
**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**


# LADICÍ NÁSTROJ PRO API VULKAN

**A DEBUGGING TOOL FOR VULKAN API**

**BAKALÁŘSKÁ PRÁCE**
**BACHELOR'S THESIS**


**AUTOR PRÁCE** **JOZEF BILKO**
**AUTHOR**

**VEDOUCÍ PRÁCE** **Ing. JAN PEČIVA, Ph.D.**
**SUPERVISOR**



**BRNO 2024**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

# Zadání bakalářské práce

156751

| | |
|---|---|
| Ústav: | Ústav počítačové grafiky a multimédií (UPGM) |
| Student: | **Bilko Jozef** |
| Program: | Informační technologie |
| Název: | **Ladicí nástroj pro API Vulkan** |
| Kategorie: | Počítačová grafika |
| Akademický rok: | 2023/24 |

Zadání:

1. Nastudujte si architekturu dnešních grafických akcelerátorů a grafické rozhraní Vulkan včetně architektury modulu Loader a použití Vulkan layers. Prozkoumejte současné nástroje pro ladění a profilování 3D grafických aplikací.
2. Navrhněte vlastní Vulkan vrstvu (layer) pro ladění 3D aplikací. Vrstva by měla být schopna zobrazovat stav běžící aplikace, obsah textur, obsah bufferů, a podobně.
3. Navrženou vrstvu implementujte. Implementace může obsahovat open-source kód jiných projektů. Tyto části řádně označte zdrojem, odkud pocházejí.
4. Projekt otestujte a vyhodnoťte zkušenosti. Diskutujte další potenciální směry vývoje. Práci prezentujte na internetu.

Literatura:
Dle doporučení vedoucího.

Při obhajobě semestrální části projektu je požadováno:
Funkční prototyp aplikace.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

| | |
|---|---|
| Vedoucí práce: | **Pečiva Jan, Ing., Ph.D.** |
| Vedoucí ústavu: | Černocký Jan, prof. Dr. Ing. |
| Datum zadání: | 1.11.2023 |
| Termín pro odevzdání: | 9.5.2024 |
| Datum schválení: | 10.11.2023 |

# Abstrakt

Cieľom tejto bakalárskej práce je vytvoriť nástroj pre ladenie 2D/3D Vulkan programov, užitočný môže byť ako pre začiatočníkov tak i pre expertov pri vývoji Vulkan aplikácií. Výstupný program tvorí generátor kódu pre vrstvu, vrstva, ktorá zachytáva dáta ladeného programu a aplikácia zobrazujúca tieto zachytené dáta. To všetko s použitím C++, Vulkan API a knižnice ImGui.



# Abstract

The purpose of this bachelor thesis is to create a debugging tool for 2D/3D Vulkan applications, this tool can be useful for the veterans and the newcomers of this graphics API. The output program is composed of a code generator for the layer, the layer itself which collects the data from the currently analysed program and an application where the collected data are displayed. All this is made possible with the use of C++, Vulkan API and ImGui library.



# Kľúčové slová

Počítačová grafika, ladenie, ladiaci nástroj, C++, Vulkan API, Vulkan validačné vrstvy, ImGui

# Keywords

Computer graphics, debugging, debugging tool, C++, Vulkan API, Vulkan validation layers, ImGui


# Citácia

BILKO, Jozef. *Ladicí nástroj pro API Vulkan*. Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Pečiva, Ph.D.

# Ladicí nástroj pro API Vulkan

## Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Jána Pečiva Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Jozef Bilko
9. mája 2024


## Poďakovanie

Týmto ďakujem môjmu pánovi vedúcemu, Ing. Jánovi Pečivovi, Ph.D. za odbornú pomoc, férový prístup, otvorenosť a tiež i motiváciu, po celú tú dobu čo ma vediete. Tiež ďakujem aj rodine a priateľom, čo sa o mňa neustále zaujímajú, starajú a majú ma radi, budem sa snažiť Vám to v živote nejakým spôsobom všetko raz vrátiť.

# Obsah

**1 Úvod** **3**

**2 Počítačová grafika, Vulkan a ladenie** **4**
2.1 Úvod do počítačovej grafiky . . . . . 4
2.2 GPU, architektura a jej pamäťový model . . . . . 5
2.3 Vulkan API . . . . . 8
2.4 Vulkan vrstvy . . . . . 10
2.5 Hľadanie chýb a ladiace nástroje . . . . . 14

**3 Požiadavky a návrh riešenia** **18**
3.1 Zistenia a možnosti zlepšenia riešení . . . . . 18
3.2 Návrh riešenia ladiacej aplikácie . . . . . 19
3.3 Špecifické požiadavky pre riešenie . . . . . 20

**4 Implementácia nástroja pre ladenie** **22**
4.1 Štruktúra projektu a jeho setup . . . . . 22
4.2 Ladiaca Vulkan vrstva a zber dát . . . . . 23
4.3 Generovanie kódu, na základe XML . . . . . 26
4.4 Winsock, komunikácia medzi aplikáciami . . . . . 29
4.5 Stavová aplikácia a spracovanie dát . . . . . 30

**5 Testovanie a výsledky** **34**
5.1 Experimenty na grafických aplikáciach . . . . . 35
5.2 Vyhodnotenie ladiaceho nástroja VkDebugger . . . . . 41

**6 Záver** **42**

**Literatúra** **43**

**A Snímok obrazovky nahrania práce na internet** **45**

**B Obsah priloženého pamäťového média** **46**

**C Postup pre spustenie programu VkDebugger** **47**

# Zoznam obrázkov


2.1 Vykresľovací reťazec . . . . . 5
2.2 Zjednodušená logická pipeline modernej Nvidia GPU architektúry . . . . . 6
2.3 Zjednodušená ukážka pamäťového modelu GPU . . . . . 7
2.4 Hierarchia rozdelenia Vulkan aplikácie . . . . . 9
2.5 Architektúra Vulkan Loader Interface . . . . . 10
2.6 Priebeh volania *VkCreateInstance* a na úrovni aplikácie pri inicializácií vrstiev 11
2.7 Vrstva na úrovni volania funkcií . . . . . 12
2.8 Vulkan Configurator . . . . . 13
2.9 Beh programu s defektom . . . . . 14
2.10 Ladenie 3D Vulkan aplikácie v programe RenderDoc . . . . . 16
2.11 Ladenie 3D Vulkan aplikácie v programe Nsight Graphics . . . . . 17

3.1 Návrh architektúry pre ladiaci nástroj . . . . . 19
3.2 Návrh modelu komunikácie medzi vrstvou a stavovou aplikáciou . . . . . 20
3.3 Flexibilná vrstva (layer.cpp) . . . . . 21

4.1 Vzor behu zachytenej funckie na ladiacej vrstve . . . . . 24
4.2 Podvrhnutie Vulkan volania na úrovni vrstvy . . . . . 26
4.3 Štruktúra generovaných súborov . . . . . 27
4.4 Obsah vzoru vygenerovanej funkcie . . . . . 28
4.5 Konečný automat prijímania správ . . . . . 30
4.6 Architektúra spracovávania dát aktuálneho stavu ladenej aplikácie . . . . . 32
4.7 Vykresľovanie aktuálneho stavu ladenej aplikácie . . . . . 33

5.1 Výsledný ladiaci nástroj VkDebugger v použití . . . . . 34
5.2 VkDebugger s obmedzenou funkčnosťou . . . . . 35
5.3 Neúspešné návratové hodnoty Vulkan volaní . . . . . 36
5.4 Vzťah medzi VkMemory a VkBuffer objektmi . . . . . 37
5.5 Chyba aplikácie pri behu . . . . . 37
5.6 Aplikácia generujúca bitmapový obrázok . . . . . 38
5.7 Aplikácia s využitím VS breakpoint funkcionality . . . . . 39
5.8 Uvoľnenie zdrojov . . . . . 40
5.9 Nesprávne vykresľovanie v nástroji VkDebugger . . . . . 40

A.1 verejne dostupný kód projektu na repozitári GIT . . . . . 45

# Kapitola 1

# Úvod

Táto práca je zameraná na problematiku ladenia 2D/3D grafických aplikácií. Aplikácie tohoto typu majú za úlohu zobrazovať obrázkové prvky na monitor alebo iné výstupné zariadenie. Nie je to ale nič zanedbateľné – tento softvér nachádza svoje využitie napríklad v modernej medicíne, hernom, filmovom priemysle a i v mnoho ďalších oblastiach. Popri vývoji, programovaní takýchto aplikácií často dochádza k chybám alebo neefektívnemu návrhu zo strany programátora. Pre pomoc pri riešení týchto problémov sa používa softvér známy aj ako „ladiace nástroje“.

Konkrétne je práca zameraná na podmnožinu grafických aplikácií – ide o tie, ktoré využívajú Vulkan API. Táto čoraz viac populárnejšia technológia/rozhranie dovoľuje programátorovi efektívne využívať hardvér grafickej karty tak, že programátor popíše dopodrobna čo sa má vypočítať, nechá to zaslať na ňu a ona to spracuje.

Hoci Vulkan robí tento proces komplikovaným, výsledkom je možnosť tvoriť vážne efektívny grafický softvér. Pretože je to tak komplikované, je vhodné pri práci využívať nejaký ten ladiaci nástroj. Jeho úloha spočíva, okrem iných využití, najmä v zobrazovaní aktuálnych informácií o stave ladenej aplikácie.

Cieľom práce je teda nájsť nedostatky v dnešných ladiacich softvéroch za účelom implementovať vlastný nástroj. Hoci je nevýhodou že tento nástroj bude použiteľný výhradne iba pre Vulkan aplikácie, je kľúčové to vyvážiť nejakou výhodou na iných miestach.

Úspech cieľa práce teda určite závisí na mnohých aspektoch. Hneď v kapitole 2 bude priblížená problematika počítačovej grafiky, moderný hardvér zameraný pre jej počítanie, rozhranie Vulkan vrátane jeho riešenia pre validáciu a ladenie spolu s modernými riešeniami. V nasledujúcej kapitole 3, sú definované bližšie podmienky, návrhy, nápady pre riešenie implementácie vrátane identifikácie nedostatkov dnešných riešení pre ladenie. To všetko je postavené na základe vedomostí a skúseností získaných z predchádzajúcej kapitoly. Definovaný návrh potom bude implementovaný, popis práce, problémov a riešení sú viac rozvinuté v kapitole 4. Výsledok je potom nasadený do praxe testovaním na rade Vulkan programov, ale o tom je možné sa dočítať viac v kapitole 5.

Osobne som si túto tému zvolil z dôvodu, že chcem aj ja prispieť k lepším počítačovým zážitkom, ale ktovie čo môže priniesť budúcnosť – možno príde i deň, kedy ľudia budú tráviť viac času vo virtuálnej realite ako v reálnom svete.

# Kapitola 2

# Počítačová grafika, Vulkan a ladenie

Pre vhodný návrh a implementáciu ladiaceho nástroja je dobré vedieť, čo je to počítačová grafika, teda čo sa v počítači stane pred tým ako sa niečo ukáže na monitore. Užitočné sú rovnako aj znalosti o chovaní hardvéru, ako to tam vyzerá a prečo vôbec je dobré mať takú súčiastku v počítači.

Programovanie ale nezačne až pokým sa bližšie nepopíše, čo grafické rozhranie Vulkan je a hoci dôležité mať na pamäti aj jeho silné a slabé stránky, celá táto práca je práve postavená na funkcionalite ktorú Vulkan ponúka – Vulkan vrstvy (ďalej už iba ako vrstva/vrstvy)[1].

Síce tieto vedomosti by už stačili k vytvoreniu niečoho funkčného – nebolo by to dostačujúce k tomu aby práca bola zmysluplná - z toho dôvodu je v poslednej časti tejto kapitoly viac o súčasnom stave verejne dostupných ladiacich nástrojov pre určenie takého merítka pre túto bakalársku prácu.

Je dôležité ale podotknúť že nejde o encyklopedický prehľad, preto je to všetko obmedzené iba na tie najrelevantnejšie poznatky týkajúce sa tejto práce.

## 2.1 Úvod do počítačovej grafiky

Na počítačovú grafiku je možné sa pozrieť dvomi spôsobmi:

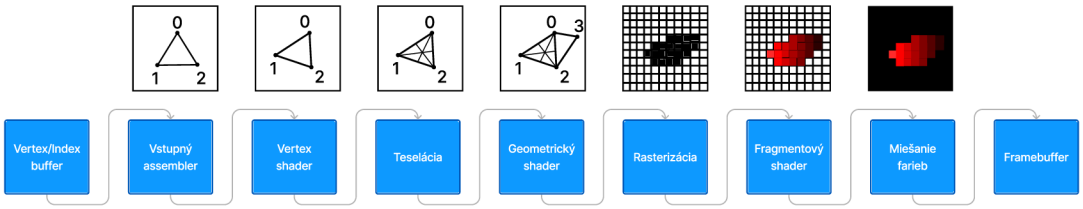
- Je to veda zaoberajúca sa vizuálnou komunikáciou prevažne medzi počítačom a užívateľom – využívajú sa tu napríklad aspekty fyziky, matematiky, ľudského vnímania a grafického dizajnu.
- Je to spôsob akým sa popisujú 2D/3D telesá v priestore, svetlo a iné javy ovplyvňujúce ich výzor, a následné vytvorenie reprezentácie vzhľadu tejto scény.

Takto je to o grafike predložené v [1]

Je dôležité ešte podotknúť, že v dnešnej dobe je táto disciplína využívaná vo filmovom, hernom alebo i reklamnom priemysle, preto stojí za to sa ňou zaoberať.

Najjednoduchšie by sa dalo povedať, že zmyslom celého tohoto orchesteru je vlastne iba kreslenie trojuholníkov na displej monitoru. Ale ako na to?

[1] Vulkan layer/s - Vulkan vrstva/y




Obr. 2.1: Vykresľovací reťazec[2]

[1] Obvykle sa používa abstraktný koncept označovaný ako **Vykresľovací reťazec**[3], body zodpovedajú vykonávaným krokom nad dátami telesa ktoré má stroj **Vykresliť**[4] na monitore v podobe **pixelov**[5].

- Vertex/index buffer, zhromaždené dáta pripravené na rendering (napr. vrcholy trojuholníku)
- **Vstupný assembler**, načíta zhromaždené dáta a vytvorí z nich tzv. primitíva
- **Vertex shader**, vezme vrcholy z primitív a vykoná nad nimi matematické operácie (napr. otočenie, posun)
- Teselácia, proces, kedy sa dáta daného útvaru/primitívu rozdelia na menšie za účelom vytvoriť detajlnejší vzhľad (napr. tehly na stene vyzerajú ako keby vyčnievali hoci je to zcela plochý povrch)
- Geometrický shader, nástroj ktorý nad každým primitívom dokáže vykonávať programovateľné operácie (napr. generovanie nových primitív)
- **Rasterizácia**, proces, kedy sa primitíva nanesú na pixely vznikájú tak zhluky pixelov ktoré sa nazývajú fragmenty
- **Fragment shader**, fragmentom sa priradia farby a hodnota hĺbky, na základe ktorej sa bude určovať ktorý pixel sa bude zobrazovať na povrchu (napr. ak sa biely a čierny pixel prekrývajú)
- Miešanie farieb, je to prípad ak viac pixelov iných farieb sa prekrýva a mení to farbu daného pixelu (napríklad ak je vrchný pixel čiastočne priehľadný a pod ním je iný)
- **Framebuffer**, sem sa ukladajú obrázky pripravené k vykresleniu, bežne napríklad v podobe sekvencie pixelov ktoré a ako majú svietiť

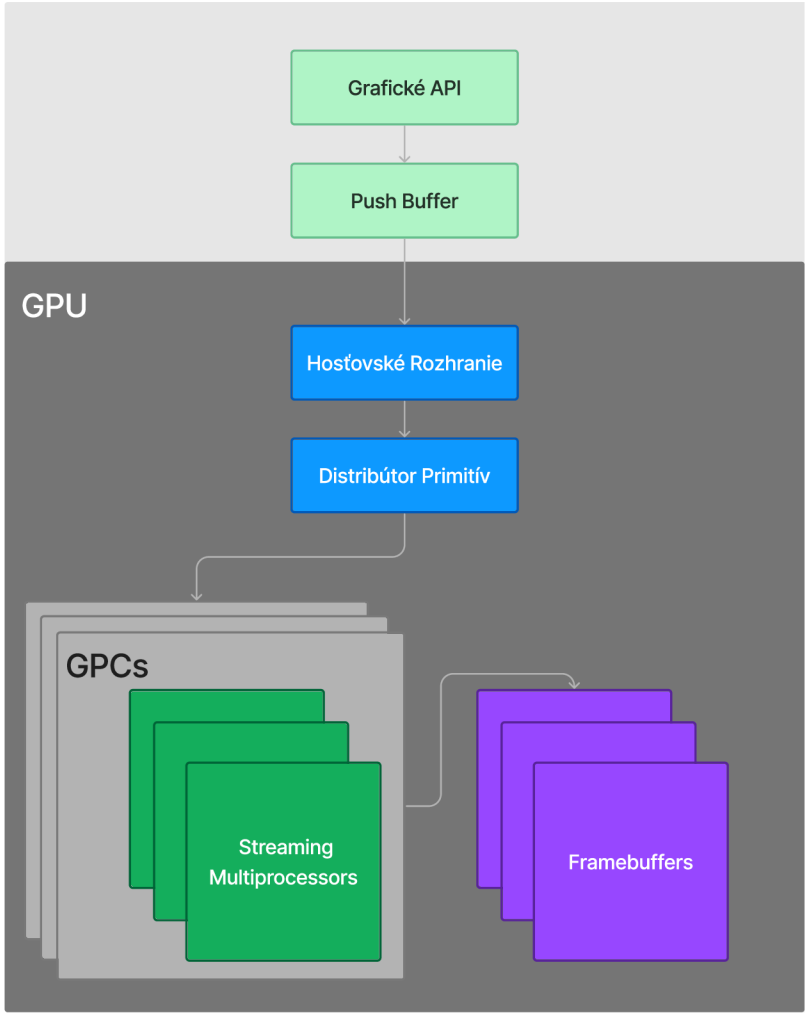
## 2.2 GPU, architektura a jej pamäťový model

Ako už bolo naznačené v kapitole 1, vykresľovací reťazec je šikovný spôsob ako niečo vykresliť na monitor, hoci na to postačí i CPU, z dôvodu náročnosti požiadavkov na rýchlosť

[2] Prevziaty z [11], sekcia Drawing a triangle/Graphics pipeline basics/Introduction
[3] vykresľovací reťazec – graphics pipeline
[4] vykresliť – render, rendering
[5] Pixely – body, z ktorých sa skladá displej monitoru

by to nebolo ideálne. Za posledné roky sa totiž objavuje čo raz viac grafických aplikácií alebo technológií s využitím v medicíne, vo filmovom priemysle – konkrétne napríklad v prípade mnoho video hier sa využíva real-time rendering[6], nároky na takúto vymoženosť sú v radách miliard pixelov za sekundu[12].

Veľký dopyt teda logicky dal vniknúť novému silikónového mikroprocesoru, ktorý sa označuje ako Grafický procesor (Ďalej už len ako GPU)[7]. Zásádný rozdiel medzi ním a jeho príbuzným CPU je ten, že GPU je hoci menej flexibilný, sila je v jeho špecializácií na paralelné operácie nad dátami.[8] Túto paralelnosť je možné vážne dobre využiť v počítačovej grafike kedy potrebujeme nad ohromným množstvom dát (trojuholníky, čiary, body, fragmenty) vykonať jednoduché operácie – tu, priamo v GPU je koncept vykresľovacieho reťazca implementovaný.




Obr. 2.2: Zjednodušená logická pipeline modernej Nvidia GPU architektúry[9]

Architektúru moderných GPU sa dá jednoducho pochopiť podľa schéma uvedené na Obr. 2.2:

[6] Vykresľovanie snímkov v reálnom čase

[7] Graphics processing unit

[8] Pozn. dnes už ale možno využívať aj viac ako iba na grafické operácie viac na https://en.wikipedia.org/wiki/CUDA

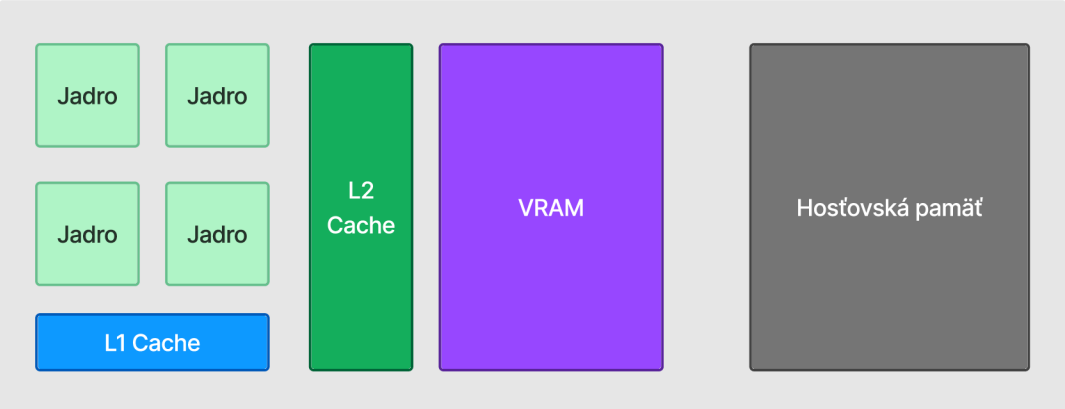
[9] Zjednodušený model architektúry Maxwell (2014) previaty z [10]

- Program zavolá príkaz na **Grafické API**, tieto príkazy sa ukladajú na **Push Buffer** vo formáte ktorý GPU vie čítať
- Po určitom čase alebo volaním *flush* zašle driver na **Hosťovské Rozhranie** prácu, ktorú **Distribútor Primitív** potom rozdelí medzi všetky dostupné **GPC**[10]
- V GPC sa rozdelí práca medzi viac jednotiek nazývaných **Streaming Multiprocessor** – kde dochádza k obsluhe dát paralelne (napríklad posun trojuholníka)
- Hotový frame[11] sa ukladá do **Framebufferu**

Bližšie detaily je možné nájsť na [10].

Táto problematika nie je ale až taká jednoduchá, komplikuje to obsluha pamäti GPU. Dôvodom existencie tejto pamäti je výkon – všetko aby bolo vypočítané v rozumnom čase, teda čím bližšie je pamäť k počítacej jednotke tým lepšie (toto je kľúčové ak sa snažíme pracovať s veľkým množstvom dát – napríklad aplikovanie textúr).

Dobrým príkladom je rada akcelerátorov AMD Instinct, explicitne sa rozlišuje **hosťovská pamäť**[12] a **pamäť zariadenia**[13] (Rovnako to takto delí aj Vulkan API – pozri 2.3). Až na malé výnimky platí, že hosťovská pamäť existuje mimo GPU a pamäť zariadenia výhradne v GPU [7]. Teda ak nejde o tzv. integrovanú grafiku, kedy je pamäť zdieľaná a nie je medzi ňou takýto rozdiel.




Obr. 2.3: Zjednodušená ukážka pamäťového modelu GPU[14]

Pamäť zariadenia sa obvykle konkretizuje na menšie časti, príklad takého rozdelenia na moderných GPU sa dá vyčítať z Obr. 2.3. Kedy určitý počet jadier číta dáta priamo z **L1 Cache**, v prípade, že tam dáta nie sú, zahajuje sa pokus o nájdenie dát v **L2 Cache**. Ak sa podarí, poskytnú sa nájdené dáta **Jadrám** do **L1 Cache**. V prípade neúspechu by sa pokračovalo na **VRAM**[15] a v najhoršom prípade až do Hosťovskej pamäti.

Čo ale komplikuje riešenie, obecne moderné GPU môže byť postavené na jednom z troch variant správy jeho pamäti:

[10] GPC – Graphics Processing Cluster
[11] Frame – obrázok pripravený k vykresleniu
[12] Host memory
[13] Device memory
[14] Konkrétne ide o model Nvidia GeForce RTX 40 Series GPU [9]
[15] VRAM – Video Random Access Memory

- **Koherentná pamäť**, kedy zmena dát na jednom mieste sa odrazí na všetkých replikách týchto dát
- **Nekoherentná pamäť**, zmeny sa neprejavujú: napríklad dáta na L2 Cache môžu byť iné ako tie na VRAM
- **Pamäť s obmedzeným prístupom**, koherentnosť je tu zachovaná vynútením nejakého obmedzenia, napríklad dáta obrázku sú iba *read only* [1]

## 2.3 Vulkan API

Vulkan je *open source* a *cross-platform* grafické rozhranie (2016) – obecne sa pokladá za následníka OpenGL (1992). Síce spôsob akým sa s nimi pracuje je odlišný, podstata je rovnaká, programujú sa s nimi predovšetkým grafické aplikácie.

Dôvodov, prečo sa dnes používa, je mnoho – na rozdiel od OpenGL napríklad dovoľuje programátorovi väčšiu kontrolu nad GPU. Okrem toho je Vulkan postavený a vyvíjaný mysliac na moderné GPU technológie, ktoré mnohé z nich ešte za čias skorého OpenGL neexistovali [5]. Avšak ako kompromis za také prednosti má programátor aj väčšiu zodpovednosť – napríklad pri nesprávnej práci s pamäťou môže bez vysvetlenia nastať pád programu. Z dôvodu náročnejšej krivky učenia sa preto doporučuje mnohé koncepty v počiatočných fázach učenia iba preskočiť a vrátiť sa k ním neskôr [14].

Hoci je náročné začať, oplatí sa poznať konvenciu [11] , ktorú sa obecne odporúča dodržiavať pri používaní tohoto API, rovnaká konvencia je potom využívaná ďalej v práci:

```
1  VkXXXCreateInfo createInfo{};
2  createInfo.sType = VK_STRUCTURE_TYPE_XXX_CREATE_INFO;
3  createInfo.param1 = ...;
4  createInfo.param2 = ...;
5
6  VkXXX object;
7  if (vkCreateXXX(&createInfo, nullptr, &object) != VK_SUCCESS)
8  {
9      throw std::runtime_error("vulkan error");
10 }
```

Na prvom riadku je štruktúra ktorá definuje informácie o objekte (referuje sa ako *handle objektu*), ktorý budeme používať. Dôležitý je atribút štruktúry *„sType“*, podľa ktorého API identifikuje o aký objekt ide[16]. Riadok 6 deklaruje objekt, ktorý je potom na nasledujúcom riadku odkazovaný do volania *„vkCreateXXX“*, kedy sa Vulkan pokúsi niečo programátorom definované vykonať, v tomto prípade by zrejme išlo o vytvorenie objektu *„object“*. Podotýkam pokúsi, pretože operácia sa nemusí podariť z rôznych dôvodov (napríklad chyba programátora alebo hardvéru) – preto sa porovnáva s enumeračným typom *„VkResult*[17] *“*. Programátorovi síce API nijak nebráni robiť túto kontrolu, ale môže to viesť k problémom pri behu programu. Pre dodatočné kontroly a pomoc pri programovaní sa používajú tzv. *„Vulkan vrstvy“*, ale o tom viac až v kapitole 2.4.

Ako už bolo spomenuté, Vulkan definuje nie len volania, štruktúry, typy ale aj objekty, pre účely tejto práce je kľúčové poznať predovšetkým tieto objekty [13]:

[16] Vulkan sa neustále rozširuje o nové typy, náhľad aktuálneho zoznamu je možné nájsť napríklad na https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VkStructureType.html

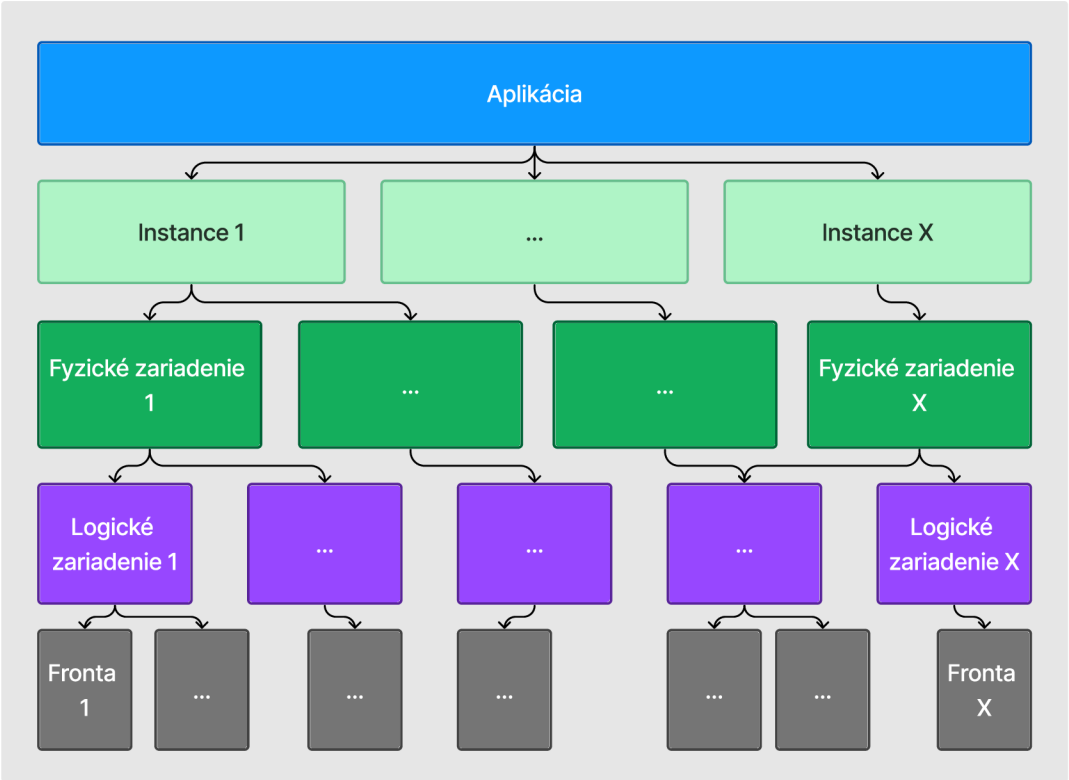
[17] https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VkResult.html

**VkInstance** – Softvérový konštrukt ktorý rozdeľuje stav jeden aplikácie od iných aplikácií na logickej úrovni. Jedna instancia môže mať viacero fyzických zariadení.

**VkPhysicalDevice**[18] – Ide o reprezentáciu kus hardvéru, ktorý môže Vulkan API použiť.

**VkDevice**[19] – Softvérový konštrukt, ktorý reprezentuje určité rezervované zdroje na fyzickom zariadení. Je možné, aby jedno fyzické zariadenie malo viacero objektov typu *VkDevice*.

**VkQueue**[20] – Softvérový konštrukt, ide o frontu, na ktorú chodia príkazy, teda že čo a ako sa má vykonať. Zariadenie to po zavolaní príkazu *vkQueueSubmit* nechá zariadeniu, ktoré to potom vykoná, avšak je dôležité podotknúť, že zatiaľ čo GPU počíta, program beží ďalej, preto sa využívajú rôzne spôsoby pre synchronizáciu ak je to potreba (napr. volanie *VkDeviceWaitIdle*).




Obr. 2.4: Hierarchia rozdelenia Vulkan aplikácie[21]

**VkCommandBuffer** – Objekt, v ktorom sa ukladá postup inštrukcií, *command buffer* sa potom odovzdá zariadeniu do jeho rady (*VkQueue*) aby sa vykonalo. Toto ukladanie inštrukcií sa nazýva „command buffer recording“, vo Vulkan API je ekvivalent príkazu začiatok a koniec *vkBeginCommandBuffer* a *vkEndCommandBuffer*.

**VkMemory** – Objekt s informáciou že kde a aká veľká je daná alokovaná pamäť, rozlišuje sa *hostovská pamäť* a *pamäť zariadenia*. Pamäť je možné vytvoriť s určitými parametrami, pomocou nich vie programátor určiť jej vlastnosti, napríklad pri nastavení flagu

[18] Fyzické zariadenie
[19] Logické zariadenie
[20] Fronta
[21] Prevzaté z [13]

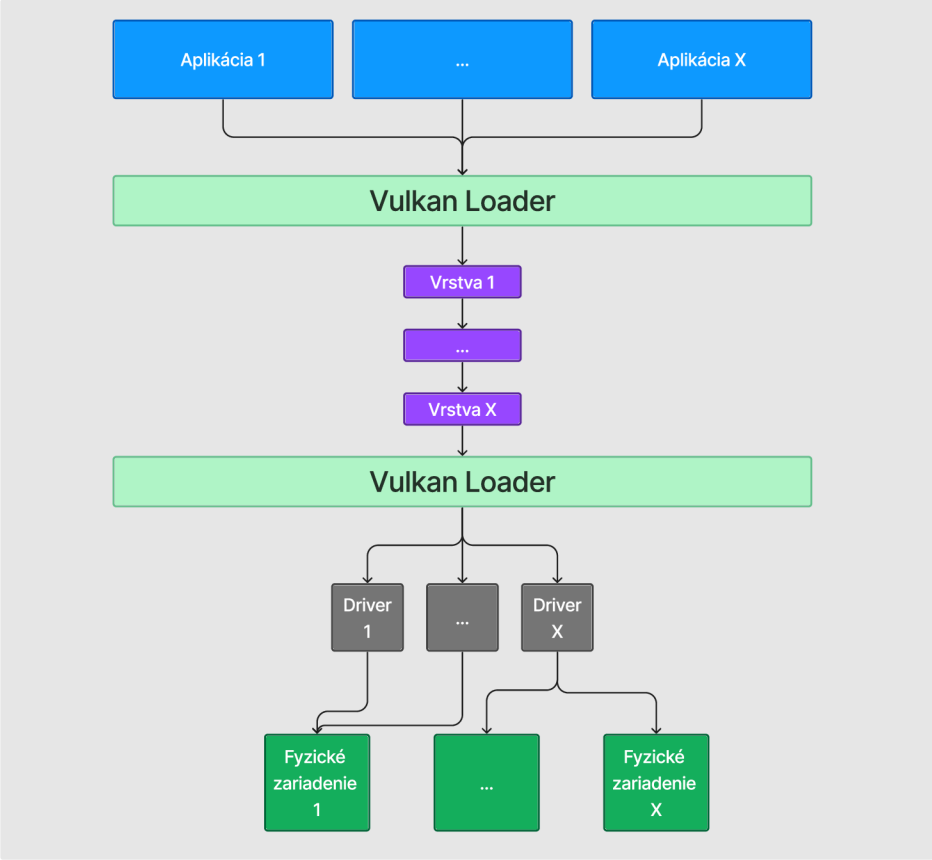
*VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT* je možné pristupovať a čítať obsah pamäti priamo z CPU (To je citlivé najmä v prípade, že počítač nemá integrovanú grafiku, teda ide o dve rôzne komponenty). Pamäť je možné potom spojiť s *buffer* alebo *image* objektom, potom daný objekt používa túto alokovanú pamäť pre operácie

**VkBuffer** – Objekt typicky používaný ako úložný priestor pre dáta, pre správne použitie sa spojí s objektom *VkMemory* a potom sa do neho vkladajú dáta. Obsah je možné potom kopírovať medzi image objektami a aj medzi ostatnými buffer objektami.

**VkImage** – Objekt podobný *VkBuffer* objektu v tom, že sa tiež spája s pamäťovým objektom, kam sa potom ukladajú dáta pre obrázok alebo textúru. Avšak, *VkImage* ešte uchováva dáta potrebné pre vykresľovanie, napríklad o aký formát ide a podobne.

## 2.4 Vulkan vrstvy

Z kapitoly 2.3 vyplíva, že Vulkan API z dôvodu byť čo najviac efektívny na GPU necháva validáciu na programátorovi. Hoci to prináša mnoho nepríjemností, ale už aj na toto sa myslelo, pre dodatočnú kontrolu a iné pomocné nástroje sa používajú *Vulkan vrstvy*[22] (Ďalej už iba ako vrstvy).

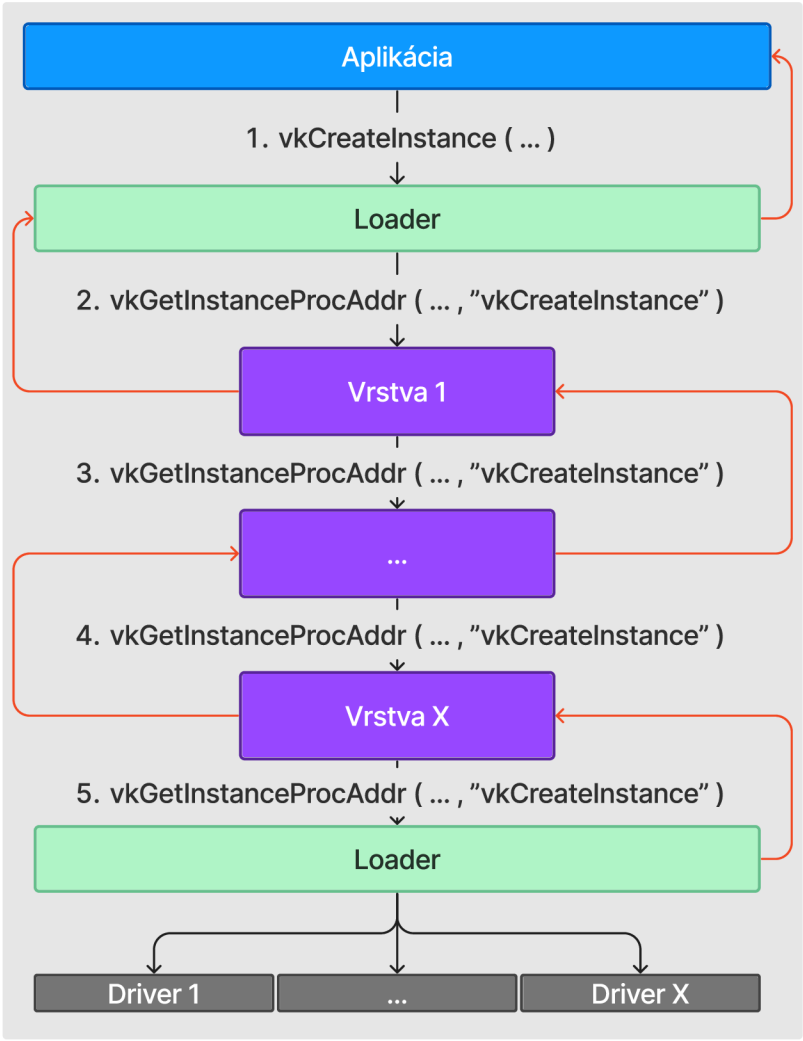



Obr. 2.5: Architektúra Vulkan Loader Interface[23]

[22] Vulkan layers
[23] Prevzaté z [4]

Vrstvy nekomunikujú s aplikáciou priamo, vidieť sa to dá podľa obrázku [4] – sú súčasťou *Vulkan Loader Interface* architektúry. Ak aplikácia zavolá Vulkan funkciu, prevezme ju **Loader**, ktorý ju (voliteľne) nechá ešte postupne najprv prejsť všetkými vrstvami, ktoré ju chcú zachytávať. Informácie ohľadom tohoto volania **Loader** napokon predloží dostupným **ovládačom**[24], tie potom pracujú s **fyzickými zariadeniami** aby vykonali prijaté požiadavky.

Spôsob, akým **Loader** vie, ktoré vrstvy zachytávajú ktoré funkcie spočíva v ukladaní odkazov na jednotlivé funkcie vo vnútri vrstiev do tzv. *dispatch tabuľky* (Ďalej už iba ako *DT*), ktorá sa inicializuje pri volaní funkcie *vkCreateInstancce* a vyzerá to takto:




Obr. 2.6: Priebeh volania *VkCreateInstance* na úrovni aplikácie pri inicializácií vrstiev[25]

1. Ladená **aplikácia** zavolá funkciu *vkCreateInstance*, ktorú prevezme **Loader**[26]

[24] Ovládač – driver
[25] Čiernou šípkou ide Vulkan funkcia smerom na ovládače, červenou sa vracia výsledok naspäť
[26] Loader trampoline function

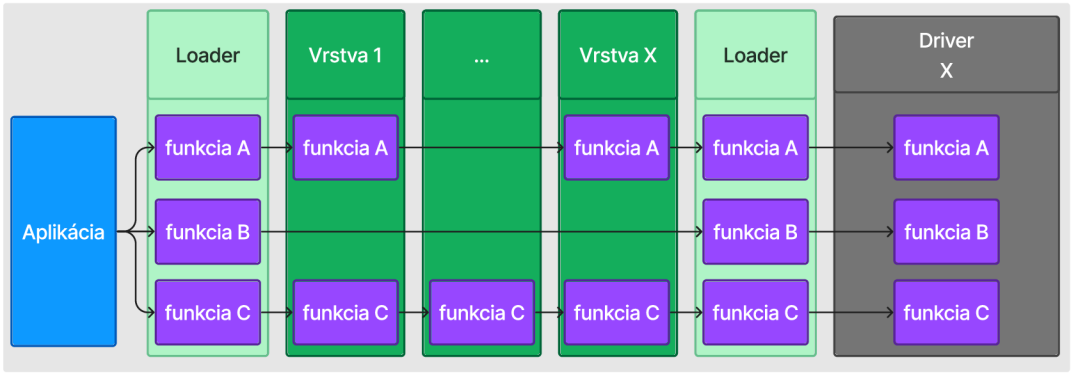
2. **Loader** na prvej vrstve zavolá *vkGetInstanceProcAddr* pre všetky Vulkan funkcie, ktoré ladená **aplikácia** pozná. Výsledkami týchto volaní si **Loader** vyplní štruktúru nazývanú *instance dispatch tabuľka* (Ďalej už iba ako *IDT*), a to nasledovne: V prípade, že **vrstva** chce túto funkciu zachytávať, tak je výsledkom volania *vkGetInstanceProcAddr* práve adresa na funkciu v programe tejto **vrstvy**. Inak táto prvá vrstva nechá zavolať túto funkciu na nasledujúcej **vrstve** a pokúsi sa adresu získať tam

3–4. Po vyplnení predchádzajúcej tabuľky sa proces zopakuje tak, že pre každú funkciu ktorú táto vrstva zachytáva si musí vyplniť vlastnú *IDT* rovnakým spôsobom – teda zavolaním *vkGetInstanceProcAddr*, s menom zachytávanej funkcie, do nasledujúcej **vrstvy**

5. Ak ide o poslednú **vrstvu**, tabuľku si vyplní adresami znovu pri volaní zachytených funkcií *vkGetInstanceProcAddr* tentokrát ale odkazujúce na **Loader** kód

Podľa vyplnených tabuliek sa potom pri volaní akejkoľvek Vulkan funkcie skáče na rôzne miesta v kóde. Napríklad: každá vrstva zachytáva funkciu *vkCreateInstance*, preto každá vrstva musí vedieť kam má po vykonaní svojej práce skočiť – jednoducho každá tabuľka týchto vrstiev obsahuje odkaz na adresu kam má po vykonaní svojej práce na vrstve skočiť ďalej.

Je dôležité ešte spomenúť, že sa rozlišuje medzi *device dispatch tabuľkou* (Ďalej už len ako *DDT*) a *instance dispatch tabuľkou*, kedy do *IDT* sa zapisujú adresy na*instance-specific* funkcie[27] a do *DDT* iba adresy na *device-specific* funkcie[28].




Obr. 2.7: Vrstva na úrovni volania funkcií[29]

Obrázok vyššie 2.7, popisuje ako to vyzerá keď sa zavolá Vulkan funkcia v prípade už inicializovaných *DT* – niektoré vrstvy zachytávajú niektoré funkcie, ako napríklad **Vrstva 1** a **Vrstva X** zachytávajú **funkciu A**, avšak obe vrstvy nijak nemodifikujú **funkciu B**. V prípade, keď sa funkcia na vrstve zachytí, vrstva môže meniť chovanie funkcie obojsmerne – teda jak cestou do ovládača tak cestou naspäť. Vrstva má taktiež aj prístup k informáciam ako napr. parametre funkcie čo môže využiť ako to programátor uzná za vhodné.

Zachytené dáta na vrstve potom možno spracovať rôzne, sú nástroje čo skontrolujú vstupné hodnoty a upozornia programátora ak je tam niečo čo nesúhlasí s platným používaním danej funkcie – napríklad je niekde nulový ukazovateľ i keď pre správny chod by

[27] Prvý parameter funkcie je buď *VkInstance*, *VkPhysicalDevice* alebo nič
[28] Prvý parameter je buď *VkDevice*, *VkQueue*, *VkCommandBuffer* alebo ich *child*
[29] Prevzaté z [3]

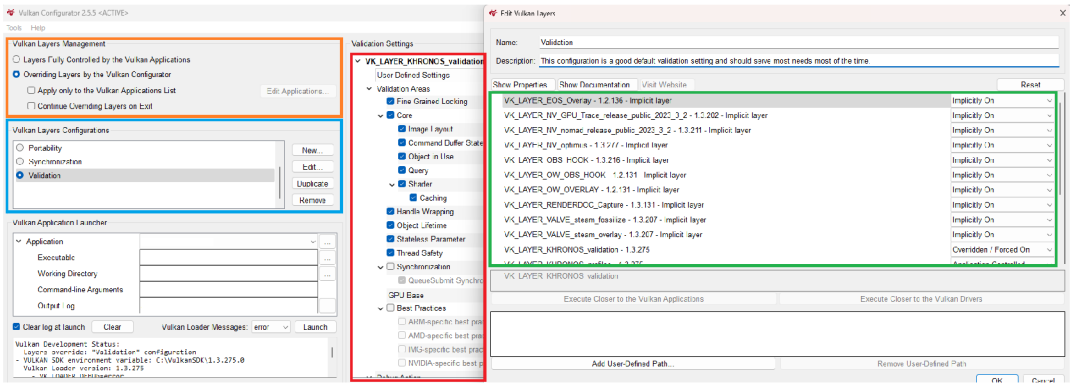
tam niečo malo byť. Funkcionalita vrstiev nie je obmedzená iba na kontrolu, sú nástroje čo dokážu omnoho viac, napríklad i zaobstarávanie snímkov z aplikácie[30].

Pre lepšiu predstavu: tento zoznam zhrňuje, aké dáta z ladenej aplikácie sa na vrstve zachytávajú[31]:

- Meno a návratový typ volanej funkcie
- Parametre funkcie, vrátane ich hodnôt
- Výsledná návratová hodnota vykonanej funkcie

Výhoda týchto vrstiev spočíva hlavne v tom, že ide o modul, ktorý je možné vypnúť alebo zapnúť za cieľom efektívnejšieho programovania, preto napríklad počas vývoja aplikácie je modul aktívny ale keď sa ide aplikácia už vydať, modul stačí deaktivovať. Konkrétne je takýto modul/vrstva implementovaná ako *knižnica*[32].

Ako už bolo naznačené, vrstvy môže nechať aktivovať či už testovaná aplikácia, alebo iné nástroje – čo sa týka nástrojov, [15] spoločnosť LunarG vyvinula program „Vulkan Configurator“ ktorý dovoľuje aktivovať vrstvy bez ohľadu na to, či si to aplikácia vyžiada alebo nie. Nástroj vezme teda *knižnicu vrstvy* a vrstvu aktivuje ak si to aplikácia vyžiada.[33]




Obr. 2.8: Vulkan Configurator[34]

Vyššie na obrázku 2.8 sú farebne oblasťami vyznačené kľúčové časti programu:

- oranžová – ovládanie možnosti aktivovania vrstiev pomocou *VkConfig*
- modrá – výber konfigurácie vrstiev pre aktuálne použitie
- červená – nastavenie parametrov jednotlivých vrstiev
- zelená – nastavenie konkrétnej konfigurácie

[30] https://github.com/LunarG/VulkanTools/blob/main/layersvt/README.md
[31] Pozn. na základe týchto dát sa dá vyčítať, že aký je stav aplikácie
[32] Napr. teda ako *.dll*
[33] Pozn. ak je vrstva aktivovaná na Vulkan aplikácií, inicializuje sa zavolaním funkcie *VkCreateInstance*
[34] https://github.com/LunarG/VulkanTools/blob/main/vkconfig/README.md

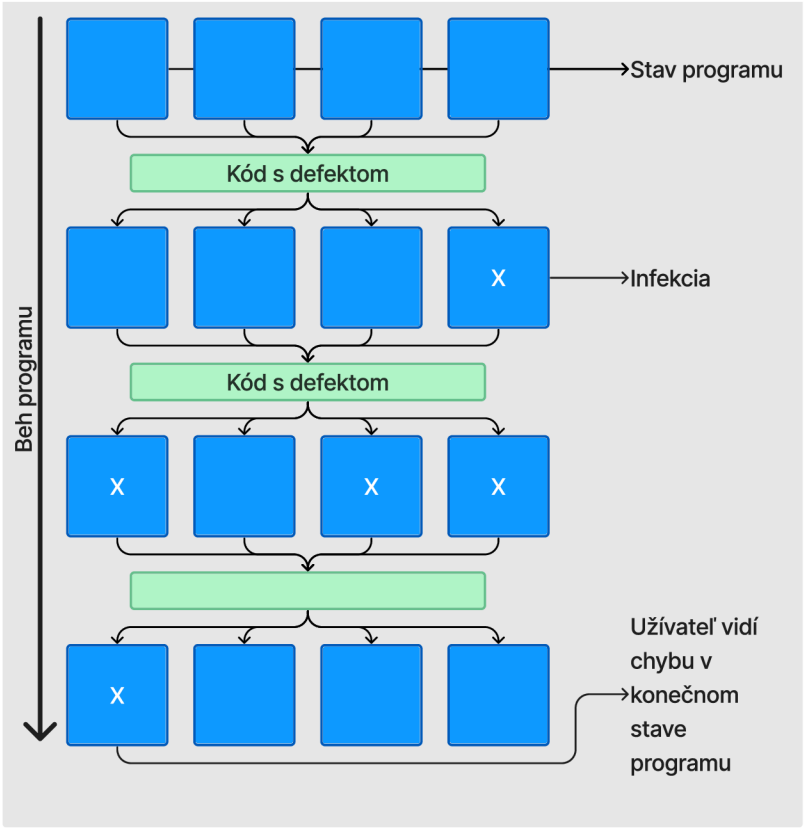
Čo sa týka konfigurátoru, tak pri výbere konfigurácie je možné si vytvoriť vlastnú ktorá sa bude skladať napríklad z 3 rôznych vrstiev. Konfigurácia sa nastavuje po stlačení tlačítka *new* alebo *edit*. Pri spustení vrstvy si ich vrstva prečíta a spracuje podľa vlastného uváženia.[35]

## 2.5 Hľadanie chýb a ladiace nástroje

Chyby sú temer vždy súčasťou programovania, čo dáva ale často zabrať je časová náročnosť programátora aby ich vôbec našiel – pre uľahčenie tejto bolesti boli vyvinuté nástroje čo majú v tomto prípade pomôcť.

### Chyby pri programovaní

Chyba ako taká je je príliš široký pojem, pre lepší popis tejto problematiky sa používa pojem **defekt** [16] – označenie miesta zdroja chyby v kóde. V prípade, že program sa vykoná aj v mieste defektu, je možné, že to spôsobí tzv. **infekciu**. Tým sa myslí to, že nejaká premenná alebo stav programu obsahuje nejakú neočakávanú hodnotu. Infekcia ako taká ešte ale nemusí vyvolať pád aplikácie, no môže uviesť program do nežiadúceho stavu, napríklad že chceme aby nakreslil modrý trojuholník ale nakoniec je červený – no pokojne by to mohlo viesť i k pádu programu.




Obr. 2.9: Beh programu s defektom[36]

[35]Pozn. nie je potreba spúšťať aplikáciu pomocou VkConfig v prípade, že vrstva v konfigurácií má zaškrtnutú možnosť *Overridden / Forced On*

Na obrázku 2.9 je zobrazený tzv. **infekčný reťazec**, kde stav programu je postupne zamorený infekciami na rôznych premennými. No v konečnom stave programu vidí užívateľ problém na jednom mieste a nemusí byť potom zrejmé kde defekt hľadať. Z toho dôvodu je pri ladení veľmi nápomocné program **krokovať**[37].

Podľa [16] ale iba samotný ladiaci nástroj nestačí, kľúčové je vedieť ako hľadať infekcie a dedukovať z nich kde asi defekt nastal. Spôsoby ako na to je mnoho, nižšie je bližšie popísaný veľmi obecný a dobre zaužívaný postup ako na to:

1. Spustiť program s vstupnými hodnotami čo vyvolali *infekčný reťazec*
2. Vystopovať infikovanú hodnotu a získať set hodnôt od ktorých závisí
3. Pozrieť sa na individuálne hodnoty tohoto setu a rozhodnúť, či je hodnota infikovaná alebo nie
4. Ak je hodnota infikovaná, zopakovať kroky 2 a 3 na tejto hodnote
5. V prípade, že žiadna z infikovaných hodnôt nie je infikovaná, našiel sa defekt
6. Opraviť defekt

Síce jednoduchá, ale efektívna metóda – ladiaci nástroj by mal dopomôcť vykonávať metódy ako je táto. Rovnaký postup je možné aplikovať aj pri hľadaní chyby grafických aplikácií.

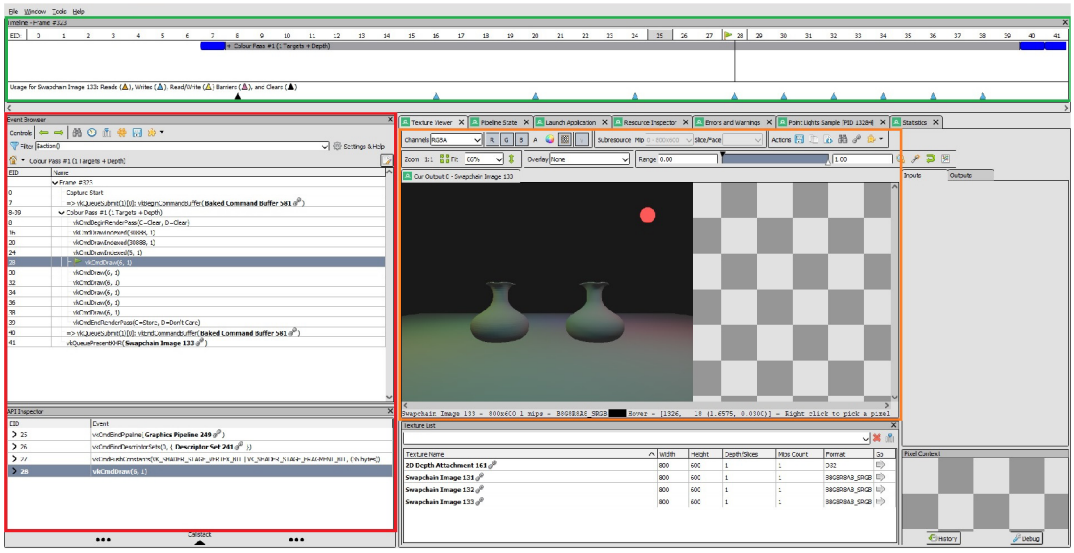
Čo ale je možné a žiadúce pri nástroji pre ladenie grafických aplikácií je hodnoty zobrazovať v grafickej podobe [6], napr. obsah textúry zobraziť rovno ako pixely na monitore, užívateľovi sa lepšie potom hodnoty čítajú, čo uľahčuje prácu.

### Moderné nástroje pre ladenie

Podobné riešenia pre ladenie grafických aplikácií už existujú, je kľúčové ich preskúmať a všimnúť si najmä kde a čo sa dá urobiť lepšie.

**Renderdoc** – [2] použitie spočíva v určení kedy alebo ktorý *frame* zachytiť a ten je možné bližšie analyzovať. Medzi možnosti ako pracovať patrí najmä *Event Browser*, kedy *frame* je rozdelený na príkazy postupne volané pomocou API. Pri otvorení eventu sa v okne *API inspector* umožní prehliadnuť parametre, postupnosť a následne ich napríklad aj filtrovať. Dá sa tiež odkazovať sa na rôzne objekty ktoré majú s týmto volaním spoločné a rovnako analyzovať.[38]

[36] Prevzaté z [16], kapitola 1
[37] Krokovať – zastaviť program po určitej podmienke, napr. čas, kus kódu alebo po určitej funkcií
[38] Pre predstavu, ide o červene vyznačenú sekciu na obrázku 2.10



Obr. 2.10: Ladenie 3D Vulkan aplikácie v programe RenderDoc

Ako už bolo vyššie uvedené 2.5, dáta grafických aplikácií sa oplatí zobrazovať vizuálne, *RenderDoc* toto dokáže v podobe vykresľovania obrázku ktorého dáta sa v aplikácií posielajú do GPU (Oranžová oblasť). Tento obrázok je možné zobraziť v rôznych štádiách vykresľovania pomocou tzv. *frame timeline*. Konkrétne podľa obrázku sa to dá predstaviť jednoducho – v evente č.24 ešte nie je zobrazená červený kruh v pravom hornom rohu, ale už v evente č. 28 tento kruh vidieť je (Možno posúvať v zelenej oblasti).

Pomocou postupnej analýzy programu takto *frame* čo *frame* vie rozumný programátor odhaliť defekt.

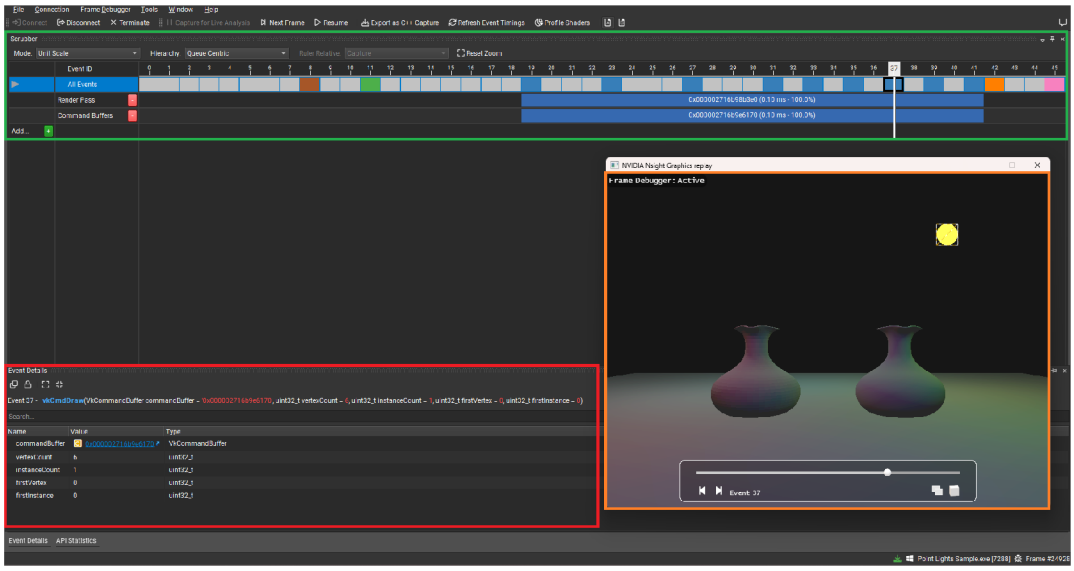
Cenu za čo *RenderDoc* platí je to, že má zabudovaný limit nahrávania z dôvodu veľkého množstva informácií ktoré pri zachytávaní zhromažďuje – každý *frame* musí byť možné analyzovať samostatne, preto napríklad na mnoho miestach sú dáta totožné ako má jeden **frame** dozadu.

*RenderDoc* dovoľuje tiež ukladať tieto volania do súboru, neskôr nahrať a znovu spustiť. Tu ale treba byť opatrný, pretože niektoré chyby nemusí byť možné replikovať na základe takejto nahrávky. Ak by bolo treba, žiaľ by sa to programátor musel pokúsiť zachytiť znovu.

Ďalšia nepríjemnosť je v maličkosti ako je presnosť časových metrík jednotlivých príkazov. *RenderDoc* totiž meria čas podľa toho ako dlho to trvá pri vykonávaní príkazov po spustení nahrávky a nie čas ktorý by príkazu reálne trval vykonať na GPU[39] – preto hoci má program možnosť sledovať performance metriky GPU, nedoporučuje sa používať ho ako *profiler*[40].

[39] Bližšie informácie ohľadne toho v diskuzií https://github.com/baldurk/renderdoc/issues/2325

[40] Nástroj čo vie na úrovni GPU určiť kde a koľko práce sa vykonáva




Obr. 2.11: Ladenie 3D Vulkan aplikácie v programe Nsight Graphics

**Nsight Graphics** – [8] Podobne ako *RenderDoc* je možné pomocou tohoto nástroja vykonať analýzu *frame* čo *frame*. Znovu, červená oblasť pre informácie ohľadne príkazu, oranžová pre zobrazenie scény a zelená pre posun. Rovnako tiež dobre je implementovaný filter pre vyhľadávanie medzi volaniami, spúšťanie testovanej aplikácie sa dá rovnako ako to robí *RenderDoc*, buď sa aplikácia spustí pomocou *Nsight Graphics* alebo sa zavesí na bežiaci proces.

Kde sa programy líšia nie je možnosť zachytávať jeden *frame* po stisku tlačítka, ale zachytávať ich viac naraz – *Nsight Graphics* nemá zabudovanú funkcionalitu nahrávať ich na základe nejakej podmienky priamo v aplikácií ako *RenderDoc*, je potreba si to vyžiadať ručne v kóde prostredníctvom *NVIDIA Nsight Perf SDK*[41], hoci flexibilnejšie riešenie, nie je to nijak prívetivé.

V čom ale vyniká *Nsight* je možnosť profilovať aplikáciu, v skratke nástroj je nad aplikáciou a zachytáva využitie GPU počas behu, čo pomáha k odhaleniu chýb.

[41] https://developer.nvidia.com/nsight-perf-sdk

# Kapitola 3

# Požiadavky a návrh riešenia

Výstupom práce je ladiaci nástroj, ktorý bude zobrazovať stav analyzovaného 2D/3D Vulkan programu počas jeho behu a pomôže pri odhaľovaní chýb. Nejde ale o jednoduchú záležitosť, architektúra tohoto výsledku pozostáva z navzájom spolupracujúcich modulov.

## 3.1 Zistenia a možnosti zlepšenia riešení

Počas štúdie problematík v kapitole 2, konkrétne v podkapitole 2.1 som zistil, že presnejšie ide v tejto práci o analýzu naprogramovaných grafických aplikácií, ktoré sa snažia na monitor nechať vykresliť telesá definované v priestore alebo na ploche uložené v určitých dátových formátoch. Ak poznáme čo je za fungovaním, môžeme tak aj programátora nechať vedieť informovať o stave ladeného programu.

Pri štúdiu materiálov bližšie popísaných v kapitole 2.2 som sa dozvedel, že pri práci s modernými GPU sa musí dávať pozor na to ako sa s nimi zaobchádza, napríklad výsledky operácií na GPU trvajú nejaký čas, preto je tu potrebný mechanizmus ktorý by pomohol synchronizovať program ktorý beží na CPU a tým čo sa deje na GPU.

Vulkan má zopár dôležitých štruktúr (kap. 2.3) ktoré budú využité neskôr pri implementácií vrstvy a aplikácie. Rozhodol som sa podľa zistení z kapitoly 2.4 využiť vrstvu pre zachytávanie dát z ladeného programu – je vhodné z dôvodu, že *Vulkan API* práve na úrovni vrstvy ponúkne všetko čo bude na vytvorenie ladiaceho nástroja potrebné. Okrem toho, posledné verzie *Vulkan SDK* obsahuje nástroj *Vulkan Configurator* ktorý umožní využívať a konfigurovať túto vrstvu na testované aplikácie.

Kľúčové je ale poznať postupy (kap. 2.5) a spôsoby ktoré sa využívajú pri vyhľadávanie defektov v aplikácií, ide najmä o to zefektívniť prácu programátora ak sa akokoľvek dopracuje k chybovému behu programu ktorý práve vyvíja. Nápomocné mu bude vyjadrovať dáta vizuálne ak to bude možné pre zjednodušenie práce s nástrojom.

Pri bližšom skúmaní dnešných riešení pre ladenie aplikácií som zistil, že už iba proces hľadania chyby nie je vôbec prívetivý – napríklad pri použití programu *RenderDoc* musí programátor poznať či už časové rozmedzie alebo priamo *frame* kde chyba nastáva. A pri *Nsight Graphics* je to ešte horšie, pretože ak by mal programátor záujem pozrieť si viac ako jeden *frame* a aj ten nasledujúci, musel by využiť o prostredie naviac a spustiť nahrávanie v kóde.

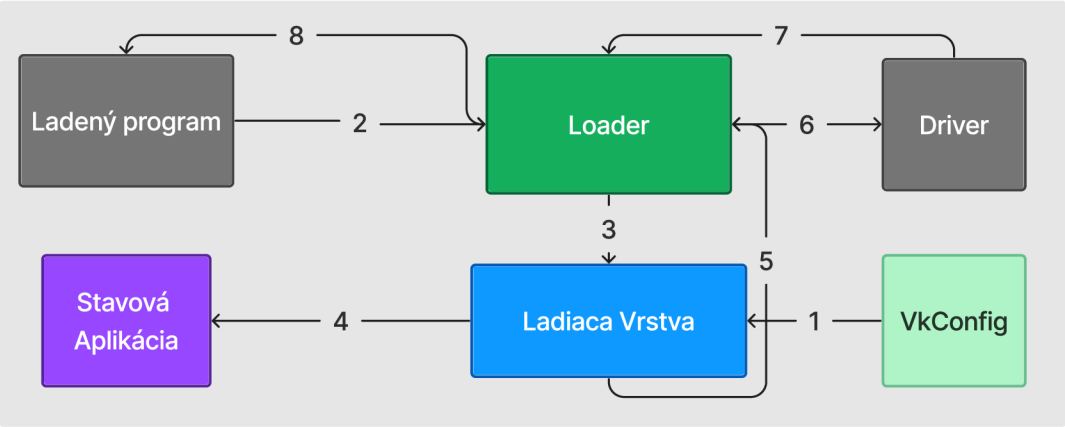
Ďalšia vec čo by pomohla by bola ak by bolo možné krokovať na základe nejakej bližšej podmienky napríklad programátor možno tuší, že chyba možno spočíva v moment kedy

vytvára textúru číslo X, tak by sa program zastavil vždy pri načítaní dát každej textúry – jednoducho nejaké viac špecifické podmienky ako krokovať je iba *frame* ID alebo čas.

Najviac pravdepodobné, že prečo je to takto obmedzené je neskutočné množstvo dát ktoré sa pri nahrávaní spracuje a ukladá. Moje riešenie ako tomu predísť a ponechať voľnosť ladiť priamo za behu je zariadiť, aby každá funkcionalitu (napríklad zhromažďovanie dát o obrázkoch) bude možné vypnúť a zapnúť na začiatku chodu programu.

## 3.2 Návrh riešenia ladiacej aplikácie

Celý návrh dobre vystihuje návrh architektúry ladiaceho nástroja s využitím Vulkan vrstiev:




Obr. 3.1: Návrh architektúry pre ladiaci nástroj[1]

1. **Vulkan Configurator** nechá nastaviť vstupné parametre pre vrstvu a označí ju ako Overriden / Forced On – to spôsobí aktivovanie vrstvy pri spustení akéhokoľvek **ladeného Vulkan programu** (s výnimkou **Stavovej Aplikácie**)
2. **Ladený program** pri spustení zavolá funkciu *VkCreateInstance*
3. **Loader** pošle zavolanú *VkCreateInstance* na **Ladiacu Vrstvu**. Tak sa zaháji inicializácia vrstvy vrátane prečítanie vstupných parametrov z nastavení v kroku 1
4. *Ladiaca vrstva*[2] pri inicializácií otvorí **Stavovú Aplikáciu** a rovno sa na ňu napojí aby jej mohla odosielať za behu zhromaždené dáta
5. Volaná funkcia vráti do Loaderu
6. Táto funkcia sa pošle na **Driver** (ovládač)
7. Nakoniec sa ide naspäť do Loaderu, znovu do ladiacej vrstvy a znovu do Loaderu (takýto priebeh majú všetky volania Vulkan API pre ladiacu vrstvu)
8. Výsledok funkcie a kontrola sa potom znovu vráti Ladenému programu

Prakticky potom pri ladení bude mať programátor otvorený *VkConfig* a program čo ladí. Ladenie sa začína pri spustení programu na základe vstupných parametrov pre vrstvu.

[2]V implementácií potom pomenováná ako aj VkDebuggerApp

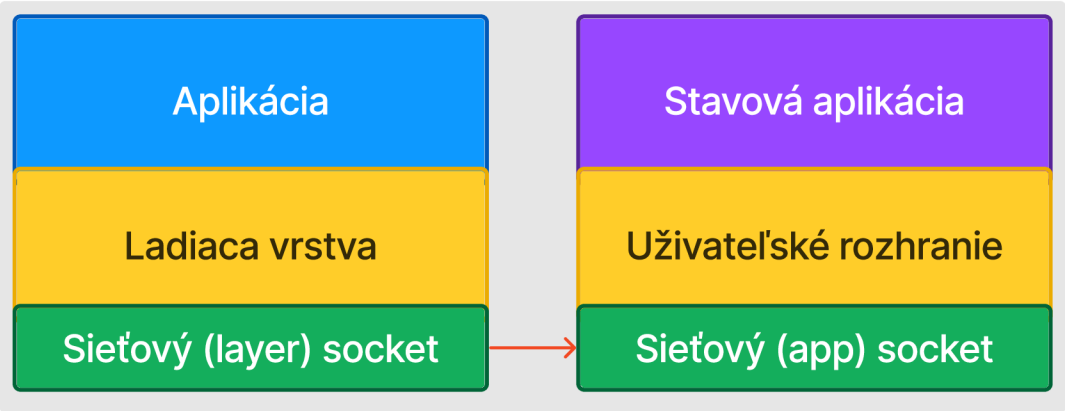
## 3.3 Špecifické požiadavky pre riešenie

Základom je zobrazovanie aktuálnych stavov týchto konštruktov: **API volania**, všetky volania API sú rozdelené spôsobom *frame by frame* vrátane ich parametrov. Je možné ich zoraďovať a filtrovať.

**VkMemory** a **VkBuffers**, oba objekty vo výslednej aplikácií možno rozlišovať, spájať medzi sebou, uvoľňovať, kopírovať – tieto stavy je možné sledovať. Okrem stavu je tu zobrazená veľkosť a aj ich obsah (Ak nejaký majú). Obsah je pravidelne aktualizovaný.

Pri **VkImages** tu platí rovnaký vzťah ako pri VkBuffers, je možné ju asociovať s pamäťou a už zmienené zmeny stavu čo tu môžu nastať. Rozdiel je ale v tom, že je dôležité uchovávať aj dimenzie obrázku a schopnosť programu na základe toho obrázok aj nechať vykresliť pre lepšiu orientáciu v programe.

**Vstupné parametre**, ktoré umožňujú krokovať program alebo posielať upozornenia na základe rôznych podmienok. Vstupné parametre môžu nechať čokoľvek vyššie deaktivovať.

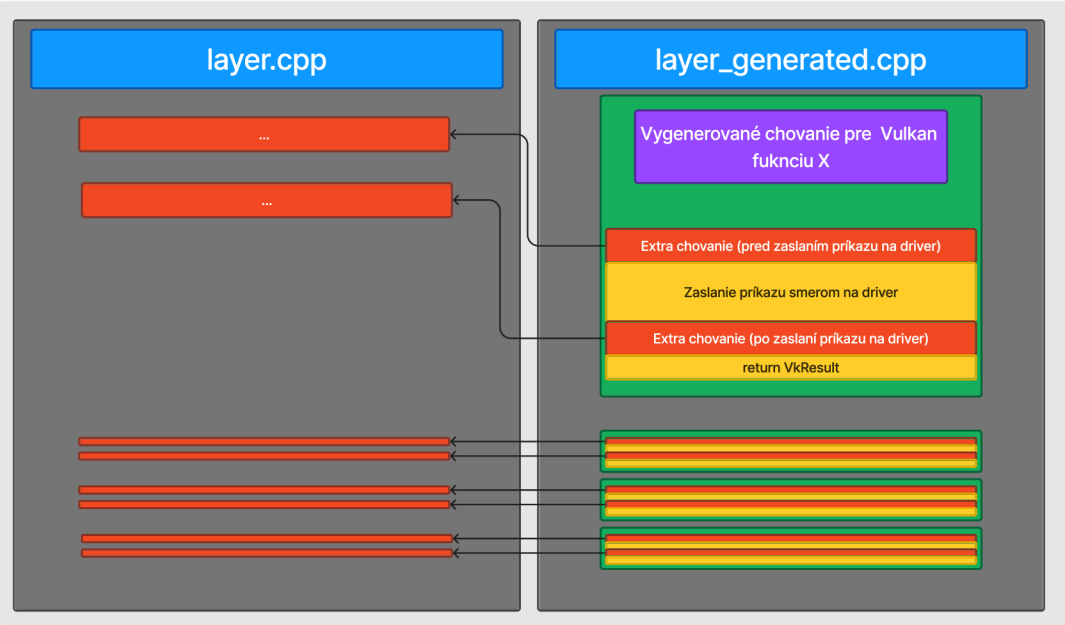



Obr. 3.2: Návrh modelu komunikácie medzi vrstvou a stavovou aplikáciou

**Komunikácia medzi vrstvou a aplikáciou** bude zaisťovať komunikácia TCP prostredníctvom sieťových socketov, príde to vhod, pretože v prípade pádu aplikácie sa ukončuje aj vrstva – vtedy by ladiaci nástroj prišiel o informácie, že v akom stave bola aplikácia v dobe pádu. Oddelená architektúra zaistí možnosť si tento stav aplikácie v tomto kritickom bode zobraziť i keď vrstva zanikne, toto je pri ladení veľmi užitočné. Okrem toho, vlastnosťou TCP komunikácie je to, že je istota, že informácie dorazia v poradí v akom boli odoslané – bez čoho by program nemal možno žiadne praktické využitie.

Možnosť **generovať vrstvu na základe XML**, Pri definovaní Vulkan vrstvy by dlho trvalo definovať každú funkciu ručne ktorú chceme zachytávať, okrem toho sa API neustále rozširuje, práve preto som sa rozhodol všetko generovať na základe definície rozhrania zapísaného do XML súboru[3].

[3]Súbor obsahuje popis Vulkan API, je ho možné nájsť na https://github.com/KhronosGroup/Vulkan-Docs




Obr. 3.3: Flexibilná vrstva (layer.cpp)[4]

**Flexibilná vrstva**, aby sa neprepísala ručne naprogramovaná funkčnosť vrstvy, tak sa dodatočná funkcionalita ktorá sa nevzťahuje na každú funkciu zapisuje do dodatočného súboru kde je definované chovanie pre programátorom vybranú sadu Vulkan funkcií.

[4]Slúži ako prevencia prepisovania vlastného chovania pomocou generátoru

# Kapitola 4

# Implementácia nástroja pre ladenie

V tejto kapitole je popísaný postup práce pri vývoji ladiaceho nástroja pre analýzu a ladenie Vulkan aplikácií. Z kapitoly 3 je jasné čo treba robiť a ako na to, zvolené technológie zahrňujú:

- *C++* – všetky zdrojové súbory používajú tento jazyk, v prípade generovania kódu pre vrstvu a aplikáciu čo zobrazuje stav skúmaného programu bol využitý objektovo orientovaný prístup programovania. Pre niektoré časti ale bude využitý jazyk C, to platí napríklad pre vrstvu, ktorá je rozšírením verejne dostupnej šablóny
- *Vulkan API* – keďže je práca zameraná na Vulkan, je využitý pre vykreslenie UI aplikácie pre zobrazenie stavu. Okrem toho vrstva využíva adresovanie na existujúce Vulkan funkcie a niekedy ich aj volá
- *GLFW* – užitočné pre jednoduchú prácu s vytvorením okna pre aplikáciu
- *ImGui* – ide o light-weight knižnicu s mnoho možnosťami vytvoriť UI pre stavovú aplikáciu
- *PugiXml* – Vulkan sa neustále rozširuje, aby bolo čo najjednoduchšie udržať relevantnosť tohoto nástroja aj do budúcna tak sa väčšina kódu vrstvy generuje na základe XML súboru čo reprezentuje celé API. Pre pohodlnejšiu prácu so súborom je použitie tejto knižnice najviac vhodné
- *winsock* – rozhranie pre programovanie schránok Windows aplikácií, využívať ju bude vrstva pre zasieľanie správ na stavovú aplikáciu

Ako vývojové prostredie som použil *Microsoft Visual Studio 2022* a implementácia je obmedzená na platformu *Windows 10/11 64/32-bit*.

## 4.1 Štruktúra projektu a jeho setup

Projekt je zložený zo zdrojových súborov, ktoré sa potom musia nechať skompilovať aby z nich vznikol použiteľný program. Za účelom udržať v zdrojových súboroch poriadok je použitý nástroj *Cmake*[1] – je jednoduchý na použitie a je možné ho použiť pre preklad projektu mimo Visual Studio. Cmake súbor vyžaduje verziu 3.11.0, C++ 20 štandard a rozlišuje 3 cieľové projekty:

[1] https://cmake.org/

- Aplikácia pre zobrazovanie stavu ladeného programu (*.exe*)
- Generátor kódu pre vrstvu (*.exe*)
- Vrstva pre zachytávanie dát z aplikácie (*.dll*)

Rovnakým spôsobom sú aj rozdelené zložky projektu, kam do každého spadajú zdrojové a hlavičkové súbory daného cieľového projektu. Výnimku predstavujú dva súbory „winsock.cpp/.h“, tento kód sa používa pre spojenie vrstvy a aplikácie – nemalo by zmysel písať to isté na dvoch iných miestach. Súbor *Cmake* pozostáva ešte z časti kedy sa aplikujú rôzne knižnice na rôzne projekty či už zo zložky *libs* alebo pomocou operačného systému. V prípade nemožnosti preložiť projekt doporučujem upraviť *CmakeLists.txt* a zadať cesty požadovaných súborov manuálne.

## 4.2 Ladiaca Vulkan vrstva a zber dát

Aby bol nástroj použiteľný a užitočný, je potrebné poznať aktuálny stav ladenej aplikácie. Tento stav sa určí na základe dát, ktoré zachytí vrstva. Pred tým ako to ale je možné, je potrebné ju konfigurovať.

**Konfigurácia** ladiacej vrstvy je definovaná v súbore *vkDebugger_windows.json*[2], sú tu popísané jej základné informácie ako je napr. meno. Kľúčové v tomto súbore sú ale dve veci:

- Atribút **„features“** – obsahuje zoznam vstupných parametrov, vrátane typu, základnej hodnoty a iné. Pred spustením vrstvy sa hodnoty týchto parametrov nastavia nástrojom *VkConfig*. Vrstva ich potom prečíta a spracuje podľa vlastného uváženia
- Atribút **„functions“** – zoznam Vulkan funkcií, ktoré vrstva zachytáva.
  Napr. „vkCreateInstance“ : „Moja_Vrstva_Create_Ins“

Konkrétne, súčasťou **„features“** sú nastavenia jednotlivých funkcionalít vrstvy, ako napr. zastavenie programu ak FPS klesne pod 25 a podobne. Čo sa týka atribútu **„functions“**, na úrovni konfigurácie zachytáva táto vrstva iba dve Vulkan funkcie *vkGetInstanceProcAddr* a *vkGetDeviceProcAddr*.

Definícia **konfigurácie** týmto končí, teraz je možné vrstvu prostredníctvom *VkConfig* nechať aktivovať, no pre správne fungovanie musí nasledovať **inicializácia**. Moment, kedy sa tento proces spúšťa je, keď sa zavolá *vkCreateInstance* – háčik je ale v tom, že v konfigurácií sú vymenované iba dve funkcie a ani jedna z nich nie je *vkCreateInstance*.

Už ako bolo bližšie opísané v podkapitole 2.4, pri **inicializácií** vrstvy sa na základe volaní funkcií *vkGetInstanceProcAddr* a *vkGetDeviceProcAddr* vyplňujú *DT*[3] podľa ktorých potom **Loader** vie, ktoré vrstvy zachytávajú ktoré funkcie.

Tieto volania pri **inicializácií** ladiaca vrstva zachytí (pretože to má nakonfigurované) a nechá vyplniť do *DT* predchádzajúceho konštruktu (či už *DT* na **Loader** úrovni alebo *DT* predchádzajúcej vrstvy). Potom pri zavolaní ďalších zachytávaných Vulkan funkcií sa urobí zastávka aj v ladiacej **vrstve**.

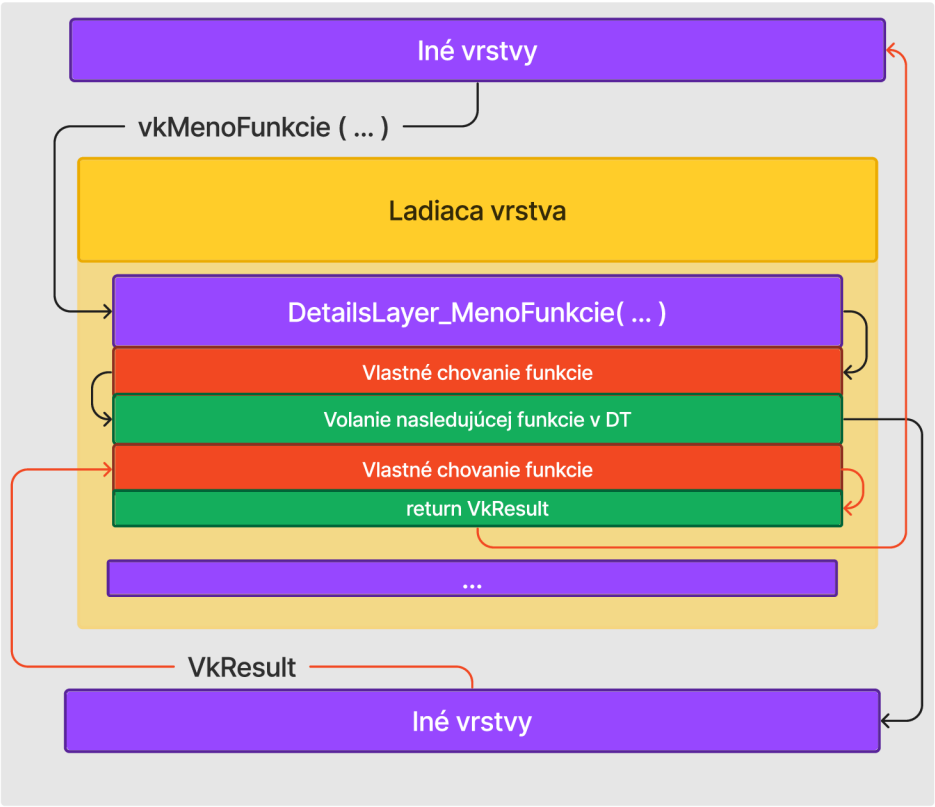
Aby ale pri zastávke program neuviazol na ladiacej vrstve, vyplní si pri zavolaní *vkCreateInstance* vlastnú *IDT* a pri volaní *vkCreateDevice* vlastnú *DDT*, po vykonaní práce na vrstve sa skočí na ďalšiu adresu v tabuľke.

[2]Formát súboru je vo validnom *layer manifest* formáte (viac na https://vulkan.lunarg.com/doc/view/1.3.216.0/mac/loader_and_layer_interface.html#user-content-layer-manifest-file-format)

[3]*DT* – dispatch tabuľka

Hoci by už vrstva bola schopná práce, bolo treba ešte ošetriť jednu maličkosť: keďže Vulkan aplikácia môže mať viac ako jeden *device* a *instance*, mapujú sa ich jednotlivé *handle* odkazy na jednotlivé tabuľky (napr. device A má *DDT* 1 a device B má *DDT* 2 Ale obe sú súčasťou rovnakého ladeného programu). A ako prevencia nedefinovaného chovania pri prístupe viacvláknového programu môže zapisovať do tejto mapy najviac jedno vlákno – toto je chránené globálnym zámkom.

Takže aplikácia zavolala *vkCreateInstance*, ladiaca vrstva predala odkazy funkcií ktoré zachytáva (táto vrstva zachytáva všetky funkcie) predchádzajúcemu konštruktu a vyplnila si vlastné tabuľky odkazmi na nadchádzajúce funkcie. Výsledkom je vrstva v **inicializovanom** stave. Teraz je možné definovať chovanie každej zachytenej funkcie osobitne:




Obr. 4.1: Vzor behu zachytenej funckie na ladiacej vrstve[4]

V tomto stave vrstva zachytáva všetky volania, vizualizácia priebehu zachytenej funkcie je možné vidieť na obrázku 4.1, pričom:

1. Do funkcie na vrstve sa predajú parametre, poznáme aj meno aj návratovú hodnotu volanej funkcie
2. Potom sa môže pracovať s dostupnými dátami
3. Skáče sa na ďalšiu adresu v *DT*
4. Znovu sa môže pracovať s dostupnými dátami
5. Smerom naspäť do programu sa posiela výsledná hodnota operácie

[4]Čierna šípka je smer volania na ovládače, červená je smer naspäť do ladeného programu

Dáta popisujúce stav programu zachytáva vrstva v červených bodoch podľa obrázku 4.1 a zasiela do stavovej aplikácie (viac o tom v kapitole 4.4). Na týchto dvoch miestach sa robí mnoho iných vecí:

1. **Kontrola podmienok** – na základe vstupných parametrov môžu nastať rôzne udalosti implementované práve tu, môže ísť napr. o zastavenie programu ak klesne FPS pod 25 alebo spustenie stavovej aplikácie vo fáze **inicializácie** vrstvy.

2. **Stav VkMemory** – obsah pamäti sa môže pri ladení hodiť, je to vyriešené kopírovaním existujúcich **VkImage** alebo **VkBuffer** objektov do vrstvou vytvoreného, CPU-čitateľného objektu typu **VkBuffer**[5] a následného zaslania jeho obsahu do stavového programu.

3. **Stav odmapovanej VkMemory** – čo sa týka odmapovanej pamäti, jej obsah je bezpečné zachytiť ešte pred zápisom na GPU, to ušetrí výkon pri ladení

Čítanie pamäti týmto spôsobom má nevýhodu v tom, že tento nástroj nerozlišuje kopírovanie iba niektorého regiónu zo zdrojových dát[6] a kopíruje výhradne dáta v celku.

Sledovať pamäť ale vyžaduje volať Vulkan funkcie. Nepríjemnosť je v tom, že tieto funkcie znovu prechádzajú touto vrstvou, to prináša potenciálne nasledujúce problémy:

1. Deadlock

2. Nekonečné cyklenie

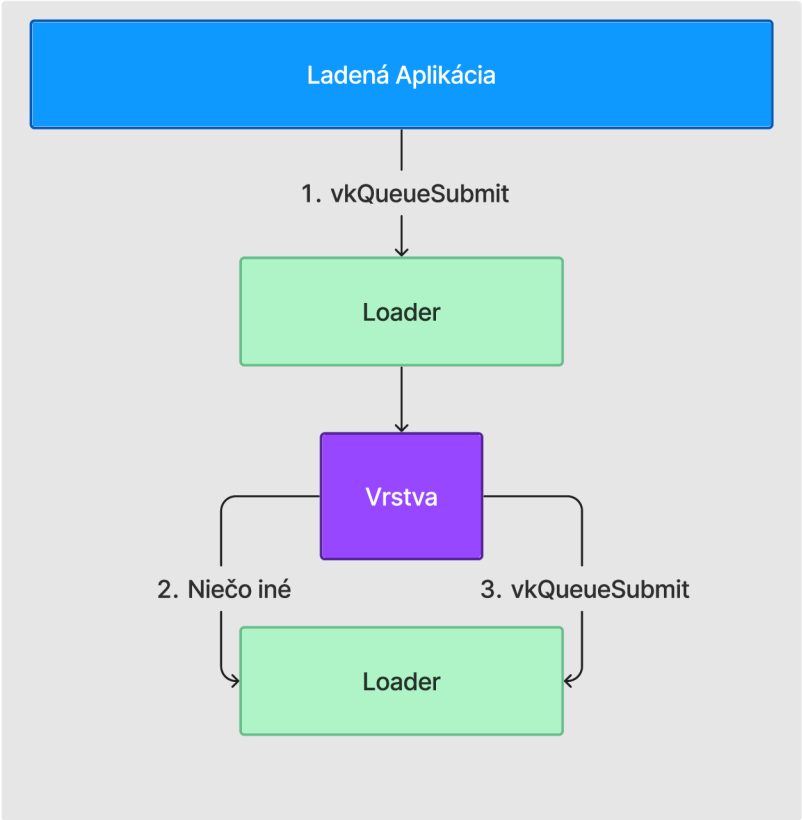
3. Zaradenie zachytenej funkcie do stavu ladeného programu

Spôsob prevencie bodu 1, je riešený cez *boolean* premennú *ignoreLock*, kedy toto volanie prejde vrstvou bez narazenia na už uzamknutý *lock*.

Ochrana proti 2 a 3 sú istené jednou *boolean* premennou *connected* – po dobu volania API z vrstvy nastavená do stavu *false*, preto sa nestane potom náhodou to, že by sa do stavu programu započítalo aj toto volanie, alebo že by sa znovu zavolala tá istá Vulkan funkcia. Názorný príklad je pre lepšiu predstavu zobrazený na obrázku nižšie:

[5] Zaistené nastavením VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT pri jeho vytváraní

[6] Napr. pomocou premennej *offset*




Obr. 4.2: Podvrhnutie Vulkan volania na úrovni vrstvy[7]

## 4.3 Generovanie kódu, na základe XML

Nevýhoda ladiacej vrstvy je v tom, že typy premenných adries je pre každé volanie špecifické (napr. pre *CreateInstance* je odkaz na funkciu typu *PFN_vkCreateInstance* a pre *GetDrmDisplayEXT* je to typ *PFN_vkGetDrmDisplayEXT*)

A keďže ladiaca vrstva zachytáva všetky známe volania, každé volanie musí mať zvlášť položku v jej všetkých *IDT* a *DDT*. Prakticky to znamená tak zhruba tisíc riadkov deklarovania rôznych premenných do štruktúr *DT*:

```
1 ...
2 PFN_vkDeviceWaitIdle DeviceWaitIdle;
3 PFN_vkAllocateMemory AllocateMemory;
4 PFN_vkFreeMemory FreeMemory;
5 PFN_vkMapMemory MapMemory;
6 PFN_vkUnmapMemory UnmapMemory;
7 ...
```

Hoci by to bolo možné napísať ručne, nie je to praktické z dvoch dôvodov: časová náročnosť a risk, že Vulkan API sa aktualizáciami stále mení, bolo by ťažké držať si prehľad

[7] Najprv príde do vrstvy volanie ktoré vrstva odchytáva, potom sa zavolá iná Vulkan funkcia a až potom originálne volanie. Dá sa to nazvať ako podvrhnutie volania, pretože sa zavolá niečo o čom ladená aplikácia ani nemusí vedieť

o všetkých zmenách a neurobiť žiadnu chybu, preto som sa rozhodol využiť generovanie kódu ako riešenie týchto problémov.

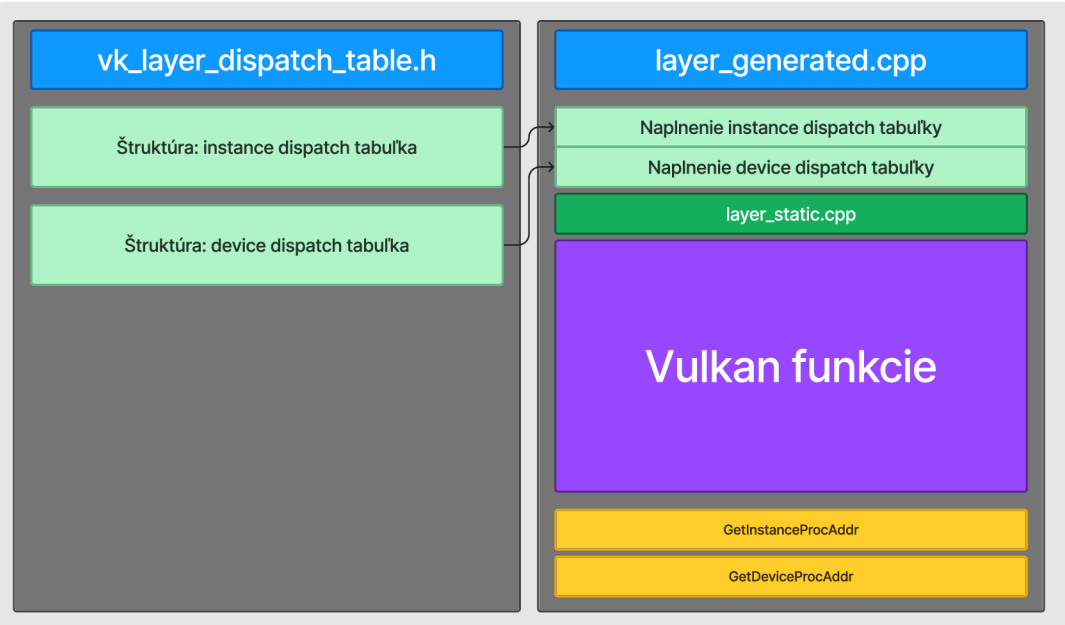
Takže, aby ladiaca vrstva správne fungovala, je potreba generovať dva súbory:

- *vk_layer_dispatch_table.h* – V tomto súbore je definovaná štruktúra *IDT* a *DDT*
- *layer_generated.cpp* – Súbor kde je definované správanie funkcií ktoré zachytávame, inicializácia vrstvy (teda vyplnenie *IDT* a *DDT*, ktoré vrstva vlastní)

Ako zdrojový súbor pre generovanie kódu sa využíva voľne dostupný a pravidelne aktualizovaný *XML* súbor, ktorý obsahuje register pre Vulkan API [8]. Pre parsing tohoto súboru som tu využil šikovnej knižnice *pugixml*[9] – pomocou nej sa na začiatku behu generátora načítajú kľúčové informácie o Vulkan API:

Mená, typ a parametre Vulkan funkcií, platformy[10] a štruktúry[11].

Tieto informácie využíva generátor pre generovanie kódu postupne:




Obr. 4.3: Štruktúra generovaných súborov

Podľa obrázku 4.3, je do súborov vygenerovaný kód.

**Vľavo**, v súbore vk_layer_dispatch_table.h ide o definíciu štruktúr *IDT* a *DDT*, použitie tu nachádzajú informácie o *funckiách* a *platformách* získaných z *XML*. Tieto štruktúry používa potom ladiaca vrstva.

**Napravo**, v súbore layer_generated.cpp je kód pre naplnenie *IDT* a *DDT*, teda do každej položky sa priradí adresa na funkciu nasledujúceho bodu (napr. adresa funckie *vkCreateInstance* nasledujúcej vrstvy). Tento generovaný kód tiež závisí od *funkcií* a *platforiem*.

Čo sa týka časti menom „layer_static.cpp“, ide o obsah tohoto súboru s týmto menom prekopírovaný do vygenerovaného kódu ladiacej vrstvy. Užitočné je to v tom, že ak prog-

[8] https://github.com/KhronosGroup/Vulkan-Docs

[9] https://pugixml.org/

[10] Zoznam funkcií, ktoré majú pre daný platformu zmysel definovať: (napr. ak sa definovalo makro *VK_USE_PLATFORM_WIN32_KHR*, je dovolené používať funkciu *vkGetFenceWin32HandleKHR*)

[11] Užitočné ak je nejaký parameter zároveň štruktúra, nechá sa rozbaliť a vidieť hodnoty tejto štruktúry

ramátor niečo potrebuje napísať niečo do generovanej vrstvy a nechce to generovať, tak to zapíše do tohoto súboru (napr. inicializácia v *vkCreateInstance*).

Body *GetInstanceProcAddr* a *GetDeviceProcAddr* sú zachytávané funkcie na základe konfigurácie ladiacej vrstvy a používané potom pri inicializácií. Obsahom týchto funkcií je dlhý zoznam makier, ktoré využívajú mená funkcií získaných z *XML* súboru.

Jadrom toho všetkého je posledná časť, „Vulkan funkcie“, kde
to vyzerá následovne:


Obr. 4.4: Obsah vzoru vygenerovanej funkcie

Jedná sa o vzor, ktorý je aplikovaný pre každé zachytávanie Vulkan volanie (Až na pár výjimiek, ako je napr. *vkCreateInstance*).

- **Platform guard** – (nepovinná)
- **Funkcia_X ( parametre )** – deklarácia mena **funkcie**, jej parametrov a návratového typu
- **Vlastné chovanie funkcie** – volanie funkcie súboru *layer.cpp* v prípade, že je definované makro pre túto funkciu, dôvodom je to, že
- **Zaslanie parametrov** – transformácia obsahu parametrov do typu *char** a následné zaslanie do stavovej aplikácie. Využíva parametrov **funkcie**, **štruktúry** a **enums**
- **Zaslanie príkazu** – volanie funkcie v ďalšom dispatch chain bode, ak má funkcia návratový typ, uloží si výslednú hodnotu a potom v **return** ju zašle naspäť do aplikácie, využité meno **funkcie**

Týmto je kód vrstvy vygenerovaný. V prípade zmien v API stačí stiahnuť najaktuálnejší register Vulkan API a znovu vygenerovať. Po vygenerovaní stačí súbor nechať preložiť.

## 4.4 Winsock, komunikácia medzi aplikáciami

Ako bolo už navrhnuté v kapitole 3, ladiaca vrstva a *stavová aplikácia*[12] sú dve oddelené entity aby bolo možné zobraziť stav ladenej aplikácie v dobe pádu. Nevýhodou toho je, že sa teraz informácia musí z vrstvy dostať do stavovej aplikácie.

Toto je implementované nasledovne, sieťový socket, jeden na strane vrstvy a druhý na stavovej aplikácií, pričom medzi nimi prebieha komunikácia protokolom TCP. Toto garantuje to, že ak sa informácia stratí počas transportu, tak stavová aplikácia si ju vyžiada znovu pokým nedorazí – táto spoľahlivosť je kľúčová k zaisteniu integrity aktuálneho stavu aplikácie v akýkoľvek moment.

Socket, jeden i druhý po inicializácií, a naviazania spojenia medzi sebou, sa nechávajú posielať informácie z vrstvy do stavovej aplikácie, posielané informácie sú rozdelené na dva typy:

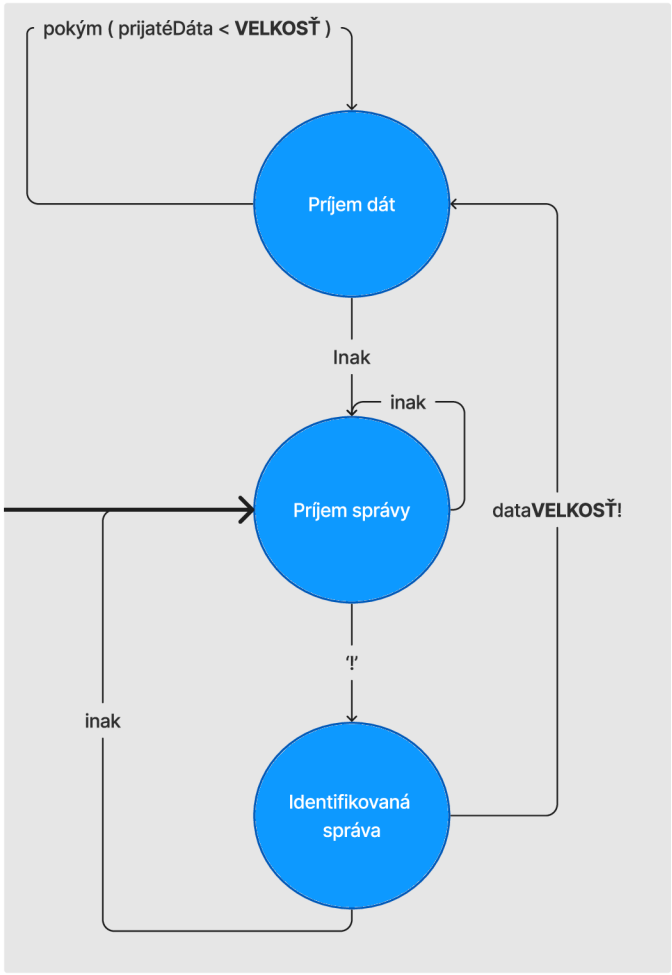
1. **Správy** – formát *char** kde je obsah správy a odkaz na schránku
2. **Dáta** – formát *void** so zdrojovými dátami, veľkosťou dát a odkaz na schránku

Pri pracovaní s TCP komunikáciou ale ešte neznamená jedno volanie jeden príkaz. Správy sa totiž zapisujú do **bufferu**, ktorého obsah sa posiela niekedy s viac jednou správou. Niekedy však správa nedorazí celá, pre lepšie pochopenie sú nižšie príklady ako správy môžu po príchode vyzerať.

1. *message1*
2. *message1message2...messageX*
3. *message1...mess*
4. *messa*

Toto som vyriešil delimitovaním správ pomocou znaku „!“. Keď sa nájde tento znak, môže sa začať správa obsluhovať. Ak nie (správy 3, 4), obsah správy sa spojí s tou nasledujúcou

[12] Aplikácia, ktorá zobrazuje stav ladeného programu, implementovaná pod neskôr menom *VkDebuggerApp*




Obr. 4.5: Konečný automat prijímania správ

Prinieslo to ale nový problém, kedy by sa tento znak mohol teoreticky vyskytnúť pri prijatí dát. To je riešené konečným automatom vyššie 4.7, že pred dátami je vždy formulovaná veľkosť dát. Počas načítavania dát sa jednoducho nerieši hľadanie výkričníku a po dosiahnutí veľkosti dát sa program znovu prepne do módu prijímania správ.

Pre rozlíšenie správ som vymyslel tieto konvencie:

- meno parametru **parameter=** + *char** hodnota + '!' – takto zapísaný parameter si spracováva stavová aplikácia sama a rozumie pod ňou parameter funkcie
- prefix **layer_** + meno parametru **parameter=** + *char** hodnota + '!' – správa tohoto tvaru pochádza z extra chovania funkcie, čaká sa, že ju programátor spracuje podľa vlastného uváženia
- prefix **data_** + dĺžka dát **XXX** + '!' – správa značí, že budú nadchádzať dáta veľkosti XXX, preto nasledujúcich XXX znakov pochop a ukladaj ako binárne dáta

## 4.5 Stavová aplikácia a spracovanie dát

Po implementácií predošlých častí by už teraz bolo možné zobrazovať stav aplikácie, bolo by to nepraktické – zozbierané dáta sú surové a ťažko by sa v takomto stave programátorovi

čítali, pre uľahčenie práce so zozbieranými informáciami o ladenom programe preto najprv príde k ich spracovaniu a až potom sa v nejakom stave dostanú do užívateľského rozhrania.

Ako už bolo v predchádzajúcej kapitole (4.4) naznačené, správy chodia na sieťový socket, kedy najprv sa identifikuje pôvod správy zapísaný v jej *prefixe*. Čo sa týka prefixu, v programe môže nastať 5 prípadov:

1–2. Prefix *beginning/end* – čo znamená začiatok a koniec jedného Vulkan API volania

3. Prefix správa nemá a je medzi správami *beginning* a *end*, jedná sa o parameter funkcie (meno aj hodnota)

4. Prefix správa nemá a je aktívny *boolean* označujúci, že ide o *surové* binárne dáta

5. Prefix *layer*, ide o správy, ktorých chovanie je špecifické pre určitú podmnožinu Vulkan funkcií – preto sa tieto správy nechávajú spracovávať osobitne

Správy **1**, **2** a **3** sa ukladajú vo forme zoznamu triedy *apiCall*, trieda obsahuje okrem mena, návratovej hodnoty a parametrov aj **ID**, na základe ktorého sa určuje v ktorom poradí bola funkcia volaná.

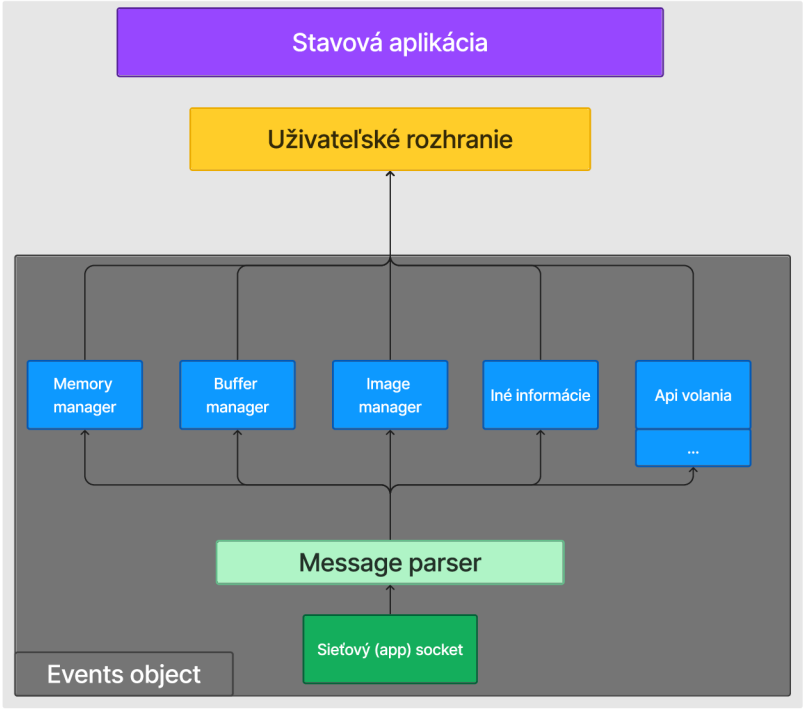
Čo sa týka prijatých binárnych dát (**bod 4**), tie sa uložia do triedy *memoryManager* v prípade, že **VkMemory** objekt bol alokovaný a boli načítané do neho tieto dáta. Pre lepšiu predstavu, trieda *memoryManager* si tieto dáta ukladá do mapy štruktúr asociované v podobe *MemoryID* : *VkMemory Handle Address* podľa ktorej sa overuje existencia tejto pamäti (To aby sa náhodou nepriradili dáta do pamäti, ktorá ani neexistuje, to by programátorovi nedalo správnu informáciu o stave programu).

**Posledný bod (5)** je najkomplexnejší v tom, že správy tohoto typu sa spracovávajú rôzne. Sú pôvodom mimo vygenerovanú vrstvu, alias definované programátorom, prakticky, ak by niekto v budúcnosti mal záujem rozšíriť funkcionalitu, je najviac pohodlné to robiť pomocou týchto správ.

Implementovaná funkcionalita väčšina funkcionality spočíva v čítaní dát z **Správ typu 5**. Pre začiatok, pri zavolaní funkcie *vkCreateInstance* (teda pri inicializácií), sú do aplikácie poslané vybrané vstupné parametre ladiacej vrstvy napr. určené pomocou nástroja *Vulkan Configurator*. Ide aj o informácie, že či daná funkcionalita má byť pre daný beh ladenia aktívna, napr. programátora nemusí zaujímať obsah obrázkov, preto informuje stavovú aplikáciu, že dáta týkajúce sa obrázkov čakať nemusí.

Súčasťou týchto správ sú aj upozornenia (tzv. *Warnings*, definované užívateľom pred spustením ladenia, tiež napr. prostredníctvom programu *VkConfig*) a upozornenia o zastavení programu na základe podmienky (tiež definovanú na základe vstupných parametrov).

Ostatné správy **typu 5** je už funkcionalita pre Vulkan objekty typu **VkBuffer**, **VkImage**, spracováva sa to na podobnom princípe ako **vkMemory** a majú korešpondujúce manažér objekty, kde sa tieto dáta ukladajú (*bufferManager* a *imageManager*).




Obr. 4.6: Architektúra spracovávania dát aktuálneho stavu ladenej aplikácie[13]

Dáta sú teda spracované a pripravené zobraziť na užívateľské rozhranie. Ešte ale pred zobrazením informácií o ladenom programe je potrebné **inicializovať** *Events object* – aby čo najskôr sa mohli ukladať dáta a nečakalo sa na kým nastane *TCP handshake* pri inicializácií ladiacej vrstvy.

Súčasťou **inicializácie** je aj kód pre inicializáciu okna a užívateľského rozhrania knižnice *ImGui*, je to riešené spôsobom, aký ukazuje autor tejto knižnice na názornej ukážke[14].

Po inicializácií sa nastavia štýly a farby jednotlivých *UI* komponent, ktoré budú aplikované vo vnútri cyklu pre vykresľovanie užívateľského rozhrania. Pre ušetrenie výkonu je podmienka pre vykreslenie pohyb myši po okne, jej stlačením alebo stlačením nejakej klávesy keď je okno v popredí.

[13] *Events object* je pomenovanie aké som dal tomuto objektu a je implementovaný v súboroch *events.cpp/.hpp*

[14] kód pre inicializáciu je, až na drobné zmeny, prebratý z https://github.com/ocornut/imgui/blob/master/examples/example_glfw_vulkan/main.cpp

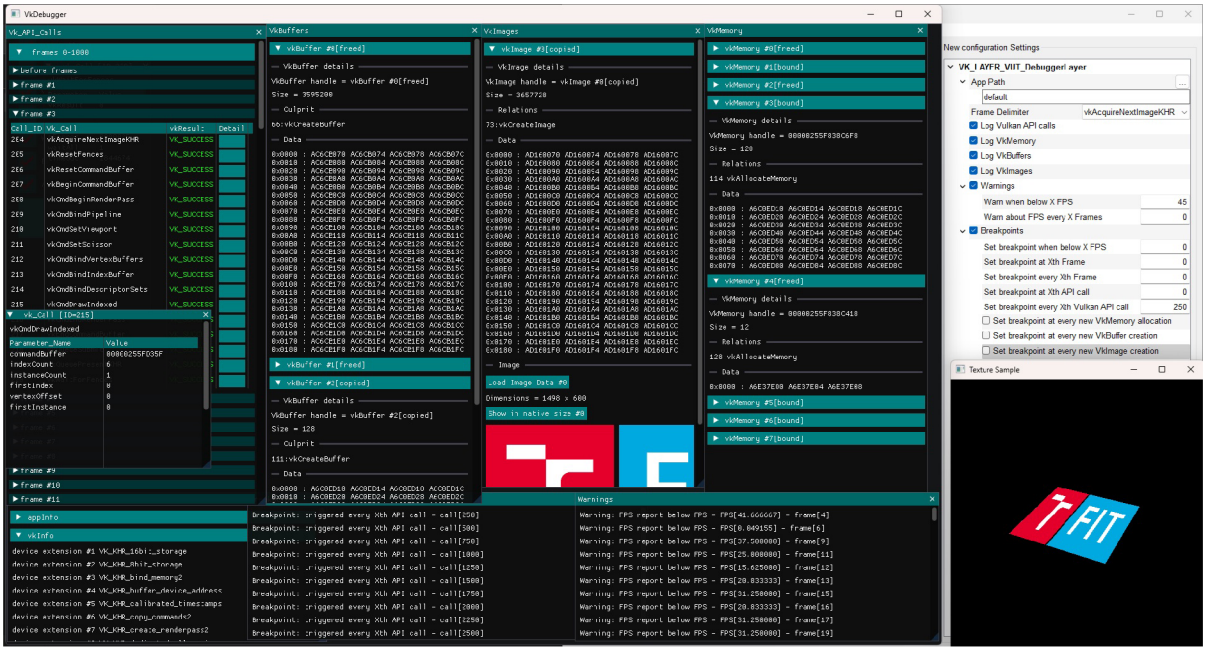

Obr. 4.7: Vykresľovanie aktuálneho stavu ladenej aplikácie

Jadro tohoto cyklu tvoria položky menu, ktoré sú užívateľovi sprístupnené na základe vstupných parametrov ladiacej vrstvy (napr. sú vypnuté upozornenia ale ostatná funkcionalita je aktívna). Druhou a tou najvýznamnejšou položkou sú volanie funkcií, v ktorých sú jednotlivé okná pre tieto funkcionality implementované. Každé okno si potom od *Events objektu* žiada jednotlivé informácie o aktuálnom stave ladeného programu. Je to tak rozdelené preto, aby sa celý stav ladenej aplikácie neprenášal naraz – v prípade výrazne veľkého množstva dát by to malo tendenciu stavovú aplikáciu spomaľovať, čo by bolo kontraproduktívne. Týmto je implementácia užívateľského rozhrania dokončená.

# Kapitola 5

# Testovanie a výsledky

V tejto časti je najprv stručný náhľad na vytvorený ladiaci nástroj **VkDebugger**, bližšie popísaná zvolená stratégia pre overenie funkčnosti, jej uskutočnenie a výsledky získané z tohoto celého procesu testovania.



Obr. 5.1: Výsledný ladiaci nástroj VkDebugger v použití[1]

Síce obrázok 5.1 už prezradil podporovanú funkcionalitu výslednej aplikácie, je podstatné za účelom testovania ich vymenovať a overiť funkčnosť za každých podmienok:

- rozdelenie volaní *frame* čo *frame* – na základe vybraného Vulkan volania
- zobrazovanie histórie volaní Vulkan API funkcií
- udržovanie stav VkMemory, VkImage a VkBuffer objektov

[1] Naľavo je VkDebugger so všetkými oknami zobrazujúce stav aplikácie zobrazenej v pravom dolnom rohu. Vpravo hore je Vulkan Configurator, pomocou ktorého sa nastavuje ladiaci nástroj a slúži aj ako spúšťač pre testované programy

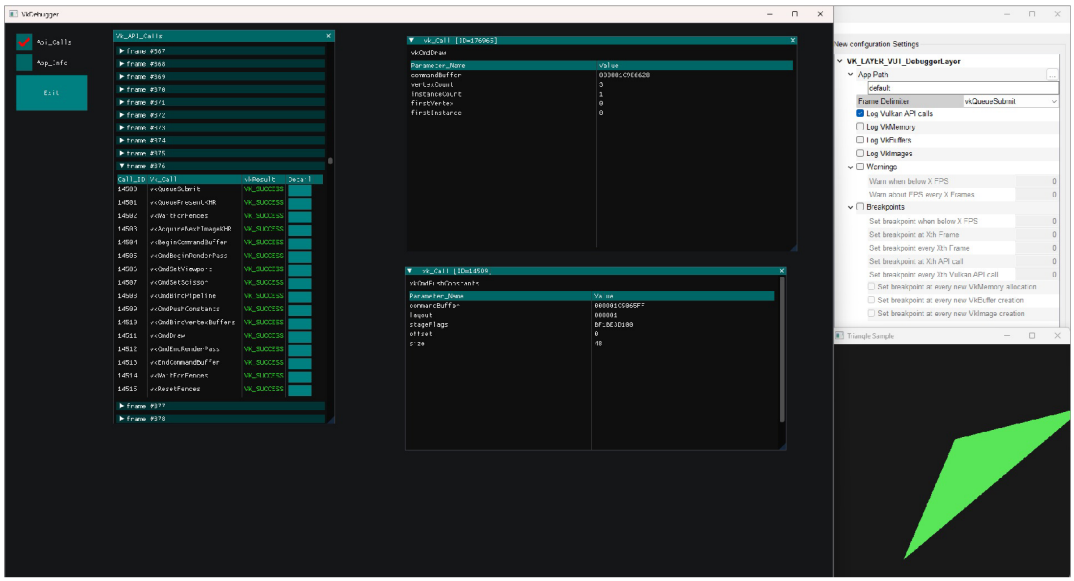
- preskúmanie obsahu VkMemory, VkImage a VkBuffers objektov
- vykreslenie obsahu VkImage objektu v podobe obrázku
- upozornenie užívateľa na základe jednoduchej podmienky/podmienok
- pozastavenie vykonávania ladeného programu na základe jednoduchej podmienky-/podmienok

Teraz je jasné že čo tento nástroj má robiť. Nebolo by ale rozumné, začať skúšať veci iba tak náhodne. Bol vybraný taký prístup, aby výsledky boli čo najkvalitnejšie, teda stratégia je dostať VkDebugger do bežných situácií ktoré by sa dali pri jeho používaní očakávať:

- použitie na rôznych grafických aplikáciach (2D, 3D, jednoduché, komplexné, ...)
- použitie na grafických aplikáciach **s chybou**
- využitie na rozdielnych zariadeniach (diskrétné, integrované GPU karty)

## 5.1 Experimenty na grafických aplikáciach

Ako prvé boli otestované jednotlivé funkcionality aplikácie a ich vzájomnej interakcie v prípade, že niektorá z nich je vypnutá:

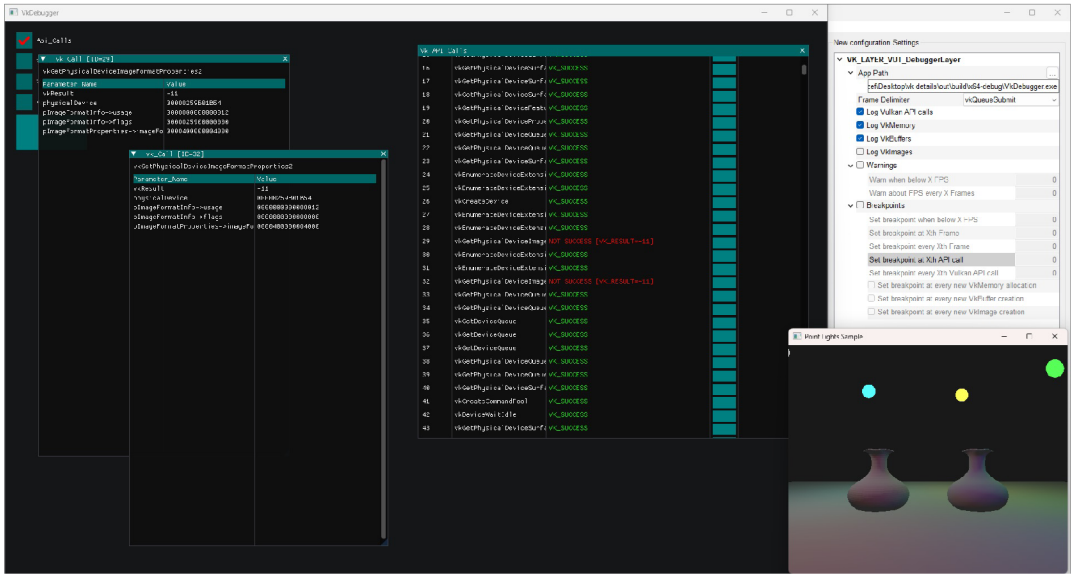


Obr. 5.2: VkDebugger s obmedzenou funkčnosťou[2]

Problém pri ostatných ladiacich nástrojov pre ladiace aplikácie bol vždy ten, že stále skúmajú celý stav aplikácie – a je značne náročné na výkon, preto také programy obvykle odchytia iba určitú časť programu ktorú budú analyzovať. VkDebugger toto obmedzenie ale nemá, pretože užívateľ si môže vybrať veci čo ho zaujímajú a tie potom za behu analyzovať všetky ako celok. Rôzne kombinácie týchto nastavení tiež nerobia problém, vždy je dovolené analyzovať iba to čo si užívateľ zvolí.

[2]Na obrázku vidieť konkrétne iba históriu Vulkan volaní a zopár jej parametrov

Avšak, už teraz pri niektorých testovaných aplikáciach je výkon dosť ovplyvnený v prípade, že je zapnutá všetka funkcionalita naraz. V tomto prípade ale užívateľ môže využiť prístup hľadania chýb s obmedzenými funkcionalitami VkDebuggeru, to by potenciálne mohlo poukázať na chybu v programe spôsobom, že ak je zapnutá funkcionalita A a všetko je v poriadku, možno bude chyba ktorú odhalí funkcionalita B. (napr. po ladení s pozorovaním VkImage objektov sa nič zlé nenašlo, ale pri sledovaní VkBuffer objektov pri druhom ladení sa zdá že dáta niekde chýbajú)



Obr. 5.3: Neúspešné návratové hodnoty Vulkan volaní[3]

Po úspešnom testovaní jednotlivej funkcionality dáva zmysel skúsiť to i na iných aplikáciach, dôraz sa tu kladie na ich rozdielne vlastnosti čo by mohli potenciálne odhaliť zásadný problém s implementáciou.

Pre účely testovania boli vybraných zopár Vulkan grafických aplikácií:

- *Triangle Sample*[4] – 2D aplikácia s pohybom, jednoduché opakovanie volaní vkCmdDraw, je možné vidieť aj na obrázku 5.3
- *Static Light Sample* – jednoduchá 3D scéna s jednoduchým osvetlením
- *Point Lights Sample* – 3D scéna s pohybom a s osvetlením
- *Resizable Window Sample*[5] – 2D aplikácia bez pohybu s flexibilným oknom
- *Simple Image Sample*[6] – aplikácia s výstupom do bitmapového súboru
- *Texture Sample*[7] – 3D aplikácia s pohybom a textúrou

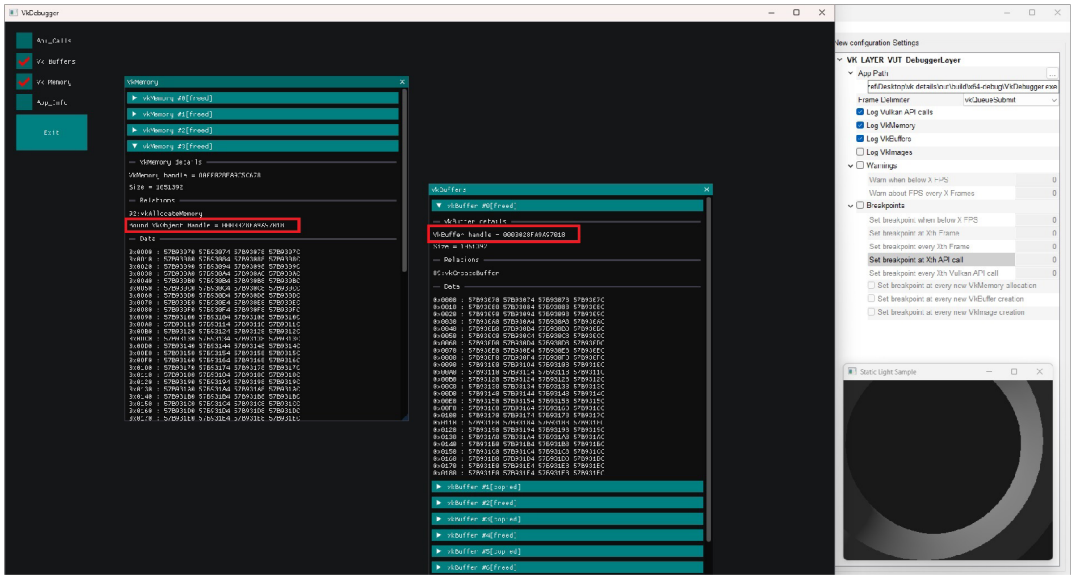
[3] V tomto prípade ale ešte nemusí ísť o chybu, ide skôr o to, že určitý formát iba nie je podporovaný tak si to musí programátor nejak zariadiť

[4] Pre prvé tri platí, že sú dostupné online z https://github.com/blurrypiano/littleVulkanEngine

[5] Dostupná online z https://www.root.cz/clanky/vulkan-okno-menitelne-velikosti/

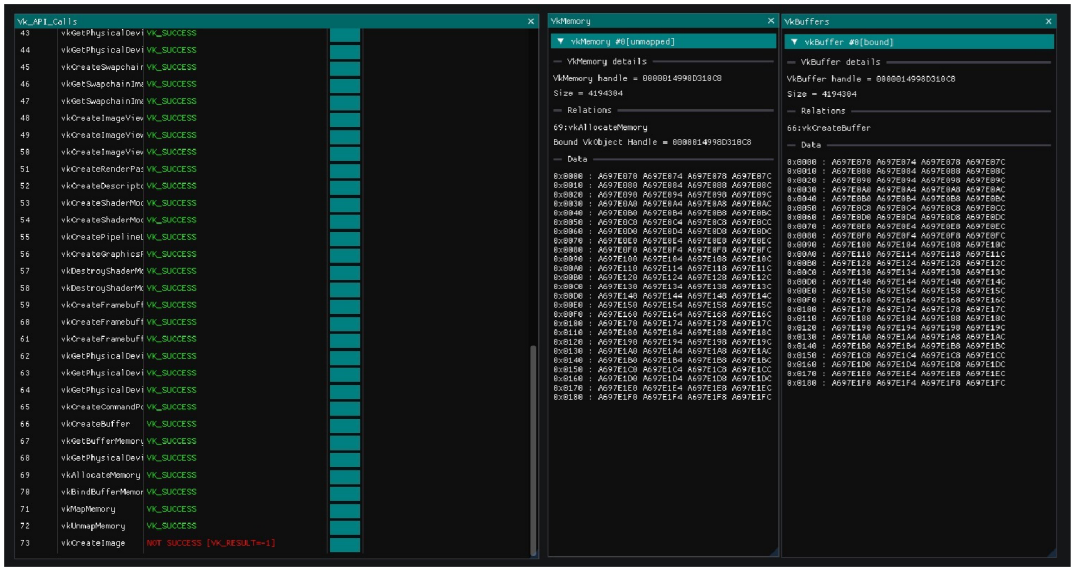
[6] Dostupná online z https://www.root.cz/clanky/vulkan-prvni-vyrenderovany-obrazek/

[7] Dostupná online z https://github.com/Overv/VulkanTutorial



Obr. 5.4: Vzťah medzi VkMemory a VkBuffer objektmi[8]

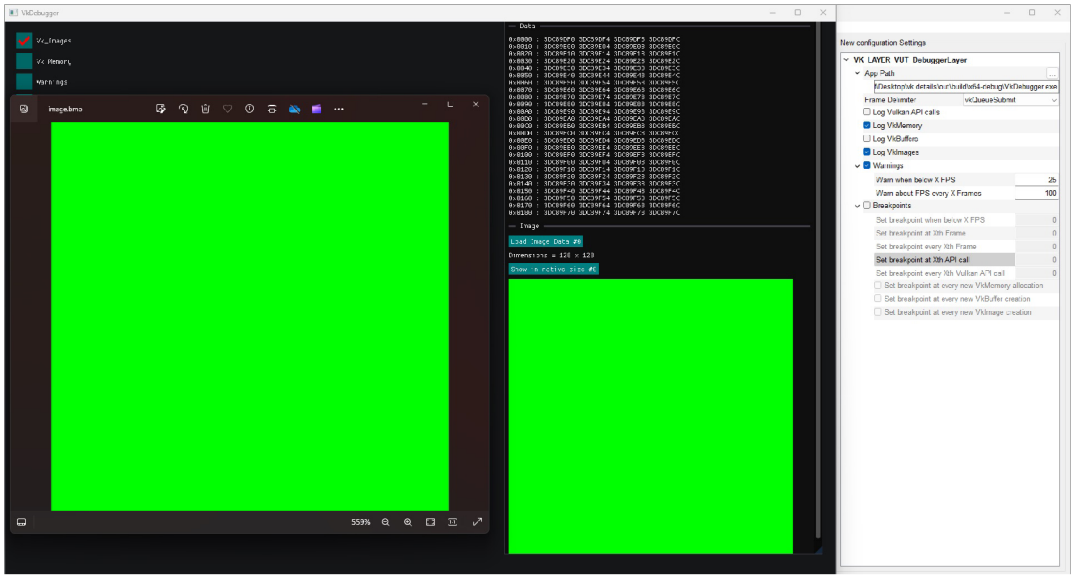
Hoci sú všetky vybrané aplikácie vhodní kandidáti pre otestovanie funkčnosti, bolo treba vniesť medzi nich aj chyby na rôznych miestach, *VkDebugger* tak bol riadne otestovaný čo sa týka variability grafických aplikácií. Mnohokrát v prípade zanesenia chyby do programu tak padol a spolu s ním aj vrstva. Našťastie má *VkDebugger* oddelenú architektúru od vrstvy, preto aj po páde bolo možné nahliadnuť do posledného stavu aplikácie a zistiť tak príčinu pádu.



Obr. 5.5: Chyba aplikácie pri behu[9]

[8] VkMemory objekt a VkBuffer alebo VkImage môžu byť na seba navzájom odkázané a dá sa zistiť kto koho ako ovplyvňoval

[9] Aplikácia v tomto prípade aj spadla, napriek tomu je možné stav programu vidieť v dobe pádu vrátane hodnoty chyby, v tomto prípade ide o chybu typu *VK_ERROR_OUT_OF_HOST_MEMORY*



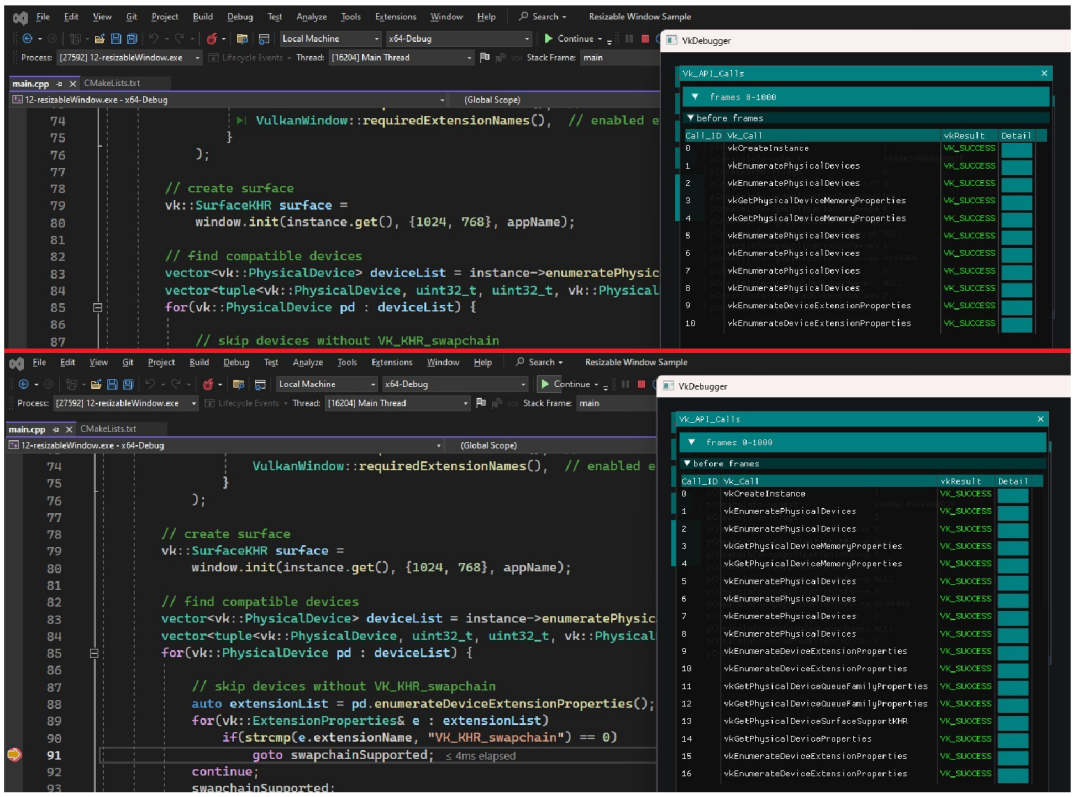
Obr. 5.6: Aplikácia generujúca bitmapový obrázok[10]

A hoci bol nástroj otestovaný so všetkou funkcionalitou na všetkých ladených programoch, bolo to treba urobiť znovu a rovno niekoľkokrát – dôvodom je to, že Vulkan je nízkoúrovňové API a nemusí byť isté, že vyvinutá aplikácia bude fungovať na každom zariadení.

V rámci možností som využil príležitosť otestovať to všetko na nasledujúcich zariadeniach:

- *NVIDIA GeForce GTX 1650* (diskrétna grafická karta, verzia ovládača 552.22.0.0)
- *Intel(R) UHD Graphics 630* (integrovaná grafická karta, verzia ovládača 0.402.537)
- *AMD Radeon(TM) Graphics* (integrovaná grafická karta, verzia ovládača 2.0.193)

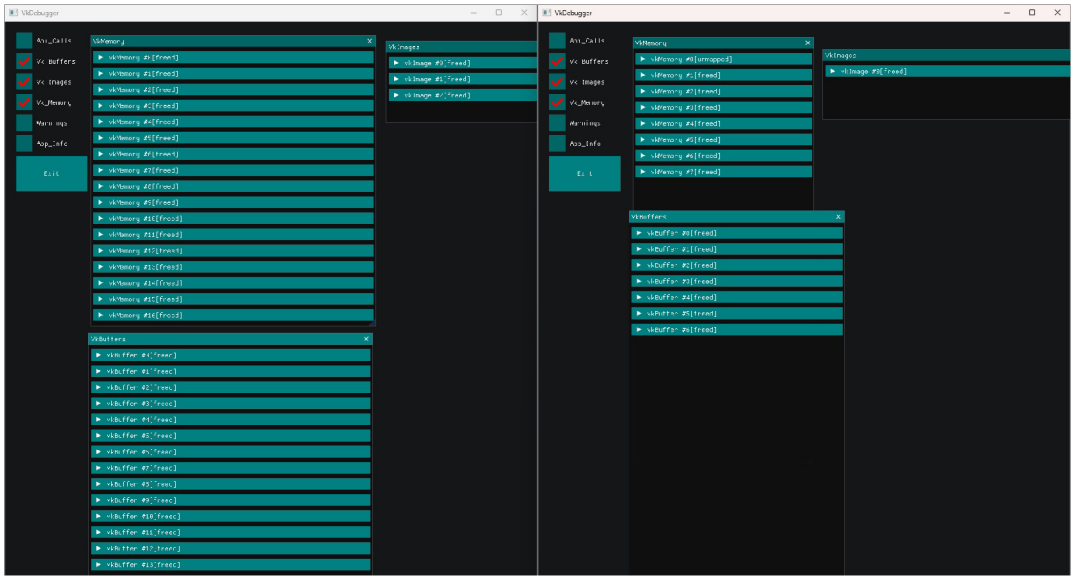
[10]Zachytenie dát obrázku nezáleží na tom, či ho aplikácia vykresľuje do okna




Obr. 5.7: Aplikácia s využitím VS breakpoint funkcionality[11]

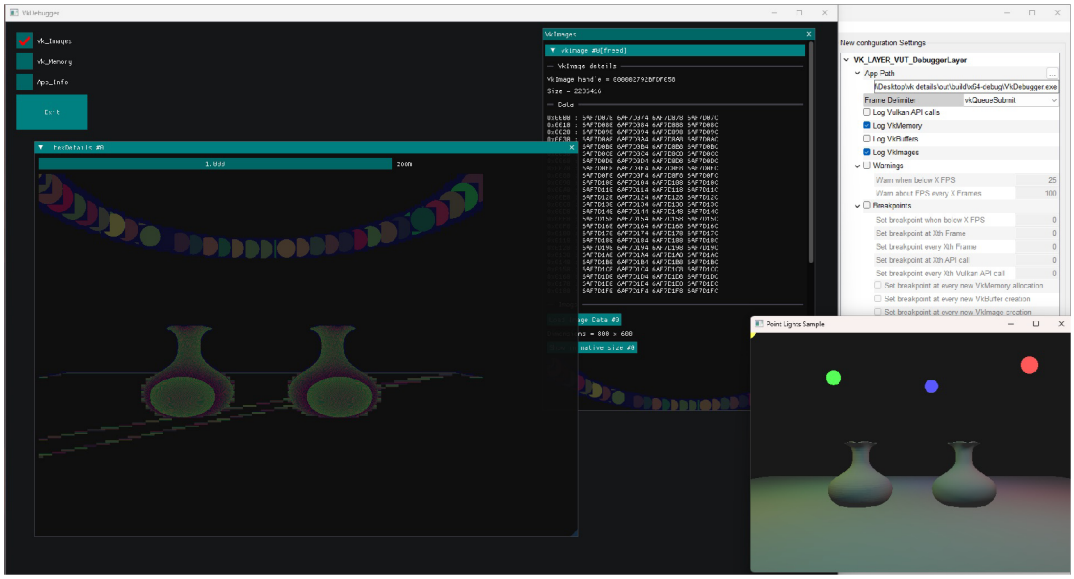
Experimenty na iných zariadeniach neprinášalo nové poznatky pokým neprišlo k testovaniu funkcionality na GPU karte *AMD Radeon(TM) Graphics*. Problém nastal vtedy, keď pri pokuse vykresliť zachytené dáta obrázku do *VkDebugger* aplikácie – jednoducho aplikácia zletela. To viedlo k dlhým hodinám ladenia ale nakoniec sa ukázalo, že chyba je v *VkDebugger* aplikácií. Konkrétne išlo o spôsob akým sa alokujú *VkCommandPool* objekty – pretože ako už bolo naznačené, niektoré grafické karty sú na určité veci citlivé a vyžadujú správne využitie prostredia Vulkan. V prípade aplikácie sa to nerobilo správne a i tak testovanie túto chybu neodhalilo až doteraz. Zistilo sa, že kľúčovým rozdielom medzi touto problémovou grafickou kartou a tými druhými práve podpora pre Vulkan extension *VK_KHR_maintenance1*, bez nej jednoducho niektoré veci neprejdú.

[11] Horná časť okna je stav programu pred zastavením a spodná časť je stav po zastavení



Obr. 5.8: Uvoľnenie zdrojov[12]

Čo neprešlo ale pri testovaní, je vykresľovanie obrázkov používajúce *VkRenderPass* objektu. Problém môže byť na viacerých miestach, zrejme to má čo dočinenia s parametrom obrázku *VkImageLayout*. Dáta obrázkov získaných mimo túto metódu (map/unmap) fungujú ale zcela správne a k vykresleniu dochádza bez problému. Preto môže byť podozrivý aj spôsob vykresľovania obrázkov do *VkDebugger* aplikácie.



Obr. 5.9: Nesprávne vykresľovanie v nástroji VkDebugger[13]

[12]Naľavo je aplikácia čo po skončení behu správne uvoľnuje svoje zdroje, zatiaľ čo napravo aplikácia vynechala jedno uvoľnenie na jeden zo svojich VkMemory objekov

[13]Ako vidieť, obrázok po zobrazení v ladiacom nástroji (naľavo) nezodpovedá obrázku vykresľovaným aplikáciou (napravo)

Okrem konštantného rizika, že sú v aplikácií iné chyby vyskytujúcich sa iba na určitých GPU kartách sa prejavili chyby na iných miestach, ktoré trochu znepríjemňujú používanie tejto aplikácie.

Konkrétne ide o nemožnosť využívať *breakpoint* funkcionalitu ak je aplikácia spustená pomocou nástroja *vkconfig* – hoci sa to dá obísť tak, že aplikáciu spustí užívateľ mimo konfigurátoru, toto komplikuje potom uzatváranie aplikácie. Ide o chybu v znení „Mutex destroyed while busy“ – to sa deje pretože volanie v programe prešlo vrstvou smerom na ovládače a uzamklo *mutex* ktorý istí bezpečnosť pri používaní vlákien.

Asi ale najväčšia slabina projektu spočíva vo funkcionalite prehliadania parametrov funkcií a ich hodnôt v programe, dôvodom je komplikovanosť Vulkan API – počiatočný návrh pre generovanie kódu pre túto funkcionalitu sa ukázal byť nedostačujúci.

## 5.2 Vyhodnotenie ladiaceho nástroja VkDebugger

Vzhľadom na povahu práce bola implementácia otestovaná v rámci možností celkom slušne. Experimenty totiž zahrňovali využitie všetkej funkcionality k odhaľovaniu chýb na ladenej aplikácií, bolo využité čo najrôznejších aplikácií, vrátane zanesenia chýb do nich a napokon bolo všetko zopakované na rade rôznych grafických kariet, ktoré veci zámerne skomplikovali.

Takéto húževnaté testovanie však nepochybne odhalilo aj slabé stránky implementácie, najdôležitejšie na tom je to, že sa ich riešeniu treba venovať ešte pred tým ako by sa mala programovať nová funkčnosť.

Napriek tomu ale program stále nachádza svoje využitie, najmä pri analýze grafických aplikácií ktoré sa zaoberajú s precíznym využitím *VkMemory*, *VkBuffer* a *VkImage* objektov. Okrem toho je vážne dobré pozastavovať ladený program pomocou *breakpoint* funkcionality, príkladom môže byť i napríklad zastavenie programu keď FPS klesne pod určitú hranicu. Taká vymoženosť dovoľuje užívateľovi analyzovať stav aplikácie hneď v tomto kritickom bode, bez potreby reštartu a hlavne bez znalostí slabín daného programu. Najspoľahlivejšia je ale vždy história volaní Vulkan funkcií, ich vzťah k objektom funkcionalít ktorú si užívateľ môže prispôsobiť kedy si zvolí spôsob akým sa bude zaplňovať každý *frame*.

Za účelom teda aj prezentovať prácu na internete je celý projekt zverejnený na GIT repozitári[14].

[14] Bližšie informácie o tom v prílohe A

# Kapitola 6

# Záver

Náplňou tejto práce bolo hlbšie preštudovať moderné GPU karty, Vulkan API, Loader a Layers, vrátane moderných riešení pre ladenie grafických aplikácií – to všetko za cieľom potom navrhnúť a vytvoriť vlastný nástroj pre ladenie, špecializovaný práve na Vulkan aplikácie.

Napokon bol cieľ i splnený – na základe znalostí získaných pri štúdiu tejto problematiky boli identifikované nedostatky riešení, na základe ktorých stojí celý návrh aplikácie. Tento návrh bol implementovaný v podobe Vulkan vrstvy pripojenú na aplikáciu, ktorá zobrazuje stav ladeného programu, a to priamo za jeho behu. Súčasťou stavu, ktorý je možný prehliadať sú okrem histórie Vulkan volaní aj objekty typov *VkMemory*, *VkBuffer* a *VkImage* – využívajú sa pre správu pamäti programu, uchovaniu dát o textúrach a podobne. Veľkou výhodou tejto práce je, že má možnosť nie iba zobrazovať stav aplikácie ale aj potenciálne odhaliť jeho nedostatky v kritických bodoch, ako napríklad pád aplikácie alebo nízky *frame rate*. Pre účely testovania sa vybrali rôznorodé grafické Vulkan programy, aby *VkDebugger* analyzoval ich stav počas behu – a to vrátane i na rôznych grafických kartách.

Nepochybne, medzi najviac zaujímavé zistenia patrí fakt, že pomocou týchto vrstiev je možné meniť chovanie Vulkan programov, a to potenciálne i pre zlomyseľné účely. Okrem toho, programovať vrstvu bolo v určitých smeroch obtiažne, pretože ak sa s niečim nezachádzalo opatrne, mohlo ľahko dôjsť k pádu „hostiteľského“ programu, dalo by sa povedať, že to je krok ďalej od symbiózy a krok bližšie k parazitizmu, čo bola jedna z vecí na ktorú bolo treba neustále myslieť a snažiť sa tomu zabrániť.

Osobne, vidím dva smery vývoja kam sa môže práca ďalej posúvať. Prvý z nich je ten bezpečnejší, teda pokračovať v rozširovaní funkcionality pre tento nástroj aby mohol pokryť ešte viac a bližšie do detailu stav analyzovaného Vulkan programu. Druhá možnosť, tá zábavnejšia – keďže je možné zmeniť parametre volaní na úrovni vrstvy, napadlo mi, že by sa možno dala vyvinúť aplikácia, ktorá by počas behu dokázala znovu vytvoriť alebo odstrániť nejaký Vulkan objekt. Napríklad že by programátor potreboval z nejakej scény vymazať všetky stromy, dočasne, za cieľom zistiť ako veľmi sú náročné čo sa týka výkonu. Ale žiaľ nemám dosť skúseností posúdiť obtiažnosť, reálne využitie alebo či sa niečoho takého dá vôbec dosiahnuť. Bakalárska práca ma nanajvýš obohatila a moje nadšenie stále pretrváva – ak dostanem možnosť pokračovať na jednej z týchto možností počas môjho nadväzujúceho štúdia, bez zaváhania toho moc rád využijem.

# Literatúra


[1] Hughes, J. et al. *Computer Graphics: Principles and Practice*. 3. vyd. Addison-Wesley Professional, 2013. ISBN 0321399528.

[2] Karlsson, B. *RenderDoc* [online]. https://renderdoc.org/ [cit. 2024-04-24]. Dostupné z: https://renderdoc.org/docs/index.html.

[3] Karlsson, B. a Kuhlmann, J. *Brief guide to Vulkan layers* [online]. https://renderdoc.org/ [cit. 2024-04-23]. Dostupné z: https://renderdoc.org/vulkan-layer-guide.html.

[4] Khronos Group. *Architecture of the Vulkan Loader Interfaces* [online]. https://www.khronos.org/ [cit. 2024-04-23]. Dostupné z: https://github.com/KhronosGroup/Vulkan-Loader/blob/main/docs/LoaderInterfaceArchitecture.md.

[5] Lapinski, P. *Vulkan Cookbook*. Packt Publishing Limited, 2017. ISBN 1786468158.

[6] Laramee, R. S. Using visualization to debug visualization software. *IEEE computer graphics and applications*. 2010, zv. 30, č. 6, s. 67–73. DOI: 10.1109/MCG.2009.154. Dostupné z: https://doi.org/10.1109/MCG.2009.154.

[7] Miller, S. *AMD Instinct™ MI200 GPU memory space overview* [online]. https://www.amd.com/, jún 2023 [cit. 2024-04-18]. Dostupné z: https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-mi200-memory-space-overview/.

[8] NVIDIA Corporation. *Nsight Graphics* [online]. https://www.nvidia.com/ [cit. 2024-04-25]. Dostupné z: https://docs.nvidia.com/nsight-graphics/index.html.

[9] NVIDIA Corporation. *A Deeper Look At VRAM On GeForce RTX 40 Series Graphics Cards* [online]. 2024 [cit. 2024-04-18]. Dostupné z: https://www.nvidia.com/en-us/geforce/news/rtx-40-series-vram-video-memory-explained/.

[10] NVIDIA Corporation. *Life of a triangle - NVIDIA's logical pipeline* [online]. 2024 [cit. 2024-04-16]. Dostupné z: https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline.

[11] Overvoorde, A. *Vulkan tutorial* [online]. vulkan-tutorial.com [cit. 2024-04-11]. Dostupné z: https://vulkan-tutorial.com/.

[12] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E. et al. GPU Computing. *Proceedings of the IEEE*. 2008, zv. 96, č. 5, s. 879–899. DOI: 10.1109/JPROC.2008.917757. Dostupné z: https://doi.org/10.1109/JPROC.2008.917757.

[13] Sellers, G. a Kessenich, J. *Vulkan programming guide: The official guide to learning vulkan.* 1. vyd. Addison-Wesley Professional, 2016. ISBN 0134464540.

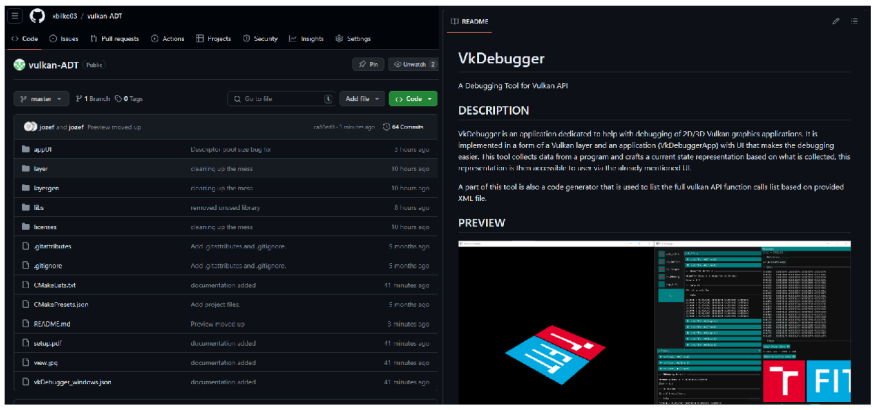
[14] Unterguggenberger, J., Kerbl, B. a Wimmer, M. Vulkan all the way: Transitioning to a modern low-level graphics API in academia. *Computers & Graphics.* 2023, zv. 111, s. 155–165. DOI: 10.1016/j.cag.2023.02.001. Dostupné z: https://doi.org/10.1016/j.cag.2023.02.001.

[15] Wright, R. *Introducing the new Vulkan Configurator* [online]. LunarG, August 2020. 8 s. Dostupné z: https://www.lunarg.com/wp-content/uploads/2020/08/Intro-to-Vulkan-Configurator_08_2020.pdf.

[16] Zeller, A. *Why Programs Fail: A Guide to Systematic Debugging.* 2. vyd. Morgan Kaufmann, 2009. ISBN 0123745152.

# Príloha A

# Snímok obrazovky nahrania práce na internet




Obr. A.1: verejne dostupný kód projektu na repozitári GIT[1]

[1] https://github.com/xbilko03/vulkan-ADT

# Príloha B

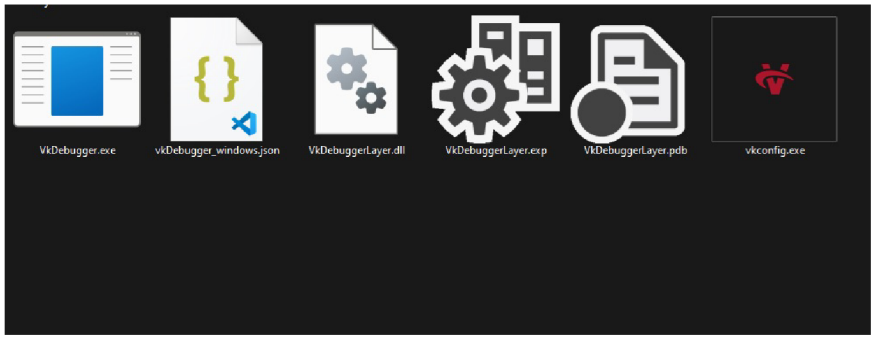
# Obsah priloženého pamäťového média

- **BilkoBP.pdf** – súbor textovej časti bakalárskej práce
- **latexBilkoBP** – zložka obsahujúca zdrojové LaTeX súbory použité k vytvoreniu písomnej časti bakalárskej práce
- **out** – zložka s obsahom binárnych súborov nástroja VkDebugger, v podobe aplikácie **VkDebugger.exe**, ladiacej vrstvy zloženej z **.json**, **.dll**, **.exp**, **.pdb** súborov
- **vkconfig.exe**[1] – program pre spustenie aplikácie **VkDebugger**
- **src** – zložka so zdrojovými súbormi projektu **VkDebugger**
- **test** – zložka s vybranými Vulkan aplikáciami použité pri testovaní
- **setup.pdf** – súbor popisujúci postup spustenia
- **README.md** – súbor popisujúci základné informácie o projekte

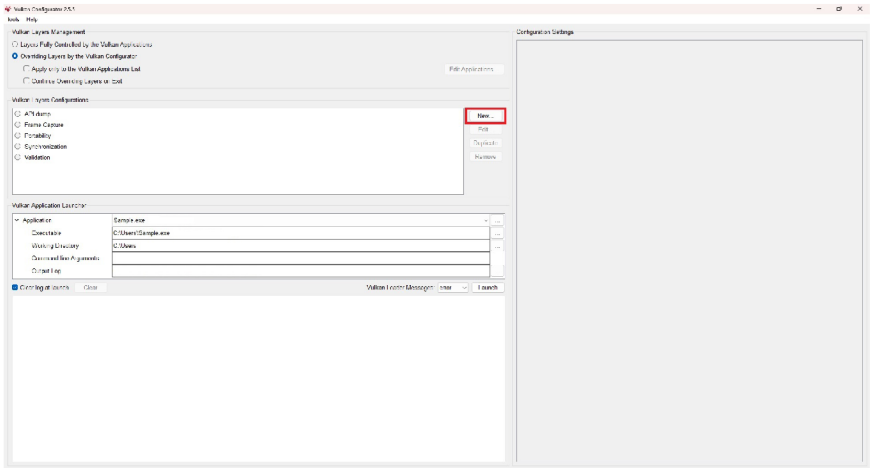
[1] Vydaný spoložnosťou *LunarG* v roku 2020 https://github.com/LunarG/VulkanTools/tree/main/vkconfig

# Príloha C

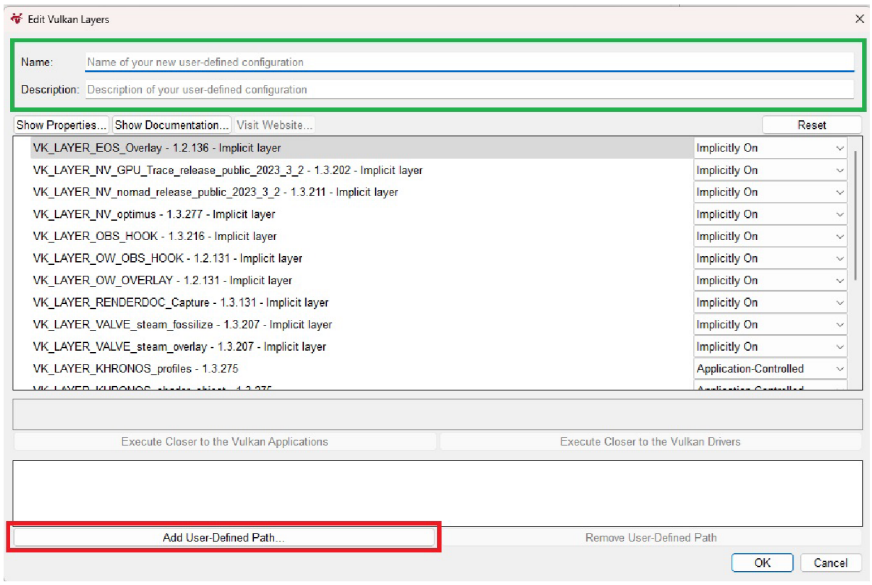
# Postup pre spustenie programu VkDebugger



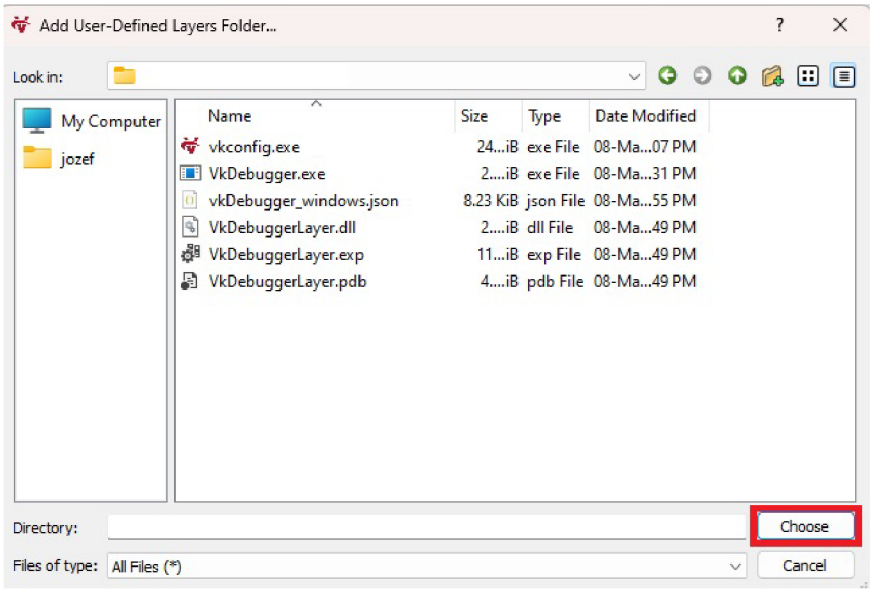

1. otvoriť zložku s **VkDebugger.exe** aplikáciou, nástrojom **vkconfig.exe** a vrstvou v podobe súborov **.json**, **.dll**, **exp** a **.pdb** a spustiť program **VkDebugger.exe**



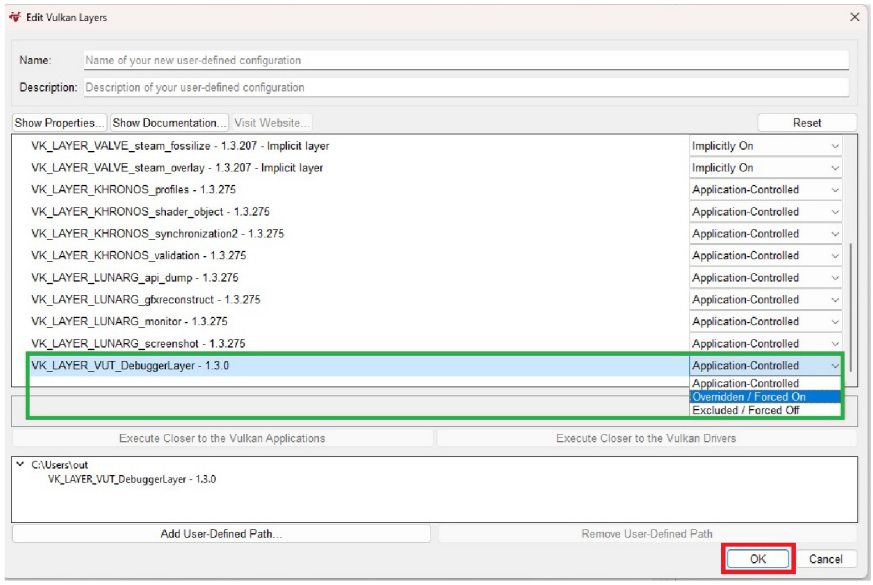
2. zvoliť možnosť vytvoriť novú konfiguráciu na tlačítko **new**



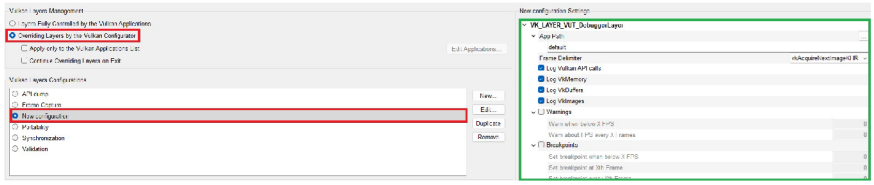

3. nastaviť meno konfigurácie a pridať nový **User-Defined Path...**



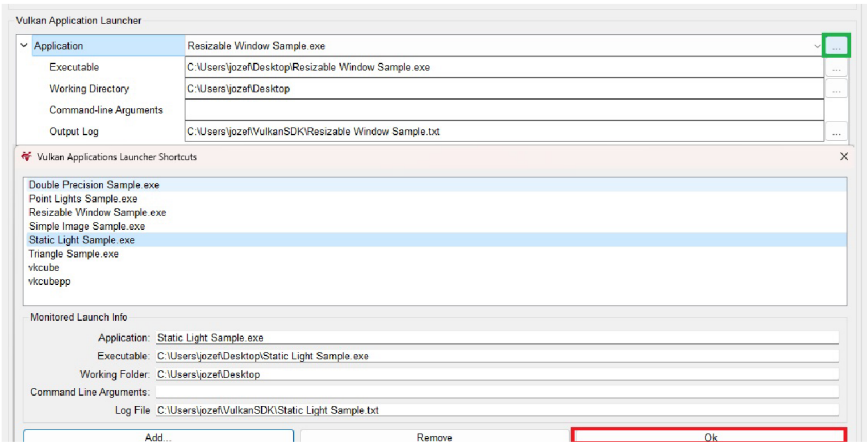

4. navigovať do zložky s s **VkDebugger.exe** aplikáciou, nástrojom **vkconfig.exe** a vrstvou v podobe súborov **.json**, **.dll**, **exp** a **.pdb** a potvrdiť výber tlačítkom **Choose**




5. nájsť v zozname nástrojov **VK_LAYER_VUT_DebuggerLayer** a zvoliť možnosť **Overriden / Forced On**. Výber potvrdiť tlačítkom OK

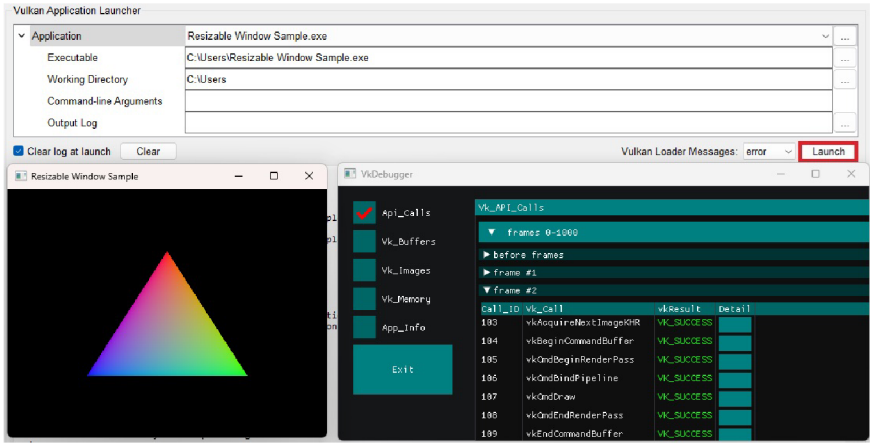


6. Nastaviť vrstvu ako aktívnu a a zvoliť novo vytvorenú konfiguráciu. Pre nastavenie vrstvy VkDebuggerLayer sa využíva sekcia *[Meno Konfigurácie]*[1] Settings




7. v sekcii **Vulkan Application Launcher** nastaviť ladenú aplikáciu a potvrdiť tlačítkom **OK**

[1] v tomto prípade *New configuration* Settings




7. Teraz počas doby behu aplikácie **vkconfig.exe** je **VkDebugger** aktívny. Pre spustenie ladenia stačí spustiť aplikáciu tlačítkom **Launch** – Hurá do ladenia![2]

[2]V prípade ak by sa **VkDebugger** nespustil, je pravdepodobné, že v počítači chýbajú niektoré knižnice C++ *.dll* – možné stiahnuť napríklad na https://www.microsoft.com/en-gb/download/details.aspx?id=48145