VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV PČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

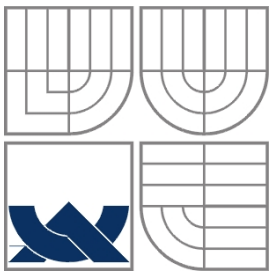
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

REALISTICKÉ ZOBRAZOVÁNÍ POMOCÍ RADIOZITY

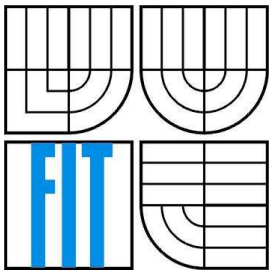
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. JIŘÍ JANOUŠEK
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

REALISTICKÉ ZOBRAZOVÁNÍ POMOCÍ RADIOZITY

REALISTIC RENDERING USING RADIOSITY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ JANOUŠEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ADAM HEROUT, Ph.D.

BRNO 2007

Abstrakt

Tato práce se zabývá jednou z globálních zobrazovacích metod – radiozitou. Je zde zpracována podstata metody, uvedeny základní rovnice a vztahy. Dále je zde přehled možných řešení této metody v praxi a popis konkrétní implementace založené na využití grafické karty a OpenGL. Práce si klade za cíl popsat stručně, jasně a srozumitelně základní problémy a přístupy při implementaci radiozity.

Klíčová slova

Globální zobrazovací metody, Radiozita, Implementace Radiozity

Abstract

This thesis deals with one of the global illumination algorithms – the radiosity algorithm. There are handled fundamentals of the radiosity algorithm including basic equations and relations. You can see review of possible solutions of this method in use and description of the implementation based on video card and OpenGL rendering. The thesis tries to explain main problems and attitudes to implementation of radiosity briefly, clearly and comprehensibly.

Keywords

Global illumination algorithms, Radiosity, Implementation of Radiosity

Citace

Jiří Janoušek: Realistické zobrazování pomocí radiozity, diplomová práce,
Brno, FIT VUT v Brně, 2007

Realistické zobrazování pomocí radiozity

Prohlášení

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

……..
Jiří Janoušek
květen 2007

Poděkování

Rád bych poděkoval svému ochotnému vedoucímu Adamu Heroutovi, který mne vždy nasměroval správným směrem.

Obsah

1 Úvod

Tato práce se zabývá jednou z hlavních globálních zobrazovacích metod – radiozitou. Globální zobrazovací metody (na rozdíl od „lokálních“ metod) vyhodnocují osvětlení daného tělesa (útvaru na jeho povrchu) ve vztahu k osvětlení a záření jiných těles v celé scéně. To je důvodem, proč je v podstatě nemožné modelovat uspokojivě lokálními osvětlovacími modely např. polostín, odraz světla od ploch a tím zabarvení (nádech) sousedící plochy apod.

Globální zobrazovací metody se snaží (více či méně úspěšně) simulovat reálné šíření světelné energie scénou a díky tomu napodobit reálné optické jevy. Existují různé metody – např. Photon tracing, který simuluje přímo pohyb hypotetických světelných částic-kvant (fotonů) apod. My se zde budeme zabývat radiozitou, která je založena na myšlence, že scéna je energeticky uzavřená a tudíž lze docílit ustáleného stavu. V tomto stavu lze vypočítat, jak je která část (např. ploška nějakého objektu) ovlivňována svým okolím a zároveň jak sama ovlivňuje své okolí. Globální metody vtšinou vedou již k tak kvalitním zobrazením scény, že je pro laika obtížné výsledný obrázek odlišit od fotografie. K takovému výsledku se většinou dospěje kombinací několika zobrazovacích metod – příkladem může být časté předpočítání scény pomocí radiozity a zobrazení pomocí ray-tracingu, který je schopen přidat fyzikální efekty, které není schopna zachytit ze své podstaty radiozita.

V první části práce uvedeme základní myšlenky a principy metody radiozita. Budeme se zabývat taktéž myšlenkami globálních zobrazovacích metod všeobecně, neboť právě z nich radiozita vychází. V další části se budeme snažit nastínit možná řešení a problémy vyplývající z podstaty metody. Následovat bude podrobný popis implementace radiozity založený na využití neustále se zvyšujícího výkonu grafických akcelerátorů a OpenGL. Budou zde popsány problémy, na které narazíme a jejich řešení. V přílohách a na přiloženém CD-ROM jsou pak k dispozici kompletní zdrojové kódy, které zde budou popsány, a také již vypočtené ukázky osvětlených scén.

2 Radiozita obecně

Jeden z prvních pokusů o fyzikální simulaci šíření světla scénou podnikli Goralová, Torrance, Greenberg a další v polovině osmdesátých let. Jimi navržená metoda radiozity aplikuje poznatky z oblasti výpočtů tepelného záření na problém výpočtu světelného záření. Základní radiozitní algoritmus předpokládá energeticky uzavřenou scénu a neuvažuje interakci světelné energie s prostředím („vzduchem“) ve scéně. Předpokládá se difúzní odraz světla na ploškově reprezentovaných objektech.

Postup při zobrazování scény metodou radiozity lze rozdělit na dvě části. V první části se zabýváme přenosem energie (světla) ve scéně mezi jednotlivými ploškami. Tím docílíme kompletního popisu scény z hlediska osvětlení. Ve druhé části je tedy možné použít jakýkoli zobrazovací algritmus řešící problém viditelnosti a scénu zobrazit. Takovým algoritmem může být např. již výše zmíněný ray-tracing.

V metodě radiozity jsou tedy zavedeny dva základní pojmy – radiozita B a odrazivost ρ.

Radiozita B – je světelný výkon vyzářený v určitém bodě vztažený na jednotkovou plochu. Hodnoty radiozity jsou vždy kladné. Velikost není nijak omezena (záleží na osvětlení scény) a číslo nemá žádný fyzikální rozměr (jednotku).

Odrazivost ρ – je difúzní odrazivost plošky v daném bodě. Udává nám, kolik přijaté energie ploška v daném bodě pohltí a kolik odrazí zpět do prostoru. Hodnoty jsou omezeny na interval od 0 do 1, přičemž 0 znamená, že povrch veškerou dopadlou enerigii pohlcuje a 1 zase, že povrch veškerou dopadlou energii odráží.

2.1 Radiozitní rovnice

Vezmeme-li v úvahu, že všechny povrchy odrážejí světlo pouze difúzně, zjednodušíme tím reálnou základní zobrazovací rovnici prostředí na tzv. *radiozitní rovnici*:

Rovnice 2-1

$$B(x) = E(x) + \rho(x)\int_S B(x')G(x,x')dx'$$

Tato rovnice vyjadřuje, že radiozita $B(x)$ vyzařovaná povrchem v bodě x je součtem vlastní vyzářené radiozity $E(x)$ v tomto bodě a radiozity dopadlé v bodě x na povrch a odražené zpět do scény. Geometrický člen $G(x,x')$ zahrnuje geometrické informace týkající se vždy dvojice povrchů: jejich vzájemnou viditelnost, odchylky od normál obou povrchů a vzdálenost obou bodů. S značí množinu všech ploch scény.

Bohužel analytické řešení této rovnice je možné (a to netriviální) pouze pro velmi jednoduché scény. Ve složitějších scénách je to problém neřešitelný. Nahradíme-li ovšem povrch celé scény

aproximací složenou z jednotlivých rovinných plošek a budeme-li předpokládat, že hodnota radiozity každé plošky je na celém jejím povrchu konstantní, můžeme radiozitní rovnici přeformulovat na její diskrétní verzi:

Rovnice 2-2

$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_j F_{ij}$$

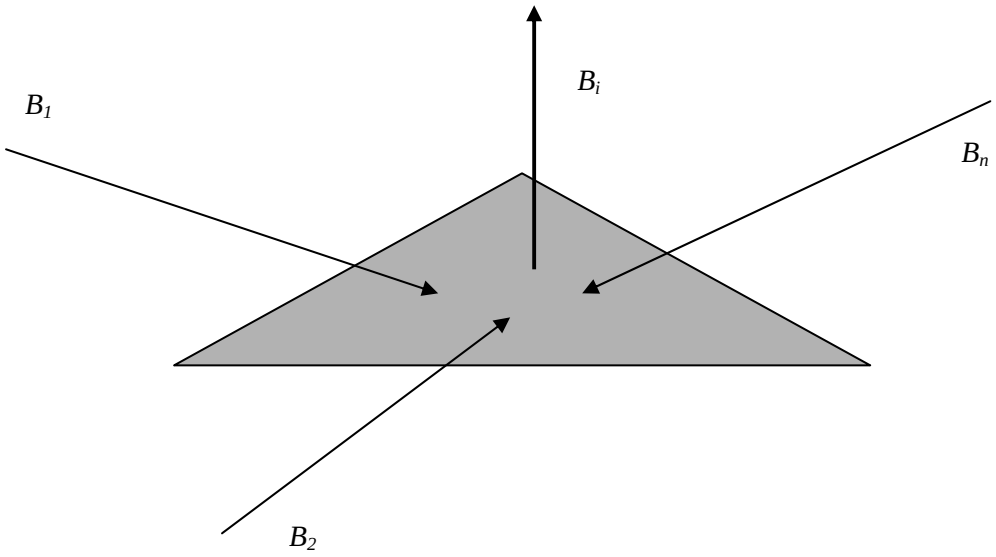
F_{ij} značí tzv. *konfigurační faktor*:

Konfigurační faktor F_{ij} – anglicky *form factor* – v diskrétním tvaru radiozitní rovnice nahrazuje geometrický člen $G(x,x')$. Určuje vzájemnou viditelnost dvojice plošek – udává, kolik z celkové energie vyzářené ploškou *j* je přijato ploškou *i*. Jeho hodnotu lze spočítat jako plošný průměr diferenciálních geometrických členů mezi všemi body obou testovaných plošek:

Rovnice 2-3

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} G(x, x') dA_i dA'_j$$

Diskrétní radiozitní rovnice umožňuje vyjádřit radiozitu *i*-té plošky scény pomocí radiozit všech ostatních plošek (viz. obrázek 2.1). Pro každou plochu scény dostáváme jednu rovnici toho typu, což vede na soustavu rovnic, jejíž řešení určuje množství vyzařovaného světla pro každou plošku ve scéně.

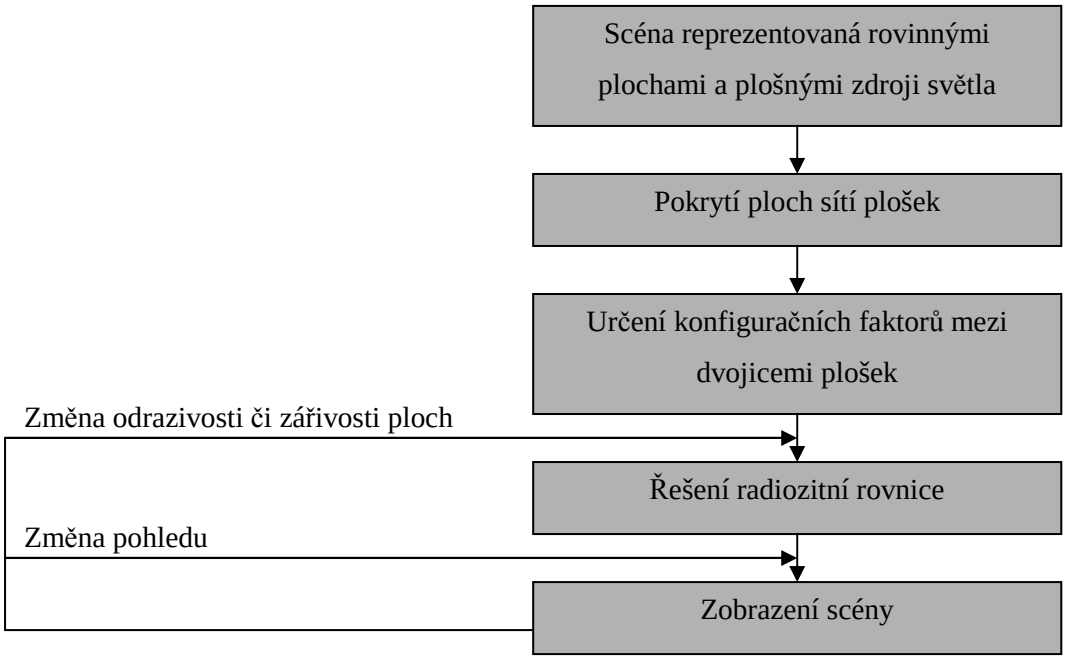


Obrázek 2.1 – radiozita přijatá a vyzářená ploškou *i*

2.2 Řešení radiozity

Řešení osvětlení metodou radiozity spočívá v několika na sobě nezávislých krocích – dá se tedy dobře segmentovat a rozdělit na několik dílčích úloh. Výhoda takového přístupu je v tom, že pokud se

změní některá část (její konfigurace, či model), není potřeba spočítat všechno od začátku, ale pouze části, které přímo používají data dotčené oblasti na svém vstupu. Tuto myšlenku rozdělelní výpočtu si můžeme podrobněji prohlédnout na obrázku 2.2.



Obrázek 2.2 – Schéma dělení radiozity na podúlohy

V další části této kapitoly se budeme zabývat jednotlivými podúlohami při řešení radiozity podrobněji.

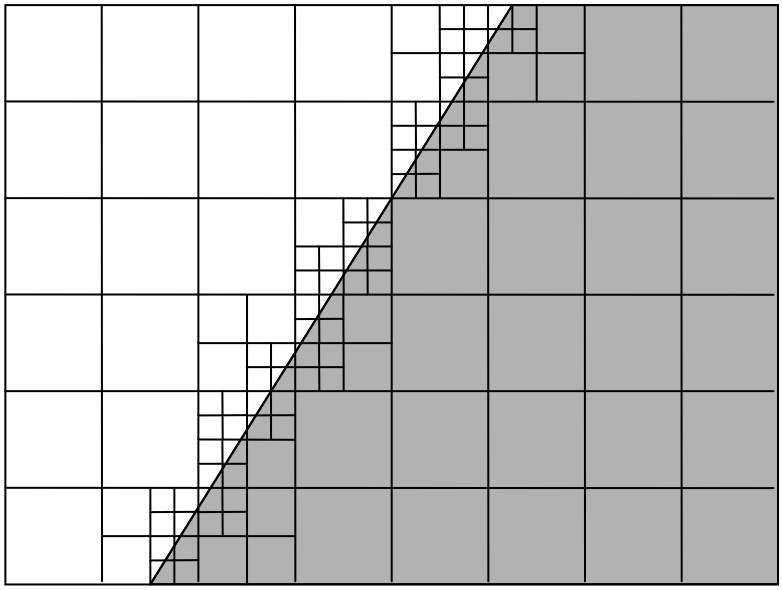
2.2.1 Vstupní model scény

Algoritmus předpokládá na vstupu co možná nejjednodušší (intuitivní) popis scény. Rovinné plochy jsou tedy popsány jedním útvarem apod. To je však pro náš záměr zcela nevhodné, neboť my budeme chtít, aby celá jedna každá plocha měla stejné vlastnosti. Nevypadalo by tedy příliš realisticky, kdyby měla celá stěna jednu barvu (míru osvětlení). Snahou tedy je rozdělit plochy ve scéně na malé plošky – a to co nejlépe se záměrem na výsledek. Existují dva základní způsoby dělení:

Uniformní metody – rozdělují povrchy ve scéně na malé pravidelné plošky (trojúhelníky nebo čtverce – každý čtverec je ale možno reprezentovat opět dvěma trojúhelníky) rovnoměrně a pravidelně. Hlavními klady tohoto přístupu je jednoduchost algoritmu, rychlost a dobrá implementovatelnost v hardwaru. Bohužel platíme za to zbytečně podrobným dělením ploch v místech, kde to není potřeba (jednolitě ozářené plochy) a naopak velmi hrubým a nedostatečným dělením v místech, kde se rychle střídají různé intenzity osvětlení (ostré stíny – rozhraní světlo/stín).

Neuniformní metody – tyto metody se snaží poučit z metod předchozí kategorie. Snahou je zohlednit geometrii scény a polohu světelných zdrojů. V praxi to tedy znamená, že při vyhodnocování

polohy plochy si algoritmus spočítá, zda daná plocha ve stínu, na rozhraní či osvětlená. Pokud bude v daném místě pravděpodobnost přechodu světlo/stín či naopak vysoká, místo bude jemně poseto ploškami, kdežto v opačném případě bude pouze aproximováno ploškou mnohem větší. Tato skupina přístupů je mnohem náročnější na výpočet, avšak dává podstatně lepší výsledky s porovnatelným výsledným počtem plošek ve scéně. Mezi tyto neuniformní metody se řadí i metoda *adaptivního dělení plošek*, kdy dochází ke zjemňování povrchu až při postupném výpočtu osvětlení. Pro reprezentaci rozdělených ploch se používají např. kvadrantové stromy. Příklad takto rozdělené plochy si můžeme prohlédnout na obrázku 2.3.



Obrázek 2.3 – Adaptivní dělení plošek dle hranice stínu

2.2.2 Určení konfiguračních faktorů

Tato část celého výpočtu je nejnáročnější a hraje v celé radiozitě klíčovou roli. Velikost konfiguračního faktoru je ovlivněna následujícími třemi věcmi:

Velikostí plošek – čím bude zářící ploška *j* menší, tím bude menší i konfigurační faktor F_{ij}. Stejně tak klesá velikost konfiguračního faktoru se čtvercem vzdálenosti obou plošek.

Vzájemnou polohou plošek – pokud bude zářící ploška *j* natočena „z profilu“, bude výsledný konfigurační faktor menší, než v případě, kdy je tato ploška natočena „čelem“.

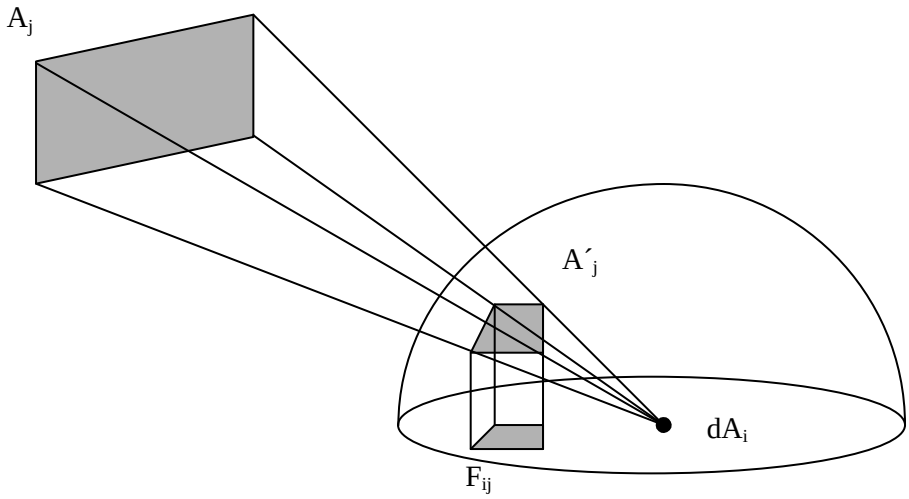
Vzájemnou viditelností plošek – pokud budou mezi ploškami ležet stínící plošky či objekty, bude konfigurační faktor menší, než v případě úplné vzájemné viditelnosti.

Z tohoto výčtu je jasně patrné, že konfigurační faktory závisejí pouze na geometrii scény. Tedy pokud se změní jenom světelné podmínky (intenzita světla vyzařovaného světelnými zdroji), je možné podstatnou část řešení urychlit použitím již jednou spočtených konfiguračních faktorů.

Tak jako v předchozí kapitole existovalo více kategorií k řešení daného problému, tak také zde, při výpočtu konfiguračních faktorů, na to můžeme jít různě. V zásadě potřebujeme vyřešit rovnici 2.3:

Analytické metody – použitelné pouze u nejjednodušších případů vzájemné polohy a tvaru ploch, navíc musejí být zcela vzájemně viditelné. Díky těmto velmi omezujícím podmínkám se tyto metody v praxi nevyužívají.

Projekční metody – jsou nejstarší a nejznámější používané aproximační metody výpočtu konfiguračních faktorů. Původně se používalo tzv. *Nusseltovy analogie,* kde se ploška *j* promítala na jednotkovou polokouli obklopující plošku *i*. Z tohoto průmětu se následně promítala do roviny plošky *i*. Vše je patrné z brázku 2.4.



Obrázek 2.4 – Nusseltova analogie

Jenže tento postup (dvojí promítání plošky) je stále výpočetně dost náročný. Polokoule se tedy nahrazuje polokrychlí. Ta je pokryta sítí malých malých čtverečků a každý z nich představuje tzv. *delta faktor* - což je vlastně vztah k původní polokouli. Konfigurační faktor je pak určen součtem delta faktorů těchto malých čtevrečků, kterých se dotkl při projekci na polokrychli. Problémem může být aliasing (síť čtverečků je přece jenom rastr), neboť bude-li průmět plošky menší, než jeden čtvereček, vůbec se neprojeví. Pokud je tato zanedbaná ploška významným zdrojem světla, bude odchylka od správné hodnoty osvětlení výrazná.

Metody vrhání paprsku – vycházejí ze stochastických numerických metod pro výpočet integrálů. Jejich princip je založen na náhodném vystřelování paprsků do okolí plošky pro zjištění viditelných ploch a odhad konfiguračních faktorů. Výhodou metod vrhání paprsků je možnost jejich aplikace i na složité scény. Problémem může být výpočet průsečíků paprsků se scénou a relativně velký počet paprsků nutný pro dosažení kvalitního výsledku. V praxi se však přesto ukazují tyto metody jako efektivní.

2.2.3 Řešení vlastní radiozity

V úvodu této kapitoly jsme si nastínili vztahy mezi jednotlivými ploškami a popsali je rovnicí. Tato rovnice je však sama o sobě analyticky neřešitelná, tudíž jsme po jistém zjednodušení převedli tuto rovnici na její diskrétní variantu. Pokud ji dále upravíme tak, že sdružíme neznámé radiozity na levé straně rovnice a známé počáteční emisní hodnoty na pravé straně, dostáváme rovnici v následujícím tvaru:

Rovnice 2-4

$$B_i - \rho_i \sum_{j=1}^{n} B_j F_{ij} = E_i$$

Vyjádříme-li si tuto jednu rovnici jako *n* rovnic pro každou neznámou B_1 až B_n, dostaneme soustavu lineárních rovnic. Vyjádříme-li tuto soustavu maticově, dostaneme krásně jednoduchou rovnici ve tvaru:

Rovnice 2-5

$$KB = E$$

Kde *K* je matice konfiguračních faktorů, *B* je vektor radiozit a *E* je vektor emisních hodnot pro každou z plošek ve scéně.

Existuje několik metod, jak tuto rovnici vyřešit – jsou to běžné metody známé z numerické matematiky. Problém však je, že pokud je scéna rozsáhlejší, trvá těmto metodám velmi dlouho, než soustavu vyřeší (není problém mít miliony rovnic, běžně několik set tisíc). K dispozici je tak až celkové, zato kompletní řešení. My bychom však dali přednost metodě, kde bychom mohli znát různé mezivýsledky a mohli tak případně výpočet zastavit, pokud bychom například při testování odhalili nedostatek, či pokud by další zpřesňování již nemělo velký vliv na výslednou kvalitu vzhledu scény. O takovýto způsob řešení se pokuslili už Cohen, Wallace a Greenberg. Metoda je známá pod názvem *Progressive refinement* nebo také *Progresivní radiozita*. Existují však i další metody a mají různé vlastnosti. Následující podkapitolky nabídnou tedy podrobnější přehled jednotlivých metod.

2.2.3.1 Progresivní radiozita

Tento přístup jde zcela proti původnímu úmyslu shromažďování radiozity na jednotlivých ploškách. Naopak se snaží vždycky najít plošku, která svou emisí nejvíce ovlivní osvětlení ve scéně a z ní „vystřelit“ radiozitu do všech plošek, které jsou v jejím dosahu. Ty se dále stávají sekundárními zdroji světla a opět se z nich vybere ta, která má největší „světelný potenciál“ a stává se „vystřelující“ ploškou. Celý algoritmus je rozepsán zde:

1. Všem ploškám přiřaď $B_i = E_i$ a $\Delta B_i = E_i$
2. Dokud výpočet nezkonvergoval, opakuj:
    1. Vyber plošku *i* s největší hodnotou $B_i A_i$

2. Spočti konfigurační faktory F_{ji} pro ostatní plochy j
3. Přiřaď všem plochám j nové radiozity spočítané jako

$$B_j = B_j + \Delta B_i \rho_j F_{ji}$$

$$\Delta B_j = \Delta B_j + \Delta B_i \rho_j F_{ji}$$

4. Přiřaď $\Delta B_i = 0$

3. Zobraz výsledek podle aktuálních hodnot v matici **B**.

Tato metoda dovoluje v každém iteračním kroku sledovat stav výpočtu. Dále je zde zajištěno, že konverguje jak nejrychleji je to možné. Důvodem je fakt, že na výsledný stav scény mají největší vliv plošky, které vyzáří nějvíce energie – a přesně takové plošky jsou v každém iteračním kroku uvažovány. V numerických metodách zmíněných výše máme pouze jistotu, že budou do výpočtu zahrnuty, nevíme však v kterém iteračním kroku se tak stane.

2.2.3.2 Hierarchická radiozita

Tato kategorie metod pro řešení radiozity je založena na myšlence uspořádání ploch ve scéně do hierarchie. V takové hierarchii pak určujeme vzájemný vliv jednotlivých plošek apod. Hierarchická radiozita má velkou výhodu oproti předešlým způsobům v tom, že dokáže velmi dobře pracovat s modelem scény během výpočtu a zpřesňovat jej dle potřeb.

Hierarchie je budována většinou několikaúrovňovými kvadrantovými stromy. Dle postupu, kterým takové stromy budujeme, můžeme zvolit metodu dělení nebo naopak seskupování plošek. První zmíněná využívá přístupu, kdy se začne s hrubou scénou a dle potřeb se dané oblasti dělí na menší a menší plošky. Zároveň se využívá faktu, že velmi malé plošky blízko sebe mají většinou stejný konfigurační faktor ve vztahu k velkým plochám a proto se konfigurační faktory snažíme počítat vždy hromadně pro více plošek najednou. U tohoto přístupu budujeme hierarchii plošek shora – začínáme s velmi hrubou scénou – kořen a pár uzlů. Postupně dělíme plochy na menší a menší a tím budujeme hierarchii. Nevýhodou této metody je, že konfiguračních faktorů, které musíme vypočítat nám v každém kroku neustále přibývá. Druhá varianta této metody postupuje zcela obráceně. Je vhodnější většinou pro hodně podrobné scény (kvalitní modely). Snaží se seskupovat malé plošky do shluků a počítat konfigurační faktory mezi těmito shluky. Vytvářejí se tak vazby a buduje hierarchie.

2.2.3.3 Některé další metody

Mezi další metody můžeme zařadit například **vlnkovou radiozitu**. Ta vychází z poměrně jednoduché myšlenky. Čím více plošek, tím jemnější je výsledný obraz, ale tím řádově náročnější výpočet, zejména rostoucím počtem konfiguračních faktorů. I když rozdělíme scénu na malé plošky a udržíme počet konfiguračních faktorů na únosné mezi, pořád ještě můžeme pozorovat artefakty jako jsou zubaté stíny apod. Zcela tento problém neřeší ani Goraudova interpolace na jednotlivých ploškách.

Základní myšlenkou této metody je vyjádřit hezké stíny nikoli zpřesněním modelu, ale zpřesněním popisu jednotlivých plošek. Metoda se snaží popsat radianci na dané plošce lineární kombinací funkcí místo jedné konstantní hodnoty. Experimenty prokázaly, že i při poměrně hrubě dělené scéně je tato metoda schopná podat poměrně velmi kvalitní výsledky.

Další metoda uvažuje jiný problém. Radiozita řeší osvětlení modelu nezávisle na pohledu do scény. Metoda **vlivu důležitosti** se snaží během výpočtu počítat na základě úhlu pohledu do scény tzv. *vliv důležitosti* (*importance*) dané plošky. V místech, která nejsou moc vidět, si můžeme dovolit mnohem hrubší výpočet, než v místech, která jsou uprostřed záběru. Výhodou tohoto přístupu je výrazné urychlení výpočtu, avšak přicházíme o jednu z havních výhod radiozity – a to nezávislost na úhlu pohledu do scény. Při změně pohledu je tedy potřeba osvětlení přepočítat.

2.2.3.4 Stochastické metody řešení

Všechny výše uvedené metody měly společného jmenovatele ve formě konfiguračních faktorů. Jejich výpočet je pro numerické metody nutný, složitý (především časově) a prostorově náročný (pro n plošek máme n^2 konfiguračních faktorů – matici – á nejméně 4 bajty na faktor, což je např. pro 100 000 plošek 40GB místa!).

Elegantním řešením tohoto problému můžou být stochastické metody. Jako typického zástupce můžeme uvést např. metodu **Monte Carlo**. Tato metoda může pro určení konfiguračního faktoru dvou plošek využít např. metodu vystřelování paprsků (zmíněna výše), kdy konfigurační faktor F_{ij} je spočten jako podíl paprsků dopadlých na plochu j ku náhodně vystřeleným paprskům z plochy i. Takto získaný konfigurační faktor je možno po použití k výpočtu zapomenout a jít na další plošky – žádnou matici konfiguračních faktorů tedy nebudujeme.

3 Implementace

3.1 Koncepce

V předchozí kapitole jsme si představili různé metody řešení radiozity a nastínili jsme taktéž jejich klady a zápory vycházející z jejich podstaty. Nyní se přesuneme od teorie k praxi a v další části této práce, která se bude zaobírat konkrétním řešením radiozity, se budeme snažit popsané principy implementovat.

Zamyslíme-li se nad tím nejzákladnějším principem zjednodušeného modelu, tak jak jej radiozita představuje, dospějeme k názoru, že vlastně jediným problémem je určit, kolik světla dopadne na tu kterou plošku. K tomu vedou různé cesty popsané výše (konfigurační faktory, …), ale co kdybychom si představili tento proces pouze jako pohled z místa plošky do scény a tímto pohledem určili kolik na ni dopadlo světla?

Zde máme několik možností, jak takový pohled vyrenderovat. My zvolíme pro jednoduchost OpenGL, čímž vyřešíme několik problémů naráz. OpenGL za nás bude kompletně řešit viditelnost jednotlivých plošek, vzdálenost plošek od naší plošky a po mírné úpravě interpretace barev také intenzitu světla. Plošky budeme reprezentovat trojúhelníky. Pro pohled kamery bude však důležité zvolit nějaký bod a ne plochu. Kameru tedy budeme umísťovat do težiště trojúhelníku, které nám bude celý trojúhelník reprezentovat.

Tento přístup má samozřejmě svá pro i proti. Některá pro jsme si už nastínili – největším z nich je asi řešení viditelnosti. Mezi zápory patří hlavně fakt, že trojúhelníky reprezentujeme jejich těžištěm – a to nezávisle na jejich tvaru a velikosti! To klade nároky především na kvalitní trojúhelníkový model scény - trojúhelníky by měly být všechny alespoň konvexní.

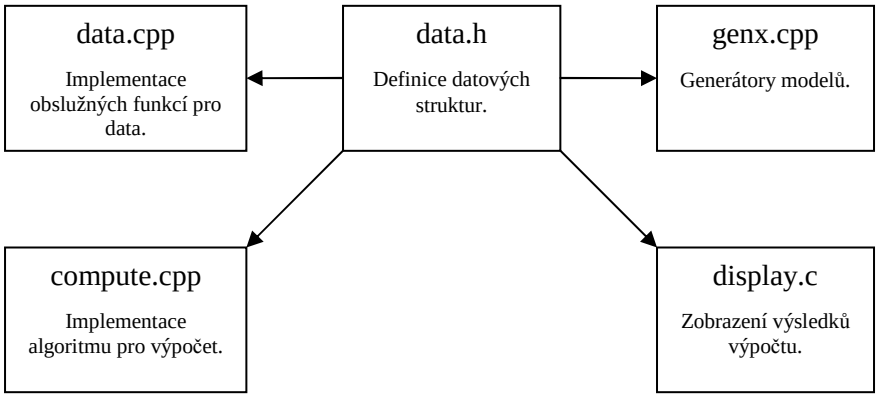
Dalším problémem je nevinně vypadající „pohled do scény“. Takový pohled je ideální realizovat tzv. rybím okem, které je ale hodně náročné na výpočet (navíc OpenGL je ochotno nám poskytnout pouze 2D obrazy). Proto budeme muset zvolit jiný přístup – zde použijeme místo polokoule polokrychli, která se bude skládat z 5-ti různých pohledů (1 centrální plný a 4 boční poloviční). Hodnoty získané z této polokrychle pak vynásobíme příslušnými koeficienty, které nám kompenzují tvar polokrychle na polokouli.

Zbývá zvolit vstupní a výstupní formát dat. Vzhledem k cíli této práce (názorně zpracovat implementaci radiozity použitelně pro další studium a rozvoj projektu) bylo zvoleno velmi jednoduché řešení. Data jsou uložena binárně (přímo struktury pro trojúhelník) v souboru. Výpočetní modul data načte do paměti, provede iteraci výpočtu a opět je uloží do souboru. V průběhu výpočtu se scéna nemění, takže je možno celou scénu vyrobit jako displaylist a ten jenom zobrazovat dle nastavení kamery, což výrazně urychlí renderování scény.

3.2 Základní schéma projektu

Nejprve je třeba rozvrhnout si strukturu a vytyčit základní záchytné body. V předchozí kapitole jsme si řekli, že pro jednoduchost využijeme OpenGL a programovacím jazykem bude C/C++. Vzhledem k tomu, že OpenGL je stavový stroj a scéna se tvoří postupným zadáváním vrcholů a jejich vlastností, optimalizujeme také další součásti, především data, pro tento účel.

Na obrázku 3.1 je uvedeno základní schéma projektu, ze kterého vyjdeme a podrobněji popíšeme jednotlivé součásti.



Obrázek 3.1 – schéma projektu

3.3 Datové struktury

Implementace datové struktury a formátu souboru pro uložení dat na disk je poměrně jednoduchá. Definice datových struktur jsou uloženy v souboru data.h. Nejprve bychom měli vyřešit problém, kde data pro výpočet vezmeme a jak budeme mezivýsledky a výsledky uchovávat. K tomuto účelu nám bude sloužit datová struktura TRIANGLE:

```
typedef struct {
  float p1[3];          // první bod
  float p2[3];          // druhý bod
  float p3[3];          // třetí bod
  float n[3];           // normála povrchu
  float c[3];           // barva
  float ro;             // odrazivost
  unsigned int e;       // vlastní radiozita
  unsigned int b;       // přijatá radiozita
} TRIANGLE;
```

Tato struktura uchovává informace o jednom trojúhelníku. Data jsou v souboru uložena jako pole za sebou jdoucích struktur TRIANGLE. Na začátku souboru je uložen počet struktur – trojúhelníků.

Pro čtení a zpracování pixelů z obrazového bufferu bude sloužit struktura PIXEL:

```
typedef struct {
  unsigned char R;       // červená složka
  unsigned char G;       // zelená složka
  unsigned char B;       // modrá složka
  unsigned char A;       // alfa kanál
} PIXEL;
```

Informace o bodech a vektorech v prostoru budeme uchovávat v poli typu VECTOR:

```
typedef float VECTOR[3];
```

V načítání a ukládání dat z a do souboru žádná věda není – naopak díky uložení trojúhelníků v poli stačí toto pole zapsat nebo naopak načíst ze souboru. Ke scéně budeme pro jednoduchost přistupovat jako ke skupině na sobě nezávislých trojúhelníků popsaných strukturou TRIANGLE.

3.4 Základní myšlenka výpočtu

Nebudeme rozlišovat mezi světelnými zdroji a ostatními ploškami. Všechny plošky (trojúhelníky) jsou reprezentovány stejně a jejich vlastnosti se liší pouze v nastavení proměnných *e* (vlastní radiozita), *b* (přijatá radiozita) a *ro* (odrazivost).

Scénu budeme považovat za energeticky uzavřený celek. Z toho vyplývá, že pokud nějaké plošky vyzáří určitou radiozitu, po čase se stav ustálí a vyrovná se radiozita do scény vyzářená a radiozita scénou (ploškami) přijatá. Toto „ustálení“ nenastane samozřejmě hned a závisí na více faktorech – především na složitosti scény (maximálním počtu odrazů). Výpočet nám tedy bude postupně konvergovat a světlo (radiozita) se bude šířit od plošek nejblíže zdroji světla na plošky vzdálenější, nepřímo osvětlené atd.

Abychom zajistili tuto konvergenci, musíme zvolit správný postup při výpočtu radiozity na plošce. Logické by bylo říct, že radiozita plošky je jedna jediná (počítá s tím i rovnice radiozity). My ovšem tyto dvě radiozity musíme oddělit. Jedná se o proměnnou *e* (vlastní vyzařovaná radiozita) a *b* (radiozita přijatá od ostatních plošek). Dále si musíme uvědomit, kdy budeme se kterou radiozitou počítat. Budeme-li zobrazovat výsledek, musíme výslednou radiozitu (světelnost) plošky vyjádřit jako:

Rovnice 3-1

$$B = e + b$$

Zatímco, budeme-li chtít spočítat radiozitu, kterou ovlivňuje naše ploška okolní plošky, budeme výslednou radiozitu počítat jako:

Rovnice 3-2

$$B = e + b \cdot ro$$

Pořád zde hovoříme o vzájemném ovlivňování plošek, což v radiozitní rovnici představují konfigurační faktory. V přístupu, který byl zvolen zde, se však s ničím jako konfigurační faktor

nesetkáme. Respektive, ony zde budou zaobaleny do jiných věcí, které budou tyto faktory aproximovat.

Jak tedy budeme počítat radiozitu přijatou ploškou? Naprosto přirozeně – podíváme se do scény a zjistíme, co daná ploška „vidí“. V tom nám pomůže OpenGL, které za nás tak vyřeší několik problémů najednou. Nebudeme muset řešit viditelnost jednotlivých plošek. Nebudeme muset také řešit vliv vzdálenosti plošek. Oba tyto problémy se promítají ve skutečnosti do konfiguračních faktorů a je zde vidět, že problému jsme se nezbavili, byl pouze přesunut jinam.

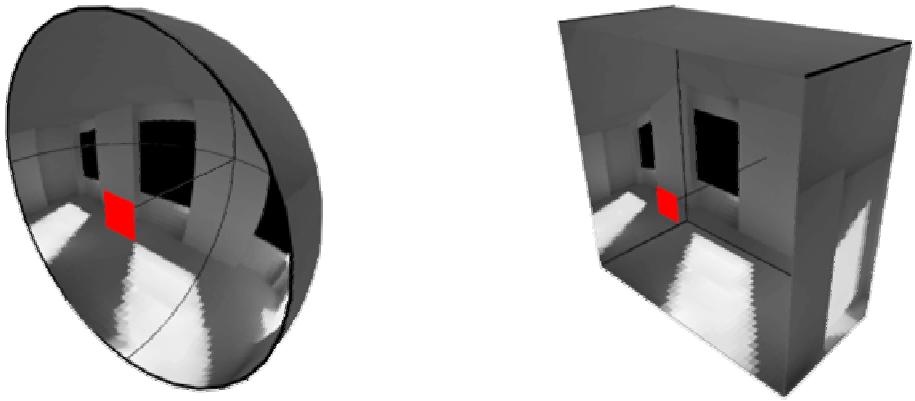
Základní myšlenkou tedy bude procházet všechny trojúhelníky ve scéně a počítat na ně dopadlou radiozitu pomocí pohledu z jejich perspektivy do scény. Tuto činnost budeme provádět iterativně a to tak dlouho, dokud bude docházet ke změnám radiozity přijaté ploškami. Zde můžeme zvolit dva způsoby procházení a výpočtu:

1. Radiozitu vypočítanou v dané iteraci zahrnout do výpočtu až v další iteraci
2. Radiozitu vypočítanou v dané iteraci zahrnout do výpočtu ještě v téže iteraci

Byly vyzkoušeny obě možnosti. Druhá možnost zpočátku rychleji konvergovala k cíli, avšak zobrazování bylo pomalejší, celý čas nutný pro výpočet tedy delší. Nakonec byla tedy zvolena možnost druhá, kdy byl pro každou iteraci vytvořen *displaylist*, který podstatně urychlil vykreslování scény v průběhu výpočtu a tím podstatně zkrátil čas.

3.5 Pohled do scény

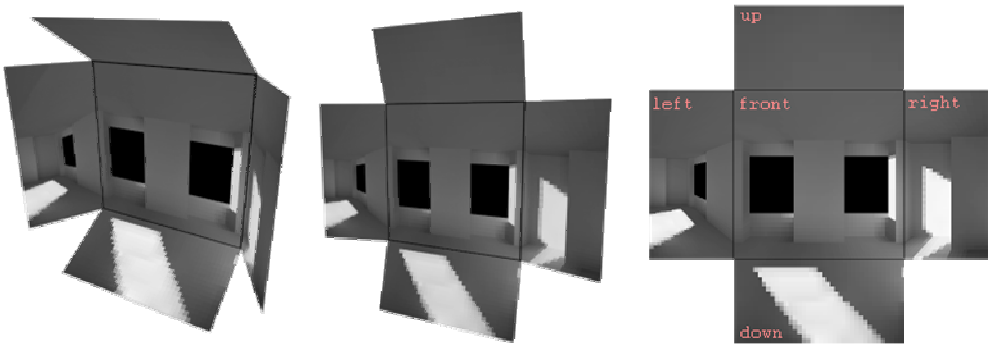
Jak vygenerovat pohled do scény je základním problémem, který musíme řešit, když chceme správně spočítat množství světla, které dopadne na plošku. Správně bychom měli nad plošky umístit polokouli (tzv. „rybí oko“) a počítat s radiozitou dopadlou na její povrch.



Obrázek 3.2 – polokoule nad ploškou je nahrazena polokrychlí [4]

Matematicky by to bylo však zbytečně složité, a tak použijeme polokrychli, kterou mnohem snáze vygenerujeme a po příslušných korekcích dostaneme ten samý výsledek. Vše naznačuje obrázek 3.2.

Polokrychle je však prostorový útvar a tudíž jej musíme rozvinout do roviny (viz obrázek 3.3).

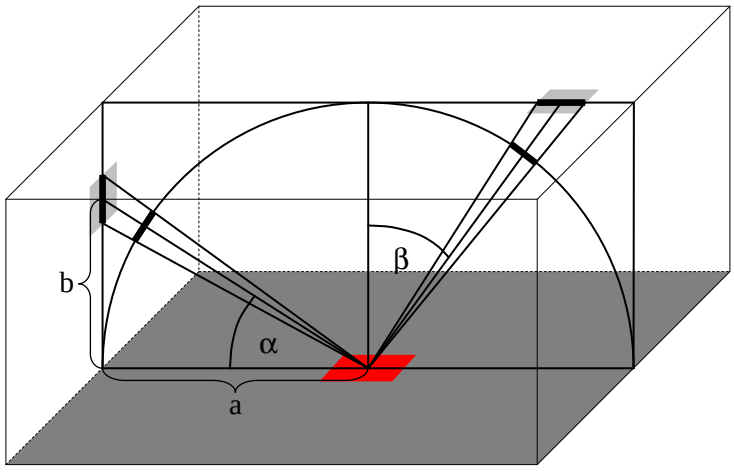


Obrázek 3.3 – rozvinutí polokrychle do prostoru [4]

Vznikne nám tak pět samostatných ploch: jeden centrální čtverec a 4 boční pohledy. Ty lze již poměrně lehce vygenerovat natočením kamery o 90° vždy do správného směru.

3.5.1 Korekce tvaru polokrychle

Jak jsme již v minulé části řekli, abychom dostali stejný výsledek z polokrychle jako z polokoule, je potřeba pixely na povrchu polokoule korigovat. Myšlenku potřebné korekce naznačuje obrázek 3.4.



Obrázek 3.4 – Korekce tvaru polokrychle na polokouli

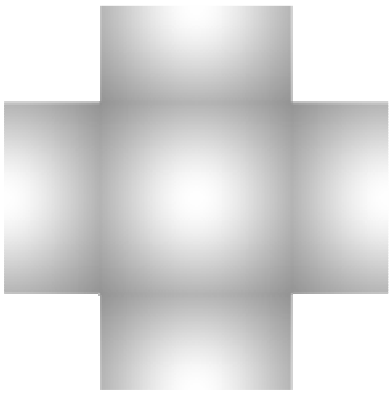
Z obrázku je jasně vidět, že průmět (hodnota) pixelu z polokrychle na polokouli závisí na úhlu α resp. β a musí bý korigován. Hodnotu korekce vyjadřuje následující rovnice:

Rovnice 3-3

$$k = \frac{a}{\sqrt{a^2 + b^2}}$$

Což vyjadřuje v podstatě kosinus úhlu α (analogicky by se spočítal druhý průmět). Druhý průmět je zde naznačen pouze proto, abychom si uvědomili, že je-li $\alpha > 45°$, počítáme již s jeho doplňkem do 90°.

Představíme-li si celou naznačenou situaci v prostoru a nikoli v řezu, jak je tomu na obrázku 3.4, a spočteme-li pro všechny pixely polokrychle korekční mapu, kterou můžeme zobrazit pomocí odstínů šedé (bílá = 1, černá = 0), dospějeme k obrázku 3.5.



Obrázek 3.5 – Korekční mapa pro tvar polokrychle [4]

3.5.2 Lambertův kosinový zákon

Jedná se o zákon, který nám říká, jaká část světla dopadajícího na plochu skutečně plochu osvítí. Toto množství je dáno kosinem úhlu mezi kolmicí dopadu a dopadajícím paprskem. Vše je patrné z obrázku 3.6. Hodnota kosinu pro daný pixel lze poměrně lehce spočíst – stejně jako v minulém případě. Výpočet pro centrální část bude totožný, lišit se bude jenom výpočet pro boční stěny. Zde využijeme toho, že

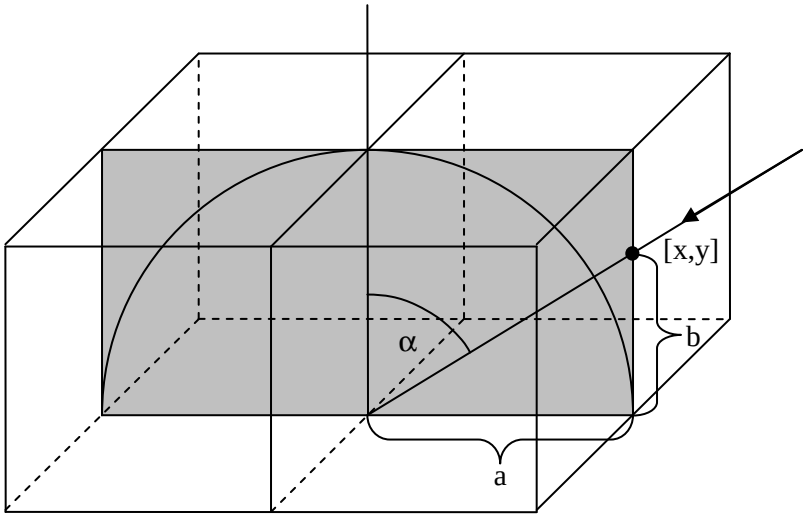
Rovnice 3-4

$$\cos(\alpha) = \sin(90° - \alpha)$$

A tudíž bude výpočet pro boční stěnu vypadat:

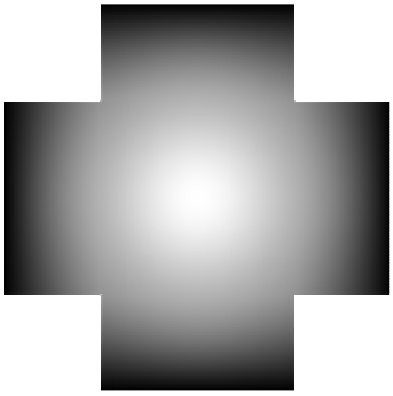
Rovnice 3-5

$$k = \frac{b}{\sqrt{a^2 + b^2}}$$



Obrázek 3.6 – Lambertův kosinový zákon

Zobrazíme-li si hodnoty pro všechny pixely na jednotkové polokrychli, dostaneme násobící mapu jako na obrázku 3.7 (bílá = 1, černá = 0).



Obrázek 3.7 – Zobrazení Lambertova zákona [4]

3.5.3 Implementace násobících map

Složíme-li obě výše popsané operace do jedné, můžeme si vytvořit násobící mapu (pole koeficientů), kterou budou tvořit čísla od 0 do 1, a kterou budeme násobit hodnoty pixelů na příslušných pozicích *x* a *y*. Pro každý z pěti pohledů polokrychle budeme potřebovat pole typů *float*, kam si jednotlivé korekční faktory uložíme. Podíváme-li se ale na obrázek 3.5, zjistíme, že všechny 4 boční části jsou vlastně z hlediska koeficientů totožné. Vyrobíme si tedy jenom 2 pole, jedno pro centrální část a jedno pro všechny 4 boční části.

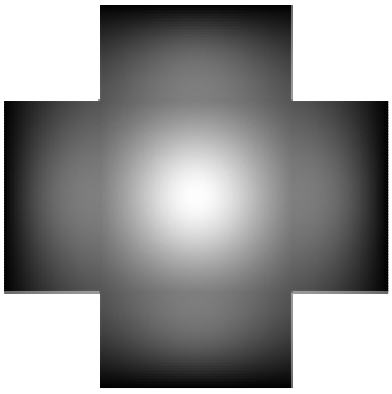
Nejprve si tedy připravíme tato pole (*cmap* pro centrální pohled a *smap* pro boční pohledy, proměnné *w* a *h* představují rozměry pohledu – zde *w* = *h*, protože pracujeme s polokrychlí, ale obecně to může být různé):

```
cmap = (float *) malloc(sizeof(float)*w*h);     // centrální pohled
smap = (float *) malloc(sizeof(float)*w*(h/2)); // boční pohledy
```

Dále provedeme dva cykly (aplikace rovnic) – jeden pro centrální a jeden pro boční mapy:

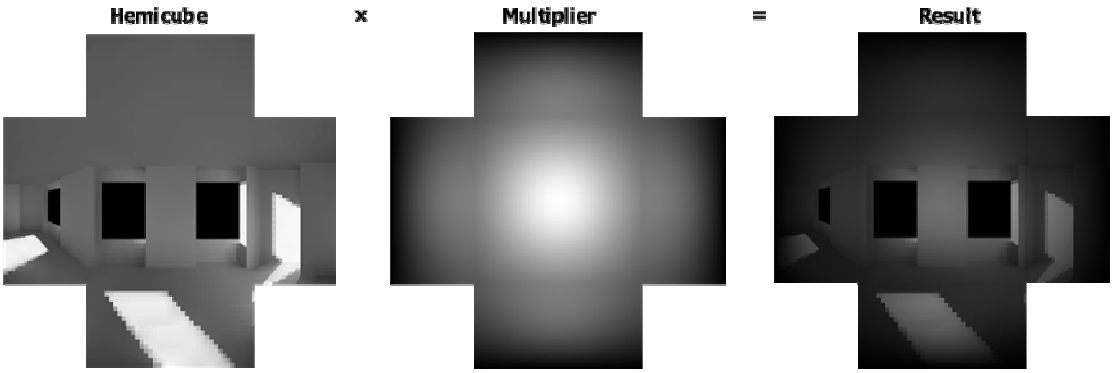
```
for (int y = 0; y < h; y++) {
  for (int x = 0; x < w; x++) {
    float a = fabs((0.5 + (x - w/2.0)) / (w/2.0));
    float b = fabs(((h/2.0 - y) - 0.5) / (w/2.0));
    // korekce tvaru
    cmap[y*w + x] = (1 / sqrt(1 + a*a + b*b));
    // Lambertuv zakon
    cmap[y*w + x] = cmap[y*w + x];
  }
}

for (int y = 0; y < h/2; y++) {
  for (int x = 0; x < w; x++) {
    float a = fabs((0.5 + (x - w/2.0)) / (w/2.0));
    float b = fabs(((h/2.0 - y) - 0.5) / (w/2.0));
    // korekce tvaru
    smap[y*w + x] = 1 / sqrt( 1 + a*a + b*b );
    // lambertuv zakon
    smap[y*w + x] = smap[y*w + x] * (b / sqrt( 1 + a*a + b*b ));
  }
}
```

Tím máme připravenou násobící mapu. Zobrazíme-li si celek, dostaneme obrázek 3.8.



Obrázek 3.8 – Výsledná násobící mapa pro jednotkovou polokrychli [4]

Mapu je vhodné před použitím taktéž normalizovat (podělit hodnotu každého pixelu součtem hodnot všech pixelů). Jak bude násobící mapa pracovat, je zobrazeno na obrázku 3.9.



Obrázek 3.9 – Funkce násobící mapy [4]

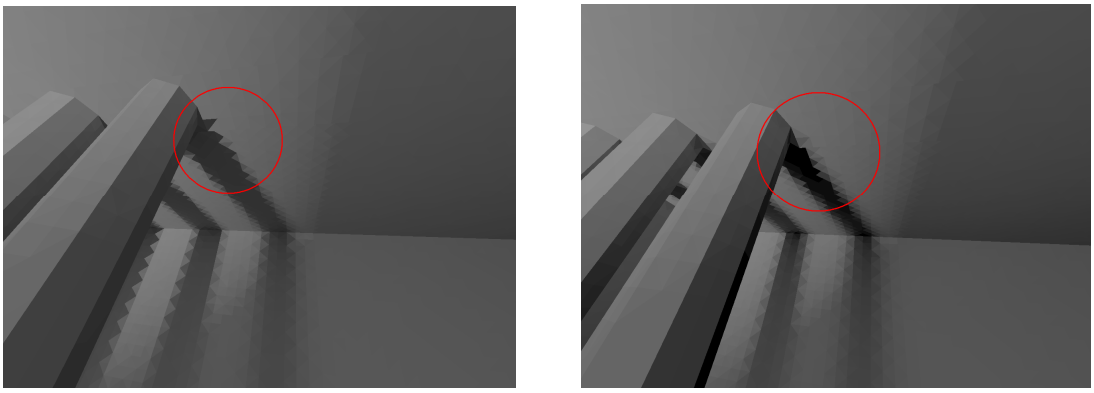
3.6 Získání pohledu do scény

Obrázky polokrychle se stěnami již pokrytými vyrenderovanými pohledy do scény vypadají pěkně, nejsou však takovou samozřejmostí, jak by se mohlo na první pohled zdát. Pravdou ovšem je, že nám zde výrazně pomůže OpenGL. Naším úkolem v této fázi je umístit kameru do stanoveného bodu a vytvořit tak 5 vzájemně kolmých pohledů do scény. Tyto pohledy musíme pak načíst z bufferu do paměti, abychom s daty mohli dále pracovat.

3.6.1 Umístění kamery

Jako první se nabízí otázka, kam umístit kameru. V předchozích částech jsme si naznačili, že jako reprezentativní bod v trojúhelníku bylo zvoleno jeho těžiště. To má své výhody i nevýhody. U malých trojúhelníků je to celkem jedno a výsledky to příliš neovlivní, ale u větších trojúhelníků, popř. u plošek, které se nacházejí na hranici světlo/stín to může představovat problém. Na obrázku 3.10 je pěkně vidět, jak umístění kamery ovlivňuje výsledek. Na obrázku a) je kamera umísťována do těžiště a tak je ovlivňováno množství světla, které dopadá na celý trojúhelník – obrázek je pak trochu „zubatější“. Zatímco na obrázku b) je umísťována kamera do všech tří vrcholů trojúhelníka a výsledek je získaný ze všech tří pohledů průměrováním. Toto řešení je však úměrně výpočetně náročnější. V praxi se to řeší adaptivním dělením modelu, popř. počítáním na již adaptovaném (předděleném) modelu scény. Dělení modelu však nebylo předmětem této práce, i když to s radiozitou přímo souvisí.

Dalším rozdílem, který si na obrázku můžeme demonstrovat je to, že umísťování kamery postupně do vrcholů přináší lepší obraz již při menším počtu iterací: obrázek a) je již skoro ustálený stav, zatímco obrázek b) je teprve napůli cesty – vypadá však již lépe.



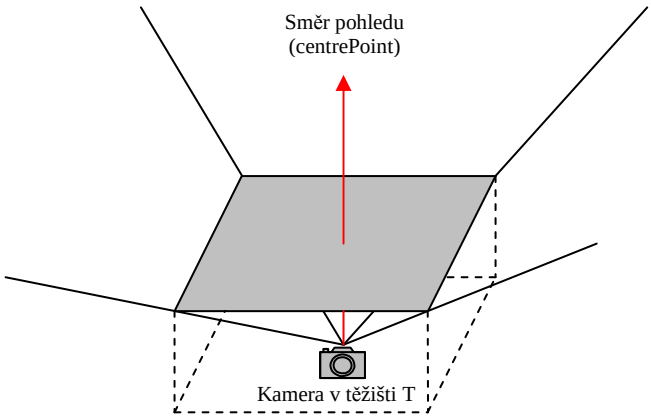
a) b)

Obrázek 3.10 – Umístění kamery v těžišti trojúhelníku (a) a ve vrcholech (b)

3.6.2 Nastavení OpenGL

Abychom mohli správně vytvořit všechny pohledy potřebné pro kompletaci polokrychle, je potřeba OpenGL správně nastavit. Na obrázku 3.11 je vidět úhel pohledu přes centrální část polokrychle. Je zde jasně vidět, že šíře záběru musí být 90°. Je tedy velmi důležité správně nastavit především úhel pohledu – k tomu slouží v OpenGL funkce *gluPerspective*:

```
gluPerspective(90, 1.0, nearPlane, farPlane);
```

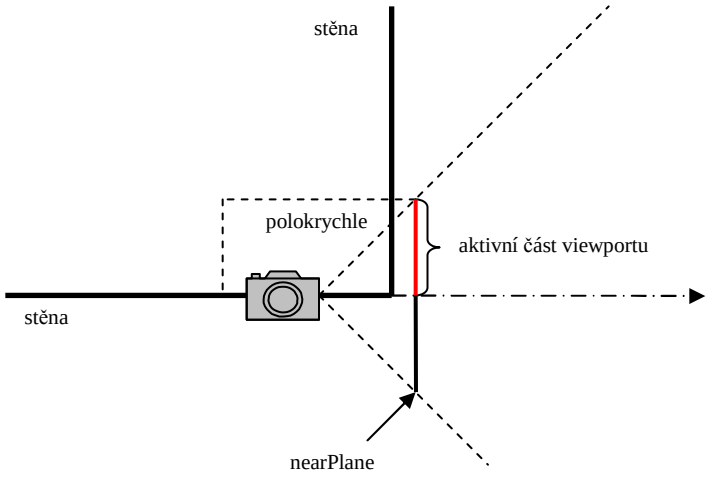


Obrázek 3.11 – Úhel pohledu

Dále musíme správně nastavit (napevno – nesmí se měnit se změnou velikosti okna) viewport. Musí mít rozlišení v obou směrech (*WindowWidth* i *WindowHeight*) stejné. Tedy např. 500×500 px nebo 200×200 px. Viewport se v OpenGL nastavuje funkcí *glViewport*:

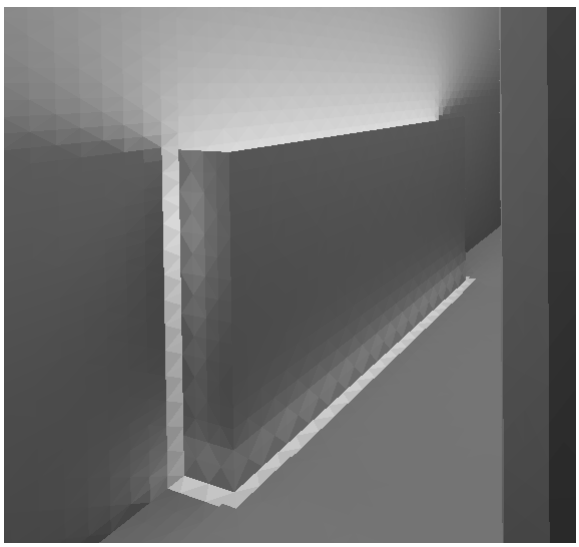
```
glViewport(0, 0, WindowWidth, WindowHeight);
```

Důležité je také nastavení hodnot *nearPlane* a *farPlane* (bližší a vzdálenější ořezávací rovina), neboť to má vliv na reálnou vzdálenost vytvoření obrazu (viewportu) od polohy kamery. V zásadě by *nearPlane* mělo být co nejmenší a *farPlane* co největší. Nastavíme-li totiž *nearPlane* moc velké, dojde k situaci zobrazené na obrázku 3.12.



Obrázek 3.12 – problém s nastavením parametru *nearPlane*

Kamera je umístěna na plošku vedle stěny, tato stěna se však v záběru neobjeví, neboť je ořezána a viewport je posunut až za stěnu. Výsledek je pak vidět na obrázku 3.13. Tento problém se může objevit také při zmenšení velikosti trojúhelníků pod limit daný nastavením hodnoty *nearPlane*.



Obrázek 3.13 – povšimněte si parazitního světlého lemování v rozích přiléhajících stěn

Dalšími nastaveními se již tak podrobněji zabývat nebudeme, neboť jsou poměrně standardní – užíváme Z-bufferu apod. (je možnost nahlédnout do zdrojových kódů). Přesto bych zde připomněl důležitou maličkost – musí být vypnuté osvětlování příkazem *glDisable(GL_LIGHTING)*. To by nám velmi zkreslilo výsledky a velmi špatně se taková chyba odhaluje.

3.6.3 Načtení vyrenderované scény

V předchozích kapitolkách jsme si řekli jak obecně nastavíme OpenGL. Zde se zmíníme o tom, jak dostat z OpenGL výsledek – zobrazenou scénu. Neboť v průběhu výpočtu v okně nic nezobrazujeme, je celkem jedno, jestli použijeme model s jedním nebo dvěma buffery. Použijeme-li model se dvěma buffery, bude nutné ještě před načtením dat zvolit, ze kterého bufferu se bude číst.

Nejprve si připravíme prostor, kam načteme data z bufferu. Několik stran výše jsme se již zmínili o struktuře PIXEL. Tu nyní využijeme pro vytvoření pole těchto struktur a do něj načteme data z bufferu.

```
// vytvoreni bufferu pro data z obrazoveho bufferu
PIXEL * pixels;
pixels = (PIXEL *) malloc(sizeof(PIXEL)*w*h);
```

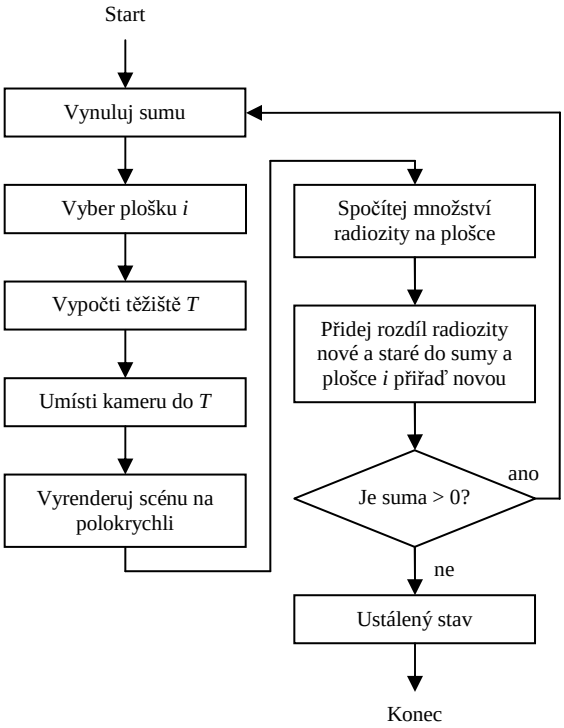
Do tohoto bufferu *pixels* budeme načítat data pomocí funkce *glReadPixels*. Ta si vezme jako první 4 parametry rozměry obdélníku, který chceme načíst počínaje levým dolním rohem, dále formát čtených dat *format* (typ *GLenum* a hodnoty GL_RGBA a GL_RGB), typ složek pixelu (zde chceme GL_UNSIGNED_BYTE) a nakonec ukazatel na volné místo – náš buffer *pixels*. Vše i s nastavením obrazového bufferu, ze kterého chceme číst data vidíme zde:

```
glReadBuffer(GL_BACK);
glReadPixels(0, 0, w, h, format, GL_UNSIGNED_BYTE, pixels);
```

Rozměry obdélníku, který chceme z obrazového bufferu načíst se budou pochopitelně lišit pro centrální a boční části polokrychle – uvedený příklad je pro centrální část.

3.7 Výpočet radiozity na plošce

Nyní již máme připraven všechen potřebný aparát a můžeme přistoupit k vlastnímu iterativnímu výpočtu radiosity na jednotlivých ploškách. Jak výpočet bude probíhat, je naznačeno na obrázku 3.14, který zachycuje celý algoritmus. Jednotlivé jeho části si rozebereme a popíšeme zvlášť.



Obrázek 3.14 – Algoritmus výpočtu radiozity ve scéně

Celý výpočet radiozity probíhá ve funkci *onDisplay*, která je v OpenGL volána při nutnosti překreslení okna. Funkce tedy neskončí dříve, než se neustálí výpočet. Celý výpočet (jak je vidět z algoritmu) probíhá, dokud se v rámci jedné iteraci vyskytla nějaká změna radiozity. Jednotlivé iterace probíhají postupným procházením všech plošek ve scéně a přiřazováním nových hodnot radiozity těmto ploškám.

3.7.1 Nasměrování kamery

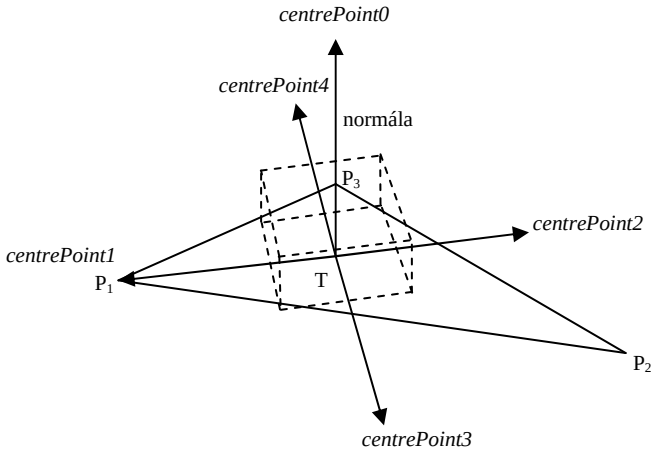
Výpočet radiozity na jedné plošce si nyní rozebereme podrobněji. Z algoritmu je patrné, že nejprve umístíme kameru do těžiště (popř. jiného bodu, ale těžiště je pro trojúhelník asi nejreprezentativnějším bodem) plošky. Těžiště lehce spočteme (*t* je trojúhelník):

```
createVector(&T,
  (t->p1[0] + t->p2[0] + t->p3[0]) / 3.0,
  (t->p1[1] + t->p2[1] + t->p3[1]) / 3.0,
  (t->p1[2] + t->p2[2] + t->p3[2]) / 3.0
);
```

Pro nastavení pohledu kamery nám v OpenGL slouží funkce *gluLookAt*. Ta si vezme tři parametry – první parametr je poloha kamery (bod – v našem případě těžiště), druhý je bod, na který se má kamera zaměřit (*centrePoint*) a třetí parametr představuje určení vrcholu obrazu (horní hranu) – je to vektor (*topVector*). Celé volání funkce pak vypadá následovně:

```
gluLookAt(
  T[0], T[1], T[2],
  centrePoint[0], centrePoint[1], centrePoint[2],
  topVector[0], topVector[1], topVector[2]
);
```

Těžiště bude pro všech pět pohledů potřebných pro polokrychli stejné. Pro centrální část určíme *centrePoint* lehce – stačí k těžišti přičíst vektor normály plošky. Jako *topVector* můžeme zvolit např. vektor z *T* do libovolného vrcholu trojúhelníka. U bočních pohledů bude situace poněkud složitější, ale ne zase tolik. *T* nám zůstává, jako *centrePoint* můžeme u prvního pohledu použít bod (vrchol) použitý u centrálního pohledu a jako *topVector* zase normálu (tu je možno použít u všech 4 bočních pohledů). Nyní zbývá u ostatních 3 pohledů dopočíst pouze *centrePoint* a to velmi jednoduše – druhý boční pohled vytvoříme lehce odečtením vektoru z *T* do *centrePointu* prvního bočního pohledu od bodu *T*. Poslední dva – třetí a čtvrtý – vytvoříme stejným způsobem, ale s vektorem kolmým na dvojici *T*,*centrePoint* a *topVector* (normálu). Zní to složitě a napsané to působí zajisté nepřehledně, pojďme si to tedy shrnout na obrázku 3.15.



Obrázek 3.15 – Parametry funkce *gluLookAt*

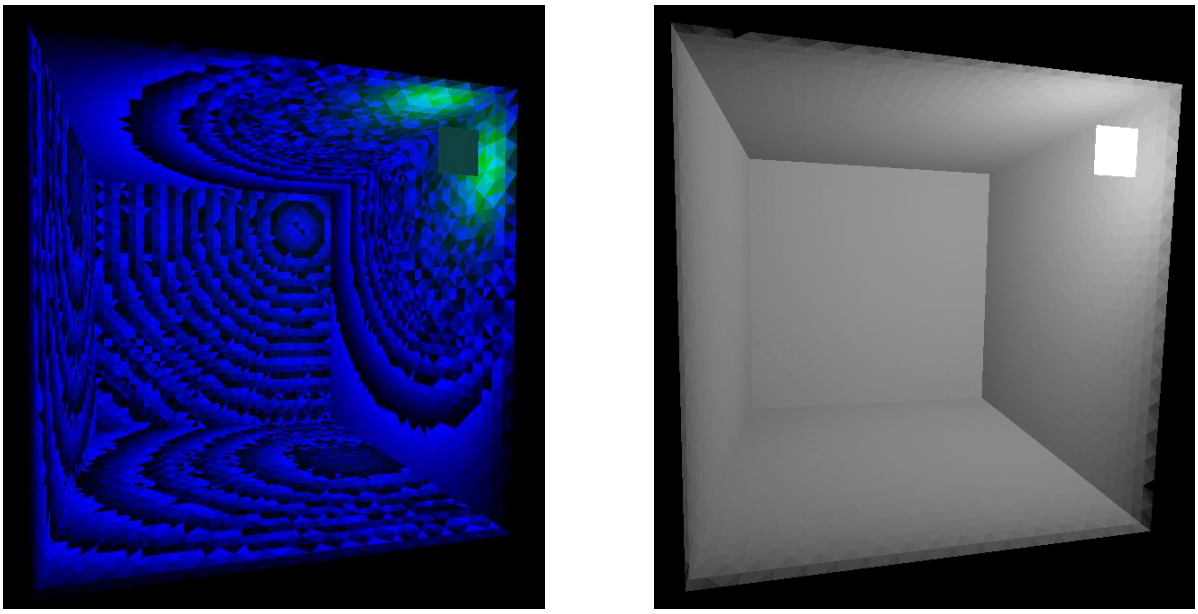
Nyní nám již nic nechybí k tomu, abychom si mohli předvést vlastní nastavení, vykreslení a načtení scény tak, jak se ve skutečnosti volá (pro jeden pohled – opakujeme tedy 5×):

```
// vymazani bufferu
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

// nastaveni kamery
gluLookAt(
  T[0],T[1],T[2],
  centrePoint[0],centrePoint[1],centrePoint[2],
  topVector[0],topVector[1],topVector[2]
```

```
);

// vykresleni sceny z displaylistu
glCallList(scene);
glFlush();

// nacteni dat z obrazoveho bufferu
glReadBuffer(GL_BACK);
glReadPixels(0,0,w,h,format,GL_UNSIGNED_BYTE,pixels);
```

3.7.2 Zpracování dat

Nyní se již nacházíme ve fázi, kdy jsme schopni pro každou plošku získat všech pět pohledů na polokrychli, jsme schopni data z těchto pohledů načíst do našeho bufferu a zbývá je již jen zpracovat. Normálně bychom si řekli, že máme hodnoty 0…255 a tudíž bude pixel více, či méně šedý. Co kdybychom ovšem využili všech tří složek RGB a chápali posloupnost bytů jako jedno 24-bitové číslo? Rázem tam získáme rozsah hodnot nikoli 0…255, ale již rozumnějších 0…16 777 216. Vůbec nám nevadí, že při zobrazování scény při výpočtu se scéna bude zobrazovat různě barevně, pro člověka nesmyslně. Výsledek si stejně zobrazíme v šedých odstínech, kterých ve výsledku již nemusí být mnoho, neboť se prokázalo, člověk je stejně schopen okem rozlišit pouze asi 32-64 různých stupňů šedi. Rozdíl zobrazený barevně (při výpočtu) a šedě (při zobrazení výsledku pro člověka) je patrný z obrázku 3.16.



a) b)

Obrázek 3.16 – Různá zobrazení stejné scény při výpočtu (a) a výsledek pro člověka (b)

Budeme tedy pracovat se složkami RGB, jako by to bylo jedno číslo. Po vyrenderování každého z pěti pohledů na polokrychli a načtení vyrenderovaného obrazu do bufferu musíme projít tento buffer pixel po pixelu a nejdříve spočíst ze složek RGB jedno číslo, které nám reprezentuje radiozitu daného bodu, a pak ještě vynásobit tuto hodnotu příslušným koeficientem z násobících map (*smap* nebo *cmap*), které jsme si připravili v jedné z předešlých kapitol. Suma radiozit všech pixelů ze všech pěti pohledů nám dá výslednou hodnotu radiozity pro danou plošku.

Jak to bude vypadat konkrétně v kódu si ukážeme zde. Nejprve získání jednoho čísla z RGB – poměrně jednoduchá záležitost – stačí posunout jednotlivé bajty čísla na správné pozice a pak vynásobit příslušným koeficientem (zde *cmap* pro centrální pohled):

```
sum += (pixels[j].R<<16 + pixels[j].G<<8 + pixels[j].B) * cmap[j];
```

Takto se k sumě *sum* přičte radiozita z každého pixelu ze všech pěti pohledů. Nakonec je třeba výslednou sumou nahradit původní radiozitu na plošce a rozdíl mezi starou a novou radiozitou přičíst k proměnné, která hlídá celkovou výměnu radiozity ve scéně:

```
// suma rozdilu radiozit ve scene
diff += abs((unsigned int)sum - t->b);

// nova radiozita pro plosku, e zustava beze zmeny
t->b = (unsigned int)(sum);
```

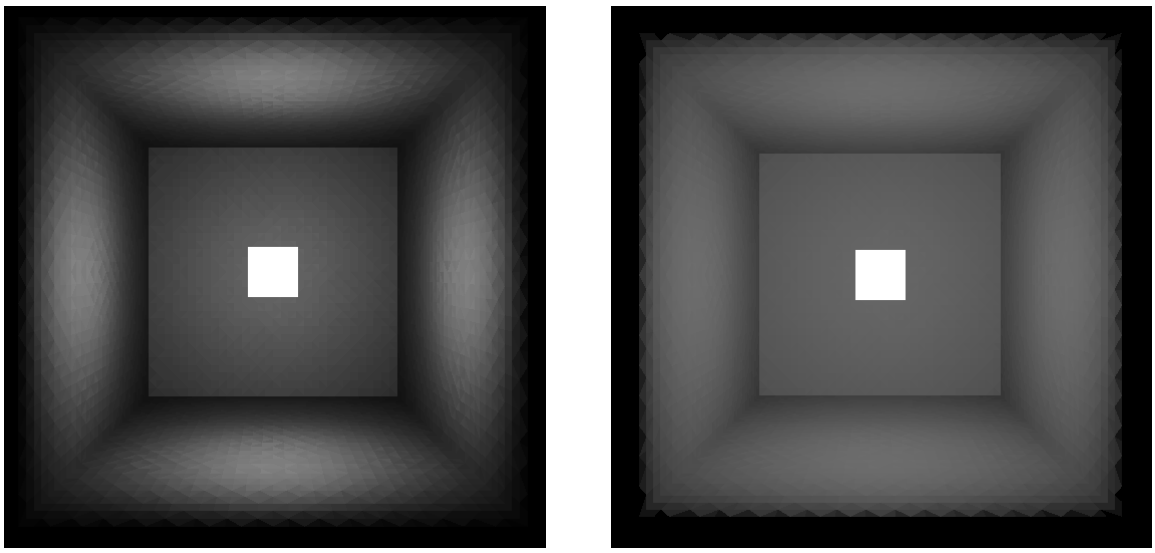
No a tím máme popsány všechny části algoritmu z obrázku 3.14 a jsme tudíž schopni nechat celý výpočet iterovat a dojít až k ustálenému stavu, kdy nedochází již k žádné výměně radiozity mezi ploškami – proměnná *diff* je tedy rovna nule.

3.8 Zobrazení výsledků

Všechno počítání bychom dělali marně, pokud bychom nebyli schopni nějak rozumně zobrazit výsledek našeho výpočtu. Zde není větších problémů, ale přesto je třeba dobře zvážit formu prezentace výsledků. Vzhledem k tomu, že počítáme s rozptýleným a odraženým světlem (nikoli se směrovým), dochází k tomu, že na jednotlivých ploškách je velmi malá část „výkonu“ světelného zdroje. Nebo jinak řečeno, hodnoty radiozity na ploškách jsou o několik řádů menší, než hodnota radiozity na světelném zdroji. Zobrazovat tedy v OpenGL tyto hodnoty lineárně a mapovat je do prostoru 0…255 není moudré, neboť nebude téměř nic vidět.

Jako první se tedy nabízí možnost vynásobit hodnoty radiozity na ploškách (ne na zdorjích) nějakou vhodně zvolenou konstantou. Toto řešení však není příliš šťastné, i když dosahuje podstatně lepších výsledků. Mnohem lepší je využít také vlastností lidského oka a namapovat hodnoty do prostoru 0…255 logaritmicky. Lidské oko je totiž mnohem méně citlivé na rozdíly ve vyšších

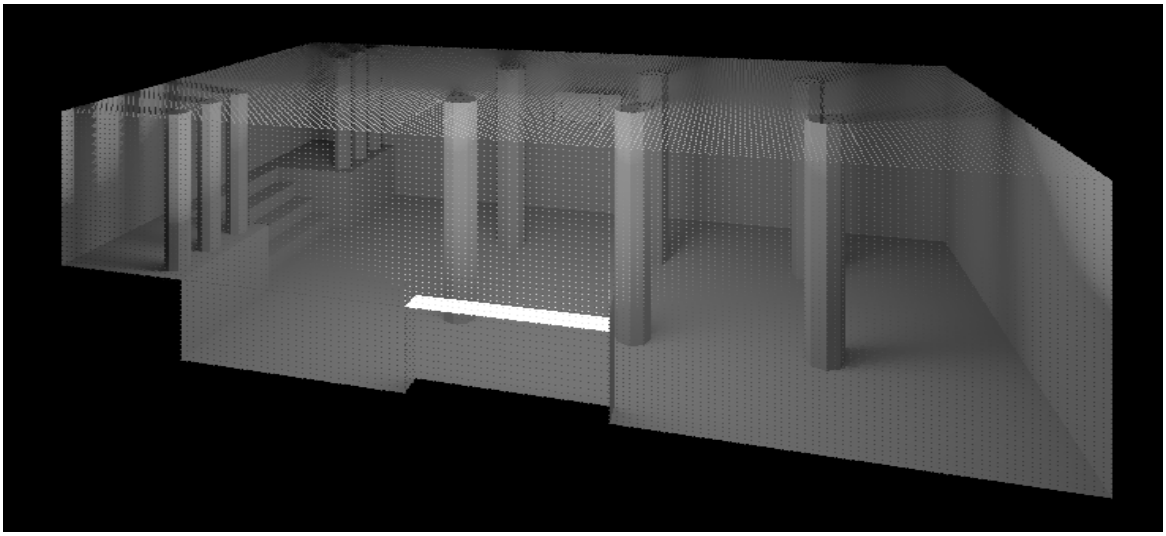
hodnotách jasu, než v nižších. Jak dopadne srovnání těchto dvou metod (násobení konstantou a logaritmus), je pěkně ilustrováno na obrázku 3.17. Metoda logaritmu (b) vypadá mnohem lépe a přirozeněji, neboť metoda násobení konstantou příliš zvýrazňuje rozdíly mezi hodnotami jednotlivých plošek.



a) b)

Obrázek 3.17 – Srovnání metod zobrazení – násobení konstantou (a) a logaritmus (b)

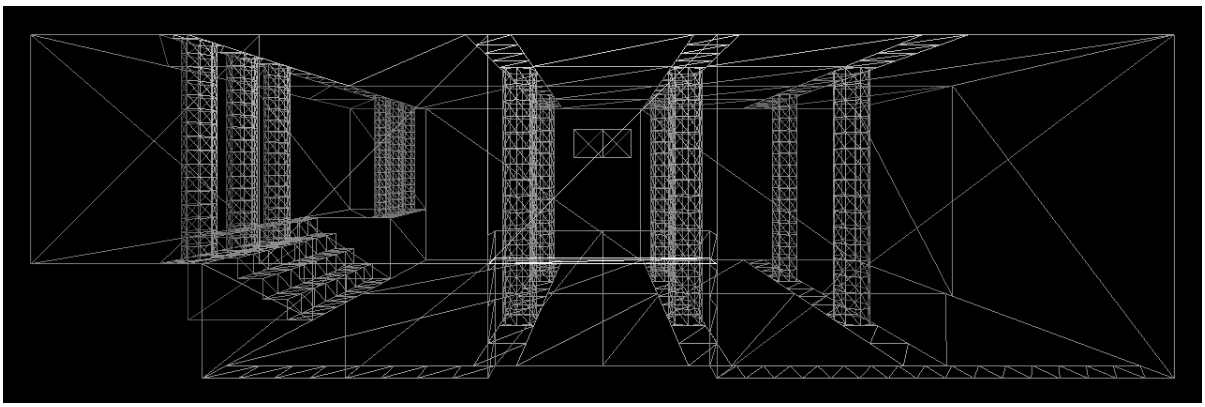
Další, spíše již jen vychytávkou, je zobrazování pouze lícních stěn, což je dobré zejména pro ladění modelu a tato možnost lze ve výsledném programu nastavit (viz. manuál). Implementace ořezávání zadních stěn polygonů byla provedena tak, že polygony jsou zobrazeny pouze jako vrcholy trojúhelníků. Jak vypadá výsledek se zprůhledněnými zadními stěnami lze vidět na obrázku 3.18.



Obrázek 3.18 – Průhledné plochy otočené zadní stěnou ke kameře

4 Výsledky

Pro účely testování a zobrazení výsledků byla vytvořena složitější scéna (použita již na obrázku 3.18), kde je schválně zabudováno několik jevů a my můžeme tak studovat chování této metody na reálně vypadající scéně. Počáteční „drátěný“ model této scény je na obrázku 4.1.



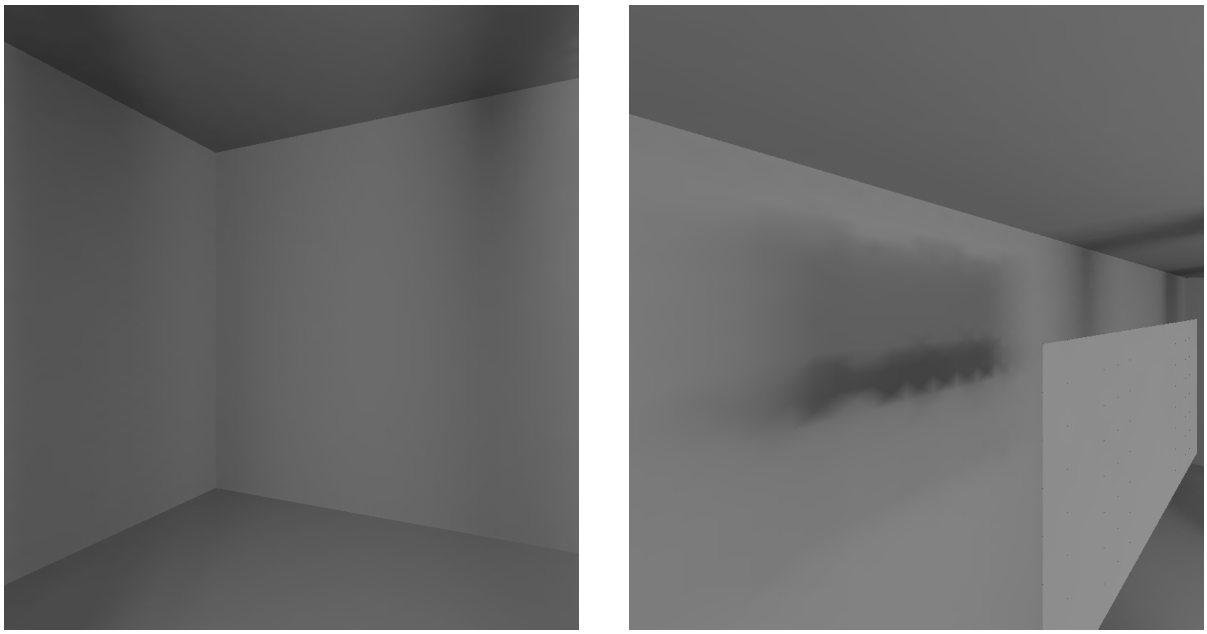
Obrázek 4.1 – počáteční drátěný model testovací scény

Tento model je zcela základní a splňuje pouze jednu podmínku (zabudovanou již v generátoru modelu, který byl pro tento účel vytvořen) – žádná strana trojúhelníka nebude více jak 2× větší než jiná strana téhož trojúhelníka. Toto pravidlo bylo zavedeno na počátku, abychom pak nedostávali artefakty způsobené podlouhlými ostrými trojúhelníky. Tuto scénu je možné dále zjemňovat a to nastavením prahové hodnoty AREA_TRESHOLD v generátoru, která dělí trojúhelník na dva menší do té doby, než jeho plocha není menší než práh. Na těchto různě jemných modelech pak můžeme sledovat chování naší metody.

Veškeré výsledky jsou pochopitelně závislé na kvalitě modelu. Chtěl bych tady zdůraznit, že tato práce se nezabývá problematikou adaptivního dělení modelu, která k radiozitě pochopitelně jinak patří a zajisté by to byl velmi zajímavý směr, kterým by se mohla práce dále ubírat a který by podstatně zvýšil kvalitu výsledků. Ze své povahy si metoda velmi dobře poradí s plynulými světelnými přechody. S ostřejšími přechody světlo-stín je to již horší a velmi to závisí na kvalitě modelu a orientaci plošek v inkriminovaném místě. Toto je způsobeno povahou metody, kdy celou plošku reprezentujeme pouze bodem. V předchozí kapitole (obrázek 3.10) bylo nastíněno jedno z možných řešení (nahrazení jednoho bodu – těžiště – více body za cenu zvýšení času potřebného k výpočtu).

Jako další vylepšení byla vyzkoušena interpolace mezi jednotlivými vrcholy trojúhelníků. Vzhledem k tomu, že obecně metoda předpokládá pouze seznam na sobě nijak nezávislých trojúhelníků (bez znalosti svých sousedů), bylo nutné prvně zjistit sousednosti a vytvořit seznam

vrcholů, které tvoří naše trojúhelníky pomocí pole indexů. Dále barva ve vrcholu vznikala interpolací z příspěvků barev všech trojúhelníků, kterých se tento vrchol účastní. Výsledky tohoto „vylepšení“ nejsou bohužel takové, jako bychom očekávali – metoda totiž občas interpoluje i to, co ve výsledku nevypadá přirozeně. Ukázkou takto interpolované scény může být obrázek 4.2.

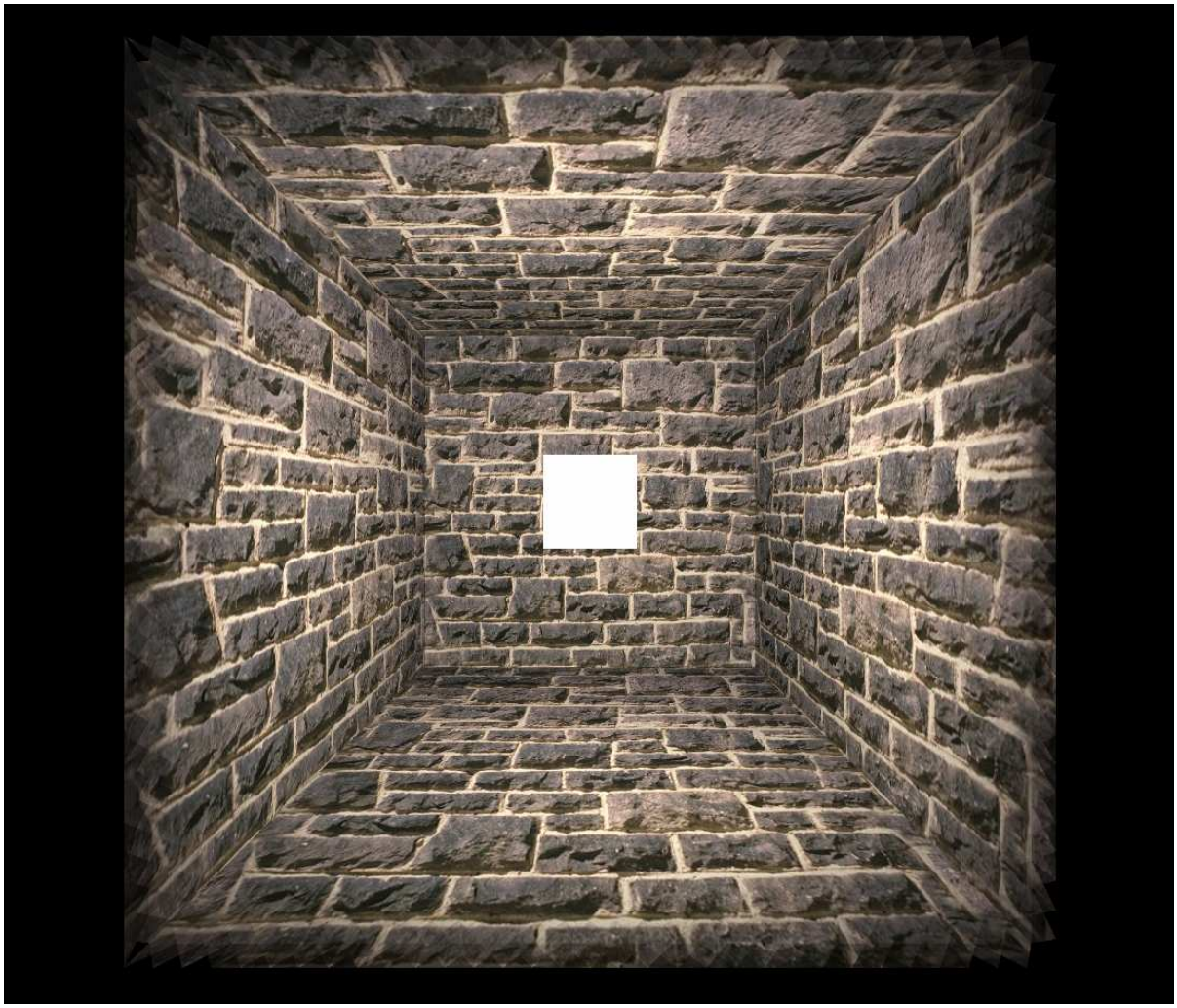


a) b)

Obrázek 4.2 – Interpolovaná scéna – plynulé přechody (a) a hrany (b)

Zatímco plynulé přechody (obrázek 4.2a) vypadají velmi pěkně a ještě plynuleji, na hranách (obrázek 4.2b) vznikají interpolací různé nežádoucí artefakty. Nehledě k tomu, že samotné procházení celé scény a vyhledávání sousednosti trojúhelníků je časově poměrně náročné (několik jednotek až desítek minut v závoslosti na počtu trojúhelníků). Lze však provést pouze jednou a výsledek uložit do souboru.

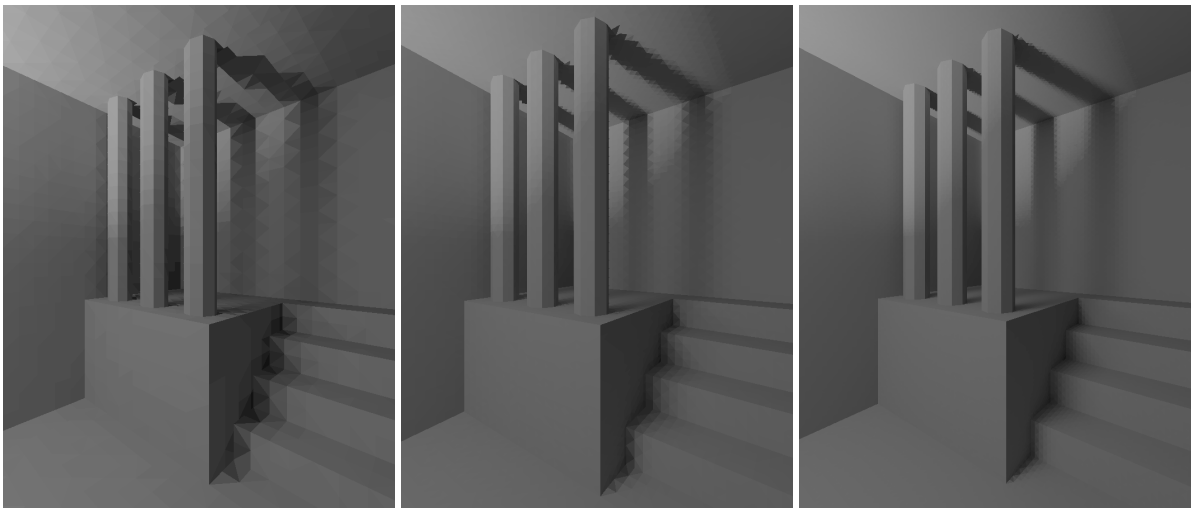
Ve finále bylo tedy od tohoto „vylepšení“ upuštěno a raději bylo vyzkoušeno nanesení textury na plochu a modulování s předpočítanou hodnotou osvětlení – výseldek překvapil svojí kvalitou. Textura zakryla přechody mezi trojúhelníky, které při normálním zobrazení pouze jednou barvou ostře vystupují a celek působil velmi dobrým dojmem. Přesvědčit se můžete sami na obrázku 4.3, kde byla na stěny velmi jednoduché krychle (malá testovací scéna zmíněná již výše v předchozích kapitolách) nanesena textura kamenné zdi a dovnitř svítí malý čtvereček. Texturování s sebou ovšem přináší další potíž – určení texturovacích souřadnic ke každému vrcholu. Při tomto pokusu byla tedy vytvořena druhá verze generátoru, který umí zadaný obdélník či trojúhelník dělit na menší včetně dělení a přepočítání texturovacích souřadnic. Počáteční texturovací souřadnice musejí být stejně jako souřadnice jednotlivých vrcholů zadány ručně.



Obrázek 4.3 – Textura modulovaná s osvětlením

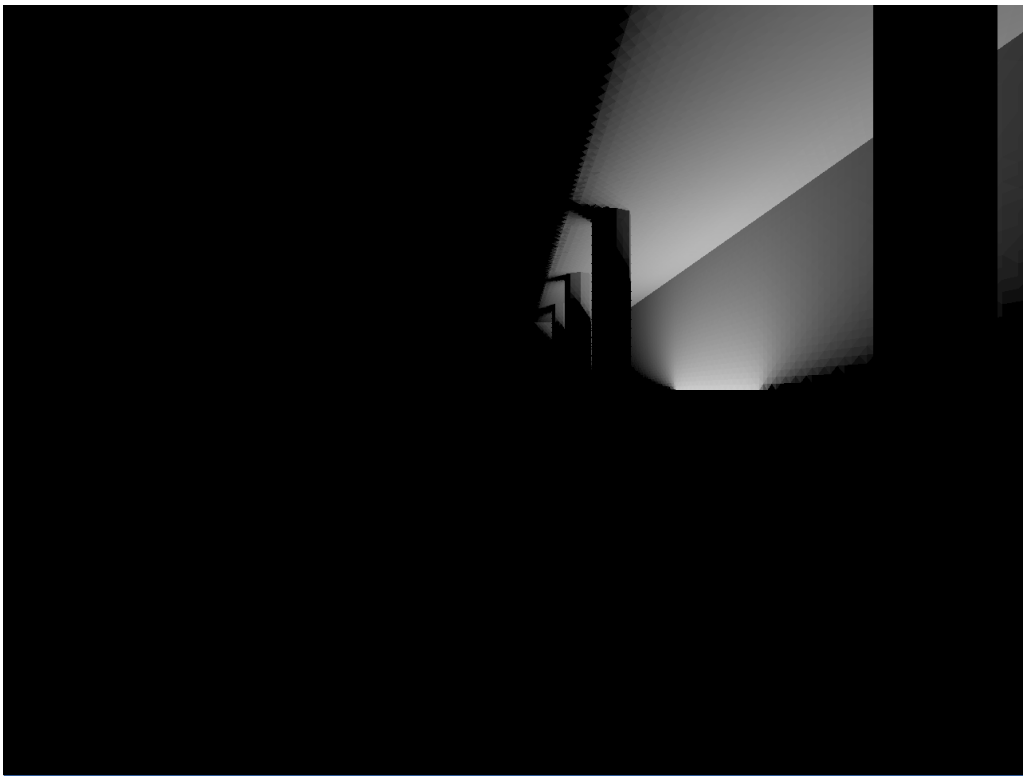
Jak jsme se zmínili již výše, na výslednou kvalitu má největší vliv kvalita modelu – především v místě ostrých přechodů. Na obrázcích 4.4a-c je vidět stejná část scény s různou hustotou trojúhelníků. Obrázek 4.4a má limit pro plochu trojúhelníku 5 jednotek, obrázek 4.4b 1 jednotku a obrázek 4.4c pouze 0,25 jednotky. Je zde také dobře patrné, že právě na kvalitu vykreslení zdí nemá počet trojúhelníků tak fatální vliv, jako na ostré stíny vržené sloupy.

Výhodou této metody je také to, že pěkně je vidět, jak se po jednotlivých itearcích světlo postupně šíří a přenáší se z plochy na plochu. Každá iterace je tedy samostatně zobrazitelná a taktéž se po kterékoli iteraci dá výpočet přerušit a poté znovu spustit. Jako atomická operace se bere jedna iterace, a to pouze z hlediska implementačního. Z hlediska teoretického je atomickou operací pouze vyřešení osvětlení jedné plošky v jedné iteraci. Tato možnost, kdykoli přerušit výpočet a data uložit a zobrazit, se velmi ocení zejména při ladění nebo renderování složitějších scén, kdy je potřeba výpočet přerušit z jiných důvodů. Ukázka, jak metoda postupně iteruje je na obrázku 4.5. Zároveň je patrné, že od druhé či třetí iterace u jednoduchých scén již nedochází k velikým změnám.



a) b) c)

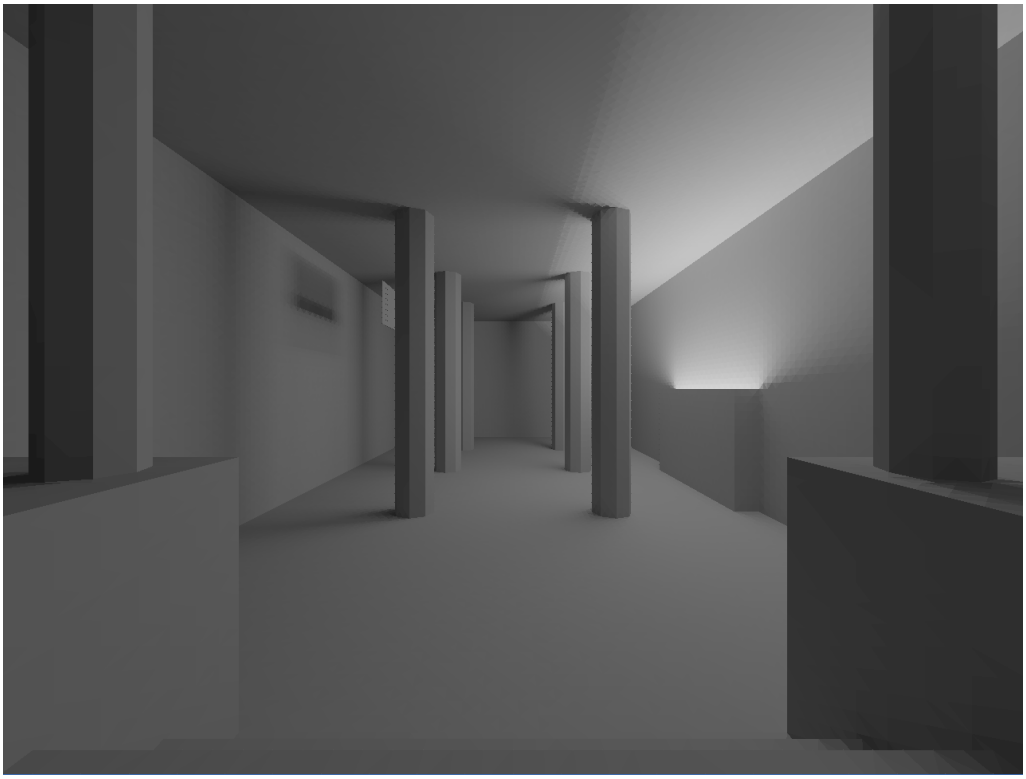
Obrázek 4.4 – Trojúhelníky s plochou do 5 (a), do 1 (b) a do 0,25 (c) jednotky



a)

Obrázek 4.5 – Iterace metody – první (a)

b)



c)

Obrázek 4.5 – Iterace metody – druhá (b), pátá (c) (z celkových 7 iterací)

Celkově jsem byl s výsledky spokojen; bohužel s vyřešením nějakého problému se objevují postupně další problémy a náměty na další řešení a vylepšení, jak už to u složitějších problémů (jakým problematika radiozity určit je) bývá. Závěrem této kapitoly bych tedy rád řekl, že výsledky jsou dobré, metoda prokázala svou životaschopnost, ale nasnadě je mnoho dalších námětů na vylepšení a materiálu k pokračování v započaté práci. Zároveň ne všechna řešení zde uvedená musejí být optimální a zcela jistě bychom řadu z nich mohli vylepšit.

4.1 Výpočetní náročnost

Jako jedna z dalších oblastí, které bychom se zde měli dotknout je oblast výpočetní náročnosti. Metoda díky využívání grafické karty spolu s procesorem pracuje poměrně svižně. Její rychlost však podstatnou měrou závisí právě na grafické kartě a to především na jejím hrubém výkonu v OpenGL. Nejvíce se zde totiž projeví, jak rychle je karta schopná zpracovávat obyčejné trojúhelníky – tedy řešit jejich viditelnost a rasterizaci. Některé věci, které nyní řeší procesor by se daly optimalizovat a přesunout například do shaderů grafické karty.

Pro představu zde uvedeme praktické výsledky na nijak výrazně neoptimalizovaném projektu. Testovací sestavou byl dvoujádrový Athlon X2 4600+ (každé jádro má takt jako jeden Athlon64 3800+) s 2048MB RAM paměti a grafickou kartou nVidia GeForce 7600GT s 256MB GDDR. Testovací scénou byla naše scéna popsaná na počátku kapitoly 4 a zobrazená na obrázcích této kapitoly (kromě obrázku 4.3). K jejímu výpočtu bylo potřeba 7 iterací. Vše je přehledně shrnuto do tabulky 4.1.

Tabulka 4.1

Viewport	546 048 troj.	136 512 troj.	26 360 troj.
100×100 px	583 min	58 min 20s	4 min 40s
200×200 px	700 min	81 min 40s	8 min 45s
500×500 px	1516 min	233 min	39 min 40s

Tato tabulka nám ukazuje několik zajímavých věcí. Bude však možná vhodné tyto věci separovat a vyčíslit je extra do další tabulky 4.2. V této tabulce 4.2 je čas přepočten na jeden trojúhelník scény.

Tabulka 4.2

Viewport	546 048 troj.	136 512 troj.	26 360 troj.
100×100 px	0,064s	0,026s	0,011s
200×200 px	0,077s	0,036s	0,020s
500×500 px	0,166s	0,102s	0,091s

Na tabulce 4.2 lze učinit několik zajímavých pozorování a z nich odvodit několik užitečných faktů. Přestože rozdíl mezi složitostí modelů je v počtu trojúhelníků 4-5 násobný, rozdíl v časech potřebných na zpracování jednoho trojúhelníku je pouze zhruba 2,5 násobek. Tento rozdíl se ale ještě více stírá, pokud zvětšujeme rozlišení rastru. Zde totiž hraje čím dál větší roli následné zpracování rastru, než jeho vygenerování.

Dalším jevem, který můžeme pozorovat, je fakt, že zvyšování rozlišení rastru zasáhne především jednodušší modely – čím složitější model, tím je rozdíl v časech menší … u jednoduchého modelu máme rozdíl mezi rozlišením 100×100 px a 500×500 px 9 násobek, zatímco u složitého modelu je to pouze 2,5 násobek.

Z obou učiněných pozorování tedy můžeme odvodit skutečnost, že vzhledem k výpočetní náročnosti a výsledné kvalitě vypočteného osvětlení se vyplatí mít složitější model s jemnějším rastrem (viewportem), neboť jeho výpočetní náročnost na trojúhelník a pixel je minimální. Tato statistika by se zřejmě mírně změnila díky některým optimalizacím výpočtu, nicméně základní trendy by zajisté zůstaly nezměněny.

4.2 Chyby a omezení

Každá metoda má své chyby a omezení, nic není zcela dokonalé. Pokusíme se tedy naznačit některé chyby a omezení, na která můžeme narazit u této metody. Probereme nejen faktické limity týkající se rozsahu číselných typů, se kterými se počítá, ale také jevy, kterým se z principu metody nelze vyhnout – řeč je zde zejména o aliasingu. Některé chyby a problémy byly již nastíněny výše (např. problém s ořezáváním před viewportem apod.), jiné rozebereme dále.

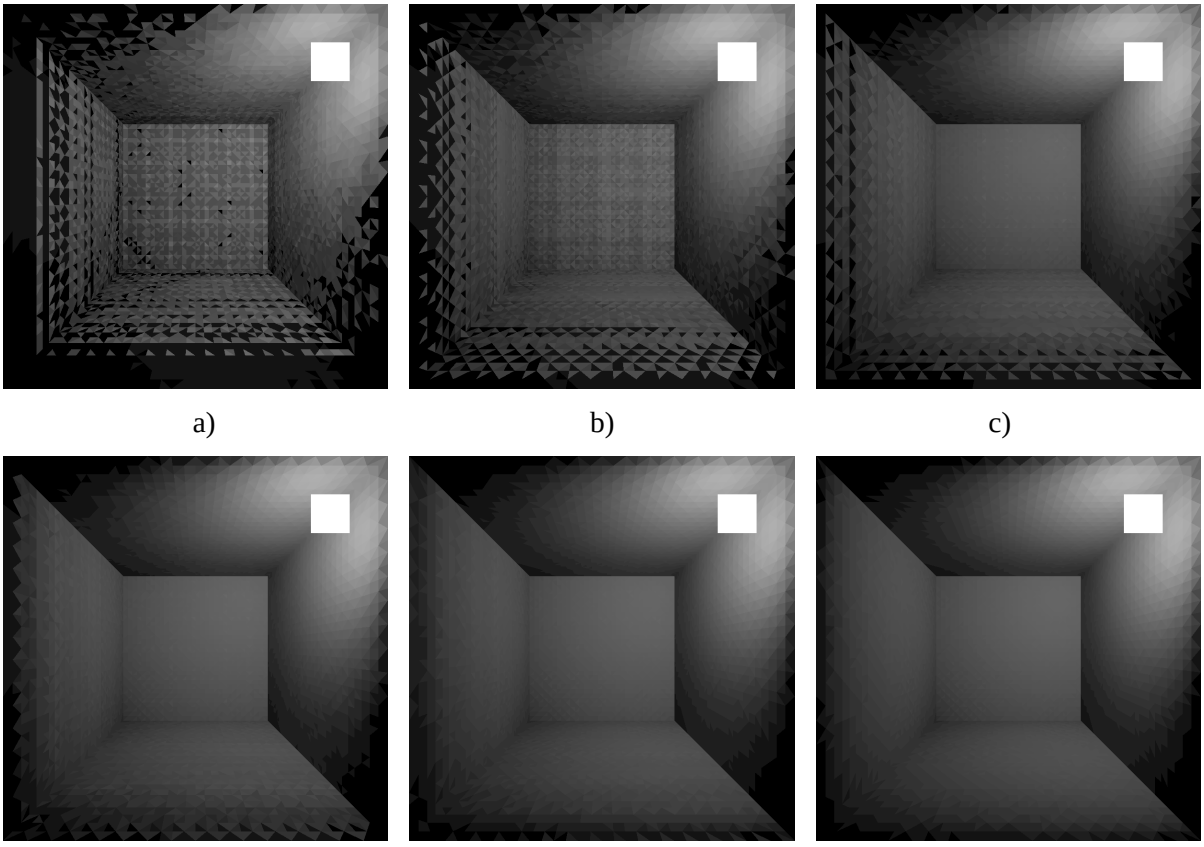
Proč jsme se tak podrobně zabývali výpočetní náročností při různých rozlišeních viewportu v předchozí kapitole si osvětlíme zde. Tato metoda je totiž založena na zpracování obrazu vygenerovaného ze scény v určitém rozlišení viewportu. Viewport neustále zabírá stejnou část scény a zobrazuje stejné objekty stejně veliké, ale pokaždé v jiném rozlišení a je tedy nutné tyto objekty do rastru navzorkovat. A jsme u základního problému vzorkování – aliasingu. Ten je u této metody problémem především ve dvou případech:

1. Malá ploška září do scény vysokým výkonem (rozlišení může být klidně větší)
2. Rozlišení je příliš malé (např. 100×100 px) a i větší objekty podléhají aliasingu

Problém se projevuje především na kvalitě jednolitých ploch, na kterých při přímém osvětlení vznikají artefakty způsobené tím, že i u vedle sebe položených plošek dochází díky aliasingu (světlo vyzařující ploška zabírá jednou 3px a jednou 7px dle rasterizace) k velkým rozdílům ve výsledném osvětlení.

Tento problém můžeme odbourávat dvojím způsobem – nejsnazší je zvednout rozlišení viewportu a počítat s větším rastrem a problém odsunout do roviny, kde již není tolik na závadu. Zde se však staví do protipólu výpočetní náročnost, která se zvedá. Dalším řešením může být užití antialiasingu při renderování, které by do značné míry mohlo při nižších rozlišeních pomoci. Chtělo by však provést testy, zda rychlost ušetřená ponecháním rozměrů viewportu bude stačit na zracování antialiasingu. Bylo by to zajímavé porovnání a vidím zde další výraznou možnost vylepšení této metody, je však nutné provést testy, co bude rychlejší – jestli zapnout AA nebo zdvojnásobit rozlišení viewportu. Bude také zajímavé porovnávat takto získané výsledky z hlediska kvality zobrazení.

Porovnání výsledků téže scény vypočtených s různým rastrem si můžeme prohlédnout na obrázku 4.6.



a) b) c)

d) e) f)

Obrázek 4.6 – různé rozměry rastru na téže scéně – 25×25px (a), 50×50px (b), 100×100px (c), 200×200px (d), 400×400px (e), 800×800px (f)

Za omezení můžeme pokládat omezený prostor pro vyjádření radiozity na ploškách. Jedná se o prostor celých nezáporných čísel na 24 bitech, jak bylo popsáno vyjádření radiozity výše. Toto omezení však není definitivní ani problematické – lze radiozitu vyjadřovat jinak apod. Navíc tento stavový prostor je většinou dostatečný – nejedná se tedy o problém, spíše o fakt, že takové omezení v této konkrétní implementaci existuje.

5 Závěr

Závěrem této práce mi dovolte malé zhodnocení metody jako celku z mého pohledu. Tato metoda (či spíše má implementace) není komplexním řešením – neřeší totiž již zmiňované adaptivní dělení modelu a některé další drobnosti. Tento problém však nezávisí na povaze metody a je možné jej doimplementovat, nepovažuji to tedy za vadu metody. Byl by to však podle mého názoru první ze směrů pokračování této práce. Dalším ze směrů pokračování bych určitě viděl možnost využití antialiasingu při renderování pohledů do scény – jak jsem se již však zmínil výše – tady je potřeba spíše testování, zda se využití AA vyplatí. Jedním ze směrů pokračování práce by zajisté byla optimalizace výpočtu. Výpočet tak jak je implementován je téměř neoptimalizovaný a je spíše ukázkou, než finálním řešením. Výhodné by totiž bylo např. využití shaderů (ať už vertex-shaderů pro zpracování vrcholů a hledání např. těžiště, nebo pixel-shaderů pro výpočet hodnot radiozity na pixelu či celé ploše viewportu) grafické karty k různým stupňům předzpracování, což by významně urychlilo výpočet. V dnešní době, kdy hrubý výpočetní výkon grafických karet (a jejich optimalizace pro výpočty s vektory) prudce stoupá a je násobně větší než obecněji zaměřený výkon hlavního procesoru počítače, by taková optimalizace zajisté přinesla významné urychlení. Optimalizace byla provedena alespoň ve využití displaylistů pro zobrazování scény (přinesla přibližně dvojnásobné urychlení celého výpočtu).

Za jednu z hlavních předností zvoleného přístupu považuji názornost výpočtu k pochopení radiozity a také možnost zastavit výpočet téměř kdykoli (zastavení je nyní implementováno pouze po iteracích, je možné to však změnit na možnost zastavení po každém zpracovaném trojúhelníku). Použití OpenGL má své výhody i nevýhody, ale v této fázi si myslím, že jeho jednoduchost a výhody převažují. Další předností metody může být poměrně malá paměťová náročnost – potřebujeme 72 bajtů paměti na jeden trojúhelník. Při dnešních běžných kapacitách paměti v řádu několika GB (osobně jsem implementaci prováděl na 2GB RAM) se jedná o možnost zpracovávat řádově miliony až desítky milionů trojúhelníků. Omezeni pak můžeme být ještě pamětí na grafické kartě, což v dnešní době také již není problém, neboť hodnoty již postačují a plně korespondují úměrně k velikostem hlavní RAM paměti. Nehledě k různým vnitřním optimalizacím jednotlivých grafických čipů.

Kapitolou, která také není uzavřená, je i zobrazování výsledků. Stejně jako problém adaptivního delení však stojí trochu bokem. Byly provedeny pokusy s interpolací, nanášením textur a některé se vydařily, jiné ne (viz kapitola 4). Vidím však v oblasti prezentace výsledků taktéž možnost zapracovat a zlepšit tak celkový dojem.

Jedna klasická poučka říká, že čím více člověk proniká do problematiky dané věci, tím více si uvědomuje, kolik toho ještě nezná a co všechno může ještě objevovat a učit se. Přesně tento pocit mám po vypracování této práce. Na počátku mého snažení jsem měl za cíl zkusit nějakou metodou

implementovat myšlenku radiozity. Hledal jsem metodu, která by byla jednoduše implementovatelná, aby bylo možno vyzkoušet si na ní konkrétní problémy, zobrazovat mezivýsledky, porovnávat apod. Po různém procházení stránek na internetu zabývajících se radiozitou jsem narazil na stránky Huga Eliase, který k radiozitě přistupoval pro mě velmi lákavým způsobem (i když výsledkem jeho výpočtu byla lightmapa, což je trochu jiný přístup). Rozhodl jsem se jeho přístup vyzkoušet a tak vznikla tato implementace radiozity. Narazil jsem pochopitelně na spoustu problémů, které se vyskytly v průběhu zpracování a souvisely ať už s povahou metody či mojí neznalostí věci. Otevřely se mi ale také obzory a směry, kudy se člověk může dále ubírat a za to jsem rád a to považuji za hlavní přínos pro sebe a také pro jiné. V průběhu práce jsem si posunul cíl své práce do trochu jiné roviny – nešlo mi o nějaké vzorové, nejlepší, nejkvalitnější nebo jiné nej- řešení, ale šlo mi o to, abych popsal a zpracoval radiozitu způsobem, který bude použitelný třeba pro další studenty či zájemce, kteří budou chtít tuto metodu rozšířit a vylepšit naznačenými cestami. Po přečtení této práce by měli být schopni nejenom se poučit a pochopit, jak vlastně radiozita pracuje, ale také získat tytéž praktické poznatky, které jsem učinil a řešil já. Nebudou tedy muset zacházet do uliček, které nikam nevedou – a to je velmi důležité, vědět kudy ne.

Literatura

[1] Žára, J. a kolektiv, *Moderní počítačová grafika*, Brno, Computer Press 2004

[2] Watt, A., *3D Computer Graphics*, London, Addison-Wesley, 2000

[3] Wikipedia, Radiosity, URL: http://en.wikipedia.org/wiki/Radiosity, prosinec 2006

[4] Hugo Elias, Radiosity, URL: http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm, prosinec 2006

[5] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the Interaction of Light Between Diffuse Surfaces. In SIGGRAPH ’84, Conference Proceeding, pages 212-222, July 1984

[6] T. Nishita and E. Nakamae. Continuous Tone Representation of Three-Dimensional Objects taking Account of Shadows and Interreflections. In SIGGRAPH ’85, Conference Proceeding, pages 23-30, July 1985

[7] M. F. Cohen and J. R. Wallace. Radiosity and Realistic Image Synthesis. Academic Press, 1993

[8] M. F. Cohen, D. P. Greenberg, D. S. Immel, and P. J. Brock, An Efficient Radiosity Approach for Realistic Image Synthesis. IEEE Computer Graphics and Applications 6(3):26-35, March 1986

[9] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A Progressive Refinement Approach to Fast Radiosity Image Generation. In SIGGRAPH ’88 Conference Proceeding, pages 75-84, August 1988

[10] P. Hanrahan, D. Salzman, and L. Appuerle. A Radpid Hierarchical Radiosity Algorithm. In SIGGRAPH ’91, Conference Proceeding, pages 197-206, July 1991

[11] P. H. Christensen, D. Lischinski, E. J. Stollnitz, and D.H. Salesin. Clustering for Gossy Global Illumination. ACM Transaction and Graphics, 16(1):3-33, January 1997

[12] E. J. Stollnitz, T. D. DeRose, and D. H. Salesin, Wavelets for Computer Graphics: Theory and Applications. Morgan Kaufmann, San Francisco, CA, 1996

[13] P. Shirley, A Ray tracing method for illumination calculation in diffuse-specular scenes. In Graphics Interface ’90, pages 205-212, May 1990

[14] různé stránky ze serveru http://softsurfer.com/, prosinec 2006

Seznam příloh

Příloha 1 – Manuál

Zdrojové soubory naleznete na přiloženém CD v adresáři `/src`. Přiložen je také soubor `makefile.windows`, kterým lze zkompilovat zdrojové texty příkazem `make` pod OS Windows. Pod jinými operačními systémy je třeba tento makefile modifikovat – místo knihoven OpenGL pro Windows tam přilinkovat knihovny vašeho OS. Pokud nemáte nainstalovánu knihovnu GLUT pro standardní OpenGL, potřebné soubory jsou přiloženy v adresáři `/glut`.

Po přeložení vzniknout tři spustitelné soubory (pro OS Windows jsou přeložené soubory v adresáři `/bin`) – `compute`, `display` a `genx`. Všechny tři soubory vyžadují pro svůj běh zadání parametrů. Při jejich spuštění bez náležitých parametrů se vypíše nápověda k použití.

Program genx slouží pouze k vygenerování modelu (scény) – není vlastně součástí řešení radiozity, je to pouze nástroj pro testování. Nejvhodnější by do budoucna bylo vytvoření knihovny, která by uměla zpracovávat nějaký 3D formát a převést jej do našeho formátu …. Program genx si vezme povinně dva parametry. Prvním je jméno souboru s výsledným modelem a druhý je číslo modelu (typ 1-3). Další 3 parametry jsou volitelné (musejí však být zadány všechny, nebo vůbec). Jedná se o specifikaci některých vlastností modelu:

`AREA_TRESHOLD`: určuje limitní velikost plochy trojúhelníku
`GENERAL_RO`: určuje obecnou odrazivost plošek (kolik z dopadnutého světla se odrazí zpět)
`LIGHT_POWER`: určuje absolutní výkon světla (k dispozici hodnoty 0 až 2^{24}).

Příklad: `genx model1.out 1 2.5 0.15 100000`

Další částí je program `compute`. Ten má povinné dva parametry – jméno vstupního souboru s daty scény a jméno souboru, kam má ukládat výsledky (může být ten stejný). Třetím nepovinným parametrem je rozměr viewportu v pixelech (default hodnota je 500px).

Příklad: `compute model1.in model1.out 200`

Poslední součást je program `display` pro zobrazení výsledků. Ten má jeden povinný parametr – jméno souboru se scénou a jeden nepovinný parametr, který může mít hodnotu „s“ nebo „c“. Jestliže chceme zobrazit vahlzené trojúhelníky, použijeme „s“, pokud chceme zobrazit scénu v barvě, jak ji vidí OpenGL při výpočtu, použijeme parametr „c“.

Příklad: `display model.out c`

Ukázková data, která jsou předpočítaná v rúzných iteracích jsou v adresáři `/demo`. Přiložen je tam také soubor `display.exe`, kterým lze všechny scény zobrazit (stejný jako v adresáři `/bin`). Jména souborů se skládají na prvním místě z typu modelu (output1-3), na druhém místě s limitu velikosti plochy plošky (0,1 ; 0,25 ; 1 ; 5) a na konci jména je stupeň iterace (1-iter, 2-iter nebo final).

Model 1 zahrnuje jednoduchou scénu - krychli, do které svítí čtvereček. Model 2 zahrnuje složitou scénu se sloupy a různými stíny. Model 3 je pak pouze jednoduchou ukázkou „lámání světla“ na překážce.