# VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

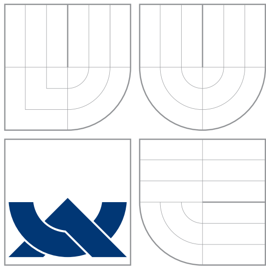

# VYTVÁŘENÍ SHADERŮ PRO SYSTÉM MENTAL RAY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

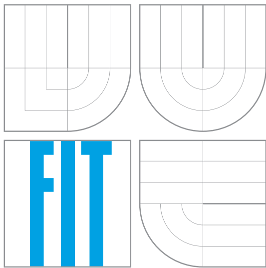

AUTOR PRÁCE — Bc. JAN DOHNAL
AUTHOR



BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

# VYTVÁŘENÍ SHADERŮ PRO SYSTÉM MENTAL RAY
MENTAL RAY SHADER WRITING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. JAN DOHNAL
AUTHOR

VEDOUCÍ PRÁCE ADAM HEROUT, Ph.D.
SUPERVISOR

BRNO 2007

## Abstrakt

Cílem diplomové práce je zmapovat vývoj počítačové grafiky v oblasti realistických zobrazovacích metod, seznámit se s renderovacím systémem mental ray, seznámit se s grafickým nástrojem Autodesk Maya, vytvořit několik shaderů pro mental ray a vytvořit návod, jak tyto shadery psát a jak je zprovoznit v programu Maya.

## Klíčová slova

vykreslování, sledování paprsku, mental ray, shader, Maya, BRDF, globální osvětlování

## Abstract

Goal of this diploma thesis is to get knowledge about history and evolution of computer graphic in area of realistic image synthesis, get knowledge about rendering system mental ray and about writing shader for it and write several shader. Create manual about writing shaders for mental ray. Get knowledge about program Maya and create a tutorial hot to get the shader into it.

## Keywords

rendering, ray tracing, mental ray, shader, Maya, BRDF, global illumination

## Citace

Jan Dohnal: Vytváření shaderů pro systém Mental Ray, diplomová práce, Brno, FIT VUT v Brně, 2007

# Vytváření shaderů pro systém Mental Ray

## Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana Adama Herouta, Ph.D.

........................
Jan Dohnal
14. května 2008

## Poděkování

Chtěl bych poděkovat svému vedoucímu za rady a podněty v průběhu práce na tomto textu.

© Jan Dohnal, 2007.
*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Obsah


**1 Úvod** **5**
1.1 Motivace . . . . . . 5

**2 Metody vykreslování scény** **6**
2.1 Metody založené na sledování paprsku . . . . . . 6
2.1.1 Dírková kamera . . . . . . 6
2.1.2 Vztah mezi paprskem a bodem v obraze . . . . . . 6
2.1.3 Dopředný ray tracing . . . . . . 7
2.1.4 Zpětný ray tracing . . . . . . 8
2.1.5 Stochastický ray tracing . . . . . . 9
2.1.6 Monte carlo . . . . . . 9
2.2 Ostatní metody . . . . . . 9
2.2.1 Objektové metody . . . . . . 9
2.2.2 Scanline . . . . . . 10
2.2.3 Reyes . . . . . . 10
2.2.4 Volume rendering . . . . . . 10

**3 Způsoby osvětlování a stínování** **12**
3.1 Základní principy . . . . . . 12
3.2 Modely . . . . . . 14
3.2.1 Phongův model a modely z něj odvozené . . . . . . 16
3.2.2 Torrance – Sparrow model . . . . . . 16
3.2.3 Oren – Nayar . . . . . . 17
3.2.4 Anizotropní modely . . . . . . 17
3.3 Globální osvětlování . . . . . . 18
3.3.1 Radiozita . . . . . . 18
3.3.2 Stochastic path tracing . . . . . . 19
3.3.3 Light tracing . . . . . . 21
3.3.4 Irradiance caching . . . . . . 21
3.3.5 Photon mapping . . . . . . 22
3.3.6 Final gathering . . . . . . 23

**4 Přehled současných používaných rendererů a jejich nasazení** **25**
4.1 RenderMan . . . . . . 26
4.2 Budoucnost rendererů . . . . . . 27

5 **mental ray** **28**
5.1 Co je to mental ray . . . 28
5.2 Historie mental ray a mental images . . . 28
5.3 Renderer . . . 29
5.3.1 Vlastnosti programu . . . 30
5.4 Scene description language . . . 31
5.5 Shading language . . . 31
5.5.1 Používání shading language pro psaní vlastních shaderů . . . 37
5.5.2 Příklady shaderů . . . 42
5.5.3 Ostatní shadery . . . 49
5.5.4 Zprovoznění vlastního shaderu v prostředí programu Autodesk Maya 51
5.5.5 Rychlost shaderu při různých operacích . . . 52

6 **Závěr** **57**

7 **Slovníček** **59**
Literatura 62

A **Obsah přiloženého DVD** **63**

B **Ukázky vytvořených shaderů** **64**

# Seznam obrázků


2.1 Příklad kamery ve 3d scéně. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2 Barevné spektrum a jeho vlnové délky. [11] . . . . . . . . . . . . . . . . . . . . 7

3.1 Sledování paprsků ve scéně a skládání výsledné barvy. . . . . . . . . . . . . 12
3.2 Ukázka složení izotropních reflexí jednotlivých soustředných kružnic do výsledného anizotropního jevu. [10] . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.3 Ukázka modelu Torrance - Sparrow. . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.4 Obrázek vyrenderovaný pomocí metody radiozity (3ds max). . . . . . . . . . 19
3.5 Obrázek demostrující výpočet osvětlení pomocí metody stochastic path tracing. 20
3.6 Ukázka různé úrovně rekurze,na obrázku vlevo je počet odrazů nastaven na dva, na obrázku vpravo na dvacet. . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.7 Obrázek vyrenderovaný metodou light tracing (3ds max). Kvalita obrázku je nižší, ale doba výpočtu je rozhodně nejvyšší ze všech použitých metod (50 minut). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.8 Ukázka photon mapy (vlevo) a caustic mapy (vpravo). . . . . . . . . . . . . 23
3.9 Obrázky vyrenderované pomocí photon mapping, nahoře bez final gathering, dole s final gathering. Na horním obrázku lze vidět artefakty po *photon mapě*, zejména v rohu místnosti. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

5.1 Arthur et les Minimoys, mentalimages.com. . . . . . . . . . . . . . . . . . . . . 29
5.2 Zpracování pole v systému mental ray. . . . . . . . . . . . . . . . . . . . . . . . . 32
5.3 Obrázek, který popisuje, kde a jak mental ray volá shadery a paprsky. . . . 34
5.4 Ukázka shaderu načteného pouze z deklaračního souboru .mi v prostředí 3ds max a Maya. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
5.5 Shader jacq_constant_simple. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.6 Shader tree materiálu. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
5.7 Obrázek shaderu advanced_glass. Referenční model z [9]. . . . . . . . . . . . 45
5.8 Ukázka různých hodnot koeficientů anizotropického materiálu. . . . . . . . 46
5.9 Obrázek shaderu anisoBrdf. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.10 Vlevo je snímek zachycující jev ve skutečném světě, vpravo výsledek shaderu. 47
5.11 Obrázky xray shaderu. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.12 Shader jacq_silk. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.13 Obrázky z průběhu vývoje prvního shaderu. . . . . . . . . . . . . . . . . . . . . 52
5.14 Vlevo je shader jacq_cd, vpravo jacq_snow. . . . . . . . . . . . . . . . . . . . . . 53
5.15 Výsledný shader s červenou barvou. . . . . . . . . . . . . . . . . . . . . . . . . . 54
5.16 Graf ukazující rozdíl v rychlosti načítání parametrů a konstant. Na ose X je počet vzorků, na ose Y čas v sekundách. . . . . . . . . . . . . . . . . . . . . . . . 55
5.17 Graf ukazující rozdíl v rychlosti trace_transparent a trace_refraction. Na ose X je počet vzorků, na ose Y čas v sekundách. . . . . . . . . . . . . . . . . . . . . 55

5.18 Graf ukazující rozdíl v rychlosti různých přístupů k proměnným. Na ose X je počet vzorků, na ose Y čas v sekundách. . . . . . . . . . . . . . . . . . . . 55
5.19 Graf ukazující nárust výpočetního času při zvyšujícím se počtu vzorků. Na ose X je počet vzorků, na ose Y čas v sekundách. . . . . . . . . . . . . . . . . 56

# Kapitola 1

# Úvod

## 1.1 Motivace

Cílem diplomové práce je zmapovat vývoj počítačové grafiky v oblasti realistických zobrazovacích metod, seznámit se s renderovacím systémem *mental ray*, seznámit se s grafickým nástrojem *Autodesk Maya* [2], vytvořit několik shaderů pro mental ray , vytvořit návod jak tyto shadery vytvářet a jak je zprovoznit v programu *Maya*.

Hlavním přínosem této práce je analýza systému *mental ray* a jeho popis pro případné zájemce, kteří by chtěli s tímto programem pracovat. V českém jazyce podobný text (alespoň pokud vím) neexistuje a v angličtině je pouze několik knih, věnující se systému mental ray. Nemyslím samozřejmě knihy pro grafiky, kde je popsána obsluha programu v prostředí některého z grafických nástrojů [19], ale opravdu publikace, která by *mental ray* rozebírala ve verzi *stand-alone* aplikace nebo více z té programátorské stránky. Kniha o psaní shaderů pro systém *mental ray*, pokud vím, existuje pouze jedna a ta je psána přímo vývojáři ze společnosti *mental images* [7].

Doufám, že tento text poslouží jako odrazový můstek pro všechny, kteří by se chtěli o tomto úžasném programu dozvědět něco víc a pochopit, jak vlastně pracuje a hlavně psát shadery. Protože pochopení základů je při řešení problémů podle mě mnohem důležitější, než několik knih o uživatelském nastavení a integraci v některém z grafických nástrojů.

Problém této oblasti počítačové grafiky je v tom, že je plná anglických výrazů, které často nejsou do češtiny přeloženy, nemají žádný vhodný ekvivalent a existují pouze ve složitých opisech. Jsem si této skutečnosti vědom a omlouvám se předem všem milovníkům českého jazyka za používání těchto anglických výrazů. Na konci je obsažen slovník důležitých pojmů, které se v tomto textu budu často vyskytovat, spolu s vysvětlením jejich významu. Kromě toho je i při prvním výskytu takového výrazu tento rozebrán, vysvětlen (buď přímo, nebo několika hesly v závorce) a dále už je používána i jeho cizojazyčná verze. Doufám, že čtenáři budou vesměs lidé, kteří se počítačové grafice věnují a znají tyto pojmy především anglicky, a proto jim to nebude tolik vadit, ostatním se ještě jednou omlouvám.

# Kapitola 2

# Metody vykreslování scény

Způsoby zobrazování a vizualizace scén lze rozdělit do tří hlavních skupin. Objektové metody, obrazové metody a komplexní metody. Jsou seřazeny podle náročnosti a rychlosti vykreslování od nejrychlejší po nejnáročnější, kvalitou pak od nejméně realistických po ty nejrealističtější.

V realistické počítačové grafice se používají komplexní vizualizační metody, které zpracovávají celou scénu, především metody založené na vysílání paprsků a sledování jejich trasy (z anglického ray tracing, tedy sledování paprsku). Nevýhodou těchto metod je jejich relativně velká náročnost (oproti některým jiným), i když v dnešní době přicházejí již téměř-realtime renderery (programy určené k vykreslování 3D scény), které jsou schopny podat realisticky vypadající obrázek během vteřiny a dynamicky ho zpřesňují, takže za pár vteřin je obrázek vykreslen.

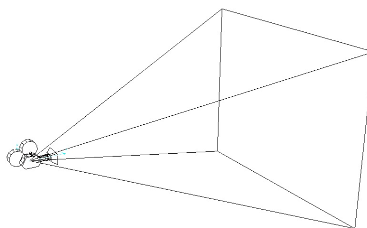
## 2.1 Metody založené na sledování paprsku

### 2.1.1 Dírková kamera

Metodu sledování paprsku si můžeme předvést na nejjednodušším fotoaparátu, takzvané dírkové kameře (kamera obscuro). Jedná se o krabici, ve které je na jedné z jejich stěn udělaná dírka a na protější stěně je umístěn film určený k expozici. Co by se stalo s filmem, kdybychom odstranili stěnu, ve které je dírka? Film by byl přeexponovaný a výsledkem by byla zcela bílá fotografie. Z pohledu ray tracingu by se stalo to, že do jednoho bodu filmu by přišlo příliš mnoho paprsků. Dírka tento problém řeší, protože umožňuje na film proniknutí jen omezeného množství paprsků. V ideálním případě by to byl jeden paprsek pro jeden bod na filmu. Kdybychom dírku v přední stěně zvětšovali, dostávali bychom obraz čím dál tím více jasnější a rozmazanější. V počítačové grafice tento základní model kamery většinou postačuje, pouze je trošku upraven. Místo dírky je umístěno oko pozorovatele a místo desky s filmem je rovina obrazu 2.1.

### 2.1.2 Vztah mezi paprskem a bodem v obraze

Když vytváříme obraz, většinou se zabýváme problémem, jakou barevnou hodnotu přiřadit konkrétnímu bodu obrazu, v oblasti počítačové grafiky tím většinou myslíme pixel. Představme si, že každý pixel je nezávislé okno, představující část scény. Jakou barvou máme obarvit celé okno, aby reprezentovalo vše, co lze skrze něj vidět? Odpověď můžeme najít v



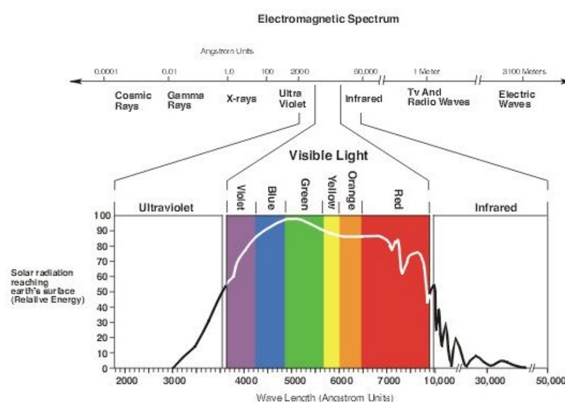
Obrázek 2.1: Příklad kamery ve 3d scéně.

každém díle 3D grafika. Nejjednodušším způsobem je zprůměrovat všechny barvy a vybrat výslednou.

### 2.1.3 Dopředný ray tracing

V minulé části jsme zjistili, že jedna ze základních věcí při vytváření obrazu je, jakou barvou obarvit jeden každý pixel obrazu, a že ji získáme tak, že zprůměrujeme barvy všech paprsků, které na daný pixel dopadají. Jak ale zjistíme, které paprsky to mají být? A jak získáme jejich barvu?

Světlo je reprezentováno fotony o různých vlnových délkách a různé energii. Při dopadu do našeho oka se pro náš mozek přemění v určitou barvu. Při odrazech od různých povrchů se energie fotonu mění, proto pro naše oko mají různé předměty různé barvy. Vlnová délka fotonu je v řádech stovek nanometrů, viditelné spektrum pro naše oči je zhruba 360-830 nm.




Obrázek 2.2: Barevné spektrum a jeho vlnové délky. [11]

Pokud do našeho oka dorazí zároveň foton červené a zelené barvy, uvidíme výslednou barvu žlutou. Fotony vycházejí ze zdroje světla. Zdroj světla bude v našem případě genero-

vat do všech směrů fotony všech viditelných vlnových délek. Mnoho fotonů poletí směrem od našeho oka, mnoho bude pohlceno při odrazech od různých materiálů, mnoho sice směrem k nám vyrazí, ale po cestě narazí na různé překážky a bude odraženo jiným směrem. Ale některé fotony dorazí až k našemu oku a zobrazí tak scénu, kterou vidíme. Podívejte se kolem sebe a najděte nějaký zdroj světla. Uvidíte, že některé paprsky se mohou odrážet od knih, stolu, židle, obrazů na stěnách, zrcadel, může se lámat přes nějakou sklenici, až dorazí k oku. Toto je sledování paprsku. Právě jste sledovali paprsek (foton) od jeho zdroje až do oka, tedy přesněji, použili jste metodu dopředného sledování paprsku (od zdroje k cíli). Jak jste si jistě všimli, je tato metoda velmi náročná, protože z obrovského množství paprsků, které ze zdroje světla vycházejí, jich jen malá část dorazí až k našemu oku. Zbylé paprsky bychom sledovali zcela zbytečně, a v mnoha případech se jedná o většinu paprsků. Metoda není nepřesná, pouze velmi pomalá. Prakticky se nepoužívá, ale slouží jako demonstrace principu sledování paprsku a vychází z ní další, už rychlejší metody.

### 2.1.4 Zpětný ray tracing

Zkusme proces obrátit. Které fotony ovlivňují výsledný obraz? Ty, které zasáhnou naši rovinu obrazu a dorazí do našeho oka. Pokud spojíme polopřímkou bod v obrazové rovině s naším okem, dostaneme dráhu fotonu, tedy paprsek. Ale foton mohl začít na libovolném bodu dané přímky. Ale od kterého objektu tedy přišel? Pokud budeme polopřímku sledovat, určitě najdeme průsečík s nějakým objektem. Je to možná dráha fotonu, protože pokud nějaký foton dorazil v daném bodě do našeho oka, musel tam dojít po této dráze. Otázkou nyní tedy je, zdali je to dráha fotonu. Sledujeme tedy fotony nikoliv od zdroje, ale od konce, směrem ke zdroji světla. Tento proces se nazývá zpětný ray tracing (backward ray tacing).

Mluvíme-li o sledování paprsku, máme na mysli většinou tuto metodu zpětného sledování paprsku, dopředný se kvůli jeho náročnosti nepoužívá.

Protože bude paprsků ve scéně i tak stále hodně, je třeba je roztřídit podle toho, k čemu se ve scéně používají.

**Paprsky obrazu** ty, které se promítají přes rovinu obrazu do kamery. V metodě backward ray tracing jsou to paprsky, které vychází z kamery a skrz rovinu obrazu se vrhají do scény.

**Světelné a stínové paprsky** ty, které nesou světelnou informaci od zdroje světla k objektu. Pomáhají určit barvu objektu na základě dopadajícího světla. Pokud na objekt dopadne obrazový paprsek, znamená to, že je pro kameru tento objekt viditelný. Z daného bodu (průsečíku objektu a dráhy paprsku) se vyšle stínový paprsek směrem ke světlu. Pokud paprsek dorazí až ke světlu, je nazván světelným a získána barva světelného zdroje. Pokud v cestě stojí neprůhledný objekt, paprsek zůstane stínovým a získá se barva stínu. Pokud narazí na lesklý nebo průhledný objekt, je potřeba vyslat z nového průsečíku další typy paprsků (viz níže).

**Odrazové paprsky** ty, které nesou světlo odražené od jiného objektu. Vznikají na lesklých objektech po odrazu světla. Pokud narazíme na lesklý objekt, je z průsečíku vyslán odrazový paprsek (klidně několikrát, pokud je ve scéně více podobných objektů) a na konci (pokud paprsek narazí na matný objekt) je získána barva objektu. Typickým materiálem je chrom.

**Transparentní paprsky** ty, které nesou světlo po průchodu objektem. Jsou podobné paprskům odrazovým, ale neodráží se od objektu ven, pokračují dál dovnitř objektu.

Typickým představitelem takového materiálu je sklo.

Opakovaným voláním těchto paprsků (rekurzí) lze projít postupně celou scénu a zobrazit výslednou barvu paprsku na konkrétním pixelu v obraze.

### 2.1.5 Stochastický ray tracing

I v syntéze obrazu se vyskytuje známý jev, zvaný aliasing [23]. Projevuje se především na hranách, které nejsou kolmé na osy obrazu, kde vytváří zoubky (*jittering* hran). Odstranění tohoto problému je velmi důležité a lze ho provádět například metodou *supersampling*. Tato metoda má však pravidelnou mřížku, která stále může zanechávat zubaté hrany. Stochastický ray tracing využívá nepravidelného rozmístění paprsků do této mřížky. Někdy se tato metoda rovněž nazývá distribuovaný ray tracing. Problémem v této metodě může být případ, kdy paprsky dorazí k nerovnému povrchu, upraveného například bump texturou. Kterým směrem paprsky odrazit? V případě této metody se vybere náhodně směr jednoho z paprsků a je považován za správný. V ideálním případě jde o to, vyslat co nejvíce paprsků směrem ke světlu, a co nejméně tam, kde světlo není. Protože v momentě vyslání paprsků výsledný směr ještě není znám, používá se toto náhodné rozložení a počítá se s tím, že alespoň některé paprsky dorazí do cíle. Tato metoda se rovněž využívá u technik jako motion blur (rozmazání pohybem) nebo depth of field (jev známý ve fotografii jako hloubka ostrosti) a měkkých stínů (takže od teď už prosím neříkejte, že pomocí ray tracingu nelze vytvářet měkké stíny). Problémem metody stochastický ray tracing je šum, který vzniká kvůli náhodnému rozložení paprsků. Zvýšením počtu paprsků samozřejmě zvyšujeme náročnost vykreslení scény.

### 2.1.6 Monte carlo

Tato metoda se používá pro výpočty simulací různých matematických modelů. Protože tyto rovnice mohou být dosti složité a výpočet klasickým deterministickým postupem by byl náročný, používá se odhad metodou *monte carlo*, která spočívá v náhodném (nebo pseudonáhodném) vzorkování daného prostoru a průměrování výpočtu (počet úspěšných a neúspěšných vzorků). Tato metoda se pro potřeby realistické vizualizace hodí, protože často potřebujeme počítat dvou nebo více rozměrné integrály a klasické algoritmy se na tyto výpočty nehodí. V obraze se mohou vyskytovat chyby (typicky šum), nicméně i přesto je tato metoda velmi účinná a často používaná, protože výpočet velmi urychluje. Existuje několik variant této metody (z nejpoužívanějších například *quasi-monte carlo*), různé metody vzorkování (*latin-cube sampling*), ale pro účely tohoto textu není potřeba tuto metodu hlouběji rozebírat.

## 2.2 Ostatní metody

Přestože drtivá většina používá metody se sledováním paprsku, ostatní metody mají v počítačové grafice také široké zastoupení, proto se o nich alespoň krátce zmíním.

### 2.2.1 Objektové metody

Do této kategorie patří například vykreslování pomocí knihoven OpenGL nebo Direct3D. Tyto metody jsou velmi rychlé, používají se v real-time vizualizacích. Stíny, odrazy a globální osvětlení se v nich sice dají počítat, ale nejsou na to primárně zaměřeny a většinou

se jedná o předpočítané textury a light mapy (textura obsahující předvypočítané informace o světle ve scéně), které se zobrazují nad základní texturou objektu. Požadavkem real-time vizualizací (například her) je hlavně rychlost, protože kamera se ve scéně pohybuje rychle. Proto ani není třeba počítat složité osvětlovací modely, protože si toho pozorovatel ani nevšimne.

### 2.2.2 Scanline

Metody zpracovávající scénu po jednotlivých obrazových bodech. Ještě stále poměrně populární metody, které vytvářejí realisticky vyhlížející obrázky. Ale například globální osvětlení se stejně předpočítává pomocí komplexních metod.

### 2.2.3 Reyes

Známý algoritmus pro výpočet scény, který byl vytvořen přímo ve filmovém průmyslu a je používán v systémech standardu RenderMan [16]. Algoritmus byl vyvinut v *Lucasfilm Computer Division* (později se z této divize vyvinulo studio *Pixar*) na začátku 80. let. Poprvé byl použit ve filmu *Star Trek 2: The Wrath of Khan* (rok 1982), poté prošel mnoha úpravami a optimalizacemi. Název Reyes pochází ze složeniny anglických slov *Renders Everything You Ever Saw* (vykreslí vše, co jste kdy viděli). Je určen pro vykreslování velkých a komplexních scén. Jeho princip je odlišný od metod založených na sledování paprsku, bohužel se mi nepodařilo dostat se k žádnému detailnějšímu popisu tohoto algoritmu. Činnost algoritmu lze popsat v šesti základních krocích.

**Bound** Vypočítat obálky kolem každého z objektů (těles) ve scéně.

**Split** Rozdělit velké objekty do menších, které budou vhodnější pro další dělení.

**Dice** Převést každý rozdělený objekt na mikropolygony, každý velikosti zhruba jeden pixel (ne více).

**Shade** Vypočítat osvětlení a stíny pro každý vertex objektu z mikropolygonů.

**Bust** Mikropolygonové sítě rozdělit na samostatné mikropolygony u kterých je individuálně určena jejich viditelnost.

**Hide** Vzorkovat mikropolygony a vytvořit výsledný 2D obrázek.

### 2.2.4 Volume rendering

Hlavním odvětvím využívající metody pro vykreslování objemu (volume rendering [18]) je asi lékařství. Počítačová grafika, jak se jí zabývám v tomto textu, využívá tyto metody málo. V oblasti medicíny se pomocí techniky volume rendering vizualizují data z různých scanerů (CT, PET), jako lidské tkáně, kosti, a podobně. Data jsou v tomto případě dosti komplexní, takže problémem je jak data vhodně vizualizovat. V oblasti „populární“ počítačové grafiky se pomocí volume renderingu vykreslují převážně věci jako mlha, mraky, efekt světla, které prosvětluje skrz okno nějakou místnost plnou prachu a podobně. Data můžou být podobná, ale obecně jsou spíše jednodušší, než medicínská data. Rovněž jejich zpracování je jednodušší.

Obecně je lze rozdělit do dvou hlavních skupin: data získaná z matematických modelů (například simulace kapalin, atmosféry) a data z scanerů.

Data z scanerů mohou mít podobu „řezů“, tedy sekvence scanů daného objektu v určitých konstantních rozestupech, jednotlivé obrázky mají stejné rozměry a stejné rozlišení. Jejich poskládáním ja sebe pak lze získat komplexní obrázek. Požadavkem je zobrazit tyto jednotlivé řezy dohromady jako jeden celek.

Při zobrazování objemových dat se využívá *voxelů*. Jsou to vlastně pixely v prostoru. Malé kvádry nebo krychličky. Nejednodušší metodou, prezentovanou v roce 1979 (Herman a Liu) je vykreslování všech voxelů (převedení všech pixelů v obraze daného řezu na voxely) s tím, že voxely mimo objekt jsou vykreslovány jako průhledné. Později byla tato metoda rozšířena o vytváření sítě polygonů z viditelných (hraničních) voxelů, čímž se snížila kostkovanost modelu.

Další metodou je tzv. 2 a 1/2 D technika, která je podobná metodě snímání ze scanerů. Scaner data snímá v řezech a tato metoda vlastně řezy vytváří a vkládá je za sebe v závislosti na pozici kamery. Pokud je síť řezů dostatečně hustá, může tato metoda podat kvalitní výstup, navíc je rychlá a dá se implementovat například v openGL. Tato metoda se často používá pro vizualizaci atmosféry, zejména mraků.

Patrně nejrozšířenější je metoda *marching cubes* (pochodující kostičky). Byla uvedena nezávisle na sobě dvěma týmy v letech 1986 a 1987 (Wavill a McPheeters byli první tým a Lorenson a Cline byli tým druhý). Každý tým zkoumal jiné odvětví, Wavill tento algoritmus použil na získání izoplochy definované analyticky pomocí několika řídících bodů, zatímco Lorenson ho použil na zobrazení medicínských dat. Algoritmus má dvě fáze. V první fázi vyplní uživatelem definovaný povrch voxely (podobně jako flood-fill algoritmus pro vyplňování 2D ploch) a označí hraniční voxely. Druhá část algoritmu zkontroluje krychle a vytvoří polygonovou síť, která je následně vykreslena. Algoritmus marching cubes tedy produkuje trojúhelníkovou síť. Kvalita sítě se odvíjí od kvality podkladových snímků.

# Kapitola 3

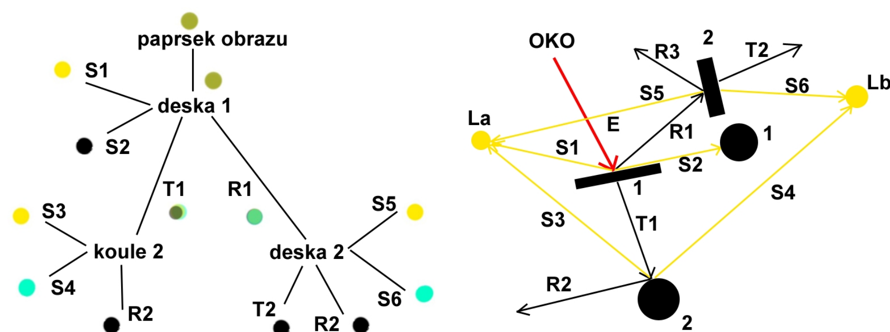
# Způsoby osvětlování a stínování

## 3.1 Základní principy

Vlastnosti povrchů lze popsat jak jednoduše, tak složitě. Čím podrobnější popis bude, tím složitější (a náročnější) bude výpočet a naopak. Reálný svět je velmi složitý a popsat nějaký povrch opravdu komplexně by bylo velmi náročné. Naštěstí lze ve většině případů použít i jednoduché popisy povrchů.

Při metodě sledování paprsku si budeme pořád dokola klást otázku: od kterého objektu daný paprsek přišel?

Na obrázku 3.1 je vidět paprsek E (paprsek obrazu), který směřuje od pozorovatele směrem k desce 1. Tato deska je zároveň reflexivní i průhledná. Nejprve je třeba vyslat stínové paprsky směrem ke světlům a zjistit tak, jestli je daný bod osvětlen. Paprsek ke světlu La dorazil bez kolize, ke světlo Lb se však nedostal, v cestě je koule 3. Proto bude počítána barva pouze světla La. Protože je povrch reflexivní i průhledný, je třeba vyslat i odrazový paprsek R1 a transparentní paprsek T1. R1 přichází od další desky 2 a T1 od koule 2. Od koule 2 má paprsek barvu na základě dalších stínových a reflexivních paprsků. Stínové paprsky S3 a S4 vyslané od koule 2 dopadají na světla La i Lb, odrazový paprsek R2 se odráží mimo scénu a bude mu přiřazena barva pozadí (okolí, prostředí). Odrazový paprsek R1 dopadá na desku, na které se opět odráží a láme. Paprsky R3 a T2 se odrážení mimo scénu. Stínové paprsky S5 a S6 dochází bez kolize k světlům La i Lb.




Obrázek 3.1: Sledování paprsků ve scéně a skládání výsledné barvy.

Výsledné barvy budou vypočítány na základě grafu (stromu) scény.

Popisujeme-li vzhled nějakého předmětu, obyčejně uvádíme i jeho barvu. Jakou bude

mít daný předmět barvu určuje foton, respektive jeho energie, který se od něj odrazil a dorazil do našeho oka. Nikdo nemůže určit, vnímají-li dva lidé tutéž barvu stejně. Můžeme však pomocí přístroje změřit její vlnovou délku a vzít ji jako referenci, na kterou se budeme odkazovat, budeme-li mít na mysli konkrétní barvu.

Foton má duální charakter – je možno na něj nahlížet jako na částici i jako na vlnění. Ray tracing vychází z modelu, kde paprsek je dráha určité částice, která nese barevnou informaci. Bylo by možné použít i vlnový charakter fotonu, ale popis by byl složitější a náročnější na výpočet. Nicméně ani jeden z modelů není úplný a přesný. Někdy je lepší a vhodnější nahlížet na foton jako na vlnění, jindy zase jako na částici.

Foton je nositelem světla. Má svou frekvenci, vlnovou délku a rychlost. Jeho rychlost se uvádí jako konstanta $3.10e^8 m.s^{-1}$. Vlnová délka určuje výslednou "barvu". Foton má rovněž energii, která se vypočte $E = f.h$, kde $h$ je Planckova konstanta $(6,63.10e^{-34} J.s)$.

K tomu, abychom dokázali vytvářet realistické obrazy, je třeba pochopit, jak se foton chová při kolizi s různými povrchy. Chování můžeme rozdělit do několika kategorií. Specular reflection, diffuse reflection, specular transmission, diffuse transmission. Tedy odrazy barevné složky a světelné složky a jejich pohlcení. Když foton narazí na předmět, změní směr a barvu, což má za následek tyto čtyři operace.

Když se podíváme na "bílou" žárovku, uvidíme "bílé" světlo. Jak vzniká bílé světlo, když v duze, viditelném barevném spektru, bílá není? Je to způsobeno aditivním skládáním barev – bílou tvoří mnoho fotonů různých barev, dopadajících do našeho oka zároveň.

Ačkoliv budeme mluvit o fotonu v jednotném čísle, bude zastupovat fotony dohromady dávající danou barvu.

Když foton dopadne na povrch tělesa, předá mu energii, kterou těleso buď pohltí, nebo odrazí ve formě fotonu.

Představme si foton denního světla $D6500$ a modrý stůl. Vibrující částice fotonu narazí do stolu, jehož atomy rovněž vibrují. Každá částice má určitou frekvenci, na které vibruje "nejlépe" (rezonanční frekvence). Pokud foton i atom mají optimální frekvence, atom pohltí foton a nabije se maximální energií, kterou převede na nový foton. Pokud nemají optimální frekvence, pohltí se energie jen částečně a uvolní se ve formě tepla. Objekt se nám zdá modrý proto, že stůl odráží jen fotony vlnové délky modrého světla, ostatní pohlcuje a přeměňuje na tepelnou energii.

Normála povrchu bude ve výpočtech důležitá veličina. Jedná se o vektor, který směřuje kolmo od povrchu v konkrétním bodě. Většinou je reprezentován jako vektor, který začíná na povrchu objektu a směřuje směrem "od" tělesa. Například normála roviny má ve všech bodech stejný směr, zatímco normála koule je průmětem polopřímky ze středu koule protínající daný bod na povrchu koule. Normálový vektor je většinou v normalizovaném tvaru.

Již výše byly zmíněny čtyři základní vlastnosti světla, které ovlivňují jeho chování. Dají se rozdělit na geometrické zásady a barevné zásady. Při výpočtu stínů je prohozen směr paprsků: paprsek směřuje ze světla na povrch, kde je (případně není) odražen do paprsků obrazu.

**Úplný spekulární odraz** Spekulární odraz je například zrcadlo, nebo odraz světla na povrchu lesklého předmětu. Jsou to ty paprsky, které splňují zákon odrazu.

**Úplný difúzní odraz** Difúzní odraz nastává například u matných povrchů. Difúzní odraz, na rozdíl od spekulárního, reaguje s povrchem objektu mnohem více. Protože je povrch tělesa matný (drsný), odráží se difúzní odraz všemi směry se stejnou intenzitou podle Lambertova reflexního modelu. Nemusíme se starat o žádné geometrické závislosti, světlo ze všech směrů se odrazí do paprsku scény.

**Úplný spekulární přenos** Přenesené světlo je takové, které projde objektem, například sklem. Na přechodu prostředí (vzduch/sklo) dochází k lomu paprsku. Každé prostředí má svůj index lomu, který určuje rychlost světla v daném prostředí v poměru k rychlosti světla ve vakuu.

**Úplný vnitřní odraz** Nastává, pokud paprsek dopadá na rozhraní mezi prostředími pod velmi malým úhlem a prostředí, ve kterém se nachází, má větší index lomu než druhé prostředí. Nenastane lom, ale paprsek se odrazí. Úhel pod kterým už dochází k odrazu se nazývá kritický úhel.

**Úplný difúzní přenos** Příkladem tělesa, na kterém vzniká difúzní přenos je průsvitný plast. Paprsky ze všech směrů se lámou do všech směrů se stejnou intenzitou.

## 3.2 Modely

Většina materiálů povrchů lze popsat některým z již existujících matematických modelů. Tyto modely jsou různě složité a různě realistické, podle toho jsou také náročné na výpočet.

Jednoduché modely, například konstantní stínování a Goraudovo stínování, nebo jednoduchý *Lambertův* model (pro zajímavost, tento model je starý více jak 200 let) jsou velmi rychlé (bývají implementovány na grafických kartách), ale pro dosažení realistických výsledků jsou nedostačující [22].

Lambertův model je základním modelem pro výpočet difúzní složky materiálu.

$$I_r = I_i \cdot \cos\theta_i$$

Základem realistických modelů jsou následující funkce:

Dvousměrová rozptylující funkce (*BRDF – bidirectional reflectance distribution function*)

Rozptylovací pravděpodobnostní funkce (*SPF – scatterign probability function*)

Z těchto funkcí se pak skládáním dají vytvářet další kombinované modely, ale základ mají v těchto dvou. Příkladem může být *BSSRDF* (*bidirectional sub-surface scattering reflection function* – dvousměrová podpovrchová rozptylující funkce), která popisuje složitou distribuci světla blízko pod povrchem objektu, například lidská kůže, vosk nebo mléko. Tyto funkce definují, kolik světla, které na povrch tělesa dopadá, se odrazí daným směrem. Základní rovnice *BRDF* funkce je

$$f\lambda(\theta_r, \phi_r, \theta_i, \phi_i) = \frac{I_{r\lambda}(\theta_r, \phi_r)}{E_{i\lambda}(\theta_i, \phi_i)} = \frac{I_{r\lambda}(\theta_r, \phi_r)}{I_{i\lambda} \cdot \cos\theta_i \cdot d\omega_i}[sr^{-1}]$$

kde

$\theta_i, \phi_i$ elevace a azimut dopadajícího paprsku
$\theta_r$, elevace a azimut odraženého paprsku
$I_{r\lambda}$ odražené zářivost (*radiance*)
$I_{i\lambda}$ ozáření, zářivost dopadající na plochu
$E_{i\lambda}$ intenzita ozáření (*irradiance*)
$\cos\theta_i$ cosinus úhlu mezi směrem dopadajícího paprsku a normálou povrchu
$d\omega_i$ prostorový úhel podél směru dopadajícího paprsku.

*BRDF* rovnice je závislá na pozici kamery, pozici světel ve scéně a normále povrchu v bodě, na který dopadá daný paprsek. Rovněž by měla být závislá na vlnové délce světla, protože různé vlnové délky se chovají jinak (například jev difrakce na skleněném hranolu).

Každá rovnice by měla splňovat 3 základní podmínky:

Zákon zachování energie – množství od povrchu odraženého světla musí být nejvýše stejné, jako množství světla dopadajícího na daný povrch.

$$_{\lambda}(\theta_r, \phi_r, \theta_i, \phi_i) \cdot \cos\theta_r \cdot d\omega_r \leq 1$$

Helmholtzova reciporocita – tento zákon říká, že v rovnici BRDF musí být zaměnitelné směry, odkud světlo přichází a kam je vyzářeno.

$$f_{\lambda}(\theta_r, \phi_r, \theta_i, \phi_i) = f_{\lambda}(\theta_i, \phi_i, \theta_r, \phi_r)$$

Funkce by měla být nezáporná pro všechny směry nad hemisférou

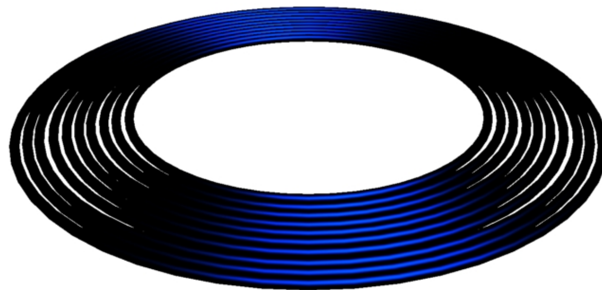
Modely, které tyto podmínky splňují, mohou být označeny za fyzikálně korektní. Například *Phongův* model fyzikálně korektní není, protože nesplňuje první podmínku (zákon zachování energie).

Modely můžeme rozdělit podle různých vlastností do několika kategorií, například podle typu povrchu, který popisují. Asi nejvhodnější je dělení na *izotropní* a *anizotropní* modely.

Izotropní povrch je takový, který můžeme libovolně otáčet kolem pomyslné normály a stále dostaneme stejný odraz. Například libovolný difúzní materiál, plast, nebo guma.

Anizotropní model odráží světlo různě, v závislosti na natočení povrchu kolem normály. Typickým představitelem anizotropního povrchu je broušený kov.

Anizotropní jev vzniká na klasických (izotropních) plochách, které mají velmi jemně upravený povrch. V případě broušeného kovu se bude jednat o stopy, které na povrchu zanechal brusný kotouč. Zvětšený zjednodušený obrázek (3.2) struktury tohoto povrchu vypadá asi takto 3.2.



Obrázek 3.2: Ukázka složení izotropních reflexí jednotlivých soustředných kružnic do výsledného anizotropního jevu. [10]

Jak je vidět, na kružnicích je aplikován izotropní materiál. Když však množství kružnic znásobíme tak, abychom dostali celý broušený povrch, získáme anizotropní model. Tento detailní pohled na to, jak vypadá a funguje anizotropní model je velmi důležitý při implementaci vlastních shaderů.

### 3.2.1 Phongův model a modely z něj odvozené

Jedná se o asi nejznámější model, který je empirický a izotropní. Není úplně realisitický, ale je jednoduchý a bývá často implementován na grafických kartách. Rovnice počítá útlum odraženého světla na základě závislosti mezi pozorovatelem a vektorem R.

Rovnice $cos^n\alpha$, kde $n$ je parametr *shinnest* (lesklost). Po přičtení difúzní a ambientní složky do rovnice dostaneme výslednou rovnici.

$$I_r = k_a I_a + I_i[k_d \cdot (\vec{n} \cdot \vec{L}) + k_s \cdot (\vec{V} \cdot \vec{R})^n]$$

**Schlickova aproximace:**

Schlick upravil výpočet modelu z $cos^n\alpha$ na

$$\frac{\cos\alpha}{n - n\cos\alpha + \cos\alpha}$$

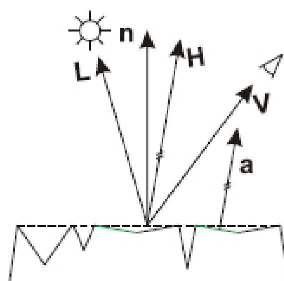
čímž výpočet urychlil. Model zůstal nadále izotropní a empirický a stále nesplňoval podmínky *BRDF*.

**Lewisův model:**

Upravil Phongův model tak, aby splňoval zákon zachování energie a Helzmanovu reciprocitu

### 3.2.2 Torrance – Sparrow model

Jejich model vychází z fyzikálních základů (ale není fyzikálně věrný). Jejich model předpokládá, že každý povrch je tvořen ze symetrických žlábků tvaru písmene „V“, které jsou ideálnmi zrcadly. Jejich orientace a tvar jsou dány normálním rozložením.




Obrázek 3.3: Ukázka modelu Torrance - Sparrow.

$$S = \frac{D \cdot G \cdot F}{\vec{n} \cdot \vec{V}}$$

**D** – k zrcadlové složce mohou přispívat jen plošky, které mají normálu orientovanou shodně s vektorem *H*. *D* je rozdělení počtu plošek orientovaných daným směrem.

**G** – *geometrical attenuation factor* – geometrický útlum. Popisuje útlum těch plošek, které stíní ostatním, představuje podíl světla, které zůstane po odstínění.

**F** – Fresnelův faktor určuje podíl světla, které se nepohltí, ale odrazí.

**Blinn:**

Upravil Torrance-Sparrow model, $D$ počítá jinak (elispoidy s excentricitou).

### 3.2.3 Oren – Nayar

Vytvářejí nový model, kde je povrch podobně jako u Torrance-Sparrow tvořen ploškami ve tvaru písmene „V“. Každá tato ploška je tvořena Lambertovským povrchem.

Výsledná *BRDF* funkce vypadá takto:

$$f(\theta_r, \theta_i, \phi_r - \phi_i, \sigma) = \frac{I_r(\theta_r, \theta_i, \phi_r - \phi_i, \sigma)}{E_0 \cdot \cos\theta_i}$$

### 3.2.4 Anizotropní modely

**Schlick**

Vychází z Torrance-Sparrow izotropního modelu, kde upravil složku $D$ tak, aby byla anizotropní. Snažil se vytvořit rychlý anizotropní model.

Výsledná *BRDF* funkce vypadá takto:

$$f(\delta, \theta_r, \theta_i, \phi) = \sum_{i=1}^{poč.vrstev} \frac{k_i}{4\pi(\vec{n} \cdot \vec{V})} D(\delta, \phi) G(\theta_i, \theta_r) F_\lambda(\vec{V} \cdot \vec{H})$$

**Ward**

Se rovněž snažil vytvořit jednoduchý model, který by měl anizotropické vlastnosti, byl empirický a fyzikálně věrohodný, a aby měl co nejméně parametrů. Použil Gaussovský model. Výsledná rovnice *BRDF* vypadá takto:

$$f(\theta_r, \theta_i, \phi_r, \phi_i) = \frac{k_d}{\pi} + \frac{k_s}{\sqrt{\cos_i \cos\theta_r}} \frac{e^{-tan^2\delta(\frac{\cos^2\phi}{a_x^2} + \frac{\sin^2\phi}{a_y^2})}}{4\pi \cdot a_x \cdot a_y}$$

kde:
$k_d$ je koeficient odrazu difúzní složky
$k_s$ je koeficient odrazu zrcadlové složky
$a_x$ je směrodatná odchylka rozdělení ve směru $\vec{x}$
$a_y$ je směrodatná odchylka rozdělení ve směru $\vec{y}$
$\delta$ je úhel mezi vektory $\vec{H}$ a $\vec{n}$ $\phi$ je azimut vektoru $\vec{H}$

**Ashikhmin, Shirley**

Vytvořili anizotropní Phongův model, empirický, splňuje podmínky *BRDF* a je jednoduchý, s jednoduchým nastavováním parametrů. Skládá se podobně jako Phongův model, ze zrcadlové složky a difúzní složky. Jejich rovnice jsou zde:

$$\rho_s(k_1, k_2) = \frac{\sqrt{(n_u+1)(n_v+1)}}{8\pi} \frac{(n \cdot h)^{\frac{n_u(hu)^2 + n_v(hv)^2}{1-(hn)^2}}}{(k \cdot k) max((n \cdot k_1), (n \cdot k_2))} F(k \cdot h)$$

$$\rho_d(k_1, k_2) = \frac{28R_d}{23\pi}(1 - R_s)(1 - (1 - \frac{n \cdot k_1}{2})^5)(1 - (1 - \frac{n \cdot k_2}{2})^5)$$

## 3.3 Globální osvětlování

Metody sledování paprsku nám dávají matematicky přesný popis scény, dokonalé odrazy a ostré stíny, ale tyto obrázky se neshodují s reálným světem. Ray tracing umí dobře a rychle zpracovávat a vypočítávat spekulární složku světla (která vlastně v reálu neexistuje, pouze realitu napodobuje), ale interakce difúzní složky se pomocí ray tracingu vypočítává obtížně.

### 3.3.1 Radiozita

*Radiozita* je velmi efektivní metoda pro výpočet právě difúzní interakce objektů ve scéně. Metoda počítá difúzní odrazy v uzavřeném prostředí, tedy je ideální pro interiérové scény, kde se paprsek může odrážet. Je komplexní, počítá se pro celou scénu a není závislá na pozici kamery (*view – independent*). Toto je výhoda při architektonických vizualizacích jako třeba průlet budovou nebo jiné sekvence pohledů, radiozita se vypočte jednou a při ostatních snímcích se již počítat nemusí. Radiozita vytváří efekt zvaný color bleeding, kde se objekty různých barev vzájemně barevně ovlivňují. Otázkou může být, jestli je tento efekt žádoucí, nebo ne. V některých situacích můžeme mít potřebu tento efekt potlačovat. Když položíte červenou krabičku na bílou čtvrtku papíru, dostane papír difúzní odrazy od červené kostičky a bude v okolí, kde je kostka položena, lehce načervenalý, a to i v reálném světě. Poprvé byla radiozita představena v roce 1984 na univerzitě v USA.

Výpočet metody je založen na teorii přenosu tepla mezi dvěma předměty. Povrchy pro výpočet radiozity mají dokonalý povrch (Lambertův model), takže jsou dokonalými difuzéry, reflektory, emitory, u kterých se světlo odráží do všech směrů se stejnou intenzitou. Metoda pracuje tak, že dělí jednotlivé plochy na menší dílky, na kterých je radiozita konstantní. Radiozita jednoho takového dílku je součet vyzářené a odražené energie. Energie vyměněná mezi dvěmi plochami je geometrická závislost na vzdálenosti a jejich vzájemné orientaci. Mezi dvěmi rovnoběžnými plochami, které jsou blízko sebe, bude vyměněná energie velká.

Radiozita × plocha = vyzářená energie + odražená energie, kde odražená energie = koeficient × energie dopadající na dílek ze všech ostatních dílků. Soustava rovnic se dá zapsat maticově a řešit maticově, nebo lze výpočet provádět iteračně.

$$B_i = \frac{1}{A_i} \int_{S^i} \int_{\Omega_x} L(x \rightarrow \Theta) \cos(\Theta, N_x) d\omega_\Theta dA_x$$

Klasické renderery potřebují znát hodnoty radiozity nebo intenzity pro vertexy. Hodnota pro vertex je průměrem hodnot okolních dílků, které daný vertex obklopují. Další vertexy (například na hranách dílků) lze vypočítat interpolací mezi koncovými vertexy.

Konfigurační faktor je hodnota každého dílku udávající, jak ovlivňuje všechny ostatní dílky ve scéně. V původní verzi prezentované v roce 1984 byla scéna konvexní, což znamená, že všechny plošky na sebe navzájem „viděly“. Ve většině scén tomu tak však není a konfigurační faktory se musí počítat. Výpočet těchto faktorů zabírá při řešení radiozity nejvíce času, takže velmi záleží na zvolených metodách a postupech. Dalším důležitým výpočtem je dělení ploch. Protože některé plochy, kde dopadá světlo (energie) přímo a které nemají v okolí žádné objekty, se kterými by si energii vyměňovaly, nepotřebují mít hustou síť dílků, zatímco objekty, které jsou osvětlovány nepřímo, jsou menší a v jejich okolí je s čím reagovat, potřebují síť hustší.

Obrázek 3.4: Obrázek vyrenderovaný pomocí metody radiozity (3ds max).

### 3.3.2 Stochastic path tracing

Algoritmy založené na sledování dráhy paprsků vycházejí z metod sledování paprsku a jsou jim velmi podobné. Výraz ray tracing a výraz *path tracing* jsou si velmi blízké, výraz path tracing se používá na označení algoritmů, ve kterých se paprsky při dopadu na povrch tělesa nedělí na několik dalších.

Pro výpočet globálního osvětlení scény pro konkrétní obrázek znamená výpočet hodnoty *radiance* (záření) pro každý bod výsledného obrazu. Nejlépe je vyjádřit tento výpočet takto:

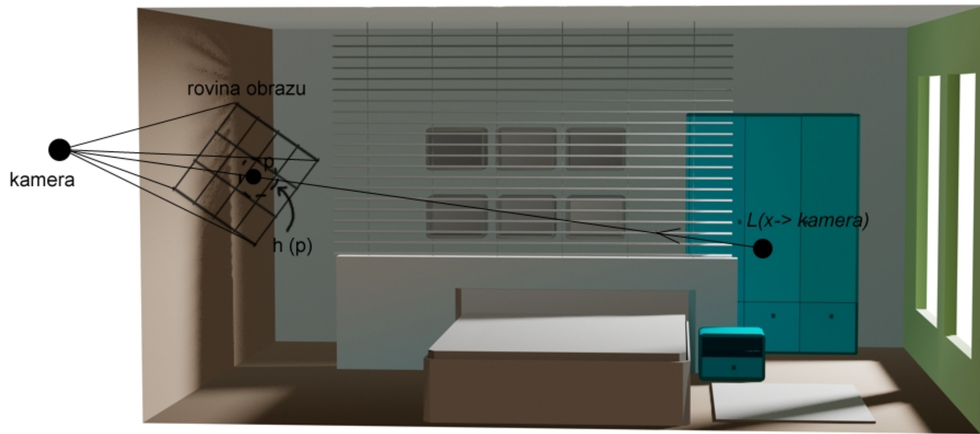
$$L_{pixel} = \int_{imageplane} L(p \rightarrow eye)h(p)dp$$
$$= \int_{imageplane} L(x \rightarrow eye)h(p)dp$$

Na výpočet integrálu lze použít metody *Monte carlo*. Nejednodušší algoritmus výpočtu radiance je pomocí klasické metody stochastický ray tracing. Předpokládejme, že chceme vypočítat radianci $L(x \rightarrow o)$ v bodě $x$.

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + L_r(x \rightarrow \Theta)$$
$$= L_e(x \rightarrow \Theta) + \int_{\Omega x} L(x \leftarrow \Psi) f_r(x, \Theta \leftrightarrow \Psi) \cos(\Psi, N_x) d\omega_\Psi$$

Na integrál aplikujeme výše zmiňovanou metodu monte carlo integrace tak, že vygenerujeme několik N náhodných směrů $\Psi_i$ na polokouli $\Omega_x$, distribuovaných náhodným rozložením $p(\Psi)$. Rovnice nyní bude vypadat takto:

$$\langle L_r(x \rightarrow \Theta) \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{L(x \leftarrow \Psi_i) f_r(x, \Theta \leftrightarrow \Psi_i) \cos(\Psi_i, N_x)}{p(\Psi_i)}$$




Obrázek 3.5: Obrázek demostrující výpočet osvětlení pomocí metody stochastic path tracing.

Cosinus a *BRDF* lze vypočítat pomocí hodnot dalšího bodu ve scéně. Hodnotu tohoto bodu dostaneme rekurzivně takto:

$$L(x \leftarrow \Psi_i) = L(r(x, \Psi_i) \rightarrow -\Psi_i)$$

Aby tento rekurzivní algoritmus vypočítal hodnotu vyzařování, je třeba, aby na konci trasy dorazil k povrchu s nenulovou intenzitou vyzařování, jinými slovy musí dorazit k některému ze zdrojů světla ve scéně. Jelikož jsou světelné zdroje v porovnání s ostatními objekty ve scéně velmi malé, bude i paprsků, které mají nějakou hodnotu záření, velmi málo. Jen pokud paprsek dorazí ke světelnému zdroji, bude konkrétnímu bodu v obraze přiřazena vypočítaná barva, takže jinak bude obraz především černý.

### Ruská ruleta

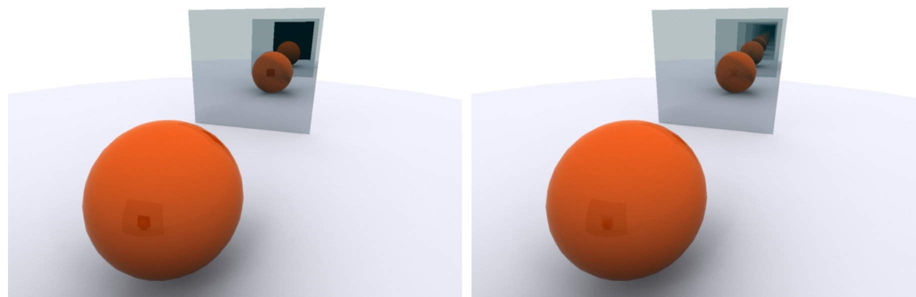
Tato metoda se používá na ukončení rekurze [20]. Protože by se paprsky ve scéně mohly odrážet do nekonečna, je potřeba je někdy zastavit. Rekurzi je možno ukončit více způsoby. Klasickou metodou je nastavení pevného limitu, po kterém se všechny paprsky (pokud nedorazí dříve ke zdroji světla nebo cíli) ukončí. Tato metoda je jednoduchá, ale nezaručuje, že se paprsek dostane na potřebné místo. Výhodou je, že je tento limit většinou uživatelsky nastavitelný. Scéna, ve které jsou jen objekty s difúzním povrchem, nepotřebuje více než 2 nebo 3 úrovně rekurze, zatímco scéna zaplněná velmi reflexními objekty může mít úroveň rekurze i několik desítek. Jiným limitem může být součet délek paprsků. Pokud bude součet větší než určitá hodnota, rekurze se zastaví. Ruská ruleta umožňuje vysílat paprsky s libovolnou hloubkou rekurze.

$$I = \int_0^1 f(x)dx$$

Pokud použijeme monte carlo integraci, můžeme integrál upravit takto:

$$\langle I_{RR} \rangle = \begin{cases} \frac{1}{P} f(\frac{x}{P}) & \text{když } x \leq P \\ 0 & \text{když } x > P. \end{cases}$$

Aplikací metody ruské rulety určíme zastavení rekurze s pravděpodobností $\alpha = 1{-}P$. Alfa je absorpční pravděpodobnostní koeficient. Pokud je hodnota $\alpha$ malá, bude rekurze pravděpodobně pokračovat dlouho, zatímco s velkou hodnotou se zastaví rychleji, ale výsledek bude nepřesnější (což se většinou promítne v podobě šumu v obraze). V globálním osvětlení je koeficient většinou $1{-}\alpha$ rovna hodnotě *hemisphere reflectance* materiálu povrchu. Takže tmavé části budou ukončeny dříve, zatímco osvětlené plochy a světlé části budou počítány déle. Toto chování je podobné chování světla v reálném světě.



Obrázek 3.6: Ukázka různé úrovně rekurze,na obrázku vlevo je počet odrazů nastaven na dva, na obrázku vpravo na dvacet.

### 3.3.3 Light tracing

Tento algoritmus je integrován v 3ds max rendereru. Trasování paprsku od kamery směrem ke zdroji světla je nepřirozené, protože v reálu se světlo pohybuje opačným směrem. Algoritmus *light tracing* postupuje směrem od zdrojů světla ke kameře. Pro každý paprsek vyslaný ze světelného zdroje je po dopadu na nějaký objekt vypočítána jeho „důležitost" pro konkrétní bod v obraze a samozřejmě také jeho světelná hodnota. Pro oba výpočty je použita monte carlo integrace.

### 3.3.4 Irradiance caching

Tuto metodu (podobnou) používá například program V-Ray. Jedná se o techniku, která urychluje metody monte carlo pro výpočet nepřímého osvětlení (*indirect illumination*) na objektech s materiály difúzní povahy. Protože klasický výpočet může potřebovat vysílat stovky paprsků z jednoho bodu, může se výsledný výpočet značně zpomalit.

Metoda *irradiance caching* využívá faktu, že odražené světlo na difúzním povrchu (ačkoliv jsou jeho výpočty náročné) má většinou podobný, jemně vyhlazený charakter. Takže tato technika ukládá hodnoty záření do datové struktury a pak hodnoty sousedních ploch interpoluje z hodnot již uložených. Interpolace je řízena různými funkcemi, například gradienty, nebo výpočtem chyby. Data jsou uložena v *octree* struktuře. Pokud je třeba vypočítat hodnotu záření v konkrétním bodě, je nejprve zkontrolována struktura a nalezeny nejbližší vzorky, které mají dostatečně přesné hodnoty na to, aby z nich byla vypočítána nová hodnota záření. Okolní hodnoty jsou většinou prohledávány v rámci určitého poloměru, kterým se určuje poměr mezi rychlostí a kvalitou výsledného obrazu. Větší poloměr znamená rychlý výpočet, velké interpolace a tedy nepřesný výsledek.

Obrázek 3.7: Obrázek vyrenderovaný metodou light tracing (3ds max). Kvalita obrázku je nižší, ale doba výpočtu je rozhodně nejvyšší ze všech použitých metod (50 minut).

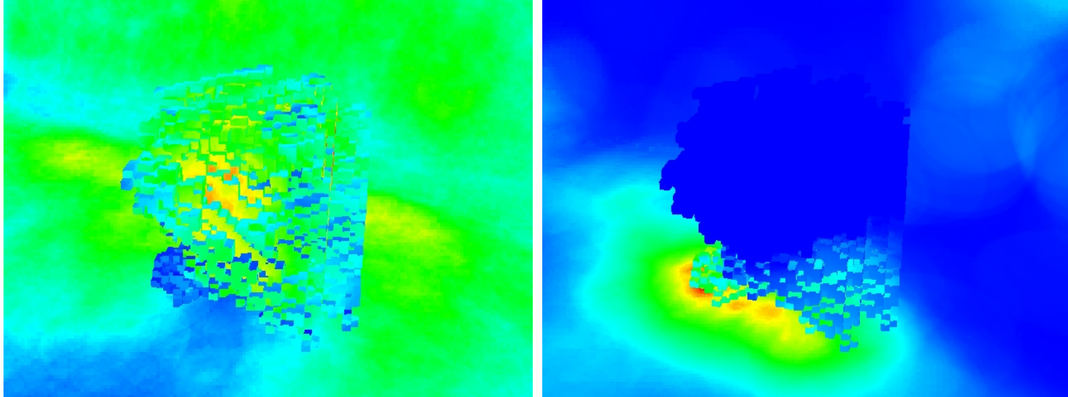
### 3.3.5 Photon mapping

Tento algoritmus používá systém mental ray. Jedná se o robustní algoritmus výpočtu globálního osvětlení, který je podobný metodě dvousměrného sledování paprsku (bidirectional path tracing). Na rozdíl od zmiňované metody *bidirectional path tracing* však používá již vypočítané hodnoty, takže je rychlejší.

Výpočet probíhá ve dvou průchodech. V prvním průchodu vyšle paprsky (fotony) ze zdrojů světla směrem do scény. Tyto fotony nesou informaci o intenzitě světla a jsou uloženy v datové struktuře, která se nazývá mapa fotonů (*photon map*). Ve druhém průchodu je pak vykreslen výsledný obraz scény pomocí dat uložených v photon mapě. Protože ukládá data do photon mapy zvlášť od definice povrchů, může být použita i na procedurální geometrii, čímž se stává univerzálnějším algoritmem. Jednotlivé druhy fotonů mohou být uloženy do několika map, takže budou jednotlivé mapy obsahovat jen konkrétní typy pro konkrétní účel. Příkladem specializované photon mapy je třeba *caustic map* (mapa kaustiky), která ukládá informace o interakci fotonů při průchodu tělesy (typicky skleněnými).

Vypočítat a zobrazit kaustické jevy například pomocí metod s monte carlo integrací by bylo velmi zdlouhavé a na výpočet by bylo potřeba velké množství vzorků. Tím, že se pro tuto mapu hledají a ukládají jen konkrétní typy fotonů, dosáhne přesného výsledku rychleji. Metoda obsahuje chyby (odchylka od předpokládané hodnoty a od skutečně vypočítané hodnoty). Protože se však používá pouze pro výpočet globálního osvětlení, většinou stačí zvýšit počet fotonů k odstranění většiny artefaktů.

První průchod je klíčový pro efektivitu a rychlost vykreslení. V tomto průchodu jsou fotony vyzářeny ze světelných zdrojů a sledovány při průchodu scénou, kde jsou odráženy, lámány, procházejí objekty a jsou pohlcovány. Při tomto průchodu jsou používány metody monte carlo integrace i ruské rulety. Pokud dorazí foton na difúzní povrch, je uložen do



Obrázek 3.8: Ukázka photon mapy (vlevo) a caustic mapy (vpravo).

photon mapy. Tato je reprezentována vyváženým kd-stromem. Příklad výpočtu globálního osvětlení pomocí photon mapping techniky vypadá například takto:

scéna obsahuje objekty, které vytváří kaustické jevy, proto bude algoritmus vytvářet dvě photon mapy. Do první bude ukládat základní data pro globální osvětlení, druhou bude používat na uložení hodnot o kaustice. Protože se kaustika projevuje jako „zaostření“ paprsků, není potřeba generovat velké množství fotonů na dosažení potřebného a kvalitního výsledku. Navíc jsou fotony vysílány jen směrem, kde se nacházejí objekty, které kaustický jev vytvářejí.

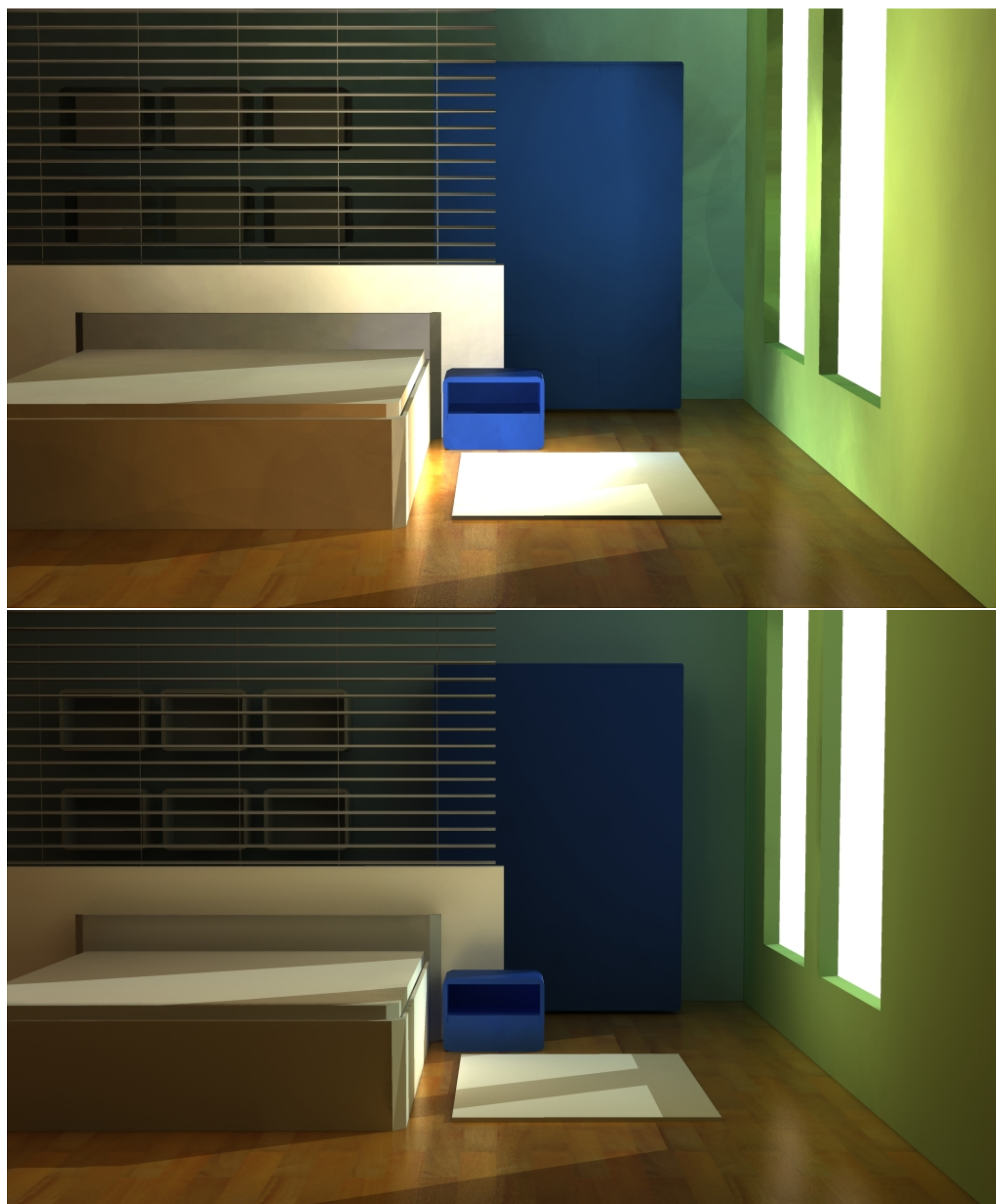
### 3.3.6 Final gathering

Tuto metodu používá systém mental ray. Jedná se o metodu, která pouze dopočítává a zpřesňuje již vypočítané osvětlení (například pomocí metody radiozita). Již z názvu metody lze vyčíst, že se jedná o konečné sesbírání nějakých hodnot. Protože je výsledek radiozity nebo jiné metody globálního osvětlení dost hrubý a obsahuje artefakty, bývá na výsledek použito například Goraudovo stínování, které výsledek vyhladí. Bohužel tímto postupem může obraz ztratit důležitý detail. Metoda *final gathering* je aplikována při vykreslování výsledného obrazu, osvětlení počítá pro výsledn obraz per-pixel a pomocí metod sledování paprsku.

$$L_{pixel} = \int_{rovinaobrazu} L(p \rightarrow eye)h(p)dp$$

Protože i v final gathering metodě je v rovnici integrál, je možno využít monte carlo integraci. Hlavní rozdíl mezi metodou final gathering a mezi klasickým stochastickým sledováním paprsku je v tom, že final gathering nepočítá rekurzivně žádné distribuce záření, protože tyto hodnoty jsou již vypočítány z předchozího řešení (například radiozity).

Protože by metoda při klasickém počítání připomínala stochastický ray tracing, tedy obraz by byl stále hodně zašuměný, protože jen málo paprsků by dorazilo ke zdroji světla, je algoritmus rozdělen do dvou částí – počítání přímé a nepřímé složky osvětlení. Tento způsob je rychlejší, protože se počítá pouze složka přímého osvětlení (*direct illumination*), zatímco nepřímé osvětlení se získává interpolací z již vypočítaného řešení radiozity nebo podobné metody.



Obrázek 3.9: Obrázky vyrenderované pomocí photon mapping, nahoře bez final gathering, dole s final gathering. Na horním obrázku lze vidět artefakty po *photon mapě*, zejména v rohu místnosti.

# Kapitola 4

# Přehled současných používaných rendererů a jejich nasazení

Programy lze rozdělit podle mnoha kritérií a různých parametrů, pro tento text bylo zvoleno rozdělení na komerční a nekomerční programy.

Mezi nekomerční programy patří například POV-Ray, Yaf-ray, Radiance, Sunflow a jistě mnoho dalších. Sunflow a Yaf-ray patří mezi moderní renderery, podporující všechny dnes populární funkce, jako globální osvětlení, osvětlování pomocí *HDR* map, měkké stíny a podobně.

POV-Ray je poměrně starý program a setkal se s ním pravděpodobně každý, kdo se o metody vykreslování alespoň trochu zajímal. První verze na Amize uměla vykreslovat kouli na podložce a ještě jen černobíle, dnes je to velmi schopný renderer, který rovněž podporuje globální osvětlování a měkké stíny. Má jednoduchý jazyk na popis scény (scene description language) ve formátu ascii a podporuje skriptování, takže je jednoduché v něm napsat i funkce na generování například fraktálů.

Komerčních programů je celá řada. Každý modelovací nástroj většinou má nějaký vlastní renderer a často také podporuje ostatní programy formou zásuvných modulů (plug-in modulů). Z těch nejpoužívanějších bych jmenoval mental ray, V-ray, final render, Maxwell render, Brazil r/s a celá řada dalších. Podle průzkumu serveru *3dstudio.cz* jsou nejpopulárnější renderery v tomto pořadí:

- V-ray 38 %
- mental ray 21 %
- Final Render 13 %
- zbytek tvoří ostatní renderery

V-ray je vyvíjen bulharskou společností *Chaosgroup*, nemá otevřený SDK, takže pro něj nelze psát žádná rozšíření a je třeba vždy čekat do další verze programu. Je to velmi rychlý systém založený na metodě sledování paprsku s monte carlo integrací. V nedávné době byla uvedena verze s *real-time* náhledem, kde se obrázek vykreslí během vteřiny a během několika dalších se postupně zpřesňuje. Často je využíván v architektonických vizualizacích a ve vizualizacích anorganických objektů (auta, elektronika a podobně), proto se zaměřuje především na tuto kategorii. Produkuje obrázky fotorealistické kvality.

Final render je vyvíjen v Německu společností *Cebas.* Jeho zaměření je podobné jako u V-ray, rovněž nemá otevřený SDK, a není proto možno vyvíjet vlastní rozšíření. Využívá se rovněž především pro technické vizualizace.

## 4.1 RenderMan

Zvláštní skupinu pak tvoří renderery založené podle standardu specifikace *RenderMan.* Tuto specifikaci vytvořila společnost Pixar v roce 1988. Jelikož je systém mental ray v mnohém podobný této specifikaci, chtěl bych ji trochu přiblížit:

Renderman se skládá ze tří hlavních částí. Samotného vykreslovacího programu s rozhraním pro uživatele, *shading language* (jazykem na psaní shaderů) a *scene description language* (jazykem pro popis scény, zkráceně SDL).

*RenderMan Interface Specification* definuje rozhraní pro komunikaci mezi modelovacím a renderovacím nástrojem. Nespecifikuje žádné předepsané algoritmy které má program umět, pouze definuje kvalitu výstupu. Render systém podle této specifikace musí umět:

- Simulovat reálnou kameru se všemi jejími vlastnostmi, motion blur, depth of field.
- Načítat všechna geometrická primitiva, křivkové plochy, patches, kvadriky, a to vše v velkém množství (miliony).
- Simulovat vlastnosti povrchů, volumetrické efekty, textury pro definování vlastností povrchů, jako bump, displace a environment mapy.
- Pracovat nezávisle na hardware a ostatních programech v render-pipeline.
- Rozhraní programu musí být kompletní, ale zároveň kompaktní a co nejefektivnější v komunikaci s vlastním renderovacím systémem. Není definován žádný „komfort“ v podobě 3d pohledů a náhledů na scénu. Požadavkem je:
  - Plátno s možností vytvářet grafická primitiva s textovými popisky a provazovat je pomocí úseček či křivek (vytvářet graf scény, nebo strom scény).
  - Vykreslovat neplošná primitiva (3d čáry a křivky).
  - UI prostředí s obsluhou vstupních zařízení.
- Standard nezahrnuje a nedefinuje žádné grafické formáty.

**Shading language**

S rozvojem rendererů přišla potřeba dát člověku možnost rozšířit a upravit vlastnosti programu, jak jej navrhnul jeho tvůrce. Pro jednoduché renderery, které podporují například jen *Goraudovo* a *Phongovo* stínování a osvětlování to samozřejmě nemá cenu, ale pro komplexnější renderery, které podporují mnoho různých modelů, už je potřeba použít Shading language. Na tuto myšlenku přišel v roce 1984 *Cook* a od něj ji převzalo a rozšířilo studio *Pixar* v roce 1988.

> Jakmile existuje program na syntézu obrazu, uživatel bude chtít vidět i shadery. Lidé jsou velmi citliví na propracovanost shaderu a tuto citlivost vyjadřují téměř nekonečnou touhou mít perfektní kontrolu nad shaderem. Čím je program kvalitnější, tím větší nároky na něj jsou kladeny, což se promítá do nároků na kvalitu shaderů.

> Jako alternativu k vytváření těchto shaderů lze dát uživateli možnost zasáhnout do samotných shaderů než jen do nastavení několika parametrů. Ideální řešení je dát uživateli přístup k užitečným a důležitým funkcím systému bez nutnosti obtěžováním ho s technickou stránkou systému. Psaním shaderu v prostředí shading language může programátor rozšiřovat staré shading modely a definovat nové, může definovat světelné zdroje s libovolnou distribucí světla, může vlastnosti povrchů pomocí různých map.
>
> Upstill [18]

*Cook* rozděluje shader na jednotlivé elementární bloky, které se počítají nezávisle na sobě a jsou postupně montovány dohromady, čímž se tvoří výsledný shader. Tento postup vytváří strom, tedy shader tree. Podobný přístup je vytváření sítě (shader network). Opět se jedná o dělení shaderu na základní bloky, které se postupně spojují dohromady podle určitých pravidel. I textury lze interpretovat pomocí těchto sítí nebo stromů.

Jazyk umožňuje psát procedury zvané shadery. Tyto jsou rozděleny podle typu do několika kategorií.

Zdroje světla – vypočítávají barvu světla, mají pozici, berou pozici bodu, na které má světlo dopadat a vrací barvu dopadajícího paprsku. Volitelnými vlastnostmi bývá vlnová délka, intenzita, směr světla a útlum světla.

Povrchové shadery – implementují lokální odrazový model a počítají množství odraženého světla.

Objemové shadery – implementují efekty světla, které se šíří prostředím o nějakém objemu.

Displacement shadery – upravují geometrii modelu.

Transformation a imager shadery. Transformation upravují transformace objektů a imager jsou výstupní shadery aplikované na výsledný 2d obraz.

## 4.2 Budoucnost rendererů

Mnoho lidí se domnívá, že realistické renderery používají ke svým výpočtům grafickou kartu, stejně jako například *opengl* nebo *direct3d*. Využití těchto karet, které mají na potřebné operace speciální instrukce, se přímo nabízí, ale bohužel to není jednoduché kvůli vzájemné nekompatibilitě mezi různými výrobci [6]. Takže téměř všechny realistické renderery se musí ve své práci spokojit s výkonem poskytnutým procesorem.

Výjimku tvoří asi jen program *gelato*, který vyvíjí společnost *nVidia*. Program podporuje (překvapivě) jen grafické karty tohoto výrobce a to jen řadu profesionálních karet *quadro*. U ostatních řad není plná podpora všech funkcí. Takže opravdu asi není lehké využívat potenciál grafických karet, když i sám výrobce podporuje jen některé své karty.

# Kapitola 5

# mental ray

## 5.1 Co je to mental ray

Mental ray je renderovací systém vyvíjený společností mental images od roku 1986 a je určen pro produkci vysoce kvalitních (fotorealistických) scén (obrázků). Je založen na metodách sledování paprsku (ray-tracing).

Mental ray umí simulovat globální osvětlení (*global ilumination*), které je fyzikálně přesné, dále umí simulovat kaustické efekty (*caustics*), *SSS* (*sub surface scattering*), ostré i měkké stíny a mnoho dalších.

Mental ray je buď jako *stand-alone* aplikace, která zpracovává vstupní soubory se scénou ve formátu **.mi* (*mental image*) nebo jako zásuvný modul v některém z 3d programů. Například *Autodesk Maya*, *Autodesk 3ds Max*, *SoftimageXSI*, *Side Effect Houdini*, *SolidWorks* nebo *Catia*.

Mental ray umožňuje definovat vlastní shadery v prostředí programovacího jazyka c/c++, kromě kompletních stromů definůjících dané jevy (*phenomenon*) i procedurální textury a geometrii, čímž se z něj stává velmi silný nástroj, který dává grafikovi (programátorovi) téměř neomezené možnosti realizace.

Systém mental ray je používán v mnoha odvětvích grafického průmyslu: od filmu (nejnověji *Arthur et les Minimoys - Arthur and the Invisibles, Poseidon, Brothers Grimm, 2046, The Day After Tomorrow, Matrix 2 a 3, Star Wars : 2* a mnoho dalších), přes reklamy, ke hrám a nejrůznějším vizualizacím (*Maybach*, *Siemens*, *Renault* a jiné).

## 5.2 Historie mental ray a mental images

V dubnu roku 1986 vzniká v Berlíně společnost mental images a začíná práce na projektu mental ray. Vedoucí projektu jsou John Andrew Berton, John Nelson a Stefen Fangmeier.

V roce 1989 vychází první komerční verze programu mental ray.

V letech 1990 - 1992 je vyvíjen geometrický modul pro mental ray, který podporuje formát *.obj (Wavefront technologies) a vychází ve verzi 1.6.

1993 V dubnu tohoto roku je mental ray integrován do programu Softimage3D a je vytvořeno první rozhraní pro shadery napsané v C. Poprvé je použit cartoon shader pro rendering 3D scén, které vypadají jako 2D ručně kreslené komixy.

1994 Během projektu DESIRE je vyvíjena paralelizace programu.

Obrázek 5.1: Arthur et les Minimoys, mentalimages.com.

1995 Začíná vývoj mental ray 2.0, který je zcela přepracován (distribuovatelná editovatelná databáze scény), založen na zkušenostech z projektu DESIRE.

1996 Mental ray 2.0 je integrován do SoftimageSumatra (dnes SoftimageXSI).

1997 Mental ray 2.1 přichází s výpočtem globálního osvětlení a příslušně přepracovaným API pro shadery.

1999 Vývoj mental ray 3.0 s novou dataflow architekturou, mental ray 2.1 je integrován do SoftimageXSI a 3ds max.

2001 Dokončení mental ray 3.0.

2002 Mental ray spolupracují s CAD 3D Working Group na standardizaci formátu pro zpracování a manipulaci 3D dat na webu, mental images úzce spolupracují s Industrial Lingt Magic, mental ray je kompletně zapracován do programu AliasWavefront Maya, vydání RealityServer 1.0, vydání mental ray 3.1.

2003 Academy Award

2007 Aktuálně poslední verze programu je 3.5

2008 Mental ray je skoupen společnosti nVidia

...opsáno z mentalimages.com [7].

## 5.3 Renderer

Jádro programu tvoří vlastní renderer. Podporuje jak více jader/procesorů, tak i práci v síti. Zejména síťový rendering a maximální využití procesoru (procesorů) jsou dnes hlavními

požadavky na renderer. Základem je dělení úkolu na menší podúkoly, které je možno kompletovat nezávisle.

Program používá pro výpočet výsledného obrazu několik algoritmů. Základní je raytracing, sledování paprsku, ale urychlení výpočtu používá i scanline algoritmus (například pro určení viditelnosti a pro primární paprsky). Celý proces sledování paprsku pak ukládá do BSP stromu. Některé parametry stromu jsou uživatelsky nastavitelné, takže uživatel může sám ovlivnit velikost a hloubku stromu.

Mental ray podporuje základní grafická primitiva: polygony (konvexní, konkávní) a free-form povrchy (včetně ořezávání). Mezi tyto patří:

- NURB splajny
- Taylorovy monoidy
- Beziérovy křivky

Mezi těmito povrchy lze definovat vazby.

Na veškerou geometrii lze aplikovat *displacement* a při použití *mental matter* lze využít i robustní subdivision pro povrchy.

Lze definovat prakticky libovolný materiál (shader), kromě toho podporuje i NPR (non photorealistic) shadery (samozřejmě jsou počítány metodou sledování paprsku, ale výstup je upraven), například *cartoon shader*.

Výstupem z programu je libovolný formát RGBA v 8 16 nebo 32 bitové hloubce na kanál.

### 5.3.1 Vlastnosti programu

Free-form povrchy program převádí na trojúhelníkovou síť. Program disponuje celou řadou algoritmů.

**Pravidelná mřížka** Dělí povrch na stejně velké intervaly podle vstupního parametru

**Edge length** Rozdělí povrch na stejně velké intervaly podle jednotek kamery nebo tělesa nebo velikosti rastru obrazu. Například při dělení plochy na trojúhelníkovou síť menší než jeden pixel by byla hodnota 0,5 a velikost rastru obrazu.

**Distance dependent** Dělí plochu, dokud se velikost trojúhelníku nevejde pod hranici určenou velikostí objektu.

**Angle dependent** Dělí povrch, dokud nedostane úhel mezi normálami dvou sousedních trojúhelníků pod zvolenou hranicí.

Mental ray od verze 3.1 obsahuje nový algoritmus *fine aproximation*, který kombinuje předešlé metody, aby dosáhl optimálního dělení v rámci celého povrchu. Předpokládá se použití displacement map, na kterých nesmí být vidět artefakty po dělení na trojúhelníky. Tento algoritmus nelze použít pro povrchy, které byly nějakým způsobem pospojovány z více základních ploch.

Free-form plochy lze ořezávat ořezovými křivkami a spojovat jejich hrany. Od verze 3.2 mental ray rovněž podporuje hierarchické dělitelné plochy.

Mental ray definuje každé prostředí, kterým se paprsek pohybuje. To nemusí být jen v objektu. Výchozí prostředí (atmosféra) je zcela průhledná a má index lomu 1. Nicméně lze definovat procedurální shader, který bude simulovat libovolnou atmosféru.

Materiál je základ. Materiál definuje, jak se paprsek bude chovat po interakci s objektem. Mental ray definuje základní typy shaderů:

**Materiál** Tento základní shader definuje barvu objektu v daném bodě. Parametry shaderu mohou být libovolné další shadery.

**Volume shader** Definuje chování paprsku, který prochází objektem.

**Photon shader** Určuje, jak bude materiál reagovat na paprsky nepřímého osvětlení. Je podobný material shaderu, který je určen pro přímé osvětlení, zatímco tento shader je určen pro nepřímé osvětlení.

**Photon volume shader** Určuje chování paprsku nepřímého osvětlení uvnitř tělesa.

**Environment shader** Definuje barvu okolí pro odrazy, které nejsou počítány sledováním paprsku.

Mental ray se svou konstrukcí hodně podobá systémům RenderMan.

## 5.4 Scene description language

Jazyk pro popis scény (SDL) je důležitou součástí každého renderovacího systému. Mental ray obsahuje jednoduše čitelný ascii SDL, ne nepodobný standardu renderman interface specifikace.

Jelikož nemáme možnost renderovat přímo scény v tomto formátu, není potřeba tento jazyk rozebírat. Maya i 3ds max mají interní konvertor, který při použití systému mental ray automaticky vyexportuje scénu včetně odpovídajících parametrů. Protože ani jeden z programů neumožňuje tento formát souboru načíst, pouze jej exportovat, není potřeba se jím v tuto chvíli více zabývat. Několik úkázkových souborů je na přiloženém dvd.

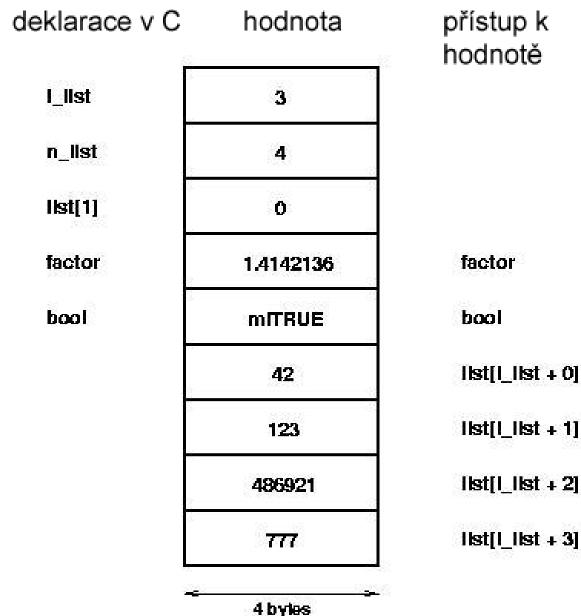
## 5.5 Shading language

Shading language v systému mental ray využívá programovacího jazyka `c/c++`. Mental ray obsahuje knihovnu a hlavičkový soubor, které stačí připojit ke kompilaci a přeložit zdrojové soubory jako dynamickou knihovnu. Veškeré funkce a datové typy pro mental ray začínají přeponou `mi_` (mental images). Podle konvence by se při psaní shaderů měly používat datové typy s předponou `mi_`. Pro čísla definuje `mi_scalar`, pro vektor `mi_vector`, pro boolean `mi_boolean` a podobně. Zvláštním způsobem musí být zpracována struktura pole. Protože je mental ray založen na virtuální sdílené databázi, která přesouvá potřebná data (jako například parametry shaderů) z jednoho počítače na druhů, nemohou data být v podobě odkazů, protože by na jiném počítači byla neplatná. Proto se pole přesouvá pomocí více parametrů, aby fungovalo i po přesunu výpočtu na jiný počítač. V deklaraci se vždy uvádí pole jako `list[1]`, takže je bezpečné přistupovat na první položku v seznamu, pro kterou je alokován prostor daný z deklarace. Ale struktura

```
struct shader_st{
    int i_list, n_list;
miIteger list[1]; miScalar factor; miBoolean bool; }
```

by vypadala v klasickém `C` takto: druhý prvek seznamu *list* by měl stejnou hodnotu jako proměnná *factor* a třetí by se překrýval s proměnnou *bool*. Proto parametr *i_list* udává počet proměnných, které je třeba „přeskočit“, abychom se dostali na další položky seznamu list.

Při deklarování parametrů shaderu máme většinou jen jeden seznam, a to seznam světel. Pokud tomu tak je, můžeme si usnadnit práci s polem tak, že umístíme seznam světel jako poslední prvek struktury, čímž nemusíme určovat hodnotu i, která určuje počet míst pole, které je potřeba „přeskočit“.




Obrázek 5.2: Zpracování pole v systému mental ray.

Součástí knihovny s příslušným shaderem je i deklarační soubor ve formátu mi (SDL), ve kterém je popsán shader se svými parametry a datovými typy. Názvy proměnných a názvu shaderu se musí shodovat v obou souborech (deklarační mi i zdrojový C).

Základem je koordinační systém. Pomocí něj systém reprezentuje vztahy mezi body a vektory v prostoru. Mental ray má vše ve svém „internal space“, tedy vnitřním prostoru. Při psaní shaderu musíme na tento fakt dát pozor a nesmíme předpokládat, že internal space je shodný s world space, ačkoliv tomu tak většinou je. Koordinační prostory jsou:

World space (prostor scény) – v tomto systému je definována scéna, modely a animace.

Object space – je systém vztažen k počátku daného objektu. Pokud tvůrce objektu nedefinoval tento počátek jinak, je počátek souřadnic ve středu obálkového tělesa (většinou kvádr).

Camera space – je systém, kde je kamera v počátku souřadnic (0,0,0) s vektorem směrem vzhůru (0,1,0) a vektorem směrem dolů se zápornou Z osou.

Screen space – je systém obrazu, je jen dvourozměrný, kde (-1, -1/a) je levý dolní roh a (1, 1/a) pravá horní roh. A je aspect ratio - poměr výšky a šířky obrazu.

Mental ray definuje několik základních typů shaderů, které se volají při sledování paprsku, další může uživatel definovat sám. Kdy jsou které typy shaderů automaticky volány, ukazuje obrázek 5.3. Základní typy shaderů jsou tyto:

- Material shaders (shadery materiálu) – základní shader, který popisuje vlastní materiál. Tento shader je volán kdykoliv, kdy viditelný paprsek (eye ray, reflected ray, refracted ray, transparency ray) narazí do daného objektu. Ve scéně musí být alespoň jeden material shader. Uvnitř material shaderu mohou být podle potřeby volány další typy shaderů.

- Volume shaders (shadery objemu) – shadery, jenž definují vlastnosti prostoru, kterým paprsek prochází. Definuje většinou atmosférické jevy. Dělí se na dva typy: klasický volume shader a refraction shader, který je volán při průchodu paprsku tělesem. Tyto shadery také jako jediné přijímají vstupní barvu, jejíž hodnotu modifikují.

- Light shaders (shadery světla) – implementují charakteristiku světelných zdrojů ve scéně. Tyto shadery jsou volány většinou material shaderem, pokud potřebuje vypočítat světlo. Light shader také vytváří shadow rays.

- Shadow shaders (shadery stínu) – tyto shadery jsou volány namísto material shaderu, pokud shadow ray protíná daný objekt.

- Environment shaders (shadery prostředí) – tyto shadery jsou volány, pokud paprsek opustí scénu bez kolize s nějakým objektem.

- Photon shaders (shadery fotonu) – tyto shadery jsou používány pro propagaci fotonů ve scéně. Mají za úkol počítat globální osvětlení nebo kaustické efekty. Trasa každého fotonu ve scéně je sledována pomocí metody photon tracing (podobná metodě ray tracing). Nejdůležitějši rozdíl oproti ray tracing metodě je, že foton shader upraví hodnoty (energii) fotonu před jeho další cestou scénou. Photon shader ukládá získané informace do photon mapy. Výsledek je pak použit material shadery při samotném vykreslování scény.

- Photon volume shaders – fungují podobně jako volume shadery, ale počítají s fotony.

- Photon emmiter shaders (shadery zdroje fotonů) – kontrolují vyzařování fotonů ze světelných zdrojů ve scéně. Pokud jsou ve scéně použity (tzn. počítá se globální osvětlení), vytvářejí photon mapu.

- Texture shaders – se dělí na tři typy, podle toho, co počítají a co je jejich výsledkem: barva, skalár, vektor. Jejich úkolem je odprostit od těchto výpočtů ostatní typy shaderů. V těchto shaderech se nepočítá informace o světlech a systém mental ray je nikdy nevolá přímo, vždy prostřednictvím jiného typu shaderu.

- Displacement shaders (vytlačení) - jsou volány v průběhu *teselace* (dělení) povrchů objektů a upravují pozici jejich vrcholů v ose jejich normál.

- Geometry shaders – jsou volány před samotným procesem vykreslování. Mají za úkol vytvořit geometrii objektu včetně jeho hierarchie.

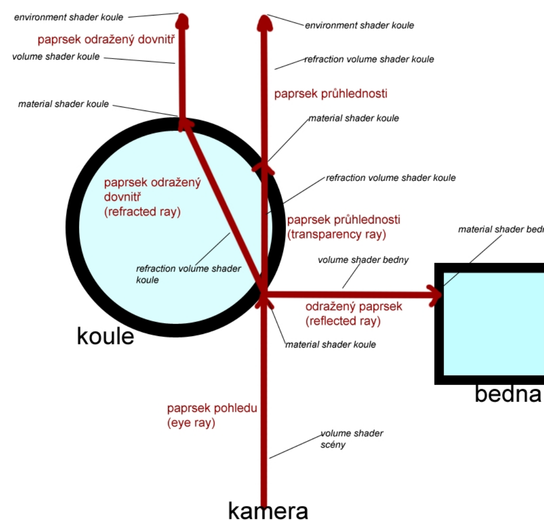
- Contour shaders - NPR shdadery pro zvýrazňování hran.

- Lens shaders (shadery čočky) – tyto shadery se volají jako primární paprsky vyslané kamerou. Mohou modifikovat počátek paprsku a směr, takže umožňují popisovat i jiné typy kamery než klasickou dírkovou kameru.

- Output shaders (výstupní shadery) – tyto shadery jsou volány na hotový vykreslený obraz. Umožňují definovat různé typy filtrů (například medián), případně kompoziční operace.

- Lightmap shaders – se používají na speciální účely, například mohou být připojeny k material shaderu a ukládat informace o světle do textury do „čitelné“ podoby. Typické využití je například proces zvaný texture baking, kdy se do textury uloží navíc informace o světle a pro další vykreslování se už světlo nemusí počítat, čímž se ušetří délka výpočtu.

- State shadery (stavové shadery) – jsou zvláštní typ shaderů, které se připojují k vlastnostem scény. Jsou volány při čtyřech příležitostech – při vytvoření stavu, při zrušení stavu, před vytvořením prvního samplovacího shaderu a před uložením výsledku samplovacího shaderu.




Obrázek 5.3: Obrázek, který popisuje, kde a jak mental ray volá shadery a paprsky.

V předcházejícím odstavci jsme zmínili několik druhů paprsků. Mental ray rozlišuje mnoho druhů paprsků, každý pro konkrétní účel. Jsou definovány takto:

- `miray_eye` – základní paprsky vysílané z kamery.
- `miray_transparent` – paprsek průhlednosti, volaný material shaderem.
- `miray_reflect` - paprsek odrazu, volaný material shaderem.
- `miray_refract` – paprsek lomu, volaný material shaderem.
- `miray_shadow` – stínový paprsek volaný light shaderem.

- `miray_light` – paprsek volaný při výpočtu světla. Pokud je v cestě tomuto paprsku nějaký objekt, vrací `miray_shadow`
- `miray_environment` – paprsky, které vzorkují mapu prostředí (reflexe).
- `miray_displace` – volán displacement shadery.
- `miray_output` – volán output shadery.
- `miray_finalgather` – paprsky počítající final gather.
- `mi_ray_probe` – je volán funkcí *mi_trace_probe*.
- `miray_none` – veškeré ostatní, zde nedefinované paprsky.

Kompletní seznam je v manuálu systému mental ray.

Každý shader má své parametry, kterými lze ovlivnit jeho chování. V souboru s popisem scény vypadá reference na daný shader velmi podobně jako volání funkce v jazyce c/c++. Za názvem shaderu je v závorce výčet hodnot vstupních parametrů. Mental ray nedefinuje žádný formát ani počet parametrů shaderu, je možno předat shaderu libovolná data. Výrazem parametr budeme označovat parametr v definičním .mi souboru a výrazem argument budeme označovat argumenty funkcí v jazyce C. Mental ray žádným způsobem neupravuje ani nečte parametry v definičním souboru, pouze je předává příslušnému shaderu, když je zavolán. Deklarační soubor vypadá takto:

```
declare shader
       [ type ] " název shaderu " (
           type " název parametru ",
           type " název parametru ",
           ...
           type " název parametru "
       )
       [ version číslo verze ]
       [ options ]
   end declare
```

Shadery se mohou skládat z malých jednoduchých „základních“ shaderů a dohromady mohou být spojovány a tvořit velký a složitý celek. Díky tomu se může autor každého ze základních shaderů soustředit jen na konkrétní problém. Takto lze jednoduše nahradit konstantní barvu jako vstupní parametr za texturu. To usnadňuje práci tvůrci shaderu, který nemusí přemýšlet nad tím, který z parametrů bude barva a který textura a nebude muset přidávat do struktury parametrů obě verze „pro jistotu“, kdyby chtěl uživatel použít barvu nebo texturu.

Shadery také často nepotřebují všechny své parametry, například velmi složitý materiál složený z několika shaderů, volaných podle toho, kterou stranu povrchu paprsek zasáhne (vnitřní nebo vnější). Bylo by velmi neefektivní, aby shader při každém volání počítal obě hodnoty a nakonec pouze podle směru paprsku vybral jednu z hodnot. Proto mental ray neinicializuje parametry shaderu před jeho voláním. Načtení parametrů musí být provedeno až v samotném shaderu. Toto se dělá pomocí funkcí `mi_eval_<datový typ>`. Jedná se o makro, které provádí potřebné přetypování. Pokud je parametrem hodnota, makro `mi_eval_` vrátí tuto hodnotu. Pokud je parametrem jiný shader, provede se a vrátí se

hodnota vrácená daným shaderem. V případě, že už byl daný shader volán, neprovádí se výpočet znovu, ale je vrácen ukazatel na výsledek předchozího výpočtu. Funkce *mi_eval_* vrací vždy ukazatel na hodnotu parametru nezávisle na tom, odkud přichází (parametr shaderu, výsledek shaderu, ...). K parametrům shaderu by se vždy mělo přistupovat pomocí těchto funkcí.

Mental ray je stavový automat. Při každém volání shaderu (tedy při kolizi paprsku s určitým objektem) mu předává v parametru aktuální stav, ve kterém se nachází. Tato struktura je velmi důležitá, protože obsahuje řadu informací, které bychom zbytečně počítali, kdybychom nevěděli, že v této struktuře jsou. Některé často používané proměnné jsou zde:

- Parent (`miState *`) - odkaz na stav rodičovského paprsku. Pokud se jedná o první paprsek, vyslaný z kamery, je tento odkaz null. Jinak ukazuje na material shader objektu, od kterého paprsek přišel.
- Child (`miState *`) - odkaz na stav paprsku potomka. Pomocí těchto funkcí (parent a child) lze procházet trasu paprsku a získat tak informace o kterékoliv změně směru paprsku a jeho kolizích.
- Type (`miRay_type`) – typ paprsku. Jednotlivé typy jsou vypsány a rozděleny výše.
- Scanline (`miCBoolean`) – pokud je hodnota true, znamená to, že daný průsečík byl nalezen pomocí scanline algoritmu.
- Cache (`void *`) - jedná se o funkci, která by měla urychlit výpočty. Pokud je nastavena na hodnotu 0, umožňuje shaderu volat funkce, které nejsou pro daný typ určeny.
- Org (`miVector`) – souřadnice počátku daného paprsku.
- Dir (`miVector`) – směr paprsku, ze kterého přichází. Toto je důležité, protože pro mnoho výpočtů používáme paprsek směrem „ke kameře“, tedy ke zdroji, odkud paprsek přišel. Je proto potřeba někdy směr otočit. Hodnota tohoto vektoru je v normalizovaném tvaru (převedena na jednotkový vektor).
- Time (`float`) – je hodnota pro motion blur, udávající dobu otevření clony.
- Volume (`miTag`) – volume shader daného objektu.
- Environment (`miTag`) – environment shader dané scény.
- Reflection a refraction_level (`int`) – udává aktuální hloubku paprsků odrazu a lomu.
- Refraction_volume (`miTag`) – obsahuje volume shader daného materiálu, který bude aplikován na paprsky lomu.
- Point (`miVector`) – obsahuje souřadnice průsečíku paprsku s objektem.
- Normal (`miVector`) – obsahuje normálový vektor povrchu v daném průsečíku. Tento vektor je normalizovaný. Při výpočtech je potřeba dát pozor při výpočtu délky normály, protože jsou hodnoty vektoru uloženy jako float, mají přesnost asi jen na 6 desetiných míst, což může být pro některé matematické výpočty málo. Například funkce acos může vracet hodnotu *NaN* (*not a number* – není číslo), což se promítne jako bílé pixely ve výsledném obraze (které tam určitě budou nežádoucí).

- Inv_normal (`miCBoolean`) – je true, pokud paprsek našel průsečík s objektem „zevnitř“ (tedy směr paprsku a směr normály je stejný). Mental ray v tom případě automaticky otáčí směr proměnné *normal* a *normal_geom*. Proměnná *inv_normal* je hodně používaná při průhledných materiálech, kde je potřeba definovat jiné chování při „vstupu do“ materálu a při „výstupu z“ materiálu.
- Dot_nd (`miScalar`) – je předvypočítaný skalární součin vektorů *dir* a *normal*. V případě paprsků od světla je to záporná hodnota součinu paprsku ze směru světla a normály v daném bodě.
- Dist (`double`) – délka paprsku, vzdálenost od *org* do *point*. Pokud je hodnota nula, pak nebyl nalezen žádný průsečík.
- Ior (`miScalar`) – index lomu material shaderu objektu.
- Ior_in (`miScalar`) – index lomu shaderu, ze kterého paprsek přišel.

Kompletní popis všech proměnných uchovávaných ve struktuře state je v manuálu systému mental ray.

Důležitou funkcí v systému mental ray je *mi_warning*, pomocí které je možno kontrolovat v okně *mental ray info window* aktuální hodnoty shaderu. Jiná metoda kontroly funkce shaderu, kromě této, tedy pomocí výpisů, pravděpodobně není.

### 5.5.1 Používání shading language pro psaní vlastních shaderů

Než začneme psát vlastní shader, je potřeba přesně vědět, co budeme chtít vytvořit. Shadery nazýváme anglickým výrazem *phenomena* (jev, úkaz), zřejmě podle toho, že popisují fyzikální jevy. O *phenomena*, který chceme popsat je důležité sesbírat podklady, například v podobě obrázků, ideálně ale rovnic, které by popisovaly chování světla na povrchu nebo pod povrchem objektu [8].

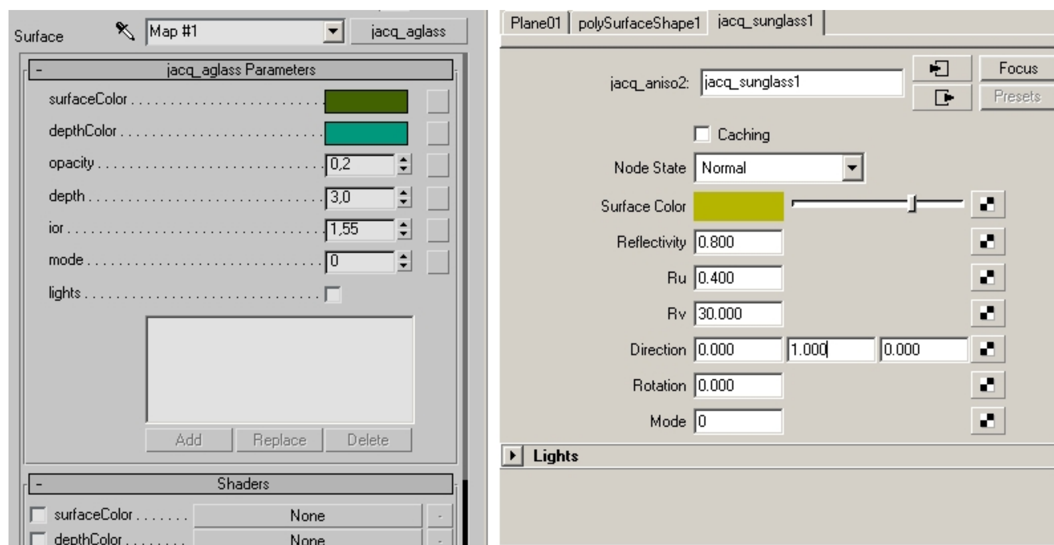
Další fází by mělo být rozepsání do tzv. *shader tree*, nebo *shader graph*, který by popisoval jednotlivé části shaderu a definoval vzájemné vazby a postup míchání barev. Před prvním shaderem ale určitě nebudeme mít zkušenosti se psaním těchto grafů, proto se k tomu vrátíme v závěru textu.

Nyní již máme potřebné informace a můžeme začít psát shader. Každý shader MUSÍ obsahovat hlavičkový soubor „shader.h“.

Takže můžeme začít tím, že si tento soubor připojíme k našemu (`#include shader.h`).

Následuje většinou deklarace parametrů shaderu, které jsou uloženy ve struktuře. Jak již bylo napsáno výše, je čistě na uživateli, jaké parametry bude shaderu předávat, ale je vhodné používat datové typy definované systémem mental ray. Typicky bývá parametrem číslo, barva, vektor a seznam světel (jak je třeba nakládat se seznamem je popsáno výše). Pro barvu je připravena struktura `miColor`, která obsahuje 4 položky *r, g, b, a*, které odpovídají barvám *red, green, blue a alpha* (průhlednost objektu). Hodnoty jednotlivých barevných kanálů mohou být libovolné číslo, většinou ale v rozmezí 0 do 1. Protože se jedná o desetinné číslo, je přesnost barevné informace vysoká a může být uložena i jako obraz s vysokým dynamickým rozsahem (*HDR*), případně může být upravena dodatečnými output shadery, které provádějí kontrolu expozice. Většina číselných hodnot, například různé koeficienty, je uložena v typu `miScalar`. Jedná se o typ čísla s desetinnou čárkou, takže je vhodné například pro hodnotu indexu lomu, nebo na koeficienty intenzity různých jevů. Dalším parametrem shaderu může být vektor. Opět se jedná o strukturu, má tři složky x, y, z.

Jednotlivé složky jsou čísla s desetinou čárkou, nabývající libovolných hodnot. Reprezentují vektor v prostoru souřadnic x, y, z. Z dalších hodnot je možno použít například `miInteger`, `miTag` (pro seznam). Pokud použijeme datové typy mental ray, usnadníme si tím práci v prostředí grafických programů, které mají mental ray integrovaný, protože většinou umí vytvořit na základě deklaračního souboru přímo grafické rozhraní pro nastavování těchto parametrů.




Obrázek 5.4: Ukázka shaderu načteného pouze z deklaračního souboru .mi v prostředí 3ds max a Maya.

Při deklarování parametrů určuje autor míru nastavitelnosti daného shaderu. Podle potřeby se může rozhodnout, jestli vytvoří shader čistě pro jeden konkrétní účel, který bude mít minimum parametrů a bude simulovat jeden konkrétní materiál, nebo bude mít parametry nastavitelné, čímž dá uživateli možnost upravovat shader a vytvářet pomocí něj větší škálu materiálů (v rámci možností shaderu). Samozřejmě důležitým prvkem je rychlost shaderu. Čím konkrétnější shader bude, tím může být rychlejší, protože parametry budou nastaveny fixně v zdrojovém kódu, nebude se muset kontrolovat hodnota parametru (například záporná hodnota v indexu lomu), což samozřejmě zdržuje, pokud se daný shader volá třeba miliónkrát a pokaždé kontroluje dané hodnoty.

Po deklarování této struktury je potřeba vytvořit funkci, která vrací verzi konkrétního shaderu. Tato verze se poté kontroluje s deklaračním .mi souborem daného shaderu a musí odpovídat verzi volaného shaderu. Tato funkce má předepsaný tvar a návratovou hodnotu. Její tvar vypadá takto:

```
int <nazev_shaderu>_version(void) {return (<cislo verze>);}
```

Verzování shaderů je velmi důležité, asi jako všude jinde, hlavně pokud se pracuje na velkých projektech, kde se neustále upravují a předělávají shadery, je potřeba, aby byly pro vykreslování vždy správné verze. Mental ray před inicializací zkontroluje verzi v deklaračním souboru se samotným shaderem a pokud verze nesedí, vypíše chybovou hlášku. Pokud verze shaderu v jednou ze souborů chybí, vypíše upozornění a použije výchozí hodnotu 0. Verzi shaderu lze zjistit pomocí funkce `mi_query`:

```
mi_query(miQ_DECL_VERSION, state, 0, &version);
```

Do proměné version funkce vrátí hodnotu čísla verze.

Nyní se již můžeme pustit do vlastního shaderu. Funkce má opět předepsaný tvar a návratový typ. Typicky vypadá funkce (její argumenty) takto:

```
miBoolean <nazev_shaderu>(
                miColor *result,
                miState *state,
                struct <struktura parametru shaderu> *paras)
```

*Result* je výsledná barva shaderu. Výstupem většiny shaderů je barva, zejména pokud se jedná o material shadery. *State* je aktuální stav, ve kterém se mental ray nachází. Je to struktura, ve které jsou obsaženy téměř všechny hodnoty užitečné při psaní shaderu (viz výše). *Paras* je odkaz na strukturu, která obsahuje parametry shaderu (opět, viz. výše). Návratový typ je `miBoolean`. Většina funkcí v systému mental ray má tento návratový typ a argumenty předává odkazem. Hodnota `miBoolean` informuje o tom, zda daná funkce proběhla v pořádku či nikoliv.

V úvodu je vhodné zkontrolovat, jaký typ paprsku tento shader volá, aby se zbytečně nepočítaly složité operace, pokud nemusí. Například pokud se jedná o stínový paprsek (`miRAY_SHADOW`), není většinou potřeba počítat nic a stačí výpočty ukončit vrácením `miFALSE`. Samozřelmě může nastat situace, při které budeme chtít i barvu stínového paprsku počítat.

Jelikož je mental ray založený na sledování paprsku, budeme i v shaderu převážně sledovat paprsky a z těchto výsledků budeme skládat výslednou barvu. Barvu objektu budeme získávat dvěma základními způsoby. Buď pomocí informací ze světelných paprsků, upravených některým z osvětlovacích modelů (případně vlastní rovnicí), pokud se jedná o objekty s difúzními povrchy, nebo pomocí sledování odrazových paprsků a paprsků lomu, pokud se jedná o lesklé či průhledné objekty. Samozřejmě je možno oba způsoby libovolně kombinovat.

První způsob zpravidla znamená projít všechna světla ve scéně a pro každé z nich vypočítat rovnici zvoleného osvětlovacího modelu. Základní rutina je však pro všechny výpočty stejná. Jedná se o smyčku, která prochází seznam světel a volá světelný paprsek k danému světlu. S vrácenými argumenty pak lze provádět potřebné výpočty. Rutina vypadá takto:

```
mode  = *mi_eval_integer(&paras->mode);
```

Mód, ve kterém jsou světla. Existují tři základní módy. Výchozí, ve kterém se zpracovávají všechna světla ve scéně, inkluzivní, ve kterém se zpracovávají jen vybraná světla a exkluzivní, ve kterém se zpracovávají všechna světla kromě zvolených světel.

```
n_l   = *mi_eval_integer(&paras->n_light);
```

Informace o počtu světel ve scéně

```
i_l   = *mi_eval_integer(&paras->i_light);
```

Informace o indexu, na kterém pokračuje pole (viz výše, obrázek XX)

```
light =  mi_eval_tag(paras->light) + i_l;
```

Načte první světlo v seznamu.
`if (mode == 1)`
`mi_inclusive_lightlist(&n_l, &light, state);`
Nastaví mód, pokud je třeba
`else if (mode == 2)`
`mi_exclusive_lightlist(&n_l, &light, state);`
Ve smyčce projde všechna světla v seznamu

```
for (n=0; n < n_l; n++, light++)
    while (mi_sample_light(&color,
                           &dir,
                           &dot_nl,
                           state,
                           *light,
                           &samples))
```

Funkce *mi_sample_light* má argumenty typu `miColor` – vrací barvu světla, `miVector` – směr paprsku od světla do daného bodu, v normalizovaném tvaru, `miScalar` – vrací hodnotu skalárního součinu normálového vektoru v daném bodě a vektoru paprsku světla, `miState` – stav, ve kterém se renderer momentálně nachází, `miInteger` – počet vzorků. Většina světel je vzorkována, to znamená, že je vysláno několik paprsků. Pokud tomu tak je, je potřeba výsledný součet hodnot ve výsledné barvě vydělit počtem vzorků, jinak bude výsledek pravděpodobně jen bílý.

Pokud funkce narazí na překážku v cestě od světla k objektu, vrátí false a barvu stínového paprsku.

V těle smyčky pak pro každé světlo, pro každý z paprsků, vypočteme rovnici osvětlovacího modelu.

Pokud je těleso lesklé, používáme pro výpočet tyto funkce:

- `mi_trace_reflection`,
- `mi_trace_refraction`,
- `mi_trace_transparent`,
- `mi_trace_environment`.

Tyto funkce jsou z kategorie funkcí, které mají za úkol sledovat paprsek, *mi_trace_*. Všechny funkce jsou vypsány v manuálu mental ray, zde se zmíníme jen o těchto nejpoužívanějších.

*Mi_trace_reflection* i *mi_trace_refraction* mají stejné parametry, liší se pouze v tom, že jedna funkce je určena pro sledování paprsku který se odráží od povrchu a druhá je určena pro sledování paprsku, který se odráží dovnitř, pod povrch. Jejich parametry jsou `miColor` – ukazatel na výslednou barvu, `miState` – ukazatel na aktuální stav a `miVector` – ukazatel na vektor, který určuje směr, ve kterém se má daný paprsek odrazit.

*Mi_trace_transparent* je jednodušší verze funkce *mi_trace_refraction*, která pokračuje ve směru paprsku, který do daného objektu narazil, tedy ve směru vektoru `state->dir`. Její parametry jsou stejné jako u předešlých funkcí, ale nemá `miVector`, protože ten je již uložený v proměnné typu `miState`.

Ještě je potřeba vysvětlit, jak získáme směr, kterým se mají paprsky odrážet. Mental ray disponuje funkcí, která tento vektor vypočítá na základě indexů lomu prostředí, ve

kterých se paprsek nachází. Tyto funkce se jmenují *mi_reflection_dir* a *mi_refraction_dir*. Jejich parametry jsou si podobné, obě funkce mají typ `miVector` – ukazatel na výsledný vektor, určující směr a typ `miState` – ukazatel na aktuální stav. *mi_refraction_dir* má navíc 2 parametry typu `miScalar` – index lomu prostředí, ve kterém se paprsek nachází a index lomu prostředí, do kterého se má paprsek odrážet. Druhá funkce tyto parametry nepotřebuje, protože se odráží do stejného prostředí, v jakém se paprsek nachází.

V případě, že paprsek výše uvedených tří funkcí nenarazí na žádný objekt, vrací hodnotu *false*. V takovém případě je potřeba zavolat funkci *mi_trace_environment*, která má stejné parametry, jako funkce *mi_trace_reflection* a *refraction*.

Ukázka toho, jak vypadá základní varianta materiálu skla (lom paprsků) je zde:

```
if (iorIn == ior) {
    if (!mi_trace_transparent(&tracedColor, state))
        if (!mi_trace_environment(&tracedColor, state, &state->dir))
            tracedColor = *result;
} else {
    if (mi_refraction_dir(&dir, state, iorIn, ior))
    {
        if (!mi_trace_refraction(&tracedColor, state, &dir))
            if (!mi_trace_environment(&tracedColor, state, &dir))
                tracedColor = *result;
    }
    else
    {
        mi_reflection_dir(&dir, state);
        if (!mi_trace_reflection(&tracedColor, state, &dir))
            if (!mi_trace_environment(&tracedColor, state, &dir))
                tracedColor = *result;
    }
}
```

Pokud je index lomu prostředí, do kterého se paprsek láme stejné, jako prostředí, ze kterého paprsek přichází (předpokládáme, že přichází z prostředí, kde je vzduch, tedy index lomu blízký 1), není třeba počítat odraz ani lom, stačí paprsek poslat dál ve směru, ze kterého přišel.

Pomocí funkce *mi_refraction_dir* se pokoušíme nalézt směr, kterým by se měl paprsek do objektu odrazit. Pokud je úhel dopadajícího paprsku větší než kritický úhel, dochází k jevu zvaném úplný vnitřní odraz, takže by se paprsek odrazil mimo objekt, a proto tato funkce vrátí *false*. Je potřeba toto kontrolovat.

V případě, že byl paprsek nalezen, můžeme vyslat paprsek lomu do tělesa a získat požadovanou barvu. Jelikož se může stát, že paprsek nenajde žádný další průsečík (například když daný objekt není uzavřený), bylo by vhodné dát i tuto funkci do podmínky, a pokud vrátí *false*, zavolat nakonec funkci *mi_trace_environment*, která vrátí barvu prostředí dané scény.

Zde vypočítáme úplný vnitřní odraz.

`mi_reflection_dir(&direction, state);`

Tedy nalezneme vektor, pod kterým máme vyslat odražený paprsek. Tato funkce za většiny „normálních“ okolností vrací *true*, proto zde není běžně potřeba kontrolovat průběh.

Funkce by vrátila *false* například při záporné hodnotě indexu lomu, což by ale mělo být kontrolováno už při načítání parametru.

```
mi_trace_reflection(result, state, &direction);
```

Vyšleme paprsek požadovaným směrem a získáme hledanou barvu. Opět bychom měli být korektní a uzavřít funkci do podmínky, protože se může stát, že odražený paprsek opustí scénu a žádný další průsečík nenajde. V takovém případě opět daným směrem vyšleme paprsek pomocí funkce *mi_trace_environment*, který vrátí barvu prostředí scény.

Pro větší přehlednost a také pro zvýšení rychlosti je možné použít inicializaci shaderu. Jedná se o předdefinovanou funkci, která má tvar `<jmeno_shaderu>_init`. Tato funkce se volá před vykreslováním scény, při načítání shaderů. Výhodou je, že je možno inicializovat potřebné hodnoty a parametry, které určitě budou potřeba, jen jednou před začátkem samotného výpočtu scény. Jak bylo zmíněno výše, mental ray při opakovaném volání funkce *mi_eval_* vrací už pouze odkaz na požadovanou hodnotu.

Po vykreslení scény je rovněž vhodné „uklidit“ po shaderu, pokud je potřeba. Mental ray opět má předdefinovanou funkci ve tvaru `<jmeno_shaderu>_exit`, která se volá po ukončení vykreslování scény.

Shader by nikdy neměl zapisovat do parametrů shaderu, protože můžou být používány jinými vlákny programu.

### 5.5.2 Příklady shaderů

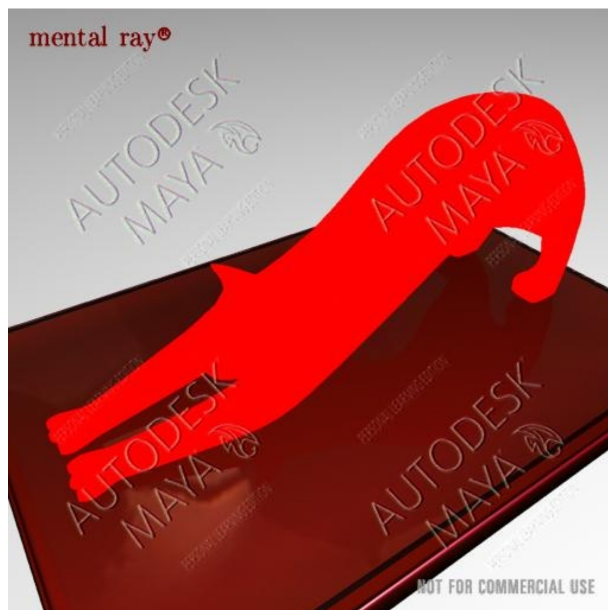
#### jacq_constant_simple

Tak jako je v programování první ukázkový příklad program „*hello world!*“, existuje podobně jednoduchý shader, i když jeho napsání a spuštění ve srovnání s programem *hello world* tak jednoduché není. Jedná se o shader, který je pojmenován *constant simple*. Jak již název napovídá, jeho výsledkem bude barva, která bude konstantní po celém objektu. Jako parametr shaderu bude uživatelsky definovatelná barva. Tuto barvu shader načte a přepošle ji na výstup. V těle shaderu je jedna kontrola – typ paprsku. Pokud se jedná o paprsek `miRAY_SHADOW` nebo `miRAY_DISPLACEMENT`, vrací shader `miFalse`. Jinak pro všechny ostatní paprsky načte hodnotu paramatru shaderu pomocí *mi_eval_color* a tuto hodnotu uloží do výsledné barvy a vrátí `miTRUE` aby oznámil, že proběhl úspěšně. Zdrojové soubory přiloženy na dvd.

#### jacq_advancedGlass

Dalším shaderem, který zde rozebereme, a který byl pro účely tohoto textu vytvořen, je shader s názvem *advanced glass*. Oproti shaderu skla, který je ve standardní knihovně mental ray se liší v způsobu výpočtu hustoty skla a v některých dalších parametrech. Tento shader má za úkol simulovat materiál přírodního skla, vltavínu. Vltavín má tu vlastnost, že propouští relativně málo světla. Je to způsobeno tím, že nechladl stejnoměrně a vytvořilo se v něm mnoho vrstev a bublinek, na jejichž rozhraní je různý index lomu. I když jsou si indexy velmi blízké, dochází na rozhraní k útlumu světla. Čím více těchto prostředí je, tím je kámen neprůhlednější. Vytvořený shader má tyto parametry:

- SurfaceColor – první barva skla, používaná na tu část, která paprsky propouští.
- DepthColor – druhá barva skla, používaná na tu část, která paprsky nepropouští.



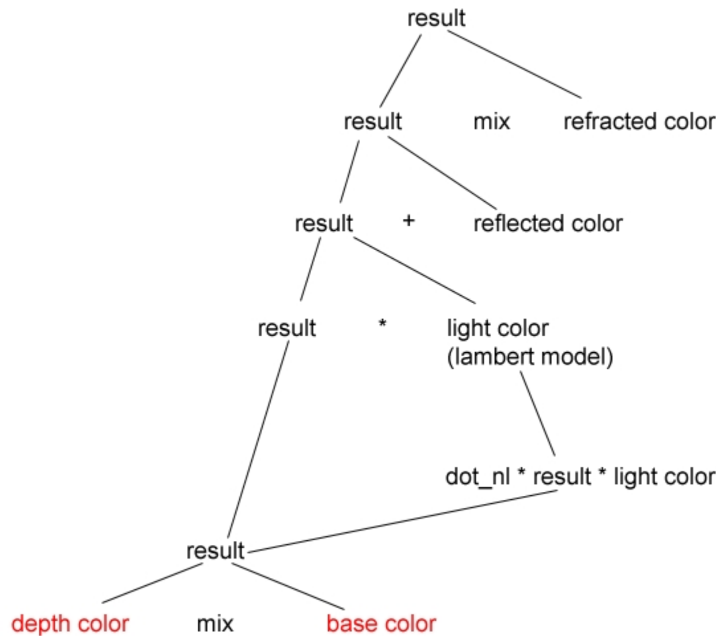

Obrázek 5.5: Shader jacq_constant_simple.

- Opacity – průhlednost, nižší hodnota, určuje, jak minimálně bude sklo průhledné. Nabývá hodnot 0, pro zcela čiré sklo až po hodnotu 1, pro neprůhledné sklo. Pokud bude hodnota například 0,5, bude sklo poloprůhledné a s rostoucí mohutností objektu bude průhlednost klesat.
- Depth – tloušťka objektu, při kterém už bude materiál zcela neprůhledný.
- Ior – index lomu materiálu.
- Na závěr parametry pro načítání světel.

Na obrázku 5.6 se *shader tree* je vidět konstrukce shaderu, kompletní zdrojový kód je na přiloženém dvd. Na začátku shaderu vypočítáme na základě indexů lomu Fresnelův koeficient, který určuje intenzitu odrazu. Čím větší je úhel mezi kamerou a normálou povrchu, tím je odraz intenzivnější. Pro tento výpočet je v systému mental ray funkce *mi_fresnel_reflection*, která má argumenty typu `miState` – aktuální stav a dvakrát typu `miScalar` pro index lomu prostředí ze kterého paprsek přišel, a do kterého se odráží.

V dalším kroku je vypočítána tloušťka objektu. Získáme ji za pomocí funkce *mi_trace_-transparent* (případně by šla použít i fuknce *mi_trace_probe*). Pokud funkce proběhne úspěšně (tedy nalezne průsečík s dalším objektem), můžeme získat vzdálenost z *state-child-distance*. Tloušťku je však potřeba počítat jen tehdy, je-li paprsek vně objektu, tedy pokud do něj teprve bude vstupovat. Tento údaj má mental ray uložený v proměnné state a zmiňovali jsme se o tom v předchozí kapitole. Pokud paprsek zasáhne objekt zevnitř, mental ray nastaví *inv_normal* na *true* a obrátí směr normály.

Nyní můžeme vypočítat základní barvu, na základě poměru parametru distance a tloušťky objektu v daném bodě. Pro mixování barev máme připravenu funkci *mij_mixColors*, která vrací výslednou barvu. Jejími parametry jsou dvakrát `miColor`, tedy dvě barvy, které chceme míchat a poměr míchání, číslo v rozmezí 0 až 1.




Obrázek 5.6: Shader tree materiálu.

Po vypočítání základní barvy k ní můžeme připočítat barvu světel, podle osvětlovacích modelů. Bude nám stačit obyčejný Lambertův model pro difůzní povrch. Výsledná barva se vypočte jako součin základní barvy, barvy světla a skalárního součinu vektorů směrem ke světlu a normály povrchu. Tento součin si mental ray uchovává v struktuře *state* a můžeme jej nalézt ve `state->dot_nl`.

Dalším krokem bude přičtení odrazů. Podle konstrukce, která je uvedena v předchozí kapitole, zjistíme nejprve směr, kterým máme paprsek vyslat a následně se pokusíme získat barvu pomocí příslušné funkce (*mi_trace_reflection*). Pokud funkce nenalezne paprsek, bude třeba získat barvu z prostředí scény pomocí *mi_trace_environment*. Jednoduchý zápis tohoto postupu vypadá takto:

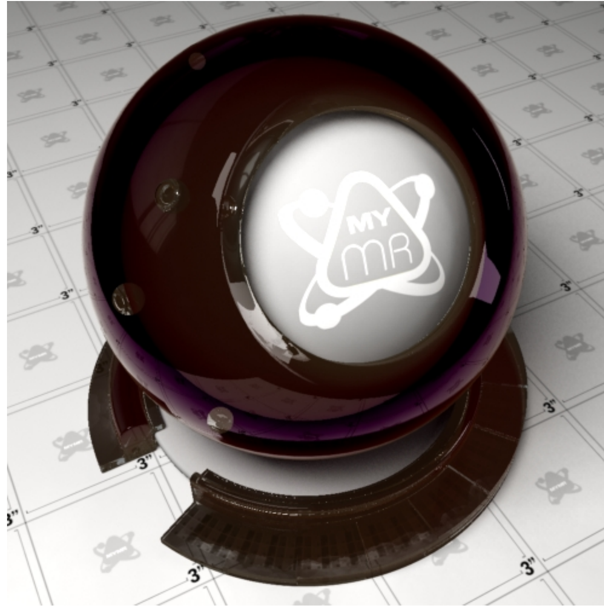
```
mi_reflection_dir(&dir, state);

if (mi_trace_reflection(&tracedColor, state, &dir) ||
mi_trace_environment(&tracedColor, state, &dir))
```

Tento postup nám zaručí, že se mental ray nejprve pokusí získat barvu pomocí *mi_trace_-reflection* a až kdyby byl neúspěšný, bude volat funkci *mi_trace_environment*. Získanou barvu vynásobíme fresnelovým koeficientem, který jsme si vypočítali na začátku a celé to potom přičteme dosavadí výsledné barvě.

Posledním krokem bude výpočet paprsku lomu. Budeme jej opět počítat postupem uvedeným v předchozí kapitole. Nejprve vypočteme směr paprsku a následně tím směrem

paprsek vyšleme. Pokud paprsek nenajde průsečík, zavoláme funkci *mi_trace_environment* a získáme barvu prostředí.

Výslednou barvu vypočteme mixováním doposud výsledné barvy a barvy získané v předchozím kroku. Poměr jednotlivých barev je podle průhlednosti objektu. Výsledný shader je na obrázku 5.7



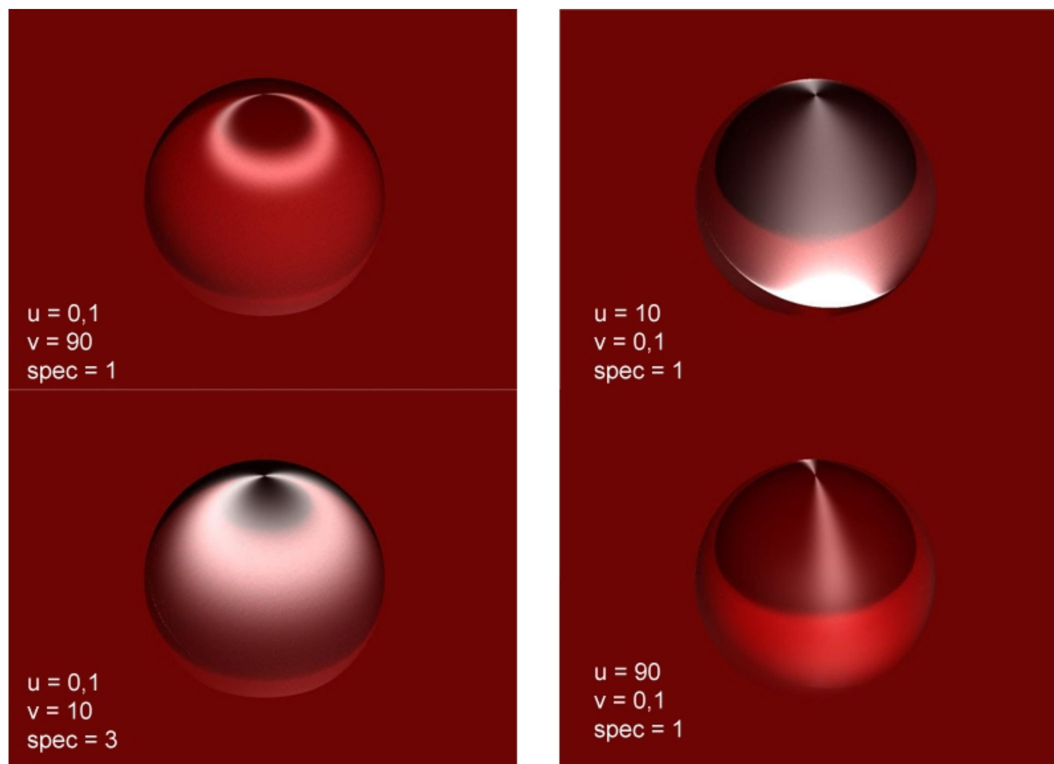

Obrázek 5.7: Obrázek shaderu advanced_glass. Referenční model z [9].

#### jacq_anisoBrdf

Tento shader je implementací anizotropního BRDF Phongova modelu. Mental ray obsahuje funkci na výpočet osvětlení pomocí Wardova anizotropického modelu *mi_ward_anisglossy* [12]. Argumenty této funkce jsou dvakrát `miVector` pro vektory určující směr dopadajícího paprsku a odrážejícího paprsku, `miVector` pro vektor normály povrchu, dvakrát `miVector` pro určení směru anizotropie a dvakrát `miScalar` pro určení intenzity anizotropie v obou směrech. Vektory $u$ a $v$ by měly být navzájem kolmé a měly by být kolmé vůči normále. Ukázka různých nastavení intenzity anizotropie je na obrázku 5.8.

V tomto shaderu ukážeme postup při implementaci vlastních rovnic pro výpočet osvětlovacího modelu. Parametry shaderu jsou tyto:

- Surfacecolor (`miColor`) – definuje základní barvu povrchu.
- SpecularColor (`miColor`) – definuje barvu spekulárního odlesku.
- SpecularReflectance (`miScalar`) – koeficient intenzity spekulárního odlesku.
- RU, rV (`miScalar`) – intenzita anizotropick reflexe ve směru $u$ a $v$.
- Direction (`miVector`) – směrový vektor pro výpočet anizotropického jevu.
- Na závěr parametry pro načítání seznamu světel.




Obrázek 5.8: Ukázka různých hodnot koeficientů anizotropického materiálu.

Nejprve je potřeba vypočítat směr vektorů $u$ a $v$, pro konkrétní bod. Tyto vektory vypočteme pomocí vektoru direction a budeme jej vztahovat k bodu 0, 0, 0. Způsob výpočtu je podle shaderu deriver od Daniela Rinda [1], s několika drobnými úpravami.

Na začátku uložíme do výsledné barvy černou. Poté provedeme běžné rutiny pro práci se světly a pak ve smyčce pro jednotlivá světla počítáme potřebné rovnice. Protože se shader bude volat v průběhu scény mnohokrát, je potřeba vhodně zvolit postup operací. Každá operace navíc se ve výsledku může promítnout velkým nárůstem výpočetního času. Je vhodné si všechny výpočty, které se budou volat více než dvakrát, vypočítat předem. V první fázi vypočteme difúzní složku. Budeme počítat tuto složku jen pro nezáporné hodnoty (tedy tu část plochy objektu, na kterou přímo dopadá světlo).

Poté přidáme spekulární složku. Opět je potřeba v průběhu výpočtu kontrolovat, jestli je výsledek stále kladné číslo.

Pro matematické výpočty obsahuje mental ray celou řadu funkcí, zejména pro operace s vektory. Kompletní seznam je v manuálu systému mental ray. Mezi nejpotřebnější patří skalární součin *mi_vector_dot*, vektorový součin *mi_vector_prod*, násobení vektoru číslem *mi_vector_mul*, příčítání čísla k vektoru *mi_vector_add* a normalizování vektoru *mi_vector_normalize*.

Pro výpočet anizotropního odrazu mental ray obsahuje funkci *mi_reflection_dir_anisglossy*. Jejími argumenty jsou `miVector` pro vektor určující směr paprsku, `miState` pro aktuální stav, vektory `miVector` pro určení směru reflexí a intenzita reflexí v těchto směrech. Všechny potřebné informace máme již vypočítány, stačí dosadit do funkce. Vypočítanou barvu přičteme k výsledné barvě. Výsledný shader je na obráku 5.9

Obrázek 5.9: Obrázek shaderu anisoBrdf.

### jacq_sunglasses

V tomto shaderu skla je přidán efekt rozkladu světla, podobný difrakci. Tento jev můžeme pozorovat například na olejové skvrně v louži. Na některých slunečních brýlích se tento jev vyskytuje v podobě zbarvených odrazů okolí a barva závisí na úhlu, který mezi sebou svírají vektory kamery a normály povrchu. Ukázka takovýchto brýlí je na obrázku 5.10. Pro náš shader jsem tento efekt pojmenoval „duhová“ barva, protože duhu připomíná.

Obrázek 5.10: Vlevo je snímek zachycující jev ve skutečném světě, vpravo výsledek shaderu.

Parametry shaderu jsou:

- SurfaceColor (miColor) pro definování základní barvy povrchu,

- Reflectivity (miScalar) je koeficient udávající intenzitu odrazů,
- Refractivity (miScalar) je koeficient určující míru průhlednosti.
- Samozřejmě je součástí i seznam světel s potřebnými parametry.

Protože jsme již předem definovali, že se jedná o materiál skla, není třeba dávat uživateli možnost nastavovat index lomu, protože index lomu skla je znám (přibližně 1,5). Pořadí výpočtu jednotlivých operací můžeme zaměnit, protože budeme sčítat jen tři složky, na sobě navzájem nezávislé.

Pomocí standardních postupů vypočteme složku reflexe, složku refrakce, podíl osvětlení od světelných zdrojů a „duhový“ efekt. Tento vypočteme na základě úhlu mezi normálou povrchu v daném bodě a paprskem pohledu dopadajícím do daného bodu. Hodnota tohoto úhlu by měla nabývat hodnot od -1 do 1. Vypočteme proto absolutní hodnotu a převedeme do „duhových“ barev.

Pro převod máme implementovánu pomocnou funkci na konverzi mezi barevnými modely *HSV* a *RGB*. Budeme předpokládat, že vypočtená hodnota označuje v barevném modelu HSV hodnotu *hue*. Hodnoty S a V nastavíme na jedna. Jelikož námi vypočítaná hodnota úhlu nabývá hodnot pouze v rozmezí $0 - 1$ (po úpravě absolutní hodnotou) a hodnota hue může nabývat až do hodnoty 2 pí, můžeme naši hodnotu touto konstantou vynásobit, abychom využili celé barevné spektrum.

Na závěr sečteme jednotlivé vypočítané barvy. Pořadí a správné smíchání je velice důležité a velmi ovlivňuje výsledné barevné podání. Většinou se vypočte základní model, přidá se barva lomených paprsků (průhlednost) a na to se přidá odražená barva. Můžeme se pokusit experimentovat i z jinými metodami než prosté násobení a přičítání barev. Bylo naimplementováno metod na pokročilenší skládání barev. Jedná se o funkce obsažené v souboru `mijAux` a jsou to:

- *mij_mixColors* – základní funkce na mixování dvou barev, buď průměrováním těchto dvou barev nebo se zadaným poměrem.
- *mij_mixMode_screen* – funkce na míchání barvy pomocí této rovnice nejprve invertuje obě hodnoty, vynásobí je, vydělí maximální hodnotou (v našem případě 1) a opět invertují.
$$E = 255 - \frac{(255 - M) * (255 - I)}{255}$$
- *mij_mixMode_overlay* – funkce na míchání barev pomocí této rovnice.
$$E = \frac{I}{255} * (I + \frac{2M}{255} * (255 - I))$$

Pořadí skládání barev v shaderu *jacq_sunglasses* je následující: začneme se základním osvětlením, přidáme průhlednost, poté odrazy a na závěr přidáme „duhu“ pomocí *mij_mixMode_overlay*. Výsledný shader je na obrázku 5.10.

### jacq_xray

Tento shader je NPR, má za úkol simulovat rentgenové paprsky tak, jak je známe z rentgenových snímků. Paprsky v reálu procházejí povrchem a různé materiály mají různou absorpční schopnost.

V shaderu tento absorpční jev budeme simulovat paprsky, které budeme vysílat dovnitř objektu a budeme hledat, jak daleko je další povrch. Na to se hodí funkce *mi_trace_probe*, nejrychlejší z funkcí, které mají za úkol sledovat paprsek. Nehledá žádnou barvu, jen vyšle paprsek zvoleným směrem a po nalezení průsečíku se vrátí. Je vhodná pro získání informace o tloušťce objektu. Dále využijeme funkci *mi_sample*, určenou na vzorkování. Je za potřebí vyslat paprsků více, abychom dostali měkké a plynulé přechody (podobně jako v případě měkkých stínů).

Parametry shaderu jsou:

- surfaceColor (`miColor`) - barva paprsků.
- dirtIntensity (`miScalar`) - intenzita paprsků.
- spreadAngle (`miScalar`) - úhel, ve kerém mají být paprsky vysílány.
- nearClip (`miScalar`) - udává práh pro délku paprsku. Pokud je paprsek kratší, Bude se vždy brát pln hodnota barvy paprsku.
- farClip (`miScalar`) - udává práh pro délku paprsku. Pokud je paprsek delší, je považován za "ztracený" a barva objektu je nastavena na průhlednou.
- nSamples (`miScalar`) - počet paprsků vyslaných z každého bodu.
- dirBlend (`miScalar`) - udává míru úpravy směru vysílaných paprsků.

Nejprve se upraví směr paparsků. Paprsky jsou vysílány ve směru normály povrchu, ale dovnitř objektu. Parametrem *dirBlend* se postupně upravuje směr násobením a přičítáním hodnoty vektoru $state \rightarrow dir$.

Následně se vyšlou paprsky pomocí vzorkovací funkce. Tato funkce automaticky generuje náhodné veličiny podle zadaných kritérií. Paprsek tedy upravíme a pošleme zvoleným směrem. Pokud paprsek narazí do objektu, zkontrolujeme, jestli je v mezích zvolených prahů.

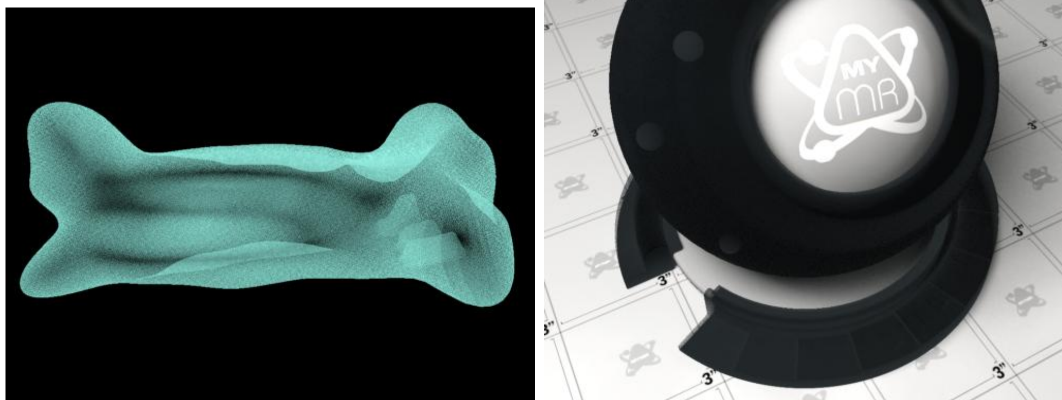
Výslednou barvu určíme pomocí mixování barvy *surfaceColor* a výsledku funkce *mi_trace_transparent*.

### 5.5.3 Ostatní shadery

#### jacq_silk

Tento shader je implementací stejnojmenného shaderu napsaného pro systém standardu RenderMan [15]. Je zároveň ukázkou, jak jsou si tyto systémy blízké, protože lze shadery přepsat z jednoho systému do druhého. Samozřejmě kód není zcela přenositelný, je potřeba jej přepsat ručně, už proto, že systém RenderMan má speciální SL, nikoliv založený na c/c++. Kromě toho RenderMan nemusí být nutně založen na metodách sledování paprsku, proto je potřeba i některé postupy řešit jinak.

Tento shader má simulovat hedvábí. Jedná se o první shader, který jsem implementoval. Má mít několik textur šumu, ale protože všechny dnešní programy v sobě procedurální texturu šumu mají implementovanou, využil jsem je. Myslím, že není potřeba implementovat něco, co je již mnohokrát (a pravděpodobně i dostatečně efektivně) hotovo a připraveno k použití.



Obrázek 5.11: Obrázky xray shaderu.

Shader má tyto parametry: Základní barvu, ambientní barvu (barvu všesměrového a všudypřítomného světla), barvu spekulárních odlesků, základní metalickou barvu a koeficient, určující rozptyl odrazů na povrchu. Samozřejmě seznam světel a potřebné parametry.

Na několika obrázcích 5.12, 5.13 z průběhu vývoje shaderu je vidět postup tvorby. Protože se jedná o první shader, při kterém jsem se teprve učil základy, byly některé výsledné obrázky zajímavé a překvapovaly i mě.

### jacq_cd

Tento shader je rovněž přepsaný z RenderMan shaderu [3]. Originální shader používá pravděpodobně anizotropní model Ashikhmin – Shirley, ale v našem shaderu je použit implementovaný anizotropní model podle Warda (*mi_ward_anisglossy*).

Tento model je upraven tak, aby vytvářel barevnou disperzi. Nejedná se o realisticky počítanou simulaci skutečné disperze, ale o úpravu spekulární reflexe. Efekt disperze lze pomocí několika parametrů upravovat. Tento shader byl vytvořen pro povrchy jako cd a dvd disky a je na tomto obrázku 5.14.

### jacq_snow

Tento shader simuluje povrch sněhu, byl inspirován letošní (opět) poměrně chudou zimou. Využívá funkce a modely implementované v systému mental ray [17], je vhodné kombinovat ho s procedurálními texturami šumu, které by lépe simulovaly povrch.

Obsahuje tyto parametry: zadní barvu pro sníh, pro paprsky dopadající pod velkým úhlem (vzhledem k normále povrchu), základní barvu, barvu pro spekulární odlesky a procedurální texturu šumu pro třpytky.

Základní a zadní barva jsou rozděleny podle koeficientu, vypočítaného funkcí *mi_fresnel_reflection*. Následně je přidáno osvětlení, pomocí Lambertova a Phongova modelu, kde se připočítává ještě spekulární barva. Na závěr je připočítána průhlednost pomocí *mi_trace_refraction* a kontroluje se délka paprsku, na základě které se výsledná barva ještě upravuje. Výsledný shader je na obrázku 5.14.

Obrázek 5.12: Shader jacq_silk.

### 5.5.4 Zprovoznění vlastního shaderu v prostředí programu Autodesk Maya

Pro testování vlastních shaderů budeme používat program Autodesk Maya PLE (Personal Learning Edition), protože je pro studijní potřeby zdarma a obsahuje systém mental ray. Tento program je zdarma ke stažení na stránkách výrobce [2]. Připojení našeho shaderu do programu Maya a systému mental ray si ukážeme na našem shaderu `jacq_constant_simple`. Předpokládejme tedy, že máme funkční kód shaderu přeložený jako dynamickou knihovnu .dll. Nyní vytvoříme deklarační soubor pro mental ray. Pro náš shader bude velmi jednoduchý:

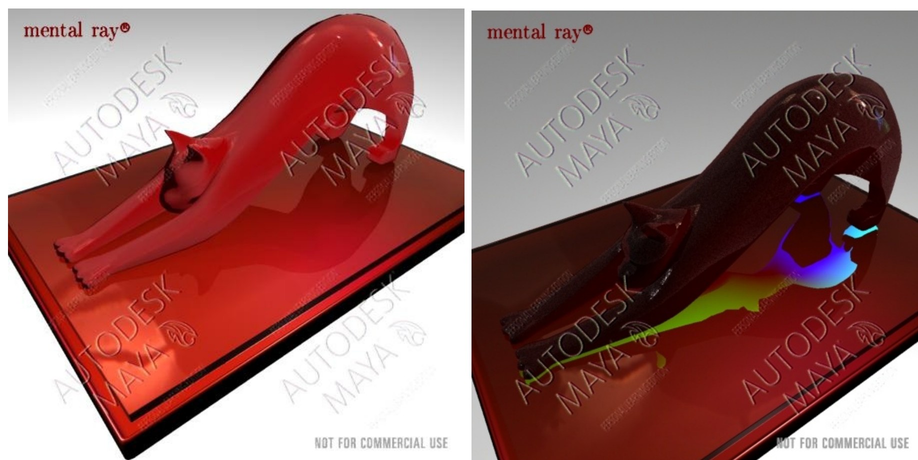
```
declare shader struct {
    color "outColor"
} "jacq_constant" (
    color "color"
) version 2 apply material end declare
```

Pro úplnost je vhodné vytvořit i mel script v prostředí Mayi. Tento skript by měl být interface mezi uživatelem a definičním souborem mental ray. Maya ale umí načíst základní parametry z definičního souboru a vytvoří interface pro shader automaticky. Takže prozatím se tvorbou mel skriptu zabývat nebudeme. Soubor se zkompilovaným shaderem (dll) a definiční soubor (mi) zkopírujeme do příslušných adresářů systému mental ray v programu Maya [14].

```
"maya_adresář"/mentalray/lib "maya_adresář"/mentalray/include
```

Do souboru *maya.rayrc* který se nachází v *"maya adresář"/mentalray/maya.rayrc* je potřeba přidat odkazy na definiční soubor a shader. Shader je uvozován návěstím *link*:

```
link "{MAYABASE}/lib/constant simple.dll"
```




Obrázek 5.13: Obrázky z průběhu vývoje prvního shaderu.

případně využít definované koncovky {DSO}.
Definiční soubor začíná návěstím *mi*:

```
mi "{MAYABASE}/include/constant_simple.mi"
```

**Použití v programu Maya**

Po otevření programu Maya zkontrolujeme, jestli se náš shader načetl.

- V hlavním menu nahoře vybereme Window → Rendering Editors → *Hypershade*.
- V okně *Hypershade* zobrazíme *mental ray nodes*. V záložce Create nahoře zvolíme místo *Create Maya Nodes* → *Create mental ray Nodes*. Pokud tato volba chybí, je třeba zapnout mental ray. Nahoře v menu Render → Render Using → mental ray.
- Rozbalíme Menu Materials a vybereme náš materiál (constant simple)
- Materiál se automaticky přidá do scény i do *work area*.

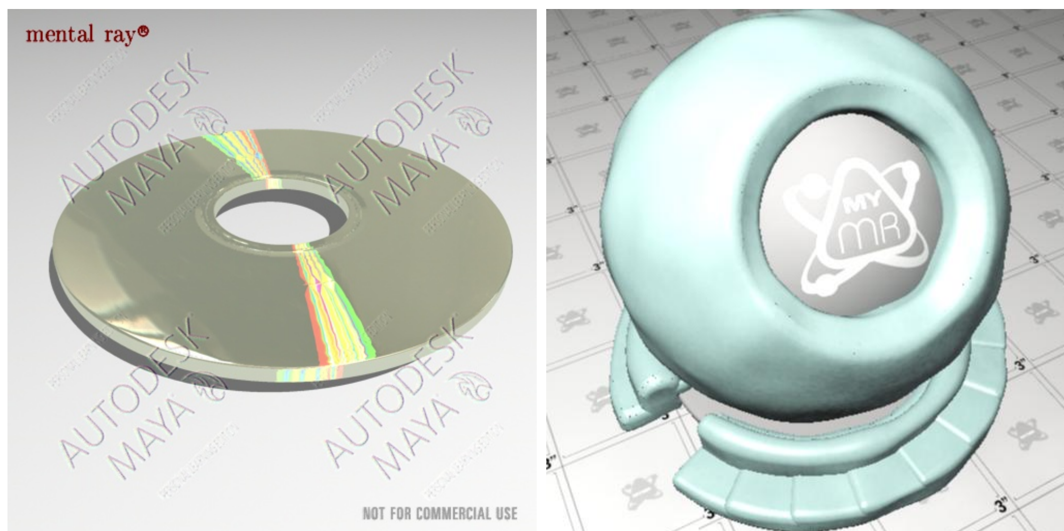
Otevřeme (nebo vytvoříme) libovolnou scénu a označíme některý z objektů.

- Přepneme se do okna Hypershade a ve work area klikneme pravým tlačítkem na shader a táhneme myší směrem nahoru (je důležité kliknout rovnou pravým tlačítkem, jinak "odznačíme" označený objekt ve scéně).
- Objeví se kontextové menu, ve kterém zvolíme možnost *Assign material to selection*.

Vyrenderujeme scénu a zkontrolujeme, jestli shader funguje. Můžeme upravovat parametry shaderu (jeho barvu). Buď v prostředí Hypershade nebo ve vlastnostech objektu.

### 5.5.5 Rychlost shaderu při různých operacích

Měření rychlosti proběhlo na konfiguraci intel core2duo T5300, 3GB RAM na 667 MHz (vliv grafické karty je zanedbatelný). Aby bylo měření úplně objektivní, bylo by potřeba




Obrázek 5.14: Vlevo je shader jacq_cd, vpravo jacq_snow.

průměrovat naměřené hodnoty například počtem volání konkrétního shaderu. Protože rychlost závisí na tom, kolikrát se daná operace volá, tedy jak velkou část obrazu objekt s daným shaderem zabírá. Ale i přesto podává měření důležité informace o tom, jak je vhodné shadery psát. Během vykreslování byl zobrazen framebuffer. Testovací scénka obsahovala jeden objekt, kouli a měla rozlišení 2048 x 1556. Měření probíhala na různých úrovní počtu vzorků:

- podvzorkování 1/4
- bez vzorkování (1)
- vzorkování 4
- vzorkování 16
- vzorkování 64

### Měření rychlosti načítání parametrů shaderu

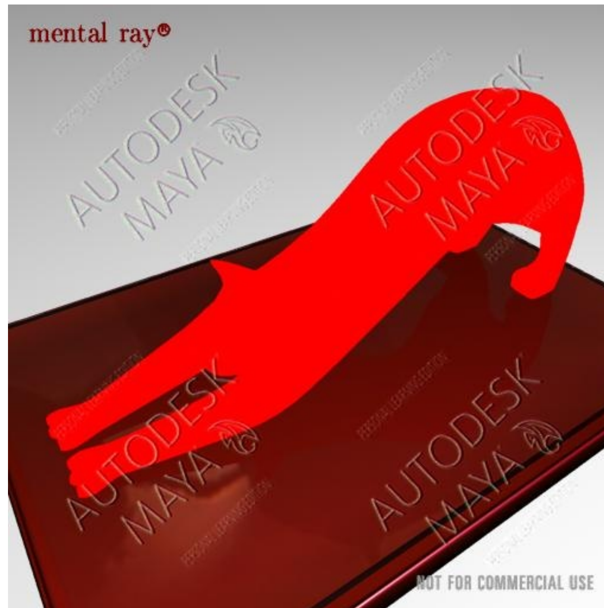
Podle manuálu systému mental ray je načítání pomocí *mi_eval* dostatečně efektivní a parametr ze struktury se načítá pouze při prvním volání shaderu. V rámci měření byly vytvořeny dva jednoduché shadery. V prvním případě se do proměnných přímo načítaly hodnoty (konstanty). V druhém případě se do proměnných načítala data z parametrů shaderu. Výsledky jsou v grafu 5.16

Z naměřených výsledků je patrné, že je opravdu vhodné používat funkci *mi_eval*, rozdíl v použití funkce a konstanty je minimální, neměřitelý.

### Rozdíl mezi trace transparent a trace refraction

Podle manuálu je funkce *trace_transparent* rychlejší než *trace_refraction*. 5.17

Z výsledků je patrné, že jsou tyto funkce skutečně různě složité. Když ještě připočteme fakt, že pro refraction je potřeba vypočítat i směr paprsku, bude v reálném shaderu rozdíl ještě viditelnější (pro transparent většinou použijeme parametr dir z proměnné state).




Obrázek 5.15: Výsledný shader s červenou barvou.

### Rozdíl v přístupu k proměnným

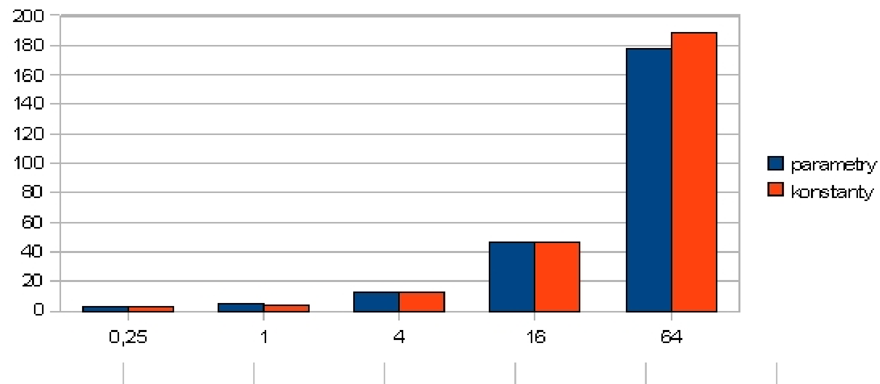
.

V prvním případě načteme *state* → *dir* nejprve do proměnné a tu poté používáme, v druhém budeme počítat rovnou s strukturou *state*. Výpočet je volán ve smyčce 5 krát. 5.18

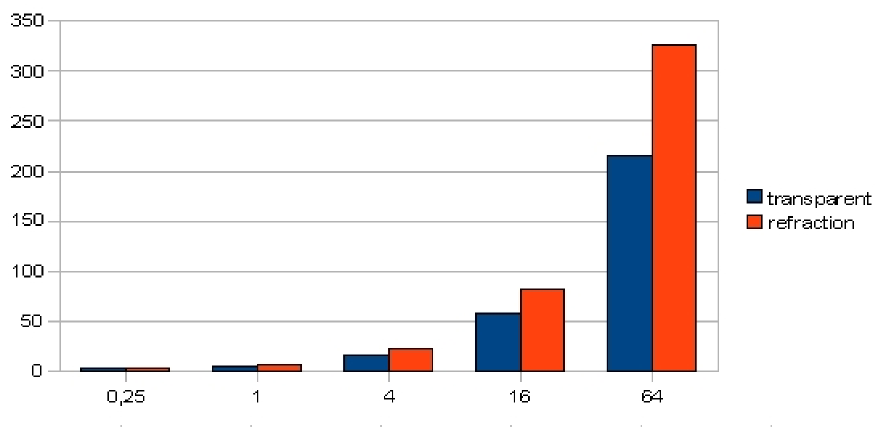
Z výsledků měření je patrné, že není velký rozdíl mezi přístupy k proměnným a k proměnným načítaným ze struktury. Možná by se nějaké rozdíly objevily při větším množství iterací.

### Měření rychlosti při různých úrovních samplování scény

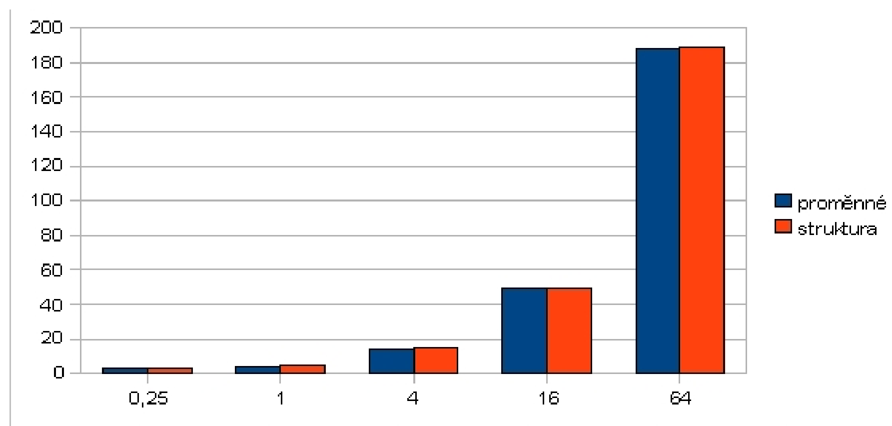
Na závěr graf ukazující rostoucí náročnost výpočtu scény při zvyšování počtu vzorků. Vizuální rozdíl při hodnotách vzorkování 16 a 64 je minimální a čas potřebný k výpočtu roste nelineárně. 5.19



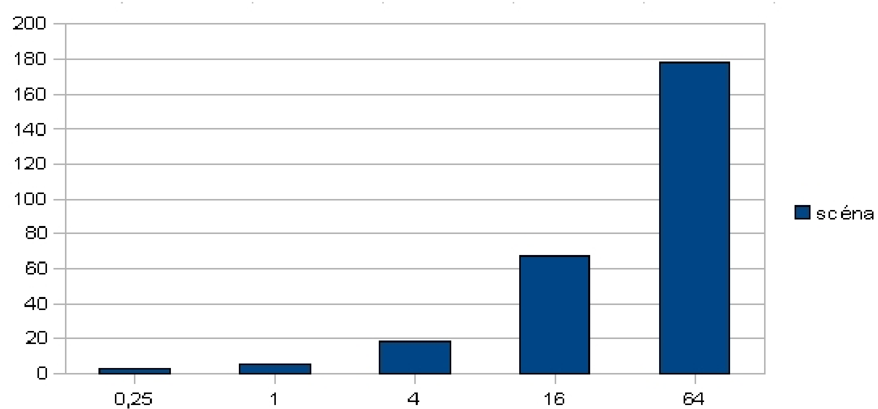

Obrázek 5.16: Graf ukazující rozdíl v rychlosti načítání parametrů a konstant. Na ose X je počet vzorků, na ose Y čas v sekundách.




Obrázek 5.17: Graf ukazující rozdíl v rychlosti trace_transparent a trace_refraction. Na ose X je počet vzorků, na ose Y čas v sekundách.




Obrázek 5.18: Graf ukazující rozdíl v rychlosti různých přístupů k proměnným. Na ose X je počet vzorků, na ose Y čas v sekundách.




Obrázek 5.19: Graf ukazující nárust výpočetního času při zvyšujícím se počtu vzorků. Na ose X je počet vzorků, na ose Y čas v sekundách.

# Kapitola 6

# Závěr

V rámci diplomové práce byl vytvořen text, který je učební pomůckou pro všechny, kteří mají zájem naučit se pracovat se systémem mental ray. Kromě praktického popisu systému a návodu na tvorbu shaderů byly zpracovány i vybrané a důležité kapitoly z oblasti počítačové grafiky, zejména pak metod vykreslování a osvětlování scény. Tyto podklady jsou důležité pro hlubší pochopení systému mental ray a dávají čtenáři určitý rozhled v problematice této oblasti počítačové grafiky. Stěžejní částí je však popis systému mental ray, zejména části shading language a návod na jeho používání, doplněný o detailně komentované postupy, které se při tvorbě vlastních shaderů používají.

V rámci této práce bylo naimplementováno několik shaderů pro systém mental ray. Některé jsou v textu rozebírány podrobněji, všechny jsou však podrobně komentovány, takže je lze rovněž využít jako doplňkovou učební pomůcku. Součástí je rovněž návod na zprovoznění a otestování vlastních shaderů v prostředí programu Autodesk Maya PLE 2008.

Čtenář by se přečtením této práce měl naučit ovládat shading language systému mental ray, měl by se naučit psát pro tento systém shadery a měl by je umět otestovat v prostředí programu Autodesk Maya.

Obtížné bylo ladění shaderů, protože v programu Maya nebylo možno kontrolovat kód jinak, než kontrolními výpisy pomocí funkce *mi_warning*. Dále program neumožňoval vypnutí a zapnutí samostatného modulu mental ray pro výměnu knihovny se shaderem za novější verzi, takže bylo potřeba celý program vypnout, přehrát novou verzi shaderu, program zapnout, načíst scénu se shaderem a znovu spustit výpočet obrazu. Tento postup velmi zdržoval při tvorbě shaderů. Hledání chyby v běhu shaderu bylo velmi problematické. Nebylo ale nalezeno žádné jiné řešení tohoto problému.

Nad rámec zadání byla přidána část s měřením rychlosti jednotlivých shaderů a rychlosti možných konstrukcí shaderů. Rovněž byly shadery přidány do programu 3ds max 9 (trial verze), ve kterém byly vyrenderovány některá videa lépe demonstrující činnost implementovaných shaderů. V tomto textu je popsána pouze část systému mental ray, takže práce a možností pokračování je ještě mnoho. Další cestou v této práci by jistě bylo implementovat a popsat některé jiné typy shaderů, které mental ray podporuje. V tomto směru je zřejmě možnost pokračování největší, ale nejstereotypnější.

V tomto textu je rozebrán pouze typ shaderu material, což je hlavní shader, který definuje základní vlastnosti povrchu. Je potřeba rozepsat i ostatní shadery, například pro výpočet globálního osvětlení a popsat vazby a práci mezi nimi.

Rovněž by bylo zajímavé srovnat přístupy k psaní shaderů: buď komplexní hotový shader určený na simulaci konkrétního povrchu (případně jiného jevu), nebo shader složený z mnoha základních elementárních shaderů a jako výsledek porovnat flexibilitu a rychlost

těchto přístupů.

Dalším možným pokračováním této práce by bylo vytvoření jakéhosi WYSIWYG editoru s rozhraním pro psaní shaderů, který by obsahoval hotové jednotlivé konstrukce nebo elementární shadery, umožňoval by jejich spojování do shader tree a vykresloval by náhled na výsledný shader. Tento program by však potřeboval pravděpodobně stand-alone verzi systému mental ray.

Při psaní shaderů a učení se se systémem mental ray bylo čerpáno mimo jiné z těchto webových stránek: [9], [13], [4], [17], [5].

# Kapitola 7

# Slovníček

- Rendering – technika syntézy obrazu, ve které se na základě trojrozměrné reprezentace scény vytváří její dvourozměrná projekce. Většinou se jedná o obrázek, uzpůsobený pro prohlížení na obrazovce nebo papíře, proto je většinou cílem této metody vytvořit dojem co nejrealističtějšího vzhledu scény [21].
- Renderer – program vykreslující scénu.
- Ray tracing - technika vytváření obrazu, ve které se do scény vysílají paprsky a sleduje jejich dráha a na základě těchto informací je výsledný obraz vykreslen.
- Shader – podprogram, dynamická knihovna, která rozšiřuje či doplňuje funkce programu. V systému mental ray se pomocí shaderu popisují vlastnosti povrchů, jejich chování při interakci s různými druhy paprsků.
- Pixel – nejmenší bod obrazu, nese barevnou informaci a každý má svou polohu (x, y) v obraze.
- Aliasing – jev, ke kterému dochází při převodu spojití informace na diskrétní, a kdy není splněna podmínka vzorkování (vzorkovací frekvence musí být minimálně 2x větší než maximální frekvence v obraze).
- Scene description language – jazyk pro popis scény. Každý renderer má svůj jazyk, kterým popisuje 3D scénu, určenou k vykreslení.
- Supersampling - technika potlačující aliasing, která vzorkuje obraz a výslednou barvu pixelu určuje průměrováním okolních vzorků.
- Jittering - jev zubatých hran, vznikajících při renderingu v důsledku aliasingu.
- Voxel - nejmenší bod ve 3D scéně, kreprezentované pravidelnou mřížkou. Jedná se o objemovou jednotku (krychle). Je podobný jako pixel ve 2D obraze.
- Radiozita - metoda globálního osvětlování scény, používá se pro výpočet osvětlení, je založena na zákoně zachování energie.
- Bump - bump mapping, technika texturování, vytváří dojem nerovnosti povrchu tím, že upravuje normálu v daním bodě, čímž ovlivňuje výpočty. Geometrii modelu nemění.
- Displace - displacement mapping, technika podobná bump mapping, ale posouvá vertex po své normále (mění geometrii). Často obsahuje algoritmy na dělení plochy.

- Shader tree - strom (graf) shaderu, určující jak se jednotlivé složky a operace skládají do výsledné barvy (případně jiné hodnoty, kterou shader počítá).
- SSS - sub surface scattering, šíření světla pod povrchem objektu. Tento jev se vyskytuje například u lidské kůže (obecně u měkkých tkání).
- NPR - non photo realistic, znamená, že se shader (výsledný obraz) nesnaží napodobit jev v reálném světě.
- Phenomena - jev, fyzikální úkaz.
- Teselace - proces, pomocí kterého se obecný polygon převádí na nepravidelnou trojúhelníkovou síť (TIN - triangular irregular network).

# Literatura


[1] animusartis. [online], rev. 5/10/2008, [cit. 2008-05-10].

[2] Autodesk - autodesk maya. [online], rev. 12/30/2007, [cit. 2007-12-30].

[3] Cd specular 0.0. [online], rev. 5/10/2008, [cit. 2008-05-10].

[4] Fuzzyphoton. [online], rev. 5/10/2008, [cit. 2008-05-10].

[5] German mental ray forum. [online], rev. 5/10/2008, [cit. 2008-05-10].

[6] Kadi bouatouch. [online], rev. 5/10/2008, [cit. 2008-05-10].

[7] Mental ray, mental images. [online], rev. 1/2/2008, [cit. 2008-01-02].

[8] Mental ray resources. [online], rev. 12/30/2007, [cit. 2007-12-30].

[9] My-mental ray. [online], rev. 5/10/2008, [cit. 2008-05-10].

[10] Neil blevins. [online], rev. 5/10/2008, [cit. 2008-05-10].

[11] Perret opticans color vision. [online], rev. 12/30/2007, [cit. 2007-12-30].

[12] Shader writing. [online], rev. 12/30/2007, [cit. 2007-12-30].

[13] trixr4kids.com. [online], rev. 5/10/2008, [cit. 2008-05-10].

[14] Tutorial: Writing a simple shader extension for mental ray, xsi. [online], rev. 5/10/2008, [cit. 2008-05-10].

[15] vm_sparking_silk_band - shaders for renderman. [online], rev. 12/30/2007, [cit. 2007-12-30].

[16] Wapedia - wiki: Reyes rendering. [online], rev. 5/10/2008, [cit. 2008-05-10].

[17] Writing mental ray shaders. [online], rev. 5/10/2008, [cit. 2008-05-10].

[18] M WATT ALAN WATT. *Advanced Animation and Rendering Techniques.* Addison-Wesley Professional, USA, 1 edition, 1992.

[19] DARIUSH DERAKHSHANI. *Maya průvodce 3D grafikou.* Grada Publishing, CZ, 1 edition, 2006.

[20] PHILIP DUTRÉ and KAVITA BALA PHILIPPE BEKAERT. *Advanced Global Illumination.* A K PETERS LTD., USA, 1 edition, 2003.

[21] A GLASSNER. *An Introduction To Ray Tracing*. Morgan Kaufmann Publishers, USA, 2 edition, 1989.

[22] MARTIN KRAHULÍK. *Reprezentace BRDF v počítačové grafice*. ČVUT FEL, Praha, 1 edition, 2005.

[23] R K MÓRLEY P SHIRLEY. *Realistic Ray Tracing*. A K Peters Ltd, USA, 2 edition, 2003.

# Dodatek A

# Obsah přiloženého DVD

Přiložené DVD obsahuje:

- Soubor `readme.txt`, ve kterém jsou všechny potřebné informace, názvy jednotlivých souborů nebo návody na spuštění programů.
- Textovou zprávu ve formátu `pdf`.
- Plakát k práci v tiskovém rozlišení.
- Komentované zdrojové soubory všech vytvořených shaderů, včetně projektu pro Visual Studio 2005.
- Manuál systému mental ray (tak jak je přiložen k programu Autodesk Maya PLE).
- Program Autodesk Maya PLE 2008 (vždy poslední verze je ke stažení na stránkách výrobce, aktuálně se jedná o verzi 2009).
- Zdrojové soubory všech scén z programů Maya a 3ds max, vytvořených pro potřeby této diplomové práce.

# Dodatek B

# Ukázky vytvořených shaderů