Zadání bakalářské práce

Název:	Vulkan API: sada výukových renderovacích scén
Student:	Jozef Valko
Vedoucí:	Ing. Radek Richtr, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Počítačová grafika
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Vulkan je moderní grafický cross platform engine pro 3D grafiku. Pro základní seznámení se s ním je vhodné poskytnout uživateli jak teoretické podklady, tak i praktické příklady, kterých se především v tuzemských jazycích nedostává.

1) Proveďte rešerši Vulkan API a základních matematických a grafických podkladů. Soustřeďte se na srozumitelnost textu pro výukové účely.
2) Analyzujte základní požadavky pro renderer ve Vulkan API.
3) Navrhněte sadu jednoduchých scén vhodných pro seznámení se s Vulkan API.
4) Vybrané scény vytvořte.
5) Vhodně otestujte výsledné scény.

Elektronicky schválil/a Ing. Radek Richtr, Ph.D. dne 6. února 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalárska práca

Vulkan API: sada výukových renderovacích scén

Jozef Valko

Katedra softwarového inženýrství
Vedúci práce: Ing. Radek Richtr, Ph.D.

13. mája 2021

Poďakovanie

Ďakujem svojmu vedúcemu práce Ing. Radkovi Richtrovi, Ph. D. za odborné vedenie bakalárskej práce, cenné rady, pripomienky a čas, ktorý mi venoval. Chcem poďakovať aj rodine a svojím blízkym za podporu, ktorú mi poskytovali.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Prahe 13. mája 2021 …………………

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Jozef Valko. Všetky práva vyhradené.
Táto práca vznikla ako školské dielo na FIT ČVUT v Prahe. Práca je chránená medzinárodnými predpismi a zmluvami o autorskom práve a právach súvisiacich s autorským právom. Na jej využitie, s výnimkou bezplatných zákonných licencií, je nutný súhlas autora.

Odkaz na túto prácu

Valko, Jozef. *Vulkan API: sada výukových renderovacích scén.* Bakalárska práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Bakalárska práca sa zaoberá knižnicou Vulkan API a jej praktickým využitím. Cieľom je pritom zmapovať často využívané funkcionality knižnice Vulkan API. Ďalším cieľom je uviesť čitateľa do niektorých pojmov používaných v 3D grafike a rozobrať niektoré problémy, na ktoré môže čitateľ pri programovaní naraziť. Záujemca o vytvorenie 3D aplikácie, získa základný prehľad nad touto knižnicou a dostane návrhy na riešenia základných problémov. Navyše získa zkompilovateľné projekty, z ktorých bude môcť svoju aplikáciu ďalej vyvíjať. Hlavný prínos práce spočíva v analýze funkcionalít Vulkan API a následnom vyvodení poznatkov z použitých funkcionalít. Tieto poznatky uvedú čitateľa do problematiky a bude možné využiť, pri programovaní už konkrétnej 3D aplikácie, či pre výuku. Prínos praktickej časti spočíva v ukážkach kódu, ktoré čitateľovi priblížia funkcionality Vulkan API, a ktoré môže ďalej využiť vo svojej aplikácií.

Klíčová slova Vulkan API, počítačová grafika, renderovanie rásterizáciou, C++, Windows OS

Abstract

This bachelor thesis deals with the Vulkan API library and its practical use. The goal is to map frequently used functionalities of the Vulkan API library. Another goal is to introduce the reader to some of the terms used in 3D graphics and to discuss some of the problems that the reader may encounter when programming. Those interested in creating a 3D application will get a basic overview of this library and receive suggestions for solving basic problems. In addition, the reader will get compileable projects, from which he will be able to further develop his application. The main contribution of the work lies in the analysis of Vulkan API functionalities and the subsequent derivation of knowledge from the used functionalities. This knowledge will introduce the reader to the issue and it will be possible to use when programming a specific 3D application, or for teaching. The benefit of the practical part lies in the code samples, which will introduce the reader to the functionality of the Vulkan API, and which he can further use in his application.

Keywords Vulkan API, computer graphics, rasterization rendering, C++, Windows OS

Obsah

Úvod **1**

1 Cieľ práce **3**

I Teoretická časť **5**

2 Analýza súčasného stavu riešenia problému **7**

3 Matematické podklady **9**
3.1 Vektory 9
3.2 Matice 13
3.3 Transformačné matice v počítačovej grafike 15
3.4 Tangenty, bitangenty a normály 18
3.5 Kvaternióny 19

4 Počítačová grafika **23**
4.1 Grafické karty 23
4.2 Rendering 24
4.3 Rásterizér 26
4.4 Pipeline 26
4.5 Shader 27
4.6 Buffer Objekty 29
4.7 Depth buffer 30
4.8 Scéna, level, model, mesh 31
4.9 Texture Mapping 32
4.10 Gimbal-Lock 34
4.11 Osvetlenie 34

II Praktická časť **35**

5 Vytvorenie knižnice **37**
5.1 Nastavenie MSVS 2019 37
5.2 Základná schéma nastavenia Vulkanu 40
5.3 Implementácia matematických a pomocných štruktúr 43
5.4 Implementácia inicializácie 45
5.5 Nastavenie rendrovacích prvkov 48
5.6 Implementácia dát rendrovacieho cyklu 52
5.7 Shadery 53
5.8 Načítavanie modelov 55
5.9 Textúry 56
5.10 Pipeline 58
5.11 Vytvorenie dll knižnice 60

6 Implementácia ukážok využitia knižnice **61**
6.1 Základná ukážka 61
6.2 Používanie viacero textúr 63
6.3 Phongovo nasvietenie 65
6.4 Použitie kvaterniónov 68

Záver **71**

Literatúra **73**

A Zoznam použitých skratiek **77**

B Obsah priloženého CD **79**

Zoznam obrázkov

3.1 Perspektívna projekcia. Foto autor. 17
3.2 Ortografická projekcia. Foto autor. 18

4.1 Grafická karta Nvidia GTX 3. radu[1] 24
4.2 Scéna rendrovaná fyzikálne založeným rendererom, Corona Renderer [2] 25
4.3 Scéna rendrovaná realtime GPU rasterizérom, CryEngine v hre Kingdome Come: Deliverance. Foto autor. 26
4.4 Kroky rásterizačneho renderera[3]. 28
4.5 Efekt hmly implementovaný pomocou depth buffera[4] 30
4.6 UV mapovanie textúr[5] 33

5.1 Nastavenie štandardu c++. Foto autor. 38
5.2 Pripojenie dodatočných hlavičkových súborov. Foto autor. 38
5.3 Pripojenie dodatočných knižníc. Foto autor. 39
5.4 Nastavenie mien knižníc. Foto autor. 39
5.5 Roztriedenie hlavičkových a knižničných súborov do priečinkov. Foto autor. 40
5.6 Nastavenie typu konfigurácie. Foto autor. 41

6.1 Textúra používaná v ukážkach[6] 62
6.2 Výstup prvej ukážky. Foto autor. 64
6.3 Výstup druhej ukážky. Foto autor. 65
6.4 Objekt osvetlený Phongovým modelom. Foto autor. 68

Úvod

Počítačová grafika sa vyvíja veľmi rýchlo. Či už spomenieme, pre počítačovú grafiku zostrojený, hardware, alebo aj software, svet počítačovej grafiky sa mení z roka na rok. Grafické karty neustále naberajú na výkone, veď len za posledných 6 rokov sa ich výkon zvýšil niekoľkonásobne a začínajú nachádzať využitie aj v iných odvetviach mimo počítačovú grafiku. Virtuálna realita je stále viac a viac prístupnejšia, pričom výkon a presnosť zobrazovania okuliarov stále rastie. Za posledné dva roky dokonca pribudla rozšírená realita.

Výkon hardvéru teda naberá na výkone. S rastúcim výkonom hardvérových zariadení, nám ale rastú aj možnosti ich využitia. Môžeme ich použiť, a asi najčastejšie využívame, na tvorbu hier alebo interaktívnych aplikácií, ale aj pre detekciu kolízií, simuláciu fyziky na objektoch, 3D vizualizáciu, či pri riešení problémov umelej inteligencie. A presne na tieto účely slúžia grafické knižnice.

V rámci odboru počítačovej grafiky na fakulte informačných technológií, sme sa učili ako programovať s knižnicou OpenGL. OpenGL bolo ale publikované v roku 1992 a i keď sa stále používa pri tvorbe 3D aplikácií, existujú aj iné, v určitých prípadoch môžeme povedať aj, lepšie varianty grafických knižníc, ktoré môžeme pri programovaní v našom projekte využiť. Jednou takou knižnicou je aj Vulkan API, ktorá spadá pod rovnakú spoločnosť, Khronos Group a ponúka nám široké spektrum funkcionalít, s ktorými môžeme pracovať. Predtým sa však musíme oboznámiť, tak ako pri každej inej knižnici, s jej výhodami a nevýhodami, ale aj so spôsobom jej využívania.

Cieľ práce má teda výskumný charakter a pôjde mi hlavne o uvedenie čitateľa do programovania s Vulkan API. Moja práca vďaka tomu poskytne zázemie pre ďalších programátorov, ktorý budú potrebovať zobrazovať dáta v 3D pre ďalšiu vedeckú činnosť, alebo pre programátorov hier, interaktívnych aplikácií, či pre programátorov, ktorý budú chcieť prejsť z inej grafickej knižnice práve na Vulkan. V teoretickej časti budem popisovať funkcionality, ktoré Vulkan poskytuje a uvediem čitateľa do terminológie 3D počítačovej grafiky. Pokúsim sa objasniť niektoré problémy, ktoré vznikajú prácou v 3D.

V rámci praktickej časti sa pokúsim na príkladoch čitateľovi ukázať, najprv ako spojazdniť knižnicu Vulkan a pripraviť ju na ďalšie programovanie, ukážem spôsob vytvorenia a linkovania knižnice DLL a popíšem postupy, ktoré čitateľa dovedú k často potrebným výsledkom.

Pre správne pochopenie tejto práce sú potrebné aspoň základné vedomosti programovania v C++.

KAPITOLA 1

Cieľ práce

Hlavným cieľom tejto práce je zmapovať funkcionality knižnice Vulkan API, ktoré sú v aplikáciách často využívané a pomocou ukážok zdrojového kódu s nimi oboznámiť čitateľa. Teoretická časť sa venuje vysvetleniu niektorých problémov, na ktoré môže čitateľ pri programovaní naraziť a zaoberá sa analýzou pojmov 3D počítačovej grafiky a analýzou.

Cieľom praktickej časti je pomocou ukážok zdrojového kódu, najprv ukázať napojenie Vulkan API do projektu a ďalej ukázať riešenia k problémom uvedením v teoretickej časti práce. V rámci praktickej časti práce, budem postupne vytvárať knižnicu typu DLL, ktorej cieľom bude zmenšenie veľkosti potrebného kódu a jeho utriedenie do celkov, čo spôsobí zlepšenie čitateľnosti a pomôže lepšie pochopiť zapojenie a nastavenie Vulkan API. Posledne je cieľom postupným riešením problémov vytvoriť niekoľko menších spustiteľných projektov, ktoré čitateľovi vysvetlia a priblížia možnosti riešení problémov vo Vulkan API.

Časť I

Teoretická časť

Kapitola **2**

Analýza súčasného stavu riešenia problému

Pri mojom prieskume, som sa často stretával s tým, že podobné práce vysvetlili len triviálne pojmy a zbytok nechali na programátorovej fantázií. Často som našiel podobné práce, ale venované iným grafickým knižniciam, čo je pochopiteľné, keďže Vulkan je stále nová knižnica, publikovaná v roku 2015.

Tutoriál vypracovaný Alexandrom Overvoordeom[7] je práve jedným z týchto prác. V jeho práci, sa nevenuje vôbec vysvetleniu potrebných matematických vedomostí, rovno skočí do programovania a popri tom vysvetľuje ako fungujú funkcie Vulkan API. Na úvodnej stránke ale spomína, že jeho práca je venovaná hlavne programátorom, ktorí sa venujú najmä programovaniu počítačovej grafiky. Ako nutnú znalosť ďalej uvádza pokročilú znalosť jazyka C++ spolu so znalosťou RAII a takisto znalosť základov počítačovej grafiky. Jeho práca mi poskytne solídny základ pri programovaní v praktickej časti práce.

Ďalšia podobná práca, od Joeyho de Vriesa[8], sa zaoberá programovaním počítačovej grafiky, ale za využitia knižnice OpenGL. Vysvetľuje problematiku od základov, a postupne sa prepracuje až k zložitejším problémom, ako je instancing, dokonca rieši aj Parallax mapping. No stále mi v jeho práci chýba úvod, či aspoň zhrnutie potrebných znalostí z matematiky, na čo ale odkazuje na začiatku svojej práce a poskytuje niekoľko referencií, kde môže dané znalosti čitateľ získať. Z hľadiska zdrojových kódov túto prácu teda nebudem môcť využiť, predovšetkým pretože využíva staršiu knižnicu OpenGL, no teóriu, ktorú de Vries zo sféry počítačovej grafiky spomína budem môcť využiť.

V práci budem, na definície, využívať prevažne zdroje poskytnuté v rámci výučby predmetov na Fakulte informačných technológií na ČVUT. Medzi tieto zdroje budú patriť napríklad nepublikované skriptá z predmetu BI-LIN[9], napísané D. Dombekom a kolektívom, a tiež knihu Úvod do algebry, zejména lineární, od Petra Olšáka[10], z ktorých využijem potrebné matematické de-

finície. Pre rozbor problematiky počítačovej grafiky použijem knihu Moderní počítačová grafika, na ktorej tvorbe sa podieľal napr. aj Petr Felkel[11].

Počas praktickej práce budem úzko pracovať s dokumentáciou Vulkan API[12], kde sú popísané všetky funkcionality tejto knižnice.

V rámci teoretickej časti bakalárskej práce sa budem venovať vysvetleniu niektorých pojmov potrebných pre praktickú časť práce. Teoretická časť je rozdelená na dve väčšie celky. Matematika, kde definujem pojmy ako sú vektor, báza či matica, vysvetlím operácie medzi nimi a tým zhrniem čo je nezbytne potrebné pre pochopenie matematických operácií, s ktorými sa stretneme v praktickej časti práce. Druhá časť je venovaná problémom a pojmom počítačovej grafiky. Vysvetlím pojmy ako sú renderer, rásterizér, shader, či čo je to gimbal lock alebo instancing, kde budem už užšie pracovať s Vulkan API.

Kapitola **3**

Matematické podklady

Programovanie grafických aplikácií je veľmi úzko späté s lineárnou algebrou. Na pochopenie niektorých operácií vykonávaných na posun modelu v scéne, jeho zrotovanie, či zväčšenie, resp. zmenšenie, alebo samotné zobrazenie 3D obsahu na 2D obrazovku, je potrebné rozumieť základným pojmom lineárnej algebry a vedieť s nimi pracovať. Najprv preto zjednotím pre každý nutný aspekt potrebné pojmy matematicky správne a čo najviac všeobecne. V počítačovej grafike si však vždy vystačíme s definíciou pre, maximálne štvorrozmerný priestor. Pojmy, ktoré definujeme v rámci tejto sekcie budeme počas praktickej časti prepisovať do zdrojového kódu a často ich budeme používať pri programovaní v 3D prostredí.

3.1 Vektory

Vektory sú nevyhnutnou súčasťou počítačovej grafiky. Môžu v našich programoch uchovávať informácie, ako sú napríklad pozícia objektu v scéne, veľkosť objektu, rotáciu objektu, smery či už, pre scénu základných, objektov ako sú smerové svetlá, alebo smer pohybu.

Existuje viacero pohľadov na vektory. Všeobecne ale platí, že vektor nepopisuje miesto alebo bod v priestore. Vektor popisuje smer a dĺžku, narozdiel od skalárov, ktoré popisujú množstvo alebo veľkosť. Skaláry, slúžia aj na škálovanie vektorov.

Pre definíciu vektoru, ale nestačí povedať, že vektor je n-tica reálnych čísel. Platia preň špeciálne pravidlá pre násobenie, budeme rozlišovať tzv. cross a dot product, sčitovanie, odčitovanie atď. Pre všeobecnosť si však pred samotnou definíciou vektora, musíme zadefinovať pojmy grupa a teleso.

Definícia 1. *Nech M je neprázdna množina a $\circ : M \times M \to M$ binárna operácia. Ak platí, že:*

1. $\forall a, b, c \in M : a \circ (b \circ c) = (a \circ b) \circ c$ *(asociativita),*

2. *$\exists e \in M, \forall a \in M : a \circ e = e \circ a = a$ (existencia neutrálneho prvku),*

3. *$\forall a \in M, \exists a^{-1} \in M : a \circ a^{-1} = a^{-1} \circ a = e$,*

hovoríme, že usporiadaná dvojica $G = (M, \circ)$ je grupa. Ak je naviac $\circ$ komutatívne, tj. $\forall a, b \in M : a \circ b = b \circ a$, hovoríme o Abelovskej grupe. [9, str. 35]

Všimnime si, že v definícií nepoužívame žiadnu konkrétnu množinu akými sú napr. $\mathbb{R}$ alebo $\mathbb{N}$ a ani žiadny konkrétny operátor. Grupa teda len popisuje špeciálny vzťah medzi operátorom(v tomto prípade binárnym), a prvkom množiny. Na to aby sme dvojicu mohli nazvať grupou musí byť dvojica asociatívna a musí mať neutrálny a inverzný prvok voči operácii. Toto nám ale na definíciu vektora stále nestačí. Pomocou grupy vieme riešiť lineárne rovnice s jednou neznámou jednoznačne. My by sme ale potrebovali vedieť tieto rovnice riešiť viaceré a preto si zadefinujeme pojem teleso.

Definícia 2. *Nech M je neprázdna množina. Majme dve binárne operácie $+$ a $\cdot$ definované ako $+ : M \times M \to M$, $\cdot : M \times M \to M$. Ak platí, že:*

1. *$(M, +)$ je Abelovská grupa s neutrálnym prvkom 0 - nulový prvok,*

2. *$(M \setminus \{0\}, \cdot)$ je grupa s neutrálnym prvkom 1 - jednotkový prvok,*

3. *$\forall a, b, c \in M : a(b + c) = ab + ac \land (b + c)a = ab + ac$*

, nazývame usporiadanú trojicu $T = (M, +, \cdot)$ telesom. Ak je navyše $(M \setminus \{0\}, \cdot)$ Abelovská grupa, nazveme trojicu $T = (M, +, \cdot)$ komutatívne teleso.[9, str. 36-37]

A napokon si zadefinujeme základné vektorové operácie nad priestorom vektorov nad nejakým telesom. Naviac si zadefinujeme skaláry, ktoré slúžia na škálovanie vektorov, čo je v angličtine očividné už z pomenovania scalor, odvodené od slova scale.

Definícia 3. *Nech T je ľubovolné komutatívne teleso, jeho neutrálne prvky voči operáciám sčítania resp. násobenia označme 0, resp. 1. Nech je ďalej daná neprázdna množina V a dve zobrazenia $\oplus : V \times V \to V$, $\odot : T \times V \to V$. Povieme, že V je vektorový priestor nad telesom T s vektorovými operáciami $\oplus$ a $\odot$, práve keď platia nasledujúce axiómy vektorového priestoru:*

1. *$\forall a, b \in V : a \oplus b = b \oplus a$,*

2. *$\forall a, b, c \in V : (a \oplus b) \oplus c = a \oplus (b \oplus c)$,*

3. *$\forall \alpha, \beta \in T, \forall a \in V : \alpha(\beta \odot a) = (\alpha\beta) \odot a$,*

4. *$\forall \alpha \in T, \forall a, b \in V : \alpha(a \oplus b) = (\alpha \odot a) \oplus (\alpha \odot b)$,*

5. *$\forall \alpha, \beta \in T, \forall a \in V : (\alpha + \beta) \odot a = (\alpha \odot a) \oplus (\beta \odot a)$,*

6. $\forall a \in V : 1 \odot a = a$,

7. $\exists \theta \in V, \forall a \in V : 0 \odot a = \theta$,

Prvky vektorového priestoru V nazývame vektory(značíme $\vec{v}$), prvky telesa T nazývame skaláry a prvok θ z axiómu 7 nazývame nulový vektor.[9, str. 43]

Bohužiaľ tu nekončíme. Pri našich výpočtoch budeme potrebovať napríklad určiť uhol zvieraný dvoma vektormi, zistiť dĺžku vektora, či kvázi orezať dĺžku vektora pri zanechaní informácie o smere vektora. Najskôr si však zadefinujeme základné vlastnosti skalárneho súčinu(dot product).

Definícia 4. *Buď V vektorový priestor nad telesom $T \subseteq C$. Zobrazenie $\cdot : V \times V \to T$ nazývame skalárny súčin, ak platia pre všetky vektory $\vec{x}, \vec{y}, \vec{z} \in V$ a každý skalár $\alpha \in T$ následujúce axiómy:*

1. $\vec{x} \cdot \alpha\vec{y} + \vec{z} = \alpha(\vec{x} \cdot \vec{y}) + (\vec{x} \cdot \vec{z})$*(linearita v druhom argumente),*

2. $\vec{x} \cdot \vec{y} = \vec{y} \cdot \vec{x}$*(hermitovská symetria),*

3. $\vec{x} \cdot \vec{x} \geq 0$ *a zároveň* $(\vec{x} \cdot \vec{x} = 0 \iff \vec{x} = \theta)$*(pozitívna definitívnosť).*

Dvojicu $(V, \cdot)$ nazývame priestorom so skalárnym súčinom(prehilbertov priestor) a značíme H. [9, str.237]

Definícia 5. *Na T^n definujeme*

$$\vec{x} \cdot \vec{y} = \sum_{j=1}^{n} \xi_j \eta_j, \tag{3.1}$$

kde $\vec{x} = (\xi_1, ..., \xi_n)$, $\vec{y} = (\eta_1, ..., \eta_n)$. Tento skalárny súčin nazývame (štandardným) skalárnym súčinom. [9, str.238]

Skalárny súčin je teda zobrazenie z $V \times V$ do T, resp. funkcia ktorá na vstupe berie dva vektory a na výstup vyhodí skalár. Na to aby sme vedeli zistiť uhol medzi dvoma vektormi to ale stále nestačí a musíme si ešte definovať dĺžku vektora a proces normalizácie.

Definícia 6. *Nech a je ľubovoľný vektor z vektorového priestoru R^n. Dĺžka vektora je zobrazenie z vektorového priestoru V do telesa T definované ako*

$$|\vec{x}| = \sqrt{\vec{x} \cdot \vec{x}} = \sqrt{x_1 + x_2 + ... + x_n}. \tag{3.2}$$

[9, str.238]

K poslednej definícii je už len potrebné dodať že každý vektor, dĺžky 1 ($|a| = 1$) nazývame jednotkový vektor. Jednotkové vektory sú užitočné práve pri získavaní uhlov zvieraných dvoma vektormi, ktoré pred tým musíme upraviť tak aby oba boli dĺžky 1. Tento proces voláme normalizácia.

Definícia 7. *Jednotkový vektor získame pomocou normalizáce vektoru vynásobením nenulového vektoru inverznou hodnotou dĺžky vektoru vzhľadom na súčin nad* $\mathbb{R}$*. T.j.:*

$$\vec{u} = \frac{1}{|\vec{u}|}(u_0, u_1, u_2). \tag{3.3}$$

[11, str.556]

Tu sa konečne dostávame k vzťahu medzi skalárnym súčinom $u \cdot v$ a uhlom dvoch vektorov φ.

Definícia 8. *Označme* $\varphi = (\vec{u}, \vec{v})$ *odchýlku vektorov(tj. uhol, ktorý zvierajú), potom platí:*

$$\vec{u} \cdot \vec{v} = |\vec{u}||\vec{v}| \cos \varphi. \tag{3.4}$$

[11, str.557]

Vzťah budeme ale používať viac vo forme

$$\varphi = \arccos \frac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|} \ , \tag{3.5}$$

ktorú je už jednoduché implementovať. Poslednou potrebnou funkciou je vektorový súčin, alebo cross product, ktorého výsledkom je vektor kolmý na dva vstupné vektory.

Definícia 9. *Vektorový súčin w vektorov u a v je možné pomocou bázových vektorov kartézskej sústavy súradníc i,j,k zapísať v tvare:*

$$\vec{w} = \vec{u} \times \vec{v} = \begin{pmatrix} i & j & k \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{pmatrix} = (u_2v_3 - u_3v_2)\vec{i} + (u_3v_1 - u_1v_3)\vec{j} + (u_1v_2 - u_2v_1)\vec{k}. \tag{3.6}$$

[11, str.557]

Posledným pojmom je lineárna nezávislosť. Povedzme, že by sme chceli pomocou súčtov troch vektorov a ich násobkov získať iný vektor toho istého priestoru, v tomto prípade priestoru $\mathbb{R}^3$. Ak by boli napríklad dva z týchto troch vektorov iba násobkom toho prvého, vedeli by sme vygenerovať len podpriestor, resp. iba podmnožinu, vektorov, ktoré v priestore $\mathbb{R}^3$ ležia. V tomto prípade by to boli body ležiace na jednej priamke a množinu týchto troch vektorov nazývame lineárne závislé vektory.

Definícia 10. *Nech* $(\vec{v_1}, \vec{v_2}, \ldots, \vec{v_n})$ *je súbor n vektorov z priestoru* $\mathbb{R}^m$ *a* $\alpha_1, \alpha_2, ..., \alpha_n$ *sú skaláry z telesa* $\mathbb{R}$*. Vektor*

$$x = \sum_{i=1}^{n} \alpha_i \vec{v_i}, \tag{3.7}$$

nazývame lineárnou kombináciou vektorov $(\vec{v_1}, \vec{v_2}, \ldots, \vec{v_n})$ *s koeficientami* α_1, $\alpha_2, ..., \alpha_n$. *Triviálna lineárna kombinácia vektorov* $(\vec{v_1}, \vec{v_2}, \ldots, \vec{v_n})$ *je taká lineárna kombinácia, kde všetky koeficienty* $\alpha_1, \alpha_2, ...m\alpha_n$ *sú rovné nule. Ak jedine triviálna lineárna kombinácia vektorov* $(\vec{v_1}, \vec{v_2}, \ldots, \vec{v_n})$ *je rovná nulovému vektoru, nazývame súbor vektorov* $(\vec{v_1}, \vec{v_2}, \ldots, \vec{v_n})$ *lineárne nezávislý. [9, str.29]*

3.2 Matice

Ďalšou dôležitou matematickou štruktúrou je napr. matica. V počítačovej grafike nachádza široké uplatnenie najmä pri transformáciách, alebo pri prechodoch medzi bázami. Samozrejme, tak ako pre vektory, aj pre matice platia niektoré špeciálne vlastnosti, ktoré zhrnieme v nasledujúcich pár definíciách. Najprv si ale musíme zadefinovať, čo to matica je, aby sme mohli nad ňou uvažovať o operáciách.

Definícia 11. *Nech* $m, n \in M$. *Usporiadaný súbor mn čísel zapísaný do tabuľky o m riadkoch a n stĺpcoch nazývame matica typu* $m \times n$. *Matice obvykle značíme takto:*

$$\mathbb{A} = \begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ ... & ... & ... & ... \\ a_{m1} & a_{m2} & ... & a_{mn} \end{pmatrix}, \tag{3.8}$$

kde a_{ij} *sú prvky matice (niekedy ich značíme tiež ako* $\mathbb{A}_{ij}$ *a nazývame ich ij-té prvky). Číslu i hovoríme riadkový a číslu j stĺpcový index. Množinu všetkých matíc typu* $m \times n$ *(s reálnymi prvkami) značíme* $(R)^{m,n}$. *Ako* $(A)_{:j} \in (R)^{m,1}$ *značíme j-tý stĺpec matice* (A)*:*

$$\mathbb{A}_{:j} = \begin{pmatrix} a_{1j} \\ a_{2j} \\ ... \\ a_{mj} \end{pmatrix}. \tag{3.9}$$

Podobne $\mathbb{A}_{i:} \in \mathbb{R}^{1,n}$ *značí itý riadok matice* $\mathbb{A}$*:*

$$\mathbb{A}_{i:} = \begin{pmatrix} a_{i1} & a_{i2} & ... & a_{im} \end{pmatrix}. \tag{3.10}$$

Dve matice sa rovnajú pokiaľ sú rovnakého typu a majú zhodné všetky odpovedajúce prvky. [9, str. 12]

V skratke je matica len skrátený prepis sústavy n lineárnych rovníc s m neznámymi kde vypúšťame pravé strany každej z rovníc. Ak by sme pravé

strany zachovali, hovorili by sme o rozšírenej matici, ktorú zapisujeme:

$$\begin{pmatrix}\mathbb{A} & | & b\end{pmatrix} = \left(\begin{array}{cccc|c} a_{11} & a_{12} & ... & a_{1n} & b_1 \\ a_{21} & a_{22} & ... & a_{2n} & b_2 \\ ... & ... & ... & ... & ... \\ a_{m1} & a_{m2} & ... & a_{mn} & b_m \end{array}\right), \tag{3.11}$$

kde b je vektor pravých strán príslušných rovníc. V počítačovej grafike ale budeme používať najmä matice typu $\mathbb{R}^{m,m}$, t.j. matice s rovnakým počtom riadkov a stĺpcov, alebo štvorcové matice. Tieto matice budeme ale potrebovať škálovať alebo sčitovať a na to využijeme nasledovnú definíciu.

Definícia 12. *Majme $m, n \in \mathbb{N}, \alpha \in \mathbb{R}$ a $\mathbb{A}, \mathbb{B} \in \mathbb{R}^{m,n}$ matice s prvkami a_{ij}, resp. b_{ij}. Súčin matice $\mathbb{A}$ a prvku α definujeme následovne:*

$$\alpha\mathbb{A} = \begin{pmatrix} \alpha a_{11} & \alpha a_{12} & ... & \alpha a_{1n} \\ \alpha a_{21} & \alpha a_{22} & ... & \alpha a_{2n} \\ ... & ... & ... & ... \\ \alpha a_{m1} & \alpha a_{m2} & ... & \alpha a_{mn} \end{pmatrix}. \tag{3.12}$$

Súčet matíc $\mathbb{A}$ a $\mathbb{B}$ definujeme ako:

$$\mathbb{A} + \mathbb{B} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & ... & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & ... & a_{2n} + b_{2n} \\ ... & ... & ... & ... \\ a_{m1} + b_{m1} & a_{m2} + a_{m2} & ... & a_{mn} + b_{mn} \end{pmatrix}. \tag{3.13}$$

Maticu $\mathbb{A} + (-1)\mathbb{B}$ nazývame rozdiel matice $\mathbb{A}$ a $\mathbb{B}$ a značíme $\mathbb{A} - \mathbb{B}$, maticu $(-1)\mathbb{A}$ značíme ako $-\mathbb{A}$. [9, str. 13]

Podľa tejto definície vyplýva, že sčitovať môžme len matice rovnakého typu tj. napr. $\mathbb{A}^{5,5}$ a $\mathbb{B}^{5,5}$. V rámci prechodov z jednej báze do báze druhej budeme potrebovať určitým spôsobom spájať dané matice zobrazenia. Toto spájanie budeme vykonávať pomocou súčinu dvoch matíc.

Definícia 13. *Majme $m, n, p \in \mathbb{N}$ a $\mathbb{A} \in \mathbb{R}^{m,n}$, maticu s prvkami a_{ij} a $\mathbb{B} \in \mathbb{R}^{n,p}$, maticu s prvkami b_{ij}. Súčinom matíc $\mathbb{A}$ a $\mathbb{B}$ je matica $\mathbb{D} \in \mathbb{R}^{m,p}$ s prvkami d_{ij}, pre ktorú platí:*

$$d_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, \tag{3.14}$$

značíme $\mathbb{D} = \mathbb{A}\mathbb{B}$. [9, str. 14]

Súčin matíc teda môže byť výpočetne náročná operácia pre väčšie matice. My si však znova vystačíme s presným množstvom operácií, a napr. pre matice veľkosti 4×4 si vystačíme so šestnástimi súčtami štyroch súčinov dvoch prvkov. Matice môžu tiež, podľa definície, zmeniť typ, napr. pri vynásobení matice $\mathbb{A}^{5,6}$ maticou $\mathbb{B}^{6,4}$ vznikne matica $\mathbb{C}^{5,4}$. Tento jav nám nastane v momente keď budeme chcieť násobiť vektor maticou. V tomto prípade budeme vektor brať ako maticu $\mathbb{R}^{1,n}$ a násobiť ju budeme maticou $\mathbb{R}^{n,m}$ následovne:

Definícia 14. *Nech $\vec{v}$ je vektor $\vec{v} \in \mathbb{R}^{1,m}$ a $\mathbb{A}$ je matica $\mathbb{A} \in \mathbb{R}^{n,m}$. Potom platí:*

$$\vec{v}A = \begin{pmatrix} a_{11} & a_{12} & ... & a_{1m} \\ a_{21} & a_{22} & ... & a_{2m} \\ ... & ... & ... & ... \\ a_{n1} & a_{n2} & ... & a_{nm} \end{pmatrix} \begin{pmatrix} v_1 & v_2 & ... & v_m \end{pmatrix}^T = \tag{3.15}$$

$$= \begin{pmatrix} v_1 * a_{11} + v_2 * a_{12} + ... + v_m * a_{1m} \\ v_1 * a_{21} + v_2 * a_{22} + ... + v_m * a_{2m} \\ ... \\ v_1 * a_{n1} + v_2 * a_{n2} + ... + v_n * a_{nm} \end{pmatrix} \tag{3.16}$$

Podľa definície je ale výsledný vektor zapísaný v stĺpci, čo vieme ale napraviť operáciou transpozície matice, ktorá zamení stĺpcový index za riadkový. Pri programovaní s týmto však nebudeme mať problém.

3.3 Transformačné matice v počítačovej grafike

Ako sme si povedali v predchádzajúcej sekcii, matice môžme medzi sebou násobiť pri splnení určitých kritérií, akými sú napr. rozmery matice. Mimo počítačove grafiky nachádzajú využitie napr. v teórii pravdepodobnosti, či ekonomike. V počítačovej grafike, čo nás zaujíma najviac, ich ale nachádzame na každom kroku. Od zmeny farebných priestorov, cez konvolúcie až po vykresľovanie objektov v 3D priestore. Rotácie, posuny(translácie), škálovanie alebo projekcie 3D obsahu na 2D plochu. To všetko sa dá vyjadriť pomocou matíc.

Definícia 15. *Transformačnú maticu škálovania $\mathbb{S} \in T^{4,4}$ a k nej inverznú maticu $\mathbb{S}^{-1} \in T^{4,4}$ definujeme pomocou vektora $\vec{s} = (s_x, s_y, s_z)$, ktorého súradnice vyjadrujú zmenu veľkosti, postupne po x, y a z osy, ako:*

$$\mathbb{S} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \ \mathbb{S}^{-1} = \begin{pmatrix} s_x^{-1} & 0 & 0 & 0 \\ 0 & s_y^{-1} & 0 & 0 \\ 0 & 0 & s_z^{-1} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.17}$$

[11, str.548]

Definícia 16. *Transformačnú maticu posunutia $\mathbb{T} \in T^{4,4}$ a k nej inverznú maticu $\mathbb{T}^{-1} \in T^{4,4}$ definujeme pomocou vektora posunutia $\vec{t} = (t_1, t_2, t_3)$, ktorého súradnice vyjadrujú posunutie, postupne pozdĺž x, y a z osy, ako:*

$$\mathbb{T} = \begin{pmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \ \mathbb{T}^{-1} = \begin{pmatrix} 1 & 0 & 0 & -t_1 \\ 0 & 1 & 0 & -t_2 \\ 0 & 0 & 1 & -t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.18}$$

[11, str.547]

Definícia 17. *Transformačné matice rotácií $\mathbb{R}_x, \mathbb{R}_y, \mathbb{R}_z \in T^{4,4}$ definujeme pomocou vektora $\vec{t} = (t_1, t_2, t_3)$ ktorého súradnice vyjadrujú rotáciu, postupne okolo x, y a z osy o uhol θ, ako:*

$$\mathbb{R}_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.19}$$

$$\mathbb{R}_y = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.20}$$

$$\mathbb{R}_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.21}$$

[11, str.547]

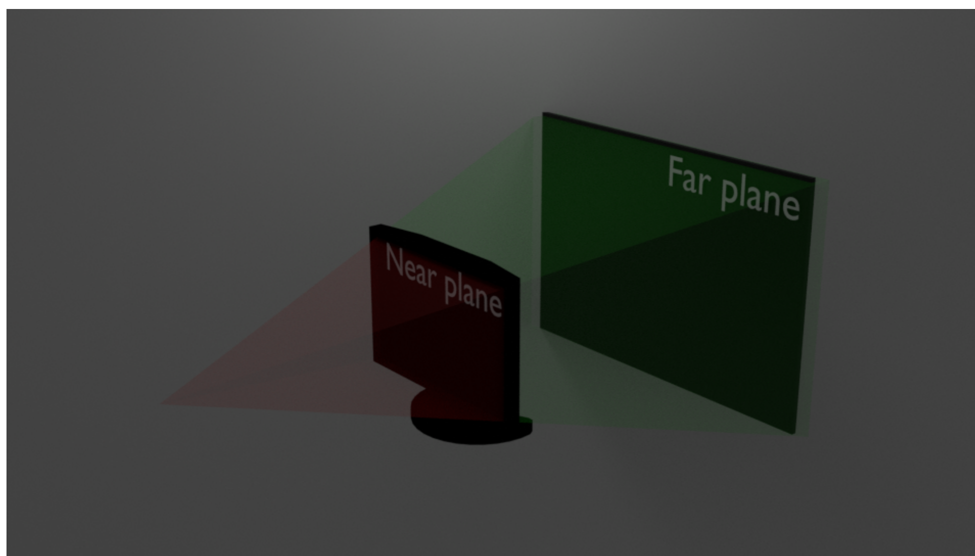
Všetky vyššie zadefinované matice sú potrebnou vedomosťou pri hýbaní objektov v 3D priestore. Stále sa ale hýbeme v 3D priestore a nevieme ako vykresliť 3D obsah na 2D rovinu, akou je napr. sietnica v monitore. Keďže budeme ale chcieť vedieť body v priestore zobrazovať na monitor a v tom istom priestore sa aj pohybovať a pozerať na rôzne strany, chýba nám teda vedomosť o tom akým spôsobom musíme vektory pred zobrazením upraviť. Prvá dôležitá transformácia, je transformácia do priestoru kamery, pomocou matice $\mathbb{C}^{4\times4}$.

Definícia 18. *Maticu kamery $\mathbb{C} \in T^{4,4}$ definujeme pomocou vektora posunutia $\vec{t} = (p_1, p_2, p_3)$, ktorého súradnice vyjadruje umiestnenie kamery v priestore, vektora $\vec{r} = (r_1, r_2, r_3)$, ktorý smeruje v pravo od kamery, vektora $\vec{u} = (u_1, u_2, u_3)$, ktorý smeruje nahor od kamery a vektora $\vec{d} = (d_1, d_2, d_3)$, ktorý určuje smer kamery, kde $\vec{r}, \vec{u}, \vec{d}$ sú normalizované vektory, ako:*

$$\mathbb{C} = \begin{pmatrix} r_1 & r_2 & r_3 & 0 \\ u_1 & u_2 & u_3 & 0 \\ d_1 & d_2 & d_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & p_1 \\ 0 & 1 & 0 & p_2 \\ 0 & 0 & 1 & p_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.22}$$

[8, str.96]

V sekcií 3.1 sme si povedali že v počítačovej grafike budeme používať troj a štvorprvkové vektory. My ale budeme predsa pracovať s troj dimenzionálnym priestorom, tak na čo by sme potrebovali štvorprvkové vektory? Odpoveď



Obr. 3.1: Perspektívna projekcia. Foto autor.

práve na túto otázku nájdeme pri zavedení projekcií. S takouto, ale veľmi jednoduchou, projekciou sme sa už mimochodom stretli pri skalárnom súčine. Ich úlohou je hlavne realizovať zobrazenie priestorov s n dimenziami do priestorov s m dimenziami. V našom prípade sa budeme venovať dvom projekciám, a to perspektívnemu a ortografickému.

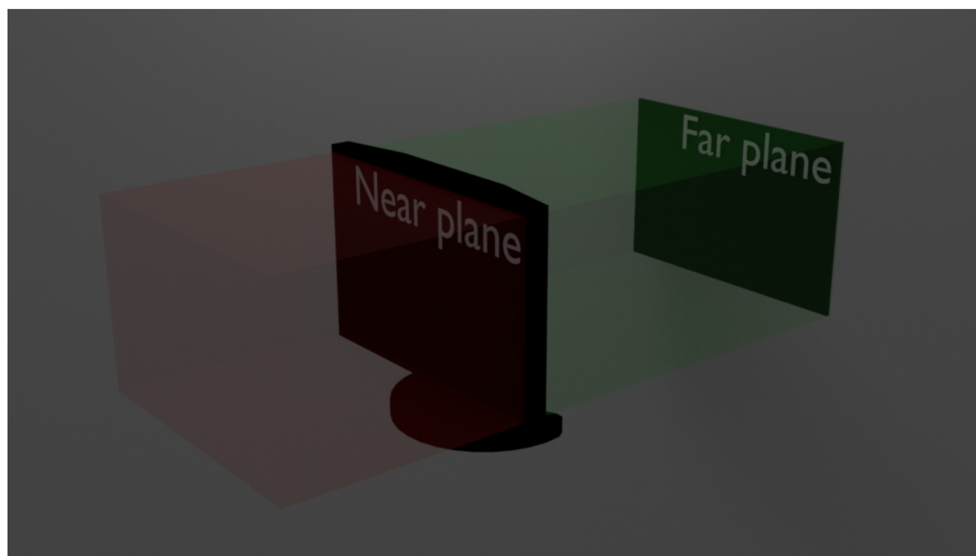
Definícia 19. *Transformačnú maticu perspektívnej projekcie* $\mathbb{C} \in T^{4,4}$ *definujeme ako:*

$$\mathbb{P}_p = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}, \tag{3.23}$$

kde ω *vyjadruje uhol zobrazenia (Field-Of-View),* n *je vzdialenosť od bližšej zobrazovanej roviny a* f *je vzdialenosť od vzdialenejšej zobrazovanej roviny, pomer šírky a výšky* $\frac{w}{h} = a_r$ *nazývame pomer strán,* $t = \tan\left(\frac{1}{2}\omega\right)n$, $b = -t$, $r = ta_r$, $l = b = -ta_r$. *[13, The OpenGL Perspective Projection Matrix]*

Ako môžme vidieť na obrázku, perspektívna projekcia má vzdialenejšiu zobrazovaciu rovinu väčšiu ako je rovina, do ktorej obraz zobrazujeme. Pomocou tejto projekcie sme schopný zobraziť 3D priestor ako keby sme v ňom boli. Vďaka nej sú vzdialenejšie objekty menšie a objekty, ktoré sú bližšie väčšie.

Definícia 20. *Transformačnú maticu ortografickej projekcie* $\mathbb{C} \in T^{4,4}$ *definu-*



Obr. 3.2: Ortografická projekcia. Foto autor.

jeme ako:

$$\mathbb{P}_o = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.24}$$

kde ω vyjadruje uhol zobrazenia (Field-Of-View), n je vzdialenosť od bližšej zobrazovanej roviny a f je vzdialenosť od vzdialenejšej zobrazovanej roviny, pomer šírky a výšky $\frac{w}{h} = a_r$ nazývame pomer strán, $t = \tan(\frac{1}{2}\omega)n$, $b = -t$, $r = ta_r$, $l = b = -ta_r$. [13, The OpenGL Orthographic Projection Matrix]

Táto projekcia síce nemá také realistické výsledky ako perspektívna, no stále nachádza použitie najmä pri modelovaní. Vďaka nej sa pri zobrazovaní dvoch bodov s rovnakými x a y súradnicami pohľadom zhora zobrazia ako jeden. Pri perspektívnej projekcií tento jav nastane len ak sú tieto súradnice x a y zároveň stredom plochy, na ktorú zobrazujeme.

3.4 Tangenty, bitangenty a normály

V rámci praktickej časti budeme často potrebovať vedieť rozlišovať medzi rôznymi priestormi, ako sú napr. obrazový priestor(screen space), lokálny priestor(local space) alebo svetový priestor(world space). Transformácie v každom z týchto priestorov rovnakými maticami môžu skončiť rôznymi výsledkami. Ak budeme napr. škálovať objekt ktorý je umiestnený niekde v priestore, potom v jeho lokálnom priestore zmeníme len jeho škálu, a však vo sveto-

vom priestore, ho môžme aj posunúť smerom k, resp. od, počiatku sústavy. Na zmenu priestoru používame matice zložené z vektorov, ktoré nazývame tangenta(dotyčnica), bitangenta a normála. Tieto vektory musia byť lineárne nezávislé. S takouto maticou sme sa už stretli pri definovaní transformačnej matice kamery.

Definícia 21. *Nech $\vec{t}, \vec{b}, \vec{n} \in \mathbb{R}^3$ sú lineárne nezávislé vektory. Potom vektor v v priestore $< t, b, n >$, značíme $\vec{v}_A$, získame pomocou rovnice*

$$\vec{v}\mathbb{A} = \vec{v}_A, \tag{3.25}$$

kde

$$\mathbb{A} = \begin{pmatrix} t_1 & t_2 & t_3 & 0 \\ b_1 & b_2 & b_3 & 0 \\ n_1 & n_2 & n_3 & 0 \\ 0 & 0 & & 1 \end{pmatrix}, \tag{3.26}$$

Normálové vektory pre nás budú mať ale hlbší zmysel, pretože popisujú umiestnenie a zrotovanie roviny v priestore. Vďaka nim môžme odhadovať vlastnosti a nasvietenie materiálov objektov v 3D scéne. Posledným potrebným je Snellov zákon, ktorý popisuje chovanie sa svetla pri priechode rôznymi prostrediami.

Definícia 22. *Nech $\alpha, \beta \in \mathbb{R}$ sú dva uhly, a $n_1, n_2 \in \mathbb{R}$ sú indexy lomu dvoch rozdielnych prostredí. Potom podľa Snellovho zákona lomu:*

$$\frac{\sin\alpha}{\sin\beta} = \frac{n_1}{n_2} \tag{3.27}$$

3.5 Kvaternióny

Transformácie pomocou matíc sme si teda zadefinovali. Bohužiaľ v počítačovej grafike nám často nemusia tieto transformácie stačiť a musíme zájsť ďalej. Preto je nevyhnutné spomenúť poslednú potrebnú matematickú štruktúru, Kvaternióny. Kvaternióny vychádzajú z komplexných čísel, čo môžme definovať ako množinu čísel následovne:

Definícia 23. *Komplexné číslo je číslo z množiny*

$$\mathbb{C} = \{a + ib \mid a, b \in \mathbb{R}, i^2 = -1\},$$

kde a nazývame reálna časť komplexného čísla a číslo b imaginárna časť komplexného čísla.[14]

Samostatne nám komplexné čísla pri rotáciách v 3D priestore nepomôžu no pri ich rozšírení o ďalšie dve imaginárne zložky, $j, k \in \mathbb{R}$ získavame kvaternióny, ktoré už vieme využiť. Kvaternióny môžme reprezentovať rôzne, najčastejšie sa však stretneme so zápisom pomocou vektorov, či usporiadanou množinou štyroch čísel:

Definícia 24. *Kvaternión je číslo $q \in \mathbb{H}$, kde*

$$\mathbb{H} = \{a + ib + jc + kd \mid a, b, c, d \in \mathbb{R}, i^2 = -1, j^2 = -1, k^2 = -1\},$$

a číslo q skrátene zapisujeme $q = [a, b, c, d]$ alebo $q = [q, \vec{q}]$, kde $\vec{q} = (b, c, d)$.[15, str. 14]

Aj pre kvaternióny platia vlastné pravidlá pre sčitovanie a násobenie, a pri násobení si narozdiel od komplexných čísel nevystačíme z definíciou samotného komplexného čísla. Môžu nám totiž nastať situácie, a často aj nastanú, kedy nám súčin dvoch zložiek kvaterniónu vo výsledku dá napr. ij alebo $-ji$.

Definícia 25. *Nech q je kvaternión $q \in \mathbb{H}$. Potom platí:*

1. $ij = -ji = k,$
2. $jk = -kj = i,$
3. $ki = -ik = j,$
4. $i^2 = j^2 = k^2 = ijk = -1$.*[10, str. 550]*

Tieto pravidlá nazývame Hamiltonove pravidlá a sú nutnou vedomosťou pri operáciách nad kvaterniónmi. Kvaternióny môžme samozrejme medzi sebou sčitovať a násobiť, pričom násobenie je v tomto prípade zložitejšie, a je nutné do počtov zahrnúť aj práve Hamiltonove pravidlá.

Definícia 26. *Nech $p = [p_0, p_1, p_2, p_3] = [p_0, \vec{p}]$, $q = [q_0, q_1, q_2, q_3] = [q_0, \vec{q}]$ a $r = (r_0, r_1, r_2, r_3)$ sú kvaternióny $p, q \in \mathbb{H}$, α skalár $\alpha \in \mathbb{R}$ a v vektor $v = (0, v_1, v_2, v_3)$ $v \in \mathbb{R}^4$. Potom platí:*

1. $q + \bar{q} = 0$, *kde* $\bar{q} = [q_0, -q_1, -q_2, -q_3]$,
2. $p \pm q = [(p_0 \pm q_0), (p_1 \pm q_1)i, (p_2 \pm q_2)j, (p_3 \pm q_3)k]$,
3. $\alpha q = [\alpha q_0, \alpha q_1, \alpha q_2, \alpha q_3]$,
4. $pq = [p_0q_0 - \vec{p} \cdot \vec{q}, p_0\vec{q} + q_0\vec{p} + \vec{p} \times \vec{q}] = [p_0q_0 - p_1q_1 - p_2q_2 - p_3q_3, p_1q_0 + p_0q_1 + p_2q_3 - p_3q_2, p_2q_0 + p_0q_2 + p_3q_1 - p_1q_3, p_3q_0 + p_0q_3 + p_1q_2 - p_2q_1] =$
5. $(\vec{p}\vec{q})\vec{r} = (\vec{u} \cdot \vec{w})\vec{v} - (\vec{u} \cdot \vec{v})\vec{w}$

[14, str.5-6]

Definícia 27. *Majme kvaternión $q = [q_1, q_2, q_3, q_4]$, $q \in \mathbb{H}$. Potom kvaternión $\bar{q} = [q_1, -q_2, -q_3, -q_4]$ nazveme komplexne združeným ku kvaterniónu q, $q, \bar{q} \in \mathbb{H}$. Ďalej kvaternión $q^-1 \in \mathbb{H}$*

$$q^{-1} = \bar{q}/(q\bar{q}) \tag{3.28}$$

nazveme inverzným kvaterniónom kvaterniónu q, číslo $|q| \in \mathbb{R}$*, kde*

$$|q| = \sqrt{q\bar{q}} = q_1^2 + q_2^2 + q_3^2 + q_4^2 \tag{3.29}$$

nazveme normou kvaterniónu q a kvaternión $q_p \in \mathbb{H}$*, kde* $q_p = [0, v_1, v_2, v_3]$ *nazveme čisto imaginárnym kvaterniónom. Nakoniec kvaternión* $q_n \in \mathbb{H}$*, kde* $q_n = |q|^{-1}q$ *nazveme normalizovaným kvaterniónom. [15, 14]*

Teraz máme definované všetky základné operácie nad kvaterniónmi. Zostáva si teda priblížiť, spôsob rotovania vektorov kvaterniónmi. Tieto rotácie možno využiť ako náhradu za maticové rotácie, a už z prvého pohľadu sú o proti maticiam úspornejšie, pričom nie sú náročné na implementovanie, pretože časť operácií vďaka zápisu pomocou vektorov s nimi zdieľajú.

Definícia 28. *Nech* $q, p \in \mathbb{H}$*, kde q je normalizovaný kvaternión tvaru*

$$q = [\sin(\theta), \vec{n}\cos\theta], \tag{3.30}$$

kde $\vec{n}$ *je normalizovaný vektor reprezentujúci os rotácie a* $p = [0, v_1, v_2, v_3]$ *je čisto imaginárny kvaternión, ktorého zložky* (v_1, v_2, v_3) *predstavujú vektor v, ktorý chceme zrotovať. Potom definujeme nad vektorom v operátor* $R_q(v)$*, následovne:*

$$R_q(v) = qp\bar{q}, \tag{3.31}$$

ktorého výsledkom je čisto imaginárny normalizovaný kvaternión $q_v = [0, q_{v1}, q_{v2}, q_{v3}] = [0, \vec{q_v}]$*, kde* $\vec{q_v}$ *je výsledný zrotovaný vektor. [15, 14]*

Tak ako pri rotáciach realizovaných maticami aj tu používame miesto stupňov radiány.

KAPITOLA 4

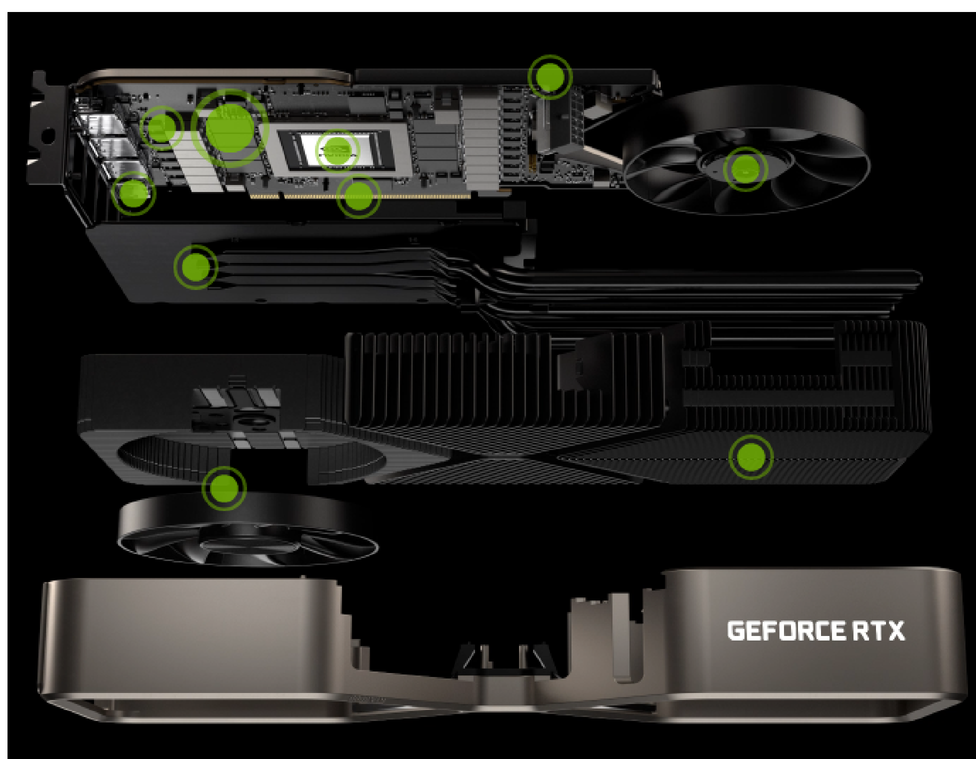
Počítačová grafika

Počítačová grafika je plná rozličných pojmov, ktoré v tejto práci ani nebude možné prejsť všetky. Väčšinu pojmov je však náročné preložiť do slovenčiny, a budeme si musieť často vystačiť i s anglickým znením. Je ale dôležité dodať, že rôzne zdroje pristupujú k týmto definíciam rôzne. Obecne v obore počítačovej grafiky sa často stretávame s rôznymi definíciami rovnakých pojmov a často je ťažké exaktne definovať niektoré pojmy.

4.1 Grafické karty

Počítačová grafika používa algoritmy, ktoré sú často časovo náročné. Môžme pre ilustráciu spomenúť napr. Fourierovú transformáciu, ktorá samotná má síce zložitosť $O(n^2)$, ale pre každý ďalší rozmer túto zložitosť zvyšuje, a preto ľahko skončíme pri obrázkoch so zložitosťou $O\left(n^4\right)$ a pri 3D objektoch $O(n^6)$. Samozrejme, že v počítačovej grafike by sme chceli mať výsledok čo najrýchlejšie dostupný a to ideálne bez zaťažovania CPU, ktorý by mal na starosti základný beh systému. Prvé grafické karty, podľa C. McClanahana [16], v 80. rokoch používali známy mikroprocesor Intel 8088. Síce to na prvý pohľad neznie ako veľký pokrok, no v 80. rokoch, kedy vtedajšie procesory disponovali rádovo s desiatkami kB pamäte, toto oddelenie ušetrilo procesoru čas a pamäť, ktoré mohol využiť na spracovanie iných úloh. Ako C. McClanahan spomína [16], od 90. rokov už môžme tomuto oddeleniu začať hovoriť GPU. Dovtedy šlo skôr len o kvázi matematický akcelerátor, ktorý v skutočnosti samotné inštrukcie a matematické operácie neurýchľoval, len ich paralelizoval.

Grafická karta, súčasne označovaná skratkou GPU, je hardvérová súčasť počítača, slúžiaca ako sekundárny, na samotný chod počítača ale nepotrebný, procesor, s prevažne matematickými optimalizovanými inštrukciami a viacerými jadrami. Väčšinou vyzerajú ako na obrázku 4.1, no nemusia mať aktívne chladenie ba ani kryt. Narozdiel od klasického procesora má grafická karta v súčasnosti ale rádovo tisíce jadier, ktoré sa spájajú do logických celkov, kde



Obr. 4.1: Grafická karta Nvidia GTX 3. radu[1]

všetky jadrá jedného celku pracujú na jednom probléme. Takisto narozdiel od procesora majú niekoľko pomocných hardvérových celkov na urýchlenie ich častých pracovných úkonov ako napr. rásterizér alebo jednotku tesselácie.

4.2 Rendering

V počítačovej grafike často používame pojem render a jeho deriváty. Rendering je zjednodušene process, kedy na objekty v scéne aplikujeme pomocou programu fyzikálne zákony osvetlenia. Výsledkom renderingu je render, či rástrový obrázok. Program, ktorý aplikuje fyzikálne zákony osvetlenia na objekty nazývame renderer. V súčasnosti existuje viacero rendererov, každý odlišný v niečom. Medzi najznámejšie patria napríklad bulharský renderer V-Ray[1], NVIDIA Mental Ray[2], alebo český Corona Renderer[3], každý je v niečom odlišný a špecializujú sa na rôzne metódy rendrovania. Všetky vyššie uvedené renderery sú ale, takzvane fyzikálne založené renderery, alebo anglicky „physically

[1] `https://www.chaosgroup.com/vray/collection`

[2] MentalRay`https://www.nvidia.com/en-us/design-visualization/solutions/rendering/product-updates`

[3] Corona`https://corona-renderer.com/`

Obr. 4.2: Scéna rendrovaná fyzikálne založeným rendererom, Corona Renderer [2]

based renderers“ skratkou PBR. Tieto renderery realizujú väčšinu svojich výpočtov na CPU, a GPU používajú menej a zameriavajú sa na presnú simuláciu fyzikálnych javov, ktoré sú zväčšiny výpočtovo náročné. Spravidla používajú metódy ako ray-tracing a path-tracing, ktoré siahajú až do 16. storočia, kedy ich zformuloval nemecký maliar A.Dürer [17]. Popisujú spôsob šírenia svetla v scéne pomocou odrazov, refrakcie a pohlcovania energie pri dopade svetla na povrch. Svetlo sa vo všeobecnosti šíri pomocou fotónov, ktoré sú neustále vyžarované zo svetelných zdrojov. Svetlo takto v každom momente vyžaruje nekonečné množstvo fotónov. Obraz, ktorý vzniká pozostáva z fotónov, ktoré dopadnú na sietnicu oka. Táto metóda je ale výpočtovo extrémne náročná, a preto sa používa kvázi inverzná metóda rendrovania, kde začíname z konca. Pre každý pixel vygenerujeme lúč, ktorý vyšleme smerom do scény. Ak sa lúč dotkne nejakého objektu, podľa materiálu objektu sa rozhodne, či sa lúč odrazí, zalomí alebo pohltí. Farbu vráti po n odrazoch alebo zalomeniach a ak sa lúč ničoho nedotkne vráti farbu prostredia [11, str. 422-426].

Tieto spôsoby renderingu sú ale výpočetne náročné a získanie kvalitného zaostreného obrazu môže v niektorých prípadoch, najmä na menej výkonnejších procesoroch trvať rádovo hodiny. My sa budeme venovať takzvanému realtime renderingu, ktorý sa skôr zameriava na rýchlo získateľný výsledok ako realistický výsledok. V tejto metóde budeme používať interpolácie na získanie farebnej interpretácie objektov a využívať budeme najmä GPU.

Obr. 4.3: Scéna rendrovaná realtime GPU rasterizérom, CryEngine v hre Kingdome Come: Deliverance. Foto autor.

4.3 Rásterizér

Jedným z hlavných rozdielov medzi realistickými renderermi a realtime renderermi je spôsob vykresľovania scény na 2D mriežku. Algoritmus rásterizácie je implementovaný hardvérovo ako súčasť grafických kariet, aj preto je táto metóda prioritizovaná pri realtime renderingu. Algoritmus prejde cez všetky primitívy, tj. väčšinou trojuholníky, z ktorých je model zložený, a rozhoduje, ktoré pixely v obraze primitíva afektuje. Vďaka tomu sa vieme pri öfarbovaní"pixelov pýtať, aká je v danom bode plocha, kam smeruje normála atď. Tento prístup sa používa súčasne najčastejšie v hernom priemysle, práve pre jeho rýchlosť, iným názvom tento spôsob voláme object centric pretože projektuje objekty na 2D mriežku, alebo display. Metódy rendrovania, ktoré vysielajú lúče do scény, pracujú práve opačným smerom, to znamená, že skrze všetky pixely, sa prechádzajú všetky primitívy. Tento spôsob voláme image centric, pretože vysielame lúče skrz obraz do scény. [13, An Overview of the Rasterization Algorithm]

4.4 Pipeline

Už od počiatkov počítačovej grafiky od nás knižnice rásterizérov vyžadovali dodržiavanie určitého poradia príkazov a sami sa správali v podstate ako jednoduché konečné automaty. Spočiatku bolo toto poradie pevne dané, nemenné, a nemohli sme s ním moc pracovať a prispôsobovať na to čo potrebujeme. V modernej počítačovej grafike vieme ale niektoré kroky upraviť ako potre-

bujeme najmä pomocou shaderov. Toto poradie v počítačovej grafike voláme pipeline, a popisuje poradie operácií, ktoré vedú k výsledku finálneho rendra. Grafické znázornenie krokov pri rasterizačnom renderingu môžme vidieť na obrázku 4.4

Pipelina sa spustí volaním rendrovacej funkcie na korektne založenom buffery, ktorý je naplnený vertexami, zlinkovaného shaderu a zlinkovaného pipelineu. Tieto vertexy špecifikujú ohraničenie primitív. Primitívy sú najzákladnejšie časti geometrie modelu, ako napr. plocha, čiara alebo bod. Nasledujú tri štádiá úprav geometrie:

1. vertex shader obdrží každý vertex ako je špecifikovaný vo vertex array objekte, upraví a pošle do tesselation shadera. Vertex shader je v súčasnosti jediný povinný.

2. Tesselácia slúži na rozdrobenie primitívy na menšie primitívy.

3. Geometry shader, generuje a transformuje výsledné primitívy.

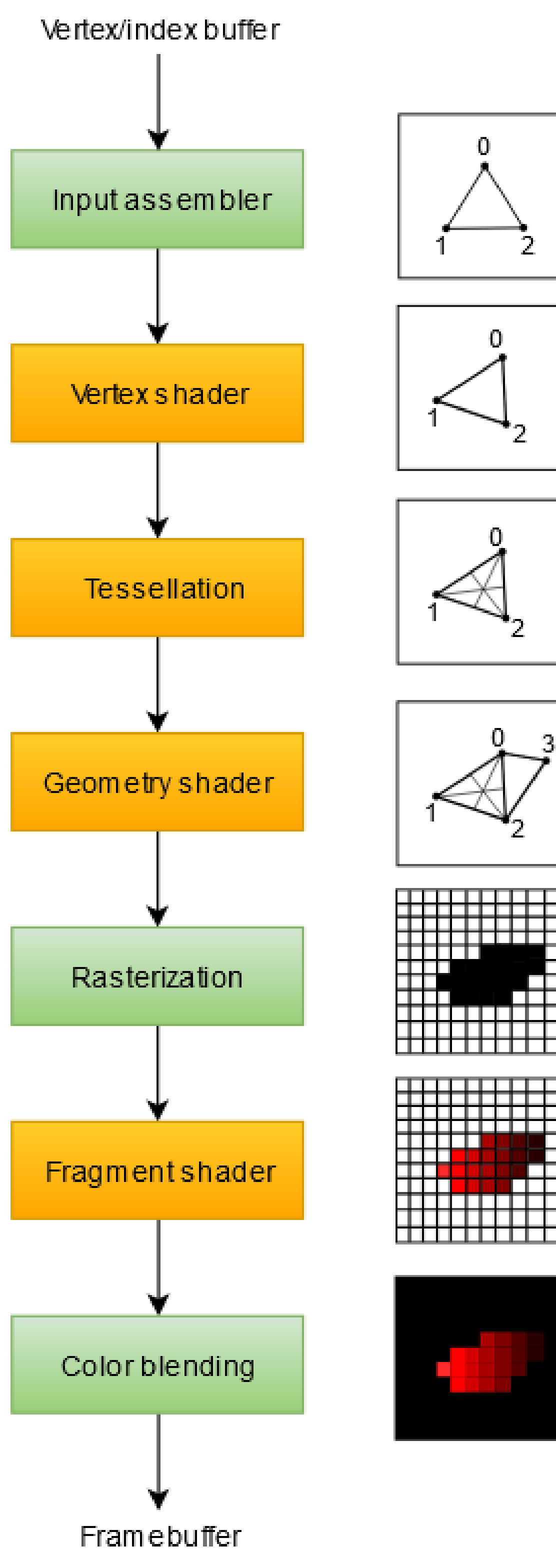
Následuje vertex postprocessing, ktoré nevieme editovať. Primitívy tvoriace sekvenciu sa zapíšu do nových bufferov, následne sa prečnievajúce primitívy z fremebufferu orežú tak, aby boli všetky body vo vnútri zobrazovaného poľa. Proces popísaného orezania voláme clipping. Tieto štádiá slúžia aj na optimalizačné účely, a vedia zastaviť render pokiaľ bod nebude pre konečný výsledok potreba rendrovať. Následuje rásterizácia a následne spustenie nepovinného fragment shadera, v ktorom vieme upraviť farbu fragmentu, resp. rástra. V poslednom rade je tu štádium persample operácií ako sú depth testing alebo blending. Po týchto operáciách sa hodnoty zapíšu do framebufferu na pozície, ktoré sme nevylúčili v maskovacích operáciách.

4.5 Shader

Shader je menší program, ktorý je zkompilovaný počas behu hlavného programu, a beží na GPU. Vďaka ním vieme určité štádia rendrovania upraviť. Shadery ďalej delíme na:

1. vertex shadery, ktoré sú schopné upravovať body, resp. vertexy, geometrie. Často v nich počítame s maticami, ktoré realizujú transformácie ako sú projekcie, matice kamery, translácie, rotácie a škály. Môžme v nich takisto realizovať osvetlenie, no vtedy budeme toto osvetlenie volať vertexové,

2. tessellation shadery, sa používajú na špeciálny typ odsadenia bodov v geometrií. V tomto prípade sa ale nejedná o posun vertexov modelu, tento posun je realizovaný rásterovo,



Obr. 4.4: Kroky rásterizačneho renderera[3].

3. fragment shadery slúžiace na výpočty osvetlenia. V tomto štádiu sa často počítajú odlesky, zafarbenie, často transparencia, za využitia fyzikálnych javov a aproximácií realizovaných rásterovo. Ak osvetlenie počítame v tomto štádiu budeme ho volať bodové,

4. compute shadery sú v renderingu počítačovej grafiky málo využívané, používajú sa najmä na rýchle paralelizovateľné výpočty.

[18, sekcie 2.11, 2.12, 2.13, 3.9]

Tieto shadery sú medzi sebou zvyčajne prepojené a dávajú nám voľnosť v realizovaní geometrických a farebných úprav. Shadery sú písané zvyčajne v jazyku, ktorý je podobný jazyku C. Často má vlastné kľúčové slová, štandardne implementované funkcie, v OpenGL shading jazyku napr. funkcia clamp, ktorá vstupnú hodnotu oreže tak, aby výsledok bol medzi špecifikovaným minimom a maximom. Vieme si v nich definovať vlastné funkcie, vstupné a výstupné parametre, tj. parametre, ktoré očakávame z predchádzajúcich shaderov, a ktoré posielame do ďalších štádií, alebo definovať rôzne makrá. Zaujímavosťou je, že OpenGL a OpenCL nepodporujú rekurziu a rekurzívne problémy musia byť naprogramované iteratívne. [8, str. 42-47]

4.6 Buffer Objekty

Dôležitou súčasťou rýchlosti realtime renderingu, je posielanie dát, z CPU na pamäť GPU, ktorá operuje, resp. disponuje, frekvenciou rádovo desiatkou GHz, resp GB. Pamäť GPU je rozdelená do viacerých celkov, kde každý zväzok vlákien má vlastnú pamäť, existuje pamäť globálna zdieľaná všetkými vláknami a zdieľaná, dočasná, pre málo časté prístupy vlákien [19]. Na tento účel používame štruktúry zvané buffer objekty, ktoré sú len neformátovaná pamäť vytvorená programovacím prostredím komunikujúcim s GPU, na ktoré vieme uložiť vertexové dáta, alebo z nich dáta vytiahnuť a je ich viacero:

1. Vertex Array objekt, ukladajúci formát vertexových dát,

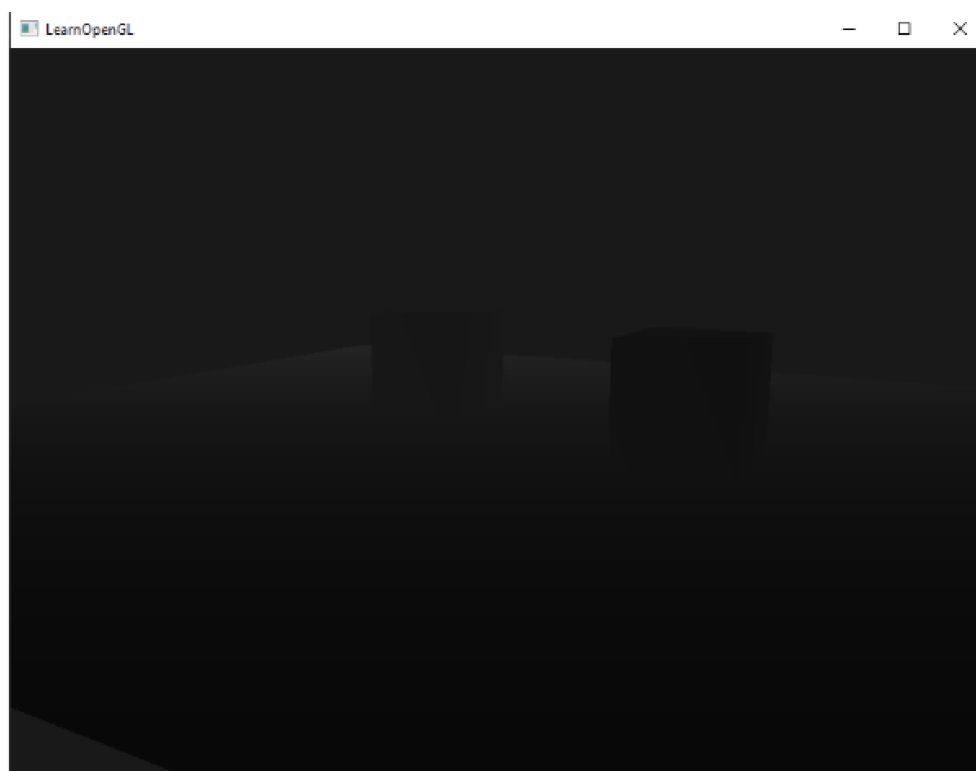
2. Vertex buffer objekt, v ktorom ukladáme dáta vertexov, ich normál, farieb ak chceme, UV coordinátov a podobne,

3. Element buffer objekt, slúžiaci najmä na spájanie vertexov do trojuholníkov, podľa ich indexov vo VBO,

4. Texture buffer objekt, slúžiaci na uloženie dát textúr na GPU pre rýchlejší prístup,

5. Framebuffer objekt, slúžiaci na rendrovanie mimo hlavný framebuffer,

6. Uniform buffer objekt, obmedzený, slúžiaci na odosielanie dát, spoločných pre viacero shaderov.



Obr. 4.5: Efekt hmly implementovaný pomocou depth buffera[4]

[20, Vertex Specification]. Prístup k týmto dátam, a všeobecne prístup k dátam na GPU stojí veľa času, preto je potrebné ho v značnej miere obmedzovať. Tieto dáta ale ovplyvňujú model ako jeden celok. My budeme ale potrebovať ovplyvniť napr. farbu modelu podľa osvetlenia, a práve na to používame premenné nazývané uniformné premenné.

4.7 Depth buffer

Depth buffer, hĺbkový buffer je buffer, šírky a výšky hlavného frame buffera, ktorý ale narozdiel od ukladania farieb ukladá jednu hodnotu, do float premenných často s presnosťou 24 bitov, mapujú vzdialenosť, resp. hĺbku, v ktorej sa daný pixel nachádza medzi hodnoty 0,0 a 1,0. Vďaka nemu vieme zastaviť výpočet pokiaľ sa práve počítaný fragment nachádza v scéne z nášho pohľadu za, už ofarbeným pixelom. Depth buffer vieme využiť ale aj na simuláciu hmly, ako vidíme na obrázku 4.5. S depth bufferom, ale prichádza prvý problém, resp. grafický artefakt. Artefakty sú grafické chyby, ktoré vznikajú pri renderingu. V tomto prípade túto chybu voláme z-fighting. Z-fighting nastáva ak sú 2 rovnobežné roviny v scéne blízko jedna k druhej. Bohužiaľ

na vyriešenie tohoto artefaktu neexistuje žiadne lepšie riešenie, objekty musíme v scénach umiestňovať tak aby artefakt nenastal. Po vyriešení problému určovania, ktorá farba pixelu by sa mala zobraziť pred ktorou, nám zostáva spomenúť blending. Blending vyjadruje spôsob miešania farieb, ak máme takéto dva pixely a pixel bližšie k nám je transparentný. Vtedy budeme musieť zobrať pixel, ktorý je transparentný a zmiešať ho s pixelom, ktorý je za ním. Transparencia síce môže vyzerať realisticky, no v prípade že ju budeme používať v scénach často, budeme v konečnom dôsledku musieť ofarbovať pixely niekoľko krát. Toto bude mať negatívny dopad na rýchlosť renderingu [8, kapitoly 22, 24]. Jednou zo zaujímavých vlastností tohto problému je že záleží od plochy, ktorú budeme musieť rendrovať spomenutým spôsobom viackrát. Čím väčšia plocha, tým väčší bude dopad na rýchlosť.

4.8 Scéna, level, model, mesh

V našich programoch budeme často pracovať s veľkým množstvom objektov. Aby sme sa ľahko vedeli orientovať v takýchto priestoroch musíme si zaviesť niektoré pojmy ako sú scéna, level, objekt, model a mesh. Už od počiatkov herného priemyslu, u hier ako Doom alebo ešte skorších Space Invaders, sa v hrách vyskytovali tieto pojmy pod rôznymi názvami. U spomenutých Space Invaders sa používal pojem level. V Doome sa využíval špecifickejšie názvoslovie, napr. E1M1, ktorý značil prvú misiu v prvej epizóde.

Level je štruktúra, združujúca objekty, ktoré majú niečo spoločné. Môže obsahovať modely budov, rastlín, terén, rôznych malých objektov, bodové, resp. smerové osvetlenie. Tieto levely na seba môžu naväzovať a tvoriť tak orientované grafy. Scénu budeme brať ako množinu levelov, ktorých objekty chceme upravovať a renderovať. Objektom nazveme abstraktne všetky možné súčasti levelov.Mesh je množina bodov v priestore, ktoré utvárajú vizuálny vzhľad nejakého komponentu v modely. Množina meshov, ktoré spolu súvisia a spolu utvárajú nejaký objekt, spoločne tvoria model. Predstavme si napríklad nejaký automobil. Asi triviálnou vedomosťou je že taký automobil sa skladá z viacerých súčiastok, napr. karoséria, kolesá, sedadlá atď. a spoločne fungujú. Presne takýmto spôsobom budeme brať modely. Ak budeme potrebovať posunúť model, posunieme všetky meshe z, ktorých model pozostáva. Ak však budeme potrebovať zrotovať objekt alebo ho vyškálovať budeme potrebovať bod, okolo ktorého môžme rotáciu previesť. Tento bod budeme volať pivot. Získame ho jednoducho tak že všetky body vektorovo sčítame a vynásobíme $1/N$, kde N je počet bodov. Tento algoritmus sa môže zdať časovo náročný, no keďže budeme mať viacero modelov a meshov vieme algoritmus ľahko zparalelizovať a spracovávať tak viacero modelov naraz. Navyše túto operáciu budeme musieť používať maximálne raz v behu programu a vieme sa jej vyhnúť ak pred exportom modelu z 3D editovacieho softwéru model umiestnime do počiatku sústavy. Celý model potom posunieme pomocou matice trans-

lácie, z kapitoly 3.3, a až potom môžme model zrotovať, resp. vyškálovať. Každý mesh môže mať iný materiál, a takisto každý mesh môže mať viacero materiálov.

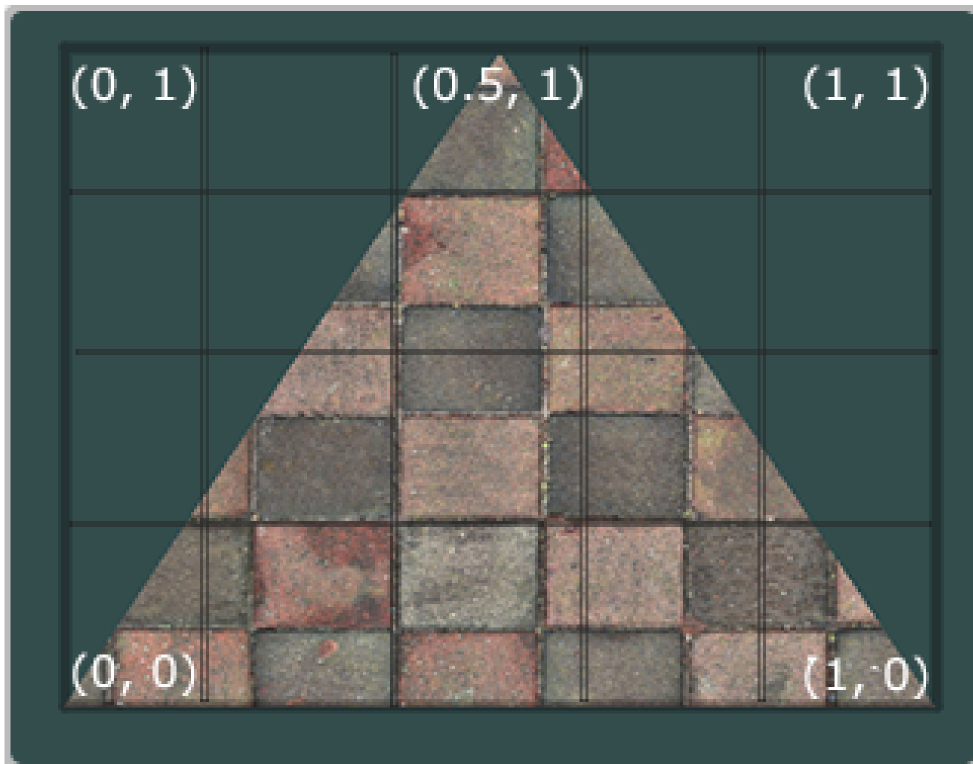
4.9 Texture Mapping

Objekty teda vieme „nafarbiť" podľa toho aký majú materiál. Čo ak, ale nebudeme schopní popísať ofarbenie objektu jednoducho jednou farbou. Napríklad drevo. Takéto objekty popisujeme textúrami, ktorých je rovno niekoľko. Definovať textúry nieje jednoduché[21, 22, 23]. V počítači sú ukladané ako obrázky vo formáte PNG alebo JPG, či BMP. Potrebujeme teda odlíšiť obrázok od textúry. Obrázok popisuje celú scénu, čo a kde sa v nej nachádza, osvetlenie, objekty a ich povrchy. Textúra však farbou popisuje povrch iba jedného materiálu. V realite závisí aj na tom z akého uhlu je taká látka nasvietená. Textúr máme ale rovno niekoľko[24], medzi najzákladnejšie patria:

1. difúzna textúra, popisuje matný povrch objektu,
2. reflexná textúra, popisujúca ktoré oblasti majú odrazovať svetlo,
3. specular textúra, používaná na zmenu farby odrazeného svetla,
4. textúra transparencie, popisujúca priesvitnosť objektu,
5. normálová textúra, obsahujúca normálové vektory mapované z koordinátov (x,y,z) do (r,g,b),
6. textúra prostredia, popisujúca prostredie,
7. textúra odsadenia(displacement), vďaka ktorej vieme body v modely odsadzovať.

Pri použití viacero textúr naraz dosiahneme lepších výsledkov z hľadiska realistického vzhľadu a túto metódu voláme multitexturing.

V perspektívnom zobrazení, objekty, ktoré sú vo väčšej vzdialenosti zaberajú na obrazovke menšiu plochu. To ale predsa platí i na textúry. Povedzme, že máme textúru dreva. Z blízkosti vidíme, veľa detailov, vidíme napr. letokruhy, vidíme diery v dreve, ryhy alebo üzly". Keď sa ale pozrieme z väčšej vzdialenosti detaily začnú zanikať a postupne vidíme len akúsi zmes farieb tejto textúry. Pri behu programu teda budeme musieť nejak zväčšovať, resp. zmenšovať textúru niekoľkokrát. Tento proces je však pri realtime rendrovaní kritické. Existuje niekoľko spôsobov ako predísť počítaniu týchto textúr, my sa zameriame na Mip-maping. Ide o proces kedy sa predpočítavajú textúry na nové veľkosti. Z prvej textúry sa vypočíta textúra polovičnej veľkosti, z tej sa vypočíta textúra znova polovičnej veľkosti atď. Pri aplikovaní sa použije textúra, ktorá je veľkosťou najbližšie k požadovanej veľkosti a navyše sa upraví na presne požadovanú veľkosť.



Obr. 4.6: UV mapovanie textúr[5]

V poslednom rade, aspoň v tejto práci, si spomeňme na fakt, že textúry sú vlastne dvojrozmerné objekty, ktoré aplikujeme na trojrozmerné objekty. Spôsob aplikovania textúry na objekt voláme texture mapping a existujú rôzne spôsoby:

1. UV texture mapping, ktorý budeme používať my, používa na určenie pixela textúry, použitého na fragment, UV koordinát, ako na obrázku 4.6
2. triplanar texture mapping, ktorý je založený na výbere textúry podľa normály, používaný najmä na terén,
3. sphere texture mapping,
4. cube texture mapping,
5. cone texture mapping,
6. cylinder texture mapping.

[11]

4.10 Gimbal-Lock

Počas programovania sa stretneme so známym artefaktom, nazývaným gimbal lock. Vzniká pri aplikácií rotácií postupne okolo napr. x,y a nakoniec z osy. Týmto spôsobom sa jedna z osí rotácie zarovná s inou, my stratíme stupeň voľnosti, čo znamená, že nebudeme môcť okolo danej osy rotovať. Tomuto problému vieme predísť vyjadrením spoločnej matice pre rotácie okolo x, y a z osy. Elegantnejšie však je riešiť tento problém pomocou kvaterniónov.

4.11 Osvetlenie

Z doposiaľ vysvetlených algoritmov sme si mohli všimnúť, že v realtime renderingu potrebujeme šetriť časom pri každej operácií. Ray tracing, ktorý simuluje realistické nasvietenie nepripadá do úvahy, už kvôli prvému štádiu úpravy scény pre rendering. Spôsobov existuje znova niekoľko. Medzi jedno z najznámejších patrí Phongov osvetľovací model. Phongov model rozdeľuje osvetlenie do troch štádií, resp. zložiek. Ambientná zložka, difúzna zložka a reflexná zložka. Ambientná zložka simuluje osvetlenie objektov v noci. I keď v noci nesvieti žiadne svetlo, stále vieme približne rozoznať farbu objektu, pretože sa na zem svetlo dostane odrazom Mesiaca alebo iných hviezd. Svetlo sa takisto vie odraziť od objektov, od reflexívnych objektov viac ale aj matné objekty vedia odraziť, i keď menšiu časť svetla. Ďalšou zložkou je zložka difúzna, popisujúca osvetlenie plochy, nejakým zdrojom osvetlenia, počítané pomocou výpočtu uhla medzi vektorom dopadu svetla a normálou. Ak svetlo dopadá na rovinu kolmo, bude osvetlená najviac, no ak svetlo dopadá na rovinu pod uhlom viac ako 90 stupňov plocha nebude nafarbená. Poslednou zložkou je reflexné osvetlenie. Toto osvetlenie počíta s ďalším vektorom, vektorom pohľadu a snaží sa vizuálne popísať odlesky na modeloch. Vektor dopadu svetla reflektujeme pozdĺž normály, a podľa uhla medzi výsledným vektorom a vektorom pohľadu popíšeme veľkosť odlesku. Zložky nakoniec kombinujeme jednoducho aditívne [8]. Samostatne bez multitextúrovania nieje tento výsledok veľmi realistický, no pri použití viacerých textúr dostávame celkom realistický výsledok.

Časť II

Praktická časť

Kapitola 5

Vytvorenie knižnice

V tejto časti sa budeme venovať už programovaniu a riešeniu konkrétnych problémov, ktoré sa naskytnú počas programovania. Využívať pritom budeme Microsoft Visual Studio, ideálne 2019. Pre tvorbu oknových aplikácií budeme používať knižnicu GLFW[4], ktorá poskytuje multiplatformovú podporu vytvárania okien. Vďaka nej sa vyhneme, naozaj až študovaniu WinAPI[5], ktoré má prekvapivo na mnohých miestach zastaralú dokumentáciu a je zbytočne zložitá kôli zachovávaniu spätnej kompatibility. Posledne budeme používať knižnice tinyobjloader[6], ktorá nám poslúži na import modelov, exportovaných z 3D editorov, ako Blender, 3DS MAX, resp. Cinema 4D a knižnicu stb[7] na načítavanie textúr. Nakoniec budeme potrebovať Vulkan-SDK[8].

5.1 Nastavenie MSVS 2019

Pred programovaním však musíme Visual Studio nastaviť. Budeme potrebovať napr. typ optional, implementovaný v c++17 štandarde. Toto nastavíme v MSVS rozkliknutím vlastností projektu[9], a pod kolonkou C/C++-Language, nastavíme C++ Language Standard na ISO C++ 17 Standard podľa obrázka 5.1. Ďalej musíme linkeru a InteliSense povedať, kde sa nachádzajú ďalšie knižnice, ktoré chceme používať. Tieto možnosti nájdeme znova vo vlastnostiach projektu, pod kolonkami C/C++-General 5.2, Linker-General 5.3, a Linker-Input 5.4. Knižnice pre jednoduchosť odporúčam roztriediť do priečinkov include a library podľa obrázka 5.5. Pre dll knižnicu musíme nastaviť v MSVC typ konfigurácie na DLL. Ostatné projekty budú mať typ konfigurácie

[4] Dostupná na `https://www.glfw.org/`

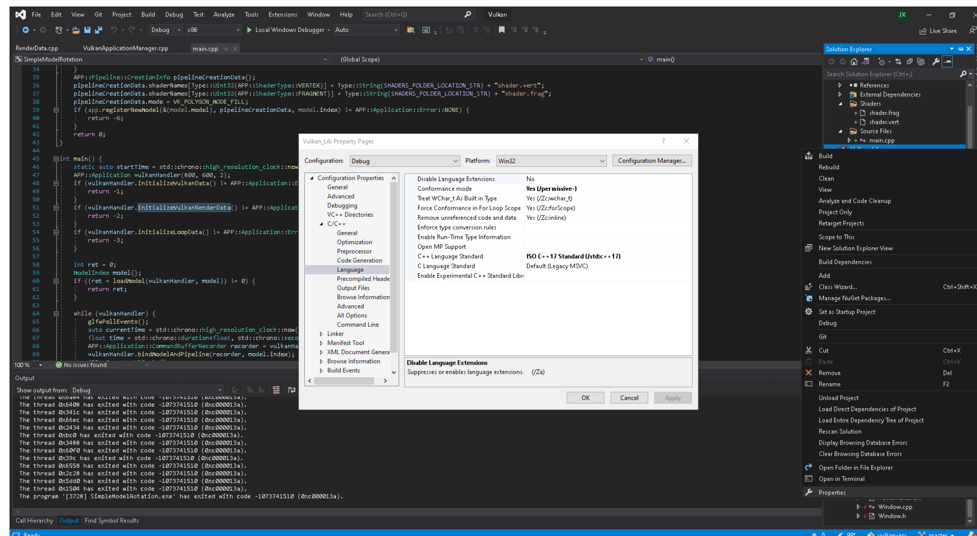
[5] Dokumentáciu nájdeme na `https://docs.microsoft.com/en-us/windows/win32/apiindex/windows-api-list`

[6] Stiahnuteľné z `https://github.com/tinyobjloader/tinyobjloader`

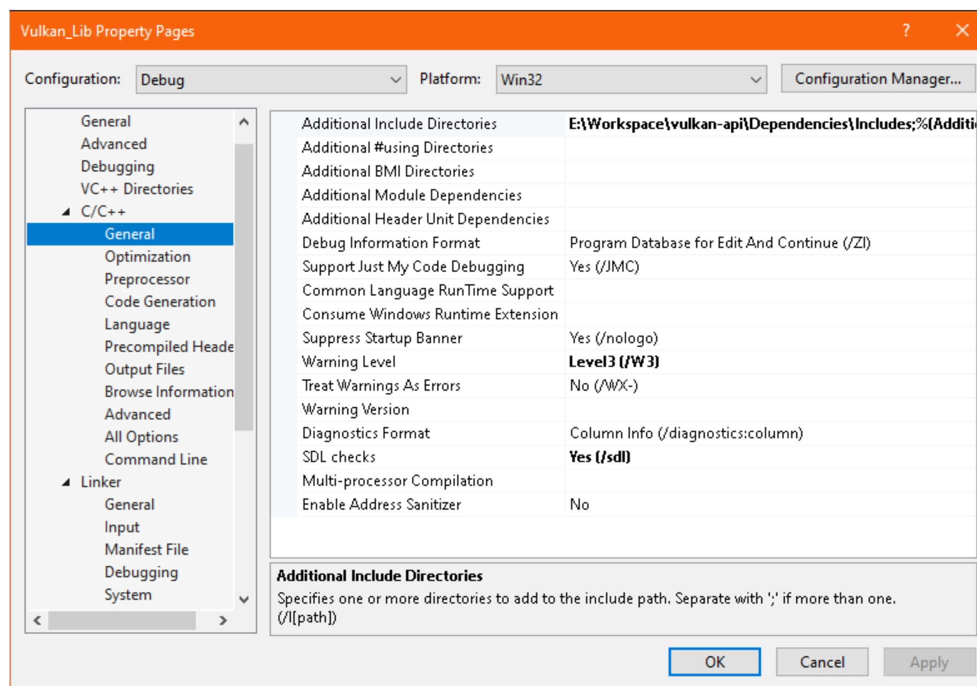
[7] Stiahnuteľné z `https://github.com/nothings/stb`

[8] Dostupné pre Windows na `https://vulkan.lunarg.com/`

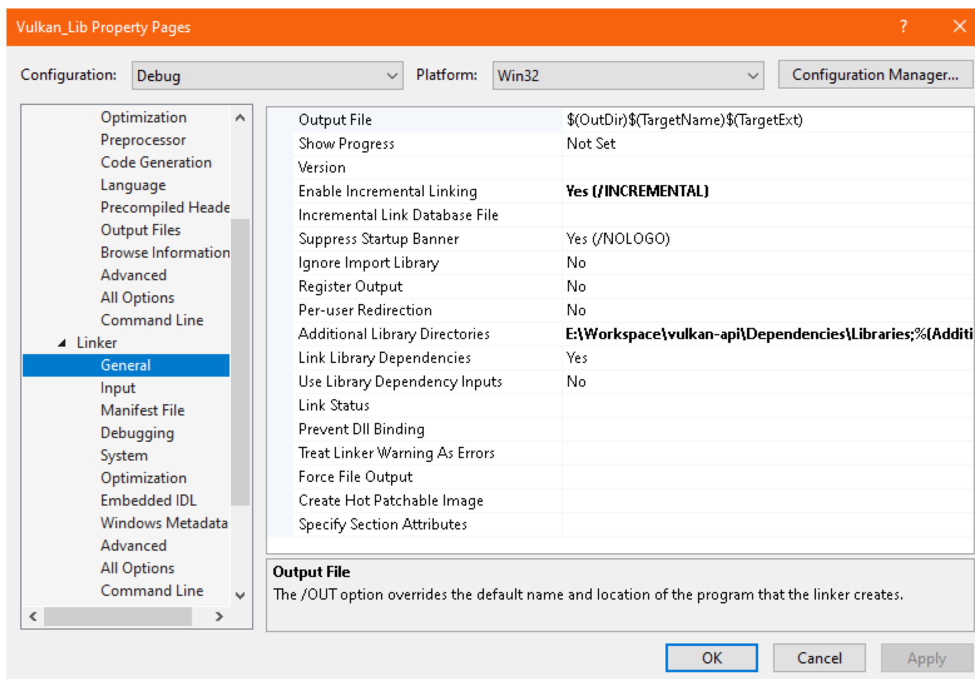
[9] Pozor, nie vlastností solution



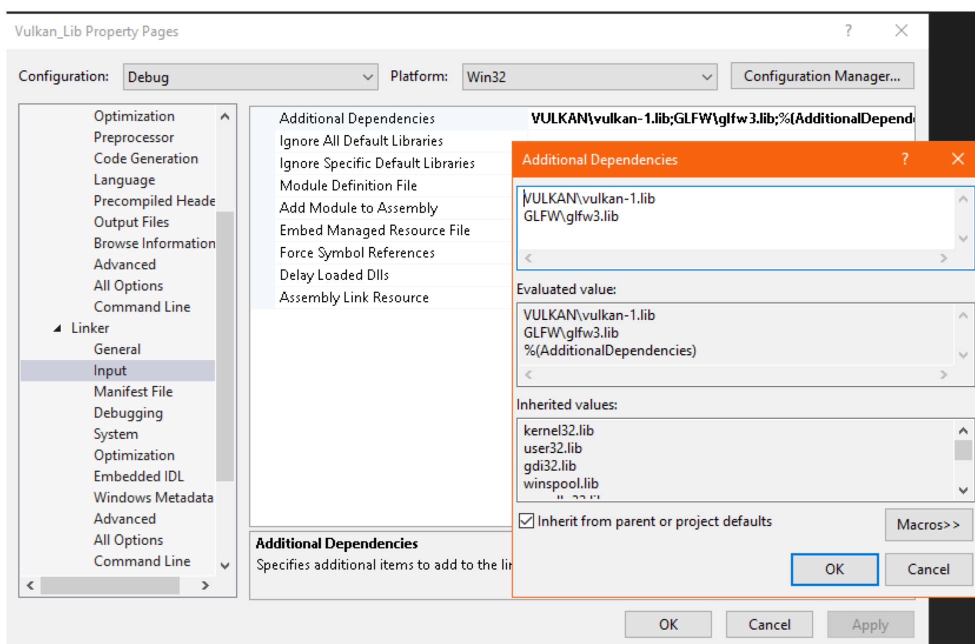
Obr. 5.1: Nastavenie štandardu c++. Foto autor.



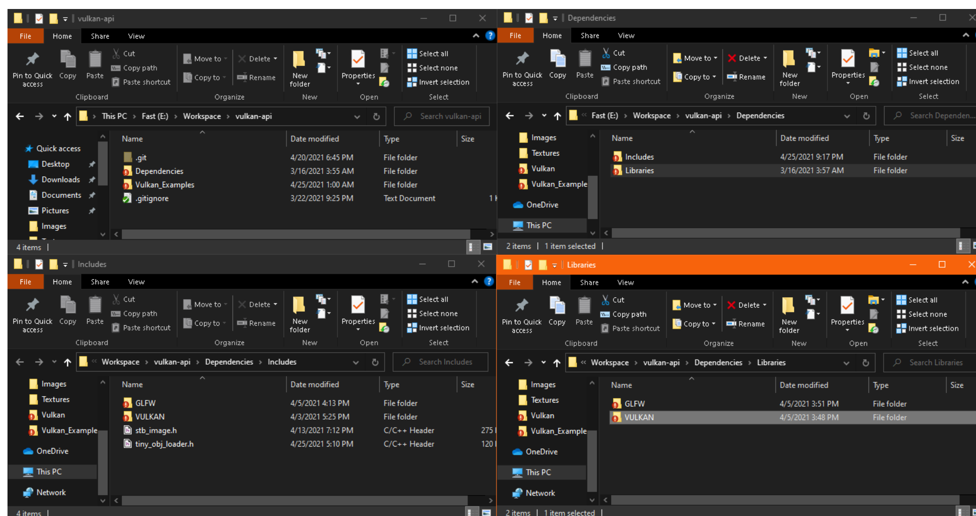
Obr. 5.2: Pripojenie dodatočných hlavičkových súborov. Foto autor.



Obr. 5.3: Pripojenie dodatočných knižníc. Foto autor.



Obr. 5.4: Nastavenie mien knižníc. Foto autor.



Obr. 5.5: Roztriedenie hlavičkových a knižničných súborov do priečinkov. Foto autor.

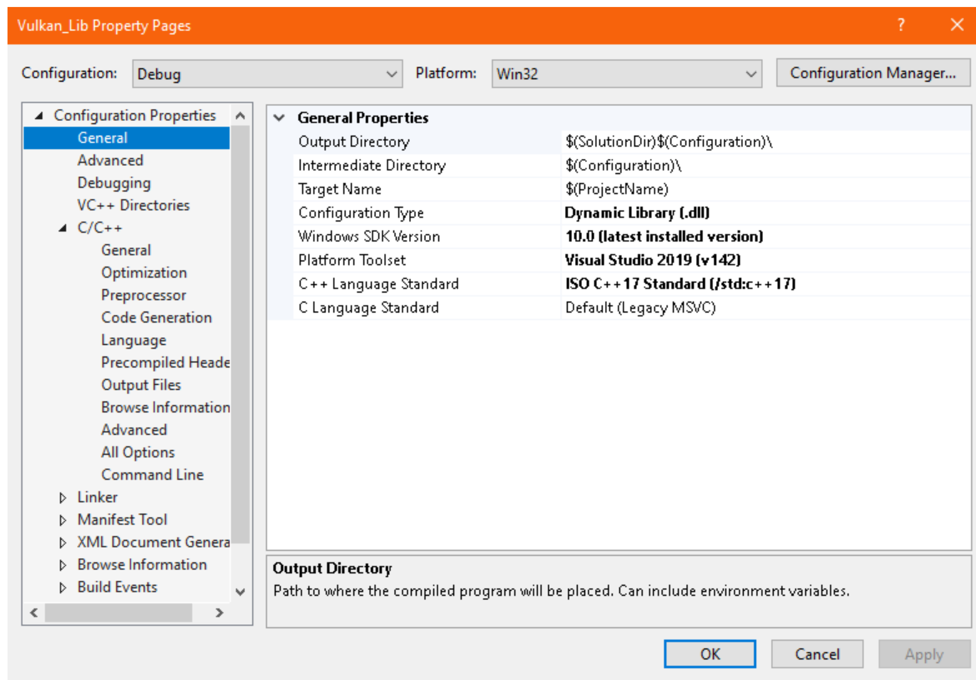
executable 5.6. Nakoniec nastavíme závislosť projektov na sebe, kde všetky executable projekty budú závislé na DLL projekte a nastavíme ktorý projekt budeme spúšťať po úspešnom builde[10]. Nesmieme ale zabudnúť pri executable projektoch linkovať navyše aj DLL projekt.

5.2 Základná schéma nastavenia Vulkanu

Vulkan API používa pri vytváraní rôznych objektov štruktúry, ktoré zhromažďujú dáta potrebné na vytvorenie iných štruktúr. Napr. na vytvorenie VkInstance, potrebujeme vytvoriť VkInstanceCreateInfo, v ktorom definujeme VkApplicationInfo, čo je ďalšia štruktúra, ktorej naplnenie je v podstate nepotrebné, no Vulkan sa vie lepšie prispôsobiť aplikácií, ak budeme čo najviac konkrétni pri vytváraní štruktúr. Celý proces pripravenia Vulkanu na rendering je oproti OpenGL určite zdĺhavejší, už na prvý pohľad. Na vyrendrovanie jedného trojuholníka pomocou Vulkan API budeme potrebovať cca. tisíc riadkov kódu. Rendrovací cyklus je ale kratší oproti kódu, v ktorom Vulkan nastavujeme. Celé nastavenie Vulkanu vieme hierarchicky rozdeliť následovne:

1. vytvorenie inštancie Vulkanu, kde špecifikujeme požadované rozšírenia Vulkanu a vrstvy pre ľahší debugging,
2. prepojenie Surface objektu Vulkanu s window systémom(u nás GLFW),
3. výber správneho fyzického zariadenia,

[10]Toto nastavujeme ale vo vlastnostiach solution



Obr. 5.6: Nastavenie typu konfigurácie. Foto autor.

4. vytvorenie logického zariadenia a získanie Queue(radov) z každej potrebnej queue família,

5. vytvorenie SwapChainu a získanie VkImage z neho,

6. vytvorenie príkazových bufferov a memory poolov, CommandPool a CommandBuffer,

7. vytvorenie synchronizačných prvkov,

8. vytvorenie bufferov, descriptor setov a uniform bufferov pre modely,

9. vytvorenie pipelineov, a načítanie shaderov do nich, spolu s uložením vertexov a indexov geometrie,

Podľa tejto schémy, môžme inicializáciu Vulkanu rozdeliť na tri štádiá. Prvé štádium budeme volať inicializácia Vulkanových dát, ktorá sa bude starať o inicializáciu VkInstance, linkovanie VkSurface, výber fyzického zariadenia a inicializácia logického zariadenia. Druhé štádium bude inicializácia rendrovacích dát a postupne nastaví VkSwapchain, VkRenderPass a VkFramebuffer. Posledné štádium inicializuje prvky používané priamo pri rendrovacích cykloch ako sú VkCommandPool, VkCommandBuffer, VkSemaphore, VkFence. Tu sme však použili rovno niekoľko pojmov, s ktorými sme sa pri OpenGL nemuseli stretnúť. Preto si ich vysvetlime:

- VkInstance ukladá informácie o stave Vulkanu a nastavuje zariadenia.
- VkLayer slúžia na debugging, a poskytujú nám dodatočné informácie, pri chode programu, no v podstate sú na chod programu nepotrebné, a vystačíme si s MSVS debuggerom.
- VkPhysicalDevice fyzické zariadenie Vulkanu, je reálne fyzické zariadenie, kus hardvéru, pomocou ktorého budeme rendrovať.
- VkQueue rady sú rôznych typov, rady sa zameriavajú každý na svoju podmnožinu operácií, sú napr. výpočtové, alebo grafické rady. Rady zameriavajúce sa na rovnaké operácie voláme queue families.
- VkDevice nazývame aj logické zariadenie, a predstavuje nastavenie fyzického zariadenia, ktoré budeme používať. Definujeme pri ňom aké rozšírenia a rady budeme používať.
- VkSurface je povrch, resp. plocha na ktorú chceme rendrovať, a musí byť zlinkovaná medzi Vulkanom a oknovým systémom, u nás je oknový systém realizovaný knižnicou GLFW,
- VkSwapchain je rozšírenie, ktoré umožňuje zadanie rendrovacích príkazov pre viaceré obrázky, ktoré sa postupne rendrujú, zatiaľ čo aplikácia môže pracovať na hlavnom vlákne na riešení svojich problémov, prípadne môže nastavovať rendrovacie commandz pre iné obrázky zo swapchainu.
- VkFramebuffer spája attachmenty so swapchain imageView.
- VkRenderPass definuje do ktorých attachmentov budeme rendrovať a ako.
- VkCommandPool vytvára miesto v pamäti na alokáciu bufferov.
- VkCommandBuffer vytvárame pomocou command poolov, a ukladajú príkazy pre daný rad, ktoré bude musieť rad vykonať, aby bol považovaný za ukončený.
- VkSemaphore slúžiaci na synchronizáciu prostriedkov medzi GPU a GPU, napr. na určenie pre prezentačný rad kedy grafický rad ukončil svoju činnosť.
- VkFence slúžiaci na synchronizáciu prostriedkov medzi GPU a CPU, napr. na určenie či už obrázok bol odprezentovaný.

Toto sú najhlavnejšie pojmy, ktoré budeme používať. Ďalej budeme ale musieť pracovať s obrázkami, napr. pri načítavaní textúr alebo získavaní obrázkov zo swapchainu. V takom prípade je na najspodnejšej vrstve VkMemory, ktorá len surovo vymedzuje niekoľko bytov pamäte, do ktorej zapisujeme dáta. Nad

touto štruktúrou vedieme napr. VkImage, ktorá umožňuje pristupovať do pamäte po texeloch namiesto po bytoch. A napokon VkImage zabaľujeme do VkImageView, ktorá umožňuje vyberať z VkImage dáta po zónach.

5.3 Implementácia matematických a pomocných štruktúr

Súčasťou celej implementácie bola aj matematická knižnica obsahujúca matice, vektory a kvaternióny, a aj iných podporných funkcií pre ďalší development renderera. Pri implementácií som sa rozhodol použiť c++ šablóny a definovať triedu Vector ako šablónovú triedu, kde šablóna definuje typ a veľkosť. To znamená že Vector<float, 4> bude štruktúra obsahujúca štyri premenné typu float. Výhodou tohto prístupu je že nemusíme definíciu písať pre vektory rôznych veľkostí a typov. Funkcia sprostredkujúca sčítanie vektorov môže vyzerať následovne.

```
1 Vector<Type, dimension>& operator += (const Vector<Type, dimension>&
      source) {
2   for (int i = 0; i < dimension; i++) {
3     mBuffer[i] += source.mBuffer[i];
4   }
5     return *this;}
```

Špeciálne funkcie ako cross product, alebo getAngle pomocou ktorej získame uhol zvieraný medzi dvoma 3D vektormi, vieme definovať pomocou šablónových špecializácií, len pre vectory obsahujúce tri prvky.

```
1 template<typename Type>
2 Vector<Type, 3> cross(const Vector<Type, 3>& v1, const Vector<Type,
     3>& v2) {
3   Vector<Type, 3> vector(
4     v1[1] * v2[2] - v1[2] * v2[1],
5     v1[2] * v2[0] - v1[0] * v2[2],
6     v1[0] * v2[1] - v1[1] * v2[0]
7   );
8   return vector;}
```

Posledne, pri vektoroch, budeme potrebovať spôsob ako im pridať dimenziu. Táto funkcionalita je realizovaná šablónovými funkciami addDimension a stripDimension.

```
1 template<typename Type, uint8_t dimension>
2 Vector<Type, dimension + 1> addDimension(const Vector<Type,
     dimension>& vec, const Type& newVal) {
3   Vector<Type, dimension + 1> v(0.0);
4   for (int i = 0; i < dimension; i++) {
5     v[i] = vec[i];
6   }
7   v[dimension] = newVal;
8   return v;}
```

Matice som navrhol takisto ako šablónovú triedu, ktorá obsahuje staticky alokované pole ako členskú premennú. Jej veľkosť je daná znova pomocou šablónového parametru dimension ako $size = dimension^2$. Funkcie ako sú createRotation<Os>Matrix, definujúce rotácie okolo určitej osy, createViewMatrix, createPerspektiveMatrix, createOrthographicMatrix, createScalingMatrix a createShiftMatrix sú špecializované na matice dimenzie 4. Funkcie vytvárajúce rotačné matice sú createRotationXMatrix, createRotationYMatrix, createRotationZMatrix.

```
1  template<typename Type>
2  Matrix<Type, 4> createRotationMatrix(const Vector<Type, 3>& axis,
       const Type& angle, TemplateReturnTakeFunc<Type> sinFunc =
       nullptr, TemplateReturnTakeFunc<Type> cosFunc = nullptr) {
3    Matrix<Type, 4> mat;
4    TemplateReturnTakeFunc<Type> sFunc = (sinFunc) ? sinFunc : mSin<
       Type>;
5    TemplateReturnTakeFunc<Type> cFunc = (cosFunc) ? cosFunc : mCos<
       Type>;
6    Type ang = toRad(angle);
7    Type cos = cFunc(ang);
8    Type sin = sFunc(ang);
9    mat = Matrix<Type, 4>(cos, -sin, 0, 0,
10     sin, cos, 0, 0,
11     0, 0, 1, 0,
12     0, 0, 0, 1
13     );
14   return mat;
15 }
```

Poslednou štruktúrou boli kvaternióny, ktoré som znova implementoval ako šablónovú triedu. Tu však nebolo potrebné použiť špecializované šablóny, pretože už z definície kvaterniónov vždy obsahujú reálnu časť a vektor špecifikujúci imaginárne časti kvaterniónu. Okrem konštruktorov a matematických operátorov som implementoval funkcie na vytvorenie kvaterniónu rotácie pomocou dvoch funkcií. Prvá z týchto funkcií vytvára rotačný kvaternión pozdĺž jednej osy o daný uhol a druhá funkcia postupne vyskladá rotácie pozdĺž všetkých ôs.

```
1  template<typename Type>
2  Quaternion<Type> createRotation(const Vector<Type, 3>& axis, const
       Type& angle, TemplateReturnTakeFunc<Type> sqrtFunc = nullptr,
       TemplateReturnTakeFunc<Type> sinFunc = nullptr,
       TemplateReturnTakeFunc<Type> cosFunc = nullptr) {
3    double a = toRad(angle);
4    Vector<Type, 3> ax = axis.getNormalized(sqrtFunc);
5    TemplateReturnTakeFunc<Type> sFunc = (sinFunc) ? sinFunc : mSin<
       Type>;
6    TemplateReturnTakeFunc<Type> cFunc = (cosFunc) ? cosFunc : mCos<
       Type>;
7    Quaternion<Type> rotQuat(cFunc(a * 0.5),
8      Vec3(ax[0] * sFunc(0.5 * a),
9        ax[1] * sFunc(0.5 * a),
```

```
      ax[2] * sFunc(0.5 * a)
    )
  );
  return rotQuat;
}
```

Napokon pre zjednodušenie rotácií realizovnaných pomocou kvaterniónov, som implementoval funkciu applyRotation, ktorej výstupom je zrotovaný 3D vektor.

```
template<typename Type>
Vector<Type, 3> applyRotation(const Vector<Type, 3>& vector, const
    Quaternion<Type>& q) {
  Quaternion<Type> qInv = q.getInverse();
  Quaternion<Type> res = q * Quaternion(vector);
  res = res * qInv;
  return res.getVectorPart();
}
```

V rámci pomocných štruktúr som implementoval triedu Logger, ktorú inštancujem ako globálnu premennú. Táto štruktúra mi umožňuje zapísať doň chybovú správy, ktoré sa v poradí zasielania za seba spájajú metódou linked listu. Po skončení programu sa zavolá jeho deštruktor, ktorý postupne dealokuje všetky nodey a zapisuje do textového súboru. Mimo tejto štruktúry som implementoval namespace Type, kde sú premenované všetky premenné cez typedef, vďaka čomu viem po implementácií napr. vlastnej triede String jednoducho zmeniť používaný typ v celom programe.

5.4 Implementácia inicializácie

Prvé čo pri inicializácií musíme spraviť je samozrejme vytvoriť okno. To je vďaka GLFW jednoduché, no musíme si nájsť a použiť správne vlajky pred samotným vytvorením. Nám sa jedná najmä aby sme okno počas behu aplikácie nemohli zväčšovať alebo zmenšovať. Funkcionalita síce sama o sebe znie lákavo ale vyžadovala by v mojej implementácií ošetrenie viacero situácií. Museli by sme napr. nanovo vytvoriť swapchain, alebo všetky pipeliny.

```
bool Window::create(Type::Uint16 width, Type::Uint16 height){
  glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
  glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
  window = glfwCreateWindow(width, height, APP_NAME, nullptr,
    nullptr);
  if (!window)
    return false;
  return true;
}
```

Po vytvorení okna sa môžme vrhnúť na Vulkan. Implementáciu inicializácie som, ako som spomenul vyššie, rozdelil na tri časti. V tejto sekcií sa budeme venovať inicializácií základných Vulkanových dát. Túto časť som implementoval v triede VulkanData a implementuje vytvorenie VkInstance, linkovanie

VkSurfaceKHR, výber fyzického zariadenia a vytvorenie logického zariadenia. Na vytvorenie VkInstance, potrebujeme poznať v podstate len rozšírenia Vulkanu. Vulkan od nás žiada aby sme pri vytváraní VkInstance boli čo najviac explicitní, pomocou čoho sa vie ale lepšie prispôsobiť naším aplikáciám. Preto som pri vytváraní VkInstance naplnil aj VkApplicationInfo, ktorému len špecifikujeme verziu našej aplikácie a verziu Vulkan API.

```
1 bool VulkanData::initializeInstance(){
2     VkApplicationInfo appInfo;
3     appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
4     appInfo.pApplicationName = APP_NAME;
5     appInfo.applicationVersion = VK_MAKE_VERSION(VERSION_MAJOR,
      VERSION_MINOR, VERSION_PATCH);
6     appInfo.pEngineName = "No Engine";
7     appInfo.engineVersion = VK_MAKE_VERSION(VERSION_MAJOR,
      VERSION_MINOR, VERSION_PATCH);
8     appInfo.apiVersion = VK_API_VERSION_1_0;
9 
10    VkInstanceCreateInfo createData;
11    createData.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
12    createData.pApplicationInfo = &appInfo;
13    auto extensions = getRequiredExtensions();
14    createData.enabledExtensionCount = static_cast<uint32_t>(
      extensions.size());
15    createData.ppEnabledExtensionNames = extensions.data();
16    createData.enabledLayerCount = 0;
17    createData.pNext = nullptr;
18    if (vkCreateInstance(&createData, nullptr, &pInstance) !=
      VK_SUCCESS) {
19        return false;
20    }
21    return true;
22 }
```

Po vytvorení VkInstance, som prelinkoval surface medzi Vulkanom a GLFW. To môžme robiť viacerými spôsobmi no mne sa osvedčilo linkovanie pomocou GLFW.

```
1 bool Window::linkVulkanSurface(VkInstance instance, VkSurfaceKHR&
    surface){
2   if (glfwCreateWindowSurface(instance, window, nullptr, &surface)
    != VK_SUCCESS)
3     return false;
4   return true;
5 }
```

Vulkan môže rendrovať na viacerých grafických kartách a zároveň som chcel používať prezentačný rad, ktorý niektoré grafické karty nemajú. Vulkan vie rendrovať ako na integrovaných aj na dedikovaných grafických kartách. Preto musíme vedieť pre našu aplikáciu zvoliť správne zariadenie. Informácie o Vulkanom podporovaných zariadeniach vieme získať volaním funkcie vkEnumeratePhysicalDevices(. . .). Preto som pri implementácií zvolil zariadenie pomocou

postupného hodnotenia, ktorého výsledky som ukladal do std::map a zvolil zariadenie s najlepším hodnotením. Pri hodnotení vo funkcií getDeviceRatings(...) som do hodnotenia započítaval podporované rozšírenia, existenciu prezentačného radu, veľkosť pamäte a udeľoval som bonusové body ak bolo zariadenie označené vlajkou VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU. Po výbere zariadenia s najlepším hodnotením mu rovno nájdeme indexy ku grafickému a prezentačnému radu a napokon zistíme aké formáty swapchainu podporuje.

```
1  bool VulkanData::pickPhysicalDevice(){
2      ...
3      std::vector<VkPhysicalDevice> devices(deviceCount);
4      vkEnumeratePhysicalDevices(pInstance, &deviceCount, devices.data
       ());
5      std::map<Type::Uint32, VkPhysicalDevice> ratings;
6      for (const auto& device : devices) {
7          ratings[getDeviceRating(device, pSurface, pDeviceExtensions)
       ] = device;
8      }
9      if (ratings.rbegin()->first > 0) {
10         pPhysicalDevice = ratings.rbegin()->second;
11     }
12     ...
13     return true;
14 }
```

Posledne v tejto časti musíme vytvoriť VkDevice. Najprv musíme naplniť VkDeviceQueueCreateInfo pre každý rad ktorý potrebujeme. Po naplnení VkDeviceQueueCreateInfo štruktúr musíme špecifikovať VkPhysicalDeviceFeatures, v ktorom chceme zapnút samplerAnisotropy, pretože chceme textúrovým samplerom zapnúť anisotropické filtrovanie. Napokon vytvoríme VkDeviceCreateInfo a naplníme ho štruktúrami, ktoré sme práve vytvorili. Nakoniec získame rady, VkQueue, ktoré sme príkazom vkCreateDevice vytvorili.

```
1  bool VulkanData::createLogicalDevice(){
2      std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
3      std::vector<uint32_t> uniqueQueueFamilies = pQueueIndices.
       getPackedAsVector();
4
5      float queuePriority = 1.0f;
6      for (Type::Uint32 queueFamily : uniqueQueueFamilies) {
7          VkDeviceQueueCreateInfo queueCreateInfo{};
8          queueCreateInfo.sType =
       VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
9          queueCreateInfo.queueFamilyIndex = queueFamily;
10         queueCreateInfo.queueCount = 1;
11         queueCreateInfo.pQueuePriorities = &queuePriority;
12         queueCreateInfos.push_back(queueCreateInfo);
13     }
14
15     VkPhysicalDeviceFeatures deviceFeatures{};
16     deviceFeatures.samplerAnisotropy = VK_TRUE;
```

```
17
18     VkDeviceCreateInfo createData{};
19     createData.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
20     createData.queueCreateInfoCount = Type::Uint32(queueCreateInfos.
       size());
21     createData.pQueueCreateInfos = queueCreateInfos.data();
22     createData.pEnabledFeatures = &deviceFeatures;
23     createData.enabledExtensionCount = Type::Uint32(
       pDeviceExtensions.size());
24     createData.ppEnabledExtensionNames = pDeviceExtensions.data();
25     createData.enabledLayerCount = 0;
26
27     if (vkCreateDevice(pPhysicalDevice, &createData, nullptr, &
       pDevice) != VK_SUCCESS) {
28         return false;
29     }
30
31     vkGetDeviceQueue(pDevice, pQueueIndices.graphics.value(), 0, &
       pQueues.graphics);
32     vkGetDeviceQueue(pDevice, pQueueIndices.presentation.value(), 0,
        &pQueues.presentation);
33     return true;
34 }
```

5.5 Nastavenie rendrovacích prvkov

Časť implementácie, ktorá se venuje inicializácií rendrovacích dát ako sú swapchain, framebuffer, depthbuffer a renderpass som umiestnil do triedy VulkanRenderData. Najprv v tomto štádiu vyberiem konkrétny formát obrazu, farebného priestoru a zóny obrazu[11] pre framebuffer pomocou funkcie chooseBestSwapchainDetails. Po správnom zvolení formátov, môžem začať vytvárať swapchain, pri čom budem potrebovať práve tieto formáty. Nastavíme použitie obrázka, vieme nastaviť aj transformácie, ktoré pred prezentáciou obrazu spraví prezentačný rad. Ak by sme potrebovali vytvoriť nový swapchain[12] vieme tu špecifikovať starý swapchain ako parameter VkSwapchainCreateInfoKHR::oldSwapchain. Posledne musíme správne zvoliť zdieľanie obrazu medzi radmi, ktorá sa líši podľa toho, či máme indexy radov zhodné, alebo rozdielne.

```
1 Type::Uint32 imageCount = swapchainSupport.capabilities.
      minImageCount + 1;
2 if (swapchainSupport.capabilities.maxImageCount > 0 &&
3     imageCount > swapchainSupport.capabilities.maxImageCount) {
4     imageCount = swapchainSupport.capabilities.maxImageCount;
5 }
6
7 VkSwapchainCreateInfoKHR createData{};
8 createData.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
```

[11] V tomto prípade to bude veľkosť okna.

[12] Napr. pri zväčšení alebo zmenšení okna.

```
9  createData.surface = surface;
10
11 createData.minImageCount = imageCount;
12 createData.imageFormat = pSwapchainDetails.format.format;
13 createData.imageColorSpace = pSwapchainDetails.format.colorSpace;
14 createData.imageExtent = pSwapchainDetails.extent;
15 createData.imageArrayLayers = 1;
16 createData.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
17
18 std::vector<Type::Uint32> queueFamilyIndices = queueIndices.
       getPackedAsVector();
19 if (queueIndices.graphics.value() != queueIndices.presentation.value
       ()) {
20     createData.imageSharingMode = VK_SHARING_MODE_CONCURRENT;
21     createData.queueFamilyIndexCount = Type::Uint32(
       queueFamilyIndices.size());
22     createData.pQueueFamilyIndices = queueFamilyIndices.data();
23 } else {
24     createData.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
25 }
26 createData.preTransform = swapchainSupport.capabilities.
       currentTransform;
27 createData.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
28 createData.presentMode = pSwapchainDetails.presentationMode;
29 createData.clipped = VK_TRUE;
30 createData.oldSwapchain = VK_NULL_HANDLE;
31
32 if (vkCreateSwapchainKHR(logicalDevice, &createData, nullptr, &
       pSwapchain) != VK_SUCCESS)
33     return false;
```

Po vytvorení VkSwapchainKHR musíme od Vulkanu získať VkImage a z nich zase vytvoriť VkImageView. Na získanie VkImage zavoláme funkciu vkGetSwapchainImagesKHR, najprv na zistenie počtu VkImage, ktoré Vulkan vytvoril a potom znova na získanie už potrebných VkImage. Na vytvorenie VkImageView zo získaných VkImage mi stačí zavolať pre každý získaný VkImage zavolať funkciu createImageView, ktorú som implementoval v súboroch Common.h a Common.cpp a používam ju vždy pri vytváraní VkImage.

```
1  Type::Uint32 count;
2  vkGetSwapchainImagesKHR(logicalDevice, pSwapchain, &count, nullptr);
3  pSwapchainImages.resize(count);
4  vkGetSwapchainImagesKHR(logicalDevice, pSwapchain, &count,
       pSwapchainImages.data());
5  ...
6  pImageViews.resize(pSwapchainImages.size());
7  for (Type::Uint32 i = 0; i < pImageViews.size(); i++) {
8      if (!createImageView(logicalDevice, pSwapchainImages[i],
       pSwapchainDetails.format.format, K_IMAGE_ASPECT_COLOR_BIT,
       pImageViews[i])) {
9          return false;
10     }
11 }
```

Po implementácií inicializácie swapchainu potrebujeme inicializovať VkRenderPass. Inicializácia VkRenderPass je realizovaná naplňovaním príloh (attachments), ktoré definujú napr. počet vzorkov, spôsob zachádzania s attachmentom pred a po subpasse alebo konečné a počiatočné layout prílohy.

```
std::array<VkAttachmentDescription, 2> attachments;
attachments[0].format = pSwapchainDetails.format.format;
attachments[0].samples = VK_SAMPLE_COUNT_1_BIT; // 1 sample per
    fragment
attachments[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR; //
    VK_ATTACHMENT_LOAD_OP_DONT_CARE
attachments[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE; //
    VK_ATTACHMENT_STORE_OP_DONT_CARE
attachments[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;  //
    Wont care about stencil
attachments[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;//
    Wont care about stencil
attachments[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;   //
    Layout before/after render, here we dont care
attachments[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR; //
    Layout before/after render, here its for swapchain
...
VkAttachmentReference colorAttachmentRef{};
colorAttachmentRef.attachment = 0;
colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL
    ;
VkAttachmentReference depthAttachmentRef{};
depthAttachmentRef.attachment = 1;
depthAttachmentRef.layout =
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

VkSubpassDescription subpass{};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorAttachmentRef;
subpass.pDepthStencilAttachment = &depthAttachmentRef;

VkSubpassDependency dependency{};
dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
dependency.dstSubpass = 0;
dependency.srcStageMask =
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT |
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
dependency.srcAccessMask = 0;
dependency.dstStageMask =
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT |
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
dependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;

VkRenderPassCreateInfo renderPassCreationData{};
renderPassCreationData.sType =
    VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassCreationData.attachmentCount = static_cast<uint32_t>(
```

```
    attachments.size());
35 renderPassCreationData.pAttachments = attachments.data();
36 renderPassCreationData.subpassCount = 1;
37 renderPassCreationData.pSubpasses = &subpass;
38 renderPassCreationData.dependencyCount = 1;
39 renderPassCreationData.pDependencies = &dependency;
40
41 if (vkCreateRenderPass(logicalDevice, &renderPassCreationData,
    nullptr, &pRenderPass) != VK_SUCCESS) {
42     return false;
43 }
```

V tomto štádiu môžme nastaviť depthbuffer, kde nám stačí nájsť požadovaný formát, a vytvoriť VkImage a VkImageView. Oproti predošlým VkImage tu musíme zmeniť VkImageUsageFlags a použiť VK_IMAGE_USAGE _DEPTH_STENCIL_ATTACHMENT_BIT a pri vytváraní VkImageView zmeniť VkImageAspectFlags na VK_IMAGE_ASPECT_DEPTH_BIT.

```
1 if (!findSupportedDepthFormat(physicalDevice,
2     { VK_FORMAT_D32_SFLOAT, VK_FORMAT_D32_SFLOAT_S8_UINT,
    VK_FORMAT_D24_UNORM_S8_UINT },
3     VK_IMAGE_TILING_OPTIMAL,
    VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT, pDepthFormat)) {
4     return false;
5 }
6 bool hasStencil = pDepthFormat == VK_FORMAT_D32_SFLOAT_S8_UINT ||
    pDepthFormat == VK_FORMAT_D24_UNORM_S8_UINT;
7 if (!createImage(logicalDevice, physicalDevice, pSwapchainDetails.
    extent.width, pSwapchainDetails.extent.height,
8     pDepthFormat, VK_IMAGE_TILING_OPTIMAL,
    VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
9     pDepthImage, pDepthImageMemory)) {
10     return false;
11 }
12 if (!createImageView(logicalDevice, pDepthImage, pDepthFormat,
    VK_IMAGE_ASPECT_DEPTH_BIT, pDepthImageView)) {
13     return false;
14 }
15 return true;
```

Posledne som vytvoril framebuffer, pre každý VkImage vytvorený zo swapchainu, pričom ako prílohy som použil VkImageView swapchainu a depthbuffera.

```
1 pFramebuffers.resize(pImageViews.size());
2 for (Type::Uint32 i = 0; i < pImageViews.size(); i++) {
3     std::array<VkImageView, 2> attachments = { pImageViews[i],
    pDepthImageView };
4     VkFramebufferCreateInfo createData{};
5     createData.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
6     createData.renderPass = pRenderPass;
7     createData.attachmentCount = attachments.size();
8     createData.pAttachments = attachments.data();
```

```
9      createData.width = pSwapchainDetails.extent.width;
10     createData.height = pSwapchainDetails.extent.height;
11     createData.layers = 1;
12     if (vkCreateFramebuffer(logicalDevice, &createData, nullptr, &
       pFramebuffers[i]) != VK_SUCCESS) {
13         return false;
14     }
15 }
```

5.6 Implementácia dát rendrovacieho cyklu

V tejto fázy som ako prvý vytvoril VkMemoryPool, pre ktorý stačí vo VkCommandPoolCreateInfo uviesť len index grafického radu.

```
1 createData.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
2 createData.queueFamilyIndex = indices.graphics.value();
3 if (vkCreateCommandPool(logicalDevice, &createData, nullptr, &
      pCommandPool) != VK_SUCCESS) {
4     return false;
5 }
6 return true;
```

Pri rendrovaní, kvôli swapchainu potrebujem synchronizovať činnosť medzi CPU a GPU, aby som vedel spravovať, ktoré VkImageView swapchain spracoval a ktoré odprezentoval. Nato mi poslúžia VkFence a VkSemaphore. Tu si zavedieme premennú maxFramesInFlight, ktorá nám bude uvádzať maximálny počet súbežne spracovávaných obrázkov.

```
1 VkSemaphoreCreateInfo semaphoreInfo{};
2 semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
3 VkFenceCreateInfo fenceInfo{};
4 fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
5 fenceInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;
6 for (size_t i = 0; i < maxFramesInFlight; i++) {
7     if (vkCreateSemaphore(logicalDevice, &semaphoreInfo, nullptr, &
      pImageAvailableSemaphores[i]) != VK_SUCCESS ||
8         vkCreateSemaphore(logicalDevice, &semaphoreInfo, nullptr, &
      pRenderFinishedSemaphores[i]) != VK_SUCCESS ||
9         vkCreateFence(logicalDevice, &fenceInfo, nullptr, &
      pInFlightFences[i]) != VK_SUCCESS) {
10        return false;
11    }
12 }
```

V rámci rendrovacieho cyklu, som musel vytvoriť VkCommandBuffer, pomocou ktorého som schopný pridať príkaz k rendrovaniu na rad. Vytvorenie VkCommandBuffera je opäť v tomto prípade jednoduché, stačí predať VkCommandBufferAllocateInfo vytvorený VkCommandPool a špecifikovať počet VkCommandBufferov, ktoré chceme vytvoriť. Chceme pre každý obrázok swapchainu vytvoriť jeden VkCommandBuffer, inak by swapchain nemal v podstate zmysel.

```
pCommandBuffers.resize(framebufferCount);
VkCommandBufferAllocateInfo allocationData{};
allocationData.sType =
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
allocationData.commandPool = pCommandPool;
allocationData.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
allocationData.commandBufferCount = Type::Uint32(pCommandBuffers.
    size());
if (vkAllocateCommandBuffers(logicalDevice, &allocationData,
    pCommandBuffers.data()) != VK_SUCCESS) {
    return false;
}
return true;
```

5.7 Shadery

Shadery vo vulkane kompilujeme do SPIR-V formy, do ktorej kompilujeme pomocou kompilátora stiahnutého pri sťahovaní Vulkan SDK a nachádza sa v lokácií

...\VulkanSDK\verzia[13]\Bin32\glslc.exe.

Kompiláciu shaderov, ale môžme realizovať aj mimo nášho programu, naozaj sa jedná o zavolanie kompilátoru glslc.exe z príkazového riadku, ktorému ako prepínače dáme názov vstupného kompilovaného súboru, ktorý je nasledovaný prepínačom -o a názvom výstupného súboru. Pri implementácií som sa rozhodol implementovať aj automatickú kompiláciu shaderov pomocou funkcie system, ktorému zadám string reprezentujúci príkaz. V makre SPIR_V_LOCATION je zapísaná absolútna cesta ku glslc.exe.

```
void Shader::compileShaders(const Type::String& filename) {
    ...
    Type::String command = SPIR_V_LOCATION;
  command.append(filename);
  command.append(SWITCHES);
  command.append(mCompiledName);
  system(command.c_str());
}
```

Pre shadery som spravil samostatnú triedu Shader, ktorá je schopná skompilovať a načítať shadery, spôsobom spomenutým vyššie, a vytvoriť z nich VkShaderModule.

```
VkShaderModule Shader::createShaderModule(VkDevice logicalDevice) {
  VkShaderModuleCreateInfo creationData{};
  creationData.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
  creationData.codeSize = mBuffer.size();
  creationData.pCode = reinterpret_cast<const uint32_t*>(mBuffer.
    data());
```

[13]pre mňa to bola verzia 1.2.170.0

```
6
7   VkShaderModule shaderModule;
8   if (vkCreateShaderModule(logicalDevice, &creationData, nullptr, &
      shaderModule) != VK_SUCCESS) {
9     mFailBit ^= uint8_t(Errors::FAILED_LOAD);
10  }
11  return shaderModule;
12 }
```

Ďalšia trieda ShaderLink zase linkuje navzájom viacero shaderov a plní VkPipelineShaderStageCreateInfo pre každý shader.

```
1 void ShadersLink::fillVertexStageInfo(
      VkPipelineShaderStageCreateInfo& stageInfo) {
2   stageInfo.sType =
      VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
3   stageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
4   stageInfo.module = mShaderModules.vertex;
5   stageInfo.pName = "main";
6 }
```

Shadery potrebujeme priamo pri vytváraní VkPipeline, čo znamená že ak chceme používať v aplikácií viacero shaderov budeme musieť mať viacero VkPipeline. Zo strany programovania shaderov syntax zostáva rovnaká ako v GLSL. Rozdiel je len v tom, že sa kompilujú do SPIR-V bajtkódu. Znova budeme mať vstupné a výstupné premenné in a out. Navyše však pre ne budeme musieť definovať layout location.

```
1 ...
2 layout(location = 0) in vec3 inPosition;
3 layout(location = 1) in vec3 normal;
4 layout(location = 2) in vec2 texCoords;
5 layout(location = 3) in vec3 color;
6 layout(location = 0) out vec3 fragColor;
7 layout(location = 1) out vec2 fragTexCoord;
8 ...
```

V tomto prípade sú location parametre premenných inPosition, normal, texCoords a color nastavené podľa štruktúry VulkanVertex do VkVertexInputBindingDescription a do poľa štruktúr VkVertexInputAttributeDescription. Tie predávame VkPipeline pri jej vytváraní, čo znova viaže jeden binding. VkVertexInputBindingDescription určuje veľkosť jedného vertexu, resp. vymedzuje čo je 1 vektor pre Vulkan a VkVertexInputAttributeDescription popisuje kde vrámci jedného vertexu sa daná informácia nachádza, location pre priradenie hodnoty do správnej premennej a formát premennej.

```
1 VkVertexInputBindingDescription VulkanVertex::getBindingDescription
      () {
2   VkVertexInputBindingDescription description{};
3   description.binding = 0;
4   description.stride = sizeof(VulkanVertex);
5   description.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
6   return description;
```

```
}
std::vector<VkVertexInputAttributeDescription> VulkanVertex::
    getAtributeDescription() {
  std::vector<VkVertexInputAttributeDescription> description(4);
  description[0] = {};
  description[0].binding = 0;
  description[0].location = 0;
  description[0].format = VK_FORMAT_R32G32B32_SFLOAT;
  description[0].offset = offsetof(VulkanVertex, position);

  description[1] = {};
  description[1].binding = 0;
  description[1].location = 1;
  ...
}
```

Pri output parametroch však musíme mať parameter location zhodný s location input parametrom shaderu v následujúcej fáze pipelineu, u nás to je fragment shader.

```
layout(location = 0) in vec3 fragColor;
layout(location = 1) in vec2 fragTexCoord;
```

Špecifikátor uniform používame naďalej pri sampleroch a pri uniformných premenných. Tu musíme však navyše nastaviť binding pri nastavovaní VkDescriptorSetLayout, ktorý znova potrebujeme poskytnúť VkPipeline pri vytváraní. Ďalej môžme používať pri vektoroch v shaderoch "skrátenú syntax", napr.:

vec4 x = vec4(vec1.rr, vec2.gb);

Posledne nesmieme zabudnúť vždy na začiatku shaderu špecifikovať verziu shaderu a zapnúť rozšírenie na oddeľovanie shader objektov.

```
#version 450
#extension GL_ARB_separate_shader_objects : enable
```

5.8 Načítavanie modelov

Načítavanie modelov je implementované členskou funkciou triedy VulkanModel, loadModel. Vďaka hlavičkovej knižnici Tiny Object Loader je tento proces jednoduchý. Knižnica dokonca poskytuje funkcionality na export informácií o materiáloch objektu, ktoré môžme neskôr použiť. Mňa však zaujímal hlavne export dát vertexov, ich indexov a textúrových koordinátov. Čítanie zo súboru je implementované v jednej funkcií tinyobj::LoadObj. Musíme však inštancovať všetky vstupné parametre, z ktorých budeme potrebovať parametre shapes a attrib.

```
bool loadModelFromFile(Type::String&& filename, std::vector<
    VulkanVertex>& vertices, std::vector<Index>& indices) {
  tinyobj::attrib_t attrib;
  std::vector<tinyobj::shape_t> shapes;
```

```
  std::vector<tinyobj::material_t> materials;
  std::string warn, err;
  if (!tinyobj::LoadObj(&attrib, &shapes, &materials, &warn, &err,
    filename.c_str())) {
    return false;
  }
  for (const auto& shape : shapes) {
    for (const auto& index : shape.mesh.indices) {
      VulkanVertex vertex{};
      vertex.position = {
        attrib.vertices[3 * index.vertex_index + 0],
        attrib.vertices[3 * index.vertex_index + 1],
        attrib.vertices[3 * index.vertex_index + 2]
      };
      vertex.texCoords = {
        attrib.texcoords[2 * index.texcoord_index + 0],
        1.0f - attrib.texcoords[2 * index.texcoord_index + 1]
      };
      vertex.color = { 1.0f, 1.0f, 1.0f };

      vertices.push_back(vertex);
      indices.push_back(indices.size());
    }
  }
  return true;
}
```

5.9 Textúry

Na načítavanie textúr som použil hlavičkovú knižnicu stb_image.h, vďaka ktorej textúru načítame jedným volaním. Funkcia nám pritom vráti dáta textúry, a cez vstupno výstupné parametre aj veľkosti obrázka.

```
VulkanTexture::VulkanTexture(Type::String filename) {
    mBuffersInitialized = false;
  pDetails.filename = filename;
    mErrorState = 0;
    pDetails.pixelData = stbi_load(filename.c_str(),
        &pDetails.width, &pDetails.height, &pDetails.channels,
        STBI_rgb_alpha);
    pDetails.imageSize = pDetails.width * pDetails.height * 4;

    if (!pDetails.pixelData) {
        mErrorState++;
    }
}
```

Po získaní dát textúry ale musíme pripraviť Vulkan na použitie tejto textúry. Najprv musíme vytvoriť VkImage. Pri vytváraní použijeme znova staging buffer. Najprv naplníme dáta staging bufferu dátami textúry a potom tieto dáta pomocou staging bufferu premiestnime na pamäť GPU.

```
bool VulkanTexture::createTextureImage(VkPhysicalDevice
    physicalDevice, VkDevice device, VkCommandPool commandPool,
    VkQueue queue){
    mDevice = device;
    VkBuffer stagingBuffer;
    VkDeviceMemory stagingBufferMemory;
    if (!registerBuffer(device, physicalDevice, pDetails.imageSize,
        VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
        VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
    VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
        stagingBuffer, stagingBufferMemory)) {
        mErrorState++;
        return false;
    }

    void* data;
    vkMapMemory(device, stagingBufferMemory, 0, pDetails.imageSize,
    0, &data);
    memcpy(data, pDetails.pixelData, static_cast<size_t>(pDetails.
    imageSize));
    vkUnmapMemory(device, stagingBufferMemory);
    stbi_image_free(pDetails.pixelData);

    if (!createImage(device, physicalDevice,
        pDetails.width, pDetails.height,
        VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_TILING_OPTIMAL,
        VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT
    , VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
        pTexture, pTextureMemory)) {
        return false;
    }
    if (!transitImageLayout(device, queue, commandPool, pTexture,
    VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_LAYOUT_UNDEFINED,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL)) {
        return false;
    }
    copyBufferToImage(device, commandPool, queue, stagingBuffer,
    pTexture, static_cast<uint32_t>(pDetails.width), static_cast<
    uint32_t>(pDetails.height));
    if (!transitImageLayout(device, queue, commandPool, pTexture,
    VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,
    VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL)) {
        return false;
    }
    mBuffersInitialized = true;
    vkDestroyBuffer(device, stagingBuffer, nullptr);
    vkFreeMemory(device, stagingBufferMemory, nullptr);
    return true;
}
```

Po vytvorení VkImage musíme vytvoriť VkImageView pomocou pomocnej funkcie createImageView. Posledne, každá textúra musí mať vlastný sampler. Potrebujeme teda naplniť štruktúru VkSamplerCreateInfo dátami. Tu špeci-

fikujeme ako sa bude textúra chovať ak budeme ťahať súradnice z rozmedzia mimo textúrových koordinátov.

```
bool VulkanTexture::initializeSampler(VkPhysicalDevice physicalDevice, VkDevice device){
    VkPhysicalDeviceProperties properties{};
    vkGetPhysicalDeviceProperties(physicalDevice, &properties);

    VkSamplerCreateInfo samplerInfo{};
    samplerInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
    samplerInfo.magFilter = VK_FILTER_LINEAR;
    samplerInfo.minFilter = VK_FILTER_LINEAR;
    samplerInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    samplerInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    samplerInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    samplerInfo.anisotropyEnable = VK_TRUE;
    samplerInfo.maxAnisotropy = properties.limits.maxSamplerAnisotropy;
    samplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK;
    samplerInfo.unnormalizedCoordinates = VK_FALSE;
    samplerInfo.compareEnable = VK_FALSE;
    samplerInfo.compareOp = VK_COMPARE_OP_ALWAYS;
    samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;

    if (vkCreateSampler(device, &samplerInfo, nullptr, &pSampler) != VK_SUCCESS) {
        return false;
    }
    return true;
}
```

Posledne, musíme upraviť description set, ktoré nastavujeme v triede VulkanModel aby sme mohli sampler a textúru referencovať v shaderoch.

5.10 Pipeline

Pri inštancovaní štruktúr VkPipeline, musíme nastaviť niekoľko vlastností. Navyše pre každý model, ktorý používa iné rozloženie vertexu alebo iné shadere, alebo používa iné uniformy, musíme vytvoriť jeho vlastnú VkPipeline. Postupne som teda všetky parametre napĺňoval a vytváral v nasledujúcom poradí:

1. VkPipelineLayout, definuje sekvenciu descriptor setov, z ktorých má každý z nich svoj vlastný layout. Ja som však používal jednu inštanciu VkPipeline pre jeden model, vďaka čomu stačí na vytvorenie VkPipelineLayout definovať jeden descriptor set layout,

2. vertexový vstup a napojenie (vertex input and binding), definujeme pri zavedení vertexu, v mojej implementácií som vytvoril statické členské funkcie v triede VulkanModel, ktoré som v tomto kontexte vedel využiť.

3. zostavovanie vstupu (input assembly), kde definujeme pomocou Vulkan API enumov, koľko vertexov sa spája a vytvára geometriu, ja som zvolil triangle list, pretože si s týmto spôsobom vystačíme vo väčšine aplikácií,

4. rendrovacia zóna (render zone), kde som zadefinoval limity zón, do ktorých môže VkPipeline rendrovať a taktiež maximálnu hĺbku a minimálnu hĺbku pre depthbuffer,

5. rasterizér, popisuje culling a spôsob rendrovania geometrie, ktorý som parametrizoval cez štruktúru Pipeline::CreationInfo,

6. vzorkovanie (multisampling), kde som nastavil jeden vzorok na fragment,

7. depth a stencil, kde som zapol depth testing a nastavil spôsob akým chcem aby Vulkan porovnával vzdialenosť objektov,

8. miešanie farieb (color blending) v podstate potrebujeme len ak je zapnutý culling a v takom prípade defunujem operácie pre miešanie farby a alfa kanálu.

Po nastavení vyššie uvedeného som vytvoril VkPipelineLayout a VkPipeline.

```
1  CreationData creationData{};
2  setPipelineLayout(creationData.pipelineLayout, descriptorSetLayout);
3  if (vkCreatePipelineLayout(mLogicalDevice, &creationData.
       pipelineLayout, nullptr, &pPipelineLayout) != VK_SUCCESS) {
4    return false;
5  }
6  //specifyVertexInput(creationData.vertexInput);
7  VkVertexInputBindingDescription bindDescription = VulkanVertex::
       getBindingDescription();
8  std::vector<VkVertexInputAttributeDescription> attributeDescription
       = VulkanVertex::getAtributeDescription();
9  creationData.vertexInput.sType =
       VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
10 // data spacing
11 creationData.vertexInput.vertexBindingDescriptionCount = 1;
12 creationData.vertexInput.pVertexBindingDescriptions = &
       bindDescription;
13 // data types
14 creationData.vertexInput.vertexAttributeDescriptionCount =
       attributeDescription.size();
```

```
15 creationData.vertexInput.pVertexAttributeDescriptions =
       attributeDescription.data();
16
17 specifyInputAssembly(creationData.inputAssembly);
18 specifyRenderZone(creationData);
19 setupRasterizer(creationData.rasterizer);
20 setupMultisampling(creationData.multisampling);
21 setupDepthAndStencil(creationData.depthAndStencil);
22 setupColorBlending(creationData.colorBlend, creationData.
       colorBlendAttachment);
23 setDynamicState(creationData.dynamics);
24 setPipelineData(creationData, renderPass);
25 if (vkCreateGraphicsPipelines(mLogicalDevice, VK_NULL_HANDLE, 1, &
       creationData.pipeline, nullptr, &pPipeline) != VK_SUCCESS) {
26   return false;
27 }
28 mInitialized = true;
29 pShaderLink->destroyShaderModules();
30 return true;
```

5.11 Vytvorenie dll knižnice

Pri vytváraní dll knižnice som použil následujúce makro.

```
1 #ifdef LIB_API_EXPORT
2   #define LIB_API __declspec(dllexport)
3 #else
4   #define LIB_API __declspec(dllimport)
5 #endif
```

Knižnicu kompilujem so zadefinovaným symbolom LIB_API_EXPORT a ukážkové programy kompilujem so symbolom LIB_API_IMPORT[14]. Makro LIB_API definujem pri každej triede, ktorú chcem exportovať z knižnice, až na šablónové triedy. Tie sú celé z pravidla definované v hlavičkových súboroch a knižnica alebo program importujúci knižnicu so šablónovými triedami si už vie šablóny inštancovať sám.

```
1 struct LIB_API VulkanVertex{
2   math::Vec3f position;
3   math::Vec3f normal;
4   math::Vec2f texCoords;
5   math::Vec3f color;
6
7   static VkVertexInputBindingDescription getBindingDescription();
8
9   static std::vector<VkVertexInputAttributeDescription>
      getAtributeDescription();
10 };
```

[14]Kompilovať ukážky so symbolom LIB_API_IMPORT nieje podstatné. Podstatné je nezadefinovať pre ukážkové programy linkujúce knižnicu symbol LIB_API_EXPORT aby makro fungovalo správne. LIB_API_IMPORT som nastavil len aby nedošlo k nedorozumeniu a pre kvázi úplnosť.

Kapitola 6

Implementácia ukážok využitia knižnice

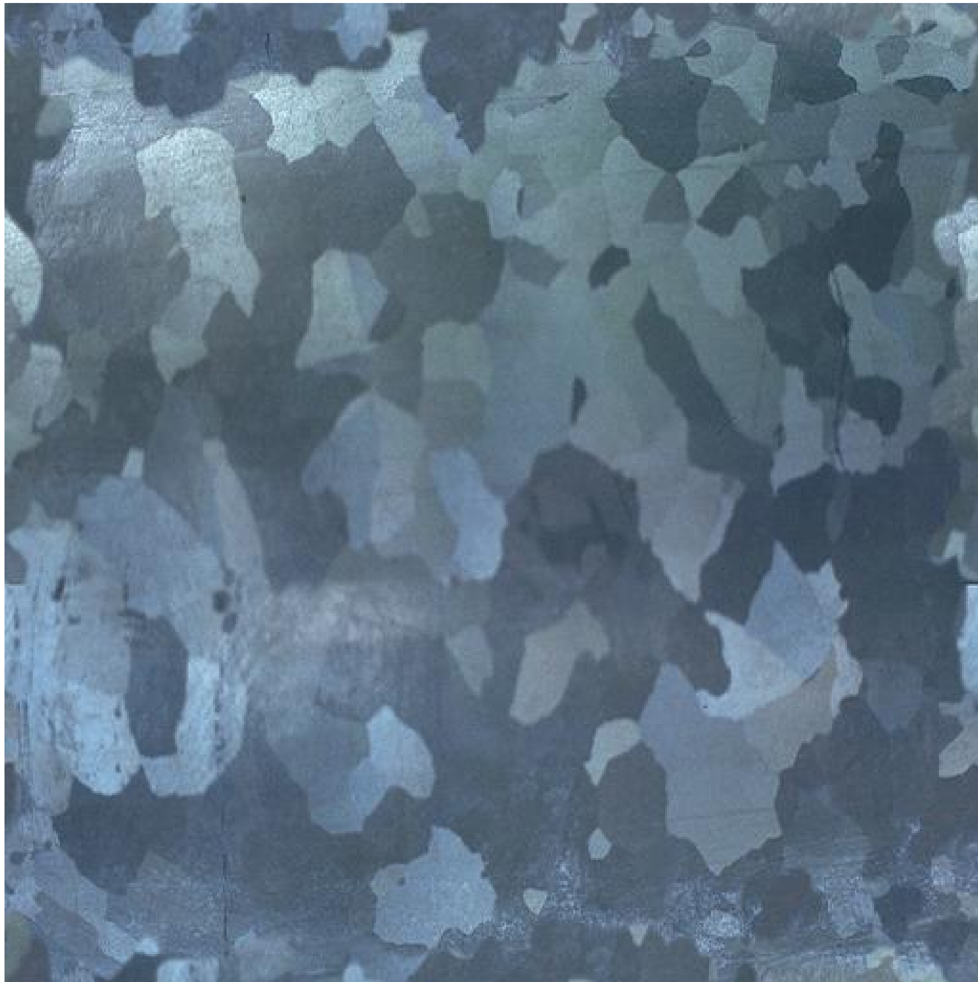
V rámci nasledujúcich ukážok som použil textúru na obrázku 6.1, stiahnutú zo stránky opengameart.org, ktorá je pod licenciou CC0. Z tejto textúry som vytvoril pomocou programu GIMP čiernobielu textúru, v ktorej pixely definujú množstvo odlesku.

6.1 Základná ukážka

V prvom programe použijeme čisto funkcionality, ktoré sme si zadefinovali v DLL knižnici. Najprv som inicializoval Vulkan v troch krokoch. Najprv inicializujeme Vulkan, potom nastavíme renderovacie štruktúry a potom štruktúry, ktoré použijeme v rendrovacom cykle.

```
1  static auto startTime = std::chrono::high_resolution_clock::now();
2  APP::Application vulkanHandler(800, 600, 2);
3  if (vulkanHandler.initializeVulkanData() != APP::Application::Error
       ::NONE) {
4    return -1;
5  }
6  if (vulkanHandler.initializeVulkanRenderData() != APP::Application::
       Error::NONE) {
7    return -2;
8  }
9  if (vulkanHandler.initializeLoopData() != APP::Application::Error::
       NONE) {
10   return -3;
11 }
```

Po nastavení základných štruktúr Vulkanu, vieme načítať model a vytvoriť preň pipelineu. Vďaka vytvorenej knižnici nám stačí vytvoriť a naplniť štruktúru Pipeline::CreationInfo dátami, ktoré máme dostupné. Vieme špecifikovať mená shaderov a vypĺňanie geometrie. Model som v tejto ukážke špecifikoval



Obr. 6.1: Textúra používaná v ukážkach[6]

vypísaním všetkých vertexov a ich indexov. Po načítaní modelu som k modelu pridal textúru a na koniec som vytvoril VkPipeline pre tento model.

```
...
if (!model.model.loadModel(vertices, indices)) {
  return -4;
}
if (app.linkTextureToModel(Type::String(TEXTURES_FOLDER_LOCATION_STR
    ) + "texture.jpg", &model.model) != APP::Application::Error::
    NONE) {
  return -5;
}
APP::Pipeline::CreationInfo pipelineCreationData{};
pipelineCreationData.shaderNames[Type::Uint32(APP::ShaderType::
    VERTEX)] = Type::String(SHADERS_FOLDER_LOCATION_STR) + "shader.
    vert";
```

```
10 pipelineCreationData.shaderNames[Type::Uint32(APP::ShaderType::
       FRAGMENT)] = Type::String(SHADERS_FOLDER_LOCATION_STR) + "shader
       .frag";
11 pipelineCreationData.mode = VK_POLYGON_MODE_FILL;
12 if (app.registerNewModel(&(model.model), pipelineCreationData, model
       .index) != APP::Application::Error::NONE) {
13   return -6;
14 }
```

Rendrovací cyklus som vďaka našej knižnici tiež zjednodušil, vďaka čomu nám stačí vypočítať časovú deltu pre animácie, nabindovať model a k nemu potrebnú inštanciu VkPipeline a špecifikovať dáta pre GeometryUBO. Trieda Application::CommandBufferRecorder je implementovaná tak, aby pri svojej deštrukcí ukončila VkRenderPass a nahrávanie príkazov na VkCommandBuffer, poslala príkazy potrebnému radu a nastavila odprezentovanie výsledkov. V tejto funkcií, ale môže nastať chyba pri volaní príkazov vkEndCommandBuffer a vkQueueSubmit, preto som umožnil uživateľovi skontrolovať túto chybu tým, že sám zavolá funkciu endRecording(. . .).

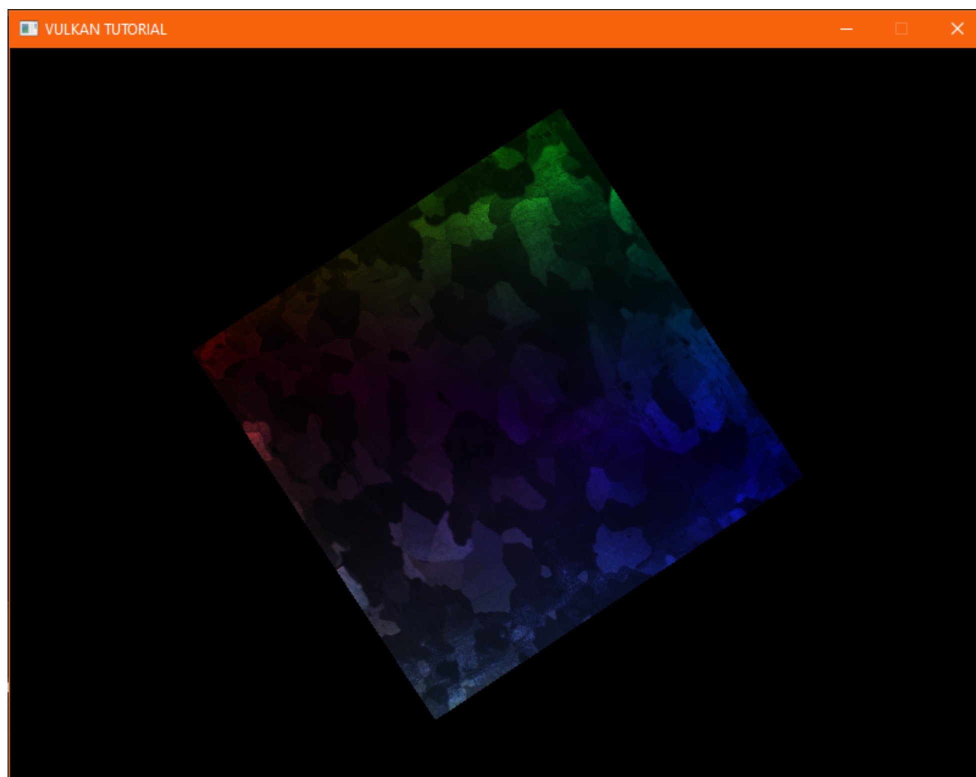
```
1 glfwPollEvents();
2 auto currentTime = std::chrono::high_resolution_clock::now();
3 float time = std::chrono::duration<float, std::chrono::seconds::
      period>(currentTime - startTime).count();
4 APP::Application::CommandBufferRecorder recorder = vulkanHandler.
      getCommandRecorder();
5 vulkanHandler.bindModelAndPipeline(recorder, model.index);
6 APP::GeometryUBO ubo{};
7 ubo.model = math::createRotationYMatrix(15.f * time);
8 ubo.view = createViewMatrix(math::Vec3f(0.f, -2.f, 0.f), math::Vec3f
      (0.f, 0.f, 0.f), math::zOne3f);
9 ubo.projection = math::createPerspektiveMatrix(45.f, 0.1f, 250.f,
      800.f, 600.f);
10 vulkanHandler.mapGeometryUBO(recorder, &(model.model), ubo);
11 if (recorder.endRecording() != APP::Application::Error::NONE){
12   return -10;
13 }
```

Vďaka našej knižnici je výsledný kód skrátený, utriedený do celkov a vo výsledku zaberá len okolo osemdesiat riadkov. Výstup programu môžme vidieť na obrázku 6.2. GeometryUBO ktorý sme nastavili v rendrovacom cykle spôsobuje, že štvorec rotuje okolo Y osy.

6.2 Používanie viacero textúr

V tejto ukážke som najväčšie zmeny spravil v shaderoch. Kedže mi stačí pre túto ukážku rendrovať jeden model vrátil som sa k implementácií loadModel z prvej ukážky kde som k modelu načítal o jednu textúru viac.

```
1 ...
```

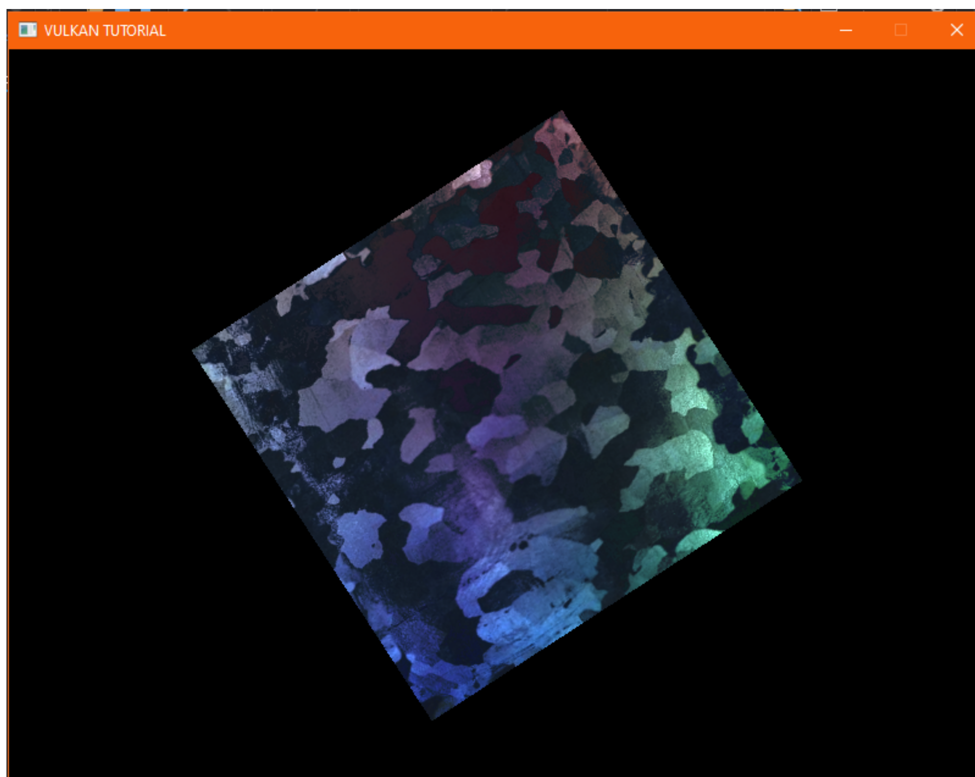


Obr. 6.2: Výstup prvej ukážky. Foto autor.

```
2 if (app.linkTextureToModel(Type::String(TEXTURES_FOLDER_LOCATION_STR
      ) + "texture.jpg", &model.model) != APP::Application::Error::
      NONE) {
3   return -5;
4 }
5 if (app.linkTextureToModel(Type::String(TEXTURES_FOLDER_LOCATION_STR
      ) + "specular.jpg", &model.model) != APP::Application::Error::
      NONE) {
6   return -5;
7 }
8 ...
```

V shadery som musel pridať ďalšiu premennú typu uniform sampler2D. Túto premennú som musel nastaviť na 2. lokáciu pomocou kľúčového slova binding. Funkcie clampColorVector a getAverage som implementoval na zjednodušenie zápisov. ClampColorVector používa funkciu clamp na všetky zložky vektora a funkcia getAverage vráti priemer zložiek vektora, inými slovami vráti jeho dĺžku.

```
1 ...
2 layout(binding = 1) uniform sampler2D diffuseSampler;
3 layout(binding = 2) uniform sampler2D specularSampler;
4 ...
```



Obr. 6.3: Výstup druhej ukážky. Foto autor.

```
5 void main() {
6     float value = getAverage(vec3(texture(specularSampler,
      fragTexCoord).rgb));
7     outColor = vec4(clampColorVector(value * fragColor + texture(
      diffuseSampler, fragTexCoord).rgb, 0.0, 1.0), 1.0);
8 }
```

Shader teda používa farbu definovanú pri definovaní vertexov modelu, ktorá sa interpoluje medzi vrcholmi, ktorú mieša s hodnotou určenou specular samplerom. Výsledok môžme vidieť na obrázku 6.3.

6.3 Phongovo nasvietenie

Pre posledný príklad som použil funkcionalitu ktorú som implementoval v knižnici, umožňujúcu definovať jeden uniform buffer objekt pre všetky modely. Tento uniform buffer objekt definuje svetlá v scéne, ich farbu, umiestnenie, počet bodových svetiel, jedno smerové a jedno reflektorové svetlo. Program sa teda líši najmä naplnením štruktúry SceneLightningUBO.

```
1 ...
```

```
APP::Application::CommandBufferRecorder recorder = vulkanHandler.
    getCommandRecorder();

APP::SceneLightningUBO sceneUBO{};
sceneUBO.pointLightCount = 1;
sceneUBO.pointLights[0].position = math::Vec3f(4 * sin(time), 4 *
    cos(time), 4.f);
sceneUBO.pointLights[0].color = math::Vec3f(1.f, 1.f, 1.f);
sceneUBO.pointLights[0].attenuation = 3.f;
sceneUBO.camera.position = cameraPosition;
vulkanHandler.mapSceneLightningUBO(recorder, sceneUBO);
vulkanHandler.bindModelAndPipeline(recorder, model.index);
...
```

Pre výpočty realizované vo fragment shadery som musel upraviť aj vertex shader. Budem napr. potrebovať poznať reálnu lokáciu fragmentu v priestore, po transformáciách. Tu nás ale zaujíma kde sa vertex nachádza v 3D priestore a nie v 2D rovine, čo znamená že nepotrebujem na vertex aplikovať maticu perspektívneho zobrazenia ba ani maticu kamery. Ďalej musíme správne vynásobiť normálu ako 3D vektor, na čo nám stačí z normálového trojrozmerného vektora spraviť štvorrozmerný vektor, ktorého štvrtý prvok je 0, transponovanou inverziou matice modelu. Získané hodnoty pošleme fragment shaderu.

```
...
layout(location = 1) out vec2 fragTexCoord;
layout(location = 2) out vec3 fragPos;
layout(location = 3) out vec3 translatedNormal;

void main() {
    fragPos          = (ubo.model * vec4(inPosition, 1.0)).xyz;
    gl_Position      = ubo.proj * ubo.view * vec4(fragPos, 1.0);
    translatedNormal = (transpose(inverse(ubo.model)) * vec4(normal,
     0.0)).xyz;
    ...
}
```

Vo fragment shadery som musel najprv prijať hodnoty z vertex shaderu. Ďalej som zadefinoval niekoľko štruktúr popisujúcich svetelné zdroje.

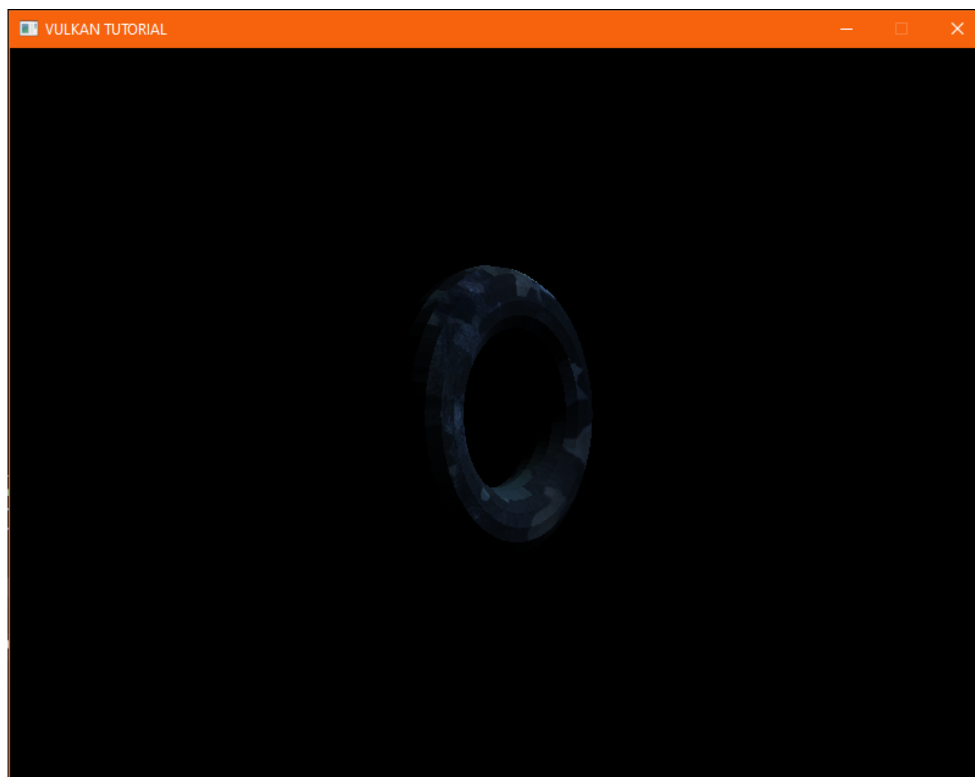
```
...
struct Camera {
  vec3  position;
};

struct DirectionalLight {
  vec3  direction;
  vec3  color;
};

struct SpotLight {
  vec3  position;
  vec3  direction;
  vec3  color;
  float attenuation;
```

```
16 };
17
18 struct PointLight {
19   vec3  position;
20   vec3  color;
21   float attenuation;
22 };
23
24 layout(binding = 1) uniform sampler2D diffuseSampler;
25 layout(binding = 2) uniform sampler2D specularSampler;
26 layout(binding = 3) uniform SceneLightningUBO {
27   Camera         camera;
28   DirectionalLight  directionalLight;
29   SpotLight      spotLight;
30   PointLight       pointLights[8];
31   uint         pointLightCount;
32   uint         flags;
33 } ubo;
34 ...
```

Posledné zmeny sa týkajú už priamo výpočtov Phongovho osvetlenia. Phongove osvetlenie sa skladá z výpočtu troch zložiek. Ambient, ktorá určuje osvetlenie modelu, pokiaľ v scéne nieje žiadne svetlo. Osvetľuje objekt nezávisle od dopadajúceho uhla a jedná sa o triviálnu simuláciu realistického osvetlenia, kde aj keď nesvietime na objekt nejakým svetelným zdrojom objekt stále v realite vidíme a vieme odhadnúť jeho farbu[15]. Diffuse zložka, osvetľuje plochu podľa toho ako je lúč odchýlený od normály pomocou skalárneho súčinu. Z toho vyplýva, že ak svetlo dopadá kolmo na povrch, povrch bude osvetlený najviac a ak lúč dopadá pod takmer pravým uhlom, plocha nebude osvetlená skoro vôbec. Posledná zložka specular, definuje odlesky. Tie sa dajú ľahko vypočítať ako skalárny súčin vektora odrazeného lúča a smerového vektora definujúceho pozíciu, z ktorej sa pozeráme do scény. Tento skalárny súčin umocníme podľa toho aký veľký odlesk chceme. Fragment shader teda vyzerá následovne.

```
1  ...
2  void main() {
3    vec3 ambient = vec3(0.0), diffuse = vec3(0.0), specular = vec3
       (0.0);
4    vec3 specularity = texture(specularSampler, fragTexCoord).rgb;
5    vec3 diffuseColor = texture(diffuseSampler, fragTexCoord).rgb;
6
7    ambient = 0.0001 * diffuseColor;
8    vec3 normNormal = normalize(normal);
9    for (uint i = 0; i < ubo.pointLightCount; i++){
10     vec3 lightDirection = normalize(ubo.pointLights[i].position -
       fragPos);
11     float diff = clamp(dot(normNormal, lightDirection), 0.0, 1.0);
12     diffuse += diff * ubo.pointLights[i].color;
```

[15] V realite objekty aj v absolútnej hmle osvetľujú objekty, i keď minimálne. Napr. v noci hviezdy a mesiac, alebo v uzavretej miestnosti, kde svetlo dopadá cez nejaké okno na steny, sa časť lúčov odrazí a dopadne na osvetľovaný objekt.



Obr. 6.4: Objekt osvetlený Phongovým modelom. Foto autor.

```
13
14      vec3 reflectionDirection = reflect(-lightDirection, normNormal);
15      vec3 directionOfView = normalize(ubo.camera.position - fragPos);
16      float spec = pow(clamp(dot(directionOfView, reflectionDirection)
        , 0.0, 1.0), 32);
17      specular += getAverage(specularity) * spec * ubo.pointLights[i].
        color;
18    }
19      outColor = vec4((ambient + diffuse + specular) * diffuseColor,
        1.0);
20  }
```

Objekt je vo výsledku osvetlený podľa toho z ktorej strany svieti svetlo a výsledok vyzerá ako na obrázku 6.4.

6.4 Použitie kvaterniónov

Poslednú ukážku som venoval kvaterniónom. V tomto príklade som musel pozmeniť napĺňanie hodnoty v štruktúre GeometryUBO. Tentokrát hodnoty nenapĺňame maticou, ale postupne po stĺpcoch definujeme vektor posunutia, zmeny veľkostí a kvaternión.

```
...
APP::GeometryUBO ubo{};
std::vector<math::Vec3f> axises{
  math::xOne3f, math::yOne3f, math::zOne3f
};
ubo.model.fillRow(0, math::Vec4f(0.5f, 0.5f, 0.5f, 0.f));
ubo.model.fillRow(1, math::Vec4f(0.f, 0.f, 0.f, 0.f));
ubo.model.fillRow(2, math::createRotation(math::Vec3f(90.f, 11.25f *
     time, 0.f), axises));
...
```

V shadery som musel definovať dve nové funkcie, jednu na násobenie kvaterniónu kvaterniónom a druhú na realizáciu rotácie. Vstupný vektor najprv zrotujem pomocou kvaterniónu, následne vynásobím vektorom veľkostí a následne pripočítam posun. Rotáciu aplikujem aj na normálu.

```
...
vec4 multiplyQuaternions(vec4 q1, vec4 q2){
    float dotP = -dot(q1.yzw, q2.yzw);
    vec3 crossP = cross(q1.yzw, q2.yzw);
    vec4 res = vec4(
        q1.x*q1.x + dotP,
        q1.x*q2.yzw + q2.x*q1.yzw + crossP
    );
    return res;
}

vec3 qRotation(vec4 q, vec4 vec){
    vec4 qInv = vec4(q.x, -q.yzw);
    vec4 res = multiplyQuaternions(q, vec);
    res = multiplyQuaternions(res, qInv);
    return res.yzw;
}

void main() {
    vec3 rotated        = qRotation(ubo.model[2], vec4(0.0,
    inPosition));
    fragPos             = (rotated * ubo.model[0].xyz);
    fragPos             += ubo.model[1].xyz;
    ...
    translatedNormal    = qRotation(ubo.model[2], vec4(0.0, normal))
    ;
    ...
```

Vo výsledku získam výstup rovnaký predošlému, ktorého výstup vidíme na obrázku 6.4.

Záver

Táto bakalárska práca mala niekoľko cieľov. Z hľadiska teórie som oboznámil čitateľa so základnými pojmami a problémami počítačovej grafiky. Počítačová grafika ale môže byť záludná a vie potrápiť aj skúseného programátora. Preto som zpracoval a vysvetlil postup pri rendrovaní, ktorým som sa dostal ku správnemu výsledku. Taktiež som čitateľa oboznámil s niektorými knižničnými funkciami Vulkan API. Behom programovania som zistil, že Vulkan API nám poskytuje omnoho väčšiu voľnosť ako OpenGL, predovšetkým ho hlavne vieme lepšie optimalizovať pre náš program. Niektoré súčasti sú však komplexnejšie a zložitejšie. Po spracovaní bakalárskej práce som získal na OpenGl a Vulkan nový názor. OpenGL stále nájde svoje uplatnenie v sférach počítačovej grafiky, kde sa nepotrebujeme príliš venovať optimalizáciám a rýchlosti programu.

V budúcnosti by som sa rád viac zameral už vytvoreniu konkrétnej aplikácie zo získaných poznatkov, rozšírením knižnice, ktorú som v tejto bakalárskej práci navrhol. Môžem sa v takom prípade venovať špecifickejším problémom, a rozsiahlejším implementáciám, postupne by som začal viac parametrizovať vytvorenie a nastavenie renderera, vytvoril robustnejšie registrovanie modelov a napokon implementoval viaceré pomocné štruktúry pre rýchlejšiu alokáciu prvkov. Z grafického prostredia by som napr. implementoval ako prvé nastavenia k multisamplingu, ktoré vyhladí ostré hrany, ktoré v ukážkach sú, implementoval samplery pre textúry prostredia a pre dynamické, animované, textúry. Mohlo by sa jednať napríklad hru alebo softvér na tvorbu hier, či editovanie 3D modelov.

Literatúra

[1] NVIDIA: NVIDIA GEFORCE. [online], [cit. 10.5.2021]. Dostupné z: https://www.nvidia.com/cs-cz/geforce/graphics-cards/30-series/rtx-3080/

[2] CHAOS CZECH: Corona. [cit. 10.5.2021]. Dostupné z: https://corona-renderer.com/

[3] OVERVOORDE, A.: Vulkan Tutorial - Introduction, [obrázok]. [online], [cit. 10.5.2021]. Dostupné z: https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction

[4] DE VRIES, J.: LearnOpenGL - Depth Testing, [obrázok]. [online], [cit. 10.5.2021]. Dostupné z: https://learnopengl.com/Advanced-OpenGL/Depth-testing

[5] DE VRIES, J.: LearnOpenGL - Textures, [obrázok]. [online], [cit. 10.5.2021]. Dostupné z: https://learnopengl.com/Getting-started/Textures

[6] BART: Golgotha Textures, [obrázok]. [online], [cit. 10.5.2021]. Dostupné z: https://opengameart.org/content/golgotha-textures

[7] OVERVOORDE, A.: Vulkan Tutorial. [online], [cit. 10.5.2021]. Dostupné z: https://vulkan-tutorial.com/Introduction

[8] DE VRIES, J.: *Learn OpenGL - graphics programming*. Kendall & Welling, prvni vydání, ISBN 987-90-9033256-7. Dostupné z: https://learnopengl.com/book/book_pdf.pdf

[9] DOMBEK, D. a. k.: Lineární algebra, 2019, skriptá k predmetu BI-LIN, [cit. 10.5.2021]. Dostupné z: https://kam.fit.cvut.cz/deploy/bi-lin//lin-text.pdf

[10] OLŠÁK, P.: *Úvod do algebry, zejména lineární*. Praha: Česká technika - nakladatelství ČVUT, druhé vydání, 2013, ISBN 978-80-01-05291-4.

[11] ŽÁRA, J.; BENEŠ, B.; SOCHOR, J.; aj.: *Moderní počítačová grafika*. Computer press, druhé vydání, 2005, ISBN 80-251-0454-0.

[12] GROUP, K. V. W.: Vulkan - Specification. [online], [cit. 10.5.2021]. Dostupné z: `https://www.khronos.org/registry/vulkan/specs/1.2-extensions/html/vkspec.html#preamble`

[13] SCRATCHAPIXEL: Scratchapixel 2.0. [online], [cit. 10.5.2021]. Dostupné z: `https://www.scratchapixel.com/index.php`

[14] GÜNAŞTI, G.: *Quaternions Algebra, Their Applications in Rotations and Beyond Quaternions*. Bakalárska práca, Linnaeus University, Faculty of Science and Engineering, School of Computer Science, Physics and Mathematics, 2012.

[15] DIEBEL, J.: Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, ročník 58, č. 15-16, 2006: s. 1–35.

[16] MCCLANAHAN, C.: History and evolution of gpu architecture. *Prieskumný príspevok*, ročník 9, 2010.

[17] HOFMANN, G.: Who invented ray tracing? *The Visual Computer*, ročník 6, 2005: s. 120–124, doi:10.1007/BF01911003.

[18] KHRONOS GROUP: [online], Mar 2010, [cit. 10.5.2021]. Dostupné z: `https://www.khronos.org/registry/OpenGL/specs/gl/glspec40.core.pdf`

[19] LINDHOLM, E.; NICKOLLS, J.; OBERMAN, S.; aj.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, ročník 28, č. 2, 2008: s. 39–55, doi:10.1109/MM.2008.31.

[20] KHRONOS GROUP: OpenGL WIKI. [online], [cit. 10.5.2021]. Dostupné z: `https://www.khronos.org/opengl/wiki_opengl/index.php?title=Main_Page&oldid=14430`

[21] HAINDL, M.; RICHTR, R.: Dynamic texture enlargement. In *Proceedings of the 29th Spring Conference on Computer Graphics*, 2013, s. 5–12.

[22] RICHTR, R.; HAINDL, M.: Dynamic texture editing. In *SCCG*, 2015, s. 133–140.

[23] RICHTR, R.: *Modelování dynamickỳch textur*. Dizertační práce, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2018.

[24] RICHTR, R.; HAINDL, M.: Dynamic Texture Similarity Criterion. In *2018 24th International Conference on Pattern Recognition (ICPR)*, IEEE, 2018, s. 904–909.

Dodatok A

Zoznam použitých skratiek

API Application Programming Interface

CPU Central Processing Unit

DLL Dynamic-link library

GLSL OpenGL Shading Language

GPU Graphics Processing Unit

HLSL High-Level Shading Language

RAII Resource Acquisition Is Initialization

SPIR Standard Portable Intermediate Representation

UBO Uniform Buffer Object

Dodatok **B**

Obsah priloženého CD

```
readme.txt ........................ stručný popis obsahu CD
└── vulkan-api
    ├── Dependencies .................. potrebné knižnice
    ├── thesis.zip ... zkomprimovaná zdrojová forma práce vo formáte LaTeX
    ├── Text .............................. text práce
    │   └── thesis.pdf ............. text práce vo formáte PDF
    └── Vulkan_Examples ................. implementácia
        ├── Models .............. modely objektov použitých v ukážkach
        ├── Multitexturing ....... implementácia ukážky použitia viac textúr
        ├── Output ............. adresár so spustiteľnou formou implementácií
        ├── Phong ............... implementácia ukážky Phongovho osvetlenia
        ├── QuaternionUse ....... implementácia ukážky použitia kvaterniónov
        ├── Shaders ............. shadere použité v ukážkach
        ├── SimpleModelRotation ......... implementácia 1. ukážky
        ├── Textures ............ textúry použité v ukážkach
        └── Vulkan .............. implementácia knižnice
```