VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Framework pro 3D vizualizaci založený na Vulkan API

3D Visualization Framework Based on Vulkan API

2017 Bc. Petr Bláha

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Petr Bláha**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Framework pro 3D vizualizaci založený na Vulkan API
3D Visualization Framework Based on Vulkan API

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je navrhnout a implementovat framework pro 3D vizualizaci s využitím Vulkan API. Předpokladem je využití posledních specifikací. Následně vznikne demo scéna ilustrující funkčnost frameworku.
Body zadání:
1. Přehled aktuálních API a vizualizačních technologií v oblasti 3D s ohledem na téma práce.
2. Analýza, návrh a implementace vlastní aplikace s využitím vybraných technologií.
3. Návrh experimentů a výkonnostních testů.
4. Zhodnocení experimentů, dosažených výsledků a cílů práce.

Seznam doporučené odborné literatury:

[1] Kessenich, John M.: Vulkan programming guide : the official guide to learning vulkan. S.l: Addison-Wesley, 2016.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Petr Gajdoš, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky 1D16 OSTRAVA

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. dubna 2017 .....

Rád bych na tomto místě poděkoval svému vedoucímu práce Ing. Petru Gajdošovi, Ph.D. a své rodině za podporu a pevné nervy při vypracovávání diplomové práce.

Abstrakt

Cílem práce bylo vytvořit framework, který bude zaměřen na vykreslování 3D objektů pomocí Vulkan API. Software byl vytvářen pod platformou Windows. Na vytvořeném frameworku si uživatel může vytvořit vlastní scénu, která může obsahovat velké množství objektů. Uživatel může používat velké množství shaderů a texturových obrazů pro dynamickou změnu scény. Následně na tomto frameworku byly vytvořeny dvě dema (vizuální a zátěžové). Zátěžové demo bylo podrobeno FPS testu.

Klíčová slova: Vulkan, 3D, Framework

Abstract

The aim of this thesis was to create a framework that focuses on rendering 3D objects using Vulkan API. The software was created under the Windows platform. Trough framework user can create their own scene that may contain a large number of objects. The user can use a large amount of shaders and texture images for dynamic change of scene. I created two demos (visual and stress) based on this framework. Stress demo was subjected to FPS test.

Key Words: Vulkan, 3D, Framework

Obsah

Seznam použitých zkratek a symbolů **8**

Seznam obrázků **9**

Seznam tabulek **10**

1 Úvod **12**

2 Existující API **13**
2.1 OpenGL 13
2.2 DirectX 14
2.3 Vulkan 17
2.4 Porovnání jmenovaných API 21

3 Návrh frameworku **22**
3.1 Vulkan rendering 24
3.2 Kontrolní vrstvy 27
3.3 SPIR-V 27
3.4 Bližší popis grafické pipeline 28
3.5 Bližší popis command bufferu 37

4 Demo **39**
4.1 Zátěžové demo 39
4.2 Vizualní demo 42
4.3 Testy na frameworku 48

5 Závěr **50**

Literatura **51**

Přílohy **52**

A Přílohy **53**

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface/ Rozhraní pro programování aplikací
IO	–	Input;output / Vstupní;Výstupní
FPS	–	Frames per second / Obrazy za sekundu
CPU	–	Central processing unit / Centrální procesorová jednotka
TCP/IP	–	Transmission Control Protocol/Internet Protocol (Síťové protokoly)
SDL	–	Simple DirectMedia Layer
GPU	–	graphic processing unit (grafický procesor)
GLSL	–	OpenGL Shading Language ((shader jazyk pro OpenGL))
HLSL	–	High Level Shading Language (shader jazyk pro DirectX)
RGB	–	Red,Green,Blue/ Červená, zelená, modra (spektrum barev)

Seznam obrázků

1 OpenGL pipeline [12] . . . 14
2 DirectX pipeline [13] . . . 16
3 Rozhodovací strom [14] . . . 17
4 Ukázka zodpovědnosti API [14] . . . 18
5 3-vrstvá architektura . . . 18
6 Vulkan pipeline [10] . . . 20
7 Architektura (nákres) . . . 22
8 Ukázka správně použitého vertex bufferu . . . 29
9 Viewport a Scissors . . . 30
10 Teorie pro chlupy [17] . . . 39
11 Použité textury . . . 40
12 Úkázka zátěžového dema . . . 42
13 Výsledná ukázka . . . 43
14 Úkázka plochy . . . 44
15 Textury listů . . . 45
16 Fáze kuličky . . . 46
17 Fáze růstu . . . 47
18 Zátěžový test . . . 49
19 Souhrn: The Talons Principle 1920 [16] . . . 54
20 Souhrn: The Talons Principle 3840 [16] . . . 55
21 Souhrn: Dota 2 1920 [16] . . . 56
22 Souhrn: Dota 2 3840 [16] . . . 57

Seznam tabulek

1 Data z testů 48

Seznam výpisů zdrojového kódu

1 Singleton v jazyce C++ 23
2 Úkázka popisu vertexových informací 28
3 Úkázka sestavování vstupu 29
4 Úkázka viewports and scissors 30
5 Úkázka rasterizace 31
6 Úkázka vícenásobné vzorkování 32
7 Úkázka mixování barev 33
8 Úkázka dynamického stavu 33
9 Úkázka načítání shaderu 34
10 Úkázka pipeline rámce 34
11 Úkázka render pass 35
12 Úkázka vytvoření pipeline 36
13 Část kódu Geometry shader pro výpočet chlupů 41

1 Úvod

Diplomová práce je zaměřena na vytváření frameworku ve Vulkan API. Vulkan API je nové rozhraní vydané v roce 2016. Toto rozhraní vytváří abstraktní vrstvu mezi ovladačem grafické karty a programátorem. Vulkan API by měl představovat nový a výkonnější prostředek pro práci s grafickou kartou a grafikou samotnou. Umožňuje programátorům např. vytvářet aplikace se zaměřením čistě na výpočty a aplikace pro vykreslování 2D a 3D objektů či jejich kombinace.

Diplomová práce začíná popisem běžně dostupných API, jejich vlastnostmi, následované jejich krátkým popisem. V další části je uveden popis architecktury frameworku, volba jednotlivých prostředků, vrstev a jazyků. Následuje všeobecný popis postupu vytváření frameworku a jednotlivých kroků, které jsou nutné k uskutečnění vykreslování pomocí Vulkan API. Po všeobecném popisu je detailnější popis vytváření důležitých částí, jako je práce s command buffery či grafické pipeline. Detailněji popsané části obsahují i ukázky jednotlivých kódů z programu pro popisované problematiky.

Na implementovaném frameworku jsou vytvořené dvě dema, přičemž první je zátěžové a druhé demo je vizuální. Zátěžové demo pojednává o problematice vytvoření dynamicky generovaných chlupů na povrchu objektu s vlivem fyzikálních jevů (síla větru, gravitace). Demo zahrnuje i popis umělého osvětlení či zkrášlení srsti. Ve vizuálním demu je zpracována teraformace terénu osazováním plochy pomocí pohybujícího se objektu. Při popisu vizuálního dema je uveden způsob použití texturových informací, tvorba dynamických objektů a aplikace fyzikálních jevů na tyto objekty. Popis dynamických objektů zahrnuje práci s měněním modelových informací a napojování různých objektů na sebe pomocí texturových dat, tak aby mezi objekty nevznikaly mezery.

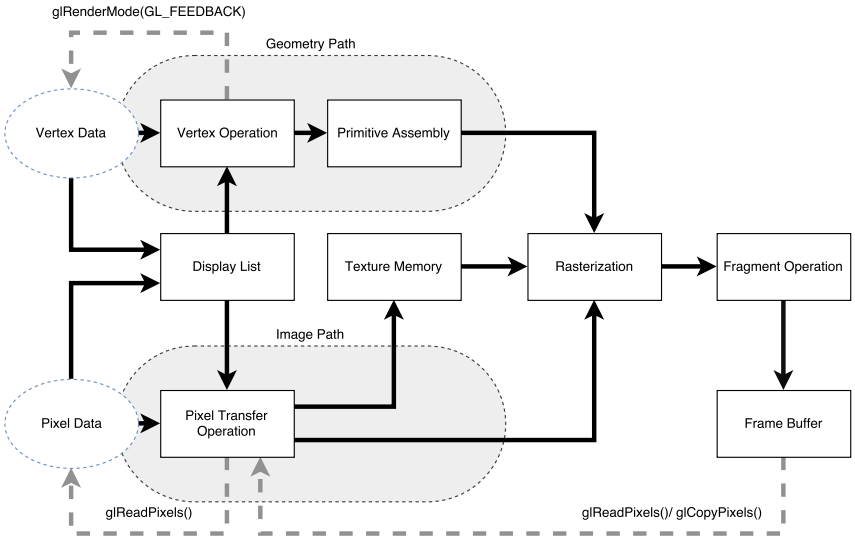
Na závěr práce jsem podrobil vytvořený framework FPS testům. Testy probíhaly na základě vytvořeného zátěžového dema. Testy obsahují tabulkové hodnoty a zároveň také grafikou reprezentaci dat.

2 Existující API

V této sekci se nachází obecný popis dosavadních API a jejich krátká historie. V popisu jednotlivých API se nacházejí i jejich výhody a nevýhody, které se vyskytnou při jejich používání. Je zde popsána i stavba jejich grafické pipeline a její podrobný popis. Probírané API jsou OpenGL[1][2][3], DirectX [4][5][6], Vulkan [7][8][9]. Nejvíce informací je uvedeno u Vulkan API, jelikož je hlavním zaměřením této práce. Informace se týkají například využití nebo architektury. Na konci sekce je krátké porovnání podle vnějších statistik.

2.1 OpenGL

OpenGL je rozhraní pro vytváření 2D a 3D aplikací. Od roku 1992 je to jedno z nejpoužívanějších API pro vytváření grafických aplikací. Umožňuje rychluje vývoj aplikací pomocí zavedení širokých souborových vykreslování, mapování textur, speciálních efektů a dalších výkonných vizualizačních funkcí. Vývojáři využívají OpenGL, protože jej lze rozšířit mezi populární platformy (Windows, Apple, Unix). OpenGL přináší spoustu výhod. Je standardní multiplatformní grafické API, které má širokou podporu na poli průmyslu. Je na trhu více než dvě destetiletí u níž jsou změny v programu dopředu oznamovány. Programátoři tak mohou upravit svůj kód a přizpůsobit se změnám. OpenGL podává konzistetní výsledky na kompatibilním hardwaru bez ohledu na platformu (Windows, Linux, aj.) nebo okenním systému (windowing system). Ačkoliv je na trhu již tak dlouhou dobu, stále umožňuje závádění nových funkcí a reaguje na vývoj hardwaru. Výhodou je lepší kontrola nad kódem, což umožňuje vývoj výkonnějších aplikací. OpenGL umožňuje různé nastavení pro různé typy přístrojů, a proto může být spuštěn na osobních/pracovních počítačích, spotřební elektronice a dokonce i na superpočítačích. Přestože API umožňuje různá nastavení, stále zůstává "User friendly". Struktura s intuitivním desingem a logickými povely umožňuje vytvořit efektivní rutiny, která pomáhají napsat program rychleji a snadněji, současně kód zůstává poměrně krátký. OpenGL je na trhu velmi dlouho a tak dokumentace k této API je velmi bohatá a snadno dostupná. K tomuto API bylo již napsáno několik knih, návodů a dokumentů a stále vychází nová literatura [11].



Obrázek 1: OpenGL pipeline [12]

Na obrázku 1 lze vidět postup vytváření obrazu pro OpenGL aplikace. Ve Vertex Operation bloku se zpracovávají vertexová data pomocí vertex shadru. Zpracovaná data jsou poté zpracována v bloku Primitive Assembly. Blok Primitive Assembly zajišťuje vytváření/redukci geometrie, která vychází z předešlého bloku. Obsahuje stejné operace, například teselace a geometry shader, jako má DirectX. Dále nastupuje Rasterization blok, který osekává prvky za hranicí viditelnosti, přeměňuje objektová data na pixely a doplňuje hrany a možné další operace. Následuje Fragment Operation blok, který zpracovává příchozí data a vytváří se obraz, který se uloží do Frame Bufferu. Frame Buffer může být znovu použit pro další zpracování, například na efekt stínů nebo reflexe a refrakce. Display list block slouží jako mezipaměť API OpenGL, ukládájí se (caching) zde data tak, aby mohla být rychle vyvolána z paměti grafické karty. A data se nemusejí zpětně přenášet z RAM paměti. Texture Memory blok slouží k načtení texturových obrazků a jejich užití pro všechny programovatelné bloky. Pixel Transfer Operation blok se využívá pro operace škálování, zkreslení, mapování a upínání.

2.2 DirectX

DirectX byl představen v roce 1995. Byl vytvořen se záměrem ulehční vývoje počítačových her programátorům na platformě Windows. Do té doby programátoři využívali DOS, ve kterém bylo nevýhodou psání nadbytečných kódů pro hardware.

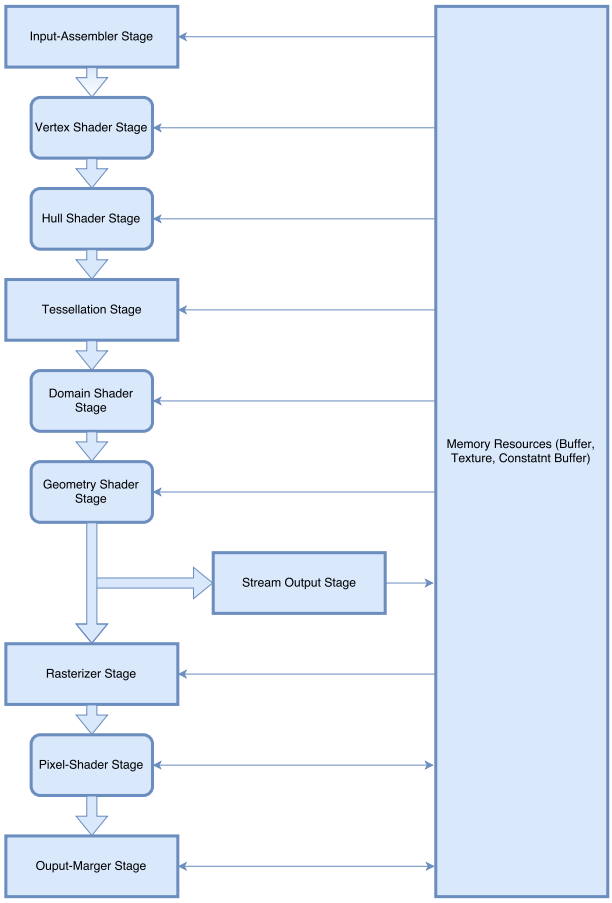
DirectX je označení pro soubor API vytvořené firmou Microsoft. Tento software umožňuje vývoj aplikací na počítači tak, aniž by se bral v potaz, jaký hardware se vyskytuje na daném zařízení. Tento benefit je nejlépe viditelný při vývoji her. Narozdíl od herních konzolí jako PlayStation nebo GameCube, počítačové hry potřebují být vytvářeny tak, aby měli co nejvyšší výkon. DirectX vytváří vrstvu mezi vývojářem a hardwarem, čímž umožňuje vývojáři soustředit se na vývoj aplikace, aniž by musel brát v potaz rozdíly mezi samotnými hardwary, na kterých

jeho aplikace poběží. Jako příklad by mohl být nynější problém mobilních aplikací, ve kterých programátor specifikuje pro každou verzi displeje/androidu vlastní nastavení.

DirectX je složením těchto API:

- DirectDraw - je rozhraní umožňující přímý přístup ke zobrazovacímu zařízení, přičemž stále udržuje kompatibilitu s grafickým rozhraním dané karty v systému Windows.
- Direct3D - API umožňující manipulaci a zobrazení 3D objektů. Také umožňuje přístup k 3D urychlovacím prvkům na grafických kartách.
- DirectSound - umožňuje přístup k přehrávání zvuků s nízkou odezvoza velkou kontrolou nad hardwarem zařizující tuto funkci.
- DirectMusic - v kombinaci s DirectSound dává plnou kontrolu programátorovi nad přehráváním hudby či jiných zvuku ve vyvíjené aplikaci.
- DirectInput - dává přístup programátorovi k perifériím jako ovladač, klávesnice a myš.
- DirectPlay - toto rozhraní dává programátorovi přístup k síti na transportní vrstvě protocolu TCP/IP nebo IPX protokolu.
- DirectShow - umožňuje šíření dat na windows platformách. Podporuje vysoce kvalitní zachytávání a přehrávání multimediálních dat.

DirectX má několik výhod oproti OpenGL. Například umožňuje programátorovi přístup nejen ke grafickým částem, ale také k hardwarovým perifériim. Zároveň se to může jevit jako nevýhoda, protože někteří programátoři tyto části nechtějí využívat, ale musí se potýkat se všemi výše uvedenými částmi. Další výhodou je vysoká optimalizace pro platformu Windows. Jelikož DirectX není multiplatformní API, neboli ho nemůžeme využít jinde než na Windows platformě, je to zároveň i nevýhodou.



Obrázek 2: DirectX pipeline [13]

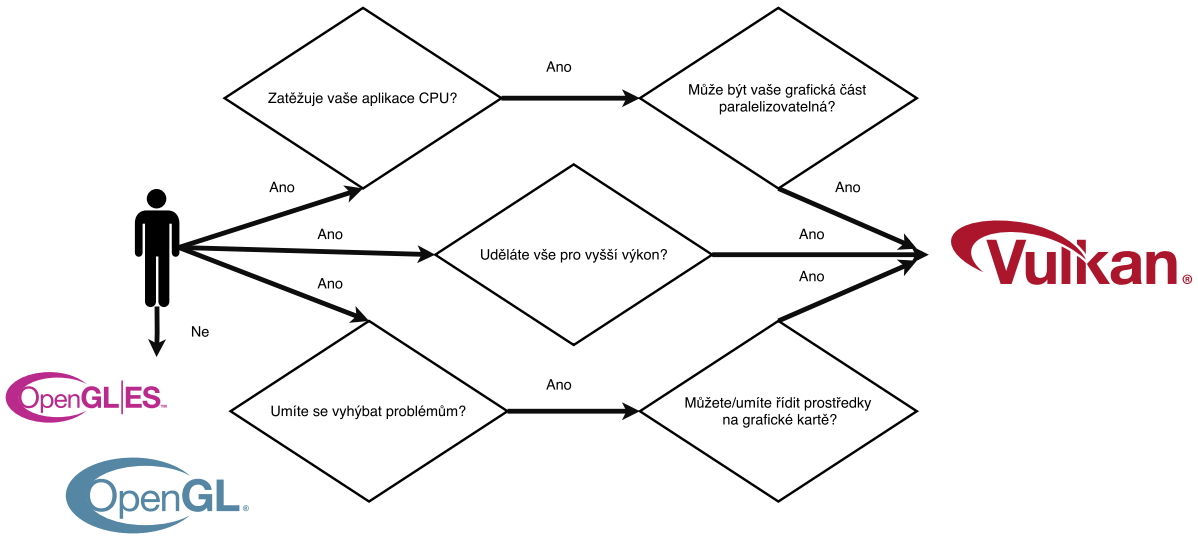
Na obrázku 2 lze vidět pipeline od DirectX. Pipeline začíná Intput-Assembeler Stage blokem, který má za úkol dodávat data o vertexech. Informace z Intput-Assembler Stage bloku jsou předány do Vertex shaderu. Vertex shader je dodáván programátorem a generuje základní geometrii scény. Následně se použije Hull shader, který může zpracovávat, jak úpravu geometrie, tak i vstup pro teselation stage, který udává "rozlišení"teselace. Dále může být využit teselační blok. Teselation blok zajišťuje dělení trojúhelníků/polygonů na menší části. Hlavní výhodou teselačního bloku je ušetření paměti a zjemnění detailů geometrie. Následuje blok Domain Shader, který převádí koordináty z teselačního bloku nebo Hull shadru na "hmatatelný"objekt ve 3D prostoru. Následuje programátorem definovaný Geometry shader, který může generovat/redukovat vertexové informace. Data můžou být vrácena zpět do paměti pro urychlení dalšího zpracování a dále pokračují do bloku rasterizace. Rasterizace má za úkol odstranit prvky mimo viditelné pole a také přeměnit geometrii na pole pixelů potřebné pro Pixel-Shader. Dále následuje Pixel-Shader, jenž produkuje pouze barvy na výslednou plochu obrazu. Output-Merger zajišťuje mísení barev v případě výpočtu průhlednosti.

2.3 Vulkan

Stejně jako předchozí grafické API i Vulkan byl navržen pro meziplatformovou abstrakční vrstvu nad grafickým ovladačem. Jedním ze závažných problémů se stávajícími API je ten, že byly navrženy a realizovány v době, kdy grafické karty měli jinou architekturu a nebyli tak vyvinuté, jako jsou nyní. Programátoři grafických aplikací v té době byli plně odkázáni na rozmary firem vydávající grafické karty. V průběhu času se architektůra grafických karet měnila, proto se musely měnit i API pro práci s nimi. To ovšem nebylo zcela jednoduché, protože integrace takových změn někdy probíhala za cenu výkonu. Proto byly vydávány často nové verze softwaru, aby se tyto změny daly lépe zpracovat. Na trhu se objevily nové mobilní zařízení s dostatečně výkonným hardwarem pro grafické úkony. Nové výkonné mobilní zařízení přispěli k možnosti vývoje nového grafického API s podporou pro tyto zařízení. Proto se začla navrhovat zcela nová API, která neobsahovala výše popsané nedostatky, jelikož by byla v souladu s moderní architekturou a modernímy přistupy.

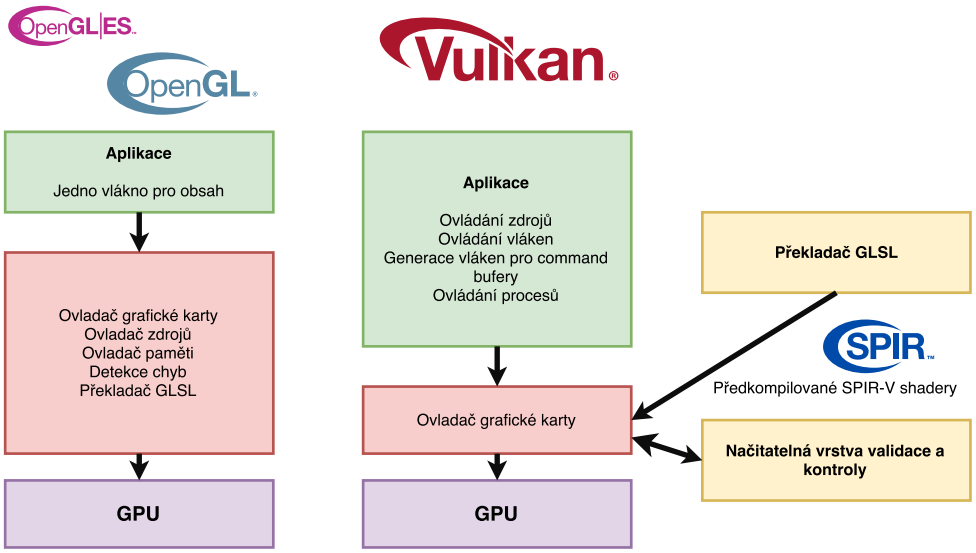
Vulkan byl spuštěn v roce 2016. Byl vytvářen stejnou skupinou, která vytvářela OpenGL, neboli KHRONOS Group společeně s podporou firem jako AMD, Nvidia a další. Vulkan je čistě grafická aplikace, neboli stejně jako OpenGL nevytváří okna, ve kterých jsou viditelné výsledky. Na tyto okna je zapotřebí mít jiné prostředky např. GLFW, SDL nebo GLUT. Vulkan podporuje tzv. "cross platform development", což znamená, že aplikace vyvíjející se pod Vulkan API, lze rozběhnout pod různými platformami. Podporované platformy jsou: Windows xp/7/8/10, SteamOS, ubuntu, radhat, tizen, android.

Ovšem oproti ostatním grafickým API, Vulkan dáva více prostoru programátorovi ke znovuvyužití stávajících prostředků a tím i urychlení programu. Nicméně je s tím spojen fakt, že programátor se musí vypořádat s věcmi navíc (např. správou paměti).



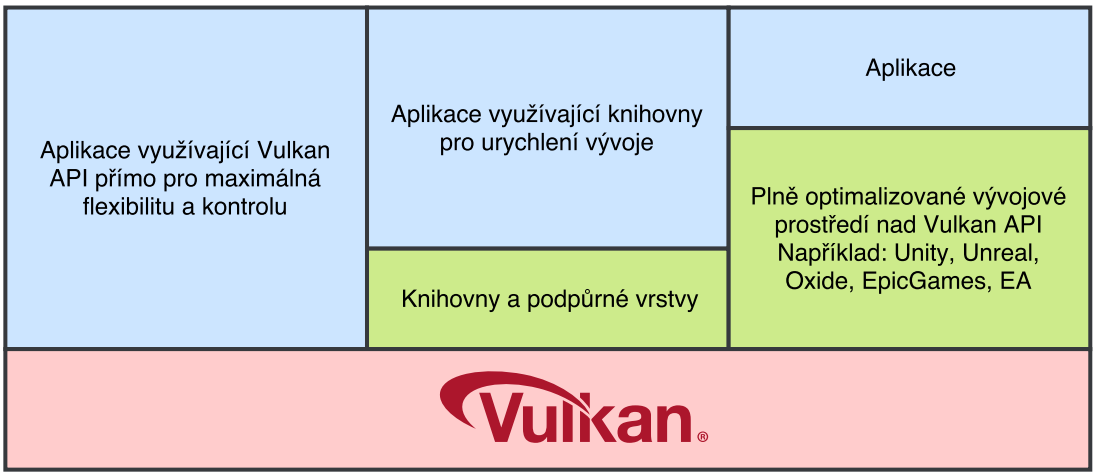
Obrázek 3: Rozhodovací strom [14]

Vývojáři Vulkan API představili rozhodující strom, který ukazuje "KDY"se uchýlit k použití Vulkan API, což je viditelné na obrázku 3. Jako u všech softwaru i Vulkan API má svoje výhody a nevýhody. Před vybráním API se vývojář musí ujistit, že využije více výhod, oproti nevýhodám, při vývoji softwaru. Tím se předejde zjištění, že zvolené API neodpovídá potřebám pro daný software a například se vývoj aplikace několikanásobně neprodraží.



Obrázek 4: Ukázka zodpovědnosti API [14]

Vulkan nutí programátora převzít velkou část zodpovědností za využití zdrojů a práci s nimi. Tento proces je ukázán na obrázku 4. Programátor může díky této možnosti nakládat se zdroji ve větším rozsahu. Například může znovu využít paměť, restartovat command buffry nebo další věci, které vedou k vyšším výkonům a umožňují lepší kontrolu nad procesem. Kdyby tyto prvky nemohl spravovat "ručně", musel by vytvářet stále nové prvky a ty staré by se rušily podle pravidel ovladače grafické karty nebo podle pravidel využité API.



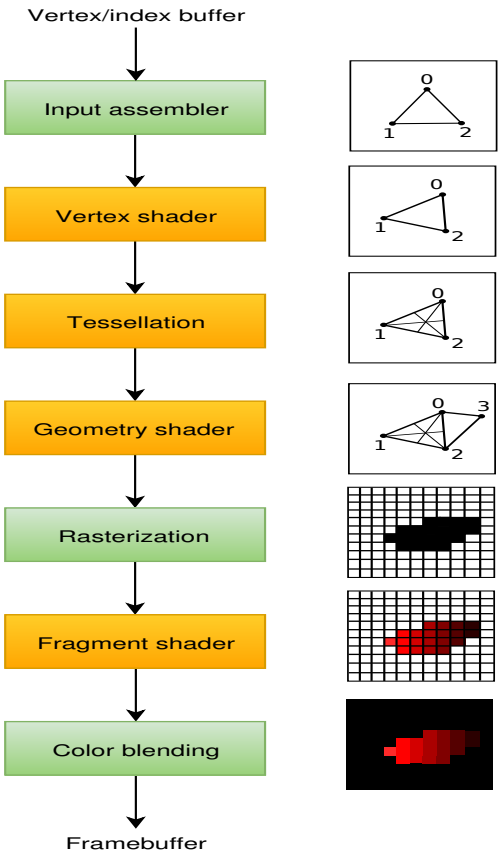
Obrázek 5: 3-vrstvá architektura

Vulkan lze rozdělit do tří přístupů vytváření architektury, což je viditelné na obrázku 5. První typ přístupu, je mít pouze vlastní aplikaci bez jakékoliv knihovny třetí strany. Při tomto přístupu má programátor kontrolu nad celým programem. Nicméně program nemusí být zcela vylazen, protože programátor nemusí znát všechny postupy a triky. Tyto znalosti mají většinou vysoce specializovaní lidé, kteří pak vytvářejí různé specializované knihovny. Tyto knihovny se pak využívají ve druhém přístupu v prostředním sloupci na obrázku 5. Tento přístup má výhodu, že se vytvoří vrstva mezi Vulkanem a aplikací. Tato vrstva se může následně kdykoliv nahradit a přejít popřípadě na jiné API. Poslední přístup je mít mezivrstvu pomocí vývojového prostředí jako je například Unity. Je důležité, aby toto prostředí podporovalo Vulkan. Jinak samozřejmě přechod není možný.

Vulkan také přichází s užitím nového shader formátu. Tím je SPIR-V [15]. Tento formát je binární souboru nezávislý na platformě. SPIR-V formát je soběstačný, plně specifikovaný pro reprezentaci etap grafických shadru a výpočetních jader pro API. Fyzicky to je proud 32 bitových slov. Logicky je to hlavička následovaná proudem instrukcí. Tyto instrukce nejdříve obsahují anotace, následně kolekce funkcí. Každá funkce následně kóduje svůj instrukční proud do grafu. Nahrávací a ukládací instrukce jsou následně použity k přístupu k proměnným, které zahrnují všechny IO operace.

Vulkan dále přichází s "command buffery". Tyto buffry mají za úkol nahradit dosavadní volání vykreslení entit. Místo toho se tyto úkony nahrají do command bufferu. Command buffery podporují paralelismus. Každý command buffer má vlastní vlákno a tím pádem se více využije CPU. Následně stačí v hlavním vláknu sputit tyto buffery, aby se dosáhl výsledek. Tyto buffry jsou uzpůsobeny k znovu-užití. To má za následek zisk na výkonu, jelikož nemusíme tyto buffry znovu deklarovat. Místo toho se přemažou novými instrukcemi tak, aby nedocházelo k zamykání vláken. KHRONOS Group přišel s takzvanou "command pool" strukturou, která se stará o tyto mezivláknové komunikace.

Command buffery nejsou rozděleny do kategorií, nebo-li každý command buffer může vykonávat jakoukoliv činnost. Ovšem fronty, do kterých se tyto buffry po naplňení přiřazují, mají již kategorie jednoznačné přiřazené.



Obrázek 6: Vulkan pipeline [10]

Vulkan pipeline je viditelná na obrázku 6. V podstatě je stejná jako u předchozích API. Input assembler se stará o přísun surových objektů z vertex/index bufferu. Vertex shader následně aplikuje transformace na tyto data. Tessellation zjemňuje strukturu modelu. Geomtery shader generuje či redukuje objekty podle účelu. Rasterizace odstraňuje neviděné části za hranicí obrazu a doplňuje obraz na pixely pro fragment buffer. Fragment buffer následně zabarvuje výsledný obraz. Color blending provádí operace na základě nastavení, například operace pro realizaci průhlednosti. V tom případě se musí operace v bloku Color blending nastavit tak, aby se načítaly na základě alfa kanálu.

Oranžové bloky jsou programovatelné moduly, kdežto zelené jsou nastavitelné pomocí popisových struktur přímo při vytváření Vulkan pipeline.

2.4 Porovnání jmenovaných API

Se znalostí hrubého přehledu o API pro vykreslování grafických úkonů, se můžeme podívat na dosavadní statiskity, než provedeme vlastní zhodnocení vytvořeného programu. Převzaté statistiky jsou uvedeny na [16].

Test výše uvedených API byl proveden na hrách Dota 2 a The Talons Principle, protože tyto hry jsou podporovány na platformě Linux. Obě hry také na platformě Linux podporují vykreslování pomocí OpenGL a Vulkan API, DirectX na platformě Windows. Díky tomu se stávají ideálním prostředníkem pro porovnávání výkonu na různých platfomách. Hardware byl stejný po celou dobu testů na všech platfomách.

V příloze jsou zahrnuty obrázky s výsledky. Výsledky jsou pro dané hry v rozlišení 1920x1080 a 3840x2160.

První testovanou hrou byl The Talon Principle (viz příloha). Z uvedené statisky lze vidět, že největší výkon vykazuje Direct3D (podnož DirectX). Nicméně DirectX není Linuxu přístupný, což je limitující pro propagaci a vývoj hry. Na dalším místě se nachází Vulkan se srovnatelným výkonem pro obě platformy. Na posledním místě se umisťuje OpenGL v FPS testech. Lze vidět, že Vulkan ma zhruba o 13% menší výkon než DirectX, kdežto OpenGL má zhruba o 50% menší výkon než DirectX.

Oproti výslednům z FPS testů pro hru The Talons Prinicples vidíme, že ve staticstice pro hru Dota 2 se pořadí mění. Nejlepší výsledek v porovnání výkonu hry má Vulkan. Také lze vidět, že záleží na grafické kartě, která se použije a na míře optimalizace pro tyto operace. Pro kartu Radeon R9 Fury je DirectX na posledním místě se ztrátou necelých 25%, ale na kartě Radeon RX 480 má ztrátu nižší, jen 15%. Oproti tomu OpenGL má celkem konstantní úbytek kolem 18%.

Je nutno vzít v potaz, že tento výkonostní test byl prováděn krátce po vydání Vulkanu. Nicméně se od API Vulkan očekává další vývoj a optimalizace v průběhu let, tak jako u ostatních API. Test byl proveden na různých grafických kartách firmy AMD, nikdoliv na kartách od různých firem s podobným výkonem grafických karet.

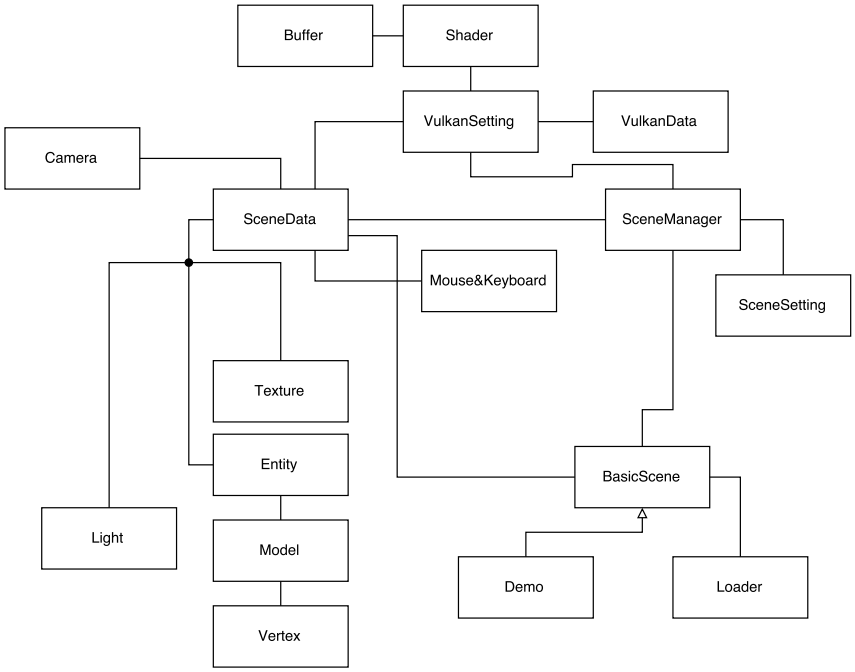
Nicméňe lze vidět, že technologie Vulkan má potencial jako nastávající API pro vývoj her či jiného grafického průmyslu, ve kterém je důležitý výkon. Nebo-li je vhodný i pro budoucnost virtuální reality a jejího rozvoje.

3 Návrh frameworku

Cílem této práce je vytvoření grafického frameworku pro vykreslování objektů pomocí technologie Vulkan. Program je navržen podle prvního způsobu z obrázku 5. Jelikož byla Vulkan API vydaná teprve v roce 2016, nebyly na začátku práce dostupné optimalizované knihovny pro psaní programu. Z toho důvodu jsem se rozhodl pro napsaní architektury bez mezivrstvy. Využití enginu, jako je například Unity, nepřipadá v úvahu vzhledem k povaze práce. Navrhnul jsem architekturu, kterou lze vidět na obrázku 7. Plnohodnotný třídní diagram zde nebude uveden z důvodu čitelnosti.

Na nákresu architektury lze vidět hlavní část SceneManager. Tento blok je zodpovědný za spuštění a řízení celého programu. Musí mít přístupná všechna data ve scéně, neboli část SceneData. Dále potřebuje mít přístup k hlavnímu nastavení, které je uchováváno ve SceneSetting. BasicScene je rozhraní mezi uživatelským kódem aplikace a implementovaným frameworkem, neboli výsledkem této práce. Demo tím pádem implementuje toto rozhraní a obsahuje veškerý kód uživatelského programu. Bližší popis s obrázky viditelný v kapitole 4.

Loader slouží jako mezivrstva pro uživatele a načítání dat. Načítaná data jsou buď obrázky (.png,.jpg) nebo objekty (.obj). Loader byl takto navržen z důvodů možné změny prostředků pro načítání objektů či texturových dat. Tohoto benefitu bylo využito při změně načítání knihovny z TinyObjectLoader na Assimp. Assimp je všestranější knihovna a umožňuje práci s více datovými formáty. Díky existenci bloku Loader bylo možné lehce změnit využívanou knihovnu a její potřeby, aniž by byla nutná zněma ve zbytku programu.



Obrázek 7: Architektura (nákres)

Dalšímy prostředky, kterými musí engine disponovat jsou kamera a přístup k periferiím počítače, jako je například počítačová myš nebo klávesnice. Metody pro obsluhu klávesnice a myši se nacházejí ve třídě Mouse&Keyboard a nástroje pro správu kamery se nacházejí ve třídě Camera. Kamera obsahuje obvyklé operace jako je posun, otočení nebo přiblížení/oddálení.

Obsluha samotného vykreslování je spravována větví VulkanSetting. Tato větev je zodpovědná za řízení a komunikaci mezi uživatelským kódem ve třidě Demo a grafickým ovladačem karty. VulkanSetting ke svému provozu potřebuje třídu VulkanData, která obsahuje veškeré informace, které se musejí dlouhodobě uchovávat pro provoz. Jednou z nejdůležitějších uchovávaných informací je například ukazatel na vybrané grafické zařízení (vkPhysicalDevice).

Pro celý program je důležité mít možnost načítat "shadery". Shadery jsou uživatelské programy pro správu geometrie, vertexů a finální barvy obrazu. Tyto shadery jsou uchovávány v programu a využívány třídou VulkanSetting pro tvoření obrazu. Tyto programy jsou definovány uživatelem, proto k nim musí mít přístup. A z toho důvodu se nacházejí pod třídou SceneData a nikoli VulkanData.

Poslední částí jsou třídy podporující prácí uživatele. Těmito třídamu jsou Entity, Model, Texture, Light, Vertex. Třída Vertex slouží k uchování dat o určitém bodě ve 3D nebo 2D prostoru, například pozici, barvu, a jiné. Třída model uchovává veškeré informace o objektu tvořeném pomocí vertexů. Entity představuje objekt v prostoru. Třída Texture uchovává data o informacích v obrázkové podobě. Třída Texture může uchovávat například výškovou či normálovou mapu nebo barvu modelu. Třída Light slouží k uchovávání informací o zdrojích světla ve scéně a jeho vlastnostech, jako je například barva, intenzita.

Singleton

V programu je hojně využíván návrhový vzor Singleton. Tento vzor má za účel uchování jen jedné instance dané třídy. V celém programu se tedy nachází jen jedna kopie dat.

```
class Singleton {
private:
    Singleton();
    static Singleton INSTANCE;
public:
    static Singleton& instance(){
        return INSTANCE;
    }
};
```

Výpis 1: Singleton v jazyce C++

3.1 Vulkan rendering

Architekturu programu je navržená a nyní je možné se zaměřit na samotnou produkci frameworku. V této kapitole je popis bloku VulkanSetting z obrázku 7. V bloku VulkanSetting jsou zpracovány všechny informace týkající se tvorby a zpracování dat, mezi aplikací a ovladačem grafické karty. Dále jsou zde informace týkající se SPIR-V formátu pro Vulkan shadery. Vykreslení objektů ve Vulkan API se dělí do několika kroků.

1. **Instance a vybrání fyzického zařízení**
   Celý proces začíná inicializací Vulkan API přes strukturu VkInstance. Instance se vytváří pomocí jejího popisu (jména, verze) a jakéhokoliv API rozšíření, kterého bude zapotřebí. Po vytvoření instance se musí vybrat vhodné fyzické zařízení, které bude provádět výpočty či jiné úkony. K tomu slouží metoda vkEnumeratePhysicalDevices, která vrací všechny fyzická zařízení vyskytující se na stroji v poli struktur typu vkPhysicalDevice. Díky této struktuře se může vybrat vhodné zařízení pro práci, či rovnou několik pro různé operace. Například některá karta může být lepší pro výpočty a jína pro zobrazování.

2. **Logické zařízení a fronty**
   Po vytvoření instance a vybrání vhodného grafického zařízení pro vyvíjenou aplikaci je zapořebí vytvořit logické zařízení (VkDevice), ve kterém se nadefinují používané funkcionality, pomocí struktury VkPhysicalDeviceFeatures. Možnými funkcioalitamy jsou například 64 bitové float parametry, povolení Geometry shadru či teselace. Je zapotřebí specifikovat rodiny front (queue families), které se budou používat pro aplikaci, jelikož vetšina operací ve Vulkanu je prováděna pomocí asynchroního volání přes tyto fronty. Každá rodina operací má předdefinované vlastnosti a účel. Ve Vulkanu existují různé rodiny front pro operace typu zobrazení, výpočty, paměťové operace. Dostupnost zvoleých front pro aplikaci může být také jedním z faktorů při výběru fyzického zařízení. Je možné, že některá zařízení (mobily/tablety) nepodporují určité rodiny front, ale všechny dostupné grafické karty s podporou Vulkan API disponují všemi výše uvedenými rodinami front.

3. **Povrch okna a výměný řetězec (Window surface and swap chain)**
   Pokud se nejdedná o aplikaci čistě pro výpočty, tak je potřeba vytvořit okno pro presentaci výsledků neboli obrazu. Okno může být vytvořeno pomocí nativních API vyskytujících se na platformě, nebo pomocí knihoven (nejlépe podporujících meziplatformí vytváření oken). Pro tento projekt byla zvolena knihovna GLFW.

   K prezentaci obrazu jsou zapotřebí další dvě komponenty. A to je porvch okna (VkSurfaceKHR) a výměný řetězec (VkSwapChainKHR). KHR přípona značí, že je to součást Vulkan rozšíření. Vulkan není uzpůsoben k vytváření oken, proto se musí využít rozšíření, která zajistí tuto funkcionalitu. Za tímto účelem byla zvolena knihovna GLFW, protože tato knihovna obsahuje tyto rozšíření a dokáže se spojit a komunikovat s nativním rozhraním pro vytváření oken zabudované v platformě.

Výměný řetězec slouží k zaručení synchronizace mezi připravenými a připravovanými snímky. Neboli snímky, které už byli dokončeny a snímky, které se teprve vytvářejí. Protože je důležité, aby se jen dokončené snímky zobrazovali uživateli. Pokaždé, když je potřeba vytvořit nový snímek, tak se požádá výměný řetězec o obraz, do kterého následně můžeme ukládat data. Po dokončení snímku se vrátí tento obraz výměnému řetězci, který tento snímek pak prezentuje. Výměný řetězec lze různě nastavovat. Běžně se používá nastavení s dvojtou nebo trojtou sekvencí snímků. Pro dvojtou sekvenci to znamená, že pokud se jeden snímek připravuje druhý je zobrazen, a tak se to stále mění.

4. **Zobrazovač obrazu (Image views) a frame buffer**
   Pro vykreslení obrazu získaného z výměného řetězce, je zapotřebí obalit okno do struktury VkImageView s přiřazeným frame bufferem (VkFramebuffer). VkImageView je využito k určení pozice, která se bude využívat, například nebudu chtít využít celou plochu obrazu, ale jen jeho část. VkFramebuffer určuje přímé vlastnosti obrazu, jako je například barva, hloubka.

5. **Render passes**
   Render passes ve Vulkan API slouží k popsání typů obrazů vutvářených pomocí vykreslovacích operací, jak budou použity a způseb zacházení s obsahem těchto obrazků. V aplikaci je nastaven tento prvek tak, aby se při obdržení nového snímku obsah vyčistil na předurčenou barvu. Dále je využita možnost hloubkového mapování pro překrývání objektů ve scéně. Render passes jen slouží k popisu obrázku. Jeho realizace je pomocí předešle zmíněného VKFramebuffer.

6. **Grafická pipeline**
   Grafický pipeline je Vulkan API je založena pomocí úspěšného vytvoření VkPipeline. Tento objekt popisuje kofugirační stav grafické karty, jako jsou například operace pro mísení výsledků pro obraz nebo rasterizace (effekt zajišťující ořezání objektů mimo obraz). Dále využití programovatelných modulů neboli shadru pomocí objektu typu VkShaderModule. Tyto moduly jsou tvořeny bytovým kodem přeložených shadru ve formatu SPIR-V. Ovladač dále potřebuje vědět, který frame buffer využít a to pomocí předání Render pass objektu.

   Jedním z rozlišujících faktorů Vulkanu oproti ostatním API je, že VkPipeline musí být definována dopředu. To znamená, že při jakékoliv změně, například shadru nebo konfigurace se musí znovu vytvořit celý VkPipeline objekt. Ovšem existují i malé vyjímky jako je nastavení základní barvy nebo úhel pohledu. Jinak se musí vytvářet mnoho VkPipeline objektů pro různé použití. Celkové nastavení se také musí provádět "ručně", jelikož VkPipeline nemá základní nastavení.

Celý tento proces má také několik výhod. Například že kompilace se provádí dopředu a tím pádem je více možností pro optimalizace ze strany ovladače a výkonost je více předvídatelná, protože výměna nastavení pomocí změny VkPipeline jsou rychlá.

7. **Vytvoření Buffer objektů**
   Pro upřesnění co se bude používat pro konfiguraci VkPipeline v jejím programovatelném bloku, neboli VkShaderModule, se musí nejprve vytvořit popisové objekty, které nesou informaci o propojení mezi programovatelným modulem a programem samotným. K tomu slouží VkBuffer objekty, které se dělí dále podle popisových značek na VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, VK_DESCRIPTOR_TYPE_SAMPLER a další, které v této práci nebyly využity. Dále se využívají také popisové struktury pro vertex buffery a instance buffery využité pro kreslení. Ty mají značky VK_VERTEX_INPUT_RATE_INSTANCE, VK_VERTEX_INPUT_RATE_VERTEX. Těmito popisovými strukturami se popisuje funcionalita a využití daného VkBuffer objektu pro práci ve VkPipeline objektu.

8. **Příkazové buffery a příkazové hromady (command buffers, command pools)**
   Vulkan API funguje na principu zaznamenání akcí do VkCommandBuffer. Díky tomu se využívá větší potenciál CPU, jelikož každý command buffer pracuje ve vlastním vlákně. Všechny tyto Buffery pocházejí z VkCommandPool objektu, který je přímo spojen s předurčenou rodinou frontou. K vykreslení objektu je zapotřebí spuštění několika základních akcí:

   - Získání render pass objektu
   - Přiřazení grafické pipeline
   - Přiřazení zdrojů (uniform buffer, vertex buffer)
   - Použití vykreslovacích funkcí
   - Odeslání render pass objektu

   Jelikož VkCommandBuffer pracují nezávisle na sobě, každý má vlastní vlákno a výměnný řetězec nezaručuje synchronizaci mezi vlákn. Nejlepší je nahrát VkCommandBuffer pro každý obraz a vybrat ten správný při vykreslování.

9. **Hlavní vykreslovací smyčka**
   Nyní jsou už zaznamenány VkCommandBuffer a hlavní vykreslovací smyčka je přímočará. Nejdříve je zapotřebí získat obraz z výměného řetězce pomocí metody vkAcquireNextImageKHR. Dále získání správného VkCommandBuffer pro tento obraz a jeho exekuci pomocí metody vkQueueSubmit. Následně se vrátí obraz zpět do výměného řetězce a zobrazí se na plochu pomocí metody vkQueuePresentKHR.

Operace, které jsou vloženy do fronty jsou spouštěny nezávisle na sobě. Nebo-li se musí využít synchronizačních objekty, jako jsou například semafory pro zajištění korektního pořadí. Jinak by mohlo dojít k přepisování obrazu, který je teprve zobrazován uživateli.

3.2 Kontrolní vrstvy

Vulkan je navržen pro co nejvyšší možný výkon a nízkou zátěž na ovladač. To ovšem znamená, že se omezily veškeré kontroly na chyby či jejich správa a také se výrazně snížily možnosti programátora pro zpětnou kontrolu. Ovladač v takových případech má tendenci spíše spadnout než vrátit programátorovi chybu. V horším případě proběhne kód s menšími změnami, ale na jiných zařízeních to následně funkční, což může vytvářet větší problém pro kontroly.

Vulkan proto vytvořil rozšíření nazývané kontrolní vrstvy. Kontrolní vrstvy jsou vloženy mezi program uživatele a ovladač grafické karty (viditelné na obrázku 4). Tato kontrolní vrstva má za úkol vytvářet chybějící kontroly, jako například hlídání vstupních parametrů či správu paměti. Další výhodou tohoto návrhu je, že program při vytváření tyto vrstvy obsahuje. Ovšem při vydání aplikace se tyto kontrolní vrstvy úplně odstraní a tím se zvýší výkon dané aplikace. Programátoři si podle vlastního uvážení mohou napsat vlastní kontrolní vrstvy. Vulkan díky spolupráci s LunarG poskytuje standartní kontrolní vrstvy, které jsem využil pro tuto práci. Kromě povolení těchto vrstev je také zapotřebí zavést zpětní volání pro získání chyby.

Jelikož Vulkan nastavení je zcela na programátorovi a vše se musí nastavit od základu, tak se stávají tyto kontrolní vrstvy silným prostředkem pro hledání chyb.

3.3 SPIR-V

Narozdíl od ostatních API, kód programovatelných modulů (shader kód) ve Vulkanu musí být specifikován pomocí bytového formátu, narozdíl od lidsky čitelné syntaxe jako je třeba GLSL (OpenGL syntaxe) a HLSL (DirectX syntaxe). Tento bytový formát je nazýván SPIR-V a je navržen pro Vulkan, ale také pro OpenCL (další produkt skupiny KHRONOS). Je to formát pro psaní programovatelných modulů jak výpočetních, tak i zobrazovacích.

Výhoda užití bytového formátu spočíva v lehce převeditelné formě pro vývojáře grafických ovladaču a karet. V minulosti se ukázalo, že syntaxe, která je lehce čitelná člověkem (např. GLSL), může být problémová, protože vývojáři grafických karet ji mohou interpretovar různými způsoby. U psaní složitější programových modůlu se pak stává, že syntaxe je některou kartou nedokonale zpracována, příchodem bytového formátu by se tomu mělo předejít.

To ovšem neznamená, že kód programovatelných modulů je zapotřebí psát přímo v bytovém formátu. KHRONOS skupina vydala vlastní nezávislý překladač pro překlad formátu GLSL do SPIR-V formátu. Tento překladač je navržen tak, aby ověřil kód v poskytnutém GLSL formátu a převedl ho na binární formát, který se může načíst a použít v aplikaci. Překladač může být načten jako knihovna, aby byla možnost přeložení kódu za běhu programu.

GLSL je jazyk používaný pro tuto práci. Má syntaxi založenou na programovacím jazyku C. Každý shader musí mít metodu main, která se bude invokovat uvnitř programu. Dále místo předávání parametrů pomocí vstupních a výstupních členů. GLSL využívá globální proměnné, aby načítal a odesílal data. GLSL pomáhá programátorovi zabudovanými vlastnostmi důležitými pro programování grafických aplikací. Těmito vlastnostmi jsou struktury typu vecN, kde N představuje dimenzi. Například vec3 je 3-dimenzionální vektor, ve kterém lze k jednotlivým členům přistupovat přes tečkovou konvenci ("vector.x"). Také je možné vytvářet nové vektory ze stávajících vektorů. Například pomocí vec3(1.0, 2.0, 3.0).xz se vytvoří nový vec2 vektor s hodnotami vec2(1.0, 3.0). Dále se dají vectory skládat vec3(vec2(1.0, 2.0), 3.0), což vytvoří vec3(1.0, 2.0, 3.0).

3.4 Bližší popis grafické pipeline

Jelikož grafická pipeline je jednou z nejdůležitějších částí, budu se jí věnovat o něco blíže v této sekci.

Starší API poskytují základní nastavení pro většinu částí při vytváření grafické pipeline. To ovšem pro Vulkan neplatí. Jak již bylo řečeno, u Vulkan API vše závisí na programátorovi. Neboli i základní nastavení se musejí definovat "ručně"v programu.

Každá grafická pipeline musí obsahovatat informace o vstupních parametrech.

1. **Vertexový vstup**
   Struktura VkPipelineVertexInputStateCreateInfo slouží k popisu vstupu pro jednotlivé vertexy do vertex shaderu. Popis je ve 2 krocích:

   - Ukotvení - prostor mezi jednotlivými daty v paměti pro jednotlivé vertexy, instance
   - Popis vlastností - popis jednotlivých vlastností vertexu a jejich odsazení od začátku vertexu.

   Tyto vlastnosti musejí být popsány předem, aby bylo možné pro grafickou kartu uchopit jednotlivé vertexy a jejich vlastnosti. Pokud se například programátor splete v určení jednotlivých vlastností, může to dopadnout špatným vykreslením objektu. Což vyústilo v posun čtení z paměti a kompletního zkreslení objektu. Správně vykreslený objet pak lze vidět na obrázku 8.

   Následný popis pak může vypadat následovně.

```
static VkVertexInputBindingDescription getBindingDescriptions() {
    VkVertexInputBindingDescription bindingDescription = {};
    bindingDescription.binding = 0;
    bindingDescription.stride = sizeof(Vertex);
    bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
    return bindingDescription;
}
```

```
static std::array<VkVertexInputAttributeDescription, 2> getAttributeDescriptions() {
    std::array<VkVertexInputAttributeDescription, 2> attributeDescriptions = {};
    attributeDescriptions[0].binding = 0;
    attributeDescriptions[0].location = 0;
    attributeDescriptions[0].format = VK_FORMAT_R32G32B32_SFLOAT;
    attributeDescriptions[0].offset = offsetof(Vertex, pos);
    attributeDescriptions[1].binding = 0;
    attributeDescriptions[1].location = 2;
    attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
    attributeDescriptions[1].offset = offsetof(Vertex, normal);
    return attributeDescriptions;
}
```

Výpis 2: Úkázka popisu vertexových informací

Obrázek 8: Ukázka správně použitého vertex bufferu

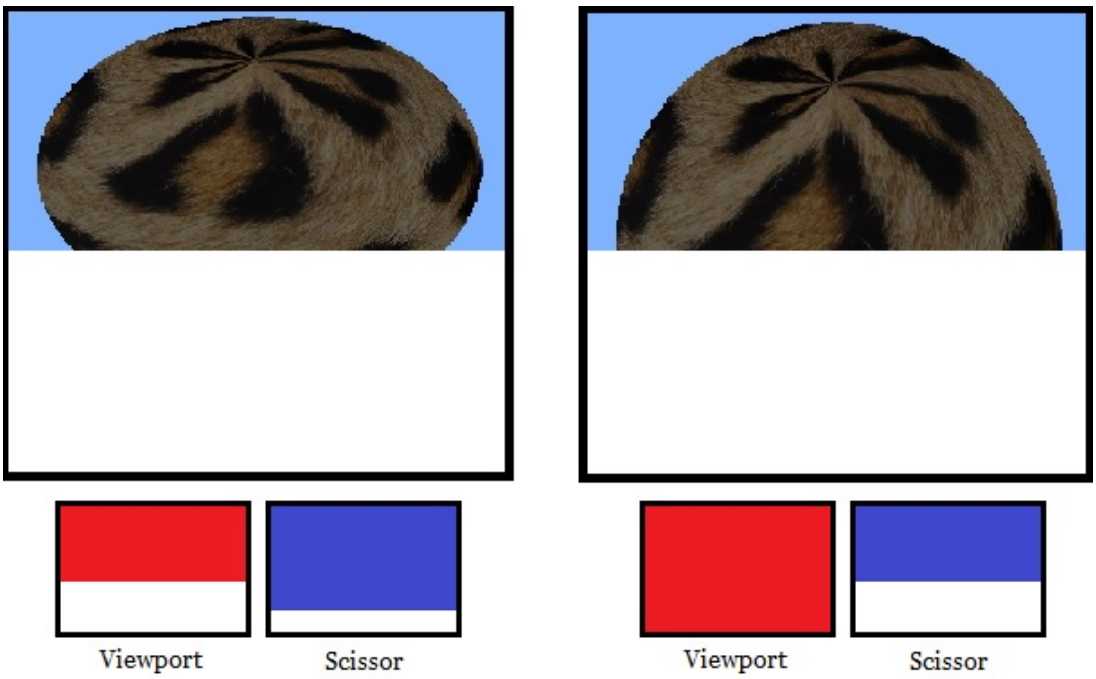
2. **Sestavování vstupu**

   Tato část představuje první blok na obrázku 6. Je to objekt reprezentován pomocí VkPipelineInputAssemblyStateCreateInfo, který určuje, jak se budou zpracovávat vertexy a jestli generovaná primitiva budou restrtovatelná (možné optimalizace). Možné nastavení zpracování vertexů je do bodů, čar, cest (každý druhý vertex je použit pro další na vytvoření cesty),nespojených trojúhelníku nebo spojených trojúhelníků (každý třetí vertex slouží jako vstup pro další trojúhelník).

```
VkPipelineInputAssemblyStateCreateInfo inputAssembly = {};
inputAssembly.sType =
    VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
inputAssembly.primitiveRestartEnable = VK_FALSE;
```

Výpis 3: Úkázka sestavování vstupu

3. **Výřezy a nůžky (Viewports and scissors)**
   Jedná se o nastavení regiónu, do kterého se bude výsledný obraz vykreslovat. Jelikož program nemusí využívat celou plochu okna k vykreslování daných informací vrácených z vytvářené pipeline. Například pro hry to může být případ minimapy, a samotné scény. Viewport označuje región ve frame bufferu, do kterého se bude vykreslovat. Scissor označuje, které části se pomocí rasteriazce ořežou. Názorná úkázka na obrázku 9.



Obrázek 9: Viewport a Scissors

Parametr viewport je defininován pomocí struktury VkViewport a scissors je difinován pomocí VkRect2D. Viewport a scissors jsou následně přiřazeny do struktury VkPipelineViewportStateCreateInfo. Některé karty mohou zpracovávat i více těchto parametru najednou, to je ovšem nutné povolit při vytváření logického zařízení.

```
VkViewport viewport = {};
viewport.x = 0.0f;
viewport.y = 0.0f;
viewport.width = (float)data->swapChainExtent.width;
viewport.height = (float)data->swapChainExtent.height;
viewport.minDepth = 0.0f;
viewport.maxDepth = 1.0f;
VkRect2D scissor = {};
 scissor.offset = { 0, 0 };
 scissor.extent = swapChainExtent;
VkPipelineViewportStateCreateInfo viewportState = {};
viewportState.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
```

```
viewportState.viewportCount = 1;
viewportState.pViewports = &viewport;
viewportState.scissorCount = 1;
viewportState.pScissors = &scissor;
```

Výpis 4: Úkázka viewports and scissors

4. **Rasterizace**

   Proces rasteriazce uchopí vertexové informace poskytnuté pomocí vertex shadru (nebo dalších fází, jako je teselace či geometry shader) a přemění je na fragmenty obrazu, které jsou pak obarveny pomocí fragment shadreru. Také provádí testování hloubky, natočení ploch a scissor test z předchozího bloku. Rasterizace může být nastavena na produkování vyplněných objektů nebo jen hran. Všechny tyto operace se nastavují pomocí struktury VkPipelineRasterizationStateCreateInfo.

   Pro určení, jaká geometrie se bude využívat v grafické pipeline, se musí nastavit rasterizer.polygonMode na jednu z možných hodnot:

   - VK_POLYGON_MODE_FILL : procec vyplní vnitřní část polygonu pomocí aproximace jeho vrcholů
   - VK_POLYGON_MODE_LINE : vrcholy geometrie jsou spojeny pomocí vykreslené přímky
   - VK_POLYGON_MODE_POINT : výsledné vertex jsou vykresleny pomocí bodů (žádné spojení)

   Následně lze v rasterizaci určit šířka čáry. Ta označuje jak "tlustá" bude čára spojující vrcholy. Pokud je zapotřebí vyšší šířky než je 1.0, tak se musí ověřit podpora pro tuto vlastnost při vytváření zařízení pomocí limitních struktur. A dále se musí povolit při tvorbě logického zařízení pomocí wideLines.

   Jako další proměnou je nastavené natočení stran vykreslovaných ploch. Zde je možnost tuto operaci úplně vypnou, ale pro urychlení je lepší pokud se berou jen hrany natočené k pozorovateli. Natočení hran lze provést pomocí vlastností rasterizer.cullMode a rasterizer.frontFace. rasterizer.cullMode může být nastaven na vykreslování přední hrany, zadní hrany nebo obojího. rasterizer.frontFace určuje natočení hrany pomocí pořadí vertexu podle směru hodinových ručiček nebo proti směru hodinových ručiček.

```
rasterizer.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rasterizer.depthClampEnable = VK_FALSE;
rasterizer.rasterizerDiscardEnable = VK_FALSE;
rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
rasterizer.lineWidth = 1.0f;
rasterizer.cullMode = VK_CULL_MODE_NONE;
rasterizer.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
```

```
rasterizer .depthBiasEnable = VK_FALSE;
```

Výpis 5: Úkázka rasterizace

5. **Vícenásobné vzorkování (Multisampling)**
   Vícenásobné vzorkování se provádí zejména pro provedení anti-aliasingu. Anti-aliasing je způsob vyhlazování hran. Ve vulkan aPi funguje tak, že kombinuje několik polygonů zobrazených na stejném pixelu. Tento jev se objevuje zejména na hranách, kde se také zapotřebí nejvíce vyhladit povrch. Diky tomu, že fragment shader se nespouští několikrát, tam kde je jen jeden polygon, tak se tyto operace provedou rychleji. Než vytvářet obraz ve větším rozlišení a následně ho zmenšovat. Tuto funkci je také nutno povolit při vytváření logického zařízení.

```
    VkPipelineMultisampleStateCreateInfo multisampling = {};
multisampling.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO
    ;
multisampling.sampleShadingEnable = VK_FALSE;
multisampling. rasterizationSamples  = VK_SAMPLE_COUNT_1_BIT;
```

Výpis 6: Úkázka vícenásobné vzorkování

6. **Míchání barev**
   Po vytvoření výsledku pomocí fragment shadreru, se musí tento výsledek vložit do frame bufferu. Tento proces se však nevstahuje na pouhé vložení, jelikož by nebylo možné dosáhnou některé efekty, například průhlednosti. Proto se hodnoty múchají. Míchání barev se rozděluje na 2 typy operací:

   - Mixování nové a staré hodnoty na základě matematických operací
   - Mixování nové a staré hodnoty na základě bitových operací

   Pro nastavení michání barev je zapotřebí dvou struktur. První je VkPipelineColorBlendAttachmentState, kde se nastavují vlastnosti pro každý frame buffer zvláště. Druhou strukturou je VkPipelineColorBlendStateCreateInfo. Zde se nachází společné nastavení pro všechny frame buffer objekty.

   Pokud programátor chce jen vkládat musí nastavit colorBlendAttachment.blendEnable na VK_FALSE. Jinak se musí nastavit operace pro výpočet nové hodnoty, která se vloží do frame bufferu.

   Jedním z nejpoužívanějších případů nastavení mixování pro získání průhlednoti je vzorec:

   výslednáBarva.rgb = nováAlfa * nováBarva + (1 - nováAfla) * staráBarva
   výslednáBarva.alfa = nováAlfa

   Pro vytvoření tohoto vzorce v programu je nutné nastavit parametry pro strukturu VkPipelineColorBlendAttachmentState následovně:

```
colorBlendAttachment.blendEnable = VK_TRUE;
colorBlendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
colorBlendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;
colorBlendAttachment.colorBlendOp = VK_BLEND_OP_ADD;
colorBlendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_ONE;
colorBlendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
colorBlendAttachment.alphaBlendOp = VK_BLEND_OP_ADD;
  }
```

Výpis 7: Úkázka mixování barev

Pokud chce programátor využít druho možnosti operací, neboli bitové operace, tak musí povolit tuto funkci v parametru logicOpEnable. Tyto operace pak mohou být specifikovány pomocí pole logicOp. Povolení této možnosti automaticky vypíná první přístup k operacím. Následně se může využít i maska pro určení kterých barevných kanálů se mají bitové operace týkat.

7. **Nastavení dynamického stavu**
   Pokud by programátor chtěl měnit vlastnosti grafické pipeline po vytvoření (ty které měnit lze), tak musí použít a vyplnit strukturu VkPipelineDynamicStateCreateInfo. Tím pádem se dané parametry budou muset nastavovat při nahrávání command buffer objektů a jejich nastavení při vytváření grafické pipeline bude ignorováno. Pokud nechceme měnit parametry po vytvoření pipeline, tak stačí při vytváření dát do kolokny pDynamicState hodnotu nullptr.

```
VkDynamicState dynamicStates[] = {
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_LINE_WIDTH};
VkPipelineDynamicStateCreateInfo dynamicState = {};
dynamicState.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
dynamicState.dynamicStateCount = 2;
dynamicState.pDynamicStates = dynamicStates;
```

Výpis 8: Úkázka dynamického stavu

8. **Načtení shaderu**
   Předtím než se může samotný kód shaderu (načtený bitově) vložit do grafiké pipeline, musí se obalid strukturou VkShaderModule. Tvorba tohoto objektu je v celku jednoduchá. Postačí vložit ukazatel na bitové pole a následně uložit šířku tohoto pole. Šířka je specifikována v bytech a ne bitech na což si programátor musí dávat pozor.

   Nyní jsou vytvořeny moduly, které ovšem nemají ještě žádné označení (vertex, fragment) a nejsou navzájem propojeny. Tyto propojení se dělají pomocí struktury VkPipelineShaderStageCreateInfo. V této struktůře se pomocí parametru shaderStageInfo.stage určuje, o který shader se jedná.

Některé z možných nastavení (neuvádím speciální případy) jsou uvedeny zde:

- VK_SHADER_STAGE_VERTEX_BIT
- VK_SHADER_STAGE_FRAGMENT_BIT
- VK_SHADER_STAGE_GEOMETRY_BIT
- VK_SHADER_STAGE_COMPUTE_BIT
- VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT
- VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT

Dále se v tomto parametru specifikuje, která funkce se bude spouštět jako hlavní (většinou je to funkce main) a to pomocí parametru shaderStageInfo.pName.

```
VkPipelineShaderStageCreateInfo fragShaderStageInfo = {};
fragShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
fragShaderStageInfo.module = fragShaderModule;
fragShaderStageInfo.pName = "main";
```

Výpis 9: Úkázka načítání shaderu

9. **Pipeline rámec (layout)**
   Pro využití dodatečných informací v shader programech, je nutné specifikovat jejich vstupní globální proměnné. Pro tento účel slouží pipeline rámec. Tento rámec je uchován ve struktuře VkPipelineLayout. Tento rámec uchovává informace o uniform buffer objektech a jejich vlastnostech. Bez těchto objektů, by se při každém volání musela celá pipeline neustále měnit, aby bylo možné měnit vstupy pro shadery. Vstupy jsou nejčastěji transformační matice, nebo texturové vzorkovače.

```
VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};
    pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    pipelineLayoutInfo.setLayoutCount = 1;
    pipelineLayoutInfo.pSetLayouts = setLayouts;
```

Výpis 10: Úkázka pipeline rámce

10. **Render pass**

    Předtím než se může vytvořit grafická pipeline, je nutné ještě vytvořit informace o render pass objektu. Informace se týkají frame buffer objektů, hloubkových bufferech, frekvence vzorkování pro každý z objektů.

    Informace jsou obaleny ve strukture VkRenderPassCreateInfo, která se použije pro vytvoření objektu VkRenderPass pomocí funkce vkCreateRenderPass.

    Do struktury VkRenderPassCreateInfo. Se ukládají předem zmíněné informace. Informace o tom, jak naložit s daty ve frame buffer objektu při inicializaci jsou obaleny v VkAttachmentDescription. Zde se nastavuje operace před započetím vykreslování a po ukončení. Možné operace před vykreslením:

    - VK_ATTACHMENT_LOAD_OP_LOAD: zachování dosavadního obsahu
    - VK_ATTACHMENT_LOAD_OP_CLEAR: nahrazení obsahu čistými hodnotami (hodnoty stanovené programátorem)
    - VK_ATTACHMENT_LOAD_OP_DONT_CARE: neidentifikovaný obsah, obsah frame buffer objektu je nemožné využít

    Operace, které je možné provéset po vykreslení:

    - VK_ATTACHMENT_STORE_OP_STORE: hodnoty jsou uchovány, k možnému dalšímu použití
    - VK_ATTACHMENT_STORE_OP_DONT_CARE: kontent se stane deidentifikovatelným po vykreslení (nevhodné pro zobrazení výsledků)

    Vulkan lze využít možnosti kaskadace render pass objektů. Tím lze dosáhnou zvýšení rychlosti programu. Například pro effekt dynamického stínování. Tyto podnože jsou ukládány ve strukturach VkSubpassDescription.

```
std :: array<VkAttachmentDescription, 2> attachments = { colorAttachment, depthAttachment };
  VkRenderPassCreateInfo renderPassInfo = {};
  renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
  renderPassInfo.attachmentCount = attachments.size();
  renderPassInfo.pAttachments = attachments.data();
  renderPassInfo.subpassCount = 1;
  renderPassInfo.pSubpasses = &subPass;
  renderPassInfo.dependencyCount = 1;
  renderPassInfo.pDependencies = &dependency;
```

Výpis 11: Úkázka render pass

11. **Vytvoření grafické pipeline**

Nyní když jsou vytvřeny všechny předešlé postupy, tak se musí zkombinovat do struktury VkGraphicsPipelineCreateInfo. Úkázáno v následujícím kódu.

```
VkGraphicsPipelineCreateInfo pipelineInfo = {};
pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
pipelineInfo.stageCount = shaderStages.size();
pipelineInfo.pStages = shaderStages;
pipelineInfo.pVertexInputState = &vertexInputInfo;
pipelineInfo.pInputAssemblyState = &inputAssembly;
pipelineInfo.pViewportState = &viewportState;
pipelineInfo.pRasterizationState = &rasterizer;
pipelineInfo.pMultisampleState = &multisampling;
pipelineInfo.pColorBlendState = &colorBlending;
pipelineInfo.pDynamicState = &dynamicState;
pipelineInfo.layout = pipelineLayout;
pipelineInfo.renderPass = renderPass;
  }
```

Výpis 12: Úkázka vytvoření pipeline

Protože Vulkan umožňuje dědění grafickách pipeline, tak je možné taky nastavit parametry pipelineInfo.basePipelineHandle, pipelineInfo.basePipelineIndex. Myšlenka za tímto postupem je, že je rychlejší vytvořit grafickou pipeline pomocí dědění a také výměna grafických pipeline s podobnými vlastnostmi se provádí rychleji. Grafickou pipeline lze předat pomocí ukazatele (pipelineInfo.basePipelineHandle) nebo pomocí indexu (pipelineInfo.basePipelineIndex).

Nyni když máme vytvořenou informační strukturu pro grafickou pipeline, můžeme jí vytvořit pomocí metody vkCreateGraphicsPipelines. Tato metoda může příjmat více informačních struktůr najednou a vytvořit tím pádem i více grafických pipeline na jedno zavolání. Dále při volání této metody se může použít uchovávací pamět pro vytvoření zálohy a zpětného použití této zálohy v pozdějších případech. To vede k urychlení programu, jelikož vytvoření grafické pipeline je náročné, vzhledem k rozšířenému nastavení co se musí provádět.

3.5 Bližší popis command bufferu

Příkazy ve Vulkan API, jako jsou například kreslící operace, přenos paměti nejsou spouštěny přímo z přes příkazy. Místo toho jsou zaznamenány do command buffer objektů. Výhoda tohoto přístupu je, že všechny tyto "těžké" operace jsou zaznamenány dopředu a ve více vláknech. Následně se pak musí tyto command buffery spustit v hlavní smyčce programu.

1. **Command pooly**

   Před vytvořením command buffer objektů se musí vytvořit command pool objekt, který je uložen ve struktuře VkCommandPool. Tento objekt spravuje paměť poskytovanou command buffer objektům.

   Každý command buffer objekt, je spouštěn pomocí vložení do fronty zařízení (grafická, prezenční). Každý command pool objekt může uchovávat command buffer objekty, které jsou pro jeden druh fronty. Pro vykreslování slouží grafická rodina. Command pool objekty jsou následně vytvořeny pomocí metody vkCreateCommandPool.

2. **Command buffery**

   Po vytvoření command pool objektů lze vytvořit a nahrát příkazy do samotných command buffer objektů. Command buffer objekty jsou vytvořeny pomocí metody vkAllocateCommandBuffers, která potřebuje popisnou strukturu VkCommandBufferAllocateInfo. VkCommandBufferAllocateInfo popisuje o který command pool objekt se jedná a počet command buffer objektů, které se budou vytvářet. Dále určuje jaké úrovně dané command buffery budou. Možné urovně :

   - VK_COMMAND_BUFFER_LEVEL_PRIMARY: Může být přiřazen do určité rodiny. Nemůže být volán z jiného command buffer objektu.
   - VK_COMMAND_BUFFER_LEVEL_SECONDARY: Nemůže být přiřazen do rodiny front, ale může být volán z command bufferu primární úrovně.

   Druhá úroveň slouží převážně ke zpětnému využití paměťových informací.

   Počátek nahrávání do command buffer objektu je pomocí metody vkBeginCommandBuffer. Tato metoda potřebuje popisnou strukturu VkCommandBufferBeginInfo, která určuje využití command buffer objektu. Možné využití:

   - VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT: command buffer bude znovu nahrán po spuštění
   - VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT: označení druhořadého command bufferu, který bude uprostřed render pass

- VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT: command buffer může být znovu použit, i když již proběhnul

Pokud již byl command buffer nahrán, tak funkce vkBeginCommandBuffer kompletně resetuje celý obsah command bufferu.

Počátek kreslení pak začíná spuštěnm render pass objektu pomocí metody vkCmdBeginRenderPass, která potřebuje popisnou strukturu VkRenderPassBeginInfo. Tato struktura obsahuje informace o vykreslovací oblasti, čistých barvách, použitých frame buffer objektech.

Všechny příkazy použité k nahrávání lze poznat dle prefixu vkCmd. Všechny tyto příkazy vrací void hodnotu dokud nejsou úspěšně nahrány. Prvním parametrem funkcí s prefixem vkCmd je vždy command buffer.

Základní příkazy jsou pak následovné:

- vkCmdBindPipeline: slouží k přiřazení grafické pipeline
- vkCmdDraw: slouží k nakreslení polygonu
- vkCmdBindVertexBuffers: slouží k přiřazení vertex buffer objektů
- vkCmdBindIndexBuffer: slouží k přiřazení index buffer objektů
- vkCmdDrawIndexed: slouží k vykreslení dat ve vertex bufferech pomocí informací v index buffer objektech

Pro ukončení nahrávání se musí ukončit render pass objekt pomocí metody vkCmdEndRenderPass a command buffer pomocí metody vkEndCommandBuffer.

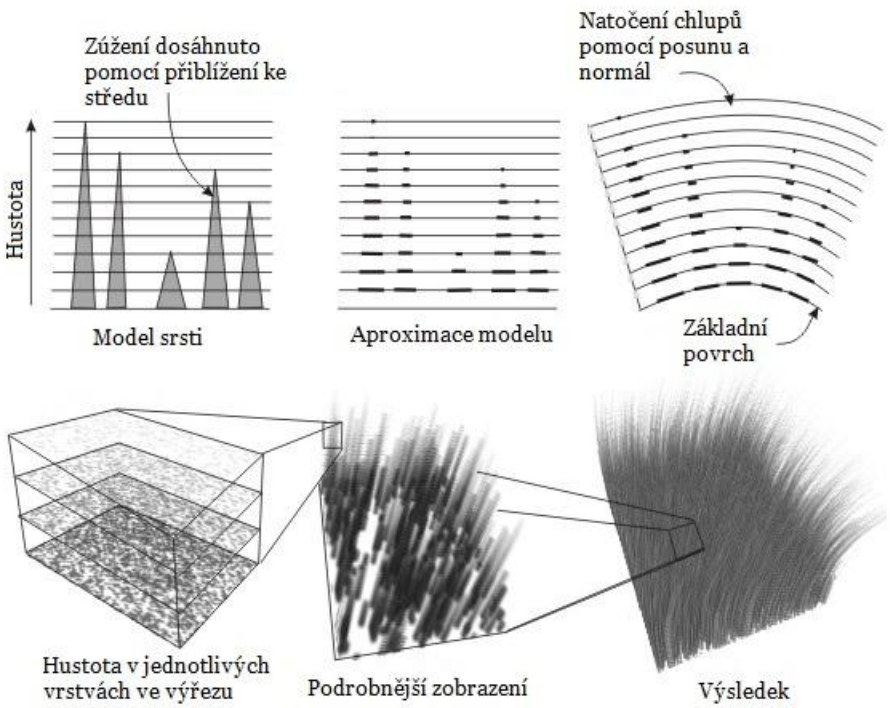
4 Demo

V této práci jsem vytvořil dva druhy Dema. První je na vytížení grafické karty tak, aby bylo možné ohodnotit zatížení framworku vytvořeného v této práci. Demo obsahuje teorii i reálnou konstrukci za vytvářením chlupů v počítačové grafice. Demo není omezeno jen na jeden objekt, ale je použitelné na jakýkoliv model. Druhé demo je vizuální pro ukázání některých možností vytvořeného Frameworku. Jedná se o specifickou scénu, která byla vymodelována za účelem donucení frameworku zpracovávat pohyb objektů, stejně jako zpracovávat větší množství objektů. Sekce je zakončena FPS testama nad samotným frameworkem a jejich shrnutím.

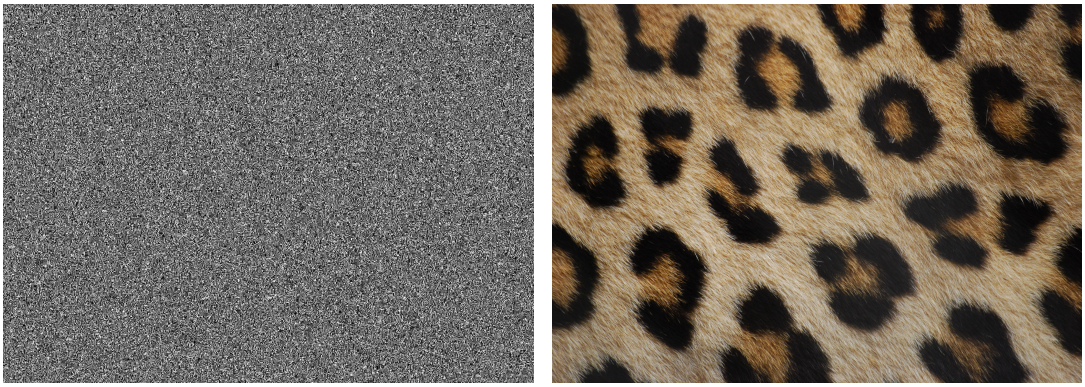
4.1 Zátěžové demo

Jako zátěžové demo jsem vytvořil generování srsti. Ke generování srsti za běhu je zapotřebí mít geometry shader. Jelikož právě geometry shader umožňuje generaci nových vertexů za běhu programu.

Způsob, který jsem si pro vykreslování zvolil je "vrstvení", nebo-li každý chlup ze srsti se rozdělí do několika vrstev. Skládáním těchto vrstev na sebe a jejich "zužováním" se následně docílí efektu jednotného chlupu. Tento způsob je samozřejmě velmi náročný na výpočty, jelikož při každém volání nového snímku se musí tyto vertexy znovu vypočítat. Nicméně k testu pro vytížení grafické karty se tento proces dá využít.



Obrázek 10: Teorie pro chlupy [17]



(a) Textura šumu (b) Barevná textura

Obrázek 11: Použité textury

Na obrázku 10 lze vidět postup za vytvářením chlupů. Nebo-li v prvním kroku potřebuji mít určenou výšku chlupů. Chlupy zvířete či člověka nemají všechny stejnou délku. K tomu jsem využil šum (obrázek 11a). Jelikož shader bude generovat stejně dlouhé chlupy, tak díky předurčené hodnotě na textuře (hodnoty se nachází různě v rozmezí <0,1>), můžu získat srst různé délky, tak aby vypadala nahodile. Stačí přenásobit velikost chlupu hodnotou získanou z textury obrázku.

Každý chlup v mém případě roste nad trojúhelníkem vertexů, které příjdou z vertex shadru do geometry shadru. Nejdříve vypočítám středový vertex průměrováním pozic mezi vertexy trojúhelníku. Dále vezmu tento středový vertex a pro každý vrchol trojúhelníku spočítám vektor posunu do středového vertexu, tak že daný vrchol odečtu od středového bodu. Tím získám vektor posunu, který využiji k vytváření chlupu do výšky. Výška ať je jakákoliv, se dá namapovat na rozmezí <0,1>. Toho využiji a získaný vector budu přičítat k vertexu po násobením aktuální výšky.

Díky tomuto postupu dostaneme obrázek, který má ovšem konstantní barvu. A jednotlivé chlupy v něm nejsou zcela vidět. Viditelnost chlupů je možné zlepšit přidáním stínů. Stíny můžeme vypočítávat dynamicky pro každý objekt. Podobného effektu se dá dosáhnout i úvahou, že čím je vertex hlouběji v srsti, tím méně světla může k němu proniktnout a u kůže se tím pádem vyskytuje tmavší barva. Díky tomu stačí texturu chlupu ztmavit. Úroveň ztmavení závisí na hloubce v srsti. To lze vidět na obrázku 12a, kde kořen (samotný objekt) je o poznání tmavější než ostatní případy z obrázku 12.

Dále je potřeba umět s chlupy pohybovat. Mohl by se vytvořit náhodný pohyb pomocí předpočítaného vektoru pro směr posunu (využit pro zúžení chlupu). Důsledkem by mohlo být, že by se každý chlup pohyboval v jiném směru. Což ve většině případů není vhodné. Například pro simulaci větru či gravitace, není vhodné aby se každý chlup choval jinak. Proto jsem si vzal vector normály a vytvořil k němu kolmici za pomocí mnou zvoleného vectoru. Tím dosáhnu směru kterého chci, abych mohl simulovat vítr či jíné jevy.

Posléze už jde jen o jednoduché nanesení textůry pomocí fragment shadru. Ten má za úkol přečíst barvu z texturové jednotky (obrázek 11b) a ztmavit jí pomocí předem popsaného postupu. Také zde je možné provést Phonguv osvětlovací model, pro ještě lepší effekt stínů.

Následuje ukázka samotného Geometry shadru. "FUR_LAYERS" je předdefinovaný počet vrstev, malé "fur_lenght" je získáno z textury šumu. Velké "FUR_LENGHT" je pevně předdefinované pro model, kvůli možnému zmenšení pomocí modelové matice. WindForce určuje sílu větru a také roste exponenciálně. Z toho důvodu, aby výše položené části se ohýbali více ve větru a kořen se pokud možno nehýbal.

```
    const float FUR_DELTA = 1.0 / float(FUR_LAYERS);
    for(int i = 0; i < gl_in.length(); i++)
    {
        medianPoint += vec3(gl_in[i].gl_Position);
    }
    medianPoint /= gl_in.length();
    for (int furLayer = 0; furLayer < FUR_LAYERS; furLayer++)
    {
        windForce = furLayer/float(FUR_LAYERS);
        windForce *= windForce * (sin(time[0]) - cos(time[0]))/(4/FUR_LENGTH);
        for(int i = 0; i < gl_in.length(); i++)
        {
            v_normal = normalize(normal[i]);
            vec4 usedPoint = gl_in[i].gl_Position * (1.0 - d) + (vec4(medianPoint,1.0) * d);
            usedPoint.a = 1;
            usedPoint += vec4(cross(v_normal.xyz,vec3(0,1,0)),0) * windForce;
            gl_Position = compute_MVP * (usedPoint + vec4(vnormal * d * FUR_LENGTH * fur_lenght,
                0.0));
            EmitVertex();
        }
        d += FUR_DELTA;
        EndPrimitive();
    }
}
```

Výpis 13: Část kódu Geometry shader pro výpočet chlupů

(a) 1 vrstva

(b) 10 vrstev

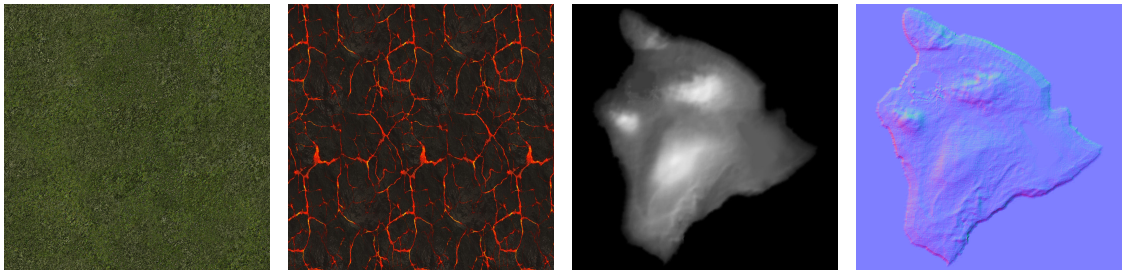
(c) 25 vrstev

(d) 40 vrstev

Obrázek 12: Úkázka zátěžového dema

4.2 Vizualní demo

Jako vizuální demo jsem zvolil objekt teraformující zem na louku. Tento objekt je v mém případě kulička. Tento objekt se pohybuje po spirálové ose. Za objektem se postupně teraformuje země z lávové plochy na zelenou krajinu. Po určitém čase se tento proces zvrátí a objekt zase "ničí" louku kterou vytvořila. Dokud se nedostane do počátečního bodu a celý proces se opakuje.



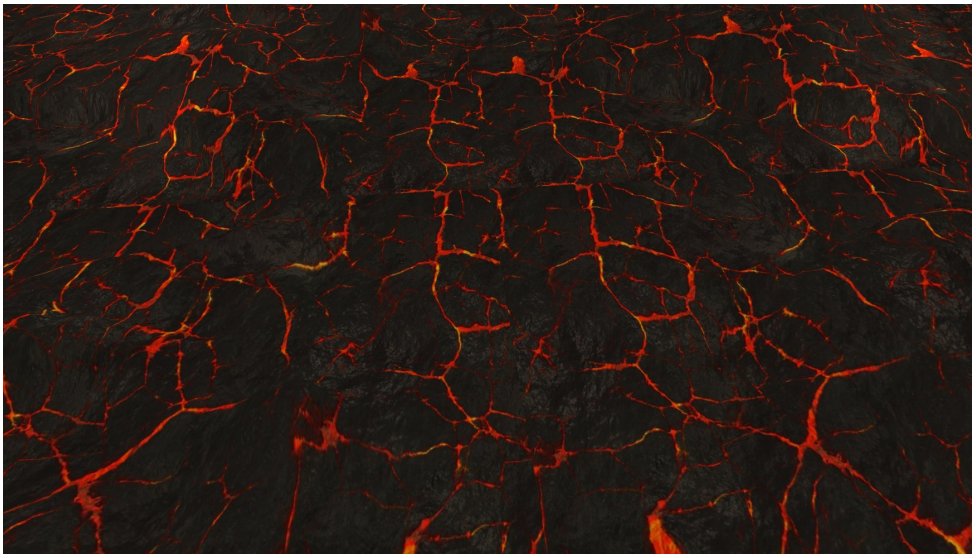
(a) Textura trávy (b) Textura lávy (c) Výšková mapa (d) Normálová mapa

Obrázek 13: Výsledná ukázka

Demo jsem zvolil, protože se zde dá využít více shadru a také protože po vizuální stránce vypadá sofistikovaněji. Stejný effekt, ale ve větším měřítku, byl využit ve filmu Moana.

Scénu jsem začal modelovat vrstvou "země". Povrch ovšem by neměl být úplně rovný. Mohl jsem si zvolit vybrat několik způsobů úpravy povrchu. Buď si vytvořím/stáhnu model nějaké krajiny, bez jakéhokoliv prostředí nebo využiji výškové mapy pro kalkulaci modelu na jakémkoliv povrchu budu chtít. Pro zlepšení synchronizace mezi shadry jsem zvolil druhý přístup, modifikováním modelu za běhu pomocí výškové a normálové mapy. Jako model jsem využil desku pokrytou trojúhelníky. Desku jsem upravil pomocí textury 13c a to tak, že deska má rozměry pevně dané a koordináty v na desce se pohybují od <0,1>. Díky tomu se dá jakákoliv textura (pro jakýkoliv model) namapovat na tuto desku, protože i daná textura má možnost rozdělit svoje rozměry ať pro výšku či šířku na rozměry <0,1>. Pokud by se stalo, že by model přesáhl dané koordináty a ukazoval například na hodnotu 1.1, tak by karta reagovala podle nastavení vzorkovací jednotky. Pokud by se vzorkovací jednotka nastavila na příkaz opakování (repeat), tak by vzorkovací jednotka načetla hodnotu na pozici 0.1. Pokud by se ovšem vzorkovací jednotka nastavila na upnutí, vzala by jednotka hodnotu z pozice 1.0. Ještě je zde možnost zrcadlového zobrazení. Pro oba případy by to znamenalo vzorkování z opačné strany, neboli pro případ "zrcadlového upnutí" se vezme hodnota 0.0 a pro případ "zrcadlového opakování" se vezme hodnota 0.9.

Po procesu modelování pomocí výškové mapy, kdy jsem posunul vertexy po ose Y ve směru do kladných čísel podle hodnoty navzorkované z textury, stačí pouze nanést texturu 13b. Tím se dostane základní podložka, po které se můžou pohybovat objekty nebo se může plocha osazovat objekty (rostlinami/ květinami). Výsledek této operace lze vidět na obrázku 14. Při pozorném zkoumání lze rozeznat náznaky vyvýšených míst i náznak pravidelného opakování textury.



Obrázek 14: Úkázka plochy

Opakování textůry a její viditelnost je problém, protože to ukazuje na nepřesnost vykreslování. Tyto rysy se objevují jen v případě opakovaného použití stejné textury. Tyto vady se dají odstranit i programově. Texturové plochy se budou pronásobovat mezi sebou pro vytvoření jiných vzorců a tím i zvětšení plochy bez opakování. Stejného výsledku dosáhnu i pomocí jedné texturovací jednotky a několika vzorků při jiné frekvenci. Například když každý vzorek posunu o dvojnásobek jeho původních koordinátů. První vzorek bude tím pádem na pozici (0.4,0.7) další na pozici (0.8,0.4), je-li vzorkovací jednotka je nastavena do režimu opakování a poslední na pozici (0.6,0.8). Přičemž jsou všechno dvoudimenzionální souřadnice. Na obrázku první souřadnice odkazuje na šířku a druhá na výšku. Následným aproximováním barev na těchto pozicích získám výslednou barvu vertexu. Tato varianta bude mít samozřejmě také opakovací vzor, ale tento by měl být do velké míry potlačen. Další možností je vytvářet symetrické vzory, neboli takové vzory, které na sebe navazují a jsou dostatečně spletité či jednoduché, aby to nevadilo uživateli při sledování obrazu. Takové vzory se využívají například ve hře Minecraft.

K plnému využití terénu ovšem potřebuji znát normálové vektory pro jednotlivé vertexy. Jelikož původní objekt, před modifikací pomocí výškové mapy, vykazuje jiné normály než upravený objekt pomocí výškové mapy. Musí se tedy původní normálové vektory upravit. Tím pádem vezmu-li v potaz mojí plochou desku, namířenou na kladnou stranu Y osy (neboli vzhůru), tak nemusím dělat žádné úpravy, abych dostal žádoucí normálové vektory. Stačí vzít normálové vektory z normálové textury (obrázek 13d). Pokud bych ovšem chtěl použít normálovou a výškovou mapu na jakýkoliv jiný model, musel bych zvolit jiný postup. Ten by musel zahrnovat aproximaci nových normálových vektorů se starými normálovými vektory a následný posun ve směru starého normálového vektoru podle jeho vzdálenosti. Protože normálové vektory pro aproximaci musejí být normalizované a tím se ztrácejí informace o jejich délce.

Takto upravený terén můžu využít k effektu teraformace. Nejprve jsem zvolil jak objekt

teraformovat. Teraformace mohla být například provedena jako rozrůstající se pláně v čase nebo mohla být teraformace provedena na základě pohybujícího se objektu (mnou zvolená možnost). Jako objekt poslouží kulička, která se bude pohybovat po terénu a rozsévat zeleň, nebo v případě zpětného chodu ji "ničit".

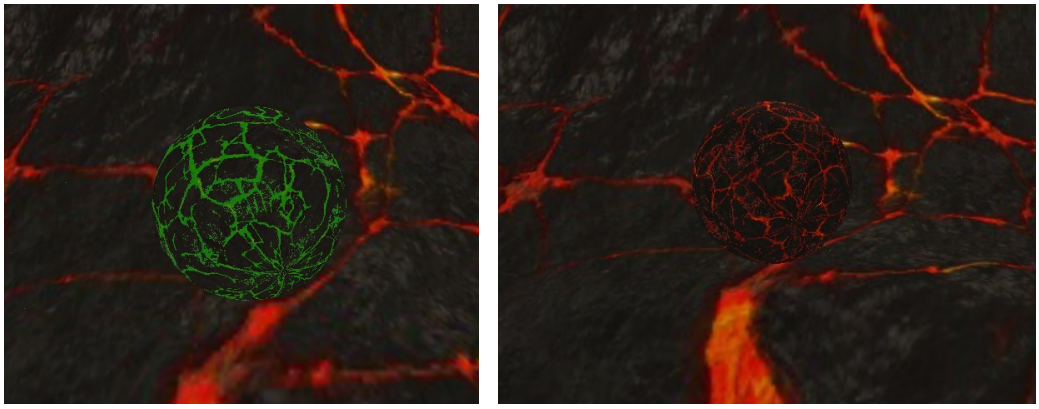
(a) Textůra listu trávy

(b) Textůra listu trávy tmavá

Obrázek 15: Textury listů

Pro teraformaci terénu jsem zvolil osazení pomocí trávy a květin. Tráva je vykreslována pouze pomocí textury viditelné na obrázku 15. Tato textura je nanesena na model ploché desky obdelníkového tvaru. Obdelníkový tvar jsem zvolil z důvodu ušetření velikosti samotné textury. Vytvoření textury čvercového tvaru by znamenalo více zablokované paměti a stejný výsledek vzhledem k tomu, že okolí samotného listu je zanedbáno. Ke správné teraformaci je zapotřebí také využít texturu trávy (obrázek 13a) a to k podpoření efektu zeleně. Vytvořím-li někde list/trs trávy a lze přes něj vidět, tak nesmí být viditelná mnou zvolená lávová plocha. Tím pádem se postupně mění i textura využívaná na povrch terénu. Tyto textury se postupně míchají dokud nepřevládne jedna úplně.

Kulička provádějící teraformaci je vytvářena čistě modelem. Na tuto kuličku jsem nanesl texturu lávy (obrázek 13b). Tuto texturu ovšem měním podle toho jestli je kulička v průběhu rozsevu či ničení zelené plochy. Pokud je kulička v pruběhu rozsevu, tak má kulička místo červených lávových "žil" zelené "žíly". Toho je dosaženo jednoduchým zaměněním pozic v RGB soustavě. Při procesu ničení má kulička klasickou červené "žíly". Kulička také čas od času pulzuje podle uražené vzdálenosti. Fáze kuličky teraformující terén jsou vidět na obrázku 16.



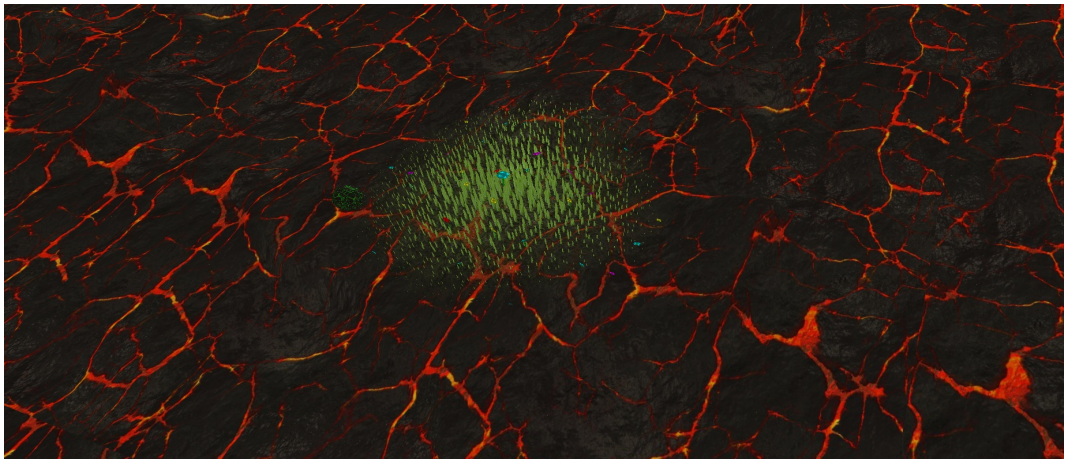
(a) Zelená fáze (b) Červená fáze

Obrázek 16: Fáze kuličky

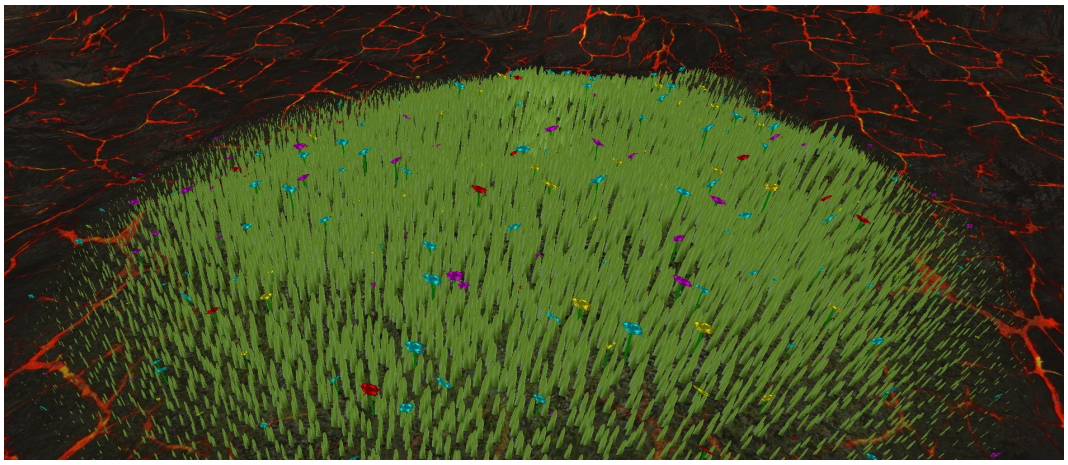
Kulička se teď pohybuje a má na sobě effekt podle jejího účelu. Dále se za kuličkou musí vysazovat zeleň nebo se rozlévat láva. Toho je dosaženo pomocí kombinace mixování terénu (popsáno výše) a růstu opravdových objektů. Kromě listů trávy se vyskytuje ve scéně ještě nemalé množství květin. Pro vytvoření květiny jsem použil model růže. Jelikož na květinu nevyužívám texturu, ale vytvářím její barvy za běhu, tak potřebuji vědět přesné rozměry modelu. To se dá jednoduše zjistit pomocí získání maximální a minimální pozice v modelu na výšku. Když mám tyto informace a vím poměr stonku ke květu, mohu obarvit květinu podle vlastního uvážení.

Květy by neměly mít konstantní barvu. Změna barvy stonku není podstatná, jelikož většina plochy je schována v trávě. Stonek má tím pádem konstantní zelenou barvu. Pro květy jsem využil různé kombinace funkcí sinus a cosinus, kde vstupnímy parametry pro tyto funkce byly rozměry modelu po Z a X ose. Tím získám různé hodnoty barev pro různé části květu.

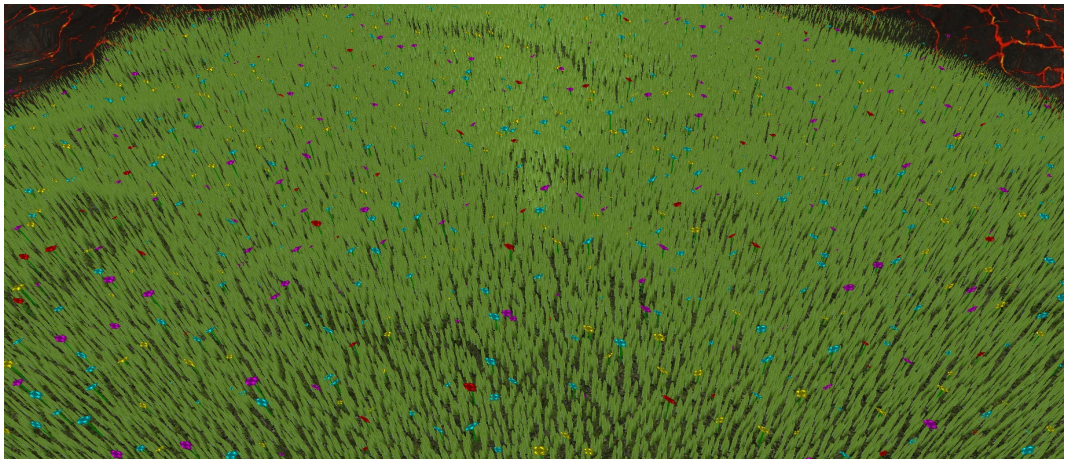
Vše jsem dokončil animací větru s využitím metody podobné v generaci chlupů. V tomto případě zohledním, že všechny rostliny rostou vzůru a tohoto faktu využiji ve svůj prospěch. Vezmu 2 rozměrný vektor větru. Zanedbám směr větru vzhůru či dolů a k pozici vertexu přičtu sílu větru. Sílu ovšem nejdříve vynásobím hodnotou celkové výšky v rozmezí od <0,1>, kdy 0 se vyskytuje na úplném kořeni květiny/listu a 1 na jejím nejvyším bodě. Pokud bych to pronásobil 1x tak bych získal lineární posun, což není zcela ideální. Proto využívám vzorec $výška^2$, jelikož $1^2 = 1$ a tím pádem vertex nikam "neuteče". Zároveň ovšem se vyšší části budou pohybovat více než níže položené části a nejníže položené části se skoro nepohnou.



(a) Počátek



(b) Rozšiřování



(c) Ústup

Obrázek 17: Fáze růstu

4.3 Testy na frameworku

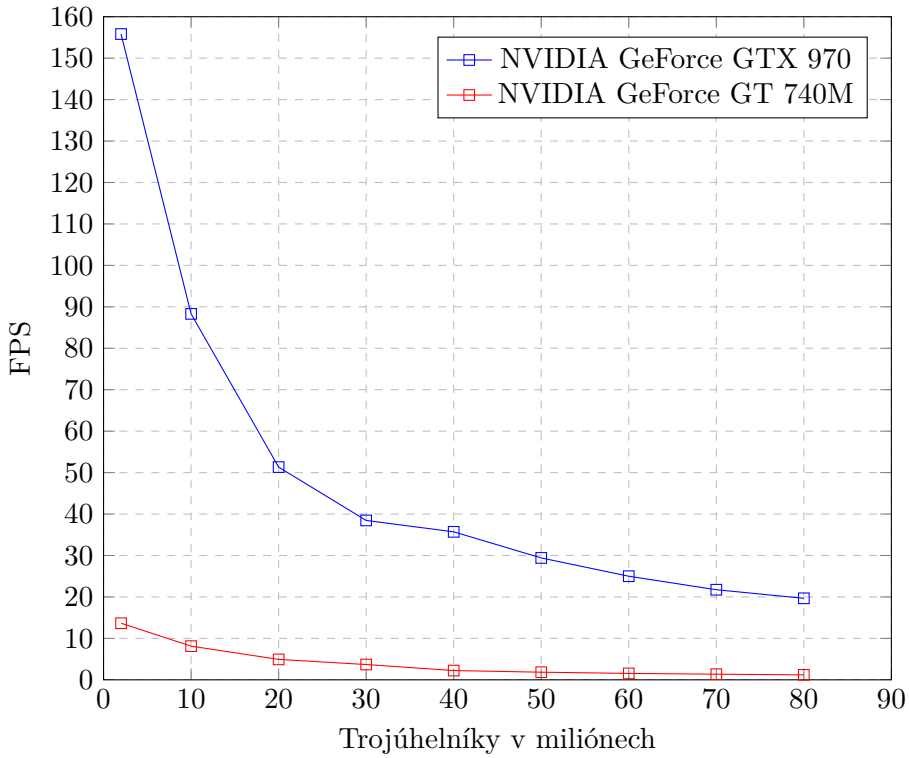
Testy jsou navrženy tak, aby vytížily kartu na co nejvíce procent. Následně se při plném vytížení měří FPS daných aplikací. FPS se přirovnávají k počtu vykreslených trojúhelníků. Zatížení grafické karty bylo měřeno pomocí programu MSI Afterburner v4.3.0 od firmy MSI. Vytížení procesoru jsem měřil pomocí správce úloh v systému Windows 10 na obou počítačích, diagnostický pomůcek obsahovaný ve Visual studiu 2015 a MSI Afterburner (program neumí rozeznávat využití dle programu, ale lze v něm vidět vytížení jednotlivých vláken). Testy bylo provedeny i mimo Visual studio přímým spuštěním.

Test byl prováděn na přístrojích s hardwarem NVIDIA GeForce GTX 970 s procesorem Intel Core i5-3570K 3.4GH a NVIDIA GeForce GT 740M s procesorem Intel Core i7-4700MQ 2.4GH.

Vytížení grafiky pro testy na stroji s kartou NVIDIA GeForce GTX 970 bylo v rozmezí 95-99%, s kartou NVIDIA GeForce GT 740M bylo vytížení v rozmezí 99-100%. Procesor byl vytížen v obou případech v rozmezí 24-25%. Přičemž při bližším zkoumání vytížení na jednotlivých jádrech procesoru, skrze program MSI Afterburner, bylo vidět, že framework nepřetěžuje jen jedno jádro, nýbrž pracuje na všech jádrech stejně, což potvrzuje udávané tvrzení s command buffery.

		NVIDIA GeForce GTX 970			NVIDIAGeForce GT 740M		
Počet vrstev	**Trojúhelníky**	**FPS**	**CPU %**	**GPU %**	**FPS**	**CPU %**	**GPU%**
1	1996800	155.82	24	95	13.66	24	99
5	9984000	88.30	24	97	8.13	24	99
10	19968000	51.33	24	98	4.93	24	100
15	29952000	38.46	24	99	3.69	24	100
20	39936000	35.71	24	99	2.24	24	100
25	49920000	29.41	24	99	1.84	24	100
30	59904000	25.00	24	99	1.55	24	100
35	69888000	21.74	24	99	1.36	24	100
40	79872000	19.67	24	99	1.20	24	100

Tabulka 1: Data z testů



Obrázek 18: Zátěžový test

V grafu (obrázek 18) lze vidět, že zátěžové demo vykazuje exponenciální pokles v závislosti na růstu trojúhelníku. Trojúhelníky rostly v závislosti na počtu přidávaných vrstev. Test byl započat s objektem, který obsahoval dva milióny trojúhelníků a jednou vrstvou chlupů (původním objektem). Následně byly přidávány vrstvy a co pět vrstev bylo měřeno FPS. Tímto způsobem se lineárně zvedala zátěž na grafickou kartu, ale výkon grafické karty klesal exponenciálně.

Vizuální demo obsahuje 167379 objektů a přes 100M trojúhelníků. Toto demo má 40 FPS na kartě NVIDIA GeForce GTX 970.

5 Závěr

Tato práce je implementačního charakteru. Z toho důvodu práce začíná porovnáním stavajících API. Popisem těchto API a jejich částečným srovnáním jsem byl schopen určit některé silné stránky nového Vulkan API, jako je například větší předpokládaný výkon či větší možnost kontroly programu.

Následně jsem vytvořil architekturu, která mi umožňovala oddělovat kód implementovaného frameworku a kód uživatelského programu (dema). Použil jsem zde znalosti z oboru softwarového inženýrství k určení návrhových vzorů potřebných pro tuto práci.

Po návrhu systému jsem přešel na samotnou implementaci. V této části jsem se potýkal s problémem nedostatečného počtu zdrojů. Implementaci jsem nejdříve testoval pomocí vykrelování jednoduchých objektů jako je trojúhelník či krychle. Následně jsem přešel na plnohodnotné modely s textůrovými informacemi a normálovými vektory. Implementaci jsem zakončoval realizací podpory geometry shaderu.

Po implementaci frameworku jsem začal realizovat dema popsané v této práci. Vizualní demo bylo inspirované effektem využitým ve filmu Moana. Zátěžové demo bylo následně vybráno pro účel zatížení grafické karty pomocí výpočtů. Na zátěžovém demu jsem pak provedl sérii FPS testů. Výsledky vyšly podle mého očekávání, nebo-li exponenciálně v závislosti na přibývajících vrstvách.

V této práci jsem se musel naučit práci s technologií Vulkan, GLSL, GLFW a prohloubit znalosti jazyka C++.

V rámci diplomové práce byly splněny tyto cíle:

- Přehled aktuálních API a vizualizačních technologií v oblasti 3D s ohledem na téma práce.
- Analýza, návrh a implementace vlastní aplikace s využitím vybraných technologií.
- Návrh experimentů a výkonnostních testů.
- Zhodnocení experimentů, dosažených výsledků a cílů práce.

V rámci možného rozvoje této práce by bylo možné rozšířit framework o práci s teselačnímy shadery, či podporou výpočetních aplikací. Dále by bylo možné vytvoření editoru či jiného prostředí.

Literatura

[1] OpenGL Overview. OpenGL [opengl.org]. [cit. 2017-03-28]. Dostupné z: https://www.opengl.org/about/

[2] DAVIS, Tom, Dave SHREINER, Mason WOO a Jackie NEIDER. OpenGL: Průvodce programátora. 3. Praha: COMPUTER PRESS, 1999. ISBN 0201604582.

[3] DAVIS, Tom, Dave SHREINER, Mason WOO a Jackie NEIDER. OpenGL: Průvodce programátora. 5. Boston: Longman Publishing Co, 2008. ISBN 9788025112755.

[4] All About DirectX. Webopedia [webopedia.com]. 2005 [cit. 2017-03-28]. Dostupné z: http://www.webopedia.com/DidYouKnow/Hardware_Software/directx.asp

[5] LUNA, Frank D. Introduction to 3D GAME PROGRAMMING WITH DIRECTX 11. 1. Canada: David Pallai, 2012. ISBN 978-1-9364202-2-3.

[6] All About DirectX. Webopedia [webopedia.com]. 2005 [cit. 2017-03-28]. Dostupné z: http://www.webopedia.com/DidYouKnow/Hardware_Software/directx.asp

[7] Vulkan. Khronos [khronos.org]. 2005 [cit. 2017-03-28]. Dostupné z: https://www.khronos.org/vulkan/

[8] SINGH, Parminder. Learning Vulkan. 1. London: Packt, 2016. ISBN 978-1-78646-084-4.

[9] Kessenich, John M.: Vulkan programming guide : the official guide to learning vulkan. S.l: Addison-Wesley, 2016. ISBN 978-0134464541

[10] Vulkan Tutorial. Vulkan Tutorial [online]. [cit. 2017-03-19]. Dostupné z: https://vulkan-tutorial.com/Introduction#page_About

[11] LO, Raymond C. H. a William C. Y. LO. OpenGL Data Visualization Cookbook. 1. United Kingdom: Packt, 2015. ISBN 978-1782169727.

[12] OpenGL Rendering Pipeline. Songho [online]. Song Ho Ahn, 2005 [cit. 2017-04-14]. Dostupné z: http://www.songho.ca/opengl/gl_pipeline.html

[13] Graphics Pipeline. Microsoft [online]. [cit. 2017-04-14]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882(v=vs.85).aspx

[14] KHRONOS Goup. Vulkan TM Overview. [online]. 2016 [cit. 2017-04-14]. Dostupné z: https://www.khronos.org/assets/uploads/developers/library/overview/vulkan-overview.pdf

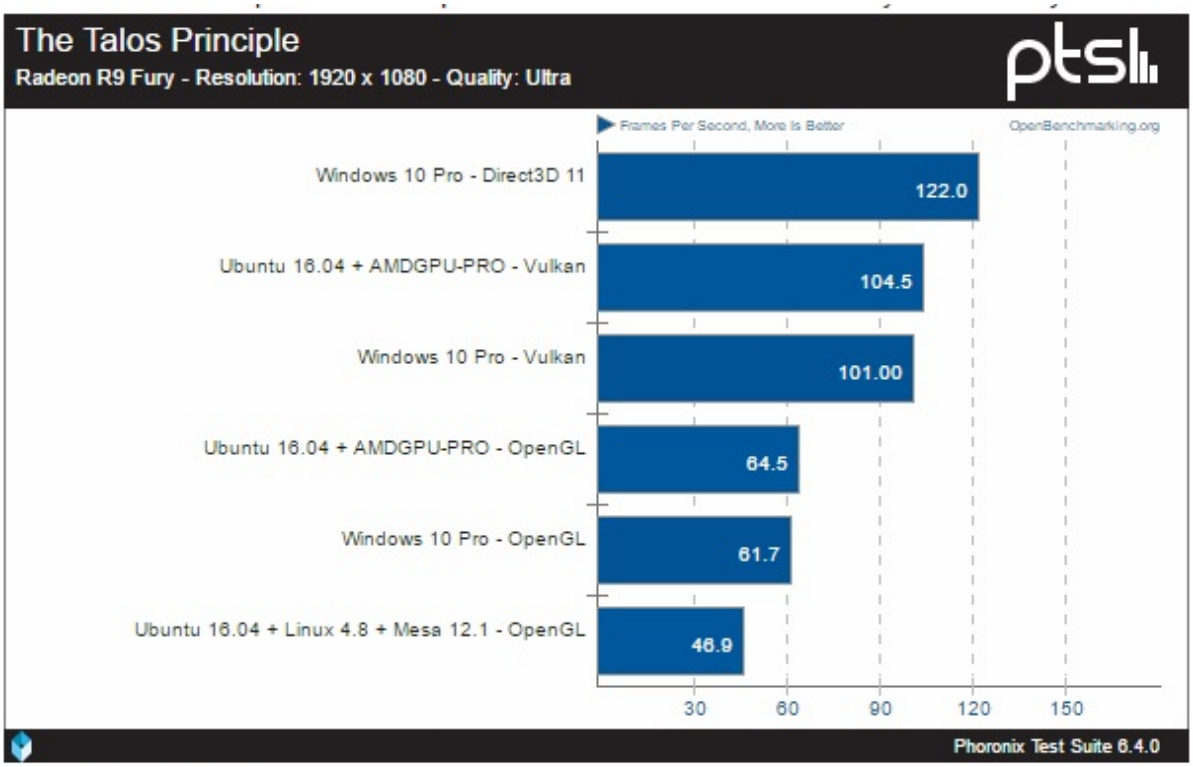
[15] SPIR-V Specification. KHRONOS Goup [online]. 2017 [cit. 2017-03-19]. Dostupné z: https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf

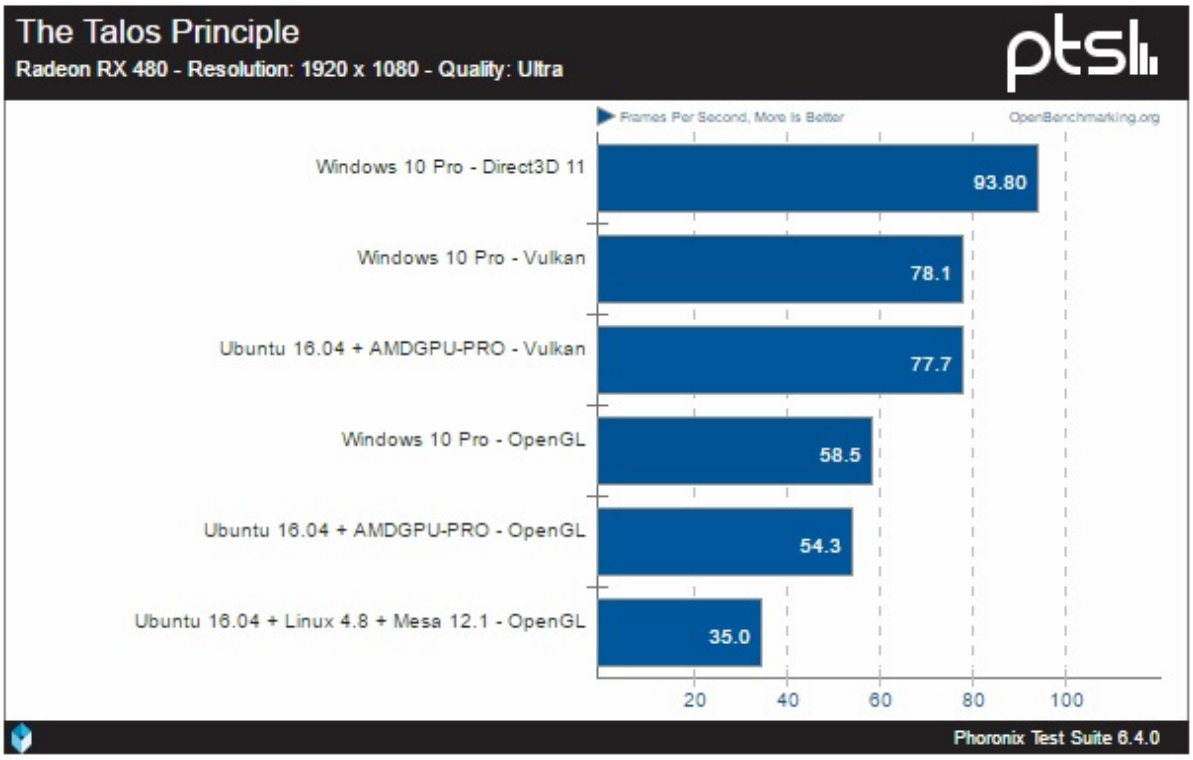
[16] Direct3D 11 vs. OpenGL vs. Vulkan Radeon Benchmarks On Windows, Linux. Phoronix [phoronix.com]. Radeon [cit. 2017-03-28]. Dostupné z: https://www.phoronix.com/scan.php?page=news_item&px=D3D11-OGL-Vulkan-Radeon-AUG

[17] Fur Effects. Xbdev.net [online]. [cit. 2017-03-19]. Dostupné z: http://www.xbdev.net/directx3dx/specialX/Fur/

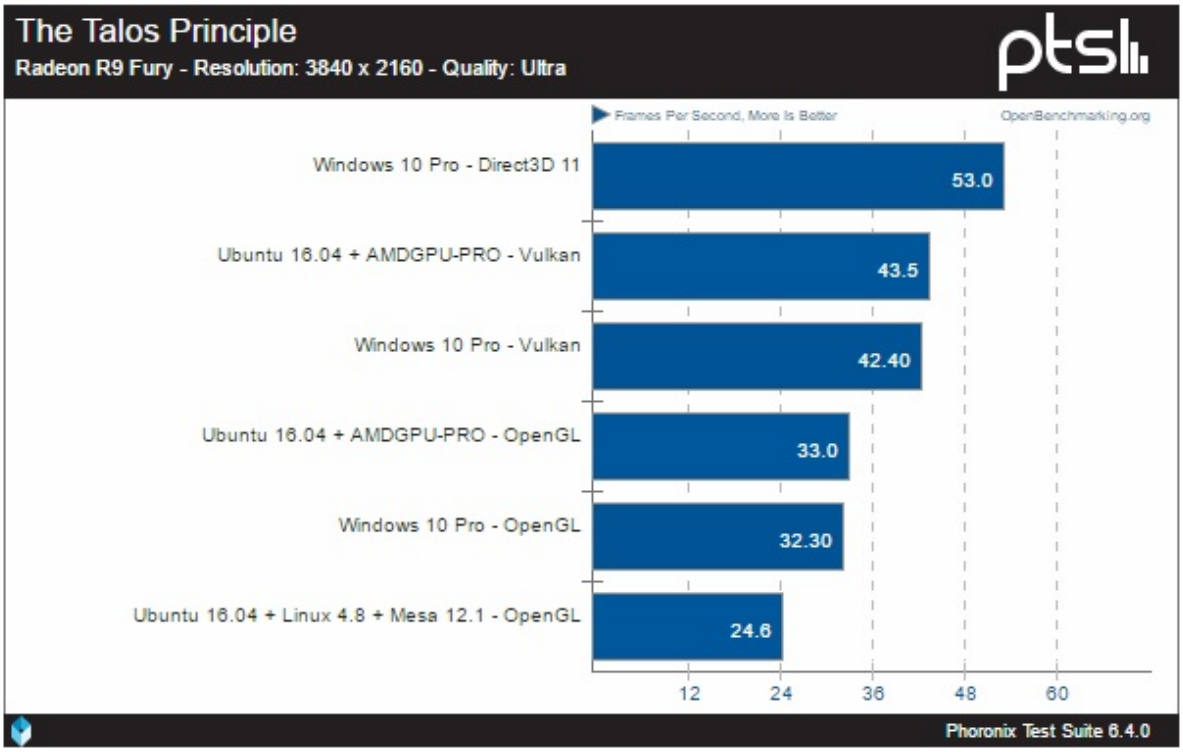
A Přílohy

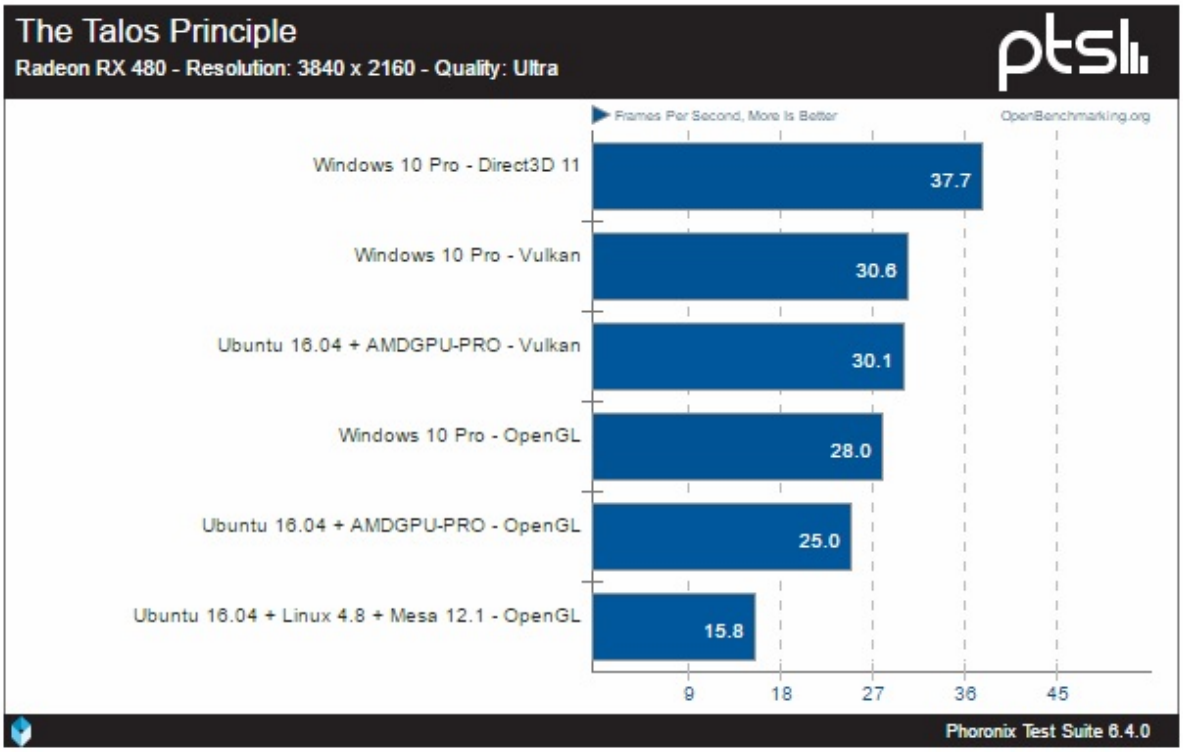
Součástí diplomové práce je CD, obsahující framework s demy a potřebnými součástmi.



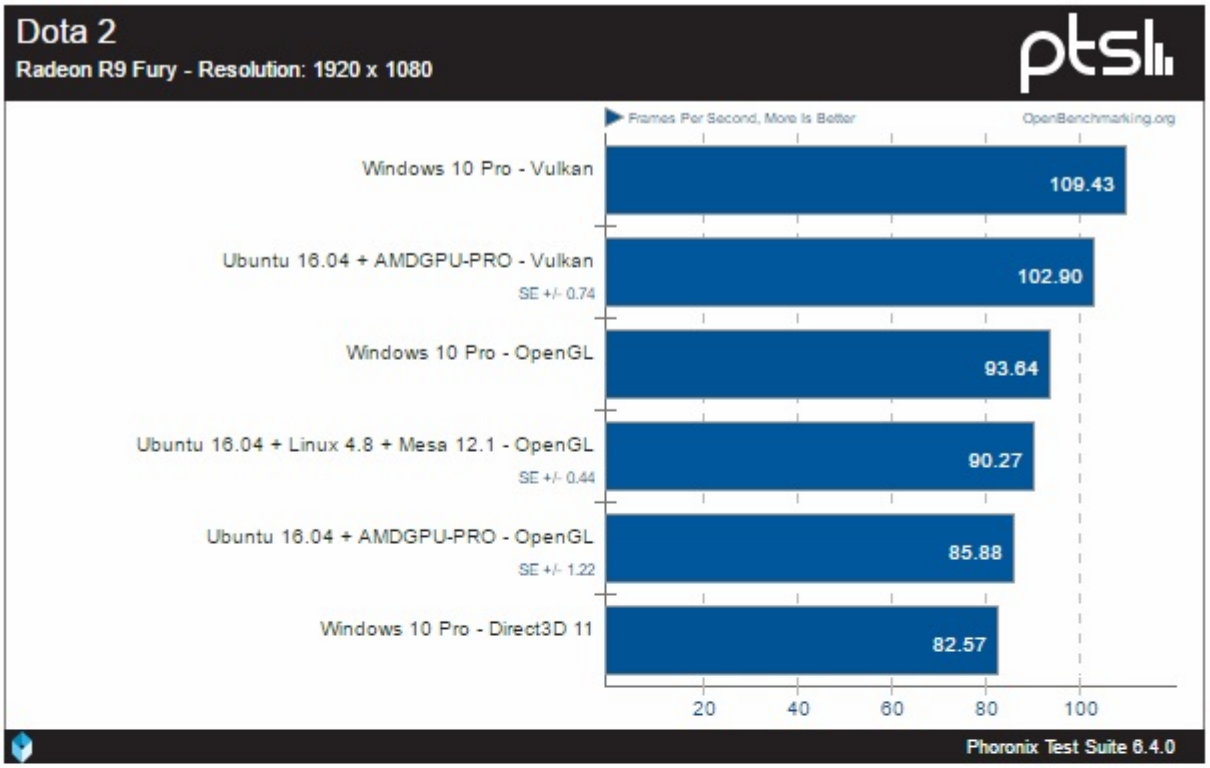


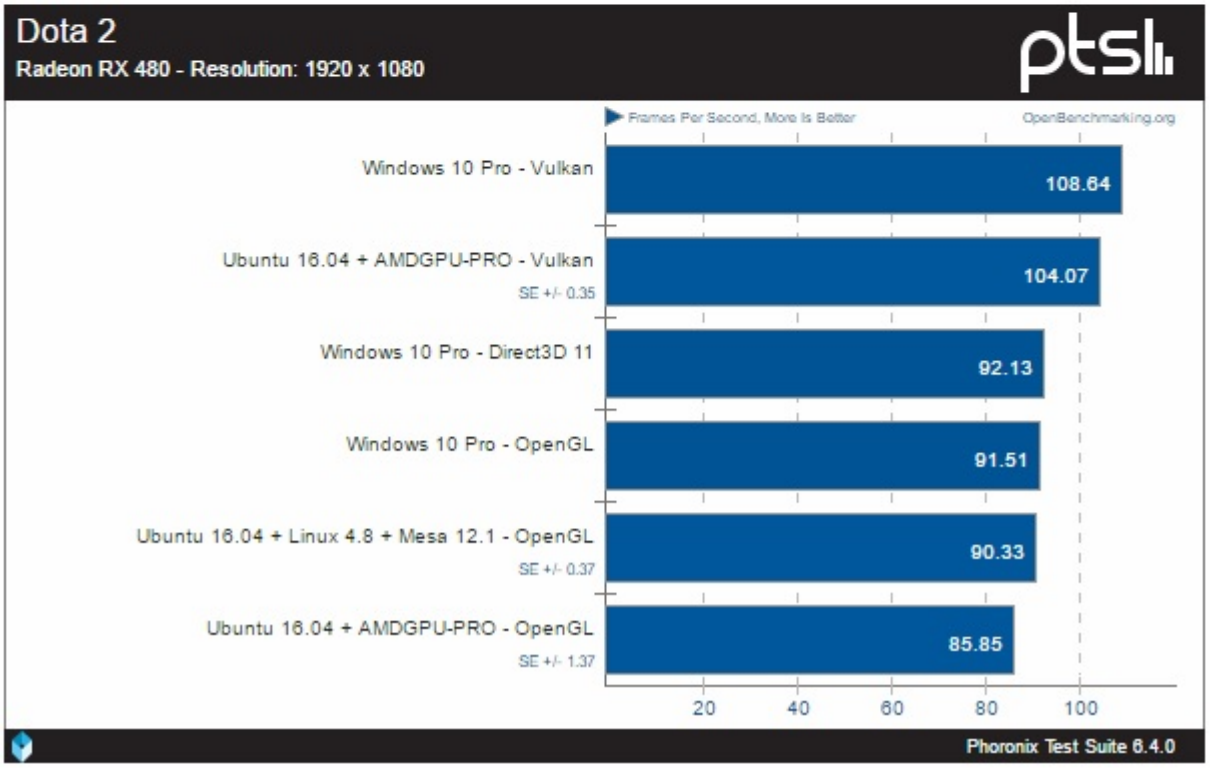
Obrázek 19: Souhrn: The Talons Principle 1920 [16]



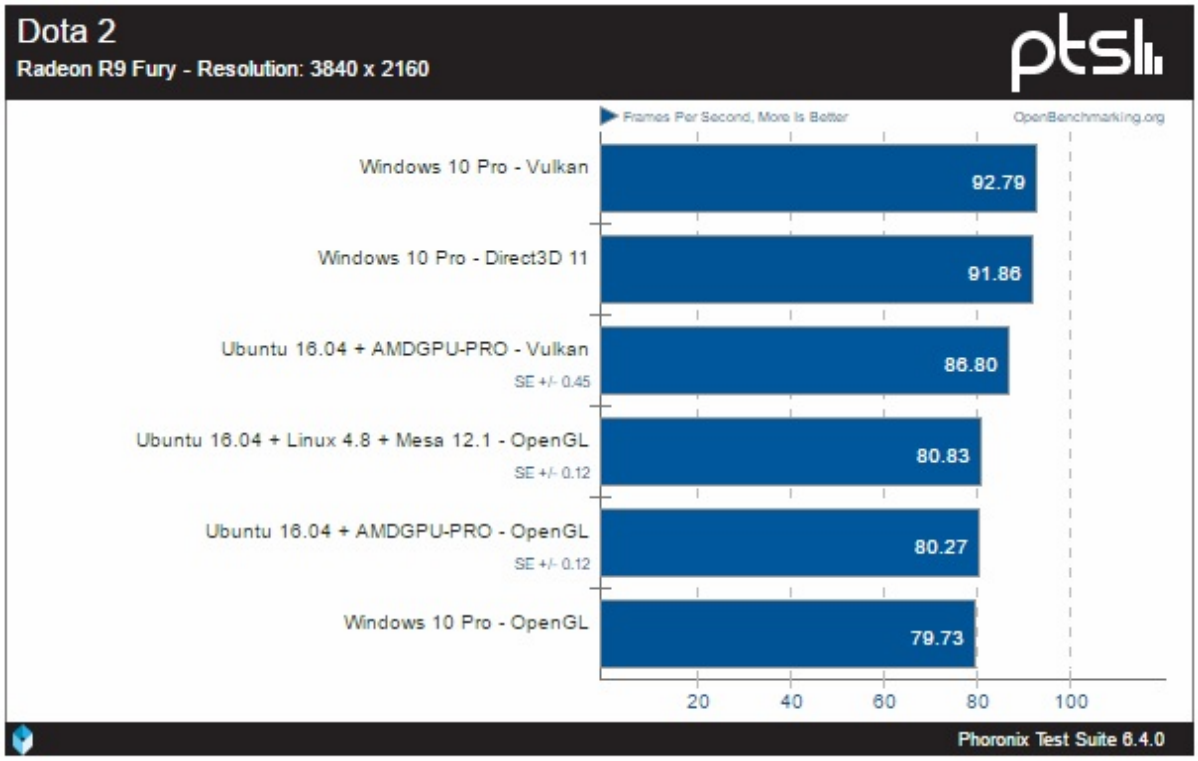


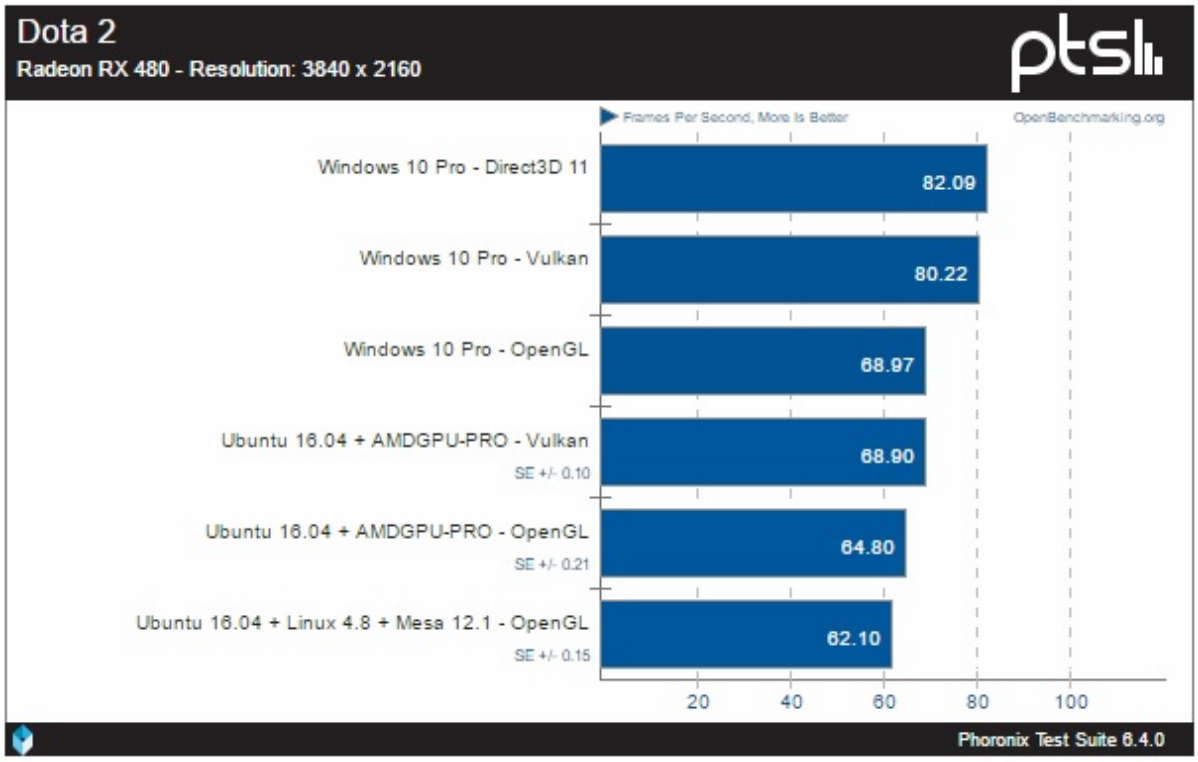
Obrázek 20: Souhrn: The Talons Principle 3840 [16]





Obrázek 21: Souhrn: Dota 2 1920 [16]





Obrázek 22: Souhrn: Dota 2 3840 [16]