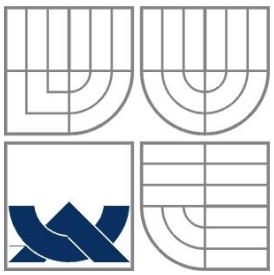
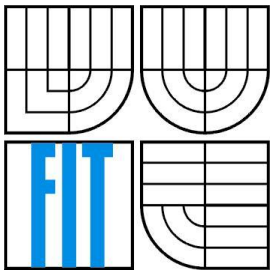


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VOXEL CONE TRACING

VOXEL CONE TRACING

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE
AUTHOR

BC. MICHAL PRACUCH

VEDOUCÍ PRÁCE
SUPERVISOR

ING. TOMÁŠ MILET

BRNO 2016

Abstrakt

Tato práce se zabývá výpočtem globálního osvětlení ve scéně pomocí metody Voxel Cone Tracing. Jejím základem je voxelizace scény z trojúhelníkové reprezentace. Voxelizace může být prováděna do plné pravidelné 3D mřížky (textury) nebo do hierarchického Sparse Voxel Octree (řídkého voxelového stromu) pro ušetření paměťového prostoru. Tato voxelová reprezentace je dále využita pro výpočty globálního nepřímého osvětlení v reálném čase v normálních trojúhelníkových scénách pro zlepšení realističnosti finálního obrazu. Hodnoty z voxelů jsou získávány sledováním kuželových paprsků z pixelů, pro které chceme zjistit osvětlení.

Abstract

This thesis deals with the global illumination in the scene by using Voxel Cone Tracing method. It is based on the voxelization of a triangle mesh scene. The voxels can be stored to a full regular 3D grid (texture) or to the hierarchic Sparse Voxel Octree for saving of the memory space. This voxel representation is further used for computations of the global indirect illumination in real time within normal triangle mesh scenes for more realistic final image. Values from the voxels are obtained by tracing cones from the pixels which we want to get illumination for.

Klíčová slova

Voxel Cone Tracing, Voxelizace, Sparse Voxel Octree, globální osvětlení, sledování kuželových paprsků

Keywords

Voxel Cone Tracing, Voxelization, Sparse Voxel Octree, Global Illumination, Cone Tracing

Citace

PRACUCH, Michal. *Voxel Cone Tracing*. Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Milet Tomáš.

Voxel Cone Tracing

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Tomáše Mileta. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……

Michal Pracuch

25. 5. 2016

Poděkování

Chtěl bych poděkovat vedoucímu práce inženýru Tomáši Miletovi za poskytnutou odbornou pomoc, znalosti, čas a trpělivost.

Obsah

1 Úvod

V dnešní době je stálá snaha o co největší realističnost renderovaných obrázků, zvláště pak v reálném čase pro interaktivní aplikace. Velmi důležitou součástí realistického zobrazení scény je globální osvětlení (podrobněji popsané v kapitole **2.1**). Globální osvětlení neboli nepřímé osvětlení je způsob výpočtu osvětlení objektu v závislosti nejen na světle přicházejícího přímo ze světelného zdroje, ale také na světle, které přichází od ostatních objektů ve scéně. Dalšími efekty globálního osvětlení jsou měkké stíny, přenos barvy mezi blízkými objekty, kaustiky a *ambient occlusion*. Existuje velké množství metod pro přesný výpočet globálního osvětlení, např. *path tracing*, *photon mapping* i aproximační metody pracující v reálném čase, např. *Reflective Shadow Maps* a *Light Propagation Volumes* (popsané v kapitolách **2.1.1** až **2.1.7**).

Další aproximační metodou je *Voxel Cone Tracing* (sekce **2.6**), na kterou je zaměřena tato diplomová práce. *Voxel Cone Tracing* je založen na tříkrokovém algoritmu. Nejprve je příchozí záření (energie a směr) z dynamických zdrojů světla uloženo do listů hierarchického *sparse voxel octree* (řídkého voxelového stromu). To je provedeno rasterizací scény ze všech světelných zdrojů a uložením fotonu pro každý viditelný fragment povrchu. Ve druhém kroku je provedena filtrace příchozích hodnot záření do vyšších úrovní stromu (podvzorkování). Nakonec je scéna vyrenderována z pohledu kamery. Pro každý viditelný fragment povrchu je zkombinováno přímé a nepřímé osvětlení. Přibližné sledování kuželového paprsku je použito pro vykonání finálního shromáždění světelné informace, vysláním několika kuželů přes polokouli nad fragmentem k posbírání osvětlení uloženého ve voxelovém stromu.

Tato metoda využívá voxelizaci scény, což je převod trojúhelníkové reprezentace geometrie na voxely (kapitola **2.2**). Jednoduchý algoritmus voxelizace pracuje ve čtyřech hlavních krocích. Nejprve je pro každý trojúhelník sítě vybrána dominantní osa jeho normály. Pak je podle této osy ortograficky promítnut a konzervativně rasterizován. Posledním krokem je uložení vygenerovaných fragmentů do korespondujících cílových voxelů podle jejich 3D souřadnic. Voxelizace může být prováděna do plné pravidelné 3D mřížky (textury) nebo do hierarchického *sparse voxel octree* (SVO) pro ušetření paměťového prostoru.

Jednoduchá voxelizace do 3D textury je popsána v sekci **2.4**, spolu s konzervativní rastericací nutnou pro zvoxelizování všech (i velmi malých) trojúhelníků (**2.4.1**) a skládáním voxel fragmentů pomocí atomických operací a simulací atomického sčítání nad neceločíselnými datovými typy (**2.4.2**). Voxelizace do SVO je v sekci **2.5**, spolu s popisem octree struktury, do které jsou ukládány výsledky (**2.5.1**), přehledem algoritmu sparse voxelizace, kterým je budován SVO v lineární paměti GPU (**2.5.2**), konstrukcí seznamu voxel fragmentů, který slouží pro dočasné uchování voxel fragmentů, aby nebylo nutné provádět voxelizaci pokaždé znovu pro každou úroveň stromu (**2.5.3**), dělením uzlů, alokací nových poduzlů a jejich inicializací ve stromu při budování samotného SVO (**2.5.4**) a

zapisování hodnot voxel fragmentů do listů stromu (voxelů) a jejich podvzorkování do vyšších úrovní stromu (**2.5.6**).

Tuto voxelovou reprezentaci lze dále využít pro výpočty globálního nepřímého osvětlení v reálném čase v normálních trojúhelníkových scénách pro zlepšení realističnosti finálního obrazu. Hodnoty z voxelů jsou získávány sledováním kuželových paprsků z pixelů, pro které chceme zjistit osvětlení. Interaktivním nepřímým osvětlením se zabývá sekce **2.6**, spolu se souhrnem algoritmu osvětlení, hierarchickou voxelovou strukturou a jejím popisem s přidanými ukazateli na sousední uzly a strukturou voxelů v centrech uzlů (**2.6.3**). V sekci **2.6.7** je popsáno aplikování této reprezentace pro výpočty ambient occlusion ve scéně; rozšíření tohoto přístupu je použito pro výpočty globálního nepřímého osvětlení ve scéně v reálném čase (**2.6.9**) a využití sledování kuželových paprsků pro finální sbírání světelných informací (**2.6.10**). Nakonec je sekce o ukládání přímého osvětlení do SVO s přenosem hodnot do sousedních uzlů a distribucí přes úrovně (**2.6.11**).

Návrh aplikace založené na *Voxel Cone Tracing* a její struktura jsou popsány v kapitole **3.2** a v jejích podkapitolách jsou pak podrobněji popsány jednotlivé části, jako jsou inicializace scény, voxelizace, budování řídkého voxelového stromu, vytvoření *bricks*, jejich MIP mapování, ukládání fotonů a finální vykreslení. Rozdíly proti původnímu *Voxel Cone Tracing* jsou v kapitole **3.1**. Následně jsou popsány použité knihovny při implementaci aplikace (**4.1**) a některé vybrané implementační detaily (**4.2**) týkající se alfa kanálu při voxelizaci, práce s bufferem seznamu voxel fragmentů, kódování vícerozměrných vektorů na celočíselný typ, výpočet klouzavého průměru pomocí atomických operací, volba lokálních velikostí v pracovních skupinách a spouštění a ovládání aplikace (**4.3**). Na závěr jsou prezentovány dosažené výsledky (**5.1**) a měření paměťových nároků použitých struktur (**5.2**) a časů provádění jednotlivých algoritmů (**5.3**).

Tato práce navazuje na semestrální projekt, jehož součástí byl teoretický popis voxelizace obecně (**2.2**), jednoduché voxelizace (**2.4**), voxelizace do sparse voxel octree (**2.5**) a interaktivního nepřímého osvětlení (**2.6**). Na základě tohoto popisu je zpracován návrh aplikace (**3.2**).

2 Úvod do problematiky

Tato kapitola začíná charakteristikou globálního osvětlení a popisuje různé přesné metody a aproximační metody pracující v reálném čase jeho vytvoření v produkovaném obraze. Další část o voxelizaci vychází z článku o voxelizaci na GPU [1], z dizertační práce zaměřené na voxely [2] a článku o globálním osvětlení s využitím voxelů [3], jejichž hlavním autorem je Cyril Crassin, a popisuje tvorbu voxelové reprezentace scény z klasické trojúhelníkové reprezentace. Nejprve je popsán algoritmus, který produkuje pravidelnou 3D texturu s využitím GPU hardwarového rasterizéru a rozhraní pro přístup k paměti v shaderech. V druhé části je popsáno rozšíření tohoto přístupu, který dovoluje budování *sparse* (*řídké*) voxelové reprezentace ve formě *octree* struktury. Tento přístup nepoužívá vytváření přechodné plné pravidelné mřížky při budování struktury a konstruuje strom přímo, aby byl použitelný pro velké scény. Dále je popsán algoritmus pro interaktivní nepřímé osvětlení scény, který využívá data z voxelové reprezentace a vytváří tak realističtější finální obraz. Hodnoty z voxelů jsou získávány sledováním kuželových paprsků z pixelů, pro které chceme zjistit osvětlení. Nakonec je popsána architektura grafických karet, jejich výpočetní jednotky, paměťová hierarchie a OpenGL pipeline.

2.1 Globální osvětlení

Globální osvětlení neboli nepřímé osvětlení je způsob výpočtu osvětlení objektu v závislosti nejen na světle přicházejícího přímo ze světelného zdroje, ale také na světle, které přichází od ostatních objektů ve scéně, kdy dochází k mnohonásobným odrazům paprsků světla ze stejného světelného zdroje od všech povrchů ve scéně [5]. Dalšími efekty globálního osvětlení jsou měkké stíny (světelný zdroj má plochu), přenos barvy (z angl. *color bleeding*) mezi blízkými objekty, způsobené barevnými odlesky, kaustiky (koncentrace lomeného světla přes sklo) a ambient occlusion [6][7][8]. Tyto efekty jsou kritické pro realistickou syntézu obrazu, bez nich osvětlení většinou vypadá ploše a uměle.

Nepřímé osvětlení velmi vylepšuje realističnost renderované scény, ale většinou přichází s významnou cenou, protože komplexní scény jsou náročné na osvětlování, zvláště s přítomností lesklých odrazů [2]. Globální osvětlení je výpočetně náročné z několika důvodů. Vyžaduje výpočet viditelnosti mezi libovolnými body v 3D scéně, což je složité u renderování založeného na rasterizaci. Dále vyžaduje integraci světelné informace přes velký počet směrů pro každý stínovaný bod. V dnešní době, kdy se komplexnost renderovaného obsahu blíží k milionům trojúhelníků dokonce i ve hrách, výpočet nepřímého osvětlení v reálném čase v takových scénách je hlavní výzva s vysokým industriálním dopadem. Kvůli omezením v reálném čase nejsou *off-line* algoritmy používané v odvětví speciálních efektů vhodné a jsou nutná rychlá, přibližná a adaptivní řešení. Spoléhání se na předpočítané osvětlení je velmi omezující, protože běžné efekty jako dynamické světelné zdroje a lesklé materiály jsou jen zřídka řešeny.

Existují dobře zavedená off-line řešení pro přesný výpočet globálního osvětlení jako *path tracing* popsaný Kajiyem [8] (kapitola **2.1.1**) nebo *photon mapping* popsaný Jensenem [6] (kapitola **2.1.2**) a radiozita od Goral et al. [9], omezená jen na difuzní povrchy (kapitola **2.1.3**). Dále byly rozšířeny o optimalizace, které často využívají geometrické zjednodušení (Tabellion a Lamorlette [10], Christensen a Batali [11]) nebo hierarchické struktury scény, ale nedosahují na vykonávání v reálném čase. Rychlé, ale paměťově náročné přepočítání osvětlení pro statické scény je možné (Lehtien et al. [12]), ale zahrnuje pomalý předzpracovávající krok. *Antiradiance* (Dachsbacher et al. [13], kapitola **2.1.4** a Dong et al. [14]) dovoluje pracovat s viditelností nepřímo pomocí vyzařování negativního světla a dosahuje interaktivní rychlosti pro několik tisíc trojúhelníků.

Pro dosažení většího počtu snímků za sekundu je často přenos světla diskretizován. Zvláště koncept *VPLs* je zajímavý, vytvořený Kellerem [15] (kapitola **2.1.5**), kde odražené přímé světlo je vypočítáno pomocí sady *virtuálních bodových světel* (z angl. *virtual point lights*). Pro každé takové VPL je vypočítána shadow mapa, což je často nákladné. Laine et al. [16] navrhli opětovné využití shadow map ve statických scénách. Zatímco je tento přístup velmi elegantní, rychlý pohyb světla a komplexní geometrie ve scéně mohou ovlivnit míru znuvupoužitelnosti. Walter et al. [17] využívají rozdělení světel do hierarchických clusterů VPLs pro každý pixel, zatímco Hašan et al. [18] rozšířili tento nápad dále o zahrnutí hrubě vzorkovaných viditelnostních vztahů. V obou případech se nelze vyhnout nákladnému výpočtu shadow map a výsledkem tak není výkon v reálném čase. *ISM* (*Imperfect Shadow Maps*) [20] a *Microrendering* [21] od Rischela et al. dosahují výkonu v reálném čase použitím aproximace scény založené na bodech pro zrychlení renderování do VPL frusta, ale nedokážou jednoduše zajistit dostatečnou přesnost pro blízkou geometrii.

Dnes dostupné nejvíce efektivní řešení v reálném čase pracují v prostoru obrázku současného pohledu jako *Reflective Shadow Maps* od Dachsbachera a Stammingera [19] (kapitola **2.1.6**), ale ignorují informace mimo obrazovku (Nichols et al. [22]). Přístup Crassina ve *Voxel Cone Tracing* [2][3] (podrobně popsaná v kapitole **2.6**) reprezentuje přímé osvětlení hierarchicky a používá sledování kuželového paprsku, které nahrazuje výpočty shadow map pro zrychlení generace obrázků. Tento přístup je méně efektivní než nejrychlejší řešení, ale nepotřebuje podobně silnou aproximaci. Především je dosaženo vysoké přesnosti v blízkosti pozorovatele, což je důležité pro správné vnímání povrchu.

Práce Crassina [2][3] odvozuje hierarchickou reprezentaci scény, která tvoří pravidelnou strukturu pro usnadnění přenosu světla a dosažení výkonu v reálném čase, podobně v duchu práce Kaplanyana et al. *Light Propagation Volumes* [23] (kapitola **2.1.7**), kteří implementují difuzní nepřímé osvětlení použitím difuzního procesu v sadě vložených pravidelných voxelových mřížek. Zatímco relativně rychlý, tento přístup trpí nedostatkem přesnosti vzniklým relativně malým rozlišením voxelové mřížky, která může být použita. Toto rozlišení je omezeno cenou difuzního procesu a také obsazením paměti. Namísto spoléhání na difuzní proces, využívá Crassin přístup sledování paprsku pro sběr záření a okluze ve scéně při zachování velké přesnosti. Tento nedostatek

přesnosti limituje Kaplanyanův přístup na difuzní nepřímé osvětlení, zatímco Crassinův zvládá jak difuzní, tak spekulární nepřímé osvětlení.

Dále budou vybrané metody pracující off-line i v reálném čase popsány podrobněji v kapitolách **2.1.1** až **2.1.7** a *Voxel Cone Tracing* je detailně popsán v kapitole **2.6**.

2.1.1 Path tracing

V roce 1986 představil Jamce T. Kajiya *renderovací rovnici* [8], která generalizovala rozmanité známé renderovací algoritmy a poskytovala sjednocený kontext pro pohled na ně jako více či méně přesnou aproximaci řešení jediné rovnice. Renderovací rovnice je:

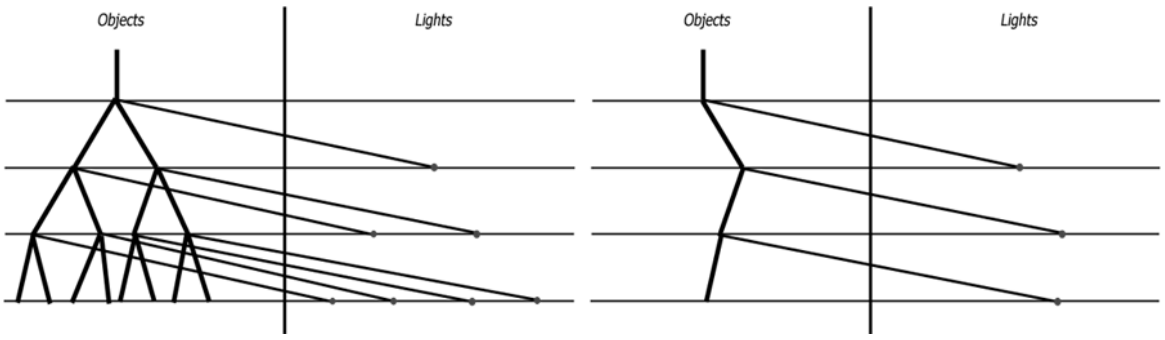
$$I(x,x') = g(x,x')\left[\varepsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')\,dx''\right], \tag{1}$$

kde:

$I(x,x')$	je intenzita světla procházející z bodu x' do x,
$g(x,x')$	je viditelnost mezi x' a x,
$\varepsilon(x,x')$	je intenzita světla vyzářená z bodu x' do x,
$\rho(x,x',x'')$	je intenzita světla rozptýleného z bodu x'' do x částí povrchu v bodu x',
S	je celá scéna.

Renderovací rovnice říká, že intenzita přeneseného světla z bodu x' do x je rovna sumě světla vyzářeného z x' směrem k x a celkovému světlu přicházejícímu z celé scény, které bylo rozptýleno, odraženo nebo vyzářeno směrem k x' a potom od x' k x.

Kromě samotné rovnice je v práci také představena nová metoda nazvaná *path tracing* (sledování cesty), která je numerickým řešením rovnice. Je porovnávána ke klasickému ray tracing algoritmu, který lze lehce převést na path tracing. V podstatě je proveden tradiční ray tracing algoritmus, ale místo větvení stromu paprsků na každém povrchu je následována pouze jedna z větví pro získání cesty ve stromu. Paprsek je vždy vržen směrem ke známému světelnému zdroji, který může být i plošný. Tak schéma ray tracingu proti metodě integrační rovnice vypadá jako na obrázku **2.1**.



Obrázek 2.1: Schéma ray tracingu (vlevo) proti metodě integrační rovnice (vpravo). Podle [8].

V tomto diagramu je zdůrazněn důležitý fenomén. Kvůli pasivitě povrchů je široce známo, že první generace paprsků, stejně jako paprsky zdroje světla, jsou nejvíce důležité z hlediska proměnlivosti, se kterou přispívají k integrálu pixelu. Paprsky druhé a vyšších generací přispívají daleko méně k proměnlivosti. Ale klasický ray tracing vynakládá velké množství práce přesně na tyto paprsky, které přispívají nejméně k proměnlivosti obrázku, vrhá příliš mnoho paprsků vyšších generací. Metoda integrační rovnice není k tomuto náchylná. Protože cesta je strom s poměrem větvení 1, vyskytuje se v ní stejné množství různých paprsků první generace jako paprsků vyšších generací. Toto je velmi důležité pro redukci proměnlivosti pro *motion blur, depth of field* a další efekty distribuovaného ray tracingu.

Diagram na obrázku **2.1** také ukazuje alternativní algoritmus klasického distribuovaného ray tracingu. Místo vrhání větvícího se stromu je vržena jen cesta s paprsky vybíranými na základě pravděpodobnosti. Pro scény s hodně odrazy a refrakcí to nesmírně snižuje počet protnutí paprsku s objektem, které je třeba vypočítat pro daný pixel a významně urychluje ray tracing s malou programátorskou námahou. Nicméně pro tento path tracing je velmi důležité udržet správný poměr odrazových, refrakčních a stínových paprsků přispívajících ke každému pixelu. Místo náhodného výběru typu paprsku jsou dvě alternativy. První je pamatovat si počet vystřelených paprsků jednotlivých typů a druhým přístupem je náhodný výběr typu paprsků, ale s vyvážením příspěvku jednotlivých typů poměrem požadované distribuce k výsledné vážené distribuci vzorků.

2.1.2 Photon mapping

Photon mapping byl vynalezen Henrikem Wann Jensenem [6] pro urychlení ray tracingu. Hlavní myšlenka je rozdělení ray tracingu do dvou průchodů: první vrhá fotony do scény ze světelných zdrojů a druhý sbírá fotony pro vytvoření obrázku. Algoritmus aproximuje řešení renderovací rovnice (1) z [8].

Konsturkce fotonové mapy v prvním průchodu. Ze světelných zdrojů jsou vyslány svazky světla jako fotony. Kdykoliv foton zasáhne povrch, tak je uložen průsečík a příchozí směr do cache zvané fotonová mapa. Typicky jsou vytvořeny dvě fotonové mapy pro scénu: jedna speciálně pro kaustiky a druhá jako globální pro ostatní světlo. Po zasažení povrchu je materiálem dána pravděpodobnost odrazu, absorbování nebo přenosu – refrakce. Když je foton pohlcen, není pro něj vypočítán nový směr a sledování fotonu končí. Když je odražen, je vypočítán poměr odraženého záření. A když je přenášen dále, funkce jeho směru závisí na povaze přenosu.

Renderování v druhém průchodu. V tomto kroku algoritmu je použita fotonová mapa vytvořená v první průchodu k odhadu jasu každého pixelu výstupního obrázku. Pro každý pixel je použit ray tracing celé scény dokud není nalezen průnik s nejbližším povrchem. V tomto bodě je použita renderovací rovnice (1) pro výpočet povrchového záření opouštějící průsečík ve směru

paprsku, který ho zasáhl. Pro větší efektivitu je rovnice rozložena na čtyři části: přímé osvětlení, spekulární odlesky, kaustiky a měkké nepřímé osvětlení.

2.1.3 Radiozita

Popisuje interakci světla mezi difúzně odrážejícími povrchy pro dosažení efektu globálního osvětlení od Gorala et al. [9] a je aplikací metody konečných prvků pro řešení renderovací rovnice (1). Procedura je založena na metodách tepelného inženýrství. Modeluje fyzikální chování viditelného světla při jeho propagaci prostředím. Protože intenzita a distribuce světla ve scéně jsou řízeny přenosem energie a principy zachování energie, je třeba je uvažovat při přesné simulaci rozdílných světelných zdrojů a materiálů ve stejné scéně.

Tato metoda může být použita k výpočtu intenzity světla difúzně odráženého v prostředí. Je založena na energetických principech a může být aplikována monochromaticky nebo na konečný počet intervalů vlnových délek. Klíčový je předpoklad, že všechny povrchy jsou ideální difúzní (Lambertovské) reflektory. Procedura je aplikovatelná na libovolné prostředí složené z takových povrchů a může počítat s přímým osvětlením z několika různých světelných zdrojů a všechny mnohonásobné odrazy v prostředí. Výhodou této metody je, že výsledné intenzity povrchů jsou nezávislé na místě pozorovatele. Tak může být informace o intenzitě prostředí předpočítána pro dynamické sekvence. Protože malé zrcadlové plochy mohou trochu přispívat k celkové světelné energii, mohou být takové odrazy povrchu přidány k řešením s difúzními odrazy s minimální námahou.

Povrchy ve scéně, které jsou renderovány, jsou rozděleny na menší povrchy - *patches*. Je vypočítán *konfigurační faktor* pro každý pár *patches*, což je koeficient, který popisuje jak dobrá je viditelnost mezi *patches*. *Patches*, které jsou daleko od sebe, nebo natočené v šikmých úhlech budou mít menší pohledový faktor. Pokud jsou v cestě další *patches*, pohledový faktor bude redukován na nulu, v závislosti na okluzi, jestli je částečná nebo úplná. Pohledové faktory jsou použity jako koeficienty v lineární soustavě renderovacích rovnic. Řešení soustavy dává jas každého z *patches*, kdy jsou brány v potaz i difúzní mnohonásobné odrazy a měkké stíny.

2.1.4 Antiradiance

Dachsbacher et al. [13] reformulovali renderovací rovnici (1), aby zmírnili nutnost explicitního výpočtu viditelnosti, a tak umožnili interaktivní globální osvětlení na grafickém hardwaru. Dosáhli toho zacházením s viditelností implicitně a propagováním přídavné veličiny „negativního světla“, zvané *antizáření*, pro kompenzaci světla přeneseného zvnějšku. Výpočty viditelnosti jsou přepracovány na jednoduché lokální iterace udržováním přídavné informace o směru antizáření vzorky ve scéně.

Když je ignorována viditelnost, nějaká část světla je přenesena zvnějšku skrz neprůhledné objekty a je třeba ji vykompenzovat. Proto je zavedena nová veličina zvaná antizáření, která koresponduje se světlem, které je třeba odebrat, protože nesprávně prostoupilo povrchem kvůli opomenuté okluzi. Každý bod na povrchu vyzařuje zpět toto negativní světlo. Reformulace rovnice dramaticky zjednodušuje zpracování párové interakce mezi dvěma objekty: propagace záření a antizáření mezi dvěma objekty nezávisí na okluzi, což velmi pomáhá paralelizaci na grafickém hardwaru.

Nevýhodou je nutnost simulovat dvě veličiny, záření a antizáření, a je nutno uvažovat směrovou distribuci antizáření. Nicméně pro nepřímé osvětlení (nebo jakékoliv nízkofrekvenční osvětlení) je viditelnost rychle „zprůměrována“ po několika odrazech světla a směrová diskretizace antizáření nemusí být extrémně přesná. Pro vysokofrekvenční přímé osvětlení je možno použít tradiční techniku jako např. shadow mapy. Navíc zacházení se směrovými veličinami je vhodné pro lesklé materiály, ačkoliv způsobují extra cenu pro lokální stínovací integrál.

2.1.5 Virtual Point Lights

Myšlenkou instantní radiozity od Alexandra Kellera [15] je koncentrace energie svítidel ve vzorcích, bez explicitní diskretizace, bez komplexních radiozitních faktorů a s jednoduchými bodovými světly. Energie se odráží ve scéně a v místech odrazu jsou zanechána *virtuální bodová světla* (z angl. *virtual point lights – VPLs*), cesty světla lze znovu použít.

Prvním krokem algoritmu je sledování fotonů ze zdroje světla do scény. S vrcholy cesty je zacházeno jako s VPLs. Je generována aproximace částic difúzního záření ve scéně – deterministická simulace světla – použitím kvazi náhodného průchodu založeného na metodě integrace kvazi Monte Carlo. Poté je scéna několikrát vyrenderována pro každý světelný zdroj. Grafický hardware renderuje obrázek se stíny pro každou částici, použitými jako bodové zdroje světla. Globální osvětlení je finálně získáno sečtením jednotlivých obrázků v akumulačním bufferu. Algoritmus počítá průměrné záření procházející přes pixel. Operuje přímo na popisu otexturované scény v prostoru obrázku a neaplikuje žádný kernel nebo řešení diskretizace na integrační rovnici (1).

2.1.6 Reflective Shadow Maps

Reflektivní shadow mapy (RSM) od Dachsbachera a Stammingera [19] jsou algoritmem pro interaktivní renderování přijatelného nepřímého osvětlení. RSM je rozšířením standardní shadow mapy, kde každý pixel je považován za zdroj nepřímého světla, který generuje nepřímé osvětlení s jedním odrazem ve scéně. Tato myšlenka je založena na pozorování, že při jednom bodovém zdroji světla je všechno nepřímé osvětlení s jedním odrazem způsobeno povrchy viditelnými v jeho shadow mapě. V takovém případě shadow mapa obsahuje všechny informace o nepřímém světle. Práce je prováděna převážně v prostoru obrazovky, takže je nezávislá na komplexnosti scény.

Všechny povrchy ve scéně jsou brány jako difúzní reflektory. V RSM je pro každý pixel uložena hloubka, světová pozice, normála a odražený zářivý tok viditelného bodu povrchu. Každý pixel je interpretován jako *pixelové světlo*, které osvětluje nepřímo scénu. RSM je generována stejně jako standardní shadow mapa, ale s více renderovacími cíly. Nepřímé ozáření v bodě na povrchu je možno aproximovat sečtením osvětlení všech pitelových světel. Takový výpočet pro všechny pixely výsledného obrázku je ovšem příliš náročné pro interaktivní aplikaci. Nicméně s využitím jednoduchého interpolačního schématu lze drasticky zredukovat počet výpočtů a použít levnou interpolaci pro většinu pixelů.

2.1.7 Light Propagation Volumes

Práce Kaplanyana et al. [23] je založena na vzorkování osvětlení scény do mřížky. To dovoluje modelovat přenos světla použitím jednoduchých lokálních operací, které mohou být jednoduše paralelizovány. Výpočet je založen na intenzitě uložené v každé buňce této mřížky. Výpočet nepřímého osvětlení je složen ze čtyř kroků.

Prvním je inicializace mřížky – *light propagation volume* (*LPV*) s povrchy způsobujícími nepřímé osvětlení a nízkofrekvenční přímé osvětlení (plošné zdroje světla). Inicializace je založena na myšlence, že nízkofrekvenční osvětlení lze převést na sadu VPLs, jako Keller [15]. Nicméně je použito daleko více VPLs, ale jejich příspěvky nejsou počítány jednotlivě, ale jsou pouze využity na inicializaci LPV.

Druhým je vzorkování povrchů scény použitím krokování hloubky kamery a několika reflektivních shadow map. Tato informace je použita pro vytvoření hrubé objemové reprezentace geometrie, která je dále využita na blokování světla během jeho propagace a tedy výpočet nepřímých stínů.

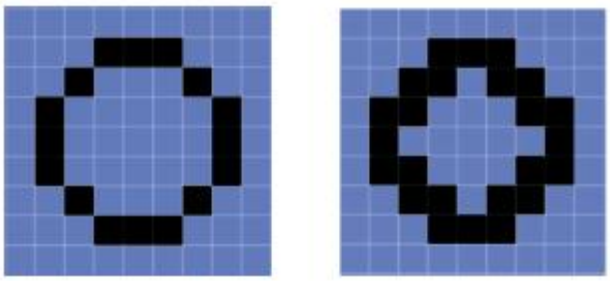
Třetí je propagace světla startující z počátečního LPV, vypočítaná postupnými lokálními iteračními kroky. Každá buňka LPV má uloženu intenzitu a světlo je pak propagováno do 6 sousedů podle směru os. Výsledek po spočítání všech propagací v LPV je akumulován v oddělené 3D mřížce po každé iteraci: součet všech pomocných výsledků je finální distribucí světla ve scéně.

Posledním krokem je osvětlení geometrie ve scéně s využitím propagovaného světla. Intenzita je získána pomocí bilineární interpolace. Pak je vyhodnocena funkce intenzity pro negativní orientaci povrchu, ale protože je ukládána intenzita je potřeba ji převést na záření. Navíc k přímé aplikaci LPV pro difúzní osvětlení lze přidat aproximaci pro nepřímé osvětlení lesklých objektů.

2.2 Voxelizace

Tato podkapitola vychází z článku o voxelizaci od Cyrila Crassina a Simona Greena [1]. Ve velkém množství výpočetních a vědeckých aplikací a zvláště v počítačové grafice se zvyšuje zájem o diskrétní voxelovou reprezentaci dat. Její využití je možné od simulace tekutin, přes detekci kolizí, simulaci přenosu energie zářením, až po detailní renderování a globální iluminaci v reálném čase. Při použití v reálném čase je kritické dosáhnout rychlé *3D scan konverze*, také zvané *voxelizace*, z tradiční reprezentace povrchu založené na trojúhelníkové síti.

Předchozí práce o 3D voxelizaci rozlišují mezi dvěma druhy povrchové voxelizace: *řídká voxelizace*, která je reprezentací *6okolí* povrchu a plnou *konzervativní voxelizací*, neboli *26okolí* (obrázek **2.2**), kde všechny voxely překryté povrchem jsou aktivovány.



Obrázek 2.2: Příklad 4okolí (vlevo) a 8okolí (vpravo) u 2D rasterizace čáry jako ekvivalent k 6okolí a 26okolí u povrchové voxelizace ve 3D. Převzato z [1].

V posledních letech bylo navrženo mnoho algoritmů, které využívají GPU pro voxelizaci trojúhelníkové sítě. Prvotní přístupy používaly řetězce s fixní funkcionalitou, které se vyskytovaly na tehdejším grafickém hardwaru. Předchozí přístupy založené na hardwaru byly relativně neefektivní a trpěly problémy s kvalitou. Kvůli chybějící možnosti náhodného přístupu do paměti v grafických shaderech, musely tyto přístupy použít několika průchodové renderování, zpracovávající objem kousek po kousku a se změnou celé geometrie s každým průchodem. Oproti tomu některé algoritmy zpracovávají několik kousků najednou, enkódováním voxelové mřížky do kompaktní binární reprezentace, díky tomu dosahují vyššího výkonu, ale jsou omezeny pouze na binární voxelizaci (uložení pouze jednoho bitu pro reprezentaci obsazeného voxelu).

Novější přístupy voxelizace využívají výhody volnosti s příchodem *compute módu* (CUDA nebo OpenCL) dostupného na moderních GPU. Namísto použití hardwaru s fixní funkcionalitou, tyto přístupy navrhují čistě datově paralelní algoritmy, které poskytují větší flexibilitu a dovolují nové originální návrhy voxelizace, jako přímá voxelizace do *sparse octree*. Nicméně použitím pouze *compute módu* na GPU znamená, že tyto přístupy nevyužívají výhod silných jednotek grafických karet s fixní funkcionalitou, zvláště pak hardwarového rasterizéru, který efektivně poskytuje velmi rychlou funkci kontroly, zda je bod uvnitř trojúhelníku a vzorkovací operace. Se zvyšujícím se

zaměřením průmyslu na energetickou efektivitu pro mobilní zařízení, efektivní využití fixního hardwaru se stává velmi důležitým. Kombinace výhod obou přístupů, s využitím rychlých jednotek GPU s fixní funkcionalitou a s nutností pouze jedno průchodového zpracování a díky evoluci nejnovějšího grafického hardwaru, dovoluje *sparse voxelizaci*.

2.3 Neomezený přístup do paměti v GLSL

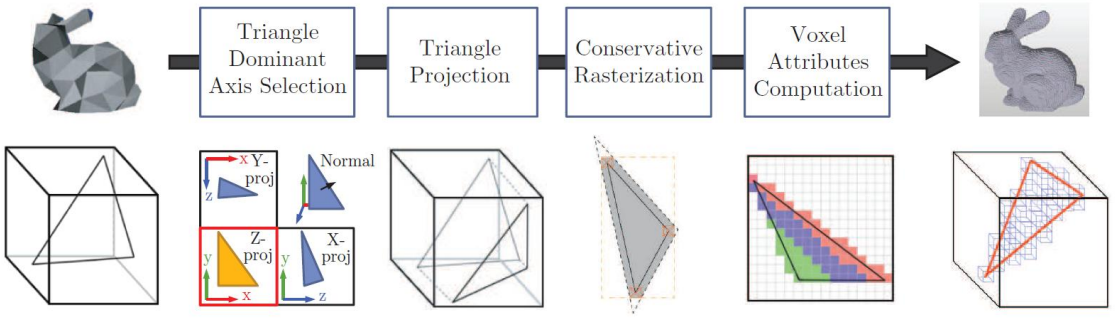
Předchozí přístupy založené na grafických kartách (bez použití *compute módu*) byly omezeny tím, že všechny operace zápisu do paměti musely být provedeny hardwarem pro zpracování fragmentů, který nedovoluje náhodný přístup do paměti a 3D adresování, protože pouze současný pixel mohl být zapsán. V poslední době byl zásadně změněn programovací model nabízený OpenGL shadery, kdy GLSL shadery získaly možnost vytváření vedlejších efektů a dynamického adresování libovolných bufferů a textur. Například OpenGL 4.2 specifikace standardizovala přístup k *image units* v GLSL (rozšíření `ARB_shader_image_load_store`). Tato vlastnost, pouze dostupná na hardwaru se Shader Model 5 (SM5), umožňuje přístup k jednomu levelu mipmapy textury z jakékoliv fáze GLSL shaderu a provádět čtení/zápis, stejně jako atomické načti-změň-zapiš operace. Kromě textur, také lineární regiony paměti (*buffer objekty* uložené v globální paměti GPU) mohou být jednoduše přistupovány díky této vlastnosti s použitím *buffer textures* svázaných s GLSL `imageBuffer`.

Dalším rozšířením je v OpenGL 4.3 `ARB_shader_storage_buffer_object`, které poskytuje podobnou funkcionalitu nad lineárními regiony paměti v GLSL, ale dělá to přes ukazatele podobné jako v jazyku C a také umožňuje získat globální adresu paměti jakéhokoliv buffer objektu. Tento postup zjednodušuje přístup k buffer objektům a dovoluje libovolný počet nesouvislých paměťových regionů (různých buffer objektů), ke kterým je můžno přistupovat ze stejné invokace shaderu, zatímco je v shaderu možné přistupovat pouze k omezenému počtu image units (tento počet je závislý na implementaci).

Tyto nové vlastnosti zásadně změnily výpočetní model v GPU shaderech a dávají možnost psát algoritmy se stejnou flexibilitou jako CUDA nebo OpenCL, ale stále s výhodou využití rychlého fixního hardwaru.

2.4 Jednoduchá voxelizace

V této kapitole bude popsán počáteční jednoduchý přístup pro přímou voxelizaci do pravidelné mřížky voxelů uložených do 3D textury. Voxelizační řetězec je založena na pozorování, že *voxelizace tenkého povrchu trojúhelníku* ***T*** může být vypočtena pro každý voxel ***V*** testováním, zda zaprvé rovina ***T*** protíná ***V*** a zadruhé, zda 2D projekce trojúhelníku ***T*** podle dominantní osy jeho normály (jedna ze tří hlavních os scény, která dává největší povrch promítnutému trojúhelníku) protíná 2D projekci ***V***.



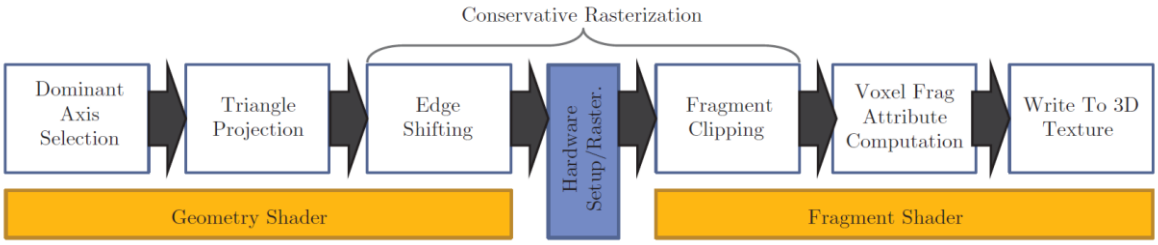
Obrázek 2.3: Ilustrace jednoduchého voxelizačního řetězce. Převzato z [1].

Na základě tohoto pozorování je založen velmi jednoduchý algoritmus voxelizace, který pracuje ve čtyřech hlavních krocích uvnitř jednoho vykreslovacího volání (zobrazeno na obrázku **2.3**). Nejprve je každý trojúhelník sítě ortograficky promítnut podle dominantní osy jeho normály, to je ta ze tří hlavních os scény, která maximalizuje promítnutou plochu a tím maximalizuje počet fragmentů, které budou vygenerovány během konzervativní rasterizace. Tato projekční osa je vybírána dynamicky podle aktuálního trojúhelníku v geometry shaderu (obrázek **2.4**), ve kterém jsou najednou dostupné informace o všech třech vrcholech trojúhelníku. Pro každý trojúhelník je vybrána osa, která dává maximální hodnotu pro $l_{\{x,y,z\}} = |\mathbf{n} \cdot \mathbf{v}_{\{x,y,z\}}|$, kde $\mathbf{n}$ je normála trojúhelníku a $\mathbf{v}_{\{x,y,z\}}$ jsou tři hlavní osy scény. Po výběru osy je projekce podle této osy klasická ortografická projekce, která je vypočítána v geometry shaderu.

Každý promítnutý trojúhelník je předán dál ve standardním rasterizačním řetězci pro vykonání *2D scan konverze* (rasterizace, obrázek **2.4**). Pro získání fragmentů odpovídajících 3D rozlišení cílové (kubické) voxelové mřížky, je nastaveno 2D rozlišení záběru (`glViewport(0, 0, x, y)`) na rozlišení voxelové mřížky (např. 512 x 512 pixelů pro 512^3 voxelovou mřížku). Algoritmus využívá přístup k buffer objektům a image jednotkám k zápisu dat do voxelové mřížky místo standardní cesty přes fragmentové operace do framebufferu, proto jsou všechny operace framebufferu vypnuty včetně zápisů hloubky, testu hloubky a zápisů barvy.

Během rasterizace každý trojúhelník generuje sadu 2D fragmentů. Pro každý 2D fragment je ve fragment shaderu vypočítán skutečně protnutý voxel trojúhelníkem v závislosti na pozici a informací o hloubce získaných z centra pixelu a interpolovaných hodnot vrcholů. Tyto informace jsou použity pro generaci tzv. *voxel fragmentů*. Voxel fragment je 3D generalizace klasického 2D fragmentu a koresponduje s voxelem protnutým daným trojúhelníkem. Každý voxel fragment má 3D celočíselné koordináty v cílové voxelové mřížce, stejně jako další možné atributy.

Atributy voxel fragmentů jsou obvykle barva, normála a jakýkoliv jiný užitečný atribut, který by bylo potřebné uložit pro každý pixel, v závislosti na aplikaci. Jako obvykle mohou být tyto hodnoty buď interpolovány v centrech pixelů z atributů vrcholů rasterizačním procesem, nebo



Obrázek 2.4: Implementace voxelizační řetězce nad GPU rasterizačním řetězcem. Převzato z [1].

vzorkovány z tradiční 2D povrchové textury modelu s využitím interpolovaných texturovacích souřadnic.

Nakonec jsou voxel fragmenty zapsány přímo ve fragment shaderu do jejich korespondujících voxelů uvnitř cílové 3D textury, kde musí být zkombinovány dohromady. K tomu jsou využity image load/store operace, detailně popsané v sekci **2.4.2**.

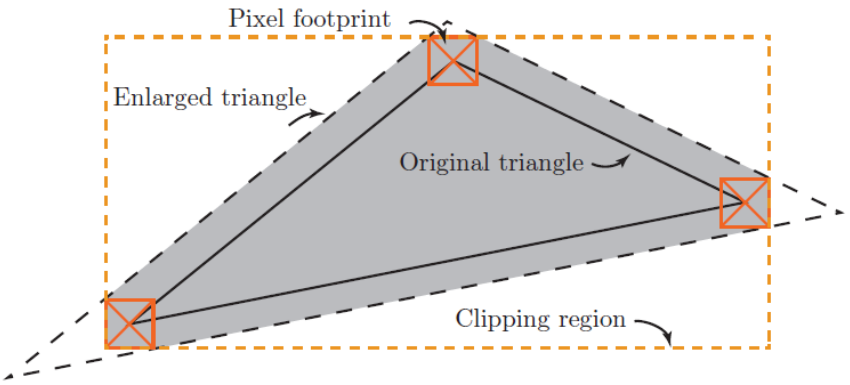
2.4.1 Konzervativní rasterizace

I když je tento přístup velmi jednoduchý, nezaručuje korektní řídkou voxelizaci (6okoolí). To je díky tomu, že jen pokrytí středu každého pixelu je testováno na přítomnost trojúhelníku během rasterizačního kroku pro generaci fragmentů. Takže je potřebná více přesná *konzervativní rasterizace*, aby bylo zajištěno, že bude generován fragment pro každý pixel, ve kterém se vyskytuje trojúhelník. Přesnost testu pokrytí by bylo možné vylepšit použitím *multisample antialiasingu* (MSAA), ale toto řešení pouze odkládá problém o kousek dále a stále míjí fragmenty v případě malých trojúhelníků. Místo toho je použit druhý algoritmus konzervativní rasterizace navržený Hasselgrenem et al. [4].

Hlavní myšlenkou je pro každý promítnutý trojúhelník vygenerovat trochu větší obalující polygon, který zajišťuje, že jakýkoliv promítnutý trojúhelník, který se dotýká pixelu, se nevyhnutelně dotkne středu tohoto pixelu, a tak bude emitován fragment ve fixním rasterizéru. Toho je docíleno tak, že každá hrana trojúhelníku je posunuta směrem ven v geometry shaderu (obrázek **2.4**) za účelem zvětšení trojúhelníku. Protože přesný obalující polygon, který by nenadhodnocoval pokrytí daného trojúhelníku nemá trojúhelníkový tvar (obrázek **2.5**), nadbytečné fragmenty mimo ohraničující obdélník jsou zahozeny ve fragment shaderu po rasterizaci. Tento přístup znamená více práce ve fragment shaderu, ale v praxi je rychlejší, než výpočet a generace úplně přesného obalujícího polygonu v geometry shaderu.

2.4.2 Skládání voxel fragmentů

Po vygenerování voxel fragmentů ve fragment shaderu mohou být jejich hodnoty zapsány přímo do cílové 3D textury použitím image load/store operací. Nicméně několik voxel fragmentů z různých trojúhelníků může spadnout do stejného cílového voxelu v libovolném pořadí. Protože jsou voxel fragmenty vytvářeny a zpracovávány paralelně, jejich pořadí, v jakém budou zapsány, nelze



Obrázek 2.5: Ohraničující polygon trojúhelníku použitý pro konzervativní rasterizaci. Převzato z [1].

předpovědět, což vede k problémům s pořadím zápisu a může způsobit blikání a časově nekonzistentní výsledky při dynamické revoxelizaci scény. Při standardní rasterizaci je tento problém řešen jednotkami pro zpracování fragmentových operací, což zajišťuje, že fragmenty jsou ve framebufferu skládány ve stejném pořadí, v jakém byla zpracována jejich zdrojová primitiva.

V případě voxel fragmentů je třeba využít atomické operace. Atomické operace garantují, že načti-změň-zapiš cyklus není přerušen žádným jiným vláknem. Když několik voxel fragmentů skončí ve stejném voxelu, nejjednodušší vhodné chování je zprůměrování všech vstupních hodnot. Pro specifické aplikace je možné použít více sofistikovanější metody skládání, jako např. kombinování založené na pokrytí pixelů.

Průměrování hodnot s využitím atomických operací. Pro zprůměrování všech hodnot, které spadnou do stejného voxelu je nejjednodušší nejprve sečíst všechny hodnoty použitím operace atomického sčítání a následné vydělení sumy celkovým počtem hodnot v dalším průchodu. K tomu je potřeba čítač pro každý voxel, kdy lze využít *alfa kanál* RGBA hodnoty barvy, která se ukládá pro voxel.

Nicméně atomické operace jsou omezeny pouze na 32bitové znaménkové/bezznaménkové celočíselné typy podle OpenGL specifikace, což jen zřídka odpovídá formátu použitému ve voxelové mřížce. Většinou je potřeba uložit `RGBA8` nebo `RGBA16F/32F` barevné komponenty pro voxel. Proto nemohou být atomické funkce sčítání použity přímo jako takové pro provedení celkového sečtení.

Nad takovými typy lze emulovat atomické sčítání s využitím *porovnej-a-vyměň* `atomicCompSwap()` operace. Myšlenkou je cyklení na každém zápisu, dokud dochází ke konfliktům a hodnota, se kterou je vypočítána suma, byla změněna jiným vláknem. Tento přístup je mnohem pomalejší než by byla nativní funkce `atomicAdd`, ale umožňuje funkčně správné chování.

Druhým problémem je použití `RGBA8` barevného formátu pro voxely. S takovým formátem je dostupných pouze 8 bitů pro barevnou složku, což rychle způsobuje problém přetečení při sčítání hodnot. Proto musí být průměr počítán průběžně pokaždé, když je přidáván voxel fragment do daného voxelu. A tak je počítán *klouzavý průměr* P_{i+1} s použitím vzorce:

$$P_{i+1} = \frac{iP_i + x_{i+1}}{i+1}, \tag{2}$$

kde P_i je průměr z minulého kroku, x_{i+1} je aktuální přidávaná hodnota a i je počet již přičtených hodnot. Tento přístup funguje pouze, pokud všechna data, která je třeba uložit, včetně čítače, mohou být *vyměněna* společně za použití jedné atomické operace.

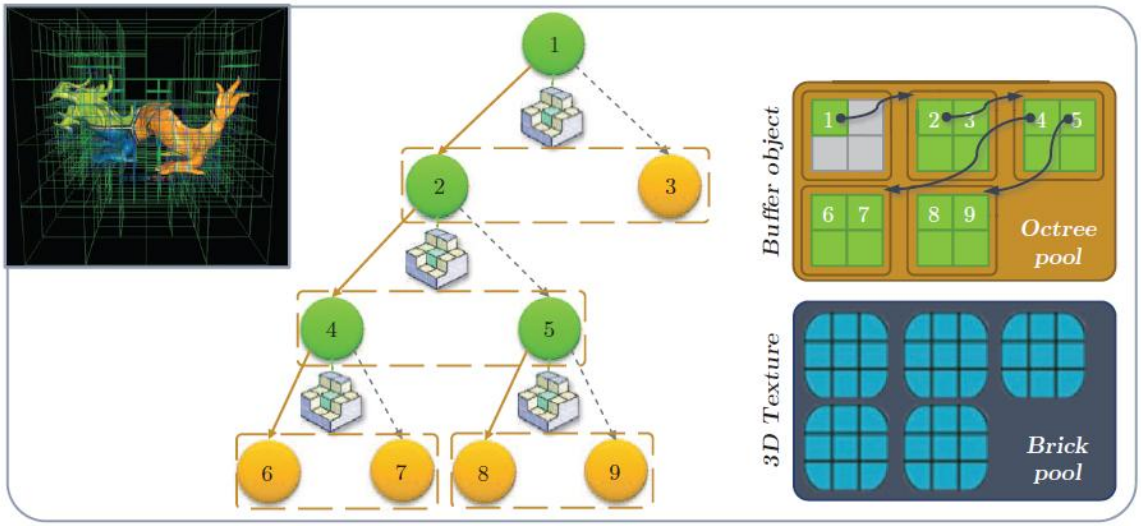
2.5 Voxelizace do Sparse Voxel Octree

Cílem *sparse voxelizace* (řídké voxelizace) je uložení jen voxelů, které jsou protnuty trojúhelníky sítě místo uložení plné mřížky za účelem zvládnutí velkých a komplexních scén a objektů. Pro efektivnost je tato reprezentace uložena ve formě *sparse voxel octree* (*SVO*).

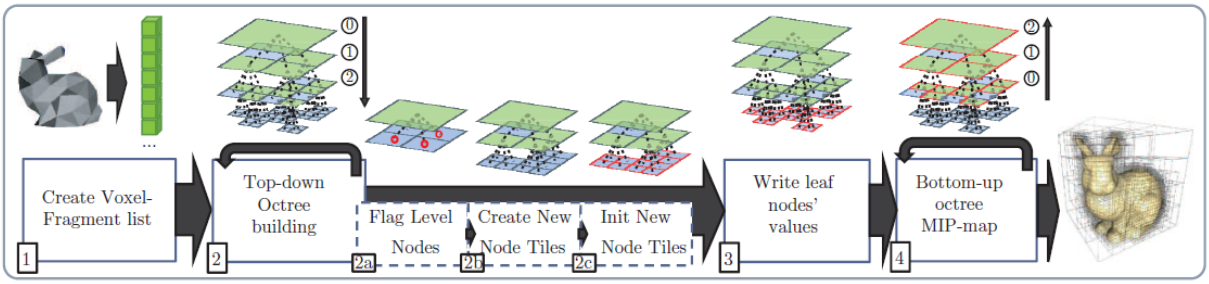
2.5.1 Octree strkuktura

Sparse voxel octree je velmi kompaktní struktura založená na ukazatelích. Její organizace paměti je zobrazena na obrázku **2.6**. Kořenový uzel stromu reprezentuje celou scénu, každý z jeho potomků představuje osminu jeho objemu a tak dále pro každý uzel.

Uzly octree jsou uloženy v lineární video paměti v buffer objektu zvaném *node pool*. V tomto bufferu jsou uzly seskupeny do $2 \times 2 \times 2$ *uzlových tiles*, což dovoluje uložit pouze jeden ukazatel v každém uzlu (je to vlastně *index* bufferu), který ukazuje na osm potomkovských uzlů. Hodnoty



Obrázek 2.6: Ilustrace octree strkutury s "*bricks*" a jejich implementací ve video paměti. Převzato z [1].



Obrázek 2.7: Ilustrace kroků budování octree. Převzato z [1].

voxelů mohou být uloženy přímo v uzlech v lineární paměti nebo mohou být drženy v tzv. *bricks* přiřazených k *uzlovým tiles* a uloženy ve velké 3D textuře – *brick pool*. Tento model uzly plus *bricks* je použit ve [2] a [3] pro rychlé trilineární vzorkování hodnot voxelů.

Tato struktura obsahuje hodnoty pro všechny levely stromu, což dovoluje získání filtrovaných voxelových dat v jakémkoliv rozlišení a s rostoucími detaily zanořováním hlouběji v hierarchii strmou. Tato vlastnost je velmi žádoucí a byla hodně využívána v aplikaci pro globální osvětlení v [2] [3].

2.5.2 Přehled sparse voxelizace

Algoritmus pro vybudování octree struktury je založen na voxelizaci do pravidelné mřížky popsané v sekci **2.4**. Celý algoritmus je zobrazen na obrázku **2.7**. Základní myšlenka tohoto přístupu je velmi jednoduchá.

Struktura je budována shora dolů, jedna úroveň najednou, začíná se z jednovoxelového kořenového uzlu a postupně jsou děleny neprázdné uzly (protnuté aspoň jedním trojúhelníkem) v každé následující úrovni octree s vyšším rozlišením (krok 2 v obrázku **2.7**). Pro každou úroveň jsou detekovány neprázdné uzly voxelizací scény s rozlišením odpovídajícím rozlišení úrovně a je vytvořen nový *tile* 2^3 poduzlů pro každý z nich. Nakonec jsou hodnoty voxel fragmentů zapsány do listových uzlů stromu a podvzorkovány do nadřazených uzlů (krok 3 a 4 v obrázku **2.7**).

2.5.3 Konstrukce seznamu voxel fragmentů

Ve skutečnosti by byla několikanásobná revoxelizace celé trojúhelníkové sítě jednou pro každou úroveň stromu velmi výpočetně nákladná. Místo toho je voxelizace provedena pouze jednou v největším rozlišení, tj. rozlišení nejhlubší úrovně stromu, a vygenerované voxel fragmenty jsou zapsány do *seznamu voxel fragmentů* (*voxel fragment list*) (krok 1 v obrázku **2.7**). Tento seznam je pak použit místo trojúhelníkové sítě pro dělení stromu během budovacího procesu.

Seznam voxel fragmentů je lineární vektor položek uložených v předalokovaném buffer objektu. Je tvořen několika poli hodnot, jedno obsahuje 3D koordináty každého voxel fragmentu a ostatní obsahují všechny atributy, které je potřeba uložit (např. barvu, normálu).

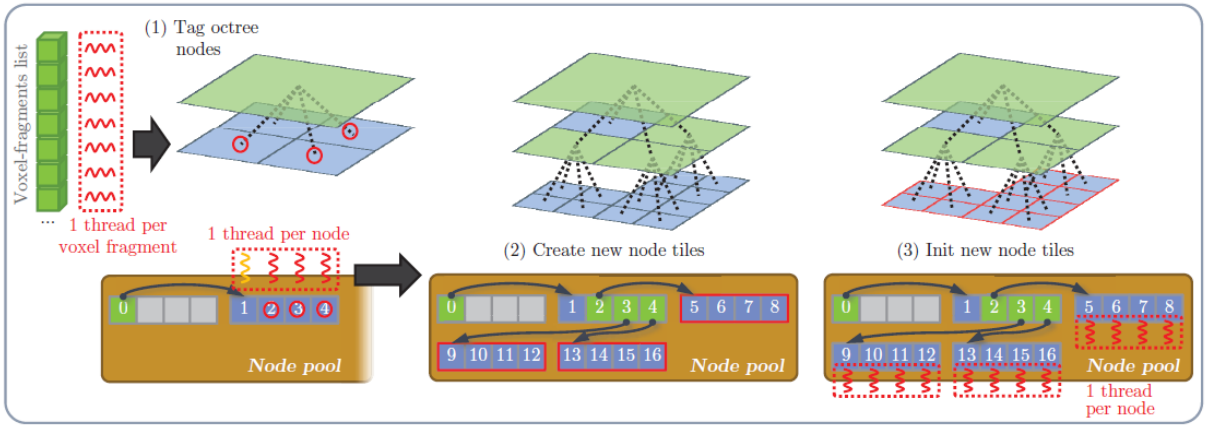
Pro naplnění seznamu voxel fragmentů je trojúhelníková scéna voxelizována podobně jako v sekci **2.4**. Rozdíl je v tom, že místo přímého zápisu voxel fragmentů do cílové 3D textury, jsou přidávány na konec seznamu voxel fragmentů. Pro řízení seznamu je uložen index další volné položky (který je také čítač počtu voxel fragmentů v seznamu) jako jedna 32bitová hodnota uložená v jiném buffer objektu.

K tomuto indexu je třeba přistupovat konkurenčně tisíci vlákny, která přidávají hodnoty voxelů, takže je implementován atomickým čítačem (představeným v OpenGL 4.2). Atomické čítače poskytují vysoce optimalizované atomické inkrementuj/dekrementuj operace pro 32bitové celočíselné proměnné. Na rozdíl od všeobecných `atomicInc` a `atomicAdd` operacím, které dovolují dynamické indexování, atomické čítače jsou navrženy pro vysoký výkon, když všechna vlákna operují nad stejným statickým paměťovým regionem.

2.5.4 Dělení uzlů

Samotné dělení všech uzlů dané úrovně stromu je provedeno ve třech krocích, jak je zobrazeno na obrázku **2.8**. Nejprve jsou uzly, které je třeba rozdělit, označeny spuštěním jednoho vlákna pro každou položku ze seznamu voxel fragmentů. Každé vlákno prochází strom shora dolů až do současné úrovně (kde nejsou žádné ukazatele na potomky) a *označí* (*flag*) uzel, kde vlákno skončilo. Několik vláken může skončit označováním stejného uzlu, tím je možno posbírat všechny požadavky na dělení daného uzlu. Toto *označení* je implementováno jednoduše nastavením nejvíce významného bitu potomkovského ukazatele uzlu.

Kdykoliv je uzel označen k dělení, je potřeba alokovat sadu $2 \times 2 \times 2$ poduzlů (*tile*) ve stromu a přiřadit je k uzlu. Tedy ve druhém kroku je provedena samotná alokace těchto poduzlů spuštěním jednoho vlákna na uzel z aktuální úrovně stromu. Každé vlákno nejprve kontroluje *označení* v přiřazeném uzlu, a pokud je označen, nový uzlový *tile* je alokován a jeho index je přiřazen do potomkovského ukazatele současného uzlu. Tato alokace nových uzlů v *octree pool* je provedena



Obrázek 2.8: Ilustrace tří kroků provedených pro každý level stromu během konstrukce shora dolů s plánováním vláken. Převzato z [1].

použitím sdíleného atomického čítače, podobně, jako u seznamu voxel fragmentů.

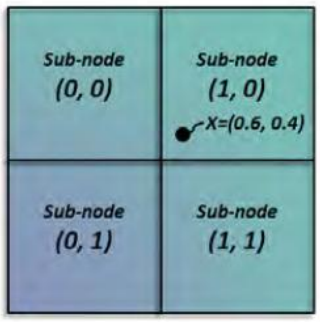
Nakonec je třeba inicializovat nové uzly na *null* ukazatele na potomkovské uzly. Toto je provedeno v dalším průchodu tak, aby jedno vlákno mohlo být přiřazeno s každým uzlem nové úrovně stromu (obrázek **2.8**, krok 3).

2.5.5 Pohyb ve stromu

Zanořování do stromu je použito při vytváření stromu i při renderování a vrhání paprsků. Toto zanořování je rychlé, protože souřadnice bodu v textuře mohou být přímo použity pro získání každého následujícího poduzlu, který je třeba následovat pro získání uzlu, který obsahuje požadovaný bod [2]. Toto schéma je zobrazeno na obrázku **2.9**.

Jak je uvedeno v kapitole **2.5.1**, *uzlovým tilem* jsou nazývány $2 \times 2 \times 2$ potomci daného uzlu, kteří jsou uloženi za sebou v *node poolu*. Uzlový *tile* má unikátní ukazatel s adresou uloženou v rodičovském uzlu. Tak jsou 3D posunutí v uzlovém *tile* a jeho adresa dostatečné informace k vybrání potomka.

Nechť $\mathbf{x} \in \langle 0,1 \rangle^3$ jsou lokální souřadnice bodu v obalovém kvádru uzlu a *c* je ukazatel na jeho potomky (uzlový *tile*). 3D posunutí v *tile* potomka, který obsahuje **x**, tak lze jednoduše získat jako $\mathbf{off} = int(2\mathbf{x})$, kde int je celá část $2\mathbf{x}$ pro každou osu. Např. pro 1D případ pro jednu osu $x_{axis} \in \langle 0,1 \rangle$ jsou dvě možné hodnoty posunutí pro potomka a to 0 ($x_{axis} < 0{,}5$) a 1 ($x_{axis} \geq 0{,}5$). Protože *node pool*, ve kterém jsou uloženy uzly, se nachází v lineární paměti, musí být 3D posunutí **off** přepočítáno na lineární posunutí $off = \mathbf{off}_x + 2\mathbf{off}_y + 4\mathbf{off}_z$. Pro zanoření do poduzlu obsahujícího bod *x* je možno použít ukazatel $c' = c + off$. Pak jsou souřadnice **x** aktualizovány jako $2\mathbf{x} - \mathbf{off}$ a zanoření může pokračovat. Toto zanořování může být zobecněno pro N^3 stromy použitím $\mathbf{off} = int(N\mathbf{x})$ a aktualizováním **x** jako $N\mathbf{x} - \mathbf{off}$.



$$\begin{aligned} Index_{2D} &= int(x * N) \\ &= int((0.6 * 2, 0.4 * 2)) \\ &= int((1.2, 0.8)) \\ &= (1,0) \end{aligned}$$

Obrázek 2.9: 2D lokalizace v uzlovém *tile* pro nalezení indexu poduzlu kde leží bod *x*. Převzato z [2].

2.5.6 Zápis a podvzorkování hodnot

Po vybudování octree struktury zbývá její naplnění hodnotami z voxel fragmentů. Tak jsou zapsány hodnoty voxel fragmentů v největším rozlišení do listových uzlů stromu. To je provedeno spuštěním jednoho vlákna na položku ze seznamu voxel fragmentů. Každé vlákno používá podobné schéma, jako vlákna pro pravidelnou mřížku pro kombinaci voxel fragmentů do listových uzlů (sekce **2.4.2**).

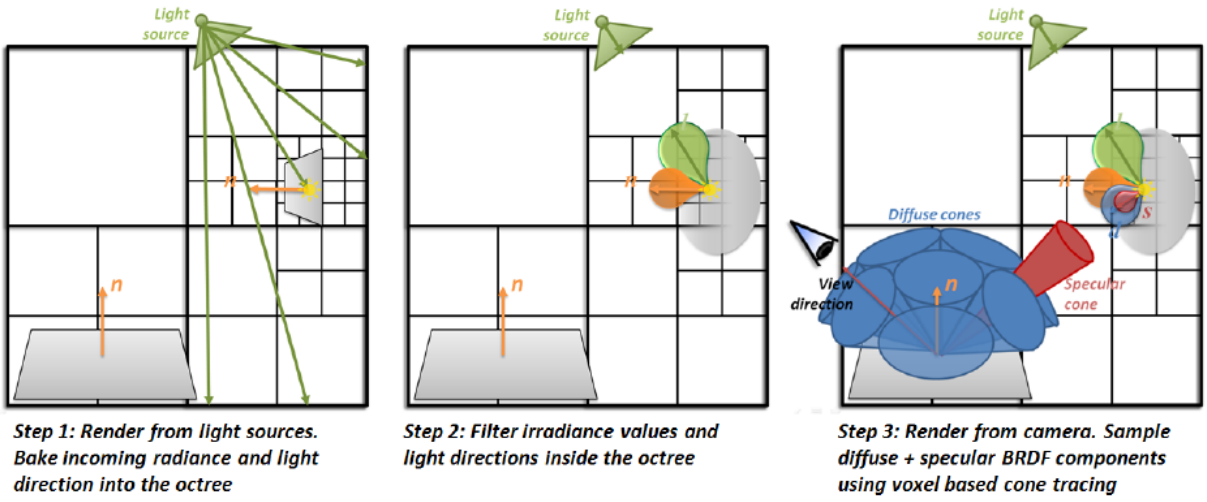
Ve druhém kroku jsou podvzorkovány tyto hodnoty do nadřazených uzlů stromu. Toto je provedeno úroveň po úrovni zdola nahoru v $n - 1$ krocích pro strom s n úrovněmi. V každém kroku je použito jedno vlákno pro zprůměrování hodnot obsažených v osmi poduzlech každého neprázdného uzlu v současné úrovni. Protože byl strom vybudován úroveň po úrovni (sekce **2.5.2**), uzlové *tiles* jsou automaticky roztřízené po úrovních v *octree pool*. Tak je jednoduché spustit vlákna pro všechny uzly alokované v dané úrovni pro provedení průměrování. Tyto dva kroky jsou zobrazeny na obrázku **2.7** (krok 3 a 4).

2.6 Interaktivní nepřímé osvětlení

Nepřímé osvětlení je důležitým elementem syntézy realistických obrazů, ale jeho výpočet je náročný a hodně závislý na složitosti scény a použitého BRDF (z angl. *Bidirectional Reflectance Distribution Function*, což je označení pro obousměrnou distribuční funkci odrazu světla, která slouží k matematickému popisu odrazivých vlastností povrchu v určitém bodě). Zatímco předpočítávání může být použitelné v některých případech, hodně aplikací (hry, simulace atd.) vyžadují přístupy pracující v reálném čase nebo interaktivní přístupy pro využití nepřímého osvětlení.

2.6.1 Souhrn algoritmu nepřímého osvětlení

Přístup je založen na tříkrokovém algoritmu – obrázek **2.10**. Nejprve je příchozí záření (energie a směr) z dynamických zdrojů světla uloženo do listů hierarchického *sparse voxel octree*. To je provedeno rasterizací scény ze všech světelných zdrojů a uložením fotonu pro každý viditelný fragment povrchu. Ve druhém kroku je provedena filtrace příchozích hodnot záření do vyšších úrovní stromu (podvzorkování). Je využito kompaktní *Gaussian-Lobe* reprezentace pro uložení vyfiltrované distribuce směrů příchozího světla. Toto je efektivně prováděno paralelně s využitím quadtree analýzy obrazového prostoru. Filtrovací schéma také zachází s NDF (normálovou distribuční funkcí) a BRDF v závislosti na pohledu. Nakonec je scéna vyrenderována z pohledu kamery. Pro každý viditelný fragment povrchu je zkombinováno přímé a nepřímé osvětlení. Přibližné sledování kuželového paprsku je použito pro vykonání finálního shromáždění světelné informace, vysláním několika kuželů přes polokouli k posbírání osvětlení rozptýleného ve stromu. Typicky pro BRDF podobné Phongově osvětlovacímu modelu několik velkých kuželů (např. 5) odpovídá difuzní energii přicházející ze scény, zatímco tenký kužel ve směru odrazu vzhledem k bodu pohledu zachycuje



Obrázek 2.10: Ilustrace tří kroků algoritmu nepřímého osvětlení. *Vpravo:* Znázornění sparse voxel octree struktury ukládající geometrii a informaci o přímém osvětlení. Převzato z [2].

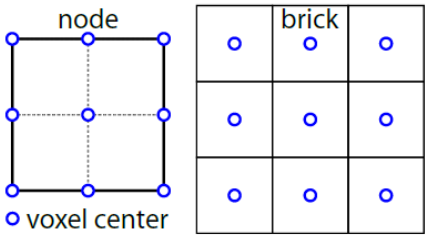
spekulární složku. Šířka spekulárního kužele je odvozena od spekulárního exponentu materiálu, což umožňuje efektivní výpočet lesklých odrazů.

2.6.2 Hierarchická voxelová struktura

Jádro tohoto přístupu je postaveno na přefiltrované hierarchické voxelové verzi geometrie scény. Pro efektivnost je tato reprezentace uložena ve formě sparse voxel octree popsané v sekci **2.5.** Použitím hierarchické struktury je možné vyhnutí se použití skutečné geometrické síti scény a lze dosáhnout nepřímého osvětlení v libovolných scénách se skoro geometricky nezávislou cenou. Je možné zvýšit přesnost blízko pozorovatele a abstrahovat energii a informaci o obsazení voxelů z dálky. Rozlišení scény lze vybrat tak, aby odpovídalo pozorovací a světelné konfiguraci bez ztráty informací, jako by se stalo u podvzorkování světla nebo geometrického LOD (z angl. *Level of Detail* – různé úrovně deailů). Tento přístup zajišťuje vždy vyhlazené výsledky, oproti path tracingu nebo photo mappingu (podle [2]).

2.6.3 Popis struktury

Sparse voxel octree je velmi kompaktní struktura založená na ukazatelích s přiřazenými *bricks* popsaná v sekci **2.5.1**. Uzly stromu jsou uloženy v lineární paměti GPU a jsou seskupeny do 2 × 2 × 2 *uzlových tiles*. Použitím *brick* místo jedné hodnoty na uzel lze použít hardwarovou texturovou trilineární interpolaci pro interpolaci hodnot. Tato struktura umožňuje použít vyfiltrované informace o scéně (intenzitu energie a její směr, pohlcování, lokální NDF) se zvětšující se přesností procházením hierarchie stromu. Tato vlastnost dovoluje dosáhnout adaptivity a zvládnutí velkých a komplexních scén.



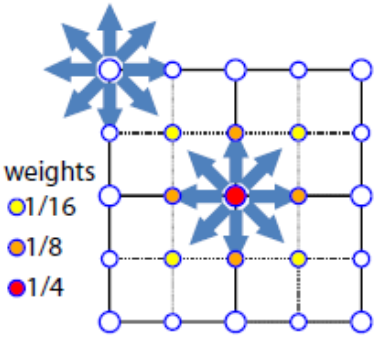
Obrázek 2.11: Je předpokládáno, že středy voxelů jsou umístěny v rozích uzlů a ne ve středech uzlů. Převzato z [2].

Jak je znázorněno v [2], použití malých *bricks* ve sparse octree je více efektivní jak ve využití úložného místa, tak i v rychlosti renderování. Proto je využívána rohově centrovaná voxelová lokalizační konfigurace s $3 \times 3 \times 3$ voxelovými *bricks*. Je předpokládáno, že středy voxelů jsou umístěny v rozích uzlů a ne ve středech uzlů (obrázek **2.11**). Toto zajišťuje, že interpolované hodnoty mohou být vždy vypočítány uvnitř *brick* pokrývající množinu $2 \times 2 \times 2$ uzlů.

Jediným rozdílem oproti struktuře popsané v sekci **2.5.1** jsou přidané sousedské ukazatele na uzly. Ty umožňují rychlé přesuny mezi prostorově sousedícími uzly během interaktivní voxelizace dynamických objektů. Tyto odkazy jsou zvláště důležité pro efektivní distribuci přímého osvětlení přes všechny úrovně stromu.

2.6.4 MIP mapování

Po vybudování octree struktury a uložení hodnot povrchu do listových uzlů, je nutno MIP mapovat a filtrovat hodnoty do vyšších uzlů stromu, jak je popsáno v kapitole **2.5.6**. Pro výpočet nového filtrovaného levelu stromu jsou algoritmem průměrovány hodnoty předchozích úrovní. Protože jsou používány rohově centrované voxely, jak je popsáno v sekci **2.6.3**, každý uzel obsahuje 3^3 *brick*, jejíž okraje jsou obsaženy v sousedních *bricks*. Proto při počítání filtrovaných dat je třeba vyvážit každý voxel převrácenou hodnotou jeho mnohonásobnosti, jak je zobrazeno pro 2D případ na obrázku **2.12.** To ve výsledku znamená 3^3 Gaussovské váhovací jádro.



Obrázek 2.12: Voxely z vyšších úrovní obklopují voxely z nižších úrovní. Během filtrování jsou sdílené voxely rovnoměrně rozděleny. Výsledkem jsou Gaussovské váhy. Převzato z [2].

2.6.5 Voxelová reprezentace

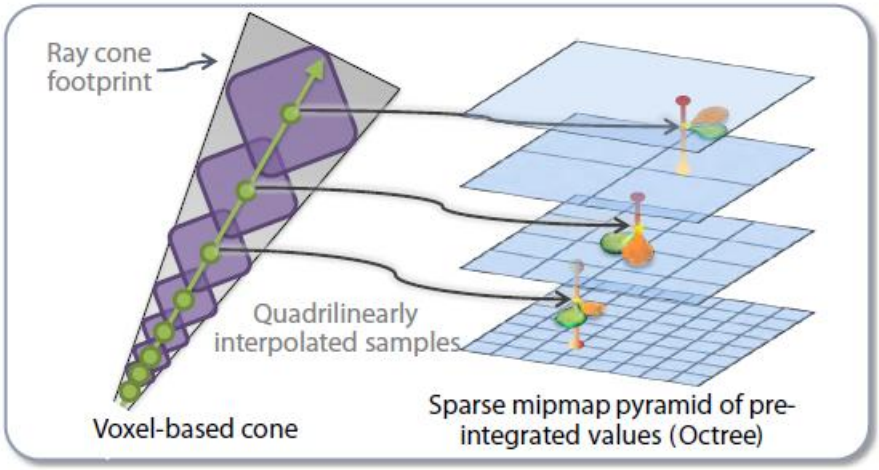
Každý voxel daného LOD musí reprezentovat chování světla nižších úrovní stromu, a tedy jakou část celé scény reprezentuje. Proto je využíván předfiltrovaný geometrický model pro modelování směrové informace s distribucemi, které popisují výchozí geometrii. K tomuto modelu je přidáno vyfiltrované příchozí záření, které bude vloženo do reprezentace všemi zdroji přímého světla. Také jsou uloženy a filtrovány distribuce všech příchozích směrů světla, aby bylo možno počítat nepřímé odlesky.

Protože ukládání různých distribucí by bylo příliš paměťově náročné, jsou ukládány pouze izotropní *Gaussian lobes* charakterizované průměrným vektorem D a standardní odchylkou σ. Pro ulehčení interpolace je odchylka zakódována pomocí normy $|D|$ jako $\sigma^2 = \frac{1-|D|}{|D|}$. V sekci **2.6.8** je popsán výpočet světelné interakce s takovou datovou reprezentací.

2.6.6 Kuželové paprsky

Globální osvětlení obecně vyžaduje, aby bylo vrženo mnoho vzorkovacích paprsků do scény, což je nákladné. Tyto paprsky jsou prostorově a směrově koherentní, což je vlastnost, kterou využívá mnoho přístupů jako *packet ray tracing*. Podobně je založena metoda *voxel cone tracing*, také využívající koherenci paprsků [3].

Zatímco původní sledování kuželového paprsku je komplexní a často nákladná, Crassin využívá filtrovanou voxelovou strukturu pro aproximaci výsledku pro všechny paprsky ze stejného svazku paralelně. Hlavní myšlenkou je postup po ose paprsku a provedení vyhledání v hierarchické MIP mapové pyramidové reprezentaci (popsané v sekci **2.6.4**) v úrovni odpovídající průměru kužele (obrázek **2.13**). Během tohoto kroku je použita interpolace pro zajištění hladkého průběhu bez



Obrázek 2.13: Kuželové sledování paprsku založené na voxelech s použitím předfiltrované geometrie a světelné infomace z řídké MIP mapové pyramidy (uložené ve voxelové stromové struktuře). Převzato z [2].

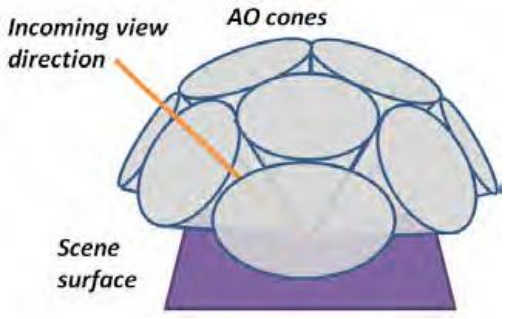
aliasingu. Podle toho, jak byla filtrována voxelová data, tato metoda aproximace produkuje přijatelné výsledky, ale může se lišit od opravdového sledování kuželového paprsku.

2.6.7 Ambient occlusion

Na vysvětlení použití přibližného sledování kuželového paprsku a ulehčení pochopení algoritmu nepřímého osvětlení je nejprve uveden jednodušší případ: *ambient occlusion* (AO). Ambient oclussion $A(p)$ v bodě p na povrchu je definována jako integrál viditelnosti přes polokouli Ω (nad povrchem) vzhledem k promítnutému úhlu. Přesněji $A(p) = \frac{1}{\pi}\int_{\Omega} V(p,\omega)(\cos\omega)d\omega$, kde $V(p,\omega)$ je funkce viditelnosti, která je rovna 0, když paprsek vycházející z bodu *s* se směrem *ω* protíná scénu, jinak je rovna 1. Pro praktické použití (typicky scény uvnitř bez otevřené oblohy) je viditelnost omezena vzdáleností, protože stěny prostředí hrají roli ambientních difuzorů. Tak je okluze α vážena funkcí $f(r)$, která slábne s rostoucí vzdáleností (je použito $\frac{1}{1+\lambda r}$). Upravená okluze je $\alpha_f(p + r\omega) \coloneqq f(r)\alpha(p + \overrightarrow{r\omega})$.

Pro efektivní výpočet integrálu $A(p)$ lze polokouli rozdělit do součtu integrálů $A(p) = \frac{1}{N}\sum_{i=1}^{N} Vc(p,\Omega_i)$, kde $Vc(p,\Omega_i) = \int_{\Omega_i} V_{p,\theta}(\cos\theta)d\theta$. Pro pravidelné rozdělení představuje každé $Vc(p,\Omega_i)$ kužel. Když je zanedbán kosinus z Vc integrálu (aproximace je nepřesná jen pro velké nebo ploché kužely), lze aproximovat jejich příspěvky sledováním kuželových paprsků založeném na voxelech, jak je zobrazeno na obrázku **2.14.** Vážený integrál viditelnosti $V(p,\omega)$ je získán akumulací pouze informací o okluzi s ohledem na váhy $f(r)$. Sečtení příspěvky všech kuželů dává aproximaci AO.

Finální renderování. Pro renderování sítě scény s AO efekty, je vyhodnoceno sledování kuželového paprsku ve fragment shaderu. Kvůli efektivnosti je použito *odložené stínování*, aby nebyly vyhodnocovány výpočty pro skrytou geometrii. Jsou tedy vyrenderovány světová pozice a povrchová normála pro pixely obrazu z aktuálního místa pohledu. Výpočet AO je pak proveden pro každý pixel s použitím výchozí normály a pozice.



Obrázek 2.14: Ambient occlusion je vypočítána pomocí množiny kuželů vypuštěných přes polokouli odpovídající danému povrchu. Převzato z [2].

2.6.8 Stínování voxelů

Pro nepřímé osvětlení nebude důležitá pouze okluze, ale také bude nutné vypočítat stínování voxelu. Pro toto stínování je využita předfiltrovaná reprezentace scény, ke které je třeba přidat informaci o příchozím záření. Toto příchozí záření bude vloženo do struktury ze světelných zdrojů a posbíráno během renderování s využitím přibližného sledování kuželových paprsků.

Výpočty stínování mohou být pohodlně převedeny na konvoluce pod podmínkou, že elementy jsou rozloženy do *lobe* tvarů. V tomto případě jsou konvoluovány BRDF, NDF, rozsah pohledového kužele, stejně jako směry příchozího světla, všechny kromě BRDF jsou již reprezentovány jako *Gaussian lobes*. Je uvažováno Phongovo BRDF, tedy velký difusní lobe a spekulární lobe, které mohou být vyjádřeny jako *Gaussian lobes*. Osvětlovací schéma tak lze snadno rozšířit na jakékoliv BRDF složené z *lobes*.

2.6.9 Nepřímé osvětlení

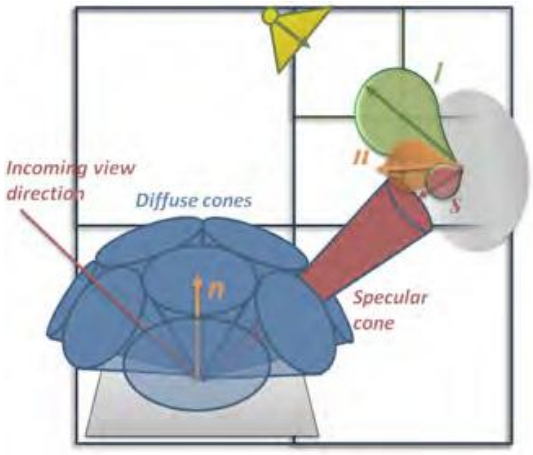
Pro výpočet nepřímého osvětlení v přítomnosti bodových světel je více složité než AO. Je použit dvou krokový přístup. Nejprve je zachyceno příchozí záření ze zdroje světla do listů stromu reprezentujícího scénu. Ukládání příchozího záření místo odchozího dovoluje simulaci lesklých povrchů. Příchozí záření je filtrováno a distribuováno do všech úrovní stromu. Nakonec je vykonáno přibližné sledování kuželového paprsku pro simulaci přenosu světla.

2.6.10 Nepřímé osvětlení se dvěma odrazy

Toto řešení funguje pro komponenty s nízkou energií – nízkou frekvencí a vysokou energií – vysokou frekvencí libovolného materiálu z BRDF, ale popis bude zaměřen na Phongovo BRDF. Algoritmus je podobný tomu pro AO popsanému v sekci **2.6.7**. Je použito odložené stínování pro zjištění, pro které body povrchu je potřeba vypočítat nepřímé osvětlení. Pro každou takovou pozici je vykonáno finální shromáždění informací vysláním několika kuželů pro zjištění osvětlení, které je rozloženo ve stromu. Typicky pro Phongův materiál (obrázek **2.15**) odpovídá několik velkých kuželů (typicky pět) difuzní energii přicházející ze scény, zatímco tenký kužel ve směru odrazu vzhledem k bodu pohledu zachycuje spekulární složku. Šířka spekulárního kužele je odvozena od spekulárního exponentu materiálu, což umožňuje výpočet lesklých odrazů.

2.6.11 Zachycení přímého osvětlení

Tento přístup je inspirován *Reflective shadow maps*. Scéna je vyrenderována z pohledu světla pomocí standardní rasterizace, ale výstupem je světová pozice. V podstatě každý pixel představuje foton, který má být odrážen ve scéně. Tato mapa je zvaná *light-view map*. Následně jsou fotony uloženy ve stromové reprezentaci. Přesněji jsou uloženy jako distribuce směrů a energie úměrné úhlu



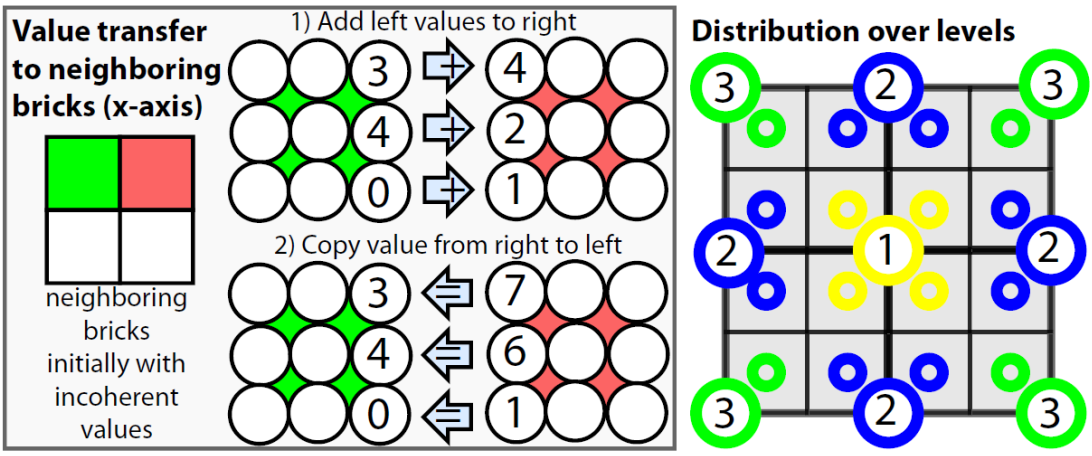
Obrázek 2.15: Nepřímé osvětlení je určeno z množiny kuželů, stínování je vypočítáno z distribučního modelu. Převzato z [2].

protilehlému k pixelu viděnému z pozice světla. Protože rozlišení light-view mapy je obvykle větší, než nejnižší úrovně voxelové mřížky, lze předpokládat, že foton může být uložen přímo od listových uzlů stromu bez způsobení děr. Pro smíchání fotonů je použit fragment shader s jedním vláknem na light-view map pixel. Protože několik fotonů může skončit ve stejném voxelu, je třeba použít atomické sčítání.

Přestože tento proces zní jednoduše, je složitější, než by se mohlo zdát. Problémem je, že voxely se opakují pro sousedící *bricks*. Tato redundance je nezbytná pro rychlé podvzorkování s podporou hardwaru. Zatímco kolize vláken jsou vzácné pro prvotní ukládání fotonů, kopírování fotonů přímo do všech potřebných pozic v sousedních *bricks* vede k mnoha kolizím, které významně ovlivňují výkon. Toto paralelní náhodné šíření dále vede k problémům s přenosovou rychlostí paměti. Je potřebné více efektivní přenosové schéma.

Přenos hodnot do sousedních *bricks*. Pro zjednodušení vysvětlování, předpokládejme, že strom je vytvořen, takže je možné spustit jedno vlákno na listový uzel. Je provedeno šest průchodů, dva pro každou osu (*x, y, z*). Při prvním průchodu pro osu *x* (obrázek **2.16**, pouze levá část), každé vlákno přičte data ze současného uzlu do odpovídajícího voxelu z *brick* na *pravé* straně. V praxi to znamená, že tři hodnoty na vlákno jsou přičteny. Další průchod pro osu *x* přesune data zprava (kde je teď suma) doleva zkopírováním hodnot. Po tomto kroku jsou hodnoty podle osy *x* koherentní a správně rozšířené. Opakováním stejného procesu pro osy *y* a *z* zajišťuje, že všechny voxely jsou správně aktualizovány. Tento přístup je velmi efektivní, protože sousední ukazatele dovolují rychlý přístup k sousedním uzlům a je tak zabráněno kolizím vláken. Dokonce nejsou zapotřebí ani atomické operace.

Distribuce přes úrovně. V tomto bodě jsou koherentní informace na nejnižší úrovni stromu a dalším krokem je filtrace hodnot a uložení výsledků do vyšších úrovní. Jednoduché řešení by bylo spuštění



Obrázek 2.16: *Vlevo:* Během zpracování fotonů je každý foton uložen jen v jednom voxelu, proto dochází k nekonzistenci pro duplikované voxely sousedních uzlů. Přičtení a zkopírování podle každé osy odstraňuje tento problém. *Vpravo:* Pro filtrování hodnot z nižší úrovně do vyšší jsou provedeny tři průchody (označené čísly). Vlákna sčítají voxely nižší úrovně (všechny kolem zvýrazněných oktantů) a ukládají je do vyšší úrovně. Převzato z [2].

jednoho vlákna na každý voxel vyšší úrovně a vybrání hodnot z nižší úrovně. Nicméně toto má důležitou nevýhodu: pro sdílené voxely jsou provedeny stejné výpočty několikrát (až osmkrát). Také výpočetní cena vláken se liší v závislosti na zpracovávaných voxelech, což vede k nevyrovnanému plánování vláken.

Řešením je provedení tří oddělených průchodů, ve kterých mají všechna vlákna přibližně stejnou cenu (obrázek **2.16**, pouze pravá část). Myšlenkou je pouze částečně spočítat filtrované výsledky a použít dříve představený přenos mezi *bricks* pro dopočítání výsledku.

První průchod počítá prostřední voxel z 27 příslušných voxelových hodnot z nižší úrovně (zvýrazněné žluté oktanty na obrázku **2.16**). Druhý průchod počítá *půlku* filtrovaného výsledku pro voxely umístěné uprostřed uzlových ploch (modré). Protože je počítána pouze půlka hodnot, je použito pouze 18 voxelových hodnot. Nakonec třetí průchod spouští vlákna pro rohové voxely (zelené), která počítají částečné filtrování voxelů z jediného oktantu.

Po těchto průchodech jsou voxely z vyšší úrovně v podobné situaci, jako byly listy stromu po úvodním ukládání fotonů: vrcholy stromu mohou obsahovat jen části výsledku, ale sečtením hodnot z *bricks* dává požadovaný výsledek. To tedy stačí k použití dříve představeného přenosu pro dokončení filtrování.

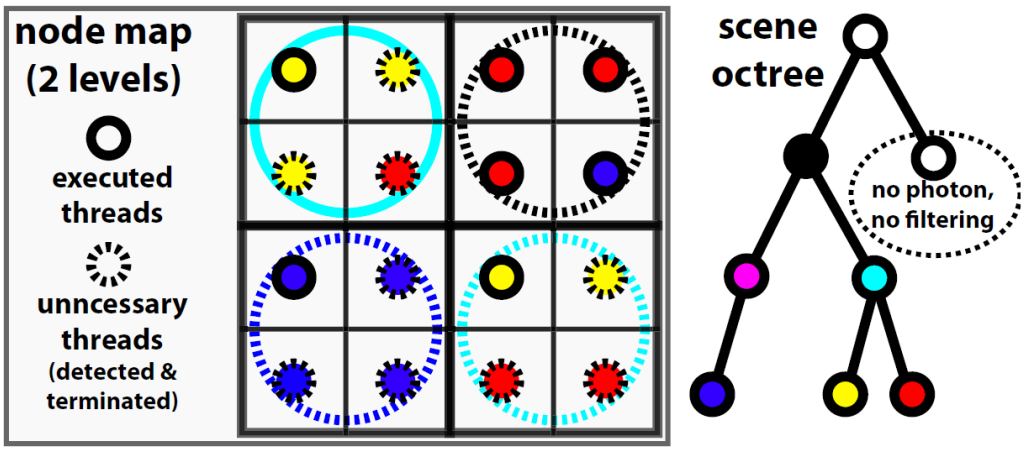
Sparse octree. Zatím bylo předpokládáno, že strom je plný, ale ve skutečnosti je řídký. Ještě k tomu dříve popsané řešení spotřebovává mnoho zdrojů. Během filtrování byla spouštěna vlákna pro všechny uzly, i pro ty, které neobsahovaly žádný foton. Proto následné filtrování bylo často prováděno nad nulovými hodnotami. Toto je velmi kritické, protože přímé světlo často ovlivňuje

pouze část scény. Zde bude představena lepší kontrola volání vláken a práce s nekompletním/řídkým stromem.

Pro vyhnutí se filtrování nulových hodnot by bylo možné spustit jedno vlákno na light-view map pixel, najít odpovídající listový uzel, a pak vystoupat vzhůru ve struktuře stromu na úroveň, ve které by mělo být filtrování provedeno a spustit ho. Výsledek by byl správný, protože filtrování je navrženo tak, aby nezavádělo konflikty čtení-zápis (jedno vlákno jednoduše přepisuje výsledek druhého). Avšak kdykoliv vlákna skončí ve stejném uzlu, práce je vykonána několikrát. Je žádoucí zmenšení tohoto přetížení, ale detekce optimálního počtu vláken je velmi nákladné, proto je použita aproximace.

Myšlenkou je využití 2D *uzlové mapy* odvozené z light-view mapy. Ta připomíná mipmapu a redukuje svoje rozlišení s každou úrovní. Pixely nejnižší úrovně uzlové mapy ukládají indexy 3D listových uzlů obsahující odpovídající fotony light-view mapy. Pixely z vyšší úrovně uzlové mapy mají uložen index nejnižšího společného předkovského uzlu pro předcházející uzly předešlé úrovně (obrázek **2.17**).

Stále je spuštěno jedno vlákno na pixel nejnižší úrovně uzlové mapy, ale když vlákno vystoupne z úrovně $i - 1$ na další, nejprve vyhledá odpovídající index předkovského uzlu v i-té úrovni uzlové mapy, který je uložen v nějakém pixelu p. Nechť jsou $p_0, \ldots, p_3$ pixely $(i - 1)$-té úrovně uzlové mapy, které byly zkombinovány do p. Když je předkovský uzel v p v nižší nebo v i-té úrovni stromu, lze odvodit, že všechna vlákna, která prošla $p_0, \ldots, p_3$ by následně skončila ve stejném uzlu. Proto by



Obrázek 2.17: Uzlová mapa (vlevo) je zkonstruována z light-view mapy. Je hierarchická (jako mipmapa) a má několik úrovní (velké kruhy představují další úroveň). Na nejnižší úrovni je pixel, který obsahuje index uzlu, ve kterém je umístěn odpovídající foton. Vyšší úrovně obsahují společného nejnižšího předka všech následných uzlů. Tato struktura umožňuje vyhnout se spouštění příliš mnoha vláken během filtrování fotonů (čárkovaná vlákna jsou zastavena). Dále s využitím light-view mapy lze zmenšit počet vláken: uzly, které nedostaly žádné fotony, také nedostanou žádná vlákna. Převzato z [2].

bylo vhodné ukončit všechna vlákna kromě jednoho, ale toho je těžké dosáhnout. Pro zastavení alespoň mnoha nepotřebných vláken je proveden heuristický proces a zastavena všechna ta vlákna, která neprošla p_0 (v praxi to je levý horní pixel). Obrázek **2.17** zobrazuje tečkovaně uzly, ve kterých jsou vlákna zastavena, když projdou.

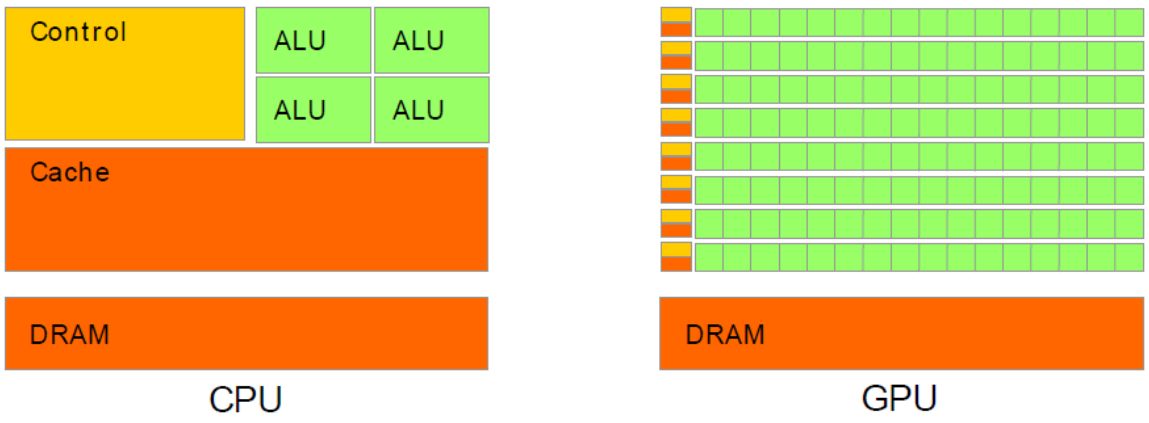
Tato strategie je docela úspěšná a dosahuje více jak dvakrát většího zrychlení v porovnání se strategií bez použití heuristiky na ukončování vláken (podle [2]). Dále toto filtrování je obecně daleko rychlejší než nativní implementace, která filtruje všechny uzly.

2.7 Architektura grafických karet

Grafická výpočetní jednotka (*Graphics processing unit – GPU*) poprvé vynalezena firmou NVIDIA v roce 1999 [24] je nejvíce rozšířený paralelní procesor současnosti. Poháněna nenasytnou touhou po grafice v reálném čase vypadající jako skutečný svět se GPU vyvinula v procesor s nebývalým výkonem s čísly s plovoucí řádovou čárkou a programovatelností. Dnešní GPU jsou ve velkém předstihu před CPU v aritmetickém výkonu a propustností paměti, což z nich dělá ideální procesor pro akceleraci různých paralelních aplikací.

Rozdíl architektury CPU a GPU [26] je zobrazen na obrázku **2.18**. CPU má málo velmi výkonných výpočetních jednotek, velkou cache, velké řízení a vykonávaní instrukcí mimo pořadí. GPU má velké množství méně výkonných jednodušších výpočetních jednotek, malé cache, malé řízení, více tranzistorů pro výpočty.

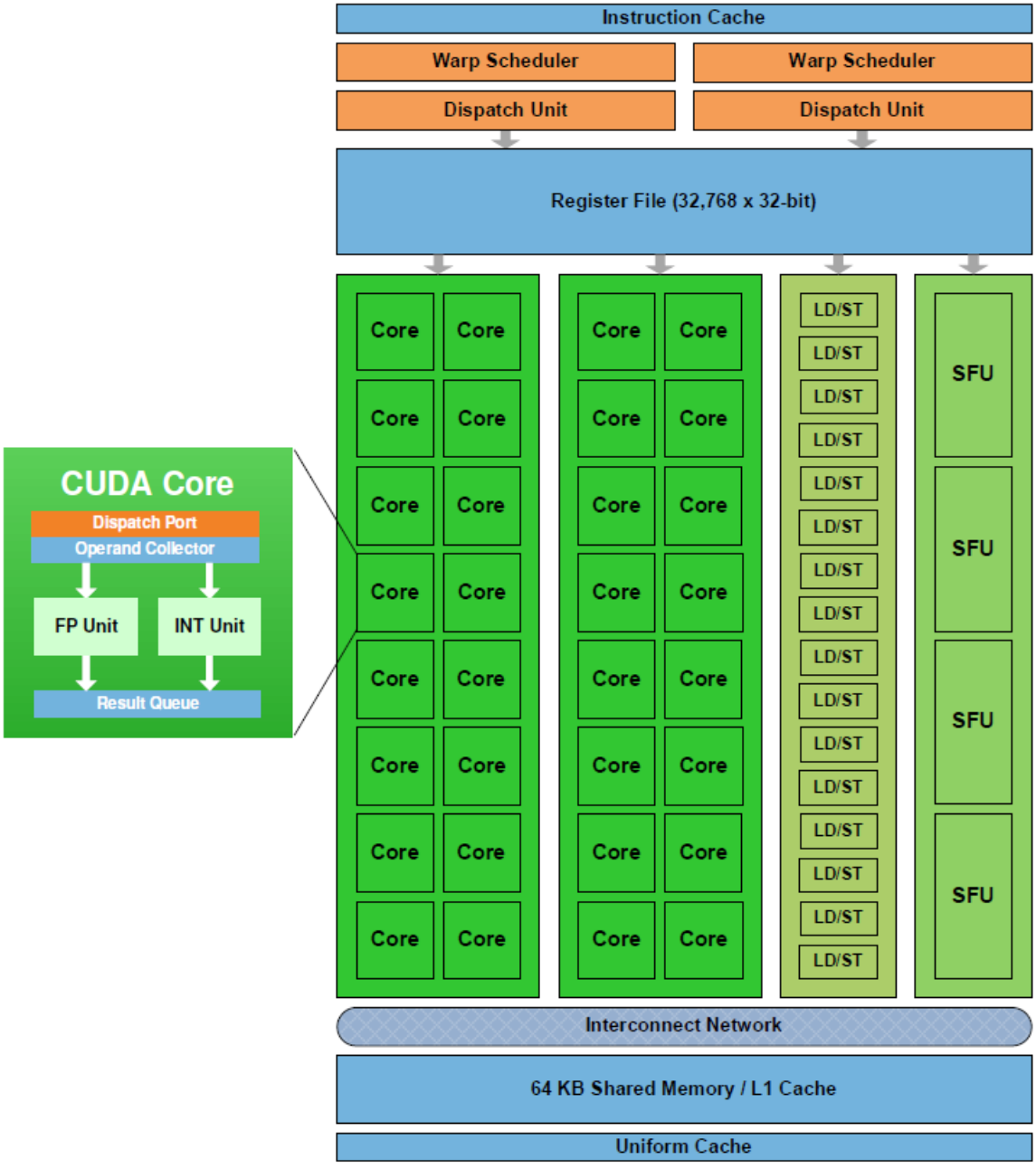
Snaha o využití GPU pro negrafické aplikace vedla ke vzniku GPGPU programům (*General purpose GPU*) [24]. První pokusy využívaly normální grafické API pro obecné výpočty, s využitím vyšších stínovacích programovacích jazyků jako DirectX, OpenGL a Cg. Zatímco GPGPU model předvedl velké zrychlení, čelil několika nevýhodám. První byla znalost grafického API a architektury GPU od programátora. Druhá byla nutnost vyjádřit problém jako souřadnice vrcholů trojúhelníků, textury a shader programů, což zvyšovalo komplexnost programů. Třetí nevýhodou bylo, že nebyly



Obrázek 2.18: Rozdíl mezi CPU a GPU. Převzato z [26].

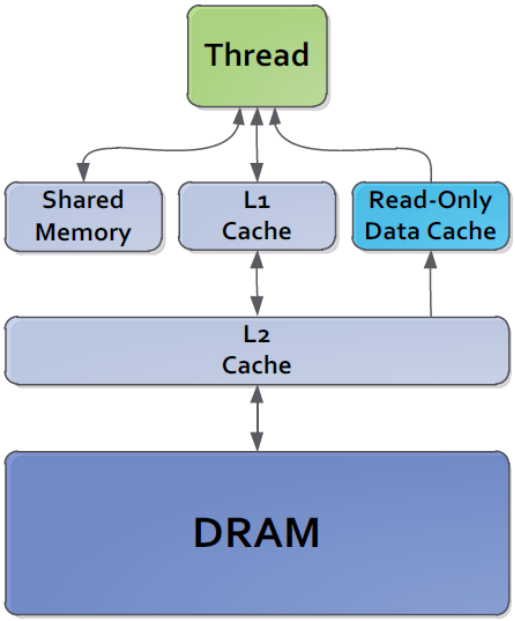
podporovány základní programovací vlastnosti, jako náhodné čtení a zápisy do paměti, což velmi omezovalo programovací model. Pro vyřešení těchto problémů byla vytvořena jednotná grafická a výpočetní architektura. Vznikl tak sjednocený shader model, kde shadery mají vstupy, výstupy, registry, přístup k texturám, bufferům a konstantám. Shader běží na výpočetních jednotkách GPU. Už není rozdíl u výpočetních jednotek mezi vertex a fragment shaderem, případně dalšími (geometry shader, teselace). Výpočetní prostředky se přidělují dynamicky: hodně trojúhelníků – více vertex shaderů, hodně fragmentů – více fragment shaderů.

GPU architektura. GPU je složena z několika multiprocesorů, grafické paměti a globálního plánovače, který rozděluje bloky vláken do plánovačů multiprocesorů. Multiprocesor je dále složen



Obrázek 2.19: Schéma multiprocesoru. Převzato z [24].

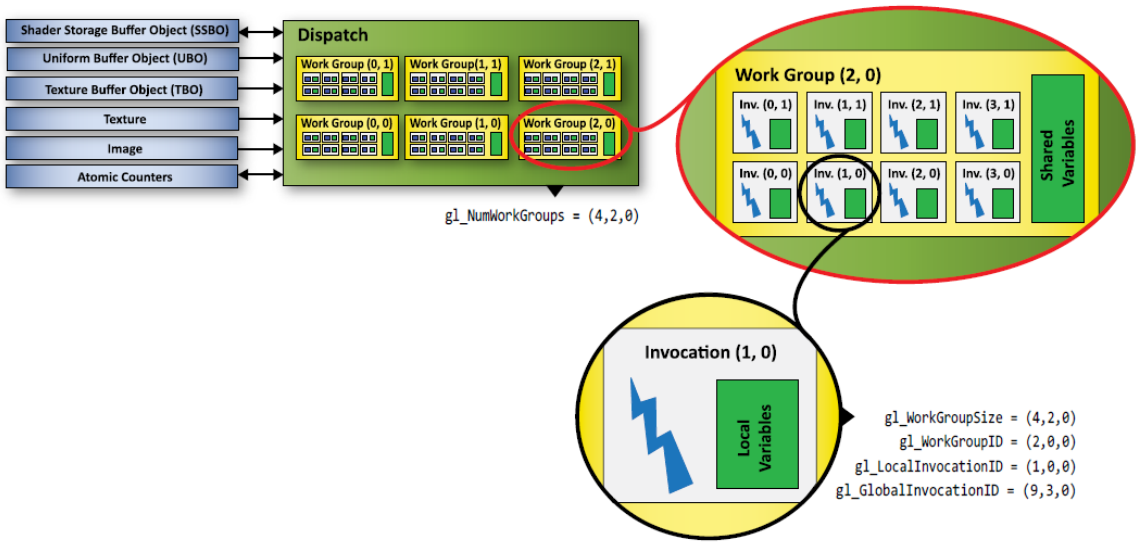
z velkého množství jader a různých druhů pamětí, jednotek nahrání/uložení do paměti, jednotky pro speciální funkce SFU (sinus, kosinus, odmocnina apod.), plánovačů *warpů* a *dispatch* jednotky, jak je zobrazeno na obrázku **2.19**. Každé jádro má plně zřetězenou celočíselnou aritmeticko-logickou jednotku (ALU) a jednotku s plovoucí řádovou čárkou (FPU). Jádro vykoná celočíselnou instrukci nebo instrukci s plovoucí řádovou čárkou ve vlákně za jeden takt.



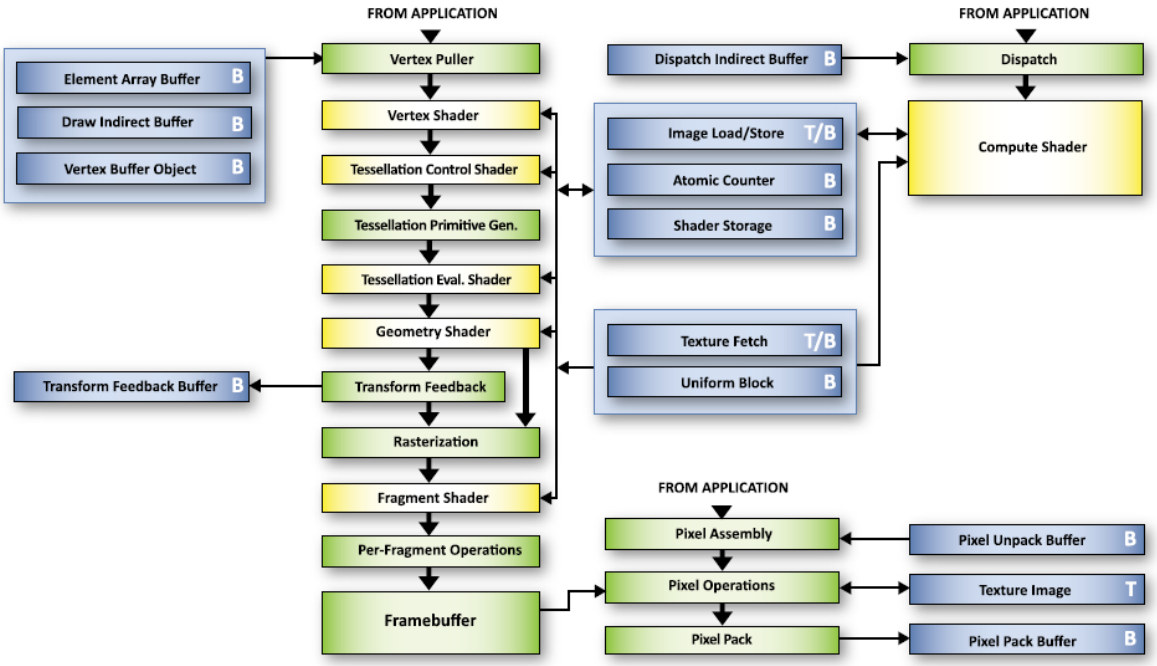
Obrázek 2.20: Hierarchie paměti GPU. Převzato z [25].

Paměťová hierarchie GPU. Spousta různých pamětí s různou velikostí a rychlostí. Obecně platí, čím blíže k jádru, tím rychlejší a tím menší. Globální paměť DRAM je velká, může mít kapacitu několik GB, ale je pomalá. Všechny multiprocesory sdílí L2 cache. Jádra v multiprocesoru se dělí o L1 cache; cache pouze na čtení dat; cache konstantní paměti; *Sdílenou paměť*, která slouží pro komunikaci mezi vlákny a registry, které jsou nejrychlejším typem paměti. Využití pamětí vláknem běžícím v jádře je zobrazeno na obrázku **2.20**.

OpenGL vykonávací programovací model a paměťový model. Schéma těchto modelů je zobrazen na obrázku **2.21**. Na GPU jsou spouštěny paralelní *kernely*, což jsou program shadery složené z instrukcí a běží ve vláknu (*invocation*) [24][27]. Instrukce v kernelu jsou spouštěny v mnoha instancích na jádrech multiprocesoru. Vlákna jsou seskupována do pracovních skupin (*work*



Obrázek 2.21: OpenGL vykonávací programovací model a paměťový model. Převzato z [27].



Obrázek 2.22: OpenGL Pipeline. Převzato z [27].

group), každé má své identifikační číslo (ID) v rámci své skupiny, programový čítač, registry, privátní paměť, vstupy a výstupní výsledky. Vlákna v pracovních skupinách mohou být na sobě závislá a v rámci jedné skupiny je lze synchronizovat pomocí bariér nebo sdílené paměti. Skupiny mohou být 1D, 2D a 3D (určuje pořadí vláken a jejich index). Mnoho pracovních skupin tvoří *dispatch* (taky může být 1D, 2D a 3D) a mají ID v rámci svého dispatch. Dispatch má přístup ke globální paměti: texturám, obrázkům, atomickým čítačům a několika druhům bufferů. Na jednom multiprocesoru je možno pustit více skupin, pokud na to vystačí zdroje (registry, sdílená paměť). Skupina je hardwarově rozřezána na *warpy*, které jsou složeny z vláken (např. 32). Vlákna ve warpu jsou vložena na různá jádra a jsou vykonávána současně (každé na jiném jádře). Velikost skupiny by měla být násobkem warpu.

OpenGL Pipeline. Typický program, který využívá OpenGL začíná voláními funkcí pro otevření okna pro framebuffer, do kterého bude program vykreslovat. Jsou provedena volání pro alokování GL kontextu, který je přiřazený k oknu, a pak mohou být prováděny OpenGL příkazy [27]. Schéma OpenGL Pipeline z verze 4.5 je zobrazeno na obrázku **2.22**. Tlusté černé šipky v této ilustraci ukazují tok dat v řetězci. Modré bloky indikují různé buffery, které plní nebo jsou plněny z OpenGL piperine, zelené bloky představují stupně s fixní funkcionalitou, žluté bloky ukazují programovatelné stupně. „T“ značí *Texture bidning* a „B“ *Buffer binding*.

3 Návrh aplikace

Návrh aplikace pro zobrazování globálního osvětlení ve scéně vychází z práce GigaVoxels od Cyrila Crassina [2], ale v některých částech se liší, tyto rozdíly jsou popsány v následující kapitole **3.1**. V kapitole **3.2** a jejích podkapitolách jsou pak podrobněji popsány jednotlivé části struktury aplikace, jako jsou inicializace scény, voxelizace, budování řídkého voxelového stromu, vytvoření *bricks*, jejich MIP mapování, ukládání fotonů a finální vykreslení.

3.1 Rozdíly oproti práci *GigaVoxels*

Práce Cyrila Crassina *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes* [2] se zabývá efektivním renderováním velkých scén a detailních objektů v reálném čase a s využitím objemové předfiltrované reprezentace geometrie a přibližného sledování kuželového paprsku založeného na voxelech. Také je v ní představen mechanizmus GPU cache poskytující velmi efektivní stránkování dat ve video paměti. Tato cache je spřažena s produkcí dat v řetězci schopném dynamicky načítat nebo vytvářet voxelová data přímo na GPU.

Moje diplomová práce je zaměřená pouze na voxelovou reprezentaci a globální osvětlení. Proto se tímto stránkováním cache, tzv. *out of core rendering* nezabývá. Dalším rozdílem je, že jsou použity pouze izotropické voxely – nezávisí u nich na směru pozorování, ale potřebují jen jednu hodnotu uloženou v paměti místo anizotropických voxelů – závisí na směru pozorování, nutno ukládat šest hodnot pro jeden voxel. Protože je pracováno pouze s povrchovou geometrií, která nemá žádný vnitřní objem, jsou tedy tvořeny jen voxely na povrchu a nejsou tvořeny vnitřní objemové voxely, není prováděn převod voxelů se stejnou barvou v celé *brick* na konstantní barvu, která by byla uložena v *node poolu* místo ukazatele na tuto *brick*, která by pak nebyla alokována uvnitř *brick poolu* a šetřila by tak paměť.

Rozdíly přímo u globálního osvětlení jsou: je prováděna voxelizace pouze statické geometrie, není tedy třeba voxelizovat každý snímek. Dynamické objekty nejsou voxelizovány (aktualizovány), takže není použit ani mechanizmus s časovými značkami pro rozeznání statických a dynamických objektů. Při zachytávání světla do stromové struktury není ukládán směr příchozího světla. Při tvorbě seznamu voxel fragmentů jsou sice ukládány normály povrchu a jsou pak i zprůměrovány do *node poolu*, ale při finálním renderování a sběru globálního osvětlení nejsou nakonec využity. Není vysílán ani spekulární kuželový paprsek pro zrcadlové odrazy. Další odlišnosti jsou u počítání MIP mapovaných hodnot fotonů, podrobněji v kapitole **3.2.8**.

3.2 Struktura aplikace

Na začátku běhu aplikace je vytvořeno okno, do kterého se bude vykreslovat a které také drží informace o grafickém kontextu. Pak je inicializováno grafické API a vytvořen objekt scény, který ukládá veškeré důležité informace o scéně a také je s ním manipulováno se scénou. Poté je provedena inicializace zbytku aplikace, po které se spustí nekonečná vykonávací smyčka. V té je prováděno měření času od posledního snímku, zpracování vstupů z myši a klávesnice a vykreslování snímků.

Součástí inicializace aplikace je inicializace objektu scény. Pokud ta proběhne v pořádku, následuje voxelizace scény, vybudování SVO, uložení ukazatelů na sousední uzly do SVO, vytvoření *brick* a jejich MIP mapování. Dále vyrenderování shadow mapy, uložení fotonů do SVO, distribuce fotonů do sousedních *bricks* a MIP mapování fotonů.

3.2.1 Inicializace scény

Podle zadaného parametru levelu stromu je vypočítáno rozlišení voxelů. Pak je proveden import modelu do scény – ze souboru je nahrán model a je vypočítán jeho obalový kvádr pro zjištění rozměrů pro změnu měřítka modelu na 1. Následuje nahrání textur ze souborů a uložení na grafickou kartu, generace bufferů pro indexy vrcholů trojúhelníků tvořících elementy, pozice vrcholů, normály vrcholů, texturovací souřadnice a materiály modelu. Následuje inicializace všech shader programů s definováním potřebných konstant, bufferů pro uložení matic, které jsou používány v shaderech a atomických čítačů, matice modelu (měřítko modelu na 1 jako převrácená hodnota velikosti modelu), nastavení pozice a vlastností kamery, vytvoření tří matic pro voxelizaci – ortogonální projekční matice je vynásobena směry hlavních os *x*, *y* a *z* (**2.4**). Dále je připraven framebuffer pro vykreslování shadow mapy – složený z barevné textury a textury hloubky s nastavenou porovnávací funkcí menší nebo rovno a porovnávací mód s referenční texturou.

3.2.2 Voxelizace

Voxelizace scény vychází z kapitoly **2.4** o jednoduché voxelizaci, která je základem kapitoly **2.5** o voxelizaci do *sparse voxel octree*. Nejprve je použit příslušný shader program pro voxelizaci, nastaveny potřebné matice a vynulován atomický čítač. Voxelizace je pak vlastně spuštění vykreslování (vykreslovací příkaz) s tímto kernelem. Protože je použit *seznam voxel fragmentů* (**2.5.3**), což je buffer předalokovaný v paměti grafické karty, je potřeba znát jeho velikost pro jeho vytvoření, ale to není v článku [1] řešeno. Ovšem velikost lze zjistit spuštěním voxelizace bez ukládání dat, pouze s počítáním množství vytvořených voxel fragmentů pomocí atomického čítače. Následně je alokován seznam voxel fragmentů, jako počet fragmentů krát velikost ukládaných dat pro každý fragment, tedy souřadnice fragmentu v rámci voxelové mřížky, barvy a normály. Poté je vynulován atomický čítač a spuštěna voxelizace již s ukládáním potřebných dat.

V geometry shaderu je vybírána osa projekce trojúhelníku, vytvářen obalující obdélník a prováděno zvětšení trojúhelníku pomocí konzervativní rasterizace. Ve fragment shaderu jsou zahazovány fragmenty mimo obalující obdélník a upravovány souřadnice fragmentu v závislosti na použité ose projekce, aby souhlasily s původními světovými souřadnicemi. Jedná se o převod hloubky fragmentu z intervalu 0 až 1 vynásobením rozlišením voxelů do intervalu 0 až rozlišení voxelů a případného rozdílu rozlišení voxelů a dané souřadnice, aby nebyla zrcadlově otočená kolem osy. Nakonec jsou podle aktuálního nastavení jen počítány fragmenty čítačem nebo i uložena data do bufferu.

3D konzervativní rasterizace. Při pokusu o implementaci algoritmu podle Hasselgrena et al. [4] (**2.4.1**) se mi však nepodařilo dosáhnout správných výsledků. Algoritmus pracuje v *clip space* a zanedbává z (hloubkovou) komponentu vrcholu trojúhelníku, tzn., že trojúhelník je zvětšen, po ortogonálním promítnutí, pouze v rovině x a y. Promítnutý zvětšený trojúhelník vypadá z pohledu kamery správně, ale zvětšený trojúhelník je oproti původnímu menšímu pootočený díky tomu, že hloubka vrcholů zůstala stejná, ale nastal posun vrcholů ve směru os x a y. Problém nastává při voxelizaci velkých trojúhelníků, kdy dochází k uložení některých zvoxelizovaných částí trojúhelníku do jiných voxelů, než do kterých by měly být uloženy.

Proto jsem navrhl vlastní *3D konzervativní rasterizaci*. Ta je založena na výpočtu vektoru posunu **m** pro každý vrchol trojúhelníku a podobně jako v práci Hasselgrena et al. [4] výpočtu ohraničujícího obdélníku, podle kterého se zahazují fragmenty. Vektor posunu **m** je pro aktuální vrchol vypočítán následovně: jsou vypočítány dva vektory hran $\mathbf{e_1}$ a $\mathbf{e_2}$ trojúhelníku směřujících do aktuálního vrcholu a znormalizovány. Tím jsou získány dva směry posunu vrcholu, jejichž součtem je získán výsledný směr posunu. Pak je nutné zjistit velikost tohoto posunu. Skalárním součinem znormalizovaných vektorů hran $\mathbf{e_1}$ a $\mathbf{e_2}$ lze zjistit kosinus úhlu α mezi těmito vektory:

$$\cos\alpha = \mathbf{e_1} \cdot \mathbf{e_2}. \tag{3}$$

Toto vychází ze vzorce:

$$\cos\alpha = \frac{\mathbf{e_1} \cdot \mathbf{e_2}}{|\mathbf{e_1}||\mathbf{e_2}|}, \tag{4}$$

kde normalizace vektorů hran $\mathbf{e_1}$ a $\mathbf{e_2}$ zajišťuje vyhnutí se dělení násobkem velikostí obou vektorů. Díky podobnosti trojúhelníků lze tento úhel využít pro výpočet velikosti přepony p (použité pro zjištění velikosti posunu) při znalosti velikosti protilehlé odvěsny h k úhlu α, což je vzdálenost mezi hranou zvětšovaného trojúhelníku a hranou zvětšeného trojúhelníku. Jako délku protilehlé odvěsny h jsem zvolil:

$$h = \frac{\sqrt{3}}{x}, \tag{5}$$

kde $\sqrt{3}$ odpovídá objemové úhlopříčce v jednotkové krychli, což je největší možný posun hrany a kde x je rozlišení voxelů v jednom směru (např. 512). Takto pevně zvolený posun může trojúhelník zvětšovat více, než by měl, ale přesné počítání délky h by bylo příliš výpočetně náročné, protože by bylo nutné ho provádět pro všechny hrany a nakonec stejně ještě dochází k ořezání podle ohraničujícího obdélníku. Pak přepona p je:

$$p = \frac{h}{\sin\alpha} = \frac{h}{\sqrt{1 - \cos\alpha \cdot \cos\alpha}}. \tag{6}$$

Sinus α je počítán z kosinu α, aby nebylo nutné počítat přímo úhel α pomocí arkuskosinu, protože to je výpočetně náročná operace. Vektor posunu **m** je potom:

$$\mathbf{m} = p(\mathbf{e_1} + \mathbf{e_2}), \tag{7}$$

který je přičten k souřadnicím polohy aktuálního vrcholu trojúhelníku a tím je získána nová 3D poloha vrcholu.

3.2.3 Budování SVO

Tato část návrhu aplikace vychází z kapitoly **2.5.4**. Podobně jako u voxelizace **3.2**.2 není v článku [1] a v pracích [2][3] řešena alokace *node poolu* ve video paměti. Pouze je zmíněna snaha alokovat co největší množství paměti, podle dostupné kapacity grafické karty a možnost škálovatelnosti techniky pro budoucí použití u zvětšujících se kapacit video pamětí. Počet dělení uzlů při postupném budování SVO a celkový počet nakonec vzniklých uzlů nelze dopředu určit. Řešením je vybudování SVO dvakrát. Poprvé je voxel octree vybudován jako plný a ne řídký strom, jsou tedy alokovány všechny jeho uzly, ale pomocí atomického čítače je zaznamenáván skutečný počet alokovaných uzlů ve stromu. Podruhé je octree vybudován již jako řídký strom s alokováním pouze skutečně využitých uzlů, tím dochází k šetření paměti, pokud není zaplněná celá scéna.

Podle pořadí budování SVO je tedy vypočítán plný nebo přesný počet uzlů a alokován základní *node pool*, obsahující ukazatele na potomky a vynulován atomický čítač. Pro urychlení budování stromu, které je prováděno v cyklu, je využito nepřímé volání a vykonávání shader programu, při kterém jsou potřebné infomace počítány a uloženy na grafické kartě a nedochází ke komunikaci s procesorem. Proto je potřeba inicializovat buffery s informacemi pro nepřímé volání. Jeden obsahuje počet pracovních skupin a druhý obsahuje současný level, současný počet uzlů, počet uzlů v minulém cyklu, počet vláken a posunutí v *node poolu* korespondující s aktuálním levelem. Pak je v cyklu s postupným zanořováním ve stromu pro všechny levely stromu spouštěn označovací a alokovací shader program. Po dokončení cyklu jsou alokovány další druhy *node poolů*: buffer pro barvu, pro voxelové souřadnice a pro normály. Poté je použit program pro uložení dat do listů stromu.

Označovací kernel je spouštěn pro každý fragment ze seznamu voxel fragmentů. Z fragmentu jsou zjištěny jeho voxelové souřadnice, ty jsou poděleny rozlišením voxelů a tak převedeny do intervalu 0 až 1, pak je možno použít zanořovací schéma pohybu stromem z kapitoly **2.5.5**.

Zanořování probíhá v cyklu až do aktuálního levelu, kdy je postupně procházeno přes potomkovské uzly. Pokud by měl být poslední uzel rozdělen, jeho nejvyšší bit je nastaven na 1. Jestliže při zanořování dojde k nalezení uzlu (a nejedná se o poslední uzel), který má nejvyšší bit roven 0, je sestup předčasně ukončen a uzel nebude rozdělen. V jednom vlákně jsou vypočítána potřebná data pro nepřímé spuštění alokačního shader programu.

V alokačním kernelu je pro nově označené uzly k rozdělení pomocí atomického čítače alokován index v rámci *node poolu*, pak je z indexu vypočítána adresa poduzlu a uložena zpět (i s nejvyšším bitem nastaveným na 1) do uzlu.

Program pro uložení dat do listů stromů je stejně jako označovací program spouštěn pro všechny fragmenty a je v něm podle voxelových souřadnic prováděno zanoření. Po nalezení listového uzlu jsou barva a normála uložená ve fragmentu atomicky uloženy s průměrováním hodnot (podle kapitoly skládání fragmentů **2.4.2**) do příslušného druhu *node poolu*, souřadnice nejsou ukládány atomicky, ale normálně, pro daný voxel budou vždy stejné.

3.2.4 Uložení ukazatelů na sousední uzly v SVO

Nejprve je vytvořen *node pool* pro ukazatele na sousední uzly. Jako sousední uzly jsou brány uzly v 6okolí voxelu, tedy 6 voxelů, které přímo sousedí a sdílí stěnu s tímto voxelem. Následně je v cyklu zvětšován level stromu a pro každý level je spuštěn shader program hledající sousední uzly pro každý voxel v daném levelu. Tento program funguje tak, že podle souřadnic aktuálního voxelu je zjištěn index v *node poolu* sousedních uzlů, ke kterému je přičteno posunutí aktuálního souseda podle směru, jak je uvedeno v tabulce **3.1** a na tento nový index je uložen ukazatel na souseda. Sousední uzel je hledán tak, že k souřadnici voxelu je přičtena nebo odečtena jednička v příslušném směru hlavních os a provedeno zanoření. Pokud je sousední uzel nalezen, je ukazatel na něj hledaným výsledkem, pokud není nalezen, jako výsledek je použit kořenový uzel s adresou 0, tak lze později poznat neúspěšné hledání. Při hledání musí být hlídán rozsah voxelových souřadnic, aby nedošlo k hledání mimo voxelový prostor.

+X	+Y	+Z	-X	-Y	-Z
0	1	2	3	4	5

Tabulka 3.1: Tabulka směrů sousedních uzlů a příslušného indexu posunutí

3.2.5 Vytvoření *bricks*

Jak je popsáno v kapitolách **2.5.1** a **2.6.3**, nejsou použity přímo voxely $2 \times 2 \times 2$ v uzlovém *tile*, ale je využívána rohově centrovaná voxelová konfigurace s $3 \times 3 \times 3$ voxelovými *bricks*. V práci [2] ani [3] však není vysvětleno, jak *bricks* z voxelů vytvořit. Proto jsem byl nucen navrhnout vlastní postup vytvoření *bricks*. Pro vypočítání rohových hodnot uzlu 2^3, které tvoří středy *bricks* 3^3 (obrázek **2.11**)

jsou načteny voxely z okolí rohu a z těchto 8 hodnot je vypočítán průměr a uložen jako středová hodnota.

Bricks pro všechny levely stromu jsou uloženy v jedné velké 3D textuře – *brick poolu*. Pro zjištění přesné velikosti této textury, aby nebylo nutné alokovat největší možnou velikost, kterou může textura mít, je nejprve spočítán počet vytvořených *bricks* v nejnižším levelu stromu v shader programu. Pak je vypočítána minimální potřebná velikost textury, ta je alokována a poté jsou *bricks* znovu vypočítány, ale už jsou ukládány do připravené textury.

Potřebná velikost textury je vypočítána z počtu vytvořených *bricks* tak, že je spočítána třetí odmocnina z tohoto množství, ta je zaokrouhlena nahoru na nejbližší celé číslo (tak je zjištěno rozlišení *bricks* v nejnižším levelu stromu) a to je umocněno na třetí. Vyšší levely stromu nejsou počítány přesně, místo toho je celkový počet *bricks* vynásoben dvakrát, což odpovídá tomu, že počet *bricks* v levelu stromu bude vždy větší, než součet *bricks* ze všech vyšších levelů. Tento celkový počet je také převeden pomocí třetí odmocniny a zaokrouhlení na rozlišení *bricks* pro krychli. Rozlišení je vynásobeno třemi pro získání počtu texelů textury – jedna *brick* obsahuje 3^3 voxelů. Nakonec je nalezena nejbližší mocnina dvou tohoto rozlišení, pro lepší vlastnosti textury. S tímto finálním rozlišením je alokován *brick pool* jak pro samotnou scénu, tak i pro fotony.

Při výpočtu *bricks* je nejprve v jediném vlákně pracovní skupiny vynulován sdílený lokální čítač. Pak jsou jednotlivými vlákny zanořováním ve stromu načteny voxely (jejich barva), uložené v *node poolu*, do sdílené paměti. Sdílená paměť je použita proto, aby co nejvíce vláken mohlo využít již načtené hodnoty a nemuselo docházet k neustálým přístupům do globální paměti. Pokud načítaný voxel neexistuje – je prázdný, je jako náhradní barva zvolena černá – nulová barva s nulovou alfa složkou (průhledností). Protože pro výpočet průměru barev je potřeba více hodnot, než je pak výsledků, jsou načítány také okolní okrajové hodnoty. Proto je potřeba hlídat rozsah voxelových souřadnic.

Načítání okrajových hodnot lze řešit dvěma způsoby (uvažováno jako 3D problém, krychle): prvním je, že okrajová vlákna načítají více hodnot – stěnová jednu, hranová tři a rohová sedm, což způsobí čekání na rohová vlákna a druhým způsobem je, že je spouštěna řada vláken navíc v každém směru, která pouze načte hodnoty, ale dále je nevyužita a vzniklý posun ID v pracovních skupinách je nutno kompenzovat.

Po načtení všech potřebných hodnot jsou tyto barvy zprůměrovány a výsledek je uložen do sdílené paměti pro pozdější zpracování. V každém vlákně je průměrováno 8 hodnot barvy použitím vzorce:

$$\mathbf{c}_r = \frac{\sum_{i=0}^{n=7} \mathbf{c}_i \alpha_i}{\sum_{i=0}^{n=7} \alpha_i}, \tag{8}$$

kde $\mathbf{c}_r$ je výsledný průměr barvy, $\mathbf{c}_i$ je barva a α_i je příslušná hodnota průhlednosti. Barva je násobena průhledností z důvodu zamezení ovlivňování výsledku zástupnou černou barvou prázdných voxelů.

Dalším krokem je výpočet, zda má být alokována nová *brick* v rámci *brick poolu*. Tedy zda 3^3 výsledných průměrů uložených ve sdílené paměti obsahuje nějakou barvu. K tomu lze využít součet průhledností z počítané *brick*. Alokace celé *brick* 3^3 je prováděna vždy jen jedním vláknem. Součet lze provést buď v co druhém vlákně, které v cyklu sečte všech 27 hodnot, nebo lze využít paralelní sumu prefixů. Pokud není průhlednost nulová, je pomocí atomického čítače získán lineární $index$ a z něj jsou vypočítány 3D souřadnice *brick poolu* **coords** jako:

$$\mathbf{coords} = \big(mod(index, dim);\ mod(div(index, dim), dim);\ div(index, dim \cdot dim)\big), \qquad (9)$$

kde mod je zbytek po celočíselném dělení, div je celočíselné dělení a dim je rozlišení *brick*. Tyto souřadnice jsou ještě vynásobeny třemi pro získání rozlišení *brick poolu* v texelech a uloženy do rodičovského uzlu v *node poolu* pro ukazatele do *brick poolu*, což je další typ *node poolu* a také uloženy do sdílené paměti, aby je mohly využít ostatní vlákna, která ukládají hodnoty do stejné *brick*, ale jsou v rámci ní posunutá.

Nakonec je třeba zprůměrované barvy uložit do příslušných *brick*. Každým vláknem jsou podle jeho ID načteny ze sdílené paměti souřadnice *brick*, do které patří. Pokud souřadnice existují, je k nim podle lokální pozice v *brick* připočítáno posunutí a do příslušného texelu je uložena barva. Protože sousedící *bricks* sdílí hodnoty, ale barva je ukládána jen do jedné *brick*, je potřeba tuto hodnotu uložit ještě jednou i do sousedící *brick*.

3.2.6 MIP mapování *bricks*

Filtrování hodnot v *bricks* vychází z kapitoly **2.6.4**. Je prováděno v cyklu od nejnižší úrovně stromu směrem nahoru ke kořenovému uzlu. Práce v kernelu je podobná jako při vytváření *bricks* (**3.2**.5). Nejprve jsou načteny hodnoty *bricks* z nižší úrovně, sdílené hodnoty na hranicích sousedících *bricks* není nutno načítat dvakrát, stačí hodnota z jedné *brick*. Pro výpočet 3^3 *brick* vyšší úrovně je potřeba načíst 7^3 hodnot z *brick* nižší úrovně, podobně jako je zobrazeno 7^2 hodnot pro 2D případ na obrázku **2.12**. Již při načítání je aplikován váhovací filtr podle **2.6.4**. Opět je využita sdílená paměť pro omezení čtení z globální paměti a znovupoužití již načtených hodnot.

Po načtení hodnot je třeba zprůměrovat 3^3 oblast pro získání hodnoty pro *brick* vyšší úrovně. To lze provést jedním vláknem sečtením všech 27 hodnot v cyklu podle vzorce (8) s nastaveným $n = 26$ nebo využitím paralelní sumy prefixů ve více vláknech. Opět je ověřena nenulová průhlednost a pomocí atomického čítače alokován index s převodem na 3D souřadnice do *brick poolu*, které jsou uloženy do rodičovského uzlu. Nakonec jsou vlákny, která počítala průměrné hodnoty, uloženy výsledky do *bricks* vyšší úrovně.

3.2.7 Shadow mapa a uložení fotonů do SVO

K vyrenderování shadow mapy, zde zvané *light-view map* je využit framebuffer připravený během inicializace **3.2.1**. Dále je nastaveno rozlišení shadow mapy. Scéna je vykreslována z pohledu světla,

pro které jsou vytvořeny pohledová a projekční matice. Jedná se o standardní renderování s ukládáním barvy a hloubky do cílových textur, které mohou být dále použity pro stíny ve scéně a uložení fotonů do SVO. Fotony vznikají pouze na povrchu geometrie, proto je pro ně vždy možné najít příslušný voxel.

Uložení fotonů do SVO je založeno na kapitole **2.6.11**. Fotony nejsou ukládány do stejného *brick poolu* jako je uložena zvoxelizovaná scéna, ale je pro ně vytvořena další stejně velká 3D textura, a také je využívána již vybudovaná hierarchie ukazatelů v *node poolu*. Nejprve je vykreslena barevná textura shadow mapy tak, že pro každý její pixel je spuštěno jedno vlákno. V něm je vyčtena hloubka z textury a souřadnice fragmentu jsou převedeny do rozsahu -1 až 1 a vynásobeny inverzní maticí pro světelnou pohledovou a projekční matici. Tím jsou zjištěny původní světové souřadnice fragmentu a ty jsou dále převedeny na voxelové souřadnice. Podle těchto souřadnic je načtena *brick* a podle pozice fragmentu uvnitř ní je do ní uložena barva fragmentu jako foton. Při větším rozlišení shadow mapy než je rozlišení voxelů připadne na jeden voxel v *brick* více fotonů, proto je nutné je atomicky sečíst. Foton může připadnout pouze do jedné *brick*, a pokud je uložen do hraničního voxelu, který by měly být stejný pro sousedící *bricks*, je nutné provést distribuci fotonů do sousedních *bricks*.

3.2.8 Distribuce fotonů do sousedních *bricks* a MIP mapování fotonů

Tato část také vychází z kapitoly **2.6.11**, ale částečně se liší. Není provedeno šest průchodů, ale pouze tři, ve směru os *x*, *y* a *z* (bez záporných směrů) z důvodu šetření přístupů do paměti, jak bude vysvětleno dále. V kernelu je nejprve z *node poolu* načten ukazatel na sousední uzel podle směru, a pak jsou načteny ukazatele na *brick* a sousední *brick*. Pomocí těchto ukazatelů jsou načteny hraniční hodnoty obou *brick*, které jsou dále zprůměrovány v lokální proměnné (odpovídá přičtení zleva doprava v kapitole **2.6.11**). Protože nedochází ke konkurenčnímu přístupu, nemusí být použity atomické operace. Protože už je výsledek v lokální proměnné, je možno ho uložit jak do pravé, tak i do levé *brick* najednou (odpovídá kopírování sumy zprava doleva). Není nutno ukládat pouze do pravé, spustit další průchod, znovu načíst hodnotu z paměti a uložit do levé *brick* (což by znamenalo ještě další čtení globální paměti kvůli ukazatelům).

MIP mapování je prováděno podobně, jako MIP mapování *bricks* (**3.2.6**). Je využíváno toho, že pro každý foton existuje voxel a ten již má pozici v rámci hierarchie stromu. Proto při filtrování fotonů jsou použity již vytvořené ukazatele na *bricks*, načítané z rodičovských uzlů a nejsou alokovány nové *bricks*. Není tedy využito schéma *Distribuce přes úrovně* z kapitoly **2.6.11** se třemi oddělenými průchody pouze s částečně spočítanými filtrovanými výsledky a použitím dříve představeného přenosu mezi *bricks* pro dopočítání výsledku. Není použita ani *Uzlová mapa* pro vyhnutí se filtrování nulových hodnot.

3.2.9 Vykonávací smyčka

V této nekonečné vykonávací smyčce je prováděno měření času od posledního snímku, který je využit při pohybu po scéně, aby byl pohyb nezávislý na frekvenci snímků za sekundu. Dále zpracování vstupů z myši a klávesnice, kde pohybem myši je ovládáno natáčení kamery ve scéně nebo přibližování a oddalování, podle stisknutého tlačítka. Klávesnicí je ovládán pohyb kamery ve scéně, případně ukončení nekonečné smyčky. Také zde probíhá vykreslování snímků, podrobněji popsané v kapitole **3.2.10**.

3.2.10 Finální vykreslení

Finální vykreslení scény je složeno ze standardního zobrazení modelu, stínů vytvářených pomocí shadow mapy a efektu globálního osvětlení získaného z voxelů. Pro každý fragment je vypočítán směr ke světlu, normála a intenzita dopadajícího světla. Dále je zjištěna viditelnost světla z fragmentu pomocí projekce hloubky shadow mapy kvůli stínům. Pak je potřeba vypočítat směry kuželových paprsků pro sběr globálního osvětlení.

Paprsků je použito pět, každý s rozpětím $\frac{\pi}{3}$ (60°), jeden ve směru normály **N** (ve světových souřadnicích) a další čtyři s odklonem $\frac{\pi}{3}$ od normály a v pravidelném rozestupu $\frac{\pi}{2}$ (90°) mezi sebou, které tak tvoří aproximaci polokoule nad fragmentem (jehož pozice je také ve světových souřadnicích). Pro získání prvního nenormálového paprsku je vypočítán vektorový součin **cross** mezi normálou **N** a osou **x**. Pokud by se stalo, že normála by byla také ve směru osy **x** a vektorový součin by byl nulový, je použita osa **y**. Výsledný směr prvního paprsku **dir** je vypočítán jako lineární kombinace vektorů **cross** a **N**:

$$\mathbf{dir} = \cos\frac{\pi}{6} \cdot \mathbf{cross} + \sin\frac{\pi}{6} \cdot \mathbf{N} \tag{10}$$

a opačný směr **opDir** druhého paprsku je:

$$\mathbf{opDir} = 2\mathbf{N}(\mathbf{dir} \cdot \mathbf{N}) - \mathbf{dir}. \tag{11}$$

Pro třetí paprsek je vypočítán vektorový součin mezi normálou **N** a předchozím vektorovým součinem **cross** a směr je vypočítán stejným postupem jako u prvního paprsku pomocí rovnice (10) a opačný směr pro čtvrtý a poslední paprsek pomocí rovnice (11).

Pro všech pět směrů s počátkem **pos** ve fragmentu je provedeno sledování kuželového paprsku a získáno tak globální osvětlení, podle **2.6.6**. Aby bylo možno sledovat paprsek, je scéna uvnitř obalové krychle, se kterou jsou počítány protnutí. Jsou tedy vypočítány vstupní a výstupní parametry protnutí t_{in} a t_{out}, pokud je vstupní menší než výstupní, došlo k protnutí a je pokračováno dále. Pokud je vstupní parametr menší než nula, sledování začalo zevnitř krychle a je potřeba tento parametr upravit na nulu. Výstupní parametr t_{out} je nastaven jako maximální parametr scény a vstupní t_{in} jako lokální maximum t_{max}. Sledování paprsku je pak prováděno v cyklu, dokud je

lokální maximum t_{max} menší než maximum scény, tedy dokud paprsek neopustí obalovou krychli, nebo dokud není nasbírána dostatečná průhlednost alfa blížící se k hodnotě jedna.

V cyklu je vypočítána přesná světová pozice **pos** pomocí přičtení násobku lokálního maxima t_{max} a směru paprsku **dir** k počátku **origin**:

$$\mathbf{pos} = \mathbf{origin} + t_{max} \cdot \mathbf{dir} \tag{12}$$

a dále poloměr kužele v tomto místě $radius$ jako vzdálenost mezi počátkem **origin** a pozicí **pos** vynásobená tangens $\frac{\pi}{6}$ (odpovídá polovině z rozpětí $\frac{\pi}{3}$):

$$radius = dist(\mathbf{origin}, \mathbf{pos}) \cdot \tan\frac{\pi}{6}. \tag{13}$$

Pozice je pak převedena do texturovacích souřadnic v intervalu 0 až 1, aby bylo možno využít schéma zanořování do voxelového stromu (**2.5.5**).

Ke standardnímu zanořování do stromu v cyklu (zde je to zanořený cyklus uvnitř cyklu sledování paprsku) je přidána kontrola zda poloměr kužele paprsku $radius$ je už větší než diagonála voxelu pro ukončení zanořování na požadované úrovni detailu (**2.6.6**). Dále je přidáno zmenšení obalové krychle na polovinu podle směru zanoření, aby odpovídala velikosti a poloze potomkovského uzlu a výpočítáno protnutí s touto novou krychlí, jehož výstupní parametr t_{out} je nastaven jako nové lokální maximum t_{max}, odpovídající posunu na další voxel, které bude použito v další iteraci cyklu sledování paprsku pro výpočet počáteční pozice v rovnici (12). Ještě je dvakrát zmenšena diagonála, odpovídající zanoření ve stromu na nižší úroveň s menšími voxely, pro pozdější porovnání s poloměrem paprsku $radius$.

Pokud existuje uzel na příslušné úrovni, je z něj načten ukazatel na *brick*, pak je zjištěna poloha v rámci *brick* ještě dalším zanořením a poté je s touto souřadnicí provedeno vzorkování 3D textur, čímž je získána interpolovaná hodnota barvy z *brick poolu* reprezentujícího zvoxelizovanou scénu i z druhého *brick poolu* reprezentujícího světelnou informaci uloženou jako fotony. Takto jsou pro všechny paprsky získány jednotlivé příspěvky ke globálnímu osvětlení.

Nakonec je načtena barva modelu z textury nebo materiálu, vynásobena intenzitou a viditelností, je k ní přičtena ambientní složka a globální osvětlení vynásobené jeho váhou. Takto je získána finální barva fragmentu scény.

4 Implementace

V této kapitole jsou popsány použité knihovny při implementaci aplikace. Dále jsou popsány některé vybrané implementační detaily týkající se alfa kanálu při voxelizaci, práce s bufferem seznamu voxel fragmentů, kódování vícerozměrných vektorů na celočíselný typ, výpočet klouzavého průměru pomocí atomických operací, volba lokálních velikostí v pracovních skupinách a spouštění a ovládání aplikace.

4.1 Použité knihovny

Aplikace je implementována v programovacím jazyce C++. Jako grafické API je použito OpenGL 4.5 [27] s použitím knihovny GLEW. Školou byla poskytnuta knihovna GPUEngine, která ulehčuje práci s OpenGL objekty v C++, obaluje standardní C rozhraní do C++ tříd a poskytuje objekty jako okno aplikace a kamera. GLM [28], neboli *OpenGL Mathematics*, je C++ matematická knihovna pouze s hlavičkovými soubory pro grafický software založená na specifikacích *OpenGL Shading Language* (GLSL). Další knihovnou je SDL 2 [29], neboli *Simple DirectMedia Layer 2*, je to multiplatformní vývojářská knihovna umožňující přístup na nízké úrovni k audiu, klávesnici, myši, joysticku a grafickému hardwaru přeš OpenGL a Direct3D. AntTweakBar [30] je malá knihovna pro rychlé přidání uživatelského rozhraní do grafické aplikace pro interaktivní úpravu parametrů na obrazovce. DevIL [31], neboli *Developer's Image Library*, je knihovna na načítání, ukládání, převody a další manipulaci s velkým množstvím obrázkových formátů. Assimp [32], neboli *Open Asset Import Library*, je knihovna pro importování různých dobře známých formátů 3D modelů jednotným způsobem. Jako ukázkový model je použit upravený model Crytek Sponzy [33].

4.2 Implementační detaily

Zde jsou popsány některé vybrané implementační detaily týkající se alfa kanálu při voxelizaci, práci s bufferem seznamu voxel fragmentů, kódování vektorů souřadnic, barvy a normál na celočíselný typ v různém formátu, výpočet klouzavého průměru pomocí atomických operací a volba lokálních velikostí v pracovních skupinách.

4.2.1 Voxelizace a alfa kanál

Při voxelizace je ignorována složka průhlednosti barvy α, protože čtvrtá složka 4D *float* vektoru barvy je později použita jako počítadlo pro klouzavý průměr (2) podle kapitoly **2.4.2**, ukázka implementace je v sekci **4.2.5**. Pro průhlednost by bylo potřeba vytvořit další *node pool*, kam by mohla být ukládána. V dalších částech je sice s průhledností pracováno, ale je přímo nastavena na 1,

nemá to však velký vliv, protože aplikace je připravena na práci i s jinými hodnotami průhlednosti než 1, stačilo by přidat další *node pool* a nastavování alfa kanálu na 1 nahradit čtením z tohoto *node poolu*.

4.2.2 Seznam voxel fragmentů

Seznam voxel fragmentů je buffer ve video paměti, je alokován jako počet fragmentů krát velikost ukládaných dat pro každý fragment, tedy souřadnice fragmentu v rámci voxelové mřížky, barvy a normály. Pro každou tuto složku jsou potřeba 3 floaty, celkem 9. Je možno přidat i další float pro průhlednost. V bufferu jsou pak jednotlivé složky uloženy za sebou. Je k nim přistupováno s použitím základního indexu (původně získaného atomickým čítačem) vynásobeného 9, což znamená přístup k devítici hodnot jako k jedné buňce bufferu. Přičtením posunutí 0 až 8 jsou pak indexované jednotlivé položky z devítice.

4.2.3 Inicializace *tiles* při budování SVO

V kapitole **2.5.4** je popsán poslední krok inicializace nových uzlů na *null* ukazatele na potomkovské uzly po jejich alokování při vytváření SVO. Pro tento krok není třeba vytvářet vlastní shader program, protože je využita funkce OpenGL na čištění bufferu, kterou je celý buffer vynulován (odpovídá *null* ukazateli).

4.2.4 Kódování souřadnic, barvy a normál

Ukládání souřadnic, barvy a normály do *node poolu* ve formě 3D float vektorů by bylo velmi paměťově náročné. Proto je každý vektor zakódován na jeden *unsigned int*. Souřadnice jsou kódovány na formát `RGB10A2`, kde složky RGB obsahují 10 bitů a složka A 2 bity (u souřadnic nevyužity). Barva a normála jsou převáděny na formát `RGBA8`, kde každá složka má 8 bitů (A slouží jako počítadlo průměru). Ukázka funkce převodu 4D float vektoru na unsigned int pro `RGBA8`:

```
uint convVec4ToRGBA8(in vec4 val)
{
      return      (uint(val.w) & 0x000000FF) << 24U |
                  (uint(val.z) & 0x000000FF) << 16U |
                  (uint(val.y) & 0x000000FF) <<  8U |
                  (uint(val.x) & 0x000000FF);
}
```

Také je nutná funkce pro zpětný převod na 4D vektor:

```
vec4 convRGBA8ToVec4(in uint val)
{
      return vec4(      float((val & 0x000000FF)),
                        float((val & 0x0000FF00) >>  8U),
                        float((val & 0x00FF0000) >> 16U),
                        float((val & 0xFF000000) >> 24U));
}
```

4.2.5 Klouzavý průměr s atomickými operacemi

Implementace *Průměrování hodnot s využitím atomických operací* z kapitoly **2.4.2** s klouzavým průměrem využívající operaci *porovnej-a-vyměň* `atomicCompSwap` s ukázkou pro průměrování barvy do *node pool* bufferu pro barvu `NPCBuffer`, kde jsou také využity funkce kódování z kapitoly **4.2.4**:

```
void atomicRGBA8Avg(uint index, vec4 val)
{
      val.rgb *= 255.0f;      //Optimalization for color
      val.a = 1.0f;           //Initialize counter

      uint newVal = convVec4ToRGBA8(val);
      uint prev = 0;
      uint cur;

      //Loop as long as destination value gets changed by other threads
      while ((cur = atomicCompSwap(NPCBuffer[index], prev, newVal))
                  != prev)
      {
            prev = cur;
            vec4 rval = convRGBA8ToVec4(cur);
            rval.xyz = rval.xyz * rval.w;       //Denormalize
            vec4 curVal = rval + val;           //Add new value
            curVal.xyz /= curVal.w;             //Renormalize
            newVal = convVec4ToRGBA8(curVal);
      }
}
```

Podobně je průměrována normála, ale místo 255 je násobena 127 a pak je přičteno 128, protože normála má rozsah -1 až 1 a ne 0 až 1 jako barva.

4.2.6 Lokální velikosti v pracovních skupinách

Při počítání *bricks* je použita lokální velikost 9^3 shader programů, aby bylo možno pracovat po 8 voxelech, kdy tento lichý počet není pro grafickou kartu vhodný, ideální jsou násobky velikosti warpu (**2.7**), ale bohužel vychází z podstaty vytváření rohově centrovaných voxelových 3^3 *bricks*. Podobně při MIP mapování *bricks* je lokální velikost 7^3 pro načtení potřebných hodnot pro vytvoření zprůměrované 3^3 *brick* vyšší úrovně odpovídající jednomu 2^3 *tile*.

Liché počty vláken také ztěžují implementaci paralelní sumy prefixů 3^3 hodnot, navržené pro počítání sumy průhlednosti při vytváření *bricks* i při jejich MIP mapování (**3.2.5** a **3.2.6**), proto je pro zjednodušení použita druhá navržená možnost – sčítání v cyklu.

Při vytváření *bricks* je použita metoda, kdy okrajová vlákna načítají více hodnot, kdežto u MIP mapování je již použita metoda, kdy jsou spouštěny dvě řady vláken navíc v každém směru, které pouze načtou hodnoty, ale dále jsou nevyužity, obě metody byly také navrženy v kapitole **3.2.5**. Tento rozdíl je způsoben chybou v původním návrh, kdy nebyly vytvářeny sdílené voxely *bricks* mezi pracovními skupinami. Tato chyba byla opravena přidáním pouze jedné řady vláken navíc, ale

ponecháním načítání na okrajových vláknech. Musela by být přidána ještě druhá řada. Což by vedlo na lokální velikost 10^3, která stále není násobkem warpu.

4.3 Spuštění a ovládání aplikace

Pro aplikaci je připravený multiplatformní CMakeList pro program CMake, pomocí kterého lze vytvořit Makefile, případně instrukce pro jiné nástroje pro překlad výsledné aplikace. Ta je pojmenovaná jako *VCT*. Spouštět ji lze z příkazové řádky s volitelnými parametry, bez nich jsou použity přednastavené hodnoty:

- -w – šířka okna aplikace (přednastaveno na 800 pixelů)
- -h – výška okna aplikace (přednastaveno na 600 pixelů)
- -f – režim celé obrazovky
- -l – počet úrovní vytvářeného voxelového stromu (přednastaveno na 8)
- --model – cesta k modelu (přednastaveno na "assets/crytek-sponza/sponza.obj").

Natáčení kamery je ovládáno pohybem myši při stisknutém levém tlačítku myši. Při stisknutém prostředním tlačítku myši dochází k přiblížení nebo oddálení. Pohyb kamery je ovládán klávesnicí. Klávesa *W* pro pohyb vpřed, *S* vzad, *D* doprava, *A* doleva, *R* výše a *F* níže.

Klávesou *G* lze zapínat a vypínat vykreslování globálního osvětlení, *B* přepíná mezi použitím *bricks* s interpolací a voxely z *node poolu* bez interpolace a *P* přepíná vykreslování s fotony nebo bez fotonů při použití *bricks*.

5 Výsledky a měření

Zde jsou prezentovány dosažené výsledky a měření paměťových nároků použitých struktur a časů provádění jednotlivých algoritmů. Implementace a pak i měření probíhalo na grafické kartě AMD Radeon HD 5850 s taktem jádra 725 MHz. Jiná karta nebyla v době vývoje k dispozici.

5.1 Dosažené výsledky

Na obrázku **5.1** je zobrazena ukázka scény s modelem Crytek Sponzy s globálním osvětlením. Je použit voxelový strom s rozlišením 256^3 a globální osvětlení je sbíráno z *brick poolu* s použitím interpolace. Pro srovnání je přiložen obrázek **5.2** bez globálního osvětlení. Hlavním rozdílem je celkově světlejší scéna při použití globálního osvětlení, což odpovídá mnohonásobnému odrážení světla ve scéně. Dále je zobrazen přenos barvy (*color bleeding*), hlavně z barevných závěsů na sloupy vedle nich a na podlahu pod nimi, ze závěsů na ochozu na sloupy ochozu a podlahu ochozu (není vidět na obrázku).

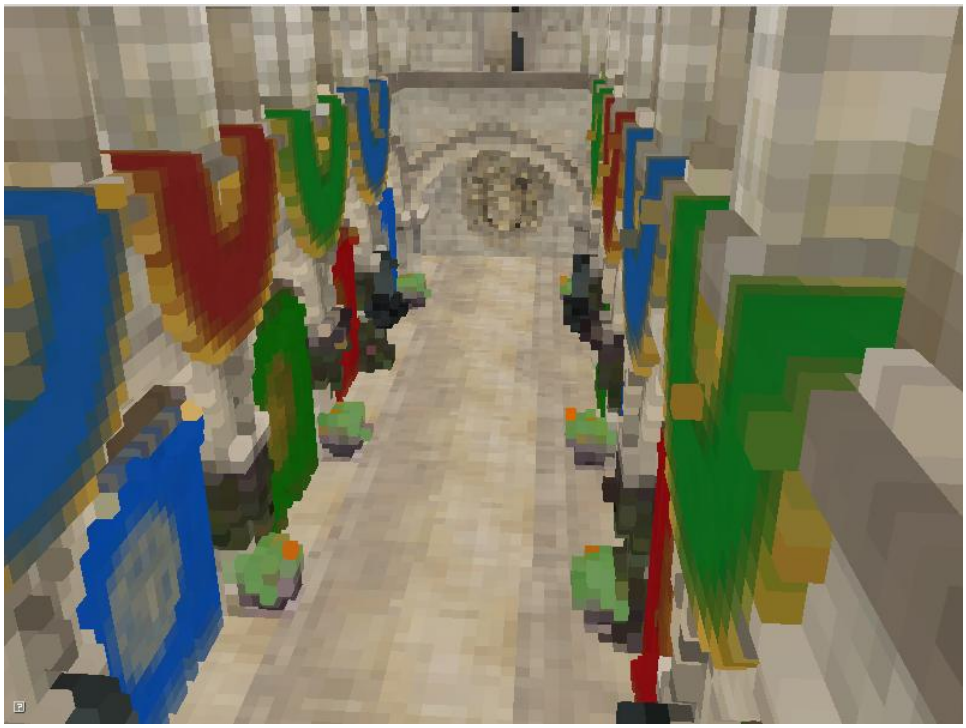
Ovšem jsou viditelné i artefakty. Přenos barvy je v některých částech čtvercový – je to způsobeno malým počtem (pět) kuželových paprsků – při použití více paprsků by byly paprsky vrhány do více směrů a došlo by k interpolaci z více *bricks* a výsledek by byl vyhlazenější. Přesto je výsledek vyhlazenější než při použití pouze *node poolu* pro barvou bez použití interpolace. Dalším artefaktem je zamlžení závěsů na ochozu vlevo, způsobené posunutím voxelů oproti trojúhelníkové

Obrázek 5.1: Scéna s globálním osvětlením. Rozlišení voxelů 256^3.

Obrázek 5.2: Scéna bez globálního osvětlení.

geometrii. Proto při sledování paprsku z fragmentu závěsu je trefen voxel stěny a ne závěsu. Pro stíny je použita shadow mapa v rozlišení $4\,096^2$, ale se stále viditelným aliasingem, ovšem stíny jsou jen vizuální doplněk a řešení jejich artefaktů nebylo součástí diplomové práce.

Na obrázku **5.3** je zobrazena ukázka zvoxelizované scény s rozlišením voxelů 512^3 s použitím konzervativní rasterizace, vykreslená pomocí ray tracingu z kamery.



Obrázek 5.3: Zvoxelizovaná scéna. Rozlišení voxelů 512^3.

5.2 Paměťové nároky

Paměťové nároky použitých struktur byly měřeny pro rozlišení voxelů 128^3, 256^3 a 512^3. Velikosti struktur byly získány vyčtením množství alokované paměti pro struktury. Souhrn velikostí je uveden v tabulce **5.1**.

3D textura není přímo použita, ale její velikost je uvedena pro srovnání mezi použitím plné 3D textury proti řídkému voxelovému stromu (SVO) – *node poolu* s *brick poolem*. Pro rozlišení 128^3 a 256^3 je součet SVO a *brick poolu* větší než velikost plné textury, kde *brick pool* má stejnou velikost jako plná textura, ale pro rozlišení 512^3 je rozdíl 2 krát 4,5 MB plus 64 MB (*node pool* pro ukazatele na potomky a pro ukazatele do *brick poolu*) proti 512 MB, což je 7 krát menší velikost.

Při prvním budování voxelového stromu je vytvářen plný strom (FVO) místo řídkého (SVO) z důvodu zjištění počtu alokovaných uzlů a celkové velikosti stromu, jak je popsáno v kapitole **3.2.3**. Plný strom je strom s alokovanými všemi možnými uzly pro všechny úrovně, tedy jeho velikost odpovídá součtu plných textur pro rozlišení všech úrovní, čemuž odpovídá jeho největší velikost v tabulce.

Při druhém budování voxelového stromu je již vytvářen SVO, jehož velikost oproti FVO je pro jednotlivá rozlišení $\frac{1}{42}$, $\frac{1}{74}$ a $\frac{1}{131}$. Zde je vidět značné ušetření video paměti. To také narůstá při použití *node poolu* pro více veličin: ukazatele na potomky, ukazatele do *brick poolu*, pro normály. Velikost SVO pro ukazatele na sousední je podle předpokladu 6 krát větší než normální SVO, protože ukládá ukazatele na 6okolí voxelu, tedy 6 krát více hodnot.

Rozlišení voxelů	**128^3**	**256^3**	**512^3**
Velikost 3D textury	8 192 kB	65 536 kB	524 288 kB
Velikost seznamu voxel fragmentů	15 354 kB	24 719 kB	51 035 kB
Velikost Full Voxel Octree (FVO)	9 362 kB	74 898 kB	599 186 kB
Velikost Sparse Voxel Octree (SVO)	224 kB	1 066 kB	4 576 kB
Velikost SVO pro ukazatele na sousední uzly	1 345 kB	6 398 kB	27 460 kB
Velikost *brick poolu*	8 192 kB	65 536 kB	65 536 kB

Tabulka 5.1: Paměťové nároky alokovaných struktur

5.3 Časy algoritmů

Měření časů vykonávání jednotlivých algoritmů bylo prováděno pomocí OpenGL *Query* objektů, které lze využít pro měření časů provádění příkazů na grafické kartě. Čas pomocí *query* je zaznamenán po dokončení všech předchozích OpenGL příkazů ve frontě, tzn., že oproti měření času pomocí CPU, nedochází k změření času po návratu příkazu před skutečným koncem vykonání příkazu na GPU. Jednotlivé časy byly získány jako průměr časů ze tří měření a jsou uvedeny v tabulce **5.2**.

Velký skok času voxelizace pro rozlišení 512^3 je nejspíš způsoben nárůstem velikosti seznamu voxel fragmentů na příliš velkou hodnotu a tak ztíženou prací s cache. První i druhé budování SVO je pro rozlišení 128^3 pomalejší než pro 256^3 a druhé budování dokonce i pro 512^3 z důvodu velkého množství atomických operací na jeden cílový voxel při průměrování hodnot ze seznamu voxel fragmentů při vkládání do SVO. Pro rozlišení 512^3 je při prvním budování vytvářen velký buffer s velkým množstvím uzlů, což je značně pomalejší než druhé budování, které je dokonce nejrychlejší oproti ostatním rozlišením z důvodu malého množství atomických operací na jeden voxel.

Časy vytváření a MIP mapování *bricks* jsou velmi vysoké kvůli velkému množství přístupů do globální paměti pro čtení a zápis a sčítání hodnot v cyklu místo paralelně a nebylo by tedy možné je využít v reálném čase v každém snímku pro dynamické aktualizace. Renderování shadow mapy v rozlišení $4\,096^2$ probíhalo pro všechna rozlišení voxelů stejně, časy jsou však z neznámých důvodů velmi rozdílné. Uložení fotonů do *brick poolu* a jejich MIP mapování je také velmi časově náročné kvůli velkému množství atomických operací pro průměrování hodnot ze shadow mapy s velkým rozlišením, což znamená nemnožnost využití v reálném čase.

Počet snímků za sekundu byl měřen nástrojem GLXOSD [34] při rozlišení obrazovky 800 krát 600 pixelů. Při pohledu kamery jako na obrázku **5.1** pro scénu bez vykreslování globálního osvětlení byl průměrný počet snímků za sekundu 83. Se zapnutým vykreslováním globálního osvětlení s rozlišením 128^3 voxelů a pěti kuželovými paprsky klesl počet snímků za sekundu na 28, pro 256^3 klesl na 19 a pro 512^3 klesl na 12. Při pohybu kamery počet snímků za sekundu kolísá, nelze jej tedy přesně změřit. Nezávisí až tak na samotném pohybu kamery, ale na části scény, která je kamerou zobrazována, tedy na počtu paprsků vrhaných z fragmentů a počtu zanoření, které paprsek provádí cestou přes voxely.

Rozlišení voxelů	**128^3**	**256^3**	**512^3**
Čas voxelizace	30 ms	30 ms	110 ms
Čas 1. budování SVO (tzn. FVO)	478 ms	207 ms	1 944 ms
Čas 2. budování SVO	454 ms	169 ms	111 ms
Čas vytváření sousedských uzlů	2 ms	10 ms	76 ms
Čas vytváření *bricks*	114 ms	2 012 ms	7 092 ms
Čas MIP mapování *bricks*	18 ms	132 ms	1 038 ms
Čas renderování shadow mapy	8 ms	253 ms	233 ms
Čas uložení fotonů do *brick poolu*	10 718 ms	9 833 ms	10 855 ms
Čas distribuce fotonů	1 ms	6 ms	31 ms
Čas MIP mapování fotonů	16 ms	123 ms	982 ms

Tabulka 5.2: Časy vykonávání jednotlivých algoritmů

6 Závěr

V této diplomové práci jsem se zabýval globálním osvětlením scény při renderování 3D grafiky. Popsal jsem efekty globálního osvětlení na scénu, jako jsou přenos barev a měkké stíny. Dále jsem popsal vybrané metody pro přesné počítání globálního osvětlení, jako jsou *path tracing*, *photon mapping* a radiozita i metody aproximační pracující v reálném čase jako jsou *Reflective Shadow Maps* a *Light Propagation Volumes*.

Dále jsem se věnoval teoretickému úvodu pro *Voxel Cone Tracing*. Popsal jsem zde metody jak jednoduché voxelizace do pravidelné 3D mřížky (textury), tak voxelizace do struktury *Sparse Voxel Octree*, která šetří paměťový prostor pro využití ve velkých scénách. K tomu jsem prezentoval algoritmus pro 3D konzervativní rasterizaci, jenž je vhodný pro konzervativní rasterizaci i velkých trojúhelníků pro správné určení 3D souřadnic při voxelizaci. Také jsem popsal techniku využití voxelové reprezentace dat pro výpočet *ambient occlusion* a globálního nepřímého osvětlení ve scéně. Nakonec jsem uvedl základy architektury grafických karet.

Pak jsem popsal návrh aplikace pro zobrazování globálního osvětlení ve scéně založené na voxelech a sledování kuželových paprsků. Návrh vychází z práce *GigaVoxels*, oproti které má určitá omezení – nezabývá se *out of core renderingem*, ale pouze globálním osvětlením, nejsou použity anizotropické voxely a voxelizace probíhá pouze pro statickou scénu bez aktualizací dynamických objektů. Dále jsem popsal strukturu aplikace: inicializaci scény, voxelizaci, budování řídkého voxelového stromu, prezentuji algoritmus vytvoření *bricks* z voxelů a jejich MIP mapování do vyšších úrovní stromu a ukládání fotonů z shadow mapy do SVO, přičemž je zanedbáván směr příchozího světla. Nakonec jsem popsal finální vykreslení scény, kdy pro jednotlivé fragmenty je počítáno globální osvětlení se sběrem informací uložených ve voxelovém stromu pomocí vysílání kuželových paprsků a interpolací hodnot z *bricks*.

Dále jsem uvedl použité knihovny a popsal vybrané implementační detaily pro práci s buffery, kódování vícerozměrného vektorového datového typu na jednorozměrný celočíselný typ, čímž je ušetřena paměť v bufferech pro SVO, práci s atomickými operacemi a jejich využití a mapování algoritmů na architekturu grafické karty. Nakonec jsem popsal spouštění a ovládání aplikace.

Jako poslední část jsem uvedl dosažené výsledky a provedená měření. Hlavním výsledkem je implementovaná aplikace zobrazující globální osvětlení ve scéně, kdy celá scéna je světlejší od mnohonásobných odrazů světla a také dochází k přenosům barvy. Výsledky nejsou vždy dostatečně hladké z důvodu malého počtu kuželových paprsků, pro jejich navýšení by byl potřeba výkonnější hardware. Nedostatkem aplikace je chybějící implementace spekulárních paprsků pro lesklé odrazy. Žádná vylepšení nakonec nebyla navržena ani implementována, kvůli celkové složitosti zadání.

Použití SVO místo plného voxelového stromu šetří paměť až 131 krát v testovací scéně, podle rozlišení voxelů. Kombinace SVO a *brick poolu* je až 7 krát menší oproti plné 3D textuře. Ovšem

časy provádění jednotlivých algoritmů jsou příliš vysoké na použití v reálném čase pro každý snímek, zvláště pak vytváření *bricks* a jejich MIP mapování do vyšších úrovní stromu a ukládání fotonů do stromu jsou v řádech jednotek sekund. Při vykreslování scény s vrháním kuželových paprsků pro sběr globálního osvětlení se počet snímků za sekundu pohybuje od 28 do 12 podle rozlišení voxelů, tedy reálně by šla tato aplikace využít pro maximální rozlišení 128^3.

Budoucí práce na projektu může zahrnovat přidání spekulárního kuželového paprsku pro odlesky, při výkonnějším hardware přidání i dalších difuzních paprsků pro větší vyhlazení vzorkovaného globálního osvětlení. Důležitou součástí by byla optimalizace kódu. Dalším rozšířením je ukládání směru příchozího světla a využití této informace spolu s normálami objektu při vykreslování globálního osvětlení, aby bylo závislé na směru pozorování. Jednoduchým rozšířením je přidání dalšího *node poolu* pro alfa kanál barvy, na což je už aplikace připravena, aby bylo možno pracovat s průhlednými objekty ve scéně. Dalším možným rozšířením je přidání systému časových značek pro dynamické aktualizace stromu pro pohyblivé objekty.

Literatura

[1] CRASSIN, Cyril a Simon GREEN. Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer. COZZI, Patrick a Christophe RICCIO. *OpenGL insights*. Boca Raton, FL: CRC Press, c2012, s. 303--318. ISBN 9781439893760. Dostupné z: http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-SparseVoxelization.pdf

[2] CRASSIN, Cyril. *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. 2011. Dostupné z: http://maverick.inria.fr/Publications/2011/Cra11/CCrassinThesis_EN_Web.pdf. Dizertační práce. UNIVERSITE DE GRENOBLE.

[3] CRASSIN, Cyril, Fabrice NEYRET, Miguel SAINZ, Simon GREEN a Elmar EISEMANN. Interactive Indirect Illumination Using Voxel Cone Tracing. *Computer Graphics Forum*. 2011, vol. 30, issue 7, s. 1921-1930. DOI: 10.1111/j.1467-8659.2011.02063.x. Dostupné z: http://maverick.inria.fr/Publications/2011/CNSGE11b/GIVoxels-pg2011-authors.pdf

[4] HASSELGREN, Jon, Tomas AKENINE-MÖLLER a Lennart OHLSSON. Conservative Rasterization. *GPU Gems 2*. 2005, č. 2. Dostupné z: https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter42.html

[5] DUTRÉ, Philip, Kavita BALA a Philippe BEKAERT. *Advanced global illumination*. 2nd ed. Wellesley, Mass.: AK Peters, c2006. ISBN 9781568813073.

[6] HENRIK WANN JENSEN. *Realistic image synthesis using photon mapping*. First paperback print. Natick. MA: A.K. Peters, Ltd, 2009. ISBN 9781568814629.

[7] JENSEN, Henrik Wann a Per CHRISTENSEN. High quality rendering using ray tracing and photon mapping. In: *ACM SIGGRAPH 2007 courses on - SIGGRAPH '07* [online]. New York, New York, USA: ACM Press, 2007, [cit. 2016-05-06]. DOI: 10.1145/1281500.1281593. ISBN 9781450318235. Dostupné z: http://dl.acm.org/citation.cfm?doid=1281500.1281593

[8] KAJIYA, James T. The rendering equation. In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques - SIGGRAPH '86* [online]. New York, New York, USA: ACM Press, 1986, s. 143-150 [cit. 2016-05-06]. DOI: 10.1145/15922.15902. ISBN 0897911962. Dostupné z: http://portal.acm.org/citation.cfm?doid=15922.15902

[9] GORAL, Cindy M., Kenneth E. TORRANCE, Donald P. GREENBERG a Bennett BATTAILE. Modeling the interaction of light between diffuse surfaces. In: *Proceedings of the 11th annual conference on Computer graphics and interactive techniques - SIGGRAPH '84* [online]. New York, New York, USA: ACM Press, 1984, s. 213-222 [cit.

2016-05-08]. DOI: 10.1145/800031.808601. ISBN 0897911385. Dostupné z: http://portal.acm.org/citation.cfm?doid=800031.808601

[10] TABELLION, Eric a Arnauld LAMORLETTE. An approximate global illumination system for computer generated films. In: *ACM SIGGRAPH 2004 Papers on - SIGGRAPH '04* [online]. New York, New York, USA: ACM Press, 2004, s. 469-476 [cit. 2016-05-07]. DOI: 10.1145/1186562.1015748. Dostupné z: http://portal.acm.org/citation.cfm?doid=1186562.1015748

[11] CHRISTENSEN, Per H. a Dana BATALI. An irradiance atlas for global illumination in complex production scenes. *Eurographics Workshop on Rendering* [online]. Eurographics Association Aire-la-Ville, 2004, s. 133-141 [cit. 2016-05-07]. DOI: 10.2312/EGWR/EGSR04/133-141. ISSN 1727-3463.

[12] LEHTINEN, Jaakko, Matthias ZWICKER, Emmanuel TURQUIN, Janne KONTKANEN, Frédo DURAND, François X. SILLION a Timo AILA. A meshless hierarchical representation for light transport. *ACM Transactions on Graphics* [online]. 2008, **27**(3), s. 1-9 [cit. 2016-05-07]. DOI: 10.1145/1360612.1360636. ISSN 07300301. Dostupné z: http://portal.acm.org/citation.cfm?doid=1360612.1360636

[13] DACHSBACHER, Carsten, Marc STAMMINGER, George DRETTAKIS a Frédo DURAND. Implicit visibility and antiradiance for interactive global illumination. *ACM Transactions on Graphics (SIGGRAPH Conference Proceedings)* [online]. 2007, **26**(3) [cit. 2016-05-07].

[14] DONG, Zhao, Jan KAUTZ, Christian THEOBALT a Hans-Peter SEIDEL. Interactive Global Illumination Using Implicit Visibility. In: *15th Pacific Conference on Computer Graphics and Applications (PG'07)* [online]. IEEE, 2007, s. 77-86 [cit. 2016-05-07]. DOI: 10.1109/PG.2007.37. ISBN 0-7695-3009-5. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4392718

[15] KELLER, Alexander. Instant radiosity. In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques - SIGGRAPH '97* [online]. New York, New York, USA: ACM Press, 1997, s. 49-56 [cit. 2016-05-07]. DOI: 10.1145/258734.258769. ISBN 0897918967. Dostupné z: http://portal.acm.org/citation.cfm?doid=258734.258769

[16] LAINE, Samuli, Hannu SARANSAARI, Janne KONTKANEN, Jaakko LEHTINEN a Timo AILA. An irradiance atlas for global illumination in complex production scenes. *Rendering Techniques* [online]. Eurographics Association Aire-la-Ville, 2007, s. 277-286 [cit. 2016-05-07]. DOI: 10.2312/EGWR/EGSR07/277-286.

[17] WALTER, Bruce, Sebastian FERNANDEZ, Adam ARBREE, Kavita BALA, Michael DONIKIAN a Donald P. GREENBERG. Lightcuts. In: *ACM SIGGRAPH 2005 Papers on - SIGGRAPH '05* [online]. New York, New York, USA: ACM Press, 2005, s. 1098-

1107 [cit. 2016-05-07]. DOI: 10.1145/1186822.1073318. Dostupné z: http://portal.acm.org/citation.cfm?doid=1186822.1073318

[18] HAŠAN, Miloš, Fabio PELLACINI a Kavita BALA. Matrix row-column sampling for the many-light problem. In: *ACM SIGGRAPH 2007 papers on - SIGGRAPH '07* [online]. New York, New York, USA: ACM Press, 2007, [cit. 2016-05-07]. DOI: 10.1145/1275808.1276410. ISBN 9781595936486. Dostupné z: http://portal.acm.org/citation.cfm?doid=1275808.1276410

[19] DACHSBACHER, Carsten a Marc STAMMINGER. Reflective shadow maps. In: *Proceedings of the 2005 symposium on Interactive 3D graphics and games - SI3D '05* [online]. New York, New York, USA: ACM Press, 2005, s. 203-231 [cit. 2016-05-09]. DOI: 10.1145/1053427.1053460. ISBN 1595930132. Dostupné z: http://portal.acm.org/citation.cfm?doid=1053427.1053460

[20] RITSCHEL, Tobias, Thorsten GROSCH, Min H. KIM, Hans-Peter SEIDEL, Carsten DACHSBACHER a Jan KAUTZ. Imperfect Shadow Maps for Efficient Computation of Indirect Illumination. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia 2008)* [online]. 2008, **27**(5), s. 129-137 [cit. 2016-05-07].

[21] RITSCHEL, T., T. ENGELHARDT, T. GROSCH, H.-P. SEIDEL, J. KAUTZ a C. DACHSBACHER. Micro-rendering for scalable, parallel final gathering. In: *ACM SIGGRAPH Asia 2009 papers on - SIGGRAPH Asia '09* [online]. New York, New York, USA: ACM Press, 2009, [cit. 2016-05-07]. DOI: 10.1145/1661412.1618478. ISBN 9781605588582. Dostupné z: http://portal.acm.org/citation.cfm?doid=1661412.1618478

[22] NICHOLS, Greg, Jeremy SHOPF a Chris WYMAN. Hierarchical Image-Space Radiosity for Interactive Global Illumination. *Computer Graphics Forum* [online]. 2009, **28**(4), 1141-1149 [cit. 2016-05-07]. DOI: 10.1111/j.1467-8659.2009.01491.x. ISSN 01677055. Dostupné z: http://doi.wiley.com/10.1111/j.1467-8659.2009.01491.x

[23] KAPLANYAN, Anton a Carsten DACHSBACHER. Cascaded light propagation volumes for real-time indirect illumination. In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D 10* [online]. New York, New York, USA: ACM Press, 2010, s. 99-107 [cit. 2016-05-07]. DOI: 10.1145/1730804.1730821. ISBN 9781605589398. Dostupné z: http://dl.acm.org/citation.cfm?doid=1730804.1730821

[24] *NVIDIA's Next Generation CUDA TM Compute Architecture: Fermi* [online]. NVIDIA Corporation, 2009 [cit. 2016-05-16]. Dostupné z: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[25] *NVIDIA's Next Generation CUDA TM Compute Architecture: Kepler TM GK110 /210* [online]. NVIDIA Corporation, 2014 [cit. 2016-05-16]. Dostupné z:

http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

[26] CUDA C Programming Guide. *NVIDIA Documentation* [online]. NVIDIA Corporation, 2015 [cit. 2016-05-16]. Dostupné z: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[27] *OpenGL 4.5 API Reference Card* [online]. Khronos Group, 2014 [cit. 2016-05-16]. Dostupné z: https://www.opengl.org/sdk/docs/reference_card/opengl45-reference-card.pdf

[28] *OpenGL Mathematics* [online]. [cit. 2016-05-22]. Dostupné z: http://glm.g-truc.net/

[29] *SDL: Simple DirectMedia Layer* [online]. [cit. 2016-05-22]. Dostupné z: https://www.libsdl.org/

[30] *AntTweakBar* [online]. [cit. 2016-05-22]. Dostupné z: http://anttweakbar.sourceforge.net/doc/

[31] *DevIL: A full featured cross-platform Image Library* [online]. [cit. 2016-05-22]. Dostupné z: http://openil.sourceforge.net/

[32] *Assimp: Open Asset Import Library* [online]. [cit. 2016-05-22]. Dostupné z: http://www.assimp.org/

[33] MCGUIRE, Morgan. Computer Graphics Archive: Meshes. *Computational Graphics at Williams College* [online]. [cit. 2016-05-22]. Dostupné z: http://graphics.cs.williams.edu/data/meshes.xml

[34] GULETSKII, Nick. *GLXOSD: An OSD and benchmarking tool for Linux* [online]. 2016 [cit. 2016-05-22]. Dostupné z: https://glxosd.nickguletskii.com/