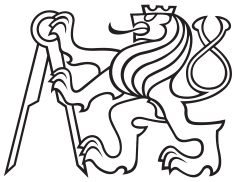
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačové grafiky a interakce

Vizualizace pokročilých reprezentací vzhledu materiálů

Bc. Vít Zadina

Vedoucí: Ing. Jiří Filip, Ph.D.
Obor: Otevřená informatika
Studijní program: Počítačová grafika
Květen 2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Zadina** Jméno: **Vít** Osobní číslo: **465951**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačová grafika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Vizualizace pokročilých reprezentací vzhledu materiálů

Název diplomové práce anglicky:

Visualization of advanced material appearance representations

Pokyny pro vypracování:

1. Seznamte se s problematikou reprezentace vzhledu materiálů pomocí BRDF a pokročilých reprezentací SVBRDF a BTF [1]. Standardní volně dostupné renderovací nástroje nástroje (MITSUBA [4], Blender [6]) nemají v současnosti podporu pro načtení BTF dat pro vizualizaci, tj. renderování vzhledu materiálů ve 3D scénách.
2. Seznamte se s dostupnými datovými formáty pro popis vzhledu materiálů a jejich možnostmi použití SVBRDF a BTF dat, např. [2,7,8]. U formátu změřených dat BIG navrženém v ÚTIA [2], se seznamte s jeho veřejně dostupným rozšířením [3]. Tato knihovna pro zápis/čtení do/z formátu zahrnuje implementaci vyrovnávací paměti pro rychlejší přístup k datům.
3. Analyzujte úzká místa a komplikace, která souvisí s přidáním podpory pro renderování ze změřených BTF dat za použití knihovny BIG [3] do nástrojů MITSUBA a Blender.
4. Implementujte podporu renderování na CPU za použití knihovny tohoto formátu [3] alespoň v jednom z těchto vizualizačních nástrojů [4,6]. Na základě informací o typu uložených dat v hlavičce souboru implementujete renderovací třídu, která na základě typu dat v hlavičce souboru (a) vybere správnou renderovací funkci, která je volána během renderování a provádí (a) úhlovou interpolaci změřených dat, (b) podporuje data uložená s podporou mip-mappingu a anisotropního filtrování.
5. V implementaci použijte renderovací metody standardně dostupné v použitém nástroji. Předpokládá se použití metod sledování paprsku. Pozn. Implementaci do vizualizačního nástroje proveďte pokud možno uživatelsky přístupným způsobem bez nutnosti zásahu do základní implementace vizualizačního nástroje, tj. nejlépe formou pluginu nebo rozšíření.
6. Pokud se ukáže, že je struktura formátu nebo přístup do vyrovnávací paměti formátu neoptimální z hlediska rychlosti renderování, navrhněte a implementujte úpravu formátu pro dosažení zlepšení.
7. Na třech zvolených testovacích scénách s jedním 3D objektem o různé složitosti detailů osvětlených: (a) bodovým světlem (b) mapou prostředí, porovnejte rychlost vizualizace v podporovaných nástrojích v závislosti na nastavení na (c) počtu uložených obrázků, (d) jejich velikosti, (e) velikosti použité cache, (f) datovém typu.
8. Na vhodné 3D scéně porovnejte (a) vizuální přínos filtrování aliasing artefaktů pomocí metod mip-mapping a anisotropního filtrování a (b) jejich výpočetní nároky.
9. Diskutujte možné úpravy datového formátu a jeho implementace pro další zrychlení vizualizace v jednotlivých nástrojích.

Seznam doporučené literatury:

[1] Haindl M., Filip J.: Visual Texture: Accurate Material Appearance Measurement, Representation and Modeling. Advances in Computer Vision and Pattern Recognition, © Springer-Verlag London 2013, (285 p., ISBN 978-1-4471-4901-9)
[2] Reference to UTIA BIG format specifications: Filip J., Kolafová M., Havlíček M., Vávra R., Haindl M., Rushmeier H.: Evaluating Physical and Rendered Material Appearance. The Visual Computer (Proceedings of Computer Graphics International - CGI), 34(6-8), pp.805-816, 2018
[3] BIG format (zdrojový kód), https://github.com/RadomirVavra/BIG
[4] MITSUBA renderer, https://www.mitsuba-renderer.org/
[5] BRDF plugin do MITSUBA rendereru, https://github.com/jdupuy/dj_brdf
[6] Blender, https://www.blender.org/
[7] MTL material format, http://paulbourke.net/dataformats/mtl/
[8] Appearance Exchange Format https://www.xrite.com/axf

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jiří Filip, Ph.D., UTIA AV ČR

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **12.01.2021** Termín odevzdání diplomové práce: **21.05.2021**

Platnost zadání diplomové práce: **30.09.2022**

Ing. Jiří Filip, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. | Datum převzetí zadání

Podpis studenta

Poděkování

Děkuji ČVUT, že mi je tak dobrou *alma mater*. Děkuji vedoucímu práce Ing. Jiřímu Filipovi, Ph.D za konzultaci a pomoc při vypracovávání diplomové práce. Také bych chtěl poděkovat Ing. Radomíru Vávrovi, Ph.D za poskytnuté rady ohledně používání MIF knihovny. Nakonec bych chtěl poděkovat rodině a přátelům, kteří mě podporovali při tvorbě této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické 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. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či příloh, a veškeré jejich dokumentace (dále souhrnně jen"Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze, 18. května 2021

.....

Abstrakt

Diplomová práce se zabývá vizualizací pokročilých materiálů a jejich vzhledu. Vysvětluje pojem vizuální textura. Věnuje se problematice popisu materiálů pomocí různých reprezentací (GRF, BRDF, SVBRDF, BTF). Popisuje vybrané formáty používané pro reprezentaci materiálů (MTL, AxF, BIG, MIF). Obsahuje rešerši vybraných rendererů (V-ray, Corona, Cycles, Mitsuba a PBRT) a jejich možná rozšíření.

Součástí práce je implementace rozšíření pro renderování z několika BTF reprezentací uložených ve formátu MIF pro Mitsuba renderer včetně možnosti mipmappingu a anizotropního filtrování. Součástí této implementace je třída BigRender, která slouží pro komunikaci mezi MIF knihovnou a rendererem. Tato třída může být použita při implementaci podpory MIF formátu i do dalších rendererů.

V rámci diplomové práce jsem také implementoval převodník (BigConvert) z původní verze BIG formátu do MIF formátu, jehož součástí je i generování mipmap pro mipmaping a anizotropické filtrování. V práci popisuji postup tohoto generování i samotnou implementaci anizotropního filtrování i mipmappingu.

Klíčová slova: BTF, Vizualizace, render, Mitsuba, MIF

Vedoucí: Ing. Jiří Filip, Ph.D.
ÚTIA AV ČR,
Pod Vodárenskou věží 4,
Praha 8

Abstract

Master's thesis deals with the visualization of advanced materials and their appearance. It explains the concept of visual texture. Also, it deals with the issue of describing materials using different representations (GRF, BRDF, SVBRDF, BTF). The thesis describes selected formats used for material representation (MTL, AxF, BIG, MIF). It also contains a search of selected renderers (V-ray, Corona, Cycles, Mitsuba, and PBRT) and their possible extensions.

The implementation of extensions for rendering several BTF representations saved in the MIF format for the Mitsuba renderer is part of the thesis. It includes the possibility of mipmapping and anisotropic filtering. A BigRender class, which is a part of this implementation, is used for communication between the MIF format and the renderer. This class can be used when implementing the MIF format support into other renderers.

As a part of my master's thesis, I also implemented a converter (BigConvert) from the original version of the BIG format to the MIF format. The converter also includes the generation of mipmaps for mipmapping and anisotropic filtering. The thesis describes the process of generating the actual implementation and anisotropic filtering and mipmapping.

Keywords: BTF, Visualization, render, Mitsuba, MIF

Obsah

Obrázky

Tabulky

Zdrojové kódy

Kapitola 1

Úvod

Cílem diplomové práce bylo zorientovat se v problematice vizualizace materiálů, jejich uložení do souboru (.mtl, .axf, .big .mif) a podpory těchto formátů ve vykreslovacích programech (rendererech). Dalším cílem práce bylo seznámit se s renderovacími programy, prozkoumat je a seznámit se s možností jejich rozšíření o podporu nových formátů. Na základě těchto informací vybrat renderovací program a navrhnout způsob implementace podpory formátu BIG (Big Image Group), který slouží pro uložení BTF textur a byl vyvinut na ÚTIA AV ČR. Implementace proběhla jako plugin do rendereru Mitsuba, který podporuje soubory .mif (tento formát vychází z knihovny MIF vyvinuté taktéž na ÚTIA AV ČR) a také byla vytvořena univerzální C++ třída BigRender, která zajišťuje komunikaci mezi rendererem a MIF knihovnou a lze ji využít při implementaci podpory MIF souborů v dalších rendererech. U výsledného rozšíření byla testována rychlost renderingu a vizuální přínos filtrování.

1.1 Přínos práce

V této kapitole bych rád představil smysl mé diplomové práce, proč vlastně vznikla a důvody vzniku formátu pro ukládání BTF textury (BIG, MIF). BTF textura je velký soubor obrázků, který může dosahovat velikosti jednotek gigabajtů. Pokud by jsme tuto texturu měli uloženou jako jednotlivé obrázky, tak by nám velmi velkou část renderovacího času zabralo čtení hlavičky jednotlivých formátů a proto ÚTIA AV ČR vyvinul formát pro jejich

uložení, který tyto obrázky bezkompresně uloží do jednoho souboru s jednou hlavičkou s informacemi o jednotlivých obrázcích.

Podpora BTF textur napříč renderovacími programy je velmi malá a většinou každý renderer používá vlastní proprietární soubor pro tyto materiály. V některých případech i vlastní upravenou dvousměrnou texturní funkci. Stejný problém se snaží vyřešit Appearance Exchange Format - AxF, který je vyvíjen společnosti X-Rite a podporuje SVBRDF i BTF textury. Zároveň je podporován mnoha renderovacími programy. Formát AxF se postupně stává průmyslovým standardem, ale přístup k jeho API není prozatím veřejně dostupný. Na rozdíl od formátů vyvíjených na ÚTIA AV ČR, které mají svůj zdrojový kód přístupný pomocí veřejného repozitáře githubu.

Dalším důvodem proč tato práce vznikla bylo to, že na ÚTIA AV ČR měli pro vizualizaci naměřených BTF textur renderer, který používal rasterizaci a byl napsaný v OpenGL. Plugin, který vznikl v rámci této práce, používá pro vizualizaci pathtraycing. Tedy vizualizace BTF textur pomocí mého pluginu bude mít mnohem realističtější výstupy díky odrazům paprsků, které mohou poukázat na nevýhody určitých druhů reprezentací BTF textur.

1.2 Navrhované řešení

Na základě analýzy jsem se rozhodl podporu formátu pro uložení BTF textur nejdříve implementovat pro Mitsuba renderer. Tento program je používán především vědeckou komunitou. Obsahuje velmi dobrou dokumentaci a podporu co se týká možností jeho rozšíření. Tyto vlastnosti jsou velmi vhodné pro implementaci. V rámci implementační části práce jsem vytvořil samostatnou třídu, která zařizuje komunikaci mezi MIF knihovnou a renderovacím programem (pluginem v Mitsuba rendereru). Během vývoje pluginu bude probíhat testování knihovny pro podporu BTF dat a budou navrženy její úpravy. Tento způsob implementace zajistí snadný přenos zdrojového kódu pro případnou implementaci v jiném rendereru.

Tato třída bude moct být použita i při implementaci podpory MIF v jiném renderovacím programu, například v rendereru Cycles (součást programu Blender), který je stejně jako Mitsuba psaný v C++, a tedy by jeho rozšíření mělo být jednodušší. Hlavní důvod proč proběhla implementace v rendereru Mitsuba je uživatelsky přívětivá instalace tohoto rozšíření pomocí pluginu, kdy se při instalaci mého pluginu bigbtf nemusí zasahovat do zdrojového kódu samotné Mitsuby.

Kapitola 2

Reprezentace materiálů

Digitální vizualizace vzhledu materiálů se používají čím dál více v různých odvětvích, například v autoprůmyslu, architektuře, filmech, herním průmyslu atd. Každým rokem kvalita vizualizací roste. Díky postupnému zvětšování výkonu grafických karet je v současné době možné použít metodu sledování paprsku (*ray tracing*) v reálném čase. Především v automobilovém průmyslu se dbá na co největší podobnost reálným materiálům. Reprezentace některých materiálů je velmi náročná, protože v nich může docházet k podpovrchovému odrazu a tedy změřit tento materiál a simulovat ho v počítači je velmi náročné.

V následujících kapitolách bude vysvětlen pojem textura a jaké metody jsou využívány pro reprezentaci realistických textur. V mnoha softwarech se pro reprezentaci vzhledu materiálů používají textury, které se jen snaží napodobit vzhled reálných materiálů různými způsoby. V této práci se budu zabývat jen texturami, které jsou vizuálně realistickými a fyzikálně co nejpřesnějšími reprezentacemi reálných materiálů.

2.1 Vizuální textura

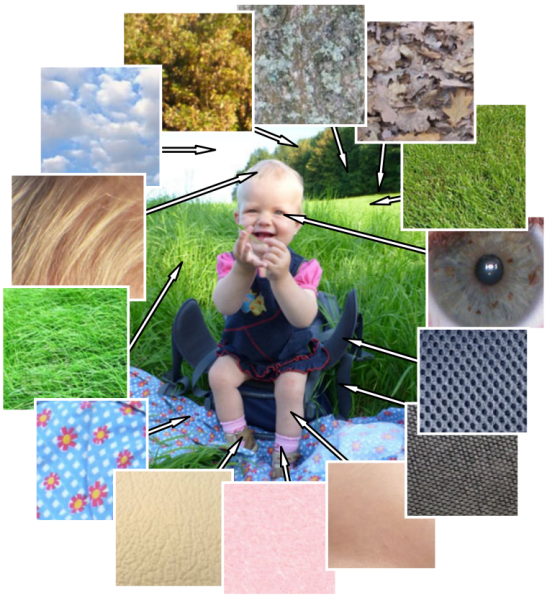
Textura je velmi abstraktní pojem, pod kterým si lidé z různých oborů představí různou věc. Pro ukázku níže uvedu několik odvětví kde se s pojmem textura (*texture*) můžeme setkat a u každého odvětví vysvětlím, co zde pojem textura znamená:

- Geologie
  - Textura v geologii představuje vlastnost hornin charakterizující prostorové uspořádání.
- Kosmologie
  - V kosmologoii se používá pro popis teoretického topologického defektu ve struktuře časoprostoru.
- Malování
  - Textura v malování odkazuje na vzhled a dojem z plátna. Tedy jak vypadá při pohledu a jak při doteku.
- Paleografie (historická věda zabývající se vývojem starého písma)
  - V paleografii je textura označení pro typ středověkého písma [1].
- Pedologie (půdoznalství)
  - Textura v pedologii určuje vlastnost půd a zemin, která charakterizuje zrnitostní jejich složení.
- Počítačová grafika
  - Texturování je v počítačové grafice technika, která umožňuje dodat realistický vzhled povrchu virtuálního trojrozměrného modelu, jehož jeden bod nazýváme ve 3D *texel* (texture pixel).

Lze najít další významy slova textura (*texture*), které jsem zde nepopsal. V mnoha zmíněných případech textura popisuje nějaký vzhled materiálu a jeho povrch. Jelikož se objektů zobrazených na monitoru nemůžeme dotýkat, tak nás v počítačové grafice zajímá hlavně jak dobře dokážeme daný materiál zobrazit (tedy jak věrně vypadá na pohled).

Vizuální textura je relativní pojem založený na rozlišení. Jakýkoli přírodní materiál má texturu, ale my ho vnímáme jako otexturovaný nebo hladce homogenní (tj. bez textur). Toto vnímání závisí pouze na rozlišení povrchu. Stejný povrch pozorovaný z dálky lze vnímat jako hladký, při bližším pozorování zjistíme, že má drsný povrch.

Neexistuje definice textury, kterou by uznávala celá věděcká komunita. Ani v oboru počítačové grafiky není jedna všemi přijímaná matematická definice. Jak oddělit normální obrázek od textury, v čem se liší? Textura by měla obsahovat nějaký vzor, který je v ní náhodně rozmístěn. Na obrázku 2.1 můžeme vidět fotografii, která sama o sobě není textura, ale mnoho textur

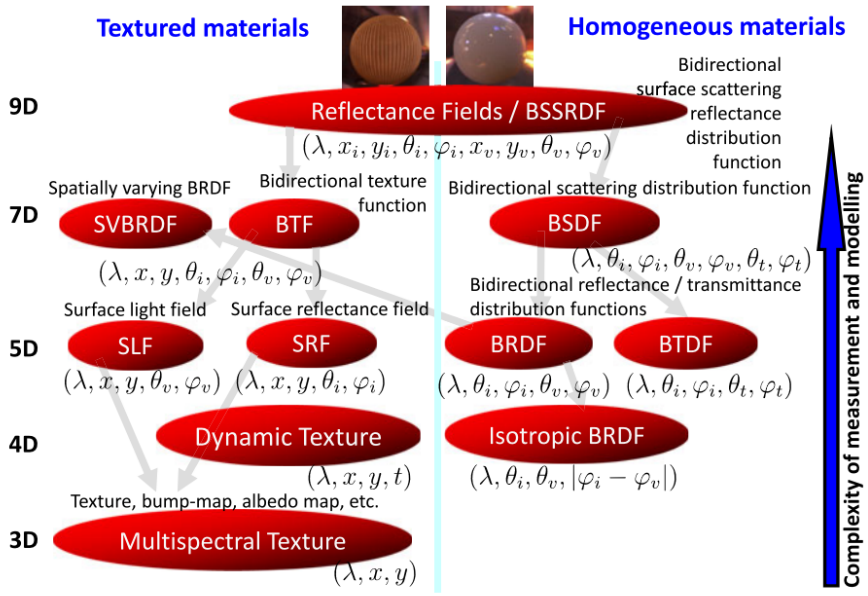


Obrázek 2.1: Vizuální textury jsou každodenní součástí našeho života. Obrázek byl převzat z [2, s. 3].

zobrazuje (texturu trávy, mraků, vlasů, …). Tedy má několik oblastí s různými texturami. Tyto textury nazýváme vizuálními texturami (*visual texture*) a snažíme se o jejich parametrizaci, která by dokázala zobrazit daný materiál pohledově nezávisle, tak aby nebyl rozeznatelný od skutečné podoby materiálu.

Vizuální struktura by měla splňovat několik podmínek [2]:

- ***Homogenita***: Textura je homogenní, pokud je její prostorová kovarianční funkce translačně invariantní.
- ***Uniformní struktura***: Textura obsahuje několik uniformně uspořádaných prvků (texelů), které mají přibližně stejné uspořádání v celé texturované oblasti.
- ***Variabilní odrazivost***: Textura obsahuje lokální proměnlivou odrazivost, ačkoliv je globálně rovnoměrně osvětlena. Tedy dochází k lokálním změnám směru odrazu světla nebo k odrazům uvnitř materiálu, který textura reprezentuje.
- ***Závislost měřítka***: Na základě zvětšení může být povrch vizuálně hladký (nízké rozlišení/zvětšení) nebo být texturovaný (velké rozlišení/zvětšení).
- ***Regionálnost***: Textura je často vlastnost jen určité části obrazu, viz obrázek 2.1.
- ***Materialita***: Textura představuje vzhled povrchového materiálu.



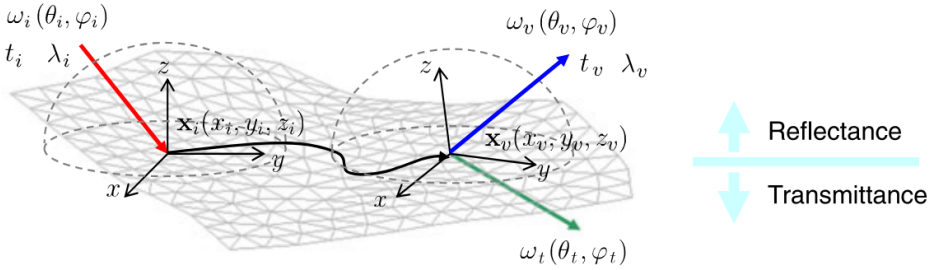
Obrázek 2.2: Modely odrazivé funkce a jejich rozdělení. Obrázek byl převzat z [2, s. 11].

2.2 Reprezentace textury v počítačové grafice

Texturu materiálu můžeme popsat pomocí obecné funkce odrazivosti GRF (*General Reflectance Function*), která je komplexní a má 16 proměnných. Aktuálně je nemožné materiál pomocí této funkce matematicky modelovat nebo provést jeho měření. Pro praktické použití se tato funkce zjednodušuje pomocí odstranění některých proměnných ve funkci. Těchto zjednodušených funkcí existuje mnoho. Na obrázku 2.2 můžeme vidět nejčastěji používané reprezentace [2]. Mezi ně patří dvousměrná distribuční funkce odrazu světla BRDF (*Bidirectional Reflectance Distribution Function*) a její prostorově proměnná varianta SVBRDF (*Spatially Varying Bidirectional Reflectance Distribution Function*). Dále pak dvousměrná funkce textury BTF (*Bidirectional Texture Function*). Tato práce se bude zabývat implementací podpory datového formátu vhodného především pro ukládání SVBRDF a BTF reprezentace.

2.2.1 Obecná funkce odrazivosti (GRF)

Odrazivost povrchu skutečného materiálu je velmi složitý fyzikální jev, který mimo jiné závisí na sférických úhlech dopadu a odrazu, času a světelném



Obrázek 2.3: Model obecné odrazivé funkce GRF. Obrázek byl převzat z [2, s. 10].

spektru viz obrázek 2.3. Pokud známe obecnou funkci odrazivosti můžeme přesně předpovědět, jak se jakýkoliv materiál zobrazuje za jakékoliv intenzity světla, směru nebo spektra osvětlení, ale také můžeme libovolný materiál rozpoznat na základě osvětlení scény.

Jak již bylo zmíněno funkce GRF má 16 dimenzí (16D) a zde je její zápis:

$$Y_r^{GRF} = GRF(\lambda_i, x_i, y_i, z_i, t_i, \theta_i, \varphi_i, \lambda_v, x_v, y_v, z_v, t_v, \theta_v, \varphi_v, \theta_t, \varphi_t);$$

Kde $r = [r_1, \ldots, r_{16}]$ používá multi-indexovou notaci s částečnými indexy. Všechny hodnoty, které může index nabývat jsou v knize [2] označovány pomocí $\bullet$, například barevné spektrum v RGB barevném prostoru popíšeme jako $Y_{\bullet,r_2,\ldots,r_{16}} = [Y_{R,r_2,\ldots,r_{16}}, Y_{G,r_2,\ldots,r_{16}}, Y_{B,r_2,\ldots,r_{16}}]$, prázdný index označíme $\varnothing$, např. mono-spektrální vstup označíme jako $Y_{\varnothing,r_2,\ldots,r_{16}}$.

Obecná funkce odrazivosti vyjadřuje dopadající světlo se spektrální hodnotou $\boldsymbol{\lambda_i}$ a jeho umístění na osvětlené ploše (bodě) $\boldsymbol{x_i, y_i, z_i}$ v čase $\boldsymbol{t_i}$ se směrem příchozího paprsku ve sférických souřadnicích $\boldsymbol{\omega_i = (\theta_i, \varphi_i)}$, které je pozorováno v čase $\boldsymbol{t_v}$ a umístěné na osvětlované ploše v bodech $\boldsymbol{x_v, y_v, z_v}$ pod sférickým úhlem odrazivosti $\boldsymbol{\omega_v = (\theta_v, \varphi_v)}$ se spektrální hodnotou $\boldsymbol{\lambda_v}$. Kde $\omega_t = (\theta_t, \varphi_t)$ je úhel propustnosti a $\omega = (\theta, \varphi)$ jsou úhly elevace (*polárního úhlu*) a azimutu. Parametry z_i, z_v indikují, že záření podél světelných paprsků není konstantní, ale záleží i na výšce odrazu. [2]

Jelikož přímé měření GRF funkce není prozatím technicky možné, používá se její zjednodušení. Na obrázku 2.2 můžeme vidět vybrané zjednodušené modely, které vycházejí z funkce GRF a jejich funkce a názvy. Existuje mnoho dalších zjednodušení obecné funkce odrazivosti, na obrázku 2.2 jsou uvedeny jen nejčastěji používané. Všechny tyto funkce se snaží zobrazit reálný materiál, tak aby odpovídal jeho fyzikálním vlastnostem a aby bylo možné tyto vlastnosti změřit nebo popsat funkcí.

V knize [2] je popsáno několik předpokladů pro zjednodušení funkce GRF. Tyto předpoklady jsou uvedeny níže včetně podoby funkce GRF za daných předpokladů. Tyto informace budou později využity při vysvětlování zjednodušených funkcí (BRDF, BTF, SVBRDF) a jejich vlastností. Jednotlivé předpoklady jsou:

P1 přenos světla materiálem probíhá v nulovém čase ($t_i = t_v$ a $t_v = \emptyset$)

$$Y_r^{GRF} = GRF_{P1}(\lambda_i, x_i, y_i, z_i, t_i, \theta_i, \varphi_i, \lambda_v, x_v, y_v, z_v, \theta_v, \varphi_v, \theta_t, \varphi_t),$$

P2 odrazivost povrchu je v čase neměnná ($t_v = t_i = const., t_v = t_i = \emptyset$)

$$Y_r^{GRF} = GRF_{P2}(\lambda_i, x_i, y_i, z_i, \theta_i, \varphi_i, \lambda_v, x_v, y_v, z_v, \theta_v, \varphi_v, \theta_t, \varphi_t),$$

P3 při interakci se nemění vlnová délka ($\lambda_i = \lambda_v$, tj., $\lambda_v = \emptyset$)

$$Y_r^{GRF} = GRF_{P3}(\lambda_i, x_i, y_i, z_i, t_i, \theta_i, \varphi_i, x_v, y_v, z_v, t_v, \theta_v, \varphi_v, \theta_t, \varphi_t),$$

P4 konstantní záření podél světelných paprsků ($z_i = z_v = \emptyset$), tedy předpokládá, že nedojde ke změnám odrazivosti během incidentu ani k odrazu během světelné cesty

$$Y_r^{GRF} = GRF_{P4}(\lambda_i, x_i, y_i, t_i, \theta_i, \varphi_i, \lambda_v, x_v, y_v, t_v, \theta_v, \varphi_v, \theta_t, \varphi_t),$$

P5 nulová průsvitnost ($\theta_t = \varphi_t = \emptyset$) (tedy žádná simultánní odrazivost a propustnost)

$$Y_r^{GRF} = GRF_{P5}(\lambda_i, x_i, y_i, z_i, t_i, \theta_i, \varphi_i, \lambda_v, x_v, y_v, z_v, t_v, \theta_v, \varphi_v),$$

P6 dopadající světlo opouští povrch ve stejném bodě $x_i = x_v, y_i = y_v$ ($x_v = y_v = \emptyset$)

$$Y_r^{GRF} = GRF_{P6}(\lambda_i, x_i, y_i, z_i, t_i, \theta_i, \varphi_i, \lambda_v, z_v, t_v, \theta_v, \varphi_v, \theta_t, \varphi_t),$$

P7 zanedbání podpovrchového rozptylu (*subsurface scattering*),

P8 neuvažování vlastního stínování (*self-shadowing*),

P9 neuvažování zastínění samo sebou (*self-occlusion*),

P10 neobsahuje interní odrazy,

P11 zákon o zachování energie říká, že veškeré dopadající světlo může být buď odraženo nebo absorbováno

$$\int_\Omega Y_r^{BRDF} \cos\theta_v d\varphi_v d\theta_v \leq 1,$$

P12 Helmholtzova reciprocita [3] říká, že BRDF se nemění když se vstupní ($\omega_i = (\theta_i = \varphi_i)$ a výstupní ($\omega_v = (\theta_v = \varphi_v)$ úhly prohodí

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

P13 pevné osvětlení ($\theta_i = \text{const.}, \varphi_i = \text{const.}$), neboli se nemění úhel dopadu světla

P14 neměnící se pozorovací úhel ($\theta_v = \text{const.}, \varphi_v = \text{const.}$)

P15 je prostorově nezávislá ($x_i = y_i = x_v = y_v = \emptyset$)

$$Y_r^{GRF} = GRF_{P15}(\lambda_i, z_i, t_i, \theta_i, \varphi_i, \lambda_v, z_v, t_v, \theta_v, \varphi_v, \theta_t, \varphi_t),$$

P16 neuvažuje odraz ($\theta_v = \varphi_v = \emptyset$)

$$Y_r^{GRF} = GRF_{P16}(\lambda_i, x_i, y_i, z_i, t_i, \theta_i, \varphi_i, \lambda_v, x_v, y_v, z_v, t_v, \theta_t, \varphi_t),$$

P17 nezávislost na azimutovém směru odrazu (izotropie), tj. ($\varphi_i - \varphi_v$ a $\varphi_i - \varphi_t$)

$$Y_r^{GRF} = GRF_{P17}(\lambda_i, x_i, y_i, z_i, t_i, \theta_i, (\varphi_i - \varphi_v), \lambda_v, x_v, y_v, z_v, t_v, \theta_v, (\varphi_i - \varphi_t)).$$

V následujících kapitolách bude uvedeno, využití předpokladů k reprezentaci reálných materiálů. Představím zde různé modely, které se často používají. Nejdříve představím funkci, která je nejméně realistická, ale pro některé materiály dostačující a to BRDF, poté její prostorovou variantu SVBRDF a nakonec dvousměrnou texturní funkci (BTF), pomocí které lze modelovat velmi přesně většinu reálných materiálů.

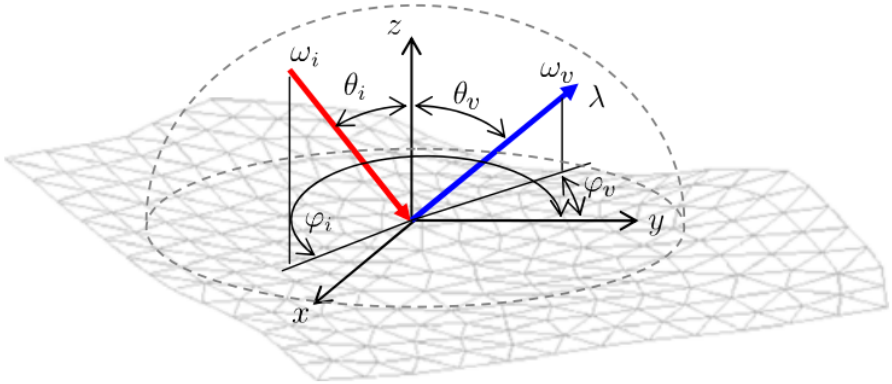
2.2.2 Dvousměrná distribuční funkce odrazu světla (BRDF)

Jak lze vidět na obrázku 2.2 dvousměrná distribuční funkce odrazu světla se používá pro homogenní materiály. Mezi tyto materiály patří hlavně nanesené barvy, laky, plasty a kovy. Vizualizaci BRDF textur můžeme vidět na obrázku 2.5. Anizotropií[1] materiálu rozumíme změnu vzhledu materiálu při jeho rotaci pro fixní pozice světla a kamery. Je typická pro materiály se směrovou mikrostrukturou, např. dřevo nebo broušený kov. Na obrázku 2.5 je velmi dobře zobrazena rozdílná hrubost a anizotropie jednotlivých materiálů, takže si lze představit jaký materiál je na jaké kouli umístěný.

Dvousměrná distribuční funkce odrazu má pět dimenzí. Zobrazení jednotlivých proměnných na kterých BRDF závisí můžete najít na obrázku 2.4. Její definice zní takto:

$$Y_r^{BRDF} = BRDF(\lambda, \theta_i, \varphi_i, \theta_v, \varphi_v),$$

[1] Také můžeme říct, že anizotropie je vlastnost, kterou se označuje závislost určité veličiny na volbě směru. Tedy vlastnosti jsou v různých směrech různé.



Obrázek 2.4: Model dvousměrné distribuční funkce odrazu světla (*BRDF*). Obrázek byl převzat z [2, s. 19].

Pokud budeme ignorovat spektrální závislost, můžeme BRDF funkci zjednodušit na čtyř-dimenzionální funkci, která bude záviset jen na úhlu dopadu světla a úhlu z kterého danou texturu sledujeme. Toto zjednodušení nazýváme Lambertian BRDF. [2]

Funkce BRDF pro své zjednodušení používá předpoklady P1 až P12 a předpoklad P15 o prostorové nezávislosti:

P1 přenos světla materiálem probíhá v nulovém čase ($t_i = t_v$ a $t_v = \emptyset$),

P2 odrazivost povrchu je v čase neměnná ($t_v = t_i = const., t_v = t_i = \emptyset$),

P3 při interakci se nemění vlnová délka ($\lambda_i = \lambda_v$, tj., $\lambda_v = \emptyset$) ,

P4 konstantní záření podél světelných paprsků ($z_i = z_v = \emptyset$),

P5 nulová propustnost ($\theta_t = \varphi_t = \emptyset$) (tedy žádná simultánní odrazivost a propustnost),

P6 dopadající světlo opouští povrch ve stejném bodě $x_i = x_v, y_i = y_v$ ($x_v = y_v = \emptyset$),

P7 neuvažování podpovrchového rozptylu světla (*subsurface scattering*),

P8 neuvažování vlastního stínování (*self-shadowing*),

P9 neuvažování zastínění samo sebou (*self-occlusion*),

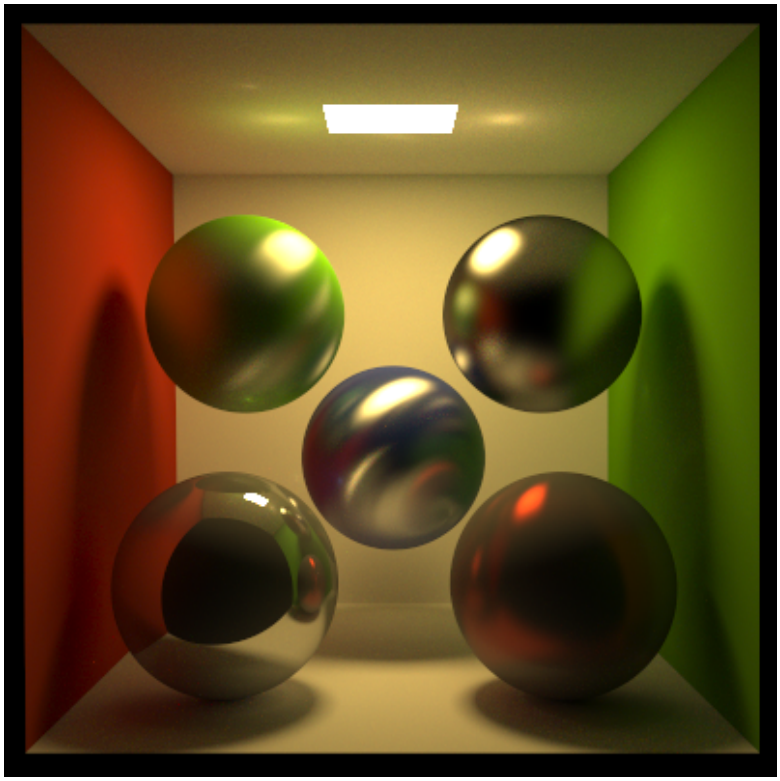
P10 neobsahuje interní odrazy,

P11 zákon o zachování energie,

P12 Helmholtzova reciprocita [3],

P15 prostorová nezávislost ($x_i = y_i = x_v = y_v = \emptyset$). [2]

BRDF lze měřit speciálními přístroji, nebo ho můžeme odvodit pomocí analytického popisu, který bude co nejvíce odpovídat pozorované skutečnosti. Tyto analytické vzorce můžeme rozdělit na empirické a fyzikálně založené.



Obrázek 2.5: Anisotropic Ward BRDF. Obrázek byl převzat z [8].

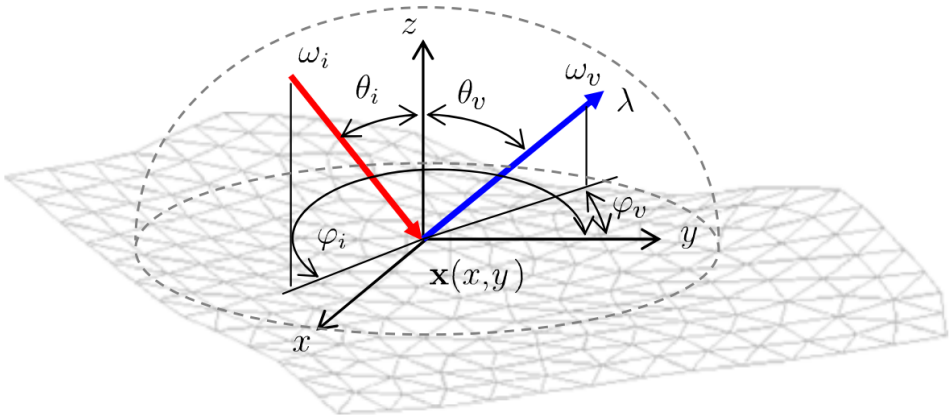
Nejznámější empirický model, který vychází z BRDF je Phongův osvětlovací model [4], dalšími zástupci empirických modelů jsou například Blinn–Phongův model [5], Torrance–Sparrowův model [6] a Cook–Torrancův model [7] a mnoho dalších. Poslední dva zmíněné modely používají pro simulaci povrchu materiálu mikroplošky. Tyto modely nejsou fyzikálně přesné, ale jejich hlavní výhodou je, že nejsou náročné na výpočetní výkon, proto jsou velmi oblíbené. Avšak nesplňují všechny požadavky pro BRDF funkci například Phongův osvětlovací model porušuje předpoklad P11 o zachování energie.

2.2.3 Prostorově se měnící dvousměrná distribuční funkce odrazu světla SVBRDF

SVBRDF je stejně jako funkce BTF sedmi-dimenzionální a její model vypadá takto:

$$Y_r^{SVBRDF} = SVBRDF(\lambda, x, y, \theta_i, \varphi_i, \theta_v, \varphi_v),$$

tedy modely BTF a SVBRDF funkce jsou stejné, ale na rozdíl od BTF obsahuje i předpoklady P7 až P10. Dále funkce neuvažuje předpoklady P11



Obrázek 2.6: BTF a SVBRDF odrazivostní model. Obrázek byl převzat z [2, s. 14].

a P12, které nám umožňují zjednodušit měření, ale zvětšují kompresi dat a tím i zhoršují vizuální kvalitu. Ukázku použití SVBRDF materiálu při renderingu a jeho srovnání s fotografií reálného materiálu můžete vidět na obrázku 2.7.

Zhoršení vizuální kvality (oproti BTF) způsobuje hlavně omezení vlastnosti reciprocity BRDF vhodným pro měření skoro plochých povrchů, ale předpoklad P5 omezuje jeho použití na neprůsvitné povrchy. Jednou z hlavních nevýhod SVBRDF oproti BTF je, že SVBRDF nebere v potaz samozastínění ve struktuře materiálu. Tento model může být vizualizován stejně jako funkce BTF viz obrázek 2.6. U SVBRDF uvažujeme tyto předpoklady:

P1 přenos světla materiálem probíhá v nulovém čase ($t_i = t_v$ a $t_v = \emptyset$),

P2 odrazivost povrchu je v čase neměnná ($t_v = t_i = const., t_v = t_i = \emptyset$),

P3 při interakci se nemění vlnová délka ($\lambda_i = \lambda_v$, tj., $\lambda_v = \emptyset$) ,

P4 konstantní záření podél světelných paprsků ($z_i = z_v = \emptyset$),

P5 nulová propustnost ($\theta_t = \varphi_t = \emptyset$) (tedy žádná simultánní odrazivost a propustnost),

P6 dopadající světlo opouští povrch ve stejném bodě $x_i = x_v, y_i = y_v$ ($x_v = y_v = \emptyset$),

P7 neuvažování podpovrchového rozptylu (*subsurface scattering*),

P8 neuvažování vlastního stínování (*self-shadowing*),

P9 neuvažování zastínění samo sebou (*self-occlusion*),

P10 neobsahuje interní odrazy,

P11 zákon o zachování energie,

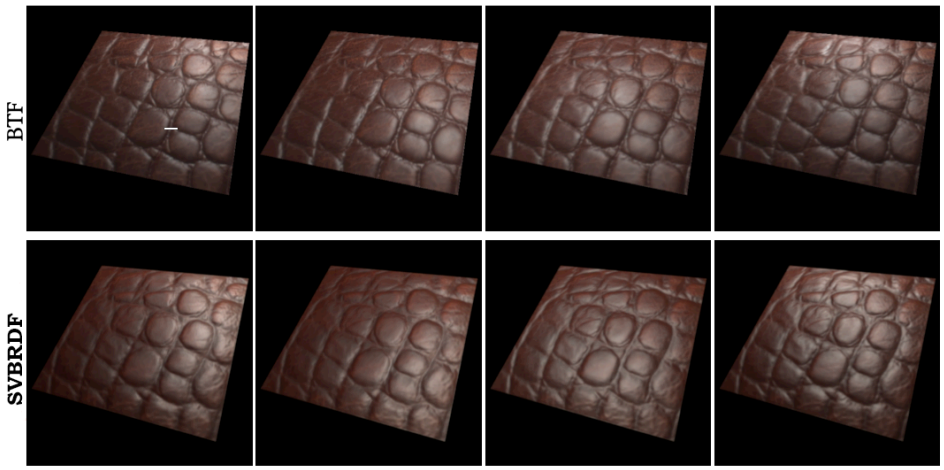
P12 Helmholtzova reciprocita [3]. [2]

Obrázek 2.7: Ukázka renderovaní materiálů pomocí SVBRDF modelu a porovnání s reálnými fotografiemi. SVBRDF bylo vytvořeno z jednoho snímku materiálu. Obrázek byl převzat z [9, s. 8]

Obrázek 2.8: Obrázek obsahuje fotografie reálného materiálu a jeho reprezentaci pomocí BTF textury. Tato reprezentace na rozdíl od SVBRDF dokáže zaznamenat a zobrazit sebezastínění materiálu. Obrázek poskytl vedoucí práce.

2.2.4 Dvousměrná texturní funkce (BTF)

Sedmi-dimenzionální dvousměrná texturní funkce (*Bidirectional Texture Function*) a její model odrazivosti (viz obrázek 2.6) je v současné době vizuálně i fyzikálně nejpřesnějším modelem GRF, který může být měřen a modelován. Velmi často se používá pro reprezentaci materiálů, které se pomocí jiných textur velmi těžko zobrazují, např. tkaniny, dřeva. Nicméně vyžaduje nejpokročilejší možnosti modelování a je velmi náročný na hardware. Jelikož BTF zachycuje vzhled materiálu běžně prostřednictvím tisíců fotografií materiálu pro různé směry pohledu a osvětlení. Největším problémem BTF reprezentace je množství těchto obrázků a jejich velikost dosahující až několika desítek gigabajtů. Na obrázku 2.8 lze vidět materiály, které jsou velmi často



Obrázek 2.9: Ukázka rozdílu u vyrendrovaných obrázků pomocí BTF a SVBRDF. Obrázek byl převzat z [9, s. 9].

reprezentovány pomocí BTF, protože SVBRDF je nedokáže reálně zobrazit, protože neumí pracovat se stíny, které vrhá materiál sám na sebe. Zde konkrétně jde o paletu několika látek. Rozdíly mezi BTF a SVBRDF modelem jsou demonstrovány na obrázku 2.9. BTF odrazivostní model:

$$Y_r^{BTF} = BTF(\lambda, x, y, \theta_i, \varphi_i, \theta_v, \varphi_v)$$

BTF přijímá prvních šest výše zmíněných předpokladů (P1 až P6) tedy:

P1 přenos světla materiálem probíhá v nulovém čase ($t_i = t_v$ a $t_v = \emptyset$),

P2 odrazivost povrchu je v čase neměnná ($t_v = t_i = const., t_v = t_i = \emptyset$),

P3 při interakci se nemění vlnová délka ($\lambda_i = \lambda_v, \text{tj.}, \lambda_v = \emptyset$) ,

P4 konstantní záření podél světelných paprsků ($z_i = z_v = \emptyset$), tedy předpokládá, že nedojde ke změnám odrazivosti během incidentu ani k odrazu během světelné cesty,

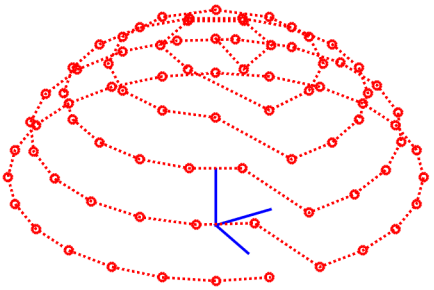
P5 nulová propustnost ($\theta_t = \varphi_t = \emptyset$) (tedy žádná simultánní odrazivost a propustnost),

P6 dopadající světlo opouští povrch ve stejném bodě $x_i = x_v, y_i = y_v$ ($x_v = y_v = \emptyset$). Ale nejsou zde uvažovány předpoklady P7 až P10, tedy dochází k podpovrchovému rozptylu, vlastnímu stínování, zastínění samo sebou a obsahuje interní odrazy. [2]

2.2.5 Testované reprezentace BTF dat

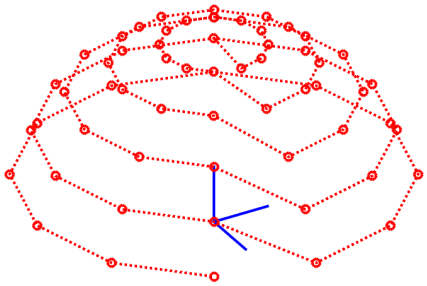
BTF data mohou být reprezentována pomocí různých datových reprezentací, které se odlišují v distribuci směrů osvětlení a pohledu pro jednotlivé měřené snímky materiálu. Tyto distribuce se často liší v závislosti na optických vlastnostech měřeného materiálu, možností měřícího zařízení nebo požadované doby měření. Pro testování datového formátu v rámci této diplomové práce jsem použil čtyři různé datové reprezentace, mimo jiné, v současnosti používané v ÚTIA AV ČR.

Hemisfericky uniformní – v implementaci označovaná jako *UBO81x81*. Jedná se o formát původně navržený na Univerzitě v Bonnu [10]. Měřené směry osvětlení a pohledu jsou identické a jsou distribuovány, tak aby rovnoměrně pokryly celou hemisféru viz obrázek 2.10. Výhodou je přibližně stejná úhlová vzdálenost mezi měřenými snímky za cenu mírně složitější interpolace dat, která vyžaduje pro každý směr předpočítané indexy sousedních směrů. Tato reprezentace zachycuje měřený materiál pomocí 6561 snímků.



Obrázek 2.10: Ukázka hemisfericky uniformního samplování hemisféry. Obrázek mi poskytl vedoucí práce.

Úhlově uniformní – v implementaci označovaná jako *Uniform*. Tato reprezentace sampluje směry v přesně daném kroku mezi elevacemi a azimuty, konkrétně $\Delta\theta = 15°$ a $\Delta\varphi = 30°$. Toto řešení umožňuje pro interpolaci dat použití jednoduché lineární interpolace ve čtyřech dimenzích avšak za cenu neefektivního zahuštění vzorku na vrchlíku hemisféry. Tato reprezentace zachycuje měřený materiál pomocí 3660 snímků. Ukázka takového samplování je vidět na obrázku 2.11.



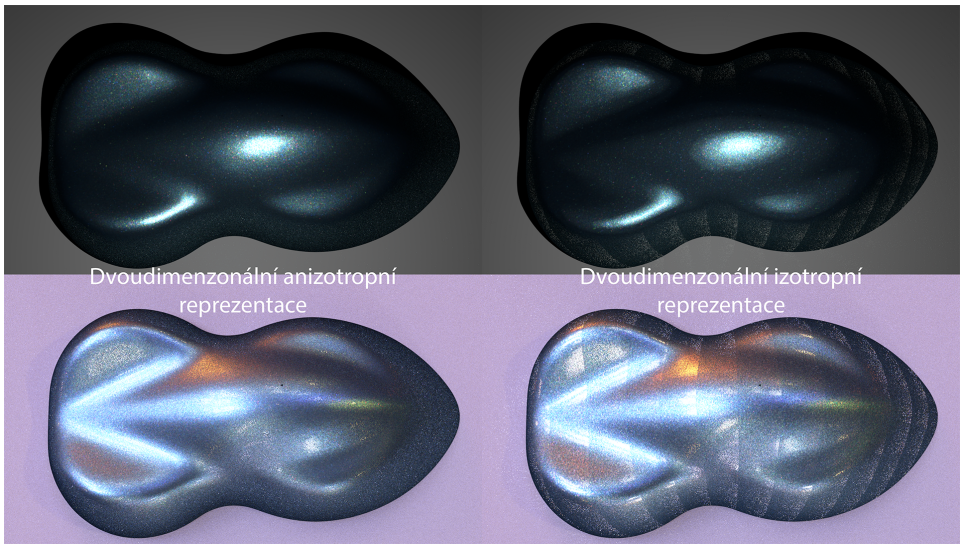
Obrázek 2.11: Ukázka úhlově uniformního samplování hemisféry. Obrázek mi poskytl vedoucí práce.

Pro měření metalických autolaků je možno využít několika zjednodušujících předpokladů, např. dokonalé rovinatosti a náhodnosti vzorku. Díky

tomu lze tyto materiály zachytit pomocí vzorkování jen ve dvou úhlových dimenzích a tedy pomocí relativně malého množství snímků. Aby bylo možné věrně zachytit tvar a intenzitu odlesku, který je charakteristický pro tyto materiály, byla použita parametrizace úhlů, která není vztažená k normále povrchu, ale k vektoru mezi směry osvětlení a pohledu, tzv. half-vektoru [11]. Ukázku použití dvoudimenzonální izotropní a anizotropní reprezentace BTF materiálů můžete vidět na obrázku 2.12.

Dvoudimenzionální izotropní – v implementaci označovaná jako *CoatingRegular*. Tato varianta vzorkuje pouze elevační úhel half vektoru θ_h a elevační úhel osvětlení θ_d vztažený half vektoru [12]. Tato reprezentace měří pouze směry osvětlení a pohledu, které jsou osově symetrické podél normály povrchu. Vzhledem k tomu, že nezávisí na azimutu nedokáže změřit anizotropní chování materiálů a tedy závisí jaký azimut pohledu je vybrán během měření. V experimentech byly použity úhlové kroky $\Delta\theta_h = 5°$ a $\Delta\theta_d = 15°$, díky kterým je měřený materiál zachycen pomocí 108 snímků.

Dvoudimenzionální anizotropní – v implementaci označovaná jako *CoatingSpecial*. Tato varianta vzorkuje pouze elevační úhel θ_h a azimutový úhel φ_h half vektoru [12]. Tato reprezentace zachycuje materiál v okolí retroodrazů, tj. blízké směry osvětlení a pohledu, pro různé azimuty. Díky tomu umožňuje efektivně zachytit anizotropní chování materiálů, tj. změnu jejich vzhledu danou různou mikrostrukturou v různých směrech. V experimentech byly použity úhlové kroky $\Delta\theta_h = 5°$ a $\Delta\varphi_h = 30°$, díky kterým je měřený materiál zachycen pomocí 216 snímků.



Obrázek 2.12: Srovnání renderování dvoudimenzionální anizotropní (vlevo) a izotropní (vpravo) reprezentace dat pro osvětlení bodovým světlem (nahoře) a mapou prostředí (dole). U izotropní reprezentace dochází k sečtení chyby jak můžeme pozorovat na obrázcích vpravo, kde u hran textury přibývá šum.

Kapitola 3

Formáty reprezentace vzhledu materiálů

Pro reprezentaci materiálů v grafických programech bohužel zatím neexistuje obecně přijatý standard. Existuje mnoho formátů pro reprezentaci materiálů, které jsou podporovány různými grafickými aplikacemi. Tyto programy si velmi často materiály uchovávají ve svých interních formátech a jen pro přenos modelů s materiály mezi různými softwary používají obecnější formáty.

Nejznámějším a nejvíce podporovaným formátem je formát MTL [13], který byl vyvinut v roce 1995 a od té doby vyšlo hned několik nových verzí. Bohužel nepodporuje BTF formát textur. Modernějším formátem a novým průmyslovým standardem se pomalu stává formát AxF [14] od společnosti X-Rite, ale přístup k jeho API není prozatím volně přístupný. Tento formát již obsahuje podporu BTF a SVBRDF textur a má nativní podporu několika rendererů.

3.1 Material Library File – MTL

Formát MTL (Material Library File) je knihovna materiálů, tedy může obsahovat více materiálů, u kterých obsahuje informaci o barvě, textuře a mapě odrazů. Tyto vlastnosti jsou přiřazeny povrchu a bodům jednotlivých objektů. Všechna data jsou uložená v ASCII formátu a soubor s materiálem používá koncovku *.mtl*. Ukázku souboru s MTL materiálem můžete vidět v ukázce kódu 1. [13]

Formát MTL je velmi zastaralý, dokonce jeho oficiální verze nepodporuje technologie jako jsou mapa zrcadlového odrazu (*specular map*) ani parallax maping (technika zohledňující zorný úhel odrazu), ale tyto funkce díky jednoduchosti lze do souboru dodat a vytvořit si tak vlastní MTL generátor. Díky jeho velmi jednoduché modifikaci je velmi oblíbený a hojně využívaný. Pokud stahujete 3D model z internetu ve formátu Wavefront .obj, velmi často jsou materiály použité v modelu přiloženy v tomto formátu.

```
Material name statement:
   newmtl my_mtl
Material color and illumination statements:
   Ka 0.0435 0.0435 0.0435
   Kd 0.1086 0.1086 0.1086
   Ks 0.0000 0.0000 0.0000
   Tf 0.9885 0.9885 0.9885
   illum 6
   d -halo 0.6600
   Ns 10.0000
   sharpness 60
   Ni 1.19713
Texture map statements:
   map_Ka -s 1 1 1 -o 0 0 0 -mm 0 1 chrome.mpc
   map_Kd -s 1 1 1 -o 0 0 0 -mm 0 1 chrome.mpc
   map_Ks -s 1 1 1 -o 0 0 0 -mm 0 1 chrome.mpc
   map_Ns -s 1 1 1 -o 0 0 0 -mm 0 1 wisp.mps
   map_d -s 1 1 1 -o 0 0 0 -mm 0 1 wisp.mps
   disp -s 1 1 .5 wisp.mps
   decal -s 1 1 1 -o 0 0 0 -mm 0 1 sand.mps
   bump -s 1 1 1 -o 0 0 0 -bm 1 sand.mpb
Reflection map statement:
   refl -type sphere -mm 0 1 clouds.mpc
```

Kód 1: Ukázka vnitřní podoby MTL formátu

3.2 Appearance Exchange Format - AxF

Formát AxF [14] byl vyvinut firmou X-Rite pro reprezentaci vzhledu materiálů pro účely jejich fotorealistické vizualizace. Na obrázku 3.1 lze vidět materiály, pro jejichž uložení je vhodné uložení do AxF formátu. V principu se jedná o škálovatelný metaformát umožňující uložení vzhledu materiálu v různých parametrizacích, např. BRDF, SVBRDF, BTF. Umožňuje ukládání variabilních spektrálních informací a reprezentaci specifických materiálů např. autolaků pomocí proprietární reprezentace.

Obrázek 3.1: Ukázka materiálů vhodných pro reprezentaci pomocí AxF formátu. Obrázek byl převzat z [15].

Tento formát se používá uvnitř TAC (Total Appearance Capture) ekosystému firmy X-Rite, hlavním cílem tohoto formátu je zavést průmyslový standart pro ukládaní BRDF,SVBRDF, BTF a dalších variant textur nezávisle na programu a operačním systému. Aktuálně existuje vývojářské rozhraní (API) pro systém Windows a Linux. Pro MacOS je podpora teprve ve vývoji. Obsahuje množinu reprezentací materiálů pro zajištění kompatibility s ostatními SVRDF pracovními prostředími, které jsou implementované v jiných formátech.

Tento formát má nativní podporu v několika renderech (Nvidia Iray, Maxwell, Radeon Prorender, …), které jsou používány v herních enginech nebo v 3D programech jakou jsou Unreal Engine, 3Ds Max, Maya, Blender a další.

3.3 Big image group - BIG

Big image group (BIG) je opensourcový souborový formát pro ukládání skupiny obrázků s velkým dynamickým rozsahem a vysokou kvalitou s podporou desetinných čísel. Byl vyvinut v Ústavu teorie informace a automatizace Akademie věd České republiky (ÚTIA AV ČR) a jeho knihovna je napsána v jazyku C++ [16, s. 7]. Formát je bezeztrátový (neobsahuje kompresi dat) a snaží se optimalizovat přístup k jednotlivým texturám a zrychlit tak jejich zpracovávání a tím i renderovací čas (uvažujeme zde data o velikosti několika gigabajtů).

ID	Jméno	Datový typ	Defaultní hodnota
1	počet obrázků	unsigned 64-bit	0
2	výška obrázku	unsigned 64-bit	0
3	šířka obrázku	unsigned 64-bit	0
4	počet rovin (kanálů)	unsigned 64-bit	1
5	pořadí dat	unsigned 64-bit [ ]	1 2 3 4
6	data	any [ ]	
7	XML	char [ ]	

Tabulka 3.1: Seznam možných typů chunků a jejich defaultních hodnot

K optimalizaci používá cache ukládání/načítání textur, ke kterým bylo naposledy přistupováno do RAM paměti počítače, pokud je to možné, lze načíst celou texturu do RAM, případně i celý BIG soubor. Formát obsahuje mnoho možností a nastavení, tak aby byl co nejvíce konfigurovatelný. I bez použití uložení do paměti RAM má tento formát tu výhodu, že místo desítek až stovek jednotlivých obrázků, máte všechny obrázky materiálu v jednom souboru. Tento fakt také velmi zrychluje renderovací čas, protože zde odpadá režie s otevíráním nových souborů a čtením jejich hlavičky a přístupu k jejich datům.

Data jsou zde uložena v little-endian formátu. Není zde použitá žádná komprese dat. Soubor začíná magickým číslem 42 49 47 00 00 00 00 00 v hexadecimální soustavě. Dále následují chunky, které tvoří hlavičku souboru, ale jsou používány i pro ukládání jednotlivých obrázků (datový chunk). V hlavičce jsou uloženy další parametry, jako je počet obrázků, výška a šířka jednotlivých obrázků, atd. Všechny aktuálně podporované parametry můžete vidět v tabulce 3.1. XML chunk v současné verzi BIG formátu je implementovaný jen částečně a zatím se nevyužívá. U pořadí dat jde použít v současné době jen řazení dat v pořadí „1 2 3 4“.

Pořadí dat s parametrem „1 2 3 4“ určuje jakým způsobem je obrázek uložen a jak má být tedy čten. Konkrétně toto pořadí znamená, že první smyčka jde přes obrázky, další cyklus jde přes jejich řádky, poté iterujeme přes individuální pixely a poslední smyčka iteruje přes hodnoty uložené v pixelu (gray-scale, RGB, …).

BIG formát podporuje několik datových typů, které můžete vidět v tabulce 3.2. Využívá šablon a kód tak může být rozšířen i o další datový typ. Implementace jednotlivých funkcí se u některých datových typů liší, kvůli jejich různé bitové délce. Lze předpokládat, že při použití cache paměti pro textury v různých datových typech bude efektivita tohoto řešení různá.

ID	Jméno	Počet bitů	popis
1	half	16-bit	číslo s desetinnou čárkou podle směrnice IEEE 754
2	float	32-bit	číslo s desetinnou čárkou podle směrnice IEEE 754
3	double	64-bit	číslo s desetinnou čárkou podle směrnice IEEE 754
4	signed char	8-bit	celé číslo se znaménkem
5	unsigned char	8-bit	celé číslo bez znaménka
6	short	16-bit	celé číslo se znaménkem
7	unsigned short	16-bit	celé číslo bez znaménka
8	int	32-bit	celé číslo se znaménkem
9	unsigned int	32-bit	celé číslo bez znaménka
10	long long	64-bit	celé číslo se znaménkem
11	unsigned long long	64-bit	celé číslo bez znaménka
12	bool	1-bit	pravdivostní hodnota

Tabulka 3.2: Datové typy které mohou být uloženy v datovém chunku [17]

Uložené data (textury) musí být ve formě n-dimenzionálního pole, kde n je v základní verzi nejvýše čtyři. Obrázky v něm uložené musí mít stejný rozměr. Jediná přípustná možnost je uložit do BIG formátu obrázky s více datovými typy, které jsou uloženy v indexových mapách.

Na webových stránkách ÚTIA AV ČR[1] se v galerii nacházejí naměřená texturní data v BIG formátu. Bohužel tyto data jsou uložena ve starší verzi formátu, s kterými nebude nová verze formátu fungovat, tedy je potřeba data nově přeuložit. Formát se od vydání článku [16] dále vyvíjel a proběhla změna některých metadat. Pro tento účel jsem vyvinul nástroj pro převod a generování mipmap a anizotropních map, o kterém se můžete dozvědět více v kapitole 6.7 „Převodník do MIF souborů“.

3.4 Multi-Image Format - MIF

Během implementace jsem narazil na problémy s implementací cache paměti v BIG knihovně, které jsou popsány níže v kapitole 6.9.2 „Problém s cache pamětí v BIG knihovně“. Jelikož Mitsuba pracuje vícevláknově, tak použití aktuální implementace pro částečnou cache v BIG knihovně nebylo možné

[1] `http://btf.utia.cas.cz/`

použít. Docházelo zde k neočekávaným pádům uvnitř standardní implementace listu a zároveň se čtená data nemazala ze `shared_ptr`. Data textur tak dokázala zabrat více paměti než reálně měla. Navíc jsme s vedoucím práce a panem Radomírem Vávrou z ÚTIA AV ČR řešili novou podobu XML formátu a jestli jeho čtení bude součástí knihovny nebo nebude. Bylo navrženo, že by mohlo být uloženo více BTF textur v jednom souboru, který by nemusel obsahovat jen BTF textury. A proto vznikl formát Multi-Image Format (MIF) [18].

MIF je opensourcový souborový formát, který může uložit několik obrazových souborů (v našem případě s BTF daty). Pro práci s tímto formátem vznikla knihovna MIF, která je napsána v C++. Ve verzi 0.4 umožňuje práci pouze s daty ve formátu float, ale podpora dalších datových typů bude přidána. Na rozdíl od BIG formátu si ukládá meta informace o materiálu a jeho měření do XML souboru, který je součástí jeho hlavičky. Tyto informace poskytuje uživateli formou jednotlivých tříd (objektů). Obsahuje také podporu pro ukládání informací o mipmapách, čehož v implementační části využiji.

MIF ve verzi 0.4 nemá implementované částečné čtení z paměti, ale umožňuje načtení celého podsouboru[2] do paměti nebo čtení textury z disku. MIF vznikl pro zlepšení práce s BTF texturami a může sloužit jak k renderování jednotlivých textur, tak i k jejich analýze pomocí dat, které jsou uloženy v XML hlavičce.

[2] Podsouborem je myšlen soubor s jednou BTF texturou.

Kapitola 4

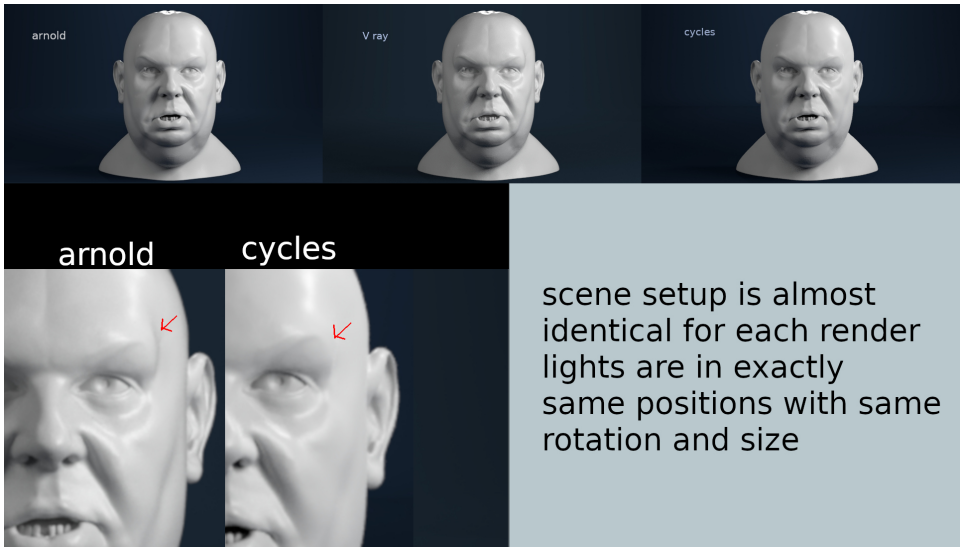
Renderovací programy

Pro vizualizaci materiálu potřebujeme trojrozměrný model, na který tento materiál umístíme. Potřebujeme tedy program, který načte a zobrazí model a poté na něj umístí materiál v požadované reprezentaci (BTF, SVBRDF) a zobrazí ho (vyrenderuje). Toto se většinou odehrává v modelovacích softwarech, kde přiřadíme objektu nebo jeho části materiál. Tyto programy obsahují renderery, které mají vlastní druhy materiálů typicky v proprietární reprezentaci a ty zobrazují.

Existují i samostatné renderery, které jsou zaměřeny hlavně na práci s materiály a jejich vizualizaci, například renderer Mitsuba [19], ale i renderer Cycles [20], který je základní součástí programu Blender. Mitsuba je velmi jednoduchý render, oblíbený hlavně ve vědecké komunitě, díky jeho jednoduché rozšiřitelnosti. Jeho nevýhodou je absence uživatelské rozhraní[1] a nemožnost editace 3D modelů. Existují i společnosti, které vyrábí vlastní proprietární renderovací program a zajišťují jeho podporu v různých modelovacích programech. Dříve byl v této kategorii velmi populární renderovací sofware V-Ray [21], za zmínku stojí i český renderer Corona [22], který je poslední dobou velmi populární. Jedním z nejvíce podporovaných rendererů největšími modelovacími programy je aktuálně Arnold renderer [23], který je používán v mnoha filmových studiích.

Proč vlastně existuje tolik rendererů, když existují fyzikálně popsané metody, které by měly fungovat všude stejně? Jednotlivé rendery se od sebe liší v různorodosti podpory materiálů a některé si vytváří i svoje vlastní verze. Také se liší místem kde probíhá výpočet, protože většina renderů používá

[1] Verze Mitsuby 2.0 neobsahuje grafické rozhraní, ovládá se pomocí konzole.

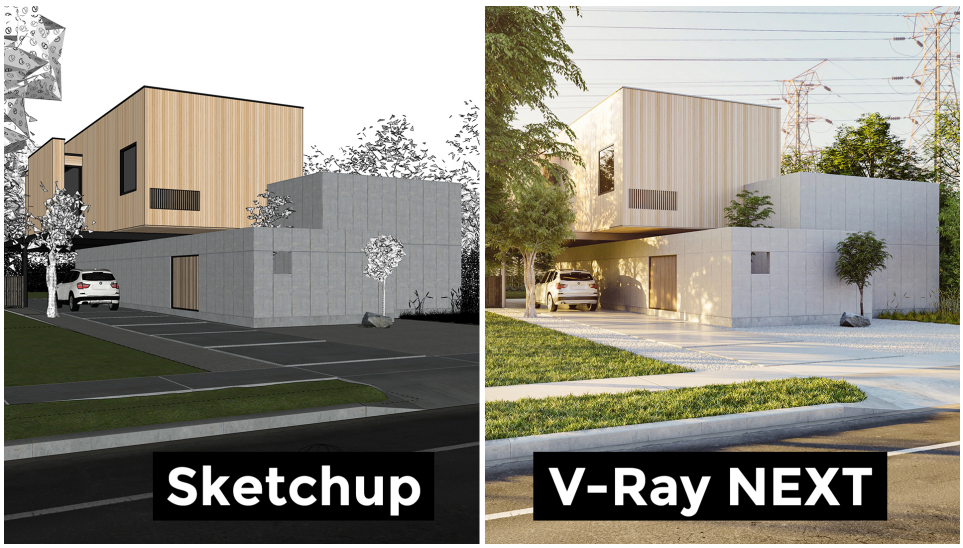


Obrázek 4.1: Porovnání výstupu rendererů Cycles, V-Ray a Arnold na stejné scéně, se stejnými světly. Obrázek byl převzat z [24].

pro výpočet procesor a nevyužívá grafické karty. Rozhodně velmi důležitým parametrem je čas rendrování stejné scény se stejnými materiály, kde každý renderer je vhodný pro jinou scénu. Renderery se také liší metodou, kterou používají pro samotnou realistickou simulaci (ray tracing, path tracing, ray tracing kernels, wavelength simulation). Pro někoho může být i důležitým parametrem, jestli renderer dokáže zobrazit odhad výsledku živě, případně jestli jde zrychlit rychlost výpočtu tím, že budeme hodnotu počítat jen pro nějaké pixely (biased/unbiased). Rozdíly mezi jednotlivými renderery pro většinu scén a materiálů nejsou velmi velké. Jde spíše o to jak dobře se s nimi pracuje a jaký mají renderovací čas. Na obrázku 4.1 můžeme vidět stejnou scénu vykreslenou renderery Cycles, V-ray a Arnold, přičemž materiály byly vytvořeny tak, aby byly téměř identické (renderery se liší v některých nastavení, takže není jednoduché dosáhnout stejného materiálu).

4.1 V-Ray

V-Ray [21] patří mezi nejznámějších fotorealistických renderery na světe. Jeho první verze vznikla v roce 1997. Je vyvíjen firmou Chaos Group. Má širokou podporu integrace do různých nástrojů. Na ukázku uvedu několik těch nejznámějších: Autodesk 3DsMax, Maya, Cinema4D, SkecthUp, Rhino, Revit, atd. Díky široké podpoře je používán k vizualizacím v mnoha oblastech, například: design, architektura, strojírenství, filmovém průmyslu a v dalších odvětvích.



Obrázek 4.2: Na obrázku, lze vidět porovnání výstupu ze základního Sketchup rendereru a výstupu z V-Ray renderu. Obrázek byl převzat z [25]

Jak je vidět na obrázku 4.2 výstup různých 3D programů dokáže velmi změnit použití různých rendererů a jejich materiálů. Samotné 3D programy většinou obsahují jednoduché renderery, tak aby dokázaly hlavně zobrazit 3D modely a umožňují přidání rendererů jiných firem pro zaručení fotorealistických výstupů. Tyto renderery jsou vyvíjeny jako samostatná aplikace a poté jim je dodávána podpora pro jednotlivé programy formou rozšíření (pluginů).

4.2 Corona Renderer

Corona Renderer je fotorealistický renderer, který byl vyvinut v České republice v roce 2009 studentem ČVUT Ondřejem Karlíkem. Tedy patří mezi mladší renderery. Postupem času se vyvinul v komerční projekt, který vedla trojice Ondřej Karlík, Adam Hotový a Jaroslav Křivánek. Pro výpočet používá procesor s možností využití grafické karty Nvidia pro redukci šumu. V roce 2017 vstoupil Corona renderer do skupiny Chaos group, ve které je i V-ray renderer. Aktuálně je distribuován jako samostatná aplikace nebo jako rozšíření pro 3ds Max a Cinema4D.

Tento renderer je proprietární a jeho zdrojový kód není k dispozici, ale uvedl jsem ho zde hlavně proto, že byl vyvinut v České Republice. Corona renderer byl použit hned v několika filmech například v Úsvit planety opic, Godzilla, v sérii Hobit a mnoha dalších filmech. Jeho výstupy jsou v ně-

Obrázek 4.3: Výstup z českého fotorealistického Corona rendereru. Obrázek byl převzat z [25].

kterých případech k nerozeznání od skutečnosti. Pro ukázku toho co tento renderer umí, jsem vybral obrázek 4.3[2].

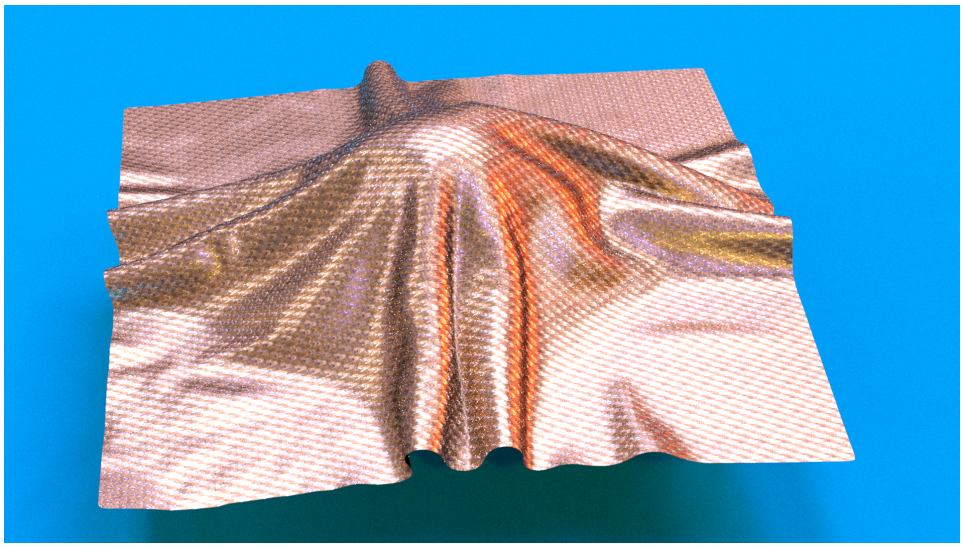
4.3 Arnold Renderer

Arnold Renderer [23] je propietární fyzikálně založený renderer, který vyvinula společnost Solid Angle. Používá metodu Monte-Carlo raytracingu. V programech od společnosti Autodeks (Maya, 3DsMax) nahradil dříve automaticky instalovaný *Mental Ray* od Nvidie, který však nebyl fyzikálně přesný. Tedy nyní se automaticky instaluje s tímto softwarem a má v těchto programech plnou podporu (při bezplatné verzi rendereru jsou obrázky překryty logem, pro plnou funkci je nutné zakoupení licence).

Tento renderer lze umístit i do programů Cinema4D, Houdini a Katana. Všechny zmíněné programy jsou hojně využívány ve filmovém průmyslu. Tedy není překvapením, že s využitím zmíněných programů a Arnold rendereru vzniklo a stále vzniká spousty filmů například – Strážci galaxie, Avangers a Thor.

[2] Více výstupů z Corona Rendereru můžete nalézt v galerii na jejich webových stránkách - https://corona-renderer.com/gallery

Jelikož jsou Vray, Corona renderer a Arnold renderer proprietární programy není přístupný jejich kód, a tedy přidání podpory BIG/MIF formátu pro tyto renderery není možné. Proto vám představím v dalších odstavcích renderery s veřejně přístupným zdrojovým kódem.



Obrázek 4.4: Výsledek renderování scény cloth s BTF texturou za pomoci bigbtf pluginu v Mitsuba rendereru. Scéna je nasvícena pomocí mapy prostředí.

4.4 Mitsuba

Mitsuba renderer [19], přesněji Mitsuba 2 je výzkumně orientovaný renderovací systém napsaný v portable C++17 s využitím knihovny Enoki [26]. Je vyvíjen v Realistic Graphics Lab at École polytechnique fédérale de Lausanne (EPFL). Na obrázku 4.4 můžete vidět výsledný render BTF textury pomocí pluginu, který jsem implementoval v rámci této práce.

Mitsuba renderer může být sestaven do mnoha variant, které zahrnují manipulaci s barvami (RGB, spektrální, monochromatické), vektorizaci (skalární, SIMD, CUDA) a diferencovatelné vykreslování. Mitsuba se skládá z malé množiny knihoven, které tvoří jádro programu a obsahuje mnoho rozšíření, které implementují široké množství funkcionalit od materiálu a světelných zdrojů po algoritmy tvořící celý renderovací proces. Jednotlivé verze Mitsuby jsou zpětně kompatibilní po verzi 0.6. [19]

Jelikož je Mitsuba výzkumně orientovaný software je jeho kód volně přístupný. Už odpočátku byla tvořena tak, aby byla lehce rozšiřitelná o nové

algoritmy a funkce. Tedy je velmi vhodná na vyzkoušení nových metod, předtím než budou implementovány do jiných např. aplikačně orientovaných programů.

Na základě těchto vlastností jsem si Mitsubu vybral pro počáteční implementaci podpory formátu pro reprezentaci BTF textur. Protože je BIG/MIF formát vyvíjen také v C++ a Mitsuba je velmi dobře rozšiřitelná nemělo by být přidání podpory tohoto formátu náročné a bude se zde dobře testovat jeho výkonost a použitelnost.

Obrázek 4.5: Ukázka vyrendrované scény Proxenta Residence od Martina Kovacika pomocí Cycles. Obrázek byl převzat z [27].

4.5 Blender - Cycles

Blender [28] je bezplatný open source 3D modelovací sofware, který podporuje celou 3D posloupnost – modelování, rigging, animaci, simulaci, vykreslování (*rendering*), kompozici a sledování pohybu a dokonce i úpravy videa a tvorbu her. Z těchto všech částí je pro tuto práci nejvíce zajímá část renderovací a konkrétně Blenderem vyvíjený renderer Cycles [20], který je open-source a jeho kód je volně přístupný. Je napsán v C++. Pro Blender byla již podpora BTF textur dříve přidána viz článek Haťky a Haindla [29], bohužel Blender v následujících verzích změnil systém pluginů a toto rozšíření podporující BTF textury zde tak přestalo fungovat.

Cycles je fyzikálně založený renderer, který je podporován několika modelovacími programy (Blender, Poser, Rhino, Cinema4D a 3Ds Max). Využívá raytracing a pro tvorbu materiálů využívá systém krabiček s určitými vlastnostmi, které tvoří výsledný materiál. Existují rozšíření pro Blender, které přidávají nové krabičky pro Cycles a tak rozšiřují jeho funkcionalitu. Cycles renderer lze používat buď přímo v Blenderu nebo je k dispozici i jako samostatný program. Ukázku výstupu z Cycles rendereru můžete vidět na obrázku 4.5.

Blender patří mezi renderery, kam by měla jít podpora MIF formátu celkem jednoduše přidat. Mohla by zde být hojně využívána, protože je dostupný zdarma a má velkou komunitu uživatelů. Existují dvě možnosti, jak tuto funkčnost přidat, buď jako add-on v Blenderu, který používá programovací jazyk Python a vytvořit tak novou možnost volby materiálu (nepovedlo se mi dohledat, jak získat potřebné informace, které potřebuji pro renderování BTF textur). Další možností je přidat tuto podporu přímo do renderu Cycles, který je napsán v C++, kde by mělo být jednodušší přidání, ale nevím jak by se pak dalo vytvořit uživatelské nastavení pro nastavení parametrů BTF textury a cestě k BTF textuře. Uživatelsky přívětivější by bylo tuto funkcionalitu přidat jako rozšíření (add-on) pro Blender, protože ty se dají volně instalovat, ale mohlo by zde dojít ke zpomalení renderovacího procesu, protože by se používal Python. Pokud by se zvolil druhý způsob, musela by být někde zveřejněna zkompilovaná verze Blenderu s upraveným rendererem Cycles.

4.6 PBRT

Physically Based Rendering: From Theory to Implementation [30] (PBRT) je renderer, který vytvořili Matt Pharr, Greg Humphreys a Wenzel Jakob současně se stejnojmennou knihou. Nyní je ve vývoji čtvrtá verze rendereru a knihy, která ho popisuje. Kódy jednotlivých verzí PBRT jsou volně přístupné na githubu. Ukázku vyrendrované scény pomocí čtvrté verze rendereru můžete vidět na obrázku 4.6. Pro PBRT renderer existují rozšíření do programů Blender, Maya, Houdini, atd. Tyto rozšíření umožňují převést scénu z těchto programů do scény, která se používá v PBRT rendereru. Vytvoření scény by tedy nemělo být obtížné. Tento renderer je společně s knihou určen k seznámení s renderovacím procesem a jeho algoritmy a používá se na několika světových univerzitách k výuce fyzikálně založeného renderovacího procesu.

Tento renderer má volně přístupný kód, který je podrobně popsán ve stejnojmenné knize a jsou zde popsány i postupy a myšlenky pro jednotlivé části tohoto rendereru. Bohužel se mi nepovedlo nalézt dokumentaci kódu ani možnost rozšíření tohoto rendereru pomocí pluginu. Jediná možnost by byla dopsat tuto podporu přímo do rendereru, což by na základě informací z knihy neměl být problém, ale z uživatelského hlediska by toto řešení nebylo velmi přívětivé.

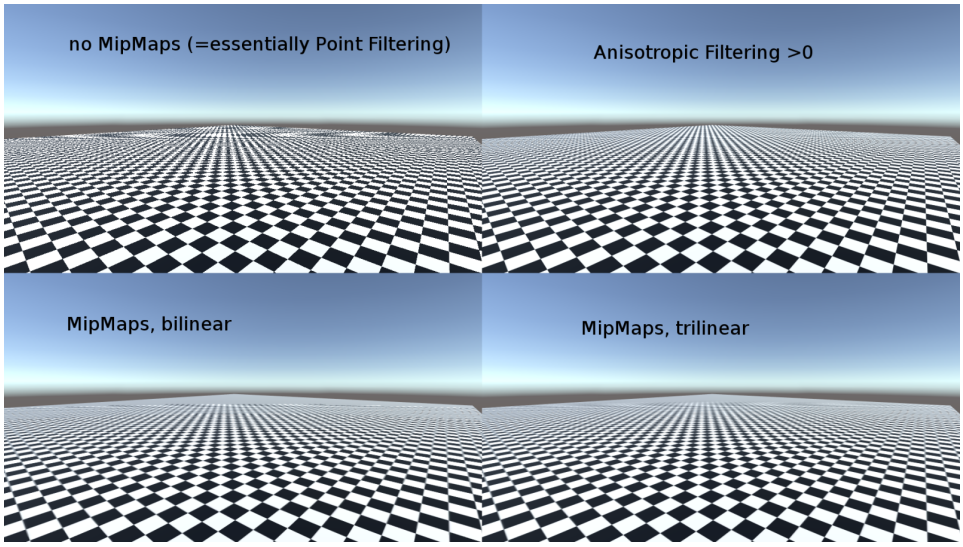
Na základě těchto faktů, jsem se rozhodl pro implementaci v rendereru Mitsuba, kde by mělo být přidání podpory formátu pro reprezentaci BTF textur nejjednodušší. Otestované a optimalizované části kódu z tohoto rendereru bude poté možno využít při implementaci v ostatních rendererech.

Obrázek 4.6: Ukázka vyrendrované scény v PBRT–v4 od San Miguel. Obrázek byl převzat z [31].

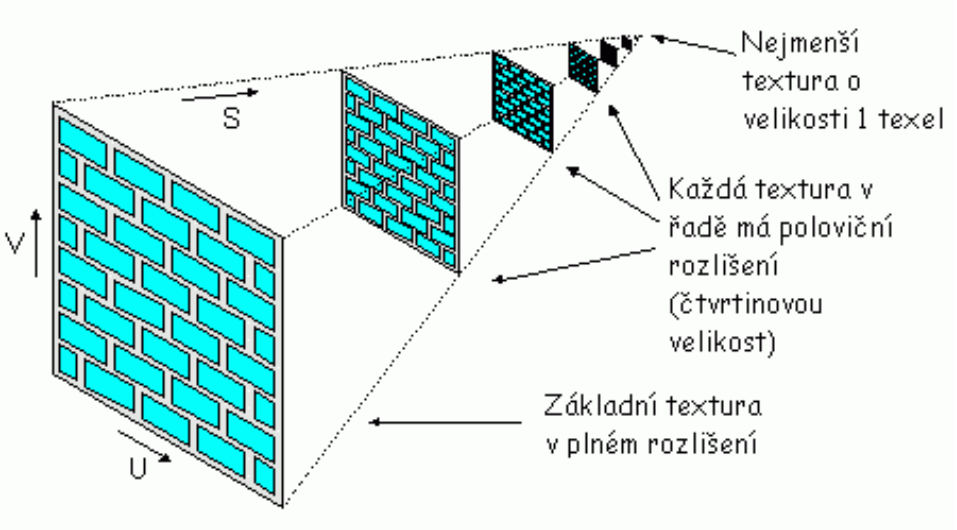
Kapitola 5

Filtrování textur

V této práci jsem v rámci vytváření rozšíření pro datový formát MIF pro Mitsubu implementoval i podporu pro filtrování textur pomocí mippmappingu a anizotropního filtrování. V následující kapitole bych rád popsal teorii ohledně implementovaných způsobů filtrování textur. Filtrování se používají pro zlepšení vizuálního výsledku renderingu a některé formy slouží k zrychlení výpočetního času. Mipmapping a anizotropní filtrování se snaží vyřešit problém aliasingu a vzniku moaré artefaktů na texturovaném objektu. Varianty, které zlepšují renderovací čas, většinou pracují na grafické kartě a využívají techniky úrovní detailu (*level of detail*).



Obrázek 5.1: Mipmaping s různými druhy filtrování, změna se odehrává na vzdálenějších texelech. Obrázek byl převzat z [32].



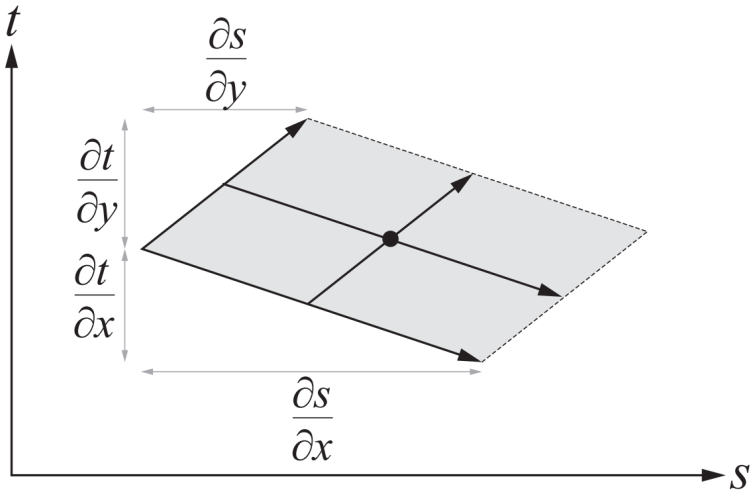
Obrázek 5.2: Mipmapa se ukládá vždy s poloviční velikostí, tedy vždy bude mít čtvrtinovou velikost a velikosti jsou násobky dvojky. Obrázek byl převzat z [33].

Jelikož použití textury může způsobovat vizuální nedostatky (moaré, aliasing), využívají se různé druhy filtrování, konkretně bilineární (prostorové filtrování v jedné textuře), izotropní (mipmapping) a anizotropní filtrování, které mohou tyto artefakty zmírnit nebo úplně odstranit. Anizotropní filtrování plně řeší všechny problémy, které vznikají na vzdálenějších texelech viz obrázek 5.1. Filtrování se také využívá pro zrychlení renderovacího času algoritmů a v rámci LOD (level of detail) technik.

U filtrování textur (mipmapping a anizotropní filtrování) se používá způsob uložení obrázků, kde nemáme uloženu texturu jen v jednom rozlišení, ale máme texturu v několika verzích rozlišení viz obrázek 5.2 (začínáme na jednom texelu a každá další úroveň má dvojnásobné rozlišení oproti předešlé). Takto uloženou texturu nazýváme mipmapa (ripmapa u anizotropního filtrování). Jaká úroveň mipmapy se bude renderovat rozhoduje velikost otexturovaného objektu na výsledné obrazovce. Menší rozlišení textury můžeme použít v případě, že objekt s texturou bude ve velkém sklonu k pozorovateli a tedy textura zabere opět jen několik pixelů na obrazovce. Výběr této úrovně je velmi složitý a blíže ho popíši v kapitole 5.1.

Filtrování lze provádět dvěma způsoby a to buďto offline nebo online. Online filtrování se provádí za běhu algoritmu. Mipmapa se při něm vygeneruje na grafické kartě při inicializaci renderingu a uloží se do její paměti. Lze tak zrychlit čas renderingu[1] a ušetřit paměť. Většinou se zde používají jed-

[1] Zrychlení se projeví hlavně u rendererů používající rasterizaci, protože bude potřeba zpracovávat menší množství pixelů textury.



Obrázek 5.3: Zobrazení zpracovávaného bloku (*footprint*) v prostoru textury. Ilustruje proměnné použité ve funkci $\rho(x, y)$. Obrázek byl převzat z [34].

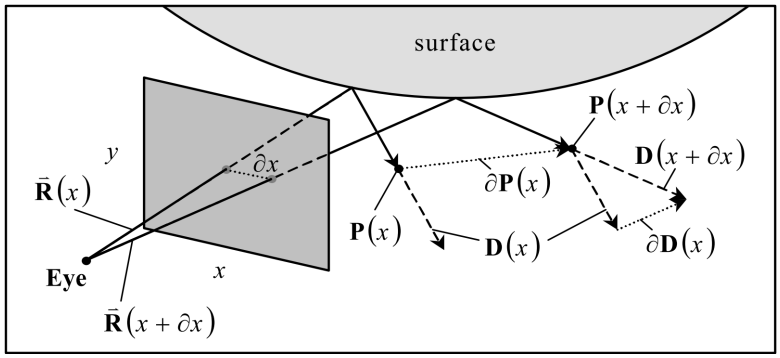
nodušší metody pro zmenšení obrázků (2x2 box filter), takže výsledek může mít mírně horší vizuální vlastnosti než při použití offline varianty filtrování. Offline filtrování znamená, že už samotná textura je uložena s jednotlivými mipmapami. Jeho výhodou oproti online variantě je, že nezpomalujeme renderovací čas generováním jednotlivých mipmapových úrovní a na zmenšení obrázku můžeme použít složitější metody. Takto vygenerované soubory si uložíme místo klasického obrázku textury. Hlavní nevýhodou je, že tyto textury zabírají více paměti.

5.1 Výběr úrovně mipmapy

Při použití filtrování potřebujeme určit jakou úroveň (*level*) mipmapy pro aktuálně zpracovávaný pixel použijeme. Potřebujeme zjistit jakou velikost bude textura zabírat a pro anizotropní filtrování i případnou deformaci textury. Výpočet úrovně se liší podle typu rendereru u rasterizátoru je postup, jednodušší než u rendererů které pracují s metodami sledování paprsku.

Postup získání úrovně v rendereru pracujícím s rasterizací popíši na ukázce z OpenGL. Používá se zde parameter λ, který je vyjádřen jako:

$$\lambda(x, y) = \log_2\lceil\rho(x, y)\rceil,$$



Obrázek 5.4: Obrázek ilustruje polohy a směry hlavního a diferenčního paprsku. Rozdíl mezi nimi je paprskový diferenciál $\partial\mathbf{P}$. Obrázek byl převzat z [35].

kde (x, y) jsou souřadnice pixelu a funkce ρ, má tuto definici:

$$\rho(x,y) = \max\left\{\sqrt{\left(\frac{\partial s}{\partial x}\right)^2 + \left(\frac{\partial t}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial s}{\partial y}\right)^2 + \left(\frac{\partial t}{\partial y}\right)^2}\right\}.$$

Funkce $\rho(x, y)$ se používá pro dvourozměrné vyhledávání textur, kde (s, t) jsou souřadnice texelu, tj. souřadnice textury ($\in [0, 1]^2$) vynásobené rozlišením textury. Tato funkce zajišťuje, aby vzorkování hierarchie mipmap mapovalo pixely obrázkového prostoru na jeden texel. [34]

Obecně probíhá výpočet na grafické kartě pomocí diferenciálů, které vždy vyhodnocuje pixel shader přes pole pixelů o velikosti 2×2 s využitím per-pixel rozdílů. Ukázku zpracovaného bloku, někdy nazývaného stopa (*fingerprint*) můžete nalézt na obrázku 5.3. Samozřejmě existují i jiné přístupy i samotné OpenGL umožňuje zvolit pro implementaci funkce ρ jiný postup. [34]

K určení úrovně se při použití metod sledování paprsku používají diferenciály paprsku (ray differentials). K jejich získání je potřeba k hlavnímu paprsku přidat další dva pomocné paprsky, z nichž jeden je posunut od hlavního paprsku v ose x a druhý paprsek v ose y. Jejich přidáním naroste náročnost výpočtu raytracingu, tedy nám naroste výpočetní čas. Naštěstí se pro tyto paprsky nemusí počítat všechny výpočty jako pro hlavní paprsek, hodně vlastností z tohoto paprsku dědí. Tyto pomocné paprsky se použijí při odrazu tělesa. Diferenciál je vlastně vzdálenost pomocného paprsku od hlavního paprsku po odraze viz obrázek 5.4. Výsledné diferenciály se získávají pomocí aproximace, konkrétněji pomocí aproximace Taylorova polynomu prvního řádu. Pro zvýšení přesnosti by jsme mohli aproximovat polynom vyššího řádu, ale výpočet by byl o dost náročnější. [35]

Na obrázku 5.4 ze článku Tracing Ray Differentials [35] můžeme vidět nákres popisující výpočet differenciálu pro osu x. Kde $\overrightarrow{\mathbf{R}}(x)$ je hlavní paprsek

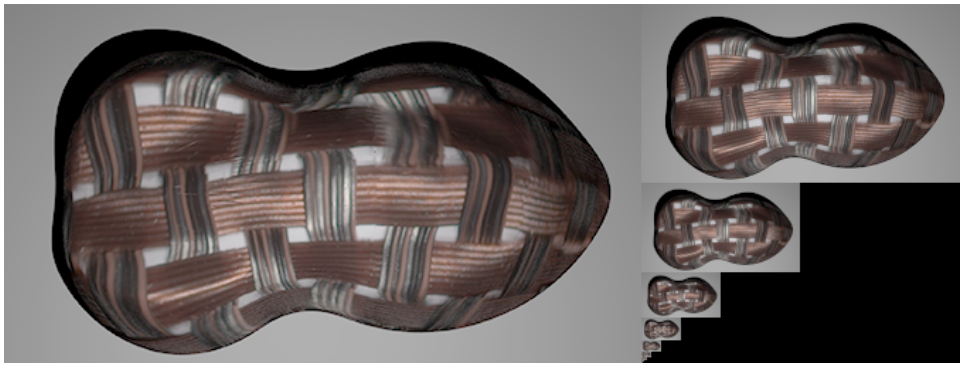
a $\overrightarrow{\mathbf{R}}(x + \partial x)$ je pomocný paprsek pro výpočet diferenciálu v ose x posunutý o danou vzdálenost (typicky jeden pixel). ∂x je rovna vzdálenosti jednoho pixelu (pokud raytraycer má pro jeden pixel více paprsků, může tato vzdálenost být menší) a $\partial\mathbf{P}$ je výsledný differenciál. Tento differenciál je poté potřeba vyjádřit v bázi obrazu (pozorovatele)[2], aby jsme získali prostorový diferenciál, který se používá pro výpočet úrovně mipmapy. Na obrázku 5.4 lze vidět, že v čím větším sklonu bude texturovaný objekt k pozorovateli, tím bude diferenciál větší, kvůli této vlastnosti se diferenciály pro výpočet úrovně používají.

Pro výpočet diferenciálů existuje mnoho metod, které se ho snaží optimalizovat. Jelikož Mitsuba má výpočet prostorových diferenciálů implementovaný, nebudu ho zde popisovat detailněji. Ale uvedu zde tři zdroje, kde je výpočet velmi dobře a podrobně popsán. Prvním je článek Tracing Ray Differentials z roku 1999, který napsal Homan Igehy a je velmi často citován v ostatních článcích, které se touto tématikou zabývají. Homan Igehy v něm popisuje základní princip výpočtu diferenciálů paprsku pro metodu sledování paprsku. Druhým je nedávná kapitola o samplování a antialiasingu z Physically Based Rendering 3rd Edition [36], konkrétně z její online verze. Třetí článek je Mipmapping with Bidirectional Techniques [37], který vychází z předchozího zdroje a popisuje i další možné metody pro výpočet prostorových diferenciálů i samotný výpočet úrovně pro mipmapu. Většina rendererů má tento výpočet již implementován a předá nám hodnoty těchto diferenciálů.

Pro výpočet úrovně potřebujeme prostorové diferenciály v souřadnicovém prostoru obrazovky. Samotné diferenciály pro výpočet nestačí, protože při jejich výpočtu se vůbec nepracuje s mipmapou, počtem jejích úrovní a její velikostí. Vlastnosti mipmapy jsou zahrnuté až při výpočtu samotné úrovně, kde se musí zohlednit.

Samotný výpočet úrovně se v různých rendererech provádí různě, ale vždy se používá maximální hodnota z hodnot prostorových diferenciálů v obou osách, jejíž dvojkový logaritmus se odečítá od počtu úrovní v mipmapě. Případně se můžou provádět úpravy prostorových diferenciálů před výpočtem jejich maxima. U anizotropního filtrování se musí počítat dvě úrovně, aby jsme mohli vybrat správně zdeformovanou mipmapu a tedy se berou maxima vždy z prostorových diferenciálů pro osu x nebo pro osu y. Konkrétně v Mitsubě jsem musel vypočítané prostorové diferenciály ještě vynásobit velikostí největší úrovně v mipmapě, abych dostával správné hodnoty úrovně.

[2] V Mitsubě se proměnné s prostorovými diferenciály v prostoru obrazovky označují duv_dx duv_dy.

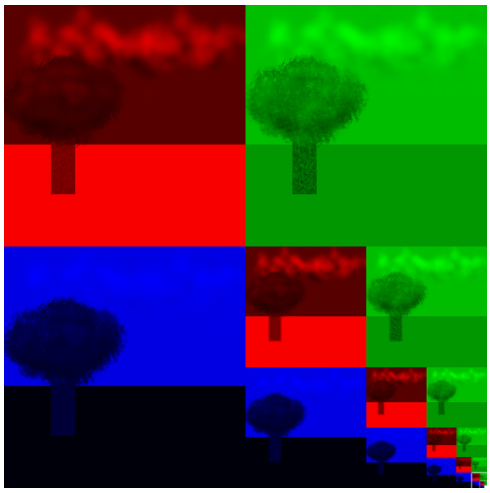


Obrázek 5.5: Varianta mipmapy, která zabera 150 % velikosti původního obrázku.

5.2 Mipmapping (izotropní filtrování)

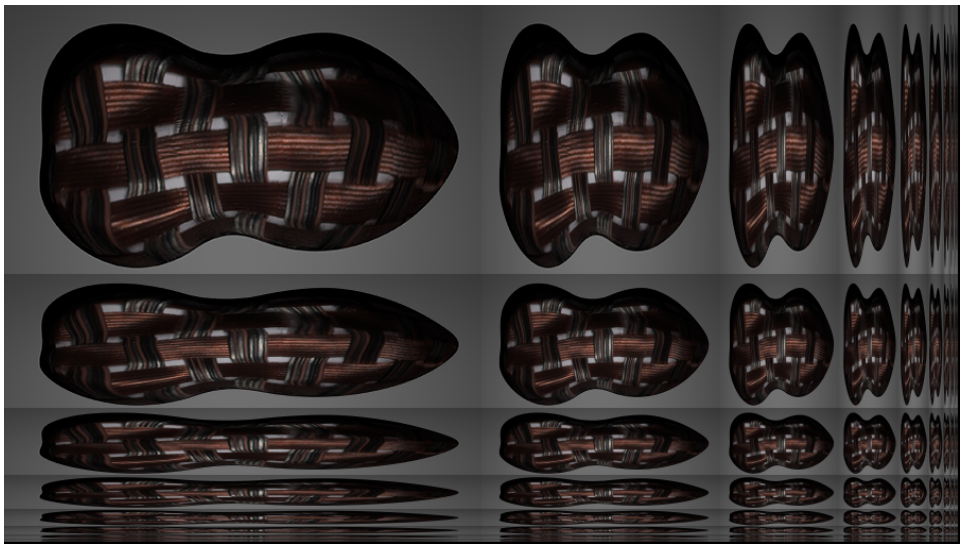
Izotropní filtrování (mipmapping) je způsob využití mipmapy při renderování. Používá se zde mipmapa, která se skládá z obrázků vždy z o polovinu menší velikostí. Renderer vypočítává úroveň mipmapy pro každý pixel a tuto úroveň použije pro získání pixelu.

V optimální případě izotropní mipmapa zabere 33 % viz obrázek 5.6. Ovšem izotropní mipmapa může zabírat více jak ideálních 33 % paměti. Pokud nebudeme obrázek/texturu ukládat po jednotlivých barevných kanálech, tak nám uložení jeho izotropní mipmapové verze textury zabere 150 % původní velikosti textury viz obrázek 5.5. Použití offline mipmapingu může znamenat zpomalení renderovacího času, protože naroste velikost obrazového souboru s texturou.



Obrázek 5.6: Uložení mipmapy po jednotlivých kanálech. Zvětší soubor o 33 procent. Obrázek byl převzat z [38].

Při výpočtu úrovně získáváme číslo s desetinnou čárkou, které buďto můžeme zaokrouhlit nebo provést lineární interpolaci mezi dvěma hodnotami (trilineární filtrování). V případě samotného zaokrouhlení můžou být vidět hranice mezi jednotlivými úrovněmi, kterou odstraní použití lineární interpolace mezi dvěma úrovněmi mipmapy. Samotný mipmapping nedokáže texturu dostatečně vyhladit, proto se velmi často používá společně s anizotropním filtrováním, které dokáže obraz vyhladit dokonale.



Obrázek 5.7: Ukázka podoby anizotropní mipmapy (ripmapy) na klasickém obrázku. Můžeme zde vidět, jak je textura deformována v jednotlivých osách.

5.3 Anizotropní filtrování

Anizotropní filtrování se využívá v počítačové grafice pro zlepšení kvality textur s kosými zornými úhly. Bere v potaz úhel pohledu, kde projekce textury působí nekolmým dojmem. Stejně jako bilineární a trilineární filtrování eliminuje anizotropní filtrování aliasing efekty, ale oproti nim vylepšuje redukci rozostření a zachovává vysoký detail i v extrémních zorných úhlech viz obrázek 5.1.

Používá se jako standardní funkce grafických karet již od devadesátých let dvacátého století. Možnost volby anizotropního filtrování by tedy měla patřit mezi první volby pro rozšíření při podpoře nového materiálu. Anizotropní filtrování se používá společně s mipmapou a odstraňuje zubatost nebo pixelovatost vzdálenějších objektů v kosém zorném úhlu.

Anizotropní filtrování sleduje danou texturu v různých bázích jednotlivých pixelů a různých orientacích anizotropie. Toto sledování (sondování) probíhá neustále během běhu algoritmu.

Anizotropní filtrování jde také implementovat pomocí předpočítaných anizotropních mipmap (ripmap), které jsou zdeformovány v osách x a y v různých poměrech. Takováto forma má nevýhodu vysoké paměťové náročnosti, protože výsledná ripmapa zabere čtyřikrát více paměti oproti původnímu souboru z texturou. Samozřejmě i zde existují offline (předpočítaná mipmapa v původním souboru) i online (implementováno na grafické kartě) verze takového použití. Přičemž použití offline řešení je zde velmi paměťově náročné, konkrétně při vygenerování anizotropní mipmapy pro BTF texturu o velikosti čtyři gigabajty bude mít její ripmapa velikost šestnáct gigabajtů.

Na obrázku 5.7 můžeme vidět ukázku anizotropní mipmapy. Na diagonále této ripmapy jsou jednotlivé úrovně mipmapy používané pro mipmapping, tedy obrázky zmenšené v obou osách o polovinu. Pod a vpravo od původního obrázku jsou úrovně, které se zmenšily jen v jedné ose o polovinu a došlo tedy nejen ke zmenšení jejich rozlišení ale i k jejich deformaci. Nejmenší úroveň v jednotlivých osách tvoří pruh široký jeden pixel.

Kapitola 6

Implementace

Podporu nového formátu pro reprezentaci BTF textur jsem se rozhodl naprogramovat nejdříve pro Mitsuba renderer, kde by mělo být díky možnosti vytvoření pluginu nejjednodušší tuto podporu přidat. Další její výhodou je velmi dobře zpracovaná dokumentace, která pomůže z orientací v kódu a také se sestavením celého renderu s mým rozšířením, které jsem pojmenoval bigbtf. Cycles má o poznání horší dokumentaci a tedy jeho rozšíření a sestavení by bylo mnohem náročnější, nejspíše by bylo potřeba překompilovat celý Blender s upraveným Cycles, což by nebylo uživatelsky moc přívětivé.

Jelikož Cycles i Mitsuba jsou napsány v C++, rozhodl jsem se podporu BIG/MIF formátu a komunikaci s jeho knihovnou implementovat v samostatné třídě s názvem **BigRender**. Tato třída bude zajišťovat čtení dat ze souboru, pro paprsek s určitými parametry. Poté jen získám informace o paprsku, které jsou potřeba pro získání dat ze souboru, tyto parametry předám dané třídě a ta zařídí přečtení těchto dat z datového formátu. Tato třída by měla implementovat i podporu mipmappingu a případné další rozšíření. Další možností je implementace mipmappingu a anizotropního filtrování jako součásti nového formátu pro uložení BTF textur (BIG/MIF), protože tyto rozšíření se týkají i podoby uložení BTF textury.

6.1 Struktura BIG formátu

BIG formát umožňuje uložit jedno 4D pole dat v libovolné permutaci, přičemž každý 3D řez (*slice*) může mít libovolný datový formát z dané množiny. BIG formát je vhodný pro uložení velkého množství stejně velkých obrázků, přičemž je možné data přeuspořádat tak, aby umožňovala rychlé renderování nebo faktorizaci atd.

V tomto odstavci bych rád vysvětlil jak v BIG knihovně funguje výše zmiňovaná cache paměť. Při čtení pixelu z nějakého obrázku (v kontextu BIG formátu o něm hovoříme jako o entitě) si tuto entitu (obrázek) uložíme do cache (RAM) paměti, jejíž velikost určujeme při načítání BIG souboru. Předpokládáme, že budeme číst i další vedlejší pixely této entity. Pro bitové velikosti (8, 16, 32, 64) jsou zvlášť vytvořené pole kde jsou tyto entity uložené[1]. Vlastně máme pro každou bitovou velikost vytvořenou indexovou mapu.

Společně s entitami si také ukládáme čas uložení, posledního použití, na základě kterého pak případně entitu smažeme. Dále pro každou bitovou délku máme frontu (implementovanou pomocí listu), kde si ukládáme index v polích entity pro každou uloženou entitu. Tyto fronty jsou potřeba, protože chceme vždy z cache paměti odstranit nejdéle nepoužívanou entitu, ale to nemusí být první entita uložená v poli. Pro vložení entity slouží funkce `pull` a pro odebrání funkce `pop`. Tyto funkce mají vždy vlastní pole pro x-bitové parametry. Mezi těmito poli musíme najít podle fronty nejdéle nepoužívaný prvek a ten případně nahradit novým prvkem. Tento postup nám zajistí, že nepřekročíme uživatelem určenou maximální velikost cache paměti. Ale při malé velikosti takto zadané paměti, zabere režie kolem správy fronty velké množství času a přístup k datům může být pomalejší než při čtení přímo z disku.

V aktuální verzi podporuje XML soubor, který má stejné jméno jako soubor s koncovkou .big. Do XML lze přidávat další informace o uložených datech, pro které neexistuje parametr v hlavičce, například distribuci obrázků. V budoucnosti by měl být tento soubor přidán přímo do BIG souboru, tak aby se stal jeho součástí.

[1] Nemůžeme používat jedno pole, protože by jeho indexování bylo složité.

6.2 Multi-Image Format - MIF

Multi-Image Format (MIF) umožňuje uložit libovolné množství obrázků, přičemž každý obrázek může mít libovolné rozměry včetně libovolného množství rovin/kanálů (šedotónových, RGB, spektrálních) a libovolný datový formát z dané množiny. MIF díky dodatečným informacím v XML umožňuje uložit data z více měření do jednoho souboru společně s dalšími informacemi z měření, případně o mipmapách, atd. Zároveň je navržen s ohledem na rychlé renderování.

Na MIF jsem přešel z důvodu problémů při použití částečné cache[2] a potřeby dalších funkcionalit, které BIG formát a jeho knihovna neobsahovaly. Jednou ze změn bylo rozhodnutí, že XML data, které budou součástí nového formátu se budou načítat do struktur/tříd. Toto řešení je pro uživatele přívětivější a uživatel vlastně nepotřebuje vůbec pracovat s XML daty (načítat a ukládat a znát jejich strukturu). Dalším problémem byla absence funkce `getPixel`. V BIG knihovně se muselo vždy žádat o hodnotu každého barevného kanálu zvlášť, což zvětšovalo režii a prodlužovalo renderovací čas. Také u ní byl problém s částečným použitím cache paměti a její implementací, která nesprávně pracovala s vícevláknovým přístupem jak popisuji v kapitole 6.9.2.

Přidat všechny tyto úpravy do aktuální verze BIG knihovny by bylo velmi pracné. Proto se rozhodlo, že pan Radomír Vávra z ÚTIA AV ČR napíše novou verzi formátu BIG, která kromě zmíněných změn bude podporovat uložení více souborů a vytvoří podporu pro mipmapping a anizotropní filtrování. Dále je knihovna navržena tak, že případné nové informace o materiálu se budou moci uložit do XML souboru, ze kterého se poté načtou do tříd. Z důvodu odlišení od původní verze BIG formátu dostal tento upravený formát název Multi-Image Format (MIF) [18]. Vývoj této knihovny stále probíhá. Všechna BTF data jsou v současné době uložena v datovém typu float. Knihovna prozatím obsahuje podporu pouze pro tento datový typ, ale kód je připravený na podporu všech datových typů, které podporuje BIG formát.

6.2.1 Důležité třídy MIF knihovny pro mojí implementaci

MIF knihovna obsahuje tři druhy cache a to `CacheNone`, `CacheWhole` a `CachePart` (ve vývoji), které se přiřazují jednotlivým BTF texturám uloženým v MIF

[2] V BIG knihovně nešlo vypnout používání částečné cache, což MIF knihovna umožňuje.

souboru. `CacheNone` slouží pro čtení z disku, `CacheWhole` načte celý soubor do paměti a `CachePart` je zatím ve vývoji a měla by pracovat částečně s pamětí na podobném principu jako BIG knihovna. Použití `CachePart` v pluginu pro Mitsubu bylo zavrhnuto, protože kvůli vícevláknovému běhu by mohlo docházet k chybám a její vývoj může trvat velmi dlouho. Po dokončení by tato cache mohla být do kódu třídy `BigRender` zařazena.

MIF knihovna obsahuje třídu pro mipmapu s názvem `ElementMipmap` a `ElementMipMapItem`. Třída `ElementMipmap` obsahuje v případě mipmappingu 1D pole třídy `ElementMipmapItem` s názvem isotropic_items a pro anizotropní filtrování 2D pole s objekty `ElementMipmapItem` s názvem anisotropic_items a metody k přístupům k těmto polím, jejich velikostem a další metody potřebné pro ukládání a načítání těchto dat z XML. Výhodou tohoto řešení je, že si tyto data načteme ze souboru a nemusíme si je počítat při renderování.

Třída `ElementMipMapItem` obsahuje informace o své úrovni, tedy pro mipmapping `level` a pro anizotropní filtrování parametry `level_x` a `level_y`, poté informace o levém horním rohu obrázku na daném levelu `x` a `y` a rozměry obrázku na dané úrovni `cols` a `rows`. Na struktuře těchto elementů jsem s panem Vávrou spolupracoval. Informace o počátku a velikosti by zde nemusely být, daly by se dopočítat na základě toho, že víme, že se původní obrázek vždy zmenšoval o polovinu. Řešení použité v MIF knihovně umožní různé implementace pro ukládání a zabrání chybě při čtení těchto dat.

Informace o jednotlivých krocích po hemisféře při měření[3] a druhu reprezentace dat se ukládají do tagu directions, konkrétně do jeho atributů. V directions jsou také uloženy úhly kamery a světla, v kterých byl daný obrázek naměřen. Tyto informace nejsou přímo potřeba pro renderovaní, ale jsou užitečné pro případnou analýzu uložených dat. XML může také obsahovat tag s informacemi o materiálu, měřícím zařízení (senzoru), barvách a měřené ploše (*area*). V ukázce kódu 2 můžete vidět strukturu XML při používání mipmappingu.

6.2.2 Přínos MIF formátu pro třídu BigRender

Při přechodu na MIF formát se rendrovací čas zrychlil cca o 10–15 % při renderování scén bez použití mipmappingu a anizotropního filtrování. U ostatních scén nemohu zrychlení srovnat, protože filtrování jsem implementoval až s použitím MIF formátu. Hlavním důvodem zrychlení byla funkce `getPixel`,

[3] Tedy o kolik stupňů se snímač a/nebo světlo posouvaly během měření.

```
<?xml version="1.0"?>
<mif>
  <btf id="btf0">
    <name>Uniform</name>
    <material reference="" />
    <device reference="" />
    <area reference="" />
    <colors reference="" />
    <mipmap type="isotropic">
      <item level="0" x="0" y="0" cols="100" rows="100" />
      <item level="1" x="100" y="0" cols="50" rows="50" />
      <item level="2" x="100" y="50" cols="25" rows="25" />
      <item level="3" x="100" y="75" cols="12" rows="12" />
      <item level="4" x="100" y="87" cols="6" rows="6" />
      <item level="5" x="100" y="93" cols="3" rows="3" />
      <item level="6" x="100" y="96" cols="2" rows="2" />
      <item level="7" x="100" y="98" cols="1" rows="1" />
    </mipmap>
    <directions type="list" name="Uniform" stepTheta="15.000000"
   stepPhi="30.000000"  units="deg" reference="directions0" count="3660" />
  </btf>
  <directions type="list" id="directions0" name="Uniform"
    stepTheta="15.000000" stepPhi="30.000000" units="deg" count="3660">
    <directions-item index="0">
      <direction index="0" type="source" theta="0.000000" phi="180.000000" />
      <direction index="1" type="sensor" theta="15.000000" phi="0.000000" />
    </directions-item>
  </directions>
</mif>
```

Kód 2: Ukázka struktury XML souboru v MIF

která při načtení dat do paměti vrací rovnou ukazatel na data uložená v paměti. Nedochází tedy k jejich kopírování a navíc načteme rovnou celý pixel a nemusíme ho číst po jednotlivých složkách jak tomu bylo v BIG knihovně.

Další výhodou použití MIF knihovny je zpřehlednění kódu, protože tato knihovna poskytuje třídy pro různá data uložená v MIF souboru. Proto nebylo potřeba vytvářet tyto třídy v rozhraní `BigRender` a jen si je pomocí knihovny MIF načíst do daného objektu. Což hlavně u mipmappingu a anizotropního filtrování velmi zlepšilo přehlednost kódu ve třídě `BigRender`, kde stačí vytvořit objekt třídy `ElementMipmap` a inicializovat ho pomocí funkce `getMipmap()` třídy `ElementBtf`. V tomto objektu jsou poté uloženy všechny informace o mipmapě potřebné k renderingu.

Jedinou nevýhodou použití MIF knihovny je, že se musí rozdělit čtení pro paměť a pro disk. I když pan Vávra po mé připomínce vytvořil abstraktní

třídu `CacheBase` s funkcí `getPixel`, bohužel tato třída nemůže obsahovat variantu této funkce, která by četla data přímo z paměti a vracela jen konstantní ukazatel na začátek pole, protože pro čtení z disku by se nedala použít. Použití této abstraktní třídy jsem zkoušel, ačkoliv ušetřilo několik řádek kódu a zpřehlednilo ho, rendering na testovací scéně se zpomalil průměrně o pět procent, proto jsem se rozhodl nechat rozdělení na dva druhy čtení a nevyužít abstraktní třídy `CacheBase`. Více informací o práci s MIF knihovnou ve třídě `BigRender` se dočtete v kapitole 6.6 „Třída BigRender“.

6.2.3 Můj podíl na vývoji knihovny MIF

Na vývoji MIF knihovny jsem se podílel na jejím testování, reimplementaci XML parsování a návrhem virtuální metody `getPixel`. Na vývoji MIF formátu jsem se podílel hlavně diskuzí o struktuře XML. MIF formát začal vznikat v dubnu, kdy už jsem měl zprovozněný plugin v Mitsubě. Testoval jsem tedy jednotlivé verze MIF knihovny přímo na scénách v Mitsubě a navrhoval úpravy knihovny a struktury formátu. Testoval jsem také rychlost jednotlivých implementací funkce `getPixel`.

U XML jsme vedli diskuzi, jestli všechny tagy musí být párové, nebo jestli u některých nebude lepší zvolit nepárový tag a data uložit do atributů tohoto tagu, což je efektivnější, protože některé tagy už nelze vnořovat. Například víme, že XML reprezentace třídy `ElementMipMapItemu` už se zanořovat nebude, díky tomu můžeme všechny jeho proměnné uložit jako atributy XML tagu.

Například pokud se do XML souboru neuloží pro každý obrázek směr (*direction*), tak při načítání souboru nelze zjistit počet obrázků, protože se určují podle počtu objektů v tomto poli. Jak je vidět v ukázce kódu 2 v tagu directions se ukládá typ interpolace pro daný BTF soubor. Při jeho čtení se v první verzi musely načíst všechny směry. Jelikož parsování XML dat do C++ tříd nebylo zrovna nejrychlejší, narostl čas pro inicializaci renderingu. Po upozornění na tento fakt byla vytvořena třída `ElementDirectionsInfo`, která obsahuje atributy z tagu directions bez potřeby číst všechny elementy, což urychlilo inicializaci renderování.

V MIF formátu jsem implementoval parsování dat z XML souboru pomocí pugixml knihovny [39]. Pan Vávra původně v kódu používal knihovnu Qt [40], která je velmi rozsáhlá a pro použití v MIF knihovně nebyla vhodná. Jelikož jsem měl s knihovnou pugixml zkušenosti, poprosil mě, jestli bych parsování

dat nepřepsal do této opensourcové knihovny, která se skládá jen ze třech zdrojových souborů.

6.3 Implementace v Mitsuba rendereru

V této kapitole popíši návrh implementace podpory vizualizace BTF textur v rendereru Mitsuba, jak tento renderer nainstalovat a strukturu některých částí jeho zdrojového kódu. Jelikož nemá Mitsuba 2.0 uživatelské rozhraní, popíši zde i základní ovládání programu a jeho nastavení. Samotná implementace v Mitsuba rendereru by měla proběhnout pomocí pluginu, pro který má Mitsuba podporu a který dokáže lehce nastavit požadované parametry pro renderování BTF textur (cesta k souboru, filtrování, velikost textury, …).

Plugin pro vizualizace BTF textur by měl dokázat číst data z MIF formátu, rozeznat jejich různé distrubuce a poznat jestli jsou v těchto datech uloženy mipmapy nebo anizotropní mapy (ripmapy). Plugin musí tyto informace získat ze třídy `BigRender`, která bude implementovat jednotlivé druhy filtrování (mipmapping a anizotropní filtrování), interpolování a získávání RGB pixelu. Naopak v pluginu musí jít nastavit typ filtrování (mipmapping, anizotropní, žádné), velikost textury, použití cache paměti a cesta k souboru s BTF daty. Tyto informace se předají univerzální třídě `BigRender` a ta na základě těchto informací vybere metodu pro zpracování pixelu.

6.3.1 Instalace Mitsuby

Jelikož je Mitsuba [19] open-source projekt zaměřený na vědeckou komunitu, není k dispozici instalátor samotného rendereru. Její zdrojové soubory jsou k dispozici na githubu[4]. Obsahuje velké možností konfigurací pro její sestavení, které lze měnit pomocí souboru mitsuba.conf. Lze změnit používanou reprezentaci barev, způsob výpočtu raytraingu (scalar, packet, gpu), polarizaci a přesnost datového typu. V následujících odstavcích budu popisovat návody a práci s Mitsubou na Windows s Visual Studiem 2019, které je pro Windows doporučené.

Konfigurační soubor Mitsuby není potřeba měnit, ale pokud chcete používat bigbtf plugin, doporučil bych změnit na řádku osmdesát základní repre-

[4] https://github.com/mitsuba-renderer/mitsuba2

zentaci barev ze scalar_spectral na scalar_rgb, protože jinak bude plugin dávat špatné výsledky. Případně by se musely všechny scény volat s parametrem -m scalar_rgb, který zaručí použití reprezentace barev pomocí RGB.

Mitsuba má velmi dobře popsanou dokumentaci, ale i tak je instalace trochu náročnější. Na Windows vyžaduje stažení Cmake [41], Visual Studia [42] a pythonu a některých jeho balíčků (pytest, numpy a sphinx), bez kterých nejde zkompilovat.

Pomocí Cmake se vygeneruje projekt pro Visual Studio, ve kterém poté můžete sestavit samotný renderer. Poté je potřeba nastavit systémové proměnné a nastavit zde cesty pro Mitsubu, aby mohla správně fungovat. Bez těchto proměnných Mitsuba nebude správně pracovat, protože některé knihovny budou pracovat špatně z cestami. Další důležitá věc, která v dokumentaci chybí, je že spustitelná verze Mitsuby se nachází v adresáři dist i se všemi přiloženými knihovnami a pluginy. Upozorňuji na to, protože jsem omylem zkoušel pustit mitsuba.exe ze src, kde chyběly některé dll soubory a pluginy. Tato verze se vytvoří po sestavení projektu INSTALL nebo ALL_BUILD. I když Visual Studio ukáže některé chyby při sestavení, nezalekněte se, Mitsuba se nejspíše správně sestavila. Jelikož docházelo k přechodu ze staré verze na verzi 2.0 zůstaly v nové verzi nějaké kódy, které nemusí být úplně funkční. Například Cmake nabízí možnost sestavení s GUI, ale tato verze nejde sestavit, protože jak jsem se dočetl na githubu pro novou verzi nebylo grafické rozhraní zatím implementováno.

Pro Windows je vytvořený skript (.bat) pro přidání Mitsuby do proměnných systému, aby poté šla spustit v příkazovém řádku kdekoliv zavoláním příkazu *mitsuba*. Tento skript mi nefungoval musel jsem tedy přidat cestu ke složce dist, v které je umístěná spustitelná verze Mitsuby ručně přes ovládací panely (systém a zabezpečení–systém–upřesnit nastavení systému–proměnné prostředí–Path a zde přidat cestu k umístění mitsuba.exe). Tento krok není povinný, ale ulehčí práci s programem, protože nebudeme muset vždy v konzoli zadávat cestu k mitsuba.exe nebo přecházet do složky s aplikací.

6.3.2 Práce s Mitsubou

Mitsuba nemá uživatelské rozhraní, proto se program spouští z příkazového řádku konzole a ovládá se pomocí parametrů, které mohou změnit chování rendereru. Scény v Mitsuba rendereru jsou popisovány v XML formátu. Tedy práce s tímto programem a jeho rozhraním je pro začátečníka dost náročná.

V XML formátu se nastavují i materiály jednotlivých objektů. Cílem této práce je do Mitsuby přidat vlastní materiál. Neboli pro distrubiční funkci obousměrného rozptylu BSDF a její tag přidat vlastní typ a jeho implementaci (`<bsdf type="bigbtf">`), která bude pracovat s BTF texturami. BSDF tag s parametrem type=bigbtf poté půjde přiřadit k objektu ve scéně. Pro vykreslení vybrané scény stačí zadat do příkazového řádku příkaz *mitsuba nazevsceny.xml*. Samozřejmě BSDF tagy mohou mít ještě další vnořené tagy, kde se můžou nastavovat vlastnosti bsdf například cesta k souborům nebo použití cache paměti, které jsou pro můj plugin (bigbtf) popsány níže v kapitole 6.3.8 „Implemementace bigbtf pluginu pro Mitsubu“.

6.3.3 Formát scény

Jak jsem uvedl dříve pro tvorbu scény se v Mitsubě používá XML formát. Pro mou implementaci budu muset vytvořit scénu a k objektu ve scéně přidat vlastní typ tagu BSDF (bigbtf), který definuje jak paprsky reagují s plochou objektu. Vlastně určuje obousměrnou distribuční funkci rozptylu (BSDF). Rád bych, zde uvedl příklad podoby souboru scény v Mitsubě a popis jednotlivých tagů.

V příloze C se nachází ukázka souboru s Mitsuba scénou (kód 14), kde lze nalézt tagy jako je transform, translate, rbg, atd. Tyto tagy jsou vnořené u tagů pro objekty (bsdf, emitter, film, integrator, rfilter, sampler, sensor, shape a texture), které potřebují bližší vysvětlení k pochopení jejich funkcionality. Co jednotlivé objekty představují a jakých mohou nabývat hodnot popisuje tabulka 6.1. Pluginy většinou rozšiřují možné hodnoty těchto objektů a implementují si vlastní typ některého z těchto objektů. Například v základu neumí Mitsuba renderovat obě dvě strany plochy, ale pomocí BSDF pluginu s názvem „two sided“ lze tento problém vyřešit.

Ve scéně lze vytvářet reference na jednotlivé části. Toho lze využít pokud máme jeden materiál pro více objektů, protože když využijeme referenci bude se muset tento materiál vytvořit jen jednou. Což nám u BTF textur může ušetřit mnoho paměti, tedy je potřeba vytvářet scény s ohledem na toto chování. Bohužel pokud budeme chtít změnit třeba velikost textury, musíme vytvořit dva materiály, které zaberou dvakrát více paměti.

Další výhodou XML formátu je, že můžeme velmi lehce změnit nastavení renderování. Například typ techniky sledování paprsků nebo hloubku rekurze při použití pathtracingu, která má velký vliv na dobu renderování i na jeho kvalitu. Podobný vliv má také počet vzorků na pixel. Ve scéně je možné

XML tag	Popis	Možné volby (*type*)
bsdf	BSDF popisuje způsob jakým světlo interaguje s povrchem ve scéně (tj. materiál).	diffuse, conductor, (bigbtf)
emitter	Emitter specifikuje světelné zdroje a jejich vlastnosti.	constant, envmap, point
film	Slouží k převedení výstupu do konečného souboru na disku.	hdrfilm
integrator	Integratory implementují renderovací techniky pro zobrazovací rovnice (*light transport equation*).	path, direct, depth
rfilter	Rekonstrukční filter určuje jak film převede množinu vzorků to výstupního souboru.	box, gaussian
sampler	Nastavuje způsob jaké vzorky bude používat integrátor.	independent
sensor	Rozšíření pro senzory jako jsou kamery, které zachycují světelné záření.	perspective, orthogonal
shape	Shape tag slouží pro vložení modelů v různých formátech.	obj, ply, serialized
texture	Texture tag reprezentuje prostorově se měnící signály na povrchu.	bitmap

Tabulka 6.1: Tabulka obsahuje informace o XML tagách, které se používají v Mitsuba scéně [43].

vytvořit proměnné, které se pak dají nastavit v příkazovém řádku, konkrétně pomocí přepínače *-D názevproměnné a $hodnota_proměnné*. Tyto proměnné se v XML definují pomocí tagu default: `<default name="spp" value="8"/>`. Ve scéně se na tuto proměnnou odkazuje pomocí "$", tedy v našem případě *$spp*. Pro změnění hodnoty spp tedy stačí zadat tento příkaz: *mitsuba.exe scene.xml -D spp=8*.

Nevýhodou Mitsuby a XML formátu scény je, že vytvoření scény a přidání objektů je velmi náročné bez grafického rozhraní. Naštěstí tvůrci Mitsuby jsou si tohoto problému vědomi a vytvořili převodník pro scény v Blenderu do Mitsuby s názvem Bleder 2 Mitsuba [44], jehož použití popíši v následující kapitole. Samozřejmě po konverzi jsou potřeba nějaké úpravy, například v XML je jednodušší nastavení enviromentálního osvětlení než v Blenderu a také nám Blender neumožní přidat materiál s BTF texturou, ten se musí po konverzi přiřadit k objektu ručně.

6.3.4 Převod scény z Blenderu do Mitsuby

Pro převod souboru scény z Blenderu (.blend) do XML formátu pro Mitsubu existuje rozšíření pro Blender, které je vyvíjeno samotnými tvůrci Mitsuba rendereru. Pro starší verzi Mitsuby 0.5 existovala i integrace pro Mitsuba renderer přímo jako rozšíření do Blenderu. Pro novou verzi Mitsuby 2.0 existuje převodník mezi scénami (.blend to .xml) s názvem Blender 2 Mitsuba [44]. V současné době je vyvíjen plugin pro integraci nové verze Mitsuby do Blenderu, ale ještě není dokončený. Díky těmto rozšířením bude mít bigbtf plugin větší využití. Hlavní výhodou Blenderu oproti Mitsubě je, že má uživatelské prostředí, ve kterém lze jednoduše vytvořit 3D scénu. Vytvořenou scénu i se základními materiály/texturami a světly můžeme převést do formátu scény pro Mitsubu (rozšíření vyexportuje i jednotlivé modely) a poté vyrendrovat. Existují zde nějaká omezení, protože Mitsuba nepodporuje vše co Blender, tedy objekt nemůže mít dva BSDF shadery a ve scéně nemůže být světelná plocha (*area light*).

Převodník obsahuje podporu pouze pro základní pluginy, které jsou součástí Mitsuby. Proto není možné definovat bigbtf materiál v Blenderu i s cestami k souborům s texturami a poté jen vygenerovat scénu. Po převodu do Mitsuba scény je potřeba změnit bsdf typ a nastavit cesty k MIF souborům pro objekty, u kterých chcete použít BTF texturu. Příklad takového objektu můžete vidět v ukázce kódu 3.

```
<shape type="ply">
    <string name="filename" value="meshes\Sphere.ply"/>
    <bsdf type="bigbtf">
      <string name="filepath" value="bigFiles/fabric106_uniform.mif"/>
    </bsdf>
</shape>
```

Kód 3: Ukázka přidání objektu koule s BTF texturou v Mitsuba XML scéně

Pro převod scény pomocí programu Blender 2 Mitsuba je potřeba mít zkompilovanou Mitsubu ve stejné verzi pythonu, jakou obsahuje Blender. Jinak rozšíření pro Blender nebude fungovat. Osobně používám konvertor s Blendrem verze 2.83, který má verzi pythonu 3.7.4. Tedy mám i Mitsubu zkompilovanou s verzí pythonu 3.7.4, kterou jsem si musel nastavit jako výchozí verzi systému. Verze pythonu jde nastavit i v CMake při sestavování Mitsuby. V rozšíření je potřeba zadat cestu ke složce kde je sestavená Mitsuba. Narazil jsem zde na problém, protože jsem nepoužil doporučenou složku k sestavení Mitsuby, kdy se má sestavit do složky build umístěné ve složce se zdrojovým kódem Mitsuby, ale já měl toto sestavení umístěné mimo tuto složku. Kvůli tomuto umístění mi plugin nefungoval, poté co jsem

Mitsubu sestavil přímo do složky build a zadal cestu k této složce, už rozšíření pro převod fungovalo správně.

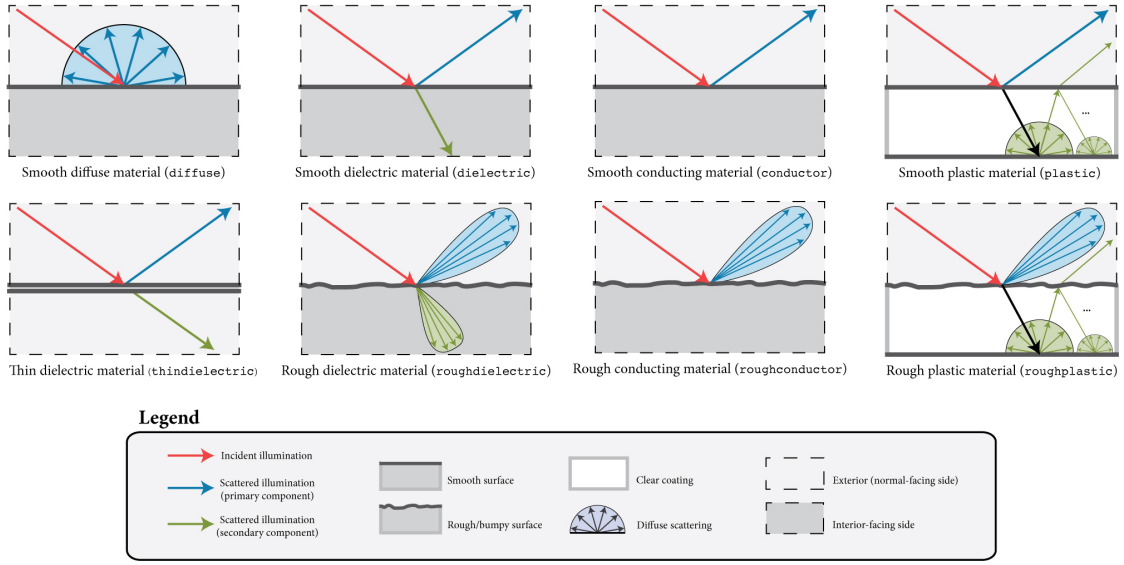
Na další problémy jsem narazil, když jsem zkoušel převést starší scény, které byly uloženy ve starší verzi Blenderu, u nichž se převod nezdařil. Musel jsem objekty z této scény přidat do mnou vytvořené nové scény a poté vše již fungovalo. Je možné, že některé ze zmíněných problémů budou časem vyřešeny, protože toto rozšíření je stále ve vývoji.

6.3.5 Návrh podpory nového formátu

Podpora nového formátu pro uložení BTF textur v Mitsubě by měla být pomocí pluginu. Mitsuba umožňuje vytvořit pluginy, které rozšiřují, případně nově implementují výše zmíněné uzly ve scéně. Proto bude potřeba vytvořit vlastní variantu uzlu BSDF, který určuje způsob interakce světla s povrchem.

Hlavní funkce tohoto pluginu by obsahovala souřadnice bodu x, y, úhly kamery θ_i, φ_i a úhly odrazu θ_v, φ_v. Tyto informace bude potřeba získat, případně vypočítat na základě informací z Mitsuby. Funkce bude dále potřebovat informace z nového formátu, o tom jaká data jsou v něm uložena, jakou mají reprezentaci a jaké filtrování se bude používat. Bez těchto informací nelze správně zvolit funkci pro získáni správného pixelu. Renderer potřebuje vědět pomocí jaké reprezentace jsou data uložena, aby se mohla provést správná interpolaci mezi pixely z různých obrázků BTF textury.

Jelikož data v novém formátu (BIG/MIF) mohou mít různou reprezentaci (UBO81x81, Uniform, CoatingRegular a CoatingSpecial), bude potřeba mít různé varianty funkce pro získání pixelu. Podpora rozšíření jako je mipmapping a anizotropní filtrování by měla být implementována v knihovně datového formátu MIF s tím, že bych funkci pro získání pixelu v Mitsuba renderu rozšířil o úroveň, která se má v mipmapě použít. Samozřejmě pro tyto rozšíření bude potřeba upravit třídu BigRender, tak aby dokázala vybírat správná data na základě zaslaných úrovní. V bigbtf pluginu pro Mitsubu bude potřeba ověřit, že BIG soubor obsahuje data pro mipmapping a přidat možnost pro výběr filtrování.



Obrázek 6.1: Přehled nejdůležitějších modelů rozptylu paprsku v Mitsubě [43].

6.3.6 Struktura pluginu v Mitsubě

Jak jsem již několikrát zmínil Mitsuba renderer má podporu pluginů. Mé rozšíření se týká konkrétně RenderAPI a rodiny rozšíření BSDF, která slouží pro rozšíření vlastností renderovaných materiálů a jejich celkového zobrazování. Vestavěné BSDF pluginy v Mitsubě umožňují renderování plastových a skleněných objektů nebo renderování obou stěn plochy. Mitsuba renderer umožňuje různé způsoby průchodu paprsků materiály. Umí simulovat nerovnou plochu materiálů, podpovrchové odrazy, atd. Ukázku některých průchodů paprsku materiálem můžete vidět na obrázku 6.1[5].

BSDF pluginy mají dané pevné rozhraní viz algoritmus 4. Základem BSDF rozšíření jsou tři funkce. První z nich je funkce `sample`, která provádí samotné vzorkování a společně s ní funkce `eval`, která vyhodnocuje příspěvky světla od světelných zdrojů. Funkce `eval` obsahuje informace o vektoru ke kameře a vektoru odrazu a na základě těchto informací vrací barevné spektrum. Poslední důležitou funkcí je `pdf`, která určuje pravděpodobnost vzorkování v daném směru na jednotku prostorového úhlu. Samozřejmě ještě každý plugin má konstruktor, v němž mu jsou jako parametr předány vlastnosti pluginu (*properties*), které jsou zde zpracovávány. Pokud povinné parametry nejsou uvedeny v XML souboru, tak se nám je v konstruktoru nepodaří načíst a Mitsuba nahlásí chybu. Navíc přesně vypíše jaký parametr chybí nebo byl zadán špatně.

[5]Pro BTF texturu nám stačí difúzní odraz do všech směrů, protože jevy jako samozastínění struktury a přenos světla v materiálu jsou již zachyceny v BTF datech.

```
template <typename Float, typename Spectrum>
class BigBTF final : public BSDF<Float, Spectrum> {
public:
    MTS_IMPORT_BASE(BSDF, m_flags, m_components)
    BigBTF(const Properties &props) : Base(props) {
       /* načtení parametru vašeho pluginu, inicializace struktur */
       //v tomto případě(big_render, m_nested_bsdf[2])
    }
    std::pair<BSDFSample3f, Spectrum>
    sample(const BSDFContext &ctx, const SurfaceInteraction3f &si,
           Float sample1, const Point2f &sample2, Mask active) const override {
       /*zpracování vzorku, zde se volá funkce get pixel,
       získává se zde úhel ke kameře a od kamery */
        return { sample, specttrum };
    }
    Spectrum eval(const BSDFContext &ctx, const SurfaceInteraction3f &si,
                  const Vector3f &wo, Mask active) const override {
        /* evaulace renderovacího procesu f(wi, wo)*/
        return spectrum;
    }
    Float pdf(const BSDFContext &ctx, const SurfaceInteraction3f &si,
              const Vector3f &wo, Mask active) const override {
       /*  výpočet pravděpodobnosti vzorkování
       v daném směru na jednotku prostorového úhlu */
        return pravdepodobnost;
    }
    MTS_DECLARE_CLASS()
protected:
    //ukázka třídních proměnných, které jsou inicializovány v konstruktoru
    ref<Base> m_nested_bsdf[2];
    BigRender *big_render;
};
```

Kód 4: Ukázka struktury pluginu v Mitsubě

Cílem mé práce je implementovat kód pro tyto funkce a získat v nich informace pro mou univerzální třídu, kterou jsem pojmenoval `BigRender`[6]. V konstruktoru se tato třída inicializuje pomocí názvu souboru a nastaví se jí parametry pro používání cache paměti, které budou součástí XML definice BSDF pluginu. Tato třída pro dané souřadnice textury a úhlů vrátí barevné spektrum, konkrétně RGB hodnoty.

[6]Původně třída pracoval s BIG knihovnou, přechod na MIF knihovnu proběhl až v průběhu vývoje, ale rozhodl jsem se názvy pluginu ani třídy neměnit, protože MIF knihovna je vlastně pokračovatelem BIG knihovny.

6.3.7 Instalace bigbtf pluginu do Mitsuby

Návod zde slouží pro instalaci pluginu na Windows. Pro instalaci bigbtf pluginu musíme vložit zdrojové soubory do Mitsuby a v projektu Visual Studia nalinkovat statické knihovny, s kterými plugin a hlavně třída `BigRender` pracuje. Samotná složka se zdrojovými soubory pluginu s názvem bigplugin, se vkládá do složky mitsuba/src/bsdfs. V této složce je potřeba změnit CMakeLists.txt, tak aby Mitsuba věděla, že jsme přidali nový plugin. Do tohoto souboru je nutné přidat řádek *`add_plugin(bigbtf bigplugin/bigbtf.cpp)`*. Nakonec je potřeba znovu použít Cmake a vygenerovat nové řešení (*solution*) pro Visual Studio. Poté po otevření souboru mitsuba.sln budou ve složce plugins/bsdf/bifbtf vidět projekty s názvy bigbtf a bigbtf-obj.

Snažil jsem se upravit soubory CMakeLists.txt v Mitsubě tak, aby se všechny potřebné cesty nastavily při generování projektu pro Visual Studio, ale bohužel jejich struktura je velmi složitá a nepodařilo se mi správně nastavit všechny závislosti. Pro samotné zprovoznění pluginu mi v rámci platformy Windows a Visual Studia přijde jednodušší nastavit cesty k potřebným knihovnám přímo pro projekt ve Visual Studiu. Proto v následujících odstavcích popíši jak nastavit cesty ke statickým knihovnám, které bigbtf plugin používá ve Visual Studiu.

Nejdříve je potřeba v průzkumníku řešení (*solution explorer*) najít soubor s projektem pluginu, který se nachází v plugins/bsdf/bigbtf. V této složce jsou dva projekty jeden s názvem bigbtf a druhý s názvem bigbtf-obj (bigbtf-obj je projekt pro kompilaci zdrojových souborů big pluginu a bigbtf pracuje s jeho výstupem a přidává další závislé třídy pro BSDF plugin). Upravit se musí vlastnosti obou[7]. Nejdříve se na bigbtf klikne pravým tlačítkem a vybere se menu vlastnosti (*properties*) a zvolíme podmenu Linker–General, zde je potřeba do položky *Additional Library Directories* přidat cestu ke složce se statickou knihovnou MIFlib.lib a pugixml.lib, které se nacházejí ve stejné složce. Poté cestu ke složce s PNGlib.lib (tato statická knihovna je ve složce ext/lib u bigbtf pluginu). Dále přejděte do Linker–Input a do položky Additional Dependencies přidejte položky *MIFlib.lib*, *pugixml.lib* a *PNGlib.lib*[8].

Poté můžete okno zavřít a otevřít okno s vlastnostmi bigbtf-obj projektu. Zde jděte do menu C/C++–General a do položky Additional Include Directories přidejte cestu ke složce include v MIF knihovně. A také ke složce ext/include v umístění zdrojového kódu bigbtf pluginu, což by mělo být `cesta_k_mitsubě/mitsuba/src/bsdfs/bigplugin/ext/include`.

[7] Budu zde pro přehlednost používat anglické názvy menu a podmenu, protože některé české překlady mohou být dost nejasné.

[8] Od verze MIFlib 0.4 by přidání pugixml.lib nemělo být nutné.

Jméno	Typ	Výchozí hodnota	popis
big_filepath	string	povinný parametr	Cesta k MIF souboru, může být relativní nebo absolutní.
memory	boolean	false	Pokud je hodnota true, tak se textura načte celá do paměti, v opačném případě se čtou data z disku.
scale	float	7.0	Slouží pro nastavení velikosti textury.
filtering	string	none	Název filtrování, které chceme, aby renderer používal.
level	integer	0	Posunutí úrovně používané u filtrování o danou hodnotu.
cubemap_path	string	povinný parametr pro UBO	Cesta ke složce s cubemapy, povinné pro soubory s UBO interpolací.

Tabulka 6.2: Parametry bigbtf pluginu.

Nyní stačí spustit projekt „INSTALL“ a Mitsuba se nainstaluje do složky *mitsuba/build/dist* i se všemi pluginy. Pokud již máte Mitsubu nainstalovanou a chcete jen přidat bigbtf plugin. Zkompilujte bigbtf projekt a přejděte do složky *mitsuba/build/src/bsdfs/{Release, Debug}* zde najděte soubor bigbtf.dll a zkopírujte ho do složky *mitsuba/build/dist/plugins* a plugin bude úspěšně přidaný do Mitsuby.

6.3.8 Implememtace bigbtf pluginu pro Mitsubu

```
<bsdf type="bigbtf" id="ace">
    <string name="big_filepath" value="F:\mifFiles\ace001.mif"/>
    <boolean name="memory" value="true"/>
    <string name="cubemap_path" value="cubemaps/081/0256"/>
    <float name="scale" value ="400.0"/>
     <integer name="level" value="2"/>
    <string name="filtering" value="none"/>
 </bsdf>
```

Kód 5: BSDF tag pro bigbtf plugin se všemi parametry, které může nabývat.

Bigbtf plugin je napsán v C++ a používá standartní rozhraní pro psaní pluginů v Mitsubě. Jeho funkčnost byla testována ve Visual Studiu verze 16.9.1 na Windows 10 a verzi Mitsuby 2.2.1. Plugin pracuje se třídou `BigRender`, která je součástí této práce a implementuje rozhraní pro komunikaci mezi rendererem a MIF knihovnou (původně BIG knihovnou, proto název `BigRender`

a bigbtf). Zdrojový kód pluginu se všemi knihovnami je přiložen na CD a také je dostupný na githubu (`https://github.com/zadinvit/BIGpluginMitsuba`).

Nejdříve bych rád popsal parametry pluginu a hodnoty, kterých můžou nabývat. Parametry pluginu jsou popsané v tabulce 6.2 s tím, že povinný je parametr cesty k souboru a v případě UBO81x81 reprezentace i cesta k umístění jednotlivých obrázků nutných k indexaci (cubemaps). Zbylé parametry jsou dobrovolné a v tabulce budou uvedeny jejich výchozí hodnoty. Podobu BSDF tagu se všemi parametry můžeme vidět v ukázce kódu 5.

Parametr filtering se v pluginu a poté ve třídě `BigRender` mapuje na enum třídu `Filtering`, která může nabývat pěti hodnot. Pokud máme data uložená s anizotropní mipmapou, lze využít všechny možnosti filtrování, v opačném případě nás renderer upozorní na špatný formát. Možné hodnoty třídy Filtering a jejich význam je popsán v následujícím seznamu:

- **MIPMAP_LINEAR**: Použití mipmappingu, bez interpolace mezi úrovněmi.
- **MIPMAP_WEIGHTED**: Použití mipmappingu s lineární interpolací mezi dvěma úrovněmi. Musíme dvakrát žádat o pixel.
- **ANIZO_1x**: Použití anizotropního filtrování, kdy se pouze zaokrouhlí úrovně v ose x a y a přečte se hodnota pro tyto zaokrouhlené hodnoty.
- **ANIZO_4x**: Použití anizotropního filtrování, kdy se interpoluje hodnota pixelu úrovně v ose x i ose y na základě euklidovské vzdálenosti se spočítají váhy, které se poté znormalizují na jedničku.
- **none**: Filtrování se nepoužívá.

Mitsuba obsahuje dokumentaci Render API[9], ale u jednotlivých proměnných ve třídě není napsáno jaký datový typ používají. Proto začátky tvorby pluginu v Mitsubě byly velmi náročné. Postupem pro zjištění vstupních a výstupních úhlů, tedy v kódu proměnné s názvy `theta_i, phi_i, theta_o, phi_o`, jsem se inspiroval v BRDF pluginu pro starší verzi Mitsuby [45]. Výpočet úhlů se v obou verzích prováděl stejně, jediné co se změnilo bylo kontrolování podmínek, aby se kód volal jen pro kladný cosinus úhlu `theta_i` a u funkce `eval` i `theta_o`. Tuto kontrolu jsem vypozoroval již v implementovaných pluginech Mitsuby, protože dokumentace nebo jiný zdroj neexistovaly. Pokud jsou cosiny těchto úhlů záporné vrací plugin 0.f, případně spektrum

[9] Dokumentace je dostupná na `https://mitsuba2.readthedocs.io/en/latest/generated/render_api.html`

plné nul, které si Mitsuba interpretuje jako špatný bod a nedochází tak ke zpomalení rendrovacího času ani ke zhoršení výsledků.

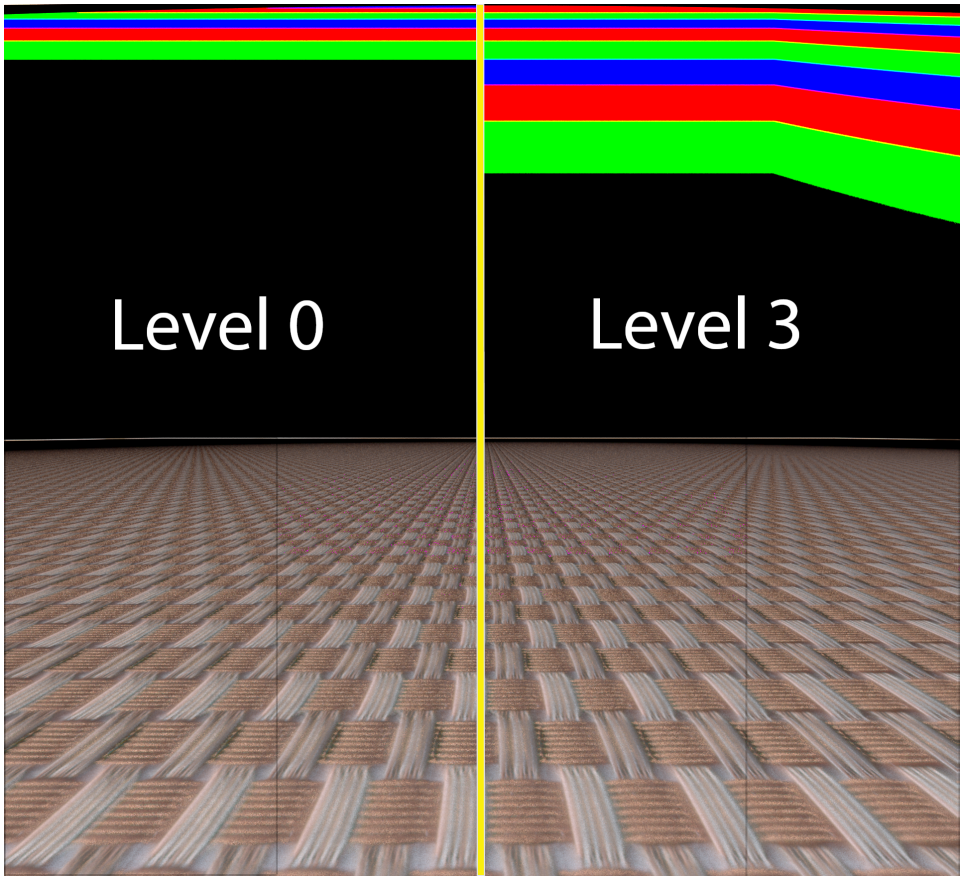
Dále Mitsuba potřebuje již v konstruktoru pluginu přiřadit vlajky (*flags*), které určují jaké pomocné hodnoty se budou při renderování počítat a jaké funkce bude plugin používat. Konkrétně se tato třída jmenuje `BSDFFlags`. U tohoto jsem narazil na problém, kdy se mi textura vykreslovala jen ve stínech a osvětlená část byla stále bez textury, tento problém popisuji v kapitole 6.9.1 „Nevykreslování osvětlených částí modelu“.

Při implementaci mipmappingu a anizotropního filtrování, je potřeba určit úroveň (*level*) textury, z kterého se bude renderovat. K tomuto se používají prostorové diferenciály, pomocí kterých se zjednodušeně řečeno zjišťuje jak velký bude daný pixel na displeji a v případě anizotropního filtrování jak je deformovaný, jestli je roztažený do výšky/šířky. Samotný výpočet těchto diferenciálů je u metod sledujících paprsek velmi složitý, proto jsem byl velmi rád, když jsem tyto parametry objevil v dokumentaci Mitsuby. Algoritmus pro výpočet těchto parametrů i pro implementaci výběru úrovně pro mipmapping je velmi dobře popsán na webové stránce v článku Mipmapping with Bidirectional Techniques od Yining Karl Li [37], který mi velmi pomohl k pochopení dané problematiky. Proměnné s diferenciály jsou v Mitsubě označeny `duv_x` a `duv_y`, ale po mém vyzkoušení byly prázdné. Problém byl v chybějící BSDF vlajce, podrobněji je jeho řešení popsáno v kapitole 6.9.3 „Problém se získáním prostorových diferenciálů“.

```
float width = max(max(si.duv_dx[0] * cols, si.duv_dx[1] * cols),
 ↪  max(si.duv_dy[0] * rows, si.duv_dy[1] * rows));
if (width == 0.0f) //log 0 není definovaný, proto musíme použít tuto
 ↪  podmínku
  level.levelx = 0.0f;
else
  level.levelx = min(float(big_render->maxMipLevel),
   ↪  float(big_render->maxMipLevel) + log2(width) + filter_level);
```

Kód 6: Výběr úrovně pro mippmaping v bigbtf.cpp, maxMipLevel je největší index v poli s pyramidou obrázků, filter_level je parametr pluginu, který může přispět k posunu filtrování.

Napsání algoritmu pro výpočet úrovně (*level*) mi zabralo mnoho času. V článku [37] je sice uveden algoritmus, který vybírá maximum ze čtyř hodnot uložených v `duv_x` a `duv_y` a poté od největší úrovně odečítá dvojkový logaritmus tohoto maxima, ale bohužel v mém případě výsledek vycházel velmi záporný, takže jsem vždy používal původní obrázek. Snažil jsem se hledat řešení v knihách (Real-time rendering) a v dokumentacích a kódech různých rendererů (PBRT, OpenGL, Mitsuba), které používaly velmi podobný po-



Obrázek 6.2: Vlevo se nachází scéna s mipmappingem na úrovni 0 a vpravo na úrovni 3. V dolní části uprostřed je vložen spojený obrázek, obou výsledků, kde růžové tečky znamenají odlišnosti. Na okrajích jsou již samotné obrázky bez spojení. Při použití úrovně 3 můžeme vidět jemnější přechod v textuře ve směru od kamery.

stup. Použití dvojkového logaritmu mi dávalo smysl, protože velikost jednotlivých úrovní v mipmapě se vždy zmenšovala o polovinu. Byl jsem si jistý, že mi chybí už jen malý krok, aby vše fungovalo a já získával správné výsledky.

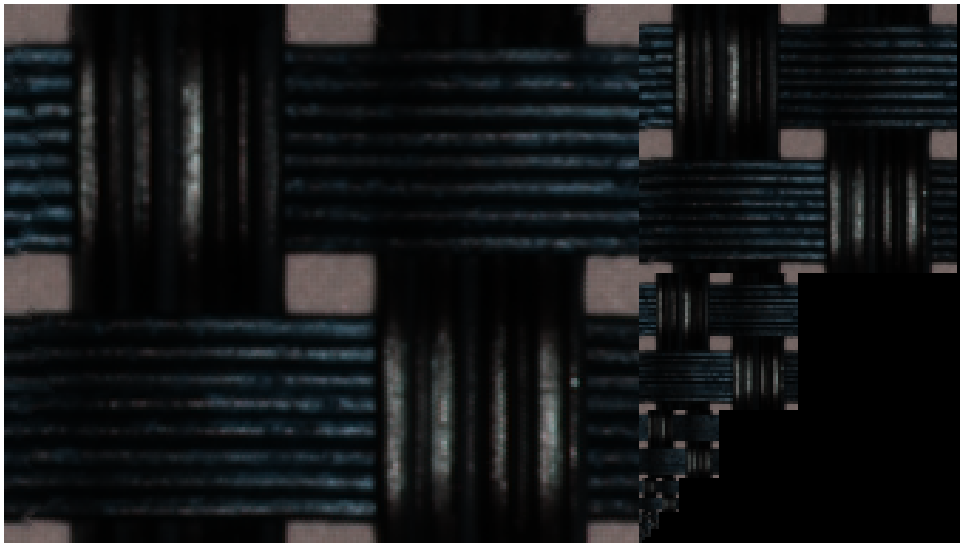
Nakonec mě na správnou cestu přivedl vedoucí práce emailem, kde mi poslal odkaz na implementaci mipmapy ve staré verzi Mitsuby (nová verze mipmapu už nepoužívá), kde jsem si všiml, že se hodnoty `duv_x` a `duv_y` při výpočtu maxima, vždy vynásobí velikostí největšího obrázku v hierarchii, tedy původního obrázku. Po této úpravě už výpočet funguje korektně, ale i tak jsem se rozhodl do pluginu přidat parametr `level`, který dokáže posunout úroveň použití filtrování, protože někdy může uživatel chtít použít toto filtrování od větší dálky, což mu tento parametr umožní. Úrovně filtrování můžete vidět na obrázku 6.2. Změna úrovně (barvy) znamená použití obrázku v o polovinu menším rozlišení, který už mám před vytvořený v MIF souboru.

```
std::pair<BSDFSample3f, Spectrum>
sample(const BSDFContext& ctx, const SurfaceInteraction3f& si,
    Float sample1, const Point2f& sample2, Mask active) const
     ↪  override {
    float cos_theta_i = Frame3f::cos_theta(si.wi);
    Spectrum spect;
    BSDFSample3f bs = zero<BSDFSample3f>();
    active &= cos_theta_i > 0.f;
    if (unlikely(none_or<false>(active) ||
     ↪  !ctx.is_enabled(BSDFFlags::DiffuseReflection)) ||
     ↪  cos_theta_i <= 0)
        return { bs, 0.f };
    bs.wo = warp::square_to_cosine_hemisphere(sample2);
    bs.pdf = warp::square_to_cosine_hemisphere_pdf(bs.wo);
    bs.eta = 1.0f;
    bs.sampled_type = +BSDFFlags::DiffuseReflection;
    bs.sampled_component = 0;
    float_t theta_i = acos(si.wi[2]);
    float_t theta_o = acos(bs.wo[2]);
    float_t phi_i = atan2(si.wi[1], si.wi[0]);
    float_t phi_o = atan2(bs.wo[1], bs.wo[0]);
    BigRender::MipLvl level;
    /* vyber urovne jsem v teto ukazce vynechal*/
    big_render->getPixel(si.uv[0], si.uv[1], theta_i, phi_i, theta_o,
     ↪  phi_o, level, RGB);
    spect = Color3f(RGB[0], RGB[1], RGB[2]);
    return { bs, select(active,spect,0.0f) };
}
```

Kód 7: Implementace funkce sample v bigbtf pluginu. Výběr úrovně je v ukázce vynechán.

Výběr úrovně pro anizotropní filtrování se liší jen v tom, že musíme vybírat různé úrovně pro osu x a y. Tedy se vypočítá maximum z hodnot prostorových diferenciálů v x a z něho se počítá úroveň pro x a maximum z hodnot v y a z něho se počítá úroveň pro y. V ukázce kódu 6 je uveden postup pro vypočítání úrovně pro mipmapping, ale tato ukázka by měla postačit k pochopení principu výpočtu i pro úroveň u anizotropního filtrování.

Bigbtf k získání pixelu používá třídu `BigRender`, které v konstruktoru předá jednotlivé parametry od uživatele a ověří jejich správnost. Poté používá funkci `BigRender::getPixel(const float &u, const float &v, float &theta_i, float &phi_i, float &theta_v, float& phi_v, MipLvl &level, float RGB[])`, pro získání pixelu. Celý kód pro funkci bigbtf sample, je vidět v ukázce kódu 7. Pro funkci eval je kód téměř identický, akorát se zde kontroluje hodnota cosinu `theta_o` a neobsahuje `BSDFsample` třídu, protože `eval` funkce má návratovou hodnotu pouze spektrum.



Obrázek 6.3: Ukázka generování obdélníkové mipmapy.

6.4 Implementace mipmappingu (izotropní filtrování)

Pro implementaci jsem se rozhodl použít offline verzi mipmappingu. Což znamená, že při použití renderingu mírně naroste rendrovací čas, protože se zvětší používaná textura a zároveň bude potřeba výpočet pro získání úrovně v mipmapě. Pro generování mipmap používám knihovnu OpenCV [46], která patří mezi nejlepší knihovny pro práci s obrazovými daty. Moje implementace zvětší původní soubor o 50 %, protože výsledný obrázek obsahuje prázdná černá místa, jak lze vidět na obrázku 6.3. Vím, že paměťově jde implementovat generování mipmap, tak aby zabraly jen 33 % paměti navíc a to pomocí ukládání jednotlivých barevných spekter zvlášť viz obrázek 5.6. To by s použitím v BIG formátu znamenalo značné problémy, protože tohle pořadí dat BIG ani MIF formát nepodporuje. Proto jsem se po konzultaci s vedoucím práce rozhodl pro variantu s prázdným místem, kde se indexuje vždy po pixelech a jednotlivé barevné kanály pixelu jsou uloženy za sebou viz obrázek 6.3.

Hlavní výhodou použití OpenCV je, že jsem nemusel implementovat algoritmus pro zmenšení obrázku. OpenCV má velmi sofistikovaný algoritmus pro zmenšování obrázku, který by dle dokumentace měl zaručit, že nevznikne moire efekt. Pro zmenšení jsem tedy využil funkci:

```
resize(img, img2, Size(), pow(0.5, level), pow(0.5, level),INTER_AREA).
```

Kde INTER_AREA je druh algoritmu, který se má použít pro změnu velikosti obrázku. Tento algoritmus je v dokumentaci doporučován pro zmenšování obrázků, používá různé transformace na základě požadované velikosti obrázku. Je tedy velmi univerzální a bude fungovat správně jak na čtvercových tak i na obdélníkových obrázcích. Prvním parametrem je původní obrázek a druhým je výstupní matice do které se uloží zmenšený obrázek. Původně jsem zmenšoval obrázky vždy o polovinu předešlého, ale docházelo zde k zaokrouhlovacím chybám a algoritmus nefungoval správně. Nyní všechny obrázky zmenšuji z původní velikosti a jen pomocí parametru level měním jejich zmenšení tj. $\frac{1}{2}^{level}$.

Funkce `resize` sice umožňuje nastavení velikosti výsledného obrázku přímo, ale po mém otestování a přečtení několika diskuzích na internetu jsem zjistil, že nastavování přesné velikosti může zhoršit kvalitu výsledných obrázků. Proto používám nastavování pomocí poměrů stran, které mi občas dá o jeden pixel rozdílný rozměr, ale obrázky vždy budou uloženy v nejlepší kvalitě.

Zbytek algoritmu je velmi jednoduchý díky využití OpenCV. Na začátku si vytvořím matici s mipmapou, která má o polovinu sloupců více než původní obrázek. Počet řádků má stejný jako originální obrázek. Poté potřebuji zjistit souřadnice horního levého bodu obrázku, do kterých se vloží o polovinu zmenšený obrázek. X souřadnice je rovna počtu sloupců původního obrázku a y souřadnici si průběžně vypočítávám. Poté jen zadám tyto souřadnice a velikost obrázku a nakopíruji zmenšený obrázek do obrázku s mipmapou. Pomocí této funkce:

```
img2.copyTo(mipmap(cv::Rect(img.cols, row, img2.cols, img2.rows))).
```

Jelikož obrázky mohou být obdélníkové musím v kódu ošetřit speciální případy, kdy se bude mipmapa zmenšovat o polovinu už jen ve sloupcích nebo řádcích. Například, když budou rozměry obrázku 3×1, tak ještě nemám nejmenší možný obrázek (1×1), ale již nemůžu obrázek zmenšovat o polovinu v obou rozměrech a musím pokračovat ve zmenšování jen v jednom. Celý kód můžete najít na přiloženém CD ve složce BigConvert[10], v souboru main.cpp a to konkrétně ve funkci `createMipMap`. Více o převodníku se můžete dočíst v kapitole 6.7.

Samozřejmě vygenerováním obrázku implementace mipmappingu nekončí, ve třídě `BigRender` bylo potřeba implementovat čtení z mipmap. Jelikož samotná BTF textura vždy interpoluje jeden pixel z několika obrázků, proto

[10]Případně je kód převodníku dostupný i na `https://github.com/zadinvit/bigConvert`.

```
std::vector<BigRender::Level> BigRender::getMipWeightLevels(const
↪  float& u, const float& v, float& level) {
    std::vector<Level> levels;
    float whole = floor(level);
    float decimal = level - whole;
    Level l1 = getCoordinatesMip(u, v, whole);
    l1.weight =  decimal;
    Level l2 = getCoordinatesMip(u, v, whole + 1);
    l2.weight = 1.0f - decimal;
    levels.push_back(l1);
    levels.push_back(l2);
    return levels;
}
```

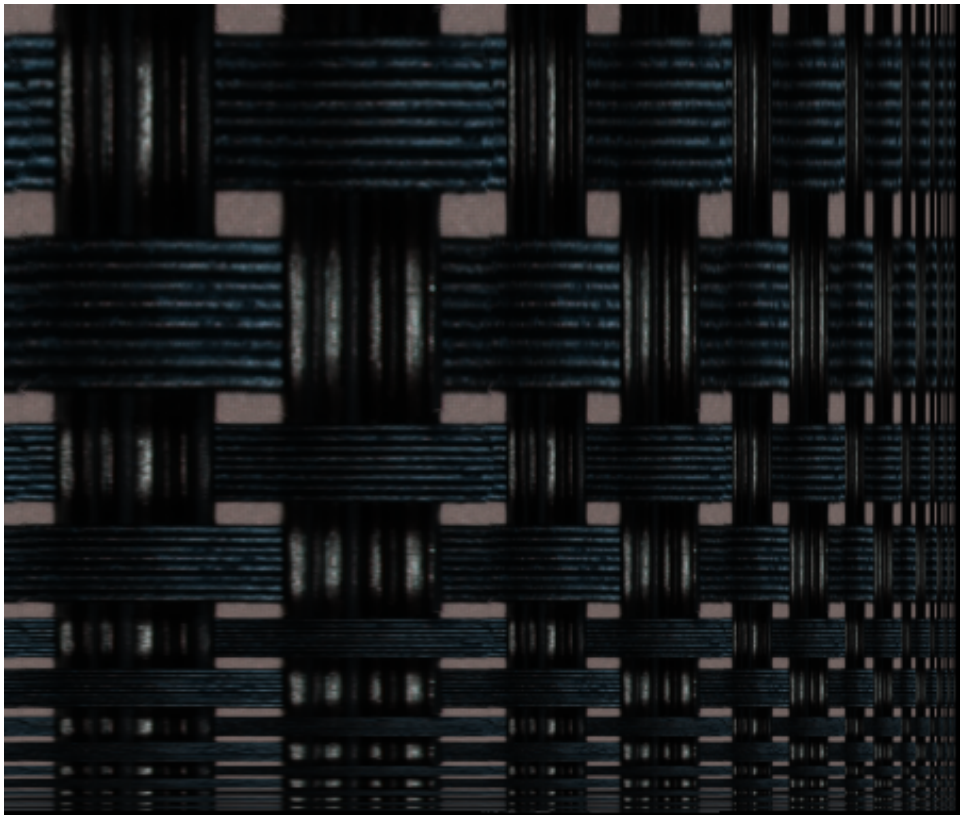
Kód 8: MIPMAP_WEIGHTED kód pro interpolaci. Funkce vrátí dvě úrovně s váhami, jejichž součet je jedna.

jsem se rozhodl ponechat verzi mipmappingu, která zaokrouhlí desetinou hodnotu úrovně (*level*) a použije tu. Tato varianta se nazývá **MIPMAP_LINEAR**.

Poté existuje varianta **MIPMAP_WEIGHTED**, která interpoluje mezi dvěma úrovněmi, což můžeme vidět v kódu 8. Tedy se musí číst pixel dvakrát, což v případě interpolace pixelu v BTF textuře může být mnoho přístupů do souboru s texturou, přesný počet záleží na druhu reprezentace dat (u uniformní to je šestnáct přístupů do souboru z texturou). Toto několika násobné volání jsem implementoval tak, aby se nemusely zbytečně počítat pomocné hodnoty pro interpolaci skrz BTF texturu. Tedy dvojnásobné volání získání pixelu neznamená zpomalení renderovacího procesu o dvojnásobek. Zpomalení je v řádu jednotek procent a blíže bude popsané v kapitole 7 „Testování a výsledky“.

6.5 Implementace anizotropního filtrování

Pro anizotropní filtrování jsem se rozhodl použít offline metodu, kdy si jednotlivé mipmapy s filtrováním vytvořím předem. Stejně jako u mipmappingu zde dojde ke zhoršení času rendrování, kvůli použití většího souboru z texturou, což zpomalí hlavně inicializaci renderovacího procesu. Především času inicializace, protože potřebujeme čtyřikrát větší texturu a samozřejmě taky o výpočet jednotlivých úrovní (`levelx` a `levely`). Podobu anizotropní mipmapy (ripmapy) můžete vidět na obrázku 6.4. Pro generování jsem opět použil knihovnu OpenCV. Celý kód pro generování funkce lze nalézt na CD ve složce BigConverter v souboru main.cpp ve funkci `createAnisoMap`.



Obrázek 6.4: Ukázka mipmapy pro anizotropní filtrování.

Je zde opět využita funkce resize z OpenCV knihovny. Rozdíl oproti klasické mipmapě je ve velikosti, kdy anizotropní mipmapa je čtyřikrát větší, protože musíme její šířku i výšku zvětšit dvakrát, takže paměťové nároky tohoto řešení jsou společně s použitím na BTF textur enormní. Co se týče algoritmu, tak cyklus nyní nejdříve zmenšuje výšku jejich obrázků a poté zmenšuje šířku. Výhodou této mipmapy (ripmapy) je, že se dá použít i na klasický mipmapping, protože na diagonále má uložené klasické mipmapy, které jsou zmenšené v obou rozměřech stejně. Jen je třeba tuto variantu správně implementovat v kódu.

U anizotropního filtrování byla těžší část implementace čtení mipmapových dat ve třídě `BigRender`. Kde existují podobně jako u mipmappingu dvě verze filtrování s názvy ANIZO_1x a ANIZO_4x, které určují různé druhy interpolace mezi jednotlivými hodnotami úrovní a jsou i rozdílně výpočetně náročné.

ANIZO_1x znamená, že se hodnota úrovně x i y standardně zaokrouhlí a vybere se textura na základě těchto hodnot. Takže u tohoto typu filtrování by při raytracingu nemělo docházet k významnému zpomalení. Nastane zde pokles rychlosti, protože zde musí probíhat výpočet jednotlivých úrovní

oproti renderovaní bez anizotropního filtrování. Také se zde musí vypočítávat prostorové diferenciály, jejichž výpočet je časově náročný.

```
std::vector<Level> levels;
float wholex = floor(level.levelx);
float decimalx = level.levelx - wholex;
float wholey = floor(level.levely);
float decimaly = level.levely - wholey;
float decimalxInv = 1 - decimalx;
float decimalyInv = 1 - decimaly;
Level l1 = getCoordinatesAnizo(u, v, wholex, wholey);
l1.weight = sqrt(decimalx * decimalx + decimaly * decimaly);
levels.push_back(l1);
l1 = getCoordinatesAnizo(u, v, wholex + 1, wholey);
l1.weight =sqrt(decimalxInv * decimalxInv + decimaly*decimaly);
levels.push_back(l1);
l1 = getCoordinatesAnizo(u, v, wholex + 1, wholey + 1);
l1.weight = sqrt(decimalxInv * decimalxInv +
↪  decimalyInv*decimalyInv);
levels.push_back(l1);
l1 = getCoordinatesAnizo(u, v, wholex, wholey + 1);
l1.weight = sqrt(decimalx * decimalx + decimalyInv *
↪  decimalyInv);
levels.push_back(l1);
normalizeLevelsWeight(levels);
```

Kód 9: ANIZO_4x kód pro interpolaci. Vytvoří se zde čtyři úrovně pomocí euklidovské vzdálenosti a jejich váhy se nakonec znormalizují, aby součet vah byl roven jedné.

ANIZO_4x je implementace anizotropního filtrování kde se interpoluje mezi úrovněmi jak v ose x, tak i v ose y. Takže nám vzniknou čtyři vzorky a tedy musíme získat čtyřikrát pixel, který poté podle vah jednotlivých vzorků spojíme do jednoho. K výpočtu vah se používá euklidovská vzdálenost, která se vždy musí vypočítat pro čtyři různé varianty. Výpočet vah můžeme vidět v ukázce kódu 9, kde struktura `Level` má v sobě uložené souřadnice mipmapy v x a y a váhu získaného pixelu.

Tedy by se dalo říct, že se musí čtyřikrát zavolat funkce `getPixel`, ale to není úplně pravda. U získávání pixelu v BTF textuře se musí interpolovat mezi jednotlivými směry a značný čas zabere výpočet pomocných hodnot pro interpolaci, ale ty v tomto případě můžeme počítat jen jednou a až interpolaci mezi hodnotami stačí provést čtyřikrát. Z toho vyplývá, že použití anizotropního filtrování ve verzi ANIZO_4x nám čtyřikrát nezpomalí čas renderování.

6.6 Třída BigRender

Třída `BigRender` slouží ke komunikaci mezi rendererem a knihovnou MIF. Je součástí implementace mého pluginu bigbtf, ale je možné ji využít i při implementaci v jiném rendereru, protože je plně nezávislá na Mitsubě. Skládá se z hlavičkového a zdrojového souboru s názvem big_render.cpp a big_render.hpp. Je napsána v C++ a pracuje s knihovnou MIFlib verze 0.4 a statickou knihovnou PNGlib[11], jejíž zdrojový kód je přiložen s kódem celého pluginu ve složce bigbtf.

Třída `BigRender` obsahuje dva konstruktory, oba vyžadují cestu k souboru a pravdivostní hodnotu pro použití cache. Liší se v tom, že druhý konstruktor slouží pro UBO81x81 reprezentaci, která potřebuje navíc cestu k obrázkům s předčítanými indexy. Pomocí konstruktoru se třemi parametry lze vytvořit jen soubory s UBO81x81 reprezentací, ostatní se musí vytvářet pomocí druhého konstruktoru. Společnou část konstruktorů tvoří funkce `init`, kde se otevírá MIF soubor a načítají základní informace o obrázcích a inicializuje se cache pro čtení z paměti, případně z disku. Mimo funkci `init` se nachází inicializace hodnot, které jsou pro každou reprezentaci BTF dat speciální. Tyto hodnoty jsou aktuálně zapsány v kódu, ale mohly by se načítat z MIF souboru. Tuto variantu jsem zvolil, protože pokud by se změnilo názvosloví pro tyto proměnné mohl by plugin přestat fungovat úplně, protože by neměl správná data pro interpolaci jednotlivých formátů. Tyto úhly jsou zároveň součástí specifikace těchto formátů, takže hodnoty by se neměly měnit.

Nejdůležitější funkcí této třídy je funkce `void getPixel(const float& u, const float& v, float& theta_i, float& phi_i, float& theta_v, float& phi_v, MipLvl& level, float RGB[])`. Jejíž parametry `u` a `v` jsou UV souřadnice v rozsahu 0–1, na základě kterých probíhá výběr bodu x a y v textuře, poté úhly ke světlu a ke kameře, u kterých se předpokládá, že jsou v radiánech. Následuje struktura `MipLvl`, která se používá pro filtrování, která obsahuje `level` (mipmapping) nebo `levelx` a `levely` (anizotropní filtrování) a pole RGB typu float, u kterého se předpokládá, že má velikost tři.

Každá reprezentace BTF dat používá trochu jinou interpolaci, proto samotná funkce `getPixel` na základě typu reprezentace dat vybere správnou funkci pro získání pixelu pro danou variantu (Uniform – `getPixelUniform`, UBO81x81 – `getPixelCubeMaps`, CoatingRegular – `getPixelBTFthph`, CoatingSpecial – `getPixelBTFthtd`). Tyto funkce mají stejné parametry jako

[11] Knihovnu PNGlib mi poskytl vedoucí práce společně s dalšími zdrojovými kódy potřebnými pro interpolaci UBO81x81 reprezentace.

funkce `getPixel`, liší se hlavně ve způsobu interpolace a získávání pixelu z texturních dat.

Pro rozlišení reprezentace dat a typu filtrování obsahuje třída `BigRender` dvě enum třídy, které řeší mapování názvů těchto tříd na enum. Každá z těchto tříd obsahuje unorder map, pomocí které se přiřazují jednotlivé stringy k enum vyjádření (konkrétně pomocí funkce `parse/parseFilter`). Tyto enumy jsem přidal do třídy hlavně proto, aby bylo jednodušší rozšířit třídu o nové druhy filtrování nebo reprezentace dat. Pokud by se změnilo pojmenovaní reprezentace BTF dat, stačí tento název změnit jen na jednom místě ve zdrojovém kódu (v enum třídě `Distribution`). Další výhodou použití enum třídy je lepší práce při volání správné funkce pro danou reprezentaci při použití switch konstrukce.

Dále třída obsahuje dvě struktury a to konkrétně `MipLvl`, který slouží pro reprezentaci úrovní předávaných rendererem pro mipmapping a anizotropní filtrování. Druhá struktura s názvem `Level` se používá pro generování jednotlivých vzorků pro filtrování, kdy obsahuje váhu vzorku a jeho souřadnice s názvem `row` a `col`. Obě tyto třídy jsou popsány v ukázce kódu 10.

```
public:
     //pro mipmapping se použije jen levelx
    struct MipLvl {
        float levelx = 0;
        float levely = 0;
    };
private:
    //Level struktura se používá při interpolaci pixelu
    struct Level {
        float weight;
        int row;
        int col;
    };
```

Kód 10: Struktury používané při implementaci filtrování pro předávání jednotlivých úrovní (MipLvl) a pro samotné filtrování (Level).

Všechny funkce pro získání pixelu obsahují část kde se inicializují struktury pro interpolaci, poté cyklus, kde se interpoluje přes jednotlivé vzorky pro filtrování a uvnitř tohoto cyklu probíhá samotná interpolace dat. Jednotlivé vzorky se postupně přidávají do výsledného pole s názvem RGB pomocí váhy vzorku. Při čtení dat ze souboru se vždy musí použít správná cache (čtení z disku/čtení z paměti) jejichž rychlosti i implementace se liší.

Pro čtení přímo z paměti se předává pouze const float ukazatel na začátek pole s hodnotami na rozdíl od čtení z disku, kdy se funkci musí předat float pole, do kterého se data překopírují, což je pomalejší. Jednotlivé interpolace pro různé reprezentace dat jsem dostal od vedoucího práce, který je měl už dříve naprogramované pro své použití při práci s BTF daty.

Funkce `getPixel` by se dala zjednodušeně popsat takto:

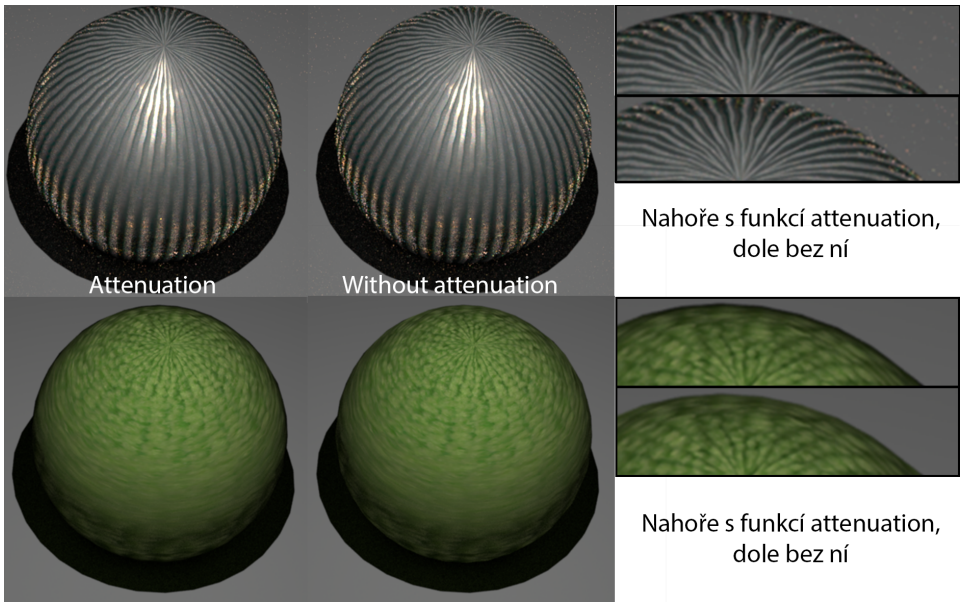
1. inicializace pomocných dat po interpolaci
2. získání vzorků pro filtrování – `getLevels`
   a. vytvoření určitého počtu vzorků (struktura `Level`)
   b. přiřazení souřadnic textury těmto vzorkům
   c. přiřazení váhy jednotlivým vzorkům
3. získání pixelu
   a. iterace přes vzorky
   b. interpolace a získání indexu textury a čtení dat ze souboru
   c. spojení jednotlivých vzorků do výsledného pixelu
4. postupný přechod stínu v textuře (u raytracingu se nepoužije) – `attenuateElevations`
5. převod z prostoru XYZ do prostoru RGB – `XYZtoRGB`
6. zaokrouhlení dat na nulu – `clampToZero`

Velmi důležitá funkce je funkce `getLevels`, která vytvoří pole struktur `Level` na základě toho, jaké filtrování se používá. Přes tyto filtrovací úrovně pak probíhá interpolování pixelu a jeho skládání. Vím, že přidání tohoto kroku mírně zpomalilo renderování bez filtrování, ale udržela se tak přehlednost třídy `BigRender`. Ve funkci `getLevels` se na základě hodnoty v proměnné `dist` (enum `Distribution`), rozhodne jaká funkce pro získání vzorku se použije. V samotných funkcích se vytvoří vzorky a spočítají se zde texturní souřadnice. Provede se tedy namapování UV souřadnice do souřadnic obrázku a jejich opakování při kterém se uplatní i nastavená velikost zvětšení textury, což lze vidět na ukázce kódu 11. Poté se pro jednotlivé vzorky spočítají jejich váhy. Tento výpočet jsem popsal výše v kapitolách o implementaci mipmappingu 6.4 a anizotropního filtrování 6.5. Pro základní rendering bez filtrování se spočítají jen souřadnice `row` a `col` v textuře a vzorku se přiřadí váha jedna.

Poté v případě klasického rasterizátoru následuje funkce `attenuateElevations`, která má zajistit volný přechod stínů textury, aby zde nebyl ostrý přechod

```
l.row = (int)(floor(abs(u) * (float)lvl.getRows() * uv_scale)) %
↪  lvl.getRows();
l.col = (int)(floor(abs(v) * (float)lvl.getCols() * uv_scale)) %
↪  lvl.getCols();
//u filtrovani se musí přičíst i počátek souřadnicového systému
l.row += lvl.getY();
l.col += lvl.getX();
```

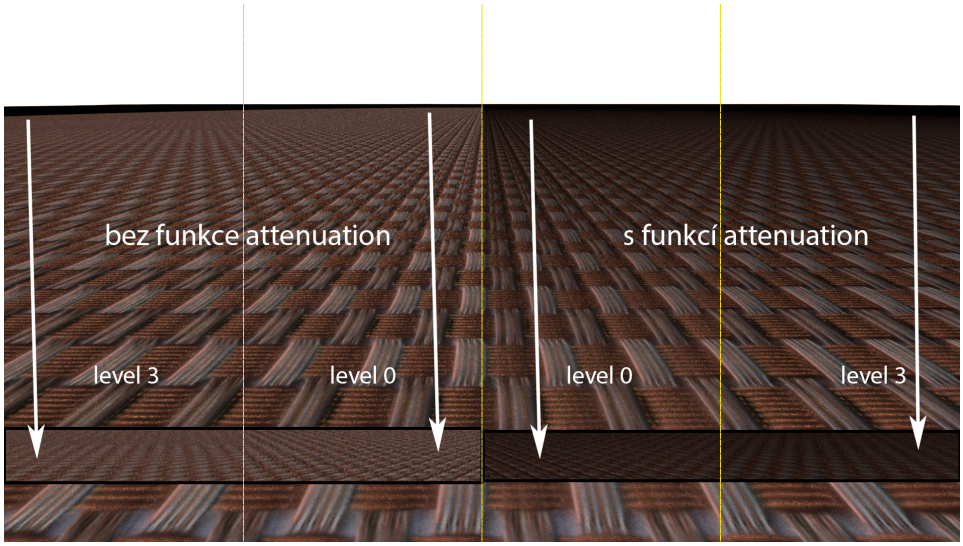
Kód 11: Převod UV souřadnic do souřadnic obrázku, zajištění opakování textury a jejího zvětšení.



Obrázek 6.5: Porovnání použití funkce attenuateElevations a nepoužití v Mitsuba rendereru. Při použití jsou některá místa tmavší. Vlevo jsou materiály vykresleny s použitím funkce a vpravo bez použití funkce.

mezi tmavým stínem a osvětlenou částí. V raytracingu jsem se po několika testovacích scénách a materiálech rozhodl tuto funkci nepoužívat. Jak je vidět na obrázku 6.5, při použití v Mitsubě je její vliv na texturu a její stínování vysoký a zhoršuje vnímání samotné textury a nasvícení scény. Při použití globálního světla lze vidět na obrázku 6.6 celkové ztmavení textury a zhoršení vlivu osvětlení na texturu.

Po těchto funkcích následuje funkce `XYZtoRGB`. Jelikož data změřená na ÚTIA AV ČR jsou uložena v XYZ barevném prostoru a je nutné provést převod do RGB prostoru, s kterým pracuje Mitsuba. S implementací této funkce jsem měl problémy, protože jsem nevěděl, jestli při převodu provádět gama korekci. Měl jsem tuto funkci implementovanou v několika podobách. Po několika pokusech a konzultacích s vedoucím práce sem zjistil, že gama



Obrázek 6.6: Scéna mipmap.xml a porovnání použití attenuation. Na obrázku lze vidět použití filterování MIPMAP_WEIGHTED a různých jeho posunutí (level=0 a level=3). Nastavení globálního osvětlení se neměnilo.

korekce, která slouží k nelineárnímu převodu do prostoru sRGB není potřeba, protože ji provádí Mitsuba při ukládání renderovaného obrázku. Takže se zde jen využívá standardní matice pro převod mezi barevnými prostory XYZ a RGB.

Poslední funkce, která byla potřeba implementovat je `clampToZero`. Jelikož během měření BTF textur dochází k nepřesnostem a po převodu do RGB dochází k tomu, že některé hodnoty mohou být záporné. Když dostávala Mitsuba tyto záporné hodnoty docházelo k výpisu do konzole, že byly vráceny nestandardní hodnoty a renderovací čas se velmi zpomalil. Tento problém řeší funkce `clampToZero`, která nezáporná data zaokrouhlí na nulu, aby vždy byly vráceny korektní RGB data.

Dále třída `BigRender` obsahuje pomocné funkce pro inicializaci dat pro interpolaci, hlavně pro interpolaci formátu UBO81x81, u kterého je potřeba načítat indexy z obrázků. Proto při použití třídy `BigRender` musíte k projektu přilinkovat knihovnu PNGlib, kterou mi poskytl vedoucí práce. Také jsou zde funkce pro nastavování některých hodnot, se kterými se během získávání pixelu pracuje a to konkrétně funkce `SetScale` a `SetFilter`, které slouží pro nastavení měřítka textury a pro nastavení typu filtrování.

6.7 Převodník do MIF souborů

ÚTIA AV ČR má v současné době uloženy textury ve staré verzi BIG formátu. V této formě jsou data dostupné i v databázi textur na webových stránkách ÚTIA AV ČR (`http://btf.utia.cas.cz/`). Bohužel v tomto formátu není uložen typ interpolace, ale v databázi MAM2014 jsou všechny data uložena v hemisferické uniformní reprezentaci (UBO81x81). Jelikož jsem se rozhodl jak mipmapping, tak anizotropní filtrování dělat pomocí offline metody, u které jsou vygenerované mipmapy uložené přímo v souboru s texturou. Potřeboval jsem vytvořit převodník z BIG formátu do nového MIF formátu, který by umožňoval uložení a vygenerování mipmap[12].

Zdrojový kód převodníku je přiložen na CD ve složce bigConvert a také zde můžete najít i zkompilovanou spustitelnou aplikaci (convertFiles.exe) ve složce PortableProgramy/BigConvert. Aplikace se ovládá pomocí konfiguračního souboru. Kde první řádek obsahuje globální definici reprezentace dat. Tedy možnosti jsou UBO81X81, uniform, CaotingRegular a CoatingSpecial. Druhý řádek definuje filtrování a možnosti jsou: none, mipmap, anisotropy. Další řádky vždy obsahují cestu k souboru, buďto relativní nebo absolutní a případně můžeme definovat typ reprezentace nebo se použije globální nastavení. Cesty k souborům nesmí obsahovat mezery. Ukázkový konfigurační soubor je vidět na ukázce kódu 12.

Pro kompilaci převodníku je potřeba nalinkovat MIFlib.lib a OpenCV knihovnu. Kód převodníku byl testován s OpenCV verze 4.5.1 a MIFlib verze 0.4. Kód převodníku je velmi jednoduchý v podstatě se na základě reprezentace dat vybere kód pro čtení a pak se provádí samotný převod. Kódy pro načítání obrázků ze starého BIG formátu pro jednotlivé interpolace mi poskytl vedoucí práce. Tyto kódy byly velmi neuspořádané a bylo vždy potřeba spouštět program ve Visual Studiu, kde se musely přepisovat cesty k souborům i druh reprezentace. Proto jsem vytvořil tento převodník s konfiguračním souborem, aby si kdokoliv, kdo bude chtít použít bigbtf plugin, mohl stáhnout soubory ve starém BIG formátu a spustitelnou Windows aplikaci (.exe) převodníku. Pomocí které si soubory lokálně převede do MIF formátu.

Jak jsem již zmínil převodník umí vytvářet anizotropní a klasické mipmapy. Konkrétně to jsou funkce `createMipMap` a `createAnisoMap` a jejich varianty s příponou info, které navíc zapisují do XML MIF souboru souřadnice mipmap. Výsledkem těchto funkcí jsou obrázky 6.3 a 6.4. Podrobně jsem tyto funkce popsal v kapitolách 6.4 a 6.5.

[12] Předtím jsem tento převodník používal pro převod ze staré verze BIG formátu do nové verze.

```
distribution: UBO81
filtering: mipmap
MAM2014_016 UBO81x81
fabric106_uniform Uniform
paints/schlenk075_BTFthtd CoatingRegular
paints/schlenk075_BTFthph CoatingSpecial
```

Kód 12: Ukázka konfiguračního souboru config.txt pro aplikaci BigConvert

6.8 Implementace v dalších rendererech

Vizualizaci podpory BTF dat uložených v MIF jsem se rozhodl implementovat pouze pro Mitsubu. Dal jsem přednost testování a doladění funkce `BigRender`, která by měla usnadnit implementaci do dalších rendererů jako je Blender nebo PBRT. Díky tomu bude potřeba implementovat jen získání informací o pozorovacích úhlech a případně spočítat úroveň pro filtrování. Bohužel seznámení se s architekturou rendereru a způsobem jak vypočítat informace potřebné pro funkci `getPixel` trvá velmi dlouhou dobu. U Mitsuby jsem jen se seznamováním se s architekturou kódu strávil desítky hodin[13].

Dalším faktorem proč jsem se rozhodl neimplementovat plugin pro Blender je to, že tvůrci Mitsuba rendereru vyvíjí plugin, který by umožnil použití Mitsuba rendereru přímo v Blenderu. Samozřejmě přidat podporu MIF přímo pro renderer Cycles by mohlo být užitečné, ale jak jsem psal již dříve byla by potřeba kompilace celého Cycles, což by nebylo uživatelsky moc přívětivé.

6.9 Problémy při implementaci a jejich řešení

V této kapitole popíši řešení několika problému, na které jsem narazil během vývoje. Některé z těchto problémů byli zmíněny v kapitolách výše, bez bližšího popisu, který zde naleznete. Jsou zde uvedeny problémy, jejichž řešení nebylo jednoduché a zabralo mi několik dnů. Jeden z problémů jsem musel řešit s komunitou na githubu Mitsuby a díky tomu jsem zjistil, že na vývoji Mitsuby se podílí mnoho lidí, kteří velmi rádi a rychle pomůžou s dotazy ohledně problémů s implementací.

[13] Do tohoto času počítám i samotnou instalaci a zprovoznění Mitsuby.

Problém s využitím částečné cache paměti měl velký vliv na směr vývoje této práce, protože jsem si uvědomil, že k datům se přistupuje vícevláknově a to způsobuje mnoho problémů a nestandardních situací, s kterými BIG knihovna nepočítala. Což vedlo k přechodu na MIF knihovnu, která v případě použití cache paměti nepotřebuje zadávat velikost cache paměti, ale umí si spočítat maximální velikost této paměti sama. Takže se změnil i parametr cache bigbtf pluginu z celého čísla, které určovalo velikost paměti na pravdivostní hodnotu.

Obrázek 6.7: Vlevo je scéna se základní texturou, slouží pro představu nasvícení, uprostřed se nachází špatně vykreslovaná BTF textura a vpravo je opravené vykreslování, tedy již se volá funkce eval.

6.9.1 Nevykreslování osvětlených částí modelu

Na tento problém jsem narazil při začátcích, kdy se mi už povedlo zprovoznit plugin a vydával nějaké smysluplné obrázky. Problém byl v tom, že místa objektu, která by měla být nejvíce osvětlená byla černá. Nasvícení scény a výsledky před opravou a po opravě můžete vidět na obrázku 6.7. Nejdříve jsem myslel, že je chyba v mém kódu. Kontroloval jsem jednotlivé úhly, jestli nejsou prohozené, indexaci, zkoušel jsem jiné mapování UV souřadnic, ale nic z tohoto nepomohlo. Další možností, která toto mohla způsobit, byli záporné RGB hodnoty, které jsem získával z BTF textury, ale tento problém jsem také vyloučil, když jsem si nastavil, aby renderer vracel vždy jednu barvu a místa na objektu byla stále černá.

Pomocí debugování jsem zjistil, že se vůbec nevolá funkce `eval` a tedy to je důvod proč se nevykreslují nejvíce nasvícená místa, ale nevěděl jsem proč. Základní struktura mého kódu a kódu ostatních BSDF pluginů byla stejná. Po dvou dnech nezdárných pokusů, jsem si konečně všiml, že ostatní rozšíření v konstruktoru nastavovala vlajky (*flags*), které určují způsob distribuce paprsků, jak jsem již zmiňoval výše na obrázku 6.1. Konkrétně bylo potřeba přidat do konstruktoru dva řádky viz kód 13.

```
m_flags    = BSDFFlags::DiffuseReflection | BSDFFlags::FrontSide;
m_components.push_back(m_flags);
```

Kód 13: Nastavení scattering modelu pro btfbif plugin, opravení chyby, kdy se osvětlená místa nevykreslovala.

6.9.2 Problém s cache pamětí v BIG knihovně

Při používání BIG knihovny jsem nejdříve četl data přímo z disku, poté jsem zkusil načíst celou texturu do paměti a vše probíhalo bezproblémově. Problémy začaly, když jsem použil velikost cache paměti menší než byla velikost souboru. Zde nejdříve docházelo k chybě programu při práci s listy. Konkrétně ve funkci `splice` (v některých případech kvůli neplatnému iterátoru, v ostatních se mi přesnou příčinu nepovedlo odhalit). Po vyřešení chyby s neplatnými iterátory program náhodně padal dále. Někdy při práci s listem jindy až po doběhnutí a čistění paměti v destruktoru listu. Nakonec jsem pro datový typ float v kterém jsou veškeré BIG soubory nyní uloženy zkusil upravit použití listu na frontu (*queue*), ale tím jsem odstranil logiku pro odstraňování vždy nejdéle nepoužívaného elementu.

Navzdory použití fronty jsem zjistil, že program nepadal, ale pamět postupně rostla nad velikost souboru s texturou. BIG knihovna pro ukládání obrazových dat používá chytré ukazovatele, konkrétně shared_ptr a na jejich mazání používal funkci `reset`, která ovšem nezaručuje smazání všech dat, pokud pointer má více referencí sníží se tím jen jeho reference. Však ani toto nebyl problém, po analýze jsem zjistil, že pointery sice ukazují, že jsou prázdné (*empty*), ale na haldě zůstávají uložená nesmazaná data a paměť stále narůstá i nad původní velikost obrázku. Toto chování nejspíše způsobuje vícevláknový přístup k datům, s kterým se při implementaci BIG knihovny nepočítalo.

Tento problém by vyžadoval kompletní předělání BIG knihovny. Po konzultaci tohoto problému s vedoucím práce mi bylo řečeno, že se vyvíjí nová verze formátu od úplného začátku (MIF) a že se na ní postupně přejde a nemám tento problém v BIG knihovně řešit. Takže bigbtf plugin bohužel v současné době neumí načíst jen část BTF textury do paměti. Velkým vylepšením MIF knihovny by byla implementace částečného ukládání dat do paměti, které by si poradila i s vícevláknovým přístupem a správně by mazala data.

6.9.3 Problém se získáním prostorových diferenciálů

Mitsuba sice obsahuje proměnné `duv_x` a `duv_y`, které by měly prostorové diferenciály obsahovat, ale tyto proměnné byly prázdné. Po delším zkoumání dokumentace a vyzkoušení různých variant `BSDFFlags`, které by mi mohly umožnit výpočet těchto diferenciálů, jsem se pokusil zeptat na githubu Mitsuby. Kde mi bylo velmi rychle odpovězeno, že výpočet těchto parametrů je náročný a tvůrci málo používaný, proto je potřeba ho zapnout pomocí BSDF vlajky, která ovšem v dokumentaci nebyla nikde uvedená (`BSDFFlags::NeedsDifferentials`).

Po přidání této vlajky již výpočet parciálních souřadnic fungoval. Ale okamžitě jsem si všiml zpomalení renderovacího času. Tedy výpočet diferenciálů se musí používat jen při použití mipmappingu nebo anizotropní filtrování. Proto vlajku `NeedDifferentials` v konstruktoru přiřadím jen při zapnutém filtrování textur.

6.10 Portable verze Mitsuby

Povedlo se mi vytvořit portable verzi Mitsuby, která se dá přenést na jiný počítač s operačním systémem Windows. Otestoval jsem ji na třech počítačích a na všech fungovala. Tato verze bude společně se sestaveným převodníkem přiložena na CD ve složce PortableProgramy.

Portable verze Mitsuby je vlastě sestavená Mitsuba společně s bigbtf pluginem. S touto verzí by měly jít pustit jednotlivé scény, které jsem připravil ve složce MitsubaTestFiles. U souborů bude jen potřeba změnit cesty, k souborům s texturami a u UBO81x81 reprezentace bude potřeba nastavit cestu k souborům s indexy, které jsou přiloženy ve složce MitsubaTestFiles/cubemaps.

U převodníku jsou přiloženy dll soubory knihovny OpenCV a mělo by jít pustit na různých počítačích. Převodník jsem testoval na třech počítačích a na všech mi fungoval. Navíc jsem dostal odezvu od vedoucího práce, že ho postupně na ÚTIA začínají používat pro převod BIG souborů do MIF souborů.

Kapitola 7

Testování a výsledky

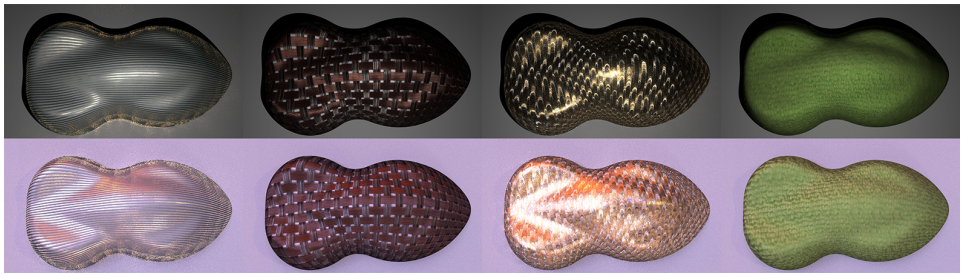
V této kapitole otestuji bigbtf plugin a zhodnotím jeho rychlost a vizuální přínos jednotlivých typů filtrování na připravených testovacích scénách. V jednotlivých kapitolách otestuji rychlost renderingu v závislosti na velikosti textury, počtu primitiv ve scéně, použitém médiu pro uložení textur a samotného rendereru a použití různých druhů filtrování. U testování filtrování navíc zhodnotím vizuální přínos těchto filtrování. Nad naměřenými výsledky jsem provedl analýzu a u každé kapitoly jsem sepsal výsledky této analýzy a doplnil je vhodnými vizualizacemi.

Testovací scény jsou uložené na přiloženém CD ve složce mitsubaTestFiles i s některými výslednými rendery. V této složce se bude nacházet několik menších MIF souborů s texturami. Další BTF textury jsou dostupné v databázi ÚTIA AV ČR, konkrétně na `http://btf.utia.cas.cz/?btf_mam2014`. MIF soubory mohou mít velikost až několik gigabajtů, proto provedu testování renderování z pevného disku (WD Blue (EZRZ), 3,5"- 2TB) a SSD disku (Seagate BarraCuda, 2,5"- 500GB), abych zjistil jak velký vliv má typ úložného média. Testování bude probíhat na Windows 10 s procesorem Intel Core i5–4460 3.2 GHz s grafickou kartou NVIDIA GeForce GTX 960. Počítač má 24 GB RAM paměti (DDR3 1600mhz). Všechna testování budou probíhat na scénách s použitím třiceti dvou vzorků na pixel, hloubkou rekurze pathtracingu osm a velikostí výsledného obrázku 1920×1080 a s použitím čtyř vláken (*threads*) v Mitsuba rendereru.

Měření času renderingu probíhá pomocí Mitsuby, která tento údaj předává. Celkový čas běhu Mitsuby, z kterého pak pomocí odečtení času rendringu počítám čas inicializace, je měřen pomocí nástroje Measure-Command ve Win-

Jméno souboru	Reprezentace	Pojmenování	velikost	počet obrázků
fabric106_uniform	Uniform	uniform	429 MB	3660
MAM2014_007	UBO81x81	grass	1,3 GB	6561
MAM2014_011	UBO81x81	gold	4,5 GB	6561
MAM2014_016	UBO81x81	basket	3,2 GB	6561
ace001	UBO81x81	ace	2,7 GB	6561
schlenk075_BTFthph	CoatingSpecial	chameleon	25 MB	216
schlenk075_BTFthtd	CoatingRegular	chameleon2	12 MB	108
schlenk082_BTFthph	CoatingSpecial	silver	25 MB	216
schlenk082_BTFthtd5	CoatingRegular	silver2	12 MB	108

Tabulka 7.1: Seznam BTF textur ve formátu MIF použitých pro testování. Grass, gold a basket můžete nalézt v databázi MAM2014 na webových stránkách ÚTIA AV ČR, zbylé soubory mi poskytl vedoucí práce.



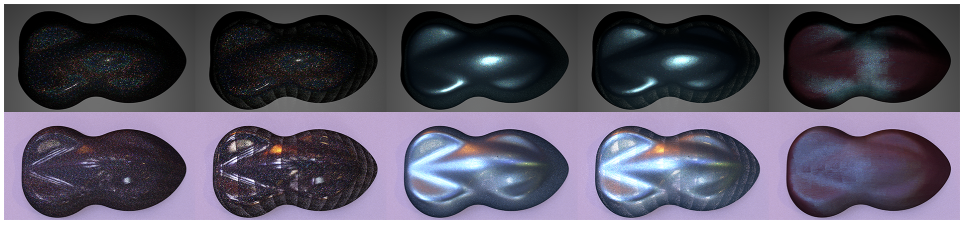
Obrázek 7.1: Ukázka UBO81x81 textur na scéně car3Top.xml a car3Top_env.xml. Postupně jsou zde textury: ace, uniform, gold, grass.

dows PowerShell, ve kterém probíhá i samotné spouštění Mitsuby. Všechna data která byla naměřena během testování, jsou uložena na CD ve složce *Testování* v souboru testovani_data.xlsx (pro každé testování je vytvořen samostatný list).

Pro testování budu používat devět BTF textur různých velikostí od 25 MB do 4,5 GB a jejich až čtyřikrát větší varianty s implementovanými variantami filtrování (MIPMAP_LINEAR, MIPMAP_WEIGHTED, ANIZO_1x, ANIZO_4x). Pro porovnání některých časů budu používat šachovnicovou texturu, která je součástí Mitsuby. BTF textury, které použiji, obsahují zástupce všech reprezentací BTF dat podporovaných bigbtf pluginem. Textury jsem si pro lepší práci pojmenoval. V tabulce 7.1 lze najít přehled pracovních jmen a jmen souborů, které jsem používal[1]. Některé z těchto textur jsou umístěny na CD s přílohami. Vizualizaci textur na modelu auta s reprezentací UBO81x81, můžete nalézt na obrázku 7.1. Ukázky zbylých textur najdete na obrázku 7.2. Ukázky textur byli vytvořeny na scéně car3Top s bodovým

[1]Ve složce MitsubaFiles je soubor materials.xml, kde jsou pro jednotlivé materiály vytvořeny bsdf tagy, které stačí přiřadit k objektu ve scéně.

a enviromentálním světlem. Jednotlivé obrázky lze najít na přiloženém CD ve složce MitsubaTestFiles ve slože car, kde jsou všechny tyto vizualizace uloženy ve vysokém rozlišení.



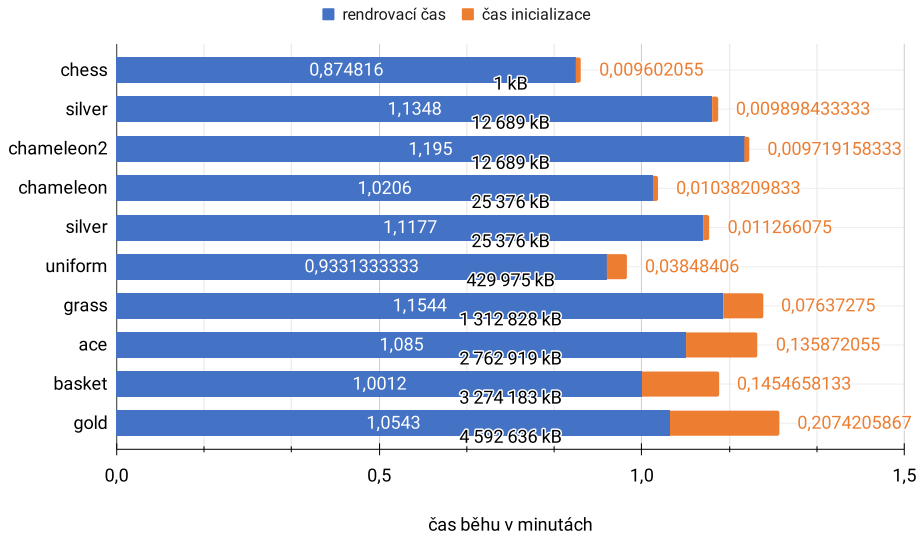
Obrázek 7.2: Ukázka CoatingRegular, CoatingSpecial a uniformní textury na scéně car3Top.xml a car3Top_env.xml. Postupně jsou zde textury: chameleon, chameleon2, silver, silver2, uniform.

7.1 Testování různých textur na jednom objektu

V této sekci budu testovat rychlost renderingu v závislosti na velikosti textury a typu osvětlení (bodové světlo, mapa osvětlení). Některé scény budou obsahovat více textur s různými velikostmi. Proto jsem se rozhodl začít tímto testem. Testování bude provedeno na scéně car3Top.xml a car3Top_env.xml, kde budou na objekt auta aplikovány textury uvedené v tabulce 7.1, které budu poté používat v ostatních testech a porovnám jejich renderovací časy a čas běhu celého renderingu i s jeho inicializací (načtením textury do paměti). Předpokládám, že čas renderigu by měl být u různých textur téměř stejný.

Během testování jsem zjistil, že pokud se některá z textur použila v předešlém renderování zůstává nejspíše uložená v nějaké cache paměti disku nebo přímo v RAM paměti, protože při druhém použití samé textury v jiné scéně, byla inicializace o dost kratší (z 0,75 minuty se inicializace zkrátila na 0,24 minuty). Pro validní výsledky, proto vždy pustím scénu vícekrát a beru minimum z načtení. Takže reálný čas inicializace při spuštění renderingu scény bude tímto jevem ovlivněn a bude typicky delší. Velmi záleží na použitém úložném médiu, jeho aktuálním zatížení a rozmístění texturních dat na tomto médiu. Pro toto testování používám pevný disk, na kterým neběží systém, takže by neměl být nijak jinak vytížený.

V testování jsem zjistil, že enviromentální nasvícení scény je náročnější na výpočet, protože u všech scén s tímto osvícením se prodloužil renderovací čas. Konkrétně celkový čas scén s enviromentálním nasvícením tvořil 58,2 % z celkového času všech dvaceti renderovaných scén. Zajímavější informace,



Obrázek 7.3: Graf ukazuje poměr mezi časem inicializace a renderovacího času různých BTF textur na scéně car3Top.xml, která používá bodové osvětlení. Pro porovnání je přiložen renderovací čas bitmapové textury šachovnice.

kterou lze z grafu 7.3 zjistit, je že velikost textury přímo neovlivňuje renderovací čas. Tento fakt bude nejspíše způsobem tím, že každá textura má jinou struktura a jinak odráží světlo. Tedy u některých textur se nemusí použít vždy rekurze zanořování paprsku osm, ale bude zde nějaká logika pro dřívější ukončení rekurze. Naopak u lesklých textur bude docházet k častým odrazům paprsků a renderovací čas se prodlouží. Tedy renderovací čas ovlivňují vlastnosti textury a ne její velikost. Čas inicializace samozřejmě velikost textury ovlivní.

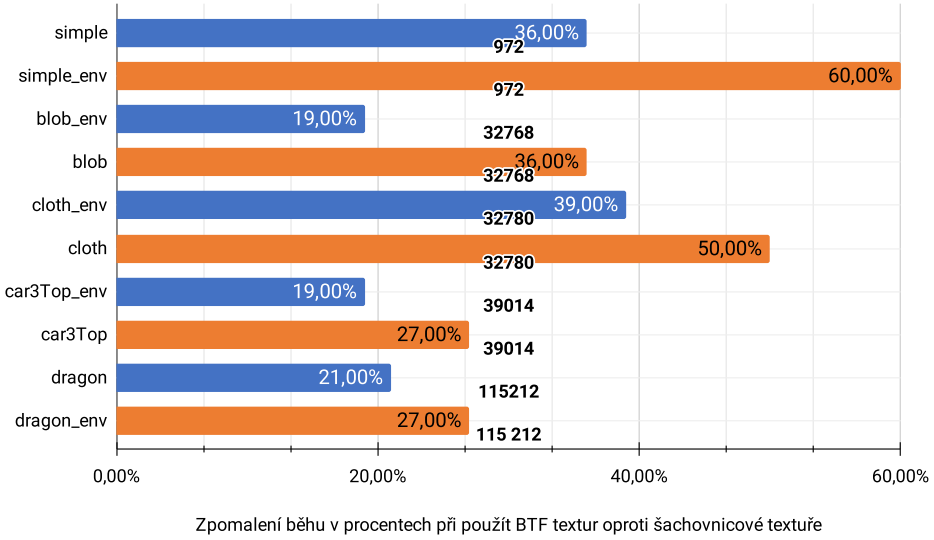
7.2 Testování na různě složitých scénách

Pro toto testování jsem vytvořil pět scén s názvy *simple*, *car3Top*, *blob*, *cloth*, *dragon*. Tyto scény vždy obsahují jeden objekt s různým počtem primitiv. Každá scéna obsahuje podložku, aby mohlo docházet k odrazu paprsků světla mezi objektem s BTF texturou a podložkou. Jedinou výjimkou je scéna *blob*, která je bez podložky. Tato scéna slouží pro ukázku rozdílu časové náročnosti pathtracingu scény s dvěma objekty a s jedním. Konkrétní počet primitiv jednotlivých scén můžete najít v tabulce 7.2.

Těchto pět scén zkusím vyrenderovat s různými texturami o různé velikosti a to konkrétně - *chameleon*, *chameleon2*, *uniform*, *basket*, *gold* a vestavěnou

Název scény	Počet primitiv	Počet objektů	Vizualizace
simple	972	2	obrázek 6.5
blob	32 768	1	obrázek 7.13
cloth	32 780	2	obrázek 4.4
car3Top	39 014	2	obrázky 7.1 a 7.2
dragon	115 212	2	obrázek 7.12
complex_front	219 762	6	obrázek 7.6

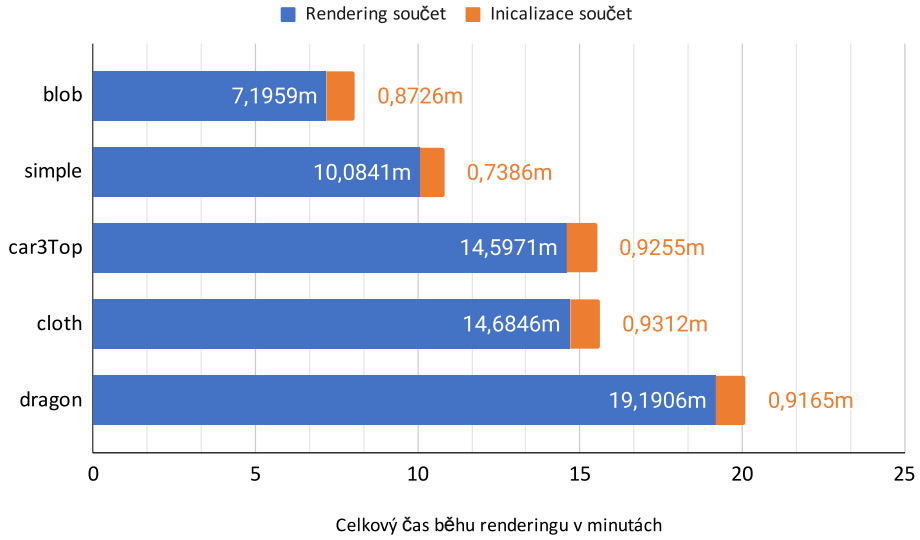
Tabulka 7.2: Přehled testovacích scén, počet primitiv a celkový počet objektů. Každá z těchto scén má i duplikát s příponou _env s mapou prostředí.



Obrázek 7.4: Zpomalení renderovacího času při použití BTF textur a pluginu bigbtf oproti použití šachovnicové textury. Černý text uprostřed udává počet primitiv.

texturou šachovnice (*chess*), abych mohl porovnat zpomalení renderingu při použití BTF textury. Každou scénu vyrenderuji vždy se stejnou pozicí kamery, jen s různými druhy osvětlení. Při renderování bude vždy použito třicet dva vzorků na pixel a hloubka rekurze osm. Výsledné obrázky budou v rozlišení 1920×1080.

První vlastností, které jsem si všiml, je že velikost textury nemá vliv na samotný renderovací čas. Samozřejmě inicializace textury trvá o něco déle, ale pokud jste tuto texturu již předtím renderovali a máte dostatečnou RAM paměť, zůstane uložená a rozdíly v této inicializaci jsou v řádech jednotek sekund. Renderovací čas samozřejmě ovlivňuje druh osvětlení a vlastnosti textury. Například textura *chameleon* se velmi často umísťovala mezi texturami s nejdelším časem renderingu a přitom má velikost jen 27 MB.

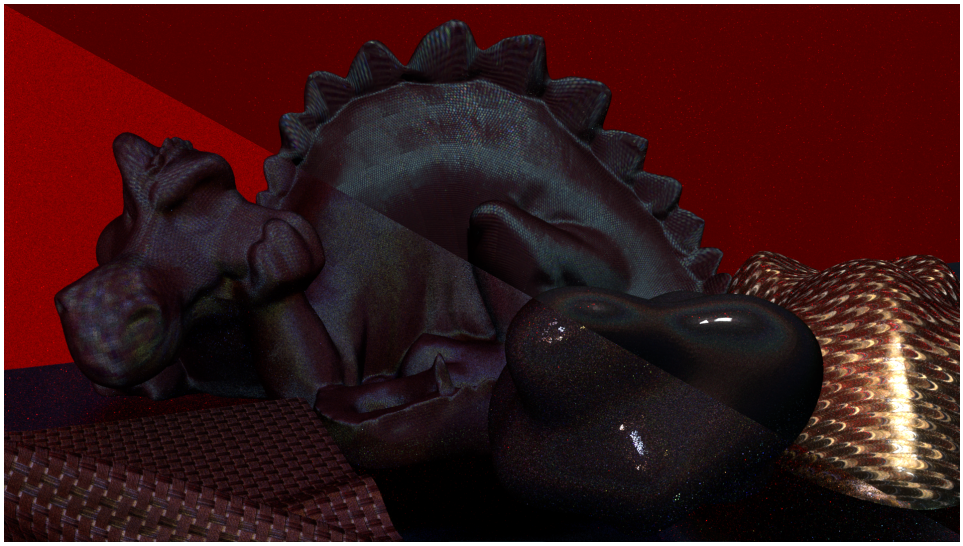


Obrázek 7.5: Čas scény se skládá z časů naměřených při obou typech osvětlení a použití všech šesti testovaných textur. Celkově jsem tedy naměřil 72 renderování scén.

Dále jsem zkoumal zpomalení renderovacího času při použití BTF textur renderovaných pomocí bigbtf pluginu oproti použití renderování šachovnicové textury. Vždy jsem porovnával čas renderingu šachovnicové textury oproti nejdelšímu času z renderovaných BTF textur. Rozlišoval jsem zde i typy osvětlení, ale na zpomalení to nemělo vliv. Neporovnával jsem celkový čas, ale jen čas renderingu udávaný Mitsubou.

Jak je vidět na grafu 7.4 zpomalení dochází a to ve velkém rozptylu od 19 % až k 60 % u scény *simple*. Samozřejmě je zde zkreslení při braní maxima z celkových textur, ale průměrné zpomalení se pohybuje okolo 20 %.

Změřil jsem čas tří scén, které obsahují velmi podobný počet primitiv – *blob*, *car3Top* a *cloth*. Z dat vyplynulo, že časy *car3Top* a *cloth* si jsou velmi podobné, jak lze vidět na grafu 7.5. Čas u scény *blob* byl poloviční, i když má téměř totožný počet primitiv. Což způsobilo to, že obsahuje jen jeden objekt. Tedy zde nemůže tedy dojít k odrazu paprsku od druhého objektu. Na základě dat v grafu 7.5 lze říci, že časy pro jednotlivé textury se liší podle různých parametrů scény, ale při porovnání času renderingu všech testovaných textur rozhoduje o času hlavně počet objektů a primitiv ve scéně.



Obrázek 7.6: Vlevo se nachází scéna osvětlená pomocí mapy prostředí a vpravo pomocí jednoho bodového světla. Scéna z enviromentální mapou obsahuje mnohem více šumu.

7.3 Testování čtecí metody

V této kapitole porovnám na scéně *blob* a textuře *uniform* renderování pomocí načtení dat do paměti a čtení dat přímo z disku. Ještě zde porovnám rozdíl mezi SDD a pevným diskem, protože zde mohou být také rozdíly. Předpokládám, že čtení z disku bude velmi pomalé, proto jsem zvolil velmi jednoduchý test s relativně malou texturou, u větší textury by byly rozdíly ještě větší, kvůli většímu rozpětí pohybu hlavičky pevného disku. Tento test je spíše pro ilustraci a zdůraznění toho, že plugin podporuje i renderování přímo z disku a porovnání mezi SSD diskem a pevným diskem. Test bude proveden na scéně *blob* s upravenou hloubkou rekurze na čtyři, počtem vzorků na pixel čtyři a sníženém rozlišení na 640×360[2].

Z dat v tabulce 7.3 lze vidět, že čtení z paměti je oproti čtení z disku obecně o několik řádů rychlejší (u SSD disku je čtení z paměti 61krát rychlejší a u pevného disku je využití paměti 76krát rychlejší). V tabulce 7.3 lze také vidět, že renderování pomocí SSD disku je o 37 % rychlejší oproti použití pevného disku. Tento fakt lze vidět i na časech inicializace při použití RAM paměti, kde je SSD asi o 10 % rychlejší. Nejvíce mě zaujalo zrychlení samotného renderovacího času, protože u SSD byl cca o 100 ms rychlejší. K těmto zlepšením došlo pouhým přesunem samotného rendereru na SSD

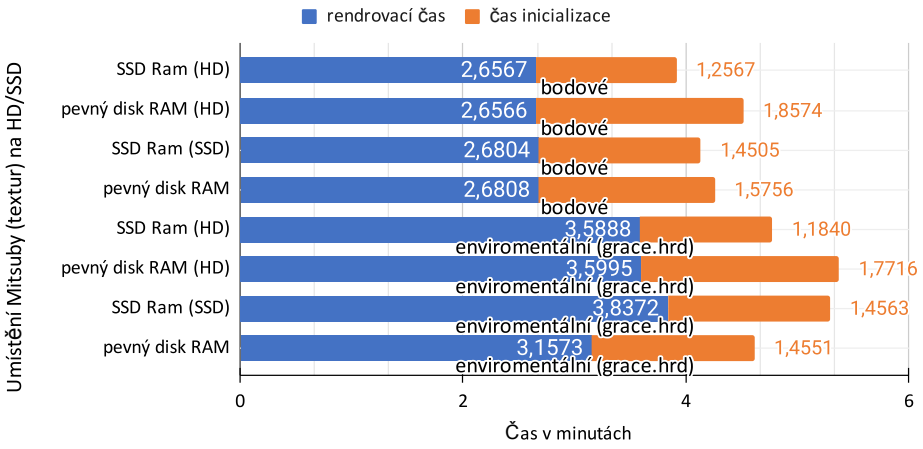
[2]Tuto konfiguraci lze najít na CD v souboru MitsubateTestFiles/Mitsubacomands.txt v položce debug, jsou zde i ukázky volání ostatních scén, jen bude potřeba změnit cesty k souborům se scénou.

Způsob čtení	Čas běhu (minuty)	Renderovací čas	Čas inicializace
pevný disk RAM	0,03662 min	0,00871666 min	0,02791 min
SSD RAM	0,03260 min	0,00701 min	0,02559 min
pevný disk	2,77196 min	2,7616 min	0,01036 min
SSD	2,01511 min	2,007 min	0,00811 min

Tabulka 7.3: Výsledné časy při testování čtení z disku/RAM paměti a použití různých typů disků.

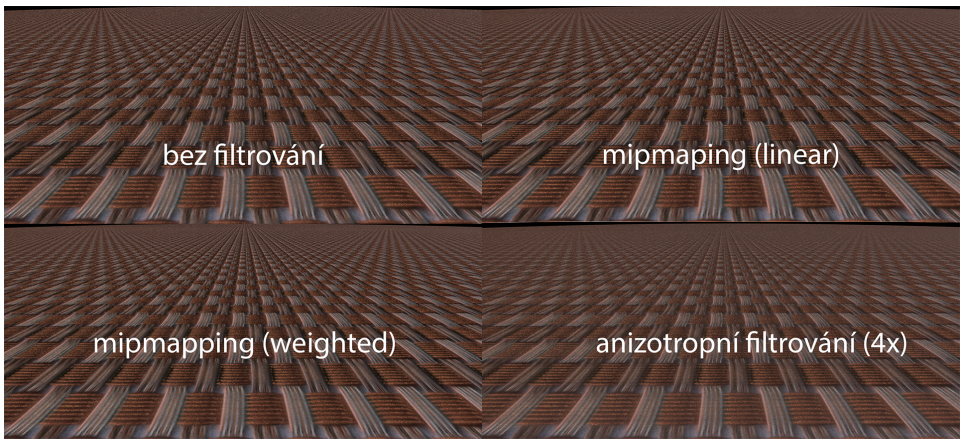
disk, přičemž soubory s texturami zůstaly na pevném disku, což vzniklo mou chybou, ale tyto výsledky mě zaujaly a proto jsem je zde uvedl.

Na základě předešlých výsledků jsem se rozhodl provést další test na scéně complex_front, která má šest objektů a její podobu můžete vidět na obrázku 7.6. Na pět z nich jsem umístil BTF textury o celkové velikosti 8,3 GB. V testu porovnám rozdíly v rychlosti renderovacího času, při použití SSD disku a pevného disku při uložení dat do RAM paměti. Nyní už se standardními parametry testování, které jsem používal i pro ostatní měření. Porovnám zde i možnost uložení textur na SSD disku, což by mělo zrychlit renderovací čas, hlavně čas inicializace.



Obrázek 7.7: Data z testování umístění Mitsuby na různých úložných médiích a umístění textur na různých typech disků. Pro pevný disk je použita v grafu zkratka HD. V závorce je uveden disk, na kterém byly uloženy textury.

Z dat, které jsem vizualizoval v grafu 7.7 vyplývá, že změna uložení textur z pevného disku na SSD disk neznamenala urychlení inicializace ani urychlení samotného renderovacího času, který zůstal téměř identický. U času inicializace se dokonce čas pro inicializaci zvětšil, což může být tím, že na tomto



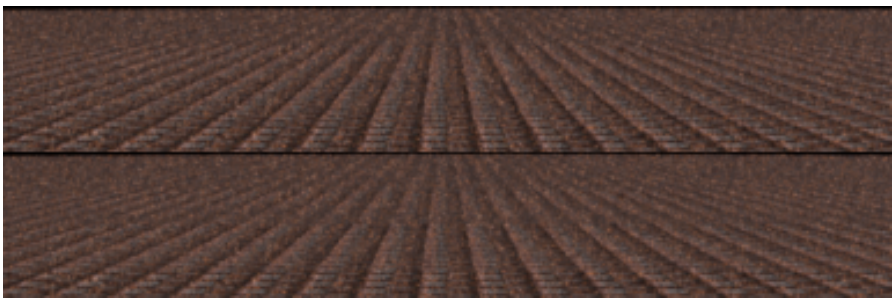
Obrázek 7.8: Srovnání různých verzí filtrování s BTF texturou basket. Pro zdůraznění filtrování byl použit parametr pluginu level=3.

disku běží i systém Windows a nejspíše nemohlo proběhnout již zmiňované cachování, které u opětovného použití textury u pevného disku velmi zrychlilo inicializaci. Za hlavní a pro mě překvapivý výsledek tohoto testu považuji to, že větší vliv na výkon rendereru má to na jakém disku je uložen samotný renderer, než na jakém typu disku jsou uložená texturní data.

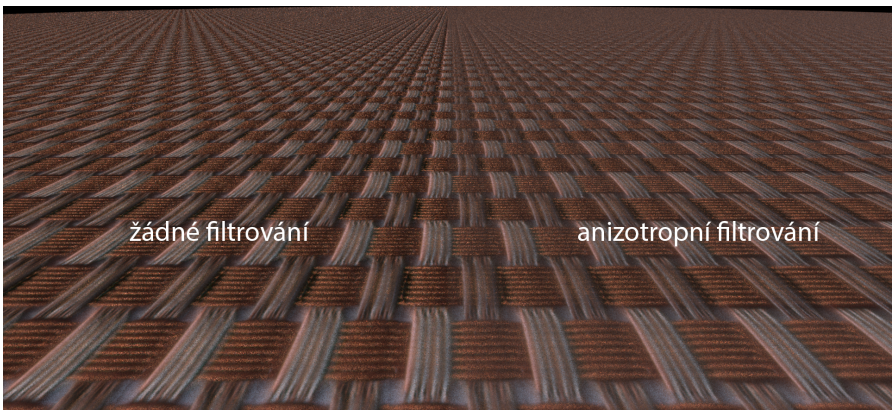
7.4 Testování filtrování

V této kapitole rozeberu vizuální přínos mipmappingu a anizotropního filtrování na renderované výsledky. Také provedu analýzu časové náročnosti, při použití souboru s mipmapami s anizotropním filtrováním oproti souboru čistě s mipmapou, případně samotné (původní) textuře. Renderovací časy by měly být téměř totožné při použití stejného druhu filtrování, jen by se měla prodloužit inicializace. Předpoklad je, že by anizotropní filtrování i mipmapping měly zpomalit renderovací čas a pro zpomalení existuje hned několik důvodů. Zaprvé musí probíhat výpočet differenciálů, z kterých se počítají úrovně pro mipmapu. Zadruhé se musí načítat větší soubor s BTF texturou a i přístup k tomuto většímu poli by měl trvat déle. Navíc u některých variant filtrování (MIPMAP_WEIGHTED a ANIZO_4x) dochází k další interpolaci, která by měla také znamenat zpomalení renderovacího času.

Co se týká vizuálních výsledků, tak všechny druhy filtrování na první pohled fungují. Pro zdůraznění filtrování jsem nastavil parametr level pluginu bigbtf na tři. Zajímavé je, že obě varianty anizotropického filtrování dávají stejný výsledek. Vizuální porovnání jednotlivých druhů filtrování na scéně mipmap.xml můžete vidět na obrázku 7.8. Filtrování zjevně funguje



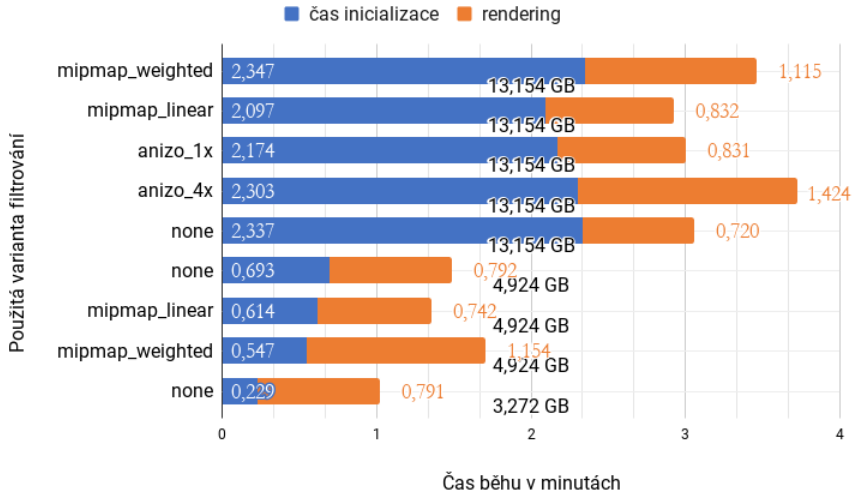
Obrázek 7.9: V horní části obrázku je výřez z MIPMAP_LINEAR a v dolní se nachází výřez z MIPMAP_WEIGHTED. V dolní části je přechod mezi úrovněmi plynulejší.



Obrázek 7.10: Vliv anizotropického filtrování na texturu. Při použití filtrování dochází k rozmazání textury směrem k horizontu.

a hlavně u anizotropního filtrování je velmi zřetelně vidět rozostřenou zadní část scény. Při bližším prozkoumání výsledků MIPMAP_LINEAR a MIPMAP_WEIGHTED je vidět, že u lineárního mipmapingu jsou vidět hranice, které znamenají přechod mezi jednotlivými úrovněmi, což u druhé verze nelze pozorovat. Tento jev lze nejlépe pozorovat při rychlém přepnutí mezi oběma obrázky. Tyto rozdíly se snaží ukázat obrázek 7.9.

Při testování jsem si všiml, že anizotropické filtrování zesvětluje texturu a zmenšuje její kontrast. Snažil jsem se najít problém v kódu a konzultoval jsem tento problém i s vedoucím práce, ale nepovedlo se nám nalézt příčinu zesvětlení. Vzhledem k tomu, že všechny obrázky jsou generovány ze stejného souboru a jenom anizotropické filtrování je světlejší, nemůže být chyba při generování anizotropické varianty textury. Poté jsme vyloučili chybu v interpolaci při ANIZO_4x, protože v ANIZO_1x se interpolace přes více vážených hodnot nepoužívá a scéna je také světlejší. Vyloučili jsme i chybu v generování koordinátorů v textuře. Zbylé části kódu jsou pro všechny druhy filtrování společné, proto tento problém nechávám otevřený. Ačkoliv bych rád vysvětlil tento jev, nepovedlo se mi najít chybu v kódu, která by mohla tento jev



Obrázek 7.11: Srovnání časové náročnosti při použití různých druhé filtrování na scéně mipmap.xml s texturou basket.

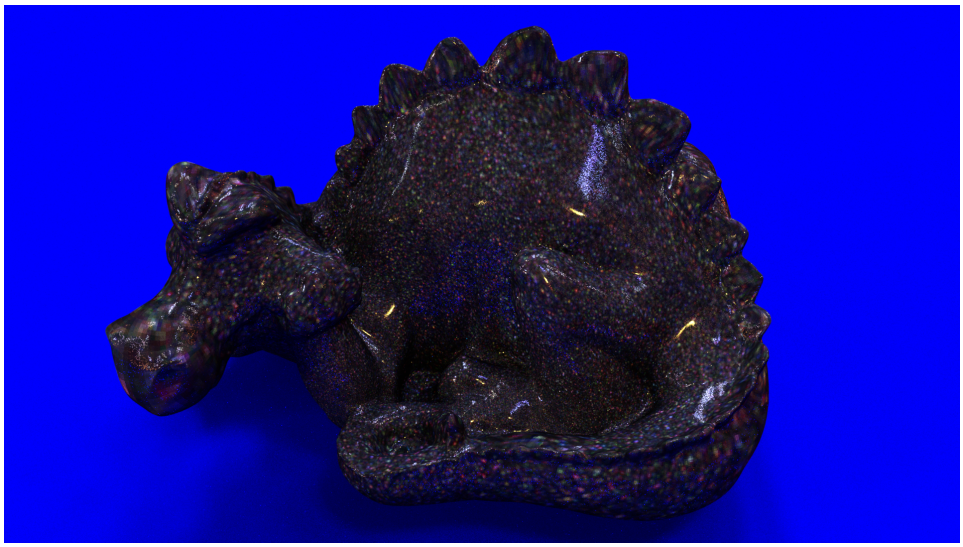
způsobovat a všechny možnosti kde by podle mě mohla nastat chyba jsem již prozkoumal a vyloučil.

Test vizuálního přínosu anizotropního filtrování a mipmapingu dopadl velmi dobře. Filtrování na uvedených scénách funguje a parametr level, umožní jeho sílu ovlivnit v případě, že by se uživatelům nelíbila hranice kde filtrování začíná. I u anizotropního filtrování navzdory problému se zesvětlením textury filtrování funguje, jak můžeme pozorovat na obrázku 7.10, kde na pravé půlce obrázku je použito anizotropní filtrování a na levé straně je textura vyrendrována bez jakéhokoliv filtrování.

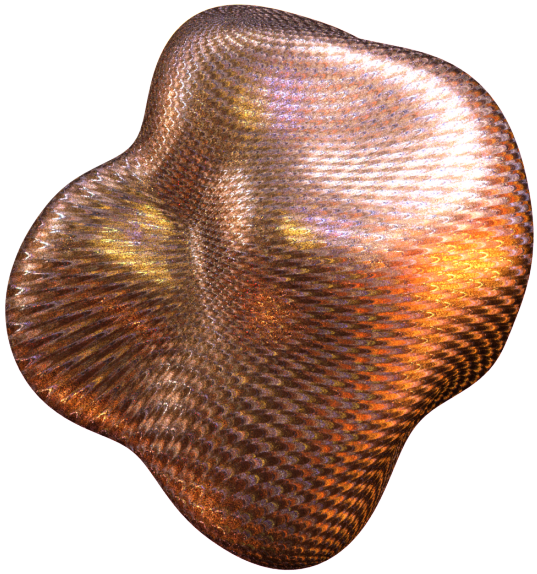
V následujících odstavcích popíši vliv filtrování na rychlost renderovacího času. Z naměřených časů, jsem zjistil, že samotná velikost souboru neovlivní renderovací čas při použití souboru s anizotropní mipmapou pro rendrování bez filtrování nebo mipmaping. To samé platí i pro ostatní možné varianty. Rendrování scény samozřejmě bude trvat déle, protože textura obsahující mipmapu je větší a proto inicializace a načtení textury do paměti bude trvat déle. Takže pokud chceme ušetřit místo na disku je lepší mít uloženou texturu ve verzi s anizotropním filtrováním a používat ji pro všechny druhy renderování, což neovlivní čas renderingu.

Dále bych rád probral vliv filtrování na renderovací čas. Vizualizaci naměřených dat můžeme vidět na obrázku 7.11. Přičemž z testů vyplynulo, že filtrování MIPMAP_LINEAR a ANIZO_1x jsou o deset až patnáct procent pomalejší než klasické rendrování, což nejspíše způsobuje výpočet prostoro-

vých diferenciálů v Mitsubě označovaných jako `duv_dx` a `duv_dy`. Použití MIPMAP_WEIGHTED zpomalý renderovací čas o polovinu, což je způsobeno dvojnásobným přístupem k pixelu a předpokládal jsem to. Při použití anizotropického filtrování, které interpoluje mezi celými hodnotami jednotlivých úrovní (ANIZO_4x) dojde k prodloužení renderovacího času na dvojnásobek, což vzhledem k tomu, že se čte 4krát více pixelů není tak velké zpomalení. Ačkoliv jak jsem již zmínil u anizotropního filtrování tato interpolace v testech nevykázala významné vizuální rozdíly. Tedy při použití anizotropního filtrování bych na základě provedených testů doporučil variantu ANIZO_1x.



Obrázek 7.12: Scéna *dragon* s BTF texturou *chameleon* reprezentující autolak. Ve scéně je použito enviromentální nasvícení.



Obrázek 7.13: Scéna *blob* s BTF texturou *gold*. Scéna je nasvícena pomocí mapy prostředí.

Kapitola 8

Závěr

V rámci diplomové práce jsem nastudoval a popsal problematiku textur. Vysvětlil jsem pojem vizuální textura a jak lze fyzické materiály popsat pomocí funkce a jaké funkce pro popis textur existují (GRF, BRDF, SVBRDF, BTF) a jaké se už i reálně používají. Prozkoumal jsem dnes používané formáty pro ukládání materiálu (.mtl a .axf). Díky této analýze jsem pochopil proč vznikl formát BIG, který jsem podrobně analyzoval. Na základě analýzy a testování BIG formátu v bigbtf pluginu, jsem odhalil chybu v práci s částečnou pamětí při vícevláknovém zpracování. Na základě mých informací o problémech s BIG knihovnou a dalších faktů se na ÚTIA AV ČR rozhodli vytvořit novou verzi formátu pro ukládání BTF dat, která vychází z BIG formátu, ale je napsána úplně od základů s názvem MIF. Na jejímž vývoji a především testování jsem spolupracoval. Díky MIF knihovně se renderovací čas bigbtf pluginu mírně zrychlil a to hlavně díky implementaci funkce `getPixel`, která umí získat celý pixel, což BIG knihovna neuměla.

V další části mé diplomové práce jsem analyzoval některé aktuálně využívané renderery a sepsal jejich výhody a nevýhody a možnosti jejich rozšiřitelnosti. Některé z těchto rendererů pro ukládání materiálů používají své vlastní formáty. Dokonce mají i své vlastní funkce pro definice těchto materiálů, což může způsobovat problémy při přenosu scén mezi renderery. Pokud by existoval otevřený formát, který by dokázal ukládat BTF textury, ale i další textury, mohlo by se rozšířit jejich používání.

Dále jsem se zabýval průzkumem implementačních možností podpory formátu pro uložení BTF textur a analýzy podpory filtrování (mipmapping a anizotropního filtrování). Na základě zjištěných informací jsem se rozhodl

implementovat podporu pro Mitsuba renderer, protože umožňuje rozšíření pomocí pluginu. Vytvořil jsem tedy rozšíření s názvem bigbtf, které umožňuje vizualizaci BTF textur a implementuje i anizotropní filtrování a mipmapping. Tento plugin používá pro komunikaci třídu `BigRender`, která je nezávislá na Mitsubě a může být použita jako rozhraní pro komunikaci mezi rendererem a texturami v MIF formátu i při implementaci v ostatních rendererech.

Pro generování mipmap a jejich anizotropních variant jsem vytvořil převodník BigConvert, který slouží k převodu původní verze BIG souborů do MIF formátu a také pro případné generování souborů obsahující mipmapy, které bigbtf plugin vyžaduje pro filtrování. BigConvert vznikl, protože na ÚTIA AV ČR mají většinu BTF textur uložených v původní verzi BIG formátu a v této verzi jsou uloženy soubory i v databázi MAM2014 na webových stránkách (`http://btf.utia.cas.cz/?btf_mam2014`). Proto bylo potřeba vytvořit nástroj pro jednoduchý převod, aby v databázi na webu nemusely být uloženy textury jak ve staré verzi BIG formátu, tak i v MIF verzi.

Nakonec jsem vytvořil několik scén, na kterých jsem otestoval rychlost rendrování BTF textur a vizuální přínos anizotropického filtrování a mipmapingu pomocí bigbtf pluginu. Na těchto scénách jsem použil několik různých textur používajících různé reprezentace pro uložení obrázků. Zajímavý výstup z tohoto testování je, že velikost textury nemá vliv na čas renderingu, protože u lesklých textur dochází k většímu přenosu světla a více odrazům paprsků, které prodlouží renderovací čas. Velmi zajímavé je i srovnání použití pevného a SSD disku jako média, na kterém jsou uloženy textury nebo samotný renderer.

Rozšíření pro Cycles renderer jsem neimplementoval. Dal jsem přednost optimalizaci třídy `BigRender` a testování knihovny MIFlib v Mitsuba rendereru. Hlavně kvůli přechodu na MIF knihovnu, který proběhl v půlce dubna mi už ani nezbývalo moc času pro případné nastudování rozhraní dalšího rendereru a implementaci tohoto rozšíření.

Do budoucna by bylo dobré pomocí třídy `BigRender` implementovat podporu MIF formátu i do dalších opensourcových rendererů. Do převodníku MIF souborů by mohla být přidána možnost uložení více BTF textur do jednoho MIF souboru a v bigbtf pluginu by se mohlo implementovat načtení více BTF textur z tohoto souboru. Za prozkoumání by stála i možná úprava distribuční funkce pro rozptyl paprsků.[1] U BTF textury by byly uloženy informace i o rozptylu paprsků na polokouli. Mohlo by to pomoci zlepšit vizuální výsledky renderování BTF textur.

[1] V Mitsubě by to znamenalo úpravu funkce `pdf`.

Bibliografie

1. RICHTEROVÁ, Alena. textura – misálové písmo písmo. *Databáze Národní knihovny ČR* [online]. © 2012 [cit. 2020-10-25]. Dostupné z: `https://aleph.nkp.cz/F/?func=find-c&local_base=KTD&ccl_term=wtr%5C%3DTextura`.
2. HAINDL, Michal; FILIP, Jiří. *Visual Texture: Accurate Material Appearance Measurement, Representation and Modeling. Advances in Computer Vision and Pattern Recognition*. Springer, 2013.
3. HELMHOLTZ, Hermann von. *Handbuch der physiologischen Optik*. Voss, 1867.
4. PHONG, Bui Tuong. Illumination for computer generated pictures. In: Communications of ACM 18, 1975, s. 311–317. 6.
5. BLINN, James F. Models of light reflection for computer synthesized pictures. *Proc. 4th annual conference on computer graphics and interactive techniques.* [online]. 1977, s. 192 [cit. 2020-10-25]. Dostupné z: `https://dl.acm.org/doi/10.1145/965141.563893`.
6. TORRANCE, K.; SPARROW, E. Theory for Off-Specular Reflection from Roughened Surfaces. In: J. Optical Soc. America, 1976, s. 1105–1114. 57.
7. COOK, R.; TORRANCE, K. A reflectance model for computer graphics. *Computer Graphics (SIGGRAPH '81 Proceedings)*. 1981, s. 301–316.
8. THIAGO, Ize. *Anisotropic Ward BRDF* [online obrázek] [cit. 2020-11-23]. Dostupné z: `http://www.sci.utah.edu/~thiago/cs7650/hw6/FinalOutput.png`.

9. DESCHAINTRE, Valentin; AITTALA, Miika; DURAND, Frédo; DRETTAKIS, George; BOUSSEAU, Adrien. Single-Image SVBRDF Capture with a Rendering-Aware Deep Network. *ACM Transactions on Graphics (SIGGRAPH Conference Proceedings)*. 2018, roč. 37, č. 128, s. 15. Dostupné také z: `http://www-sop.inria.fr/reves/Basilic/2018/DADDB18`.
10. SATTLER, M.; SARLETTE, R.; KLEIN, R. Efficient and Realistic Visualization of Cloth. In: *Eurographics Symposium on Rendering*. 2003, s. 167–178.
11. RUSINKIEWICZ, S.M. A new change of variables for efficient BRDF representation. In: *Rendering techniques' 98*. 1998, s. 11–22.
12. FILIP, J.; VÁVRA, R. Image-based Appearance Acquisition of Effect Coatings. *Computational Visual Media*. 2019, č. 5, s. 73–89.
13. DIANE RAMEY, Linda Rose; TYERMAN, Lisa. MTL material format (Lightwave, OBJ). *Alias|Wavefront, Inc.* [online]. 1995 [cit. 2020-11-26]. Dostupné z: `http://paulbourke.net/dataformats/mtl/`.
14. X-RITE. Appearance Exchange Format. *https://www.xrite.com/* [online]. 2019 [cit. 2020-12-21]. Dostupné z: `https://www.xrite.com/learning-color-education/whitepapers/appearance-exchange-format-axf`.
15. X-RITE. *Appearance Exchange Format* [online obrázek] [cit. 2020-12-22]. Dostupné z: `https://www.xrite.com/-/media/xrite/images/whitepaper/axf/axf-whitepaper-800x400.jpg?h=400&w=800&la=en&hash=E815ECD362F459AECFD28072C31A121DB65C6E72`.
16. FILIP, Jiří; KOLAFOVA, Martina; HAVLÍČEK, Michal; VÁVRA, Radomír; HAINDL, Michal; RUSHMEIER, Holly. Evaluating physical and rendered material appearance. *The Visual Computer*. 2018, roč. 34. Dostupné z DOI: `10.1007/s00371-018-1545-3`.
17. VÁVRA, Radomír. *BIG File Format* [`https://github.com/RadomirVavra/BIG`]. GitHub, 2018.
18. VÁVRA, Radomír; FILIP, Jiří. *Multi-Image Format Libraryt* [online]. GitHub, 2021 [cit. 2021-04-30]. Dostupné z: `%5Curl%7Bhttps://github.com/RadomirVavra/MIFlib%7D`.
19. NIMIER-DAVID, Merlin; VICINI, Delio; ZELTNER, Tizian; JAKOB, Wenzel. Mitsuba 2: A Retargetable Forward and Inverse Renderer. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*. 2019, roč. 38, č. 6. Dostupné z DOI: `10.1145/3355089.3356498`.
20. BLENDER FOUNDATION. *Cyles – renderer*. 2020. Verze 2.9.0. Dostupné také z: `https://www.cycles-renderer.org/`.
21. CHAOS SOFTWARE. *V-Ray*. 2020. Verze 5.0.0. Dostupné také z: `https://www.chaosgroup.com/`.
22. CHAOS CZECH. *Corona*. 2020. Verze 6.0.0. Dostupné také z: `https://corona-renderer.com/`.

23. SOLID ANGLE. *Arnold*. 2020. Verze 6.1. Dostupné také z: `https://www.arnoldrenderer.com/`.

24. CZESIEK. *Cycles vs Vray vs Arnold* [online obrázek] [cit. 2020-12-22]. Dostupné z: `https://blenderartists.org/t/cycles-vs-vray-vs-arnold/1132409`.

25. "FRONT PORCH DIGITAL, Masstech; SGL". *Modern House Exterior Rendering | Vray for Sketchup*. Architecture Inspirations, © 2019. Dostupné také z: `https://www.archinspirations.com/blog/modern-house-exterior-rendering-vray-for-sketchup`.

26. JAKOB, Wenzel. *Enoki: structured vectorization and differentiation on modern processor architectures*. 2019. https://github.com/mitsuba-renderer/enoki.

27. KOVACIK, Martin. *Proxenta Residence* [online obrázek] [cit. 2021-05-03]. Dostupné z: `https://www.cycles-renderer.org/#&gid=psgal_109_1&pid=14`.

28. BLENDER FOUNDATION. *Blender*. 2020. Verze 2.9.0. Dostupné také z: `https://www.blender.org/`.

29. 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`.

30. PHARR, Matt; JAKOB, Wenzel; HUMPHREYS, Greg. *Physically Based Rendering: From Theory to Implementation*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN 0128006455.

31. MIGUEL, San. *Countryard* [online obrázek] [cit. 2021-05-03]. Dostupné z: `https://pharr.org/matt/blog/images/sanmiguel-courtyard-second.jpg`.

32. SIKACHEV, Peter. *Mipmap filtering* [online obrázek] [cit. 2020-12-28]. Dostupné z: `http://petersikachev.blogspot.com/2018/10/minlod-cheap-method-to-increase-texture.html`.

33. TIŠNOSVKÝ, Pavel. *Mip map* [online obrázek] [cit. 2020-12-28]. Dostupné z: `https://www.root.cz/clanky/opengl-25-mipmapping/`.

34. AKENINE-MÖLLER, Tomas; NILSSON, Jim; ANDERSSON, Magnus; BARRÉ-BRISEBOIS, Colin; TOTH, Robert; KARRAS, Tero. Texture Level of Detail Strategies for Real-Time Ray Tracing. *Apress* [online]. 2019 [cit. 2021-05-12]. Dostupné z: `https://media.contentapi.ea.com/content/dam/ea/seed/presentations/2019-ray-tracing-gems-chapter-20-akenine-moller-et-al.pdf#page=20&zoom=100,94,482`.

35. IGEHY, Homan. Tracing Ray Differentials. *SIGGRAPH '99 Proceedings* [online]. 1999 [cit. 2021-05-12]. Dostupné z: `https://graphics.stanford.edu/papers/trd/`.

36. Physically Based Rendering 3rd Edition: Sampling and Antialiasing. *Physically Based Rendering 3rd Edition* [online] [cit. 2021-05-11]. Dostupné z: https://www.pbr-book.org/3ed-2018/Texture/Sampling_and_Antialiasing.

37. LI, Yining Karl. Mipmapping with Bidirectional Techniques. *CODE & VISUALS* [online] [cit. 2021-04-22]. Dostupné z: https://blog.yiningkarlli.com/2018/10/bidirectional-mipmap.html.

38. PHORGAN1. *Mipmap illustration2* [online obrázek] [cit. 2021-04-04]. Dostupné z: https://commons.wikimedia.org/wiki/File:Mipmap_illustration2.png.

39. ARSENY KAPOULKINE. *pugixml*. 2021. Verze 1.11. Dostupné také z: https://pugixml.org/.

40. THE QT COMPANY. *Qt*. 2021. Verze 6.1. Dostupné také z: https://www.qt.io/.

41. KITWARE. *Cmake*. 2021. Verze 3.19.4. Dostupné také z: https://cmake.org/.

42. MICROSOFT. *Microsoft Visual Studio Community 2019*. 2021. Verze 16.9.1. Dostupné také z: https://visualstudio.microsoft.com/cs/.

43. Mitsuba 2: A Retargetable Forward and Inverse Renderer. *Mitsuba 2 Documentation* [online] [cit. 2020-12-27]. Dostupné z: https://mitsuba2.readthedocs.io/en/latest/index.html.

44. NICOLET, Baptiste. *Blender 2 Mitsuba* [https://github.com/mitsuba-renderer/mitsuba2-blender]. GitHub, 2021 [cit. 2021-03-22].

45. DUPUY, Jonathan. *BRDF plugin pro Mitsubu* [https://github.com/jdupuy/dj_brdf]. GitHub, 2018 [cit. 2021-04-22].

46. FOUNDATION, Open Source Vision. *OpenCV*. 2021. Verze 4.5.1. Dostupné také z: https://opencv.org/opencv-4-5-1/.

Příloha A

Seznam použitých zkratek

BIG Big image group

BSDF surface scattering model, přesněji distribuční funkce obousměrného rozptylu

BRDF funkce distribuce obousměrné odrazivosti (*Bidirectional Reflectance Distribution Function*)

BTF dvousměrná funkce textury (*Bidirectional Texture Function*)

GUI Grafické uživatelské rozhraní (*Graphical User Interface*)

MIF Multi-Image Format

MTL Material Library File

SVBRDF prostorově se měnící funkce distribuce obousměrné odrazivosti (*Spatially Varying Bidirectional Reflectance Distribution Function*)

ÚTIA AV ČR Ústav teorie informace a automatizace Akademie věd České republiky

Příloha B

Obsah přiloženého CD

- `ZadinaDP.pdf` ..... text diplomové práce
- `readme.txt` ..... stručný popis obsahu CD
- `bigConvert` ..... zdrojové kód BigConvertoru
  - `readme.md` ..... návod k používání konvertoru.
- `BIGpluginMitsuba` ..... obrázkové přílohy
  - `readme.md` ..... návod ke zprovoznění bigbtf pluginu.
  - `bigplugin` ..... složka se zdrojovými kódy bigbtf pluginu
  - `MIFlib` ..... MIF knihovna verze 0.4
  - `cubemaps` ... obsahuje obrázky nutné pro interpolaci UBO81x81 BTF reprezentace
- `mitsubaTestFiles` ..... obsahuje scény pro Mitsuby a výsledky renderů
  - `readme.txt` ..... přesný popis obsahu složky
  - `oldBigFiles` ..... textury ve starém BIG formátu
    - `readme.txt` ..... popis uložených textur
  - `mifFiles` ..... textury ve starém BIG formátu
    - `readme.txt` ..... popis uložených textur
- `PortablePrograms` ..... textury v MIF formátu
  - `BigConvert` ..... portable verze BigConvert programu
  - `Mitsuba` ..... portable verze Mitsuby s bigbtf pluginem
- `Testování` ..... obrázkové přílohy
  - `testovani_data.xlsx` ..... data naměřená během testování
  - `testovani_data.pdf` ..... data naměřená během testování
- `thesis` ..... zdrojová forma práce ve formátu LaTeX

Příloha C

Přílohy

```
<scene version="2.0.0">
    <integrator type="path">
        <!-- Instantiate a path tracer with a max. path length of 8 -->
        <integer name="max_depth" value="8"/>
    </integrator>
    <!-- Instantiate a perspective camera with 45 degrees field of view -->
    <sensor type="perspective">
        <!-- Rotate the camera around the Y axis by 180 degrees -->
        <transform name="to_world">
            <rotate y="1" angle="180"/>
        </transform>
        <float name="fov" value="45"/>
        <!-- Render with 32 samples per pixel using a basic
             independent sampling strategy -->
        <sampler type="independent">
            <integer name="sample_count" value="32"/>
        </sampler>
        <film type="hdrfilm"> <!-- Generate an EXR image at HD resolution -->
            <integer name="width" value="1920"/>
            <integer name="height" value="1080"/>
        </film>
    </sensor>
    <!-- Add a dragon mesh made of rough glass (stored as OBJ file) -->
    <shape type="obj">
        <string name="filename" value="dragon.obj"/>
        <bsdf type="roughdielectric">
            <!-- Tweak the roughness parameter of the material -->
            <float name="alpha" value="0.01"/>
        </bsdf>
    </shape>
</scene>
```

Kód 14: Ukázka podoby XML defince scény