Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Využití API Vulkan v kontextu výuky počítačové grafiky

Bakalářská práce

Autor: Dominik Prokop
Studijní obor: Aplikovaná informatika / 1802R001

Vedoucí práce: Ing. Bruno Ježek, Ph.D.

Hradec Králové Duben 2024

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 8.4.2024 Dominik Prokop

Poděkování:
Tímto bych rád poděkoval vedoucímu mé bakalářské práce Ing. Brunovi Ježkovi, Ph.D. za jeho cenné rady, metodické vedení práce a věnovaný čas. Rád bych také poděkoval mé rodině za podporu a obětavost, které byly zásadní pro dokončení této práce.

Abstrakt

Bakalářská práce se zabývá možnostmi a potenciálem grafického rozhraní Vulkan jako možného nástroje pro výuku počítačové grafiky. Teoretická část této práce se věnuje představení rozhraní Vulkan a následnému porovnání několika volně dostupných knihoven, napříč různými programovacími jazyky, které umožnují práci s tímto grafickým rozhraním. Dále je pozornost věnována krátkému představení programovacího jazyka Rust s důrazem na jeho unikátní a inovativní přístup ke správě paměti. Praktická část je následně zaměřena na představení základních principů práce s knihovnou Vulkano (Rust). Za pomoci této knihovny je vytvořeno celkem dvacet jedna různě náročných ukázkových úloh, které společně pokrývají široké spektrum různě náročných konceptů, primárně z oblasti počítačové grafiky. Samotné ukázkové úlohy jsou doplněny o rozsáhlou dokumentaci (tutoriál), vytvořenou za pomoci nástroje Writerside. Dokumentace slouží jako podkladový (výukový) materiál, který představuje základní práci s knihovnou Vulkano. Poslední část této práce je věnována testování kompatibility všech vytvořených ukázkových úloh s vybranými operačními systémy, konkrétně Windows, macOS a Linux.

Klíčová slova:

API Vulkan, OpenGL, Knihovna Vulkano, Rust, Ash, Počítačová grafika, Výuka

Abstract

Title: Utilization of the Vulkan API in the context of teaching computer graphics

This bachelor's thesis explores the possibilities and potential of the API Vulkan as a tool for teaching computer graphics. The theoretical part of this work primarily focuses on introducing the Vulkan interface and comparing several freely available libraries, across various programming languages, that allow working with this interface. Furthermore, attention is paid to a brief introduction of the programming language Rust, emphasizing its unique and innovative approach to memory management. The practical part is focused on presenting the basic principles of working with the Vulkano library (Rust). With the help of this library, a total of twenty-one variously challenging example tasks are created, which together cover a broad spectrum of different concepts, primarily in the field of computer graphics. The example tasks are accompanied by extensive documentation (tutorial), created using the Writerside tool. This documentation primarily serves as a educational material, introducing the basic work with the Vulkano library. The last part of this work is devoted to testing the compatibility of all created example tasks with selected operating systems, specifically Windows, macOS, and Linux.

Key words:
API Vulkan, OpenGL, Vulkano Library, Rust, Ash, Computer graphics, Education

Obsah

1 Úvod 1
2 Cíl a metodika práce 2
3 API Vulkan 4
3.1 Historie a vývoj 4
3.2 Porovnání s OpenGL 7
3.3 Zobrazovací řetězec 8
3.4 Shadery 10
3.4.1 Vertex shader 10
3.4.2 Tesselation shaders 11
3.4.3 Geometry shader 12
3.4.4 Fragment shader 12
3.4.5 Compute shader 12
3.4.6 Ray tracing shaders 13
3.5 Představení základních pojmů 13
3.5.1 Instance 13
3.5.2 Fyzické zařízení 14
3.5.3 Logické zařízení 14
3.5.4 Swap chain 14
3.5.5 Render pass 15
3.5.6 Framebuffer 15
3.5.7 Command buffer 16
4 Vybrané knihovny pro práci s API Vulkan 17
4.1 LWJGL 17
4.2 PyVulkan 17
4.3 Ash 18
4.4 Vulkano 18
4.5 WGPU 18

4.6 Silk.NET 19
4.7 Veldrid 19
5 Programovací jazyk Rust 23
5.1 Představení jazyka 23
5.2 Popularita 24
5.3 Systém vlastnictví a výpůjčky 24
6 Návrh a implementace ukázkových úloh 26
6.1 Inicializace projektu 29
6.1.1 Instalace VulkanSDK 29
6.1.2 Instalace MoltenVK 29
6.1.3 Instalace programovacího jazyka Rust 29
6.1.4 Vývojové prostředí 29
6.1.5 Vytvoření projektu 30
6.1.6 Použité knihovny 30
6.2 Implementace základní vykreslovací smyčky 32
6.2.1 Okno aplikace 32
6.2.2 Instance 32
6.2.3 Fyzické a logické zařízení 33
6.2.4 Swap chain 34
6.2.5 Render pass, Frame buffer 34
6.2.6 Načítání shaderů 35
6.2.7 Grafická pipeline 35
6.2.8 Deskriptory 36
6.2.9 Command buffer 36
6.3 Knihovna vulkano_text_renderer 37
6.4 Dokumentace 39
6.5 Představení jednotlivých ukázkových úloh 40
7 Testování ukázkových úloh 47

8 Shrnutí a diskuse výsledků 52
9 Závěry a doporučení 54
10 Seznam použité literatury 56
11 Přílohy 58

Seznam obrázků

Obr. 1: Časová osa postupného vývoje grafického rozhraní OpenGL až po rozhraní Vulkan 5

Obr. 2: Porovnání rozdílného přístupu ke komunikaci aplikací využívajících rozhraní OpenGL a Vulkan s hardwarem 7

Obr. 3: Zjednodušené schéma zobrazovacího řetězce, zobrazující části fixního bloku modře a programovatelného bloku (shadery) bíle 9

Obr. 4: Rozdělení trojúhelníků za pomoci teselace 11

Obr. 5: Diagram základních objektů používaných v rozhraní Vulkan 13

Obr. 6: Diagram zobrazující vztahy mezi přílohami a podprůchody 15

Obr. 7: Diagram zobrazující vztah mezi framebufferem a přílohami definovanými v render pass 16

Obr. 8: Vývoj počtu hvězdiček jednotlivých knihoven v průběhu času 21

Obr. 9: Ukázka p01_basic_triangle – Vykreslení 2D trojúhelníku 40

Obr. 10: Ukázka p03_geometry – Vykreslení 3D objektů do okna aplikace 41

Obr. 11: Ukázka p04_textures – Mapování textur na různé tvary 41

Obr. 12: Ukázka p06_obj – Čajová konvice načtená z formátu OBJ 42

Obr. 13: Ukázka p07c_offset_rendering – Simultánní vykreslení objektů 43

Obr. 14: Ukázka p08_light – Implementace Phongova osvětlovacího modelu 43

Obr. 15: Ukázka p11_geometry_shader – Transformace sféry za pomoci geometry shaderu 44

Obr. 16: Ukázka p13_post_processing – Efekt barevného filtru 45

Obr. 17: Ukázka p14b_tesselation – Sféra vzniklá za pomoci teselace 46

Obr. 18: Ukázka p15a_deferred_shading – Výpočet osvětlení s využitím techniky deferred shading 46

Obr. 19: Ukázka vzhledu dokumentace – „Instalace a nastavení“ 62

Obr. 20: Ukázka vzhledu dokumentace – „Ukázkové úlohy“ 62

Obr. 21: Ukázka vzhledu dokumentace – „Načítání shaderů“ 63

Obr. 22: Ukázka vzhledu dokumentace – „Dodatečné informace“ 63

Obr. 23: Ukázka vzhledu dokumentace – „Rozdíly Java vs. Rust“ – dark mode 64

Seznam tabulek

Tabulka 1: Přehled základních informací o vybraných knihovnách, které umožnují využívat rozhraní Vulkan..... 20

Tabulka 2: Popularita a aktuální stav vývoje jednotlivých knihoven..... 21

Tabulka 3: Licence, pod kterými jsou jednotlivé knihovny distribuovány..... 22

Tabulka 4: Základní seznam všech implementovaných ukázkových úloh..... 28

Tabulka 5: Funkčnost ukázkových úloh na vybraných operačních systémech..... 48

Tabulka 6: Porovnání různých scénářů využití knihovny vulkano_text_renderer.. 50

Seznam zkratek

Zkratka	**Plný název**
API	**A**pplication **P**rogramming **I**nterface
DOCX	**Doc**ument Open **X**ML
EXE	**Exe**cutable File
FPS	**F**rames **P**er **S**econd
GLSL	Open**GL** **S**hading **L**anguage
HLSL	**H**igh-**L**evel **S**hading **L**anguage
JPG	**J**oint **P**hotographic Experts **G**roup
OBJ	**Obj**ect File
PDF	**P**ortable **D**ocument **F**ormat
PNG	**P**ortable **N**etwork **G**raphics
SPIR-V	**S**tandard **P**ortable **I**ntermediate **R**epresentation – **V**ersion X
WEBP	**Web** **P**icture Format
XML	E**x**tensible **M**arkup **L**anguage

1 Úvod

Oblast počítačové grafiky prochází již několik desetiletí dynamickým vývojem, který je primárně poháněn rychlým vývojem technologií. Spolu s tímto vývojem postupně přišly i nezanedbatelné změny v architektuře hardwaru a výrazný nárůst výkonu prakticky všech počítačových komponent (grafická karta, procesor, operační paměť, disky, …). Tento rychlý vývoj spolu s mnoha dalšími inovacemi a změnami přispěl k rozšiřování možností jednotlivých vývojářů, kteří se postupem času mohli zaměřovat na tvorbu stále výkonnostně náročnějších a rozsáhlejších projektů.

Aby bylo možné efektivně a relativně jednoduše využívat všech výhod, jež tyto změny a inovace postupně přinesly, bylo nutné vyvinout nástroj, který to bude umožnovat. Za jeden z takovýchto nástrojů lze považovat grafické rozhraní OpenGL, jež bylo oficiálně představeno v roce 1992. Toto grafické rozhraní poskytlo vývojářům přímou a relativně flexibilní kontrolu nad samotným hardwarem, což jim umožnilo využít potencionál daných zařízení (počítač, notebook). Od první dekády tohoto tisíciletí můžeme být svědky monumentálního vzestupu mobilních zařízení, která jsou postavena na zcela odlišné architektuře než většina klasických stolních počítačů případně notebooků. Postupem času začalo být stále více zřejmé, že OpenGL již posloužilo svému účelu a je nutné vytvořit zcela nový nástroj, který bude reflektovat technologický vývoj i změny v architektuře hardwaru. Odpovědí na tyto požadavky se stalo zcela nové grafické a výpočetní rozhraní Vulkan, které bylo uvedeno na trh v roce 2016 jako přímý nástupce rozhraní OpenGL. Hlavním cílem tohoto rozhraní bylo odstranit většinu technických omezení, která byla do té doby kladena vývojářům, a poskytnout jim co největší kontrolu nad hardwarem, bez jakékoliv zbytečné abstrakce.

I přesto, že je rozhraní Vulkan v oblasti počítačové grafiky již několik let, OpenGL si stále zachovává relativně vysokou popularitu, a to i v kontextu výuky počítačové grafiky. Hlavním faktorem, který stále brání rozhraní Vulkan se rozšířit do této oblasti, je jeho výrazně větší složitost a nevhodnost pro začínající programátory. I přes mnohé výzvy se v průběhu let vyvinulo několik více či méně populárních nástrojů, napříč různými programovacími jazyky, které se tyto problémy snaží určitým způsobem řešit. Toto všechno vyvolává jednoduchou otázku, zda by bylo možné v budoucnu použít některý z těchto nástrojů v rámci výuky počítačové grafiky.

2 Cíl a metodika práce

Cílem této bakalářské práce je prozkoumat možnosti a potencionál grafického rozhraní Vulkan jako možného nástroje pro výuku počítačové grafiky. V rámci úvodní kapitoly bude představeno rozhraní Vulkan, včetně historického vývoje, architektury zobrazovacího řetězce a porovnání s grafickým rozhraním OpenGL. V druhé kapitole budou postupně představeny a následně i porovnány vybrané volně dostupné knihovny, které umožnují využít toto grafické rozhraní. Jednotlivé knihovny budou porovnány na základě různých parametrů, jako je například používaný programovací jazyk, popularita, dostupnost dokumentace, případně licence, pod kterými jsou šířeny. V rámci třetí kapitoly bude představen programovací jazyk Rust, s důrazem na jeho unikátní přístup ke správě paměti. Tento programovací jazyk, spolu s knihovnou Vulkano, bude následně použit pro implementaci sady různě náročných ukázkových úloh, které se budou zabývat různými koncepty z oblasti počítačové grafiky. Kromě jednotlivých ukázkových úloh bude zpracována i pomocná dokumentace (tutoriál), která se bude zabývat představením práce s knihovnou Vulkano. Samotná dokumentace i jednotlivé ukázkové úlohy budou navrhnuty takovým způsobem, aby mohly být v budoucnu použity jako podkladový materiál pro výuku počítačové grafiky. Cílem je vytvořit takové ukázkové úlohy i dokumentaci, které studentům postupně přiblíží práci s knihovnu Vulkano a programovacím jazykem Rust. Funkčnost všech ukázkových úloh bude následně otestována napříč hlavními operačními systémy, jmenovitě Windows, Linux a macOS.

Hlavní inspirací pro napsání této bakalářské práce se stala studie profesora Johannese Unterguggenbergera z Technologické Univerzity ve Vídni. Jeho studie, nazvaná „Vulkan all the way: Transitioning to a modern low-level graphics API in academia“, detailně popisuje osobní zkušenost s přechodem z dlouhodobě (tradičně) používaného grafického rozhraní OpenGL k modernějšímu a flexibilnějšímu grafickému rozhraní Vulkan. Studie nabízí cenný pohled na různé technické, praktické i pedagogické aspekty, případně výhody a výzvy, které jsou přímo spojeny s takto výraznou změnou. Kromě toho studie přináší i zajímavé srovnání výsledků a osobních zkušeností studentů s vybraným rozhraním (OpenGL a Vulkan), doplněné o kritické zhodnocení dosažených výsledků a krátkého zamyšlení o budoucnosti tohoto rozhraní v rámci výuky počítačové grafiky.

Při zpracování této bakalářské práce byl důraz primárně kladen na výběr co nejaktuálnější a nejvíce relevantní literatury, odborných článků a oficiální dokumentace s cílem zabezpečit aktuálnost a přesnost informací.

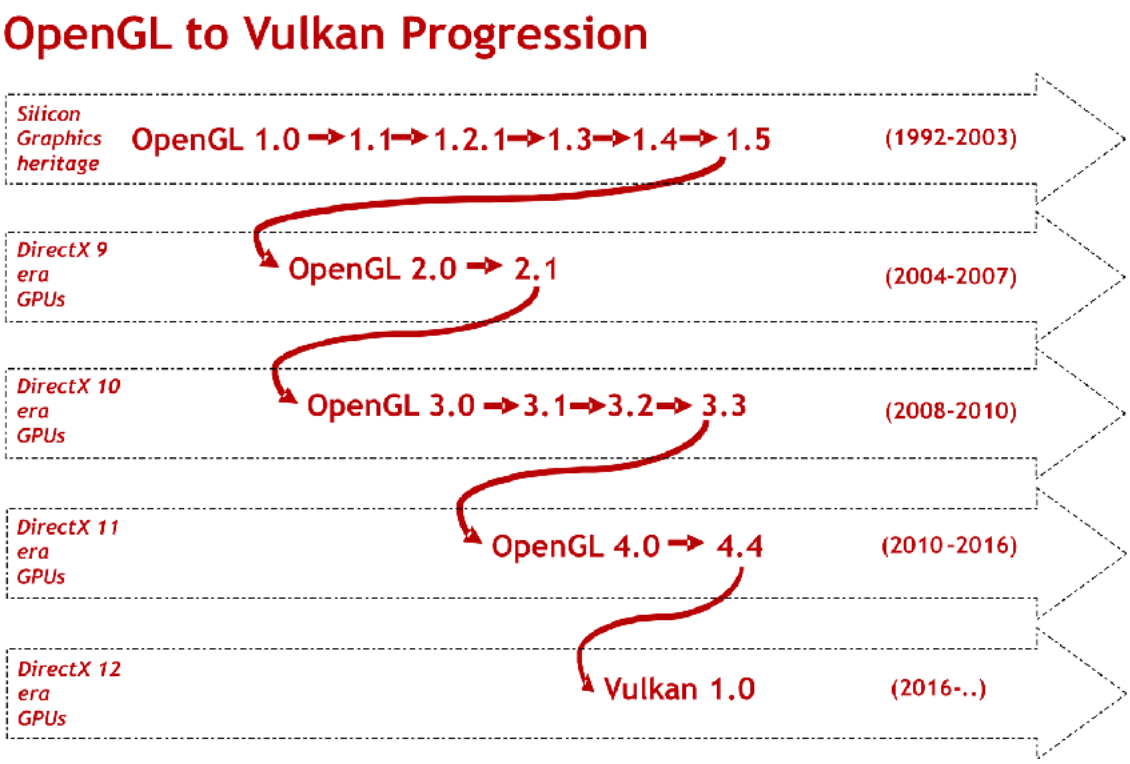
Nástroje umělé inteligence, konkrétně ChatGPT (OpenAI) a Gemini (Google), byly využívány výhradně pro účely základní kontroly pravopisu a poskytování konstruktivní zpětné vazby (například „Řekni mi, co by bylo dobré vylepšit.“, „Zhodnoť následující část textu a poskytni mi zpětnou vazbu.“, „Které části textu se špatně čtou a měl bych je přepsat?“). Tyto nástroje nebyly žádným způsobem využívány jako zdroj informací nebo jako přímý prostředek pro napsání jakékoliv části tohoto textu.

3 API Vulkan

Vulkan je nové nízkoúrovňové multiplatformní grafické a výpočetní rozhraní, vyvinuté konsorciem Khronos Group. Tato nezisková organizace sdružuje více než 180 různých společností, které se společně zabývají vytvářením standardů v oblastech, jako je počítačová grafika, virtuální a augmentovaná realita či paralelní zpracování dat. Vulkan byl specificky navržen s cílem poskytnout efektivnější a flexibilnější alternativu k již existujícím grafickým rozhraním, zejména OpenGL. Z tohoto důvodu Vulkan poskytuje znatelně větší kontrolu nejen nad implementací jednotlivých struktur, ale i nad samotným hardwarem. Vulkan je primárně určen pro vytváření graficky náročnějších aplikací, jako jsou například videohry nebo simulace, a aplikací, které vyžadují provádění vysoce paralelizovaných výpočtů na grafických kartách. V současné době se Vulkan vyznačuje širokou podporou napříč všemi běžně používanými operačními systémy. Aplikace napsané s využitím tohoto rozhraní lze bez problému spustit na operačních systémech Windows, Linux a Android, nebo v případě operačních systémů macOS, iOS a tvOS prostřednictvím speciální překladové vrstvy. [1, 2]

3.1 Historie a vývoj

První verze OpenGL byla uvedena na trh v roce 1992 společností Silicon Graphics. Toto rozhraní se rychle stalo široce přijímaným standardem napříč různými oblastmi a díky své flexibilitě a výkonu našlo uplatnění v široké škále aplikací. [3] Během mnoha let své existence se OpenGL postupně stalo základním stavebním kamenem pro mnoho více či méně graficky náročných aplikací od různých videoher přes simulace až po aplikace sloužící k vizualizaci informací. [4] Za zmínku stojí například Blender (program pro modelování objektů a vytváření animací), Google Earth (webová aplikace pro vizualizaci zemského povrchu), Minecraft (nejprodávanější počítačová hra historie) nebo herní série Doom. Během své dlouhé historie prošlo OpenGL několika více či méně významnými aktualizacemi, které postupně přinášely nové funkce, opravovaly známé chyby, případně přinášely změny v rámci celkové architektury API. [5] I přesto, že bylo OpenGL dlouhodobě populární a stále značně využívané, začalo být postupem času stále více zřejmé, že již nedokáže držet krok s neustále se vyvíjejícími technologiemi a hardwarem. Tato situace postupem času vyústila ve vývoj zcela nového rozhraní (API) Vulkan.



Obr. 1: Časová osa postupného vývoje grafického rozhraní OpenGL až po rozhraní Vulkan

Zdroj: Obrázek převzat z [3]

První verze API Vulkan byla oficiálně vydána 16. února 2016 konsorciem Khronos Group, jako reakce na změny v rámci architektury hardwaru i rostoucí potřeby a rozsah moderních aplikací. Hlavním cílem bylo vytvořit moderní multiplatformní grafické a výpočetní rozhraní, které by vyřešilo řadu problémů a limitací, které do té doby trápily existující grafická rozhraní, zejména OpenGL. Samotná architektura i mnoho jiných klíčových aspektů tohoto rozhraní přímo vychází z grafického API Mantle, které bylo vytvořeno společností AMD. Mantle bylo inovativní nízkoúrovňové grafické rozhraní výhradně určené pro grafické karty od společnosti AMD. Vulkan přebral značnou část inovací i architektury, kterou přineslo rozhraní Mantle a k nim přidal řadu dalších vlastních změn a inovací. Mezi inovace, které Vulkan přinesl, lze zařadit například vylepšení v oblastech, jako je správa paměti a systémových zdrojů, snížení komunikační zátěže mezi procesorem a grafickou kartou, nebo nutnost zkompilovat shadery do nového a flexibilnějšího formátu SPIR-V. [3, 6] Díky těmto i mnoha dalším změnám a inovacím se Vulkan stal robustnějším a flexibilnějším grafickým rozhraním, které poskytuje znatelné vylepšení výkonu a efektivnější využití systémových zdrojů.

Od svého vydání prošel Vulkan řadou různých větších i menších aktualizací, které postupně přidávaly nové funkce, zlepšovaly výkon, případně opravovaly známé chyby. V průběhu času byly vydány tři hlavní aktualizace, a to konkrétně verze 1.1, 1.2 a 1.3, společně s nespočtem menších vedlejších aktualizací.

První velká aktualizace 1.1 byla vydána v roce 2018 a přinesla řadu změn. Za zmínku stojí například [7]:

- Podpora HLSL shaderů;
- Podpora vykreslování do více oken najednou;
- Možnost využít v rámci jedné aplikace více grafických karet najednou.

Další velká verze 1.2 byla vydána v roce 2020 a primárně se zaměřovala na přidávání a vylepšování nejčastěji žádaných funkcí. Za hlavní změny lze považovat [8]:

- Standardizaci rozšíření;
- Zcela nový paměťový model;
- Vylepšenou podporu HLSL shaderů;
- Speciální timeline semafory pro jednodušší synchronizaci.

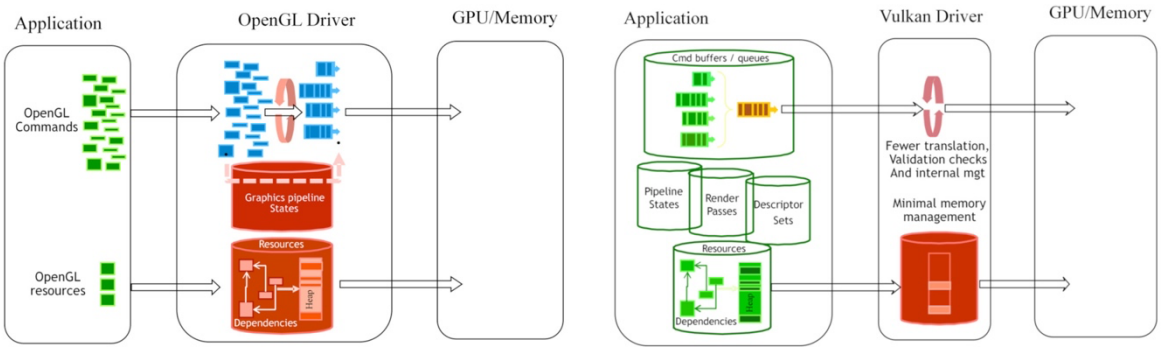
Zatím poslední velká aktualizace 1.3 byla vydána v roce 2022 a přinesla řadu menších i větších změn a vylepšení stávajících funkcí. Jedná se například o [9]:

- Přidání nových dynamických stavů;
- Možnost využít dynamické vykreslování;
- Rozšířená kontrola kompilace grafických pipeline;
- Zcela nové profily zařízení, které slouží k poskytování informací o výkonu a schopnostech daného zařízení (grafické karty).

V době psaní této práce je nejaktuálnější verze 1.3.280.0, vydaná 20. března 2024, která přináší převážně dílčí opravy chyb a menší vylepšení stávajících funkcí. [10]

3.2 Porovnání s OpenGL

Vulkan se od svého předchůdce odlišuje zcela novou a výrazně přepracovanou architekturou, která poskytuje vývojářům znatelně větší volnost a flexibilitu spojenou s detailní kontrolou nad daným hardwarem. Hlavní a klíčový aspekt této změny spočívá v úplném nahrazení tradiční fixní grafické pipeline za mnohem flexibilnější a téměř plně přizpůsobitelnou programovatelnou pipeline. Za další výraznou a důležitou změnu lze považovat využívání command bufferu. Command buffer si je možné představit jako sekvenci příkazů, které mají být najednou odeslány grafické kartě k provedení. To vede k razantnímu snížení komunikační zátěže mezi procesorem a grafickou kartou, což má za následek výrazné zvýšení výkonu aplikace a celkově lepší správu systémových zdrojů. Mezi další vylepšení lze zařadit například podporu paralelního vykreslování, lepší podporu pro vytváření vícevláknových aplikací nebo zcela nový a přepracovaný systém pro správu paměti a systémových zdrojů. Tyto ale i další změny a inovace vedly k výraznému vylepšení výkonu a poskytly vývojářům znatelně větší volnost v rámci návrhu, tvorby i optimalizace aplikací. Současně však tyto změny otevřely cestu k hledání nových inovativních způsobů, jakými lze toto rozhraní využít. [1, 3]



Obr. 2: Porovnání rozdílného přístupu ke komunikaci aplikací využívajících rozhraní OpenGL a Vulkan s hardwarem
Zdroj: Obrázek převzat z [3]

S výše uvedenými výhodami přináší Vulkan i určité výzvy, se kterými je nutné se určitým způsobem vypořádat. Velmi flexibilní přístup s minimální úrovní abstrakce vede k přenesení značné míry zodpovědnosti na samotné vývojáře, kteří jsou následně nuceni vlastnoručně implementovat všechny potřebné funkce, které budou chtít využívat. Tímto způsobem Vulkan klade vyšší požadavky na technické dovednosti a znalosti vývojářů, kteří musí mít hlubší představu o tom, jakým způsobem funguje nejen samotné rozhraní, ale i daný hardware, především grafická karta.

3.3 Zobrazovací řetězec

Zobrazovací řetězec si lze představit jako sérii nebo posloupnost za sebou jdoucích kroků, které postupně zpracovávají a transformují jednotlivá vstupní data (vrcholy, textury, …) a na základě nich vykreslují požadovanou scénu (objekty) do okna aplikace. V rámci rozhraní Vulkan je možné rozdělit zobrazovací řetězec na dva hlavní typy bloků, a to konkrétně na uživatelsky programovatelný a fixní blok. [1]

Programovatelný blok zobrazovacího řetězce

Programovatelný blok zobrazovacího řetězce se skládá z několika shaderových (hardwarových) jednotek, zkráceně shaderů, které byly z historického hlediska používány k efektivnímu výpočtu osvětlení. Samotný termín „shader" se nevztahuje pouze pro označení hardwarové jednotky, ale slouží také k označení samotného shaderového programu, který je na příslušné hardwarové jednotce vykonáván. [13] Samotné shaderové programy, zkráceně shadery, představují malé, výkonné a flexibilní programy, které jsou postupně vykonávány na vybrané grafické kartě. Tyto programy slouží k přímému řízení jednotlivých modulů v rámci programovatelného bloku zobrazovacího řetězce. Samotní vývojáři mají nad touto částí absolutní kontrolu a mohou ji libovolně upravit na základě potřeb a požadavků dané aplikace. [11]

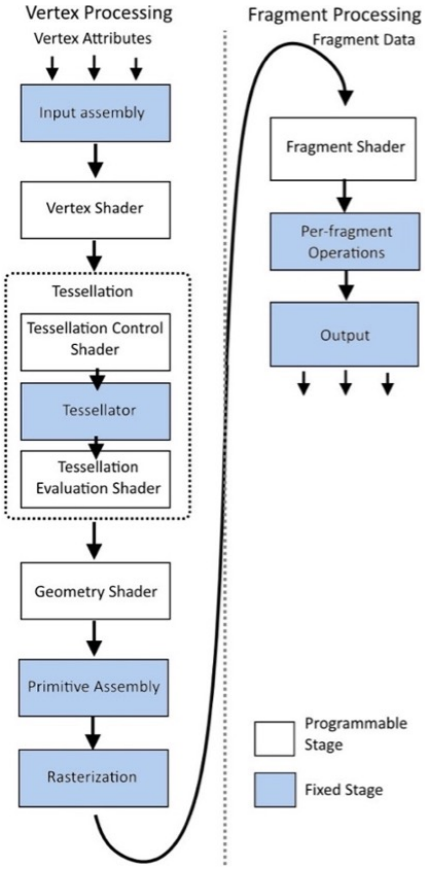
Fixní blok zobrazovacího řetězce

Funkcionalita fixního bloku zobrazovacího řetězce je typicky řešena přímo v rámci konkrétní hardwarové implementace dané grafické karty, a proto nad ní mají vývojáři jen omezenou kontrolu. Přesto mohou vývojáři stále tuto část částečně přizpůsobit potřebám konkrétní aplikace prostřednictvím konfigurace různých parametrů. Tento blok typicky zahrnuje základní, nezbytné operace, jako je rasterizace (převod 3D geometrie na 2D pixely), depth testing (určení viditelnosti objektů na základě jejich polohy a vzdálenosti od kamery), nebo blending (míšení barev). [1, 11]

Popis zobrazovacího řetězce

Aby bylo možné za pomoci zobrazovacího řetězce vykreslit určitou geometrii, je nejdříve nezbytně nutné shromáždit všechna data, která budou v rámci celého procesu potřeba. Tento krok, známý pod názvem (vertex) input state, je v rámci zobrazovacího řetězce realizován jako první a určuje, jakým způsobem mají být interpretována jednotlivá nezpracovaná data, která jsou postupně načítána z bufferů. Po počátečním zpracovaní a rozdělení vstupních dat následuje jejich předání vertex shaderu, který

představuje první programovatelnou a povinnou část zobrazovacího řetězce. Vertex shader je spouštěn vždy jednou pro každý vrchol a slouží k transformování pozice vrcholů. Každý z těchto vrcholů je součástí určitého geometrického primitiva, které může být dále volitelně zpracováno v rámci procesu teselace nebo za pomoci dnes málo používaného geometry shaderu. Teselace je další a první zcela volitelnou částí zobrazovacího řetězce, která slouží k rozdělení jednotlivých primitiv na několik menších částí. Po dokončení procesu teselace mohou být takto zpracovaná primitiva zaslána do geometry shaderu, nebo dále zpracována za pomoci procesu zvaného rasterizace. Geometry shader je další volitelnou částí zobrazovacího řetězce, která umožňuje vytvářet či upravovat jednotlivá primitiva na základě přesně stanovených pravidel. Po dokončení geometry shaderu jsou vrcholy znovu seskupeny do primitiv a za pomoci rasterizace převedeny na fragmenty. Tyto fragmenty jsou následně zpracovány v rámci fragment shaderu. Fragment shader je poslední povinnou a zcela přizpůsobitelnou částí zobrazovacího řetězce, která slouží k výpočtu finální barvy pixelů. Posledním krokem je řada pevně definovaných a částečně přizpůsobitelných operací, jako je například blending nebo další operace s fragmenty. [11, 12]



Obr. 3: Zjednodušené schéma zobrazovacího řetězce, zobrazující části fixního bloku modře a programovatelného bloku (shadery) bíle

Zdroj: Obrázek převzat z [11]

3.4 Shadery

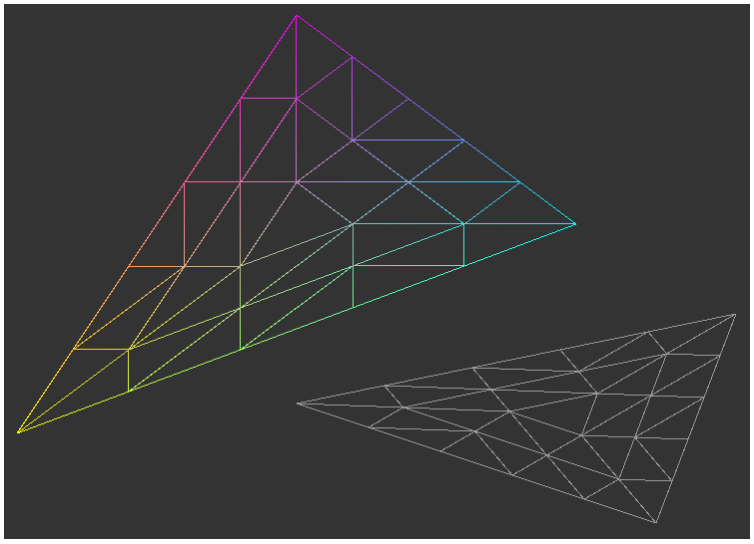
Díky shaderům mohou vývojáři relativně lehce vytvářet složitější vizuální efekty, jako je například zobrazení realistických stínů, výpočet osvětlení, animace, ray tracing a mnohé další efekty, které primárně slouží k vylepšení celkového vzhledu aplikace. Vulkan primárně podporuje shadery napsané za pomoci programovacího jazyka GLSL (OpenGL Shading Language) nebo také za pomoci jazyka HLSL (High-Level Shader Language). Aby bylo možné tyto shadery v rámci rozhraní Vulkan využít, musejí být nejdříve zkompilovány do speciálního formátu SPIR-V. SPIR-V je nízkoúrovňový multiplatformní binární formát, který byl specificky navržena tak, aby byl zcela nezávislý na operačním systému a grafické kartě. Vulkan podporuje pět základních typů shaderů, a to konkrétně vertex, fragment, tesselation, geometry a compute shader, kdy každý z těchto shaderů hraje v rámci zobrazovacího řetězce určitou roli a slouží zcela jinému účelu. Kromě těchto pěti základních shaderů Vulkan podporuje i relativně nové mesh shadery a dalších pět speciálních typů shaderů, které lze využít například k implementaci ray tracingu.

3.4.1 Vertex shader

Vertex shader je jednou z povinných programovatelných částí zobrazovacího řetězce, kterou musí každý vývojář implementovat na základě potřeb dané aplikace. Jeho primárním úkolem je transformovat pozice jednotlivých vrcholů (vertexů) z lokálních souřadnic objektu do globálních souřadnic scény. Tato transformace je nezbytná pro správné umístění jednotlivých objektů v rámci 2D i 3D prostoru. [11] Transformace jsou primárně prováděny za pomoci různých transformačních matic, díky kterým lze na jednotlivé vrcholy aplikovat různé typy transformací. Mezi základní typy transformací můžeme zařadit například posun, rotaci nebo škálování. Kromě těchto základních transformací může vertex shader provádět i jiné složitější operace, jako jsou různé typy deformací, například zkosení. Vertex shadery předávají jako výstup jednotlivé transformované vrcholy spolu s dalšími souvisejícími informacemi. Typicky se bude jednat o normálový vektor, barvu, polohu vrcholu, souřadnice do textury, případně další uživatelsky definované atributy. Tyto informace mohou být následně využity v rámci dalších programovatelných částí zobrazovacího řetězce k určení finální barvy určitého pixelu.

3.4.2 Tesselation shaders

Teselační shadery jsou speciální typy shaderů, jejichž použití není v rámci zobrazovacího řetězce povinné, a lze je zcela přeskočit. Na rozdíl od geometry shaderu, který manipuluje s již existujícími primitivy, slouží teselační shadery na rozdělení určité geometrie, typicky trojúhelníku, na několik menších částí. [11] Tento proces, známý pod pojmem teselace, umožnuje různým typům aplikací, primárně videohrám, vytvářet složitější geometrii s vysokou úrovní detailu, bez nutnosti definovat vysoký počet vrcholů. Teselaci je možné využít mnoha způsoby od generování terénu, přes simulaci vodní hladiny, až po implementaci různých metod level of detail. Teselace se skládá ze dvou základních a přizpůsobitelných částí, a to konkrétně z tessellation control shader a tessellation evaluation shader.



Obr. 4: Rozdělení trojúhelníků za pomoci teselace
Zdroj: Vlastní zpracování

Tessellation control shader

Tesselation control shader je první programovatelnou částí v rámci procesu teselace. Jeho primárním účelem je definování a specifikace všech potřebných parametrů, na základě kterých bude určen rozsah i samotný průběh teselace. [11]

Tessellation evaluation shader

Tesselation evaluation shader je druhou programovatelnou částí v rámci procesu teselace. Tento shader je spuštěn vždy až po samotném rozdělení jednotlivých geometrických primitiv a slouží k relativně podobnému účelu jako vertex shader. Konkrétně tedy slouží k určení finální polohy jednotlivých vrcholů, které ale byly vytvořeny na základě samotného procesu teselace. [11]

3.4.3 Geometry shader

Geometry shader je speciální a v rámci zobrazovacího řetězce nepovinným typem shaderu, který slouží k manipulaci s různými geometrickými primitivy, jako jsou body, čáry a trojúhelníky. Jedním ze zásadních rysů tohoto shaderu je jeho schopnost modifikovat již existující primitiva, případně vytvářet primitiva zcela nová. [1] Díky tomu je možné, aby do tohoto shaderu vstoupil zcela jiný počet nebo typ primitiv než z něho nakonec vystupuje. Geometry shadery lze využít k implementaci různých grafických efektů, jako je například voxelizace (převod scény (objektů) na voxely), teselace (rozdělení geometrie na menší části) nebo morphing (přeměna objektu z jedné formy do jiné).

3.4.4 Fragment shader

Fragment shader představuje poslední povinnou část celého zobrazovacího řetězce. Tento shader je vždy prováděn pro každý fragment, který je získán na základě procesu rasterizace jednotlivých geometrických primitiv. Jeho primárním úkolem je stanovení finální barvy každého pixelu obrazu před tím, než bude daný obraz zobrazen v okně aplikace. [11] Fragment shader může volně přistupovat k mnoha hodnotám, které jsou automaticky interpolovány na základě dat získaných z vertex a geometry shaderu. Ve většině případů se bude jednat o barvu fragmentu, souřadnice do textury, normálový vektor, případně relativní pozici daného fragmentu. Díky vysoké flexibilitě tohoto shaderu je možné vytvářet širokou škálu různě složitých vizuálních efektů. Patří sem například výpočet osvětlení, realistická simulace různých materiálů, mapování textury či implementace pokročilejších post-processing efektů. To umožnuje vývojářům vytvářet vizuálně přitažlivé aplikace s relativně nízkým dopadem na celkový výkon, což je zásadní pro naprostou většinu aplikací, včetně videoher.

3.4.5 Compute shader

Compute shader je speciálním typem shaderu, který na rozdíl od všech ostatních shaderů není přímou součástí zobrazovacího řetězce. Jak již název napovídá, hlavním účelem tohoto shaderu je provádění širokého spektra matematických a výpočetních operací, přímo na grafické kartě. [11] Na rozdíl od ostatních shaderů se tedy compute shader nezaměřuje na vizuální aspekt aplikace, ale výhradně se zaměřuje na provádění požadovaných výpočtů. Výstup tohoto shaderu může být následně dále předán jiným shaderům nebo může být uložen do bufferu pro pozdější použití. Compute shader je

možné využít mnoha různými způsoby, například k simulaci fyzikálních jevů, generování částic, provádění různě složitých matematických výpočtů nebo ke zpracování a modifikaci dat, například k dekódování a zpracování videa.

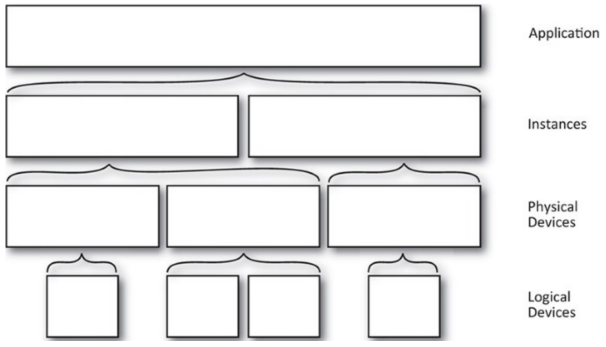
3.4.6 Ray tracing shaders

Ray tracing je pokročilá technika používaná v oblasti počítačové grafiky, díky které je možné poskytnout extrémně realistické zobrazování různých světelných či optických efektů, stínů a odrazů. Tato metoda je založena na principu sledování cesty světelných paprsků od zdroje až k pozorovateli. Většina moderních grafických karet od předních výrobců, jako jsou Nvidia a AMD, v současné době poskytuje podporu pro hardwarově akcelerovaný ray tracing. Tato technologie výrazně urychluje výpočty prováděné v rámci ray tracingu, který vyžaduje značnou výpočetní kapacitu a je časově relativně velmi náročný. Díky této technologii je možné dosáhnout realistického zobrazení osvětlení, stínů, případě různých typů odrazů, a to zcela v reálném čase. Samotný Vulkan podporuje ray tracing prostřednictví speciálního rozšíření, které umožnuje vývojářům využívat speciální ray tracing pipeline spolu s několika typy shaderů, které jsou přímo určené pro implementaci ray tracingu a sledování paprsků. [14]

3.5 Představení základních pojmů

3.5.1 Instance

Instance je objekt, který funguje jako komunikační spojení (kanál) mezi určitou aplikací a knihovnou Vulkan. V rámci instance mohou být uchovávány různé globální informace (stavy) o dané aplikaci, jako je například název a aktuální verze aplikace, minimální/maximální požadovaná verze rozhraní Vulkan nebo seznam požadovaných rozšíření. Aplikace mohou obsahovat i více na sobě zcela nezávislých instancí, které mohou obsahovat různé sady objektů. [1, 11]



Obr. 5: Diagram základních objektů používaných v rozhraní Vulkan
Zdroj: Obrázek převzat z [1]

3.5.2 Fyzické zařízení

Fyzické zařízení představuje libovolnou hardwarovou komponentu, která podporuje rozhraní Vulkan. Nejčastěji se jedná o grafickou kartu, ale v praxi se může jednat o jakoukoliv jinou komponentu, jako je například procesor s integrovanou grafickou kartou. Každé fyzické zařízení podporuje specifickou sadu volitelných rozšíření a poskytuje přístup k rozsáhlému seznamu různých specifikací. Tyto specifikace mohou být využity k vyhledání nejvhodnějšího fyzického zařízení. Výběr vhodného fyzického zařízení představuje velice důležitý úkol, který může mít výrazný dopad na celkový výkon a funkčnost celé aplikace. Je důležité, aby bylo vybráno takové fyzické zařízení, které nejenže splňuje předem definované požadavky aplikace, ale i podporuje všechna vyžadovaná rozšíření a poskytuje dostatečný výkon pro plánované činnosti. [1, 11]

3.5.3 Logické zařízení

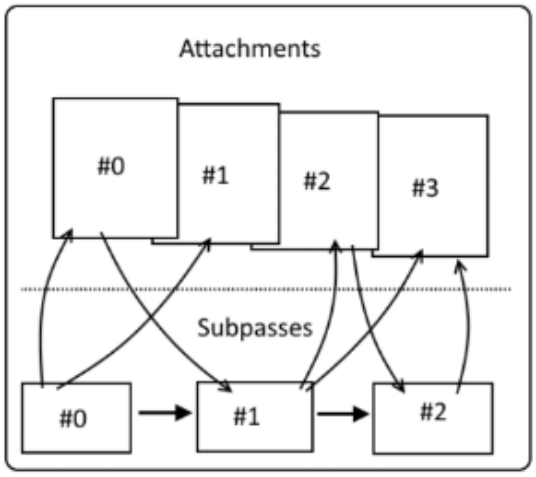
Po výběru vhodného fyzického zařízení je nutné vytvořit korespondující logické zařízení, které funguje jako abstraktní vrstva, která umožnuje komunikaci s daným hardwarem. Při vytváření logického zařízení je možné specifikovat seznam vyžadovaných rozšíření, která mají být automaticky aktivována na korespondujícím fyzickém zařízení. Rozšíření poskytují přímý přístup k velkému množství speciálních funkcí a nástrojů, které standardně nemusejí být dostupné a funkční na všech fyzických zařízeních. Jedná se například o možnost využít pokročilé nástroje pro ladění a diagnostiku chyb, podporu pro ray tracing či různé speciální typy shaderů nebo možnost pracovat se specializovanými datovými typy. Z tohoto důvodu je důležité použít seznam požadovaných rozšíření jako filtr pro výběr vhodného fyzického zařízení. [11]

3.5.4 Swap chain

Swap chain si je možné představit jako sekvenci snímků, obvykle dvou nebo tří, které se mezi sebou cyklicky střídají. Z této sekvence snímků je vždy vybrán jeden konkrétní snímek, do kterého bude ukládán výsledek vykreslení. Ve stejné době je jiný snímek z této sekvence, do kterého byl již uložen výsledek vykreslování, zobrazen v okně aplikace. Po dokončení procesu vykreslení jsou dané snímky vyměněny a snímek, který byl právě zobrazen v okně aplikace, je vrácen zpět do fronty pro opětovné použití. Frekvence výměny jednotlivých snímků se může lišit na základě několika faktorů, jako je například obnovovací frekvence monitoru. [11]

3.5.5 Render pass

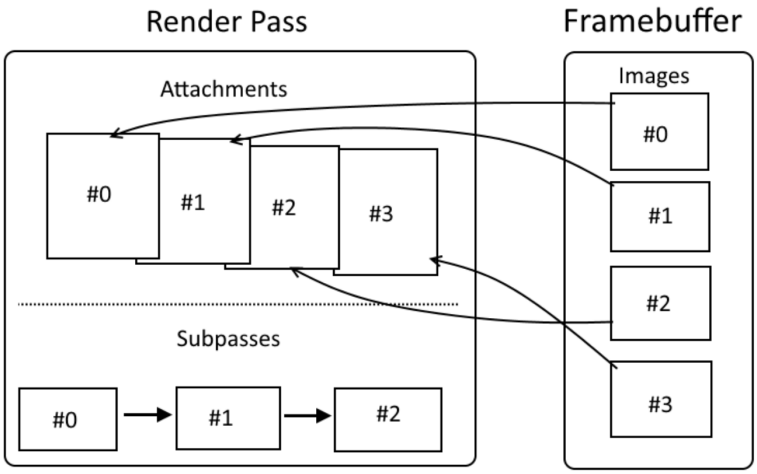
Render pass neboli vykreslovací průchod si je možné představit jako objekt, který slouží k přesnému definování i organizaci jednotlivých kroků v rámci vykreslovacího procesu. Render pass se skládá z jednoho a více podprůchodů (subpass), které rozdělují celý vykreslovací průchod na několik logicky rozdělených částí. Jednotlivé podprůchody mohou vykonávat zcela rozdílné operace, které průběžně přispívají k celkovému výsledku vykreslování. Díky tomu je možné relativně rychle a efektivně implementovat různé pokročilejší techniky, jako je například post processing nebo deferred shading. Kromě jednotlivých podprůchodů mohou být v rámci render pass definovány i přílohy (attachments), které slouží k určení konkrétních vlastností obrázků (images), do kterých budou ukládána příslušná data. Přílohy mohou být využity v rámci jednotlivých podprůchodů nebo mohou sloužit jako samotný výstup vykreslování. Mezi běžné typy příloh patří color attachments (slouží k ukládání barevných výstupů) nebo depth attachments (slouží k ukládání informací o hloubce scény). [1, 11]



Obr. 6: Diagram zobrazující vztahy mezi přílohami a podprůchody
Zdroj: Obrázek převzat z [11]

3.5.6 Framebuffer

Framebuffer je důležitý objekt, který reprezentuje kolekci nebo sadu obrázků (images), do kterých budou postupně ukládány výsledky, případně mezivýsledky vykreslování. Jednotlivé obrázky jsou za pomoci framebufferu propojeny s příslušnými přílohami, které jsou definovány v render pass. Vlastnosti i formát jednotlivých obrázků musí přímo korespondovat s vlastnostmi daných příloh. [11]



Obr. 7: Diagram zobrazující vztah mezi framebufferem a přílohami definovanými v render pass

Zdroj: Obrázek převzat z [11]

3.5.7 Command buffer

Command buffer představuje klíčový objekt, který slouží k uchovávání sekvence příkazů, které mají být simultánně odeslány ke zpracování příslušnému fyzickému zařízení (nejčastěji grafické kartě). Tento postup vede k výraznému snížení komunikační zátěže mezi procesorem a příslušným fyzickým zařízením, což vede ke zlepšení celkového výkonu dané aplikace. Jednotlivé command buffery jsou alokovány z command poolů, což jsou speciální typy objektů, které spravují určitou část paměti, která má být následně využita k efektivní alokaci jednotlivých command bufferů. [11]

4 Vybrané knihovny pro práci s API Vulkan

V současné době existuje široká škála více či méně populárních knihoven napříč různými programovacími jazyky, které umožňují vývojářům přímo či nepřímo využívat rozhraní Vulkan. Správný výběr knihovny a s tím spojený výběr programovacího jazyka může mít zásadní dopad na vytvářený projekt, jelikož se každá z těchto knihoven specializuje na trošku jinou oblast zájmu a nabízí zcela rozdílnou sadu funkcí a nástrojů. Kromě nich se od sebe jednotlivé knihovny odlišují také zcela rozdílnou syntaxí kódu, dokumentací, výkonem, případně úrovní abstrakce, kterou nabízejí.

Z tohoto důvodu je následující kapitola věnována krátkému představení vybraných knihoven, které jakýmkoliv způsobem umožnují využít rozhraní Vulkan a jsou nadále aktivně udržované a aktualizované. Jednotlivé knihovny jsou mezi sebou porovnány pomocí sady různých parametrů a tyto výsledky budou následně reflektovány v rámci praktické části této bakalářské práce. Hodnocení těchto knihoven je prováděno co nejobjektivnějším způsobem, avšak podobně jako jakékoliv jiné hodnocení je i toto hodnocení do určité míry ovlivněno osobními preferencemi.

4.1 LWJGL

LWJGL neboli Lightweight Java Game Library je multiplatformní open-source knihovna, která byla vyvinuta s cílem zjednodušit proces vývoje počítačových her a graficky náročných aplikací v programovacím jazyce Java. V současné době knihovna poskytuje přístup k několika různým nízkoúrovňovým rozhraním, které primárně slouží pro tvorbu počítačové grafiky, zvuku a paralelní zpracovávání dat. Mezi nejznámější rozhraní patří například Vulkan, OpenGL, GLFW a OpenAL. [15]

4.2 PyVulkan[1]

PyVulkan slouží jako wrapper (obálka) okolo grafického rozhraní Vulkan. Jeho primárním cílem je poskytnout vývojářům přístup k pokročilým grafickým funkcím při zachování jednoduché a intuitivní syntaxe programovacího jazyka Python. [16] Je důležité zmínit, že z hlediska výkonu není Python jako interpretovaný programovací jazyk ideální volbou pro tvorbu náročnějších aplikací a bylo by tedy vhodné využít jiný programovací jazyk.

[1] Oficiálním názvem této knihovny je Vulkan. Aby bylo možné jednoduše odlišit tuto knihovnu od samotného rozhraní Vulkan, bude v rámci tohoto textu knihovna označována jako PyVulkan.

4.3 Ash

Ash je open-source knihovna specificky určená pro programovací jazyk Rust, která slouží jako wrapper (obálka) okolo grafického rozhraní Vulkan. Stejně jako LWJGL je Ash navržen tak, aby vývojářům poskytoval co největší volnost v rámci implementace dané aplikace, ale zároveň poskytoval všechny funkce, které nabízí samotný Vulkan, a to bez jakékoliv zbytečné abstrakce. [17] Tento přístup umožnuje vývojářům využít všech výhod programovacího jazyka Rust, ale zároveň pracovat se syntaxí, která je do značné míry inspirována originálním rozhraním.

4.4 Vulkano

Knihovna Vulkano, specificky navržená pro programovací jazyk Rust, představuje moderní a bezpečný wrapper (obálku) okolo grafického rozhraní Vulkan. Knihovna byla vytvořena s cílem nabídnout rychlé, ale zároveň bezpečné a snadno použitelné nízkoúrovňové rozhraní pro vytváření graficky náročných aplikací a počítačových her. Od ostatních knihoven se odlišuje především svým důrazem na bezpečnost kódu a efektivní správu paměti. Knihovna chytře využívá kompilátor k automatické detekci a řešení širokého spektra potencionálních problémů, které mohou v průběhu vývoje aplikace vzniknout. K tomu jí napomáhá i obrovský počet pomocných metod a funkcí, jejichž cílem je usnadnit a urychlit vývoj aplikací. Příkladem může být automatická synchronizace příkazů, které mají být odesílány grafické kartě. [18]

4.5 WGPU

WGPU je moderní multiplatformní grafická knihovna, vytvořená pro programovací jazyk Rust, jejíž první verze byla vydána v roce 2019. [19] Knihovna je postavena na základě grafického rozhraní WebGPU a jejím cílem je poskytnout bezpečný a spolehlivý přístup k různým funkcím grafické karty. Aplikace napsané za pomoci této knihovny mohou nativně běžet na všech hlavních operačních systémem a vybraných grafických rozhraních. Knihovna poskytuje vývojářům jednotné rozhraní pro přístup k širokému spektru grafických funkcí bez ohledu na rozhraní, na kterém bude daná aplikace běžet. Toho je dosaženo za pomoci speciální překladové vrstvy, která automaticky překládá specifické příkazy do jednoho z následujících rozhraní: Vulkan, Metal, DirectX nebo OpenGL. Dodatečně mohou aplikace napsané skrze WGPU běžet nativně skrze webový prohlížeč za pomoci WebGL, nebo WebGPU. [20] Oproti jiným knihovnám, které primárně slouží jako wrapper (obálka) okolo rozhraní Vulkan, WGPU poskytuje

značnou úroveň abstrakce, která usnadňuje a zrychluje vývoj aplikací. Kromě vysoké míry abstrakce knihovna dále poskytuje i robustní systém pro zabezpečení správy paměti a automatickou kontrolu kódu, která je prováděna při kompilaci aplikace.

4.6 Silk.NET

Silk.NET je rozsáhlá, multiplatformní open-source knihovna, speciálně navržena pro programovací jazyk C#. Hlavním cílem této knihovny je poskytnout všechny možné nástroje, které vývojáři mohou potřebovat při vývoji multimediálních nebo graficky náročných aplikací a her. Z tohoto důvodu knihovna poskytuje jednoduchý a rychlý přístup k mnoha klíčovým nízkoúrovňovým rozhraním, a to nejen z oblasti počítačové grafiky a zvuku, ale také z oblasti paralelního zpracování dat a haptiky. Mezi nejpopulárnější rozhraní, které je možné v rámci knihovny využít lze zařadit například OpenGL, Vulkan DirectX, OpenCL a OpenAL. Kromě nich Silk.NET poskytuje řadu pomocných funkcí a nástrojů, jejichž primárním účelem je usnadnit a zrychlit vývoj multiplatformních aplikací. [21]

4.7 Veldrid

Veldrid je multiplatformní nízkoúrovňová grafická a výpočetní knihovna primárně navržená a optimalizovaná pro vývoj 2D i 3D počítačových her, simulací a dalších graficky náročných aplikací. Tato knihovna je implementována v programovacím jazyku C# a poskytuje efektivní a flexibilní přístup k většině populárních grafických rozhraní. Mezi podporovaná rozhraní, která lze v rámci knihovny využít, patří Direct3D, Metal, OpenGL a Vulkan. Díky tomu je možné aplikace vytvořené za pomoci této knihovny spustit na všech běžně používaných operačních systémech, aniž by bylo nutné provádět jakékoliv další konfigurace nebo úpravy kódu. [22]

Tabulka 1: Přehled základních informací o vybraných knihovnách, které umožnují využívat rozhraní Vulkan

	LWJGL	PyVulkan	Ash	Vulkano	WGPU	Silk.NET	Veldrid
Programovací jazyk	Java	Python	Rust	Rust	Rust	C#	C#
Podpora API Vulkan	✓	✓	✓	✓	✓	✓	✓
Podpora dalších API	✓	✗	✗	✗	✓	✓	✓
Multiplatformní	✓	✓	✓	✓	✓	✓	✓
Výkon	Vysoký	Nízký	Velmi vysoký	Velmi vysoký	Vysoký	Vysoký	Vysoký
Snadnost použití	Nízká	Střední	Nízká	Střední	Vysoká	Střední	Střední
Popularita	Vysoká	Nízká	Vysoká	Střední	Vysoká	Střední	Střední
Aktuálně vyvíjené	✓	✓	✓	✓	✓	✓	✓
Četnost aktualizací	Časté	Řídké	Občasné	Občasné	Časté	Časté	Řídké
Oficiální dokumentace	Dobrá	Špatná	Dobrá	Kvalitní	Kvalitní	Špatná	Dobrá
Výukové materiály	Několik	Málo	Několik	Několik	Mnoho	Mnoho	Málo
Vhodné pro začátečníky	✗	✗	✗	✓	✓	✓	✗

Zdroj: Vlastní zpracování na základě informací získaných z [15 – 22]

✓ - Knihovna úspěšně splňuje příslušný parametr

✗ - Knihovna nesplňuje příslušný parametr

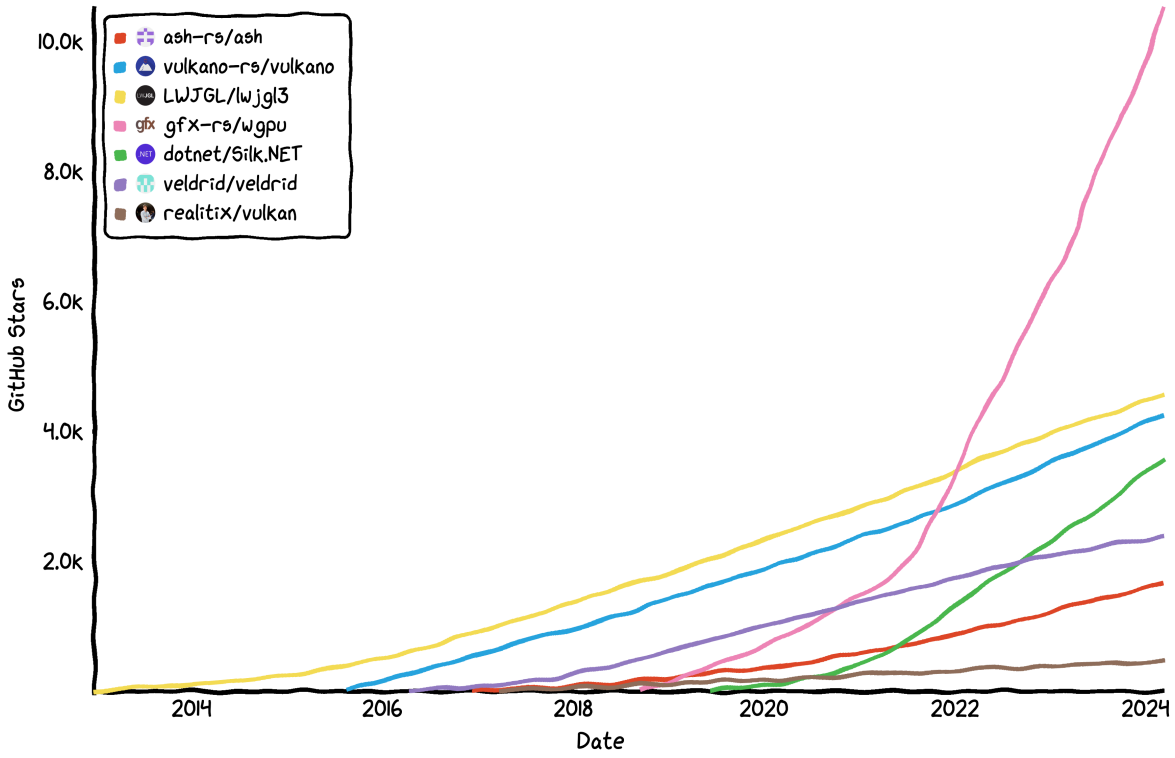
Následující tabulka (2) poskytuje základní přehled o aktuálním stavu vývoje všech dříve představených knihoven, a to na základě oficiálních informací získaných z platformy GitHub. Sloupec „Počet stažení“ reflektuje informace o celkovém počtu stažení jednotlivých knihoven (k datu 24. březen 2024), které byly získány z veřejných zdrojů. Pokud nejsou žádné informace o počtu stažení k dispozici, je toto pole označeno symbolem „-“.

Tabulka 2: Popularita a aktuální stav vývoje jednotlivých knihoven

Název knihovny	Množství hvězdiček	Aktuální verze	Datum poslední aktualizace	Počet stažení
LWJGL	4 497	3.3.3	16. září 2023	-
PyVulkan	454	1.3.275	18. ledna 2024	-
Ash	1 618	0.37.3	29. května 2023	5 073 746
Vulkano	4 296	0.34	25. října 2023	332 480
WGPU	10 109	0.19.1	21. ledna 2024	4 011 775
Silk.NET	3 466	2.20.0	12. prosince 2023	-
Veldrid	2 357	4.9.0	3. února 2023	-

Zdroj: Vlastní zpracování na základě dat z platforem GitHub [23] a Cargo [24]

Následující graf (obrázek 8) zobrazuje postupný vývoj popularity (množství hvězdiček) jednotlivých knihoven na platformě GitHub. Popularita daných knihoven je vždy zobrazena za období od data jejich vzniku až do 14. března 2024.



Obr. 8: Vývoj počtu hvězdiček jednotlivých knihoven v průběhu času
Zdroj: Zpracováno za pomoci nástroje: https://star-history.com/

Všechny zkoumané knihovny jsou k dispozici pod open-source licencemi, díky čemuž je mohou vývojáři zcela bezplatně využívat v rámci jakéhokoliv typu projektu. Níže uvedená tabulka (3) poskytuje základní přehled jednotlivých knihoven spolu s licencemi, pod kterými jsou oficiálně distribuovány.

Tabulka 3: Licence, pod kterými jsou jednotlivé knihovny distribuovány

Název knihovny	Licence
LWJGL	BSD 3 - Clause
PyVulkan	MIT
Ash	Apache 2.0, MIT
Vulkano	Apache 2.0, MIT
WGPU	Apache 2.0, MIT
Silk.NET	MIT/X11
Veldrid	MIT

Zdroj: Vlastní zpracování na základě informací získaných z [15 – 22]

5 Programovací jazyk Rust

Programovací jazyky lze rozdělit a charakterizovat na základě mnoha různých kritérií. Jedním ze způsobů, jakým můžeme tyto programovací jazyky rozdělit, je způsob, jakým přistupují ke správě paměti. Obecně existují dva hlavní přístupy, které využívá naprostá většina programovacích jazyků. Jazyky jako například C a C++ poskytují programátorům úplnou a detailní kontrolu nad tím, kdy a jakým způsobem má být paměť alokována a následně uvolněna. Tento přístup je obecně výhodný z hlediska výkonu a flexibility, ale na stranu druhou značně zvyšuje riziko nechtěného úniku paměti. Na opačném konci spektra stojí programovací jazyky, jako jsou například Java, C# nebo Javascript, které využívají automatickou správu paměti. Ta je ve většině případů realizována za pomoci garbage kolektoru, který se stará o automatické uvolňování paměti, která již nebude dále využívána. Tento přístup eliminuje riziko nechtěného úniku paměti, ale zároveň může výrazně ovlivnit celkový výkon dané aplikace. Rust představuje zcela unikátní alternativu, která kombinuje to nejlepší z předcházejících přístupů. Tím umožnuje eliminovat většinu problémů spojených se správou paměti, aniž by nad ní vývojáři ztratili úplnou kontrolu.

5.1 Představení jazyka

Rust je moderní, víceúčelový kompilovaný programovací jazyk představený společností Mozilla, který se od většiny programovacích jazyků odlišuje svým unikátním přístupem ke správě paměti. Tento přístup je založen na dvou základních principech vlastnictví a výpůjčky, které společně umožnují efektivní a bezpečnou správu paměti bez nutnosti spoléhat se na garbage kolektor nebo manuální správu paměti. Jednou z hlavních předností a výhod tohoto programovacího jazyka je jeho vysoký výkon, který je ve většině případů srovnatelný s dalšími nízkoúrovňovými programovacími jazyky, jako je C a C++. Dále Rust také nabízí výkonný kompilátor a mnoho různých pomocných funkcí či nástrojů pro tvorbu a implementaci vícevláknových nebo asynchronních aplikací. S tím je spojena i další výhoda tohoto programovacího jazyka, a to je volnost při výběru programovacího paradigmatu. Rust v základu podporuje většinu hlavních paradigmat, a to včetně funkcionálního, procedurálního a objektově orientovaného paradigmatu. Díky tomu je možné vytvářet široké spektrum různě důležitých aplikací, jako jsou například databázové a operační systémy, ovladače (např. grafických karet) nebo různé síťové služby, které musejí v reálném čase zpracovávat obrovské množství požadavků a dat. [25, 26]

Další nezanedbatelnou předností tohoto programovacího jazyka je jeho rozsáhlá a detailní dokumentace spolu s nezměrným množstvím různých knihoven, které jsou dostupné zdarma za pomoci integrovaného správce balíčků Cargo. Rust samotný proces tvorby dokumentace značně zjednodušuje, jelikož je dokumentace automaticky generována přímo z komentářů, které se nacházejí ve zdrojovém kódu dané aplikace nebo knihovny. [27]

5.2 Popularita

Popularita i využití programovacího jazyka Rust neustále narůstá, což dokazují i jeho nedávné úspěchy. Na konci roku 2022 dosáhl Rust významného milníku, když se stal teprve druhým programovacím jazykem po jazyku C, který je možné využít v rámci jádra operačního systému Linux. [28] V roce 2023 na tento trend navázala i firma Microsoft, která oznámila záměr přepsat určité části operačního systému Windows do programovacího jazyka Rust. [29] Rostoucí popularitu toho jazyka dále potvrzují i pravidelné průzkumy, pořádané platformou Stack Overflow, které se zaměřují na zkušenosti programátorů s různými programovacími jazyky. Do posledního průzkumu z roku 2023 se zapojilo více než 85 000 vývojářů z celého světa. Výsledky tohoto průzkumu ukázaly, že Rust v minulém roce využilo alespoň jednou přibližně 13 % vývojářů, z nichž téměř 85 % vyjádřilo zájem využít tento jazyk i pro budoucí projekty. Toto číslo bylo nejvyšší ze všech zkoumaných programovacích jazyků. [30] Výše zmíněné výsledky potvrzují stále rostoucí popularitu tohoto programovacího jazyka, který má potenciál stát se po Javě a Pythonu dalším široce využívaným programovacím jazykem.

5.3 Systém vlastnictví a výpůjčky

Prvním klíčovým konceptem je unikátní systém vlastnictví (v anglickém jazyce ownership), který definuje základní soubor pravidel, která musejí být splněna pro efektivní a bezpečnou správu paměti. Hlavním cílem celého systému je za pomoci kompilátoru eliminovat většinu běžných problémů, které jsou spojené se správou paměti, a to ještě před samotným spuštěním daného programu v rámci procesu kompilace. Celý systém je založen na základním pravidle, které říká, že každá hodnota v rámci programu musí mít definovaného jednoho jediného vlastníka, který za ni přebírá plnou zodpovědnost. Tento přístup umožnuje předcházet širokému spektru potencionálních chyb, a to již při kompilaci programu. Celý koncept vlastnictví je

postaven na několika jednoduchých pravidlech, která musejí být za všech okolností dodržena: [27]

1. Každá hodnota musí mít právě jednoho jednoznačně určeného vlastníka.
2. Vlastník hodnoty musí být vždy znám.
3. Vlastnictví hodnoty může být přeneseno na nového vlastníka.
4. Hodnota může být vypůjčena neomezenému počtu vlastníků.
5. Hodnotu může vypůjčit pouze vlastník této hodnoty.
6. Pokud vlastník hodnoty přestane existovat, musí být hodnota i s ní spojené prostředky automaticky uvolněny z paměti.
7. Vlastník hodnoty je zodpovědný za uvolnění paměti v případě, že danou hodnotu již nebude potřebovat.

Dalším konceptem, který úzce souvisí se systémem vlastnictví, je systém výpůjčky (v anglickém jazyce borrowing). Tento systém slouží k zajištění bezpečného a efektivního sdílení dat v rámci programu. Rust umožnuje vytvářet a využívat reference neboli odkazy, na určité hodnoty. Tyto reference následně umožnují sdílet hodnoty mezi různými částmi programu, například mezi několika funkcemi, bez nutnosti změny vlastnictví nebo zkopírování dané hodnoty. V principu existují dva typy referencí, a to defaultní immutable nebo mutable. Immutable reference umožnuje, aby byla daná hodnota přečtena, a to bez možnosti ji následně prostřednictvím této reference nějakým způsobem modifikovat. Oproti tomu mutable reference umožnuje, aby mohla být daná hodnota prostřednictvím této reference přečtena i upravena. [27]

Posledním klíčovým konceptem, který spolupracuje se systémem vlastnictví a výpůjček je „lifetime“, neboli doba života. Lifetime definuje časový interval, během něhož může existovat reference na určitou hodnotu. Cílem je zajistit, aby žádná z referencí neodkazovala na hodnotu, která již byla odstraněna z paměti. Celý koncept je postaven na základě jednoduchého pravidla, které říká, že platnost reference nikdy nesmí překročit dobu života hodnoty, na kterou se odkazuje. Jinými slovy reference může existovat pouze, dokud existuje samotná hodnota, na kterou odkazuje. Lifetime je ve většině případů určen automaticky při kompilaci programu, pouze ve specifických situacích je nutné jej explicitně specifikovat za pomoci speciální anotace. [27]

6 Návrh a implementace ukázkových úloh

Detailně a pečlivě zpracovaný návrh představuje klíčový předpoklad pro úspěšný vývoj a nasazení jakékoliv softwarové aplikace. Tento krok vyžaduje nejen důkladnou analýzu a následné porozumění jednotlivým potřebám a požadavkům koncových uživatelů, ale také pochopení prostředí, v němž bude daná aplikace nasazena.

Cílem této bakalářské práce je prozkoumat možnosti a potenciál grafického rozhraní Vulkan jako nástroje pro výuku počítačové grafiky. S tímto cílem se pojí nutnost prozkoumat různé aspekty práce s tímto grafickým rozhraním, počínaje samotnou instalací všech potřebných nástrojů a vývojových prostředí a konče konkrétním využitím tohoto rozhraní pro vytváření aplikací. V rámci této bakalářské práce bylo navrženo a následně i implementováno celkem dvacet jedna ukázkových úloh, které pokrývají široké spektrum různě náročných konceptů, primárně z oblasti počítačové grafiky. Základem pro návrh jednotlivých ukázkových úloh se stala studie [31] Johannese Unterguggenbergera z Technologické Univerzity ve Vídni. Tato studie se zabývá praktickým využitím rozhraní Vulkan jako nástroje pro výuku počítačové grafiky, jakožto modernější alternativy k tradičně používanému rozhraní OpenGL. V rámci studie měli všichni studenti za úkol navrhnout a implementovat pět různě náročných a komplexních aplikací, které se věnovaly základním konceptům, jako je implementace vykreslovací smyčky, práce s buffery, mapování textur nebo výpočet osvětlení za pomoci Phongova osvětlovacího modelu. Do studie bylo zapojeno celkem 123 studentů, z nichž 17 (14 %) si vybralo Vulkan a zbylých 106 (86 %) se rozhodlo pro OpenGL. Celkem tuto studii i samotný kurz dokončilo 81 studentů, z nichž si 71 vybralo OpenGL a zbylých 10 Vulkan. Po absolvování samotného kurzu měli jednotliví studenti možnost dobrovolně vyplnit anonymní dotazník, který zahrnoval řadu různých otázek zaměřených na hodnocení jejich spokojenosti a osobní zkušenosti s daným rozhraním. Dotazník vyplnilo celkem 61 studentů, z nichž 9 pracovalo s rozhraním Vulkan a 52 s OpenGL. Analýza výsledků studie odhalila, že samotný výběr rozhraní měl na celkovou úspešnost studentů jen minimální dopad, přičemž studenti, bez ohledu na výběr rozhraní, museli vynaložit na implementaci jednotlivých ukázkových úloh přibližně stejné množství času. Výsledky studie mohly být částečně zkresleny nízkým množství studentů, případně i tím, že nejčastěji uváděným důvodem pro výběr rozhraní OpenGL byla snaha zjednodušit si práci. Za největší problém, bez ohledu na vybrané rozhraní, označila většina studentů programovací jazyk C/C++.

Výběr vhodné knihovny představuje důležitý krok, který ovlivní celý průběh vývoje jednotlivých ukázkových úloh. Aby bylo možné provést informované rozhodnutí, bylo provedeno porovnání a následné zhodnocení různých parametrů, viz tabulka číslo 1. Při výběru knihovny byl kladen největší důraz na následující parametry, které jsou seřazeny dle jejich relativní (subjektivní) důležitosti:

1. **Dostupnost a přístupnost**
   Kritérium se soustředí na jednoduchost instalace, komplexnost knihovny, popularitu daného programovacího jazyka, případně na různá licenční omezení.

2. **Dokumentace a učební zdroje**
   Kritérium se zaměřuje na porovnání kvality, dostupnosti a množství dokumentace, ukázek, tutoriálů, případně jiných výukových (podpůrných) materiálů.

3. **Uživatelská přívětivost**
   Kritérium slouží k posouzení obtížnosti práce s danou knihovnou a programovacím jazykem, včetně dostupnosti vývojových prostředí.

4. **Flexibilita**
   Kritérium slouží k posouzení, do jaké míry lze danou knihovnu přizpůsobit konkrétním požadavkům a potřebám daného projektu – podpora pokročilejších funkcí, podpora pro různé operační systémy atd.

5. **Výkon**
   Kritérium slouží ke zhodnocení, jak dobře je daná knihovna a programovací jazyk optimalizován pro tvorbu náročnějších aplikací z oblasti počítačové grafiky.

Po pečlivé analýze a následném porovnání jednotlivých knihoven, s ohledem na specifické potřeby a požadavky, byla pro implementaci ukázkových úloh zvolena knihovna Vulkano spolu s programovacím jazykem Rust. Knihovna Vulkano nabízí vhodný kompromis mezi příliš vysokou komplexností, která je spojena s přímým využitím rozhraní Vulkan, a příliš vysokou úrovní abstrakce, jakou nabízí například knihovna WGPU. Vzhledem k vysokému výkonu, relativně jednoduché syntaxi a rostoucí popularitě programovacího jazyka Rust se tato volba jeví jako více než ideální. Samotné ukázkové úlohy byly navrženy takovým způsobem, aby pokrývaly široké spektrum různě složitých konceptů a technik, primárně z oblasti počítačové grafiky. Seznam všech ukázkových úloh je dostupný jako příloha číslo 1.

Následující tabulka (4) obsahuje seznam všech implementovaných ukázkových úloh spolu se seznamem základních konceptů, na které je v rámci dané ukázky kladen největší důraz.

Tabulka 4: Základní seznam všech implementovaných ukázkových úloh

Název ukázky	**Pokryté koncepty**
p00_window	Základy vytváření a manipulace s okny, práce s knihovnou Winit
p01_basic_triangle	Implementace základní vykreslovací smyčky, tvorba a načtení fragment a vertex shaderů
p02_uniform_buffer	Práce s uniform buffery a push constants, zasílání dat do shaderů, základy práce s knihovnou cgmath
p03_geometry	Výpočet viditelnosti za pomoci Z-bufferu, implementace index bufferu, definování různých 3D objektů (solidů)
p04_textures	Načítání a mapování textur na různé objekty (solidy)
p05_camera	Implementace kamery
p06_obj	Načítání 3D modelů uložených ve formátu OBJ
p07a_loop_rendering, p07b_instancing, p07c_offset_rendering	Implementace různých technik pro vykreslování více objektů (solidů) najednou
p08_light	Implementace Phongova osvětlovacího modelu a práce s materiály
p09_texture_array	Základy práce s polem textur, mapování textur na více objektů najednou
p10a_compute_shader	Implementace základní compute pipeline, provádění výpočtů na grafické kartě
p10b_compute_shader	Použití compute shaderů v kombinaci s vykreslovacím řetězcem (implementace částicového systému)
p11_geometry_shader	Úvod do práce s geometry shadery
p12_multiple_shaders	Načítání a použití shaderů uložených ve formátu SPIR-V, vykreslení objektů za pomoci několika fragment shaderů
p13_post_processing	Post-processing, základy techniky vykreslování do textury, základy víceprůchodového zpracování
p14a_tessellation, p14b_tessellation	Úvod do práce s teselačními shadery, použití teselačních shaderů pro zvýšení detailů geometrie
p15a_deferred_shading, p15b_deferred_shading	Rozdílné způsoby implementace techniky deferred shading, víceprůchodové zpracování

Zdroj: vlastní zpracování

6.1 Inicializace projektu

6.1.1 Instalace VulkanSDK

Pro zjednodušení vývoje i samotného ladění aplikací využívajících rozhraní Vulkan je vhodné nainstalovat VulkanSDK (Vulkan Software Development Kit), což je komplexní a rozsáhlá sada různých nástrojů a knihoven, které primárně slouží ke zjednodušení práce s tímto rozhraním. VulkanSDK je možné bezplatně stáhnout z oficiálních stránek na adrese: https://vulkan.lunarg.com/home/welcome.

6.1.2 Instalace MoltenVK

Aplikace využívající rozhraní Vulkan nejsou sami o sobě kompatibilní s operačními systémy macOS, iOS a tvOS od společnosti Apple. Z tohoto důvodu je nezbytné na těchto operačních systémech nainstalovat MoltenVK. MoltenVK funguje jako překladová vrstva, která automaticky překládá jednotlivé příkazy a shadery tak, aby jim rozumělo rozhraní Metal, používané na novějších zařízeních od společnosti Apple. MoltenVK je automaticky nainstalován při instalaci VulkanSDK, nebo je možné jej dodatečně zdarma stáhnout a následně i nainstalovat z oficiálního GitHub repositáře na adrese: https://github.com/KhronosGroup/MoltenVK.

6.1.3 Instalace programovacího jazyka Rust

Všechny ukázkové úlohy byly vytvořeny v programovacím jazyce Rust. Pro vývoj a následnou kompilaci aplikací, napsaných v tomto programovacím jazyce, je nezbytné mít nainstalovaný kompilátor zdrojového kódu spolu se správcem balíčků Cargo. Tyto nástroje jsou zdarma dostupné pro operační systémy Windows, Linux a macOS a je možné je stáhnout z oficiálních stránek na adrese: https://www.rust-lang.org/.

6.1.4 Vývojové prostředí

Výběr vhodného vývojového prostředí představuje důležitý krok, který má zásadní vliv na efektivitu, pohodlí a celkovou zkušenost vývojáře s daným programovacím jazykem. Z tohoto důvodu je doporučováno využít jedno z několika oficiálně doporučovaných vývojových prostředí, které podporují programovací jazyk Rust.

První doporučovanou možností je instalace zcela nového vývojového prostředí RustRover od společnosti JetBrains. Prostředí RustRover bylo specificky navrženo pro programovací jazyk Rust a poskytuje širokou paletu nástrojů, které by měly usnadnit

samotný vývoj aplikací. RustRover je dostupný na všech hlavních operačních systémech, a to konkrétně Windows, Linux a macOS. Pro vývojáře, kteří dávají přednost bezplatnému řešení, představuje Visual Studio Code (VS Code) od společnosti Microsoft ideální volbu. VS Code je vysoce univerzální vývojové prostředí, které podporuje širokou škálu programovacích jazyků, a to buď přímo, nebo za pomoci volně dostupných rozšíření. Samotný programovací jazyk Rust je dostupný za pomoci bezplatného rozšíření rust-analyzer, které poskytuje podobné funkce a nástroje, které lze nalézt v rámci vývojového prostředí RustRover.

RustRover i VC Code poskytují vynikající podporu pro vývoj, testování a ladění aplikací, a to ať už jde o malé projekty, nebo o rozsáhlejší systémy. Konečná volba prostředí by tedy měla reflektovat osobní preference daného vývojáře, jeho rozpočet a konkrétní požadavky na dané vývojové prostředí.

6.1.5 Vytvoření projektu

Pro vygenerování základní struktury jednotlivých ukázkových úloh byl použit příkaz `cargo new project_name`, který je možné využít v případě, kdy je na daném zařízení nainstalován správce balíčků Cargo. Základní struktura projektu se vždy skládá ze dvou klíčových souborů:

Cargo.toml

Tento soubor tvoří základní stavební kámen každého projektu v jazyce Rust. Primárně funguje jako konfigurační soubor, ve kterém lze definovat širokou škálu různých informací, které se přímo týkají dané aplikace.

Adresář src

V tomto adresáři se nachází hlavní zdrojový kód dané aplikace, přičemž obvykle tento adresář obsahuje soubor main.rs (aplikace) respektive lib.rs (knihovna).

6.1.6 Použité knihovny

Vulkano a vulkano-shaders

Knihovna Vulkano spolu s doplňkovou knihovnou vulkano-shader tvoří kompletní soubor nástrojů pro jednoduchý vývoj multiplatformních aplikací běžících na rozhraní Vulkan. Pomocná knihovna vulkano-shaders je navržena s cílem co nejvíce zjednodušit práci s různými typy shaderů. Tato knihovna umožnuje vývojářům jednoduše načítat shadery napsané v jazyce GLSL, a to bez nutnosti je předem zkompilovat do binárního

formátu SPIR-V. Tohoto cíle bylo dosaženo za pomoci kompilátoru, který automaticky kompiluje jednotlivé shadery do formátu SPIR-V a následně je automaticky integruje do spustitelného souboru aplikace.

Winit

Winit je multiplatformní knihovna navržená pro jednoduchou a efektivní práci s okny aplikací. Knihovna umožnuje jednoduše vytvářet i spravovat okna aplikací bez nutnosti upravovat zdrojový kód pro jednotlivé operační systémy. Winit nativně podporuje většinu hlavních operačních systémů včetně Windows, Linux, macOS, Android a iOS.

Cgmath

Cgmath je matematická knihovna speciálně navržená a optimalizovaná pro tvorbu počítačové grafiky a počítačových her. Knihovna obsahuje širokou sadu různých matematických struktur a nástrojů, včetně vektorů, bodů nebo různých typů matic.

Image

Knihovna image poskytuje komplexní sadu nástrojů pro snadné načítání, ukládání a manipulaci s obrázky, které jsou uloženy v jednom v mnoha běžně používaných formátů. Jmenovitě se jedná například o formáty PNG, JPG či WEBP.

Png

Png je specializovaná knihovna primárně určená pro zjednodušení práce s PNG obrázky. Tato knihovna poskytuje nástroje pro zakódování a následné dekódování jednotlivých obrázků stejně jako nástroje pro manipulaci s nimi.

```
// Struktura souboru Cargo.toml spolu s použitými závislostmi
[package]
name = "p02_uniform_buffer"
version = "0.1.0"
edition = "2021"

[dependencies]
vulkano = "0.34.1"
vulkano-shaders = "0.34.0"

winit = "0.28.7"
cgmath = "0.18.0"
image = "0.24.8"

vulkano_text_renderer = { path = "../libs/vulkano_text_renderer" }

// Nastavení optimalizace zdrojového kódu při kompilaci programu
[profile.dev]
opt-level = 1

[profile.release]
opt-level = 3
lto = true
codegen-units = 1
```

6.2 Implementace základní vykreslovací smyčky

Implementace základní vykreslovací smyčky v rámci rozhraní Vulkan i samotné knihovny Vulkano představuje relativně komplexní a náročný proces, který se skládá z mnoha dílčích částí a vyžaduje napsání stovek řádků kódu. Následující část textu je věnována obecnému popisu implementace jednotlivých struktur (objektů), které musejí být implementovány pro úspěšné sestavení funkční vykreslovací smyčky. Je nutné podotknout, že konkrétní implementace se může napříč jednotlivými ukázkami více či méně lišit v závislosti na konkrétních potřebách a požadavcích dané ukázky.

6.2.1 Okno aplikace

Před tím, než bude možné zobrazit jakýkoliv obsah na obrazovce, je nutné za pomoci knihovny Winit vytvořit nové okno, které nám to bude umožnovat. Samotný proces vytvoření okna je možné rozdělit do tří základních kroků. Prvním krokem je vytvoření zcela nové event loop, kterou si je možné představit jako donekonečna běžící smyčku, která zpracovává různé systémové a uživatelské události. Pro každou aplikaci je možné vytvořit pouze jeden event loop, který musí běžet na hlavním vláknu dané aplikace. Druhým krokem je samotné vytvoření nového okna aplikace za pomoci struktury WindowBuilder. Tato struktura umožnuje vytvářet libovolný počet oken, přičemž každé z těchto oken může být libovolně upraveno za pomoci metod, kterými lze nastavit různé parametry, jako je například pozice, název nebo ikonka. Z takto vytvořeného okna lze následně za pomoci metody from_windows získat surface (povrch), díky kterému bude možné zobrazit příslušný výsledek vykreslování do okna aplikace. Zpracování jednotlivých událostí je možné řídit za pomoci speciální metody run (doporučovaný způsob) nebo run_return (omezená funkčnost, podporuje jen omezený počet operačních systémů, je možné, že se objeví nečekané chyby).

```
// Vytvoření nového okna o rozměrech 800x600
let window_size: LogicalSize<f64> = LogicalSize::new(800.0, 600.0);
let window: Arc<Window> = Arc::new(WindowBuilder::new().with_position(position)
                                                       .build(&event_loop)
                                                       .unwrap());

// Nastavení parametrů okna – set_window_icon nebude spuštěn na zařízeních s operačním systémem macOS
#[cfg(not(target_os = "macos"))]
window.set_window_icon(Some(load_icon("resources/icons/rust-project-icon.png")));
window.set_title("Vulkano Examples | P02 - Uniform buffer | Dominik Prokop | 2024");
```

6.2.2 Instance

Po úspěšném vytvoření a konfiguraci okna aplikace je možné vytvořit zcela novou instanci, která bude sloužit jako komunikační kanál mezi danou aplikací a knihovnou

Vulkan. Aby bylo možné tuto instanci vytvořit, je nejdříve nezbytně nutné inicializovat knihovnu Vulkan. To lze provést za pomoci struktury VulkanLibrary, která se na daném zařízení automaticky pokusí vyhledat a následně načíst příslušnou knihovnu. Jakmile je knihovna Vulkan úspěšně načtena, je množné přistoupit k samotnému vytvoření nové instance. Tento proces zahrnuje specifikaci několika různých parametrů, jež je možné nastavit manuálně, automaticky načíst z konfiguračního souboru TOML nebo se spolehnout na přednastavené (defaultní) hodnoty poskytované knihovnou Vulkano.

```
// Vytvoření nové instance s příslušnými parametry
let required_extensions: InstanceExtensions = get_required_extensions(&event_loop);
let create_info: InstanceCreateInfo = InstanceCreateInfo {
        enabled_extensions: required_extensions,
        flags: InstanceCreateFlags::ENUMERATE_PORTABILITY,
        application_name: Some(application_name.to_string()),
        ..Default::default()
};

let instance: Arc<Instance> = Instance::new(library, create_info).expect("Unable to create instance.“);
```

6.2.3 Fyzické a logické zařízení

Po vytvoření instance přichází na řadu výběr vhodného fyzického zařízení, na kterém budou prováděny příslušné příkazy. Výběr vhodného fyzického zařízení představuje náročný úkol, který ovlivní celkovou funkčnost a rychlost dané aplikace. Každé zařízení (mobil, počítač, notebook) obsahuje určitý počet fyzických zařízení (grafická karta, procesor s integrovanou grafickou kartou,…), jejichž seznam je možné získat za pomoci metody enumerate_physical_devices, kterou lze nalézt na příslušné instanci. Konkrétní způsob výběru vhodného fyzického zařízení se může značně lišit na základě konkrétních požadavků a potřeb dané aplikace. Konkrétně lze výběr vhodného fyzického zařízení provést manuálně nebo za pomoci filtru, který na základě předem daných parametrů automaticky vybere vhodné fyzické zařízení. Filtr by měl ideálně filtrovat jednotlivá zařízení na základě podpory požadovaných rozšíření a na základě dalších volitelných parametrů, které jsou relevantní pro danou aplikaci.

Po úspěšném výběru vhodného fyzického zařízení je následně možné za pomoci struktury Device vytvořit zcela nové korespondující logické zařízení. Při vytváření logického zařízení je nutné specifikovat konkrétní rozšíření, která mají být automaticky aktivována na daném fyzickém zařízení. Pokud vybrané fyzické zařízení nepodporuje určité rozšíření, nebude možné spustit danou aplikaci a do konzole bude vypsána příslušná chybová hláška.

6.2.4 Swap chain

Po vytvoření logického zařízení přichází na řadu vytvoření a konfigurace swap chainu, a to na základě přesně stanovených požadavků dané aplikace. Při konfiguraci swap chainu je důležité správně specifikovat všechny požadované vstupní parametry, jako je například rozměr, využití a formát snímků či minimální počet snímků, které mají být automaticky vytvořeny. Kdykoliv, kdy dojde ke změně velikosti okna aplikace, nebo v případě, kdy se stávající swap chain stane nevhodným pro další použití, je vhodné zavolat na daném swap chainu pomocnou metodu recreate. Tato metoda znovu vytvoří daný swap chain, a tím eliminuje různé typy potencionálních problémů, které mohou nastat.

```
// Znovu vytvoření swap chainu s jedním poupraveným parametrem image_extent
let create_info: SwapchainCreateInfo = SwapchainCreateInfo {
        image_extent: dimensions.into(),
        ..swap_chain.create_info()
};

let swap_chain: (Arc<Swapchain>, Vec<Arc<Image>>) = swap_chain.recreate(create_info)
                                                              .expect("Failed to recreate swap chain.");
```

6.2.5 Render pass, Frame buffer

Na základě specifických požadavků a potřeb dané aplikace je nyní možné vytvořit nový render pass. Pro vytvoření základního jednoprůchodového render pass je možné využít speciální makro single_pass_renderpass!, zatímco pro vytvoření složitějšího víceprůchodového render pass je možné využít makro ordered_passes_renderpass! Skrze tato makra lze kromě samotných podprůchodů definovat i libovolný počet příloh. Ty musejí mít definovány příslušné parametry, jako je například formát, zda mají být dané hodnoty ukládány, případně zda mají být automaticky vymazány. Po úspěšném vytvoření render pass je možné vytvořit korespondující frame buffer, jenž musí obsahovat identické množství a specifikace snímků (images), které odpovídají jednotlivým přílohám definovaných v render pass.

```
// Vytvoření nového jednoprůchodového renderpass, který obsahuje jednu přílohu s názvem color
let render_pass: Arc<RenderPass> = single_pass_renderpass!(
        logical_device.clone(),
        attachments: {
                color: {
                        format: swap_chain.image_format(),
                        samples: 1,
                        load_op: Clear,
                        store_op: Store,
                },
        },
        pass: {
                color: [color],
                depth_stencil: {},
        },
)
.expect("Failed to create render pass. Ensure that the device and swap chain are properly set up.");
```

6.2.6 Načítání shaderů

Aby bylo možné využít všech výhod, které přináší knihovna Vulkano, je vhodné využít speciální pomocnou knihovna vulkano-shaders. Tato knihovna nabízí sadu pomocných nástrojů a metod specificky navržených pro zjednodušení práce se shadery. Jedním z nástrojů, které tato knihovna poskytuje, je makro vulkano_shaders::shaders!, které umožnuje efektivní a bezpečné načítání GLSL shaderů. Při kompilaci programu shadery načtené za pomoci tohoto makra procházejí automatickou validací a následně jsou zkompilovány do formátu SPIR-V a připojeny ke spustitelnému souboru dané aplikace.

```
// Definice modulů, které budou sloužit jako rozhraní pro práci s vybranými shadery
pub mod vs {
        vulkano_shaders::shader! {
                ty: "vertex",
                path: "src/shaders/vertex_shader.glsl",
        }
}

pub mod fs {
        vulkano_shaders::shader! {
                ty: "fragment",
                path: "src/shaders/fragment_shader.glsl",
        }
}

// Načtení a následná kompilace (GLSL → SPIR-V) shaderů za pomoci makra vulkano_shaders::shaders!
let vertex_shader: EntryPoint = vs::load(logical_device.clone()).unwrap().entry_point("main").unwrap();
let fragment_shader: EntryPoint = fs::load(logical_device.clone()).unwrap().entry_point("main").unwrap();
```

6.2.7 Grafická pipeline

Po úspěšném načtení všech potřebných shaderů je možné přejít k samotnému sestavení nové grafické pipeline, která bude zodpovědná za samotné vykreslování určité scény. Tento krok je možné realizovat skrze strukturu GraphicsPipeline, která jako vstupní parametr přijímá seznam konfiguračních parametrů, na základě kterých vytvoří a následně vrátí zcela novou grafickou pipeline. Skrze parametry je možné nakonfigurovat jednotlivé programovatelné i fixní části zobrazovacího řetězce. Konkrétně lze skrze parametry nakonfigurovat například rasterizaci, blending barev, multisampling, topologii, případně testování viditelnosti. Kromě parametrů je také nutné připojit uspořádaný seznam shaderů, které mají být v rámci dané grafické pipeline využity. Je důležité zmínit, že ačkoliv lze určité parametry upravit i po samotném vytvoření grafické pipeline, hlavní struktura a většina parametrů již nemůže být dále upravena.

6.2.8 Deskriptory

V případě, kdy aplikace potřebuje přístup k různým datům uložených v bufferech, je nutné vytvořit novou sadu deskriptorů. Zjednodušeně řečeno, každý buffer, ke kterému se bude přistupovat v rámci shaderů, musí mít vlastní deskriptor.

```
// Vytvoření zcela nové sady deskriptorů pro připojení uniform bufferu
let uniform_set_layout: &Arc<DescriptorSetLayout> = pipeline.layout().set_layouts().get(0).unwrap();

let uniform_set: Arc<PersistentDescriptorSet<StandardDescriptorSetAlloc>> = PersistentDescriptorSet::new(
        descriptor_set_allocator,
        uniform_set_layout.clone(),
        [
            WriteDescriptorSet::buffer(0, uniform_sub_buffer)
        ],
        [],
)
.expect("Unable to create descriptor set.");
```

6.2.9 Command buffer

Pro samotné vykreslení geometrie je nutné navrhnout a následně sestavit command buffer, který bude obsahovat sekvenci za sebou jdoucích příkazů, které mají být odeslány vybranému fyzickému zařízení k provedení. K tomuto účelu je možné využít speciální strukturu AutoCommandBufferBuilder, která umožnuje za pomoci různých metod sestavit požadovanou sekvenci příkazů a z nich následně vytvořit nový command buffer.

Ve většině případů tato sekvence začíná zavoláním metody begin_render_pass, která slouží pro zahájení specifického render pass. Následně jsou za sebou volány metody k postupnému připojení všech požadovaných objektů, jako je grafická pipeline, vertex a index buffer, deskriptory, případně push constants. Pro samotné vykreslení geometrie je možné zavolat jednu ze tří specifických metod. Nejzákladnější metoda draw slouží k vykreslení geometrie na základě předem definované topologie a jednotlivých vrcholů uložených ve vertex bufferu. V případě, kdy je definován index buffer, je pro správné vykreslení geometrie nutné použít metodu draw_indexes, která vykresluje jednotlivá primitiva na základě indexů uložený v index bufferu. Třetí a nejméně používaná metoda draw_indirect funguje podobným způsobem jako metoda draw, ale parametry jsou uloženy v bufferu, který lze následně libovolně modifikovat přímo na vybraném fyzickém zařízení. Po přidání konkrétní metody, která slouží k vykreslení geometrie, je možné zahájit další podprůchod nebo ukončit render pass. Po ukončení daného render pass je možné za pomoci metody build sestavit daný command buffer. Pokud během procesu sestavovaní command bufferu dojde k jakékoli

chybě, je tato chyba automaticky zobrazena v rámci konzole a následně dochází k ukončení dané aplikace.

Samotný command buffer je následně možné odeslat příslušnému fyzickému zařízení, na kterém jsou postupně vykonány jednotlivé příkazy. Po úspěšném ukončení vykreslování dané scény (geometrie) je možné zobrazit daný výsledek do okna aplikace. Nyní již stačí pouze za pomoci metody cleanup_finished provést potřebný úklid a uvolnění již nepotřebných systémových zdrojů.

```
// Sekvence příkazů z nichž je složen command buffer
builder
        .begin_render_pass(render_pass_begin_info, subpass_begin_info)
        .expect("Failed to initiate the rendering process with the provided render pass info.")
        .set_viewport(0, [viewport.clone()].into_iter().collect())
        .expect("Failed to define the area of the frame buffer to output to.")
        .bind_pipeline_graphics(pipeline.clone())
        .expect("Failed to attach the graphics pipeline to the command builder.")
        .bind_descriptor_sets(
            PipelineBindPoint::Graphics,
            pipeline.layout().clone(),
            0,
            uniform_set.clone(),
        )
        .expect("Failed to attach the descriptor set to the pipeline.")
        .push_constants(pipeline.layout().clone(), 0, time_constant)
        .expect("Failed to push time constant.")
        .bind_vertex_buffers(0, vertex_buffer.clone())
        .expect("Failed to attach the vertex buffer to the pipeline.")
        .draw(vertex_buffer.len() as u32, 1, 0, 0)
        .expect("Failed during the execution of the draw command.")
        .end_render_pass(Default::default())
        .expect("Failed to conclude the rendering pass.");

// Sestavení command bufferu
let command_buffer: Arc<PrimaryAutoCommandBuffer> = builder.build()
                                                            .expect("Failed to build command buffer.");
```

6.3 Knihovna vulkano_text_renderer

Pro zjednodušení a zefektivnění práce s textovými elementy byla navržena a následně i implementována speciální pomocná knihovna vulkano_text_renderer. Knihovna nabízí sadu různých struktur a metod, které umožnují nejen rychlé a jednoduché přidávání textových elementů, ale také jejich následnou modifikaci. Tyto elementy lze pomocí sady několika různých parametrů volně přizpůsobit potřebám dané aplikace. Skrze parametry lze nastavit font, barvu jednotlivých textových elementů (RGBA), jejich pozici a optimální velikost, případně vybrat jedno z několika typů pozadí. Každý textový element může využít jedno ze tří typů podporovaných pozadí: žádné, jednobarevné (RGB) nebo poloprůhledné (RGBA). Samotná knihovna je postavena na základě speciální pomocné knihovny rusttype, která nabízí různé nástroje pro načtení a zpracování jednotlivých fontů definovaných ve formátech TFF (TrueType Font) nebo OTF (OpenType Font).

Aby bylo možné použít knihovnu vulkano_text_renderer, je nejdříve nutné ji přidat jako závislost do souboru Cargo.toml. Po přidání knihovny bude možné vytvořit zcela novou instanci struktury TextUI, která je primárně zodpovědná za uchovávání a následné vykreslení textových elementů. Pro definování neomezeného počtu jednotlivých textových elementů je možné využít metodu define_elements. Textové elementy jsou za pomoci knihovny rusttype automaticky uloženy do textury, která je automaticky uložena v paměti daného fyzického zařízení. Po samotném zpracování jednotlivých textových elementů metoda define_elements automaticky zavolá pomocné vnitřní metody, které slouží k vytvoření a následnému naplnění vertex a index bufferu. Vertex buffer bude uchovávat odpovídající počet vrcholů skládajících se ze dvou základních parametrů, pozice a souřadnic do textury. Pro samotné vykreslení textu je nezbytně nutné na vybraném AutoCommandBufferBuilder zavolat metodu draw_text s příslušnými parametry, která zajistí, aby byly vykresleny jednotlivé vrcholy tvořící trojúhelníky, na které bude namapována příslušná část textury.

```
// Ukázka definování nového textového elementu za pomoci pomocné knihovny vulkano_text_renderer
let mut text_ui: TextUI = TextUI::new(&logical_device, &swap_chain, &images, &allocator, Fonts::Roboto);
text_ui.define_elements(|ui| {
	ui.add_text(
		"shortcuts",
		"[Q/E/W/S/A/D] Rotate, [O/P/K/L] Scale [R]eset Matrices, [H]ide text",
		TextCreateOptions {
			background: Background::SolidColor([0.0, 0.0, 0.0]),
			..Default::default()
		},
	);
	ui.add_text(
		"movement",
		"[Arrow Up/Down/Left/Right/Numpad +/-] Move",
		TextCreateOptions {
			position: [5.0, 35.0],
			color: [0.0, 0.0, 0.0, 1.0],
			background: Background::SolidColor([0.0, 0.0, 0.0]),
			..Default::default()
		},
	);

});
```

Po vytvoření textových elementů za pomoci metody define_elements je možné jednotlivé elementy dále libovolně upravovat za pomoci pomocné metody update_text. Díky této metodě je tedy možné upravit libovolné množství parametrů určitého textového elementu. Konkrétně lze upravit barvu, pozici, samotný text a pozadí.

```
// Ukázka modifikace textového elementu s id „shortcuts“
VirtualKeyCode::Z => {
	text_ui.update_text(„shortcuts“, TextParams::new().size(20.0)
                                                      .position([5.0, 5.0])
                                                      .color([1.0, 0.0, 0.0, 1.0])
                                                      .text(„This is a new text!“)
                                                      .background(Background::Transparent);
}
```

6.4 Dokumentace

Pro účely pochopení základních principů práce s knihovnou Vulkano byla v rámci této bakalářské práce vytvořena rozsáhlá dokumentace. Tato dokumentace byla vytvořena za pomoci specializovaného nástroje Writerside od společnosti JetBrains, který umožnuje jednoduše a rychle vytvářet rozsáhlou dokumentaci, za pomoci jazyků Markdown a XML. Vytvořená dokumentace je primárně rozdělena do tří hlavních segmentů a slouží jako komplexní studijní (výukový) materiál, který spolu s ukázkovými úlohami umožní jednotlivým čtenářům prozkoumat a osvojit si různé koncepty a techniky potřebné pro práci s knihovnou Vulkano, případně s programovacím jazykem Rust. První segment dokumentace je věnován instalaci všech potřebných nástrojů a vývojových prostředí určených pro vývoj aplikací v programovacím jazyce Rust (instalace je vždy popsána pro vybrané operační systémy: Windows, Linux a macOS). Druhý segment je věnován krátkému představení jednotlivých ukázkových úloh včetně krátkého popisu způsobu, jakým lze tyto ukázky ovládat. Třetí a nejrozsáhlejší segment je rozdělen do několika menších částí, které jsou věnovány postupnému představení základních konceptů, se kterými je možné se setkat v rámci knihovny Vulkano. Tento segment je konkrétně zaměřen na vysvětlení implementace tří vybraných základních témat: vykreslení trojúhelníku, definování 3D solidů a mapování textur. Každé z těchto témat je obohaceno o rozsáhlou sbírku okomentovaných příkladů zdrojového kódu, které by měly spolu s doprovodným textem postupně rozšiřovat a prohlubovat znalosti jednotlivých čtenářů. Strukturu dokumentace je možné najít jako přílohu číslo 3, zatímco obrázky, které zobrazují vybrané stránky dokumentace, je možné najít jako přílohu číslo 4.

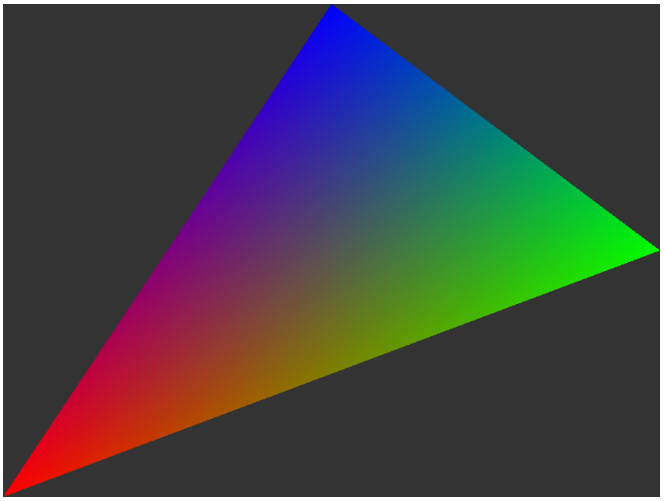
6.5 Představení jednotlivých ukázkových úloh

p00_window

Ukázková úloha slouží k ilustraci a představení základních principů práce s knihovnou Winit. Jejím hlavním účelem je poskytnout základní přehled o procesu vytvoření a základní konfigurace okna aplikace. Ukázka je rovněž doplněna o názorné představení hlavního způsobu, jakým lze zpracovávat různé typy událostí od kliknutí tlačítka myši až po stisk určité klávesy.

p01_basic_triangle

Ukázková úloha je věnována implementaci všech funkcí, struktur a komponent, které jsou nezbytné pro úspěšné vykreslení jednoduchého trojúhelníku do okna aplikace. Konkrétně se jedná například o implementaci instance, fyzického a logického zařízení, shaderů nebo grafické pipeline. Tato ukázka také slouží jako základ, na který lze navázat a postupně přidávat nové funkce.



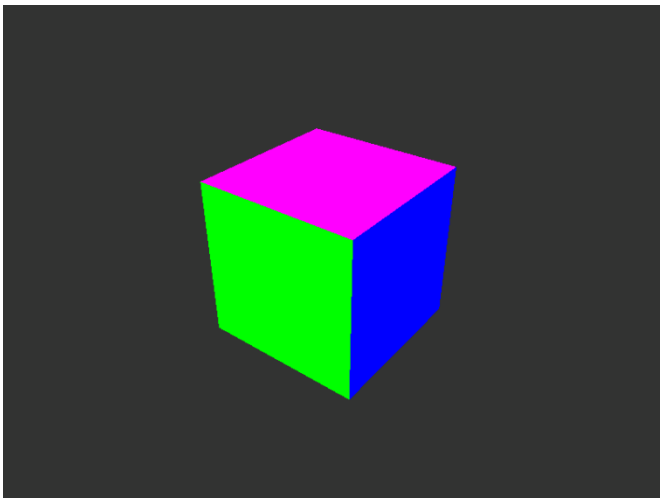
Obr. 9: Ukázka p01_basic_triangle – Vykreslení 2D trojúhelníku
Zdroj: Vlastní zpracování

p02_uniform_buffer

Ukázková úloha se primárně soustředí na představení základní práce s buffery. Hlavním cílem této ukázky je ukázat, jakým způsobem lze efektivně využít buffery pro uchovávání a snadný přístup k datům. V případě této ukázky se jedná o transformační matice. Zároveň se také úloha soustřední na využití a implementaci push constants, které umožnují jednoduché a přímé odesílání dynamicky se měnících dat do shaderů.

p03_geometry

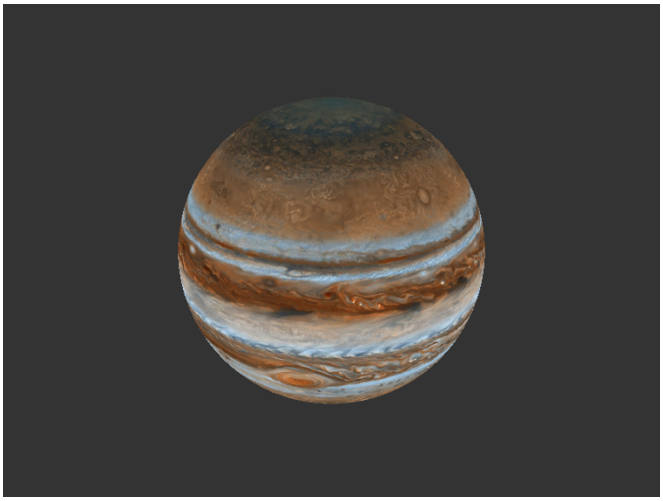
Ukázková úloha se zaměřuje na řešení viditelnosti objektů v rámci 3D scény s využitím z-bufferu. V rámci samotné ukázky je implementováno šest různých geometrických tvarů (solidů), mezi kterými se lze volně přepínat. Konkrétně se jedná o krychli (cube), osmistěn (octahedron), UV sféru (uv_sphere), ICO sféru (ico_sphere), torus a pyramidu (pyramid).



Obr. 10: Ukázka p03_geometry – Vykreslení 3D objektů do okna aplikace
Zdroj: Vlastní zpracování

p04_textures

Ukázková úloha slouží jako úvod do základů práce a manipulace s texturami. V rámci ukázky jsou na jednotlivé tvary (solidy), které již byly definovány v předcházející ukázce, postupně namapovány příslušné 2D textury.



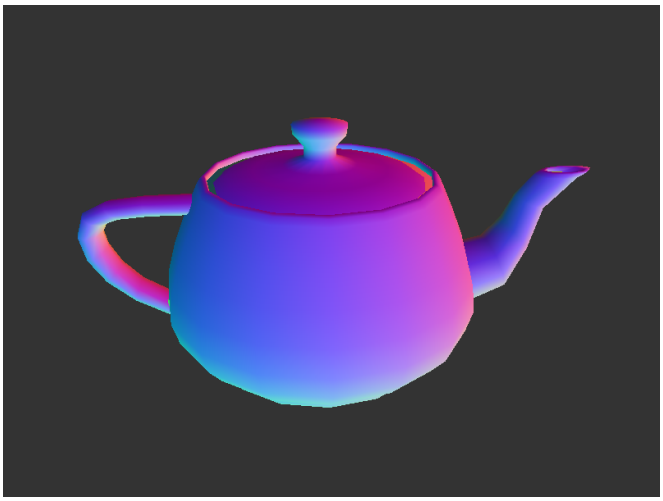
Obr. 11: Ukázka p04_textures – Mapování textur na různé tvary
Zdroj: Vlastní zpracování

p05_camera

Ukázková úloha navazuje na předcházející ukázky a je specificky věnována návrhu a následné implementaci jednoduché perspektivní kamery. Kamera je specificky navržena tak, aby umožnovala jednoduché ovládání jakékoliv aplikace prostřednictvím klávesnice a myši. Pro efektivní a rychlou práci s maticemi, vektory a body je v rámci ukázky využívána pomocná matematická knihovna cgmath.

p06_obj

Ukázková úloha se zaměřuje na implementaci procesu načítání a následného zpracování různých 3D modelů, které jsou uloženy ve formátu OBJ. Pro zpracování jednotlivých modelů byla vytvořena speciální pomocná funkce, která umožnuje již při samotném načtení jednotlivých modelů provádět nezbytné transformace (například otočit daný model), změnit pořadí vrcholů (wind order), případně upravit velikost celého modelu. Tímto způsobem byl odstraněn problém, kdy byly jednotlivé modely definovány v jiných souřadnicových systémech, než standardně používá rozhraní Vulkan. Pro načítání jednotlivých OBJ souborů byla využita pomocná knihovna tobj.

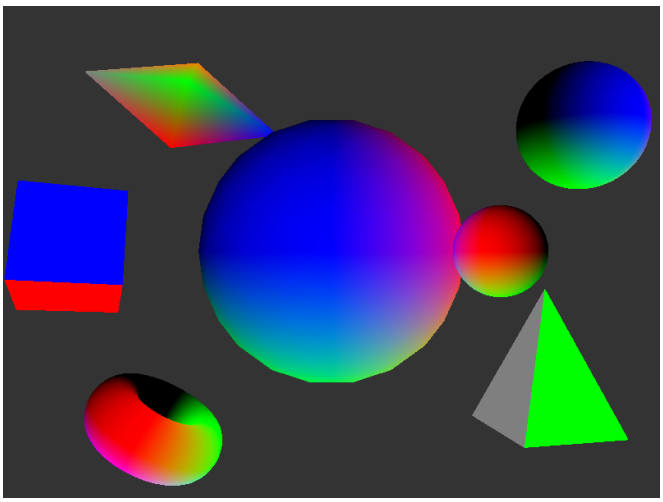


Obr. 12: Ukázka p06_obj – Čajová konvice načtená z formátu OBJ
Zdroj: Vlastní zpracování

p07a_loop_rendering, p07b_instancing a p07c_offset_rendering

Následující ukázkové úlohy se zaměřují na implementaci různých metod pro simultánní vykreslování více objektů (solidů). Konkrétně se tyto úlohy věnují třem specifickým přístupům. První přístup spočívá ve vykreslení jednotlivých objektů prostřednictvím jednoduchého cyklu, který pro každý definovaný objekt přidá do command bufferu nový příkaz k vykreslení daného objektu. Druhý přístup využívá

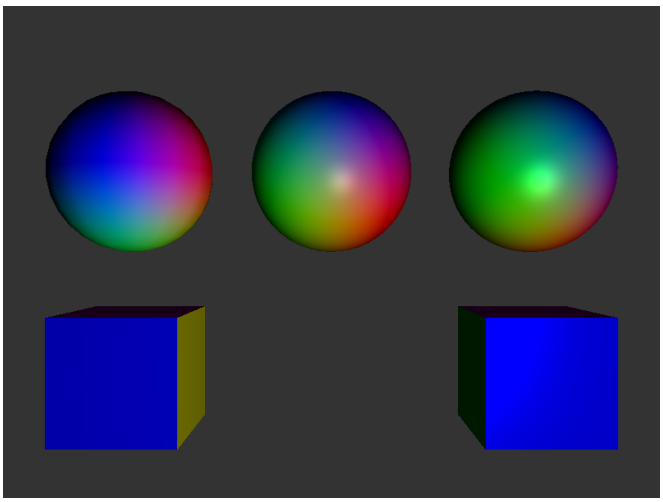
instancing, což je speciální technika, která je primárně určena k efektivnímu vykreslení mnoha podobných instancí téhož objektu. Třetí a poslední přístup spočívá ve využití jednoho vertex a index buffer, ve kterých jsou uloženy všechny potřebné informace o objektech v rámci scény. Jednotlivé objekty jsou následně simultánně vykresleny za pomoci offsetů.



Obr. 13: Ukázka p07c_offset_rendering – Simultánní vykreslení objektů
Zdroj: Vlastní zpracování

p08_light

Ukázková úloha je věnována implementaci Phongova osvětlovacího modelu, který je v rámci ukázky využit pro simulaci reálného osvětlení v zadané scéně. V této úloze jsou specifickým objektům přiřazeny různé typy materiálů, které ovlivňují způsob, jakým světlo působí na jednotlivé povrchy. Celkové chování materiálu může být ovlivněno zvlášť pro každou složku: ambientní, difúzní i spektakulární.



Obr. 14: Ukázka p08_light – Implementace Phongova osvětlovacího modelu
Zdroj: Vlastní zpracování

p09_texture_array

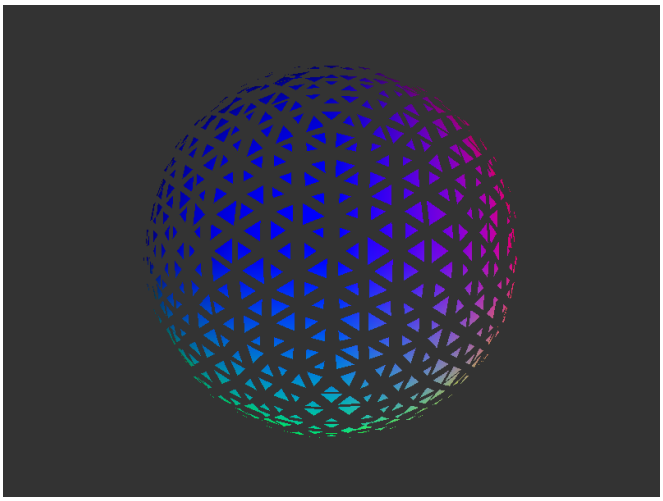
Ukázková úloha se zaměřuje na základní načítání několika textur současně a jejich následné uložení do pole textur. Takto načtené textury jsou následně namapovány na jednotlivé objekty v rámci scény, kdy je příslušná textura vybrána na základě definovaného indexu (čísla dané textury).

p10a_compute_shader a p10b_compute_shader

Následující ukázkové úlohy se soustředí na praktickou implementaci compute shaderů. První z těchto úloh je zaměřena na naprosté základy práce s compute shadery, kdy je vygenerováno několik zcela náhodných čísel, která jsou následně zpracována prostřednictvím compute shaderu. Druhá z těchto úloh ilustruje jednoho z praktických (možných) využití compute shaderů, a to konkrétně implementaci jednoduchého částicového systému.

p11_geometry_shader

Ukázková úloha demonstruje implementaci geometry shaderu, jenž v průběhu času transformuje polohu jednotlivých vrcholů, z nichž je tvořena sféra (koule). Tato ukázka není dostupná na počítačích, které využívají operační systém macOS, a to z důvodu technických omezení, která spočívají v absenci podpory geometry shaderu. Pokud dojde ke spuštění této ukázky na operačním systému macOS, je tato aplikace automaticky ukončena a následně je zobrazena příslušná chybová hláška informující uživatele.



Obr. 15: Ukázka p11_geometry_shader – Transformace sféry za pomoci geometry shaderu
Zdroj: Vlastní zpracování

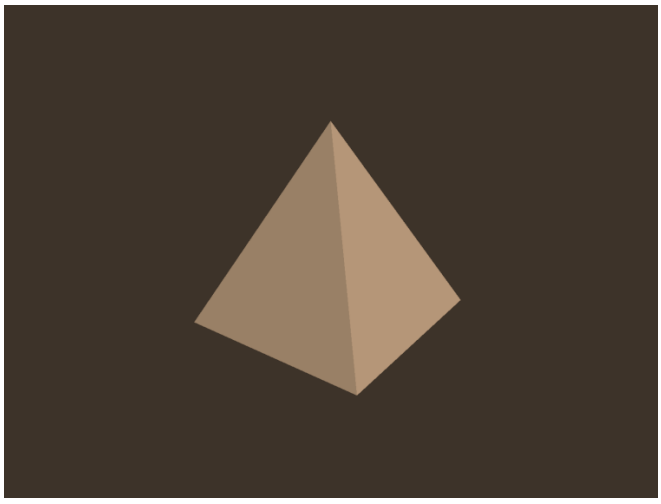
p12_multiple_shaders

Ukázková úloha je věnována práci a manipulaci s několika již předem zkompilovanými shadery ve formátu SPIR-V. V rámci této ukázky je definována jednoduchá scéna, která je složena ze tří trojúhelníků, na něž jsou postupně aplikovány mírně odlišné shadery, které byly již předem zkompilovány do formátu SPIR-V.

p13_post_processing

Ukázková úloha se věnuje implementaci několika post processing efektů pomocí techniky nazývané vykreslování do textury. V rámci této ukázky jsou implementovány tři odlišné druhy efektů:

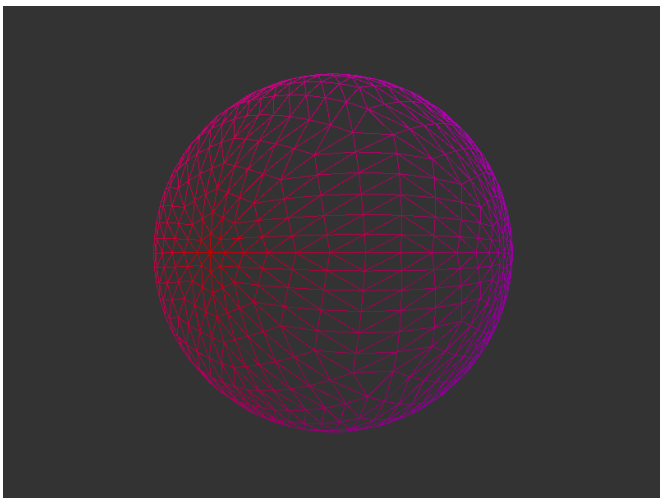
1. Retro efekt „staré obrazovky“, který obrazu dodává nostalgický vzhled díky přidání charakteristických horizontálních čar, které imitují vzhled starých CRT monitorů.
2. Efekt barevného filtru, který mění celkový barevný tón obrazu.
3. Efekt zvýraznění hrany objektů, jenž využívá hranový detektor ke zvýraznění a zobrazení jednotlivých kontur.



Obr. 16: Ukázka p13_post_processing – Efekt barevného filtru
Zdroj: Vlastní zpracování

p14a_tessellation a p14b_tessellation

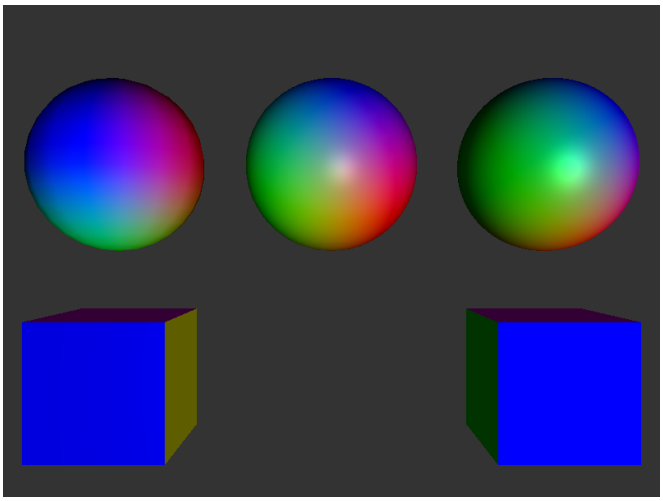
Následující ukázky jsou zaměřeny na implementaci a praktické využití teselace. První z těchto ukázek demonstruje základní principy práce s teselačními shadery, zatímco druhá ukázka je věnována jednomu z možných praktických využití teselace, konkrétně je teselace použita k transformování osmistěnu na sféru (koule).



Obr. 17: Ukázka p14b_tesselation – Sféra vzniklá za pomoci teselace
Zdroj: Vlastní zpracování

p15a_deferred_shading a p15b_deferred_shading

Následující ukázkové úlohy se soustředí na představení dvou rozdílných přístupů k implementaci techniky zvané deferred shading. Deferred shading je technika, která umožnuje efektivní práci s velkým množstvím světelných zdrojů. V rámci ukázek dochází nejdříve k vykreslení samotné geometrie a shromáždění potřebných informací, které jsou následně uloženy do bufferů. Tyto informace jsou v dalším kroku využity k vypočítání finálního osvětlení scény. Jednotlivé ukázky tedy demonstrují způsob, jakým lze za pomoci techniky deferred shading na různé typy objektů aplikovat potencionálně neomezený počet světelných zdrojů – konkrétně ambientního osvětlení, bodového osvětlení a směrového osvětlení.



Obr. 18: Ukázka p15a_deferred_shading – Výpočet osvětlení s využitím techniky deferred shading
Zdroj: Vlastní zpracování

7 Testování ukázkových úloh

Funkčnost všech ukázkových úloh byla detailně otestována napříč třemi hlavními operačními systémy, a to konkrétně Windows, macOS a Linux. Samotné testování probíhalo na následujících zařízeních:

Stolní počítač

- **OS:** Windows 11 Home 64-bit
- **Nainstalovaná verze VulkanSDK:** 1.3.275
- **Procesor:** Ryzen 5 3600
- **Grafická karta:** RTX 3070
- **Rozlišení monitoru:** 3440x1440
- **Operační paměť:** 32 GB RAM
- **Disk:** 512 GB M.2 SSD, 2000 GB HDD, 2000 GB SSD

Lenovo Yoga 7

- **OS:** Windows 11 Home 64-bit
- **Nainstalovaná verze VulkanSDK:** 1.3.275
- **Procesor:** Intel Core i7
- **Grafická karta:** Intel Iris Xe Graphics
- **Rozlišení monitoru:** 1920x1080
- **Operační paměť:** 16 GB RAM
- **Disk:** 1 000 GB SSD

Apple MacBook Pro 16“ 2023

- **OS:** macOS Sonoma 14.3.1
- **Nainstalovaná verze VulkanSDK:** 1.3.275
- **Procesor:** 12 jádrové CPU
- **Grafická karta:** 19 jádrové GPU
- **Rozlišení monitoru:** 3456x2234
- **Operační paměť (Unified memory):** 16 GB
- **Disk:** 512 GB SSD

Některé z ukázkových úloh nemusejí být plně funkční na všech operačních systémech. Následující tabulka (5) obsahuje kompletní seznam ukázkových úloh, který je doplněn o informace týkající se jejich kompatibility s daným operačním systémem.

Tabulka 5: Funkčnost ukázkových úloh na vybraných operačních systémech

Název ukázky	Windows	macOS	Linux
p00_window	✓	✓	✓
p01_basic_triangle	✓	✓	✓
p02_uniform_buffer	✓	✓	✓
p03_geometry	✓	~	✓
p04_textures	✓	~	✓
p05_camera	✓	~	✓
p06_obj	✓	~	✓
p07a_loop_rendering	✓	~	✓
p07b_instancing	✓	✓	✓
p07c_offset_rendering	✓	~	✓
p08_light	✓	~	✓
p09_texture_array	✓	~	✓
p10a_compute_shader	✓	✓	✓
p10a_compute_shader	✓	✓	✓
p11_geometry_shader	✓	✗	✓
p12_multiple_shaders	✓	~	✓
p13_post_processing	✓	~	✓
p14a_tessellation	✓	~	✓
p14b_tessellation	✓	~	✓
p15a_deferred_shading	✓	~	✓
p15b_deferred_shading	✓	~	✓

Zdroj: Vlastní zpracování

✓ - Ukázka je na dané platformě plně funkční

~ - Některé funkce nemusejí být k dispozici nebo mohou fungovat jiným způsobem

✗ - Ukázka není na dané platformě dostupná a není ji možné spustit

Z podrobné analýzy všech dostupných informací je patrné, že funkčnost jednotlivých ukázkových úloh závisí nejen na operačním systému, ale i na samotném hardwaru. Jednotlivé ukázkové úlohy byly během vývoje podrobeny testování, které primárně probíhalo na zařízeních s operačním systémem Windows. V důsledku toho je možné konstatovat, že jsou všechny ukázkové úlohy plně kompatibilní s tímto operačním systémem. Je důležité zdůraznit, že pro bezproblémové spuštění jednotlivých ukázkových úloh je nezbytně nutné, aby zařízení, na kterém jsou spouštěny tyto ukázky, disponovalo hardwarem, obvykle grafickou kartou, který podporuje nejnovější verzi rozhraní Vulkan.

Při testování jednotlivých ukázkových úloh na operačním systému macOS bylo zjištěno, že ačkoliv byla většina ukázkových úloh spustitelná, některé funkce se chovaly neočekávaným způsobem, nebo dokonce způsobovaly samotný pád dané aplikace. Pro řešení tohoto problému bylo klíčové postupně identifikovat všechny problematické části kódu a následně najít vhodný způsob, jakým by šlo tento problém vyřešit. Bohužel většina těchto problémů přímo vyplývala z omezení na straně samotného hardwaru i softwaru a nešlo je tedy přímo opravit. Z tohoto důvodu byly do zdrojového kódu přidány speciální podmínky, které zajišťují, aby byly dané funkce při kompilaci daného programu (pouze pokud je aplikace kompilována pro operační systém macOS) zcela eliminovány. Tímto způsobem byly odstraněny všechny problémové části kódu za cenu, že některé z funkcí nemusejí být na operačním systému macOS k dispozici. Primárně se jedná o vedlejší funkce, jako je například přepínání způsobu, jakým mají být vykreslovány jednotlivé objekty – zda mají být vykreslovány pouze body, čáry nebo celé vyplněné plochy. Z celkového počtu jedna dvaceti ukázkových úloh se na operačním systému macOS nepodařilo zprovoznit pouze jednu jedinou ukázkovou úlohu, a to konkrétně p11_geometry_shader. Tato ukázka bohužel nemohla být zprovozněna, protože geometry shader není žádným způsobem podporován.

Pokud jde o kompatibilitu jednotlivých ukázkových úloh s operačním systémem Linux, je situace značně podobná jako v případě operačního systému Windows. Pro spuštění jednotlivých ukázek je důležité, aby dané zařízení disponovalo dostatečně výkonným hardwarem, který podporuje nejnovější verzi rozhraní Vulkan a zároveň aby byly na daném zařízení nainstalovány všechny nezbytné ovladače.

Kromě ukázkových úloh byla také otestována i funkčnost pomocné knihovny vulkano_text_renderer. Výsledky ukázaly, že tato knihovna funguje bezproblémově napříč vybranými operačními systémy. Aby bylo možné posoudit teoretický dopad využití této knihovny na celkový výkon aplikací, byl vytvořen speciální soubor (sada) testů. Tento soubor se skládá z celkem deseti různých testovacích úloh, které jsou označeny identifikačními kódy T001 až T010. Pro zajištění co nejpřesnějších a nejobjektivnějších výsledků měření byly jednotlivé hodnoty FPS (snímky za sekundu) určeny jako průměr pěti na sobě nezávislých měření. Jednotlivá měření byla prováděna vždy pro každou testovanou úlohu samostatně a každé měření trvalo přesně 2 minuty. Jednotlivé testovací úlohy se mezi sebou lišily pouze dvěma základními parametry, a to množstvím textu (žádný text, jedna věta, sto vět, tisíc vět) a pozadím, které je vykresleno pod daným textem (bez pozadí, s pozadím, s poloprůhledným pozadím). Úvodní testovací úloha, T001, byla stanovena jako základní referenční bod, vůči kterému jsou srovnávány výsledky ostatních testů. V následující tabulce (6) jsou uvedeny konkrétní výsledky měření spolu s relativním procentuálním výkonem jednotlivých testovacích úloh v porovnání s úlohou T001.

Přesnost jednotlivých měření se pohybuje přibližně v rozmezí **± 1,0 %.**

Tabulka 6: Porovnání různých scénářů využití knihovny vulkano_text_renderer

ID Testu	Množství textu	Pozadí textu	FPS	%
T001	**Žádný text**	**-**	**2 684**	**100 %**
T002	Krátký text	Bez pozadí	2 584	96,27%
T003	Střední text	Bez pozadí	2 554	95,16%
T004	Dlouhý text	Bez pozadí	2 590	96,50%
T005	Krátký text	Jednobarevné pozadí	2 538	94,56%
T006	Střední text	Jednobarevné pozadí	2 568	95,68%
T007	Dlouhý text	Jednobarevné pozadí	2 593	96,61%
T008	Krátký text	Částečně průhledné pozadí	2 610	97,24%
T009	Střední text	Částečně průhledné pozadí	2 586	96,35%
T010	Dlouhý text	Částečně průhledné pozadí	2 571	95,79%

Zdroj: Vlastní zpracování

Žádný text – testovací úloha neobsahuje žádný text
Krátký text – testovací úloha obsahuje jednu větu (přibližně 20 písmen)
Střední text – testovací úloha obsahuje sto vět (přibližně 2 000 písmen)
Dlouhý text – testovací úloha obsahuje tisíc vět (přibližně 20 000 písmen)

Z analýzy výsledků lze vyvodit několik zajímavých závěrů o teoretickém dopadu této knihovny na celkový výkon aplikací (měřeno ve snímcích za sekundu). Jako referenční bod slouží testovací úloha T001 (označená tučně), u které byla naměřena průměrná hodnota 2 684 FPS, která reprezentuje 100 %.

Zbývající testovací úlohy je možné primárně rozdělit do tří hlavních skupin podle typu pozadí textu. První skupina, která zahrnuje úlohy T002 až T004, se vyznačuje tím, že jsou jednotlivé textové elementy vykresleny zcela bez jakéhokoliv pozadí. Nejvyšší hodnota byla zaznamenána u testovací úlohy T004, zatímco nejnižší hodnota u testovací úlohy T005. Samotný rozdíl ve výkonu mezi těmito úlohami činí 36 FPS. Druhá skupina zahrnuje úlohy T005 až T007. V rámci těchto úloh jsou jednotlivé textové elementy vykresleny s jednobarevným a zcela neprůhledným pozadím. V této kategorii byla nejvyšší hodnota zaznamenána u testovací úlohy T007 a nejnižší hodnota u testovací úlohy T005. Rozdíl mezi těmito úlohami činí 55 FPS. Třetí skupina zahrnuje zbývající testovací úlohy tedy T008 až T010. Jednotlivé textové elementy jsou v rámci této části vykresleny s jednobarevným a částečně průhledným pozadím. V této kategorii byla nejvyšší hodnota zaznamenána u testovací úlohy T008, zatímco nejnižší hodnota u testovací úlohy T010. Rozdíl ve výkonu v rámci této kategorie činí 39 FPS.

Jednotlivé testy přinášejí do určité míry neočekávané výsledky, jelikož množství vykreslovaného textu, ani samotný typ pozadí, nemají přímý (úměrný) vliv na celkový výkon aplikace. Z výsledků jednotlivých testů je patrné, že klíčovým faktorem, který ovlivňuje celkový výkon aplikace, a to přibližně o 100 FPS, je samotné využití knihovny vulkano_text_renderer. Tento pokles výkonu lze připsat způsobu, jakým je realizováno samotné vykreslování textu, kdy je v rámci prvního průchodu vykreslena samotná scéna a následně za pomoci dalšího průchodu daný text. Celkový dopad na výkon aplikace má tedy charakter logaritmické křivky, kdy je možné zpočátku spatřit znatelný dopad, ale následné přidávání textu nevede k dalšímu proporcionálnímu snižování výkonu.

8 Shrnutí a diskuse výsledků

Při návrhu i následné implementaci jakéhokoli typu aplikace je klíčové vybrat vhodné nástroje a technologie, které co nejpřesněji odpovídají specifickým požadavkům a potřebám daného projektu. Po důkladném zhodnocení všech přínosů i potencionálních omezení byla pro implementaci jednotlivých ukázkových úloh zvolena knihovna Vulkano spolu s programovacím jazykem Rust. V průběhu vývoje se tato kombinace projevila jako relativně všestranně výhodné řešení, a to především díky vysoké flexibilitě, relativně jednoduché syntaxi a ideální míře abstrakce. Kromě toho samotná knihovna Vulkano značně zjednodušuje mnoho různých aspektů, které přímo vyplývají z nízkoúrovňové podstaty rozhraní Vulkan, zejména v oblastech, jako je správa paměti, validace kódu, synchronizace či práce se shadery. Dále tato knihovna nabízí podporu pro většinu dnes používaných typů shaderů, různorodé typy bufferů, víceprůchodové zpracování a většinu funkcí, které nabízí samotné rozhraní Vulkan. Z tohoto důvodu se tato knihovna jeví jako vhodná volba pro ty vývojáře, kteří již mají alespoň základní zkušenosti s programováním a chtějí relativně jednoduše a rychle pochopit základní principy práce s rozhraním Vulkan. V kombinaci s rychle rostoucí popularitou programovacího jazyka Rust se tato kombinace jeví také jako obzvláště perspektivní i pro samotnou výuku počítačové grafiky.

Implementace jednotlivých ukázkových úloh odhalila také několik výzev, které je nezbytné určitým způsobem překonat, pokud se má tato knihovna v budoucnu využívat v rámci výuky počítačové grafiky. Jednou z hlavních výzev, kterou je nutné překonat, je nutnost napsat relativně velké množství kódu, aby bylo možné vykreslit jakýkoliv objekt do okna aplikace. V tomto ohledu sice poskytuje knihovna Vulkano významné zjednodušení ve srovnání s přímým využitím rozhraní Vulkan, přesto je ale stále nezbytné napsat řádově stovky řádků kódu. Za další potencionální problém lze považovat fakt, že ne všechna dostupná rozšíření a funkce, které jsou poskytovány rozhraním Vulkan, je možné v rámci knihovny Vulkano využít. Podpora jednotlivých funkcí i rozšíření je sice vývojáři průběžně aktivně rozšiřována, přesto se stále najdou některé funkce, které nejsou k dispozici. Další relativně výrazný problém byl odhalen v rámci implementace a testování jednotlivých ukázkových úloh, kdy bylo potřeba vyřešit podporu rozhraní Vulkan na zařízeních s operačním systémem macOS. Pro spuštění aplikací využívajících rozhraní Vulkan je na tomto operačním systému nutné nainstalovat pomocný nástroj MoltenVK. MoltenVK slouží jako speciální překladová

vrstva, která automaticky překládá jednotlivé příkazy i shadery. To může způsobit snížení celkového výkonu dané aplikace, nebo nemusejí být některé funkce k dispozici (tento fakt může způsobit i samotný pád dané aplikace). Jednu z ukázkových úloh, konkrétně ukázkovou úlohu p11_geometry_shader, nebylo možné z tohoto důvodu na operačním systému macOS zprovoznit. Většina ostatních ukázkových úloh musela být pro tento operační systém speciálně upravena tak, aby nedocházelo k nečekaným chybám. To má za následek, že některé funkce nejsou na tomto operačním systému k dispozici, nebo pracují trošku jiným způsobem.

Pro potřeby jednotlivých ukázkových úloh byla vytvořena a následně i otestována pomocná knihovna vulkano_text_renderer. Tato knihovna primárně slouží k vykreslování libovolného množství textových elementů do okna aplikace. Knihovna musela být v průběhu vývoje několikrát přepsána a optimalizovaná takovým způsobem, aby byl dopad na výkon co nejnižší. Poslední verze této knihovny tedy má přibližně 4% dopad na celkový výkon aplikace s tím, že celkové množství textu nemá na daný výkon prakticky žádný vliv.

Pro tvorbu dokumentace byl zvolen nástroj Writerside od společnosti JetBrains, což se později ukázalo jako docela vhodná volba. Tento nástroj umožnuje jednoduše, rychle a přesně vytvářet dobře vypadající dokumentaci za pomoci jazyků Markdown a XML. Vytvořenou dokumentaci lze rychle a bezproblémově převést do formátu HTML tak, aby mohla být daná dokumentace nasazena jako standardní webová stránka. Samotnou šablonu, z níž jsou následně generovány jednotlivé stránky dokumentace, lze libovolně přizpůsobovat konkrétním potřebám daného projektu. V současné době je tento nástroj stále v relativně rané fázi vývoje a prochází řadou změn. Z tohoto důvodu stále není podporováno zvýrazňování kódu, který je napsán v programovacím jazyku Rust. Z tohoto důvodu bylo nutné najít jiný alternativní programovací jazyk (JavaScript), který by správně zvýrazňoval syntaxi kódu. Podpora programovacího jazyka Rust by měla být přidána v jedné z budoucích aktualizací tohoto nástroje.

9 Závěry a doporučení

Cílem této bakalářské práce bylo prozkoumat možnosti a potencionál grafického rozhraní Vulkan jako možného nástroje pro výuku počítačové grafiky. Úvodní kapitola teoretické části této práce je věnována představení rozhraní Vulkan, včetně historického vývoje a popisu základních konceptů (pojmů), se kterými je možné setkat se v rámci praktické části této bakalářské práce. V následující kapitole jsou popsány a následně i porovnány jednotlivé volně dostupné knihovny, které přímo umožnují využít rozhraní Vulkan. Tyto knihovny byly vybrány napříč různými programovacími jazyky (Java, Rust, C#, Python) a podmínkou pro jejich výběr bylo, aby podporovaly rozhraní Vulkan a byly v současné době stále aktivně vyvíjeny. Třetí kapitola se zabývá představením programovacího jazyka Rust s důrazem na jeho unikátní systém správy paměti. Tento programovací jazyk je v následujících kapitolách použit pro implementaci sady ukázkových úloh.

Praktická část této bakalářské práce je primárně věnována navržení a následné implementaci dvaceti jedna různě rozsáhlých a složitých ukázkových úloh, které ilustrují různé aspekty práce s knihovnou Vulkano. Spolu s jednotlivými ukázkovými úlohami byla implementována i pomocná knihovna vulkano_text_renderer, která slouží k jednoduchému přidávání různých textových elementů a jejich následnému vykreslení do okna aplikace. Jednotlivé ukázkové úlohy jsou navrženy s předpokladem jejich potencionálního využití v rámci výuky počítačové grafiky. Z tohoto důvodu byla za pomoci nástroje zvaného Writerside vytvořena relativně rozsáhlá dokumentace (tutoriál). Tato dokumentace je rozdělena do několika dílčích částí, které na sebe postupně logicky navazují a jejich cílem je systematicky rozšířit znalosti jednotlivých čtenářů. Úvodní část dokumentace je věnována instalaci a konfiguraci všech potřebných nástrojů pro práci s rozhraním Vulkan i knihovnou Vulkano. Následující části jsou věnovány krátkému představení jednotlivých ukázkových úloh a základnímu porovnání programovacích jazyků Rust a Java. Hlavní část dokumentace je věnována popisu implementace jednoduché aplikace s využitím knihovny Vulkano a tato část obsahově zahrnuje všechny informace nutné k implementaci prvních čtyř ukázkových úloh. Poslední část obsahuje seznam volně dostupných zdrojů či tutoriálů, které je možné využít k dalšímu procvičování získaných znalostí. Kompletní seznam všech ukázkových úloh spolu se strukturou dokumentace je možné najít jako přílohy číslo 1 až 3.

Všechny ukázkové úlohy společně s přidruženou dokumentací a pomocnou knihovnou vulkano_text_renderer byly úspěšně otestovány na operačních systémech Windows, Linux a macOS. Kvůli různým technickým omezením musely být jednotlivé ukázkové úlohy speciálně přizpůsobeny potřebám operačního systému macOS. Z tohoto důvodu nejsou některé funkce na tomto operačním systému k dispozici. Zvláštní pozornost byla věnována samotnému testování pomocné knihovny vulkano_text_renderer, s cílem posoudit celkový vliv na výkon dané aplikace. Analýza výsledků odhalila, že použití této knihovny má za následek přibližně 4% snížení výkonu, přičemž samotný objem (množství) zobrazovaného textu nemá na celkový výkon zásadní vliv. Samotné počáteční snížení výkonu je spojeno s nutností využívat další vykreslovací průchod, který do již vykresleného snímku vykreslí požadovaný text, který přesně odpovídá příslušným zcela přizpůsobitelným parametrům (barva, pozadí, pozice, velikost, font).

V rámci samotného úvodu byla položena otázka, zda by bylo možné použít některou z knihoven jako nástroj pro výuku počítačové grafiky. Získat jednoznačnou odpověď na tuto otázku je relativně složité a každý člověk na ni bude mít svůj vlastní názor. Na základě mé osobní zkušenosti se domnívám, že existuje značný potencionál pro úspěšnou integraci jedné z knihoven přímo do výuky počítačové grafiky. Realizace tohoto kroku bude ale vyžadovat pečlivé zvážení a dostatek času na přípravu.

Existuje mnoho více či méně zajímavých možností a námětů, se kterými lze přímo i nepřímo navázat na tuto bakalářskou práci. První z logicky se nabízejících možností je porovnání rozhraní Vulkan s ostatními dnes běžně používanými grafickými API, jako je například DirectX nebo Metal. V kontextu samotné výuky počítačové grafiky by bylo obzvláště zajímavé prozkoumat, jakým způsobem mohou různé pedagogické přístupy a metody ovlivnit schopnost studentů si osvojit práci s tímto rozhraním. To by mohlo zahrnovat porovnání různých formátů výuky, případně různou formu pomocných či výukových materiálů. Další významnou možností je hlubší prozkoumání jedné z výše představených knihoven pro práci s rozhraním Vulkan. Kvůli současné popularitě přenositelných technologií by bylo dále zajímavé prozkoumat možné využití rozhraní Vulkan při vytváření mobilních či AR (augmented reality) aplikací. Poslední, ale neméně zajímavou možností, je vytvoření vlastního wrapperu (obálky) okolo rozhraní Vulkan, a to v libovolném programovacím jazyce. Tato možnost by mohla zahrnovat návrh, implementaci i samotné testování tohoto wrapperu, případně i popis jeho možného využití v případě výuky počítačové grafiky.

10 Seznam použité literatury

[1] SELLERS, Graham a John KESSENICH. *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Boston: Addison-Wesley Professional, 2017. OpenGL series, 1. ISBN 978-0-13-446454-1.

[2] About The Khronos Group. *The Khronos Group* [online]. 21. květen 2023 [vid. 2023-05-21]. Dostupné z: https://www.khronos.org/about/

[3] KENWRIGHT, Benjamin. Getting started with computer graphics and the vulkan API. In: *SA '17: SIGGRAPH Asia 2017*: *SIGGRAPH Asia 2017 Courses* [online]. Bangkok: Association for Computing Machinery, 2017, s. 86 [vid. 2023-05-11]. ISBN 978-1-4503-5403-5. Dostupné z: doi:10.1145/3134472.3136556

[4] LEHN, Karsten, Merijam GOTZES a Frank KLAWONN. *Introduction to Computer Graphics: Using OpenGL and Java*. B.m.: Springer Nature, 2023. ISBN 978-3-031-28135-8.

[5] *History of OpenGL - OpenGL Wiki* [online]. [vid. 2024-01-21]. Dostupné z: https://www.khronos.org/opengl/wiki/History_of_OpenGL

[6] SINGH, Parminder. *Learning Vulkan*. B.m.: Packt Publishing Ltd, 2016. ISBN 978-1-78646-980-9.

[7] Khronos Group Releases Vulkan 1.1. *The Khronos Group* [online]. 7. březen 2018 [vid. 2024-01-27]. Dostupné z: https://www.khronos.org/news/press/khronos-group-releases-vulkan-1-1

[8] Khronos Group Releases Vulkan 1.2. *The Khronos Group* [online]. 15. leden 2020 [vid. 2024-01-27]. Dostupné z: https://www.khronos.org/news/press/khronos-group-releases-vulkan-1.2

[9] Vulkan 1.3 and Roadmap 2022. *The Khronos Group* [online]. 25. leden 2022 [vid. 2024-01-27]. Dostupné z: https://www.khronos.org/blog/vulkan-1.3-and-roadmap-2022

[10] *Vulkan SDK 1.3.275.0 Release Notes* [online]. [vid. 2024-01-24]. Dostupné z: https://vulkan.lunarg.com/doc/view/latest/windows/release_notes.html

[11] LAPINSKI, Pawel. *Vulkan Cookbook*. Birmingham: Packt Publishing Ltd, 2017. ISBN 978-1-78646-815-4.

[12] *Introduction - Vulkan Tutorial* [online]. [vid. 2024-02-01]. Dostupné z: https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction

[13] PEDDIE, Jon. What is a GPU? In: Jon PEDDIE, ed. *The History of the GPU - Steps to Invention* [online]. Cham: Springer International Publishing, 2022 [vid. 2024-03-12], s. 333–345. ISBN 978-3-031-10968-3. Dostupné z: doi:10.1007/978-3-031-10968-3_7

[14] MONTOTO GONZÁLEZ, Manuel. Hands-on study on Vulkan and the hardware ray-tracing extensions [online]. 2021 [vid. 2024-01-30]. Dostupné z: https://ddd.uab.cat/record/248457

[15] *LWJGL - Lightweight Java Game Library* [online]. [vid. 2024-01-25]. Dostupné z: https://www.lwjgl.org/

[16] B, Jean-Sébastien. *realitix/vulkan* [online]. C++. 20. leden 2024 [vid. 2024-01-25]. Dostupné z: https://github.com/realitix/vulkan

[17] *ash-rs/ash* [online]. Rust. B.m.: ash-rs. 24. leden 2024 [vid. 2024-01-25]. Dostupné z: https://github.com/ash-rs/ash

[18] *vulkano-rs/vulkano* [online]. Rust. B.m.: vulkano-rs. 25. leden 2024 [vid. 2024-01-25]. Dostupné z: https://github.com/vulkano-rs/vulkano

[19] *GitHub - gfx-rs/wgpu: Cross-platform, safe, pure-rust graphics api.* [online]. [vid. 2024-01-26]. Dostupné z: https://github.com/gfx-rs/wgpu

[20] *Introduction | Learn Wgpu* [online]. [vid. 2024-01-26]. Dostupné z: https://sotrh.github.io/learn-wgpu/

[21] *Silk.NET - High-Speed & Advanced .NET Graphics & Compute* [online]. [vid. 2023-05-26]. Dostupné z: https://dotnet.github.io/Silk.NET/

[22] *Veldrid | Introduction* [online]. [vid. 2024-01-26]. Dostupné z: https://veldrid.dev/articles/intro.html

[23] GitHub. *GitHub* [online]. [vid. 2023-05-26]. Dostupné z: https://github.com/

[24] *crates.io: Rust Package Registry* [online]. [vid. 2023-05-27]. Dostupné z: https://crates.io/

[25] ESHWARLA, Prabhu. *Practical System Programming for Rust Developers: Build fast and secure software for Linux/Unix systems with the help of practical examples*. B.m.: Packt Publishing Ltd, 2020. ISBN 978-1-80056-201-1.

[26] BUGDEN, William a Ayman ALAHMAR. *Rust: The Programming Language for Safety and Performance* [online]. B.m.: arXiv. 11. červen 2022 [vid. 2024-02-02]. Dostupné z: doi:10.48550/arXiv.2206.05503. arXiv:2206.05503 [cs]

[27] KLABNIK, Steve a Carol NICHOLS. *The Rust Programming Language, 2nd Edition*. B.m.: No Starch Press, 2023. ISBN 978-1-71850-310-6.

[28] *Rust — The Linux Kernel documentation* [online]. [vid. 2024-01-24]. Dostupné z: https://www.kernel.org/doc/html/next/rust/index.html

[29] *BlueHat IL 2023 - David Weston - Default Security* [online]. 2023 [vid. 2024-01-24]. Dostupné z: https://www.youtube.com/watch?v=8T6ClX-y2AE

[30] Stack Overflow Developer Survey 2023. *Stack Overflow* [online]. [vid. 2024-01-24]. Dostupné z: https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023

[31] UNTERGUGGENBERGER, Johannes, Bernhard KERBL a Michael WIMMER. Vulkan all the way: Transitioning to a modern low-level graphics API in academia. *Computers & Graphics* [online]. 2023, 111, 155–165. ISSN 0097-8493. Dostupné z: doi:10.1016/j.cag.2023.02.001

11 Přílohy

1) Struktura projektu
2) Obecná struktura jednotlivých ukázkových úloh
3) Struktura dokumentace
4) Vzhled dokumentace

Dokumentaci, zdrojové kódy i spustitelné soubory jednotlivých projektů je možné najít na platformě GitLab: https://gitlab.com/VulkanoThesis/vulkano_examples

Základní struktura projektu:

```
theses
documentation
├── HTML
└── Writerside
examples
├── libs
│   └── vulkano_text_renderer
├── libs_benchmarks
│   └── text_renderer_benchmark
├── p00_window
├── p01_basic_triangle
├── p02_uniform_buffer
├── p03_geometry
├── p04_textures
├── p05_camera
├── p06_obj
├── p07a_loop_rendering
├── p07b_instancing
├── p07c_offset_rendering
├── p08_light
├── p09_texture_array
├── p10a_compute_shader
├── p10a_compute_shader
├── p11_geometry_shader
├── p12_multiple_shaders
├── p13_post_processing
├── p14a_tessellation
├── p14b_tessellation
├── p15a_deferred_shading
└── p15b_deferred_shading
win-exe
```

theses – Složka obsahuje finální verzi bakalářské práce ve formátech PDF a DOCX.

documentation – Složka obsahuje dokumentaci/návody popisující základy práce s knihovnou Vulkano.

examples – Složka obsahuje zdrojové kódy všech aplikací, knihoven a benchmarků.

win-exe – Složka obsahuje všechny ukázkové úlohy zkompilované do formátu EXE. Tyto zkompilované ukázky lze spustit pouze na operačním systému Windows.

Obecná struktura jednotlivých ukázkových úloh:

```
pxx_example_name
├── resources
│   ├── obj
│   ├── icons
│   └── textures
└── src
    ├── shaders
    ├── solid
    ├── utils
    └── vulkano
        ├── buffers
        ├── core
        ├── devices
        ├── graphics_pipeline
        └── sub_buffers
```

resources – Tato složka obsahuje různé pomocné soubory, jako jsou například textury, ikonky nebo objekty ve formátu OBJ.

src – Hlavní adresář, ve kterém se nachází veškerý zdrojový kód dané ukázky.

shaders – Složka se shadery ve formátu GLSL.

solid – Složka s funkcemi a strukturami spojenými s definováním jednotlivých objektů (solidů).

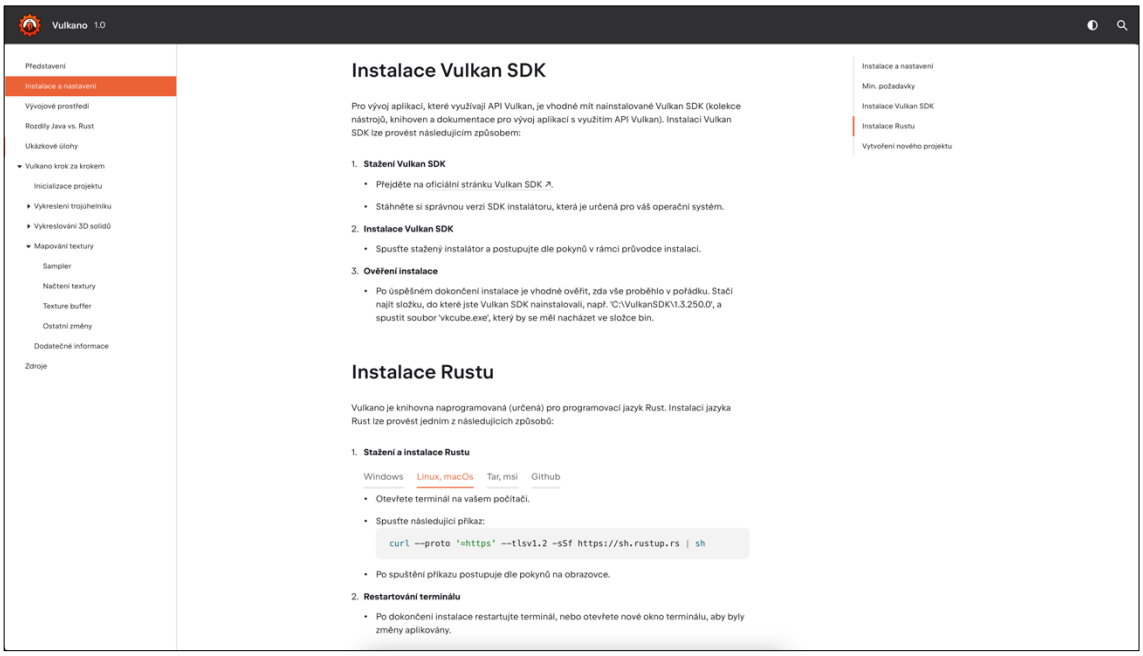
utils – Složka s pomocnými funkcemi a metodami.

vulkano – Složka s implementací všech potřebných funkcí a struktur pro sestavení vykreslovací smyčky.

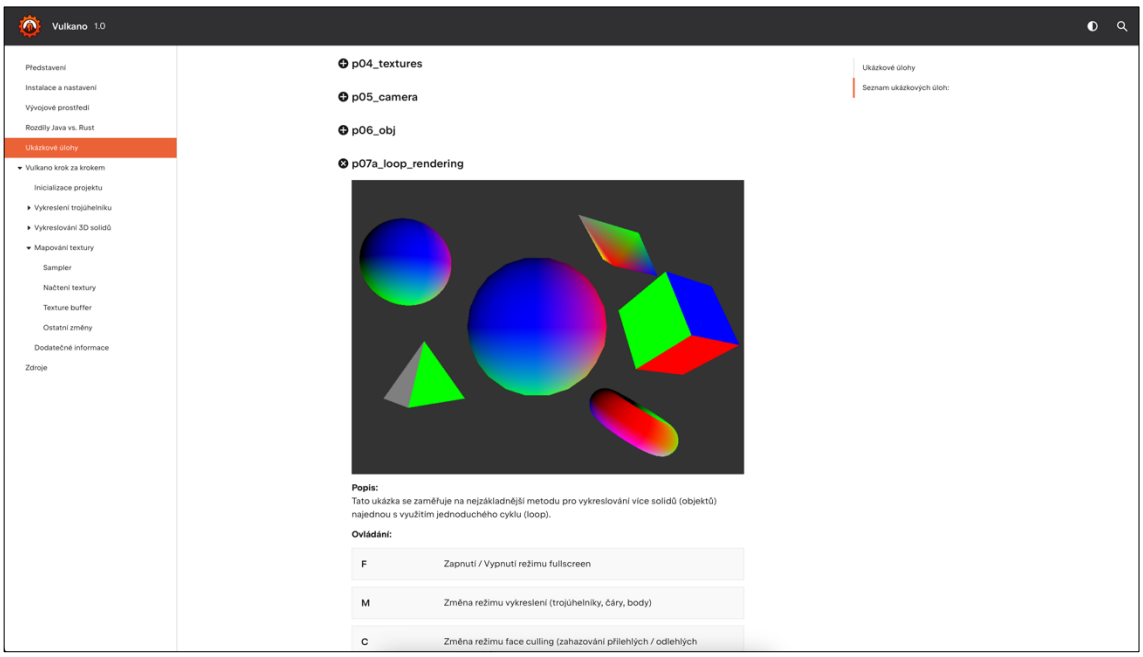
Struktura dokumentace:

```
Představení
Instalace a nastavení
Vývojové prostředí
Rozdíly Java vs. Rust
Ukázkové úlohy
Vulkano krok za krokem
├── Inicializace projektu
├── Vykreslení trojúhelníku
│   ├── Instance
│   ├── Fyzické a logické zařízení
│   ├── Grafická pipeline
│   │   ├── Načítání shaderů
│   │   ├── Vertex buffer
│   │   ├── Swap chain
│   │   ├── Render pass a framebuffer
│   │   └── Grafická pipeline
│   └── Vykreslení trojúhelníku
├── Vykreslování 3D solidů
│   ├── Z-buffer
│   ├── Solid
│   ├── Index buffer
│   ├── Uniform buffer
│   └── Ostatní změny
├── Mapování textury
│   ├── Sampler
│   ├── Načítání textury
│   ├── Texture buffer
│   └── Ostatní změny
└── Dodatečné informace
Zdroje
```

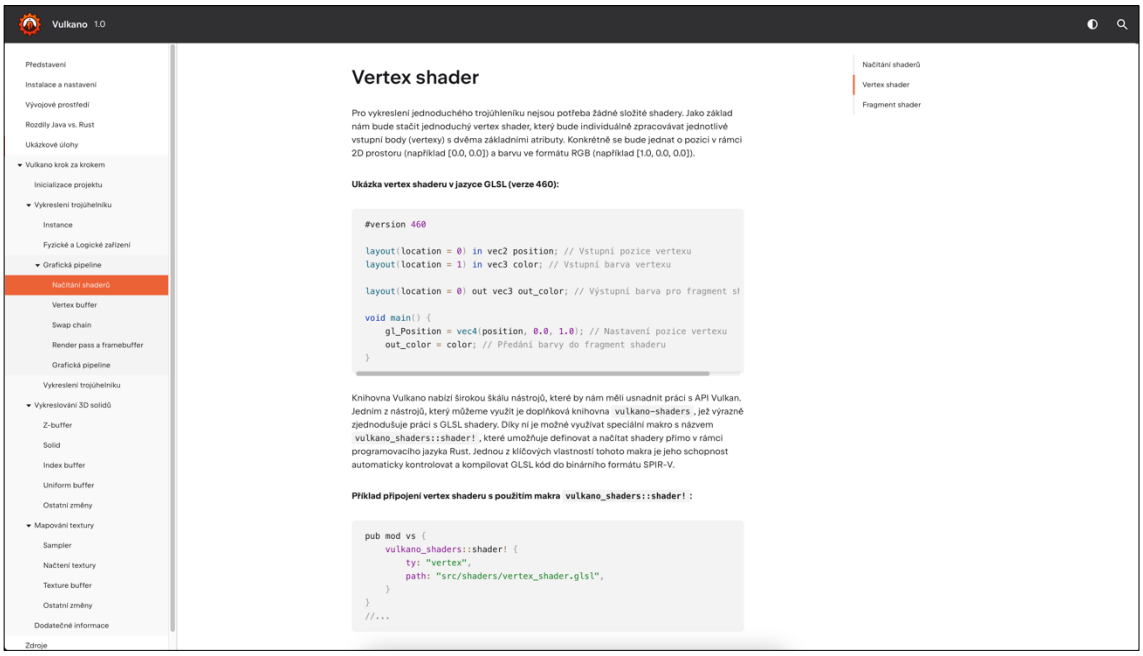
Vzhled dokumentace:



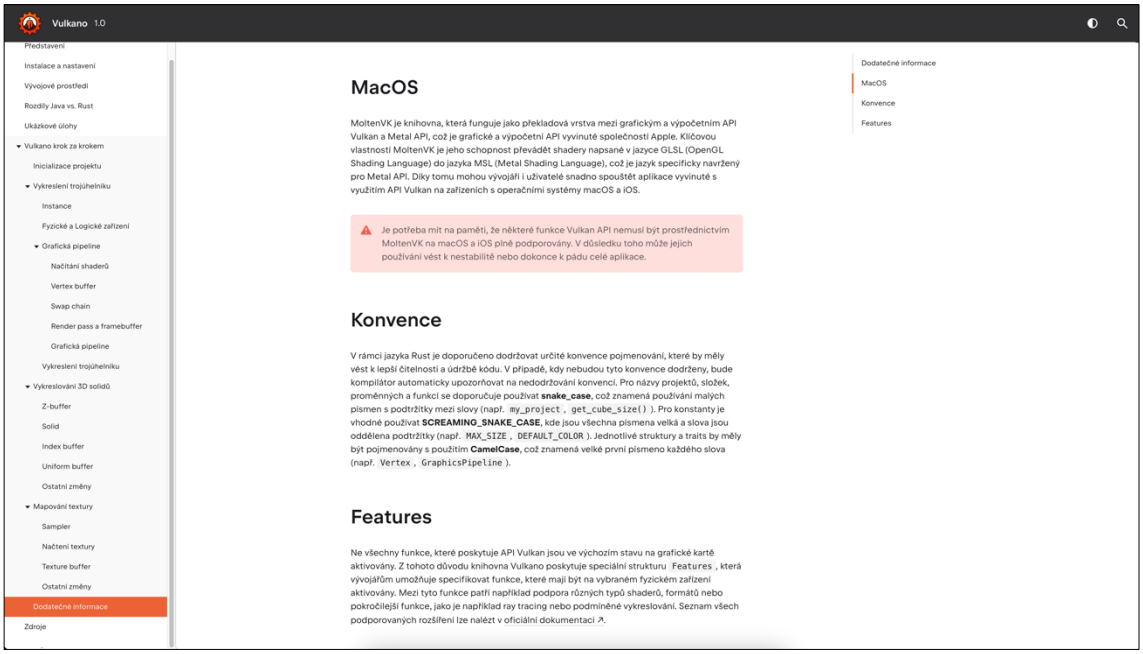
Obr. 19: Ukázka vzhledu dokumentace – „Instalace a nastavení“
Zdroj: Vlastní zpracování



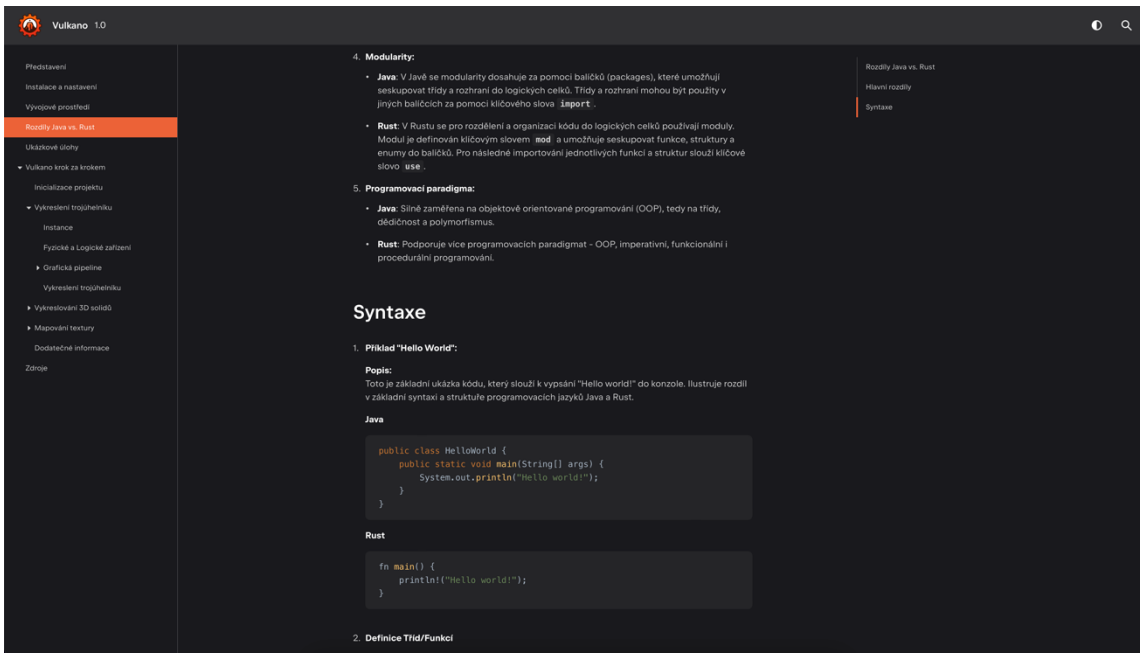
Obr. 20: Ukázka vzhledu dokumentace – „Ukázkové úlohy“
Zdroj: Vlastní zpracování



Obr. 21: Ukázka vzhledu dokumentace – „Načítání shaderů“
Zdroj: Vlastní zpracování



Obr. 22: Ukázka vzhledu dokumentace – „Dodatečné informace“
Zdroj: Vlastní zpracování



Obr. 23: Ukázka vzhledu dokumentace – „Rozdíly Java vs. Rust“ – dark mode
Zdroj: Vlastní zpracování

12 Zadání práce z IS (eVŠKP)

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Dominik Prokop**
Studium:	I2100267
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Využití API Vulkan v kontextu výuky počítačové grafiky**
Název bakalářské práce AJ:	Utilization of the Vulkan API in the context of teaching computer graphics

Cíl, metody, literatura, předpoklady:

Cíl práce:

Cílem práce je prozkoumat možnosti využití API Vulkan jako nástroje pro výuku počítačové grafiky. Navrhnout a implementovat sadu ukázkových úloh, které demonstrující a usnadňující vývoj vlastních aplikací.

Postup prací:

1. Prozkoumat možné využití API Vulkan v kontextu výuky počítačové grafiky.
2. Vytvořit přehled možných knihoven pro práci s API Vulkan.
3. Navrhnout sadu ukázkových úloh demonstrujících možnosti knihovny Vulkano.
4. Navržené úlohy implementovat a otestovat.
5. Zhodnotit dosažené výsledky.

[1] SELLERS, Graham a John KESSENICH. Vulkan Programming Guide: The Official Guide to Learning Vulkan. Boston: Addison-Wesley Professional, 2017. OpenGL series, 1. ISBN 978-0-13-446454-1.

[2] LAPINSKI, Pawel. Vulkan Cookbook. Birmingham: Packt Publishing Ltd, 2017. ISBN 978-1-78646-815-4.

[3] UNTERGUGGENBERGER, Johannes, Bernhard KERBL a Michael WIMMER. Vulkan all the way: Transitioning to a modern low-level graphics API in academia. Computers & Graphics [online]. 2023, 11. ISSN 0097-8493. Dostupné z: doi:10.1016/j.cag.2023.02.001

Zadávající pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	Ing. Bruno Ježek, Ph.D.
Datum zadání závěrečné práce:	26.1.2021