sú alokované priamo z pamäte zariadenia, ale z predpripraveného Vulkan objektu *command pool*. Keďže sa neoperuje priamo s pamäťou zariadenia, command pool nespadá pod správcu pamäte zariadenia, ale bol navrhnutý ako súčasť riadiace časti. Aplikácia predpokladá využitie rôznych command poolov s odlišnými vlastnosťami, ktoré budú stanovené jednotlivými submodulmi.

Navrhnuté riešenie

Diagram na obrázku 3.1 pomerne priamočiaro modeluje popísaný postup incializácie.

Konkrétne implementácie rozhrania `VSurfaceInterface` slúžia pre podporu vykresľovania pomocou rôznych knižníc umožňujúcich správu okna. Pre modul VRender je zatiaľ navrhnutá implementácia v triede `VSurfaceWX` podporujúca vytvorenie Vulkan surface pri použití knižnice wxWidgets.

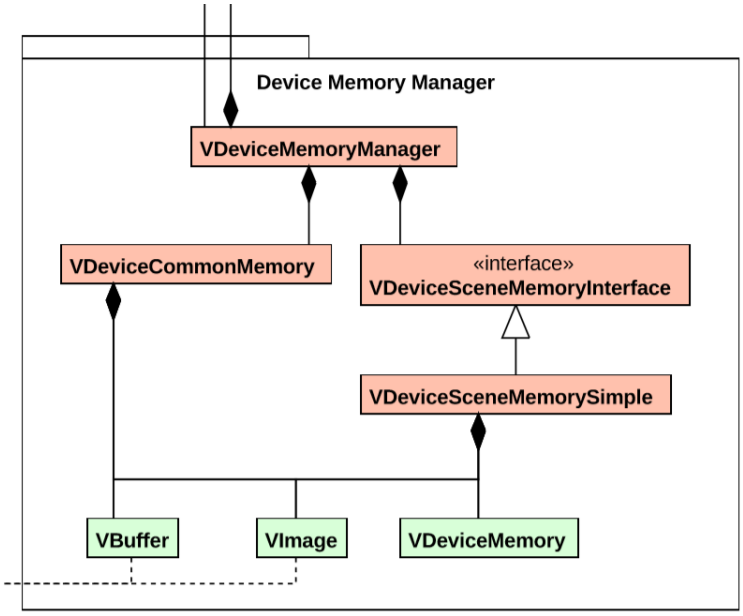
K vytvoreniu surface je tiež potrebný objekt triedy `VWindow`, ktorý abstrahuje nevyhnuté informácie o okne použitom pre vykresľovanie. Bol zavedený najmä pre zachovanie multiplatformnej podstaty modulu.

3.2.2 Správca pamäte zariadenia

Z dôvodov popísaných v sekciách 2.1.4 a 3.1.1 tvorí subsystém správy pamäte zariadenia naprosto kritickú súčasť modulu VRender a jeho služby sú využívajú prakticky všetky ostatné subsystémy modulu.

Navrhnuté riešenie

Všetky potrebné alokácie pamäte zariadenia budú vykonané v správe tohoto subsystému, budú ním vlastnené a s jeho zničením zaniknú.



Obr. 3.2: Redukovaný diagram tried vyjadrujúci vzťahy v správcovi pamäte zariadenia. Pre úplný diagram viď. príloha C.

Pre obecnú podporu viacúrovňovej subalokácie pamäte v module boli navrhnuté tieto triedy (jednotlivé úrovne zodpovedajú číslovaniu z obr. 2.3):

- Alokácia z hromady (1. úroveň)
  Skutočná alokácia fyzickej pamäte zariadenia. Pre jej realizáciu poslúži inštancia triedy `VDeviceMemory`. Táto následne uchováva metadáta potrebné pre vnútornú správu alokovanej pamäte.

- Prepojenie pamäťového zdroja a alokovanej pamäte (2. úroveň)
  Pre pamäťové zdroje *buffer* a *image* sú pripravené triedy `VBuffer` a `VImage`. Aplikácia definuje požiadavky na daný buffer (image) a následne ho spojí (ang. *binding*) s fyzickou pamäťou reprezentovanou objektom triedy `VDeviceMemory`. O tomto vzťahu samotné objekty nevedia (ako je zrejmé z diagramu na obrázku 3.2).

  Jeden alokovaný blok takto môže slúžiť viacerým zdrojom, pokiaľ spĺňa ich požiadavky (bol alokovaný z hromady, ktorá spĺňa tieto požiadavky).

  Triedy `VBuffer` a `VImage` okrem podpory základných operácií nad konkrétnym zdrojom (podobne ako trieda `VDeviceMemory`) uchovávajú metadáta pre vnútornú správu priradenej pamäte.

- Subalokácia zo zdroja (3. úroveň)
  Ako bolo naznačené v podkapitole 2.1.4, zdroje môžu v rámci svojich pamäťových možností vytvárať virtuálne podzdroje. Pre buffer je to virtuálny buffer, definovaný veľkosťou a offsetom v rámci rodičovského bufferu, zatiaľ čo pre image sú to pohľady (ang. *image views*), umožňujúce rôzne interpretovať časti pôvodného image.

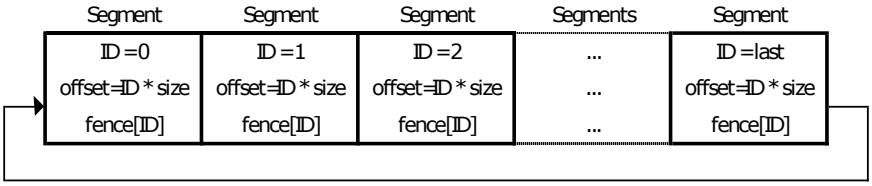
  Táto úroveň alokácie je navrhovaná ako súčasť tried `VBuffer` a `VImage`, kde adekvátne metódy budú realizovať vytvorenie podzdroja.

Konkrétne pamäťové požiadavky modulu bude obsluhovať trieda `VDeviceMemoryManager`. Obsahuje sadu obecných pomocných metód pre prácu s pamäťou (zarovnanie, spojenie), vlastní command pool (viď. podkapitoly 2.1.3 a 3.2.1) určený na alokáciu krátkodobých command bufferov pre kopírovanie dát a zároveň poskytuje rozhranie dvom špecializovaným podmanažérom, `VDeviceCommonMemory` a `VDeviceSceneMemoryInterface`. Z názvov je zrejmý aj ich účel — prvý manažér spravuje obecné pamäťové požiadavky a druhý spravuje všetky požiadavky spojené s uložením dát scény. Druhý manažér je definovaný formou rozhrania, ktorého rôzne implementácie umožňujú efektívne meniť stratégie alokácie pamäte scény podľa konkrétnych požiadaviek scény. Táto práca počíta s jednou, základnou implementáciou v triede `VDeviceSceneMemorySimple`.

`VDeviceCommonMemory`

Navrhnutý model počíta s dvoma obecnými pamäťovými požiadavkami. Prvá je uloženie *depth image*, pokiaľ použitý renderer požaduje hĺbkový test, zatiaľ čo druhá je tzv. *staging buffer*.

Koncept, odkazovaný ako *staging*, popisuje dvojfázové kopírovanie dát na zariadenie, kde sa v prvom kroku kopírujú dáta z operačnej pamäte do do mapovateľnej časti pamäte zariadenia (hromada s príznakom *host visible* popísaná v podkapitole 2.1.4) a následne sa v druhom kroku z dočasného priestoru kopírujú v rámci zariadenia na svoje trvalé miesto (hromada s príznakom *device local* popísaná v podkapitole 2.1.4). Tento prístup sa obzvlášť hodí pre dáta s nízkou frekvenciou zmien — v grafickej aplikácii typicky dáta geometrie a materiálov.



Obr. 3.3: Model kruhového staging bufferu.

Návrh predpokladá riešenie stagingu pomocou kruhového bufferu. Jeden buffer vytvorený nad pamäťou s príznakom *host visible* bude rozdelený na konštantný počet podbufferov s určenou veľkosťou a bude slúžiť ako dočasná prekládková stanica. S prichádzajúcimi požiadavkami sa postupne budú prideľovať časti tohoto bufferu, zatiaľ čo s vybavenými požiadavkami sa budú používané časti opäť uvoľňovať. V prípade zahltenia sa bude čakať na vybavenie požiadaviek vo fronte v poradí FIFO. Koncept je načrtnutý na obrázku 3.3.

Keďže k staging bufferu sa pristupuje ako zo strany CPU (cez perzistentné mapovanie), tak aj zo strany GPU (navyše aynchrónne vzhľadom k behu programu), je nutné zaviesť synchronizáciu v prístupe k tomuto pamäťovému bloku. Pre synchronizáciu medzi CPU a GPU slúži synchronizačné primitívum bariéra (vo Vulkane *fence*). Každý blok staging bufferu je preto strážený svojou bariérou, ktorú CPU pri nahrávaní dát do bloku nastaví do základného stavu (*unsignaled*) a GPU ju po vykonaní asynchrónneho kopírovania v rámci svojich pamätí označí (*signaled*). Pokiaľ CPU vytvorí (po pretočení celého kruhového bufferu) požiadavku na zablokovaný blok, musí čakať, kým GPU obslúži požadované kopírovanie.

Pre kopírovanie pamäťového zdroja image je možné použiť tzv. *staging image*, vzhľadom na odporúčania v článku [4]) však návrh počíta s využitím staging bufferu aj pre dáta zdroja typu image.

`VDeviceSceneMemorySimple`

Trieda implementuje rozhranie správcu pamäte scény `VDeviceSceneMemorySimple` a určuje základné alokačné stratégie pre dáta scény. Tie vychádzajú z povahy dát v aplikácií VRUT a sú rozdelené podľa potrieb správcu scény (viď. podkapitola 3.2.3). Stanovené sú nasledovne:

- Geometria
  Dáta geometrie sú rozdelené na dve časti — dáta vrcholov (zložený dátový typ, kde pre každý vrchol sú definované 3D súradnice a voliteľne sú definované normála povrchu v danom bode a textúrovacie UV súradnice) a indexy. Obe časti sú popísané pamäťovým zdrojom buffer, pre dáta vrcholov je to *vertex buffer*, pre indexy je to *index buffer*.

  Od správcu scény sa očakáva informácia o celkových pamäťových nárokoch geometrie v scéne. Držiac sa postupu popísaného v podkapitole 3.2.2, je vykonaná jedna alokácia (1. úroveň) pamäte, na ktorú sú následne priamo za sebou pripojené (2. úroveň) oba buffery.

  Samotné dáta sú do pripravenej pamäte nahrávané cez staging buffer, postupom popísaným v podkapitole 3.2.2.

  Pokiaľ by renderer nepoužíval indexované vykresľovanie (viď. podkapitola 3.2.4), je možné pre pamäť indexov určiť nulovú veľkosť.

- Materiál
  Dáta materiálu sa v zásade tiež skladajú z dvoch častí — sada atribútov materiálu (konštantná veľkosti pre každý materiál) a prípadná textúra, pokiaľ ju materiál obsahuje. Atribúty materiálu sú popísané pamäťovým zdrojom buffer, konkrétne *uniform buffer*. Dáta textúry sú popísané ako image.

  Držiac sa postupu popísaného v podkapitole 3.2.2, je vykonaná jedna alokácia (1. úroveň) pamäte fixnej veľkosti, na ktorú je následne pripojený (2. úroveň) jeden buffer určený pre uchovávanie dát materiálov. Z neho sú postupne subalokované zdroje (3. úroveň) pre uniformné dáta jednotlivých materiálov. Nezávisle od tohoto procesu je pre každý materiál scény obsahujúci textúru alokovaný (1. úroveň) blok pamäte o veľkosti tejto textúry, na ktorý je následne pripojený pamäťový zdroj image.

  Uniformné dáta materiálu aj dáta textúry sú do pripravenej pamäte nahrávané cez staging buffer, postupom popísaným v podkapitole 3.2.2.

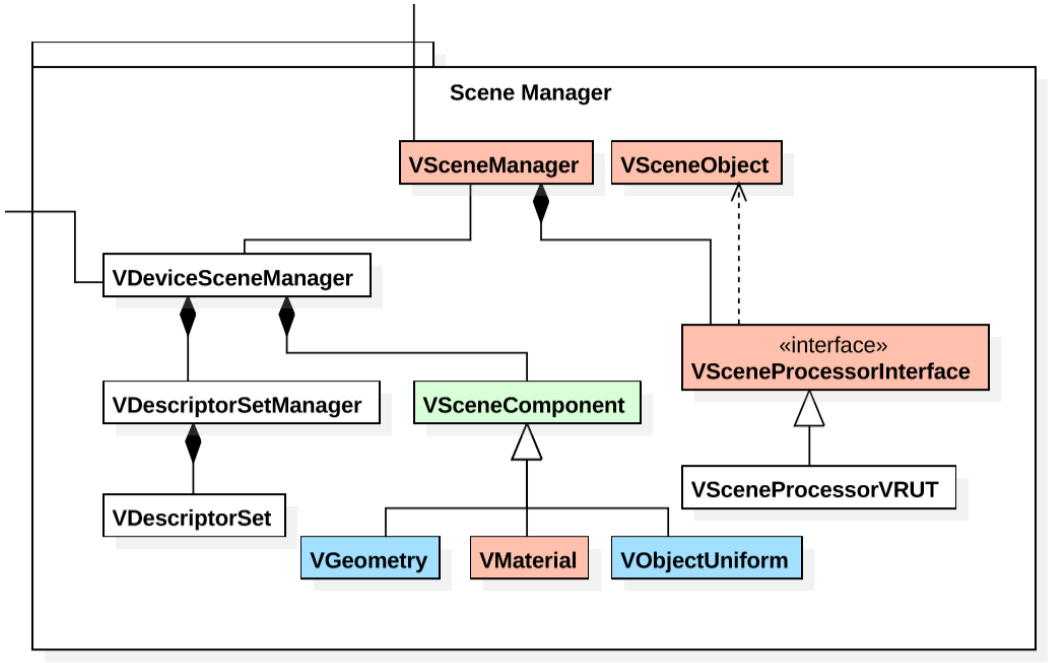
- Uniformné dáta
  V definovanom návrhu do tejto kategórie spadá len transformačná matica každého objektu v scéne. Je zrejmé, že pre každý objekt bude veľkosť matice rovnaká.

  Na rozdiel od predchádzajúcich typov dát nie je pre transformačné matice vhodné použiť staging koncept nahrávania dát. Typicky má totiž platnosť týchto dát relatívne nízku životnosť a teoreticky sa môžu meniť aj pre každý jeden snímok. Preto je pre tieto dáta vyhradený blok pamäte s príznakom *host visible* (alokovaný na 1. úrovni), ktorý je perzistentne namapovaný do adresného priestoru CPU. Na tomto bloku je pripojený jeden *uniform buffer* (2. úroveň), ktorý v pravidelnom intervale danom zarovnaním pamäte obsahuje jednotlivé matice.

3.2.3 Správca scény

Tento subsystém je zodpovedný za prípravu scény pre renderovanie. Sleduje nahrané scény, prevezme ich dáta vo formáte aplikácie (v tomto prípade VRUT), zorganizuje ich podľa požiadaviek používaného renderera (viď. 3.2.4) a pomocou správcu pamäte popísaného v 3.2.2 ich uloží na zariadenie.

Navrhnuté riešenie



Obr. 3.4: Redukovaný diagram tried vyjadrujúci vzťahy v správcovi scény. Pre úplný diagram viď. príloha C.

Jadro VRUTu správami (popísané v podkapitole 2.2.2) informuje modul o všetkých zmenách spojených so scénou. Riadiaca časť modulu (viď. 3.2.1) tieto správy spracuje a podľa ich typu vysiela požiadavky na správcu scény. Jednotlivé scény sú spravované inštanciou triedy `VSceneManager` a v programe sú reprezentované ako objekty triedy implementujúcej rozhranie `VSceneProcessorInterface`. Pre potreby modulu VRender je navrhnutá jedna implementácia, a to triedou `VSceneProcessorVRUT` umožňujúca príjmať dáta od scény v aplikácií VRUT.

`VSceneManager` uchováva aj informáciu o aktívnej scéne, ktorá sa má renderovať. Túto scénu posunie o jednu logickú úroveň nižšie, inštancii triedy `VDeviceSceneManager`. Tá tvorí spojnicu medzi fyzickými dátami v pamäti (vytvára požiadavky na správcu pamäte) a ich logickou reprezentáciou (vo forme descriptor setov, viď. 2.1.7) vhodnou pre použitie v rendereri.

Načítanie scény

Pre načítanie scény bol navrhnutý nasledovný pracovný postup:

1. zmazanie starých dát (pokiaľ nejaké existujú)
2. vytvorenie objektov triedy `VSceneComponent`
   - prechod stromom scény pomocou `VSceneProcessorVRUT`
   - objekty uchovávajú metadáta o jednotlivých zložkách scény (veľkosť, ukazatele na dáta, ...)
3. príprava pamäte pre dáta geometrie (`VGeometry`), správca pamäte
4. príprava descriptor setov pre scénu
   (a) príprava pamäte pre uniformné dáta (`VObjectUniform`), správca pamäte
   (b) vytvorenie descriptor setu pre uniformné dáta objektov scény
      - tento set je pripravený pre použitie vo vertex shaderi
      - obsahuje jeden dynamický uniform buffer (viď. 2.1.7)
   (c) príprava pamäte pre dáta materiálov (`VMaterial`), správca pamäte
   (d) vytvorenie descriptor setu pre dáta materiálov
      - tento set je pripravený pre použitie vo fragment shaderi
      - obsahuje jeden uniform buffer a jeden sampler (pre textúru)

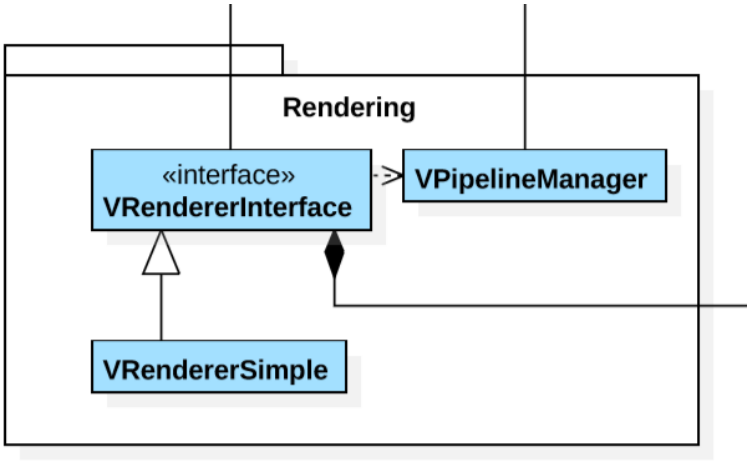
Po ukončení popísaného procesu je scéna predpripravená na použitie, a to s posledným nedostatkom — stále nedošlo k fyzickému skopírovaniu dát scény na zariadenie. Toto kopírovanie je vynútené až v procese renderovania, keď príde reálna požiadavka na vyrendrovanie daného objektu. Renderer vtedy sám vygeneruje požiadavku na nahratie dát tohoto objektu na zariadenie (viď. časť 3.2.4).

Poznámka k descriptor setom

Z obsahu tejto kapitoly, najmä zo sekcie popisujúcej vytvorenie setov, je zrejmé, že deskriptor sety sú veľmi úzko zviazané s použitým rendererom. Keďže sú však zároveň úzko spojené s obsahom scény, ich vytvorenie prebieha práve počas činnosti tohto submodulu. Layouty definuje renderer a pre vytvorenie descriptor setov ich poskytne manažérovi scény.

3.2.4 Rendering

V prípade, že máme k dispozícii tie správne dáta, môžeme pristúpiť k procesu ich vizualizácie. V prostredí rozhrania Vulkan sa jedná o komplexnú schému zahŕňajúcu mnoho aspektov, ktoré je nevyhnutné zohľadniť. Jednotlivé kroky reprezentované pomocou Vulkan objektov sú popísané v podkapitole 2.1.7.

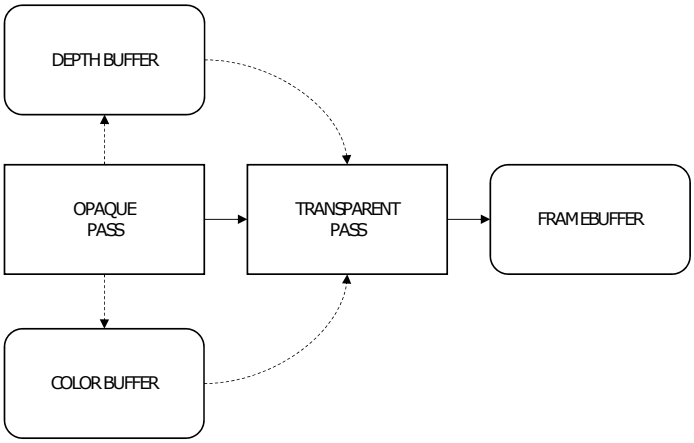


Obr. 3.5: Redukovaný diagram tried vyjadrujúci vzťahy v renderovacej časti. Pre úplný diagram viď. príloha C.

Navrhnuté riešenie

Pre renderer je pripravené rozhranie `VRendererInterface`, ktoré zastrešuje väčšinu prebratých častí renderera. Samostatne stojí vytvorenie pipeline, ktorá je v zásade od renderera nezávislá — rovnaká pipeline môže slúžiť rôznym rendererom. Ich správu, spolu s pipeline cache, má na startosti trieda `VPipelineManager`.

Pre modul VRender bola navrhnutá jedna implementácia spomenutého rozhrania renderera, a to `VRendererSimple`. Renderpass definovaný pre tento renderer je zložený z dvoch subpassov a vyžaduje hĺbkový test. Graf (viď. 2.1.7) použitého renderpassu je znázornený na obrázku 3.6.



Obr. 3.6: Orientovaný acyklický graf vyjadrujúci závislosti v navrhnutom renderpasse.

Súčasťou framebufferu je hĺbkový buffer, ktorý dodá správca pamäte a color buffer, ktorý obsahuje renderovaný obraz a je dodaný od prezentačného submodulu (viď. 3.2.5).

Renderer potrebuje k svojej činnosti jednu grafickú pipeline (viď. 2.1.7), ktorú mu poskytne objekt triedy `VPipelineManager`.

Táto pipleine spĺňa nasledujúce parametre:

- má svoj program pre vertex shader a pre fragment shader
- príjma geometriu s topológiou *triangle list*
- rasterizér vypĺňa celý obsah trojuholníka, nie len hrany
- nevyužíva multisampling, zmiešavanie farieb, stecil buffer
- prebieha hĺbkový test

Nahrávanie command bufferov (viď. ďalej) prebieha nasledovne:

1. vytvorí sa primárny command buffer, ktorý spúšťa samotný renderpass
2. pre každý subpass sa vytvorí samostatný sekundárny command buffer
3. tieto sa postupne volajú na vykonanie z primárneho command bufferu
4. pri zmene vo vykresľovaní sa znovunahrávajú zodpovedajúce sekundárne buffery

Obsah sekundárnych command bufferov sú príkazy portebné pre samotný rendering, ako je pripojenie potrebnej geometrie, descriptor setov a kresliaci príkaz. `VRendererSimple` používa indexované vykresľovanie.

Command buffers

Command buffery pre rendering obsahujúce vykresľovacie príkazy. Renderer si od riadiacej časti obstará svoj vlastný command pool (viď. 3.2.1), z ktorého bude alokovať potrebné command buffery. Z podkapitoly 2.1.5 je evidentné, že pokiaľ by sa všetky príkazy nahrávali do jedného obrieho command bufferu, stratila by sa možnosť paralelizácie tohoto nahrávania. Vulkan preto umožňuje vytvárať dva typy command bufferov na rozdielnej logickej úrovni:

- *Primary command buffers*
  Primárne buffery sú pripravené na priame zaslanie do fronty zariadenia.
- *Secondary command buffers*
  Sekundárne buffery nemôžu byť samostatne zaslané do fronty, ale môžu byť volané z primárneho bufferu, ktorý bol zaslaný na vykonanie.

Vďaka tomuto konceptu je možné rozdeliť veľký command buffer na podčasti a tie nahrávať samostatne.

3.2.5 Prezentácia

Podstatou submodulu prezentácia je prevzať výsledok renderovania a poskytnúť ho užívateľovi v požadovanej forme. Podpora pre túto činnosť však nie je súčasťou jadra Vulkanu — napriek tomu, že sa jedná primárne o grafické API, jeho implementácia je obecne schopná behu aj na zariadeniach bez grafického výstupu. K dispozícii je preto vo forme rozšírení, ktorých podpora sa vyžaduje od konkrétneho zariadenia.

Prezentácia môže v zásade prebehnúť dvoma spôsobmi, a to:

- Onscreen
  Priama prezentácia výsledku na obrazovku do príslušného okna. Vulkan poskytuje podporu pre túto formu pomocou rozšírení *surfaceKHR* a *swapchainKHR*.
- Offscreen
  Výsledok renderovania nie je priamo zobrazený, je však pripravený v pamäti zariadenia na ďalšie použitie, napríklad ako vstup pre ďalší modul v aplikácii (postprocessing, zaslanie po sieti, ...) či uloženie do súboru.

Synchronizácia

Keďže zápis aj čítanie z videopamäte prebieha zaslaním príkazov do front GPU a vieme, že z povahy zariadenia sú zasielané príkazy spracovávané asynchrónne, je nevyhnutné tieto prístupy synchronizovať. Toto je možné realizovať už pomocou predstavenej bariéry (viď. 3.2.2), ale vzhľadom na skutočnosť, že požadujeme synchronizáciu len v rámci zariadenia (len na strane GPU), je výhodnejšie použiť implementačne jemnejšie synchronizačné primitívum, semafor.

Navrhnuté riešenie

Pre aplikáciu bolo navrhnuté rozhranie `VPresentationInterface` sprostredkujúce potrebné informácie potrebné pre zobrazenie výsledku. Návrh počíta s jeho implementáciou vo forme triedy `VPresentationSwapChain`, ktorá umožňuje zobrazovanie obrázkov do okna aplikácie VRUT pomocou Vulkan objektu swapchain (viď. 2.1.7). Využíva k tomu prezentačný mód mailbox popísaný v odkazovanej sekcii a pre odstránenie problémov s trhaním obrazu implementuje triple-buffering.

Pre synchronizáciu využijeme dva semafory:

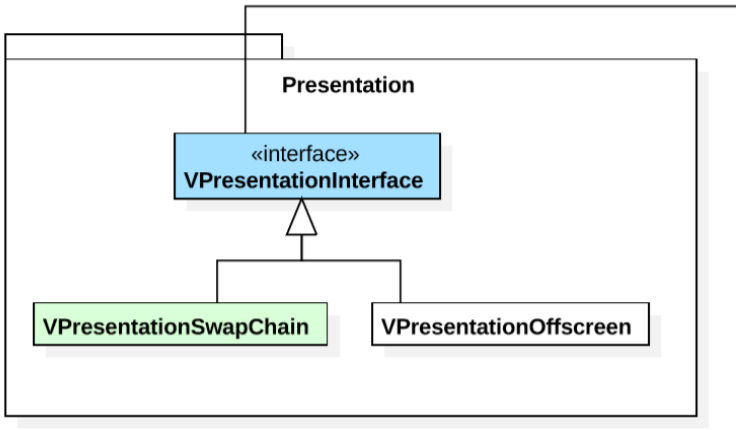
- *Image available*
  signalizuje pripravenosť obrázku na začatie renderovania (obrázok už bol prezentovaný)
- *Render finished*
  renderovanie obrázku bolo ukončené a je pripravený na zobraznie

3.3 Zhrnutie

Korektné načítanie a spustenie modulu zabezpečí jadro VRUTu (viď. 2.2.2). Kompatibilitu s VRUTom z pohľadu modulu zabezpečíme implementáciou definovaného rozhrania `RenderModule`. Vstupným bodom modulu potom bude práve táto implementácia, ktorá bude obsahovať inštanciu triedy `VulkanEngine`. Tak, ako je znázornené v diagrame v prílohe C, `VulkanEngine` je top-level trieda zastrešujúca pod sebou celú funkcionalitu modulu.

V nasledujúcich bodoch je zhrnutý štandardný beh modulu po jeho úspešnom spustení:

1. inicializácia Vulkanu a všetkých submodulov popísaná v časti 3.2.1
2. modul čaká od jadra správu o scéne na renderovanie
3. po jej obdržaní scénu spracuje postupom popísaným v časti 3.2.3



Obr. 3.7: Redukovaný diagram tried vyjadrujúci vzťahy v prezentačnej časti. Pre úplný diagram viď. príloha C.

4. požadované objekty zo scény sú nahraté na zariadenie a začína rendering v nekonečnej slučke
5. scéna je na zariadení renderovaná v súlade s časťou 3.2.4 a výsledky sú užívateľovi predkladané formou popísanou v časti 3.2.5
6. kedykoľvek v tomto procese môže jadro VRUTu požiadať o renderovanie inej scény, v tom prípade sa všetky používané dáta zmažú a pokračuje sa od bodu 3

3.3.1 Vstupno-výstupné rozhranie

Napriek tomu, že primárne je modul určený pre beh v systéme VRUT, jeho návrh bol koncipovaný na prípadné v rámci možností jednoduché osamostatnenie. V rámci textu podkapitoly 3.2 boli tieto komunikačné body popísané, na tomto mieste sú pre prehľadnosť zhrnuté k sebe.

- Vstup — Scéna
  Pre abstrakciu získavania dát scény boli navrhnuté triedy:
  - `VSceneObject`
  - `VSceneProcessorInterface`
- Výstup — Prezentácia do okna
  Pre abstrakciu prezentácie výsledkov do okna boli navrhnuté triedy:
  - `VWindow`
  - `VSurfaceInterface`

Jedinou priamou závislosťou modulu na VRUTe, s ktorou tento návrh počíta, zostáva využitie matematickej knižnice VRUTu.

Kapitola 4

Realizácia

Táto kapitola vo svojej prvej časti popisuje špecifiká procesu implementácie predstaveného návrhu. Predstaví základné informácie o použitých technológiách, ako aj niekoľko implementačne zaujímavých alebo zložitých použitých postupov.

Jej druhá časť sa venuje predstaveniu výslednej aplikácie, a to formou prezentácie výstupu a najmä porovnaním s existujúcimi riešeniami v programe VRUT.

4.1 Implementácia

Špecifikácia VRUTu (viď. [8]) definuje základné požiadavky na kvalitu zdrojového kódu. Modul VRender sa snaží tieto podmienky dodržať spolu s ďalšími obecnými konvenciami, danými zvolenými technológiami.

4.1.1 Obecné informácie

Nasledujúce informácie informujú o zvolených technológiách, používaných nástrojoch a verziách.

Jazyk implementácie

Implementačným jazykom systému VRUT je jazyk C++ vo verzii C++11. V súlade s informáciami predstavenými v podkapitole 2.1.8 a pre zachovanie kompatibility s VRUTom bol ako implementačný jazyk modulu tiež C++ vo verzii C++11. Pre Vulkan bol využitý C++ wrapper (viď. 2.1.8).

Verzia SDK

Počas práce na tomto projekte prešiel Vulkan množstvom zmien, ktoré sa aplikácia snažila vždy reflektovať. Odovzdaná aplikácia je otestovaná a plne kompatibilná s SDK v1.0.46.

Platforma

Aplikácia VRUT (vrátane všetkých modulov) sa k prekladu pripravuje pomocou nástroja cmake. Preklad modulu VRender bol testovaný a je funkčný pomocou prekladača dostupného v nástroji Visual Studio 2013 update 5, ako aj Visual Studio 2015 update 3. Taktiež bol testovaný preklad pomocou prekladača GCC verzie 7.1. Vo všetkých prípadoch bola použitá 64bitová architektúra.

Aplikácia požaduje 100% kompatibilitu s implementáciou Vulkanu v ovládačoch od firmy NVIDIA, keďže v spoločnosti ŠKODA sa používajú výhradne riešenia od tohoto výrobcu. Aplikácia je preto otestovaná a plne funkčná s použitím ovládačov pre triedu Quadro (v377.35) a pre triedu GeForce (v382.05). Kompatibilita s implementáciami iných výrobcov sa síce predpokladá, nebola však explicitne testovaná.

Zdrojový kód

Väčšina zdrojového kódu vznikla ako implementácia prezentovaného návrhu z kapitoly 3 a je pôvodná.

Prvotné demo Vulkan aplikácie vychádza z príkladov distribuovaných spolu s Vulkan SDK (viď. 2.1.8) a z online tutoriálu[1]. Niekoľko pomocných metód sa v podobe kompatibilnej s použitým C++ wrapperom zachovalo až do súčasnej verzie.

Pokiaľ konkrétna metóda v programe rieši špecifický a nejednoznačný problém, v komentároch sú uvedené odkazy do špecifikácie [3] alebo na web, kde je daná problematika vysvetlená.

Časť zdrojového kódu, ktorá sa po preklade stane súčasťou modulu VRender, pochádza z iných modulov systému. Týka sa problematiky načítania scény z VRUTu (využíva ich len trieda `VSceneProcessorVRUT`). Presne vyhradený zoznam týchto tried sa nachádza v prílohe A a ich autor je často uvedený v záhlaví jednotlivých súborov (tak ako aj pri ostatných súboroch modulu VRender a pri väčšine zdrojových súborov VRUTu).

4.1.2 Podpora

Aplikácia požadovala pre svoju implementáciu vyriešiť niekoľko funkčných záležitostí, ktoré nie sú priamo späté s činnosťou modulu.

RAII

Ako je uvedené v podkapitole 2.1.8, SDK pre Vulkan je šírené primárne v jazyku C. Použitý C++ wrapper stavia na tejto implementácii a je ňou obmedzený. Vulkan objekty predstavené v podkapitole 2.1.3 aj napriek použitému wrapperu nezodpovedajú C++ objektom a strácajú tak niektoré vlastnosti.

Pre zachovanie platnosti idiómu RAII (ang. *resource acquisition is initialization*, efektívne spojenie platnosti zdroja s životným cyklom objektu) a tým pádom umožnenie správy dynamicky alokovaných Vulkan objektov pomocou tzv. *smart pointers* bola zavedená pomocná šablónovaná trieda `VDeleter<T>`.

Je implementovaná v pomocnom súbore `VUtil.h` a ako je z predchádzajúcej časti zrejmé, spája vytvorenie a zničenie Vulkan objektu do jediného C++ objektu. S jej pomocou sú spravované bez výnimky všetky Vulkan objekty.

SPIR-V

Kód použitých shaderov je zapísaný v jazyku glsl v4.5. Preklad do požadovaného jazyka SPIR-V je zabezpečený prekladačom šíreným spolu s Vulkan SDK a na jeho zjednodušenie je pripravený jednoduchý skript v súbore `./shaders/compileShaders.bat`. Preložené shadery sú následne premiestnené na správne miesto (k binárnym súborom) automaticky pomocu nástroja cmake.

[1] https://vulkan-tutorial.com/

4.1.3 Ladenie

Aplikácia beží súčasne na CPU aj GPU, k ladeniu jednotlivých častí bolo nutné zvoliť osobitý prístup.

Ladenie CPU

Na strane CPU boli použité štandardné metódy ladenia programu, konkrétne debugger a profilovacie nástroje prítomné v prostredí Visual Studio 2015 update 3. Navyše (okrem vrstiev popísaných v 3.2.1) bola využitá ladiaca vrstva *api_dump* implementovaná vo Vulkan SDK. Táto vrstva zbiera a chronologicky za sebou ukladá informácie o všetkých volaniach do rozhrania Vulkan, spolu so všetkými parametrami.

Ladenie GPU

Na ladenie programu na strane GPU bol využitý grafický ladiaci nástroj RenderDoc[2] šírený pod licenciou MIT. Jeho výhodou oproti nástroju NSight je plná podpora rozhrania Vulkan.

4.2 Výstup aplikácie

Výstup aplikácie prebieha na dvoch úrovniach, a to ako vizuálny vo forme prezentovaného obrazu a textový vo forme logovaných výstupov. Obe tieto formy boli využité pri hodnotení kvality aplikácie.

4.2.1 Testovacie dáta

Dáta použité na testovanie poskytla spoločnosť ŠKODA na základe uzavretia zmluvy o záverečnej práci. Detaily jednotlivých scén sú uvedené v tabuľke 4.1, v scéne KODIAQ sú obsiahnuté produkčné dáta zodpovedajúce aktuálnemu modelu na trhu.

Tabuľka 4.1: Scény použité pre testovanie a porovnanie modulu VRender.

Názov	Uzly	Geometrie	Materiály	Počet trojuholníkov
Q7	93	72	28	78 360
Auta	1081	777	179	845 334
KODIAQ	n/a	n/a	n/a	39 678 128

4.2.2 Priebeh testov a výsledky

Testovanie prebehlo na počítačoch popísaných v tabuľke 4.2. Počítač označený ako Professional je nadštandardne vybavený a bol mi k dispozícii na pracovisku v ŠKODA.

Hlavná časť testovania spočívala v príprave testovacieho skriptu, ktorý využíva možnosť aplikácie VRUT akceptovať inštrukcie zapísané v jazyku javascript. Podstatou tohoto skriptu je nastaviť všetky potrebné parametre modulu a následne formou transformácií vytvoriť prechod scénou. Tento prechod zohľadňuje rôzne pohľady na scénu, ktoré majú priamy vplyv na výkon zobrazovania.

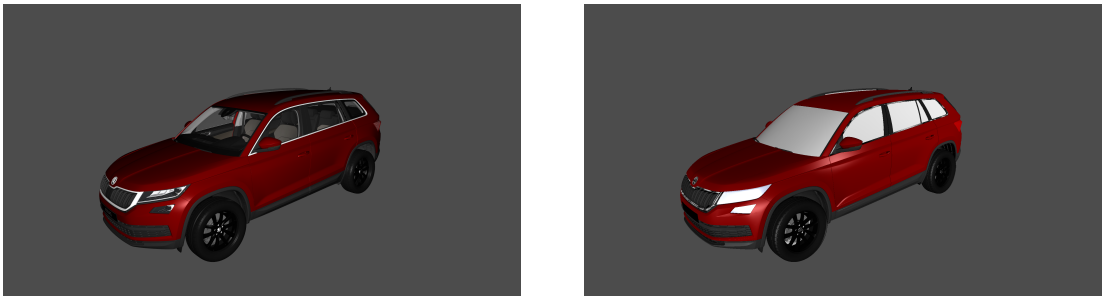
[2] https://github.com/baldurk/renderdoc

Tabuľka 4.2: Špecifikácia počítačov použitých pre testovanie a porovnanie modulu VRender.

Názov	Intel CPU	RAM	NVIDIA GPU	GPU mem
Personal	Core i7-950; 3,06 GHz	16 GB	gtx970 (Maxwell)	4 GB
Professional	Xeon E5-2667v2; 3,3 GHz	64 GB	Quadro K6000 (Kepler)	12 GB

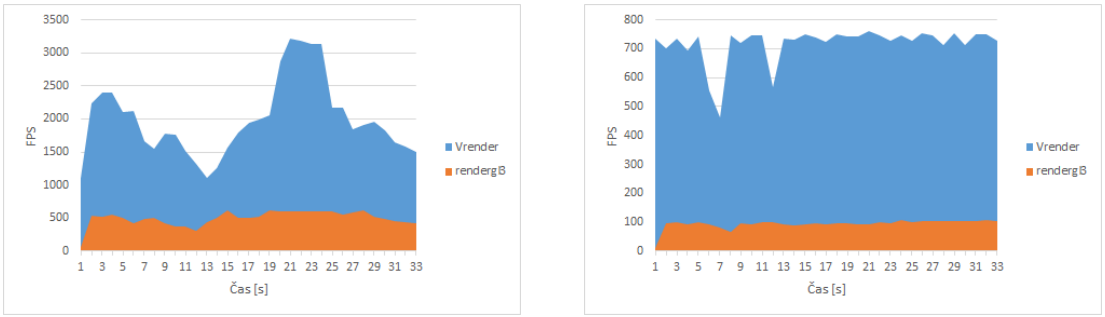
V čase, keď bolo možné testovať na stroji Professional z tabuľky 4.2 však bohužiaľ ešte nebolo testovacie prostredie plne pripravené, a preto sú k dispozícii len výsledky pri testovaní statického obrazu.

Modul rendergl3 (viď. 2.2.3) bol stanovený ako jediný existujúci modul, voči ktorému majú testy relevantnú výpovednú hodnotu, a to najmä z hľadiska očakávaného výstupu.



(a) rendergl3 (b) VRender

Obr. 4.1: Obrazový výstup renderovacích modulov pri použití scény KODIAQ z tabuľky 4.1.



(a) Scéna Q7 (b) Scéna Auta

Obr. 4.2: Porovnanie výkonu modulov pri renderovaní scén z tabuľky 4.1. Hodnoty boli získané na stroji Personal z tabuľky 4.2 v rozlíšení 1920x1200.

4.2.3 Zhodnotenie výsledkov

Ako je možné pozorovať na obrázku 4.1), výsledky zobrazenia sú si značne podobné. Pri tvorbe modulu VRender však nebola kladená priorita na dojem z výsledného obrazu a ten je tak zaťažený niekoľkými vizuálnymi nedostatkami. Prvý je pozorovateľný priamo na obrázku, a tým je chýbajúca podpora pre zobrazovanie priehľadných materiálov. Druhý nedostatok už nie je z obrázku viditeľný, prejaví sa len pri určitom pohľade do scény. Je ním

Tabuľka 4.3: Porovnanie výkonu modulov pri renderovaní scény KODIAQ z tabuľky 4.1. Hodnoty boli získané na stroji Professional z tabuľky 4.2 v rozlíšení 2560x1440

Modul	FPS
VRender	84
rendergl3	134

artefakt vznikajúci pri mapovaní textúry spôsobený chýbajúcou podporou mipmapovania a filtrovania (trilineárneho alebo anisotropického) textúr. Na meraných výsledkoch rozoberaných ďalej v tejto kapitole sa oproti modulu rendergl3 podpisujú oba zmienené problémy, pričom priehľadnosť materiálov pozitívne a chýbajuce mip mapy naopak negatívne. Podpora týchto techník z nebola povahy práce súčasťou návrhu.

Pri porovnaní výkonu pri rôznom objeme scén (viď. obr. 4.2 a tab. 4.3) je zrejmé, že modul VRender dominuje pri zobrazovaní relatívne malých scén, zatiaľ čo za modulom rendergl3 zaostáva pri zobrazovaní tých relatívne veľkých. Pri menších scénach pravdepodobne modul rendergl3 dopláca na svoju robustnosť, aj tak je však miera vzniknutého rozdielu nečakane výrazná. Pri veľkých scénach sa však prejavuje jeho kvalita spočívajúca v niekoľkých rokoch vývoja a optimalizácií. VRender pri veľkých scénach trpí najmä absentujúcim triedením geometrie podľa použitého materiálu, v dôsledku čoho dochádza na GPU k častému prepínaniu zdrojov (viď. 2.1.7). Podpora roztriedených geometrií je zo strany Vulkanu v module pripravená, treba ju však doplniť na strane prípravy scény.

V neposlednom rade sa ukázalo, že pre CAD scénu na úrovni komplexnosti scény KODIAQ renderovanú na adekvátnom hardvéri sa podarilo dosiahnúť stanovenú métu z podkapitoly 3.1.1 pre plynulé renderovanie virtuálnej reality v okuliaroch HTC Vive.

Kapitola 5

Záver

Cieľom práce bolo vytvoriť zobrazovací plugin pre aplikáciu VRUT. Tento plugin mal byť založený na novom, nízkoúrovňovom API s názvom Vulkan.

Obe technológie, ako aj možnosti ich prepojenia, boli predstavené v kapitole 2. Po oboznámení sa s technológiami a princípmi ich fungovania bolo pristúpené k návrhu modulu, ktorý umožňuje ich spojenie. V kapitole 3 sú popísané požiadavky kladené na tento modul a následne je prezentovaný samotný návrh. Podľa vytvoreného návrhu bolo možné pristúpiť k implementácii a následne k testovaniu a porovnávaniu výkonnosti. Kapitola 4 popisuje technológie a techniky použité pri vývoji ako aj dosiahnuté výsledky. Jej záver je venovaný porovnaniu týchto výsledkov s inými modulmi v aplikácii VRUT.

Vytvorený modul spĺňa stanovené požiadavky. Je schopný pracovať so scénami obsahujúcimi desiatky miliónov trojuholníkov, za behu je stabilný. Vizuálna kvalita výstupu a úroveň spracovania interných správ vo VRUTe zatiaľ neumožňuje jeho produkčné nasadenie, nameraný výkon však ukazuje, že po vhodnom doplnení by modul mohol vytvoriť priamu konkurenciu aktuálne používaným riešeniam.

Prioritou pri tvorbe návrhu bolo umožnenie čo najjednoduchšieho rozširovania aplikácie, s minimálnou potrebou upravovať už existujúci kód. Bez výraznejších komplikácií je možné do programu doplniť techniky ako zobrazovanie priehľadných materiálov či podporu pre komprimované textúry, čo by boli prvé kroky pri snahe doviesť modul do produkčného stavu. Ďalším krokom k zvýšeniu vizuálnej kvality by bolo pridanie zobrazovania tieňov. Modul je možné rozšíriť aj na nižšej úrovni, a to napríklad implementáciou nových stratégií pri správe pamäte zariadenia.

Taktiež je možné z modulu osamostatniť vykresľovacie jadro a s minimálnymi úpravami ho využiť pre zobrazovanie v inej aplikácii, než je práve VRUT.

Zo strany ŠKODA AUTO a.s. je o nasadenie zobrazovacieho modulu založeného na rozhraní Vulkan záujem, a preto je pravdepodobné, že na výsledkoch tejto práce vznikne riešenie nasadené priamo v produkčnom prostredí.

Literatúra

[1] Group, K.: *Graphics and Compute Belong Together*. 2016, [Online; navštívené 10.01.2017].
URL https://www.khronos.org/assets/uploads/developers/library/overview/2015_vulkan_v1_Overview.pdf

[2] Group, K.: *Khronos Releases Vulkan 1.0 Specification*. 2016, [Online; navštívené 10.01.2017].
URL https://www.khronos.org/news/press/khronos-releases-vulkan-1-0-specification

[3] Group, K.: *Vulkan® 1.0.38 - A Specification (with all registered Vulkan extensions)*. 2016, [Online; navštívené 10.01.2017].
URL https://www.khronos.org/registry/vulkan/specs/1.0-extensions/pdf/vkspec.pdf

[4] Hebert, C.: *Vulkan Memory Management*. 2016, [Online; navštívené 10.01.2017].
URL https://developer.nvidia.com/vulkan-memory-management

[5] Hector, T.: *Vulkan: Scaling to multiple threads*. 2015, [Online; navštívené 10.01.2017].
URL https://imgtec.com/blog/vulkan-scaling-to-multiple-threads/

[6] Kubisch, C.: *Vulkan Shader Resource Binding*. 2016, [Online; navštívené 16.05.2017].
URL https://developer.nvidia.com/vulkan-shader-resource-binding

[7] Kyba, V.: *Modulární 3D prohlížeč*. Diplomová práce. České vysoké učení technické v Praze, Fakulta elektrotechnická. Vedoucí práce doc. Ing. Bittner Jiří, Ph.D., 2009, [Online; navštívené 11.01.2017].
URL https://dip.felk.cvut.cz/browse/pdfcache/kybav1_2009dipl.pdf

[8] Míšek, A.; aj.: *Dokumentace aplikace VRUT*. 2011.

[9] Sellers, G.; Kessenich, J.: *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Pearson Education, Inc., 2016, ISBN 978-0134464541.

[10] Shilov, A.: *AMD: Vulkan absorbed 'best and brightest' parts of Mantle*. 2015, [Online; navštívené 10.01.2017].
URL http://www.kitguru.net/components/graphic-cards/anton-shilov/amd-vulkan-absorbed-best-and-brightest-parts-of-mantle/

[11] Wellings, M.: *The new Vulkan Coordinate System*. 2016, [Online; navštívené 16.05.2017].
URL https://matthewwellings.com/blog/the-new-vulkan-coordinate-system/

Prílohy

Príloha A

Obsah priloženého pamäťového média

- `/dip/`
  - zdrojové súbory pre preklad technickej správy pomocou latex
- `/doc/`
  - manuál k inštalácii a spusteniu
- `/src/`
  - zdrojové súbory modulu VRender
  - `/src/temp/`
    - súbory obsiahnuté v tomto podadresári sú súčasťou výsledného modulu VRender
    - nevznikli ako súčasť práce, boli prevzaté z iných častí aplikácie VRUT
- `/test/`
  - javascript určený pre testovanie modulu, slúži aj ako demo

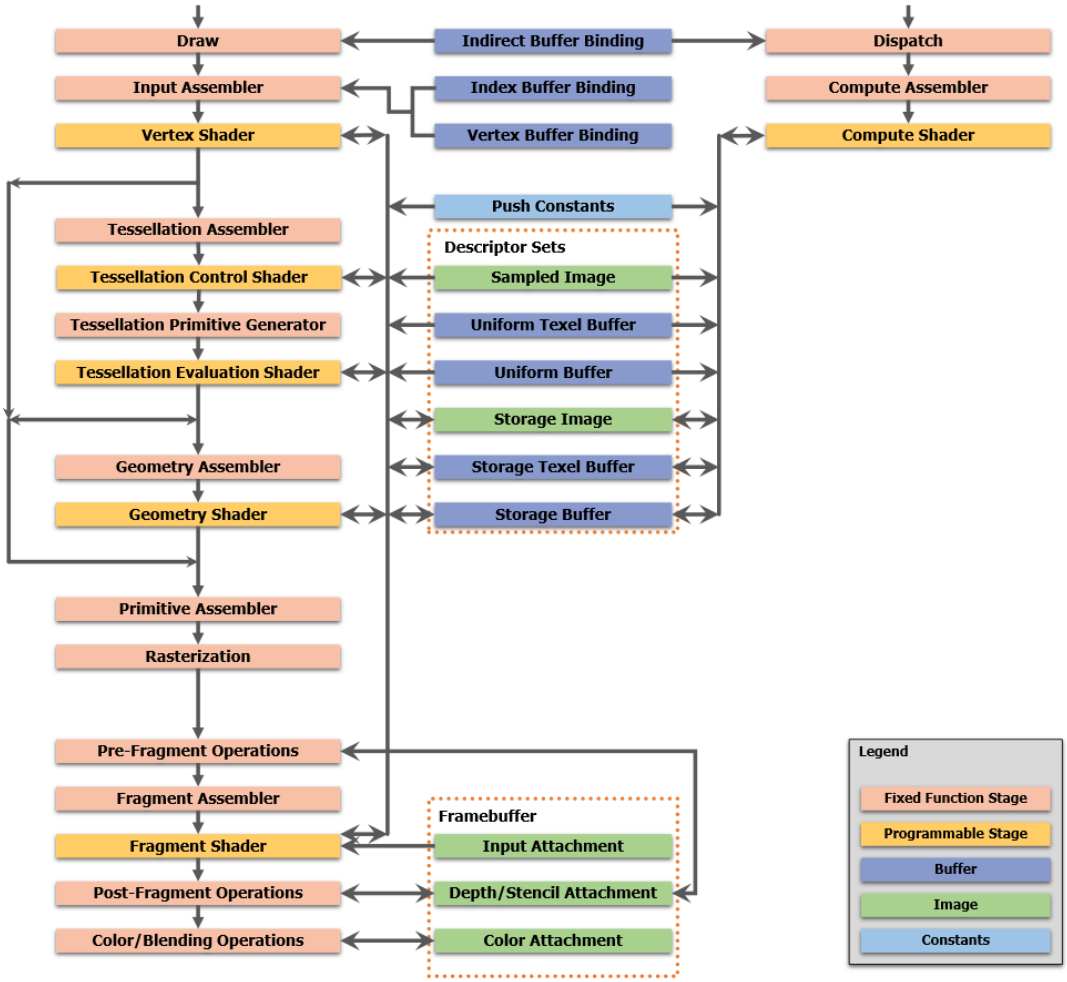
A.1 Poznámka

Program VRUT aj jeho zdrojové kódy sú majetkom spoločnosti ŠKODA AUTO a.s., zdrojové kódy modulu VRender dostupné na tomto CD boli v súlade s uzavretou zmluvou uvoľnené na zverejnenie v tejto práci.

Priložené zdrojové kódy nemôžu byť samostatne preložené ani spustené.

Príloha B

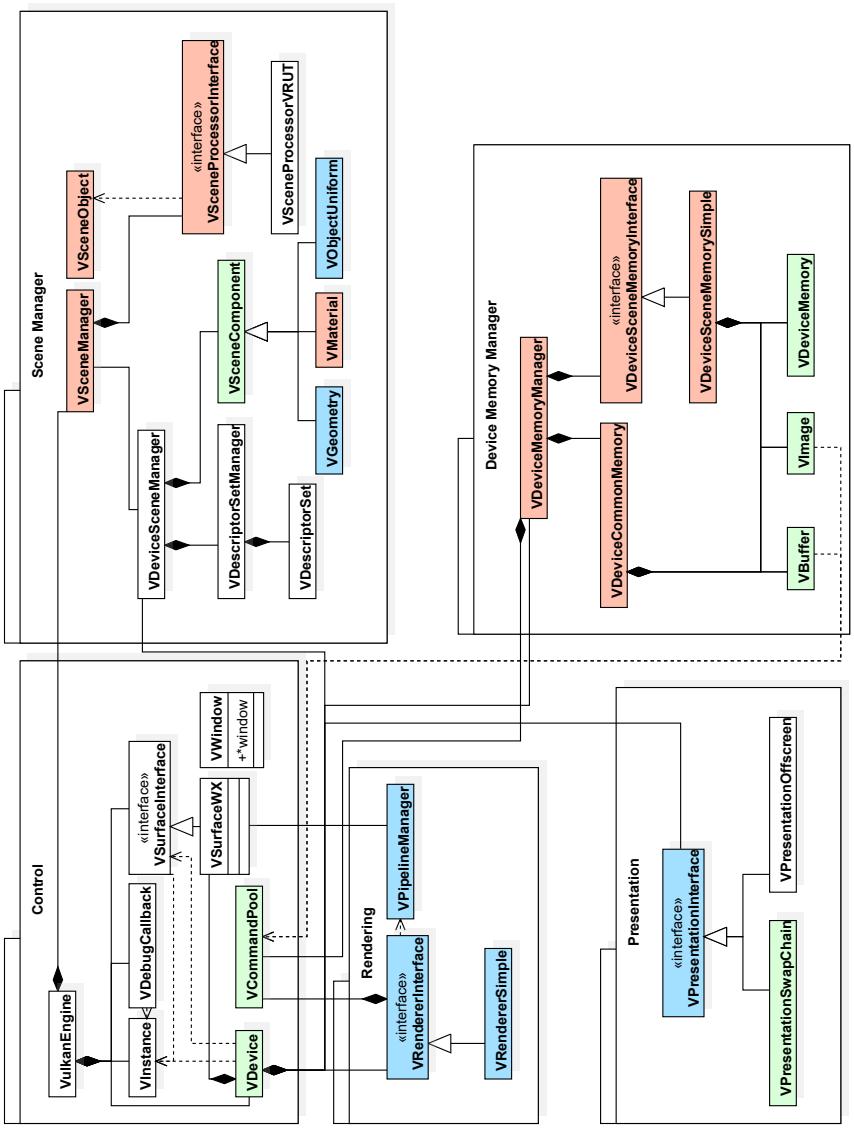
Blokový diagram pipeline v rozhraní Vulkan



Obr. B.1: Blokový diagram pipeline v rozhraní Vulkan [3].

Príloha C

Diagram tried modulu VRender



Obr. C.1: Diagramu tried navrhnutého modulu ako celok.