FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání bakalářské práce

Název:	Real time rendering 3D grafiky za použití pokročilých texturálních modelů.
Student:	Zdeněk Nejedlý
Vedoucí:	Ing. Radek Richtr, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Počítačová grafika 2021
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

1) Proveďte rešerši dostupných řešení problematiky real-time renderingu a rozeberte jejich použití v nejznámějších herních enginech.
2) Formulujte požadavky pro 3D real-time renderer
3) Prozkoumejte možnost praktického využití a kompatibility pokročilých texturálních modelů BTF a DBTF v porovnání s používanými metodami.
4) Sestrojte návrh a implementujte prototyp rendereru využívající BTF.
5) Otestujte výslednou implementaci a porovnejte s renderery dostupných v různých herních enginech.

Elektronicky schválil/a Ing. Radek Richtr, Ph.D. dne 17. února 2024 v Praze.

Bakalářská práce

REAL TIME RENDERING 3D GRAFIKY ZA POUŽITÍ POKROČILÝCH TEXTURÁLNÍCH MODELŮ

Zdeněk Nejedlý

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Radek Richtr, Ph.D.
14. května 2024

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2024 Zdeněk Nejedlý. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Nejedlý Zdeněk. *Real time rendering 3D grafiky za použití pokročilých texturálních modelů.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Poděkování vii

Prohlášení viii

Abstrakt ix

Seznam zkratek x

I Teoretická část 2

1 Radiometrie 3
1.1 Radiantní energie (Radiant energy) . . . 3
1.2 Zářivý tok (Radiant flux) . . . 4
1.3 Hustota zářivého toku (Radiant flux density) . . . 4
1.4 Radiance . . . 5
1.5 Zářivost (Radiant intensity) . . . 7

2 Materiálový model 8
2.1 Obecný materiálový model . . . 8
2.2 Rodina zjednodušených modelů . . . 9
2.2.1 Bidirectional Surface Scattering Reflectance Distribution Function . . . 10
2.2.2 Bidirectional Texture Function . . . 11
2.2.3 Bidirectional Scattering Distribution Function . . . 12
2.2.4 Bidirectional Reflectance Distribution Function . . . 12

3 Renderování 14
3.1 Typy renderování . . . 14
3.2 Model . . . 15
3.3 Kamera . . . 15
3.3.1 Souřadnicový systém . . . 16
3.3.2 Pohledová transformace . . . 16
3.3.3 Projekční transformace . . . 16
3.4 Zdroje osvětlení . . . 17
3.5 Logická struktura scény . . . 18
3.6 Renderovací řetězec . . . 18
3.7 Osvětlovací modely . . . 20
3.7.1 Osvětlení v moderních herních enginech . . . 21

II Praktická část 22

4 Návrh real-time rendereru 23
4.1 Požadavky . . . 23

4.2 Návrh architektury 24
4.3 Diagram tříd 25

5 Integrace modelu BTF 26
5.1 Reprezentace dat 26
5.1.1 Rozložení UTIA BTF database 27
5.1.2 Vlastní zjednodušené rozložení 29
5.2 Optimalizace atlasu textur 30
5.3 Použití modelu DBTF 31

6 Implementace real-time rendereru 33
6.1 Použité technologie 33
6.2 Datové struktury závislé na renderovacím API 34
6.3 Datové struktury nezávislé na renderovacím API 40
6.4 Definice hlavních Renderer tříd 45

7 Implementace aplikace BTF Visualizer 47
7.1 Funkcionalita vrstvy BTFVisualizerLayer 47
7.2 Demonstrační scény 48
7.3 Implementace BTF shaderu 49
7.3.1 Vertex shader 49
7.3.2 Fragment shader 49
7.4 Ovládání aplikace 57

8 Výsledky a porovnání 58
8.1 Porovnání s dostupnými real-time renderery 59

A Ukázky z aplikace BTF Visualizer 63

Obsah příloh 74

Seznam obrázků

1.1 Iradiance (vlevo) a radiozita (vpravo) [1] 4
1.2 Prostorový úhel 6
1.3 Radiance s příchozím zářivým tokem 6
1.4 Inverse square law 7

2.1 Model GRF 8
2.2 Taxonomie vybraných modelů materiálu 10
2.3 Model BSSRDF 11

3.1 Pohledový souřadnicový prostor 16
3.2 Pohledový frustum 17
3.3 Graf scény 18

4.1 Diagram tříd rendereru 25

5.1 Jednotkové sférické souřadnice 27
5.2 Rozložení vzorkovacích bodů na polokouli pro materiály z databáze UTIA BTF . 28
5.3 Atlas textur pro materiály z databáze UTIA BTF 29
5.4 Atlas textur pro vlastní zjednodušené BTF materiály 30
5.5 Problém měření BTF materiálů 31
5.6 Sekvence atlasů textur pro DBTF 32

7.1 Ukázka transformace pohledového vektoru podle normály 50
7.2 Jednoduché vzorkování vlastního BTF materiálu 53
7.3 Ukázka počítání vah bilineárního vzorkování pro čtyři body 54
7.4 Nákres hledání bodů bilineární interpolace pro UTIA BTF materiály 56

8.1 BTF materiál travnatého povrchu (Demo 1) 58
8.2 Materiál látky z databáze UTIA BTF (Demo 1) 58
8.3 Materiály látky a travnatého povrchu (Demo 3) 59
8.4 Porovnání renderů materiálu látky mezi vybranými renderery 60
8.5 Graf odpovědí kvantitativní části testování 61

A.1 Ukázka materiálu látky na čtvercové ploše při kolmém pohledu shora. Pořízena bilineárním vzorkováním bez filmového mapování tónů 64
A.2 Ukázka materiálu látky na čtvercové ploše při kolmém pohledu shora. Pořízena bilineárním vzorkováním s využitím filmového mapování tónů 65
A.3 Ukázka materiálu travnatého povrchu na čtvercové ploše pozorované pod úhlem. Pořízena bilineárním vzorkováním s využitím filmového mapování tónů. Demonstruje plynulý přechod do černé u horizontu 66
A.4 Ukázka materiálu travnatého povrchu bez bilineárního vzorkování na komplexním objektu. Pořízeno s využitím filmového mapování tónů 67
A.5 Ukázka materiálu travnatého povrchu bez bilineárního vzorkování z opačné strany komplexního objektu. Pořízeno s využitím filmového mapování tónů 68

A.6 Ukázka materiálu látky bez bilineárního vzorkování z opačné strany komplexního objektu. Pořízeno s využitím filmového mapování tónů . . . 68
A.7 Ukázka materiálu látky s využitím bilineárního vzorkování na modelu torusu. Pořízeno s využitím filmového mapování tónů . . . 69
A.8 Ukázka materiálu travnatého povrch bez využití filmového mapování tónů. Je vidět, že materiál je takto výrazně tmavší. Pořízen pomocí jednoduchého vzorkování 69
A.9 Ukázka materiálu látky s využitím bilineárního vzorkování na čtvercové ploše pod úhlem. Pořízeno bez využití filmového mapování tónů . . . 70
A.10 Ukázka materiálu látky s využitím bilineárního vzorkování na čtvercové ploše pod úhlem. Pořízeno bez využití filmového mapování tónů . . . 71

Seznam tabulek

5.1 Počet bodů na hladinách polokoule v UTIA BTF database . . . 27

8.1 Tabulka nativní dostupnosti materiálových modelů v real-time rendering aplikacích 60

Seznam výpisů kódu

6.1 Metoda pro instancování VertexBufferu . . . 34
6.2 Rozhraní třídy VertexBuffer . . . 35
6.3 Rozhraní třídy ElementBuffer . . . 35
6.4 Rozhraní třídy Shader . . . 36
6.5 Rozhraní třídy VertexArray . . . 37
6.6 Ukázka základní třídy Texture . . . 38
6.7 Rozhraní specializovaných textur . . . 39
6.8 Rozhraní třídy Framebuffer a její pomocná struktura . . . 40
6.9 Třída pro reprezentaci světla . . . 41
6.10 Třída pro reprezentaci materiálu . . . 42
6.11 Datová třída Transform . . . 43
6.12 Ukázka základní třídy Camera . . . 44
6.13 Ukázka třídy Mesh . . . 44
6.14 Rozhraní třídy RendererAPI . . . 45
6.15 Rozhraní třídy Renderer . . . 46

7.1 Definice aplikace BTF Visualizer . . . 47
7.2 Vertex shader pro BTF materiály . . . 49
7.3 Struktura pro reprezentaci BTF materiálu . . . 51
7.4 Fragment shader pro BTF materiály . . . 51
7.5 Jednoduché vzorkování vlastního BTF materiálu . . . 52

7.6 Ukázka funkce pro výpočet indexu nejbližší textury v atlasu UTIA BTF 55

Především bych chtěl poděkovat mému vedoucímu Ing. Radku Richtrovi, Ph.D., za skvělé vedení této práce. Dále pak jmenovitě děkuji mojí krásné přítelkyni Bc. Markétě Pleškové za trpělivost a péči a mému kamarádovi Tomáši Weissovi za vášnivé debaty o grafických konceptech v této práci. Na závěr děkuji své rodině, zejména pak rodičům, za nehynoucí podporu při psaní.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 14. května 2024

Abstrakt

Tato bakalářská práce zkoumá použití pokročilých texturálních materiálových modelů pro renderování v reálném čase. V teoretické části popisuje různé materiálové modely a člení je podle zjednodušujících předpokladů. Dále se věnuje pojmům z pole real-time renderingu. V praktické části předkládá návrh real-time rendereru a debatuje nad integrací materiálového modelu BTF. Tento návrh je částečně implementován jako prototyp a pomocí něj je vytvořena aplikace pro renderování BTF materiálů. V závěru práce je tento prototyp porovnán s konvenčními postupy.

Klíčová slova real-time renderování, BTF, obousměrná texturová funkce, renderer, materiálový model, renderovací systém

Abstract

This bachelor thesis explores the use of advanced textural material models for real-time rendering. The theoretical part describes different material models and classifies them according to simplifying assumptions. It also discusses concepts from the field of real-time rendering. In the practical part, it presents a design of a real-time renderer and discusses possible integration of a BTF material model. This proposal is partially implemented as a prototype and used to create an application for rendering BTF materials. The paper concludes by comparing this prototype with conventional approaches.

Keywords real-time rendering, BTF, bidirectional texture function, renderer, material model, rendering system

Seznam zkratek

API Application Programming Interface
BRDF Bidirectional Reflectance Distribution Function
BSDF Bidirectional Scattering Distribution Function
BSSRDF Bidirectional Surface Scattering Reflectance Distribution Function
BTF Bidirectional Texture Function
EBO Element Buffer Object
GRF General Reflectance Function
GUI Graphical User Interface
IBRDF Isotropic Bidirectional Reflectance Distribution Function
MSAA Multisampling Anti Aliasing
PBR Physically Based Rendering
VAO Vertex Array Object
VBO Vertex Buffer Object

Úvod

Tato bakalářská práce se věnuje renderování počítačové grafiky v reálném čase (real-time renderování). Jedná se o problematiku, která se řeší především v herních enginech a herním odvětví obecně. V moderní době roste nátlak na realitě věrohodnější grafické vizuály čím dál víc. Výkon hardwaru každým rokem roste, což vývojářům stále dává možnosti posouvat své výpočetní modely, a tím vytvářet realističtější grafiku běžící v reálném čase. Tato práce zkoumá použití komplexnějšího materiálového modelu pro popis povrchů, který realističtěji popisuje reálné materiály.

Cílem této práce je zjistit, zda-li dnešní hardware dokáže pokročilejší materiálový model renderovat dostatečně rychle a jestli je výsledkem realističtější a lépe vypadající povrch objektů. Konkrétně zkoumá použití modelu Bidirectinal Texture Function (BTF) a jeho dynamické verzi (DBTF). Navrhuje postup pro renderování materiálu pomocí modelu BTF a možnou optimalizaci. Tento postup je následně implementován v aplikaci BTF Visualizer. Pro aplikaci byl také navrhnut a implementován prototyp renderovacího systém (renderer).

V teoretické části práce je popsán koncept renderování se zaměřením specificky na materiálové modely. V praktické části je navržen renderovací systém, který je následně implementován. S využitím tohoto systému je vytvořena základní verze aplikace pro vykreslování BTF materiálu.

První kapitola je zaměřena na popis a kvantifikaci světla. Ve druhé kapitole jsou představeny významné materiálové modely, které popisují interakci světla s povrchy objektů. Závěrem teoretické rešerše je třetí kapitola, která představuje real-time renderování a základní koncepty v tomto odvětví. Ve čtvrté kapitole je sestrojen návrh rendereru pro real-time renderování. Pátá kapitola obsahuje detaily integrace modelu BTF do renderovacího systému a nabízí možnou optimalizaci. V kapitolách šest a sedm je popsána implementace nejdříve rendereru a následně aplikace pro zobrazování BTF materiálů. Poslední, osmá kapitola, prezentuje výsledky vykreslených BTF materiálů na různých površích a porovnává je s výsledky z běžně dostupných real-time rendering aplikací.

Výstupem práce je aplikace BTF Visualizer, která demonstruje renderování části BTF materiálů v reálném čase za pomoci navržených metod. Aplikace je založena na renderovacím systému, který je také dostupný jako výstup práce.

Část I

Teoretická část

Kapitola 1

Radiometrie

Pro modelování a výpočet osvětlení ve virtuálních scénách je nutné porozumět chování světla a umět kvantifikovat jeho vlastnosti. Radiometrie je věda zabývající se měřením a popisu světla z jakékoliv části elektromagnetického spektra. Světlo jako takové je chápáno jako elektromagnetická radiace, ke kterému radiometrie nabízí aparát, jak světlo objektivně kvantifikovat a popisovat.

Lidské oko je schopné vidět pouze zlomek z celého elektromagnetického světla – viditelné světlo. To se na spektru nachází v rozmezí vlnové délky 380 až 770 nanometrů. Věda, která se zabývá měřením viditelného světla v jednotkách, které jsou váženy subjektivně dle citlivosti lidského oka, se nazývá fotometrie. [1]

Překlady jednotlivých veličin jsou přejaty z knihy Moderní Počítačová Grafika [2].

1.1 Radiantní energie (Radiant energy)

Základní částicí světla je foton. Jeho energie na vlnové délce λ je

$$e_\lambda = \frac{h \cdot c}{\lambda},$$

kde $h \approx 6,63 \cdot 10^{-34}\ J \cdot s$ je Planckova konstanta a $c = 299\ 792\ 458\ m \cdot s^{-1}$ je rychlost světla ve vakuu. Tato energie je udávána v joulech (J). [2]

▶ **Definice 1.1** (Radiantní energie Q)**.** *Množství energie, které světlo přenáší. Vypočítá se pomocí integrálu*

$$Q = \int_0^\infty n_\lambda e_\lambda \ d\lambda,$$

kde n_λ je počet fotonů o vlnové délce λ a e_λ je energie jednoho fotonu vlnové délky λ. Základní jednotkou je joule (J).

Elektromagnetická radiace přenáší energii skrz prostor. Může být chápána jako částice i jako vlnění, záleží na měření. Když je světlo absorbováno fyzickým objektem, tak je jeho energie přeměněna v jinou podobu.

Pro mnohé radiometrické veličiny se také definuje jejich spektrální varianta. Ta vyjadřuje hodnotu dané veličiny pro konkrétní vlnovou délku. Spektrální radiantní energie popisuje účinek záření pro jednu konkrétní vlnovou délku. [1]

▶ **Definice 1.2** (Spektrální radiantní energie Q_λ)**.** *Množství radiantní energie na jednotku vlnové délky pro vlnovou délku λ*

$$Q_\lambda = \frac{dQ}{d\lambda}.$$

Základní jednotkou je joule na nanometr ($J \cdot nm^{-1}$).

1.2 Zářivý tok (Radiant flux)

▶ **Definice 1.3** (Zářivý tok Φ)**.** *Změna radiantní energie za jednotku času*

$$\Phi = \frac{dQ}{dt}.$$

Základní jednotkou je joule za vteřinu, neboli watt (W). [1]

Může být také označena jako zářivý výkon – vychází z definice výkonu (množství práce za jednotku času).

Místy se tok světla znázorňuje pomocí geometrických paprsků světla. Je možno si je představit jako nekonečně úzké úsečky nakreslené napříč prostorem, které indikují směr toku zářivé energie.

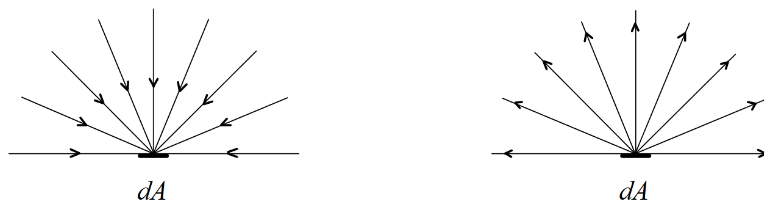
▶ **Definice 1.4** (Spektrální zářivý tok Φ_λ)**.** *Množství zářivého toku na jednotku vlnové délky pro vlnovou délku λ*

$$\Phi_\lambda = \frac{d\Phi}{d\lambda}.$$

Základní jednotkou je watt na nanometr ($W \cdot nm^{-1}$).

1.3 Hustota zářivého toku (Radiant flux density)

▶ **Definice 1.5** (Hustota zářivého toku)**.** *Zářivý tok na jednotku plochy v bodě na povrchu (povrch může být jak reálný, tak imaginární). Dále se dělí dle směru toku vůči bodu na povrchu na* ***radiozitu*** *a* ***iradianci*** *(obrázek 1.1). [1]*



Obrázek 1.1 Iradiance (vlevo) a radiozita (vpravo) [1]

▶ **Definice 1.6** (Radiozita M)**.** *Pokud má zářivý tok směr z bodu od povrchu pryč, například vyzářením, nebo odražením, nazýváme hustotu zářivého toku radiozitou*

$$M(\boldsymbol{x}) = \frac{d\Phi}{dA},$$

kde Φ je zářivý tok odcházející z bodu $\boldsymbol{x}$ a dA je diferenciální plocha obklopující tento bod. Tok může opouštět bod jakýmkoliv směrem. Základní jednotkou je watt na metr čtvereční ($W \cdot m^{-2}$).

▶ **Definice 1.7** (Iradiance E)**.** *Pokud má zářivý tok směr do bodu na povrchu, nazýváme hustotu zářivého toku iradiancí*

$$E(\boldsymbol{x}) = \frac{d\Phi}{dA},$$

kde Φ je zářivý tok přicházející do bodu $\boldsymbol{x}$ a dA je diferenciální plocha obklopující tento bod. Tok může přicházet do bodu jakýmkoliv směrem. Základní jednotkou je watt na metr čtvereční ($W \cdot m^{-2}$).

Možnost povrchu být jak reálný, tak imaginární znamená, že hustotu zářivého toku je možné měřit kdekoliv v tří-dimenzionálním prostoru.

Alternativně lze iradianci definovat [3] pomocí radiance (definice 1.12), konkrétně pomocí integrálu přes hemisféru Ω:

$$E(\mathbf{x}) = \int_{\Omega} L_i(\mathbf{x}, \omega) \cos\theta \; d\omega,$$

kde L_i je příchozí radiance do bodu $\mathbf{x}$ ze směru ω, ω je prostorový úhel, θ je úhel mezi normálou v bodě $\mathbf{x}$ a směrem ω a Ω je hemisféra se středem v bodě $\mathbf{x}$. Analogicky se pak definuje radiozita, s jediným rozdílem – radiance je z bodu $\mathbf{x}$ odchozí

$$M(\mathbf{x}) = \int_{\Omega} L_o(\mathbf{x}, \omega) \cos\theta \; d\omega.$$

▶ **Definice 1.8** (Spektrální hustota zářivého toku)**.** *Hustota zářivého toku na jednotku vlnové délky pro vlnovou délku λ. Základní jednotkou je watt na metr čtvereční na nanometr ($W \cdot m^{-2} \cdot nm^{-1}$).*

▶ **Definice 1.9** (Spektrální radiozita M_λ)**.** *Radiozita na jednotku vlnové délky pro vlnovou délku λ*

$$M_\lambda(\boldsymbol{x}) = \frac{dM(\boldsymbol{x})}{d\lambda}.$$

Základní jednotkou je watt na metr čtvereční na nanometr ($W \cdot m^{-2} \cdot nm^{-1}$).

▶ **Definice 1.10** (Spektrální iradiance E_λ)**.** *Iradiance na jednotku vlnové délky pro vlnovou délku λ*

$$E_\lambda(\boldsymbol{x}) = \frac{dE(\boldsymbol{x})}{d\lambda}.$$

Základní jednotkou je watt na metr čtvereční na nanometr ($W \cdot m^{-2} \cdot nm^{-1}$).

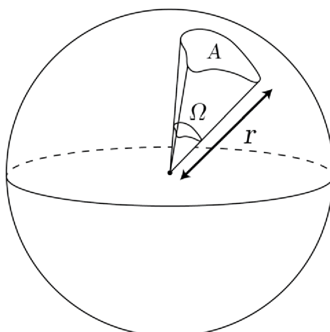
1.4 Radiance

Pro definici radiance je potřeba nejdříve zavést prostorový úhel. O radianci se dá přemýšlet jako o hustotě zářivého toku v bodě na povrchu pro právě jeden paprsek světla. Tento paprsek světla lze chápat jako nekonečně úzký (elementární) kužel, tedy jako prostorový úhel. Na rozdíl od hustotě zářivého toku není rozdíl mezi tím, jestli paprsek jde směrem do bodu, nebo od něj.

▶ **Definice 1.11** (Prostorový úhel ω)**.** *Prostorový úhel objektu je dán plochou segmentu koule, která má střed v bodě, ze kterého objekt pozorujeme. Plocha tohoto segmentu odpovídá ploše, kterou by měl pozorovaný objekt promítnutý na povrch jednotkové koule (obrázek 1.2)*

$$\omega = \frac{A}{r^2}.$$

Základní jednotkou je steradián (sr). Z anglického solid angle. [4]



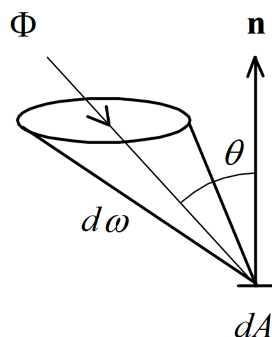
Obrázek 1.2 Vizualizace prostorového úhlu. A je plocha promítnutého objektu na kouli s poloměrem r. Prostorového úhel Ω je pak roven ploše A promítnuté na jednotkovou kouli [4]

Se znalostí prostorového úhlu se již dá definovat radiance. Důležité je ještě zmínit, že v potaz se musí brát i úhel paprsku od povrchu, na který dopadá.

▶ **Definice 1.12** (Radiance L).

$$L(\boldsymbol{x}, \omega) = \frac{d^2\Phi}{dA(d\omega \cdot \cos\theta)},$$

kde Φ je zářivý tok, dA je diferenciální plocha kolem bodu, $d\omega$ je diferenciální prostorový úhel elementárního kuželu a θ je úhel paprsku od normály povrchu (obrázek 1.3). Základní jednotkou je watt na metr čtvereční na steradián ($W \cdot m^{-2} \cdot sr^{-1}$). [1]



Obrázek 1.3 Radiance s příchozím zářivým tokem. Zářivý tok Φ je zde vyobrazen jako kužel s prostorovým úhlem $d\omega$. Vektor n je normálovým vektorem povrchu dA a mezi ním a prostorovým úhlem je úhel θ [1]

▶ **Definice 1.13** (Spektrální radiance L_λ). *Radiance na jednotku vlnové délky pro vlnovou délku λ*

$$L_\lambda(\boldsymbol{x}, \omega) = \frac{d^3\Phi}{dA(d\omega \cos\theta)d\lambda}.$$

Základní jednotkou je watt na metr čtvereční na steradián na nanometr ($W \cdot m^{-2} \cdot sr^{-1} \cdot nm^{-1}$).

1.5 Zářivost (Radiant intensity)

Máme-li nekonečně malý bodový zdroj světla (point light), který vyzařuje zářivý tok do každého směru, pak množství zářivého toku vyzářeného do daného směru můžeme reprezentovat paprskem (elementárním kuželem).

▶ **Definice 1.14** (Zářivost I)**.**

$$I = \frac{d\Phi}{d\omega},$$

kde $d\omega$ značí diferenciální prostorový úhel odpovídající danému směru. Základní jednotkou je watt na steradián ($W \cdot sr^{-1}$). [1]

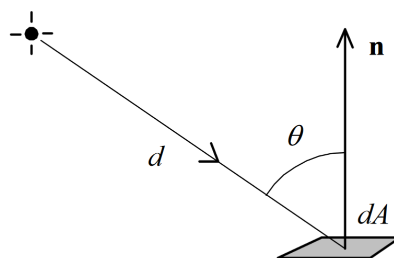
Pokud se podíváme na definici irradiance 1.7 a na definici prostorového úhlu 1.11, dostaneme:

$$E = \frac{d\Phi}{dA} = \frac{d\Phi}{r^2 d\omega} = \frac{I}{r^2}.$$

V těchto rovnicích r značí vzdálenost povrchu koule od jejího středu, ve kterém je umístěn bodový zdroj světla. Obecněji, zářivý tok dorazí na plochu dA pod úhlem θ. Z toho plyne *inverse square law* pro bodové zdroje světla (obrázek 1.4):

$$E = \frac{I \cos\theta}{d^2},$$

kde I je intenzita zdroje v daném směru, θ je úhel příchozího paprsku od normály povrchu a d je vzdálenost zdroje od povrchu dA.



Obrázek 1.4 Znázornění inverse square law [1]

▶ **Definice 1.15** (Spektrální zářivost I_λ)**.** *Zářivost na jednotku vlnové délky pro vlnovou délku λ*

$$I_\lambda = \frac{dI}{d\lambda}.$$

Základní jednotkou je watt na steradián na nanometr ($W \cdot sr^{-1} \cdot nm^{-1}$).

Kapitola 2

Materiálový model

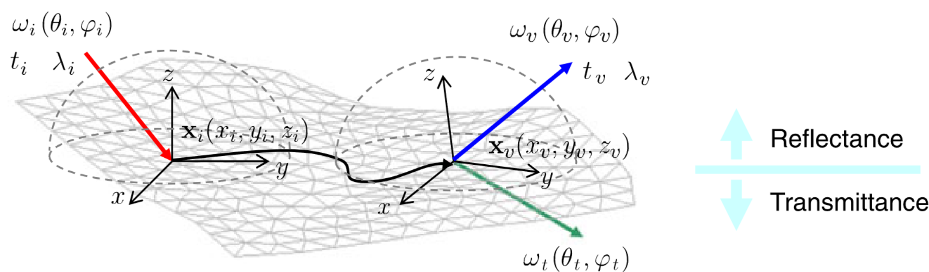
Pro popis interakce světla s povrchy objektů se zavádí materiálové modely. Různé modely se mohou používat pro různé povrchy a dokonce mohou být měřeny z reálných materiálů. Čím komplexnější materiálový model, tím realističtější výsledek, ovšem za ceny zvýšené náročnosti měření nebo používání modelu. Některé modely jsou čistě teoretické a není možno je v praxi nijak změřit ani namodelovat. Všechny v tomto textu zmíněné materiálové modely jsou odvozeny z obecného modelu, který slouží jako komplexní základ, který se postupně zjednodušuje.

2.1 Obecný materiálový model

Přesný popis interakcí materiálu se světlem je možné popsat komplexní 16-dimenzionální funkcí zvanou General Reflectance Function (GRF):

$$GRF(\lambda_i, x_i, y_i, z_i, t_i, \theta_i, \varphi_i, \lambda_v, x_v, y_v, z_v, t_v, \theta_v, \varphi_v, \theta_t, \varphi_t),$$

kde GRF charakterizuje interakce materiálu se světlem – λ_i popisuje vlnovou délku příchozího světla a (x_i, y_i, z_i) jeho pozici na povrchu, kam světlo dopadlo v čase t_i. Úhel dopadu tohoto světla je $\omega_i = [\theta_i, \varphi_i]$. Odražené světlo je pozorováno v bodu (x_v, y_v, z_v) v čase t_v pod úhlem $\omega_v = [\theta_v, \varphi_v]$ a vlnovou délkou λ_v. Úhel $\omega_t = [\theta_t, \varphi_t]$ značí úhel propustnosti světla. Značením $\omega = [\theta, \varphi]$ jsou myšleny jednotkové sférické souřadnice, kde θ je úhel elevace a φ je úhel azimutu (obrázek 2.1) [5][6].



Obrázek 2.1 Model GRF [5]

Funkce GRF je příliš komplexní pro měření nebo modelování. Praktičtější přístup je zavést zjednodušení ve formě různých předpokladů, které efektivně snižují dimenzi obecné funkce za ceny ztráty fyzikální přesnosti. Dle zvolených zjednodušení se odvíjí členové rodiny modelu GRF, jejichž ukázka se nachází na obrázku 2.2.

Mezi hlavní předpoklady dle Haindla a Filipa [5] patří:

P(1) Světlo se šíří nekonečnou rychlostí: $t_i = t_v$. Tedy t_v můžeme vypustit;

P(2) Interakce materiálu se světlem nezávisí na čase: $t_v = t_i = konst$. Tedy t_v a t_i můžeme vypustit;

P(3) Interakce světla nemění jeho vlnovou délku: $\lambda_i = \lambda_v$. Tedy λ_v můžeme vypustit;

P(4) Radiance je konstantní podél světelných paprsků. Tedy z_i a z_v můžeme vypustit;

P(5) Materiál nepropouští světlo. Tedy můžeme vypustit ω_t;

P(6) Světlo přichází a odráží se ze stejného bodu: $x_i = x_v$, $y_i = y_v$. Tedy x_v a y_v můžeme vypustit;

P(7) Světlo se nemůže rozptylovat pod povrchem materiálu (žádný *subsurface scattering*);

P(8) Povrch materiálu nemůže stínit sebe sama;

P(9) Povrch materiálu nemůže zakrývat sebe sama;

P(10) Zanedbáme odrazy vni materiálu;

P(11) Příchozí světlo může být buď odraženo, nebo absorbováno;

P(12) Prohození příchozího úhlu a úhlu odrazu nemění výslednou hodnotu (Helmholtzův princip reciprocity);

P(13) Úhel příchozího světla se nemění: $\omega_i = konst.$;

P(14) Úhel pozorovaného světla se nemění: $\omega_v = konst.$;

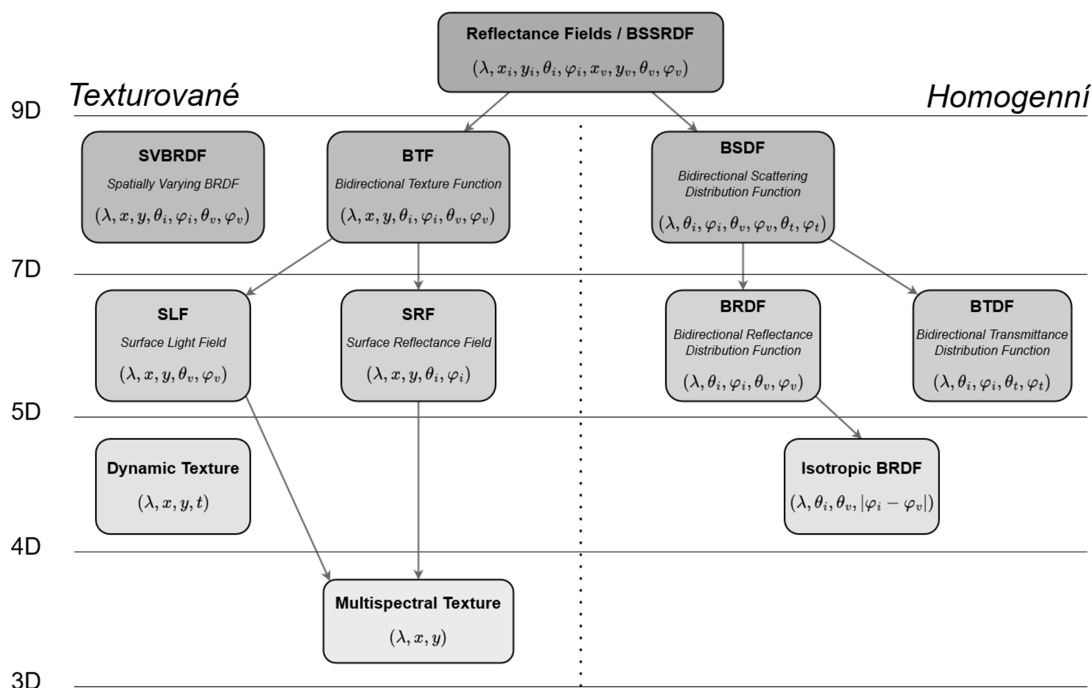
P(15) Nezáleží na pozici v prostoru. Tedy můžeme vypustit x_i, y_i a x_v, y_v;

P(16) Světlo se neodráží od povrchu. Tedy můžeme vypustit ω_v;

P(17) Odraz světla záleží na rozdílu azimutů (materiál je izotropní).

2.2 Rodina zjednodušených modelů

Na základě předpokladů zjednodušení obecné GRF jsou definovány různé modely (obrázek 2.2), které jsou její aproximací. Tyto modely se rozdělují na texturované a homogenní (netexturované). Homogenní modely se vyznačují tím, že mimo jiné používají zjednodušení P(15), které model zbavuje závislosti na pozici v prostoru. Lze je chápat jako povrch, který je pozorován z velké vzdálenosti.



Obrázek 2.2 Taxonomie vybraných modelů materiálu odvozených z devíti-dimenzionálního modelu Bidirectional Surface Scattering Reflectance Distribution Function (BSSRDF). Tento model ještě není rozlišován na homogenní a texturované modely, ovšem modely v hierarchii pod ním již takto kategorizované jsou a v obrázku jsou odděleny středovou přerušovanou úsečkou. U každého modelu je uveden jeho název zkratkou a níže rozepsán. Uvedeny jsou také parametry jednotlivých modelů

2.2.1 Bidirectional Surface Scattering Reflectance Distribution Function

Model Bidirectional Surface Scattering Reflectance Distribution Function (BSSRDF) je založen na těchto zjednodušujících předpokladech:

- P(1);
- P(2);
- P(3);
- P(4);
- P(5).

Z čehož plyne, že, na rozdíl od GRF, BSSRDF nepracuje s časem ($t_i = t_v = \varnothing$), změnou vlnové délky ($\lambda_i = \lambda_v = \lambda$), proměnnou radiancí podél světelných paprsků ($z_i = z_v = \varnothing$) a propustností ($\omega_t = \varnothing$).

Model BSSRDF závisí na devíti proměnných

$$BSSRDF(\lambda, x_i, y_i, \theta_i, \varphi_i, x_v, y_v, \theta_v, \varphi_v).$$

Tento model pracuje s pohybem světla pod povrchem materiálu, tedy s možným jiným bodem dopadu a odrazu (obrázek 2.3). Díky této vlastnosti model lépe popisuje strukturově komplexnější materiály než některé jednodušší modely (např. BRDF). [7]

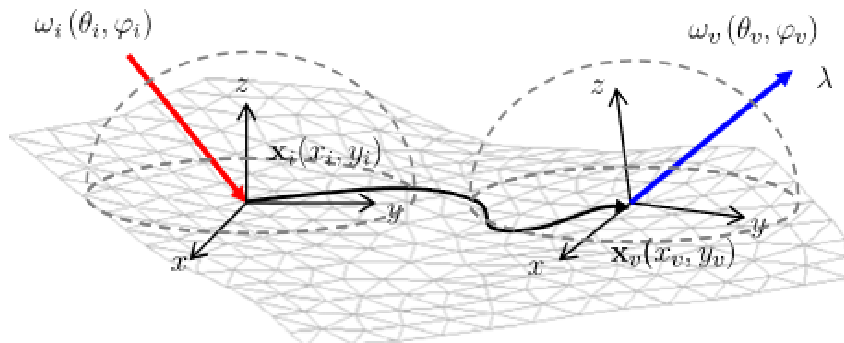
Pro výpočet odchozí radiance dL_0 ve směru ω_0 z bodu $\mathbf{x}_0$ se využívá přímé úměry mezi odchozí radiancí a příchozím zářivým tokem

$$dL_0(\mathbf{x}_0, \omega_0) = S \cdot d\Phi_i.$$

Příchozí zářivý tok můžeme dále rozepsat takto:

$$dL_0(\mathbf{x}_0, \omega_0) = S \cdot d\Phi_i = S \cdot L_i(\mathbf{x}_i, \omega_i) \cdot \cos\theta_i \; d\omega_i \cdot dA.$$

Člen S odpovídá hodnotě BSSRDF pro daný povrch. BSSRDF tedy udává poměr mezi příchozí a odchozí radiancí. Kvůli jeho vysoké dimenzionalitě není jednoduchý na měření pro reálné materiály a není tedy příliš praktický pro použití. Je ovšem nejkomplexnějším modelem, který bylo doposud možné změřit. [3]



Obrázek 2.3 Model BSSRDF [7]

2.2.2 Bidirectional Texture Function

Model Bidirectional Texture Function (BTF) je založen na těchto předpokladech:

- P(1);
- P(2);
- P(3);
- P(4);
- P(5);
- P(6).

Na rozdíl od obecného GRF, BTF uvažuje, že bod dopadu a odrazu světla se neliší ($x_i = x_r = x$ a $y_i = y_v = y$). Stejně jako BSSRDF nepracuje s časem ($t_i = t_v = \varnothing$), změnou vlnové délky ($\lambda_i = \lambda_v = \lambda$), proměnnou radiancí podél světelných paprsků ($z_i = z_v = \varnothing$) a propustností ($\omega_t = \varnothing$). [5]

Model BTF závisí na sedmi proměnných

$$BTF(\lambda, x, y, \theta_i, \varphi_i, \theta_v, \varphi_v).$$

Fakt, že BTF uvažuje stejný bod dopadu a odrazu, implikuje, že světlo se nebude pohybovat vni povrchu. Obdobně jako BSSRDF popisuje vztah mezi příchozí a odchozí radiancí, ovšem díky zanedbání pohybu světla pod povrchem je možno ji přímo měřit. [3]

2.2.3 Bidirectional Scattering Distribution Function

Model Bidirectional Scattering Distribution Function (BSDF) je postaven na těchto předpokladech:

- P(1);
- P(2);
- P(3);
- P(4);
- P(6);
- P(7);
- P(8);
- P(9);
- P(10);
- P(11);
- P(12);
- P(15).

BSDF tedy uvažuje, že bod dopadu a odrazu světla se neliší. Zároveň nezáleží na pozici v prostoru a nepočítá s proměnnou radiancí podél světelných paprsků ($x_i = y_i = z_i = \varnothing$ a $x_v = y_v = z_v = \varnothing$). Zanedbává možnost pohybu světla pod povrchem materiálu. Světlo se také nemůže rozptylovat pod povrchem materiálu a povrch nemůže stínit, ani zakrývat, sebe sama. Jakékoliv odrazy uvnitř materiálu jsou zanedbány. Příchozí světlo může být buď odraženo, nebo propuštěno. Prohození příchozího úhlu a úhlu odrazu nemá efekt na výslednou hodnotu. Nepracuje s časem ($t_i = t_v = \varnothing$) a změnou vlnové délky ($\lambda_i = \lambda_v = \lambda$).

Model BSDF závisí na sedmi proměnných

$$BSDF(\lambda, \theta_i, \varphi_i, \theta_v, \varphi_v, \theta_t, \varphi_t).$$

Nezávislost na pozici v prostoru z BSDF dělá homogenní (netexturovaný) model. Je chápán jako sjednocení modelů Bidirectional Transmittance Distribution Function (BTDF) a Bidirectional Reflectance Distribution Function (BRDF). BTDF popisuje jak světlo prochází skrz průhledné nebo částečně průhledné povrchy a je definována jako $BTDF(\lambda, \theta_i, \varphi_i, \theta_t, \varphi_t)$. BRDF popisuje jak se světlo odráží od povrchu a je definována pomocí $BRDF(\lambda, \theta_i, \varphi_i, \theta_v, \varphi_v)$. Tedy kde BTDF závisí na $\omega_t = [\theta_t, \varphi_t]$ – úhlu propustnosti, BRDF závisí na $\omega_v = [\theta_v, \varphi_v]$ – úhlu odrazu (úhlu pohledu). [5]

2.2.4 Bidirectional Reflectance Distribution Function

Model Bidirectional Reflectance Distribution Function (BRDF) pracuje s těmito předpoklady:

- P(1);
- P(2);
- P(3);
- P(4);
- P(5);
- P(6);
- P(7);
- P(8);
- P(9);
- P(10);
- P(11);
- P(12);
- P(15).

Stejně jako BSDF uvažuje, že bod dopadu a odrazu světla se neliší, nezáleží na pozici v prostoru a nepočítá s proměnnou radiancí podél světelných paprsků ($x_i = y_i = z_i = \varnothing$ a $x_v = y_v = z_v = \varnothing$). Zanedbává možnost pohybu a rozptylu světla pod povrchem materiálu a povrch nemůže stínit ani zakrývat sebe sama. Jakékoliv odrazy uvnitř materiálu jsou zanedbány.

Příchozí světlo může být buď odraženo, nebo absorbováno. Prohození příchozího úhlu a úhlu odrazu nemá efekt na výslednou hodnotu. Nepracuje s časem ($t_i = t_v = \varnothing$) a změnou vlnové délky ($\lambda_i = \lambda_v = \lambda$). Jediným rozdílem od BSDF je zanedbání propouštění světla ($\theta_t = \varphi_t = \varnothing$).

Model BRDF závisí na pěti proměnných

$$BRDF(\lambda, \theta_i, \varphi_i, \theta_v, \varphi_v).$$

Pokud zanedbáme závislost na vlnové délce, tak je BRDF čtyř-dimenzionální funkce závislá na úhlu příchozího světla a úhlu pohledu. Pokud naopak počítáme s konstantními úhly pozorování a příchozího světla, tak model $BRDF(\lambda)$ se nazývá Lambertovský BRDF.

Speciální případ BRDF je takzvaná izotropická BRDF (IBRDF). Tento model je založen na stejných předpokladech jako BRDF, s výjimkou přídavku předpokladu P(17). Ten říká, že výsledek BRDF se nezmění, pokud oba azimutální úhly jsou stejně rotované. IBRDF [5] je pak dáno následovně:

$$IBRDF(\lambda, \theta_i, |\varphi_i - \varphi_v|, \theta_v).$$

Kapitola 3
Renderování

Procesu vytváření obrazu z virtuální scény, ať už dvou-dimenzionální či tří-dimenzionální, se říká renderování. Počeštěný termín renderování vychází z anglického *rendering* a dá se přeložit jako zobrazování či vykreslování. V této práci se pro tento pojem bude využívat jak počeštěný termín renderování, tak české varianty zobrazování a vykreslování. Tato kapitola čerpá primárně z knih Moderní Počítačová Grafika [2] a Real-Time Rendering [8].

Ve virtuální scéně se mohou nacházet různé objekty. V zásadě můžeme tyto objekty kategorizovat jako:

- zobrazované objekty – modely, pozadí scény;
- nezobrazované objekty – kamery, zdroje osvětlení, silová pole, emitory;
- objekty definující logickou strukturu scény – kolekce, instance, hierarchie rodičů.

Scéna definuje obsah, ovšem pro renderování je nutné dodat přídavné informace ohledně zpracování tohoto obsahu. Například jak pracovat se světlem napříč scénou, jaký materiálový model použít a nebo v jakém formátu má být výsledek tohoto procesu. Dohromady tyto informace udávají, co s čím má počítač dělat a jak výsledná data interpretovat. Výsledná data tohoto procesu se nazývají render, který má typicky formu obrázku nebo videa.

3.1 Typy renderování

Cílem renderování může být co nejvíce fyzicky přesná reprezentace vizuální stránky reálného světa (fotorealismus). Tomuto typu se říká fotorealistické renderování (*photorealistic rendering*). K získání fotorealistického renderu se používají výpočetní modely založené na fyzikálním porozumění reality. Takovéto modely spadají do kategorie fyzikálně založeného renderingu (z anglického *Physically Based Rendering*) a je jím například materiálový model obousměrné odrazové distribuční funkce (BRDF) detailněji popsaný v kapitole 2. Jedná se o určitou aproximaci reálného povrchu a jeho interakcí se světlem.

Aproximace jsou hlavním nástrojem pro další typ renderingu – real-time rendering, neboli rendering v reálném čase. Pro dosažení rychlých výpočtů, tak aby výsledný obraz působil plynule a živě, je nutné zavádět určitá zjednodušení, která významně urychlí výpočet za cenu ztráty věrnosti reality a kvality. Tohoto hojně využívají herní enginy – nástroje pro tvorbu počítačových her, ve kterých plynulost zážitku je jednou z klíčových vlastností. Fotorealistickým real-time renderingem se dále zabývá tato práce.

Protějškem fotorealistického renderování je nefotorealisitcké renderování, často označováno za stylizované. Pro zobrazení objektů využívá expresivních vizuálních stylů, které často neodpovídají realitě. Jako příklad je možno uvést Toy Story: Příběh Hraček – animovaný film z roku 1995 od studia Pixar, který byl prvním celovečerním filmem vygenerovaným počítačem.

3.2 Model

Model říkáme množině informací, které dohromady umožňují vykreslit těleso. Povrch tohoto tělesa je definován jeho geometrií. Jedná se o síť (*mesh*) trojúhelníků, případně různých n-úhelníků, které svou plochou určují plochu tělesa. Tyto n-úhelníky jsou v paměti počítače reprezentovány pomocí vrcholů (*vertices*) a hran (*edges*). Trojúhelníky se využívají nejhojněji, jelikož se jedná o nejjednodušší n-úhelníky, kterými lze definovat povrch. Jeden trojúhelník tvoří 3 vrcholy a 3 hrany.

Samotný povrch neurčuje vzhled modelu. Barvu, lesklost, texturu a další vlastnosti definuje materiál. Ten popisuje interakce příchozího světla s povrchem a určuje tak vzhled. Světlo se od povrchu může odrazit, nebo materiál světlo propustí pod povrch. Tyto jevy se jmenují odrazivost, respektive propustnost (z anglického *reflectance*, respektive *transmittance*). Může se také stát, že materiál vyzařuje světlo ze svého povrchu (*emittance*). Některé materiálové modely rozlišují, zda-li je materiál kovový, či nikoliv (*metalness*). Materiál popisuje různé vlastnosti které se odvíjejí od zvoleného modelu (rozebrané v kapitole 2). Pro robustnější materiálové modely je nutno poskytnout více dat a komplexnost výpočtu značně roste.

Jako nástroj pro reprezentaci vzhledu různých materiálů slouží textury. Textura je popisem vlastností povrchu a je důležitá pro vnímání jeho struktury, barvy a kvality. Její prvek se nazývá texel (zkráceno z anglického *texture element*). Textura je vzorek, který může být buď pravidelný, nebo nepravidelný. Popisuje nejrůznější vlastnosti materiálu a povrchu – od barvy, odrazu světla a průhlednosti až po změny normálového vektoru. Textury mohou být definovány s různými rozměry, od nejjednodušší jednorozměrné přes, dvourozměrné a trojrozměrné, až po čtyřrozměrné. Jednorozměrné mohou být použity pro definici opakujících se podélných vzorů. Dvourozměrné textury jsou mapovány na povrch tělesa a mohou tak přímo popisovat jeho vlastnosti. Trojrozměrné definují hodnoty v prostoru a říká se jim objemové. Čtyřrozměrné mohou být použity pro animaci trojrozměrných textur. V praxi je běžné, že je na povrch použito více textur různými způsoby. Takové technice se říká *multitexturing*. Pro použití textury je nutné určit, kam se na objekt přiloží. Tomu se říká mapování textury (*texture mapping*) a je možné zde použít různé techniky filtrací (například průměrování sousedních texelů, průměrování hodnot z různých textur, . . .).

Pokud je model součástí nějaké simulace, pak obsahuje simulační data. Může se jednat o jednoduchou simulaci pádu, nebo o pokročilejší simulaci tuhého tělesa (*rigid-body simulation*). Tyto simulace potřebují doplňující informace o modelu jako například jeho váhu, rychlost nebo zrychlení.

3.3 Kamera

Kamera ve scéně reprezentuje virtuálního pozorovatele scény. Byť u kamery typicky definujeme pozici a orientaci v prostoru, nejedná se o žádné těleso viditelně přítomné ve scéně. S kamerou jsou spojeny dvě základní transformace, které převádějí modely ve scéně mezi různými souřadnicovými systémy. Všechny modely, mají nějakou pozici a orientaci ve světovém souřadném systému (*world space*). Úkolem kamery je definovat transformace, které převedou tyto modely do kanonického pohledového objemu [8] (*canonical view volume*).

3.3.1 Souřadnicový systém

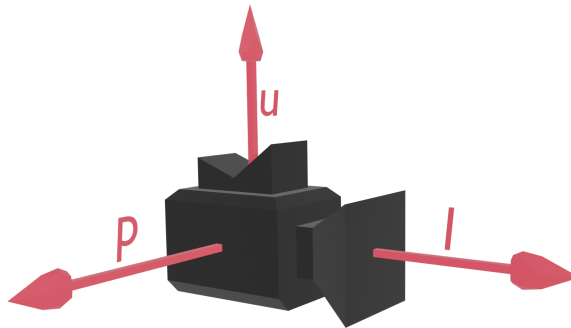
K reprezentaci bodů a vektorů hodnotami x, y a z ve tří-dimenzionálním prostoru je nutné určit, vůči jaké referenci jsou tato čísla vztažena. Jako tato reference funguje počátek prostoru a tři lineárně nezávislé vektory – báze souřadnicového systému. Vlastnost lineární nezávislosti vektorů znamená, že žádný vektor z těchto tří se nedá vyjádřit jako součet libovolně přeškálovaných těchto vektorů. Tyto bázové vektory se často nazývají jako **osy** (*axis*) [9].

Tedy počátek a báze o třech vektorech dohromady definují souřadnicový systém pro tří-dimenzionální prostor. Obecně pro n-dimenzionální prostor je třeba n lineárně nezávislých vektorů [10]. Dříve zmíněné proměnné x, y a z pak tvoří **souřadnice** vzhledem k souřadnicovému prostoru a obvykle se udávají jako uspořádaná trojice uspořádaná (x, y, z).

Aby mohly být definovány různé souřadnicové systémy, je nutné definovat kanonický souřadný systém, vůči kterému můžou být ostatní systémy uváděny. Tento systém bude mít počátek v $(0, 0, 0)$ s bázovými vektory $(1, 0, 0)$, $(0, 1, 0)$ a $(0, 0, 1)$ a nazývá se světový souřadnicový systém (*world space*). [11]

3.3.2 Pohledová transformace

Pohledová transformace, z anglického *view transformation*, přemístí všechny modely do pohledového souřadnicového systému (*view space*). To je souřadný systém, který má počátek na pozici kamery a osy orientované podle kamery (obrázek 3.1). U kamery se často pracuje se třemi jednotkovými vektory, které udávají její orientaci (tyto vektory se berou jako osy pohledového souřadného systému). Prvním je směr pozorování – $\mathbf{l}$, také známý jako optická osa kamery. Druhý, $\mathbf{u}$, reprezentuje natočení kolem optické osy $\mathbf{l}$ a uvažuje se kolmý na optickou osu. Posledním je vektor $\mathbf{p}$, který je kolmý na jak $\mathbf{l}$, tak $\mathbf{u}$. [2]



Obrázek 3.1 Pohledový souřadnicový prostor. Jednotkové vektory l, u a p tvoří bázi, a zároveň je každá dvojice vektorů na sebe kolmá. Počátkem tohoto prostoru je pozice kamery ve světovém souřadném systému

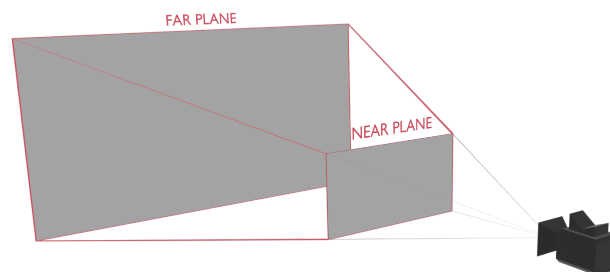
3.3.3 Projekční transformace

Cílem projekční transformace, z anglického *projection transformation*, je promítnout modely (3D objekty) do 2D roviny, která se nazývá průmětna (*viewing plane*, překlad z [2]). Tato průmětna je kolmá na směr, kterým kamera snímá (obrázek 3.2), kde *near plane* je zároveň i průmětnou, což obecně neplatí). Oblast prostoru ve scéně, která se může objevit na obrazovce, se nazývá pohledový frustum (*viewing frustum*) nebo pohledový objem (*viewing volume*). Pouze modely v tomto prostoru budou podstupovat následující transformaci a ostatní jsou vypuštěny. Pohledový frustum je vyhrazen šesti rovinami - levou, pravou, horní, spodní, přední a zadní.

Nejdůležitější z nich jsou přední a zadní (*near plane* a *far plane*), jelikož ty udávají, v jaké vzdálenosti budou blízké a daleké modely vypuštěny. Projekční transformace pak tento obecný pohledový frustum přetransformuje do krychle $(-1, 1)^3$, které se říká kanonický pohledový objem.

Tvar pohledového frustumu určuje typ promítání. Může být definován libovolně, ale v praxi se používají 2 nejčastější typy:

- **rovnoběžné promítání** – definuje pohledový frustum jako kvádr, projekční transformace tedy pouze škáluje a posunuje;
- **perspektivní promítání** – definuje pohledový frustum jako komolý jehlan (obrázek 3.2), projekční transformace musí po škálování a posunu ještě dělit [9].



Obrázek 3.2 Znázornění pohledového frustumu kamery mezi *near plane* a *far plane*. V tomto případě se *near plane* rovná *viewing plane*, obecně to tak nemusí být. Frustum zde má tvar komolého jehlanu

3.4 Zdroje osvětlení

Zdrojem osvětlení je obecně jakýkoliv objekt, který vyzařuje (emituje) světlo. Nejčastěji se používají následující světelné zdroje:

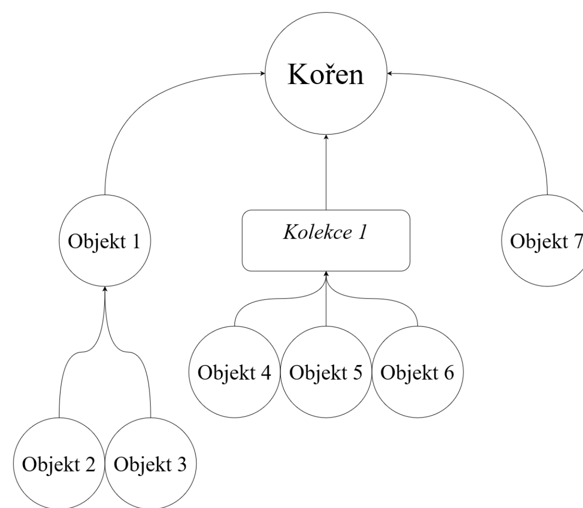
- **bodový světelný zdroj** (*point light*) – bod, který vyzařuje světlo rovnoměrně do všech směrů. Nasvícení bodovým zdrojem není realistické, jelikož v reálném světě takový zdroj neexistuje (pro realističtější výsledky je dobré místo bodu uvažovat kouli s nenulovým poloměrem);
- **směrový světelný zdroj** (*directional light*) – je chápán jako bodový zdroj v nekonečné vzdálenosti. Reprezentuje se jako směr, ze kterého světlo přichází. Aproximuje se jím světlo přicházející z velmi vzdálených objektů (jako například světlo přicházející ze Slunce);
- **plošný světelný zdroj** (*area light*) – rovinný (dvou-dimenzionální) povrch konečné plochy, který vyzařuje paprsky do předního poloprostoru všemi směry;
- **reflektor** (*spot light*) – je určen polohou a směrem. Z této polohy daným směrem vyzařuje světlo kuželovitým tvarem. Vyzařuje nejvíce světla ve směru osy tohoto kuželu a kolmo na tento směr klesá intenzita záření exponenciálně.

3.5 Logická struktura scény

Pro snadnější práci s komplexní scénou je vhodné zadefinovat jakousi logickou strukturu scény. Objekty se mohou sdružovat dohromady a tvořit kolekce, nebo mohou být mezi sebou uspořádány do vztahu rodič–potomek.

Ideálním stavem je takzvaný graf scény (obrázek 3.3). Graf scény je n-ární strom, neboli graf z teorie grafů, kde pro každý uzel existuje právě jeden předchůdce. Jedinou výjimkou je kořen stromu, ze kterého jsou všechny následovníci definováni.

Důležitou vlastností objektů v grafu je dědičnost. Změny v rodiči se propagují i mezi potomky (například přemístí-li se rodičovský objekt, pak budou přemístěni i všichni jeho potomci). Může být nastavena implicitně, ovšem může být také pro určité podstromy vypnuta.



Obrázek 3.3 Graf scény. Na vrcholu je kořen, do kterého jsou napojeni potomci. Objekt 1 je rodič objektů 2 a 3. Objekty 4, 5 a 6 jsou členy kolekce 1

3.6 Renderovací řetězec

Renderovací řetězec (neboli *rendering pipeline*) je sekvence operací, která z dat modelů a scény vykresluje obraz. Je možné ji rozčlenit do tří fází – aplikační, geometrická a rasterizační [8]. Aplikační část jako jediná běží na procesoru, kdežto geometrická a rasterizační fáze běží na grafické kartě. V každé fázi je popsáno, které části může uživatel ovlivnit. Nejtypičtějším zásahem do renderovacího řetězce je v podobě shaderů, což jsou instrukce psané uživatelem spouštěné na grafické kartě. Existuje více druhů shaderů, ovšem v tomto textu se vyskytují pouze vertex a fragment shader.

Aplikační fáze

V aplikační fázi uživatel definuje scénu, jednotlivé objekty v ní a jejich vlastnosti. Všechna data potřebná k vykreslování jsou na konci této fáze předána ve formě renderovacích primitiv

do geometrické fáze. Tato renderovací primitiva jsou body (*vertices*, počeštěně vertexy), které dohromady tvoří trojúhelníky. Tato fáze je celá v rukou uživatele.

Geometrická fáze

V geometrické fázi se pracuje s vertexy a obecně geometrií objektů. Tato fáze se dále dělí na modelovou, pohledovou a projekční transformaci, ořezávání a mapování na obrazovku. Zmíněné transformace převádí vertexy (a tedy celý objekt) mezi jednotlivými souřadnými soustavami (také se nazývají prostory).

Objekt se ze začátku nachází v modelovém souřadném systému. To je souřadný systém, v jehož počátku se nachází daný objekt/model. Z tohoto prostoru je modelovou transformací převeden do světového prostoru, ve kterém jsou vůči sobě rozmístěny všechny objekty. Následně je převeden do pohledového prostoru, který již byl zmíněn v sekci Pohledová transformace 3.3.2. Z tohoto prostoru je převeden pomocí projekční transformace popsané v sekci Projekční transformace 3.3.3.

Po této transformaci jsou vertexy, které leží mimo kanonický pohledový objem, oříznuty a nejsou dále zpracovávány v rasterizační fázi. Posledním krokem je namapovat vertexy na souřadnice na obrazovce – souřadnice x a y vertexů jsou postupně transformovány do obrazových souřadnic (*screen coordinates*) a spolu s původní souřadnicí z tvoří souřadnice v okně (*window coordinates*).

Aplikaci modelové, pohledové a projekční uživatel musí zajistit sám ve vertex shaderu. Ten definuje, co se bude dít během geometrické fáze (v rámci toho co může uživatel ovlivnit – takzvaná *programmable pipeline*). Mezi jednotlivými transformacemi může uživatel provádět libovolné výpočty, typicky vztažené ke konkrétnímu vertexu. Vertex shader se spouští pro každý vertex v modelu. Vypočtené hodnoty je možno předat (přímo nebo interpolované) do fragment shaderu, který je součástí programovatelné části rasterizační fáze.

Rasterizační fáze

Cílem rasterizační fáze je určit barvu pixelů (zkrácený výraz pro *picture element*), které pokrývají přetransformovaný objekt na obrazovce. Tomuto procesu se říká rasterizace a je to v podstatě proces přetváření geometrických dat do pixelů na obrazovce. Z vertexů jsou utvořeny plošky ve formě trojúhelníků. Plocha těchto trojúhelníků je převáděna na fragmenty, což jsou všechny pixely na obrazovce, které jsou zakryty těmito trojúhelníky.

Pro jednotlivé fragmenty probíhá výpočet jejich barvy. Jeho podobu definuje uživatel ve fragment shaderu. Zde se počítá barva na základě zvoleného osvětlovacího modelu. Do výpočtu vstupují parametry jako světla ze scény, materiál objektu, pozice fragmentu, normála fragmentu, různé textury a potenciálně mnoho dalších paramterů specifických pro konkrétně zvolený osvětlovací model. Tyto parametry uživatel může definovat v aplikační části, nebo je předat z vertex shaderu během geometrické fáze.

Fragmenty jsou následně podrobeny hloubkovému testu. Ten má za úkol vyřešit překryvy fragmentů na základě jejich souřadnice z. Podle té je zvolen nejbližší fragment, který je nakonec zobrazen na obrazovku, a ostatní, které jsou v prostoru dále od kamery, jsou zahozeny. Hloubkový test může být z části ovlivněn uživatelem, který může zvolit porovnávací funkci.

Výsledné pixely jsou postupně ukládány do *frame bufferu*. Aby uživatel neviděl, jak se na obrazovce postupně rastrují obrazce, je vhodné využít konceptu takzvaného *double bufferingu*. Jeden buffer s kompletně vykresleným snímkem je vždy na obrazovce, zatímco se do druhého postupně zapisují pixely. Po dokončení vykreslování se tyto buffery prohodí a celý proces se opakuje.

3.7 Osvětlovací modely

Osvětlovací model určuje postup výpočtu světla (radiance) odcházejícího z bodu na povrchu objektu směrem k pozorovateli – kameře. Pokud model pracuje pouze se světlem přímo ze světelných zdrojů (primárních zdrojů), pak je označován jako lokální osvětlovací model. Pokud do výpočtu zahrnuje i světlo ze sekundárních zdrojů, například odražené od ostatních objektů, tak model patří do kategorie globálních osvětlovacích modelů. V této práci se věnuji výhradně lokálním osvětlovacím modelům.

Zobrazovací rovnice [2] (*rendering equation*) je matematickým formalismem, který popisuje problém renderování scény. Její řešení udává pro každý bod povrchu každé plochy ve scéně a pro každý směr z ní vycházející radianci. Její zápis je následující:

$$L_o(\mathbf{x}, \omega) = L_e(\mathbf{x}, \omega) + L_r(\mathbf{x}, \omega),$$

$$L_r(\mathbf{x}, \omega) = \int_\Omega f(\mathbf{x}, \omega, \omega_i) L_i(\mathbf{x}, \omega_i) \cos\theta \; d\omega_i.$$

Po dosazení je plné znění zobrazující rovnice

$$L_o(\mathbf{x}, \omega) = L_e(\mathbf{x}, \omega) + \int_\Omega f(\mathbf{x}, \omega, \omega_i) L_i(\mathbf{x}, \omega_i) \cos\theta \; d\omega_i,$$

kde

- $L_o(\mathbf{x}, \omega)$ je celková odchozí radiance v bodě $\mathbf{x}$ ve směru prostorového úhlu ω;
- $L_e(\mathbf{x}, \omega)$ je celková vyzářená radiance z bodu $\mathbf{x}$ do směru ω;
- $L_r(\mathbf{x}, \omega)$ je celková odražená radiance z bodu $\mathbf{x}$ do směru ω.

Pro celkovou odraženou radianci L_r

- $\int_\Omega$ značí integrál přes všechny prostorové úhly ω_i polokoule Ω;
- $f(\mathbf{x}, \omega, \omega_i)$ je obousměrná odrazová distribuční funkce (BRDF) představená v sekci 2.2.4;
- $L_i(\mathbf{x}, \omega_i)$ je celková příchozí radiance do bodu $\mathbf{x}$ ze směru prostorového úhlu ω_i;
- θ je úhel mezi příchozím prostorovým úhlem ω_i a normálovým vektorem povrchu $\mathbf{n}$.

Tento vztah udává, že světlo odcházející z daného bodu na povrchu objektu do určitého směru se skládá ze světla vyzářeného tím objektem a z odraženého světla. Odražené světlo je součet všeho příchozího světla, přeškálovaného o poměr odrazu, z hemisféry kolem bodu na povrchu objektu.

Standardně udávanou rovnici pro lokální osvětlovací model pak získáme zanedbáním vyzářené radiance a úvahou, že příchozí radiance je pouze z primárních světelných zdrojů. Odchozí radiance v bodě $\mathbf{x}$ směrem ω je pak dána vztahem

$$L_o(\mathbf{x}, \omega) = L_r(\mathbf{x}, \omega) = \int_\Omega f(\mathbf{x}, \omega, \omega_i) L_i(\mathbf{x}, \omega_i) \cos\theta \; d\omega_i.$$

Odchozí radiance z bodu směrem do kamery přímo udává barvu bodu, který kamera vidí. V praxi se tato barva typicky počítá přes empirický model, který aproximuje reálné hodnoty, a tudíž není tak výpočetně náročný. Výpočetně nenáročné empirické modely jsou perfektními kandidáty pro real-time rendering.

3.7.1 Osvětlení v moderních herních enginech

Většina moderních herních enginů, jako Unity nebo Unreal Engine, je schopná v reálném čase používat globální osvětlovací modely – Unity používá systém *Enlighten* [12] a Unreal používá systém *Lumen* [13] k docílení fotorealistického real-time renderingu.

Pro potenciálně lepší výsledky se používá technik, kde se světlo předpočítá pro celou scénu, a následně je možné si scénu procházet v reálném čase. Oba zmíněné enginy nabízí tuto funkcionalitu a zároveň možnost tyto dva přístupy (předvýpočet a real-time výpočet) kombinovat.

Část II

Praktická část

Kapitola 4

Návrh real-time rendereru

Renderovací systém – renderer – jako takový je zde chápán jako systém tříd a funkcí, které umožňují uživateli efektivně vykreslovat obsah na obrazovku. Tento obsah může být obecně jak 2D, tak 3D, ovšem tato práce se zaměří výhradně na renderování 3D scén, tedy jakoukoliv zmínkou o rendereru je myšlen 3D renderer. Tato kapitola je zaměřena na samotný návrh a požadavky real-time rendereru.

4.1 Požadavky

Jako pro každý software je dobré stanovit obecné požadavky, **kvality**, které by výsledný produkt měl mít. Některé požadavky z následujícího výčtu je možné aplikovat i na obecný renderer, ovšem v této práci se vztahují konkrétně na real-time renderer, který je primárně určen do herního enginu.

Rychlost

Pro nejkvalitnější výsledky aplikací používající vykreslovací program (převážně her) je velmi důležitá plynulost obrazu. Rychlost je esenciální pro renderování v reálném čase. Je nutné aby uživatel viděl reakce na jeho příkazy co nejrychleji. Pomalejší zážitek kazí dojem z výsledné aplikace. V rendereru je tak velmi důležité dělat optimalizace a uvažovat o nich již v procesu návrhu.

Obecnost

Renderer by měl mít možnost používání více technologií. Hlavním bodem je podpora více renderovacích API (*Rendering Application Programming Interface*). Žádné renderovací API není možné použít na všech systémech (alespoň ne efektivně). Některá jsou specifická pro konkrétní platformu (*Direct3D* pro Windows a Xbox, *Metal* pro Apple). Aby renderer fungoval na co nejvíce zařízeních, je důležité mít funkční systém pro dynamické přepínání mezi těmito renderovacími API.

Rozšiřitelnost a Flexibilita

Nové lepší technologie a postupy vznikají každým dnem. Pro zachování relevantnosti rendereru do budoucnosti je nutné počítat s rozšířeními, případně i předefinováním celých systémů. Určitý systém modularity, kde se změnou jednoho systému se nepokazí žádný další, by měl být přítomen.

Jednoduchost

Používání rendereru by neměla být vysoce technická záležitost. Funkce a metody poskytnuty pro uživatele by měly být jasně a zřetelně definované. K dispozici by měly být *high-level* funkce a metody, které pomohou uživateli plnit komplexní příkazy pomocí malého množství potřebných instrukcí.

4.2 Návrh architektury

Tento renderer je navrhnut jako jeden ze systému herního enginu. Je přepokládáno, že bude využívat určité dostupné systémy z herního enginu. Je jimi například systém událostí (*event system*) a logovací systém (*logging system*). Jejich integrace do rendereru je nutná pro koherentní práci s enginem.

Samotný renderer není spustitelná aplikace, nýbrž množina tříd a funkcí, které jsou uživateli dostupné jako část herního enginu ve formě knihovny. Konkrétně v tomto případě jako statická knihovna. To znamená, že při kompilaci cílové aplikace uživatele se tato knihovna stane součástí spustitelného kódu. Výhodou je, že kompilátor může lépe optimalizovat tento kód, jelikož zná veškerý kontext. Nevýhodou je vetší velikost souboru aplikace. Alternativou ke statické knihovně je knihovna sdílená, která se do aplikace linkuje dynamicky a až v době běhu aplikace. To může být místy výhodou, ovšem zde jsem usoudil, že potenciální optimalizace jsou výhodnější.

Uživatel by měl možnost vykreslit 3D model s geometrií dodanou z externího souboru (formáty *obj*, *fbx*, *gltf*, . . .) a materiálem specifikovaným ať už v externím souboru (formát *mlt*), nebo ručně vytvořeným v rozhraní rendereru. Způsob vykreslování si může přizpůsobit zvolením shaderu pro konkrétní objekty. Tento shader může uživatel dodat vlastní, nebo by měl být dostupný alespoň jeden výchozí shader definovaný jako součást rendereru. Pokud materiál/geometrie/shader budou chybné, renderer by měl uživateli zahlásit chybu a zaevidovat ji do logovacího systému. Neměl by však ukončit aplikaci, nýbrž nastavit výchozí chybné hodnoty, aby uživatel mohl vizuálně zjistit, o který element se ve scéně jedná. Pro chybný materiál by tato výchozí chybná hodnota mohla vypadat jako materiál s výchozími hodnotami, ale s výraznou barvou. Chybný shader by, obdobně jako materiál, byl nahrazen výchozím shaderem, který by ovšem přepisoval všechny barvy na jednu výraznou. Chybná geometrie by se mohla nahradit předem daným výchozím tvarem, který by sloužil podobně jako výrazná barva k upozornění uživatele na chybu.

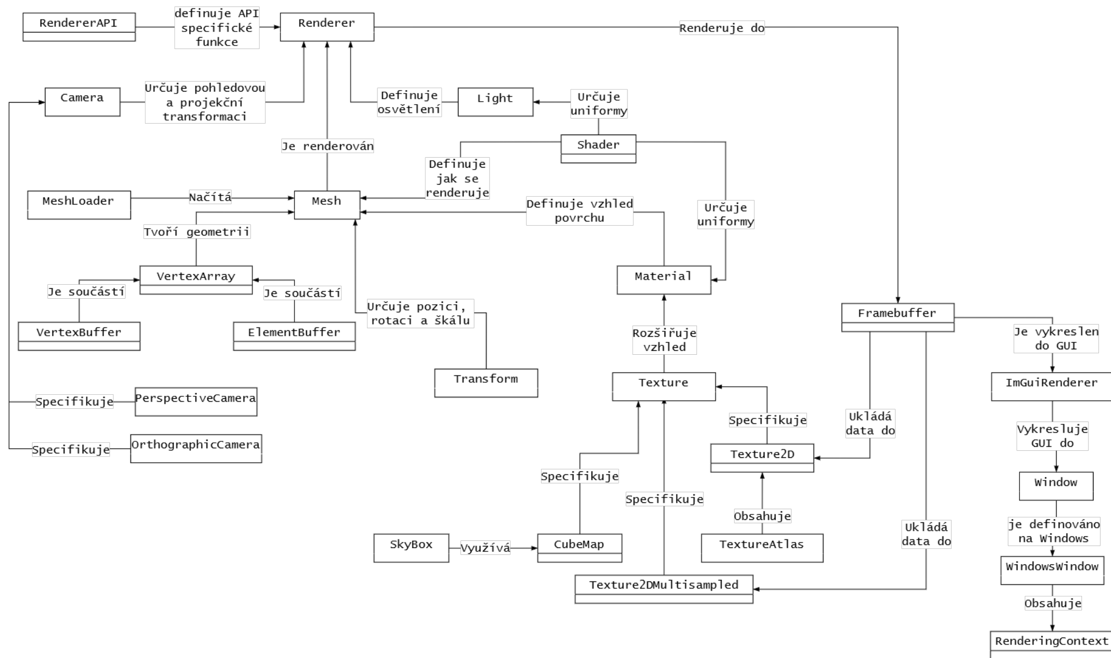
K upozornění uživatele na chyby by mělo dojít pouze v omezených případech. Pokud by například uživatel vyvíjel hru, renderer by byl v režimu *debug*, který je zaměřen na hledání a odstraňování chyb. Ovšem ve finální aplikaci, kterou uživatel plánuje sdílet, by měl mít možnost vypnout explicitní vizuální upozorňování na chyby. Režim, který na chyby explicitně neupozorňuje, se nazývá *release*. Na příkladu s vývojem hry – poté co uživatel prohlásí hru za dokončenou, tak ji sestaví v režimu *release* a tu bude dále distribuovat.

Pro definici objektů určených k renderování a zjednodušení práce uživatele se definují mimo tříd rendereru také třídy, které uchovávají data a definují speciální funkcionalitu pro jejich úpravy – datové struktury. Většina datových struktur je závislá na zvoleném renderovacím API a jejich implementace se proto pro různé API liší. Tento fakt je vyřešen definováním abstraktního rozhraní každé z takovýchto tříd, například *Shader*, který poté pro konkrétní API bude mít svoji implementaci – například *OpenGLShader*. Renderovací API je zvoleno před spuštěním aplikace a při tvorbě objektu *Shader* se vytvoří instance *OpenGLShader*. Tento fenomén se nazývá polymorfismus.

Některé datové struktury je vhodné modelovat pomocí objektově orientovaného přístupu, jiné pomocí kompozice. Pro potenciálně větší výkon se hojněji využívá principu kompozice, jelikož na rozdíl od dědičnosti a objektů nepoužívá velké množství abstrakcí, ale skládá datové struktury do sebe. Objektově orientovaný přístup modeluje data pomocí vztahů tříd mezi sebou. Nadtřída může definovat obecné rozhraní, které záleží na konkrétní implementaci, které poskytuje až podtřída dědící z ní (*OpenGLShader* dědí z *Shader*).

4.3 Diagram tříd

Na obrázku 4.1 je vidět diagram tříd s jejich vzájemnými vztahy. Obdélník vždy značí třídu a šipky značí vztahy, které jsou popsány ve směru šipek. Třídy, které mají svůj obdélník rozdělený ve spodní části, jsou pouze abstraktními rozhraními, které jsou dodefinovány pro každé renderovací API zvlášť v separátní třídě.



Obrázek 4.1 Diagram tříd rendereru. Jména tříd jsou napsaná v obdélnících a tyto obdélníky jsou spojeny šipkami znázorňující nějaký vztah, který je vždy popsán ve směru šipky. Třídy, které mají svůj obdélník rozdělený ve spodní části, jsou pouze abstraktními rozhraními, které jsou dodefinovány pro každé renderovací API zvlášť

Kapitola 5
Integrace modelu BTF

Pro popis interakce světla s povrchem se v real-time renderování běžně používá model BRDF nebo BSDF – záleží, zda-li u materiálu uvažujeme propustnost světla (BSDF), nebo ne (BRDF). Model BTF nemodeluje tuto vlastnost, tudíž jeho použití se bude výhradně týkat neprůhledných materiálů a bude tak nahrazovat běžně užívaný model BRDF.

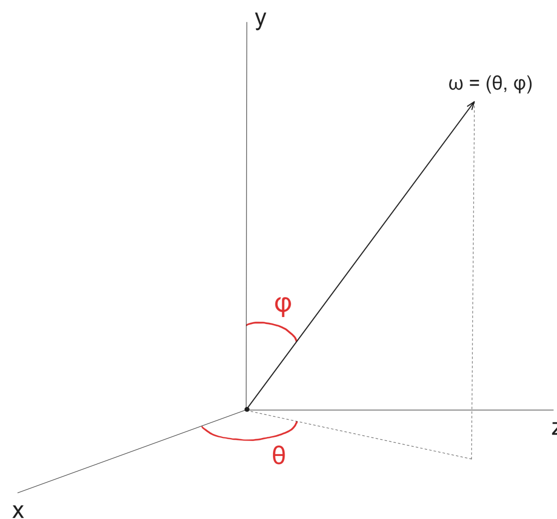
Na rozdíl od BRDF, BTF závisí na pozici na povrchu materiálu a patří do rodiny texturovaných modelů. Jelikož má BRDF homogenní povrch, lze jej popsat jak měřením reálných materiálů, tak i matematickými modely. Tyto matematické modely se mohou lišit komplexitou a teorií, která za nimi stojí. Slouží avšak pouze jako aproximace, kdežto reálná měření poskytují téměř realistický výsledek. U BTF se pak pracuje pouze s reálnými měřeními a díky jejich texturované povaze poskytují realističtější dojem z materiálu než při použití modelu BRDF. Některá měření jsou dostupná z veřejných databází. Jednou z nejstarších je databáze Columbia–Utrecht Reflectance And Texture Database (CUReT)[1], zveřejněna v roce 1999 autory Kristin J. Dana, Bram Van Ginneken, Shree K. Nayar a Jan J. Koenderink. Novější databází, ze které jsou použity materiály v této práci, je UTIA[2] BTF database [14] od autorů Haindl M., Filip J. a Vávra R.

BTF je funkce, která je závislá na směru příchozího světla, směru pohledu a pozici na povrchu. Ve formálním modelu je také závislost na vlnové délce, ovšem ta se zde pro zjednodušení vypouští. Hodnoty této funkce se v praxi měří v diskrétních hodnotách směru pohledu a směru světla, které se typicky reprezentují pomocí jednotkových sférických souřadnic. V praktické části této práce jsou definovány jako uspořádaná dvojice $\omega = (\theta, \varphi)$, kde $\theta \in \langle 0, 2\pi)$ značí azimut a $\varphi \in \langle 0, \frac{\pi}{2})$ značí elevaci. Pro popsání celé koule by elevace musela nabývat hodnot $\langle 0, \pi)$, ovšem materiálové modely operují na polokouli nad bodem povrchu, a tedy hodnoty elevace jsou omezené pouze do $\frac{\pi}{2}$.

5.1 Reprezentace dat

Měření BTF reálných materiálů je proces, ve kterém se získávají snímky materiálu z určitých úhlů pohledu s určitým úhlem příchozího osvětlení. V praxi se jedná o aparát, kde jsou z bodů na polokouli pořizovány snímky. Do těchto bodů je umístěna kamera, která odpovídá úhlu pohledu (sférickým souřadnicím tohoto bodu). Na jiný (nebo ten samý) bod se umístí osvětlení, které má také sférické souřadnice odpovídající tomuto bodu. Výsledkem takového jednoho měření pro jeden bod osvětlení a jeden bod pohledu je jeden snímek, který obsahuje nasnímaný povrch materiálu kamerou. Toto se provede pro všechny předdefinované body, tak aby pro všechny úhly

[1]Dostupná z https://www.cs.columbia.edu/CAVE/software/curet/.
[2]Dostupná z http://btf.utia.cas.cz/.



Obrázek 5.1 Znázornění jednotkových sférických souřadnic. Pozice v prostoru je určena úhlem θ kolem osy y, který se nazývá azimut, a úhlem φ kolem osy x, který se nazývá elevace. Azimut může nabývat hodnot od 0 do 2π a elevace nabývá od 0 do $\frac{\pi}{2}$

pohledu byly nasnímány všechny úhly osvětlení. Výsledkem je kolekce $n \cdot m$ snímků, kde n je počet bodů kamer a m je počet bodů osvětlení. Dohromady pak tvoří jakousi tabulku, kde pro daný směr osvětlení a směr pohledu vrací snímek povrchu. Velikost snímků by měla být stejná pro jednotlivé snímky a typicky čtvercová.

Způsobů, jak tyto body na polokouli rozmístit, je mnoho. V následujících sekcích jsou rozebrány dvě možnosti, které jsou použity v této práci. Od zvolení rozmístění bodů se odvíjí také formát, ve kterém jsou výsledné snímky uloženy.

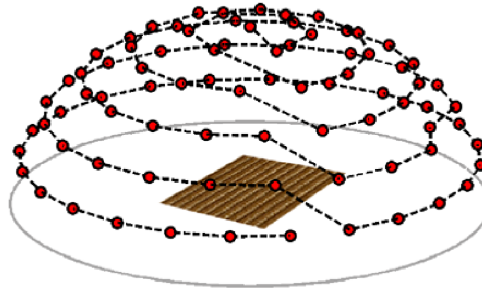
5.1.1 Rozložení UTIA BTF database

Body pro měření jsou rozmístěny na polokouli na elevačních hladinách. Tyto hladiny odpovídají kružnicím, které jsou od sebe odsazeny o 15° elevace. Celkově je jich šest, kde poslední hladina odpovídá elevaci $\varphi = 75°$. Na každé vrstvě se nachází různý počet bodů – čím vyšší hladina od nultého stupně elevace, tím více bodů. V tabulce 5.1 jsou rozepsány počty bodů pro jednotlivé hladiny. Rozložení všech bodů na polokouli je vidět na obrázku 5.2.

Hladina (φ)	Počet bodů
0°	1
15°	6
30°	12
45°	18
60°	20
75°	24

Tabulka 5.1 Počet bodů na hladinách polokoule v UTIA BTF database

Výsledkem měření je pak $81 \cdot 81$ snímků. Tyto snímky jsou pak pojmenovány podle úhlů, kterým odpovídají. Například název nejvyššího bodu na souřadnicích $\omega_i = \omega_v = (0, 0)$ by byl *tl000_pl000_tv000_pv000.png* pro formát snímku *png*. Takové názvové schéma mají materiály stažené přímo z databáze UTIA BTF database. Zde je nutno podotknout, že tato databáze



Obrázek 5.2 Rozložení bodů na polokouli pro měření BTF v databázi UTIA BTF database. Elevace je odstupňovaná po 15°, ovšem pouze do 75°, celkově je tedy 6 hladin bodů. Počet bodů na jednotlivých hladinách se liší – čím vyšší hladina od nultého stupně elevace, tím více bodů. Celkově se jedná o 81 bodů [5]

používá jiné značení sférických souřadnic – θ značí elevaci a v uspořádané dvojici souřadnic se nachází jako první, a naopak φ značí azimut a v uspořádané dvojici se nachází jako druhý. Názvy souborů pak lze chápat jako po sobě jdoucí azimut světla (tl), elevaci (pl) světla, azimut pohledu (tv), elevaci pohledu (pv) ve stupních.

Pro jednodušší použití při renderování jsem materiály z této databáze spojil dohromady do atlasu textur. Atlas textur je pouze jeden velký obrázek, ve kterém jsou uloženy všechny textury (v tomto případě snímky měření). Vytvářím vždy jeden atlas pro jeden úhel příchozího světla. Tedy z 81·81 jednotlivých snímků se stane pouze 81 atlasů textur. Ty jsou optimálnější pro použití v *rendering pipeline*. Tyto jednotlivé atlasy se dále dají spojit do jedné 3D textury, čímž se docílí ještě optimálnějšího výsledku, jelikož celý materiál se dá předat jako jedna textura. Tato finální textura je velmi datově objemná textura (materiály z databáze se pohybují mezi 1,8 GB a 3,8 GB), a proto je pro praktické použití v real-time renderování s více materiály najednou nutné zavést optimalizace (sekce *Optimalizace atlasů textur* 5.2).

Konkrétně tento atlas tvořím tak, že skládám po sobě jdoucí snímky seřazené primárně podle elevace a sekundárně podle azimutu. Dělám tak od levého dolního rohu směrem vpravo. Jelikož se celkem jedná o 81 snímků, které ukládám do atlasu, tak výsledný atlas je čtvercový a skládá se z 9 x 9 snímků. Vždy, když zaplním jednu řadu, začnu skládat snímky opět zleva o jednu řadu výše. Výsledný atlas je znázorněn na obrázku 5.3.

(225, 75)	(240, 75)	(255, 75)	(270, 75)	(285, 75)	(300, 75)	(315, 75)	(330, 75)	(345, 75)
(90, 75)	(105, 75)	(120, 75)	(135, 75)	(150, 75)	(165, 75)	(180, 75)	(195, 75)	(210, 75)
(306, 60)	(324, 60)	(342, 60)	(0, 75)	(15, 75)	(30, 75)	(45, 75)	(60, 75)	(75, 75)
(144, 60)	(162, 60)	(180, 60)	(198, 60)	(216, 60)	(234, 60)	(252, 60)	(270, 60)	(288, 60)
(340, 45)	(0, 60)	(18, 60)	(36, 60)	(54, 60)	(72, 60)	(90, 60)	(108, 60)	(126, 60)
(160, 45)	(180, 45)	(200, 45)	(220, 45)	(240, 45)	(260, 45)	(280, 45)	(300, 45)	(320, 45)
(330, 30)	(0, 45)	(20, 45)	(40, 45)	(60, 45)	(80, 45)	(100, 45)	(120, 45)	(140, 45)
(60, 30)	(90, 30)	(120, 30)	(150, 30)	(180, 30)	(210, 30)	(240, 30)	(270, 30)	(300, 30)
(0, 0)	(0, 15)	(60, 15)	(120, 15)	(180, 15)	(240, 15)	(300, 15)	(0, 30)	(30, 30)

Obrázek 5.3 Ukázka rozložení snímků v atlasu textur pro materiál z databáze UTIA BTF. Dvojice (θ, φ) značí sférickou souřadnici bodu, ze kterého byl snímek pořízen. Hodnoty jsou udány ve stupních pro jednoduší demonstraci. Atlas začíná v levém dolním rohu a postupuje po řádcích vpravo. Po konci řádku je navazující snímek na vyšším řádku opět v levém rohu

5.1.2 Vlastní zjednodušené rozložení

K vytvoření vlastního materiálu jsem použil svůj Blender *add-on* jménem *DBTF Creator*[3], který slouží ke tvorbě DBTF a BTF materiálů z jakékoliv virtuální scény (model DBTF je podobný jako BTF, jen se v něm místo textur používají dynamické textury, více v sekci 5.3 *Použití modelu DBTF*). Vytvořil jsem v něm scénu s travnatým povrchem a pořídil BTF materiál s azimutálním a elevačním dělením 16 x 8. Tato dělení rozdělují uniformně polokouli napříč azimuty a elevacemi, konkrétně u dělení 16 x 8 bude bod každých $\frac{360°}{16} = 22,5°$ v azimutálním kontextu a každých $\frac{90°}{8} = 11,25°$ v elevačním kontextu. Pro většinu scén je nutné nastavit maximální úhel elevace, aby měření scény neprobíhalo zpod povrchu a aby povrch bylo vůbec možné měřit. Během tvoření materiálu travnatého povrchu jsem tuto maximální elevaci nastavil na 55°. Rozestupy jednotlivých bodů v elevačním kontextu pak budou dané vztahem $\frac{\text{maxElevace}}{n}$, kde n značí počet dělení elevace. U travnatého povrchu pak tyto body budou elevačně rozestoupeny $\frac{55°}{8} = 6,875°$ od sebe.

Takto získané snímky jsem podobně jako u materiálu z databáze UTIA BTF spojil do atlasů textur. Jeden atlas v sobě má snímky ze všech pohledů korespondujících k jednomu směru světla. Schéma indexů snímků tohoto atlasu je vyobrazeno v obrázku 5.4. Konkrétně pro materiál travnaté plochy obsahuje jeden atlas $16 \cdot 8 = 128$ snímků, každý z nich má rozměry 512 x 512 pixelů

[3]Dostupný jako projekt z předmětu BI-PGA na FIT ČVUT zde: https://gl.iga.ksi.fit.cvut.cz/bi-pga/b231/3d/nejedzd3.

a velikost v průměru okolo 500 kB. Celkem je $16 \cdot 8 = 128$ atlasů, a tedy celkový počet snímků je rovný $128 \cdot 128 = 16\ 384$. Jednoduchý výpočet pak udá odhad celkové velikosti materiálu na $16\ 384 \cdot 500$ kB $\approx 8,2$ GB. Upravením parametrů dělení nebo velikostí snímků by se tento datový objem mohl snížit, ovšem za ceny výsledné kvality materiálu. Případné další optimalizace jsou rozebrány v sekci 5.2 *Optimalizace atlasů textur*.

(0, 7)	(1, 7)	(2, 7)	(3, 7)	(4, 7)	(5, 7)	(6, 7)	(7, 7)	(8, 7)	(9, 7)	(10, 7)	(11, 7)	(12, 7)	(13, 7)	(14, 7)	(15, 7)
(0, 6)	(1, 6)	(2, 6)	(3, 6)	(4, 6)	(5, 6)	(6, 6)	(7, 6)	(8, 6)	(9, 6)	(10, 6)	(11, 6)	(12, 6)	(13, 6)	(14, 6)	(15, 6)
(0, 5)	(1, 5)	(2, 5)	(3, 5)	(4, 5)	(5, 5)	(6, 5)	(7, 5)	(8, 5)	(9, 5)	(10, 5)	(11, 5)	(12, 5)	(13, 5)	(14, 5)	(15, 5)
(0, 4)	(1, 4)	(2, 4)	(3, 4)	(4, 4)	(5, 4)	(6, 4)	(7, 4)	(8, 4)	(9, 4)	(10, 4)	(11, 4)	(12, 4)	(13, 4)	(14, 4)	(15, 4)
(0, 3)	(1, 3)	(2, 3)	(3, 3)	(4, 3)	(5, 3)	(6, 3)	(7, 3)	(8, 3)	(9, 3)	(10, 3)	(11, 3)	(12, 3)	(13, 3)	(14, 3)	(15, 3)
(0, 2)	(1, 2)	(2, 2)	(3, 2)	(4, 2)	(5, 2)	(6, 2)	(7, 2)	(8, 2)	(9, 2)	(10, 2)	(11, 2)	(12, 2)	(13, 2)	(14, 2)	(15, 2)
(0, 1)	(1, 1)	(2, 1)	(3, 1)	(4, 1)	(5, 1)	(6, 1)	(7, 1)	(8, 1)	(9, 1)	(10, 1)	(11, 1)	(12, 1)	(13, 1)	(14, 1)	(15, 1)
(0, 0)	(1, 0)	(2, 0)	(3, 0)	(4, 0)	(5, 0)	(6, 0)	(7, 0)	(8, 0)	(9, 0)	(10, 0)	(11, 0)	(12, 0)	(13, 0)	(14, 0)	(15, 0)

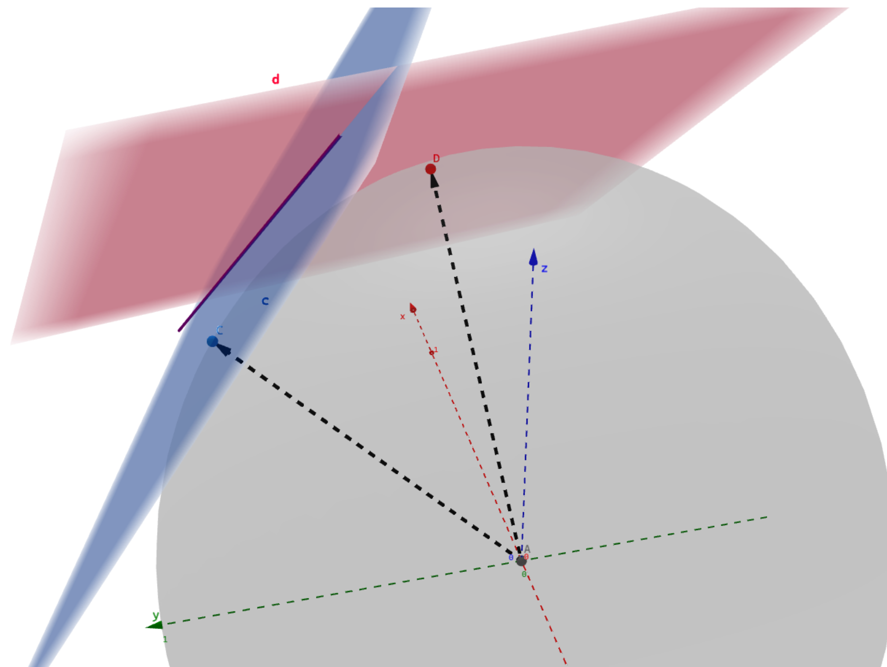
Obrázek 5.4 Ukázka indexů azimutu a elevace v zjednodušeném atlasu textur pro materiál s dělením 16 x 8. Dvojice (i, j) v obrázku značí bod (θ_i, φ_j), kde $\theta_i = \frac{2\pi}{16} \cdot i$ a $\varphi_j = \frac{\text{maxElevace}}{8} \cdot j$. Počátek atlasu je v levém dolním rohu, elevace roste směrem nahoru a azimut směrem vpravo

Tento styl rozložení není optimální a v porovnání s rozložením použitým v UTIA BTF databázi je výrazně horší – je datově objemnější, jelikož obsahuje přebytečné informace. Uniformní rozložení bodů na polokouli je špatné, jelikož počet snímků potřebných k vykreslení je vyšší blíže u středu polokoule (nejvyšší elevace) a nižší u vrcholu polokoule (nejnižší elevace). Tento fenomén lépe vystihuje rozložení UTIA BTF databáze (tabulka 5.1). Byť tento vlastní formát není optimální pro praktické použití, tak přímočarost tvorby atlasu je nespornou výhodou. Pouhým pohledem na atlas je možno vypozorovat některé vlastnosti materiálu a jejich změny v různých pohledech.

5.2 Optimalizace atlasu textur

Atlas textur není optimální reprezentací měření povrchu materiálu při konkrétních světelných podmínkách. Uvažujme jeden atlas textur, neboli sadu snímků pořízených z bodů na polokouli, vždy pro jeden směr osvětlení. Jak je demonstrováno na obrázku 5.5, pokud tyto snímky vyneseme na polokouli z měření jako obdélníkové části rovin, které jsou kolmé na vektor reprezentující sférické souřadnice bodů korespondujících k daným snímkům, tak v závislosti na velikosti těchto obdélníků mohou nastat různé nežádoucí situace. Velikost obdélníků je přímo úměrná zvolenému rozlišení snímků, tedy pokud zvolíme dostatečně velké rozlišení, tak vynesené obdélníky se budou překrývat. Pokud naopak zvolíme dostatečně malé rozlišení, tak na sebe obdélníky nebudou navazovat. Pokud by nastala situace, kde se sousedící obdélníky v jedné elevační hladině budou perfektně dotýkat a nenastane překryv, pak při uvážení návaznosti mezi nižší či vyšší hladinou vznikne nedokonalost – buď se celé hladiny budou překrývat, nebo se nebudou dotýkat vůbec. Pokud se budou překrývat, pak se v atlasu nacházejí redundantní informace a může docházet ke konfliktům při vzorkování materiálu během renderování. Pokud naopak budou mezi jednot-

livými snímky díry, materiál nebude moci být vzorkován spojitě (spojitost zde není v pravém matematickém smyslu, jelikož snímky jsou tvořeny diskrétními pixely).



Obrázek 5.5 Znázornění problému s měřením materiálu. Mějme jednotkovou polokouli, ze které měříme materiál v bodě v počátku. Pro sousední body měření C a D na stejné elevační hladině budou finální snímky okupovat rovinu c a d. Fialová úsečka znázorňuje, kde se roviny c a d protínají. V závislosti na zvolené velikosti snímků se budou tato měření buď překrývat, nebo na sebe nebudou navazovat. Je možné, že snímky budou nastaveny perfektně tak, že se budou přesně dotýkat. To ovšem může být narušeno snímky o elevační hladinu výše

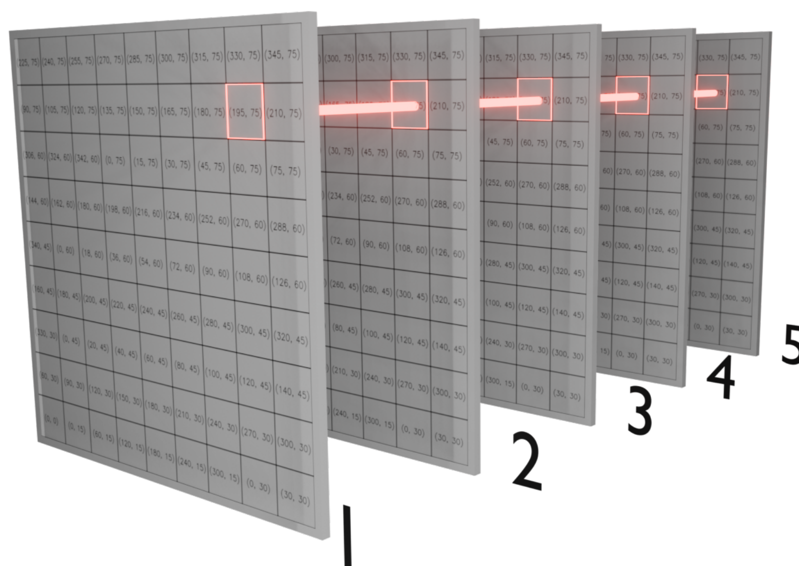
K vyřešení tohoto problému navrhuji použití sférické textury místo atlasu textur. Pro korektní vytvoření sférické textury je nutné, aby se snímky částečně překrývaly. Takové snímky se dají transformovat promítnutím na sférický povrch, který rozměrově odpovídá polokouli, ze které byly měřeny. Jelikož mezi snímky existoval překryv, je možné je po transformaci složit přesně tak, aby na sebe pasovaly. Výsledná sférická textura by pak byla koherentní reprezentací povrchu bez redundancí a nedostatků.

Pro vzorkování materiálu reprezentovaného sférickou texturou během renderování je možno vybrat část plochy sférické textury kolem bodu, který odpovídá sférickým souřadnicím daného směru pohledu, a přetransformovat ji zpátky ze sférické projekce do obdélníkové textury. Takto by bylo možno měnit velikost získané textury dynamicky během renderování jednoduše podle velikosti vybrané plochy kolem bodu ve sférické textuře.

5.3 Použití modelu DBTF

Model Dynamic Bidirectional Texture Function [15] (DBTF) je podobně jako model BTF a BRDF používaný k popisu interakce materiálu se světlem. Je velmi podobný jako BTF, s tím rozdílem, že uvažuje navíc jeden parametr – čas t. Pokud uvažujeme, že BTF pracuje s texturami, pak DBTF pracuje s dynamickými texturami. Dynamická textura je taková textura, která kromě prostorové homogenity vykazuje také časovou homogenitu. Je to tedy textura, která se v čase předvídatelně mění.

V praxi se dynamické textury reprezentují typicky jako sekvence snímků nebo videa. Materiál pro model DBTF by tedy nejjednodušeji mohl vypadat jako sekvence BTF materiálů. Pokud uvažujeme uložení BTF materiálů v atlasech textur (vždy jeden atlas pro jeden směr osvětlení), pak můžeme vytvořit sekvenci atlasů pro jeden směr osvětlení. Pokud se zaměříme na jednu konkrétní pozici snímku v atlase, pak snímky na této pozici v sekvenci atlasů tvoří dynamickou texturu (obrázek 5.6).



Obrázek 5.6 Znázornění dynamické textury v sekvenci atlasů textur. Na zvýrazněné pozici napříč atlasy je uložena dynamická textura. Čísla jedna až pět značí index atlasu v sekvenci

Přidání dimenze výrazně zvětší objem dat potřebný pro uložení celého materiálu. Záleží na počtu snímků v dynamické textuře, jelikož velikost roste lineárně vůči nim. Při úvaze pouhých 60 snímků dynamické textury u materiálu travnatého povrchu, který původně měl přibližně 8,2 GB, se velikost dostává k téměř 500 GB. Zpracovat takový objem dat v reálném čase je velmi těžký, potenciálně až nemožný úkol. Na základě této úvahy označuji model DBTF jako nepraktický pro použití v real-time renderingu a nadále budu pracovat pouze s modelem BTF.

Kapitola 6

Implementace real-time rendereru

Renderer jsem navrhoval jako součást vlastního herního enginu Crank, který je inspirován sérií videí Game Engine [16] na platformě Youtube od Yan Chernikova (The Cherno). V této sérii autor popisuje konkrétní implementaci herního enginu, ze které jsem čerpal inspiraci a částečně přejímal. Návrh rendereru jsem již tvořil sám.

Celková implementace se skládá ze dvou částí – renderer a aplikační část. Tato kapitola obsahuje implementaci rendereru. Implementace aplikační části je obsahem následující kapitoly 7.

V následujícím textu jsou obsaženy pouze nejdůležitější třídy. Některé zřídka používané nebo málo rozvinuté třídy nejsou v následujícím textu zmíněny. Třídy definované v rámci herního enginu, ale ne rendereru, také nejsou zmíněny. Mezi ně patří i v návrhu zmíněný renderovací systém GUI (třída *ImGuiRenderer*), abstrakce okna a implementace okna pro Windows (třídy *Window* a *WindowsWindow*), renderovací kontext (třída *RenderingContext*) a skybox (třída *SkyBox*).

6.1 Použité technologie

Renderer jako součást herního enginu Crank je implementován v programovacím jazyce C++, konkrétně standardem C++17. Renderovacím API je OpenGL verze 4.5. Engine je zatím implementovaný pouze pro platformu Windows. Součástí herního enginu jsou také následující C/C++ knihovny:

- *Dear ImGui* – tvorba grafického rozhraní;
- *OpenGL Mathematics (GLM)* – poskytnutí matematických funkcí a datových struktur pro matematické objekty;
- *GLFW* – jednoduché API na práci s okny, kontexty a událostmi;
- *GLAD* – načtení funkcí OpenGL;
- *Open Asset Import Library (assimp)* – načítání 3D objektů v různých formátech;
- *spdlog* – logovací systém;
- *stb_image* – načítání obrázků do paměti.

6.2 Datové struktury závislé na renderovacím API

Třídy v této sekci jsou závislé na renderovacím API, a tedy jsou definované obecně jako abstraktní rozhraní, které je pak implementováno ve specializované třídě pro jednotlivé API. Schéma pojmenovávání tříd je takové, že pro třídu *Class* by její OpenGL implementace byla ve třídě *OpenGLClass*.

Pokud chce uživatel instancovat některou z těchto tříd, pak musí volat statickou funkci *Create*, která vytvoří instanci specializované třídy. Na příkladu v ukázce kódu 6.1 je tato funkce vidět pro třídu *VertexBuffer*. Pokud není renderovací API zvoleno, nebo je neznámé, je zahlášena chyba do logovacího systému a metoda vrací prázdný pointer. Úspěšné volání této metody pro OpenGL API vrací instanci *OpenGLVertexBufferu* obalenou jako referenci *Ref*, která funguje stejně jako *std::shared_ptr*.

```
Ref<VertexBuffer> VertexBuffer::Create(size_t byteSize)
{
    switch (RendererAPI::GetAPI())
    {
    case RendererAPI::API::None:
        CR_CORE_WARN(
            "VertexBuffer::Create(): No Rendering API is selected."
                );
        return nullptr;

    case RendererAPI::API::OpenGL:
        return CreateRef<OpenGLVertexBuffer>(byteSize);
    }

    CR_CORE_ERROR("VertexBuffer::Create(): Unknown Rendering API.");
    return nullptr;
}
```

Ukázka kódu 6.1 Metoda pro instancování VertexBufferu

Pro zjednodušení čtení kódu jsem u tříd, kde se přistupuje k jejich soukromým atributům, vypustil takzvané *getters* a *setters*. To jsou metody, které umožňují nastavovat a získávat hodnoty těchto soukromých atributů. Pokud je nějaká z těchto metod něčím atypická, tak jsem ji v ukázce kódu ponechal.

VertexBuffer

Třída *VertexBuffer* (ukázka kódu 6.2) abstrahuje *vertex buffer object* (VBO), pomocí kterého se ukládají data do paměti grafické karty. V konkrétní terminologii se tato data nazývají atributy a patří mezi ně například pozice, normály a texturové souřadnice jednotlivých vertexů.

Rozhraní nabízí základní metody pro *Bind* a *Unbind* a možnost nahrání dat typu *float*. V budoucnosti by bylo vhodné doplnit i jiné datové typy pro více možností optimalizací. Po nebo před nahráním by měl uživatel specifikovat rozložení dat ve VBO. K tomu slouží pomocná struktura *BufferLayout* a je nutné pomocí ní nastavit správné rozložení k nahraným datům pro správnou funcki VBO.

Tuto třídu nelze instancovat přímo, jelikož je definována jako abstraktní. Instance se získávají za použití statické funkce *Create* (představena na začátku sekce). Je možné ji vytvořit bez dat, pouze s předalokovanou pamětí, nebo rovnou s daty.

```
class VertexBuffer
{
public:
    virtual ~VertexBuffer() {}

    virtual void Bind() const = 0;
    virtual void Unbind() const = 0;

    virtual void SetData(size_t byteSize, float* data, size_t offset) = 0;
    virtual void SetLayout(const BufferLayout& layout) = 0;

    static Ref<VertexBuffer> Create(size_t byteSize);
    static Ref<VertexBuffer> Create(size_t byteSize, float* data);
};
```

Ukázka kódu 6.2 Rozhraní třídy VertexBuffer

ElementBuffer

Třída *ElementBuffer* (ukázka kódu 6.3) slouží jako abstrakce pro *element buffer object* (EBO), který se používá k optimalizaci velikosti dat. Jednotlivé trojúhelníky nejsou interpretovány přímo z vertexů ve VBO, nýbrž jsou vytvářeny z vertexů ve VBO na základě trojic indexů v EBO. EBO tedy slouží k uložení trojic indexů.

Rozhraní nabízí základní metody pro *Bind* a *Unbind* a možnost nahrání dat typu *unsigned int*. Nahrání dat může proběhnout do jakékoliv části alokovaného prostoru díky specifikování bytového offsetu od počátku v parametru metody *SetData*. V budoucnu by bylo dobré mít možnost nahrát data ve velikostně menším datovém typu jako například *unsigned short int* nebo *unsigned char*. Metoda *GetCount* vrací počet alokovaných míst pro indexy.

```
class ElementBuffer
{
public:
    virtual ~ElementBuffer() {}

    virtual void Bind() const = 0;
    virtual void Unbind() const = 0;

    virtual unsigned int GetCount() const = 0;
    virtual void SetData(
            unsigned int count, unsigned int* data, size_t offset
                ) = 0;

    static Ref<ElementBuffer> Create(unsigned int count);
    static Ref<ElementBuffer> Create(
            unsigned int count, unsigned int* data
                );
};
```

Ukázka kódu 6.3 Rozhraní třídy ElementBuffer

Shader

Úkolem třídy *Shader* (ukázka kódu 6.4) je definovat rozhraní pro tvorbu a práci s shadery. Shadery jsou definovány v jednom souboru formátu *glsl*, a to jak vertex, tak fragment shader. Jiné shadery nejsou zatím podporovány a OpenGL implementace počítá s tím, že vertex shader a fragment shader jsou definovány spolu v jednom souboru. Implementace by měla spojit tyto dva shadery do jednoho shader programu, který se pak používá pro vykreslování objektů.

Toto rozhraní umožňuje uživateli vybrat a zrušit výběr shaderu přes metody *Bind* a *Unbind*. Zajišťuje také předání uniformů (speciálních proměnných) do shaderu a uživatel se může dotázat na hodnotu a umístění jednotlivých uniformů, atributů a textur.

```
class Shader
{
public:
    virtual ~Shader() {}

    virtual void Bind() const = 0;
    virtual void Unbind() const = 0;

    virtual int GetLocation(const std::string& input) const = 0;

    // metoda PassUniform je také definována pro mat3, uvec3, uvec2,
    // vec4, vec3, vec2, float a int
    virtual void PassUniform(
            const std::string& name, const glm::mat4& value
                    ) = 0;

    using DataUMap = std::unordered_map<std::string, Shader::DataDescription>;
    virtual const DataUMap& GetUniforms() const = 0;
    virtual const DataUMap& GetAttributes() const = 0;
    virtual const DataUMap& GetTextures() const = 0;

    static Ref<Shader> Create(const std::string& fileName);

public:
    enum class DataType
    {
        None = 0, Bool, Float, Int, Vec2, Vec3, Vec4, Mat2,
        Mat3, Mat4, Sampler1D, Sampler2D, Sampler3D, SamplerCube
    };

    struct DataDescription
    {
        int Location;
        DataType Type;
    };
};
```

Ukázka kódu 6.4 Rozhraní třídy Shader a jejích pomocných struktur a definic

VertexArray

Třída *VertexArray* (ukázka kódu 6.5) je používána jako abstrakce *vertex array object* (VAO), který spojuje dohromady VBO a EBO. Rozhraní je navrženo pro párování jednoho EBO k více VBO. Informace o vertexech jako pozice, normály a texturovací souřadnice mohou být uloženy v separátních VBO, a tedy vzniká nutnost pro více VBO, kterým odpovídá právě jeden EBO.

Pro tvorbu třídy *VertexArray* je nutné poskytnou shader. To je nutné kvůli automatizaci přípravy dat do vhodného formátu pro daný shader. Uživatel by již při tvorbě VAO měl vědět, jakým shaderem data ve *vertex* a *element bufferech* chce vykreslovat, aby mohla být připravena na předání do shaderu.

Rozhraní nabízí metody *Bind* a *Unbind* pro zvolení VAO pro vykreslování a pro zrušení této volby. Nabízí možnost nastavit EBO a přidat jednotlivě VBO. Uživatel se může dotázat na počet vertexů a počet indexů v EBO.

```
class VertexArray
{
public:
    virtual ~VertexArray() {}

    virtual void Bind() const = 0;
    virtual void Unbind() const = 0;

    virtual void AddVertexBuffer(
                    const Ref<VertexBuffer>& vertexBuffer
                        ) = 0;
    virtual void AddElementBuffer(
                    const Ref<ElementBuffer>& elementBuffer
                        ) = 0;

    virtual const std::vector<Ref<VertexBuffer>>& GetVertexBuffers() = 0;
    virtual const Ref<ElementBuffer>& GetElementBuffer() = 0;
    virtual unsigned int GetIndexCount() const = 0;
    virtual unsigned int GetVertexCount() const = 0;

    static Ref<VertexArray> Create(const Ref<Shader>& shader);
};
```

Ukázka kódu 6.5 Rozhraní třídy VertexArray

Texture

Třída *Texture* je použita jako základní třída pro ostatní více specifické typy textur, které jsou odvozeny z této třídy. Jmenovitě to jsou *Texture2D*, *Texture2DMultisampled* a *CubeMap*.

Základní třída *Texture* (ukázka kódu 6.6) obsahuje tedy pouze metody společné pro všechny druhy textur. Je jimi nastavení slotu textury v shaderu (metoda *Bind*), nastavení parametru textury a nahrání dat.

Pro definici parametrů textury je použita struktura *Texture::Specification*. Ta obsahuje informaci o formátu textury, počtech bitů na kanál, šířce a výšce, počtu vrstev mip map a počet Multisample Anti Aliasing (MSAA) vzorků.

```
class Texture
{
public:
    virtual ~Texture() {}

    virtual unsigned int GetHandle() const = 0;

    virtual void SetParameter(Parameter parameter, int value) = 0;
    virtual void SetData(unsigned char* data) = 0;

    virtual void Bind(unsigned int slot = 0) const = 0;

    public:
    enum class Format { None = 0, RGB, RGBA, DEPTH24_STENCIL8 };

    struct Specification
    {
        Format Format;
        unsigned int BitsPerChannel;
        unsigned int Width, Height;
        unsigned int MipMapLevels;
        unsigned int MSAASamples;
    };
};
```

Ukázka kódu 6.6 Ukázka základní třídy Texture a struktur pro specifikování technických parametrů textury

Třída *Texture2D* slouží jako rozhraní pro dvou-dimenzionální textury. Může být vytvořena ze souboru (k načítání je použita knihovna *stb_image*) a nebo může být vytvořena prázdná dle zadané specifikace.

Třída *Texture2DMultisampled* je rozhraní pro dvou-dimenzionální texturu pro využití s algoritmem MSAA. Ten se používá pro potlačení jevu aliasingu, který zobrazuje jisté části textury jako ostře čtverečkované. Tato textura je hlavně použita ve *framebufferu*, do kterého je scéna renderována, který využívá metody MSAA. Může být vytvořena pouze prázdná dle dané specifikace.

Třída *CubeMap* slouží k načtení a použití textury typu *cubemap*. Jedná se o texturu skládající se z šesti dvou-dimenzionálních textur, které jsou namapovány na jednotlivé stěny krychle. Používá se k vykreslování pozadí scény. Může být vytvořena z šesti textur načtených ze souboru.

Definice tříd odvozených ze základní třídy *Texture* jsou vidět v ukázce kódu 6.7.

```
class Texture2D : public Texture
{
public:
    static Ref<Texture2D> Create(const std::string& path, bool useMipMaps);
    static Ref<Texture2D> Create(const Specification& spec);
    virtual Specification& GetSpec() = 0;
};
```

```
class Texture2DMultiSampled : public Texture
{
public:
    static Ref<Texture2DMultiSampled> Create(const Specification& spec);
    virtual Specification& GetSpec() = 0;
};
```

```
class CubeMap : public Texture
{
public:
    static Ref<CubeMap> Create(const std::vector<std::string>& paths);
};
```

Ukázka kódu 6.7 Rozhraní specializovaných textur Texture2D, Texture2DMultiSampled a CubeMap

Framebuffer

Třída *Framebuffer* (ukázka kódu 6.8) definuje rozhraní pro datovou strukturu, která se používá, jako paměť do které se přímo vykreslují data. Může obsahovat více vrstev, například pro barvu a hloubku. V této konkrétní implementaci je použita pouze k získávání vykresleného snímku v podobě 2D textury.

V OpenGL implementaci je možnost využít algoritmu MSAA pro potlačení efektu aliasingu. Toho se dosáhne renderováním do multisampled textury, která je pak uživateli dostupná jako klasická dvou-dimenzionální textura.

Rozhraní nabízí možnost zvolení a zrušení zvolení přes *Bind* a *Unbind* a také možnost změnit velikost framebufferu. Pro získání vyrenderovaného snímku v podobě dvou-dimenzionální textury slouží metoda *GetColorAttachment*. Parametry *framebufferu* jako velikost, počet MSAA vzorků nebo zda-li je framebuffer právě využíván pro vykreslování, jsou definovány ve struktuře *Framebuffer::Specification*.

```
class Framebuffer
{
public:
    virtual void Overwrite() = 0;
    virtual void Resize(unsigned int width, unsigned int height) = 0;

    virtual void Bind() const = 0;
    virtual void Unbind() const = 0;

    virtual unsigned int GetHandle() const = 0;
    virtual Ref<Texture2D> GetColorAttachment() const = 0;
    virtual const Specification& GetSpecification() const = 0;

    static Ref<Framebuffer> Create(const Specification& spec);

public:
    struct Specification
    {
        unsigned int Width, Height;
        unsigned int MSAASamples;
        bool SwapChainTarget;
    };
};
```

Ukázka kódu 6.8 Rozhraní třídy Framebuffer a její pomocná struktura

6.3 Datové struktury nezávislé na renderovacím API

Obecné datové struktury nezávislé na konkrétním renderovacím API nejsou designované jako abstraktní rozhraní, nýbrž jako plnohodnotné třídy. Následující třídy jsou právě takto realizované.

Light

Třída *Light* (ukázka kódu 6.9) je datová třída založena na kompozici. Různé shadery mohou světlo požadovat v různém formátu, tedy zde je navrženo jako soubor dat, kde uložená data mají přidělené jméno pro předání do shaderu. Nemá žádnou předdefinovanou strukturu dat, je tedy na uživateli, jaká data jsou relevantní pro světla v různých shaderech.

K vytvoření instance světla je nutné předat referenci na shader, ve kterém bude použita. Počítá s tím, že v shaderu bude vytvořena struktura pro různé typy světel. Data světla jsou uložena do *std::unordered_map*, kde jako indexy slouží jména dat k jejich hodnotám. Tato data mohou nabývat hodnot definovaných v *std::variant* jménem *UniformVariant*. *UniformVariant* slouží k tomu, aby v jednom datovém kontejneru mohly být uloženy různé datové typy.

Pro jednotlivá data je možno nastavit detaily pro vykreslení v GUI. Mezi tyto detaily patří minimální a maximální hodnota, velikost kroku mezi jednotlivými hodnotami a jestli daná data reprezentují barvu. Tyto detaily jsou pak využity při jejich vykreslení knihovnou Dear ImGui, pomocí které je vytvořeno grafické rozhraní aplikace. Uživatel tak může upravovat hodnoty dat světla přímo za běhu aplikace.

Data se do shaderu předají v momentu zavolání metody *PassData* a do GUI se vykreslí po zavolání funkce *OnImGuiRender*.

```
class Light
{
public:
    Light(const Ref<Shader>& shader);
    ~Light() = default;

    const Shader::UniformVariant& GetData(const std::string& name);
    Light& SetData(const std::string& name,
                   const Shader::UniformVariant& value,
                   const ImGuiDataDetail& imguiDetail
                       );

    void PassData();
    void OnImGuiRender();

private:
    Ref<Shader> m_Shader;
    std::string m_InShaderName;

    std::unordered_map<std::string, Shader::UniformVariant> m_Data;
    std::unordered_map<std::string, ImGuiDataDetail> m_ImGuiDetails;
};
```

Ukázka kódu 6.9 Třída pro reprezentaci světla

Material

Obecná datová třída *Material* (ukázka kódu 6.10) je založena na kompozici. Je velmi podobná jako třída pro světlo, jen slouží k ukládání dat o materiálu. Na rozdíl od světla se v ní pracuje s texturami. Jeden materiál může mít mnoho textur, které jsou zde explicitně odděleny od ostatních dat.

K vytvoření instance materiálu je nutno předat referenci na shader, ke kterému materiál patří. Následně je možno nahrávat data a textury, které předaný shader od materiálu potřebuje. Pokud nějaká data nejsou uživatelem poskytnuta, třída materiálu zahlásí varování do logovacího systému.

K datům je možno také nahrát jejich GUI detaily pro následné vykreslení pomocí knihovny ImGui. To funguje naprosto stejně jako u třídy *Light*.

Data se do shaderu předají v momentu zavolání metody *PassData* a do GUI se vykreslí po zavolání funkce *OnImGuiRender*.

```
class Material
{
public:
    Material(const Ref<Shader>& shader);
    ~Material() = default;

    Material& SetData(const std::string& name,
                    const Shader::UniformVariant& value,
                    const ImGuiDataDetail& imguiDetail
                        );
    Material& SetTexture(const std::string& name,
                        const Ref<Texture>& texture
                            );

    void SetTextureSlotRange(const glm::uvec2& range);
    void UpdateTextureSlotRangeStart(unsigned int newStart);

    const Shader::UniformVariant& GetData(const std::string& name);
    Ref<Texture> GetTexture(const std::string& name);

    void PassData();
    void OnImGuiRender();
private:
    Ref<Shader> m_Shader;
    std::string m_InShaderMaterialName;

    std::unordered_map<std::string, Shader::UniformVariant> m_Data;
    std::unordered_map<std::string, ImGuiDataDetail> m_ImGuiDetails;

    std::unordered_map<std::string, Ref<Texture>> m_Textures;
    glm::uvec2 m_TextureSlotRange;
};
```

Ukázka kódu 6.10 Třída pro reprezentaci materiálu

Transform

Datová třída *Transform* (ukázka kódu 6.11) slouží k reprezentaci pozice, orientace a škály objektu. Z těchto informací se počítá modelová transformace, tedy transformace, která z počátku při výchozí orientaci do kladné poloosy **z** a jednotkovou škálou přemístí objekt do dané pozice, natočí ho do dané orientace a přeškáluje danými škálovými faktory v jednotlivých osách.

Modelová transformace a její inverze jsou uloženy v maticích 4 x 4, které se přepočítávají pouze po změně některé z informací. Jako dodatečné informace je možné dostat přední vektor, pravý vektor a vektor vzhůru vzhledem k orientaci.

```
class Transform
{
public:
    Transform();
    ~Transform() = default;

    glm::vec3 GetForwardVector() const;
    glm::vec3 GetUpVector() const;
    glm::vec3 GetRightVector() const;

    const glm::mat4& GetModelMatrix();
    const glm::mat4& GetInverseModelMatrix();

private:
    void RecalculateCached();

private:
    glm::vec3 m_Position;
    glm::vec3 m_Scale;
    glm::quat m_Orientation;

    glm::mat4 m_ModelMatrix, m_InverseModelMatrix;
    bool m_Changed;
};
```

Ukázka kódu 6.11 Datová třída Transform

Camera

Třída *Camera* (ukázka kódu 6.12) je základní třída, ze které jsou odvozeny perspektivní a ortografická kamera. Drží informaci o projekční a pohledové transformaci. Pohledová transformace se odvíjí od pozice a orientace kamery a je pro oba odvozené typy kamer stejná a proto může být definována již v této základní třídě pomocí třídy *Transform*. Pohledová transformace je definována jako inverzní k modelové transformaci kamery. Projekční transformace je ovšem ovlivněna typem kamery, a proto v základní třídě není definován její výpočet.

Třída *PerspectiveCamera* a *OrthographicCamera* pak implementují tvorbu projekční matice na základě dalších parametrů, které uživatel již o konkrétním typu kamery dodá.

```
class Camera
{
public:
    Camera();
    Camera(const glm::mat4& projectionMatrix);
    virtual ~Camera() = default;

    const glm::mat4& GetProjectionMatrix() const;
    Transform& GetTransform();

protected:
    Transform m_Transform;
    glm::mat4 m_ProjectionMatrix;
};
```

Ukázka kódu 6.12 Ukázka základní třídy Camera

Mesh

Mesh (ukázka kódu 6.13) je třída, která spojuje dohromady informace o objektu. Obsahuje jeho geometrii ve formě třídy *VertexArray*, materiál asociovaný k objektu a shader, kterým se objekt vykresluje. Umístění, orientace a škála objektu jsou definovány pomocí instance třídy *Transform*.

K načítání objektu ze souboru slouží zpřátelená třída *MeshLoader*. Ta obsahuje statickou funkci, která načte objekt ze souboru (s pomocí knihovny assimp) a spojí ho s přiřazeným shaderem. Separace načítací logiky od samotné třídy *Mesh* je záměrná, jelikož třída *Mesh* by neměla obsahovat jakoukoliv interakci se soubory.

Podporován je pouze jeden materiál pro jeden objekt, ovšem do budoucna by bylo dobré umožnit použití více materiálů pro jeden objekt.

```
class Mesh
{
public:
    Mesh(const Ref<Shader>& shader, const Ref<VertexArray>& vao,
        const Transform& transform);
    ~Mesh() = default;

    void PassMaterialUniforms() const;
    void SetMaterial(const Ref<Material>& material);

private:
    Ref<VertexArray> m_VertexArray;
    Ref<Shader> m_Shader;
    Ref<Material> m_Material;

    Transform m_Transform;
};
```

Ukázka kódu 6.13 Ukázka třídy Mesh

6.4 Definice hlavních Renderer tříd

RendererAPI

Třída *RendererAPI* (ukázka kódu 6.14) je abstraktní rozhraní renderovacích API. Předpokládá se, že pro jednotlivá renderovací API bude toto rozhraní implementováno. Metody zde deklarované slouží jako přímé abstrakce funkcí renderovacích API.

Z jakéhokoliv místa v programu, kde je dostupná tato třída, je možné se dotázat na používané API pomocí statické funkce *GetAPI*, která vrací statickou instanci enumu API. Jediným dostupným API je OpenGL, které je implementováno ve třídě *OpenGLRendererAPI*. Do budoucna by se mohl přidat například Direct3D pro Windows, nebo Vulkan pro univerzálnější použití na více platformách.

Renderer API specifické funkce, které jsou v rozhraní implementovány, jsou *SetClearColor*, *Clear* a *SetDepthFunction*. Pro vykreslování slouží metody *DrawElements* a *DrawArrays*, které na vstupu očekávají geometrii ve formě třídy *VertexArray*.

```
class RendererAPI
{
public:
    enum class API
    {
        None = 0, OpenGL
    };

public:
    virtual void Init() = 0;

    virtual void SetClearColor(const glm::vec4& color) = 0;
    virtual void Clear() = 0;
    virtual void SetDepthFunction(DepthFunction depthFunction) = 0;

    virtual void DrawElements(const Ref<VertexArray>& vertexArray) = 0;
    virtual void DrawArrays(const Ref<VertexArray>& vertexArray) = 0;

    static API GetAPI();

private:
    static API s_API;
};
```

Ukázka kódu 6.14 Rozhraní třídy RendererAPI

Renderer

Hlavním spojem s renderovacím systémem je třída *Renderer* (ukázka kódu 6.15). Všechny její metody jsou statické, tudíž není třeba vytvářet instanci této třídy k použití veřejných metod. Obsahuje také statickou instanci třídy *RenderingAPI*, která je vytvořena na základě specifikovaného renderovacího API a která zaštiťuje všechny funkce onoho API.

Základní metodou je metoda *Submit*, kterou uživatel dává najevo, že chce vykreslit objekt. Tato metoda přijme data o objektu, buď pouze geometrii, nebo celý objekt s materiálem, předá je

do shaderu a zavolá vykreslení. Konkrétní implementaci vykreslování zaštiťuje třída jednotlivých renderovacích API. Přístup ke statické instanci *RenderingAPI* by měl být dostupný z třídy *Renderer* pro volání dostupných metod definovaných na úrovni API.

Před začátkem renderingu by měla nastat inicializace scény (případně pouze získání informací o již inicializované scéně), která slouží jako určitý kontext pro rendering – jsou zde obsaženy informace o použité kameře a potenciálně dalších věcech vztažených konkrétně k jedné scéně. Uživatel může také dát informaci o ukončení scény.

Struktura použita pro reprezentaci dat scény je zde velmi strohá – v budoucnu by měla být definována separátní třída *Scene*.

```
class Renderer
{
public:
    static void Init();

    static void BeginScene(const Ref<PerspectiveCamera>& camera);
    static void EndScene();

    static void Submit(const Ref<Mesh>& mesh);
    static void Submit(const Ref<VertexArray>& vertexArray);
    static void Submit(const Ref<SkyBox>& skybox);

    static Ref<RendererAPI>& GetAPI();

private:
    static Ref<RendererAPI> s_RendererAPI;

private:
    struct SceneData
    {
        Ref<PerspectiveCamera> m_Camera;
        glm::mat4 m_ProjectionViewMatrix;
    };

    static Ref<SceneData> m_SceneData;
};
```

Ukázka kódu 6.15 Rozhraní třídy Renderer

Kapitola 7
Implementace aplikace BTF Visualizer

Samotný renderer je pouze součástí knihovny, pomocí které je vytvořena spustitelná aplikace, která demonstruje užití modelu BTF k renderování. Tuto aplikaci jsem nazval *BTF Visualizer*. Herní engine Crank nabízí tvorbu aplikací přes rozšíření jím poskytnuté třídy *Application* a definováním funkce *Crank::CreateApplication*. Tyto kroky jsou vidět v ukázce kódu 7.1. V kódu je vidět, že do aplikace bude přidána vrstva *BTFVisualizerLayer*, která definuje obsah aplikace.

```
class BTFVisualizer : public Crank::Application
{
public:
    BTFVisualizer()
    {
        PushLayer(new BTFVisualizerLayer());
    }

    ~BTFVisualizer() = default;
};

Crank::Application* Crank::CreateApplication()
{
    return new BTFVisualizer();
}
```

Ukázka kódu 7.1 Definice aplikace BTF Visualizer

7.1 Funkcionalita vrstvy BTFVisualizerLayer

Vrstva *BTFVisualizerLayer* je odvozená ze abstraktní třídy *Crank::Layer*, a musí tedy přepsat určité metody, které jsou deklarované v této třídě. Jsou jimi metoda *OnAttach*, *OnUpdate*, *OnImGuiRender* a *OnEvent*.

Metoda *OnAttach* je inicializační částí aplikace. Je spuštěna po připojení vrstvy do aplikace. Konkrétně vytváří framebuffer, do kterého se celá scéna renderuje. Součástí inicializace je

také tvorba kamery, která je obalena v kontroléru, který definuje její pohyb na základě vstupů uživatele. Dále se zde načítají tři BTF materiály – travnaté plochy, látky z UTIA BTF databáze a testovací materiál. Ty jsou uloženy do datové struktury *TextureAtlas*, která slouží k načítání BTF materiálů reprezentovaných atlasy textur. Třída *TextureAtlas* je pouze takzvanou *wrapper* třídou pro dvou-dimenzionální texturu, která navíc řeší načtení metadat atlasu. Součástí materiálu je jak atlas textur, tak soubor s metadaty, které obsahují informace ohledně atlasu (počet textur v řádku a sloupci, velikost jediné textury, první sférickou souřadnici a zda-li je materiál z UTIA BTF databáze). Součástí inicializací je i vytvoření dvou demonstračních scén (a jedné výchozí), do kterých jsou předány BTF materiály jako reference spolu s kamerou.

V metodě *OnUpdate* probíhá volání renderovacích metod. Na začátku je zvolen framebuffer, do kterého bude scéna vykreslena. Tento výběr je na konci metody zrušen. Po výběru framebufferu následuje aktualizace kamery, která zahrnuje výpočet změny pozice kamery a reflektuje změnu velikosti okna. Následuje inicializace scény přes rozhraní třídy *Renderer* a vykreslení objektů zvolené demonstrační scény. Poté je scéna ukončena. Tato metoda je volána v nekonečné smyčce, dokud uživatel neukončí aplikaci.

Metoda *OnImGuiRender* vykresluje grafické rozhraní aplikace pomocí knihovny Dear ImGui. Knihovna ImGui umožňuje jednotlivá okna vykreslovat v prostření *Dockspace*. Jednotlivá okna tak mohou být oddělena, nebo vložena vedle sebe. Vykreslovaná OpenGL scéna je zobrazovaná pomocí textury získané z framebufferu jako obrázek v GUI nazývaný *viewport*. Vedle viewportu je možnost zobrazit debugovací data (počet snímků za vteřinu, dobu vykreslování jednoho snímku, parametry světla, parametry materiálu, ...). Vykreslována je také hlavní lišta v horní části okna. Zde si uživatel může zvolit libovolnou demonstrační scénu. Podobně jako *OnUpdate* běží v nekonečné smyčce, dokud uživatel neukončí aplikaci.

OnEvent je metoda, která je zavolána pouze pokud se stane nějaká událost. Ta je pak argumentem této metody a je možno ji zpracovat v rámci systému událostí. Zde konkrétně je událost předána pouze do kontroléru kamery, který na základě vstupů uživatele pohybuje s kamerou ve scéně.

7.2 Demonstrační scény

Uživatel si může vybírat mezi celkově čtyřmi demonstračními scénami. První je výchozí scéna, která nezobrazuje BTF materiály, ale pouze ukázku použití BRDF modelu, který je typický pro konvenční herní enginy. Ve druhé scéně je uživateli BTF materiál prezentován na povrchu čtvercové plochy a ve třetí je materiál aplikován na objekt s komplexní geometrií. Čtvrtá scéna slouží pro zobrazování uživatelem nahraného modelu. Mezi přiloženými soubory je možno ve složce assets/models nahradit soubor *custom.obj* libovolným modelem ve formátu *obj*. Tento model by měl mít přiložené informace o normálách a texturovacích souřadnicích.

Uživatel si v GUI může zobrazit debugovací informace, ve kterých jsou vidět detaily běhu aplikace a také parametry materiálu, které jsou předávané do shaderu. Tyto parametry zde může i měnit. Dále si může vybrat, zda-li chce materiál zobrazovat s bilineární filtrací, nebo bez. Další možností je použití jednoduchého pohledového vektoru. Ten se po zaškrtnutí v shaderu počítá z počátku scény namísto z pozice na povrchu objektu (detailněji popsáno v sekci o implementaci BTF fragment shaderu). Poslední možností je použití filmového mapování tónů.

Každá scéna má příkazy pro vykreslení objektu s materiálem v metodě *OnUpdate*, která je spouštěna z metody *OnUpdate* ve vrstvě *BTFVisualizerLayer*. Tato metoda pro jednotlivé demonstrační scény BTF materiálu zvolí BTF shader, předá mu potřebné parametry a pošle demonstrovaný objekt na vykreslení.

V každé scéně je fixně daný směr světla, jelikož tento prototyp nepočítá s proměnlivou pozicí světla. Pro výpočet vizuálů tedy stačí pouze jeden atlas textur, což výrazně zmenší paměťové nároky BTF materiálu.

7.3 Implementace BTF shaderu

Celý program BTF shaderu se zkládá ze dvou částí – vertex a fragment shader. Vertex shader pracuje s vertexy objektu, kdežto fragment shader pracuje s fragmenty. Ve fragment shaderu je definován výpočet barvy na základě modelu BTF. Počítá se zjednodušenou verzí modelu BTF, jelikož pracuje pouze s jedním atlasem pro fixně daný směr světla. Do budoucna by bylo možné dodefinovat sekvenci atlasů (například pomocí tří-dimenzionální textury) a výběr správného atlasu pro daný směr osvětlení, aby bylo možné renderovat plnohodnotné BTF.

7.3.1 Vertex shader

Vertex shader je poměrně krátký a plní standardní roli transformace vertexů pomocí matice PVM. Matice PVM reprezentuje transformaci mezi souřadnicovými systémy – z modelového do světového prostoru, následně do pohledového prostoru a nakonec do kanonického pohledového objemu.

Matice PVM je do vertex shaderu předána jako uniform. Uniformy jsou speciální proměnné, které jsou předány z aplikační části a jsou stejné pro všechny běhy shaderu. Tyto proměnné slouží pouze ke čtení a mohou být použité jak ve vertex shaderu, tak i ve fragment shaderu.

Dále předává hodnoty světové pozice, světové normály a texturovací souřadnice do fragment shaderu, kde jsou tyto hodnoty interpolovány pro jednotlivé fragmenty. Jeho podoba je vidět v ukázce kódu 7.2.

```
in vec3 a_Position;
in vec3 a_Normal;
in vec2 a_TexCoord;

uniform mat4 u_PVM;
uniform mat4 u_ModelMatrix;

out vec3 v_PositionWorld;
out vec3 v_NormalWorld;
out vec2 v_TexCoord;

void main()
{
    v_PositionWorld = (u_ModelMatrix * vec4(a_Position, 1.0)).xyz;
    v_NormalWorld = (u_ModelMatrix * vec4(a_Normal, 1.0)).xyz;
    v_TexCoord = a_TexCoord;
    gl_Position = u_PVM * vec4(a_Position, 1.0);
}
```

Ukázka kódu 7.2 Vertex shader pro BTF materiály

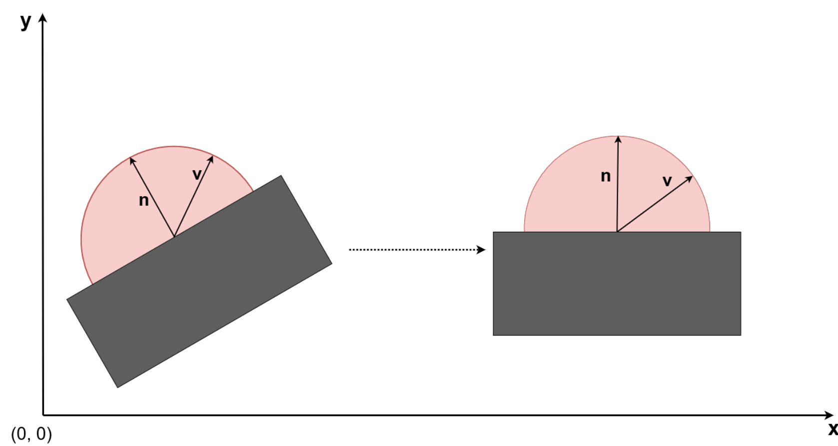
7.3.2 Fragment shader

Složitější částí celkového BTF shaderu je fragment shader. Ten se skládá z vícero funkcí, které slouží k vzorkování BTF materiálu a jeho následné zobrazení na obrazovku.

Hlavní část

Hlavní částí je funkce *main* spolu s uniformy a proměnnými z vertex shaderu. V ukázce kódu 7.4 ve funkci *main* je nejdříve určen pohledový vektor *view* buď jako vektor od počátku $(0, 0, 0)$ do pozice kamery, nebo jako vektor od pozice fragmentu do pozice kamery. Uživatel si může v GUI vybrat mezi těmito způsoby pomocí přepínače *Simple View Vec*, který je pak jako uniform předán do shaderu.

Následně proběhne rotace pohledového vektoru. Tato rotace odpovídá rotaci normálového vektoru přejatého z vertex shaderu tak, aby po rotaci byl normálový vektor shodný s vektorem (0, 1, 0) – jednotkovým vektorem vzhůru ve světovém prostoru. Nejdříve je získána osa rotace pomocí vektorového součinu normálového vektoru a vektoru (0, 1, 0). Kosinus úhlu, o který musí být vektor otočen, je roven skalárnímu součinu vektoru (0, 1, 0) a normálového vektoru. Následně je rotace o tento úhel kolem získané osy aplikována pomocí Rodriguesova vzorce [17] pro rotaci. Ta efektivně přetransformuje (znázorněno na obrázku 7.1) vektor pohledu do prostoru, kde normálový vektor je shodný s vektorem vzhůru. Normálový vektor tedy určuje orientaci polokoule, vůči které jsou definovány směry osvětlení a směry pohledu BTF modelu.



Obrázek 7.1 Ukázka transformace pohledového vektoru podle normály ve světovém prostoru

Správně orientovaný vektor pohledu je převeden do jednotkových sférických souřadnic a v této formě předán do funkce, která vzorkuje BTF materiál. Tato funkce vrátí barvu pro fragment, která je poté upravena pomocí filmového mapování tónů. Úprava barvy pomocí filmového mapování tónu probíhá dle následujícího vzorce:

$$\text{barva}_{\text{film}} = \frac{\text{barva} \cdot (6,2 \cdot \text{barva} + 0,5)}{\text{barva} \cdot (6,2 \cdot \text{barva} + 1,7) + 0,06}.$$

Barva je zde chápána jako trojice (R, G, B) hodnot, kde každý kanál může nabývat hodnot z intervalu $\langle 0, 1 \rangle$. Před použitím vzorce pro výpočet tónově namapované barvy je nutné barvu upravit pomocí vztahu

$$\text{barva} = \max((0, 0, 0),\ \text{barva} - 0,004).$$

S tímto postupem přišli J. Hejl a Richard Burgess-Dawson a ve formě kódu je dostupný z blogu Filmic Tonemapping Operators [18] od autora John Hable.

BTF materiál je uložen v datové struktuře *BTFMaterial*, která je ve fragment shaderu definována jako v ukázce kódu 7.3. Byť některé hodnoty v ní je možné dopočítat z jiných (například $single_tex_res \cdot tex_cnt = atlas_res$), tak z důvodu ušetření těchto operací jsem se rozhodl předávat přes uniformy všechny hodnoty.

```
struct BTFMaterial
{
    sampler2D   atlas;              // samotná textura atlasu
    uvec2       atlas_res;          // rozlišení textury atlasu
    uvec2       tex_cnt;            // počet textur v jednotlivých směrech
    uvec2       single_tex_res;     // rozlišení jedné textury v atlasu
    vec2        sph_coord_offset;   // odsazení sférických souřadnic textur
    bool        is_utia;            // zda-li je materiál z databáze UTIA BTF
};
```

Ukázka kódu 7.3 Struktura pro reprezentaci BTF materiálu

```
in vec3 v_PositionWorld;
in vec3 v_NormalWorld;
in vec2 v_TexCoord;

uniform vec3 u_CameraPosition;
uniform BTFMaterial um_Material;

uniform bool u_Bilinear;
uniform bool u_SimpleViewVec;

out vec4 fragColor;

void main()
{
    vec3 view;
    if (u_SimpleViewVec)
        view = normalize(u_CameraPosition); // ignoruj pozici fragmentu
    else
        view = normalize(u_CameraPosition - v_PositionWorld);

    vec3 n = normalize(v_NormalWorld);
    vec3 axis = cross(n, vec3(0.0, 1.0, 0.0));
    float cos_theta = dot(vec3(0.0, 1.0, 0.0), n);

    // Rodriguesův vzorec pro rotaci
    vec3 view_rot = view * cos_theta + cross(axis, view) * sin(acos(cos_theta))
                    + axis * dot(axis, view) * (1.0 - cos_theta);
    // (azimut, elevace)
    vec2 view_sph_coord = CartesianToSpherical(view_rot);

    vec3 color = SampleBTF(
        view_sph_coord, v_TexCoord, um_Material, u_Bilinear
            );
    color = FilmicToneMapping(color);
    fragColor = vec4(color, 1.0);
}
```

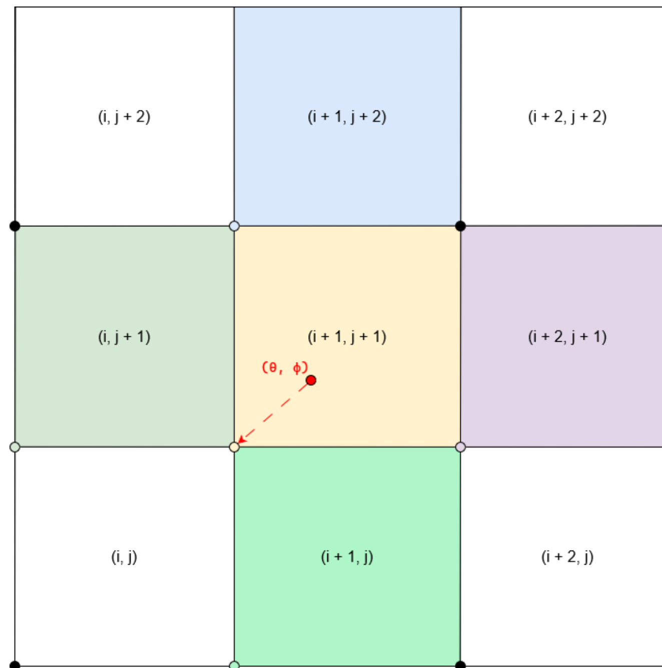
Ukázka kódu 7.4 Fragment shader pro BTF materiály

Jednoduché vzorkování atlasu vlastního BTF materiálu

Atlas textur pro vlastní BTF materiál byl designován, aby vzorkování z něj bylo co nejvíc přímočaré. Pokud tedy na vstupu jsou zadány texturovací souřadnice, sférické souřadnice směru pohledu a BTF materiál, probíhá jednoduché vzorkování pomocí funkce z ukázky kódu 7.5. Tato funkce vypočítá přesné texturovací souřadnice v atlasu textur a vrátí navzorkovanou barvu na těchto souřadnicích. Nejprve získá index nejbližší textury, ze které má v atlasu proběhnout vzorkování (obrázek 7.2). K tomu přidá informaci o texturovacích souřadnicích z parametrů funkce, které určí přesnou pozici v jedné textuře z atlasu, a provede vzorkování. Na výsledném vykresleném povrchu jsou pak vidět ostré hrany mezi jednotlivými texturami. Pokud je úhel elevace směru pohledu vyšší než největší elevace z měření materiálu, je vrácena černá barva.

```
vec3 SampleBTF_Simple(vec2 view_sph_coord, vec2 tex_coord, BTFMaterial mat)
{
    vec2 sph_ratio = view_sph_coord / mat.sph_coord_offset;
    uvec2 start_sample_idx = round(sph_ratio.x), round(sph_ratio.y);

    // elevace je mimo hranice materiálu
    if (start_sample_idx.y >= mat.tex_cnt.y)
        return vec3(0.0, 0.0, 0.0);

    // resetuj azimut (nastane po zaokrouhlení azimutu na 360 stupňů)
    if (start_sample_idx.x >= mat.tex_cnt.x)
        start_sample_idx.x = 0;

    uvec2 start_sample_px = start_sample_idx * mat.single_tex_res;
    vec2 uv_scaled =
         (tex_coord * mat.single_tex_res + start_sample_px) / mat.atlas_res;

    return texture(mat.atlas, uv_scaled).xyz;
}
```

Ukázka kódu 7.5 Jednoduché vzorkování vlastního BTF materiálu



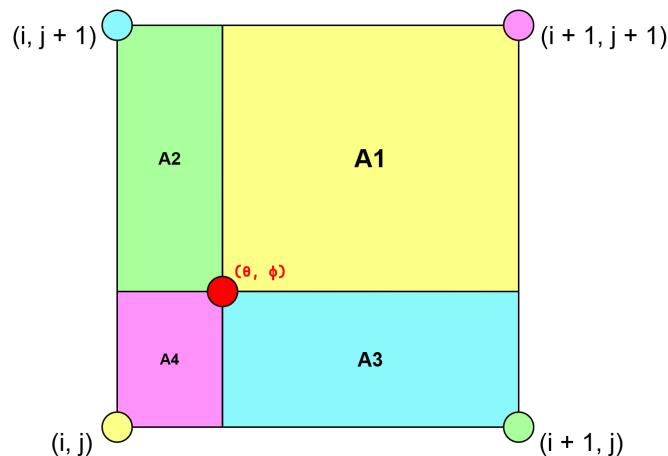
Obrázek 7.2 Jednoduché vzorkování vlastního BTF materiálu z atlasu textur. Ukázka obsahuje pouze sekci atlasu textur, kde jednotlivé textury jsou znázorněny čtverci s jejich indexy. Bod v levém dolním rohu je vždy začátkem textury. Dvojice (i, j) určuje index textury. Červený bod značí sférickou souřadnici směru pohledu, ze které se počítá index nejbližší textury v atlasu – zde je to textura na indexu (i + 1, j + 1). Jednotlivé textury odpovídají bodům na polokouli, proto je možné přímo přepočítat sférickou souřadnici na index textury

Bilineární vzorkování atlasu vlastního BTF materiálu

Bilineární vzorkování na rozdíl od jednoduchého vzorkování nepracuje s jednou nejbližší texturou, ale počítá vážený průměr ze čtyř krajních (obrázek 7.3). Jelikož je rozdělení bodů pro vlastní BTF materiál rovnoměrné, je možné pro libovolný směr pohledu určit čtyři body, které tvoří čtverec[1], kterým směr pohledu prochází. Pro výpočet barvy se použijí čtyři textury, které korespondují k vrcholům onoho čtverce.

V shaderu se naleznou čtyři krajní body – v kontextu vzorkování atlasu textur jsou těmito body indexy jednotlivých textur v atlasu. Pro každý z indexů se vypočítá váha dle obrázeku 7.3. Výsledná barva se následně získá jako součet čtyř jednoduchých vzorkování textur na indexech, kde barva jednotlivých vzorků je násobena váhou dané textury.

[1]Korektní by bylo místo čtverce uvážit čtverec promítnutý na polokulovou plochu tak, že jeho vrcholy se promítnou na body na polokouli, ze kterých byl snímán BTF materiál. V tomto textu je pro zjednodušení výpočtů tento kulový čtverec aproximován normálním čtvercem.



Obrázek 7.3 Ukázka počítání vah bilineárního vzorkování pro čtyři body. Bod, který se vzorkuje (červený bod), rozdělí čtverec krajních bodů na čtyři. Obsahy těchto menších čtverců tvoří váhy pro protější body (barevně znázorněno v obrázku – vždy jedna barva plochy a bodu).

Jednoduché vzorkování atlasu materiálu z UTIA BTF databáze

Vzorkování atlasu textur materiálu z UTIA BTF databáze je až na jeden krok stejný proces jako vzorkování vlastního BTF materiálu. To platí jak pro jednoduché, tak pro bilineární vzorkování. Jednoduché vzorkování spočívá v tom, že se vypočítá index textury v atlasu, která je nejblíže směru pohledu. Dále se přidá informace z texturovacích souřadnic pro získání vzorku z konkrétní textury v atlasu.

Rozdílem však je získání indexu nejbližší textury. Funkce, která na základě směru pohledu vypočítá index nejbližší textury, je vidět v ukázce kódu 7.6. Pro tuto funkci je dodefinováno pole, ve kterém na indexu elevační hladiny je uložen počet bodů na této hladině. Tyto hodnoty jsou použity spolu se směrem pohledu pro výpočet takzvaného rozvaleného indexu (*raveled index*). Ten odpovídá indexu v atlasu textur, který byl rozvalen ze dvou dimenzionální matice textur do jedno dimenzionálního pole textur. Jelikož je pevně dané, že atlas materiálů z UTIA BTF databáze má vždy rozměr devět na devět textur, je možné z rozvaleného indexu přímo vypočítat index textury.

Bilineární vzorkování atlasu materiálu z UTIA BTF databáze

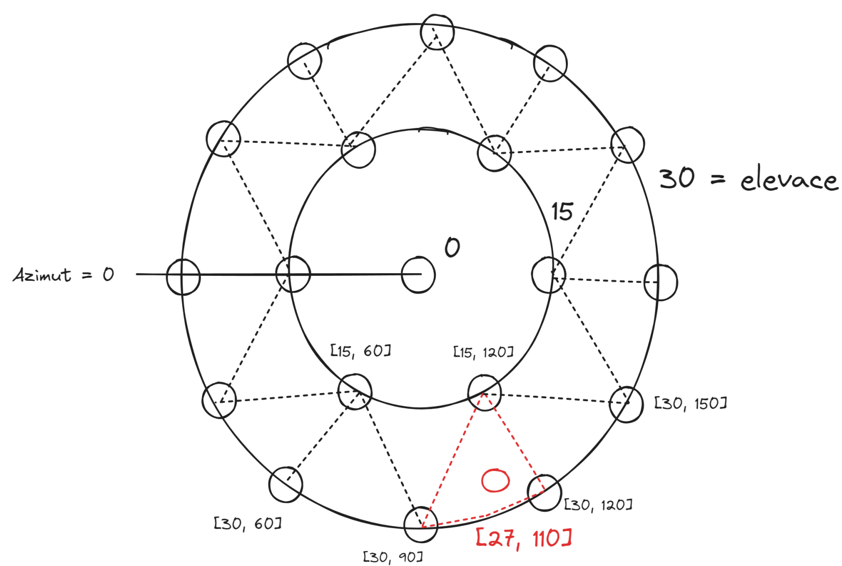
Rozdělení bodů při pořizování snímků materiálu z UTIA BTF databáze není rovnoměrné. To je nutné vzít v úvahu při bilineárním vzorkování takovéhoto materiálu. Není tedy optimální průměrovat výslednou barvu ze čtyřech sousedních bodů, ale ze tří (ukázka pro konkrétní případ na obrázku 7.4). Pro směr pohledu tvoří tři nejbližší body trojúhelník[2], který bod směru pohledu na polokouli rozděluje na tři trojúhelníky. Plochy těchto trojúhelníků slouží jako váhy k protějším bodům podobně jako u bilineárního vzorkování vlastního materiálu. Tyto váhy je ovšem ještě nutno znormalizovat, aby se sečetly do jedné.

Proces hledání nejbližších bodů ke směru pohledu je zde komplikovanější. Nejprve se z elevace směru pohledu naleznou dvě nejbližší elevační hladiny. Na obou se naleznou dva nejbližší body z pohledu azimutu. Celkově to jsou čtyři body a je tedy nutné bod s nejvzdálenějším azimutem od směru pohledu zahodit. Následně proběhne třikrát jednoduché vzorkování a výpočet vah pro jednotlivé vzorky. Vzorky se vynásobí váhami a sečtou, což vytvoří finální barvu, která je z funkce vrácena.

[2]Korektní by bylo místo trojúhelníku uvážit trojúhelník promítnutý na polokulovou plochu tak, že jeho vrcholy se promítnou na body na polokouli, ze kterých byl snímán UTIA BTF materiál. V tomto textu je pro zjednodušení výpočtů tento kulový trojúhelník aproximován normálním trojúhelníkem.

```
// sedmá hodnota pouze pro výpočty - žádná data nejsou na této hladině
const int UTIA_AZIMUTH_DIVISONS[7] = int[](1, 6, 12, 18, 20, 24, 30);

uvec2 GetUtiaAtlasStartSampleIdx(vec2 view_sph_coord)
{
    int elev_idx = round(view_sph_coord.y / radians(15.0));
    int azi_idx = round(view_sph_coord.x /
                        radians(360.0 / UTIA_AZIMUTH_DIVISONS[elev_idx])
                            );

    // resetuj azimut (nastane po zaokrouhlení azimutu na 360 stupňů)
    if (azi_idx == UTIA_AZIMUTH_DIVISONS[elev_idx])
        azi_idx = 0;

    int raveled_idx = 0;
    for (int i = 0; i < elev_idx; ++i)
    {
        raveled_idx += UTIA_AZIMUTH_DIVISONS[i];
    }
    raveled_idx += azi_idx;

    float raveled_ratio = raveled_idx / 9.0;

    uvec2 start_sample_idx = uvec2(0, 0);
    start_sample_idx.x = (raveled_ratio - floor(raveled_ratio)) * 9.0;
    start_sample_idx.y = floor(raveled_ratio);

    return start_sample_idx;
}
```

Ukázka kódu 7.6 Ukázka funkce pro výpočet indexu nejbližší textury v atlasu materiálu z UTIA BTF databáze



Obrázek 7.4 Nákres hledání bodů bilineární interpolace pro UTIA BTF materiály. Na obrázku je vidět nultá, první a druhá elevační hladina na polokouli při pohledu shora. Jsou zobrazeny jako soustředné kružnice, které mají vedle sebe napsaný jejich úhel elevace (0°, 15°a 30°). Naznačen je také nultý úhel azimutu. Mezi body první a druhé hladiny jsou naznačena jejich rozčlenění do trojúhelníků. Souřadnice bodů jsou jejich jednotkové sférické souřadnice ve stupních. Červený bod značí směr pohledu a jeho příslušný trojúhelník je také označen červeně. Pro bilineární vzorkování červeného bodu [27, 110] by se použily body [15, 120], [30, 90] a [30, 120]

7.4 Ovládání aplikace

Kamera může být ovládána jak klávesnicí, tak myší. Držení pravého tlačítka na myši odemkne možnost rotovat kamerou kolem objektu pomocí pohybu myši. Kolečkem myši je možno přiblížit nebo oddálit pohled. Na klávesnici se kamera ovládá pomocí tlačítek W, A, S a D, které rotují kamerou.

Jednotlivá okna mohou být přemístěna na libovolnou pozici na obrazovce nebo zadokována do sebe. Manipulace s okny probíhá přes jejich horní lištu.

V hlavní liště si uživatel v sekci *Demos* může zvolit demonstrační scénu. Výchozí scénou je *Plane Demo*, ovšem může být přepnuta na *Object* a *Custom Demo*. K dispozici je také PBR (Physically Based Rendering) scéna, která nezobrazuje BTF materiály, ale slouží jako vizuální reference. Ta se nazývá *Default PBR* a také je možno se na ní přepnout. V sekci *View* si uživatel může vybral, zda-li chce zobrazovat debugovací informace.

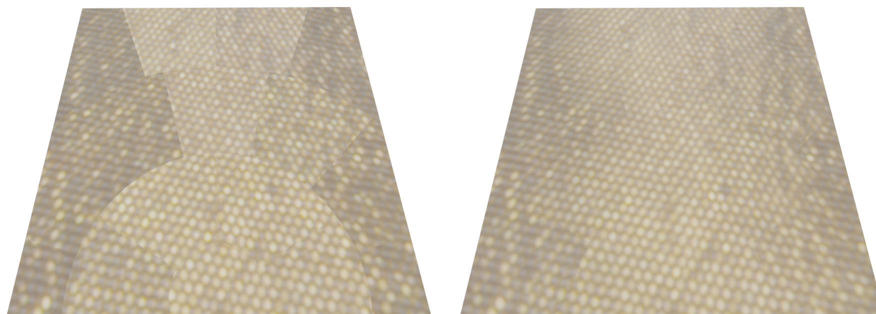
V demonstračních scénách je přes GUI možno v debugovacích informacích přímo interagovat s parametry materiálu, které jsou předávány do shaderu. Také je možno zvolit specifické možnosti pro vykreslování, jako zvolení jednoduchého výpočtu směru pohledu, zapnutí bilineárního vzorkování nebo použití filmového mapování tónů. Pomocí tlačítek se zde volí materiál, který je na daný objekt ve scéně aplikován. Na výběr je materiál trávy (tlačítko *Grass*), látky (tlačítko *Fabric*) a testovací materiál (tlačítko *Test*), který slouží pouze jako vizualizace testovacího atlasu textur.

Aplikace může být ukončena z hlavní lišty v záložce *File* pomocí tlačíka *Close*, nebo pomocí stisknutí klávesy *Esc* kdekoliv z aplikace.

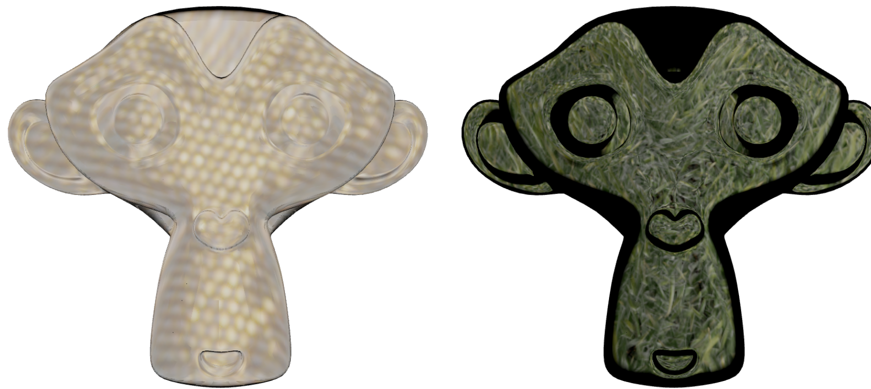
Kapitola 8
Výsledky a porovnání

Rendery BTF materiálu pomocí aplikace BTF Visualizer z demonstračních scén jsou ukázány na obrázcích níže. Ze scény čtvercové plochy jsou materiály travnatého povrchu a látky vidět na obrázcích 8.1 a 8.2. Demonstrační scéna, kde jsou materiály aplikovány na komplexní objekt, je zobrazena na obrázku 8.3.

Obrázek 8.1 BTF materiál travnatého povrchu s jednoduchým vzorkováním (vlevo) a bilineárním vzorkováním (vpravo) na čtvercové ploše. Oba snímky byly upraveny filmovým mapováním tónů



Obrázek 8.2 Materiál látky z databáze UTIA BTF s jednoduchým vzorkováním (vlevo) a bilineárním vzorkováním (vpravo) na čtvercové ploše. Oba snímky byly upraveny filmovým mapováním tónů



Obrázek 8.3 Materiály látky (vlevo) a travnatého povrchu (vpravo) s bilineárním vzorkováním na komplexním objektu. Oba snímky byly upraveny filmovým mapováním tónů

Na obrázku 8.1 je vidět, že bilineární interpolace při vzorkování travnatého materiálu není ideální. Jednotlivá stébla trávy se částečně slijí do sebe a výsledek celkově působí rozmazaně. Důvodem je tří-dimenzionální povaha travnatého povrchu – stébla trávy vystupují výrazně nad povrch. Bilineární interpolace pak není ideálním nástrojem pro vzorkování tohoto materiálu. Samotné měření tohoto materiálu tomu muselo být uzpůsobeno limitováním maximální elevace. Kvůli tomu je na demonstrační scéně s objektem (obrázek 8.3) vidět znatelně více černé než u materiálu látky.

U materiálu látky z databáze UTIA BTF naopak bilineární interpolace vyhlazuje ostré přechody mezi jednotlivými snímky měření. Látka se chová spíše jako dvou-dimenzionální povrch a její měření byla více předzpracována v rámci databáze UTIA BTF. Rozložení bodů pro měření tohoto materiálu je optimálnější a efektivnější, což výrazně zmenšuje paměťové nároky materiálu oproti travnatému povrchu (2 GB oproti 8 GB pro celé materiály).

Renderování těchto materiálů není výpočetně náročné – hlavní část výpočtu tvoří nalezení správných texturových souřadnic pro vzorkování atlasu textur. V demonstračních scénách, kde je aktivní vždy jeden materiál, se počet snímků za vteřinu drží stabilně okolo 1 500. Tyto demonstrační scény byly spouštěny na laptopu s procesorem Intel Core i5-10300H a grafickou kartou NVIDIA GeForce RTX 3060 Laptop. Nutno podotknout, že materiály v těchto scénách nejsou plné BTF materiály, ale pouze jejich zjednodušení pro jeden směr osvětlení. Problémem pro celé BTF materiály ovšem není doba výpočtů, ale jejich velikost. Pro materiál látky je celková velikost 2 GB. Použitím nižších jednotek materiálů je možno zaplnit celou paměť grafické karty a pro komplexní scény, kde mohou být použity až stovky materiálů, je praktické použití nemožné.

8.1 Porovnání s dostupnými real-time renderery

Dostupnost materiálových modelů ve vybraných aplikacích využívající real-time rendering je jednotlivě rozepsána v tabulce 8.1. Mezi aplikace, které se porovnávají s výstupem této práce (BTF Visualizer), jsem zařadil Blender (s využitím rendereru *Eevee*) a Unity. Blender není herním enginem, ovšem Eevee vykresluje snímky v reálném čase, a spadá tedy do kategorie real-time rendererů.

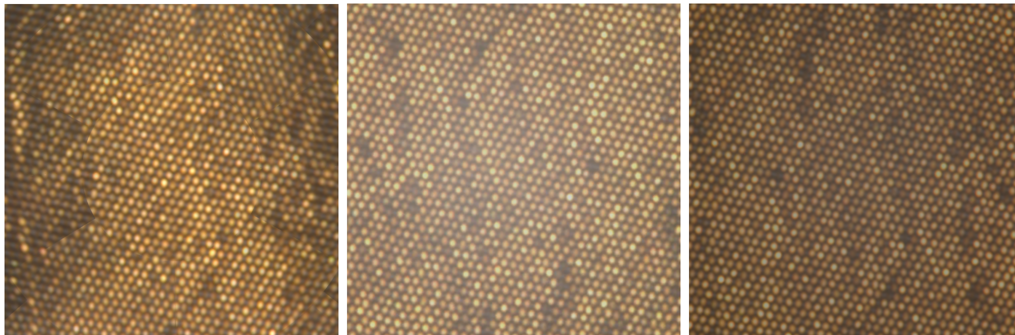
Pro Blender ani pro Unity není nativně dostupný materiálový model BTF. V obou případech je ovšem možné jej v jisté míře doplnit s použitím vlastních shaderů a reprezentace materiálu. Pro Blender existuje plugin [19] od autorů M. Hatky a M. Haindla, který umožňuje zobrazovat BTF materiály, ovšem byl vytvořen v roce 2011 a nelze jej použít v nejnovějším Blenderu 4.1. Místo něj jsem pracoval s výchozím nastavením rendereru Eevee. Jako výchozí materiálový model používá BSDF. V Unity se pak BSDF používá také ve dvou ze tří dostupných renderovacích

	BRDF	BSDF	BTF
BTF Visualizer	✓	×	✓
Blender (Eevee)	✓	✓	×
Unity	✓	✓	×

Tabulka 8.1 Tabulka nativní dostupnosti materiálových modelů v uvedených aplikacích zaměřených na real-time rendering

metodách – *high definition rendering pipeline* a *universal rendering pipeline*. Jelikož model BRDF je v jistém smyslu podmnožinou modelu BSDF, tak uvažuji, že pokud program pracuje s BSDF, pak automaticky umí pracovat s BRDF.

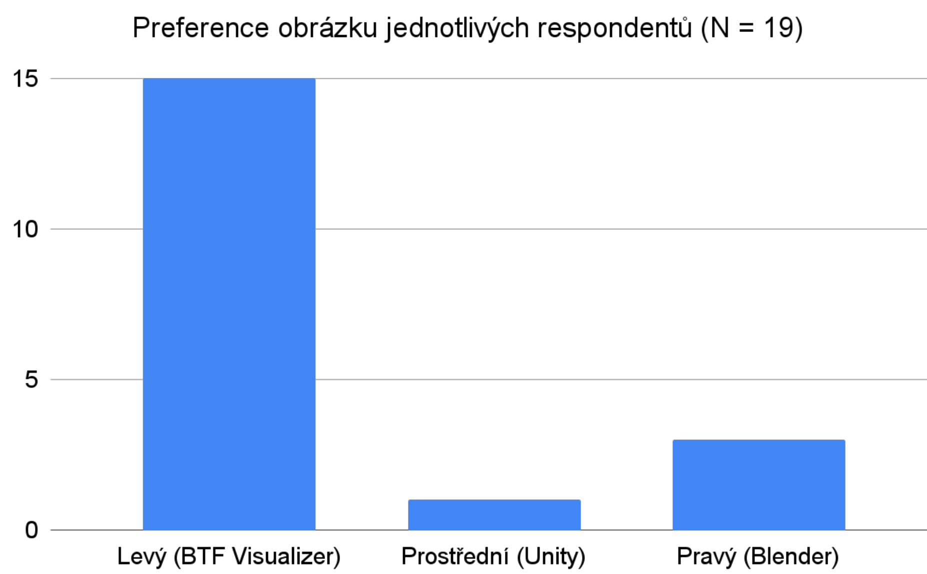
Na obrázku 8.4 je vidět porovnání renderů materiálu látky mezi aplikací BTF Visualizer, Unity a Blenderem. Tuto trojici obrázků jsem ukázal 19 respondentům a formou dotazníku jsem zkoumal, kterou ukázku vizuálně preferují nejvíce a z jakého důvodu.



Obrázek 8.4 Porovnání renderů materiálu látky mezi vybranými renderery. Ukázka z aplikace BTF Visualizer s použitím materiálového modelu BTF (vlevo) je porovnána s rendery z Unity (uprostřed) a Blenderu (vpravo). Povrch v Unity byl nasvícen směrovým světlem kolmo na povrch a použita byla textura odpovídající tomuto směru osvětlení a kolmému směru pohledu na povrch. Tato textura byla použita jako difuzní složka materiálu a zbytek parametrů zůstal s výchozími hodnotami. Stejná scéna byla zrekreována i v Blenderu a pomocí Eevee byla vyrenderována. Během vytváření snímků nedošlo k úpravě jejich barev pomocí mapování tónů

Výsledek kvantitativní části, tedy kolik respondentů označilo jaký obrázek za vizuálně nejlepší, je vidět v podobě grafu na obrázku 8.5. Celkem 15 z 19 celkem zúčastněných respondentů označilo obrázek z aplikace BTF Visualizer jako vizuálně nejlepší. Obrázek z Unity označil za preferovaný jeden respondent a obrázek z Blenderu preferovali 3 respondenti.

Vybrané zdůvodnění volby preferovaného obrázku jsou rozebrány v následujícím textu. Hromadný konsenzus byl, že obrázek z BTF Visualizeru nabízí nejlepší vyvážení světlosti mezi tmavším obrázkem z Blenderu a světlejším z Unity. Velká část respondentů uvedla, že se jim také líbil jeho kontrast barev. Dva respondenti zmínili fakt, že u tohoto obrázku jsou světlejší barvy (odlesky) koncentrovanější na středu, kdežto u zbylých dvou jsou distribuovány více rovnoměrně.



Obrázek 8.5 Graf odpovědí kvantitativní části testování. Ve sloupcích je znázorněn počet respondentů, kteří preferovali daný obrázek

Závěr

Cílem práce bylo zhodnotit praktičnost a výsledky použití pokročilého texturovacího modelu Bidirectional Texture Function (BTF) a jeho dynamické verze (DBTF) pro renderování v reálném čase. Ke splnění tohoto cíle byl vytvořen návrh renderovacího systému (rendereru), který byl v podobě prototypu implementován. Tento renderer byl využit jako základ pro aplikaci BTF Visualizer, která umožňuje vykreslovat materiály pomocí modelu BTF.

V této práci byl navrhnut postup vykreslování BTF materiálů, který byl částečně implementován v aplikaci BTF Visualizer. Jedná se o postup, ve kterém je BTF materiál uložen do atlasu textur, který je v aplikaci různými metodami vzorkován. Navrhnuta byla také optimalizace, která uvažuje použití sférické textury namísto atlasu textur pro reprezentaci BTF materiálu.

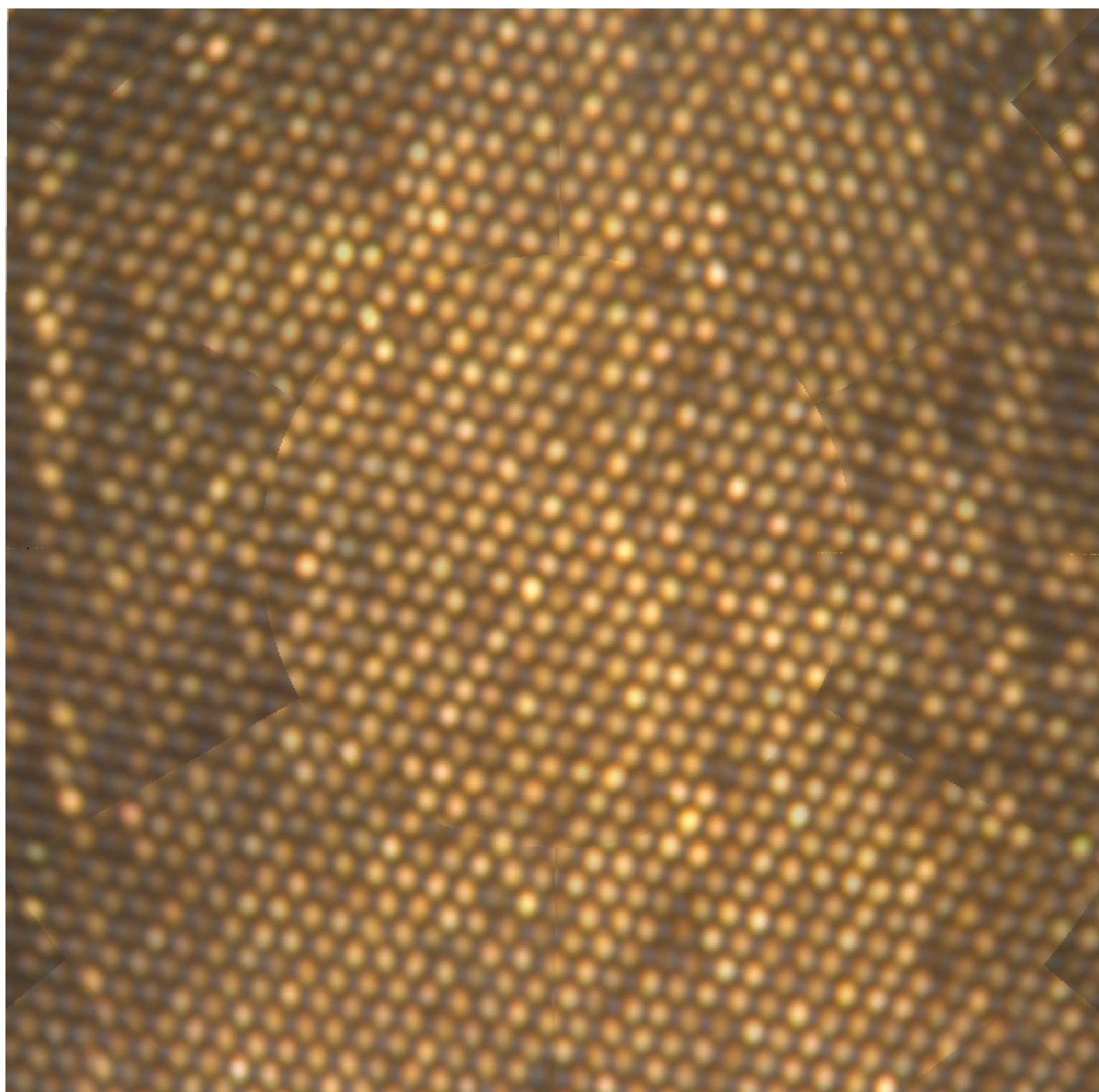
Renderování BTF materiálů je v aplikaci BTF Visualizer docíleno v reálném čase. Rychlost výpočtu zde není limitujícím faktorem. Problém vychází ze samotné podstaty BTF – k reprezentaci materiálu je nutné mít velký objem dat. Jeden materiál není problém načíst a zobrazit, ale práce s více materiály v jedné scéně rychle zaplní dostupnou paměť grafické karty. Model dynamického BTF čelí stejnému problému jako BTF, ovšem kvůli ještě většímu objemu dat nutných pro reprezentaci materiálu je v této práci označen za nepraktický pro použití v real-time renderingu.

V aplikaci BTF Visualizer je možné zobrazit dva BTF materiály. Jedním je vlastní materiál travnatého povrchu a druhým je detail povrchu látky z databáze UTIA BTF. Je možné pozorovat rozdílné formáty a vzorkování těchto materiálů. K dispozici jsou tři demonstrační scény, ve kterých jsou materiály aplikovány na různé objekty. Do jedné demonstrační scény je možné nahrát vlastní 3D model a zobrazit ho s jedním z dostupných materiálů.

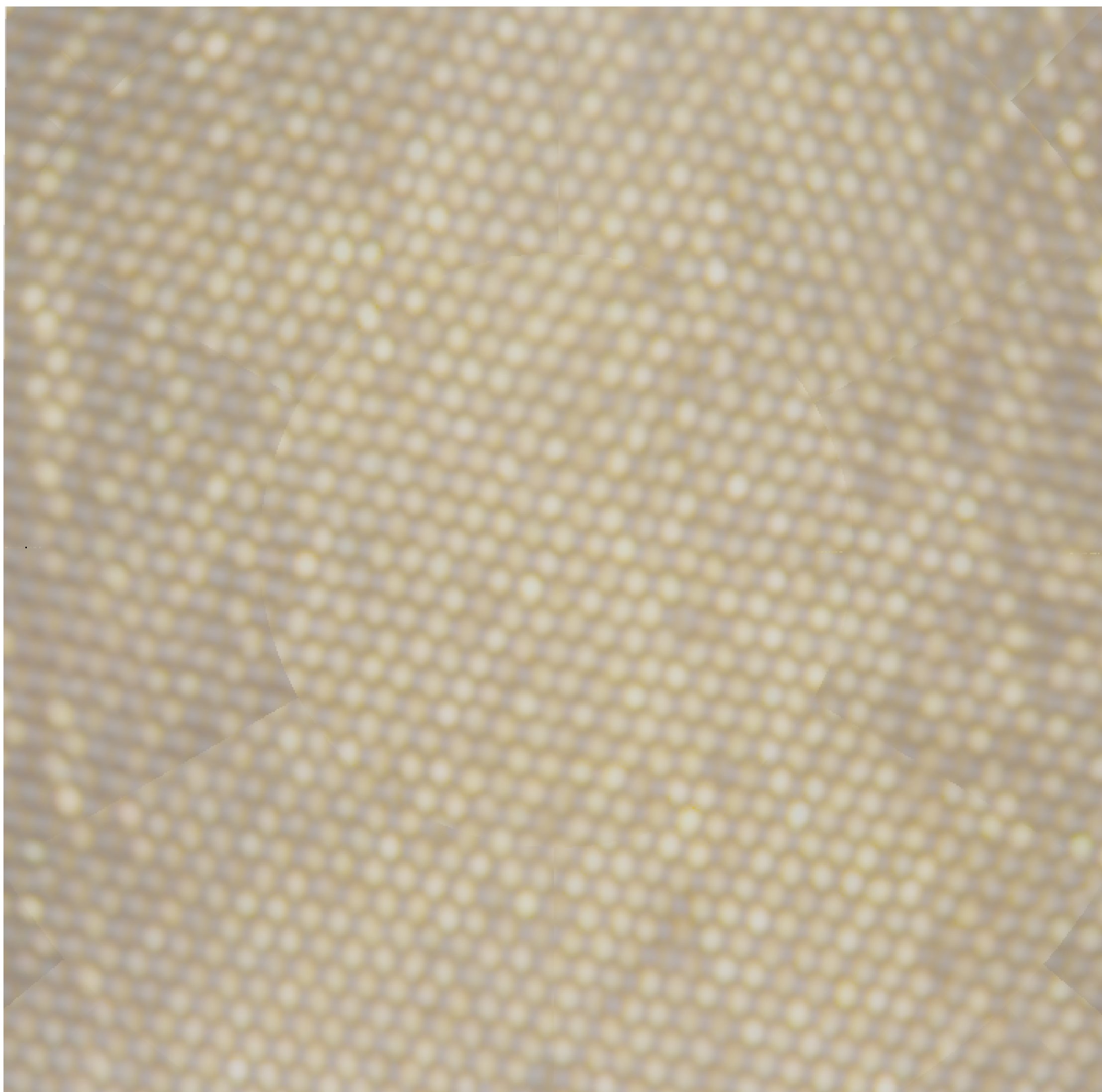
Ukázku vykresleného materiálu látky jsem v rámci testování poslal formou dotazníku skupině respondentů. Spolu s ní jim byly předloženy ukázky stejného materiálu vykresleny nativně dostupnými metodami ve vybraných real-time rendering aplikacích – Blenderu (s použitím rendereru Eevee) a Unity. Většina z respondentů vizuálně preferovala ukázku materiálu BTF před ostatními možnostmi. Limitem této části výzkumu je, že některým respondentům se mohly ukázky zobrazovat odlišně (podle typu displeje zařízení, aplikace či vlivem okolních podmínek).

Příloha A

Ukázky z aplikace BTF Visualizer



Obrázek A.1 Ukázka materiálu látky na čtvercové ploše při kolmém pohledu shora. Pořízena bilineárním vzorkováním bez filmového mapování tónů

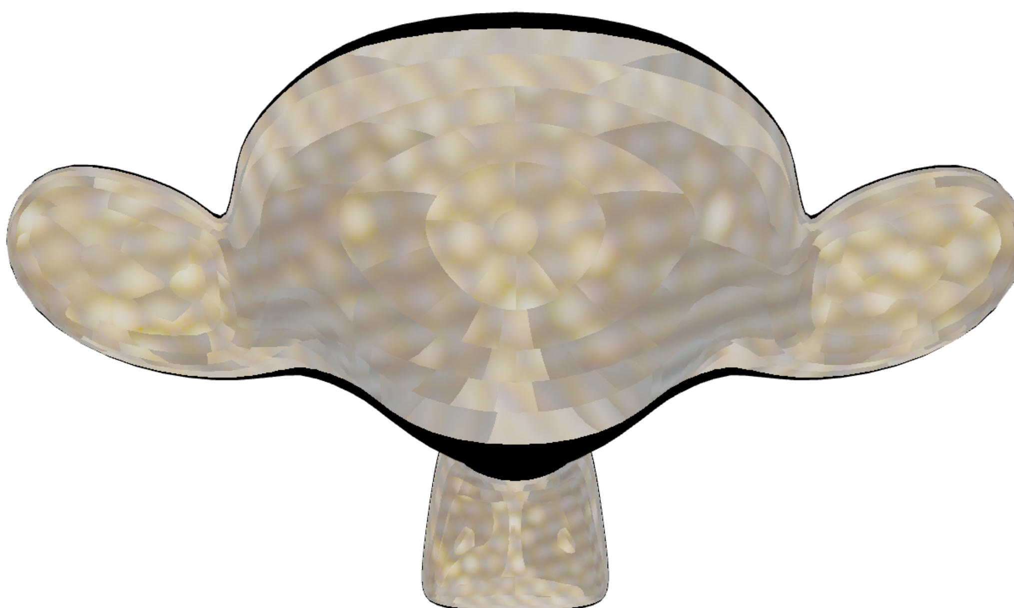


Obrázek A.2 Ukázka materiálu látky na čtvercové ploše při kolmém pohledu shora. Pořízena bilineárním vzorkováním s využitím filmového mapování tónů

Obrázek A.3 Ukázka materiálu travnatého povrchu na čtvercové ploše pozorované pod úhlem. Pořízena bilineárním vzorkováním s využitím filmového mapování tónů. Demonstruje plynulý přechod do černé u horizontu

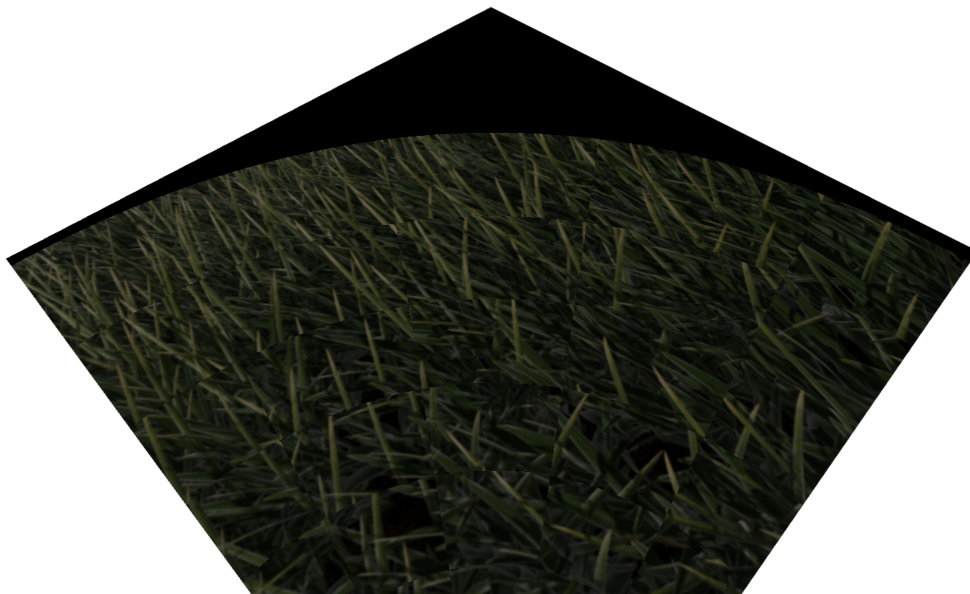
Obrázek A.4 Ukázka materiálu travnatého povrchu bez bilineárního vzorkování na komplexním objektu. Pořízeno s využitím filmového mapování tónů

Obrázek A.5 Ukázka materiálu travnatého povrchu bez bilineárního vzorkování z opačné strany komplexního objektu. Pořízeno s využitím filmového mapování tónů

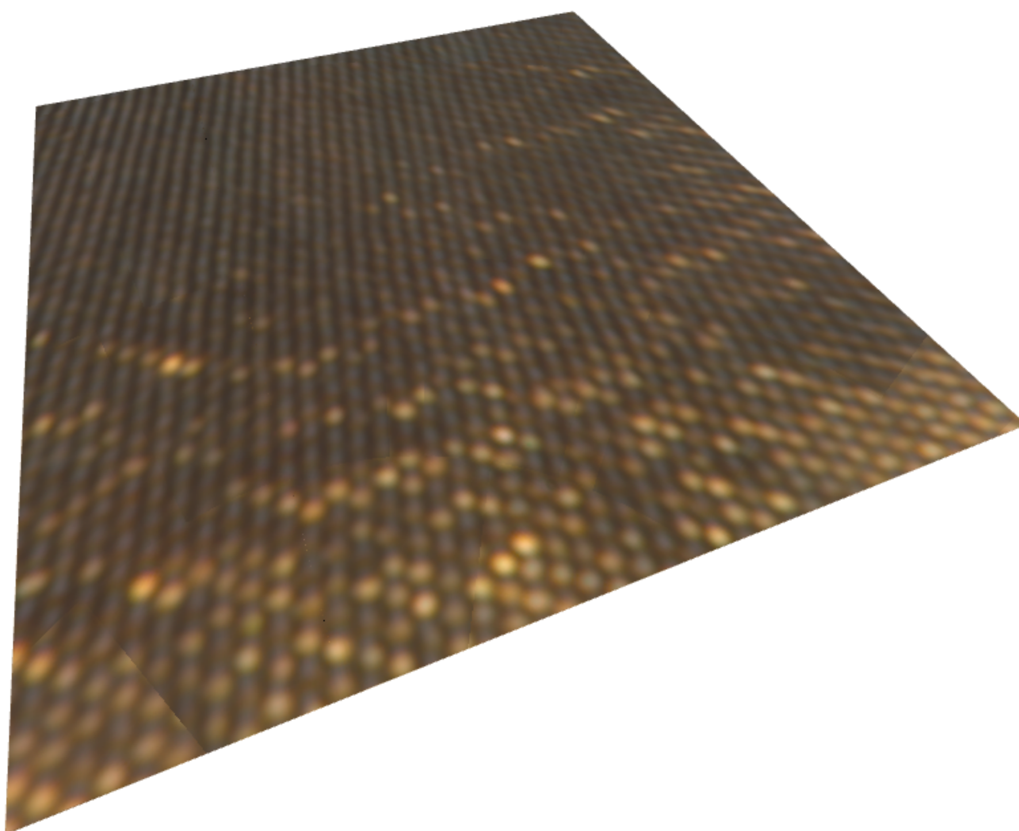


Obrázek A.6 Ukázka materiálu látky bez bilineárního vzorkování z opačné strany komplexního objektu. Pořízeno s využitím filmového mapování tónů

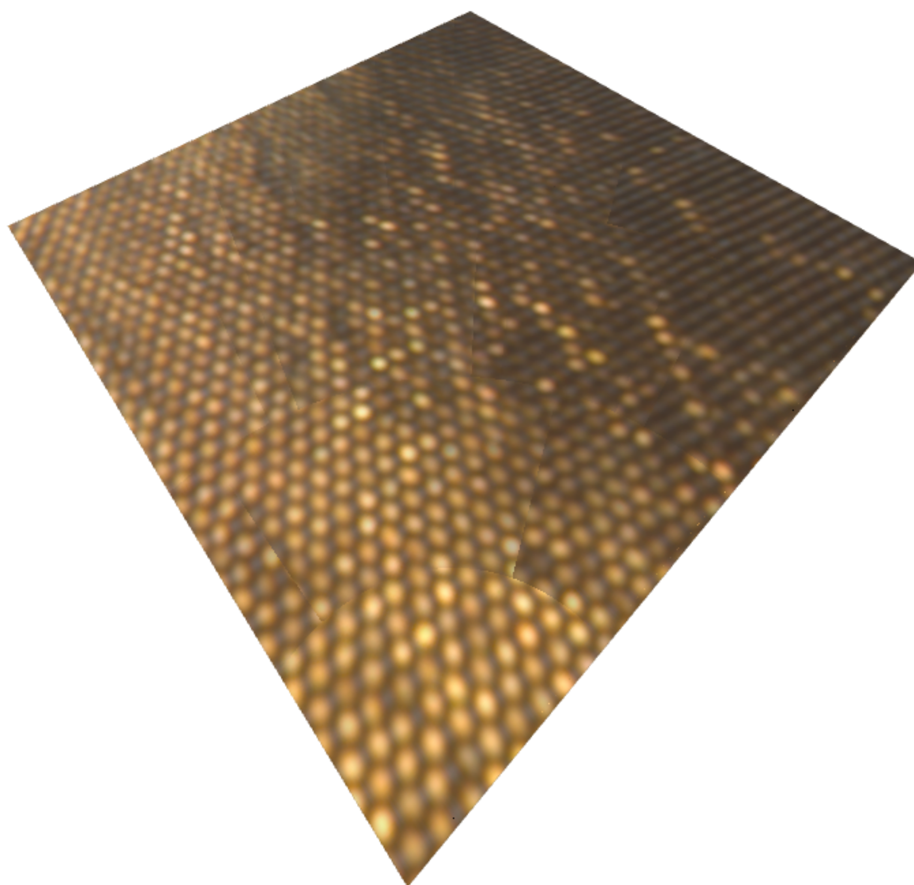
Obrázek A.7 Ukázka materiálu látky s využitím bilineárního vzorkování na modelu torusu. Pořízeno s využitím filmového mapování tónů



Obrázek A.8 Ukázka materiálu travnatého povrch bez využití filmového mapování tónů. Je vidět, že materiál je takto výrazně tmavší. Pořízen pomocí jednoduchého vzorkování



Obrázek A.9 Ukázka materiálu látky s využitím bilineárního vzorkování na čtvercové ploše pod úhlem. Pořízeno bez využití filmového mapování tónů



Obrázek A.10 Ukázka materiálu látky s využitím bilineárního vzorkování na čtvercové ploše pod úhlem. Pořízeno bez využití filmového mapování tónů

Bibliografie

1. ASHDOWN, Ian; ENG, P. Photometry and radiometry. *President by Heart Consultants Limited* [online]. 2002 [cit. 2024-04-19]. Dostupné z: https://www.cs.ubc.ca/~lsigal/teaching08/MeasuringLight.pdf.
2. ŽÁRA, Jiří; FELKEL, Petr; BENEŠ, Bedřich. *Moderní Počítačová grafika*. Computer Press, 2005. ISBN 80-251-0454-0.
3. AZARI, Banafsheh. *Bidirectional Texture Functions: Acquisition, Rendering and Quality Evaluation* [online]. 2018. [cit. 2024-04-17]. Dostupné z: https://www.researchgate.net/publication/327126983_Bidirectional_Texture_Functions_Acquisition_Rendering_and_Quality_Evaluation. Dis. pr.
4. QUINCEY, Paul. *Solid angles in perspective* [online]. 2021. [cit. 2024-04-19]. Dostupné z: https://www.researchgate.net/publication/353838553_Solid_angles_in_perspective.
5. HAINDL, Michal; FILIP, Jiri. *Visual texture* [online]. London, England: Springer, 2013 [cit. 2024-04-07]. Advances in Computer Vision and Pattern Recognition. ISBN 978-1-4471-4901-9. Dostupné z: https://link.springer.com/book/10.1007/978-1-4471-4902-6.
6. GUARNERA, Dar'ya; GUARNERA, Giuseppe Claudio. Reflectance Functions. In: *Virtual Material Acquisition and Representation for Computer Graphics* [online]. Cham: Springer International Publishing, 2018, s. 5–14 [cit. 2024-04-12]. ISBN 978-3-031-02595-2. Dostupné z DOI: 10.1007/978-3-031-02595-2_2.
7. JENSEN, Henrik Wann; MARSCHNER, Stephen R; LEVOY, Marc; HANRAHAN, Pat. A practical model for subsurface light transport. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* [online]. 2001, s. 511–518 [cit. 2024-04-21]. Dostupné z: http://www.graphics.stanford.edu/papers/bssrdf/bssrdf.pdf.
8. AKENINE-MÖLLER, T.; HAINES, E.; HOFFMAN, N. *Real-Time Rendering*. CRC Press, 2019. ISBN 9781315362007. Dostupné také z: https://books.google.cz/books?id=Y-KEDwAAQBAJ.
9. FELKEL, Petr; SLOUP, Jaroslav [online]. 2021. [cit. 2024-04-20]. Dostupné z: https://cent.felk.cvut.cz/courses/PGR/lectures/05_Transform_2.pdf. Prezentace z přednášky Transformace 2 z PGR.
10. DOMBEK, Daniel; KLEPRLÍK, Luděk; KLOUDA, Karel; JAKUB, Šístek [online]. 2024. [cit. 2024-04-21]. Dostupné z: https://courses.fit.cvut.cz/BI-LA2/@master/textbook/index.html. Studijní text Lineární algebra 2.
11. PHARR, Matt. *Physically based rendering, fourth edition: From theory to implementation* [online]. MIT Press, 2023 [cit. 2024-04-21]. Dostupné z: https://www.pbr-book.org/4ed/contents.

12. INC., Unity Software. *Realtime Global Illumination using Enlighten* [online]. [cit. 2024-05-02]. Dostupné z: `https://docs.unity3d.com/6000.0/Documentation/Manual/realtime-gi-using-enlighten.html`.
13. INC., Epic Games. *Lumen Global Illumination and Reflections* [online]. [cit. 2024-05-02]. Dostupné z: `https://dev.epicgames.com/documentation/en-us/unreal-engine/lumen-global-illumination-and-reflections-in-unreal-engine`.
14. HAINDL M. Filip J., Vávra R. Digital Material Appearance: the Curse of Tera-Bytes. *ERCIM News*. 2012, č. 90, s. 49–50.
15. RICHTR, Radek. *Dynamic Texture Modeling* [online]. 2018. [cit. 2024-05-01]. Dostupné z: `https://dspace.cvut.cz/handle/10467/79051`. PhD thesis. Czech Technical University.
16. CHERNIKOV, Yan. *Game Engine series* [online]. Youtube, 2018-09-30. [cit. 2024-05-04]. Dostupné z: `https://www.youtube.com/watch?v=JxIZbV_XjAs&list=PLlrATfBNZ98dC-V-N3m0Go4deliWHPFwT`.
17. FRAITURE, Luc. A History of the Description of the Three-Dimensional Finite Rotation. *The Journal of the Astronautical Sciences*. 2009, roč. 57, s. 207–232. ISSN 2195-0571. Dostupné z DOI: `10.1007/bf03321502`.
18. HABLE, John. *Filmic Tonemapping Operators* [online]. 2010. [cit. 2024-05-14]. Dostupné z: `http://filmicworlds.com/blog/filmic-tonemapping-operators/`.
19. HATKA, Martin; HAINDL, Michal. BTF rendering in Blender. *Proceedings of VRCAI 2011: ACM SIGGRAPH Conference on Virtual-Reality Continuum and its Applications to Industry*. 2011. Dostupné z DOI: `10.1145/2087756.2087794`.

Obsah příloh

- `readme.md` ... stručný popis obsahu média
- `exe`
  - `manual.md` ... manuál pro spuštění a modifikaci aplikace
  - `build` ... adresář se spustitelnou formou implementace
- `src`
  - `implementation`
    - `gitlab.txt` ... odkaz na gitlab repozitář s celkovou implementací herního enginu
    - `btf_visualizer` ... zdrojové kódy implementace aplikace BTF Visualizer
    - `renderer` ... zdrojové kódy implementace renderovacího systému
  - `thesis` ... zdrojová forma práce ve formátu LaTeX
- `media` ... adresář video ukázkami aplikace
  - `object_demo.mp4` ... video ukázka z první demonstrační scény aplikace
  - `plane_demo.mp4` ... video ukázka z druhé demonstrační scény aplikace
- `text` ... text práce
  - `thesis.pdf` ... text práce ve formátu PDF