## 1.1 Namen dela

Eden od osnovnih izzivov v računalništvu je tudi razvoj algoritmov, programskih orodij in aplikacij, ki bi izkoristile razpršeno računalniško moč z uporabo distribuiranih sistemov na lokalnih omrežjih. Z razvojem računalniških omrežij, kot tudi paralelno strojno opremo, se bo povečal tudi komercialni interes za nove tehnologije, ki bodo ceneje in bolje izkoriščale obstoječe in nove vire računalniške moči.

Računalniška omrežja so seveda le potreben pogoj za njihovo izkoriščanje. Zaradi različnih standardov na najnižjem nivoju prenosa informacije se pojavlja tudi problem povezovanja različnih arhitektur računalnikov, ki pogosto pogojujejo tudi operacijski sistem in s tem način komunikacije s sorodnimi računalniki. Tako imamo v praksi opravka z različnimi protokoli prenosa, ki so posledica razvoja in/ali licenčnih klavzul proizvajalcev omrežne programske opreme. Še posebej je to očitno na področju osebnih računalnikov, kjer imamo široko paleto protokolov kot so IPX, SMB, AppleTalk, Netbios, TCP/IP, ... z različnimi topologijami (ARCnet, Ethernet, Token ring, ...) in različnimi tipi paketov (OSI 802.2, Ethernet II, Ethernet 802.3, Token ring, Novell RX-NET, IBM PC2N2, ...). Vsa našteta raznolikost programske in strojne opreme povzroča težave pri povezovanju računalnikov v distribuirani sistem.

Problem povezovanja je največkrat na nivoju programske opreme, kjer imamo različne specifikacije proizvajalcev za način klicanja omrežnih servisov. Tu sicer obstajajo standardi OSI[1], ki le razvrščajo omrežne modele na sedem nivojev povezovanja. Podrobnejši podatki o izvedbi posameznih slojev pa so zbrani v neformalnih *de-facto* standardih zbranih v dokumentih RFC [2]. Na področju osebnih računalnikov z operacijskim sistemom DOS sta v veljavi še standarda na nivoju protokola NDIS (Microsoft) in ODI (Novell), od katerih je slednji splošno dostopen in ni licenčno pogojen. V svetu delovnih postaj so razmere veliko bolj jasne, saj tu prevladuje skupina Internet protokolov kot so TCP, (C)SLIP, PPP.

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.

Temeljni namen pričujočega dela je pregled obstoječega stanja na področju distribuiranih sistemov in njihova uporaba v računalniški grafiki. Na konkretni aplikaciji se je želelo prikazati primernost, uporabnost in zahtevnost implementacije v izbranem področju – upodabljanju. Pomembna želja je bila tudi v pridobitvi novega znanja za nadaljnje raziskave z obeh področij, ki jih magistrska naloga obsega.

[1] Open Systems Interconnection
[2] Request For Comments

Poglavje 6 predstavi knjižnico za razvoj paralelnih aplikacij, ki je bila tudi praktično uporabljena v sistemu za upodabljanje z metodo sledenja žarku. Zbirka rutin PVM (*Parallel Virtual Machine*) je osnova, na kateri lahko enostavno gradimo paralelne programe. PVM je sistem, ki omogoča heterogeni skupini Unix računalnikov, povezani med seboj z mrežo, da so za uporabnika vidni kot en sam paralelni računalnik. Sistem PVM deluje tudi na računalnikih z večjim številom tesno povezanih procesorjev, kjer so možni tudi drugi načini komunikacije predstavljeni v dodatku D. V tem poglavju so predstavljena tudi grafična orodja, s katerimi lahko analiziramo delovanje naših programov, kot tudi osnovni napotki pri gradnji distribuiranega sistema. Diskutirane so težave pri razvoju distribuiranih sistemov. V tem kontekstu je predstavljeno tudi integrirano razvojno okolje, ki lajša delo v najbolj zamudnih delih razvoja in testiranja programov.

Praktična izvedba distribuiranega sistema je predstavljena v poglavju 7. Diskutirani so pomembni deli implementacije s sistemom PVM in prilagoditev programa za sledenje žarku, ki je bil dostopen na Internetu. Diskutirana je tudi izdelava enostavnega objektno orientiranega programa z metodo sledenja žarku. Zaradi njegove objektne orientiranosti je možno enostavno dodajati nove geometrijske objekte in modele osvetlitve. Testi zmogljivosti so tudi eden od namenov magistrske naloge, ki je prikazal možnost uporabe obstoječe računalniške opreme v namene, za katere se običajno uporabljajo visoko zmogljivi[3] računalniki.

Končno, poglavje 8 strni delo in našteje množico odprtih vprašanj.

V dodatku so predstavljena poglavja, ki so pomembna za razumevanje in podrobneje opisujejo postopke, uporabljene v magistrski nalogi. Dodatek A tako opisuje objektno–orientirani program sledenja žarku s pomembnejšimi podatkovnimi strukturami (*razredi*) in metodami, ki operirajo s podatki.

Dodatek B podrobneje opisuje algoritem distribuiranega programa za sledenje žarku. Prikazani so bistveni deli programa in način komunikacije med nadzornikom in delavci.

V dodatku C je predstavljen sistem za klicanje oddaljenih procedur – RPC. Knjižnica RPC skupaj z XDR predstavja zaključen sistem (ONC) za gradnjo distribuiranih sistemov v omrežjih s protokolom IP. Je ena prvih sistemov s katerim je bilo možno graditi distribuirane sisteme. Zaradi odprtosti (izvorna koda je dostopna na Internetu) je tudi najbolj razširjena in praktično uporabljana v distribuiranih datotečnih sistemih kot je NFS in ostalih servisih. Razumevanje delovanja, ki je prikazano v tem poglavju je bistveno za pravilno planiranje komuniciranja med računalniki. Velika večina naprednejših sistemov za distribuirano procesiranje temelji prav na knjižnici RPC.

V dodatku D so diskutirani različni pristopi, ki jih lahko najdemo v večini modernih operacijski sistemov. Prikazani so različni načini komunikacije med procesi (*Inter Process Communication – IPC*), njihova sinhronizacija in nadzor pretoka podatkov. Podobni prijemi komunikacij se uporabljajo tudi v posebnih računalnikih z večjim številom procesorjev, ki so tesno povezani med seboj.

[3] beri dragi

opisu. Slike z izrazitim efektom polarizacije, zaradi redke pojavnosti v naravi izgledajo zelo umetne.

V računalniški grafiki se običajno uporabi geometrijski model zaradi relativne nepomembnosti interference, polarizacije in uklona svetlobe. Če je v geometrijskem modelu uporabljena tudi kakšna od valovnih lastnosti, se to obravnava kot posebni primer.

Ena od pomembnih lastnosti svetlobe je tudi valovna dolžina, saj le ta predpisuje barvo. Vključitev v geometrijski model ni težavna, je pa potrebno za oblikovanje realistične slike. Prav razumevanje barve kot lastnosti površine je tudi ključ za tvorbo realističnih slik.

## 2.2 Fizikalni model površine

Razumevanje vplivnih dejavnikov pri interakciji s svetlobo vpliva tudi na izvedbo matematičnega modela površine. Izkustveno lahko ugotovimo, da lastnosti materiala in obdelava površine bistveno vplivajo na spekter svetlobe, ki se odbije ali prehaja skozi površino. Hrapavost, kot eden osnovnih parametrov, vpliva tudi na porazdelitev odbite svetlobe.

Glede na material in obdelavo površine ločimo tudi štiri glavne modele interakcije svetlobe in površine: *zrcalni in razpršeni (difuzni) odboj, zrcalni in razpršeni prehod svetlobe*.

## 2.3 Gladke površine

Za gladke površine čistih materialov lahko matematično dokaj dobro opišemo odboj svetlobe. *Fresnelove enačbe* [BW89, Wat90] predpisujejo, kako se del svetlobe odbije ali prehaja (lomi) skozi površino. Normalno jih zapišemo na stiku dveh dieletrikov, ki imata vsak svojim lomni količnik. Lomni količnik je razmerje med hitrostjo širjenja elektromagnetnega valovanja v vakuumu in hitrostjo širjenja v snovi. Tipične vrednosti lomnih količnikov [Gla91c, Sea56] so podane v tabeli 2.1

| Snov | Lomni količnik $\eta$ |
|---|---|
| Voda | 1.33 |
| Zrak | 1.63 |
| Okensko steklo | 1.52 |
| Gosto steklo | 1.72 |
| Diamant | 2.419 |
| Sol | 1.50 |

Tabela 2.1: Lomni količnik za različne dielektrike

Izkaže se, da se svetloba različnih valovnih dolžin lomi različno. To lahko opazimo na prizmi, skozi katero pošiljamo belo svetlobo.

Pri prehodu svetlobe v snov z manjšim lomnim količnikom, se lahko zgodi, da se enačbe (2.3) ne more izpolniti. Kot pri katerem se celotna svetloba začne odbijati, se imenuje *kritični odbojni kot* ali *kot popolnega odboja*.

Razmerja deležev svetlobe, ki se odbijejo in lomijo skozi površino, se lahko izračunajo s Fresnelovimi enačbami.

### 2.3.2 Fresnelove enačbe

Fresnelove enačbe temeljijo na fizikalnem optičnem modelu, saj obravnavajo polarizacijo svetlobe na gladki površini. Prevodniki ali kovine so opisani z lomnim količnikom, sposobnostjo odboja (*reflektanca*) in indeksom izgub $\kappa$, ki je merilo za zmanjšanje intenzitete svetlobe na enoto dolžine. Indeks izgub je za dielektrike po definiciji enak nič.

Svetlobo, ki pod določenim kotom $\theta$ pada na površino, lahko razdelimo na dve med seboj pravokotni ravnini valovanja električnega in magnetnega polja. Naravna nepolarizirana svetloba je zmes polariziranih valovanj v vseh smereh. Vsako valovanje pa lahko razdelimo na valovanje v dve med seboj pravokotni komponenti $R_s$ in $R_p$. Odbojnost nepolarizirane svetlobe je povprečje $R_s$ in $R_p$.

$$R = \frac{R_s + R_p}{2} = \frac{1}{2}\left[\frac{tan^2(\theta_i - \theta)}{tan^2(\theta_i + \theta)} + \frac{sin^2(\theta_i - \theta)}{sin^2(\theta_i + \theta)}\right] \tag{2.6}$$

Če uporabimo enačbo (2.2) in (2.6), ter indeks izgub $\kappa$, lahko zapišemo enačbi odboja za polarizacijski ravnini na stiku med snovjo vpadnega žarka z lomnim količnikom $\eta_i$ in snovjo z lomnim količnikom $\eta$, kot

$$R_s = \frac{a^2 + b^2 - 2a\cos\theta + \cos^2\theta}{a^2 + b^2 + 2a\cos\theta + \cos^2\theta} \tag{2.7}$$

$$R_p = Rs\frac{a^2 + b^2 - 2a\sin\theta\tan\theta + \sin^2\theta\tan^2\theta}{a^2 + b^2 + 2a\sin\theta\tan\theta + \sin^2\theta\tan^2\theta} \tag{2.8}$$

kjer je $\theta$ vpadni kot glede na normalo; $a$ in $b$ pa sta dana kot

$$a^2 = \frac{1}{2\eta_i^2}\left[\sqrt{(\eta^2 - \kappa^2 - \eta_i^2\sin^2\theta)^2 + 4\eta^2\kappa^2} + \eta^2 - \kappa^2 - \eta_i^2\sin^2\theta\right] \tag{2.9}$$

$$a^2 = \frac{1}{2\eta_i^2}\left[\sqrt{(\eta^2 - \kappa^2 - \eta_i^2\sin^2\theta)^2 + 4\eta^2\kappa^2} - (\eta^2 - \kappa^2 - \eta_i^2\sin^2\theta)\right] \tag{2.10}$$

Za obravnavanje razmer pri stiku površin, kjer sta obe snovi dielektrika, lahko postavimo $\kappa$ na nič in $\eta$ uporabimo kot lomni količnik za drugi dielektrik. Pomembna razlika v odbojnosti pri dveh dielektrikih je razmerje med lomnim količnikom vstopne in izstopne snovi, kar lahko vidimo na sliki 2.2 in 2.3.

Iz navedenih slik lahko spoznamo dve pomembni lastnosti Fresnelovih enačb. Ko gre kot $\theta$ proti $90^\circ$ (takrat je vpadni žarek vzporeden stični ravnini), je odbojnost za vse snovi enaka ena.

Druga pomembna lastnost enačb je, da veljajo za vse vpadne kote $\theta$. Za posebni primer, kjer je vpadni kot nič (žarek pravokotno na površino) in kjer je prva snov zrak ($\eta_i \approx 1$), se lahko enačba (2.6) z uporabo Snellovega zakona (2.2), krajša na obliko

$$R = \frac{(\eta - 1)^2}{(\eta + 1)^2} \tag{2.11}$$

Iz tega tudi sledi, da je odbojnost za steklo, ki ima tipično vrednost lomnega količnika $\eta = 1.5$, enaka

$$R(\eta = 1.5) = \frac{(1.5 - 1)^2}{(1.5 + 1)^2} = 0.04 \tag{2.12}$$

Razmerje $R = I_r/I_0$ med odbito svetlobo $I_r$ in vpadno svetlobo $I_0$ je torej pri steklu, pri pravokotnem gledanju na površino, približno $4\%$. Zato se pri gladkih površinah normalno tudi vidi odblesk, ker vedno obstaja del svetlobe, ki se zrcalno odbije.

Iz slike 2.3 je pri prehodu iz snovi z višjim lomnim količnikom v snov z nižjim (zrak) opaziti posebno lastnost, kjer se pri določenemu kotu vpadni žarek prične popolnoma odbijati. Žarek se pri tem giblje v notranjosti dielektrika ne da bi utrpel večje izgube. Ta fenomen se imenuje *popolni notranji odboj*. Ker lahko svetloba pri *popolnem notranjem odboju* potuje znotraj prozornega dielektrika skoraj brez izgub, je ta lastnost našla uporabo pri prenosu informacij v optičnih kablih.

Lomni količnik $\eta$ je za dielektrike odvisen tudi od valovne dolžine. To je opaziti na prizmi, ki razprši belo svetlobo v spekter barv. Pri kovinah pa je poleg lomnega količnika, od valovne dolžine odvisen tudi indeks izgub $\kappa$. V prevodnikih je potrebno obravnavati lomni količnik kot kompleksno veličino, ki je definirana kot

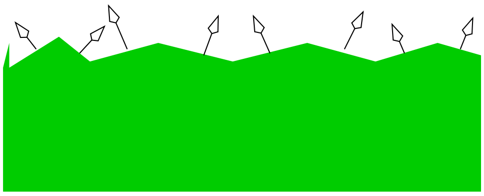
$$\bar{\eta}(\lambda) = \eta(\lambda) - i\kappa(\lambda) \tag{2.13}$$

Podrobnejše podatke o konstantah za večino materialov lahko najdemo v [Pal85]. Koeficienti se za geometrijski model določajo eksperimentalno tako, da se obravnava posamezne primarne barve (rdeča, zelena, modra) posamično. Iz tega sledi, da imamo ločene tudi lomne količnike $\eta_r$, $\eta_g$, $\eta_b$. Iz enačbe (2.11) lahko izrazimo lomni količnik kot

$$\eta = \frac{1 + \sqrt{R_0}}{1 - \sqrt{R_0}} \tag{2.14}$$

kar nam omogoča, da lahko dokaj enostavno izmerimo lomne količnike za posamezne primarne barve. Za glajen baker so koeficienti odboja $R$ in lomni količniki $\eta$ primarnih barv podani v tabeli 2.2 (glej A. Watt [Wat90, strani 74-78]).

Potek jakosti odboja posameznih primarnih barv za glajen baker in $\kappa = 0.3$ je podan na sliki 2.4.

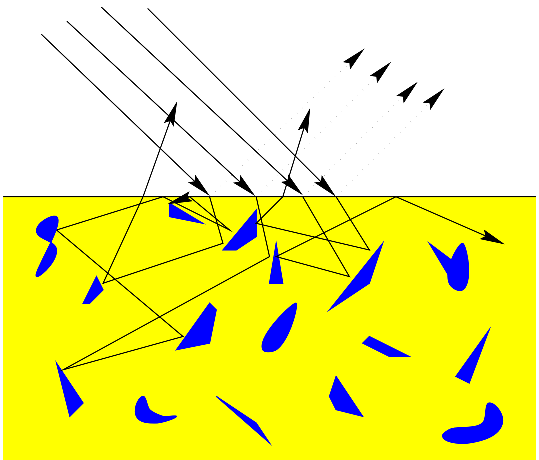


Slika 2.5: Podroben pogled na hrapavo površino z normalami mikropovršin

prepuščeno svetlobo tudi pri najmanjših debelinah.

## 2.5 Kompozitne površine

Sestavljene površine se pogosto pojavljajo v praksi kot sestav (kompozit) dveh ali več materialov. Običajno je to osnovni material, ki ima primešane majhne delce. Osnovna snov je lahko prozorna, kar pri osvetlitvi z žarkom pomeni, da se del svetlobe lahko lomi v površino in nato dodatno odbija, lomi in absorbira v primešanih delcih.



Slika 2.6: Optične razmere na sijajni površini kompozitne snovi

Klasičen primer sestavljene površine je barva s primešanimi delčki s srebrnim pigmentom (slika 2.6). Ker imajo primešani delci različen lomni količnik od lomnega količnika osnovne snovi, se tudi svetloba zelo razprši. Z večjim lomnim količnikom $\eta$ osnovne snovi se manjša kot popolnega notranjega odboja, kar onemogoča svetlobi, da bi izstopila iz površine potem ko je enkrat že vstopila vanjo. Z večanjem lomnega količnika se tako veča tudi motnost. Podoben primer je tudi kocka ledu z mehurčki kot delci, z drugačnim lomnim količnikom. Tudi tu je z večanjem števila mehurčkov motnost površine večja.

Primešani delci imajo lahko tudi barvo, kar vpliva na absorbcijo svetlobe točno določene valovne dolžine. Taka površina je pri osvetlitvi z belo svetlobo, vidna kot obarvana bitega in vpadnega žarka. S Fresnelovimi enačbami lahko izračunamo odbojnost za dielektrike in kovine, če poznamo lomni količnik in indeks izgub.

Površine, ki niso gladke, obravnavamo na mikroskopskem nivoju. Razumevanje razmer na takih površinah se lahko pridobi z opazovanjem strukture medija in površine. Glajene površine lahko izkazujejo delno zrcaljenje, ki pa ni nujno popolno, ampak je lahko tudi razpršeno. Valovite gladke površine ukrivljajo svetlobo in s tem pačijo zrcalne podobe na površini.

Snovi, ki vsebujejo delce drugih medijev z različnimi optičnimi lastnostmi, imenujemo *kompozitni mediji*. Delci razpršijo svetlobo v odvisnosti od lomnega količnika osnovnega medija. Velikost delcev v snovi vpliva tudi na obliko razpršitve in je v direktni povezavi z valovno dolžino svetlobe.

Kompozitne površine izražajo še lastnosti, kot sta sijajnost in motnost. Materiali, kot je papir, imajo lastnost prosojnosti.

Iz širokega področja optike so v tem poglavju predstavljene le lastnosti površin, ki so pomembne pri modeliranju v računalniški grafiki. Obširnejše obravnavanje optike najdemo v [BW89, Gue90, PP93, Kla89, Sea56].

pri katerih je možno zapisati linearno odvisnost intenzitete svetlobe in sevane energije. Belo svetlobo je zato možno obravnavati komponentno ali celo monokromatsko[FvD90]. V tem primeru je intenziteta svetlobe $I$ kar enaka lastni svetilnosti $k_i$

$$I = k_i \tag{3.1}$$

Namesto samoosvetljenosti objekta lahko predpostavimo, da v prostoru obstaja globalna osvetlitev, ki ni usmerjena in je zaradi tega osvetljenost objektov z vseh strani enaka. Pri motnih površinah lahko zapišemo podoben model osvetlitve, kjer se podaja intenziteta osvetlitve ambienta $I_a$ in koeficient motnosti $k_a$

$$I = k_a I_a \tag{3.2}$$

Koeficient difuznega odboja površine $k_a$ ima vrednosti med nič in ena. Prednost modela (3.2) pred (3.1) je tudi v tem, da lahko globalno spreminjamo osvetlitev prostora. Poudariti pa je treba, da tako enostavni modeli pri projekciji prikažejo le silhuete objektov, kar je uporabno le za preizkus širjenja primarnih žarkov v metodi sledenja.

## 3.1 Osnovni model odboja - Phongov model

Enostavni modeli odboja poskušajo minimizirati računsko zahtevnost in podrobno obravnavanje interakcije svetlobe s površino. Standardni model, ki je kompromis med uporabnimi rezultati in potrebnim računanjem, je *Phongov model odboja*.

Kaj se dogaja z žarkom, ko zadane površino trdnega telesa? Iz opazovanja, kot tudi iz zakonov optike, lahko ugotovimo štiri načine prenosa svetlobe pri interakciji žarka s površino:

- odboj svetlobe (zrcalni in difuzni)
- sipanje svetlobe v trdnem telesu
- absorbcija svetlobe v trdnem telesu
- prehod svetlobe

Razmere pri osvetlitvi so prikazane na sliki 3.1. Svetloba lahko prihaja na površino iz točkastega izvora, objektnega izvora ali globalna osvetlitev prostora.

Zaradi geometrijske obravnave svetlobe lahko posamezene prispevke obravnavamo ločeno in jih algebraično seštevamo. Za intenziteto odboja svetlobe na površini lahko v splošnem zapišemo dvosmerno odbojnostno funkcijo $R_{bd}(\theta_{in}, \phi_{in}, \theta_{out}, \phi_{out})$ (*Bidirectional Reflectance Distribution Function*), ki je odvisna od kotov $\theta$ in $\phi$ v krogelnih koordinatah.

Lastnost materiala podajamo s konstanto $k_d$; Intenziteta svetlobe v smeri $\vec{L}$ je določena s konstanto $I_i$. Za pravilen izračun kosinusa morata biti vektorja $\vec{N}$ in $\vec{L}$ enotska.

$$\vec{L} = -\frac{\vec{I}}{|\vec{I}|} \tag{3.5}$$

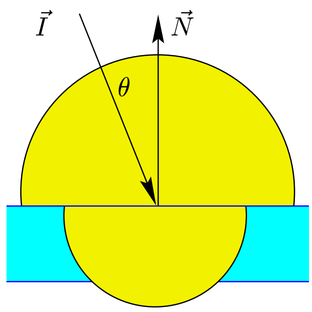
Prispevke več vpadnih žarkov algebraično seštejemo:

$$I_d = k_d \sum_n I_{i,n} (\vec{L}_n \cdot \vec{N}) \tag{3.6}$$

### 3.1.2 Difuzni prehod

Nekateri materiali, kot je papir, difuzno odbijajo svetlobo, pri tem pa nekaj svetlobe prepustijo skozi površino. Svetloba, ki se prepusti ne potuje nemoteno, ampak se razprši v materialu.

Prosojne materiale lahko modeliramo tako, da zgornji del površine ustreza difuznemu odboju, ostanek svetlobe se absorbira in emitira na drugi strani površine kot difuzna svetloba. Razmere na prosojnem materialu prikazuje slika 3.3.




Slika 3.3: Difuzni prehod svetlobe pri prosojnih površinah

Podobno kot v enačbi difuznega odboja (3.4), tudi tu ni potrebno uporabiti posebnih geometrijskih značilnosti površine, zato lahko zapišemo enačbo prehoda difuzne svetlobe skozi prosojne materiale kot

$$I_{dt} = I_i k_{dt} (\vec{L} \cdot \vec{N}) \tag{3.7}$$

Zmožnost difuzije svetlobe materiala je določena s konstanto $k_{dt}$, ki je običajno manjša od $k_d$.

### 3.1.3 Zrcalni odboj

Zrcalna površina se zelo pogosto pojavlja v računalniški grafiki, saj učinkovito prikazuje zmožnosti upodabljanja z računalnikom. Večina površin ima tudi del zrcalnih lastnosti.

S tem določimo vektor odboja $R$ iz enačbe (3.9) kot

$$\vec{R} = \vec{I} - 2(\vec{N} \cdot \vec{I})\vec{N} \tag{3.13}$$

ali če uporabimo vektor luči $\vec{L}$ iz enačbe (3.5)

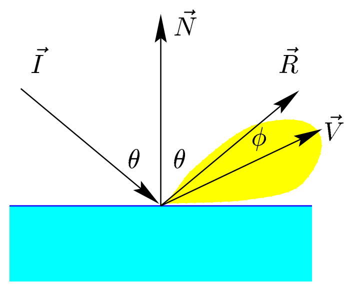
$$\vec{R} = 2(\vec{N} \cdot \vec{L})\vec{N} - \vec{L} \tag{3.14}$$

Ker je bilo že v enačbi (3.11) predpostavljeno, da so vsi vektorji enotski, se lahko pokaže, da je tudi $\vec{R}$ enotski [Gla91c] ob upoštevanju lastnosti (3.12) in $\alpha = 1$.

V praksi je vztrajanje na popolnem odboju svetlobe preveč stroga in nerealna zahteva, saj obstajajo gladke površine, ki jih ne moremo modelirati kot kombinacijo difuznega in zrcalnega odboja. To bi pomenilo, da bi bili točkasti izvori svetlobe na takih površinah vidni le v eni točki. Iz izkušenj pa vemo, da so točkasti izvori svetlobe vidni v določeni bližini kota zrcalnega odboja in da krožni odbleski na površini zvezno pojemajo. Za pravilno porazdelitev lahko uporabimo ustrezno empirično porazdelitveno funkcijo. Bui-Toung Phong [Pho75] je predlagal porazdelitev s funkcijo $\cos^n \phi$. Razlog izbora take porazdelitvene funkcije je tudi v enostavnosti izračuna, saj lahko znova uporabimo skalarni produkt med vektorjem odboja $\vec{R}$ in vektorjem gledanja $\vec{L}$. Intenziteta porazdeljenega zrcalnega odboja je modelirana kot

$$I_{rd} = k_s \cos^n \phi = k_s(\vec{R} \cdot \vec{V})^n \tag{3.15}$$

kar je grafično predstavljeno na sliki 3.5.




Slika 3.5: Izračun porazdelitve zrcalnega odboja

Z rastjo potence $n$ se manjša polje zrcalnega odboja. Če gre $n$ proti neskončnosti potem se enačba (3.15) približuje zrcalnemu odboju. Intenziteto odboja v smeri gledanja $\vec{V}$ določamo s konstanto $k_s$. Grafična predstavitev za intenziteto difuzno in zrcalno odbite svetlobe je podana za nekatere potence $n$ na sliki 3.6. Videti je, da so uporabne vrednosti potence pri $n = 10$ ali več.

Enačba (3.15) kaže tudi to, da je pri kotu gledanja, ki je enak odbitemu kotu, funkcija intenzitete maksimalna. Če uporabimo polovični vektor

$$\vec{H} = \frac{\vec{L} + \vec{V}}{|\vec{L} + \vec{V}|} \tag{3.16}$$

se lahko temu enostavno izognemo, vendar je potrebno ob uvedbi novega svetlobnega izvora ponovno nastaviti vse konstante, kar zna biti zamudno opravilo. Rezanje intenzitet na maksimalni nivo vsekakor ni dobra rešitev. Možno je tudi uporabiti intenzitete z večjim razponom vrednosti, ki jih kasneje normiramo tako, da poiščemo maksimalno vrednost vseh točk in vse točke delimo s to vrednostjo. Druga uporabna metoda je tudi ta, da vnaprej predvidimo možne maksimalne vrednosti intenzitete že kar iz konstant materialov in intenzitet svetlobnih izvorov. Ker ta metoda temelji na konzervativni oceni osvetlitve, se lahko zgodi, da zgubimo dinamični razpon vrednosti osvetlitve na celotni sliki.

### 3.1.5 Barva

Za barvne površine je pri modeliranju odbleskov najlažje uporabiti belo svetlobo. To pomeni tudi, da imamo izvore bele svetlobe, barvo objektov pa nastavljamo z izborom ustreznih konstant. Klasični barvni model RGB je ortogonalen in s tem tudi primeren za modeliranje, saj je potrebno izračunavati tri intenzitete:

$$
\begin{aligned}
I_r &= k_{ar} I_a + \sum_n I_{in} \left[ k_{dr} (\vec{L}_n \cdot \vec{N}) + k_s (\vec{N} \cdot \vec{H}_n)^n \right] \\
I_g &= k_{ag} I_a + \sum_n I_{in} \left[ k_{dg} (\vec{L}_n \cdot \vec{N}) + k_s (\vec{N} \cdot \vec{H}_n)^n \right] \\
I_b &= k_{ab} I_a + \sum_n I_{in} \left[ k_{db} (\vec{L}_n \cdot \vec{N}) + k_s (\vec{N} \cdot \vec{H}_n)^n \right]
\end{aligned}
\tag{3.20}
$$

Tak princip reševanja barv je groba aproksimacija saj je za odblesk uporabljena skupna konstanta $k_s$ in bela vstopna svetloba $I_i$. Predpostavljene so gladke površine, ki bistveno ne spreminjajo odbite svetlobe. Če je površina groba ali kompozitna, se kaj lahko zgodi, da se žarek večkrat odbije in s tem pridobi barvo materiala. Sledeči model obravnava prav te pojave na površini.

## 3.2 Cook-Torrance model

Vpliv materiala na barvo odbite svetlobe lahko modeliramo z upoštevanjem hrapavosti. Hrapavost lahko empirično modeliramo glede na meritve različnih materialov, lahko pa tako površino predpostavimo kot površino sestavljeno iz množice majhnih popolnih reflektorjev – *mikropovršin*. Menjava barve odbite svetlobe na takih površinah temelji na Fresnelovih zakonih (2.6). Z primera grobe površine na sliki 3.7 je pod (a) razvidno, da se lahko svetloba večkrat odbije na površini. Pri vsakem odboju na površini žarek (a) dobi del barve materiala. Material pri skoraj pravokotnih vpadnih kotih žarka torej bistveno vpliva na barvo svetlobe, ki se na koncu odbije. Pri položnejših kotih se lahko svetloba povsem absorbira (c), ali pa se odbije (b). Odboj svetlobe pri položnejših kotih ni pod večjim vplivom materiala. Svetloba je sicer delno izgubljena, vendar skoraj nima primesi barve materiala. Tako razmišljanje kaže, da Phongov model, kjer so svetlobni izvori in zrcalni odboji bele barve, za hrapave površine ni ustrezen. Phongov model izkazuje lastnosti plastičnih materialov, polirane kovine pa z njim ni možno pravilno modelirati.

zrcalne komponente v smeri gledanja vektorja $\vec{V}$ je

$$I_{sr}(\lambda) = \frac{F(\lambda,\theta)}{\pi} \frac{D(\alpha)\, G(\vec{N},\vec{V},\vec{L})}{(\vec{N}\cdot\vec{L})(\vec{N}\cdot\vec{V})} \tag{3.21}$$

kjer je

| | |
|---|---|
| $I_{sr}(\lambda)$ | intenziteta svetlobe v odvisnosti od valovne dolžine (barva) |
| $F(\lambda,\theta)$ | Fresnelova funkcija odboja, ki določa barvo površine |
| $G(\vec{N},\vec{V},\vec{L})$ | geometrijski faktor slabljenja zaradi orientacije mikro površin |
| $D(\alpha)$ | podaja verjetnost orientacije mikropovršin v smeri vektorja $\vec{H}$ |
| $\vec{N}$ | normala površine |
| $\vec{L}$ | vektor luči |
| $\vec{V}$ | vektor smeri gledanja |

V enačbi (3.21) imamo tri izraze, ki skupaj vplivajo na intenziteto svetlobe $I_{sr}$. Za difuzni odboj svetlobe, čeprav minimalen, lahko predpostavimo, da je konstanten.

### 3.2.1 Porazdelitvena funkcija orientacije mikropovršin

Zaradi različne orientacije mikropovršin se tudi svetloba odbija v različne smeri. Intenziteta svetlobe v določeni smeri je odvisna od števila mikropovršin orientiranih tako, da je zrcalni odboj največji. Podobno kot v Phongovem modelu lahko tudi tu uporabimo vektor $\vec{H}$ (3.16) kot merilo odboja v določeni smeri. Izraz $D(\alpha)$ je porazdelitvena funkcija intenzitete zrcalnega odboja v odvisnosti od kota $\alpha$, ki je kot med vektorjema $\vec{L}$ in $\vec{H}$. Za porazdelitveno funkcijo bi lahko uporabili različne funkcije, vendar se izkaže, da Gaussova porazdelitvena funkcija dovolj dobro opisuje realne razmere.

Širino porazdelitvene funkcije kontroliramo s konstanto $m$, ki predstavlja povprečni (*RMS*) nagib mikropovršin. S spreminjanjem $m$ tako kontroliramo hrapavost površine. Z malimi vrednostmi $(0.0 - 0.3)$ predstavljamo gladke, z večjimi $(0.3 - 0.8)$ pa hrapave površine. Uporabna formula, za izračun $D$ je

$$D(m,c,\alpha) = ce^{-(\alpha/m)^2} \tag{3.22}$$

kjer je c poljubna konstanta
e Eulerjeva konstanta.

Za bolj realen opis in s tem tudi računsko zahtevnejši opis lahko uporabimo Beckmannovo

porazdelitveno funkcijo:

$$D(m,\alpha) = \frac{1}{m^2 \cos^4\alpha} e^{-(\tan\alpha/m)^2} \tag{3.23}$$

žarek ne posveti na mikropovršino, ker je le ta v senci in žarek se popolnoma odbije. Med tema skrajnima primeroma pa obstajajo tudi delna zasenčitev in s tem onemogočen odboj svetlobe ali pa je na vso mikropovršino prišla svetloba, ki jo kasneje zasenči druga mikropovršina. Faktor slabljenja je tako možno zapisati kot

$$G = 1 - a_1/a_2 \tag{3.25}$$

kjer je $a_1/a_2$ razmerje med površino prestrežene svetlobe in površino mikropovršine. Iz enačbe (3.25) je razvidno, da je v primeru, ko svetloba ni blokirana $G = 1$. Za primer, ko je celotna mikropovršina osvetljena, vendar je odbita svetloba kasneje blokirana, velja izraz

$$G_m = \frac{2(\vec{N} \cdot \vec{H})(\vec{N} \cdot \vec{V})}{\vec{V} \cdot \vec{H}} \tag{3.26}$$

Podoben je izraz za primer, ko je mikropovršina v senci, le da sta vektorja $\vec{L}$ in $\vec{V}$ zamenjana:

$$G_s = \frac{2(\vec{N} \cdot \vec{H})(\vec{N} \cdot \vec{L})}{\vec{V} \cdot \vec{H}} \tag{3.27}$$

Pri določanju faktorja slabljenja $G$ je potrebno uporabiti tisti izraz, ki najbolj blokira svetlobo. Tako izbiramo minimalni faktor slabljenja kot

$$G(\vec{N}, \vec{V}, \vec{L}) = \min\{1, G_s, G_m\} \tag{3.28}$$

### 3.2.3 Fresnelova funkcija odboja

Spreminjanje barve v odblesku je opisano s Fresnelovim zakonom (2.6) in izmerjenimi lastnostmi materiala. Fresnelov izraz v enačbi (3.21) je odbojnost, ki jo predpostavimo za popolnoma gladke površine. Ker običajno odvisnost Fresnelovih enačb (2.6) od valovne dolžine ni znana, sta Cook in Torrance predlagala poenostavitev enačbe za nepolarizirano svetlobo pri podani valovni dolžini:

$$F(g,c) = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left\{1 + \frac{c(g+c)-1)^2}{c(g-c)-1)^2}\right\} \tag{3.29}$$

kjer je $c = cos\theta = \vec{V} \cdot \vec{H}$
$g^2 = \eta_\lambda^2 + c^2 - 1$
$\eta_\lambda =$ lomni količmik pri določeni valovni dolžini $\lambda$

Prednost uporabe Fresnelovih enačb je tudi v tem, da lahko za neznani material izmerimo lomne količnike pri različnih valovnih dolžinah ob pravokotni osvetlitvi s pomočjo enačbe

$$\eta_\lambda = \frac{1 + \sqrt{F_0(\lambda)}}{1 - \sqrt{F_0(\lambda)}} \tag{3.30}$$

kjer je $F_0(\lambda)$ izmerjena odbojnost pri izbrani valovni dolžini.

Pri izrazu za transformacijo $x$ je potrebno uporabiti funkcijo $\max$ zaradi možne numerične napake in negativnih vrednosti. S tem pa zagotovimo pozitivnost tudi ob napaki in $x$ v mejah. Intenziteto lahko zapišemo kot

$$C_(\theta, \lambda) = [C_{\pi/2}(\lambda) - C_0(\lambda)] \frac{\max(0, F(\theta, \lambda) - F_0(\lambda))}{F_{\pi/2}(\lambda) - F_0(\lambda)} + C_0(\lambda) \tag{3.34}$$

ali če uporabimo osnovna izraza v enačbah (3.31) in (3.32)

$$C(\theta, \lambda) = I_i(\lambda) \left\{ [C_{\pi/2}(\lambda) - C_0(\lambda)] \frac{\max(0, F_\theta(\lambda) - F_0(\lambda))}{F_{\pi/2}(\lambda) - F_0(\lambda)} + F_0(\lambda) \right\} \tag{3.35}$$

kar kaže na to, da lahko izračunamo intenziteto odbitega žarka pri željeni valovni dolžini in kotu $\theta$ že, če poznamo intenziteto vpadnega žarka $I_i$. Iz slik 2.2, 2.3 in 2.4, kot tudi pripadajočih enačb, je razvidno, da je pri vpadnem kotu $\theta = \pi/2$ odbojnost vedno enaka ena. Zato lahko v enačbi (3.35) uvedemo substitucijo $F_{\pi/2}(\lambda) = 1$ in s tem dobimo poenostavljeno enačbo

$$C(\theta, \lambda) = I_i(\lambda) \left\{ [1 - F_0(\lambda)] \frac{\max(0, F_\theta(\lambda) - F_0(\lambda))}{1 - F_0(\lambda)} + F_0(\lambda) \right\} \tag{3.36}$$

Za klasični barvni model RGB lahko pri znanih komponentah odbojnosti pod pravim kotom $F_0(\lambda = \{R, G, B\})$ in intenziteti vpadnega žarka $I_i(\lambda)$ zapišemo

$$\begin{aligned}
R(\theta) &= I_i(R) \left\{ [1 - F_0(R)] \tfrac{\max(0, F(\theta, R) - F_0(R))}{1 - F_0(R)} + F_0(R) \right\} \\
G(\theta) &= I_i(G) \left\{ [1 - F_0(G)] \tfrac{\max(0, F(\theta, G) - F_0(G))}{1 - F_0(G)} + F_0(G) \right\} \\
B(\theta) &= I_i(B) \left\{ [1 - F_0(B)] \tfrac{\max(0, F(\theta, B) - F_0(B))}{1 - F_0(B)} + F_0(B) \right\}
\end{aligned} \tag{3.37}$$

Računska zahtevnost enačb (3.37) je le v izračunavanju Fresnelove funkcije $F(\theta, \lambda)$. Z ustreznim tabeliranjem celotnih izrazov (3.37) in (3.21), je možna hitra določitev intenzitete odboja za izbrani material. Z nekaj več spomina je možno vpeljati model, ki bolje opisuje razmere, kot je to primer pri Phongovem modelu, primernemu predvsem za opis površin plastike. Kovine kažejo le majhno primes difuznega odboja in imajo spreminjajočo barvo pri spreminjanju vpadnega kota.

## 3.3 Hallov model osvetlitve

Prejšnja modela odboja svetlobe na površni nista upoštevala svetlobe, ki prehaja skozi površino, kot tudi ne svetlobe, ki jo površine sevajo. V mnogih računalniških programih se barva izračunava z enačbo:

$$C = k_{dr} C_l(\vec{N} \cdot \vec{L}) + k_s C(\vec{R}) + k_{st} C(\vec{T}) + k_{sr} C_l(\vec{R} \cdot \vec{V})^n \tag{3.38}$$

po štirih načinih prenosa svetlobe, opisanih na strani 18. Hallov model, kot razširitev standardnega modela (3.38), z upoštevanjem večjega števila svetlobnih izvorov, zapišemo v naslednji obliki:

$$\begin{aligned} C &= k_{dr} \sum_j C_{l_j} (\vec{N} \cdot \vec{L}_j) && (3.39)\\ &+ k_{sr} \sum_j C_{l_j} (\vec{R}_j \cdot \vec{V})^n \\ &+ k_{st} \sum_j C_{l_j} (\vec{N} \cdot \vec{H}'_j)^{n'} \\ &+ k_{sr} C(\vec{R}) T_r^{\Delta sr} \\ &+ k_{st} C(\vec{T}) T_t^{\Delta st} \\ &+ k_a C_a && (3.40) \end{aligned}$$

kjer je:

$k_{dr} \sum_j C_{l_j} (\vec{N} \cdot \vec{L}_j)$ Difuzni odboj svetlobe, ki je prišla iz svetlobnega izvora. Konstanta $k_{dr}$ podaja difuzno sposobnost površine. Hrapave površine imajo $k_{dr}$ blizu ena, medtem ko zrcalne površine nimajo difuznih lastnosti. Ker je svetlobih izvorov lahko več, je potrebno za vsak izvor določiti kosinus kota med izvorom svetlobe $\vec{L}$ in normalo površine $\vec{N}$. Razpršitev svetlobe torej ustreza Lambertovem zakonu.

$k_{sr} \sum_j C_{l_j} (\vec{R}_j \cdot \vec{V})^n$ Zrcalni odboj. Upoštevati je potrebno več svetlobnih izvorov. Popolni odboj ima največjo intenziteto v smeri odbitega vektorja $\vec{R}$. Za porazdelitev odbleska na večje področje se uporablja Phongova kontrola s potenco kosinusa kota med odbitim žarkom v smeri $\vec{R}$ in kotom gledanja $\vec{V}$. V izrazu $(\vec{R}_j \cdot \vec{V})$ se lahko uporabi tudi polovični vektor $\vec{H}$ (3.16), kot ustrezno merilo za $\cos^n \phi = (\vec{H}_j \cdot \vec{N})^n$.

$k_{st} \sum_j C_{l_j} (\vec{N} \cdot \vec{H}'_j)^{n'}$ Zrcalni prehod svetlobe upošteva količino prepuščene svetlobe iz svetlobnega izvora s konstanto $k_{st}$. Podobno kot pri zrcalnemu odboju, je tudi tu uporabljena Phongova distribucijska funkcija, le da je tu potrebno uporabiti zrcalni vektor prehoda $\vec{T}$. Ustrezni vektor je pri prepuščeni svetlobi $\vec{H}'$, le da je tu potrebno upoštevati drug kot loma svetlobe zaradi različnih lomnih količnikov dveh medijev:

$$\vec{H}'_j = \frac{\vec{V} - \beta \vec{L}_j}{\beta - 1}, \qquad \beta = \frac{\eta_2}{\eta_1} \tag{3.41}$$

$k_{sr} C(\vec{R}) T_r^{\Delta sr}$ Zrcalni odboj svetlobe, ki prihaja z drugih teles. Prehodnost svetlobe na enoto dolžine $T$, skupaj z dolžino žarka $\Delta$, določa jakost

## 3.4 Povzetek

Optične lastnosti površine opišemo z ustreznim matematičnim modelom. Večja stopnja zahtevnosti modela narekuje tudi algoritmično in računsko zahtevnost. Phongov model odboja dobro modelira plastične materiale. Vgrajeno ima tudi kontrolo velikosti površine zrcalnega odbleska. Je enostaven in velikokrat uporabljen na področjih, kjer se zahteva hiter prikaz objekta. Kot izvor svetlobe se pri Phongovem modelu uporablja bela svetloba in točkast izvor.

Kovine in drugi hrapavi materiali imajo posebne lastnosti odboja. Z modeliranjem hrapavosti in spremembo barve se model Cook-Torrance zelo približa realnim razmeram. Odbleski so pravilnejši tudi zaradi tega, ker se v tem modelu uporabljajo Fresnelove enačbe, ki so v prvi vrsti namenjene gladkim zrcalnim površinam, tu pa so razširjene na hrapave površine.

Popolnejše modeliranje uporabno pri metodi sledenja žarku mora upoštevati tudi svetlobo, ki prihaja iz drugih teles in ne samo od svetlobnih izvorov. Hallov model uporablja Phongov način kontrole velikosti odbleskov. Modelira tudi prosojne materiale in upošteva žarke, ki so zrcalni odboj svetlobe prihajajoče iz drugih objektov.

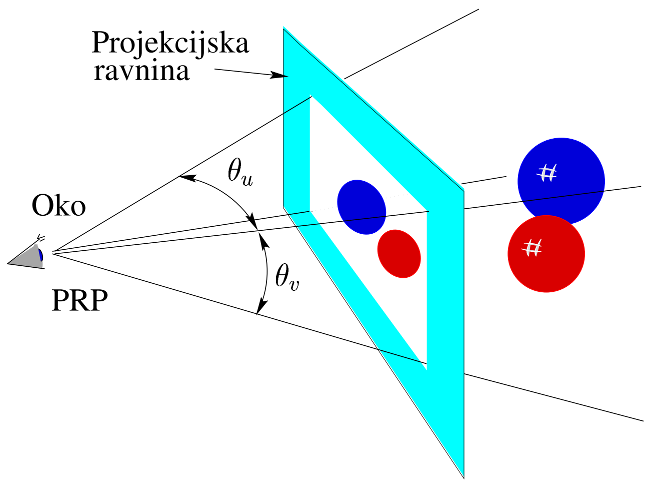
Večina predstavljenih modelov ima vgrajene predpostavke, ki izključujejo nekatere lastnosti fizikalne optike. Primer je polarizacija svetlobe, ki je vidno kot sprememba barve pri dielektrikih. V naravi obstajajo tudi površine, ki ne odbijajo svetlobe v vse smeri enako. Take površine imenujemo *anizotropne površine*. Noben od predstavljenih modelov ne zajema take lastnosti površine. Možne so tudi poenostavitve ali izboljšave posameznega modela, pač glede na potrebe ali zahteve uporabljene metode prikaza.

metodo se predvsem odpravlja slabosti metode žarčenja, ki se odlikuje po pravilni medsebojni osvetlitvi difuznih površin, nima pa vgrajene možnosti zrcalnih odbleskov. Gledano posplošeno, ima metoda žarčenja naravo sledenja žarku v smeri od izvora svetlobe do vseh ostalih vidnih površin, kar ustreza sledenju naprej. Kombinacija s sledenjem žarku nazaj je ena od uporabnih združitev, kjer se izkoristijo prednosti posamezne metode.

## 4.1 Sistem gledanja

Gledanje, kot sistem zaznavanja, se v računalniški grafiki uporablja v povezavi s projekcijo tridimenzionalnih objektov na dvodimenzionalno ploskev. Človeški vid ima zapleten način projekcije prostora. Možne so različne poenostavitve, katerih skupni cilj je projekcija željenega prostora na velikost fotografskega filma. V računalniški grafiki se namesto filma uporablja zaslon ali datoteka.

Izbira sistema gledanja in načina projekcije je odvisna od namena uporabe. Različni modeli z uporabo kamere so predstavljeni v [Wat94]. Največkrat je v uporabi perspektivna projekcija podobna projekciji kamere z majhno odprtino (*camera obscura*). Modifikacija omenjene projekcije, kjer je uporabljena le polovica piramide gledanja, brez obračanja slike, je prikazana na sliki 4.1. Velikost filma je določena s kotom gledanja $(\theta_u, \theta_v)$ in oddaljenostjo projekcijske ravnine od centra projekcije $PRP$.




Slika 4.1: Poenostavljen model gledanja s projekcijsko ravnino

Poleg perspektivne in paralelne projekcije so možne tudi izpeljanke, ki bolje posnemajo človeški vid na projekcijski ravnini. Take projekcije imajo vgrajen model tankih leč s katerimi je možno kontrolirati fokusiranje, kar naredi objekte, ki niso v bližini ravnine fokusa, nekoliko zamegljene.

Pri metodi sledenja žarku se kot osnovni gradniki slike uporabljajo žarki, ki gredo skozi določene točke v naprej predvidenem rastru. S tem projekcijo digitaliziramo na željeno

Pri določanju primarnih žarkov $\vec{E}$ variramo celoštevilčne indekse pozicije $x$ in $y$ v mejah

$$0 \ \leq x < res_x$$
$$0 \ \leq y < res_y$$

Izračun smeri primarnega žarka $\vec{E}$ za podana indeksa $x$ in $y$ se določi po sledečem postopku:

1. Določimo smer gledanja in jo normiramo
$$\vec{dop} = \frac{\vec{to} - \vec{from}}{|\vec{to} - \vec{from}|} \tag{4.2}$$

2. Z vektorskim produktom določimo enotski vektor $\vec{u}$ na projekcijski ravnini v horizontalni smeri
$$\vec{u} = \frac{\vec{dop} \times \vec{up}}{|\vec{dop} \times \vec{up}|} \tag{4.3}$$

3. Izračunamo enotski vektor $\vec{v}$ na projekcijski ravnini v vertikalni smeri
$$\vec{v} = \frac{\vec{u} \times \vec{dop}}{|\vec{u} \times \vec{dop}|} \tag{4.4}$$

4. Ob podanih kotih gledanja $\theta_u$, $\theta_v$ in oddaljenosti projekcijske ravnine $h$ zapišemo polovično velikost filma v ustrezni smeri, kot:
$$u_{max} = h \tan(\theta_u/2) \tag{4.5}$$
$$v_{max} = h \tan(\theta_v/2) \tag{4.6}$$

5. Prestavimo koordinatno središče $(\vec{u}, \vec{v})$, tako da bo pri indeksu $x = y = 0$ vektor $\vec{E}$ kazal v zgornji levi kot. Vektor $\vec{D}$ podaja ustrezni odmik v projekcijski ravnini in se ga zapiše kot:
$$\vec{D} = \left[\frac{2x}{res_x} - 1\right] u_{max}\vec{u} - \left[\frac{2y}{res_y} - 1\right] v_{max}\vec{v} \tag{4.7}$$
V enačbi (4.7) se tudi obrne orientacija vertikalne osi $\vec{v}$, ker je taka orientacija bolj primerna za pisanje v rastrsko datoteko.

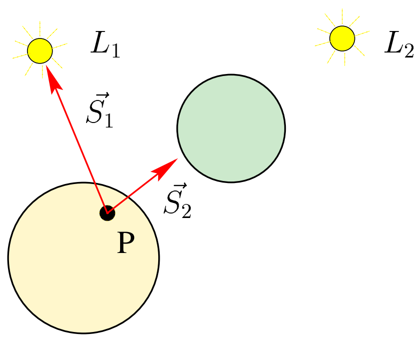
6. Končno določimo smerni vektor primarnih žarkov $\vec{E}$, ki ga po potrebi normiramo
$$\vec{E} = \frac{h\,\vec{dop} + \vec{D}}{|h\,\vec{dop} + \vec{D}|} \tag{4.8}$$

Iz slike 4.2 je razvidno, da je izračun smeri primarnega žarka $\vec{E}$ v enačbi (4.8), neodvisen od izbire oddaljenosti projekcijske ravnine $h$. To pa zato, ker se ob predpisanih kotih gledanja $\theta_u$, $\theta_v$ in ločljivosti slike, velikost ene rastrske točke (*piksla*) sorazmerno veča z oddaljenostjo $h$. Če izberemo $h = 1$ se izračun vektorja $\vec{E}$ v enačbi (4.8) še dodatno poenostavi.

### 4.2.1 Senčni žarki

Phongov model odboja svetlobe je namenjen le za neprosojne površine. Izvori svetlobe so točkasti in bele barve. Intenziteto za izbrano točko lahko določimo že, če poznamo položaj svetlobnih virov.

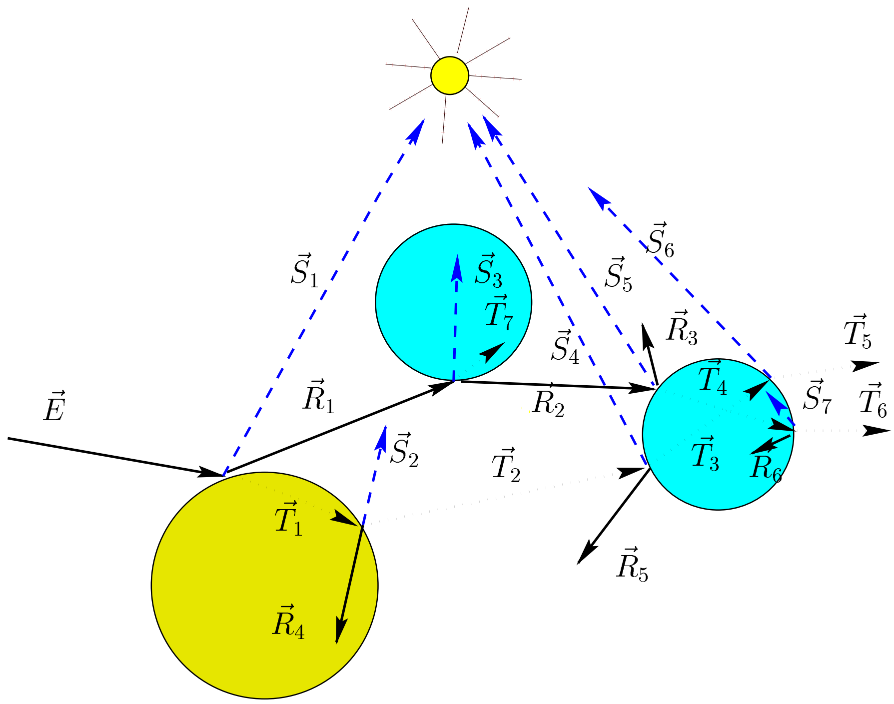
Na sliki 4.4 je prikazan primer z dvema svetlobnima izvoroma $L_1$ in $L_2$. V točki $P$ je potrebno določiti prispevek zaradi direktne osvetlitve vseh svetlobnih izvorov. Iz točke $P$ pošljemo žarke v smeri vsakega svetlobnega izvora. Če je žarek pri potovanju do svetlobnega izvora presekal kakšen objekt, potem točka $P$ ni direktno osvetljena s tem svetlobnim virom in se ga zato ne upošteva pri določitvi prispevka direktne osvetlitve. Če bi




Slika 4.4: Direktna osvetlitev točke $P$ z blokirano svetlobo v smeri $\vec{S}_2$

bili objekti, ki blokirajo svetlobo prosojni, bi bilo potrebno upoštevati tudi ustrezno zmanjšanje svetlobe. Problem, ki pri tem nastopi, je prav v določitvi intenzitete tako zmanjšane svetlobe. Upoštevati je namreč potrebno, da se pri transparentnih objekti svetloba tudi lomi. Dodatna težava je tudi v tem, da se v modelih normalno podaja lastnost površine in ne lastnost materiala v notranjosti, kjer se svetloba že pri manjši motnosti razprši. Večina osnovnih programov nima vgrajenega dušenja blokiranih žarkov, saj si pomagajo z prepuščenimi žarki $\vec{T}$ in s tem manjšajo pomembnost zanemaritve blokirane svetlobe na transparentnih objektih.

Iskanje presekov od točke $P$ do vseh izvorov svetlobe $L_i$ ni nič bolj zahtevno kot določitev preseka žarka $\vec{E}$ z vsemi objekti. Z dodajanjem novih svetlobnih virov je potrebno za vse točke v rastru opraviti dodatne teste, kar lahko zelo upočasni proces določanja senčnih žarkov (*shadow feelers*). Ker je globina rekurzivnega iskanja žarkov največkrat omejena na dve ali tri nivoje, je očitno, da je določanje direktne osvetlitve lahko dominantno delo v procesu upodabljanja. Phongov, kot tudi Cook-Torrance model, lahko zadovoljivo apliciramo ne da bi upoštevali svetlobo, ki prihaja z drugih površin. S tem zavestno izključimo zrcalne in transparentne površine, kar včasih tudi ni toliko pomembno.




Slika 4.5: Potovanje žarka $\vec{E}$. Na vsaki presečni površini se generirajo odbiti $\vec{R}$, prepuščeni $\vec{T}$ in senčni žarki $\vec{S}$

kjer je prvi izraz lokalna osvetlitev, drugi izraz direktna osvetlitev. Zadnja dva izraza sta globalna osvetlitev, ki jo prispevata zrcalni in prepuščeni žarek.

Določitev globalne osvetlitve se algoritmično ne razlikuje od določitve osvetlitve za primarne žarke, kar poenostavi izvedbo programa z rekurzivno naravo določanja globalne osvetlitve. Primarne žarke generiramo po sledečem algoritmu:

**proc** **Screen**(Comp *scene, *Camera *view, Display *display, Color *pic)
**begin**
  { določi meje rastrskega pomnilnika in koren vseh objektov }
  **for** $y := y_{min}$ **to** $y_{max}$ **do**
    **for** $x := x_{min}$ **to** $x_{max}$ **do**
      { izračunaj smer primarnega žarka $\vec{E}$, ki gre skozi
        zaslonsko točko $(x + 0.5, y + 0.5)$ }
      **Trace**(0, $o\vec{k}o$, $\vec{E}$, $barva$);
      { shrani $barvo$ v rastrski pomnilnik *pic }
    **od**
  **od**
**end**

Algoritem sledenja žarkov **Trace** je isti za vse primarne in sekundarne žarke. Je rekurziven z omejitvijo globine. Nekatere implementacije delijo **Trace** na dva podprograma.

možno brez velikih težav razšiti sistem iz enostavnih objektov, kot so krogle in in planarni mnogokotniki, na zahtevnejše površine (NURBS-i, (super)kvadriki, blobs-i, ...).

### 4.3.1 Krogle

Krogle, kot enostavne in zaprte površine, so bile tudi najprej uporabljene v metodi sledenja žarku. Uporabljajo se tudi za omejitvene prostore (*bounding volumes*) kompliciranih objektov v algoritmih z vnaprejšnjim izločanjem možnih presečnih objektov.

Predpostavimo, da imamo žarek $\vec{r} = \vec{o} + t\vec{v}$ in kroglo $S$ definirano s centrom v točki $\vec{c}$ in radijem $R$. Enačba površine na krogli je:

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = R^2 \tag{4.13}$$

kar lahko zapišemo tudi v vektorski obliki za točko $\vec{p}$ na površini:

$$(\vec{p} - \vec{c}) \cdot (\vec{p} - \vec{c}) = R^2 \tag{4.14}$$

Žarek $\vec{r}$ bo prebodel kroglo $S$ pri vsaki vrednosti $t$, ki zadovoljuje enačbo

$$(\vec{o} + t\vec{v} - \vec{c}) \cdot (\vec{o} + t\vec{v} - \vec{c}) = R^2 \tag{4.15}$$

Z razširitvijo dobimo kvadratno enačbo za parameter $t$:

$$\vec{v} \cdot \vec{v}\, t^2 + 2(\vec{o} - \vec{c}) \cdot \vec{v}\, t + (\vec{o} - \vec{c}) \cdot (\vec{o} - \vec{c}) - R^2 = 0 \tag{4.16}$$

Rešitev kvadratne enačbe nam da

$$t = \frac{-4(\vec{o} - \vec{c}) \cdot \vec{v} \pm \sqrt{4[(\vec{o} - \vec{c}) \cdot \vec{v}]^2 - 4\vec{v} \cdot \vec{v}[(\vec{o} - \vec{c}) \cdot (\vec{o} - \vec{c}) - R^2]}}{2\vec{v} \cdot \vec{v}} \tag{4.17}$$

Diskriminanta (izraz pod korenom) ogotavlja ali preseka obstajata. Če je pozitivna dobimo dva presečišča. Če je izraz nič se žarek le dotika krogle. Pri negativnem izrazu pa presek žarka in krogle ne obstaja. Smerni vektor $\vec{v}$ je lahko tudi enotski. Tedaj velja $\vec{v} \cdot \vec{v} = 1$, kar dodatno poenostavi enačbo (4.17). Z uvedbo enotskega vektorja je lahko $t$ tudi merilo za oddaljenost presečišča od izhodišča $\vec{o}$.

### 4.3.2 Trikotniki

Zahtevne površine se velikokrat v praksi raje aproksimirajo z množico trikotnikov ali množico planarnih konveksnih mnogokotnikov. Izračun presekov je lažji, napaka zaradi diskretizacije pa normalno ni opazna, če uporabimo tehniko inkrementalnega senčenja in prilagodljivo delitev parametričnih površin z oceno ukrivljenosti v ploskvi.

Predpostavimo, da imamo žarek $\vec{r} = \vec{o} + t\vec{v}$ in trikotnik $T$ definiran s tremi nekolinearnimi ogljišči $\vec{p}_1$, $\vec{p}_2$ in $\vec{p}_3$.

3. $u + v \leq 1$

Inverzne koordinate $u$ in $v$ določimo iz enačbe (4.22) tako, da vektorsko enačbo napišemo po komponentah. Ker so komponentne enačbe tri, neznanki pa dve, je ena od enačb odveč. Osnovno pravilo pri odločanju, katero koordinato izločiti je, da izločimo dominantno koordinato; to je tista, ki ima v normali ravnine $\vec{N}$ absolutno največjo vrednost. Izberimo primer ravnine, ki ima največjo vrednost v koordinati $z$ normale $\vec{N}$. Komponentni enačbi sta:

$$\begin{aligned} p_x - p_{1x} &= u(p_{2x} - p_{1x}) + v(p_{3x} - p_{1x}) \\ p_y - p_{1y} &= u(p_{2y} - p_{1y}) + v(p_{3y} - p_{1y}) \end{aligned} \tag{4.23}$$

iz katerih lahko izrazimo položaj točke v koordinatnemu sistemu ravnine $(u, v)$:

$$\begin{aligned} u &= \frac{(p_x - p_{1x})(p_{3y} - p_{1y}) - (p_y - p_{1y})(p_{3x} - p_{1x})}{(p_{2x} - p_{1x})(p_{3y} - p_{1y}) - (p_{2y} - p_{1y})(p_{3x} - p_{1x})} \\ v &= \frac{(p_x - p_{1x})(p_{2y} - p_{1y}) - (p_y - p_{1y})((p_{2x} - p_{1x})}{(p_{3x} - p_{1x})(p_{2y} - p_{1y}) - (p_{3y} - p_{1y})(p_{2x} - p_{1x})} \end{aligned} \tag{4.24}$$

Po določitvi presečne točke, je potrebno določiti pravilno stran ravnine, tako da velja enačba $\vec{I} \cdot \vec{N} \geq 0$. V dosedanjem obravnavanju se za normalo ni zahtevala enotska dolžina, je pa običajno potrebna pri osvetlitvenem modelu.

### 4.3.3 Pregled ostalih površin

Poleg predstavljenih osnovnih primitivov, se v računalniški grafiki uporablja še množica raznoraznih površin s specifičnimi lastnostmi. Primernost posameznih primitivov je odvisna od namena in zahtevnosti teorije, ki jo želimo aplicirati.

Logična razšititev trikotnikov na mnogokotnike je tudi ta, da mnogotnike sestavimo iz trikotnikov. Delaunay triangulacija[Mul94] je ena od uspešnih metod za planarne mnogokotnike poljubnih oblik. Razdelitev na množico trikotnikov pa nujno poveča računski čas. Druge metode testiranja točke v mnogokotniku so prikazane v [Hai94]. Hitrost testiranja notranjosti je poleg števila vozlišč odvisna tudi od konveksnosti mnogokotnika.

Kvadriki (*quadrics*) so implicitne površine druge stopnje, za katere obstaja analitična rešitev presekov. Z variacijo parametrov v enačbi dobimo elipsoide, cilindre, svitke in druge objekte. Razširitev z večjim naborom osnovnih primitivov (kvadri, krogle, stožci) je možna z uporabo superkvadrikov[Bar81] za katere obstaja analitičen test preseka z žarkom, vendar je potrebno presek izračunati numerično.

Splošna metoda pri reševanju parametričnih površin je diskretizacija na štirikotnike in kasnejše inkrementalno senčenje. Tak način seveda ni natančen. Pri grobih delitvah takih površin so opazni Mach-ovi pasovi. Točnejše obravnavanje v veliki večini zahteva numerično določanje lokalnih koordinat $(u, v)$. Numerično reševanje presekov pa ima lahko težave s singularnostjo zaradi raznih vzrokov.

V uporabi najdemo tudi druge implicitne, eksplicitne, algebraične in proceduralne površine. Naj jih le naštejem:

- Kako naj bo upoštevana indirektna osvetlitev in ali zadostuje enostavni model z enim odbitim in prepuščenim žarkom na presek?
- Do kakšne stopnje je potrebno filtrirati teksture in s katerimi algoritmi, da se ne pojavi *aliasing*.
- V sistemih s teksturami so možne tudi simulacije globine (*bump mapping*). Ali je potrebno modulirati le normale na površini, ali je nujno upoštevati tudi realni dvig teksture.
- Kako upoštevati osvetlitev okolice in medija kot je zrak?
- Kako učinkovito opišemo scene za animacijo in kako lahko uporabimo prejšnje sekvence in s tem zmanjšamo potreben čas.

Oceno stopnje zahtevnosti trenutnih algoritmov v upodabljanju in druga vprašanja glede kompleksnosti in subjektivne ocene realističnih slik najdemo v [CS94]. Možne so tudi izboljšave v pogledu časovne optimizacije posameznih algoritmov.

Ena osnovnih izboljšav, ki se lahko enostavno aplicirajo v obstoječe algoritme s predpisano maksimalno globino rekurzije, temelji na dejstvu, da se pri večkratnem odbijanju žarka, manjša tudi pomembnost zadnjih žarkov. Prispevki k osvetlitvi zaporedno generiranih žarkov se na koncu rekurzije množijo med seboj. Tudi pri visoko odbojnih scenah se kaj hitro zgodi, da že po nekaj odbojih zadnji žarek ne vpliva bistveno na osvetlitev prvega presečišča. Zaradi tega je možno vpeljati adaptivno kontrolo pomembnosti sukcesivnih sekundarnih žarkov, ki lahko prekine rekurzivno generiranje žarkov še predno doseže maksimalno predpisano globino. Tako se v rutino **Trace()** na strani 43 normalno uvede dodatna utežna spremenljivka $weight$, ki se kumulativno računa (množi) z lokalnim koeficientom za zrcalni odboj ali zrcalni prehod. Predpisani minimalni prispevek normalno velja globalno za celotno sceno.

Drugi pomemben pristop k povečanju hitrosti računanja presekov je uporaba omejitvenih volumnov (*bounding volume*) za objekte pri katerih je računanje presekov zelo zahtevno. Osnovna ideja je v tem, da za kompleksne objekte poiščemo enostavnejše objekte, ki v celoti vsebujejo kompleksni objekt (površino). Ko iščemo presek žarka z vsemi možnimi objekti se za komplekne objekte raje uporabi presečni test omejitvenega volumna, ki ga veliko hitreje izračunamo. Šele če obstaja presek med žarkom in omejitvenim volumnom, se pristopi k izračunavanju preseka med kompleksnim objektom in žarkom. Najenostavnejši omejitveni volumen je seveda krogla, ki pa nima lastnosti tesnega prileganja in je lahko zaradi tega neučinkovita.

Kot omejitveni volumni so možni tudi različni kvadri poljubno orientirani v prostoru. Kot omejitveni volmuen se lahko uporabi tudi presek dveh ali več poljubnih omejitvenih prostorov. Pri taki omejitvi se pri testiranju možnosti obstoja preseka kompleksnega objekta in žarka, testira presek z vsemi volumni, ki kompleksni objekt omejujejo. Primer takega objekta je prikazan na sliki 4.7. Presek s kompleksnim objektom je smiselno pa največkrat ni primerna zaradi spominske zahtevnosti, kot tudi algoritmične počasnosti.

Boljše sheme, kot sta oktalno drevo (*octree*) in binarna razdelitev prostora (*Binary Space Partitioning*), racionalneje razdelijo prostor. Omenjeni shemi imata strogo urejeno drevo z možnostjo adaptivne delitve prostora na mestih z večjo gostoto objektov. Določanje položaja točke v drevesu je dokaj hitro tudi pri podrobnejših delitva prostora. Iskanje položaja točke se prične pri korenu drevesa in konča na končnem listu v katerem je zapisan objekt, ki vsebuje testirano kocko v prostoru. V scenah z neenakomerno porazdelitvijo se lahko zgodi, da drevo pretirano raste v nekaterih smereh, zato je potrebno uporabiti modificirane sheme, ki uravnotežijo drevo tako, da je dostopni čas do vsakega lista enak.

V metodo sledenja žarku lahko enostavno vpeljemo tudi konstruktivno geometrijo trdnih teles (*Constructive Solid Geometry*). Z uvedbo CSG pa je potrebno adaptirati večino algoritmov: od osnovnih za iskanje najbližjega presečišča pa do algoritmov za apriori delitve prostora.

Poleg omenjenih algoritmov s katerimi je možno pospešiti osnovno metodo sledenja žarku, obstajajo tudi nekateri drugi, z manjšim prispevkom k hitrosti [AK91]. Paralelizacija vseh algoritmov je ena od metod, ki bistveno pripomore k zmanjšanju časa potrebnega za upodabljanje in bo podrobneje predstavljena na konkretnem primeru v poglavju 7.

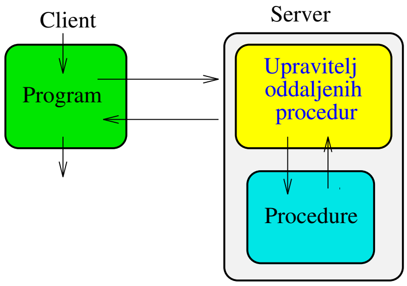
## 4.5 Povzetek

Metoda sledenja žarku temelji na geometrijski optiki, pri kateri je osnova žarek, ki potuje od izvora do sprejemnika (očesa). Poznamo dve metodi sledenja žarku: sledenje naprej in sledenje nazaj. Sledenje nazaj je v izključni uporabi v računalniški grafiki. Pri sledenju žarku od očesa proti možnim izvorom svetlobe, pošiljamo primarne žarke skozi pravokotno mrežo točk in s tem tudi digitaliziramo tridimenzionalni prostor.

Osnovna metoda sledenja žarku je rekurzivna. Primarni žarki ob dotiku z objektom v splošnem generirajo dva dodatna žarka, ki jima rekurzivno sledimo do maksimalne predpisane globine. Glavno delo pri metodi je določanje najbližjih presečnih točk žarka z objekti. Ob znani presečni točki se določi normala površine in aplicira osvetlitveni model za katerega je potrebno določiti še senčne žarke in upoštevati rekurzivne prispevke odbitih in prepuščenih žarkov.

Poznavanje geometrije je osnova za izračunavanje presekov in normal na objektih. Različni objekti imajo različne težavnosti pri določanju preseka. Preseki za krogle in trikotnike so predstavljeni kot osnova vseh programov za sledenje žarku. Diskutirane so tudi ostale površine, ki jih tudi najdemo v računalniški grafiki.

Osnovno metodo je mogoče izboljšati na več načinov. Osnovno vodilo je zmanjšanje celotnega časa in povečanje realističnosti pri upodabljanju. Diskutirani so različni prijemi, ki lahko zmanjšajo potreben čas za upodobitev. Paralelizacija je eden od učinkovitih načinov, s katerim povečamo hitrost metode.




Slika 5.1: Princip izvajanja oddaljenih procedur z mehanizmom RPC

V praksi obstajata dva tipa strežnikov, ki se uporabljata v distribuiranih sistemih. Prvi tip strežnika hrani informacije o zahtevah odjemalcev. S takim pristopom strežnik ve, kateri odjemalci trenutno zahtevajo servise in se lahko prilagodi njihovim specifičnim zahtevam. Poleg tega lahko strežnik tudi hrani informacije o posameznih odjemalcih, ki tako nastavljajo notranje stanje strežnika za vsakega odjemalca posebej. Strežniki s stanjem hranijo informacije o odprtih datotekah, številu zahtev, stanju obdelave, ... Drugi tip strežnika je brez hranjenja informacij o odjemnikih. Tak tip strežnika tako za vsako operacijo zahteva popoln nabor podatkov, kar pa je lahko obremenjujoče za komunikacijo po mreži saj strežniki s stanjem lahko hranijo začasne podatke do zaključka dela odjemalca.

Tipični primer strežnika s stanjem je strežnik za bazo podatkov, kjer strežnik hrani transakcije odjemalcev ter s tem omogoči t.i. *rollback* in popolno kontrolo pri sinhronizaciji z reševanjem konfliktov med različnimi odjemalci. Težava strežnikov s stanjem pa je tudi ta, da ob morebitnem padcu sistema (*crash*) ni zagotovljena obnovitev prejšnjega stanja strežnika. Seveda so tudi možnosti za obnovitev stanja, ki pa niso enostavno izvedljive.

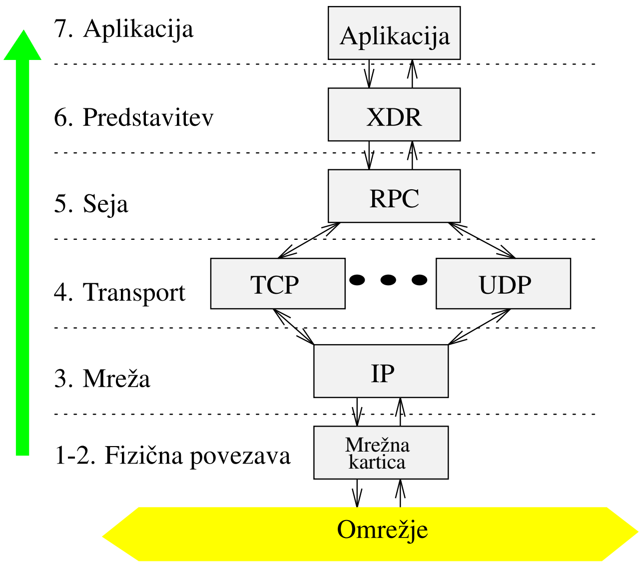
Pri razvijanju distribuiranih aplikacijah je v primerjavi s klasičnimi programi potrebno upoštevati sledeče značilnosti:

- Količina podatkov, ki se bo izmenjevala med računalniki mora biti razumno velika in časovno čim bolj enakomerno razporejena.

- Mrežne aplikacije morajo strožje kontrolirati možne napake pri komunikaciji in jih v zadovoljivi meri reševati. V več-strežniškem okolju je možno zagotoviti redundantnost in s tem večjo robustnost aplikacij.

- Če je varnost podatkov pomembna, je potrebno upoštevati, da so mrežni sistemi občutljivi na vdore v sistem predvsem zaradi nepazljivosti razvijalcev programske opreme. Vse knjižnice za distribuirane sisteme vsebujejo tudi možnost šifriranja podatkov po standardu DES.

- Za optimalne lastnosti distribuirane aplikacije je potrebno skrbno predvideti, katere procedure bodo lokalne in katere oddaljene. Aplikacija bi morala biti sama zmožna poiskati ustrezne servise na mreži.




Slika 5.2: Sedem slojev OSI z RPC

### 5.1.3 Sistem P4

Odpravljanje težav s kodiranjem podatkov med različnimi sistemi lahko izvedemo z nadgradnjo sistema RPC. *P4* je ena takih knjižnic, razvita v Argonne National Laboratory, s katero lahko enostavno programiramo različne paralelne programe. Poleg pošiljanja sporočil (*message passing*), sistem p4 omogoča tudi model z deljenim spominom (*shared memory model*), ki se ga uporablja na sistemih z več procesorji.

Tako kot pri večini distribuiranih sistemov je tu omogočen sistem *master-slave* z več stopnjami hierarhije in združevanja. Kreiranje novih procesov ni veliko boljše kot pri osnovnem RPC, kjer mora strežnik že prej delovati, preden lahko uporabljamo njegove servise. Procesi pri *p4* so tako statični. Distribuirani sistem se nastavlja s konfiguracijsko datoteko v katero zapišemo računalnike in servise, ki so dostopni na mreži. S sistemom *p4* je možno kreiranje dinamičnih procesov preko vnaprej postavljenih statičnih procesov. Sporočila se pošiljajo podobno kot v vseh sistemih za izmenjavo podatkov, le da je veliko dela potrebno s kreiranjem vmesnih pomnilnikov (*bufferjev*).

### 5.1.4 PVM

Leta 1992 so na univerzi Tennesee, univerzi Emory (Atlanta) in v Oak Ridge National Laboratory razvili platformo *Parallel Virtual Machine System*, ki pa za razliko od prejšnjih omogoča, da si posamezni računalniki lahko delijo procedure na mreži, kot tudi lokalno med procesi v multiprocesorskih ali vektorskih sistemih[GBD$^+$94]. Šele v zadnji verziji združuje multiprocesorsko arhitekturo in mrežo delovnih postaj. Trenutno deluje na večini operacijskih sistemov Unix in na nekaterih paralelnih računalnikih kot so: iPSC/860 (Intel), Paragon XPS5 (Intel) in 32-node CM5 (Thinking Machines Corp.).

prostor distribuiran. Tako kot večina sistemov, ima tudi ta model knjižnice, ki nadgrajujejo znane programske jezike. Za vzdrževanje skupnega sistema pa je izdelana nadgradnja operacijskega sistema.

## 5.2 Paralelno procesiranje

Paralelni računalnik je množica procesorjev, ki so zmožni delovati skupaj, z namenom, da rešijo računski problem. Taka definicija zajema paleto računalnikov kot so: super računalniki z velikim številom procesorjev, mreža računalnikov, delovne postaje z večimi procesorji in posebni računalniki, namensko grajeni (npr. transputerji povezani v skupino).

### 5.2.1 Terminologija

Za razumevanje distriburiranega procesiranja je potrebno poznati različne algoritme paralelnega procesiranja, ki pa so v veliki meri odvisni od arhitekture sistema na katerem problem rešujemo. Arhitekture sistemov ločimo po naslednjih bistvenih značilnostih:

**Odvisnost nalog** govori o odvisnosti podnalog, za izvršitev skupnega cilja. Naloge se ločijo na med seboj odvisne in neodvisne naloge, kar zahteva sinhronizacijo nalog. Tu se pojavijo tudi časovni zastoji in slaba uravnoteženost procesov, ki jo lahko z večjo povezanostjo procesorjev in ustrezno delitvijo zmanjšamo.

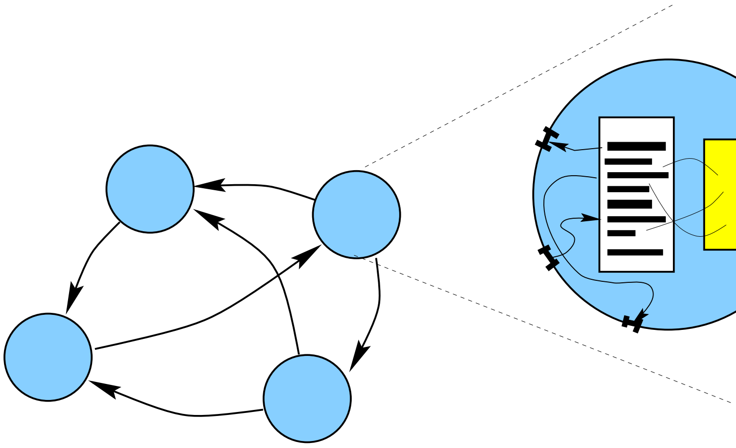
**Povezanost procesorjev** zajema način komunikacije med procesorji. Tesno vezani procesorji si delijo spomin preko skupnega vodila ali pa celo uporabljajo t.i. *multi-port* spomin, ki je običajno sinhronega značaja. Ohlapno vezani procesorji imajo porazdeljen spomin in nadzor. Komunikacija med takimi procesorji mora upoštevati zakasnitve in nepredvidene težave pri komunikaciji.

**Razdrobljenost procesov** opisuje velikost paralelnih procesov. Groba delitev običajno zajema manjše število procesorjev z večjimi zmogljivostmi. Zaradi velikih deležev nalog, je pri grobi delitvi opaziti trenutne in goste pretoke informacij. Pri grobi delitvi je odgovornost programerja, da pravilno razdeli naloge. Pri fini delitvi pa je delitev na procesorje lahko izvedena že na nivoju prevajalnika oz. specializiranega programskega jezika za ciljno arhitekturo računalnika.

Glede na gornje pojme se delijo tudi algoritmi, ki jih uporabljamo pri paralelnem procesiranju:

- Skupno procesiranje (*Concurrent processing*) povezuje več med seboj odvisnih procesov, ki komunicirajo med seboj. To je običajno več procesorjev, ki si izmenjujejo podatke s skupnim spominom (*Multi-port RAM*).

- Večprocesni paralelizem (*Multi-processing parallelism*) se izvaja na večih procesorjih, ki so med seboj neodvisni, vsak s svojim spominom in procesno enoto. Tako



Slika 5.3: Enostavni paralelni programski model

od vozlov je podrobno prikazan kot von Neumanov programski model, v katerem je razvidna procesna enota, spominska enota in vhodno/izhodni komunikacijski kanali. Kanali po katerih se pretakajo podatki imajo lahko tudi vmesne pomnilnike (*bufferje*), ki so zmožni postavljanja več sporočil (podatkov) v vrsto (*message queue*). Prikazani programski model ima naslednje lastnosti:

- Paralelno računanje je sestavljeno iz več nalog, ki se izvajajo vzporedno. Število nalog se lahko spreminja tekom izvajanja programa.
- Vsaka naloga lahko lokalno izvaja štiri osnovne operacije. Te so: pisanje in branje v lokalni spomin, branje in pisanje v/iz komikacijskega kanala, kreiranje novih nalog in končanje naloge.
- Vsaka naloga je sekvenčni program, ki uporablja svoj lokalni spomin.
- Pošiljanje sporočil je asinhrono – oddaja se izvrši takoj. Sprejem sporočil je sinhron – izvajanje stoji na mestu toliko časa, dokler ne pride novo sporočilo.

Pri razvoju paralelnih programov je potrebno izvajati modularno zasnovo na način, s katerim je možno učinkovito izdelati osnovne naloge, ki bodo dovolj majhne in hkrati tako velike, da bo čim manj nepotrebnega komuniciranja med nalogami, ter s tem hitrejše izvrševanje celotnega paralelnega programa. Pomemba lastnost, ki jo je potrebno imeti v mislih pri razvoju paralelnih programov je tudi *skalabilnost*. To pomeni, da je potrebno program napisati program tako, da brez težav deluje na enem ali več procesorjev.

Izvrševanje posameznih nalog naj bi bilo neodvisno od tega, kje se naloga izvrši. Procesorji naj bi bili s stališča lokacije enakovredni. To je posebno pomembno pri računalnikih v mreži, kjer imamo različne procesorje in različne operacijske sisteme. Klasičen primer, pri katerem naloge niso neodvisne od lokacije, je implementacija metode Monte–Carlo, kjer najdemo različne generatorje naključnih števil na različnih operacijskih sistemih.

Paralelno programiranje v distribuiranih sistemih je model, kjer je vsak vozel hkrati strežnik in odjemalec. Paralelni programski model s posredovanjem sporočil (*message passing*) je najbolj razširjen model paralelnega programiranja. Model z deljenim spominom (*shared memory model*) je model, pri katerem si vozli izmenjujejo podatke preko skupnega pomnilnika. Tak način programiranja je lažji, ker ni potrebno razporejati pretoka podatkov med vozli.

## 6.1 Značilnosti PVM

Šele v zadnji verziji sistem PVM omogoča razvoj paralelnih programov, tako za model masivnih paralelnih procesorjev, kot tudi distribuirani model. Razvoj programov je možen v jezikih C, C++ in Fortran. Omenjeni jeziki so tudi najbolj razširjeni in primerni za različne namene. Knjižnice so izvedene v obliki funkcij, ki vračajo status uspešnosti izvršitve posamezne funkcije. To je običajna praksa tudi v drugih knjižnicah, saj se pri vsaki funkciji lahko zgodi, da se ne konča uspešno zaradi različnih vzrokov, ki so pogojeni s komunikacijo po mreži in drugimi omejitvami v sistemskih virih računalnika. Za Fortran so knjižnice izvedene v obliki podprogramov in ne kot funkcije, kar povzroča nekaj več težav pri kontroliranju uspešnosti izvršitve klicanega podprograma. Osnovne značilnosti sistema PVM so:

- Računski model s posredovanjem sporočil (*message passing*). Skupina procesov si med seboj izmenjuje podatke, s ciljem rešitve prvotnega problema, ki je lahko med procesi razdeljen po funkcijah, po podatkih ali mešano. Velikost sporočila je omejena le s spominom računalnika.

- Osnovna računska enota je proces, ki na računalnikih z operacijskim sistemom Unix ustreza enemu procesu v sistemu. Izvrševanje naloge ni vezano na procesor, kar pomeni, da lahko na enemu procesorju izvršujemo več procesov hkrati, če to operacijski sistem omogoča.

- Dostop do računskih zmogljivosti je transparenten. Računalniki, ki so prijavljeni v sistem PVM so vidni le kot posredniki računske moči. Paralelni program lahko prosto izbira, katere računalnike bo izbral za izvajanje.

- Sestavljanje virtualnega paralelnega računalnika je možno s programom za konfiguracijo. Tako lahko računalniki z enim procesorjem, kot tudi MPP sistemi, sodelujejo v sistemu. Nove računalnike lahko dodajamo k skupini in jih po želji tudi brišemo iz sistema PVM.

- Podpora heterogenosti v sistemu PVM velja za računalnike, omrežja in aplikacije. Pri posredovanju podatkov je možno imeti različna kodiranja podatkov simultano. Na sistemih MPP je možna izmenjava podatkov med procesorji brez kodiranja, kar pospeši prenos podatkov.

Programski model PVM ustreza von Neumanovemu modelu navideznih računalnikov. Osnovne enote v PVM obdelave so procesi (*tasks*). Vsak proces je odgovoren za del računske obremenitve. Možne so razdelitve po funkcijah, kjer procesi izvajajo različne naloge. Vsak proces zna opravljati le nekatere naloge. Bolj pogosta delitev pa je delitev podatkov, kjer vsak proces dobi za nalogo rešitev podproblema. V tem modelu so vsi procesi enaki, operirajo le z različnimi podatki. Tak programski model je znan pod imenom SPMD (*Single Program Multiple Data*). Po rešitvi podproblemov se delni rezultati vseh procesov statično (*static scheduling*). Problem, ki se pri tem pojavi je ta, da ni vedno znan čas, ki ga bo vozel porabil za izračun naloge in se lahko zgodi, da procesi čakajo na sinhronizacijo z nalogami, ki še vedno tečejo zaradi slabe vnaprejšnje ocene hitrosti procesa. Dinamičen pristop k delitvi dela je ta, da so naloge bolj razdrobljene in jih je običajno več, kot pa delavcev. Naloge se tako dinamično dodeljujejo: proces dobi novo delo takoj, ko je končal s prejšnjim.

Razdelitev dela v vozlih je možna tudi po funkcijah, kar pomeni, da imamo v vozliščih različne programe. Primer takih programov so cevovodi. Tu se podatki pretakajo od vozlišča do vozlišča. Vsak vozel obdela prihajajoče podatke in jih pošilja naprej. Ker so procesi med seboj različni, je predstavljena shema ustrezna modelu MPSD (*Multiple Program Single Data*).

## 6.2 Programski vmesnik

Tako kot večina distribuiranih sistemov ima tudi PVM knjižnico podprogramov, ki jo uporabljamo v standardnih sekvenčnih programskih jezikih. Komunikacijski model je v osnovi asinhron za pošiljanje in za sprejemanje podatkov. Zaradi take sheme morajo imeti sodelujoči računalniki dovolj velike pomnilnike, da lahko shranjujejo prispela sporočila v vmesni pomnilnik. Omogočen je tudi asinhron sprejem brez blokiranja izvajanja programa. PVM zagotavlja urejenost prihajanja podatkov v sprejemni pomnilnik. Tudi pri pošiljanju podatkov se ohranja vrstni red. Če proces 1 najprej pošlje sporočilo A procesu 2 in nato pošlje sporočilo B procesu 2, bo sporočilo A prišlo pred sporočilom B. Vmesni pomnilniki za sprejem se alocirajo avtomatsko (ob prispetju sporočila). To pomeni, da ni vnaprej definirane velikosti sprejemnega pomnilnika, kar omogoča sprejem sporočil poljubnih velikosti, ki so omejene le z velikostjo pomnilnika. Ker je sprejem asinhron, se sporočila zlagajo v vrsto in če sprejemnik ne zajema sporočil, se lahko zgodi, da se porabi celotni pomnilnik, ne da bi sprejemnik prej obvestil druge procese, da mu nehajo pošiljati sporočila.

### 6.2.1 Nadzor procesov

Vsakemu procesu se ob zagonu dodeli unikatna cela številka, s katero se sklicujemo na proces v drugih procesih, kot tudi v trenutnem procesu. Proces ima možnost končanja dela z ukazom `pvm_exit()`, ki pa ne konča izvajanja procesa na računalniku, ampak ga le odjavi iz sistema PVM. Ostali procesi o eksplicitnem izhodu procesa niso obveščeni in lahko dobijo stanje le preko informacijskih ukazov ali signalov.

Procesi lahko kreirajo nove procese z ukazom `pvm_spawn()`. Običajno nove procese kreira vodja (*master*), ki razdeli podatke in skrbi za zajem rezultatov. Odločitev, kje se poganjajo novi procesi je povsem v pristojnosti procesa. Možno je tudi zaganjanje procesov, kjer predpišemo arhitekturo, na kateri želimo, da se program izvede. Hkrati se lahko zažene več procesov. Ker proces na distribuiranih sistemih pomeni tudi program, se argupošiljanjem. Število odprtih povezav je na velikem številu sistemov zelo omejeno in zato TCP ni primeren za PVM, ker omejuje skalabilnost. Pošiljanje sporočil je zato izvedeno z UDP, PVM pa skrbi za ponovitve slabo poslanih sporočil, kot tudi za delitve velikih sporočil na velikosti paketov primernih za UDP.

Sporočila lahko združujejo večje število različnih tipov podatkov. Običajno predpišemo sporočilom tudi celoštevilčne oznake, da je možno v celotnem sistemu pravilno razpoznavati tip sporočil in jih tako pravilno sprejeti. Pri prenosu imamo dve glavni skupini rutin za prenos podatkov: pakiranje in razpakiranje. Prvo je potrebno pri pošiljanju, drugo pri sprejemu podatkov. Vmesni pomnilniki se kreirajo avtomatsko. Normalno imamo le en vmesni pomnilnik v katerega združujemo podatke, lahko pa naredimo tudi več vmesnih pomnilnikov, vendar je naloga programerja, da pravilno sestavlja sporočila. Priprava sporočila za pošiljanje ima sledeče potrebne korake:

1. inicializacija vmesnega pomnilnika z `pvm_initsend()`
2. pakiranje lokalnih podatkov v vmesni pomnilnik
3. zaključek sporočila s pošiljanjem na cilj in specifikacijo številke sporočila, ki unikatno označi tip sporočila

Pošiljanje ima običajno en sam ciljni proces. Obstaja pa tudi ukaz za hkratno posredovanje sporočila vsem procesom v navideznem računalniku (*multicasting*).

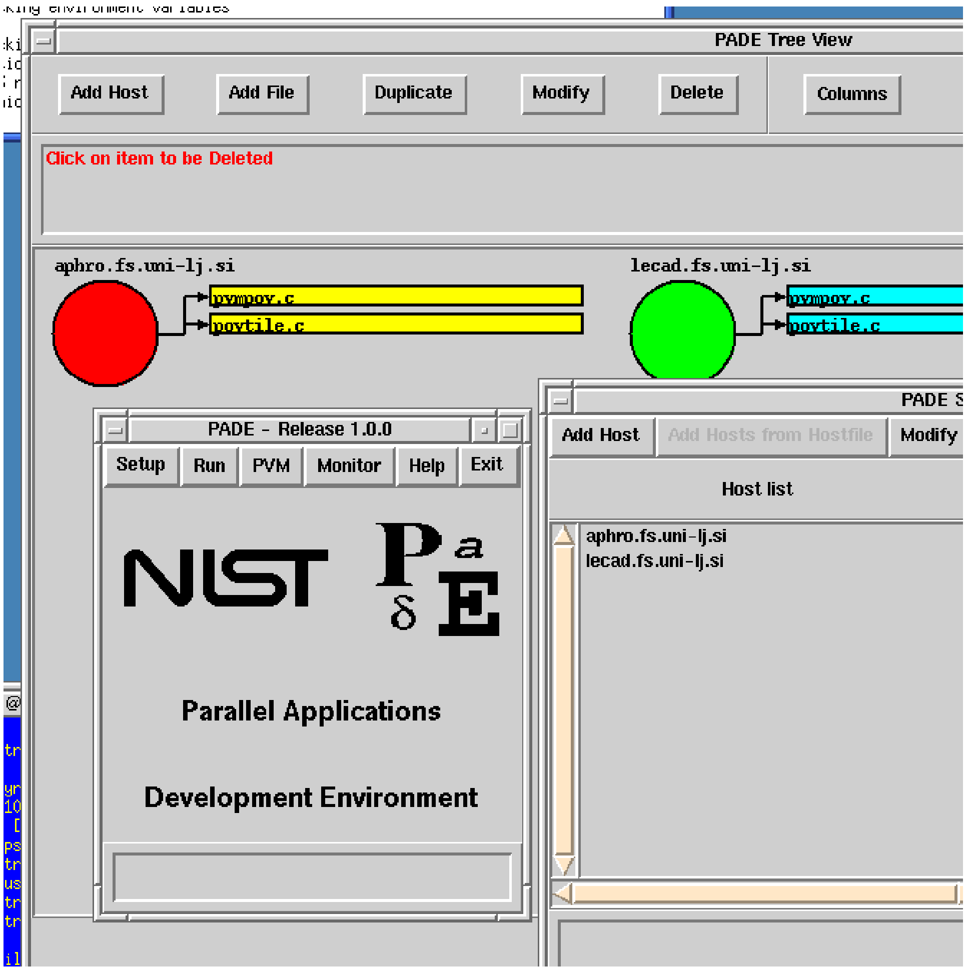
Sprejem sporočil je komplement pošiljanja. PVM omogoča tudi sprejem brez blokiranja procesa, vendar je tak način uporaben le za posebne primere. Ker lahko sporočila prihajajo nedeterministično, obstajajo tudi ukazi, ki testirajo tip sporočila, ki je prispelo. Razpakiranje sporočila je podobno opisanemu pri pakiranju in je vedno komplement pakiranja. Če temu ni tako, je razpakiranje neuspešno. Pri sprejemu, kjer proces čaka točno določeno sporočilo, je možno predpisati tudi maksimalni čas prispetja sporočila, kar je pomembna lastnost pri programih z odkrivanjem napake.

## 6.3 Orodja za razvoj paralelnih programov

Grafična predstavitev je pomembna lastnost vsakega uporabniškega vmesnika, saj lajša zaznavanje in interakcijo uporabnika z računalnikom. Za PVM obstaja nekaj orodij, s katerimi lahko opazujemo delovanje navideznega računalnika. Osnovni program pvm je interpreter ukazov (konzola), s katerim nadzorujemo delovanje sistema PVM. Z njim dodajamo in brišemo nove računalnike, zaganjamo in nadzorujemo procese. Omogoča tudi enostavno sledenje sporočilom vseh procesov in s tem tudi primitivno razhroščevanje paralelnih programov.

### 6.3.1 XPVM

Na računalnikih z operacijskim sistemom Unix je standardno grafično okolje sistem X-window. XPVM [GBD+94] je grafični program, ki ima vse lastnosti konzole *pvm*. Omostribuiranem okolju. Večino časa se pri razvoju porabi pri distribuciji izvorne kode po omrežju, testiranju na posameznih arhitekutrah in razhroščevanju, ki je v paralelnih sistemih bolj zapleteno. Na sliki 6.3 so prikazana nekatera okna, ki kažejo lastnosti okolja, ki omenjene težave lajša z grafičnim vmesnikom in ustreznimi programi.




Slika 6.3: PADE - Integrirano okolje za razvoj paralelnih aplikacij

## 6.4 Povzetek

Sistem PVM (*Parallel Virtual Machine*) je dobra osnova za pisanje paralelnih programov v distribuiranih sistemih, kot tudi v masivno paralelnih sistemih (MPP). PVM je knjižnica podprogramov za jezike C, C++ in Fortran, ki skupaj z orodji za nadzor, predstavljajo učinkovito okolje za distribuirane sisteme. Poleg potrebnega programskega vmesnika obstajajo tudi programi za grafično sledenje paralelnim programom, kot tudi programi za lažji razvoj na različnih arhitekturah.

- matrika = matrika + matrika
- matrika = matrika * matrika

Poleg omenjenih operatorjev so na voljo tudi konstruktorji za vse potrebne primitive, ki jih srečamo v računalniški grafiki, za dve, tri in štiridimenzionalne prostore.

### 7.1.1 Vhodni podatki

Interpreter datoteke z opisom scene je bil apliciran za format NFF[Hai88]. Neutral File Format je specifikacija za opis scene, ki se klasično uporablja za testiranje hitrosti algoritmov z metodo sledenja žarku. Obstaja v proceduralni obliki in poleg NFF generira tudi druge formate za opis scene.

Interpreter datoteke vhodnih podatkov je bil generiran v jeziku YACC[LMB92]. Skupaj s programom Flex je tako možno enostavno izdelati izvorno kodo v jeziku C++ za zelo zahtevne sintakse od katerih lahko NFF prištevamo k enostavnim jezikom.

Slovar jezika YACC podamo s pravili LARL (*LookAhead Left Recursive*). Pravila LARL so poenostavljena oblika pravil BNF (*Backus-Naur Form*), ki so normalno uporabljena za predstavitev sintakse programskih jezikov.

Razpoznavanje ključnih besed in ostalih elementov opisne datoteke je izvedeno s programom Flex, ki za razliko od ostalih programov tipa Lex (*Lexical analyzer*) zna generirati programsko kodo v C++.

LARL slovar datoteke NFF je sledeč:

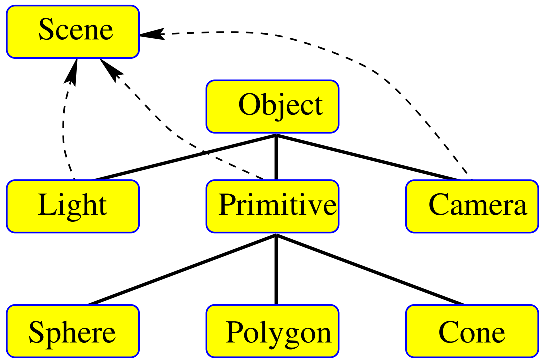
```
scene:  camera elementlist;

elementlist: elementlist element
           | /* prazen */ ;

element:  light
        | background
        | surface
        | object  ;

object:   cone
        | sphere
        | polygon
        | ppatch ;

camera: VIEWPOINT
        FROM point
        AT point
        UP point
        ANGLE num
        HITHER num
        RESOLUTION num num;
```




Slika 7.1: Zgradba programa za sledenje žarku

**Camera** implementira enostaven sistem gledanja, kot je opisan v podpoglavju 4.1. Parametri kamere ustrezajo specifikaciji NFF. Možna je tudi uvedba drugih parametrov gledanja, saj je implementacija dovolj splošna.

**Light.** Svetlobni izvor točkaste oblike. Jakost izvora je odvisna od celotnega števila luči v sceni. Vsi svetlobni izvori imajo enako jakost, ki se normira ob začetku določanja barve žarkov v posameznih točkah. Normiranje je potrebno zaradi pravilne določitve območja barv v modelu RGB (0..255). Svetlobne izvore je možno tudi razširiti na druge tipe kot sta reflektorska luč ali svetlobni snopi. Uvedba omenjenih izvorov svetlobe v program za sledenje žarku ne bi povečala računske zahtevnosti, saj se taka izvora lahko še vedno obravnava kot točkasta svetlobna izvora. Problem pri metodi sledenja žarku lahko predstavljajo le linijski in površinski svetlobni izvori, saj le ti mečejo drugačne sence od točkastih svetlobnih izvorov.

**Primitive.** Abstraktni razred primitivov združuje lastnosti objekta z lastnostmi površine (barva, difuzijska in odbojna sposobnost, prepustnost in lomni količnik). Razred *Primitive* je osnova za izpeljane primitive, ki jih vstavljamo v bazo podatkov. Določanje preseka in normale v izbrani točki sta edini potrebni virtualni metodi, ki ju mora izpeljani primitiv implementirati. Specifikacija NFF predvideva sledeče primitive:

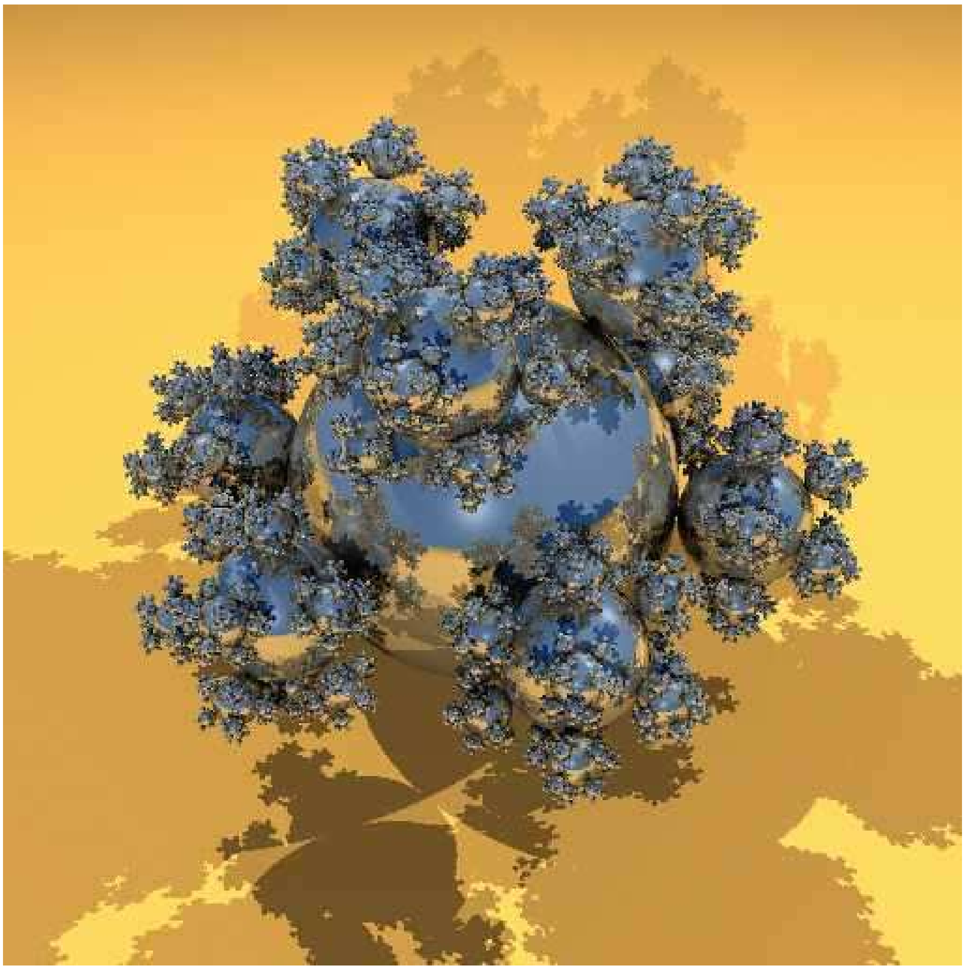
**Sphere.** Kot najenostavnejši primitiv za katerega je potrebno določiti presečno točko in normalo. Krogla je primitiv, ki ga najdemo prav v vseh programih za upodabljanje.

**Polygon.** Mnogokotnik je ravnina v prostoru omejena z množico vozliščnih točk na površini. Smer normale je podana s položajem prvih treh točk mnogokotnika, ki pa ne smejo biti kolinearne. Določanje presečne točke žarka in površine ne predstavlja velikega problema (glej stran 45). Hitrost določanja notranjosti točke je odvisna predvsem od števila vozlišč in konkavnosti mnogokotnikov [Hai94].

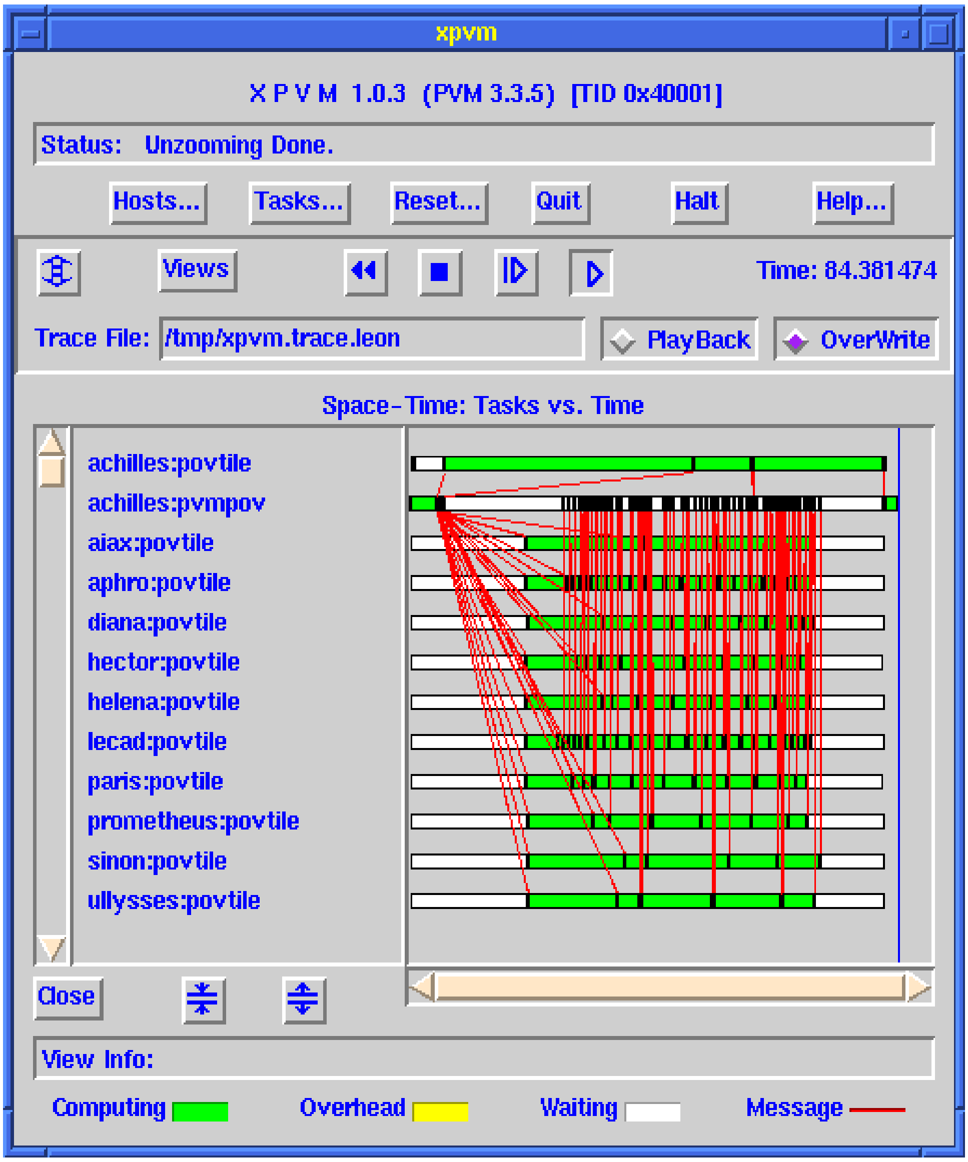
## 7.2 Distribuirani sistem

Distribuirani sistem, ki implementira paralelni program sledenja žarku, je bil narejen na osnovi sistema PVM. Algoritem, ki uravnava obremenitev računalnikov je bil izbran iz skupine s programskim modelom *vodja – delavci*. Osnovni pristop je kreiranje množice nalog, katere vodja dodeljuje delavcem. Ker se podatki po omrežju prenašajo s paketnim protokolom UDP/IP, je bilo smiselno izbrati tako shemo, ki bo zagotavljala, da bo ena naloga procesorja, ko vrne rezultat v mejah velikost enega paketa – *datagrama*. Zagotovljena maksimalna velikost UDP paketka je 8K bytov. Naloge delavcem glavni program `pvmpov` vedno dodeljuje tako, da sistemu PVM ni potrebno deliti na manjše enote. Izbrana je bila shema delitve nalog po vrsticah rastrskega zaslona. Velikost enega pasu je bila omejena do največ 16 vrstic naenkrat.

Na sliki 7.3 prikazan model z 7381 kroglami. Uporabljen je bil v hitrostnih testih distribuiranega sistema.



Slika 7.3: Testna slika velikosti $512 \times 512$ točk z 7381 kroglami




Slika 7.5: Pogled na časovno razporeditev pošiljanja sporočil

```
Server statistics:
NUM                      HOST     DTID      TID     ARCH TILES [ % ]

  0                  achilles    40000    40004    LINUX     4   3.9
  1          lecad.fs.uni-lj.si    80000    80001     SGI5    21  20.4
  2          aphro.fs.uni-lj.si    c0000    c0001     HPPA    22  21.4
  3          paris.fs.uni-lj.si   100000   100001     SGI5    11  10.7
  4         hector.fs.uni-lj.si   140000   140001     SGI5    11  10.7
  5           aiax.fs.uni-lj.si   180000   180001    LINUX     4   3.9
  6          diana.fs.uni-lj.si   1c0000   1c0001     HPPA     8   7.8
  7         helena.fs.uni-lj.si   200000   200001     HPPA     8   7.8
  8     prometheus.fs.uni-lj.si   240000   240001    LINUX     6   5.8
  9          sinon.fs.uni-lj.si   280000   280001    LINUX     4   3.9
 10       ullysses.fs.uni-lj.si   2c0000   2c0001    LINUX     4   3.9

1.97user 1.44system 0:57.04elapsed 5%CPU (0avgtext+0avgdata 0maxres)k
0inputs+0outputs (0major+0minor)pagefaults 0swaps
```

Tabela 7.1: Hitrostni test v maksimalni konfiguraciji

```
Server statistics:
NUM                      HOST     DTID      TID     ARCH TILES [ % ]

  0                  achilles    40000    40007    LINUX    16  15.5
  1           aiax.fs.uni-lj.si   180000   180002    LINUX    21  20.4
  2     prometheus.fs.uni-lj.si   240000   240002    LINUX    28  27.2
  3          sinon.fs.uni-lj.si   280000   280002    LINUX    19  18.4
  4       ullysses.fs.uni-lj.si   2c0000   2c0002    LINUX    19  18.4

1.70user 1.95system 3:01.33elapsed 2%CPU (0avgtext+0avgdata 0maxres)k
0inputs+0outputs (0major+0minor)pagefaults 0swaps
```

Tabela 7.2: Hitrostni test na arhitekturi Linux (PC-Unix) za oceno v tej nalogi:

- Interpretirane scene v spominu so običajno manjše, kot pa datoteke z opisom scene. Z predprocesiranjem in ustrezno shemo, ki bi omogočala prenos povezanega opisa objektov na različne arhitekture.

- Distribuirani sistem na osnovi PVM je možno implementirati tudi tako, da je možno nove vozle dinamično vključevati in brisati iz konfiguracije sistema.

- Uravnoteženi sistem je mogoče izpopolniti na način, ki predvideva čas potreben za zaključek dela in s tem zmanjša zaključni čas.

- V sistemih za animacijo bi bilo ugodno zmanjšati čas predprocesiranja z vnaprejšnjim pošiljanjem scen in delitev na podprocese.

- Obstajajo mnoga druga področja aplikacij, v katerih bi bilo vredno uporabiti v distribuirane sisteme. Predstavljeno delo dokazuje, da to niti ni tako težko.

- Poleg PVM obstajajo tudi druge knjižnice za distribuirane sisteme z drugačnimi pristopi k reševanju problemov. Malo je bilo narejenega, da bi dobili dobre primerjave med njimi.

- Kot je razvidno iz poglavja 4, kjer je predstavljena metoda sledenja žarku, bi lahko pri praktični izvedbi uporabil mnoge algoritme in s tem povečal uporabnost osnovnega programa sledenja žarku.

Če pogledam nazaj, je magistrska naloga uspešno dosegla zastavljene cilje. Uspešno je bil implementiran distribuiran sistem na programu za sledenje žarku. Prikazan je bil razvoj in diskusija metod v upodabljanju tridimenzionalnih objektov. Za tiste, ki bi želeli implementirati podobne sisteme, so zadovoljivo prikazani detajli potrebni za uspešno implementacijo. V delu je prikazano: pregled upodabljanja z kazalci na ustrezno literatutro, diskutirani so distribuirani sistemi, navodila za paralelizacijo sekvenčnih algoritmov in pregled problemov na katere lahko naletimo pri implementaciji. Naj zaključim z besedami Paula Heckberta: "Če bi imeli več časa, bi lahko naredili vse".

[CS94] Keneth Chiu in Peter Shirley. Rendering, Complexity, and Perception. Tehnično poročilo, Indiana University, Bloomington IN 47405, USA, 1994.

[CW93] Michael F. Cohen in John R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press, 1993.

[dB93] Mark de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal*. Številka 703 v Lecture Notes in Computer Science. Springer Verlag, 1993.

[Dou94] Jean-Francois Doue. C++ Vector and Matrix Algebra Routines. V Heckbert [Hec94].

[Duc88] David Duce, urednik. *Proceedings of the Eurographics Conference*. Elsevier Science, September 1988.

[Fos94] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1994.

[Fre88] John R. Freer. *Computer Communications and Networks*. Pitman publishing, London UK, 1988.

[FvD90] James D. Foley in Andries van Dam. *Fundamentals of Interactive Computer Graphics*. Systems Programming Series. Addison Wesley, druga izdaja, 1990.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek in Vaidy Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, 1994.

[Gla89] Andrew S. Glassner. *3D Computer Graphics*. The Herbert Press, 1989.

[Gla91a] Andrew S. Glassner, urednik. *Graphics Gems I*, Zbornik 1 v *Graphics Gems Series*. Academic Press, Inc., 1991.

[Gla91b] Andrew S. Glassner, urednik. *An introduction to Ray Tracing*. Academic Press, 1991.

[Gla91c] Andrew S. Glassner. *Surface Physics for Ray Tracing*, Poglavje 4, strani 121–160. V knjigi Introduction to Ray Tracing, glej [Gla91b], 1991.

[Gue90] Robert D. Guenther. *Modern Optics*. John Willey & Sons, Inc., 1990.

[Hai88] Eric A. Haines. Neutral File Format. *Ray Tracing News*, 1, Oktober 1988. electronic journal.

[Hai94] Eric Haines. Point in Polygon Strategies. V Paul Heckbert, urednik, *Graphics Gems IV*, strani 24–46. Academic Press, Boston, 1994.

[Han91] Pat Hanrahan. *A Survey of Ray-Surface Intersection Algorithms*, Poglavje 4, strani 121–160. V knjigi Introduction to Ray Tracing, glej [Gla91b], 1991.

[Pal85] Edvard D. Palik. *Handbook of Optical Constants of Solids*. Academic Press, New York, 1985.

[Pho75] Bui-Tuong Phong. Illumination for computer-generated pictures. *Comm. ACM*, 8(18):311–317, 1975.

[PP93] S.J. Pedroti, Frank L. in Leno S. Pedroti. *Introduction to Optics*. Prentice-Hall, Inc., druga izdaja, 1993.

[PTVF92] W. Press, S. Teukolski, W. Vetterling in B. Flannery. *Numerical Recipes in C*. The Art of Scientific Computing. Cambridge University Press, Cambridge, druga izdaja, 1992.

[Rag93] Stephen A. Rago. *Unix System V Network Programming*. Addison Wesley, 1993.

[Rim90] Steve Rimmer. *Bit-Mapped graphics*. McGraw-Hill, Inc., 1990.

[Sea56] Francis Weston Sears. *Optics*. Addison Wesley, tretja izdaja, 1956.

[Shi90] Peter Shirley. Physically Based Lighting Calculations for Computer Graphics: a Modern Perspective. V *Proceedings Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*, strani 67–81, Rennes, France, Junij 1990.

[SHM94] John Shirley, Wei Hu in David Magid. *Guide to Writing DCE Applications*. O'Reilly & Associates, Inc., druga izdaja, 1994.

[Sil93] Peter P. Silvester. *Data Structures for Engineering Software*. Computational Mechanics Publicatinas, 1993.

[Sph94] Gary Sphere. Balancing distributed computing. *Information Week*, stran 58, Februar 1994.

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wessley, 1992.

[Str91] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, druga izdaja, 1991.

[Sub90] Prem Subrahmanyam. Intersecting Quadrics. *Ray Tracing News*, 3(2), Marec 1990. electronic journal.

[Swa90] Ray Swartz. *UNIX Applications Programming*. SAMS, 1990.

[Ter93] Mark A. Terrible. *Practical C++*. UNIX/C. McGraw-Hill, Inc., 1993.

```
 int Shadow( vec3& rayDir, double tMax);
 int Trace(int depth, double weight, vec3& rayOrg,
     RGBColor& color);
public:
  Scene();
  ~ Scene();
 void Describe();
 unsigned char* Screen ();
  vec2 getResolution() { return camera.getRes(); }
 friend int yyparse(void);
};
```

## Metode

**Scene::Scene**()

Konstruktor razreda za opis scene. Nastavi začetno barvo ozadja, utež pomembnosti žarka, velikost zaokrožitvene napake in največjo globino rekurzivnega sledenja žarku.

**Scene::~Scene**()

Destruktor razreda zbriše vse alocirane objekte (primitivi, luči).

`unsigned char*` **Scene**`::Screen ()`

Ta metoda generira primarne žarke, ki se rekurzivno odbijajo in lomijo do predpisanega maksimalnega števila. Rezultat se zapisuje v linearni RGB pomnilnik, ki ga po končanem sledenju metoda tudi vrne v obliki vektorja nepredznačenih trojic bytov RGB. Uporabnik mora po uporabi sam sprostiti vrnjeni vektor.

`int` **Scene**`::Shadow(`**vec3**`& rayDir, double tMax)`

Testira obstoj senčnega žarka iz *rayOrg* v smeri *rayDir*. Če svetloba ni blokirana, vrne 1, sicer je žarek v senci in zato vrne 0. Pri ugotavljanju senčnih žarkov je potrebno za izbrano smer preveriti prav vse objekte, ali obstaja presek. Ker se lahko luč nahaja tudi v notranjosti objektov ali pa obstajajo objekti tudi za lučjo, se z argumentom *tMax* omeji največja dolžina testiranega žarka. *tMax* torej predstavlja razdaljo od *rayOrg* do testne luči v smeri *rayDir*.

`int` **Scene**`::Trace(int depth, double weight,` **vec3**`& rayOrg,` **RGBColor**`& color)`

Osnovna rekurzivna rutina, ki sledi žarku od izhodišča *rayOrg* v smeri *rayDir*. Maksimalno število odbojev in prehodov se kontrolira z *depth*. Akumulirani prispevek sekundarnih žarkov je podan z *weight*. Izračunana barva žarka se vrne z vektorjem *color*. Metoda združuje senčenje točke po Phongovi metodi in generiranje rekurzivnih žarkov za odbito in prepuščeno svetlobo.

```
typedef Object* ObjectPtr;
```

Definicija kazalca na **Object**, ki je potrebna za zbirko (*SLBag*).

## A.3 Camera

Implementacija enostavnega sistema gledanja, ki ustreza parametrom podanim v specifikaciji NFF.

### Definicija

```
class Camera: public Object
{
protected:
   vec2 res;           // ločljivost filma
   vec3  u, v;          // smer osi u in v
   vec3  dop;           // smer v katero gledamo
   vec2  uvMax;         // maxsimalne koordinate filma
  double h;             // oddaljenost filma od gorišča

public:

  void setup(int resx, int resy,
      double from[3], double at[3], double up[3],
      double angleU, double angleV, double hither );
   vec2 getRes() {return res;}
  void Describe(int indent);
   vec2& p);
};
```

### Metode

```
void Camera::setup(int resx, int resy, double from[3]
        double at[3], double up[3], double angleU,
        double angleV, double hither )
```

Nastavi parametre kamere. Ločljivost filma je podana z *resx* in *resy*. Položaj kamere je podan z vektorjem $\vec{from}$. Točka v katero je kamera usmerjena je podana z vektorjem $\vec{at}$. Zavrtitev kamere okoli osi leč je podana z vektorjem $\vec{up}$, ki ni nujno normiran in ni potrebno, da je pravokoten na os leč (from – at). Vektor *up* podaja smer koordinate *y* (*v*) na filmu. Kot zaznavanja kamere v smeri koordinat filma *u* in *v* se določa s spremenljivkama *angleU* in *angleV*. Oddaljenost filma od izhodišča kamere je določena s *hither*. Ker pa je `typedef` **Light**`*` **LightPtr**`;`

Definicija kazalca na **Light**, ki je potrebna pri zbirki (*SLBag*)

## A.5 Primitive

Osnova vseh primitivov, ki jih uporabljamo za opis **Scene**. **Primitive** je abstraktni razred.

### Podrazredi

**Sphere**, **Polygon** in **Cone**

### Definicija

```
class Primitive: public Object
{
protected:
   RGBColor  color;                // barva primitiva
   double    surfaceProperty[5];   // koeficienti površine

public:

   RGBColor getColor() { return color; };
   void setColor(double color[3]);
   double   getProp(int i) { return surfaceProperty[i]; }
   void setSurfaceProperties(double surfaceProperty[5]);
   double closestIntersection(const double *x, int xNum) const;
   virtual ~ Primitive();
   virtual  double nearestIntersection(vec3& rayDir) const = 0;
   virtual void Describe(int indent) = 0;
   virtual  vec3& p) const = 0;

};
```

### Metode

**Primitive**`::`~**Primitive**`()`

Destruktor gradnika scene

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

Nastavi barvo primitiva

**Primitive**`::setSurfaceProperties(double surfaceProperty[5])`

Nastavi optične lastnosti površine

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

Izpiše lastnosti krogle

## A.7 Polygon

Mnogokotnik sestavljajo vozlišča *vList*. Prva tri vozlišča ne smejo biti kolinearna, saj definirajo orientacijo in normalo mnogokotnika.

### Definicija

```
class Polygon : public Primitive
{
protected:
  vec3 *vList;      // seznam vozlišč mnogokotnika
 int vertexN;     // število vozlišč

public:

  Polygon(Vec *vList, int vertices, double color[3],
          double prop[5]);
  ~ Polygon();
   virtual double nearestIntersection( vec3& rayDir) const;
   virtual  vec3& p) const;
   virtual void Describe(int indent);
};
```

### Metode

**Polygon::Polygon**`(Vec *vList, int vertices, double color[3], double prop[5])`

Konstruktor mnogokotnika s seznamom vozlišč *vList*. Število vozlišč podaja spremenljivka *vertices*. Barva mnogokotnika je podana s *color*. Lastnosti površine podaja vektor *prop*.

**Polygon::~Polygon** `()`

Destruktor razreda, ki zbriše alocirani seznam vozlišč.

`double` **Polygon**`::nearestIntersection(`**vec3**`& rayDir)`

Poišče presečišče mnogokotnika z žarkom, ki je v izhodišču *rayOrg* in ima smer *rayDir*. Površino mnogokotnika definirajo prva tri vozlišča. Preseke žarka z mnogokotnikom se določa na osnovi trikotnikov. Zaradi tega ta metoda (*triangle fan*) pravilno deluje le za konveksne mnogokotnike. Za mnogokotnike z majhnim številom vozlišč, je to najhitrejša

```
vec3 & operator *= (const double d); //množenje s konstanto
vec3 & operator /= (const double d); //deljenje s konstanto
double &operator[] (int i);          //indeks

// Posebne funkcije

double length ();              //dolžina vec3
double length2 ();             //kvadrat dolžine vec3
vec3 & normalize ();           //normala vec3
vec3 & apply (V_FCT_PTR fct);  //apliciraj funkcijo na komponente

// Prijatelji

friend vec3 operator - (const vec3 & v);  //-v1
friend vec3 operator + (const vec3 & a, const vec3 & b);  //v1 + v2
friend vec3 operator - (const vec3 & a, const vec3 & b);  //v1 - v2
friend vec3 operator * (const vec3 & a, const double d);  //v1 * 3.0
friend vec3 operator * (const double d, const vec3 & a);  //3.0 * v1
friend vec3 operator * (const mat4 & a, const vec3 & v);  //M.v
friend vec3 operator * (const vec3 & v, mat4 & a);        //v.M
friend double operator *(const vec3 & a, const vec3 & b); //skalarni
friend vec3 operator / (const vec3 & a, const double d);  //v1 / 3.0
friend vec3 operator ^ (const vec3 & a, const vec3 & b);  //vekt. produkt
friend int operator == (const vec3 & a, const vec3 & b);  //v1 == v2 ?
friend int operator != (const vec3 & a, const vec3 & b);  //v1 != v2 ?
friend ostream & operator << (ostream & s, vec3 & v);     //izpis
friend istream & operator >> (istream & s, vec3 & v);     //branje
friend void swap (vec3 & a, vec3 & b);                    //zamenjaj v1 & v2
friend vec3 min (const vec3 & a, const vec3 & b);         //min (v1, v2)
friend vec3 max (const vec3 & a, const vec3 & b);         //max (v1, v2)
friend vec3 prod (const vec3 & a, const vec3 & b);        //množi po koord.

//potrebne deklaracije prijateljev

friend class vec2;
friend class vec4;
friend class mat3;
friend vec2 & v);
//linearna transformacija
friend mat3 & b);
//množenje z matriko
```

`double` **vec3**`::length ()`
Dolžina vektorja

`double` **vec3**`::length2 ()`
Kvadrat dolžine vektorja. Hitrejše kot **vec2**::length ()

**vec3** `&` **vec3**`::normalize ()`
Normalizacija vektorja

**vec3** `&` **vec3**`::apply (V_FCT_PTR fct)`
Apliciraj funkcijo *fct* na vektor

## B.1 Upravitelj

Računski moduli v sistemu PVM so izvršni programi, ki jih je potrebno prevesti iz izvorne kode za vsako arhitekturo. V ta namen je razvit program `aimk` (*Achitecture Indenpendent MaKe*), ki olajša izgradnjo programov za različne platforme. Za razliko od delavcev, pa nadzornik teče le na eni platformi, kar nekoliko zmanjša skrb za prenosljivost in olajša razvoj programa upravitelja. V nadaljevanju so predstavljeni bistveni deli programa nadzornika distribuiranega sistema, napisanega v jeziku C.

Osnovne globalne spremenljivke hranijo kazalce na dinamično dodeljena spominska polja, katerih velikost se določi pri zagonu programa, ko je znana željena velikost slike in vhodna datoteka.

```
static int verbose = 1;      /* Nivo sporočil upravitelja */
static int mytid;            /* ID številka upravitelja */
static int nserv;            /* število delavcev */
static int nhost;            /* število strežnikov hosts >= nserv */
static int *servtids = 0;    /* seznam ID strežnikov */
static int *servtiles = 0;   /* polje statistike pasov */
static int *servflist = 0;   /* seznam prostih delavcev */
static int *servjobs = 0;    /* dodeljene naloge */
static int servfree = -1;    /* prvi prosti delavec */
static int njobs;            /* skupno število nalog */
static char **image;         /* rezultat (slika) iz pasov */
static int jobstogo;         /* preostalo število nalog */
static int jobnext;          /* naslednja naloga, ki bo dodeljena */
static int height = 100;     /* višina slike */
static int width = 100;      /* širina slike */
static struct pvmhostinfo *hostp; /* informacije o strežnikih */
```

Če imena vhodne in izhodne datoteke niso podane se privzameta podani vrednosti. Vhodna datoteka je v formatu POV, izhodna v formatu PPM (P6 raw).

```
static char inputFileName[MAXFILENAME] = "scene.pov";
static char outputFileName[MAXFILENAME] = "scene.ppm";
```

### B.1.1 Glavni program

Glavni program interpretira ukazno vrstico. Argumente ustrezno filtrira in posreduje delavcem.

```
int main(argc, argv)
     int argc;
     char **argv;
{
  if (argc == 1)
    {
      usage ();
      return -1;
    }
```

```
  if (start_workers (argv) < 0)
    {
      pvm_exit();
      return 1;
    }
```

Datoteko z opisom prosotora je potrebno podati kot nalogo vsem delavcem. Datoteka, ki opisuje sceno je tekstovna in je zato lahko zelo velika in s tem tudi vzrok za ozko grlo pri komunikaciji med upravnikom in delavci. V ta namen je bilo izvedeno sprotno stiskanje podatkov, kar je omogočilo tudi do desetkrat manjše datoteke in s tem ustrezno krajše čase prenosa po mreži.

```
  send_scene(inputFileName);
```

Upodabljanje scene v kateri so podani vsi objekti se odvija v proceduri, ki skrbi za izmenjavo podatkov in združevanje rezultatov.

```
  render_scene ();
```

Po končanem delu je potrebno ustaviti vse delavce navideznega računalnika in zapreti pripadajoče komunikacijske kanale.

```
  stop_workers ();
```

Rezultat distribuiranega programa za upodabljanje z metodo sledenja žarku je polje barvnih trojčkov RGB, ki jih je potrebno shraniti v datoteko željenega grafičnega formata. Format PPM je bil izbran zaradi razširjenosti in primernosti aplikaciji.

```
  save_scene();
```

Poleg klasičnega ukaza `exit()` je v sistemu PVM potrebno zapreti vse uporabljene vire navideznega računalnika.

```
  return pvm_exit();
}
```

### B.1.2 Določitev in zagon delavcev

```
static int
start_workers (argv)
char **argv;
{
  int info, i;
  int n;          /* število delavcev, ki so usposobljeni za delo */
  int narch;      /* število različnih podatkovnih formatov (little/big endian) */
  int numt;       /* celotno čtevilo nalog, ki jih je potrebno obdelati */
```

Podatke o ustroju navideznega računalnika dobimo s klicem konfiguracijske funkcije, ki vrne število strežnikov, število različnih arhitektur in podrobne podatke o posameznih računalnikih, ki so trenutno prijavljeni v sistemu.

Po kontroli izpišemo potrebne podatke o računalnikih, ki se niso odzvali na poziv k sodelovanju.

```
  for ( i = n = 0; i < nhost; i++)
    {
      if (servtids[i] < 0)
        {
          fprintf(stderr, "can't initiate worker '%s'"
   "TID: %d ERROR: %d %s\n",
                  CHILDNAME, i, servtids[i], pvm_errlist[-servtids[i]]);
        }
      else
        {
          servtids[n] = servtids[i];
          servflist[n]  = servfree;
          servfree = n++;
        }
    }

  if (n != nhost)
    {
      if ((nserv = n))
        fprintf(stderr, "using only %d workers\n", n);
      else
        {
          fprintf(stderr, "ack, no workers\n");
          return -1;
        }
    }
  else
    nserv = nhost;

  return 0;
}
```

### B.1.3 Razpošiljanje naloge delavcem

Upravnik ima možnost komprimiranega ali nekomprimiranega razpošiljanja podatkov scene. Naloga vseh delavcev je skupna, le da nadzornik dodeljuje pozamezne podnaloge.

```
static int
send_scene(file_name)
char *file_name;
{
  FILE *f;
  static char *buffer = 0;
  static long length = 0;
  long position = 0;
  long file_size;
  int info;

  length = 10 * SCENEBUFSIZE * sizeof(char);
  buffer = (char *)malloc(length);
```

```
  if (verbose)
    fprintf(stdout, "Multicasting scene '%s' size = %ld\n",
            file_name, file_size);
```

Razpošiljanje sporočil med vozlišči navideznega računalnika je normalno izvedeno z enim od ukazov iz skupine *send*. Za razpošiljanje enakih sporočil vsem vozliščm pa je bolje uporabiti ukaz za vzporedno distribucijo sporočil. Vsako pošiljanje sporočil se začne z inicializacijo razpošiljanja, ki inicializira vmesni pomnilnik in kodiranje sporočil. Sporočilo je lahko sestavljeno iz več različnih podatkov, ki so zlepljeni in poslani naslovniku. Pri pošiljanju datoteke scene se tako pošlje ime datoteke, velikost in datoteka, kot eno samo sporočilo. Vsako sporočilo je opremljeno tudi s številko, da je možno enolično razpoznavati sporočila. Sistem PVM sicer skrbi za sinhronizacijo sporočil v celotnem sistemu, vendar se lahko zaporedje, ki ga je programer predvidel, podre pri nepravilnem delovanju enega od vozlišč. V takem primeru pa pride oznaka sporočila še kako prav.

```
  pvm_initsend( PvmDataDefault );
  pvm_pkstr(file_name);              /* Pošlji datoteko */
  pvm_pklong(&file_size, 1, 1);
  if (pvm_pkbyte(buffer, file_size, 1) < 0)
    pvm_perror("packing scene data");
  info = pvm_mcast(servtids, nserv, MSGSCENE);

  if (info < 0)
    pvm_perror("multicasting scene file");

  return 0;
}
```

### B.1.4 Upodabljanje

Dodeljevanje dela in nadzor delavcev je sestavljeno iz več podprogramov. Zaradi razlike med procesorskimi močmi posameznih vozlišč je bilo potrebno uvesti algoritem, ki učinkovito uravnava delo med delavci. Algoritem je zasnovan na dejstvu, da lahko celoten problem (rastrsko sliko) razdelimo na množico neodvisnih podproblemov, ki so jih nekateri delavci sposobni rešiti hitreje od ostalih. Če je teh nalog več, potem hitrejši delavci opravijo več nalog kot drugi. Vsi delavci pa so pri tem polno zaposleni.

Zaradi omejenega števila odprtih povezav se v sistemu PVM raje uporablja paketni prenos sporočil s protokolom UDP/IP. Velikost enega paketa je omejena, zato je ugodno, če določimo velikost sporočil v obsegu enega paketa, saj tako sistemu PVM ni potrebno skrbeti za razbijanje sporočil na več delov. Največja sporočila, ki se bodo vračala kot rezultat delavcev, bodo izračunane vrstice, katerih velikost je začetno ocenjena glede na normalno širino slike.

```
static int
render_scene ()
{
  int tile_height = 5;  /* Začetna velikost enega pasu */
```

```
        printf("%3d %28s %8x %8x %8s %5d %5.1f\n",
               i, hostp[s].hi_name, dtid, servtids[i],
               hostp[s].hi_arch, servtiles[i],
               (float)servtiles[i]/(float)njobs*100.0);

      }
    }
  return 0;
}
```

### Dodeljevanje naslednje naloge

Podprogram send_nextjob skrbi za alokacijo prostega računalnika, ki mu dodeli naslednjo nalogo iz množice neobdelanih nalog.

```
static int
send_nextjob ()
{
  int y1, y2;
  y2 = jobnext-- * height;  /* Pozor enaka prioriteta operatorjev */
  y2 /= njobs;              /* za množnje in deljenje */
  y1 = jobnext * height;    /* y1 <= y && y < y2 */
  y1 /= njobs;              /* y1 je v pasu, toda y2 NI */

  servjobs[servfree] = jobnext;
```

Sporočilo prostemu delavcu je sestavljeno iz številke naloge, začetne in končne vrstice v rastrski sliki.

```
  pvm_initsend(PvmDataDefault);
  pvm_pkint(&jobnext, 1, 1);
  pvm_pkint(&y1, 1, 1);
  pvm_pkint(&y2, 1, 1);
  if (pvm_send(servtids[servfree], MSGTILE))
    {
      pvm_perror("sending to tile server");
      return -1;
    }
  if (verbose)
    {
      printf("[%d@%d:%d-%d]", jobnext, servfree, y1, y2 - 1);
      fflush(stdout);
    }

#ifdef DEBUG
  printf("sent job %d to serv %d: y2 = %d, y1 = %d\n",
         jobnext, servfree, y2, y1);
#endif

  servfree = servflist[servfree];
  return 0;
}
```

### B.1.5 Ustavljanje delavcev

Ko končamo z delom, je potrebno pravilno odsloviti vse delavce in sprostiti zasedeni pomnilnik.

```
static int stop_workers()
{
  if (servtids)
    {
      while (nserv-- > 0)
        {
          pvm_kill(servtids[nserv]);
        }
      free(servtids);
      free(servflist);
      free(servjobs);
      free(servtiles);
    }
  servtids = 0;
  servflist = 0;
  servjobs = 0;
  servtiles = 0;
  return 0;
}
```

### B.1.6 Shranjevanje rezultatov

Ko je celotna naloga izračunana shranimo datoteko v formatu PPM.

```
static int
save_scene ()
{
  FILE *f;
  int i;

  if ((f = fopen (outputFileName, "w")) == NULL)
    {
      perror(outputFileName);
      return -1;
    }

  fprintf(f, "P6\n%d %d\n255\n", width, height); /* raw PPM format */
  for (i = jobnext = 0  ; i < njobs; i++)
    {
      int y1, y2;
      y1 = jobnext++ * height;  /*
      y1 /= njobs;              /*
      y2 = jobnext  * height;   /* y1 <= y && y < y2 */
      y2 /= njobs;              /* y1 is in tile, but y2 is NOT */

      fwrite(image[i], (y2 - y1) * width * 3, 1, f);
    }
```

```
  if (f == NULL)
    {
      pvm_perror("Writing scene file");
      pvm_exit();
      return -1;
    }

  fwrite(buffer, file_size, 1, f);

  fflush(f);

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

#ifdef DEBUG
  fputs("Scene file successfuly received, waiting ...\n", stderr);
#endif
```

Klic glavnega programa, ki interpretira vhodno datoteko in se pripravi na izvrševanje nalog.

```
  alt_main(argc, argv);

  width = Last_Column - First_Column;

  while (pvm_recv(mastertid, MSGTILE) > 0)
    {
      int job, y1, y2;

      pvm_unpackf("%d %d %d", &job, &y1, &y2);
#ifdef DEBUG
      fprintf(stderr, "Received job #%d (%d-%d)\n", job, y1, y2);
#endif
      calc_tile(y1, y2);

#ifdef DEBUG
      fprintf(stderr, "tile #%d calculated (%d-%d)\n", job, y1, y2);
#endif
```

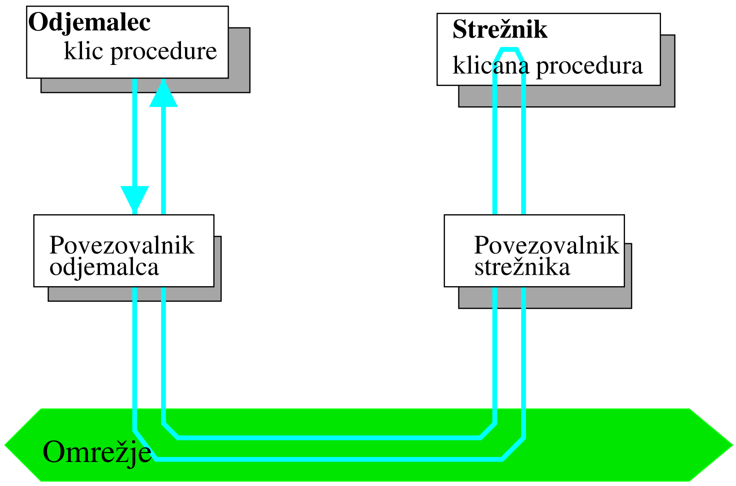
Zapakiramo podatke in pošljemo rezultat upravitelju.

```
      pvm_initsend(PvmDataDefault);
      pvm_pkint(&job, 1, 1);
      pvm_pkint(&y1, 1, 1);
      pvm_pkint(&y2, 1, 1);
      pvm_pkbyte(PixelMap, (y2 - y1) * width * 3, 1);
      if (pvm_send(mastertid, MSGDATA))
        {
          fprintf(stderr, "error sending image back\n");
        }
      free (PixelMap);
      PixelMapSize = 0;
#ifdef DEBUG
      fprintf(stderr, "result #%d (%d-%d) sended back\n", job, y1, y2);
#endif
```

```
[t200005] Parsing...
[t240005] Parsing...
[t180005] Parsing...
[t2c0005] Parsing...
[t280005] Parsing...
[t80005]
[tc0005]
[t100005]
[t140005]
[t80005]
[t80005] Preprocessing...
[tc0005]
[tc0005] Preprocessing...
[t240005]
[t200005]
[t100005]
[t100005] Preprocessing...
[t140005]
[t140005] Preprocessing...
[t1c0005]
[t4000c] <93>[91@1:452-456]
[t4000c] <94>[90@2:447-451]
[t180005]
[t4000c] <91>[89@1:442-446]
[t2c0005]
[t280005]
[t4000c] <90>[88@2:437-441]
[t4000d]
[t4000c] <89>[87@1:432-436]
[t4000c] <88>[86@2:427-431]
[t4000c] <95>[85@3:422-426]

...
...

[t4000d] EOF
[t4000c] <3><5><10><4><13><2><16><17><0><1><11>
[t4000c] Server statistics:
[t4000c] NUM                   HOST   DTID    TID   ARCH TILES [%]
[t4000c]
[t4000c]  0               achilles  40000  4000d  LINUX   2   1.9
[t4000c]  1      lecad.fs.uni-lj.si  80000  80005   SGI5  21  20.4
[t4000c]  2      aphro.fs.uni-lj.si  c0000  c0005   HPPA  22  21.4
[t4000c]  3      paris.fs.uni-lj.si 100000 100005   SGI5  12  11.7
[t4000c]  4     hector.fs.uni-lj.si 140000 140005   SGI5  12  11.7
[t4000c]  5       aiax.fs.uni-lj.si 180000 180005  LINUX   4   3.9
[t4000c]  6      diana.fs.uni-lj.si 1c0000 1c0005   HPPA   8   7.8
[t4000c]  7     helena.fs.uni-lj.si 200000 200005   HPPA   8   7.8
[t4000c]  8 prometheus.fs.uni-lj.si 240000 240005  LINUX   6   5.8
[t4000c]  9      sinon.fs.uni-lj.si 280000 280005  LINUX   4   3.9
[t4000c] 10   ullysses.fs.uni-lj.si 2c0000 2c0005  LINUX   4   3.9
[t4000c] EOF
```




Slika C.1: Klicanje oddaljenih procedur

format tako, da direktno komunicira s transportnim medijem.

## C.2 Načini komunikacije RPC

Narava problema pogojuje tudi način komunikacije med strežnikom in odjemalcem. Najpomembnejša razlika, ki jo diktira obremenjenost strežnika, je razlika med *sinhronim* in *asinhronim* dostopom do povratnih informacij, ki jih dobi odjemalec kot rezultat od strežnika.

Pri sinhroni komunikaciji odjemalec sporoči strežniku zahtevo za oddaljeno proceduro; strežnik nalogo izvrši in vrne rezultate. Ves čas obdelave na strežniku je odjemalec blokiran in čaka rezultate. Na obeh računalnikih je pri tem odprtega komunikacijskega kanala. Če je čas obdelave daljši, je večja tudi možnost prekinitve povezave, zato se pri daljših nalogah na serverju največkrat uporabi *asinhrona* komunikacija, ki nima stalno odprt komunikacijski kanal.

Prednost asinhrone komunikacije je prav v tem, da lahko odjemalec po tem, ko je dal ukaz za operacijo na serverju, nadaljuje z delom na drugih nalogah, ki so lahko tudi nadzor ali statistika dela posameznega strežnika. Časovno zahtevni procesi na strežniku so lahko izvedeni tudi tako, da strežnik ne vrača rezultatov, ampak samo izvršuje ukaze. Tak način dela je evidentno enosmeren in ravno tako sprošča komunikacijski kanal. Možna je tudi enosmerna izvedba z vračanjem rezultatov samo takrat, ko je to potrebno; npr. pri sporočilih o napakah pri obdelavi.

**rwall** Pošiljanje sporočila vsem uporabnikom na oddaljenem računalniku

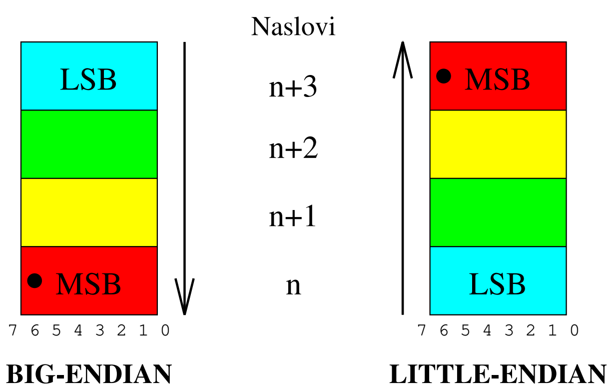
**Srednji nivo** skriva pred uporabnikom značilnosti omrežja. Tako uporabniku ni potrebno skrbeti za vtičnice in način prenosa. Funkcije kot so `registerrpc`, `callrpc` in `svc_run` omogočajo klicanje oddaljenih procedur. Funkcije na najvišjem nivoju uporabljajo srednji nivo za klicanje procedur RPC. Podobno kot najvišji nivo je programiranje na srednjem nivoju omejeno s klicanjem oddaljenih procedur, za katere vemo, da obstajajo in ne potrebujejo kakšne posebne kontrole nad pretokom podatkov (enosmeren prenos). Zaradi te omejitve srednji nivo tudi ni veliko uporabljan pri distribuiranih programih, saj ni mogoče kontrolirati poteka programa v primeru napak.

**Najnižji nivo** omogoča programiranje vseh podrobnosti, ki jih distribuirani programi zahtevajo. Za lažje pisanje programov RPC na tem nivoju obstaja tudi program *rpcgen*, ki olajša pisanje rutin, ki so vezane na protokol prenosa.

## C.6 Kodiranje in dekodiranje podatkov – XDR

Knjižnica XDR je množica podprogramov v jeziku C. XDR skrbi za konverzijo internih zapisov spremenljivk v dogovorjeni zapis. Pri spremenljivkah s plavajočo vejico sovpada z zapisom po IEEE standardu in s padajočo pomembnostjo spominskih celic (*Big-Endian byte ordering*). Protokol je XDR predpisan z ARPA NIC dokumentom RFC1014, ki ustreza predstavitvenemu sloju po ISO X.409 predlogu sintakse zapisa.

Kot že omenjeno imajo računalniki različne procesorje z različno osnovno ločljivostjo, od zastarelih osem bitnih do novejših 64 bitnih procesorjev, ki pa se ne ločijo le po tem, kako velika so cela števila, ampak tudi po tem, kako so števila zapisana v spominu, ki je še vedno v osnovi urejen z byti (8 bitov). Tako ločimo med *Little-Endian* in *Big-Endian* načinom zlaganja spremenljivk večjih od enega byta. Na sliki C.2 je prikazana razlika v




Slika C.2: Primer različnega sestavljanja 32-bitnega števila

Poleg konverzije enostavnih in sestavljenih tipov je za večje količine podatkov možno uporabiti tokove (*streams*). Tokovi so po funkcionalnosti zelo podobni klasičnim vhodno/izhodnim operacijam za branje in pisanje datotek. Poleg omenjenih tokov je možno uporabiti tudi kvazi datoteke, ki pa se nahajajo v spominu računalnika. Praktična uporabnost takih tokov se pokaže pri paketnem prenosu večjega števila podatkov. Posebna vrsta tokov je tudi tok z zapisi (*Record Stream*), ki je podoben direktnim datotekam v jeziku Fortran, le da je tu velikost zapisa lahko spremenljiva. Uporabnost takega toka se kaže pri prenosu standardiziranih manjših sestavljenih podatkov. Tipična aplikacija, ki uporablja take zapise je relacijska baza podatkov.

Osnovni filtri oz. rutine za konverzijo obstajajo za interne tipe:

**xdr_char()** pretvorba nizov (stringov)

**xdr_short(), xdr_u_short()** skrajšana cela predznačena in nepredznačena števila

**xdr_int(), xdr_u_int()** cela predznačena in nepredznačena števila

**xdr_long()** podaljšana cela števila

**xdr_float()** realna števila

**xdr_void()** prazno

**xdr_enum()** naštevni tipi

Navedene rutine obstajajo tudi v predprocesorski (*macro*) obliki.

Filtri za sestavljene tipe so lahko le tisti za katere poznamo osnovne tipe iz katerih so sestavljeni. Tako je možno sestaviti spremenljivo dolžino bytov z `xdr_bytes()`, nize znakov `xdr_string`, vektorje z `xdr_vector()`, matrike z `xdr_array` in `xdr_opaque` za vektor poljubnih tipov, ki pa se ne konvertira.

Iz naštetih sestavljenih in osnovnih tipov lahko uporabnik naredi svoje izpeljane filtre, ki jih v aplikaciji potrebuje. Pri pisanju uporabniških filtrov je potrebno upoštevati omejitve, ki so posledica načina konverzije. Tako je potebno večdimenzionalne tabele in strukture razbiti na več osnovnih ali sestavljenih tipov.

## C.7 Prevajalnik protokola RPCGEN

S programom *rpcgen*, ki je priložen okolju RPC se iz posebne izvorne kode generira koda za aplikacijo strežnika in odjemalca. Izvorna koda za *rpcgen* ima torej strukturo, ki je podobna programom (*Lex[LMB92], Yacc, Bison, Equel*) z namenom konsistentnosti aplikacij strežnika in odjemalca. Kot je že bilo omenjeno, se podprogrami iz knjižnice XDR uporabljajo za kodiranje in dekodiranje podatkov, zato bi bilo neumestno, če bi oba dela celotne aplikacije uporabljala svoje rutine za konverzijo.

S programom *rpcgen* tako iz izvorne kode, ki ima običajno končnico `.x` naredimo tri izvorne datoteke `.c` in skupno datoteko `.h`. Tri izvorne datoteke so:

```
 */
union readdir_res switch (int errno) {
case 0:
        namelist list;  /* brez napake: vrni seznam direktorija */
default:
        void;           /* napaka: vrni prazno */
};
/*
 * Definicija programa z podprogramom READDIR
 */
program DIRPROG {
        version DIRVERS {
                readdir_res
                READDIR(nametype) = 1;
        } = 1;
} = 76;
```

Poleg definicije protokola, je potrebno napisati še specifične podprograme, ki dejansko izvajajo delo in vračajo rezultate. Implementacija je seveda odvisna od specifičnosti celotne aplikacije.

## C.8 Prijavljanje strežnika sistemu

Ob izdelanem programu za stežnika in odjemalca je potrebno prijaviti servis sistemu, tako da ga sistem ob zahtevi odjemalca avtomatsko požene. Tak način registracije programa ni nujen za vse strežne programe, saj lahko uporabnik brez pomoči administratorja sam požene strežni program. Najboljša rešitev za poganjanje programa je z uporabo najvišjega demona (*top level daemon*), ki skrbi za večino mrežnih servisov. To je demon *inetd*, ki poganja tudi večino RPC strežnih programov kot so *nfs*, `rstatd`, `statd`, ... S tem *super demonom* se v veliki meri zmanjša obremenjenost sistema, saj se novi demoni zbujajo le na zahtevo, *inetd* pa skrbi za to, da jih v času daljšega mirovanja odstrani iz seznama procesov.

Prevajalnik protokola *rpcgen* podpira generiranje kode za *inetd*. Tako je možno nastaviti trajanje strežnika ob neaktivnosti strežnika RPC. Ob zahtevi za končanje se bo servis RPC sam odjavil in normalno končal.

Prijavljanje novih servisov pomeni popravljanje nekaterih datotek v imeniku `/etc`. Datoteka `/etc/inetd.conf` nastavlja delovanje demona *inetd*, seznam mrežnih servisov se nahaja v `/etc/services`, medtem ko za programe tipa RPC obstaja datoteka `/etc/rpc`.

## C.9 Povzetek

V tem poglavju je prikazan distribuiran sistem ONC (*Open Network Computing*) za klicanje oddaljenih procedur RPC (*Remote Procedure Calling*). Knjižnica RPC je ena prvih

datoteke) s sistemom. Tako delovanje seveda največkrat ni zaželjeno, zato se lahko uporabi kreiranje kopije originalnega procesa, v kateri se potem izvrši ukaz `execl()`. V ta namen na sistemu UNIX obstaja funkcija `fork()`, ki kreira kopijo originalnega procesa (*child process*). Funkcija `fork()` vrne celoštevilčno vrednost, ki je nič za kopijo *(child)*, originalu pa vrne številko procesa kopije. Primer uporabe je sledeč:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void)
{
  pid_t pid;
  pid = fork();
  if ( pid )
     {
        int status;
        wait(&status);
        printf("Kopija procesa je imela stevilko %d\n", pid);
     }
  else
     printf("Pozdrav iz podprocesa\n");
  return 0;
}
```

Iz primera je razvidno, da original čaka podproces, da konča. Če ga ne bi čakal s funkcijo `wait()`, bi vsak program odbrzel v svoji smeri. Pri kreiranju podprocesov je pomembno tudi to, da vsi opisniki odprtih datotek delujejo tudi v podprocesih. V podprocesu bi lahko uporabili tudi eno izmed funkcij `exec()` in s tem pognali zunanji program. Ko je podproces končal, je original dobil status o končanju podprocesa in bi lahko glede na to ustrezno ukrepal. Za aktivno komunikacijo med procesi, pa je potrebno uporabiti dodatne funkcije, ki omogočajo različne načine komunikacije, kot so *cevi, imenovane cevi, semaforji, vrste, deljeni spomin, vtičnice, tokovi* (*pipes, FIFOs, semaphores, message queues, shared memory, sockets, streams*).

## D.2 Signali

Pošiljanje sporočil s signali je način obdelave asinhronih dogodkov. Ti signali so mišljeni kot prekinitve programa, ki dobi sporočilo in se mora odzvati na način, ki je predpisan. Sporočila programu lahko pridejo od uporabnika, ki npr. pritisne določeno tipko ali da ukaz za generiranje signala; lahko pa je to tudi sporočilo od drugega programa, ki komunicira s prvim. Signalov je več vrst, so poimenovani in imajo predpisan način, kako se program odziva na določen signal. Privzeto odzivanje programa na signal lahko spremenimo s funkcijo `signal()`. Uporabnik lahko nastavi tri načine odzivanja na signal:

s funkcijama `popen()` in `pclose()`. Slednji način je tudi enostavnejši, saj je komunikacija z uporabo navidezne datoteke uporabna tudi pri testiranju in učenju, ker lahko naredimo dva programa, ki klasično komunicirata med seboj s pomočjo ukaznega interpreterja. Po uspešnem testu je ta dva programa enostavno združiti v en program, ki uporablja IPC s cevmi. Ob pravilni zgradbi programov je potrebno `fp = stdin` zamenjati z `fp = popen("ukaz", "r")`.

Če želimo dvosmerno komunikacijo med procesoma, enostavno odpremo dve cevi, vsako v svojo smer. Interaktivno taka komunikacija ni možna brez uporabe poimenovanih cevi *(named pipe – FIFO)*. V operacijskem sistemu naredimo tak komunikacijski kanal z ukazom `mkfifo`, ki postane oporna točka dveh programov, ki na dogovorjenem mestu uporabita navidezno datoteko (FIFO) in jo uporabita za komunikacijo. FIFO je tako le vezna točka med nepovezanimi programi. Operacije na datoteki FIFO so enake kot za navadne datoteke. Podoben način komunikacije obstaja v UNIX-u med jedrom operacijskega sistema in ostalimi programi z uporabo specialnih datotek, ki se nahajajo na področju `/dev`.

## D.4 Sporočila

Sporočila IPC se pošiljajo v vrsti, za kar poskrbi jedro operacijskega sistema, ki tudi hrani sporočila in skrbi za previlen pretok sporočil med procesi. S funkcijo `msgget` se kreira nova vrsta *(message queue)* ali pa se odpre obstoječa vrsta. Pošiljanje sporočil in s tem dodajanje novih sporočil v vrsto se izvede z `msgsend`. Pri prevzemanju sporočil ni nujno, da sprejmemo sporočila tako, kot so prihajala (prvi noter – prvi ven), ampak lahko tudi izbiramo vrsto sporočil ter tako filtriramo glede na našo prioriteto. Prevzemanje je izvedeno s funkcijo `msgrcv`. Sporočila, ki se pošiljajo po kanalu v vrsti so seveda uporabniško nastavljiva in poljubne oblike. Tako ima funkcija `msgsnd` sledeče argumente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msgid, const void *ptr, size_t nbytes, int flag);
```

Razvidno je, da se med procesi lahko izmenjujejo zelo kompleksni podatki, kot so strukture ali seznami. Procesi morajo imeti le enake definicije za tipe podatkov in dogovor o številkah sporočil iz katerih je razvidno, kakšen tip podatkov je v sporočilu. Ker se celoten proces izmenjave podatkov odvija na istem operacijskem sistemu, tu ni potrebno dodatno kodirati in dekodirati, kot je to potrebno pri distribuiranih sistemih z različnimi operacijskimi sistemi. Obstaja pa omejitev pri pošiljanju sporočil. Omejitev obstaja zaradi tega, ker jedro sistema skrbi za shranjevanje in pošiljanje sporočil. Tako je velikost enega sporočila največkrat omejena na 4048 bytov. Omejeno je tudi celotno število sporočil, ki je naenkrat v bazenu sporočil. Z dejstvom, da je tako pošiljanje paketno in omejeno,

## D.7 Berkley Sockets

Eden od pomembnejšin načinov *mrežne IPC* so prav gotovo vtičnice (*sockets*), ki so se prvič pojavile v distribuciji BSD 4.3+. Vtičnica je IPC kanal skozi katerega kahko procesi berejo ali pišejo po tem, ko so vzpostavili zvezo na enem koncu kanala. Večina RPC knjižnic temelji na *socket* procedurah s tem, da skrivajo pred uporabnikom večino sistemskih in mrežnih posebnosti. Je pa za boljše razumevanje distribuiranih procesov pomembno tudi poznavanje vtičnic, saj se s tem lahko izognemo težavam pri iskanju logičnih napak v programih.
Proces priklapljanja vtičnic in komunikacije med procesi je sledeč:

1. Proces, ki bo vršil funkcijo strežnika mora kreirati vtičnico in na njej čakati na vzpostavitev zveze.
2. Odjemalec mora ugotoviti naslov strežnika in priključka na strežniku, ki ga bo uporabil.
3. Podobno kot strežnik tudi odjemalec naredi svojo vtičnico in jo uporabi za priklop na strežnik.
4. Ob pravilnem priklopu strežnik prestreže zahtevo odjemalca, jo obdela, ter se ponovno vrne k poslušanju na vtičnici.

S funkcijo `socket` se kreira vtičnica tako, da povemo območje povezovanja (*domain*) in tip povezave. Trenutno so v uporabi sledeči načini (območja) povezave:

**AF_UNIX** Interno povezovanje (UNIX)

**AF_INET** Povezovanje na Internet omrežju

**AF_ISO** Povezave kot jih predpisuje ISO

**AF_NF** Protokol Xerox Network Systems

**AF_IMPLINK** IMP povezava

Pri načinu prenosa ločimo dva bistveno različna načina transporta podatkov: paketni in tokovni prenos, ki sta vezana na tip vtičnic. Paketne vtičnice omogočajo prenos *datagramov* in se obnašajo tako kot bločne enote. Tokovne vtičnice (*streams*) pa so povezava do navideznega toka, podobega tokovom bytov v sistemu UNIX. Glede na ta dva tipa vtičnic tako ločimo med dvema različnima načinoma prenosa:

- Prenos s *trajno povezavo* uporabi vtičnico s funkcijo `socket()`. Strežnik se nato s funkcijo `bind` poveže z lokalnim naslovom in z `listen()` posluša na naslovu. Odjemalec podobno kreira vtičnico s `socket` in se poveže na strežnik s funkcijo `connect`. Ko je zveza vzpostavljena, odjemalec ali strežnik uporabita funkcijo