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

HIERARCHICKÉ TECHNIKY PRO VÝPOČET OSVĚTLENÍ

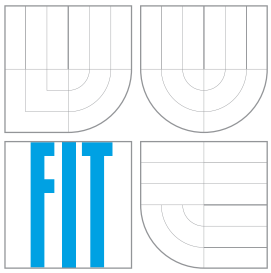
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ LIGMAJER

BRNO 2012

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

HIERARCHICKÉ TECHNIKY PRO VÝPOČET OSVĚTLENÍ

HIERARCHICAL TECHNIQUES IN LIGHTING COMPUTATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ LIGMAJER

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN NAVRÁTIL

BRNO 2012

Abstrakt

Tato diplomová práce se věnuje studiu a popisu hierarchických technik pro výpočet globálního osvětlení. Vysvětluje proč je dobré se zabývat hierarchickými technikami pro výpočet osvětlení a ukazuje postup, jak zakomponovat tyto hierarchické techniky do výpočtu radiozity v reálném čase a následné rozšíření pro výpočet dynamického osvětlení z plošných světelných zdrojů. Tyto dvě techniky jsou podrobně popsány v první části této práce. V druhé části je uveden návrh a implementace aplikace, která bude provádět výpočet dynamického osvětlení z plošných světelných zdrojů.

Abstract

This master thesis deals with description of hierarchical techniques in global lighting computation. Here is explaining the importance of hierarchical techniques in lighting computation and shows method, how to use these hierarchical techniques in realtime radiosity and its extension to dynamic area lighting. These two techniques are described in detail in the first part of this project. In the other part is desing and implementation of application for dynamic area lighting computation.

Klíčová slova

hierarchický, technika, osvětlení, radiozita, přímé, nepřímé, globální, voxelizace, světlo, interaktivní, výpočet, viditelnost, dynamický, nespojitosti, realistický, realtime, vertex, fragment, shader

Keywords

hierarchical, technique, lighting, radiosity, direct, indirect, global, voxelization, light, interactive, computation, visibility, dynamic, discontinuities, realistic, realtime, vertex, fragment, shader

Citace

Jiří Ligmajer: Hierarchické techniky
pro výpočet osvětlení, diplomová práce, Brno, FIT VUT v Brně, 2012

Hierarchické techniky pro výpočet osvětlení

Prohlášení

Prohlašuji, že jsem tento diplomový projekt vypracoval samostatně pod vedením Ing. Jana Navrátila. Uvedl jsem všechny publikace, z kterých jsem čerpal.

.....
Jiří Ligmajer
21. května 2012

Poděkování

Děkuji Ing. Janu Navrátilovi za cenné rady a připomínky při psaní mé diplomové práce a při návrhu a implementaci aplikace. Rovněž děkuji všem, kteří mě při této práci podporovali a pomáhali.

Obsah

Kapitola 1

Úvod

Většina našeho vnímání reality je dáná tím, jak nám ji dokáže zprostředkovat náš vizuální smyslový systém. Tento systém nám dovoluje rozpoznávat svět kolem nás pomocí světla v prostředí, v kterém se každodenně pohybujeme. Ještě před tím, než se světlo dostane do našeho oka a my jsme tak schopni rozpoznat okolí kolem nás, může se světlo odrážet, lomit nebo dokonce i částečně pohltit. Tímto sbírá světlo informaci o prostředí, v kterém se nacházíme. Informaci obsaženou ve světle je naše oko schopno rozpoznat a pomocí mozku reprodukovat vizuální vjem, rozpoznávat objekty kolem nás, vnímat jejich barvu, zjišťovat vztahy mezi jednotlivými objekty, ale také odhadnout povrch a materiál, z kterého jsou vytvořeny.

Od pradávna se lidé pokoušeli vytvářet co nejlepší obrazy světa pomocí různých druhů umění. V dnešní době se stále více rozvíjí cesta k co nejlepšímu vytvoření realistických obrazů pomocí výpočetní techniky. K tomu je potřeba dobře znát a modelovat průchod světla umělým prostředím. Simulovat vlastnosti světla, jako jsou odraz, lom, pohlcení, rozptyl a další, je jednoduché, ale náročné na výpočet. Navzdory těmto problémům již existují aplikace, které dokáží sestavit obraz prostředí pomocí těchto metod.

V dnešní době se v počítačové grafice klade stále větší důraz na generování interaktivních realistických obrazů prostředí, kdy generovaní jednoho obrazu trvá přibližně 30 milisekund. Takovýto přístup umožnuje uživatelům pohybovat se v zobrazovaném prostředí a pohybovat s objekty v reálném čase. Není tedy ničím překvapivým, že většina vývoje v počítačové grafice se zaměřuje právě na vytváření fotorealistických obrazů v reálném čase. Jednou z možností, jak docílit tohoto nastoleného trendu, je použití hierarchických technik pro výpočet globálního osvětlení, kterým se tato práce věnuje. Tyto techniky pracují kompletně pouze s obrazem scény, který vidí pozorovatel, proto nezahazují výkon počítače na výpočet nepotřebných věcí. Další nespornou výhodou je, že jsou počítány kompletně s využitím grafické karty, a jak je známo, dnešní grafické karty mají hodně výkonu na rozdávání. Všechny tyto výhody umožňují počítat globální osvětlení ve scéně v reálném čase.

Práce je dělena do sedmi kapitol. První kapitolou je samotný úvod, kde jsem nastínil, proč je výhodné se věnovat právě tématům výpočtu osvětlení pomocí hierarchických technik.

Druhá kapitola pojednává o základních přístupech výpočtu osvětlení v počítačové grafice, popisuje jejich výhody a nevýhody a dále se zaměřuje na výpočet globálního osvětlení v umělé scéně.

Ve třetí kapitole se budu zabývat hierarchickými technikami, z kterých vychází mnou nastudovaná a implementovaná metoda výpočtu přímého osvětlení v kompletně dynamických scénách s plošnými zdroji světla. První z nich je hierarchická radiozita, druhou je vý-

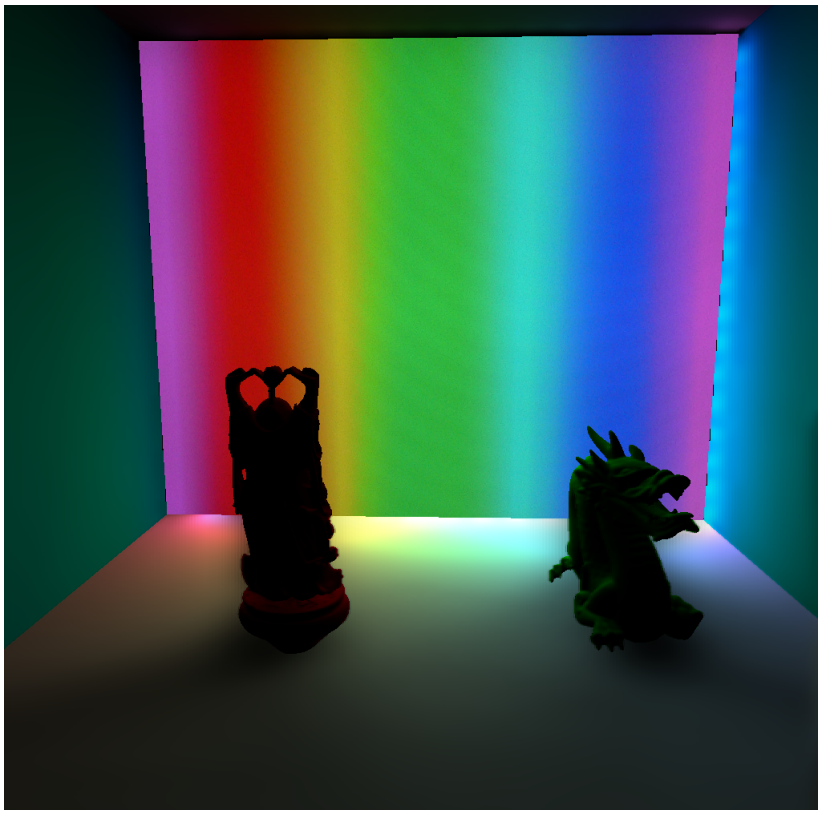
počet dynamického osvětlení z plošných zdrojových světel, která vychází právě z poznatků hierarchické radiozity.

Čtvrtou kapitolou je popis a návrh scény a programu. Tento program bude demostrovat mnou implementovanou techniku výpočtu dynamického osvětlení z plošných zdrojů světel, jenž jsou založeny na hierarchických technikách.

Pátá kapitola obsahuje detailní implementaci hierarchické techniky, která je využita k výpočtu přímého osvětlení a viditelnosti z plošných světelných zdrojů. Tato je natolik rychlá, že se dá použít v interaktivních aplikacích.

V šesté kapitole se budu zabývat hodnocením mnou implementované hierarchické techniky a více proberu problémy, které nastali při implementaci mnou zvolené hierarchické techniky.

Poslední kapitolou je závěr, kde shrnu všechny podstatné věci, které jsou potřeba k výpočtu přímého osvětlení. Zhodnotím výhody a nevýhody mé vybrané techniky pro výpočet přímého osvětlení a uvedu další postup práce.



Obrázek 1.1: Plošná světla přidávají do aplikací počítačové grafiky obrovský realismus. Nadruhou stranu je složité spočítat osvětlení a viditelnost z těchto světel.

Kapitola 2

Problematika výpočtu osvětlení

V této kapitole se budu zabývat problematikou výpočtu osvětlení. Výpočet osvětlení pro daný bod ve scéně se dá rozdělit do dvou variant. První variantou je lokální osvětlení, druhou je globální osvětlení.

U obou variant je princip výpočtu stejný. Každý pixel ve výsledném obraze scény zobrazuje nějaký prostorový 3D bod v této scéně. Pro tento 3D bod jsme schopni pomocí nějaké lokální nebo globální metody spočítat osvětlení a tím rozhodneme a určíme, jakou barvu bude mít aktuálně zpracovávaný pixel.

Nejprve se zaměřím na lokální variantu výpočtu osvětlení. Následně popíši princip globální varianty.

2.1 Lokální metody výpočtu osvětlení

Z úvodu této kapitoly víme, že každý pixel obrazu scény představuje nějaký 3D bod ve scéně. Tato nejjednodušší metoda spočívá ve sbírání lokálních informací tohoto 3D bodu a použití těchto informací k výpočtu osvětlení. Lokální informace bodu, pro který počítáme osvětlení, jsou povrchová normála v tomto bodě, pozice všech světelných zdrojů ve scéně a směr, kterým se na tento bod dívá pozorovatel.

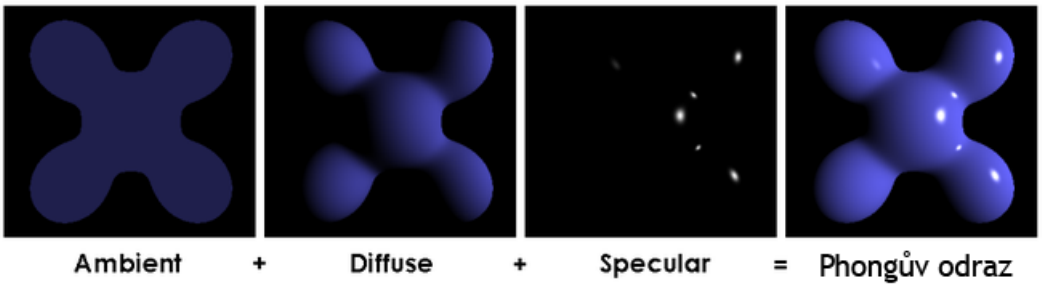
Světelné zdroje jsou v případě lokálních metod také velice jednoduché. Ve skutečnosti mají všechny světelné zdroje menší či větší plochu a vydávají různě silné záření. Pro případy lokálních metod se musíme spokojit s tím, že každý světelný zdroj je reprezentován nekonečně malým bodem. Dalším problémem těchto metod je, že každá plocha ve scéně, která je přivrácená k tomuto bodovému zdroji světla, od něj získává osvětlení. Toto osvětlení ovšem nemusí být validní, protože mezi světelným zdrojem a osvětlovanou plochou může ležet jiný objekt, který tento světelný zdroj zastíní. Díky těmto omezením pozbývají lokální metody výpočtu osvětlení schopnosti aproximovat stíny, odrazy a lom světelných paprsků. Všechny tyto omezení jsou totiž potřeba k přidání rozhodující úrovně realismu.

Lokální metody výpočtu osvětlení používají následující stínovací techniky objektů. Jsou to, konstantní stínování, Gouraudovo stínování [3] a Phongovo stínování [13]. Tyto lokální osvětlovací metody jsou celkem účinné, ale jejich výsledky jsou zdaleka nedostačující. Ve skutečnosti světlo, které dopadá na bod v prostoru, nenese pouze informaci o přímém osvětlení, tedy o osvětlení ze zdroje světla k tomuto bodu, ale také informaci o nepřímém osvětlení, které dodává světlo, jež se odráží ve scéně a dopadá do tohoto bodu. Dále, některé plochy ve scéně mohou blokovat světelné paprsky a vytvářet tak stíny ve scéně. Tyto stíny jsou potom hlavním zdrojem našeho vjemu, že výsledný obraz zobrazuje prostorovou scénu,

nikoli pouze obyčejný 2D prostor.

Phongův stínovací model

Tento empirický model je založen na výpočtu osvětlení v každém bodě zobrazovaného objektu. Pro tento výpočet je použito aproximace odrazu světelného paprsku podle Phongova odrazového modelu. Phongův odrazový model popisuje, jak povrch objektu odráží světlo na matných površích (difúzní odraz) a na lesklých površích (zrcadlový odraz). Tento odrazový model také aproximuje nepřímé osvětlení přidáním ambientního koeficientu, který určuje množství světla rozptýleného po celé scéně (obrázek 2.1[1]).



Obrázek 2.1: Ukázka jednotlivých složek Phongova stínovacího modelu.

Rovnice, která vypočítá barvu objektu pomocí Phongova stínovacího modelu, je (2.1):

$$I = k_a + \sum_{i \in L} (\vec{L_i}.\vec{N}) k_d + (\vec{R}.\vec{V})^n k_s \tag{2.1}$$

Kde:

- k_a ambientní konstanta materiálu
- k_d difůzní konstanta materiálu
- k_s odrazivá konstanta materiálu
- $\vec{L_i}$ vektor od zdroje světla k stínovanému bodu
- $\vec{N}$ normála povrchu ve stínovaném bodě
- $\vec{R}$ odražený vektor paprsku podle normály
- $\vec{V}$ vektor od pozorovatele k stínovanému bodu
- n ostrost odlesku materiálu

2.2 Globální metody výpočtu osvětlení

K výpočtu fyzikálního modelu globálního osvětlení je třeba co nejpřesněji aproximovat chování světla ve skutečném světě. Tato podkapitola se zaměřuje na popis fotometrických veličin a zobrazovací rovnice (2.4), v které se tyto fotometrické veličiny používají.

[1] Obrázek převzat z wikipedia.org

Ve skutečnosti, světlo dopadající na bod v prostoru zahrnuje jak přímé, tak i nepřímé osvětlení. Přímé osvětlení je reprezentováno světlem, které dopadá na bod v prostoru přímo ze světelného zdroje. Nepřímé osvětlení je tvořeno světlem, které se nejprve odrazí od jednoho nebo více povrchů. Další povrchy potom mohou blokovat dopadající světlo a tím produkují stíny. K vytvoření těchto efektů je třeba znát všechny objekty ve scéně a umět rozpoznat, jak mohou ovlivnit příchozí světelné paprsky. Globální osvětlení přidává základní stupeň realismu do zobrazovaného obrazu, tudíž je třeba se zaměřit na jeho co nejefektivnější výpočet.

Fotometrické veličiny

Každé elektromagnetické záření má energii, která se ze zdroje přenáší do okolního prostoru. Například žárovka svítí a světlo z ní se šíří prostorem. Účinek záření však vnímáme teprve při jeho dopadu na povrch tělesa. Těleso je osvětlené a můžeme ho vidět naším zrakem. Energii vyzářeného světla posuzujeme buď subjektivně na základě učinků na zrak, nebo použijeme vhodné měřící přístroje. Měřením v této oblasti se zabývá fotometrie [8].

Ve fotometrii je definována řada fyzikálních veličin, kterými jsou popisovány vlastnosti zdrojů světla, přenos světla volným prostorem a děje spojené s dopadem světla na osvětlené předměty. Z těchto fotometrických veličin si čtyři popíšeme podrobněji, neboť tyto veličiny se vyskytují v zobrazovací rovnici pro výpočet globálního osvětlení.

Základní jednotkou fotometrie je zářivá energie. Označuje se E_z a je měřená v joulech $[J]$. Popisuje energii přenášenou světelnými vlnami od zdroje k ploše, na kterou světlo dopadá.

Světelný tok (2.2) vyjadřuje intenzitu zrakového vjemu normálního lidského oka, který vyvolá záření o dané energii vyzářené světelným zdrojem za jednotku času (popř. procházející za jednotku času určitou plochou). Světelný tok se označuje Φ a měří se ve watech [W].

$$\Phi = E_z t \tag{2.2}$$

Další jednotkou je osvětlení (2.3). Osvětlení určuje účinky světla při jeho dopadu na povrch tělesa. Závisí na části světelného toku $\Delta\Phi$, který dopadá na plochu o obsahu ΔS. Je označováno jako E a jednotkou je $\left[\frac{W}{m^2}\right]$.

$$E = \frac{\Delta\Phi}{\Delta S} \tag{2.3}$$

Svítivost je světelný tok, který dopadá na plochu pod nějakým prostorovým úhlem za jednotku času. Označuje se I a udává se v jednotkách $\left[\frac{W}{sr \cdot m^2}\right]$. Jednoduše řečeno svítivost popisuje, jak moc energie opustí danou plochu pod nějakým úhlem. Toto množství energie může být použito k popsání, jak vypadá povrch objektu, pokud se na něj díváme z určitého úhlu.

BRDF

Vzhled povrchu poznáme až tehdy, pokud na něj dopadá světlo. Toto světlo se nějak odráží a dává nám informaci o tom, jak vypadá povrch objektu, od kterého se odrazilo. Povrch, který perfektně odráží všechny příchozí světelné paprsky, se bude jevit jako zrcadlový. Naopak povrch, který odráží všechny dopadající světelné paprsky do všech směrů stejně

nezávisle na tom, pod jakým úhlem paprsky dopadají, se bude jevit jako difúzní povrch (např. dřevo). V metodách lokálního osvětlení byly povrchy buďto úplně difúzní nebo úplně zrcadlivé.

V reálném světě jsou ovšem materiály jen zřídkakdy úplně difúzní nebo zrcadlivé. K lepšímu popisu odrazivých vlastností materiálu nám slouží funkce BRDF (Bidirectional Reflectance Distribution Function) [12]. BRDF funkce definuje, jak velká energie světla se odrazí po směru odrazu z příchozího směru v nějakém bodě.

Tato funkce se stala všudypřítomnou metodou k definování odrazivých vlastností materiálů tak, aby odpovídali skutečnosti. Bylo navrženo mnoho různých BRDF modelů. Některé z nich jsou Cook-Torrance [1], Ward [19] nebo Lafortune [6]. Všechny tyto vyjmenované modely jsou založeny na empirických datech, které byly změřeny studováním materiálů reálného světa. Předpokladem BRDF funkce je, že dopadající světelný paprsek se vždy odrazí od povrchu v tom samém bodě, do kterého dopadne. Ve skutečnosti světelný paprsek dopadající na povrch nějakého tělesa se pod povrchem různě rozptýlí a odrazí se v jiném bodě, než v tom, do kterého dopadl. Tento princip šíření světelných paprsků popisuje BSSRDF (Bidirectional Scattering Surface Reflectance Distribution Function).

Zobrazovací rovnice globálního osvětlení

Zobrazovací rovnice (2.4), jak je definována v [4], se používá v grafických aplikacích, aby bylo dosaženo co největšího fotorealizmu. Pomocí definice fotometrických veličin v podkapitole 2.2 a BRDF distribuční funkce v podkapitole 2.2 můžeme popsat zobrazovací rovnici pro výpočet globálního osvětlení. Tato rovnice nám říká, jak se světlo pohybuje ve scéně, jak se odráží, lomí a pohlcuje. Výsledkem této rovnice je světelná energie, která vychází z nějakého bodu scény x v daném směru $\vec{\omega_o}$:

$$L_o(x, \vec{\omega_o}) = L_e(x, \vec{\omega_o}) + \int_\Omega L_i(x, \vec{\omega_i}) f_r(x, \vec{\omega_i} \rightarrow \vec{\omega_o})(\vec{\omega_i}.\vec{n}) d\vec{\omega_i} \tag{2.4}$$

Kde:

- $L_o(x, \vec{\omega_o})$ výsledek rovnice, světelná energie, která vychází z bodu x ve směru $\vec{\omega_o}$
- $L_e(x, \vec{\omega_o})$ světelná energie emitovaná ze samotného povrchu v bodě x ve směru $\vec{\omega_o}$
- $L_i(x, \vec{\omega_i})$ dopadající světelná energie do bodu x ve směru $\vec{\omega_i}$
- $f_r(x, \vec{\omega_i} \rightarrow \vec{\omega_o})$ je BRDF funkce, která definuje množství světelné energie odražené ve směru $\vec{\omega_o}$ v bodě x
- $(\vec{\omega_i}.\vec{n})$ je útlum dopadající světelné energie založená na úhlu mezi směrem dopadu a normálou povrchu

Kapitola 3

Hierarchické techniky výpočtu globálního osvětlení

Algoritmy, které se používají k výpočtu osvětlení, potřebují k vytvoření správného výsledku několik globálních informací o scéně. Na druhou stranu ovšem neberou v úvahu, jak se tyto informace budou používat. Můžeme si vzít jako příklad radiositu. Tato technika globálního výpočtu osvětlení potřebuje ke správnému výpočtu výsledného obrazu scény znát všechny difúzní povrchy ve scéně. Avšak už nebere v potaz, z jakého úhlu se budeme na tyto povrchy dívat. Takovýmto algoritmům, jakým radiosita je, se říká, že pracují v objektovém nebo modelovém prostoru. Neberou tedy do úvahy, kde stojí pozorovatel a odkud se na scénu dívá. Stále více se v dnešní době zaměřujeme na vývoj algoritmů, které pracují v obrazovém prostoru. Obrazovým prostorem rozumíme to, co vidí pozorovatel ze své pozice. Tyto algoritmy používají jako vstup jeden nebo několik obrazů (textur), provádí nad nimi různé operace a jejich kombinací poté produkují výstupní obraz scény, který vidí pozorovatel.

Takovéto algoritmy se dají jednoduše hardwarově akcelerovat na dnešních grafických kartách, protože umožňují rychlý přístup a práci s texturami, které jsou nahráty v paměti grafické karty. Algoritmy pracující v obrazovém prostoru se tedy stávají velkou oblastí ve vývoji v počítačové grafice a technikách výpočtu globálního osvětlení. Abychom pochopili jejich principy, je třeba si uvést a vysvětlit, několik základních pojmů, které jsou pro tyto algoritmy charakteristické a dále rozvíjejí hierarchické techniky výpočtu globálního osvětlení [11] a [10].

3.1 Virtual point lights

Virtual Point Lights[1] (dále jen VPL) se používají při výpočtu osvětlení k aproximaci plošných světelných zdrojů. Každý plošný světelný zdroj se popíše určitým počtem VPL světel. Záleží na počtu světel, jež jsou použity k aproximaci, protože malý počet VPL bude mít nedostatky při popisu všech bodů plošného světla. Naopak velký počet VPL bude sice dobře popisovat plošný světelný zdroj, ale výpočet výsledného osvětlení bude pomalý. Je tedy třeba brát v úvahu, jak kvalitní výstup chceme produkovat.

VPL světla se nepoužívají jenom k popisu plošných světelných zdrojů, ale také k výpočtu nepřímého osvětlení ve scéně. Můžeme totiž všechny body difúzních povrchů ve scéně popsat

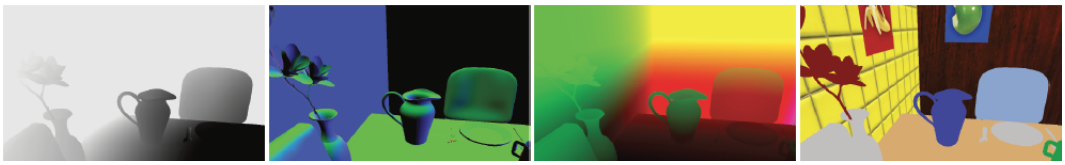
[1] Virtuální bodová světla vychází z překladu anglického Virtual Point Light. Představuje bodové světlo, které se používá k popisu vlastnosti plošného zdroje.

jako sekundární zdroje světla, které ve skutečnosti odráží dopadající světlo z primárních světel ve scéně. Každý sekundární zářič může být opět aproximován jedním VPL světlem.

VPL světlo musí o sobě obsahovat určité informace, stejně jako původní světla ve scéně. Tyto informace jsou, pozice světla ve scéně, jeho barvu (svítivost) a normálu plochy, na které se toto VPL nachází.

3.2 G-buffer

Dnešní grafické karty umožňují velice efektivně pracovat s texturami. Dále také dokáží zapisovat do více textur najednou v jednom jediném průchodu. Této výhody se používá při výpočtu osvětlení v takzvaném algoritmu Deferred shading [7]. *G-buffer*[2] [16] (obrázek 3.1) je datová struktura, která obsahuje několik textur, do kterých se uloží informace potřebné k výpočtu osvětlení. Jsou to hloubka jednotlivých pixelů obrazu ve scéně, normála povrchu, kterou daný pixel zobrazuje, barva tohoto povrchu a 3D pozice pixelu ve scéně. Všechny tyto informace se dají vypočítat a uložit do G-bufferu v jednom jediném průchodu právě díky novodobým grafickým kartám. Nakonec je potřeba dodat, že G-buffer ukládá informace o scéně z pohledu pozorovatele.



Obrázek 3.1: G-buffer: zleva z-buffer, normály povrchu, pozice fragmentů v pohledovém prostoru a barva materiálu (difůzní složka). [11]

3.3 Reflective shadow map

Reflective Shadow Map[3] (dále jen RSM) je v podstatě G-buffer pouze s tím rozdílem, že obsahuje informace o scéně tak, jak ji vidíme z pohledu světelného zdroje. Pomocí této mapy poté můžeme vytvořit jednotlivá VPL světla, která budou pokrývat celou plochu RSM. Tyto VPL světla se v dalším průběhu výpočtu používají jako sekundární zářiče k výpočtu nepřímého osvětlení ve scéně. Problémem, takovéhoto výpočtu nepřímého osvětlení pomocí RSM, je pokud máme mnoho VPL světel. Jelikož v takovémto přístupu každé VPL světlo ovlivňuje všechny pixely výsledného obrazu, pak pomocí RSM vzorkování vzniká úzké hrdlo při výpočtu nepřímého osvětlení. Abychom se tomuto problému vyhnuli, do výpočtu osvětlení pro daný pixel započítáváme pouze VPL světla, která mají určitou vzdálenost. Všechny ostatní VPL světla jsou zahozena a do výsledného osvětlení pixelu se nezapočítávají. Toto ovšem velice snižuje výsledný jas obrazu oproti skutečnosti. Je proto třeba výsledný jas obrazu upravit.

[2] G-buffer, neboli geometrický buffer ukládá informace o geometrii scény. Jeho použití je vhodné při výpočtu osvětlení.

[3] Zrcadlová stínová mapa vychází z anglického Reflective Shadow Map a je to datová struktura tvořená několika texturami, které obsahují podobné informace jako G-buffer. S tím rozdílem, že scénu zobrazujeme z pohledu světla.

3.4 Hierarchická radiozita s interaktivním globálním osvětlením

Tato podkapitola se bude zabývat radiozitou počítanou pomocí obrazového prostoru. Algoritmy, které počítají výsledný obraz v obrazovém prostoru, mají často výhody v interaktivních aplikacích oproti algoritmům, které pracují v objektovém nebo modelovém prostoru. Výhody jsou zvláště v tom, že takovéto algoritmy neztrácejí čas s počítáním osvětlení, které je mimo viditelnou část scény. Počítají tedy pouze osvětlení v části scény, která je viditelná z pohledu pozorovatele.

Nyní si uvedeme jednotlivé body algoritmu, podle kterých se hierarchická radiozita počítá:

1. Vytvoření RSM a výpočet přímého osvětlení
2. Vytvoření vzorků VPL světel na základě RSM a jejich uložení do textury
3. Vygenerování mipmap k detekci nespojitostí
4. Použitím stencil bufferu označit pixely v každé úrovni mipmapy, kde je potřeba počítat osvětlení
5. Projít všechny stupně mipmapy a zkombinovat je do výsledného obrázku, který ponese informaci o nepřímém osvětlení
6. Složit dohromady přímé a nepřímé osvětlení a vygenerovat finální obraz scény

Těchto 6 kroků je podstatných pro výpočet hierarchické interaktivní radiozity. V následujících podkapitolách si popíšeme body 3 až 6. Body 1 a 2 jsme si popsali v podkapitolách 3.1 resp. 3.3.

Hloubkové nespojitosti

Nejprve si popíšeme vytvoření hloubkové mapy. Hloubková mapa je obsah z-bufferu, jehož hodnoty jsou převedeny na lineární hloubku a uloženy do textury. Následně pustíme nad touto mapou algoritmus, který vypočítá hloubkové nespojitosti na základní úrovni mipmapy. Poté budeme tuto základní úroveň opět procházet a spočítáme hloubkové nespojitosti na další nižší úrovni mipmapy. Takto budeme pokračovat, dokud nevygenerujeme všechny úrovně mipmapy. Hloubkové nespojitosti se spočítají porovnáním čtyř-okolí v aktuální úrovni hloubkové mipmapy. Vybere se vždy minimální a maximální hodnota a zapíše se do následující nižší úrovně mipmapy. Hloubková nespojitost je poté zjištěna tím, zda rozdíl mezi minimální a maximální hodnotou je vetší než námi zadaný práh. Problémy vznikají při výpočtu hloubkových nespojitostí, pokud se díváme na nějakou plochu ze šikma. Potom jsou na této ploše nekorekně detekovány nespojistosti. Abychom předešli tomuto problému, tak se nespojitosti počítají pomocí rovnice (3.1).

$$depthDerivation = \sqrt{\left(\frac{d_z}{d_x}\right)^2 + \left(\frac{d_z}{d_y}\right)^2} \tag{3.1}$$

Takto se nespojitosti vypočítají pouze na základní nejvyšší úrovni mipmapy. Následné počítání nižších úrovní mipmapy se provádí vybráním maximální hodnoty hloubkové derivace z čtyř-okolí aktuálně zpracovávaného pixelu. Následně pokud je maximální hloubková derivace vetší než námi zadaný práh, pak se v tomto místě vyskytuje hloubková nespojitost.

Později při výpočtu osvětlení je třeba převzorkovat na vyšší úroveň mipmapy pouze u vzorků, které obsahují hloubkovou nespojitost. Ta se určí porovnáním příslušných vzorků hloubkové derivace z nižší a vyšší úrovně mipmapy. Pokud je jejich rozdíl větší než předem daný práh, potom je potřeba osvětlení spočítat na vyšší úrovni mipmapy.

Normálové nespojitosti

V této podkapitole se zaměříme na výpočet a detekci normálových nespojitostí. Detekování normálových nespojitostí je méně přímočaré. Cílem je určit významné rozdíly v orientaci normál na povrchu objektu. Toto zjistíme tak, že se normála hodně mění alespoň v jedné její ose. Při generování mipmapy opět bereme čtyř-okolí pixelu a vybereme největší hodnotu v každé ze tří komponent normály. Tedy musíme vytvořit celkem tři mipmapy. Vždy pro každou komponentu jednu mipmapu. Tímto ovšem spotřebujeme mnoho paměti na grafické kartě. Proto je možné určit normálovou nespojitost podle zakřivení povrchu.

Mějme dvě normály $\vec{N_1}$ a $\vec{N_2}$ dvou sousedních pixelů. Zakřivení κ povrchu mezi těmito dvěmi normálami je určeno pomocí vzorce (3.2)

$$\kappa = 2 \sin \arccos \frac{\vec{N_1}.\vec{N_2}}{2} \tag{3.2}$$

Použitím tohoto vzorce můžeme normálové nespojitosti generovat stejně jako při hloubkových nespojitostech. Základní úroveň mipmapy se vygeneruje vždy vybráním maximální hodnoty zakřivení normál z čtyř-okolí pixelu. Další nižší úrovně počítáme vždy vybráním maximální hodnoty zakřivení z čtyř-okolí pixelu v předchozí úrovni. Stejně jako v předchozím případě, pokud maximální hodnota normálové nespojitosti je větší než hodnota prahu, potom v tomto místě existuje normálová nespojitost.

Určení hloubkových a normálových nespojitostí

Nejprve je potřeba převést námi vytvořenou mipmapu pro hloubkové a normálové nespojisti z hierarchické struktury do obyčejné velké textury. To provedeme tak, že vytvoříme texturu, do které postupně za sebe naskládáme všechny úrovně mipmapy. Takto dostaneme 2D obraz hloubkové a normálové mipmapy.

Jelikož v dnešních grafických kartách se vyskytuje stencil buffer, což je buffer, do kterého se dají nastavit různé celočíselné hodnoty na základě nějaké operace s jednotlivými pixely obrazu. Potom se tento stencil buffer dá použít pro označení hloubkových a normálových nespojitostí v našem vygenerovaném 2D obraze. Tedy nastavíme stencil bit tam, kde se nachází hloubková nebo normálová nespojitost a bude tedy třeba vypočítat osvětlení na vyšší úrovni mipmapy. Bity jednotlivých pixelů se nastavují podle následujícího pseudokódu:

```
for all (fragments f in quad) do
    if (for each i, f not in MipmapLevel(i)) then
        continue; // vzorek není v hierarchickém bufferu

    i = CurrentMipmapLevel(i);
    if (hasDepthOrNormalDiscontinuity(f, i)) then
        continue; // vzorek potřebuje rozdělení

    if (noDiscontinuity(f, i+1)) then
        continue; // vzorek je validní a nepotřebuje rozdělení
```

```
setStencil(f);
```

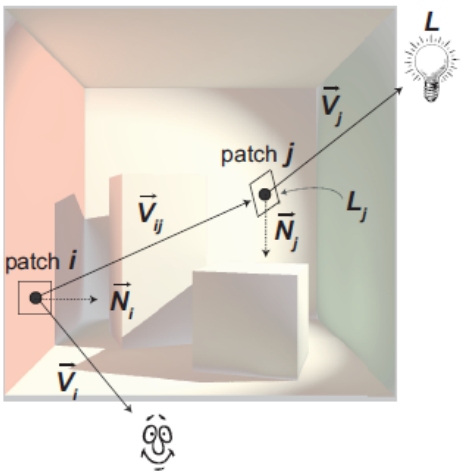
Takový přístup poznamenává, že potřebujeme počítat osvětlení pouze v pixelech, které nepotřebují rozdělení, ale příslušná nižší úroveň v mipmapě už toto dělení potřebuje. Po nastavení všech příslušných bitů v stencil bufferu bude výsledný 2D obraz mipmapy obsahovat bílé pixely na místech, které potřebují počítat osvětlení a černé pixely tam, kde se bude osvětlení interpolovat a kombinovat z již spočítaných pixelů v nižší úrovni mipmapy.

Shromažďování osvětlení ve vzorcích

V této podkapitole si popíšeme, jak se bude počítat a shromaždovat osvětlení jednotlivých vzorků hierarchického bufferu, které jsme si vypočítali v předchozím kroku.

V hierarchické radiozitě je nepřímé osvětlení shromažďováno nebo rozptýleno do pixelů během jednoduchého průchodu přes všechny pixely. V tomto algoritmu všechny VPL světla ve scéně ovlivňují všechny pixely výsledného obrazu. Nejprve je tedy třeba vytvořit texturu, do které uložíme informace o jednotlivých VPL světlech. Tyto informace jsou pozice VPL, barva VPL a normála povrchu, na kterém se toto VPL nachází.

Následně, pokud pixel splní stencil test, tedy v hierarchickém bufferu je bílá barva, potom pro příslušnou plošku v RSM mapě spočítáme přínos osvětlení od každého VPL světla. Tento výpočet se provede podle rovnice (3.3) [18]. Jednotlivé proměnné v rovnici jsou znázorněny na obrázku (obrázek 3.2).



L	Scene light source
E	Viewpoint
N_i	Surface normal at patch i
$\vec{V}_i$	Vector from i to the eye E
N_j	Surface normal at patch j
$\vec{V}_j$	Vector from j to the light L
L_j	VPL corresponding to patch j
$\vec{V}_{ij}$	Vector from patch i to patch j
$\vec{V}_{ji}$	Vector from patch j to patch i

Obrázek 3.2: Znázorněné proměnné, které se používají v hierarchické radiozitě [10].

$$F_{j \to i} = \frac{A_j(\vec{N}_i.\vec{V}_{ij})(\vec{N}_j.\vec{V}_{ji})}{\pi\|\vec{V}_{ij}\|^2 + A_j} \tag{3.3}$$

Kde:

- L světlo ve scéně
- E pozice pozorovatele
- $\vec{N}_i$ normála povrchu pro plošku i

- $\vec{N_j}$ normála povrchu pro plošku j
- $\vec{V_i}$ vektor od plošky i k pozorovateli
- $\vec{V_j}$ vektor od plošky j ke zdroji světla
- L_j příslušné VPL k plošce j
- $\vec{V_{ij}}$ vektor z plošky i k j
- $\vec{V_{ji}}$ vektor z plošky j k i

Dále, abychom dostali celkové nepřímé osvětlení pro plošku L_j vypočteme rovnici (3.4):

$$C_i^{indirect} = \rho_i \sum_j I_j \rho_j F_{j \to i} \tag{3.4}$$

Kde:

- ρ_i a ρ_j jsou difůzní barvy plošky i resp. j

Nakonec máme spočítané osvětlení, které je uložené v hierarchickém bufferu. Tento buffer následně interpolujume a zkombinujeme, abychom dostali výsledný obraz, ke kterému přičteme přímé osvětlení.

3.5 Interaktivní osvětlení scény pomocí dynamických plošných světel

Tato kapitola a její podkapitoly se budou zabývat novou hierarchickou technikou, která dokáže počítat přímé osvětlení ve scéně z plošných světel v reálném čase [9]. Plošná světla mohou dokonce dynamicky měnit svoji barvu, jako například obrazovka monitoru nebo televize a dalších přístrojů, které umí produkovat vizuální vjem tedy obraz.

Tato technika využívá hierarchických datových struktur pro uložení informací během výpočtu přímého osvětlení. Dále se provádí pouze v obrazovém prostoru, proto nám stačí zpracovávat jednoduché textury, tak abychom dostali výsledný obraz scény. Nedávné techniky, které demonstrují výpočet osvětlení scény pomocí plošných zdrojů světla, mohou být akcelerovány pomocí výpočtů, které se provádějí v obrazovém prostoru a spoléhají na hrubé určení viditelnosti jednotlivých bodů scény.

Tato nová technika, kterou si popíšeme, přináší tři hlavní výhody oproti již vytvořeným technikám. Jsou to:

- Můžeme počítat osvětlení scény z plošných dynamických zdrojů světla. Pokud nepoužíváme výpočet viditelnosti, potom takovýto přístup dokáže běžet v reálném čase pro difůzní i zrcadlové povrchy
- Pro výpočet viditelnosti jednotlivých ploch ve scéně můžeme použít jednoduchý a velice rychlý algoritmus zvaný voxelizace. Voxelizace jednoduše vytvoří 3D mřížku scény a určí, v kterých voxelech se nachází nějaká geometrie scény
- Nakonec používáme inkrementální zlepšení při výpočtu osvětlení a stínů. Tento přístup je stejný jako u výpočtu nepřímého osvětlení u hierarchické radiozity, kterou jsme si popsali v předchozí kapitole

Techniky generování stínů

V následující podkapitole si probereme jednotlivé techniky vytváření stínů ve scéně. Nejprve se zaměříme na jednoduché stínové mapy, poté zjistíme, jak je můžeme rozšířit tak, aby brali v potaz světelné zdroje s měnícím se osvětlením. Nakonec si popíšeme hierarchické techniky generování stínů.

Standardní stínové mapy dokáží velice rychle vypočítat stíny ve scéně. Tyto stíny jsou pouze tvrdé, což znamená, že s nimi nelze zobrazit plynulý přechod ze stínu do osvětlené plochy a polostín. Tuto techniku lze celkem jednoduše rozšířit o další algoritmy, které dokáží vygenerovat měkké stíny. Bohužel tato technika ignoruje proměnnou záři světelného zdroje. *Back-projection Shadow Maps*[4] nahrazují každý pixel stínové mapy za mikropolygon. Zpětná projekce vytváří seznam potenciálních ploch, které zastíní daný pixel. Abychom se vyhnuli prohledávání přes všechny mikropolygony, používáme hierarchické stínové mapy. Tato technika umí přesně počítat stíny plošných zdrojů světla, ale opět neumí identifikovat změny světelného záření.

Další možnou technikou je použití více stínových map, kde každá mapa reprezentuje jedno bodové světlo, které aproximuje plošný světelný zdroj. Základní nevýhodou tohoto přístupu je vytvoření mnoha stínových map, které zabírají hodně paměti na grafické kartě, a vysoké nároky na vzorkování mapy, což se prokazuje jako úzké hrdlo výpočtu. Možností, jak se tohoto hrdla částečně zbavit je použití hrubých *Imperfect Shadow Maps*[5]. Nedokonalá stínová mapa má nízké rozlišení, takže její vytváření nezabere tolik času. Abychom mohli vytvořit nedokonalé stínové mapy, je potřeba předzpracovat scénu tím, že se hustě povzorkuje body [14]. Tento přístup ovšem omezuje dynamiku objektů ve scéně.

Stínové mapy jsou zajímavým způsobem, jak akcelerovat mnoho komplexních osvětlovacích problémů, ale rozšířit z-buffer tak, aby obsahoval 3D data, se projevuje, jako zajímavá alternativa. Jednou z těchto alternativ je použití více úrovňových stínových map.

Pro techniku, kterou si v této kapitole popíšeme, používáme další z těchto více úrovňových technik pro výpočet stínů a tou je voxelizace scény. Voxelizace [2] (obrázek 3.3) diskretizuje prostor scény, která je zapouzdřena v pohledovém frustrumu. Každý bit pixelu framebufferu, do kterého se voxelizace ukládá, představuje jeden voxel ve scéně. Voxelizace scény je velice rychlá a účinná. Tato reprezentace scény se používá v mnoha aplikacích, jako jsou refrakce, průsvitnost a detekce kolizí.

Interaktivní přímé osvětlení z plošných zdrojů

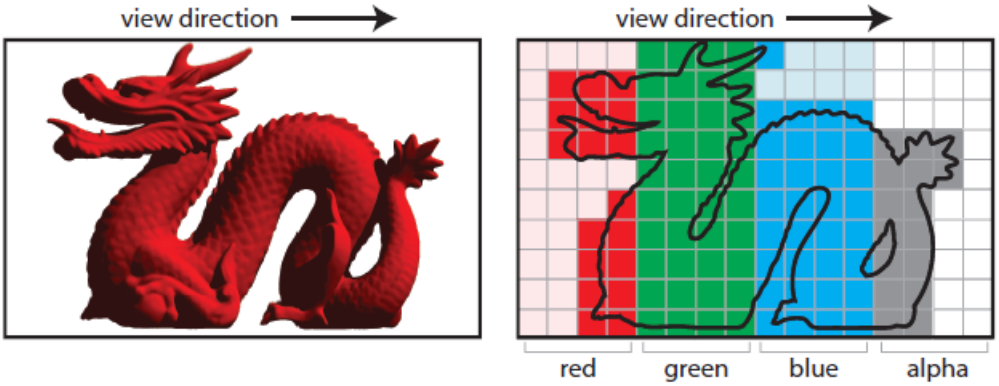
Výpočet přímého osvětlení z plošných zdrojů světla potřebuje integrovat světlo pro každý jeho pixel. Použitím zobrazovací rovnice a její transformací pro výpočet osvětlení přímého osvětlení plošného zdroje světla dostaneme rovnici (3.5)

$$L(x,\vec{\omega}) = \int_{y \in S} f_r(\vec{\omega}, x \to y) I(y) V(x \to y) G(x \to y) dA \tag{3.5}$$

Kde x je bod k zastínění a $\vec{\omega}$ je směr, kterým se na tento bod dívá pozorovatel. S je plocha světla, I je intenzita svetla, $x \to y$ je vektor z x ke světelnému vzorku y. V je viditelnost bodu x ze vzorku y a $G(x \to y)$ je geometrický term (3.6):

$$G(x \to y) = \frac{\cos(x \to y, \vec{N}_x)\cos(y \to x, \vec{N}_y)}{\|x \to y\|^2} \tag{3.6}$$

[4] Zpětně projekční stínové mapy vychází z anglického Backprojection shadow maps
[5] Nedokonalá stínová mapa je přeložena z anglického Imperfect Shadow Map



Obrázek 3.3: Voxelizace modelu z bočního pohledu. Každý bit RGBA textury představuje jeden voxel [9].

Kde:

- $\vec{N_x}$ normála plochy, na které leží bod x
- $\vec{N_y}$ normála plochy, na které leží bod y
- $x \to y$ je vektor z x do y
- $y \to x$ je vektor z y do x

Převedením integrálu na sumu vzorků dostaneme výsledné osvětlení z plošného zdroje světla, které je pokryto vzorky (3.7):

$$L(x, \vec{\omega}) = \sum_{i=0}^{N} f_r(\vec{\omega}, x \to y_i) I(y_i) V(x \to y_i) G(x \to y_i) A(y_i) \tag{3.7}$$

Kde:

- y_i vzorek plošného zdroje světla
- f_r BRDF funkce, koeficient stínění bodu x ze vzorku y_i pod pozorovacím úhlem ω

Sbírání přímého osvětlení bez viditelnosti

V této podkapitole si popíšeme algoritmus výpočtu přímého osvětlení bez uvažování viditelnosti. Proto v rovnici (3.7) položíme $V = 1$. Za předpokladu, že uvažujeme difůzní materiál s difůzní barvou ρ, dostáváme vztah (3.8):

$$L(x, \vec{\omega}) = \frac{\rho}{\pi} \sum_{i=0}^{N} f_r(\vec{\omega}, x \to y_i) I(y_i) G(x \to y_i) A(y_i) \tag{3.8}$$

Tento hierarchický přístup počítá přímé osvětlení z plošného zdroje světla pomocí následujících bodů:

1. Vytvoří G-buffer scény z pohledu pozorovatele

2. Vypočítá hloubkové a normálové nespojitosti nad G-bufferem a uloží je do textur z kterých vygeneruje mipmapy

3. Vytvoří hierarchický buffer, do kterého zapíše všechny úrovně mipmapy a zároveň označí pixely, v kterých se nachází hloubkové a normálové nespojitosti

4. Výpočet přímého osvětlení v pixelech, které byly označeny v předchozím bodě

5. Zkombinování a interpolace všech úrovní mipmapy v hierarchickém bufferu a vytvoření finálního obrazu scény

Bod jedna vytvoření G-bufferu je lehce proveditelný. Jednoduše vytvoříme textury, do kterých zapíšeme potřebné informace o vzdálenosti pixelu od kamery, normálách povrchů a difůzní barvě povrchu.

U bodu číslo dva si můžeme všimnout, že nespojitosti v osvětlení vznikají na podobných místech jako při výpočtu nepřímého osvětlení v předchozí technice hierarchické radiozity. Lze použít stejného vztahu (3.1) při výpočtu hloubkových derivací jako u předchozího postupu. Následně z těchto hloubkových derivací vytvoříme mipmapu. Podobný přístup použijeme při výpočtu normálových derivací (3.2), kdy procházíme jednotlivé komponenty normály a vybíráme z nich maximální a minimální hodnoty. Opět ze základní úrovně vygenerujeme zbylé úrovně mipmapy.

Dalším bodem je bod tři. V tomto bodě vezmeme jednotlivé úrovně vytvořené mipmapy v předchozím kroku a zapíšeme je do hierarchického bufferu, což je obyčejná textura o velkých rozměrech takových, aby se tam vešli všechny úrovně mipmapy. Nad tímto hierarchickým bufferem provedeme stencil test, který nám obarví pixely černou barvou, v kterých se bude počítat osvětlení z VPL světel. Výpočet stencil testu probíhá podle následujícího pseudo algoritmu:

```
for all (fragments f in quad) do
    if (for each i, f not in MipmapLevel(i)) then
        continue; // vzorek není v hierarchickém bufferu

    i = CurrentMipmapLevel(i);
    if (hasDepthOrNormalDiscontinuity(f, i)) then
        continue; // vzorek potřebuje rozdělení

    if (noDiscontinuity(f, i+1)) then
        continue; // vzorek je validní a nepotřebuje rozdělení

    setStencil(f);
```

U čtvrtého bodu algoritmu budeme z hierarchického bufferu s provedeným stencil testem, počítat a shromažďovat přímé osvětlení pouze v pixelech, které byli nastaveny při stencil testu. Osvětlení se bude shromažďovat jako přírůstky každého VPL světla, pomocí kterých aproximujeme plošný světelný zdroj. Nepotřebujeme shromažďovat osvětlení na samotných plošných světelných zdrojích, protože je ve stencil testu neoznačíme. Díky tomu se zvedne výkon algoritmu. Dále potom z předchozí techniky víme, že nepřímé osvětlení často nemá předem daný směr, takže nevíme, odkud dopadne na aktuální zpracovávaný pixel. Na druhou stranu přímé osvětlení má vždy předem daný směr od zdroje k bodu, pro který aktuálně počítáme osvětlení. Můžeme tedy použít jednoduchý a rychlý výpočet $N.L < 0$,

který identifikuje pixely, které od daného VPL světla nebudou přijímat žádné osvětlení. Zjednodušeně zahodíme pixely, které jsou odvráceny od všech rohů plošného světelného zdroje. Dále také zahodíme pixely, které jsou za plochou světla. Všechny tyto předpoklady platí pro difúzní povrchy. Pokud budeme chtít počítat osvětlení pro zrcadlové povrchy, budeme muset provést další průchod hierarchickým bufferem. V tomto průchodě poté musíme zlepšit a převzorkovat pixely, které budou obsahovat *specular highlight*[6] od všech rohů plošného světelného zdroje.

V posledním bodě našeho algoritmu postupně zkombinujeme a interpolujeme všechny úrovně naší mipmapy, které jsou uloženy v hierarchickém bufferu a vytvoříme výsledný obraz scény.

Výpočet stínů použitím voxelizace

V této podkapitole si popíšeme, jak lze aproximovat a přidat k výpočtu přímého osvětlení i stíny. V předchozí kapitole jsme si popsali přístup, kterým lze vypočítat přímé osvětlení pro dynamické plošné světlo. Nyní k němu uvedeme algoritmus, pomocí kterého přidáme stíny.

Nejjednoduším přístupem je použít voxelizaci. Voxelizace vytvoří uniformí mřížku scény. Následně pro každý pixel výsledného obrazu vrhneme paprsek z jednotlivých VPL světel do této mřížky a spočítáme, zda bude daný pixel zastíněný. To zjistíme tím že paprsek projde voxelem, který obsahuje nějakou geometrii. Tím nám vznikne textura, která bude obsahovat informace o tom, které VPL je zastíněno a které nikoliv. Pro průchod mřížkou můžeme použít 3DDA algoritmus.

Dalším způsobem je použít hierarchický průchod voxelovou mřížkou. Nejprve vytvoříme voxelizaci pro několik nižších rozlišení, které následně uložíme do mipmapy. Následně přepočítáváme stíny pouze tam, kde se hodně mění počet zastínění jednotlivých VPL světel mezi dvěma úrovněmi mipmapy.

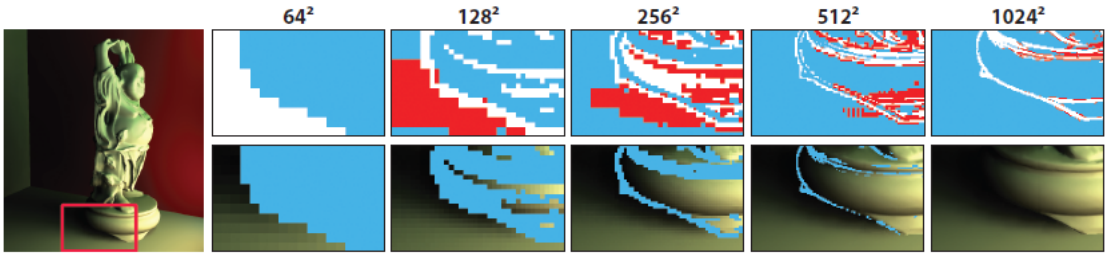
Nyní je potřeba upravit třetí a čtvrtý bod v předchozím přístupu tím, že zahrneme výpočet viditelnosti. Nejprve je potřeba upravit stencil test, který bude detekovat i nespojitosti ve viditelnosti jednotlivých pixelů. Následující pseudokód popisuje tento přístup:

```
i = CurrentMipmapLevel(f);
for all (fragments f in quad(mip-level(i)) do
    if (hasIllumDiscontinuity(f, i+1)) then
        setStencil(f); // hrubší vzorek má nespojitost v osvětlení
        continue;

    if (hasDepthOrNormalDiscontinuity(f, i)) then
        continue; // vzorek není validní potřebuje rozdělení

    if (noDiscontinuity(f, i+1)) then
        continue; // hrubší vzorek nepotřebuje dělení

    setStencil(f);
```

Následně ve čtvrtém bodě, kde počítáme osvětlení, musíme pro pixely, které potřebují převzorkovat na vyšší rozlišení, provést opětovný výpočet viditelnosti pro všechny VPL

[6] Specular highlight je zrcadlový odlesk na osvětleném povrchu, tzv. prasátko.

světla (obrázek 3.3). Vytvoření výsledného obrázku je již stejné jak v předchozím přístupu v bode pět. Tedy zkombinování a interpolace všech pixelů, které již nepotřebují přepočítat na vyšší rozlišení.



Obrázek 3.4: Příklad interpolace stencil bufferu. Nahoře: stencil buffer pro každé rozlišení. Bílé pixely mají hloubkové nebo normálové nespojitosti. Modré pixely nejsou zobrazovány. Červené pixely mají nespojitosti v osvětlení. Dole: výsledný obrázek po interpolaci mezi jednotlivými rozlišeními [9].

Kapitola 4

Návrh aplikace

V této kapitole se zaměřím na popis návrhu aplikace, která bude demonstrovat výpočet dynamického osvětlení z plošných světelných zdrojů. Celá aplikace bude vyvíjena v jazyce C++ pomocí technologie OpenGL. Většina algoritmů, které budu implementovat bude využívat technologických možností grafické karty. Z tohoto důvodu použii jazyk GLSL pro popis a implementaci shaderů na grafické kartě. Moje aplikace je navržena tak, aby co nejlépe splňovala podmínky, které jsou potřeba při výpočtu přímého osvětlení z plošných světelných zdrojů. Podmínky, jenž musí aplikace splňovat, vycházejí z kapitoly 3.5, kde jsem se zabýval právě mnou vybranou technikou pro výpočet dynamického osvětlení z plošných světelných zdrojů.

Jednou z nejdůležitějších podmínek je mít dobře definovanou scénu, která se bude zobrazovat v mojí aplikaci. Dnes existuje několik formátů, jak takovou scénu uložit. Nebudu zde rozebírat a vyjmenovávat jednotlivé formáty pro popis scény, jako jsou XML, AFF [1] a další textové [2] či binární varianty. Po pečlivém nastudování jednotlivých formátů, jsem si uvědomil, že nepotřebuju, aby moje scéna uměla udržovat pozice objektů v jednotlivých snímcích. Proto jsem mohl významně zjednodušit animace jednotlivých objektů v mé scéně.

V následující podkapitole se zaměřím na popis a rozbor mého navrženého formátu pro uložení scény.

4.1 Vlastní formát pro popis scény

Mnou navržený formát souboru pro popis scény je velice jednoduchý. Scéna se popisuje pomocí textového souboru, v kterém se definují jednotlivé objekty scény. Také se zde definují materiály jednotlivých objektů a světelných zdrojů. Každá entita je definována základním objektem. Různé typy entit se poté liší jednotlivými parametry. Tyto parametry si uvedeme v následujících odstavcích při popisu jednotlivých objektů, které mohou být definovány ve scéně.

První entitou, kterou lze definovat, je plošný zdroj světla. Tato entita představuje plošné světlo, které bude osvětlovat scénu. Pro zjednodušení budu uvažovat, že plošné světlo je čtvercová plocha natočená nějakým směrem. Taková plocha lze popsat čtyřmi body a normálou povrchu, na které tyto body leží. Čtyři body světla představují rohy plošného zdroje. Každému světlu je možné přiřadit materiál. Tento materiál má charakteristické vlastnosti

[1] Animated File Formal, www.ce.chalmers.se/BART/aff.html

[2] POV-Ray Scene description language, www.povray.org/documentation/view/3.6.1/224/

pro světelné zdroje, jako je barva záření. Tato barva může být proměnná, což umožňuje simulovat obrazovku monitoru nebo televize. Příklad definice plošného světla je níže.

```
area_light light1
    pos1 0 0 0
    pos2 10 0 0
    pos3 10 10 0
    pos4 0 10 0
    at 3 0 3
    mat red
end
```

Další entitou ve scéně je kamera. Kamera nám ve scéně simuluje pozorovatele. Pro jednoduchost lze definovat pouze jednu kameru pro danou scénu. Atributy, které můžeme kameře nastavit jsou její pozice, bod kam se ve scéně dívá, a další potřebné parametry, které jsou stejné jako při definici perspektivní projekce v OpenGL. Rozměry zobrazovací plochy jsou určeny šířkou a výškou, což nám dává rozlišení výsledného obrazu. Příklad definice kamery je na následujícím pseudokódu.

```
camera cam1
    pos 0.0 0.0 15.0
    up 0.0 1.0 0.0
    at 0.0 0.0 0.0
    fovy 45.0
    near 1.0
    far 50.0
    screen_size 1024 1024
end
```

Po vytvoření světelných zdrojů a kamery ve scéně je potřeba definovat objekty, které budou v námi vytvářené scéně zobrazeny. Jednotlivé modely se do scény přidávají pomocí klíčového slova object tak, jak je ukázáno na následujícím pseudokódu.

```
object kostka.obj
    scale 1 1 1
    anim rotate 1 0 0 0.02 # anim <typ> <x> <y> <z> <krok>
    mat red
end
```

Každý model je definován ve zvlášním externím souboru. K popisu geometrie modelu se používá formát wavefront .obj. Následně mohou být každému modelu přidány geometrické transformace, jakými jsou posuv, rotace a meřítko. Tyto transformace zaručí správné umístění modelu ve scéně tak, jak požadujeme. Tyto transformace se opět definují stejně jako v OpenGL.

Nedílnou součástí definice modelů jsou i jejich animace. Každý model ve scéně může být animován jednou či více transformacemi. Povolené animace jsou posuv a rotace nebo jejich kombinace. Animaci modelu nastavíme pomocí atributu anim (viz předchozí pseudokód).

Poslední částí definice scény jsou její materiály. Každý model, který je vytvořen ve scéně má nějaký materiál. Materiál modelu nám určuje jeho vizuální podání. Každý materiál

definuje tři základní složky. Těmi jsou ambientní, difúzní a odrazivá barva materiálu. Pokud je definován atribut tex, potom se místo těchto tří složek bude používat textura pro určení vlastností materálu. Příklad materiálu je v následujícím pseudokódu.

```
material red
    amb 0.12 0.12 0.12
    dif 1.0 0.0 0.0
    spec 1.0 1.0 1.0
    shin 50.0
    tex -1
end
```

Wavefront .obj

V této krátké podkapitolce stručně popíši wavefront .obj formát, který reprezentuje 3D modely v mé scéně.

Jednotlivé body modelu jsou uvozeny klíčovým slovem v jako vertex. Každý vertex potom má x, y a z souřadnici daného bodu. Následně jsou definovány normály povrchu v jednotlivých vertexech. Tyto normály začínají klíčovým slovem vn (vertex normal) a opět obsahují x, y a z souřadnici vektoru normály. Nakonec je třeba definovat, které vertexy, a které normály tvoří jednotlivé polygony, z kterých se daný model skládá. Tyto polygony se nazývají face a jsou popsány klíčovým slovem f, za nímž následuje vždy trojice identifikátorů, které určují jednotlivé vertexy a normály, které danému polygonu přísluší.

Tento formát jsem vybral z důvodu, že je textový a jednoduše čitelný. Zároveň umožnuje ukládat, jak modely s malým počtem polygonů, tak i komplexní modely s několika statisíci polygony.

4.2 Analýza aplikace

V této podkapitole se budu zabývat analýzou a popisem aplikace a jeji funkční části. Nejprve uvedu jednotlivé použité technologie a následně popíši funkční prvky a ovládání mé aplikace, která byla navržena pro hierarchickou techniku výpočtu osvětlení z dynamických plošných zdrojových světel.

Při výběru technologie, pomocí které budu implementovat mou aplikaci, jsem musel zohlednit požadavek zadání, kdy výsledná aplikace musí být přenositelná. Z tohoto důvodu jsem zvolil technologii OpenGL, která umožňuje vytvářet 3D grafiku a její nejnovější verze dokáže využívat všech technologických vymožeností, které jsou dostupné na grafických akcelerátorech firem AMD a NVIDIA. Jelikož je OpenGL API komplentě implementováno pomocí jazyka C, byl jsem nucen vytvořit si objektovou nadstavbu pomocí jazyka C++. Má aplikace tedy plně podporuje objektový návrh a je jednoduše čitelná a lze v ní jednoduše doimplementovat nové části nebo již stávající části změnit.

Hierarchické techniky pro výpočet osvětlení lze jednoduše akcelerovat použitím grafických karet. Tím získáváme podstatný náskok ve výkonu oproti jiným technikám (raytracing, radiozita). Hierarchické techniky tedy dokáží zobrazovat scénu v reálném čase. K docílení této vlastnosti jsem musel vetšinu algoritmů, které tyto techniky implmentují, přesunout do grafické karty. V dnešní době mají grafické karty unifikovanou architekturu, tedy mají určitý počet miniaturních paralelních procesorů. Tyto procesory si rozdělují zpracovávání

algoritmů, které jsou implementovány pomocí vertex, geometry a fragment shaderů. K využití těchto shaderů jsem použil jazyk GLSL, který umí dobře spolupracovat s technologií OpenGL a poskytuje funkce, které budu potřebovat k implementaci hierarchických technik.

V tomto odstavci popíši ovládání aplikace pro výpočet osvětlení pomocí hierarchických technik. Ovládání aplikace je navrženo tak, aby bylo co nejvíce intuitivní a aby uživatel už při prvním pohledu věděl, jak se aplikace používá. Z tohoto důvodu je nejlepší mít co nejvíc ovládacích prvků na hlavním okně aplikace. Při spuštění aplikace se otevře okno OpenGL aplikace, které zobrazí námi zvolenou scénu, která bude zadána jako parametr programu v příkazové řádce. Následně se v aplikaci zobrazí nápověda s ovládáním. Tato nápověda se dá vypnout klávesou h. V aplikaci se dají dále nastavit parametry, s kterými se má daná scéna zobrazovat. Tyto parametry jsou počet VPL světel, jednotlivé prahy, které se používají při určování nespojitostí v geometrii scény a počet vzorků paprsku, který se použije při určování viditelnosti. Pohyb po scéně je zajištěn pomocí kláves nebo pomocí trackballu.

Kapitola 5

Implementace

V této kapitole a následujících podkapitolách se budu zabývat implementací hierarchické techniky pro výpočet osvětlení z dynamických plošných světelných zdrojů. Jednotlivé části hierarchické techniky, kterou jsem implementoval budou podrobněji rozebrány v následujících podkapitolách. Postup, jež vede k výpočtu osvětlení, byl již teoreticky nastíněn v kapitole 3.5. Nyní se zaměřím na jednotlivé kroky tohoto postupu a popíši, jak jsem je implementoval pomocí shaderů na grafické kartě.

Jak již bylo uvedeno v předchozí kapitole návrhu 4, při implementaci jsem použil OpenGL API, které jsem upravil do svých objektových tříd v jazyce C++. Nejdůležitější třídy, které používám nejčastěji jsou *ShaderProgram* a *FrameBuffer*.

ShaderProgram je třída, jenž zapouzdřuje práci se shadery. Poskytuje tedy několik metod, které dokáží pracovat se shadery z OpenGL aplikace. Hlavní metodou je konstruktor jenž bere dva parametry. Těmi jsou názvy zdrojových souboru pro vertex a fragment shader. Konstruktor následně vytvoří vertex a fragment shader objekty, které zkompiluje a sestaví z nich program, který se nahraje do paměti grafické karty. Pokud chceme tento shader program použít je potřeba zavolat metodu *bindProgram*, která řekne grafické kartě, aby začala používat námi vytvořený a nahraný shader program na grafické kartě. Případně metodou *unBindProgram* přestaneme používat námi zvolený program. Nastavení uniformích proměnných ve vertex nebo fragment shaderu lze zavoláním metody *setUniformVariable*. Uniformní proměnné předávají data z OpenGL programu do shaderů na grafické kartě. Jejich obsah lze změnit pouze před zpracováním primitiva vertex shaderem.

Druhou nejpoužívanější třídou je *FrameBuffer*. *FrameBuffer* je třída, která zapouzdřuje FrameBuffer Object, což je uživatelský buffer v OpenGL, do kterého lze zobrazovat výstup shader programu po jeho provedení. Tento framebuffer není napojený na systémový framebuffer, tudíž jeho výstup nelze vidět v okně aplikace. Každý framebuffer má svoji velikost. Typicky je to velikost okna OpenGL aplikace. Framebuffer object obsahuje sadu výstupů pro barvu, hloubku a stencil test. Aby bylo možné do těchto výstupů ukládat hodnoty, je třeba jim připojit nějakou paměť. Nabízí se dvě možnosti a těmi jsou 2D textura nebo renderbuffer. Pro potřeby svého programu jsem implementoval pouze připojení 2D textury.

Objekt framebufferu se opět vytvoří pomocí konstruktoru, kterému předáme jako parametry rozměry tohoto bufferu. Následně můžeme ještě určit, zda chceme, aby náš framebuffer obsahoval depth buffer a stencil buffer. Hlavně stencil buffer je z našeho pohledu hodně důležitý. Jeho důležitost si připomeneme v následujících podkapitolách. Pro připojení 2D textury k jednomu z barevných výstupu použijeme metodu *AttachRenderTarget*. Této metodě můžeme říct, jaký datový formát má daná textura mít a zda se z ní má generovat mipmapa.

V úvodu této kapitoly jsem popsal základní a nejvyužívanější třídy v mé aplikaci. Nyní se podíváme detalněji na implementaci jednotlivých bodů algoritmu, který počítá osvětlení z dynamických plošných světelných zdrojů.

5.1 Přímé osvětlení pomocí hierarchické techniky bez viditelnosti

V kapitole 3.5 jsme si uvedli zobrazovací rovnici (3.7), s níž umíme vypočítat přímé osvětlení jakéhokoli bodu ve scéně. V kapitole Implementace se budu zabývat, jak tuto rovnici aproximovat použitím algoritmů, které se dají implementovat pomocí vertex a fragment shaderů. Pro zjednodušení nejprve uvedu, jak vypočítat přímé osvětlení z plošných světelných zdrojů pomocí hierarchické techniky bez řešení viditelnosti. V rovnici (3.7) je potřeba položit V = 1, neboť právě funkce V(x) nám říká, zda je bod, pro který počítáme osvětlení, zastíněn. Jinými slovy tento bod není viditelný ze světelného zdroje.

Takto upravenou rovnicí lze vypočítat přímé osvětlení jakéhokoli bodu ve scéně, bez uvažování viditelnosti. K výpočtu použijeme hierarchickou techniku, která byla popsána v kapitole 3.5 a implementuje pět hlavních kroků, které byli zmíněny a uvedeny v téže kapitole. V následujících podkapitolách rozeberu implementační detaily jednotlivých kroků výpočtu přímého osvětlení použitím hierarchické techniky.

Generování Reflective Shadow Map

Z teoretického pohledu byla popsána reflective shadow map více v kapitole 3.3. V této části práce se zaměřím na její implementační detaily. Víme, že tato mapa sestává ze tří datových struktur, do kterých se ukládají jednotlivé informace o světelném zdroji. Tyto informace jsou pozice světla ve scéně, normála povrchu a barevné záření světla. Jelikož všechny tyto informace můžeme lehce získat pomocí shader programu na grafické kartě, lze na uložení těchto informací použít moje *FrameBuffer* třída se třemi připojenými výstupy ve formě 2D textur. Každá textura bude potom obsahovat jeden typ informací o světelném zdroji.

Funkce, jenž vytváří tento buffer v rámci mé aplikace se nazývá *renderLightSpaceBuffer*. Tato funkce řekne grafické kartě, aby začala používat správný shader program, který vypočte námi požadované informace. Jednotlivé důležité části fragment shaderu jsou uvedeny v následujícím pseudokódu 5.1.

Algoritmus 5.1: Pseudokód popisující funkci fragment shaderu.

```
   /* Do první textury ukládáme pozice jednotlivých fragmentů
      v pohledovém prostoru.                                    */
1  lightEyePos ← modelViewMatrix · vertexPosition;
   /* Druhá textura obsahuje normály v jednotlivých fragmentech. */
2  lightDirection ← normalize(normalMatrix · normal);
   /* Poslední textura obsahuje barvu jednotlivých fragmetů.     */
3  lightFlux ← lightColor;
```

Vytvoření VPL bufferu

V podkapitole 5.1 jsem uvedl, jak lze vypočítat reflective shadow map. Nyní potřebujeme tuto mapu navzorkovat a uložit si jednotlivé vzorky VPL světel do bufferu, který představuje VPL cache, jenž uchovává informace o jednotlivých VPL světlech.

Princip vzorkování RSM je velmi jednoduchý. Fragment shader dostane vždy aktuální ID VPL světla, které potřebujeme získat. Podle tohoto ID vybereme v RSM vzorek a informace z tohoto vzorku uložíme do VPL bufferu. Podstatná část fragment shaderu je uvedena na následujícím pseudokódu 5.2. Metoda, která se stará o vytváření VPL bufferu v rámci aplikace se nazývá *computeVPLBuffer*.

Algoritmus 5.2: Pseudokód nastiňující algoritmus fragment shaderu.

1 $vplSqrt \leftarrow \sqrt{VPLCount}$;
2 $vplDelta \leftarrow \frac{0.5}{vplSqrt}$;
3 $vplOffset \leftarrow \frac{1.0}{vplSqrt}$;
4 získáme ID aktuálního VPL světla a uložíme ho do $vplID$;
`/* Nyní vygenerujeme 2D texturovací souřadnici, která určí správný vzorek VPL z RSM mapy. */`
5 $x \leftarrow vplSqrt \cdot fract(\frac{vplID}{vplSqrt})$;
6 $y \leftarrow floor(\frac{vplID}{vplSqrt})$;
7 $texCoord \leftarrow vplOffset \cdot vec2(x, y) + vec2(vplDelta, vplDelta)$;
8 zapíšeme požadované informace z RSM mapy do VPL bufferu;

Vytvoření G-bufferu

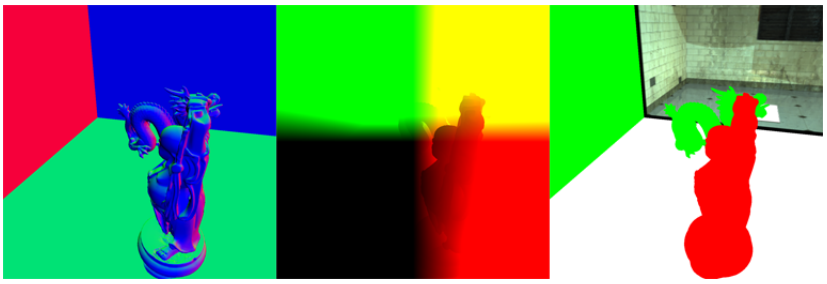
Získání G-bufferu 5.1 je ve většině případů stejné jako u RSM. K tomu, abychom byli schopni vypočítat osvětlení scény, musíme znát její geometrii. Právě z tohoto důvodu je potřeba vytvořit G-buffer, který ukládá informace o geometrii scény. Tyto informace jsou normály jednotlivých povrchů, pozice povrchů v pohledovém prostoru, vzdálenost jednotlivých objektů od kamery a v neposlední řadě barva, kterou jednotlivé plochy ve scéně vyzařují (difůzní složka). Vzdálenost jednotlivých objektů od kamery lze získat jednoduše. Můžeme přímo použít hodnoty uložené v z-bufferu. Hodnoty v z-bufferu jsou ale ukládány podle logaritmické osy. Tím ztrácíme detaily o hloubce objektů, které jsou více vzdálené od kamery. Z tohoto důvodu jsem jednotlivé hodnoty ze z-bufferu přepočítal do lineárního rozsahu, který poskytuje lepší informaci o hloubce a dají se tak snadněji detekovat hloubkové nespojitosti. Všechny zbylé informace o geometrii se dají lehce získat pomocí vertex a fragment shaderů, jejichž nejdůležitější části jsou opět uvedeny v pseudokódu 5.3. K generování G-bufferu slouží funkce *renderGbuffer*.

Mipmapa hloubkových derivací

K výpočtu hloubkových derivací se používají informace o hloubce objektů, které jsme si uložili při generování G-Bufferu. Nejjednoduší přístup, jak vytvořit nižší úrovně mipmapy, je

Algoritmus 5.3: Pseudokód popisující generování G-Bufferu.

```
   /* pozice fragmentu v pohledovém prostoru                    */
1  eyePosition ← modelViewMatrix · vertex;
   /* vzdálenost fragmentu od kamery                            */
2  linearDepth ←length(eyePosition);
   /* normála plochy v aktuálním fragmentu                      */
3  fragNormal ←normalize(normalMatrix · normal);
   /* barva fragmentu                                           */
4  fragColor ← materialDiffuseColor;
5  informace o geometrii scény uložíme do G-Bufferu;
```



Obrázek 5.1: Vlevo normály povrchu. Uprostřed pozice fragmentů v pohledovém prostoru. Vpravo difúzní barva materiálu jednotlivých objektů. Jako alpha složka je uvedena vzdálenost fragmentů od kamery.

použití čtyř sousedních pixelů v aktuální úrovni mipmapy a vybrání jejich maximální a minimální hodnoty. Tyto dvě hodnoty se zapíšou příslušnému vzorku v nižší úrovni. Tímto způsobem lze jednoduše vygenerovat zbývající úrovně mipmapy. Při určování hloubkových nespojitostí se potom porovnává rozdíl mezi maximalní a minimální hodnotou s námi definovaným prahem. Pokud je rozdíl větší než práh, potom v tomto bodě existuje hloubková nespojitost. Tento přístup je velice jednoduchý a na první pohled i účinný. Problémy nastanou, pokud se na plochy ve scéně díváme pod velkým pozorovacím úhlem. Na těchto plochách jsou tímto přístupem špatně určeny nespojitosti.

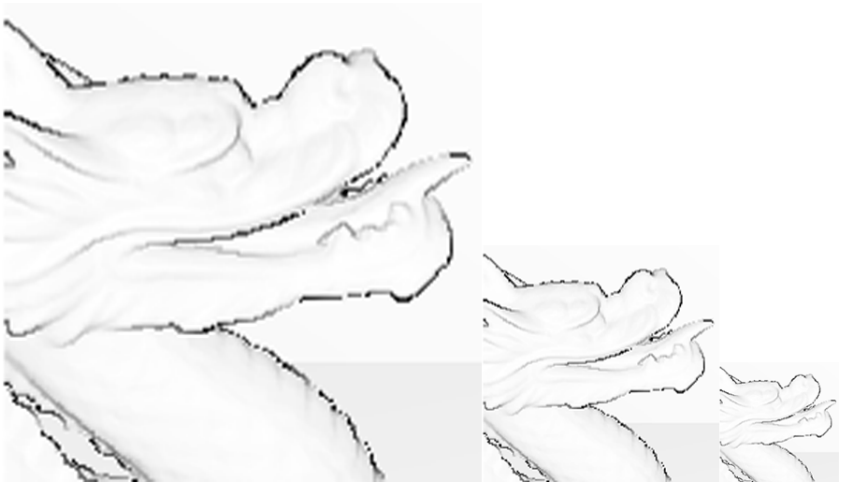
Lepší metodou je vypočítat hloubkovou derivaci pro daný vzorek. Hloubková derivace se vypočítá podle rovnice (3.1), která byla uvedena a blíže popsána v kapitole 3.4. Derivace jsou vypočteny na nejvyšší úrovni mipmapy. Další úrovně mipmap se vytvoří vybráním maximalní hloubkové derivace z devíti okolí aktuálního pixelu. Tato maximální hodnota je poté zapsána do nižší úrovně mipmapy. Při určování nespojitostí se potom vezme tato maximální hodnota derivace a pokud je větší než námi stanovený práh, pak v tomto bodě existuje hloubková nespojitost. Mipmapa s prvními třemi úrovněmi je zobrazena na obrázku 5.2. Na následujícím pseudokódu 5.4 je zjednodušený algoritmus, který počítá derivace použitím fragment shaderu.

Mipmapa normálových derivací

Ke korektnímu určení osvětlení potřebujeme znát nejen změny v hloubce, které nejčastěji vznikají na obrysech objektů, ale musíme umět rozpoznat i zakřivení povrchu jednotlivých objektů. K tomu nám slouží normálové derivace. Normály jednotlivých povrchů máme

Algoritmus 5.4: Pseudokód fragment shaderu generující hloubkové derivace, které zapisuje do mipmapy.

```
   /* Devět vzorků, z kterých počítáme hloubkovou derivaci          */
1  float texels[9];
2  float maxDerivation ← texels[0];
   /* získáme maximální derivaci                                    */
3  for x ← 0 to 8 do
4  |  maxDerivation ←max(texels[x], maxDerivation);
5  end
6  maxDerivation hodnotu zapíšeme do mipmapy na příslušnou pozici;
```



Obrázek 5.2: První tři úrovně mipmapy, která představuje hloubkové derivace. Jednotlivá rozlišení jsou 1024^2, 512^2 resp. 256^2. Z mipmapy je pro ilustraci vybrán detail hlavy modelu draka.

uloženy v 2D textuře, která je součástí G-Bufferu.

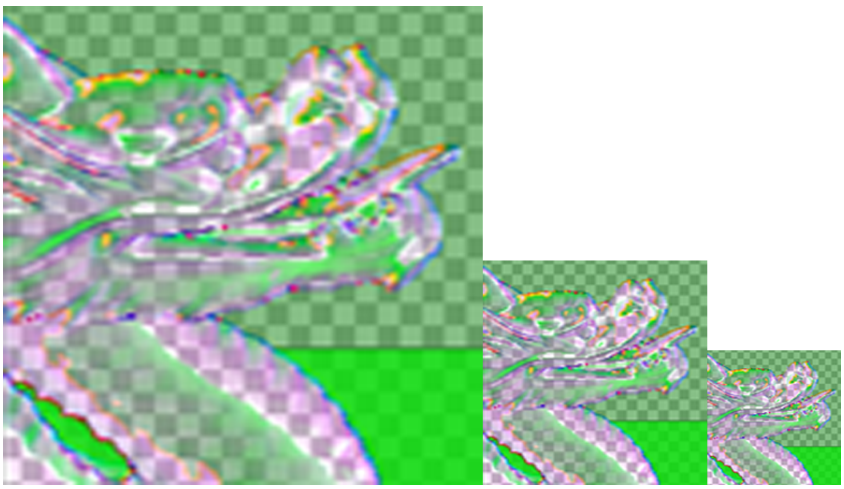
Nejjednodušším přístupem, jak vypočítat normálové derivace, je určení minimální a maximální hodnoty jednotlivých komponent (x, y, z) normál ve čtyřech sousedních pixelech. Následně je třeba tyto minimální a maximální hodnoty uložit do následující nižší úrovně mipmapy. Takto jednoduše získáme tři mipmapy, kdy jednotlivé mapy budou obsahovat min-max hodnoty v jednotlivých komponentách normál. Tento přístup, ale vede k velkým nárokům na paměť, kdy musíme ukládat v paměti grafické karty tři mipmapy.

Druhou metodou je neporovnávát, v kterých osách se sousední normály nejvíce liší, ale vypočítat zakřivení povrchu pomocí těchto sousedních normál, tedy normálové derivace. Zakřivení povrchu se vypočítá podle rovnice (3.2). Hodnoty zakřivení uložíme do nejvyšší úrovně mipmapy. Následující nižší úrovně generujeme nalezením maximální hodnoty zakřivení v devíti sousedních vzorcích aktuálně zpracovávaného texelu. Tuto maximální hodnotu následně zapíšeme do nižší úrovně mipmapy. Normálová nespojitost je následně detekována v bodech, kde maximální zakřivení povrchu je větší než daný práh.

Na následujícím pseudokódu 5.5 je nastíněn zjednodušený algoritmus výpočtu normálových derivací pomocí fragment shaderu. První tři úrovně mipmapy normálových derivací jsou znázorněny na obrázku 5.3.

Algoritmus 5.5: Pseudokód fragment shaderu generující normálové derivace, které zapisuje do mipmapy.

```
   /* Devět vzorků, z kterých počítáme normálovou derivaci          */
1  float texels[9];
2  float maxDerivation ← texels[0];
3  float minDerivation ← texels[0];
   /* získáme maximální derivaci                                    */
4  for x ← 0 to 8 do
5  |  maxDerivation ←max(texels[x], maxDerivation);
6  |  minDerivation ←min(texels[x], minDerivation);
7  end
8  maxDerivation a minDerivation hodnoty zapíšeme do mipmapy na příslušnou
   pozici;
```



Obrázek 5.3: První tři úrovně mipmapy, která představuje normálové derivace. Jednotlivá rozlišení jsou 1024^2, 512^2 resp. 256^2. Z mipmapy je pro ilustraci vybrán detail hlavy modelu draka.

Určení hloubkových a normálových nespojitostí

Nejjednoduším způsobem, jak určit texely v hierarchickém bufferu, v kterých se nacházejí nespojitosti, je použití a testovaní if else podmínky. Jelikož tento algoritmus je opět implementován ve fragment shaderu a použití if else podmínky v shaderech značně ovlivňuje jejich výkon, je lepší použít technologii stencil bufferu, jež je hardwarově implementovaná na grafické kartě.

Stencil buffer neukládá barvu jednotlivých pixelů, ale celočíselné hodnoty, které jsou nastaveny každému pixelu na základě nějaké operace. Těmito operacemi lze určit, které pixely hierarchického bufferu budou obsahovat nespojitost, a které nikoliv.

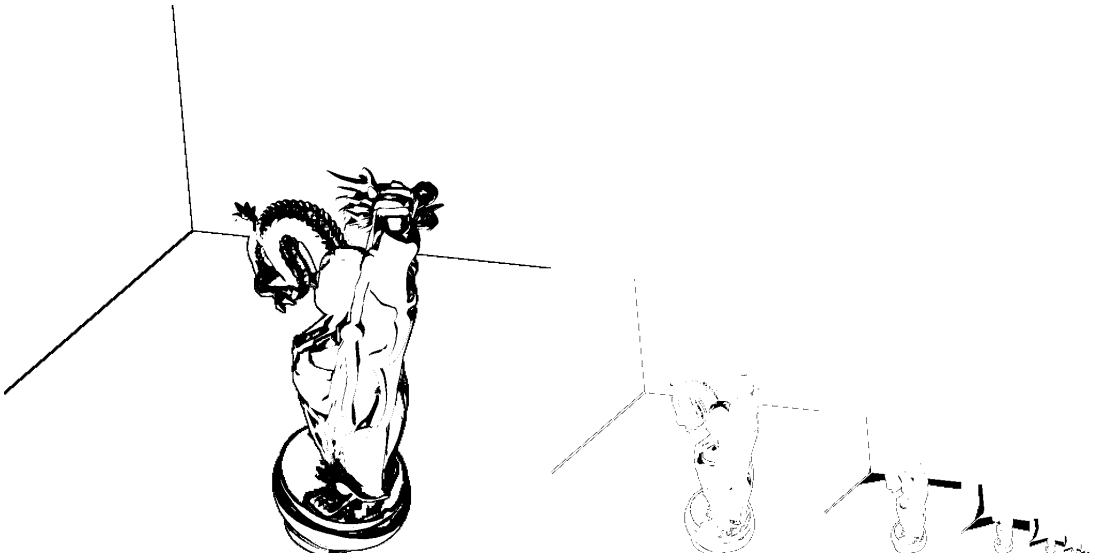
Nyní stačí vzít jednotlivé mipmapy, které jsme získali v kapitolách 5 resp. 6 a pomocí fragment shaderu zahodit pixely, které neobsahují nespojitosti a zároveň nastavit v stencil bufferu celočíselnou hodnotu pro pixely, které nespojitosti obsahují.

Následující obrázek 5.4 zobrazuje obsah stencil bufferu po určení hloubkových a normálových nespojitostí. Tento stencil buffer je generován na základě hierarchického bufferu,

do kterého jsou naskládány jednotlivé úrovně mipmapy od nejvyšší po nejnižší. Postup, kterým určíme nespojitosti, je nastíněn v pseudokódu 5.6.

Algoritmus 5.6: Algoritmus určující hloubkové a normálové nespojitosti.

```
/* Zjištění, zda je potřeba počítat osvětlení na vyšší úrovni nebo nám
   dostačuje vzorek z nižší úrovně.                                    */
needRefinement ← (depthDerivFiner > depthThreshold || normalDerivFiner >
normalThreshold);
coarseLevelSufficent ← (depthDerivCoarse > depthThreshold ||
normalDerivCoarse > normalThreshold);
if needRefinement and !atFinerLevel then
    discard;
end
if coarseLevelSufficent and !atCoarseLevel then
    discard;
end
nastav stencil bit ve stencil bufferu;
```



Obrázek 5.4: Obsah stencil bufferu mého hierarchického bufferu, který obsahuje černé pixely ve vzorcích, které obsahují nespojitosti a bílé pixely, které nespojitosti nemají.

Výpočet přímého osvětlení

V předchozí kapitole 8 jsme si vytvořili hierarchický buffer, jež obsahuje označené vzorky, v nichž se bude počítat přímé osvětlení. Jednotlivé vzorky mohou mít různou velikost ve výsledném obrazu. Tato velikost závisí na úrovni mipmapy, z které byly vybrány.

Výpočet přímého osvětlení se provede ve vzorcích, které jsou označeny v hierarchickém bufferu. Výpočet se provádí pomocí rovnice (3.8) bez uvažování viditelnosti. Viditelnost je vypočtena a modulována s výsledným obrazem v dalších krocích algoritmu, který používá mnou implementovaná hierarchická technika. Pseudokód výpočtu přímého osvětlení je

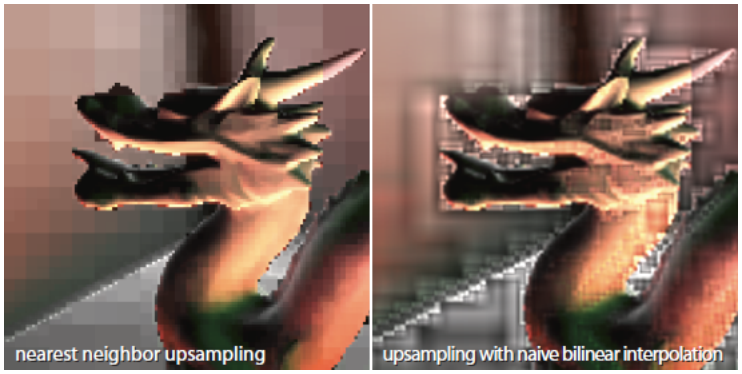
uveden v 5.7.

Algoritmus 5.7: Algoritmus pro výpočet osvětlení v daném vzorku.

```
   /* Informace o fragmentu z G-Bufferu                                    */
1  fragPosition ← fragPositionsTexture;
2  fragNormal ← fragNormalsTexture;
   /* Vypočítáme přínos osvětlení v daném fragmentu z každého VPL světla
      */
3  for x ← 0 to vplCount do
4  |  dirToVPL ← vplPosition − fragPosition;
5  |  distanceToVPL ←dot(dirToVPL, dirToVPL);
6  |  dotNormalFrag ←dot(fragNormal, dirToVPL);
7  |  dotNormalVPL ←dot(fragNormal, −dirToVPL);
8  |  illumination ← (dotNormalFrag · dotNormalVPL) / distanceToVPL · vplColor;
9  end
```

Zvyšování rozlišení a kombinace vzorků

Každá úroveň hierarchického bufferu obsahuje části výsledného přímého osvětlení. Každá část osvětlení má jiné rozlišení, které závisí na úrovni hierarchického bufferu, v kterém bylo vypočítáno. V této kapitole si uvedeme, jak získat celkové přímé osvětlení scény pomocí kombinace jednotlivých úrovní hierarchického bufferu.



Obrázek 5.5: Vlevo metoda, která kombinuje úrovně hierarchického bufferu pomocí nejbližších sousedů. Vpravo jednoduchá bilineární interpolace, která vytváří kruhové artefakty.

Obrázek 5.5 znázorňuje dvě metody, jak interpolovat a zkombinovat úrovně hierarchického bufferu do výsledného obrazu. První metodou je výběr nejbližšího vzorku v nižší úrovni, který chceme převzorkovat do vyšší úrovně. Tímto způsobem vzniknou čtverečkované artefakty, kdy ve výsledném obrazu vidíme úrovně hierarchického bufferu, z kterých byly jednotlivé pixely získány.

Druhou metodou je použití bilineární interpolace. Tato metoda ovšem používá i vzorky, které neobsahují korektní osvětlení. Ve výsledném obraze vznikají kruhové artefakty, které jsou opět nežádoucí.

Hierarchický buffer je velice komplexní struktura a z ní plyne, proč nemůžeme použít tyto dvě jednoduché metody ke kombinaci jednotlivých úrovní. Každá úroveň tohoto bufferu

má nějaké rozlišení a obsahuje buď texely s validním osvětlením nebo texely, které neobsahují vůbec žádné osvětlení. V těchto texelech se osvětlení doplní z nižších nebo vyšších úrovní bufferu. Z tohoto důvodu jsem implementoval interpolaci, která interpoluje výsledný obraz od nejnižší úrovně po nejvyšší. Tato metoda probíhá následujícím způsobem:

1. Interpolujeme nižší úroveň bufferu na rozlišení vyšší úrovně, ale pouze v texelech, které obsahují validní osvětlení.
2. Zkombinujeme převzorkovanou nižší úroveň s vyšší úrovní.
3. Výsledek převzorkujeme na rozlišení následující vyšší úrovně.
4. Tento postup opakujeme, dokud nedostaneme výsledný zkombinovaný obraz s rozlišením nejvyšší úrovně hierarchického bufferu.

Tyto jednotlivé kroky, jsou implementovány pomocí následujícího pseudokódu 5.8. Jejich znázornění je uvedeno na obrázku 5.6.

Po zpracování všech úrovní hierarchického bufferu dostaneme zkombinovaný a převzorkovaný obraz na nejvyšším rozlišení. Tento obraz obsahuje vypočtené osvětlení, které se plynule mění a je bez kruhových artefaktů. Navíc tato metoda nešíří energii osvětlení do hlavních nespojitostí. Osvětlení v těchto nespojitostech je počítáno na vyšších úrovních hierarchického bufferu, protože každý vzorek je interpolován pouze se vzorky, které mají stejnou velikost a tyto vzorky jsou nakonec převzorkovány na vyšší rozlišení.

Algoritmus 5.8: Algoritmus pro výpočet osvětlení v daném vzorku.

```
   /* Získáme texely z aktuální úrovně a z nižší úrovně. Následně
      definujeme váhy jednotlivých texelů.                          */
 1 finerTexels[9];
 2 coarserTexels[9];
 3 weights[9];
   /* Interpolujeme jednotlivé texely. Výsledné interpolované osvětlení
      zapíšeme do výstupního bufferu.                               */
 4 for i ← 0 to 8 do
 5 |   if hasIllum(finerTexels[i]) or hasIllum(coarserTexels[i]) then
 6 |   |   totalWeight += weights[i];
 7 |   |   outputTexel += weights[i] · (finerTexels[i] + coarserTexels[i]);
 8 |   end
 9 end
10 outputTexel ← outputTexel / totalWeight;
```

5.2 Aplikace hierarchické techniky na výpočet viditelnosti

V předchozích částech kapitoly Implementace jsem popsal implementační detaily výpočtu přímého osvětlení použitím hierarchické techniky. Nyní už víme jakým způsobem implementovat hierarchickou techniku použitím OpenGL technologie a jak plně využít vertex a fragment shadery grafické karty.

Obrázek 5.6: Jeden krok interpolační metody, která korektně kombinuje úrovně hierarchického bufferu.

V této kapitole a dalších podkapitolách si uvedeme, jak použít hierarchickou techniku pro výpočet viditelnosti. Viditelnost nám určuje, které body ve scéně jsou viditelné z pozice světla, a které nikoliv.

Výpočet viditelnosti lze řešit několika přístupy. Tyto přístupy byly popsány v kapitole 3.5 a obsahují několik omezení, kterým se chceme vyhnout. Tyto omezení vznikají při použití plošných světelných zdrojů s proměnným zářením. Při použití stínových map musíme generovat desítky těchto map v závislosti na počtu použitých VPL světel k aproximaci světla. To vyúsťuje k vysokým paměťovým nárokům a velkému počtu vzorkování jednotlivých stínových map ke korektnímu určení viditelnosti. Místo stínových map můžeme použít jejich odlehčenou verzi v podobě Imperfect shadow maps. Takovéto mapy mají malé rozlišení, tudíž obsahují méně detailů. Tímto přístupem se sice zbavíme paměťové náročnosti, ale velký počet vzorkování map zůstává.

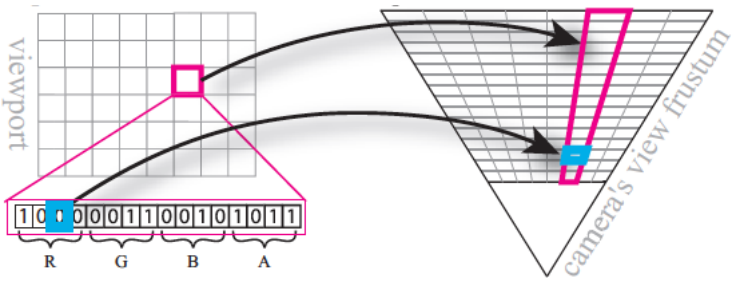
Z těchto a dalších důvodů jsem použil techniku zvanou voxelizace. Voxelizace poskytuje jednoduchou metodu, jak identifikovat místa ve scéně, které obsahují geometrii. Bohužel použitím samotné voxelizace a procházením paprsku voxelovou mřížkou pro každý pixel obrazu, se dostáváme do dalších problémů. Tímto problémem je rychlost výpočtu viditelnosti, která je sice vyšší než při použití stínových map, ale pro naše potřeby, kdy se snažíme docílit aplikace, která umí zobrazovat scénu v reálném čase, značně nevhodná. Tomuto problému jsem se vyhnul tak, že jsem upravil hierarchickou techniku a použil ji na výpočet viditelnosti. Tento postup je popsán v následujících podkapitolách.

Voxelizace scény

Voxelizace vytváří 3D mřížku buněk okolo scény. Pro každou buňku ve scéně je určeno, která primitiva scény v ní leží. Pro tento přístup je třeba umět vypočítat průsečík primitiva s buňkou. Při implementaci voxelizace jsem ovšem použil grafického hardwaru, který poskytuje rychlou možnost, jak scénu voxelizovat. Nemusel jsem se tedy zabývat výpočtem průsečíků, které značně zpomalují vytváření mřížky. Voxelizace implementovaná pomocí grafické karty umí voxelizovat i relativně složité scény během několika málo milisekund.

Pokud se lépe zamyslíme, tak při zobrazování scény pomocí OpenGL a vertex a fragment shaderů je viditelná část scény diskretizována na pixely. Proto můžeme jednoduše určit voxelovou mřížku podle dvou jednoduchých kritérií. První je, že scéna se zobrazuje z pohledu kamery a potom tedy tato část scény je obsažena v pohledovém prostoru. Roz-

lišení tohoto prostoru je dáno rozlišením kamery a framebufferu, do kterého tuto scénu zobrazujeme. Zadruhé grafická karta při zobrazovaní scény rasterizuje všechna primitiva, která jsou obsažena v pohledovém prostoru a tedy počítá jejich průsečíky s paprsky, které jsou jakoby vysílány z pohledu kamery. Následně každý fragment, který je vygenerován rasterizací, má x-, y-souřadnici. Dále u tohoto fragmetu můžeme získat jeho z-souřadnici, která je uložena v z-bufferu. Tímto zjistíme, v které buňce voxelové mřížky fragment leží. Jelikož používám grafickou kartu k vytvoření voxelové mřížky, je třeba celou mřížku popsat pomocí jednoduché 2D textury a do ní uložit informace o jednotlivých buňkách (Obrázek 5.7).



Obrázek 5.7: Schéma voxel bufferu. Jde o 2D texturu, kde každý pixel přestavuje sloupec voxelů v z-ose. Voxely jsou reprezentovány bity RGBA kanálu.

Nejprve je třeba nastavit správné transformační matice pro pohledový a projekční prostor. Korektně určit rozlišení výstupního framebufferu pomocí funkce *glViewport*. Toto rozlišení musí být stejné jako velikost 2D textury, která je připojena k výstupnímu framebufferu. Každý pixel této textury nám představuje jeden sloupec ve voxelové mřížce, který je rovnoběžný s z-osou. Při zpracovávání polygonální scény pomocí grafické karty dostaneme vždy rasterizovaný polygon, jehož fragmenty mají x-, y- a z-souřadnici. Použitím x- a y-souřadnice zjistíme korektní pixel v 2D textuře. Následně pomocí z-souřadnice musíme nastavit správný bit v RGBA kanálu toho pixelu. Jednotlivé bity totiž představují voxely ve voxelové mřížce. Hodnota upraveného RGBA kanálu se zapíše do výstupního framebufferu, kde se operací XOR zkombinuje s původní hodnotou pixelu.

Po provedení rasterizace všech polygonů dostaneme texturu, kde jsou v jednotlivých pixelech nastaveny příslušné bity, které reprezentují voxely, v kterých se nachází jednotlivé polygony scény. Jednoduchá 2D textura obsahuje 8 bitů na kanál, z čehož dostaneme celkem 32 bitů a tedy 32 voxelů. Tato hodnota je pro naše potřeby nevhodná, protože potřebujeme uložit více detailů ve scéně. Ve své implementaci používám texturu, která má integerový formát pixelu a obsahuje 32 bitů na kanál. Celkem jsem dostal 128 bitů a potom 128 voxelů, jak rozdělit z-osu. Tento počet bitů se ukázal jako dostačující a přidáním dalších výstupních textur k framebufferu, lze tuto hodnotu rozšířit až na 2048 bitů.

Průchod voxelovou mřížkou

V předchozí podkapitole jsme si popsali, jak vytvořit texturu, která obsahuje voxelizaci scény. Nyní uvedu, jak co nejvýhodněji implementovat procházení paprsku voxelizační mřížkou. Průchod paprsku 3D mřížkou je velice jednoduchá záležitost, ale je třeba se zamyslet, jak ji implementovat s využitím vertex a fragment shaderů na grafické kartě.

První možností je použít jednoduchý DDA algoritmus, který rasterizuje úsečku ve 2D prostoru. Tento DDA algoritmus je třeba upravit, aby uvažoval průchod 3D prostorem. Varianta takového algoritmu se nazývá 3DDA algoritmus, který se hojně využívá v ray-

tracingu při procházení paprsku uniformní mřížkou, která je vytvořena okolo scény. Jak je vidět, uniformní mřížka je shodná s voxelizační mřížkou, kde jsou jednotlivé voxely uniformě rozprostřeny po celé scéně. Lze tedy aplikovat 3DDA algoritmus i na průchod paprsku voxelizační mřížkou. Tímto algoritmem jsem sice docílil kvalitních výsledků, ale za cenu značné ztráty výkonu. Ztráta výkonu je zaprvé způsobena tím, že musíme při každém vykonání fragment shaderu inicializovat parametry pro průchod mřížkou a zadruhé tím, že fragment shader musí vykonávat rozdílný počet cyklů při průchodu mřížkou, což se projevilo jako největší úzké hrdlo tohoto algoritmu. Proto jsem použil metodu uniformního vzorkování paprsku. Metoda spočívá v tom, že paprsek nadělíme na několik stejně velkých částí a v každé části paprsku vypočteme jeden bod. Tím dostaneme několik bodů, které nám popisují paprsek. Pro tyto body následně určíme, v kterém voxelu leží. Nyní už stačí zjistit, zda některý z těchto voxelů obsahuje nějaký polygon scény. Pseudokód 5.9, který popisuje metodu uniformího vzorkování paprsku je uveden níže.

Algoritmus 5.9: Algoritmus fragment shaderu pro průchod voxelovou mřížkou a jeho výstup, který vytváří bitovou stínovou mapu.

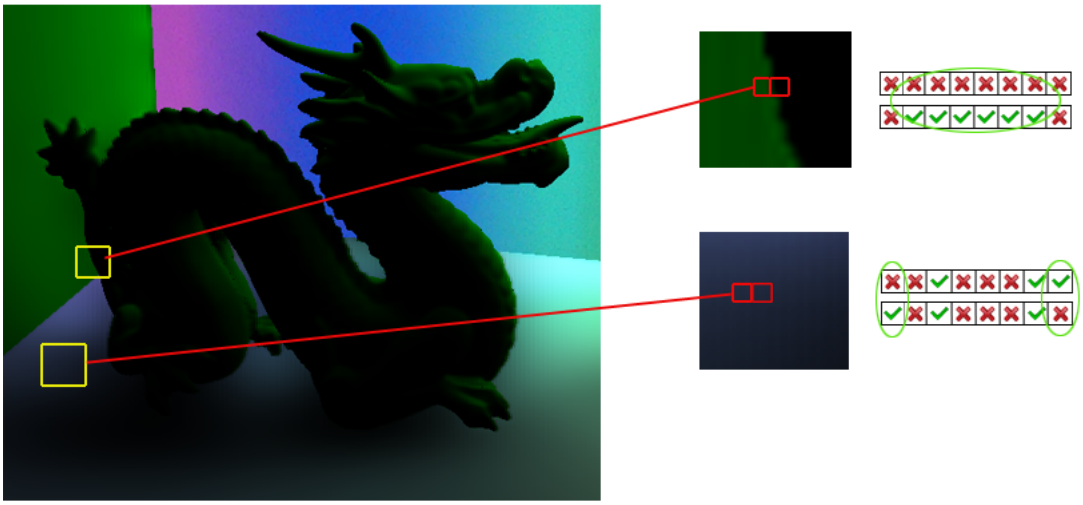
```
1  Vytvoř a nastav směr a počátek paprsku;
2  Inicializuj proměnné potřebné pro průchod paprsku mřízkou;
3  isVisible ← false;
   /* Určíme viditelnost paprsku                                    */
4  for i ← 0 to numberOfSteps do
5  |  voxelIndex ← worldPosToVoxelIndex(i);
6  |  if voxelHasGeometry(voxelIndex) then
7  |  |  isVisible ← true;
8  |  end
9  |  i += stepSize;
10 end
11 vplVisible ← setBitIndex(isVisible);
```

Generování stínové mapy

Nyní už víme, jak funguje voxelizace a umíme procházet pomocí paprsku voxelovou mřížku. Dalším bodem výpočtu viditelnosti je vygenerování bitové stínové mapy [15]. Bitová se nazývá z důvodu, kdy každý bit RGBA kanálu jednoho pixelu ve stínové mapě určuje, zda je VPL reprezentované tímto bitem zastíněné nebo nikoliv.

Výstupem algoritmu, který jsme použili při průchodu voxelovou mřížkou, je informace o zastínění VPL světla, pro které jsme prováděli průchod voxelovou mřížkou. Jelikož známe identifikátor VPL, který je uložen v VPL bufferu, můžeme jeho použitím jednoduše nastavit požadovaný bit ve stínové mapě. Bit nastavujeme pouze v případě, že VPL je zastíněné. Jinak necháme bit nastaven na hodnotu 0 (Obrázek 5.8).

Jak jsem nastínil na konci kapitoly 5.2, tento přístup k řešení viditelnosti je značně nevhodný při výpočtu pro plné rozlišení obrazu, kterého chceme docílit. Hlubší pohled na stínové mapy nám ukazuje, že stíny, které jsou generovány z bodového světla, jsou koherentní kromě okraje stínu. Toto platí u všech rozlišení stínové mapy. Proto jsem nejprve vytvořil bitovou stínovou mapu, která má rozlišení 64^2. Toto rozlišení plně dostačuje ke korektnímu určení viditelnosti pomocí hierarchické techniky. Zároveň vygenerování stínové mapy s tímto rozlišením (64^2) a použitím průchodu voxelovou mřížkou je značně rychlé



Obrázek 5.8: Každý pixel obsahuje bitové pole viditelných a zastíněných světel. Viditelnost přepočítáváme pouze v oblastech, které hodně mění viditelnost. Červené křížky značí zastínené VPL. Zelený zaškrtávací symbol reprezentuje viditelné VPL.

a dosahuje průměrně 56 snímků za sekundu. Tímto přístupem se zbavíme úzkého hrdla a zároveň dostaneme výkon, který se dá použít v interaktivních aplikacích. Pokud bychom stínovou mapu s takto nízkým rozlišením použili pro zobrazení stínů na plném rozlišení obrazu, dostaneme značně nekvalitní výsledky. Ty odstraníme určením oblastí, kde je třeba stínovou mapu přepočítat na vyšším rozlišení. Právě k tomuto použijeme hierarchickou techniku, která nám tento přístup umožní.

Při použití bitové stínové mapy, kde každý bit pixelu obsahuje informaci o viditelnosti příslušného VPL, potřebujeme dostatečný počet bitů, abychom byli schopni uložit viditelnost pro všech 256 VPL světel. Opět použijeme texturu s integerovým formátem pixelů, která nám dá celkem 128 bitů na RGBA kanál, což je polovina toho co potřebujeme. Zbylé bity dostanu připojením další textury k výstupnímu framebufferu.

Výpočet nespojitostí ve viditelnosti

Nespojitosti se určí na základě vygenerované bitové stínové mapy. Pro tento případ jsem použil techniku, která spočítá počet nastavených bitu v RGBA kanálu daného pixelu stínové mapy.

Nejprve si ale uvedeme, jak detekovat nespojitosti pomocí fragment shaderu. Při implementaci jsem vycházel z myšlenky, kterou jsem použil při určování nespojitostí v geometrii scény. Konkrétně při určovnání hloubkových nespojitostí. Jelikož víme, že hloubkové nespojitosti nejčastěji označí hrany objektů ve scéně, lze stejného efektu docílit i při určování nespojitostí ve viditelnosti.

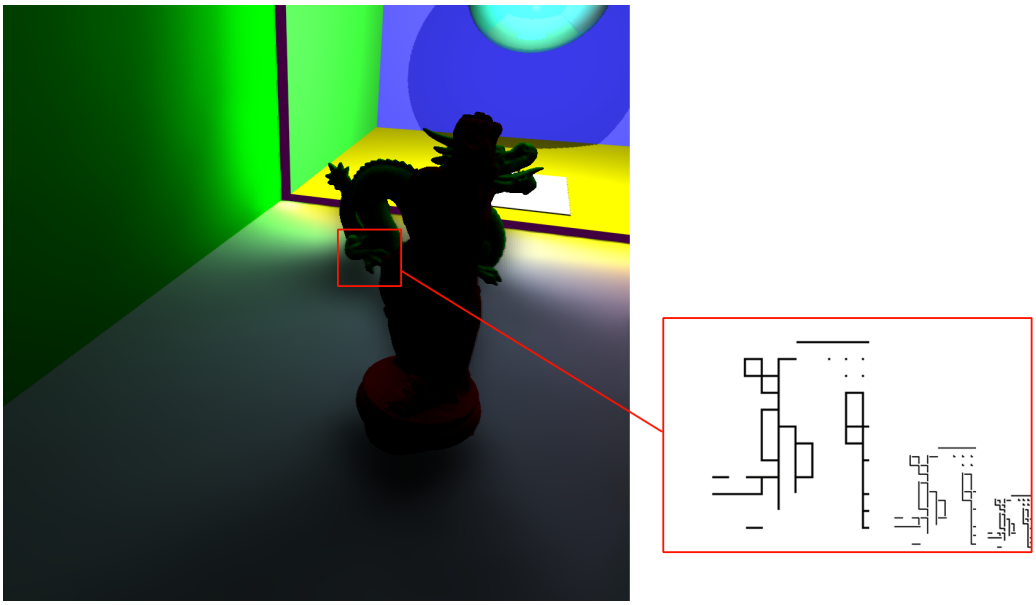
Bitová stínová mapa obsahuje nastavené bity v RGBA kanálu pro VPL světla, která jsou zastíněná a bity s nulovou hodnotou pro VPL světla, která jsou viditelná. Nespojitost následně vzniká v místech, kde se jednotlivé zastínění VPL světel markantně liší. Ve většině případů se zastínění liší v 5% VPL. Tím bychom přepočítávali mnoho pixelů stínové mapy, proto stačí vybrat pouze pixely, kde se viditelnost velmi liší.

Při vytváření stínové mipmapy použijeme vzorky z nižší úrovně mipmapy, kde se viditelnost jednotlivých VPL neliší. Tam, kde se viditelnost jednotlivých VPL liší provedeme

nový průchod paprsku voxelovou mřížkou a vypočítáme viditelnost VPL světel na vyšší úrovni mipmapy. Při kontrole nespojitostí ve viditelnosti používáme vždy 9-okolí aktuálně zpracovávaného vzorku. Výsledkem tohoto procesu je mipmapa, kde každý pixel mipmapy obsahuje maximální a minimální počet VPL světel, které se liší ve viditelnosti.

Následně je potřeba na základě mipmapy, kterou jsme získali v předchozím odstavci, určit pixely v hierarchickém bufferu, v kterých musíme přepočítat viditelnost. Viditelnost se přepočítá pouze v pixelech, které obsahují nespojitosti tedy, když rozdíl mezi maximální a minimální hodnotou změny ve viditelnosti bude vyšší než námi určený práh (Obrázek 5.9). Tento práh může být relativně vysoký. Ve svých scénách jsem použil práh o velikosti poloviny celkového počtu VPL světel. K identifikaci pixelů v hierarchickém bufferu opět použijeme stencil buffer jako v případě hloubkových a normálových nespojitostí.

Algoritmus, který ukazuje postup při určování nespojitostí ve viditelnosti je uveden v následujícím pseudokódu 5.10.



Obrázek 5.9: Výřez stencil bufferu. Pixely označeny černě potřebují přepočítat viditelnost.

Inkrementální výpočet stínové mapy a její kombinace

Na základě stencil bufferu z předchozí kapitoly jsme dostali pixely v hierarchickém bufferu, které obsahují nespojitosti ve viditelnosti. V označených pixelech provedeme nový výpočet viditelnosti pro VPL světla, u kterých se mění zastínění. Výpočet viditelnosti vždy provádíme na rozlišení stínové mapy, které odpovídá rozlišení úrovně v hierarchickém bufferu, z něhož jsme vybrali aktuální pixel. Viditelnost opět určíme průchodem paprsku přes voxelovou mřížku.

Nyní jsme dostali hierarchický buffer se znovu vypočítanou viditelností v místech, kde jsme detekovali nespojitosti ve viditelnosti. Následujícím krokem je jednotlivé úrovně hierarchického bufferu interpolovat a zkombinovat, abychom dostali výslednou stínovou mapu na plném rozlišení naší aplikace. V mém případě je plné rozlišení 1024^2.

Interpolace a kombinace hierarchického bufferu obsahujícího viditelnost funguje na úplně stejném principu. Začneme od nejnižší úrovně hierarchického bufferu, kterou převzorkujeme

Algoritmus 5.10: Fragment shader, který nastaví stencil bit tam, kde už není třeba dál přepočítávat viditelnost.

```
1  finerTexel ← textureLod(visibilityDeriv, texCoord, level);
2  coarserTexel ← textureLod(visibilityDeriv, texCoord, level + 1);
3  finerDiff ← finerTexel.max - finerTexel.min;
4  coarserDiff ← coarserTexel.max - coarserTexel.min;
   /* Zjištění, zda je potřeba počítat viditelnost na vyšší úrovni nebo
      nám dostačuje vzorek z nižší úrovně.                              */
5  needRefinement ← (finerDiff > visibilityThreshold);
6  coarseLevelSufficent ← (coarserDiff > visibilityThreshold);
7  if needRefinement and !atFinerLevel then
8  |  discard;
9  end
10 if coarseLevelSufficent and !atCoarseLevel then
11 |  discard;
12 end
13 nastav stencil bit ve stencil bufferu;
```

na rozlišení vyšší úrovně. Následně takto upravenou úroveň zkombinujeme s vyšší úrovní hierarchického bufferu. Princip je stejný, jak při výpočtu přímého osvětlení.

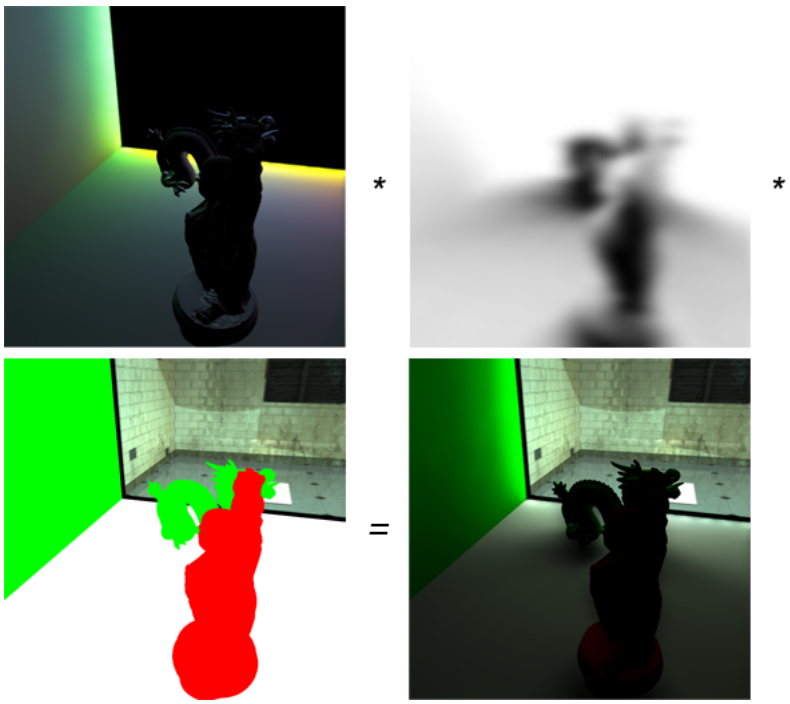
Po dokončení kombinace dostaneme interpolovanou a zkombinovanou stínovou mapu na nejvyšší úrovni hierarchického bufferu. Tato stínová mapa má rozlišení 1024^2, kterého jsme chtěli dosáhnout. Jednotlivé hodnoty pixelů ve stínové mapě nám aproximují funkci $V(x)$, která pochází z rovnice (3.7).

Zobrazení výsledného obrazu scény

V kapitole 5.1 jsem uvedl, jak vypočítat přímé osvětlení z plošného zdroje světla, které mi osvětluje scénu složenou z polygonů. Výsledné přímé osvětlení je uloženo v hierarchckém bufferu na jeho nejvyšší úrovni.

Kapitola 5.2 popsala implementaci viditelnosti pomocí hierarchické techniky, s jejíž pomocí jsem dokázal zrychlit vytváření bitové stínové mapy pomocí procházení voxelové mřížky natolik, aby se dala použít v interaktivních aplikacích. Výsledná stínová mapa je opět uložena v hierarchickém bufferu na jeho nejvyšší úrovni.

K dosažení výsledného obrazu je potřeba buffer s přímým osvětlením modulovat s difúzní barvou scény. Tím dostaneme výsledný obraz scény bez stínů. Stíny přidáme modulací toho obrazu se stínovou mapou, která ukládá poměr zastíněných a viditelných VPL světel v jednotlivých pixelech výsledného obrazu. Modulace k dosažení výsledného obrazu je znázorněna na obrázku 5.10. Výsledný obraz je předán do systémového framebufferu a následně zobrazen v okně OpenGL aplikace.



Obrázek 5.10: Vlevo nahoře: Přímé osvětlení scény z plošného světla. Vpravo nahoře: Výsledná stínová mapa. Vlevo dole: Difůzní barva scény. Vpravo dole: Finální složený obraz scény.

Kapitola 6

Zhodnocení aplikace implementující hierarchickou techniku

V této kapitole se budu zabývat zhodnocením a implementačními problémy hierarchické techniky, jenž jsem využil k implementaci přímého osvětlení z plošných světelných zdrojů s řešením viditelnosti. Následně se zaměřím na výkon, kterého jsem byl schopen dosáhnout použitím hierarchické techniky pro výpočet osvětlení.

6.1 Zhodnocení implementace

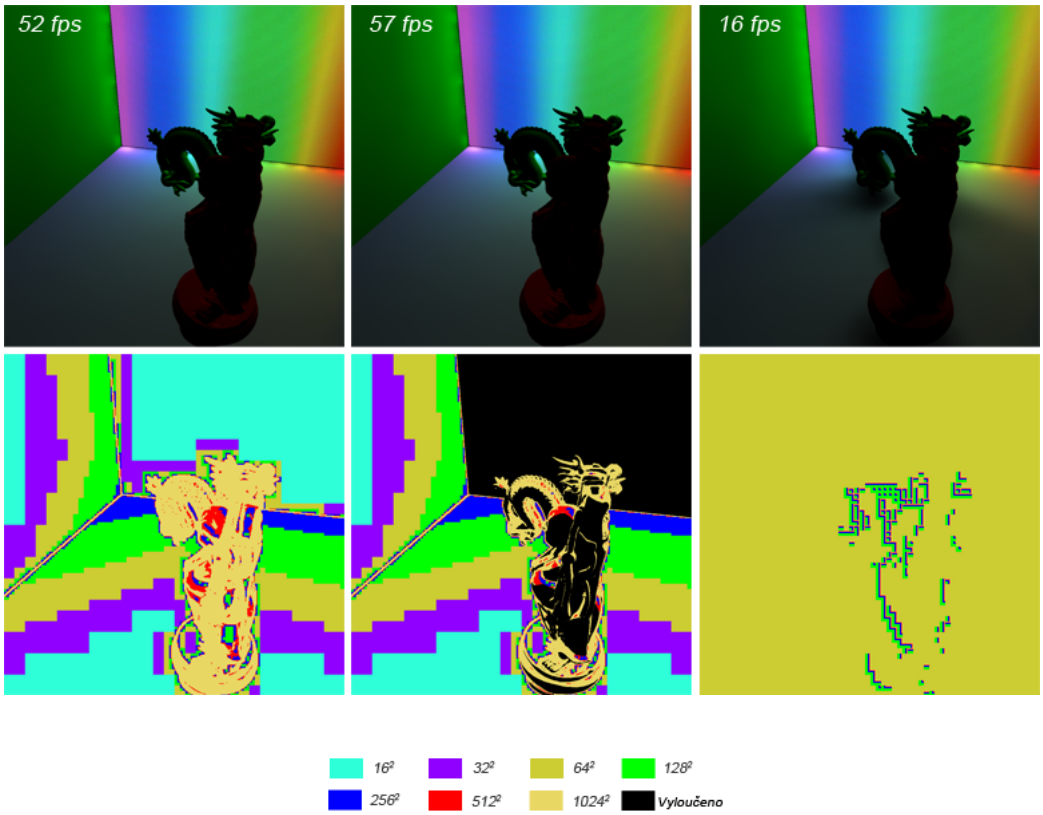
Interaktivní aplikaci pro výpočet přímého osvětlení s řešením viditelnosti jsem implementoval na sestavě s 3.7GHz AMD PhenomII X6 a grafické kartě AMD Radeon HD5770. K implementaci jsem použil technologii OpenGL v její nejnovější verzi 4.2. Všechny naměřené výsledky výkonu i problémy spojené s implementací pocházejí právě z implementace hierarchické techniky na této sestavě. Pokud není uvedeno jinak, všechny obrázky a výsledný výkon aplikace pochází z výstupního rozlišení mé aplikace, které činí 1024^2. Toto rozlišení jsem použil, abych mohl porovnat svoje výstupní výsledky aplikace, která implementuje hierarchickou techniku s výsledky, jež jsou uvedeny v dokumentu Interactive Multiresolution Image-Space Rendering for Dynamic Area Lighting [9], podle kterého jsem svoji aplikaci implementoval. Obrázek 1.1 demonstruje přínos realismu plošných světelných zdrojů přidaných do scény.

S technikou OpenGL jsem byl schopen docílit fungování celé aplikace a hierarchické techniky na grafické kartě. Klíčovou funkcí grafické karty je schopnost vytvářet a používat integerové textury a stencil buffer, abychom byli schopni odstranit nepotřebné fragmenty ze spracování fragment shaderem.

Integerové textury jsou využívány k uložení výstupu voxelizace a bitové stínové mapy. Textury s tímto formátem pixelů jsou nezbytné kvůli bitovým operacím, které se provádějí při průchodu voxelovou mřížkou a při nastavování viditelnosti jednotlivých VPL světel v rámci bitové stínové mapy. V mé implementaci jsem použil 2D textury s 32 bitovým integerovým vnitřním formátem RGBA kanálů pixelu. Tím jsem byl schopen u voxelizace vytvořit mřížku o velikosti $1024^2 \times 128$, kde 128 reprezentuje počet bitů pixelu integerové textury. Následně jsem experimentoval s rozlišením textury, která obsahuje voxelizaci. Viditelnost určená z $128^2 \times 128$ voxelové mřížky je skoro nerozeznatelná od viditelnosti vy-

počítané přes $1024^2 \times 128$ mřížku. Zatímco mřížky s nižším rozlišením dávají viditelně větší rozdíly. Tyto mřížky můžou postačovat k určení viditelnosti ve scénách s velkou nečlenitou geometrií, u nichž se viditelnost mění pouze pomalu. Doplňujícím faktorem je, že voxelové mřížky s menším rozlišením zvyšují rychlost průchodu paprsku mřížkou a tím zlepšují výkon aplikace. Nejdůležitělší u voxelových mřížek je rozlišení v z-ose. Malé rozlišení v tomto směru vede k samo stínícím artefaktům, kdy paprsek v dalším kroce zůstane uvnitř stejné buňky, což vede ke špatnému určení viditelnosti. Těmto artefaktům se říká shadow acne a lze se jim vyhnout přidáním biasu k počátku paprsku.

Ve své aplikaci používám k aproximaci plošného světla 256 VPL. Jelikož jedna integerová textura dokáže uložit 128 VPL světel s jejich viditelností, byl jsem nucen přidat další zobrazovací výstup k mnou používanému výstupnímu framebufferu. Tím jsem schopen uložit viditelnost všech 256 VPL. Toto jednoduché řešení má menší problém se ztrátou výkonu, kdy musím zapisovat do dvou výstupních textur zároveň. Při pozorování změn viditelnosti ve většině scén jsem zjistil, že lze použít prokládané vzorkovaní [5] na blocích pixelů o velikosti 2×2, právě s ohledem na postupné změny viditelnosti ve scéně. Tím bych byl schopen uložit viditelnost do jedné integerové textury, kde každý pixel by měl 16 bitů na jeden z RGBA kanálů. Tento postup jsem nebyl schopen implementovat z nedostatku času, proto je předmětem dalšího pokračování práce.



Obrázek 6.1: Použité rozlišení k zobrazení jednotlivých fragmentů obrazu. Výpočet probíhá na všech fragmentech (vlevo). Vyřazeny fragmenty, které jsou odvráceny od světla nebo leží za zdrojem světla (uprostřed). Výsledná kombinace stínové mapy, ukazuje fragmenty a rozlišení, z kterého byli použity (vpravo).

Další nutnou podmínkou ke zvýšení výkonu je nutnost použití stencil bufferu. Pomocí

stencil bufferu jsem byl schopen vyřadit nepotřebné fragmenty ze zpracování pomocí grafické karty. Jedným z možných způsobů, jak se vyhnout použití stencil bufferu, je implementovat if podmínku a nastavit požadovanou barvu pixelu ve fragment shaderu. Tento způsob je výrazně pomalejší a ústí k razantnímu snížení výkonu celé aplikace. Proto je doporučeno při implementaci hierarchických technik používat stencil buffer. Přínos stencil bufferu ukazuje obrázek 6.1, kde obrázek vlevo ukazuje využití stencil bufferu k určení plošek, které jsou hierarchicky rozděleny podle úrovně hierarchického bufferu. Obrázek uprostřed zobrazuje tu samou scénu, kde se navíc pomocí stencil bufferu vyberou zbytečné fragmenty, které jsou odvráceny od světelného zdroje, což ústí ke zvýšení výkonu. Poslední obrázek zobrazuje scénu s aplikovanou viditelností.

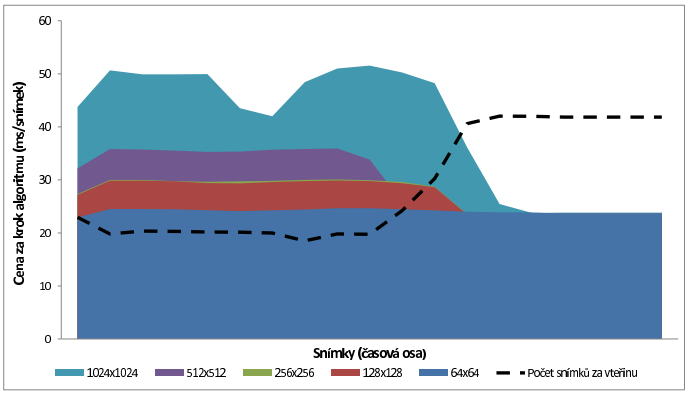
Generování stínů ve scéně je založeno na voxelizaci. Voxelizace je sama o sobě velice rychlá a její výpočet trvá okolo jedné milisekundy i pro relativně složité scény z pohledu geometrie scény. V mé aplikaci jsem použil voxelizaci, která vyplňuje i objem objektu, proto si neumí poradit s objekty, které jsou uvnitř duté, nebo obsahují otvory. Z tohoto pohledu jsem neměl žádný problém, protože ve svých scénách používám objekty, které mají celistvou geometrii. Největším problémem při výpočtu viditelnosti je dostatečně rychlý průchod paprsku voxelovou mřížkou. 3DDA algoritmus jsem neshledal jako dostatečně výkoný, protože někdy je třeba procházet mnoho zbytečných buňek, které nebudou obsahovat žádnou geometrii. Toto ústí k rozdílnému počtu cyklů ve fragment shaderu a k degradaci jeho výkonu. K řešení tohoto problému jsem implementoval uniformní vzorkování paprsku, které ve většině případů dává stejné výsledky jako 3DDA algoritmus, ale při mnohem vyšším výkonu. Problém s touto metodou nastává při detekování úzké geometrie ve scéně (např.: noha u židle, tenká stěna, atd.). Tento problém jsem vyřešil zahrnutím sousedních voxelů do určení viditelnosti, což prakticky rozšiřuje paprsek.

S problémem určování zastínění paprsku souvisí i velikost bitové stínové mapy. Čím větší stínová mapa je, tím větší musí být počet vysílaných paprsků. Můžeme si jednoduše odvodit, že pokud máme stínovou mapu s rozlišením 1024^2 a máme určit viditelnost pro 256 VPL světel, musíme vyslat celkem obrovský počet paprsků, což vede k vysoké degradaci výkonu. Naštěstí lze použít lehce upravenou hierarchickou techniku, kterou jsem použil při výpočtu přímého osvětlení. Stačí mi vytvořit stínovou mapu na rozlišení 64^2 a dodatečně ji přepočítat pouze v místech, kde hierarchická technika detekuje nespojitosti ve viditelnosti. Když jsem experimontoval se základní velikostí stínové mapy, došel jsem k závěru, že plně dostačující je právě moje použité rozlišení o velikosti 64^2. Při použití vyššího rozlišení jsem dostal stejné výsledky. Naopak nižší rozlišení než 64^2 už snižuje výslednou kvalitu stínů, protože nedokáže detekovat většinu geometrických složitostí ve scéně. Na obrázku 6.4 je ukázán graf, který ukazuje rozdíl mezi výpočtem stínové mapy s naivní metodou průchodu paprsku a metodou s použitím hierarchické techniky.

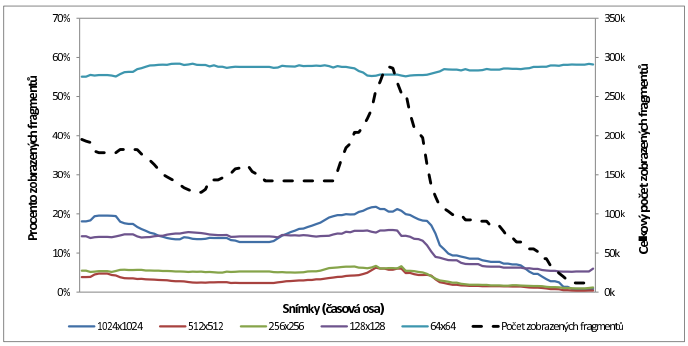
6.2 Zhodnocení výkonu

Po zhodnocení některých implementačních částí hierarchické techniky, které byly z mého pohledu důležité, se dostaváme ke stránce výkonu. Hodnotit výkon budu podle výsledků mé aplikace, která implementuje hierarchickou techniku pro výpočet přímého osvětlení z plošných světelných zdrojů včetně řešení viditelnosti. Všechny výsledky v rámci výkonu jsou uvedeny při použití scény s jedním plošným světlem, na kterém běží video a dvěma modely (buddha, stanford dragon).

U hierarchických technik nezáleží na geometrické složitosti scény. I pro scény s velkým počtem polygonů můžeme dostat relativně vysoký výkon s aplikací, která hierarchickou

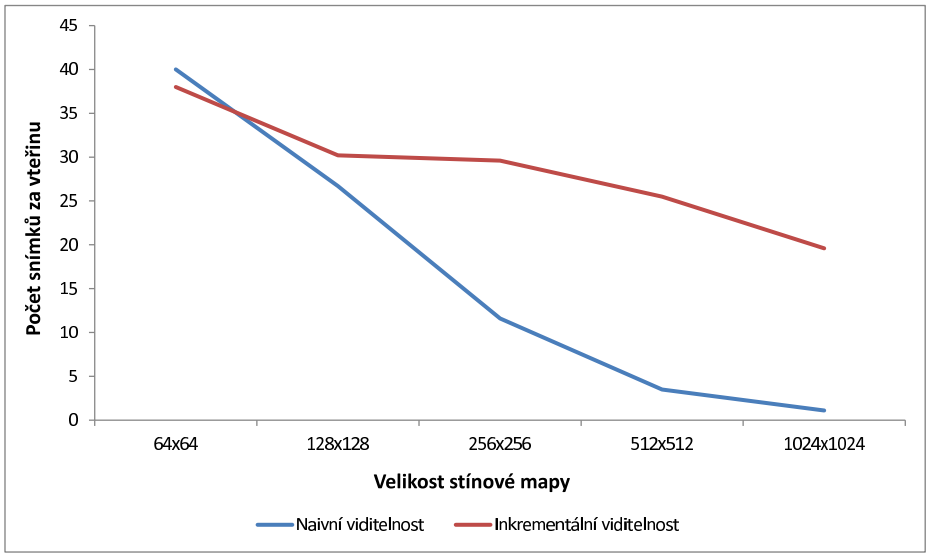


Obrázek 6.2: Graf znázorňuje čas potřebný k zobrazení jednotlivých úrovní hierarchického bufferu při průletu kamery scénou. Uvedena je také výsledná křivka výkonu.



Obrázek 6.3: Procentuální statistika, která ukazuje jaká část fragmentů se zobrazí z jednotlivých úrovní hierarchického bufferu. Opět byl použit stejný průlet kamerou jako u obrázku 6.2. Graf také ukazuje celkový počet fragmentů, které se v jednotlivých snímcích musely zobrazit.

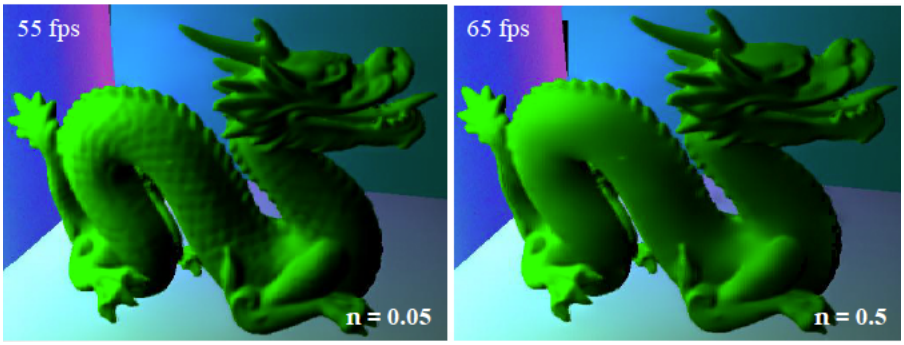
techniku používá. Je to z důvodu, kdy se polygony ve scéně rasterizují pouze dvakrát za výpočet jednoho snímku a to v případech výpočtu voxelizace a geometrického bufferu. Všechny další kroky hierarchické techniky používají pouze fragment shader a tedy pracují pouze s fragmenty hierarchického bufferu. Jednoduchou dedukcí dostaneme, že největší dopad na výkon má právě vizuální zpracování scény. Víceméně počet fragmentů, které musíme zpracovat fragment shaderem, ovlivňuje výkon. Celkový počet fragmentů, jenž musíme zpracovat, nám udává hierarchický buffer, v kterém jsou označeny jednotlivé fragmenty po určení nespojitostí v geometrii scény. Další fragmenty na zpracování nám vzniknou při výpočtu nespojitostí ve viditelnosti, z kterých vygenerujeme korektní stíny ve scéně. Scény s vysokými detaily v osvětlení nebo viditelnosti potřebují více fragmentů k zachycení těchto detailů. Tuto skutečnost nám zachycují grafy na obrázku 6.2 a 6.3, kde jsem použil scénu s pohybující se kamerou, která v různých snímcích zachycuje scénu z různých uhlů s různou vizuální složitostí. Dolní graf ukazuje procenta z celkového počtu fragmentů, které byly zobrazeny na jednotlivých úrovních hierarchického bufferu. Z úrovně s rozlišením 64^2 zobrazujeme skoro polovinu celkového počtu fragmentů. Vyšší úrovně většinu těchto fragmentů znovu použíjí a interpolují. Pouze několik málo procent fragmentů potřebuje být spočítáno na nejvyšší úrovní hierarchického bufferu. Zpracovávání malého procenta fragmentů v rozlišení 1024^2 je stejně náročnější než zpracování velkého procenta fragmentů na 64^2. Tuto



Obrázek 6.4: Rozdíl ve výkonu mezi naivní metodou výpočtu stínové mapy (per-pixel) a inkrementální metodou.

skutečnost odhaluje horní graf, který ukazuje časovou náročnost, kterou musíme zaplatit za každou úroveň hierarchického bufferu.

Vizuální složitost hierarchických technik také významně ovlivňují prahy, s nimiž určujeme nespojitosti v osvětlení a viditelnosti. Největší dopad na výkon má práh při výpočtu nespojitostí v zakřivení povrchu, jak je ukázáno na obrázku 6.5. Čím nižší práh použijeme, tím více fragmentů musíme zobrazit na vyšších úrovních hierarchického bufferu, což vede ke snížení výkonu aplikace. Na druhé straně dostaneme vyšší vizuální kvalitu scény. Podobného efektu dosáhneme s prahem pro viditelnost. Z mých zkušeností ovšem plyne, že práh při určování viditelností stačí nastavit na polovinu VPL světel. S tímto prahem jsem dosáhl stejné vizuální kvality stínů, jako s nižšími prahy.



Obrázek 6.5: Ukázka vizuální kvality modelu draka při použití různých prahů při detekci nespojitostí. S nízkým prahem při určování nespojitostí v zakřivení povrchu dosáhneme vysoké vizuální kvality za cenu nižšího výkonu (vlevo). Naopak s vysokým prahem ztratíme detail, ale výkon stoupne.

Samozřejmě je možné vylepšit výkon celé hierarchické techniky například tím, že fragmenty zobrazujeme pouze do úrovně s rozlišením 256^2 nebo 512^2. Výsledný obraz na rozlišení 1024^2 poté dopočítáme aplikováním bilaterálních filtrů [17].

Kapitola 7

Závěr

V mé práci jsem se věnoval interaktivním hierarchickým technikám, které umožňují počítat globální efekty světelného záření pomocí grafických karet. Z těchto globálních efektů jsem se konkrétně zaměřil na výpočet přímého a nepřímého osvětlení.

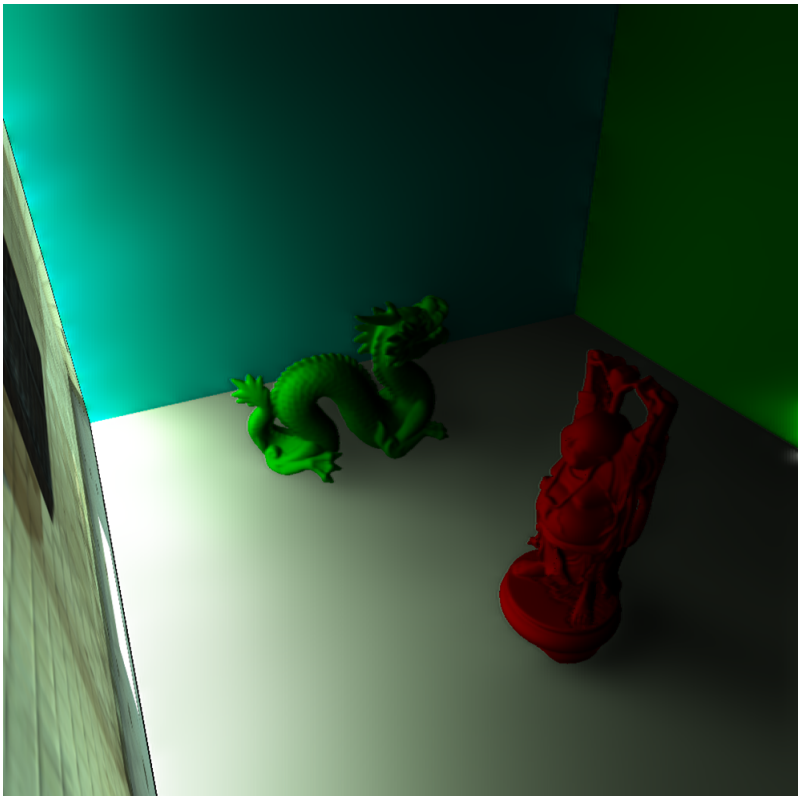
V druhé kapitole práce 2 jsem uvedl rozdíly mezi lokálními a globálními metodami pro výpočet osvětlení. Zjistil jsem, že lokální metody jsou sice velice jednoduché na výpočet, ale neumožňují vytvářet pokročilé světelné jevy, jako jsou odrazy a lomy světla v prostředí. Naopak globální metody umí aproximovat osvětlení ve scéně tak, aby se co nejvíce podobalo skutečnosti. Bohužel tyto metody jsou velice náročné na výpočet, a proto se nedají použít v interaktivních aplikacích.

Třetí kapitola 3 se nejprve zabývala hierarchickou technikou, která umožňuje počítat nepřímé osvětlení ve scéně. Tento přístup byl následně použit v hierarchické radiozitě. Samotný algoritmus pracuje pouze s daty obrazového prostoru, tedy pouze s tím co vidí pozorovatel. Tím a dalšími vylepšeními v podobě výpočtu osvětlení v pixelech, v kterých je to nezbytné, lze docílit vysoké rychlosti výpočtu, která se dá následně použít v interaktivních aplikacích. Vylepšený přístup, který je aplikován v hierarchické radiozitě, lze použít i pro výpočet přímého osvětlení z plošných světelných zdrojů. Právě tato technika je ve třetí kapitole uvedena po hierarchické radiozitě.

Tato práce obsahuje zajímavé způsoby výpočtu globálního osvětlení ve scéně, které se používají v hierarchických algoritmech. Tyto algoritmy mají několik zásadních výhod oproti již vynalezeným metodám výpočtu globálního osvětlení (radiozita, raytracing, atd). První výhodou je, že pracují pouze s částí scény, kterou vidí pozorovatel, tedy neplýtvají výpočetními prostředky na výpočet osvětlení v celé scéně. Nevyžadují žádné předzpracování scény, které významně ničí interaktivnost jednotlivých metod, jakými radiozita a raytracing jsou. Hierarchické metody používají stencil buffer, jenž je využit při výpočtu osvětlení tak, abychom byli schopni identifikovat pixely, v kterých je důležilé osvětlení počítat. Tento přístup značně urychluje výpočet globálního osvětlení, neboť ho lze akcelerovat použitím grafické karty.

Ve čtvrté kapitole 4 jsem popsal návrh a analýzu aplikace, která implementuje hierarchickou techniku výpočtu přímého osvětlení z plošných světelných zdrojů (Obrázek 7.1). Zabýval jsem se také technologiemi, které jsem použil k implementaci hierarchické techniky.

Pátá kapitola 5 obsahuje implementační detaily hierarchické techniky, která byla použita pro výpočet přímého osvětlení z plošných světelných zdrojů a dále k řešení viditelnosti. Celá kapitola je rozdělena na dvě části. První část se zabývá pouze implementací přímého osvětlení pomocí hierarchické techniky. Druhá část obsahuje detailní implementaci viditelnosti, která byla opět adaptována na hierarchickou techniku. U některých částí implementace jsou



Obrázek 7.1: Ukázka výstupu mé aplikace, která implementuje hierarchickou techniku pro výpočet přímého osvětlení a viditelnosti z plošných světelných zdrojů.

přiloženy zdrojové pseudokódy pro lepší názornost algoritmů, které hierarchická technika využívá.

Poslední kapitola 6 hodnotí dosažené výsledky aplikace a více rozebírá implementační problémy, které jsem řešil při implemetaci hierarchické techniky pro výpočet přímého osvětlení a viditelnosti z plošných světelných zdrojů.

Nakonec bych chtěl připomenout, že při studování hierarchických technik, které dokáží počítat globální osvětlení, které se dá použít v interaktivních aplikacích, jsem si připoměl základní pojmy, které se při výpočtu osvětlení používají. Následně jsem se naučil používat nové techniky a možnosti, které nabízejí dnešní grafické karty a významně tak urychlují výpočet globálního osvětlení ve scéně.

Literatura

[1] Cook, R. L.; Torrance, K. E.: *A reflectance model for computer graphics.* ACM SIGGRAPH, 1982, 307–316 s.

[2] Eisemann, E.; Décoret, X.: *Fast scene voxelization and applications.* ACM SIGGRAPH, 2006, 71–78 s.

[3] Gouraud, H.: *Continuous shading of curved surfaces.* IEEE Transactions on Computers, 1976, 623–629 s., iSSN 0018-9340.

[4] Kajiya, J. T.: *The rendering equation.* SIGGRAPH, 1986.

[5] Keller, A.; Heidrich, W.: *Interleaved Sampling.* In Proc. Eurographics Workshop on Rendering, 2001, 269–276 s.

[6] Lafortune, E. P. F.; Foo, S.-C.; Torrance, K. E.; aj.: *Non-linear approximation of reflectance functions.* SIGGRAPH, 1997, 117–126 s.

[7] Lauritzen, A.: *Deferred Rendering for Current and Future Rendering Pipelines.* Beyond Programmable Shading, SIGGRAPH 2010, 2010.

[8] Lepil, O.: *Fyzika pro gymnázia Optika.* Prometheus Praha, 2002, 135–140 s., iSBN 80-7196-237-6.

[9] Nichols, G.; Penmatsa, R.; Wyman, C.: *Interactive Image-Space Rendering for Dynamic Area Lighting.* Computer Graphics Forum 29(4), 2010, 1279–1288 s.

[10] Nichols, G.; Shopf, J.; Wyman, C.: *Hierarchical Image-Space Radiosity for Interactive Global Illumination.* Computer Graphics Forum 28(4), 2009, 1141–1149 s.

[11] Nichols, G.; Wyman, C.: *Multiresolution splatting for indirect illumination.* ACM Symposium on Interactive 3D Graphics and Games, 2009, 83–90 s.

[12] Nicodemus, F. E.; Richmond, J. C.; Hsia, J. J.; aj.: *Geometrical considerations and nomenclature for reflectance.* NBS Monograph 160, 1977.

[13] Phong, B. T.: *Illumination for computer generated images.* Communications of the ACM, 1975, 311-317 s., article.

[14] Ritschel, T.; Grosch, T.; Kim, M. H.; aj.: *Imperfect shadow maps for efficient computation of indirect illumination.* ACM SIGGRAPH, 2008, iSSN 0730-0301.

[15] Schwarz, M.; Stamminger, M.: *Bitmask soft shadows.* Computer Graphics Forum 26(3), 2007, 515–524 s.

[16] Takafumi, S.; Tokiichiro, T.: *Comprehensible rendering of 3-D shapes.* ACM SIGGRAPH, 1990, 197–206 s., iSSN 0097-8930.

[17] Tomasi, C.; R.Manduchi: *Bilateral Filtering for Gray and Color Images.* IEEE Computer Society, 1998, 839– s., iSBN 81-7319-221-9.

[18] Wallace, J. R.; Elmquist, K. A.; Haines, E. A.: *A ray tracing algorithm for progressive radiosity.* ACM SIGGRAPH, 1989, iSBN 0-89791-312-4.

[19] Ward, G. J.: *Measuring and modeling anisotrophic relfection.* ACM SIGGRAPH, 1992, 265–272 s., iSBN 0-89791-479-1.

Příloha A

Obsah CD

- Implementované zdrojové kódy vertex a fragment shaderů
- Zdrojové kódy implementovaných tříd v jazyce C++
- Zdrojový kód aplikace implementující hierarchickou techniku popsanou v této diplomové práci
- Elektronická verze diplomové práce
- Demo videa demonstrující moji aplikaci využívající hierarchickou techniku