Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Bakalářská práce

Tvorba 3D aplikací s pomocí Vulkan API

Lukáš Veteška

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Lukáš Veteška

Informatika

Název práce

Tvorba 3D aplikací s pomocí Vulkan API

Název anglicky

3D applications development using Vulkan API

Cíle práce

Bakalářská práce je tematicky zaměřena na problematiku vývoje 3D aplikací s Vulkan API. Hlavním cílem práce je ověření využitf Vulkan API pro tvorbu 3D terénu na základě spojení s mapovými podklady k tisku na 3D tiskárně. Dílčí cíle práce jsou:
- charakteristika současných technologií a grafických API,
- charakteristika nástrojů pro tvorbu 3D terénu,
- vytvoření a otestování reálné aplikace.

Metodika

Metodika řešení bakalářské práce je založena na analytickém a syntetickém přístupu. Bude provedena analýza odborných informačních zdrojů, jejíž souhrn bude uveden v teoretické části práce. Na základě syntézy zjištěných poznatků bude zpracován popis současného stavu, nástrojů a technologií pro tvorbu 3D terénu ve spojení s mapovými podklady k tisku na 3D tiskárně.

Dále bude vytvořena ukázková aplikace s využitfm Vulkan API. Na základě aplikace bude ověřena možnost převodu mapových podkladů na 3D terén, který bude možno vytisknout na 3D tiskárně. Na základě výsledků teoretické a praktické části budou shrnuty poznatky a formulovány závěry.

Doporučený rozsah práce

45 stran

Klíčová slova

3D, editor, Vulkan, Qt, C++, grafická aplikace

Doporučené zdroje informací

CHROBOCZEK, Martin. Grafická uživatelská rozhraní v Qt a C++: [plně kompatibilní s Qt 5]. Brno: Computer Press, 2013. ISBN 978-80-251-4124-3.

KENWRIGHT. Introduction to Computer Graphics and the Vulkan API. CreateSpace Independent Publishing Platform, 2017. ISBN 978-1548616175.

SELLERS, Graham a John. Vulkan Programming Guide: The Official Guide to Learning Vulkan. Addison-Wesley Professional, 2016. ISBN 0134464540.

Předběžný termín obhajoby

2018/19 LS – PEF

Vedoucí práce

Ing. Jan Masner, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 11. 9. 2018

Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 10. 2018

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 27. 02. 2019

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Tvorba 3D aplikací s pomocí Vulkan API" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 28. 2. 2019 _____

Poděkování

Rád bych touto cestou poděkoval Ing. Janu Masnerovi, Ph.D. za vstřícné Jednání a rady při psaní práce. Také mé rodině za podporu během mého studia.

Tvorba 3D aplikací s pomocí Vulkan API

Abstrakt

Bakalářská práce se zaměřuje na problematiku tvorby 3D terénu s grafickým API – Vulkan na základě mapových podkladů pro tisk na 3D tiskárně. Teoretická část se zabývá porovnáním konkurenčních grafických API a seznámením s jejich historií. Následuje srovnání programovacích jazyků, které jsou vhodné pro použití s grafickými API. V dalších částech dochází k popisu renderování grafickou API od samotného základu k renderování scény, současné a dříve používané metody renderingu 3D terénu, využití mapových podkladů z webových služeb a převedení terénu na 3D model pro 3D tisk.

Praktická část se nejdříve zabývá předpoklady pro implementaci aplikace s Vulkan API a dále je vytvořena počáteční infrastruktura zdrojového kódu. V další fázi dochází k samotné implementaci infrastruktury a jejímu ověření v praxi. Praktickou část uzavírá zhodnocení aplikace a dosažených výsledků.

Klíčová slova: 3D, terén, grafické, API, Vulkan, Qt, C++, grafická aplikace, rendering

3D applications development using Vulkan API

Abstract

The bachelor thesis focuses on issues of 3D terrain creation with graphics API – Vulkan based on map data for print on 3D printer. Theoretical part is focused on the comparison of competitive graphics API and introducing their history. Following the comparison of programming languages which are suitable for using graphics API. Next parts describe graphics API rendering since the beginning to scene rendering, current and previous used 3D terrain rendering techniques, use of map data from web services and transforming terrain to 3D model for 3D print.

Practical part deals first with assumptions for implementation of application with Vulkan API then is created initial infrastructure of application. The next stage occurs implementation of application infrastructure itself and verification in practice. Practical part is closed by evaluation of application and achieved results.

Keywords: 3D, terrain, graphics, API, Vulkan, Qt, C++, graphics application, rendering

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

Seznam použitých zkratek

API - Application programming interface (Rozhraní pro programování aplikací)
GUI - Graphic User Interface (Grafické uživatelské prostředí)
LOD - Level of Detail (Úroveň detailů)
Primitiva - Geometrický prvek
Vertex - Bod v prostoru
Patch - Obecně-učelová primitiva, kde každý n vertex je nová patch primitiva
Pipeline - Koncepční model popisující kroky systému
OS - Operační systém
CPU - Cental processing unit (Centrální procesorová jednoka)
GPU - Graphics processing unit (Grafická procesorová jednotka)
Pointer - Datový typ pro uložení adresy v paměti
Buffer - Část paměti pro uchování dat
VS - Vertex shader
TCS - Tessellation Control Shader (Tesselační kontrolní shader)
TES - Tessellation Evaluation Shader (Tesselační evaluační shader)
FG - Fragment shader

Grid	- Mřížka
Tile	- Dlaždice
Chunk	- Díl (kus)
STL	- Stereografický souborový formát
Flag	- V programování bit nebo bitová sekvence obsahující binární hodnotu
Wrapper	- Tenká vrstva kódu převádějící nativní kód do kompatibilního rozhraní

1 Úvod

Na počátku éry počítačů se pracovalo s děrnými štítky, příkazovým řádkem a textovým uživatelským rozhraním. Dnes se pracuje především s grafickým prostředím, ve kterém je možné nalézt intuitivní ovládací prvky. V současné době se používá ve většině zaměstnání k práci počítač, tablet, mobil či vestavěné zařízení. V přítomnosti ani v blízké budoucnosti se zdá, že nebude sjednocený operační systém na všech zařízeních, jelikož každé z nich má svůj účel, jiný grafický čip, každý uživatel má odlišný požadavek.

Grafické API je standard rozhraní pro komunikaci aplikace s grafickým ovladačem, který umožňuje aplikacím pracovat na různých operačních systémech s grafickými čipy. Grafické API je implementováno grafickým ovladačem od výrobce grafického čipu. Například: aplikace pro operační systém Windows bude fungovat, přestože architektury dodávaných grafických čipů jsou odlišené, protože grafický ovladač pro svůj čip obsahuje stejné API. Větší část grafických API je dostupná pouze pro vybrané operační systémy z důvodu konkurenční války mezi operačními systémy.

V roce 2016 byla dokončena specifikace Vulkan API. Jde o grafické a výpočetní API, jenž má představovat velkého hráče na poli grafických API zásluhou rychlosti ve srovnání s konkurencí. Khronos Group v minulosti vyvinulo vysokoúrovňové OpenGL, nyní přichází s nízkoúrovňovým Vulkan API, které vychází z AMD Mantle API. Přichází podpora klíčové funkce multi-threading, se kterou se snižuje spotřeba a současně se zvyšuje výkon procesoru. Zároveň dává vývojářům více kontroly nad hardwarovými prostředky.

Vulkan API zejména cílí na grafické aplikace, tedy herní enginy, strojírenské a průmyslové softwary. V herním průmyslu se využívá pro vykreslování herních scén, ve strojírenství pak především pro vytváření modelů pro CNC stroje či dnes už tak k populárnímu 3D tisku. U oblasti 3D tisku lze předpokládat velký rozvoj, neboť jde o levnou technologii k vytváření prototypů. Vývoj 3D tiskáren se za poslední desetiletí rapidně zvýšil a rozhodně není na svém technologickém maximu. Nově se 3D tiskárny používají např. k tisku domů čímž dochází k velkému snížení nákladů.

2 Cíl práce a metodika

2.1 Cíl práce

Bakalářská práce je tematicky zaměřená na problematiku vývoje 3D aplikací s Vulkan API. Toto grafické API je nové ve svém oboru. Budou rozebrány jeho přednosti i nedostatky.

Hlavním cílem práce je ověření využití Vulkan API pro tvorbu 3D terénu na základě spojení s mapovými podklady k tisku na 3D tiskárně. Dílčí cíle práce jsou:

- charakteristika současných technologií a grafických API,
- charakteristika nástrojů pro tvorbu 3D terénu,
- vytvoření a otestování reálné aplikace.

2.2 Metodika

Metodika řešení bakalářské práce je založena na analytickém a syntetickém přístupu. Bude provedena analýza odborných informačních zdrojů, jejíž souhrn bude uveden v teoretické části práce. Na základě syntézy zjištěných poznatků bude zpracován popis současného stavu, nástrojů a technologií pro tvorbu 3D terénu ve spojení s mapovými podklady k tisku na 3D tiskárně.

Dále bude vytvořena ukázková aplikace s využitím Vulkan API. Na základě aplikace bude ověřena možnost převodu mapových podkladů na 3D terén, který bude možno vytisknout na 3D tiskárně. Na základě výsledků teoretické a praktické části budou shrnuty poznatky a formulovány závěry.

3 Teoretická východiska

3.1 Grafické API

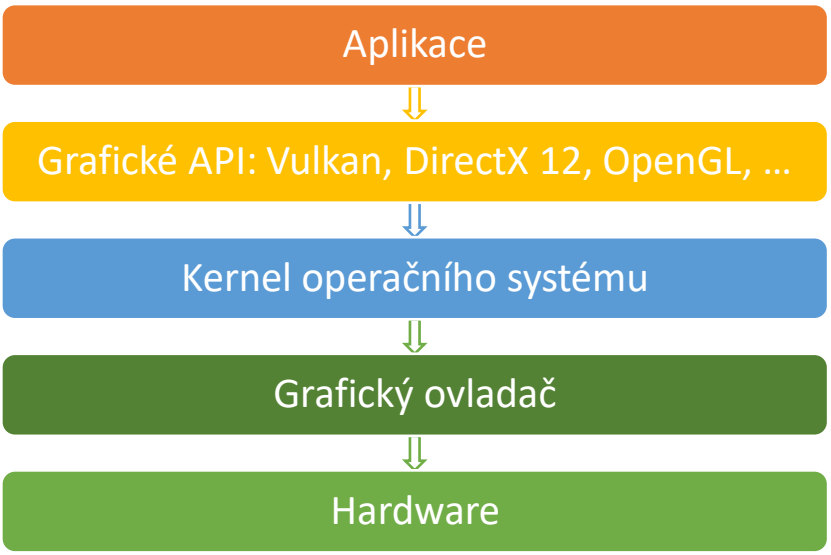
Základní kámen pro komunikaci se softwarem tvoří tzv. API (Application programming interface neboli rozhraní pro programování aplikací). Jedná se o sadu definicí, komunikačních protokolů a nástrojů pro vytváření softwaru. Obecně je to sada metod definující komunikaci mezi komponentami. Kvalitní API usnadňuje vývoj počítačového programu poskytováním všech stavebních komponent, které posléze poskládá programátor. [1]

API mohou nabývat mnoho forem. Windows API ve srovnání s POSIX jsou dvě rozdílné věci. Mezi jednu z forem API patří grafické API. Operační systémy (dále OS) byly navrženy tak, aby uživatelé a programátoři neměli přístup k samotnému hardwaru. Pro vykreslování je nutné pracovat s grafickými kartami/čipy, ke kterým má přístup pouze OS. Vznikly tedy standardy grafických API, které jsou typicky implementované grafickým ovladačem od výrobce grafického čipu. [2]

Standardizací grafického API můžeme spustit aplikace, které budou mít stejný výstup na různých architekturách grafického čipu, protože grafické ovladače implementují stejné standardizované grafické API pro všechny druhy architektur. Na obrázku č. 1 lze vidět diagram vykreslování. [3]

Mezi nejznámější standardizované grafické API patří:

- DirectX
- Mantle (vývoj ukončen)
- Metal
- MonoGame
- OpenGL
- OpenGL ES
- Vulkan
- WebGL



Obrázek 1: Diagram vykreslování [36]

3.1.1 Vulkan API

Vulkan je nízkoúrovňová, cross-platform 3D grafická a výpočetní API. Cílí na vysoce výkonné realtime 3D grafické aplikace jako počítačové hry, strojírenské aplikace a interaktivní media na všech platformách. V porovnání s ostatními API nabízí lepší výkon a rovnovážné využití CPU/GPU. Vulkan má schopnost snižovat spotřebu CPU a zároveň zvýšit výkon díky lepší distribuci práce mezi ostatní jádra CPU. [4]

Vulkan byl poprvé ohlášen Khronos Group konsorciem v roce 2015 na Game Developers Conference. Vulkan API mělo být původně iniciací další generace OpenGL neboli „OpenGL next“ od Khronos Group. Vulkan vychází a je postaven na AMD Mantle API, kterou AMD darovalo Khronosu se záměrem nastartování vývoje nízkoúrovňové API, jež mohla být standardizována pro celé odvětví přesně jako OpenGL. [5]

Vulkan API bylo napsáno v jazyce C, vydáno bylo 16. února 2016. V současné době je dostupná verze 1.1.91 z 4. listopadu 2018. Podporuje operační systémy Android, iOS, Linux, macOS, Microsoft Windows, Nintendo Switch a Tizen.

Vulkan stejně jako jeho předchůdce OpenGL má několik výhod oproti ostatním API. Nabízí větší kontrolu GPU a nižší využití CPU. Celkový koncept a seznam vlastností je velice podobný DirectX 12, Metal a Mantle. [6]

Výhody Vulkanu oproti API předchozích generací [6]:

- Výborné vlastnosti pro high-end grafické karty, stejně jako pro grafický hardware na mobilní zařízení.
- V porovnání s DirectX 12 je dostupný na několika moderních operačních systémech stejně jako OpenGL. Vulkan API není zaměřena na jediný OS nebo zařízení.
- Sníženo přetížení ovladače, tím se snižuje využití CPU.
- Lepší podpora škálovatelnosti pro více jádrové CPU.
- Vulkan předkompilovává shadery do binární podoby nazývané SPIR-V. Předkompilací se zrychluje aplikace a je možné použít více shaderů v dané scéně. Ovladač Vulkanu potřebuje pouze optimalizaci pro specifické GPU a generaci kódu, která vyústí v jednodušší údržbu ovladače a eventuálně menšímu balíčku ovladače.
- Sjednocený management výpočetních kernelů a grafických shaderů, jenž eliminují potřebu jiného výpočetního API ve spojení s grafickou API.

7. března 2018 oznámilo vydání první minor verze Vulkan API 1.1 společně se SPIR-V 1.3. Dochází k rozšíření funkcionality jádra na požadavky vývojářů jako např.: podskupinové operace (vysoce účinné sdílení a manipulace s daty mezi paralelními úlohami na GPU), integrace široké škály rozšíření z verze 1.0 (multi-image views, cross-process API) a podpora

HLSL. Vulkan běží nativně skoro na všech grafických platformách jako Windows 7, 8.X, 10, Android 7.0+ a Linux.

Vulkan se široce rozšiřuje díky vedoucím herním enginům jako Unreal Engine 4, Unity a Source 2. Vulkan je použit již v přes 30 AAA titulech jako Doom, Dota 2, Quake. [7]

3.1.2 OpenGL

Open Graphics Library (OpenGL) je vysoko úrovňová cross-platform API pro vykreslování 2D a 3D vektorové grafiky. API komunikuje s GPU k dosažení hardwarové akcelerace.

V roce 1991 Silicon Graphics Inc. započalo vývoj OpenGL a vydalo jej v lednu 1992. Od roku 2006 přešlo OpenGL pod Khronos Group konsorcium. OpenGL je napsáno v jazyce C stejně jako jeho nástupce Vulkan. Ke dni 31. července 2017 byla vydána poslední verze 4.6.

OpenGL specifikace představuje abstraktní API pro vykreslování 2D a 3D grafiky. Ačkoliv je možné, aby API byla implementována zcela na úrovni softwaru, její návrh je takový, aby byla především implementována, nebo alespoň většinově v hardwaru. [8]

API definuje několik funkcí, které mohou být zavolané klientským programem. OpenGL má mnoho jazykových vazeb, např. WebGL (API, založená na OpenGL ES 2.0 pro 3D vykreslování ve webovém prohlížeči).

OpenGL oproti Direct3D není vázáná na operační systém, proto ho nalezneme nejčastěji na Linuxu a starších verzí MacOS. Specifikace OpenGL neříkají nic o získávání a řízení OpenGL kontextu, což tedy znamená, že toto řízení přenechává okennímu systému daného OS. Proti Vulkan API je OpenGL čistě zaměřeno na vykreslování, nenabízí žádnou API spojenou se vstupem, audiem nebo správou oken.

OpenGL má speciální verzi pro vestavěné zařízení „OpenGL for Embedded Systems“ (OpenGL ES). Je navržena pro vestavěné zařízení jako mobilní zařízení, tablety, herní konzole a PDA. OpenGL ES je nejrozšířenější 3D API v historii. Je jak cross-language tak cross-platform. [9]

OpenGL využívá vysokoúrovňového jazyka GLSL pro psaní shaderů, které nutí každý ovladač implementovat vlastní kompilátor GLSL. GLSL je překládáno v runtimu, čímž se snižuje rychlost. [8] [9]

3.1.3 DirectX a Metal API

DirectX a Metal API jsou rivalové pro Vulkan API. Benefitují na tom na čem prakticky vyhořívají – cross-platform. Většině vývojářů ani nevadí, že jsou vzhledem, ve srovnání

OpenGL a Vulkan API closed-source. Pro vývojáře, který bude cílit na více platforem, je výběr DirectX nebo Metal cesta do pekel. Naproti tomu, pokud budeme vyvíjet čistě na danou platformu, je volba DirectX a Metal API jistá.

3.1.3.1 DirectX

DirectX je kolekce API pro úlohy od multimédia, herní programování po videa na Microsoftí platformě. DirectX obsahuje SDK, kde nalezneme všechny runtime knihovny v binární formě společně s dokumentací. DirectX přišlo na svět společně s Windows NT 4.0, běželo již na Windows 95 OSR 2. S vydáním Windows 8 byla kolekce DirectX integrována do Windows SDK. Nejnovější verzí je DirectX 12 uvedená společně s Windows 10. Bohužel nevýhoda DirectX 12 je, že neobsahuje zpětnou kompatibilitu se staršími systémy Windows. Hlavní výhodou DirectX 12 je multi-threading. DirectX má vlastní shaderovací jazyk HLSL. [10]

3.1.3.2 Metal API

Metal API je nízkoúrovňová hardwarově akcelerovaná 3D grafická a výpočetní API od společnosti Apple. Metal API kombinuje funkce podobné OpenGL a OpenCL. Záměrem této API bylo přinést výkonnostní benefity. Metal API bylo uvedeno společně s iOS 8 v roce 2014, tedy o 2 roky dříve, než bylo vydáno Vulkan API. Metal lze využít v programovacím jazyce Swift nebo Objective-C. Metal má vlastní shaderovací jazyk založený na C++14. Je nutno poznamenat, že než byl vydán Metal API jediným API pro MacOS bylo OpenGL.

Metal API lze použít na operačních systémech iOS, macOS a tvOS. Podporuje pouze vlastní SoC (Systém on chip) od verze Apple A7 a vyšší. MacOS podporuje Metal API pouze na grafických čipech Intel HD a Intel Iris Graphics od verze HD 4000 a vyšší, AMD GCN a NVIDIA Kepler nebo novější.

Metal API 2 bylo oznámeno na konferenci WWDC 5. června 2017. Zvyšuje grafický a výpočetní výkon, snižuje interakci s CPU. Dochází k předání ještě většího množství kontroly nad grafickou pipelinou, akcelerací neuronové sítě, profilování a debugování shaderů v Xcode. [11]

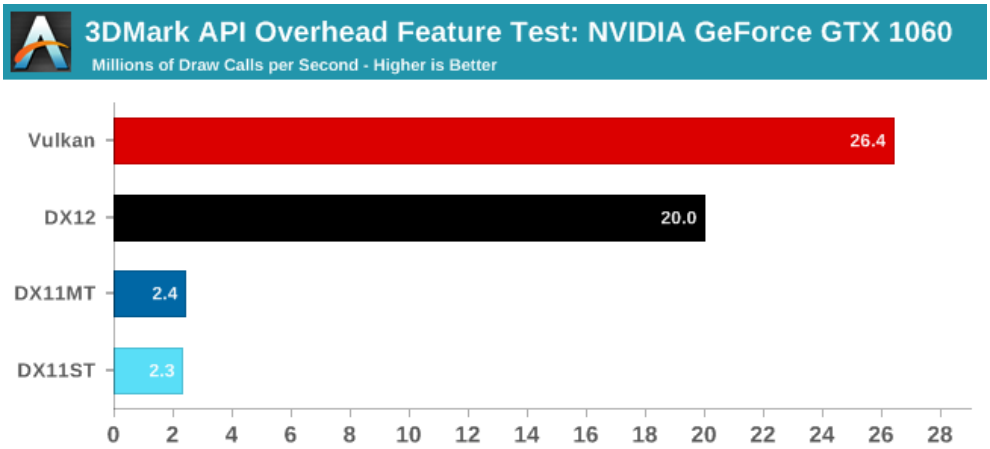
3.1.4 Srovnání

V tabulce č. 1 lze vidět srovnání grafických API. Z tabulky je zřejmé, že Vulkan API především nabízí vývojářům pokračovat v jejich oblíbených shader jazycích: GLSL a HLSL.

	Vulkan	OpenGL	DirectX 12	Metal
Multi-threading	✓	✗	✓	✓
Cross-platform	✓	✓	✗	✗
GLSL	✓	✓	✗	✗
HLSL	✓	✗	✓	✗
Open-source	✓	✓	✗	✗
Platformově zaměřený	✗	✗	✓	✓
Validační vrstva na úrovni aplikace	✓	✗	✓	✓

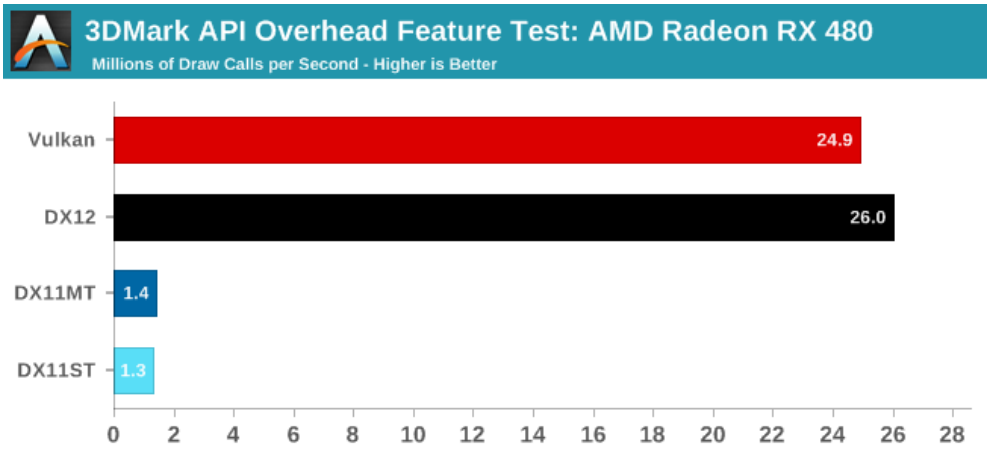
Tabulka 1: Srovnání grafických API [12] [8] [10] [11]

V tabulce č. 1 chybí položka výkon, jelikož každá API je implementovaná jinak a má na daném typu OS výhodu. Dalším ovlivňujícím faktorem je grafická architektura.
Na následujících obrázcích budou ukázané benchmarky ze stejné benchmark aplikace s využitím rozdílných grafických karet.



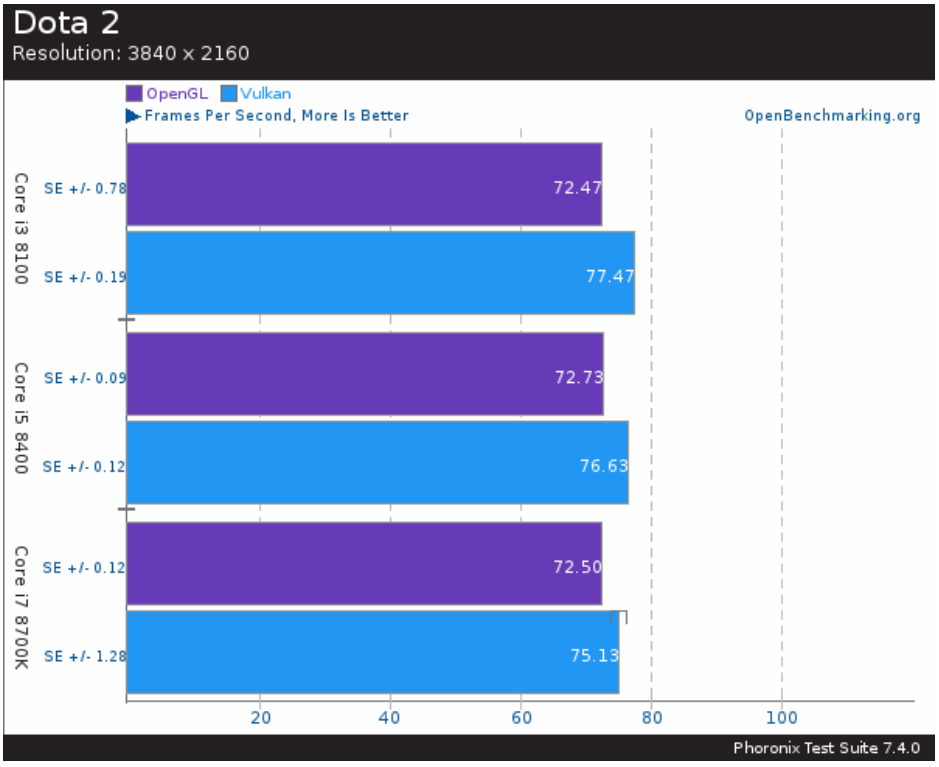
Obrázek 2: Benchmark Vulkan API vs DirectX 12, NVIDIA GeForce GTX 1060 [13]

Na obrázku č. 2 a na obrázku č. 3 lze vidět srovnání benchmark testů Vulkan API vs DirectX 12. Čím vyšší skóre, tím lépe. Jedná se o počet vykreslených snímků za sekundu (fps). Vulkan API si vede oproti DirectX 12 mnohem lépe na grafické kartě od společnosti NVIDIA, kdežto na grafické kartě od společnosti AMD zaostává. Důvodem může být nevyladěný grafický ovladač. [13]



Obrázek 3: Benchmark Vulkan API vs DirectX 12, AMD Radeon RX 480 [13]

Na obrázku č. 4 lze vidět benchmark Vulkan API vs OpenGL z hry Dota 2. Testují se zde tři různé procesory od společnosti Intel pro ověření rychlosti single-threaded renderingu proti multi-threaded renderingu. Za povšimnutí stojí, že nejslabší procesor vychází nejlépe, ovšem je nutné upozornit, že scéna, při které byly tyto výsledky zaznamenané, mohly být náročně odlišné. Je nutno podotknout, že Vulkan API jako nástupce OpenGL vyhrálo. [14]



Obrázek 4: Benchmark Vulkan API vs OpenGL [14]

Vulkan API nelze porovnat s Metal API, a to z toho důvodu, že Vulkan API se na platformě iOS, macOS a tvOS překládá pomocí knihovny MoltenVK od Khronos Group do Metal API. Společnost Apple nemá zájem implementovat jiné řešení než vlastní. [15]

3.2 Programovací jazyky

Použití daného grafického API vyžaduje specifický jazyk. Mezi nejznámější jazyky pro grafické vykreslování patří C, C++, Java a C#. V případě platformy společnosti Apple je nutno dodat, že by nejvíce profitoval nový jazyk Swift. Pokud bude bráno v úvahu, že Vulkan API je napsáno v jazyce C, bude kompatibilní i s jazykem C++. Pro ostatní jazyky existují tzv. wrappery, které popisují interface daného jazyka. V pozadí wrapperu dochází k překladu na funkci jazyka C v případě Vulkan API. [4] [6] [15]

3.2.1 C a C++

V roce 1972 Dennis Ritchie navrhl jazyk C jako obecný jazyk pro programování počítačů, tehdy pro operační systém UNIX. Bjarne Stroustrup vytvořil v roce 1983 vylepšení pro jazyk C pojmenovaný jako C++.

Hlavní rozdíl byl přidáním tříd (čili je objektově orientovaný), virtuálních funkcí, přetížení operátorů, mnohonásobné dědění, templaty, řízení chyb atd. C++ bylo standardizováno jako ISO/IEC 14882:1998 v roce 1998. Nejnovější verze je C++17.

Velkou výhodou obou jazyků je, že jsou snadno přenosné, je prakticky možné je spustit kdekoliv s minimální změnou zdrojového kódu. Jazyk C je znám především tím, že je standard pro to, aby šel spustit kdekoliv. Jazyky jsou rychlejší než většina. Jediný jazyk, který je rychlejší než jazyk C, je pouze Assembler. Udává se, že jazyk C++ ztrácí kolem 5% rychlosti na jazyk C. Pokud se hovoří o realtime programování, oba jazyky jsou vhodné, naproti tomu jazyky jako Java a C# nepřipadají v úvahu. [16]

3.2.2 Java a C#

Programovací jazyk Java byl vydán v roce 1995 firmou Sun Microsystems s cílem „napiš jednou, spusť kdekoliv“ (pozn. - neplatí pro mikročipy). Java společně s C# jsou kompilovány do mezi-jazyka, který se kompiluje ve virtuálním stroji v době spuštění aplikace. Využitím těchto virtuálních strojů vzniká schopnost kompilovaný program dekompilovat, což není možné s nízkoúrovňovými jazyky. Java společně se C# jsou vůči jazyku C a C++ vysokoúrovňové, kdežto jazyk C je nízkoúrovňový a C++ se řadí na střední úroveň. [17]

Java a C# jsou navrhnuté, aby byly typově bezpečné. Největší výhodou je tzv. Garbage Collector, který uvolňuje paměť objektů v mezičase, když nejsou používané. Také pomáhá k tomu, aby nedocházelo k úniku paměti. Bohužel oba jazyky můžou dědit pouze

jednu třídu. Novinkou jsou interfaces, jedná se o abstraktní třídu, kde jsou všechny metody abstraktní, tedy neobsahují žádnou implementaci. [17] [18]

C# je programovací jazyk od společnosti Microsoft, první verze byla vydána v roce 2000. Ke dni 13.11.2018 existuje již sedmá verze. C# těží především z .NET Frameworku, čímž byl doposud odkázán na život pouze v Microsoftí platformě. Nově od roku 2016 se začalo pracovat na .NET Core, jedná se o jednodušší verzi .NET Frameworku s tím, že jej lze spustit na ostatních platformách. [18]

Oba dva jazyky jsou vhodné pro moderní aplikace, ať již webové či desktopové. Lze v nich naprogramovat aplikaci mnohem jednodušeji a rychleji než v jazyce C nebo C++ na úkor rychlosti a řízení paměti, což jsou klíčové vlastnosti u vykreslování 3D grafiky.

3.2.3 Shaderovací jazyky (Shading languages)

Shadery jsou programy určené k řízení jednotlivých fází programovatelného grafického řetězce grafické karty neboli GPU (viz kapitola 3.3.5 - Shadery). Dávají programátorovi větší kontrolu nad vykreslovacím procesem a dodávají bohatší obsah. Nové funkce zvyšují flexibilitu ve vykreslování, jimiž jsou dnešní grafické karty obohaceny. Assembler mezi jazyky pro GPU se nazývá ARB assembly language. Jedná se o nízkoúrovňový shaderovací jazyk, který připomíná assembler. [12] [6]

3.2.3.1 OpenGL Shading Language (GLSL)

GLSL je vysokoúrovňový programovací jazyk pro psaní shaderů, který vychází ze syntaxe jazyka C. Byl vytvořen konsorciem OpenGL ARB (Architecture Review Board), aby vývojář měl více přímé kontroly nad grafickou pipelinou bez použití ARB assembly language nebo hardwarově specifického jazyka. Původní GLSL verze přišla s verzí OpenGL 1.4 jako rozšíření. Oficiálně bylo GLSL představeno až ve verzi OpenGL 2.0 (2004). Nejnovější GLSL verze je 4.60 a běží s OpenGL 4.6 (2017). [12] [6]

3.2.3.2 High-Level Shading Language (HLSL)

HLSL je proprietární vysokoúrovňový shaderovací jazyk vytvořen společností Microsoft pro DirectX 9 API. Vychází ze syntaxe jazyka C a je velice podobný GLSL. [10]

3.2.3.3 Metal Shading Language (MSL)

Apple vytvořil společně s Metal API vlastní vysokoúrovňový shaderovací jazyk MSL. MSL je založeno na C++14, implementováno pomocí kompilátoru Clang a LLVM. [11]

3.3 Vulkan API rendering

Následující podkapitoly jsou parafrázovány z anglického jazyka a založeny převážně na knihách *Vulkan programming guide: the official guide* [4] *to learning vulkan* a *Learning Vulkan* [6], následně na odborně založených webech [19] [20].

3.3.1 Předpoklady

Pro vykreslování počítačové grafiky je bezpodmínečná znalost základní 3D matematiky, zejména matic, vektorů, goniometrie a lineární algebry.

Nejčastější matice v 3D grafice jsou o velikosti 3x3 nebo 4x4. Používají se k transformaci kamery, přes kterou se díváme skrze okno aplikace do prostoru. Matice v počítačové grafice se dělí zejména na:

- Prohlížející
  - směr, kterým se kamera „dívá“ (tzv. pohledová transformace),
- Modelovací
  - skrze ni vkládáme objekty do scény,
- Projekční
  - tvar záběru (měníme ohnisko, vzdálenost, atd.),
- Zobrazovací
  - zvětšení/zmenšení a na mapování výsledné scény

Vektory v počítačové grafice od 2D až po 4D (x, y, z, w):

- 2D vektor
  - kurzor na obrazovce v operačním systému, pozice ve 2D grafice,
- 3D vektor
  - pozice kamery, rotace, prohlížející vektor kamery, data mapy.

3.3.2 Počátek

Pro vykreslení scény je nejprve nutné založit instanci Vulkanu, vytvořit validační vrstvy, vybrat fyzické zařízení, vybrat logické zařízení atd.

3.3.2.1 Instance

Inicializace Vulkanu se provede založením její instance skrze datový typ *VkInstance*. Instance je propojení mezi aplikací a Vulkan API. Vytvořením instance se specifikují potřebné údaje o aplikaci ovladači.

3.3.2.2 Validační vrstvy

Vulkan API je navrhnuta s minimální zátěží na ovladač, a tím tedy došlo k velké limitaci chybových validací. Pouhá chyba nastavením enumerace na špatnou hodnotu nebo předání prázdného pointeru do potřebného parametru nejsou obecně řešeny na straně ovladače a způsobí pád aplikace nebo neznámé chování. Vulkan tedy říká explicitně, že člověk musí vědět, co dělá.

Ačkoliv to neznamená, že nelze tyto kontroly přidat do API. Vulkan představuje elegantní systém známý jako *validační vrstvy*. Validační vrstvy jsou volitelnou komponentou, které řeknou funkcím Vulkanu, aby zavolaly další operace.

Obecné validační vrstvy jsou:

- Kontrola hodnot parametrů vůči specifikaci.
- Sledování vytváření a destrukcí objektů k nalezení úniků.
- Kontrola bezpečnosti vlákna sledováním vláken, odkud volají.
- Logování každé operace a jejich parametrů na standardní výstup.
- Sledování Vulkan operací pro profilování a přehrávání.

3.3.2.3 Rodinné fronty

Téměř každá operace ve Vulkan API od kreslení po nahrávání textur vyžaduje příkazy odeslané do fronty. Existuje několik typů front, které pochází z různých rodinných front. Každý rodina front povoluje určité příkazy. Například může být rodinná fronta povolující zpracování výpočetních příkazů nebo jedna, která zpracovává příkazy paměti.

3.3.2.4 Fronty

Momentálně dostupné ovladače povolují vytvoření pouze malého počtu front pro každou rodinu front. Je možné vytvářet příkazové buffery na více vláknech, a poté je společně odeslat na hlavní vlákno s jediným zavoláním. Vulkan API umožňuje upřednostňovat fronty naplánováním command bufferu (viz kapitola 3.3.10 – Command buffers) za pomocí datového typu *float* čísly v rozsahu 0.0 až 1.0. Je nutné nastavit prioritu i pokud jde o jedinou frontu v bufferu.

Nejčastější typy fronty lze vidět v tabulce č. 2.

Název fronty	Popis
VkDeviceQueueCreateInfo	Fronta pro vytvoření logického zařízení
VkQueueSubmit	Fronta pro odeslání sekvence semaforů nebo příkazových bufferů do fronty
VkQueueWaitIdle	Fronta pro počkání na frontu

Tabulka 2: Fronty [12]

3.3.2.5 Fyzické zařízení

Po inicializaci Vulkan API je potřebné nalézt a zvolit grafickou kartu systému, která podporuje vyžadované funkce. Lze zvolit několik grafických karet a použít je současně.

3.3.2.6 Logické zařízení

Po vybrání fyzického zařízení je potřeba sestavit logické zařízení. Logické zařízení slouží jako rozhraní mezi aplikací a fyzickým zařízením.

3.3.3 Zobrazení

3.3.3.1 Povrch okna

Vulkan API z důvodu funkce cross-platform nemůže přímo pracovat se systémem oken. K propojení systému oken a Vulkanem pro zobrazení výsledku na obrazovku je nezbytné využit rozšíření WSI (Window System Integration). Nejběžněji se využívá rozšíření *VK_KHR_surface* na úrovni rozšíření instancí s datovým objektem *VkSurfaceKHR*, který reprezentuje abstraktní typ povrchu k zobrazení renderovaných obrázků.

Povrch okna je nutné vytvořit hned po vytvoření instance, jelikož může ovlivnit výběr fyzického zařízení. Pro vytvoření se využívá datová struktura *VkWin32SurfaceCreateInfoKHR* (v případě OS Windows). Datová struktura obsahuje dva důležité parametry: *hwnd* a *hinstance*, které se předávají oknu a procesu. Jakmile je datová struktura vyplněná můžeme vytvořit povrch pomocí funkce *vkCreateWin32SurfaceKHR* (v případě OS Windows), který se musí explicitně načíst.

Je na místě zmínit, že komponenta povrchu okna je zcela volitelná, pokud potřebujeme renderování mimo obrazovku. Vulkan dovoluje tuto techniku bez tzv. špinavého kódu jako je vytvoření neviditelného okna (v případě OpenGL).

Fronta zobrazení

Pro dokončení zobrazení se musí upravit vytváření logického zařízení, aby došlo k vytvoření fronty pro zobrazení a získání *VkQueue*.

3.3.3.2 Swap chain

Vulkan API nemá koncept výchozího framebufferu (viz kapitola 3.3.9 – Framebuffery), proto potřebuje infrastrukturu, která vlastní buffery, do kterých se bude renderovat před zobrazením na obrazovce. Tato infrastruktura je známá jako swap chain a musí být vytvořena explicitně. Swap chain je v podstatě fronta obrázků, které čekají na zobrazení na obrazovce. Aplikace získá obrázek k vykreslení, a poté ho vrátí do fronty. Jak přesně fronta funguje spolu s podmínkami pro zobrazení obrázku z fronty záleží na tom, jak je nastavený swap chain. Obecný důvod swap chainu je synchronizace zobrazení obrázků s obnovovací frekvencí obrazovky.

Ne všechny grafické karty jsou schopné zobrazení obrázků přímo na obrazovku, a to z několika důvodu. Například protože jsou navrhnuté pro servery a nemají žádný výstupní konektor. Za další, zobrazení obrázků je úzce spjaté se systémem oken a povrchy asociované s okny, která nejsou součástí jádra Vulkanu. Je tedy nutné povolit rozšíření zařízení *VK_KHR_swapchain*.

Kontrola dostupnosti swap chainu není dostatečná, neboť nemusí být kompatibilní s naším povrchem okna. Vytvoření swap chainu zahrnuje mnohem více nastavení než pouhá instance a vytvoření zařízení. Než bude možné pokračovat, je nutné získat další vlastnosti.

Jsou tři základní vlastnosti:

- Základní povrchové schopnosti (minimální/maximální počet obrázků ve swap chainu, minimální/maximální šířka a výška obrázků).
- Formát (pixelový, prostor barev).
- Dostupné zobrazovací módy.

Vhodné nastavení

Pokud je zjištěno, že podmínky byly splněné, pak je podpora dostatečná, ovšem může být mnoho různých módů nesoucí optimálnost. Pro zajištění nejlepšího možného swap chainu musíme zjistit tři typy nastavení:

- Formát (hloubka barev).
- Mód zobrazení (podmínky pro výměnu obrázků na obrazovce).
- Rozlišení výměny (rozlišení obrázků swap chainu).

Formát

Každý formát specifikuje barevné kanály, jejich typy a prostor barev. Například *VK_FORMAT_B8G8R8A8_UNORM* znamená, že obsahuje složku B, G, R a alfa kanály v tomto pořadí s datovým typem *unsigned integer* o velikosti 8 bitů, celkem tedy 32 bitů na pixel. Prostor barev indikuje, zda je dostupný prostor barev SRGB nebo musí použít flag *VK_COLOR_SPACE_SRGB_NONLINEAR_KHR*.

Je-li dostupný barevný prostor SRGB, je vhodné jej využít, výsledkem budou mnohem více přesné barvy. Práce se samotným barevným prostorem SRGB je ovšem náročnější.

Zobrazovací módy

Zobrazovací mód je nejdůležitější nastavení pro swap chain. Reprezentuje podmínky pro výměnu obrázků na obrazovku. Vulkan API nabízí čtyři dostupné módy:

- *VK_PRESENT_MODE_IMMEDIATE_KHR*
  - Odeslané obrázky jsou přesunuté ihned na obrazovku, čímž může dojít k trhání obrazu.
- *VK_PRESENT_MODE_FIFO_KHR*
  - Swap chain je fronta, kde obrazovka bere obrázek z fronty, když je obrazovka obnovena a program vloží renderovaný obrázky zpět do fronty.
  - V případě plné fronty program čeká.
  - Jde o velice podobnou techniku vertikální syncrhonizace v moderních hrách.
  - Moment, kdy je obrazovka obnovena je známý jako „vertikálně prázdný“.
- *VK_PRESENT_MODE_FIFO_RELAXED_KHR*
  - Liší se od předchozího, pokud je aplikace zpožděna a fronta byla prázdná v momentě obnovení obrazovky.
  - Místo čekání na další vertikálně prázdný moment, obrázek je přesunut ihned poté co dorazí.
  - Může dojít k viditelnému trhání obrazu.
- *VK_PRESENT_MODE_MAILBOX_KHR*
  - Další varianta druhého módu.
  - Místo blokování aplikace v případě plné fronty, obrázky, které byly ve frontě jsou nahrazeny novými.
  - Tento mód může být použit pro implementaci tzv. trojitého bufferingu (tripple buffering), který předchází trhání se značným snížením latence než standardní vertikální synchronizace, jenž používá dvojitý buffer (double buffering).

Rozlišení výměny

Ve většině případů je rozlišení výměny rovno rozlišení okna aplikace, do kterého kreslíme. Rozsah možných rozlišení je definován v struktuře *VkSurfaceCapabilitiesKHR*. Vulkan specifikace říká, aby se nastavila šířka a výška proměnné *currentExtent* na stejné hodnoty jako má okno aplikace.

Vytvoření

Struktura *VkSwapchainCreateInfoKHR* slouží pro vytvoření swap chainového objektu. Je nezbytné vyplnit proměnné:

- *surface*
  - Povrch swap chainu.
- *minImageCount*
  - Minimální počet obrázků.
- *imageFormat*
  - Obrázkový formát.
- *imageColorSpace*
  - Barevný prostor obrázků.
- *imageExtent*
  - Proporce obrázků.
- *imageArrayLayers*
  - Specifikuje počet vrstev každého obrázku.
  - Nabývá hodnotu 1, pokud se nevyvíjí stereoskopická 3D aplikace.
- *imageUsage*
  - Bitové pole sloužící k specifikaci druhu operací pro obrázky ve swap chainu.
- *queueFamilyIndexCount*
  - Počet vstupů v parametru *pQueueFamilyIndices.*
- *pQueueFamilyIndices*
  - Pointer na list rodinných front.
- *imageSharingMode*
  - Určuje zacházení s obrázky swap chainu, které budou použiti napříč několika rodinnými frontami.
  - To bude v případě, že rodina grafické fronty je rozdílná od zobrazovaní fronty.
  - S obrázky lze zacházet dvěma způsoby napříč rodinami.
    - *VK_SHARING_MODE_EXCLUSIVE*
      - Obrázek je vlastněn jednou rodinou frontou v čase a vlastnictví musí být explicitně převedeno před použitím v jiné rodinné frontě.
      - Tato možnost nabízí nejlepší výkon.

        - *VK_SHARING_MODE_CONCURRENT*
            - Obrázky můžou být použité napříč rodinnými frontami bez explicitního převedení vlastnictví.

Volitelné proměnné:

- *preTransform*
    - Slouží pro specifikaci transformace.
- *compositeAlpha*
    - Určuje, zda má být alfa kanál použít pro blednutí s ostatními okny v systému oken. Ve většině případu se ignoruje pomocí flagu *VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR*.
- *presentMode*
    - Určuje mód zobrazení.

Nyní je už jen potřeba definovat vlastní proměnnou typu *VkSwapchainKHR* a zavolat funkci *vkCreateSwapchainKHR* pro vytvoření swap chainu.

Obrázky swap chainu

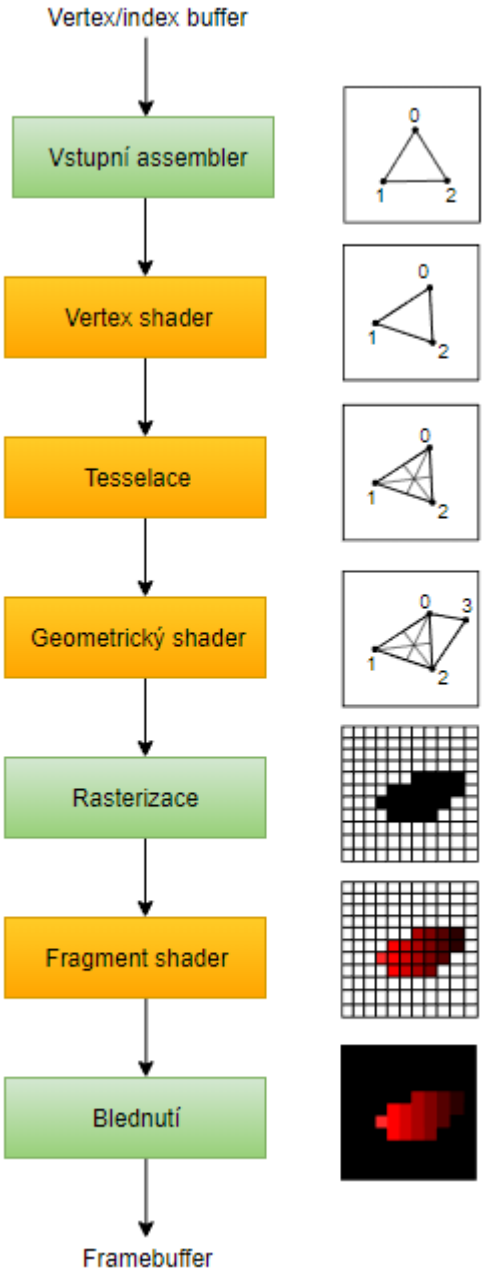
Po vytvoření swap chainu již zbývá získat *VkImage* objekty pro další manipulaci. Funkcí *vkGetSwapchainImagesKHR* nejprve získáme počet obrázků swap chainu s hodnotou *nullptr* pro argument *pSwapchainImage*, a poté je možné funkci opětovně zavolat s argumentem *pSwapchainImages* odkazující na pole *VkImage* v paměti.

3.3.3.3 Zobrazení obrázků

Pro použití jakéhokoliv *VkImage*, včetně ve swap chainu, v renderovací pipeline (viz kapitola 3.3.4 – Grafická pipeline) je potřeba vytvořit objekt *VkImageView*. Zobrazení obrázků je skutečnosti pohled na obrázek. Popisuje, jak přistupovat k obrázku a ke které části.

3.3.4 Grafická pipeline

Pipeline je konceptuální model popisující kroky grafického systému potřebné pro vykreslení 3D nebo 2D scény. Jakmile je vytvořen 3D model, grafická pipeline je proces, který promění 3D model na to, co počítač zobrazí. Pojem „pipeline“ je použit v podobném smyslu jako pipeline procesoru: individuální kroky pipeline běží paralelně, ale jsou blokovány, dokud nejpomalejší krok není dokončen.



Obrázek 5: Zjednodušený postup grafické pipeline [19]

Na obrázku č. 5 lze vidět zjednodušené kroky grafické pipeline. Žlutě vyznačené kroky jsou shadery (viz kapitola 3.3.5 – Shadery) a zeleně vyznačené jsou fixní funkce.

Fixní funkce provádějí:

- Vstupní assembler
  - Shromáždí vstupní data z vertex bufferu (buffer vrcholů), popřípadě index bufferu (indexový buffer), který slouží k zamezení duplicit vertex dat.
- Rasterizace
  - Přetváří primitiva do fragmentů. To jsou pixelové prvky vyplňující framebuffer.
  - Každý fragment mimo obrazovku je zahozen a výstupní atributy vertex shaderu jsou interpolovány mezi fragmenty, jak je vidět na obrázku č. 5.
  - Fragmenty skrývající se za dalšími fragmenty bývají běžně zahozeny díky testu hloubky.
- Blednutí
  - Provádí míchání fragmentů, který mapují stejný pixel ve framebufferu.
  - Fragmenty se mohou přepisovat, přidat nebo míchat na základě průhlednosti.

3.3.5 Shadery

Shadery popisují vlastnosti vertexů (vrcholů) nebo pixelů. Shadery jsou součástí grafické pipeline a lze je programovat. Tesselace se dělí na tři stavy, z toho dva jsou programovatelné, konkrétně: kontrolní a evaluační shader. Mezi těmito dvěma stavy je fixní stav nazývaný tesselátor.

Na rozdíl od předchozích grafických API je ve Vulkan API nutné kompilovat shadery a jsou nečitelné na rozdíl od zdrojových souborů jako GLSL a HLSL. Bytecode formát je nazýván SPIR-V, byl navrhnut pro použití ve Vulkan API a OpenCL (taktéž od Khronos Group). Výhodou tohoto bytecode formátu je, že kompilátory napsané GPU výrobci promění shaderový kód do nativního kódu, který je méně komplexní. Není nutné se obávat a Khronos vydal kompilátory pro GLSL a HLSL do SPIR-V.

3.3.5.1 Vertex Shader (VS)

Provede na každém vrcholu vstupní geometrie operace, nejčastěji transformace vrcholu. Lze provést např. simulaci pohybu vodní hladiny. [21]

3.3.5.2 Tessellation Control Shader (TCS)

Určuje, kolik tesselace má zpracovat (může pouze upravit aktuální patch data). Kontrolní shader je primárně zodpovědný za garantaci kontinuity mezi patchi. Tedy pokud jsou dva přilehlé patche, které potřebují jinou úroveň tesselace, kontrolní shader invokuje pro

rozdílné patche, aby použili vlastní kontrolní shader, čímž zajistí sdíleným okrajům mezi patchemi k použití stejné úrovně tesselace. Bez této ochrany by mohly vzniknout mezery nebo by mohlo dojít k porušení patchů.

Pro tesselaci je tento shader volitelný.

[22]

3.3.5.3 Tessellation Evaluation Shader (TES)

Poté, co tesselátor rozdělí patche na základě vstupu z TCS, provede evaluační shader výpočty vertexových hodnot pro každý generovaný vertex. [22]

3.3.5.4 Geometry Shader (GS)

Invokací geometrického shaderu se vezme jedna primitiva, výstupem může být žádná nebo více výstupních primitiv. Jsou pevně dané limity pro maximální výstup primitiv z jedné invokace GS.

V minulosti byl GS používán ke zvyšování množství výstupu z důvodu absence tesselace, dnes by šlo o drsnou formu implementace tesselačního shaderu.

Hlavní důvody využití GS:

- Vrstvené renderování: vykreslení jednoho primitiva do více obrázků bez nutnosti změny cíle renderu.
- Transformační zpětná vazba: často využívané pro výpočetní úlohy na GPU

GS je volitelný a nemusí být tedy použit.

[23]

3.3.5.5 Fragment Shader (FS)

Fragment shader je krok, který zpracovává fragment vytvořený krokem rasterizace. Každý fragment má svojí pozici v okně, ostatní hodnoty a obsahuje všechny interpolované výstupy pro každý vertex.

Výstupem FS je hloubková hodnota, žádné nebo více barevných hodnot k potencionálnímu zapsání do bufferu v současném framebufferu.

FS je technicky volitelný shader. Jestli-že nebude implementován, pak mají barevné hodnoty výstupního fragmentu nedefinované hodnoty.

[24]

3.3.5.6 Compute Shader (CS)

Výpočetní shader slouží k libovolným výpočetním úkonům. Zatímco může provádět renderovací funkci, je obecně využíván k úlohám, které nejsou spojené s vykreslováním trojúhelníků a pixelů. [25]

3.3.6 Fixní funkce

Starší grafické API měli výchozí stavy pro většinu kroků grafické pipeline. Ve Vulkan API musí být vše explicitně specifikováno. Od rozlišení scény po blednutí barev.

3.3.6.1 Vstupní vertex

Datová struktura *VkPipelineVertexInputStateCreateInfo* popisuje formát vertex dat, které budou předány do vertex shaderu. Lze je popsat dvěma způsoby:

- Bindování
  - Vzdálenost mezi daty a zda jsou data na vertex nebo na instanci.
- Popisy atributů
  - Typy atributů jsou předány do vertex shaderu.

3.3.6.2 Vstupní assembler

Datová struktura *VkPipelineInputAssemblyStateCreateInfo* popisuje dvě věci: jaký druh geometrie se bude vykreslovat z vertexů a má-li být povolen restart primitiv. Geometrie může mít typ topologie:

- *VK_PRIMITIVE_TOPOLOGY_POINT_LIST*
  - Body z vertexů.
- *VK_PRIMITIVE_TOPOLOGY_LINE_LIST*
  - Čára z každého druhého vertexu bez znovupoužití.
- *VK_PRIMITIVE_TOPOLOGY_LINE_STRIP*
  - Konec vertexu každé čáry je použit jako začátek další čáry.
- *VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST*
  - Trojúhelník z každých tří vertexů bez znovupoužití.
- *VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP*
  - Druhý a třetí vertex každého trojúhelníku je znovu použit jako první dva vertexy následujícího trojúhelníku.
- *VK_PRIMITIVE_TOPOLOGY_PATCH_LIST*
  - Patch primitiva.

3.3.6.3 Viewport a scissors

Viewport popisuje oblast framebufferu, do které bude výstup renderován. Lze nastavit počáteční body X a Y, šířka a výška, minimální a maximální hloubka v rozmezí 0 až 1. Viewport je definován datovou strukturou *VkViewport*.

Zatímco viewport definuje transformaci z obrázku do framebufferu, oblast scissors definuje, které pixely budou zachovány. Jakýkoliv pixel mimo oblast scissors bude zahozen rasterizační funkcí. Scissors fungují spíše jako filtr než jako transformace. Scissors jsou definovány datovou strukturou *VkRect2D*.

Viewport a scissors musí být zkombinovány do viewport stavu použitím datové struktury *VkPipelineViewportStateCreateInfo*. Je možné využit více viewportů, tak více scissors na některých grafických kartách.

3.3.6.4 Rasterizace

Jak již bylo zmíněno v kapitole 3.3.4 – Grafická pipeline, zabývá se proměnou primitiv do fragmentů. Rasterizace se dále zabývá depth testingem, face cullingem a testem scissors. Je možné nakonfigurovat rasterizaci na výstup fragmentů, aby vyplnili celý polygony nebo pouze hrany (wireframe rendering). Ke konfiguraci slouží datová struktura. *VkPipelineRasterizationStateCreateInfo*.

Zahození lze konfigurovat pomocí proměnné *rasterizerDiscardEnable*. Pokud je nastaveno na *true*, žádná geometrie neprojde rasterizační funkcí. Šířku hrany polygonu lze nastavit za pomocí proměnné *lineWidth*. Typ face cullingu lze nastavit proměnnou *cullMode* a pořadí vertexů pro stěnu proměnnou *frontFace*.

Nastavení generování fragmentů z geometrie lze nastavit pomocí proměnné *polygonMode*:

- *VK_POLYGON_MODE_FILL*
  - Vyplní polygon.
- *VK_POLYGON_MODE_LINE*
  - Vykresleny jsou pouze hrany polygonu.
- *VK_POLYGON_MODE_POINT*
  - Vertexy polygonu jsou vykresleny jako body.

3.3.6.5 Multisampling

Datová struktura *VkPipelineMultisampleStateCreateInfo* konfiguruje multisampling, což je jedna z možností k dosažení anti-aliasingu. Funguje na způsob kombinace výstupu několika polygonů z fragment shaderu, které rasterizují stejný pixel. Nastává to především u

okrajů, kde je možné zaznamenat artefakty. Pokud pouze jeden polygon mapuje pixel, je to méně náročnější kvůli nevyžadujícímu několikanásobnému spuštění fragment shaderu, než renderování vyššího rozlišení, které by se poté zmenšilo.

3.3.6.6 Depth test a stencil test

Je-li použit depth nebo stencil buffer, je nutné nakonfigurovat jeho použití datovou strukturou *VkPipelineDepthStencilStateCreateInfo*.

3.3.6.7 Blednutí

Existují dva typy struktur k definici blednutí. První z nich je *VkPipelineColorBlendAttachmentState* obsahující konfiguraci pro každý framebuffer. Druhá struktura *VkPipelineColorBlendStateCreateInfo* obsahuje globální nastavení blednutí.

V OpenGL šlo pomocí funkce *glBlendFunc* změnit blednutí pipeline. To ovšem ve Vulkan API nelze, a je proto nutné vytvořit pipeline zcela znova.

3.3.6.8 Dynamický stav

Limitovaný počet stavů v předchozích datových strukturách může být změněn bez nutnosti znovuvytvoření pipeline. Například velikost viewportu, šířka čáry a konstanta blednutí. K využití dynamických stavů je nutné vyplnit strukturu *VkPipelineDynamicStateCreateInfo*. Vyplněním hodnotami je bude nutné uvést v době renderování.

3.3.6.9 Pipeline layout

Ve shaderech lze využít tzv. uniform hodnoty, které se chovají podobně dynamickým stavům čili mohou být změněny v průběhu renderování bez nutnosti znovuvytvoření. Obecně se uniformy využívají pro předání transformačních matic do vertex shaderu nebo k předání texturových samplerů do fragment shaderu.

Uniform hodnoty musí být specifikované při vytváření pipeline, vytvořením objektu *VkPipelineLayout*.

3.3.7 Render pass

Před vytvořením pipeline je potřeba říci Vulkan API o tzv. framebuffer attachments, které budou použity v průběhu renderování. Specifikuje se, kolik barevných a depth bufferu bude, kolik samplů se má použít pro každý z nich a jak bude řízen jejich koncept po celou

dobu renderovacích operací. Všechny tyto informace jsou zabaleny v objektu *VkRenderPass*. Render pass se vytváří vyplněním struktury *VkRenderPassCreateInfo* a zavoláním funkce *vkCreateRenderPass*.

Framebuffer attachment lze popsat strukturou *VkAttachmentDescription*. Formát přílohy by se měl rovnat formátu obrázků swap chainu. Je nutné specifikovat proměnnou *loadOp* a *storeOp*. Proměnné definují, co provádět s daty v attachmentu před renderingem a po něm. Pro proměnnou *loadUp* je možné použít následující možnosti:

- *VK_ATTACHMENT_LOAD_OP_LOAD*
  - Zachová existující obsah attachmentu.
- *VK_ATTACHMENT_LOAD_OP_CLEAR*
  - Nastaví hodnoty na konstantu na počátku.
- *VK_ATTACHMENT_LOAD_OP_DONT_CARE*
  - Existující obsah je nedefinován, nezáleží nám na něm.

Pro proměnnou *storeOp* je možné použít následující možnosti:

- *VK_ATTACHMENT_STORE_OP_STORE*
  - Renderovaný obsah bude uložen do paměti, která může být přečtena později.
- *VK_ATTACHMENT_STORE_OP_DONT_CARE*
  - Obsah framebufferu bude nedefinovaný po renderu.

Textury a framebuffery reprezentovány objekty *VkImage* obsahují specifický formát, každopádně layout pixelů v paměti může být změněn na základě toho, co se snažíme s obrázkem dělat. Nejvíce používané layouty jsou:

- *VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL*
  - Obrázky použité jako barevná příloha.
- *VK_IMAGE_LAYOUT_PRESENT_SRC_KHR*
  - Obrázky pro zobrazení ve swap chainu.
- *VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL*
  - Obrázky sloužící jako cíl pro operaci kopírování paměti.

Proměnná *initialLayout* specifikuje, který layout obrázku bude použit, než začne render pass. Proměnná *finalLayout* specifikuje layout, ke kterému se automaticky přejde po dokončení render pass.

3.3.7.1 Subpassy a attachment reference

Samotný render pass může obsahovat několik subpassů. Subpassy jsou další operace záležející na obsahu framebufferu v předchozích render passech, např.: sekvence post-processing efektů, které jsou aplikovány po sobě. Pokud je skupina těchto renderovacích operací v jednom render passu, Vulkan je schopen přeřadit operace a ušetřit přenos paměti pro

lepší výkon. Subpass lze definovat strukturou *VkSubpassDescription*. Následující typy příloh lze odkazovat v subpassu:

- *pColorAttachments*
  - Attachmenty použité pro barvy.
- *pInputAttachments*
  - Attachmenty pro čtení v shaderu.
- *pResolveAttachments*
  - Attachmenty použité pro multisampling.
- *pDepthStencilAttachment*
  - Attachmenty použité pro depth a stencil data.
- *pPreserveAttachments*
  - Attachmenty, které nejsou použité v současném subpassu, ale jejich data se musí zachovat.

Reference attachmentu lze definovat strukturou *VkAttachmentReference*. Struktura specifikuje a odkazuje na referenci pomocí indexu. Layout specifikuje, který layout chceme, aby měl attachment v průběhu subpassu v této referenci.

3.3.8 Vytvoření grafické pipeline

Nyní lze zkombinovat všechny struktury a objekty z předchozích kapitol k vytvoření grafické pipeline. Pro rychlou rekapitulaci máme k dispozici typy objektů:

- Shader fáze
  - Shaderové moduly definují funkcionalitu programovatelných fází na grafické pipeline.
- Fixní funkce
  - Všechny struktury definující fixní funkce na pipeline jako: vstupní assembler, rasterizer, viewport a blednutí.
- Pipeline layout
  - Uniformy a push hodnoty v shaderu aktualizovatelné po dobu renderování.
- Render pass
  - Attachmenty ve fázích pipeline a jejich použitím.

Všechny tyto sktruktury kombinují funkcionalitu grafické pipeline. Nyní je možné vyplnit strukturu *VkGraphicsPipelineCreateInfo*. Vulkan dovoluje vytvořit novou grafickou pipeline z již existující. Myšlenka dědění spočívá v menší náročnosti sestavení pipeline se sdílenými funkcemi a rychlejší změnou pipeline v případě stejného rodiče. Výstupem bude objekt *VkPipeline* vytvořený funkcí *vkCreateGraphicsPipelines* s argumentem struktury *VkGraphicsPipelineCreateInfo*.

3.3.9 Framebuffery

Framebuffery představují kolekci specifických paměťových příloh využívané v instancích render pass. Jsou popsány strukturou *VkFramebuffer*.

Framebuffer objekt obsahuje reference všech *VkImageView* objektů reprezentující attachmenty. Vytváří se zavoláním funkce *vkCreateFramebuffer* obsahující argument s naplněnou strukturou *VkFramebufferCreateInfo* informacemi o velikosti, počtu vrstev, přílohách a příslušném render pass.

3.3.10 Command buffers

Operace sloužící pro renderování nebo přesun paměti nejsou prováděny voláním funkcí. Všechny operace, které chceme provést, se musí nahrát do objektů command bufferu (příkazový buffer). Výhoda tohoto těžkého nastavování renderovacích commandů je, že může být provedena v několika vláknech, poté už stačí jen říct Vulkanu, aby je provedl.

3.3.10.1 Command pools

Než lze začít vytvářet command buffer, je nutné vytvořit command pool (příkazový bazén). Command pool spravuje paměť, která je použita pro uchovávání bufferů, ze kterých jsou command buffery alokovány. Vytvoření command poolu se provádí zavoláním funkce *vkCreateCommandPool* s argumentem vyplněné struktury *VkCommandPoolCreateInfo* informacemi indexu rodiny a flagů.

Command buffery jsou prováděny odesláním na jednu z front zařízení stejně, jako byly grafické a zobrazovací fronty obdrženy. Každý command pool může alokovat command buffer, který je odeslán na jednotný typ fronty.

3.3.10.2 Alokace

Nyní je možné začít alokovat command buffery a nahrát na ně renderovací příkazy. Protože jedna z renderovacích operací zahrnuje bindování správného *VkFramebuffer* objektu, musíme nahrát command buffer pro každý obrázek ze swap chainu.

Command buffery jsou automaticky uvolněny poté, co je smazán command pool, proto je nemusíme explicitně mazat.

Alokace se provádí funkcí *vkAllocateCommandBuffers* s argumentem struktury *VkCommandBufferAllocateInfo* naplněnou informacemi command poolu, úroveň command bufferu a počet command bufferů. Úroveň specifikuje, zda jde o primární nebo sekundární typ:

- *VK_COMMAND_BUFFER_LEVEL_PRIMARY*
  - Může být odeslán do fronty po vykonání, ale nemůže být zavolán z ostatních command bufferů.
- *VK_COMMAND_BUFFER_LEVEL_SECONDARY*
  - Nemůže být odeslán do fronty přímo, ale může být zavolán z primárních command bufferů.

3.3.10.3 Nahrávání

Nahrávání probíhá zavoláním funkce *vkBeginCommandBuffer* s malou strukturou *VkCommandBufferBeginInfo* informující o nastavení skrze flag a dědění. Typy nastavení:

- *VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT*
  - Command buffer bude znovu nahrán hned po jeho vykonání.
- *VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT*
  - Jedná se o sekundární command buffer, který bude působit v jednom render passu.
- *VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT*
  - Command buffer může být přeposlán, pokud čeká na vykonání.

Ukončení nahrávání se provádí funkcí *vkEndCommandBuffer*, která vrací objekt *VkResult*.

3.3.10.4 Začátek render passu

Rendering začíná nahráním render passu funkcí *vkCmdBeginRenderPass*. Poslední parametr funkce specifikuje kontrolu renderovacích operací. Dělí se na:

- *VK_SUBPASS_CONTENTS_INLINE*
  - Operace render passu budou v primárním command bufferu a žádný sekundární command buffer nebude vykonán.
- *VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS*
  - Operace render passu budou vykonány ze sekundárních command bufferů.

Render pass je konfigurován parametry struktury *VkRenderPassBeginInfo* s informacemi o render passu, framebufferu, render oblasti a tzv. čistými hodnotami.

Ukončení render passu se provádí funkcí *vkCmdEndRenderPass*.

3.3.10.5 Základní vykreslovací příkazy

Pro základní vykreslení slouží dva jednoduché příkazy:

- vkCmdBindPipeline
  - Provede bindování grafické pipeline.
  - Specifikuje se argumentem, jestli se jedná o výpočetní nebo grafickou pipeline.

- vkCmdDraw
    - Jednoduchý příkaz pro vykreslení s parametry:
        - vertexCount
            - Počet vertexů.
        - instanceCount
            - Počet použitých instancí pro vykreslení.
        - firstVertex
            - Offset prvního vertexu v bufferu, definuje nejnižší hodnotu *gl_VertexIndex* ve vertex shaderu.
        - firstInstance
            - Offset pro instancované renderování, definuje nejnižší hodnotu *gl_InstanceIndex* v shaderech.

3.3.11 Renderování

3.3.11.1 Synchronizace

Pro renderování se musí provést následující operace:

- Získat obrázek ze swap chainu
- Provést command buffer s obrázkem jako attachment ve framebufferu
- Vrátit obrázek do swap chainu pro zobrazení

Každá z těchto událostí je nastavena za chodu, ale každá je vykonaná asynchronně. To je ovšem špatně, jelikož každá operace závisí na předchozí dokončené operaci.

Existují dva způsoby synchronizace swap chainových událostí: fences (zábradlí) a semafory. Oba dva typy mohou být použity pro koordinaci operací, kdy jedna operace slouží jako signál a druhá operace čeká na fence nebo semafor, ze kterého přejde z nesignalizovaného stavu na signalizovaný.

Rozdíl je v tom, že fence může být přístupný z volání programu pomocí funkce *vkWaitForFences*. Fences jsou primárně navrhnuty pro synchronizace aplikace s operací renderování, kdežto semafory jsou postaveny pro synchronizaci operací s nebo přes příkazové fronty. Jelikož chceme synchronizovat frontu operací renderování a zobrazení, je vhodnější použít semafory.

Pro základní renderování je potřeba jeden semafor k signalizaci získání obrázku a připravenosti k renderování a druhý k signalizaci dokončení renderování, aby mohlo nastat zobrazení. Semaforový objekt se definuje pomocí typu *VkSemaphore*. Vytváří se funkcí *vkCreateSemaphore* s argumentem struktury *VkSemaphoreCreateInfo*.

3.3.11.2 Získání obrázku ze swap chainu

Jelikož je swap chain rozšíření, je nutné u funkcí používat dodatek „KHR“. Tedy pro získání obrázku využijeme funkci *vkAcquireNextImageKHR* s argumenty zařízení, swap chainu, timeoutu, synchronizační objekt pro signalizaci zobrazovacího enginů a pointer do paměti na výstupní index swap chainového obrázku.

3.3.11.3 Odeslání příkazového bufferu

Zařazení do fronty a synchronizační nastavení je konfigurováno parametry struktury *VkSubmitInfo*. Odeslání do fronty probíhá příkazem *vkQueueSubmit* a vrací objekt *VkResult*.

3.3.11.4 Závislosti subpassu

Subpassy v render passu se automaticky starají o přechody obrázkového layoutu. Tyto přechody jsou ovládány závislostmi subpassu, které specifikují paměť a závislosti vykonání. Jsou dvě vestavěné závislosti, které se starají o přechody na začátku a na konci render passu.

Závislosti subpassu jsou specifikovány strukturou *VkSubpassDependency* s proměnnými: zdroj a cíl subpassu, zdroj a cíl stavové masky, zdroj a cíl přístupové masky, počet závislostí a odkaz na závislosti v paměti.

3.3.11.5 Zobrazení

Poslední krok renderování obrázku je odeslání a vrácení výsledku zpět do swap chainu k možnému zobrazení na obrazovce. Zobrazení je konfigurováno skrze strukturu *VkPresentInfoKHR* s informacemi o počtu čekacích semaforu a odkaz na semafory.

Dále je potřeba vyplnit strukturu *VkSwapchainKHR* informacemi o počtu swap chainu, pointeru na swap chainy, pointeru na obrázkové indexy a pointeru na výsledek (volitelný, pro případ kontroly výstupu). Tato struktura je společně se strukturou *VkPresentInfoKHR* použita jako argumenty pro funkci *vkQueuePresentKHR* jenž odešle požadavek k zobrazení obrázku do swap chainu.

3.3.11.6 Snímky v letu

V případě, že je CPU rychlejší než GPU, pak je možné sledovat vzrůstající spotřebu paměti. Děje se to kvůli tomu, že CPU odesílá rychleji požadavky na GPU, které je nestíhá zpracovávat. Jednoduchá funkce, která řeší čekání na dokončení před odesláním dalšího požadavku, se nazývá *vkQueueWaitIdle*.

3.4 Metody renderingu 3D terénu

Renderování terénu obsahuje velké množství metod, ať už se jedná o reálný svět nebo imaginární. Mezi nejběžnější renderování terénu patří povrch země.

Renderování terénu se dělí na dva druhy:

- Perspektivní projekce
    - Projekce vychází ze společného bodu.
- Ortografická projekce
    - Projekce seshora dolů.

Terénní algoritmy se dělí do kategorií:

- Diskrétní LOD
- Pokročilé LOD
    - Gridově založený.
        - Quadtree
        - ROAM
    - TIN
        - Progresivní síť
    - Voxelově založený.

Bylo provedeno hodně výzkumu a vývoje v oblasti systémů pro rendering terénu. Nejlepší systémy dokáží vyrenderovat svět, ve kterém prohlížející může procházet plynule od individuálních kamenů a kytek po prohlížení celé planety z vesmíru. Perspektiva, umění a geometrické chyby v průběhu času zůstávají nezměněny, ale hardwarové architektury a dostupné zdroje se změnily dramaticky. Znamená to, že dříve používané techniky jsou dnes zastaralé. Naštěstí poslední generace hardwaru se posunula směrem, kde momentálně preferované implementace jsou snadnější, než bylo doposud.

[26]

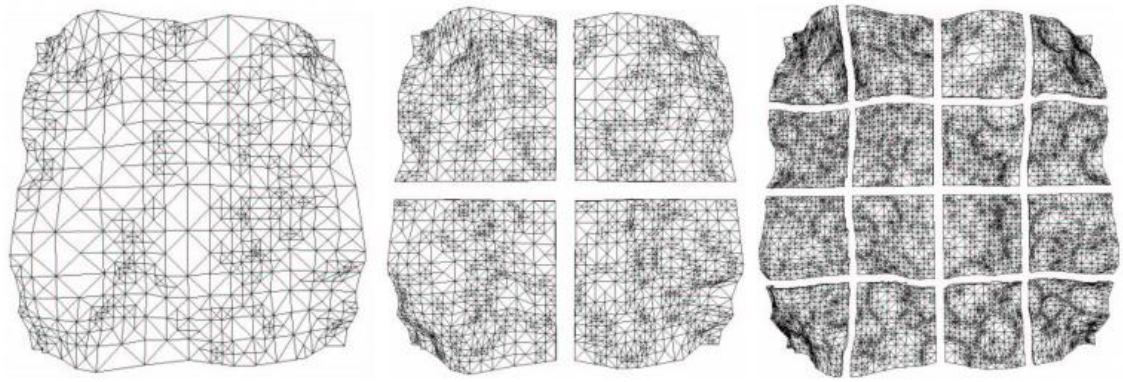
3.4.1 Level of Detail (LOD)

Neboli úrovně detailů (level of detail) je typ renderování terénu. Tento typ byl důležitý pro aplikace jako letecké simulátory, počítačové hry a geografické informační systémy. Geometrie byla více konzistentní a byly vytvořeny více specializované a jednodušší algoritmy.

3.4.1.1 Chunked LoD

Chunked Level of Detail je algoritmus snižující vytížení CPU a zachovává geometrii konzistentní (zůstává velký počet trojúhelníků). Má ovšem své nevýhody:

- Náročný preprocessing.
- Data musí být statická.
- Používá vyšší počet trojúhelníků než primitivní algoritmus.



Obrázek 6: První tři fáze Chunked LOD [27]

Na obrázku č. 6 je možné vidět první tři fáze Chunked LOD. V první fázi máme základní síť. Ve druhé fázi je síť rozdělená na čtyři části a je možné vidět více trojúhelníků čili více detailů. V poslední fázi je síť rozdělená již na 16 chunků (dílů) s maximálním detailem.

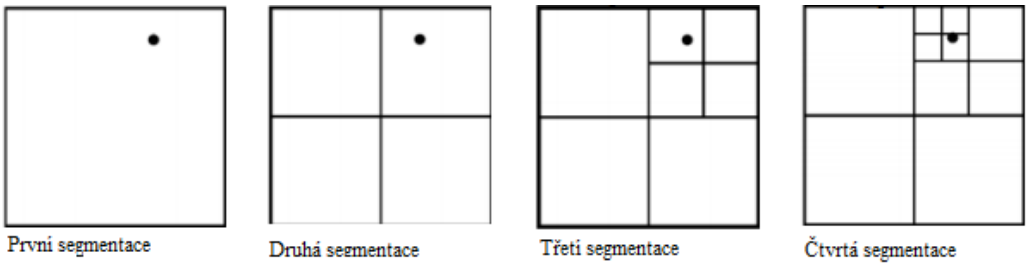
[27]

3.4.1.2 Quadtree

Algoritm Quadtree navrhl Petr Lindstrom. Je založen na myšlence rozdělení terénu do malých nekonzistentních částí (Chunked LoD) za předpokladu, že uživatel bude akceptovat ztrátu kvality. Dále šlo o zachování počtu trojúhelníků potřebných pro renderování v rozumné velikosti.

V procesu rozdělování, Quadtree struktura zachovává výsledky segmentace a každý uzel Quadtree reprezentuje blok terénu. Nejvyšší uzel reprezentuje celý datový blok, uzly reprezentují terénní bloky pro přímé renderování než rozdělení a prostřední uzel bez poduzlů, reprezentuje terénní blok, který pokračuje v segmentování, protože nesplňuje pravidla přesnosti. Princip rozdělování terénu pomocí Quadtree je reprezentován na obrázku č. 7.

[28]



Obrázek 7: Segmentace algoritmu quadtree [28]

3.4.1.3 ROAM

Real-time optimally adapting mesh (ROAM) je pokročilá technika LOD algoritmu, která optimalizuje terénní síť. Mezi hlavní klíčové koncepty patří:

- Výškové pole.
- Binární trojúhelníkový strom.
- Operace rozdělení a spojení.

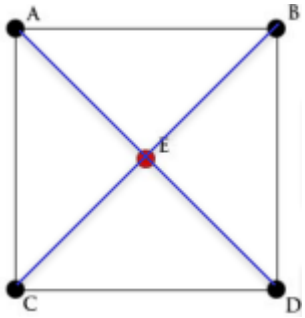
Ostatní důležité koncepty:

- Dvě fronty priority.
  - Pro rozdělení
  - Pro sloučení
- Chybová metrika pro rozdělení a spojení.

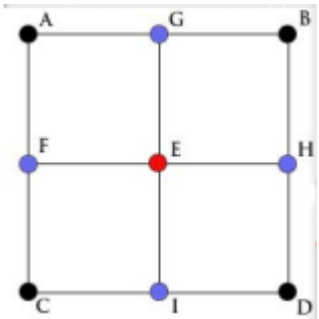
[29]

3.4.1.4 Diamond-square

Diamond-square algoritm také nazývaný mrakový fraktál, plasma fraktál nebo random midpoint displacement. Nejlépe funguje, pokud běží na čtvercovém gridu o šířce 2^n. Algoritmus začíná jako grid o velikosti 2x2. Následuje diamantový krok viz obrázek č. 8 a poté čtvercový krok viz obrázek č. 9. Kroky diamantový a čtvercový se opakují, dokud se nedosáhne definované šířky gridu. [29]



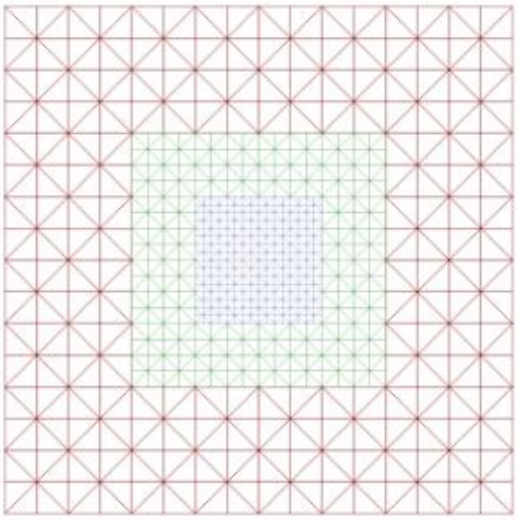
Obrázek 8: Diamantový krok [29]



Obrázek 9: Čtvercový krok [29]

3.4.1.5 Geoclipmapping

Geoclipmappingový algoritmus využívá sadu čtverců ve tvaru prstence, kde každý prstenec je dvakrát větší než předchozí a má poloviční velikost viz obrázek č. 10.



Obrázek 10: Geoclipmapping [30]

Výsledkem je konzistentní rozlišení terénu z jakékoliv dálky. Nejvíce vnitřní prstenec (s největším rozlišením) má vyplněný střed a stává se z něj jednoduchými čtvercový grid trojúhelníků. Geometrie se opakuje podle gridového vzorce, čímž dostává skvělou vlastnost. Je možné posunout přesně násobky velikosti gridu bez viditelné změny na straně uživatele kromě náznaku posunutí okrajů. Tato vlastnost nám tedy dává možnost posunout geometrii kolem středu kamery bez znatelného posunutí.

[30]

3.4.2 Moderní rendering terénu

Moderní rendering terénu vyžaduje grafickou kartu podporující OpenGL 4.0+, DirectX 10+, Vulkan API nebo Metal API. Tyto API obsahují klíčovou funkci, kterou je tesselace, díky které se výpočet LOD neprovádí na straně CPU, ale na straně GPU, a je tedy několikanásobně rychlejší. Nejvíce se využívá algoritmus geoclipmappingu, který nevyžaduje explicitní aktualizování a dále má vysoké rozlišení kolem středu kamery.

Vertex shader provádí dvě hlavní transformace sítě:

- Transformace sítě pro udržení středu kolem kamery.
- Pozměňuje výšku pro dosažení výšky terénu.

Dochází k tesselování gridu jako běžné čtverce, které se samostatně dělí do trojúhelníků. Implementace se mohou lišit a může docházek k rozdělování sítě do kvadrantů nebo oktáv.

[26]

3.5 Mapové podklady

Pro renderování terénu na základě mapových podkladů je potřeba získat vstupní data. Těmito daty se myslí body obsahující informace zeměpisné šířky, délky a nadmořské výšky.

Data lze získat skrze webové služby. Některé jsou zdarma, jiné jsou placené. Zeměpisnou délku a šířku je možné získat zdarma z mapových portálů jako je např.: Google Maps, mapy.cz a podobné. Nadmořskou výšku lze získat hůře, je pochopitelně možné využít zdarma webové služby jako:

- Open Elevation (https://open-elevation.com/)
- Elevation API (https://elevation-api.io/)
- Bing Elevation API

Webové služby pro nadmořskou výšku mají své limity a mezi ně se řadí počet požadavků a především rozlišení. Rozlišení značí maximální vzdálenost mezi body, ze kterých je hodnota interpolováno. Např.: s rozlišením 90 m se zjistí v bodě A nadmořská výška 1 metr, v bodě B vzdálený od bodu A 10 metrů se zjistí opět nadmořská výška 1 metr, v bodě C vzdálený od bodu A 91 metrů se zjistí již rozdílná nadmořská výška 3 metry a vznikne dvoumetrový skok. [34]

Placené webové služby dosahují rozlišení až 1 metr, je zde však nutné podotknout, že se to ve většině případů týká polohy USA.

3.5.1 Srovnání

V tabulce č. 3 je možné vidět srovnání dostupných API pro získání nadmořské výšky. Nejlépe vychází Bing Elevation API v poměru rozlišení/max. počet poloh/zdarma.

Název služby	Rozlišení	Max. počet poloh v požadavku	Zdarma
Google Elevation API	1 m[1]	512	✗
Bing Elevation API	10 m[2]	1024	✓
Open Elevation	10 m	bez limitu	✓
Elevation API	1 km[3]	10	✓

Tabulka 3: Srovnání mapových API (Zdroj: vlastní)

[1] Platí pro vybrané polohy
[2] pro USA, 90 m pro 56° S - 60° J, zbytek 900 m
[3] Rozlišení 90 m $0.0001/požadavek, 30 m $0.0005/požadavek (dostupné mezi šířkou 52 a -60)

3.6 3D tisk

K tisku na 3D tiskárně je nezbytné získat tzv. *gcode*. Jedná se o kód s instrukcemi pro pohyb jejími částmi. *Gcode* lze vygenerovat z programů jako je např.: Slic3r. Pro vygenerování *gcode* bude potřebovat 3D model. Nejběžnější a zároveň nejvíce podobný strukturou pro renderování 3D grafiky je souborový formát STL. Tento souborový formát obsahuje název modelu a facety tvořené třemi vertexy. Formát STL lze zapsat dvěma způsoby:

- ASCII
- Binárně

Na obrázku č. 11 je možné vidět příklad zápisu STL souboru způsobem ASCII.

```
solid name
facet normal ni nj nk
    outer loop
        vertex v1x v1y v1z
        vertex v2x v2y v2z
        vertex v3x v3y v3z
    endloop
endfacet
endsolid name
```

Obrázek 11: Příklad STL souboru (Zdroj: vlastní)

[35]

3.7 Shrnutí

Vulkan API je nové grafické API, které nahrazuje předchůdce OpenGL a je stejně jako předchůdce dostupný na velkém množství platforem, kde tato vlastnost schází DirectX a Metal API. Pro využití Vulkan API bylo zjištěno, že nejvhodnější je programovací jazyk C++.

V současné době se pro renderování terénu využívají především shaderové programy. Mezi nejpopulárnější algoritmus patří Geoclipmapping skládající se ze sady prstencových čtverců.

Volně dostupné mapové podklady (získání nadmořské výšky) v podobě API je aktuálně velmi omezený a roli hraje hlavní parametr, čímž je rozlišení. Nejvhodnější současně zdarma dostupnou API je Bing Elevation API.

4 Vlastní práce

Implementace aplikace pro ověření hlavního cíle bude prováděna v programovacím jazyce C++ kvůli nespočetným výhodám popsaných v kapitole 3.2 Programovací jazyk C++ doplní knihovny Qt zejména pro řešení GUI. Aplikace bude koncipována, aby ji bylo možné ji spustit na co nejvíce možných zařízeních, tedy nebude využívat žádný kód zaměřený na daný OS.

4.1 Požadavky

Pro implementaci aplikace budou potřeba následující položky:

- C++ kompilátor
- Vývojové IDE
- Qt
- Vulkan SDK
- API pro nadmořskou výšku

Kompilátor je dostupný ve vývojovém IDE Visual Studio. Konkrétně verze 2017 podporuje nejnovější C++17 funkce. Verze 2017 je dostupná zdarma v podobě komunitní verze nebo v případě studenta lze získat ultimátní edici. Pro účel kompilace postačí komunitní verze.

Qt společně s knihovnami nabízí vlastní vývojové IDE nazývané Qt Creator. Má nativní podporu Qt knihoven, které by se musely v IDE Visual Studio přidat pomocí pluginu. K vývoji aplikace bude využito IDE - Qt Creator.

Vulkan SDK lze získat pro platformu Windows a Linux od společnosti LunarG, která se specializuje na vývojové ovladače pro grafické karty. SDK obsahuje všechny nezbytně nutné komponenty pro vývoj aplikací založené na Vulkan API.

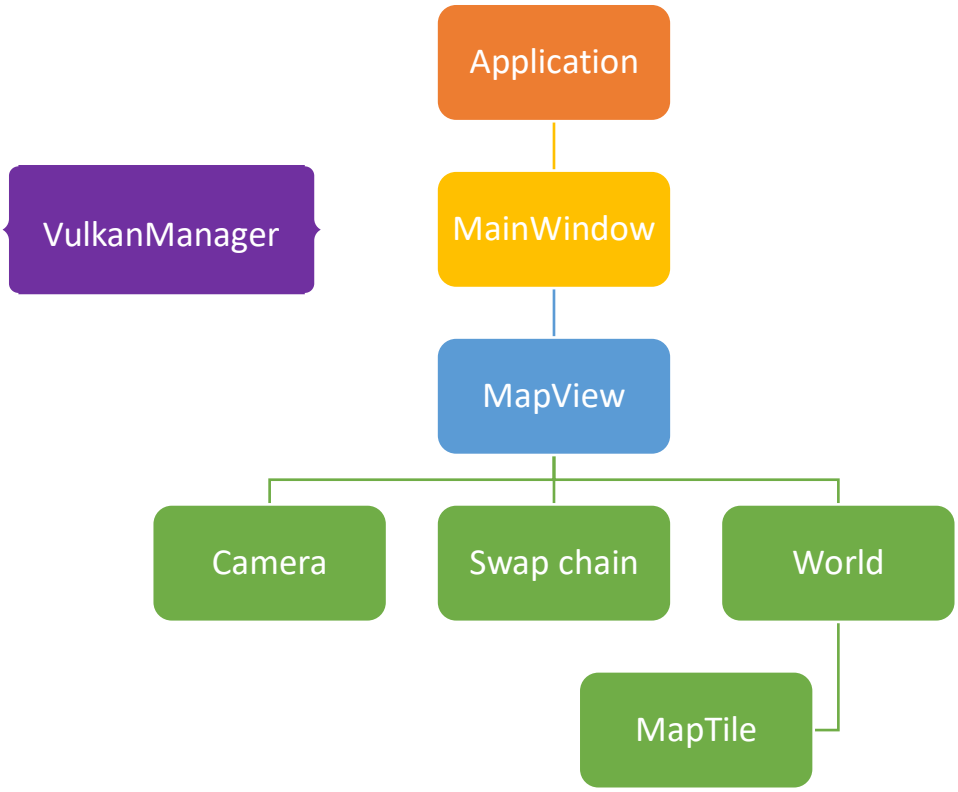
Z tabulky v kapitole 3.5 je možné zjistit, že Bing Elevation API je jednoznačně nejvhodnější služba v poměru rozlišení/počet požadavků/zdarma.

4.2 Infrastruktura aplikace

Před samotným začátkem implementace je vhodné si sestavit infrastrukturu aplikace. Diagram infrastruktury aplikace je zobrazen na obrázku č. 12. Jednotlivé objekty diagramu obrázku č. 12 budou provádět činnosti:

- Application
    - Správa základního nastavení aplikace a zajištění zobrazení hlavního okna *MainWindow*.

- VulkanManager
  - Jediný objekt daného typu v celém životním cyklu aplikace (Singleton).
  - Spravuje objekty pro práci s Vulkan API (*VkDevice*, *VkPhysicalDevice*, …).
  - Obsahuje funkce pro vytvoření bufferů a shader modulů.
- MainWindow
  - Spravuje hlavní okno aplikace a její ovládací prvky jako:
    - Menu
    - Vulkan okno
    - Statusbar
- MapView
  - Spravuje objekty pro renderování scény a pohybu uvnitř scény, tedy implementace klávesových událostí a myši pro ovládání kamery.
- World
  - Spravuje objekty pro renderování světu.
- MapTile
  - Správa dat o terénu a jeho renderovacího nastavení



Obrázek 12: Infrastruktura aplikace (Zdroj: vlastní)

4.3 Implementace infrastruktury

4.3.1 Application

Třída *Application* dědí z třídy *QApplication*, jenž spravuje tok aplikace a nastavení. Implementace třídy *Application* lze vidět na zdrojovém kódu č. 1 společně s komentářem.

```
1.  class Application : public QApplication
2.  {
3.      Q_OBJECT
4.
5.  public:
6.      Application(int& argc, char**); // Nastavení Vulkan instance a zobrazení MainWin
    dow
7.      ~Application();
8.
9.      QVulkanInstance* getVkInstance() const { return vulkanInstance; }
10.
11. private:
12.     MainWindow* mainWindow = Q_NULLPTR; // Hlavní okno aplikace
13.
14.     QVulkanInstance* vulkanInstance = Q_NULLPTR; // Instance Vulkan API
15. };
```

Zdrojový kód 1: Hlavičkový soubor třídy Application (Zdroj: vlastní)

Funkce konstruktoru nastaví validační vrstvu proměnné vulkanInstance na standard *VK_LAYER_LUNARG_standard_validation*. Provede se nastavení na verzi Vulkan API – 1.1.74.

4.3.2 MainWindow

MainWindow třída dědí z třídy *QMainWindow*. Implementuje se konstruktor a funkce *updateStatusBar* sloužící pro debug hodnot instance *MapView*.

```
1.  class MainWindow : public QMainWindow
2.  {
3.      Q_OBJECT
4.
5.  public:
6.      MainWindow(QVulkanInstance* vulkanInstance, QWidget* parent = Q_NULLPTR); // Vyt
    voření instance MapView a připojení signálů
7.      ~MainWindow();
8.
9.  public slots:
10.     void loadFromFile(); // Načítání mapy ze souboru
11.     void loadFromMap(); // Načítání mapy z Bing Elevation API
12.
13.     void updateStatusBar(); // Aktualizace Statusbaru
14.
15. private:
16.     Ui::MainWindowClass ui; // Instance GUI pro MainWindow
17.
18.     MapView* mapView = Q_NULLPTR; // Instance MapView
19. };
```

Zdrojový kód 2: Hlavičkový soubor třídy MainWindow (Zdroj: vlastní)

Konstruktor vytváří okno *MapView*, které je typu *QWindow*, pro nastavení podokna je nutné použít *QWidget*, čili se dále vytvoří wrapper pro *MapView*, který umožní být nastaven jako podokno.

Funkce *loadFromFile* a *loadFromMap* implementují uživatelské akce, tedy zobrazení uživatelského dialogu pro výběr souboru v případě *loadFromFile* nebo zadání souřadnic počátku a konce pro funkci *loadFromMap*.

Funkce *loadFromMap* po vyplnění souřadnic provede výpočet bodů a hromadně pošle dotaz na Bing Elevation API pro získání nadmořských výšek pro dané body. Z vrácených dat se vytvoří obrázek o velikosti 1024x1024 pixelů s formátem RGB o velikosti 8 bitů pro každý barevný kanál. Po vytvoření obrázku k převodu na texturu ve formátu KTX.

Hlavičkový soubor *MainWindow* lze vidět na zdrojovém kódu č. 2.

4.3.3 MapView

MapView třída dědí z třídy *VulkanWindow*, která obsahuje implementaci volání renderování a swap chainu. Hlavičkový zdrojový kód lze vidět na zdrojovém kódu č. 3.

Funkce lze popsat následujícím způsobem:

- load
  - Ověří existenci souboru a předá jej instanci třídy *World*.
- mouseMoveEvent
  - Provádí rotaci kamery na základě pohybu kurzoru, pokud je stisknuto tlačítko myši.
- buildCommandBuffers
  - Zavolání funkce *buildCommandBuffers* v instanci třídy *World*.
- initializeResources
  - Nastavení perspektivy kamery a vytvoření světu.
- releaseResources
  - Zničení objektů instance třídy *World*.
- render
  - Příprava snímku v rodičovské třídě *VulkanWindow*.
  - Zavolání funkce update, kde dochází k pohybu kamery.
  - Zavolání funkce *draw* instance třídy *World*.
  - Odeslání fronty s command bufferem.
  - Odeslání snímku v rodičovské třídě *VulkanWindow*.

```
class MapView : public VulkanWindow
{
    Q_OBJECT

public:
    MapView(QVulkanInstance* vulkanInstance); // Nastavení instance a rychlosti kame
ry
    ~MapView();

    void load(const QString& fileName); // Načtení terénu ze souboru

signals:
    void updated(); // Signalizace, že byl snímek aktualizován

private:
    void mousePressEvent(QMouseEvent*) override; // Zaznamenání stisknutí tlačítek m
yši
    void mouseReleaseEvent(QMouseEvent*) override; // Zaznamenání uvolnění tlačítek
myši
    void mouseMoveEvent(QMouseEvent*) override; // Zaznamenání pohybu myši
    void keyPressEvent(QKeyEvent*) override; // Zaznamenání stisknutí klávesy
    void keyReleaseEvent(QKeyEvent*) override; // Zaznamenání uvolnění klávesy

    void buildCommandBuffers() override; // Sestavení příkazových bufferů
    void initializeResources() override; // Inicializace zdrojů
    void releaseResources() override; // Uvolnění zdrojů
    void render() override; // Vykreslení
    void update(); // Aktualizace kamery

    World* world; // Instance světu
};
```

Zdrojový kód 3: Hlavičkový soubor třídy MapView (Zdroj: vlastní)

4.3.3.1 VulkanWindow

Qt nabízí implementaci Vulkan okna v podobě třídy *QVulkanWindow*, stará se především o swap chain, command pool a grafické fronty. *QVulkanWindow* bylo implementované s verzí 5.11 a není ještě zcela optimální pro náročnější renderování. *VulkanWindow* tedy představuje implementaci děděním *QWindow* pro vlastní řízení swap chainu a podobně.

Konstruktor okna nastaví správnou plochu, tedy na Vulkan a předá instanci Vulkanu.

Reimplementují se funkce:

- exposeEvent
  - Dochází k sestavení okna voláním funkcí *initialize*, *createCommandPool*, *setupSwapChain* atd.
- resizeEvent
  - Zajišťuje získání nových proporcí okna.
- event
  - Zajišťuje v případě požadavku na aktualizaci renderování a zobrazení snímku na okno funkcí *renderFrame*.

Vlastní funkce třídy zajišťují vytvoření command poolu, command bufferů, synchronizačních signálů, swap chainu, framebufferů, render passu a funkce pro uvolnění. Funkce pro vytváření nezbytných Vulkan objektů se skládají z:

- createCommandPool
  - Alokace command poolu s flagem *VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT*
  - Flag umožní reset individuálního command bufferu.
- createCommandBuffers
  - Alokace command bufferů podle počtu obrázků ve swap chainu.
- createSynchronizationPrimitives
  - Alokace synchronizačních primitiv typu *VkFence* o velikosti command poolu.
- setupSwapChain
  - Inicializace swap chainu v instanci třídy *SwapChain.*
- setupDepthStencil
  - Vytvoření obrázku naplněním struktury *VkImageCreateInfo.*
  - Alokace lokální paměti pro obrázek skrze strukturu. *VkMemoryAllocateInfo*
  - Nabindování obrázku funkcí *vkBindImageMemory*.
  - Vytvoření pohledu obrázku funkcí *vkCreateImageView.*
- setupFramebuffer
  - Vytvoření framebuffer objektů o velikosti command poolu.
  - Pro alokaci se využívá depth stencil *VkImageView* z funkce *setupDepthStencil.*
- setupRenderPass
  - Použití referenčních attachmentů.
    - Barevná reference ze swap chainu
    - Depth reference
  - Nastavení dvou layout přechodů skrze struktury *VkSubpassDependency*.
  - Alokace render passu funkcí *vkCreateRenderPass* s referenčními attachmenty a přechody.

Renderovací funkce se skládají z:

- prepareFrame
  - Příprava snímku získáním obrázků ze *SwapChain* třídy funkcí *acquireNextImage.*
  - V případě nekompability dojde k znovuvytvoření swap chainu.
- renderFrame
  - V případě, že jsou všechny požadavky splněny, proběhne zavolání funkce *render* **(**přetížena v třídě *MapView***).**
  - Zavolání funkce *QWindow::requestUpdate,* která invokuje funkci *event.*

- submitFrame
    - Odeslání snímku do fronty *SwapChainu* funkcí *queuePresent* a vyčkání na dokončení.
    - V případě nekompability dojde k znovuvytvoření swap chainu.
- render
    - Implementuje se v třídě *MapView* přetížením funkce

4.3.3.2 SwapChain

Třída *SwapChain* implementuje funkce, které se starají o proměnou *VkSwapchainKHR*. Dostupné funkce lze popsat:

- acquireNextImage
    - Získání následujícího obrázku od Vulkan API.
- connect
    - Získání rozšiřujících funkcí od Vulkan API.
- create
    - Vytvoření swap chainu.
    - Získání obrázku.
    - Alokace obrázkového pohledu.
    - Snaží se najít nejvhodnější zobrazovací mód.
- initSurface
    - Získání dostupných front rodin pro zajištění dostupnosti všech funkcí.
    - Získání swap chain formátu na základě fyzického zařízení.
- queuePresent
    - Zobrazení fronty skrze strukturu *VkPresentInfoKHR*.

4.3.4 World

Třída *World* implementuje základní funkce pro objekty světu. Hlavičkový soubor lze vidět na zdrojovém kódu č. 4.

Konstruktor neimplementuje vytváření objektů a destruktor neimplementuje jejich zničení pro případ, že by se chtěla načíst jiná terénní data. Funkce jsou velice jednoduché a popisují:

- buildCommandBuffers
    - U všech objektů třídy zavolá funkci na sestavení command bufferů.
- create/destroy
    - Vytvoření/destrukce objektů.
- draw
    - Renderování objektů.
- loadTerrain
    - Předání názvu souboru objektu *MapTile*.

```
1.  class World
2.  {
3.  public:
4.      World();
5.
6.      void buildCommandBuffers(); // Vytvoření command bufferů
7.
8.      void create(); // Vytvoření světu
9.      void destroy(); // Smazání objektů
10.
11.     void draw(const Camera& camera); // Vykreslení světu
12.
13.     void loadTerrain(const QString& fileName); // Načtení terénu
14.
15. private:
16.     MapTile tile;
17.
18.     Frustum frustum;
19. };
```

Zdrojový kód 4: Hlavičkový soubor třídy World (Zdroj: vlastní)

4.3.5 MapTile

Ve třídě *MapTile* se odehrává celé renderování terénu, respektive sestavení command bufferů, které provádějí renderovací operace.

Implementuje se několik veřejných funkcí:

- buildCommandBuffers (zjednodušený zdrojový kód č. 5)
    - Sestavení výchozích hodnot (barva, hloubka) strukturou *VkClearValue*.
    - Nastavení render pass strukturou *VkRenderPassBeginInfo*.
    - Získání command bufferů a framebufferů.
    - Pro každý command buffer.
        - Nastaví framebuffer dle indexu.
        - Začne nahrávat operace do command bufferu funkcí *vkBeginCommandBuffer*.
        - Zahájení render passu funkcí *vkCmdBeginRenderPass*.
        - Nastavení viewportu a scissors.
        - Bindování pipeline funkcí *vkCmdBindPipeline* s typem *VK_PIPELINE_BIND_POINT_GRAPHICS*.
        - Bindování descriptorů funkcí *vkCmdBindDescriptorSets*.
        - Bindování vertex bufferu funkcí *vkCmdBindVertexBuffers*.
        - Bindování index bufferu funkcí *vkCmdBindIndexBuffer*.
        - Vykreslení indexů funkcí *vkCmdDrawIndexed*.
        - Ukončení render passu funkcí *vkCmdEndRenderPass*.
        - Ukončení nahrávání command bufferu funkcí *vkEndCommandBuffer*
- create
    - Načtení terénních dat ze souboru.
    - Vytvoření vertex bufferů, uniform bufferů, pipeline a descriptorů.
    - Zavolání funkce *buildCommandBuffers*.

- destroy
    - Zničení objektů.
- draw
    - Aktualizace uniform bufferů.
- load
    - Načtení terénních dat ze souboru.
    - Aktualizace vertex bufferů.

Mezi privátní funkce patří (všechny jsou volány z funkce *create*):

- createDescriptorPool
    - Vytvoření dvou descriptor poolů funkcí *vkCreateDescriptorPool*.
        - Pro uniform buffery.
        - Pro textury.
    - Funguje na stejném principu jako command pool s rozdílem, že uchovává uniform buffery nebo textury.
- createDescriptorSetLayouts
    - Vytvoření dvou descriptor set layoutu funkcí *vkCreateDescriptorSetLayout*.
        - Sdílený uniform buffer pro tesselační shadery
        - Výškova mapa v podobě textury
    - Vytvoření descriptor layoutu obsahující reference na set layouty.
        - Popisuje typy bindů a z jakého shaderu jsou dostupné
    - Vytvoření pipeline layoutu funkcí *vkCreatePipelineLayout* se strukturou *VkPipelineLayoutCreateInfo*, která má referenci na descriptor layout.
- createDescriptSets
    - Popisuje, kde v paměti najít data pro daný bind.
    - Struktura *VkDescriptorSetAllocateInfo* obsahuje:
        - Descriptor pool.
        - Descriptor set layout.
    - Alokace descriptor setu funkcí *vkAllocateDescriptorSets*.
    - Aktualizace descriptor setů (odkázání na správné místo v paměti):
        - Uniform buffer pro tesselační shadery se nachází v proměnné *uniformTessellationBuffer.descriptor*.
        - Výšková mapa se nachází v proměnné *textures.heightMap.descriptor*.

- createPipeline
    - Využití dynamických stavů strukturou *VkDynamicState* pro viewport, scisscors a šířku čáry.
    - Nastavení vstupního assembleru na geometrii *VK_PRIMITIVE_TOPOLOGY_PATCH_LIST*.
    - Nastavení rasterizace na:
        - Výplň polygonů (*VK_POLYGON_MODE_FILL*).
        - Cull mode na zahození zpětně orientovaných trojúhelníků (*VK_CULL_MODE_BACK_BIT*).
        - Front face specifikuje, že trojúhelník s kladnou plochou je považován za čelní (*VK_FRONT_FACE_COUNTER_CLOCKWISE*).
    - Zakázání blednutí.
    - Nastavení depth testingu.
    - Zakázání multisamplingu.
    - Popsání vertex attributů strukturou *VkVertexInputAttributeDescription*.
        - Pozice je 32 bitová typu float (*VK_FORMAT_R32G32B32_SFLOAT*).
    - Použití shaderů strukturu *VkPipelineShaderStageCreateInfo*.
    - Vytvoření grafické pipeline funkcí *vkCreateGraphicsPipelines*.
- createUniformBuffers
    - Vytvoření uniform bufferů.
    - Probíhá skrze funkci definovanou Singletonu *VulkanManager – createBuffer*.
- createVertexBuffers
    - Počet vertexů je stanoven na 4096.
    - Pro každý vertex je vypočítána pozice a přidělena nadmořská výška ze souboru výškové mapy.
    - Vytvoření index bufferu, aby nedocházel k duplicitám vertexů.
    - Překopírování dat z lokálních bufferů do *VkBuffer* objektů.

```
1.  void MapTile::buildCommandBuffers()
2.  {
3.      VkCommandBufferBeginInfo cmdBufInfo = Vulkan::Initializers::commandBufferBeginIn
    fo(); // struktura pro vytvoření command bufferu
4.
5.      VkClearValue clearValues[2]; // nastavení výchozí barvy pipeline a její hloubky

6.      clearValues[0].color = vkManager->defaultClearColor;
7.      clearValues[1].depthStencil = { 1.0f, 0 };
8.
9.      // Nastavení vykreslovacího přechodu
10.     VkRenderPassBeginInfo renderPassBeginInfo = Vulkan::Initializers::renderPassBegi
    nInfo();
11.     renderPassBeginInfo.renderPass = window->getRenderPass();
12.     renderPassBeginInfo.renderArea... // ostatní nastavení pro oblast vykreslení
13.     renderPassBeginInfo.clearValueCount = 2; // počet čistých hodnot
14.     renderPassBeginInfo.pClearValues = clearValues; // odkaz na čisté hodnoty
15.
16.     QVector<VkCommandBuffer> commandBuffers = window->getCommandBuffers();
17.     QVector<VkFramebuffer> frameBuffers = window->getFrameBuffers();
18.
19.     for (auto commandBuffer : commandBuffers) // pro každý command buffer
20.     {
21.         renderPassBeginInfo.framebuffer = frameBuffers[commandBuffers.indexOf(comman
    dBuffer)];  // nastavení framebufferu dle indexu
22.
23.         vkBeginCommandBuffer(commandBuffer, &cmdBufInfo); // nahrávání command buffe
    ru
24.
25.         vkCmdBeginRenderPass(commandBuffer, &renderPassBeginInfo, VK_SUBPASS_CONTENT
    S_INLINE); // zahájení render passu
26.
27.         VkViewport viewport = Vulkan::Initializers::viewport((float)window-
    >width(), (float)window->height(), 0.0f, 1.0f);
28.         vkCmdSetViewport(commandBuffer, 0, 1, &viewport); // nastavení viewportu
29.
30.         VkRect2D scissor = Vulkan::Initializers::rect2D(window->width(), window-
    >height(), 0, 0);
31.         vkCmdSetScissor(commandBuffer, 0, 1, &scissor); // nastavení scissors
32.
33.         vkCmdSetLineWidth(commandBuffer, 1.0f); // nastavení šířky čáry
34.
35.         VkDeviceSize offsets[1] = { 0 };  // offset vykreslení - 0
36.
37.         // Vykreslení
38.         vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, pipeline);
      // bind pipeline
39.         vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, pipe
    lineLayout, 0, 1, &descriptorSet, 0, NULL);  // bind deskriptorů
40.         vkCmdBindVertexBuffers(commandBuffer, VERTEX_BUFFER_BIND_ID, 1, &model.verti
    ces.buffer, offsets);  // bind vertex bufferu
41.         vkCmdBindIndexBuffer(commandBuffer, model.indices.buffer, 0, VK_INDEX_TYPE_U
    INT32);  // bind index bufferu
42.         vkCmdDrawIndexed(commandBuffer, model.indexCount, 1, 0, 0, 0); // vykresli i
    ndexy
43.
44.         vkCmdEndRenderPass(commandBuffer); // ukončení render passu
45.         vkEndCommandBuffer(commandBuffer); // ukončení nahrávání
46.     }
47. }
```

Zdrojový kód 5: Sestavení command bufferů (Zdroj: vlastní)

4.4 Shadery

V aplikaci byly využité všechny shadery až na geometrický a výpočetní (computed shader). Tesselační shadery využívají algoritmus *Geoclipmapping*.

Shadery byly napsané v jazyce GLSL verzi 4.5.

4.4.1 Vertex Shader (VS)

VS má jedinou vstupní proměnnou: pozici, která je rozšířená ze 3D vektoru na 4D vektor ve funkci *main*. Rozšíření doplňuje poslední hodnotu hodnotou 1. Výstup pro pozici je formou *gl_PerVertex*. Program Vertex Shaderu je možné vidět na zdrojovém kódu č. 6.

```
1.  #version 450
2.
3.  #extension GL_ARB_separate_shader_objects : enable
4.  #extension GL_ARB_shading_language_420pack : enable
5.
6.  layout(location = 0) in vec3 pos;
7.
8.  out gl_PerVertex
9.  {
10.     vec4 gl_Position;
11. };
12.
13. void main()
14. {
15.     gl_Position = vec4(pos, 1.0);
16. }
```

Zdrojový kód 6: Vertex Shader (Zdroj: vlastní)

4.4.2 Tessellation Control Shader (TCS)

TCS provádí výpočet tesselačního faktoru založený na prostoru obrazovky a GPU verzi algoritmu „Frustum Culling“. Frustum Culling algoritmus provádí výpočet, zda se daný patch nachází v ohnisku kamery. V případě, že se nenachází je patch, neprovádí tesselaci.
V opačném případě se provede tesselace patche.

4.4.3 Tessellation Evaluation Shader (TES)

TES po provedení generace primitiv tesselátorem interpoluje pozici. Proběhne výpočet výšky terénu na základě vstupního parametru (sampler), jimž je textura obsahující výškovou mapu.

Pomocí vstupních parametrů projekční a modelové matice dochází k výpočtu pozice. Zjednodušeně by se dalo říci, že se terén přizpůsobuje pohybu kamery.

4.4.4 Fragment Shader (FS)

FS provádí mixování konstantní barvy terénu s konstantní barvou mlhy v poměru 1:4. Výpočet společně s FS lze vidět na zdrojovém kódu č. 7.

```
#version 450

#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable

layout(location = 0) out vec4 fragColor;

float fog(float density)
{
    const float LOG2 = -1.442695;

    float dist = gl_FragCoord.z / gl_FragCoord.w * 0.05;
    float d = density * dist;

    return 1.0 - clamp(exp2(d * d * LOG2), 0.0, 1.0);
}

void main()
{
    const vec4 fogColor = vec4(0.47, 0.5, 0.67, 0.0);

    fragColor = mix(vec4(0.25, 1.0, 0.25, 1.0), fogColor, fog(0.25));
}
```

Zdrojový kód 7: Fragment Shader (Zdroj: vlastní)

4.5 STL soubor

Terén aplikace je tvořen gridem, to pro 3D tisk ovšem nestačí. Grid není totiž uzavřený objekt. Tedy tam, kde končí jeden vertex, druhý nezačíná. V případě neuzavřeného objektu může dojít k tomu, že program Sli3r a podobné budou mít s vysokou pravděpodobností chybný proces slicování, popř. tiskárna začne vrstvit ve vzduchu.

Je tedy nutné provést uzavření a to jednoduchým způsobem. Podstavu bude tvořit rovná plocha tvořená dvěma trojúhelníky. Stěny se dopočítají na základě podstavy a okrajů terénu. Tímto vznikne pomyslný kvádr s tím, že horní podstava bude tvořit terén.

Pro implementaci budou potřeba 3 třídy:

- STL
  - Vezme vstupní *MapTile* pro získání terénních dat.
  - Provede výpočty pro chybějící stěny a podstavu.
  - Zapíše do souboru.
- Face
  - Reprezentace dat dané stěny (celkem 6).
- Vertex
  - Reprezentace 4 vektorů.

STL třída má dvě základní funkce:

- process
  - Získání terénních dat.
  - Vytvoření 6 objektů *Face* a vložením vstupních dat.
  - Vygenerování vertexů pro každý *Face* objekt.
  - Optimalizace *Face* objektu.
- save
  - Nabídne uživateli dialogové okno, kam uložit výstupní soubor.
  - Zavolání funkce *proces.*
  - Zapsání do souboru.

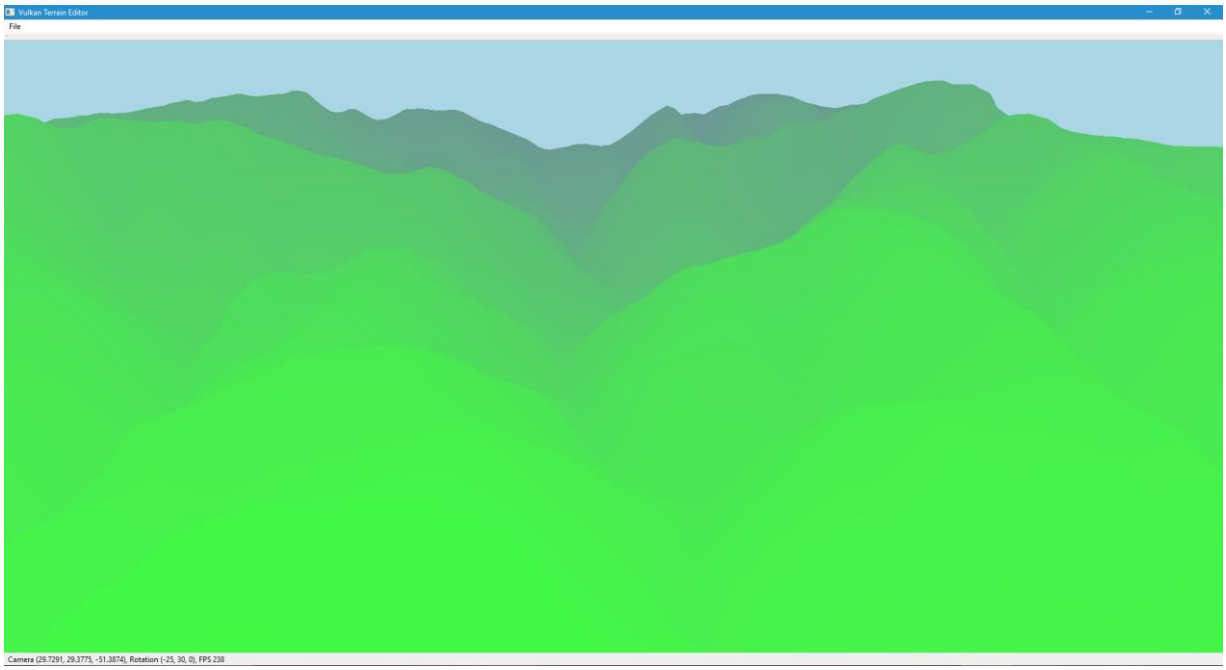
Zápis do souboru lze vidět na zdrojovém kódu č. 8.

```
1.  for(auto& vertex: vertexs)
2.  {
3.      QVector<glm::vec3> vertices = vertex.getVertexs();
4.
5.      glm::vec3 normal;
6.
7.      switch(facet)
8.      {
9.          ...
10.     }
11.
12.     QString facet_normal = QString("facet normal %1 %2 %3").arg(normal.x).arg(normal
    .y).arg(normal.z);
13.
14.     *stream << facet_normal << endl;
15.     *stream << "  outer loop" << endl;
16.     *stream << "    vertex  " << vertices[0].x << "  " << vertices[0].y << " " << ve
    rtices[0].z << endl;
17.     *stream << "    vertex  " << vertices[3].x << "  " << vertices[3].y << " " << ve
    rtices[3].z << endl;
18.     *stream << "    vertex  " << vertices[2].x << "  " << vertices[2].y << " " << ve
    rtices[2].z << endl;
19.     *stream << "  endloop" << endl;
20.     *stream << "endfacet" << endl;
21.
22.     *stream << facet_normal << endl;
23.     *stream << "  outer loop" << endl;
24.     *stream << "    vertex  " << vertices[0].x << "  " << vertices[0].y << " " << ve
    rtices[0].z << endl;
25.     *stream << "    vertex  " << vertices[1].x << "  " << vertices[1].y << " " << ve
    rtices[1].z << endl;
26.     *stream << "    vertex  " << vertices[3].x << "  " << vertices[3].y << " " << ve
    rtices[3].z << endl;
27.     *stream << "  endloop" << endl;
28.     *stream << "endfacet" << endl;
29. }
```

Zdrojový kód 8: Zápis STL do souboru (Zdroj: vlastní)

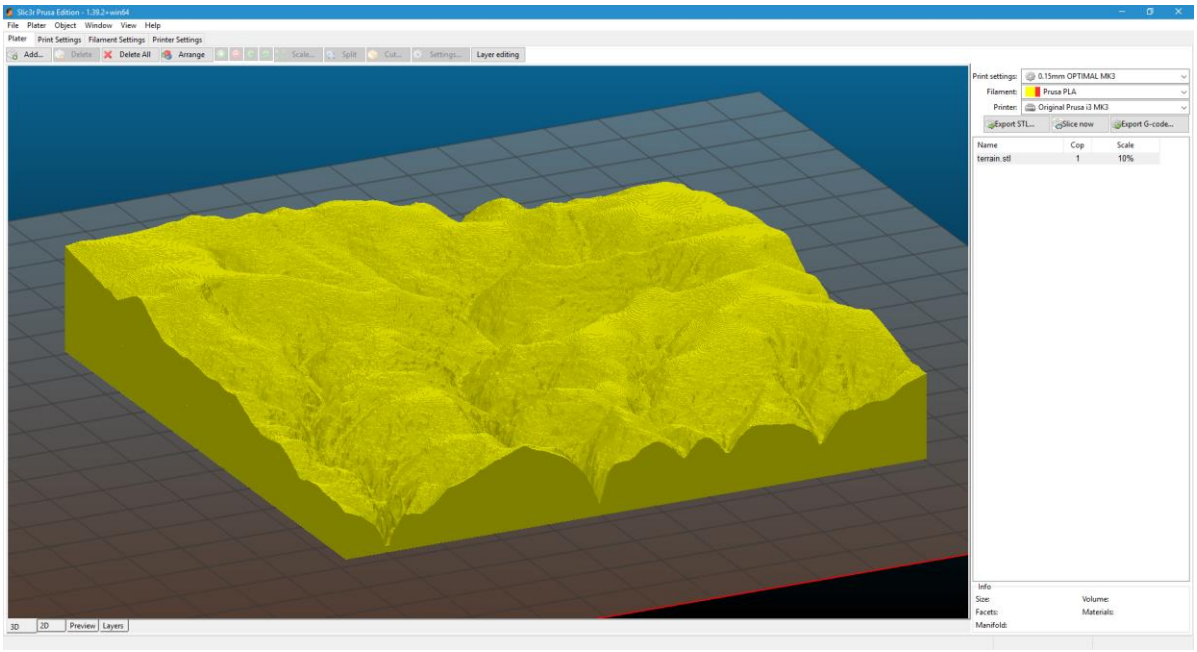
4.6 Zhodnocení aplikace

Výsledkem je poměrně jednoduchá intuitivní aplikace ze strany použití, avšak ze strany zdrojového kódu jde o poměrně složitou část. Výsledek tvorby 3D terénu je možné vidět na obrázku č. 13. Jde konkrétně o výškovou mapu na základě počátečních souřadnic 50° 46' 58.0188" severní šířky, 15° 30' 40.9464" východní délky a konečných souřadnic 50° 40' 8.886" severní šířky, 15° 41' 33.99" východní délky. Oblast znázorněná v aplikaci na obrázku č.13 se pohybuje kolem 13 km^2 se středem v bodě centra Špindlerova mlýnu. Veřejná data pokulhávají s přesností rozlišení, a hodí se tak maximálně na znázornění oblasti v řádech desítek nebo stovek kilometrů čtverečních. Pro detailnější zobrazení v řádech metrů či km se nehodí, neboť nejsou k dispozici data.



Obrázek 13: 3D terén (Zdroj: vlastní)

Na základě algoritmu pro převod terénních dat z kapitoly 4.5 – STL soubor je znázorněný výsledek programu Sli3r Prusa Edition na obrázku č. 14. Pokud se srovnají obrázky č. 13 a č. 14, je možné vidět náznak shody. Obrázek č. 13 zkresluje porovnání efektem mlhy z Fragment Shaderu, a proto nemusí být shoda na první pohled jasná.



Obrázek 14: 3D terén jako STL soubor v softwaru Sli3r (Zdroj: vlastní)

Velkým přínosem této aplikace je ověření využití Vulkan API pro tvorbu 3D terénu na základě spojení s mapovými podklady, které lze poté exportovat jako STL soubor pro případný 3D tisk.

4.6.1 Možnosti rozšíření

K poměrně jednoduchým funkcím z pohledu uživatele lze aplikaci rozšířit o několik desítek možností, které jsou z hlediska programování časově velice náročné. Pokud se uváží např.: herní studio, které má několik softwarových inženýrů a specialistů na daný druh problematiky, přesto studiu trvá několik let vývoj hry nebo enginu.

Top 10 možných rozšíření:

- Texturování terénu
- Barvení podle nadmořské výšky
  - Výpočet ve fragment shaderu na základě výškové mapy.
  - Volba použití textur.
- Tilesety
  - Rozdělení mapy na vícero částí pro větší množství detailů, technika používaná ve hře World of Warcraft.
  - Každý tile by měl své chunky, které zvýší možný počet použití textur.
- Skybox
  - Metoda pro vytváření pozadí, aby prostor nevypadal prázdně.
  - Ve většině případu slouží pro vykreslení mraků a oblohy.

- Occlusion culling
    - Jedná se o pokročilejší algoritmus, který má podobný princip jako Frustum culling.
    - Princip tohoto algoritmu spočívá v tom, že se nevykreslují objekty nacházející se v zorném poli, ale nejsou vidět, jsou například schované za větším objektem.
- Lighting (osvětlení)
    - Možnost přidání virtuálního slunce.
    - Pokročilejší výpočty pro odraz světla.
- 3D modely
    - Vkládání 3D modelů do scén.
    - Auta, budovy, kameny, stromy, postavy, …
- Simulace vodních toků
    - Přidání vody do scény.
- Vlastní souborový formát
    - Vlastní datová struktura.
    - Všechny potřebná data v jednom souboru.
        - Data terénu a textur
        - Pozice světelných objektů (viz Lighting)
        - Pozice 3D modelů
        - Ostatní
- Nástroje pro
    - Úpravu terénu.
    - Texturaci terénu.
    - Přidání/odebrání vody.
    - Přidání/odebrání 3D modelů.
    - Přidání/odebrání světelných zdrojů.

5 Závěr

V teoretické části v kapitole 3.1 – Grafické API byla nejprve provedena charakteristika současných grafických API. Byla stručně popsána jejich historie a aktuální stav. Byly zmíněny přednosti jednotlivých API a zároveň jejich nevýhody. Byla provedena charakteristika programovacích jazyků v kapitole 3.2 – Programovací jazyky, kde po analýze byl zvolen programovací jazyk C++ jako nejvhodnější na základě vlastností pro vývoj aplikace s Vulkan API.

Kapitola 3.4 – Metody renderingu 3D terénu charakterizovala tvorbu 3D terénu. Analýza metod renderování 3D terénu odhalila různé algoritmy pro renderování terénu a bylo zjištěno, že se dříve používalo CPU pro LOD algoritmy, které v současné době nahradilo GPU fází *tesselace*.

Bylo provedeno seznámení s mapovými podklady v kapitole 3.5 – Mapové podklady, které čerpají z dostupných online služeb. Pro mapové podklady je nejdůležitějším parametrem rozlišení, které může značně ovlivnit výsledek.

Byl vysvětlen princip zápisu do souborového formátu STL v kapitole 3.6 – 3D tisk, který je využíván pro 3D tisk, respektive jako vstupní soubor pro získání instrukčního kódu *gcode*, který ovládá osy 3D tiskárny.

V praktické části bylo ověřeno na základě vytvoření a otestování aplikace, že lze využít Vulkan API pro tvorbu 3D terénu na základě mapových podkladů k tisku na 3D tiskárně. Implementace probíhala s knihovnami *Qt* ve vývojovém prostředí *Qt Creator*. Přirozeně byly nalezeny i nedostatky v podobě mapových podkladů, které jsou momentálně nevhodné pro zobrazení v řádech metrů či kilometrů. Lze tedy mapové podklady využít pro zobrazení plochy v řádech desítek či stovek km^2. Bylo zjištěno, že pro náročnější aplikace jako renderování 3D terénu je nevhodné dědit z třídy *QVulkanWindow*, která striktně ukládá, jak nakládat se swap chainem a několika dalšími Vulkan objekty. Je tedy vhodnější provést vlastní implementaci z třídy *QWindow*.

Bylo provedeno ověření převodu mapových podkladů na souborový formát STL pro tisk na 3D tiskárně. Bylo však nutné vytvořit uzavřený objekt, tedy dopočítat jednotlivé stěny a podstavu pro vytvoření pomyslného kvádru. Výstupem je uzavřený 3D model bez chybějících polygonů, který lze vytisknout na 3D tiskárně. Zdrojové kódy aplikace budou použity jako vstupní bod pro nové projekty, které budou pracovat s Vulkan API a 3D terénem.

6 Seznam použitých zdrojů

[1] What is an API? In English, please. *FreeCodeCamp* [online]. freeCodeCamp, 2018 [cit. 2018-11-21]. Dostupné z: https://medium.freecodecamp.org/what-is-an-api-in-english-please-b880a3214a82

[2] RAJAGOPAL, Raj. *Windows NT, UNIX, NetWare migration and coexistence: a professional's guide*. 1st edition. Boca Raton: CRC Press, 1998. ISBN 978-084-9316-692.

[3] What Is a Graphics API?. *Samsung* [online]. Jížní Korea: Samsung, 2018 [cit. 2018-11-21]. Dostupné z: https://developer.samsung.com/tech-insights/vulkan/what-is-a-graphics-api

[4] SELLERS, Graham. *Vulkan programming guide: the official guide to learning vulkan*. 1st edition. Boston, MA: Addison-Wesley, 2016. ISBN 978-013-4464-541.

[5] Khronos Reveals Vulkan API for High-efficiency Graphics and Compute on GPUs. *The Khronos Group Inc* [online]. San Francisco: Khronos Group, 2015 [cit. 2018-11-21]. Dostupné z: https://www.khronos.org/news/press/khronos-reveals-vulkan-api-for-high-efficiency-graphics-and-compute-on-gpus

[6] SINGH, Parminder. *Learning Vulkan*. 1st edition. Birmingham: Packt Publishing Ltd, 2016. ISBN 9781786460844.

[7] Khronos Group Releases Vulkan 1.1. *The Khronos Group Inc* [online]. San Francisco: Khronos, 2018 [cit. 2018-11-21]. Dostupné z: https://www.khronos.org/news/press/khronos-group-releases-vulkan-1-1

[8] BENSTEAD, Luke, Dave ASTLE a Kevin HAWKINS. *Beginning OpenGL game programming*. 2nd ed. London: Cengage Learning [distributor], 2009. ISBN 978-159-8635-287.

[9] MEHTA, Prateek. *Learn OpenGL ES: For Mobile Game and Graphics Development*. 1st edition. New York City: Apress, 2013. ISBN 9781430250548.

[10] LUNA, Frank. *Introduction to 3d game programming with directx 12*. 1st edition. Duxbury, MA: Mercury Learning and Information, 2016. ISBN 978-194-2270-065.

[11] Metal 2. *Apple Developer* [online]. San Francisco: Apple, 2018 [cit. 2018-11-21]. Dostupné z: https://developer.apple.com/metal/

[12] SELLERS, Graham a John KESSENICH. *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. 1. vyd. Addison-Wesley Professional, 2016. ISBN 0134464540.

[13] SMITH, Ryan. Quick Look: Comparing Vulkan & DX12 API Overhead on 3DMark. *AnandTech* [online]. AnandTech, 2017 [cit. 2018-11-21]. Dostupné z: https://www.anandtech.com/show/11223/quick-look-vulkan-3dmark-api-overhead

[14] LARABEL, Michael. Dota 2 Vulkan Performance Across MacOS, Windows 10 & Linux Benchmarks. *Linux Hardware Reviews, Open-Source Benchmarks & Linux Performance - Phoronix* [online]. Phoronix Media, 2018 [cit. 2018-11-21]. Dostupné z: https://www.phoronix.com/scan.php?page=article&item=dota2-mac-vulkan&num=2

[15] MoltenVK. *Github* [online]. Khronos, 2018 [cit. 2018-11-21]. Dostupné z: https://github.com/KhronosGroup/MoltenVK

[16] PRATA, Stephen. *Mistrovství v C*. 3., aktualiz. vyd. Brno: Computer Press, 2007. Bestseller (Computer Press). ISBN 978-80-251-1749-1.

[17] SCHILDT, Herbert. *Mistrovství - Java*. Brno: Computer Press, 2014. Mistrovství. ISBN 978-80-251-4145-8.

[18] NAGEL, Christian. *Professional C# 7 and .NET Core 2.0*. 1st edition. Hoboken: Wiley, 2018. ISBN 9781119449270.

[19] *Vulkan Tutorial* [online]. Alexander Overvoorde, 2016 [cit. 2018-11-21]. Dostupné z: https://vulkan-tutorial.com/

[20] LunarXchange. *LunarG: 3D Graphics Driver and Software Development Services* [online]. Fort Collins: LunarG, 2018 [cit. 2018-12-06]. Dostupné z: https://vulkan.lunarg.com/doc/sdk

[21] Vertex Shader. *The Khronos Group Inc* [online]. Beaverton: Khronos, 2017 [cit. 2018-11-21]. Dostupné z: https://www.khronos.org/opengl/wiki/Vertex_Shader

[22] Tessellation. *The Khronos Group Inc* [online]. Beaverton: Khronos, 2017 [cit. 2018-11-21]. Dostupné z: https://www.khronos.org/opengl/wiki/Tessellation

[23] Geometry Shader. *The Khronos Group Inc* [online]. Beaverton: Khronos, 2017 [cit. 2018-11-21]. Dostupné z: https://www.khronos.org/opengl/wiki/Geometry_Shader

[24] Fragment Shader. *The Khronos Group Inc* [online]. Beaverton: Khronos, 2017 [cit. 2018-11-21]. Dostupné z: https://www.khronos.org/opengl/wiki/Fragment_Shader

[25] Compute Shader. *The Khronos Group Inc* [online]. Beaverton: Khronos, 2017 [cit. 2018-11-21]. Dostupné z: https://www.khronos.org/opengl/wiki/Compute_Shader

[26] MCGUIRE, Morgan. Fast Terrain Rendering with Continuous Detail on a Modern GPU. *Casual Effects* [online]. 2014 [cit. 2018-11-22]. Dostupné z: http://casual-effects.blogspot.com/2014/04/fast-terrain-rendering-with-continuous.html

[27] THATCHER, Ulrich. Rendering Massive Terrains using Chunked Level of Detail Control. *Tulrich.com* [online]. 2002, , 14 [cit. 2018-11-22]. Dostupné z: http://tulrich.com/geekstuff/sig-notes.pdf

[28] WU, Jian, Yan-yan CAO, Zhi-ming CHUI a Xiao-jun WANG. *A New Quadtree-based Terrain LOD Algorithm* [online]. Suzhou, 2010 [cit. 2018-11-22]. Dostupné z: https://pdfs.semanticscholar.org/0b0b/121b30440c959b58444683c8f4b18246eed0.pdf. Seminární práce. Soochow University.

[29] Terrain Modeling. In: *University of Beira Interior* [online]. Convento de Sto. António: University Beira Interior, b.r. [cit. 2018-11-22]. Dostupné z: http://www.di.ubi.pt/~agomes/tjv/teoricas/04-terrainmodeling.pdf

[30] KENT, Jasmine. WebGL Terrain Rendering in Trigger Rally - Part 2. *Gamasutra - The Art & Business of Making Games* [online]. San Francisco: UBM, 2009 [cit. 2018-11-22]. Dostupné z: https://www.gamasutra.com/blogs/JasmineKent/20130908/199798/WebGL_Terrain_Rendering_in_Trigger_Rally__Part_2.php

[31] Landscape Outdoor Terrain. *Unreal Engine 4* [online]. Cary: Epic Games, 2018 [cit. 2018-11-22]. Dostupné z: https://docs.unrealengine.com/en-us/Engine/Landscape

[32] ADT/v18. *WoW Dev Wiki* [online]. 2018 [cit. 2018-11-22]. Dostupné z: https://wowdev.wiki/ADT/v18

[33] How does WoW's terrain generation work?. *GameDev.net* [online]. 2010 [cit. 2018-11-22]. Dostupné z: https://www.gamedev.net/forums/topic/558603-how-does-wows-terrain-generation-work/

[34] Elevation API. *Google Developers* [online]. Mountain View: Google, 2018 [cit. 2018-11-22]. Dostupné z: https://developers.google.com/maps/documentation/elevation/intro

[35] STL 2.0 May Replace Old, Limited File Format. *RapidToday - The Independent Resource for all Users of Rapid Prototyping, Rapid Manufacturing, & 3D Printing* [online]. RapidToday, 2017 [cit. 2018-11-22]. Dostupné z: http://www.rapidtoday.com/stl-file-format.html

[36] Linux kernel interfaces. In: *Wikiwand* [online]. Wikiwand, 2018 [cit. 2018-11-21]. Dostupné z: http://www.wikiwand.com/en/Linux_kernel_interfaces

[37] CHROBOCZEK, Martin. *Grafická uživatelská rozhraní v Qt a C: [plně kompatibilní s Qt 5]*. Brno: Computer Press, 2013. ISBN 978-80-251-4124-3.

[38] Qt (software). *Portable Contacts* [online]. 2018 [cit. 2018-11-21]. Dostupné z: http://portablecontacts.net/wiki/development/qt-software/