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

# LIGHT PROPAGATION VOLUMES

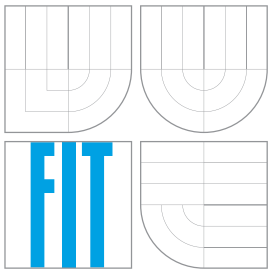
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ MIKULICA

BRNO 2015

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

# LIGHT PROPAGATION VOLUMES

LIGHT PROPAGATION VOLUMES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ MIKULICA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JOZEF KOBRTEK

BRNO 2015

# Abstrakt

Tato práce se zabývá problémem výpočtu globálního osvětlení v reálném čase a jsou v ní popsány dvě metody. Reflective Shadow Maps a Light Propagation Volumes. První z nich řeší daný problém rozšířením Shadow mapping algoritmu. Zatímco druhá pokryje scénu 3D mřížkou a za použití sférických harmonických funkcí spočítá šíření světla ve scéně. Dále tato práce obsahuje výsledky měření rychlosti vykreslení algoritmu Light Propagation Volumes stejně jako vyhodnocení vizuální kvality výstupu.



# Abstract

This thesis deals with problem of computation of global illumination in real-time. Two methods are described. Namely Reflective Shadow Maps and Light Propagation Volumes. The first of them deals with the problem by using extended Shadow Mapping algorithm. The second one uses scene embedded into a 3D grid together with Spherical harmonics to compute light propagation in the scene. Furthermore this thesis contains results of measurement of the rendering speed of the Light Propagation Volumes algorithm with various settings on several machines. Quality of the resulting output of the algorithm is also evaluated.



# Klíčová slova

OpenGL, Globální osvětlení, Reflective Shadow Maps, Light Propagation Volumes, Sférické harmonické funkce

# Keywords

OpenGL, Global Illumination, Reflective Shadow Maps, Light Propagation Volumes, Spherical harmonics


# Citace

Tomáš Mikulica: Light Propagation Volumes, diplomová práce, Brno, FIT VUT v Brně, 2015

# Light Propagation Volumes

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jozefa Kobrtka.

.....
Tomáš Mikulica
26. května 2015

## Poděkování

Rád bych poděkoval svému vedoucímu Ing. Jozefu Kobrtkovi za náměty a připomínky, které mi pomohly dovést tuto práce do konce. Dále bych tímto rád poděkoval Ing. Tomáši Miletovi za cenné rady ohledně samotné implementace práce. A v neposlední řadě bych rád poděkoval svým spolužákům za pomoc při testování práce a své rodině a přítelkyni za trpělivost.

Věnováno Bc. Igorovi Pavlů.

# Obsah

# Seznam obrázků

# Seznam tabulek

# Kapitola 1

# Úvod

Počítačová grafika ušla od svým začátků pořádný kus cesty. Mezitím se výkon počítačů obrovsky zvětšil. Tak se není čemu divit, že s tím výkonem chceme nějak naložit. Počítačová grafika může být krásným příkladem. Je snaha udělat zobrazení scény, co možná nejrealističtější a pokud možno, aby to běželo v reálném čase.

Metody globálního osvětlení určitě patří do kategorie těch, co přidávají zobrazované scéně na uvěřitelnosti zobrazení. Nutno říct, že i přes všechen ten výkon, co máme k dispozici se jedná o poměrně nesnadnou úlohu a to vzhledem ke komplexní povaze světla. V této oblasti probíhá stále mnoho výzkumů s cílem najít způsob, jak počítat globální osvětlení v nejkratším čase za použití co nejméně prostředků. Existuje poměrně dost metod, které se snaží s tímto problémem vypořádat.

V této práci jsou popsány dva algoritmy. Jeden z nich je založen na šíření světla v mřížce a jmenuje se *Light Propagation Volumes*. Druhý algoritmus patří do kategorie algoritmů, které pracují ve *screen–space*, a jedná se konkrétně o *Reflective Shadow Maps*. Tento algoritmus je využit hlavně pro jeden z kroků inicializace *Light Propagation Volumes*.

V kapitole 2 je stručně představena knihovna *OpenGL* a její historie. Mimo to je v této kapitole také představen jazyk pro programování shaderů *GLSL*. Následuje kapitola 3 popisující stručně problém globálního osvětlení. Kapitoly 4 a 5 se věnují samotným algoritmům pro výpočet nepřímého osvětlení. Konkrétně v kapitole 4 je popsán algoritmus *Reflective Shadow Maps* a v kapitole 5 algortimus *Light Propagation Volumes*. Dále jsou v kapitole 6 popsány problémy a nejastnosti, které vyvstaly během samotné implementace algoritmu, a způsob jejich řešení. Mimo to tato kapitola obsahuje stručný návrh aplikace (to zejména z hlediska funcionality a použitých knihoven). Následuje kapitola 7, kde jsou prezentovány výsledky měření výkonu algoritmu spolu s vizuálním porovnáním. Ná závěr si shrneme dosažené výsledky a možné pokračování projektu.

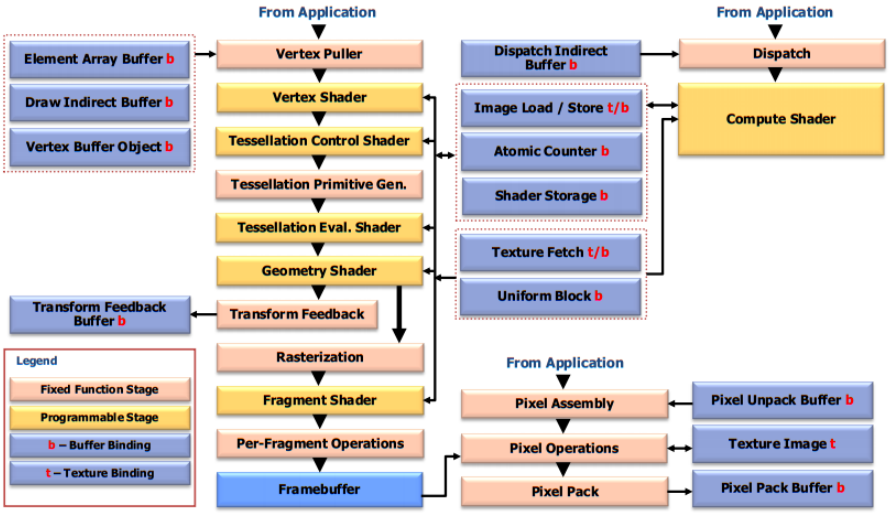
# Kapitola 2

# Open Graphic Library

*Open Graphic Library* (ve zkratce *OpenGL*) je multiplatofrmní knihovna, která definuje aplikační rozhraní (API) pro programování počítačové grafiky na grafickém hardwaru[15]. Toto rozhraní se skládá z množiny několika stovek procedur a funkcí, které umožňují programátorovi specifikovat shader programy (nebo zkráceně shadery), objekty a operace, které jsou zahrnuty v procesu vytváření grafického obrazu. *OpenGL* je používáno pro hry, CAD systémy, různé věděcké aplikace a mnoho dalšího.

## 2.1 Stručná historie

První verze *OpenGL* byla vydána společností *Silicon Graphics Inc.* v roce 1992. Jako základ posloužila proprietarní knihovna *Iris GL*. Od svého prvního vydání prošla knihovna relativně velkými změnami. K těm výrazným změnám určitě patří s příchodem technologie *Transform and Lighting* (zkráceně *T&L*) přechod z fixního zobrazovacího řetězce k programovatelnému řetězci (jeho aktuální verze je znázorněna na obrázku 2.1) ve verzi 2.0 a s tím spojené oznámení jazyka *OpenGL Shading Language*. Ovšem i přes tyto změny knihovna pořád zaostávala za knihovnou *DirectX* od firmy *Microsoft* a to především díky tomu, že se vývojáři snažili pořád držet zpětnou kompatibilitu s předchozími verzemi. To se ovšem změnilo ve verzi 3.0, kdy byl zavedený tzv. *deprecation model* – věci které budou v příštích verzích odstraněny. Dále byla ve verzi 3.0 přidána "odbočka" do vykreslovacího řětezce tzv. *Transform Feedback*, která umožňovala uložit výsledek z přechozích fází do bufferu místo toho, aby byl rasterizován. Jako další významný bod bych uvedl přidání dalšího stupně do programovatelného vyskreslovacího řetězce ve formě geometry shaderů. Ve verzi 3.3 se objevila podpora tzv. *timer queries*, které umožňují snadné a přesné měření času stráveného výpočty na GPU. Verze 4.0 opět přinesla nový stupeň do vykreslovacího řetězce ve formě teselačních shaderů. Vykreslovací řetězec se následně ještě obohatil ve verzi 4.3 o compute shadery. *OpenGL* funguje jako stavový stroj a dosud tomu bylo tak, že nějaký stav se musel aktualizovat pomocí mechanismu výběru stavu, což způsobovalo v komplexnějších aplikacích velkou režii. Řešení přišlo ve formě *Direct State Access*, které umožňuje tyto stavy měnit přímo bez použití selektoru. Tato funkcionalita byla zavedena ve aktuální verzi 4.5. Nicméně byla dostupná ve formě rozšíření i ve verzi 4.4.




Obrázek 2.1: Vykreslovací řetězec. Obrázek převzat z [9].

## 2.2 OpenGL Shading Language

S příchodem *OpenGL* verze 2.0 byla představena i první verze jazyka pro programování shaderů – *OpenGL Shading Language* nebo zkráceně *GLSL*[9]. Ten vychází z programovacího jazyka C, který prošel jistými úpravami jako například odstranění ukazatelů a dalšími. Je samozřejmé, že i verze toho jazyka se vyvíjela s rostoucí verzí *OpenGL*. Aktuální verze je pak 450 (květen 2015).

## 2.3 Vykreslovací řetězec

Nyní si ve zkratce popíšeme některé důležité části (programovatelné) vykreslovacího řetězce. Ten se skládá z mnoha kroků. Uveďme si alespoň následující:

- `Vertex Shader` – Tento shader pracovává jednotlivé vrcholy. Typicky se zde provádí transformace vrcholů.
- `Tessellation Control Shader` – Pracuje s novým typem primitiv – záplatou (*patch* – `GL_PATCHES`) a určují se zde parametry samotné teselace (jak moc teselovat) – míra vnitřní a vnější teselace.
- `Tessellation Primitive Gen.` – Vytváří nové primitiva ze vstupní záplaty. Není programovatelný.
- `Tessellation Evaluation Shader` – Provádí převod abstrakních souřadnic vypočtených v předchozím kroku do souřadnic jednotlivých nových vrcholů.
- `Geometry Shader` – Umožňuje vytvářet nová primitiva.
- `Fragment Shader` – Zpracovává jednotlivé pixely po rasterizaci a určuje výstupní barvu pixelu a umožňuje tvorbu různých efektů jako např. *bumb mapping*, ale také umožňuje vytvářet *per-pixel* osvětlení.

# Kapitola 3

# Globální osvětlení

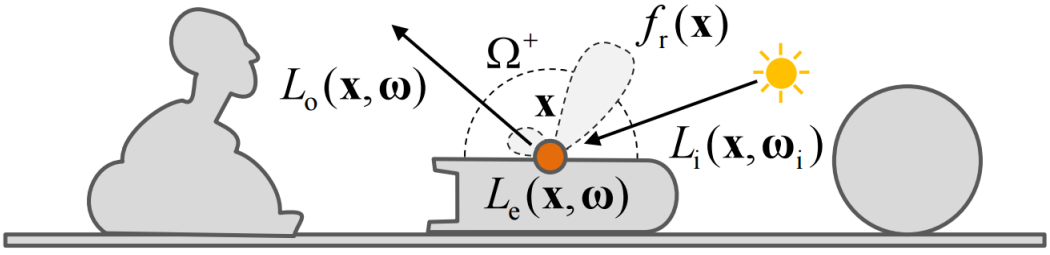
Globální osvětlení je založeno na fyzikálně přesném popisu šíření světla, což samozřejmě zvyšuje realičnost zobrazovaný scény. Algoritmy pro globální osvětlení berou v potaz jak přímé osvětlení, které přichází od světelného zdroje (*direct illumination*), tak osvětlení, které vzniká odrazem od povrchů ve scéně (*indirect illumination*). Veškeré dění se točí okolo řešení zobrazovací rovnice [14, 5] (zobrazená na obrázku 3.1):

$$L_o(x,\omega) \quad = \quad L_e(x,\omega) + L_r(x,\omega) \tag{3.1}$$

Kde $L_o(x,\omega)$ je výsledná zář (*radiance*) v bodě $x$ odražená směrem $\omega$ k pozorovateli, $L_e(x,\omega)$ je zář vyzařovaná z povrchu výsledného bodu a $L_r(x,\omega)$ je odražená zář, která je definována následovně:

$$L_r(x,\omega \quad = \quad \int_{\Omega^+} L_i\left(x,\omega_i\right) f_r\left(x,\omega_i \rightarrow \omega\right) \left\langle N(x),\omega_i \right\rangle_+ d\omega_i \tag{3.2}$$

Kde $\omega^+$ je horní hemisféra orientovaná okolo normály povrchu $N(x)$ v bodě $x$, $f_r\left(x,\omega_i \rightarrow \omega\right)$ je obousměrná distribuční funkce odrazu světla (*BRDF – Bidirectional Reflectance Distribution Function*) a $\langle|\rangle_+$ je skalární součin se zápornými čísly nahrazenými nulou.




Obrázek 3.1: Ilustrace zobrazovací rovnice. Obrázek převzat z [14].

Ačkoliv je tato rovnice už nějaký čas známa, tak její řešení je časově náročné i na současném hardwaru. Různé metody byly navrženy pro řešení určitých jevů. Některé z metod se zaměřují na možná co nejvíce cest fotonů, které jsou schopné aproximovat, a jiné zase na různé typy jevů (například kaustika), které mohou případně simulovat. Zobrazovací rovnici lze značně zjednodušit tím, že vynecháme určité jevy. Příkladem může být to, že například

omezíme povrchy, se kterými budeme počítat, jen na difuzní. Nebo tím, že omezíme počet odrazů na nějaké malé fixní číslo. Obě tyto omezení jsou případ právě algoritmu *Light Propagation Volumes* popsaného v kapitole 5.

Pokud bychom chtěli metody nějak rozškálovat, tak nejhrubější dělění metod by nejspíše bylo:

- Offline metody
- Interaktivní metody

## 3.1 Offline metody

Tyto metody typicky poskytují co možná nejvyšší kvalitu, ale to za cenu výpočetního času, který se může pohybovat v řádu minut pro jednoduché scény až řádu hodin/dnů v případě složitějších scén (či simulovaných jevů ve scéně). Tyto metody typicky simulují lom a šíření světla ve scéně. Mezi zástupce takovýchto metod se řadí metody sledování paprsku, metody sledování cest či photon mapping metoda. Některé z těchto metod se dají při aplikování jistých omezení a použití různých struktur pro akceleraci výpočtu provádět interaktivně.

## 3.2 Interaktivní metody

Výše uvedená skupina metod se hodí zejména na vizualizace či propagační materiály, kde výsledná kvalita musí být velká a na výpočetní čas se až tak nehledí. Situace je ovšem jiná v případě interaktivního zobrazování či v počítačových hrách, kde se počítá každá milisekunda či každý snímek k dobru. Vzhledem k tomu, že v této oblasti bylo provedeno mnoho výzkumů a stále se provádí další, tak popis všech algoritmů je mimo rozsah této práce. Nicméně některé vybrané metody si nyní v krátkosti představíme.

### 3.2.1 Ambient occlusion

Pokud se bavíme o interaktivních metodách zastnínění okolím (*ambient occlusion*), tak se bavíme pouze o jejich aproximaci. Principem metod je, že snažíme v každém místě povrchu modelu spočítat množství světla, které na toto místo může dopadanout. Děje se tak pomocí vysílání paprsků do polokoule okolo bodu na povrchu a počítání poměru paprsků, které protnuli geometrii a které nikoliv. Výsledkem je informace o tom, jak je dané místo zastíněno. U těchto metod se rozlišuje v jakém prostoru pracují:

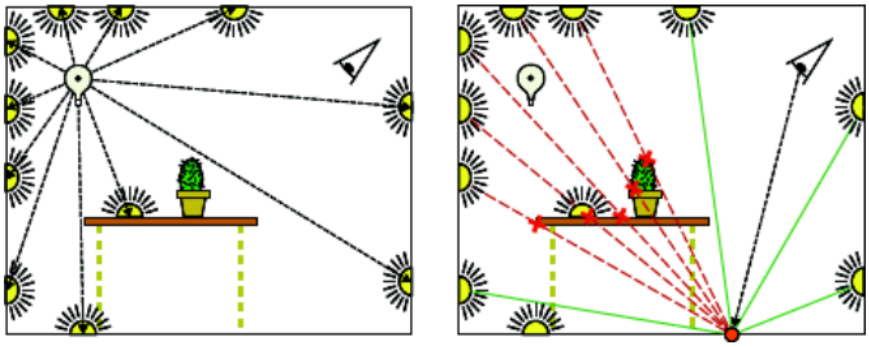
- *Object–space* – tyto metody poskytují stabilní výstup relativně vysoké kvality ovšem jsou závislé na geometrii scény a tudíž jsou i výpočetně náročnější. Příkladem může být metoda, kterou představil Michael Bunnel [1]. Jedná se o dvou průchodový algoritmus, který počítá zastínění okolím pro každý vrchol (*per vertex*), s tím, že geometrii reprezentuje jako množinu kruhů, které mají střed právě ve vrcholech. Výsledné zastínění je pak počítáno analyticky.
- *Screen–space* – výhodou těchto technik je, že jsou nezávislé na geometrii. Příkladem může být metoda *Screen space ambient occlussion*[12], kdy se berou pro bod pixely v jeho okolí a pomocí porovnání na hloubku se spočte zastínění pixelu (viz obrázek 3.2).

Obrázek 3.2: Výstup metody *Screen space ambient occlussion*.

### 3.2.2 Instant radiosity

Původní metodu *instant radiosity* (IR) představil v roce 1997 Alexander Keller[8] a tím dal vzniknout celé rodině algoritmů, které aproximují nepřímé osvětlení za pomoci *Virtual Point Lights* (VPL). Mimo jiné mezi ně patří i algoritmus *Reflective Shadow Maps*, který je popsán dále v kapitole 4.

Původní metoda pracovala na pouze na CPU a skládá se ze dvou kroků. V prvním kroku se vytvoří cesty fotonů za pomocí algoritmu náhodné procházky. V každém průsečíku s geometrií je pak vytvořen VPL, který reprezentuje světelný tok vycházející z tohoto bodu. Ve druhém kroku se scéna osvětlí z právě vzniklých VPL pomocí techniky shadow mapping. Každý vystínovaný bod se pak projdou všechny VPL a kontroluje se, zda je daný VPL aktuálním zdrojem světla. Tím docílíme toho, že je integrál ve vztahu 3.2 nahrazen přímým osvětlením z VPL, které bod osvětlují (viz obrázek 3.3). Nevyhodou toto přístupu je, že potřebujeme velký počet VPL k tomu, abychom konvergovali ke správnému řešení. Tím, že zvýšíme počet VPL, také pochopitelně negativně ovlivníme výkon.

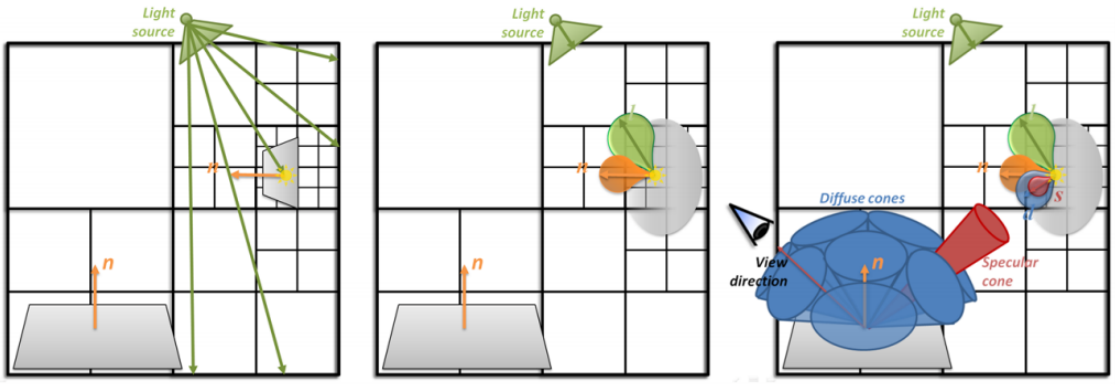


Obrázek 3.3: Princip metody IR. Obrázek převzat z [11].

### 3.2.3 Voxel Cone Tracing

Tento algoritmus představil Cyril Crassin[3] v roce 2011. Jedná se o algoritmus, který umožňuje spočítat i druhý odraz nepřímého osvětlení a je založený na voxelizaci scény (rozdělení scény na voxely – volumetrické pixely) a uložení této informace do hierarchické struktury (*sparse voxel octree*). Samotné voxely obsahují informace jako barva, optická hustota... Tento algoritmus se skládá ze několika kroků (viz obrázek 3.4), které je nutné provést pro každý snímek. Kromě těchto kroků je nutný ještě jeden krok a to proces voxelizace scény (popsaný v [2]), který stačí pro statickou geometrii spočítat pouze jednou. Případná dynamická geometrie je přidávána do struktury za běhu.

V prvním kroku se vloží do listů octree stomu voxelů dopadající zář – *radiance* (energie a směr), čímž vytvoříme hierarchii světla. Toho se docílí pomocí rasterizace scény ze všech zdrojů světla a následného spočítání odražené (difuzní) záře. Ve druhém kroku se provede filtrace záře z listů do vyšších úrovní octree stromu. Posledním krokem je vykreslení scény z pohledu kamery a pro každý viditelný fragment se zkombinuje přímé a nepřímé osvětlení. To se provede tak, že se pro každý viditelný fragment se vrhne několik kuželů a pomocí nich se posbírají voxely z vyšších úrovní, na které jsme narazili.




Obrázek 3.4: Ilustrace algoritmu *Voxel Cone Trancing*. Obrázek převzat z [3].

# Kapitola 4

# Reflective Shadow Maps

Algoritmus *Reflective Shadow Maps* (dále jen RSM) byl představen [4] a dalo by se říct, že rozšiřuje *Shadow mapping* o nepřímé osvětlení (pouze o první odraz světla). Myšlenka celého algoritmu spočívá v tom, že všechny pixely uložené ve stínové mapě chápeme jako nepřímé zdroje osvětlení (uvažuje se pouze jeden odraz nepřímého osvětlení) ve scéně.

## 4.1 Uložená data, jejich vytváření a vyhodnocení

RSM je vytvářena velmi podobně jako standardní stínová mapa s tím rozdílem, že se pro každý pixel $p$ uchovávají nějaké informace navíc. Konkrétně se pro každý pixel $p$ uchovává informace o hloubce, pozice $x_p$ ve světových souřadnicích (*World space*), normála $n_p$ a odražený zářivý tok $\Phi_p$ (*reflected radiant flux*) viditelného bodu povrchu. Jak už bylo řečeno každý pixel je chápán jako nepřímý zdroj světla ve scéně. Zářivý tok $\Phi_p$ definuje jeho jas a jeho normála $n_p$ určuje prostorovou vyzařovací charakteristiku, jak možné vidět na obrázku 4.1. Zářivost vyzařovaná do směru $\omega$ je pak:

$$I_p(\omega) \quad = \quad \Phi_p \langle n_p | \omega \rangle_+ \tag{4.1}$$

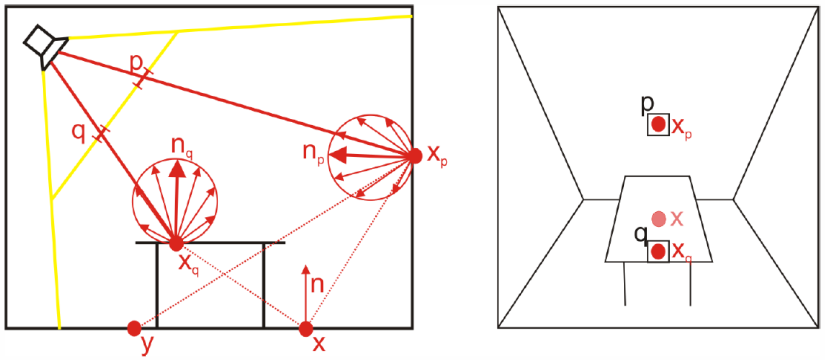
Kde $\langle | \rangle_+$ je skalární součin, kde se záporné výsledky nahradí nulou. Intezitu záření v povrchovém bodě $x$ s normálou $n$ lze vzhledem ke zdroji nepřímého světla v pixelu $p$ pak spočítat následovně:

$$E_p(x, n) \quad = \quad \Phi_p \frac{\langle n_p | x - x_p \rangle_+ \langle n | x_p - x \rangle_+}{\|x - x_p\|^4} \tag{4.2}$$

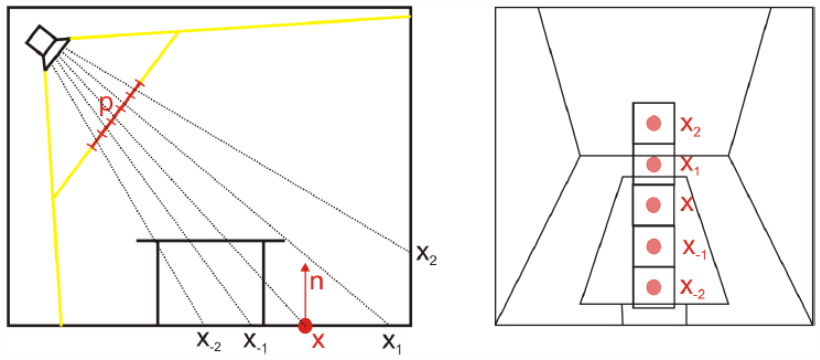
Co obsahuje RSM bylo již zmíněno. Vetšina těchto hodnot je poměrně jasná. Jediná, kterou zbývá objasnit je odražený zářivý tok $\Phi_p$. Její výpočet je poměrně jednoduchý a skládá se ze dvou menších kroků. Nejdříve je nutné spočítat vyzařovaný tok skrz jeden pixel. U jednotného směrového zdroje je to konstatní hodnota. Pokud budeme uvažovat světelný zdroj typu reflektor, pak hodnota toku klesá tím víc, čím jsme dále od směru světla (dále od směru bude hodnota měnší). Výsledný odražený tok je pak právě zářivý tok vynásobený koeficientem odrazivosti daného povrchu.

Intenzita nepřímého osvětlení v povrchovém bodě $x$ s normálou $n$ může být aproximována tak, že sečteme osvětlení ze všech zdrojů nepřímého světla.

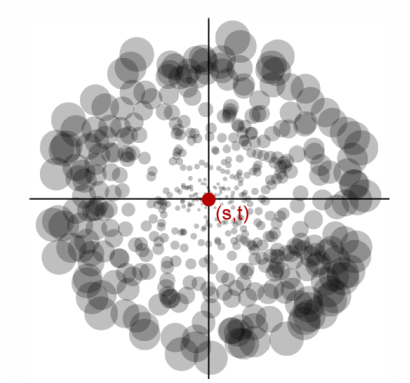
$$E(x, n) \quad = \quad \sum_{pixelsp} E_p(x, n) \tag{4.3}$$




Obrázek 4.1: Dva pixely ($x_p$ a $x_q$), které jsou nepřímými zdroji světla a jim odpovídající pixely $p$ a $q$ v RSM. Obrázek převzat z [4].




Obrázek 4.2: Výběr relevatních světelných zdrojů z RSM. Obrázek převzat z [4].




Obrázek 4.3: Schéma pro výběr vzorků. Obrázek převzat z [4].

Jelikož typické stínové mapy mají poměrně velký počet pixelů (např. $1024 \times 1024$) a vyčíslení této sumy (4.3) by bylo velmi výpočetně náročné, tak se používá fixní počet světelných zdrojů (např. 400). Toto snížení je kompeznováno tím, že je snaha brát co možná nejrelevantnější vzorky se světelnými zdroji (pixely) – tzv. *importance-driven approach*. Tento princip ilustruje obrázek 4.2. Zaměříme se na bod $x$. Tento bod není přímo osvícený (není

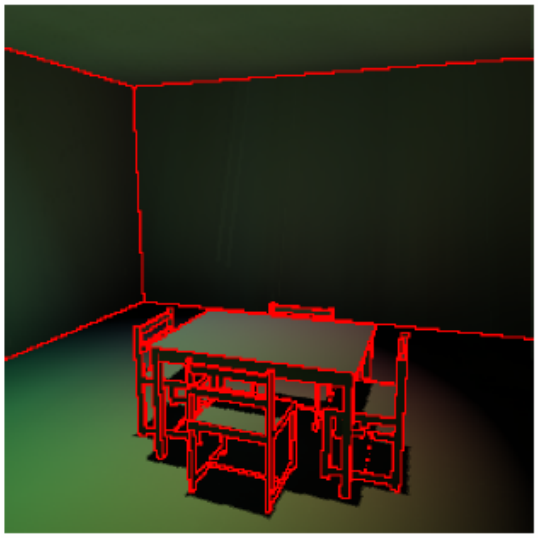
vidět ve stínové mapě). Po promítnutí tohoto bodu do stínové mapy zjistíme, že blízko jsou body $x_{-1}$ a $x_{-2}$. Ovšem jejich normály míří směrech od bodu $x$ a tudíž nijak nepřispívají nepřímým osvětlením. Další kandidát je bod $x_1$, ale tento bod leží na stejné rovině jako $x$ a tudíž také nepřispívá. Nejrelevatnější světelný zdroj bude $x_2$.

Lze říct, že pixely, které jsou blízko ve stínové mapě, jsou blízko i ve světových souřadnicích. Zdroje nepřímého světla jsou také blízko sobě a z toho plyne že musí být i blízko sebe ve stínové mapě. Výběr takovýchto pixelů lze realizovat pomocí schématu, které lze vidět na obrázku 4.3. Kdy se nejdříve promítne bod $x$ do stínové mapy a hledají se okolní body. Hustota vzorkování klesá se čtvercem vzdálenosti od středu vzorkování. Jednotlivé vzorky jsou pak díky měnící se hustotě vzorků ještě váhovány – čím dále od středu vzorkování, tím větší váha. Vzorkování pak probíhá v polárních souřadnicích a pozice vybraného pixelu (zdroje světla) je pak určena následujícím vztahem:

$$(s\ + r_{max}\xi_1 sin(2\pi\xi_2), t + r_{max}\xi_1 cos(2\pi\xi_2)) \tag{4.4}$$

Kde $s$ a $t$ jsou souřadnice, který jsme získali promítnutím bodu $x$ do stínové mapy, $\xi_1$ a $\xi_2$ jsou náhodná čísla s rovnoměrným rozložením. Tento vzor lze předpočítat v aplikaci jednou a potom používat pro všechny zdroje stejný.

Tento algoritmus se dá ještě optimalizovat tím, že se spočítá nepřímé osvětlení pro podvzorkovaný obraz a ten se pak vykreslí v plné kvalitě a zkoumá se pro každý pixel, zda pro něj může být spočítáno nepřímé osvětlení interpolací okolních 4 pixelů (podvzorkovaných). Interpolace se provádí pouze případě, pokud jsou 3 nebo 4 vzorky vyhodnoceny jako vhodné. V opačném případě je pixel v tomto průchodu zahozen a je pak pro něj dopočítáno nepřímé osvětlení až v posledním průchodu a to podle rovnice 4.3. Vzorky vhodné pro interpolaci jsou takové, pro které platí, že normála vzorku je podobná normále pixelu a zároveň pokud je pozice vzorku ve světových souřadnicích blízko pozici pixelu. Tato optimalizace je vhodná především pro hladké povrchy (viz obrázek 4.4). Pro různě pokřivené povrchy a složitější geometrii není vhodná (ta se vyhodnotí plnohodnotným průchodem).



Obrázek 4.4: Efektivita interpolace – pro červené pixely není interpolace vhodná. Obrázek převzat z [4].

# Kapitola 5

# Light Propagation Volumes

*Light Propagation Volumes* byly představeny v roce 2009 jako součást herního enginu *CryEngine 3* vyvíjeného německou firmou *Crytek*[6]. V roce 2010 byla představena technika *Cascaded Light Propagation Volumes*[7], která oproti původí technice přináší nějaká vylepšení – například nahrazení jedné mřížky o pevné velikosti několika mřížkami menších velikostí. Jedná se o algoritmus, který má za cíl aproximaci nepřímého osvětlení s důrazem na plně dynamické scény a na zobrazení v reálném čase. Tento algoritmus lze upravit tak, aby počítal i s více odrazy nepřímého osvětlení. Další možné rožšíření je počítání odlesků nebo úprava algoritmu tak, aby bral v potaz vliv prostředí, ve kterém se světlo šíří.

Základní myšlenka je, že celá scéna pokryta mřížkou, kterou se osvětlení šíří. Této mřízce se také říká *light propagation volume* (dále jen LPV) a jedná se o trojrozměrnou mřížku fixních rozměrů – toto omezení odstraňují právě *Cascaded Light Propagation Volumes*, které přidávají kaskádu několika mřížek. Ve skutečnosti jsou potřeba mřízky dvě (v základním algoritmu). První mřížka obsahuje intenzitu nepřímého osvětlení (LPV) a druhá obsahuje aproximaci geometrie (*Geometry volume* – dále jen GV). S tím, že GV je posunuto o polovinu buňky ve všech směrech (středy buněk GV leží přesně na rozích buňek LPV). Obě tyto mřížky obsahují aproximaci nízko-frekvenční sférické harmonické funkce (dále jen SH) a jsou inicializovány úplně od základu v každém snímku. Samotný výpočet nepřímého osvětlení lze rozdělit do několika po sobě jdoucích kroků:

- Inicializace LPV povrchy, které způsobují nepřímé osvětlení a přímým světlem o nízké frekvenci (plošné zdroje světla).
- Vytvoření hrubé aproximace blokující geometrie.
- Samotná propagace světla z LPV a akumulování průběžných výsledků – výsledná distribuce světla.
- Konečné vykreslení s využitím informace z předchozích kroků.

## 5.1 Inicializace LPV

LPV se používá pro účely výpočtu nízko-frekvenčního světla – zejména toho nepřímého. Pro přímé osvětlení se použijí standardní techinky (např. *shadow mapping*). Inicializace je ideově založena na tom, že můžeme převést světlo o nízké frekvenci na množinu *virtual point lights* (dále VPL)[8].

Nejdříve je nutné vytvořit VPL pro nepřímé osvětlení. K tomuto účelí se hodí již zmíněná technika RSM (viz sekce 4), kde každý texel může být. Po tomto následuje další krok, což je převedení všech VPL do reprezentace pomocí sférických harmonických funkcí a uložení jejich příspěvků do jednotlivých buněk LPV.

SH reprezentují směrovou distribuci intenzity (jinými slovy jakým směrem se šíří světlo z dané buňky). Pokud použijeme $n$ pásem (*bands*) SH, tak dostaneme $n^2$ koeficientů $c_{l,m}$ pro bázové funkce $y_{l,m}(\omega)$, kde $l$ je pásmo (*band*) a $m$ je stupeň a musí platit podmínka $-l \leq m \leq l$. Pro naše účely budeme používat SH až do pásma 2 (dostaneme vektor 4 koeficientů)[6] a polynomiální reprezentaci bázových funkcí lze pak vyjádřit následovně[16]:

$$y_{0,0}(\omega) = \frac{1}{2\sqrt{\pi}} \tag{5.1}$$

$$y_{1,-1}(\omega) = -\frac{\sqrt{3}}{2\sqrt{\pi}}y \tag{5.2}$$

$$y_{1,0}(\omega) = \frac{\sqrt{3}}{2\sqrt{\pi}}z \tag{5.3}$$

$$y_{1,1}(\omega) = -\frac{\sqrt{3}}{2\sqrt{\pi}}x \tag{5.4}$$

Pro každou barevnou složku (červená, zelená a modrá) je směrová distribuce dané barvy reprezentována vynásobením toku a SH koeficientů pro sevřený kosinový lalok *clamped cosine lobe*, který je natočený ve směru normály $n_p$ VPL. Tento sevřený kosinový lalok je popsán pomocí tzv. *zonal harmonics*[16, 13] – projekce SH, které mají rotační symetrii kolem osy. Koeficienty pak lze spočítat podle následujícího vztahu následovně[16]:

$$f_{l,m} = \sqrt{\frac{4\pi}{2l+1}}z_l y_{l,m}(d) \tag{5.5}$$

Kde $l$ pásmo, $z_l$ je nenulový koeficient pásma $l$ a $d$ je směr rotace a výsledné koeficienty jsou následující:

$$c_{0,0} = \frac{\sqrt{\pi}}{2} \tag{5.6}$$

$$c_{1,-1} = -\sqrt{\frac{\pi}{3}}y \tag{5.7}$$

$$c_{1,0} = \sqrt{\frac{\pi}{3}}z \tag{5.8}$$

$$c_{1,1} = -\sqrt{\frac{\pi}{3}}x \tag{5.9}$$

Tento postup se aplikuje na všechny VPL, které jsou pak následně vloženy do LPV.

Pro přímé světlo o nízkých frekvencích (například to z plošných světelných zdrojů) se aplikují úplně stejné kroky.

## 5.2 Geometrie

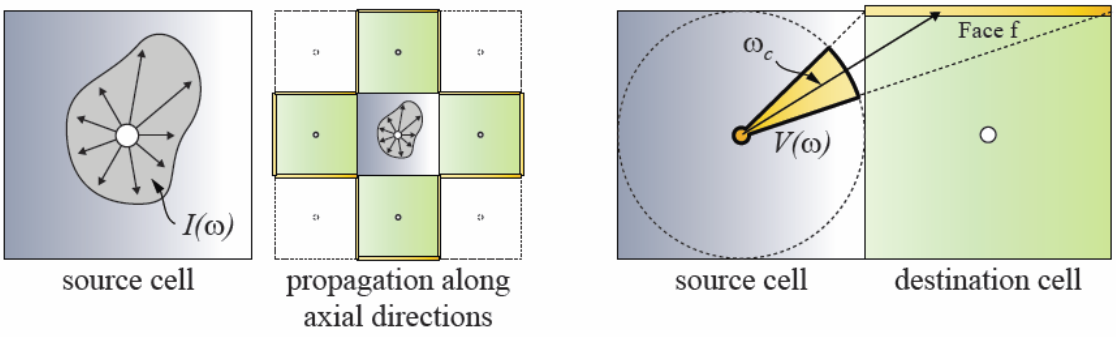
Mimo počátečního rozložení intenzity se je ještě vytváří hrubá aproximace geometrie ve scéně a to za účelem blokování světla při jeho propagaci a tím pádem i pro výpočet nepřímých stínů. K tomu slouží tzv. *accumulated blocking potencial* – pravděpodobnost zablokování světla z určitého směru, které prochází buňkou v GV. Pro jeden *surfel* (nejjednodušší

primitivum k popsání vlastností povrchu objektu) závisí na jeho velikosti a na úhlu mezi jeho normálou a směrem světla. A to následovně: $B(\omega) = A_s s^{-2} \langle n_s | \omega \rangle_+$, kde $A_s$ je plocha *surfelu*, $n_s$ jeho normála a $s$ je velikost buňky v GV.

Stejně jako když jsem zakomponovávali VPL do LPV, tak stejně tady akumulujeme projekce SH blokujícího potenciálu do GV. Vzhledem k tomu, že povrch může být samplován jak z pohledu kamery, tak z RSM (jedné či více), tak je třeba se ujistit, že *blocking potencial* nebyl přidán vícekrát. Rešení je více GV, které jsou po zakomponování (*injection*) sloučeny do jednoho a to tak, že se vybere největší vektor koeficientů SH.

## 5.3 Propagace světla

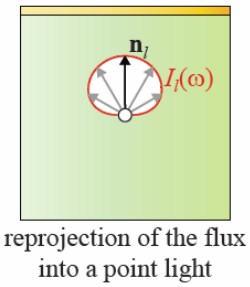
Samotná propagace probíhá iterativně. Pro první krok iterace se bere LPV z fáze inicializace. Je jasné, že další iterace budou brát jako vstup LPV z předchozího kroku. Každá buňka obsahuje intenzitu jako vektor koeficientů SH a světlo je pak propagováno od 6 směrů a to po jednotlivých osách (viz obrázek 5.1, kde je znázorněná propagace ve 2D). Teď pár slov k samotné propagaci. Označme si aproximaci SH intenzity ve zdrojové buňce jako $I(\omega) \approx \sum_{l,m} c_{l,m} y_{l,m}$, kde $c_{l,m}$ jsou koeficienty a $y_{l,m}$ je bázová funkce. Dále je nutné spočítat tok na každou přilehlou stranu buňky. K tomu slouží funkce viditelnosti $V(\omega)$ stěny $f$ (v cílové buňce) definovaná následovně. $V(\omega) = 1$ pokud paprsek, který začíná ve středu zdrojové buňky protíná stěnu $f$ a $V(\omega) = 0$ jinak. Celkový tok je pak možné vyjádřit jako $\Phi_f = \int_\Omega V(\omega) I(\omega) d\omega$. Funkce viditelnosti pro jednotlivé strany lze promítnout do SH a tím dostaneme vector koeficientů $v_{l,m}$, potom $V(\omega) \approx \sum_{l,m} v_{l,m} y_{l,m}(\omega)$. Potom lze celkový tok spočítat jako kartézský součin vektorů koeficientů SH $c_{l,m}$ a $v_{l,m}$. V [7] se autoři rozhodli místo přenosových vektorů $v_{l,m}$ (z důvodu nízké přesnosti pro nízké řády SH aproximace) počítat prostorový úhel (*solid angle*) $\Delta\omega_f = \int_\Omega I(\omega) d\omega$ pro každou stěnu v cílově buňce a dále ještě určí centrální směr $\omega_c$ vzniklého kuželu. Potom tedy výsledný tok, který dopadá na stěnu $f$ je $\Phi_f = \Delta\omega_f / (4\pi) \cdot I(\omega_c)$ (obrázek 5.1).



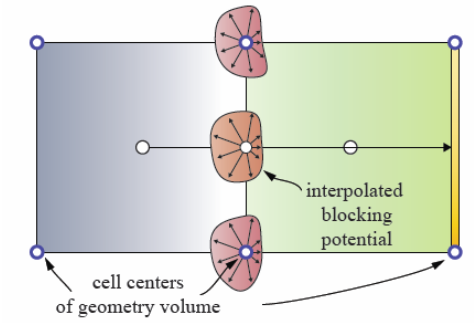

Obrázek 5.1: Znázornění schématu propagace světla (vlevo) a znázornění výpočtu toku na stěnu (vpravo). Obrázek převzat z [7].

Po tom, co se spočítá dopadající tok na kokrétní stěnu nějaké cílové buňky, je třeba provést reprojekci – dopadající tok je převeden do směrové distribuce ze středu buňky (obrázek 5.2). A to následovně. Ve středu buňky se vytvoří nové světlo s tokem $\Phi_l = \Phi_f / \pi$ ve směru čelem ke stěně. Následně je sevřený kosinový lalok (*clamped cosine lobe*) otočený ve směru směrového vektoru tohoto nového světla promítnut do SH koeficientů a ty jsou následně vynásobeny $\Phi_l$. Tento krok je nutné provést opět pro všechny barevné složky a pro

všechny stěny (a samozřejmě pro všechny sousedy zdrojové buňky). Koeficienty se v buňce sčítají a jsou následně použity jako výchozí v další iteraci.




Obrázek 5.2: Znázornění reprojekce toku do bodového světelného zdroje. Obrázek převzat z [7].




Obrázek 5.3: Bilineární interpolace blokujícího potenciálu. Obrázek převzat z [7].

Dále je potřeba vzít v potaz blokování světla geometrií. K tomuto slouží GV vytvořené při inicialzaci, které právě obsahuje pravděpodobnosti zastínění. Jelikož je celé GV posunuté o půl buňky (středy GV leží na rozích LPV), tak při propagaci je nutné provést interpolaci SH koeficientů v GV na střed strany, kterou propagujeme (viz obrázek 5.3) a následně vyhodnotit zastínění pro danou směr propagace a patřičně upravit intenzitu.

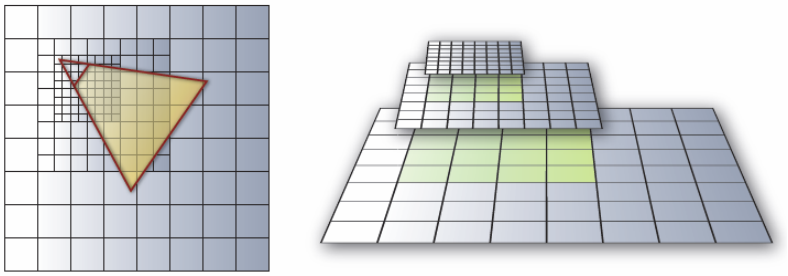
Výsledky každé iterace se uloží do zvláštní 3D mřížky. Výsledná mřížka $G_r$ bude pak:

$$G_r = \sum_{i=0}^{k} G_k \tag{5.10}$$

Kde $k$ je celkový počet iterací a $G_k$ jsou mřížky z jednotlivých iterací. Počet iterací pak závisí na rozlišení mřížky a měl by dosahovat dvojnásobku nejdelší strany mřížky. Ovšem takovýto počet kroků pro velké scény není vhodný pro interaktivní zobrazení. Toto vedlo k rožíšření, které používá vícero mřížek pohybujících se s kamerou (*Cascaded Light Propagation Volumes* – viz kapitola 5.5).

## 5.4 Vykreslení

Jedná se o poslední fázi algoritmu, která pro dotazování se na nepřímé osvětlení ve scéně používá výsledná mřížka $G_r$ z předchozího kroku. Tento krok může být proveden až při



Obrázek 5.4: Kaskádové mřížky pro světlo a geometrii. Obrázek převzat z [7].

vykreslovacím průchodu, kde se aplikuje přímé osvětlení. Na základě pozice zpracovávaného fragmentu vyčteme SH koeficienty z $G_r$. Dále provedeme projekci do SH koeficientů ve směru záporné normály fragmentu a provedeme skalární součin s SH koeficienty vyčtenými z mřížky.

## 5.5 Cascaded Light Propagation Volumes

Jedná se o rožšíření algoritmu *Light propagation volumes*, které umožňuje místo jedné mřížky použít mřížek vícero. Důvod je vcelku jednoduchý. Pokrýt celou velkou scénu mřížkou o rozumném rozlišení není rozumně možné z důvodu rostoucí paměťové složitosti. Jednotlivé mřížky nejsou vycentrovány okolo pozorovatele, ale jsou předsunuty ve směru pozorování.

Propagace světla je počítána pro všechny mřížky nezávisle. Při výpočtu výsledného osvětlení se postupuje od nejjemnější mřížky (nejblíže k pozorovateli) – získá se vzorek na dané pozici v mřížce a zkombinuje se všemi vzorky, které na dané pozici spadají do mřížek o úroveň výše. Na hranici této mřížky se vytvoří interpolací přechod do další mřížky. Princip je znázorněn na obrázku 5.4. Dále se musí provést tzv. *snapping*, což není nic jiného než posun mřížky po celých násobcích velikosti buňky. Tímto značně snížíme problikávání při pohybu kamery.

# Kapitola 6

# Implementace a demonstrační aplikace

V této kapitole si popíšeme nejdůležitější problémy, na které jsem během implementace algoritmu narazil, a popíšeme si způsob, jakým byly vyřešeny. Mimo to si stručně představíme použité techologie ve výsledné aplikaci.

## 6.1 Demonstrační aplikace

Cílem této práce je naimplementovat metodu *Light Propagation Volumes* za použití knihovny *OpenGL*. Dalším bodem je otestování vizuální kvality naimplementovné metody a změření výkonu na demonstrační scéně. Pro implementaci byl zvolen programovací jazyk C++ za podpory několika knihoven:

- *OpenGL* – Pro zobrazení je použita verze 4.3 (s ohledem na starší karty) spolu s jazykem pro programování shaderů *GLSL* (ve verzi 430).
- *Assimp* – Jedná se o multiplatformní open–source knihovnu pro načítání nesčetně formátů uložení 3D data, za tímto účelem je také ve výsledné aplikaci použita.
- *GLew* – Multiplatformní open–source knihovna, která poskytuje mechanismus pro zjištění rozšíření podporovaných na dané grafické kartě. Mimo to se stará o samotné načítání rozšíření.
- *GLM* – Jedná se o knihovnu, která je distribuována ve formě hlavičkového souboru a která je založena na specifikaci *OpenGL Shading Language (GLSL)*. Poskytuje spoustu funkcí pro práci s maticemi a vektory. Výhodou je, že používá vpodstatě stejnou syntaxi jako právě *GLSL*.
- *SDL 2.0* – Tato knihovna zajišťuje obsluhů vstupů z klávesnice a myši, dále je zodpovědná za samotnou zprávu oken a výtváření příslušného *OpenGL* kontextu.
- *DevIL* – Samotný název je zkrácenina výrazu *Developer's Image Library*, která zajišťuje načítání obrázků různých formátů a podporuje právě i *OpenGL* pro zobrazování.

Ve výsledné aplikaci by mělo být možné se volně pohybovat po demostrační scéně. Mimo to by měl být rámci aplikace ještě zabudovaný benchmark, který by otestoval výkon

metody. Za tím účelem bude nutné nějakým způsobem měřit výkon a udělat předdefinovaný pohyb kamery po dané trajektorii ve scéně (využití spline křivek právě pro pohyb kamery). K měření se budou hodit *timer queries*, které umožňují relativně lehce a přesně měřit čas strávený výpočty na grafické kartě.

## 6.2 Prostorový úhel, korekce

V kapitole 5.1 jsme si popsali, že autoři článku se rozhodli používat prostorový úhel (*solid angle*) místo přenosových vektorů, ovšem už neuvádí, jak tento úhel spočítat. V [10] uvádí autor postup výpočtu tohoto úhlu. Vypočtené hodnoty jsou použity ve výsledné aplikaci.

Mimo to autor v tomto článku uvádí korekce týkající se zejména korekčních koeficientů pro několik částí algoritmu, kdy se snaží dosáhnout fyzikálně přesnějšího popisu. Jedná se zejména intenzitu VPL při inicializaci mřížky, kdy autor navrhuje vztah 4.1 podělit konstantou $\pi$. Další korekce se týká propagace intezity, která je v původním článku definována $\Phi_f = \Delta\omega_f/(4\pi) \cdot I(\omega_c)$, kdy autor ukazuje, že dělení konstantou $4\pi$ je zbytečné. Poslední korekcí je úprava reprojekce toku, kdy za použití korekce intenzity VPL navrhuje odstranit ze vztahu $\Phi_l = \Phi_f/\pi$ dělení konstantou $\pi$.

## 6.3 Reflective shadow maps

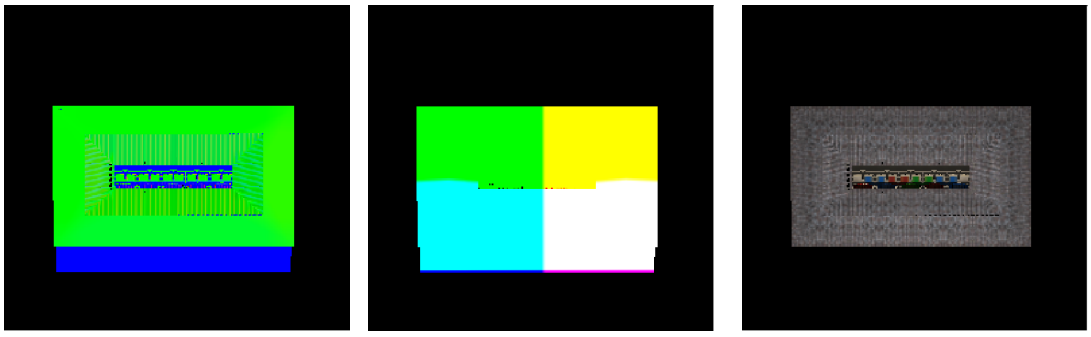
Co obsahují jednotlivé textury bylo již popsáno v kapitole 4 a jejich obsah je vidět na obrázku 6.1 a je možné si všimnout, že mezi nimi není hloubková mapa a to z důvodů popsaných v sekci 6.3.1. Samotné plnění využívá schopnosti vykreslovat do více textur najednou (*Multiple Render Targets*). Odražený světlný tok (*flux*) je spočítán následovně:

```
lightColor * diffuse * clamp( dot( lightDir, worldNormal ),0.0,1.0 )
```

kde `diffuse` je difuzní barva získaná z příslušné textuy.

### 6.3.1 Hloubková mapa

S ohledem na rychlost inicializace mřížky by měl být počet VPL, co možná nejmenší. Ovšem pokud bychom použili hloubkovou mapu z takto malého utrpěly by na kvalitě výsledné stíny. Proto jsem se v rámci své práce rozhodl, že hloubkovou mapu budu generovat zvlášť a ve vetším rozlišení. Rozdíl ve výkonu není velký a výsledné stíny pak vypadají lépe.



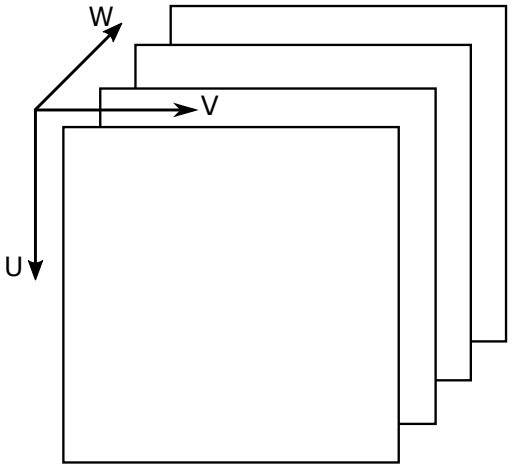
(a) Normály v RSM. (b) Pozice v RSM. (c) Tok (*flux*) v RSM.

Obrázek 6.1: Jednotlivé složky v RSM.

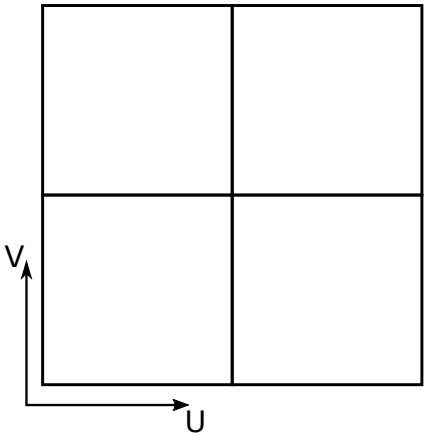
## 6.4 Reprezentace mřížky

Jelikož je třeba nějak reprezentovat mřížku, ve které se osvětlení šíří, a autoři neuvádí, jak ji reprezentují, tak první problém nastal právě s její reprezentací. Možností, jak tuto mřížku reprezentovat, je několik:

- Jako pole 2D textur naskládáných vedle sebe ve dlaždicích (*texture atlas* –viz obrázek 6.3).
- Jako 3D texturu – tu jsi lze představit jako několik 2D textur naskládaných v $w$ vrstách na sobě (viz obrázek 6.2), kde $w$ je hloubka 3D textury.




Obrázek 6.2: Možná představa 3D textury.




Obrázek 6.3: Možná vizualizace 2D atlasu textur.

Já jsem si vybral druhou možnost a to reprezentaci pomocí 3D textury, protože mi tato reprezentace přijde intuitivnější. Na druhou stranu to má i své omezení a to hlavně díky způsobu, jak tuto texturu naplnit, kdy je třeba aby hardware podporoval programovatelný *geometry shader*, který je dostupný ve verzi *OpenGL* 3.2 a vyšší. V opačném případě by bylo

nutné použít druhou variantu s atlasem textur, která by měla běžet i na starším hardwaru. V této práci se ovšem předpokládá verze 4.3 a vyšší.

### 6.4.1 Vytvoření a pohyb mřížky

Vzhledem k tomu, že součástí práce je také implementace rozšíření původní metody o kaskádu pohybujících se mřížek, bylo nutné vytvořit strukturu, která reprezentuje mřížku a umožňuje s ní pohybovat. Samotné plnění mřížky (viz kapitola 6.4.2) a propagace zůstávají nezměněny.

Třída `Grid` reprezentuje mřížku, která je popsána minimem, velikostí buňky (viz obrázek 6.4) a středem mřížky. Samotný pohyb mřížky (metoda `translateGrid()`) se skládá z několika kroků:

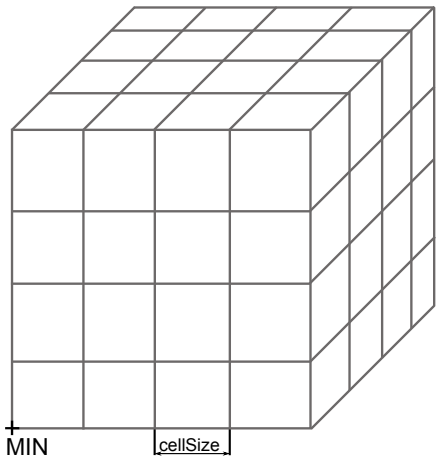
1. Posun středu mřížky na pozici kamery a výpočet nového minima mřížky. S tím, že nutné zajistit, aby se posun provedl pouze o celé násobky velikosti buňky.

2. Posun mřížky (resp. jejího minima) ve směru pohledu kamery, neprovádí se ovšem posun až na hranu mřížky, ale pouze o $0.8 * s_c * 0.5$, kde $s_c$ je velikost buňky. Opět je nutné zajistit, aby se provedl posun pouze o celé násobky velikosti buňky.

Důvodem proč se je důležitý posun právě minima mřížky je ten, že výpočet indexu buňky v mřížce je založen právě na minimu mřížky. Výpočet se pak provádí podle následujícího vztahu:

$$index = f\left(\frac{p - v_m}{s_c}\right) \tag{6.1}$$

Kde $p$ značí pozici aktuálního vrcholu, $v_m$ je minimum mřížky a $s_c$ značí velikost buňky pro danou mřížku. Funkce $f(x)$ reprezentuje zaokrouhlování na celá čísla. Zvláště při použití kaskády mřížek se může stát, že se pomocí tohoto vztahu dostaneme mimo rozsah dané mřížky. V aplikaci je to řešeno nastavením sampleru textury na hodnotu `GL_CLAMP_TO_BORDER` pro všechny texturovací souřadnice ($s$,$t$,$r$). Tím zajistíme, že pokud se dostaneme mimo rozsah textury, tak vrácená hodnota bude odpovídat specifikované barvě okraje (*border color*), která je ve výchozím nastavení rovna samým nulám.




Obrázek 6.4: Mřížka.

### 6.4.2 Naplnění mřížky

Jak již bylo zmíněno, tak 3D textura se hodí reprezentaci mřížky, avšak nastává problém, jak ji naplnit. Určitě k tomu lze využít *framebuffer object* (dále FBO) a nastavit tuto texturu jako *Render target* (dále jen RT). Problém je, že v *OpenGL* se musí zapisovat do objemové textury po vrstvách. Tady vystvává další problém a to počet barevných bufferů (`GL_COLOR_ATTACHMENTi`) připojených k FBO, do kterých se kreslí. Ten je velikostně omezen (např. na 8) a tudíž toto řešení je pro větší mřížky nepoužitelné. Řešením je navázat na FBO 3D texturu na nulté vrstvě:

```
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, textureId, 0);
```

A následně použít *geometry shader*, ve kterém se vybere vrstva do které se bude zapisovat (viz útržek kódu 6.1).

```
for (int i = 0; i < gl_in.length(); i++)
{
        gl_Position=gl_in[i].gl_Position;
        gl_Layer = v_volumeCellIndex[i].z;
        //...
        EmitVertex();
        EndPrimitive();
}
```

Kód 6.1: Výběr vrstvy 3D textury.

Kde `v_volumeCellIndex` je souřadnice do 3D textury jejíž výpočet je popsán v sekci 6.4.1 vztahem 6.1. Jelikož je velmi pravděpodobné, že budeme chtít k hodnotě na jednom indexu přičíst nějakou hodnotu musíme tento fakt nějak zohlednit. K tomu se dá využít aditivní blending (*additive blending*), který je ovšem nutné nejdříve povolit, což se provede následovně:

1. Povolení blending `glEnable(GL_BLEND)`.
2. Nastavení zdrojové, cílové hodnoty a rovnice pro blending `glBlendFunc(GL_ONE, GL_ONE)`, `glBlendEquation(GL_FUNC_ADD)`.
3. Vykreslení scény.
4. Zakázání blendingu `glDisable(GL_BLEND)`.

Alternativou k tomuto postupu je použití atomických operací nad obrazem[1] v shaderech. To nám umožní atomicky přičíst hodnotu do paměti, která reprezentuje obraz, a to pomocí volání pouze jediné funkce:

```
imageAtomicAdd(image,coords,data)
```

Avšak i tento postup má několik omezení:

- Tyto operace jsou dostupné ve verzi *OpenGL* 4.2 nebo případně pomocí rozšíření.

[1] Dostupné na: https://www.opengl.org/wiki/Image_Load_Store

- Pracují pouze se novým typem proměnných tzv. *image variables*. Tyto proměnné jsou založeny na typu zdrojové textury (ne všechny typy textur mají odpovídající *image type*). Například pro 3D texturu je odpovídající *image type* `gimage3D`.
- Dalším omezením při častém používání atomických operací je snížení výkonu.
- Ovšem hlavní nevýhodou je, že tyto operace umožňují atomicky přičítat pouze celočíselné hodnoty (znaménkové nebo beznaménkové).

S ohledem na poslední bod jsou tyto operace nad obrazem z důvodu nemožnosti atomicky přičítat čísla s plovoucí řádovou čárkou nepoužitelné pro náš případ. V [2] přišli autoři s možností, jak emulovat právě tyto operace v shaderech. Ovšem, jak sami autoři říkají toto řešení je značně pomalé. Řešení přinesl výrobce grafických karet *NVIDIA* (ovšem až od architektury *Kepler*), kdy zpřístupnil atomické operace nad čísly s plovoucí řádovou čárkou skrze rozšíření `NV_shader_atomic_float`[2]. Pro atomické sčítání vektoru čísel v plovoucí řádové čárce je nutné ještě jedno rozšíření a to `NV_shader_atomic_fp16_vector`[3]. Útržek kódu 6.2 ukazuje jednoduchý shader využívající tyto rozšíření.

```
#version 430

#extension GL_NV_shader_atomic_float : require
#extension GL_NV_shader_atomic_fp16_vector : require
#extension GL_NV_gpu_shader5 : require
layout(rgba16f ,location = 0) uniform image3D LPVGridR;

void main()
{
    imageAtomicAdd(LPVGridR, ivec3(1,1,1), f16vec4(1.23));
}
```

Kód 6.2: Fragment shader demonstrující atomické operace s čísly v plovoucí řádové čárce.

Ve výsledné aplikaci jsou naimplementovány oba postupy (aditivní blending a použití atomických operací), ovšem pokud grafická karta nepodporuje potřebné rozšíření, tak nejsou shadery využívající atomické operace vůbec zkompilovány. Výkon obou těchto řešení je porovnán v kapitole 7.1.

### 6.4.3 Image textury

Jak již bylo zmíněno výše, pro použití atomických operací v shaderech je v GLSL zavedenen nový typ proměnné pro práci texturami `gimage`. Dále je trochu odlišný způsob napojení těchto textur na obrazové jednotky (*image units*) ve zdrojovém kódu. Slouží k tomu funkce `glBindImageTexture()`. U tohoto typu textury je nutné brát v potaz některá omezení. Zejména omezení počtu aktivních jednotek na 8 na stroji, kde se vyvíjelo (grafická karta GTX 960).

[2] Dostupné na: https://www.opengl.org/registry/specs/NV/shader_atomic_float.txt

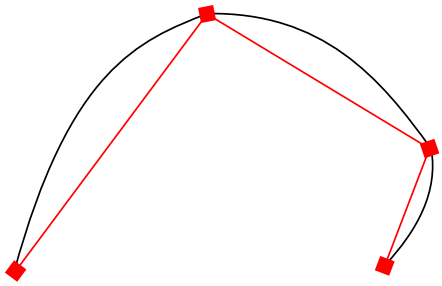
[3] Dostupné na: https://developer.nvidia.com/sites/default/files/akamai/opengl/specs/GL_NV_shader_atomic_fp16_vector.txt

### 6.4.4 Mazání obsahu textur

Vzhledem k tomu, že se mřížka plní každý snímek od nuly, je nutné zajistit, aby všechny 3D textury byly v dalším snímku vynulovány. Pro tyto účely se hodí funkce `glClearTexImage()`, která je přímo pro tyto účely dělaná avšak je dostupná až od verze *OpenGL* 4.4. Alternativou je použití funkce `glTexSubImage3D()`, která je ovšem řádově pomalejší. Průměr ze 100 časů je 0,0049 ms pro první funkci a 0,09958 ms pro druhou. Výsledná aplikace kontroluje, zda je dostupné rozšíření zřístupňující tuto funkci, pokud není dostupné, pak se použije druhá varianta.

## 6.5 Kamera

Pro volný pohyb ve scéně je implementována klasická kamera z pohlednu první osoby. Tato kamera je reprezentována třídou `CControlCamera`. Mimo tuto kameru je v práci implementována ještě animační kamera, která slouží jako pomocný nástroj pro měření výkonu a je reprezentována třídou `spline`. Tato kamera využívá pro interpolaci parametrů kamery (vektrory jako směr pohledu atp.) mezi klíčovými body kubický spline. Mezi dvěma klíčovými snímky je 200 interpolovaných pohledových matic. Ty vznikly z interpolovaných vektorů pozice kamery – *pos*, směru pohledu kamery – *dir* a směru vzhůru – *up*. Klíčové body (nachází se v souboru `keyFrames.txt`) byly vybrány pro tuto scénu ručně s ohledem na to, aby byla viditelná funkčnost algoritmu. Po skončení animace se aplikace automaticky ukončí. Vzhledem k tomu, že se interpolace počítá před samotným spuštěním vykreslování, tak se může stát, že pokud se snímek vykresluje příliš dlouho, tak se animace protáhne. Za účelem potlačení tohoto efektu bylo v rámci práci naimplementováno jednoduché „přeskakování“ interpolovaných snímků – tj. pokud se vykresluje snímek příliš pomalu, tak se přeskočí několik interpolovaných snímku.



Obrázek 6.5: Kubický spline pro interpolaci mezi klíčovými body.

## 6.6 Měření času

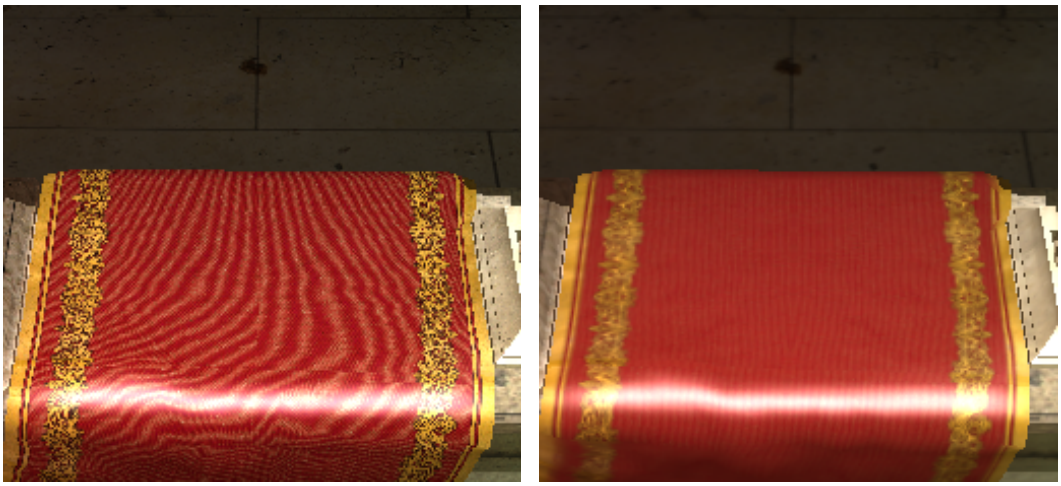
Jelikož cílem práce je také otestování implementovaného algoritmu, tak je nutné vyřešit, jak změřit čas strávený různými výpočty. V rámci práce jsem měřil, jak dlouho zaberou jednotlivé kroky algoritmu (vytvoření RSM, propagace...). První řešení je použití časovače operačního systému (například funkcí `SDL_getTicks()`), pak výsledný čas je jednoduchým rozdílem časů na začátku vykreslovací smyčky a konci vykreslovací smyčky. Problémem tohoto řešení je fakt, že změřený čas by nemusel být správný. Jelikož se měří čas strávený

na procesoru a *OpenGL* provádí nějaké výpočty na pozadí, tak systémový časovač vrátí hodnotu rovnou nebo blízkou nule.

Druhým řešením je použití časovače v knihovně *OpenGL*, který umožňuje zjistit, kolik času zabralo vykonání příkazu na grafické kartě. Jedná se o rozšíření *OpenGL Timer Queries* a je dostupné ve verzi *OpenGL 3.3* a novější. Při měření se postupuje následovně:

1. Začít dotazování (*query*) na výpočetní čas – `glBeginQuery(GL_TIME_ELAPSED, query);`.
2. Provést vykreslovací operace.
3. Zastavit dotazování na výpočetní čas – `glEndQuery(GL_TIME_ELAPSED);`.
4. Zažádat *OpenGL* o výsledek dotazu – `glGetQueryObjectui64v(query, GL_QUERY_RESULT, &elapsedTime);`.

Problém může nastat, když se pokusíme získat výsledek a ten není ještě připravený. V tom případě musí *OpenGL* počkat než se provedou měřené operace na pozadí, čož může mít za následek snížení výkonu. V aplikaci se o měření času stará třída `TimeQuery`.



(a) Bez mipmapingu. (b) S mipmapingem.

Obrázek 6.6: Porovnání vzhledu textur s vypnutým a zapnutým mipmapingem.

## 6.7 Formát pro uložené modely

Aplikace načítá modely ve formátu *.obj*. Jedná se o formát vyvinutý společností *Wavefront Technologies* pro textové uložení geometrické reprezentace dat jako jsou vrcholy, normály, texturovací souřadnice atp. Samotný formát sám o sobě ovšem nenese žádné informace o materiálech daného obejktu. K tomu slouží soubor s koncovkou *.mtl*. Mezi výhody tohoto formátu patří již zmíněná textová reprezentace dat a také fakt, že se jedná o otevřený formát. Jako nevýhodu bych zmínil fakt, že tento formát nepodporuje animace. V současnosti je tento formát podporován snad ve většině modelovacích programů.

O načítání modelů modelů se stará třída `Mesh`. Mimo samotného načtení modelu (za použití knihovny *Assimp*) tato třída ještě zajišťuje naplnění všech bufferů (VBO, VAO...) potřebných pro vykreslení. Poskytuje ovšem pouze tyto buffery a nezajištuje k vykreslení,

pro něj je potřeba aktivovat shader program před voláním metody `render()`. V aktivním shaderu jsou pak data dostupná na následujících indexech:

- 0 – informace o pozicích jednotlivých vrcholů (3 složkový vektor `vec3`).
- 1 – texturovací souřadnice (`vec2`)
- 2 – normály (`vec3`).

Dále tato třída při načítání modelu vytvoří difuzní textury, pokud daná textura není nalezena, metoda se ji snaží nahradit základní předdefinovanou texturou. Pro samotné vytváření textur z obrázků byla použita knihovna *DevIL* a těchto textur jsou pak automaticky generovány mipmapy (viz obrázek 6.6) pomocí funkce `glGenerateMipmap()`.

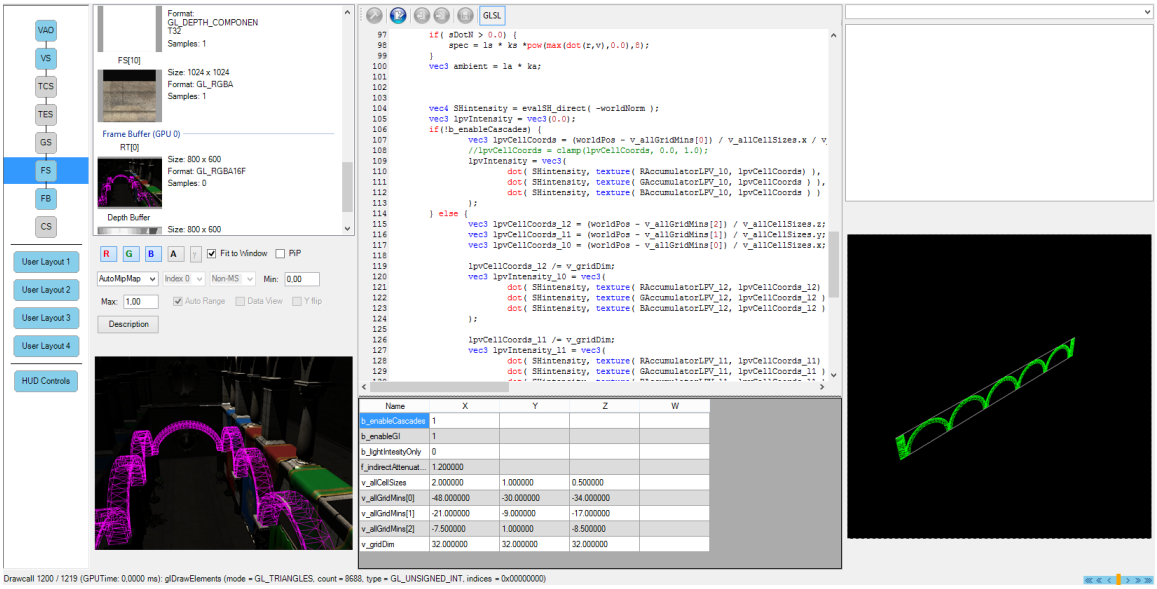

(a) Grafické rozhraní zobrazené přes scénu. (b) Zachycení jednoho snímku (*frame capture*).

Obrázek 6.7: Debugování pomocí nástroje *NSight*.

## 6.8 Ladění

Jelikož se jedná o grafickou aplikaci, jejíž většina běží na grafické kartě, tak ladění této aplikace je celkem nesnadný úkol. Naštěstí existují nástroje, které nám umožňují si zobrazovat obsahy jednotlivých bufferů, textur, zobrazovat hodnoty uniformních proměnných pro shader programy a mnoho dalšího. Za zmínku stojí například *CodeXL* od společnosti *AMD*, který je volně k dispozici a lze s ním pracovat i na *NVIDIA* (s jistými malými omezeními) a poskytuje mimo jiné plnou integraci s vývojovým prostředím *Visual Studio*. Dalším nástrojem od společnosti *AMD* je *GPU PerfStudio*, které obsahuje nástroje pro ladění (frame debugger), pro profilování a další.

Vzhledem k tomu, že práce byla vyvíjena na grafické kartě *NVIDIA*, tak byl použit nástroj *NSight*. Tento nástroj poskytuje velké množství nástrojů pro ladění a profilování grafických aplikací a také poskytuje plnou integraci do vývojového prostředí *Visual Studio*. Mezi jeho nejužitečnější vlastnosti, které jsem při vývoji použil, jednoznačně patří zachycení snímku (*frame capture*, zobrazen na obrázku 6.7) a také grafické rozhraní, které zobrazuje akutální využití grafické karty, histogram počtu primitiv vykreslených za jeden snímek, snímkovou frekvenci a další. Po tom, co zachytíme snímek, je možné se pomocí posuvníku dole (obrázek 6.7b) posouvat mezi jednotlivými voláními vykreslení (*draw call*) a zobrazovat si aktuální obsah bufferů, textury, shader programy a mnohé další. Některé z techto




Obrázek 6.8: Frame debugger nástroje *GPU PerfStudio* Je zde vidět aktuálně použitý shader program, obsah framebufferu a aktuálně vykreslovaný objekt.

informací (konkrétně textury) jsou viditelné přímo v plovoucím grafickém rozhraní přímo v okně, kde se vykresluje. Další vlastností je možnost debugování shaderů (jak v jazyce *GLSL*, tak v jazyce *HLSL*). Tato možnost ovšem zatím nefunguje na kartách s architekturou *Maxwell*.

# Kapitola 7

# Testování

V této kapitole se podíváme na výkon metody při různém nastavení na dvou různých sestavách, zejména se zaměříme na časy potřebné pro naplnění mřížky a časy následné propace světla v mřížce. Dále se podíváme na vizuální kvalitu metody. Vetšina testů probíhala na obou sestavách vyznačených v tabulce 7.1 s vyjímkou testů s atomickými operacemi, které probíhali pouze na sestavě s grafickou kartou NVIDIA.

| | **NVIDIA PC** | **AMD PC** |
|---|---|---|
| CPU | Intel Core i5-4590 @3,3GHz | Intel Core i5-4590 @3,3GHz |
| GPU | NVIDIA GTX 960 2GB | AMD R9 270 2GB |
| RAM | 8 GB | 8 GB |
| OS | Windows 8.1 | Windows 8.1 |

Tabulka 7.1: Testovací stroje.

## 7.1 Porovnání časů

Důležitým faktorem z hlediska výkonu je množství času, které algoritmus spotřebuje v jednotlivých fázích. V následujích testech se budeme zajímat o následující fáze algoritmu:

- **Rychlost naplnění RSM textury** – tento čas se příliš nemění, je to dáno tím, že je závislý na geometrii a na rozlišení RSM textury. Všechny následující testy (pokud není řečeno jinak) byly provedeny při rozlišení RSM textury nastaveným na $256 * 256$ pixelů.
- **Inicializace mřížky** – čas potřebný pro počáteční naplnění mřížky pomocí VPL. Tento čas také zahrnuje inicializaci mřížky pro blokující geometrii.
- **Čas propagace** – čas samotného propagování světla ve scéně.

V tabulkách 7.2, 7.4 jsou zaneseny časy potřebné pro naplnění textury pro 8 a 12 kroků propagace. Tyto hodnoty zahrnují i výpočet blokujícího potenciálu při propagaci. Naopak tabulky 7.3, 7.5 ukazují časy, kdy je tento výpočet zakázán. Ukázalo se, že výpočet blokujícího poteciálu je časově relativně náročný a prodlužuje výpočet řádově až o milisekundy. Dále je vidět, že kaskáda mřížek $x$ mřížek prodlouží výpočet propagace $x$krát, jak se dalo předpokládat. Toto škálování je vidět zejména na kartě od AMD.

| 8 iterací s blokující geometrií (bez atomických operací) – mřížka $32^3$ | | | | |
|---|---|---|---|---|
| **PC** | **Počet mřížek** | **RSM [ms]** | **inicializace [ms]** | **propagace [ms]** |
| NVIDIA PC | 1 | 0.274104 | 0.375929 | 3.3704 |
| NVIDIA PC | 3 | 0.274104 | 0.999607 | 8.75379 |
| AMD PC | 1 | 0.363047 | 1.35305 | 9.99295 |
| AMD PC | 3 | 0.363047 | 3.48448 | 29.2733 |

Tabulka 7.2: Porovnání časů na jednotlivé kroky algoritmu s blokující geometrií pro 8 iterací.

| 8 iterací bez blokující geometrie (bez atomických operací) – mřížka $32^3$ | | | | |
|---|---|---|---|---|
| **PC** | **Počet mřížek** | **RSM [ms]** | **inicializace [ms]** | **propagace [ms]** |
| NVIDIA PC | 1 | 0.265721 | 0.397134 | 3.05226 |
| NVIDIA PC | 3 | 0.265721 | 0.932677 | 7.31519 |
| AMD PC | 1 | 0.362724 | 1.35732 | 6.81374 |
| AMD PC | 3 | 0.362724 | 3.50141 | 19.7886 |

Tabulka 7.3: Porovnání časů na jednotlivé kroky algoritmu bez blokující geometrie pro 8 iterací.

| 12 iterací s blokující geometrií (bez atomických operací) – mřížka $32^3$ | | | | |
|---|---|---|---|---|
| **PC** | **Počet mřížek** | **RSM [ms]** | **inicializace [ms]** | **propagace [ms]** |
| NVIDIA PC | 1 | 0.274104 | 0.401629 | 6.07258 |
| NVIDIA PC | 3 | 0.274104 | 0.999594 | 13.8813 |
| AMD PC | 1 | 0.363047 | 1.35242 | 15.5127 |
| AMD PC | 3 | 0.363047 | 3.4849 | 45.6326 |

Tabulka 7.4: Porovnání časů na jednotlivé kroky algoritmu s blokující geometrií pro 12 iterací.

| 12 iterací bez blokující geometrie (bez atomických operací) – mřížka $32^3$ | | | | |
|---|---|---|---|---|
| **PC** | **Počet mřížek** | **RSM [ms]** | **inicializace [ms]** | **propagace [ms]** |
| NVIDIA PC | 1 | 0.265721 | 0.399961 | 5.02879 |
| NVIDIA PC | 3 | 0.265721 | 0.96792 | 11.3734 |
| AMD PC | 1 | 0.362724 | 1.35757 | 10.4721 |
| AMD PC | 3 | 0.362724 | 3.50252 | 30.7741 |

Tabulka 7.5: Porovnání časů na jednotlivé kroky algoritmu bez blokující geometrie pro 12 iterací.

Jak již bylo zmíněno v kapitole 6.4.2, tak v rámci práce byly implementovány dva způsoby plnění mřížky. Výkon prvního z nich byl změřen výše. Nyní se podíváme na výkon řešení využívajícího atomické operace. Je nutné podotknout, že následující testy byly provedeny na testovací sestavě s kartou NVIDIA, jelikož AMD karta nemá potřebná rozšíření. Ještě se ukázalo, že zatímco jedno z potřebných rozšíření (`NV_shader_atomic_float`) mají

i starší NVIDIA karty, tak druhé potřebné rozšíření (`NV_shader_atomic_fp16_vector`) neměla z nich ani jedna (testováno na kartách GTX 750m a GTX 660m). Z toho usuzuji, že toto rozšíření bude dostupné nejspíše až od architektury *Maxwell*. V tabulce 7.6 jsou vyneseny hodnoty pro oba přístupy a v grafu 7.1 jsou tyto hodnoty vizualizovány. Jak je vidět z tabulky 7.1, tak rozdíl ve výkonu obou řešení není tak velký, jak jsem původně čekal. Jedná se o zhoršení zhruba o 2ms. Dále je v tabulce 7.7 je vidět, že použitím kaskády mřížek snížíme výsledný čas zhruba o 7ms u prvního přístupu a o 15ms u atomických operací oproti případu, kdy je použita jedna velká mřížka, což je očekávané chování.

| **Metoda** | **Počet mřížek** | **Velikost mřížky** | **inicializace [ms]** | **propagace [ms]** |
|---|---|---|---|---|
| Přepínání vrstvy v GS | 1 | $32^3$ | 0.375929 | 3.3704 |
| Přepínání vrstvy v GS | 3 | $32^3$ | 0.999607 | 8.75379 |
| Atomické operace | 1 | $32^3$ | 0.397899 | 3.76333 |
| Atomické operace | 3 | $32^3$ | 0.998954 | 10.4171 |

Tabulka 7.6: Porovnání dvou přístupů (atomické operace a 3D textura s GS) pro 8 iterací.

| **Metoda** | **Počet mřížek** | **Velikost mřížky** | **inicializace [ms]** | **propagace [ms]** |
|---|---|---|---|---|
| Přepínání vrstvy v GS | 1 | $64^3$ | 0.796672 | 15.6416 |
| Přepínání vrstvy v GS | 3 | $32^3$ | 0.999607 | 8.75379 |
| Atomické operace | 1 | $64^3$ | 0.44032 | 25.0573 |
| Atomické operace | 3 | $32^3$ | 0.998954 | 10.4171 |

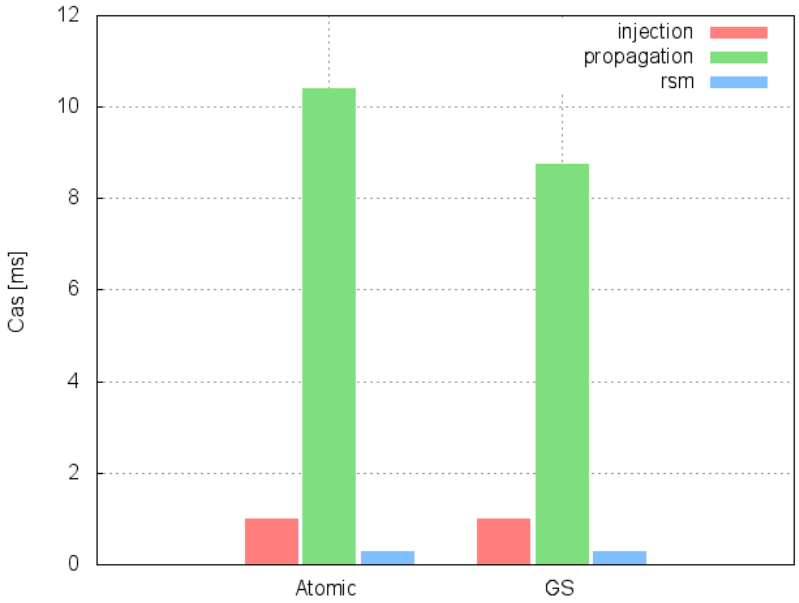
Tabulka 7.7: Porovnání výkonu pro jednu velkou mřížku a kaskádu mřížek pro 8 iterací.

| **Rozlišení RSM [px]** | **RSM [ms]** | **Inicializace [ms]** |
|---|---|---|
| 128*128 | 0.232448 | 0.41984 |
| 256*256 | 0.246784 | 0.999594 |
| 512*512 | 0.37376 | 4.46874 |

Tabulka 7.8: Vliv počtu VPL na čas inicializace mřížky.

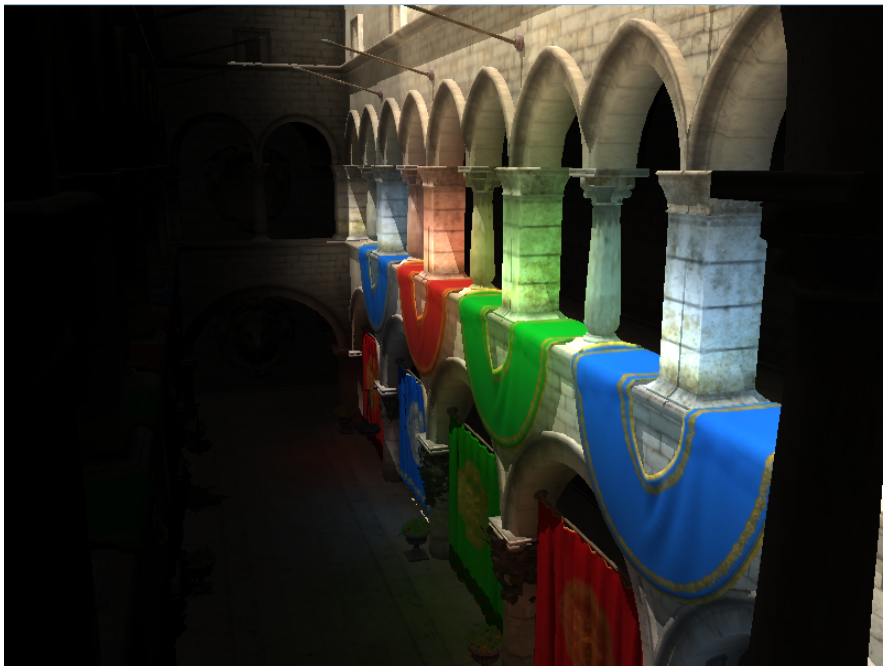
Dalším důležitým parametrem, který ovlivňuje výkon algoritmu je počet VPL (který závisí na rozlišení RSM textury), za pomoci kterých je mřížka inicializována. U tohoto parametru je vynechán sloupec propagace a to z důvodu, že počet VPL nemá na samotnou propagaci vliv. V tabulce 7.8 jsou vyneseny časy pro různé rozlišení. Je vidět, že rozlišení *512*512* pixelů už značně prodlouží dobu inicializace mřížky. I z tohoto důvodu výsledná aplikace pracuje s rozlišením RSM *256*256* pixelů.

Pokud srovnáme výkon obou grafických karet je rozdíl viditelný na první pohled. Grafická karta NVIDIA dosahuje řádově lepších výsledků – dokonce i za použití atomických operací je rychlejší než konkurenční karta od AMD.




Obrázek 7.1: Porovnání atomických operací s druhým přístupem.

## 7.2 Vizuální kvalita



Obrázek 7.2: Výsledná scéna – nepřímé osvětlení bylo zesíleno, aby jej bylo lépe vidět.

Kromě samotného výkonu je druhým důležitým aspektem výsledná vizuální kvalita výstupu. Hlavní parametry ovlivňující výslednou kvalitu jsou následující:

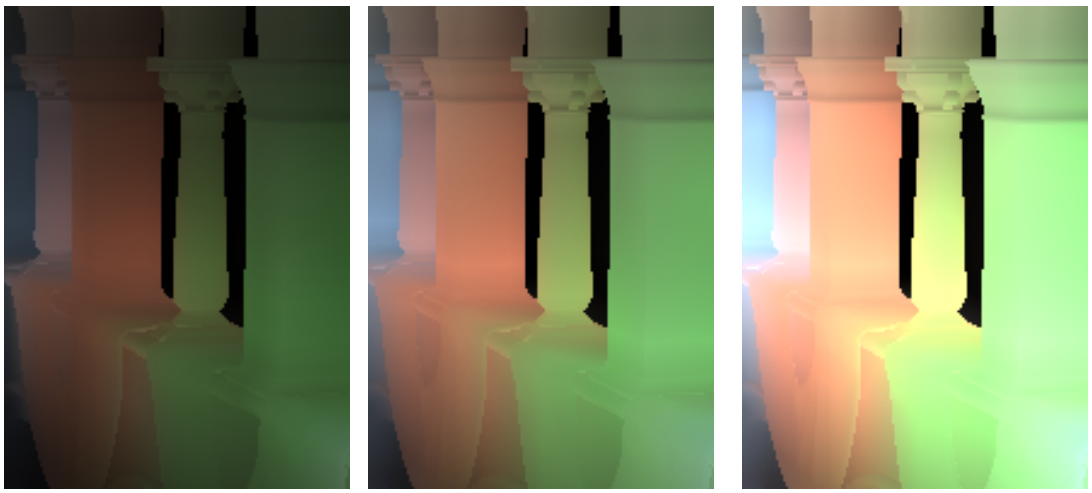
- Velikost buňky – pokud je velikost buňky větší potom i dosah, kam se světlo rožšíří je vetší (jelikož se šíří po buňkách).
- Počet iterací – tento parametr ovlivňuje kolikrát bude provedena propagace, vetší počet iterací poskytuje kvalitnější výsledky, ale negativně ovlivňuje výkon (viz kapitola 7.1).
- Počet mřížek v kaskádě – pokud zvýšíme počet mřížek v kaskádě, opět dostaneme o něco kvalitnější výstup (blízko kamery je nejvíce detailů).
- Rozlišení RSM (počet VPL) – čím více VPL, tím více zdrojů ze kterých se šíří osvětlení a tím lepší výsledná kvalita obrazu.
- Blokující geometrie – díky ní se světlo respektuje překážky a nešíří se skrze ni.

Pro samotné testování byl použit jeden ze standardních modelů pro testování (nejen) metod globálního osvětlení *Sponza Atrium*. Jedná se o upravenou verzi od německé firmy *Crytek* (počet vrcholů je uveden v tabulce 7.9), která do původního modelu přidala například závěsy do oblouků, závěsné květináče atp.

| **Název** | **Počet vrcholů** | **Počet trojúhelníků** | **Formát** |
|---|---|---|---|
| Crytek Sponza | 184 330 | 262 267 | obj |

Tabulka 7.9: Vybraná scéna.

Pro většinu následujících obrázků bylo použito (pokud není zmíněno jinak) 3 kaskád mřížek o velikosti $32^3$ pro propagaci světla, spolu s rozlišením RSM *256*256* pixelů, 8 kroků propagace a stínové mapy o rozlišení *2048*2048* pixelů. Na obrázku 7.2 lze vidět výslednou scénu s aplikovaným nepřímým osvětlením. Samotnou výslednou intezitu nepřímého osvětlení lze vidět na obrázku 7.6. Samotný vliv nepřímého osvětlení na výsledné vnímání scény je pak zobrazen na obrázku 7.7.



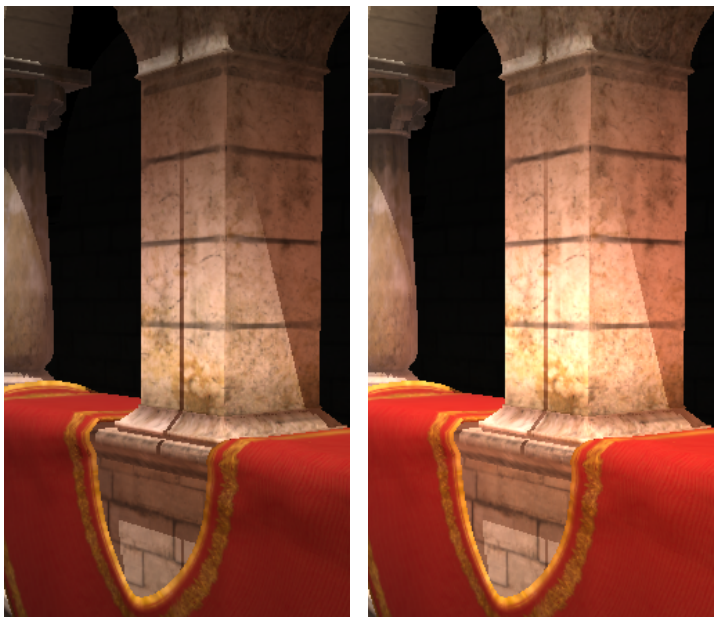
(a) RSM textura velikosti *128*128* px.

(b) RSM textura velikosti *256*256* px.

(c) RSM textura velikosti *512*512* px.

Obrázek 7.3: Porovnání výsledné intenzity pro různý počet VPL.

Pokud použijeme pro inicializaci mřížky více VPL, tak tím docílíme toho, že se světlo bude šířit ze více zdrojů a dostaneme tak větší hodnotu výsledné intenzity, jak je ukázáno na obrázku 7.3. Po provedení sérií testů jsem dospěl k závěru, že rozumný kompromis mezi výkonem a kvalitou je nastavení rozlišení RSM na hodnotu *256*256* pixelů. Použití více VPL na testovací scéně vedlo k přepalování obrazu (na obrázku bílá místa s vysokou intenzitou) a bylo nutné pak upravovat množství nepřímého osvětlení, čímž tento krok ztrácel význam. Ovšem je nutno podotknout, že se jedná o případ konkrétní scény. Na jiných či větších scénách se může výsledek lišit a zejména v případě rozlehlých scén bude nutný vyšší počet VPL.



(a) 1 mřížka ($64^3$). (b) Kaskáda 3 mřížek ($32^3$).

Obrázek 7.4: Porovnání výsledného osvětlení pro 1 mřížku a kaskádu 3 mřížek.

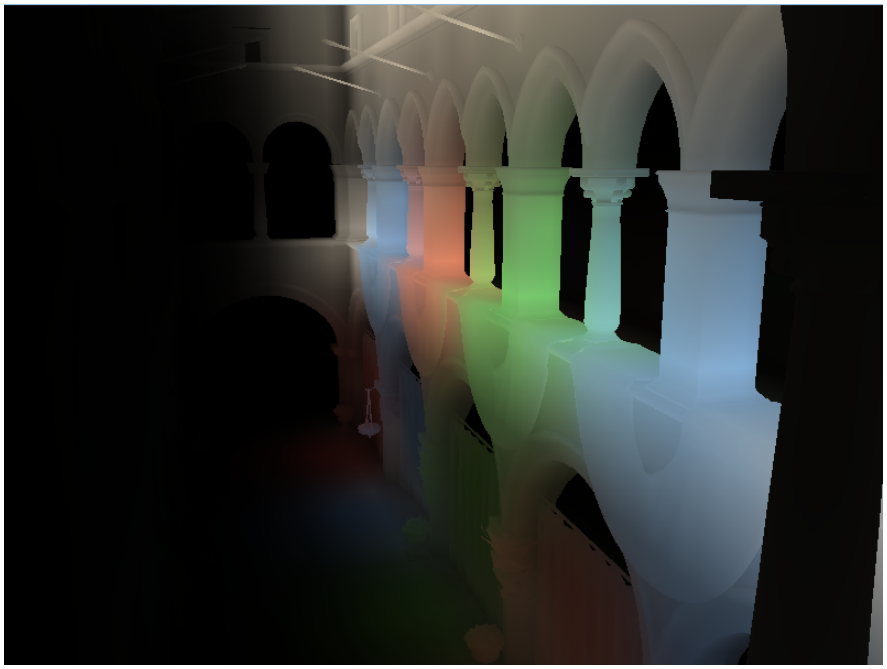
(a) Bez blokující geometrie. (b) S blokující geometrií.

Obrázek 7.5: Vizuální porovnání vlivu blokující geometrie.

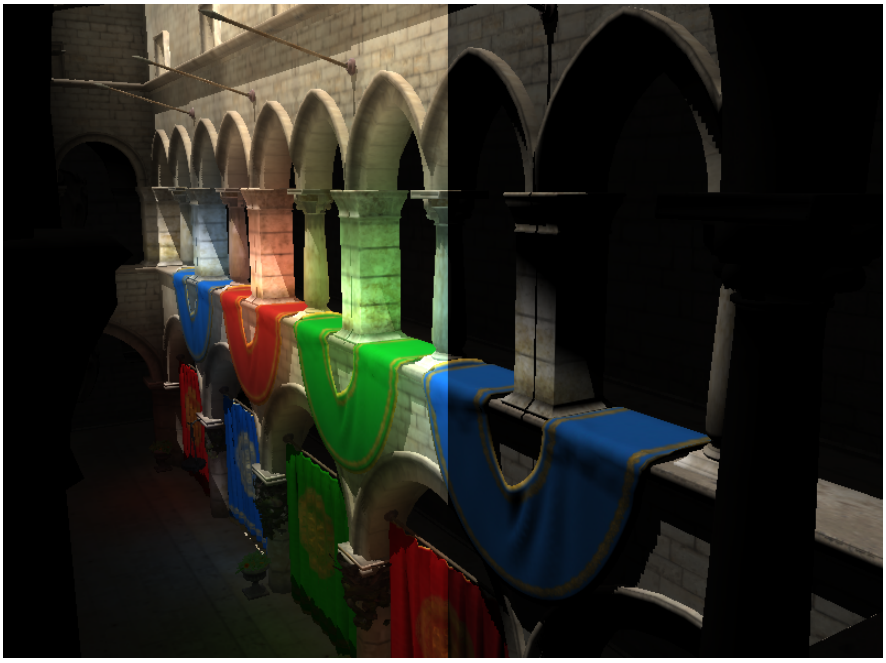
Dalším faktorem ovlivňující výslednou kvalitu je použití kaskády více mřížek oproti

jedné velké mřížce. Tím docílíme toho, že u pozorovatele je největší míra detailů. Bylo celkem složité najít místo, odkud by byl rozdíl patrný, proto jsem zvolil pilíř, kde je efekt nejviditelnější. Samotný rozdíl je zobrazen na obrázku 7.4 a to pokud se podíváme například na přední stranu pilíře, vidíme, že výsledná intenzita je vetší – světlejší místo na pilíři. Dále si můžeme všimnout, že také intenzita v půlkruhové výseči přehozu je vyšší – více červené barvy. V kapitole 6.4.1 jsme si popsali, že je důležité posouvat mřížku o násobky velikosti buňky za účelem snížení problikávání. I přesto, že je takovýto pohyb zajištěn, je problikávání občas viditelné.

Nyní se podíváme, jak se změní výsledný vzhled scény při použití blokující geometrie. Tuto změnu ilustruje obrázek 7.5. Je možné vidět, že bez použití blokující geometrie se intenzita propaguje i skrze sloup i skrze závěs. Toto chování je potlačeno právě blokující geometrií, kdy je vidět, že intezita se šíří pouze vedle a pod závěsem a dokonce je vidět i stín, který závěs vrhá. Toto chování by se dalo ještě vylepšit tím, že by se vytvářela textura obsahující blokující geometrii i z pohledu kamery (viz kapitola 5.2).



Obrázek 7.6: Finální intenzita světla po 8 iteracích – opět zesvětlená pro lepší viditelnost.



Obrázek 7.7: Vliv nepřímého osvětlení na zobrazení scény. Vlevo scéna s nepřímým osvětlením a vpravo bez něj.

# Kapitola 8

# Závěr

Cílem práce bylo seznámit se s problematikou výpočtu nepřímého osvětlení v reálném čase. Za tímto účelem byly nastudovány metody *Reflective Shadow Maps* a *Light Propagation Volumes* pro výpočet nepřímého osvětlení ve scéně (viz kapitola 4 a 5). Dále byl naimplementován algoritmus *Cascaded Light Propagation volumes*, což je rožšíření algoritmu *Light Propagation Volumes*, které využívá kaskádu více pohybujících se mřížek za účelem zvýšení výsledné kvality výstupu. V práci byly popsány a naimplementovány dva možné způsoby plnění mřížky (kapitola 6.4.2). První z nich využívá aditivního blendingu a přepínání vrstvy 3D textury v geometry shaderu, kdežto druhý se spoléhá na atomické operace nad obrazem v shaderech. Nutno podotknout, že druhá varianta využívá několik rozšíření přístupných pouze pro grafické karty NVIDIA. V rámci testování popsaného v kapitole 7 bylo provedeno měření výkonu algoritmu s různými parametry na dvou sestavách a bylo provedeno vizuální porovnání pro několik parametrů. Dále bylo v rámci testování provedeno srovnání výkonu obou přístupů plnění mřížky.

Výsledky experimentů ukázaly, že při použití kaskády mřížek lze dosáhnout zrychlení v řádu milisekund (8 až 15ms dle metody plnění) oproti případu, kdy je použita jedna velká mřížka. Dále se ukázalo, že použití blokující geometrie má relativně velký dopad výsledný výkon (v řádu 0.3 až 10ms). Další zajímavý fakt, který z testů vyplunul je ten, že algoritmus je výrazně rychlejší na kartě NVIDIA než na konkrureční výkonově srovnatelné kartě od AMD. Osobně se domnívám, že tento fakt je způsobený tím, že NVIDIA má novější architekturu jádra a umí efektivněji kartu využívat.

Pokud se budeme bavit o výsledné vizuální kvalitě, tak nejlepšího výsledku dosahuje algortimus právě při použití kaskády mřížek. Dalším parametrem, který z velké části ovlivňuje výslednou kvalitu výstupu, je použití blokující geometrie. Ovšem s použitím kaskád mřížek přichází i problém s problikáváním při pohybu. Ačkoli byl v rámci práce naimplementován pohyb po celých násobcích velikosti buňky, nepodařilo se tento jev zcela eliminovat.

Možností pokračování v práci se nabízí hned několik. Už v původním článků se nachází několik zajímavých možností, jak projekt rozšířit. Zejména o vícenásobný odraz nepřímého osvětlení, což by obnášelo úpravu algoritmu propagace. Dále pak rozšíření o nepřímé spekulární odrazy (odlesky) nebo simulaci média, ve kterém se světlo šíří. Pokud pomineme samotný algoritmus *Light Propagation Volumes*, tak by práci určitě pomohl lepší výpočet stínů jako například *Cascaded Shadow Maps* nebo by bylo vhodné práci rožšířit o podporu všesměrových světelných zdrojů.

# Literatura


[1] Bunnel, M.: Dynamic Ambient Occlusion and Indirect Lighting. In *GPU Gems 2*, Addison-Wesley Professional, 2005, ISBN 0321335597.

[2] Crassin, C.; Green, S.: Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer. In *OpenGL Insights*, editace P. Cozzi; C. Riccio, CRC Press, July 2012, ISBN 978-1439893760, s. 303–317.
URL http://www.openglinsights.com/

[3] Crassin, C.; Neyret, F.; Sainz, M.; aj.: Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, ročník 30, Wiley Online Library, 2011, s. 1921–1930.

[4] Dachsbacher, C.; Stamminger, M.: Reflective Shadow Maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, New York, NY, USA: ACM, 2005, ISBN 1-59593-013-2, s. 203–231, doi:10.1145/1053427.1053460.
URL http://doi.acm.org/10.1145/1053427.1053460

[5] Kajiya, J. T.: The Rendering Equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, New York, NY, USA: ACM, 1986, ISBN 0-89791-196-2, s. 143–150, doi:10.1145/15922.15902.
URL http://doi.acm.org/10.1145/15922.15902

[6] Kaplanyan, A.: Light Propagation Volumes in CryEngine 3 [online]. 2009, [cit. 25. 5. 2015].
URL http://www.crytek.com/download/Light_Propagation_Volumes.pdf

[7] Kaplanyan, A.; Dachsbacher, C.: Cascaded Light Propagation Volumes for Real-time Indirect Illumination. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, New York, NY, USA: ACM, 2010, ISBN 978-1-60558-939-8, s. 99–107, doi:10.1145/1730804.1730821.
URL http://doi.acm.org/10.1145/1730804.1730821

[8] Keller, A.: Instant Radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, ISBN 0-89791-896-7, s. 49–56, doi:10.1145/258734.258769.
URL http://dx.doi.org/10.1145/258734.258769

[9] Kessenich, J.; Baldwin, D.; Rost, R.: The OpenGL Shading Language [online]. 2014, [cit. 25. 5. 2015].
URL https://www.opengl.org/registry/doc/GLSLangSpec.4.50.pdf

[10] Kirsch, A.: Light Propagation Volumes - Annotations [online]. 2010, [cit. 23. 5. 2015].
URL
http://blog.blackhc.net/wp-content/uploads/2010/07/lpv-annotations.pdf

[11] Milet, T.: Shadows, Visibility and Interactive Global Illumination [online]. 2014, [cit. 23. 5. 2015].
URL https://www.fit.vutbr.cz/study/courses/PGP/private/lectures/PGP-Interactive_GI.pdf

[12] Mittring, M.: Finding next gen: Cryengine 2 [online]. 2007, [cit. 25. 5. 2015].
URL http://developer.amd.com/wordpress/media/2012/10/Chapter8-Mittring-Finding_NextGen_CryEngine2.pdf

[13] Ramamoorthi, R.; Hanrahan, P.: On the relationship between radiance and irradiance: determining the illumination from images of a convex Lambertian object. *JOSA A*, ročník 18, č. 10, 2001: s. 2448–2459.

[14] Ritschel, T.; Dachsbacher, C.; Grosch, T.; aj.: The State of the Art in Interactive Global Illumination. *Comput. Graph. Forum*, ročník 31, č. 1, Únor 2012: s. 160–188, ISSN 0167-7055, doi:10.1111/j.1467-8659.2012.02093.x.
URL http://dx.doi.org/10.1111/j.1467-8659.2012.02093.x

[15] Segal, M.; Akeley, K.: The OpenGL Graphics System: A Specification [online]. 2014, [cit. 25. 5. 2015].
URL https://www.opengl.org/registry/doc/glspec45.core.pdf

[16] Sloan, P.-P.: Stupid Spherical Harmonics (SH) Tricks [online]. 2008, [cit. 25. 5. 2015].
URL http://www.ppsloan.org/publications/StupidSH36.pdf

# Příloha A

# Obsah CD

Přiložený datový nosič obsahuje následující věci:

- Adresář `bin` – obsahuje spustitelnou aplikaci pod operačním systémem *Microsoft Windows* včetně `.dll` knihoven.
- Adresář `src` – obsahuje zdrojové kódy, projekt a solution pro *Microsoft Visual Studio*, dále obsahuje všechny potřebné knihovny a hlavičkové soubory.
- Adresář `text_src` – obsahuje zdrojové kódy textové části pro systém LaTeX.
- Adresář `videos` – obsahuje ukázkové videa.
- `Thesis.pdf` – tento dokument v elektornické verzi.
- `README.txt` – stručný popis ovládání aplikace a nastavení projektu pro *Microsoft Visual Studio*.