Univerza v Ljubljani

Fakulteta za strojništvo

Distribuirani sistem za upodabljanje tridimenzionalnih objektov

magisterij

Predložil Falulteti za strojništvo Univerze v Ljubljani za dosego akademske stopnje magistra znanosti s področja strojništva

Leon Kos

Mentor: izr.prof.dr. Jože Duhovnik, dipl.inž.

Ljubljana, 1995

Dne 20.6.1994 je Komisija za podiplomski študij fakultete za strojništvo v Ljubljani odobrila temo magistrske naloge in za mentorja imenovala izr.prof.dr.Jožeta Duhovnika, dipl.inž.

Zahvala

Za nekatere študente je lahko iskanje ustrezne teme za magisterij zapleten in dolgotrajen proces. Tako je bilo tudi v mojem primeru; In vendar sem z nasveti in pomočjo mentorja prof. Jožeta Duhovnika dobil zlato priložnost, da usmerja moje navdušenje za računalniško grafiko v reševanje trenutnih problemov upodabljanja tridimenzionalnih objektov z distribuiranim sistemom računalnikov.

Moje zanimanje za računalniško grafiko se ne bi zgodilo, če me ne bi prof. Duhovnik, še kot dodiplomskega študenta, s predavanji pri predmetu Računalniško podprto konstruiranje, seznanil z grafiko in mi omogočil delo na takrat malo kje uporabljanih delovnih postajah, kot tudi kasneje, ko je z neverjetno iznajdljivostjo omogočil, da je Laboratorij za računalniško podprto konstruiranje – LECAD dobil visoko zmogljivo računalniško opremo, potrebno za raziskave. Z razumevanjem, potrpljenjem in odprtostjo za mnoga področja mi je omogočil, da raziščem nova znanja, ki so bila povezana s temo moje naloge ali pa raziskave, s katerimi smo v laboratoriju dobili nova spoznanja in razumevanja.

Zahvala gre tudi sodelavcem v laboratoriju; Romanu Žavbiju, za pomoč pri študiju Umetne inteligence in administrativnih delih, Janezu Buhu, Tomažu Kolšku in Jožetu Tavčarju za nasvete in nesebično pomoč pri dnevnih problemih v in izven laboratorija.

Ne smem pozabiti tudi na Ministrstvo za znanost in tehnologijo, ki mi je v okviru programa mladih raziskovalcev dalo štipendijo, kot tudi Fakulteto za strojništvo, ki mi je ponudila ustrezno mesto za podiplomski študij.

Vesel sem tudi, da sem doživel razvoj projekta GNU, ki mi je omogočil, da sem se seznanil z mnogimi programskimi paketi, katere sem tudi večinoma uporabil pri svojem študiju. Tako sem se naučil programske jezike kot so C++, Prolog, Lisp, Postscript, Awk, Perl, LaTeX in druge. Brez razvoja operacijskega sistema Linux tudi ne bi dobil podrobnega vpogleda v delovanje sistema Unix in s tem ne uspešnega zaključka moje magistrske naloge.

Omeniti moram tudi dve referenci, brez katerih bi težko začel z delom na nalogi. Knjiga “Power Programming with RPC”[Blo92] avtorja Johna Bloomerja, me je seznanila s prenosljivo knjižnico za distribuirano programiranje. Najbolj vplivno delo pa je bila knjiga skupine avtorjev “An Introduction to Ray Tracing” [Gla91b], ki je zbistrila moje zgodnje razumevanje metode sledenja žarku in fizikalnih modelov odboja svetlobe na površini.

koda, tekoča št.: M110/939 UDK 519.688:681.3.066(043.2)

Distribuirani sistem za upodabljanje tridimenzionalnih objektov

Leon Kos

Ključne besede: distribuirani sistemi
paralelni algoritmi
sledenje žarku
računalniška grafika
Parallel Virtual Machine – PVM
Remote Procedure Calling – RPC

Izvleček:

Upodabljanje v računalniški grafiki je v zadnjih letih pridobilo na pomembnosti na različnih področjih uporabe. Za upodabljanje se uporabljajo različni algoritmi, ki so običajno kompromis med hitrostjo in realističnostjo prikaza. Večino algoritmov je možno enostavno paralelizirati. Na paralelnih sistemih je tako možno računati kompleksne scene v razumnem času.

V delu je predstavljena teoretična formulacija generiranja realistične slike z distribuiranim sistemom v mreži računalnikov. Prikazane rešitve se lahko uporabijo v upodabljanju, kot tudi na drugih področjih računsko zahtevnega procesiranja.

Upodabljanje tridimenzionalnih objektov z metodo sledenja žarku (*ray tracing*) je bilo prirejeno in implementirano na lokalni mreži delovnih postaj in osebnih računalnikov. Sistem PVM (*Parallel Virtual Machine*) je bil uporabljen kot osnova distribuiranega programa z balansiranjem obremenitve na računalnikih, ki imajo zelo različne procesorske moči. V delu so prikazani hitrostni testi, kot tudi poglobljen pregled vseh faz razvoja distribuiranega tridimenzionalnega sistema.

code, No.: M110/939 UDK 519.688:681.3.066(043.2)

Rendering on distributed systems
by
Leon Kos

Keywords: distributed systems
parallel algorithms
computer graphics
ray tracing
Parallel Virtual Machine – PVM
Remote Procedure Calling – RPC

Abstract:

Rendering in computer graphics has matured and gained importance in various areas of applications. Several different methods can be used for rendering. Current realistic methods in real applications solve problems only to certain degree which is compromise between speed and complexity. Many time consuming algorithms can be rewriten for use on parallel systems. Parallel programs will run faster, enabling users to render more complex scenes than before.

This thesis presents a theoretical formulation of realistic image synthesis with distributed system on local area network. Presented solution techniques can be aplied in computer graphics and in other computational intensive problems.

Rendering with ray tracing has been adapted and implemented on nework of workstations. PVM system (Parallel Virtual Machine) has been used as basis of the load balancing distributed rendering program, running on systems with different processor power. Thesis presents speed tests and also overview of all development steps needed to build distributed rendering system.

Kazalo

Slike

Tabele

Poglavje 1

Uvod

V zadnjih desetih letih je računalniška grafika iz laboratorijev univerz pričela zahajati v vsakdanja področja uporabe. Srečamo jo pri vizualizaciji v medicini, kot orodja CAD v strojništvu, arhitekturi in drugod. Stalni cilj večine raziskav v razvoju je generiranje realističnih prikazov objektov z računalnikom. Metoda sledenja žarku (*ray-tracing*) je ena od pomembnih pristopov pri reševanju modela osvetlitve, ki je ključna naloga pri generiranju realistične slike.

Paralelizacija je ena od najbolj obetajočih smeri k povečanju zmogljivosti, v aplikacijah CAD, kot tudi v drugih področjih uporabe. To je še posebno res za področje računalniške grafike.

Za razliko od drugih naporov v računalniški grafiki, je v tem delu poudarek na integraciji računalniške grafike in paralelizacije. Paralelizacija v računalniški grafiki je možna na različnih področjih, kot so:

- **Geometrijsko modeliranje**: Algoritemske rešitve problemov pri prikazu geometrijskih objektov (npr. Izračunavanje potovanja orodja za NC stroje). Veliko algoritmov na področju geometrijskega modeliranja je računsko zelo intenzivno in primerno za paralelizacijo.

- **Upodabljanje in animacija**: Vizualizacija geometrijskih objektov na optičnih izhodnih napravah. Večino metod v upodabljanju je možno direktno paralelizirati, brez večjih težav s komunikacijo.

- **Obdelava slik**: Pomembno področje, ki zajema izboljšavo kvalitete slik, različne efekte za popačitve, kot tudi korekcije v osvetlitvi.

- **Razpoznavanje vzorcev in gledanje**: Avtomatsko generiranje geometrijskih objektov v računalniku iz dvodimenzionalnih slik. Razpoznavanje je računsko zelo intenzivna operacija (npr. OCR - *Optical Character Recognition*).

- **Vizualizacija**: Splošen pojem s katerim zajamemo grafični prikaz informacij aplikacije, največkrat rezultate obdelave kakšnega znanstvenega problema.

1.2 Pregled magistrskega dela

Upodabljanje v računalniški grafiki se je razširilo na mnoga področja praktične uporabe. Razvite so bile različne metode upodabljanja, ki se razlikujejo po algoritmični zahtevnosti in stopnji realističnosti. V poglavju 2 so prikazani osnovi principi pri obravnavanju interakcije svetlobe in površine, kot osnove pri matematičnem modeliranju objektov z računalnikom. Prikazana je razlika med fizikalnim in geometrijskim modelom svetlobe. Obravnavane so optične lastnosti različnih medijev in osnovni zakoni, ki jih lahko uporabimo pri modeliranju nekaterih površin. Večine površin, ki jih v naravi srečamo, ne moremo enostavno opisati z zakoni optike, zato je potrebno uporabiti ustrezne empirične enačbe, ki so največkrat kompromis med realnimi modeli in računsko zahtevnostjo osvetlitve na površini.

V poglavju 3 so podrobneje predstavljeni modeli osvetlitve na površini in osnovni zakoni, ki jih lahko apliciramo v empiričnih modelih. Splošni model osvetlitve mora zajemati vse možne načine prenosa svetlobe, kot tudi lastnosti površine in transportnega medija pod površino. Za pravilnejše obravnavanje odboja svetlobe lahko uporabimo model Cook-Torrance, ki ima za osnovo poenostavljen fizikalni model površine. Modeli osvetlitve, ki temeljijo na fizikalnih lastnostih površine tudi bolje modelirajo odbleske, ki jih običajno vidimo na kovinah. Ustrezen model osvetlitve je osnova za različne metode upodabljanja.

Poglavje 4 nas seznani z metodo sledenja žarku (*ray tracing*), ki je ena od metod s katero se da doseči tridimenzionalne prikaze modeliranih objektov, z visoko stopnjo realističnosti. Osnovni algoritem je razmeroma enostaven in že dolgo uporabljan v računalniški grafiki. Žal pa se, tako kot je praksa tudi na drugih področjih, z večanjem zahtev po realističnosti, nesorazmerno veča tudi čas, potreben za izračune scene. Čeprav je možno metodo sledenja žarku optimirati in s tem zmanjšati računsko zahtevnost z različnimi prijemi, je v tem poglavju predstavljena le osnovna metoda, ki prikaže delovanje programa sledenja žarku. Večanje zahtev po realističnosti avtomatsko tudi poveča potrebni čas za izračun, tako da je za demonstracijske namene v bistvu nepomembno, do katere stopnje je sistem za upodabljanje izveden, če želimo prikazati enostavno povečanje računskih zmogljivosti s paralelizacijo obstoječih algoritmov.

Osnovni pojmi s področja paralelnih algoritmov in distribuiranega procesiranja so prikazani v poglavju 5. Predstavljeni so različni sistemi za razvoj programov na omrežju heterogenih računalnikov. Različni problemi imajo v splošnem različne prijeme pri pretvarjanju sekvenčnih algoritmov v paralelne. V arhitekturah, kjer si podatke računalniki izmenjujejo po omrežju, je potrebno upoštevati tudi hitrost prenosa podatkov po komunikacijskem mediju. Pomembna je tudi enakomerna porazdelitev dela na vse računalnike, ki sodelujejo pri reševanju problema.

Osnovne strategije komuniciranja med računalniki so omejene na transportni medij. V arhitekturah, kjer hitrost komunikacije ne predstavlja bistvene omejitve, so se uveljavili tudi drugi načini komuniciranja med osnovnimi računskimi enotami.

1.3 Prispevki magistrskega dela

Osnovni prispevki predstavljenega tega dela so:

- prikazan je podroben opis korakov, potrebnih za izdelavo osnovnega sistema za upodabljanje z metodo sledenja žarku. Diskutirane so različne metode s katerimi lahko nadgradimo osnovni sistem, tako v hitrosti, kot tudi v realističnosti prikaza računalniško modeliranih objektov.
- pregled različnih načinov komunikacije med procesi v računalniku, kot tudi na omrežjih.
- primerjava in razlaga različnih pristopov k paralelizmu. Distribuirani model na mreži delovnih postaj in osebnih računalnikov je pokazal uporabnost pristopa v reševanju numerično intenzivnih problemov, ne samo v računalniški grafiki, ampak tudi aplikativne možnosti v drugih področjih računsko zahtevnih problemov.
- razširitev obstoječega programa za upodabljanje na distribuirani sistem z avtomatskim uravnoteženjem na omrežju računalnikov z zelo različnimi procesorskimi močmi[4]. V obstoječi literaturi je bilo možno dobiti le relativne zmogljivostne teste distribuiranih sistemov, ki ne diskutirajo podrobneje komunikacijske potrebe zahtevnih scen.
- primerjava zmogljivosti različnih arhitektur pri implementaciji sistema za upodabljanje. Diskutirane so možnosti racionalnejše porabe računskih moči večjega števila delovnih postaj, brez dodatnega investiranja v drago namensko opremo.
- razvoj programa za sledenje žarku v objektno orientiranem programskem jeziku, z abstraktnimi objekti, ki omogočajo enostavno nadgradnjo z novimi geometrijskimi objekti ali osvetlitvenimi modeli.
- skupaj z aplikativnim delom so bili razviti razni pomožni programi, ki omogočajo sočasno programiranje in pisanje dokumentacije v hipertekstni obliki.

[4]Razmerje med najzmogljivejšo delovno postajo HP 735 in najmanj zmogljivim osebnim računalnikom PC/i486/66 je bilo preko 1:7

Poglavje 2

Odboj in lom svetlobe na površini

2.1 Fizikalna in geometrijska optika

Skoraj vsak grafični program modelira optiko na nek način. Največkrat se tu uporabijo različne poenostavitve v fizikalnem modelu optike.

Fiziki opisujejo svetlobo kot elektromagnetno valovanje ali kot gibajoče se delce energije (fotone) [BW89]. Valovanje je opisano z valovanjem električnega in magnetnega polja, ki sta si med seboj pravokotni. Model optike zasnovan na valovanju se imenuje *fizikalni model*, model temelječ na premočrtnem gibanju delcev pa *geometrijski model svetlobe*. Slednji model, z delcem kot nosilcem energije je tudi enostavnejši in zato tudi večkrat uporabljan.

Fizikalni model je možno poenostaviti tudi tako, da svetlobe ne obravnavamo kot vektorsko valovanje, ampak le kot skalar z določeno energijo. Seštevanje je tako veliko lažje, vendar je kljub temu geometrijski model v računalniški grafiki skoraj v izključni uporabi.

Z uporabo geometrijskega modela modeliramo svetlobo kot *neodvisne žarke*. V geometrijskem modelu lahko zanemarimo nekatere pojave, kot so uklon svetlobe, interferenca in polarizacija, zato je potrebno te lastnosti modelirati posebej.

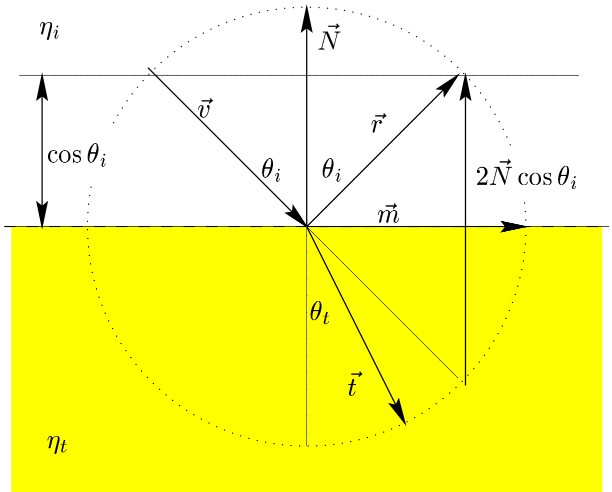
Uklon svetlobe nastopi pri širjenju svetlobe v bližini ovire. Modeliranje uklona je v geometrijskem modelu izredno težavno, zato so do sedaj razvite le metode za posebne primere, za katere je možno določiti gladke prehode senc v t.i. *umbri in penmbri*. Frekvenca valovanja vidne svetlobe je tudi dokaj visoka, zato lahko modeliranje teh podrobnosti največkrat opustimo.

Interferenca je povezana z zamikom faze valovanja svetlobe, ki nastopi zaradi uklona, večkratnega loma ali odboja svetlobe. V naravi primere interference opazimo pri milnih mehurčkih, kjer se pojavijo raznobarvni pasovi. Podobni efekti so vidni tudi na vseh tankih prosojnih plasteh, kot so npr. tanka zaščitna plast filma na fotografskih lečah, plast olja na vodi...

Polarizacija je v geometrijskih modelih najmanj upoštevana, tudi zaradi tega, ker jo je v praksi težko neposredno zaznati. V primerih natančnega modeliranja nekaterih pojavov v atmosferi [TTOO90], pa lahko tudi upoštevanje polarizacije prispeva k boljšemu

2.3.1 Snellov zakon

Ko svetloba zadene prozorno gladko površino, se del svetlobe odbije, preostali del pa prodre v snov. Odboj in lom svetlobe na stiku dveh snovi z lomnima količnikoma η_i in η_t je prikazan na sliki 2.1.



Slika 2.1: Geometrija žarka, ki zadene gladko površino

Smer odbite svetlobe $\vec{t}$ podaja *zakon zrcalnega odboja*:

> Kot odboja je enak kotu vpadnega žarka glede na normalo površine $\vec{N}$.

Pri implementiranju grafične simulacije je potrebno to zakonitost formulirati v uporabni obliki za numerično obdelavo

$$\vec{r} = \vec{v} + 2\vec{N}\cos\theta_i = \vec{v} - 2\vec{N}(\vec{v}\cdot\vec{N}) \tag{2.1}$$

Smer žarka $\vec{t}$, ki potuje skozi površino, pa je podan s Snell-ovim zakonom.

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t \tag{2.2}$$

ali zapisano drugače

$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{\eta_t}{\eta_i} \tag{2.3}$$

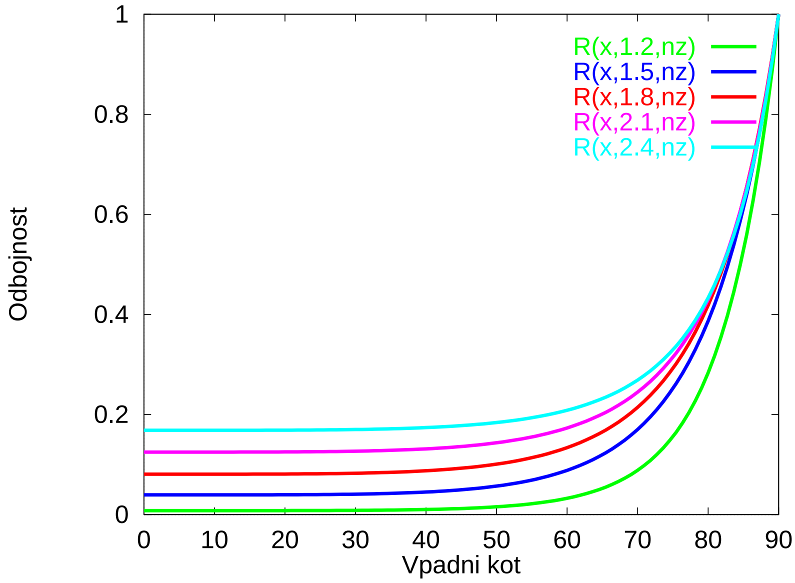
kjer je η_i lomni količnik snovi vpadnega žarka in η_t lomni količnik snovi žarka, ki prehaja.

Izračun $\vec{t}$ za numerično obdelavo je bolj zahteven . Iz slike 2.1 lahko vidimo, da je

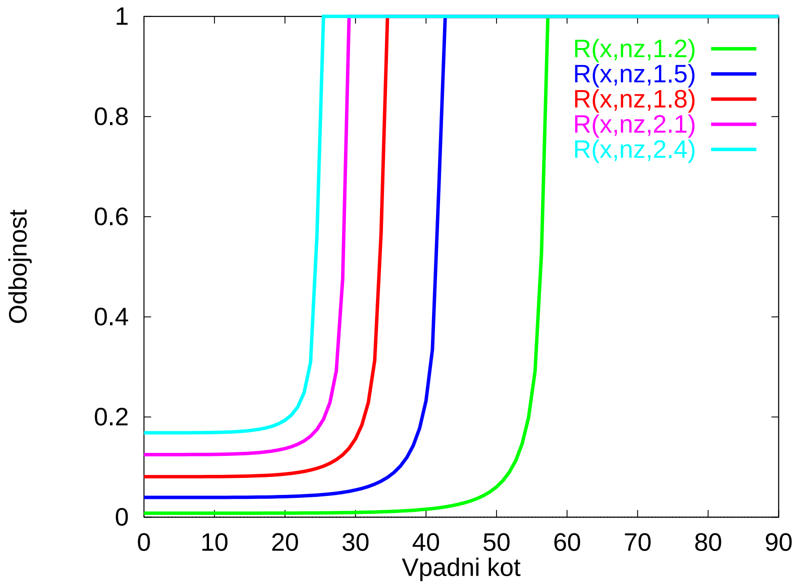
$$\vec{t} = -\vec{N}\cos\theta_t + \vec{m}\sin\theta_t \tag{2.4}$$

kar vodi k enačbi [Hec91]:

$$\vec{t} = \left[-\frac{\eta_i}{\eta_t}(\vec{v}\cdot\vec{N}) - \sqrt{1 - \left(\frac{\eta_i}{\eta_t}\right)^2 (1 - (\vec{v}\cdot\vec{N})^2)} \right] \vec{N} + \frac{\eta_1}{\eta_t}\vec{v} \tag{2.5}$$



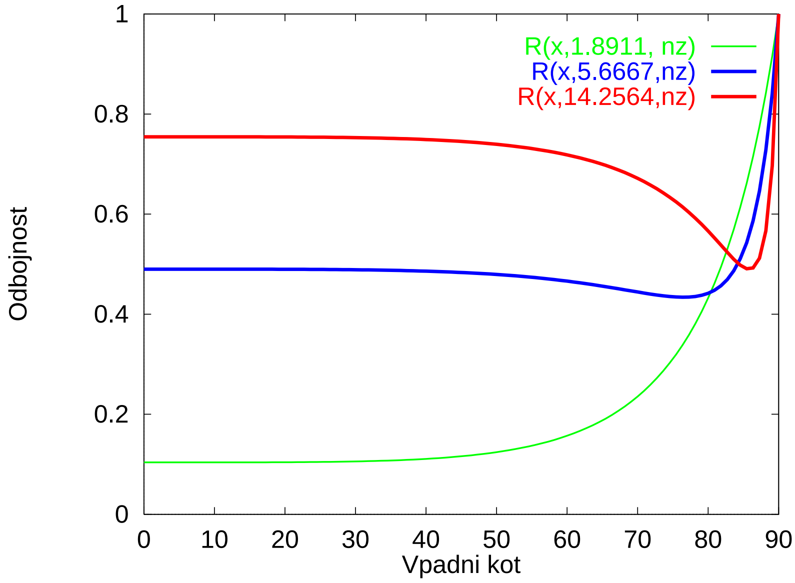
Slika 2.2: Odbojnost pri prehodu iz zraka v dielektrik z različnimi lomnimi količniki. Najvišja krivulja ima lomni količnik 2.4. Krivulje so izračunane direktno iz Fresnelovih enačb.



Slika 2.3: Odbojnost pri prehodu dielektrika z različnimi lomnimi količniki v zrak. Najvišja krivulja ima lomni količnik 2.4. Krivulje so izračunane direktno iz Fresnelovih enačb.

Primarna barva	R_0	η
Rdeča	0.755	14.2564
Zelena	0.490	5.6667
Modra	0.095	1.8911

Tabela 2.2: Optične lastnosti bakra v geometrijskem modelu



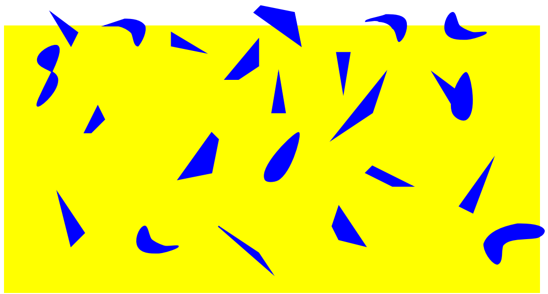
Slika 2.4: Odbojnost primarnih barv v geometrijskem modelu glajenega bakra. Rdeča barva je najvišja krivulja; modra najnižja. Krivulje so izračunane direktno iz Fresnelovih enačb.

2.4 Hrapave površine

Le malo je površin v naravi, za katere bi lahko trdili, da so popolnoma gladke. Za take površine ne moremo direktno uporabiti Fresnelovih enačb. Problematika hrapavih površin ni domena samo računalniške grafike ali optike, saj lahko bistvene zakonitosti najdemo tudi v literaturi, ki obravnava prenos toplote s sevanjem [Thy92, Bej93]. Različne oblike in mikro modeli hrapave površine vplivajo tudi na implementacijo modelov odboja. Primer hrapave površine, katero moramo obravnavati na mikro nivoju, je prikazan na sliki 2.5.

Pri obravnavanju in formuliranju matematičnega modela je potrebno upoštevati normale mikropovršin, kot tudi povprečje in absorbcije zaradi hrapavosti. Podrobnejše obravnavanje matematičnih modelov je prikazano v poglavju 3.

Absorbcija svetlobe pa se pojavlja tudi v snoveh, ki imajo gladke površine. Nivo prepustnosti (*transparentnost*) je odvisen od stukture snovi. V dielektrikih poznamo izotropno razporeditev absorbcijskih molekul (npr. barvasto steklo). Kovine v celoti absorbirajo površina. Če je površina tudi gladka, potem se del svetlobe normalno odbija in s tem delno deluje kot zrcalo.



Slika 2.7: "Mat" površina kompozitne snovi

Nekatere barve pa na površini nimajo gladkega sloja in s tem tudi ne spoznavnega učinka zrcala. Take površine imenujemo *mat površine*. Za razliko od sijajnih površin imajo take barve delce pigmenta tudi na površini osnovne snovi. Pri sušenju nekaterih barv je tudi opaziti spremembo lomnega količnika, kot tudi tip površine, ki lahko prehaja iz sijajne v mat tudi s staranjem.

2.6 Prosojne površine

Za razliko od prozornih površin, prosojne le delno prepuščajo svetlobo. Klasičen primer prosojne površine je papir. Snovi iz prosojnih površin imajo nehomogeno strukturo z množico naključno razporejenih delcev, ki razpršujejo svetlobo na vse možne strani. Če je snov dovolj tanka, lahko prepušča svetlobo z ene strani površine na drugo. Na drugi strani torej svetloba seva v okolico, kar moramo dodatno upoštevati pri modeliranju razmer na površini.

2.7 Povzetek

Za računalniško grafiko je potrebno poznavanje optičnih lastnosti površine. Svetlobo lahko obravnavamo kot valovanje (fizikalna optika) ali kot žarke gibajočih se delcev (geometrijska optika). Geometrijska optika je lažja za modeliranje v računalniški grafiki zaradi medsebojne neodvisnosti žarkov. S privzemom geometrijske optike izgubimo lastnosti fizikalne optike kot so uklon, interferenca in polarizacija svetlobe.

Optična lastnost površine je podana z materialom iz katerega je narejena. V računalniški grafiki je vsaka površina opisana z opisom medija, ki je zbirka posameznih lastnosti materiala.

Za gladke površine lahko s Fresnelovimi enačbami dokaj natančno opišemo, kako se žarki odbijejo in lomijo v prozorno površino. Odbojnost je razmerje med intenziteto od-

Poglavje 3

Modeli osvetlitve na površini

V poglavju 2 smo si ogledali osnovne zakonitosti svetlobe pri interakciji s površino. Za matematično modeliranje razmer na površini je potrebno uvesti ustrezne poenostavitve, ki so običajno kompromis med točnostjo modela in potrebnim računskim časom. Tako je zaradi zahtevnosti in potrebne količine podatkov, moč v programih z interaktivnim dostopom najti le najenostavnejše modele odboja. Taki modeli imajo običajno vgrajen le en točkast vir svetlobe v neskončnosti in Lambertov model difuznega odboja na površini.

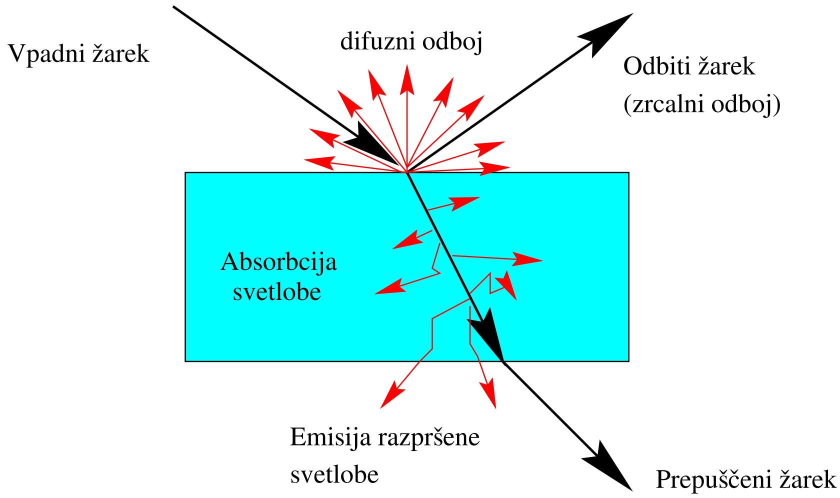
Z zahtevami po realističnih slikah raste tudi točnost modelov in računski čas, ki pa ga lahko s skupino in namenskimi procesorji zmanjšamo na zadovoljivo raven. Kljub temu, pa razmeram na površini ne smemo dati prevelike pomembnosti, saj so lahko za upodabljanje bistvenejši drugi efekti realističnosti, kot so teksture in moduliranje globine na površini.

Ugodno je tudi, če lahko kontroliramo stopnjo realističnosti. V modelirniku za CAD običajno ne želimo velike kvalitete izrisa ampak hitrost. Grafične delovne postaje imajo za interaktivno delo vgrajene posebne grafične procesorje, ki zmorejo v realnem času osenčiti nekaj tisoč mnogokotnikov. Zaradi vgrajene strojne opreme se lahko zgodi tudi paradoks, da modelirnik deluje hitreje z osenčenimi, kot pa z žičnimi modeli.

Ko je potrebno izdelek predstaviti, pa se zahteva čimbolj verna predstavitev, ki največkrat poudarja odbleske na gladkih površinah, teksture in fraktalna ozadja. Če se zahteva še animacija, je lahko potreben čas za izračun vseh sekvenc tudi nekaj dni na enem samem procesorju.

Svetlobo lahko količinsko opredelimo na več načinov [Ash94, CW93]. Ločimo predvsem dva pristopa k obravnavanju svetlobe: Radiometrija je znanost fizikalnega merjenja elektromagnetne energije; Fotometrija podaja oceno intenzitete svetlobe v enotah, ki upoštevanjo občutljivost človeškega očesa. Pomembno je predvsem to, da je edina razlika med fotometrično teorijo in teorijo sevanja le v enotah merjenja. V večini grafičnih sistemov so optične količine podane s trojico barv: rdeča, zelena in modra (*RGB*).

Za enostavno kontrolo objektov v sceni lahko uporabimo model, ki predpostavlja, da objekti sevajo svetlobo sami od sebe. Tako v barvnem modelu RGB podamo svetilnosti za rdečo, zeleno in modro barvo. Podobno je možno uporabiti tudi druge barvne modele,



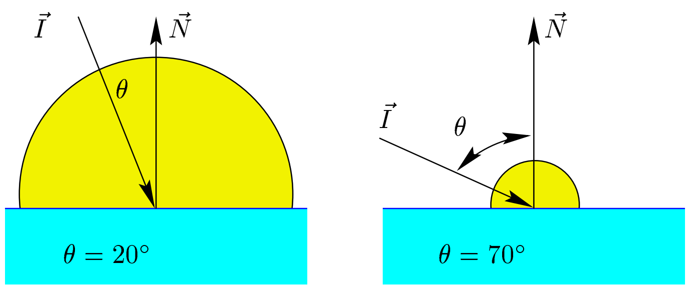
Slika 3.1: Različni načini prenosa svetlobe pri interakciji svetlobe s trdnim telesom

3.1.1 Difuzni odboj

Čeprav so odbojne lastnosti površin zelo različne, lahko gladke motne površine obravnavamo kot idealne difuzne površine. Idealizirane *motne* površine včasih imenujemo tudi *Lambertove* površine. Za njih velja Lambertov zakon

$$I_d = I_i k_d \cos\theta \tag{3.3}$$

Enačba (3.3) je zadovoljiva aproksimacija za veliko večino motnih površin. Barva takih površin se s spremembo kota gledanja ne spremeni bistveno, saj se vpadni žarek I na površini razprši enakomerno na vse strani. Intenziteta svetlobe (slika 3.2) je sorazmerna kosinusu kota med normalo površine in vpadnim žarkom.

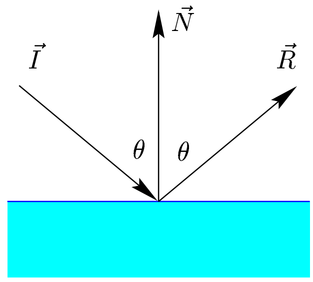


Slika 3.2: Razpršitev svetlobe na motni površini

Matematični model za računalniško grafiko je hitrejši, če v enačbi (2.6) zamenjamo kosinus s skalarnim produktom normale površine $\vec{N}$ in smeri vpadnega žarka $\vec{L}$ oziroma smeri luči L.

$$I_d = I_i k_d (\vec{L} \cdot \vec{N}) \tag{3.4}$$

Gladke površine, kot so kovine, stekla, ipd., odbijajo svetlobo pod istim kotom, kot je vpadni kot svetlobnega žarka. To dejstvo absolutno velja za popolno zrcalo, pri katerem je površina tako gladka, da se vsa svetloba odbije. Če je površina prekrita s tanko plastjo, se del svetlobe lahko tudi absorbira v plasti. Primer take površine je navadno stekleno zrcalo iz stekla . Svetloba se tu deloma absorbira v steklu, večji del pa se odbije. Podoben primer je tudi plast iz polirne paste, pri kateri se večina svetlobe odbije že na plasti, preostali delež pa se lomi in nato absorbira.



Slika 3.4: Popolni zrcalni odboj

Dvosmerno odbojnostno funkcijo $R_{bd}(\theta_{in}, \phi_{in}, \theta_{out}, \phi_{out})$ lahko zapišemo kot kombinacijo dveh osnovnih komponent: difuznega in zrcalnega odboj. Za zrcalni odboj je značilen odboj pod istim kotom, kar kaže slika 3.4. Prikazano lastnost zapišemo kot

$$(\theta_{in}, \phi_{in}) = (\theta_{out}, \phi_{out}) \tag{3.8}$$

Smer odboja žarka je smotrno formulirati v obliki, ki je primerna za računalniško obdelavo. Odbiti žarek $\vec{R}$ zapišemo kot linearno kombinacijo vektorjev $\vec{N}$ in $\vec{I}$

$$\vec{R} = \alpha\vec{I} + \beta\vec{N} \tag{3.9}$$

kjer sta α in β zaenkrat še neznani konstanti in ju določimo z upoštevanjem zakonitosti podani v enačbi (3.8). V dveh dimenzijah je vpadni kot θ_i enak odbitemu kotu θ_r. S tem sta enaka tudi kosinusa

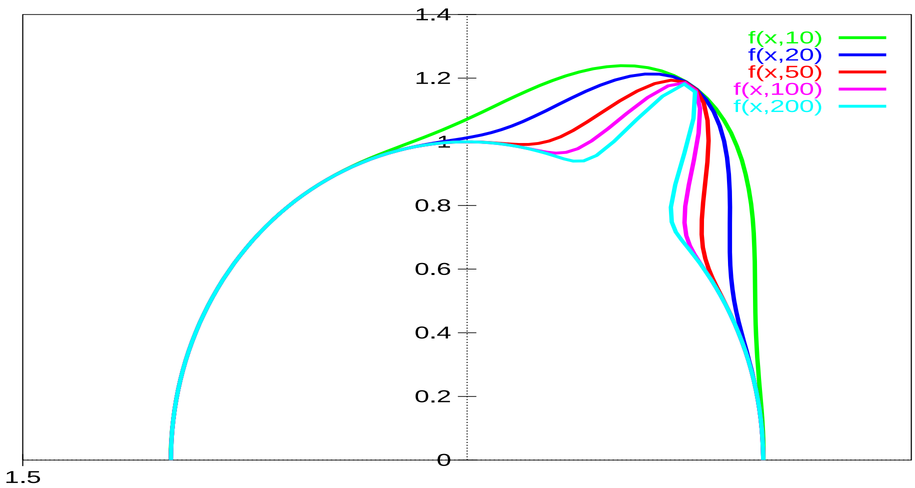
$$\cos\theta_i = \cos\theta_r \tag{3.10}$$

Enačbo (3.10) lahko zapišemo tudi s skalarnim produktom in pri tem določimo konstanti α in β.

$$\begin{aligned} -\vec{I}\cdot\vec{N} &= \vec{N}\cdot\vec{R} \\ -\vec{I}\cdot\vec{N} &= \vec{N}\cdot(\alpha\vec{I}+\beta\vec{N}) \\ -\vec{I}\cdot\vec{N} &= \alpha(\vec{N}\cdot\vec{I})+\beta(\vec{N}\cdot\vec{N}) \end{aligned} \tag{3.11}$$

Če je vektor nomale enotski $|\vec{N}| = 1$ velja $(\vec{N}\cdot\vec{N}) = 1$; Lahko izberemo $\alpha = 1$ in izrazimo konstanto β kot

$$\beta = -2(\vec{N}\cdot\vec{I}) \tag{3.12}$$



Slika 3.6: Odboj svetlobe kot funkcija kota gledanja $\vec{V}$ in konstante n za primer difuznega in zrcalnega odboja $I(\theta, n) = 1 + 0.4\ cos^n(\theta - 1)$

ki je povprečje vektorjev $\vec{V}$ in $\vec{L}$, je v položaju, ko je $\vec{V} = \vec{R}$ vektor $\vec{H}$ enak vektorju $\vec{N}$. Z uporabo vektorja $\vec{H}$ ni potrebno izračunavati $\vec{R}$, če uporabimo funcijsko odvisnost skalarnega produkta med vektorjema $\vec{H}$ in $\vec{N}$. Enačbo (3.15) uporabimo za vzor nove enačbe, pri kateri je potrebno uporabiti dvakrat manjše potence n za isto porazdelitveno funkcijo.

$$I_{rd} = k_s(\vec{N} \cdot \vec{H})^n \tag{3.17}$$

3.1.4 Kombinacija difuznega in zrcalnega odboja

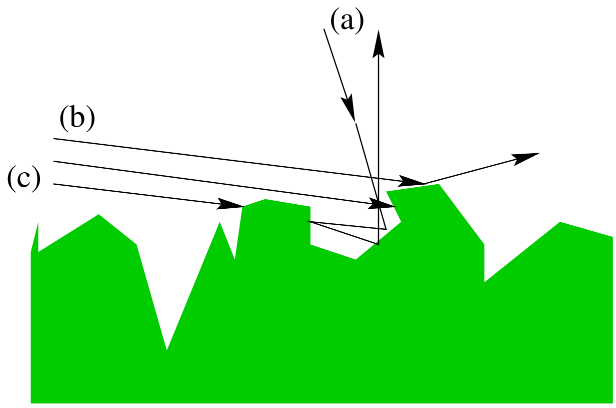
Veliko grafičnih algoritmov uporablja linearno kombinacijo osnovnih odbojnih enačb. Tak model se navadno uporablja za aproksimacijo zglajenih (poliranih) površin. Osnovni modeli so lahko empirični [Pho75] ali pa temeljijo na fizikalnih osnovah, kot je to primer pri Fresnelovih enačbah (2.6), ki imajo spremenljivo odbojnost pri različnih vpadnih kotih svetlobe. Če upoštevamo še osvetlitev okolice (3.2), lahko zapišemo Phongov model odboja na površini kot

$$I = k_a I_a + I_i \left[k_d(\vec{L} \cdot \vec{N}) + k_s(\vec{N} \cdot \vec{H})^n \right] \tag{3.18}$$

Intenziteta svetlobe v smeri gledanja $\vec{V}$ je funkcija normale površine $\vec{N}$. Če je projekcija slike paralelna, potem je vektor $\vec{V}$ konstanten. Svetlobni izvor lahko postavimo v neskončnost in s tem zagotovimo konstantnost vektorja luči $\vec{L}$, kar dodatno poenostavi računanje, saj je pri tem konstanten tudi $\vec{H}$. Pri takih poenostavitvah je potrebno računati le $\vec{N} \cdot \vec{H}$. Podobno kot v enačbi (3.6) lahko zapišemo model za več svetlobnih izvorov:

$$I = k_a I_a + \sum_n I_{in} \left[k_d(\vec{L}_n \cdot \vec{N}) + k_s(\vec{N} \cdot \vec{H}_n)^n \right] \tag{3.19}$$

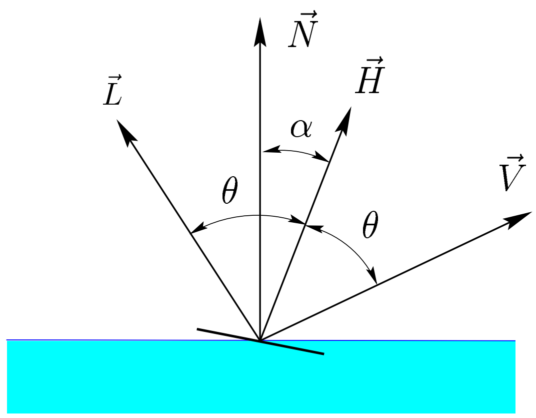
Pri več svetlobnih izvorih se lahko zgodi, da maksimum intenzitete svetlobe prekorači maksimalni nivo svetlosti za posamezo točko. S pravilnim izborom konstant materiala



Slika 3.7: Odboj (a, b) in absorbcija (c) svetlobe na hrapavi površini

Cook-Torrance model upošteva povprečno orientacijo mikropovršin in njihovo odbojnost, kot povprečje v odvisnosti od vpadnega kota, in s tem tudi vpliv na barvo. Tako modeliranje ne kaže potrebe po ločenem difuznem in zrcalnem odboju, saj je možno z zrcalnim odbojem in porazdelitvijo v odvisnosti od vpadnega kota opisati vse značilnosti.

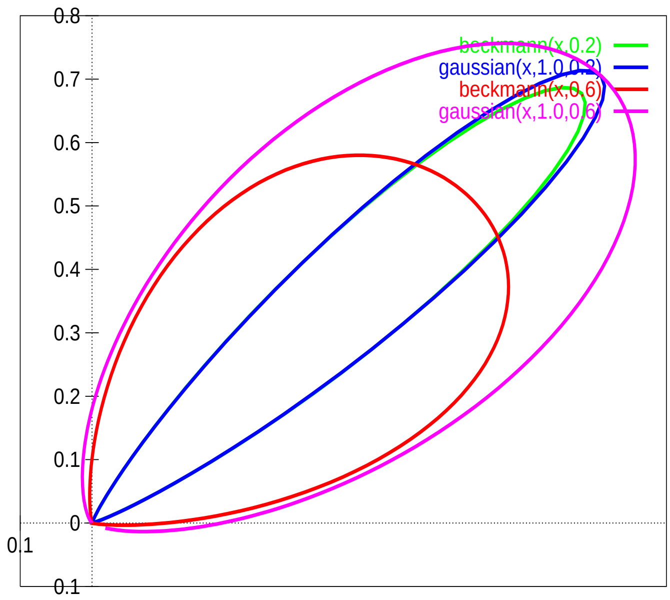
Model Cook-Torrance temelji na fizikalnem modelu svetlobe. Zaradi tega je možno modelirati tudi svetlobne izvore, ki niso točkasti. Za razliko od Phongovega modela, ki je empirično podajal zrcalni odboj z $\cos^n \phi$ je Cook-Torrance model v osnovi fizikalni. Upošteva barvo svetlobe z vpeljavo valovne dolžine λ in gostoto porazdelitve mikropovršin v smeri α. Na sliki 3.8 so prikazane geometrijske razmere na eni mikropovršini, ki je v modelu uporabljena kot popolnoma gladka zrcalna površina, orientirana glede na povprečno orientacijo vseh mikropovršin $\vec{N}$.



Slika 3.8: Odboj svetlobe na mikropovršini nagnjeni za kot α od normale $\vec{N}$

Svetloba prihaja iz smeri vektorja $\vec{L}$ in se odbija na vse strani. Intenziteta svetlobe

Prednost uporabe je poleg točnosti tudi v tem, da enačba (3.23) ne vsebuje konstante c, ki bi jo moral uporabnik določiti. Na sliki 3.9 sta primerjalno prikazani obe porazdelitveni funkciji (3.22 in 3.23) iz katerih je razvidno, da je Beckmannova porazdelitev širša in nekoliko nižja od Gaussove, če izberemo konstanto $c = 1.0$.



Slika 3.9: Gaussova in Beckmannova porazdelitvena funkcija $D(m, c, \alpha)$ za $m = 0.2$ in $m = 0.6$ predstavljena s polarnim diagramom in vektorjem odboja $\vec{R}$ pod kotom $45°$

Izračun porazdelitvene enačbe lahko pospešimo tako, da uporabimo interpolacijsko tabelo v odvisnost od kota α in uporabljenih površin podanih s hrapavostjo m. Poleg osnovne porazdelitve, lahko uporabimo tudi linearno kombinacijo različnih tipov površin in s tem dobimo večje možnosti za popolnejši opis, kar zapišemo kot utežnostno vsoto porazdelitvenih funkcij:

$$D(\alpha) = \sum_j w_j D(m_j, \alpha), \qquad \sum_j w_j = 1 \tag{3.24}$$

3.2.2 Geometrijski faktor slabljenja

Izraz $G(\vec{N}, \vec{V}, \vec{L})$ določa slabljenje intenzitete svetlobe zaradi medsebojnega zasenčevanja mikropovršin. To pomeni, da se intenziteta pod velikimi vpadnimi kot θ manjša zaradi tega, ker mikropovršine ustvarjajo senco druga drugi in tako odboj ni popoln v smeri gledanja. Zaradi lažjega obravnavanja je pri modeliranju uporabljena površina s simetričnimi utori v obliki črke V.

Vektor $\vec{H}$ se lahko uporabi kot merilo pri katerem se zgodi popolni zrcalni odboj. Enačba (3.16) se lahko uporabi tudi za mikropovršine. Za mikropovršino je pomembno razmerje med prepuščeno in blokirano svetlobo pri odboju. Obstajata dva mejna primera:

Sprememba barve zrcalnega odboja.

Določanje barve zrcalnega odboja je lahko zelo težavno, če moramo uporabiti tabelarično podane vrednosti za odbojnost in jih uporabiti v Fresnelovih enačbah. Barva odbite svetlobe je odvisna od vpadnega kota na mikropovršino in je za razliko od Phongovega modela, kjer je možna uporaba le bele svetlobe, pri Cook-Torrance modelu odvisna tudi od barve svetlobnega izvora. Če svetloba pade na površino pod pravim kotom, je barva odbite svetlobe enaka barvi materiala. Pri premikanju vpadnega žarka proti površini je barva odbite svetlobe vedno bolj podobna barvi vpadnega žarka. Izračun spremembo barve iz Fresnelovih enačb je lahko zamuden, zato je bolje, če uporabimo ustrezno interpolacijsko shemo za izračun barve pri vmesnih kotih.

Pri pravokotni osvetlitvi vidimo barvo izvora pomnoženo s spektrom odboja svetlobe objekta. Pri nizkih vpadnih kotih pa je videti odbito barvo svetlobnega izbora skoraj nespremenjeno. Za izračun pri vmesnih kotih lahko uporabimo linearno interpolacijo. Najprej pa je potrebno določiti barvo za oba ekstremna primera. Pravokotna osvetlitev $\theta = 0$ ima vrednost odbojnostne krivulje $F_0(\lambda) = F(\theta = 0, \lambda)$. Intenziteta odbite svetlobe pri valovni dolžini λ je enaka produktu intenzitete vhodnega žarka pri tej valovni dolžini $I_i(\lambda)$ in vrednosti Fresnelovega odboja $F_0(\lambda)$, kar lahko zapišemo kot

$$C_0(\lambda) = F_0(\lambda) I_i(\lambda) \tag{3.31}$$

Pri kotu $\theta = \pi/2$ je vpadna svetloba vzporedna z mikropovršino. Fresnelov izraz pri tem kotu zapišemo kot $F_{\pi/2}(\lambda) = (\theta = \pi/2, \lambda)$. Odbita svetloba ima enako intenziteto kot vpadna svetloba, torej

$$C_{\pi/2}(\lambda) = I_{\pi/2}(\lambda) \tag{3.32}$$

Odbojnost pri poljubnem kotu θ označimo kot $F(\theta, \lambda)$. Končno lahko zapišemo intenziteto pri valovni dolžini λ tako, da uporabimo mejne vrednosti barv za določitev razpona in vrednosti Fresnelove funkcije pri izbranem kotu za vmesne vrednosti. Pri standardni linerani interpolaciji zapišemo enačbo kot

$$y(x) = kx + n, \qquad y(x_{min}) = y_{min},\ y(x_{max}) = y_{max} \tag{3.33}$$

za katero lahko zapišemo ekvivalente pri barvni interpolaciji:

$y = C(\theta, \lambda)$	Intenziteta barve pri željenem kotu θ
$k = C_{\pi/2}(\lambda) - C_0(\lambda)$	Nagib premice zagotavlja, da bo pri intenziteta pri nizkem vpadnem kotu $C_\theta(\pi/2) = I_{\pi/2}(\lambda)$ in pri pravokotni osvetlitvi $C_\theta(0) = C_0(\lambda)$
$x = \frac{\max(0, F(\theta,\lambda) - F_0(\lambda))}{F_{\pi/2}(\lambda) - F_0(\lambda)}$	Transformacija odbojnosti $F(\lambda)$ v meje $x = [0, 1]$
$n = C_0(\lambda)$	Začetna (najnižja) intenziteta

kjer je:

C	barva površine
C_l	barva svetlobnega izvora
$\vec{N}$	enotski vektor orientacije površine
$\vec{R}$	enotski vektor v smeri zrcalno odbite svetlobe
$\vec{V}$	enotski vektor v smeri gledanja
$\vec{T}$	enotski vektor v smeri lomljene svetlobe
k_{dr}	difuzna odbojnost
k_s	zrcalna odbojnost
k_{st}	zrcalna prepustnost
k_{sr}	Phongova odbojnost
n	Phongov eksponent

V tem modelu je predpostavljeno, da je izvor svetlobe točkast in leži v neskončnosti, kar omogoča uporabo konstante C_l na vseh vpadnih točkah. Osvetlitev okolice je potrebno upoštevati v vseh konstantah izrazov. Vsak izraz podaja različen efekt:

$k_{dr}C_l(\vec{N} \cdot \vec{L})$	Difuzno odbita svetloba. k_{dr} je različen od nič za vse matirane površine.
$k_sC(\vec{R})$	Zrcalno odbita svetloba. $C(\vec{R})$ je barva vidna v smeri $\vec{R}$, zmanjšana za zrcalno odbojnost k_s. Za gladke površine bi se morala uporabiti le konstanta k_s, ki je različna od nič za vse zglajene površine. Če se ne uporabi Fresnelovih enačb, bi moral biti k_s večji pri nizkih vpadnih kotih ($\theta > 60°$).
$k_{st}C(\vec{T})$	Prepuščena svetloba. $C(\vec{T})$ je barva vidna v smeri vektorja $\vec{T}$, zadušena z zrcalno prepustnostjo k_{st}. Vsota k_s in k_{st} naj bi bila ena.
$k_{sr}C_l(\vec{R} \cdot \vec{V})^n$	Phongov izraz za odblesk. Razprši zrcalni odboj v točki. k_{sr} je izrazit na kompozitnih površinah.

Hallov model [HG83] dodatno deli vpadno svetlobo na dva razreda: svetloba, ki prihaja direktno iz svetlobnih izvorov in na svetlobo, ki prihaja od ostalih objektov. Potreba po taki delitvi je v tem, da je pri metodi sledenja žarka potrebno upoštevati tudi svetlobo, ki jo generirajo drugi objekti. To je svetloba, ki je prišla z odbojem od površine ali pa skozi površino. V primerjavi z modelom (3.38) upošteva tudi razpršitev svetlobe pri zrcalnem prehodu, kot tudi dušenje na enoto dolžine pri prehodu svetlobe skozi prosojni material. Torej modelira vse značilnosti interakcije svetlobe s površino in se zelo približa zahtevam absorbcije svetlobe v mediju, skozi kateri žarek potuje. Barva svetlobe v smeri zrcalnega odboja $\vec{R}$ se ne izračunava s porazdelitveno funkcijo, kot je to primer pri določanju svetlobe pri svetlobnih izvorih. Razlog je v tem, da bi bilo potrebno za svetlobo, ki prihaja iz drugih teles, pri metodi sledenja žarku, generirati večje število žarkov in ne samo enega v smeri zrcalnega odboja. Barva takega odboja se tako določi, kot če bi to bil popolni zrcalni odboj. Za pravilno določitev barve lahko tu uporabimo Fresnelovo enačbo (2.6). Prehodnost T pri medijih, kot je zrak lahko postavimo na ena in s tem ne vplivamo na slabljenje žarka.

$k_{st}C(\vec{T})T_t^{\Delta st}$ Zrcalna prehodnost svetlobe, ki prihaja iz drugih teles skozi površino. Podobno kot pri zrcalnem odboju, tudi tu kontroliramo prepuščeno sveltobo v smeri lomljene svetlobe $\vec{T}$. Absorbcija svetlobe $T_t^{\Delta st}$ je tu običajno manjša od ena, saj so snovi skozi katere žarek potuje največkrat voda ali drugi materiali, ki veliko bolj slabijo svetlobo, kot pa zrak.

k_aC_a Osvetlitev, ki prihaja iz okolice. S tem členom upoštevamo svetlobo, ki prihaja iz drugih teles, a jo ni možno določiti z zrcalnim odbojem svetlobe, ki prihaja od drugih teles. To je torej difuzni člen, ki manjka pri obravnavanju svetlobe, ki prihaja iz drugih teles, vendar iz neznane smeri.

Členi z vsotami torej predstavljajo svetlobo več svetlobnih izvorov. Ostali členi zajemajo svetlobo, ki je prišla na površino iz drugih površin ali iz okolice (zadnji člen). Konstante so zaradi upoštevanja prehoda med seboj odvisne:

$$k_{dr} + k_{sr} = 1 \tag{3.42}$$

$$k_{dt} + k_{st} = 1 \tag{3.43}$$

Vse konstante morajo biti tudi v ustreznih mejah $[0..1]$. Barva C se v modelih običajno izrazi kot

$$C = I(\lambda)F(\lambda) \tag{3.44}$$

kjer je $F(\lambda)$ funkcija odboja ali prehoda v odvisnosti od valovne dolžine λ; $I(\lambda)$ pa je intenziteta vpadne svetlobe. Za zrcalni odboj $F(\lambda)$ ustreza Fresnelovi funkciji odboja. Za funkcijo prehoda lahko zapišemo podobno odvisnost kot pri konstantah

$$F_{dt}(\lambda, \theta) + F_{dr}(\lambda, \theta) = 1 \tag{3.45}$$

Hallov model lahko tudi izboljšamo, če direktno uporabimo model Cook-Torrance pri odboju svetlobe izvorov, namesto kontrole območja odbleskov s Phongovim modelom. Dodatne izboljšave so možne z vpeljavo intenzitete okolice $I_a(\lambda)$ v izraze za difuzni prenos svetlobe.

Poglavje 4

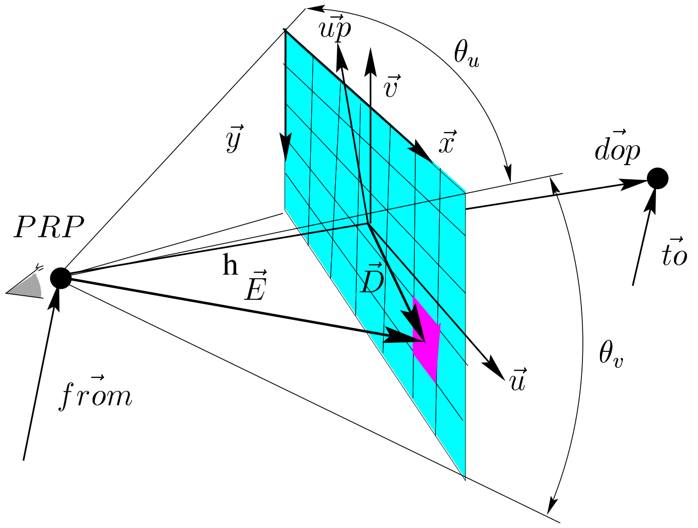
Sledenje žarku

Sledenje žarku (*Ray Tracing*) je metoda, s katero so v računalniški grafiki dosegli eno od najbolj realističnih upodobitev objektov[Gla91b]. Zaradi svoje rekurzivne narave lahko simulira večkratne odboje svetlobe med objekti, kot tudi transparentnost površin in sence, ter s tem rešuje problem globalne osvetlitve (*global illumination*). Reševanje globalne osvetlitve pomeni, da se pri določanju barve površine uporabi poleg direktne osvetlitve tudi svetloba, ki prihaja od drugih objektov. Največkrat metoda temelji na geometrijskem modelu, kjer je osnova žarek, ki potuje po prostoru in se odbija od površin. Teoretično obstajati dve vrsti metode sledenja žarku:

Sledenje naprej. Tu sledimo žarku od izvora svetlobe, do površine, na kateri se odbije na vse strani. Ko sledimo žarek od izvora svetlobe do končnih površin, aproksimiramo realne razmere pri potovanju fotonov. Od vsakega svetlobnega izvora je potrebno poslati množico žarkov v vse smeri in slediti njihovem potovanju skozi prostor. Pri potovanju se žarki odbijajo, absorbirajo in prehajajo skozi površine. Če imamo srečo se žarek odbije tudi v smeri gledanja in s tem zaznamo barvo in intenziteto žarka v določeni točki. Velika večina žarkov pa tudi ob večkratnih odbojih ne doseže očesa. Taki žarki tako ne prispevajo k rezultatu končne projekcije in so zato računani brez potrebe.

Sledenje nazaj. Žarku sledimo od očesa v željeni smeri in iščemo morebitne preseke žarka in objektov. To je sledenje v naprotnem smislu gibanja fotonov v izbranem žarku. Ob uspešnem preseku določimo osvetlitev na površini in morebitne prispevke svetlobe drugih površin. Ker je sledenje žarku v smeri od očesa proti objektom veliko manj potratna, je zato tudi v edina praktično uporabna metoda sledenja žarku v računalniški grafiki. Tako je z imenom *sledenje žarku* (*Ray Tracing*) izključno mišljeno sledenje žarku nazaj.

Možne so seveda tudi kombinacije obeh pristopov. Vprašanje je le, kaj lahko pridobimo s kombinacijo obeh metod. V zadnjem času se uporablja kombinacija metode sledenja žarku nazaj skupaj z metodo žarčenja svetlobe (*Radiosity*) [Shi90]. S to kombinirano število rastrskih enot v smeri $\vec{u}$ in $\vec{v}$. Kot parametre, ki določajo vidni prostor na sliki 4.1, je možno izbrati različne spremenljivke kot so: velikost filma, smer, točka gledanja, orientacija in oddaljenost očesa od projekcijske ravnine. Na sliki 4.2 je prikazan sistem gledanja, primeren za metodo sledenja žarku saj upošteva tudi smer gradnje rastrske slike, ki se pri velikem številu zaslonov in rastrskih datotek začenja v zgornjem levem kotu.



Slika 4.2: Perspektivna projekcija za metodo sledenja žarku

Cilj vsakega rastrskega sistema gledanja je določitev smeri povratnih žarkov $\vec{E}$, ki imajo izhodišče v točki $\vec{from}$. Osnovni parametri, ki definirajo sistem gledanja na sliki 4.2 so:

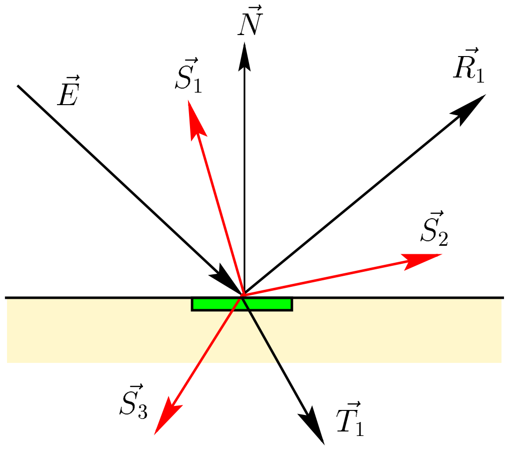
$\vec{from}$	položaj očesa
$\vec{to}$	točka v katero gledamo (središče dogajanja)
$\vec{up}$	orientacija projekcijske ravnine v vertikalni smeri
h	oddaljenost projekcijske ravnine od očesa
θ_u	kot gledanja po horizontali
θ_v	kot gledanja po vertikali
res_x	število rastrskih enot v horizontalni smeri
res_y	število rastrskih enot v vertikalni smeri

Vektor orientacije $\vec{up}$ določa smer vertikalne osi $\vec{v}$, vendar ni nujno ortogonalen na vektor smeri gledanja $\vec{dup}$, ki ga izračunamo kot

$$\vec{dop} = \frac{\vec{to} - \vec{from}}{|\vec{to} - \vec{from}|} \tag{4.1}$$

4.2 Osnovni rekurzivni algoritem

Modeli osvetlitve, predstavljeni v poglavju 3, se lahko uporabijo pri določanju barve površine. Ker se normalno sledi žarku v smeri od očesa proti objektu, je potrebno poiskati najbližji presek med žarkom in vsemi objekti. Po določitvi presečne točke na najbližji površini moramo določiti barvo v tej točki. V ta namen lahko uporabimo enega od osvetlitvenih modelov. Za določitev osvetlitve v izbrani točki vsi osvetlitveni modeli zahtevajo znano normalo na površini, kot tudi ostale optične lastnosti površine.



Slika 4.3: Različni prispevki svetlobe v točki

Na sliki 4.3 so prikazani možni prispevki svetlobe v točki za katero želimo določiti barvo. Povratni žarek $\vec{E}$ se uporabi kot vektor smeri gledanja $\vec{V}$. Povratne žarke $\vec{E}$ imenujemo tudi *primarne žarke*, medtem ko so ostali žarki, ki se generirajo naknadno ob izračunu presečišča $\vec{E}$ s površino, *sekundarni žarki*. V presečni točki so možni različni načini prenosa svetlobe:

Lokalna osvetlitev. Svetloba se proizvaja iz samosvetleče površine. Sem spada tudi osvetlitev za katero nimamo točno določenega vira. Ambientna svetloba je tudi lokalna osvetlitev v presečni točki.

Direktna osvetlitev. Prihajajoča svetloba svetlobnih virov $\vec{S}_i$, ki so največkrat točkasti.

Indirektna osvetlitev. Osvetlitev drugih površin vpliva tudi na osvetlitev v točki. Največji prispevek odbite svetlobe, ki prihaja od drugih površin, je prav gotovo v smeri zrcalnega odboja $\vec{R}$. Drugi pomembni prispevek k osvetlitvi prosojne površine je tudi svetloba v smeri lomljenega žarka $\vec{T}$.

Natančnost določitve posameznih prispevkov je odvisna od načina modeliranja, ki pa se lahko zelo zakomplicira, če želimo pravilno določiti prispevek svetobnih virov, ki niso točkasti. Pri indirektni osvetlitvi je za podrobnejše obravnavanje potrebno upoštevati tudi prispevke, ki niso samo v smeri maksimalnega prispevka $\vec{R}$ in $\vec{T}$.

4.2.2 Odbiti in prepuščeni žarki

Na sliki 4.3 sta prikazana tudi prispevka svetlobe z indirektno osvetlitvijo. Posamezen prispevek je odvisen od lastnosti površine. Gladke površine imajo visoko stopnjo odbojnosti, zato je možno, da bo svetloba, ki prihaja od drugih površin opazno prispevala k skupni osvetlitvi v točki. Podobno lahko trdimo tudi za prozorne površine. Možne so tudi kombinacije obeh lastnosti. V preseku generiramo sekundarne žarke za odbite in prepuščene žarke, vendar le, če ima površina take lastnosti, da je generiranje smiselno.

V smeri zrcalnega odboja $\vec{R}$ je moč pričakovati indirektno osvetlitev z največjo intenziteto. Ob znani normali v presečišču lahko smer odbitega žarka izrazimo kot

$$\vec{R} = \vec{E} - 2(\vec{N} \cdot \vec{E})\vec{N} \tag{4.9}$$

Pri določaju lomljenega žarka lahko uporabimo delni rezultat iz $c_1 = \cos\theta_1 = -\vec{E} \cdot \vec{N}$. Obstaja več metod za določitev $\vec{T}$, ki se razlikujejo po številu operacij[Hec91]. Katera metoda je hitrejša, je odvisno od hitrosti osnovnih operacij ciljnega računalnika, kot so množenje, deljenje, seštevanje, ... Smer lomljenega žarka po Heckbertovi metodi, ki je najenostavnejša, določimo kot:

$$\vec{T} = \eta\vec{E} + (\eta c_1 - c_2)\vec{N} \tag{4.10}$$

kjer je

$$c_2 = \cos\theta_2 = \sqrt{1 - \eta^2(1 - c_1^2)} \tag{4.11}$$

in lomni količnik $\eta = \eta_1/\eta_2$. Vektorji morajo biti normirani. Popolni notranji odboj na površini se zgodi, ko je rezultat enačbe (4.11) imaginarno število – negativen izraz pod korenom.

4.2.3 Whittedov algoritem

Izvedba splošnega sledenja žarku je zahtevna zaradi velikega števila izračunov presekov, kot tudi množice možnih sekundarnih žarkov s katerimi katerih lahko prihaja osvetlitev. Na sliki 4.5 je prikazan primarni žarek $\vec{E}$, ki na prvi presečni površini generira odbiti žarek $\vec{R}_1$ in prepuščeni žarek $\vec{T}_1$. V točki preseka se določi tudi direktna osvetlitev v smeri $\vec{S}_1$.

Whittedov pristop določanja indirektne osvetlitve upošteva samo žarke v smeri $\vec{R}$ in $\vec{T}$. Če je površina odbojna, se generira sekundarni žarek v smeri $\vec{R}$ tolikokrat, kolikorkrat je podrebno, da je prispevek k osvetlitvi primarnega žarka še bistven ali če je doseženo predpisano število zrcalnih odbojev. Podobno velja za prozorne površine. Na sliki 4.5 so vse krogle delno prozorne in odbojne. V splošnem vsak sekundarni žarek generira tri žarke, od katerih sta dva rekurzivna.

Poenostavljen osvetlitveni model v vsaki presečni točki, na osnovi Hallovega modela, lahko zapišemo kot:

$$I = I_a k_a + I_i(k_d(\vec{N} \cdot \vec{L}) + k_s(\vec{N} \cdot \vec{H})^n + k_t(\vec{N} \cdot \vec{H}')^n) + k_{tg} I_t + k_{rg} I_r \tag{4.12}$$

Prvi sledi žarku in išče presečišča z objekti, drugi pa skrbi za izračun osvetlitve v najbližjem presečišču.

proc **Trace**(int globina, Vec3D $\vec{zacetek}$, Vec3D $\vec{smer}$, Barva *barva)
begin
  if globina $>$ maxglobina **then**
    barva := črna
  else
    begin
      { poišči najbližje presečišče za podani žarek $\vec{zacetek}$ in $\vec{smer}$}
      if presečišče obstaja **then**
        begin
          { določi normalo v presečišču }
          { izračunaj lokalno osvetlitev }
          { določi senčne žarke $\vec{S}$ in direktno osvetlitev }
          { izračunaj smer zrcalnega odboja $\vec{R}$ }
          Trace(globina+1, presečišče, $\vec{R}$, zrcalna-barva)
          { izračunaj smer prepuščenega žarka $\vec{T}$ }
          Trace(globina+1, presečišče, $\vec{T}$, prepuščena-barva)
          { združi barvo lokalne osvetlitve, direktne osvetlitve, zrcalno barvo
            in prepuščeno barvo z upoštevanjem lastnosti površine }
        end
    end
end{ Trace }

V prikazanem algoritmu se lahko uvede dodatna optimizacija, ki preverja smiselnost generiranja zrcalno odbitih žarkov glede na prispevek, ki je odvisen od odbojnosti površine. Podobna optimizacija je možna tudi pri kontroli prozornosti površine. Pri obeh testih se primerja prispevek osvetlitve s sekundarnim žarkom z neko predpisano minimalno vrednostjo, ki je še smiselna za rekurzivno generiranje novih žarkov.

4.3 Geometrija presekov žarka in površine

Običajno se za določanje presekov žarka z objekti porabi več kot 90% celotnega časa pri metodi sledenja žarku. Potreben čas za izračun celotne scene ni odvisen samo od števila objektov, ampak tudi od geometrijske zahtevnosti izračuna preseka in normal uporabljenih osnovnih primitivov. Sheme, ki vnaprej izločijo objekte, ki nimajo možnosti za presek z žarkom, lahko bistveno prispevajo k zmanjšanju celotnega računskega časa.

Določanje presekov zahtevnih površin, ki so podane parametrično, ni vedno analitično rešljivo. Zato je pri takih površinah potrebno uporabiti numerične metode, ki dodatno upočasnijo računanje presekov. Z uporabo abstraktnih geometrijskih objektov v programu je

Površino P, ki vsebuje trikotnik T, zapišemo kot

$$\vec{N}(\vec{p} - \vec{p_i}) = 0 \tag{4.18}$$

kjer je $\vec{p}$ poljubno izbrana točka na površini, $\vec{p_i}$ izhodiščna točka, ki definira lego površine v koordinatnem sistemu, $\vec{N}$ pa normala, ki določa orientacijo ravnine. Izhodiščna točka je lahko katerokoli ogljišče trikotnika.

Normalo ravnine definirajo ogljišča trikotnika. Ravnina ima dve strani in s tem dve možni usmerjenosti normal. Za določanje preseka izbor vrstnega reda ogljišč v vektorskem produktu ni pomemben:

$$\vec{N} = (\vec{p}_3 - \vec{p}_1) \times (\vec{p}_2 - \vec{p}_1) \tag{4.19}$$

Žarek $\vec{r}$ je na površini takrat, ko oddaljenost t_0 zadovoljuje enačbo

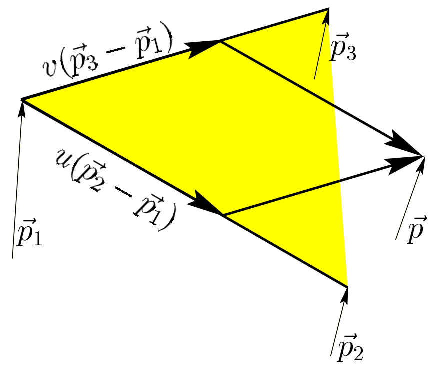
$$\vec{N}(\vec{o} + t_0\vec{v} - \vec{p_i}) = 0 \tag{4.20}$$

kar ob rešitvi da

$$t_0 = -\frac{\vec{N} \cdot \vec{o} - \vec{N} \cdot \vec{p_i}}{\vec{N} \cdot \vec{v}} \tag{4.21}$$

Ob določitvi presešiča žarka z ravnino je potrebno ugotoviti še ali se presečišče nahaja znotraj trikotnika. Ogljišča trikotnika lahko uporabimo za nastavitev enostavnega koordinatnega sistema (u, v) na ravnino P. Vsaka točka na ravnini je podana v tem koordinatnem sistemu kot:

$$\vec{p} = \vec{p}_1 + u(\vec{p}_2 - \vec{p}_1) + v(\vec{p}_3 - \vec{p}_1) \tag{4.22}$$



Slika 4.6: Koordinatni sistem trikotnika v ravnini

Točka $\vec{p} = \vec{0} + t_0\vec{v}$ se nahaja v trikotniku, če je:

1. $0 \leq u \leq 1$
2. $0 \leq v \leq 1$

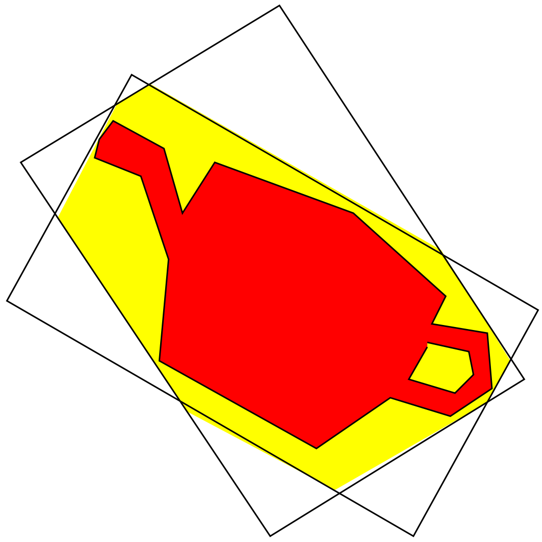
- blobi
- proste površine
- bilinearni in bikubični zlepki
- sovisne površine
- štirikotniki
- vrtenine
- fraktali
- voxli

Podrobnejši pregled presekov različnih primitivov najdemo v [Han91]. Večino geometrijskih primitivov je možno tudi deformirati z različnimi funkcijami, kot so raztezanje, tlačenje, zavijanje, striženje in podobne. Čeprav je možno deformirati objekte že v bazi, je enostavneje deformirati žarek, ki potuje skozi objekt. S takim pristopom je možno enostavneje izračunati presek, kot pa če bi imeli že deformiran objekt.

4.4 Izboljšave osnovnega algoritma

Osnovni algoritem metode sledenja žarku je mogoče izboljšati na različne načine, ki pa so odvisne od tega, kaj želimo doseči. Zaradi časovne zahtevnosti metode sta očitna dva izključujoča trenda: povečanje hitrosti obstoječih algoritmov ali zamenjava z boljšimi ter nasprotne težnje k čim bolj realističnim prikazom. Vprašanja, ki si jih normalno postavimo pri ocenjevanju potreb v upodabljanju so:

- Kakšne scene dovolj dobro opisujejo običajne prikaze in kaj so kompleksne scene? Kolikšna je normalna velikost opisa, ki ustreza testom?
- Kako ocenimo napake, ki se pojavljajo z aproksimacijo in empiričnimi modeli osvetlitve? Potrebna merila za oceno napake.
- Zaznavanje in subjektivna ocena osvetlitve je eden od pomembnih cenilcev realističnosti. Kako to upoštevamo pri generiranju dvodimenzionalne slike?
- Ali je potrebno upoštevati tudi anizotropne modele osvetlitve? Ali so potrebni boljši fizikalni modeli odboja?
- Kako natančni morajo biti osvetlitveni modeli za aplikacije v računalniški grafiki?
- Ali je potrebno modelirati tudi izvore svetlobe s porazdeljeno intenzivnostjo?



Slika 4.7: Presek dveh omejitvenih prostorov

izračunati le če obstaja presek z vsemi omejitvenimi prostori kompleksnega objekta. Koliko omejitvenih volumnov se še splača postaviti okoli enega samega objekta, je predvsem odvisno od razmerja zahtevnosti izračuna preseka osnovnega objekta in omejitvenega prostora. Ker se presek žarka običajno izračunava za vse objekte v sceni, se lahko zgodi, da obstaja več presekov kompleksih objektov z enim samim žarkom. Če se preseki omejitvenih prostorov med seboj ne stikajo, lahko enostavno opustimo točno računanje presekov tistih kompleksih objektov, ki so za prvim objektom.

Podobno strategijo lahko uvedemo tudi pri mešanih okoljih, kjer imamo enostavne objekte brez omejitvenih prostorov in kompleksne objekte z omejitvenimi prostori. Pri določanju preseka se lahko zgodi, da obstaja najbližji presek za enostavni objekt, nato pa mu sledi presek z omejitvenim prostorom. V takih primerih prav tako ni potrebno točneje določiti preseka za kompleksni objekt.

Sheme z omejitvenimi prostori sicer nekako linearizirajo potreben računski čas, tudi če imamo v sceni večje število zahtevnih objektov. Problem se pojavi z večanjem števila osnovnih primitivov. Tako naletimo na problem kompleksnih scen, kjer število objektov bistveno vpliva na računski čas, saj je še vedno potrebno za vsak žarek izračunati (testirati) presek z vsemi objekti v sceni. Če uporabimo nekoliko spremenjen pristop pri gradnji seznama objektov, pri katerem bi se že vnaprej vedelo ali sploh obstaja možnost preseka žarka in objekta, bi bil čas neodvisen od števila objektov. Eden od enostavnejših prijemov je ta, da naredimo prostorski seznam, ki razdeli upodabljani prostor na enako velike kocke (*voxle*). Vsaki kocki pripišemo seznam objektov, ki jih kocka vsaj deloma vsebuje. Pri iskanju presekov žarka s še neznanimi objekti, najprej poiščemo preseke žarka s kockami. Ko dobimo presečne kocke, lahko iz seznamov, ki pripadajo kockam enostavno dobimo seznam možnih objektov, ki jih je potrebno testirati na presek. Uniformna delitev prostora

Poglavje 5

Distribuirani sistemi

Reševanje problemov v mreži heterogenih računalnikov se je včasih obravnavalo kot zanimivo in nepomembno področje v računalništvu. Vpogled v trende pa kaže, da je vse več zahtev po takem računanju. Novi operacijski sistemi, kot tudi uporabniški vmesniki, zahtevajo vsaj osnovne stopnje paralelnosti pri izvajanju nekaterih zamudnih operacij. Tu je vključena tudi komunikacija med procesi, ki si izmenjujejo podatke – sporočila. Izmenjava sporočil med procesi lahko poteka na nivoju enega procesorja ali med več procesorji, ki so povezani med seboj v omrežju računalnikov.

Uporabnik bi moral brez velikega truda imeti možnost uporabe računalniških zmogljivosti v omrežju. Sistemi, ki omogočajo transparenten dostop do računalniških zmogljivosti se imenujejo *distribuirani sistemi*. Izgradnja takega sistema predstavlja svojevrsten izziv. Transparentnost lažje dosežemo z opremo, ki je podobna, vendar pa pri tem ne izkoristimo zmogljivosti raznovrstne računalniške opreme, ki je na voljo.

Vsi distribuirani sistemi temeljijo na podobnem mehanizmu izvajanja oddaljenih procedur. Sistem je običajno ločen na dva dela, ki ga v računalniški terminologiji imenujemo kot pristop Odjemalec – Strežnik (*Client – Server*). Klicanje oddaljenih procedur ne pomeni nujno, da imamo dva računalnika, ki komunicirata preko mreže, ampak to pomeni, da dva programa komunicirata preko mrežnega protokola. Tako imamo lahko model Odjemalec – Strežnik, ki se izvaja na enem samem računalniku. Ta možnost tudi omogoča razvijalcem programske opreme, da razvijajo in testirajo distribuirane programe na enem računalniku. Programi narejeni na ta način pa delujejo brez sprememb na mreži računalnikov.

Izvajanje oddaljenih procedur poteka tako, kot je prikazano na sliki 5. Procedura v lokalni aplikaciji sproži zahtevo za izvajanje tako, da naslovi računalnik, kjer se oddaljena procedura nahaja. Pri tem poleg zahteve o vrsti (največkrat je to dogovorjena številka) procedure, posreduje še dodatne argumente in podatke, ki jih oddaljena procedura potrebuje za uspešno izvršitev naloge. Na strežniku se običajno nahaja upravitelj oddaljenih procedur (*portmapper*), ki ugotavlja prisotnost zahtevanih procedur in skrbi za zagon zahtevane procedure. Ko procedura (proces na strežniku) konča z delom, vrne rezultate in status uspešnosti opravljene naloge programu, ki je servis zahteval.

- Strežniki naj bi hranili čim manj podatkov o odjemalcih. Ugodno bi bilo, da se servis vzpostavi le takrat, ko je potreben, ne pa da odjemalci ob začetku dela zakupijo dele servisov in s tem po nepotrebnem obremenjujejo strežnik.

5.1 Paketi za razvoj distribuiranih sistemov

Trenutno je na voljo več osnov (*platform*) na kateri lahko razvijalci programske opreme gradijo svoje aplikacije. Ločijo se po namenu in stopnji implementacije. Nizkonivojski ONC/RPC in DCE so primerni le za izgradnjo novih distribuiranih servisov, ki imajo dobro definiran način izmenjave podatkov. Za razvoj paralelnih algoritmov je bolje uporabiti sisteme, ki imogočajo enostavno izmenjavo podatkov, brez dodatnih skrbi kot so: kodiranje/dekodiranje podatkov, alokacija vmesnih pomnilnikov in kreiranje procesov.

5.1.1 RPC

Najbolj razširjena platforma je bila razvita pri prvem in nekdaj največjem proizvajalcu delovnih postaj *Sun Microsystems*, ki je leta 1985 predstavil platformo *Open Network Computing* (ONC). ONC[Mic87a] se sestoji iz knjižnice za transparenten prenos podatkov *eXternal Data Representation* (XDR)[Mic87b] in knjižnice za izvajanje oddaljenih procedur *Remote Procedure Calling* (RPC). Sun ni bilo prvo podjetje, ki je razvilo take rutine za transparenten dostop, je pa bilo prvo, ki je dalo izvorno kodo in navodila za uporabo v splošno uporabo s težnjo, da postane mehanizem RPC/XDR ne-licenčen. RPC najdemo na večini operacijskih sistemov in je osnova za veliko večino knjižnic, ki omogočajo pisanje distribuiranih sistemov z enostavnejšim posredovanjem podatkov med različnimi arhitekturami računalnikov. Mehanizem RPC/XDR omogoča predvsem transparenten prenos podatkov v omrežju tipa IP. Na sliki 5.2 je prikazan nivo implementacije sistema RPC v OSI razdelitvi omrežnih nivojev.

Sistem ONC/RPC je privzelo več kot 90 proizvajalcev sistemske programske opreme in ga vključujejo v distribucijo svojih operacijskih sistemov. Čeprav večina aplikacij deluje na sistemu Unix, pa obstajajo knjižnice RPC tudi za druge sisteme kot so VMS, MS-DOS, VM, OSF/1.

5.1.2 DCE

Neodvisno so kasneje v *Digital Equipment Corporation* (DEC) razvili podobno plaformo *Distributed Computing Environment*[SHM94] (DCE), vendar trenutno ni splošno dostopna, ampak ima licenčno vezavo pri skupini proizvajalcev *Open Software Foundation*. Prav zaradi tega razloga in zaradi tega, ker je sistem razmeroma nov, ni veliko uporabljan, čeprav ima nekaj prednosti pred ONC/RPC. Večji sistem, ki temelji na platformi DCE je *Andrew File System* in imogoča boljšo zaščito podatkov na mreži, večjo transparentnost in kontroliran prenos podatkov med računalniki.

PVM omogoča dinamično kreiranje procesov – novi računalniki se lahko prijavljajo v skupino, procesi lahko brišejo in ustavljajo druge procese. Komunikacija med procesi je možna z rutinami v katerih ni potrebno vnaprej alocirati vmesnih pomnilnikov. Kodiranje in dekodiranje podatkov je transparentno ne glede na sistem, ki ga uporabljamo. Poleg osnovnih ukazov za komunikacijo so možna tudi združevanja v skupine (*group operations*) s sinhronizacijo nalog in globalno pošiljanje sporočil (*broadcasting*). Vsak proces lahko komunicira in se sinhronizira z drugimi. Zaradi popolnosti in enostavnosti je knjižnica PVM dobila široko podporo na različnih področjih paralelnega računanja.

5.1.5 MPI

Message Passing Interface[WGS94] je specifikacija, ki je bila zaključena v letu 1994 in poskuša združiti napore pri standardizaciji knjižnice za programiranje na masivno paralelnih računalniških sistemih. Predstavljeno ogrodje, naj bi bila osnova za sedanje in prihodnje paralelne računalniške sisteme. S standardizacijo uporabniki ne bi bili več vezani na arhitekturo računalnika in bi s tem pridobili na prenosljivosti in lažjem programiranju. Eden glavih ciljev je torej definiranje nabora ukazov, ki bi se ga proizvajalci novih sistemov morali držati, ne da bi bili omejeni pri uvedbi novih strojnih rešitev. Skalabilnost je torej eden od pomembnih izhodišč sistema MPI.

MPI ne vsebuje potrebnih rutin za dinamično kreiranje procesov in konfiguracijo računalnikov v mreži. Je bolj komunikacijski vmesnik, na katerem se lahko zgradijo knjižnice z višjim nivojem abstrakcije. MPI je nivojsko podoben RPC, le da ni mišljen samo za izmenjavo sporočil v omrežju, ampak izkorišča različne tipe komunikacij, ki jih dobavitelji vgradijo v sistem.

5.1.6 Ostali sistemi

Poleg omenjenih sistemov za razvoj obstajajo tudi drugi namenski sistemi, ki so razmeroma zelo vezani na arhitekturo in se normalno prodajajo skupaj z računalniškim sistemom. Sem spadajo različne knjižnice, namensko pisane za (super)računalnike in izbrane jezike, kot so Strand, Compositional C++ (CC++), Fortran M, High Performance Fortran, Occam in drugi [Fos94].

Express. Je skupina orodij za razvoj paralelnih programov v izdaji podjetja ParaSoft, ki omogoča pretvorbo sekvenčnih programov v paralelne z optimizacijo v ciklu konverzije. Osnova sistema Express so knjižnice za komunikacijo, vhod/izhod in paralelno grafiko.

Linda. Model razvit na univerzi Yale. Za razliko od drugim sistemov, ki si pošiljajo sporočila imamo tu "tuple-space" – abstrakcijo preko katere procesi komunicirajo. Abstrakcija je podobna sistemu z deljenim spominom (*shared memory*), le da je tu deljeni procesiranje pa ni mišljeno kot procesiranje večjih računalnikov s počasnimi komunikacijami, ampak hitri procesorji z malimi količinami spomina, kot so na primer transputerska omrežja.

- Cevovodje (*Pipelining*) je najenostavnejši način medprocesne komunikacije in s tem tudi paralelnega procesiranja. Poleg praktične uporabe v različnih operacijskih sistemih, se tak postopek uporablja za komunikacijo med osnovnimi elementi računalnika. Zaradi pogoste enosmerne komunikacije med procesi je možno izdelati tudi zaporednja, kjer procesorji med seboj niso sinhronizirani; možno je tudi to, da procesorji spoh nimajo ure, ampak delajo v analogno/digitalnem načinu.

- Kombinirano procesiranje zajema iskanje različnih algoritmov, ki rešujejo podprobleme za načine kot so prepletanje, cevovodje, večprocesno obdelavo. Problemi, ki jih želimo reševati običajno največkrat niso enoznačno rešljivi, zato je lahko tak pristop zelo učinkovit, vendar težko izvedljiv in ozko namenski, če je implementiran v strojni opremi.

- Vektorsko procesiranje uporablja več tesno povezanih procesorjev, ki delujejo tako, da vsi skupaj uporabijo isto operacijo, vsak na svojem delu podatkov. Tak tip procesiranja je znan pod imenom SIMD (*Single Instruction, Multiple Data*). Tipičen problem, ki ga takšno procesiranje enostavno rešuje je računanje diferencialnih enačb z začetnimi pogoji, kar je največkrat simulacija sistemov.

- Distribuirano procesiranje vključuje več ohlapno povezanih procesorjev, ki so povezani s počasnim medijem, kot je npr. omrežje ethernet. Nadzor je izveden po omrežju in mora zato upoštevati tudi lastnosti medija. Tako je potrebna večja kontrola pri komunikaciji, kar pa pri prej navedenih tipih paralelizma ni bil pogoj. Procesorji so avtonomni, vsak s svojim spominom in logiko. Tako procesiranje je tipično za MIMD procesorje (*Multiple Instructions, Multiple Data*). Lokalna omrežja so tipičen primer MIMD procesiranja.

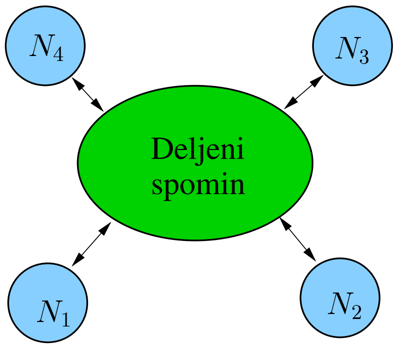
5.2.2 Paralelni programski modeli

Obstaja več različnih programskih modelov s katerimi lahko konstruiramo paralelne algoritme na osnovi von Neumanovega modela računalnika. Von Neumanov model računske enote predpostavlja procesor in spomin, ki sta skupaj sposobna izvajati zaporedje ukazov. Je torej zaključena celota, ki prek vhodno/izhodnih naprav komunicira z drugimi računskimi enotami. Fizično je to lahko en sam procesor s spominom ali pa samo proces v operacijskem sistemu, ki omogoča deljenje procesorskega časa večjemu številu procesov.

Modularnost programov je pomembna osnova pri razvoju paralelnih programov. Zaključene procedure kažejo osnovne dele celotnega programa in s tem lažji vpogled na kasnejšo razdelitev na več računskih enot, ki med seboj komunicirajo po komunikacijskih kanalih. Na sliki 5.3 je prikazan enostavni paralelni programski model s štirimi vozli. Eden

Majhna modifikacija v načinu posredovanja sporočil, kot je prikazana na sliki 5.3, spremeni model nalog s kanali v model s posredovanjem sporočil (*message passing*). Razlika je v tem, da se ne pošilja sporočila v kanal, ampak se pošilja direktno nalogi, ne da bi komunikacijski kanal vnaprej obstajal. Programski model s posredovanjem sporočil je tudi najbolj razširjen paralelni model, ki ga je možno aplicirati na veliki večini računalniških arhitektur. Kreiranje novih procesov ni običajno. Vsi vozli paralelnega programa se običajno kreirajo na začetku in si nato izmenjujejo sporočila. Taki sistem implementirajo programski model SPMD, saj vsak vozel poganja le en program na različnih podatkih.

Ena od omejitev modela s posredovanjem sporočil je ta, da morajo imeti vsi vozli paralelnega programa dovolj velik lokalni spomin. Potrebna velikost je odvisna od implementacije in spominskih zahtev problema. Na sistemih z velikom številom procesorjev je velikost lokalnega spomina lahko tudi omejitev, ki narekuje način izvedbe.



Slika 5.4: Paralelni programski model z deljenim spominom

Na sliki 5.4 je prikazan paralelni programski model z deljenim spominom. Vozli v tem modelu uporabljajo skupni naslovni prostor v katerega pišeijo in berejo asinhrono. Vozli imajo lahko še vedno svoj lokalni spomin, ki pa se običajno uporablja le za manjše podatke lokalnega značaja. Vsa komunikacija poteka preko globalnega spomina. Naloge se med seboj sinhronizirajo z zaklepanjem ali semaforji. Prednost tega model je s programerskega stališča v tem, da ni potrebno deliti podatkov na naloge, kar lahko poenostavi razvoj programov.

5.3 Povzetek

V tem poglavju so predstavljeni osnovni atributi distribuiranih sistemov in paralelnih programskih modelov. Distribuirani sistemi so sistemi, ki omogočajo transparenten dostop do računskih zmogljivosti v omrežju. V distribuiranih sistemih imamo vpeljan princip klicanja oddaljenih procedur. Distribuiran sistem loči programe na strežnike in odjemalce (*server-client*). Znanih je več sistemov, ki omogočajo razvoj paralelnih programov, ki delujejo tako v omrežnem modelu, kot tudi v tesnejših povezavah procesorjev (*Massively Parallel Pracessors*).

Poglavje 6

Paralelni navidezni računalnik – PVM

Sistem PVM (*Parallel Virtual Machine*) je skupina orodij in knjižnic za razvoj paralelnih programov na heterogenem omrežju računalnikov[GBD$^+$94]. Za razliko od masivno paralelnih procesorjev (MPP) imamo v omrežju običajno opravka z različnimi tipi heterogenosti:

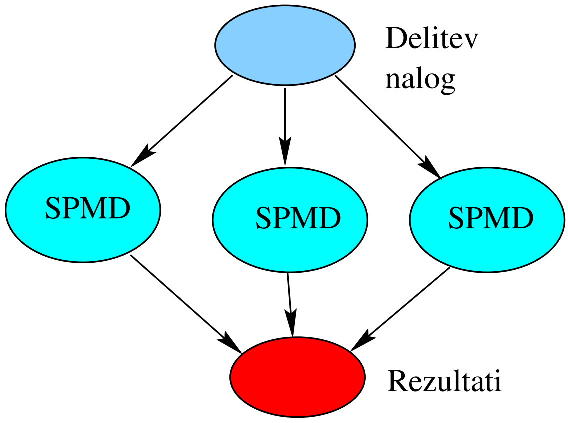
- arhitektura
- oblika podatkov
- računska moč
- obremenitev računalnika
- obremenitev omrežja

PVM omogoča učinovito izrabo tako MPP, kot tudi distribuiranih sistemov z različnimi arhitekturami. Možne so kombinacije z različnimi računskimi zmogljivostmi, kjer osebni računalniki PC386/48 sodelujejo z delovnimi postajami z nekajkrat večjimi zmogljivostmi.

Oblika zapisa podatkov v računalnikih je največkrat med arhitekturami različna. Osnovna razlika je v zlaganju bytov v besede. Najbolj pogosta sta *Little* in *Big Endian* modela sestavljanja v večje besede. Pri posredovanju sporočil je pomembno, da imajo vsi sistemi enoten zapis, ki ga lahko brez težav prevedejo v interno obliko, ki je normalna na računalniku. Sistem PVM omogoča transparentno pretvorbo različnih formatov zapisa podatkov v spominu pri komuniciranju na omrežju.

Zaradi heterogenosti v hitrostih uporabljenih računalnikov je potrebno že pri razvoju paralelnih algoritmov upoštevati, da nekateri računalniki hitreje opravijo naročeno delo. Različne hitrosti ne smejo ustvarjati blokad pri katerih bi hitrejši računalniki čakali, da počasnejši končajo z nalogo, da bi lahko nato ponovno začeli z delom. Tudi če imamo delovne postaje enakega tipa, se lahko zgodi, da so nekatere zaradi trenutne uporabe bolj obremenjene in zaradi tega počasnejše za izvajanje danih nalog.

združijo in tvorijo rešitev prvotnega problema. Primer računskega modela PVM je prikazan na sliki 6.1. Razvidni so trije procesi (SPMD), ki imajo vsak svoje podatke. Razdelitev dela med procese je vnaprej razdelil glavni proces (*master*). Delavci (*workers* ali *slaves*) pa med seboj sodelujejo s ciljem dosege delnih rezultatov, ki se na koncu združijo v končni rezultat.



Slika 6.1: Računski model PVM

6.1.1 Programski modeli PVM

Za razvoj programov PVM lahko uporabimo večino znanih pristopov pri razvoju paralelnih algoritmov. Splošno uporabljan algoritem v sistemu PVM je model množice nalog (*"crowd" computation*). Osnovna ideja je ta, da množica procesov izvaja isto nalogo na različnih delih podatkov celotnega problema. Model množice nalog je v praksi izdelan na dva različna načina:

- Model vodja – delavec (*master – slave*) striktno deli nadzor nad izvajanjem programa v dve skupini. Nadzorni program (*master*) ne izvaja računanja ampak je njegova naloga le dodeljevanje nalog podrejenim procesom, nadzor nad izvajanjem dela in zbiranje rezultatov dela podrejeni procesov. Delavci so zadolženi za konkretno računanje delov celotnega problema, tako kot jim ukaže vodja.

- Drevesno računanje nima razdeljenega dela med delavce in nadzorni program. Tu imamo množico samostojnih vozlišč z enakimi programi, ki enakovredno sodelujejo pri reševanju problema. Tak način reševanja je ustrezen za probleme pri katerih vnaprej ne vemo velikost problema. Primeri taki problemov so iskanje *alpha-beta*, *branch-and-bound* algoritmi in drugi rekurzivni algoritmi za iskanje.

Pri zasnovi programov je potrebno upoštevati pravilno razporeditev nalog na procese. Problem, ki ga rešujemo na več procesih, lahko dodelimo vnaprej za proces, glede na predvideno računsko zmogljivost vozlišča, ki mu dodelimo delo. Tako dodeljevanje nalog je menti programu posredujejo tako, kot je to navada v programskem jeziku C. Predviden je tudi zagon novih procesov brez predpisane lokacije. Odločitev, kje se proces izvede, je prepuščena sistemu PVM.

Z ukazom `pvm_kill()` zahtevamo končanje drugih procesov s tem, da podamo številko procesa ki ga želimo uničiti. Ukaz ni mišljen za končanje lastnega procesa, saj je v te namene bolj primeren `pvm_exit()`.

6.2.2 Informacje

Za uspešno komuniciranje med procesi so potrebne tudi informacije o ostalih procesih. Delavci morajo imeti številko vodje, da mu lahko vračajo rezultate. Za podrobne informacije o arhitekturi in hitrostih ter imenih računalnikov, ki so dostopni v navideznem računalniku, uporabimo `pvm_config()`. Ostale rutine, ki so dostopne, omogočajo zajem informacij o trenutno delujočih procesih v sistemu.

6.2.3 Dinamična konfiguracija

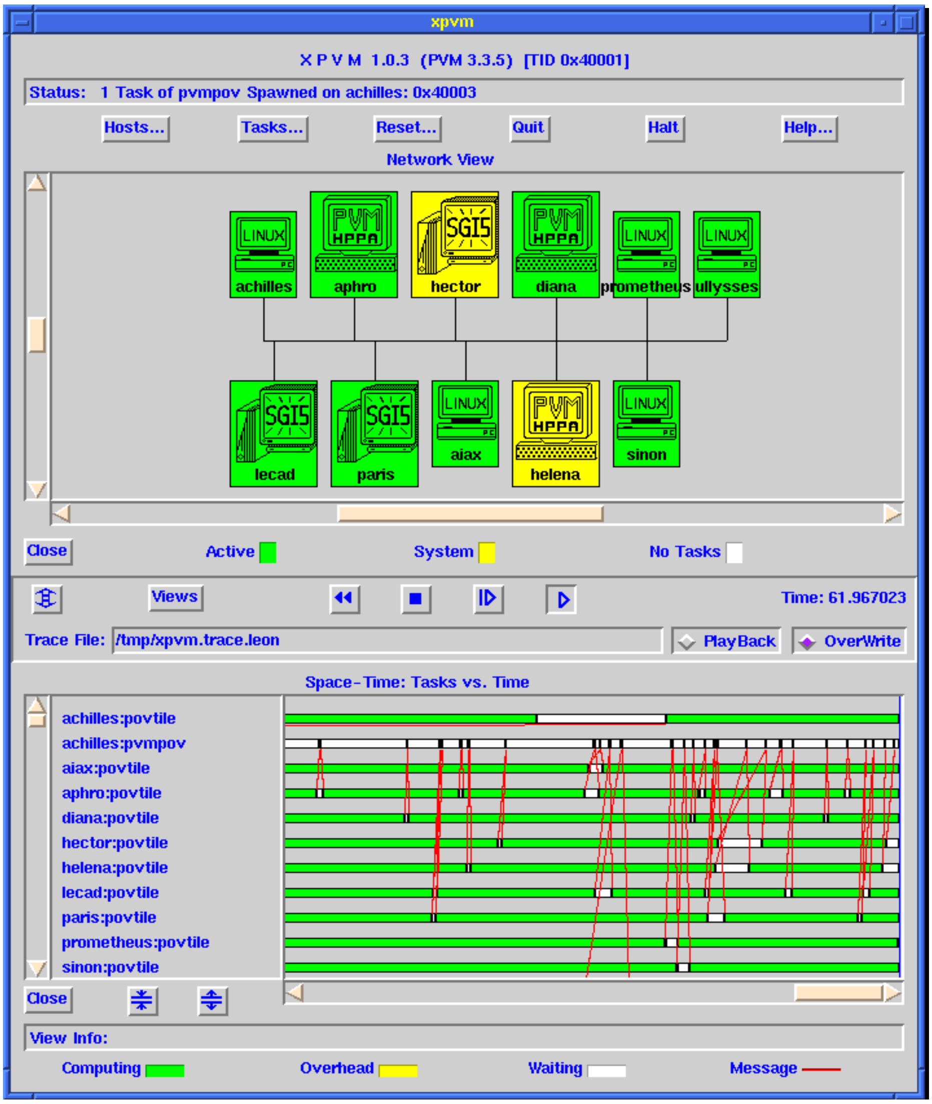
Sistem PVM omogoča dinamično spreminjanje števila delujočih računalnikov v sistemu. Ob znanih imenih je možno hkrati dodati ali brisati več računalnikov. Uporabnost rutin za spreminjanje konfiguracije je v tem, da je možno izdelati tudi programe s tolerantnostjo za napake ali pa programe z redundančnimi računalniki. Ugodnost lahko uporabimo tudi pri problemih, ki imajo različne stopnje zahtevnosti. Ko je na začetku problem majhen, ga je smiselno poganjati le na manjšem številu računalnikov. Ko pa problem naraste preko predvidene meje, lahko dinamično dodajamo nove računalnike, ki pomagajo pri reševanju povečanega problema.

6.2.4 Signalizacija

Klasičen način obveščanja med procesi, prikazan v poglavju D, je izveden tudi v sistemu PVM. Namen signalizacije je predvsem v sinhronizaciji ali v obveščanju ostalih procesov o stanjih, ki niso običajna. Neobičajna stanja so dodajanje ali brisanje novih računalnikov iz sistema. Ostali procesi morajo biti o takih stanjih obveščeni, da lahko popravijo strategijo komuniciranja.

6.2.5 Posredovanje sporočil

Programski model PVM temelji na posredovanju sporočil (*mesage passing*). Sporočila so podatki, ki se posredujejo med procesi. Za pravilen prenos skrbi sistem PVM, ki usmerja in optimira prenos, tako da potujejo sporočila kar najhitreje. V distribuiranem modelu se uporablja protokol UDP/IP. Protokol UDP ima osnovno enoto prenosa paket z omejeno velikostjo. Hitrost prenosa s protokolom UDP je v primerjavi s TCP večja, vendar je dostava paketov nezanesljiva. Protokol TCP zahteva tudi odpiranje povezave pred kakršnimkoli goča tudi grafični prikaz izvajanja programa, merjenje časa in obremenitve računalnikov, kar je pomembno pri dokazovanju pravilnosti delovanja programov in opazovanju učinkovitosti. Na sliki 6.2 je prikazan grafični vmesnik na katerem so razvidne osnovne lastnosti programa. V zgornjem oknu je prikazana konfiguracija računalnikov, ki je bila uporabljena pri implementaciji distribuiranega programa za upodabljanje. Spodnje okno kaže časovni potek pošiljanja sporočil med procesi v navideznem računalniku.



Slika 6.2: Okolje XPVM. Zgornji del predstavlja testno konfiguracijo distribuiranega sistema, ki je bil uporabljen pri implementaciji.

6.3.2 Razvojno okolje PADE

Parallel Applications Development Environment je integrirano razvojno okolje z grafičnim vmesnikom, ki skrbi za najbolj zamudne faze v razvoju paralelnih programov v di-

Poglavje 7

Implementacija

Implementacija sistema za distribuirano upodabljanje obsega dva dela. V prvem delu je bil razvit program z metodo sledenja žarku, v drugem delu pa je bil implementiran distribuirani program za upodabljanje. Prvi program, ki je pokazal delovanje metode sledenja žarku ni bil osnova za kasnejšo razširitev z paralelizacijo. Glavni vzrok temu je dejstvo, da je bila dostopna izvorna koda na Internetu za program s sledenjem žarku – *povray*, ki ima vgrajeno večje število različnih osnovnih primitivov in proceduralnih tekstur. Program je napisan v zelo prenosljivi C kodi, ki se brez težav prevede na različnih operacijskih sistemih.

7.1 Program za sledenje žarku

Razvit je bil rekurzivni program za metodo sledenja žarku. Program omogoča osnovne primitive, ki so bili predstavljeni v poglavju 4. Uporabljeni so bili sodobni prijemi v programiranju z objektno orientiranim pristopom v jeziku C++. Program je zgrajen na knjižnici razredov za operacije v prostoru s hierarhijo kot je predlagana v [Dou94]. Metode omogočajo matematične operacije na vektorjih in matrikah kot so:

- vektor = vektor + vektor
- vektor = vektor - vektor
- vektor = vektor $\cdot$ vektor – skalarni produkt
- vektor = vektor $\times$ vektor – vektorski produkt
- vektor = skalar * vektor
- vektor = dolžina(vektor)
- vektor = normiraj(vektor)
- vektor = matrika * vektor

```
light:  LIGHT point;

background: BACKGROUND primcolor;

surface: SURFACE primcolor num num num num num;

cone:   CONE point num point num;

sphere: SPHERE point num;

polygon: POLYGON num ;

ppatch: PATCH num pointlist;

primcolor: num num num
         | TOKEN;

point:  num num num;

pointlist: pointlist point
         | /* prazno */ ;

num:    NUM;
```

Datoteke NFF so namenjene uporabnikom, ki sami sestavijo ustrezno sceno in jo zapišejo v datoteki. Specifikacija NFF pa je namenjena tudi testiranju različnih algoritmov. Na osnovi NFF je bil razvit programski paket, ki vsebuje 10 različnih testnih scen. Testne scene so rekurzivne oblike. Lahko določamo globino rekurzije in s tem natančnost in število osnovnih gradnikov. Skupina programov NFF omogoča izhod tudi v drugih formatih, ki jih podpirajo različni programi za sledenje žarku.

7.1.2 Zgradba programa

Program za sledenje žarku je napisan v objektno orientiranem jeziku C++[Str91, Ter93]. Z objektnim pristopom k programiranju je program možno brez težav aprilagoditi. Pravilna hierarhija omogoča vgraditev novih primitivov z enostavnim dopisovanjem modulov.

Na sliki 7.1 je prikazana osnovna zgradba programa, predstavljena z razredi, ki so vgrajeni v program. Temeljni gradniki baze podatkov so tridimenzionalni objekti. Razred *Object* združuje podatke in metode skupne vsem tridimenzionalnim objektom. *Object* je abstraktni razred, kar pomeni, da ne more obstajati v bazi gradnikov element, ki bi bil primer (*instance*) razreda Object, ampak lahko v bazo vstavimo le izpeljani objekt.

Izpeljani objekti morajo implementirati vse virtualne metode, če jih želimo vstaviti v bazo gradnikov. Iz slike je razvidno, da so iz razreda *Object* izpeljani trije razredi:

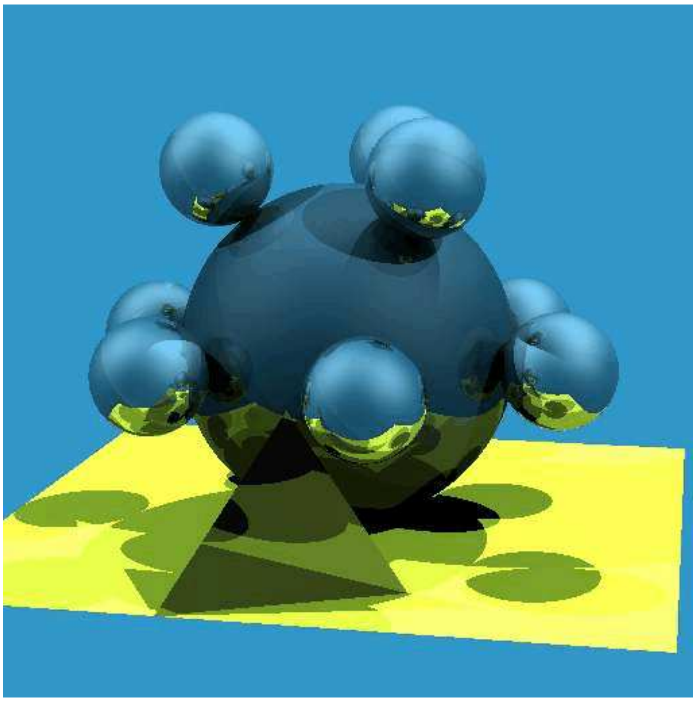
Cone. Prisekani stožec je lahko v posebnem primeru tudi valj. Pri določanju preseka je potrebno upoštevati, da imamo vedno tri možne površine in dva preseka.

PPatch. Mnogokotniški zlepek v vsakem vozlišču podaja tudi normalo. Točke preseka določamo podobno kot pri primitivu *Polygon*. Normale v točki pa linearno interpoliramo.

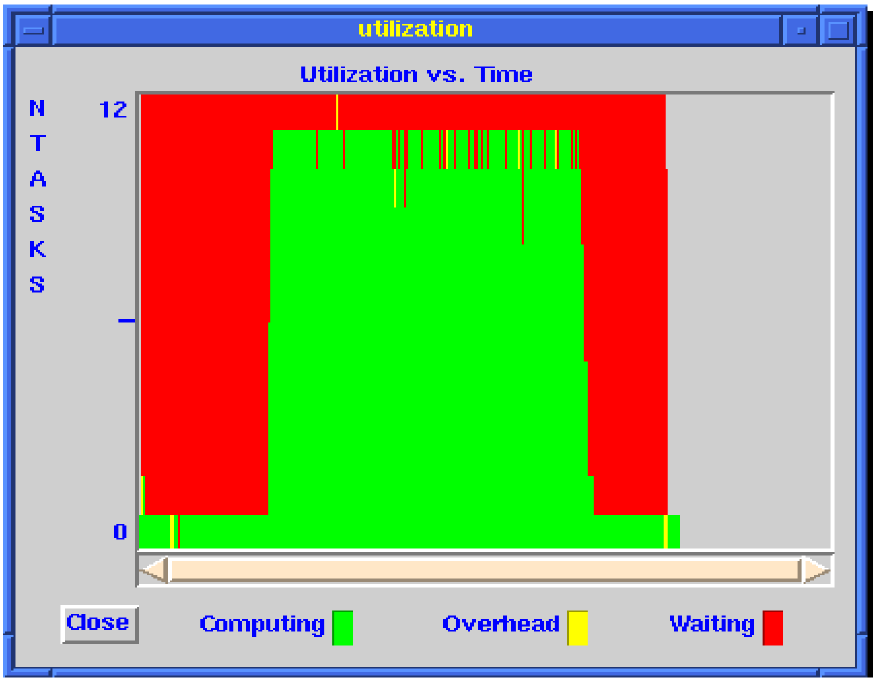
Dodajanje novih primitivov v program je enostavno. Potrebno je le izpeljati primitiv in poskrbeti za vnos v bazo primitivov in izdelati vitrualni metodi za presek in normalo.

Razred *Scene* združuje vse osnovne gradnike scene v enoten razred. Skrbi za sestavo baze primitivov in luči. Vsebuje metodi za rasterizacijo filma kamere (*Screen()*) in rekurzivno sledenje žarku (*Trace()*). Rekurzivni algoritem je izveden po Whittedovem algoritmu z utežnostno kontrolo smiselnosti generiranja novih žarkov. Rezultat rekurzivnega programa za sledenje žarku se zapiše v prenosljivem formatu PPM[MvR94] (*Portable Pixel Map*). PPM format datoteke je zelo razširjen in je bil zato tudi uporabljen.

Podrobnejši opis posameznih razredov in pripadajočih metod je podan v dodatku A. Na sliki 7.2 so prkazane osnovne lastnosti izdelanega programa za sledenje žarku: zrcalni odboj, razpršitev, prozornost in sence; manjše krogle imajo visok koeficient zrcalnosti, osrednja krogla razpršuje svetlobo na površini, trikotnik je prozoren, z lomnim količnikom $i_r = 1.4$.



Slika 7.2: Krogle z visoko stopnjo zrcalnosti in prozorni trikotnik



Slika 7.4: Izkoristek procesov

Na sliki 7.4 je prikazana obremenitev procesorjev v odvisnosti od časa. Razvidna sta dva področja na začetku in na koncu, kjer je virtualni stroj razmeroma neobremenjen. Začetni čas mirovanja je posledica kreiranja procesov delavcev in čas potreben za pošiljanje datoteke scene delavcem.

Ker so datoteke scene lahko zelo velike, je bila uvedena "on-line" kompresija podatkov. Scena za opis testne slike 7.3 je bila velika 557331 bytov, stisnjena velikost, ki se je poslala po mreži pa je bila velika 89725 bytov. Pomembno je poudariti, da je za glavni sistem, kjer se poganja vodja, bolje uporabiti čim hitrejši računalnik, ki hitro stisne datoteko podatkov in s tem prihrani na času.

Časovna porazdelitev pošiljanja sporočil je prikazana na sliki 7.5. Prikazan je zagon programa nadzornika na PC računalniku, ki je nekajkrat počasnejši od najhitrejše delovne postaje. Primer je bil izbran zaradi tega, da se pokaže uravnoteženje in pa jasno razviden čas na koncu, kjer so končali vsi procesi razen enega, ki je bil pognan na zelo obremenjenem računalniku. Dokler program ni končal, ni bilo možno v glavnem programu sestaviti celotne slike. Primer je netipičen in je t.i. "worst-case", ki kaže lastnosti implementiranega balansiranega algoritma. Čas potreben za izračun testne slike je bil 84 sekund zaradi tega, ker je bil PC namenoma obremenjen in je zato izračunal le dva pasova, medtem ko so ostali PC-ji 4 in več.

7.2.1 Rezultati

Opravljeno je bio več hitrostnih testov, ki so pokazali uspešnost distribiuranega modela metode sledenja žarku. Uporabljena je bila sledeča strojna oprema:

HP 737 HP-UX 9.03m, 80 MB RAM-a, 99 MHz PA

SGI Indigo 2 Irix 5.2, 80 MB RAM-a, 150 MHz R4400

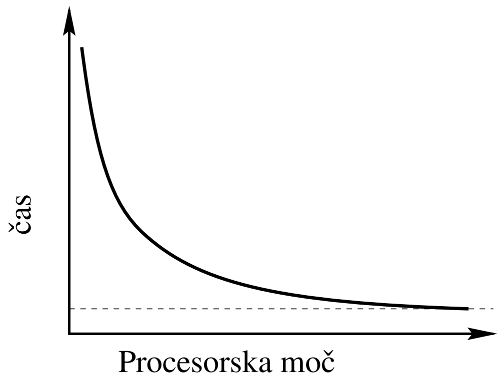
SGI Indy SC Irix 5.2, 48 MB RAM-a, R4000 – $2\times$

HP 712 HP-UX 9.03m, 48 MB RAM-a, PA – $2\times$

PC 486 Linux 1.2, 16 MB RAM-a, 66 MHz, – $4\times$

PC 486 Linux 1.2, 16 MB RAM-a, 100 MHz

V tabeli 7.1 in 7.2 sta prikazana dva primera testov, iz katerih so razvidne relativne hitrosti posameznih računalnikov. Ker so bile arhitekture zelo različne, ni smiselno prikazati odvisnosti absolutnega časa od števila uporabljenih računalnikov, ampak le od količine uporabljene procesorske moči. Na sliki 7.6 je prikazan potreben čas za upodabljanje v odvisnosti od skupne procesorske moči navideznega računalnika. Vedno bo obstajal nek minimalni čas, zaradi prenosa podatkov, kot tudi zaradi interpretiranja datoteke scene. Pri skalabilnih algoritmih, kot je metoda sledenja žarku, je možno predvideti prispevek enega računalnika k zmogljivosti paralelnega sistema. Čeprav računska moč sistema z dodajanjem novih procesorjev raste linearno, se čas potreben za rešitev naloge zmanjšuje asimptotično. Prispevek enega računalnika je torej največji, ko imamo dva računalnika v paralelnemu sistemu.



Slika 7.6: Potreben čas za upodabljanje v odvisnosti od skupne procesorske moči navideznega računalnika

Iz tabele, kjer je prikazan model PVM, sestavljen iz samih osebnih računalnikov, vidimo tudi razlike zaradi različnih hitrosti in obremenitev. Na enem samem računalniku PC486/66 je model iz slike 7.3 potreboval 14 minut. Polni sistem enajstih računalnikov pa 57 sekund.

Poglavje 8

Zaključek

V tem delu sem poskušal postaviti upodabljanje tridimenzionalnih objektov v uporabno matematično formulacijo in prikazati pregled prejšnjih in tekočih rešitev, tako v upodabljanju kot tudi v praktični uporabi distribuiranih sistemov. Prikazane so optične razmere na površinah in njihova matematična formulacija, ki jo lahko, tako kot je predstavljena, direktno uporabimo v programih za upodabljanje. Diskutirani so bili različni sistemi, ki omogočajo razvoj paralelnih aplikacij. Od vseh je imel le sistem PVM vse potrebne rutine, ki omogočajo dinamično kontrolo procesov in s tem enostavno prenašanje programov na različne platforme.

Predstavljena je bila enostavna shema, ki učinkovito uravnava obremenitev računalnikov pri uvajanju metode sledenja žarku. Jedro operacijskega sistema Unix že samo po sebi skrbi za pravilno razporeditev obremenitve procesorja in dodeljuje čas vrsti procesov, ki stalno tečejo na sistemu. S predvkupnim procesiranjem je mogoče nastavljati prioriteto procesom, tako da minimalno obremenjuje delo na gostujočem računalniku. Z uporabo sistema PVM ni potrebno imeti velikih privilegijev na gostujočem računalniku. V večini primerov zadostuje že klasična zaščita, kot jo ima navadni uporabnik na sistemu.

Kar nekaj problemov je bilo potrebno preseči v procesu razvoja distribuiranega sistema. Težišče je bilo na prilagoditvi obstoječega programa za upodabljanje z ustrezno balansirano shemo. Upoštevati je bilo potrebno tudi komunikacijske omejitve. Pri razvoju distribuiranega sistema so bile vedno v mislih kompleksne scene, ki jih je bilo potrebno posredovati po omrežju prav do vseh vozlov v omrežju. V ta namen je bila izvedena sprotna kompresija podatkov, ki je učinkovito zmanjšala čas prenosa in s tem razbremenila omrežje in povečala celotno učinkovitost paralelnega programa.

Žal je v računalništvu tako, da se teoretični algoritmi največkrat zaradi ozkega vpogleda v problem, kot tudi zaradi pomanjkanja časa, največkrat potrdijo le na minimalnih sistemih. Za razvoj novega znanja je potrebna velika količina temeljnih znanj z različnih področij, v prvi vrsti pa računalniških tehnologij, katere si pridobimo le s praktičnim delom. Razvoj testnega okolja za distribuiran sistem je odprl mnoge nove možnosti za nadaljnje raziskave. Ne bi želel napovedovati, kaj te možnosti lahko prinesejo, ampak bi rad le omenil probleme, ki bi bili vredni podrobnejšega vpogleda in tiste, za katere ni bilo časa

Literatura

[AK91] James Arvo in David Kirk. *A Survey of Ray Tracing Acceleration Techniques*, Poglavje 6, strani 201–262. V knjigi Introduction to Ray Tracing, glej [Gla91b], 1991.

[Arv91] James Arvo, urednik. *Graphics Gems II*, Zbornik 2 v *Graphics Gems Series*. Academic Press, Inc., 1991.

[Ash94] Ian Ashdown. *Radiosity: a programmer's perspective*. John Wiley & Sons, 1994.

[AT&87] AT&T. *UNIX System V Programmer's Guide*. Prentice-Hall, Inc., 1987.

[Bar81] A. Barr. Superquadrics and angle-preserving transformations. *IEEE Computer Graphics and Applications*, 1(1):11–23, 1981.

[Bej93] Adrian Bejan. *Heat transfer*. John Wiley and Sons, 1993.

[Blo92] John Bloomer. *Power Programming with RPC*. O'Reilly & Associates, Inc., 1992.

[BqDy89] Su Bu-qing in Liu Ding-yuan. *Computaional Geometry – Curve and Surface Modeling*. Academic Press, 1989.

[BW89] Max Born in Emil Wolf. *Principles of Optics*. Pergamon Press, 6. izdaja, 1989.

[Cal94] R. E. Callan. *Building Object-oriented Systems*. Computational Mechanics Publications, Southampton, UK, 1994.

[Cor90] John R. Corbin. *The Art of Distributed Applications*. Springer-Verlag, 1990.

[Cot88] Marion Cottingham. Pseudo ordering of CSG-trees. V David Duce, urednik, *Eurographics 88*, strani 471–482. Elsevier Science, 1988.

[Cra94] Bob Crambit. Rendering on the network. *Computer Graphics World*, strani 26–28, Avgust 1994.

[Hec91] Paul S. Heckbert. *Writing a Ray Tracer*, Poglavje 4, strani 264–293. V knjigi Introduction to Ray Tracing, glej [Gla91b], 1991.

[Hec94] Paul S. Heckbert, urednik. *Graphics Gems IV*, Zbornik 4 v *Graphics Gems Series*. Academic Press, Inc., 1994.

[Hew92] Hewlett Packard. *Programming on HP-UX*. Hewlett Packard, 1992.

[HG83] Roy A. Hall in Donald P. Greenberg. A Testbed for Realistic Image Synthesis. *IEEE Computer Graphics And Applications*, 3:10–20, November 1983.

[Kir92] David Kirk, urednik. *Graphics Gems III*, Zbornik 3 v *Graphics Gems Series*. Academic Press, Inc., 1992.

[Kla89] Rudolf Kladnik. *Valovni pojavi – Akustika in optika*, Zbornik 3 v *Visokošolska fizika*. Državna založba Slovenije, 1989.

[LMB92] John R. Levine, Tony Mason in Doug Brown. *lex & yacc*. O'Reily & Associates, Inc., druga izdaja, 1992.

[Lud93] Thomas Ludvig. Load Managment in Multiprocessor Systems. V Arndt Bode in Mario Pal Cin, urednika, *Parallel Computer Architectures*, Lecture notes in Computer Science. Springer Verlag, 1993.

[Mic87a] Sun Microsysterms. RPC: Remote Procedure Call Protocol Specification. Tehnično poročilo RFC 1057, Sun Microsystems, 1987.

[Mic87b] Sun Microsysterms. XDR: External Data Representation Standard. Tehnično poročilo RFC 1014, Sun Microsystems, 1987.

[Mic93a] Sun Microsystems. Remote Procedure Call Programming Guide. Tehnično poročilo, Sun Microsystems, 1993.

[Mic93b] Sun Microsystems. Rpcgen Programming Guide. Tehnično poročilo, Sun Microsystems, 1993.

[Mor85] Michael E. Mortenson. *Geometric Modeling*. John Wiley & Sons, 1985.

[Mul87] Sape J. Mullender. Process Managment in a Distributed Operating System. V J. Nehmer, urednik, *Experiences with Distributed Systems*, Lecture notes in Computer Science. Springer Verlag, 1987.

[Mul94] Ketan Mulmuley. *Computational Geometry: An introduction through randomized algorithms*. Prentice-Hall Inc., 1994.

[MvR94] James D. Murray in William van Ryper. *Encyclopedia of Graphics File Formats*. O'Reily & Associates, Inc., prva izdaja, 1994.

[Thy92] S. T. Thynell, urednik. *Developments in radiative heat transfer*, Zbornik 203 v *HTD*, New York, Avgust 1992. National heat transfer conference.

[TTOO90] Atsushi Takagi, Hiroshi Takaoka, Tesuya Oshima in Yoshinori Ogata. Accurate technique based on colorimetric conception. V *Computer Graphics*, Zbornik ACM Siggraph '90 Conference Poceedings, strani 263–272, Avgust 1990.

[Uli90] Robert Ulichney. *Digital Halftoning*. The MIT Press, 1990.

[Wat90] Alan Watt. *Fudamentals of Three-Dimensional Computer Graphics*. Addison Wesley, 1990.

[Wat94] Allan Watt. *3D Computer Graphics*. Addison Wesley, druga izdaja, 1994.

[WCF92] Cristopher D. Watkins, Stephen B. Coy in Mark Finlay. *Photorealism and Ray Tracing in C*. M&T, 1992.

[WGS94] Ewing Lusk William Gropp in Anthony Skjellum. *Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engineering Computation series. The MIT Press, 1994.

Dodatek A

Razredi programa za sledenje žarku

Osnovni program za sledenje žarku je napisan v objektno orientiranem jeziku C++ v katerem je "razred" temeljni gradnik modularnega programa. Razredi združujejo podatke in postopke v zaključeno celoto z različnimi stopnjami zaščite podatkov in metodami, ki operirajo s podatki.

Opisi razredov so izvlečeni iz izvorne kode, ki je bila ustrezno komentirana in nato pretvorjena v obliko HTML in LaTeX.

A.1 Scene

Glavni razred za program metodo sledenja žarku. Omogoča rekurzivno sledenje po Whittedovi metodi. Vhodni podatki so podani v datoteki formata NFF (*Neutral File Format*), ki jo intepretiramo z *yyparse()* (generator YACC). Razred združuje vse objekte in luči v neurejenih seznamih (*Bag*), ki lahko neomejeno rastejo. Količina podatkov je tako omejena le z razpoložljivim pomnilnikom. S kamero (**Camera**) so nastavljeni parametri gledanja in vse ostalo, kar je potrebno za upodobitev.

Definicija

```
class Scene
{
protected:
 PrimitivePtrSLBag primitives;  // neurejen seznam primitivov
 LightPtrSLBag lights;          // neurejen seznam luči
 RGBColor background;           // barva ozadja
 Camera camera;                 // sistem gledanja

 double minWeight;              // meja pomembnosti žarka
 double rayEps;                 // zaokrožitvena napaka
 int maxDepth;                  // največja globina rekurzije
```

`void` **Scene**`::Describe()`

Opiše lastnosti scene. Na *cerr* izpiše vsebujoče objekte, lokacije luči in parametre gledanja. Uporablja se pri testiranju in kontroli s stikalom -d pri zagonu programa.

A.2 Object

Ta razred je osnova za **Camera** in **Light**. To je abstraktni razred in ima lahko le izpeljane primere. Združuje lastnosti, ki so skupne vsem objektom. Virtualna metoda *Describe* izpiše lastnosti objekta.

Podrazredi

Primitive

Definicija

```
class Object
{
protected:
 vec3 origin;
 mat4 orient;

public:

 void setPosition(const vec3& position);
 void setPosition(const double position[3]);
 vec3 position() const { return origin; };
 void setOrientation(const mat4& o) { orient = o;};
 mat4 orientation() const { return orient; };
 virtual void Describe(int indent) = 0;
};
```

Metode

`void` **Object**`::setPosition(const` **vec3**`& position)`

Nastavi pozicijo objekta. Vektor *position* podaja najbolj značilne koordinate objekta (izhodišče).

`void` **Object**`::setPosition(const double position[3])`

Nastavi pozicijo objekta. Vektor *position* podaja najbolj značilne koordinate objekta (izhodišče).

`void` **Object**`::Describe(int indent)`

Izpiše osnovno lastnost objekta – izhodišče

`typedef` **vec3 RGBColor**`;`

Definicija barve RGB, ki je v mejah of 0.0 do 1.0

ločljivost filma definirana v točkah (*resx, resy*), oddaljenost *hither* nima vpliva na sistem gledanja.

`void` **Camera**`::Describe(int indent)`

Izpiše lastnosti kamere. Te so: položaj, smer projekcije, vektorja u in v na filmu, ločljivost in maksimalne koordinate (u,v).

vec3 `Camera::pointToRay(` **vec2**`& p)`

Ta metoda vrne normirano smer vektorja, če gledamo na izbrano točko na filmu kamere, ki je v mejah ločljivosti filma. Koordinate p so v rastrskih enotah, vendar lahko izberemo tudi dele velikosti rastra, kar je uporabno pri "supersamplingu". Izhodišče žarka dobimo z getPosition() kamere.

A.4 Light

Ta razred vpeljuje enostaven točkasti svetlobni izvor.

Definicija

```
class Light: public Object
{
protected:

  vec3 col;      // barva svetlobnega izvora

public:

  vec3 color() { return col; };
  void setColor(RGBColor& color);
  void setColor(double color[3]);
  virtual void Describe(int indent);
};
```

Metode

`void` **Light**`::setColor(`**RGBColor**`& color)`

Nastavi barvo luči s trojčkom RGB.

`void` **Light**`::setColor(double color[3])`

Nastavi barvo luči s trojčkom RGB.

`void` **Light**`::Describe(int indent)`

Izpiše osnovne lastnosti luči.

`double` **Primitive**`::closestIntersection(const double *x,`
`int xNum)`
Iz vektorja števil *x* velikosti *x_num* poišče najmanjše število in ga vrne kot rezultat.

`void` **Primitive**`::Describe(int indent)`
Izpiše osnovne lastnosti primitiva in objekta. Barva in lastnosti površine so skupne vsem primitivom.

`typedef` **Primitive**`*` **PrimitivePtr**`;`
Definicija kazalca na **Primitive**, ki je potrebna pri zbirki (*SLBag*)

A.6 Sphere

Krogla je najenostavnejši primitiv.

Definicija

```
class Sphere : public Primitive
{
protected:
 double radius;

public:
  Sphere(double origin[3], double radius, double color[3],
         double prop[5]);
  virtual double nearestIntersection( vec3& rayDir) const;
  virtual  vec3& p) const;
  virtual void Describe(int indent);
};
```

Metode

Sphere`::`**Sphere**`(double origin[3], double radius,`
`double color[3], double surfaceProperty[5])`
Naredi novo kroglo

`double` **Sphere**`::nearestIntersection(`**vec3**`& rayDir )`
Ta metoda izračuna preseke med žarkom in kroglo. Vrne najbližje presečišče ali 0.0, če presečišče ne obstaja.

vec3 **Sphere**`::normalAt (`**vec3** `& p)`
Ta metoda izračuna normalo na krogli v točki *p*, ki je na površini. Normalizacija je vgrajena v enačbi za kroglo.

metoda določanja notranjosti točke. Če presek žarka in mnogokotnika obstaja, funkcija vrne razdaljo od *rayOrg* do presečišča. Če presek ne obstaja, vrne vrednost nič.

vec3 Polygon::normalAt(**vec3**& /* p */)

Vrne normalo mnogokotnika. Ker se normala mnogokotnika ne spreminja, je lahko točka *p* poljubna in se ne upošteva pri določanju normale. Orientacija in smer normale je določena s prvimi tremi vozlišči mnogokotnika.

void **Polygon**::Describe(int indent)

Na izhodni tok *clog* izpiše vse pomembne lastnosti mnogokotnika. Te so: število vozlišč, koordinate vozlišč, lastnosti površine in normala.

A.8 vec3

Tridimenzionalni vektor

Definicija

```
class vec3
{
protected:

  double n[3];

public:

  //Konstruktorji

  vec3 ();
  vec3 (const double x, const double y, const double z);
  vec3 (const double d);
  vec3 (const vec3 & v);              //kopija
  vec3 (const vec2 & v);              //konverzija v2 v v3
  vec3 (const vec2 & v, double d);    //konverzija v2 v v3
  vec3 (const vec4 & v);              //konverzija v4 v v3
  vec3 (const vec4 & v, int dropAxis); //konverzija v4 v v3

  // Prireditveni operatorji

  vec3 & operator = (const vec3 & v);  //prireditev vektorja vec3
  vec3 & operator += (const vec3 & v); //prištevanje vektorja vec3
  vec3 & operator -= (const vec3 & v); //zmanjšanje za vec3
```

```
};
```

Metode

vec3::vec3 `()`
Tridimenzionalni konstruktor

vec3::vec3 `(const double x, const double y, const double z)`
Kreira vektor iz podanih konstant x, y, in z

vec3::vec3 `(const double d)`
Diagonalni vektor

vec3::vec3 `(const` **vec3** `& v)`
Kopija

vec3::vec3 `(const` **vec2** `& v)`
Konverzija dvodimenzionalnega vektorja v v tridimenzionalni homogeni koordinatni sistem. $z == 1.0$

vec3::vec3 `(const` **vec2** `& v, double d)`
Konverzija dvodimenzionalnega vektorja v v tridimenzionalni homogeni koordinatni sistem. $z ==$ d

vec3::vec3 `(const` **vec4** `& v)`
Konverzija štiridimenzionalnega vektorja v (homogeni vektor) v tridimenzionalni kartezijev koordinatni sistem.

vec3::vec3 `(const` **vec4** `& v, int dropAxis)`
Konverzija štiridimenzionalnega vektorja v v tridimenzionalni koordinatni sistem. *dropAxis* podaja indeks koordinate, ki se izloči.

vec3 `&` **vec3::**`operator = (const` **vec3** `& v)`
Prireditveni operator

vec3 `&` **vec3::**`operator += (const` **vec3** `& v)`
Prištevanje vektorja obstoječemu vektorju.

vec3 `&` **vec3::**`operator -= (const` **vec3** `& v)`
Odštevanje vektorja obstoječemu vektorju.

vec3 `&` **vec3::**`operator *= (const double d)`
Množenje vektorja s konstanto d.

vec3 `&` **vec3::**`operator /= (const double d)`
Deljenje vektorja s konstanto d.

`double &`**vec3::**`operator[](int i)`
Izbor ene koordinate vektorja. Možne vrednosti: VX, VY in VZ.

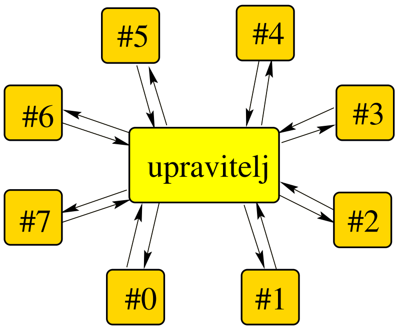
Dodatek B

Algoritem distribuiranega programa

V tem dodatku je podrobneje predstavljen distribuirani program, ki je bil praktično izveden. Sistem je izdelan iz dveh ločenih programov:

Upravitelj (master) skrbi za nadzor delavcev. Dodeljuje naloge in zbira rezultate. Po končani skupni nalogi sledenja žarku, združi rezultate v sliko in jo zapiše v datoteko.

Delavec (worker) Izračunava dele slike v obsegu, ki mu ga dodeli nadzornik. Kot rezultat vrne izračunani del slike v obliki toka trojic RGB.



Slika B.1: Komunikacijska shema distribuiranega programa

Na sliki B.1 je razvidno, da komunikacija poteka le med nadzornikom in delavci. V sistemu PVM s heterogeno mrežno arhitekturo je možno poganjati več delavcev na enem vozlišču (računalnik), vendar to ne prispeva k hitrejšem računanju, saj se procesorska moč v tem primeru razdeli.

Za razliko od delavca, upravitelj le malo obremenjuje procesor, saj skrbi predvsem za komunikacijo in nadzor del. Tako je možno brez problemov na enem računalniku poganjati upravitelja in enega delavca. Upravitelj lahko spremlja delo delavcev in skrbi za alokacijo prostih računskih zmogljivosti. Ta lastnost je v sistemu PVM uporabna za toleriranje napak (*fault tolerant*) in dinamično dodajanje/odvzemanje strežnikov v navidezni računalnik PVM.

Večina programov v sistemu PVM najprej zahteva od strežnika na katerem se program požene, da mu sporoči številko procesa (*Task ID*). To je celoštevilčna spremenljivka, ki se uporablja za komunikacijo med procesi. Delavci jo npr. uporabljajo za vračanje rezultatov upravniku del. Druga pomembna uporaba te spremenljivke se pokaže v razhroščevanju programov. Ker ta številka enožnačno določa številko procesa, lahko s posebnimi razhroščevalci sledimo sporočilom procesa v navideznem računalniku.

```
mytid = pvm_mytid ();
```

Ker je sistem PVM vezan na operacijski sistem in na komunikacijo ni zagotovljena izvršitev nobene funkcije v navideznem računalniku. Programer nosi odgovornost, da zagotovi pravilno delovanje distribuiranega programa tako, da kontrolira kritične funkcije. Vsaka funkcija PVM vrača tudi status izvršbe. Če je celoštevilčna vrednost negativna se je pojavila napaka, ki jo je potrebno ustrezno odpraviti. Te napake se lahko rešujejo na različne načine. Pri redundančnih sistemih se iščejo nadomestni računski viri. Druga možnost je čakanje na sprostitev virov (ponavljanje izvršbe funkcije), vendar se lahko tu pojavijo mrtve (neskončne) zanke (*deadlock*). Nekatere funkcije, ki so vezane na zakasnitve v komunikacijah med računalniki, imajo tudi možnost določitve največjega časa čakanja na sprostitev komunikacijskih virov (*timeout*). Pravilo je, da se preveri pravilnost delovanja PVM vsaj na začetku programa.

```
if (mytid < 0) /* Kontrola napake */
  {
    pvm_perror(argv[0]);
    return -1;
  }
```

Ker je večino argumentov ukazne vrstice distribuiranega programa namenjeno prav delavcem, jih nadzornik le pregleda in izloči ukaze, ki so namenjene njemu. Iz osnovne tabele argumentov nato zgradi okrnjeno število argumentov, ki se kasneje posredujejo vsem delavcem navideznega računalnika.

```
parse_options(argc, argv);
```

Distribuirani program je izdelan tako, da ugotovi kateri računalniki so v navideznemu računalniku. Preveri delovanje delavcev in šele nato zgradi strategijo razporejanja nalog. Ker so v navideznem računalniku lahko zelo različne procesorske zmogljivosti, je potrebno tudi pravilno uravntežiti naloge, ki vodijo k čim večji izkoriščenosti navideznega računalnika. V trenutni verziji (v3.3) sistem sicer predvideva razpoznavanje relativnih hitrosti posameznih vozlišč, vendar to zaenkrat še ni implementirano. Predvideno je tudi hevristično določanje ciljnih računalnikov s prostimi kapacitetami. Ker pa to v sistemu PVM še trenutno še ni implementirano, je bilo potrebno izdelati ustrezen algoritem, ki sam razpozna razlike med računskimi močni vozlišč navideznega računalnika.

```
info = pvm_config(&nhost, &narch, &hostp);
```

Če podatki o navideznem računalniku niso dostopni, se nadaljnje akcije prekinejo. Pred tem se še izpiše informacija o vrsti napake, da je odpravljanje nepravilnosti pri zagonu navideznega računalnika lažje. Na vsakem računalniku sistema PVM mora teči demon (*PVM daemon*), ki skrbi za zagon programov (delavcev).

```
if (info < 0) /*Če se pvmd upravnika ne odziva */
  {
    pvm_perror(argv[0]);
    return -1;
  }
```

Izpis lastnosti navideznega računalnika vsebuje tudi lastnosti, ki trenutno še niso implementirane (relativne hitrosti, obremenjenost računalnikov).

```
if (verbose)
  {
    printf ("Currently available %d hosts with %d data type(s):\n",
            nhost, narch);       for (i = 0; i < nhost; i++)
      {
        printf("t%x\t%20s %15s %10d\n", hostp[i].hi_tid,
               hostp[i].hi_name, hostp[i].hi_arch,
               hostp[i].hi_speed);
      }
  }
```

Ker sestava navideznega računalnika vnaprej ni poznana, je potrebno dinamično alocirati vsa spominska polja, ki so odvisna od števila vozlišč.

```
servtids  = (int *) malloc(nhost * sizeof(int));
servflist = (int *) malloc(nhost * sizeof(int));
servjobs  = (int *) malloc(nhost * sizeof(int));
servtiles = (int *) calloc(nhost, sizeof(int));
```

Na začetku del so vsi strežniki prosti vseh del.

```
servfree = -1;
```

Sistem PVM omogoča vzporeden zagon vseh delavcev.

```
numt = pvm_spawn (CHILDNAME, argv + 1, PvmTaskDefault, "", nhost, servtids);
```

Ker ni nujno, da obstajajo delavci na vseh računalnikih, je potrebno preveriti, kateri računalniki so se odzvali na poziv in izločiti iz obravnavanja neustrezne delavce.

```
if (numt < 0)
  {
    pvm_perror ("Error spawning slaves.");
    pvm_exit();
    return -1;
  }
```

```
  if (buffer == NULL)
    {
      fputs("Out of memory in send_scene()", stderr);
      return -2;
    }
```

Za stiskanje podatkov podatkov se uporabi enostaven pristop k medprocesni komunikaciji s pomočjo cevi. Uporabnik ima možnost izbire standardnega programa za kompresijo *compress* ali boljšega kot je *gzip*.

```
#if COMPRESSED
  {
    char command[MAXFILENAME + 20];
    sprintf(command, "%s -c %s", COMPRESSPRG, file_name);
    f = popen(command, "r");
  }
#else
  f = fopen(file_name, "r");
#endif

  if (f == NULL)
    return -1;

  if (verbose)
    puts("Reading and compressing scene file...");


  while ((file_size = fread(buffer + position, 1, SCENEBUFSIZE, f)) != 0)
    {
      position += file_size;
      if ( (position + SCENEBUFSIZE) > length)
        {
          length *= 2;
          buffer = realloc(buffer, length);
          if (buffer == NULL)
            {
              fputs("Out of memory in send_scene()", stderr);
              return -2;
            }
          if (verbose > 2)
            fprintf(stderr, "+%ld", file_size);

        }
    }

#if COMPRESSED
  pclose (f);
#else
  fclose (f);
#endif

  file_size = position;
```

Na začetku se izračuna celotno število nalog in določijo prosti delavci.

```
njobs = height/tile_height + 1;
jobstogo = njobs;
jobnext = njobs;
```

Alocira se matrika za pomnjenje rezultatov, saj le ti lahko prihajajo v neurejenem vrstnem redu. Naloga upravitelja je, da pravilno zlaga prispele rezultate.

```
 image = (char **) malloc(njobs * sizeof(char *));

if (verbose)
  puts("Claiming tiles [job@serverto:first-last] from workers...");
```

Na začetku razpošlejmo delo vsem prostim delavcem.

```
 while (servfree >= 0 && jobnext)
   send_nextjob ();
```

Po razpošiljanju čakamo na rezultate delavcev. Takoj, ko delavec konča z enim delom, dobi drugo še neobdelano nalogo. S takim pristopom je zagotovljeno uravnoteženo računanje, ki se lahko dinamično spreminja, tudi če se zaradi drugih procesov spreminjajo obremenitve v računalniku.

```
if (verbose)
  {
    printf("\nReceiving tiles <job>...");
    fflush(stdout);
  }
 while (jobstogo)
   claim_tile ();
```

Po končanem delu izpišemo statistiko celotnega dela navideznega računalnika.

```
if (verbose)
  {
    int i;
    printf("\nServer statistics:\n");
    puts("NUM    HOST  DTID  TID ARCH TILES [ \% ]\n");
    for (i = 0; i < nserv; i++)
      {
        int s;
        int dtid;

        dtid = pvm_tidtohost(servtids[i]);

        for(s = nhost; s-- > 0; )
          if (hostp[s].hi_tid == dtid)
            break;
        if (s < 0)
          {
            fprintf(stderr, "unknown host #%d\n", i);
            continue;
          }
```

Zajem rezultatov

Sledeči podprogram zbira rezultate in dodeljuje nove naloge delavcem.

```
static int
claim_tile ()
{
  int info, bufid, tid, y1, y2, size, job, s;

  /* Čakaj na rezultat kateregakoli delavca */
  bufid = pvm_recv (-1, MSGDATA);
  pvm_bufinfo(bufid,  (int *)0,  (int *)0, &tid );
```

Iz sporočila je potrebno izločiti informacijo o pošiljatelju.

```
  for(s = nserv; s-- > 0; )
    if(tid == servtids[s])
      break;

  if ( s < 0 )
    {
      fprintf (stderr, "bogus tile data message from t%x ?\n", tid);
      fputs("Valid tids:", stderr);
      for(s = 0; s < nserv; s++)
        printf("<t%x>", servtids[s]);
      fputs("\n", stderr);
      return -1;
    }
```

Odpakira rezultate in jih shrani na ustrezno mesto.

```
  pvm_upkint(&job, 1, 1);
  pvm_upkint(&y1, 1, 1);
  pvm_upkint(&y2, 1, 1);
  size = (y2 - y1) * width * 3; /* število RGB trojic */
  image[job] = (char *) malloc( size );
  info = pvm_upkbyte(image[job], size, 1);

  servtiles[s]++; /* statistika porabe računalnikov */
  jobstogo--;

  if (verbose)
    {
      printf("<%d>", job);
    }
```

Označi strežnik kot prosti

```
  servflist[s] = servfree;
  servfree = s;

  /* Dodeli novo delo, če ga je še kaj */
  if ( jobnext )
      send_nextjob();

  return 0;
}
```

```
  fflush (f);
  fclose (f);
  return 0;
}
```

B.2 Delavec

Program delavca je ločen od izvršnega programa upravitelja. Največ procesorskega časa se v navideznem računalniku porabi prav na delavcih.

```
extern unsigned int Options;
extern int First_Line, Last_Line;
extern int First_Column, Last_Column;

MAIN_RETURN_TYPE alt_main PARAMS((int argc, char **argv));

static char *calc_tile PARAMS((int y1, int y2));
static int width;
char   *PixelMap;              /* izračunana slika */
int     PixelMapSize;

int main(argc, argv)
        int argc;
        char **argv;
{
  int mytid;                                  /* ID delavca */
  int mastertid;                              /* ID upravitelja */
  FILE *f;
  char *buffer;
  long file_size;

  static char filename[MAXFILENAME];

  PixelMapSize = 0;
  PixelMap = NULL;

  mytid = pvm_mytid();

  mastertid = pvm_parent();

  /* Sprejem podatkov */
  pvm_recv(mastertid, MSGSCENE);
  pvm_upkstr(filename);
  pvm_upklong(&file_size, 1, 1);
  buffer = malloc(file_size);
  pvm_upkbyte(buffer, file_size, 1);

#if COMPRESSED
  {
    char command[MAXFILENAME + 20];
    sprintf(command, "%s -d -c > %s", COMPRESSPRG, filename);
    f = popen(command, "w");
  }
#else
  f = fopen(filename, "w");
#endif
```

```
    }
  fprintf(stderr, "error receiving work to do\n");
  pvm_exit();
  exit(1);
}
```

B.2.1 Izračun pasu

Naloge delavcev se izračunavajo po pasovih v obsegu, kot ga ukaže upravitelj.

```
static char *
calc_tile (y1, y2)
int y1, y2;
{
  PixelMap = (char *)malloc ((y2 - y1) * width * 3);

  First_Line  = y1;
  Last_Line   = y2;

  Start_Tracing ();

  return PixelMap;
}
```

B.3 Primer izmenjave sporočil

```
[t4000c] Currently available 11 hosts with 2 data type(s):
[t4000c] t40000                     achilles       LINUX    1000
[t4000c] t80000           lecad.fs.uni-lj.si        SGI5    1000
[t4000c] tc0000           aphro.fs.uni-lj.si        HPPA    1000
[t4000c] t100000          paris.fs.uni-lj.si        SGI5    1000
[t4000c] t140000         hector.fs.uni-lj.si        SGI5    1000
[t4000c] t180000           aiax.fs.uni-lj.si       LINUX    1000
[t4000c] t1c0000          diana.fs.uni-lj.si        HPPA    1000
[t4000c] t200000         helena.fs.uni-lj.si        HPPA    1000
[t4000c] t240000     prometheus.fs.uni-lj.si       LINUX    1000
[t4000c] t280000          sinon.fs.uni-lj.si       LINUX    1000
[t4000c] t2c0000        ullysses.fs.uni-lj.si      LINUX    1000
[t4000c] Reading and compressing scene file...
[t4000c] Multicasting scene 'balls.pov' size = 89725
[t4000c] Claiming tiles [job@serverto:first-last] from workers...
[t4000c] [102@10:507-511][101@9:502-506]
[t4000c] [100@8:497-501][99@7:492-496][98@6:487-491]
[97@5:482-486][96@4:477-481][95@3:472-476][94@2:467-471]
[93@1:462-466][92@0:457-461]
[t4000c] Receiving tiles <job>...
[t4000d] Parsing...
[tc0005] Parsing...
[t100005] Parsing...
[t140005] Parsing...
[t80005] Parsing...
[t1c0005] Parsing...
```

Dodatek C

Klicanje oddaljenih procedur – RPC

Remote Procedure Calling je eden prvih sistemov za razvoj distribuiranih sistemov na omrežju z Internet Protokolom[Blo92]. Klicanje oddaljenih procedur je uporabno zaradi tega, ker omogoča programerjem, da razvijajo aplikacije, ki skrivajo podrobnosti v komunikacijah med različnimi računalniki in hkrati omogočajo vse potrebne prijeme pri komuniciranju. To poglavje opisuje osnovne značilnosti komuniciranja, ki jih je potrebno poznati tudi pri drugih distribuiranih sistemih, saj večina distribuiranih sistemov gradi prav na tej knjižnici.

C.1 Razvoj aplikacij s knjižnico RPC

S knjižnico RPC, eno najbolj razširjenih knjižnic za pisanje distribuiranih aplikacij, je možno izdelati programe, ki tečejo na različnih računalnikih, kar pomeni, da tečejo na različnih operacijskih sistemih z različnimi procesorji. RPC programi so torej prenosljivi in med seboj kompatibilni. Delujejo na višjem nivoju kot pa vtičnice (*Berkley sockets*) in pred uporabnikom skrivajo posebnosti mrežne povezave. Tako brez težav komunicirata programa, na katerem je na enem procesorju celoštevilčna spremenljivka `int` velika 16 bitov na drugem pa v velikosti 32 bitov. Poleg kompatibilnosti celih števil so tu še drugi tipi spremenljivk, kot so realna števila, nizi znakov, polja in strukture. Te podatke si programi izmenjujejo po enem od protokolov.

Da bi dosegli kompatibilnost je bilo potrebno uvesti enoten zapis za vse tipe spremenljivk, ki si jih programi izmenjujejo. Programi odjemalcev in strežnikov imajo sicer interni zapis spremenljivk v spominu zapisan tako, kot to ustreza procesorju in operacijskemu sistemu; ko pa komunicirajo z drugimi programi, pa morajo pretvoriti interni zapis v skupni dogovorjeni format znan pod imenom *eXternal Data Representation* (XDR). Odjemalci in strežniki imajo v svoji kodi poleg njim lastnim algoritmom še posebni del kode, ki skrbi za pretvorbo notranjega zapisa v dogovorjeni zunanji zapis. Na sliki C.1 je prikazan del način komunikacije dveh programov preko mrežnega protokola, kjer je videti, da imata programa vsak svoj povezovalnik osnovne kode z mrežo.

Povezovalnik (`stub`) skrbi za konverzijo internih spremenljivk v zunanji neodvisni

C.3 Procedure s povratnim klicanjem (Callback)

Včasih je uporabno, če strežnik začasno prevzame vlogo odjemalca in pokliče proces, ki je odjemalec. Primer potrebe po takem načinu komunikacije so dolgotrajne obdelave na strežniku ali pa aplikacije z večimi strežniki in enim odjemalcem, ki nadzoruje delo stežnikov in skrbi za pravilno razporeditev dela (*load balancing*). Povratno klicanje je najbolj zmogljiva asinhrona povezava strežnika in odjemalca. *Callback* aplikacija je hkrati strežnik in odjemalec. Klasičen pristop v datotečnih strežnikih, ki ustreza povratnemu klicanju, je omrežje *vsak z vsakim* (*pear-to-pear*).

C.4 Tipi strežnikov

Glede na čas obdelave strežnega servisa se ločijo tudi strežniki: *Iterativni strežnik* porabi le kratek čas za serviranje podatkov odjemalcem. Primer takega strežnika je lahko "*dan v tednu*". Strežniki, ki zahtevajo več časa in simultane dostope večih odjemalcev, ne morejo delovati kot iterativni strežniki, saj v času obdelave blokirajo dostop drugih odjemalcev. Zato taki stežniki naredijo podproces s funkcijo `fork` in prepustijo nadzor prvotnemu procesu, ki se vrne na čakanje novih zahtev. Taki strežniki se imenujejo *skupni strežniki* (*concurrent servers*).

V praksi obstajati tudi dve inačici skupnih strežnikov: tisti, ki naredijo več kot en podproces za izvedbo servisa, ter primeri z enim samim procesom. Kdaj je kakšen uporaben, je odvisno od narave servisa, ki ga strežnik rešuje. Največkrat je zaradi potrebe po integriteti datotek uporabljen le en glavni servis, v strežniku pa se uporabi komunikacija med procesi (IPC), ki postavlja v vrsto zahteve odjemalcev.

C.5 Nivoji rutin RPC

Knjižnica RPC omogoča tri nivoje programiranja v distribuiranem okolju. Med seboj se ločijo po nivoju abstrakcije programov, ki komunicirajo po omrežju:

Najvišji nivo je popolnoma neodvisen od operacijskega sistema, mreže in tipa računalnika na katerem program teče. Število funkcij na tem nivoju je majhno in ne omogoča pravega programiranja za izmenjavo podatkov med aplikacijami. Funkcije tudi niso na voljo na vse sistemih. Običajno so na voljo le naslednje funkcije, ki jih lahko uporabimo v C programih:

- **rnusers** Vrne število uporabnikov na oddaljenem računalniku
- **rusers** Vrne informacije o uporabnikih
- **havedisk** Testiranje prisotnosti trdega diska na oddaljenem računalniku
- **rstats** Informacije o trenutni izkoriščenosti oddaljenega računalnika

zlaganju celoštevilčne spremenljivke po Big-Endian principu, katerega tipična predstavnika sta Motorola in SPARC, ter Little Endian zlaganje bytov, ki ga najdemo v računalnikih PC ali VAX. S piko je označen bit za predznak. Ocenjevati, kateri način sestavljanja je boljši, tu ne bi imelo smisla, pomembno je le, da se zavedamo razlik.

Rutine knjižnice XDR torej pretvarjajo različne interne zapise v enotni format, ki se posreduje nižjemu sloju (slika 5.2). Knjižnica XDR dostopa do transportnega sloja s pomočjo rutin RPC, ki skrbijo še za ostale komunikacije z nižjimi sloji. Pomembno je poudariti, da se za transportni sloj pri XDR lahko uporabijo tudi drugi mediji, kot je na primer datoteka. Uporabnost dela knjižnice XDR je že v tem, da imamo na voljo rutine, ki jih lahko uporabimo tudi v namene, ki niso vezani na knjižnico RPC. Tako jih je možno uporabiti za generiranje in interpretiranje različnih grafičnih datotek.

Podprogrami se v knjižnici XDR delijo na štiri skupine:

- Kreiranje tokov (*streams*)
- Nadzor nad tokovi
- Konverzija enostavnih (vgrajenih) tipov spremenljivk
- Konverzija sestavljenih (kompleksnih) tipov

Rutine za konverzijo so dvosmerne, kar pomeni, da imamo en sam podprogram za kodiranje in dekodiranje podatkov istega tipa. Te rutine imajo podobno zgradbo, vsaka z dvema argumentoma od katerih je prvi struktura XDR, drugi pa je kazalec na spremenljivko ustreznega tipa, ki jo kodirano ali dekodiramo.

```
bool_t xdr_proc(xdrs, argresp)
       XDR *xdrs;
       <tip> *argresp;
```

Iz navedenega je opaziti, da so deklaracije funkcij v jeziku C zapisane po standardu K&R, ki je na sistemih UNIX še vedno najbolj razširjen. Obstajajo tudi prototipi ANSI, vendar njihova uporaba zmanjša prenosljivost programa napisanega za RPC.

Struktura oz. izpeljani tip XDR ima poleg ostalih internih spremenljivk, ki niso veliko uporabljene, še enumerirano spremenljivko `xdr_op` s katero se določa tip operacije pri klicanju procedure. Te operacije so:

```
enum xdr_op {
        XDR_ENCODE = 0,
        XDR_DECODE = 1,
        XDR_FREE = 2
};
```

Poleg samo razumljivih operacij za kodiranje in dekodiranje se pri sestavljenih (kompleksnih) tipih uporablja še operacija XDR_FREE, ki sprosti spomin pri brisanju teh tipov.

- osnova (stub) strežnika (app_svc.c)
- osnova odjemalca (app_clnt.c)
- skupni podprogrami za konverzijo podatkov (app_xdr.c)

Prevajalnik protokola ne naredi vseh rutin za aplikacijo, ampak le osnovo za komunikacijo. Seveda je možno napisati aplikacijo tudi brez uporabe prevajalnika protokola, vendar moramo pri tem skrbno paziti na kompatibilnost med strežnikom in odjemalcem. V praksi se je izkazalo, da je uporaba programa *rpcgen* skoraj nuja, saj se je kaj lahko zmotiti pri razvijanju programa; iskanje napak pa je pri takih aplikacijah zelo zahteven proces. Uporabnik tudi največkrat ne predvidi vse možne napake, ki se lahko pojavijo pri komunikaciji, *rpcgen* pa pri tvorjenju izvorne kode vgrajuje tudi testiranja o morebitnih napakah. Še pomembnejši razlog pa je, da je program z uporabo *rpcgen* krajši in lažje razumljiv.

Jezik prevajalnika je podoben jeziku C, poleg tega pa lahko vsebuje tudi bloke v jeziku C, ki je ne interpretira, ampak le prepiše na ustrezno mesto. Prevajalnik protokola je v prvi vrsti namenjen generiranju povezovalnih rutin, zato ima v jeziku največ možnosti prav za definiranje tipa podatkov, ki se pošiljajo med strežnikom in odjemalcem. Ker v knjižnici XDR ni bilo mehanizma za konverzijo sestavljenih tipov kot so strukture (struct, typedef) in unije, je eden od namenov *rpcgen* tudi ta, da omogoči pretvorbo struktur v osnovne tipe, ki jih poznamo v knjižnici XDR.

Iz dela, kjer so definirani podatki za transport, prevajalnik naredi izvorno kodo s filtri v posebni datoteki s končnico _xdr.c. Programska sekcija, ki je tudi sestavni del jezika, pa omogoča pravilno povezavo med programom strežnika in odjemalca. Tu opišemo procedure, ki jih bosta programa imela in podamo verzijo programa. Ena od ugodnosti aplikacij RPC je tudi ta, da na enem računalniku lahko teče več verzij istega programa. S tem si zagotovimo, da lahko starejši programi še vedno delujejo ob uvedbi novejših, praviloma testnih verzij strežnikov. To je tudi eden od rašlogov za priljubljenost protokola RPC, saj mnogokrat zanesljivost delovanja strežnika pomemben faktor pri delu distribuiranih aplikacij.

Primer vhodne datoteke prevajalnika *rpcgen* za izpis imenika na serverju je:

```
/*
 * dir.x: Izpis imenika na serverju
 */
const MAXNAMELEN = 255;  /* Najvecja dolzina imena direktorija */
typedef string nametype<MAXNAMELEN>;    /* imenik */
typedef struct namenode *namelist;      /* povezava v seznamu */
/*
 * Vozel v seznamu direktorija
 */
struct namenode {
        nametype name;          /* ime direktorija */
        namelist next;          /* naslednje ime */
};
/*
 * Razultati branja procedure READDIR
```

knjižnic, ki je omogočala razvoj distribuiranih programov in je osnova za večino kasnejših knjižnic, ki so bolj prirejene pisanju paralelnih programov.

V omrežjih obstajata dva načina komunikacije med procesi: sinhroni in asinhroni. Pri sinhroni komunikaciji je odjemalec blokiran ves čas, ko oddaljena procedura procesira želeno nalogo. Asinhrona komunikacija nima te omejitve, zato lahko odjemalec hkrati komunicira z več oddaljenimi procedurami in dela ostale koristne stvari. Najbolj splošen pristop pri klicanju oddaljenih procedur je model strežnik – odjemalec. Obstajata dva tipa strežnikov, ki se razlikujeta v hranjenju informacije o odjemalcih.

Rutine RPC se ločijo po stopnji abstrakcije komunikacije programov s prenosnim medijem. Za transparenten prenos podatkov med arhitekturami skrbi knjižnica XDR (*eXternal Data Representation*). Podatke je potrebno za prenos kodirati iz lokalnega formata v dogovorjeni format, pri sprejemu pa dekodirati nazaj. Za lažje pisanje programov v ONC/RPC obstaja prevajalnik protokola, ki generira kompatibilne programe odjemalca in strežnika.

Dodatek D

Komunikacija med procesi – IPC

Osnova za razumevanje skupnega izvajanja problema v večih procesih je nadzor in komunikacija med procesi *(Inter Process Communications)*. Delovanje IPC, ki se za razliko od distribuiranih programov odvija na enem računalniku, se konceptualno ne razlikuje od distribuiranega procesiranja. Uporaba IPC lahko zmanjša čas procesiranja pri nalogah, ki so časovno zahtevne, niso pa računsko potratne. Operacijski sistem Unix v veliki meri izkorišča vse načine komunikacij med procesi in s tem kar najmanj obremenjuje procesor (do 5 %), ki je bolj potreben pravim računskim nalogam. Poleg navedenih načinov komunikacije pa obstajajo na *Sistemu V* še dodatni načini komunikacij med procesi, ki drugače med seboj niso povezani:

Sporočila si lahko pošiljajo med seboj katerikoli procesi ali strežniki

Semaforji omogočajo sinhronizacijo nepovezanih procesov

Deljeni spomin lahko uporabijo nepovezani procesi in si na tak način izmenjujejo podatke

Zaščita pri dostopu do podatkov, ki si jih procesi med seboj izmenjujejo, je enaka kot pri datotečnem sistemu, kjer se nastavljajo `rwx` dostopi za sistem, skupino in lastnika. Pri izmenjavi podatkov med procesi lahko procesi dobijo podrobne podatke o tem, kateri proces je podatke kreiral, kdaj in kateri procesi lahko te podatke uporabijo.

D.1 Nadzor procesov

Najenostavnejša uporaba funkcije pri izvajanju drugih programov (procedur) je z ukazom kot je `system("date")`. Ta ukaz je podoben sinhronemu delovanju brez vračanja rezultatov. Funkcija `system()` vrača le status uspešnosti izvršitve. Po končanju funkcije se program normalno nadaljuje. Podobne so tudi funkcije `exec()`, ki v svojih variantah znajo iskati podani ukaz po podani poti (*path*) in za razliko od `system()`, nadomestijo originalni program z ukazom, ki je bil podan. Originalni program po zagonu ni več aktiven in je zato potrebno pri takih ukazih prej zapreti vse aktivne komunikacije (odprte

1. Ignoriranje signala je možno za vse signale razen `SIGKILL` in `SIGSTOP`, ki sta predvidena za brezpogojno končanje in zaustavljanje programa. Tako ukaz

   ```
   signal(SIGQUIT, SIG_IGN);
   ```

   ignorira ukaze za končanje *(quit)*

2. Z ukazom kot je

   ```
   signal(SIGQUIT, SIG_DFL);
   ```

   pa nastavimo predvideno *(default)* odzivanje na signal

3. Uporabniško odzivanje na signal je možno tako, da uporabnik poda funkcijo, ki se izvede ob prispetju ustreznega signala. Za kritične signale, ki jih uporabnik prestreza, se največkrat zahteva tudi končanje programa, s tem, da ima uporabnik tu še dodatno možnost, da reši čim več dela v programu, izpiše obvestilo o kritičnem signalu in podobne informacije. Poleg predpisanih tipov signalov, sta tu še dva signala, namenjena samo uporabniku in jih sistem ne generira sam ampak le uporabnik ali pripadajoči programi.

Pošiljanje signalov med procesi je izvedeno s funkcijama `kill()` in `raise()`. Funkcija `raise` lahko pošilja signale le istemu procesu, medtem ko `kill` zmore pošiljati tudi drugim procesom. Čakanje na signal je izvedeno s `pause()`. Možna je tudi nastavitev za avtomatsko generiranje signala s funkcijo `alarm(unsigned int` *`seconds`*`)`, kar uporabno za kontrolo delovanja programa s t.i. *watch-dog timer*-jem, kjer nastavimo največji čas, ki ga lahko kakšna operacija opravlja. Tipična opravila so to razna branja iz enot, kjer nastavimo t.i. *timeout*, po katerem prekinemo nadaljnje branje zaradi nedelovanja enote.

Na novejših sistemih obstajajo tudi funcije za nadzor skupine signalov, kjer lahko nastavimo čakanje, začasno ustavitev signalov in veriženje akcij (*hooks*) za signale.

D.3 Cevi

Kot osnova in najstarejši princip komunikacije so *cevi* (*pipes*), tudi najbolj uporabne v interaktivnem delu. Smer pretoka podatkov med dvema procesoma je le enosmeren *(half-duplex)*. Procesi, ki med seboj komunicirajo, morajo imeti istega predhodnika. To je običajno proces kreiran s funkcijo `fork()`. Pri interaktivnem delu večina operacijskih sistemov pozna tudi cevi, ki delujejo med programi tako, da ukazni interpreter poveže standardni izhod `stdout` in standardni vhod `stdin` dveh programov. V operacijskem sistemu DOS, pa za razliko od UNIX-a, programa, ki si na ta način izmenjujeta podatke, ne tečeta vzporedno. Programi kreirani s `fork()` ne delujejo na ta način, ampak je potrebno uporabiti funkcijo `pipe()`, ki uporablja opisnik *(descriptor)* ali princip navidezne datoteke ter v primerjavi z neprekinjenimi tokovi (*streams*) nič hitrejše, se ga za nove aplikacije ne priporoča.

D.5 Semaforji

Semaforji omogočajo komunikacijo med več procesi z izmenjavo vrednosti semaforjev. Uporabljajo se predvsem za zaklepanje in pravilno deljenje virov (*resource sharing*) med različnimi procesi.

Semaforji so pozitivne številčne vrednosti in omogočajo dostop do polja semaforjev s funkcijo `semop`. Če je vrednost, ki jo semafor vsebuje pozitivna, potem proces lahko uporabi vire (podatke). Ob uporabi se vredost semaforja zmanjša za ena. Če je vrednost semaforja enaka nič, se proces začasno ustavi (*sleep*), dokler ne dobi semafor vrednosti ena in s tem dovoljenje za uporabo.

Podobno kot pri pošiljanju sporočil je tudi tu omejeno število semaforjev. Pri uporabi je potrebno paziti na pravilno sproščanje virov (semaforjev) ob končanju programa v vseh možnih variantah. Te omejitve vnašajo težave pri izdelavi programa, zato se za enostavnejše programe raje uporablja zaklepanje z uporabo datotek (*advisory record locking*)[1], ki je enostavnejše, saj sistem skrbi za to, da se ob končanju procesa sprosti zaklenjena datoteka. Zaklepanje datotek je pogosta praksa, kjer se kreira datoteka minimalne velikosti in zaklene proti pisanju.

D.6 Deljeni spomin

Če si več procesov želi deliti isti kos spomina in ga uporabiti za globalne spremenljivke lahko uporabijo *deljeni spomin* (*shared memory*). Taka izmenjava podatkov je tudi najhitrejša od vseh komunikacij IPC, saj si procesi podatkov ne pošiljajo ampak jih le uporabljajo. Sistem skrbi, da je ta del spomina vedno na voljo procesom. Procesi komunicirajo na tak način, da z uporabo semaforjev ali zaklepanjem datoteke omogočijo pravilno sinhronizacijo pri dostopu do skupnega spomina, saj se ne sme zgoditi, da nek proces bere podatke, vmes pa drugi popravlja iste podatke.

S funkcijo `shmget` proces alocira spomin v sistemu željene velikosti, ki ni tako omejen kot pri prejšnjih načinih komunikacije. Maksimalne velikosti deljenega spomina so običajno nekaj megabytov, kar je dovolj za večino aplikacij. Po kreiranju deljenega spomina se procesi priklapljajo nanj s funkcijo `shmat` (*attach*) in odklapljajo s funkcijo `shmdt` (*detach*).

[1] Tak način zaklepanja ne dostopa direktno do datotečnega sistema in je zato hitrejše od klasičnega (ekskluzivnega) zaklepanja

read() ali write() za pošiljanje podatkov med računalnikoma. Tak način prenosa ustreza tokovnemu načunu prenosa (TCP), kjer se piše s funkcijo write() in bere z read().

- Prenos brez trajne povezave podobno kreira vtičnice. Strežnik namesto vzpostaviteve zveze z listen() kreira povezavo za sprejemanje podatkov z recvform(). Pošiljanje zahteve pa odjemalec sporoči s funkcijo sendto(). Taka povezava je paketnega tipa, kjer se pošiljajo t.i. *datagrami*.

Prenosi s paketi ali tokovi se razlikujejo tako v hitrosti, kot tudi v zanesljivosti povezave (prenosa). Paketni prenos je hiter, vendar v principu nezanesljiv; paketki ne pridejo nujno v pravem vrstnem redu, lahko so podvojeni ali se celo izgubijo. Prenos s tokom (stream) je zanesljiv in počasnejši od paketnega prenosa. Povezava je v trajanju zveze stalna, zato se lahko pri takem prenosu stalno kontrolira povezava tudi če se ne prenašajo podatki, kar jemlje sistemske vire operacijskemu sistemu.

D.8 Povzetek

Komunikacija med procesi je klasičen način izmenjave podatkov med procesi v okolju Unix, kjer je znana pod imenom *Interprocess Communications*. Razumevanje komunikacije, ki se običajno odvija na enem računalniku, je možno razširiti tudi na distribuirane sisteme. V tem poglavju so predstavljeni različni načini komunikacije kot so: pošiljanje sporočil, semaforji, deljeni spomin, signali in cevi. Vsak od omenjenih načinov komunikacije ima svoje prednosti in slabosti, zato je potrebno poznavanje vseh pristopov za ustrezen pristop pri razvoju aplikacij.

Življenjepis

Leon Kos se je rodil 9. junija 1966 v Postojni. Osnovno šolo je končal v Cerknici. Po osnovni šoli je obiskoval Srednjo tehnično in naravoslovno šolo v Postojni, program strojni tehnik. Leta 1987 je pričel s študijem na Fakulteti za strojništvo v Ljubljani in leta 1991 diplomiral s področja računalniške grafike pri prof.dr. Jožetu Duhovniku, dipl.ing.

Leta 1992 je vpisal podiplomski študij na Fakulteti za strojništvo. V okviru podiplomskega študija se v laboratoriju za računalniško podprto konstruiranje ukvarja z upodabljanjem tridimenzionalnih objektov z računalniki.

Na področju izobraževanja sodeluje na dodiplomskem študiju in v izobraževanju strokovnjakov iz industrije na področju računalniško podprtega konstruiranja.

Magistrsko delo predstavlja rezultate lastnega znanstveno raziskovalnega dela na osnovi sodelovanja z mentorjem izr.prof.dr.Jožetom Duhovnikom, dipl.inž.

Leon Kos

Magisterij je bil stavljen s programom LaTeX 2ε[2]. Slike so bile narisane s programoma xfig in gnuplot. Na zahtevo je delo mogoče dobiti pri avtorju tudi v postscript ali hipertekstni obliki.[3]

Avtorjev naslov: Leon Kos
Cesta 4. maja 11a
61360 Cerknica
e-mail: `leon.kos@uni-lj.si`

[2] LaTeX 2ε je nadgradnja LaTeX-a. LaTeX je skupina makrojev za TeX. TeX je blagovna znamka American Mathematical Society.

[3] HTML – HyperText Markup Language je opisni jezik za katerega obstajajo bralniki za različne operacijske sisteme.